FUG Game Engine

“You can make a game or a game engine”
- Albert Einstein

So I heard you want to make a game engine

- Why (seriously)?
  - You want to make a cool game
    - Existing free, popular alternatives (Unreal Engine, Unity)
    - Active supporting community
    - Bad Idea
  - You want to make a small game
    - No need for full scale engine
    - Chance to learn
    - Chance to make mistakes
    - Possibly a good idea
  - You want to make a game engine
    - Chance to learn A LOT
    - Chance to question one's sanity
    - A questionable idea

FUG

- What FUG is
  - An experiment in requirements of a “real” game engine
  - A way to learn (the hard way)
  - Something to #include into one’s portfolio
  - Something to work on with friends
  - An obsession
- What FUG isn’t
  - A game engine
  - A project
  - Documented

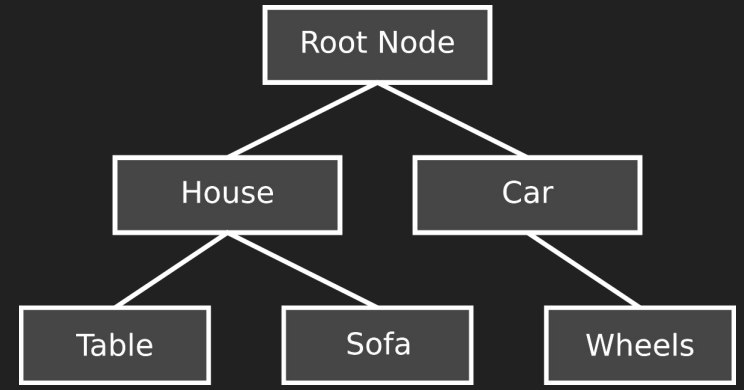
Requirements for a game engine

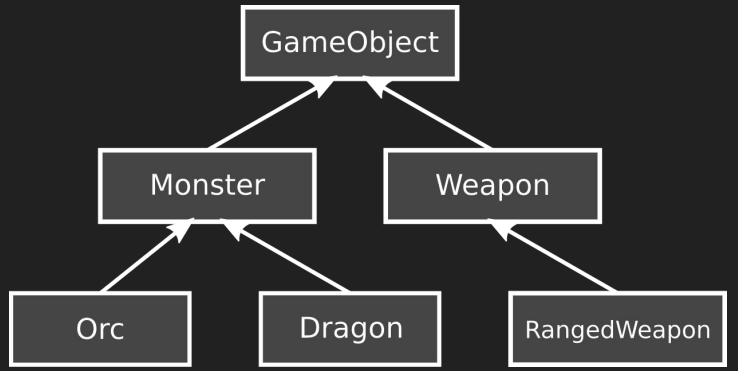
- Main data structure for game objects
- Communication (user input, inter-object messaging)
- Resource management
- Rendering

Journey from everyone’s first mistakes to this day

Main data structure, before

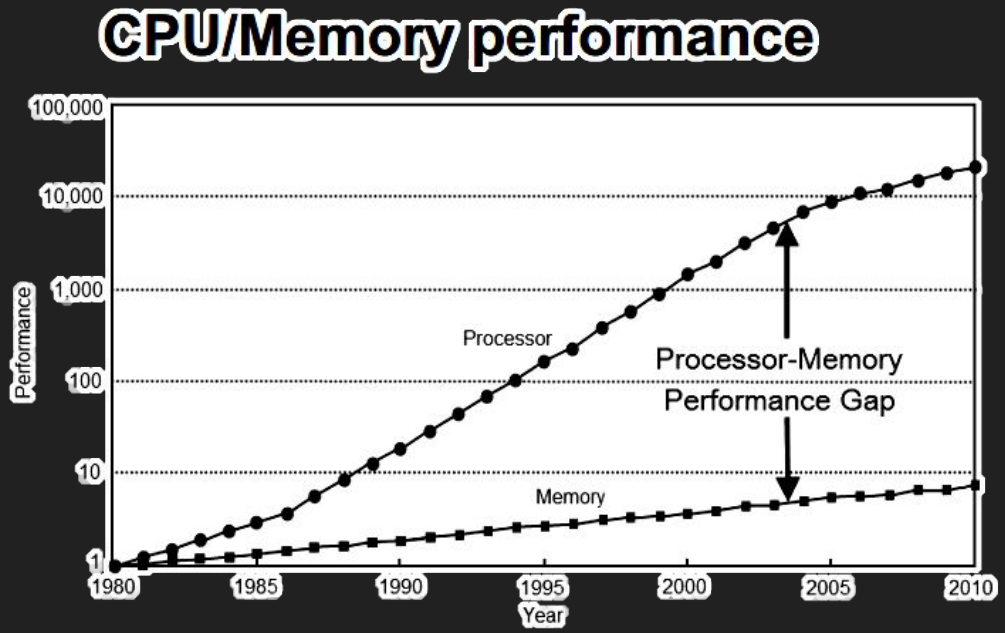
- What everyone thinks is the greatest thing ever after their first C++ course
- Scene graphs
  - Organize objects using a node hierarchy
- Polymorphic objects
  - Inherit and reuse object properties
- Polymorphic objects in scene graphs
  - Could be useful in small games / restricted cases
  - Terrible idea in general





