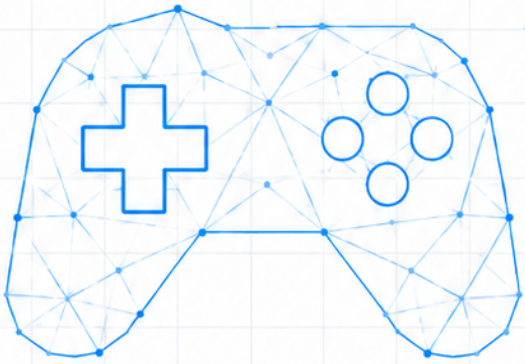
Beginning Game Programming with Go



Luigi Vanacore

Beginning Game Programming with Go

A hands-on, project-based guide to 2D game development in Go

Luigi Vanacore

This book is available at https://leanpub.com/gameprogramminggolang

This version was published on 2026-07-15

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

To my Girlfriend, Friends and Family

Contents

Introduction

Before you dive into code, here is what you'll find in these pages—and why Go and Ebitengine make a great pair for building games.

What This Book Is About

If you've ever wanted to build a 2D game from scratch without the weight of a full engine, this book is for you. It uses **Go** and **Ebitengine** (formerly Ebiten), a lightweight 2D game library, and you build everything yourself: the game loop, the scene graph, input handling, collisions, particles. No drag-and-drop editors, no black boxes—just code you can understand and extend.

By the end, you'll have built **Gopher Survivor**, a Vampire Survivor–style game. Your character moves around a map, fights waves of enemies, collects experience orbs, levels up, and picks upgrades to get stronger. You'll see how a game loop works, how to structure a game with nodes and transforms, how to handle collisions and state machines, and how to add polish with particles, audio, and UI. The journey from a blank window to a playable game is the whole point.

Here is the finished game you will build across this book:

Figure 1. Gopher Survivor, the finished game — a hit landing with blood particles, a floating damage number, and enemies closing in

One thing to be clear about: this book assumes you already know how to program, and how to program in Go. It won't teach syntax, structs, or goroutines. It focuses on *game development*—the patterns, the architecture, the tricks that make a game tick. If Go is new to you, spend some time with it first; you'll get much more out of this book once the language feels familiar.

What Is Go (Golang)?

Go was born at Google in the late 2000s. Ken Thompson and Rob Pike, among others, wanted a language that was simple, fast to compile, and suitable for large systems. Today it powers countless backend services, DevOps tools, and CLI applications. The name "Golang" stuck from the early days when the community site was golang.org; the official home is now go.dev.

What makes Go appealing for our purposes? It compiles to native code in seconds, produces a single executable you can ship anywhere, and its syntax is straightforward—no templates or inheritance maze. You get garbage

collection, concurrency built in (goroutines, channels), and cross-compilation out of the box. For 2D games and indie projects, that's a solid foundation.

What Is Ebitengine?

Figure 2. Ebitengine

Ebitengine (formerly Ebiten) is an open-source 2D game library for Go. It gives you a window, a game loop, rendering, input, and audio—the essentials—without pretending to be a full engine. There are no editors, no physics engine, no scripting layer. You write your game logic in Go and lean on Ebitengine for the low-level stuff. That simplicity is intentional: you stay in control.

It was created by Hajime Hoshi and has been used in commercial games like *Fishing Paradiso*, with millions of downloads. It runs on desktop (Windows, macOS, Linux), the web via WebAssembly, and mobile (iOS, Android). We'll use it as our foundation and build a small framework on top as we go.

Why Go for Games?

Game development is often associated with C++, C#, or JavaScript. Go is less common in this space, but it has a lot going for it. The compile times are short,

so you iterate quickly. You ship one binary—no runtime to bundle, no DLL jungle. Cross-compilation is trivial: build for Windows, Linux, and macOS from a single machine. And when you need concurrency (loading assets, network calls, background AI), goroutines keep things manageable without the usual threading headaches.

Is Go the right tool for a 3D AAA blockbuster? Probably not—that world belongs to C++ and specialized engines. For 2D games, indie projects, prototypes, and learning, Go and Ebitengine are a great fit. The ecosystem is growing, the API is clean, and you're never fighting the framework.

Conventions Used in This Book

To help you navigate the content, we use the following conventions throughout:

Convention	Meaning
`Code in backticks`	Code identifiers, filenames, paths, and commands (e.g. `game.go`, `go run ./cmd`)
Bold	New terms when first introduced, important concepts, and UI elements
Italics	Emphasis and technical terms (e.g. *game loop*)
Code blocks	Complete, runnable examples. We use ```` ```go ```` for Go and ```` ```bash ```` for shell commands.
> **Note**	General information worth highlighting
> **Tip**	Practical advice to use the subject matter more effectively
> **Warning**	Potential pitfalls or risks to be aware of
> **Important**	Critical information you need to succeed

We avoid referring to colors in images or diagrams, since many copies are read in grayscale or on e-readers. When we describe UI or visuals, we use position, labels, or shape instead.

The Game We Are Building: Gopher Survivor

We build a **clone of Vampire Survivors**. It helps to understand the original game and what our clone includes.

Vampire Survivors: The Game We Are Mimicking

Vampire Survivors (2022, poncle) is an independent roguelike that blends bullet-hell with roguelike progression. It pioneered the "bullet heaven" style: instead of dodging bullets, the player becomes a mobile tower of destruction, filling the screen with projectiles.

What You Do in the Game

A typical session works like this:

1. **Start a run** – Choose a character (each has a unique starting weapon) and a stage.
2. **Move** – Use the analogue stick or keys to move your character on a top-down map. The camera follows you. The map is large; you explore while avoiding danger.
3. **Auto-attack** – You **never press an attack button**. All weapons fire automatically on cooldowns. Whips, garlic auras, holy water, spinning bibles, axes—they all trigger on their own. Your only job is to **move and position** yourself so enemies die before they reach you.
4. **Collect XP orbs** – When enemies die, they drop experience gems (orbs). You walk over them to collect. Each orb fills your XP bar.
5. **Level up** – When the XP bar is full, the game **pauses** and shows 3–4 random options: a new weapon, a passive item (e.g. +damage, +area), or an upgrade to an existing weapon. You pick one. If you already have that weapon, it levels up (stronger, faster, etc.). You can hold up to **six weapons** plus passives.
6. **Weapon evolution** – When a base weapon is max level and you have its matching passive, picking up a Treasure Chest **evolves** it into a much stronger form. Evolution is a key power spike.
7. **Survive** – Enemies spawn in waves, move toward you, and deal damage on contact. Your health bar depletes; if it hits zero, you die and the run ends. The goal is to **last 30 minutes** (or clear the stage). If you survive, you win.

What the Game Is Based On

- **Bullet heaven** – You generate the bullets; enemies walk into them. The screen fills with your projectiles.
- **Synergy** – Weapons and passives combo: garlic + area boosts, whip + speed, multiple projectiles stacking.
- **Roguelike loop** – Each run is self-contained. Death resets progress for that run.
- **Power curve** – You start weak and scale to absurd strength. By minute 25, dozens of enemies die per second around you.
- **Horde management** – The number and toughness of enemies escalate. You must keep killing to level and stay ahead of the curve.

What We Implement in This Book vs. the Original

Our clone **Gopher Survivor** captures the spirit of Vampire Survivors but with a smaller scope, suitable for learning. Here is what we implement compared to the original:

Aspect	Vampire Survivors (original)	Gopher Survivor (this book)
Movement	8-directional, one character	Same: top-down, arrow keys (Ch4)
Weapons	6 weapon slots, dozens of weapons, evolution	Four weapons: Knife (Ch7), Sacred Book and Holy Shield (Ch9), Flying Axe (Ch10). No evolution.
Attacking	Fully automatic, no input	Same: weapons fire on timers
Aiming	Most weapons auto-aim (nearest enemy or fixed direction)	Knife aims toward the **mouse cursor** (Ch7); other weapons have fixed behaviour

Aspect	Vampire Survivors (original)	Gopher Survivor (this book)
Enemies	Many types, different behaviours, bosses	One chasing enemy at first (Ch6); later five kinds with hit points and time-scaled stats, plus a Cyclops mini-boss (Ch11). All move toward the player.
Enemy spawner	Complex waves, varied spawn points	Spawner at screen edges, waves of 2–3 (Ch7); difficulty tiers escalate over time (Ch11)
XP / Level up	Orbs, pause, 3–4 choices, exponential XP curve	XP orbs (Ch8); the world freezes with **3 choices** at level up (Ch10); exponential XP
Upgrades	Weapons, passives, evolution	Weapon upgrades (speed, cooldown) (Ch10) plus global passives and unlock gating (HP, move speed, XP, damage, cooldown) (Ch11)
Health	Health bar, damage on contact	Same: health bar and contact damage (Ch8)
Drops	XP orbs, gold, chests, power-ups	XP orbs (Ch8), **health potion** (5% drop) (Ch9)
Win condition	Survive 30 minutes (or stage end)	**No timed win** — endless survival; the run ends when your HP reaches zero, and your time survived is the score (Ch10–Ch11)

Aspect	Vampire Survivors (original)	Gopher Survivor (this book)
UI	Health, XP, minimap, pickups	Health bar, XP bar, level-up popup (Ch8), upgrade panels (Ch10)
State management	Menus, pause, run	State machine for game flow – main menu, pause, options (Ch12)

We focus on the **core loop**: move, auto-attack, collect XP, level up, choose upgrades, survive waves. We omit evolution, multiple characters, and most of the content depth—so the code stays readable and each chapter adds one clear mechanic.

What Comes Next

In **Chapter 1**, we set up an Ebitengine project, create a window, and draw an image on screen. That's where the game loop and the core concepts come to life. From there we build the framework and the full Gopher Survivor game, step by step. Let's go.

* * *

All the code for this book lives in the companion GitHub repository at https://github.com/LuigiVanacore/beginning-game-programming-go-ebitengine, organized one directory per chapter (`ch01`, `ch02`, and so on). Each chapter ends with a direct link to its folder.

Chapter 1: Ebiten Basics

This chapter introduces Ebiten, explains how to use it in Go, and walks through a complete example that loads and displays a texture (the Go Gopher icon) on screen.

If you have never used Ebitengine: do not worry. We assume no prior knowledge. Every function is explained, every step is detailed. By the end of this chapter you will understand how a minimal Ebitengine program works and how to display an image on screen. The concepts here—the game loop, loading assets, and drawing—are the foundation for everything that follows in this book.

Before diving into Ebitengine, you need Go installed. The following sections cover installation and a quick sanity check. If you already have Go 1.22 or later, you can skip to What Is Ebiten?.

In this chapter, we will cover the following topics:

- Installing Go and Ebitengine and verifying the toolchain
- Understanding Ebitengine's game loop and its `Update`, `Draw`, and `Layout` methods
- Creating a Go module and structuring a minimal project
- Embedding an image asset with `go:embed` and decoding it at startup
- Drawing a sprite and positioning it with the `GeoM` transform
- The screen coordinate system Ebitengine uses

Technical requirements

- **Go:** version 1.22 or later, from go.dev/dl. This chapter walks you through installing it if you do not have it yet.
- **Ebitengine:** v2 (the latest v2.x release). This chapter walks you through adding it to your module.
- **Assets:** the Go Gopher icon (`golang_icon.png`), used as the image you draw on screen.
- **Code:** the complete, runnable project for this chapter lives in `ch01/`.

Getting Go on Your Machine

If Go isn't installed yet, here's how to get it. The official downloads are at go.dev/dl. You need **Go 1.22 or later** for Ebitengine—the examples in this book assume that minimum. We'll go through each platform; pick the one that matches your setup.

Windows

On Windows, the easiest route is the MSI installer. Download the file named something like `go1.22.x.windows-amd64.msi` (or `arm64` if you're on an ARM machine), double-click it, and follow the wizard. The installer puts Go in `C:\Program Files\Go` and adds it to your system `PATH` automatically.

Important: After installation, close any existing Command Prompt or PowerShell windows and open a new one. The `PATH` is read when the terminal starts, so old windows won't see the new `go` command.

In the new terminal, run:

```
# Run from: (any directory) — verify Go installation
go version
```

You should see something like `go version go1.22.x windows/amd64`. If you get "go: command not found" or similar, the `PATH` wasn't updated—you may need to log out and back in, or add `C:\Program Files\Go\bin` to your user PATH manually (Settings ▫ System ▫ About ▫ Advanced system settings ▫ Environment variables).

If you use WSL (Windows Subsystem for Linux): Go can run inside WSL as well. Install it using the Linux instructions below. When we run Ebitengine later, you'll typically want to target Windows (`GOOS=windows`) so the game opens a native window rather than trying to use Linux graphics inside WSL.

macOS

Two options. The official route: download the `.pkg` from go.dev/dl (choose amd64 for Intel Macs, arm64 for Apple Silicon), run it, and you're done. The installer places Go in `/usr/local/go` and sets up your PATH.

If you use Homebrew, you can instead run:

```
# Run from: (any directory) — macOS Homebrew install
brew install go
```

Homebrew installs to a different location, but the result is the same. Open a new Terminal window (or a new tab) and run:

```
# Run from: (any directory) — verify Go installation
go version
```

You should see something like `go version go1.22.x darwin/arm64` (Apple Silicon) or `darwin/amd64` (Intel). If `go` isn't found, check that Homebrew's bin directory is in your PATH—Homebrew usually prints the exact path when you run `brew install go`.

Linux

Linux offers the most flexibility. Here are a few approaches.

Official tarball (works on any distro): Download the `.tar.gz` from go.dev/dl. If you have an old Go installation, remove it first (`sudo rm -rf /usr/local/go`). Then extract:

```
# Run from: (directory containing the downloaded tarball)
sudo tar -C /usr/local -xzf go1.22.x.linux-amd64.tar.gz
```

Replace the filename with the one you downloaded. Next, add Go's `bin` directory to your PATH. Open `~/.bashrc` (or `~/.profile`, or `~/.zshrc` if you use Zsh) and add:

```
# Add to ~/.bashrc or ~/.zshrc
export PATH=$PATH:/usr/local/go/bin
```

Save the file, then run `source ~/.bashrc` (or the appropriate file) or open a new terminal. Run `go version` to confirm.

Package manager (simpler, but version may vary): Many distros ship Go. On Ubuntu or Debian:

```
# Run from: (any directory) — Ubuntu/Debian
sudo snap install go --classic
```

On Fedora:

```
# Run from: (any directory) — Fedora
sudo dnf install golang
```

On Arch:

```
# Run from: (any directory) — Arch
sudo pacman -S go
```

Check the installed version with `go version`. If it's older than 1.22, use the tarball method instead to get a newer release.

Verifying the installation

Whatever route you took, run:

```
# Run from: (any directory) — verify installation
go version
```

If you see `go version go1.22.x` (or higher) followed by your OS and architecture, you're good. Ebitengine will work with that. If the command fails or the version is too old, revisit the steps for your platform—most issues come from a missing or incorrect PATH.

