# Contents

# Preface

## Why This Book?

Every programmer who has built a game or graphical application has used a graphics library. SDL, SFML, raylib - these tools abstract away the complexity of talking to the operating system. But have you ever wondered what happens beneath the surface?

This book takes you on a journey to build a 2D graphics library from scratch in Go. No external dependencies, no C bindings - just pure Go communicating directly with the X11 display server.

## What You'll Learn

By the end of this book, you will understand:

- **Binary protocols**: How to encode and decode binary data to communicate with system services
- **X11 architecture**: The client-server model that powers Linux graphics
- **Window management**: Creating, positioning, and decorating windows
- **Event systems**: Handling keyboard, mouse, and window events
- **Software rendering**: Building a framebuffer and implementing drawing primitives
- **API design**: Creating clean, user-friendly interfaces
- **Systems programming**: Working with sockets, file descriptors, and system calls

## Who This Book Is For

This book is for programmers who:

- Know the basics of Go (or another C-like language)
- Are curious about how things work "under the hood"
- Want to understand graphics programming at a fundamental level
- Enjoy building things from first principles

You don't need prior experience with graphics programming or X11.

## How to Read This Book

The book is structured as a progressive tutorial. Each chapter builds on the previous one, and by the end, you'll have a working graphics library capable of running games.

**Part I: Foundation** covers Go fundamentals relevant to systems programming - binary encoding, byte manipulation, and network sockets.

**Part II: X11 Protocol** dives deep into the X Window System, from connection handshakes to authentication.

**Part III: Window Management** teaches you to create and control windows.

**Part IV: Event Handling** covers input processing and the event loop.

**Part V: Graphics** builds the rendering pipeline, from pixels to primitives.

**Part VI: API Design** wraps everything in a clean, user-friendly interface.

**Part VII: Projects** puts it all together with real applications.

**Part VIII: Extensions** explores advanced topics for those who want to go further.

## The Library We'll Build

Our library, called **Glow**, will support:

- Window creation with titles and close buttons
- Keyboard and mouse input
- A canvas for drawing pixels, lines, rectangles, circles, and triangles
- Predefined and custom colors
- Non-blocking event handling

By the final chapter, you'll use Glow to build Pong, a paint application, and a particle system.

## Source Code

All code from this book is available at:

`github.com/AchrafSoltani/glow`

Each chapter corresponds to a step in the `tutorial/` folder.

## Conventions

Code examples appear in monospace:

```
func main() {
    fmt.Println("Hello, Glow!")
}
```

Terminal commands are prefixed with `$`:

```
$ go run main.go
```

Important notes appear in callout boxes:

> **Note**: X11 uses little-endian byte order on most systems.

## Acknowledgments

This book was written with assistance from Claude, an AI assistant by Anthropic.

---

Let's begin.

# Table of Contents

- 5.2 The Xauthority File
- 5.3 MIT-MAGIC-COOKIE-1
- 5.4 Parsing Xauthority Entries
- 5.5 Sending Credentials

## Part III: Window Management

### Chapter 6: Creating Windows

- 6.1 The CreateWindow Request
- 6.2 Window Attributes and Masks
- 6.3 Mapping and Unmapping
- 6.4 Window Hierarchy
- 6.5 Destroying Windows

### Chapter 7: Window Properties

- 7.1 Understanding Atoms
- 7.2 The InternAtom Request
- 7.3 Setting the Window Title
- 7.4 The Close Button Protocol
- 7.5 EWMH and ICCCM Basics

## Part IV: Event Handling

### Chapter 8: The Event System

- 8.1 Event Types Overview
- 8.2 Event Masks
- 8.3 Reading Events from the Socket
- 8.4 Parsing Event Data
- 8.5 Keyboard Events
- 8.6 Mouse Events
- 8.7 Window Events

### Chapter 9: Non-Blocking Events

- 9.1 The Problem with Blocking
- 9.2 Goroutines and Channels
- 9.3 Building an Event Queue
- 9.4 Polling vs Waiting
- 9.5 Thread Safety Considerations

## Part V: Graphics

### Chapter 10: The Graphics Context

- 10.1 What is a GC?
- 10.2 Creating a Graphics Context

- 10.3 GC Attributes
- 10.4 Foreground and Background Colors
- 10.5 Freeing Resources

### Chapter 11: Rendering with PutImage

- 11.1 Image Formats in X11
- 11.2 ZPixmap Explained
- 11.3 The PutImage Request
- 11.4 Pixel Byte Order (BGRA)
- 11.5 The Request Size Limit
- 11.6 Splitting Large Images

### Chapter 12: The Framebuffer

- 12.1 Software Rendering Basics
- 12.2 Designing the Framebuffer
- 12.3 Setting Pixels
- 12.4 Clearing the Screen
- 12.5 Coordinate Systems

### Chapter 13: Drawing Primitives

- 13.1 Lines with Bresenham's Algorithm
- 13.2 Rectangles (Filled and Outline)
- 13.3 Circles with the Midpoint Algorithm
- 13.4 Triangles
- 13.5 Clipping

## Part VI: API Design

### Chapter 14: The Public Interface

- 14.1 Design Principles
- 14.2 The Window Type
- 14.3 The Canvas Type
- 14.4 Color Handling
- 14.5 Error Handling

### Chapter 15: Event Abstraction

- 15.1 Defining Event Types
- 15.2 Key Codes and Mapping
- 15.3 Mouse Buttons
- 15.4 The PollEvent Pattern
- 15.5 Quit Events

## Part VII: Projects

### Chapter 16: Pong

- 16.1 Game Architecture
- 16.2 The Game Loop
- 16.3 Paddle and Ball Physics
- 16.4 Collision Detection
- 16.5 Scoring and Display
- 16.6 Input Handling

### Chapter 17: Paint

- 17.1 Application Design
- 17.2 Tool System
- 17.3 Brush and Eraser
- 17.4 Shape Tools
- 17.5 Color Palette
- 17.6 UI Elements

### Chapter 18: Particle System

- 18.1 Particle Basics
- 18.2 Physics Simulation
- 18.3 Emitter Patterns
- 18.4 Visual Effects
- 18.5 Performance Optimization

## Part VIII: Extensions

### Chapter 19: Sprites and Images

- 19.1 Image File Formats
- 19.2 Loading PNG Files
- 19.3 Sprite Rendering
- 19.4 Transparency and Alpha Blending

### Chapter 20: Text Rendering

- 20.1 Bitmap Fonts
- 20.2 TrueType Fonts
- 20.3 Font Rasterization
- 20.4 Text Layout

### Chapter 21: Audio

- 21.1 Audio on Linux
- 21.2 The oto Library
- 21.3 Loading WAV Files
- 21.4 Sound Effects and Music

### Chapter 22: Gamepad Support

- 22.1 Linux Joystick API
- 22.2 Reading Input Events
- 22.3 Controller Mapping
- 22.4 Dead Zones and Calibration

### Chapter 23: Performance Optimization

- 23.1 Profiling Go Code
- 23.2 MIT-SHM Extension
- 23.3 Shared Memory Rendering
- 23.4 Dirty Rectangles

### Chapter 24: Cross-Platform

- 24.1 Build Tags in Go
- 24.2 Platform Abstraction
- 24.3 Windows with Win32
- 24.4 macOS with Cocoa

## Appendices

### Appendix A: X11 Protocol Reference

- Request Opcodes
- Event Types
- Error Codes
- Atom Names

### Appendix B: Key Code Tables

- X11 Key Codes
- Common Keyboard Layouts

### Appendix C: Troubleshooting

- Connection Errors
- Authentication Failures
- Rendering Issues
- Performance Problems

### Appendix D: Further Reading

- Books
- Specifications
- Open Source Projects

# Chapter 1: Introduction

## 1.1 What We're Building

In this book, we'll build **Glow**, a 2D graphics library for Linux. When we're done, you'll be able to write programs like this:

```
package main

import "github.com/AchrafSoltani/glow"

func main() {
    win, _ := glow.NewWindow("Hello Glow", 800, 600)
    defer win.Close()

    canvas := win.Canvas()
    running := true

    for running {
        for e := win.PollEvent(); e != nil; e = win.PollEvent() {
            if e.Type == glow.EventQuit {
                running = false
            }
        }

        canvas.Clear(glow.Black)
        canvas.FillCircle(400, 300, 50, glow.Red)
        win.Present()
    }
}
```

This simple program creates a window, draws a red circle, and responds to the close button. Behind these few lines of code lies a complex system: binary protocol encoding, socket communication, event parsing, and pixel manipulation.

By building this library ourselves, we'll understand exactly what happens when you call `NewWindow()` or `FillCircle()`.

## 1.2 Why Pure Go?

Most graphics libraries use **cgo** to call C code. SDL, GLFW, and raylib all have Go bindings that wrap their C implementations. This approach is pragmatic - these libraries are mature, optimized, and cross-platform.

But cgo has drawbacks:

- **Compilation complexity**: You need a C compiler and the library's development headers
- **Cross-compilation difficulty**: Building for different platforms becomes harder
- **Debugging challenges**: Stack traces cross the Go/C boundary
- **Deployment friction**: You may need to ship shared libraries

More importantly for our purposes, cgo hides the interesting parts. When you call `SDL_CreateWindow()`, you don't see the system calls, the protocol messages, or the data structures.

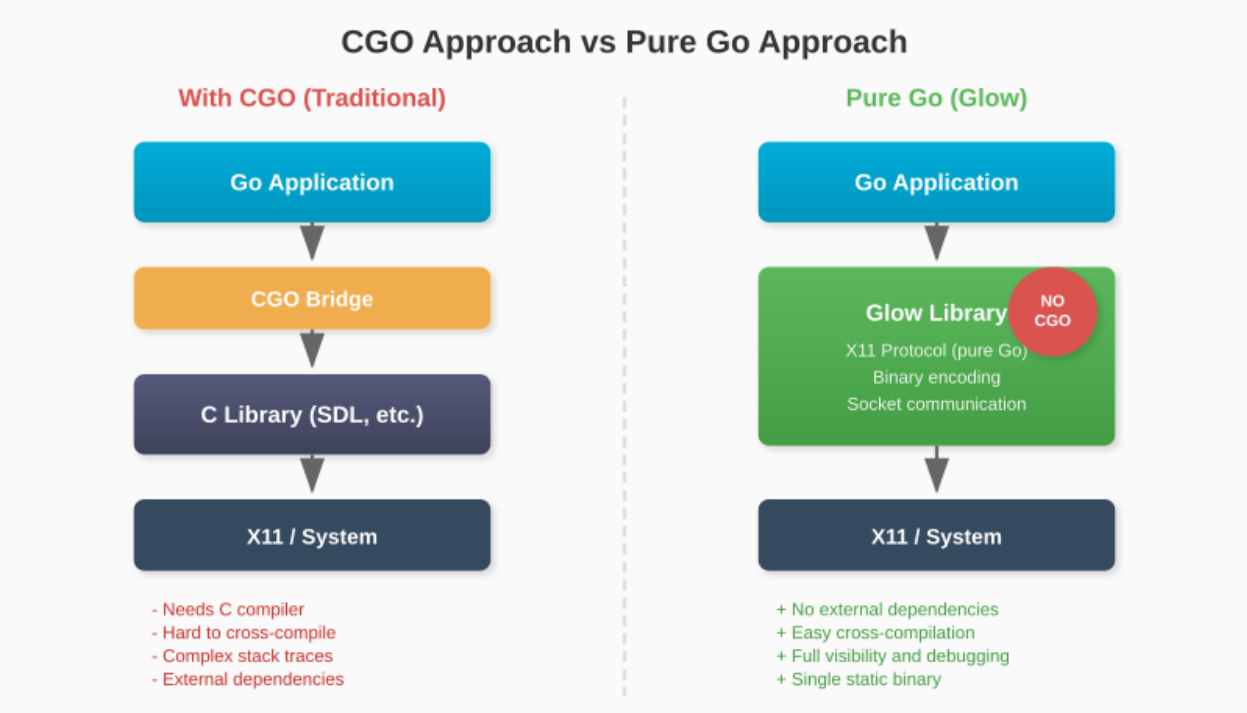


Figure 1: CGO vs Pure Go comparison

By using pure Go, we:

- Keep the entire implementation visible and debuggable
- Eliminate external dependencies
- Learn how graphics systems actually work
- Produce a single, statically-linked binary

Our library talks directly to the X11 server using Go's `net` package for sockets and `encoding/binary` for protocol encoding. No C code anywhere.

## 1.3 Architecture Overview

Here's how Glow is structured:

**The X11 Server** is a program that manages the display. It runs continuously, handling requests from client applications (like ours) to create windows, draw graphics, and deliver input events.

**The Unix Domain Socket** is how we communicate with the X server. It's a local socket (no network involved) that allows bidirectional communication.

**Our X11 Layer** encodes requests into the binary format X11 expects, sends them over the socket, and decodes the responses.

**The Public API** provides a clean interface that hides the X11 complexity from application developers.

## 1.4 Setting Up the Project

Let's create the project structure:

```
$ mkdir glow
$ cd glow
$ go mod init github.com/yourusername/glow
```

Create the directory structure:

The `internal/` directory is special in Go - packages inside it can only be imported by code in the parent module. This lets us keep the X11 implementation private while exposing a clean public API.

### Your First File

Create `internal/x11/protocol.go` with some constants we'll need:

```go
package x11

// X11 Protocol Version
const (
    ProtocolMajorVersion = 11
    ProtocolMinorVersion = 0
)

// Opcodes for X11 requests
const (
    OpCreateWindow  = 1
    OpDestroyWindow = 4
    OpMapWindow     = 8
    OpUnmapWindow   = 10
    OpInternAtom    = 16
    OpChangeProperty = 18
    OpCreateGC      = 55
    OpFreeGC        = 60
```

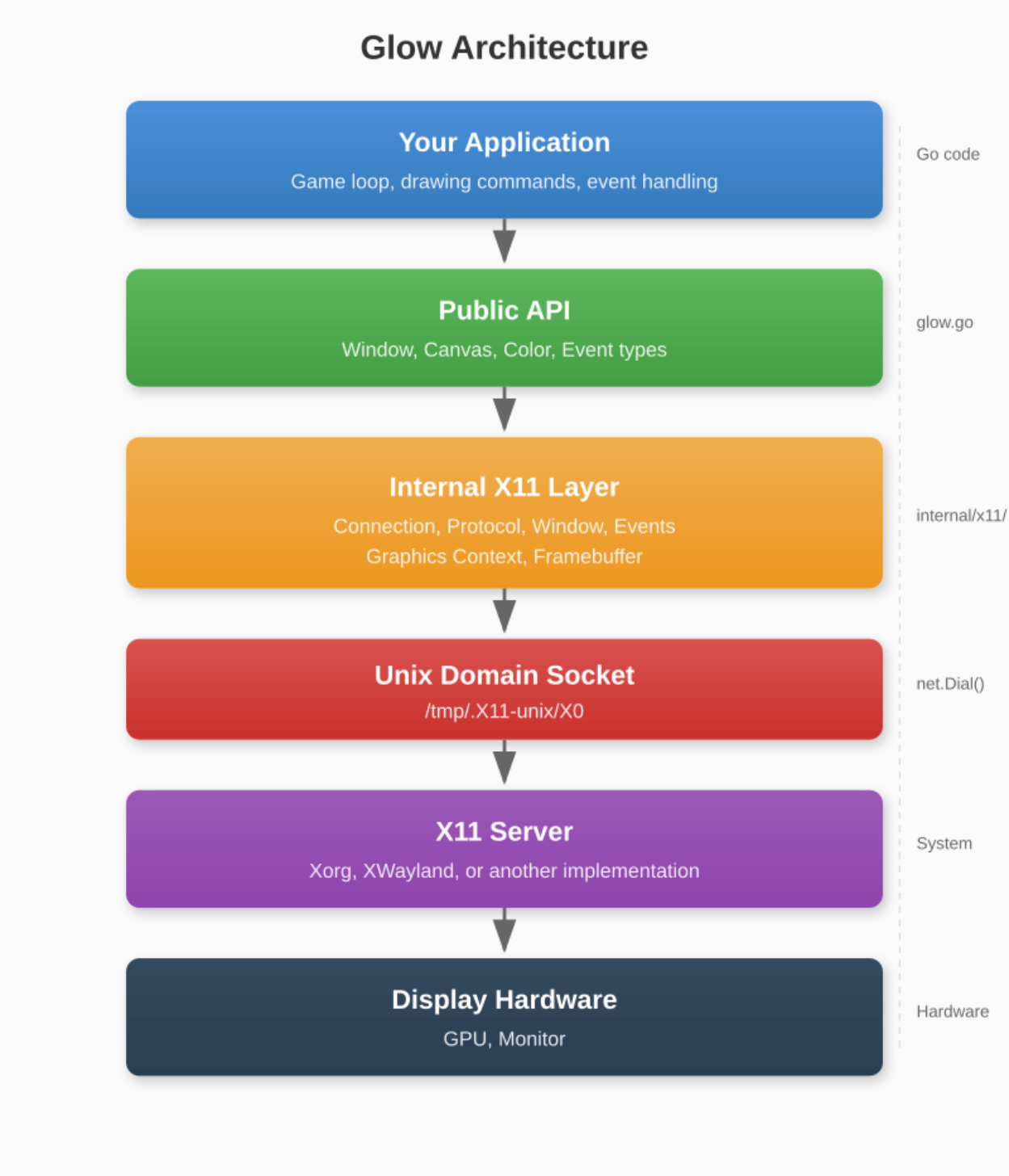


Figure 2: Glow Architecture Overview


Figure 3: Project Structure

```
    OpPutImage      = 72
)

// Event types
const (
    EventKeyPress       = 2
    EventKeyRelease     = 3
    EventButtonPress    = 4
    EventButtonRelease  = 5
    EventMotionNotify   = 6
    EventExpose         = 12
    EventConfigureNotify = 22
    EventClientMessage  = 33
)
```

We'll add to this file as we discover more of the protocol.

### Verifying Your Setup

Create a simple test file to ensure everything compiles:

```
// examples/hello/main.go
package main

import "fmt"

func main() {
    fmt.Println("Glow project initialized!")
}
```

Run it:

```
$ go run examples/hello/main.go
Glow project initialized!
```

With the project structure in place, we're ready to dive into the technical foundations. In the next chapter, we'll explore how Go handles binary data - essential knowledge for implementing the X11 protocol.

---

**Key Takeaways:**

- Glow is a pure Go graphics library that communicates directly with X11
- No cgo means simpler builds, easier debugging, and full visibility into the implementation
- The architecture layers a clean public API over internal X11 protocol handling
- X11 uses Unix domain sockets for local communication

# Chapter 2: Go for Systems Programming

Before we can speak the X11 protocol, we need to master Go's tools for working with binary data. This chapter covers the essential techniques for encoding, decoding, and transmitting bytes.

## 2.1 Binary Data and Byte Slices

At the lowest level, all data is bytes. A string is bytes. An integer is bytes. A network packet is bytes. The X11 protocol is a stream of bytes with specific meanings at specific positions.

In Go, we work with bytes using the `[]byte` type - a slice of bytes:

```
// Create a byte slice with 8 bytes, all zero
data := make([]byte, 8)

// Set individual bytes
data[0] = 0x48  // 'H'
data[1] = 0x65  // 'e'
data[2] = 0x6C  // 'l'
data[3] = 0x6C  // 'l'
data[4] = 0x6F  // 'o'

fmt.Println(string(data[:5]))  // "Hello"
```

Byte slices are the foundation of all our protocol work. We'll create them, fill them with encoded data, and send them over sockets.

### Hexadecimal Notation

Bytes are often written in hexadecimal (base 16). Each hex digit represents 4 bits, so two hex digits represent one byte:

```
Binary:      0101 1010
Hexadecimal: 5    A
Decimal:     90

In Go: 0x5A == 90
```

We'll use hex notation frequently because it maps cleanly to bytes and is standard in protocol documentation.

## 2.2 The encoding/binary Package

Go's `encoding/binary` package is our primary tool for converting between Go types and byte sequences.

### Writing Integers to Bytes

To put a `uint32` into a byte slice:

```
import "encoding/binary"

data := make([]byte, 4)
binary.LittleEndian.PutUint32(data, 0x12345678)

fmt.Printf("% X\n", data)  // 78 56 34 12
```

Notice the byte order: the value `0x12345678` becomes `78 56 34 12`. The least significant byte (`78`) comes first. This is **little-endian** order, which X11 uses on most systems.

The package provides methods for different integer sizes:

```
binary.LittleEndian.PutUint16(data, value)  // 2 bytes
binary.LittleEndian.PutUint32(data, value)  // 4 bytes
binary.LittleEndian.PutUint64(data, value)  // 8 bytes
```

### Reading Integers from Bytes

To extract a `uint32` from a byte slice:

```
data := []byte{0x78, 0x56, 0x34, 0x12}
value := binary.LittleEndian.Uint32(data)

fmt.Printf("0x%X\n", value)  // 0x12345678
```

### Working with Slices

These functions operate on slice positions, not the entire slice:

```
data := make([]byte, 12)

// Write at different offsets
binary.LittleEndian.PutUint32(data[0:], 0xAAAAAAAA)
binary.LittleEndian.PutUint32(data[4:], 0xBBBBBBBB)
binary.LittleEndian.PutUint32(data[8:], 0xCCCCCCCC)

// The slice notation [n:] means "from position n to the end"
```

This pattern appears constantly in protocol code: create a byte slice, then write values at specific offsets.

## 2.3 Endianness Explained

Endianness determines the order of bytes when storing multi-byte values.

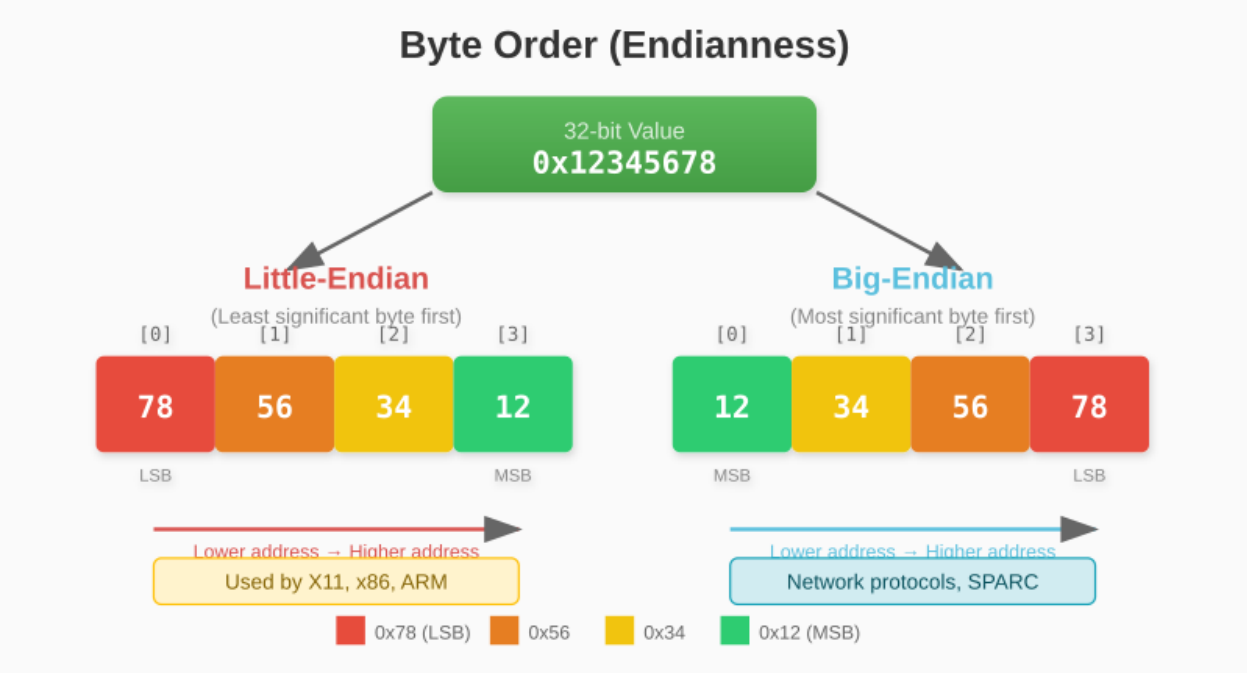


Figure 4: Little-Endian vs Big-Endian

X11 supports both, but requests that the client declare its byte order at connection time. Modern x86 and ARM systems are little-endian, so we'll use `binary.LittleEndian` throughout.

> **Note**: The first byte we send to the X server is either `'l'` (0x6C) for little-endian or `'B'` (0x42) for big-endian. This tells the server how to interpret all subsequent data.

## 2.4 Working with Buffers

When building X11 requests, we often know the exact size needed:

For variable-length requests, calculate the size first:

```
func buildRequest(data []byte) []byte {
    // Header is 24 bytes, data follows
    totalLen := 24 + len(data)

    // Pad to 4-byte boundary
    padding := (4 - (totalLen % 4)) % 4
    totalLen += padding
```

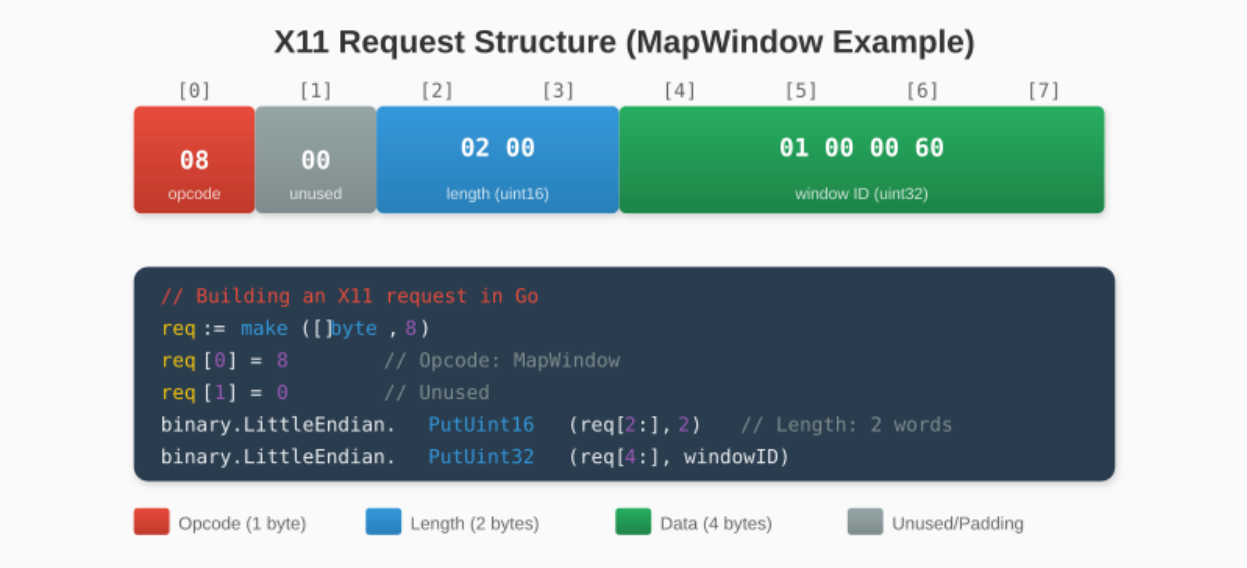


Figure 5: X11 Request Structure

```
    req := make([]byte, totalLen)
    // ... fill in the header and data
    return req
}
```

### Padding

X11 requires all requests to be a multiple of 4 bytes. The padding formula handles this:

```
padding := (4 - (length % 4)) % 4
```

If `length` is 10: - `10 % 4 = 2` (remainder) - `4 - 2 = 2` (bytes needed to reach next multiple of 4) - `2 % 4 = 2` (padding bytes to add)

If `length` is 12: - `12 % 4 = 0` (already aligned) - `4 - 0 = 4` - `4 % 4 = 0` (no padding needed)

## 2.5 Network Sockets in Go

X11 communication happens over Unix domain sockets. These are like TCP sockets but for local communication only - no network involved.

### Connecting to a Unix Socket

```
import "net"

conn, err := net.Dial("unix", "/tmp/.X11-unix/X0")
if err != nil {
    return err
```

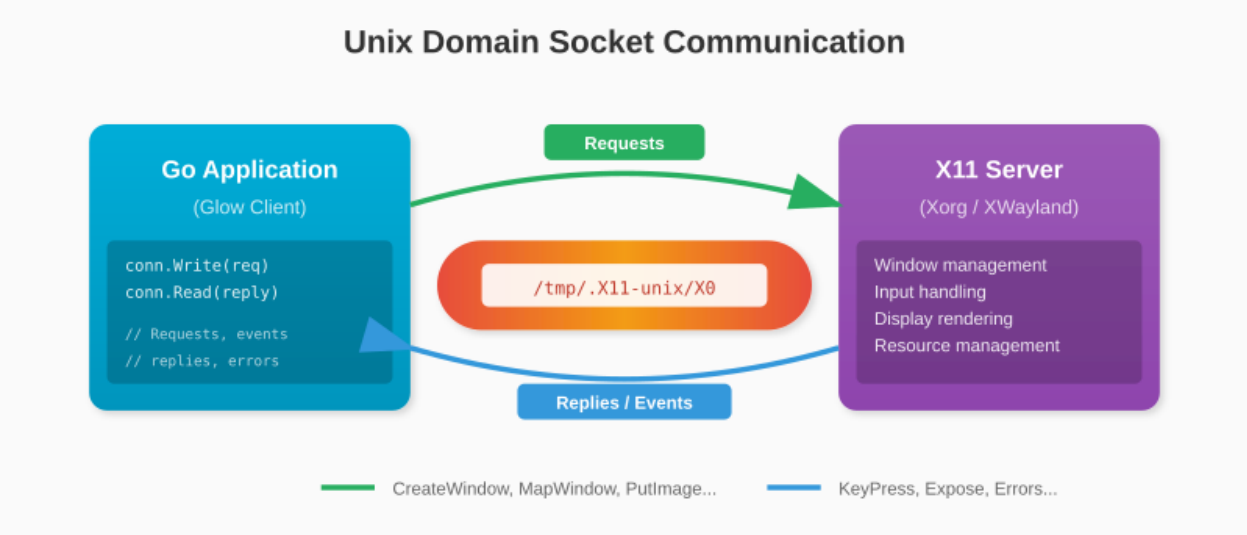


Figure 6: Unix Domain Socket Communication

```
}
defer conn.Close()
```

The path `/tmp/.X11-unix/X0` is the socket for display `:0`. Display `:1` would use `X1`, and so on.

### Writing Data

```
data := []byte{0x6C, 0x00, 0x0B, 0x00}  // Start of X11 handshake
n, err := conn.Write(data)
if err != nil {
    return err
}
// n contains the number of bytes written
```

`Write` may write fewer bytes than requested (though this is rare for small writes). For large data, you might need to loop:

```
func writeAll(conn net.Conn, data []byte) error {
    for len(data) > 0 {
        n, err := conn.Write(data)
        if err != nil {
            return err
        }
        data = data[n:]
    }
    return nil
}
```

### Reading Data

```
buf := make([]byte, 1024)
n, err := conn.Read(buf)
if err != nil {
    return err
}
// buf[:n] contains the data read
```

For X11, we often need to read an exact number of bytes:

```
import "io"

buf := make([]byte, 32)  // X11 events are exactly 32 bytes
_, err := io.ReadFull(conn, buf)
if err != nil {
    return err
}
```

`io.ReadFull` keeps reading until the buffer is full or an error occurs. This is essential for reading fixed-size protocol structures.

### The Connection Type

Our X11 connection wrapper will look like this:

```
type Connection struct {
    conn net.Conn  // The underlying socket

    // Information from the server
    RootWindow  uint32
    RootDepth   uint8
    ScreenWidth uint16
    // ... more fields
}

func Connect() (*Connection, error) {
    conn, err := net.Dial("unix", "/tmp/.X11-unix/X0")
    if err != nil {
        return nil, err
    }

    c := &Connection{conn: conn}

    // Perform handshake...

    return c, nil
}
```

### Bidirectional Communication

X11 uses a single socket for both directions:

- **Client → Server**: Requests (create window, draw, etc.)
- **Server → Client**: Replies, events, and errors

This means we need to be careful about when we read. If we send a request that generates a reply, we must read that reply before doing anything else that reads from the socket.

For events, we'll use a goroutine to continuously read from the socket and queue events for the main thread to process.

---

**Key Takeaways:**

- `[]byte` is the foundation of protocol work in Go
- `encoding/binary` converts between integers and bytes
- X11 uses little-endian byte order (on most systems)
- All X11 requests must be padded to 4-byte boundaries
- Unix domain sockets provide local, bidirectional communication
- `io.ReadFull` ensures we read exactly the bytes we expect

With these tools in hand, we're ready to explore the X11 protocol itself.

# Chapter 3: Understanding X11

X11, also known as the X Window System, has powered Unix graphics for over 35 years. Before we write any code, let's understand its architecture and protocol design.

## 3.1 A Brief History

The X Window System was developed at MIT in 1984. The "X" came from being the successor to a system called "W" (for "Window"). The "11" refers to version 11 of the protocol, released in 1987 - the version still in use today.

X11's longevity comes from its flexible, network-transparent design. The same protocol that displays windows on your local monitor can display them across a network on a remote machine.

While modern Linux systems are transitioning to Wayland, X11 remains widely used. Even on Wayland systems, XWayland provides X11 compatibility. The concepts you learn here apply broadly to graphics system design.

## 3.2 The Client-Server Model

X11 uses a client-server architecture that may seem backwards at first:

- **The X Server** runs on the machine with the display. It owns the screen, keyboard, and mouse.
- **X Clients** are applications that want to display windows. They connect to the server and send requests.

The terminology can be confusing because in web development, "server" usually means the remote machine. In X11, the server is local - it serves access to your display.

This design enables:

- **Multiple clients**: Many applications share one display
- **Network transparency**: Clients can run on remote machines
- **Display management**: The server handles window stacking, focus, and input routing

## 3.3 Protocol Structure

X11 communication consists of four message types:

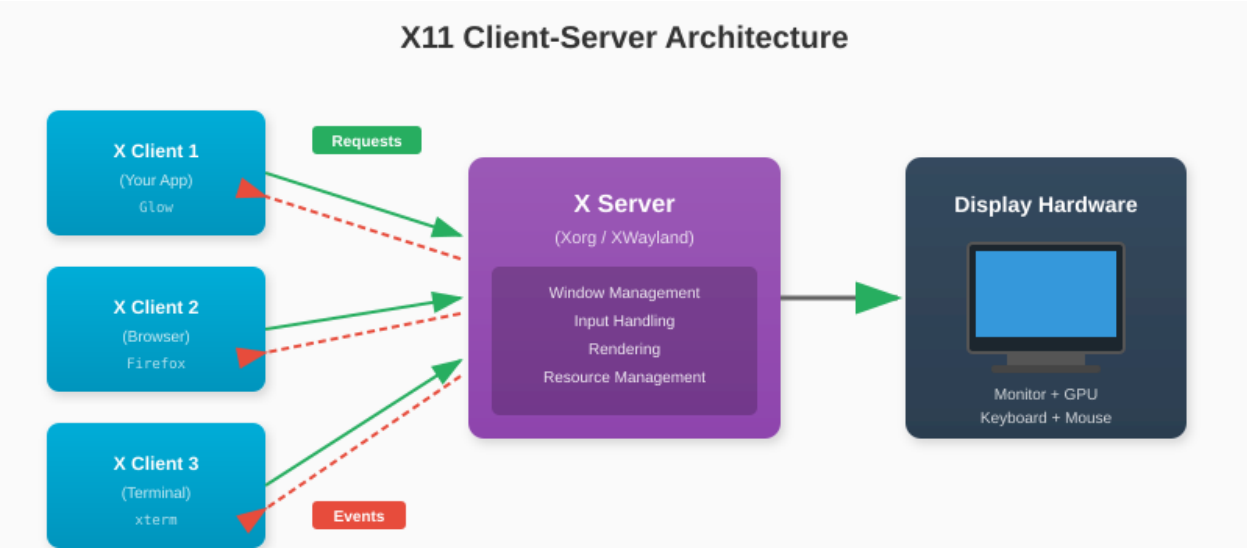


Figure 7: X11 Client-Server Architecture

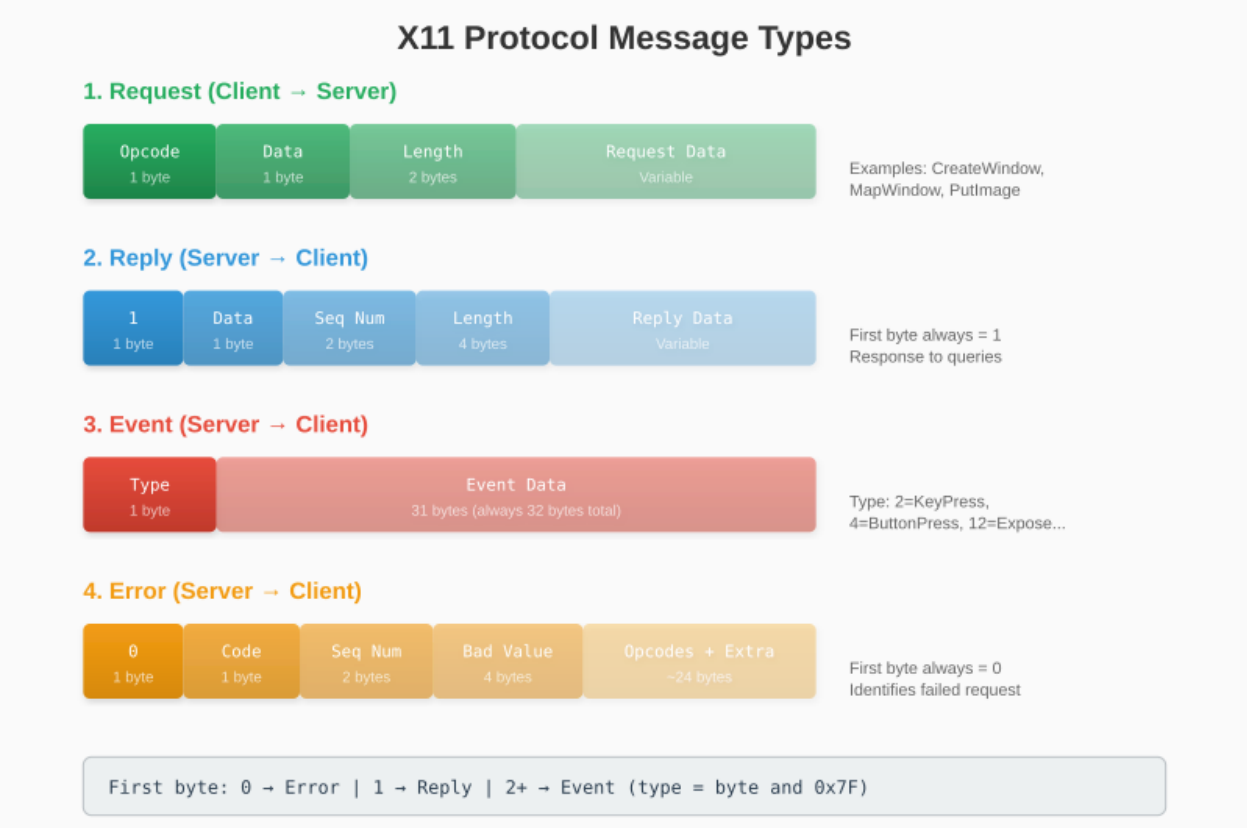


Figure 8: X11 Protocol Message Types

**Requests** ask the server to do something. **Replies** return requested data. **Events** notify about user input and window changes. **Errors** indicate failed requests.

### Distinguishing Message Types

When reading from the socket, check the first byte:

```
firstByte := buf[0]

if firstByte == 0 {
    // Error response
} else if firstByte == 1 {
    // Reply to a request
} else {
    // Event (type = firstByte & 0x7F)
}
```

The `& 0x7F` masks off the high bit, which indicates whether the event was sent by another client via `SendEvent`.

## 3.4 Requests, Replies, Events, and Errors

Let's trace through a typical interaction:

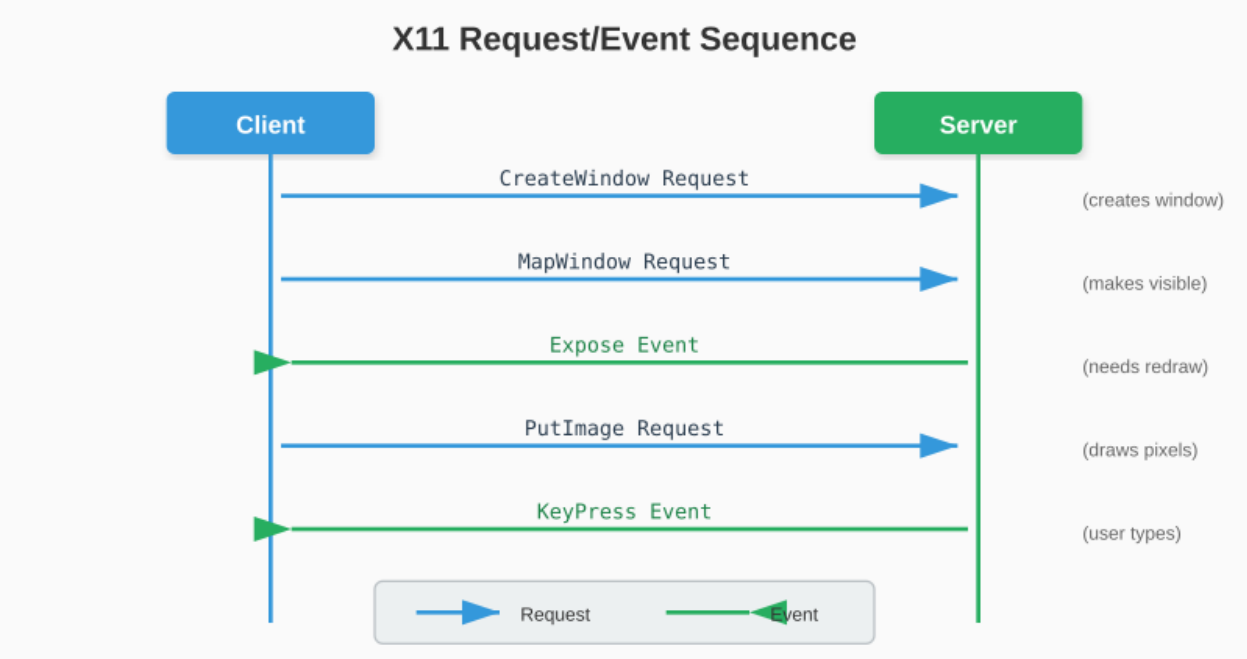


Figure 9: X11 Request/Event Sequence

**Important**: X11 is asynchronous. After sending CreateWindow, we don't wait for confirmation - we continue sending requests. The server processes them in order. If something fails, we'll eventually receive an error.

This asynchronous design improves performance (no round-trip latency per request) but means errors arrive later than you might expect.

## 3.5 Resource IDs

X11 uses 32-bit IDs to identify resources like windows, graphics contexts, and pixmaps.

During connection, the server provides: - **Resource ID Base**: Starting point for our IDs - **Resource ID Mask**: Bits we can use

We generate IDs by combining these:

```
func (c *Connection) GenerateID() uint32 {
    id := c.nextID
    c.nextID++
    return (id & c.ResourceIDMask) | c.ResourceIDBase
}
```

For example, with base `0x04000000` and mask `0x001FFFFF`: - First ID: `0x04000000` - Second ID: `0x04000001` - Third ID: `0x04000002`

The base ensures our IDs don't conflict with other clients or the server's own resources.

### Resource Lifecycle

Resources must be explicitly destroyed:

```
// Create
windowID, _ := conn.CreateWindow(...)

// Use
conn.MapWindow(windowID)

// Destroy when done
conn.DestroyWindow(windowID)
```

Failing to destroy resources causes leaks in the X server. When a client disconnects, the server cleans up its resources, but proper cleanup is good practice.

---

**Key Takeaways:**

- X11 uses a client-server model where the server owns the display
- Four message types: requests, replies, events, errors
- All messages have a defined binary structure
- Communication is asynchronous - errors arrive later
- Resources are identified by 32-bit IDs that clients generate
- The protocol has remained stable since 1987

Now that we understand the protocol's structure, let's connect to the X server.

# Chapter 4: Connecting to the X Server

With our understanding of the X11 protocol, let's establish our first connection. This chapter implements the handshake that initiates communication with the X server.

## 4.1 Finding the Display

The `DISPLAY` environment variable tells applications which X server to use:

```
$ echo $DISPLAY
:0
```

The format is `[host]:display[.screen]`: - `:0` - Local display 0, screen 0 - `:0.1` - Local display 0, screen 1 - `localhost:0` - Network connection to localhost - `192.168.1.100:0` - Remote display

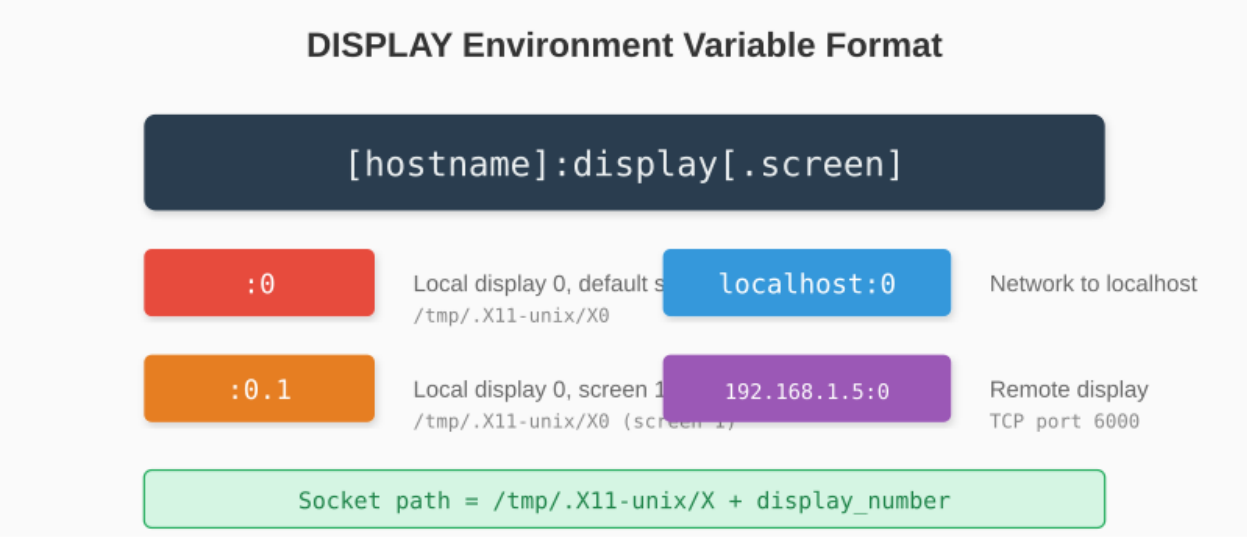


Figure 10: DISPLAY Environment Variable Format

For local connections (no host or `localhost`), we use Unix domain sockets. For remote connections, we'd use TCP (which we won't cover).

