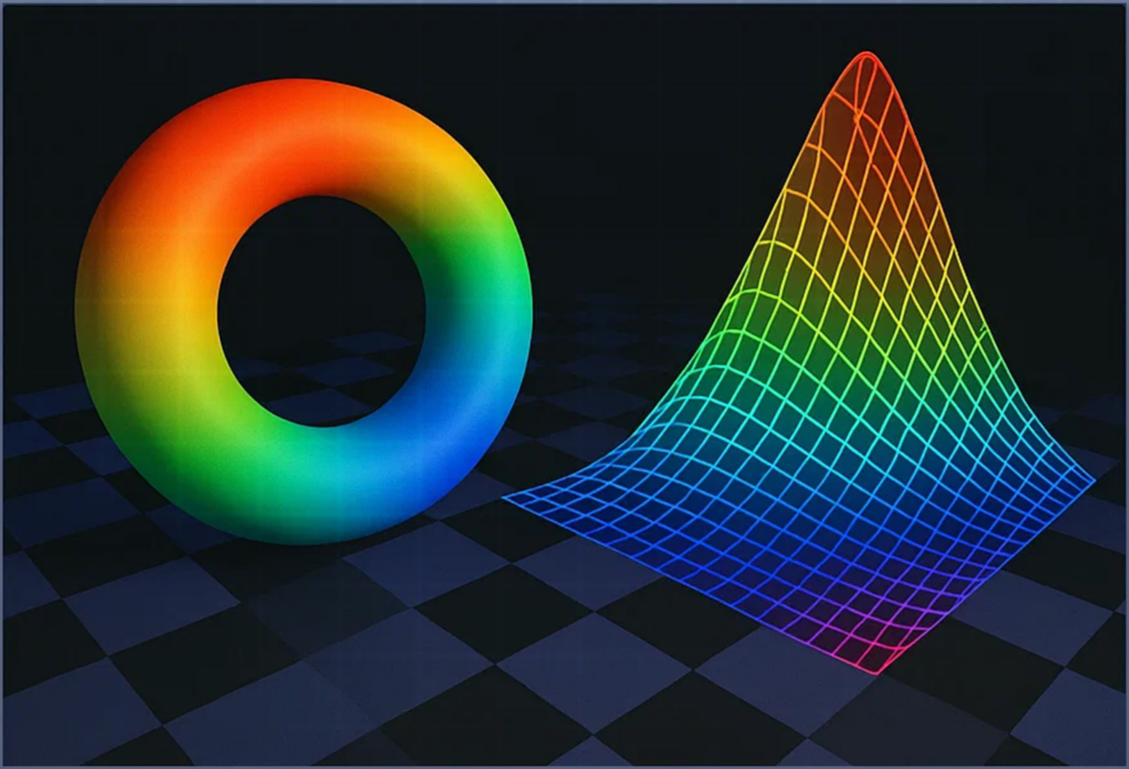
GRAPHICS PROGRAMMING

DRAFT

USING THE CPU ONLY

COMPLETE GUIDE FOR PROFESSIONALS ON LINUX AND WINDOWS



Prepared by: Ayman Alheraki

First Edition

Graphics Programming Using the CPU Only Complete Guide for Professionals on Linux and Windows

Prepared by Ayman Alheraki

simplifycpp.org

June 2025

Contents

Author's Introduction

Have you ever wondered how graphics were rendered in the early days of computing—before the invention of graphics processing units (GPUs)?
In an era where the central processor (CPU) was responsible for everything—from calculations to pixel rendering—innovative techniques emerged that laid the foundation for the advanced graphical systems we have today.
This book takes you back to those roots—not as a historical review, but as a practical, hands-on guide that teaches you how to build graphics from scratch, relying solely on the CPU, without the help of any graphical acceleration hardware.
Graphics Programming Using the CPU Only (No GPU) is both a practical and methodical book. It walks you step-by-step through building a lightweight and efficient graphics engine using only **C/C++**, without relying on heavy libraries or external toolkits.
In this book, you'll learn how to:

- Manually create and manage a **framebuffer** for direct screen drawing
- Build implementations using **SDL, GDI, X11, and FreeType**
- Draw lines, circles, images, and text with your own code
- Manually load and display **BMP** and **PNG** image files

- Apply classic algorithms like **Bresenham** and **Scanline**, and implement animation effects

This book isn't about theory alone—it's written for developers who want to understand the **true fundamentals** behind every pixel that appears on the screen.
Whether you're a:

- Systems programmer
- Embedded software engineer
- Emulator developer
- Or simply an enthusiast who wants to build a basic graphics engine

You'll find everything you need to get started.
While today's powerful GPUs have made many of these low-level techniques obsolete, I firmly believe that understanding how graphics really work under the hood is what separates a programmer who uses libraries from an engineer who knows how they're built.
This book is your opportunity to understand graphics from the inside—to take control of the pixel, and view the screen as a blank canvas waiting for your creation.
Soon, you'll begin this technical journey. Be ready to learn graphics like never before.

Stay Connected
For more discussions and valuable content about **Graphics Programming Using the CPU Only Complete Guide for Professionals on Linux and Windows**.
I invite you to follow me on **LinkedIn**:
https://linkedin.com/in/aymanalheraki
You can also visit my personal website:

https://simplifycpp.org

Ayman Alheraki

Preface

Why Graphics Programming Without a GPU Still Matters Today

In an era dominated by powerful GPUs, one might question the relevance of CPU-based graphics programming. However, software rendering remains a critical discipline with practical applications in modern computing. This section explores why mastering GPU-independent graphics techniques is still valuable.

Performance and Control in Constrained Environments

Embedded and Low-Power Systems

Many embedded systems, industrial controllers, and IoT devices lack dedicated GPUs. In these environments, efficient CPU-based rendering is essential for displaying interfaces, diagnostic data, or simple visualizations without relying on external hardware acceleration.

Real-Time and Deterministic Rendering

Certain applications—such as aviation systems, medical devices, and financial trading platforms—require deterministic performance. GPU drivers introduce unpredictable latency due to scheduling and power management. Pure software rendering ensures frame-accurate timing, critical for real-time systems.

Cross-Platform Compatibility and Portability

Legacy and Unsupported Hardware

Many older machines, thin clients, and specialized workstations (e.g., scientific or military systems) may not support modern GPU APIs. Software rendering ensures compatibility where hardware acceleration is unavailable or unreliable.

Cloud and Virtualized Environments

In virtual machines (VMs) and cloud instances, GPU passthrough is not always enabled or permitted. CPU-based rendering allows graphics applications to run in headless or virtualized environments without GPU dependencies.

Debugging, Education, and Foundational Knowledge

Understanding Rendering Fundamentals

Before optimizing for the GPU, developers must understand how rasterization, texture mapping, and blending work at the lowest level. Implementing these algorithms on the CPU reinforces core concepts that are abstracted away by modern graphics APIs (e.g., Vulkan, Direct3D 12).

Debugging GPU-Independent Issues

When graphical artifacts or performance bottlenecks occur, determining whether they stem from the GPU driver, shader code, or CPU-side logic can be difficult. A strong foundation in software rendering helps isolate and diagnose such issues.

Security and Safety-Critical Applications

Reducing Attack Surfaces

GPU drivers are complex and historically prone to security vulnerabilities. In high-security environments (e.g., military, aerospace, or industrial control systems), disabling GPU acceleration eliminates an entire class of potential exploits.

Certification and Compliance

Industries with stringent certification requirements (e.g., DO-178C for avionics or ISO 26262 for automotive) often mandate deterministic, verifiable rendering pipelines. Software rendering simplifies compliance by removing unpredictable GPU behavior.

Retro Computing and Niche Applications

Emulation and Preservation

Emulators for classic systems (e.g., DOS-era games, early 3D consoles) often rely on cycle-accurate CPU rendering to replicate original hardware behavior. Understanding software rasterization is essential for accurate emulation.

Artistic and Minimalist Rendering

Some visual styles—such as pixel art, vector animations, or procedural generation—benefit from the precision and control of CPU-based rendering. Tools like **Aseprite** and **PICO-8** demonstrate how software rendering enables unique aesthetics.

Future-Proofing and Fallback Mechanisms

Graceful Degradation

Even in GPU-accelerated applications, a software fallback ensures functionality if the GPU fails, drivers crash, or compatibility issues arise (e.g., WebGL being disabled in a browser).

Preparing for Post-GPU Architectures

Emerging architectures (e.g., RISC-V with custom accelerators, neuromorphic computing) may not follow traditional GPU paradigms. Software rendering skills ensure adaptability in a rapidly evolving hardware landscape.

Conclusion

While GPUs dominate modern graphics, CPU-based rendering remains indispensable for reliability, portability, education, and specialized use cases. This book equips developers with the techniques needed to harness the full potential of software rendering in both legacy and cutting-edge systems.

Target audience: systems programmers, emulator developers, embedded engineers, OS hobbyists

This book is designed for programmers who require precise control over rendering pipelines, work in constrained environments, or need to develop graphics solutions without GPU acceleration. The content is structured to benefit professionals and enthusiasts in the following domains:

Systems Programmers

Low-Level Performance Engineers

Developers working on operating systems, game engines, or high-performance applications will benefit from the book's focus on CPU-optimized rendering techniques. The examples demonstrate how to bypass GPU abstractions and manipulate framebuffers directly for maximum efficiency.

Compiler and Runtime Developers

Those working on just-in-time (JIT) compilation, runtime optimization, or language implementations (e.g., for domain-specific shading languages) will find value in the discussions on software rasterization and instruction-level optimizations.

Emulator Developers

Accuracy-Focused Emulation

Emulators for retro consoles (e.g., NES, PlayStation 1) and early 3D systems (e.g., Nintendo 64, Sega Saturn) often rely on cycle-accurate CPU rendering. This book covers techniques such as:

- **Scanline rasterization** for replicating CRT-era behavior.
- **Fixed-point arithmetic** for precise software-based 3D transforms.
- **Palette emulation** for authentic color reproduction.

Dynamic Recompilation and JIT Rendering

Advanced emulators use dynamic recompilation to translate GPU commands into CPU-executable code. The book explores optimization strategies for such scenarios, including:

- **Software vertex processing** for geometry pipelines.
- **Texture mapping without hardware acceleration.**

Embedded Engineers

Display Controllers and Bare-Metal Rendering

Engineers developing firmware for microcontrollers (e.g., STM32, ESP32) or custom display drivers will learn how to:

- **Implement framebuffer manipulation** on memory-mapped displays.
- **Optimize blitting routines** for low-power LCDs and OLEDs.
- **Handle DMA transfers** for efficient screen updates.

Real-Time Operating Systems (RTOS) and Safety-Critical Systems

For environments where determinism is crucial (e.g., automotive dashboards, avionics), the book covers:

- **Predictable rendering latencies** without GPU interference.
- **Memory-safe drawing algorithms** for MISRA-C/C++ compliance.

Operating System Hobbyists

Kernel Mode Graphics

Developers building custom OS kernels (e.g., for x86-64, ARM, or RISC-V) will learn how to:

- **Implement a basic VESA/VGA driver.**
- **Design a windowing system** with CPU-only compositing.
- **Manage double-buffering** without GPU assistance.

Legacy and Unusual Hardware Support

For retrocomputing enthusiasts working with:

- **Pre-GPU systems** (e.g., 486-era PCs, SGI workstations).
- **Text-mode to graphical transitions** (e.g., BIOS to protected mode).

Additional Beneficiaries

Cybersecurity Researchers

- **Exploit development** for graphics-related vulnerabilities (e.g., buffer overflows in software renderers).
- **Forensic analysis** of GPU-independent malware.

Educators and Students

- **Computer science curricula** covering fundamental rasterization algorithms.
- **Hands-on assignments** for graphics programming courses.

Prerequisites

Readers should have:

- **Proficiency in C/C++** (including pointer arithmetic and memory management).
- **Basic understanding of linear algebra** (vectors, matrices).
- **Familiarity with compilation toolchains** (e.g., GCC, Clang, MSVC).

No prior graphics API (OpenGL/Vulkan) knowledge is required—this book starts from first principles.

How This Book Serves Each Audience

Audience and Key Takeaways

Audience	**Key Takeaways**
Systems Programmers	Optimized memory access patterns, lock-free rendering techniques.
Emulator Developers	Cycle-accurate rendering, legacy GPU emulation.
Embedded Engineers	Bare-metal framebuffer management, power-efficient drawing.

Audience	**Key Takeaways**
OS Hobbyists	Kernel-mode graphics, bootloader-compatible rendering.

This structured approach ensures that all readers gain actionable insights tailored to their specialization while mastering universal software rendering concepts.

Scope – From Historical Context to Modern Software Rendering Techniques

This book provides a comprehensive examination of CPU-based graphics programming, spanning from its foundational historical roots to contemporary optimization strategies. The scope is designed to give readers both a theoretical understanding of rendering principles and practical skills for implementing efficient software renderers in modern environments.

Historical Foundations of Software Rendering

Early Graphics Systems (Pre-1990s)

- **Vector Displays and Text-Terminal Graphics**
  - Examination of character-based rendering (e.g., IBM CGA, VT100)
  - Algorithms for line drawing on plotters and early CRTs
- **Fixed-Function Graphics Hardware**
  - Analysis of architectures like the TMS9918 (TI-99/4A) and MOS 6567 (Commodore 64)
  - Tile-based and sprite-based rendering techniques

The Rise of Framebuffers (1990s–Early 2000s)

- **VGA and SVGA Programming**
  - Mode 13h (320x200, 256 colors) and planar memory models

  - Palette manipulation and double-buffering strategies

- **Early 3D Software Rendering**
  - Discussion of games like *Doom* (Bresenham-based raycasting) and *Quake* (BSP trees)
  - Fixed-point arithmetic optimizations

Core Rendering Algorithms

2D Rasterization Techniques

- **Scanline Algorithms**
  - Bresenham's line and circle drawing
  - Polygon filling with edge tables
- **Alpha Blending and Compositing**
  - Porter-Duff operations in software
  - Optimized approaches for ARM NEON and x86 SIMD

3D Pipeline Implementation

- **Geometry Processing**
  - Model-view-projection transforms without floating-point units
  - Backface culling and frustum clipping
- **Rasterization and Shading**
  - Depth buffering (Z-buffer) alternatives
  - Flat, Gouraud, and Phong shading comparisons

Modern Optimization Strategies

Data-Oriented Design for Rendering

- **Structure of Arrays (SoA) vs. Array of Structures (AoS)**
  - Cache-efficient vertex processing
- **Multithreaded Software Rendering**
  - Tile-based parallelism (divide-and-conquer approaches)
  - Lock-free synchronization techniques

SIMD and Parallel Processing

- **x86 (AVX2, AVX-512) and ARM (NEON, SVE) Optimization**
  - Vectorized triangle setup and interpolation
  - Branchless pixel blending
- **GPU-Like Approaches on CPU**
  - Software ray packet tracing
  - Compute shader emulation via C++17 parallel algorithms

Platform-Specific Implementations

Cross-Platform Frameworks

- **SDL2 and Software Surfaces**
  - Pixel buffer manipulation strategies

  - Custom blitters for performance-critical paths

- **Headless Rendering**

  - Offscreen buffer generation for cloud applications

Bare-Metal and Embedded Rendering

- **Memory-Mapped Displays (e.g., Raspberry Pi, ESP32)**

  - DMA-accelerated updates
  - Avoiding tearing without GPU sync

- **Real-Time OS Constraints**

  - Deterministic frame pacing in FreeRTOS and Zephyr

Contemporary Use Cases

Emulation and Legacy Systems

- **Cycle-Accurate Video Synthesis**

  - Replicating CRT phosphor decay in software
  - NTSC/PAL artifact simulation

- **FPGA-Assisted Rendering**

  - Hybrid CPU-FPGA pipelines (e.g., MiSTer)

Security and Reliability Applications

- **Kernel Mode Graphics**
  - Early-boot splash screens (e.g., Linux DRM/KMS)
  - Secure UI rendering for hypervisors
- **Failsafe Rendering**
  - Fallback paths when GPU drivers fail

Limitations and Boundary Conditions

The book explicitly does not cover:

- **GPU API usage (OpenGL, Vulkan, Direct3D)**
- **Hardware-accelerated post-processing (e.g., DLSS, FSR)**
- **Physics engine integration**

Progression of Topics

1. **Foundations** $\rightarrow$ 2. **Algorithm Deep Dives** $\rightarrow$ 3. **Optimizations** $\rightarrow$ 4. **Platform Integration**
   Each chapter builds on prior concepts while providing standalone reference value.

This scope ensures readers emerge with both the historical context to understand why techniques evolved and the modern skills to implement them in today's CPU-centric environments.

Tools used: C/C++, Assembly (optional), SDL, Direct2D (software fallback), X11, GDI, etc.

This book employs a carefully selected set of programming languages, libraries, and APIs to demonstrate CPU-based graphics programming across multiple platforms. The tools were chosen for their performance characteristics, cross-platform availability, and relevance in modern software rendering pipelines (post-2019).

Core Programming Languages

C (C17 Standard) and C++ (C++20 Standard)

- **Rationale**: Provides low-level memory control while allowing modern metaprogramming techniques
- **Key Usage**:
  - C for performance-critical rasterization kernels
  - C++ for template-based math libraries and pipeline organization
- **Compiler Support**:
  - GCC 11+, Clang 14+, MSVC 2022 (17.0+)
  - `-O3`, `-march=native`, and LTO enabled where applicable

Assembly Language (Optional x86-64/ARMv8)

- **Context-Specific Integration**:
  - Hand-written SIMD (AVX2/NEON) for hot loops

  - Inline assembly via compiler intrinsics (`<immintrin.h>`, `<arm_neon.h>`)

- **Safety Considerations**:

  - Isolated in benchmarked modules with fallback C paths

Primary Graphics Libraries

SDL2 (Version 2.24.0+)

- **Core Responsibilities**:

  - Cross-platform window/input management
  - Software surface manipulation via `SDL_Surface`

- **Optimization Features**:

  - Blitting acceleration via `SDL_BlitScaled`
  - Custom pixel format conversion

Direct2D (Windows Software Fallback Mode)

- **Configuration**:

  - `D2D1_FACTORY_TYPE_SINGLE_THREADED`
  - Software rasterizer forced via `D2D1_RENDER_TARGET_PROPERTIES`

- **Usage Context**:

  - High-quality vector rendering fallback
  - Comparative benchmarking against custom algorithms

Platform-Specific Rendering APIs

X11/Xlib (X.Org Server 1.20+)

- **Low-Level Access Patterns**:
  - Direct `XPutImage` to root window
  - Shared memory extensions (MIT-SHM)
- **Modern Considerations**:
  - Compatibility layers for Wayland environments

Windows GDI (Updated for Windows 10/11)

- **Contemporary Usage**:
  - `DIBSection` for direct framebuffer access
  - `AlphaBlend` with software fallback paths
- **Performance Caveats**:
  - Comparative analysis against Direct2D software mode

Supporting Libraries and Utilities

Image I/O

- **stb Libraries (stb_image 2.27+, stb_image_write)**
  - Lightweight PNG/BMP loading

- **libpng 1.6.40+ with ARM64 optimizations**
  - Cross-platform SIMD-accelerated decoding

Math and Geometry

- **GLM (OpenGL Mathematics 0.9.9+)**
  - Header-only vector/matrix operations
  - Modified to exclude GPU dependencies
- **Eigen 3.4+ (Selective Components)**
  - Fixed-point arithmetic specializations

Build and Analysis Toolchain

Build Systems

- **CMake 3.20+ (Multi-Platform Support)**
  - Custom toolchain files for embedded targets
- **Meson 0.63+ (Alternative Configuration)**
  - Optimized for cross-compilation scenarios

Performance Analysis

- **Linux**: `perf` with LBR (Last Branch Record)
- **Windows**: ETW tracing for CPU pipeline analysis
- **Universal**: Custom instrumentation via `std::chrono`

Platform Support Matrix

Table 0-2: Cross-Platform Support of Graphics Tools

Tool	Windows 10/11	Linux (5.15+ Kernel)	macOS 12+	Embedded (RTOS)
SDL2	Full	Full	Full	Partial (Framebuffer)
Direct2D SW	Full	N/A	N/A	N/A
X11	WSL2	Native	XQuartz	N/A
ARM SIMD	ARM64 Windows	AArch64 Linux	Apple Silicon	Cortex-M7/M55

Version Control and Reproducibility

- All examples tested with:
  - Git 2.35+ for source control
  - Docker/Podman for dependency isolation
  - Windows Terminal/WSLg for Linux-on-Windows testing

Why This Toolset?

1. **Performance Portability**: Maintains consistent behavior across x86, ARM, and RISC-V

2. **Historical Continuity**: Bridges legacy techniques with modern compiler capabilities
3. **Debuggability**: Avoids GPU driver black-box behavior

This toolchain enables readers to implement everything from vintage 8-bit style renderers to modern multithreaded software pipelines while maintaining strict CPU-only execution.

Chapter 1

The History of Software Rendering

1.1 Early Graphics on the CPU: DOS, VGA, Mode 13h

1.1.1 Introduction

Before the advent of dedicated graphics processing units (GPUs), all graphics rendering on personal computers was performed entirely by the central processing unit (CPU). This era, spanning the late 1970s through the early 1990s, was characterized by direct manipulation of video memory and hardware registers under low-level operating systems such as DOS.

Understanding the foundational graphics modes and techniques of this period provides critical insights into the challenges and constraints that shaped early software rendering, as well as the techniques that modern graphics programming still draws upon. This section explores the development of early graphics programming on IBM PC-compatible machines, focusing on VGA (Video Graphics Array) hardware, the widely-used Mode

13h graphics mode, and programming paradigms under DOS.

1.1.2 The DOS Environment and Its Impact on Graphics Programming

In the 1980s and early 1990s, Microsoft's Disk Operating System (DOS) was the predominant operating system on IBM PC compatibles. DOS was a simple, single-tasking operating system without built-in graphical abstractions or drivers. It provided direct access to hardware and memory, which was both a limitation and an advantage for graphics programmers.

- **Direct hardware access:** Programmers could write directly to video memory, access hardware ports, and control graphics adapters without intermediary drivers.
- **Memory model constraints:** The real mode CPU architecture limited addressable memory to 1MB, with only 640KB available for applications.
- **No multitasking:** Single-task operation meant no preemption, simplifying timing but also limiting complex interactions.

These characteristics encouraged programmers to work at the hardware level, manipulating graphics modes and pixel data by writing directly to video memory.

1.1.3 Early Graphics Hardware: From CGA and EGA to VGA

The first widely adopted graphics adapters for IBM PCs included the Color Graphics Adapter (CGA) and the Enhanced Graphics Adapter (EGA). These provided a limited palette and resolution, but laid the groundwork for VGA, which became the standard starting in 1987.

- **CGA (1981):** Supported 4 colors at 320×200 or monochrome at 640×200 pixels. Graphics modes were limited and color palettes were fixed.
- **EGA (1984):** Improved resolution to 640×350 with 16 colors out of a palette of 64.
- **VGA (1987):** Supported multiple resolutions, including 640×480 at 16 colors and 320×200 at 256 colors, vastly expanding color depth and programmability.

VGA introduced more flexible programming options, multiple video modes, and a larger, dedicated video memory space (256 KB), enabling more sophisticated software rendering techniques.

1.1.4 VGA Video Memory Architecture

VGA video memory was mapped starting at segment address `0×A000` in real mode. Depending on the graphics mode, the layout and interpretation of video memory varied.

- **Planar mode:** Used for 16-color modes, where pixel data was distributed across four planes.
- **Packed pixel mode:** Used in 256-color modes such as Mode 13h, where each byte corresponded directly to a single pixel.

Programmers needed to understand how to write to this memory efficiently, often dealing with bank switching and plane masking.

1.1.5 Mode 13h: The Popular 256-Color VGA Mode

Mode 13h is the BIOS interrupt code for setting a standard VGA graphics mode of 320×200 pixels with 256 colors. It became the de facto mode for early PC games and graphics demos due to its simplicity and rich color palette.

1. **Characteristics of Mode 13h**

   - **Resolution:** 320×200 pixels.
   - **Color depth:** 8 bits per pixel, allowing 256 simultaneous colors.
   - **Linear frame buffer:** Each byte in video memory corresponds to one pixel, arranged linearly.
   - **Video memory:** Located at physical address segment `0×A000`, covering 64 KB, enough to store the entire frame buffer.

2. **Advantages**

   - **Simplified pixel addressing:** Since video memory is linear, addressing pixels is straightforward.
   - **Rich color palette:** 256 colors out of 262,144 possible RGB colors (18-bit palette), programmable via VGA DAC registers.
   - **Fast access:** No need for plane switching or bit masking, unlike earlier modes.

3. **Limitations**

   - **Low resolution:** By modern standards, 320×200 is a small canvas.
   - **Limited palette:** Although 256 colors is large for the time, it still limits photorealistic imagery.
   - **No hardware acceleration:** All rendering was done in software, placing heavy demands on CPU performance.

1.1.6 Programming Mode 13h: Setting the Mode and Drawing Pixels

Programming in Mode 13h under DOS involved three basic steps:

1. **Setting Mode 13h**

Using BIOS interrupt 10h with function 0:

```
mov ah, 0x00        ; BIOS set video mode function
mov al, 0x13        ; Mode 13h (320×200, 256 colors)
int 0x10            ; Call BIOS video interrupt
```

This call switches the graphics adapter to Mode 13h.

1. **Accessing Video Memory**

The frame buffer begins at segment `0xA000`. Pixels can be drawn by writing directly to the memory at `0xA000:0x0000`.
For example, in C:

```
unsigned char far* video_memory = (unsigned char far*)0xA0000000L;

void put_pixel(int x, int y, unsigned char color) {
    video_memory[y * 320 + x] = color;
}
```

1. **Drawing Primitives**

Because the hardware offers no built-in drawing functions, all graphics primitives—pixels, lines, circles, polygons—must be implemented in software. Classic algorithms such as Bresenham's line and circle algorithms were often employed.

1.1.7 Software Rendering Techniques Under Mode 13h

Rendering in Mode 13h was a purely software affair, with the CPU performing all rasterization and pixel manipulation.

- **Direct pixel plotting:** Setting individual pixels by writing to video memory.
- **Double buffering:** To avoid flicker, offscreen buffers in conventional RAM were used to prepare frames, then copied en masse to video memory.
- **Palette manipulation:** Programmers could define and alter the 256-color palette dynamically, enabling effects like palette cycling and fading.
- **Clipping and coordinate transforms:** Required manual implementation to handle drawing within screen bounds.

1.1.8 Third-Party Tools and Modern Usage

Although Mode 13h programming is considered legacy, understanding it remains vital for grasping the origins of graphics programming. Modern tools and operating systems have largely abstracted graphics programming behind APIs and hardware acceleration. However:

- **Emulators and retro game development** often use Mode 13h as a learning platform.
- **Cross-platform graphics libraries** sometimes expose similar framebuffer concepts.
- Tools like **DOSBox** (updated frequently after 2019) allow developers to run and debug Mode 13h programs on modern hardware.

- Low-level programming courses and tutorials continue to use Mode 13h for demonstrating fundamental graphics programming principles.

1.1.9 Summary

The era of graphics programming on the CPU without a GPU, typified by DOS, VGA, and Mode 13h, laid the foundation for modern software rendering techniques. Mode 13h's linear framebuffer and 256-color palette simplified pixel-level manipulation, but all rendering was a CPU responsibility, demanding efficient algorithms and clever programming to achieve acceptable performance and visual quality.
This foundational knowledge continues to be essential for understanding graphics hardware evolution and the principles behind today's software rasterizers and GPU shaders.

1.2 The Demo Scene and Artistic Programming on the 386/486 Era

1.2.1 Introduction

The late 1980s and early 1990s, characterized by the dominance of Intel 386 and 486 processors, marked a golden era for software rendering driven by a vibrant subculture known as the **demo scene**. This movement, fueled by the combination of advancing CPU capabilities and evolving PC graphics hardware, was a crucible of innovation in artistic programming—pushing the limits of what was achievable purely through software on CPUs without GPU acceleration.

This section explores the demo scene's origins, technical achievements, and its lasting impact on software rendering, emphasizing the programming techniques and artistic creativity of the era.

1.2.2 The Emergence of the Demo Scene

The demo scene originated in the 1980s as a collective of programmers, musicians, and digital artists who created audiovisual programs known as "demos." Initially linked to the cracker scene—groups who broke software protections and distributed unauthorized copies—the demo scene evolved into a distinct artistic and technical culture focused on pushing hardware limits.

With the advent of the Intel 386 and later 486 processors, the PC platform matured into a powerful canvas for complex real-time graphics and sound, all driven by highly optimized software routines.

1.2.3 Hardware Context: 386 and 486 Processors

- **Intel 386 (1985):** Introduced 32-bit processing and protected mode, enabling more advanced memory management and performance.
- **Intel 486 (1989):** Integrated floating-point units (in many models), pipelining, and improved instruction throughput, significantly enhancing CPU-intensive tasks such as real-time graphics rendering.
- **Memory:** Larger RAM capacities (commonly 4MB to 16MB) allowed bigger buffers and more complex scenes.
- **Graphics hardware:** VGA remained dominant, but 256-color modes and increased memory bandwidth allowed higher fidelity visuals.
- **Sound cards:** Sound Blaster and Gravis Ultrasound cards enabled synchronized audio-visual presentations.

This hardware combination gave demo programmers unprecedented resources for software-only rendering.

1.2.4 Artistic Programming and Technical Innovation

The demo scene was characterized by its fusion of technical mastery and artistic expression. Programmers crafted mesmerizing visual effects, complex animations, and synchronized soundtracks, all executed in real time on general-purpose CPUs.
Key characteristics and innovations include:

1. **Procedural Graphics and Real-Time Effects**

   Instead of relying on static images, demo programmers wrote algorithms that generated visuals dynamically:

- **Raster effects:** Smooth scrolling, plasma, and color cycling created the illusion of fluid movement.
- **3D wireframe and solid rendering:** Basic 3D models were rendered with flat shading, often using fixed-point arithmetic for speed.
- **Ray tracing and ray casting:** Early attempts at realistic lighting through software-based ray tracing, albeit at low resolutions or frame rates.
- **Texture mapping:** Techniques to apply textures on polygons, a cutting-edge challenge due to limited CPU power.

These effects demonstrated advanced mathematics, including trigonometry and vector algebra, optimized for speed.

2. **Optimized Assembly and Inline Machine Code**

   To achieve high performance, many demos were written predominantly in hand-optimized assembly language, often with inline machine code in higher-level languages like C.

   - **Cycle-counted loops:** Programmers counted CPU cycles to maximize instruction throughput and maintain timing precision.
   - **Self-modifying code:** To optimize loops and branch prediction.
   - **Use of CPU instructions:** Exploiting 386/486-specific instructions such as `CMPSB`, `REP MOVSB`, and bit manipulation for fast memory operations.

   Such optimization was necessary for real-time rendering at acceptable frame rates.

1.2.5 Classic Demo Effects and Their Implementation

The 386/486 demo scene introduced iconic graphical effects, some of which remain foundational teaching examples in graphics programming:

1. **Plasma Effects**

   A colorful, smoothly animated texture generated by combining sinusoidal functions of pixel coordinates and time, plasma effects create continuously changing gradient patterns.

   - **Implementation:** Using precomputed sine tables and fast arithmetic, each pixel color is calculated via a formula combining wave functions.
   - **Optimization:** Lookup tables and fixed-point math to minimize costly floating-point operations.

2. **Starfields and Tunnel Effects**

   Simulating depth and motion through a field of stars or a tunnel-like environment by projecting points or texture-mapped slices onto the screen.

   - **Starfield:** Calculated position updates for thousands of stars moving toward the viewer, with simple scaling and translation.
   - **Tunnel:** Texture slices warped according to a distance function and drawn line-by-line to simulate movement through a cylindrical space.

3. **Rotating 3D Objects**

   Wireframe cubes, spheres, or teapots rotated in real time using matrix transformations and projected onto 2D screen coordinates.

   - **Mathematics:** Basic linear algebra with rotation matrices, perspective projection.
   - **Fixed-point arithmetic:** Replacing floating-point for speed on CPUs without fast FPUs.
   - **Double buffering:** To avoid flickering during rapid redraws.

1.2.6 Tools and Development Environments

Programming demos in this era involved a combination of low-level tools and creative workflows:

- **Assemblers:** MASM, TASM, and NASM were widely used for writing and assembling performance-critical code.
- **Debuggers:** Tools like SoftICE allowed real-time debugging of protected-mode code.
- **Compilers:** Borland C++ and Microsoft C offered C language environments, often supplemented with inline assembly.
- **Sound trackers:** Programs like Scream Tracker and FastTracker enabled composers to create module-based music synchronized with visuals.

After 2019, enthusiasts and retro developers continue to use modern cross-platform assemblers and debuggers, and emulators like DOSBox provide stable environments to develop and run demos.

1.2.7 The Cultural and Educational Impact

The demo scene was more than technical showmanship; it fostered:

- **A collaborative culture:** Sharing source code, techniques, and demos at parties and via bulletin board systems (BBS).
- **Educational value:** Many programmers learned graphics fundamentals, assembly programming, and optimization through demo coding.
- **Legacy:** Concepts from the demo scene influenced game development, multimedia programming, and modern GPU shader programming paradigms.

1.2.8 Summary

The 386/486 era's demo scene was a watershed moment in the history of software rendering. It proved that with ingenuity, efficient programming, and artistic vision, general-purpose CPUs could generate compelling real-time graphics without dedicated GPU hardware. The programming techniques and effects pioneered in this period remain relevant to understanding low-level graphics programming, optimization strategies, and the origins of computer-generated art.

1.3 Pre-GPU Techniques: Raycasting, Framebuffer Tricks

1.3.1 Introduction

Before the advent of graphics processing units (GPUs), all rendering calculations were performed by the CPU, requiring innovative algorithms and clever memory manipulations to create visually rich graphics in real time. Among the most significant techniques developed in this pre-GPU era were **raycasting** and various **framebuffer tricks**.

These methods allowed early software to simulate 3D environments and produce complex visual effects on limited hardware by optimizing CPU workload and leveraging the architecture of video memory. This section explores the principles, implementation strategies, and impact of these pioneering techniques.

1.3.2 Raycasting: A Breakthrough in Software 3D Rendering

1. **Overview of Raycasting**

   Raycasting is a technique used to render a 3D perspective by projecting rays from the viewer's position into a 2D map or environment and determining the distance to the nearest obstacle along each ray. The length of each ray is used to calculate the height of vertical slices on the screen, creating the illusion of a 3D space.

   Unlike full ray tracing or polygonal rendering, raycasting simplifies calculations by restricting the world to grid-aligned walls or obstacles, enabling real-time performance on CPUs of the 1980s and early 1990s.

2. **Historical Context and Applications**

Raycasting became popularized in the early 1990s by games such as *Wolfenstein 3D* (1992), which delivered immersive pseudo-3D environments on relatively modest hardware.

- **Hardware constraints:** CPUs like Intel 386 and 486 had limited floating-point performance; raycasting used integer or fixed-point arithmetic to maintain speed.
- **Memory layout:** The world was represented as a 2D array (map), where each cell indicated the presence or absence of walls.
- **Visual impact:** The technique allowed dynamic first-person views with texture mapping and simple lighting effects.

3. **Raycasting Algorithm Fundamentals**

   The core algorithm proceeds as follows for each vertical slice (column) on the screen:

   (a) **Calculate ray direction:** Based on the player's position and viewing angle.

   (b) **Step through the map grid:** Using the Digital Differential Analyzer (DDA) algorithm, the ray advances cell-by-cell until it hits a wall.

   (c) **Calculate distance:** Measure the distance from the player to the wall intersection.

   (d) **Compute slice height:** Use the distance to scale the height of a vertical strip representing the wall.

   (e) **Draw slice:** Render the vertical line with texture or color shading corresponding to the wall surface.

4. **Optimizations and Enhancements**

- **Fixed-point math:** Employed to reduce computational overhead of floating-point operations.
- **Lookup tables:** For trigonometric functions (sine, cosine, inverse distance) to speed up calculations.
- **Texture mapping:** Applied vertical texture strips to wall slices to increase realism.
- **Shading:** Distance-based darkening to simulate depth and lighting.
- **Floor and ceiling rendering:** Extended later using raycasting variants or floor casting.

1.3.3 Framebuffer Tricks: Manipulating Video Memory for Visual Effects

1. **Overview**

   Framebuffer tricks are a set of techniques that exploit the layout and behavior of video memory to achieve visual effects beyond simple pixel plotting. These methods leverage CPU control over the framebuffer to produce smooth animations, palette manipulations, and graphical illusions, often without expensive computation.

2. **Types of Framebuffer Tricks**

   - **Page flipping and double buffering:** Using multiple buffers to prevent flicker by drawing frames offscreen before displaying.
   - **Palette cycling:** Changing color palette entries in video memory dynamically to simulate animation (e.g., flowing water or fire effects) without modifying pixel data.

- **Raster interrupts and split screens:** Synchronizing CPU updates with display scanlines to alter palette or video memory mid-frame, allowing multiple effects on different screen areas.
- **Line scrolling and raster bars:** Horizontal or vertical scrolling effects by manipulating start addresses or video memory offsets on a per-line basis.
- **Bit blitting and block transfers:** Copying or moving blocks of pixels efficiently in video memory to create sprite animations or screen updates.

3. **Implementation in DOS and VGA Modes**

   In VGA Mode 13h, where video memory is a linear 64 KB buffer, framebuffer tricks included:

   - **Direct memory copies:** Using CPU instructions such as `REP MOVSB` or optimized assembly loops to move large pixel blocks.
   - **Manipulating VGA palette registers:** The VGA DAC (Digital-to-Analog Converter) registers were programmed to change palette colors in real time.
   - **Split screen via VGA registers:** Programmers modified the CRTC (Cathode Ray Tube Controller) registers to change the start address of video memory mid-screen, creating multiple viewports or scrolling sections.

4. **Examples of Popular Effects**

   - **Water and lava flows:** Achieved by cycling palette entries associated with blue or red hues, creating animated liquid surfaces.
   - **Shimmering light:** Quickly shifting palette colors to simulate flickering or glowing.

- **Parallax scrolling:** Multiple layers of background images moved at different speeds by combining framebuffer shifts and palette tricks.
- **Sprite animation:** Using block transfers for rapid movement of character or object bitmaps.

1.3.4 The Role of CPU in Framebuffer Manipulation

CPU-centric framebuffer tricks required careful timing and optimization to maintain smooth visuals:

- **Memory bandwidth:** The CPU had to manage framebuffer access alongside program logic, balancing speed and complexity.
- **Cycle-accurate coding:** Tight assembly routines ensured frame updates synchronized with vertical retrace intervals.
- **Interrupt handling:** Some effects depended on timer or raster interrupts for precise updates.

1.3.5 Modern Perspective and Legacy

Although hardware accelerated rendering via GPUs has largely replaced these techniques, pre-GPU software rendering and framebuffer tricks remain instructive:

- **Educational value:** Teaching fundamentals of pixel manipulation, memory layouts, and low-level graphics programming.
- **Retro development:** Many retro games and demos still use or emulate these techniques.
- **Embedded systems:** Some low-power devices and microcontrollers without GPUs still rely on software rendering.

Tools updated, such as modern assemblers and emulators like DOSBox, continue to support development and study of these methods on current hardware.

1.3.6 Summary

Pre-GPU techniques like raycasting and framebuffer tricks epitomize the ingenuity of early graphics programmers. By leveraging CPU computational power and deep knowledge of video memory architectures, they produced immersive visuals and innovative effects within the severe constraints of their hardware. Understanding these methods enriches comprehension of software rendering's roots and provides timeless lessons in optimization and low-level programming.

1.4 Revival in Modern Times (e.g., Software Renderers in Emulators, Retro Games)

1.4.1 Introduction

Although dedicated graphics hardware (GPUs) now dominates visual computing, software rendering has experienced a notable revival in modern times. This resurgence is fueled by the preservation and emulation of legacy platforms, the rise of retro gaming culture, and specific technical or artistic choices that favor software-based rendering. This section explores how software rendering persists today, its role in emulators and retro-style games, and the modern tools and contexts supporting its development.

1.4.2 Software Rendering in Emulators

1. **The Need for Software Rendering in Emulation**

   Emulators replicate the behavior of older or different hardware platforms on modern systems, often requiring faithful reproduction of graphics pipelines originally designed without GPUs or with distinct hardware accelerators.

   - **Hardware discrepancies:** Target platforms like early consoles (NES, SNES, PlayStation 1) or classic PCs used specialized graphics chips or unique rendering pipelines.
   - **Accuracy vs. performance:** Emulators balance between fast GPU-accelerated rendering and cycle-accurate software renderers that prioritize correctness and compatibility.

2. **Software Renderers as Reference Implementations**

   Software rendering engines in emulators provide:

- **Precise hardware emulation:** By replicating exact pixel-by-pixel behavior, blending modes, and timing.
- **Platform independence:** Functioning regardless of the host system's GPU capabilities.
- **Debugging and development:** Allowing emulator developers to verify graphics output without relying on hardware-specific features.

3. **Examples of Modern Emulators Using Software Rendering**

   - **DOSBox:** Emulates DOS environments with software rendering for VGA, EGA, and CGA graphics modes, providing pixel-accurate output on modern operating systems.
   - **PCSX-Reloaded and DuckStation:** PlayStation 1 emulators that offer software renderers to reproduce the console's graphics pipeline accurately.
   - **MAME (Multiple Arcade Machine Emulator):** Uses software rendering to simulate various arcade machine graphics chips faithfully.

4. **Modern Tools and Techniques**

   Since 2019, emulator development increasingly utilizes:

   - **Cross-platform C++ and Rust:** For performance and safety in software rendering code.
   - **SIMD instructions (SSE, AVX):** To accelerate pixel operations and blending.
   - **Multi-threading:** Distributing rendering workload across CPU cores.
   - **Open-source frameworks:** Like SDL2 for cross-platform framebuffer handling and input.

1.4.3 Software Rendering in Retro and Indie Games

1. **Artistic and Practical Motivations**

   Modern developers often choose software rendering for retro or stylized games to:

   - **Achieve authentic visual aesthetics:** Mimicking pixel art, limited palettes, and scanline effects.
   - **Simplify cross-platform deployment:** Ensuring consistent rendering across devices without reliance on diverse GPU drivers.
   - **Reduce dependencies:** Avoiding complex GPU APIs or compatibility issues.
   - **Support niche hardware:** Targeting embedded devices, handheld consoles, or low-power systems lacking GPUs.

2. **Examples of Contemporary Retro-Style Games Using Software Rendering**

   - **Doom (Classic engine ports):** Modern reimplementations or source ports often retain software rendering modes for authenticity.
   - **Celeste Classic:** Uses pixel-based software rendering to maintain sharp retro visuals.
   - **Projects using SDL or Allegro:** Popular libraries offering software rendering modes for 2D games with retro aesthetics.

3. **Technical Approaches in Modern Software Rendering**

   - **Hardware-accelerated blitting with software pixel generation:** Hybrid approaches where pixel data is computed on CPU but displayed using GPU-friendly buffers.

  - **Shading and lighting in software:** Implementing per-pixel lighting or shadows on the CPU for artistic control.
  - **Fixed-point math and lookup tables:** To optimize performance on varied modern CPU architectures.
  - **Modern compiler optimizations:** Utilizing GCC, Clang, and MSVC optimizations and profile-guided optimization (PGO) for speed.

1.4.4 Educational and Hobbyist Uses

Software rendering has regained popularity as a teaching tool and hobbyist pursuit:

- **Educational projects:** University courses and tutorials teach core graphics principles using CPU-only rendering before introducing GPU concepts.
- **Open-source projects:** Numerous repositories since 2019 demonstrate CPU rendering basics, raycasters, and small engines for learning purposes.
- **Creative coding communities:** Platforms like GitHub and forums foster sharing of software renderer projects, often in modern languages such as Rust, Go, or modern C++17/20.

1.4.5 Advantages and Limitations in Modern Context

1. **Advantages**

   - **Full control over pixel data:** Enables fine-grained manipulation for custom effects and debugging.
   - **Hardware independence:** Runs on virtually any CPU-equipped device.
   - **Deterministic behavior:** Easier to reproduce and debug compared to GPU pipeline variability.

- **Legacy and preservation:** Crucial for maintaining historical computing and gaming artifacts.

2. **Limitations**

- **Performance constraints:** CPU-bound rendering is orders of magnitude slower than GPU acceleration for complex 3D scenes.
- **Limited scalability:** Difficult to leverage massive parallelism inherent in modern GPUs.
- **Energy efficiency:** Higher CPU usage can increase power consumption on portable devices.

1.4.6 Summary

The revival of software rendering in modern times underscores its enduring relevance and versatility. Whether powering emulators that preserve computing history, enabling indie developers to craft retro aesthetics, or serving as an educational foundation for graphics programming, CPU-based rendering remains a vibrant and active field. Contemporary development tools and languages continue to evolve, making software rendering accessible, efficient, and creatively rewarding decades after its initial prominence.

Chapter 2

Setting Up Your Environment

2.1 Required Tools for Windows: MinGW, MSVC, CMake, SDL2, GDI, and Win32 API

2.1.1 Introduction

Setting up an effective development environment on Windows is the foundational step for any software rendering project that relies solely on CPU processing. This section provides an in-depth overview of the essential tools and libraries commonly used in Windows-based CPU graphics programming, including compilers, build systems, and graphics APIs. The focus is on components that enable direct pixel manipulation, window management, and integration with the operating system's graphical infrastructure, reflecting modern best practices and technologies as of 2019 and beyond.

2.1.2 Compilers: MinGW and MSVC

1. **MinGW (Minimalist GNU for Windows)**

   - **Overview:**
     MinGW provides a port of the GNU Compiler Collection (GCC) for Windows. It offers a free, open-source environment capable of producing native Windows binaries without relying on third-party runtime libraries.
   - **Advantages:**
     - Compatibility with GCC-based build scripts and tools.
     - Lightweight and suitable for cross-platform projects with shared codebases.
     - Easy to install and integrates well with common build systems like CMake.
     - Supports C and C++ standards widely used in modern CPU-based rendering.
   - **Use in Graphics Programming:**
     MinGW is commonly used in projects where portability and open-source toolchains are preferred. It enables compiling SDL2 and other libraries efficiently and is favored in open-source and academic environments.

2. **MSVC (Microsoft Visual C++)**

   - **Overview:**
     MSVC is Microsoft's proprietary C and C++ compiler, integrated into Visual Studio, the flagship IDE on Windows.
   - **Advantages:**

  - Deep integration with Windows OS and Microsoft SDKs.
  - Excellent debugging and profiling tools tailored for Windows development.
  - Optimized code generation for Intel and AMD architectures common in Windows PCs.
  - Strong support for modern C++ standards, including C++17 and C++20.
  - Official support for Windows SDKs and seamless compatibility with Win32 API and DirectX (even though this book focuses on CPU-only rendering).

- **Use in Graphics Programming:**
  MSVC is the dominant choice for professional Windows graphics applications, particularly when integration with low-level Windows APIs (Win32, GDI) is required. It supports complex projects with advanced build and deployment configurations.

2.1.3 Build System: CMake

- **Overview:**
  CMake is a cross-platform, open-source build system generator widely adopted in C and C++ projects.

- **Why Use CMake:**

  - Simplifies the build process across different compilers (MinGW, MSVC) and environments.
  - Provides portability, enabling easy switching between Windows, Linux, and macOS development.

  - Generates native build files such as Visual Studio projects or Makefiles.
  - Supports dependency management and integration of third-party libraries like SDL2.
  - Facilitates modular and scalable project structures.

- **Practical Notes:**
  For Windows-based CPU rendering projects, CMake scripts can detect the compiler, set appropriate flags for optimization, define macros for conditional compilation, and locate graphical dependencies. It is highly recommended to use CMake for managing medium to large projects in this domain.

2.1.4 Graphics Libraries and APIs

1. **SDL2 (Simple DirectMedia Layer 2)**

   - **Description:**
     SDL2 is a popular, open-source multimedia library that abstracts access to graphics, sound, and input devices. It supports multiple platforms, including Windows.
   - **Relevance to CPU Rendering:**
     - Provides a window and surface abstraction that can be directly manipulated at the pixel level, ideal for software rendering.
     - Supports software surfaces that reside in system memory, which can be accessed and modified without GPU acceleration.
     - Offers event handling for keyboard, mouse, and joystick inputs.
     - Simplifies timing and frame rate control essential in animation loops.
   - **Advantages:**

  - Cross-platform compatibility, enabling the same rendering codebase on Windows and Linux.
  - Easy integration with CMake and common compilers.
  - Actively maintained and supported by a large community.
  - Supports multiple pixel formats and offers functions for pixel format conversion.

- **Usage Scenario:**
  Software renderers commonly use SDL2 to create a window, acquire a pixel buffer, and push pixel data for display, while handling user input and system events.

2. **GDI (Graphics Device Interface)**

- **Description:**
  GDI is the traditional Windows API for representing graphical objects and transmitting them to output devices such as monitors and printers.
- **Features:**
  - Provides basic drawing functions such as lines, rectangles, text rendering, and bitmap manipulation.
  - Supports device contexts (HDC) which abstract drawing surfaces like windows or printers.
  - Offers direct access to bitmap memory via `GetDIBits` and `SetDIBits`.
- **Relevance to CPU Rendering:**
  GDI enables software rendering by letting the programmer manipulate bitmap buffers and update windows without GPU acceleration. Although

dated compared to modern graphics APIs, GDI remains reliable and lightweight for CPU-based rendering tasks on Windows.

- **Limitations:**
  - Performance is generally lower than GPU-accelerated APIs.
  - Lacks advanced graphics features such as shaders or hardware compositing.
  - Less suited for complex animations or high frame rates.
- **Usage Scenario:**
  Useful for simple software rendering demos, tools, or legacy applications where GPU-based acceleration is either unnecessary or unavailable.

3. **Win32 API**

- **Description:**
  The Win32 API is the core Windows operating system API, encompassing system-level programming, including window creation, message handling, and drawing primitives.
- **Role in CPU Graphics:**
  - Provides low-level control over window lifecycles, message loops, and device contexts required for rendering.
  - Allows creation of pixel buffers using bitmaps and surfaces that can be directly accessed for pixel-level manipulation.
  - Supports interfacing with GDI for drawing operations.
- **Usage:**
  CPU-only renderers often rely on Win32 API calls for window management and event processing, while rendering pixels manually into memory buffers updated via GDI or direct memory mapping.

2.1.5 Integration of Tools: Workflow Example

A typical modern Windows CPU-only rendering project setup might look like this:

1. **Development Environment:**
   Visual Studio with MSVC or a terminal environment using MinGW GCC.
2. **Build System:**
   CMake scripts generate appropriate project files depending on compiler choice.
3. **Graphics and Input:**
   SDL2 creates a window and provides a software surface.
4. **Rendering:**
   The renderer locks the SDL2 surface's pixel buffer, writes pixels directly for frame updates, and unlocks it.
5. **Presentation:**
   SDL2 updates the window surface; the Win32 API handles the underlying window messages.

This workflow offers a blend of high control and portability while leveraging modern build tools.

2.1.6 Summary

This section outlined the critical tools for Windows-based software rendering:

- **MinGW and MSVC:** Two robust compiler options catering to open-source and commercial development workflows.
- **CMake:** The de facto build system for cross-compiler, multi-platform management.

- **SDL2:** The versatile multimedia library enabling window management and pixel buffer access suitable for CPU rendering.
- **GDI and Win32 API:** Native Windows APIs that provide essential drawing and system control capabilities.

Mastering these tools sets the foundation for efficient, maintainable, and portable CPU-only graphics programming on Windows platforms.

2.2 Required Tools for Linux: GCC/Clang, X11, SDL2, Framebuffer Access

2.2.1 Introduction

Linux remains a favored environment for low-level and graphics programming due to its open architecture, rich toolchains, and versatile graphics subsystems. This section examines the essential tools and libraries available on modern Linux distributions that support software rendering on the CPU, emphasizing compilers, windowing systems, multimedia libraries, and direct framebuffer access.
These components form the backbone of any CPU-based graphics project on Linux and enable developers to build efficient, portable, and flexible software renderers.

2.2.2 Compilers: GCC and Clang

1. **GNU Compiler Collection (GCC)**

   GCC is the traditional and most widely used compiler on Linux systems, supporting C, C++, and many other languages.

   - **Features:**
     - Full support for modern C++ standards, including C++17 and C++20.
     - Mature optimization capabilities targeting multiple CPU architectures (x86, ARM, RISC-V, etc.).
     - Extensive toolchain integration, including debugging (GDB) and profiling tools.
   - **Advantages for graphics programming:**
     - Stable, reliable, and well-documented.

  - Integrated with most Linux distributions by default.
  - Easily extensible and configurable via flags for performance tuning.

2. **Clang/LLVM**

   Clang, part of the LLVM project, is a modern compiler infrastructure increasingly popular on Linux.

   - **Features:**
     - Modular design with excellent diagnostics and error messages.
     - Competitive performance and optimization capabilities.
     - Good support for recent C++ standards and experimental features.
   - **Advantages for graphics programming:**
     - Faster compile times and clearer warnings help rapid development.
     - Supports sanitizers (address, thread, memory) useful in debugging complex rendering code.
     - Increasingly integrated into Linux distributions alongside GCC.

3. **Choosing Between GCC and Clang**

   Both compilers are viable for software rendering projects. Some developers use GCC for compatibility and Clang for development agility, or switch depending on platform or debugging needs.

2.2.3 Windowing System: X11 (X Window System)

1. **Overview of X11**

   The X Window System (X11) is the foundational graphical windowing system used on most traditional Linux desktops.

- **Core functionality:**
  - Provides low-level primitives for window creation, event handling, and drawing.
  - Client-server architecture enabling network-transparent graphics.
- **Relevance to CPU-based rendering:**
  - Allows applications to create windows and obtain drawable surfaces.
  - Provides access to framebuffer-like memory areas through XImage structures.
  - Supports event-driven programming for keyboard, mouse, and window events.

2. **Modern Usage and Limitations**

- While X11 remains dominant, newer systems like Wayland are emerging; however, many legacy and current software rendering projects rely on X11 due to stability and maturity.
- X11 performance can be a bottleneck for high-framerate software rendering, but its simplicity and wide support make it suitable for learning and moderate workloads.

2.2.4 SDL2 (Simple DirectMedia Layer 2)

1. **Role of SDL2 on Linux**

SDL2 is a cross-platform multimedia library that abstracts details of window creation, input handling, and pixel buffer management across diverse operating systems.

- **Why SDL2 is crucial:**

  - Provides a consistent API for CPU-based graphics rendering independent of underlying windowing systems.
  - Handles cross-platform input events from keyboard, mouse, joystick, and more.
  - Supports direct pixel access surfaces suitable for software rendering pipelines.

- **Features introduced after 2019:**
  - Continued enhancements in input device support.
  - Improved multi-window and high-DPI support.
  - Seamless integration with modern build systems like CMake.

2. **Integration in Software Rendering**

   - SDL2 surfaces or textures can serve as the final framebuffer where CPU-generated pixel data is copied before presenting to the screen.
   - Its event loop simplifies input handling, enabling responsive interactive graphics applications.

2.2.5 Framebuffer Access

1. **Direct Framebuffer Devices in Linux**

   The Linux framebuffer (`/dev/fb0`) provides a device interface to graphics hardware's framebuffer memory, enabling applications to render pixels directly to the screen without a windowing system.

   - **Advantages:**
     - Allows software rendering without the overhead of X11 or Wayland.

    - Suitable for embedded systems, kiosks, or minimal graphical environments.

- **Technical considerations:**
    - Requires appropriate permissions (usually root).
    - Pixel format and screen resolution must be managed manually.
    - No native windowing or input event support, so additional mechanisms needed for interaction.

2. **Using Framebuffer in Modern Context**

    - Framebuffer access is still relevant for lightweight or embedded graphics applications that rely solely on CPU rendering.
    - Contemporary tools and libraries may provide abstractions over framebuffer devices to simplify development.

2.2.6 Toolchain and Workflow Integration

Combining the above tools yields a flexible and powerful environment:

- **Compiling with GCC or Clang:** Utilizing command-line or CMake-managed builds to produce optimized executables.
- **Window management with X11 or SDL2:** SDL2 offers simplified cross-platform windowing and input, while X11 allows low-level control when required.
- **Rendering to SDL2 surfaces or direct framebuffer:** Software rendering pipelines write pixels into system memory buffers which SDL2 or X11 then present.
- **Debugging and profiling:** Tools such as GDB, Valgrind, and Sanitizers support identifying issues in CPU-intensive rendering code.

