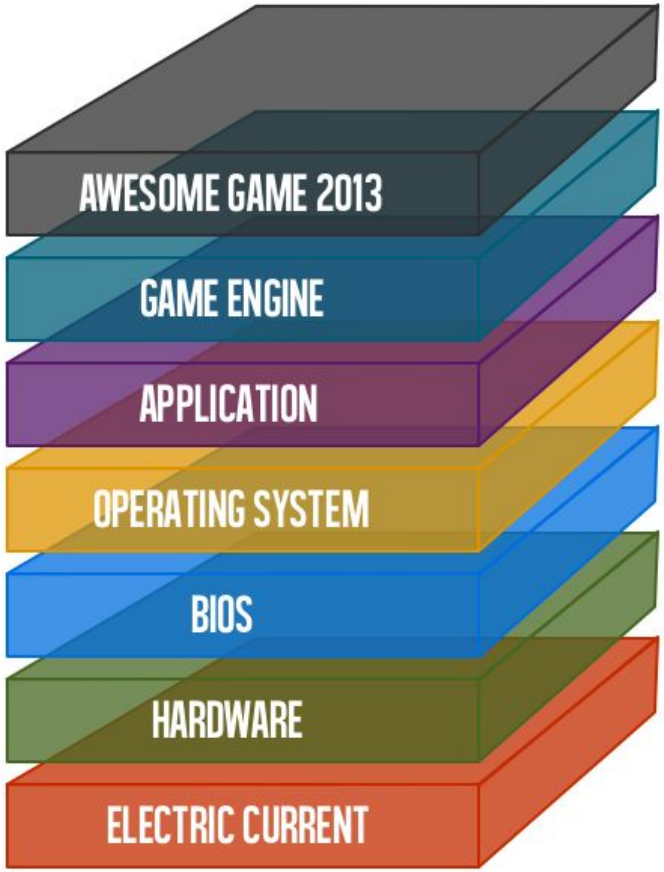
MVD: Engine Programming

01 - The Basics

alunthomas.evans@salle.url.edu

laSalle ENG
Universitat Ramon Llull


AWESOME GAME 2013
GAME ENGINE
APPLICATION
OPERATING SYSTEM
BIOS
HARDWARE
ELECTRIC CURRENT

Game vs Engine

Engine ‘made for’ game:

vs

Game ‘made with’ engine:

What we're going to do...

1) Study 'modern' engine design

2) Make a 'modern' 'generic' 3D game engine with:

- input
- resources
- collisions
- custom components
- course deliverable: AI implementation

3) Prepare for 'Advanced Graphics I & II'

4) Do **a lot** of programming in C++

Course overview structure

Data-driven engine design

(OpenGL revision)

Meshes, Materials and Geometry

Camera and Light (matrices revision)

Input System and Debug drawing (OpenGL buffer revision)

Collision detection and FPS

Custom behaviour scripting

Quarternions, loading levels

Project - AI Implementation

User interfaces

Frustum Culling

Advanced Graphics I

Skybox and Reflection mapping

Uniform Buffer Objects

Light Casters (point, directional, spot)

Render-to-Texture

Post-processing FX project

Shadow Mapping

Deferred Rendering

Multimaterials

Multitexture

Terrain

Terrain rendering project

Particles, Transform feedback

Advanced Graphics II

Animation timeline, fixed animations

Drawing and moving bones

Mesh skinning

Animated character project

Morph target Animation

Final project - putting it all together

(Geometry shaders

Water

Billboards

SSAO)


Goliat
7:59 / POGODNIE
21
BESTIA Z TOUSSAINT
Pokonaj olbrzyma.
ZADANIE ZAKTUALIZOWANE
BESTIA Z TOUSSAINT
5 Udko kurczaka
5 Wyśmienita Jaskółka
∞ Bełty
Miecz srebrny
Szybki atak
Silny atak
Unik
Rzuć Znak
Wzywanie

RENDERING

- BACKGROUND
- STATIC GEOMETRY
- DYNAMIC GEOMETRY
- ANIMATION
- SHADOWS/WATER/ PARTIC.
- POST-PROCESSING
- USER-INTERFACE

RENDERING

- BACKGROUND
- STATIC GEOMETRY
- DYNAMIC GEOMETRY
- ANIMATION
- SHADOWS/WATER/ PARTIC.
- POST-PROCESSING
- USER-INTERFACE

MECHANICS

- PLAYER INPUT
- GAME LOGIC
- ENEMY INTELLIGENCE
- ALLY INTELLIGENCE
- STORY/QUEST MECHANICS

RENDERING

- BACKGROUND
- STATIC GEOMETRY
- DYNAMIC GEOMETRY
- ANIMATION
- SHADOWS/WATER/ PARTIC.
- POST-PROCESSING
- USER-INTERFACE

MECHANICS

- PLAYER INPUT
- GAME LOGIC
- ENEMY INTELLIGENCE
- ALLY INTELLIGENCE
- STORY/QUEST MECHANICS

AUDIO

- MUSIC
- FX

RENDERING

- BACKGROUND
- STATIC GEOMETRY
- DYNAMIC GEOMETRY
- ANIMATION
- SHADOWS/WATER/ PARTIC.
- POST-PROCESSING
- USER-INTERFACE

MECHANICS

- PLAYER INPUT
- GAME LOGIC
- ENEMY INTELLIGENCE
- ALLY INTELLIGENCE
- STORY/QUEST MECHANICS

AUDIO

- MUSIC
- FX

PHYSICS

- TRIGGERS
- COLLISIONS
- PHYSICS

RENDERING

- BACKGROUND
- STATIC GEOMETRY
- DYNAMIC GEOMETRY
- ANIMATION
- SHADOWS/WATER/ PARTIC.
- POST-PROCESSING
- USER-INTERFACE