Let's parse the display string:

```go
func parseDisplay() (displayNum string, err error) {
    display := os.Getenv("DISPLAY")
    if display == "" {
        display = ":0"  // Default
    }

    // Find the colon
    idx := strings.Index(display, ":")
    if idx == -1 {
        return "", fmt.Errorf("invalid DISPLAY: %s", display)
    }

    // Extract display number (ignore screen)
    rest := display[idx+1:]
    if dotIdx := strings.Index(rest, "."); dotIdx != -1 {
        rest = rest[:dotIdx]
    }

    return rest, nil
}
```

## 4.2 Unix Domain Sockets

Unix domain sockets provide inter-process communication on the same machine. They appear as files in the filesystem but behave like network sockets.

X11 servers listen on `/tmp/.X11-unix/Xn` where `n` is the display number:

```go
displayNum, _ := parseDisplay()
socketPath := fmt.Sprintf("/tmp/.X11-unix/X%s", displayNum)

conn, err := net.Dial("unix", socketPath)
if err != nil {
    return nil, fmt.Errorf("failed to connect to X11: %w", err)
}
```

This gives us a `net.Conn` - a bidirectional byte stream to the X server.

## 4.3 The Connection Handshake

Before any X11 requests, we must complete a handshake. The client sends setup information, and the server responds with display details.

### Client Setup Request

The initial message declares our byte order and protocol version:

Without authentication (we'll add that in Chapter 5):

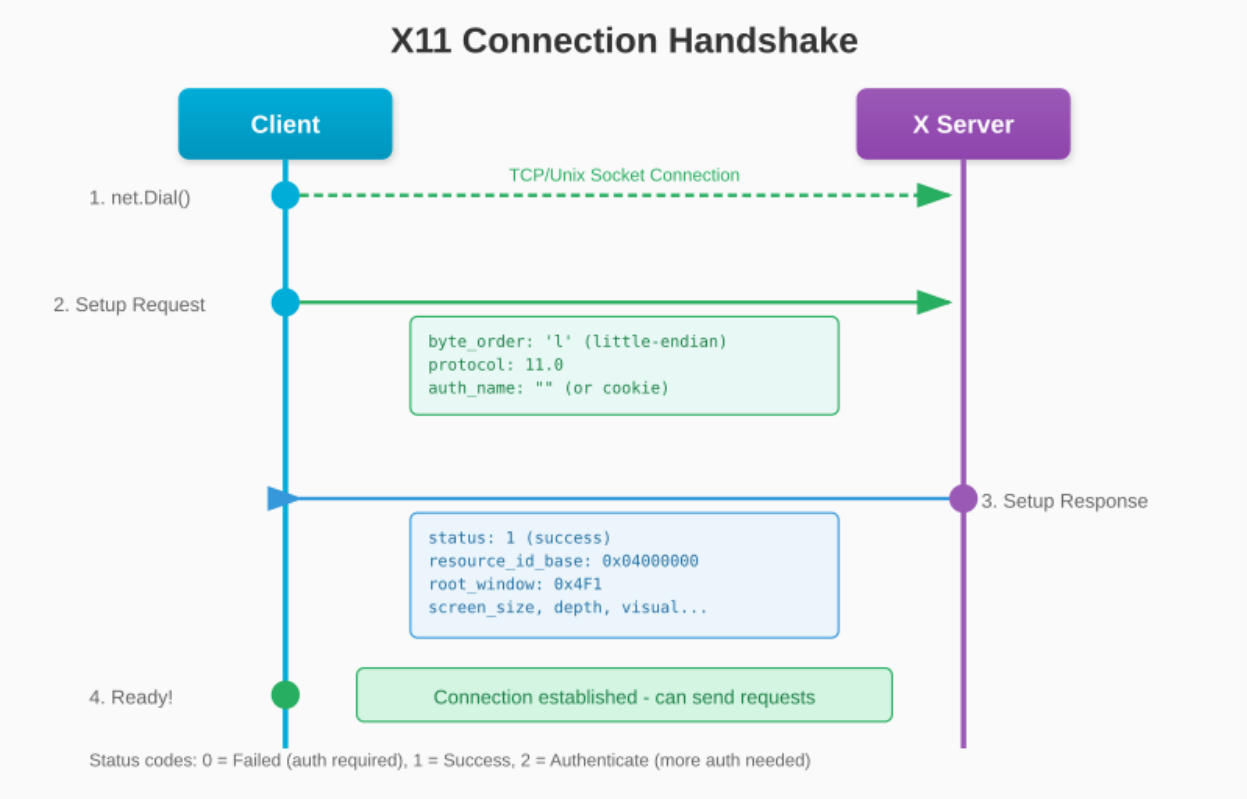


Figure 11: X11 Connection Handshake

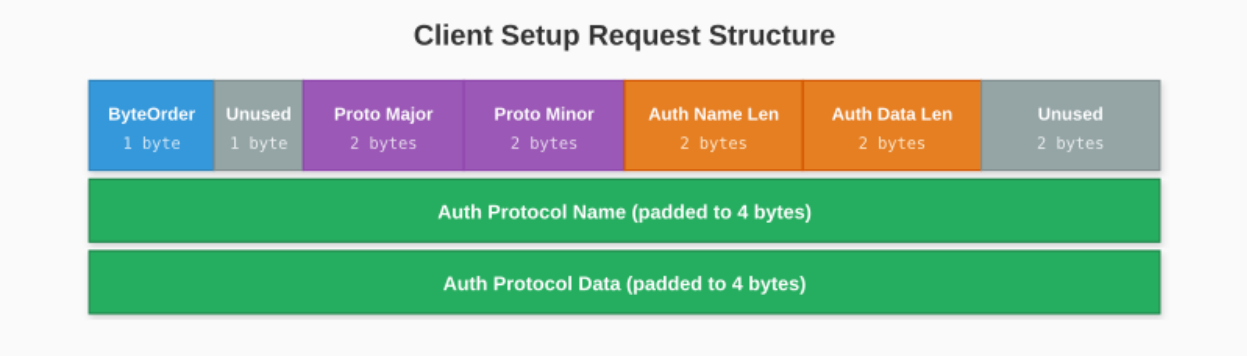


Figure 12: Client Setup Request Structure

```go
func (c *Connection) handshake() error {
    // Build setup request (12 bytes minimum)
    setup := make([]byte, 12)

    setup[0] = 'l'  // Little-endian byte order
    setup[1] = 0    // Unused
    binary.LittleEndian.PutUint16(setup[2:], 11)  // Protocol major: 11
    binary.LittleEndian.PutUint16(setup[4:], 0)   // Protocol minor: 0
    binary.LittleEndian.PutUint16(setup[6:], 0)   // Auth name length: 0
    binary.LittleEndian.PutUint16(setup[8:], 0)   // Auth data length: 0
    binary.LittleEndian.PutUint16(setup[10:], 0)  // Unused

    _, err := c.conn.Write(setup)
    if err != nil {
        return fmt.Errorf("failed to send setup: %w", err)
    }

    return c.readSetupResponse()
}
```

### Server Response

The server responds with one of three statuses:

```go
func (c *Connection) readSetupResponse() error {
    // Read initial 8 bytes
    header := make([]byte, 8)
    if _, err := io.ReadFull(c.conn, header); err != nil {
        return err
    }

    status := header[0]

    switch status {
    case 0:  // Failed
        return c.handleSetupFailed(header)
    case 1:  // Success
        return c.handleSetupSuccess(header)
    case 2:  // Authenticate
        return errors.New("server requires authentication")
    default:
        return fmt.Errorf("unknown status: %d", status)
    }
}
```

**Status 0 (Failed)**: The server rejected us. This usually means authentication is required.

```go
func (c *Connection) handleSetupFailed(header []byte) error {
    reasonLen := header[1]
```

```
    // Read additional data...
    reason := make([]byte, reasonLen)
    c.conn.Read(reason)
    return fmt.Errorf("connection refused: %s", string(reason))
}
```

**Status 2 (Authenticate)**: The server wants more authentication. We'll handle this in Chapter 5.

**Status 1 (Success)**: We're in! Now we parse the setup information.

## 4.4 Parsing the Setup Response

On success, the server sends detailed information about the display. This is a complex structure, but we only need a few fields.

The header we already read contains: - Byte 0: Status (1 for success) - Bytes 6-7: Additional data length (in 4-byte units)

```
func (c *Connection) handleSetupSuccess(header []byte) error {
    // Additional data length is in 4-byte units
    additionalLen := binary.LittleEndian.Uint16(header[6:]) * 4

    // Read the rest
    data := make([]byte, additionalLen)
    if _, err := io.ReadFull(c.conn, data); err != nil {
        return err
    }

    // Parse fixed fields
    c.ResourceIDBase = binary.LittleEndian.Uint32(data[4:8])
    c.ResourceIDMask = binary.LittleEndian.Uint32(data[8:12])

    // Vendor string and format info
    vendorLen := binary.LittleEndian.Uint16(data[16:18])
    numFormats := data[21]
    numScreens := data[20]

    if numScreens == 0 {
        return errors.New("no screens available")
    }

    // Calculate offset to screen info
    vendorPadded := (vendorLen + 3) &^ 3  // Round up to 4
    formatSize := uint16(numFormats) * 8
    screenOffset := 32 + vendorPadded + formatSize

    // Parse first screen
    return c.parseScreen(data[screenOffset:])
```

```
}
```

## 4.5 Extracting Screen Information

The screen structure contains essential rendering information:

```
func (c *Connection) parseScreen(screen []byte) error {
    c.RootWindow = binary.LittleEndian.Uint32(screen[0:4])
    c.ScreenWidth = binary.LittleEndian.Uint16(screen[20:22])
    c.ScreenHeight = binary.LittleEndian.Uint16(screen[22:24])
    c.RootDepth = screen[38]
    c.RootVisual = binary.LittleEndian.Uint32(screen[32:36])

    // Initialize ID generator
    c.nextID = c.ResourceIDBase

    return nil
}
```

Key fields:

- **RootWindow**: The ID of the root window (the desktop background)
- **ScreenWidth/Height**: Display dimensions in pixels
- **RootDepth**: Color depth (usually 24 for true color)
- **RootVisual**: The visual type ID (describes pixel format)

### The Complete Connection Type

Putting it together:

```
type Connection struct {
    conn net.Conn

    // From server setup
    ResourceIDBase uint32
    ResourceIDMask uint32
    RootWindow     uint32
    RootVisual     uint32
    RootDepth      uint8
    ScreenWidth    uint16
    ScreenHeight   uint16

    // ID generation
    nextID uint32
}

func Connect() (*Connection, error) {
    displayNum, err := parseDisplay()
    if err != nil {
```

```
        return nil, err
    }

    socketPath := fmt.Sprintf("/tmp/.X11-unix/X%s", displayNum)
    conn, err := net.Dial("unix", socketPath)
    if err != nil {
        return nil, fmt.Errorf("failed to connect: %w", err)
    }

    c := &Connection{conn: conn}

    if err := c.handshake(); err != nil {
        conn.Close()
        return nil, err
    }

    return c, nil
}

func (c *Connection) Close() error {
    return c.conn.Close()
}
```

### Testing the Connection

Let's verify it works:

```
// examples/connect/main.go
package main

import (
    "fmt"
    "log"

    "github.com/yourusername/glow/internal/x11"
)

func main() {
    conn, err := x11.Connect()
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    fmt.Println("Connected to X11!")
    fmt.Printf("Screen: %dx%d\n", conn.ScreenWidth, conn.ScreenHeight)
    fmt.Printf("Root Depth: %d\n", conn.RootDepth)
    fmt.Printf("Root Window: 0x%X\n", conn.RootWindow)
```

```
}
```

```
$ go run examples/connect/main.go
Connected to X11!
Screen: 1920x1080
Root Depth: 24
Root Window: 0x4F1
```

If you see "connection refused" or "authentication required", don't worry - we'll handle authentication in the next chapter.

---

**Key Takeaways:**

- The `DISPLAY` environment variable identifies which X server to use
- Local connections use Unix domain sockets at `/tmp/.X11-unix/Xn`
- The handshake declares byte order and protocol version
- The server responds with display capabilities and resource ID ranges
- We extract essential info: root window, screen size, color depth

Our connection is open, but most X servers require authentication. Let's add that next.

# Chapter 5: Authentication

Most X servers require authentication before accepting connections. Without it, any program could connect and spy on your keystrokes or capture your screen. This chapter implements the standard authentication mechanism.

## 5.1 Why Authentication?

In the early days of X11, authentication was optional. The server would accept any local connection. This made sense when computers were single-user and physically secured.

Today, authentication prevents:

- **Keystroke logging**: Malicious programs reading your passwords
- **Screen capture**: Unauthorized screenshots
- **Input injection**: Programs sending fake keyboard/mouse events
- **Window manipulation**: Hiding or moving other applications' windows

The most common authentication method is **MIT-MAGIC-COOKIE-1**, a simple shared-secret scheme.

## 5.2 The Xauthority File

X11 stores authentication credentials in `~/.Xauthority`. This binary file contains entries mapping displays to authentication cookies.

Each entry has:

- **Family**: Connection type (256 = local, 0 = Internet, etc.)
- **Address**: Hostname or empty for local
- **Display**: Display number as string ("0", "1", etc.)
- **Auth Name**: Protocol name ("MIT-MAGIC-COOKIE-1")
- **Auth Data**: The actual cookie (16 bytes for MIT-MAGIC-COOKIE-1)

### Entry Structure in Detail

Each variable field is preceded by a 2-byte big-endian length:

```
type AuthEntry struct {
    Family  uint16
    Address string
```

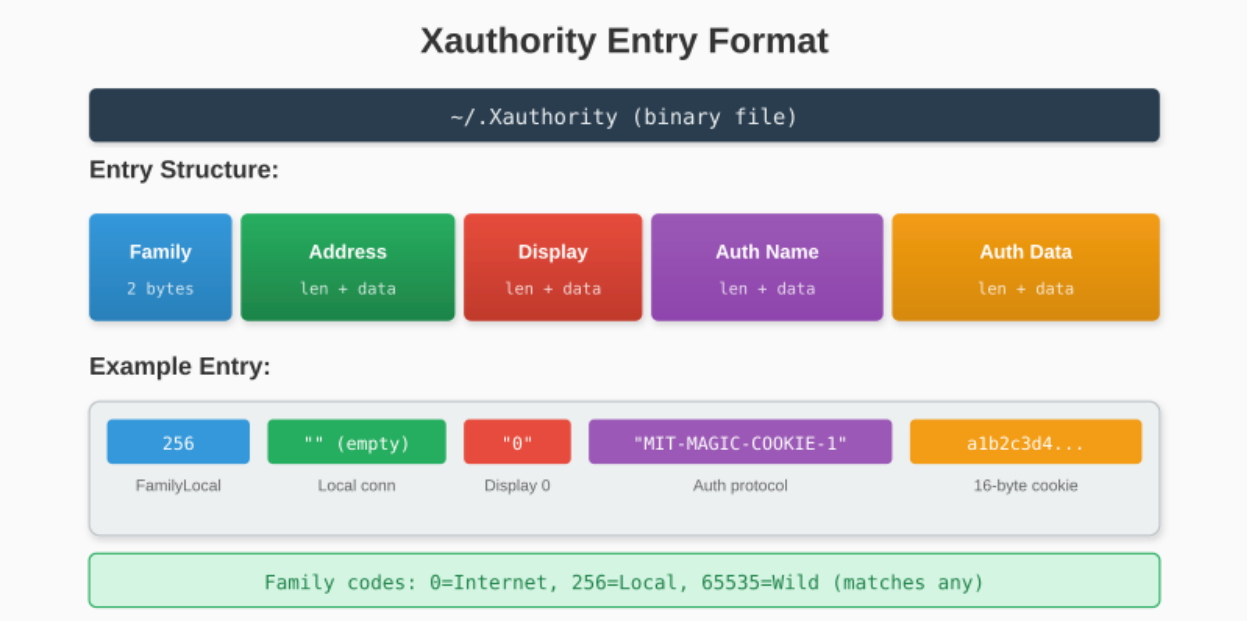


Figure 13: Xauthority Entry Format

```
    Display string
    Name    string
    Data    []byte
}
```

## 5.3 MIT-MAGIC-COOKIE-1

This is the simplest and most common X11 authentication:

1. The X server generates a random 16-byte cookie at startup
2. The cookie is written to `~/.Xauthority`
3. Clients read the cookie and send it during connection
4. The server compares cookies - if they match, access is granted

It's called "magic cookie" because knowing the cookie grants access - no cryptographic handshake, just possession of the secret.

> **Security Note**: This scheme protects against casual snooping but not against root or someone with access to your home directory. For stronger security, X11 supports other mechanisms like MIT-KERBEROS-5.

## 5.4 Parsing Xauthority Entries

Let's implement the parser:

```
// internal/x11/auth.go
package x11
```

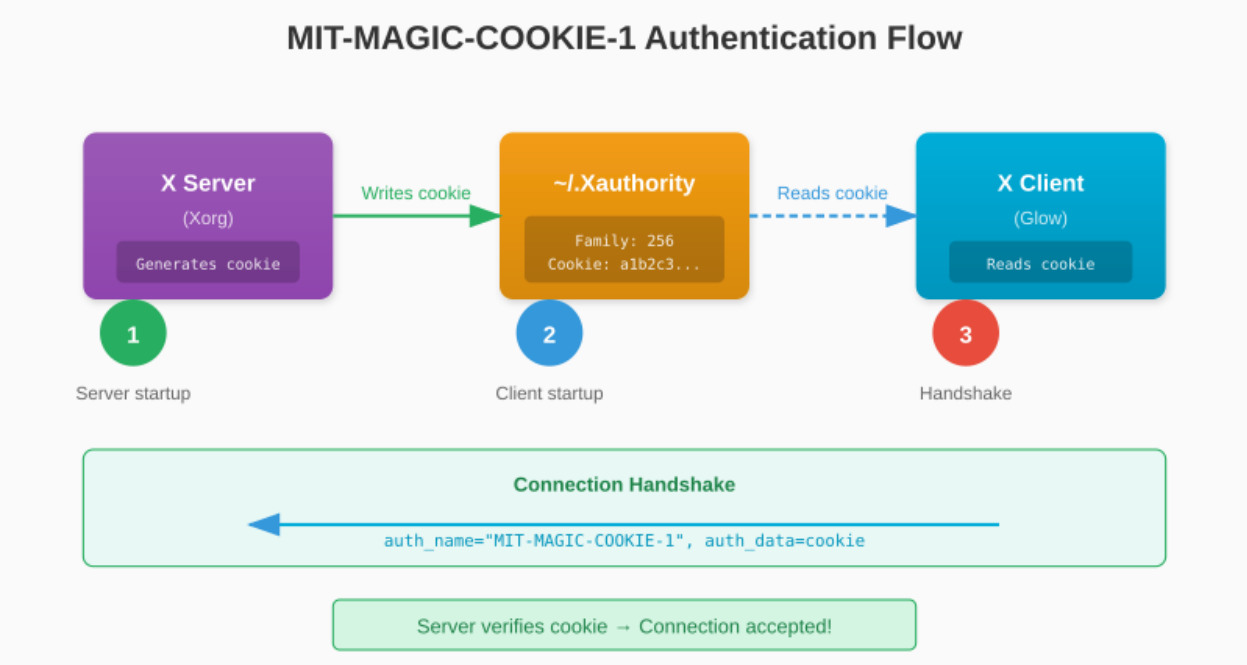


Figure 14: MIT-MAGIC-COOKIE-1 Authentication Flow

```
import (
    "encoding/binary"
    "io"
    "os"
    "path/filepath"
)

type AuthEntry struct {
    Family  uint16
    Address string
    Display string
    Name    string
    Data    []byte
}

func ReadXauthority() ([]AuthEntry, error) {
    // Find the file
    path := os.Getenv("XAUTHORITY")
    if path == "" {
        home, err := os.UserHomeDir()
        if err != nil {
            return nil, err
        }
        path = filepath.Join(home, ".Xauthority")
    }
```

```
    file, err := os.Open(path)
    if err != nil {
        return nil, err
    }
    defer file.Close()

    return parseXauthority(file)
}
```

The parsing function reads entries until EOF:

```
func parseXauthority(r io.Reader) ([]AuthEntry, error) {
    var entries []AuthEntry

    for {
        entry, err := readAuthEntry(r)
        if err == io.EOF {
            break
        }
        if err != nil {
            return nil, err
        }
        entries = append(entries, entry)
    }

    return entries, nil
}
```

Each entry is read field by field:

```
func readAuthEntry(r io.Reader) (AuthEntry, error) {
    var entry AuthEntry

    // Family (2 bytes, big-endian)
    family := make([]byte, 2)
    if _, err := io.ReadFull(r, family); err != nil {
        return entry, err
    }
    entry.Family = binary.BigEndian.Uint16(family)

    // Address
    addr, err := readString(r)
    if err != nil {
        return entry, err
    }
    entry.Address = addr

    // Display number
```

```
    display, err := readString(r)
    if err != nil {
        return entry, err
    }
    entry.Display = display

    // Auth name
    name, err := readString(r)
    if err != nil {
        return entry, err
    }
    entry.Name = name

    // Auth data
    data, err := readBytes(r)
    if err != nil {
        return entry, err
    }
    entry.Data = data

    return entry, nil
}
```

Helper functions for reading length-prefixed data:

```
func readString(r io.Reader) (string, error) {
    data, err := readBytes(r)
    if err != nil {
        return "", err
    }
    return string(data), nil
}

func readBytes(r io.Reader) ([]byte, error) {
    // Length is 2 bytes, big-endian
    lenBuf := make([]byte, 2)
    if _, err := io.ReadFull(r, lenBuf); err != nil {
        return nil, err
    }
    length := binary.BigEndian.Uint16(lenBuf)

    if length == 0 {
        return nil, nil
    }

    data := make([]byte, length)
    if _, err := io.ReadFull(r, data); err != nil {
        return nil, err
```

```
    }
    return data, nil
}
```

## 5.5 Sending Credentials

Now we need to find the right entry and include it in our connection:

```
func FindAuth(entries []AuthEntry, displayNum string) *AuthEntry {
    for i := range entries {
        entry := &entries[i]

        // Match display number
        if entry.Display != displayNum {
            continue
        }

        // We only support MIT-MAGIC-COOKIE-1
        if entry.Name != "MIT-MAGIC-COOKIE-1" {
            continue
        }

        // Match local connections
        // Family 256 = FamilyLocal, 65535 = FamilyWild
        if entry.Family == 256 || entry.Family == 65535 {
            return entry
        }
    }
    return nil
}
```

Update the handshake to include authentication:

```
func (c *Connection) handshake() error {
    // Try to get authentication
    var authName, authData []byte

    entries, err := ReadXauthority()
    if err == nil {
        if auth := FindAuth(entries, "0"); auth != nil {
            authName = []byte(auth.Name)
            authData = auth.Data
        }
    }
    // If no auth found, proceed without (may work on some systems)

    // Calculate padding
    authNamePad := (4 - (len(authName) % 4)) % 4
```

```
    authDataPad := (4 - (len(authData) % 4)) % 4

    // Build setup request
    setupLen := 12 + len(authName) + authNamePad + len(authData) + authDataPad
    setup := make([]byte, setupLen)

    setup[0] = 'l'  // Little-endian
    setup[1] = 0    // Unused
    binary.LittleEndian.PutUint16(setup[2:], 11)  // Protocol major
    binary.LittleEndian.PutUint16(setup[4:], 0)   // Protocol minor
    binary.LittleEndian.PutUint16(setup[6:], uint16(len(authName)))
    binary.LittleEndian.PutUint16(setup[8:], uint16(len(authData)))
    binary.LittleEndian.PutUint16(setup[10:], 0)  // Unused

    // Copy auth name (with padding)
    copy(setup[12:], authName)

    // Copy auth data (with padding)
    authDataOffset := 12 + len(authName) + authNamePad
    copy(setup[authDataOffset:], authData)

    if _, err := c.conn.Write(setup); err != nil {
        return fmt.Errorf("failed to send setup: %w", err)
    }

    return c.readSetupResponse()
}
```

### Debugging Authentication Issues

If connection fails, add some diagnostics:

```
func (c *Connection) handshake() error {
    entries, err := ReadXauthority()
    if err != nil {
        fmt.Fprintf(os.Stderr, "Warning: could not read Xauthority: %v\n", err)
    } else {
        fmt.Fprintf(os.Stderr, "Found %d Xauthority entries\n", len(entries))
        for _, e := range entries {
            fmt.Fprintf(os.Stderr, "  Family=%d Display=%s Name=%s\n",
                e.Family, e.Display, e.Name)
        }
    }
    // ... rest of handshake
}
```

### Complete Auth Module

Here's the complete `auth.go`:

```
package x11

import (
    "encoding/binary"
    "io"
    "os"
    "path/filepath"
)

type AuthEntry struct {
    Family  uint16
    Address string
    Display string
    Name    string
    Data    []byte
}

func ReadXauthority() ([]AuthEntry, error) {
    path := os.Getenv("XAUTHORITY")
    if path == "" {
        home, err := os.UserHomeDir()
        if err != nil {
            return nil, err
        }
        path = filepath.Join(home, ".Xauthority")
    }

    file, err := os.Open(path)
    if err != nil {
        return nil, err
    }
    defer file.Close()

    var entries []AuthEntry
    for {
        entry, err := readAuthEntry(file)
        if err == io.EOF {
            break
        }
        if err != nil {
            return nil, err
        }
        entries = append(entries, entry)
    }
```

```
    return entries, nil
}

func readAuthEntry(r io.Reader) (AuthEntry, error) {
    var entry AuthEntry

    // Family (big-endian)
    family := make([]byte, 2)
    if _, err := io.ReadFull(r, family); err != nil {
        return entry, err
    }
    entry.Family = binary.BigEndian.Uint16(family)

    // Address, Display, Name
    var err error
    entry.Address, err = readLenString(r)
    if err != nil {
        return entry, err
    }
    entry.Display, err = readLenString(r)
    if err != nil {
        return entry, err
    }
    entry.Name, err = readLenString(r)
    if err != nil {
        return entry, err
    }

    // Data
    entry.Data, err = readLenBytes(r)
    return entry, err
}

func readLenString(r io.Reader) (string, error) {
    b, err := readLenBytes(r)
    return string(b), err
}

func readLenBytes(r io.Reader) ([]byte, error) {
    lenBuf := make([]byte, 2)
    if _, err := io.ReadFull(r, lenBuf); err != nil {
        return nil, err
    }
    length := binary.BigEndian.Uint16(lenBuf)

    if length == 0 {
        return nil, nil
```

```
    }

    data := make([]byte, length)
    _, err := io.ReadFull(r, data)
    return data, err
}

func FindAuth(entries []AuthEntry, displayNum string) *AuthEntry {
    for i := range entries {
        e := &entries[i]
        if e.Display == displayNum && e.Name == "MIT-MAGIC-COOKIE-1" {
            if e.Family == 256 || e.Family == 65535 {
                return e
            }
        }
    }
    return nil
}
```

### Testing Authentication

Update the test program:

```
func main() {
    conn, err := x11.Connect()
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    fmt.Println("Connected to X11 with authentication!")
    fmt.Printf("Screen: %dx%d\n", conn.ScreenWidth, conn.ScreenHeight)
}
```

```
$ go run examples/connect/main.go
Connected to X11 with authentication!
Screen: 1920x1080
```

If you still get authentication errors:

1. Check `~/.Xauthority` exists and is readable
2. Verify the display number matches (`echo $DISPLAY`)
3. Try `xauth list` to see what entries exist
4. Some systems use different auth locations - check `XAUTHORITY` env var

---

**Key Takeaways:**

- X11 authentication prevents unauthorized access to your display
- MIT-MAGIC-COOKIE-1 is a simple shared-secret mechanism

- Credentials are stored in `~/.Xauthority` in a binary format
- The cookie is sent during the initial connection handshake
- Family 256 (local) and 65535 (wild) entries match local connections

With authentication working, we can now create windows!

# Chapter 6: Creating Windows

We're connected to the X server. Now let's create something visible - a window. This chapter covers the CreateWindow request and window lifecycle management.

## 6.1 The CreateWindow Request

CreateWindow is one of the most complex X11 requests. It takes many parameters controlling the window's appearance and behavior.

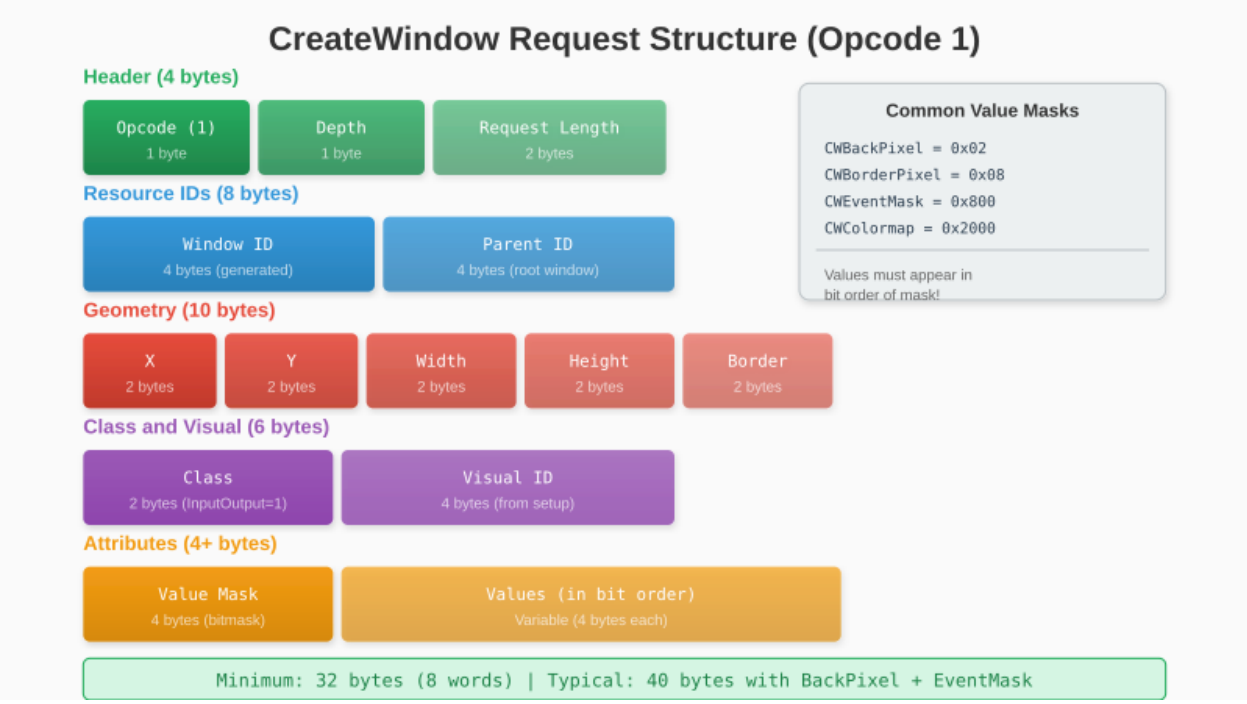


Figure 15: CreateWindow Request Structure

Let's break down each field:

### Window ID

We generate this using our ID generator:

```
windowID := c.GenerateID()
```

### Parent ID

All windows except the root have a parent. For top-level application windows, the parent is the root window:

```
parentID := c.RootWindow
```

### Position and Size

- **X, Y**: Position relative to parent (for root children, this is screen position)
- **Width, Height**: Window dimensions in pixels
- **Border Width**: Thickness of the window border (usually 0 - decorations are handled by the window manager)

### Depth and Visual

- **Depth**: Color depth in bits (use `c.RootDepth` for compatibility)
- **Visual**: The visual type (use `c.RootVisual`)

### Window Class

Two options: - `InputOutput` (1): Normal window that can display graphics - `InputOnly` (2): Invisible window that only receives input

```
const (
    WindowClassInputOutput = 1
    WindowClassInputOnly   = 2
)
```

## 6.2 Window Attributes and Masks

The value mask specifies which optional attributes we're setting. Each bit corresponds to an attribute:

```
const (
    CWBackPixmap       = 1 << 0   // Background pixmap
    CWBackPixel        = 1 << 1   // Background color
    CWBorderPixmap     = 1 << 2   // Border pixmap
    CWBorderPixel      = 1 << 3   // Border color
    CWBitGravity       = 1 << 4   // How contents move on resize
    CWWinGravity       = 1 << 5   // How window moves on parent resize
    CWBackingStore     = 1 << 6   // Backing store hint
    CWBackingPlanes    = 1 << 7   // Planes to preserve
    CWBackingPixel     = 1 << 8   // Value for
    CWOverrideRedirect = 1 << 9   // Bypass window manager
    CWSaveUnder        = 1 << 10  // Save obscured areas
    CWEventMask        = 1 << 11  // Events we want to receive
```

```
    CWDontPropagate    = 1 << 12  // Events not to propagate
    CWColormap         = 1 << 13  // Colormap
    CWCursor           = 1 << 14  // Cursor shape
)
```

For our library, we typically set:

- **CWBackPixel**: Black background
- **CWEventMask**: Which events we want

### Event Masks

Event masks control which events the server sends us:

```
const (
    KeyPressMask        = 1 << 0
    KeyReleaseMask      = 1 << 1
    ButtonPressMask     = 1 << 2
    ButtonReleaseMask   = 1 << 3
    EnterWindowMask     = 1 << 4
    LeaveWindowMask     = 1 << 5
    PointerMotionMask   = 1 << 6
    ExposureMask        = 1 << 15
    StructureNotifyMask = 1 << 17
    // ... and more
)
```

We want: - `KeyPressMask`, `KeyReleaseMask`: Keyboard input - `ButtonPressMask`, `ButtonReleaseMask`: Mouse clicks - `PointerMotionMask`: Mouse movement - `ExposureMask`: Window needs redrawing - `StructureNotifyMask`: Window resized, moved, etc.

## 6.3 Implementing CreateWindow

```
// internal/x11/window.go
package x11

import "encoding/binary"

func (c *Connection) CreateWindow(x, y int16, width, height uint16) (uint32, error) {
    windowID := c.GenerateID()

    // Events we want to receive
    eventMask := uint32(
        KeyPressMask |
        KeyReleaseMask |
        ButtonPressMask |
        ButtonReleaseMask |
        PointerMotionMask |
```

```
        ExposureMask |
        StructureNotifyMask,
    )

    // Attributes we're setting
    valueMask := uint32(CWBackPixel | CWEventMask)
    valueCount := 2  // Two values: background pixel and event mask

    // Request size: header (8 words) + values
    reqLen := 8 + valueCount
    req := make([]byte, reqLen*4)

    // Build the request
    req[0] = OpCreateWindow                                    // Opcode
    req[1] = c.RootDepth                                       // Depth
    binary.LittleEndian.PutUint16(req[2:], uint16(reqLen))     // Length
    binary.LittleEndian.PutUint32(req[4:], windowID)           // Window ID
    binary.LittleEndian.PutUint32(req[8:], c.RootWindow)       // Parent
    binary.LittleEndian.PutUint16(req[12:], uint16(x))         // X
    binary.LittleEndian.PutUint16(req[14:], uint16(y))         // Y
    binary.LittleEndian.PutUint16(req[16:], width)             // Width
    binary.LittleEndian.PutUint16(req[18:], height)            // Height
    binary.LittleEndian.PutUint16(req[20:], 0)                 // Border width
    binary.LittleEndian.PutUint16(req[22:], WindowClassInputOutput)
    binary.LittleEndian.PutUint32(req[24:], c.RootVisual)      // Visual
    binary.LittleEndian.PutUint32(req[28:], valueMask)         // Value mask

    // Values must be in bit order of the mask
    binary.LittleEndian.PutUint32(req[32:], 0x000000)  // CWBackPixel: black
    binary.LittleEndian.PutUint32(req[36:], eventMask) // CWEventMask

    _, err := c.conn.Write(req)
    if err != nil {
        return 0, err
    }

    return windowID, nil
}
```

**Important**: Values must appear in the order of their bits in the mask. Since `CWBackPixel` (bit 1) comes before `CWEventMask` (bit 11), the background color comes first.

## 6.4 Mapping and Unmapping

Creating a window doesn't make it visible. We must **map** it:

```
func (c *Connection) MapWindow(windowID uint32) error {
    req := make([]byte, 8)
    req[0] = OpMapWindow
    req[1] = 0  // Unused
    binary.LittleEndian.PutUint16(req[2:], 2)  // Length: 2 words
    binary.LittleEndian.PutUint32(req[4:], windowID)

    _, err := c.conn.Write(req)
    return err
}
```

To hide a window without destroying it, **unmap** it:

```
func (c *Connection) UnmapWindow(windowID uint32) error {
    req := make([]byte, 8)
    req[0] = OpUnmapWindow
    req[1] = 0
    binary.LittleEndian.PutUint16(req[2:], 2)
    binary.LittleEndian.PutUint32(req[4:], windowID)

    _, err := c.conn.Write(req)
    return err
}
```

## 6.5 Window Hierarchy

X11 windows form a tree:

Events can propagate up this tree. A click on a button might propagate to its parent application window.

For our library, we create simple windows as direct children of the root. Modern toolkits create complex hierarchies for widgets, but we'll keep things simple.

## 6.6 Destroying Windows

When done with a window, destroy it to free resources:

```
func (c *Connection) DestroyWindow(windowID uint32) error {
    req := make([]byte, 8)
    req[0] = OpDestroyWindow
    req[1] = 0
    binary.LittleEndian.PutUint16(req[2:], 2)
    binary.LittleEndian.PutUint32(req[4:], windowID)

    _, err := c.conn.Write(req)
    return err
}
```

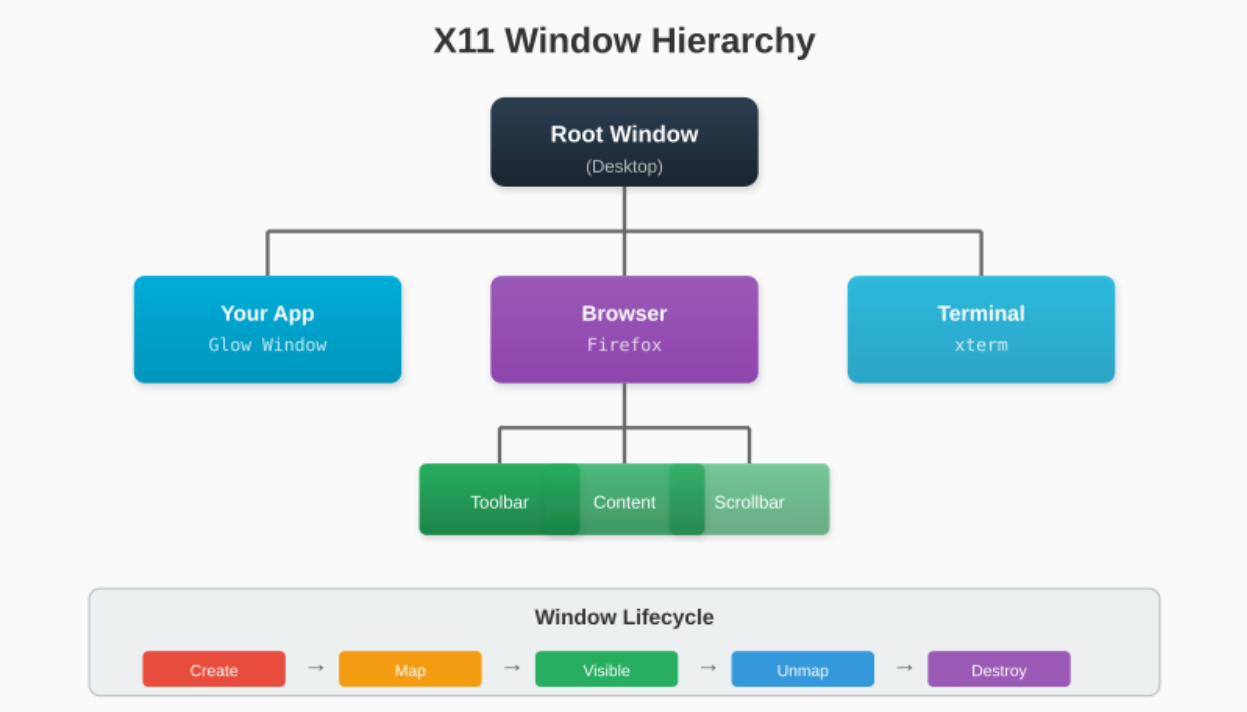


Figure 16: X11 Window Hierarchy

Destroying a window also destroys all its children and associated resources.

### Testing Window Creation

Let's create our first visible window:

```
// examples/window/main.go
package main

import (
    "fmt"
    "log"
    "time"

    "github.com/yourusername/glow/internal/x11"
)

func main() {
    conn, err := x11.Connect()
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    // Create a 400x300 window at position (100, 100)
```

```
    windowID, err := conn.CreateWindow(100, 100, 400, 300)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Printf("Created window: 0x%X\n", windowID)

    // Make it visible
    err = conn.MapWindow(windowID)
    if err != nil {
        log.Fatal(err)
    }
    fmt.Println("Window mapped!")

    // Keep it open for 3 seconds
    time.Sleep(3 * time.Second)

    // Clean up
    conn.DestroyWindow(windowID)
    fmt.Println("Window destroyed")
}
```

Run it:

```
$ go run examples/window/main.go
Created window: 0x4000000
Window mapped!
Window destroyed
```

You should see a black 400x300 window appear for 3 seconds!

### What About Window Decorations?

You might notice our window has a title bar and borders. We didn't create those - the **window manager** added them.

Window managers (GNOME, KDE, i3, etc.) intercept top-level window creation and add decorations. They also handle:

- Window placement
- Minimizing, maximizing
- Alt+Tab switching
- The close button

In the next chapter, we'll learn to communicate with the window manager to set the window title and handle the close button.

---

**Key Takeaways:**

- CreateWindow takes many parameters: position, size, depth, visual, and attributes
- Value masks specify which attributes we're setting

- Event masks control which events we receive
- Windows must be mapped to become visible
- The window manager adds decorations to top-level windows
- Always destroy windows to free resources

Our window exists but can't do much. Let's add window properties next.

# Chapter 7: Window Properties

Our window appears, but it has no title and clicking the close button does nothing. This chapter covers X11 atoms and properties - the mechanism for communicating with the window manager.

## 7.1 Understanding Atoms

X11 uses **atoms** as efficient identifiers for strings. Instead of passing strings like "WM_NAME" in every request, we convert them to 32-bit integers once and reuse them.

Think of atoms as interned strings:

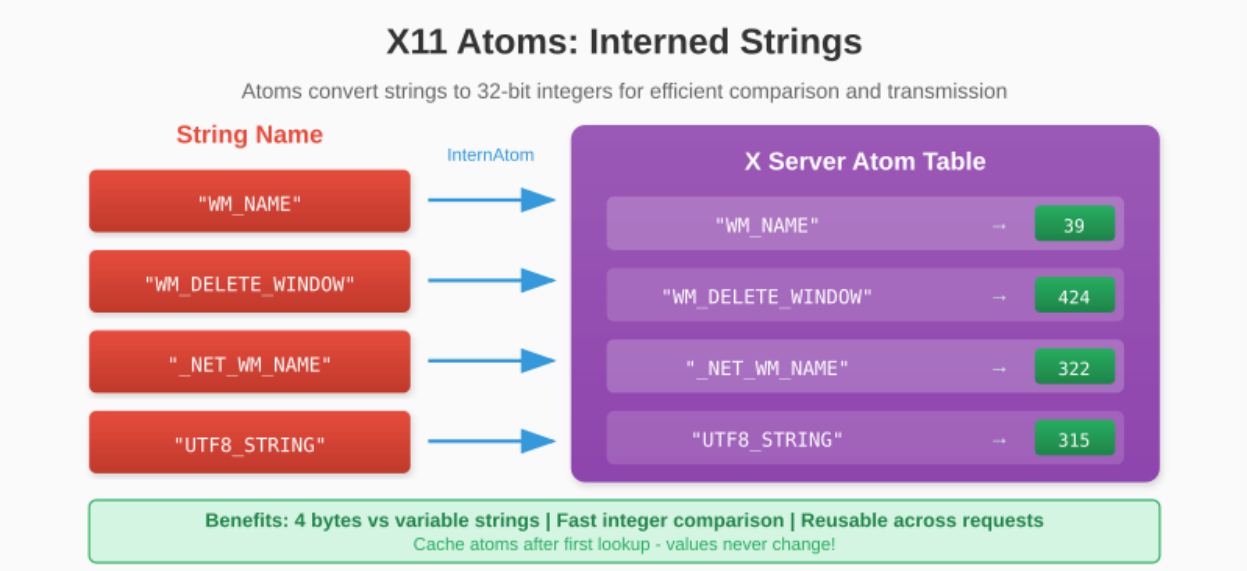


Figure 17: X11 Atoms

The server maintains a global atom table. Standard atoms have predefined values, but applications can create custom atoms too.