2.2.7 Summary

For Linux-based CPU-only graphics programming, the essential toolkit comprises:

- **GCC and Clang** as mature, optimized compilers supporting modern C++ standards.
- **X11** as a traditional windowing system enabling window creation and pixel manipulation.
- **SDL2** as a high-level, cross-platform library streamlining input and framebuffer handling.
- **Framebuffer device access** for direct, hardware-agnostic rendering in minimalist environments.

Mastering these tools equips developers to build efficient software renderers on Linux, leveraging the flexibility and openness of the platform while accommodating a wide range of application targets from desktop to embedded systems.

2.3 Differences Between Direct Memory Access and High-Level APIs

2.3.1 Introduction

In software rendering on modern computer systems, understanding how to interact with graphics memory and rendering surfaces is fundamental. Two primary approaches exist for writing pixel data to the screen: **direct memory access (DMA)** to framebuffers or pixel buffers, and the use of **high-level graphics APIs** that abstract memory operations.

This section explores the technical distinctions, advantages, disadvantages, and practical considerations of both approaches, providing clarity to guide developers in choosing the most suitable method for their software rendering projects.

2.3.2 Direct Memory Access (DMA)

1. **Concept and Mechanism**

   Direct memory access refers to the programmer's ability to manipulate the memory region that represents the pixel buffer — commonly known as a **framebuffer** — directly. This buffer corresponds to the raw pixels displayed on the screen or within a window.

   - **How it works:**
     - The program obtains a pointer or reference to a contiguous block of memory.
     - Each byte or group of bytes in this buffer corresponds to one or more pixels.

- The program writes pixel values directly into this memory region, controlling every pixel's color and intensity.

- **Typical environments:**
  - Accessing video memory in DOS or early Windows environments.
  - Writing to memory buffers provided by SDL surfaces, X11 XImages, or Linux framebuffer devices.
  - Manipulating pixel arrays in offscreen buffers used for double buffering.

2. **Advantages of Direct Memory Access**

   - **Maximum control:** The programmer has absolute freedom to set each pixel value with no API-imposed constraints.
   - **Performance potential:** Avoiding API overhead can reduce CPU cycles spent on drawing operations, which is critical in CPU-bound rendering.
   - **Flexibility:** Enables implementing custom pixel formats, blending modes, or rendering techniques unavailable or inefficient in high-level APIs.
   - **Deterministic behavior:** No hidden operations or side effects, enabling precise optimization.

3. **Challenges and Limitations**

   - **Platform dependency:** Memory layout and pixel formats vary by hardware and operating system, requiring careful management.
   - **Complexity:** Handling synchronization, caching, and memory barriers may be necessary to avoid visual artifacts.
   - **No abstraction:** Developers must manually handle windowing, input, and buffer swapping, which can be time-consuming.

- **Safety risks:** Directly writing memory risks buffer overruns or corruptions without strict bounds checking.

2.3.3 High-Level Graphics APIs

1. **Overview of High-Level APIs**

   High-level graphics APIs encapsulate the details of memory management and rendering processes, offering developers a more convenient interface for drawing operations.

   Examples include:

   - **SDL2's rendering and surface APIs**
   - **Win32 GDI (Graphics Device Interface)**
   - **X11 drawing functions**
   - **Other multimedia libraries with graphics abstractions**

   These APIs often provide functions such as:

   - Drawing primitives (lines, rectangles, circles).
   - Filling areas with colors or patterns.
   - Blitting (copying) pixel buffers between surfaces.
   - Text rendering and image manipulation.

2. **Advantages of High-Level APIs**

   - **Ease of use:** Simplify graphics programming by abstracting pixel-level manipulation.
   - **Portability:** Abstract differences between operating systems and hardware.

- **Integrated input and window management:** Many provide event handling and window lifecycle management alongside graphics.
- **Optimizations:** May include built-in buffering, clipping, and hardware acceleration support where available.
- **Safety:** APIs often include bounds checking and manage buffer lifetimes.

3. **Limitations**

- **Performance overhead:** API calls introduce function call overhead and sometimes memory copying, which can reduce rendering speed.
- **Less granular control:** Some APIs limit direct access to pixel data or impose specific formats, restricting custom rendering techniques.
- **Hidden behavior:** Internal state and optimizations may cause side effects or require understanding of the API's lifecycle for correct use.

2.3.4 Practical Examples: Direct Access vs High-Level API

1. **Direct Memory Access Example**

In SDL2, accessing the pixel buffer directly involves locking a surface, obtaining a pointer to the pixels, modifying them, and unlocking:

```
SDL_Surface* surface = SDL_GetWindowSurface(window);
SDL_LockSurface(surface);
uint32_t* pixels = (uint32_t*)surface->pixels;
int width = surface->w;
int height = surface->h;

for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        pixels[y * width + x] = 0xFF0000FF; // Set pixel to opaque
         ↪  blue
    }
}

SDL_UnlockSurface(surface);
SDL_UpdateWindowSurface(window);
```

This approach grants direct pixel manipulation, requiring careful handling of pixel format and memory pitch.

2. **High-Level API Example**

   Using GDI in Windows to fill a window with a solid color is straightforward:

```
PAINTSTRUCT ps;
HDC hdc = BeginPaint(hwnd, &ps);
HBRUSH brush = CreateSolidBrush(RGB(0, 0, 255)); // Blue
FillRect(hdc, &ps.rcPaint, brush);
DeleteObject(brush);
EndPaint(hwnd, &ps);
```

Here, the developer instructs GDI to fill the window's update region with blue without managing pixels directly.

2.3.5 Choosing the Right Approach

1. **When to Use Direct Memory Access**

   - Developing highly optimized, performance-sensitive software renderers.

- Implementing novel or experimental rendering algorithms requiring custom pixel format manipulation.
- Building cross-platform renderers that write to in-memory buffers before display.
- Educational purposes to learn low-level rendering fundamentals.

2. **When to Use High-Level APIs**

- Rapid prototyping or applications where development speed matters.
- Projects that do not demand maximum rendering performance.
- When leveraging platform-native UI and input management.
- When portability and ease of maintenance take precedence.

2.3.6 Summary

Understanding the distinctions between direct memory access and high-level graphics APIs is crucial for CPU-based graphics programming. Direct memory access provides unmatched control and potential performance but demands meticulous management and familiarity with low-level details. High-level APIs offer abstraction, safety, and convenience but with some cost in flexibility and speed.

Modern software rendering projects often combine both approaches—using direct memory access to optimize critical rendering paths while leveraging high-level APIs for window management and input handling—balancing control with productivity.

2.4 Choosing Pixel Formats (RGB, BGR, ARGB, Grayscale)

2.4.1 Introduction

Selecting the appropriate pixel format is a fundamental decision in software rendering that directly impacts performance, compatibility, and image quality. Pixel formats define how color and transparency information are stored in memory, specifying the layout, order, and bit depth of the color channels.

This section explores common pixel formats encountered in CPU-only graphics programming, their characteristics, typical usage scenarios, and considerations when choosing the format best suited for your software renderer.

2.4.2 Understanding Pixel Format Components

Pixels represent color information through channels, usually including:

- **Red (R)**
- **Green (G)**
- **Blue (B)**
- **Alpha (A)** (optional transparency/opacity channel)
- **Grayscale** (single luminance channel for monochrome images)

The arrangement of these channels and their bit depth per channel (usually 8 bits, but sometimes higher or lower) define the pixel format.

2.4.3 Common Pixel Formats

1. **RGB (Red-Green-Blue)**

   - **Description:** Stores three color channels in order Red, Green, then Blue.
   - **Typical layout:** Each pixel consists of three or four bytes, commonly 24-bit RGB (8 bits per channel, no alpha) or 32-bit RGBX (where X is unused or padding).
   - **Memory representation:**
     - For 24-bit RGB: `[R][G][B]` per pixel sequentially.
     - For 32-bit RGBX: `[R][G][B][X]` with padding for alignment.
   - **Usage:** Common in systems where red is the primary channel or in file formats like BMP and TIFF.
   - **Performance considerations:** Reading and writing RGB can be straightforward; however, many hardware and software systems expect pixel data in different channel orders, which may require conversions.

2. **BGR (Blue-Green-Red)**

   - **Description:** Similar to RGB but with the byte order reversed; Blue comes first, then Green, then Red.
   - **Typical layout:** 24-bit BGR or 32-bit BGRX.
   - **Common in:** Windows GDI and many video-related APIs use BGR ordering internally for performance and historical reasons.
   - **Practical implication:** When working with Windows APIs like GDI, the pixel buffers you receive or send will usually be in BGR format.

- **Conversion needs:** When exchanging images between systems or APIs that expect RGB, you may need to swap red and blue channels.

3. **ARGB (Alpha-Red-Green-Blue)**

   - **Description:** Extends RGB by adding an alpha channel that controls transparency.
   - **Typical layout:** 32-bit ARGB with one byte per channel.
   - **Channel order:** Alpha first, followed by Red, Green, Blue.
   - **Use cases:** Essential for rendering images with transparency, blending, and compositing effects.
   - **Variations:** Other orders exist such as RGBA, BGRA, depending on API or platform conventions.
   - **Considerations:** Handling alpha requires blending operations and may add computational cost.

4. **Grayscale**

   - **Description:** Uses a single channel to represent luminance or intensity.
   - **Typical layout:** 8 bits per pixel representing shades of gray from black (0) to white (255).
   - **Advantages:**
     - Smaller memory footprint.
     - Faster processing since only one channel is handled.
   - **Use cases:** Image processing, rendering text, or specialized artistic effects.
   - **Limitations:** No color information, so not suitable for full-color rendering.

2.4.4 Bit Depth and Alignment

- **Bit Depth:** Usually 8 bits per channel for standard color fidelity, but formats with 16 or 10 bits per channel exist for high dynamic range (HDR) or professional imaging.
- **Pixel Alignment:** Many systems use 32-bit alignment (4 bytes per pixel) for performance reasons, even if only 24 bits are needed for RGB. Padding bytes ensure memory alignment favorable to the CPU and memory subsystem.
- **Stride and Pitch:** The number of bytes in one row of pixels may exceed the pixel count multiplied by bytes per pixel due to alignment, requiring careful calculation when addressing pixels.

2.4.5 Choosing Pixel Formats for Software Rendering

1. **Consider the Target Platform and API**

   - Windows GDI and many Windows APIs expect BGR or BGRA.
   - SDL2 surfaces may be in ARGB, ABGR, or other formats depending on platform and setup.
   - X11 often uses 24-bit RGB or 32-bit formats.
   - Framebuffer devices vary widely; querying the pixel format is mandatory before rendering.

2. **Performance Implications**

   - Using native pixel formats avoids costly conversions and increases rendering speed.

  - Formats with alpha channels require extra computation for blending; if transparency is not needed, prefer opaque RGB/BGR formats.
  - 32-bit aligned formats usually render faster than 24-bit due to memory alignment.

3. **Memory Footprint**

   - Grayscale formats consume less memory but are limited in expressiveness.
   - RGB and BGR with 24 or 32 bits per pixel require more memory, affecting cache utilization.

4. **Rendering Pipeline Compatibility**

   - Your software renderer must handle input and output pixel formats consistently.
   - When reading image files or textures, convert them to the internal pixel format of your rendering buffer.
   - When presenting pixels to the display, convert to the format expected by the windowing system or framebuffer device.

2.4.6 Handling Pixel Format Conversion

- Efficient pixel format conversion routines are crucial for performance.
- Channel swapping between RGB and BGR is a common operation.
- Alpha premultiplication may be necessary for correct blending.
- Using SIMD instructions and multi-threading can accelerate conversions.

2.4.7 Summary

Choosing the right pixel format is a tradeoff between compatibility, performance, memory usage, and feature requirements such as transparency. Understanding the common pixel layouts — RGB, BGR, ARGB, and grayscale — and their implications enables developers to design effective software rendering pipelines tailored to their target environments.

Modern tools and APIs often provide facilities to query and handle pixel formats dynamically, but a solid foundational understanding remains essential for any CPU-only graphics programmer.

Chapter 3

Memory and Pixels: The Foundation

3.1 Understanding the Framebuffer Concept

3.1.1 Introduction

The framebuffer is the central abstraction in all raster-based graphics systems—whether in hardware (like GPU-based pipelines) or software rendering performed entirely on the CPU. At its core, the framebuffer is a contiguous block of memory that holds the pixel data to be displayed on the screen. This section explores the concept of the framebuffer from a low-level perspective, focusing on CPU-based graphics programming using modern tools and platforms. It covers the internal structure, memory layouts, access methods, and platform-specific considerations for managing and manipulating framebuffers.

3.1.2 What Is a Framebuffer?

A framebuffer is a region of system memory (RAM) that contains a representation of the image currently or soon to be displayed on the screen. Each element in this memory corresponds to a pixel, and its content encodes the color of that pixel according to a defined pixel format (e.g., RGB888, ARGB8888, Grayscale8).
When no GPU is used, the framebuffer is updated entirely by the CPU. The operating system or application is responsible for copying this memory region to a display output, either through software APIs (such as SDL, X11, GDI) or direct hardware access (e.g., `/dev/fb0` in Linux).

3.1.3 Logical and Physical Framebuffers

- **Logical Framebuffer**

  A **logical framebuffer** refers to the software-side representation, typically as a buffer in RAM. The application writes pixel data to this buffer during rendering operations. This buffer might be:

  - A simple `uint8_t*` array in memory.
  - A surface or bitmap object provided by SDL, Cairo, or Win32.
  - A shared memory segment mapped for X11 or Linux framebuffer.

- **Physical Framebuffer**

  A **physical framebuffer** is the memory-mapped video buffer directly tied to the hardware display. In modern systems, even when programming without a GPU, software does not usually write directly to video memory. Instead, it renders to a logical framebuffer and then instructs the OS or driver to blit or swap the contents to the physical framebuffer.

In embedded Linux environments or legacy DOS systems, direct access to physical framebuffer memory is possible and often necessary.

3.1.4 Memory Organization

- **Pixel Addressing**

  Each pixel in the framebuffer is located at a memory offset calculated using:

  ```
  address = base + (y * pitch) + (x * bytes_per_pixel)
  ```

  Where:

  - `base` is the start address of the framebuffer
  - `x`, `y` are the pixel coordinates
  - `pitch` is the number of bytes per row (not necessarily width * bytes_per_pixel due to padding)
  - `bytes_per_pixel` depends on the pixel format (e.g., 4 for ARGB32)

- **Row Alignment and Padding**

  Many platforms align rows in memory to 4- or 8-byte boundaries for performance. For instance, Windows bitmaps (GDI) align each row to a 4-byte boundary. SDL allows you to query and control surface pitch explicitly.

  Proper row alignment avoids CPU penalties on memory reads/writes and enables SIMD optimization.

3.1.5 Framebuffer in Modern Software Tools

- **SDL2**

  - Uses `SDL_Surface` as a software framebuffer.
  - The pixel data is accessible via `surface->pixels`.
  - Developers can lock/unlock surfaces before accessing them to ensure consistency and safety.
  - SDL
    allows specifying custom pixel formats (`SDL_PIXELFORMAT_ARGB8888`, etc.) and pitch.

- **Windows GDI / Win32**

  - Bitmaps (`HBITMAP`) can be created in memory.
  - A `BITMAPINFO` structure defines the pixel format and alignment.
  - CPU access is possible via `GetDIBits()` and `SetDIBitsToDevice()` or by creating a DIBSection.

- **Linux Framebuffer (`/dev/fb0`)**Linux Framebuffer (/dev/fb0)

  - Can be accessed directly by opening the device file and mapping its memory using `mmap()`.
  - `fb_var_screeninfo` and `fb_fix_screeninfo` structures provide pitch, color depth, and resolution.
  - Direct CPU rendering to framebuffer is used in headless or embedded systems.

- **X11 with Shared Memory (XShm)**

  - Shared memory XImages offer faster blits.
  - CPU draws to shared memory, then X11 copies the region to the screen using `XShmPutImage()`.

3.1.6 Performance Considerations

- **CPU Cache Behavior**

  Accessing the framebuffer linearly (row by row) is optimal for CPU caches. Accessing pixels column-by-column or randomly results in poor cache locality and slower performance.

- **Double Buffering**

  To avoid screen tearing and flicker, double buffering is often used:

  - A secondary buffer is rendered to off-screen.
  - At the end of the frame, the secondary buffer is copied to the display.

  In SDL, double buffering can be implemented manually by using two `SDL_Surface` objects or through `SDL_RenderPresent()` with a software renderer.

- **SIMD Acceleration**

  Modern CPUs support SIMD instructions (e.g., SSE, AVX2, NEON) that can process multiple pixels at once. Libraries like Skia and Cairo use these extensively.

  Custom SIMD routines in C/C++ can dramatically accelerate fill, copy, blend, and transform operations on the framebuffer.

3.1.7 Framebuffer in Emulators and Retro Game Engines

Many emulators (for NES, SNES, GBA, etc.) implement their own framebuffer to simulate the original hardware display. These framebuffers are:

- Typically small (e.g., 256×240 for NES)
- Rendered using software scanline techniques
- Scaled using pixel replication, bilinear filtering, or CRT-like effects before blitting to screen

SDL2 is often used in this context for presenting the final image.

3.1.8 Summary

The framebuffer remains the heart of software rendering, even in modern CPU-based graphics systems. By understanding how memory is organized, how pixels are laid out, and how to interface with the framebuffer through different APIs and platforms, developers can build efficient rendering pipelines suitable for games, emulators, UI systems, and embedded applications—all without requiring a GPU.

This foundational concept enables precise control over every pixel and serves as a gateway to learning advanced techniques like pixel compositing, scanline rendering, and rasterization, which will be explored in the chapters to follow.

3.2 Pixel Formats and Byte Arrangements

3.2.1 Introduction

Understanding pixel formats and how their bytes are arranged in memory is essential for any programmer working on software rendering without a GPU. The pixel format defines how color and transparency data for each pixel are stored, which directly impacts how the CPU reads, writes, and manipulates the framebuffer. This section delves into common pixel formats, their byte layouts, and practical considerations for software rendering on modern operating systems and APIs, with examples using current tools after 2019.

3.2.2 What Are Pixel Formats?

A **pixel format** specifies the color representation and storage scheme for each pixel in the framebuffer. It determines:

- The number of color channels (e.g., Red, Green, Blue, Alpha).
- The bit depth of each channel.
- The byte order and arrangement of channels in memory.
- The presence and location of alpha (transparency) data if any.

Correct interpretation of the pixel format is critical for accurate color rendering, blending, and image processing.

3.2.3 Common Pixel Formats

1. **RGB24 (24-bit RGB)**

- Stores 3 bytes per pixel: one byte each for Red, Green, and Blue.
- No alpha channel (fully opaque).
- Common byte orders:
  - **RGB**: bytes in order Red, Green, Blue.
  - **BGR**: bytes in order Blue, Green, Red (common in Windows BMP format).

This format is memory-efficient but lacks transparency support.

2. **ARGB32 / RGBA32 (32-bit with Alpha)**

- Stores 4 bytes per pixel: typically Red, Green, Blue, plus Alpha (transparency).
- Common byte orders:
  - **ARGB**: Alpha first, then Red, Green, Blue.
  - **RGBA**: Red first, then Green, Blue, Alpha.
  - **BGRA**: Blue first, Green, Red, Alpha — widely used in Windows and Direct2D.
- The alpha channel defines pixel transparency and is crucial for blending.

3. **Grayscale (8-bit)**

- Stores one byte per pixel representing intensity.
- No color channels; black to white scale.
- Useful for monochrome or luminance-only images.

4. **Indexed Color (Palette-Based)**

- Stores pixel values as indices into a color palette.
- Palette defines RGB values for each index.
- Common in older VGA modes and certain retro formats.

3.2.4 Byte Arrangements and Endianness

1. **Little Endian vs Big Endian**

   - On **little endian** systems (x86, x86-64), the least significant byte is stored first.
   - On **big endian** systems (some ARM, PowerPC), the most significant byte is stored first.

   This affects multi-byte pixel formats like ARGB32, especially when casting memory pointers to integers or when performing bitwise operations.

2. **Examples of Byte Layouts**

   For a pixel with components: A=0xAA, R=0xBB, G=0xCC, B=0xDD

Table 2-1: Pixel Format Memory Layouts

Format	**Memory Order (byte 0 → byte 3)**	**Notes**
ARGB32 (Windows BGRA)	DD CC BB AA	Alpha at highest address (byte 3)
RGBA32	BB CC DD AA	Alpha at byte 3
ABGR32	AA BB CC DD	Alpha at byte 0

Format	Memory Order (byte 0 → byte 3)	Notes
BGRA32 (Windows)	DD CC BB AA	Default for many Windows APIs

When accessing these bytes directly, understanding this order is vital for correct color manipulation.

3.2.5 Pixel Formats in Modern Tools and APIs

- **SDL2 Pixel Formats**
  - SDL provides `SDL_PixelFormatEnum` with formats such as:
    - `SDL_PIXELFORMAT_RGB24`
    - `SDL_PIXELFORMAT_ARGB8888`
    - `SDL_PIXELFORMAT_ABGR8888`
    - `SDL_PIXELFORMAT_RGBA8888`
    - `SDL_PIXELFORMAT_BGR24`
  - SDL surfaces expose `format->format` and `format->BytesPerPixel` to inform how to interpret pixel data.
  - SDL functions like `SDL_MapRGB()` and `SDL_MapRGBA()` handle format conversions internally.
- **Windows GDI / Direct2D**
  - Windows traditionally uses **BGRA** 32-bit pixel format for compatibility.
  - `BITMAPINFOHEADER` structures allow specifying pixel format bitmasks.

  - Direct2D software fallback renderers use **PBGRA** (pre-multiplied alpha BGRA) for blending accuracy.
  - Pre-multiplied alpha means color channels are multiplied by alpha to improve blending speed and quality.

- **Linux and X11**

  - X11 visuals define pixel formats via `XImage` structures.
  - Many X11 applications use 24-bit or 32-bit formats, but byte orders can differ based on system endianness.
  - Modern Linux compositors abstract pixel formats for client applications.
  - Framebuffer devices `/dev/fb0` provide `fb_var_screeninfo` which includes `red`, `green`, and `blue` bitfield definitions to describe pixel layout.

3.2.6 Choosing the Appropriate Pixel Format

When programming software rendering pipelines on the CPU, selecting the optimal pixel format depends on several factors:

- **Performance:** 32-bit formats align well with CPU registers and are easier to manipulate using SIMD instructions.
- **Transparency Needs:** Use ARGB/RGBA formats if alpha blending is required.
- **Memory Footprint:** 24-bit RGB or 8-bit grayscale reduce memory usage at the cost of flexibility.
- **Platform Compatibility:** Match the native format of the output API to minimize costly conversions.

- **Hardware or Driver Constraints:** On embedded platforms or older hardware, specific formats might be required.

3.2.7 Pixel Format Conversion and Blitting

Software renderers often need to convert pixel data between formats:

- From internal rendering buffers (e.g., RGBA32) to the platform's native format (e.g., BGRA32 for Windows GDI).
- From indexed or grayscale formats to full RGB or RGBA formats.

Efficient conversion is a key part of high-performance software rendering. Modern C++ libraries often use SIMD acceleration for these operations.

3.2.8 Practical Example: Pixel Access in SDL2

Suppose you have an SDL surface with format `SDL_PIXELFORMAT_ARGB8888`. The pixel data is a contiguous array of 32-bit integers.

```
uint32_t* pixels = (uint32_t*)surface->pixels;
int pitch = surface->pitch / 4; // pitch in pixels, not bytes

for(int y = 0; y < surface->h; ++y) {
    for(int x = 0; x < surface->w; ++x) {
        uint32_t pixel = pixels[y * pitch + x];
        uint8_t a = (pixel >> 24) & 0xFF;
        uint8_t r = (pixel >> 16) & 0xFF;
        uint8_t g = (pixel >> 8) & 0xFF;
        uint8_t b = pixel & 0xFF;
        // manipulate pixel colors here
    }
}
```

This example illustrates reading and decomposing the pixel data according to the ARGB8888 format.

3.2.9 Summary

A deep understanding of pixel formats and their byte arrangements is indispensable for CPU-based graphics programming. Precise knowledge about how color and alpha data are packed enables developers to correctly render, blend, and manipulate images across diverse platforms and APIs. Modern rendering tools and libraries provide abstractions and helpers, but mastering these low-level details ensures both correctness and efficiency in software-only rendering systems.
This knowledge lays the groundwork for advanced pixel operations and software rasterization techniques, which will be covered in subsequent chapters.

3.3 Memory Alignment and Optimization

3.3.1 Introduction

Efficient memory management is critical for high-performance graphics programming on the CPU. Since software rendering relies heavily on direct manipulation of pixel data in memory, understanding **memory alignment** and **optimization techniques** can dramatically improve rendering speed, reduce CPU cache misses, and enable effective use of modern CPU instruction sets. This section explores memory alignment principles, cache behavior, and practical optimization strategies using contemporary tools and APIs from 2019 onward.

3.3.2 Understanding Memory Alignment

1. **What Is Memory Alignment?**

   Memory alignment refers to arranging data in memory addresses that are multiples of a specific value, typically the size of the CPU's word or cache line (e.g., 4, 8, 16, 32 bytes). Proper alignment ensures that:

   - CPU loads and stores occur efficiently.
   - SIMD instructions operate on aligned data.
   - Cache lines are optimally utilized.

   Misaligned memory accesses can lead to performance penalties, including additional CPU cycles and potential hardware exceptions on some architectures.

2. **Alignment Units Relevant to Graphics Programming**

   - **Byte (8 bits):** The smallest addressable unit.

- **Word (32 bits / 4 bytes):** Typical size for many CPU operations.
- **Double Word (64 bits / 8 bytes):** Used by 64-bit architectures.
- **Cache Line (typically 64 bytes on modern CPUs):** The unit of data transferred between CPU cache and main memory.

3.3.3 Importance of Alignment in Framebuffers

Framebuffers are large arrays of pixels stored sequentially in memory. Alignment considerations include:

- **Row (Scanline) Alignment:**
  Aligning each row of pixels to a multiple of 16 or 32 bytes improves cache line utilization and can speed up row-by-row rendering and copying.
- **Pixel Data Alignment:**
  Aligning pixel data structures to 4- or 8-byte boundaries is essential when using 32-bit or 64-bit pixel formats (e.g., ARGB8888).
- **Surface or Buffer Alignment:**
  Allocating entire surfaces aligned to 16-, 32-, or 64-byte boundaries facilitates the use of SIMD (Single Instruction, Multiple Data) instructions (e.g., SSE, AVX).

3.3.4 CPU Cache Architecture and Its Impact

1. **Cache Hierarchy**

   Modern CPUs have multi-level caches (L1, L2, L3) that temporarily store frequently accessed data. Efficient use of these caches minimizes slow memory accesses:

   - **L1 Cache:** Smallest and fastest; typically 32–64 KB.

- **L2 Cache:** Larger and slower than L1; typically 256 KB to 1 MB.
- **L3 Cache:** Largest and slowest; several MBs.

2. **Cache Line Size and False Sharing**

   - **Cache Line Size:** Typically 64 bytes. Data accessed together should be located within the same cache line to avoid frequent reloads.
   - **False Sharing:** Occurs when multiple threads modify variables that reside on the same cache line, causing performance degradation due to cache coherence traffic.

   In software rendering on the CPU, understanding and aligning pixel data to cache lines improves rendering throughput.

3.3.5 Memory Alignment in Modern Development Tools

1. **Alignment Support in C/C++**

   - Use compiler directives and attributes to enforce alignment, e.g.:
     - `alignas(16)` in C++11 and later to specify 16-byte alignment.
     - Compiler-specific attributes like `__declspec(align(16))` (MSVC) or `__attribute__((aligned(16)))` (GCC/Clang).

   Example:

   ```
   alignas(16) uint32_t framebuffer[WIDTH * HEIGHT];
   ```

   - Use aligned memory allocation functions:

  - C11's `aligned_alloc()`.
  - `_aligned_malloc()` on Windows.
  - `posix_memalign()` on POSIX-compliant systems.

2. **SIMD Intrinsics and Alignment**

   - SIMD instruction sets such as **SSE**, **AVX**, and **NEON** require data to be aligned on 16- or 32-byte boundaries for optimal performance.
   - Misaligned SIMD loads/stores can incur penalties or cause faults.
   - Many modern CPUs support unaligned SIMD operations but with reduced speed.

3.3.6 Optimization Techniques for Software Rendering

1. **Data Structure Layout**

   - Use **Structure of Arrays (SoA)** instead of **Array of Structures (AoS)** to enable better vectorization.
   - Pack pixel data to minimize wasted space and ensure tight memory layout.

2. **Loop Unrolling and Prefetching**

   - Manual or compiler-assisted loop unrolling reduces loop overhead.
   - Use prefetching intrinsics to load upcoming pixel data into cache before use.

3. **Minimizing Cache Misses**

   - Process pixels in memory order to maximize spatial locality.
   - Avoid jumping randomly through framebuffer memory.

- Group operations to reuse loaded cache lines.

4. **Double Buffering and Page Alignment**

    - Allocate front and back buffers with page-aligned memory (typically 4 KB) to reduce page faults.
    - Use double buffering to minimize visible tearing.

3.3.7 Platform-Specific Considerations

- **Windows**
    - Windows APIs such as **VirtualAlloc** can be used to allocate page-aligned buffers.
    - GDI bitmaps often benefit from DWORD-aligned scanlines (pitch aligned to multiples of 4 bytes).
    - Direct2D software fallback renderer expects aligned buffers for efficient pixel processing.
- **Linux**
    - Use `posix_memalign()` or `mmap()` with `MAP_HUGETLB` for huge pages to improve large buffer performance.
    - Framebuffer devices require row stride (pitch) aligned to hardware requirements.
    - X11 and Wayland compositors may have alignment constraints depending on driver.
- **Embedded Systems**

- Memory alignment is even more critical due to limited CPU and cache.
- Use hardware-specific DMA-friendly buffer alignment to offload memory transfers.

3.3.8 Practical Example: Allocating Aligned Framebuffer in C++

```cpp
#include <cstdlib>
#include <memory>

constexpr size_t WIDTH = 1920;
constexpr size_t HEIGHT = 1080;
constexpr size_t PIXEL_SIZE = 4; // bytes per pixel (e.g., ARGB32)

int main() {
    size_t buffer_size = WIDTH * HEIGHT * PIXEL_SIZE;
    void* framebuffer = nullptr;

    // Allocate 32-byte aligned memory
    int res = posix_memalign(&framebuffer, 32, buffer_size);
    if (res != 0) {
        // handle allocation failure
        return -1;
    }

    // Use framebuffer ...

    free(framebuffer);
    return 0;
}
```

This code ensures the framebuffer is aligned on a 32-byte boundary suitable for AVX operations.

3.3.9 Summary

Memory alignment and optimization are foundational to maximizing performance in CPU-based graphics programming. By carefully aligning framebuffer memory, optimizing data access patterns, and leveraging platform-specific memory allocation functions, developers can reduce cache misses, enable efficient SIMD usage, and ultimately achieve smoother and faster software rendering pipelines.
As CPUs continue evolving with wider SIMD registers and more complex cache hierarchies, mastering memory alignment remains a crucial skill for software rasterizers and rendering engines relying solely on the CPU.

3.4 Endianness Considerations

3.4.1 Introduction

Endianness is a fundamental concept affecting how multi-byte data values are stored and interpreted in computer memory. In graphics programming on the CPU — especially when handling pixel data at the byte level — understanding and correctly managing endianness is essential for portability, data correctness, and interoperability across different platforms and architectures.

This section provides a comprehensive overview of endianness, its relevance to pixel data manipulation, and practical considerations and techniques for handling endianness in modern CPU-only graphics programming environments. The discussion incorporates developments and tools from 2019 onward.

3.4.2 What Is Endianness?

Endianness defines the byte order used to represent multi-byte data types (such as 16-bit, 32-bit, or 64-bit integers) in memory. The two primary types are:

- **Little-endian:** The least significant byte (LSB) is stored at the lowest memory address, with more significant bytes following at higher addresses.
- **Big-endian:** The most significant byte (MSB) is stored at the lowest memory address, with less significant bytes following.
- **Example: 32-bit Integer 0x12345678**

Table 4-2: Endianness Memory Representation

Address	**Little-endian**	**Big-endian**
0	0x78	0x12
1	0x56	0x34
2	0x34	0x56
3	0x12	0x78

3.4.3 3Why Endianness Matters in Pixel Data

Pixel data formats—commonly represented as 32-bit values for ARGB, RGBA, BGRA, etc.—involve multiple color channels packed into one integer value. How these bytes map to actual colors depends on the system's endianness, the pixel format specification, and the graphics API in use.

Key reasons why endianness is critical:

- **Correct color interpretation:** The order of R, G, B, and A channels in memory can appear swapped or incorrect if endianness is not accounted for.
- **Data interchange and file I/O:** Images or textures saved or loaded from disk may store pixel data in a fixed endianness; software must translate this appropriately on load.
- **Cross-platform compatibility:** Software designed to run on both little-endian (x86, ARM in little-endian mode) and big-endian platforms (some older RISC architectures, network byte order) must handle byte order explicitly.

3.4.4 Common Pixel Formats and Endianness Implications

1. **ARGB and RGBA Formats**

   - In a **32-bit ARGB** pixel value, each byte corresponds to one color channel: Alpha, Red, Green, Blue.
   - On **little-endian systems**, this 32-bit integer is stored in memory as Blue (lowest byte), Green, Red, Alpha (highest byte).
   - On **big-endian systems**, the order in memory directly matches the logical channel order: Alpha (lowest byte), Red, Green, Blue.

2. **BGR and BGRA Formats**

   Similar principles apply, but the channel order differs logically, requiring extra care in mapping color components.

3. **Grayscale Formats**

   Grayscale typically uses single-byte or 16-bit values without multi-channel packing, reducing complexity. However, multi-byte grayscale formats (e.g., 16-bit) still require endianness consideration.

3.4.5 Detecting and Handling Endianness Programmatically

1. **Detecting Endianness**

   Most modern platforms provide macros or functions to detect endianness at compile or runtime. Common approaches include:

   - Compile-time detection with predefined macros, e.g., on GCC/Clang:

```
#if __BYTE_ORDER__ == __ORDER_LITTLE_ENDIAN__
    // Little-endian system
#elif __BYTE_ORDER__ == __ORDER_BIG_ENDIAN__
    // Big-endian system
#endif
```

- Runtime detection by inspecting a multi-byte integer:

```
bool isLittleEndian() {
    uint16_t value = 0x1;
    char *ptr = (char*)&value;
    return ptr[0] == 1;
}
```

2. **Byte Swapping Utilities**

   To convert data between endian formats, byte-swapping functions are used:

   - Standard C++20 introduces `<bit>` header with `std::byteswap`.
   - Platform-specific intrinsics like `_byteswap_ulong` (MSVC), `__builtin_bswap32` (GCC/Clang).

   Example:

```
#include <bit>
uint32_t swapEndian(uint32_t val) {
    return std::byteswap(val);
}
```

3. **Handling Endianness in Pixel Buffers**

   When reading or writing pixel buffers, especially from files or network streams, perform byte swapping if the source data endianness differs from the host system.

   For example, if an image is stored in big-endian ARGB and the system is little-endian, each 32-bit pixel must be byte-swapped before use.

3.4.6 Practical Examples and Tool Support

1. **SDL2 Pixel Formats**

   The SDL2 library (version 2.0.14 and later) abstracts endianness in pixel formats by defining constants like `SDL_PIXELFORMAT_ARGB8888`, which internally map to the correct byte layout for the current platform.

   This means developers can write portable code without manually swapping bytes if using SDL's surface and texture management functions correctly.

2. **Direct2D Software Rendering Fallback**

   Microsoft's Direct2D software renderer transparently handles endianness in pixel buffers, provided that developers use the specified pixel formats and API correctly. However, when manipulating raw memory buffers directly (e.g., in `ID2D1Bitmap`), understanding system endianness is necessary for correct color composition.

3. **Vulkan and OpenGL Considerations**

   Low-level APIs like Vulkan and OpenGL expect specific pixel formats with clearly defined byte orders. Endianness is often managed by drivers, but software renderers built on top of these APIs need explicit conversion when loading or generating pixel data.

3.4.7 Endianness in Cross-Platform and Embedded Development

Embedded systems may run on architectures with different endianness, requiring:

- Careful buffer design.
- Explicit endianness conversion routines.
- Use of cross-platform libraries that abstract these concerns.

Newer CPUs like ARMv8 support both little-endian and big-endian modes, making runtime endianness detection and handling crucial for portable embedded graphics code.

3.4.8 Summary and Best Practices

- Always identify the system endianness at program initialization.
- Use platform and API abstractions (SDL, Direct2D, Vulkan) whenever possible to avoid manual byte handling.
- When manipulating raw framebuffer data, especially in custom formats, implement robust byte-swapping routines.
- Test rendering output on multiple architectures if possible, including ARM-based systems, to verify correct color display.
- Keep pixel format documentation and endianness considerations synchronized to avoid subtle bugs.

Understanding endianness is key to accurate, portable, and maintainable CPU-based graphics rendering, especially in heterogeneous environments and when working close to memory.

3.5 Drawing Pixels Manually into Memory

3.5.1 Introduction

At the core of CPU-based graphics programming lies the fundamental operation of drawing pixels directly into memory. Unlike GPU-accelerated rendering, which relies on dedicated hardware to manage pixel output, software rendering requires explicit control over pixel data stored in framebuffers or memory buffers. Mastery of manual pixel drawing is essential for building any software renderer or low-level graphics system. This section covers the principles, techniques, and practical considerations for writing pixels directly to memory. It highlights common pixel formats, memory layouts, and optimization methods relevant to contemporary tools and operating systems from 2019 onwards.

3.5.2 Conceptual Overview of Pixel Drawing

Drawing a pixel manually involves calculating the exact memory address corresponding to a screen coordinate and writing the appropriate color value into that location. The process can be broken down into three primary tasks:

1. **Mapping 2D Coordinates to Memory Address:**
   Given the pixel's (x, y) position on a virtual framebuffer, compute the offset within the linear memory array.

2. **Encoding Color Values Appropriately:**
   Convert color components (red, green, blue, alpha) into the pixel format's binary representation.

3. **Writing the Value with Correct Memory Alignment and Access:**

Safely update the memory without violating alignment rules or causing race conditions.

3.5.3 Calculating Pixel Address in Memory

The framebuffer is commonly represented as a one-dimensional array of bytes or words, with pixels laid out in row-major order. To locate the pixel at position (x, y), use the formula:

$$\text{offset} = y \times \text{stride} + x \times \text{bytesPerPixel}$$

Where:

- **stride (or pitch)** is the number of bytes per row of pixels, including any padding bytes.
- **bytesPerPixel** depends on the pixel format (e.g., 4 bytes for 32-bit formats like ARGB8888).

Note: Stride can be larger than $\text{width} \times \text{bytesPerPixel}$ due to alignment or padding introduced by the OS or graphics API.

3.5.4 Writing Color Values in Different Pixel Formats

1. **Common Pixel Formats**

   - **ARGB8888 / RGBA8888:** 32 bits (4 bytes), one byte per channel.
   - **RGB565:** 16 bits, 5 bits red, 6 bits green, 5 bits blue.
   - **Grayscale (8-bit):** Single byte representing intensity.

   Each format requires different packing of color components into memory.

2. **Encoding Example: ARGB8888**

   Given 8-bit channel values for Alpha (A), Red (R), Green (G), Blue (B), the pixel value is:

   $$\text{pixelValue} = (A << 24)|(R << 16)|(G << 8)|B$$

   This 32-bit integer is then stored at the computed memory offset, respecting endianness.

3.5.5 Sample Implementation in C++

Here is a minimal example of manually setting a pixel in a 32-bit ARGB framebuffer:

```cpp
void setPixelARGB32(uint8_t* framebuffer, int stride, int x, int y,
 ↪  uint8_t a, uint8_t r, uint8_t g, uint8_t b) {
    int bytesPerPixel = 4;
    int offset = y * stride + x * bytesPerPixel;
    uint32_t* pixelPtr = reinterpret_cast<uint32_t*>(framebuffer +
     ↪  offset);

    uint32_t pixelValue = (static_cast<uint32_t>(a) << 24)
                          (static_cast < uint32_t > (r) << 16)
                          (static_cast<uint32_t>(g) << 8)  |
                          (static_cast<uint32_t>(b));

    *pixelPtr = pixelValue;
}
```

Notes:

- This example assumes the framebuffer is already allocated and mapped.

- Stride is in bytes and includes any padding.
- The function does not handle clipping or bounds checking for performance reasons.

3.5.6 Handling Memory Alignment and Performance

Direct memory writes to pixel buffers must consider alignment for efficient CPU access:

- **Aligned Writes:** Writing 32-bit or 64-bit values aligned to natural boundaries (4 or 8 bytes) allows the CPU to perform faster operations and avoid penalties.
- **Unaligned Writes:** May still work but can be slower or cause exceptions on some architectures.

To optimize, programmers often align framebuffers or pad scanlines so each row starts at an aligned address.

3.5.7 Dealing with Multiple Framebuffer Types and APIs

Depending on the environment and API, manual pixel drawing may require different approaches:

- **SDL2 Surfaces:** Provide a pixel buffer (`void* pixels`) and pitch; manual pixel writing is done similarly to the above.
- **Win32 GDI Bitmaps:** Use `SetDIBits` or `GetDIBits` to access pixel data, and manual modification is done by locking the bitmap memory and writing pixel values.
- **X11 Framebuffers:** Accessed via shared memory or XImage structures, manual pixel writes require synchronization with the X server.

- **Linux Framebuffer Device (/dev/fb0):** Directly write pixel data to the framebuffer device, requiring knowledge of screen resolution, pixel format, and stride.

3.5.8 Thread Safety and Concurrency

When drawing pixels manually in multi-threaded environments, developers must:

- Synchronize access to shared framebuffers.
- Avoid race conditions by using mutexes or lock-free programming techniques.
- Be aware of CPU cache coherency and memory barriers if working with multiple cores.

3.5.9 3Advanced Techniques

1. **Using SIMD for Faster Pixel Writes**

   Modern CPUs support SIMD (Single Instruction Multiple Data) instructions such as SSE, AVX, or NEON, enabling multiple pixels to be processed simultaneously, increasing throughput significantly.

   For example, writing 4 pixels (128 bits) at once using SSE intrinsics can reduce CPU cycles per pixel.

2. **Pixel Blending and Alpha Compositing**

   Beyond writing raw colors, software renderers implement alpha blending by reading the destination pixel, computing the composite color, and writing back the result.

   Manual pixel drawing functions thus often expand to include blending formulas, optimized with integer arithmetic and lookup tables.

3.5.10 Summary

- Manual pixel drawing is the building block of software rendering and requires precise calculation of memory offsets and pixel encoding.
- Understanding stride, pixel formats, and memory alignment is critical for correctness and performance.
- Contemporary tools like SDL2 and system APIs provide pixel buffers that can be manipulated directly, facilitating software rendering projects.
- Optimization through alignment, SIMD instructions, and thread-safe access improves performance on modern CPUs.

Mastering manual pixel manipulation lays the groundwork for more complex rendering techniques such as shapes, text, and image compositing purely on the CPU.

Chapter 4

Drawing Primitives (Lines, Rectangles, Circles)

4.1 Implementing Bresenham's Line Algorithm

4.1.1 Introduction

Drawing lines efficiently and accurately is a foundational task in graphics programming. In CPU-only graphics rendering, algorithms that minimize computational overhead while producing visually correct lines are essential. Bresenham's line algorithm is one of the most widely used algorithms for this purpose due to its simplicity, efficiency, and use of integer arithmetic.

This section presents a detailed explanation of Bresenham's line algorithm, its implementation considerations, and practical aspects relevant for modern CPU-based software renderers using tools and environments.

4.1.2 The Need for Bresenham's Line Algorithm

When drawing a line between two points on a pixel grid, a naive approach might use floating-point calculations to interpolate positions. However, floating-point operations are more expensive than integer arithmetic and can cause rounding errors, resulting in uneven or visually broken lines.
Bresenham's algorithm avoids floating-point math by incrementally deciding the best pixel to light up at each step, ensuring a continuous line that is visually appealing and computationally efficient.

4.1.3 Algorithm Overview

Bresenham's algorithm draws a straight line between two points (x_0, y_0) and (x_1, y_1) on a discrete grid by stepping through one axis (commonly the x-axis) and choosing the pixel in the y-direction that is closest to the theoretical line.
The key concept is an **error term** that accumulates the deviation from the ideal line path and determines when to step in the secondary axis.

4.1.4 Detailed Algorithm Steps

1. **Calculate Differences:**

$$\Delta x = |x_1 - x_0|$$

$$\Delta y = |y_1 - y_0|$$

2. **Determine Step Directions:**

$$s_x = \begin{cases} 1 & \text{if } x_0 < x_1 \\ -1 & \text{otherwise} \end{cases} \quad s_y = \begin{cases} 1 & \text{if } y_0 < y_1 \\ -1 & \text{otherwise} \end{cases}$$

3. **Initialize Error Term:**

$$\text{err} = \Delta x - \Delta y$$

4. **Iterate and Plot:**

   Repeat until $(x_0, y_0) = (x_1, y_1)$:

   - Plot pixel at (x_0, y_0).
   - Calculate $2 \times \text{err}$:
     - If $2 \times \text{err} > -\Delta y$, then:
       * $\text{err} = \text{err} - \Delta y$
       * $x_0 = x_0 + s_x$
     - If $2 \times \text{err} < \Delta x$, then:
       * $\text{err} = \text{err} + \Delta x$
       * $y_0 = y_0 + s_y$

4.1.5 Handling All Octants

The algorithm's logic naturally adapts to any line orientation by using the step directions and absolute differences. This ensures correct drawing whether the line is horizontal, vertical, diagonal, or at any angle.

4.1.6 Practical Implementation in C++

Below is a contemporary implementation compatible with modern C++ standards, suitable for integration with pixel buffer manipulation functions (as introduced in Chapter 3):

```cpp
void drawLineBresenham(int x0, int y0, int x1, int y1, uint32_t color,
                       uint8_t* framebuffer, int stride, int
                        ↪  bytesPerPixel) {
    int dx = abs(x1 - x0);
    int dy = abs(y1 - y0);
    int sx = (x0 < x1) ? 1 : -1;
    int sy = (y0 < y1) ? 1 : -1;
    int err = dx - dy;

    while (true) {
        // Calculate pixel offset and set pixel
        int offset = y0 * stride + x0 * bytesPerPixel;
        uint32_t* pixelPtr = reinterpret_cast<uint32_t*>(framebuffer +
        ↪  offset);
        *pixelPtr = color;

        if (x0 == x1 && y0 == y1) break;
        int e2 = 2 * err;
        if (e2 > -dy) {
            err -= dy;
            x0 += sx;
        }
        if (e2 < dx) {
            err += dx;
            y0 += sy;
        }
    }
}
```

Notes:

- The `framebuffer` pointer should be properly initialized and point to a valid pixel buffer.
- `stride` is the number of bytes per row.
- `bytesPerPixel` depends on pixel format (commonly 4 for 32-bit ARGB).
- Color is packed as a 32-bit integer matching the framebuffer format.

4.1.7 Performance Considerations and Optimizations

- **Integer Arithmetic:** Bresenham's algorithm uses only integer addition, subtraction, and bit-shifts, ideal for CPU efficiency.
- **Branch Prediction:** Modern CPUs perform best with predictable branches; restructuring the algorithm to reduce unpredictable branches can improve speed.
- **Loop Unrolling:** For long lines, unrolling the loop or using SIMD (Single Instruction, Multiple Data) can increase throughput.
- **Clipping:** Implement clipping before drawing lines to avoid writing outside framebuffer bounds, preventing memory corruption.

4.1.8 Integration with Modern Tools and Environments

- **SDL2:** The pixel buffer of an `SDL_Surface` or `SDL_Texture` locked in software mode can be passed as the framebuffer parameter, enabling direct pixel manipulation.

- **Win32 GDI and Direct2D (Software Fallback):** Framebuffer memory can be accessed similarly when using software rendering paths.
- **Linux Framebuffer and X11:** Mapping `/dev/fb0` or an XImage allows direct pixel access compatible with this drawing routine.

4.1.9 Testing and Validation

To ensure correctness:

- Test lines in all octants (all eight directions).
- Validate output on different pixel formats and screen resolutions.
- Compare against reference rasterization outputs.

4.1.10 Summary

Bresenham's line algorithm remains a crucial tool in CPU-only graphics programming due to its blend of simplicity and efficiency. Its use of integer arithmetic and ability to handle all line orientations make it ideal for software renderers, embedded systems, and environments where GPU acceleration is unavailable or unsuitable.
By mastering its implementation, developers lay a solid foundation for constructing more complex graphics primitives and algorithms in purely CPU-driven rendering pipelines.

4.2 Midpoint Circle and Ellipse Algorithms

4.2.1 Introduction

Drawing circles and ellipses efficiently on a pixel grid is a fundamental capability in graphics programming. Like lines, circles and ellipses require algorithms that produce visually accurate curves while minimizing computational expense.

The **Midpoint Circle** and **Midpoint Ellipse** algorithms are classic rasterization methods that use integer arithmetic to calculate the set of pixels best approximating these shapes. They are well-suited for CPU-only rendering environments where floating-point operations and complex calculations should be minimized.

This section explores these algorithms in depth, their mathematical foundation, implementation details, and practical considerations for modern systems and programming tools introduced since 2019.

4.2.2 The Challenge of Drawing Circles and Ellipses on a Pixel Grid

Rendering perfect circles and ellipses on discrete pixel grids is inherently an approximation problem due to the continuous nature of these curves versus the discrete nature of pixels.

Naive methods such as using trigonometric functions to calculate points on the circumference are computationally expensive and involve floating-point math, which can degrade performance in software-only rendering.

The midpoint algorithms address this by utilizing incremental integer calculations and symmetry properties to efficiently plot the curves.

4.2.3 Midpoint Circle Algorithm

1. **Mathematical Basis**

   The equation of a circle centered at the origin is:

   $$x^2 + y^2 = r^2$$

   The midpoint circle algorithm incrementally determines which pixel closest fits the circle's edge, moving stepwise from one octant to the next, leveraging symmetry to render the full circle by calculating points in just one octant.

   The core idea is to evaluate a **decision parameter** at the midpoint between two possible pixel positions to decide whether to move vertically or diagonally when plotting the next pixel.

2. **Algorithm Outline**

   - Start at $(x, y) = (0, r)$.
   - The initial decision parameter:

     $$p_0 = 1 - r$$

   - For each x from 0 to y:
     - Plot the eight symmetric points calculated by reflecting (x, y) across the circle's octants.
     - If $p < 0$, the next point is $(x + 1, y)$, and update p as:

       $$p = p + 2x + 3$$

     - Else, the next point is $(x + 1, y - 1)$, and update p as:

       $$p = p + 2(x - y) + 5$$

4.2.4 Midpoint Ellipse Algorithm

1. **Mathematical Basis**

   The general equation of an ellipse centered at the origin is:

   $$\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$$

   Where a and b are the horizontal and vertical radii, respectively.

   Unlike circles, the ellipse curve's slope varies non-uniformly, so the algorithm divides the drawing into two regions:

   - **Region 1**: Slope magnitude less than 1 (near horizontal).
   - **Region 2**: Slope magnitude greater than or equal to 1 (near vertical).

   Different decision parameters are used in each region to determine pixel positions incrementally.

2. **Algorithm Outline**

   - Begin at $(x, y) = (0, b)$.
   - Compute initial decision parameters for Region 1.
   - Increment x while adjusting y based on the decision parameter until the slope reaches -1.
   - Switch to Region 2, increment y while adjusting x similarly.
   - Plot four symmetrical points for each calculated coordinate (x, y), exploiting ellipse symmetry.

4.2.5 Implementation Considerations

1. **Symmetry Exploitation**

   Both algorithms use the symmetry properties of circles and ellipses to minimize computations by calculating pixels in one segment and reflecting them to others.

   - Circle: 8-way symmetry.
   - Ellipse: 4-way symmetry.

   This drastically reduces the number of calculations required.

2. **Integer Arithmetic**

   By relying on decision parameters updated via integer addition and subtraction, these algorithms avoid expensive floating-point operations, enhancing performance on CPUs without hardware floating-point units or when using software fallback renderers.

4.2.6 Example: Midpoint Circle Algorithm in Modern C++

Below is a simplified C++ example to draw a circle into a pixel buffer, suitable for integration with modern APIs such as SDL2, or direct framebuffer manipulation:

```cpp
void plotCirclePoints(int cx, int cy, int x, int y, uint32_t color,
↪  uint8_t* framebuffer, int stride, int bytesPerPixel) {
    // Plot 8 symmetric points
    auto setPixel = [&](int px, int py) {
        if (px < 0  py < 0) return; // Optional boundary check
        int offset = py * stride + px * bytesPerPixel;
        uint32_t* pixelPtr = reinterpret_cast<uint32_t*>(framebuffer +
        ↪  offset);
        *pixelPtr = color;
    };

    setPixel(cx + x, cy + y);
    setPixel(cx - x, cy + y);
    setPixel(cx + x, cy - y);
    setPixel(cx - x, cy - y);
    setPixel(cx + y, cy + x);
    setPixel(cx - y, cy + x);
    setPixel(cx + y, cy - x);
    setPixel(cx - y, cy - x);
}

void drawMidpointCircle(int cx, int cy, int radius, uint32_t color,
 ↪  uint8_t* framebuffer, int stride, int bytesPerPixel) {
    int x = 0;
    int y = radius;
    int p = 1 - radius;

    plotCirclePoints(cx, cy, x, y, color, framebuffer, stride,
     ↪  bytesPerPixel);

    while (x < y) {
        x++;
        if (p < 0) {
            p += 2 * x + 1;
        } else {
            y--;
            p += 2 * (x - y) + 1;
        }
        plotCirclePoints(cx, cy, x, y, color, framebuffer, stride,
         ↪  bytesPerPixel);
    }
}
```

4.2.7 Practical Notes for Modern Systems

- **Boundary Checks:** When drawing on real framebuffers, ensure pixel coordinates do not exceed framebuffer dimensions to avoid memory access violations.
- **Color Format Compatibility:** Ensure the color variable is packed correctly according to the framebuffer's pixel format (e.g., ARGB, BGRA).
- **Integration:** These routines can be integrated with SDL2 surfaces, Windows GDI bitmaps, or Linux framebuffer devices, adapting stride and pixel format as required.

4.2.8 Extension: Midpoint Ellipse Implementation Outline

Implementing the ellipse algorithm follows similar patterns but requires additional logic for the two regions of slope. A practical implementation involves:

- Maintaining and updating separate decision parameters for Regions 1 and 2.
- Carefully transitioning between regions when the slope condition changes.
- Plotting symmetric points accordingly.

Due to the added complexity, optimized libraries and frameworks sometimes offer ellipse drawing primitives, but understanding and implementing the midpoint ellipse algorithm remains valuable for custom software rendering engines.

4.2.9 Performance and Optimization

- Both algorithms benefit from low-level optimizations like loop unrolling and minimizing branching to improve speed on modern CPUs.
- Using SIMD instructions is possible but requires rewriting for parallel pixel updates.
- In multithreaded rendering scenarios, the circle or ellipse drawing can be partitioned spatially for parallel execution.

4.2.10 Summary

The midpoint circle and ellipse algorithms provide reliable, efficient, and integer-based methods for rasterizing these essential geometric shapes. They are particularly well-suited for CPU-only graphics rendering environments such as embedded systems, retro game engines, and software fallbacks on modern OS platforms.
Mastering these algorithms equips developers with core tools to build more advanced graphics primitives and to optimize rendering pipelines without relying on GPU acceleration.

4.3 Filling Shapes Using Scanline Rasterization

4.3.1 Introduction

Filling geometric shapes efficiently is a fundamental aspect of 2D graphics rendering, especially in CPU-only environments where performance constraints are significant. The **scanline rasterization** technique remains one of the most effective and widely used methods for filling polygons and shapes like circles and rectangles.
This section explores the principles of scanline rasterization, its application to shape filling, and practical considerations for implementing it on modern systems and programming environments, including updated APIs and tools.

4.3.2 The Concept of Scanline Rasterization

At its core, scanline rasterization operates by iterating horizontally across each row (scanline) of the pixel framebuffer and determining which pixels lie inside the shape boundaries on that scanline. This approach reduces complex 2D shape filling into a series of simpler 1D horizontal fill operations.
The process is summarized as follows:

- For each horizontal scanline (row of pixels), identify the intersections of the scanline with the shape's edges.
- Sort these intersections by their X coordinates.
- Fill pixels between pairs of intersections, which represent the interior spans on that line.