Troubleshooting

"go: command not found" — The `go` binary isn't in your PATH. On Windows, run the installer again and ensure "Add to PATH" is checked, then restart your terminal (or log out and back in). On Linux and macOS, double-check that you've added `/usr/local/go/bin` (or the correct path) to your shell config file and that you've sourced it or opened a new terminal.

Wrong or old version — If you have multiple Go installs (e.g., an old tarball plus a package manager install), they can conflict. Remove the old one and

keep a single installation. On Linux, `which go` shows which binary is used; `go env GOROOT` shows the installation directory.

Permission denied on Linux – If you get permission errors when running `go`, make sure the Go binary and your project folder are readable. Avoid installing to `/usr/local/go` without `sudo` if you don't have write access there; alternatively, extract Go to `$HOME/go` and add `$HOME/go/bin` to your PATH.

Proxy or corporate network – If `go get` or `go mod download` fails behind a corporate firewall, you may need to set `GOPROXY`, `GOPRIVATE`, or `GONOSUMDB`. For most home setups, the defaults work fine.

A Quick Sanity Check: Hello, World

If you've just installed Go or want to double-check your setup, run through a quick "Hello, World!"

Create a folder, add a `main.go` file:

```
1  // File: main.go (standalone sanity check – create in new project folder)
2  package main
3
4  import "fmt"
5
6  func main() {
7      fmt.Println("Hello, World!")
8  }
```

From that folder, run `go run main.go`. You should see "Hello, World!" in the terminal. If so, you're ready to move on. (For a real project you'd run `go mod init gopher-survivor` first; we'll do that when we create the actual game in this chapter.)

What Is Ebiten?

Ebiten (now officially called **Ebitengine**) is an open-source 2D game library for the Go programming language. In a nutshell: it gives you a window, a game loop that calls your code every frame, and the primitives to draw images, handle input, and play audio. You write game logic in Go; Ebitengine handles the low-level graphics, platform integration, and the rhythm of Update–Draw. It is not a full engine with editors or physics–it is a *library* you compose into your own architecture.

History and Creator

Ebiten was created by **Hajime Hoshi**, a software engineer and CTO at Odencat in Tokyo, Japan. He started the project to bring a simple, productive 2D game development experience to Go. The library has grown into a mature, production-ready engine with thousands of stars on GitHub and has been used in commercial games. Hoshi has received the Google Open Source Peer Bonus award for his contributions. The project is licensed under the Apache 2.0 license.

What Can You Build With It?

Ebiten is suited for a wide range of applications:

- **2D games**: Platformers, top-down shooters, puzzle games, RPGs, arcade classics.
- **Tools and visual applications**: Level editors, animation preview tools, data visualizations.
- **Web games**: Compile to WebAssembly and run in the browser with the same codebase.
- **Mobile games**: Deploy to iOS and Android with native performance.
- **Desktop games**: Windows, macOS, Linux, and even Nintendo Switch (with additional tooling).

Notable examples include *Fishing Paradiso*, a commercial game built with Ebiten that has been downloaded over 2 million times. The engine is designed to scale from simple prototypes to full commercial releases.

How Is It Structured?

Ebiten is organized into a set of packages, each focused on a specific concern:

Package	Purpose
`github.com/hajimehoshi/ebiten/v2`	Core package: game loop, `ebiten.Image`, basic input and drawing.
`ebiten/v2/ebitenutil`	Utilities: debug printing, loading images from file, drawing simple shapes.
`ebiten/v2/inpututil`	Input helpers: "just pressed" detection, touch events, gamepad.
`ebiten/v2/audio`	Audio playback; supports WAV, MP3, OGG.
`ebiten/v2/text`	Text rendering with fonts.
`ebiten/v2/vector`	Vector graphics: lines, rectangles, circles (GPU-accelerated).

Architectural principles:

- **Universal image system**: Everything drawable is an `ebiten.Image`—the screen, offscreen buffers, sprites, tiles. They all behave the same: you draw one image onto another. This uniformity simplifies rendering and makes it easy to add effects (render to texture, then draw the texture).
- **Single-threaded game loop**: `Layout`, `Update`, and `Draw` run one after the other on a single goroutine. No locks are needed inside these methods; your game logic stays simple and predictable.
- **Minimal abstraction**: Ebiten gives you building blocks (images, transforms, input) rather than a full engine. You compose them into your own architecture—which is exactly what this book does as we build our framework.

Features and Capabilities

Ebitengine provides a focused set of features that cover the essentials of 2D game development:

Feature	Description
Game loop	`Update`, `Draw`, and `Layout` — you implement these three methods; Ebitengine handles timing, vsync, and window management.
Rendering	Draw images (sprites, tiles, textures) with position, scale, rotation, and color modulation. Everything is an `ebiten.Image`; you draw one image onto another. GPU-accelerated via OpenGL, Metal, DirectX, or WebGL.
Input	Keyboard, mouse, touch, and gamepad. The core package offers basic input; `inpututil` adds "just pressed" detection and touch events.
Audio	Play WAV, MP3, and OGG files. The `ebiten/v2/audio` package handles playback and streaming.
Text	Render text with custom fonts via `ebiten/v2/text`.
Vector graphics	`ebiten/v2/vector` provides GPU-accelerated shapes: lines, rectangles, circles, paths. Useful for UI, debug overlays, and simple graphics.
Offscreen rendering	Create render targets and draw to them, then draw the result to the screen. Enables post-processing, minimaps, and layered rendering.

Ebitengine does *not* provide a built-in physics engine, a scene graph, or an entity-component system. You build those yourself—or use third-party libraries—which gives you full control. That minimalism is intentional: the library stays small and you stay in charge.

Key Characteristics

- **Pure Go**: Written in Go with minimal C dependencies (mainly for window creation and graphics backends).

- **Simple API**: The core interface has just three methods: `Update`, `Draw`, and `Layout`.
- **Cross-platform**: Windows, macOS, Linux, FreeBSD, Web (WebAssembly), iOS, Android.
- **Performance**: Uses hardware acceleration (OpenGL, Metal, DirectX, WebGL) and batches draw calls automatically.

Ebiten is ideal for 2D games, tools, and visual applications. Throughout this book, we will build a complete game framework on top of Ebiten, learning both the library and the concepts that power real game engines.

Installing Ebitengine

Ebitengine is a Go library, so installation is straightforward: add it to your project with `go get`. You must, however, satisfy a few prerequisites.

1. Go

Install Go 1.22 or later. Ebitengine requires this minimum version.

2. C compiler (except on Windows)

Ebitengine uses both Go and C. A C compiler is required on macOS, Linux, and FreeBSD. **On Windows, no C compiler is needed**—Ebitengine uses pure-Go rendering there.

- **macOS**: Run `clang` in the terminal; a dialog will prompt installation if needed. If you see an `xcrun` error, run `xcode-select --install`.
- **Linux**: Install GCC, e.g. `apt install gcc` on Ubuntu.
- **FreeBSD**: Run `pkg install clang`.

3. Platform-specific dependencies

On Linux and FreeBSD, you need development libraries for graphics, input, and audio. Examples:

- **Debian / Ubuntu**:
  ```
  sudo apt install libc6-dev libgl1-mesa-dev libxcursor-dev
  libxi-dev libxinerama-dev libxrandr-dev libxxf86vm-dev
  libasound2-dev pkg-config
  ```

- **Fedora**:
  `sudo dnf install libglvnd-devel libXrandr-devel libxcursor-devel libxinerama-devel libXi-devel libXxf86vm-devel alsa-lib-devel pkg-config`
- **Arch Linux**:
  `sudo pacman -S mesa libxrandr libxcursor libxinerama libxi pkg-config`

macOS and Windows typically need no extra packages.

4. Verify your setup

Run the official rotating Gophers example:

```
# Run from: (any directory) — verify Ebitengine
go run github.com/hajimehoshi/ebiten/v2/examples/rotate@latest
```

If a window appears with a rotating Gopher image, your environment is correctly configured.

On **WSL (Windows Subsystem for Linux)**, you must target Windows when running:

```
# Run from: (any directory) — WSL
GOOS=windows go run github.com/hajimehoshi/ebiten/v2/examples/rotate@latest
```

Supported Platforms and Architectures

Ebitengine runs on many platforms and CPU architectures:

Platform	Architectures	Notes
Windows	amd64, arm64, 386	No C compiler needed on amd64/arm64
macOS	amd64, arm64	Intel and Apple Silicon
Linux	amd64, arm64, 386, arm, loong64, ppc64le, riscv64, s390x	Requires dev libraries above

Platform	Architectures	Notes
FreeBSD	amd64, arm64	Less tested by the author
Web	wasm (WebAssembly)	Run in the browser
Android	amd64, arm64	Mobile app; may need 386/arm for older devices
iOS	amd64 (simulator), arm64	Mobile app

Cross-compilation: You can cross-compile to **Windows** and **WebAssembly** easily. For other targets (e.g. Linux from Windows), cross-compilation is difficult because of Cgo and platform-specific build requirements—building on the target platform or in a CI environment is usually simpler.

WebAssembly: To build for the web:

```
# Run from: ch01 (or project root)
GOOS=js GOARCH=wasm go build -o game.wasm .
```

You then serve `game.wasm` and a small JavaScript loader (Ebitengine provides one) from a web server.

The Ebiten Game Interface

To create a game with Ebiten, you implement the `ebiten.Game` interface. An **interface** in Go is a contract: it specifies which methods your type must have. Ebitengine calls these methods automatically. You do not call them yourself—you just implement them.

```
// Interface: ebiten.Game (from Ebitengine library)
type Game interface {
    Update() error
    Draw(screen *ebiten.Image)
    Layout(outsideWidth, outsideHeight int) (int, int)
}
```

Understanding the Game Loop

A game runs in a **loop**: over and over, the engine updates the game state and draws the screen. This happens dozens of times per second so the player sees smooth animation. Ebitengine controls this loop for you; your job is to provide the logic in three methods.

The Three Methods Explained

`Update() error` – Game logic

- **When it is called**: Every **tick**. By default, Ebitengine runs at 60 ticks per second (60 TPS), so `Update` is called 60 times per second.
- **What it does**: You put all your game logic here: moving characters, checking keyboard input, updating scores, running physics, and so on. Think of it as "what happens each moment in the game."
- **Return value**: Return `nil` to continue; return an error to stop the game (for example, when the player quits).
- **Important**: Do not draw anything here. Drawing happens only in `Draw`.

`Draw(screen *ebiten.Image)` – Rendering

- **When it is called**: Every **frame**. The number of frames per second (FPS) depends on your display and hardware; it can be 60, 120, or more.
- **What it does**: You draw everything the player should see. The `screen` parameter is the image that will be shown in the window. You draw sprites, shapes, and text onto it.
- **Important**: Do not update game state here. Keep `Draw` fast and focused on drawing only.

`Layout(outsideWidth, outsideHeight int) (int, int)` – Screen size

- **When it is called**: When the window is created or resized.
- **Parameters**: `outsideWidth` and `outsideHeight` are the actual window size in pixels.

- **Return value**: You return the **logical** size you want for your game. Ebitengine then scales your drawing to fit the window. For example, if you return `(640, 480)`, your game "thinks" it has a 640×480 canvas; Ebitengine stretches or shrinks it to match the real window.

Tick vs frame: A **tick** is one step of game logic; a **frame** is one drawn image. At 60 TPS, `Update` runs 60 times per second. `Draw` typically runs as often as the display allows (often 60 or 120 FPS). For most 2D games, 60 updates per second is enough; the logic stays stable even if the frame rate varies.

The Update—Draw Flow: How the Loop Works

Understanding the order and rhythm of the game loop is essential.

Here is how Ebitengine runs your game.

Sequence of execution:

When you call `ebiten.RunGame(game)`, Ebitengine enters a loop. Each iteration looks like this:

```
1. Layout (if needed)  —  window created or resized?
2. Update              —  run your game logic (60 times/sec by default)
3. Draw                —  render the current state to the screen
4. Swap buffers        —  show the new frame to the player
5. Repeat
```

`Layout` is called only when the window is first created or when the user resizes it. `Update` and `Draw` run every iteration. There is no separate "physics step" or "input step"—you do all of that inside `Update`. Input, movement, collisions, AI, and timers belong in `Update`. Drawing belongs in `Draw`.

Why separate Update and Draw?

- **Determinism**: Game logic runs at a fixed rate (60 TPS). If you tied logic to frames, a slow machine would run the game slower—enemies would move slower, timers would stretch. With fixed TPS, one second of game time is always one second of logic, regardless of frame rate.
- **Performance**: Drawing can run at the display's refresh rate (60, 120, 144 Hz). On a high-refresh monitor, `Draw` runs more often than `Update`. The screen looks smoother; the logic stays consistent.

- **Simplicity**: One place for logic, one place for rendering. No mixing concerns. No risk of reading input in the middle of a draw.

Timing in practice:

Setting	Default	Meaning
TPS (ticks per second)	60	How often `Update` is called. You can change it with `ebiten.SetTPS()`.
Max FPS	Unlimited	`Draw` runs as often as possible, up to the display refresh rate (vsync on) or higher (vsync off).
Uncapped FPS	–	If vsync is off, `Draw` can run hundreds of times per second. `Update` still runs at TPS.

How Ebitengine manages timing: The engine works hard to keep `Update` running at a stable 60 calls per second, regardless of how fast or slow your machine is. It waits or adjusts between ticks so that one second of real time corresponds to 60 ticks of game logic. For drawing, Ebitengine typically syncs with the monitor's refresh rate (vsync): if your display runs at 60 Hz, you get 60 frames per second; at 120 Hz, you get 120 frames. This separation means your game logic stays deterministic and predictable, while the visuals stay smooth. You do not need to write any timing code yourself–Ebitengine handles it.

Rule of thumb: Read input and change game state in `Update`. In `Draw`, only read the current state and render it. Do not mutate state in `Draw`–you would get inconsistent behavior if `Draw` is called multiple times between `Update` calls (which can happen when FPS > TPS).

Creating a Go Module for Your Project

Before writing any code, you need a **Go module**. If you are new to Go: a module is a collection of Go source files that form a unit. It has a name (a "module path"), a `go.mod` file that lists dependencies, and one or more packages. When

you want to use an external library like Ebitengine, you add it as a **dependency** of your module. The `go.mod` file records which libraries you use and their versions, so your project can be built consistently on any machine.

Step 1: Create the Project Directory

Create a directory for your game. The name is up to you; for this book we use `gopher-survivor`:

```
# Run from: (parent of project directory)
mkdir gopher-survivor
cd gopher-survivor
```

Step 2: Initialize the Module

Run `go mod init` with a module path. The path identifies your module and should be unique—convention is to use a URL you control, e.g. `github.com/yourusername/gopher-survivor`:

```
# Run from: ch01 (or project root)
go mod init github.com/yourusername/gopher-survivor
```

You can use any path (`example.com/gopher-survivor`, `gopher-survivor`, etc.) as long as you do not publish the module publicly. You can change it later.

