Making Games without an Engine

Aidan Hall

March 21, 2024

What are we Talking About?

- Most game developers use large, general-purpose game engines like Unity and Unreal, even for small and simple games.
- These provide helpful features like scene editors, physics simulation and asset stores.
- We're going to throw all that away.

What does that Mean?

- Making games with just code, and basic libraries.
- Implement necessary engine functionality ourselves, in addition to normal gameplay code.
- Alternative title: Making your own Game Engine.

Problems

- Things engines would deal with for you.
  - Physics
  - Scene management
  - Rendering
- Potentially harder to port (especially true of C++).
- Slows development down.

Why Bother?

> *A user can think they understand what they are doing, but they're really just copy-and-pasting code around. Programming thus becomes akin to magical rituals: you put certain bits of code before other bits, and everything seems to work.*

– Jason L. McKesson, Learn Modern 3D Graphics Programming.

Motivation

- Learn how engine functionality is implemented.
- There's a satisfaction to understanding how games work at a deeper level.
- Acquire more general game development skills, rather than being tied down to a specific engine or language.
- Might be all you need.
- Much less work to make a specialised engine.
- Not constrained by the engine.

Examples

- Braid
- Minecraft
- My games

Structuring Games

- Game engines provide the structure for what happens in a game on each frame, and a mechanism to add functionality.
  - In Unity, attach scripts to objects in the scene.
- There is always a main loop that defines this structure.
- It usually takes this general form:

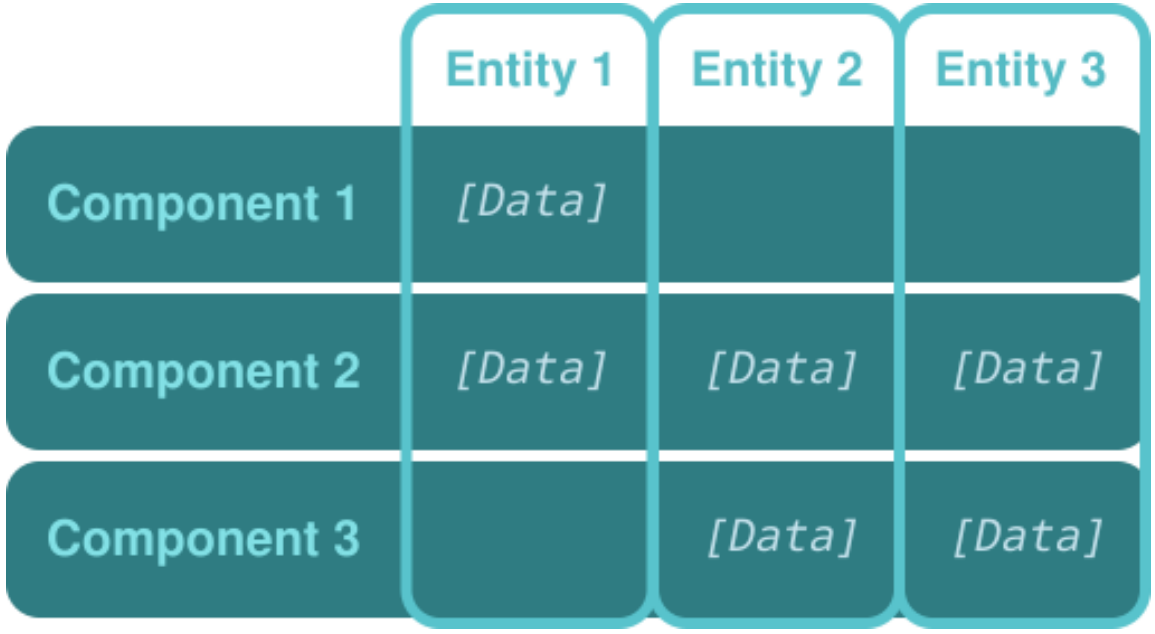
```
int main() {
  while (!GameOver()) {
    ReadInput();
    SimulateWorld();
    DrawToScreen();
  }
}
```

- But how is the data organised?

Data-Oriented Design

- "The only purpose of any code is to transform data."
- By focusing on the data, we can write programs that are simpler and more efficient.

Entity Component Systems



Entities Individual objects in a game world.