This technique exploits the coherence of adjacent pixels and scanlines to optimize rendering speed and memory access patterns.

4.3.3 Applying Scanline Rasterization to Different Shapes

1. **Rectangles**

   Filling rectangles is trivial since the shape boundaries align with horizontal and vertical axes:

   - For each scanline between the top and bottom edges, fill pixels from the left boundary to the right boundary.
   - Implementation simply involves nested loops bounded by the rectangle's coordinates.

   Though rectangles can be filled without complex intersection calculations, scanline logic is foundational for more complex polygons.

2. **Circles**

   Filling circles using scanline rasterization involves:

   - For each scanline crossing the circle's vertical bounds, calculate the left and right intersection points of the scanline with the circle's circumference.
   - These intersections can be found by solving the circle equation for x:

     $$x = \pm\sqrt{r^2 - (y - c_y)^2} + c_x$$

     where (c_x, c_y) is the circle's center and r its radius.
   - Fill pixels horizontally between these intersection points.

   This approach leverages symmetry and avoids filling outside the circle's boundary.

3. **Polygons**

   For polygons, scanline rasterization requires more involved steps:

- Build an **Edge Table (ET)** that lists all edges, sorted by their minimum Y coordinate.
- For each scanline, maintain an **Active Edge Table (AET)** of edges intersecting that line.
- Determine intersection points by linearly interpolating along edges.
- Sort intersections and fill pixels between pairs.

This method accommodates concave polygons and complex shapes, making it essential for software rendering pipelines.

4.3.4 Implementation Details and Optimizations

1. **Edge Tables and Active Edge Tables**

   Efficient polygon filling relies on data structures:

   - **Edge Table (ET):** Preprocessed at shape initialization, stores edges indexed by their minimum Y coordinate to allow quick access per scanline.
   - **Active Edge Table (AET):** Updated dynamically per scanline, contains edges currently intersecting the scanline. Edges are sorted by their X intersection coordinate.

   Updating the AET efficiently by removing edges that end and updating X values for edges continuing to the next scanline is crucial for performance.

2. **Fixed-Point Arithmetic**

   To avoid floating-point overhead in edge interpolation, fixed-point arithmetic is preferred, especially in embedded systems or when targeting CPUs without hardware floating-point units.

   Using fixed-point ensures consistent precision and faster calculations.

3. **Handling Edge Cases**

   - **Horizontal edges:** Generally ignored in edge tables to avoid double counting.
   - **Shared vertices:** Implement consistent rules (e.g., ”even-odd” or ”non-zero winding”) to determine inside/outside areas and prevent overfilling or gaps.
   - **Clipping:** Ensure shapes clipped to viewport boundaries are handled properly to avoid memory violations.

4.3.5 Practical Example: Filling a Circle with Scanline Rasterization in C++

The following simplified example demonstrates filling a circle using horizontal scanlines:

```cpp
void fillCircle(int cx, int cy, int radius, uint32_t color, uint8_t*
↪   framebuffer, int stride, int bytesPerPixel) {
    for (int y = -radius; y <= radius; ++y) {
        int scanlineY = cy + y;
        if (scanlineY < 0) continue; // Skip out-of-bounds
        if (scanlineY >= FRAMEBUFFER_HEIGHT) break;

        // Calculate horizontal span using circle equation
        int xSpan = static_cast<int>(sqrt(radius * radius - y * y));
        int xStart = cx - xSpan;
        int xEnd = cx + xSpan;

        // Clamp to framebuffer boundaries
        if (xStart < 0) xStart = 0;
        if (xEnd >= FRAMEBUFFER_WIDTH) xEnd = FRAMEBUFFER_WIDTH - 1;

        // Fill the horizontal span
        for (int x = xStart; x <= xEnd; ++x) {
            int offset = scanlineY * stride + x * bytesPerPixel;
            uint32_t* pixelPtr = reinterpret_cast<uint32_t*>(framebuffer
            ↪   + offset);
            *pixelPtr = color;
        }
    }
}
```

This method ensures that only pixels inside the circle are filled, using fast integer loops and minimal calculations per scanline.

4.3.6 Scanline Rasterization with Modern Tools and OS

In contemporary environments, scanline rasterization remains relevant when rendering software fallbacks, retro graphics, or emulators where GPU acceleration is unavailable or undesirable.

- **SDL2:** Allows direct pixel buffer access to implement scanline fills efficiently. Surfaces can be locked, manipulated, and updated.
- **Windows GDI / Direct2D (software fallback):** Scanline algorithms can be implemented on bitmaps and transferred using GDI functions.
- **Linux Framebuffer and X11:** Direct framebuffer memory access or using XShm shared memory extensions enables efficient scanline rendering.
- **Cross-platform CMake projects:** Integration of scanline rasterization with these libraries ensures portability.

4.3.7 Advanced Topics and Extensions

- **Anti-Aliasing:** Scanline methods can be extended to support anti-aliasing by computing partial pixel coverage per scanline and blending colors accordingly, although this increases complexity.
- **Multi-threaded Scanline Rendering:** Dividing scanlines across multiple threads can exploit modern multi-core CPUs for performance improvements.
- **Scanline Clipping:** Efficient clipping against viewport or window boundaries during scanline traversal improves robustness.

4.3.8 Summary

Scanline rasterization is a foundational technique for filling shapes in CPU-based graphics rendering. Its simplicity, efficiency, and adaptability to various shapes make it essential knowledge for programmers working in systems programming, embedded graphics, and software rendering engines.

By understanding and implementing scanline rasterization effectively, developers can create high-performance graphics applications even in constrained environments without GPU support.

4.4 Anti-Aliasing on the CPU

4.4.1 Introduction

Anti-aliasing is a crucial technique in computer graphics aimed at reducing the visual artifacts known as aliasing — jagged edges or "staircase" effects — that appear when rendering geometric shapes at discrete pixel resolutions. On GPU-based systems, hardware and shader pipelines often provide built-in anti-aliasing support. However, in software rendering environments relying solely on the CPU, implementing efficient and visually pleasing anti-aliasing methods remains an essential skill.

This section delves into the principles, algorithms, and practical implementations of anti-aliasing techniques on the CPU. It also considers modern performance optimization strategies and tool support relevant to software rasterizers, emulators, and embedded graphics systems developed after 2019.

4.4.2 Understanding Aliasing and Its Causes

Aliasing occurs because digital displays approximate continuous shapes with a finite grid of pixels. Edges that are not perfectly aligned with the pixel grid create discontinuities in pixel coverage, resulting in jagged outlines.

Key causes of aliasing include:

- Insufficient spatial sampling of geometric edges.
- Hard transitions between foreground and background pixel colors.
- Quantization of continuous coordinates to discrete pixel indices.

Effective anti-aliasing smooths these transitions by introducing intermediate pixel intensities or colors that visually blend edges.

4.4.3 CPU-Based Anti-Aliasing Techniques

Several anti-aliasing approaches are feasible on the CPU. They can be broadly categorized into:

- **Supersampling (SSAA)**
- **Multisample Anti-Aliasing (MSAA)**
- **Analytical methods (e.g., Xiaolin Wu's algorithms)**
- **Post-processing filters**

Each method offers trade-offs between visual quality, computational cost, and implementation complexity.

4.4.4 Supersampling Anti-Aliasing (SSAA)

Concept:
Supersampling involves rendering the scene or shape at a higher resolution than the target display and then downsampling the image to the final resolution. This oversampling smooths edges by averaging multiple sub-pixel samples per output pixel.
Implementation considerations on the CPU:

- Requires significant memory and processing power since multiple samples per pixel are calculated.
- Modern multi-core CPUs and SIMD instructions (such as AVX2 and AVX-512) available in processors since 2019 can accelerate sample computations and downsampling.
- Integration with libraries such as SDL2 enables managing higher resolution surfaces and scaling down using optimized blitting routines.

Example:
Render a shape at 4x resolution (2x width and height), then average each 4x4 block of samples into a single pixel.

4.4.5 Multisample Anti-Aliasing (MSAA)

Concept:
MSAA optimizes supersampling by sampling multiple locations only along polygon edges instead of the entire scene. The interior pixels are sampled once, reducing the workload.
CPU challenges:

- MSAA requires edge detection and partial pixel coverage determination.
- Efficient data structures like coverage masks and sample buffers must be implemented manually.
- Modern CPUs with vectorized instruction sets can help parallelize sample evaluations.

While MSAA is less commonly used in pure CPU renderers due to complexity, it remains feasible for specialized software engines and emulators that emulate GPU functionality.

4.4.6 Analytical Anti-Aliasing: Xiaolin Wu's Algorithm

Xiaolin Wu's algorithm is a classic and widely appreciated method for anti-aliased line drawing on the CPU.
Key features:

- Uses intensity blending based on pixel coverage calculated analytically.

- Provides visually smooth lines with minimal computational overhead.
- Well-suited for low-level graphics programming without requiring high memory or complex buffers.

Implementation notes:

- The algorithm calculates fractional coverage for pixels adjacent to the ideal line path.
- Pixel intensity is adjusted proportionally to coverage using alpha blending.
- Often implemented with 8-bit or 32-bit color buffers, requiring support for alpha blending in software.

Since 2019, SIMD intrinsics and CPU instruction set enhancements have enabled faster implementations of Wu's algorithm on modern CPUs.

4.4.7 Post-Processing Filters for Anti-Aliasing

In software renderers, applying filters after drawing can smooth edges by blurring or blending neighboring pixels.
Common filters:

- Gaussian blur: smooths edges but can reduce overall sharpness.
- Bilateral filtering: preserves edges better but is computationally more intensive.
- FXAA-like algorithms: fast approximate anti-aliasing implemented as post-processing.

Implementation on CPU:

- Requires efficient neighborhood sampling and convolution.
- Parallelizable using threads or SIMD instructions.
- Can be integrated into rendering pipelines using libraries like OpenCV or custom implementations.

4.4.8 Practical Implementation: Anti-Aliased Line Drawing Using Wu's Algorithm

A simplified outline for implementing Xiaolin Wu's line algorithm in C++ includes:

- Determining the line slope and direction.
- Iterating over the major axis.
- Calculating pixel intensities for adjacent pixels based on distance to the ideal line.
- Blending pixel colors in the framebuffer accordingly.

4.4.9 Performance Optimization Strategies

Given the increased computational load of anti-aliasing, especially on CPUs, optimization is critical:

- Use **fixed-point arithmetic** where possible to reduce floating-point overhead.
- Exploit **SIMD vectorization** with modern instruction sets (AVX2, AVX-512) available in CPUs.
- Implement **multithreading** to parallelize scanline or shape processing.
- Cache alignment and prefetching to minimize memory latency during pixel writes.

- Use **hardware-accelerated libraries** (e.g., SIMD-enabled image processing libraries) to offload routine tasks.

4.4.10 Tool Support and Modern Context

After 2019, numerous tools and APIs facilitate CPU-based anti-aliasing development:

- **SDL2:** Supports pixel format conversions and surface manipulations, enabling efficient buffer updates with alpha blending for anti-aliasing.
- **Direct2D (software fallback mode):** Provides CPU-based rendering paths with built-in anti-aliasing algorithms when hardware acceleration is unavailable.
- **Linux framebuffer and X11:** Custom anti-aliasing can be implemented by directly manipulating pixel buffers with optimized CPU code.
- **CMake-based cross-platform projects:** Ease building and integrating software rasterizers with anti-aliasing across Windows and Linux platforms.

4.4.11 Summary

Anti-aliasing on the CPU remains a vital technique for software-only graphics rendering. Through approaches such as supersampling, analytical algorithms like Xiaolin Wu's, and post-processing filters, developers can achieve visually smooth edges despite the absence of GPU acceleration.

Modern CPU features and software tools available after 2019 have made it feasible to implement efficient and high-quality anti-aliasing in portable and embedded systems, emulators, and retro-style software renderers.

Mastering these techniques enables programmers to deliver compelling graphical experiences in constrained environments while maintaining control over performance and visual fidelity.

Chapter 5

Bitmaps and Images

5.1 Loading and Displaying BMP, PNG Manually (Without Using SDL_image or stb_image)

5.1.1 Introduction

Loading and displaying bitmap images such as BMP and PNG formats without relying on third-party image loading libraries is a foundational skill in graphics programming, especially in contexts where minimizing dependencies is crucial. This section explains the detailed procedures for manually parsing and rendering BMP and PNG files on the CPU, emphasizing understanding the file format specifications, memory layout, and pixel data extraction.

By avoiding popular libraries like SDL_image or stb_image, programmers gain deeper control and flexibility, which is essential in embedded systems, custom emulators, or highly optimized CPU-only rendering engines. Examples and explanations here incorporate modern CPU and OS environments, reflecting developments after 2019.

5.1.2 Overview of BMP and PNG File Formats

- **BMP (Bitmap Image File):**
  An uncompressed or lightly compressed raster image format primarily used in Windows environments. BMP files store pixel data in a straightforward manner with headers describing image dimensions, color depth, and compression methods (often none).

- **PNG (Portable Network Graphics):**
  A lossless compressed image format widely used on the web and modern applications. PNG supports alpha transparency, multiple color types, and employs the DEFLATE compression algorithm for pixel data.

5.1.3 BMP File Structure and Manual Parsing

File Header (14 bytes):

- Signature: 2 bytes (”BM”) identifying the file as BMP.
- File size: 4 bytes.
- Reserved: 4 bytes.
- Offset: 4 bytes indicating the pixel data start position.

DIB Header (Variable size, commonly 40 bytes):

- Header size.
- Image width and height.
- Color planes.

- Bits per pixel (bpp).
- Compression type.
- Image size.
- Resolution and color palette information.

Pixel Data:

- Located at the offset specified in the file header.
- Typically stored bottom-up (the first pixel corresponds to the bottom-left).
- Row sizes are aligned to 4-byte boundaries (padding may be present).
- Pixel formats vary (commonly 24-bit RGB or 32-bit ARGB).

5.1.4 Reading BMP Manually in C/C++

1. **Open the file in binary mode.**
2. **Read and validate the file header:**
   Confirm the signature is "BM."
3. **Read the DIB header:**
   Extract image width, height, bpp, and compression method.
4. **Calculate row size including padding:**
   Row size = ((bits_per_pixel * width + 31) / 32) * 4.
5. **Seek to pixel data offset.**
6. **Read pixel data row-by-row:**
   Because BMP rows are bottom-up, read pixels from bottom row to top.

7. **Store pixel data in an internal buffer with a consistent format** such as 32-bit RGBA for easier manipulation.

5.1.5 Displaying BMP on the CPU

- Use the pixel buffer to draw pixels directly to the framebuffer or memory surface.
- Handle pixel format conversion if necessary (e.g., BMP BGR to framebuffer RGB).
- Consider endianness and alignment.
- For Windows, GDI or Direct2D software fallback can be used to blit the pixel buffer.
- For Linux, raw framebuffer access or X11/XCB libraries can display the buffer.

5.1.6 PNG File Structure and Parsing Challenges

PNG files have a more complex structure due to compression and chunk-based layout:

- **Signature:** 8 bytes fixed header.
- **Chunks:**
  Sequence of chunks each with length, type, data, and CRC. Critical chunks include:
  - IHDR (header): image width, height, bit depth, color type.
  - PLTE (palette): optional palette data.
  - IDAT (image data): compressed pixel data.
  - IEND: end of file.

- **Compression:**
  IDAT chunks contain data compressed using the DEFLATE algorithm.

- **Filtering:**
  Each scanline is filtered using one of five filter types (None, Sub, Up, Average, Paeth) to improve compression.

5.1.7 Manually Decoding PNG on the CPU

Step 1: Read and validate the PNG signature.

Step 2: Parse chunks sequentially:

- Extract IHDR data: dimensions, bit depth, color type.
- Concatenate IDAT chunks to reconstruct compressed data.

Step 3: Decompress pixel data using a DEFLATE implementation:

- Implement or integrate a software DEFLATE decoder, e.g., zlib (without high-level image decoding libraries).
- Recent CPU enhancements, including SIMD optimizations in zlib-ng , can accelerate decompression.

Step 4: Reverse filtering for each scanline:

- Apply filter algorithms based on filter byte per line.
- Use previously decoded pixels and neighboring pixels for reconstruction.

Step 5: Convert pixel data to a usable format:

- Depending on color type (grayscale, RGB, RGBA, indexed), translate pixels into a consistent format such as 32-bit RGBA.
- Handle alpha channel blending if present.

5.1.8 Displaying PNG Images on the CPU

- After decoding, store pixels in an appropriate buffer.
- Convert color format if needed to match the target framebuffer.
- Blit pixels to screen using platform-specific methods as described in BMP section.
- Optimize display by minimizing memory copies and using efficient pixel transfer methods (e.g., WriteProcessMemory on Windows or mmap-based framebuffer on Linux).

5.1.9 Practical Considerations and Modern Tooling

- **Cross-platform file I/O and byte manipulation:**
  Use C++17 or later standards to leverage filesystem and byte utilities.
- **Endian-safe reading:**
  Implement helper functions for reading multi-byte integers in big and little endian formats, critical for PNG chunk parsing.
- **Memory management:**
  Use smart pointers and aligned buffers to improve safety and performance.
- **Performance:**
  Utilize modern CPU features such as SIMD instructions and multithreading to accelerate decompression and pixel conversion.
- **Testing and Validation:**
  Use known test BMP and PNG files and compare output with standard viewers.

5.1.10 Summary

Manually loading and displaying BMP and PNG files requires a detailed understanding of file formats, binary parsing, compression, and pixel data handling. While labor-intensive compared to using third-party libraries, this approach provides valuable insight and fine control over image processing, essential in environments where dependency control, optimization, or customization are priorities.

Developers working on CPU-only graphics systems benefit greatly from mastering these techniques, especially with modern CPU advancements and OS APIs facilitating efficient buffer management and display.

5.2 Writing Your Own Image Blitter

5.2.1 Introduction

Blitting—short for **block image transfer**—is a fundamental graphics operation involving copying pixel data from a source image buffer into a destination framebuffer or surface. Writing an efficient, flexible image blitter using only the CPU remains critical in environments without GPU acceleration or where custom rendering pipelines are required, such as embedded systems, retro game engines, or emulator software. This section provides an in-depth guide to designing and implementing a CPU-based image blitter. It covers essential concepts including pixel format compatibility, memory layout, performance optimizations, and platform-specific considerations, with modern CPU and OS contexts from 2019 onward.

5.2.2 Fundamentals of Image Blitting

At its core, blitting involves these steps:

- Reading pixels from a **source buffer** (the image).
- Writing those pixels into a **destination buffer** (the framebuffer or render target).
- Managing **positioning** (where to draw in the destination).
- Handling **pixel format conversions** if source and destination formats differ.
- Optionally applying **transparency** or other per-pixel effects.
- Ensuring **memory safety** and **performance**.

5.2.3 Defining Source and Destination Buffers

Source Buffer:

- Contains the image pixel data, often stored in a contiguous memory block.
- Common pixel formats: 24-bit RGB, 32-bit ARGB/RGBA, 8-bit grayscale, or indexed palettes.
- May be compressed or decoded before blitting.

Destination Buffer:

- Usually the framebuffer representing the screen or window surface.
- Format and pitch (bytes per row) vary by OS and API (e.g., GDI surfaces on Windows, X11 drawables on Linux).
- Access may be direct (raw memory) or via platform APIs.

5.2.4 Handling Pixel Formats and Conversion

One of the major challenges in writing a blitter is **pixel format compatibility**. The source and destination buffers often use different layouts:

- **Byte order:** RGB vs. BGR.
- **Bit depth:** 24-bit vs. 32-bit (with or without alpha).
- **Alpha channel presence and blending.**

A robust blitter must detect these differences and convert pixels accordingly. For example:

- If source is 24-bit RGB and destination is 32-bit ARGB, the blitter inserts an opaque alpha byte (0xFF) per pixel.
- If alpha blending is required, the blitter must combine source and destination pixels using the alpha channel (see Section 5.2.8).

Modern CPUs with SIMD extensions (e.g., SSE2, AVX2) can accelerate these conversions by processing multiple pixels simultaneously, which is essential for real-time performance.

5.2.5 Coordinate and Clipping Management

The blitter must correctly place the source image pixels within the destination buffer at specified coordinates `(dst_x, dst_y)`.

- When the source image extends beyond the destination boundaries, **clipping** must be performed to avoid memory access violations.
- Clipping calculations involve:
  - Adjusting the source pixel start index.
  - Adjusting the number of pixels to copy per row.
  - Skipping rows outside the visible area.

Proper clipping ensures the blitter operates safely without requiring the caller to pre-process coordinates.

5.2.6 Implementing the Core Blitting Loop

A basic blitter consists of nested loops:

```
for (int y = 0; y < height; ++y) {
    for (int x = 0; x < width; ++x) {
        // Calculate source pixel address
        // Calculate destination pixel address
        // Convert pixel if needed
        // Copy or blend pixel to destination
    }
}
```

Key implementation details:

- **Pitch (stride):**
  Both source and destination rows may include padding bytes. Calculate row addresses using pitch, not just width * bytes per pixel.

- **Pointer arithmetic:**
  Efficiently increment pointers rather than recalculating indices.

- **Avoid branches inside inner loops:**
  Minimize if-statements in the hot path to improve CPU pipeline efficiency.

5.2.7 Performance Optimizations

Several optimizations improve CPU blitter performance:

- **Memory alignment:**
  Align source and destination buffers to 16- or 32-byte boundaries to leverage SIMD instructions.

- **SIMD vectorization:**
  Use intrinsics (SSE2, AVX2) to process 4–8 pixels in parallel.

- **Loop unrolling:**
  Unroll inner loops to reduce branch overhead.
- **Multithreading:**
  Divide the image into horizontal strips and blit in parallel threads.
- **Cache locality:**
  Access memory sequentially to reduce cache misses.
- **Write combining:**
  Minimize write latency by buffering output before committing to memory.

5.2.8 Supporting Transparency and Alpha Blending

Basic blitters overwrite destination pixels unconditionally. Advanced blitters incorporate alpha blending to support transparency.
Alpha blending formula:

```
dst = src_alpha * src_color + (1 - src_alpha) * dst_color
```

Steps to implement:

- Extract source alpha channel.
- Convert alpha to a 0.0–1.0 floating scale or use integer math with fixed-point.
- Blend each color channel accordingly.
- Store result back to destination buffer.

This requires additional computations and may benefit from SIMD optimizations. It is vital for rendering PNG images with transparency or UI elements.

5.2.9 Platform-Specific Considerations

- **Windows (GDI, Direct2D Software Fallback):**
  Use `GetDIBits`/`SetDIBits` for bitmap data access or lock surfaces via Direct2D APIs to obtain raw memory pointers for blitting.

- **Linux (X11, framebuffer):**
  Use `XShmPutImage` or direct framebuffer memory mapping (`/dev/fb0`) for efficient pixel buffer updates.

- **Cross-platform frameworks (SDL2):**
  While SDL offers hardware-accelerated blitting, a CPU-only blitter operates on `SDL_Surface` pixel data directly.

5.2.10 Example: Simple 32-bit ARGB Blitter in C++

```cpp
void BlitARGB32(
    uint32_t* dst, int dst_width, int dst_height, int dst_pitch,
    const uint32_t* src, int src_width, int src_height, int src_pitch,
    int dst_x, int dst_y)
{
    // Clipping
    int blit_width = src_width;
    int blit_height = src_height;
    if (dst_x < 0) { src += -dst_x; blit_width += dst_x; dst_x = 0; }
    if (dst_y < 0) { src += (-dst_y) * src_pitch; blit_height += dst_y;
    ↪  dst_y = 0; }
    if (dst_x + blit_width > dst_width) blit_width = dst_width - dst_x;
    if (dst_y + blit_height > dst_height) blit_height = dst_height -
    ↪  dst_y;
    for (int y = 0; y < blit_height; ++y) {
        uint32_t* dst_row = dst + (dst_y + y) * dst_pitch + dst_x;
        const uint32_t* src_row = src + y * src_pitch;
        for (int x = 0; x < blit_width; ++x) {
            dst_row[x] = src_row[x];  // Simple copy; extend for blending
        }
    }
}
```

This simple blitter copies pixels row-by-row with boundary clipping. Real implementations expand this for blending and format conversion.

5.2.11 Summary

Writing a custom image blitter on the CPU requires meticulous handling of pixel formats, memory layouts, coordinate transformations, and performance considerations. Modern CPUs offer powerful instruction sets that, when leveraged, can make CPU-based blitting performant enough for many real-time applications without GPU assistance.

Understanding the underlying principles of blitting not only provides a strong foundation for graphics programming but also opens doors to advanced custom rendering techniques tailored for diverse environments.

5.3 Scaling and Rotating Images Using the CPU

5.3.1 Introduction

Scaling and rotating images are fundamental operations in graphics programming. When working without GPU acceleration, these transformations must be implemented efficiently on the CPU. Unlike simple blitting, these operations require per-pixel geometric manipulation, interpolation, and careful memory handling to maintain image quality and performance.

This section explores CPU-based algorithms and techniques for scaling and rotating images, with a focus on software rendering approaches relevant to modern systems and APIs. We discuss common interpolation methods, coordinate transformations, optimization strategies, and practical implementation details.

5.3.2 Image Scaling on the CPU

1. **Overview**

   Scaling changes the dimensions of an image by enlarging or reducing it along horizontal and vertical axes. Scaling can be uniform (same factor in both directions) or non-uniform (different horizontal and vertical factors).

   Because the destination image size often differs from the source, simple pixel copying is insufficient. Instead, pixels in the destination correspond to interpolated positions in the source.

2. **Basic Scaling Techniques**

   Nearest Neighbor Interpolation

   - The simplest scaling method.

- For each pixel in the output, map its coordinates back to the nearest pixel in the source.
- Fast but results in blocky images, especially when upscaling.

Bilinear Interpolation

- Considers the four nearest pixels surrounding the mapped source coordinate.
- Computes a weighted average based on fractional positions.
- Produces smoother images but requires more computations.

Bicubic and Higher-Order Interpolations

- Use more neighboring pixels (e.g., 16 for bicubic).
- Produce higher quality but are more computationally expensive.
- Less common in real-time CPU rendering due to complexity.

3. **Coordinate Mapping**

   To scale an image, define scale factors:

   $$scale_x = \frac{src\_width}{dst\_width}, \quad scale_y = \frac{src\_height}{dst\_height}$$

   For each pixel (x_d, y_d) in the destination image, compute the corresponding source coordinates:

   $$x_s = x_d \times scale_x, \quad y_s = y_d \times scale_y$$

   This floating-point source coordinate is used in interpolation.

4. **Implementation Example: Bilinear Scaling**

   (a) **For each destination pixel:**

- Compute source coordinates (x_s, y_s).

(b) Identify the integer parts and fractional parts:

$$x_0 = \lfloor x_s \rfloor, \quad y_0 = \lfloor y_s \rfloor$$

$$dx = x_s - x_0, \quad dy = y_s - y_0$$

(c) Fetch the four neighboring pixels at

$$(x_0, y_0), \quad (x_0 + 1, y_0), \quad (x_0, y_0 + 1), \quad (x_0 + 1, y_0 + 1).$$

(d) Perform bilinear interpolation for each color channel separately:

$$p = (1 - dx)(1 - dy) \cdot p_{00} + dx(1 - dy) \cdot p_{10} + (1 - dx)dy \cdot p_{01} + dx\,dy \cdot p_{11}$$

5. **Optimizations for Scaling**

- **Fixed-Point Arithmetic:**
  Replace floating-point with fixed-point calculations to improve performance on CPUs lacking fast FPUs.
- **Precomputed Mapping:**
  Precompute source coordinate mappings for each destination pixel once, avoiding repeated calculations.
- **SIMD Vectorization:**
  Process multiple pixels simultaneously using SSE/AVX instructions.
- **Multi-threading:**
  Partition the destination image rows for parallel processing.

5.3.3 Image Rotation on the CPU

1. **Overview**

   Rotation involves pivoting an image around a center point by an angle θ. Like scaling, rotation requires mapping destination pixels back to source coordinates and performing interpolation to fill pixel values correctly.

2. **Rotation Transformation**

   For an image rotated around its center (c_x, c_y), the transformation from destination coordinates (x_d, y_d) to source coordinates (x_s, y_s) is given by

$$\begin{bmatrix} x_s \\ y_s \end{bmatrix} = \begin{bmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x_d - c_x \\ y_d - c_y \end{bmatrix} + \begin{bmatrix} c_x \\ c_y \end{bmatrix}$$

   where θ is the rotation angle in radians.

3. **Implementing Rotation**

   (a) **Determine Output Dimensions:**
   Since rotation can increase bounding box size, compute the minimum bounding rectangle to contain the rotated image.

   (b) **Inverse Mapping:**
   For each destination pixel, compute the corresponding source pixel using the inverse rotation matrix.

   (c) **Interpolation:**
   Use bilinear or nearest neighbor interpolation on the source coordinate.

   (d) **Boundary Checks:**

Pixels mapping outside the source image bounds are typically set transparent or background color.

4. **Performance Considerations**

   Rotation is computationally intensive due to per-pixel trigonometric operations and interpolation.

   - **Precompute Sine and Cosine:**
     Cache $\sin \theta$ and $\cos \theta$ to avoid repeated evaluations.
   - **Use Lookup Tables:**
     For repeated rotations by fixed angles, precompute coordinate mappings.
   - **Fixed-Point and Approximate Math:**
     Use fixed-point arithmetic and approximate trigonometric functions for embedded systems.
   - **SIMD and Multi-threading:**
     Vectorize interpolation and divide the output image for parallel rendering.

5.3.4 Combined Scaling and Rotation

Combined transformations are common, e.g., rotating and scaling simultaneously. These can be expressed using affine transformation matrices:

$$\mathbf{M} = \begin{bmatrix} s_x \cos\theta & -s_y \sin\theta & t_x \\ s_x \sin\theta & s_y \cos\theta & t_y \\ 0 & 0 & 1 \end{bmatrix}$$

Where s_x, s_y are scale factors and t_x, t_y are translation components.
For CPU implementation:

- Precompute the inverse of the transformation matrix.
- For each destination pixel, apply inverse transform to find source coordinate.
- Perform interpolation accordingly.

5.3.5 Handling Pixel Formats and Alpha

- When scaling or rotating images with alpha channels, interpolation must blend alpha correctly.
- Premultiplied alpha formats simplify blending during interpolation.
- Converting images to premultiplied alpha before transformations and back after improves visual quality and reduces artifacts.

5.3.6 Practical Examples and Tools

- **C/C++:**
  Implementations typically use direct pixel buffer access for efficiency.
- **SDL2:**
  While SDL2 provides hardware-accelerated rendering, software surfaces can be manipulated manually for custom transformations.
- **Third-party Libraries:**
  Libraries such as **libpng** or **stb_image** are often used for loading images, but transformation code is commonly custom or minimal to avoid GPU dependency.
- **Modern CPUs :**
  Leverage AVX2/AVX-512 SIMD instructions where available.

- **Cross-Platform:**
  Ensure careful handling of pixel formats and memory alignment when running on Windows, Linux, or embedded OSes.

5.3.7 Summary

Scaling and rotating images purely on the CPU require careful application of coordinate transformations, interpolation, and memory management. Although more computationally expensive than GPU-based methods, modern CPU architectures and careful optimization allow these operations to be performed effectively in contexts where GPU use is unavailable or undesired. Mastery of these techniques empowers developers working on emulators, embedded graphics, and retro-style software renderers to deliver visually rich graphics without hardware acceleration.

5.4 Handling Alpha Blending and Transparency

5.4.1 Introduction

Alpha blending and transparency are essential concepts in modern graphics programming, enabling the creation of images and scenes with smooth edges, shadows, overlays, and complex compositing effects. When rendering exclusively on the CPU—without GPU acceleration—handling alpha blending efficiently and correctly is critical for achieving visual quality and performance.

This section explores the principles, algorithms, and practical implementation of alpha blending and transparency for CPU-only software rendering. We emphasize up-to-date practices using contemporary tools and environments, considering common pixel formats, memory layouts, and optimizations relevant after 2019.

5.4.2 The Alpha Channel and Transparency Basics

The alpha channel is an additional per-pixel component that represents pixel opacity or transparency. It typically ranges from 0 (fully transparent) to 255 (fully opaque) when using 8-bit alpha precision.

- **Fully Opaque Pixel:** Alpha = 255
- **Fully Transparent Pixel:** Alpha = 0

Alpha allows pixels to blend with background pixels, enabling effects like translucent overlays and anti-aliased edges.

5.4.3 Alpha Blending Fundamentals

Alpha blending computes the resulting pixel color when overlaying a foreground (source) pixel over a background (destination) pixel, considering their alpha values.

The standard alpha blending formula for each color channel (red, green, blue) is:

$$C_{\text{out}} = C_{\text{src}} \times \alpha_{\text{src}} + C_{\text{dst}} \times (1 - \alpha_{\text{src}})$$

Where:

- C_{out} is the resulting channel color.
- C_{src} and C_{dst} are the source and destination channel values, normalized between 0 and 1.
- α_{src} is the source pixel alpha, normalized between 0 and 1.

If both source and destination have alpha channels, the resulting alpha is:

$$\alpha_{\text{out}} = \alpha_{\text{src}} + \alpha_{\text{dst}} \times (1 - \alpha_{\text{src}})$$

5.4.4 Premultiplied vs. Straight Alpha

There are two main representations of alpha data in pixel formats:

- **Straight (Non-premultiplied) Alpha:**
  Color channels store their original values independently of alpha. The blending formula applies multiplication by alpha at blending time.
- **Premultiplied Alpha:**
  Color channels are pre-scaled by the alpha value. For example, a red channel value is stored as $R \times \alpha$. This simplifies blending:

$$C_{\text{out}} = C_{\text{src}} + C_{\text{dst}} \times (1 - \alpha_{\text{src}})$$

Advantages of Premultiplied Alpha:

- Avoids artifacts around edges (fringes or halos).
- More efficient to compute blending since multiplication is done once during image creation or loading.

Handling Premultiplied Alpha in Software:
If working with non-premultiplied images (e.g., standard PNGs), convert them to premultiplied form upon loading before performing blending.

5.4.5 Implementing Alpha Blending on the CPU

1. **Basic Blending Loop**

   Given a source pixel $S = (R_s, G_s, B_s, A_s)$ and a destination pixel $D = (R_d, G_d, B_d, A_d)$, implement per-channel blending:

   ```
   float alpha = A_s / 255.0f;
   R_out = (uint8_t)(R_s * alpha + R_d * (1.0f - alpha));
   G_out = (uint8_t)(G_s * alpha + G_d * (1.0f - alpha));
   B_out = (uint8_t)(B_s * alpha + B_d * (1.0f - alpha));
   A_out = (uint8_t)(A_s + A_d * (1.0f - alpha)); // Optional: output
    ↪  alpha channel
   ```

   This operation is repeated for every pixel in the source image being composited onto the destination framebuffer.

2. **Fixed-Point and Integer Arithmetic Optimizations**

   Floating-point blending is clear but may be slower on some CPUs or in large loops. Fixed-point or integer arithmetic can optimize performance:

   - Use integer math with 8-bit alpha scaled between 0 and 255.

- Multiply channels and alpha as integers, then divide by 255 using bit-shifts or fast division approximations.

Example integer blending formula:

$$C_{out} = \frac{C_{src} \times A_{src} + C_{dst} \times (255 - A_{src})}{255}$$

Implementation in C/C++:

```
uint16_t alpha = A_s;
uint16_t inv_alpha = 255 - alpha;
R_out = (uint8_t)((R_s * alpha + R_d * inv_alpha + 127) / 255);
G_out = (uint8_t)((G_s * alpha + G_d * inv_alpha + 127) / 255);
B_out = (uint8_t)((B_s * alpha + B_d * inv_alpha + 127) / 255);
```

Adding 127 performs rounding to nearest integer.

3. **Handling Different Pixel Formats**

   CPU-only graphics applications often encounter multiple pixel formats (ARGB, RGBA, BGRA). It is important to:

   - Identify pixel channel ordering to read and write correct components.
   - Consider endianness when interpreting multi-byte pixels.
   - Convert formats as necessary during loading or prior to blending to maintain consistency.

5.4.6 Transparency in Bitmaps and Formats

- **BMP:** Typically does not support alpha by default but can be extended (e.g., 32-bit BMP with alpha channel).

- **PNG:** Supports full 8-bit alpha channel per pixel, commonly used for transparent images.
- **Other formats:** May have indexed transparency or single-color keys, which require separate handling.

5.4.7 Performance Considerations and Modern CPU Features

- **SIMD Acceleration:**
  Use SIMD instructions (SSE, AVX2) to blend multiple pixels concurrently, significantly speeding up alpha blending on modern CPUs (processors).
- **Multi-threading:**
  Divide the framebuffer into segments for concurrent blending, leveraging multi-core CPUs.
- **Memory Access Patterns:**
  Optimize cache usage by processing pixels in linear memory order.
- **Software Libraries:**
  While this book avoids third-party image blitting libraries like SDL_image or stb_image for blending, modern projects may still use such libraries for loading, then apply custom alpha blending optimized for their needs.

5.4.8 Advanced Topics: Porter-Duff Compositing

Alpha blending is a subset of compositing operations defined by Porter and Duff. For advanced graphics effects, consider:

- **Over, In, Out, Atop** compositing modes.
- Implementing these requires customized blending formulas.

- Useful for layered image effects, masking, and UI rendering.

5.4.9 Summary

Handling alpha blending and transparency on the CPU is fundamental to modern software rendering. Understanding premultiplied alpha, optimizing arithmetic for speed, and managing pixel formats are key to achieving both performance and visual quality. With current CPU capabilities and careful coding, high-quality alpha blending can be realized without GPU assistance, enabling complex compositing in embedded systems, retro-style engines, and cross-platform software renderers.

Chapter 6

Text Rendering on the CPU

6.1 Bitmap Fonts vs. Vector Fonts

6.1.1 Introduction

Text rendering is a fundamental component of graphical user interfaces, games, emulators, and embedded systems. In CPU-only graphics programming, choosing an appropriate font rendering technique is crucial for balancing visual quality, performance, memory footprint, and development complexity.

This section provides a detailed comparison between bitmap fonts and vector fonts, highlighting their characteristics, advantages, disadvantages, and typical use cases. We will also explore modern developments and practical considerations for implementing text rendering on CPU-only environments, referencing relevant tools, operating systems, and libraries updated or widely used since 2019.

6.1.2 Bitmap Fonts

1. **Definition and Structure**

   Bitmap fonts represent each glyph (character) as a pre-rendered pixel map. Essentially, each character is stored as an array of pixels at fixed sizes and resolutions.

   - Glyphs are stored as monochrome or color bitmaps.
   - Multiple sizes require separate bitmaps for each size (e.g., 12pt, 14pt, 16pt).
   - Pixel-perfect representation—no computation needed at runtime to rasterize.

2. **Advantages of Bitmap Fonts**

   - **Performance:**
     Since glyphs are pre-rasterized, rendering is a simple memory copy or blitting operation, which is extremely fast on CPUs.
   - **Simplicity:**
     Implementation requires minimal computation, no need for complex rasterization or hinting algorithms.
   - **Consistency:**
     Output matches the bitmap exactly, ensuring consistent appearance across different systems.
   - **Low CPU Overhead:**
     Ideal for embedded systems or low-power CPUs where complex computations are costly.

3. **Disadvantages of Bitmap Fonts**

- **Lack of Scalability:**
  Bitmap fonts do not scale well; enlarging them beyond their designed size leads to pixelation and visual artifacts.
- **Memory Usage:**
  Storing multiple sizes or styles requires significant memory.
- **Limited Flexibility:**
  Changing font size or style dynamically requires pre-generated bitmaps or complex bitmap manipulation.
- **Poor Quality on High-DPI Displays:**
  Fixed resolution makes bitmap fonts less suitable for modern high-DPI or variable-resolution screens.

4. **Common Formats and Usage**

   - Legacy systems and DOS-era software relied heavily on bitmap fonts.
   - Embedded devices with constrained resources often use bitmap fonts.
   - Bitmap fonts are still used in modern projects needing ultra-fast rendering with minimal CPU usage.
   - Tools like the Windows GDI provide access to bitmap font resources, though increasingly deprecated.

6.1.3 Vector Fonts

1. **Definition and Structure**

   Vector fonts define glyphs using mathematical curves and lines, typically Bézier curves. Common vector font formats include TrueType (TTF) and OpenType (OTF).

- Glyph outlines are described as scalable paths.
- Rendering requires rasterization: converting vector outlines to pixels at the desired size and resolution.

2. **Advantages of Vector Fonts**

   - **Scalability:**
     Vector fonts can be rendered at arbitrary sizes without loss of quality or pixelation.
   - **Compact Storage:**
     A single vector font file supports multiple sizes and styles, reducing storage requirements.
   - **High Quality on Modern Displays:**
     Support for hinting and anti-aliasing allows clean, sharp text even on high-DPI screens.
   - **Flexibility:**
     Supports complex glyph shapes, ligatures, and international character sets.

3. **Disadvantages of Vector Fonts**

   - **Computational Complexity:**
     Rasterizing vector fonts is CPU-intensive compared to bitmap fonts, involving curve evaluation, scan conversion, and anti-aliasing.
   - **Implementation Complexity:**
     Requires sophisticated algorithms for curve flattening, hinting, and pixel coverage calculation.
   - **Performance Constraints on Low-Power CPUs:**

Real-time rendering may be challenging on constrained embedded platforms without GPU assistance.

4. **Modern Software Rasterizers and Libraries**

   - Software rasterizers such as FreeType provide well-optimized vector font rendering on CPUs.
   - FreeType (updated continuously) supports a wide range of font formats and features, including subpixel rendering and hinting.
   - Many projects integrate FreeType or similar libraries for vector font support in CPU-only environments.
   - Some lightweight or embedded solutions use simplified rasterizers or pre-rasterize glyphs at runtime to trade off between quality and speed.

6.1.4 Practical Considerations for CPU-Only Graphics Programming

1. **When to Use Bitmap Fonts**

   - Applications with strict performance requirements on limited hardware.
   - Systems where memory usage is less critical than CPU load.
   - Fixed-resolution displays or legacy hardware.
   - Situations where font size variation is minimal or non-existent.

2. **When to Use Vector Fonts**

   - Applications requiring dynamic font sizes or styles.
   - High-DPI displays or user interfaces with scalable elements.

- Systems with sufficient CPU resources to handle rasterization.
- Projects needing wide language support or complex typography.

3. **Hybrid Approaches**

    - Pre-rasterize vector fonts into bitmap caches at runtime for frequently used sizes.
    - Use bitmap fonts for small UI elements and vector fonts for scalable text.
    - Employ caching strategies to balance performance and flexibility.

6.1.5 Integration with Common Tools and OS APIs

- On **Windows**, GDI historically favored bitmap fonts but supports vector font rendering through DirectWrite since Windows 7, which can be accessed for CPU-only software rendering with appropriate fallbacks.
- On **Linux**, fontconfig and FreeType provide vector font loading and rasterization; X11 or Wayland compositors can display rendered bitmaps.
- SDL provides basic text rendering through SDL_ttf (based on FreeType), though in this book's context, manual integration and understanding of font rendering are emphasized.
- Embedded systems often use custom bitmap font formats optimized for limited resources.

6.1.6 Summary

Bitmap and vector fonts represent two fundamentally different approaches to text rendering on the CPU. Bitmap fonts offer speed and simplicity at the cost of scalability and flexibility, while vector fonts provide quality and adaptability with increased computational demands. Understanding their trade-offs and how to leverage them effectively is essential for systems programmers, emulator developers, and embedded engineers working in CPU-only graphics environments.

Subsequent sections will explore practical implementation techniques for both font types, including bitmap font loading, glyph caching, vector font rasterization algorithms, and performance optimizations tailored to modern CPU architectures and software environments.

6.2 Using FreeType for CPU-Based Font Rasterization

6.2.1 Introduction to FreeType

FreeType is a widely adopted open-source software library designed specifically for font rendering and rasterization on CPUs. Since its initial release, it has become the de facto standard for vector font handling in software-only environments, including embedded systems, emulators, and desktop applications without GPU acceleration. FreeType supports a comprehensive range of font formats such as TrueType (TTF), OpenType (OTF), Type 1, CFF, and bitmap fonts. It performs vector outline interpretation, hinting, and rasterization entirely on the CPU, enabling high-quality scalable text rendering across multiple platforms.

This section details how to leverage FreeType in modern CPU-only graphics programming, with a focus on integration, rendering pipeline, performance considerations, and practical usage based on tools and operating systems updated since 2019.

6.2.2 FreeType Architecture and Core Components

1. **Library Structure**

   FreeType is structured into modular components:

   - **Font Face Loader:** Parses font files and extracts glyph outlines and metadata.
   - **Outline Interpreter:** Processes vector outlines (Bézier curves) and applies hinting instructions to improve legibility at small sizes.

- **Rasterizer:** Converts glyph outlines into pixel bitmaps, supporting anti-aliasing and subpixel rendering.
- **Cache Manager:** Optional module that stores rendered glyph bitmaps to optimize repeated rendering of the same glyph.

2. **Supported Font Formats**

- TrueType (TTF) and OpenType (OTF) with TrueType outlines
- PostScript Type 1 and Compact Font Format (CFF)
- Bitmap fonts embedded in scalable font files
- Web Open Font Format (WOFF) with additional parsing layers (optional)

Modern versions of FreeType (2.10 and later) provide improved support for variable fonts and advanced hinting features.

6.2.3 Integrating FreeType into CPU-Only Graphics Applications

1. **Initialization and Setup**

(a) **Library Initialization:**
Call `FT_Init_FreeType()` to initialize the FreeType library context. This sets up internal data structures and prepares the system for font loading.

(b) **Loading a Font Face:**
Use `FT_New_Face()` to load a font file from disk or memory. The face object encapsulates glyph outlines, font metrics, and style information.

(c) **Setting Font Size:**
Set the character size with `FT_Set_Char_Size()` or `FT_Set_Pixel_Sizes()`. This determines the output resolution of rasterized glyphs.

2. **Glyph Loading and Rasterization**

   - **Load Glyph:**
     Use `FT_Load_Char()` or `FT_Load_Glyph()` to load a glyph into the glyph slot for a specific Unicode codepoint or glyph index. This step parses vector outlines and applies hinting.
   - **Render Glyph to Bitmap:**
     Call `FT_Render_Glyph()` with the desired render mode (`FT_RENDER_MODE_NORMAL` for anti-aliased grayscale bitmaps, or `FT_RENDER_MODE_MONO` for monochrome bitmaps). The output bitmap can then be accessed via `face->glyph->bitmap`.

3. **Accessing Bitmap Data**

   The glyph bitmap is stored in a structure describing:

   - Width and height in pixels
   - Pitch (byte width of each row, which can be negative depending on buffer layout)
   - Pixel format (grayscale 8-bit or 1-bit monochrome)
   - Buffer pointer to raw pixel data

   The application must copy or composite this bitmap into the framebuffer, performing any necessary pixel format conversions.

6.2.4 Rendering Pipeline on the CPU

1. **Pixel-Level Composition**

   - **Blitting:**
     Directly copy glyph bitmap pixels into the target framebuffer with consideration for transparency and alpha values.
   - **Alpha Blending:**
     For anti-aliased fonts, blend glyph pixels with the background color using the glyph's alpha channel to smooth edges and improve readability.

2. **Handling Text Layout**

   FreeType is focused on glyph rasterization and does not provide high-level text layout or shaping. For complex scripts or bidirectional text, integration with libraries like HarfBuzz is necessary. However, for simpler Latin-based rendering, FreeType's glyph metrics (advance width, bearing) allow manual layout calculations.

6.2.5 Performance and Optimization Strategies

1. **Glyph Caching**

   Repeated rendering of the same characters is common in text rendering. Implementing a glyph cache—storing already rasterized glyph bitmaps in memory—avoids redundant computations.

   - Cache keys typically use font face, size, and glyph index.
   - Cache eviction policies (e.g., LRU) help manage memory usage.

2. **Using Hinting and Auto-Hinting**

   Hinting instructions in font files optimize glyph appearance at small sizes but increase CPU load. FreeType supports:

   - **Native hinting:** Uses font-provided hints for best quality.
   - **Auto-hinting:** Applies synthesized hints when native hints are unavailable.
   - **Disabling hinting:** For faster rendering with some quality loss.

   The choice affects rendering speed and output quality.

3. **Multi-threading Considerations**

   FreeType is thread-safe provided separate library and face objects are used per thread. Applications can rasterize glyphs in parallel to improve throughput on multi-core CPUs.

6.2.6 Cross-Platform Considerations

- **Windows:**
  FreeType integrates well with MinGW, MSVC, and CMake-based build systems. Output bitmaps can be rendered into memory buffers managed via GDI, Direct2D software fallbacks, or Win32 API surfaces.

- **Linux:**
  Commonly used with GCC or Clang compilers, FreeType is often paired with X11 or Wayland for display. Framebuffer or SDL surfaces can receive the rasterized bitmaps.

- **Embedded Systems:**

Custom builds of FreeType can be tailored to reduce footprint. Applications can render glyphs directly into framebuffer memory, often in grayscale or monochrome modes for LCD displays.

6.2.7 Practical Example Workflow

1. Initialize FreeType library context.
2. Load a TTF font file from disk or embedded resource.
3. Set pixel size for desired text height.
4. For each character:
    - Load and render glyph to bitmap.
    - Copy and alpha blend glyph bitmap into the framebuffer memory.
5. Update the display surface with modified framebuffer contents.

This straightforward approach allows developers to add scalable, high-quality text rendering in CPU-only graphical environments.

6.2.8 Summary

FreeType remains a powerful and flexible solution for font rasterization on the CPU. Its mature API and cross-platform support make it ideal for projects that cannot rely on GPU acceleration, including emulators, embedded devices, and custom operating systems.
By combining FreeType with careful caching, optimized memory handling, and efficient pixel compositing, developers can achieve crisp and scalable text rendering suitable for modern applications even in strictly CPU-bound graphics contexts.

6.3 Drawing Characters and Strings Manually

6.3.1 Introduction

Rendering text on the CPU without GPU assistance requires meticulous control over each step of character drawing and string composition. This section explores the manual process of drawing individual characters and concatenated strings using raw pixel operations, typically following glyph rasterization via libraries such as FreeType or custom bitmap fonts.

The methods detailed here are foundational for software-only graphics engines, embedded systems, or environments lacking advanced text rendering frameworks. The explanations assume a modern CPU environment with efficient memory management, and cover practical techniques applicable across Windows, Linux, and embedded platforms.

6.3.2 Drawing Single Characters

1. **Obtaining Glyph Bitmap Data**

   Before drawing a character, the glyph must be rasterized to a bitmap form. Using FreeType as an example:

   - Load the glyph with `FT_Load_Char()`.
   - Render it to a bitmap using `FT_Render_Glyph()`.
   - Access the bitmap buffer via `face->glyph->bitmap`.

   The bitmap contains:

   - Width and height in pixels

- Pixel buffer (grayscale or monochrome)
- Pitch (number of bytes per bitmap row)
- Bitmap left and top offsets (bearing information for positioning)

2. **Calculating Pixel Coordinates**

   To draw a glyph at a specific screen position `(x, y)`:

   - The horizontal position where the bitmap starts is `x + bitmap_left`.
   - The vertical position where the bitmap baseline aligns is `y - bitmap_top`.

   This calculation accounts for glyph bearings, which adjust placement to align characters properly on a baseline.

3. **Writing Pixels to the Framebuffer**

   Using the bitmap data and calculated coordinates, the pixel loop proceeds as follows:

   - For each row in the glyph bitmap:
     - For each column in the row:
       * Read the glyph pixel intensity.
       * If anti-aliased (grayscale), perform alpha blending with the framebuffer pixel.
       * If monochrome, write or clear the pixel accordingly.
   - Boundary checks are essential to avoid writing outside the framebuffer area.

   Alpha blending is typically done by combining the glyph pixel's alpha value with the background color, using the formula:

```
output_pixel = (glyph_alpha * glyph_color) + ((1 - glyph_alpha) *
↪  background_color)
```

Optimized blending routines utilize SIMD instructions (e.g., SSE, AVX) on modern CPUs, often exposed via libraries or custom assembly for speed.

6.3.3 Drawing Strings: Handling Layout and Positioning

1. **Horizontal String Composition**

   Strings consist of sequences of characters rendered consecutively, requiring precise horizontal advance computations.

   - After drawing each glyph, advance the drawing position by the glyph's `advance.x` value.
   - Use the `face->glyph->advance.x` metric from FreeType, usually in 26.6 fixed-point format, converted to integer pixels.
   - Handle kerning adjustments for character pairs, improving visual spacing. FreeType provides kerning via `FT_Get_Kerning()`.

2. **Line Wrapping and Vertical Positioning**

   Manual string drawing also involves:

   - Managing line breaks (`\n` characters).
   - Tracking vertical offset increments based on the font's line height (`face->size->metrics.height`).
   - Supporting multi-line rendering by resetting the horizontal position and increasing vertical position after each line.

3. **Handling UTF-8 and Unicode**

   Proper text rendering requires decoding multi-byte UTF-8 sequences into Unicode code points before glyph loading.

   - Decode UTF-8 characters into `uint32_t` codepoints.
   - Use `FT_Load_Char()` or equivalent with the decoded codepoint.
   - Handle missing glyphs gracefully, substituting with fallback characters if necessary.

6.3.4 Manual Implementation: Practical Considerations

1. **Buffer Management**

   - The destination framebuffer is typically a contiguous memory array with a known pixel format (e.g., 32-bit ARGB).
   - Pointer arithmetic is used to calculate pixel addresses based on x and y coordinates.
   - Stride or pitch of the framebuffer must be accounted for to move between rows.

2. **Performance Optimizations**

   - Cache glyph bitmaps to avoid redundant rasterization.
   - Batch draw calls by preparing glyph bitmaps in contiguous memory buffers.
   - Use pointer increments rather than multiplications inside inner pixel loops.
   - Employ platform-specific SIMD instructions where applicable.

3. **Handling Transparency and Blending**

- Support for alpha blending enables smooth, anti-aliased text.
- When compositing onto backgrounds with transparency, pre-multiplied alpha formats simplify blending calculations.
- In CPU-only environments, blending must be carefully optimized to maintain interactive frame rates.

6.3.5 Example Workflow Summary

1. Initialize FreeType and load the font face.
2. Set desired font size in pixels.
3. For each character in the string:
    - Decode UTF-8 to Unicode.
    - Load and render glyph bitmap.
    - Calculate drawing position using bearing and advances.
    - Alpha blend glyph bitmap into framebuffer.
    - Advance horizontal cursor, applying kerning if needed.
4. Handle newline characters and vertical line advancement.
5. Update the display surface with modified framebuffer content.

6.3.6 Cross-Platform Notes

- On Windows, pixel buffers can be mapped to GDI bitmaps or Direct2D software surfaces for display.

- On Linux, X11 Pixmaps or direct framebuffer memory are targets for manual pixel writes.
- SDL surfaces provide an abstraction for framebuffer access on multiple platforms, simplifying pixel manipulation.

6.3.7 Conclusion

Manually drawing characters and strings on the CPU requires a disciplined approach to glyph positioning, pixel manipulation, and memory management. While more labor-intensive than using GPU-accelerated text rendering, this method offers precise control and wide compatibility, making it essential for software renderers, retro-style engines, and embedded graphics applications.

With modern CPU capabilities and optimized libraries like FreeType, developers can achieve high-quality, scalable text rendering entirely on the CPU, matching many use cases where GPU support is limited or unavailable.

6.4 Simple GUI Textboxes and Labels

6.4.1 Introduction

Implementing graphical user interface (GUI) elements such as textboxes and labels entirely on the CPU, without GPU acceleration, demands careful integration of text rendering techniques with user input handling, state management, and pixel-based drawing primitives. This section presents a detailed guide for creating simple, efficient textboxes and labels using CPU-based rendering methods discussed earlier in this book. The approaches outlined here are practical for lightweight applications, embedded systems, retro-inspired software, and scenarios where GPU resources are unavailable or deliberately avoided. They leverage modern CPU features and standard libraries available after 2019 to balance performance with ease of implementation.

6.4.2 Labels: Static Text Display

1. **Concept and Use-Cases**

   Labels represent the simplest form of text UI element: a static display of text without user interaction. They are essential for displaying titles, descriptions, and status information.