MECHANICS

- PLAYER INPUT
- GAME LOGIC
- ENEMY INTELLIGENCE
- ALLY INTELLIGENCE
- STORY/QUEST MECHANICS

AUDIO

- MUSIC
- FX

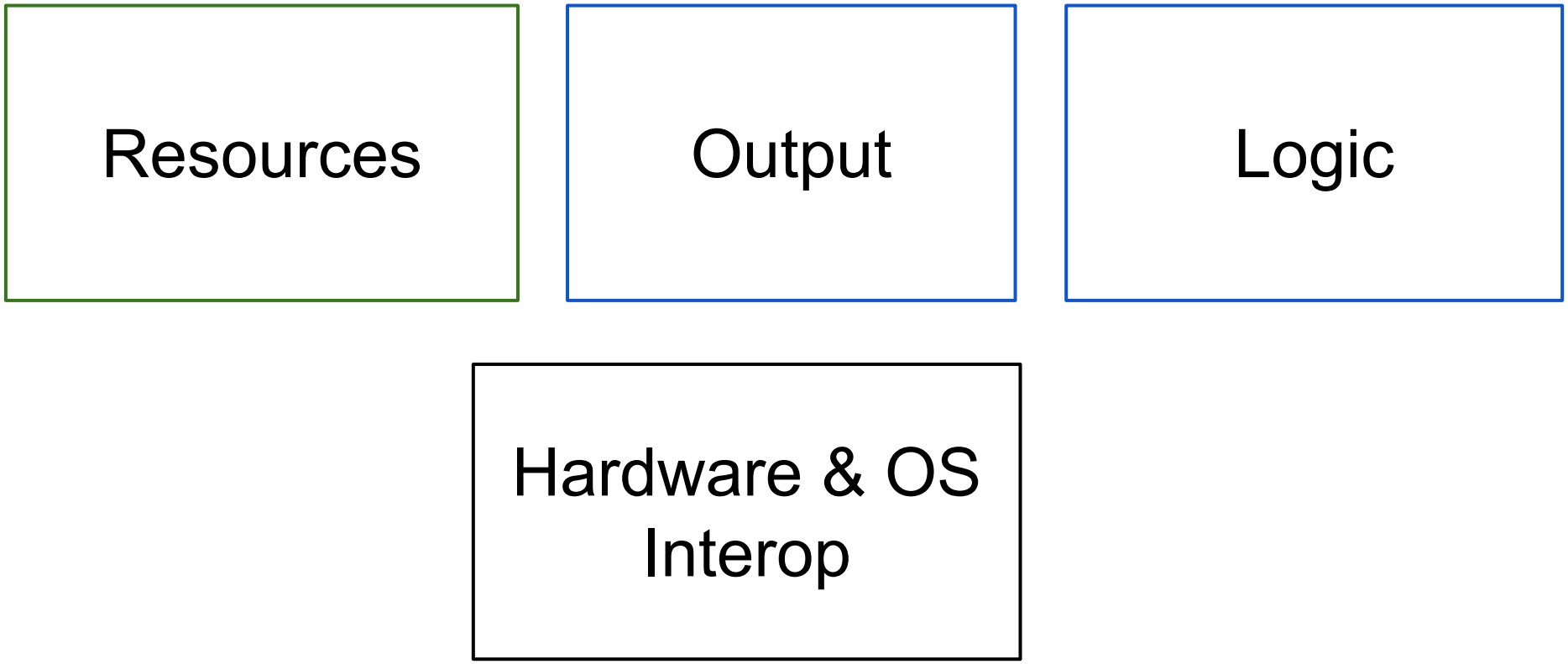
PHYSICS

- TRIGGERS
- COLLISIONS
- PHYSICS

OTHER

- NETWORK
- MENUS
- INVENTORY
- OS
- …

Core elements of a game engine



Game Engines are written in C++ to make speaking to underlying OS and hardware as fast as possible

Resources

Need to be loaded into memory

Meshes, materiales, textures, animations, audio etc

Question: how fast should this be?

Logic and Output: the game loop

```
while (1) {
  updateGame();
}
```

`updateGate()`:

- **reads** input
- **calculates** new situation of game
- **presents** something to player

Question: how fast is the game loop?

Overview of Game Engine Structures:

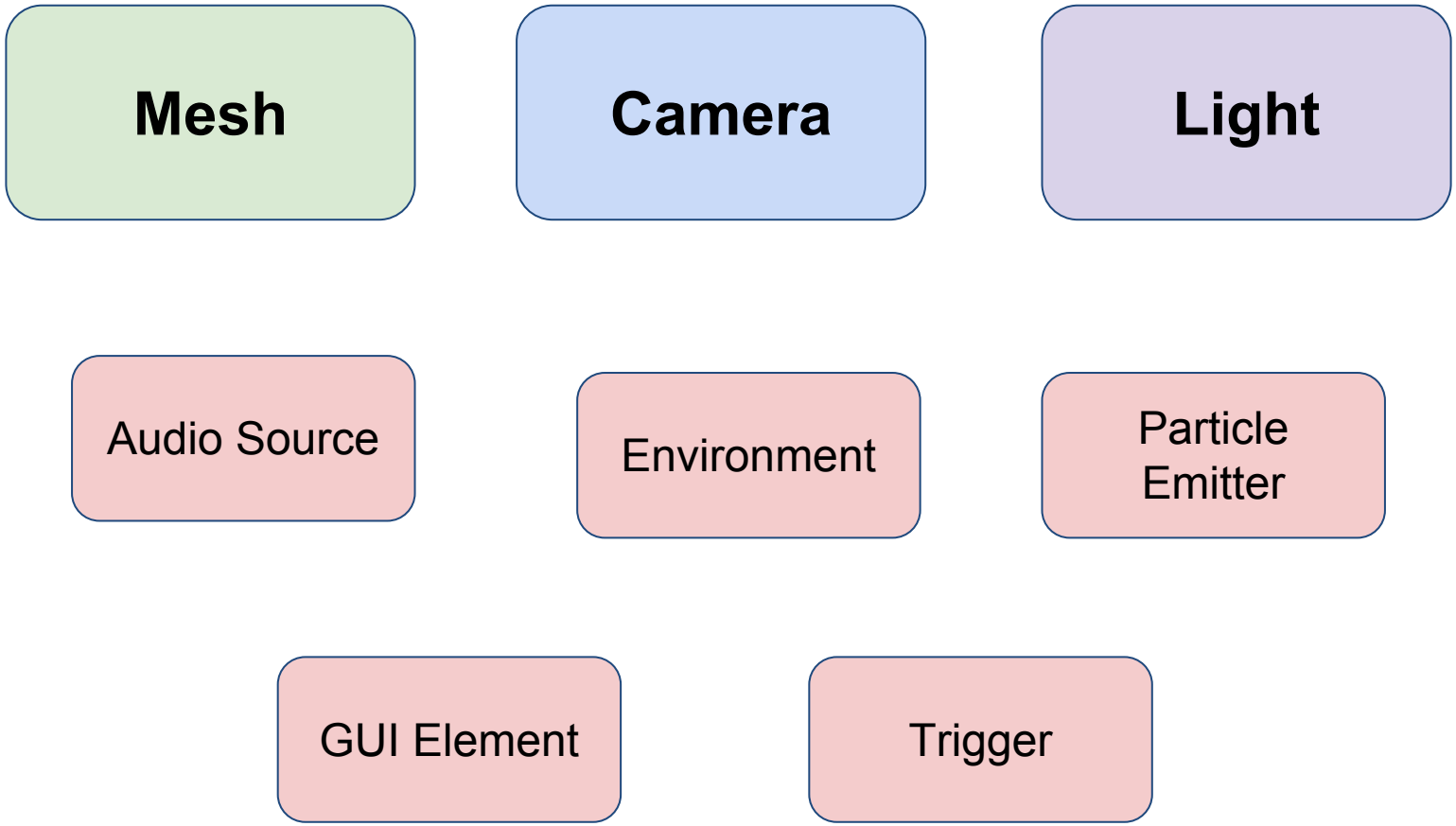
1. ‘Classic OOP’
   - used by 1000s and 1000s of games
   - Fits OOP paradigm perfectly
   - Unflexible: behaviour tied to structure
   - ‘Slow’
   - not used by anybody serious any more

Overview of Game Engine Structures

2. Entity-Component Model
   - Much more flexible
   - Current: made popular by engines like Unity

3. Data-driven (E.C.System) Model
   - ‘fast’
   - flexible
   - ‘the future’.... (in fact, the present!)
     - (ref: https://unity3d.com/unity/features/job-system-ECS)

