Building GBA games in Go

Brandon Atkinson (Weave DevX)

The GBA

- Released in 2001
- 240x160 LCD (not backlit)
- Successor to the GBC
- Sold 81.51 million lifetime units

The GBA was Nintendo’s first 32-bit handheld console.

Tiny Go

Tiny Go

https://tinygo.org/

- A Compiler For Small Places
- Leverages LLVM
- Supports most (but not all) of Go
- Supports many unusual targets

The GBA cont.

- 16.78 MHz CPU
- 32 KB internal, 256 KB external, 96 KB VRAM
- 16-bit color (BRG)
- 6 video modes
- No OS, or System Calls

Getting Started

Install Tiny Go https://tinygo.org/getting-started/install/

Getting Started

There are very few resources available on go development

https://dev.to/aurelievache/learning-go-by-examples-part-5-create-a-game-boy-advance-gba-game-in-go-5944

https://github.com/tinygo-org/tinygba

https://tinygo.org/docs/reference/microcontrollers/gameboy-advance/

But, there are lots of great C resources available.

https://www.coranac.com/tonc/text/toc.htm

https://www.cs.rit.edu/~tjh8300/CowBite/CowBiteSpec.htm

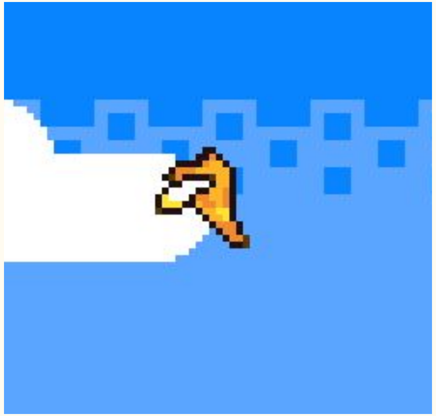
https://problemkaputt.de/gbatek.htm

https://www.reinterpretcast.com/writing-a-game-boy-advance-game

Getting Started

Don't do what I did and do this all from scratch. It's probably the worst possible way to make a game. Instead steal from my code to make your life easier.

https://github.com/bjatkin/flappy-boot



Talking To Hardware

Memmap

- Map hardware registers to go constants and variables
- Provides facilities for working with hardware registers
- https://github.com/bjatkin/flappy-boot/blob/main/internal/hardware/memmap/memmap.h
- https://github.com/bjatkin/flappy-boot/blob/main/internal/hardware/memmap/memmap.go

GetReg and SetReg

- The Go Memory Model does not like to share memory
- *Do not communicate by sharing memory; instead, share memory by communicating.* (Effective Go)
- https://github.com/bjatkin/flappy-boot/blob/57a2f9561d0202ece08ae7550400bf7b9f44d637/internal/hardware/memmap/memmap.go#L76
- TinyGo volatile package

The Type System

- Registers are all just uint16 values
- Don't rely on just naming conventions, leverage the type system
- https://github.com/bjatkin/flappy-boot/blob/main/internal/hardware/display/display.go

A GBA Game Engine

Requirements

- Small, simple, and light-weight
- Fit to purpose (only do what the game needs)
- Prevent the gameplay code from touching hardware
- Useful enough to speed up gameplay development