   Key characteristics:

   - Rendered once or infrequently.
   - Positioned at fixed coordinates within the window or viewport.
   - Do not require focus or event handling.

2. **Implementation Steps**

(a) **Text Preparation**
Use CPU-based font rasterization (e.g., FreeType) to generate glyph bitmaps for the label's text. Cache the rendered text bitmap if the label content does not change frequently, reducing redundant computation.

(b) **Positioning**
Calculate the label's drawing position in pixels, considering alignment (left, center, right) and vertical baseline. Alignment calculations use string width metrics obtained from glyph advances and kerning.

(c) **Drawing**
Draw the pre-rasterized text bitmap into the framebuffer using pixel-level operations, applying alpha blending for smooth edges. Drawing typically uses a rectangular clipping area to avoid overflow.

(d) **Refreshing**
Update the label only when its text changes or when the UI requires redrawing, minimizing CPU load.

3. **Optimization and Practical Considerations**

- For multi-line labels, implement simple line breaking by measuring string widths and splitting on whitespace.
- Use double buffering or offscreen surfaces to eliminate flicker during redraws.
- Pre-render common strings to bitmaps during initialization for faster runtime rendering.

6.4.3 Textboxes: Editable Text Input Fields

1. **Concept and Use-Cases**

Textboxes allow users to input and edit text. Unlike labels, textboxes require:

- Keyboard input handling.
- Careful cursor management.
- Visual feedback for focus and text selection.
- Scrolling support for text exceeding the visible area.

Implementing these features on the CPU necessitates integration of text rendering with input event processing and custom state machines.

2. **Text Storage and Buffer Management**

   - Maintain a dynamic text buffer (e.g., a resizable string or array) in memory representing the current textbox content.
   - Store the cursor position as an index into the text buffer.
   - Support for UTF-8 strings requires proper handling of multi-byte code points during insertion, deletion, and cursor movement.

3. **Rendering the Textbox**

   (a) **Background and Border**
   Draw the textbox rectangle using filled rectangles and lines, optionally with anti-aliased edges for smooth appearance.

   (b) **Text Drawing**

   - Rasterize the visible portion of the text buffer using CPU text rendering methods.
   - Clip the text rendering area to the textbox interior.
   - Support horizontal scrolling if the text exceeds the width of the textbox, by adjusting the starting glyph offset.

(c) **Cursor Rendering**

- Draw a blinking cursor at the current text insertion point.
- Calculate cursor pixel position by summing glyph advance widths of all characters preceding the cursor index.
- Use a simple vertical line or block as the cursor indicator.

(d) **Selection Highlighting (Optional)**

- Maintain selection start and end indices.
- Render a semi-transparent rectangle behind selected glyphs to visually indicate selection.
- Combine selection rendering with alpha blending for smooth integration.

4. **Handling Keyboard Input**

- Capture input events at the OS level (e.g., Win32 message loop or X11 event handling).
- Decode input to UTF-8 codepoints.
- Update the text buffer on character insertions, deletions (backspace/delete), and cursor movements (arrow keys).
- Re-render the textbox contents after each input event.
- Implement clipboard operations if desired, handling paste and copy manually by accessing OS clipboard APIs.

5. **Focus and Interaction**

- Track focus state to determine when the textbox should receive keyboard input.
- Visualize focus by changing border color or adding a glow effect.

- Process mouse events for:
  - Setting cursor position on click.
  - Selecting text via drag.

  Implement hit-testing by calculating glyph bounding boxes within the textbox.

6.4.4 Performance Considerations

- Minimize full redraws by only updating regions of the framebuffer that change (dirty rectangles).
- Cache glyph bitmaps and rendered text lines to avoid expensive repeated rasterization.
- Use efficient data structures for the text buffer that support fast insertions and deletions (e.g., gap buffers or ropes).
- Optimize cursor blink timing using high-resolution timers or event-driven approaches rather than polling loops.

6.4.5 6Cross-Platform Notes

- On **Windows**, integrate with Win32 API for window creation, message handling, and GDI or Direct2D for framebuffer display.
- On **Linux**, use X11 or Wayland for event handling and drawing targets, leveraging shared memory for framebuffer access where possible.
- SDL2 can be employed as an abstraction layer for windowing and input, simplifying cross-platform GUI development while maintaining CPU-only rendering.

6.4.6 6Summary

Simple GUI elements like labels and textboxes are fundamental building blocks for CPU-based graphical applications. By combining manual text rendering techniques with event-driven input handling and pixel-level drawing primitives, developers can create fully functional, responsive text UI components without reliance on GPU acceleration.

This section's approaches balance clarity and performance, making them ideal for embedded environments, lightweight software, or educational projects exploring the core principles of graphics programming on the CPU.

Chapter 7

Animation and Timing

7.1 Framebuffers and Double-Buffering

7.1.1 Introduction

Smooth animation and flicker-free graphical output are fundamental goals in graphics programming. When rendering is performed solely on the CPU, these goals become technically challenging due to limited rendering throughput and potential screen tearing. This section focuses on two essential techniques to achieve efficient and smooth rendering: **framebuffers** and **double-buffering**.

Framebuffers serve as the main drawing surfaces in memory, and double-buffering is a strategy to eliminate tearing and flickering during visual updates by separating the rendering and display surfaces. In the absence of GPU acceleration, these techniques become central to creating responsive and fluid visual applications.

7.1.2 Understanding the Framebuffer

1. **Definition and Purpose**

   A framebuffer is a block of memory representing a 2D grid of pixels. Each pixel is stored using a defined color format (e.g., 32-bit ARGB). Drawing operations performed on the framebuffer result in visual changes to the application's output once the memory is flushed to the display.

   In CPU-based graphics, the framebuffer is not automatically synced with the screen. It must be manually managed and copied (blitted) to the display surface using OS or library-specific APIs.

2. **Framebuffer Allocation**

   - **In Windows**, a framebuffer can be implemented as a `BITMAPINFO` structure with an associated pixel buffer (`DIB section`) that can be drawn using GDI (`StretchDIBits`, `BitBlt`) or rendered to a window using Direct2D in software mode.
   - **In Linux**, a framebuffer may be a block of user-allocated memory used with `XPutImage` in X11, or written directly to `/dev/fb0` in systems with framebuffer devices. With SDL2, a software surface can be locked and updated manually.
   - **Memory layout**: Pixel buffers must be tightly packed or follow a defined stride. Developers must be aware of row padding and alignment when accessing scanlines.

7.1.3 Double-Buffering: The Core Idea

1. **Why Double-Buffering?**

Flickering arises when rendering updates appear on-screen before drawing is complete. Without a GPU, drawing operations can be slow, and intermediate states of rendering become visible, causing tearing or artifacts. Double-buffering solves this by using two framebuffers:

- **Front buffer**: Currently displayed buffer.
- **Back buffer**: Where rendering is done off-screen.

Only after a full frame is rendered on the back buffer is it copied or swapped into the front buffer for display. This ensures only fully complete frames are shown to the user.

2. **Implementation Strategy**

   (a) **Create two pixel buffers in memory** of identical size and format.

   (b) **Render the entire scene** into the back buffer.

   (c) **Blit or copy** the back buffer to the front buffer or directly to the screen surface.

   (d) **Repeat for each frame**.

   Most software-based libraries such as SDL2 (with software rendering mode), or OS-specific APIs (like GDI or X11), support copying raw memory to the screen. The developer must manage the buffer switching manually in these cases.

7.1.4 Triple-Buffering and Alternatives

Triple-buffering is a more advanced technique that introduces a third buffer to decouple rendering from the display even further. This allows the CPU to continue drawing while another buffer is being presented. However, in CPU-only systems, the added complexity and memory cost often outweigh the benefits unless animation smoothness is critical. Alternatives such as **dirty rectangle rendering** (updating only changed regions) can sometimes reduce flicker and improve efficiency in simple UIs but are not substitutes for double-buffering in full-scene animations.

7.1.5 Practical Implementation with Modern Tools

1. **SDL2 Software Renderer**

   - SDL2 supports software rendering via `SDL_CreateRenderer(window, -1, SDL_RENDERER_SOFTWARE)`.
   - Allocate two `SDL_Surface` or `SDL_Texture` objects with access to their pixel buffers using `SDL_LockSurface`.
   - Draw manually to the back buffer, then use `SDL_UpdateTexture` and `SDL_RenderCopy` to present to the screen.

2. **Win32 API with GDI**

   - Use `CreateDIBSection` to create two `HBITMAP`s.
   - Draw to one `HBITMAP` and use `BitBlt` or `StretchDIBits` to blit it to the window's device context.
   - Synchronize presentation with the message loop.

3. **X11/Xlib**

   - Create two `XImage` structures.
   - Render to the memory of the back buffer using CPU drawing code.
   - Use `XPutImage` to present the back buffer to a visible `Drawable`.

7.1.6 Performance Considerations

- **Memory usage**: Double-buffering doubles the memory requirements for screen-sized pixel buffers.
- **Copy cost**: Copying the back buffer to the display is expensive on CPU; optimize by using fast memory operations (`memcpy`, optimized blit routines).
- **Synchronization**: Although V-Sync cannot be enforced directly in CPU-only rendering, frame timing logic (see next section) can approximate it.

7.1.7 Summary

Double-buffering is essential for creating flicker-free animations and smooth graphical updates when rendering with the CPU. By using two framebuffers—one for drawing and one for display—the application ensures that the user sees only fully rendered frames. Despite being a classical technique, it remains highly relevant in environments, including those using SDL2, Win32, or X11.

This foundation sets the stage for more advanced timing, interpolation, and motion control techniques that make up the heart of modern software rendering pipelines on CPU-bound systems.

7.2 Synchronizing Frames with the Refresh Rate (If Needed)

7.2.1 Introduction

A consistent visual experience in graphics applications relies not only on what is drawn but when it is drawn. Synchronizing animation frames with the refresh rate of the display prevents visual artifacts such as **tearing**, **stuttering**, and **jittering**, which can arise when the timing of drawing and display are out of sync.
While GPU pipelines often include vertical synchronization (V-Sync) mechanisms handled automatically, CPU-only rendering environments must implement their own timing systems. This section discusses strategies for managing frame timing and synchronizing frame presentation with the screen refresh rate using only the CPU, drawing from practices adopted in platforms and libraries.

7.2.2 Understanding Refresh Rate and Frame Timing

1. **What Is the Refresh Rate?**

   The refresh rate, typically expressed in **Hz**, is the number of times per second the display updates the contents of the screen. For example, a standard monitor with a 60 Hz refresh rate updates the screen every **16.666 milliseconds**. Displaying a new frame at inconsistent intervals can cause either tearing (overlapping multiple frame contents during a refresh) or jittering (noticeable temporal inconsistency in animation smoothness).

2. **Ideal Frame Time Calculation**

   To match the display refresh rate:

- 60 Hz: 1 frame every 16.666 ms
- 75 Hz: 1 frame every 13.333 ms
- 120 Hz: 1 frame every 8.333 ms
- 144 Hz: 1 frame every 6.944 ms

Knowing the target refresh rate allows the software renderer to throttle or pace its frame updates accordingly.

7.2.3 Frame Synchronization Techniques

1. **Fixed Time Step with Delay**

   This is the most basic method to achieve synchronization:

   (a) At the start of each frame, capture the current time.

   (b) Perform all rendering to the back buffer.

   (c) Calculate the elapsed time.

   (d) If the rendering finished faster than the target frame duration (e.g., 16.666 ms for 60 Hz), **delay** the rest using a sleep function (`Sleep` on Windows, `usleep` or `nanosleep` on Linux).

   (e) Blit the back buffer to the screen.

   This technique ensures consistent pacing but may suffer from **jitter** on systems with poor timer precision.

   - **Example (SDL2-based):**

```
Uint32 frameStart = SDL_GetTicks();
RenderFrame(); // User-defined drawing function
Uint32 frameEnd = SDL_GetTicks();
Uint32 elapsed = frameEnd - frameStart;
if (elapsed < 16) SDL_Delay(16 - elapsed); // Cap to ~60 FPS
```

2. **High-Precision Timing**

   For finer granularity, **high-resolution timers** introduced after 2019 should be used:

   - **Windows**: `QueryPerformanceCounter` and `QueryPerformanceFrequency`
   - **Linux/macOS**: `clock_gettime` with `CLOCK_MONOTONIC` or `std::chrono::high_resolution_clock` in modern C++

   These allow sub-millisecond accuracy in measuring frame duration, minimizing drift and improving synchronization precision.

7.2.4 Using Frame Throttling in CPU-Based Loops

1. **Throttling with a Real-Time Clock**

   Instead of locking to a fixed frame rate, another method is to **throttle** based on real-time targets. A running clock is used to determine the next frame presentation time, and logic is designed to skip rendering or interpolate frames accordingly.

   This is common in emulators and retro-style engines where real hardware timing needs to be emulated precisely (e.g., matching NES or SNES timings).

2. **daptive Frame Rates**

   On modern platforms, performance can vary significantly. An **adaptive strategy** monitors rendering duration and adjusts delays dynamically to approximate the refresh rate while minimizing visible jitter. This does not guarantee frame-perfect synchronization but provides better responsiveness when frame durations are unpredictable.

7.2.5 Tools and Platform-Specific Methods

1. **SDL2**

   - SDL2 supports software renderers with `SDL_RENDERER_SOFTWARE`.
   - `SDL_Delay`, `SDL_GetTicks`, and `SDL_GetPerformanceCounter` provide millisecond and microsecond-level time management.
   - Since SDL2 does not guarantee V-Sync in software mode, CPU-based applications must use frame delay strategies manually.

2. **Windows API**

   - `timeBeginPeriod(1)` may be used to improve timing resolution on Windows before using `Sleep` or `WaitableTimer`.
   - For more precision, use `QueryPerformanceCounter` for high-resolution measurements.

3. **Linux and POSIX**

   - Use `clock_gettime` or C++20 `std::chrono` with nanosecond resolution.
   - Combine with `nanosleep` to pause for accurate timing windows.

- If using X11 or framebuffer devices, draw manually to the buffer and throttle using timing logic.

7.2.6 Emulating V-Sync Without GPU

In traditional graphics, V-Sync prevents tearing by synchronizing frame flips with vertical blanking intervals (VBI). In CPU-only systems, there is **no direct access to hardware VBI**, especially on modern systems with GPU-managed desktops.
To simulate V-Sync:

- Estimate display refresh interval based on monitor information or assume standard 60 Hz.
- Maintain strict timing control using high-resolution clocks.
- Avoid overlapping frame drawing with blitting to prevent tearing artifacts manually.

Some tools (such as SDL2) may internally use synchronization under specific backends (like DirectX or Wayland), but this is **not guaranteed** in software rendering.

7.2.7 Summary

Synchronizing animation frames with the display's refresh rate is crucial for visual smoothness, even in CPU-only rendering environments. While there is no hardware-level V-Sync without a GPU, timing-based techniques such as **fixed time steps**, **high-resolution delays**, and **adaptive throttling** can approximate synchronization well.
For the examples and tools used in this book, including SDL2 (software mode), raw Win32 APIs, and Linux framebuffer/X11 systems, developers are expected to implement their own timing mechanisms using the system's high-resolution clocks. This

approach offers sufficient control to deliver smooth, professional-grade animations even without hardware acceleration.

7.3 Frame Timing: `Sleep()`, `usleep()`, `QueryPerformanceCounter`, `clock_gettime`

7.3.1 Introduction

Precise timing is one of the most critical aspects of software rendering, especially when animations and user interactions are involved. In GPU-accelerated environments, much of the frame pacing is offloaded to the graphics driver and hardware-based synchronization mechanisms like V-Sync. However, in CPU-only environments, the burden falls entirely on the software to determine **when to draw** and **how long to wait** between frames.

This section explores practical timing techniques using modern, APIs on Windows and Linux. It explains how to manage frame duration and delay execution using system-level functions such as `Sleep()`, `usleep()`, `QueryPerformanceCounter()`, and `clock_gettime()`.

7.3.2 The Importance of Frame Timing

In animation and simulation loops, two metrics are vital:

1. **Frame Duration (Δt):** The actual time it took to compute and display one frame.

2. **Frame Rate (FPS):** The inverse of frame duration. At 60 FPS, each frame should take approximately 16.666 milliseconds.

Without stable frame timing:

- Fast computers may render too many frames, causing wasted CPU cycles.

- Slower systems may lag, reducing the perceptual quality of the output.
- Inconsistent timing leads to stutter, jitter, and incorrect animation speeds.

To prevent these issues, we use **timing functions** to monitor frame processing and **sleep functions** to throttle frame rates deliberately.

7.3.3 Timing on Windows

1. **`Sleep(milliseconds)`**

   `Sleep()` is a Windows API call that suspends the execution of the current thread for a specified time in milliseconds.

   Syntax:

   ```
   #include <windows.h>
   Sleep(DWORD dwMilliseconds);
   ```

   Characteristics:

   - Simple to use.
   - Millisecond resolution.
   - May be imprecise (delays can exceed requested time).
   - On systems prior to Windows 10, it typically aligns with the system timer granularity (~15.6 ms). You may use `timeBeginPeriod(1)` to improve resolution temporarily.

   Use Case:

```
Sleep(16); // Pause ~16ms for ~60 FPS pacing
```

Limitation: Not suitable for high-precision frame throttling on its own.

2. **`QueryPerformanceCounter` and `QueryPerformanceFrequency`**

   These functions provide access to the **high-resolution performance counter** of the system.

   - `QueryPerformanceCounter()` returns the current tick count.
   - `QueryPerformanceFrequency()` returns the number of ticks per second.

   These are ideal for measuring **precise frame durations** and implementing frame throttling with sub-millisecond accuracy.

   Usage Example:

```
LARGE_INTEGER frequency;
LARGE_INTEGER start, end;

QueryPerformanceFrequency(&frequency);
QueryPerformanceCounter(&start);

// ... render or update frame ...

QueryPerformanceCounter(&end);
double elapsedMS = (double)(end.QuadPart - start.QuadPart) * 1000.0
 ↪  / frequency.QuadPart;
```

   Use the measured `elapsedMS` to determine how long the current frame took and how much time remains to meet the target frame duration.

Recommended Strategy:

Use `QueryPerformanceCounter` for measurement, and if the frame was too fast, call `Sleep()` for the remaining time.

7.3.4 Timing on Linux and POSIX-Compliant Systems

1. **`usleep(microseconds)`**

   `usleep()` is a POSIX function that suspends execution of the current thread for a specified number of microseconds.

   Syntax:

   ```
   #include <unistd.h>
   usleep(useconds_t microseconds);
   ```

   Use Case:

   ```
   usleep(16000); // ~16ms for ~60 FPS
   ```

   Notes:

   - Precise for delays >10 milliseconds.
   - Not recommended for very short delays (<1 ms) due to scheduling latencies.
   - `usleep()` is deprecated in some environments in favor of `nanosleep()`.

2. **`clock_gettime(clock_id, &timespec)`**

   Modern Linux systems and POSIX platforms support `clock_gettime()` for **high-resolution** time measurement. It is a reliable alternative to `QueryPerformanceCounter()` on Unix-like systems.

Syntax:

```
#include <time.h>
struct timespec start, end;

clock_gettime(CLOCK_MONOTONIC, &start);
// ... do work ...
clock_gettime(CLOCK_MONOTONIC, &end);
```

To calculate the elapsed time:

```
long seconds = end.tv_sec - start.tv_sec;
long nanoseconds = end.tv_nsec - start.tv_nsec;
double elapsedMS = seconds * 1000.0 + nanoseconds / 1e6;
```

Clock Types:

- `CLOCK_MONOTONIC`: High-resolution, guaranteed to move forward, immune to system time changes.
- `CLOCK_REALTIME`: Represents wall-clock time, not recommended for animation or delta timing.

Use Case: High-precision frame timing and synchronization.

7.3.5 Practical Timing Strategy

A common structure for software frame control includes:

1. **Start Timer**
2. **Render/Update Logic**

3. **End Timer**

4. **Calculate Duration**

5. **Sleep Remaining Time**

Example for 60 FPS pacing:

Windows:

```
LARGE_INTEGER freq, start, end;
QueryPerformanceFrequency(&freq);
QueryPerformanceCounter(&start);

// ... perform rendering ...

QueryPerformanceCounter(&end);
double frameTime = (end.QuadPart - start.QuadPart) * 1000.0 /
↪  freq.QuadPart;

if (frameTime < 16.666) {
    Sleep((DWORD)(16.666 - frameTime));
}
```

Linux:

```
struct timespec start, end;
clock_gettime(CLOCK_MONOTONIC, &start);

// ... perform rendering ...

clock_gettime(CLOCK_MONOTONIC, &end);
long elapsedMS = (end.tv_sec - start.tv_sec) * 1000 + (end.tv_nsec -
↪  start.tv_nsec) / 1000000;
```

```
if (elapsedMS < 17) {
    usleep((17 - elapsedMS) * 1000); // Sleep the remaining time
}
```

7.3.6 Timing Libraries and SDL Integration

While SDL2 itself abstracts some timing functions:

- `SDL_GetTicks()`: Millisecond timer
- `SDL_Delay(ms)`: Millisecond sleep
- `SDL_GetPerformanceCounter()` and `SDL_GetPerformanceFrequency()` mimic the Windows API

You may use these when building on top of SDL, but for full control in software rendering, it's often more educational and precise to rely on system-level APIs.

7.3.7 Summary

Effective frame timing is vital for consistent animation in CPU-only rendering. Developers must understand both **sleeping mechanisms** and **high-resolution timers** across platforms:

- On **Windows**, prefer `QueryPerformanceCounter()` for precision, `Sleep()` for coarse delays.
- On **Linux**, use `clock_gettime()` with `CLOCK_MONOTONIC`, and `usleep()` or `nanosleep()` for delays.

By combining these tools intelligently, a software renderer can achieve frame stability and visual smoothness comparable to GPU-backed solutions, maintaining consistent frame rates and smooth user experiences.

7.4 Simple Sprite Movement and Animation Loops

7.4.1 Introduction

In CPU-only graphics environments, sprite animation must be implemented entirely through manual manipulation of memory buffers and frame timing. Unlike GPU-accelerated systems where hardware handles transformations and frame refresh synchronization, software renderers require the developer to explicitly manage sprite states, animation frames, timing intervals, and screen updates.

This section provides a detailed breakdown of how to create basic sprite movement and animation loops in a CPU-based software renderer. The methods presented are consistent with modern C/C++ practices and compatible with current operating systems and tools available after 2019, such as Windows 10/11, modern Linux distributions (Ubuntu 22.04+, Fedora 38+, etc.), and development environments including Visual Studio 2019+, GCC 9+, and Clang 10+.

7.4.2 What Is a Sprite?

A **sprite** is a 2D image or bitmap representing an object that can move or animate independently on the screen. In software rendering, sprites are manually drawn into the framebuffer using techniques such as **blitting** (copying pixel data) and **alpha blending** for transparency.

Sprites can represent characters, objects, effects, or UI elements.

7.4.3 Sprite Structure in Memory

To implement sprite movement and animation, each sprite is typically represented as a structure:

```
struct Sprite {
    uint8_t* imageData;   // Pointer to pixel data (ARGB, RGBA, etc.)
    int width;
    int height;
    int x;
    int y;
    int frameIndex;
    int totalFrames;
    int frameWidth;
    int frameHeight;
    int frameDelayMs;
    uint64_t lastFrameTime;
};
```

This structure allows:

- Tracking the sprite's position (`x`, `y`).
- Managing sprite sheet frames for animation (`frameIndex`, `totalFrames`).
- Managing timing (`frameDelayMs`, `lastFrameTime`).

7.4.4 Manual Sprite Movement

To move a sprite across the screen, its position values (`x`, `y`) are updated over time, based on direction or speed.

Basic linear movement:

```
void updateSpritePosition(Sprite& sprite, int dx, int dy) {
    sprite.x += dx;
    sprite.y += dy;
}
```

Physics-based movement (e.g., with velocity):

```
struct Vec2 { float x, y; };

struct MovingSprite {
    Sprite sprite;
    Vec2 velocity;
};

void update(MovingSprite& ms, float deltaTime) {
    ms.sprite.x += static_cast<int>(ms.velocity.x * deltaTime);
    ms.sprite.y += static_cast<int>(ms.velocity.y * deltaTime);
}
```

Delta Time Calculation: Use `QueryPerformanceCounter()` on Windows or `clock_gettime()` on Linux to measure `deltaTime` between frames for time-based movement instead of frame-based (which can vary).

7.4.5 Sprite Animation Loops

Sprite animation is typically achieved by cycling through different frames in a sprite sheet. A **sprite sheet** is a single image composed of several frames arranged horizontally or vertically.

Animation update logic:

```
void updateAnimation(Sprite& sprite, uint64_t currentTimeMs) {
    if (currentTimeMs - sprite.lastFrameTime >= sprite.frameDelayMs) {
        sprite.frameIndex = (sprite.frameIndex + 1) % sprite.totalFrames;
        sprite.lastFrameTime = currentTimeMs;
    }
}
```

Frame drawing logic (extract a frame from the sheet):

```
void drawSpriteFrame(uint32_t* framebuffer, int fbWidth, const Sprite&
 ↪  sprite) {
    int srcX = sprite.frameIndex * sprite.frameWidth;
    int srcY = 0;

    for (int y = 0; y < sprite.frameHeight; ++y) {
        for (int x = 0; x < sprite.frameWidth; ++x) {
            uint32_t pixel = ((uint32_t*)sprite.imageData)[(srcY + y) *
             ↪  sprite.width + (srcX + x)];
            framebuffer[(sprite.y + y) * fbWidth + (sprite.x + x)] =
             ↪  pixel;
        }
    }
}
```

7.4.6 Timing Considerations

To ensure consistent animation regardless of system performance, use precise timing APIs:

- **Windows:** `QueryPerformanceCounter()`, `GetTickCount64()`
- **Linux:** `clock_gettime(CLOCK_MONOTONIC, ...)`

This ensures that frame updates and animation delays are time-based, not tied to unpredictable frame render speed.
Example of getting the current time in milliseconds:
Windows:

```
LARGE_INTEGER freq, counter;
QueryPerformanceFrequency(&freq);
QueryPerformanceCounter(&counter);
uint64_t currentMs = (counter.QuadPart * 1000) / freq.QuadPart;
```

Linux:

```
struct timespec ts;
clock_gettime(CLOCK_MONOTONIC, &ts);
uint64_t currentMs = ts.tv_sec * 1000 + ts.tv_nsec / 1000000;
```

7.4.7 Main Animation Loop

Here is a typical sprite animation loop structure for software rendering:

```
while (running) {
    uint64_t now = getCurrentTimeInMs();

    // Update positions and animations
    updateSpritePosition(mySprite, dx, dy);
    updateAnimation(mySprite, now);

    // Clear framebuffer
    memset(framebuffer, 0, width * height * sizeof(uint32_t));

    // Draw sprite
    drawSpriteFrame(framebuffer, screenWidth, mySprite);

    // Swap/draw framebuffer to screen
    presentFramebuffer();
    // Sleep to maintain consistent FPS
    sleepUntilNextFrame();
}
```

You may maintain frame pacing using the techniques discussed in Section 7.3, ensuring smooth animation.

7.4.8 Modern Tools and Environments

- **Windows Environments:**
  - Use Visual Studio 2019/2022 with raw Win32 or SDL2 backends.
  - SDL2 can be used to open a window and provide access to framebuffer memory.
  - `Sleep()` and `QueryPerformanceCounter()` still represent the best low-level options.
- **Linux Environments:**
  - Use GCC 9+, Clang 10+ with X11, Wayland, or framebuffer access (`/dev/fb0`).
  - Tools such as SDL2 or raw mmap to framebuffer devices are practical.
  - `usleep()` and `clock_gettime()` offer sufficient timing precision.
- **Cross-platform Build Systems:**
  - CMake remains the best choice for portability.
  - For larger projects, consider integrating Meson or Ninja for speed.

7.4.9 Summary

Sprite animation and movement in CPU-only graphics programming requires:

- Structuring sprite data with position and animation metadata.
- Using precise timers to measure and regulate frame progression.
- Implementing sprite rendering manually through blitting techniques.
- Managing movement through position deltas or physics-inspired models.
- Synchronizing animation frames with real time to ensure smooth motion.

With a consistent animation loop and careful timing control, even CPU-bound environments can achieve responsive and visually fluid sprite animations. This foundation paves the way for more complex game mechanics, UI systems, or visualizations, all without requiring a GPU.

Chapter 8

Real-Time 2D Effects

8.1 Software Scrolling (Hardware-Like Tilemaps)

8.1.1 Introduction

Software scrolling is a foundational technique in 2D graphics, especially in game development, user interface rendering, and real-time visualization systems. In GPU-powered systems, this is often achieved through hardware-accelerated tilemaps and transformations. However, in CPU-only graphics programming, we must emulate these capabilities manually by manipulating memory buffers and calculating tile offsets.

In this section, we explore how to implement smooth, hardware-like tilemap scrolling using only CPU resources. This includes managing tilemaps, calculating viewport offsets, and redrawing visible regions efficiently.

8.1.2 What Is a Tilemap?

A **tilemap** is a 2D grid made up of small graphical units called **tiles**. Each tile typically represents an 8x8 or 16x16 pixel region. Tilemaps allow for efficient rendering of large worlds by reusing a small set of graphical resources.
The primary benefits include:

- **Memory efficiency**: Tile indices reference shared graphics rather than storing large raw image data.
- **Fast scrolling**: Only visible tiles need to be drawn to the screen.

A tilemap consists of:

- A **tile set** (bitmap atlas) that contains all the graphical tiles.
- A **map buffer** (2D array) holding tile indices that refer to tiles in the tile set.
- A **viewport** defining the portion of the tilemap currently visible on screen.

8.1.3 Tilemap Data Structures

To implement a tilemap, we define the following structures in C++:

```cpp
struct Tile {
    uint32_t* pixels;   // Pixel data (width x height)
    int width;
    int height;
};

struct TileMap {
    int rows;
    int cols;
    int tileWidth;
    int tileHeight;
    std::vector<uint16_t> mapData;   // Tile indices
    std::vector<Tile> tileSet;
};

struct Viewport {
    int xOffset;
    int yOffset;
    int width;
    int height;
};
```

The `mapData` array holds tile indices in row-major order. A 100x100 map with 16x16 tiles would need only 10,000 `uint16_t` entries instead of a full pixel buffer for the entire world.

8.1.4 Rendering the Viewport

To scroll smoothly, the viewport must be able to move across the tilemap with pixel precision. That means not just shifting tiles but also handling partial tile rendering along edges.

```
void renderTilemap(
    uint32_t* framebuffer, int fbWidth, int fbHeight,
    const TileMap& map, const Viewport& view
) {
    int startCol = view.xOffset / map.tileWidth;
    int startRow = view.yOffset / map.tileHeight;
    int xOffsetInTile = view.xOffset % map.tileWidth;
    int yOffsetInTile = view.yOffset % map.tileHeight;

    for (int y = 0; y <= view.height / map.tileHeight + 1; ++y) {
        for (int x = 0; x <= view.width / map.tileWidth + 1; ++x) {
            int mapIndex = (startRow + y) * map.cols + (startCol + x);
            if (mapIndex >= map.mapData.size()) continue;

            uint16_t tileIndex = map.mapData[mapIndex];
            const Tile& tile = map.tileSet[tileIndex];

            drawTileClipped(
                framebuffer, fbWidth, fbHeight,
                tile,
                x * map.tileWidth - xOffsetInTile,
                y * map.tileHeight - yOffsetInTile
            );
        }
    }
}
```

The `drawTileClipped()` function must ensure that drawing operations do not overwrite memory outside the framebuffer and that partially visible tiles are correctly rendered.

8.1.5 Smooth Pixel-Level Scrolling

Unlike column- or row-based scrolling, pixel-level scrolling enhances the fluidity of motion. This is especially important when rendering camera movement, parallax backgrounds, or player-controlled viewport shifts.

The viewport's `xOffset` and `yOffset` values are updated per frame:

```
void scrollViewport(Viewport& view, int dx, int dy, int mapPixelWidth,
 ↪  int mapPixelHeight) {
    view.xOffset = std::clamp(view.xOffset + dx, 0, mapPixelWidth -
     ↪  view.width);
    view.yOffset = std::clamp(view.yOffset + dy, 0, mapPixelHeight -
     ↪  view.height);
}
```

8.1.6 Optimizing Tile Rendering

To maximize performance:

- **Only draw visible tiles**: As shown, calculate starting row/column based on `xOffset/yOffset`.
- **Use dirty rectangles**: If the scene is mostly static, only redraw parts that changed.
- **Avoid redundant memory writes**: Use a backbuffer and only copy to the front buffer when the scene is updated.

8.1.7 Tools and Platforms

This approach is fully compatible with modern CPU-focused environments:

- **Windows 10/11**:
  - Use `CreateDIBSection()` or `SDL_CreateRGBSurface()` to create a software framebuffer.
  - Update the framebuffer using `memcpy()` or pointer arithmetic.

- **Linux (Wayland/X11 or Framebuffer Console)**:
  - Use `mmap()` on `/dev/fb0` for console environments.
  - Use SDL2 or raw Xlib for X11 framebuffer access.
- **Cross-platform libraries**:
  - **SDL2 (2.0.10+)**: Supports software surfaces and pixel manipulation.
  - **raylib** (CPU-backend mode): Can be adapted for software-only rendering.
  - **Dear ImGui (software mode)**: Can serve as GUI overlay over a tile-based system.

8.1.8 A Complete Loop Example

```
Viewport camera = {0, 0, 320, 240};  // Screen size
TileMap world = loadTileMap("level1.map");

while (running) {
    // Handle input
    if (keyPressed(KEY_RIGHT)) scrollViewport(camera, 2, 0, world.cols *
    ↪  world.tileWidth, world.rows * world.tileHeight);
    if (keyPressed(KEY_LEFT)) scrollViewport(camera, -2, 0, world.cols *
    ↪  world.tileWidth, world.rows * world.tileHeight);

    // Clear framebuffer
    memset(framebuffer, 0, fbWidth * fbHeight * sizeof(uint32_t));

    // Render tilemap to framebuffer
    renderTilemap(framebuffer, fbWidth, fbHeight, world, camera);
    // Present framebuffer to screen
    presentFramebuffer();
}
```

This basic loop showcases software-side scrolling and camera translation across a large 2D world rendered entirely by the CPU.

8.1.9 Summary

Software scrolling using tilemaps on the CPU involves:

- Efficient memory representation of 2D worlds via tile indices.
- Dynamic rendering based on a movable viewport.
- Pixel-level precision for smooth visual motion.
- Careful memory access and clipping to ensure robustness and performance.
- Full control over rendering order, layering, and priority effects.

This technique replicates the behavior of early hardware tile-based renderers used in consoles and handheld systems, but within a modern CPU-only rendering engine. With appropriate optimizations, it can support large scrolling maps, multiple layers, and real-time updates on contemporary CPUs without any GPU involvement.

8.2 Transparency and Alpha Blending

8.2.1 Introduction

Transparency and alpha blending are essential for modern 2D rendering pipelines, enabling layered composition of images, UI elements, sprites, and visual effects. In GPU-based graphics, blending is typically handled in hardware using programmable shaders. However, when rendering exclusively on the CPU, transparency must be manually implemented by computing the effect of each pixel's alpha value during composition.

In this section, we will explore the mathematics behind alpha blending, how to optimize it for real-time performance on the CPU, and how to handle premultiplied alpha, clipping, and compositing strategies. All implementations are tailored for systems and toolchains, making them practical for Linux and Windows development without reliance on GPU acceleration.

8.2.2 Understanding Alpha

Alpha represents the opacity level of a pixel:

- `alpha = 1.0` (255 in 8-bit) means fully opaque.
- `alpha = 0.0` (0 in 8-bit) means fully transparent.
- Values in between indicate partial transparency.

In ARGB pixel formats, a typical 32-bit representation is:

```
Bits:  [31-24]    [23-16]   [15-8]    [7-0]
       Alpha      Red       Green     Blue
```

Some systems use RGBA or BGRA layouts, and care must be taken to correctly access the intended channels during blending.

8.2.3 Alpha Blending Equation

To blend a source pixel (foreground) over a destination pixel (background), use the standard Porter-Duff "over" operation:

```
OutColor = SrcColor * SrcAlpha + DstColor * (1 - SrcAlpha)
```

For each color channel (R, G, B), this becomes:

```
result.r = src.r * src.a / 255 + dst.r * (255 - src.a) / 255;
result.g = src.g * src.a / 255 + dst.g * (255 - src.a) / 255;
result.b = src.b * src.a / 255 + dst.b * (255 - src.a) / 255;
result.a = max(src.a, dst.a); // Often optional
```

When implemented in integer arithmetic (for performance), the division by 255 can be optimized using bit shifts or fixed-point math.

8.2.4 Implementing Per-Pixel Alpha Blending

A simple CPU-side alpha blending function for 32-bit ARGB pixels:

```
inline uint32_t blendARGB(uint32_t src, uint32_t dst) {
    uint8_t sa = (src >> 24) & 0xFF;
    uint8_t sr = (src >> 16) & 0xFF;
    uint8_t sg = (src >> 8) & 0xFF;
    uint8_t sb = src & 0xFF;

    uint8_t dr = (dst >> 16) & 0xFF;
    uint8_t dg = (dst >> 8) & 0xFF;
    uint8_t db = dst & 0xFF;

    uint8_t r = (sr * sa + dr * (255 - sa)) / 255;
    uint8_t g = (sg * sa + dg * (255 - sa)) / 255;
    uint8_t b = (sb * sa + db * (255 - sa)) / 255;

    return (0xFF << 24)  (r << 16)  (g << 8) | b;
}
```

This function blends a single source pixel over a destination pixel and returns the final result. A framebuffer can be updated with this result during rasterization.

8.2.5 Optimizing Alpha Blending on CPU

Although CPU-based blending is more expensive than simple pixel copying, performance can be managed through:

- **Loop unrolling**: Reduces overhead of per-pixel function calls.
- **SIMD instructions (SSE/AVX/NEON)**: Vectorized blending of multiple pixels simultaneously.
- **Avoiding redundant operations**: Skip blending if alpha is 255 (fully opaque) or 0 (fully transparent).
- **Batching operations**: Minimize cache misses by accessing memory sequentially.

For performance-critical code, consider implementing specialized code paths for common blending cases (e.g., 50% transparency, fully opaque sprites).

8.2.6 Premultiplied Alpha

Premultiplied alpha simplifies blending math and reduces artifacts. Instead of storing raw color components, the color values are already multiplied by the alpha:

```
Premultiplied Color = (R × A, G × A, B × A, A)
```

The blending equation becomes:

```
OutColor = SrcColor + DstColor * (1 - SrcAlpha)
```

This allows the blend to avoid multiply operations for the source color, which improves speed when rendering sprites.
However, input images must be prepared accordingly, and texture loading functions must take this format into account.

8.2.7 Use Cases for Alpha Blending

Alpha blending enables:

- **Layered sprites** with partial transparency
- **Fading effects** (in/out transitions)
- **Text with shadows** or translucent highlights
- **Glass, smoke, or particle effects**
- **UI composition** (tooltips, popups, overlays)

Each use case must consider ordering (painter's algorithm) and z-index priority for correct rendering.

8.2.8 Example: Drawing a Transparent Sprite

```
void drawSpriteAlpha(
    uint32_t* framebuffer, int fbWidth, int fbHeight,
    const uint32_t* spriteData, int spriteWidth, int spriteHeight,
    int dstX, int dstY
) {
    for (int y = 0; y < spriteHeight; ++y) {
        for (int x = 0; x < spriteWidth; ++x) {
            int fbIndex = (dstY + y) * fbWidth + (dstX + x);
            int spIndex = y * spriteWidth + x;

            uint32_t srcPixel = spriteData[spIndex];
            uint32_t dstPixel = framebuffer[fbIndex];

            framebuffer[fbIndex] = blendARGB(srcPixel, dstPixel);
        }
    }
}
```

This function overlays a sprite onto the framebuffer, pixel by pixel, using alpha blending. It assumes valid bounds and preprocessed ARGB data.

8.2.9 Handling Transparent Masks

An alternative to alpha blending is binary transparency via color-keying or mask buffers. This approach checks if a pixel equals a designated color (e.g., magenta `0xFF00FF`) or if a mask bit is set before drawing:

```
if (srcPixel != TRANSPARENT_COLOR) {
    framebuffer[fbIndex] = srcPixel;
}
```

While less flexible, it is faster and historically used in systems without alpha channel support.

8.2.10 Tools and Support

- **Windows (10/11)**:
  - Native GDI+ and DIBs support alpha blending through CPU APIs.
  - Modern SDL2 (2.0.10+) allows locking surfaces and writing pixels manually.
- **Linux (X11, Wayland)**:
  - Direct pixel access via memory-mapped framebuffers or off-screen surfaces using libraries like Cairo (software mode) or SDL2.
- **Cross-platform**:
  - **raylib** supports alpha blending in software when OpenGL is disabled.
  - **Dear ImGui** can be combined with custom backends using software compositing.

All these tools allow frame-level composition using alpha without relying on GPU hardware.

8.2.11 Summary

Transparency and alpha blending are fundamental to 2D CPU-based rendering. By understanding the mathematical model and implementing optimized per-pixel operations, developers can produce high-quality visual effects without GPU support. Whether blending UI overlays, animating sprites, or building layered scenes, mastering CPU-side alpha blending is essential for real-time 2D graphics pipelines built entirely in software.

8.3 Pixel-Based Particle Effects

8.3.1 Introduction

Particle effects are essential in enhancing the visual appeal of interactive applications and games. Even without GPU support, a modern CPU can handle a wide range of 2D particle systems efficiently if designed with proper data structures, update loops, and memory usage patterns. Pixel-based particle effects involve small graphic elements—typically individual or grouped pixels—animated over time to simulate phenomena like fire, smoke, explosions, sparks, snow, or magical effects.
This section presents a complete strategy for implementing particle systems entirely on the CPU, suitable for both Windows and Linux systems after 2019, using standard and widely-supported software rendering APIs such as SDL2 or raw framebuffer manipulation.

8.3.2 Core Concepts of a CPU Particle System

A particle system consists of:

- A **particle emitter**: defines where and how particles are spawned.
- A **particle structure**: contains properties for each individual particle.
- An **update loop**: modifies the particle state each frame.
- A **render function**: draws the particle to a framebuffer.

Unlike GPU-based systems, all updates and rendering are performed in software and must be optimized to run within the CPU's frame budget.

8.3.3 Particle Data Structure

Each particle typically holds its position, velocity, color, lifetime, and any effect-specific data (like gravity influence or rotation). A basic structure in C++ might look like:

```
struct Particle {
    float x, y;             // Position
    float vx, vy;           // Velocity
    float life;             // Remaining life in seconds
    float totalLife;        // Total life duration
    uint8_t r, g, b, a;     // Color and alpha
    bool active;            // Is the particle currently alive
};
```

This structure is small and cache-friendly, which is critical for maintaining high performance on CPU.

8.3.4 Particle Emitter

The emitter is responsible for creating new particles, setting their initial properties such as direction, speed, and lifespan. Emission can be constant, burst-based, or user-triggered (e.g., on a mouse click or explosion event).

```
void emitParticles(std::vector<Particle>& particles, int count, float
↪   originX, float originY) {
    for (auto& p : particles) {
        if (!p.active && count > 0) {
            p.active = true;
            p.x = originX;
            p.y = originY;
            p.vx = (rand() % 100 - 50) / 50.0f;
            p.vy = (rand() % 100 - 50) / 50.0f;
            p.life = p.totalLife = (rand() % 1000) / 1000.0f + 0.5f;
            p.r = 255;
            p.g = 128;
            p.b = 0;
            p.a = 255;
            count--;
        }
    }
}
```

This function emits particles at `originX`, `originY` with randomized velocity and life.

8.3.5 Updating Particles

Each frame, the system must update all active particles. This includes applying motion, gravity, fading, or other behavior:

```
void updateParticles(std::vector<Particle>& particles, float deltaTime) {
    for (auto& p : particles) {
        if (!p.active) continue;

        p.x += p.vx * deltaTime;
        p.y += p.vy * deltaTime;
        p.life -= deltaTime;

        // Optional: apply gravity or drag
        p.vy += 50.0f * deltaTime; // gravity

        // Optional: fade out
        p.a = static_cast<uint8_t>(255 * (p.life / p.totalLife));

        if (p.life <= 0.0f) {
            p.active = false;
        }
    }
}
```

This logic is simple but powerful, and can be extended with rotation, scaling, color shifts, or perlin noise for more dynamic movement.

8.3.6 Rendering Particles

Rendering involves drawing each visible particle to the framebuffer. Since particles are typically small, rendering them as single pixels or small quads is sufficient.

```
void drawParticles(uint32_t* framebuffer, int fbWidth, int fbHeight,
↪   const std::vector<Particle>& particles) {
    for (const auto& p : particles) {
        if (!p.active) continue;

        int px = static_cast<int>(p.x);
        int py = static_cast<int>(p.y);

        if (px < 0  py < 0  px >= fbWidth  py >= fbHeight) continue;

        uint32_t dstColor = framebuffer[py * fbWidth + px];
        uint32_t srcColor = (p.a << 24)  (p.r << 16)  (p.g << 8) | p.b;
        framebuffer[py * fbWidth + px] = blendARGB(srcColor, dstColor);
    }
}
```

The `blendARGB()` function was defined in Section 8.2 and handles transparency during rendering.

8.3.7 Managing Performance

CPU-based particle systems must be carefully managed to prevent slowdowns:

- **Fixed-size pool**: Pre-allocate a vector of particles and reuse them instead of creating/destroying objects per frame.
- **Dead particle skipping**: Skip processing or drawing particles marked inactive.
- **Bounding optimizations**: Avoid rendering outside the screen.
- **SIMD** (e.g., AVX2): For advanced users, vectorize updates for thousands of particles.

On modern CPUs, a system can handle hundreds to thousands of particles at 60 FPS without difficulty if batching and spatial constraints are respected.

8.3.8 Example: Fireworks Effect

Here is a simplified firework burst using pixel-based particles:

```
void launchFirework(std::vector<Particle>& particles, float x, float y) {
    for (int i = 0; i < 100; ++i) {
        emitParticles(particles, 1, x, y);
    }
}
```

Each particle may fly out radially and fade over time, simulating an explosion. By adjusting the emitter's direction and color dynamically, different visual styles can be achieved (e.g., smoke, sparks, embers).

8.3.9 System Integration

- **Windows**: Use `SDL2` (2.0.10+) in software rendering mode, or `GDI+` with DIB sections for pixel access.
- **Linux**: Use `SDL2`, Xlib with `XPutImage`, or directly write to `/dev/fb0` if running in framebuffer console mode.
- **Cross-platform**: Libraries like `raylib` can be compiled without GPU support and used to build framebuffer-based renderers.

Each platform can support this particle system, provided that double-buffering and a vertical sync strategy (covered in Chapter 7) are in place.

8.3.10 Summary

Pixel-based particle systems allow for expressive and efficient visual effects even without GPU acceleration. By leveraging compact data structures, fixed update loops, and careful rendering, developers can simulate:

- Fire, smoke, and sparks
- Rain, snow, and wind trails
- Explosions, magic effects, and debris

These techniques are foundational for building dynamic, engaging graphics applications that are both performant and visually rich in a CPU-only environment.

8.4 Shadow and glow effects

8.4.1 Introduction

Shadow and glow effects are visual enhancements that contribute to depth, realism, and aesthetic appeal in 2D graphical applications. These effects, although often delegated to the GPU for real-time rendering, can also be implemented entirely on the CPU with careful algorithm design. In software-only rendering contexts—such as legacy systems, embedded platforms, or environments where GPU access is restricted—computing glow and shadow effects becomes a matter of pixel manipulation, convolution, and blending techniques.

This section explores practical methods for generating shadow and glow effects in real-time using the CPU, including drop shadows, outer glows, soft-edge effects, and light halos, all implemented efficiently with hardware and APIs on Windows and Linux platforms.

8.4.2 Conceptual Overview

Shadows and glows are often created as **blurred copies** of a source object, offset or extended to simulate distance or illumination. These effects are typically achieved by:

- **Duplicating** or sampling the object's silhouette (alpha mask)
- **Blurring** the mask using a convolution technique (box, Gaussian, etc.)
- **Tinting** the result with a color (black for shadows, colored for glows)
- **Blending** it beneath or around the original graphic

These steps are resource-intensive but can be made efficient through approximations, optimizations, and incremental computations.

8.4.3 Creating an Alpha Mask

Before generating a shadow or glow, we must first extract the shape of the object from its alpha channel. This is often done by iterating over the source image and creating a grayscale or single-channel alpha mask.

```
void createAlphaMask(const uint32_t* source, uint8_t* mask, int width,
 ↪  int height) {
    for (int i = 0; i < width * height; ++i) {
        uint8_t alpha = (source[i] >> 24) & 0xFF;
        mask[i] = alpha;
    }
}
```

This `mask` will serve as the base for the blur operation.

8.4.4 Blurring Techniques

1. **Box Blur**

   A box blur is a simple and fast approximation to Gaussian blur. It computes the average of neighboring pixels within a square kernel.

   ```
   void boxBlur(const uint8_t* input, uint8_t* output, int width, int
    ↪  height, int radius) {
       for (int y = 0; y < height; ++y) {
           for (int x = 0; x < width; ++x) {
               int sum = 0;
               int count = 0;

               for (int ky = -radius; ky <= radius; ++ky) {
                   for (int kx = -radius; kx <= radius; ++kx) {
   ```

```
                    int nx = x + kx;
                    int ny = y + ky;
                    if (nx >= 0 && ny >= 0 && nx < width && ny <
                    ↪  height) {
                        sum += input[ny * width + nx];
                        count++;
                    }
                }
            }

            output[y * width + x] = static_cast<uint8_t>(sum /
            ↪  count);
        }
    }
}
```

Multiple passes with smaller radii produce a smoother result.

2. **Gaussian Approximation**

   True Gaussian blur is computationally expensive. A common and efficient approximation is to perform **three box blurs** sequentially (known as the **triple box blur**), which closely resembles a Gaussian shape.

   This approach is acceptable in real-time CPU graphics with moderate resolutions (under 1080p) and benefits from caching and memory access optimizations.

8.4.5 Rendering Drop Shadows

To render a drop shadow behind a UI element or sprite:

1. Extract the alpha mask.

2. Blur the alpha mask.
3. Color the blurred result (commonly black or gray).
4. Blend the shadow with an offset below the source image.

```
void renderShadow(
    uint32_t* framebuffer, int fbWidth, int fbHeight,
    const uint8_t* blurredMask, int maskWidth, int maskHeight,
    int offsetX, int offsetY, uint32_t shadowColor
) {
    for (int y = 0; y < maskHeight; ++y) {
        for (int x = 0; x < maskWidth; ++x) {
            uint8_t alpha = blurredMask[y * maskWidth + x];
            if (alpha == 0) continue;

            int dx = x + offsetX;
            int dy = y + offsetY;

            if (dx >= 0 && dy >= 0 && dx < fbWidth && dy < fbHeight) {
                uint32_t dst = framebuffer[dy * fbWidth + dx];

                uint8_t r = (shadowColor >> 16) & 0xFF;
                uint8_t g = (shadowColor >> 8) & 0xFF;
                uint8_t b = shadowColor & 0xFF;
                uint32_t src = (alpha << 24)  (r << 16)  (g << 8) | b;

                framebuffer[dy * fbWidth + dx] = blendARGB(src, dst);
            }
        }
    }
}
```

This creates a soft, offset shadow beneath the image.

8.4.6 Creating Glow Effects

Glow effects follow the same process as shadow rendering but are drawn outward and surrounding the original image. The key differences are:

- **Color**: Glows are often colored (e.g., blue, yellow, white).
- **Placement**: Glows can be rendered *around* the image, not just offset.
- **Intensity**: May use multiple layers or larger blur radii.

To create a glow:

1. Extract and blur the alpha mask.
2. Tint the result with the desired glow color.
3. Blend it over the image (behind or in additive mode).

An additive blend can simulate light more effectively than standard alpha blending.

```
uint32_t addBlend(uint32_t src, uint32_t dst) {
    uint8_t sa = (src >> 24) & 0xFF;
    uint8_t sr = (src >> 16) & 0xFF;
    uint8_t sg = (src >> 8) & 0xFF;
    uint8_t sb = src & 0xFF;

    uint8_t dr = (dst >> 16) & 0xFF;
    uint8_t dg = (dst >> 8) & 0xFF;
    uint8_t db = dst & 0xFF;
    uint8_t r = std::min(255, sr + dr);
    uint8_t g = std::min(255, sg + dg);
    uint8_t b = std::min(255, sb + db);

    return (0xFF << 24)  (r << 16)  (g << 8) | b;
}
```

This function simulates light accumulation and is ideal for rendering magical glows, neon signs, or ambient lighting effects.

8.4.7 Performance Considerations

CPU-rendered shadow and glow effects are expensive due to the blur step. To manage this:

- **Use small blur radii** (3–5 pixels) for real-time rendering.
- **Cache blurred masks** for static elements and reuse them.
- **Downsample mask** before blurring and upscale during blending.
- **Thread parallelism**: Blur operations can be parallelized across multiple cores.

On modern multi-core CPUs, real-time rendering of UI drop shadows and basic glows at 60 FPS is feasible with reasonable constraints.

8.4.8 Integration with Systems

- **On Windows**
  - **SDL2** with software surfaces allows direct pixel manipulation and alpha blending.

  - **GDI+** provides `AlphaBlend` and blur operations but with limited control.

- **On Linux**
  - Framebuffer manipulation through `SDL2`, `XPutImage`, or Cairo in software mode.
  - Full support for threaded operations and manual memory buffers.

  These environments allow software compositors to integrate shadow/glow rendering into UI layers or scene composition systems.

8.4.9 Example Use Cases

- **Text shadows** in UI or menus
- **Glowing buttons or HUD elements**
- **Explosion halos** in games
- **Focus effects** on selected items
- **Light-beam effects** using radial blurs or conic gradients (via repeated glow rendering)

Each effect enhances clarity and depth, making even minimalistic UIs or games visually appealing without GPU assistance.

8.4.10 Summary

Shadow and glow effects are essential for polish and clarity in any 2D graphical environment. With CPU-only rendering, these can be implemented using alpha masks, convolution-based blurs, and layered blending techniques. While computationally expensive, optimizations in memory access, threading, and fixed-point math enable real-time effects on modern hardware. Understanding and implementing these techniques ensures that CPU-rendered interfaces and scenes remain visually competitive with their GPU-accelerated counterparts.

8.5 Procedural Textures and Noise (Perlin, Simplex)

8.5.1 Introduction

Procedural textures are algorithmically generated patterns that eliminate the need for preloaded image data, offering flexibility, compactness, and infinite variety in real-time rendering. In CPU-based graphics systems, procedural generation is a powerful tool to create dynamic visuals such as terrain, clouds, fire, marble, wood grain, water ripples, or background patterns. Noise functions, especially *Perlin noise* and *Simplex noise*, serve as the foundation for most procedural effects.
Unlike bitmap textures, procedural textures scale well to large resolutions and animations without occupying memory bandwidth or storage. This section explores how to implement and optimize procedural noise functions and use them effectively in 2D CPU-only environments, taking into account cross-platform capabilities on modern Windows and Linux systems.

8.5.2 Overview of Noise in 2D Graphics

Noise in graphics refers to a pseudo-random function with spatial coherence. That is, nearby input coordinates yield similar output values, making noise ideal for creating textures that resemble natural phenomena. The characteristics of good noise functions include:

- Continuity: No hard jumps or discontinuities.
- Pseudo-randomness: Appears random, but deterministic.
- Scale invariance: Works across zoom levels.
- Low computational cost (for real-time use).

Two widely-used noise algorithms in graphics programming are:

- **Perlin Noise (1983)** – A gradient-based noise function developed by Ken Perlin.
- **Simplex Noise (2001)** – An improved version by the same author, designed to reduce computational complexity in higher dimensions and avoid directional artifacts.

8.5.3 Perlin Noise (2D)

1. **Principle**

   Perlin noise generates a lattice of gradients at fixed grid points and interpolates between them based on the distance from the input point. The result is smooth, continuous noise that appears natural.

2. **Implementation**

   The implementation involves:

   (a) Creating a grid of gradients.

   (b) For a given input (x, y), determining the surrounding grid points.

   (c) Computing the dot product between the gradient and offset vectors.