Classic model logic - game objects



Object properties

Every object should have:

- a position
- a rotation
- a scale

} → **Model matrix**

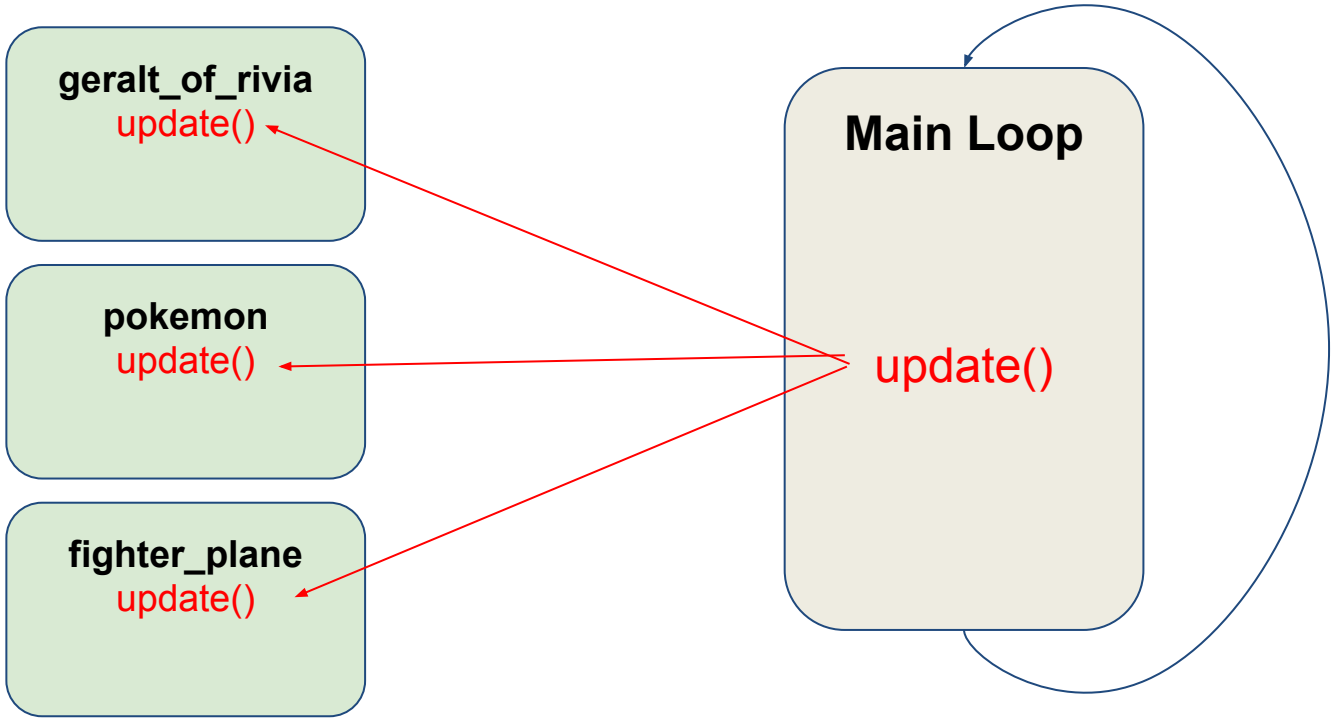
- a name

and possibly:

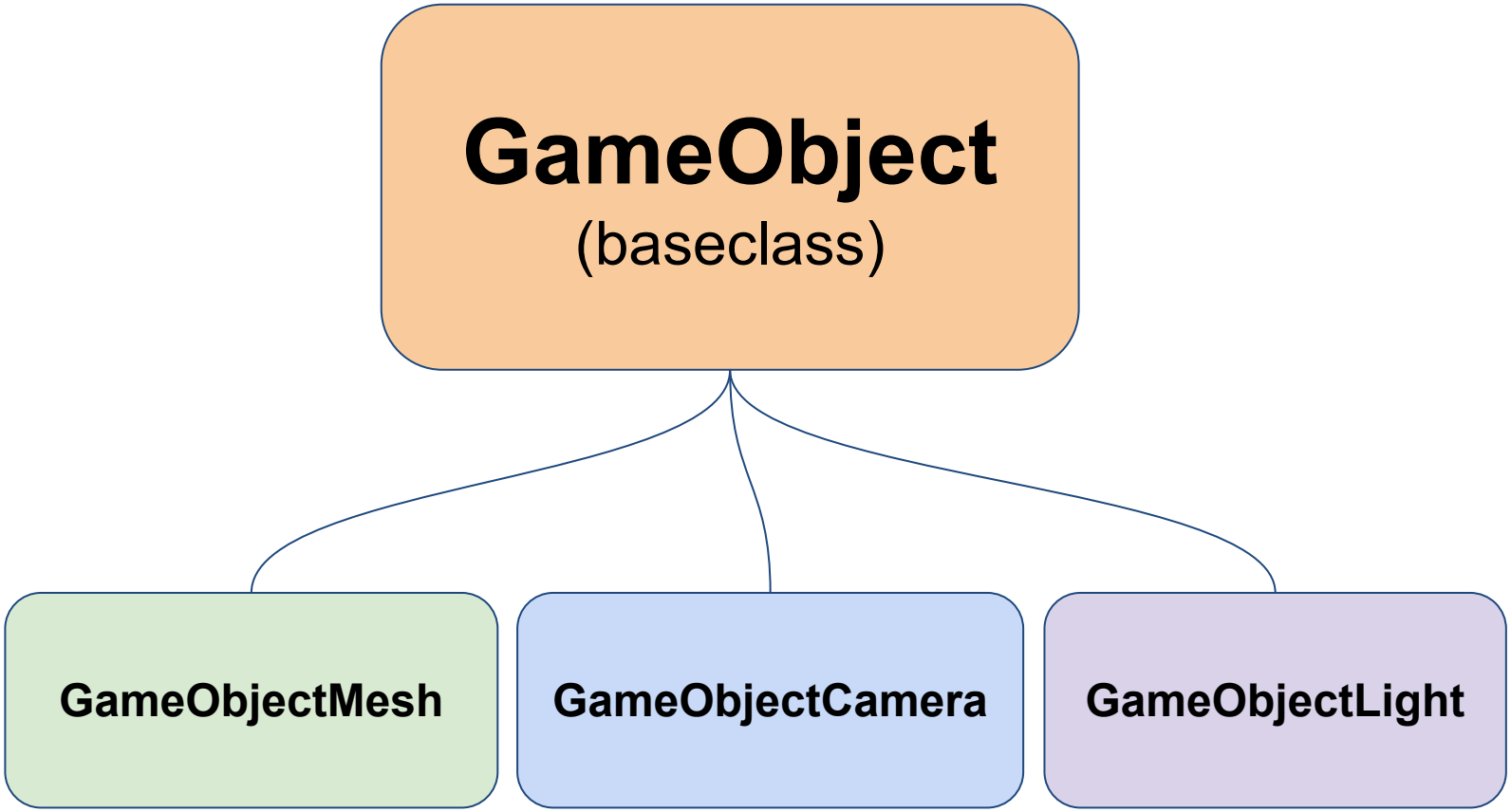
- enabled/visibility status
- dimensions
- a material (shader + textures)
- custom properties (audio file, particle properties etc)

Updating obejcts

The main loop functions call the equivalent functions for **every object in the game**.