Let’s talk about performance

- It’s all about memory
  - RAM is slow
- So how we get anything done?
  - Caching
    - Smaller quantities of faster memory
    - Fetch bigger chunks and store them in cache
      - Crucial to make use of this chunk



Accessing Data

- Trees
  - Sometimes great, usually terrible
    - Traversing pointers leads to cache misses
  - Scene graphs and polymorphic objects are both tree structures
- C++ peculiarities
  - `std::vector<BaseObject*>`
    - Every object is new-allocated and located randomly in memory
    - Have mercy on your hardware, don’t do this

OOP and code maintainability

- Inheritance
  - Sounds great at first
  - Inherit properties you don’t need or make almost identical duplicate
  - Leads to tightly coupled code
    - Make a change somewhere, everything falls apart
    - Absolute horror to maintain in the long run (ask Java programmers)
- OOP ties data and functionality together
  - Again, tightly coupled
- Modularity
  - Key to maintainable code

Main data structure, now

- Entity-Component system
  - Data and functionality decoupled
    - Maintainability
  - Entities are collections of components
    - Use only what you need
  - Systems access and modify combinations of components
  - Components accessed sequentially, number of cache misses minimized
- Examples of components:
  - PositionComponent
  - PhysicsComponent
  - ModelComponent
- Examples of systems:
  - PhysicsSystem
    - Uses PhysicsComp, PositionComp
  - RendererSystem
    - Uses PositionComp, ModelComp

Communication, then

- Observer pattern
  - Directly modify other object once required
    - Complex data access patterns, bad
  - Depends on properties of both objects
    - Tightly coupled code, again, bad

Communication, now

- Events and Event Manager
  - Distribute and receive events in centralized manner
    - Event Manager acts as a “post office”
  - React to events once relevant (or don’t)
    - Completely decoupled from event sender
    - Coherent data access

Resource management, then

- What’s a resource anyway?
  - Anything you need only a single copy of
  - Loaded / created in run-time
  - 3D meshes, textures, shaders etc..
- Just load everything you need at the moment
  - Loading screens

Resource management, now(soon[maybe])