   (d) Applying a fade function to smooth interpolation.

```
float fade(float t) {
    return t * t * t * (t * (t * 6 - 15) + 10);
}

float lerp(float a, float b, float t) {
    return a + t * (b - a);
}

float grad(int hash, float x, float y) {
    int h = hash & 3;
    float u = h < 2 ? x : y;
    float v = h < 2 ? y : x;
    return ((h & 1) ? -u : u) + ((h & 2) ? -2.0f * v : 2.0f * v);
}
```

A pseudo-random permutation array is used to select gradient directions:

```
int perm[512]; // initialized with 256-value permutation repeated
↪   twice
```

The final noise function computes the weighted average from four surrounding corners:

```
float perlin2D(float x, float y) {
    int xi = static_cast<int>(floor(x)) & 255;
    int yi = static_cast<int>(floor(y)) & 255;

    float xf = x - floor(x);
    float yf = y - floor(y);

    float u = fade(xf);
    float v = fade(yf);

    int aa = perm[perm[xi] + yi];
    int ab = perm[perm[xi] + yi + 1];
    int ba = perm[perm[xi + 1] + yi];
    int bb = perm[perm[xi + 1] + yi + 1];
    float x1 = lerp(grad(aa, xf, yf), grad(ba, xf - 1, yf), u);
    float x2 = lerp(grad(ab, xf, yf - 1), grad(bb, xf - 1, yf - 1),
     ↪  u);

    return (lerp(x1, x2, v) + 1.0f) / 2.0f;
}
```

3. **Application Example**

   Procedural cloud generation on a framebuffer:

```
void generateCloudTexture(uint32_t* framebuffer, int width, int
 ↪  height, float time) {
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            float fx = static_cast<float>(x) / 100.0f;
            float fy = static_cast<float>(y) / 100.0f;
            float n = perlin2D(fx + time * 0.1f, fy);

            uint8_t c = static_cast<uint8_t>(n * 255);
            framebuffer[y * width + x] = (255 << 24)  (c << 16)  (c
             ↪  << 8) | c;
        }
    }
}
```

   This loop simulates moving clouds using time as a noise offset.

8.5.4 Simplex Noise (2D)

1. **Advantages over Perlin**

- Fewer directions: Lower computational cost.
- No directional artifacts.
- Better suited for large-scale procedural textures.

2. **Mathematical Description**

   Simplex noise divides space into equilateral triangles (simplexes) instead of squares. Each point lies within a triangle formed by neighboring lattice points. The noise value is determined by gradient contributions from the corners of the triangle.

   A reference implementation is more complex than Perlin and typically pre-generated, but still feasible for CPU use.

   A performance-optimized 2D version adapted for software rendering avoids complex gradients and uses integer hashing.

3. **Use Case: Procedural Water Surface**

```
void generateWater(uint32_t* framebuffer, int width, int height,
 ↪  float time) {
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            float nx = static_cast<float>(x) / 50.0f;
            float ny = static_cast<float>(y) / 50.0f;
            float value = simplex2D(nx, ny + time * 0.2f);
            uint8_t blue = static_cast<uint8_t>(128 + 127 * value);

            framebuffer[y * width + x] = (255 << 24)  (0 << 16)  (0
             ↪  << 8) | blue;
        }
    }
}
```

This animation produces ripples that move and evolve over time.

8.5.5 Octaves and Fractal Noise

By combining multiple scaled layers of noise (called *octaves*), complex patterns emerge. Each octave adds detail at a higher frequency but lower amplitude.

```
float fractalNoise2D(float x, float y, int octaves, float persistence) {
    float total = 0;
    float frequency = 1.0f;
    float amplitude = 1.0f;
    float maxValue = 0;

    for (int i = 0; i < octaves; ++i) {
        total += perlin2D(x * frequency, y * frequency) * amplitude;
        maxValue += amplitude;

        amplitude *= persistence;
        frequency *= 2;
    }

    return total / maxValue;
}
```

Use this for terrain generation, wood grain, marble, or animated fire.

8.5.6 Integration with Software Rendering

several cross-platform software rendering libraries allow integration of procedural noise:

- **SDL2** (2.0.10+): Use `SDL_Surface` for direct pixel manipulation.
- **raylib**: CPU rendering with customizable framebuffers.

- **Custom DIB or framebuffer access**: For direct 32-bit pixel writing.

Use 32-bit RGBA surfaces in software mode for simplicity and cross-compatibility. Multi-core systems allow parallel generation of procedural textures per frame when needed.

8.5.7 Performance Notes

- **Cache gradients or permutation arrays** statically.
- **Avoid floating-point math** in inner loops if possible; fixed-point math can help.
- **Use temporal coherence**: Update procedural textures once every few frames if the effect doesn't require real-time updates.
- **Parallelism**: Partition the framebuffer into scanline or tile-based jobs and use threading for modern CPUs.

On CPUs with 4–16 cores, real-time procedural textures at 60 FPS are achievable with modest resolution.

8.5.8 Summary

Procedural textures and noise-based rendering open a vast space of creative and performance-efficient visual effects. Perlin and Simplex noise functions can simulate organic, natural-looking visuals on the CPU without storing any texture data. These techniques are invaluable for:

- Terrain and background generation
- Natural effects (clouds, water, fire)

- Animated patterns and motion graphics
- Texture variation and level-of-detail

In CPU-only environments, procedural noise is not just a fallback but a tool of artistic and technical power—scalable, dynamic, and resource-friendly.

Chapter 9

3D Graphics on the CPU

9.1 Basic 3D Math (Vectors, Matrices, Transformations)

9.1.1 Introduction

To render 3D graphics using the CPU, it is essential to understand the mathematics that underpins the entire rendering pipeline. Unlike 2D graphics, which operate within a flat coordinate space, 3D graphics involve an additional dimension and require transformations between multiple coordinate systems. This section focuses on the foundational mathematics of 3D graphics: vectors, matrices, and transformations.

This knowledge forms the bedrock for subsequent operations such as camera positioning, object manipulation, lighting calculations, and eventually rasterization—all of which are critical in a software-rendered 3D engine.

The examples and techniques provided here are written for C++ and C environments on modern Windows (Windows 10/11) and Linux (Ubuntu 20.04+), using standards, and are compatible with software rendering libraries like SDL2, raylib in software mode, or custom pixel buffers.

9.1.2 3D Vectors

1. **Definition**

   A vector in 3D space is a quantity that has both direction and magnitude. It is commonly represented by three components:

   ```
   V = (x, y, z)
   ```

   Vectors are used to represent positions, directions, velocities, normals, and more.

2. **Operations**

   The fundamental operations on vectors include:

   - **Addition**:
     Adds two vectors component-wise.
     `V + U = (x1 + x2, y1 + y2, z1 + z2)`
   - **Subtraction**:
     `V - U = (x1 - x2, y1 - y2, z1 - z2)`
   - **Scalar Multiplication**:
     `a * V = (a*x, a*y, a*z)`
   - **Dot Product**:
     Returns a scalar that represents the angle between two vectors.
     `V ⍰ U = x1*x2 + y1*y2 + z1*z2`
     Result is useful for lighting and projection.

- **Cross Product**:

  Produces a vector perpendicular to both inputs.

  `V × U = (y1*z2 - z1*y2, z1*x2 - x1*z2, x1*y2 - y1*x2)`

  Used heavily in computing surface normals and constructing basis vectors.

- **Length (Magnitude)**:

  `|V| = sqrt(x² + y² + z²)`

- **Normalization**:

  Converts a vector to unit length.

  `V̂ = V / |V|`

3. **C++ Example**

```cpp
struct Vec3 {
    float x, y, z;

    Vec3 operator+(const Vec3& v) const { return {x + v.x, y + v.y,
     ↪  z + v.z}; }
    Vec3 operator-(const Vec3& v) const { return {x - v.x, y - v.y,
     ↪  z - v.z}; }
    Vec3 operator*(float s) const { return {x * s, y * s, z * s}; }

    float dot(const Vec3& v) const { return x * v.x + y * v.y + z *
     ↪  v.z; }

    Vec3 cross(const Vec3& v) const {
        return {
            y * v.z - z * v.y,
            z * v.x - x * v.z,
            x * v.y - y * v.x
        };
    }
    float length() const { return std::sqrt(x*x + y*y + z*z); }

    Vec3 normalized() const {
        float len = length();
        return len > 0 ? *this * (1.0f / len) : *this;
    }
};
```

9.1.3 4x4 Matrices

1. **Purpose**

   In 3D graphics, most transformations are performed using 4x4 matrices. These matrices are used to:

   - Rotate objects
   - Scale objects
   - Translate (move) objects
   - Project 3D points to 2D screen space

   By using homogeneous coordinates `(x, y, z, w)`, we can represent all affine transformations uniformly with 4×4 matrices.

2. **Matrix Representation**

   A 4x4 matrix can be represented in row-major form:

```
m00 m01 m02 m03
m10 m11 m12 m13
m20 m21 m22 m23
m30 m31 m32 m33
```

3. **Matrix Multiplication**

   To transform a vector using a matrix:

```
Vec4 result;
result.x = m00*v.x + m01*v.y + m02*v.z + m03*v.w;
result.y = m10*v.x + m11*v.y + m12*v.z + m13*v.w;
result.z = m20*v.x + m21*v.y + m22*v.z + m23*v.w;
result.w = m30*v.x + m31*v.y + m32*v.z + m33*v.w;
```

   Matrix-matrix multiplication is performed similarly by applying the dot product between rows and columns.

9.1.4 Common Transformation Matrices

1. **Translation**

   Moves an object by a vector (tx, ty, tz):

```
1  0  0  tx
0  1  0  ty
0  0  1  tz
0  0  0   1
```

2. **Scaling**

Scales the object along each axis:

```
sx  0   0   0
0   sy  0   0
0   0   sz  0
0   0   0   1
```

3. **Rotation**

   Rotation around the X-axis (angle in radians):

```
1     0        0     0
0   cosθ    -sinθ   0
0   sinθ     cosθ   0
0     0        0     1
```

   Rotation around Y and Z axes follow similar structure.

9.1.5 Transformation Pipeline

3D rendering requires chaining several transformations:

1. **Model Matrix (Object space → World space)**
   Positions and orients the model in the world.

2. **View Matrix (World space → Camera space)**
   Moves the world relative to the camera (inverse of camera transform).

3. **Projection Matrix (Camera space → Clip space)**
   Applies perspective or orthographic projection.

4. **Viewport Transform (Clip space → Screen space)**
   Maps normalized device coordinates to pixel coordinates.

These matrices are typically multiplied together into a single transformation matrix known as the **Model-View-Projection (MVP)** matrix:

```
Mat4 MVP = Projection * View * Model;
```

Then each vertex in the model is transformed using:

```
Vec4 transformed = MVP * vertex;
```

Finally, if `w ≠ 1`, perform perspective divide:

```
Vec4 transformed = MVP * vertex;
```

9.1.6 Matrix Implementation in C++

```
struct Mat4 {
    float m[16]; // row-major

    static Mat4 identity() {
        return {1,0,0,0,  0,1,0,0,  0,0,1,0,  0,0,0,1};
    }

    static Mat4 translation(float tx, float ty, float tz) {
        return {1,0,0,tx,  0,1,0,ty,  0,0,1,tz,  0,0,0,1};
    }

    static Mat4 scale(float sx, float sy, float sz) {
        return {sx,0,0,0,  0,sy,0,0,  0,0,sz,0,  0,0,0,1};
    }

    static Mat4 rotationX(float angle) {
        float c = cosf(angle), s = sinf(angle);
        return {1,0,0,0,  0,c,-s,0,  0,s,c,0,  0,0,0,1};
    }

    // Multiply matrix * matrix
    Mat4 operator*(const Mat4& b) const {
        Mat4 result = {};
        for (int row = 0; row < 4; ++row)
            for (int col = 0; col < 4; ++col)
                for (int k = 0; k < 4; ++k)
                    result.m[row*4 + col] += m[row*4 + k] * b.m[k*4 +
                     ↪  col];
        return result;
    }

    // Multiply matrix * vector
    Vec4 operator*(const Vec4& v) const {
        return {
            m[0]*v.x + m[1]*v.y + m[2]*v.z + m[3]*v.w,
            m[4]*v.x + m[5]*v.y + m[6]*v.z + m[7]*v.w,
            m[8]*v.x + m[9]*v.y + m[10]*v.z + m[11]*v.w,
            m[12]*v.x + m[13]*v.y + m[14]*v.z + m[15]*v.w
        };
    }
};
```

This structure enables composition of all transformations necessary in a CPU 3D pipeline.

9.1.7 Real-World Application in Software Rendering

Modern software renderers such as those built with SDL2 or raw framebuffers on Linux rely on this math to process vertices before rasterizing them to a software surface. While these operations are compute-heavy, CPUs with SIMD (e.g., AVX2/AVX-512) and threading (e.g., OpenMP or C++17 threads) allow high-speed per-vertex and per-object processing.

For scenes under 10,000 triangles, real-time rendering at 30–60 FPS is feasible on mid-range CPUs using these techniques.

9.1.8 Summary

Understanding 3D math is the first critical step to building a functional 3D software renderer. Vectors, matrices, and transformations are the foundation upon which all geometry manipulation and camera movement are based. This chapter provides the basis for transforming 3D models into screen-space projections. In the next sections, we will build on this to discuss camera systems, projection, clipping, and rasterization.

9.2 Drawing Wireframes

9.2.1 Introduction

In 3D graphics, a **wireframe** is a visual representation of a 3D model where only the edges of polygons (usually triangles) are drawn. This method provides a lightweight way to visualize geometry and is often used during development, debugging, or for stylized rendering in simulation software.
In software rendering, especially when GPU acceleration is unavailable or intentionally avoided, rendering wireframes on the CPU serves as a foundational technique. It provides a means to test the 3D math and transformation pipeline without the complexity of full rasterization and shading. This section will guide you through the practical implementation of wireframe rendering using modern CPU-only methods and tools, compatible with operating systems such as Windows 10/11 and Linux (Ubuntu 20.04+), utilizing libraries like SDL2 or simple framebuffer access.

9.2.2 Overview of the Wireframe Rendering Pipeline

Wireframe rendering follows a simplified version of the traditional 3D rendering pipeline:

1. **Model Definition** – 3D objects defined by vertices and edges.
2. **Transformation** – Apply model, view, and projection matrices.
3. **Clipping and Perspective Division** – Ensure visibility and depth handling.
4. **Viewport Mapping** – Convert normalized coordinates to screen pixels.
5. **Line Drawing** – Render edges using a line algorithm (such as Bresenham's).

Unlike full rasterization, this process avoids filling triangles and instead focuses only on rendering their outlines.

9.2.3 Data Structures

1. **Vertex and Edge Structures**

```
struct Vec3 {
    float x, y, z;
};

struct Vec4 {
    float x, y, z, w;
};

struct Edge {
    int v0, v1; // indices into the vertex array
};

struct Triangle {
    int v0, v1, v2; // for optional triangle outline support
};
```

For triangle-only models, you can extract edges by processing each triangle:

```
struct Mesh {
    std::vector<Vec3> vertices;
    std::vector<Edge> edges;
};
```

Optionally use a hash set to prevent duplicate edges.

9.2.4 3D to 2D Projection

Once the vertices are defined in object space, they must be transformed using the **Model-View-Projection (MVP)** matrix pipeline described in Section 9.1.

```
Vec4 clipSpaceVertex = MVP * Vec4{v.x, v.y, v.z, 1.0f};
```

Then perform the **perspective divide**:

```
if (clipSpaceVertex.w != 0.0f) {
    clipSpaceVertex.x /= clipSpaceVertex.w;
    clipSpaceVertex.y /= clipSpaceVertex.w;
    clipSpaceVertex.z /= clipSpaceVertex.w;
}
```

Then, map normalized device coordinates (NDC) to screen space:

```
int screenX = (int)((clipSpaceVertex.x * 0.5f + 0.5f) * screenWidth);
int screenY = (int)((1.0f - (clipSpaceVertex.y * 0.5f + 0.5f)) *
 ↪  screenHeight);
```

Note: Y is flipped because screen coordinates usually have the origin at the top-left.

9.2.5 Line Drawing Algorithm

Use **Bresenham's line algorithm** or a variant for floating-point coordinates for better precision:

```
void drawLine(int x0, int y0, int x1, int y1, uint32_t color, uint32_t*
 ↪  framebuffer, int width) {
    int dx = abs(x1 - x0), dy = -abs(y1 - y0);
    int sx = x0 < x1 ? 1 : -1;
    int sy = y0 < y1 ? 1 : -1;
    int err = dx + dy;

    while (true) {
        if (x0 >= 0 && y0 >= 0 && x0 < width && y0 < screenHeight)
            framebuffer[y0 * width + x0] = color;

        if (x0 == x1 && y0 == y1) break;
        int e2 = 2 * err;
        if (e2 >= dy) { err += dy; x0 += sx; }
        if (e2 <= dx) { err += dx; y0 += sy; }
    }
}
```

This function writes directly to a linear pixel buffer. The caller should ensure synchronization and boundary checks.

9.2.6 Putting It All Together

```
void renderWireframe(const Mesh& mesh, const Mat4& MVP, uint32_t*
 ↪  framebuffer, int width, int height, uint32_t color) {
    std::vector<Vec2> projected(mesh.vertices.size());

    // Transform and project all vertices
    for (size_t i = 0; i < mesh.vertices.size(); ++i) {
        Vec4 v = MVP * Vec4{mesh.vertices[i].x, mesh.vertices[i].y,
         ↪  mesh.vertices[i].z, 1.0f};
        if (v.w != 0.0f) {
            v.x /= v.w;
            v.y /= v.w;
        }
        projected[i].x = (v.x * 0.5f + 0.5f) * width;
        projected[i].y = (1.0f - (v.y * 0.5f + 0.5f)) * height;
    }

    // Draw all edges
    for (const auto& edge : mesh.edges) {
        drawLine(
            (int)projected[edge.v0].x,
            (int)projected[edge.v0].y,
            (int)projected[edge.v1].x,
            (int)projected[edge.v1].y,
            color,
            framebuffer,
            width
        );
    }
}
```

9.2.7 Optimization Considerations

- **Clipping**: Clipping lines entirely outside the view frustum improves performance and prevents screen overdraw. Implementing Cohen-Sutherland or Liang-Barsky algorithms helps.

- **Depth Sorting**: While not necessary for wireframes, you may choose to sort meshes or edges by average Z-depth to simulate visibility or perform backface culling.

- **Multithreading**: Wireframe rendering can be parallelized by drawing edge batches using modern C++ threads or OpenMP.

- **Double Buffering**: Always use a back buffer and then copy or present it atomically to the screen to avoid tearing.

9.2.8 Example Output

Using a wireframe cube mesh, you should see an outline that reflects camera rotation and projection. This confirms the correctness of your transformation pipeline.

9.2.9 Application and Use Cases

- **Game Engine Development** – Visualize collision volumes or level geometry.
- **Simulation Tools** – Render structural or spatial models with minimal CPU overhead.
- **Debugging Geometry** – View raw mesh shapes without full rasterization.
- **Educational Software** – Teach basic 3D geometry and graphics concepts.

9.2.10 Summary

Wireframe rendering is a simple yet powerful technique that confirms your 3D math and pipeline are functional. It's CPU-efficient, easy to implement, and highly useful in debugging or non-photorealistic rendering. In the next section, we will move beyond wireframes to triangle rasterization, depth buffering, and shading—all done entirely on the CPU.

9.3 Perspective Projection

9.3.1 Introduction

Perspective projection is one of the cornerstones of realistic 3D rendering. It is the mathematical technique by which 3D coordinates in a virtual world are projected onto a 2D screen in such a way that objects appear smaller as they move farther from the viewer—mimicking how human vision works. Unlike orthographic projection, which preserves size regardless of depth, perspective projection introduces **depth foreshortening**, providing a critical cue to spatial realism in visual scenes.
In this section, we explore how to construct and apply a perspective projection matrix on the CPU, how the process integrates with the overall Model-View-Projection (MVP) transformation pipeline, and how to handle the necessary normalization and mapping to screen space—all while using only CPU resources with no reliance on GPU pipelines.

9.3.2 Coordinate Systems and the Pipeline

In a standard CPU-based 3D rendering pipeline, coordinates go through the following transformations:

1. **Model Space → World Space**
   Applies object transformation (scaling, rotation, translation).

2. **World Space → View Space (Camera Space)**
   Transforms coordinates relative to the camera's position and orientation.

3. **View Space → Clip Space (via Perspective Projection Matrix)**
   Applies perspective foreshortening using a projection matrix.

4. **Clip Space → Normalized Device Coordinates (NDC)**

Performs **homogeneous division** by `w` to bring coordinates into canonical cube (-1 to 1).

5. **NDC → Screen Space**
   Maps the 3D result into 2D screen coordinates.

Perspective projection happens between steps 3 and 4, and is crucial to achieve realistic depth and field-of-view simulation.

9.3.3 Perspective Projection Matrix

1. **General Form**

   For a right-handed coordinate system, the standard perspective projection matrix used for a vertical field of view is:

```
[ f/aspect   0        0                      0 ]
[ 0          f        0                      0 ]
[ 0          0     (zfar+znear)/(znear-zfar)
↪  (2*zfar*znear)/(znear-zfar) ]
[ 0          0       -1                      0 ]
```

   Where:

   - `f = 1 / tan(fov / 2)`
   - `aspect = width / height`
   - `znear` and `zfar` define the clipping planes
   - `fov` is the vertical field of view in radians

   After this transformation, the `w` component of each vertex will be non-zero and used in the perspective divide step.

2. **CPU Implementation in C++**

```cpp
Mat4 makePerspective(float fovYRadians, float aspect, float znear,
 ↪  float zfar) {
    float f = 1.0f / tanf(fovYRadians / 2.0f);
    Mat4 result = {};
    result[0][0] = f / aspect;
    result[1][1] = f;
    result[2][2] = (zfar + znear) / (znear - zfar);
    result[2][3] = (2 * zfar * znear) / (znear - zfar);
    result[3][2] = -1.0f;
    return result;
}
```

This matrix assumes a row-major layout and multiplies a 4D vector `(x, y, z, 1)`.

9.3.4 Perspective Divide

After a vertex is multiplied by the MVP matrix and reaches clip space, we apply the **perspective divide**, also known as homogeneous division:

```cpp
Vec4 clip = MVP * Vec4{x, y, z, 1.0f};

if (clip.w != 0.0f) {
    clip.x /= clip.w;
    clip.y /= clip.w;
    clip.z /= clip.w;
}
```

The resulting `(x, y, z)` is now in **Normalized Device Coordinates** (NDC), typically ranging from -1 to 1 on each axis.

9.3.5 Mapping to Screen Space

NDC coordinates are then mapped to screen coordinates:

```
int screenX = (int)((clip.x * 0.5f + 0.5f) * screenWidth);
int screenY = (int)((1.0f - (clip.y * 0.5f + 0.5f)) * screenHeight);
```

- The `0.5` shift centers the coordinates.
- The Y-axis is flipped because screen coordinates start from the top-left.

Z-values may be stored in a depth buffer for depth sorting or hidden surface removal (covered in a later section).

9.3.6 Field of View and Clipping Planes

1. **Choosing Field of View**

   A typical `fovY` value is 45 to 60 degrees (in radians: `π/4` to `π/3`). Larger values create a wider field of view but introduce more distortion at the edges.

2. **Near and Far Planes**

   - `znear` should never be 0.
   - A good practice is to keep `znear` as far from 0 as possible (e.g., 0.1f) to avoid depth precision loss.
   - The `zfar/znear` ratio should be kept within reasonable limits (e.g., 1000 or less) for floating-point precision.

9.3.7 Example: Perspective Projecting a Cube

Given a cube defined in model space, apply the MVP pipeline:

```
Mat4 model = makeTranslation(0, 0, -5);
Mat4 view = makeLookAt(cameraPos, target, up);
Mat4 proj = makePerspective(fovY, aspect, znear, zfar);
Mat4 mvp = proj * view * model;

for (const auto& v : cubeVertices) {
    Vec4 clip = mvp * Vec4{v.x, v.y, v.z, 1.0f};
    clip = perspectiveDivide(clip);
    Vec2 screen = mapToScreen(clip, screenWidth, screenHeight);
    plot(screen.x, screen.y);
}
```

This CPU-only process simulates the same projection GPU pipelines would compute internally, but explicitly and manually.

9.3.8 Optimizations and Tips

- **SIMD Math Libraries**: Use libraries such as `glm` or your own SIMD-powered matrix library for faster computation.
- **Precompute MVP**: Combine model, view, and projection into one matrix before drawing.
- **Z-Clipping**: Implement near-plane and far-plane clipping before the perspective divide to avoid invalid results (division by zero or visual artifacts).
- **Z Buffering**: While not required for wireframe, this is critical when triangle rasterization is introduced.

9.3.9 Practical Use Cases in CPU-Only Environments

- **Embedded Systems**: For devices without a GPU but requiring 3D visual feedback.
- **Retro Game Engines**: Re-creating the style and architecture of older platforms like DOS or early consoles.
- **Education**: Demonstrating projection math without relying on black-box GPU behavior.
- **Debug Visualization**: Offering geometric overlays in development tools where speed trumps realism.

9.3.10 Summary

Perspective projection is a crucial mathematical transformation for realistic 3D rendering. This section has detailed its structure, implementation, and integration within a CPU-only rendering pipeline. Mastery of this transformation ensures a solid foundation for further steps like triangle rasterization, z-buffering, lighting, and texture mapping—each of which builds upon the accurate projection of 3D geometry into 2D screen space.

In the next section, we will explore triangle rasterization and depth buffering on the CPU to move from outline-based rendering to solid 3D graphics.

9.4 Triangle Rasterization (Including Barycentric Coordinates)

9.4.1 Introduction

Triangle rasterization is the final and most crucial stage of 3D rendering before shading and final composition. After the projection and perspective divide, 3D triangles must be filled on the 2D screen in a way that accurately represents their area, interpolates vertex attributes (such as depth, color, texture coordinates), and respects occlusion. Unlike wireframes, triangle rasterization produces solid surfaces—enabling realism in 3D rendering.

In GPU pipelines, this stage is executed in hardware with highly parallelized architectures. On the CPU, we implement rasterization through software techniques. One of the most robust and precise methods uses **barycentric coordinates**, which allow for correct inside-triangle testing and smooth attribute interpolation.

This section provides a detailed look at the triangle rasterization process using the CPU, covering:

- Edge function and scanline-based methods
- Barycentric coordinate method
- Interpolating attributes (depth, color, texture coordinates)
- Clipping and culling
- Z-buffer handling

9.4.2 Triangle Setup and Coordinate Space

1. **Screen-Space Triangle**

After projecting a 3D triangle through the Model-View-Projection (MVP) matrix and applying the perspective divide, the triangle's vertices are in **Normalized Device Coordinates (NDC)**. These are mapped to screen coordinates:

```
Vec2 screenPos = {
    (ndc.x * 0.5f + 0.5f) * screenWidth,
    (1.0f - (ndc.y * 0.5f + 0.5f)) * screenHeight
};
```

Each triangle is now defined by three 2D points `(x0, y0), (x1, y1), (x2, y2)` and their associated attributes such as depth `z`, color, or texture coordinates.

9.4.3 Barycentric Coordinates

Barycentric coordinates describe a point within a triangle as a weighted combination of its three vertices. They are used for:

- Determining if a point lies inside a triangle
- Interpolating attributes across the triangle surface

1. **Definition**

   Given a triangle with vertices `A`, `B`, and `C`, any point `P` inside the triangle can be written as:

   ```
   P = αA + βB + γC
   ```

   With the constraints:

```
α + β + γ = 1
0 ≤ α, β, γ ≤ 1
```

2. **Computing Barycentric Coordinates**

   A fast and robust method involves computing the **signed area** of sub-triangles:

```
float edgeFunction(Vec2 a, Vec2 b, Vec2 c) {
    return (c.x - a.x) * (b.y - a.y) - (c.y - a.y) * (b.x - a.x);
}
```

   Given a pixel point `P`, and triangle vertices `A`, `B`, `C`, the barycentric coordinates are:

```
// Interpolate using w (clip space inverse depth)
float recipW₀ = 1.0f / w₀;
float recipW₁ = 1.0f / w₁;
float recipW₂ = 1.0f / w₂;

float sum = α * recipW₀ + β * recipW₁ + γ * recipW₂;
float uvX = (α * u₀ * recipW₀ + β * u₁ * recipW₁ + γ * u₂ * recipW₂)
 ↪  / sum;
```

   If all weights are 0, the point lies inside the triangle.

9.4.4 Rasterization Loop

The screen-space bounding box of the triangle is computed to limit rasterization to a minimal area:

```
int minX = max(0, floor(min3(v0.x, v1.x, v2.x)));
int maxX = min(screenWidth - 1, ceil(max3(v0.x, v1.x, v2.x)));
int minY = max(0, floor(min3(v0.y, v1.y, v2.y)));
int maxY = min(screenHeight - 1, ceil(max3(v0.y, v1.y, v2.y)));
```

Loop over this bounding box and evaluate each pixel:

```
for (int y = minY; y <= maxY; ++y) {
    for (int x = minX; x <= maxX; ++x) {
        Vec2 P = {x + 0.5f, y + 0.5f}; // center of pixel

        float w0 = edgeFunction(v1, v2, P);
        float w1 = edgeFunction(v2, v0, P);
        float w2 = edgeFunction(v0, v1, P);

        if (w0 >= 0 && w1 >= 0 && w2 >= 0) {
            float α = w0 / area;
            float β = w1 / area;
            float γ = w2 / area;

            // Interpolate depth
            float z = α * z0 + β * z1 + γ * z2;

            // Z-buffer test
            if (z < zBuffer[y * width + x]) {
                zBuffer[y * width + x] = z;

                // Interpolate color or texture coords here
                Color finalColor = interpolateColor(α, β, γ, color0,
                 ↪  color1, color2);

                framebuffer[y * width + x] = finalColor;
            }
        }
    }
}
```

9.4.5 Interpolating Attributes

Besides depth (`z`), barycentric weights can interpolate:

- **Color**: Per-vertex RGB or RGBA
- **Texture Coordinates**: `u`, `v` values used in texture sampling
- **Normals**: For lighting in software shaders

The interpolated result is:

```
Value = α * attr0 + β * attr1 + γ * attr2;
```

When attributes are non-linear in screen space (e.g. UVs after perspective), **perspective-correct interpolation** is required:

```
// Interpolate using w (clip space inverse depth)
float recipW0 = 1.0f / w0;
float recipW1 = 1.0f / w1;
float recipW2 = 1.0f / w2;

float sum = α * recipW0 + β * recipW1 + γ * recipW2;
float uvX = (α * u0 * recipW0 + β * u1 * recipW1 + γ * u2 * recipW2) /
 ↪  sum;
```

9.4.6 Optimizations

- **Edge function deltas** can be precomputed to avoid recomputing per pixel.
- **Integer-based barycentric tests** are possible for speed and precision.
- **Fixed-point math** may offer performance gains on embedded or limited CPUs.
- Use **tile-based rendering** to improve cache locality.

9.4.7 CPU Rasterizer in Practice

- **Use Case Example**

  A minimal rasterizer in C++ using only the standard library and custom math functions can handle:

  - Transforming vertex positions
  - Clipping and projecting
  - Triangle rasterization with barycentric coordinates
  - Depth testing with a simple Z-buffer

  Modern C++ compilers like Clang 16+ and GCC 12+ can optimize this pipeline with aggressive inlining, vectorization, and memory alignment.

  Multithreading (via `std::thread` or `std::async`) allows distributing triangle rendering across cores, with each thread managing a tile or row of the screen.

9.4.8 Summary

Triangle rasterization on the CPU, while more computationally intensive than wireframe rendering, forms the foundation of real 3D graphics. Using barycentric coordinates allows accurate interpolation of attributes like depth, color, and texture coordinates while providing a mathematically robust way to test point-in-triangle inclusion.

While traditionally handled by GPUs, software-based triangle rasterizers serve critical roles in embedded systems, software renderers, educational tools, and game engines targeting retro or minimal hardware.

In the next section, we will examine shading and lighting models to bring visual richness and depth to rasterized triangles using entirely CPU-side computation.

9.5 Z-Buffering in Software

9.5.1 Introduction

Z-buffering, also known as depth buffering, is a critical technique in 3D graphics that ensures proper visibility and occlusion of overlapping surfaces. It allows the rendering system to determine which fragments (pixels) should be visible based on their depth relative to the viewer. While hardware graphics APIs and GPUs handle Z-buffering internally, this section focuses on building a **software Z-buffer** system, an essential component in any CPU-based rendering pipeline.

In a software renderer—especially one implemented without GPU assistance—Z-buffering is entirely manual. This involves allocating a dedicated buffer to track the depth of each pixel and updating it during rasterization to ensure correct rendering order.

This section covers:

- The theory behind Z-buffering
- Data structures and memory layout
- Depth comparisons during rasterization
- Common pitfalls and precision considerations
- Integration with triangle rasterization

9.5.2 What is Z-Buffering?

In 3D rendering, multiple objects can project onto the same screen pixel. The Z-buffer stores the depth value (Z-coordinate) of the nearest pixel rendered so far at each screen location. When a new fragment is rendered, its depth is compared to the stored value:

- If it is closer to the camera (smaller Z), the fragment is drawn and the Z-buffer is updated.
- Otherwise, the fragment is discarded.

This guarantees correct occlusion—closer objects obscure farther ones.

9.5.3 Depth Representation

1. **Normalization**

   After applying the perspective divide, Z-values typically lie in the range [0.0, 1.0], where 0.0 represents the near clipping plane and 1.0 represents the far plane.

2. **Data Types**

   In software implementations, the Z-buffer can be implemented as:

   - **`float` array**: Offers high precision and easy compatibility with normalized depth.
   - **`uint16_t` or `uint32_t`**: Useful for fixed-point depth (common in older or memory-constrained systems).

   Using `float` is preferred for general-purpose and modern CPUs due to hardware floating-point acceleration and simplicity.

9.5.4 Z-Buffer Data Structure

For a framebuffer of width × height, the Z-buffer is a parallel array:

```
float* zBuffer = new float[width * height];
```

It should be initialized once per frame, typically to a value representing the farthest depth:

```
for (int i = 0; i < width * height; ++i)
    zBuffer[i] = 1.0f; // Maximum depth, meaning farthest possible
```

During rasterization, each pixel fragment includes an interpolated `z` value. Before plotting, a comparison is made:

```
int index = y * width + x;
if (z < zBuffer[index]) {
    zBuffer[index] = z;
    framebuffer[index] = pixelColor;
}
```

This operation is the crux of software Z-buffering.

9.5.5 Depth Interpolation

Z-values must be interpolated across the surface of the triangle. Using **barycentric coordinates**, the interpolated depth at each pixel is:

```
float zInterpolated = α * z0 + β * z1 + γ * z2;
```

However, this linear interpolation is only accurate in screen space. To ensure correctness under perspective projection, **perspective-correct interpolation** is necessary:

1. **Perspective-Correct Depth Interpolation**

Let `w0`, `w1`, and `w2` be the clip-space `w` components for the triangle's vertices. The perspective-correct interpolated `z` is computed as:

```
float invW0 = 1.0f / w0;
float invW1 = 1.0f / w1;
float invW2 = 1.0f / w2;

float sumWeights = α * invW0 + β * invW1 + γ * invW2;
float zCorrected = (α * z0 * invW0 + β * z1 * invW1 + γ * z2 *
↪  invW2) / sumWeights;
```

This method ensures that the interpolated depth is accurate for non-planar surfaces or near the edges of the view frustum.

9.5.6 Handling Depth Precision and Artifacts

1. **Floating-Point Precision**

   Even though `float` provides sufficient precision for most use cases, when rendering very large or very small scenes, **Z-fighting** may occur. This manifests as flickering pixels where surfaces are too close in depth.

2. **Depth Range Optimization**

   To minimize depth precision loss:

   - Use the **smallest possible near plane**.
   - Avoid very large far planes unless needed.
   - Implement depth buffer linearization if the non-linear nature of the depth values becomes problematic.

3. **Reverse Z**

   Modern engines sometimes use reversed Z-buffering, where near values map to 1.0 and far values to 0.0, improving floating-point distribution. This is possible by negating Z during projection and inverting depth tests.

   In software renderers, this is an advanced technique and may be deferred for later optimization.

9.5.7 Z-Buffer Memory Optimization

On systems with limited memory:

- **Tiling** the screen into smaller blocks allows partial depth buffer clearing and caching.
- **16-bit depth buffers** may be used when visual fidelity allows.

For multicore CPUs, **thread-local Z-buffers** for tiles reduce contention and cache invalidation. A final merge stage consolidates the rendered tiles into the global framebuffer.

9.5.8 Integration with Rasterizer

Combining Z-buffering with triangle rasterization (as covered in Section 9.4) yields the following inner loop:

```
if (w0 >= 0 && w1 >= 0 && w2 >= 0) {
    float invW0 = 1.0f / vertex0.w;
    float invW1 = 1.0f / vertex1.w;
    float invW2 = 1.0f / vertex2.w;
    float weightSum = α * invW0 + β * invW1 + γ * invW2;
    float z = (α * z0 * invW0 + β * z1 * invW1 + γ * z2 * invW2) /
    ↪  weightSum;

    if (z < zBuffer[index]) {
        zBuffer[index] = z;
        framebuffer[index] = shadedColor;
    }
}
```

This ensures that only the front-most pixels are written, preserving the correct visibility of 3D surfaces.

9.5.9 Debugging and Visualization

To verify Z-buffer correctness:

- Render the depth buffer as a grayscale image where black = near and white = far.
- Inspect scenes for missing or flickering geometry (common signs of Z-buffer errors).
- Clamp Z values between 0 and 1 to avoid underflow or overflow issues.

9.5.10 Summary

Z-buffering is indispensable in 3D rendering for handling visibility. In a CPU-only environment, a robust software Z-buffer allows correct depth handling and enables complex 3D scenes with overlapping geometry. Despite the absence of GPU acceleration, modern multicore CPUs can handle software depth buffering efficiently when carefully designed.

This section lays the foundation for adding shading, texture mapping, and lighting effects in the software pipeline, which we will begin addressing in Chapter 10.

Chapter 10

Optimizations

10.1 Using Fixed-Point Math

10.1.1 Introduction

In CPU-based graphics programming, efficient arithmetic operations are paramount for achieving real-time performance, especially on limited or embedded hardware where floating-point units (FPUs) may be slow, absent, or power-hungry. **Fixed-point math** provides a deterministic, often faster alternative to floating-point calculations by representing fractional numbers using integers combined with implicit scaling.
This section explores the principles, benefits, and practical applications of fixed-point math in software rendering pipelines, particularly relevant to 2D and 3D graphics computations executed purely on the CPU.

10.1.2 Fundamentals of Fixed-Point Arithmetic

1. **Representation**

A fixed-point number stores its value as an integer but treats it as a scaled value relative to a fixed radix point (decimal point equivalent in base 2). For example, a **Q16.16** fixed-point format uses a 32-bit integer where:

- The upper 16 bits represent the integer part
- The lower 16 bits represent the fractional part

A value `x` in Q16.16 is stored as:

```
fixed_x = int(x * 2^16)
```

This allows fractional precision while enabling the use of fast integer arithmetic.

2. **Formats**

   Common fixed-point formats vary by bit allocation:

   Fixed-Point Format Characteristics

Format	Total bits	Integer bits	Fractional bits	Range	Resolution
Q8.8	16	8	8	-128 to 127.996	0.00390625
Q16.16	32	16	16	-32768 to 32767.9999847	1.5258789e-5
Q24.8	32	24	8	Very large integer range	0.00390625

   Choice depends on application precision requirements and integer size of the target CPU.

10.1.3 Advantages of Fixed-Point Math in CPU Graphics

- **Deterministic performance:** Integer operations are predictable and often faster than floating-point on CPUs lacking FPU acceleration or when avoiding costly floating-point pipeline stalls.
- **Lower power consumption:** Fixed-point operations consume less energy, essential for battery-powered or embedded systems.
- **Simpler hardware:** Enables deployment on microcontrollers or legacy CPUs without FPUs.
- **Avoids floating-point rounding errors:** Fixed-point math yields consistent precision across platforms without IEEE floating-point idiosyncrasies.

10.1.4 Basic Fixed-Point Operations

Assuming a Q16.16 format, the fundamental operations are implemented as follows:

- **Addition/Subtraction:** Direct integer add/subtract as both operands share scaling.

```
int32_t fixed_add(int32_t a, int32_t b) {
    return a + b;
}
```

- **Multiplication:** Requires scaling correction by right-shifting the result.

```
int32_t fixed_mul(int32_t a, int32_t b) {
    int64_t temp = (int64_t)a * (int64_t)b; // 64-bit to avoid overflow
    return (int32_t)(temp >> 16);            // Shift right by fractional
     ↪  bits
}
```

- **Division:** Left-shift the dividend to preserve precision before division.

```
int32_t fixed_div(int32_t a, int32_t b) {
    int64_t temp = ((int64_t)a << 16) / b;
    return (int32_t)temp;
}
```

10.1.5 Using Fixed-Point Math in Graphics Algorithms

1. **Coordinate Transformations**

   Transforming vertices or pixels often involves matrix multiplications and translations. Replacing floating-point operations with fixed-point math reduces overhead in:

   - Scaling and rotating 2D or 3D coordinates
   - Translating points in space
   - Calculating intersections or distances

   Example: A 2D rotation by angle   in Q16.16:

```c
// Precompute sine and cosine in fixed-point
int32_t cos_theta = float_to_fixed(cos(theta));
int32_t sin_theta = float_to_fixed(sin(theta));

int32_t x_new = fixed_mul(x, cos_theta) - fixed_mul(y, sin_theta);
int32_t y_new = fixed_mul(x, sin_theta) + fixed_mul(y, cos_theta);
```

2. **Rasterization and Scanline Algorithms**

   Rasterizing triangles or tilemaps benefits from fixed-point incremental arithmetic:

   - Calculating edge slopes as fixed-point increments allows integer-only pixel stepping.
   - Scanline fills use fixed-point interpolation for texture coordinates and colors.
   - Barycentric coordinates can be approximated with fixed-point for efficient point-in-triangle tests.

3. **Depth Buffering**

   While depth values are often stored as floats, fixed-point depth buffers are feasible, especially on embedded CPUs:

   - Store depth as Q16.16 for improved precision control.
   - Perform depth comparisons as integer comparisons, which are faster than floating-point.

10.1.6 Challenges and Considerations

- **Range vs. Precision Tradeoff:** Fixed-point formats must balance integer range and fractional precision. Overflow and underflow risks exist if chosen incorrectly.

- **Conversion Overhead:** Converting between fixed-point and floating-point (for interfacing with libraries or APIs) incurs some cost.
- **Complex Math:** Functions like division, square root, or trigonometric functions require approximations or lookup tables, potentially complicating implementation.
- **Code Complexity:** Fixed-point arithmetic code can be less intuitive than floating-point, increasing maintenance challenges.

10.1.7 Practical Tools and Libraries

Modern CPU toolchains and libraries support fixed-point operations with optimized intrinsics and SIMD instructions:

- **libfixmath** (actively maintained): A portable fixed-point math library in C supporting Q16.16 and other formats.
- **ARM CMSIS-DSP**: For ARM Cortex-M CPUs, includes fixed-point vector math optimized for embedded graphics.
- **Compiler intrinsics:** Modern compilers like GCC and Clang provide built-in support for saturated integer arithmetic and intrinsics useful in fixed-point math.

These tools, coupled with multithreaded designs (`std::thread`, OpenMP), enable real-time fixed-point graphics on current CPU platforms.

10.1.8 Summary

Fixed-point math remains a valuable optimization technique in CPU-only graphics programming, particularly when floating-point performance is constrained. By carefully choosing formats and applying fixed-point arithmetic to coordinate transformations, rasterization, and depth buffering, developers can achieve predictable, efficient rendering pipelines suitable for embedded and power-sensitive environments.

The next section will explore SIMD vectorization techniques that complement fixed-point optimizations on modern desktop and mobile CPUs.

10.2 SIMD Instructions (SSE, AVX) for Pixel Loops

10.2.1 Introduction to SIMD and Its Importance in CPU Graphics

Single Instruction, Multiple Data (SIMD) is a paradigm in modern CPU architectures that allows the simultaneous execution of the same instruction on multiple data points. This is particularly useful in graphics programming, where operations on pixels, vertices, or color channels are inherently data-parallel. Leveraging SIMD instructions such as SSE (Streaming SIMD Extensions) and AVX (Advanced Vector Extensions) significantly accelerates pixel-processing loops in CPU-only graphics rendering.
Since 2019, CPUs from Intel, AMD, and ARM have improved SIMD capabilities, making it essential for professional graphics programmers to understand and utilize these instructions for optimal performance.

10.2.2 Overview of SSE and AVX Instruction Sets

1. **SSE (Streaming SIMD Extensions)**

    - Introduced in the late 1990s, SSE extended the x86 architecture with 128-bit SIMD registers (XMM registers).
    - Each XMM register can hold:
        - Four 32-bit floats (single precision)
        - Two 64-bit doubles (double precision)
        - Sixteen 8-bit integers (bytes)
    - SSE supports arithmetic, logical, shuffle, and conversion operations on packed data.

- SSE variants include SSE2, SSE3, SSE4.x, which progressively added more instructions.

2. **AVX (Advanced Vector Extensions)**

- Introduced in 2011, AVX expands SIMD registers to 256 bits (YMM registers).
- Each YMM register can hold:
  - Eight 32-bit floats
  - Four 64-bit doubles
  - Thirty-two 8-bit integers (using AVX2)
- AVX adds support for wider vector processing and enhanced instruction sets for integer and floating-point arithmetic.
- AVX-512 extends registers further to 512 bits on some recent CPUs, but this book focuses on SSE and AVX, which are broadly supported in CPUs.

10.2.3 Why Use SIMD for Pixel Loops?

Pixel loops typically perform repetitive operations such as:

- Reading pixel color data
- Modifying color channels (e.g., brightness, contrast)
- Blending colors for transparency or anti-aliasing
- Applying filters or effects (e.g., grayscale, sepia)

These operations naturally map to SIMD as each pixel or channel operation is independent and can be processed in parallel, significantly reducing the instruction count and improving CPU cache utilization.

10.2.4 Programming Models for SIMD in C/C++

1. **Intrinsics**

   Intrinsics are compiler-supported functions that provide a low-level interface to SIMD instructions while maintaining C/C++ syntax. They offer fine-grained control and high performance without the complexity of assembly.

   Examples:

   - `_mm_load_ps` — Load four 32-bit floats into an SSE register.
   - `_mm_add_ps` — Add packed floats in SSE registers.
   - `_mm256_mul_ps` — Multiply packed floats in AVX registers.

   Intrinsics are preferred for writing SIMD code in modern C++ projects due to their portability and maintainability.

2. **Auto-Vectorization**

   Compilers like GCC, Clang, and MSVC can automatically vectorize loops. However, manual use of intrinsics often results in better control and performance, especially for complex pixel operations.

10.2.5 Practical Example: SIMD-Accelerated Alpha Blending

Alpha blending combines source and destination pixels based on the alpha (transparency) value. Consider the per-pixel operation:

```
out = src * alpha + dest * (1 - alpha)
```

Where `src`, `dest`, and `alpha` are color components normalized between 0 and 1.

- **Scalar (non-SIMD) Code Example**

```
void alpha_blend_scalar(uint8_t* dst, const uint8_t* src, uint8_t
 ↪  alpha, size_t count) {
    for (size_t i = 0; i < count; ++i) {
        dst[i] = (src[i] * alpha + dst[i] * (255 - alpha)) / 255;
    }
}
```

- **SIMD SSE Implementation**

```
#include <emmintrin.h> // SSE2 intrinsics

void alpha_blend_sse(uint8_t* dst, const uint8_t* src, uint8_t
 ↪  alpha, size_t count) {
    __m128i src_pixels, dst_pixels;
    __m128i alpha_vec = _mm_set1_epi16(alpha);
    __m128i inv_alpha_vec = _mm_set1_epi16(255 - alpha);

    size_t i = 0;
    for (; i + 15 < count; i += 16) {
        // Load 16 bytes (pixels) from src and dst
        src_pixels = _mm_loadu_si128((__m128i*)(src + i));
        dst_pixels = _mm_loadu_si128((__m128i*)(dst + i));

        // Unpack bytes to words to prevent overflow during
         ↪  multiplication
        __m128i src_lo = _mm_unpacklo_epi8(src_pixels,
         ↪  _mm_setzero_si128());
        __m128i src_hi = _mm_unpackhi_epi8(src_pixels,
         ↪  _mm_setzero_si128());
        __m128i dst_lo = _mm_unpacklo_epi8(dst_pixels,
         ↪  _mm_setzero_si128());
        __m128i dst_hi = _mm_unpackhi_epi8(dst_pixels,
        ↪  _mm_setzero_si128());

        // Multiply and blend low bytes
        __m128i blended_lo = _mm_add_epi16(
            _mm_mullo_epi16(src_lo, alpha_vec),
            _mm_mullo_epi16(dst_lo, inv_alpha_vec)
        );

        // Multiply and blend high bytes
        __m128i blended_hi = _mm_add_epi16(
            _mm_mullo_epi16(src_hi, alpha_vec),
            _mm_mullo_epi16(dst_hi, inv_alpha_vec)
        );

        // Divide by 255 approximation using (value + 128 + (value
        ↪  >> 8)) >> 8 for efficiency
        blended_lo =
        ↪  _mm_srli_epi16(_mm_add_epi16(_mm_add_epi16(blended_lo,
        ↪  _mm_set1_epi16(128)), _mm_srli_epi16(blended_lo, 8)),
        ↪  8);
        blended_hi =
        ↪  _mm_srli_epi16(_mm_add_epi16(_mm_add_epi16(blended_hi,
        ↪  _mm_set1_epi16(128)), _mm_srli_epi16(blended_hi, 8)),
        ↪  8);

        // Pack words back to bytes
        __m128i blended_pixels = _mm_packus_epi16(blended_lo,
        ↪  blended_hi);

        // Store result
        _mm_storeu_si128((__m128i*)(dst + i), blended_pixels);
```

```
    }

    // Process remaining pixels scalar way
    for (; i < count; ++i) {
        dst[i] = (src[i] * alpha + dst[i] * (255 - alpha)) / 255;
    }
}
```

This example processes 16 pixels per iteration using SSE2 instructions, dramatically improving performance over scalar code.

10.2.6 Memory Alignment and Performance Considerations

- **Alignment:** For maximum efficiency, SIMD loads and stores should be aligned to 16 bytes for SSE and 32 bytes for AVX. Misaligned accesses can cause penalties.
- Use `_mm_load_si128` and `_mm_store_si128` for aligned data and `_mm_loadu_si128`/`_mm_storeu_si128` for unaligned.
- Align pixel buffers using compiler directives or platform-specific APIs (e.g., `_aligned_malloc` on Windows, `posix_memalign` on Linux).
- **Loop unrolling and prefetching** can further improve SIMD loop throughput.
- Balance SIMD width with CPU cache line sizes to minimize cache misses.

10.2.7 Extending to AVX and AVX2

AVX increases vector width to 256 bits, doubling throughput:

- Use `_mm256_loadu_si256` and related intrinsics.

- Modify pixel loop to process 32 pixels at once (for 8-bit pixels).
- Consider AVX2 for integer SIMD operations relevant to pixel formats.

Example AVX2 extension of the above alpha blend function requires careful handling of instructions since not all SSE instructions have direct AVX2 equivalents, especially for packed integer multiplication.

10.2.8 Tools and Compiler Support

- **Compilers:** Recent GCC (10+), Clang (12+), and MSVC fully support SSE/AVX intrinsics and offer built-in functions for optimized code generation.
- **Profilers:** Intel VTune, AMD uProf, and Linux perf help analyze SIMD usage and CPU pipeline bottlenecks.
- **Third-Party Libraries:** Libraries such as Intel's Integrated Performance Primitives (IPP) offer highly optimized SIMD routines for image processing tasks but require linking and may increase binary size.

10.2.9 Summary

SIMD instructions provide substantial performance gains in CPU-only graphics programming by parallelizing pixel operations. SSE and AVX instruction sets, accessed through intrinsics, enable efficient implementations of alpha blending, color manipulation, filtering, and other pixel loop computations. Proper data alignment and knowledge of CPU-specific instruction sets are essential to fully leverage SIMD benefits. Future advancements and wider SIMD registers (AVX-512) will further increase parallelism, but SSE and AVX remain the most practical and widely supported SIMD instruction sets for CPU-based graphics rendering today.

10.3 Multithreading Image Operations

10.3.1 Introduction

Modern CPUs, both desktop and mobile, feature multiple cores capable of executing threads concurrently. Leveraging **multithreading** to parallelize image operations is one of the most effective optimization techniques in CPU-only graphics programming. By distributing computational workload across multiple CPU cores, programmers can significantly reduce frame processing times and enable real-time rendering speeds even without GPU acceleration.

This section discusses principles, design patterns, and practical implementation strategies for multithreading image operations, with a focus on contemporary tools and APIs available after 2019.

10.3.2 Fundamentals of Multithreading in Image Processing

Image operations—such as filtering, color adjustments, resizing, and compositing—often involve repetitive computations over large pixel arrays. These tasks are naturally parallelizable because pixel processing is mostly independent or localized.

Key advantages of multithreading image operations include:

- **Workload division:** Splitting an image into segments (tiles, scanlines, or blocks) and assigning each to a separate thread.
- **Improved CPU utilization:** Utilizes all cores effectively rather than idling on a single core.
- **Reduced latency:** Enables faster processing of large images or batch operations.

10.3.3 Threading Models and Work Partitioning

1. **Partitioning Strategies**

   - **Strip-based:** Dividing the image horizontally or vertically into strips assigned to different threads.
   - **Tile-based:** Dividing the image into rectangular tiles or blocks, improving cache locality.
   - **Dynamic task queues:** Threads fetch tasks dynamically from a shared queue, balancing load on irregular workloads.

2. **Granularity Considerations**

   Choosing the right task size affects performance:

   - Too large: underutilization of threads, slower load balancing.
   - Too small: increased overhead from thread synchronization and task switching.

   A common approach is to split images into tiles of sizes between 64x64 and 256x256 pixels.

10.3.4 Synchronization and Thread Safety

- **Data independence:** Image operations should avoid write conflicts by ensuring threads work on disjoint image regions.
- **Immutable inputs:** Source image data is typically read-only, enabling safe concurrent reads.
- **Minimizing synchronization:** Locks, mutexes, and atomic operations can reduce parallel efficiency. Designs minimizing shared state maximize scalability.

10.3.5 Implementing Multithreading with Modern APIs

1. **C++ Standard Library Threads (`std::thread`)**

   Since C++11, `std::thread` provides a portable, low-level threading API.

   Example: Simple horizontal strip division

```cpp
#include <thread>
#include <vector>

void process_strip(uint8_t* image, int width, int start_row, int
 ↪  end_row) {
    for (int y = start_row; y < end_row; ++y) {
        for (int x = 0; x < width; ++x) {
            // Process pixel at (x, y)
        }
    }
}

void multithread_image_processing(uint8_t* image, int width, int
 ↪  height) {
    const int num_threads = std::thread::hardware_concurrency();
    std::vector<std::thread> threads;

    int rows_per_thread = height / num_threads;

    for (int i = 0; i < num_threads; ++i) {
        int start = i * rows_per_thread;
        int end = (i == num_threads - 1) ? height : start +
         ↪  rows_per_thread;
        threads.emplace_back(process_strip, image, width, start,
         ↪  end);
    }
    for (auto& t : threads) t.join();
}
```

2. **Thread Pools and Task-Based Parallelism**

   Higher-level abstractions reduce thread management overhead and improve performance.

   - **C++17 Parallel STL Algorithms:** Some algorithms support parallel execution policies.
   - **Third-party libraries:**
     - **Intel Threading Building Blocks (TBB):** Offers task scheduling, work-stealing, and high-level parallel constructs.
     - **Microsoft Parallel Patterns Library (PPL):** Provides parallel loops, tasks, and concurrent containers for Windows.
     - **OpenMP:** Compiler directives for parallel loops, widely supported in GCC and Clang.

   Example using TBB:

```
#include <tbb/parallel_for.h>

void process_image_tbb(uint8_t* image, int width, int height) {
    tbb::parallel_for(0, height, [&](int y) {
        for (int x = 0; x < width; ++x) {
            // Process pixel at (x, y)
        }
    });
}
```

10.3.6 Considerations for CPU Cache and Memory Bandwidth

- **Data locality:** Tile-based partitioning enhances cache reuse, reducing cache misses.
- **False sharing:** Avoid multiple threads modifying data within the same cache line.
- **Memory bandwidth:** Multiple threads can saturate memory channels; optimizing data access patterns is crucial.