Mapping this to code, using inheritance



derived classes

Classic Model

LOGICAL: every game object looks after itself

EASY TO UNDERSTAND: and easy to code

OOP ADVANTAGES: objects can expose what's needed

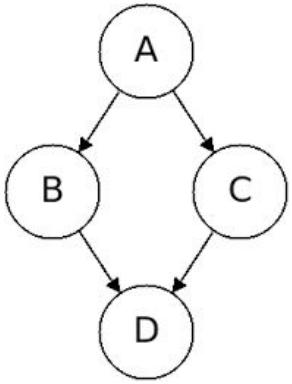
Classic Model: SceneGraph

ATOMIC DATA: loads of code has to be reused

MULTIPLE INHERITANCE: “deadly diamond of death”

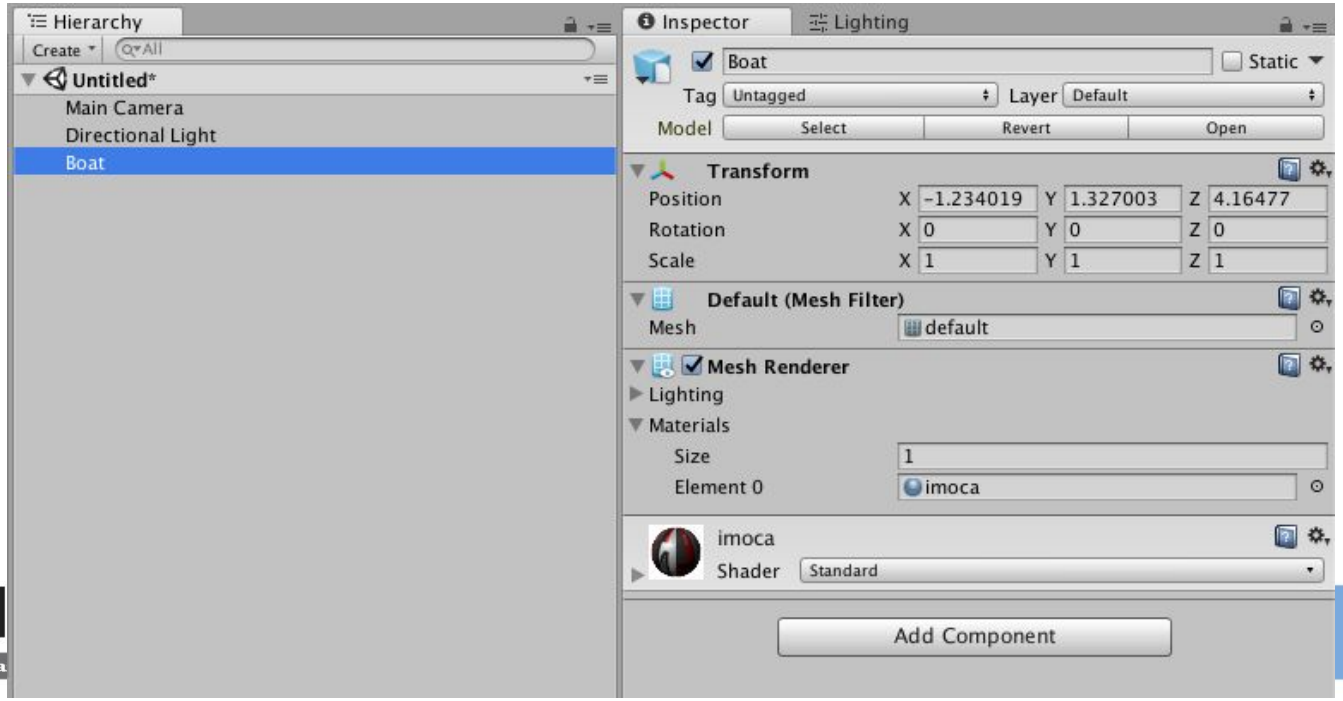
DEPENDENCIES: must wait to update

INEFFICIENT MEMORY: load all information for each GO



Entity Component Model

- GAME OBJECTS (OR ***ENTITIES***) BECOME GENERIC CONTAINERS
- THERE IS NO CLASS HIERARCHY
- ENTITY BEHAVIOUR IS ENCODED IN THE **COMPONENTS**



Code reuse

Behaviour is coded in **components**

So no need to hard-code behaviour into class, just add the relevant component

Entity Component Model code sample

```
class Component {

};

class TransformComponent : public Component {
    Matrix44 transform;
};

class Entity {
    std::vector<Component*> components;

    void update()
        for (auto comp : components)
            comp->update();

};
```

Solved classic model problems:

~~ATOMIC DATA: loads of code has to be reused~~

~~MULTIPLE INHERITANCE: "deadly diamond of death"~~

DEPENDENCIES: must wait to update

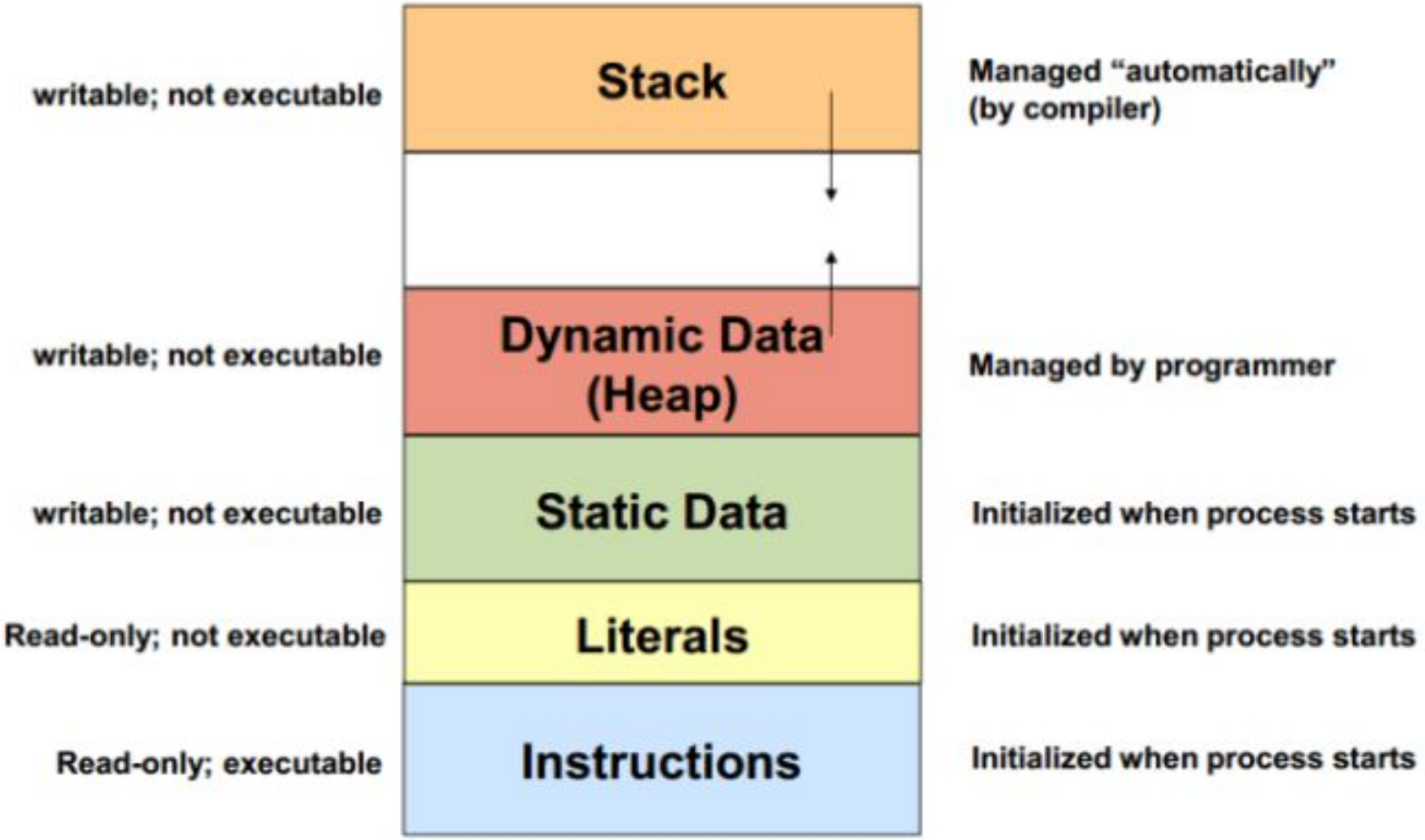
INEFFICIENT MEMORY: load all information for each GO