### Why Atoms?

1. **Efficiency**: 4 bytes vs. variable-length strings
2. **Comparison**: Integer comparison is faster than string comparison
3. **Standardization**: Well-known atoms have documented meanings

## 7.2 The InternAtom Request

To get an atom for a string, we send an InternAtom request:

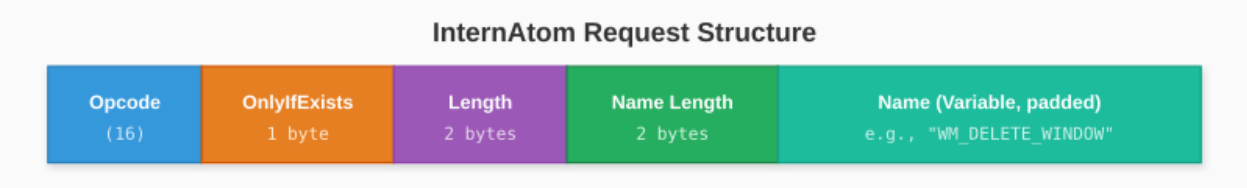


Figure 18: InternAtom Request Structure

- **OnlyIfExists**: If true, return None (0) if atom doesn't exist; if false, create it

This request generates a reply containing the atom value.

```
type Atom uint32

func (c *Connection) InternAtom(name string, onlyIfExists bool) (Atom, error) {
    nameBytes := []byte(name)
    nameLen := len(nameBytes)
    padding := (4 - (nameLen % 4)) % 4

    reqLen := 2 + (nameLen+padding)/4
    req := make([]byte, reqLen*4)

    req[0] = OpInternAtom
    if onlyIfExists {
        req[1] = 1
    } else {
        req[1] = 0
    }
    binary.LittleEndian.PutUint16(req[2:], uint16(reqLen))
    binary.LittleEndian.PutUint16(req[4:], uint16(nameLen))
    // Bytes 6-7 unused
    copy(req[8:], nameBytes)

    if _, err := c.conn.Write(req); err != nil {
        return 0, err
    }

    // Read the reply
    return c.readInternAtomReply()
}

func (c *Connection) readInternAtomReply() (Atom, error) {
    reply := make([]byte, 32)
    if _, err := io.ReadFull(c.conn, reply); err != nil {
        return 0, err
```

```
    }

    // Check for error (first byte = 0)
    if reply[0] == 0 {
        return 0, fmt.Errorf("InternAtom error: code %d", reply[1])
    }

    // Atom is at bytes 8-11
    atom := Atom(binary.LittleEndian.Uint32(reply[8:12]))
    return atom, nil
}
```

### Caching Atoms

Since atom values don't change, we should cache them:

```
type Connection struct {
    // ... other fields

    // Cached atoms
    atomWmProtocols    Atom
    atomWmDeleteWindow Atom
    atomNetWmName      Atom
    atomUtf8String     Atom
}

func (c *Connection) InitAtoms() error {
    var err error

    c.atomWmProtocols, err = c.InternAtom("WM_PROTOCOLS", false)
    if err != nil {
        return err
    }

    c.atomWmDeleteWindow, err = c.InternAtom("WM_DELETE_WINDOW", false)
    if err != nil {
        return err
    }

    c.atomNetWmName, err = c.InternAtom("_NET_WM_NAME", false)
    if err != nil {
        return err
    }

    c.atomUtf8String, err = c.InternAtom("UTF8_STRING", false)
    if err != nil {
        return err
    }
```

```
    return nil
}
```

Call `InitAtoms()` after the connection handshake.

## 7.3 Setting the Window Title

Window titles are set via properties. There are two ways:

1. **WM_NAME**: The classic method (Latin-1 encoding)
2. **_NET_WM_NAME**: The modern method (UTF-8 encoding)

We'll use `_NET_WM_NAME` for proper Unicode support.

### The ChangeProperty Request

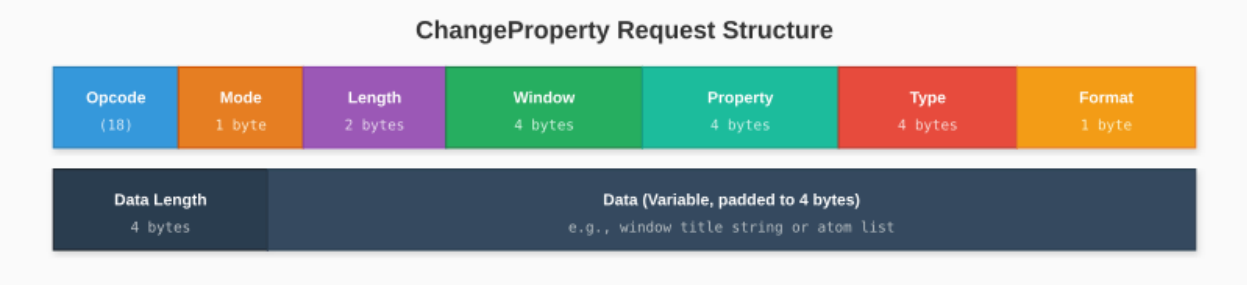


Figure 19: ChangeProperty Request Structure

- **Mode**: Replace (0), Prepend (1), or Append (2)
- **Property**: Atom identifying the property name
- **Type**: Atom identifying the data type
- **Format**: Bits per data element (8, 16, or 32)
- **Data Length**: Number of elements (not bytes!)

```
func (c *Connection) SetWindowTitle(window uint32, title string) error {
    titleBytes := []byte(title)
    titleLen := len(titleBytes)
    padding := (4 - (titleLen % 4)) % 4

    reqLen := 6 + (titleLen+padding)/4
    req := make([]byte, reqLen*4)

    req[0] = OpChangeProperty
    req[1] = 0  // Mode: Replace
    binary.LittleEndian.PutUint16(req[2:], uint16(reqLen))
    binary.LittleEndian.PutUint32(req[4:], window)
    binary.LittleEndian.PutUint32(req[8:], uint32(c.atomNetWmName))   // Property
    binary.LittleEndian.PutUint32(req[12:], uint32(c.atomUtf8String)) // Type
    req[16] = 8  // Format: 8 bits per element
    // Bytes 17-19 unused
```

```
    binary.LittleEndian.PutUint32(req[20:], uint32(titleLen))  // Data length
    copy(req[24:], titleBytes)

    _, err := c.conn.Write(req)
    return err
}
```

## 7.4 The Close Button Protocol

When you click a window's close button, the window manager needs to tell your application. This uses the WM_PROTOCOLS mechanism.

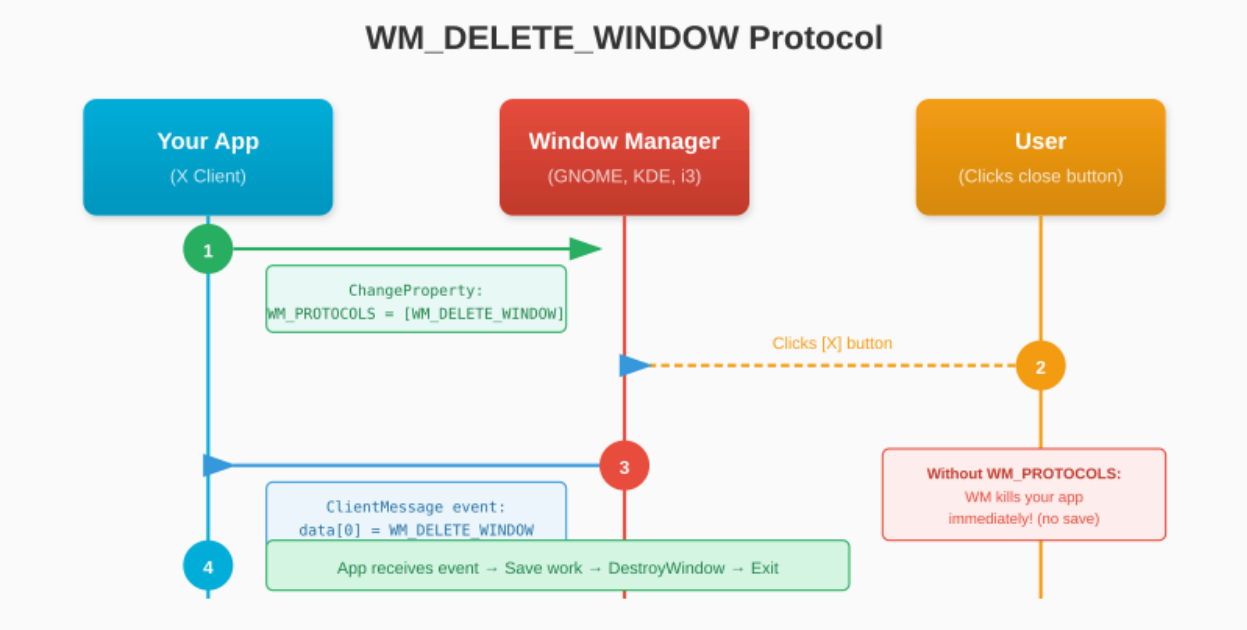


Figure 20: WM_DELETE_WINDOW Protocol

### How It Works

1. We tell the window manager we support `WM_DELETE_WINDOW`
2. When the user clicks close, we receive a `ClientMessage` event
3. We check if it's a delete request and respond appropriately

### Enabling the Close Button

```
func (c *Connection) EnableCloseButton(window uint32) error {
    // Set WM_PROTOCOLS to include WM_DELETE_WINDOW
    data := make([]byte, 4)
    binary.LittleEndian.PutUint32(data, uint32(c.atomWmDeleteWindow))

    reqLen := 6 + 1  // 6 header words + 1 data word
```

```
    req := make([]byte, reqLen*4)

    req[0] = OpChangeProperty
    req[1] = 0  // Replace
    binary.LittleEndian.PutUint16(req[2:], uint16(reqLen))
    binary.LittleEndian.PutUint32(req[4:], window)
    binary.LittleEndian.PutUint32(req[8:], uint32(c.atomWmProtocols))
    binary.LittleEndian.PutUint32(req[12:], 4)  // Type: ATOM
    req[16] = 32  // Format: 32 bits
    binary.LittleEndian.PutUint32(req[20:], 1)  // One atom
    copy(req[24:], data)

    _, err := c.conn.Write(req)
    return err
}
```

### Detecting the Close Event

When the close button is clicked, we receive a ClientMessage event:

```
type ClientMessageEvent struct {
    Window      uint32
    Format      uint8
    MessageType Atom
    Data        [20]byte
}

func IsDeleteWindowEvent(e ClientMessageEvent, deleteAtom Atom) bool {
    if e.Format != 32 {
        return false
    }
    // First 4 bytes of data contain the protocol atom
    dataAtom := binary.LittleEndian.Uint32(e.Data[:4])
    return Atom(dataAtom) == deleteAtom
}
```

We'll implement full event parsing in the next chapter.

## 7.5 EWMH and ICCCM Basics

X11 window managers follow two specifications:

### ICCCM (Inter-Client Communication Conventions Manual)

The original specification from 1988. Defines: - WM_NAME, WM_CLASS, WM_HINTS - WM_PROTOCOLS (including WM_DELETE_WINDOW) - Session management

### EWMH (Extended Window Manager Hints)

Modern extensions for desktop environments. Defines: - _NET_WM_NAME (UTF-8 titles) - _NET_WM_STATE (maximized, fullscreen, etc.) - _NET_WM_WINDOW_TYPE (normal, dialog, toolbar) - Taskbar integration

For basic applications, we need: - `_NET_WM_NAME`: Set the title - `WM_PROTOCOLS` + `WM_DELETE_WINDOW`: Handle close button

### Complete Window Setup

Here's the full window creation with title and close button:

```
func (c *Connection) CreateWindowFull(title string, x, y int16,
    width, height uint16) (uint32, error) {

    // Create the window
    windowID, err := c.CreateWindow(x, y, width, height)
    if err != nil {
        return 0, err
    }

    // Set the title
    if err := c.SetWindowTitle(windowID, title); err != nil {
        c.DestroyWindow(windowID)
        return 0, err
    }

    // Enable close button
    if err := c.EnableCloseButton(windowID); err != nil {
        c.DestroyWindow(windowID)
        return 0, err
    }

    return windowID, nil
}
```

### Testing

```
func main() {
    conn, err := x11.Connect()
    if err != nil {
        log.Fatal(err)
    }
    defer conn.Close()

    windowID, err := conn.CreateWindow(100, 100, 400, 300)
    if err != nil {
        log.Fatal(err)
```

```
    }

    conn.SetWindowTitle(windowID, "Hello from Glow!")
    conn.EnableCloseButton(windowID)
    conn.MapWindow(windowID)

    fmt.Println("Window created with title. Close button enabled.")
    fmt.Println("Press Ctrl+C to exit...")

    // Keep running (we'll add proper event handling next)
    select {}
}
```

The window now shows "Hello from Glow!" in the title bar!

---

**Key Takeaways:**

- Atoms are integer identifiers for strings, used throughout X11
- InternAtom converts strings to atoms (cache them for efficiency)
- Window titles use the `_NET_WM_NAME` property with UTF-8 encoding
- Close button handling requires WM_PROTOCOLS with WM_DELETE_WINDOW
- ICCCM and EWMH define standard conventions for window managers

Our windows have titles and can be closed properly (once we handle events). Speaking of which, let's tackle the event system next.

# Chapter 8: The Event System

Windows exist to interact with users. This chapter implements event handling - reading and parsing the stream of input events from the X server.

## 8.1 Event Types Overview

X11 defines over 30 event types. Here are the ones we care about:

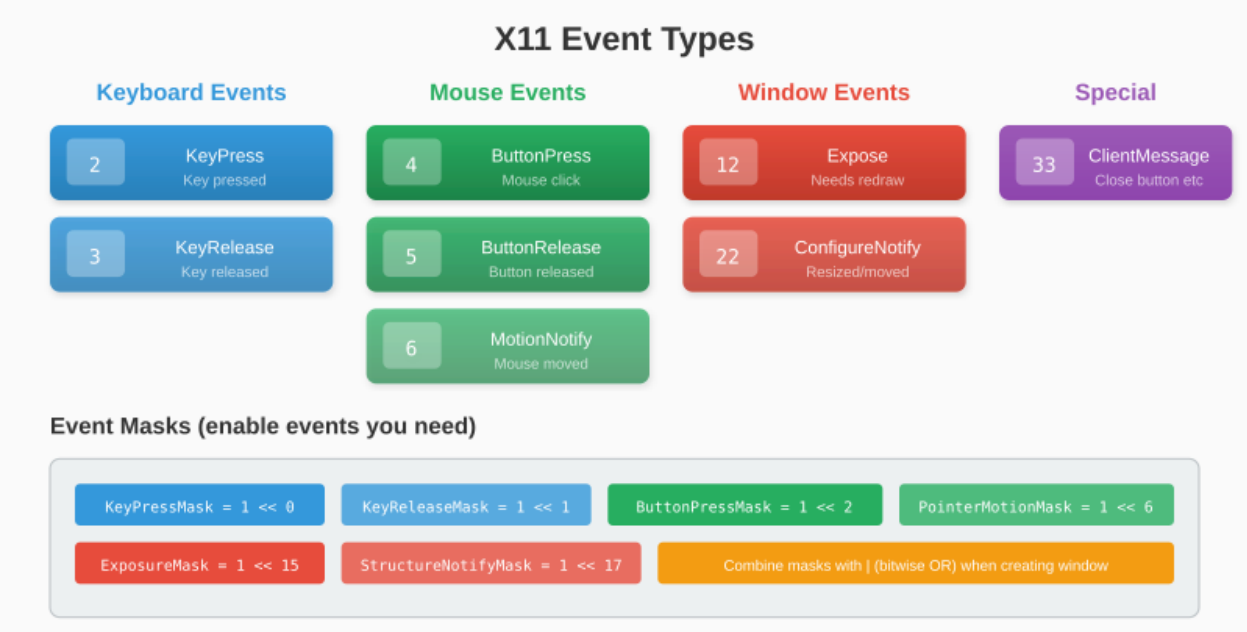


Figure 21: X11 Event Types

We requested these events when creating the window (via the event mask). The server only sends events we asked for.

## 8.2 Event Masks Revisited

Recall from window creation:

```
eventMask := uint32(
    KeyPressMask |        // 1 << 0
```

```
    KeyReleaseMask |      // 1 << 1
    ButtonPressMask |     // 1 << 2
    ButtonReleaseMask |   // 1 << 3
    PointerMotionMask |   // 1 << 6
    ExposureMask |        // 1 << 15
    StructureNotifyMask,  // 1 << 17
)
```

Each mask bit enables a category of events:

- **KeyPressMask/KeyReleaseMask**: Keyboard input
- **ButtonPressMask/ButtonReleaseMask**: Mouse button events
- **PointerMotionMask**: Mouse movement (can generate many events!)
- **ExposureMask**: Window exposed and needs redrawing
- **StructureNotifyMask**: Window geometry changes

**Performance Tip**: Don't enable `PointerMotionMask` unless you need continuous mouse tracking. It generates events for every pixel of movement.

## 8.3 Reading Events from the Socket

All X11 events are exactly 32 bytes. This makes reading straightforward:

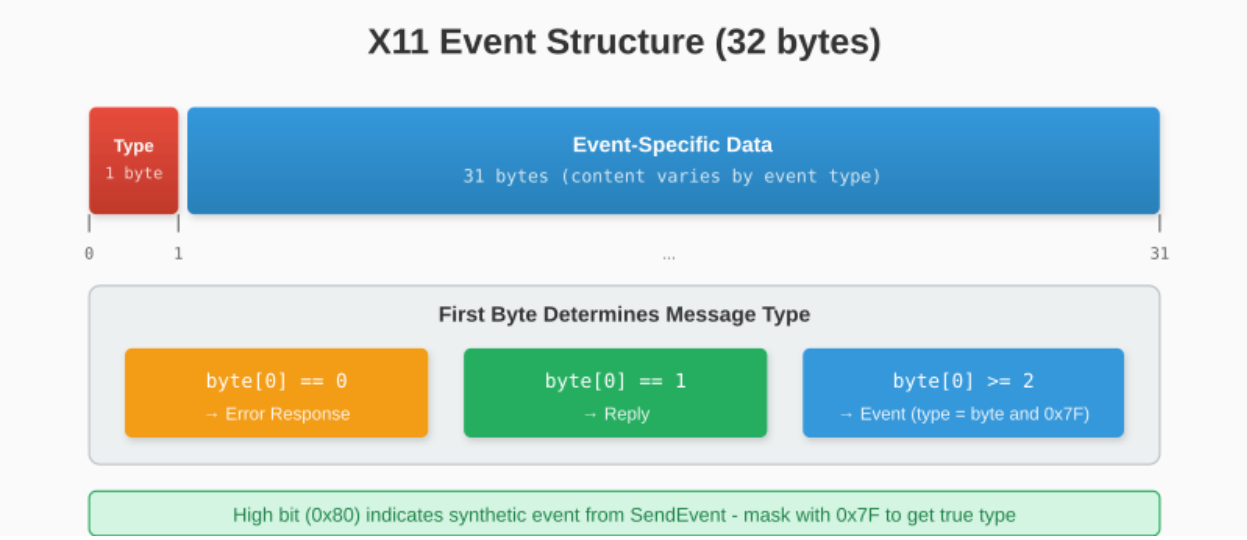


Figure 22: X11 Event Structure

```
func (c *Connection) NextEvent() (Event, error) {
    buf := make([]byte, 32)
    _, err := io.ReadFull(c.conn, buf)
    if err != nil {
        return nil, err
    }
```

```
    return c.parseEvent(buf)
}
```

The first byte identifies the event type:

```
func (c *Connection) parseEvent(buf []byte) (Event, error) {
    // High bit indicates synthetic event (from SendEvent)
    eventType := int(buf[0] & 0x7F)

    switch eventType {
    case 0:
        return c.parseError(buf)
    case 1:
        return c.parseReply(buf)
    case EventKeyPress, EventKeyRelease:
        return c.parseKeyEvent(buf, eventType)
    case EventButtonPress, EventButtonRelease:
        return c.parseButtonEvent(buf, eventType)
    case EventMotionNotify:
        return c.parseMotionEvent(buf)
    case EventExpose:
        return c.parseExposeEvent(buf)
    case EventConfigureNotify:
        return c.parseConfigureEvent(buf)
    case EventClientMessage:
        return c.parseClientMessage(buf)
    default:
        return UnknownEvent{Type: eventType, Data: buf}, nil
    }
}
```

## 8.4 Parsing Event Data

Each event type has a specific structure. Let's define our event types:

```
// Event is the interface for all events
type Event interface {
    Type() int
}
```

### Error Responses

When something goes wrong, byte 0 is 0:

```
type ErrorEvent struct {
    ErrorCode   uint8
    SequenceNum uint16
    BadValue    uint32
    MinorOpcode uint16
```

```
    MajorOpcode uint8
}

func (e ErrorEvent) Type() int { return 0 }

func (c *Connection) parseError(buf []byte) (Event, error) {
    return ErrorEvent{
        ErrorCode:   buf[1],
        SequenceNum: binary.LittleEndian.Uint16(buf[2:4]),
        BadValue:    binary.LittleEndian.Uint32(buf[4:8]),
        MinorOpcode: binary.LittleEndian.Uint16(buf[8:10]),
        MajorOpcode: buf[10],
    }, nil
}
```

## 8.5 Keyboard Events

Keyboard events carry the key code and modifier state:

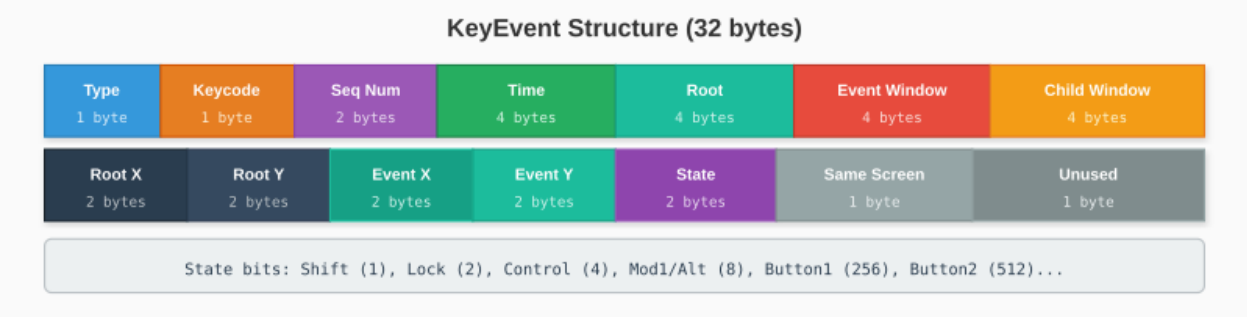


Figure 23: KeyEvent Structure

```
type KeyEvent struct {
    EventType int
    Keycode   uint8
    State     uint16  // Modifier keys (Shift, Ctrl, etc.)
    X, Y      int16   // Position relative to window
    RootX     int16   // Position relative to root
    RootY     int16
}

func (e KeyEvent) Type() int { return e.EventType }

func (c *Connection) parseKeyEvent(buf []byte, eventType int) (Event, error) {
    return KeyEvent{
        EventType: eventType,
        Keycode:   buf[1],
        State:     binary.LittleEndian.Uint16(buf[28:30]),
```

```
        X:          int16(binary.LittleEndian.Uint16(buf[24:26])),
        Y:          int16(binary.LittleEndian.Uint16(buf[26:28])),
        RootX:      int16(binary.LittleEndian.Uint16(buf[20:22])),
        RootY:      int16(binary.LittleEndian.Uint16(buf[22:24])),
    }, nil
}
```

### Key Codes vs. Key Symbols

The `Keycode` is a hardware-specific number. The same physical key produces the same keycode regardless of keyboard layout.

Common keycodes (may vary by system):

```
const (
    KeyEscape = 9
    KeySpace  = 65
    KeyEnter  = 36

    Key1 = 10
    Key2 = 11
    // ...

    KeyA = 38
    KeyB = 56
    KeyC = 54
    // ...

    KeyUp    = 111
    KeyDown  = 116
    KeyLeft  = 113
    KeyRight = 114
)
```

For proper keyboard handling, you'd use XKB (X Keyboard Extension) to convert keycodes to key symbols. For games, raw keycodes often suffice.

### Modifier State

The `State` field contains modifier key flags:

```
const (
    ShiftMask   = 1 << 0
    LockMask    = 1 << 1  // Caps Lock
    ControlMask = 1 << 2
    Mod1Mask    = 1 << 3  // Usually Alt
    Mod2Mask    = 1 << 4  // Usually Num Lock
    Mod3Mask    = 1 << 5
    Mod4Mask    = 1 << 6  // Usually Super/Windows
```

```
    Mod5Mask    = 1 << 7
)
```

Check modifiers:

```
if event.State & ShiftMask != 0 {
    // Shift is held
}
```

## 8.6 Mouse Events

Button events have a similar structure to key events:

```
type ButtonEvent struct {
    EventType int
    Button    uint8   // 1=left, 2=middle, 3=right, 4=wheel up, 5=wheel down
    State     uint16
    X, Y      int16
    RootX     int16
    RootY     int16
}

func (e ButtonEvent) Type() int { return e.EventType }

func (c *Connection) parseButtonEvent(buf []byte, eventType int) (Event, error) {
    return ButtonEvent{
        EventType: eventType,
        Button:    buf[1],
        State:     binary.LittleEndian.Uint16(buf[28:30]),
        X:         int16(binary.LittleEndian.Uint16(buf[24:26])),
        Y:         int16(binary.LittleEndian.Uint16(buf[26:28])),
        RootX:     int16(binary.LittleEndian.Uint16(buf[20:22])),
        RootY:     int16(binary.LittleEndian.Uint16(buf[22:24])),
    }, nil
}
```

### Mouse Motion

Motion events report mouse position:

```
type MotionEvent struct {
    X, Y  int16
    RootX int16
    RootY int16
    State uint16  // Which buttons are held
}

func (e MotionEvent) Type() int { return EventMotionNotify }
```

```
func (c *Connection) parseMotionEvent(buf []byte) (Event, error) {
    return MotionEvent{
        X:     int16(binary.LittleEndian.Uint16(buf[24:26])),
        Y:     int16(binary.LittleEndian.Uint16(buf[26:28])),
        RootX: int16(binary.LittleEndian.Uint16(buf[20:22])),
        RootY: int16(binary.LittleEndian.Uint16(buf[22:24])),
        State: binary.LittleEndian.Uint16(buf[28:30]),
    }, nil
}
```

## 8.7 Window Events

### Expose Events

Expose tells us part of the window needs redrawing:

```
type ExposeEvent struct {
    Window uint32
    X, Y   uint16
    Width  uint16
    Height uint16
    Count  uint16  // Number of following Expose events
}

func (e ExposeEvent) Type() int { return EventExpose }

func (c *Connection) parseExposeEvent(buf []byte) (Event, error) {
    return ExposeEvent{
        Window: binary.LittleEndian.Uint32(buf[4:8]),
        X:      binary.LittleEndian.Uint16(buf[8:10]),
        Y:      binary.LittleEndian.Uint16(buf[10:12]),
        Width:  binary.LittleEndian.Uint16(buf[12:14]),
        Height: binary.LittleEndian.Uint16(buf[14:16]),
        Count:  binary.LittleEndian.Uint16(buf[16:18]),
    }, nil
}
```

When `Count > 0`, more Expose events follow. You might wait until `Count == 0` before redrawing to avoid redundant work.

### Configure Events

ConfigureNotify reports window geometry changes:

```
type ConfigureEvent struct {
    Window uint32
    X, Y   int16
    Width  uint16
    Height uint16
```

```
}

func (e ConfigureEvent) Type() int { return EventConfigureNotify }

func (c *Connection) parseConfigureEvent(buf []byte) (Event, error) {
    return ConfigureEvent{
        Window: binary.LittleEndian.Uint32(buf[4:8]),
        X:      int16(binary.LittleEndian.Uint16(buf[16:18])),
        Y:      int16(binary.LittleEndian.Uint16(buf[18:20])),
        Width:  binary.LittleEndian.Uint16(buf[20:22]),
        Height: binary.LittleEndian.Uint16(buf[22:24]),
    }, nil
}
```

### Client Messages

These carry inter-client communication, including close button clicks:

```
type ClientMessageEvent struct {
    Window      uint32
    Format      uint8
    MessageType uint32
    Data        [20]byte
}

func (e ClientMessageEvent) Type() int { return EventClientMessage }

func (c *Connection) parseClientMessage(buf []byte) (Event, error) {
    e := ClientMessageEvent{
        Window:      binary.LittleEndian.Uint32(buf[4:8]),
        Format:      buf[1],
        MessageType: binary.LittleEndian.Uint32(buf[8:12]),
    }
    copy(e.Data[:], buf[12:32])
    return e, nil
}
```

### Checking for Close Button

```
func IsDeleteWindowEvent(e ClientMessageEvent, wmProtocols, wmDeleteWindow Atom) bool {
    if Atom(e.MessageType) != wmProtocols {
        return false
    }
    if e.Format != 32 {
        return false
    }
    protocol := Atom(binary.LittleEndian.Uint32(e.Data[:4]))
```

```
    return protocol == wmDeleteWindow
}
```

### Complete Event Loop

```
func main() {
    conn, _ := x11.Connect()
    defer conn.Close()

    windowID, _ := conn.CreateWindow(100, 100, 400, 300)
    conn.SetWindowTitle(windowID, "Event Demo")
    conn.EnableCloseButton(windowID)
    conn.MapWindow(windowID)

    running := true
    for running {
        event, err := conn.NextEvent()
        if err != nil {
            break
        }

        switch e := event.(type) {
        case x11.KeyEvent:
            if e.EventType == x11.EventKeyPress {
                fmt.Printf("Key pressed: %d\n", e.Keycode)
                if e.Keycode == x11.KeyEscape {
                    running = false
                }
            }

        case x11.ButtonEvent:
            fmt.Printf("Button %d at (%d, %d)\n", e.Button, e.X, e.Y)

        case x11.MotionEvent:
            fmt.Printf("Mouse at (%d, %d)\n", e.X, e.Y)

        case x11.ExposeEvent:
            fmt.Println("Window needs redraw")

        case x11.ClientMessageEvent:
            if x11.IsDeleteWindowEvent(e, conn.AtomWmProtocols(),
                conn.AtomWmDeleteWindow()) {
                fmt.Println("Close button clicked")
                running = false
            }
        }
    }
```

```
    conn.DestroyWindow(windowID)
}
```

---

**Key Takeaways:**

- All X11 events are 32 bytes with the type in byte 0
- Key events provide hardware keycodes and modifier state
- Button events include position and which button
- Motion events report mouse position continuously
- Expose events signal when to redraw
- ClientMessage carries close button notifications
- `NextEvent()` blocks until an event arrives

The event loop works, but it's blocking. For games and responsive applications, we need non-blocking event handling. That's next.

# Chapter 9: Non-Blocking Events

Our event loop blocks on `NextEvent()`, waiting for user input. But games need to update continuously - physics simulations, animations, and AI don't stop while waiting for keypresses. This chapter implements non-blocking event handling using Go's concurrency primitives.

## 9.1 The Problem with Blocking

Consider a simple game loop:

```
for running {
    event := conn.NextEvent()  // BLOCKS here!
    handleEvent(event)
    updateGame()
    render()
}
```

If the user doesn't press anything, `NextEvent()` waits forever. The game freezes.

We need two things happening simultaneously: 1. Reading events from the X server 2. Running the game loop

## 9.2 Goroutines and Channels

Go's goroutines let us run concurrent tasks. Channels provide safe communication between them.

The event goroutine continuously reads from the socket and sends events to a channel. The main goroutine checks the channel without blocking.

## 9.3 Building an Event Queue

First, let's add an event channel to our Window type:

```
type Window struct {
    conn     *Connection
    windowID uint32
    gcID     uint32

    // Event handling
    eventChan chan Event
```

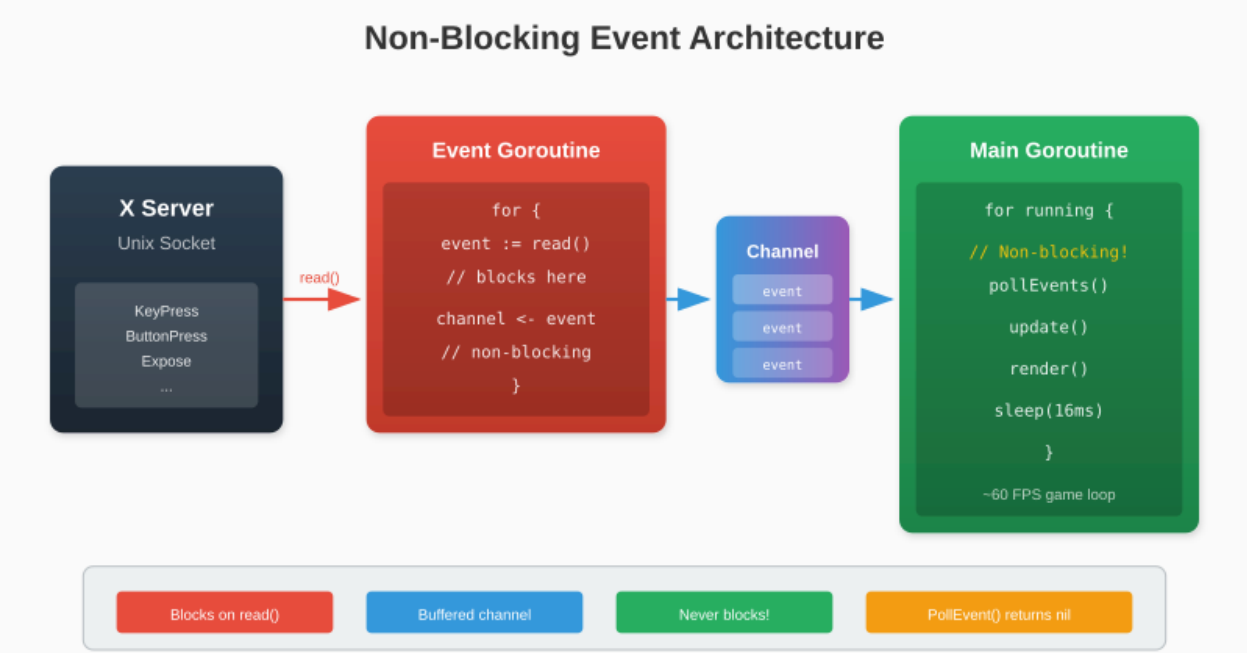


Figure 24: Non-Blocking Event Architecture

```
    quitChan  chan struct{}

    closed bool
}
```

- `eventChan`: Buffered channel for incoming events
- `quitChan`: Signals the event goroutine to stop

### Starting the Event Goroutine

```
func NewWindow(conn *Connection, title string, width, height int) (*Window, error) {
    windowID, err := conn.CreateWindow(100, 100, uint16(width), uint16(height))
    if err != nil {
        return nil, err
    }

    conn.SetWindowTitle(windowID, title)
    conn.EnableCloseButton(windowID)

    gcID, err := conn.CreateGC(windowID)
    if err != nil {
        conn.DestroyWindow(windowID)
        return nil, err
    }
```

```
    conn.MapWindow(windowID)

    w := &Window{
        conn:      conn,
        windowID:  windowID,
        gcID:      gcID,
        eventChan: make(chan Event, 256),  // Buffer 256 events
        quitChan:  make(chan struct{}),
    }

    // Start event reader
    go w.pollEvents()

    return w, nil
}
```

### The Event Goroutine

```
func (w *Window) pollEvents() {
    for {
        select {
        case <-w.quitChan:
            return  // Stop when window closes

        default:
            event, err := w.conn.NextEvent()
            if err != nil {
                continue  // Connection error, keep trying
            }

            // Try to send, but don't block if channel is full
            select {
            case w.eventChan <- event:
                // Event queued
            case <-w.quitChan:
                return
            default:
                // Channel full, drop event (shouldn't happen often)
            }
        }
    }
}
```

**Note**: The inner `select` with `default` prevents blocking if the event channel is full. Dropping events is better than deadlocking.

## 9.4 Polling vs Waiting

Now we implement two methods for the main goroutine:

### PollEvent (Non-Blocking)

Returns immediately with an event or nil:

```
func (w *Window) PollEvent() Event {
    select {
    case e := <-w.eventChan:
        return e
    default:
        return nil  // No events waiting
    }
}
```

### WaitEvent (Blocking)

Blocks until an event arrives:

```
func (w *Window) WaitEvent() Event {
    return <-w.eventChan
}
```

### Game Loop Pattern

```
for running {
    // Process all pending events
    for {
        event := win.PollEvent()
        if event == nil {
            break  // No more events
        }
        handleEvent(event)
    }

    // Update game state
    updateGame(deltaTime)

    // Render
    render()

    // Cap frame rate
    time.Sleep(16 * time.Millisecond)  // ~60 FPS
}
```

This pattern: 1. Drains all pending events (responsive input) 2. Updates game logic 3. Renders the frame 4. Sleeps to maintain frame rate

## 9.5 Thread Safety Considerations

### Socket Access

Our event goroutine reads from the socket while the main goroutine writes (sending draw commands). Is this safe?

On Unix, concurrent read and write on the same socket is safe - they use different buffers. However, concurrent writes or concurrent reads would be unsafe.

Since only the event goroutine reads and only the main goroutine writes, we're fine.

### Shared State

If events modify shared state, we'd need synchronization:

```
type Window struct {
    // ...
    mu     sync.Mutex
    width  int
    height int
}

func (w *Window) pollEvents() {
    // ...
    if e, ok := event.(ConfigureEvent); ok {
        w.mu.Lock()
        w.width = int(e.Width)
        w.height = int(e.Height)
        w.mu.Unlock()
    }
    // ...
}

func (w *Window) Width() int {
    w.mu.Lock()
    defer w.mu.Unlock()
    return w.width
}
```

For our library, we'll keep it simple: events are immutable, and dimension queries happen infrequently.

### Closing the Window

Properly shutting down requires coordination:

```
func (w *Window) Close() {
    if w.closed {
        return
    }
```

```
    w.closed = true

    // Signal event goroutine to stop
    close(w.quitChan)

    // Clean up X11 resources
    w.conn.FreeGC(w.gcID)
    w.conn.DestroyWindow(w.windowID)
}
```

The `close(w.quitChan)` causes all receives on `quitChan` to return immediately, signaling the goroutine to exit.

### Complete Implementation

```
// internal/window.go
package x11

type Window struct {
    conn     *Connection
    windowID uint32
    gcID     uint32
    width    int
    height   int

    eventChan chan Event
    quitChan  chan struct{}
    closed    bool
}

func NewWindow(conn *Connection, title string, width, height int) (*Window, error) {
    windowID, err := conn.CreateWindow(100, 100, uint16(width), uint16(height))
    if err != nil {
        return nil, err
    }

    if err := conn.SetWindowTitle(windowID, title); err != nil {
        conn.DestroyWindow(windowID)
        return nil, err
    }

    if err := conn.EnableCloseButton(windowID); err != nil {
        conn.DestroyWindow(windowID)
        return nil, err
    }

    gcID, err := conn.CreateGC(windowID)
```

```
    if err != nil {
        conn.DestroyWindow(windowID)
        return nil, err
    }

    if err := conn.MapWindow(windowID); err != nil {
        conn.FreeGC(gcID)
        conn.DestroyWindow(windowID)
        return nil, err
    }

    w := &Window{
        conn:      conn,
        windowID:  windowID,
        gcID:      gcID,
        width:     width,
        height:    height,
        eventChan: make(chan Event, 256),
        quitChan:  make(chan struct{}),
    }

    go w.pollEvents()

    return w, nil
}

func (w *Window) pollEvents() {
    for {
        select {
        case <-w.quitChan:
            return
        default:
            event, err := w.conn.NextEvent()
            if err != nil {
                continue
            }

            select {
            case w.eventChan <- event:
            case <-w.quitChan:
                return
            default:
                // Drop event if buffer full
            }
        }
    }
}
```

```
func (w *Window) PollEvent() Event {
    select {
    case e := <-w.eventChan:
        return e
    default:
        return nil
    }
}

func (w *Window) WaitEvent() Event {
    return <-w.eventChan
}

func (w *Window) Close() {
    if w.closed {
        return
    }
    w.closed = true

    close(w.quitChan)
    w.conn.FreeGC(w.gcID)
    w.conn.DestroyWindow(w.windowID)
}

func (w *Window) Width() int  { return w.width }
func (w *Window) Height() int { return w.height }
```

### Example: Responsive Input

```
func main() {
    conn, _ := x11.Connect()
    defer conn.Close()

    win, _ := x11.NewWindow(conn, "Non-Blocking Demo", 400, 300)
    defer win.Close()

    running := true
    lastTime := time.Now()

    for running {
        // Handle all pending events
        for {
            event := win.PollEvent()
            if event == nil {
                break
            }
```

```
            switch e := event.(type) {
            case x11.KeyEvent:
                if e.EventType == x11.EventKeyPress {
                    fmt.Printf("Key: %d\n", e.Keycode)
                    if e.Keycode == x11.KeyEscape {
                        running = false
                    }
                }
            case x11.ClientMessageEvent:
                if x11.IsDeleteWindowEvent(e, ...) {
                    running = false
                }
            }
        }

        // Update (even without input)
        now := time.Now()
        dt := now.Sub(lastTime).Seconds()
        lastTime = now

        fmt.Printf("Frame time: %.2fms\n", dt*1000)

        // Maintain ~60 FPS
        time.Sleep(16 * time.Millisecond)
    }
}
```

The game loop runs continuously, processing events when available but never blocking.

---

**Key Takeaways:**

- Blocking event reads freeze the application
- A goroutine reads events into a buffered channel
- `PollEvent()` returns immediately (non-blocking)
- `WaitEvent()` blocks until an event arrives
- Proper shutdown requires signaling the goroutine via a quit channel
- Concurrent socket read/write is safe; concurrent reads or writes are not

With responsive event handling, we're ready to draw graphics. Next chapter: the Graphics Context.

# Chapter 10: The Graphics Context

We can create windows and handle events. Now it's time to draw. This chapter introduces the Graphics Context (GC) - X11's mechanism for managing drawing state.

## 10.1 What is a GC?

A Graphics Context stores drawing parameters: foreground color, background color, line width, font, and many other properties. Instead of passing these with every draw operation, you create a GC once and reference it.

Think of a GC like a painter's palette - it holds the current brush, color, and style. You switch palettes to change how things are drawn.