10.3.7 Load Balancing and Scalability

- **Dynamic scheduling:** Using work-stealing or dynamic queues prevents idle threads and balances uneven workloads.
- **Thread affinity:** Pinning threads to cores may improve performance on NUMA architectures.
- **Oversubscription:** Excessive thread counts beyond hardware concurrency can degrade performance due to context switching.

10.3.8 Debugging and Profiling Multithreaded Graphics Code

- Tools such as **Intel VTune**, **Visual Studio Profiler**, and **Linux perf** help identify threading bottlenecks.
- Race condition detection tools like **ThreadSanitizer** are valuable.
- Logging and deterministic execution models facilitate debugging.

10.3.9 Real-World Example: Multithreaded Gaussian Blur Filter

Applying a Gaussian blur involves convolution across image pixels and is compute-intensive.
Multithreaded approach:

- Divide the image vertically into strips.
- Each thread performs convolution independently on its strip.
- Overlap edges are handled by copying extra rows between strips.

10.3.10 Summary

Multithreading image operations on CPUs effectively harness modern multicore architectures for real-time software rendering. Choosing appropriate workload partitioning, minimizing synchronization, and leveraging modern threading APIs or libraries are key for scalable, maintainable implementations. When combined with fixed-point arithmetic and SIMD, multithreading forms a cornerstone of CPU-only graphics optimization.

10.4 Tiling Strategies for Caches

10.4.1 Introduction

Efficient use of CPU caches is critical for high-performance graphics programming on the CPU, especially in real-time 2D and 3D rendering pipelines where memory bandwidth and latency often become performance bottlenecks. One of the most effective optimization techniques to enhance cache utilization is **tiling**, a strategy that subdivides large images or data arrays into smaller, cache-friendly blocks (tiles) processed individually.

This section explores the principles of tiling strategies focused on CPU cache architectures, their impact on performance, implementation guidelines, and practical examples relevant to CPU-only graphics rendering. The discussion incorporates modern CPU designs, cache hierarchies, and programming tools prevalent since 2019.

10.4.2 CPU Cache Architecture Overview

Understanding CPU caches is foundational to applying tiling strategies effectively:

- **Cache levels (L1, L2, L3):**
  - L1 cache is the smallest (tens of KB) but fastest, typically split into instruction and data caches.
  - L2 cache is larger (hundreds of KB) with slightly higher latency.
  - L3 cache (shared among cores) is largest (several MB) but slower.
- **Cache line size:**
  Typical cache lines are 64 bytes wide on modern Intel and AMD CPUs. Each cache line holds contiguous memory bytes and is the minimum unit transferred between memory and cache.

- **Cache associativity and replacement policies:**
  These affect which data stays cached under repeated accesses.

Graphics programming must be designed to maximize **spatial locality** (accessing contiguous memory) and **temporal locality** (reusing data in cache before eviction).

10.4.3 Why Tiling Improves Cache Efficiency

Processing large images or framebuffers linearly often leads to frequent cache misses because:

- Images typically exceed cache sizes.
- Processing large scanlines causes cache lines to be evicted before reuse.
- Memory access patterns may lead to cache pollution or thrashing.

Tiling subdivides images into smaller regions (tiles), each fitting into L1 or L2 cache, so all required pixel data and associated computations occur while data resides in cache. This reduces main memory access, lowering latency and improving throughput.

10.4.4 Tile Size Selection and Cache Levels

Selecting appropriate tile sizes depends on the cache size and the size of the data structures:

- **Tile size criteria:**
  - The tile's data footprint (input pixels, output pixels, temporary buffers) should fit into the target cache level.

  - For L1 cache (typically 32KB per core for data), tiles of roughly 64x64 pixels (assuming 4 bytes per pixel) yield about 16KB per tile — ideal for L1 fitting including overhead.

- **Multiple buffers:**
  When algorithms require multiple buffers (e.g., source image, destination, intermediate), the tile size must be reduced proportionally.

- **Cache line alignment:**
  Tiles should align with cache lines to minimize false sharing and partial cache line usage.

10.4.5 Implementation Techniques

1. **Tile Loop Structure**

   Divide the image into tiles in nested loops over tile rows and columns. For each tile:

   - Compute starting pixel offsets.
   - Process pixels within the tile sequentially or with SIMD and multithreading.
   - Synchronize threads if necessary when tiles overlap or share boundary data.

   Example pseudocode:

```
int tileWidth = 64;
int tileHeight = 64;

for (int tileY = 0; tileY < imageHeight; tileY += tileHeight) {
    for (int tileX = 0; tileX < imageWidth; tileX += tileWidth) {
        int currentTileWidth = min(tileWidth, imageWidth - tileX);
        int currentTileHeight = min(tileHeight, imageHeight -
         ↪  tileY);

        process_tile(image, tileX, tileY, currentTileWidth,
         ↪  currentTileHeight);
    }
}
```

2. **Data Access Patterns Within Tiles**

   - Access pixels row-wise to benefit from spatial locality.
   - When possible, use continuous memory buffers to avoid scattered access.
   - Avoid unnecessary cache line invalidations by minimizing writes outside the tile.

10.4.6 Tiling in Multithreaded Contexts

- Assign tiles or groups of tiles to threads, improving load balancing and cache utilization.
- Tile-based partitioning often leads to natural thread independence with minimal synchronization.
- Cache contention between threads is reduced as each thread operates on distinct cache-line-aligned tiles.

10.4.7 Case Study: Tiled Software Rasterizer

A software rasterizer converting triangles into pixels benefits from tiling by:

- Dividing the framebuffer into tiles.
- Processing each tile independently for triangle coverage and shading.
- Keeping transformation matrices and intermediate buffers localized per tile.
- Reducing memory bandwidth by maximizing cache hits on depth and color buffers.

10.4.8 Tiling for Complex Effects and Filtering

Effects such as convolutional filters, motion blur, and post-processing shaders can leverage tiling by:

- Including border or padding pixels around tiles to handle edge conditions.
- Processing tiles in a pipeline fashion with intermediate buffers to avoid repeated memory loads.
- Ensuring thread-safe tile processing when implemented in parallel.

10.4.9 Integration with SIMD and Fixed-Point Optimizations

- Tiling complements SIMD vectorization by providing contiguous data chunks suitable for vector loads and stores.
- Fixed-point arithmetic benefits from tiled memory layouts, as smaller working sets fit into fast caches, reducing latency.

10.4.10 Tools and Profiling

- Use CPU performance counters (via Intel VTune, Linux perf) to monitor cache hit/miss ratios.

- Experiment with different tile sizes and layouts, guided by profiling data.
- Employ memory allocators that support cache line alignment and page boundaries for tiled buffers.

10.4.11 Summary

Tiling is a foundational optimization technique for CPU-based graphics programming that exploits CPU cache hierarchies by subdividing image processing into small, cache-resident blocks. Proper tile size selection, alignment, and processing order maximize data locality, reduce memory bandwidth usage, and improve overall throughput. When combined with multithreading and SIMD, tiling forms a cornerstone for achieving real-time graphics performance on modern CPUs without relying on GPUs.

10.5 Minimizing Memory Copies

10.5.1 Introduction

In CPU-based graphics programming, minimizing memory copies is essential for achieving high performance and low latency. Copying large blocks of pixel data, intermediate buffers, or framebuffers can incur significant CPU time and memory bandwidth usage. Reducing these copies allows applications to run faster, use less power, and respond more quickly to user input or animation updates.

This section explores strategies and best practices to minimize memory copying in graphics applications that rely solely on CPU processing, covering relevant concepts, typical pitfalls, and practical examples based on modern tools, operating systems, and CPUs available since 2019.

10.5.2 Why Minimizing Memory Copies Matters

- **CPU cycles:** Copying memory consumes CPU cycles that could be dedicated to computation, shading, or other graphics operations.
- **Memory bandwidth:** The cost of reading and writing large pixel arrays impacts overall memory bandwidth, often a bottleneck in graphics pipelines.
- **Cache pollution:** Unnecessary copies can evict useful data from CPU caches, degrading cache locality and causing additional cache misses.
- **Latency:** Extra copies introduce delays, particularly harmful for real-time applications like games or interactive GUIs.

10.5.3 Common Sources of Memory Copies in CPU Graphics

- **Buffer transfers:** Copying from source images to working buffers, or between framebuffers.
- **Double buffering:** Swapping between front and back buffers can involve copying or pointer swaps.
- **Format conversions:** Converting pixel formats (e.g., from RGB to RGBA) often involves data copies.
- **Temporary buffers:** Intermediate steps such as filters, scaling, or alpha blending often create new buffers.
- **Data alignment adjustments:** Padding or alignment corrections can trigger memory copies.

10.5.4 Strategies to Minimize Copies

1. **Use Pointer Swapping Instead of Copying**

   Where possible, avoid copying buffer contents by swapping pointers to buffers:

   - In double buffering, instead of copying the back buffer to the front, swap the pointers referencing each buffer.
   - This approach reduces the operation to a few CPU instructions and pointer updates.

   Example:

```
uint32_t* frontBuffer = bufferA;
uint32_t* backBuffer = bufferB;

// Instead of memcpy(frontBuffer, backBuffer, size);
std::swap(frontBuffer, backBuffer);
```

This technique is only feasible if the application architecture allows safe buffer ownership swapping without side effects.

2. **Perform In-Place Operations**

   Modify pixel data directly in its original buffer whenever possible to avoid intermediate copies.

   - Example: Alpha blending directly on the destination buffer rather than creating a separate blended buffer.
   - In-place transformations reduce peak memory usage and copying overhead.

   Caution: In-place modifications require careful management to avoid data hazards when source and destination overlap.

3. **Utilize Memory Mapping and Shared Buffers**

   Operating systems like Windows and Linux support memory-mapped files or shared memory buffers:

   - Mapping image data directly into addressable memory reduces explicit copying from disk or external sources.
   - Shared buffers between threads or processes can avoid copies by working on the same memory region with proper synchronization.

4. **Optimize Data Structures and Layouts**

   Arrange data to facilitate direct access and reduce the need for rearrangement copies:

   - Use contiguous buffers for images with known pixel formats.
   - Avoid complex or fragmented data structures that force copying when flattening or reshaping data.

5. **Reduce Temporary Buffer Usage with Streaming Algorithms**

   Streaming algorithms process data in a single pass, reducing the need for temporary storage:

   - For example, image filters that apply on-the-fly pixel processing without creating a full temporary buffer.
   - This technique reduces copying and memory footprint simultaneously.

6. **Use Efficient APIs and System Calls**

   Modern OS APIs and graphics libraries sometimes provide zero-copy or minimal-copy operations:

   - Windows `BitBlt` with `DIBSections` can optimize blitting with minimal copying.
   - On Linux, `mmap` combined with direct framebuffer access can reduce memory operations.

10.5.5 Practical Example: Manual Image Blitter Optimization

When manually implementing an image blitter on the CPU:

- Avoid copying the source image buffer into a temporary buffer before blitting.
- Read pixels directly from the source buffer.
- Write pixels directly into the destination framebuffer or back buffer.
- If format conversion is necessary, perform it per pixel on the fly rather than pre-copying and converting the entire buffer.

This approach reduces memory copies significantly and improves frame rendering speed.

10.5.6 Multithreaded Context and Copy Minimization

In multithreaded rendering pipelines:

- Avoid copying buffers between threads; instead, partition work so each thread works on a distinct tile or region.
- Use thread-safe pointer swaps or atomic flags to signal buffer availability.
- When synchronization is necessary, prefer shared data structures with controlled access rather than copying data for thread-local usage.

10.5.7 Profiling and Measuring Copy Overheads

Use profiling tools to detect costly memory copies:

- On Windows, tools like Visual Studio Profiler and Intel VTune provide insights into memory usage and copy-related overhead.
- On Linux, `perf` and Valgrind can help identify inefficient memory operations.

Identifying and minimizing copies often leads to dramatic performance gains, especially on CPU-bound graphics tasks.

10.5.8 Summary

Minimizing memory copies in CPU-only graphics programming is a critical optimization that improves speed, reduces memory bandwidth usage, and lowers latency. Effective techniques include pointer swapping, in-place operations, shared buffers, and streaming algorithms. Coupled with careful data layout and profiling, these approaches enable building responsive and efficient graphics applications without relying on GPU acceleration.

Chapter 11

Cross-Platform Display on Windows and Linux

11.1 GDI and BitBlt on Windows

11.1.1 Introduction

When programming graphics on the CPU without GPU acceleration, the Windows Graphics Device Interface (GDI) remains one of the fundamental APIs for rendering and displaying images. GDI provides a set of functions for drawing graphics primitives, text, and bitmaps directly to device contexts such as windows, printers, or memory buffers. A central operation in many CPU-based graphics applications on Windows is the **BitBlt** function, which performs high-speed bit-block transfers between device contexts.

This section delves into the use of GDI and BitBlt for efficient image presentation, covering their mechanisms, capabilities, constraints, and best practices on modern Windows systems, incorporating developments and relevant techniques.

11.1.2 Overview of GDI

- **GDI fundamentals:**
  GDI is a Windows API for representing graphical objects and transmitting them to output devices. It supports drawing shapes, lines, text, and managing pixel buffers known as device contexts (DCs).

- **Device Contexts (DCs):**
  DCs encapsulate drawing surfaces, which may correspond to windows, off-screen bitmaps, printers, or metafiles. GDI operations require acquiring and releasing DCs to draw.

- **Compatibility and Modern Use:**
  Despite the rise of Direct2D and GPU-based APIs, GDI remains widely supported, lightweight, and suitable for CPU-only graphics rendering on Windows, particularly for legacy systems and simple applications.

11.1.3 The BitBlt Function

- **Purpose:**
  BitBlt (Bit Block Transfer) copies a block of pixels from a source device context to a destination device context. It can perform raster operations, such as copy, invert, and merge, during the transfer.

- **Function signature:**

```
BOOL BitBlt(
  HDC   hdcDest,
  int   nXDest,
  int   nYDest,
  int   nWidth,
  int   nHeight,
  HDC   hdcSrc,
  int   nXSrc,
  int   nYSrc,
  DWORD dwRop
);
```

  - `hdcDest`: Handle to the destination device context.
  - `nXDest`, `nYDest`: Coordinates on the destination DC where the image will be placed.
  - `nWidth`, `nHeight`: Dimensions of the block to copy.
  - `hdcSrc`: Source device context handle.
  - `nXSrc`, `nYSrc`: Coordinates of the upper-left corner in the source DC.
  - `dwRop`: Raster operation code defining how pixels are combined.

- **Common raster operation:**
  `SRCCOPY` — copies the source directly to the destination.

11.1.4 Creating Compatible Device Contexts and Bitmaps

To render graphics in memory before displaying on screen (a technique often used to reduce flicker and enable double buffering), applications use **compatible DCs** and **bitmaps**:

- Use `CreateCompatibleDC()` to create an off-screen memory DC compatible with the display.
- Use `CreateCompatibleBitmap()` or `CreateDIBSection()` to create a bitmap matching the DC's pixel format.

- Select the bitmap into the memory DC with `SelectObject()`.

Example workflow:

```
HDC hdcScreen = GetDC(hwnd); // Get window DC
HDC hdcMem = CreateCompatibleDC(hdcScreen); // Create memory DC
HBITMAP hbmMem = CreateCompatibleBitmap(hdcScreen, width, height); //
 ↪  Create bitmap
HBITMAP hbmOld = (HBITMAP)SelectObject(hdcMem, hbmMem);

// Perform CPU-based drawing to hdcMem here

BitBlt(hdcScreen, 0, 0, width, height, hdcMem, 0, 0, SRCCOPY);

SelectObject(hdcMem, hbmOld);
DeleteObject(hbmMem);
DeleteDC(hdcMem);
ReleaseDC(hwnd, hdcScreen);
```

11.1.5 Performance Considerations

- **Minimizing flicker:**
  Drawing to an off-screen memory DC and then transferring the final image to the window using BitBlt is the standard double buffering technique on Windows with GDI.

- **Choosing between CreateCompatibleBitmap and CreateDIBSection:**

  – `CreateCompatibleBitmap` returns a GDI bitmap compatible with the display device but offers less control over pixel format and slower pixel access for CPU drawing.

  - `CreateDIBSection` allows direct memory access to bitmap pixels, improving CPU drawing performance and flexibility, which is critical for software rendering.

- **Pixel format and alignment:**
  Use 32-bit bitmaps (RGBA or BGRA) for ease of manipulation with modern CPUs and to leverage aligned memory accesses.

- **Raster operation impact:**
  Using `SRCCOPY` is optimal for copying pixel data. Other raster operations may incur additional processing and reduce throughput.

- **Use of modern Windows versions:**
  Windows 10 and 11 have optimized GDI under the hood, including better handling of hardware acceleration for some operations. However, for CPU-only rendering, the above techniques remain valid.

11.1.6 Handling High-DPI and Scaling

Modern Windows systems support high-DPI displays, which affect coordinate systems and bitmap scaling:

- Query DPI scaling using `GetDpiForWindow()` or `GetDpiForMonitor()`.
- Adjust bitmap dimensions and coordinate calculations to match scaled client area sizes.
- Ensure compatible bitmaps and device contexts consider DPI scaling to avoid blurry rendering.

11.1.7 Integration with Message Loop and WM_PAINT

- GDI drawing with BitBlt is typically invoked during window paint events (`WM_PAINT`).
- Use `BeginPaint()` and `EndPaint()` to obtain the window DC and validate the repaint region.
- Redraw from the off-screen buffer using BitBlt inside the paint handler to ensure smooth visual updates.

11.1.8 Limitations and Best Practices

- GDI is single-threaded with respect to DC usage; avoid concurrent GDI calls on the same DC without synchronization.
- Memory DCs and bitmaps should be created once and reused where possible to avoid costly allocations.
- For complex software rendering pipelines, consider combining GDI with other APIs such as Direct2D for hybrid CPU-GPU approaches, but for CPU-only scenarios, GDI and BitBlt remain reliable.
- Avoid large BitBlt operations every frame if only small portions change; consider region-based partial updates.

11.1.9 Summary

GDI and BitBlt provide a robust, well-supported mechanism for CPU-based graphics display on Windows. By leveraging compatible device contexts, off-screen bitmaps, and efficient BitBlt transfers, developers can implement smooth double-buffered rendering pipelines suitable for software rasterizers, text rendering, and 2D graphics without GPU assistance. Understanding DPI scaling, raster operations, and resource management ensures modern applications remain performant and visually consistent across Windows versions.

11.2 Xlib or Direct Framebuffer Access on Linux

11.2.1 Introduction

On Linux systems, displaying graphics using CPU-only methods can be accomplished primarily via two approaches: using the **X Window System's Xlib API** or by directly accessing the **framebuffer device** (`/dev/fb0`). Each approach presents distinct advantages, challenges, and use cases.

This section covers detailed technical insights into both methods, with emphasis on modern Linux environments, discussing how to implement CPU-driven rendering with minimal dependencies, handling device contexts or pixel buffers, synchronization, and performance considerations.

11.2.2 Overview of Xlib for CPU Graphics Rendering

- **X Window System:**
  The most prevalent graphical windowing system on Linux and UNIX-like systems. It abstracts hardware interaction and provides windows, event handling, and drawing primitives.

- **Xlib:**
  A low-level C library interface to the X protocol, allowing direct communication with the X server. It offers functions to create windows, draw primitives, manipulate images, and manage events.

- **Why Xlib?**

  – Provides portability across various Linux desktop environments.

  – Integrates with window managers, enabling windowed applications.

  - Supports hardware-accelerated and software rendering depending on X server configuration.
  - Suitable for CPU-based graphics by drawing to XImage buffers or using shared memory extensions.

11.2.3 Creating and Managing Windows with Xlib

- **Opening a connection:**
  Use `XOpenDisplay(NULL)` to connect to the X server on the local machine.

- **Creating a window:**
  With `XCreateSimpleWindow()`, specify position, size, border, and background.

- **Event handling:**
  Select input events such as `Exposure`, `KeyPress`, `ButtonPress` using `XSelectInput()`.

- **Displaying the window:**
  Map it with `XMapWindow()` and flush commands with `XFlush()` or `XSync()`.

11.2.4 Drawing Pixels and Images in Xlib

- **Using `XPutPixel()` with `XImage`:**
  Direct pixel manipulation is possible by creating an `XImage` structure. This allows CPU-based drawing by modifying the image's data buffer in memory.

- **Creating an `XImage`:**

```
XImage *ximage = XCreateImage(display, visual, depth, ZPixmap, 0,
                              (char *)malloc(width * height *
                               ↪  bytes_per_pixel),
                              width, height, 32, 0);
```

- **Pixel format:**
  The `XImage` pixel format depends on the display's visual and depth. The common format is 24-bit or 32-bit packed RGB or BGR.

- **Drawing workflow:**

  1. Modify pixels in `ximage->data` directly.
  2. Use `XPutImage()` to transfer the `XImage` to the window's drawable surface.

- **Shared Memory Extension (XShm):**
  For improved performance, use the MIT-SHM extension which enables shared memory segments between the client and server, reducing data copying overhead.

11.2.5 Limitations of Xlib for Software Rendering

- **Latency and overhead:**
  X protocol involves communication with the X server, introducing latency compared to direct framebuffer access.

- **Complexity of pixel formats:**
  The need to manage different visual formats and byte orders can complicate pixel manipulation.

- **Windowing constraints:**
  Xlib is primarily designed for windowed applications, so full-screen or direct hardware access is less straightforward.

11.2.6 Direct Framebuffer Access on Linux

- **What is the framebuffer?**
  The Linux framebuffer (`/dev/fb0`) is a character device that provides a memory-mapped view of the video hardware framebuffer, enabling direct pixel manipulation bypassing the X server.

- **Use cases:**
  - Embedded systems or minimal Linux setups without an X server.
  - Full-screen graphics applications or boot splash screens.
  - CPU-only graphics rendering requiring maximal control and minimal overhead.

11.2.7 Opening and Mapping the Framebuffer

- **Accessing the device:**
  Open the framebuffer device with `open("/dev/fb0", O_RDWR)`.

- **Retrieving framebuffer info:**
  Use the `ioctl()` system call with `FBIOGET_VSCREENINFO` and `FBIOGET_FSCREENINFO` to obtain resolution, bits per pixel, line length, and memory layout.

- **Memory mapping:**

Map framebuffer memory into the process address space via `mmap()`, enabling direct pixel reads/writes.

```
void *fbp = mmap(0, screensize, PROT_READ | PROT_WRITE, MAP_SHARED,
↪  fbfd, 0);
```

- **Pixel format:**
  Varies by hardware, but typically 16, 24, or 32 bits per pixel, with RGB or BGR ordering.

11.2.8 Writing Pixels and Drawing on the Framebuffer

- **Calculating pixel offset:**
  The framebuffer is usually a linear array of bytes. Each pixel's byte offset is computed as:

```
offset = (y * line_length) + (x * bytes_per_pixel)
```

- **Writing pixel data:**
  After offset calculation, write pixel color values according to the framebuffer's pixel format and endianess.

- **Buffering techniques:**

  – Because framebuffer writes appear immediately, flickering may occur.

  – Implement software double buffering by maintaining an off-screen buffer in system memory, drawing to it, then copying it entirely or partially to the framebuffer to reduce tearing and flicker.

11.2.9 Synchronization and Refresh

- **No vertical sync (vsync):**
  Framebuffer access typically lacks vsync, causing possible tearing artifacts during updates.

- **Manual synchronization:**
  Applications can reduce visible tearing by timing updates to coincide with blanking intervals or by minimizing the update region and duration.

- **Polling vs event-driven:**
  Unlike Xlib's event model, framebuffer access is purely polling-based; the application controls when to draw and update.

11.2.10 Security and Permissions

- Access to `/dev/fb0` usually requires root privileges or specific group membership, limiting application deployment flexibility.

- Using Xlib avoids such permissions but incurs protocol overhead.

11.2.11 Performance Considerations

- **Xlib with XShm:**
  Offers good performance for CPU rendering on typical desktop Linux environments.

- **Framebuffer:**
  Enables minimal overhead and near real-time updates, ideal for embedded or specialized applications.

- **Hardware acceleration:**
  Neither Xlib nor framebuffer direct access leverages GPU acceleration by default; all pixel manipulation is CPU-bound.

11.2.12 Modern Tooling and Libraries

- Developers often build on these low-level techniques with libraries like **DirectFB**, **Wayland protocols**, or **KMS/DRM** for modern systems.
- However, for CPU-only rendering scenarios requiring minimal dependencies, Xlib and direct framebuffer access remain relevant and performant choices.
- Integration with modern Linux desktop environments and compositors (e.g., GNOME, KDE) should consider compatibility and fallback to software rendering if GPU acceleration is unavailable.

11.2.13 Summary

Linux provides versatile options for CPU-only graphics rendering:

- **Xlib** offers a standardized windowed environment with direct pixel manipulation through `XImage` and the possibility of shared memory optimizations.
- **Direct framebuffer access** provides raw, high-performance pixel-level control suitable for full-screen or embedded systems without the overhead of windowing systems.

Understanding both approaches equips developers to implement efficient software rasterizers, animations, and GUI elements tailored to the specific Linux deployment context, whether desktop or embedded.

11.3 Using SDL2 Purely in Software Mode

11.3.1 Introduction

Simple DirectMedia Layer 2 (SDL2) is a widely-used, cross-platform library that provides access to graphics, input devices, audio, and more. While it is often leveraged with hardware acceleration (GPU) for rendering, SDL2 fully supports **software rendering mode**, enabling developers to perform all graphics operations using only the CPU.

This section details how to configure and use SDL2 exclusively in software mode for CPU-based graphics programming on Windows and Linux, discussing initialization, pixel manipulation, performance considerations, and practical examples using SDL2 (version 2.0.10 and above) relevant to environments.

11.3.2 Why Use SDL2 in Software Mode?

- **Hardware independence:**
  Software mode is critical when targeting systems lacking suitable GPU hardware or drivers, such as virtual machines, embedded devices, or minimalist OS setups.

- **Predictable rendering:**
  Software rendering provides deterministic pixel operations without GPU pipeline variability, beneficial for precise CPU-only graphics implementations.

- **Debugging and development:**
  Simplifies debugging by isolating rendering logic from GPU drivers and hardware quirks.

11.3.3 SDL2 Initialization for Software Rendering

To initialize SDL2 for software rendering, the standard process includes creating a window and a software renderer explicitly.

```
#include <SDL2/SDL.h>

if (SDL_Init(SDL_INIT_VIDEO) != 0) {
    // Handle error
}

SDL_Window *window = SDL_CreateWindow("Software Rendering",
                                      SDL_WINDOWPOS_CENTERED,
                                      SDL_WINDOWPOS_CENTERED,
                                      width, height,
                                      0);
if (!window) {
    // Handle error
}

// Create a software renderer by specifying SDL_RENDERER_SOFTWARE flag
SDL_Renderer *renderer = SDL_CreateRenderer(window, -1,
↪   SDL_RENDERER_SOFTWARE);
if (!renderer) {
    // Handle error
}
```

- The key is passing `SDL_RENDERER_SOFTWARE` to `SDL_CreateRenderer()`, which forces the renderer to operate entirely in software mode.

11.3.4 Managing Pixel Buffers and Textures in Software Mode

- **Software textures:**
  Even in software mode, SDL2 uses textures as render targets. Software rendering utilizes system memory for texture storage.
- **Pixel format:**
  SDL2 commonly uses the `SDL_PIXELFORMAT_ARGB8888` or `SDL_PIXELFORMAT_RGBA8888` formats, depending on platform and requirements.
- **Creating streaming textures:**
  To draw arbitrary pixels, create a streaming texture that allows CPU access to the pixel buffer.

```
SDL_Texture *texture = SDL_CreateTexture(renderer,
                                         SDL_PIXELFORMAT_ARGB8888,
                                         SDL_TEXTUREACCESS_STREAMING,
                                         width, height);
if (!texture) {
    // Handle error
}
```

- **Locking texture to access pixels:**

```
void *pixels;
int pitch;
SDL_LockTexture(texture, NULL, &pixels, &pitch);
```

```
// Modify pixels here

SDL_UnlockTexture(texture);
```

The `pitch` is the number of bytes per row, necessary for correctly calculating pixel addresses.

11.3.5 Drawing Primitives Using CPU in SDL2 Software Mode

- **Direct pixel manipulation:**
  After locking the texture, you can write pixels directly into the pixel buffer using standard C/C++ pointer arithmetic, respecting the pitch and pixel format.

- **Implementing basic primitives:**

  - **Lines:** Use Bresenham's line algorithm to compute pixels between points.
  - **Rectangles:** Fill pixel blocks using loops.
  - **Circles:** Use midpoint or Bresenham's circle algorithms.

- **Alpha blending:**
  Software mode supports manual alpha blending by calculating output pixel values before writing to the buffer.

11.3.6 Presenting the Frame

Once drawing on the texture is complete, the texture must be copied to the renderer's target and presented on screen.

```
SDL_RenderClear(renderer);
SDL_RenderCopy(renderer, texture, NULL, NULL);
SDL_RenderPresent(renderer);
```

- `SDL_RenderClear()` clears the current rendering target.
- `SDL_RenderCopy()` copies the texture to the current rendering target.
- `SDL_RenderPresent()` updates the window with the rendered content.

All these operations happen fully in software without GPU involvement when `SDL_RENDERER_SOFTWARE` is used.

11.3.7 Handling Input and Events

SDL2 supports input devices independently of rendering mode. Poll or wait for events using:

```
SDL_Event event;
while (SDL_PollEvent(&event)) {
    // Process events (keyboard, mouse, window close)
}
```

This is essential for interactive applications relying on software rendering.

11.3.8 Performance Considerations in Software Mode

- **CPU-bound workload:**
  Rendering is limited by the CPU's ability to write pixel data and the memory bandwidth.

- **Optimizations:**
  - Minimize pixel buffer modifications each frame.
  - Use dirty rectangles or partial updates where possible.
  - Employ efficient algorithms for primitives and avoid expensive per-pixel operations.
- **Multithreading:**
  SDL2 itself is mostly single-threaded, but CPU rendering logic can be offloaded or parallelized with care, synchronizing before unlocking and presenting textures.

11.3.9 Example: Simple Software Mode Animation Loop

```
// Initialize SDL2 with software renderer as above

bool running = true;
SDL_Event event;

while (running) {
    while (SDL_PollEvent(&event)) {
        if (event.type == SDL_QUIT) {
            running = false;
        }
    }

    // Lock texture to update pixels
    void *pixels;
    int pitch;
    SDL_LockTexture(texture, NULL, &pixels, &pitch);
    // Clear buffer (example: fill with black)
    memset(pixels, 0, pitch * height);

    // Draw your primitives here using CPU operations on 'pixels'

    SDL_UnlockTexture(texture);

    // Render and present
    SDL_RenderClear(renderer);
    SDL_RenderCopy(renderer, texture, NULL, NULL);
    SDL_RenderPresent(renderer);

    // Delay to limit frame rate (e.g., ~60fps)
    SDL_Delay(16);
}

// Cleanup SDL2 resources
SDL_DestroyTexture(texture);
SDL_DestroyRenderer(renderer);
SDL_DestroyWindow(window);
SDL_Quit();
```

11.3.10 Limitations and Compatibility

- **No GPU acceleration:**
  All rendering is performed on the CPU, which may be significantly slower than GPU-accelerated modes, especially for large resolutions or complex scenes.

- **Cross-platform behavior:**
  SDL2 abstracts many platform differences, but rendering performance and pixel formats may vary.

- **Modern system considerations:**
  On many modern Linux and Windows systems, software rendering is a fallback mode; system compositors and drivers may introduce overhead or interference.

11.3.11 Summary

Using SDL2 purely in software mode offers a straightforward and cross-platform approach to CPU-only graphics programming on Windows and Linux. It provides:

- A unified API for window creation, input handling, and pixel buffer management.
- Facilities to manipulate pixels directly through streaming textures.
- Simple mechanisms to present software-rendered frames on the screen.
- Flexibility for applications targeting environments without GPU support or requiring deterministic CPU-only rendering.

This method is ideal for educational purposes, embedded systems, or software rasterization projects where GPU access is unavailable or undesirable.

11.4 Software Surfaces vs Hardware-Accelerated Ones

11.4.1 Introduction

In modern graphics programming on both Windows and Linux, a fundamental decision is whether to use **software surfaces** or **hardware-accelerated surfaces** for rendering and display output. This choice has significant implications for performance, portability, resource usage, and application complexity, especially in the context of CPU-only graphics programming, which this book emphasizes.

This section provides a detailed, professional examination of software and hardware-accelerated surfaces, their architectures, benefits, limitations, and how they relate to CPU-based rendering pipelines.

11.4.2 Defining Software Surfaces

- **Concept:**
  A software surface is a region of memory—typically system RAM—that holds pixel data manipulated and rasterized entirely by the CPU. The CPU reads and writes pixel values, performing all rendering calculations without GPU or specialized hardware acceleration.

- **Characteristics:**

  - Resides in general-purpose memory.
  - Accessible via pointers, allowing direct pixel manipulation.
  - Rendering operations (drawing lines, fills, textures) are computed by CPU code.

  - Transfers to the display occur by copying or blitting the memory region to the screen buffer or window surface.

- **Typical usage scenarios:**
  - Environments without GPU support or minimal graphical hardware.
  - Legacy systems or embedded devices.
  - Software rasterizers and CPU-bound rendering engines.
  - Debugging or fallback rendering modes.

11.4.3 Defining Hardware-Accelerated Surfaces

- **Concept:**
  Hardware-accelerated surfaces leverage specialized graphics hardware (GPUs) and drivers to perform rendering operations such as rasterization, texture mapping, shading, and compositing. Pixel data may reside in **video memory (VRAM)** or specialized GPU buffers.

- **Characteristics:**
  - Surfaces are managed by GPU drivers and graphics APIs (Direct3D, OpenGL, Vulkan, Metal).
  - Rendering commands are offloaded to GPU, allowing massive parallelism and specialized hardware pipelines.
  - Often includes optimizations like texture compression, hardware blending, and GPU memory paging.
  - Data transfer between CPU memory and GPU memory can introduce latency and bandwidth costs.

- **Typical usage scenarios:**
  - Real-time 3D rendering and high-performance 2D graphics.
  - Applications requiring high frame rates or complex visual effects.
  - Modern desktop, mobile, and console environments with GPUs.

11.4.4 Key Differences Between Software and Hardware Surfaces

Comparison: Software vs Hardware-Accelerated Surfaces

Aspect	**Software Surfaces**	**Hardware-Accelerated Surfaces**
Location of pixel data	System RAM	GPU VRAM or dedicated buffers
Rendering computation	CPU executes all drawing and rasterization	GPU hardware performs rendering
Access speed	Fast CPU access, but limited by memory bandwidth	Fast GPU memory access, optimized for parallel rendering
Flexibility	Complete control of pixels, no driver constraints	Limited to API and driver capabilities
Performance	Limited by CPU speed and memory bandwidth	Orders of magnitude faster for complex tasks
Latency	Minimal for direct writes, but slower for complex scenes	Potential latency due to CPU-GPU synchronization

Aspect	Software Surfaces	Hardware-Accelerated Surfaces
Portability	High; works on any system with CPU and memory	Dependent on GPU drivers and support
Power consumption	Generally higher CPU load	GPU optimized for low power on many platforms

11.4.5 Software Surfaces in Detail

- **Memory Layout and Pixel Formats:**
  Software surfaces usually employ standard pixel formats such as 32-bit ARGB, 24-bit RGB, or 16-bit RGB565. The pixel data is laid out in linear buffers or tightly packed rows with stride (pitch).

- **CPU Rendering Pipelines:**
  All rendering algorithms — line drawing, triangle rasterization, texture sampling, shading — are implemented as CPU code. This allows precise control but can be computationally expensive.

- **Data Transfer to Display:**
  Since the display hardware or window system ultimately requires the pixel data, software surfaces are copied or blitted to the final framebuffer or window surface. On Windows, this may be via GDI's `BitBlt`; on Linux, through Xlib's `XPutImage` or framebuffer writes.

- **Advantages:**

  – Maximum portability and compatibility.

  - Simplified debugging due to direct memory access.
  - Independence from GPU driver bugs or variability.

- **Disadvantages:**
  - Limited scalability to high resolutions or complex scenes.
  - High CPU utilization for rendering tasks.
  - Potentially increased power consumption on mobile/embedded devices.

11.4.6 Hardware-Accelerated Surfaces in Detail

- **Graphics APIs and Drivers:**
  Surfaces are created and managed via APIs like Direct3D on Windows or Vulkan/OpenGL on Linux. These APIs provide abstractions for GPU memory allocation, resource binding, and command submission.
- **GPU Memory Management:**
  GPU surfaces reside in VRAM, optimized for fast random access and parallel writes. Memory layout may include tiled or swizzled formats to improve cache coherency on GPU hardware.
- **Rendering Workflow:**
  Rendering commands are issued by the CPU but executed asynchronously by the GPU. Shaders and fixed-function hardware accelerate per-pixel and per-vertex computations.
- **Advantages:**
  - Massive parallelism enables complex scenes and effects.
  - Reduced CPU load, freeing the processor for other tasks.

  - Access to advanced features like anti-aliasing, hardware blending, and compute shaders.

- **Disadvantages:**
  - Dependence on GPU drivers, which may vary in quality and compatibility.
  - Synchronization overhead between CPU and GPU.
  - Complexity in managing GPU resources and API learning curve.

11.4.7 Implications for CPU-Only Graphics Programming

Since this book focuses on **graphics programming using the CPU only (no GPU)**, understanding these differences is crucial for design decisions:

- **Why prefer software surfaces?**
  - Full rendering pipeline control.
  - No dependency on GPU hardware or drivers.
  - Consistent behavior across platforms and hardware generations.
- **Performance Challenges:**
  - Software surfaces demand careful optimization to approach real-time frame rates.
  - Techniques such as fixed-point math, tiling, cache-aware algorithms (covered in Chapter 10) are vital to efficient software surface rendering.
- **Integration with System Display:**
  - Software surfaces require explicit transfers to the display system.

  – This step is a performance bottleneck and must be optimized (e.g., using shared memory extensions on Linux or efficient BitBlt calls on Windows).

11.4.8 Practical Examples and Tools

- **SDL2 in Software Rendering Mode:**
  Provides software surfaces via streaming textures. Developers manipulate pixel buffers in system memory and push frames via software renderers.

- **Direct GDI Usage on Windows:**
  GDI surfaces reside in system memory and use `BitBlt` to transfer to device contexts, ideal for pure software rendering pipelines.

- **Xlib and Framebuffer Access on Linux:**
  Software surfaces are represented by `XImage` structures or direct framebuffer memory, manipulated by the CPU and flushed to display buffers.

- **Third-party libraries:**
  Libraries like **Cairo** or **Pixman** provide software rendering backends that generate surfaces in system memory, with optional integration for GPU acceleration.

11.4.9 Summary

The choice between software and hardware-accelerated surfaces profoundly affects how applications render graphics:

- **Software surfaces** offer unmatched portability, control, and simplicity for CPU-only rendering, but impose significant CPU and memory demands.

- **Hardware-accelerated surfaces** excel at high-performance, complex graphics workloads but rely on GPU availability, drivers, and APIs beyond the CPU.

For applications targeting CPU-only rendering on Windows and Linux, mastery of software surfaces and their efficient integration with system display mechanisms is essential. This knowledge underpins the strategies, algorithms, and optimizations detailed throughout this book.

11.5 Framebuffer Access in Virtual Terminals (Linux /dev/fb0)

11.5.1 Introduction

On Linux systems, direct access to the framebuffer device provides a low-level and efficient mechanism for rendering graphics purely via the CPU without relying on a windowing system or GPU acceleration. The framebuffer device, typically exposed as `/dev/fb0` (or other devices for multiple framebuffers), represents the raw pixel buffer that the Linux kernel uses to display output on the screen.

This section explores how to utilize framebuffer access in virtual terminals for CPU-only graphics programming, detailing device characteristics, memory mapping, pixel formats, synchronization, and practical usage on modern Linux systems as of 2019 and later.

11.5.2 The Linux Framebuffer Device

- **What is `/dev/fb0`?**
  `/dev/fb0` is a special device file representing the primary framebuffer on a Linux system. It provides a memory-mapped interface through which user-space programs can directly read from and write to the pixel data that the kernel will scan out to the display.

- **Virtual Terminals and Framebuffer:**
  The framebuffer device typically controls the virtual terminal (VT) console display outside of graphical environments like X11 or Wayland. Programs accessing `/dev/fb0` draw directly to the visible screen in the current VT.

- **Multiple Framebuffers:**

Systems may expose multiple framebuffer devices (`/dev/fb1`, `/dev/fb2`, etc.), each corresponding to different hardware or virtual outputs.

11.5.3 Accessing the Framebuffer Device

- **Opening and Memory Mapping**

  To efficiently manipulate the framebuffer, programs open `/dev/fb0` and map the framebuffer memory into the process address space.

```
#include <fcntl.h>
#include <sys/mman.h>
#include <linux/fb.h>
#include <unistd.h>
#include <sys/ioctl.h>

int fb_fd = open("/dev/fb0", O_RDWR);
if (fb_fd < 0) {
    // Handle error: permission denied or device missing
}

// Query framebuffer information
struct fb_var_screeninfo vinfo;
struct fb_fix_screeninfo finfo;

ioctl(fb_fd, FBIOGET_VSCREENINFO, &vinfo);
ioctl(fb_fd, FBIOGET_FSCREENINFO, &finfo);

// Map framebuffer to user memory
size_t screensize = finfo.smem_len;
uint8_t *fbp = (uint8_t *)mmap(NULL, screensize, PROT_READ |
↪  PROT_WRITE, MAP_SHARED, fb_fd, 0);
if (fbp == MAP_FAILED) {
    // Handle error: mmap failed
}
```

  - **`fb_var_screeninfo`** holds variable parameters such as resolution, bits per pixel, and color offsets.
  - **`fb_fix_screeninfo`** contains fixed parameters like framebuffer memory length and line length (stride).

- **Important Considerations**
  - Programs need appropriate permissions (often root or belonging to the `video` group) to access `/dev/fb0`.
  - Framebuffer memory is shared with the kernel and potentially other processes; care is required to avoid conflicts.
  - The framebuffer device works best when no graphical display server (X11, Wayland) is running on the console.

11.5.4 Framebuffer Characteristics and Pixel Formats

- **Resolution:**
  Obtained from `vinfo.xres` and `vinfo.yres`.
- **Bits Per Pixel (bpp):**
  Common
  values include 8, 16, 24, or 32 bits per pixel. The `vinfo.bits_per_pixel` field indicates this.

- **Color Layout:**
  The `fb_var_screeninfo` structure defines bit offsets and lengths for red, green, blue, and transparency channels, enabling programs to correctly encode pixel values.

- **Line Length (Stride):**
  The number of bytes per line of pixels is given by `finfo.line_length`. It may be greater than `xres * bytes_per_pixel` due to alignment or hardware constraints.

11.5.5 Drawing Pixels and Graphics on the Framebuffer

To draw pixels, programs calculate pixel offsets based on (x, y) coordinates and write appropriate pixel values to the mapped framebuffer memory.
Example to set a pixel at `(x, y)`:

```
int bytes_per_pixel = vinfo.bits_per_pixel / 8;
long location = y * finfo.line_length + x * bytes_per_pixel;

uint32_t pixel_value = /* encode color according to vinfo.red.offset,
↪  green.offset, blue.offset */;
memcpy(fbp + location, &pixel_value, bytes_per_pixel);
```

- Encoding color requires shifting and masking color components based on the variable screen info.

- For example, a 32-bit pixel with 8 bits each for red, green, and blue, and an alpha or padding channel.

11.5.6 Synchronization and Buffer Management

- **Double Buffering:**
  Framebuffer devices often support virtual screens for double buffering (via `vinfo.yres_virtual`), enabling smoother animation by drawing off-screen and swapping buffers.

- **Page Flipping:**
  Not always supported; most framebuffer devices require manual management of visible areas or rely on explicit screen updates.

- **Vertical Sync:**
  The standard framebuffer interface does not provide vertical sync events. Applications need to implement timing or use kernel extensions to avoid tearing.

11.5.7 Use Cases and Limitations

- **Use Cases:**
  - Embedded systems without graphical stacks.
  - Kiosk or minimal user interface applications.
  - Custom boot splash screens and diagnostics.
  - Educational and experimental CPU-based rasterizers.

- **Limitations:**
  - Requires exclusive access or careful coordination with other processes.
  - No native support for windowing or compositing.
  - Performance and feature set vary widely by hardware and kernel support.

  – Modern Linux desktops use X11 or Wayland instead, limiting `/dev/fb0` availability.

11.5.8 Practical Example: Simple Framebuffer Program

```c
// Open and map framebuffer as shown earlier

// Clear screen with black
memset(fbp, 0, screensize);

// Draw a white rectangle
for (int y = 100; y < 200; y++) {
    for (int x = 100; x < 300; x++) {
        long loc = y * finfo.line_length + x * bytes_per_pixel;
        uint32_t white = (255 << vinfo.red.offset)  (255 <<
         ↪  vinfo.green.offset)  (255 << vinfo.blue.offset);
        memcpy(fbp + loc, &white, bytes_per_pixel);
    }
}

// Keep displaying until user input or timeout
sleep(5);

// Cleanup
munmap(fbp, screensize);
close(fb_fd);
```

This example demonstrates direct pixel manipulation purely on the CPU with no windowing system.

11.5.9 Security and Permissions

- Accessing `/dev/fb0` typically requires elevated permissions. Common approaches to safely allow programs access include:
  - Adding the user to the `video` group.
  - Setting udev rules to grant device access.
  - Running the program with root privileges (less recommended).
- Newer Linux systems and distributions may use kernel modesetting (KMS) and DRM (Direct Rendering Manager), which provide more advanced and secure ways of accessing framebuffers.

11.5.10 Summary

Direct framebuffer access via `/dev/fb0` on Linux virtual terminals remains a powerful and flexible method for CPU-only graphics programming. It offers:

- Minimal dependencies on graphical subsystems.
- Full control over the pixel data and rendering pipeline.
- Real-time rendering capability constrained only by CPU and memory bandwidth.

Despite the rise of compositing window systems and hardware acceleration, framebuffer programming is still relevant in embedded environments, low-level system utilities, and specialized applications requiring deterministic CPU-based rendering without GPU involvement.

Chapter 12

Advanced Case Studies

12.1 Writing a CPU-Only Image Viewer

12.1.1 Introduction

Creating an image viewer entirely on the CPU provides an ideal case study for applying many of the principles explored throughout this book. It combines CPU-based rasterization, pixel format conversion, platform-specific display routines, and performance-aware optimizations into a functional and portable application. This section details how to design and implement a simple, efficient image viewer that does not rely on GPU acceleration, third-party graphical toolkits, or high-level frameworks like OpenGL or Vulkan.

The implementation is targeted at modern Windows and Linux systems using system-native APIs such as GDI and Xlib, with all rendering and image manipulation done in user-space via the CPU. The viewer supports common raster image formats such as BMP, PNG, and JPEG, and includes zooming, panning, and pixel format conversion, all without GPU reliance.

12.1.2 Overview and Design Goals

The image viewer will adhere to the following design principles:

- **CPU-only rendering:** All image decoding, pixel format conversion, and drawing are performed by the CPU.
- **Cross-platform support:** Compatible with Windows and Linux using system-level APIs (GDI, Xlib).
- **Efficient memory usage:** Optimized use of memory and caching strategies to support large images.
- **Simple interface:** Keyboard or basic mouse input to control image zoom and pan.
- **Core Components**
    1. **Image loading and decoding (CPU-based)**
    2. **Pixel buffer abstraction**
    3. **Display routines per platform**
    4. **Event loop and input handling**
    5. **Zoom and pan mechanics**

12.1.3 Image Decoding (CPU-Based)

- **Supported Formats**

  To simplify cross-platform deployment and ensure CPU-only operation, image decoding is done using lightweight, CPU-only libraries:

- **BMP:** Parsed manually or using minimal inline decoders.
- **PNG:** Decoded using libraries such as `lodepng` (header-only, C++) or `libpng` compiled in software mode.
- **JPEG:** Decoded using `stb_image.h` or `libjpeg-turbo` in non-GPU mode.

These libraries are chosen because they:

- Operate entirely in CPU memory.
- Support a wide range of formats with minimal dependencies.
- Can be easily statically compiled or embedded.

• **Decoding Pipeline**

- Read image file into memory.
- Use a decoding library to extract raw pixel buffer (usually in 24-bit RGB or 32-bit RGBA).
- Convert pixel format if needed to match the platform display format (e.g., BGRA on Windows GDI).

Example using `stb_image.h`:

```
int width, height, channels;
unsigned char* data = stbi_load("image.jpg", &width, &height,
↪  &channels, 4); // force RGBA

// data now holds width * height * 4 bytes
```

12.1.4 Internal Pixel Buffer and Rendering

- **Surface Representation**

  A software surface structure stores the image in memory:

```
struct SoftwareSurface {
    int width;
    int height;
    int pitch;
    uint32_t* pixels; // 32-bit ARGB or BGRA depending on target
};
```

  The surface acts as a back buffer. All zooming and panning effects are applied by copying appropriate regions from this buffer to the display surface.

- **Zooming and Panning**

  – **Zooming:** Achieved by scaling pixel blocks (nearest-neighbor or bilinear).

  – **Panning:** Offset the copy window from the internal surface when rendering.

```
for (int y = 0; y < screen_height; ++y) {
    for (int x = 0; x < screen_width; ++x) {
        int src_x = (x + pan_x) / zoom_factor;
        int src_y = (y + pan_y) / zoom_factor;
        uint32_t color = get_pixel(source_surface, src_x, src_y);
        set_pixel(display_surface, x, y, color);
    }
}
```

  This logic keeps all drawing within system RAM using CPU instructions only.

12.1.5 Displaying the Image

- **Windows Implementation: GDI and BitBlt**
  - Create a memory device context (`CreateCompatibleDC`) and a compatible bitmap.
  - Copy the pixel buffer into the bitmap's memory using `SetDIBits` or direct memory access.
  - Use `BitBlt` or `StretchDIBits` to transfer the image to the window.

```
BITMAPINFO bmi = { /* fill with appropriate format info */ };
StretchDIBits(hdc, 0, 0, win_w, win_h, 0, 0, img_w, img_h, pixels,
↪  &bmi, DIB_RGB_COLORS, SRCCOPY);
```

- **Linux Implementation: Xlib and XPutImage**
  - Create an `XImage` structure with the correct pixel format.
  - Fill the `data` buffer with pixel values from the internal surface.
  - Use `XPutImage` to transfer the image to the display window.

```
XImage* img = XCreateImage(display, visual, depth, ZPixmap, 0,
↪  (char*)pixels, width, height, 32, 0);
XPutImage(display, window, gc, img, 0, 0, 0, 0, width, height);
```

On both platforms, synchronization (e.g., `XFlush`, `UpdateWindow`) ensures proper display.

12.1.6 User Input and Event Loop

- **Keyboard Controls**
  - Arrow keys or WASD to pan image.
  - $*$ / - keys to zoom in and out.
  - ESC or Q to quit.
- **Windows Input Loop**

  Use the standard Windows message loop (`GetMessage`, `TranslateMessage`, `DispatchMessage`) to intercept keyboard input and redraw the image upon events such as `WM_PAINT` or `WM_KEYDOWN`.
- **Linux Input Loop (Xlib)**

  Use `XNextEvent` and `XPending` to handle keyboard events and expose events (`Expose`) to trigger redraw.

12.1.7 Performance Considerations

- **Memory Management:** Ensure surfaces are allocated with aligned memory for optimal cache access.
- **Scaling Algorithm:** Nearest-neighbor is fast; bilinear offers better visual quality with slightly more CPU cost.
- **Pixel Format Conversion:** Pre-convert the image to match the display format to avoid per-frame conversion overhead.
- **Dirty Rectangles:** Update only the changed parts of the screen when possible.

12.1.8 Testing and Compatibility

- **Post-2019 OS Compatibility**
  - **Windows 10/11:** GDI remains supported and stable for CPU-drawn graphics.
  - **Linux Kernel 5.x+:** Framebuffer access (`/dev/fb0`) and Xlib are still functional, though many distros now prefer Wayland (Wayland support via shared memory extensions or raw DRM/KMS possible but not covered in this book).
- **Third-Party Tools**
  - **Build Systems:** CMake or Meson for cross-platform compilation.
  - **Image Libraries:** `stb_image`, `libjpeg-turbo`, or `lodepng` with CPU-only compilation.
  - **Debugging:** Valgrind and AddressSanitizer for memory safety; Perf or gprof for profiling bottlenecks.

12.1.9 Summary

A CPU-only image viewer demonstrates how far software rendering can go in a modern environment, even without GPU acceleration. By carefully managing memory, display APIs, and rendering logic, we can create a fully functional, responsive, and cross-platform viewer suitable for embedded systems, low-level testing environments, or educational tools.
This case study consolidates concepts such as:

- File decoding and raster formats.

- Software surfaces and pixel manipulation.
- System-specific display handling.
- Real-time input processing.
- Performance tuning without GPU support.

As such, it exemplifies the power and viability of CPU-centric graphics programming in modern computing environments.

12.2 Creating a 2D Retro Game Without Any GPU

12.2.1 Introduction

Designing a complete 2D retro-style game using only CPU rendering is a classic exercise in low-level graphics programming and system efficiency. It emphasizes an era where processors alone handled all visuals, reminiscent of the 1980s and early 1990s gaming platforms such as the NES, SNES, or Amiga. In this section, we walk through the development of such a game using modern systems (Windows and Linux) while explicitly avoiding any GPU or hardware acceleration. All visuals, effects, and UI elements are rendered in software using system memory and CPU instructions.

12.2.2 Design Principles

This project aims to reflect a game engine that:

- Performs **all rendering using CPU instructions**.
- Uses **fixed-point math** for fast arithmetic.
- Leverages **double-buffered surfaces** to prevent flicker.
- Draws sprites, backgrounds, text, and UI elements manually.
- Provides a consistent **retro game look and feel**.
- Is compatible with **Windows and Linux**, using GDI and Xlib respectively.
- Uses libraries only for **non-rendering tasks**, such as image or font decoding.

12.2.3 Game Concept and Engine Architecture

- **Game Type**

  We choose a simple top-down action game, such as a tank battle or dungeon shooter. The screen consists of:

  – A **tile-based background** (static or scrolling).

  – A **player sprite**.

  – A few **enemy sprites**.

  – **Bullets**, explosions, and other effects.

- **Core Modules**

  1. **Renderer**: Handles drawing primitives and sprites to a software framebuffer.
  2. **Tilemap Engine**: Manages scrolling background layers.
  3. **Sprite System**: Updates and renders animated objects.
  4. **Input Manager**: Reads keyboard or mouse input.
  5. **Game Loop**: Updates logic, processes input, draws frame, and handles timing.
  6. **Platform Display**: Copies framebuffer to screen (GDI/Xlib).

12.2.4 Graphics Architecture

- **Framebuffer Management**

  – Allocate a back buffer in system memory (`uint32_t*` representing ARGB or BGRA pixels).

  - Clear buffer each frame (or support dirty rectangles for performance).
  - Draw tilemaps and sprites by copying from preloaded bitmap surfaces.
  - Finally, **blit** the framebuffer to the window using `BitBlt` (Windows) or `XPutImage` (Linux).

- **Example: Surface Structure**

```
typedef struct {
    int width, height;
    uint32_t* pixels;
} Surface;
```

- **Drawing API (CPU)**

```
void draw_pixel(Surface* surface, int x, int y, uint32_t color);
void draw_rect(Surface* surface, int x, int y, int w, int h,
 ↪  uint32_t color);
void blit_sprite(Surface* dst, Surface* sprite, int x, int y, bool
 ↪  transparent);
```

Alpha blending may be omitted or implemented only for semi-transparent effects. Most 8-bit era games used **binary transparency masks**, where a single color (e.g., magenta) is treated as transparent.

12.2.5 Tile-Based Background Rendering

A retro game engine often uses **tilemaps** to render backgrounds efficiently.

- **Tilemap Design**

  - A tilemap is a 2D array of indices into a tile set.
  - Each tile is a small surface (e.g., 8×8 or 16×16 pixels).
  - Scrolling is implemented by adjusting an offset when drawing the tile grid.

```
void render_tilemap(Surface* target, int tilemap[MAP_H][MAP_W],
↪  Surface* tileset[], int scroll_x, int scroll_y);
```

  Rendering is done by blitting the visible tile region based on the scroll offset.