A break to talk about resources

`Each component` contains pointers to mesh data, material data, textures

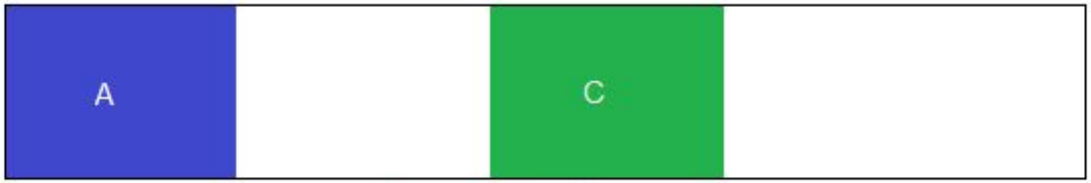
Where is all this data stored? In which part of the memory?

Resources: need to be loaded into memory

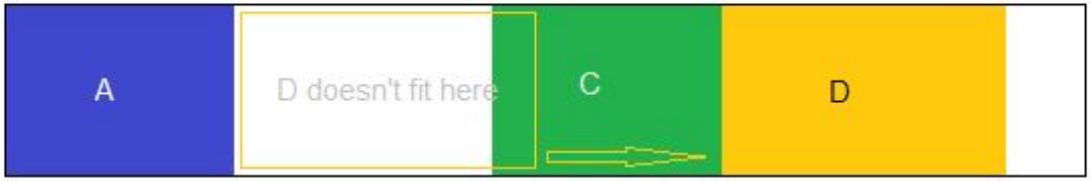


Resource memory allocation

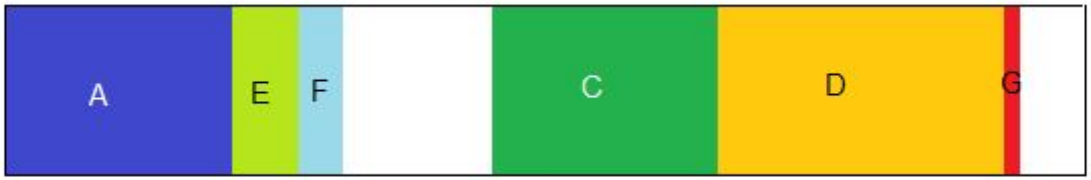
Some Arrays A, B and C in Memory, nicely organized



Let's free one of them
and let's try creating a bigger Array than B called D in the next step.



Ohh...



Oh No.

So much free Space, yet no possibility to create a continuous Array anymore.

Resource allocation in the heap is, by default, **fragmented**.

This is a problem that affects all game engines.

All game engines use some sort of custom memory pool to avoid fragmentation as much as possible

(ref: http://www.gamasutra.com/blogs/MichaelKissner/20151104/258271/Writing_a_Game_Engine_from_Scratch__Part_2_Memory.php)

Contiguous memory

Game engines go to great lengths to **keep memory contiguous.** This is for two reasons:

i) To avoid fragmentation (as mentioned above)

ii) To maximise CPU cache usage

Where's my data?