Organization

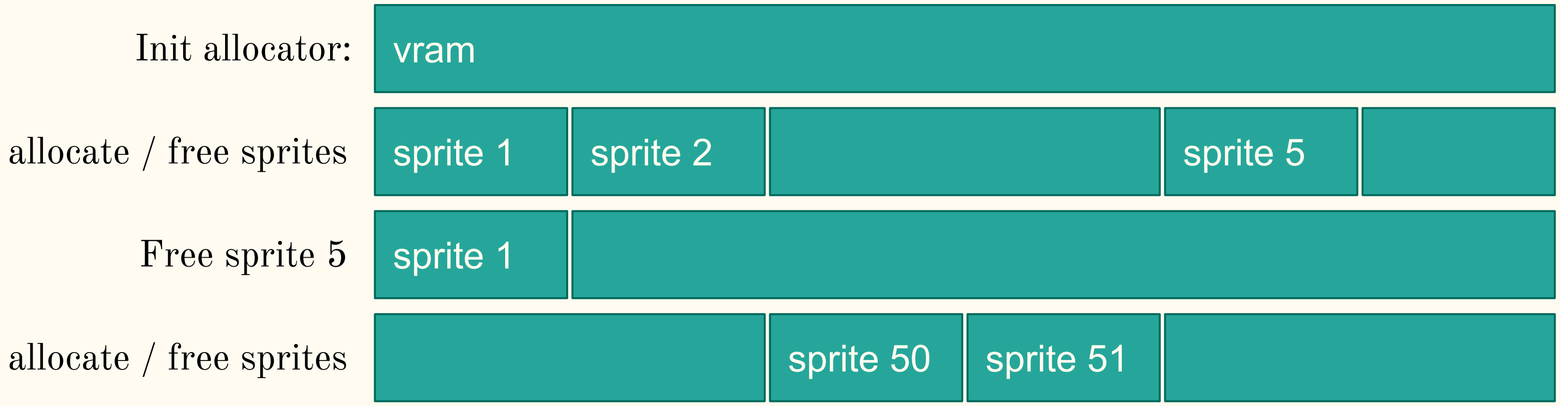
How It’s Organized:

- Assets
- Cmd
- Gameplay
  - Scenes
  - Game Objects
- Internal
  - Alloc
  - Asset
  - Display
  - Game
  - Hardware
  - Key
  - Lut
  - Math

Allocators

Sprite, Background, and Palette data all use custom allocators.

https://github.com/bjatkin/flappy-boot/blob/main/internal/alloc/vram.go



Backgrounds

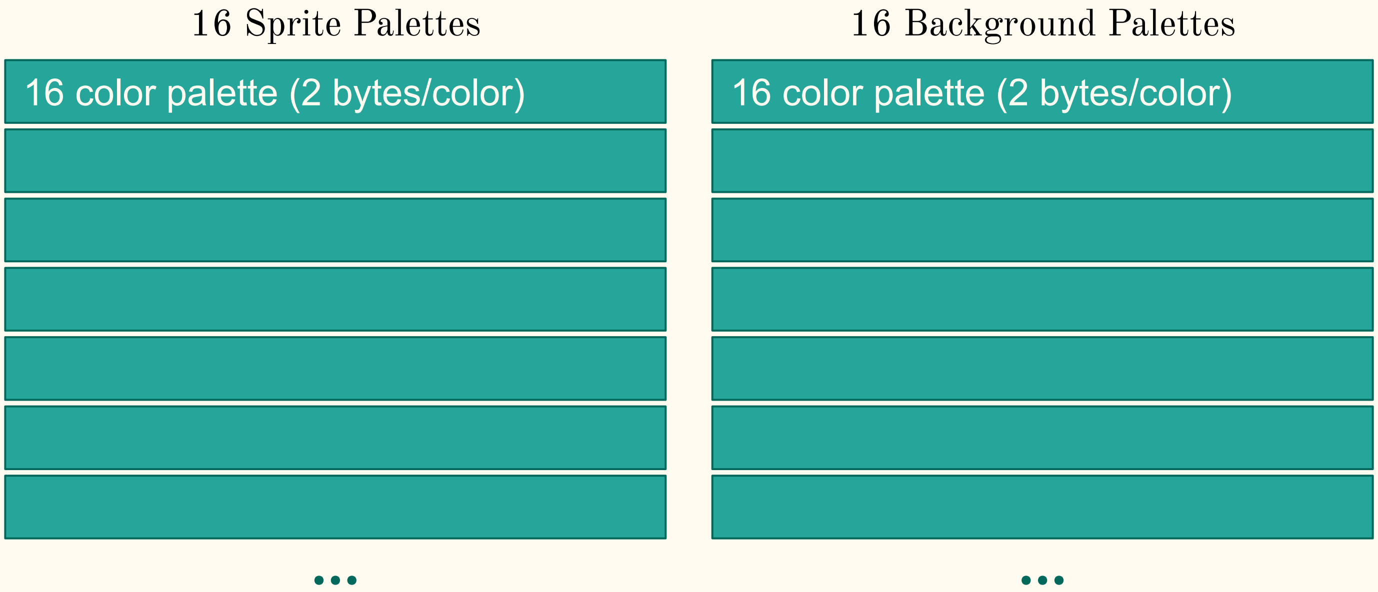
- GBA has 96k of vram
- 6 x 16k char blocks
- There are 6 video modes for backgrounds
    - Mode 0 - tile mode 4 regular bgs
    - Mode 1 - tile mode 2 regular bgs 1 affine
    - Mode 2 - tile mode 2 affine bgs
    - Mode 3 - bitmap mode, 8bpp 1x240x160
    - Mode 4 - bitmap mode, 4bpp 2x240x160
    - Mode 5 - bitmap mode, 4bpp 2x160x128