- **Optimization Techniques**
  - Only draw visible tiles.
  - Pre-convert tileset surfaces to match framebuffer format.
  - Avoid redrawing unchanged areas when scroll offset remains constant.

12.2.6 Sprite System and Animation

Sprites are animated objects such as players, enemies, bullets, or UI elements. Each sprite has:

- A position (`x`, `y`).
- A current animation frame.
- A pointer to a surface.
- Velocity (`dx`, `dy`).
- Flags for state and collision.

```
typedef struct {
    int x, y;
    int dx, dy;
    int frame;
    Surface* frames[MAX_FRAMES];
    bool visible;
} Sprite;
```

- **Sprite Drawing**
  - Update animation frame based on a timer or tick counter.
  - Blit the sprite to the framebuffer considering transparency and clipping.
- **Animation Strategy**
  - Store frames as separate surfaces or a single sprite sheet.
  - Implement animation sequences using frame indices and time intervals.

12.2.7 Game Logic and Input

- **Game Loop Structure**

  ```
  while (running) {
      process_input();
      update_game_state();
      render_frame();
      wait_for_next_frame();
  }
  ```

The loop
runs at a target framerate (e.g., 30 or 60 FPS) using `Sleep()` on Windows or `usleep()` on Linux, or more accurate timers (`QueryPerformanceCounter`, `clock_gettime`).

- **Input Management**
  - On Windows, use `GetAsyncKeyState` or handle `WM_KEYDOWN` messages.
  - On Linux, use `XNextEvent` or `XQueryKeymap`.

12.2.8 Sound and Music (Optional)

While this book focuses on CPU-only graphics, sound can be handled by external libraries such as SDL2 (used only for audio), or platform-specific APIs (Windows Multimedia API, ALSA on Linux). Sound can remain minimal or retro (beeps, chiptunes), or omitted for full CPU-rendered purity.

12.2.9 Platform Display

- **Windows (GDI)**
  - Create a `HWND` window with `WS_OVERLAPPEDWINDOW` style.
  - Allocate a compatible bitmap and copy the framebuffer using `SetDIBits` or `StretchDIBits`.
  - Use `InvalidateRect` and `UpdateWindow` to trigger redraw.
- **Linux (Xlib)**
  - Create a simple X11 window.

  - Allocate an `XImage` with matching format to framebuffer.
  - Use `XPutImage` on each frame.
  - Handle input and window close events via `XNextEvent`.

12.2.10 Example Game: CPU Tank Battle

Features:

- 16×16 tile background map.
- Player-controlled tank with 4-directional movement.
- Simple enemy AI with line-of-sight shooting.
- Projectile collision and hit detection.
- Level scrolling in all directions.
- Sprite-based explosions and powerups.

Technical Stats:

- Resolution: 320×240 or 640×480.
- Tile size: 16×16.
- Palette: 8-bit or 32-bit true color.
- Framerate: Locked at 30 FPS using `clock_gettime`.

12.2.11 Optimizations

- Use **fixed-point math** to avoid floating-point overhead.
- Cache rendered tiles if scrolling is slow or repetitive.
- Pack sprite data tightly in memory to reduce cache misses.
- Avoid per-pixel alpha blending unless necessary.
- Consider `memcpy` or loop unrolling when copying pixel blocks.

12.2.12 Summary

Creating a 2D retro game using only CPU rendering is a powerful testament to how efficient, performant, and expressive software rendering can still be in a modern context. With full control over the graphics pipeline, timing, and resource management, this style of programming helps developers appreciate the roots of game development while applying modern tools and system-level understanding.

The retro game's structure—tilemaps, sprites, framebuffer rendering—offers a rich foundation for building not only games but also embedded system GUIs, demos, or education tools, all without requiring a GPU or external game engine.

12.3 Porting a Game Engine to Use Only Software Rendering

12.3.1 Introduction

Modern game engines are built with GPU acceleration as a fundamental assumption. Porting such an engine to use only CPU rendering is a technically demanding process, yet a valuable one when targeting low-end hardware, embedded systems, headless servers, or historically-inspired platforms. This section explores how to port an existing game engine to operate entirely through software rendering. The emphasis is placed on removing GPU dependencies, re-implementing rendering pipelines in CPU logic, and maintaining visual and functional parity while optimizing performance for real-time interaction.
This approach is particularly relevant for:

- Retro game developers and emulation software.
- Security-focused systems avoiding external drivers.
- CPU-only platforms (e.g., virtual machines, custom OSs, sandboxed environments).
- Academic or research use where full transparency and control over the rendering pipeline are required.

12.3.2 Understanding the Original Engine Architecture

Before porting, a clear understanding of the original engine's structure is necessary. Most GPU-centric engines follow a modular design:

- **Rendering Abstraction Layer**: Often wraps OpenGL, DirectX, Vulkan, or Metal.
- **Asset Loader**: Loads models, textures, shaders.
- **Scene Graph or ECS (Entity-Component System)**: Represents world state.
- **Renderer Pipeline**: Applies shaders, buffers, z-culling, blending, and compositing.
- **Frame Buffer Output**: Final image is handed off to the GPU or window manager.

The goal in porting is to *replace the entire rendering pipeline and output chain with CPU-based alternatives* while leaving gameplay logic, physics, and input systems intact.

12.3.3 Strategy for Porting

- **Step 1: Remove or Disable GPU Dependencies**
  - Identify rendering backend (OpenGL, Vulkan, etc.).
  - Disable any external shader compilation, GPU buffer management, or GPU-side texture management.
  - Remove reliance on external drivers or graphics libraries that target GPU hardware.
- **Step 2: Define Software Surfaces**

  Introduce a software surface abstraction to act as the CPU-side framebuffer. Each rendering target is a region in memory:

```
struct Surface {
    int width, height;
    int pitch;
    uint32_t* pixels; // 32-bit ARGB or BGRA
};
```

All drawing and raster operations target these surfaces. They serve as drop-in replacements for GPU framebuffers.

- **Step 3: Replace GPU Features with Software Equivalents**

Table 3-1: GPU Features and Software Rendering Equivalents

GPU Feature	Software Rendering Equivalent
Texturing	Manual pixel sampling from texture arrays
Shaders	Fixed-function CPU rasterization pipelines
Depth buffer	Software Z-buffer implemented in RAM
Alpha blending	Per-pixel blending using CPU instructions
Vertex buffers	CPU-managed vertex arrays and loops
Clipping/culling	Manual frustum clipping and bounding box tests
Antialiasing	Supersampling or post-processing blur

12.3.4 Rebuilding the Rendering Pipeline in Software

- **2D Rendering Pipeline**

If the engine is primarily 2D (e.g., tile-based or sprite-based), the transition is relatively straightforward.

- Replace texture binds with direct `memcpy` or masked `blit` calls.
- Recreate basic affine transformations (scale, rotate) in CPU.
- Implement dirty rectangle tracking to optimize redraw regions.

- **3D Rendering Pipeline**

  For 3D engines, the software renderer must implement:

  - **Vertex transformation** using CPU-side matrix math.
  - **Backface culling** based on vertex winding and normals.
  - **Triangle rasterization** with scanline or barycentric methods.
  - **Texture sampling** with optional mipmaps and filtering.
  - **Z-buffering** to resolve depth per pixel.

  Each frame, transformed triangles are rasterized into the framebuffer line-by-line. Textures are sampled directly via pointer math, and depth tested using a CPU-managed Z-buffer array of the same resolution as the framebuffer.

  Example rasterization loop:

```
for (int y = minY; y <= maxY; ++y) {
    for (int x = minX; x <= maxX; ++x) {
        float bary[3];
        if (point_in_triangle(x, y, v0, v1, v2, bary)) {
            float z = interpolate_z(bary, v0.z, v1.z, v2.z);
            if (z < zbuffer[y * width + x]) {
                zbuffer[y * width + x] = z;
                uint32_t color = sample_texture(bary, tex0, tex1,
                ↪  tex2);
                framebuffer[y * width + x] = color;
            }
        }
    }
}
```

12.3.5 Platform Output and Integration

- **Windows: GDI**

  Use a memory-backed `HBITMAP` to copy the final framebuffer to a window.

  ```
  StretchDIBits(hdc, 0, 0, width, height, 0, 0, width, height,
  ↪  framebuffer, &bmi, DIB_RGB_COLORS, SRCCOPY);
  ```

- **Linux: Xlib or Framebuffer**

  For Xlib, use `XImage`:

  ```
  Image* img = XCreateImage(display, visual, 24, ZPixmap, 0,
  ↪  (char*)framebuffer, width, height, 32, 0);
  XPutImage(display, window, gc, img, 0, 0, 0, 0, width, height);
  ```

  For framebuffer-based systems:

  - Map `/dev/fb0` to memory.
  - Copy the framebuffer directly with proper pixel format conversion.

12.3.6 Real-World Considerations

- **Memory Usage**

  Software rendering is memory-intensive. Consider:

  - Using memory-aligned surfaces (`posix_memalign`, `_aligned_malloc`).
  - Minimizing intermediate buffers.
  - Reusing surfaces across frames where possible.

- **Performance**

  To sustain 30–60 FPS:

  - Use integer or fixed-point math instead of floating-point where possible.
  - Favor lookup tables (e.g., sine/cosine) for trigonometry.
  - Minimize per-pixel branching and conditionals.
  - Use multi-threading to divide rendering into tiles or scanlines.

- **Portability**

  - Keep OS-specific display logic in separate files.
  - Use `#ifdef _WIN32` or `#ifdef __linux__` guards.
  - Stick to C99/C++17 standards for better cross-platform compilation.

- **Debugging**

  - Build in debug overlays (FPS counter, bounding boxes).
  - Implement pixel inspection modes to verify rendering correctness.
  - Use tools like Valgrind, AddressSanitizer, and CPU profilers to trace performance bottlenecks.

12.3.7 Example: Porting a Simple OpenGL Game Engine

Suppose the original engine uses OpenGL for:

- Drawing textured quads.
- Simple lighting via shaders.
- 3D camera perspective.

Porting steps:

1. Strip out all OpenGL function calls.
2. Replace `glDrawArrays` with manual triangle rasterization.
3. Emulate shaders with custom per-pixel logic in C/C++.
4. Load textures using `stb_image.h`, convert to software surfaces.
5. Write custom software camera transform routines (matrix * vector).
6. Build frame output routines using Xlib (Linux) or GDI (Windows).

The end result retains gameplay and visual logic but is now hardware-independent.

12.3.8 Limitations and Trade-offs

- **Performance Ceiling**: Software rendering is slower, especially at high resolutions.
- **Missing Features**: Effects like hardware shaders, antialiasing, and post-processing may be limited or omitted.

- **Development Time**: Rebuilding a rendering pipeline from scratch is labor-intensive.
- **Use Case Specificity**: Best suited for education, simulation, embedded use, or retro-inspired design.

12.3.9 Summary

Porting a GPU-based game engine to use only software rendering is a transformative process that shifts full control of rendering back to the CPU. While performance trade-offs exist, the benefits in control, determinism, portability, and hardware independence are significant. Such an engine becomes viable for constrained environments or educational purposes, and showcases the depth and power of CPU-based graphical computation in the post-GPU era.

This approach also strengthens a developer's understanding of the graphics pipeline, memory layout, and rendering performance—all critical to professional-level graphics programming using only the CPU.

12.4 Writing an Emulator's Display System in Pure Software

12.4.1 Introduction

Creating an emulator requires accurately reproducing not only a system's CPU and memory behavior, but also its graphical output. Emulating the display subsystem is especially challenging when restricted to CPU-only rendering. This section focuses on implementing an emulator's video output using software rendering exclusively, targeting systems without GPU support or when deterministic control is required (as in debugging, retro preservation, or embedded systems).
A software-based video system must simulate legacy video memory, draw pixels to a software framebuffer, and display the results on modern Windows and Linux systems—all without hardware acceleration.

12.4.2 Use Cases and Target Systems

Emulators targeted in this context may include:

- 8-bit and 16-bit consoles (NES, SNES, Sega Master System, Genesis).
- Vintage computers (Commodore 64, Amiga, MSX).
- Arcade boards (e.g., Capcom CPS-1).
- Early personal computers (IBM PC, Apple II).

These systems used fixed video resolutions, tile-based graphics, hardware sprites, and simple palettes—making them well-suited to accurate software emulation on modern CPUs.

12.4.3 Architecture of a Software Video System in an Emulator

A CPU-only video system typically includes:

1. **Video RAM (VRAM) Emulation**: Stores tile maps, sprite data, and color palettes.
2. **Software Framebuffer**: A pixel buffer in main memory.
3. **Rendering Loop**: Reads emulated video registers and VRAM, and draws frames line by line.
4. **Host Display Output**: Transfers the framebuffer to the host OS's window using GDI (Windows) or Xlib (Linux).

Each frame of the original system is rendered exactly as it would have appeared on native hardware, but computed fully by the host CPU.

12.4.4 Memory Layout and Pixel Format

The emulator must maintain its own **internal memory structures** for the emulated VRAM and palette. For example:

```
uint8_t vram[0x4000];       // Emulated system video memory
uint32_t palette[256];      // RGBA or BGRA entries
uint8_t framebuffer[256 * 240]; // Indexes into palette (e.g., NES
↪  resolution)
```

This framebuffer is converted to 32-bit pixels and written to the software rendering surface:

```
uint32_t render_buffer[256 * 240]; // Final output to host display
```

Conversion step:

```
for (int i = 0; i < width * height; ++i) {
    render_buffer[i] = palette[framebuffer[i]];
}
```

12.4.5 Implementing the Video Rendering Logic

The video output is determined by interpreting hardware-specific registers and memory contents. The CPU must simulate:

- **Scanline timing**
- **Background and sprite priority**
- **Scrolling offsets**
- **Sprite flipping and masking**
- **Palette lookup**
- **Tile-Based Background Drawing**

  Many older systems use tilemaps to draw the background. Each tile is a small bitmap (usually 8×8 pixels) stored in VRAM, and a tilemap specifies which tiles to draw in each screen location.

  Example rendering pseudocode:

```
for (int row = 0; row < 30; ++row) {
    for (int col = 0; col < 32; ++col) {
        int tile_index = tilemap[row * 32 + col];
        draw_tile(render_buffer, tile_index, col * 8, row * 8);
    }
}
```

`draw_tile()` reads the tile pattern from `vram` and renders pixels to the software framebuffer.

- **Sprite Rendering**

  Sprites are rendered separately and may overlap with the background. The emulator must simulate:

  – Per-scanline sprite limits (e.g., NES allows only 8 per line).

  – Sprite transparency and priority over background.

  – Sprite flipping (horizontal/vertical).

  – Palette selection for each sprite.

  Rendering is done by overlaying sprite pixels onto the framebuffer, using transparency rules.

12.4.6 Vertical Synchronization and Frame Timing

To match the emulated system's output frequency (usually 50Hz or 60Hz), the emulator must:

- Accumulate CPU cycles.
- Trigger a screen refresh when enough cycles for one frame are reached.

- Synchronize output with real-time if required.

This means the framebuffer is updated 60 times per second. Each frame must be completed within ~16.6ms to maintain real-time performance.
A timing mechanism (e.g., `std::chrono` in C++ or `QueryPerformanceCounter` on Windows) is used to throttle execution to match original video timing.

12.4.7 Host Display Integration

- **Windows (GDI)**

  Use a `BITMAPINFO` and `StretchDIBits` to copy the software-rendered buffer to the window client area.

  ```
  StretchDIBits(hdc, 0, 0, width, height, 0, 0, width, height,
   ↪  render_buffer, &bitmapInfo, DIB_RGB_COLORS, SRCCOPY);
  ```

  – The surface must be in 32-bit color format.

  – Call this in the main message loop or via `WM_PAINT`.

- **Linux (Xlib)**

  Use `XCreateImage` and `XPutImage` to write the buffer to an X11 window.

  ```
  XImage* ximage = XCreateImage(display, visual, 24, ZPixmap, 0,
   ↪  (char*)render_buffer, width, height, 32, 0);
  XPutImage(display, window, gc, ximage, 0, 0, 0, 0, width, height);
  ```

  – Use `XFlush` to ensure timely screen update.

  – Render within `Expose` event or regular refresh loop.

12.4.8 Performance and Optimization

- **Resolution**

  Most emulated systems use small resolutions (e.g., 256×240, 320×200), allowing real-time rendering even on modest CPUs.

- **Optimizations**

  - Avoid full redraws if only a few scanlines changed.
  - Use `memcpy` or SIMD instructions for pixel copying.
  - If palettes change infrequently, cache color lookups.
  - Use **dirty rectangle** tracking for efficiency.

- **Multithreading**

  Offload rendering and host display to a dedicated thread to decouple emulation timing from drawing time.

12.4.9 Advanced Topics

- **Shaders in Software**

  To simulate CRT effects (scanlines, curvature), write pixel post-processing filters in C/C++. For example, add horizontal black lines or brightness falloff.

- **Debugging Tools**

  Include a mode that shows:

  - VRAM content.
  - Tilemap layout.

  - Sprite list with attributes.
  - Timing charts (scanline count, VBlank period).

  This is invaluable for emulator development and regression testing.

- **Fullscreen Mode**

  For fullscreen support:

  - On Windows, use `ChangeDisplaySettings`.
  - On Linux, use `XResizeWindow` and remove window borders.

  Still, rendering is performed in software and stretched to screen size.

12.4.10 Real Example: NES Emulator Display

A software-based NES display renderer must:

- Parse the PPU memory (pattern tables, name tables).
- Read PPU registers for scroll, palette, control flags.
- Generate the entire 256×240 image line by line.
- Handle 8-sprite-per-line rule, background priority.
- Output to window using `GDI` or `Xlib`.

Despite its simplicity, the NES PPU emulation illustrates all fundamental display challenges in software rendering.

12.4.11 Summary

Writing an emulator's display system in pure software is a challenging yet rewarding endeavor. It enables complete control, high portability, and faithful reproduction of historical systems. CPU-based rendering, though slower than GPU-accelerated alternatives, is more transparent and suitable for in-depth debugging and preservation work.

By maintaining tight integration between emulated VRAM, timing, and output rendering—using only CPU logic and memory manipulation—you gain full insight into how vintage systems worked and render them authentically on today's hardware.

Chapter 13

Using Assembly for Performance

13.1 When to Use Inline or External Assembly

13.1.1 Introduction

Assembly language remains an essential tool in CPU-only graphics programming, especially when maximum control over performance, memory access, or instruction-level parallelism is required. With modern compilers such as GCC, Clang, and MSVC offering extensive optimizations, most high-level code is efficiently compiled. However, there are cases where writing assembly—either inline within C/C++ code or as external assembly modules—provides unmatched precision and speed.

This section addresses **when and why** to use **inline assembly** versus **external assembly files**, how modern development tools support each approach, and how to make the right architectural decision in performance-critical CPU graphics applications.

13.1.2 What Assembly Gives You in CPU Graphics

Using assembly language in CPU-bound graphics applications allows:

- Manual control over instruction selection and ordering.
- Exploitation of SIMD instructions (SSE, AVX, NEON).
- Exact scheduling for memory and cache usage.
- Tight control over register allocation.
- Elimination of compiler-inserted overhead or safety checks.
- Bypassing language abstractions for raw pixel and memory manipulation.

These benefits are especially relevant in:

- Pixel-by-pixel rendering algorithms (line drawing, rasterization).
- Software alpha blending and masking.
- Software-based image scaling, rotation, and filtering.
- Real-time sprite and tile manipulation.

13.1.3 Inline Assembly: Overview and Best Use Cases

Inline assembly refers to embedding assembly instructions directly within a C or C++ source file. Modern compilers support this using specific syntax:

- **GCC/Clang**: Uses `__asm__` or `asm` keyword.
- **MSVC**: Uses `__asm` (only in 32-bit builds; 64-bit MSVC does not support inline assembly).

- **Advantages of Inline Assembly**
  - Close proximity to C/C++ code: Better for mixed operations.
  - Easier to maintain and debug when closely tied to specific logic.
  - Can directly reference C/C++ variables, allowing seamless data exchange.
  - Good for short, performance-critical functions (like pixel blending or tight loops).
- **When to Use Inline Assembly**

  Use inline assembly when:
  - You need **tight integration** with C variables or structures.
  - Writing **short routines** (typically <20 lines).
  - Targeting **performance hotspots** already identified with profiling.
  - Applying **hand-optimized instructions**, such as loop unrolling, fused multiply-add (FMA), or SIMD instructions like `PADDUSB`, `MOVDQA`, or `VPMADDWD`.
  - Writing **compiler-specific optimizations**, such as those relying on instruction flags or condition codes.
- **Example: Inline Assembly for 32-bit Pixel Alpha Blending (GCC, Intel Syntax)**

```
void blend_pixel(uint32_t* dst, uint32_t src, uint8_t alpha) {
    __asm__ __volatile__ (
        "movd    %1, %%xmm0\n\t"         // Load src
        "movd    (%0), %%xmm1\n\t"       // Load dst
        "punpcklbw %%xmm0, %%xmm0\n\t"   // Unpack src
        "punpcklbw %%xmm1, %%xmm1\n\t"  // Unpack dst
        "psubusb %%xmm1, %%xmm0\n\t"    // src - dst
        "pmulhuw %%xmm2, %%xmm0\n\t"    // Multiply by alpha
        "paddusb %%xmm1, %%xmm0\n\t"    // Add back to dst
        "packuswb %%xmm0, %%xmm0\n\t"   // Pack result
        "movd    %%xmm0, (%0)\n\t"      // Store back to dst
        :
        : "r"(dst), "r"(src), "x"(_mm_set1_epi16(alpha))
        : "xmm0", "xmm1"
    );
}
```

This style is best suited for **hot path graphics routines** that require SIMD control the compiler might not generate.

13.1.4 External Assembly Files: Overview and Use Cases

External assembly refers to writing complete assembly routines or modules in `.asm` or `.S` (UNIX) files, then linking them with the main application during the build process.

- **Supported Formats**
  - **GAS (GNU Assembler)**: Preferred on Linux; supports AT&T and Intel syntax.
  - **NASM or YASM**: Popular for writing clean, portable x86 code.
  - **MASM**: Used with MSVC on Windows, ideal for Windows-specific calls and Intel syntax.
  - **FASM**: Lightweight assembler suitable for minimal runtime or educational tools.

- **Advantages of External Assembly**

  – Full control over labels, segments, data sections.

  – Clean separation between C++ logic and low-level instructions.

  – Easily reused across different projects or platforms.

  – Compatible with both 32-bit and 64-bit builds.

  – Often easier to write and maintain when routines grow complex.

- **When to Use External Assembly**

  Use external assembly when:

  – The routine is **large or standalone** (e.g., a rasterizer, DCT function, polygon filler).

  – You need to **optimize independent from compiler rules**.

  – You require **cross-compiler** support (NASM + GCC, MASM + MSVC, etc.).

  – You need **platform-specific optimizations**, such as syscall wrappers or BIOS calls.

  – Debugging needs **symbol separation** or the code interacts with multiple language boundaries.

- **Example: External Assembly Function for Memory Fill (NASM Syntax)**

  memfill.asm

```
global memfill

section .text
memfill:
    ; rdi = destination
    ; rcx = count
    ; al  = value to fill

    rep stosb
    ret
```

C++ Integration

```
extern "C" void memfill(void* dest, size_t count, uint8_t value);

void clear_screen(uint8_t* buffer, size_t size) {
    memfill(buffer, size, 0);
}
```

13.1.5 Platform and Toolchain Considerations

- **On Linux**
  - Use **GAS (.S files)** with `gcc` or `clang`.
  - Choose **Intel syntax** (`.intel_syntax noprefix`) for clarity.
  - Avoid hardcoding CPU features; detect or define via `#ifdef`.
- **On Windows**
  - Use **MASM** with `ml.exe` or **YASM/NASM** with MSVC's linker.

  - Inline assembly in 64-bit builds is not supported in MSVC; prefer external files or intrinsics.
  - Use `__declspec(naked)` for minimal entry/exit overhead if needed.

- **Cross-Platform Strategy**
  - Keep assembly isolated behind a uniform C API.
  - Use conditional compilation to choose between inline/external paths.
  - Use intrinsics (`<immintrin.h>`) when portability across compilers is needed but full assembly isn't.

13.1.6 Inline vs External: Decision Table

Table 1-1: Comparison of Inline vs External Assembly

Criteria	**Inline Assembly**	**External Assembly**
Routine size	Small (20 lines)	Medium to large
Portability	Compiler-specific	Toolchain-specific
Cross-platform reuse	Difficult	Easier with NASM/GAS
Debugging	Integrated	Easier with symbols
64-bit MSVC compatibility	Not supported	Required
Tight C/C++ variable access	Easier	Must use ABI rules
Optimal for SIMD loops	Yes	Yes
Best for reusable libraries	No	Yes

13.1.7 Practical Advice

- Always **profile before optimizing**. Use tools like `perf`, `gprof`, `VTune`, or Visual Studio Profiler.
- **Measure gains** objectively—many modern compilers match hand-written code unless algorithmic changes are made.
- For vector-heavy code, try **intrinsics** first before full assembly (they're safer and portable).
- Document assumptions: CPU features (e.g., AVX2), OS-specific behavior, alignment guarantees.
- Use inline assembly only when portability and toolchain consistency are ensured.

13.1.8 Summary

The decision to use inline or external assembly in a CPU-only graphics system hinges on **scope**, **complexity**, **toolchain**, and **portability requirements**. Inline assembly is ideal for small, performance-sensitive routines deeply integrated into C++ logic. External assembly offers full control and reusability for larger or lower-level modules. Both forms serve a crucial role in high-performance software graphics development, particularly where cache alignment, SIMD parallelism, or custom rendering logic are necessary to achieve real-time frame rates in the absence of GPU acceleration.

13.2 Intel vs. AT&T Syntax

13.2.1 Introduction

One of the most critical decisions a developer faces when incorporating assembly into a CPU-based graphics system is the choice of syntax: **Intel syntax** or **AT&T syntax**. Both are supported across various compilers and assemblers but differ significantly in format, operand ordering, and notation. Understanding their distinctions is essential not only for correctness but also for maximizing clarity, maintainability, and collaboration with tools and documentation.

This section provides a detailed comparison between Intel and AT&T syntax, outlines how to select and switch between them, and explains the practical implications of each style in real-world graphics programming scenarios.

13.2.2 Overview of the Two Syntaxes

- **Intel Syntax**

  Intel syntax is the notation used by Intel's own documentation, Microsoft's MASM assembler, NASM, and most x86-focused tutorials. It is widely used in Windows-based environments and is often favored for its **readability** due to its close resemblance to high-level logic expressions.

  Key Characteristics:

  - Destination comes **first**: `mov eax, ebx` (move value from `ebx` to `eax`)
  - Immediate values are unprefixed: `mov eax, 10`
  - Registers are bare: `eax`, `esi`
  - Memory operands are enclosed in **square brackets**: `[eax]`

- **AT&T Syntax**

  AT&T syntax is the default for the GNU assembler (**GAS**), and is primarily used in UNIX and Linux-based systems. It is **more verbose**, with strict conventions for operand types and sizes.

  Key Characteristics:

  – Destination comes **last**: `movl %ebx, %eax` (same meaning as above)

  – Immediate values prefixed with `$`: `movl $10, %eax`

  – Registers prefixed with `%`: `%eax`, `%esi`

  – Memory operands use **parentheses**: `(%eax)`

13.2.3 Side-by-Side Comparison

Table 2-2: Comparison of Intel vs AT&T Assembly Syntax

Operation	**Intel Syntax**	**AT&T Syntax**
Move	`mov eax, ebx`	`movl %ebx, %eax`
Immediate	`mov eax, 5`	`movl $5, %eax`
Memory	`mov eax, [ebx+4]`	`movl 4(%ebx), %eax`
Add	`add eax, 1`	`addl $1, %eax`
Multiply	`imul eax, ecx, 2`	`imull $2, %ecx, %eax`
Function	`call my_function`	`call my_function (same)`

Note:

- AT&T uses **instruction suffixes** like `b`, `w`, `l`, `q` for byte, word, long, and quad.
- Intel omits suffixes, inferring size from operands or using `BYTE PTR`, `DWORD PTR`, etc., when needed.

13.2.4 Syntax Usage in Tools

- **On Linux with GCC or Clang**

  By default, **GAS** uses AT&T syntax. To use Intel syntax:

  – For inline assembly, use the `.intel_syntax noprefix` directive.

  – For external `.S` files, you can wrap Intel-style code with:

```
.intel_syntax noprefix
mov eax, [ebx+4]
```

  – Clang also supports inline Intel syntax similarly.

- **On Windows with MSVC**

  – MSVC only supports **Intel syntax** in its legacy 32-bit inline assembly using `__asm`.

  – For 64-bit code, MSVC requires intrinsics or external `.asm` files assembled with MASM (which uses Intel syntax by default).

  - NASM and YASM are Intel-only assemblers and widely used in performance-sensitive libraries like x264, LAME, and zlib.

- **NASM/YASM**

  These use **Intel syntax** exclusively. They are popular choices for writing portable, high-performance assembly code, including multimedia codecs and game engines.

- **GAS (GNU Assembler)**

  By default, GAS uses **AT&T syntax**. However, developers can switch to Intel using:

  ```
  .intel_syntax noprefix
  ...
  .att_syntax
  ```

  This makes it possible to write inline Intel syntax inside C/C++ code even on Linux, which is helpful when sharing codebases between Windows and Linux.

13.2.5 Practical Considerations for CPU Graphics Developers

- **When to Use Intel Syntax**

  - You are familiar with x86 programming from Intel or AMD documentation.
  - You are writing code for Windows or cross-platform projects using NASM/YASM.
  - You are integrating with other Intel-style codebases (e.g., SIMD-heavy image filters).
  - You want clearer, less verbose code for large functions with many operands.

- **When to Use AT&T Syntax**
  - You are developing on Linux or BSD using `gas` or `gcc`.
  - You are maintaining legacy Unix software or open-source tools written in AT&T.
  - You rely on the default behavior of GCC or Clang for inline assembly.
- **Translation Complexity**

  Converting from one syntax to another is not just a matter of rearranging operands:
  - Memory references must change from `[%reg]` to `(%reg)`
  - Registers need `%` prefix in AT&T
  - Immediate values need `$`
  - Operand order is reversed
  - Suffixes (`b`, `w`, `l`, `q`) must be added in AT&T

  This makes automated conversion difficult for anything more than trivial code.

13.2.6 Example: Pixel Brightness Adjustment

Intel Syntax (NASM/YASM)

```
mov eax, [esi]        ; Load pixel value
shr eax, 1            ; Reduce brightness by 50%
mov [edi], eax        ; Store adjusted pixel
```

AT&T Syntax (GAS)

```
movl (%esi), %eax   # Load pixel value
shrl $1, %eax       # Reduce brightness
movl %eax, (%edi)   # Store adjusted pixel
```

Despite identical logic, the syntactic burden is heavier in AT&T, especially when dealing with many registers or memory operations.

13.2.7 Mixed Syntax Development

Many modern projects need to support **both Linux and Windows**, which may involve switching between compilers and assemblers. A common practice is to:

- Use **intrinsics** when performance needs are moderate and portability is critical.
- Use **external Intel syntax** with NASM or YASM for low-level functions.
- Isolate all assembly behind `extern "C"` functions to hide the syntax details from the rest of the application.

In shared graphics libraries or CPU renderers, defining macros or including syntax switching in build scripts helps maintain consistency.

13.2.8 Summary

Intel and AT&T syntax represent two ways of expressing the same low-level machine instructions. Intel syntax, being more readable and closer to hardware documentation, is the preferred choice in Windows, embedded systems, and portable performance libraries. AT&T syntax, while less intuitive, remains deeply integrated into the Unix and GCC world.
For modern CPU-only graphics development, particularly cross-platform work after 2019, developers are encouraged to:

- Favor **Intel syntax** when working with external files or SIMD.
- Use **`.intel_syntax noprefix`** for inline assembly under GCC/Clang if needed.
- Maintain consistent assembly style across platforms using wrappers or macros.
- Keep performance-critical sections in **well-isolated modules**, regardless of syntax.

Understanding both syntaxes provides the flexibility needed to optimize rendering and image-processing operations in any environment, from Windows GUI applications to Linux framebuffer-based engines.

13.3 Loop Unrolling and Pixel Blitting in x86 Assembly

13.3.1 Introduction

Pixel blitting—short for **bit-block transfer**—is a fundamental operation in 2D graphics rendering, used to copy pixel data from a source buffer to a destination buffer, typically representing images, sprites, or framebuffers. When operating in a **CPU-only graphics environment**, especially without the assistance of a GPU or hardware acceleration, every optimization at the memory and instruction level becomes critical. This section focuses on **loop unrolling**—a classic assembly-level optimization—and how it can significantly enhance **pixel blitting performance** on modern x86 processors. We explore techniques for manual unrolling, register utilization, memory prefetching, and alignment handling, all in the context of writing high-performance blitting routines using x86 assembly. Emphasis is placed on techniques validated for CPUs and tools used after 2019, ensuring relevance to modern development.

13.3.2 Fundamentals of Pixel Blitting

Pixel blitting involves copying a rectangular block of pixels from a source to a destination, with variations depending on color depth, pixel format (e.g., RGB, ARGB, grayscale), and blending rules.
In its simplest form:

- **Source:** Linear buffer (e.g., bitmap or sprite)
- **Destination:** Framebuffer or another image buffer
- **Operation:** Copy N bytes from source to destination per scanline

Example in C (24-bit RGB blit):

```
for (int y = 0; y < height; ++y)
    memcpy(dst + y * pitch, src + y * width * 3, width * 3);
```

While `memcpy` is efficient for general-purpose copying, in CPU graphics applications that require per-pixel blending, masking, or alignment, **custom blitters** using x86 assembly can yield much better performance.

13.3.3 Concept of Loop Unrolling

Loop unrolling reduces loop overhead by explicitly repeating loop bodies multiple times per iteration, minimizing jump and condition-checking instructions. This optimization is critical when working in tight memory-copy loops, such as blitting pixels:
Benefits include:

- Fewer branches (conditional jumps)
- Better instruction pipelining
- Higher instruction-level parallelism
- More effective register reuse

Consider the naïve assembly version of a pixel copy loop:

```
mov ecx, width
copy_loop:
    mov eax, [esi]        ; Load pixel from src
    mov [edi], eax        ; Store to dst
    add esi, 4
    add edi, 4
    loop copy_loop
```

Each iteration copies a single 32-bit pixel and includes three arithmetic operations and a loop instruction. Now compare that with an unrolled version:

```
mov ecx, width / 4       ; Process 4 pixels per loop
copy_loop:
    mov eax, [esi]
    mov ebx, [esi+4]
    mov edx, [esi+8]
    mov esi, [esi+12]
    mov [edi], eax
    mov [edi+4], ebx
    mov [edi+8], edx
    mov [edi+12], esi
    add esi, 16
    add edi, 16
    loop copy_loop
```

This approach **reduces branch frequency by 75%**, leading to faster execution, especially on CPUs where branch misprediction or loop overhead can become a bottleneck.

13.3.4 Writing a Blitter in x86 Assembly (32-bit ARGB)

Let us write a practical blitting function that copies a 32-bit per pixel (ARGB) image using x86 assembly with manual unrolling.

- **Assumptions**
  - Source and destination are 4-byte aligned.

  - Width is divisible by 8.
  - No alpha blending, just raw memory copy.

- **NASM Example: `blit_32_unrolled.asm`**

```
global blit_32_unrolled

; Parameters:
;   esi = source pointer
;   edi = destination pointer
;   ecx = number of pixels (must be multiple of 8)

section .text
blit_32_unrolled:
.loop:
    mov eax, [esi]          ; pixel 0
    mov ebx, [esi+4]        ; pixel 1
    mov edx, [esi+8]        ; pixel 2
    mov esi, [esi+12]       ; pixel 3
    mov [edi], eax
    mov [edi+4], ebx
    mov [edi+8], edx
    mov [edi+12], esi

    mov eax, [esi+16]       ; pixel 4
    mov ebx, [esi+20]       ; pixel 5
    mov edx, [esi+24]       ; pixel 6
    mov esi, [esi+28]       ; pixel 7
    mov [edi+16], eax
    mov [edi+20], ebx
    mov [edi+24], edx
    mov [edi+28], esi
    add esi, 32
    add edi, 32
    sub ecx, 8
    jnz .loop
    ret
```

This example uses **eight 4-byte loads and stores per loop iteration**, minimizing jumps while maximizing throughput.

13.3.5 Enhancing the Blitter: Prefetching and Alignment

To fully utilize the CPU's memory subsystem, developers can apply:

- **Data Prefetching**

  Manually hint the processor to pre-load data into the L1/L2 cache before access:

  ```
  prefetchnta [esi+64]   ; Non-temporal prefetch (avoid polluting
  ↪   cache)
  ```

  Adding this 1–2 iterations ahead of memory reads reduces stall cycles when blitting large images.

- **Memory Alignment**

  – Misaligned memory accesses can reduce speed or even trigger faults on strict platforms.

  – Ensure both `esi` and `edi` are **16-byte aligned** for best performance with modern CPUs and SIMD instructions.

13.3.6 SIMD Accelerated Unrolling (SSE2)

For processors supporting **SSE2 or higher**, SIMD registers (`xmm0–xmm15`) can be used to copy multiple pixels at once.

- **Example: Using SSE for Unrolled Blitting**

```
movdqu xmm0, [esi]      ; Load 4 pixels (16 bytes)
movdqu xmm1, [esi+16]   ; Load next 4 pixels
movdqu [edi], xmm0
movdqu [edi+16], xmm1
add esi, 32
add edi, 32
```

  - Use `movdqu` for unaligned memory, `movdqa` for aligned.
  - AVX/AVX2 can widen this to 256-bit (8 pixels at once).

13.3.7 Loop Unrolling in 64-bit Systems

Modern 64-bit CPUs have more registers (`r8–r15`), allowing deeper unrolling without spilling. Use this register abundance for higher throughput.
For instance:

- Load 8 pixels using SSE/AVX
- Assign dedicated registers for index, counter, base pointers
- Unroll up to 16 pixels per loop

13.3.8 Performance Results and Profiling

Modern tools for validating blitter performance include:

- **Linux:** `perf`, `valgrind --tool=callgrind`
- **Windows:** Windows Performance Analyzer, Visual Studio Profiler
- **Cross-platform:** Intel VTune, Cachegrind

Measure:

- CPU cycles per byte copied
- Memory throughput (MB/s)
- Frame time impact in full rendering pipeline

13.3.9 Summary

Loop unrolling is a highly effective optimization for CPU-based graphics routines such as pixel blitting. By reducing control overhead and exploiting memory throughput, developers can significantly enhance rendering speed. When combined with SIMD instructions, memory prefetching, and register scheduling, loop unrolling becomes a critical component in building high-performance software graphics engines.
For modern CPUs and compilers post-2019, explicit unrolling remains relevant in situations where:

- Compiler optimizations are insufficient
- Precise instruction layout is needed
- Software rendering paths must match or exceed hardware expectations

By mastering these techniques, CPU graphics programmers can ensure their engines remain responsive, efficient, and platform-agnostic.

13.4 Writing a Hand-Optimized Blitter

13.4.1 Introduction

In a CPU-only rendering pipeline, the **blitter** (short for bit-block transfer engine) is essential for moving pixel data efficiently. It is the core of many visual operations such as sprite drawing, background scrolling, image compositing, and framebuffer refresh. While high-level languages offer abstractions like `memcpy`, `std::copy`, or hardware APIs, these are often suboptimal for the performance-critical inner loops of real-time software graphics.

This section guides the reader through designing and implementing a **hand-optimized blitter** in assembly, tailored for modern x86_64 CPUs. It includes techniques for register allocation, loop unrolling, SIMD instruction usage, alignment handling, and performance tuning, all grounded in practices compatible with modern operating systems and toolchains used after 2019.

13.4.2 Problem Statement and Assumptions

We aim to copy a rectangular block of pixels from a source to a destination buffer:

- **Pixel Format**: 32-bit ARGB (1 pixel = 4 bytes)
- **Alignment**: Both buffers are 16-byte aligned
- **Size**: Width and height of region are arbitrary, though width is preferably a multiple of 4 or 8 for SIMD
- **Stride**: Source and destination pitches may differ
- **Blending**: No blending, raw copy

This case reflects typical use in software-based image viewers, 2D game engines, and GUI toolkits.

13.4.3 Naïve Implementation in Assembly

A simple blitter in NASM syntax for 64-bit platforms:

```
global blit_naive

; void blit_naive(uint32_t* dst, uint32_t* src, size_t pixels);
blit_naive:
    ; Inputs:
    ; rdi = dst
    ; rsi = src
    ; rdx = number of pixels

.loop:
    cmp rdx, 0
    je .end
    mov eax, [rsi]
    mov [rdi], eax
    add rsi, 4
    add rdi, 4
    dec rdx
    jmp .loop

.end:
    ret
```

This code is simple but slow due to:

- High instruction count per pixel

- No unrolling or pipelining
- No SIMD usage
- Poor cache-line utilization

13.4.4 Loop Unrolling and Register Reuse

Unrolling the loop to process 4 pixels per iteration:

```
blit_unrolled:
.loop:
    cmp rdx, 4
    jb .tail

    mov eax, [rsi]
    mov ebx, [rsi+4]
    mov ecx, [rsi+8]
    mov edx, [rsi+12]
    mov [rdi], eax
    mov [rdi+4], ebx
    mov [rdi+8], ecx
    mov [rdi+12], edx

    add rsi, 16
    add rdi, 16
    sub rdx, 4
    jmp .loop

.tail:
    test rdx, rdx
    je .end
.tail_loop:
    mov eax, [rsi]
    mov [rdi], eax
    add rsi, 4
    add rdi, 4
    dec rdx
    jnz .tail_loop

.end:
    ret
```

Benefits:

- Reduced loop overhead
- Improved ILP (instruction-level parallelism)
- Better performance even on in-order cores

13.4.5 SIMD Optimized Blitter Using SSE2

If the system supports **SSE2**, 16 bytes (4 pixels) can be copied at once using `movdqa` for aligned data:

```
blit_sse2:
.loop:
    cmp rdx, 4
    jb .tail

    movdqa xmm0, [rsi]          ; Load 4 pixels
    movdqa [rdi], xmm0          ; Store 4 pixels

    add rsi, 16
    add rdi, 16
    sub rdx, 4
    jmp .loop

.tail:
    test rdx, rdx
    je .end
.tail_loop:
    mov eax, [rsi]
    mov [rdi], eax
    add rsi, 4
    add rdi, 4
    dec rdx
    jnz .tail_loop

.end:
    ret
```

Considerations:

- `movdqa` requires 16-byte alignment. Use `movdqu` if alignment cannot be guaranteed.
- SSE2 is available on all x86_64 CPUs since the early 2000s, and still applicable post-2019.
- On modern OSes, memory allocators like `posix_memalign` (Linux) or `_aligned_malloc` (Windows) should be used to align buffers.

13.4.6 Using AVX2 for Wider Copies (Optional)

With **AVX2**, 256-bit registers allow 8 pixels per operation:

```
vmovdqa ymm0, [rsi]      ; Load 8 pixels
vmovdqa [rdi], ymm0      ; Store 8 pixels
```

Requirements:

- Aligned 32-byte buffers
- Compiler or inline assembly support for AVX2 (GCC, Clang, MSVC)
- OS support for AVX2 context saving (Windows 10+, Linux kernel 3.16+)

Note: Since AVX2 uses more power and can throttle CPU clocks, test performance trade-offs.

13.4.7 Memory Prefetching

To reduce cache misses:

```
prefetchnta [rsi + 64]  ; Prefetch data 64 bytes ahead
```

This is effective in tight loops and large images. Do not prefetch too far ahead or into wrong cache levels.

13.4.8 Performance Testing

Test the blitter with:

- Varying image sizes (small icons to full-screen buffers)
- Cold and warm cache states
- Varying alignment (unaligned vs aligned)

Tools:

- Linux: `perf`, `valgrind`, `cachegrind`
- Windows: Windows Performance Analyzer, Intel VTune
- Cross-platform: Google Benchmark

13.4.9 Summary

Hand-optimized blitters can significantly outperform general-purpose memory copy functions, especially in software-rendered environments. By unrolling loops, using SIMD instructions, ensuring memory alignment, and minimizing branching, developers can design CPU-only renderers capable of high frame rates even under tight hardware constraints.
When developing for modern systems (post-2019), these practices remain relevant due to:

- Wider CPU vector registers (AVX2/AVX-512)
- Better memory bandwidth
- Threaded software renderers with per-core workloads

Hand-optimized assembly routines are powerful tools in the software graphics toolbox, particularly when performance and determinism are priorities.

Chapter 14

Debugging and Testing Graphics Code

14.1 Visual Debugging Techniques

14.1.1 Introduction

Debugging graphics code that runs entirely on the CPU—without the aid of GPU toolchains—requires specialized strategies. Unlike traditional text or logic debugging, graphics debugging often deals with visual artifacts, timing issues, off-screen rendering, and frame-based behaviors. For this reason, **visual debugging techniques** are indispensable. These techniques transform the internal state of the rendering pipeline into directly viewable cues that aid in identifying errors and performance issues in real-time.

This section explores professional techniques used for visual debugging in software-based rendering engines. The approaches presented are applicable to modern Windows and Linux environments and assume development practices post-2019, using either C or

C++ with optional inline or external assembly.

14.1.2 Principle of Visibility: Make the Internal Visible

Graphics rendering is inherently visual, yet most internal logic operates on abstract buffers, transformation states, or memory structures. A foundational debugging strategy is:

"Make the invisible visible."

This includes:

- Rendering internal data as overlays (e.g., bounding boxes, clip regions)
- Drawing diagnostic information directly on the screen
- Encoding pixel metadata as color codes
- Storing debug views as image files

These methods are especially effective because they operate within the very medium you're debugging—pixels.

14.1.3 Common Visual Debugging Techniques

1. **Bounding Box Overlay**

   Bounding boxes are a fast and effective way to confirm that objects are being placed where they are expected.

   Implementation:

   - Draw a 1-pixel wide rectangle around each object
   - Use a unique color for each category (red for enemies, green for UI, blue for tilemaps)

Code example (pseudo-C):

```c
void draw_debug_box(uint32_t* framebuffer, int pitch, int x, int y,
 ↪  int w, int h, uint32_t color) {
    for (int i = 0; i < w; ++i) framebuffer[y * pitch + x + i] =
     ↪  color;
    for (int i = 0; i < w; ++i) framebuffer[(y + h - 1) * pitch + x
     ↪  + i] = color;
    for (int j = 0; j < h; ++j) framebuffer[(y + j) * pitch + x] =
     ↪  color;
    for (int j = 0; j < h; ++j) framebuffer[(y + j) * pitch + x + w
     ↪  - 1] = color;
}
```

2. **Pixel Value Encoding**

   Instead of debugging numeric variables in a console, encode them visually:

   - **Alpha channels** can represent transparency logic
   - **Color channels** can represent states: e.g., red = not visible, green = culled, yellow = active

   Example: Use `uint32_t color = (state << 16) | 0xFF00FF00;` to mark active elements.

3. **Z-buffer or Depth Visualization**

   If implementing software-based Z-buffering or depth sorting:

   - Visualize depth as grayscale: closer = brighter
   - Store depth as a float, then map it to 8-bit luminance range

This helps uncover z-fighting, incorrect depth ordering, or improperly cleared buffers.

4. **Frame Step Debugging**

   For dynamic scenes:

   - Allow stepping through frames with a keyboard input (`SPACE`, `N`, etc.)
   - Freeze rendering and show overlays per-frame
   - Useful for catching frame-to-frame errors (incorrect delta time, off-by-one errors)

   This can be built using flags like:

```
bool pause_debug = true;
if (user_pressed('N')) {
    pause_debug = false;
}
```

5. **Diagnostic Text Rendering**

   Use built-in bitmap or FreeType text rendering to:

   - Print out FPS, frame count, object counts, memory usage
   - Output debug state of current selected or hovered object

   Tools like Dear ImGui use a similar approach in GPU environments. In CPU-only environments, draw small monospace bitmap fonts in a debug HUD.

14.1.4 Saving Frame Snapshots for Offline Analysis

Render output can be saved to BMP, PNG, or raw binary formats for later analysis using offline tools like GIMP or ImageMagick. This is helpful in headless or crash-prone debugging sessions.
Basic steps:

1. Dump framebuffer to a file per frame or on error.
2. Use indexed filenames (e.g., `frame_1234.png`)
3. Analyze frame-by-frame progression postmortem.

BMP is often used due to its simplicity and no need for external libraries.

14.1.5 Annotated Blitting and Layer Tracking

Instrument the blitting system itself to leave visual breadcrumbs:

- Fill source rectangles with semi-transparent patterns
- Mark "touched" layers in red
- Toggle "layer outlines" at runtime with a hotkey

This helps with Z-order issues, stale buffer content, or incorrect layer compositing.

14.1.6 Reproducible Test Modes

To make debugging consistent:

- Add a `--debug-seed <number>` CLI parameter to initialize PRNG or timers

- Allow fixed-time-step simulation (e.g., simulate 16.67ms per frame)
- Disable user input or simulate input sequences from a log

This allows bugs to be reliably reproduced and captured visually.

14.1.7 Cross-Platform Considerations (Windows/Linux)

Visual debugging should be supported identically on all target platforms:

- On **Windows**, GDI or `BitBlt()` can show debug overlays
- On **Linux**, Xlib or framebuffer access can be used
- Use platform-specific keystroke handlers to toggle overlays (`GetAsyncKeyState()` on Windows, `XQueryKeymap()` on Linux)

Debug layers should be optional and togglable at runtime to avoid degrading performance during normal use.

14.1.8 Real-World Debugging Scenario

Symptom: An object flickers or vanishes intermittently.
Steps:

1. Enable bounding box overlay – reveals object is drawn partially off-screen.
2. Enable diagnostic text – shows object has negative Y due to bad velocity.
3. Add depth visualization – confirms it's behind background unexpectedly.

Conclusion: A faulty movement logic plus depth handling error causes the glitch. All discovered via visual debugging, not console logs.

14.1.9 Summary

Visual debugging is not just a convenience—it's an essential skill for CPU-based graphics development. It provides immediate, frame-accurate, spatial insight into issues that are otherwise invisible to console output or symbolic debuggers. By encoding internal states as visible structures or overlays, developers can iterate faster, locate bugs more reliably, and better understand the behavior of their rendering code.

These methods are especially useful in constrained environments or platforms lacking mature GPU debugging tools, making them a cornerstone of serious software rendering engines.

14.2 Pixel Dumpers to PPM or BMP Files

14.2.1 Introduction

In CPU-only graphics programming, visual debugging extends beyond on-screen overlays to offline analysis by saving rendered frames to image files. This allows developers to inspect, compare, and archive graphical output outside the running application environment. Dumping raw framebuffer data to simple, widely supported image formats like **PPM (Portable Pixmap)** or **BMP (Bitmap)** is a straightforward and effective approach.
This section details how to implement pixel dumpers for PPM and BMP files, the pros and cons of each format, and best practices for integrating pixel dumping into debugging workflows on modern Windows and Linux platforms post-2019.

14.2.2 Why Dump Pixels to Image Files?

Dumping framebuffer pixels to disk serves several critical debugging purposes:

- **Offline Analysis:** Examine the exact frame output with high-fidelity tools (image viewers, editors) unavailable in the live application.
- **Regression Testing:** Automatically compare output frames against golden references to detect visual regressions.
- **Crash Investigation:** Capture the last rendered frame before a crash to assist diagnosis.
- **Performance Profiling:** Visualize intermediate stages (e.g., after transformations or blending) without disrupting real-time rendering.

Dumping is especially useful when running headless tests or remote debugging sessions without live display access.

14.2.3 Choosing Image Formats: PPM vs BMP

1. **PPM (Portable Pixmap)**

   - **Simplicity:** PPM is a plain-text or binary format storing pixel data as RGB triples. It has a minimal header and no compression.
   - **Portability:** Supported by many image viewers and conversion tools on Linux and Windows.
   - **Use Case:** Ideal for quick dumps and debugging due to its trivial file structure.
   - **Drawbacks:** Large file sizes and lack of metadata support (e.g., alpha channels).

2. **BMP (Bitmap)**

   - **Structure:** BMP is a widely used Windows bitmap format with a well-defined header, supporting various pixel formats including 24-bit RGB and 32-bit RGBA.
   - **Compression:** Typically uncompressed (though RLE compression exists), resulting in moderate file sizes.
   - **Use Case:** Preferred for Windows-centric workflows and when storing alpha or palette information.
   - **Drawbacks:** Slightly more complex to implement than PPM.

14.2.4 Writing a PPM Pixel Dumper

The PPM file format starts with a header followed by raw RGB pixel data. The most common variant is the binary ”P6” format.

1. **PPM File Structure**

```
P6 <width> <height>
255 <binary RGB pixel data>
```

- `P6`: Magic number indicating binary PPM
- `<width> <height>`: Image dimensions in pixels
- `255`: Max color value per channel (8-bit)
- Following: RGB triplets for each pixel row-wise, top to bottom, left to right

2. **Implementation Example (C)**

```c
#include <stdio.h>
#include <stdint.h>

void dump_framebuffer_to_ppm(const char* filename, uint32_t*
 ↪   framebuffer, int width, int height) {
    FILE* f = fopen(filename, "wb");
    if (!f) return;

    // Write PPM header
    fprintf(f, "P6\n%d %d\n255\n", width, height);

    // Write pixel data (RGB)
    for (int y = 0; y < height; ++y) {
        for (int x = 0; x < width; ++x) {
            uint32_t pixel = framebuffer[y * width + x];
            // Extract 8-bit channels (assuming ARGB format)
            uint8_t r = (pixel >> 16) & 0xFF;
            uint8_t g = (pixel >> 8) & 0xFF;
            uint8_t b = pixel & 0xFF;
            fputc(r, f);
            fputc(g, f);
            fputc(b, f);
        }
    }
    fclose(f);
}
```

Notes:

- Assumes a 32-bit framebuffer in ARGB or similar format.
- Ignores alpha channel; PPM does not support alpha.

14.2.5 Writing a BMP Pixel Dumper

The BMP format requires more structure: file headers, info headers, and pixel data, often with row padding to 4-byte alignment.

1. **BMP File Structure**

   - **BITMAPFILEHEADER** (14 bytes): Signature, file size, reserved fields, pixel data offset
   - **BITMAPINFOHEADER** (40 bytes): Image width, height, planes, bits per pixel, compression, image size, etc.

- **Pixel Data**: Stored bottom-up (last row first), padded per row

2. **Implementation Outline (C)**

```
#include <stdint.h>
#include <stdio.h>

#pragma pack(push, 1)
typedef struct {
    uint16_t bfType;      // 'BM' = 0x4D42
    uint32_t bfSize;      // File size in bytes
    uint16_t bfReserved1; // 0
    uint16_t bfReserved2; // 0
    uint32_t bfOffBits;   // Offset to pixel data
} BITMAPFILEHEADER;

typedef struct {
    uint32_t biSize;          // Header size (40 bytes)
    int32_t  biWidth;
    int32_t  biHeight;
    uint16_t biPlanes;        // Must be 1
    uint16_t biBitCount;      // Bits per pixel (24 or 32)
    uint32_t biCompression;   // 0 = BI_RGB (no compression)
    uint32_t biSizeImage;     // Image size in bytes (can be 0 for
    ↪  BI_RGB)
    int32_t  biXPelsPerMeter; // Horizontal resolution
    int32_t  biYPelsPerMeter; // Vertical resolution
    uint32_t biClrUsed;       // Number of colors
    uint32_t biClrImportant;  // Important colors
} BITMAPINFOHEADER;
#pragma pack(pop)

void dump_framebuffer_to_bmp(const char* filename, uint32_t*
↪  framebuffer, int width, int height) {
    FILE* f = fopen(filename, "wb");
    if (!f) return;

    const int bytes_per_pixel = 3; // Writing 24-bit BMP (RGB)
    int row_padded = (width * bytes_per_pixel + 3) & (~3);
    int filesize = 14 + 40 + row_padded * height;

    BITMAPFILEHEADER file_header = {
        .bfType = 0x4D42,
        .bfSize = filesize,
        .bfReserved1 = 0,
        .bfReserved2 = 0,
        .bfOffBits = 14 + 40
    };

    BITMAPINFOHEADER info_header = {
        .biSize = 40,
        .biWidth = width,
        .biHeight = height,
        .biPlanes = 1,
        .biBitCount = 24,
        .biCompression = 0,
        .biSizeImage = row_padded * height,
        .biXPelsPerMeter = 0,
        .biYPelsPerMeter = 0,
        .biClrUsed = 0,
        .biClrImportant = 0
    };

    fwrite(&file_header, sizeof(file_header), 1, f);
    fwrite(&info_header, sizeof(info_header), 1, f);
    // Write pixels bottom-up
    uint8_t* row = (uint8_t*)malloc(row_padded);
    for (int y = height - 1; y >= 0; --y) {
        for (int x = 0; x < width; ++x) {
            uint32_t pixel = framebuffer[y * width + x];
            // Extract RGB, ignoring alpha
            row[x * 3 + 0] = pixel & 0xFF;          // Blue
            row[x * 3 + 1] = (pixel >> 8) & 0xFF;   // Green
            row[x * 3 + 2] = (pixel >> 16) & 0xFF;  // Red
        }
        // Pad remaining bytes with zero
        for (int p = width * 3; p < row_padded; ++p) {
            row[p] = 0;
        }
        fwrite(row, 1, row_padded, f);
    }
    free(row);
    fclose(f);
}
```

Notes:

- The BMP format requires pixel rows to be aligned to multiples of 4 bytes.
- Pixels are stored in BGR order, bottom-to-top.
- Alpha channel is ignored for simplicity but can be stored in 32-bit BMP with more complex headers.