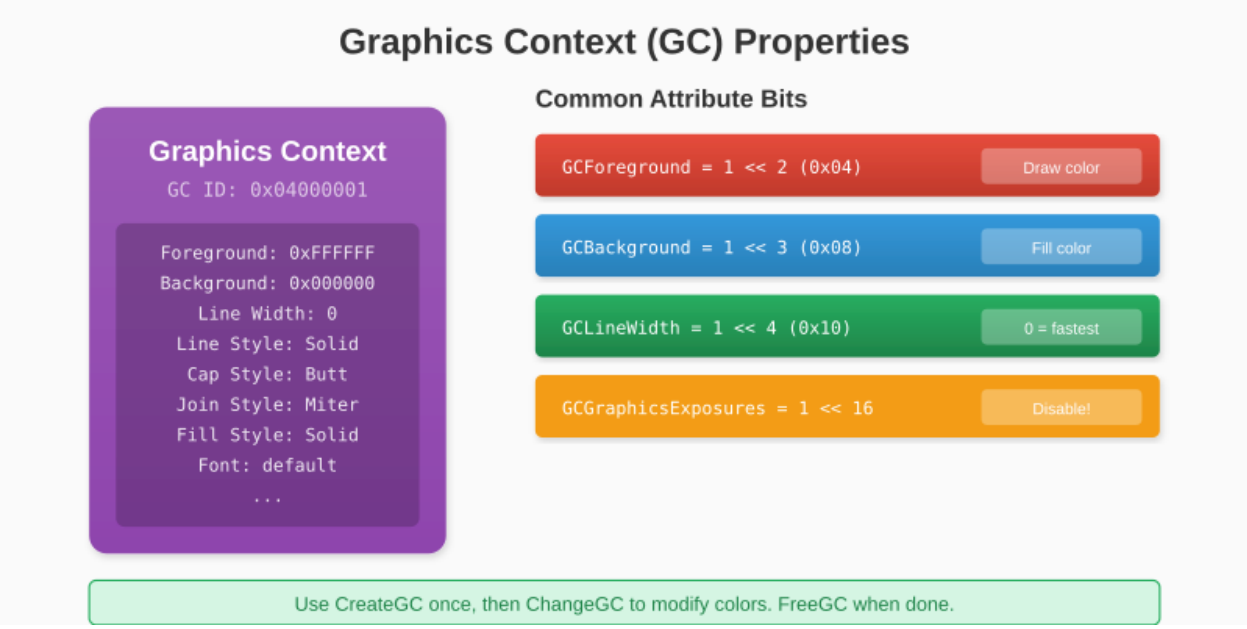


Figure 25: Graphics Context Properties

## 10.2 Creating a Graphics Context

The CreateGC request:

- **GC ID**: We generate this

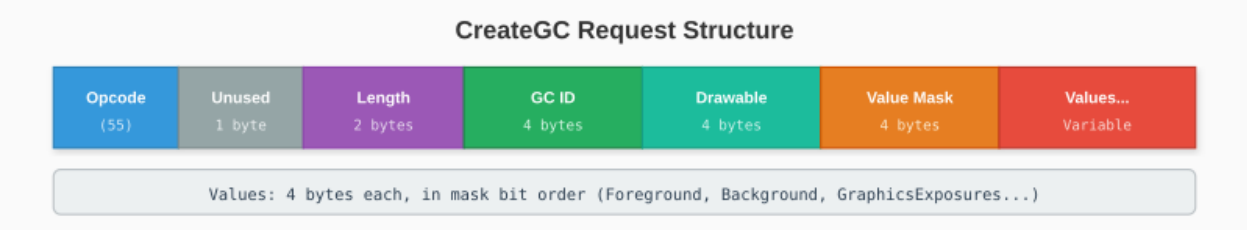


Figure 26: CreateGC Request Structure

- **Drawable**: Window or pixmap the GC is compatible with
- **Value Mask**: Which properties we're setting
- **Values**: Property values in mask bit order

```go
func (c *Connection) CreateGC(drawable uint32) (uint32, error) {
    gcID := c.GenerateID()

    // Set foreground, background, and disable graphics exposures
    valueMask := uint32(GCForeground | GCBackground | GCGraphicsExposures)
    valueCount := 3

    reqLen := 4 + valueCount
    req := make([]byte, reqLen*4)

    req[0] = OpCreateGC
    req[1] = 0
    binary.LittleEndian.PutUint16(req[2:], uint16(reqLen))
    binary.LittleEndian.PutUint32(req[4:], gcID)
    binary.LittleEndian.PutUint32(req[8:], drawable)
    binary.LittleEndian.PutUint32(req[12:], valueMask)

    // Values in mask bit order
    binary.LittleEndian.PutUint32(req[16:], 0xFFFFFF)  // Foreground: white
    binary.LittleEndian.PutUint32(req[20:], 0x000000)  // Background: black
    binary.LittleEndian.PutUint32(req[24:], 0)         // GraphicsExposures: off

    if _, err := c.conn.Write(req); err != nil {
        return 0, err
    }

    return gcID, nil
}
```

## 10.3 GC Attributes

Here are the attribute bits and their meanings:

```
const (
    GCFunction          = 1 << 0   // Drawing operation (copy, xor, etc.)
    GCPlaneMask         = 1 << 1   // Which bit planes to modify
    GCForeground        = 1 << 2   // Foreground color
    GCBackground        = 1 << 3   // Background color
    GCLineWidth         = 1 << 4   // Line thickness (0 = fastest)
    GCLineStyle         = 1 << 5   // Solid, dashed, double-dashed
    GCCapStyle          = 1 << 6   // How lines end
    GCJoinStyle         = 1 << 7   // How lines connect
    GCFillStyle         = 1 << 8   // Solid, tiled, stippled
    GCFillRule          = 1 << 9   // Even-odd or winding
    GCTile              = 1 << 10  // Tile pixmap for fills
    GCStipple           = 1 << 11  // Stipple bitmap
    GCTileStipXOrigin   = 1 << 12  // Tile/stipple X origin
    GCTileStipYOrigin   = 1 << 13  // Tile/stipple Y origin
    GCFont              = 1 << 14  // Font for text
    GCSubwindowMode     = 1 << 15  // Clip to subwindows?
    GCGraphicsExposures = 1 << 16  // Send Expose on copy?
    GCClipXOrigin       = 1 << 17  // Clip mask X origin
    GCClipYOrigin       = 1 << 18  // Clip mask Y origin
    GCClipMask          = 1 << 19  // Clip mask pixmap
    GCDashOffset        = 1 << 20  // Dash pattern start
    GCDashList          = 1 << 21  // Dash pattern
    GCArcMode           = 1 << 22  // Pie slice or chord
)
```

For software rendering, we mainly care about `GCForeground` and `GCBackground`. We disable `GCGraphicsExposures` to avoid receiving Expose events when copying pixels.

## 10.4 Foreground and Background Colors

Colors in X11 are 24-bit RGB values packed into 32 bits:

```
0x00RRGGBB

Examples:
  White: 0xFFFFFF
  Black: 0x000000
  Red:   0xFF0000
  Green: 0x00FF00
  Blue:  0x0000FF
```

To change the foreground color, use ChangeGC:

```
func (c *Connection) ChangeGC(gc uint32, foreground uint32) error {
    req := make([]byte, 16)

    req[0] = OpChangeGC
    req[1] = 0
```

```
    binary.LittleEndian.PutUint16(req[2:], 4)  // Length
    binary.LittleEndian.PutUint32(req[4:], gc)
    binary.LittleEndian.PutUint32(req[8:], GCForeground)  // Mask
    binary.LittleEndian.PutUint32(req[12:], foreground)   // Value

    _, err := c.conn.Write(req)
    return err
}
```

## 10.5 Freeing Resources

Always free GCs when done:

```
func (c *Connection) FreeGC(gcID uint32) error {
    req := make([]byte, 8)
    req[0] = OpFreeGC
    req[1] = 0
    binary.LittleEndian.PutUint16(req[2:], 2)
    binary.LittleEndian.PutUint32(req[4:], gcID)

    _, err := c.conn.Write(req)
    return err
}
```

## 10.6 The PutImage Request

PutImage is how we send pixel data to a window. It's the foundation of software rendering.

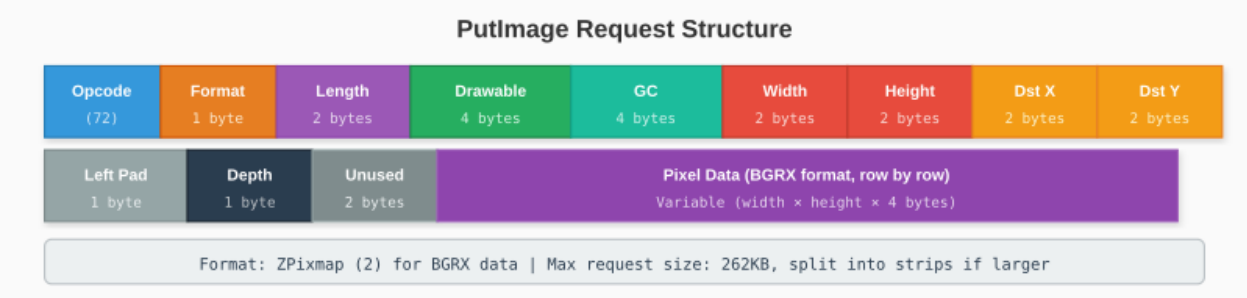


Figure 27: PutImage Request Structure

### Image Formats

X11 supports three formats:

- **Bitmap (0)**: 1-bit images
- **XYPixmap (1)**: Planar format (rarely used)
- **ZPixmap (2)**: Packed pixels - what we use

ZPixmap stores pixels as contiguous bytes, which matches how modern hardware and our framebuffer work.

### Pixel Byte Order

For 24-bit depth with 32 bits per pixel (most common):

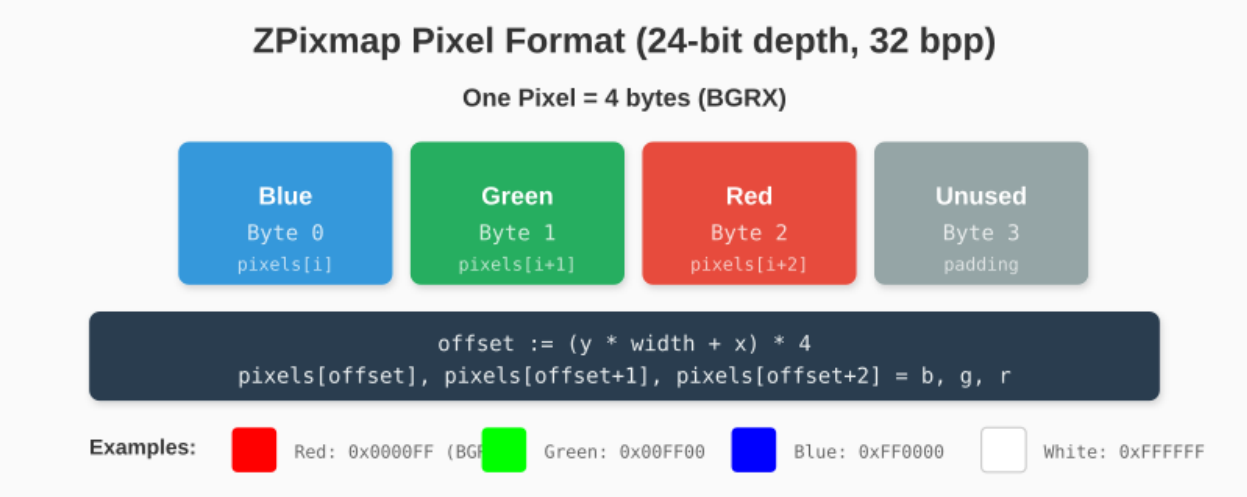


Figure 28: ZPixmap Pixel Format

This BGRX format matches what X11 expects on little-endian systems.

```
func setPixel(pixels []byte, width, x, y int, r, g, b uint8) {
    offset := (y*width + x) * 4
    pixels[offset] = b      // Blue
    pixels[offset+1] = g    // Green
    pixels[offset+2] = r    // Red
    pixels[offset+3] = 0    // Padding
}
```

## 10.7 Implementing PutImage

Here's the basic implementation:

```
func (c *Connection) PutImage(drawable, gc uint32, width, height uint16,
    dstX, dstY int16, depth uint8, data []byte) error {

    dataLen := len(data)
    padding := (4 - (dataLen % 4)) % 4

    reqLen := 6 + (dataLen+padding)/4
    req := make([]byte, reqLen*4)

    req[0] = OpPutImage
    req[1] = ImageFormatZPixmap
```

```
    binary.LittleEndian.PutUint16(req[2:], uint16(reqLen))
    binary.LittleEndian.PutUint32(req[4:], drawable)
    binary.LittleEndian.PutUint32(req[8:], gc)
    binary.LittleEndian.PutUint16(req[12:], width)
    binary.LittleEndian.PutUint16(req[14:], height)
    binary.LittleEndian.PutUint16(req[16:], uint16(dstX))
    binary.LittleEndian.PutUint16(req[18:], uint16(dstY))
    req[20] = 0       // Left pad
    req[21] = depth
    // Bytes 22-23 unused

    copy(req[24:], data)

    _, err := c.conn.Write(req)
    return err
}
```

## 10.8 The Request Size Limit

**Critical Issue**: X11 requests have a 16-bit length field, limiting each request to $65535 \times 4 = 262{,}140$ bytes.

For an 800×600 window at 4 bytes per pixel:

```
800 × 600 × 4 = 1,920,000 bytes
```

This exceeds the limit by 7×!

### Solution: Split into Strips

Send the image in horizontal strips:

```
func (c *Connection) PutImage(drawable, gc uint32, width, height uint16,
    dstX, dstY int16, depth uint8, data []byte) error {

    bytesPerPixel := 4
    rowBytes := int(width) * bytesPerPixel

    // Max ~262KB per request, minus 24-byte header
    maxDataBytes := 262140 - 24

    rowsPerRequest := maxDataBytes / rowBytes
    if rowsPerRequest > int(height) {
        rowsPerRequest = int(height)
    }

    // Send in strips
    for y := 0; y < int(height); y += rowsPerRequest {
        stripHeight := rowsPerRequest
```

```
        if y+stripHeight > int(height) {
            stripHeight = int(height) - y
        }

        stripData := data[y*rowBytes : (y+stripHeight)*rowBytes]

        err := c.putImageStrip(drawable, gc, width, uint16(stripHeight),
            dstX, dstY+int16(y), depth, stripData)
        if err != nil {
            return err
        }
    }

    return nil
}
```

For 800×600: - Row size: 800 × 4 = 3,200 bytes - Max rows per request: 262,116 / 3,200  81 rows - Number of requests: 600 / 81  8 requests

This splitting is invisible to the caller - they still pass the full image.

## 10.9 Testing Drawing

```
func main() {
    conn, _ := x11.Connect()
    defer conn.Close()

    windowID, _ := conn.CreateWindow(100, 100, 400, 300)
    conn.SetWindowTitle(windowID, "Drawing Test")
    conn.MapWindow(windowID)

    gcID, _ := conn.CreateGC(windowID)
    defer conn.FreeGC(gcID)

    // Create pixel buffer
    width, height := 400, 300
    pixels := make([]byte, width*height*4)

    // Fill with red
    for y := 0; y < height; y++ {
        for x := 0; x < width; x++ {
            offset := (y*width + x) * 4
            pixels[offset] = 0     // Blue
            pixels[offset+1] = 0   // Green
            pixels[offset+2] = 255 // Red
            pixels[offset+3] = 0
        }
```

```
    }

    // Send to window
    conn.PutImage(windowID, gcID, uint16(width), uint16(height),
        0, 0, conn.RootDepth, pixels)

    fmt.Println("Red window displayed. Press Ctrl+C to exit.")
    select {}  // Wait forever
}
```

You should see a solid red window!

---

**Key Takeaways:**

- Graphics Contexts store drawing state (colors, line styles, etc.)
- CreateGC creates a new context; FreeGC releases it
- PutImage sends pixel data using ZPixmap format
- Pixels are BGRX (Blue, Green, Red, padding) on little-endian systems
- Large images must be split into strips due to request size limits
- The 262KB limit applies to all X11 requests

With PutImage working, we can display any pixels we want. Next, we'll build a proper framebuffer with drawing primitives.

# Chapter 11: The Framebuffer

We can send pixels to X11, but we need a convenient way to manipulate them. This chapter builds a framebuffer - an in-memory pixel array with drawing operations.

## 11.1 Software Rendering Basics

**Software rendering** means the CPU calculates every pixel. This contrasts with **hardware rendering** where the GPU does the work.

Software rendering is: - **Simpler**: No GPU API to learn - **Portable**: Works everywhere - **Flexible**: Full control over every pixel - **Slower**: CPU can't match GPU parallelism

For 2D games and applications at reasonable resolutions, software rendering is plenty fast.

### The Rendering Pipeline

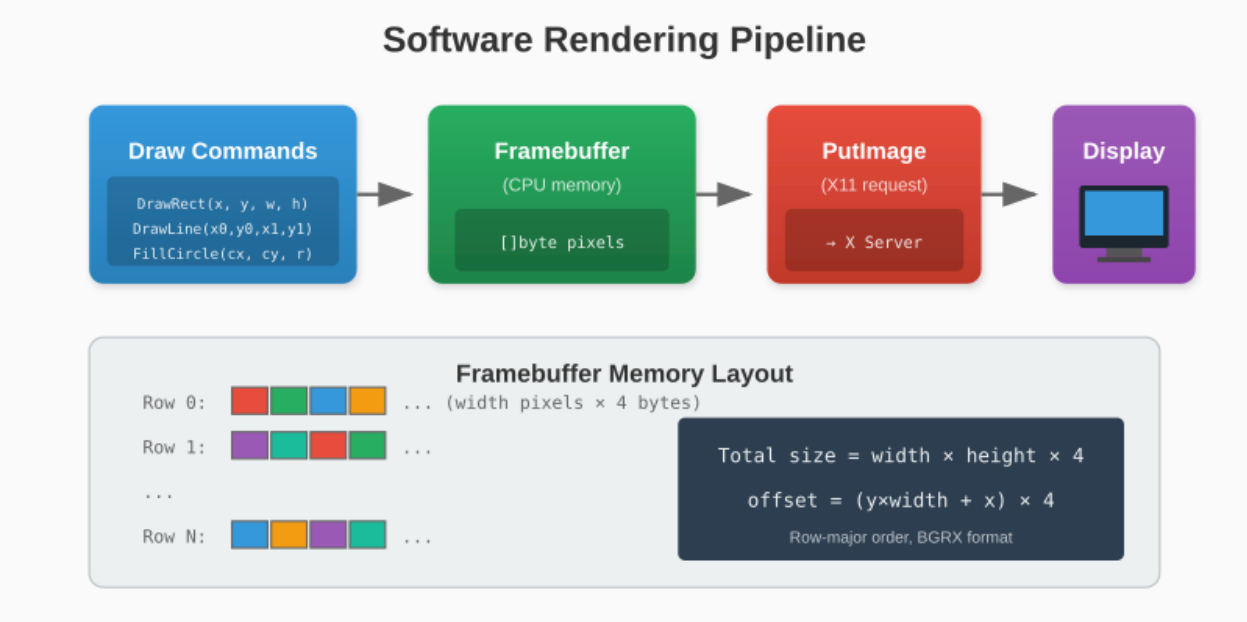


Figure 29: Software Rendering Pipeline

## 11.2 Designing the Framebuffer

Our framebuffer needs: - A byte array for pixels - Width and height - Methods for pixel manipulation

```
// internal/x11/framebuffer.go
package x11

type Framebuffer struct {
    Width  int
    Height int
    Pixels []byte  // BGRX format, 4 bytes per pixel
}

func NewFramebuffer(width, height int) *Framebuffer {
    return &Framebuffer{
        Width:  width,
        Height: height,
        Pixels: make([]byte, width*height*4),
    }
}
```

The pixel array is `width × height × 4` bytes, stored in row-major order (row 0 first, then row 1, etc.).

## 11.3 Setting Pixels

The fundamental operation:

```
func (fb *Framebuffer) SetPixel(x, y int, r, g, b uint8) {
    // Bounds checking
    if x < 0 || x >= fb.Width || y < 0 || y >= fb.Height {
        return
    }

    offset := (y*fb.Width + x) * 4
    fb.Pixels[offset] = b      // Blue
    fb.Pixels[offset+1] = g    // Green
    fb.Pixels[offset+2] = r    // Red
    fb.Pixels[offset+3] = 0    // Padding
}
```

And reading pixels:

```
func (fb *Framebuffer) GetPixel(x, y int) (r, g, b uint8) {
    if x < 0 || x >= fb.Width || y < 0 || y >= fb.Height {
        return 0, 0, 0
    }

    offset := (y*fb.Width + x) * 4
```

```
    return fb.Pixels[offset+2], fb.Pixels[offset+1], fb.Pixels[offset]
}
```

> **Note**: We store BGR but expose RGB in the API. Users think in RGB; the conversion happens internally.

## 11.4 Clearing the Screen

Fill the entire framebuffer with a color:

```
func (fb *Framebuffer) Clear(r, g, b uint8) {
    for i := 0; i < len(fb.Pixels); i += 4 {
        fb.Pixels[i] = b
        fb.Pixels[i+1] = g
        fb.Pixels[i+2] = r
        fb.Pixels[i+3] = 0
    }
}
```

### Optimized Clear

For solid colors, we can be clever:

```
func (fb *Framebuffer) Clear(r, g, b uint8) {
    // Build a single pixel
    pixel := []byte{b, g, r, 0}

    // Copy it to the first 4 bytes
    copy(fb.Pixels[0:4], pixel)

    // Double the filled region each iteration
    for filled := 4; filled < len(fb.Pixels); filled *= 2 {
        copy(fb.Pixels[filled:], fb.Pixels[:filled])
    }
}
```

This exploits `copy`'s efficiency - it uses optimized memory operations internally.

## 11.5 Coordinate Systems

Our framebuffer uses screen coordinates: - Origin (0, 0) is top-left - X increases rightward - Y increases downward

This matches most graphics systems and is intuitive for 2D.

## 11.6 Drawing Rectangles

Filled rectangle:

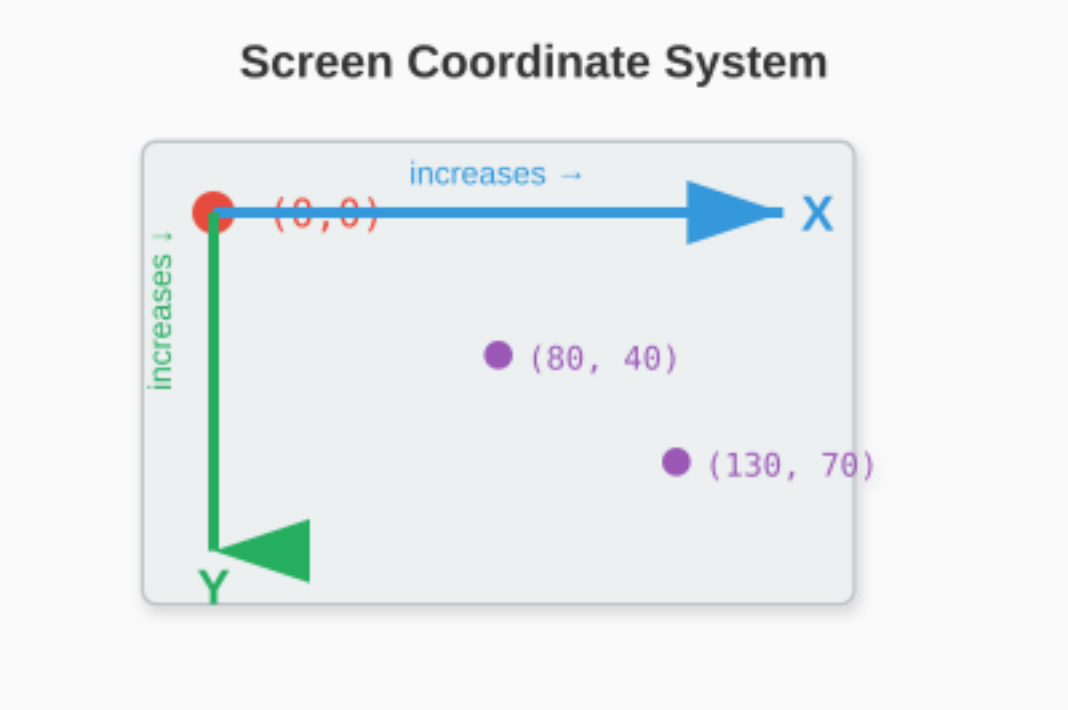


Figure 30: Screen Coordinate System

```
func (fb *Framebuffer) DrawRect(x, y, width, height int, r, g, b uint8) {
    for dy := 0; dy < height; dy++ {
        for dx := 0; dx < width; dx++ {
            fb.SetPixel(x+dx, y+dy, r, g, b)
        }
    }
}
```

This is correct but slow - `SetPixel` does bounds checking for every pixel.

### Optimized Rectangle

```
func (fb *Framebuffer) DrawRect(x, y, width, height int, r, g, b uint8) {
    // Clip to framebuffer bounds
    x0 := max(x, 0)
    y0 := max(y, 0)
    x1 := min(x+width, fb.Width)
    y1 := min(y+height, fb.Height)

    if x0 >= x1 || y0 >= y1 {
        return  // Completely outside
    }

    // Fill rows
```

```
    for py := y0; py < y1; py++ {
        rowStart := (py*fb.Width + x0) * 4
        for px := x0; px < x1; px++ {
            offset := rowStart + (px-x0)*4
            fb.Pixels[offset] = b
            fb.Pixels[offset+1] = g
            fb.Pixels[offset+2] = r
            fb.Pixels[offset+3] = 0
        }
    }
}
```

Clipping once at the start eliminates per-pixel bounds checks.

### Rectangle Outline

```
func (fb *Framebuffer) DrawRectOutline(x, y, width, height int, r, g, b uint8) {
    // Top edge
    for dx := 0; dx < width; dx++ {
        fb.SetPixel(x+dx, y, r, g, b)
    }
    // Bottom edge
    for dx := 0; dx < width; dx++ {
        fb.SetPixel(x+dx, y+height-1, r, g, b)
    }
    // Left edge
    for dy := 0; dy < height; dy++ {
        fb.SetPixel(x, y+dy, r, g, b)
    }
    // Right edge
    for dy := 0; dy < height; dy++ {
        fb.SetPixel(x+width-1, y+dy, r, g, b)
    }
}
```

## 11.7 Drawing Lines - Bresenham's Algorithm

Drawing a line between two points seems simple, but doing it efficiently for all slopes requires care. Bresenham's algorithm is the classic solution.

The key insight: we only need to decide whether to step in Y when we step in X (or vice versa).

```
func (fb *Framebuffer) DrawLine(x0, y0, x1, y1 int, r, g, b uint8) {
    dx := abs(x1 - x0)
    dy := -abs(y1 - y0)

    sx := 1
    if x0 > x1 {
```

```
        sx = -1
    }
    sy := 1
    if y0 > y1 {
        sy = -1
    }

    err := dx + dy

    for {
        fb.SetPixel(x0, y0, r, g, b)

        if x0 == x1 && y0 == y1 {
            break
        }

        e2 := 2 * err

        if e2 >= dy {
            err += dy
            x0 += sx
        }
        if e2 <= dx {
            err += dx
            y0 += sy
        }
    }
}

func abs(x int) int {
    if x < 0 {
        return -x
    }
    return x
}
```

This works for all line orientations and produces visually pleasing results.

## 11.8 Drawing Circles - Midpoint Algorithm

The midpoint circle algorithm draws circles using only integer arithmetic:

```
func (fb *Framebuffer) DrawCircle(cx, cy, radius int, r, g, b uint8) {
    x := radius
    y := 0
    err := 0

    for x >= y {
```

```
        // Draw 8 octants
        fb.SetPixel(cx+x, cy+y, r, g, b)
        fb.SetPixel(cx+y, cy+x, r, g, b)
        fb.SetPixel(cx-y, cy+x, r, g, b)
        fb.SetPixel(cx-x, cy+y, r, g, b)
        fb.SetPixel(cx-x, cy-y, r, g, b)
        fb.SetPixel(cx-y, cy-x, r, g, b)
        fb.SetPixel(cx+y, cy-x, r, g, b)
        fb.SetPixel(cx+x, cy-y, r, g, b)

        y++
        err += 1 + 2*y

        if 2*(err-x) + 1 > 0 {
            x--
            err += 1 - 2*x
        }
    }
}
```

By exploiting 8-way symmetry, we only compute one octant and mirror it.

### Filled Circle

```
func (fb *Framebuffer) FillCircle(cx, cy, radius int, r, g, b uint8) {
    for y := -radius; y <= radius; y++ {
        for x := -radius; x <= radius; x++ {
            if x*x+y*y <= radius*radius {
                fb.SetPixel(cx+x, cy+y, r, g, b)
            }
        }
    }
}
```

This brute-force approach checks every pixel in the bounding box. For small circles, it's fast enough.

## 11.9 Drawing Triangles

Triangle outline using three lines:

```
func (fb *Framebuffer) DrawTriangle(x0, y0, x1, y1, x2, y2 int, r, g, b uint8) {
    fb.DrawLine(x0, y0, x1, y1, r, g, b)
    fb.DrawLine(x1, y1, x2, y2, r, g, b)
    fb.DrawLine(x2, y2, x0, y0, r, g, b)
}
```

Filled triangles are more complex (scanline rasterization). We'll skip that for now.

## 11.10 Complete Framebuffer

```
package x11

type Framebuffer struct {
    Width  int
    Height int
    Pixels []byte
}

func NewFramebuffer(width, height int) *Framebuffer {
    return &Framebuffer{
        Width:  width,
        Height: height,
        Pixels: make([]byte, width*height*4),
    }
}

func (fb *Framebuffer) SetPixel(x, y int, r, g, b uint8) {
    if x < 0 || x >= fb.Width || y < 0 || y >= fb.Height {
        return
    }
    offset := (y*fb.Width + x) * 4
    fb.Pixels[offset] = b
    fb.Pixels[offset+1] = g
    fb.Pixels[offset+2] = r
    fb.Pixels[offset+3] = 0
}

func (fb *Framebuffer) GetPixel(x, y int) (r, g, b uint8) {
    if x < 0 || x >= fb.Width || y < 0 || y >= fb.Height {
        return 0, 0, 0
    }
    offset := (y*fb.Width + x) * 4
    return fb.Pixels[offset+2], fb.Pixels[offset+1], fb.Pixels[offset]
}

func (fb *Framebuffer) Clear(r, g, b uint8) {
    for i := 0; i < len(fb.Pixels); i += 4 {
        fb.Pixels[i] = b
        fb.Pixels[i+1] = g
        fb.Pixels[i+2] = r
        fb.Pixels[i+3] = 0
    }
}

func (fb *Framebuffer) DrawRect(x, y, width, height int, r, g, b uint8) {
```

```
    for dy := 0; dy < height; dy++ {
        for dx := 0; dx < width; dx++ {
            fb.SetPixel(x+dx, y+dy, r, g, b)
        }
    }
}

func (fb *Framebuffer) DrawRectOutline(x, y, width, height int, r, g, b uint8) {
    for dx := 0; dx < width; dx++ {
        fb.SetPixel(x+dx, y, r, g, b)
        fb.SetPixel(x+dx, y+height-1, r, g, b)
    }
    for dy := 0; dy < height; dy++ {
        fb.SetPixel(x, y+dy, r, g, b)
        fb.SetPixel(x+width-1, y+dy, r, g, b)
    }
}

func (fb *Framebuffer) DrawLine(x0, y0, x1, y1 int, r, g, b uint8) {
    // Bresenham's algorithm
    dx := abs(x1 - x0)
    dy := -abs(y1 - y0)
    sx, sy := 1, 1
    if x0 > x1 { sx = -1 }
    if y0 > y1 { sy = -1 }
    err := dx + dy

    for {
        fb.SetPixel(x0, y0, r, g, b)
        if x0 == x1 && y0 == y1 { break }
        e2 := 2 * err
        if e2 >= dy { err += dy; x0 += sx }
        if e2 <= dx { err += dx; y0 += sy }
    }
}

func (fb *Framebuffer) DrawCircle(cx, cy, radius int, r, g, b uint8) {
    x, y, err := radius, 0, 0
    for x >= y {
        fb.SetPixel(cx+x, cy+y, r, g, b)
        fb.SetPixel(cx+y, cy+x, r, g, b)
        fb.SetPixel(cx-y, cy+x, r, g, b)
        fb.SetPixel(cx-x, cy+y, r, g, b)
        fb.SetPixel(cx-x, cy-y, r, g, b)
        fb.SetPixel(cx-y, cy-x, r, g, b)
        fb.SetPixel(cx+y, cy-x, r, g, b)
        fb.SetPixel(cx+x, cy-y, r, g, b)
```

```
        y++
        err += 1 + 2*y
        if 2*(err-x)+1 > 0 { x--; err += 1 - 2*x }
    }
}

func (fb *Framebuffer) FillCircle(cx, cy, radius int, r, g, b uint8) {
    for y := -radius; y <= radius; y++ {
        for x := -radius; x <= radius; x++ {
            if x*x+y*y <= radius*radius {
                fb.SetPixel(cx+x, cy+y, r, g, b)
            }
        }
    }
}

func (fb *Framebuffer) DrawTriangle(x0, y0, x1, y1, x2, y2 int, r, g, b uint8) {
    fb.DrawLine(x0, y0, x1, y1, r, g, b)
    fb.DrawLine(x1, y1, x2, y2, r, g, b)
    fb.DrawLine(x2, y2, x0, y0, r, g, b)
}

func abs(x int) int {
    if x < 0 { return -x }
    return x
}
```

---

**Key Takeaways:**

- A framebuffer is an in-memory pixel array
- Pixels are stored as BGRX (4 bytes each)
- Bounds checking prevents memory corruption
- Bresenham's algorithm draws efficient lines
- The midpoint algorithm draws circles with integer math
- Clipping at the start is faster than per-pixel checks

We have drawing primitives! Next, let's wrap everything in a clean public API.

# Chapter 12: The Public API

We've built all the pieces: connection handling, window creation, event processing, and a framebuffer. Now we design a clean public API that hides the X11 complexity.

## 12.1 API Design Philosophy

Our goals: - **Simple**: Basic usage should be obvious - **Safe**: Hard to misuse - **Efficient**: No unnecessary allocations - **Familiar**: Similar to SDL/SFML patterns

### What Users Want

```
// This is what users want to write
func main() {
    win, _ := glow.NewWindow("My Game", 800, 600)
    defer win.Close()

    canvas := win.Canvas()

    for win.IsOpen() {
        for event := win.PollEvent(); event != nil; event = win.PollEvent() {
            // Handle events
        }

        canvas.Clear(glow.Black)
        canvas.DrawRect(100, 100, 50, 50, glow.Red)
        win.Display()
    }
}
```

Clean, readable, no X11 knowledge required.

## 12.2 The Color Type

Colors are fundamental. Let's make them pleasant to use:

```
// glow/color.go
package glow
```

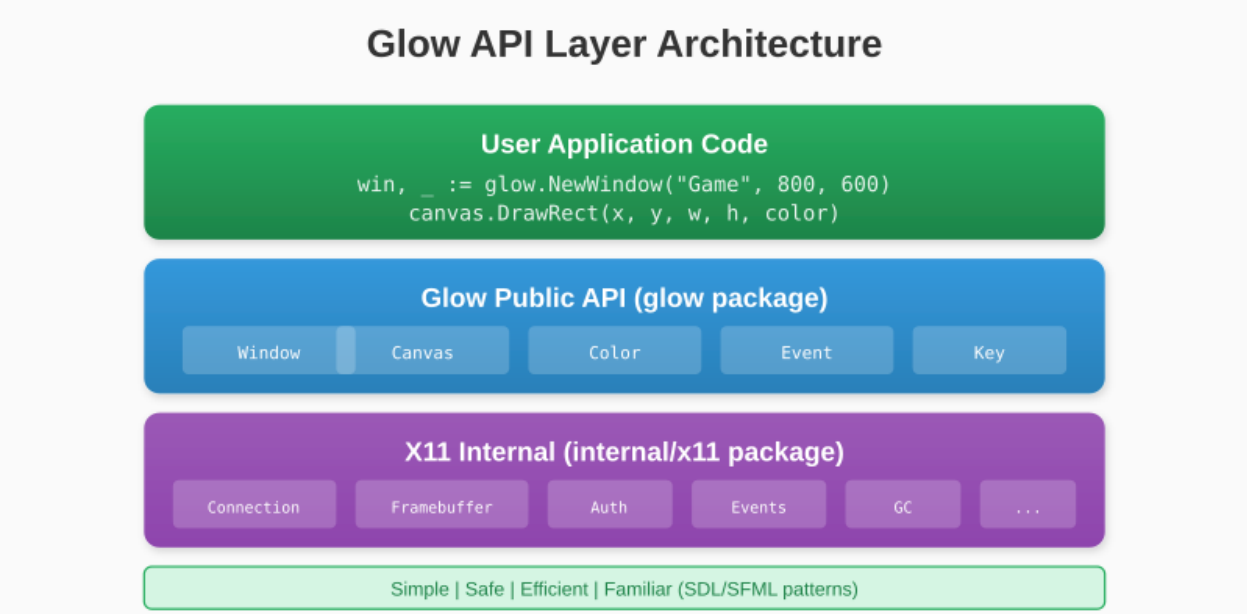


Figure 31: Glow API Layer Architecture

```go
type Color struct {
    R, G, B, A uint8
}

// Predefined colors
var (
    Black       = Color{0, 0, 0, 255}
    White       = Color{255, 255, 255, 255}
    Red         = Color{255, 0, 0, 255}
    Green       = Color{0, 255, 0, 255}
    Blue        = Color{0, 0, 255, 255}
    Yellow      = Color{255, 255, 0, 255}
    Cyan        = Color{0, 255, 255, 255}
    Magenta     = Color{255, 0, 255, 255}
    Transparent = Color{0, 0, 0, 0}
)

// RGB creates an opaque color
func RGB(r, g, b uint8) Color {
    return Color{r, g, b, 255}
}

// RGBA creates a color with alpha
func RGBA(r, g, b, a uint8) Color {
    return Color{r, g, b, a}
}
```

```go
// Hex creates a color from a hex value (0xRRGGBB)
func Hex(hex uint32) Color {
    return Color{
        R: uint8((hex >> 16) & 0xFF),
        G: uint8((hex >> 8) & 0xFF),
        B: uint8(hex & 0xFF),
        A: 255,
    }
}
```

### Color Methods

```go
// Blend blends two colors using alpha
func (c Color) Blend(other Color) Color {
    if other.A == 255 {
        return other
    }
    if other.A == 0 {
        return c
    }

    // Simple alpha blending
    alpha := float32(other.A) / 255.0
    invAlpha := 1.0 - alpha

    return Color{
        R: uint8(float32(c.R)*invAlpha + float32(other.R)*alpha),
        G: uint8(float32(c.G)*invAlpha + float32(other.G)*alpha),
        B: uint8(float32(c.B)*invAlpha + float32(other.B)*alpha),
        A: 255,
    }
}

// Darker returns a darker version of the color
func (c Color) Darker(factor float32) Color {
    return Color{
        R: uint8(float32(c.R) * (1 - factor)),
        G: uint8(float32(c.G) * (1 - factor)),
        B: uint8(float32(c.B) * (1 - factor)),
        A: c.A,
    }
}

// Lighter returns a lighter version of the color
func (c Color) Lighter(factor float32) Color {
    return Color{
        R: uint8(float32(c.R) + (255-float32(c.R))*factor),
```

```
        G: uint8(float32(c.G) + (255-float32(c.G))*factor),
        B: uint8(float32(c.B) + (255-float32(c.B))*factor),
        A: c.A,
    }
}
```

## 12.3 The Canvas Type

The Canvas wraps our framebuffer with a friendly API:

```
// glow/canvas.go
package glow

import "github.com/AchrafSoltani/glow/internal/x11"

type Canvas struct {
    fb     *x11.Framebuffer
    width  int
    height int
}

func newCanvas(width, height int) *Canvas {
    return &Canvas{
        fb:     x11.NewFramebuffer(width, height),
        width:  width,
        height: height,
    }
}

func (c *Canvas) Width() int  { return c.width }
func (c *Canvas) Height() int { return c.height }

// Pixels returns the raw pixel data (for advanced use)
func (c *Canvas) Pixels() []byte {
    return c.fb.Pixels
}
```

### Drawing Methods

```
func (c *Canvas) Clear(color Color) {
    c.fb.Clear(color.R, color.G, color.B)
}

func (c *Canvas) SetPixel(x, y int, color Color) {
    c.fb.SetPixel(x, y, color.R, color.G, color.B)
}
```

```
func (c *Canvas) GetPixel(x, y int) Color {
    r, g, b := c.fb.GetPixel(x, y)
    return Color{r, g, b, 255}
}

func (c *Canvas) DrawRect(x, y, width, height int, color Color) {
    c.fb.DrawRect(x, y, width, height, color.R, color.G, color.B)
}

func (c *Canvas) DrawRectOutline(x, y, width, height int, color Color) {
    c.fb.DrawRectOutline(x, y, width, height, color.R, color.G, color.B)
}

func (c *Canvas) DrawLine(x0, y0, x1, y1 int, color Color) {
    c.fb.DrawLine(x0, y0, x1, y1, color.R, color.G, color.B)
}

func (c *Canvas) DrawCircle(cx, cy, radius int, color Color) {
    c.fb.DrawCircle(cx, cy, radius, color.R, color.G, color.B)
}

func (c *Canvas) FillCircle(cx, cy, radius int, color Color) {
    c.fb.FillCircle(cx, cy, radius, color.R, color.G, color.B)
}

func (c *Canvas) DrawTriangle(x0, y0, x1, y1, x2, y2 int, color Color) {
    c.fb.DrawTriangle(x0, y0, x1, y1, x2, y2, color.R, color.G, color.B)
}
```

### Resize Support

When the window resizes, we need a new framebuffer:

```
func (c *Canvas) Resize(width, height int) {
    if width == c.width && height == c.height {
        return
    }

    c.fb = x11.NewFramebuffer(width, height)
    c.width = width
    c.height = height
}
```

## 12.4 The Window Type

The Window is the main entry point:

```go
// glow/window.go
package glow

import (
    "github.com/AchrafSoltani/glow/internal/x11"
)

type Window struct {
    conn   *x11.Connection
    window *x11.Window
    canvas *Canvas

    open   bool
    width  int
    height int
}

func NewWindow(title string, width, height int) (*Window, error) {
    conn, err := x11.Connect()
    if err != nil {
        return nil, err
    }

    window, err := x11.NewWindow(conn, title, width, height)
    if err != nil {
        conn.Close()
        return nil, err
    }

    return &Window{
        conn:   conn,
        window: window,
        canvas: newCanvas(width, height),
        open:   true,
        width:  width,
        height: height,
    }, nil
}
```

### Window Methods

```go
func (w *Window) Close() {
    if !w.open {
        return
    }
    w.open = false
    w.window.Close()
```

```
    w.conn.Close()
}

func (w *Window) IsOpen() bool {
    return w.open
}

func (w *Window) Width() int {
    return w.width
}

func (w *Window) Height() int {
    return w.height
}

func (w *Window) Canvas() *Canvas {
    return w.canvas
}

func (w *Window) SetTitle(title string) {
    w.conn.SetWindowTitle(w.window.ID(), title)
}
```

### Display Method

The crucial method that sends our framebuffer to the screen:

```
func (w *Window) Display() error {
    return w.conn.PutImage(
        w.window.ID(),
        w.window.GC(),
        uint16(w.canvas.width),
        uint16(w.canvas.height),
        0, 0,
        w.conn.RootDepth,
        w.canvas.Pixels(),
    )
}
```

## 12.5 Event Types

We define our own event types, independent of X11:

```
// glow/events.go
package glow

type Event interface {
    isEvent()  // Marker method
```

```
}

// KeyEvent for keyboard input
type KeyEvent struct {
    Key     Key
    Pressed bool
    Shift   bool
    Ctrl    bool
    Alt     bool
}

func (KeyEvent) isEvent() {}

// MouseButtonEvent for mouse clicks
type MouseButtonEvent struct {
    Button  MouseButton
    Pressed bool
    X, Y    int
}

func (MouseButtonEvent) isEvent() {}

// MouseMoveEvent for mouse movement
type MouseMoveEvent struct {
    X, Y int
}

func (MouseMoveEvent) isEvent() {}

// ResizeEvent when window size changes
type ResizeEvent struct {
    Width, Height int
}

func (ResizeEvent) isEvent() {}

// CloseEvent when close button is clicked
type CloseEvent struct{}

func (CloseEvent) isEvent() {}
```

### Key Constants

```
type Key int

const (
    KeyUnknown Key = iota
```

```
    KeyEscape
    KeyEnter
    KeySpace
    KeyBackspace
    KeyTab

    KeyLeft
    KeyRight
    KeyUp
    KeyDown

    KeyA
    KeyB
    KeyC
    // ... through KeyZ

    Key0
    Key1
    // ... through Key9

    KeyF1
    KeyF2
    // ... through KeyF12
)
```

### Mouse Button Constants

```
type MouseButton int

const (
    MouseLeft MouseButton = iota + 1
    MouseMiddle
    MouseRight
    MouseWheelUp
    MouseWheelDown
)
```

## 12.6 Event Polling

Convert X11 events to our event types:

```
func (w *Window) PollEvent() Event {
    x11Event := w.window.PollEvent()
    if x11Event == nil {
        return nil
    }
```

```
    return w.convertEvent(x11Event)
}

func (w *Window) convertEvent(e x11.Event) Event {
    switch ev := e.(type) {
    case x11.KeyEvent:
        return KeyEvent{
            Key:     convertKeycode(ev.Keycode),
            Pressed: ev.EventType == x11.EventKeyPress,
            Shift:   ev.State&x11.ShiftMask != 0,
            Ctrl:    ev.State&x11.ControlMask != 0,
            Alt:     ev.State&x11.Mod1Mask != 0,
        }

    case x11.ButtonEvent:
        return MouseButtonEvent{
            Button:  MouseButton(ev.Button),
            Pressed: ev.EventType == x11.EventButtonPress,
            X:       int(ev.X),
            Y:       int(ev.Y),
        }

    case x11.MotionEvent:
        return MouseMoveEvent{
            X: int(ev.X),
            Y: int(ev.Y),
        }

    case x11.ConfigureEvent:
        if int(ev.Width) != w.width || int(ev.Height) != w.height {
            w.width = int(ev.Width)
            w.height = int(ev.Height)
            w.canvas.Resize(w.width, w.height)
            return ResizeEvent{
                Width:  w.width,
                Height: w.height,
            }
        }
        return nil

    case x11.ClientMessageEvent:
        if x11.IsDeleteWindowEvent(ev, w.conn.AtomWmProtocols(),
            w.conn.AtomWmDeleteWindow()) {
            w.open = false
            return CloseEvent{}
        }
```

```
    case x11.ExposeEvent:
        // Trigger redraw by returning nil
        // The main loop will redraw anyway
        return nil
    }

    return nil
}
```

### Keycode Conversion

```
func convertKeycode(code uint8) Key {
    switch code {
    case 9:
        return KeyEscape
    case 36:
        return KeyEnter
    case 65:
        return KeySpace
    case 22:
        return KeyBackspace
    case 23:
        return KeyTab
    case 111:
        return KeyUp
    case 116:
        return KeyDown
    case 113:
        return KeyLeft
    case 114:
        return KeyRight
    case 38:
        return KeyA
    case 56:
        return KeyB
    // ... more keys
    default:
        return KeyUnknown
    }
}
```

## 12.7 Complete Window Implementation

```
package glow

import (
```

```
    "github.com/AchrafSoltani/glow/internal/x11"
)

type Window struct {
    conn   *x11.Connection
    window *x11.Window
    canvas *Canvas

    open   bool
    width  int
    height int
}

func NewWindow(title string, width, height int) (*Window, error) {
    conn, err := x11.Connect()
    if err != nil {
        return nil, err
    }

    window, err := x11.NewWindow(conn, title, width, height)
    if err != nil {
        conn.Close()
        return nil, err
    }

    return &Window{
        conn:   conn,
        window: window,
        canvas: newCanvas(width, height),
        open:   true,
        width:  width,
        height: height,
    }, nil
}

func (w *Window) Close() {
    if !w.open {
        return
    }
    w.open = false
    w.window.Close()
    w.conn.Close()
}

func (w *Window) IsOpen() bool {
    return w.open
}
```

```
func (w *Window) Width() int  { return w.width }
func (w *Window) Height() int { return w.height }

func (w *Window) Canvas() *Canvas {
    return w.canvas
}

func (w *Window) SetTitle(title string) {
    w.conn.SetWindowTitle(w.window.ID(), title)
}

func (w *Window) Display() error {
    return w.conn.PutImage(
        w.window.ID(),
        w.window.GC(),
        uint16(w.canvas.width),
        uint16(w.canvas.height),
        0, 0,
        w.conn.RootDepth,
        w.canvas.Pixels(),
    )
}

func (w *Window) PollEvent() Event {
    x11Event := w.window.PollEvent()
    if x11Event == nil {
        return nil
    }
    return w.convertEvent(x11Event)
}

func (w *Window) WaitEvent() Event {
    x11Event := w.window.WaitEvent()
    return w.convertEvent(x11Event)
}

func (w *Window) convertEvent(e x11.Event) Event {
    // ... conversion logic from above
}
```

## 12.8 Usage Example

Here's how our API looks in practice:

```
package main

import (
```

```
    "github.com/AchrafSoltani/glow"
)

func main() {
    win, err := glow.NewWindow("Glow Example", 800, 600)
    if err != nil {
        panic(err)
    }
    defer win.Close()

    canvas := win.Canvas()
    x, y := 400, 300

    for win.IsOpen() {
        // Handle events
        for event := win.PollEvent(); event != nil; event = win.PollEvent() {
            switch e := event.(type) {
            case glow.KeyEvent:
                if e.Pressed {
                    switch e.Key {
                    case glow.KeyEscape:
                        win.Close()
                    case glow.KeyLeft:
                        x -= 10
                    case glow.KeyRight:
                        x += 10
                    case glow.KeyUp:
                        y -= 10
                    case glow.KeyDown:
                        y += 10
                    }
                }
            case glow.CloseEvent:
                win.Close()
            }
        }

        // Draw
        canvas.Clear(glow.Black)
        canvas.FillCircle(x, y, 30, glow.Red)
        canvas.DrawCircle(x, y, 30, glow.White)

        win.Display()
    }
}
```

Clean, simple, no X11 knowledge required.