Backgrounds

- Backgrounds include
  - Pallets
  - Tiles
  - Tile Maps
- Each Char block has 8 screen blocks for tile map data
- I use the bottom 2 char blocks for tile map data and the top 2 char blocks for tile data
- Palettes are stored separately

Color Palettes

- The GBA is capable of displaying 512 unique colors on screen in any one of it's 3 tile modes.



Drawing Backgrounds

- Load the background palette into vram
- Load the background tiles into vram
- Load the tile map into vram
- Enable one of the available backgrounds in the display control register (0x4000000:bits 8-B)
- Configure the BGControll register (0x4000008)
    - Priority (0-4): bits 0-1
    - Set CBB for tiles: bits 2-3
    - Set Color Mode (4bpp or 8bpp): bit 7
    - Set SBB for tile map: bits 8-C
    - Set BG size (small, wide, tall, large): bits E-F
- Configure BGHOffset (0x4000010) and BGVOffset (0x40000012)

Sprites

- GBA has 96k of vram
- 6 x 16k char blocks
- The number of sprites you can load depends on the video mode.
- My engine assumes 4bpp sprites in the bottom 2 char blocks

Drawing Sprites

- Load the sprite palette into vram
- Load the sprite tiles into vram
- Enable sprites in the display control register (0x4000000:bit C)
- Copy sprite data into the OAM buffer
    - X/Y position
    - Color mode
    - Size/ Shape
    - Horizontal/ Vertical mirror
    - Tile index
    - Priority (0-3)
- Copy the OAM buffer into OAM memory

Handling Input

Input Register (0x04000130)

F	E	D	C	B	A	9	8	7	6	5	4	3	2	1	0
-	-	-	-	-	-	L	R	↓	↑	←	→	Start	Select	B	A

- Default high
- Each bit indicates one of the GBA buttons
- A : (0b1111111111111110)
- B & R : (0b1111111011111101)

Handling Input

Once per frame the game engine

1) Loads `currentKeys` into `previousKeys`
2) Loads the hardware input register into `currentKeys`

This makes the following key tests possible

- KeyPressed - the key is being held down this frame
- KeyReleased - the key is not being held down this frame
- KeyJustPressed - last frame the key was not pressed but this frame it is
- KeyJustReleased - last frame the key was pressed but this frame it is not

https://github.com/bjatkin/flappy-boot/blob/main/internal/game/key.go

Playing Audio

Audio is hard, we’ll come back to this if we have time

These will byte you

The GBA has a bunch of things that could “byte” you. Here’s a rapid fire list…

These will byte you

VRAM (and other hardware mapped memory) is managed by the hardware in 16 bit pieces (not 8 like you might think). Only write to this memory at even addresses (e.g. 0x05000000, 0x06000004) and in with `uint16` or `uint32` sized chunks.

Go does not like to share memory. The Go memory model does not allow for hardware mapped memory. This means you'll need to leverage CGo and C's **volatile** keyword or tiny-go's **volatile.Register16** type when reading/writing to hardware registers. I used CGo in flappy-boot since I had issues compiling **volatile.Register16** types on linux. Leveraging generics makes this fairly painless.