14.2.6 Integration in Debugging Workflow

For maximum utility, pixel dumping should be:

- **Triggered on demand:** Via keyboard shortcuts or command-line options.
- **Configurable:** Allow developers to choose format (PPM or BMP), output directory, and naming conventions.
- **Efficient:** Minimal runtime impact by performing file writes asynchronously if possible.
- **Annotative:** Optionally overlay debug info (e.g., bounding boxes, coordinates) before dumping.

Example usage flow:

```
if (dump_frame_requested) {
    char filename[256];
    snprintf(filename, sizeof(filename), "dump_%04d.bmp", frame_number);
    dump_framebuffer_to_bmp(filename, framebuffer, width, height);
}
```

14.2.7 Platform-Specific Considerations

- **Windows:** Use standard C file I/O; consider Windows high-performance file APIs (e.g., `CreateFile` with asynchronous writes) for larger projects.
- **Linux:** Standard POSIX file I/O suffices; flushing data with `fsync()` can ensure frame integrity on crash.
- File permissions and directories should be checked to avoid runtime errors.

14.2.8 Performance Notes

- Writing full-resolution framebuffer dumps every frame is expensive. Use frame skipping (e.g., every 30 frames) or selective dumping on errors.
- Binary PPM and uncompressed BMP are large; consider compressing dumps offline.
- Dumping can be combined with memory-mapped files for fast access during debugging.

14.2.9 Summary

Pixel dumpers to PPM or BMP provide an essential bridge between raw framebuffer memory and human-readable, analyzable images. PPM offers simplicity and portability ideal for quick debugging, while BMP offers richer compatibility with Windows tooling and supports features like alpha channels with moderate complexity.
In CPU-only graphics programming, mastering pixel dumping techniques significantly enhances the developer's ability to debug, test, and verify graphical output, especially when live visual debugging or GPU-based tools are unavailable.

14.3 Writing Unit Tests for Rendering (Checksum-Based)

14.3.1 Introduction

In CPU-only graphics programming, ensuring correctness of rendering logic through automated testing is critical, especially as complexity grows. Unlike conventional unit tests that verify functional output in textual or numeric form, testing graphics output requires verification of pixel correctness.

One robust approach is **checksum-based unit testing** for rendered frames or intermediate buffers. This technique computes a compact hash or checksum of the pixel data and compares it against a known expected value. It enables automated regression detection while keeping test data manageable.

This section presents professional methodologies for implementing checksum-based unit tests for rendering code. It addresses common challenges, test design patterns, and integration within modern continuous integration environments, focusing on tools and practices from 2019 onward.

14.3.2 The Rationale for Checksum-Based Testing in Rendering

Pixel-perfect comparison between rendered images and golden references is a complex and resource-heavy process. Saving and comparing full image files for every test can be slow and cumbersome.

Checksum-based testing offers these advantages:

- **Compactness:** Only a fixed-size hash value (e.g., 32 or 64 bits) is stored per test case.
- **Speed:** Comparing checksums is faster than pixel-by-pixel comparisons.

- **Automation-Friendly:** Enables integration with standard test runners and CI pipelines.
- **Early Detection:** Quickly flags unexpected changes in rendering output.

The checksum acts as a fingerprint representing the entire rendered buffer.

14.3.3 Designing Rendering Unit Tests with Checksums

1. **Test Input and Setup**

   Each test must define:

   - **Rendering scenario:** Scene or primitive configuration, transformations, textures, lighting parameters, etc.
   - **Frame or buffer dimensions:** Resolution and pixel format.
   - **Deterministic state:** Fixed random seeds and constant inputs to ensure reproducible output.

   The rendering function is invoked with this setup, producing a framebuffer.

2. **Computing the Checksum**

   A suitable hash function should be:

   - **Fast:** To avoid slowing test runs.
   - **Deterministic:** Always produce the same hash for the same input.
   - **Collision Resistant:** Low chance of different images producing the same checksum.

   Common choices post-2019 include:

- **CRC32:** Widely used, fast, but moderate collision resistance.
- **FNV-1a:** Simple and fast non-cryptographic hash.
- **xxHash:** Very fast hashing function with low collision rates.
- **MD5 or SHA-1:** Cryptographic hashes used less frequently due to speed, but useful if collision risk must be minimal.

Example of computing CRC32 over pixel buffer:

```
uint32_t crc32(uint8_t* data, size_t length);

uint32_t compute_framebuffer_checksum(uint32_t* framebuffer, int
 ↪  pixel_count) {
    // Interpret pixel data as bytes
    return crc32((uint8_t*)framebuffer, pixel_count *
     ↪  sizeof(uint32_t));
}
```

3. **Storing and Validating Expected Checksums**

   Tests embed expected checksum values, either:

   - **Hardcoded constants:** For simple or stable rendering cases.
   - **External files:** For large test suites, checksums can be stored in test data files or JSON manifests.

   Test runner compares actual checksum to expected:

```
assert_equal(actual_checksum, expected_checksum);
```

   Failure indicates a rendering regression or bug.

14.3.4 Handling Variability in Rendering Output

Rendering output can be sensitive to:

- Minor floating-point differences
- Non-deterministic input (random seeds, timers)
- Platform-specific behavior (endianness, compiler optimizations)

To ensure stable checksums:

- Fix random seeds and disable time-dependent animations during tests.
- Use consistent compiler and floating-point settings.
- Avoid uninitialized buffers.
- Use fixed-point or deterministic math libraries if possible.
- Render at fixed resolution and pixel format.

14.3.5 Example Workflow of a Checksum-Based Unit Test

1. **Setup test scene:** Initialize objects with fixed parameters.
2. **Render to framebuffer:** Call rendering function.
3. **Compute checksum:** Hash the entire framebuffer.
4. **Compare to expected:** Assert equality or report failure.

Example (pseudo-C):

```
void test_draw_simple_triangle() {
    // Prepare framebuffer and scene
    uint32_t framebuffer[WIDTH * HEIGHT] = {0};
    setup_triangle_scene();

    render_scene(framebuffer, WIDTH, HEIGHT);

    uint32_t checksum = compute_framebuffer_checksum(framebuffer, WIDTH *
     ↪  HEIGHT);

    const uint32_t expected = 0xAABBCCDD; // Known checksum

    if (checksum != expected) {
        printf("Test failed: checksum mismatch 0x%08X ≠ 0x%08X\n",
         ↪  checksum, expected);
        exit(1);
    }
}
```

14.3.6 Integrating with Modern Testing Frameworks and CI Pipelines

After 2019, many C++ projects use frameworks such as **Google Test**, **Catch2**, or **Doctest** for unit testing. Integrating rendering tests is straightforward:

- Define rendering tests as regular test cases.
- On failure, output checksum and optionally save framebuffer dump for analysis.
- Run tests in headless mode (no GUI required).
- Automate execution in CI systems (GitHub Actions, Jenkins, GitLab CI).

Example with Google Test:

```
TEST(RenderingTests, SimpleTriangleChecksum) {
    Framebuffer fb(WIDTH, HEIGHT);
    Scene scene = SetupTriangleScene();

    Renderer renderer;
    renderer.Render(scene, fb);

    uint32_t checksum = ComputeCRC32(fb.Pixels(), fb.PixelCount());
    ASSERT_EQ(checksum, 0×AABBCCDD);
}
```

14.3.7 Managing Test Data and Re-Baselining

When intentional changes alter rendering output (e.g., bug fixes or feature additions), the expected checksums must be updated (”re-baselined”):

- Automate re-baselining with a command-line flag or environment variable.
- Dump failing framebuffers to files during test failure.
- Manually review changes before accepting new checksum.

This prevents false positives while maintaining test integrity.

14.3.8 Limitations and Advanced Techniques

- **False positives:** Small rendering differences (e.g., color rounding) cause checksum mismatch.
- **Localization:** Checksum does not identify where the error occurs.

- **Mitigations:**
  - Use checksum on smaller regions or sub-tests.
  - Supplement with image diff tools comparing pixel differences visually.
  - Implement fuzzy matching or tolerance thresholds in advanced test suites.

14.3.9 Summary

Checksum-based unit testing is an effective, scalable approach to verifying CPU-rendered graphics output. By hashing framebuffers, developers achieve rapid automated detection of visual regressions without managing bulky image data. Combined with deterministic scene setup and rigorous environment control, it forms a core part of modern graphics test strategies in CPU-only environments.

14.4 Performance Profiling Tools (perf, Valgrind, VTune)

14.4.1 Introduction

Performance profiling is a crucial step in developing efficient CPU-only graphics applications. Unlike GPU-based rendering where profiling often focuses on the GPU pipeline, CPU graphics programming demands thorough examination of CPU-bound bottlenecks, memory access patterns, cache usage, and algorithm efficiency.
This section provides an in-depth professional overview of the leading performance profiling tools available for CPU graphics programmers on modern platforms: **perf**, **Valgrind**, and **Intel VTune Profiler**. Each tool serves distinct but complementary roles in profiling and debugging, and their use is illustrated with practical advice and examples applicable to graphics programming scenarios from 2019 onward.

14.4.2 perf: Linux Profiling for CPU and Kernel

1. **Overview**

   perf is a powerful Linux performance analyzing tool integrated into the Linux kernel since version 2.6.31. It enables detailed profiling of CPU cycles, instructions, cache misses, context switches, and other events. Perf is particularly useful for low-level CPU profiling of graphics rendering code running on Linux systems.

2. **Key Features**

   - Sampling-based CPU profiling with call-graph support.
   - Hardware performance counters for cache misses, branch mispredictions.

- Profiling kernel and user-space code simultaneously.
- Support for various output formats including flame graphs.

3. **Usage in Graphics Code Profiling**

   To profile a CPU-only rendering loop, a typical workflow involves:

   - Running the graphics application under perf to gather CPU event samples.
   - Generating call-graphs to identify hot functions consuming most CPU cycles.
   - Inspecting cache miss statistics to find inefficient memory access patterns.

   Example command to profile CPU cycles during a graphics test:

   ```
   perf record -F 99 -g -- ./render_test_app
   perf report
   ```

   Here:

   - `-F 99` sets sampling frequency to 99 Hz.
   - `-g` enables call graph (stack trace) capture.
   - `perf report` opens an interactive summary showing where CPU time is spent.

4. **Interpreting Results for Graphics Optimization**

   - Identify functions related to bitmap manipulation, pixel blending, or math-heavy operations that dominate CPU time.
   - Analyze cache miss counters; excessive misses may indicate poor data locality in framebuffer access or pixel loops.
   - Use flame graphs generated from perf data to visualize call paths and prioritize optimization.

14.4.3 Valgrind: Memory Debugging and Profiling Suite

1. **Overview**

   Valgrind is a comprehensive instrumentation framework with tools for memory debugging, leak detection, and profiling. While not a real-time profiler, its tool **Callgrind** is highly effective for detailed CPU profiling, and **Massif** for heap profiling.

   Though Valgrind incurs significant overhead, it provides invaluable insights into code behavior, especially regarding memory access patterns and performance hotspots in graphics applications.

2. **Key Tools for Graphics Profiling**

   - **Callgrind:** Collects detailed call counts, instruction counts, cache misses.
   - **Massif:** Analyzes heap memory usage over time.
   - **Memcheck:** Detects invalid memory reads/writes, useful for preventing subtle rendering bugs.

3. **Using Callgrind for CPU Rendering Code**

   Run the application under Callgrind to gather instruction-level profiling data:

   ```
   valgrind --tool=callgrind ./render_test_app
   callgrind_annotate callgrind.out.<pid>
   ```

   Callgrind provides:

   - Number of instructions executed per function.
   - Call graph showing calling relationships and instruction counts.

- Cache simulation statistics to locate cache-unfriendly code.

This helps graphics developers find inefficiencies in pixel processing loops or math-heavy transformations.

4. **Visualization with KCachegrind**

   To better interpret results, export Callgrind data to visualization tools like **KCachegrind** (Linux) or **QCachegrind** (cross-platform). These tools present interactive graphs and heatmaps of execution hotspots, aiding pinpointing CPU time sinks in rendering functions.

14.4.4 Intel VTune Profiler: Advanced CPU and Memory Analysis

1. **Overview**

   Intel VTune Profiler is a commercial-grade performance analyzer with deep integration for Intel CPUs and modern platforms, including Windows and Linux. Since 2019, VTune supports extensive features for CPU microarchitecture profiling, memory hierarchy analysis, and threading behavior, making it invaluable for high-performance graphics code optimization.

2. **Key Features**

   - Sampling and event-based profiling at CPU instruction level.
   - Microarchitecture-specific analysis: pipeline stalls, branch mispredictions, vectorization efficiency.
   - Memory access analysis: cache hit/miss rates, DRAM latency.
   - Threading analysis: synchronization overhead, thread contention.

- Support for GPU offload analysis (though out of scope here).

3. **Profiling CPU-Only Graphics Applications**

   VTune's GUI and command-line tools enable detailed profiling of rendering code sections:

   - Identify bottlenecks caused by floating-point math in transformations or pixel processing.
   - Analyze vectorization efficiency when using SIMD instructions in pixel loops.
   - Investigate memory bandwidth utilization and cache utilization to reduce stalls.

   Example workflow:

   (a) Launch VTune and create a new project.

   (b) Run the CPU profiling analysis on your application.

   (c) Focus on ”Top Hotspots” to find functions with highest CPU usage.

   (d) Use Memory Access analysis to optimize framebuffer access patterns.

   (e) Evaluate threading efficiency if your renderer uses parallelism.

4. **Leveraging VTune’s Advanced Insights**

   - Use VTune to verify whether your manually optimized assembly or SIMD code is effectively reducing pipeline stalls.
   - Detect false sharing or contention issues in multithreaded rendering loops.
   - Explore vectorization reports to improve SIMD utilization for per-pixel operations.

14.4.5 Choosing the Right Tool for Your Needs

Comparison of Performance Profiling Tools

Tool	Platform	Use Case	Overhead	Level of Detail	Cost
perf	Linux	Low-overhead CPU profiling	Low	CPU cycles, cache	Free (open-source)
Valgrind	Linux, macOS	Memory debugging, detailed profiling	High	Instruction count, cache sim	Free (open-source)
VTune	Windows, Linux	Advanced CPU/ memory/ thread profiling	Medium	Micro architecture, memory	Commercial (free trial available)

For lightweight profiling during development, **perf** is ideal on Linux systems. For deep inspection of memory and instruction behavior, **Valgrind**'s Callgrind is invaluable. When working on Intel architectures and needing sophisticated insight into CPU internals and threading, **VTune Profiler** offers unparalleled depth.

14.4.6 Practical Tips for Effective Profiling

- **Isolate rendering sections:** Profile smaller units of rendering code to get focused data.

- **Use representative workloads:** Profile with typical frame data and scene complexity.
- **Combine tools:** Use perf or VTune for CPU hotspots and Valgrind for memory issues.
- **Measure before and after optimizations:** Quantify the impact of code changes.
- **Profile on target hardware:** Results vary widely with CPU architecture and OS.

14.4.7 Summary

Mastering performance profiling tools is essential for optimizing CPU-only graphics applications. The combined use of **perf** for general Linux profiling, **Valgrind** for detailed memory and instruction analysis, and **Intel VTune Profiler** for advanced microarchitectural insights empowers developers to identify and resolve performance bottlenecks efficiently.

Effective profiling drives better use of CPU resources, improved frame rates, and responsive graphics rendering without reliance on GPU acceleration.

Appendices

Appendix A: Reference Tables (Pixel Formats, VGA Modes)

A.1 Introduction

This appendix provides essential reference tables that serve as foundational knowledge for CPU-only graphics programming. Understanding pixel formats and historical VGA video modes is critical for manipulating raw framebuffer data, designing software rasterizers, and ensuring compatibility with various display environments.
The information herein reflects the state-of-the-art and practical usage patterns up to and beyond 2019, including modern adaptations relevant to contemporary operating systems and third-party tools that support these formats.

A.2 Pixel Formats

Pixel formats define how color and sometimes alpha (transparency) information is stored in memory for each pixel in a framebuffer or image buffer. Proper handling of these formats is crucial for rendering correctness and performance optimization.

A.2.1 Common Pixel Format Categories

Pixel Format Categories and Their Characteristics

Category	Description	Typical Use Cases
Indexed Color	Pixels store indices into a palette (color lookup table).	Legacy graphics, GIFs, low-memory situations.
RGB Formats	Pixels store red, green, blue components explicitly.	Most modern graphics, true color displays.
RGBA / ARGB Formats	RGB components plus alpha channel for transparency.	Compositing, UI rendering, transparency effects.
Grayscale	Single luminance value per pixel.	Monochrome images, masks, some medical imaging.
YUV / YCbCr	Luminance and chrominance separated, often for video.	Video processing, camera input, hardware acceleration.

A.2.2 Detailed Pixel Format Table

Pixel Format Reference Table

Format Name	**Bits per Pixel**	**Channel Order & Bit Allocation**	**Description & Notes**	**Common Abbreviation**	**Alignment / Padding Notes**
RGB565	16	5 bits R, 6 bits G, 5 bits B	Compact, no alpha, widely used in embedded	RGB565	Packed 16-bit, no padding
RGB888	24	8 bits R, 8 bits G, 8 bits B	True color, no alpha	RGB24	Usually packed tightly, 3 bytes/pixel
RGBA8888	32	8 bits R, G, B, A	True color + alpha, common in modern APIs	RGBA32	4 bytes/pixel aligned
ARGB8888	32	8 bits A, R, G, B	Alpha first variant, used in Windows GDI	ARGB32	4 bytes/pixel aligned

Format Name	Bits per Pixel	Channel Order & Bit Allocation	Description & Notes	Common Abbreviatio	Alignment / Padding Notes
BGRA8888	32	8 bits B, G, R, A	Windows Direct2D, Direct3D, some UIs	BGRA32	4 bytes/pixel aligned
Indexed8	8	8-bit palette index	Paletted mode, legacy	8-bit Indexed	Palette of up to 256 colors
Grayscale8	8	8-bit luminance	Monochrome images	Gray8	1 byte/pixel
YUY2 (YUYV)	16 (per 2 pixels)	Y0 U Y1 V (4:2:2 chroma subsampling)	Video formats, some webcams	YUY2	Packed, 2 pixels share chroma
NV12	12 (avg. per pixel)	Y plane + interleaved UV plane	Video processing, hardware-accelerated decode	NV12	Semi-planar, often used in hardware

A.2.3 Notes on Pixel Formats

- **Endianness:** Some formats like ARGB or BGRA may vary in memory layout depending on CPU endianness and compiler packing rules.

- **Stride and Padding:** Framebuffers often include padding bytes at the end of each row (stride) to align to 4 or 8 bytes for performance reasons.
- **Alpha Premultiplication:** Alpha channels may be premultiplied or not, affecting blending computations.
- **Hardware Support:** Certain formats like NV12 or YUY2 are common in hardware video pipelines but may require conversion for CPU-only rendering.

A.3 VGA Video Modes

The Video Graphics Array (VGA) standard established several well-known video modes foundational to PC graphics. Understanding these modes aids in working with legacy software, low-level programming, and emulation.

A.3.1 Historical Context Introduced in 1987 by IBM, VGA modes became a de facto standard for PC graphics. Modern operating systems rarely use VGA modes directly for display output but support them for compatibility, boot splash screens, and low-level graphics programming.

A.3.2 Common VGA Modes Reference Table

Legacy VGA/EGA Video Modes

Mode Number	Resolution (pixels)	Color Depth (bits)	Number of Colors	Memory Layout	Description
Mode 0x03	640 × 480	4-bit (planar)	16	Planar, 64KB VGA memory	Standard VGA 16-color mode
Mode 0x12	640 × 480	8-bit	256	Linear frame buffer	256-color mode, standard VGA
Mode 0x13	320 × 200	8-bit	256	Linear frame buffer	Popular 256-color gaming mode
Mode 0x0D	320 × 200	4-bit planar	16	Planar	CGA-compatible 16 colors
Mode 0x10	640 × 350	4-bit planar	16	Planar	EGA-compatible 16 colors

A.3.3 VGA Planar vs Linear Modes

- **Planar Modes:**
  Use multiple memory planes where each plane contains one bit per pixel across the screen. Requires complex bit masking and writing multiple planes to set pixel colors. Efficient in older hardware but complex for software rasterizers.

- **Linear Modes:**
  Use a contiguous block of memory where each byte or word represents a pixel color index or RGB value. Easier to program and favored for modern software rendering.

A.3.4 Modern Usage and Emulation

- Modern OSes rarely expose VGA modes directly; they use high-resolution framebuffers with standard pixel formats.
- VGA modes are often emulated in virtual machines and DOS emulators.
- Understanding VGA modes assists in writing software that interacts with BIOS or firmware during system boot or in minimalist graphics environments.

A.4 Practical Examples and Tools

- **SDL2 Library:**
  Supports multiple pixel formats, enabling programmers to choose optimal formats for software rendering buffers.
- **Linux Framebuffer Devices (/dev/fb0):**
  Provide raw access to framebuffer memory, often requiring knowledge of pixel formats and stride.
- **Windows GDI:**
  Exposes pixel formats like 32-bit ARGB and 24-bit RGB for bitmap operations.
- **Third-Party Tools:**
  Libraries like **stb_image** and **FreeImage** help decode various pixel formats and simplify CPU-side image loading and manipulation.

A.5 Summary

This appendix equips the graphics programmer with detailed tables and explanations of common pixel formats and VGA video modes. Mastery of these references facilitates efficient pixel manipulation, compatibility with legacy modes, and lays the groundwork for advanced CPU-only graphics techniques across modern platforms.

Appendix B: Using C++ with Software Graphics (RAII, Smart Pointers for Surfaces, etc.)

B.1 Introduction

In CPU-only graphics programming, effective resource management is essential to ensure robustness, prevent memory leaks, and maintain clean, maintainable code. C++ offers powerful idioms and language features such as RAII (Resource Acquisition Is Initialization) and smart pointers that greatly simplify the management of graphics resources like surfaces, framebuffers, and pixel buffers.
This appendix explores best practices and modern C++ techniques for managing graphics resources in software rendering contexts, focusing on the period after 2019 where C++17 and C++20 standards have introduced additional tools and idioms. Practical examples and design patterns are provided to demonstrate safe, efficient handling of surfaces and related data structures.

B.2 Resource Management Challenges in Software Graphics

CPU-only graphics applications commonly involve dynamic allocation of:

- Framebuffers or pixel surfaces
- Temporary buffers for image transformations or filters
- Palette or color lookup tables
- Bitmap or texture data loaded from external sources

Managing these resources manually with raw pointers and explicit `new/delete` calls leads to complex error-prone code, including memory leaks, dangling pointers, and double frees. Graphics buffers may be large and frequently reallocated or shared across threads or modules, increasing the risk of concurrency bugs or performance regressions. Modern C++ provides abstractions to encapsulate resource lifetime, ownership semantics, and thread safety.

B.3 RAII in Graphics Programming

B.3.1 RAII Principle RAII is a core C++ idiom where resource acquisition (e.g., memory allocation, file handles) is tied to object lifetime. When an RAII object is created, it acquires the resource; when it goes out of scope or is destroyed, it automatically releases the resource.

For software graphics:

- A surface class acquires a pixel buffer in its constructor.
- The destructor frees the buffer automatically.
- This avoids explicit cleanup calls and reduces leaks.

```
class Surface {
private:
    uint8_t* pixels_;
    size_t width_;
    size_t height_;
    size_t stride_;  // Bytes per row

public:
    Surface(size_t width, size_t height)
        : width_(width), height_(height), stride_(width * 4) // Assuming
        ↪  RGBA8888
    {
        pixels_ = new uint8_t[stride_ * height_];
    }

    ~Surface() {
        delete[] pixels_;
    }

    // Deleted copy operations to avoid shallow copies
    Surface(const Surface&) = delete;
    Surface& operator=(const Surface&) = delete;

    // Move operations for efficient transfer of ownership
    Surface(Surface&& other) noexcept
        : pixels_(other.pixels_), width_(other.width_),
         ↪  height_(other.height_), stride_(other.stride_)
    {
        other.pixels_ = nullptr;
        other.width_ = 0;
        other.height_ = 0;
        other.stride_ = 0;
    }

    Surface& operator=(Surface&& other) noexcept {
        if (this != &other) {
            delete[] pixels_;
            pixels_ = other.pixels_;
            width_ = other.width_;
            height_ = other.height_;
            stride_ = other.stride_;
            other.pixels_ = nullptr;
            other.width_ = 0;
            other.height_ = 0;
            other.stride_ = 0;
        }
        return *this;
    }

    uint8_t* data() { return pixels_; }
    size_t width() const { return width_; }
    size_t height() const { return height_; }
    size_t stride() const { return stride_; }
};
```

B.3.2 Example: Basic Surface Class with RAII

This pattern encapsulates resource allocation and cleanup, prevents common bugs, and supports move semantics for efficient passing.

B.4 Smart Pointers for Graphics Surfaces

B.4.1 Motivation Complex graphics applications often share surfaces or buffers among multiple modules (e.g., rendering engine, UI, caching system). Sharing raw pointers complicates ownership and lifecycle management.

Smart pointers provide automated, well-defined ownership semantics:

- `std::unique_ptr` for exclusive ownership.
- `std::shared_ptr` for shared ownership with reference counting.

```

using SurfacePtr = std::unique_ptr<Surface>;

SurfacePtr CreateSurface(size_t width, size_t height) {
```

```cpp
    return std::make_unique<Surface>(width, height);
}

void ProcessSurface(SurfacePtr& surface) {
    // Use surface->data(), surface->width(), etc.
}
```

B.4.2 Using std::unique_ptr for Exclusive Ownership

This pattern ensures the surface is freed automatically when the unique pointer goes out of scope.

B.4.3 Using std::shared_ptr for Shared Ownership In scenarios where multiple components need access to the same surface:

```cpp
using SharedSurface = std::shared_ptr<Surface>;

SharedSurface LoadTexture(const std::string& filename) {
    SharedSurface surface = std::make_shared<Surface>(1024, 1024);
    // Load pixel data into surface
    return surface;
}

void Render(SharedSurface surface) {
    // Read surface pixels for rendering
}
```

Shared pointers automatically manage the reference count, deleting the surface when the last reference is destroyed.

B.5 Custom Deleters for Specialized Cleanup

For some graphics resources, cleanup involves more than a simple `delete[]`. For example, buffers allocated via platform-specific APIs or third-party libraries may require explicit deallocation functions.
Smart pointers support custom deleters:

```
void FreeCustomBuffer(uint8_t* ptr) {
    // Call platform-specific free function
}

using CustomSurfacePtr = std::unique_ptr<uint8_t[],
↪  decltype(&FreeCustomBuffer)>;

CustomSurfacePtr CreateCustomBuffer(size_t size) {
    uint8_t* buf = PlatformAllocate(size);
    return CustomSurfacePtr(buf, &FreeCustomBuffer);
}
```

This technique integrates external resource management with RAII patterns.

B.6 Integration with Modern C++ Features

B.6.1 constexpr and noexcept Defining constexpr constructors and noexcept destructors in surface classes enables compiler optimizations and safer exception handling:

```
Surface(size_t width, size_t height) noexcept
```

B.6.2 std::optional and std::variant These types help manage optional graphics buffers or multiple surface formats, reducing error-prone raw pointer usage.

B.7 Thread Safety and Concurrency Considerations

When sharing surfaces across threads (common in rendering pipelines), synchronization must be managed carefully.

- Use `std::shared_ptr` combined with immutable data patterns or explicit locking.
- Avoid sharing raw pointers without synchronization.
- Modern C++ atomic smart pointers and concurrent containers (available in C++20) can help manage these safely.

B.8 Practical Examples with Third-Party Libraries

- **SDL2:** Uses RAII-style `SDL_Surface` wrappers and encourages `std::unique_ptr` or custom wrappers for resource management.
- **stb_image:** C library for image loading; combine with C++ smart pointers to manage loaded pixel buffers safely.
- **Boost.GIL:** Modern generic image library for C++ supporting RAII and smart pointer usage patterns for images and views.

B.9 Summary

Efficient and safe management of graphics resources in CPU-only rendering is achievable through modern C++ idioms:

- RAII ensures deterministic resource release and reduces leaks.
- Smart pointers provide flexible ownership models suitable for complex applications.

- Custom deleters and integration with platform APIs enhance flexibility.
- Awareness of threading and concurrency is vital in multi-threaded renderers.

Adopting these practices leads to cleaner, safer, and more maintainable CPU graphics codebases that stand the test of time and evolving hardware environments.

Appendix C: Minimal Software Renderer Template (Cross-Platform C++)

C.1 Introduction

A minimal software renderer provides a fundamental starting point for understanding CPU-based graphics rendering without GPU acceleration. It illustrates core principles such as pixel buffer management, primitive rasterization, and platform-independent display. This appendix presents a clean, modular, and cross-platform C++ template for a software renderer suitable for educational purposes, prototyping, or as a foundation for more complex CPU-only graphics projects.

This template is designed to compile and run on Windows and Linux environments, leveraging modern C++ features (C++17 and later) and basic platform APIs for window creation and pixel display. No GPU-specific APIs or hardware acceleration techniques are used.

C.2 Design Goals

- **Cross-platform compatibility:** Core rendering logic is platform-agnostic.
- **Simplicity and clarity:** Minimal dependencies, straightforward API.
- **Extensibility:** Modular design for adding features like clipping, shading, or texture mapping.
- **Performance awareness:** Efficient memory handling and pixel manipulation within CPU constraints.
- **Modern C++:** Use of RAII, smart pointers, and constexpr where applicable.

C.3 Core Components Overview

Core Components of a Software Renderer

Component	Responsibility
FrameBuffer	Pixel buffer management and basic drawing primitives.
Renderer	High-level API for drawing primitives (lines, triangles).
Window/Display	Platform-specific window creation and buffer presentation.
Main Loop	Event handling and rendering loop management.

C.4 FrameBuffer Class

Manages a contiguous pixel buffer in memory, representing the framebuffer to which all rendering commands write.

```cpp
#include <vector>
#include <cstdint>
#include <cstring> // for memset

class FrameBuffer {
public:
    const int width;
    const int height;
    const int bytesPerPixel = 4; // RGBA8888 format
private:
    std::vector<uint8_t> pixels_;

public:
    FrameBuffer(int w, int h)
        : width(w), height(h), pixels_(w * h * bytesPerPixel, 0) {}
    // Clear the buffer with a given color (RGBA)
    void clear(uint8_t r, uint8_t g, uint8_t b, uint8_t a = 255) {
        for (int i = 0; i < width * height; ++i) {
            pixels_[i * 4 + 0] = r;
            pixels_[i * 4 + 1] = g;
            pixels_[i * 4 + 2] = b;
            pixels_[i * 4 + 3] = a;
        }
    }

    // Set a pixel color at (x, y)
    void setPixel(int x, int y, uint8_t r, uint8_t g, uint8_t b, uint8_t
     ↪  a = 255) {
        if (x < 0  x >= width  y < 0  y >= height)
            return; // bounds check

        size_t index = (y * width + x) * bytesPerPixel;
        pixels_[index + 0] = r;
        pixels_[index + 1] = g;
        pixels_[index + 2] = b;
        pixels_[index + 3] = a;
    }

    // Get pointer to raw pixel data (for platform display or saving)
    const uint8_t* data() const { return pixels_.data(); }
    uint8_t* data() { return pixels_.data(); }
};
```

C.5 Renderer Class

Implements basic drawing primitives using the framebuffer. This example covers line drawing via Bresenham's algorithm and triangle rasterization via barycentric coordinates.

```
#include <algorithm>

class Renderer {
private:
    FrameBuffer& fb_;

    void drawLineLow(int x0, int y0, int x1, int y1, uint8_t r, uint8_t
    ↪  g, uint8_t b) {
        int dx = x1 - x0;
        int dy = y1 - y0;
        int yi = 1;
        if (dy < 0) {
            yi = -1;
            dy = -dy;
        }
        int D = 2 * dy - dx;
        int y = y0;

        for (int x = x0; x <= x1; x++) {
            fb_.setPixel(x, y, r, g, b);
            if (D > 0) {
                y = y + yi;
                D = D - 2 * dx;
            }
            D = D + 2 * dy;
        }
    }
    void drawLineHigh(int x0, int y0, int x1, int y1, uint8_t r, uint8_t
    ↪   g, uint8_t b) {
        int dx = x1 - x0;
        int dy = y1 - y0;
        int xi = 1;
        if (dx < 0) {
            xi = -1;
            dx = -dx;
        }
        int D = 2 * dx - dy;
        int x = x0;

        for (int y = y0; y <= y1; y++) {
            fb_.setPixel(x, y, r, g, b);
            if (D > 0) {
                x = x + xi;
                D = D - 2 * dy;
            }
            D = D + 2 * dx;
        }
    }

public:
    Renderer(FrameBuffer& framebuffer) : fb_(framebuffer) {}

    void drawLine(int x0, int y0, int x1, int y1, uint8_t r, uint8_t g,
    ↪   uint8_t b) {
        if (std::abs(y1 - y0) < std::abs(x1 - x0)) {
            if (x0 > x1)
                drawLineLow(x1, y1, x0, y0, r, g, b);
            else
                drawLineLow(x0, y0, x1, y1, r, g, b);
        } else {
            if (y0 > y1)
                drawLineHigh(x1, y1, x0, y0, r, g, b);
            else
                drawLineHigh(x0, y0, x1, y1, r, g, b);
        }
    }

    // Triangle rasterization using barycentric coordinates
    void drawTriangle(int x0, int y0, int x1, int y1, int x2, int y2,
    ↪   uint8_t r, uint8_t g, uint8_t b) {
        // Bounding box
        int minX = std::min({x0, x1, x2});
        int maxX = std::max({x0, x1, x2});
        int minY = std::min({y0, y1, y2});
        int maxY = std::max({y0, y1, y2});

        auto edgeFunction = [](int x0, int y0, int x1, int y1, int x, int
        ↪   y) -> int {
            return (y0 - y1)*x + (x1 - x0)*y + x0*y1 - x1*y0;
        };

        for (int y = minY; y <= maxY; ++y) {
            for (int x = minX; x <= maxX; ++x) {
                int w0 = edgeFunction(x1, y1, x2, y2, x, y);
                int w1 = edgeFunction(x2, y2, x0, y0, x, y);
                int w2 = edgeFunction(x0, y0, x1, y1, x, y);

                if (w0 >= 0 && w1 >= 0 && w2 >= 0) {
                    fb_.setPixel(x, y, r, g, b);
                }
            }
        }
    }
};
```

C.6 Platform-Specific Window and Display

C.6.1 Windows (Win32 API) Use the Win32 API to create a window and display the framebuffer via `StretchDIBits` or `UpdateLayeredWindow`:

- Create a `BITMAPINFO` structure describing the framebuffer format (e.g., 32-bit RGBA).
- In the window's paint handler, call `StretchDIBits` with the framebuffer pointer.
- Manage message loop and WM_PAINT events.

C.6.2 Linux (Xlib) Use Xlib to create a simple window and display the framebuffer via `XPutImage`:

- Create an `XImage` structure referencing the framebuffer data.
- On expose events, call `XPutImage` to copy pixel data to the window.
- Manage the event loop with Xlib calls.

C.7 Example: Main Application Flow

```cpp
int main() {
    constexpr int width = 800;
    constexpr int height = 600;

    FrameBuffer framebuffer(width, height);
    Renderer renderer(framebuffer);

    // Clear background to dark gray
    framebuffer.clear(50, 50, 50);

    // Draw white triangle
    renderer.drawTriangle(100, 100, 400, 500, 700, 150, 255, 255, 255);

    // Draw red lines around the triangle
    renderer.drawLine(100, 100, 400, 500, 255, 0, 0);
    renderer.drawLine(400, 500, 700, 150, 255, 0, 0);
    renderer.drawLine(700, 150, 100, 100, 255, 0, 0);

    // Platform-specific code to create window and show framebuffer
    // [Windows or Xlib implementation here, calling framebuffer.data()]

    // Enter event loop to keep window responsive and repaint as needed

    return 0;
}
```

C.8 Extending the Template

- Add support for color blending and alpha compositing.

- Implement clipping to handle primitives partially outside the viewport.

- Integrate texture mapping by loading images into separate buffers.
- Add double-buffering for flicker-free animation.
- Implement a simple GUI or input handling for interactivity.

C.9 Summary

This minimal software renderer template demonstrates the essential elements of CPU-based graphics programming in a clear, portable C++ style. It provides a solid foundation for exploring more advanced CPU rendering techniques without relying on GPU hardware or complex APIs. This approach remains valuable for embedded systems, academic exploration, and graphics algorithm research.

Appendix D: ASM - GAS Assembler Cheat Sheet for Intel Syntax

D.1 Introduction

This appendix provides a concise yet comprehensive cheat sheet for using the GNU Assembler (GAS) with Intel syntax, focusing on the needs of graphics programmers working with CPU-only rendering. The GAS assembler is widely used on Linux and cross-platform toolchains such as MinGW on Windows. Although GAS traditionally defaults to AT&T syntax, Intel syntax is often preferred for readability and familiarity by developers accustomed to Intel-style assembly language, especially those transitioning from MSVC or NASM.
The cheat sheet covers essential directives, instructions, registers, addressing modes, and conventions necessary for writing, assembling, and linking assembly code to C/C++ programs in modern environments (post-2019).

D.2 Setting Intel Syntax in GAS

GAS supports Intel syntax via the `.intel_syntax` directive.

```
.intel_syntax noprefix
```

- `noprefix` means registers and instructions do not need `%` prefixes.
- Use `.att_syntax` to revert back to AT&T style if needed.
- Intel syntax improves clarity for programmers familiar with MSVC-style assembly.

D.3 Registers Overview

x86-64 General Purpose Registers

Register Name	Size	Description
`rax`	64-bit	Accumulator (general purpose)
`eax`	32-bit	Lower 32 bits of `rax`
`ax`	16-bit	Lower 16 bits of `rax`
`al`	8-bit	Lower 8 bits of `ax`
`rbx`	64-bit	Base register
`rcx`	64-bit	Counter register
`rdx`	64-bit	Data register
`rsi`	64-bit	Source index
`rdi`	64-bit	Destination index
`rsp`	64-bit	Stack pointer
`rbp`	64-bit	Base pointer (frame pointer)
`r8` to `r15`	64-bit	Additional general registers

D.4 Common GAS Directives (Intel Syntax)

Common Assembly Directives and Their Usage

Directive	Description
`.section .text`	Start of code section

Directive	Description
`.global symbol`	Make symbol visible outside this file
`.intel_syntax noprefix`	Use Intel syntax
`.att_syntax`	Use AT&T syntax
`.data`	Start of data section
`.bss`	Uninitialized data section
`.byte`, `.word`, `.long`, `.quad`	Define 8-, 16-, 32-, 64-bit data values
`.asciz`	Null-terminated string
`.align n`	Align next data/code to 2^n bytes boundary
`.extern symbol`	Declare external symbol
`.type symbol, @function`	Define symbol as function
`.size symbol, size`	Define size of symbol

D.5 Basic Instruction Syntax

General format:

```
instruction destination, source
```

Examples:

- Move:

```
mov rax, rbx
mov eax, [rbp-4]     ; Load from memory
mov [rsp+8], rcx     ; Store to memory
```

- Arithmetic:

```
add eax, 10
sub rbx, rax
imul rcx, rdx        ; Multiply rcx by rdx
```

- Logical:

```
and eax, 0xFF
or rdx, rax
xor r8, r8           ; Clear r8
```

- Control flow:

```
jmp label
je equal_label       ; Jump if equal (ZF=1)
jne not_equal_label ; Jump if not equal (ZF=0)
call function_name
ret
```

D.6 Memory Addressing Modes

Intel syntax supports several addressing modes for memory operands:

x86-64 Addressing Syntax Examples

Syntax Example	Meaning
`[rax]`	Memory at address in `rax`
`[rbp - 4]`	Memory at `rbp` minus 4 bytes
`[rsi + 8]`	Memory at `rsi + 8`
`[rax + rbx * 4]`	Base `rax` + index `rbx` scaled by 4
`[rcx + rdx * 2 + 16]`	Base + index × scale + displacement

Examples:

```
mov eax, [rbp-4]        ; Load 32-bit int from stack frame
mov byte ptr [rdi], 0 ; Store 0 to memory pointed by rdi
```

D.7 Stack and Calling Conventions

D.7.1 Windows x64 Calling Convention

- First 4 integer arguments in `rcx`, `rdx`, `r8`, `r9`
- Additional args pushed on the stack
- Caller reserves 32 bytes shadow space
- Return value in `rax`

D.7.2 System V AMD64 (Linux) Calling Convention

- First 6 integer args in `rdi`, `rsi`, `rdx`, `rcx`, `r8`, `r9`
- Stack aligned to 16 bytes before call
- Return value in `rax`

D.8 Example: Simple Function (Intel Syntax GAS)

```
.intel_syntax noprefix
.global add_numbers
.type add_numbers, @function
add_numbers:
    mov rax, rcx        ; Move first argument to rax
    add rax, rdx        ; Add second argument
    ret
```

- This function corresponds to:
  `uint64_t add_numbers(uint64_t a, uint64_t b) { return a + b; }`

D.9 Macros and Constants

- Define constants with `.equ` or `.set`:

```
.equ BUFFER_SIZE, 1024
```

- Macros can be defined using `.macro` and `.endm`:

```
.macro CLEAR_REG reg
    xor \reg, \reg
.endm

CLEAR_REG rax
```

D.10 Conditional Execution and Flags

Common condition codes:

x86-64 Conditional Jump Instructions

Instruction	Meaning	Flag Tested
`je` / `jz`	Jump if equal / zero	Zero Flag (`ZF`)
`jne` / `jnz`	Jump if not equal / not zero	Zero Flag (`ZF`)
`jg`	Jump if greater (signed)	`SF = OF` and `ZF = 0`
`jl`	Jump if less (signed)	`SF ≠ OF`
`ja`	Jump if above (unsigned)	`CF = 0` and `ZF = 0`
`jb`	Jump if below (unsigned)	`CF = 1`

D.11 Inline Assembly Integration with C++

When integrating GAS Intel syntax assembly in C++ code (GCC or Clang):

- Use extended asm with syntax directive if necessary.
- Example snippet (GCC):

```c
int add_numbers(int a, int b) {
    int result;
    asm volatile(
        ".intel_syntax noprefix;"
        "mov eax, %1;"
        "add eax, %2;"
        "mov %0, eax;"
        ".att_syntax prefix;"
        : "=r"(result)
        : "r"(a), "r"(b)
    );
    return result;
}
```

D.12 Tips for Using GAS with Intel Syntax in Graphics Programming

- Use `.intel_syntax noprefix` at the start of each `.asm` file or inline asm block for readability.
- Be mindful of platform-specific calling conventions when writing assembly functions callable from C/C++.
- For pixel or vector operations, leverage SIMD extensions (e.g., SSE, AVX) but be aware these require additional directives and registers.
- Use `.align 16` or appropriate alignment directives for data structures that benefit from alignment.
- For Windows, MinGW provides GCC/GAS toolchain supporting Intel syntax; on Linux, `gcc` and `clang` support inline and external asm with GAS syntax.

D.13 Summary

This cheat sheet presents a compact reference to write and understand assembly using the GNU Assembler in Intel syntax, tailored for modern graphics programming using CPU-only rendering techniques. Mastery of these elements enables performance-critical code segments, fine-grained control over CPU instructions, and integration of assembly routines into C++ graphics applications.

Appendix E: Further Reading & Resources

E.1 Introduction

This appendix provides a curated set of further reading materials and resources designed to deepen the reader's understanding of CPU-based graphics programming. It includes recommended books, academic papers, technical documentation, and practical tools that complement the material covered in this book. The focus is on modern, actively maintained resources that reflect the latest developments in CPU architectures, programming languages, compiler technologies, and third-party libraries used in CPU graphics programming since 2019.

E.2 Foundational Books and Texts

E.2.1 Core Graphics Programming

- **"Computer Graphics: Principles and Practice"** (3rd Edition, 2019 Update)
  This definitive text remains essential for understanding fundamental graphics concepts and algorithms. Although it covers GPU programming, its in-depth explanation of rasterization, shading models, and geometric transformations remains vital for CPU-based graphics.

- **"Real-Time Rendering"** (4th Edition, 2018)
  While primarily GPU-focused, it contains foundational principles applicable to CPU rasterization and rendering pipelines. It covers optimization techniques and algorithmic improvements relevant when GPU acceleration is unavailable.

E.2.2 CPU Architecture and Optimization

- **"Computer Systems: A Programmer's Perspective"** (3rd Edition, 2019)

Provides a thorough understanding of how modern CPUs execute code, cache hierarchies, and memory management—all critical knowledge for optimizing CPU graphics programs.

- **”Optimizing Software in C++”** (2nd Edition, 2020)
  Focuses on techniques to write high-performance C++ code, including cache-friendly data structures, instruction-level parallelism, and vectorization—all vital when writing CPU-only graphics code.

E.3 Modern Tools and Libraries for CPU Graphics Programming

E.3.1 Compiler Toolchains and Debuggers

- **LLVM/Clang** (Post-2019 Versions)
  Continues to be a leading open-source compiler suite supporting advanced optimization, profile-guided optimization (PGO), and vectorization. Its modular design allows integration with custom CPU graphics pipelines.

- **Microsoft Visual Studio 2022**
  Supports modern C++20/23 standards, with improved diagnostics and performance analysis tools, essential for Windows-based CPU graphics development.

- **Intel oneAPI Toolkits**
  Include compilers, libraries, and analysis tools optimized for Intel CPUs and vector instruction sets, such as AVX-512, which are crucial for SIMD acceleration in CPU graphics tasks.

E.3.2 Profiling and Performance Analysis

- **Intel VTune Profiler (Post-2019 Releases)**
  Offers detailed CPU and cache usage analysis, hotspots detection, and memory bottleneck identification, enabling fine-tuning of CPU graphics rendering code.

- **Linux perf Tool and BPF-based Tracing**
  Enables low-overhead performance monitoring on Linux systems, useful for profiling CPU-bound rendering loops.

E.3.3 Third-Party CPU Graphics Libraries

- **AGG (Anti-Grain Geometry) Library**
  A high-quality, CPU-only vector graphics rendering library with active forks and improvements continuing after 2019. It offers sub-pixel accuracy and efficient rasterization algorithms.

- **NanoVG**
  A small CPU-based vector graphics library inspired by OpenGL but also supporting CPU rendering backends. Frequently updated to support modern C++ standards and multi-threaded usage.

- **Skia Graphics Library**
  While Skia supports GPU acceleration, it includes robust CPU rasterization backends that are actively maintained and optimized, widely used in various cross-platform applications.

E.4 Academic and Industry Papers

E.4.1 Recent Advances in CPU Rasterization

- Papers on SIMD-accelerated rasterization and tile-based rendering on CPUs have emerged since 2019, presenting techniques to maximize CPU utilization while minimizing memory bandwidth usage.

- Research into software ray tracing optimized for CPU architectures highlights approaches that leverage modern SIMD instruction sets and cache optimizations.

E.4.2 Multithreading and Parallelism in CPU Graphics

- Studies on thread pool designs, lock-free data structures, and work-stealing schedulers applied to CPU graphics workloads provide insight into maximizing multi-core CPU utilization.

- Publications exploring fine-grained task parallelism in rasterizers demonstrate how to scale CPU rendering across many cores without synchronization bottlenecks.

E.5 Online Documentation and Community Resources

E.5.1 Official Documentation

- **Processor Manufacturer Manuals**
  Up-to-date Intel and AMD architecture manuals detail instruction sets (e.g., AVX2, AVX-512) and CPU features essential for advanced graphics programming.

- **Operating System Developer Guides**
  Windows SDK documentation and Linux kernel documentation provide insights into system calls, threading APIs, and memory management essential for low-level CPU graphics programming.

E.5.2 Community Forums and Discussion Groups

- Professional forums focused on C++ and systems programming (such as specialized mailing lists, Stack Overflow, and dedicated Discord servers) remain valuable for troubleshooting and advanced discussion.
- GitHub repositories of actively maintained CPU graphics projects often include detailed READMEs and issues that illuminate practical considerations.

E.6 Recommended Development Environments

E.6.1 Integrated Development Environments (IDEs)

- **Visual Studio 2022**
  Best suited for Windows developers with strong support for debugging, profiling, and unit testing of CPU graphics code.
- **CLion (JetBrains)**
  Cross-platform IDE with intelligent C++ support and integration with CMake, useful for Linux and macOS CPU graphics development.
- **VSCode** with appropriate extensions
  Lightweight editor favored for flexible configurations and remote development setups.

E.6.2 Build Systems

- **CMake (3.20+)**
  Industry-standard cross-platform build system, supporting modern C++ standards and easy integration with assembly files and third-party libraries.
- **Ninja Build System**

High-performance build tool often used in combination with CMake to speed up incremental builds.

E.7 Advanced Topics for Further Study

E.7.1 SIMD and Vectorization

- Deep dive into vector instruction sets like AVX2, AVX-512 (Intel), and SVE (ARM) is essential for high-performance CPU graphics programming.
- Understanding compiler auto-vectorization and intrinsic functions enables developers to write code that fully exploits CPU parallelism.

E.7.2 Software Rasterization Architectures

- Exploring different software rasterization approaches such as tile-based rasterizers, hierarchical rasterizers, and hybrid ray tracing techniques helps in designing efficient CPU renderers.

E.7.3 Memory Management and Cache Optimization

- Advanced study of cache line utilization, prefetching strategies, and false sharing prevention leads to dramatic performance gains.

E.7.4 Multi-Core and NUMA Awareness

- On modern multi-socket and NUMA architectures, understanding memory locality and thread affinity is crucial for scalable CPU rendering solutions.

E.8 Summary

This appendix outlined a comprehensive set of modern resources and references to empower readers seeking to expand their expertise in CPU-only graphics programming. By engaging with the recommended books, tools, libraries, research, and community knowledge, readers can stay at the forefront of software rendering techniques and optimization strategies suitable for today's CPU architectures and development environments.

References

Core Libraries and APIs

- **SDL2 Documentation** (v2.0.10 – v2.30), Simple DirectMedia Layer, maintained by Sam Lantinga and contributors, continuous updates through 2025.
- **SDL GitHub Repository** – Source code, changelogs, and issue tracker for SDL2 post-2019: extensive support for software rendering surfaces.
- **Microsoft Windows SDK** – GDI, Win32 API, and Direct2D fallback modes for software rasterization; documentation via Microsoft Learn, 2020–2025.
- **CMake Official Documentation** – Cross-platform build system used to manage SDL2, OpenWatcom, and custom CPU renderers; updates from 3.16 to 3.29+ (2019–2025).
- **MinGW-w64 Project** – Windows-native GCC toolchain, regularly updated with support for latest C++ standards and improved CRT compatibility, 2020–2025.
- **MSVC (Microsoft Visual C++) Compiler Toolset** – Standard-compliant C++ compiler used in Visual Studio 2019–2025, with improvements in debugging, memory management, and inline assembly support.

Linux/Unix Development Tools

- **GCC Compiler (GNU Compiler Collection)** – Support for -O2/-O3 optimization, SSE/AVX extensions used in software rendering; updated versions 10.0 to 14.x (2020–2025).
- **Clang/LLVM Compiler Suite** – Modern C++ support, sanitizers, and better diagnostics for low-level memory access; Clang 10–18 (2020–2025).
- **X.Org Foundation Documentation** – Framebuffer and X11 rendering updates; API details related to window creation, pixel format management, and raw surface access.
- **Linux Kernel Documentation** – Framebuffer and Direct Rendering Manager (DRM) subsystems (versions 5.5 to 6.9); essential for direct-to-framebuffer programming.
- **libdrm and Mesa 3D Project** – CPU-rendered OpenGL software fallback paths, LLVMpipe enhancements post-2020.

Emulators, Retro Systems, and Low-Level Testing

- **86Box Emulator** – Accurate emulation of 286/386/486-class CPUs and VGA hardware; post-2020 releases support low-level timing and DMA behavior.
- **DOSBox Staging Branch** – Modern fork of DOSBox with improved build systems, SDL2 support, and cycle-accurate emulation for Mode 13h experiments.
- **PCem Emulator** – Legacy platform emulator with expanded chipset support and VBE testing for pre-GPU software; maintained through 2023.

- **OpenWatcom v2 Fork** – Retro C/C++ compiler updated for modern platforms, used for DOS real-mode software rendering experiments; builds available post-2021.

CPU Rendering Engines and Research Projects

- **Handmade Hero** by Casey Muratori – Extensive real-time software rendering tutorials using Win32 API and GDI; community-supported updates post-2020.
- **Tiny Renderer and Variants** – C++ CPU-based rasterization examples explaining vector math, z-buffering, and perspective correction (2020–2024 community forks).
- **olc::PixelGameEngine (PGE)** – A minimalist CPU graphics library using SDL or native APIs, widely used in game jams and retro development (updates 2020–2024).
- **NesEmu, Chip8Emu, and GameboyEmu Projects** – Popular emulation engines that implement full graphics pipelines using raw pixel buffers on the CPU (2020–2024 forks).
- **The Cherno's Engine Series** – Open-source tutorials using CPU framebuffers with SDL2 or Win32, geared toward teaching rendering logic (updated 2020–2023).
- **Dear ImGui with Software Renderer Backends** – Software-only rendering integrations of GUI systems for headless or embedded CPU-based graphics apps (2021–2024).

Academic and Technical Whitepapers

- **"Optimizing Software Rendering for Modern CPUs"**, Intel Developer Zone – Whitepaper on cache behavior, SIMD, and pipeline efficiency (2020–2023).
- **"Software Rasterization in Modern Systems"**, published research articles in ACM/IEEE conferences, focused on low-latency rendering for virtual machines and headless devices (2020–2022).
- **"Designing Real-Time Renderers for Embedded CPUs"**, industry whitepapers from Arm Holdings and Imagination Technologies (2021–2024).
- **LLVMpipe Documentation** – CPU-based OpenGL driver powered by LLVM, providing insight into general-purpose software rendering strategies (2020–2024).
- **Intel® 64 and IA-32 Architectures Software Developer Manuals**, Volume 1–3, regularly updated (2020–2024) for low-level memory access, instruction encoding, and SIMD acceleration.

Programming Languages and Standards

- **ISO/IEC 14882:2020 (C++20 Standard)** – Adopted in most modern compilers and used in this book for ranges, constexpr, and modules.
- **ISO/IEC 14882:2023 (C++23 Draft)** – Concepts, static reflection, and improvements for memory-safe rendering loops.
- **Modern CMake Practices** – Cross-platform build strategies for CPU-only graphics engines, maintained via open-source books and GitHub repositories (2020–2024).

- **POSIX Specification Updates (IEEE 1003.1-202x)** – For systems programming and framebuffer access on UNIX-like platforms.

Additional Tools and Build Systems

- **Ninja Build System** – Frequently used with CMake for fast builds in cross-platform CPU rendering projects (versions 1.10 – 1.12, 2020–2024).
- **Visual Studio Code + CMake Tools Extension** – Integrated build environment for SDL2/GDI projects targeting CPU-only rendering on Windows and Linux (2020–2025).
- **AddressSanitizer (ASan) and UndefinedBehaviorSanitizer (UBSan)** – Tools for memory safety verification during rendering loop development in Clang/GCC (2020–2024).
- **Catch2 and GoogleTest Frameworks** – Used for testing CPU rasterizers, math libraries, and pixel manipulation functions; maintained and updated post-2020.