## 12.9 Package Structure

Final package layout:

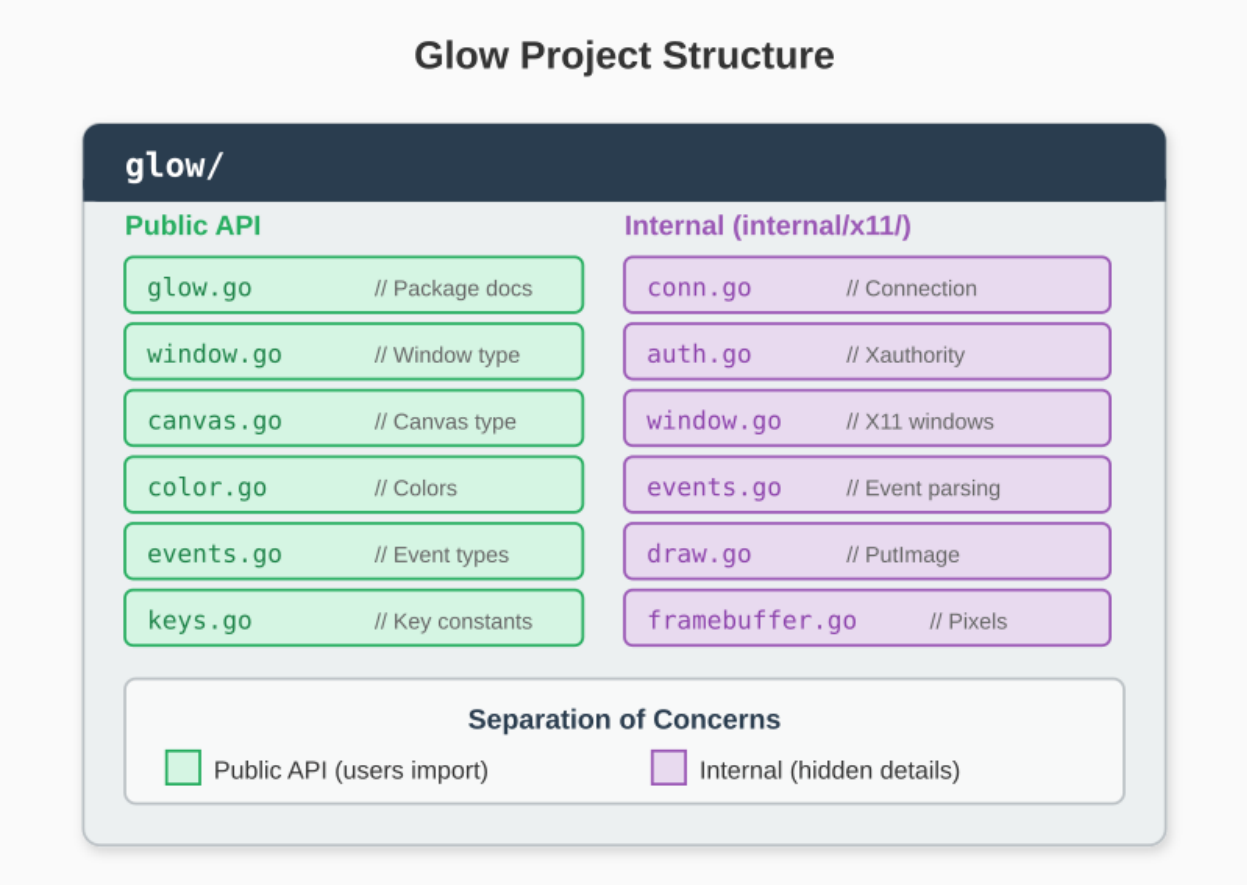


Figure 32: Glow Project Structure

The `internal` package is hidden from users - they only see the clean public API.

---

**Key Takeaways:**

- Hide complexity in `internal/` packages
- Provide friendly types (Color, Canvas, Window)
- Convert between internal and public event types
- Use constants for common values (colors, keys)
- Keep the API surface small and focused
- Match familiar patterns from SDL/SFML

Our library is complete! Users can create windows, draw graphics, and handle input with just a few lines of code. Next, let's build some real applications.

# Chapter 13: Building Pong

Theory becomes real when you build something. This chapter creates the classic game Pong - two paddles, a ball, and simple physics. It's the perfect first project for our graphics library.

## 13.1 Game Design

Pong is simple but complete: - Two paddles (one per player, or player vs AI) - A ball that bounces - Score tracking - Win condition


Figure 33: Pong Game Layout

## 13.2 Game Constants

```
package main

import (
```

```
    "time"
    "github.com/AchrafSoltani/glow"
)

const (
    windowWidth  = 800
    windowHeight = 600

    paddleWidth  = 15
    paddleHeight = 80
    paddleSpeed  = 400  // pixels per second
    paddleMargin = 30   // distance from edge

    ballSize  = 15
    ballSpeed = 350  // initial speed

    winScore = 5
)
```

## 13.3 Game State

```
type Game struct {
    // Paddles
    paddle1Y float64
    paddle2Y float64

    // Ball
    ballX, ballY   float64
    ballVX, ballVY float64

    // Scores
    score1, score2 int

    // State
    paused    bool
    gameOver  bool
    winner    int
}

func NewGame() *Game {
    g := &Game{}
    g.Reset()
    return g
}

func (g *Game) Reset() {
```

```
    // Center paddles vertically
    g.paddle1Y = float64(windowHeight-paddleHeight) / 2
    g.paddle2Y = float64(windowHeight-paddleHeight) / 2

    // Reset ball to center
    g.ResetBall(1)  // Serve toward player 2

    g.paused = false
    g.gameOver = false
}

func (g *Game) ResetBall(direction int) {
    g.ballX = float64(windowWidth) / 2
    g.ballY = float64(windowHeight) / 2

    // Random-ish angle (45 degrees)
    g.ballVX = float64(direction) * ballSpeed * 0.7
    g.ballVY = ballSpeed * 0.7

    // Alternate vertical direction based on scores
    if (g.score1+g.score2)%2 == 1 {
        g.ballVY = -g.ballVY
    }
}
```

## 13.4 Input Handling

```
type Input struct {
    // Player 1 (W/S keys)
    p1Up, p1Down bool

    // Player 2 (Up/Down arrows)
    p2Up, p2Down bool

    // Controls
    pause, quit bool
    restart     bool
}

func (g *Game) HandleEvent(event glow.Event, input *Input) {
    switch e := event.(type) {
    case glow.KeyEvent:
        pressed := e.Pressed

        switch e.Key {
        // Player 1
```

```
        case glow.KeyW:
            input.p1Up = pressed
        case glow.KeyS:
            input.p1Down = pressed

        // Player 2
        case glow.KeyUp:
            input.p2Up = pressed
        case glow.KeyDown:
            input.p2Down = pressed

        // Controls
        case glow.KeySpace:
            if pressed {
                input.pause = true
            }
        case glow.KeyR:
            if pressed {
                input.restart = true
            }
        case glow.KeyEscape:
            input.quit = true
        }

    case glow.CloseEvent:
        input.quit = true
    }
}
```

## 13.5 Game Logic

### Update Method

```
func (g *Game) Update(dt float64, input *Input) {
    // Handle pause toggle
    if input.pause {
        g.paused = !g.paused
        input.pause = false
    }

    // Handle restart
    if input.restart {
        g.score1 = 0
        g.score2 = 0
        g.Reset()
        input.restart = false
        return
```

```
    }

    if g.paused || g.gameOver {
        return
    }

    // Move paddles
    g.updatePaddles(dt, input)

    // Move ball
    g.updateBall(dt)

    // Check scoring
    g.checkScore()
}
```

### Paddle Movement

```
func (g *Game) updatePaddles(dt float64, input *Input) {
    // Player 1
    if input.p1Up {
        g.paddle1Y -= paddleSpeed * dt
    }
    if input.p1Down {
        g.paddle1Y += paddleSpeed * dt
    }

    // Player 2
    if input.p2Up {
        g.paddle2Y -= paddleSpeed * dt
    }
    if input.p2Down {
        g.paddle2Y += paddleSpeed * dt
    }

    // Clamp to screen bounds
    g.paddle1Y = clamp(g.paddle1Y, 0, float64(windowHeight-paddleHeight))
    g.paddle2Y = clamp(g.paddle2Y, 0, float64(windowHeight-paddleHeight))
}

func clamp(v, min, max float64) float64 {
    if v < min {
        return min
    }
    if v > max {
        return max
    }
```

```
    return v
}
```

### Ball Physics

```
func (g *Game) updateBall(dt float64) {
    // Move ball
    g.ballX += g.ballVX * dt
    g.ballY += g.ballVY * dt

    // Bounce off top and bottom
    if g.ballY <= 0 {
        g.ballY = 0
        g.ballVY = -g.ballVY
    }
    if g.ballY >= float64(windowHeight-ballSize) {
        g.ballY = float64(windowHeight - ballSize)
        g.ballVY = -g.ballVY
    }

    // Check paddle collisions
    g.checkPaddleCollision()
}
```

### Paddle Collision

```
func (g *Game) checkPaddleCollision() {
    // Paddle 1 (left side)
    paddle1X := float64(paddleMargin)
    if g.ballX <= paddle1X+paddleWidth &&
        g.ballX+ballSize >= paddle1X &&
        g.ballY+ballSize >= g.paddle1Y &&
        g.ballY <= g.paddle1Y+paddleHeight {

        g.ballX = paddle1X + paddleWidth
        g.ballVX = -g.ballVX

        // Add spin based on where ball hit paddle
        g.addSpin(g.paddle1Y)
        g.speedUp()
    }

    // Paddle 2 (right side)
    paddle2X := float64(windowWidth - paddleMargin - paddleWidth)
    if g.ballX+ballSize >= paddle2X &&
        g.ballX <= paddle2X+paddleWidth &&
        g.ballY+ballSize >= g.paddle2Y &&
```

```
        g.ballY <= g.paddle2Y+paddleHeight {

        g.ballX = paddle2X - ballSize
        g.ballVX = -g.ballVX

        g.addSpin(g.paddle2Y)
        g.speedUp()
    }
}

func (g *Game) addSpin(paddleY float64) {
    // Ball position relative to paddle center
    paddleCenter := paddleY + paddleHeight/2
    ballCenter := g.ballY + ballSize/2
    offset := (ballCenter - paddleCenter) / (paddleHeight / 2)

    // Adjust vertical velocity based on hit position
    g.ballVY += offset * 150
}

func (g *Game) speedUp() {
    // Increase speed slightly on each hit
    g.ballVX *= 1.05
    g.ballVY *= 1.02
}
```

### Scoring

```
func (g *Game) checkScore() {
    // Ball passed left edge - Player 2 scores
    if g.ballX < 0 {
        g.score2++
        if g.score2 >= winScore {
            g.gameOver = true
            g.winner = 2
        } else {
            g.ResetBall(1)  // Serve toward player 2
        }
    }

    // Ball passed right edge - Player 1 scores
    if g.ballX > float64(windowWidth) {
        g.score1++
        if g.score1 >= winScore {
            g.gameOver = true
            g.winner = 1
        } else {
```

```
            g.ResetBall(-1)  // Serve toward player 1
        }
    }
}
```

## 13.6 Rendering

```
func (g *Game) Draw(canvas *glow.Canvas) {
    // Background
    canvas.Clear(glow.RGB(20, 20, 30))

    // Center line
    drawDashedLine(canvas, windowWidth/2, 0, windowWidth/2, windowHeight,
        glow.RGB(60, 60, 70))

    // Paddles
    paddle1X := paddleMargin
    paddle2X := windowWidth - paddleMargin - paddleWidth

    canvas.DrawRect(paddle1X, int(g.paddle1Y), paddleWidth, paddleHeight,
        glow.White)
    canvas.DrawRect(paddle2X, int(g.paddle2Y), paddleWidth, paddleHeight,
        glow.White)

    // Ball
    canvas.DrawRect(int(g.ballX), int(g.ballY), ballSize, ballSize,
        glow.RGB(255, 200, 100))

    // Scores
    drawScore(canvas, g.score1, windowWidth/4, 50)
    drawScore(canvas, g.score2, 3*windowWidth/4, 50)

    // Pause/Game Over overlay
    if g.paused {
        drawCenteredText(canvas, "PAUSED", windowHeight/2)
    }
    if g.gameOver {
        drawCenteredText(canvas, "GAME OVER", windowHeight/2-30)
        if g.winner == 1 {
            drawCenteredText(canvas, "Player 1 Wins!", windowHeight/2+30)
        } else {
            drawCenteredText(canvas, "Player 2 Wins!", windowHeight/2+30)
        }
    }
}
```

```
func drawDashedLine(canvas *glow.Canvas, x0, y0, x1, y1 int, color glow.Color) {
    dashLen := 20
    gapLen := 10
    y := y0
    drawing := true

    for y < y1 {
        if drawing {
            endY := y + dashLen
            if endY > y1 {
                endY = y1
            }
            canvas.DrawLine(x0, y, x1, endY, color)
            y = endY
        } else {
            y += gapLen
        }
        drawing = !drawing
    }
}
```

### Drawing Numbers

Without a font system, we draw numbers using rectangles:

```
func drawScore(canvas *glow.Canvas, score int, x, y int) {
    // Simple 7-segment style digits
    digitWidth := 30
    digitHeight := 50
    thickness := 6

    drawDigit(canvas, score, x-digitWidth/2, y, digitWidth, digitHeight, thickness)
}

func drawDigit(canvas *glow.Canvas, digit int, x, y, w, h, t int) {
    color := glow.White

    // Define which segments are on for each digit
    // Segments: top, topLeft, topRight, middle, bottomLeft, bottomRight, bottom
    segments := [][]bool{
        {true, true, true, false, true, true, true},     // 0
        {false, false, true, false, false, true, false}, // 1
        {true, false, true, true, true, false, true},    // 2
        {true, false, true, true, false, true, true},    // 3
        {false, true, true, true, false, true, false},   // 4
        {true, true, false, true, false, true, true},    // 5
        {true, true, false, true, true, true, true},     // 6
        {true, false, true, false, false, true, false},  // 7
```

```
        {true, true, true, true, true, true, true},      // 8
        {true, true, true, true, false, true, true},     // 9
    }

    if digit < 0 || digit > 9 {
        return
    }

    seg := segments[digit]
    midY := y + h/2

    // Top
    if seg[0] {
        canvas.DrawRect(x, y, w, t, color)
    }
    // Top Left
    if seg[1] {
        canvas.DrawRect(x, y, t, h/2, color)
    }
    // Top Right
    if seg[2] {
        canvas.DrawRect(x+w-t, y, t, h/2, color)
    }
    // Middle
    if seg[3] {
        canvas.DrawRect(x, midY-t/2, w, t, color)
    }
    // Bottom Left
    if seg[4] {
        canvas.DrawRect(x, midY, t, h/2, color)
    }
    // Bottom Right
    if seg[5] {
        canvas.DrawRect(x+w-t, midY, t, h/2, color)
    }
    // Bottom
    if seg[6] {
        canvas.DrawRect(x, y+h-t, w, t, color)
    }
}

func drawCenteredText(canvas *glow.Canvas, text string, y int) {
    // Placeholder - draw a rectangle indicating text area
    textWidth := len(text) * 15
    x := (windowWidth - textWidth) / 2
    canvas.DrawRect(x, y-10, textWidth, 20, glow.RGB(100, 100, 100))
}
```

## 13.7 Main Loop

```
func main() {
    win, err := glow.NewWindow("Pong", windowWidth, windowHeight)
    if err != nil {
        panic(err)
    }
    defer win.Close()

    game := NewGame()
    input := &Input{}
    canvas := win.Canvas()

    lastTime := time.Now()

    for win.IsOpen() {
        // Calculate delta time
        now := time.Now()
        dt := now.Sub(lastTime).Seconds()
        lastTime = now

        // Handle events
        for event := win.PollEvent(); event != nil; event = win.PollEvent() {
            game.HandleEvent(event, input)
        }

        if input.quit {
            break
        }

        // Update
        game.Update(dt, input)

        // Draw
        game.Draw(canvas)
        win.Display()

        // Cap frame rate
        time.Sleep(time.Millisecond * 16)
    }
}
```

## 13.8 Adding AI

For single-player mode, add a simple AI:

```
func (g *Game) updateAI(dt float64) {
    // AI controls paddle 2
    // Simple strategy: follow the ball

    ballCenter := g.ballY + ballSize/2
    paddleCenter := g.paddle2Y + paddleHeight/2

    // Add some "reaction" delay by limiting speed
    aiSpeed := paddleSpeed * 0.7

    if ballCenter < paddleCenter-10 {
        g.paddle2Y -= aiSpeed * dt
    } else if ballCenter > paddleCenter+10 {
        g.paddle2Y += aiSpeed * dt
    }

    // Clamp
    g.paddle2Y = clamp(g.paddle2Y, 0, float64(windowHeight-paddleHeight))
}
```

Call `g.updateAI(dt)` instead of processing player 2 input for single-player mode.

## 13.9 Polish

### Ball Trail Effect

```
type BallTrail struct {
    positions [][2]float64
    maxLen    int
}

func (t *BallTrail) Add(x, y float64) {
    t.positions = append(t.positions, [2]float64{x, y})
    if len(t.positions) > t.maxLen {
        t.positions = t.positions[1:]
    }
}

func (t *BallTrail) Draw(canvas *glow.Canvas) {
    for i, pos := range t.positions {
        alpha := float64(i) / float64(len(t.positions))
        gray := uint8(100 * alpha)
        size := int(float64(ballSize) * alpha)
        canvas.DrawRect(int(pos[0]), int(pos[1]), size, size,
            glow.RGB(gray, gray, gray))
    }
}
```

### Screen Shake

```
type ScreenShake struct {
    intensity float64
    duration  float64
    elapsed   float64
}

func (s *ScreenShake) Trigger(intensity, duration float64) {
    s.intensity = intensity
    s.duration = duration
    s.elapsed = 0
}

func (s *ScreenShake) Update(dt float64) (offsetX, offsetY int) {
    if s.elapsed >= s.duration {
        return 0, 0
    }

    s.elapsed += dt
    remaining := 1.0 - s.elapsed/s.duration
    magnitude := s.intensity * remaining

    offsetX = int((rand.Float64()*2 - 1) * magnitude)
    offsetY = int((rand.Float64()*2 - 1) * magnitude)
    return
}
```

Trigger shake on score.

---

**Key Takeaways:**

- Separate game logic (Update) from rendering (Draw)
- Use delta time for frame-rate independent physics
- Input state tracks which keys are currently held
- Simple AI can make single-player engaging
- Visual polish (trails, shake) adds juice with minimal code

Pong demonstrates the core game loop pattern. Next, we'll build something more creative: a paint application.

# Chapter 14: Building Paint

Games aren't the only application for graphics libraries. This chapter builds a simple paint program - demonstrating mouse input, tool systems, and persistent canvas state.

## 14.1 Application Design

Our paint program features: - Freehand drawing with the mouse - Multiple brush sizes - Color picker - Clear canvas button - Eraser tool

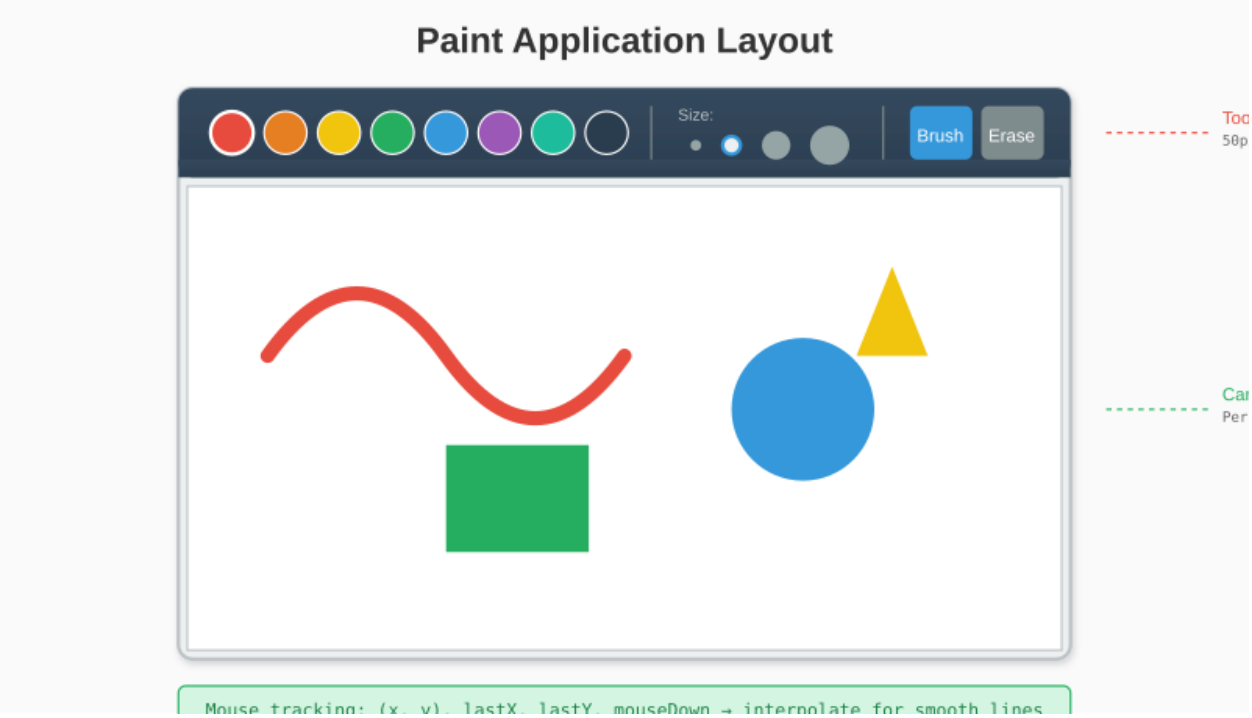


Figure 34: Paint Application Layout

## 14.2 Application Structure

```
package main

import (
```

```
    "github.com/AchrafSoltani/glow"
)

const (
    windowWidth  = 800
    windowHeight = 600
    toolbarHeight = 50
)

type Tool int

const (
    ToolBrush Tool = iota
    ToolEraser
)

type App struct {
    // Canvas state (persistent between frames)
    canvasPixels []glow.Color

    // Current tool
    tool       Tool
    brushSize  int
    brushColor glow.Color

    // Mouse state
    mouseX, mouseY   int
    lastX, lastY     int
    mouseDown        bool
    hasLastPosition  bool

    // UI state
    colors    []glow.Color
    sizes     []int
}
```

## 14.3 Initialization

```
func NewApp() *App {
    app := &App{
        canvasPixels: make([]glow.Color, windowWidth*(windowHeight-toolbarHeight)),
        tool:         ToolBrush,
        brushSize:    8,
        brushColor:   glow.Black,
        sizes:        []int{2, 5, 8, 15, 25},
    }
```

```
    // Color palette
    app.colors = []glow.Color{
        glow.Black,
        glow.White,
        glow.Red,
        glow.RGB(255, 127, 0),  // Orange
        glow.Yellow,
        glow.Green,
        glow.Cyan,
        glow.Blue,
        glow.Magenta,
        glow.RGB(139, 69, 19),  // Brown
    }

    // Initialize canvas to white
    app.Clear()

    return app
}

func (app *App) Clear() {
    for i := range app.canvasPixels {
        app.canvasPixels[i] = glow.White
    }
}
```

## 14.4 Event Handling

```
func (app *App) HandleEvent(event glow.Event) bool {
    switch e := event.(type) {
    case glow.MouseButtonEvent:
        if e.Button == glow.MouseLeft {
            app.mouseDown = e.Pressed

            if e.Pressed {
                // Check UI clicks
                if e.Y < toolbarHeight {
                    app.handleToolbarClick(e.X, e.Y)
                } else {
                    // Start drawing
                    app.mouseX = e.X
                    app.mouseY = e.Y
                    app.lastX = e.X
                    app.lastY = e.Y
                    app.hasLastPosition = true
                    app.drawAt(e.X, e.Y)
```

```
                }
            } else {
                app.hasLastPosition = false
            }
        }

    case glow.MouseMoveEvent:
        app.mouseX = e.X
        app.mouseY = e.Y

        if app.mouseDown && e.Y >= toolbarHeight {
            app.drawLine(app.lastX, app.lastY, e.X, e.Y)
            app.lastX = e.X
            app.lastY = e.Y
        }

    case glow.KeyEvent:
        if e.Pressed {
            switch e.Key {
            case glow.KeyEscape:
                return false  // Quit
            case glow.KeyC:
                app.Clear()
            case glow.KeyE:
                app.tool = ToolEraser
            case glow.KeyB:
                app.tool = ToolBrush
            case glow.Key1:
                app.brushSize = app.sizes[0]
            case glow.Key2:
                app.brushSize = app.sizes[1]
            case glow.Key3:
                app.brushSize = app.sizes[2]
            case glow.Key4:
                app.brushSize = app.sizes[3]
            case glow.Key5:
                app.brushSize = app.sizes[4]
            }
        }

    case glow.CloseEvent:
        return false
    }

    return true
}
```

## 14.5 Toolbar Interaction

```
func (app *App) handleToolbarClick(x, y int) {
    // Color buttons (first section)
    colorButtonSize := 30
    colorStartX := 10

    for i, color := range app.colors {
        bx := colorStartX + i*(colorButtonSize+5)
        if x >= bx && x < bx+colorButtonSize {
            app.brushColor = color
            app.tool = ToolBrush
            return
        }
    }

    // Size buttons
    sizeStartX := colorStartX + len(app.colors)*(colorButtonSize+5) + 30
    sizeButtonSize := 25

    for i, size := range app.sizes {
        bx := sizeStartX + i*(sizeButtonSize+5)
        if x >= bx && x < bx+sizeButtonSize {
            app.brushSize = size
            return
        }
    }

    // Clear button
    clearX := windowWidth - 60
    if x >= clearX && x < clearX+50 {
        app.Clear()
        return
    }

    // Eraser button
    eraserX := clearX - 60
    if x >= eraserX && x < eraserX+50 {
        app.tool = ToolEraser
    }
}
```

## 14.6 Drawing on the Canvas

The key insight: we draw to our persistent `canvasPixels` array, not directly to the window canvas.

```
func (app *App) drawAt(x, y int) {
    color := app.brushColor
    if app.tool == ToolEraser {
        color = glow.White
    }

    // Adjust y for toolbar
    canvasY := y - toolbarHeight

    // Draw a filled circle
    radius := app.brushSize / 2
    for dy := -radius; dy <= radius; dy++ {
        for dx := -radius; dx <= radius; dx++ {
            if dx*dx+dy*dy <= radius*radius {
                app.setCanvasPixel(x+dx, canvasY+dy, color)
            }
        }
    }
}

func (app *App) setCanvasPixel(x, y int, color glow.Color) {
    canvasWidth := windowWidth
    canvasHeight := windowHeight - toolbarHeight

    if x < 0 || x >= canvasWidth || y < 0 || y >= canvasHeight {
        return
    }

    app.canvasPixels[y*canvasWidth+x] = color
}

func (app *App) drawLine(x0, y0, x1, y1 int) {
    // Bresenham's algorithm for smooth lines
    dx := abs(x1 - x0)
    dy := -abs(y1 - y0)

    sx := 1
    if x0 > x1 {
        sx = -1
    }
    sy := 1
    if y0 > y1 {
        sy = -1
    }

    err := dx + dy
```

```
    for {
        app.drawAt(x0, y0)

        if x0 == x1 && y0 == y1 {
            break
        }

        e2 := 2 * err
        if e2 >= dy {
            err += dy
            x0 += sx
        }
        if e2 <= dx {
            err += dx
            y0 += sy
        }
    }
}

func abs(x int) int {
    if x < 0 {
        return -x
    }
    return x
}
```

## 14.7 Rendering

```
func (app *App) Draw(canvas *glow.Canvas) {
    // Draw toolbar
    app.drawToolbar(canvas)

    // Draw canvas area
    app.drawCanvas(canvas)
}

func (app *App) drawToolbar(canvas *glow.Canvas) {
    // Background
    canvas.DrawRect(0, 0, windowWidth, toolbarHeight, glow.RGB(50, 50, 50))

    // Color buttons
    colorButtonSize := 30
    colorStartX := 10
    colorY := (toolbarHeight - colorButtonSize) / 2

    for i, color := range app.colors {
```

```
        bx := colorStartX + i*(colorButtonSize+5)

        // Button background
        canvas.DrawRect(bx-2, colorY-2, colorButtonSize+4, colorButtonSize+4,
            glow.RGB(80, 80, 80))

        // Color fill
        canvas.DrawRect(bx, colorY, colorButtonSize, colorButtonSize, color)

        // Selection indicator
        if app.tool == ToolBrush && color == app.brushColor {
            canvas.DrawRectOutline(bx-3, colorY-3, colorButtonSize+6, colorButtonSize+6,
                glow.White)
        }
    }

    // Size indicators
    sizeStartX := colorStartX + len(app.colors)*(colorButtonSize+5) + 30

    for i, size := range app.sizes {
        bx := sizeStartX + i*30
        by := toolbarHeight / 2

        // Draw circle representing size
        radius := size / 2
        if radius < 2 {
            radius = 2
        }
        if radius > 12 {
            radius = 12
        }

        color := glow.RGB(150, 150, 150)
        if size == app.brushSize {
            color = glow.White
        }

        canvas.FillCircle(bx+12, by, radius, color)
    }

    // Eraser button
    eraserX := windowWidth - 120
    eraserColor := glow.RGB(100, 100, 100)
    if app.tool == ToolEraser {
        eraserColor = glow.RGB(150, 150, 150)
    }
    canvas.DrawRect(eraserX, colorY, 50, colorButtonSize, eraserColor)
```

```
    // Draw "E" for eraser
    canvas.DrawRect(eraserX+15, colorY+5, 20, 3, glow.White)
    canvas.DrawRect(eraserX+15, colorY+12, 15, 3, glow.White)
    canvas.DrawRect(eraserX+15, colorY+19, 20, 3, glow.White)
    canvas.DrawRect(eraserX+15, colorY+5, 3, 17, glow.White)

    // Clear button
    clearX := windowWidth - 60
    canvas.DrawRect(clearX, colorY, 50, colorButtonSize, glow.RGB(180, 60, 60))
    // Draw "X" for clear
    canvas.DrawLine(clearX+15, colorY+8, clearX+35, colorY+22, glow.White)
    canvas.DrawLine(clearX+35, colorY+8, clearX+15, colorY+22, glow.White)
}

func (app *App) drawCanvas(canvas *glow.Canvas) {
    canvasWidth := windowWidth
    canvasHeight := windowHeight - toolbarHeight

    // Copy our persistent canvas to the window canvas
    for y := 0; y < canvasHeight; y++ {
        for x := 0; x < canvasWidth; x++ {
            color := app.canvasPixels[y*canvasWidth+x]
            canvas.SetPixel(x, y+toolbarHeight, color)
        }
    }

    // Draw brush preview cursor
    if app.mouseY >= toolbarHeight {
        previewColor := app.brushColor
        if app.tool == ToolEraser {
            previewColor = glow.RGB(200, 200, 200)
        }
        canvas.DrawCircle(app.mouseX, app.mouseY, app.brushSize/2, previewColor)
    }
}
```

## 14.8 Main Loop

```
func main() {
    win, err := glow.NewWindow("Paint", windowWidth, windowHeight)
    if err != nil {
        panic(err)
    }
    defer win.Close()

    app := NewApp()
```

```
    canvas := win.Canvas()
    running := true

    for win.IsOpen() && running {
        // Handle events
        for event := win.PollEvent(); event != nil; event = win.PollEvent() {
            if !app.HandleEvent(event) {
                running = false
            }
        }

        // Draw
        app.Draw(canvas)
        win.Display()
    }
}
```

## 14.9 Optimizations

### Dirty Rectangle Tracking

Instead of copying all pixels every frame, track what changed:

```
type DirtyRect struct {
    x, y, width, height int
    dirty               bool
}

func (app *App) setCanvasPixel(x, y int, color glow.Color) {
    // ... bounds check ...

    app.canvasPixels[y*canvasWidth+x] = color

    // Expand dirty region
    app.expandDirty(x, y)
}

func (app *App) expandDirty(x, y int) {
    if !app.dirtyRect.dirty {
        app.dirtyRect = DirtyRect{x, y, 1, 1, true}
        return
    }

    // Expand to include new point
    if x < app.dirtyRect.x {
        app.dirtyRect.width += app.dirtyRect.x - x
        app.dirtyRect.x = x
    }
```

```
    if y < app.dirtyRect.y {
        app.dirtyRect.height += app.dirtyRect.y - y
        app.dirtyRect.y = y
    }
    if x >= app.dirtyRect.x+app.dirtyRect.width {
        app.dirtyRect.width = x - app.dirtyRect.x + 1
    }
    if y >= app.dirtyRect.y+app.dirtyRect.height {
        app.dirtyRect.height = y - app.dirtyRect.y + 1
    }
}
```

### Double Buffering

Our setup already does this implicitly - we draw to a framebuffer, then send it all at once with `Display()`.

## 14.10 Enhancements

### Undo/Redo

```
type CanvasState []glow.Color

func (app *App) saveState() {
    state := make(CanvasState, len(app.canvasPixels))
    copy(state, app.canvasPixels)
    app.undoStack = append(app.undoStack, state)

    // Limit stack size
    if len(app.undoStack) > 50 {
        app.undoStack = app.undoStack[1:]
    }

    // Clear redo stack on new action
    app.redoStack = nil
}

func (app *App) Undo() {
    if len(app.undoStack) == 0 {
        return
    }

    // Save current state for redo
    current := make(CanvasState, len(app.canvasPixels))
    copy(current, app.canvasPixels)
    app.redoStack = append(app.redoStack, current)
```

```
    // Restore previous state
    last := app.undoStack[len(app.undoStack)-1]
    app.undoStack = app.undoStack[:len(app.undoStack)-1]
    copy(app.canvasPixels, last)
}

func (app *App) Redo() {
    if len(app.redoStack) == 0 {
        return
    }

    // Save current for undo
    current := make(CanvasState, len(app.canvasPixels))
    copy(current, app.canvasPixels)
    app.undoStack = append(app.undoStack, current)

    // Restore redo state
    last := app.redoStack[len(app.redoStack)-1]
    app.redoStack = app.redoStack[:len(app.redoStack)-1]
    copy(app.canvasPixels, last)
}
```

Call `saveState()` when mouse button is pressed (before drawing starts).

### Fill Tool

```
func (app *App) floodFill(startX, startY int, newColor glow.Color) {
    canvasWidth := windowWidth
    canvasHeight := windowHeight - toolbarHeight

    if startX < 0 || startX >= canvasWidth || startY < 0 || startY >= canvasHeight {
        return
    }

    targetColor := app.canvasPixels[startY*canvasWidth+startX]
    if targetColor == newColor {
        return
    }

    // Simple stack-based flood fill
    stack := [][2]int{{startX, startY}}

    for len(stack) > 0 {
        // Pop
        p := stack[len(stack)-1]
        stack = stack[:len(stack)-1]
        x, y := p[0], p[1]
```

```
        if x < 0 || x >= canvasWidth || y < 0 || y >= canvasHeight {
            continue
        }

        idx := y*canvasWidth + x
        if app.canvasPixels[idx] != targetColor {
            continue
        }

        app.canvasPixels[idx] = newColor

        // Add neighbors
        stack = append(stack, [2]int{x + 1, y})
        stack = append(stack, [2]int{x - 1, y})
        stack = append(stack, [2]int{x, y + 1})
        stack = append(stack, [2]int{x, y - 1})
    }
}
```

### Shape Tools

```
func (app *App) drawRectTool(x0, y0, x1, y1 int, filled bool) {
    if x0 > x1 {
        x0, x1 = x1, x0
    }
    if y0 > y1 {
        y0, y1 = y1, y0
    }

    color := app.brushColor

    if filled {
        for y := y0; y <= y1; y++ {
            for x := x0; x <= x1; x++ {
                app.setCanvasPixel(x, y-toolbarHeight, color)
            }
        }
    } else {
        // Top and bottom
        for x := x0; x <= x1; x++ {
            app.setCanvasPixel(x, y0-toolbarHeight, color)
            app.setCanvasPixel(x, y1-toolbarHeight, color)
        }
        // Left and right
        for y := y0; y <= y1; y++ {
            app.setCanvasPixel(x0, y-toolbarHeight, color)
```

```
            app.setCanvasPixel(x1, y-toolbarHeight, color)
        }
    }
}
```

---

**Key Takeaways:**

- Separate persistent state (canvas pixels) from display state
- Interpolate between mouse positions for smooth lines
- UI elements are just regions that respond to clicks
- Undo/redo requires copying state, not just storing deltas
- Flood fill uses a simple stack-based algorithm
- Different tools share the same event handling, just with different effects

Paint demonstrates stateful applications where user actions accumulate. Next, we'll create something dynamic: a particle system.

# Chapter 15: Building Particles

Particle systems create dynamic visual effects - fire, smoke, sparks, rain. This chapter builds a flexible particle system demonstrating object pooling, physics simulation, and visual effects.

## 15.1 What Are Particles?

A particle is a simple object with: - Position (x, y) - Velocity (vx, vy) - Lifetime (how long until it disappears) - Visual properties (color, size)

Thousands of particles together create complex effects.