This creates a `go.mod` file:

```
module github.com/yourusername/gopher-survivor

go 1.22
```

- `module` is the import path for packages in this project.
- `go` is the minimum Go version required.

Step 3: Add Ebitengine as a Dependency

Add Ebitengine to your module:

```
# Run from: ch01
go get github.com/hajimehoshi/ebiten/v2
```

Go downloads the library and records it in `go.mod` and `go.sum`:

```
module github.com/yourusername/gopher-survivor

go 1.22

require github.com/hajimehoshi/ebiten/v2 v2.8.6
```

From now on, you can import Ebitengine in your code with `"github.com/hajimehoshi/ebiten/v2"`. When you run `go build` or `go run`, Go ensures the dependency is available.

Project Structure

For the Chapter 1 example, the structure looks like this:

```
gopher-survivor/
├── go.mod
├── game.go
├── settings.go
├── cmd/
│   └── main.go
└── assets/
    ├── embed.go
    └── golang_icon.png
```

We split the code into two packages: `game` (the `Game` struct and its methods, in `game.go` and `settings.go`) and `cmd` (the entry point with `main`). The `game` package contains the game logic; `cmd` contains the program that starts the loop and turns embedded bytes into an `*ebiten.Image`. Static files such as the Gopher PNG live under `assets/`; the next section shows how you bake them into the binary with `go:embed` so you do not rely on the working directory at runtime.

Building the Code Step by Step

We will build a small program that loads the Go Gopher icon from a PNG file and draws it centered on the screen. Follow each step: create the files, add the

code, and read the explanation. By the end you will have a complete, runnable program.

Step 1: Create `game.go` — Package and the Game Struct

Inside `ch01/`, create a file named `game.go`. It defines the `game` package – the `Game` struct and its methods. Add the following:

```go
// File: ch01/game.go
package game

import (
    "github.com/hajimehoshi/ebiten/v2"
)

// Game holds the game state and implements the ebiten.Game interface.
type Game struct {
    gopherImage *ebiten.Image
}

// NewGame creates a new Game with the given gopher image.
func NewGame(gopherImage *ebiten.Image) *Game {
    return &Game{gopherImage: gopherImage}
}
```

What we added:

- **`package game`** – This file belongs to the `game` package. We separate game logic from the entry point (`main`).
- **`import "github.com/hajimehoshi/ebiten/v2"`** – We need the core Ebitengine package for `ebiten.Image`, `DrawImageOptions`, and the game loop.
- **`type Game struct { gopherImage *ebiten.Image }`** – The `Game` struct holds our game data. The field `gopherImage` stores the loaded texture. We load it once at startup and reuse it every frame in `Draw`. In Ebitengine, everything drawable is an `ebiten.Image`–the screen, sprites, textures. You draw one image onto another.
- **`NewGame(gopherImage *ebiten.Image) *Game`** – A constructor that creates a `Game` with the loaded image. Since `gopherImage` is unexported, other packages use this function to create a `Game`.

Store the Screen Size: `settings.go`

Rather than scatter magic numbers such as `640` and `480` through the code, keep the logical screen size in one place. Inside `ch01/`, next to `game.go`, create `settings.go`:

```
// File: ch01/settings.go
package game

// GameSettings holds all tunable parameters for the game.
// Changing a value here is the only edit needed to adjust the configuration.
type GameSettings struct {
    ScreenWidth  int
    ScreenHeight int
}

// Settings is the single source of truth for all game parameters.
var Settings = GameSettings{
    ScreenWidth:  640,
    ScreenHeight: 480,
}
```

What we added:

- **`GameSettings`** – A struct that groups tunable parameters. For now it holds only the logical screen size, but as the game grows you add fields here instead of sprinkling constants across files.
- **`Settings`** – A single package-level value: the one source of truth for these numbers. `Draw`, `Layout`, and the `cmd` entry point all read `Settings.ScreenWidth` and `Settings.ScreenHeight`, so changing the window size is a one-line edit.

Because `settings.go` belongs to the same `game` package as `game.go`, the `Settings` value is available to every method on `Game` without an import.

Step 2: Implement `Update` in `game.go`

Add this method to `game.go`, right after the `NewGame` function:

```go
// File: ch01/game.go
// Update is called every tick (default 60 times per second).
func (g *Game) Update() error {
    return nil
}
```

What we added:

- **`func (g *Game) Update() error`** – This method is part of the `ebiten.Game` interface. Ebitengine calls it 60 times per second. Here you would put game logic: input, movement, collisions. For this example we only display a static image, so we do nothing.
- **`return nil`** – `nil` means "no error"; the game continues. If you return an error, Ebitengine stops the loop and exits.

Step 3: Implement `Draw` in `game.go`

Add this method after `Update`. `Draw` receives the screen as an `*ebiten.Image`; we draw our texture onto it.

```go
// File: ch01/game.go
func (g *Game) Draw(screen *ebiten.Image) {
    gopherWidth := float64(g.gopherImage.Bounds().Dx())
    gopherHeight := float64(g.gopherImage.Bounds().Dy())
    x := (float64(Settings.ScreenWidth) - gopherWidth) / 2
    y := (float64(Settings.ScreenHeight) - gopherHeight) / 2

    op := &ebiten.DrawImageOptions{}
    op.GeoM.Translate(x, y)
    screen.DrawImage(g.gopherImage, op)
}
```

Getting the image size: `Bounds()`, `Dx()`, `Dy()`

- `g.gopherImage.Bounds()` returns a `image.Rectangle` that describes the image's bounds (its minimum and maximum X and Y coordinates).
- `.Dx()` returns the width in pixels (right minus left).
- `.Dy()` returns the height in pixels (bottom minus top).
- We convert to `float64` because `Translate` and other transform functions use `float64` coordinates. This allows sub-pixel positioning when scaling.

Centering the image

- To center the Gopher horizontally, we place its left edge at `(float64(Settings.ScreenWidth) - gopherWidth) / 2`. We cast `Settings.ScreenWidth` (an `int`) to `float64` so it mixes with the float widths; dividing by two leaves equal space on both sides.
- Similarly, vertically: `(float64(Settings.ScreenHeight) - gopherHeight) / 2`.

`ebiten.DrawImageOptions`

- This struct controls *how* an image is drawn: where, at what size, with what rotation, and with optional color effects.
- We create an empty one with `&ebiten.DrawImageOptions{}`. By default, the image is drawn at full size, no rotation, at position (0, 0).

`GeoM` and `Translate(x, y float64)`

- `GeoM` is a 2D affine transformation matrix. It can represent translation (moving), scaling, and rotation.
- `Translate(x, y)` adds a translation: it moves the image so that its top-left corner ends up at `(x, y)`.
- You can chain multiple transforms: for example, `Scale` then `Translate` to draw a scaled image at a given position. Order matters: transforms apply from right to left in the matrix.

Coordinate System and GeoM: A Closer Look

Understanding how Ebitengine handles coordinates and the `GeoM` matrix is essential for positioning, scaling, and rotating sprites correctly.

The Coordinate System

Ebitengine uses a coordinate system that may differ from what you learned in mathematics:

Aspect	Ebitengine	Traditional math
Origin	Top-left corner (0, 0)	Often center or bottom-left
X axis	Increases to the **right**	Same
Y axis	Increases **downward**	Usually increases **upward**

So a point at (100, 200) is 100 pixels from the left edge and 200 pixels *below* the top edge. This matches how 2D images and most graphics APIs work: the first row of pixels is at y=0, the second at y=1, and so on.

Each `ebiten.Image` has its own coordinate system. When you draw onto the screen, the screen's top-left is (0, 0). When you draw onto an offscreen image, that image's top-left is (0, 0). The destination image defines the coordinate space.

Why Y goes down: In image formats (PNG, JPEG) and framebuffers, rows are stored top-to-bottom. Pixel (0, 0) is the first pixel; pixel (0, 1) is the second row. Mapping this directly to coordinates makes rendering straightforward: no need to flip or invert.

What Is GeoM?

`GeoM` (Geometry Matrix) is Ebitengine's way of specifying *where* and *how* to draw an image. It is a **2D affine transformation matrix**—a 3×3 matrix where the last row is always (0, 0, 1):

```
[a   b   tx]
[c   d   ty]
[0   0   1 ]
```

- **a, b, c, d** – Control scaling and rotation. A 2×2 matrix alone cannot move the origin; it can only scale and rotate around (0, 0).
- **tx, ty** – Translation: how much to shift the image in X and Y. These allow moving the image to any position.

The matrix defines a *mapping*: for each point (x, y) in the source image, it computes where that point ends up in the destination. By adjusting the matrix, you control position, scale, and rotation in one unified way.

Identity matrix: A new `GeoM` starts as the identity matrix (no transformation). In that state, the image is drawn at (0, 0)—its top-left corner at the destination's top-left.

GeoM Operations: Translate, Scale, Rotate

`GeoM` provides three main methods; each *left-multiplies* a new transformation into the matrix:

Method	Effect	Origin (pivot point)
`Translate(x, y)`	Moves the image by x pixels right, y pixels down	–
`Scale(sx, sy)`	Scales the image by sx in X, sy in Y	Top-left corner
`Rotate(angle)`	Rotates by the given angle (radians)	Top-left corner

Translate: `op.GeoM.Translate(100, 50)` moves the image so its top-left corner ends up at (100, 50) in the destination. Positive x = right; positive y = down.

Scale and Rotate: The *origin* for both is the **top-left corner** of the image. If you scale by 2, the image grows to the right and downward from that corner. If you rotate, the top-left corner stays fixed and the rest of the image spins around it. To rotate around the center, you typically: translate to center, rotate, then translate back.

Order of Operations

When you chain operations, the order matters. Each method *left-multiplies* a new transformation into the matrix. The *first* operation you call is applied *first* to the image (to each pixel); the last is applied last.

```
1 // File: ch01/game.go — GeoM usage example
2 op.GeoM.Scale(2, 2)         // Applied first: scale 2x (origin at top-left)
3 op.GeoM.Translate(100, 100)  // Applied second: move the scaled image to (100,
  ↪  100)
4 // Result: image is scaled, then moved. The scaled image's top-left ends at
  ↪  (100, 100).
```

If you reversed the order (Translate then Scale), you would move first, then scale—and the translation would be scaled too, so the top-left would end up at

(200, 200) instead of (100, 100). A common pattern is: Scale □ Rotate □ Translate (scale and rotate around the top-left origin, then move to the final position).

```
screen.DrawImage(src *ebiten.Image, op *DrawImageOptions)
```

- **Purpose**: Draws `src` onto `screen` using the options in `op`.
- **Parameters**:
  - `src` – The image to draw (our Gopher).
  - `op` – How to draw it (position, scale, rotation, etc.).
- **Important**: In Ebitengine, drawing is always "draw one image onto another." The screen is an image; your sprites are images. You compose them by calling `DrawImage` repeatedly.

Step 4: Implement `Layout` in `game.go`

Add this method after `Draw`:

```
1  // File: ch01/game.go
2  func (g *Game) Layout(outsideWidth, outsideHeight int) (int, int) {
3      return Settings.ScreenWidth, Settings.ScreenHeight
4  }
```

What we added: `Layout` is called when the window is created or resized. We return the **logical** size from `Settings` (640×480); Ebitengine scales our drawing to fit the real window. A fixed logical size keeps positions and layouts consistent.

Embedding the PNG with `go:embed`

Your `main` package will decode PNG **bytes** and pass an `*ebiten.Image` into `NewGame`. You could read `golang_icon.png` from disk at runtime, but then the path depends on the working directory and how you ship the game. For a small icon, **embedding** is simpler: the Go toolchain copies the file into the executable at compile time, and your code reads a `[]byte` variable. After `go build`, you do not need to ship the PNG next to the binary unless you want to.

Go provides this through the **`embed`** package and the **`//go:embed`** directive (available since Go 1.16). You put the directive in a source file in the same directory as the file you embed (or use a path relative to that file).

Create `assets/embed.go` in your `assets` folder:

```go
// File: ch01/assets/embed.go
package assets

import _ "embed"

//go:embed golang_icon.png
var GopherPNG []byte
```

What this does:

- **`import _ "embed"`** - A **blank import** of `embed` registers the compiler support for `//go:embed`. You do not call `embed` from your code; the directive does the work.
- **`//go:embed golang_icon.png`** - Tells the compiler to read `golang_icon.png` from the `assets` directory (same folder as this `.go` file) and initialize the variable on the next line with its raw bytes.
- **`var GopherPNG []byte`** - Holds the PNG file exactly as on disk. Other packages import `yourmodule/assets` and use `assets.GopherPNG` like any `[]byte` slice (for example with `bytes.NewReader` and `image.Decode`).

Important: `go:embed` reads the file at **compile** time. `golang_icon.png` must exist in `assets/` before you run `go build` or `go run`; otherwise the build fails.

Tip: You can embed multiple files or whole folders; see the embed package documentation for patterns and constraints.

Step 5: Create `cmd/main.go` - Decode the Embedded PNG and Start the Game

Create a `cmd` folder and inside it a new file `main.go`. The entry point decodes `assets.GopherPNG` (from `assets/embed.go` above), converts the result to `*ebiten.Image`, creates the game, and starts Ebitengine:

```go
// File: ch01/cmd/main.go
package main

import (
    "bytes"
    "image"
    _ "image/png"
    "log"

    "book/ch01"
    "book/ch01/assets"

    "github.com/hajimehoshi/ebiten/v2"
)

func main() {
    img, _, err := image.Decode(bytes.NewReader(assets.GopherPNG))
    if err != nil {
        log.Fatalf("failed to decode embedded gopher image: %v", err)
    }
    eimg := ebiten.NewImageFromImage(img)

    g := game.NewGame(eimg)

    ebiten.SetWindowSize(game.Settings.ScreenWidth, game.Settings.ScreenHeight)
    ebiten.SetWindowTitle("Chapter 1: Hello Ebiten - Go Gopher")

    if err := ebiten.RunGame(g); err != nil {
        log.Fatal(err)
    }
}
```

What each part does:

1. **`import "book/ch01"`** – Imports the chapter package that defines `Game`, `NewGame`, and `Settings`.
2. **`assets.GopherPNG`** - The `[]byte` filled by `//go:embed` in `assets/embed.go` (see Embedding the PNG with `go:embed`).
3. **`image.Decode(bytes.NewReader(...))`** – Decodes those bytes into a generic `image.Image`.
4. **`ebiten.NewImageFromImage(img)`** – Converts the decoded image into an `*ebiten.Image`, which Ebitengine can draw efficiently.
5. **`g := game.NewGame(eimg)`** – Creates the game and passes it the loaded texture.

6. **`ebiten.SetWindowSize(...)`** – Uses `Settings.ScreenWidth` and `Settings.ScreenHeight` from the chapter package, keeping window size in one place.
7. **`ebiten.RunGame(g)`** – Starts the game loop and repeatedly calls `Layout`, `Update`, and `Draw`.