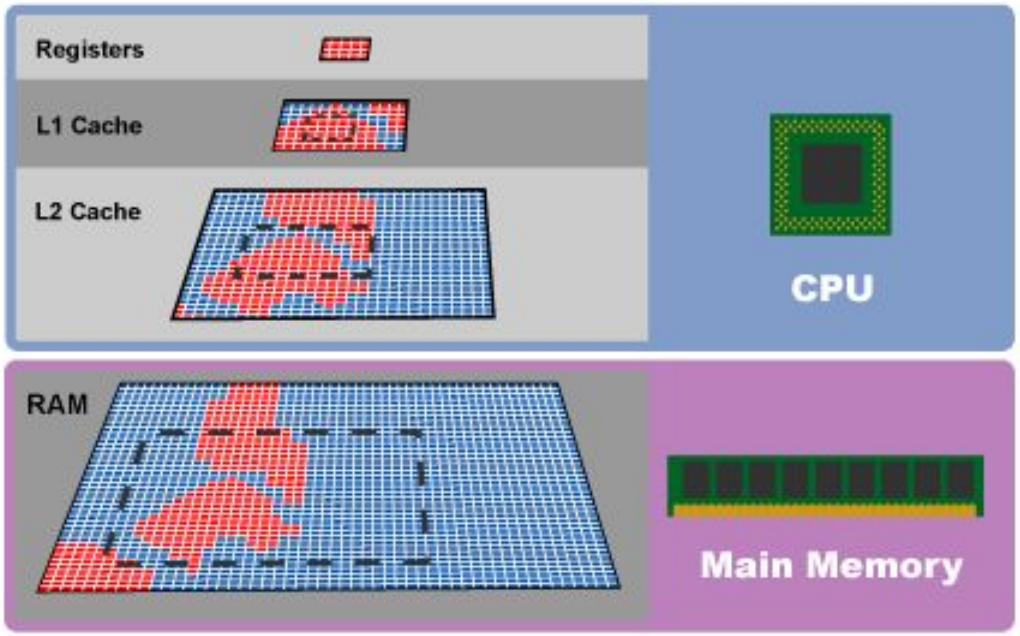
CPU REGISTERS

L1 CACHE

L2 CACHE

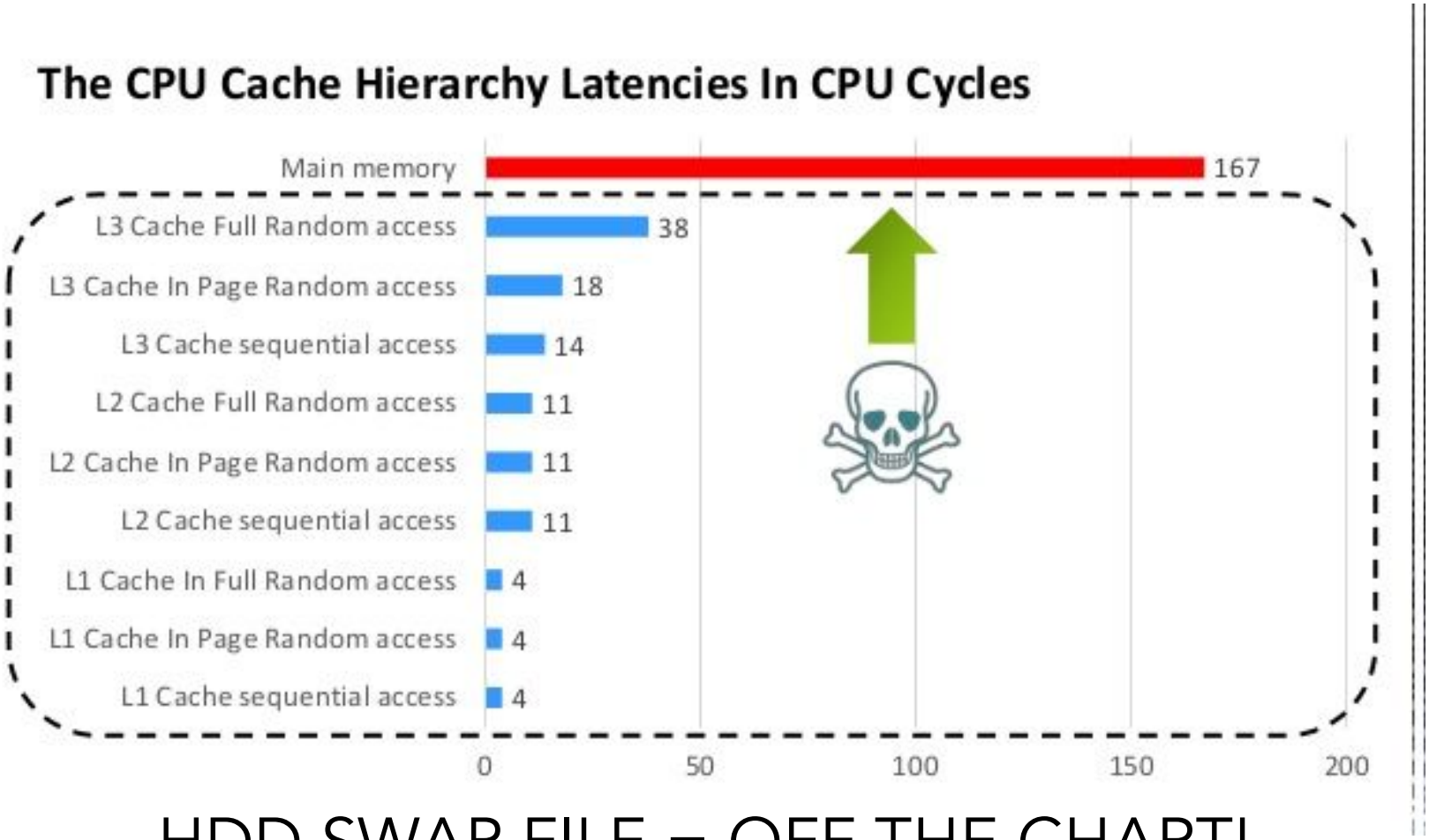
(L3 CACHE)