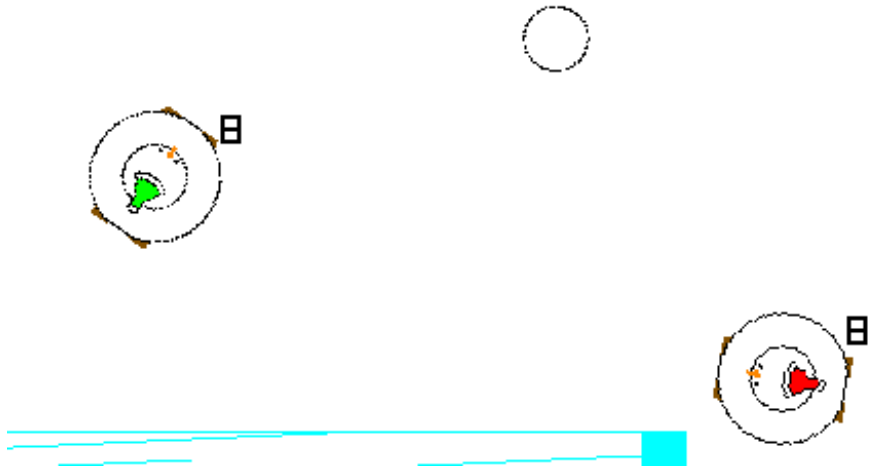
Components Plain data attached to specific Entities.

- E.g. `Vec2 position`, `Vec2 velocity`, `int health`.

Systems Functions that apply a transformation to every Entity with a certain set of Components.

- E.g. Move an Entity: `position += velocity`

Snowmen Sledding

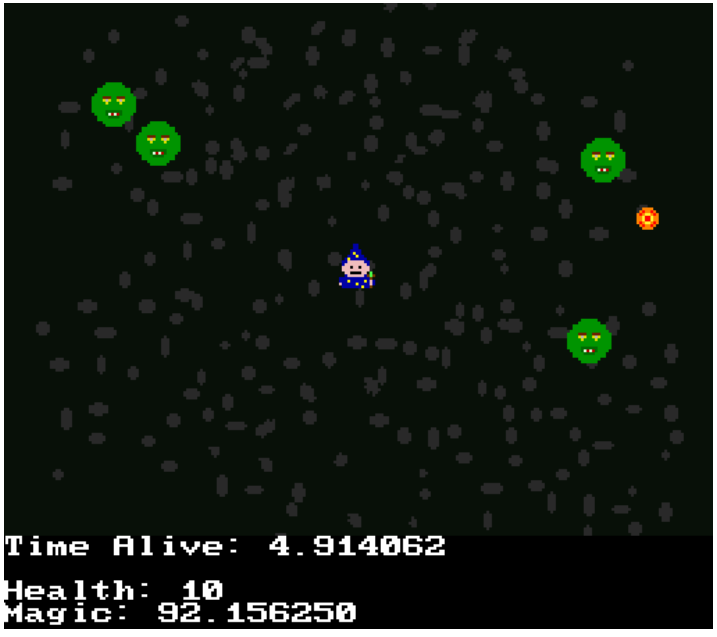


- Based on my own "Tiny ECS", TECS (c. 300 loc).
- Made with Raylib, so it runs on Windows & Linux!
- `KeyboardInput` component: Expresses game controls as data.

Retro Consoles

- Older hardware can't run large game engines, or managed languages like C# and Python.
- You have to use a native-compiled language (or assembly!) to make games for these systems.
- There may be platform-specific constraints a game engine designer didn't account for.

Magic Battle



- A DS game, made using libnds.
- Created with TECS.
- No floating point hardware: Used fixed point instead.

Making Games in C++: Building

- C++ is a standardised language, with several implementations.
  - GCC
  - MSVC
  - LLVM/Clang
- There are also several build systems:
  - Make
  - Visual Studio
  - SCons
- Most of these are (effectively) platform-specific.
- CMake is a cross-platform build system that handles platform-specific details for us.
- It even supports cross-compilation.

Workshop

Windows	`winget install LLVM.LLVM Kitware.CMake`
MacOS	`brew install llvm cmake` (requires Homebrew: https://brew.sh)
Debian & Ubuntu	`sudo apt install llvm cmake`

Thank You

Website https://aidan.sites.uwcs.co.uk

Itch https://aidan-hall.itch.io