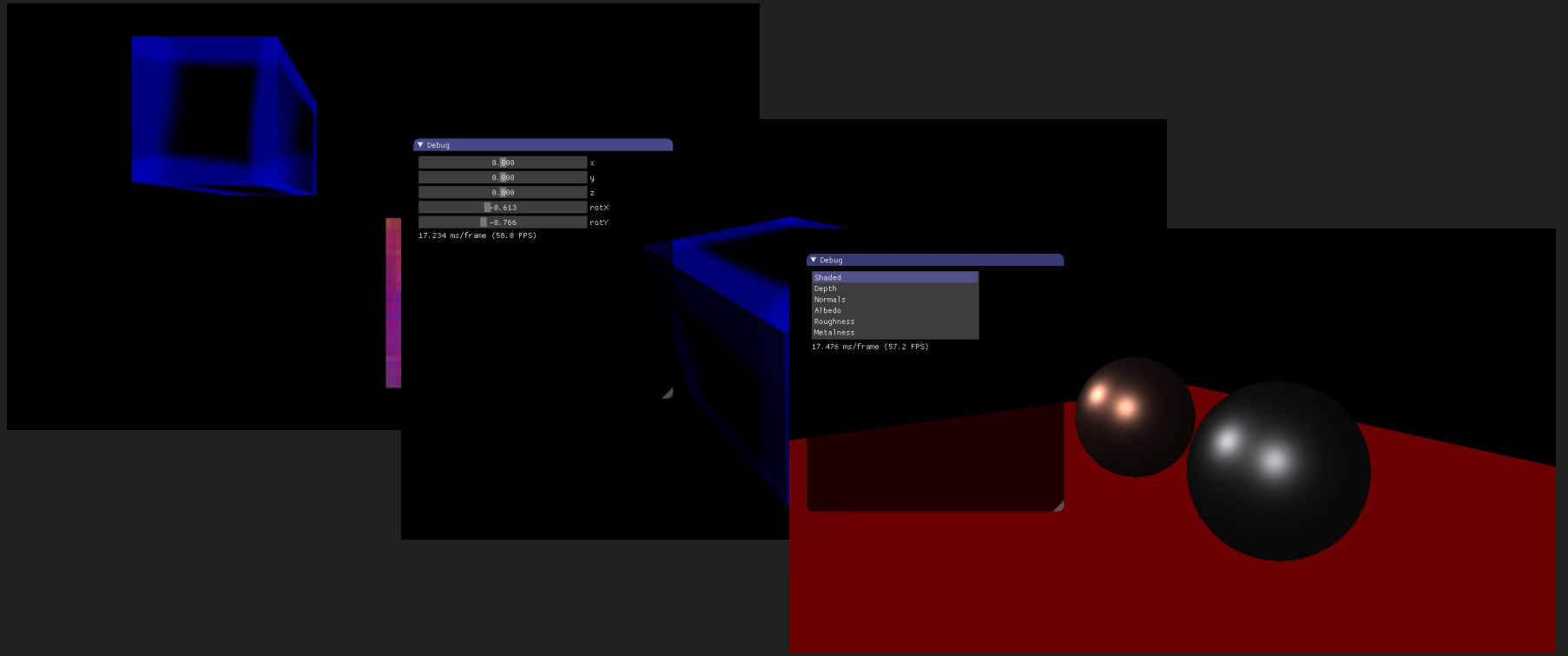
- Case Skyrim
  - What to keep in memory, what to load?
- Hierarchical resources
  - Unload only parts of a resource
  - Loading dependencies
- Multiple ways of initializing a resource
  - e.g., texture can be loaded from file or procedurally generated
- Keeping track of what’s in use
  - Smart pointers with reference counting

Rendering FUG

Goals (tbc)

- Try cool stuff
  - First time doing most of this
- Generic interfaces
  - Avoid slightly differing copies of e.g. resources
  - Avoid “hard coding” where possible -> map things at runtime
- Optimize later™

Story so far


Debug
0.000
0.000
0.000
-0.613
-0.766
x
y
z
rotX
rotY
17.234 ms/frame (58.0 FPS)
Debug
Shaded
Depth
Normals
Albedo
Roughness
Metalness
17.476 ms/frame (57.2 FPS)

What we have

- Generic types for common resources
- Physically based materials, UE4-style
- Deferred shading
    - directional lights at least

What we have

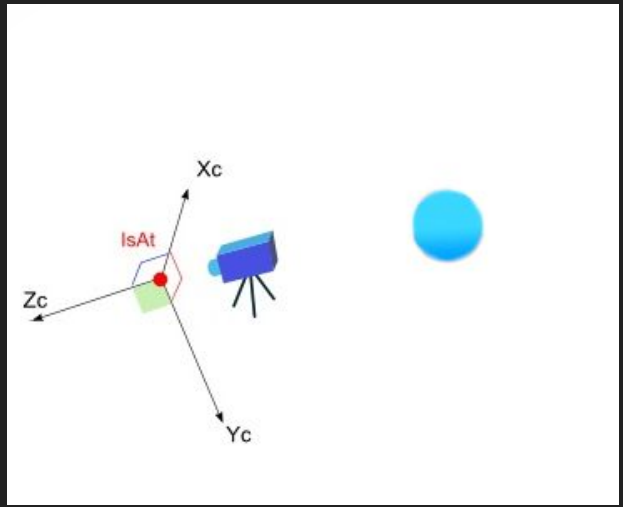
- Generic types for common resources
- Physically based materials, UE4-style
- Deferred shading
  - directional lights at least

What would be nice

- More types of lights
- Shadows
- Reflections
- Ambient Occlusion
- Skinning
- Subsurface scattering
- Transparency
- Level Of Detail -switching
- Occlusion culling
- GPU particle engine
- Post processing
  - Bloom
  - Depth Of Field
  - Gamma Correction (!)
- Concurrent rendering
- [insert other cool stuff here]

Newbie lessons

- Plan ahead
  - Rewriting is part of the journey
  - …but changing interfaces is not fun
- Verify the math (by hand)
  - Fun fact: some reference material or library functions might flip your z-axis



Resources

CS-C3100 Computer Graphics

Möller et. al. : Real-Time Rendering

- almost a decade old now but basics still apply

Rendering tutorials (modern OpenGL)

https://learnopengl.com, http://ogldev.atspace.co.uk