RAM



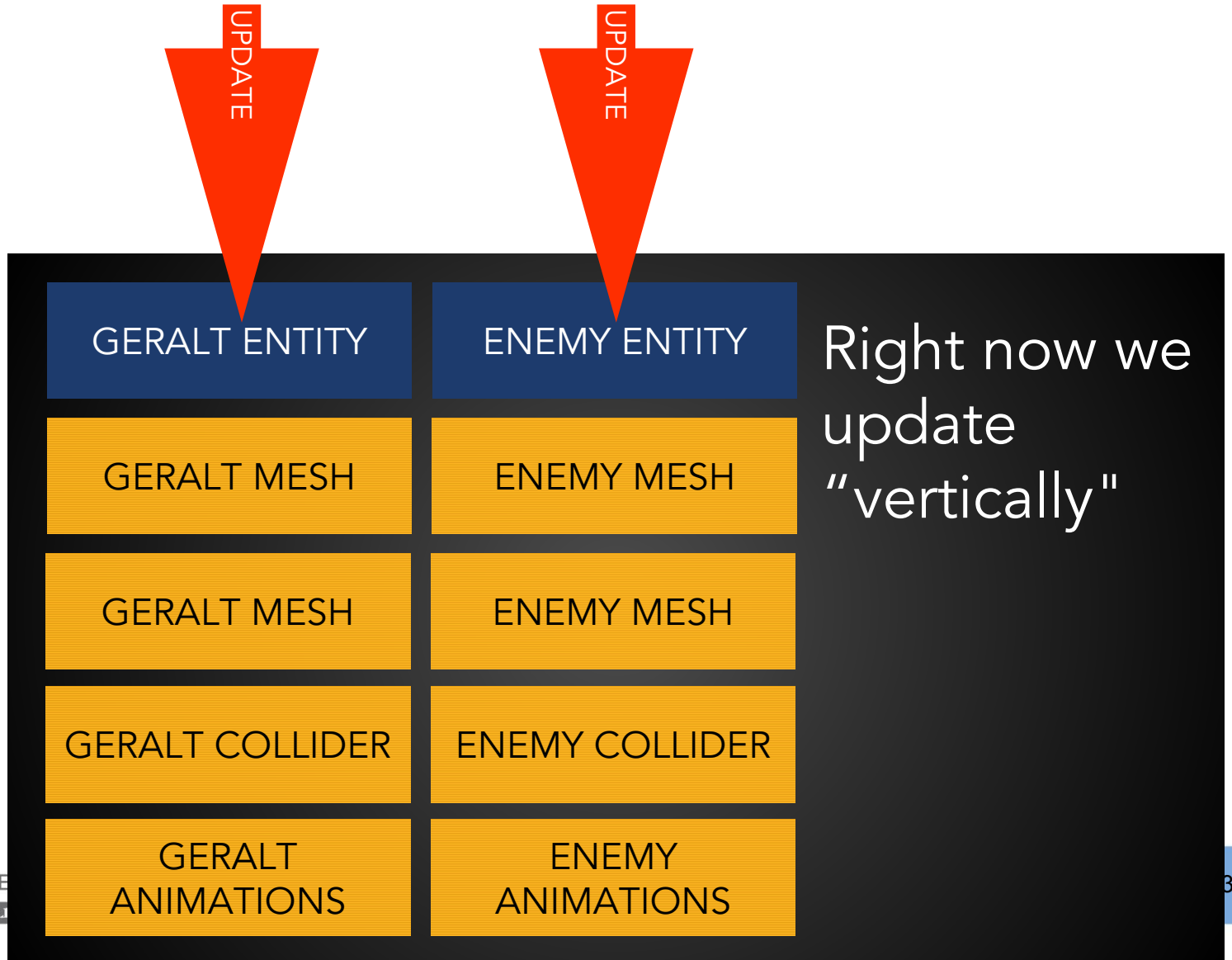
HARD-DISK SWAP FILE

Data Access time



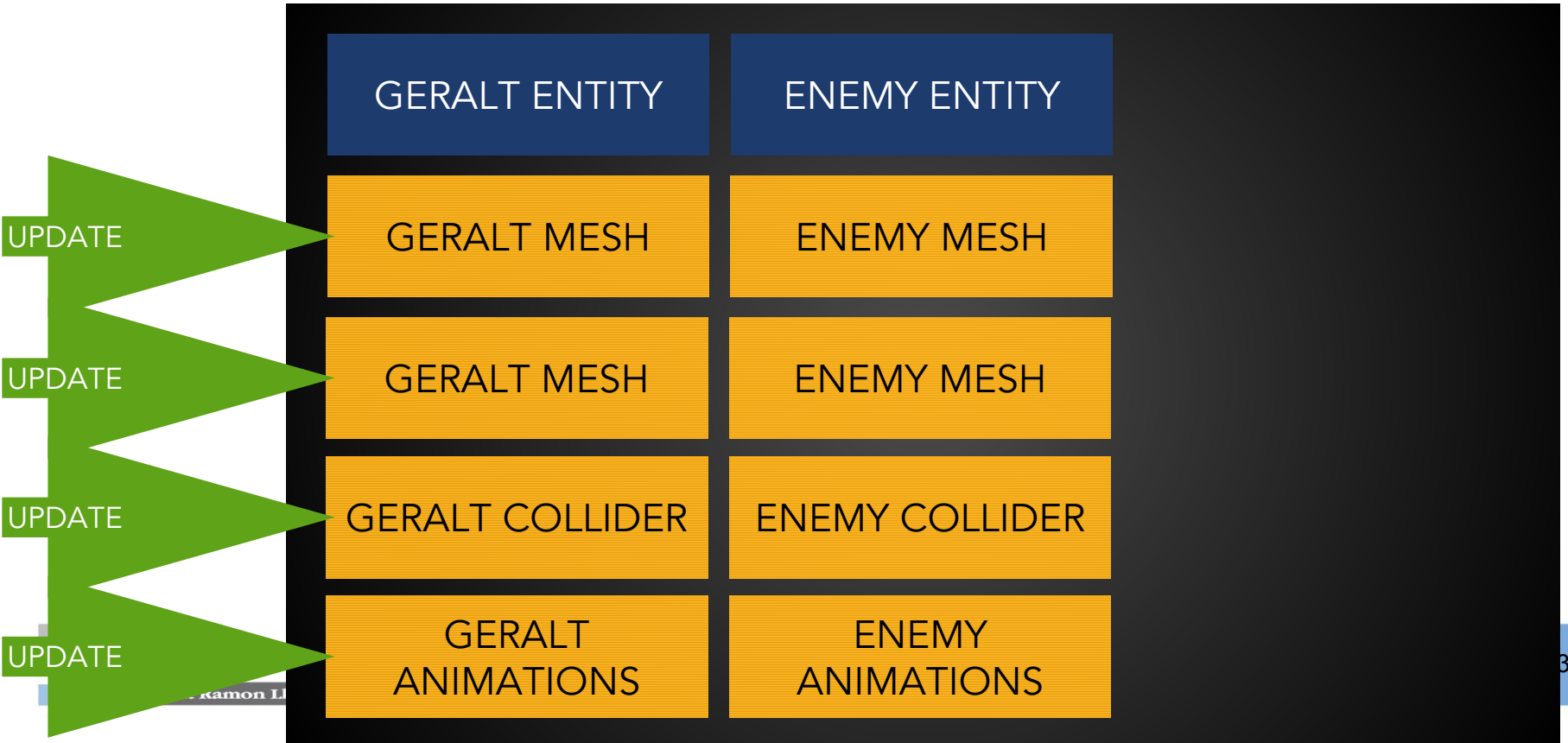
HDD SWAP FILE = OFF THE CHART!

ENTITIES HAVE LOTS OF DIFFERENT COMPONENTS



WHY DON'T WE….

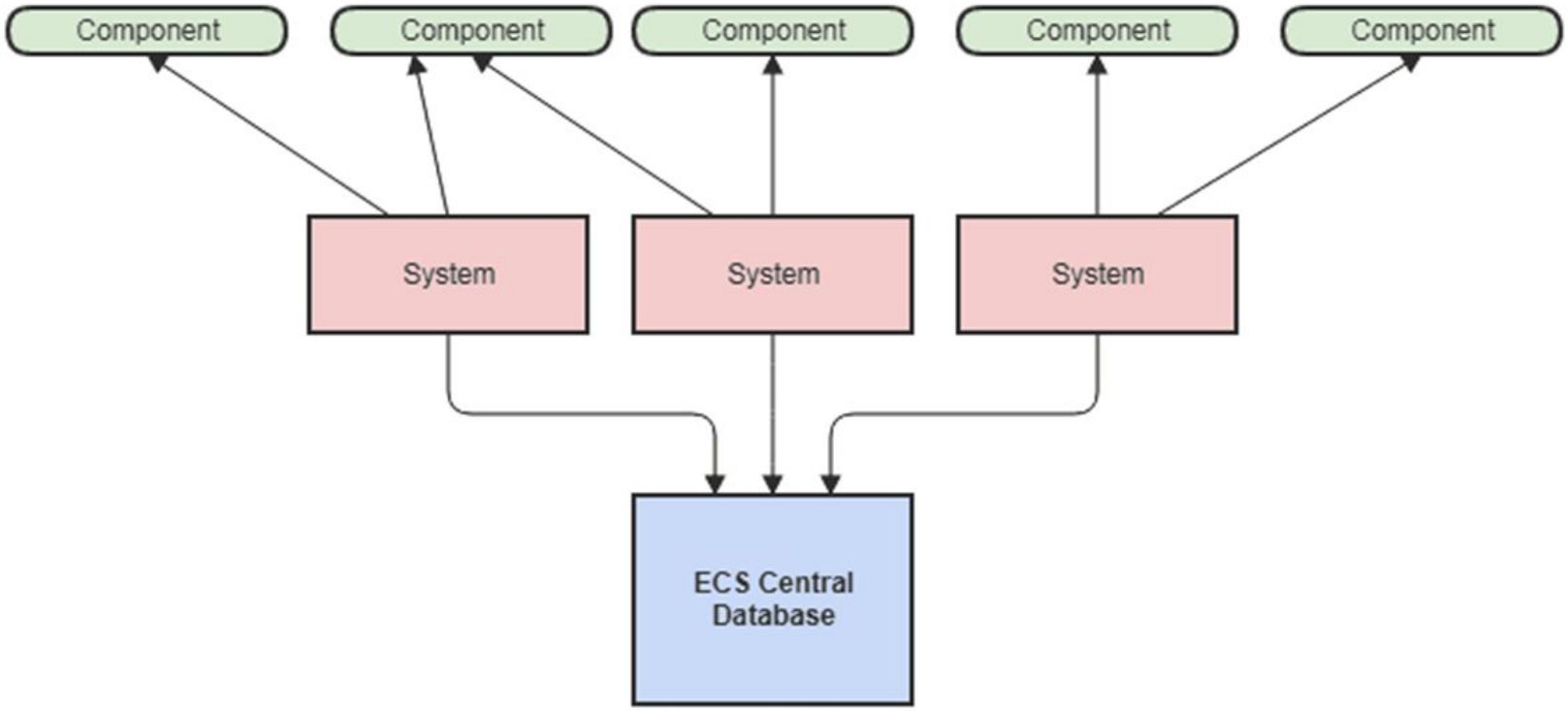
Update
"horizontally"?