Step 6: Add the Asset and Run

Create an `assets` folder at the project root if you have not already (you need it for `embed.go`). Place the Go Gopher icon as `golang_icon.png` in that folder next to `embed.go`. From your project directory, run:

```
# Run from: ch01
go run ./cmd
```

A window opens with the Go Gopher centered on the screen. You have built your first Ebitengine program.

Full Source Code

Here is the complete code. `ch01/game.go` contains the `Game` struct and its methods; `ch01/settings.go` stores the logical window size; `ch01/assets/embed.go` embeds the PNG at compile time; `ch01/cmd/main.go` contains the executable entrypoint and decodes those bytes into an `*ebiten.Image`.

`game.go`:

```go
// File: ch01/game.go
package game

import (
    "github.com/hajimehoshi/ebiten/v2"
)

type Game struct {
    gopherImage *ebiten.Image
}

func NewGame(gopherImage *ebiten.Image) *Game {
    return &Game{gopherImage: gopherImage}
}

func (g *Game) Update() error {
    return nil
}

func (g *Game) Draw(screen *ebiten.Image) {
    gopherWidth := float64(g.gopherImage.Bounds().Dx())
    gopherHeight := float64(g.gopherImage.Bounds().Dy())
    x := (float64(Settings.ScreenWidth) - gopherWidth) / 2
    y := (float64(Settings.ScreenHeight) - gopherHeight) / 2

    op := &ebiten.DrawImageOptions{}
    op.GeoM.Translate(x, y)
    screen.DrawImage(g.gopherImage, op)
}

func (g *Game) Layout(outsideWidth, outsideHeight int) (int, int) {
    return Settings.ScreenWidth, Settings.ScreenHeight
}
```

settings.go:

```go
// File: ch01/settings.go
package game

type GameSettings struct {
    ScreenWidth  int
    ScreenHeight int
}

var Settings = GameSettings{
    ScreenWidth:  640,
    ScreenHeight: 480,
}
```

cmd/main.go:

```go
// File: ch01/cmd/main.go
package main

import (
    "bytes"
    "image"
    _ "image/png"
    "log"

    "book/ch01"
    "book/ch01/assets"

    "github.com/hajimehoshi/ebiten/v2"
)

func main() {
    img, _, err := image.Decode(bytes.NewReader(assets.GopherPNG))
    if err != nil {
        log.Fatalf("failed to decode embedded gopher image: %v", err)
    }
    eimg := ebiten.NewImageFromImage(img)

    g := game.NewGame(eimg)

    ebiten.SetWindowSize(game.Settings.ScreenWidth, game.Settings.ScreenHeight)
    ebiten.SetWindowTitle("Chapter 1: Hello Ebiten - Go Gopher")

    if err := ebiten.RunGame(g); err != nil {
        log.Fatal(err)
    }
}
```

The complete example is in `ch01/`. The structure:

```
ch01/
├── go.mod
├── game.go
├── settings.go
├── cmd/
│   └── main.go
└── assets/
    ├── embed.go
    └── golang_icon.png
```

Run it with:

```
# Run from: project root
cd ch01
go run ./cmd
```

The following screenshot shows the result of this chapter:

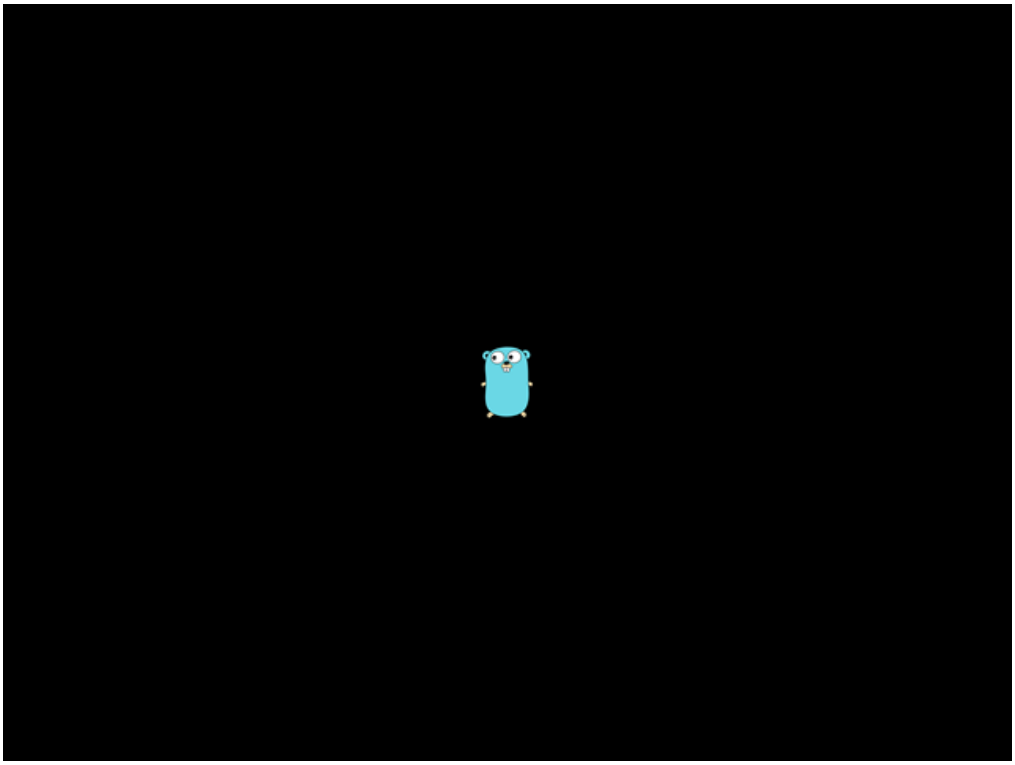


Figure 3. Chapter 1 result: the Go Gopher centered in a 640x480 window

Summary

In this chapter, you learned:

- **`go mod init`** – creates a Go module and a `go.mod` file.
- **`go get`** – adds a dependency to the module.
- **`//go:embed` and the `embed` import** – bake static files into the binary as `[]byte` (or an `embed.FS`) at compile time.
- **`ebiten.Game`** – the interface you implement with `Update`, `Draw`, and `Layout`.
- **`Update()`** – called every tick; put game logic here.
- **`Draw(screen)`** – called every frame; put all drawing here.
- **`Layout()`** – returns the logical screen size; Ebitengine scales it to the window.
- **`image.Decode` + `ebiten.NewImageFromImage`** – decode embedded image bytes and convert them to an `*ebiten.Image`.
- **`ebiten.Image`** – the type for every drawable image in Ebitengine.
- **The coordinate system** – origin at the top-left; Y increases downward; each image has its own coordinate space.
- **`GeoM`** – the affine matrix for position, scale, and rotation. `Translate` moves; `Scale` and `Rotate` use the top-left as pivot.
- **`screen.DrawImage`** – draws one image onto another.

* * *

Get the code: the complete, runnable project for this chapter is in the `ch01` directory of the companion GitHub repository.

Chapter 2: The Scene Graph and Framework Structure

This chapter introduces the core architecture of our game framework: the **scene graph**, a parent-child structure of nodes that organizes the game world and its transforms. You will implement the `Engine`, `SceneNode`, `Node`, `Node2D`, and `Sprite` types, and you will connect them to Ebitengine's draw pipeline so the scene graph turns into pixels on screen.

In this chapter, we will cover the following topics:

- How a scene graph models parent-child relationships in a 2D game
- How local transforms become world transforms through hierarchy traversal
- How to map our custom transform data to `ebiten.GeoM`
- How the `Engine` and `World` types traverse the graph and submit draw calls
- How to structure a small but extensible framework in Go packages (including `internal/core`)
- How to build a rotating sprite example on top of the framework

Technical requirements

To follow this chapter, you need:

- Go `1.22.0`
- Ebitengine `v2.8.6`
- The embedded `golang_icon.png` asset in `ch02/assets`

Why a Scene Graph?

In Chapter 1, we drew a single image directly to the screen. Real games have hundreds of objects: characters, enemies, projectiles, UI elements. Managing each one separately quickly becomes unmanageable. A **scene graph** solves this by organizing everything as a graph of parent and child nodes: each node has a parent and can have children. Transforms (position, rotation, scale) are **relative** to the parent. Move a parent, and all its children move with it—just like a skeleton: move the torso, and the arms follow.

This pattern is used by Godot, Unity, and many other engines. Our framework follows the same idea.

Before we build the scene graph, we need a type for **positions** and **directions** in 2D space. That leads us to vectors and a bit of 2D math.

Vector2D: 2D Coordinates and a Bit of Vector Math

In 2D games, every object has a **position** (where it is) and often a **direction** (where it faces or moves). Both can be represented with a **2D vector**: a pair of numbers (x, y). The same structure serves as:

- **Point/position**: "the sprite is at (100, 200) pixels"
- **Direction/offset**: "move 3 units right and 2 units up"
- **Delta/velocity**: "the bullet moves 5 pixels per frame along (1, 0)"

Some 2D Vector Math

Operation	Formula	Use case
Addition	(x1,y1) + (x2,y2) = (x1+x2, y1+y2)	Position + movement, translate by offset
Scalar mult.	k * (x,y) = (k*x*, k*y*)	Scaling, speed, "move N units in direction D"
Magnitude	□v□ = sqrt(x*x* + y*y*)	Distance, collision radii

Operation	Formula	Use case
Normalization	v_unit = v / ▯v▯ (if ▯v▯ > 0)	Unit vector (length 1) in same direction
Rotation	x_new = x*cos(theta)* - *y*sin(theta) y_new = x*sin(theta)* + *y*cos(theta)	Rotate vector by angle theta around origin (used when combining transforms)

Addition: move by one vector, then another — the result is the combined displacement.

Scalar multiplication: stretch or shrink a vector. Multiply both components by k.

Magnitude: the distance from the origin. Used for distances, collision radii, and normalization.

Normalization: divide the vector by its length to get a unit vector (length 1) in the same direction.

Rotation: the 2D rotation matrix. Given angle theta (in radians), the new coordinates are computed as in the table. This rotates the vector around the origin; we use it when combining transforms (e.g. parent rotation × child position).

Our framework uses these ideas through a simple `Vector2D` type. In this project, vectors describe values in **world space** (and later, **screen space** once you add a camera), both expressed as `float64` so they map cleanly to Ebitengine's transform API. Even if your sprites are aligned to pixels most of the time, using floating-point vectors keeps rotation, interpolation, and camera movement smooth as the framework grows.

The table above is a **reference** for the vector operations you will use in almost every 2D game. The Chapter 2 source in `vector2d.go` is intentionally small: it gives you constructors, component access, and `RotateVector`, because transform hierarchy already needs rotation of offsets inside `Transform.Concat`. You can add helpers later (`Add`, `Sub`, `Mul`, length, normalization, dot product) without changing the scene graph design—keep the API consistent and prefer methods that return new values when you want immutable-style call chains.

Where to put framework code (internal and core packages)

The first file you add for this chapter belongs at **`ch02/internal/core/vector2d.go`**. Under your module root (`ch02/`), create the directories `internal/` and `internal/core/`, then add `vector2d.go` with `package core` as the first line (after imports). Every other framework type in this chapter lives in that same package path with the same `package core` declaration.

Why `internal/`? — In Go, the folder name `internal` is special. Packages whose import path contains `/internal/` can be imported only by code **inside** the tree rooted at the parent of that `internal` directory. Here, only packages under `ch02` may import `book/ch02/internal/...`. Another module or a standalone tool cannot import your scene graph by accident. Exported names (`Node`, `World`, and so on) remain available to **your** `game` package in this module; `internal` simply blocks **external** importers.

Why `core/`? — All foundational types for this chapter (`Vector2D`, `Node`, `World`, `Engine`, and the rest) live in one package named **`core`**. Files at the module root (`game.go`, `run.go`) use **`package game`**: they connect Ebitengine, load assets, and build the demo. That split keeps responsibilities clear: open `internal/core` for reusable engine mechanics; open `game.go` for the sample's `Update` / `Draw` glue. Later chapters can add sibling packages under `internal/` without crowding a single directory.

> **Tip:** The inner folder could be named `scenegraph` or `engine` instead of `core`. The book uses `core` for "shared foundation of the game." The part that matters most for tooling and readers is **`internal`**.

The concrete code starts with `Vector2D`; you will use it everywhere for positions, pivots, scales, and movement.

```go
// File: ch02/internal/core/vector2d.go
type Vector2D struct {
    x, y float64
}

func NewVector2D(x, y float64) Vector2D       { return Vector2D{x: x, y: y} }
func ZeroVector2D() Vector2D                  { return Vector2D{0, 0} }

func (v Vector2D) X() float64                  { return v.x }
func (v Vector2D) Y() float64                  { return v.y }
func (v *Vector2D) SetX(x float64)            { v.x = x }
func (v *Vector2D) SetY(y float64)             { v.y = y }
func (v *Vector2D) SetPosition(other Vector2D) { v.x, v.y = other.x, other.y }
```

`NewVector2D` and `ZeroVector2D` create vectors. The getters and setters read or write the components. `SetPosition` copies another vector's coordinates.

```go
// File: ch02/internal/core/vector2d.go
func (v Vector2D) RotateVector(radians float64) Vector2D {
    s, c := math.Sin(radians), math.Cos(radians)
    return Vector2D{v.x*c - v.y*s, v.x*s + v.y*c}
}
```

Rotation returns a new vector rotated by the given angle (in radians). It implements the 2D rotation formula above. We use it when combining transforms—e.g. rotating the child's position by the parent's rotation.

With `Vector2D` in place, we can build the scene graph nodes and their transforms.

The SceneNode Interface

```go
// File: ch02/internal/core/scenenode.go
type SceneNode interface {
    GetID() uint64
    GetName() string
    AddChildren(child SceneNode)
    GetChildren() []SceneNode
    AttachParent(node SceneNode)
    GetParent() SceneNode
    DetachChild(node SceneNode) bool
    MarkDirty()
}
```

The foundation of our scene graph is the **SceneNode** interface. Every element in the scene graph implements it: the root, layers, sprites, and any custom node type. The interface defines the minimal operations needed to build and traverse a hierarchy.

- **`GetID()`** – Returns the node's unique identifier. Useful for lookups, debugging, and serialization.
- **`GetName()`** – Returns the human-readable name (e.g. `"player"`, `"enemy_01"`). Helps when debugging.
- **`AddChildren(child)`** – Attaches a child to this node. The child becomes part of this node's subtree.
- **`GetChildren()`** – Returns all direct children. Used when traversing the scene graph (Update, Draw). The slice is the **live backing store** for that node's child list: if you add or remove children while iterating over the same slice, you can skip nodes or see inconsistent order. For simple games you mutate the scene graph only from `Update` before traversal, or you copy children to a temporary slice when you need a stable snapshot.
- **`AttachParent(node)`** – Sets the parent of this node. Called by `AddChildren` on the child; you rarely call it directly.
- **`GetParent()`** – Returns the parent node, or `nil` if this is the root.
- **`DetachChild(node)`** – Removes a specific child from the list. Returns `true` if the child was found and removed.
- **`MarkDirty()`** – Marks this node (and optionally its descendants) as needing transform recalculation.