These will byte you

The garbage collector will eat all your frames. GC in tiny-go waits for the fully heap to fill up before running a GC cycle by default. This means when GC finally does run, you will drop several frames. Avoid creating garbage whenever you can

- Run a GC cycle every frame (few frames?)
- Use tiny-go's `--print-allocs=.` Compiler flag

I chose to do the latter since flappy bird was small enough for all structs to live safely in the heap. I just had to avoid re-creating structs.

Use fixed point numbers, not floating point. Floating point will suck up valuable cycles and you likely don't need the flexibility they provide. I use 8 fractional bits which is plenty for my needs. Math is a little different with fixed point numbers so be aware of that. The fixed point number type that I use for flappy-boot is defined here.

These will byte you

The GBA is little endian. This is probably the inverse of how you think of byte order for larger numbers. Remember that this applies to palette colors, tile indexes, and sprite pixels (these are treated as **uint16**'s by the hardware). If you see unusual striping or invalid colors in your graphical data, check to make sure your data is using the correct endian-ness. This is especially important for tools that convert graphics data.

You only have 32k of stack memory and 256k of heap memory (that's how tiny-go organizes memory by default). It's really easy to blow this memory up, especially once you start working with audio data. Go automatically determines if data should live in ROM or stack/heap memory. If you're not careful, compilation will fail due to insufficient memory.

** you might be able to mitigate this by leveraging embed.FS and file-streaming to a buffer in RAM. I have not tested this though*

These will byte you

The button input register (KEYINPUT) is default high. This mean the underlying value is 0xFF_FF when no buttons are pressed and 0xFE_00 when all buttons are pressed. See https://www.coranac.com/tonc/text/keys.htm for more information on handling button presses. I think this is done because it's easier to set the hardware up this way given Nintendo used this convection on the NES and other predecessor consoles.

Don't update VRAM while the screen is still drawing. The GBA splits each frame up into 3 segments; VDraw, HBlank, and VBlank. VDraw is the time when the hardware is updating the LCD. This is when you should do gameplay calculations. VBlank occurs after the screen has been drawn. This is when you should update OAM, Color Palettes and other graphics data. You can see where flappy boot waits for VBlank here.