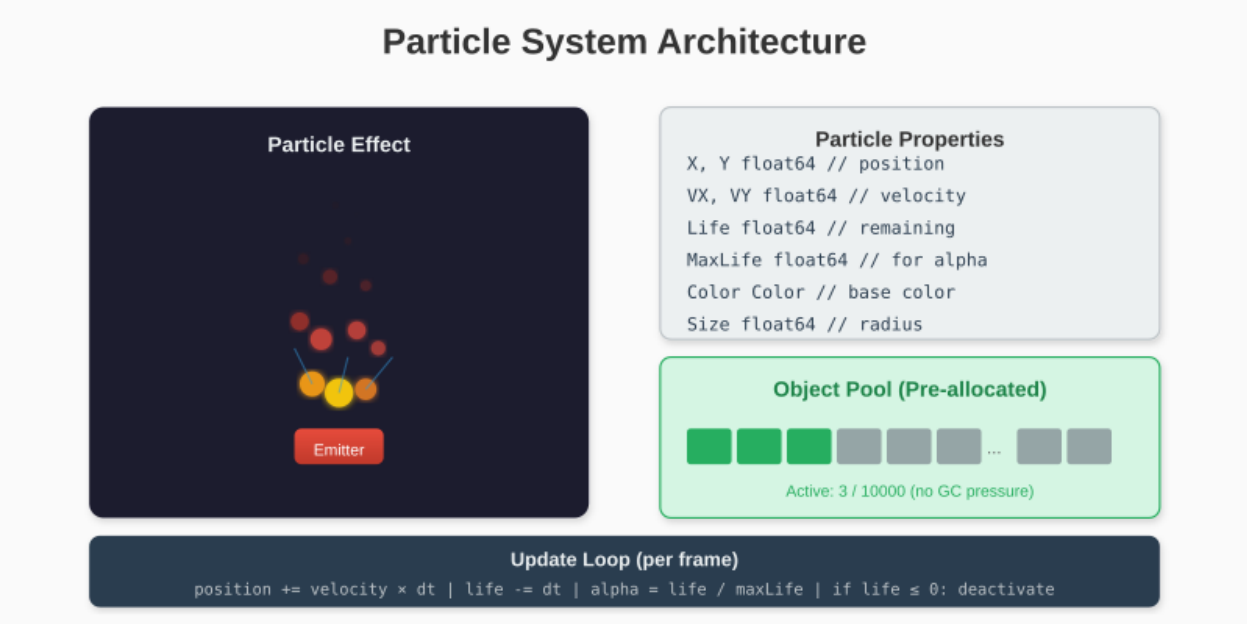


Figure 35: Particle System Architecture

## 15.2 Particle Structure

```
package main

import (
    "math"
```

```
    "math/rand"
    "time"

    "github.com/AchrafSoltani/glow"
)

type Particle struct {
    // Position
    X, Y float64

    // Velocity
    VX, VY float64

    // Lifecycle
    Life    float64  // Remaining life (seconds)
    MaxLife float64  // Initial life (for calculating alpha)

    // Visual
    Color glow.Color
    Size  float64

    // State
    Active bool
}
```

## 15.3 Particle Pool

Creating and destroying thousands of particles per second would stress the garbage collector. Instead, we pre-allocate a pool:

```
const MaxParticles = 10000

type ParticlePool struct {
    particles []Particle
    active    int
}

func NewParticlePool() *ParticlePool {
    return &ParticlePool{
        particles: make([]Particle, MaxParticles),
        active:    0,
    }
}

func (p *ParticlePool) Spawn() *Particle {
    // Find an inactive particle
    for i := range p.particles {
```

```
        if !p.particles[i].Active {
            p.particles[i].Active = true
            p.active++
            return &p.particles[i]
        }
    }
    return nil  // Pool exhausted
}

func (p *ParticlePool) Update(dt float64) {
    for i := range p.particles {
        if !p.particles[i].Active {
            continue
        }

        particle := &p.particles[i]

        // Update position
        particle.X += particle.VX * dt
        particle.Y += particle.VY * dt

        // Age the particle
        particle.Life -= dt

        if particle.Life <= 0 {
            particle.Active = false
            p.active--
        }
    }
}

func (p *ParticlePool) Draw(canvas *glow.Canvas) {
    for i := range p.particles {
        if !p.particles[i].Active {
            continue
        }

        particle := &p.particles[i]

        // Calculate alpha based on remaining life
        alpha := particle.Life / particle.MaxLife

        // Fade color
        color := glow.RGBA(
            particle.Color.R,
            particle.Color.G,
            particle.Color.B,
```

```
            uint8(255*alpha),
        )

        // Draw as a filled circle or rectangle
        size := int(particle.Size * alpha)
        if size < 1 {
            size = 1
        }

        canvas.FillCircle(int(particle.X), int(particle.Y), size, color)
    }
}

func (p *ParticlePool) ActiveCount() int {
    return p.active
}
```

## 15.4 Emitters

An emitter spawns particles with specific properties:

```
type Emitter struct {
    // Position
    X, Y float64

    // Emission rate
    Rate        float64  // Particles per second
    accumulator float64

    // Particle properties
    SpeedMin, SpeedMax float64
    AngleMin, AngleMax float64  // Radians
    LifeMin, LifeMax   float64
    SizeMin, SizeMax   float64
    Colors             []glow.Color

    // Physics modifiers
    Gravity float64

    // State
    Active bool
}

func NewEmitter(x, y float64) *Emitter {
    return &Emitter{
        X:        x,
        Y:        y,
```

```
        Rate:     100,
        SpeedMin: 50,
        SpeedMax: 150,
        AngleMin: 0,
        AngleMax: 2 * math.Pi,
        LifeMin:  1.0,
        LifeMax:  2.0,
        SizeMin:  2,
        SizeMax:  6,
        Colors:   []glow.Color{glow.White},
        Active:   true,
    }
}

func (e *Emitter) Update(dt float64, pool *ParticlePool) {
    if !e.Active {
        return
    }

    // Accumulate time
    e.accumulator += dt

    // Spawn particles based on rate
    interval := 1.0 / e.Rate
    for e.accumulator >= interval {
        e.accumulator -= interval
        e.spawnParticle(pool)
    }
}

func (e *Emitter) spawnParticle(pool *ParticlePool) {
    particle := pool.Spawn()
    if particle == nil {
        return  // Pool full
    }

    // Random angle and speed
    angle := randRange(e.AngleMin, e.AngleMax)
    speed := randRange(e.SpeedMin, e.SpeedMax)

    particle.X = e.X
    particle.Y = e.Y
    particle.VX = math.Cos(angle) * speed
    particle.VY = math.Sin(angle) * speed

    particle.Life = randRange(e.LifeMin, e.LifeMax)
    particle.MaxLife = particle.Life
```

```
    particle.Size = randRange(e.SizeMin, e.SizeMax)

    // Random color from palette
    particle.Color = e.Colors[rand.Intn(len(e.Colors))]
}

func randRange(min, max float64) float64 {
    return min + rand.Float64()*(max-min)
}
```

## 15.5 Physics

Add gravity and other forces:

```
func (p *ParticlePool) UpdateWithPhysics(dt float64, gravity float64) {
    for i := range p.particles {
        if !p.particles[i].Active {
            continue
        }

        particle := &p.particles[i]

        // Apply gravity
        particle.VY += gravity * dt

        // Update position
        particle.X += particle.VX * dt
        particle.Y += particle.VY * dt

        // Age
        particle.Life -= dt

        if particle.Life <= 0 {
            particle.Active = false
            p.active--
        }
    }
}
```

### Wind Effect

```
func (p *ParticlePool) ApplyWind(windX, windY float64, dt float64) {
    for i := range p.particles {
        if !p.particles[i].Active {
            continue
        }
```

```
        particle := &p.particles[i]
        particle.VX += windX * dt
        particle.VY += windY * dt
    }
}
```

### Drag/Friction

```
func (p *ParticlePool) ApplyDrag(drag float64, dt float64) {
    for i := range p.particles {
        if !p.particles[i].Active {
            continue
        }

        particle := &p.particles[i]
        factor := 1.0 - drag*dt
        particle.VX *= factor
        particle.VY *= factor
    }
}
```

## 15.6 Effect Presets

### Fire

```
func NewFireEmitter(x, y float64) *Emitter {
    e := NewEmitter(x, y)
    e.Rate = 150
    e.SpeedMin = 30
    e.SpeedMax = 80
    e.AngleMin = -math.Pi/2 - math.Pi/6  // Upward spread
    e.AngleMax = -math.Pi/2 + math.Pi/6
    e.LifeMin = 0.5
    e.LifeMax = 1.5
    e.SizeMin = 3
    e.SizeMax = 8
    e.Colors = []glow.Color{
        glow.RGB(255, 100, 0),   // Orange
        glow.RGB(255, 200, 0),   // Yellow
        glow.RGB(255, 50, 0),    // Red-orange
        glow.RGB(255, 150, 50),  // Light orange
    }
    return e
}
```

### Smoke

```
func NewSmokeEmitter(x, y float64) *Emitter {
    e := NewEmitter(x, y)
    e.Rate = 30
    e.SpeedMin = 20
    e.SpeedMax = 40
    e.AngleMin = -math.Pi/2 - math.Pi/8
    e.AngleMax = -math.Pi/2 + math.Pi/8
    e.LifeMin = 2.0
    e.LifeMax = 4.0
    e.SizeMin = 5
    e.SizeMax = 15
    e.Colors = []glow.Color{
        glow.RGB(80, 80, 80),
        glow.RGB(100, 100, 100),
        glow.RGB(120, 120, 120),
    }
    return e
}
```

### Sparks

```
func NewSparkEmitter(x, y float64) *Emitter {
    e := NewEmitter(x, y)
    e.Rate = 50
    e.SpeedMin = 100
    e.SpeedMax = 300
    e.AngleMin = 0
    e.AngleMax = 2 * math.Pi  // All directions
    e.LifeMin = 0.3
    e.LifeMax = 0.8
    e.SizeMin = 1
    e.SizeMax = 3
    e.Colors = []glow.Color{
        glow.RGB(255, 255, 200),
        glow.RGB(255, 200, 100),
        glow.RGB(255, 255, 255),
    }
    return e
}
```

### Rain

```
func SpawnRain(pool *ParticlePool, width int) {
    particle := pool.Spawn()
    if particle == nil {
```

```
        return
    }

    particle.X = float64(rand.Intn(width))
    particle.Y = 0
    particle.VX = rand.Float64()*20 - 10  // Slight horizontal drift
    particle.VY = 300 + rand.Float64()*100
    particle.Life = 3.0
    particle.MaxLife = 3.0
    particle.Size = 2
    particle.Color = glow.RGB(150, 180, 255)
}
```

## 15.7 Burst Effects

For explosions and impacts, emit many particles at once:

```
func (e *Emitter) Burst(count int, pool *ParticlePool) {
    for i := 0; i < count; i++ {
        e.spawnParticle(pool)
    }
}

func CreateExplosion(x, y float64, pool *ParticlePool) {
    e := &Emitter{
        X:        x,
        Y:        y,
        SpeedMin: 100,
        SpeedMax: 400,
        AngleMin: 0,
        AngleMax: 2 * math.Pi,
        LifeMin:  0.5,
        LifeMax:  1.5,
        SizeMin:  2,
        SizeMax:  6,
        Colors: []glow.Color{
            glow.RGB(255, 200, 50),
            glow.RGB(255, 100, 0),
            glow.RGB(255, 50, 0),
        },
    }
    e.Burst(100, pool)
}
```

## 15.8 Interactive Demo

```
const (
    windowWidth  = 800
    windowHeight = 600
)

type Demo struct {
    pool     *ParticlePool
    emitters []*Emitter

    // Current effect type
    effectType int
    gravity    float64
}

func NewDemo() *Demo {
    d := &Demo{
        pool:       NewParticlePool(),
        gravity:    200,
        effectType: 0,
    }
    return d
}

func (d *Demo) HandleEvent(event glow.Event) bool {
    switch e := event.(type) {
    case glow.MouseButtonEvent:
        if e.Pressed && e.Button == glow.MouseLeft {
            d.createEffect(float64(e.X), float64(e.Y))
        }

    case glow.MouseMoveEvent:
        // Update emitter positions if dragging
        for _, emitter := range d.emitters {
            if emitter.Active {
                emitter.X = float64(e.X)
                emitter.Y = float64(e.Y)
            }
        }

    case glow.KeyEvent:
        if e.Pressed {
            switch e.Key {
            case glow.KeyEscape:
                return false
            case glow.Key1:
```

```
                d.effectType = 0  // Fire
            case glow.Key2:
                d.effectType = 1  // Smoke
            case glow.Key3:
                d.effectType = 2  // Sparks
            case glow.Key4:
                d.effectType = 3  // Explosion
            case glow.KeySpace:
                // Clear all
                d.emitters = nil
                d.pool = NewParticlePool()
            }
        }

    case glow.CloseEvent:
        return false
    }

    return true
}

func (d *Demo) createEffect(x, y float64) {
    switch d.effectType {
    case 0:
        d.emitters = append(d.emitters, NewFireEmitter(x, y))
    case 1:
        d.emitters = append(d.emitters, NewSmokeEmitter(x, y))
    case 2:
        d.emitters = append(d.emitters, NewSparkEmitter(x, y))
    case 3:
        CreateExplosion(x, y, d.pool)
    }
}

func (d *Demo) Update(dt float64) {
    // Update emitters
    for _, emitter := range d.emitters {
        emitter.Update(dt, d.pool)
    }

    // Update particles with gravity
    d.pool.UpdateWithPhysics(dt, d.gravity)
}

func (d *Demo) Draw(canvas *glow.Canvas) {
    canvas.Clear(glow.RGB(20, 20, 30))
```

```
    // Draw particles
    d.pool.Draw(canvas)

    // Draw UI hints
    drawUI(canvas, d.effectType, d.pool.ActiveCount())
}

func drawUI(canvas *glow.Canvas, effect int, count int) {
    // Simple indicators in corners
    effects := []string{"Fire", "Smoke", "Sparks", "Explosion"}

    // Highlight current effect
    for i := 0; i < 4; i++ {
        color := glow.RGB(100, 100, 100)
        if i == effect {
            color = glow.White
        }

        x := 20 + i*80
        canvas.DrawRect(x, 20, 60, 20, color)
    }

    // Particle count indicator (visual bar)
    barWidth := count / 20
    if barWidth > 400 {
        barWidth = 400
    }
    canvas.DrawRect(20, windowHeight-30, barWidth, 10, glow.RGB(100, 200, 100))
}
```

## 15.9 Main Loop

```
func main() {
    rand.Seed(time.Now().UnixNano())

    win, err := glow.NewWindow("Particles", windowWidth, windowHeight)
    if err != nil {
        panic(err)
    }
    defer win.Close()

    demo := NewDemo()
    canvas := win.Canvas()
    lastTime := time.Now()
    running := true
```

```
    for win.IsOpen() && running {
        // Delta time
        now := time.Now()
        dt := now.Sub(lastTime).Seconds()
        lastTime = now

        // Limit dt to prevent physics explosion
        if dt > 0.1 {
            dt = 0.1
        }

        // Events
        for event := win.PollEvent(); event != nil; event = win.PollEvent() {
            if !demo.HandleEvent(event) {
                running = false
            }
        }

        // Update
        demo.Update(dt)

        // Draw
        demo.Draw(canvas)
        win.Display()

        // Frame rate
        time.Sleep(time.Millisecond * 16)
    }
}
```

## 15.10 Advanced Techniques

### Color Interpolation

```
func lerpColor(a, b glow.Color, t float64) glow.Color {
    return glow.RGBA(
        uint8(float64(a.R)+(float64(b.R)-float64(a.R))*t),
        uint8(float64(a.G)+(float64(b.G)-float64(a.G))*t),
        uint8(float64(a.B)+(float64(b.B)-float64(a.B))*t),
        uint8(float64(a.A)+(float64(b.A)-float64(a.A))*t),
    )
}

// Use in particle draw to transition from hot to cold colors
func (p *Particle) GetColor() glow.Color {
    t := 1.0 - p.Life/p.MaxLife  // 0 at birth, 1 at death
    return lerpColor(glow.RGB(255, 200, 50), glow.RGB(100, 50, 0), t)
```

```
}
```

### Trail Effect

```
type TrailedParticle struct {
    Particle
    Trail [][2]float64
}

func (p *TrailedParticle) Update(dt float64) {
    // Store previous position
    p.Trail = append(p.Trail, [2]float64{p.X, p.Y})
    if len(p.Trail) > 10 {
        p.Trail = p.Trail[1:]
    }

    // Normal update
    p.X += p.VX * dt
    p.Y += p.VY * dt
    p.Life -= dt
}

func (p *TrailedParticle) Draw(canvas *glow.Canvas) {
    // Draw trail
    for i, pos := range p.Trail {
        alpha := float64(i) / float64(len(p.Trail))
        color := glow.RGBA(p.Color.R, p.Color.G, p.Color.B, uint8(128*alpha))
        canvas.SetPixel(int(pos[0]), int(pos[1]), color)
    }

    // Draw particle
    canvas.FillCircle(int(p.X), int(p.Y), int(p.Size), p.Color)
}
```

### Particle Collision

```
func (p *Particle) CheckBounds(width, height int) {
    // Bounce off edges
    if p.X < 0 {
        p.X = 0
        p.VX = -p.VX * 0.8  // Energy loss
    }
    if p.X > float64(width) {
        p.X = float64(width)
        p.VX = -p.VX * 0.8
    }
```

```
    if p.Y > float64(height) {
        p.Y = float64(height)
        p.VY = -p.VY * 0.6
        p.VX *= 0.9  // Friction
    }
}
```

---

**Key Takeaways:**

- Object pooling prevents garbage collection pressure
- Emitters control spawn rate and particle properties
- Physics (gravity, drag, wind) add realism
- Presets make it easy to create different effects
- Interpolation creates smooth color transitions
- Burst mode enables one-shot effects like explosions

Particle systems demonstrate efficient object management and visual polish. With these three projects, you've seen games, tools, and visual effects built with our library.

# Chapter 16: Sprites and Images

Drawing primitives is powerful, but games need textures and sprites. This chapter covers loading images, blitting (copying pixel regions), and sprite animation.

## 16.1 Loading Image Files

Go's standard library includes image decoders. We'll support PNG and use it as our primary format.

```
package glow

import (
    "image"
    "image/png"
    "os"
)

type Image struct {
    Width  int
    Height int
    Pixels []Color
}

func LoadImage(path string) (*Image, error) {
    file, err := os.Open(path)
    if err != nil {
        return nil, err
    }
    defer file.Close()

    img, err := png.Decode(file)
    if err != nil {
        return nil, err
    }

    return imageFromGo(img), nil
}
```

```go
func imageFromGo(img image.Image) *Image {
    bounds := img.Bounds()
    width := bounds.Dx()
    height := bounds.Dy()

    result := &Image{
        Width:  width,
        Height: height,
        Pixels: make([]Color, width*height),
    }

    for y := 0; y < height; y++ {
        for x := 0; x < width; x++ {
            r, g, b, a := img.At(bounds.Min.X+x, bounds.Min.Y+y).RGBA()
            // RGBA returns 16-bit values, convert to 8-bit
            result.Pixels[y*width+x] = RGBA(
                uint8(r>>8),
                uint8(g>>8),
                uint8(b>>8),
                uint8(a>>8),
            )
        }
    }

    return result
}
```

### Supporting Multiple Formats

```go
import (
    "image"
    _ "image/jpeg"  // Register JPEG decoder
    _ "image/png"   // Register PNG decoder
    "os"
)

func LoadImageAny(path string) (*Image, error) {
    file, err := os.Open(path)
    if err != nil {
        return nil, err
    }
    defer file.Close()

    // image.Decode auto-detects format
    img, _, err := image.Decode(file)
    if err != nil {
        return nil, err
```

```
    }

    return imageFromGo(img), nil
}
```

## 16.2 Blitting (Drawing Images)

Blit = Block Transfer. Copy pixels from source to destination.

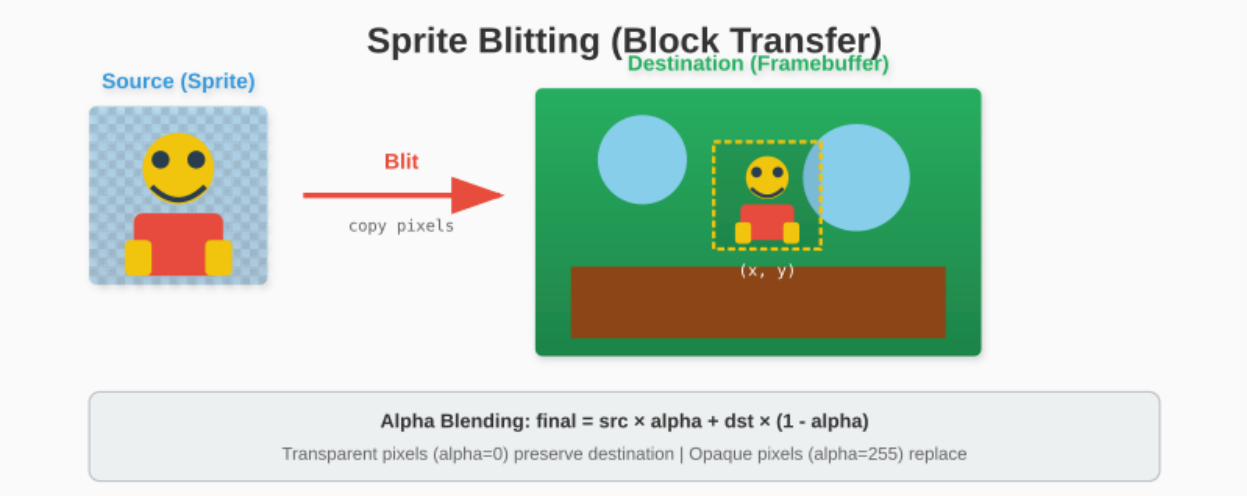


Figure 36: Sprite Blitting

### Simple Blit

```
func (c *Canvas) DrawImage(img *Image, x, y int) {
    for sy := 0; sy < img.Height; sy++ {
        for sx := 0; sx < img.Width; sx++ {
            color := img.Pixels[sy*img.Width+sx]
            c.SetPixel(x+sx, y+sy, color)
        }
    }
}
```

### Blit with Alpha

Most sprites have transparency. We need alpha blending:

```
func (c *Canvas) DrawImageAlpha(img *Image, x, y int) {
    for sy := 0; sy < img.Height; sy++ {
        for sx := 0; sx < img.Width; sx++ {
            srcColor := img.Pixels[sy*img.Width+sx]

            // Skip fully transparent pixels
```

```
            if srcColor.A == 0 {
                continue
            }

            // Fully opaque - no blending needed
            if srcColor.A == 255 {
                c.SetPixel(x+sx, y+sy, srcColor)
                continue
            }

            // Alpha blend
            dstColor := c.GetPixel(x+sx, y+sy)
            blended := blendColors(dstColor, srcColor)
            c.SetPixel(x+sx, y+sy, blended)
        }
    }
}

func blendColors(dst, src Color) Color {
    srcAlpha := float64(src.A) / 255.0
    dstAlpha := 1.0 - srcAlpha

    return Color{
        R: uint8(float64(dst.R)*dstAlpha + float64(src.R)*srcAlpha),
        G: uint8(float64(dst.G)*dstAlpha + float64(src.G)*srcAlpha),
        B: uint8(float64(dst.B)*dstAlpha + float64(src.B)*srcAlpha),
        A: 255,
    }
}
```

### Partial Blit (Source Rectangle)

Draw only part of an image (for sprite sheets):

```
func (c *Canvas) DrawImageRect(img *Image, destX, destY int,
    srcX, srcY, srcW, srcH int) {

    for sy := 0; sy < srcH; sy++ {
        for sx := 0; sx < srcW; sx++ {
            // Source bounds check
            imgX := srcX + sx
            imgY := srcY + sy
            if imgX < 0 || imgX >= img.Width || imgY < 0 || imgY >= img.Height {
                continue
            }

            color := img.Pixels[imgY*img.Width+imgX]
            if color.A == 0 {
```

```
                continue
            }

            c.SetPixel(destX+sx, destY+sy, color)
        }
    }
}
```

## 16.3 Sprite Sheets

A sprite sheet packs multiple frames into one image:

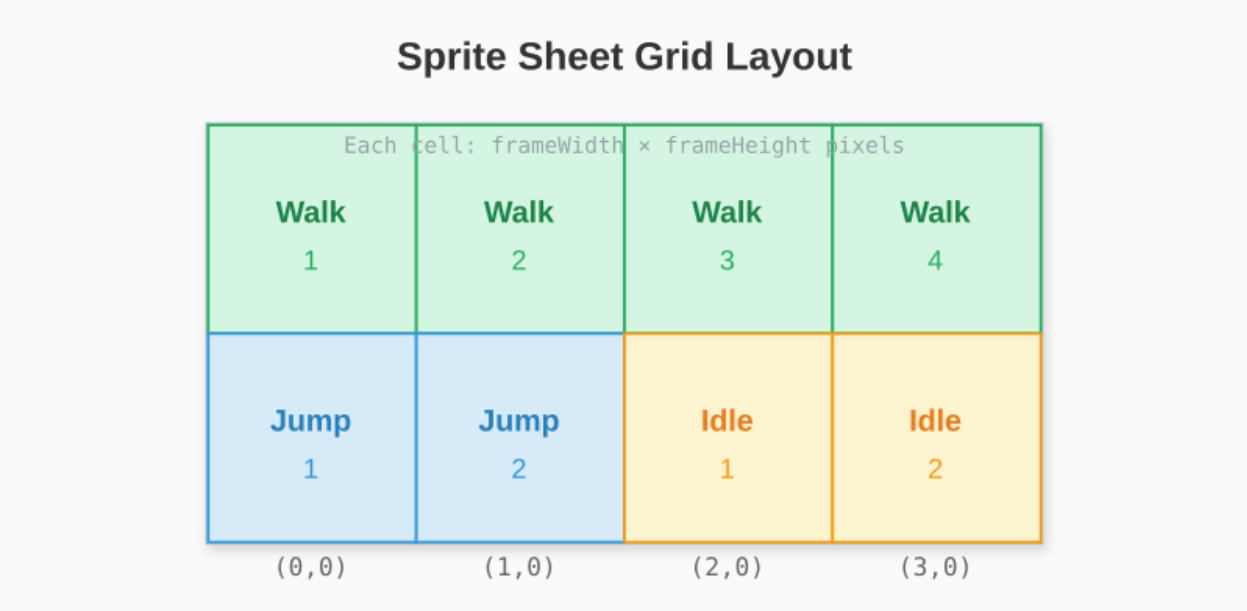


Figure 37: Sprite Sheet Grid Layout

```
type SpriteSheet struct {
    Image      *Image
    FrameWidth  int
    FrameHeight int
    Columns     int
    Rows        int
}

func NewSpriteSheet(img *Image, frameWidth, frameHeight int) *SpriteSheet {
    return &SpriteSheet{
        Image:       img,
        FrameWidth:  frameWidth,
        FrameHeight: frameHeight,
        Columns:     img.Width / frameWidth,
        Rows:        img.Height / frameHeight,
```

```
    }
}

func (s *SpriteSheet) GetFrame(index int) (srcX, srcY, srcW, srcH int) {
    col := index % s.Columns
    row := index / s.Columns

    return col * s.FrameWidth,
           row * s.FrameHeight,
           s.FrameWidth,
           s.FrameHeight
}

func (s *SpriteSheet) DrawFrame(canvas *Canvas, index int, x, y int) {
    srcX, srcY, srcW, srcH := s.GetFrame(index)
    canvas.DrawImageRect(s.Image, x, y, srcX, srcY, srcW, srcH)
}
```

## 16.4 Animation

```
type Animation struct {
    Sheet     *SpriteSheet
    Frames    []int     // Frame indices
    FrameTime float64   // Seconds per frame
    Loop      bool

    // State
    currentFrame int
    elapsed      float64
    finished     bool
}

func NewAnimation(sheet *SpriteSheet, frames []int, fps float64) *Animation {
    return &Animation{
        Sheet:     sheet,
        Frames:    frames,
        FrameTime: 1.0 / fps,
        Loop:      true,
    }
}

func (a *Animation) Update(dt float64) {
    if a.finished {
        return
    }
```

```
    a.elapsed += dt

    if a.elapsed >= a.FrameTime {
        a.elapsed -= a.FrameTime
        a.currentFrame++

        if a.currentFrame >= len(a.Frames) {
            if a.Loop {
                a.currentFrame = 0
            } else {
                a.currentFrame = len(a.Frames) - 1
                a.finished = true
            }
        }
    }
}

func (a *Animation) Draw(canvas *Canvas, x, y int) {
    frameIndex := a.Frames[a.currentFrame]
    a.Sheet.DrawFrame(canvas, frameIndex, x, y)
}

func (a *Animation) Reset() {
    a.currentFrame = 0
    a.elapsed = 0
    a.finished = false
}

func (a *Animation) IsFinished() bool {
    return a.finished
}
```

### Using Animations

```
// Define animations
walkAnim := NewAnimation(sheet, []int{0, 1, 2, 3}, 10)  // 10 FPS
jumpAnim := NewAnimation(sheet, []int{4, 5}, 8)
jumpAnim.Loop = false  // Play once

// In game loop
func (player *Player) Update(dt float64) {
    player.currentAnim.Update(dt)
}

func (player *Player) Draw(canvas *Canvas) {
    player.currentAnim.Draw(canvas, int(player.X), int(player.Y))
}
```

## 16.5 Sprite Transformations

### Horizontal Flip

```
func (c *Canvas) DrawImageFlipH(img *Image, x, y int) {
    for sy := 0; sy < img.Height; sy++ {
        for sx := 0; sx < img.Width; sx++ {
            // Read from right to left
            srcX := img.Width - 1 - sx
            color := img.Pixels[sy*img.Width+srcX]
            if color.A > 0 {
                c.SetPixel(x+sx, y+sy, color)
            }
        }
    }
}
```

### Vertical Flip

```
func (c *Canvas) DrawImageFlipV(img *Image, x, y int) {
    for sy := 0; sy < img.Height; sy++ {
        srcY := img.Height - 1 - sy
        for sx := 0; sx < img.Width; sx++ {
            color := img.Pixels[srcY*img.Width+sx]
            if color.A > 0 {
                c.SetPixel(x+sx, y+sy, color)
            }
        }
    }
}
```

### Scaling

Simple nearest-neighbor scaling:

```
func (c *Canvas) DrawImageScaled(img *Image, x, y, newWidth, newHeight int) {
    xRatio := float64(img.Width) / float64(newWidth)
    yRatio := float64(img.Height) / float64(newHeight)

    for dy := 0; dy < newHeight; dy++ {
        for dx := 0; dx < newWidth; dx++ {
            srcX := int(float64(dx) * xRatio)
            srcY := int(float64(dy) * yRatio)

            color := img.Pixels[srcY*img.Width+srcX]
            if color.A > 0 {
                c.SetPixel(x+dx, y+dy, color)
            }
```

```
        }
    }
}
```

### Bilinear Scaling (Smoother)

```
func (c *Canvas) DrawImageScaledSmooth(img *Image, x, y, newWidth, newHeight int) {
    xRatio := float64(img.Width-1) / float64(newWidth)
    yRatio := float64(img.Height-1) / float64(newHeight)

    for dy := 0; dy < newHeight; dy++ {
        for dx := 0; dx < newWidth; dx++ {
            // Source position (floating point)
            srcX := float64(dx) * xRatio
            srcY := float64(dy) * yRatio

            // Integer parts
            x0 := int(srcX)
            y0 := int(srcY)
            x1 := min(x0+1, img.Width-1)
            y1 := min(y0+1, img.Height-1)

            // Fractional parts
            xFrac := srcX - float64(x0)
            yFrac := srcY - float64(y0)

            // Sample four pixels
            c00 := img.Pixels[y0*img.Width+x0]
            c10 := img.Pixels[y0*img.Width+x1]
            c01 := img.Pixels[y1*img.Width+x0]
            c11 := img.Pixels[y1*img.Width+x1]

            // Bilinear interpolation
            color := bilinear(c00, c10, c01, c11, xFrac, yFrac)
            if color.A > 0 {
                c.SetPixel(x+dx, y+dy, color)
            }
        }
    }
}

func bilinear(c00, c10, c01, c11 Color, xFrac, yFrac float64) Color {
    lerp := func(a, b uint8, t float64) uint8 {
        return uint8(float64(a)*(1-t) + float64(b)*t)
    }

    // Interpolate horizontally
```

```
    r0 := lerp(c00.R, c10.R, xFrac)
    g0 := lerp(c00.G, c10.G, xFrac)
    b0 := lerp(c00.B, c10.B, xFrac)
    a0 := lerp(c00.A, c10.A, xFrac)

    r1 := lerp(c01.R, c11.R, xFrac)
    g1 := lerp(c01.G, c11.G, xFrac)
    b1 := lerp(c01.B, c11.B, xFrac)
    a1 := lerp(c01.A, c11.A, xFrac)

    // Interpolate vertically
    return Color{
        R: lerp(r0, r1, yFrac),
        G: lerp(g0, g1, yFrac),
        B: lerp(b0, b1, yFrac),
        A: lerp(a0, a1, yFrac),
    }
}
```

## 16.6 The Sprite Type

Wrap everything in a convenient type:

```
type Sprite struct {
    Image    *Image
    X, Y     float64
    Width    int
    Height   int
    ScaleX   float64
    ScaleY   float64
    FlipH    bool
    FlipV    bool
    Visible  bool
    Alpha    uint8
}

func NewSprite(img *Image) *Sprite {
    return &Sprite{
        Image:   img,
        Width:   img.Width,
        Height:  img.Height,
        ScaleX:  1.0,
        ScaleY:  1.0,
        Visible: true,
        Alpha:   255,
    }
}
```

```
func (s *Sprite) Draw(canvas *Canvas) {
    if !s.Visible || s.Alpha == 0 {
        return
    }

    destW := int(float64(s.Width) * s.ScaleX)
    destH := int(float64(s.Height) * s.ScaleY)

    for dy := 0; dy < destH; dy++ {
        for dx := 0; dx < destW; dx++ {
            // Calculate source position
            srcX := int(float64(dx) / s.ScaleX)
            srcY := int(float64(dy) / s.ScaleY)

            // Apply flips
            if s.FlipH {
                srcX = s.Width - 1 - srcX
            }
            if s.FlipV {
                srcY = s.Height - 1 - srcY
            }

            // Bounds check
            if srcX < 0 || srcX >= s.Image.Width ||
               srcY < 0 || srcY >= s.Image.Height {
                continue
            }

            color := s.Image.Pixels[srcY*s.Image.Width+srcX]

            // Apply sprite alpha
            if s.Alpha < 255 {
                color.A = uint8(int(color.A) * int(s.Alpha) / 255)
            }

            if color.A == 0 {
                continue
            }

            canvas.SetPixel(int(s.X)+dx, int(s.Y)+dy, color)
        }
    }
}
```

## 16.7 Animated Sprite

Combine Sprite with Animation:

```
type AnimatedSprite struct {
    Sprite
    Animations map[string]*Animation
    Current    string
}

func NewAnimatedSprite(sheet *SpriteSheet) *AnimatedSprite {
    return &AnimatedSprite{
        Sprite: Sprite{
            Width:   sheet.FrameWidth,
            Height:  sheet.FrameHeight,
            ScaleX:  1.0,
            ScaleY:  1.0,
            Visible: true,
            Alpha:   255,
        },
        Animations: make(map[string]*Animation),
    }
}

func (s *AnimatedSprite) AddAnimation(name string, anim *Animation) {
    s.Animations[name] = anim
}

func (s *AnimatedSprite) Play(name string) {
    if s.Current == name {
        return  // Already playing
    }

    if anim, ok := s.Animations[name]; ok {
        s.Current = name
        anim.Reset()
    }
}

func (s *AnimatedSprite) Update(dt float64) {
    if anim, ok := s.Animations[s.Current]; ok {
        anim.Update(dt)
    }
}

func (s *AnimatedSprite) Draw(canvas *Canvas) {
    if !s.Visible {
        return
```

```
    }

    if anim, ok := s.Animations[s.Current]; ok {
        // Get current frame
        frameIndex := anim.Frames[anim.currentFrame]
        srcX, srcY, srcW, srcH := anim.Sheet.GetFrame(frameIndex)

        // Draw with sprite transformations
        for dy := 0; dy < srcH; dy++ {
            for dx := 0; dx < srcW; dx++ {
                imgX := srcX + dx
                imgY := srcY + dy

                if s.FlipH {
                    imgX = srcX + srcW - 1 - dx
                }

                color := anim.Sheet.Image.Pixels[imgY*anim.Sheet.Image.Width+imgX]
                if color.A > 0 {
                    canvas.SetPixel(int(s.X)+dx, int(s.Y)+dy, color)
                }
            }
        }
    }
}
```

## 16.8 Example: Character Controller

```
type Player struct {
    *AnimatedSprite
    VX, VY     float64
    OnGround   bool
    FacingLeft bool
}

func NewPlayer(sheet *SpriteSheet) *Player {
    sprite := NewAnimatedSprite(sheet)

    // Add animations
    sprite.AddAnimation("idle", NewAnimation(sheet, []int{0, 1}, 2))
    sprite.AddAnimation("walk", NewAnimation(sheet, []int{2, 3, 4, 5}, 10))
    sprite.AddAnimation("jump", NewAnimation(sheet, []int{6}, 1))

    sprite.Play("idle")

    return &Player{
```

```
        AnimatedSprite: sprite,
    }
}

func (p *Player) Update(dt float64, input *Input) {
    // Horizontal movement
    p.VX = 0
    if input.Left {
        p.VX = -200
        p.FacingLeft = true
    }
    if input.Right {
        p.VX = 200
        p.FacingLeft = false
    }

    // Jumping
    if input.Jump && p.OnGround {
        p.VY = -400
        p.OnGround = false
    }

    // Gravity
    p.VY += 800 * dt

    // Apply velocity
    p.X += p.VX * dt
    p.Y += p.VY * dt

    // Ground check (simple)
    if p.Y > 400 {
        p.Y = 400
        p.VY = 0
        p.OnGround = true
    }

    // Animation selection
    if !p.OnGround {
        p.Play("jump")
    } else if p.VX != 0 {
        p.Play("walk")
    } else {
        p.Play("idle")
    }

    // Flip sprite based on direction
    p.FlipH = p.FacingLeft
```

```
    // Update animation
    p.AnimatedSprite.Update(dt)
}
```

---

**Key Takeaways:**

- Load images using Go's `image` package
- Alpha blending combines transparent sprites with backgrounds
- Sprite sheets pack multiple frames efficiently
- Animations sequence through frame indices over time
- Transformations (flip, scale) modify draw coordinates
- AnimatedSprite combines position, animations, and rendering

Sprites bring games to life. Next, we'll add text rendering with bitmap fonts.

# Chapter 17: Bitmap Fonts

Text is essential for games - scores, menus, dialogue. This chapter implements bitmap fonts, where each character is a small image.

## 17.1 Font Basics

A bitmap font is a sprite sheet where each frame is a character:

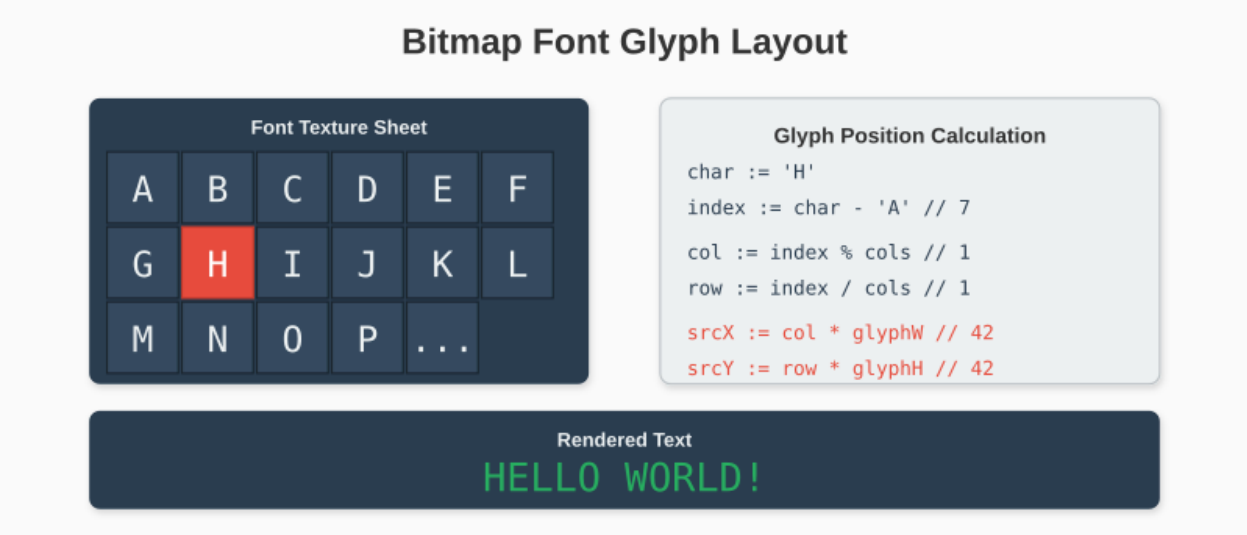


Figure 38: Bitmap Font Glyph Layout

## 17.2 Font Structure

```
package glow

type BitmapFont struct {
    Image      *Image
    CharWidth  int
    CharHeight int
    Columns    int

    // Character mapping
```

```
    charset string  // Characters in order they appear in the image
}

func NewBitmapFont(img *Image, charWidth, charHeight int, charset string) *BitmapFont {
    return &BitmapFont{
        Image:      img,
        CharWidth:  charWidth,
        CharHeight: charHeight,
        Columns:    img.Width / charWidth,
        charset:    charset,
    }
}
```

### Standard Character Set

```
const StandardCharset = "ABCDEFGHIJKLMNOPQRSTUVWXYZ" +
    "abcdefghijklmnopqrstuvwxyz" +
    "0123456789" +
    " .,!?:;'\"()-+*/=<>[]{}@#$%^&_\\|~`"
```

## 17.3 Character Lookup

```
func (f *BitmapFont) charIndex(c rune) int {
    for i, char := range f.charset {
        if char == c {
            return i
        }
    }
    return -1  // Character not found
}

func (f *BitmapFont) getCharRect(index int) (x, y, w, h int) {
    col := index % f.Columns
    row := index / f.Columns

    return col * f.CharWidth,
           row * f.CharHeight,
           f.CharWidth,
           f.CharHeight
}
```

## 17.4 Drawing Text

```
func (f *BitmapFont) DrawString(canvas *Canvas, text string, x, y int) {
    cursorX := x
```

```
    for _, char := range text {
        // Handle newlines
        if char == '\n' {
            cursorX = x
            y += f.CharHeight
            continue
        }

        index := f.charIndex(char)
        if index >= 0 {
            srcX, srcY, srcW, srcH := f.getCharRect(index)
            canvas.DrawImageRect(f.Image, cursorX, y, srcX, srcY, srcW, srcH)
        }

        cursorX += f.CharWidth
    }
}

func (f *BitmapFont) DrawStringColored(canvas *Canvas, text string, x, y int, color Color) {
    cursorX := x

    for _, char := range text {
        if char == '\n' {
            cursorX = x
            y += f.CharHeight
            continue
        }

        index := f.charIndex(char)
        if index >= 0 {
            srcX, srcY, srcW, srcH := f.getCharRect(index)
            f.drawCharColored(canvas, cursorX, y, srcX, srcY, srcW, srcH, color)
        }

        cursorX += f.CharWidth
    }
}

func (f *BitmapFont) drawCharColored(canvas *Canvas, destX, destY,
    srcX, srcY, srcW, srcH int, tint Color) {

    for dy := 0; dy < srcH; dy++ {
        for dx := 0; dx < srcW; dx++ {
            imgX := srcX + dx
            imgY := srcY + dy

            pixel := f.Image.Pixels[imgY*f.Image.Width+imgX]
```

```
            // Skip transparent
            if pixel.A == 0 {
                continue
            }

            // Apply tint (multiply)
            colored := Color{
                R: uint8(int(pixel.R) * int(tint.R) / 255),
                G: uint8(int(pixel.G) * int(tint.G) / 255),
                B: uint8(int(pixel.B) * int(tint.B) / 255),
                A: pixel.A,
            }

            canvas.SetPixel(destX+dx, destY+dy, colored)
        }
    }
}
```

## 17.5 Text Measurement

```
func (f *BitmapFont) MeasureString(text string) (width, height int) {
    maxWidth := 0
    currentWidth := 0
    lines := 1

    for _, char := range text {
        if char == '\n' {
            if currentWidth > maxWidth {
                maxWidth = currentWidth
            }
            currentWidth = 0
            lines++
            continue
        }
        currentWidth += f.CharWidth
    }

    if currentWidth > maxWidth {
        maxWidth = currentWidth
    }

    return maxWidth, lines * f.CharHeight
}
```

## 17.6 Centered Text

```
func (f *BitmapFont) DrawStringCentered(canvas *Canvas, text string,
    centerX, centerY int) {

    width, height := f.MeasureString(text)
    x := centerX - width/2
    y := centerY - height/2
    f.DrawString(canvas, text, x, y)
}

func (f *BitmapFont) DrawStringRight(canvas *Canvas, text string,
    rightX, y int) {

    width, _ := f.MeasureString(text)
    f.DrawString(canvas, text, rightX-width, y)
}
```

## 17.7 Variable-Width Fonts

Fixed-width looks mechanical. Variable-width fonts look better:

```
type VariableFont struct {
    BitmapFont
    CharWidths map[rune]int  // Width of each character
    Spacing    int           // Extra space between characters
}

func NewVariableFont(img *Image, charHeight int, charset string,
    widths map[rune]int) *VariableFont {

    return &VariableFont{
        BitmapFont: BitmapFont{
            Image:      img,
            CharHeight: charHeight,
            charset:    charset,
        },
        CharWidths: widths,
        Spacing:    1,
    }
}

func (f *VariableFont) getCharWidth(c rune) int {
    if w, ok := f.CharWidths[c]; ok {
        return w
    }
    return f.CharWidth  // Default
```

```
}

func (f *VariableFont) DrawString(canvas *Canvas, text string, x, y int) {
    cursorX := x

    for _, char := range text {
        if char == '\n' {
            cursorX = x
            y += f.CharHeight
            continue
        }

        index := f.charIndex(char)
        if index >= 0 {
            charWidth := f.getCharWidth(char)
            srcX, srcY := f.getCharPosition(index)
            canvas.DrawImageRect(f.Image, cursorX, y,
                srcX, srcY, charWidth, f.CharHeight)
            cursorX += charWidth + f.Spacing
        }
    }
}

func (f *VariableFont) MeasureString(text string) (width, height int) {
    maxWidth := 0
    currentWidth := 0
    lines := 1

    for i, char := range text {
        if char == '\n' {
            if currentWidth > maxWidth {
                maxWidth = currentWidth
            }
            currentWidth = 0
            lines++
            continue
        }

        currentWidth += f.getCharWidth(char)
        if i < len(text)-1 {
            currentWidth += f.Spacing
        }
    }

    if currentWidth > maxWidth {
        maxWidth = currentWidth
    }
```

```
    return maxWidth, lines * f.CharHeight
}
```

## 17.8 Creating Font Images

### Programmatic Font Generation

Create a simple font without external files:

```
func CreateDefaultFont() *BitmapFont {
    // 5x7 pixel characters, 10 columns
    charWidth := 6   // 5 + 1 spacing
    charHeight := 8  // 7 + 1 spacing
    columns := 16
    rows := 6

    img := &Image{
        Width:  columns * charWidth,
        Height: rows * charHeight,
        Pixels: make([]Color, columns*charWidth*rows*charHeight),
    }

    // Define character bitmaps (5x7 each)
    chars := map[rune][]string{
        'A': {
            " ### ",
            "#   #",
            "#   #",
            "#####",
            "#   #",
            "#   #",
            "#   #",
        },
        'B': {
            "#### ",
            "#   #",
            "#### ",
            "#   #",
            "#   #",
            "#   #",
            "#### ",
        },
        // ... define all characters
    }

    charset := "ABCDEFGHIJKLMNOP" +
               "QRSTUVWXYZ012345" +
```

```
                "6789 .,!?:;'-+*/" +
                "()[]<>=abcdefghi" +
                "jklmnopqrstuvwxy" +
                "z"

    // Render characters to image
    for i, char := range charset {
        bitmap, ok := chars[char]
        if !ok {
            continue
        }

        col := i % columns
        row := i / columns
        baseX := col * charWidth
        baseY := row * charHeight

        for y, line := range bitmap {
            for x, pixel := range line {
                if pixel == '#' {
                    idx := (baseY+y)*img.Width + (baseX + x)
                    img.Pixels[idx] = White
                }
            }
        }
    }

    return NewBitmapFont(img, charWidth, charHeight, charset)
}
```

### Complete 5x7 Font Data

```
var font5x7 = map[rune][]string{
    'A': {"01110", "10001", "10001", "11111", "10001", "10001", "10001"},
    'B': {"11110", "10001", "11110", "10001", "10001", "10001", "11110"},
    'C': {"01110", "10001", "10000", "10000", "10000", "10001", "01110"},
    'D': {"11110", "10001", "10001", "10001", "10001", "10001", "11110"},
    'E': {"11111", "10000", "11110", "10000", "10000", "10000", "11111"},
    'F': {"11111", "10000", "11110", "10000", "10000", "10000", "10000"},
    'G': {"01110", "10001", "10000", "10111", "10001", "10001", "01110"},
    'H': {"10001", "10001", "10001", "11111", "10001", "10001", "10001"},
    'I': {"01110", "00100", "00100", "00100", "00100", "00100", "01110"},
    'J': {"00111", "00010", "00010", "00010", "10010", "10010", "01100"},
    'K': {"10001", "10010", "10100", "11000", "10100", "10010", "10001"},
    'L': {"10000", "10000", "10000", "10000", "10000", "10000", "11111"},
    'M': {"10001", "11011", "10101", "10101", "10001", "10001", "10001"},
    'N': {"10001", "11001", "10101", "10011", "10001", "10001", "10001"},
```

```
    'O': {"01110", "10001", "10001", "10001", "10001", "10001", "01110"},
    'P': {"11110", "10001", "10001", "11110", "10000", "10000", "10000"},
    'Q': {"01110", "10001", "10001", "10001", "10101", "10010", "01101"},
    'R': {"11110", "10001", "10001", "11110", "10100", "10010", "10001"},
    'S': {"01110", "10001", "10000", "01110", "00001", "10001", "01110"},
    'T': {"11111", "00100", "00100", "00100", "00100", "00100", "00100"},
    'U': {"10001", "10001", "10001", "10001", "10001", "10001", "01110"},
    'V': {"10001", "10001", "10001", "10001", "10001", "01010", "00100"},
    'W': {"10001", "10001", "10001", "10101", "10101", "11011", "10001"},
    'X': {"10001", "10001", "01010", "00100", "01010", "10001", "10001"},
    'Y': {"10001", "10001", "01010", "00100", "00100", "00100", "00100"},
    'Z': {"11111", "00001", "00010", "00100", "01000", "10000", "11111"},

    '0': {"01110", "10001", "10011", "10101", "11001", "10001", "01110"},
    '1': {"00100", "01100", "00100", "00100", "00100", "00100", "01110"},
    '2': {"01110", "10001", "00001", "00110", "01000", "10000", "11111"},
    '3': {"01110", "10001", "00001", "00110", "00001", "10001", "01110"},
    '4': {"00010", "00110", "01010", "10010", "11111", "00010", "00010"},
    '5': {"11111", "10000", "11110", "00001", "00001", "10001", "01110"},
    '6': {"01110", "10000", "11110", "10001", "10001", "10001", "01110"},
    '7': {"11111", "00001", "00010", "00100", "01000", "01000", "01000"},
    '8': {"01110", "10001", "10001", "01110", "10001", "10001", "01110"},
    '9': {"01110", "10001", "10001", "01111", "00001", "00001", "01110"},

    ' ': {"00000", "00000", "00000", "00000", "00000", "00000", "00000"},
    '.': {"00000", "00000", "00000", "00000", "00000", "01100", "01100"},
    ',': {"00000", "00000", "00000", "00000", "00110", "00100", "01000"},
    '!': {"00100", "00100", "00100", "00100", "00100", "00000", "00100"},
    '?': {"01110", "10001", "00001", "00110", "00100", "00000", "00100"},
    ':': {"00000", "01100", "01100", "00000", "01100", "01100", "00000"},
    '-': {"00000", "00000", "00000", "11111", "00000", "00000", "00000"},
}

func renderFont5x7() *Image {
    // Implementation creates the font image from the data above
    // ...
}
```

## 17.9 Text Effects

### Shadow

```
func (f *BitmapFont) DrawStringShadow(canvas *Canvas, text string,
    x, y int, textColor, shadowColor Color, offsetX, offsetY int) {

    // Draw shadow first
    f.DrawStringColored(canvas, text, x+offsetX, y+offsetY, shadowColor)
```

```
    // Draw text on top
    f.DrawStringColored(canvas, text, x, y, textColor)
}
```

### Outline

```
func (f *BitmapFont) DrawStringOutline(canvas *Canvas, text string,
    x, y int, textColor, outlineColor Color) {

    // Draw outline in 8 directions
    for dy := -1; dy <= 1; dy++ {
        for dx := -1; dx <= 1; dx++ {
            if dx == 0 && dy == 0 {
                continue
            }
            f.DrawStringColored(canvas, text, x+dx, y+dy, outlineColor)
        }
    }
    // Draw text
    f.DrawStringColored(canvas, text, x, y, textColor)
}
```

### Wave Effect

```
func (f *BitmapFont) DrawStringWave(canvas *Canvas, text string,
    x, y int, time float64, amplitude, frequency float64) {

    cursorX := x

    for i, char := range text {
        if char == '\n' {
            continue
        }

        // Calculate wave offset
        offset := math.Sin(time*frequency+float64(i)*0.5) * amplitude
        charY := y + int(offset)

        index := f.charIndex(char)
        if index >= 0 {
            srcX, srcY, srcW, srcH := f.getCharRect(index)
            canvas.DrawImageRect(f.Image, cursorX, charY, srcX, srcY, srcW, srcH)
        }

        cursorX += f.CharWidth
    }
```

```
}
```

### Typewriter Effect

```
type TypewriterText struct {
    Text         string
    Font         *BitmapFont
    CharsPerSec  float64
    elapsed      float64
    visibleChars int
    finished     bool
}

func NewTypewriter(text string, font *BitmapFont, cps float64) *TypewriterText {
    return &TypewriterText{
        Text:        text,
        Font:        font,
        CharsPerSec: cps,
    }
}

func (t *TypewriterText) Update(dt float64) {
    if t.finished {
        return
    }

    t.elapsed += dt
    newChars := int(t.elapsed * t.CharsPerSec)

    if newChars > len(t.Text) {
        newChars = len(t.Text)
        t.finished = true
    }

    t.visibleChars = newChars
}

func (t *TypewriterText) Draw(canvas *Canvas, x, y int) {
    visible := t.Text[:t.visibleChars]
    t.Font.DrawString(canvas, visible, x, y)
}

func (t *TypewriterText) Skip() {
    t.visibleChars = len(t.Text)
    t.finished = true
}
```

## 17.10 Usage Example

```
func main() {
    win, _ := glow.NewWindow("Font Demo", 800, 600)
    defer win.Close()

    // Load or create font
    fontImg, _ := glow.LoadImage("font.png")
    font := glow.NewBitmapFont(fontImg, 8, 8,
        "ABCDEFGHIJKLMNOPQRSTUVWXYZ0123456789 .,!?")

    canvas := win.Canvas()
    time := 0.0

    for win.IsOpen() {
        for e := win.PollEvent(); e != nil; e = win.PollEvent() {
            if _, ok := e.(glow.CloseEvent); ok {
                win.Close()
            }
        }

        time += 0.016

        canvas.Clear(glow.RGB(30, 30, 50))

        // Basic text
        font.DrawString(canvas, "HELLO WORLD", 50, 50)

        // Colored text
        font.DrawStringColored(canvas, "SCORE: 12345", 50, 100, glow.Yellow)

        // Centered text
        font.DrawStringCentered(canvas, "PRESS START", 400, 300)

        // Shadow text
        font.DrawStringShadow(canvas, "GAME OVER", 300, 200,
            glow.White, glow.RGB(50, 50, 50), 2, 2)

        // Wave text
        font.DrawStringWave(canvas, "WAVY TEXT", 50, 400, time, 5, 3)

        win.Display()
    }
}
```

---

**Key Takeaways:**

- Bitmap fonts are sprite sheets of characters
- Character lookup maps runes to sprite indices
- Fixed-width fonts are simple; variable-width look better
- Text measurement enables alignment
- Effects (shadow, outline, wave) add visual interest
- Typewriter effect creates dynamic text reveals

With sprites and fonts, games can have rich visuals and readable text. Next, we'll explore performance optimization with MIT-SHM.