Why an interface? The engine and the World traverse nodes without knowing their concrete type. They only need to call `GetChildren()`, `GetParent()`, etc. This lets us add new node types (cameras, particle emitters, UI panels) without changing the traversal code.

The Node Struct: Base Implementation

```
// File: ch02/internal/core/node.go
type Node struct {
    id       uint64
    name     string
    children []SceneNode
    parent   SceneNode
}
```

`Node` is the simplest implementation of `SceneNode`. It holds an ID, a name, a parent reference, and a slice of children. It has **no spatial data**—no position, no rotation. Use it for logical groupings (e.g. a "Game" node that groups all gameplay nodes) or nodes that do not need to be drawn.

- **`id`** – Unique identifier for the node. Assigned at creation via `atomic.AddUint64`; used for lookups, debugging, and serialization. No two nodes share the same ID.
- **`name`** – Human-readable label (e.g. `"player"`, `"enemy_01"`, `"root"`). Useful when debugging or inspecting the scene graph.
- **`children`** – Slice of `SceneNode`; the direct children of this node. Traversal (Update, Draw) iterates over this slice to visit descendants.
- **`parent`** – Reference to the parent `SceneNode`. `nil` for the root. The child stores the parent; the parent stores the child in `children`. Both links must stay consistent.

NewNode and the ID

```
// File: ch02/internal/core/node.go
var globalNodeID uint64

func NewNode(name string) *Node {
    id := atomic.AddUint64(&globalNodeID, 1)
    return &Node{id: id, name: name, children: make([]SceneNode, 0)}
}
```

Each node needs a unique ID. We use `atomic.AddUint64` to generate IDs safely from multiple goroutines (or when creating nodes concurrently).

Understanding `atomic.AddUint64(&globalNodeID, 1)`: This is an **atomic operation** from the `sync/atomic` package. It does three things in a single, indivisible step: (1) read the current value of `globalNodeID`; (2) add `1` to it; (3) write the result back and return the new value. No other goroutine can see an intermediate state or interleave between these steps. Without atomicity, two goroutines creating nodes at the same time could read the same value (e.g. 5), both add 1, and both get ID 6—a race condition. With `atomic.AddUint64`, each call gets a unique, monotonically increasing ID. The first call returns 1, the second 2, and so on. The `&globalNodeID` is a pointer to the shared variable; the `1` is the delta to add. For a single-threaded game loop, a plain `globalNodeID++` would work, but using atomics makes the code safe if you ever create nodes from another goroutine (e.g. during async asset loading).

GetID, GetName, SetName, GetParent

```
1  // File: ch02/internal/core/node.go
2  func (n *Node) GetID() uint64        { return n.id }
3  func (n *Node) GetName() string      { return n.name }
4  func (n *Node) SetName(name string)  { n.name = name }
5  func (n *Node) GetParent() SceneNode { return n.parent }
```

These getters and setters read and write the node's basic fields:

- **`GetID()`** – Returns the node's unique numeric ID. Use it for lookups (e.g. find a node by ID), debugging, or serialization. You typically do not change the ID after creation.
- **`GetName()`** – Returns the human-readable name (e.g. `"player"`, `"root"`). Useful when inspecting the scene graph or logging.
- **`SetName(name string)`** – Changes the node's name after creation. Call this when you need to rename a node dynamically (e.g. when reusing a node for a different purpose).
- **`GetParent()`** – Returns the parent `SceneNode`, or `nil` if this node has no parent (it is the root). `Node2D.GetWorldTransform()` uses this to walk up the scene graph and combine transforms.

AddChildren and Parent-Child Linking

When you add a child, two links must stay in sync: (1) the child's `parent` field must point to its parent; (2) the parent's `children` slice must include the child. We factor this into a helper that separates *whose children list we append to* from *who is stored as the parent*. To understand why we need that separation, we first need a short primer on Go methods and struct embedding.

Go methods and the receiver

In Go, a method is a function that "belongs" to a type. The **receiver** is the value the method operates on—it appears in parentheses before the method name:

```
// File: ch02/internal/core/node.go
func (n *Node) AddChildren(child SceneNode) {
    // n is the receiver: the specific Node this method was called on
}
```

When you call `someNode.AddChildren(sprite)`, the receiver `n` inside the method is `someNode`. The method sees that exact value. For a plain `*Node`, there is nothing subtle: `n` is the node you called the method on.

Struct embedding: Node2D contains a Node

`Node2D` embeds `Node`—it has a `Node` as its first field (without a field name), so the `Node` lives *inside* `Node2D`:

```
// File: ch02/internal/core/node2d.go
type Node2D struct {
    Node              // embedded: Node2D has a Node inside it
    localTransform Transform
    // ...
}
```

In memory, a `Node2D` looks like: `[Node fields][localTransform][worldTransform][`
The `Node` is a real field; you can access it as `logo.Node` if needed. Because it is embedded (no field name), its fields and methods are **promoted**: you can write `logo.GetParent()` and Go treats it as if `Node2D` had that method—it forwards to the embedded `Node`.

Method promotion: which value becomes the receiver?

Here is the subtle part. When you call `logo.AddChildren(sprite)` on a `*Node2D`, Go looks for an `AddChildren` method. `Node2D` does not define it, so Go uses the **promoted** method from the embedded `Node`. The crucial rule: when a promoted method runs, the receiver passed to it is the **embedded field**, not the outer struct.

So `logo.AddChildren(sprite)` effectively becomes `(&logo.Node).AddChildren(spr` Inside `Node.AddChildren`, the receiver `n` is `&logo.Node`—the address of the `Node` *inside* `Node2D`—not `logo` itself. They are different values: `logo` is the full `Node2D`; `logo.Node` is just the embedded `Node` part.

Why that causes a bug

Inside `Node.AddChildren` we have `n.addChildWithParent(n, child)`. The receiver `n` is `&logo.Node`, so we pass `&logo.Node` as the logical parent. That means `sprite.parent` is set to `&logo.Node` (a `*Node`), not to `logo` (a `*Node2D`).

Later, when the sprite draws, it calls `GetWorldTransform()`. To combine with its parent's transform, it does:

```
1  parent := sprite.GetParent()  // returns &logo.Node (a *Node)
2  if pt, ok := parent.(Transformable); ok {
3      world = pt.GetWorldTransform()
4  }
```

The type assertion `parent.(Transformable)` asks: "Does the value in `parent` implement `Transformable`?" A `*Node` does *not*—it has no position, rotation, or `GetWorldTransform`. So `ok` is `false`, we skip the block, and the sprite ignores its parent's transform. It draws as if it had no parent, typically at the wrong position.

The fix: separate "whose children" from "who is the parent"

The helper `addChildWithParent(logicalParent, child)` takes two separate concerns:

1. **`logicalParent`** – Who gets stored in `child.parent`? This is the node the child will call `GetParent()` on and use for world-transform inheritance. It must implement `Transformable` if we want the child to inherit position and rotation.

2. **n (the receiver)** – Whose `children` slice do we append to? This is always the `Node` we're modifying–either a standalone `Node` or the embedded `Node` inside a `Node2D`.

For a plain `Node`, the receiver *is* the parent: `AddChildren` calls `addChildWithParent(n, child)` and both roles are the same.

For `Node2D`, we override `AddChildren` and call `n.Node.addChildWithParent(n, child)`. Here, `n` is the `*Node2D`, so we pass the full `Node2D` as `logicalParent`. The receiver `n.Node` tells us which `children` slice to update (the one inside the embedded `Node`). Now `sprite.parent` is `*Node2D`, which implements `Transformable`, so the sprite correctly inherits the logo's transform.

```
// File: ch02/internal/core/node.go
// addChildWithParent attaches child to logicalParent and appends to
↪  n.children.
// logicalParent: the node stored in child.parent (used when computing world
↪  transforms).
// n: the Node whose children slice we append to (receiver).
func (n *Node) addChildWithParent(logicalParent SceneNode, child SceneNode) {
    child.AttachParent(logicalParent)
    n.children = append(n.children, child)
}

func (n *Node) AddChildren(child SceneNode) {
    n.addChildWithParent(n, child)
}
```

DetachChild

```
// File: ch02/internal/core/node.go
func (n *Node) DetachChild(node SceneNode) bool {
    for i, c := range n.children {
        if c == node {
            // Swap with last and shrink
            n.children[i] = n.children[len(n.children)-1]
            n.children = n.children[:len(n.children)-1]
            node.AttachParent(nil)  // Clear the child's parent
            return true
        }
    }
    return false
}
```

We call `AttachParent(nil)` on the removed child so both links are updated. The parent no longer lists the child; the child no longer points to the parent.

AttachParent and GetChildren

```
// File: ch02/internal/core/node.go
func (n *Node) AttachParent(node SceneNode) { n.parent = node }
func (n *Node) GetChildren() []SceneNode   { return n.children }
```

The remaining methods manage the parent link and child list:

- **`AttachParent(node SceneNode)`** – Sets the parent reference. You rarely call this directly; `AddChildren` and `DetachChild` call it on the child. Both the parent's `children` slice and the child's `parent` field must stay in sync.
- **`GetChildren()`** – Returns the slice of direct children. The World traverses the scene graph by calling `GetChildren()` on each node and recursing. The slice is the actual backing storage; modifications to it affect the scene graph structure.

MarkDirty

```
// File: ch02/internal/core/node.go
func (n *Node) MarkDirty() {
    for _, c := range n.children {
        c.MarkDirty()
    }
}
```

When a node's structure or data changes, we may need to signal that cached data (e.g. world transforms) is stale. `MarkDirty` propagates to all children. For a plain `Node`, this has no immediate effect (it has no transform cache). For `Node2D` and `Sprite`, they override `MarkDirty` to set `isDirty = true` and then call the parent implementation, so the dirty flag propagates down the scene graph.

Transform and Transformable

```go
// File: ch02/internal/core/transform.go
type Transform struct {
    position Vector2D
    pivot    Vector2D
    rotation float64
    scale    Vector2D
}
```

A **Transform** holds position, pivot, rotation (radians), and scale. It is the data that `Node2D` uses for its local and world transforms.

- **`position`** – Where the node is placed in its parent's coordinate system. For a sprite at (100, 50), the sprite's origin (or pivot) will appear 100 pixels right and 50 down from the parent's origin. This is the translation that moves the node.
- **`pivot`** – The point around which scaling and rotation happen. For a sprite, (0, 0) is typically the top-left corner; if you set the pivot to the centre of the texture, rotating scales around that centre. The transform pipeline moves the pivot to the origin, applies scale and rotation, then moves it back.
- **`rotation`** – The angle in **radians** (0 = no rotation, $\pi/2$ = 90° clockwise in a Y-down coordinate system). The node is rotated around its pivot.
- **`scale`** – Stretch or shrink along X and Y. (1, 1) means no scaling; (2, 2) doubles the size; (0.5, 1) halves the width. Scale is applied around the pivot.

NewTransform

```go
// File: ch02/internal/core/transform.go
func NewTransform(position, pivot Vector2D, rotation float64) Transform {
    return Transform{
        position: position,
        pivot:    pivot,
        rotation: rotation,
        scale:    NewVector2D(1, 1),
    }
}
```

`NewTransform` creates a new `Transform` with the given position, pivot, and rotation. The `scale` is always initialized to `(1, 1)`—no scaling by default. You

pass in the initial position (e.g. `ZeroVector2D()` for origin), the pivot point (e.g. centre of a sprite), and the rotation in radians (typically 0). Use this when constructing a `Node2D`'s local or world transform.

Transform getters and setters

```
// File: ch02/internal/core/transform.go
func (t *Transform) GetPosition() Vector2D      { return t.position }
func (t *Transform) SetPosition(v Vector2D)     { t.position.SetPosition(v) }
func (t *Transform) GetPivot() Vector2D        { return t.pivot }
func (t *Transform) SetPivot(x, y float64)     {
↪  t.pivot.SetPosition(NewVector2D(x, y)) }
func (t *Transform) GetRotation() float64      { return t.rotation }
func (t *Transform) SetRotation(r float64)     { t.rotation = r }
func (t *Transform) GetScale() Vector2D        { return t.scale }
func (t *Transform) SetScale(x, y float64)     {
↪  t.scale.SetPosition(NewVector2D(x, y)) }
func (t *Transform) Translate(x, y float64)   { t.position.x += x; t.position.y
↪  += y }
func (t *Transform) Rotate(r float64)          { t.rotation += r }
```

The getters and setters read or write these fields. `Translate(x, y)` adds to the position; `Rotate(r)` adds to the rotation.

```
// File: ch02/internal/core/transform.go
func (t *Transform) Concat(other Transform) {
    sx := other.position.X() * t.scale.X()
    sy := other.position.Y() * t.scale.Y()
    rotated := NewVector2D(sx, sy).RotateVector(t.rotation)
    t.Translate(rotated.X(), rotated.Y())
    t.scale.SetPosition(NewVector2D(t.scale.X()*other.scale.X(),
    ↪  t.scale.Y()*other.scale.Y()))
    t.Rotate(other.rotation)
    t.pivot.SetPosition(other.GetPivot())
}
```

Concat combines two transforms: the receiver `t` (typically the parent's **world** transform) and the argument `other` (the child's **local** transform). The result is `t × other` in the sense used by this codebase: the child's position is scaled by the parent's scale, rotated by the parent's rotation, then added to the parent's translation; parent and child scales multiply; rotations add; the child's pivot replaces the combined pivot because pivot lives in the drawable node's local (texture) space. When you later reason about matrices, remember that

each `Translate` / `Scale` / `Rotate` on `ebiten.GeoM` composes in the engine's fixed order too—Chapter 1 already showed why order matters.

Transformable

```
// File: ch02/internal/core/transformable.go
type Transformable interface {
    GetTransform() Transform
    SetTransform(t Transform)
    GetWorldTransform() Transform
}
```

Transformable is the interface for any node that has a transform. The World and draw logic use it to obtain the world transform. `Node2D` implements it; a plain `Node` does not.

Node2D: Nodes with Transforms

`Node2D` embeds `Node` and adds **spatial transform** data: position, rotation, scale, and pivot. It is the base for everything that has a place in 2D space: sprites, cameras, collision shapes, and so on.

The Node2D Struct

```
// File: ch02/internal/core/node2d.go
type Node2D struct {
    Node
    localTransform  Transform   // Position, rotation, scale relative to parent
    worldTransform  Transform   // Cached result: combined transform from root
    ↪  to here
    isDirty         bool        // True when local changed; world must be
    ↪  recomputed
}
```

- **`Node`** – An embedded struct: `Node2D` *contains* a full `Node` as its first field. The `Node` holds `id`, `name`, `children`, and `parent`—everything needed to participate in the scene graph hierarchy. By embedding it, `Node2D` gets the scene graph structure (parent-child links, traversal) without reimplementing it. We explain *how* embedding works and what promotion means in the subsection below.