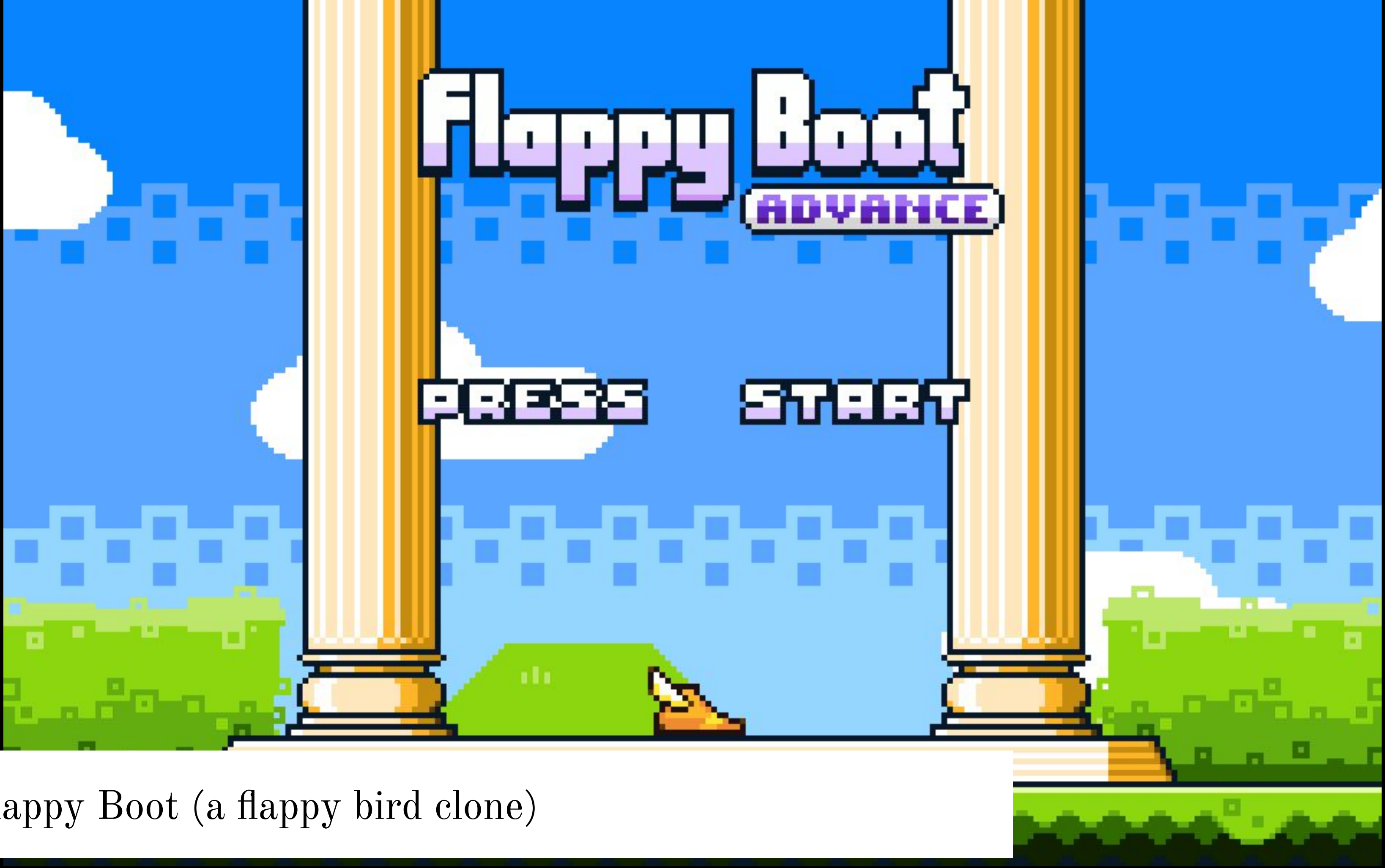
These will byte you

Know hexadecimal and binary and how to convert between them. You'll naturally get good at this as you read through docs and write GBA code. GBA hardware registers and documentation will heavily use both these notations. Just remember easy byte is exactly 2 hex digits. Each hex digit is exactly 1 nibble. Also, Go allows you to break up large numbers with _. This is especially useful for large binary numbers.

Tiny-go is a great tool. It is not however, the Go compiler. Not all go packages are supported. Go code that complies with `go build` may not work all the time. I've had the tiny-go compiler hang forever, or break on seemingly valid code. Make small changes, use version control, test compilation often. Troubleshooting a failing compiler is no fun and the smaller your changes are the easier it will be.

These will byte you

Wherever reasonable, consider using a lookup table (LUT) rather than a function. CPU power is extremely limited on the GBA so any cycles you can save will probably be worth it. Flappy boot uses a LUT to quickly find the sin() of a fixed point value. Tricks like this lead to quick wins when you're running up against the limits of the GBA's CPU.



Flappy Boot (a flappy bird clone)

Conclusion

Future Work (hardest to easiest)

- A better solution for avoiding the GC (I have some ideas on this)
- A sound engine (I made some early attempts at this but it really needs hardware interrupts)
- Debugging (GDB is supposedly supported but I've never been able to get it working)
- A text system (the basics are all there so this would be easy I think)
- Hardware interrupts (These are definitely supported, I just haven't used them yet)
- Better error handling (relies on a good text system)
- Better TinyGo docs
- More Games! (This is really the most important one)

My Final Pitch

I would love to see more small games

- Pico-8
- Tic-80
- Pixel Vision 8
- A Short Hike
- Untitled Goose Game

My Final Pitch

- Small games are easier (and faster) to finish
- Restrictions breed creativity
- 16x16 pixel art is easier to pull off than most art styles
- As developers, we’re capable of handling the technical challenge
- The GBA is small, but has depth for those that want it