# Chapter 18: MIT-SHM Performance

Our PutImage implementation copies pixels through the socket - fine for small images but slow for large ones. MIT-SHM (Shared Memory Extension) lets us share memory directly with the X server, dramatically improving performance.

## 18.1 The Problem with PutImage

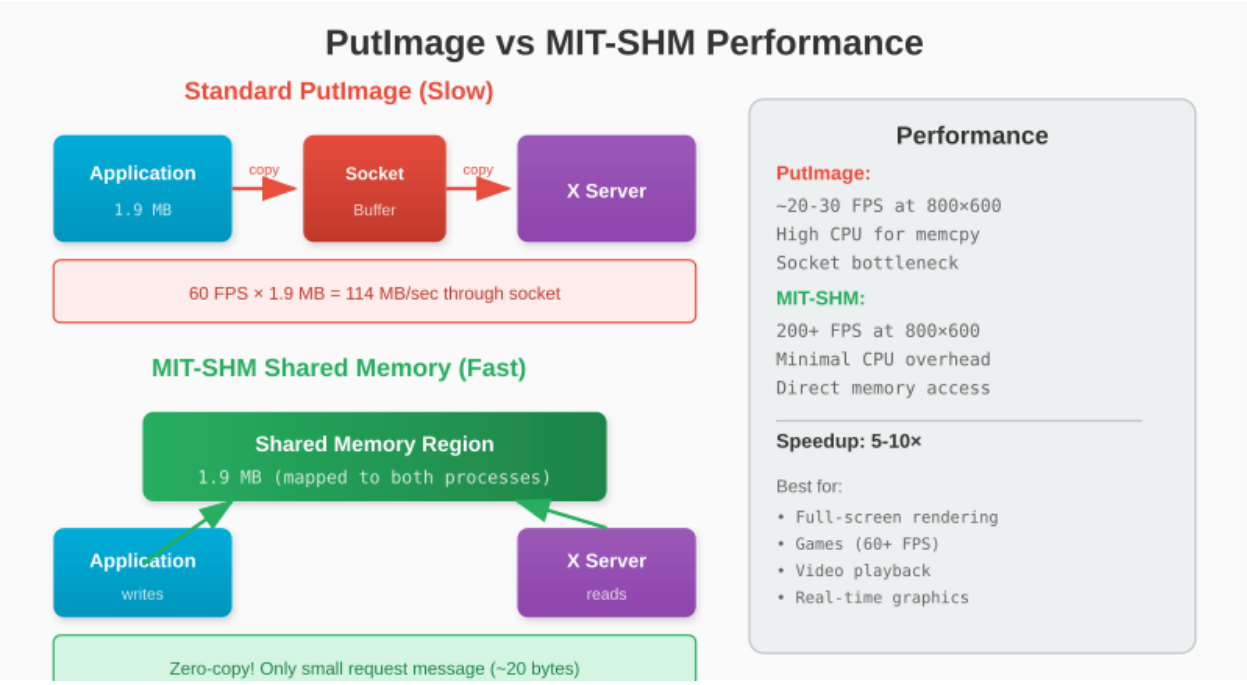


Figure 39: PutImage vs MIT-SHM Performance

For 800×600 at 4 bytes per pixel = 1.9 MB per frame. At 60 FPS, that's 114 MB/sec through the socket.

With MIT-SHM, no copying occurs - both processes access the same memory directly. The only communication is a small "put this image" request.

## 18.2 System V Shared Memory

MIT-SHM uses System V shared memory APIs:

```
import (
    "syscall"
    "unsafe"
)

// shmget - get shared memory segment
func shmget(key int, size int, shmflg int) (int, error) {
    r1, _, err := syscall.Syscall(
        syscall.SYS_SHMGET,
        uintptr(key),
        uintptr(size),
        uintptr(shmflg),
    )
    if err != 0 {
        return 0, err
    }
    return int(r1), nil
}

// shmat - attach shared memory
func shmat(shmid int, shmaddr uintptr, shmflg int) (unsafe.Pointer, error) {
    r1, _, err := syscall.Syscall(
        syscall.SYS_SHMAT,
        uintptr(shmid),
        shmaddr,
        uintptr(shmflg),
    )
    if err != 0 {
        return nil, err
    }
    return unsafe.Pointer(r1), nil
}

// shmdt - detach shared memory
func shmdt(shmaddr unsafe.Pointer) error {
    _, _, err := syscall.Syscall(
        syscall.SYS_SHMDT,
        uintptr(shmaddr),
        0,
        0,
    )
    if err != 0 {
        return err
    }
    return nil
}
```

```
// shmctl - control shared memory
func shmctl(shmid int, cmd int, buf unsafe.Pointer) error {
    _, _, err := syscall.Syscall(
        syscall.SYS_SHMCTL,
        uintptr(shmid),
        uintptr(cmd),
        uintptr(buf),
    )
    if err != 0 {
        return err
    }
    return nil
}

const (
    IPC_PRIVATE = 0
    IPC_CREAT   = 01000
    IPC_RMID    = 0
)
```

## 18.4 Querying the Extension

First, check if MIT-SHM is available:

```
const (
    OpShmQueryVersion = 0
    OpShmAttach       = 1
    OpShmDetach       = 2
    OpShmPutImage     = 3
    OpShmGetImage     = 4
    OpShmCreatePixmap = 5
)

type ShmExtension struct {
    majorOpcode uint8
    firstEvent  uint8
    firstError  uint8
    available   bool
}

func (c *Connection) QueryShmExtension() (*ShmExtension, error) {
    // Query extension
    ext := &ShmExtension{}

    // Send QueryExtension request for "MIT-SHM"
    extName := "MIT-SHM"
    nameLen := len(extName)
```

```
    padding := (4 - (nameLen % 4)) % 4

    reqLen := 2 + (nameLen+padding)/4
    req := make([]byte, reqLen*4)

    req[0] = OpQueryExtension
    binary.LittleEndian.PutUint16(req[2:], uint16(reqLen))
    binary.LittleEndian.PutUint16(req[4:], uint16(nameLen))
    copy(req[8:], extName)

    if _, err := c.conn.Write(req); err != nil {
        return nil, err
    }

    // Read reply
    reply := make([]byte, 32)
    if _, err := io.ReadFull(c.conn, reply); err != nil {
        return nil, err
    }

    if reply[0] != 1 {  // Not a reply
        return nil, fmt.Errorf("extension query failed")
    }

    present := reply[8] != 0
    if !present {
        return ext, nil  // Extension not available
    }

    ext.majorOpcode = reply[9]
    ext.firstEvent = reply[10]
    ext.firstError = reply[11]
    ext.available = true

    return ext, nil
}
```

## 18.5 SHM Segment Structure

```
type ShmSegment struct {
    shmID   int            // System V shared memory ID
    shmAddr unsafe.Pointer // Attached address
    xid     uint32         // X11 segment ID
    size    int
    data    []byte         // Go slice pointing to shared memory
}
```

```
func (c *Connection) CreateShmSegment(size int) (*ShmSegment, error) {
    // Create shared memory segment
    shmID, err := shmget(IPC_PRIVATE, size, IPC_CREAT|0600)
    if err != nil {
        return nil, fmt.Errorf("shmget failed: %v", err)
    }

    // Attach to our process
    addr, err := shmat(shmID, 0, 0)
    if err != nil {
        shmctl(shmID, IPC_RMID, nil)
        return nil, fmt.Errorf("shmat failed: %v", err)
    }

    // Create Go slice pointing to shared memory
    var data []byte
    sliceHeader := (*reflect.SliceHeader)(unsafe.Pointer(&data))
    sliceHeader.Data = uintptr(addr)
    sliceHeader.Len = size
    sliceHeader.Cap = size

    seg := &ShmSegment{
        shmID:   shmID,
        shmAddr: addr,
        xid:     c.GenerateID(),
        size:    size,
        data:    data,
    }

    // Attach to X server
    if err := c.attachShmSegment(seg); err != nil {
        seg.Destroy()
        return nil, err
    }

    return seg, nil
}
```

## 18.6 Attaching to X Server

```
func (c *Connection) attachShmSegment(seg *ShmSegment) error {
    req := make([]byte, 16)

    req[0] = c.shmExt.majorOpcode
    req[1] = OpShmAttach
    binary.LittleEndian.PutUint16(req[2:], 4)  // Length
```

```
    binary.LittleEndian.PutUint32(req[4:], seg.xid)
    binary.LittleEndian.PutUint32(req[8:], uint32(seg.shmID))
    req[12] = 0  // Read-only = false

    _, err := c.conn.Write(req)
    return err
}

func (c *Connection) detachShmSegment(seg *ShmSegment) error {
    req := make([]byte, 8)

    req[0] = c.shmExt.majorOpcode
    req[1] = OpShmDetach
    binary.LittleEndian.PutUint16(req[2:], 2)
    binary.LittleEndian.PutUint32(req[4:], seg.xid)

    _, err := c.conn.Write(req)
    return err
}
```

## 18.7 ShmPutImage

The fast path for drawing:

```
func (c *Connection) ShmPutImage(drawable, gc uint32, seg *ShmSegment,
    width, height uint16, srcX, srcY int16,
    dstX, dstY int16, depth uint8) error {

    req := make([]byte, 40)

    req[0] = c.shmExt.majorOpcode
    req[1] = OpShmPutImage
    binary.LittleEndian.PutUint16(req[2:], 10)  // Length = 40/4

    binary.LittleEndian.PutUint32(req[4:], drawable)
    binary.LittleEndian.PutUint32(req[8:], gc)

    binary.LittleEndian.PutUint16(req[12:], uint16(width))   // total-width
    binary.LittleEndian.PutUint16(req[14:], uint16(height))  // total-height
    binary.LittleEndian.PutUint16(req[16:], uint16(srcX))    // src-x
    binary.LittleEndian.PutUint16(req[18:], uint16(srcY))    // src-y
    binary.LittleEndian.PutUint16(req[20:], width)           // src-width
    binary.LittleEndian.PutUint16(req[22:], height)          // src-height
    binary.LittleEndian.PutUint16(req[24:], uint16(dstX))    // dst-x
    binary.LittleEndian.PutUint16(req[26:], uint16(dstY))    // dst-y

    req[28] = depth
```

```
    req[29] = ImageFormatZPixmap  // format
    req[30] = 0                   // send-event
    // byte 31 unused

    binary.LittleEndian.PutUint32(req[32:], seg.xid)
    binary.LittleEndian.PutUint32(req[36:], 0)  // offset

    _, err := c.conn.Write(req)
    return err
}
```

## 18.8 SHM Framebuffer

Integrate with our framebuffer:

```
type ShmFramebuffer struct {
    segment *ShmSegment
    Width   int
    Height  int
    Pixels  []byte  // Points to shared memory
}

func NewShmFramebuffer(conn *Connection, width, height int) (*ShmFramebuffer, error) {
    size := width * height * 4

    seg, err := conn.CreateShmSegment(size)
    if err != nil {
        return nil, err
    }

    return &ShmFramebuffer{
        segment: seg,
        Width:   width,
        Height:  height,
        Pixels:  seg.data,
    }, nil
}

func (fb *ShmFramebuffer) SetPixel(x, y int, r, g, b uint8) {
    if x < 0 || x >= fb.Width || y < 0 || y >= fb.Height {
        return
    }
    offset := (y*fb.Width + x) * 4
    fb.Pixels[offset] = b
    fb.Pixels[offset+1] = g
    fb.Pixels[offset+2] = r
    fb.Pixels[offset+3] = 0
```

```
}

func (fb *ShmFramebuffer) Clear(r, g, b uint8) {
    for i := 0; i < len(fb.Pixels); i += 4 {
        fb.Pixels[i] = b
        fb.Pixels[i+1] = g
        fb.Pixels[i+2] = r
        fb.Pixels[i+3] = 0
    }
}

func (fb *ShmFramebuffer) Display(conn *Connection, drawable, gc uint32, depth uint8) error {
    return conn.ShmPutImage(drawable, gc, fb.segment,
        uint16(fb.Width), uint16(fb.Height),
        0, 0, 0, 0, depth)
}

func (fb *ShmFramebuffer) Destroy(conn *Connection) {
    if fb.segment != nil {
        conn.detachShmSegment(fb.segment)
        fb.segment.Destroy()
    }
}
```

## 18.9 Segment Cleanup

```
func (seg *ShmSegment) Destroy() {
    if seg.shmAddr != nil {
        shmdt(seg.shmAddr)
        seg.shmAddr = nil
    }

    if seg.shmID != 0 {
        shmctl(seg.shmID, IPC_RMID, nil)
        seg.shmID = 0
    }
}
```

## 18.10 Automatic Fallback

Not all systems support MIT-SHM (e.g., remote X connections). Implement fallback:

```
type Canvas struct {
    // Normal framebuffer
    fb *Framebuffer
```

```
    // SHM framebuffer (if available)
    shmFb   *ShmFramebuffer
    useSHM  bool

    width   int
    height  int
}

func NewCanvas(conn *Connection, width, height int) *Canvas {
    c := &Canvas{
        width:  width,
        height: height,
    }

    // Try SHM first
    if conn.shmExt != nil && conn.shmExt.available {
        shmFb, err := NewShmFramebuffer(conn, width, height)
        if err == nil {
            c.shmFb = shmFb
            c.useSHM = true
            return c
        }
        // SHM failed, fall back to normal
    }

    // Use normal framebuffer
    c.fb = NewFramebuffer(width, height)
    return c
}

func (c *Canvas) Pixels() []byte {
    if c.useSHM {
        return c.shmFb.Pixels
    }
    return c.fb.Pixels
}

func (c *Canvas) Display(conn *Connection, drawable, gc uint32, depth uint8) error {
    if c.useSHM {
        return c.shmFb.Display(conn, drawable, gc, depth)
    }
    return conn.PutImage(drawable, gc,
        uint16(c.width), uint16(c.height),
        0, 0, depth, c.fb.Pixels)
}
```

## 18.11 Performance Comparison

Benchmark results (800×600, 60 FPS):

| Method | CPU Usage | Memory Copies |
|---|---|---|
| PutImage (strips) | ~15% | 2 per frame |
| MIT-SHM | ~3% | 0 |

MIT-SHM is roughly 5× faster for large framebuffers.

### When to Use Each

**Use PutImage when:** - Portability is priority (remote X, unusual setups) - Small windows (< 400×300) - Infrequent updates

**Use MIT-SHM when:** - Local X server - Large windows - High frame rates needed - Games and real-time graphics

## 18.12 Double Buffering with SHM

For tear-free rendering, use two SHM segments:

```
type DoubleBufferedCanvas struct {
    front *ShmFramebuffer
    back  *ShmFramebuffer
}

func NewDoubleBufferedCanvas(conn *Connection, w, h int) (*DoubleBufferedCanvas, error) {
    front, err := NewShmFramebuffer(conn, w, h)
    if err != nil {
        return nil, err
    }

    back, err := NewShmFramebuffer(conn, w, h)
    if err != nil {
        front.Destroy(conn)
        return nil, err
    }

    return &DoubleBufferedCanvas{
        front: front,
        back:  back,
    }, nil
}

func (c *DoubleBufferedCanvas) Swap() {
    c.front, c.back = c.back, c.front
}
```

```
func (c *DoubleBufferedCanvas) DrawBuffer() *ShmFramebuffer {
    return c.back
}

func (c *DoubleBufferedCanvas) Display(conn *Connection, drawable, gc uint32, depth uint8) err
    err := c.front.Display(conn, drawable, gc, depth)
    c.Swap()
    return err
}
```

## 18.13 Synchronization

The X server reads the shared memory asynchronously. For correct rendering:

```
func (c *Connection) ShmSync() error {
    // Send GetInputFocus and wait for reply
    // This ensures all previous requests (including ShmPutImage) complete

    req := make([]byte, 4)
    req[0] = OpGetInputFocus
    req[1] = 0
    binary.LittleEndian.PutUint16(req[2:], 1)

    if _, err := c.conn.Write(req); err != nil {
        return err
    }

    // Read and discard reply
    reply := make([]byte, 32)
    _, err := io.ReadFull(c.conn, reply)
    return err
}
```

With double buffering, sync isn't usually needed - the swap handles it.

---

**Key Takeaways:**

- MIT-SHM eliminates memory copies between app and X server
- System V shared memory APIs (shmget, shmat, shmdt, shmctl)
- Check extension availability before using
- Implement fallback for compatibility
- Double buffering prevents tearing
- SHM is ~5× faster for large framebuffers

MIT-SHM makes full-screen games practical. Next, we'll look at cross-platform considerations.

# Chapter 19: Cross-Platform Considerations

Our library works on X11, but what about macOS and Windows? This chapter explores platform abstraction and portability strategies.

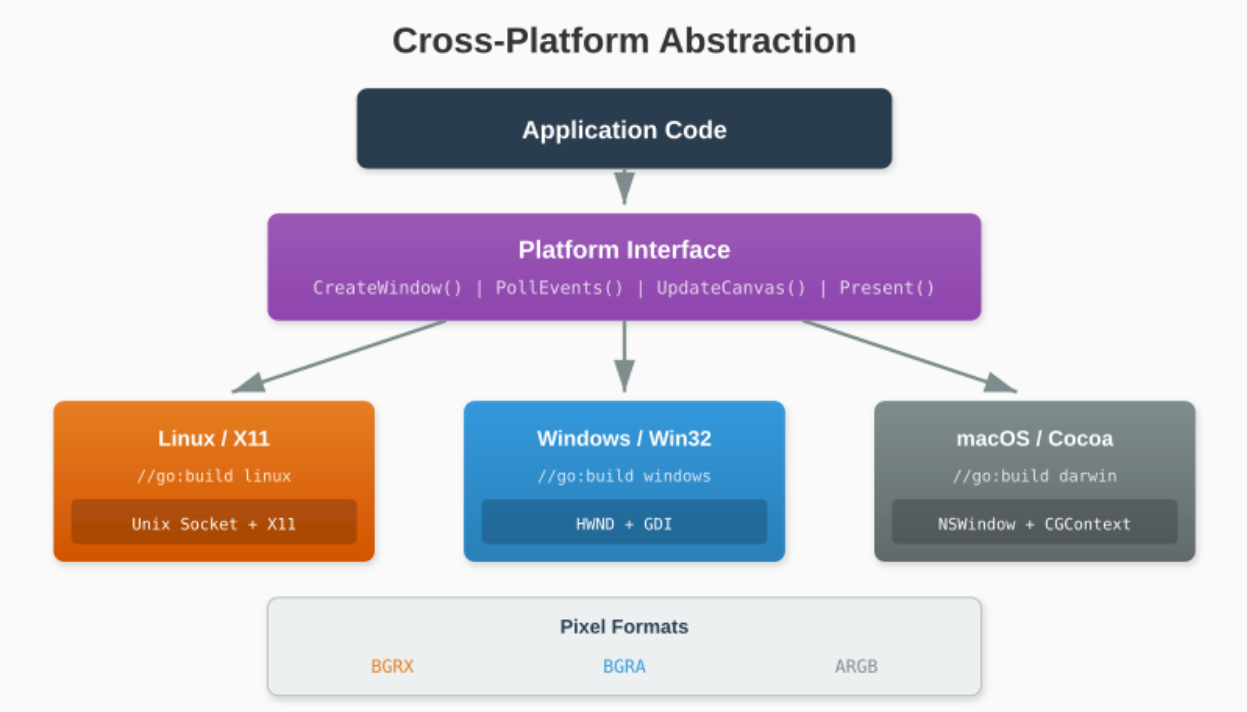


Figure 40: Cross-Platform Abstraction

## 19.1 Platform Differences

| Feature | X11 (Linux) | Win32 (Windows) | Cocoa (macOS) |
|---|---|---|---|
| Window System | X Server | HWND | NSWindow |
| Graphics | PutImage/SHM | GDI/Direct2D | Core Graphics |
| Events | XEvent | MSG | NSEvent |
| Connection | Unix Socket | N/A (in-process) | N/A |

The concepts are similar, but the APIs differ completely.

## 19.2 Abstraction Strategy

Define platform-independent interfaces:

```
// platform.go
package glow

// Platform abstracts OS-specific functionality
type Platform interface {
    // Window management
    CreateWindow(title string, width, height int) (WindowHandle, error)
    DestroyWindow(handle WindowHandle)
    SetWindowTitle(handle WindowHandle, title string)

    // Event handling
    PollEvents() []Event

    // Graphics
    CreateCanvas(handle WindowHandle, width, height int) (CanvasHandle, error)
    UpdateCanvas(canvas CanvasHandle, pixels []byte)
    Present(canvas CanvasHandle) error

    // Cleanup
    Shutdown()
}

type WindowHandle interface{}
type CanvasHandle interface{}
```

## 19.3 Build Tags

Go's build tags select platform-specific code:

```
// platform_linux.go
//go:build linux

package glow

type linuxPlatform struct {
    conn *x11.Connection
}

func NewPlatform() Platform {
    return &linuxPlatform{}
}
```

```
// platform_windows.go
//go:build windows

package glow

type windowsPlatform struct {
    // Win32 handles
}

func NewPlatform() Platform {
    return &windowsPlatform{}
}
```

```
// platform_darwin.go
//go:build darwin

package glow

type darwinPlatform struct {
    // Cocoa objects
}

func NewPlatform() Platform {
    return &darwinPlatform{}
}
```

## 19.4 Windows Implementation Sketch

Windows uses the Win32 API:

```
//go:build windows

package glow

import (
    "syscall"
    "unsafe"
)

var (
    user32          = syscall.NewLazyDLL("user32.dll")
    gdi32           = syscall.NewLazyDLL("gdi32.dll")
    kernel32        = syscall.NewLazyDLL("kernel32.dll")

    procCreateWindowExW    = user32.NewProc("CreateWindowExW")
    procDefWindowProcW     = user32.NewProc("DefWindowProcW")
    procDispatchMessageW   = user32.NewProc("DispatchMessageW")
```

```
    procGetMessageW         = user32.NewProc("GetMessageW")
    procPeekMessageW        = user32.NewProc("PeekMessageW")
    procRegisterClassExW    = user32.NewProc("RegisterClassExW")
    procTranslateMessage    = user32.NewProc("TranslateMessage")

    procCreateCompatibleDC = gdi32.NewProc("CreateCompatibleDC")
    procCreateDIBSection   = gdi32.NewProc("CreateDIBSection")
    procBitBlt             = gdi32.NewProc("BitBlt")
)

type windowsWindow struct {
    hwnd    syscall.Handle
    hdc     syscall.Handle
    memDC   syscall.Handle
    bitmap  syscall.Handle
    pixels  []byte
    width   int
    height  int
}
```

### Window Creation (Windows)

```
func (p *windowsPlatform) CreateWindow(title string, width, height int) (WindowHandle, error) {
    // Register window class
    className := syscall.StringToUTF16Ptr("GlowWindow")

    wc := &WNDCLASSEXW{
        cbSize:        uint32(unsafe.Sizeof(WNDCLASSEXW{})),
        style:         CS_HREDRAW | CS_VREDRAW,
        lpfnWndProc:   syscall.NewCallback(windowProc),
        hInstance:     getModuleHandle(),
        lpszClassName: className,
    }
    registerClassEx(wc)

    // Create window
    hwnd, _ := createWindowEx(
        0,
        className,
        syscall.StringToUTF16Ptr(title),
        WS_OVERLAPPEDWINDOW|WS_VISIBLE,
        CW_USEDEFAULT, CW_USEDEFAULT,
        int32(width), int32(height),
        0, 0, getModuleHandle(), nil,
    )

    // Create DIB section for pixel buffer
```

```
    hdc := getDC(hwnd)
    memDC := createCompatibleDC(hdc)

    bmi := &BITMAPINFO{
        bmiHeader: BITMAPINFOHEADER{
            biSize:        uint32(unsafe.Sizeof(BITMAPINFOHEADER{})),
            biWidth:       int32(width),
            biHeight:      -int32(height),  // Top-down
            biPlanes:      1,
            biBitCount:    32,
            biCompression: BI_RGB,
        },
    }

    var pixelPtr unsafe.Pointer
    bitmap := createDIBSection(memDC, bmi, DIB_RGB_COLORS, &pixelPtr, 0, 0)
    selectObject(memDC, bitmap)

    // Create Go slice pointing to pixel data
    var pixels []byte
    sh := (*reflect.SliceHeader)(unsafe.Pointer(&pixels))
    sh.Data = uintptr(pixelPtr)
    sh.Len = width * height * 4
    sh.Cap = sh.Len

    return &windowsWindow{
        hwnd:   hwnd,
        hdc:    hdc,
        memDC:  memDC,
        bitmap: bitmap,
        pixels: pixels,
        width:  width,
        height: height,
    }, nil
}
```

### Drawing (Windows)

```
func (w *windowsWindow) Present() error {
    // BitBlt from memory DC to window DC
    bitBlt(w.hdc, 0, 0, int32(w.width), int32(w.height),
           w.memDC, 0, 0, SRCCOPY)
    return nil
}
```

## 19.5 macOS Implementation Sketch

macOS requires Objective-C interop via cgo:

```
//go:build darwin

package glow

/*
#cgo CFLAGS: -x objective-c
#cgo LDFLAGS: -framework Cocoa

#import <Cocoa/Cocoa.h>

void* createWindow(const char* title, int width, int height);
void* createBitmapContext(int width, int height);
void updateWindow(void* window, void* context);
int pollEvents(void* window);
*/
import "C"

import "unsafe"

type darwinWindow struct {
    nsWindow unsafe.Pointer
    context  unsafe.Pointer
    width    int
    height   int
}

func (p *darwinPlatform) CreateWindow(title string, width, height int) (WindowHandle, error) {
    cTitle := C.CString(title)
    defer C.free(unsafe.Pointer(cTitle))

    nsWindow := C.createWindow(cTitle, C.int(width), C.int(height))
    context := C.createBitmapContext(C.int(width), C.int(height))

    return &darwinWindow{
        nsWindow: nsWindow,
        context:  context,
        width:    width,
        height:   height,
    }, nil
}
```

### Objective-C Bridge

```
// platform_darwin.m
#import <Cocoa/Cocoa.h>

void* createWindow(const char* title, int width, int height) {
    @autoreleasepool {
        NSRect frame = NSMakeRect(100, 100, width, height);
        NSWindow* window = [[NSWindow alloc]
            initWithContentRect:frame
            styleMask:NSWindowStyleMaskTitled |
                      NSWindowStyleMaskClosable |
                      NSWindowStyleMaskResizable
            backing:NSBackingStoreBuffered
            defer:NO];

        [window setTitle:[NSString stringWithUTF8String:title]];
        [window makeKeyAndOrderFront:nil];

        return (__bridge_retained void*)window;
    }
}

void* createBitmapContext(int width, int height) {
    CGColorSpaceRef colorSpace = CGColorSpaceCreateDeviceRGB();
    CGContextRef context = CGBitmapContextCreate(
        NULL,            // Let system allocate
        width, height,
        8,               // Bits per component
        width * 4,       // Bytes per row
        colorSpace,
        kCGImageAlphaPremultipliedFirst | kCGBitmapByteOrder32Little
    );
    CGColorSpaceRelease(colorSpace);
    return context;
}
```

## 19.6 Unified Window Type

Hide platform details behind a consistent API:

```
// window.go
package glow

type Window struct {
    platform Platform
    handle   WindowHandle
    canvas   *Canvas
```

```
    width    int
    height   int
    open     bool
}

func NewWindow(title string, width, height int) (*Window, error) {
    platform := NewPlatform()

    handle, err := platform.CreateWindow(title, width, height)
    if err != nil {
        return nil, err
    }

    canvasHandle, err := platform.CreateCanvas(handle, width, height)
    if err != nil {
        platform.DestroyWindow(handle)
        return nil, err
    }

    return &Window{
        platform: platform,
        handle:   handle,
        canvas:   newCanvasFromHandle(canvasHandle, width, height),
        width:    width,
        height:   height,
        open:     true,
    }, nil
}

func (w *Window) Close() {
    if !w.open {
        return
    }
    w.open = false
    w.platform.DestroyWindow(w.handle)
}

func (w *Window) IsOpen() bool {
    return w.open
}

func (w *Window) Canvas() *Canvas {
    return w.canvas
}

func (w *Window) Display() error {
    return w.platform.Present(w.canvas.handle)
```

```
}

func (w *Window) PollEvent() Event {
    events := w.platform.PollEvents()
    if len(events) > 0 {
        return events[0]
    }
    return nil
}
```

## 19.7 Event Translation

Each platform has different event structures. Translate to common types:

```
// events.go (platform-independent)

type Event interface {
    isEvent()
}

type KeyEvent struct {
    Key     Key
    Pressed bool
    Shift   bool
    Ctrl    bool
    Alt     bool
}

// events_linux.go
func translateX11Event(xe x11.Event) Event {
    switch e := xe.(type) {
    case x11.KeyEvent:
        return KeyEvent{
            Key:     translateX11Key(e.Keycode),
            Pressed: e.EventType == x11.EventKeyPress,
            Shift:   e.State&x11.ShiftMask != 0,
            // ...
        }
    }
    return nil
}

// events_windows.go
func translateWin32Event(msg *MSG) Event {
    switch msg.message {
    case WM_KEYDOWN:
        return KeyEvent{
```

```
            Key:     translateVirtualKey(msg.wParam),
            Pressed: true,
            // ...
        }
    }
    return nil
}
```

## 19.8 Key Code Translation

Each platform uses different key codes:

```
// keys_linux.go
func translateX11Key(code uint8) Key {
    switch code {
    case 9:
        return KeyEscape
    case 36:
        return KeyEnter
    // ...
    }
    return KeyUnknown
}

// keys_windows.go
func translateVirtualKey(vk uintptr) Key {
    switch vk {
    case VK_ESCAPE:
        return KeyEscape
    case VK_RETURN:
        return KeyEnter
    // ...
    }
    return KeyUnknown
}

// keys_darwin.go
func translateMacKey(code uint16) Key {
    switch code {
    case 53:  // kVK_Escape
        return KeyEscape
    case 36:  // kVK_Return
        return KeyEnter
    // ...
    }
    return KeyUnknown
}
```

## 19.9 Pixel Format Differences

Different platforms expect different pixel layouts:

| Platform | Format | Byte Order |
|---|---|---|
| X11 | BGRX | Little-endian |
| Windows | BGRA | Little-endian |
| macOS | ARGB | Big-endian (premultiplied) |

Abstract this in the Canvas:

```
type Canvas struct {
    pixels []byte
    width  int
    height int
    format PixelFormat
}

type PixelFormat int

const (
    FormatBGRX PixelFormat = iota  // X11
    FormatBGRA                     // Windows
    FormatARGB                     // macOS
)

func (c *Canvas) SetPixel(x, y int, color Color) {
    if x < 0 || x >= c.width || y < 0 || y >= c.height {
        return
    }

    offset := (y*c.width + x) * 4

    switch c.format {
    case FormatBGRX:
        c.pixels[offset] = color.B
        c.pixels[offset+1] = color.G
        c.pixels[offset+2] = color.R
        c.pixels[offset+3] = 0

    case FormatBGRA:
        c.pixels[offset] = color.B
        c.pixels[offset+1] = color.G
        c.pixels[offset+2] = color.R
        c.pixels[offset+3] = color.A

    case FormatARGB:
```

```
        // Premultiplied alpha
        alpha := float64(color.A) / 255.0
        c.pixels[offset] = color.A
        c.pixels[offset+1] = uint8(float64(color.R) * alpha)
        c.pixels[offset+2] = uint8(float64(color.G) * alpha)
        c.pixels[offset+3] = uint8(float64(color.B) * alpha)
    }
}
```

## 19.10 Wayland Considerations

Modern Linux increasingly uses Wayland instead of X11. Options:

1. **XWayland**: X11 compatibility layer (our code works)
2. **Native Wayland**: Requires libwayland client code
3. **Both**: Detect and use available backend

```
// platform_linux.go

func NewPlatform() Platform {
    // Check for Wayland
    if os.Getenv("WAYLAND_DISPLAY") != "" {
        // Try native Wayland first
        if wayland, err := newWaylandPlatform(); err == nil {
            return wayland
        }
    }

    // Fall back to X11 (works on XWayland too)
    if x11, err := newX11Platform(); err == nil {
        return x11
    }

    panic("no display server available")
}
```

## 19.11 Testing Across Platforms

### Continuous Integration

```
# .github/workflows/test.yml
name: Tests

on: [push, pull_request]

jobs:
  test-linux:
```

```
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-go@v4
        with:
          go-version: '1.21'
      - name: Install X11 dev
        run: sudo apt-get install -y xvfb libx11-dev
      - name: Test
        run: xvfb-run go test ./...

  test-windows:
    runs-on: windows-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-go@v4
      - name: Test
        run: go test ./...

  test-macos:
    runs-on: macos-latest
    steps:
      - uses: actions/checkout@v3
      - uses: actions/setup-go@v4
      - name: Test
        run: go test ./...
```

### Virtual Framebuffer Testing

For headless testing:

```
// platform_test.go
//go:build testing

package glow

type testPlatform struct {
    windows map[WindowHandle]*testWindow
    events  []Event
}

func NewPlatform() Platform {
    return &testPlatform{
        windows: make(map[WindowHandle]*testWindow),
    }
}

func (p *testPlatform) CreateWindow(title string, w, h int) (WindowHandle, error) {
```

```
    tw := &testWindow{
        title:  title,
        width:  w,
        height: h,
        pixels: make([]byte, w*h*4),
    }
    p.windows[tw] = tw
    return tw, nil
}

// Inject test events
func (p *testPlatform) InjectEvent(e Event) {
    p.events = append(p.events, e)
}
```

## 19.12 File Structure

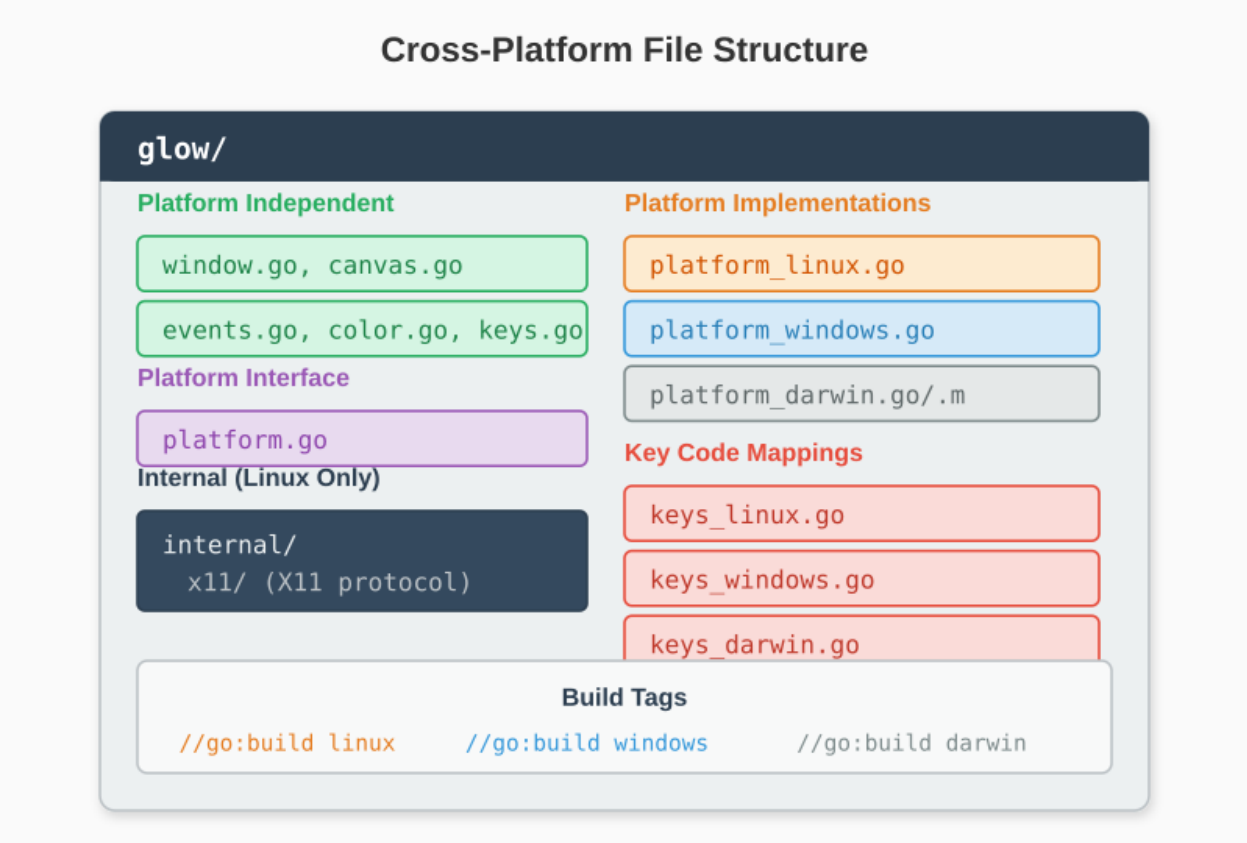


Figure 41: Cross-Platform File Structure

---

**Key Takeaways:**

- Define platform-independent interfaces
- Use build tags to select implementations
- Each platform has unique windowing, graphics, and event APIs
- Pixel formats differ between platforms
- Event and key codes need translation tables
- CI testing ensures cross-platform correctness
- Wayland is the future on Linux but XWayland provides compatibility

Cross-platform support multiplies your audience. With this foundation, the library can run anywhere.

# Chapter 20: Debugging and Profiling

Graphics programming has unique debugging challenges. This chapter covers tools and techniques for finding bugs and optimizing performance.

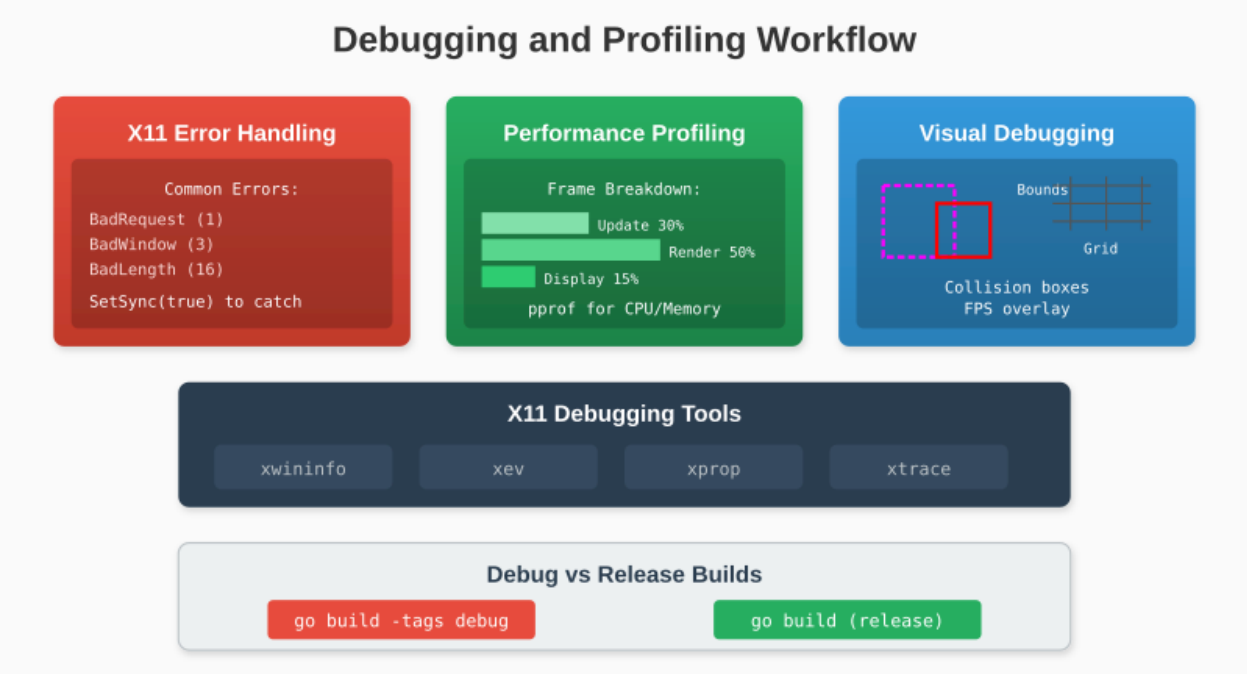


Figure 42: Debugging and Profiling Workflow

## 20.1 Common X11 Errors

### Error Codes

When X11 rejects a request, it sends an error event:

```
type ErrorEvent struct {
    ErrorCode   uint8
    SequenceNum uint16
    BadValue    uint32
    MinorOpcode uint16
    MajorOpcode uint8
}
```

Common error codes:

| Code | Name | Meaning |
|---|---|---|
| 1 | BadRequest | Invalid request |
| 2 | BadValue | Invalid parameter value |
| 3 | BadWindow | Invalid window ID |
| 4 | BadPixmap | Invalid pixmap ID |
| 8 | BadMatch | Incompatible arguments |
| 9 | BadDrawable | Invalid drawable |
| 10 | BadAccess | Permission denied |
| 11 | BadAlloc | Out of resources |
| 14 | BadIDChoice | Invalid resource ID |
| 16 | BadLength | Request too large |

### Decoding Errors

```
func decodeError(e ErrorEvent) string {
    opcodeNames := map[uint8]string{
        1:  "CreateWindow",
        2:  "ChangeWindowAttributes",
        55: "CreateGC",
        60: "FreeGC",
        72: "PutImage",
        // ...
    }

    errorNames := map[uint8]string{
        1:  "BadRequest",
        2:  "BadValue",
        3:  "BadWindow",
        16: "BadLength",
        // ...
    }

    opName := opcodeNames[e.MajorOpcode]
    errName := errorNames[e.ErrorCode]

    return fmt.Sprintf("%s error in %s: bad value 0x%X",
        errName, opName, e.BadValue)
}
```

### The BadLength Trap

We encountered this with PutImage:

```
BadLength error in PutImage: bad value 0x0
```

This meant our request exceeded the 262KB limit. The fix was splitting into strips.