- **`localTransform`** – The node's position, rotation, scale, and pivot *relative to its parent*. When you call `SetPosition(100, 50)`, you write to this field. It answers: "where am I in my parent's coordinate system?"
- **`worldTransform`** – The *combined* transform from the root of the scene graph down to this node. It answers: "where am I on screen?" We compute it by concatenating the parent's world transform with our local transform. It is **cached**: we recompute only when the local transform (or any ancestor's) changes.
- **`isDirty`** – A boolean flag: when `true`, the cached `worldTransform` is stale and must be recomputed. Set by `MarkDirty()` whenever we change position, rotation, scale, or pivot. Also propagated to children, since their world transform depends on ours.

Struct Embedding in Go

Go has no inheritance (no `extends` or subtyping). Instead, **struct embedding** provides composition with automatic **promotion** of fields and methods.

Syntax: `Node` is written as a field with *only the type*, no name. This makes it an **embedded** struct. The type name becomes the implicit field name: you can still access the embedded value as `n.Node`.

Memory layout: `Node2D` contains a `Node` as its first field, followed by `localTransform`, `worldTransform`, and `isDirty`. The embedded struct is a real field—it occupies memory inside `Node2D`, and it must be initialized when you construct it.

Field promotion: The embedded type's fields are **promoted** to the outer type. You can write `n.id`, `n.name`, `n.children`, and `n.parent` on a `*Node2D`—the compiler rewrites these to `n.Node.id`, `n.Node.name`, etc. If `Node2D` had its own field named `id`, that would shadow the embedded one; you would then use `n.Node.id` to access the embedded field explicitly.

Method promotion: Methods defined on `*Node` (e.g. `GetParent`, `AddChildren`, `GetChildren`) are promoted to `*Node2D`. When you call `n.GetParent()` on a `*Node2D`, the receiver passed to `Node.GetParent` is the *embedded* `*Node`—i.e. the address of `n.Node`, not the whole `Node2D`. So `n.GetParent()` behaves like `(&n.Node).GetParent()`. If `Node2D` defines its own `GetParent` with receiver `*Node2D`, that **overrides** the promoted method; the outer type's methods take precedence.

Interface satisfaction: Go uses structural typing for interfaces: a type implements an interface if it has the required methods. `*Node` implements `SceneNode`. Because `*Node2D` gets all of `Node`'s methods through promotion, `*Node2D` also implements `SceneNode`—no forwarding code needed. You can pass a `*Node2D` wherever a `SceneNode` is expected.

Initialization: When constructing a `Node2D`, you must initialize the embedded `Node` explicitly: `Node: *NewNode(name)`. The embedded struct is not created automatically.

NewNode2D

```
// File: ch02/internal/core/node2d.go
func NewNode2D(name string) *Node2D {
    n := &Node2D{
        Node:           *NewNode(name),
        localTransform:  NewTransform(ZeroVector2D(), ZeroVector2D(), 0),
        worldTransform:  NewTransform(ZeroVector2D(), ZeroVector2D(), 0),
        isDirty:         true,
    }
    return n
}
```

Creates a new `Node2D` with an identity local transform (position 0,0, rotation 0, scale 1) and identity world transform. `isDirty` starts as `true` so the first `GetWorldTransform()` call computes the world from the parent.

AddChildren Override

`Node2D` overrides `AddChildren` so that the stored parent is the `Node2D` (which implements `Transformable`), not the embedded `Node`:

```
// File: ch02/internal/core/node2d.go
// AddChildren overrides Node.AddChildren so the stored parent is the Node2D
↪  (Transformable),
// not the embedded Node. Delegates to Node.addChildWithParent to avoid code
↪  duplication.
func (n *Node2D) AddChildren(child SceneNode) {
    n.Node.addChildWithParent(n, child)
}
```

Without this override, when you call `logo.AddChildren(sprite)` on a `*Node2D`, the promoted `Node.AddChildren` would pass the embedded `*Node` as the parent. The sprite's `GetWorldTransform()` would then call `parent.(Transformable)`, which fails for `*Node`—so the sprite would not inherit the logo's position and rotation. By delegating to `addChildWithParent(n, child)`, we ensure the child's parent is the full `Node2D`, and world transform inheritance works correctly.

Transform Accessors

```
// File: ch02/internal/core/node2d.go
func (n *Node2D) GetTransform() Transform          { return n.localTransform }
func (n *Node2D) SetTransform(t Transform)        { n.localTransform = t;
↪  n.MarkDirty() }
func (n *Node2D) SetPosition(x, y float64)        {
↪  n.localTransform.SetPosition(NewVector2D(x, y)); n.MarkDirty() }
func (n *Node2D) GetPosition() Vector2D            { return
↪  n.localTransform.GetPosition() }
func (n *Node2D) SetRotation(r float64)            {
↪  n.localTransform.SetRotation(r); n.MarkDirty() }
func (n *Node2D) GetRotation() float64             { return
↪  n.localTransform.GetRotation() }
func (n *Node2D) SetScale(x, y float64)            {
↪  n.localTransform.SetScale(x, y); n.MarkDirty() }
func (n *Node2D) GetScale() Vector2D              { return
↪  n.localTransform.GetScale() }
func (n *Node2D) GetPivot() Vector2D              { return
↪  n.localTransform.GetPivot() }
func (n *Node2D) SetPivot(x, y float64)            {
↪  n.localTransform.SetPivot(x, y); n.MarkDirty() }
```

All setters call `MarkDirty()` because changing the local transform invalidates the cached world transform. Getters simply read from `localTransform`. These satisfy the `Transformable` interface required for drawing.

MarkDirty

```go
// File: ch02/internal/core/node2d.go
func (n *Node2D) MarkDirty() {
    if n.isDirty {
        return
    }
    n.isDirty = true
    n.Node.MarkDirty()
}
```

When does a node become dirty? — A node is marked dirty when its cached world transform is no longer valid. This happens when: (1) we change *this* node's local data (`SetPosition`, `SetRotation`, `SetScale`, `SetPivot`, or `SetTransform`); (2) our *parent* calls `MarkDirty()` and propagates to us (via `Node.MarkDirty()`); (3) at creation—`NewNode2D` sets `isDirty = true` because we have no valid cache yet. In short: we are dirty whenever our own transform changed or any ancestor's did (since our world transform is `parent_world × local`).

Why the early return? — `if n.isDirty { return }` avoids redundant work. We can receive `MarkDirty()` multiple times before the next `GetWorldTransform()`: e.g. `SetPosition(10, 20)` then `SetRotation(0.5)` in the same frame—each calls `MarkDirty()`. Or a parent may already have propagated: when the root is marked dirty, it recurses to all descendants; if something later marks an intermediate node dirty again, we must not re-walk its subtree. The early return ensures each node sets `isDirty = true` at most once per "dirty wave" and propagates to children at most once.

Why track `isDirty` at all? — `GetWorldTransform()` recomputes only when dirty. If nothing changed, we return the cached `worldTransform` immediately. Without this flag, every `GetWorldTransform()` would walk up the parent chain and concatenate transforms—expensive when hundreds of sprites are drawn each frame. The dirty flag is a common pattern for lazy, cached computation.

GetWorldTransform: Combining Parent and Local

```go
// File: ch02/internal/core/node2d.go
func (n *Node2D) GetWorldTransform() Transform {
    if !n.isDirty {
        return n.worldTransform  // Use cache
    }

    world := NewTransform(ZeroVector2D(), ZeroVector2D(), 0)
    if parent := n.GetParent(); parent != nil {
        if pt, ok := parent.(Transformable); ok {
            world = pt.GetWorldTransform()  // Start from parent's world
        }
    }
    world.Concat(n.localTransform)   // Apply our local on top
    n.worldTransform = world
    n.isDirty = false
    return n.worldTransform
}
```

It returns the world transform—the combined position, rotation, and scale from the root of the scene graph down to this node. If the cache is valid (`!n.isDirty`), it returns `worldTransform` immediately. Otherwise it recomputes: it starts from the parent's world transform (or identity if the parent has no transform, e.g. the root), concatenates our local transform on top, stores the result in `worldTransform`, clears the dirty flag, and returns it. This is what the World uses when drawing: it needs the final transform on screen, not the local one.

In Go, the two-result form `value, ok := x.(SomeType)` is a **type assertion**. It checks whether the concrete value stored in the interface variable `x` has type `SomeType` (or implements `SomeType`, if it is an interface). You obtain two values: (1) `value` – the same value seen as type `SomeType`, usable only when the assertion succeeds; (2) `ok` – a boolean: `true` if the assertion succeeded, `false` otherwise. When `ok` is `false`, `value` is the zero value of `SomeType` and must not be used. This form never panics.

In this specific case: The relevant snippet is:

```
// File: ch02/internal/core/node2d.go
world := NewTransform(ZeroVector2D(), ZeroVector2D(), 0)
if parent := n.GetParent(); parent != nil {
    if pt, ok := parent.(Transformable); ok {
        world = pt.GetWorldTransform()
    }
}
world.Concat(n.localTransform)
```

In the snippet, the receiver `n` is a `*Node2D`—the node whose world transform we are computing. We call `n.GetParent()`, which returns a `SceneNode`. At this point we only know the parent through the `SceneNode` interface: it has `GetChildren`, `AddChildren`, etc., but we do *not* know its concrete type. The parent could be a `*Node` (e.g. the root of the scene graph, a plain container with no position or rotation), a `*Node2D`, or a `*Sprite`. Each has a different concrete type, but all implement `SceneNode`.

Because we work with interfaces, the code does not depend on the concrete type. We never write "if the parent is a Node2D, do X; if it is a Node, do Y". We write: "if the parent implements `Transformable`, use its world transform". Whether the parent is a `Node`, `Node2D`, or `Sprite` is irrelevant; what matters is whether it has a world transform. A `Node` does not implement `Transformable`—it has no position, scale, or rotation. A `Node2D` and a `Sprite` (which embeds `Node2D`) do implement it. The type assertion discovers this at runtime.

So: `parent` holds a concrete value (a `*Node`, `*Node2D`, or `*Sprite`) wrapped in the `SceneNode` interface. The assertion `parent.(Transformable)` asks: does that concrete value implement `Transformable`? We get `parentTransformable` (the `Transformable` view) and `ok`. When `ok` is `true`, we call `GetWorldTransform()` and use it as the starting transform. When `ok` is false (parent is a plain `Node`, typically the root), the block is skipped and `world` stays as the identity transform. We then concatenate this node's local transform onto `world`. A `Node2D` directly under the root ends up with world = local, which is correct.

The call `world.Concat(local)` in this function is the same `Transform.Concat` you saw earlier: parent world on the left, child local on the right, with the pivot semantics described in the `Transform` section.

The Drawable Interface

```go
// File: ch02/internal/core/drawable.go
type Drawable interface {
    Transformable
    GetLayer() int
    Draw(target *ebiten.Image, op *ebiten.DrawImageOptions)
}
```

Drawable is the interface for any node that can be rendered on screen. It embeds `Transformable`: anything drawable must also know its world position, rotation, and scale—otherwise the World could not build a `GeoM` for it. In Go, embedding an interface inside another interface **requires** implementors to satisfy both: `Drawable` means "`Transformable` **and** `GetLayer() int` **and** `Draw(...)`". Callers that accept a `Drawable` can use every promoted method without a second type assertion.

- **Transformable** – Provides `GetTransform()` and `GetWorldTransform()` so the World can build the `GeoM` and know where to draw.
- **GetLayer()** – Returns an integer **layer id** for draw ordering. Lower values mean "further back" when you eventually sort drawables globally; higher values mean closer to the viewer.
- **Draw(target, op)** – Renders the node onto the target image. The `op` already contains the correct `GeoM` computed by the World from the node's world transform.

> **Note:** In Chapter 2, `World.drawNode` does **not** yet sort by `GetLayer()`. Draw order follows **depth-first traversal**: it visits the root, recurses into children in `GetChildren()` order, and draws each `Drawable` when that node is visited. Parent nodes are drawn before their own subtree continues, so a parent `Drawable` appears underneath its descendants if both draw opaque pixels. `GetLayer()` is part of the interface now so `Sprite` and future types stay consistent; the next chapter uses layers and the camera to refine ordering and screen space.

The Sprite: A Drawable Node2D

A **Sprite** is a `Node2D` that holds a texture and knows how to draw itself. It implements **Drawable**.

```
// File: ch02/internal/core/sprite.go
type Sprite struct {
    Node2D
    texture *ebiten.Image
    layer   int
    visible bool
}
```

The struct embeds `Node2D`, so a Sprite has position, rotation, scale, and hierarchy like any Node2D. In addition:

- **`texture`** – The image to draw (e.g. decoded from embedded PNG bytes or loaded from disk). Can be `nil` for a placeholder sprite; `Draw` will skip drawing.
- **`layer`** – Integer carried on the sprite for future global ordering; see the `Drawable` note above for Chapter 2 traversal behavior.
- **`visible`** – When `false`, `Draw` returns immediately. The node stays in the scene graph, so you can toggle HUD elements or paused actors without rebuilding the scene graph.

NewSprite

```
// File: ch02/internal/core/sprite.go
func NewSprite(name string, texture *ebiten.Image, layer int, centerPivot bool)
↪  *Sprite {
    s := &Sprite{
        Node2D:  *NewNode2D(name),
        texture: texture,
        layer:   layer,
        visible: true,
    }
    if texture != nil && centerPivot {
        s.SetPivotToCenter()
    }
    return s
}
```

The constructor builds a Sprite with the given name and layer. The last argument, `centerPivot`, controls whether to call `SetPivotToCenter()` when a texture is provided. If `true`, the sprite rotates around its center instead of the top-left corner; if `false`, the pivot stays at the default (top-left). When `texture` is nil, `centerPivot` has no effect.

Getters and Setters

```
// File: ch02/internal/core/sprite.go
func (s *Sprite) GetTexture() *ebiten.Image   { return s.texture }
func (s *Sprite) SetTexture(tex *ebiten.Image) {
    s.texture = tex
}
func (s *Sprite) GetLayer() int       { return s.layer }
func (s *Sprite) SetLayer(l int)     { s.layer = l }
func (s *Sprite) GetVisible() bool    { return s.visible }
func (s *Sprite) SetVisible(v bool)  { s.visible = v }
```

- **`GetTexture` / `SetTexture`** – Read or replace the source image. Swapping textures does not automatically re-center the pivot; call `SetPivotToCenter()` again if you rely on centred rotation.
- **`GetLayer` / `SetLayer`** – Read or change the layer id (implements `Drawable`). Sorting by layer arrives in Chapter 3.
- **`GetVisible` / `SetVisible`** – Read or toggle visibility. A hidden sprite still exists in the scene graph but is not drawn.