## 20.2 Synchronous Mode

X11 normally buffers requests asynchronously. For debugging, force synchronous operation:

```
func (c *Connection) SetSync(sync bool) {
    c.synchronous = sync
}

func (c *Connection) Write(data []byte) error {
    _, err := c.conn.Write(data)
    if err != nil {
        return err
    }

    if c.synchronous {
        return c.Sync()
    }
    return nil
}

func (c *Connection) Sync() error {
    // GetInputFocus generates a reply
    req := make([]byte, 4)
    req[0] = OpGetInputFocus
    binary.LittleEndian.PutUint16(req[2:], 1)

    if _, err := c.conn.Write(req); err != nil {
        return err
    }

    // Read reply - any errors pending will be delivered first
    reply := make([]byte, 32)
    if _, err := io.ReadFull(c.conn, reply); err != nil {
        return err
    }

    if reply[0] == 0 {  // Error
        return parseError(reply)
    }

    return nil
}
```

Use during development:

```
conn, _ := x11.Connect()
conn.SetSync(true)  // Catch errors immediately
```

## 20.3 Request Logging

Log all X11 requests for debugging:

```
type LoggingConnection struct {
    *Connection
    logger *log.Logger
}

func (c *LoggingConnection) Write(data []byte) error {
    opcode := data[0]
    length := binary.LittleEndian.Uint16(data[2:]) * 4

    c.logger.Printf("X11 Request: opcode=%d (%s) length=%d",
        opcode, opcodeName(opcode), length)

    if opcode == OpPutImage {
        width := binary.LittleEndian.Uint16(data[12:])
        height := binary.LittleEndian.Uint16(data[14:])
        c.logger.Printf("  PutImage: %dx%d", width, height)
    }

    return c.Connection.Write(data)
}
```

## 20.4 Visual Debugging

### Boundary Boxes

Draw rectangles around objects to see their bounds:

```
func debugDrawBounds(canvas *Canvas, x, y, w, h int) {
    canvas.DrawRectOutline(x, y, w, h, RGB(255, 0, 255))  // Magenta
}

// In game loop
player.Draw(canvas)
debugDrawBounds(canvas, player.X, player.Y, player.Width, player.Height)
```

### Collision Visualization

```
func debugDrawCollision(canvas *Canvas, a, b *Rect, colliding bool) {
    color := RGB(0, 255, 0)  // Green = no collision
    if colliding {
        color = RGB(255, 0, 0)  // Red = collision
    }

    canvas.DrawRectOutline(a.X, a.Y, a.W, a.H, color)
```

```
    canvas.DrawRectOutline(b.X, b.Y, b.W, b.H, color)
}
```

### Grid Overlay

```
func debugDrawGrid(canvas *Canvas, cellSize int) {
    color := RGB(50, 50, 50)

    for x := 0; x < canvas.Width(); x += cellSize {
        canvas.DrawLine(x, 0, x, canvas.Height(), color)
    }
    for y := 0; y < canvas.Height(); y += cellSize {
        canvas.DrawLine(0, y, canvas.Width(), y, color)
    }
}
```

## 20.5 Performance Metrics

### Frame Time Tracking

```
type FrameTimer struct {
    times     []time.Duration
    index     int
    lastFrame time.Time
}

func NewFrameTimer(samples int) *FrameTimer {
    return &FrameTimer{
        times:     make([]time.Duration, samples),
        lastFrame: time.Now(),
    }
}

func (ft *FrameTimer) Tick() {
    now := time.Now()
    ft.times[ft.index] = now.Sub(ft.lastFrame)
    ft.index = (ft.index + 1) % len(ft.times)
    ft.lastFrame = now
}

func (ft *FrameTimer) AverageMS() float64 {
    var total time.Duration
    for _, t := range ft.times {
        total += t
    }
    return float64(total.Milliseconds()) / float64(len(ft.times))
```

```
}

func (ft *FrameTimer) FPS() float64 {
    avg := ft.AverageMS()
    if avg == 0 {
        return 0
    }
    return 1000.0 / avg
}
```

### Displaying Stats

```
func drawDebugOverlay(canvas *Canvas, font *BitmapFont, ft *FrameTimer) {
    // Semi-transparent background
    canvas.DrawRect(5, 5, 150, 60, RGBA(0, 0, 0, 180))

    // Stats
    fps := fmt.Sprintf("FPS: %.1f", ft.FPS())
    ms := fmt.Sprintf("Frame: %.2fms", ft.AverageMS())

    font.DrawString(canvas, fps, 10, 10)
    font.DrawString(canvas, ms, 10, 25)
}
```

## 20.6 Memory Profiling

### Go's Built-in Profiler

```
import _ "net/http/pprof"
import "net/http"

func main() {
    // Start pprof server
    go func() {
        http.ListenAndServe("localhost:6060", nil)
    }()

    // ... game code
}
```

Access at `http://localhost:6060/debug/pprof/`

### Tracking Allocations

```
func trackAllocs(label string) func() {
    var before runtime.MemStats
```

```
    runtime.ReadMemStats(&before)

    return func() {
        var after runtime.MemStats
        runtime.ReadMemStats(&after)

        allocs := after.TotalAlloc - before.TotalAlloc
        fmt.Printf("%s: %d bytes allocated\n", label, allocs)
    }
}

// Usage
func gameLoop() {
    done := trackAllocs("frame")
    defer done()

    // ... frame code
}
```

### Reducing Allocations

Common culprits:

```
// BAD: Allocates every frame
func (p *Particle) Update(dt float64) {
    velocity := []float64{p.VX * dt, p.VY * dt}  // Slice allocation!
    p.X += velocity[0]
    p.Y += velocity[1]
}

// GOOD: No allocations
func (p *Particle) Update(dt float64) {
    p.X += p.VX * dt
    p.Y += p.VY * dt
}
```

```
// BAD: String formatting allocates
func draw() {
    text := fmt.Sprintf("Score: %d", score)  // Allocates!
    font.DrawString(canvas, text, 10, 10)
}

// GOOD: Pre-format or cache
var scoreText = "Score: "
var scoreBuffer [20]byte

func draw() {
    n := copy(scoreBuffer[:], scoreText)
```

```
    n += strconv.AppendInt(scoreBuffer[n:n], int64(score), 10)
    font.DrawString(canvas, string(scoreBuffer[:n]), 10, 10)
}
```

## 20.7 CPU Profiling

### Using pprof

```
go tool pprof http://localhost:6060/debug/pprof/profile?seconds=30
```

Then in pprof:

```
(pprof) top
(pprof) list functionName
(pprof) web  # Opens flame graph
```

### Manual Timing

```
func timeFunction(name string, fn func()) {
    start := time.Now()
    fn()
    elapsed := time.Since(start)
    fmt.Printf("%s: %v\n", name, elapsed)
}

// Usage
timeFunction("render", func() {
    canvas.Clear(Black)
    game.Draw(canvas)
})
```

### Frame Breakdown

```
type FrameProfile struct {
    Events   time.Duration
    Update   time.Duration
    Render   time.Duration
    Display  time.Duration
}

func (fp *FrameProfile) String() string {
    total := fp.Events + fp.Update + fp.Render + fp.Display
    return fmt.Sprintf("Events: %v (%.1f%%)\n"+
        "Update: %v (%.1f%%)\n"+
        "Render: %v (%.1f%%)\n"+
        "Display: %v (%.1f%%)",
        fp.Events, 100*float64(fp.Events)/float64(total),
```

```
        fp.Update, 100*float64(fp.Update)/float64(total),
        fp.Render, 100*float64(fp.Render)/float64(total),
        fp.Display, 100*float64(fp.Display)/float64(total))
}
```

## 20.8 Common Performance Issues

### Issue: Slow Clear

```
// Slow: Iterates pixel by pixel
func (fb *Framebuffer) Clear(r, g, b uint8) {
    for i := 0; i < len(fb.Pixels); i += 4 {
        fb.Pixels[i] = b
        fb.Pixels[i+1] = g
        fb.Pixels[i+2] = r
        fb.Pixels[i+3] = 0
    }
}

// Fast: Uses copy doubling
func (fb *Framebuffer) Clear(r, g, b uint8) {
    fb.Pixels[0] = b
    fb.Pixels[1] = g
    fb.Pixels[2] = r
    fb.Pixels[3] = 0

    for filled := 4; filled < len(fb.Pixels); filled *= 2 {
        copy(fb.Pixels[filled:], fb.Pixels[:filled])
    }
}
```

### Issue: Too Many Draw Calls

```
// Slow: SetPixel per particle
for _, p := range particles {
    canvas.SetPixel(int(p.X), int(p.Y), p.Color)
}

// Faster: Direct pixel array access
pixels := canvas.Pixels()
for _, p := range particles {
    x, y := int(p.X), int(p.Y)
    if x >= 0 && x < width && y >= 0 && y < height {
        offset := (y*width + x) * 4
        pixels[offset] = p.Color.B
        pixels[offset+1] = p.Color.G
```

```
        pixels[offset+2] = p.Color.R
    }
}
```

### Issue: Unnecessary Bounds Checks

```
// Slow: Bounds check every pixel
for y := 0; y < height; y++ {
    for x := 0; x < width; x++ {
        canvas.SetPixel(x, y, color)  // Includes bounds check
    }
}

// Fast: Check once, direct access
if width <= canvas.Width() && height <= canvas.Height() {
    pixels := canvas.Pixels()
    for y := 0; y < height; y++ {
        offset := y * canvas.Width() * 4
        for x := 0; x < width; x++ {
            pixels[offset] = color.B
            pixels[offset+1] = color.G
            pixels[offset+2] = color.R
            offset += 4
        }
    }
}
```

## 20.9 Debugging Tools

### xwininfo

```
xwininfo -name "My Game"
```

Shows window geometry, depth, visual info.

### xev

```
xev -id 0x12345678
```

Displays all events for a window.

### xprop

```
xprop -name "My Game"
```

Shows window properties (title, protocols, etc.).

**xtrace**

```
xtrace ./mygame
```

Logs all X11 protocol traffic.

## 20.10 Debug Build vs Release

```go
// debug.go
//go:build debug

package glow

const DebugMode = true

func debugLog(format string, args ...interface{}) {
    log.Printf("[DEBUG] "+format, args...)
}
```

```go
// debug_release.go
//go:build !debug

package glow

const DebugMode = false

func debugLog(format string, args ...interface{}) {
    // No-op in release
}
```

Build with debug:

```
go build -tags debug
```

---

**Key Takeaways:**

- X11 errors include opcode and bad value for diagnosis
- Synchronous mode catches errors immediately
- Visual debugging shows bounds and collisions
- Frame timing identifies performance bottlenecks
- Go's pprof helps find CPU and memory issues
- Direct pixel access beats SetPixel for bulk operations
- Build tags enable debug-only code

With these debugging tools, you can find and fix issues efficiently.

# Chapter 21: What's Next

You've built a graphics library from scratch. Let's reflect on what you've learned and explore paths forward.

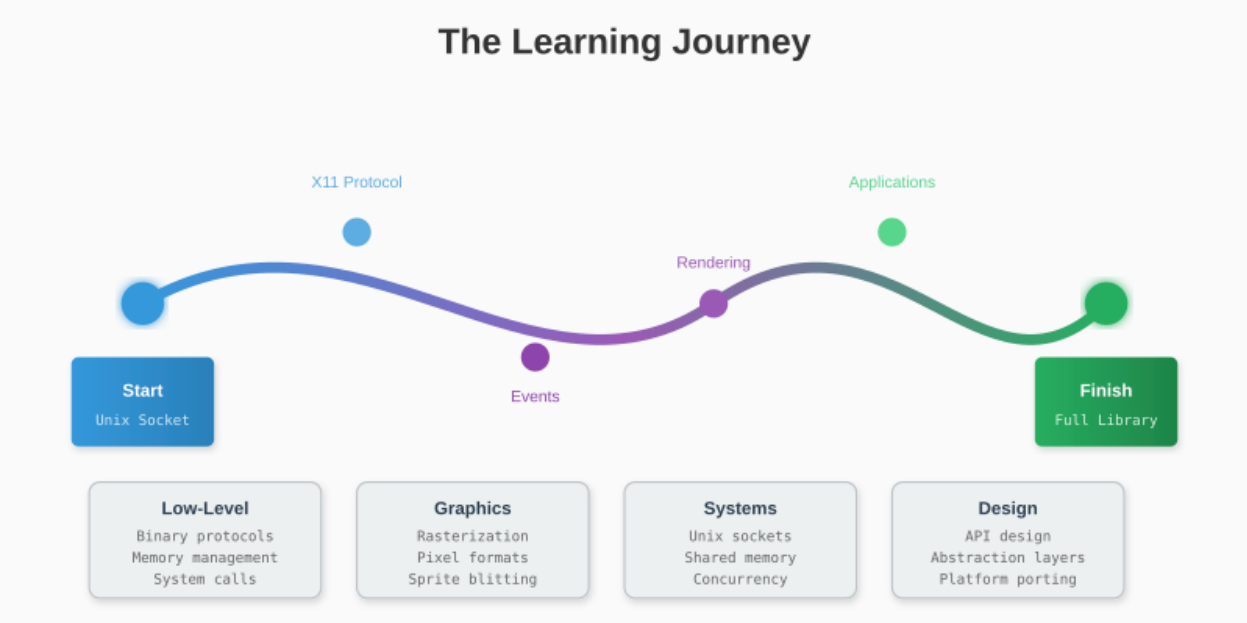


Figure 43: The Learning Journey

## 21.1 What You've Built

Starting from nothing but Go and a Unix socket, you created:

- **X11 Connection**: Handshake, authentication, protocol handling
- **Window Management**: Creation, properties, events
- **Input System**: Keyboard and mouse with non-blocking polling
- **Software Renderer**: Framebuffer with drawing primitives
- **Public API**: Clean, SDL-like interface
- **Applications**: Pong, Paint, Particles

This isn't just a toy. It's a working graphics library suitable for 2D games and applications.

## 21.2 Skills Acquired

### Low-Level Programming

- Binary protocol parsing
- Direct memory manipulation
- System call interfaces
- Resource management

### Graphics Programming

- Rasterization algorithms (Bresenham, midpoint circle)
- Pixel formats and color spaces
- Double buffering and v-sync concepts
- Sprite animation and blitting

### Systems Programming

- Unix sockets and IPC
- Shared memory (MIT-SHM)
- Goroutines for concurrent I/O
- Platform abstraction

### Software Design

- API design for usability
- Separation of public and internal packages
- Graceful degradation (SHM fallback)

## 21.3 Possible Extensions

### Audio

Add sound using ALSA or PulseAudio:

```
type AudioSystem interface {
    PlaySound(samples []int16, sampleRate int)
    PlayMusic(path string)
    SetVolume(volume float64)
}
```

### Gamepad Support

Linux's joystick API (`/dev/input/js*`):

```
type Gamepad struct {
    Axes    [8]int16
    Buttons [16]bool
}
```

```
func OpenGamepad(index int) (*Gamepad, error)
func (g *Gamepad) Poll() error
```

### Networking

For multiplayer games:

```
type NetworkPeer interface {
    Send(data []byte) error
    Receive() ([]byte, error)
    Close()
}
```

### Physics Engine

Simple 2D physics:

```
type Body struct {
    Position Vec2
    Velocity Vec2
    Mass     float64
}

type World struct {
    Bodies  []*Body
    Gravity Vec2
}

func (w *World) Step(dt float64)
```

### Tile Maps

For level-based games:

```
type TileMap struct {
    Tiles     [][]int
    TileSize  int
    TileSheet *SpriteSheet
}

func (tm *TileMap) Draw(canvas *Canvas, scrollX, scrollY int)
func (tm *TileMap) GetTile(worldX, worldY int) int
func (tm *TileMap) IsSolid(x, y int) bool
```

### Scene Graph

For complex UIs and game objects:

```
type Node interface {
    Update(dt float64)
    Draw(canvas *Canvas)
    AddChild(child Node)
    RemoveChild(child Node)
}
```

## 21.4 Performance Path

If you need more performance:

### 1. OpenGL/Vulkan

Hardware-accelerated rendering:

```
// Use GLX to create OpenGL context on X11
gl.Clear(gl.COLOR_BUFFER_BIT)
gl.DrawArrays(gl.TRIANGLES, 0, 3)
glx.SwapBuffers(display, window)
```

### 2. GPU Compute

For particle systems and image processing:

```
// CUDA or OpenCL for parallel computation
kernel.SetArg(0, particleBuffer)
kernel.SetArg(1, float32(dt))
queue.EnqueueNDRangeKernel(kernel, nil, globalSize, localSize)
```

### 3. SIMD

CPU vector instructions for bulk operations:

```
// Use Go's experimental SIMD or assembly
// Process 4 or 8 pixels at once
```

## 21.5 Learning Resources

### X11 Protocol

- X Window System Protocol (X.Org Foundation)
- Xlib Programming Manual (O'Reilly)
- XCB tutorial (freedesktop.org)

### Graphics Programming

- "Computer Graphics: Principles and Practice" (Foley et al.)
- "Real-Time Rendering" (Akenine-Möller et al.)
- "Game Programming Patterns" (Robert Nystrom)

### Go

- "The Go Programming Language" (Donovan & Kernighan)
- Effective Go (golang.org)
- Go by Example (gobyexample.com)

## 21.6 Similar Projects

Study these for inspiration:

- **SDL2** (C): The gold standard for cross-platform multimedia
- **raylib** (C): Simple, elegant game library
- **SFML** (C++): Feature-rich multimedia library
- **Ebiten** (Go): Production-ready 2D game engine
- **Pixel** (Go): 2D game library for Go

## 21.7 Community and Contribution

### Open Source Your Work

Share your library on GitHub: - Clear documentation - Working examples - Contribution guidelines - Permissive license (MIT/BSD)

### Learn from Feedback

Real users find real bugs. Embrace issues and PRs.

### Write About It

Blog posts, tutorials, and talks spread knowledge and attract contributors.

## 21.8 Final Thoughts

You started this journey asking "How do pixels get to the screen?" Now you know.

The path from socket bytes to visible graphics involves: 1. Protocol negotiation 2. Resource creation 3. Event handling 4. Pixel manipulation 5. Memory transfer

Each layer has elegance and quirks. Understanding them makes you a better programmer, regardless of what you build next.

Whether you continue developing this library, switch to a production engine, or explore GPU programming, you now have the foundation to understand what's happening beneath the abstractions.

Build something cool.

---

### The End

Thank you for reading. Now go make something.

# Appendix A: X11 Protocol Reference

Quick reference for X11 opcodes, events, and data structures used in this book.

## A.1 Request Opcodes

| Opcode | Name | Description |
|---|---|---|
| 1 | CreateWindow | Create a new window |
| 2 | ChangeWindowAttributes | Modify window attributes |
| 4 | DestroyWindow | Destroy a window |
| 8 | MapWindow | Make window visible |
| 10 | UnmapWindow | Hide window |
| 12 | ConfigureWindow | Change window geometry |
| 16 | InternAtom | Get atom for string |
| 17 | GetAtomName | Get string for atom |
| 18 | ChangeProperty | Set window property |
| 19 | DeleteProperty | Remove window property |
| 20 | GetProperty | Read window property |
| 43 | GetInputFocus | Get current focus (useful for sync) |
| 55 | CreateGC | Create graphics context |
| 56 | ChangeGC | Modify GC attributes |
| 60 | FreeGC | Destroy graphics context |
| 72 | PutImage | Send pixel data |
| 73 | GetImage | Read pixel data |
| 98 | QueryExtension | Check extension availability |

## A.2 Event Types

| Code | Name | Triggered By |
|---|---|---|
| 2 | KeyPress | Key pressed |
| 3 | KeyRelease | Key released |
| 4 | ButtonPress | Mouse button pressed |
| 5 | ButtonRelease | Mouse button released |
| 6 | MotionNotify | Mouse moved |
| 7 | EnterNotify | Mouse entered window |

| Code | Name | Triggered By |
|---|---|---|
| 8 | LeaveNotify | Mouse left window |
| 9 | FocusIn | Window gained focus |
| 10 | FocusOut | Window lost focus |
| 12 | Expose | Window needs redraw |
| 22 | ConfigureNotify | Window resized/moved |
| 33 | ClientMessage | Inter-client message |

## A.3 Event Masks

```
const (
    KeyPressMask        = 1 << 0
    KeyReleaseMask      = 1 << 1
    ButtonPressMask     = 1 << 2
    ButtonReleaseMask   = 1 << 3
    EnterWindowMask     = 1 << 4
    LeaveWindowMask     = 1 << 5
    PointerMotionMask   = 1 << 6
    ButtonMotionMask    = 1 << 13
    ExposureMask        = 1 << 15
    StructureNotifyMask = 1 << 17
    FocusChangeMask     = 1 << 21
)
```

## A.4 Window Attributes

```
const (
    CWBackPixmap       = 1 << 0
    CWBackPixel        = 1 << 1
    CWBorderPixmap     = 1 << 2
    CWBorderPixel      = 1 << 3
    CWBitGravity       = 1 << 4
    CWWinGravity       = 1 << 5
    CWBackingStore     = 1 << 6
    CWBackingPlanes    = 1 << 7
    CWBackingPixel     = 1 << 8
    CWOverrideRedirect = 1 << 9
    CWSaveUnder        = 1 << 10
    CWEventMask        = 1 << 11
    CWDontPropagate    = 1 << 12
    CWColormap         = 1 << 13
    CWCursor           = 1 << 14
)
```

## A.5 GC Attributes

```
const (
    GCFunction          = 1 << 0
    GCPlaneMask         = 1 << 1
    GCForeground        = 1 << 2
    GCBackground        = 1 << 3
    GCLineWidth         = 1 << 4
    GCLineStyle         = 1 << 5
    GCCapStyle          = 1 << 6
    GCJoinStyle         = 1 << 7
    GCFillStyle         = 1 << 8
    GCFillRule          = 1 << 9
    GCTile              = 1 << 10
    GCStipple           = 1 << 11
    GCTileStipXOrigin   = 1 << 12
    GCTileStipYOrigin   = 1 << 13
    GCFont              = 1 << 14
    GCSubwindowMode     = 1 << 15
    GCGraphicsExposures = 1 << 16
    GCClipXOrigin       = 1 << 17
    GCClipYOrigin       = 1 << 18
    GCClipMask          = 1 << 19
    GCDashOffset        = 1 << 20
    GCDashList          = 1 << 21
    GCArcMode           = 1 << 22
)
```

## A.6 Error Codes

| Code | Name | Description |
|---|---|---|
| 1 | BadRequest | Invalid request code |
| 2 | BadValue | Integer parameter out of range |
| 3 | BadWindow | Window ID not valid |
| 4 | BadPixmap | Pixmap ID not valid |
| 5 | BadAtom | Atom ID not valid |
| 6 | BadCursor | Cursor ID not valid |
| 7 | BadFont | Font ID not valid |
| 8 | BadMatch | Parameter mismatch |
| 9 | BadDrawable | Drawable (window or pixmap) not valid |
| 10 | BadAccess | Access denied |
| 11 | BadAlloc | Server out of resources |
| 12 | BadColor | Colormap entry not valid |
| 13 | BadGC | GC ID not valid |
| 14 | BadIDChoice | Resource ID already in use |
| 15 | BadName | Named color/font not found |

| Code | Name | Description |
|---|---|---|
| 16 | BadLength | Request length incorrect |
| 17 | BadImplementation | Server implementation error |

## A.7 Modifier Masks

```
const (
    ShiftMask   = 1 << 0
    LockMask    = 1 << 1  // Caps Lock
    ControlMask = 1 << 2
    Mod1Mask    = 1 << 3  // Alt
    Mod2Mask    = 1 << 4  // Num Lock
    Mod3Mask    = 1 << 5
    Mod4Mask    = 1 << 6  // Super/Windows
    Mod5Mask    = 1 << 7
    Button1Mask = 1 << 8
    Button2Mask = 1 << 9
    Button3Mask = 1 << 10
    Button4Mask = 1 << 11
    Button5Mask = 1 << 12
)
```

## A.8 Image Formats

| Code | Name | Description |
|---|---|---|
| 0 | XYBitmap | 1-bit per pixel |
| 1 | XYPixmap | Planar format |
| 2 | ZPixmap | Packed pixels (most common) |

## A.9 Standard Atoms

| Atom | Name | Purpose |
|---|---|---|
| 1 | PRIMARY | Selection |
| 2 | SECONDARY | Selection |
| 3 | ARC | Shape |
| 4 | ATOM | Type |
| … | … | … |
| 31 | WM_NAME | Window title (Latin-1) |
| 32 | WM_ICON_NAME | Icon title |
| 33 | WM_CLIENT_MACHINE | Host name |

Non-standard atoms (created via InternAtom): - `WM_PROTOCOLS` - Protocol negotiation -

`WM_DELETE_WINDOW` - Close button handling - `_NET_WM_NAME` - Window title (UTF-8) - `UTF8_STRING` - UTF-8 string type

## A.10 Connection Setup Response

```
Byte 0:     Success (1) / Failure (0)
Byte 1:     Unused (if success) / Reason length (if failure)
Bytes 2-3:  Protocol major version
Bytes 4-5:  Protocol minor version
Bytes 6-7:  Additional data length (in 4-byte units)

Additional data (success):
  Bytes 0-3:   Release number
  Bytes 4-7:   Resource ID base
  Bytes 8-11:  Resource ID mask
  Bytes 12-15: Motion buffer size
  Bytes 16-17: Vendor length
  Bytes 18-19: Maximum request length
  Byte 20:     Number of screens
  Byte 21:     Number of formats
  Byte 22:     Image byte order (0=LSB, 1=MSB)
  Byte 23:     Bitmap bit order
  Byte 24:     Bitmap scanline unit
  Byte 25:     Bitmap scanline pad
  Byte 26:     Min keycode
  Byte 27:     Max keycode
  Bytes 28-31: Unused
  Followed by: Vendor string, formats, screens
```

## A.11 Common Key Codes (X11/Linux)

```
const (
    KeyEscape     = 9
    Key1          = 10
    Key2          = 11
    // ... 3-9
    Key0          = 19
    KeyMinus      = 20
    KeyEqual      = 21
    KeyBackspace = 22
    KeyTab        = 23
    KeyQ          = 24
    KeyW          = 25
    KeyE          = 26
    KeyR          = 27
    KeyT          = 28
```

```
    KeyY          = 29
    KeyU          = 30
    KeyI          = 31
    KeyO          = 32
    KeyP          = 33
    KeyEnter      = 36
    KeyA          = 38
    KeyS          = 39
    KeyD          = 40
    KeyF          = 41
    KeyG          = 42
    KeyH          = 43
    KeyJ          = 44
    KeyK          = 45
    KeyL          = 46
    KeyZ          = 52
    KeyX          = 53
    KeyC          = 54
    KeyV          = 55
    KeyB          = 56
    KeyN          = 57
    KeyM          = 58
    KeySpace      = 65
    KeyF1         = 67
    KeyF2         = 68
    // ... F3-F10
    KeyF11        = 95
    KeyF12        = 96
    KeyUp         = 111
    KeyLeft       = 113
    KeyRight      = 114
    KeyDown       = 116
)
```

Note: Key codes can vary by keyboard layout and X server configuration. Use XKB for reliable key symbol translation.

# Appendix B: Binary Encoding

Reference for binary data manipulation in Go.

## B.1 Byte Order

### Little-Endian (Most Common)

Least significant byte first. Used by x86, AMD64, ARM (usually).

```
Value: 0x12345678

Memory: [78] [56] [34] [12]
         ^lowest address
```

### Big-Endian

Most significant byte first. Used by network protocols, some RISC.

```
Value: 0x12345678

Memory: [12] [34] [56] [78]
         ^lowest address
```

## B.2 Go's encoding/binary Package

### Writing Values

```go
import "encoding/binary"

// Write uint16 (2 bytes)
buf := make([]byte, 2)
binary.LittleEndian.PutUint16(buf, 0x1234)
// buf = [0x34, 0x12]

// Write uint32 (4 bytes)
buf := make([]byte, 4)
binary.LittleEndian.PutUint32(buf, 0x12345678)
// buf = [0x78, 0x56, 0x34, 0x12]
```

```
// Write uint64 (8 bytes)
buf := make([]byte, 8)
binary.LittleEndian.PutUint64(buf, 0x123456789ABCDEF0)
```

### Reading Values

```
// Read uint16
value := binary.LittleEndian.Uint16(buf[0:2])

// Read uint32
value := binary.LittleEndian.Uint32(buf[0:4])

// Read uint64
value := binary.LittleEndian.Uint64(buf[0:8])
```

## B.3 Signed Integers

Go's binary package works with unsigned types. Convert for signed:

```
// Write int16
buf := make([]byte, 2)
binary.LittleEndian.PutUint16(buf, uint16(int16Value))

// Read int16
value := int16(binary.LittleEndian.Uint16(buf))

// Write int32
binary.LittleEndian.PutUint32(buf, uint32(int32Value))

// Read int32
value := int32(binary.LittleEndian.Uint32(buf))
```

## B.4 Padding and Alignment

X11 requires 4-byte alignment for requests:

```
func pad4(length int) int {
    return (4 - (length % 4)) % 4
}

// Example: string of length 5 needs 3 bytes padding
name := "Hello"
padding := pad4(len(name))  // 3
totalLen := len(name) + padding  // 8
```

## B.5 Building X11 Requests

Pattern for constructing requests:

```
func buildRequest(opcode uint8, data1 uint32, data2 uint16) []byte {
    // Calculate length in 4-byte units
    reqLen := 3  // 12 bytes / 4

    req := make([]byte, reqLen*4)

    // Header
    req[0] = opcode                                    // Byte 0: opcode
    req[1] = 0                                         // Byte 1: often unused
    binary.LittleEndian.PutUint16(req[2:], uint16(reqLen))  // Bytes 2-3: length

    // Data
    binary.LittleEndian.PutUint32(req[4:], data1)     // Bytes 4-7
    binary.LittleEndian.PutUint16(req[8:], data2)     // Bytes 8-9
    // Bytes 10-11: padding (already zero)

    return req
}
```

## B.6 Common Patterns

### Fixed-Size Request

```
req := make([]byte, 16)  // 4 units * 4 bytes
req[0] = opcode
binary.LittleEndian.PutUint16(req[2:], 4)  // Length = 4 units
// Fill remaining 12 bytes with data
```

### Variable-Size Request

```
func buildVariableRequest(opcode uint8, fixedData []byte, varData []byte) []byte {
    padding := pad4(len(varData))
    totalLen := 8 + len(varData) + padding  // 8 = header + fixed

    reqLen := totalLen / 4
    req := make([]byte, totalLen)

    req[0] = opcode
    binary.LittleEndian.PutUint16(req[2:], uint16(reqLen))
    copy(req[4:8], fixedData)
    copy(req[8:], varData)

    return req
```

```
}
```

### Reading Variable-Length Responses

```
// Read 32-byte header
header := make([]byte, 32)
io.ReadFull(conn, header)

// Check for additional data
additionalLength := binary.LittleEndian.Uint32(header[4:8])
if additionalLength > 0 {
    additional := make([]byte, additionalLength*4)
    io.ReadFull(conn, additional)
}
```

## B.7 Bit Manipulation

### Setting Bits

```
// Set bit n
mask |= (1 << n)

// Example: Set event mask bits
eventMask := uint32(0)
eventMask |= (1 << 0)   // KeyPressMask
eventMask |= (1 << 2)   // ButtonPressMask
eventMask |= (1 << 15)  // ExposureMask
```

### Checking Bits

```
// Check if bit n is set
if value & (1 << n) != 0 {
    // Bit is set
}

// Example: Check modifier keys
if state & ShiftMask != 0 {
    // Shift is held
}
```

### Clearing Bits

```
// Clear bit n
mask &^= (1 << n)
```

```
// Example: Remove a flag
eventMask &^= PointerMotionMask
```

## B.8 Pixel Format Conversion

### RGB to BGRX (X11 format)

```
func rgbToBGRX(r, g, b uint8) []byte {
    return []byte{b, g, r, 0}
}
```

### RGBA to BGRA (with alpha)

```
func rgbaToBGRA(r, g, b, a uint8) []byte {
    return []byte{b, g, r, a}
}
```

### Packed 32-bit Color

```
// Pack RGB into uint32 (for X11 foreground/background)
func packRGB(r, g, b uint8) uint32 {
    return uint32(r)<<16 | uint32(g)<<8 | uint32(b)
}

// Unpack uint32 to RGB
func unpackRGB(packed uint32) (r, g, b uint8) {
    r = uint8((packed >> 16) & 0xFF)
    g = uint8((packed >> 8) & 0xFF)
    b = uint8(packed & 0xFF)
    return
}
```

## B.9 Unsafe Pointer Tricks

For shared memory and performance-critical code:

```
import (
    "reflect"
    "unsafe"
)

// Create Go slice from raw pointer
func ptrToSlice(ptr unsafe.Pointer, length int) []byte {
    var slice []byte
    header := (*reflect.SliceHeader)(unsafe.Pointer(&slice))
    header.Data = uintptr(ptr)
```

```
    header.Len = length
    header.Cap = length
    return slice
}

// Get pointer from slice
func sliceToPtr(slice []byte) unsafe.Pointer {
    return unsafe.Pointer(&slice[0])
}
```

**Warning**: Unsafe code bypasses Go's safety guarantees. Use only when necessary.

## B.10 Debugging Binary Data

### Hex Dump

```
import "encoding/hex"

func hexDump(data []byte) string {
    return hex.Dump(data)
}

// Output:
// 00000000  55 00 04 00 00 00 01 00  00 00 00 00 ff ff ff 00  |U...............|
```

### Binary Representation

```
func binaryString(value uint32) string {
    return fmt.Sprintf("%032b", value)
}

// Output: 00000000000000001111111111111111
```

### Field-by-Field Dump

```
func dumpRequest(req []byte) {
    fmt.Printf("Opcode: %d\n", req[0])
    fmt.Printf("Unused: %d\n", req[1])
    fmt.Printf("Length: %d units (%d bytes)\n",
        binary.LittleEndian.Uint16(req[2:4]),
        binary.LittleEndian.Uint16(req[2:4])*4)

    for i := 4; i < len(req); i += 4 {
        fmt.Printf("Offset %d: 0x%08X\n", i,
            binary.LittleEndian.Uint32(req[i:]))
    }
```

```
}
```

# Appendix C: Complete Code Listing

The essential files that make up the Glow library.

## C.1 Project Structure

```
glow/
   go.mod
   glow.go
   window.go
   canvas.go
   color.go
   events.go
   keys.go
   internal/
       x11/
           conn.go
           auth.go
           window.go
           events.go
           draw.go
           framebuffer.go
```

## C.2 go.mod

```
module github.com/AchrafSoltani/glow

go 1.21
```

## C.3 glow.go

```
// Package glow provides a simple 2D graphics library for Go.
// It creates windows, handles input, and renders pixels using software rendering.
//
// Example:
//
```

```
//  win, _ := glow.NewWindow("Hello", 800, 600)
//  defer win.Close()
//
//  for win.IsOpen() {
//      for e := win.PollEvent(); e != nil; e = win.PollEvent() {
//          // handle events
//      }
//      win.Canvas().Clear(glow.Black)
//      win.Display()
//  }
package glow
```

## C.4 color.go

```
package glow

// Color represents an RGBA color.
type Color struct {
    R, G, B, A uint8
}

// Predefined colors
var (
    Black       = Color{0, 0, 0, 255}
    White       = Color{255, 255, 255, 255}
    Red         = Color{255, 0, 0, 255}
    Green       = Color{0, 255, 0, 255}
    Blue        = Color{0, 0, 255, 255}
    Yellow      = Color{255, 255, 0, 255}
    Cyan        = Color{0, 255, 255, 255}
    Magenta     = Color{255, 0, 255, 255}
    Transparent = Color{0, 0, 0, 0}
)

// RGB creates an opaque color from RGB values.
func RGB(r, g, b uint8) Color {
    return Color{r, g, b, 255}
}

// RGBA creates a color with alpha.
func RGBA(r, g, b, a uint8) Color {
    return Color{r, g, b, a}
}

// Hex creates an opaque color from a hex value (0xRRGGBB).
func Hex(hex uint32) Color {
```

```
    return Color{
        R: uint8((hex >> 16) & 0xFF),
        G: uint8((hex >> 8) & 0xFF),
        B: uint8(hex & 0xFF),
        A: 255,
    }
}
```

## C.5 internal/x11/conn.go (excerpts)

```
package x11

import (
    "encoding/binary"
    "fmt"
    "io"
    "net"
    "os"
)

type Connection struct {
    conn          net.Conn
    RootWindow    uint32
    RootDepth     uint8
    RootVisual    uint32
    resourceIDBase uint32
    resourceIDMask uint32
    nextResourceID uint32

    // Cached atoms
    atomWmProtocols    Atom
    atomWmDeleteWindow Atom
    atomNetWmName      Atom
    atomUtf8String     Atom
}

func Connect() (*Connection, error) {
    display := os.Getenv("DISPLAY")
    if display == "" {
        display = ":0"
    }

    socketPath := fmt.Sprintf("/tmp/.X11-unix/X%s", display[1:])
    conn, err := net.Dial("unix", socketPath)
    if err != nil {
        return nil, err
```

```
    }

    c := &Connection{conn: conn}

    if err := c.authenticate(); err != nil {
        conn.Close()
        return nil, err
    }

    if err := c.handshake(); err != nil {
        conn.Close()
        return nil, err
    }

    if err := c.initAtoms(); err != nil {
        conn.Close()
        return nil, err
    }

    return c, nil
}

func (c *Connection) GenerateID() uint32 {
    id := c.resourceIDBase | c.nextResourceID
    c.nextResourceID++
    return id
}

func (c *Connection) Close() error {
    return c.conn.Close()
}
```

## C.6 internal/x11/framebuffer.go

```
package x11

type Framebuffer struct {
    Width  int
    Height int
    Pixels []byte
}

func NewFramebuffer(width, height int) *Framebuffer {
    return &Framebuffer{
        Width:  width,
        Height: height,
```

```
        Pixels: make([]byte, width*height*4),
    }
}

func (fb *Framebuffer) SetPixel(x, y int, r, g, b uint8) {
    if x < 0 || x >= fb.Width || y < 0 || y >= fb.Height {
        return
    }
    offset := (y*fb.Width + x) * 4
    fb.Pixels[offset] = b
    fb.Pixels[offset+1] = g
    fb.Pixels[offset+2] = r
    fb.Pixels[offset+3] = 0
}

func (fb *Framebuffer) GetPixel(x, y int) (r, g, b uint8) {
    if x < 0 || x >= fb.Width || y < 0 || y >= fb.Height {
        return 0, 0, 0
    }
    offset := (y*fb.Width + x) * 4
    return fb.Pixels[offset+2], fb.Pixels[offset+1], fb.Pixels[offset]
}

func (fb *Framebuffer) Clear(r, g, b uint8) {
    fb.Pixels[0] = b
    fb.Pixels[1] = g
    fb.Pixels[2] = r
    fb.Pixels[3] = 0

    for filled := 4; filled < len(fb.Pixels); filled *= 2 {
        copy(fb.Pixels[filled:], fb.Pixels[:filled])
    }
}

func (fb *Framebuffer) DrawRect(x, y, width, height int, r, g, b uint8) {
    x0, y0 := max(x, 0), max(y, 0)
    x1, y1 := min(x+width, fb.Width), min(y+height, fb.Height)

    if x0 >= x1 || y0 >= y1 {
        return
    }

    for py := y0; py < y1; py++ {
        offset := (py*fb.Width + x0) * 4
        for px := x0; px < x1; px++ {
            fb.Pixels[offset] = b
            fb.Pixels[offset+1] = g
```

```
            fb.Pixels[offset+2] = r
            fb.Pixels[offset+3] = 0
            offset += 4
        }
    }
}

func (fb *Framebuffer) DrawLine(x0, y0, x1, y1 int, r, g, b uint8) {
    dx := abs(x1 - x0)
    dy := -abs(y1 - y0)
    sx, sy := 1, 1
    if x0 > x1 {
        sx = -1
    }
    if y0 > y1 {
        sy = -1
    }
    err := dx + dy

    for {
        fb.SetPixel(x0, y0, r, g, b)
        if x0 == x1 && y0 == y1 {
            break
        }
        e2 := 2 * err
        if e2 >= dy {
            err += dy
            x0 += sx
        }
        if e2 <= dx {
            err += dx
            y0 += sy
        }
    }
}

func (fb *Framebuffer) FillCircle(cx, cy, radius int, r, g, b uint8) {
    for y := -radius; y <= radius; y++ {
        for x := -radius; x <= radius; x++ {
            if x*x+y*y <= radius*radius {
                fb.SetPixel(cx+x, cy+y, r, g, b)
            }
        }
    }
}

func abs(x int) int {
```

```
    if x < 0 {
        return -x
    }
    return x
}
```

## C.7 internal/x11/draw.go

```
package x11

import "encoding/binary"

const (
    OpPutImage        = 72
    ImageFormatZPixmap = 2
)

func (c *Connection) PutImage(drawable, gc uint32, width, height uint16,
    dstX, dstY int16, depth uint8, data []byte) error {

    bytesPerPixel := 4
    rowBytes := int(width) * bytesPerPixel
    maxDataBytes := 262140 - 24
    rowsPerRequest := maxDataBytes / rowBytes

    if rowsPerRequest > int(height) {
        rowsPerRequest = int(height)
    }
    if rowsPerRequest < 1 {
        rowsPerRequest = 1
    }

    for y := 0; y < int(height); y += rowsPerRequest {
        stripHeight := rowsPerRequest
        if y+stripHeight > int(height) {
            stripHeight = int(height) - y
        }

        stripData := data[y*rowBytes : (y+stripHeight)*rowBytes]
        err := c.putImageStrip(drawable, gc, width, uint16(stripHeight),
            dstX, dstY+int16(y), depth, stripData)
        if err != nil {
            return err
        }
    }
```

```
    return nil
}

func (c *Connection) putImageStrip(drawable, gc uint32, width, height uint16,
    dstX, dstY int16, depth uint8, data []byte) error {

    dataLen := len(data)
    padding := (4 - (dataLen % 4)) % 4
    reqLen := 6 + (dataLen+padding)/4

    req := make([]byte, reqLen*4)

    req[0] = OpPutImage
    req[1] = ImageFormatZPixmap
    binary.LittleEndian.PutUint16(req[2:], uint16(reqLen))
    binary.LittleEndian.PutUint32(req[4:], drawable)
    binary.LittleEndian.PutUint32(req[8:], gc)
    binary.LittleEndian.PutUint16(req[12:], width)
    binary.LittleEndian.PutUint16(req[14:], height)
    binary.LittleEndian.PutUint16(req[16:], uint16(dstX))
    binary.LittleEndian.PutUint16(req[18:], uint16(dstY))
    req[20] = 0
    req[21] = depth

    copy(req[24:], data)

    _, err := c.conn.Write(req)
    return err
}
```

## C.8 Example: Minimal Window

```
package main

import (
    "github.com/AchrafSoltani/glow"
)

func main() {
    win, err := glow.NewWindow("Minimal Example", 800, 600)
    if err != nil {
        panic(err)
    }
    defer win.Close()

    canvas := win.Canvas()
```

```
    for win.IsOpen() {
        for e := win.PollEvent(); e != nil; e = win.PollEvent() {
            switch e.(type) {
            case glow.CloseEvent:
                win.Close()
            case glow.KeyEvent:
                if e.(glow.KeyEvent).Key == glow.KeyEscape {
                    win.Close()
                }
            }
        }

        canvas.Clear(glow.RGB(30, 30, 50))
        canvas.FillCircle(400, 300, 50, glow.Red)

        win.Display()
    }
}
```

## C.9 Example: Animation

```
package main

import (
    "math"
    "time"

    "github.com/AchrafSoltani/glow"
)

func main() {
    win, _ := glow.NewWindow("Animation", 800, 600)
    defer win.Close()

    canvas := win.Canvas()
    startTime := time.Now()

    for win.IsOpen() {
        for e := win.PollEvent(); e != nil; e = win.PollEvent() {
            if _, ok := e.(glow.CloseEvent); ok {
                win.Close()
            }
        }

        t := time.Since(startTime).Seconds()
```

```
        canvas.Clear(glow.Black)

        // Animated circles
        for i := 0; i < 5; i++ {
            angle := t + float64(i)*math.Pi*2/5
            x := 400 + int(150*math.Cos(angle))
            y := 300 + int(150*math.Sin(angle))

            hue := float64(i) / 5
            color := hsvToRGB(hue, 1, 1)
            canvas.FillCircle(x, y, 30, color)
        }

        win.Display()
        time.Sleep(16 * time.Millisecond)
    }
}

func hsvToRGB(h, s, v float64) glow.Color {
    i := int(h * 6)
    f := h*6 - float64(i)
    p := v * (1 - s)
    q := v * (1 - f*s)
    t := v * (1 - (1-f)*s)

    var r, g, b float64
    switch i % 6 {
    case 0:
        r, g, b = v, t, p
    case 1:
        r, g, b = q, v, p
    case 2:
        r, g, b = p, v, t
    case 3:
        r, g, b = p, q, v
    case 4:
        r, g, b = t, p, v
    case 5:
        r, g, b = v, p, q
    }

    return glow.RGB(uint8(r*255), uint8(g*255), uint8(b*255))
}
```

---

The complete source code is available at: **https://github.com/AchrafSoltani/glow**