SetPivotToCenter

```
// File: ch02/internal/core/sprite.go
func (s *Sprite) SetPivotToCenter() {
    w := float64(s.texture.Bounds().Dx())
    h := float64(s.texture.Bounds().Dy())
    s.SetPivot(w/2, h/2)
}
```

Sets the pivot to the center of the texture (half width, half height). Rotation and scaling then use the sprite center as origin instead of the top-left corner. Called by `NewSprite` when `centerPivot` is true and a texture is provided.

Draw

`Sprite.Draw` is small on purpose: the World already folded the node's **world** transform into `op.GeoM`. The sprite only checks visibility and texture, then issues one `DrawImage`.

```
// File: ch02/internal/core/sprite.go
func (s *Sprite) Draw(target *ebiten.Image, op *ebiten.DrawImageOptions) {
    if s.texture == nil || !s.visible {
        return
    }
    target.DrawImage(s.texture, op)
}
```

Why split responsibilities? — Keeping matrix math in `buildGeoMFromTransform` and `GetWorldTransform()` means every `Drawable` can share the same pipeline. Later, you can add tinting, blend mode, or shader uniforms by extending `DrawImageOptions` in one place, while individual sprites stay focused on **which** texture to blit.

The **World** still owns traversal: `World.Draw` walks from `rootScene`, type-asserts each node to `Drawable`, builds `op`, and calls `Draw` (see How the World Traverses the Scene). The `parentGeoM` parameter on `drawNode` exists so a future optimization could multiply matrices while walking down the scene graph; Chapter 2 leaves it unused because every drawable already requests a full world transform.

buildGeoMFromTransform

```
// File: ch02/internal/core/world.go
func buildGeoMFromTransform(t Transform) ebiten.GeoM {
    g := ebiten.GeoM{}
    pivot := t.GetPivot()
    pos := t.GetPosition()
    scale := t.GetScale()
    rot := t.GetRotation()

    // Move pivot to origin -> Scale -> Rotate -> Place at world position
    g.Translate(-pivot.X(), -pivot.Y())
    g.Scale(scale.X(), scale.Y())
    g.Rotate(rot)
    g.Translate(pos.X(), pos.Y())
    return g
}
```

This function converts our engine-level `Transform` into an `ebiten.GeoM`, which is Ebitengine's 2D transform matrix type. The important detail is how this chapter interprets `position`: it is the **world-space location of the pivot**, not the top-left corner of the image. So we first move the texture so the pivot sits at the origin, then apply scale and rotation around that origin, and finally place that origin at `position`. That is why the code does **not** translate by the pivot again after rotation.

The Engine: Delegating to the World

`Engine` is the thin object your `Game` talks to in `Update`, `Draw`, and `Layout`. It owns exactly one `World` and forwards calls—no scene logic lives here yet. That keeps `ebiten.Game` wiring stable as the book grows: later you can add timing, profiling, or multiple worlds without rewriting the game struct.

```
// File: ch02/internal/core/engine.go
type Engine struct {
    world *World
}

func NewEngine() *Engine {
    return &Engine{world: NewWorld()}
}

func (e *Engine) World() *World { return e.world }

func (e *Engine) Update() error {
    e.world.Update()
    return nil
}

func (e *Engine) Draw(target *ebiten.Image) {
    e.world.Draw(target)
}

func (e *Engine) Layout(outsideWidth, outsideHeight int) (int, int) {
    return 640, 480
}
```

`Layout` returns the **logical** resolution Ebitengine uses for your game canvas (640×480 here, matching `GameSettings` in `run.go`). If you change the window size in settings, update `Layout` as well—or refactor later so both read from a single exported settings struct on the engine.

How the World Traverses the Scene

`NewWorld` installs an anonymous root `*Node` named `"root"`. `AddNodeToDefaultLayer` simply calls `AddChildren` on that root so everything you build hangs under one stable parent. `RemoveNode` walks up to the parent, calls `DetachChild`, and clears the child's parent link—use it when despawning nodes so parent pointers do not dangle.

```
// File: ch02/internal/core/world.go (excerpt)
func NewWorld() *World {
    return &World{rootScene: NewNode("root")}
}

func (w *World) AddNodeToDefaultLayer(node SceneNode) {
    w.rootScene.AddChildren(node)
}
```

The World holds that root and recursively visits the hierarchy each frame:

```
// File: ch02/internal/core/world.go
func (w *World) Update() {
    w.updateNode(w.rootScene)
}

func (w *World) updateNode(node SceneNode) {
    if node == nil {
        return
    }
    for _, child := range node.GetChildren() {
        w.updateNode(child)
    }
}

func (w *World) Draw(target *ebiten.Image) {
    w.drawNode(w.rootScene, ebiten.GeoM{}, target)
}

func (w *World) drawNode(node SceneNode, parentGeoM ebiten.GeoM, target
↪  *ebiten.Image) {
    if node == nil {
        return
    }
    if drawable, ok := node.(Drawable); ok {
        op := &ebiten.DrawImageOptions{}
        op.GeoM = buildGeoMFromTransform(drawable.GetWorldTransform())
```

```
26          drawable.Draw(target, op)
27      }
28      for _, child := range node.GetChildren() {
29          w.drawNode(child, ebiten.GeoM{}, target)
30      }
31  }
```

1. **Update**: `updateNode` walks depth-first. This chapter leaves a comment placeholder for per-node gameplay hooks; the walk still establishes a predictable visit order if you add timers or components later.
2. **Draw**: `drawNode` performs the same depth-first walk. Whenever a node satisfies `Drawable`, the World builds fresh `DrawImageOptions`, fills `GeoM` from `GetWorldTransform()`, and calls `Draw`.
3. **Children**: recursion uses `GetChildren()`. Because each child's world transform already folded in its ancestors, you do not multiply `parentGeoM` in Chapter 2 even though the parameter is reserved.

The key idea is that **hierarchy still drives rendering**. The World does not pass an accumulated matrix downward in this version; instead, `GetWorldTransform()` walks upward through the parent chain when needed. From the point of view of the rendered result, the effect is the same: child placement stays relative to the parent.

`RemoveNode` (not used in the rotating-logo demo, but ready for gameplay) detaches a node from whichever parent currently owns it:

```
1   // File: ch02/internal/core/world.go
2   func (w *World) RemoveNode(node SceneNode) bool {
3       parent := node.GetParent()
4       if parent == nil {
5           return false
6       }
7       if !parent.DetachChild(node) {
8           return false
9       }
10      node.AttachParent(nil)
11      return true
12  }
```

> **Tip:** Prefer `RemoveNode` over manually splicing slices so `DetachChild` and `AttachParent(nil)` stay paired—otherwise the scene graph can end up with children listed under a parent while the child's `GetParent()` still points elsewhere.

Complete Example: Rotating Logo

We put the framework to work by displaying the Go Gopher logo as a `Sprite` and animating it with a continuous rotation. This example runs through the full flow: loading the image, creating the engine and scene graph, adding the sprite, and updating its rotation each frame.

Setup: Run, main, and the Game Struct

In the current codebase, the executable entrypoint is intentionally small. `cmd/main.go` only calls `game.Run()`, while `run.go` loads the embedded asset, builds the engine and scene, and starts Ebitengine. This separation keeps framework bootstrapping in the chapter package and leaves the binary wrapper minimal.

```go
// File: ch02/cmd/main.go
func main() {
    if err := game.Run(); err != nil {
        log.Fatal(err)
    }
}
```

```go
// File: ch02/run.go
func Run() error {
    decoded, _, err := image.Decode(bytes.NewReader(assets.GopherPNG))
    if err != nil {
        return fmt.Errorf("failed to decode embedded gopher image: %w", err)
    }
    img := ebiten.NewImageFromImage(decoded)

    engine := NewEngine()
    world := engine.World()

    logo := NewNode2D("logo")
    logo.SetPosition(320, 240)
    sprite := NewSprite("gopher", img, 0, true)
    sprite.SetPosition(0, 0)
    logo.AddChildren(sprite)
    world.AddNodeToDefaultLayer(logo)

    ebiten.SetWindowSize(GameSettings.ScreenWidth, GameSettings.ScreenHeight)
    ebiten.SetWindowTitle("Chapter 2: Scene Graph Framework")
```

```
21
22      game := &Game{engine: engine, logo: logo}
23      return ebiten.RunGame(game)
24  }
```

```
1   // File: ch02/game.go
2   type Game struct {
3       engine *Engine
4       logo   *Node2D
5   }
```

- **Embedded asset loading**: this chapter uses `//go:embed` from `ch02/assets/embed.go`, then decodes the PNG bytes into an `*ebiten.Image`. This keeps the example self-contained and avoids depending on a relative file path at runtime.
- **Create the engine and world** with `NewEngine()` and `engine.World()`. The engine owns the world and exposes it to the game setup code.
- **Create a `Node2D` container** called `logo` at `(320, 240)`, the center of the default logical screen. This node holds the transform for the whole subtree.
- **Create the sprite** with `NewSprite("gopher", img, 0, true)`. Layer `0` is enough for this chapter, and `centerPivot = true` means rotation happens around the texture centre, not the top-left corner.
- **Attach the sprite to the logo** with `logo.AddChildren(sprite)`. Because the sprite's local position is `(0, 0)`, its pivot is placed exactly at the logo node's position.
- **Add the logo to the world** with `world.AddNodeToDefaultLayer(logo)`. From that point on, traversal, world-transform calculation, and drawing all flow from the root of the scene graph.
- **Use `GameSettings` for the window size**. Even in a tiny example, having a central settings value makes later expansion easier.

Update: Rotating the Logo Each Frame

Each frame, Ebitengine calls `Update`. We use it to increment the logo's rotation. Because the sprite is a child of the logo, rotating the logo rotates the entire subtree—the sprite spins with it:

```go
// File: ch02/game.go
const rotationSpeed = 0.02  // Radians per frame at 60 TPS

func (g *Game) Update() error {
    g.logo.SetRotation(g.logo.GetRotation() + rotationSpeed)
    if g.logo.GetRotation() >= 2*math.Pi {
        g.logo.SetRotation(0)
    }
    return g.engine.Update()
}
```

- **`GetRotation()`** returns the current local rotation in radians. The logo (Node2D) stores this in `localTransform`.
- **`SetRotation(r)`** sets the new angle and calls `MarkDirty()`, so the next time the World asks for the world transform of the logo (and its descendants), it will be recomputed with the new rotation.
- **`rotationSpeed`** is 0.02 radians per frame. At 60 TPS, one full rotation ($2\pi \approx 6.28$ radians) takes about $6.28 / 0.02 \approx 314$ frames, i.e. about five seconds.
- **Angle wrapping**: When the angle reaches or exceeds 2π, we reset it to 0. This avoids the angle growing without bound over long sessions, which could cause precision issues. It also keeps the rotation visually seamless.

Order of operations: We update the logo first, then call `g.engine.Update()`. The engine updates the world (which traverses the scene; for now it does not run per-node logic). When `Draw` runs, the logo's world transform includes the new rotation, and the sprite (as a child) inherits it—so the sprite appears rotated on screen.

Draw and Layout

```go
// File: ch02/game.go
func (g *Game) Draw(screen *ebiten.Image) {
    g.engine.Draw(screen)
}

func (g *Game) Layout(outsideWidth, outsideHeight int) (int, int) {
    return g.engine.Layout(outsideWidth, outsideHeight)
}
```

- **Draw** delegates to the engine. The engine calls `world.Draw(screen)`. The World traverses the scene graph, finds our sprite (a Drawable), gets its world transform via `GetWorldTransform()`, builds the `GeoM`, and calls `sprite.Draw(screen, op)`. The sprite draws its texture with the correct position and rotation—no manual matrix math on our side.
- **Layout** forwards to `Engine.Layout`, which returns `640×480` in this chapter—the same logical size passed to `ebiten.SetWindowSize` via `GameSettings`. Keeping those two in sync matters: `Layout` defines your virtual canvas; the window is how Ebitengine scales that canvas to pixels.

How the Rotation Flows Through the Framework

1. **Update**: We call `logo.SetRotation(angle + 0.02)`. The logo (Node2D) stores the new angle in its `localTransform` and sets `isDirty = true`.
2. **Draw**: The World traverses the scene graph: root □ logo □ sprite. For the sprite (a Drawable), it calls `sprite.GetWorldTransform()`.
3. **GetWorldTransform**: The sprite's parent is the logo (a Node2D). Because `Node2D.AddChildren` stored the `Node2D` as parent (not the embedded Node), `parent.(Transformable)` succeeds. The sprite gets the logo's world transform and concatenates its own local (position 0,0, no rotation). The result is the logo's position and rotation combined with the sprite's local—so the sprite appears at (320, 240) with the logo's rotation.
4. **buildGeoMFromTransform**: The World converts the transform to an `ebiten.GeoM`: translate by `-pivot`, scale, rotate, then place the pivot at the final world position. The rotation comes from the logo's world transform.
5. **Draw**: The sprite's `Draw` method receives the `op` with this GeoM and draws the texture. The image appears rotated on screen.

The pivot is at the center of the texture. When we rotate the logo, the sprite (at local 0,0) orbits around the logo's position; with the sprite's pivot at center, it spins in place—exactly what we want for a logo.

Summary of the Example

Step	What happens
cmd/main + Run	Call `game.Run()`, decode embedded PNG, create engine/world, build logo subtree, start Ebitengine
Update	Increment logo rotation, wrap at 2π, call engine.Update()
Engine.Update	Delegates to world.Update()
World.Update	Traverses scene (no Updatable nodes yet)
Draw	Engine draws to screen
World.Draw	Traverses scene, for each Drawable calls GetWorldTransform(), builds GeoM, calls Draw
Sprite.Draw	Draws texture with op.GeoM (inherits logo's world transform)

Run the example with `cd ch02 && go run ./cmd`. You will see the Go Gopher logo rotating in the center of the window.

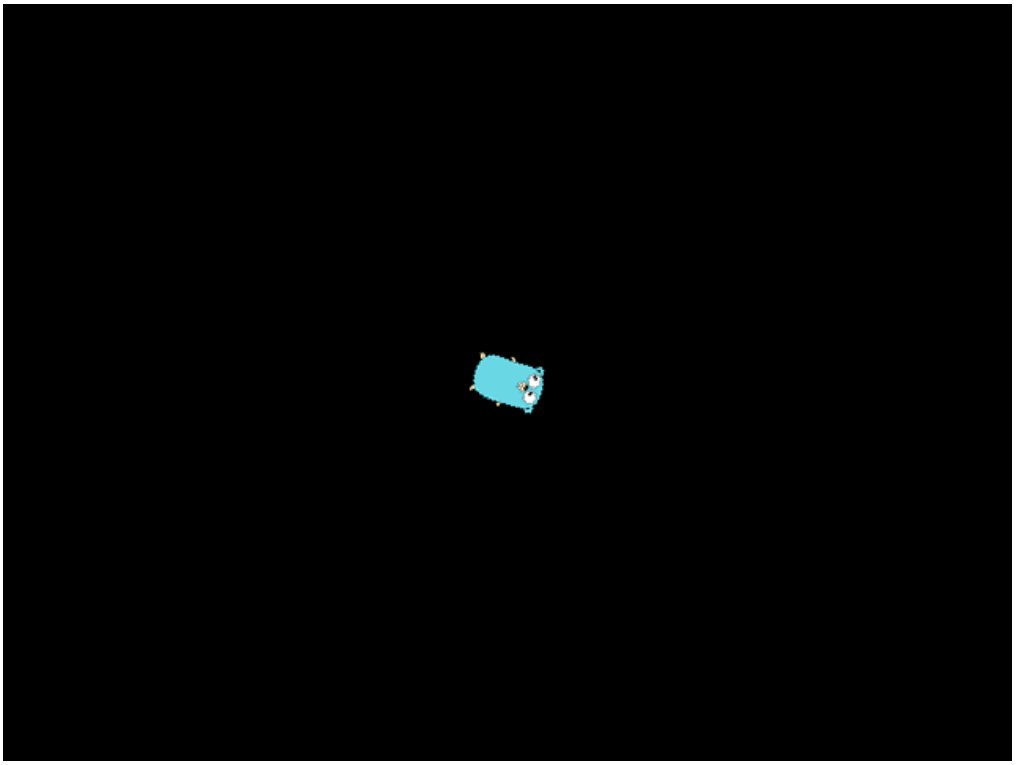


Figure 4. Chapter 2 result: the Go Gopher logo rotating about its center

Summary

Component	Role
SceneNode	Interface for any node in the scene graph; defines parent-child operations.
Node	Base implementation: ID, name, parent, children. No spatial data.
Node2D	Extends Node; adds local/world transform, dirty flag, `GetWorldTransform()`.
Sprite	Node2D + texture; implements Drawable; draws with accumulated transform.
Engine	Top-level hub; owns `World`; delegates `Update` / `Draw`; `Layout` returns logical size (640×480 here, keep aligned with `GameSettings`).

Component	Role
World	Owns a `"root"` node; `AddNodeToDefaultLayer` / `RemoveNode`; traverses for Update and Draw; converts world transforms to `ebiten.GeoM`.

Relative transforms: A node's position is in its parent's space. The world transform is `parent_world × local`. Children move with their parent. The dirty flag avoids redundant recomputation.

Code Structure for Chapter 2

The framework code lives in `ch02/`. Structure:

```
ch02/
|- go.mod
|- run.go
|- game.go
|- cmd/
|  `- main.go
|- assets/
|  |- embed.go
|  `- golang_icon.png
`- internal/
   `- core/
      |- asset_path.go
      |- drawable.go
      |- engine.go
      |- node.go
      |- node2d.go
      |- scenenode.go
      |- settings.go
      |- sprite.go
      |- transform.go
      |- transformable.go
      |- vector2d.go
      `- world.go
```

You introduced this layout when you created the first file at `internal/core/vector2d.go` (see Where to put framework code (internal and core packages)). The tree above lists every file for quick reference.

Run the example:

```
# Run from: ch02
cd ch02
go run ./cmd
```

You will see the Go Gopher logo rotating in the center of the window.

In the next chapter, we will add a **Resource Manager** for textures and a **Layer** system for draw order, attaching a floor and the player sprite on separate layers. The **Camera** that scrolls the world in screen space arrives in Chapter 5.

* * *

Get the code: the complete, runnable project for this chapter is in the `ch02` directory of the companion GitHub repository.

Chapter 3: Resource Manager, Layers and Sprites

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The Resource Manager and asset loading

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

What changed in `ch03` (before we dive in)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

ResourceManager structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

What `embed.FS` is, and why this project uses it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Constructor and `UseEmbeddedFS`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Wiring embedded sprites: `//go:embed`, path constants, and `loadTextures`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Loading and retrieving textures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The Layer Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

What Are Layers?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Implementing Layers: We Need a Stack First

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The Stack — LIFO and Go Generics

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The Layers Implementation — ensureLayer, AddDraw, DrawAll

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The Layers System: Queue Then Draw

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Layer Roots in the World

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Structural layers vs `GetLayer()` on sprites

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The World — queueNode and Draw Flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Recursion and DrawAll: Two Phases

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The Engine — World and ResourceManager

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Settings and Overall Flow

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Load Textures and Create Sprites

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Load Textures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Create Sprites from Textures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Draw Order — Floor First, Player Centered

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Floor Sprite — Full Screen

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Player Node — Centered

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Result

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Code Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Chapter 4: Input and Player Movement

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Overview of the Input System

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

rawinput.go — Physical Input Abstraction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

1.1 Imports and InputButtonType

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

RawInputButton Struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Constructors for RawInputButton

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

IsPressed, IsJustPressed, IsJustReleased — The Temporal Difference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

IsPressed — Current Held State

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

IsJustPressed — First Frame Only

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

IsJustReleased — Release Frame Only

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

PressDuration — Frames Held

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

action.go — Action with Trigger Mode

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

ActionMode — When the Action Fires

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

ActionMode.Has — Flag Check

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

ActionType — Input Device

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Action Struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Action Constructors

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Action Getters

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

actionmap.go — ActionID to Action Mapping

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

ActionID and ActionMap Struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

NewActionMap and AddAction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

3.3 RemoveAction, GetAction, ClearActions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

SetKeyBinding and ErrActionNotFound

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

statebuffer.go — Tracking State Across Frames

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

InputState and StateBuffer Struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

NewStateBuffer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Update — Keyboard Loop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Update — Mouse Loop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

IsActionActive and checkState

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

inputmanager.go — Two Input APIs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Frame flow through `InputManager`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

ActionBinding and InputManager Struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

NewInputManager

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Update

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

AddAction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

RemoveAction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

IsActionPressed, IsActionJustPressed, IsActionJustReleased, IsActionReleased — Comparing the Four Methods

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

IsActionPressed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

IsActionJustPressed, IsActionJustReleased, IsActionReleased

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

RegisterAction and IsActionActive (ActionID API)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

ActionMap Accessor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

engine.go — Adding InputManager and Update

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Engine Struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

NewEngine

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Input Accessor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Update with input.Update()

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Draw and Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Game Setup and Movement

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Game Struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

NewGame — Building the Scene

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

game_init.go — Loading Textures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

game_init.go — Binding Arrow Keys

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

game_init.go — Floor Setup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

player.go — Player Constants and Type

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

player.go — Creating the Player

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

player.go — Reading Movement Input

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

player.go — Normalize and Move

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

game.go — Update, Draw, and Layout

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

run.go and cmd/main.go — Entry Point

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Code Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Chapter 5: Tileset, Tilemap, and Camera

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Tileset - Cutting Tiles from a Spritesheet

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

What is a tile?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

What is a spritesheet?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Identify the floor tiles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Convert a tile cell into pixels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The `Tileset` struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Cutting out one tile with `GetTileRect`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Tilemap - From a Finite Pattern to an Infinite Floor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

A tilemap is a grid of choices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Store the tilemap in `floor.map`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Map text numbers to tileset cells

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Parse the text file into Go data

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Place tilemap cells in world space

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Load the finite pattern during setup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Repeat the finite pattern forever

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Draw only the visible cells

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Camera - The Moving Window over the World

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Why you need a camera

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

World space vs screen space

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The `Camera` struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Make the camera follow the player

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Apply the camera offset

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

World - Orchestrating the Render Pass

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The `World.Draw` sequence

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Prepare the draw surface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Queue drawables with the camera offset

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Chapter 5 Setup - File by File

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

`cmd/main.go` - Program entry point

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

`run.go` - Starting Ebitengine

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

`game.go` - Creating the session

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

`game_init.go` - Loading textures and creating the tilemap

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

`player.go` - Player movement in world space

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

`settings.go` - Shared configuration

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Embedded assets

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Optional Extensions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Smooth camera following

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Multiple tilemap layers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Different patterns

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Code Structure

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Chapter 6: Enemy and Collisions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Understanding collisions in 2D games

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

node2d.go — Add GetWorldPosition

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

collision.go — CollisionCircle and Overlap Test

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Struct and constructor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Overlap test

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

collisionMask.go — Layers and Collision Filtering

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Layer constants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

CollisionMask struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

CanCollideWith and GetIdentity

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

collider.go — Node2D with Collision

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Struct Collider

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

SetOnCollide

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Getters and CanCollideWith

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

collisionManager.go — CheckCollision

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Where the manager lives

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Struct, registration, and NewCollider

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

CollisionManager struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

NewCollisionManager

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

AddCollider

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

NewCollider

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Registration in Player and Enemy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

How many pairs per frame?

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Why `j = i + 1` (no duplicate pairs)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Sampling each collider once per outer step

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Scene graph vs collider slice — two parallel structures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

When to call CheckCollision

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Timeline — the frame contact happens

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

CheckCollision

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

`circleSample` — cached circle geometry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

`sampleCircle` — read centre and radius from a collider

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

`pairWantsCollision` — layer mask filter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

`samplesOverlap` — narrow-phase circle test

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

`notifyPair` — invoke callbacks

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

`checkColliderPair` — one unordered pair end-to-end

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

`CheckCollision` — pair loops only

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Cost per pair (unchanged behaviour)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Chapter 7 note — `combineCollisionIDs`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

What the manager does not do

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

O(n^2) loop and when to optimise

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Debugging CheckCollision

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

spawnEnemyFarFrom — Enemy Spawn for Infinite Map

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Why spawning is different on an infinite map

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Distance choice: 200 pixels

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Struct and constant

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

spawnEnemyFarFrom and NewEnemyFarFromPlayer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

game.go — Setup and Logic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Imports

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Constants

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Paths and layers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Game struct

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

NewGame — Engine, textures, input

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

NewGame — Tileset and tilemap

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

NewGame — Player

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

NewGame — Enemy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

NewGame — Camera and callback

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Update — gameOver and player movement

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Update — enemy AI and collisions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Draw — engine draw and HUD overlay

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

HUD — the GAME OVER overlay

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Running the chapter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

go.mod — Dependencies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Chapter 7: Weapons and Projectiles

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Weapons and projectiles in games

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The cursor: a pointer that follows the mouse

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Screen space versus world space

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Defining the Cursor type

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Updating the cursor each frame

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The timer: a cooldown for the weapon

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Implementing the Timer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Driving the weapon cooldown with the timer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Wiring the weapon and pool into the game

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The projectile: layers and spawn logic

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Spawn position and direction

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Rotating the projectile

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Collision: a projectile kills an enemy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Why the queue must deduplicate

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Walking the queue

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The three removal branches

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Projectile movement and cleanup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The projectile type and its pool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The bloody knife weapon

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Constants and data structures

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Running the chapter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Enemy waves: the spawner and the EnemyManager

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Spawning off-screen

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The enemy node

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Wave timing and escalation

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Chapter 8: UI, Health, XP, and Level Up

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Drawing the UI in screen space

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The reusable progress bar

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The floating message

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Composing the HUD

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Giving the player health, XP, and level

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Adding pickup collision layers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Spawning and collecting XP orbs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Wiring it together in game.go

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Running the chapter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Chapter 9: Health Potion, Sacred Book, and Holy Shield

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

How loot drops on enemy death

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Loading the potion and sacred-book sprites

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Extending the PickupManager

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Modeling the health potion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Dropping loot when an enemy dies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Collecting orbs and potions

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

A common interface for player weapons

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The Sacred Book: an orbiting weapon

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The Holy Shield: a static ring weapon

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Mounting all three weapons

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Keeping XP and the level-up popup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Running the chapter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Chapter 10: Weapon Upgrade UI, Additional Weapons, and a Survival Timer

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Centralizing weapon stats in WeaponManager

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Threading the manager through the weapon interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Reading multipliers inside each weapon

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The Flying Axe: a spinning projectile weapon

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Mounting all four weapons

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The upgrade flow and pausing the world

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

An exponential XP curve

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Modeling upgrades: options, factories, and random sampling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

From options to HUD choices

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The HUD: panels, input, and overlays

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The upgrade panel and stable icons

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Drawing gameplay versus the upgrade screen

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The survival timer and game over

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Restarting from the game-over screen

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Running the chapter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

File layout (highlights)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Full listing pointers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Chapter 11: Weapon Unlock and Gated Upgrades

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

What changes since Chapter 10

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Setting the upgrade caps and weapon IDs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Upgrade caps

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Weapon IDs for the random start

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

New texture keys

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Reusing the infinite floor, input, and movement

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

From mount-all to construct-all, mount-one

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Adding unlock state to the manager

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Constructing the loadout without mounting

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Mount and the new Unregister

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Rolling one random starter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Counting upgrades and scaling difficulty over time

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

checkLevelUp asks the strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Reading the survival clock for difficulty

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Surviving longer: enemy variety and time scaling

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

A stat table for each kind

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Hit points on the enemy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Weapons that subtract damage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Scaling strength faster than reward

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Introducing kinds over time and the Cyclops mini-boss

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Flashing “Enemies Grow Stronger”

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

From flat factories to gated strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

UpgradeOption and upgradeStrategy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The strategy registry

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Building the pool

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Biasing the panel toward bonus items

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Writing the strategies

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Unlocking a weapon

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

A stat upgrade

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Rebuilding the Holy Shield on a radius upgrade

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

One-time bonus items

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Bonus flags on Player

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The boots strategy

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Falling back when the pool is capped

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Splitting display from apply

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Loading the new sprites

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Drawing: unchanged from Chapter 10

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Full listing reference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Chapter 12: Audio and a State Machine for Menus, Gameplay, and Pause

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Sound: looping music

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

How Ebitengine handles audio

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The audio manager

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Loading the sounds

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Where the music plays

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

What is a state machine, and why games use it

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Understanding the state machine

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The four states

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The level-up panel stays inside `Game`

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Decoupling the core from the game package

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The State and StateMachine types

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The State interface

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

IDs, registration, and switching

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Quitting the process

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

App: the single ebiten.Game

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Run and window setup

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Shared menu helpers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The main menu state

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The game state: delegating to Game

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The pause state

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The options state

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The pause action and IsPausePressed

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The main package

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Transition diagram

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

File layout (new and changed)

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Full listing reference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Chapter 13: Game Feel — Particles and Visual Effects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Understanding game feel

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Sound effects

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

One-shot effects in the audio manager

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Loading the effect WAVs

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Making the SFX toggle work

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Where each effect fires

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The particle system

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Emitting blood

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Weapon trails and the level-up fountain

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Updating and drawing

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Emitting effects on damage

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Weapon trails

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Floating damage numbers

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Flashing enemies white

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Shaking the camera

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Wiring it into the frame loop

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Full listing reference

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Chapter 14: Packaging, Distribution, and the Road Ahead

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Technical requirements

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

What you have built

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Preparing for a production build

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Building native executables

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Shipping to the web with WebAssembly

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Measuring performance

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Extending Gopher Survivor

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Resources and community

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Summary

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Conclusion

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

The journey, chapter by chapter

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

What you actually learned

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Thank you

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.

Where you go now

This content is not available in the sample book. The book can be purchased on Leanpub at https://leanpub.com/gameprogramminggolang.