The ‘Entity Component System’ model

Game is organised around ‘Systems’ of different types.

Each system updates all the components of a certain type *regardless of which entity they belong to.*

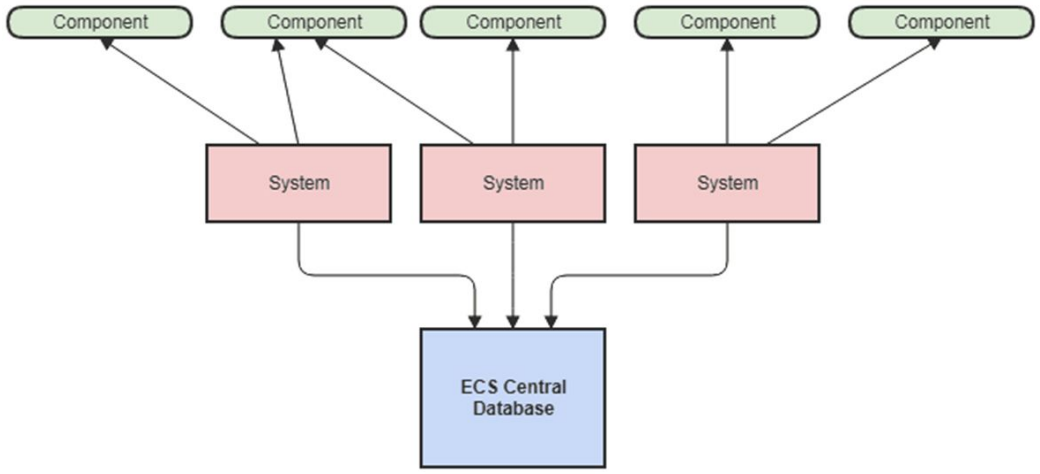


Components, Systems and behviour

In OOP model, behaviour is coded into **Object**

In Entity-Component model, behaviour is in **Component**

In ECS, behaviour is in **SYSTEM**

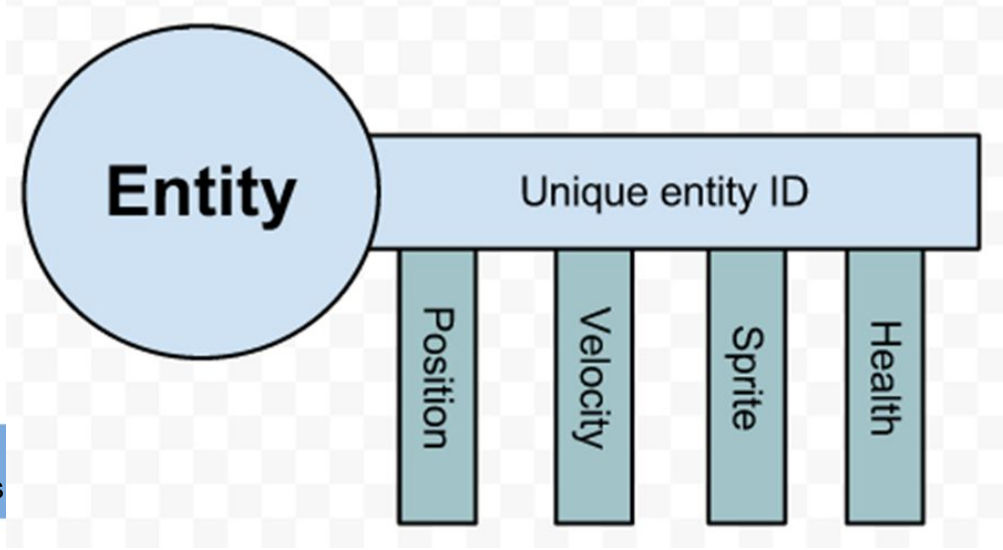


Entities are just IDs

An Entity is now essentially nothing more than a number.

Components, when they are created, are told which ID is their owner.

The Entity maintains some sort of reference to the components, but nothing else



Systems do all the work

We **never call update() on entities**.

We call update on *Systems*.

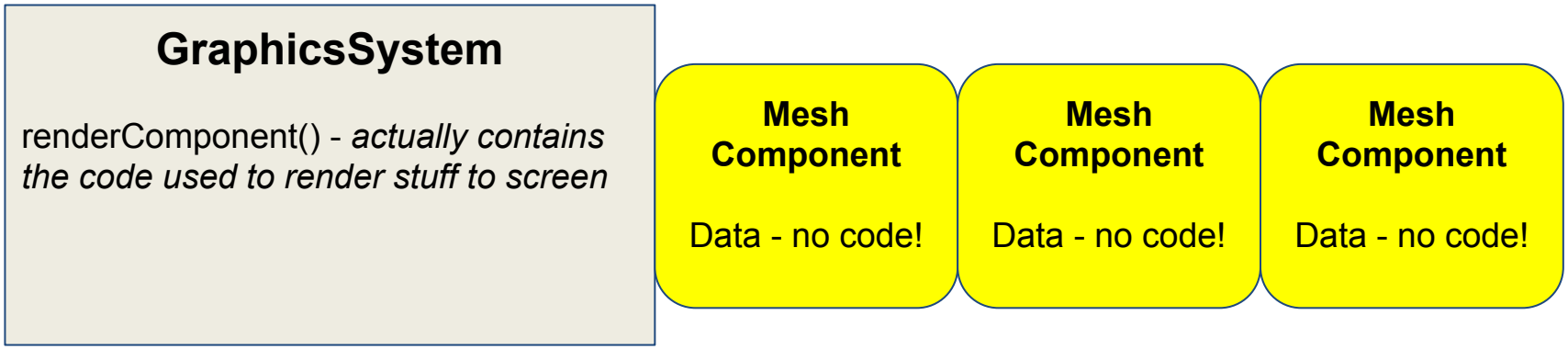
Systems access and update **only the components they need to.**

Systems example

e.g. GraphicsSystem has a renderComponent() function, which renders a ‘Mesh’ component

Component just stores data - no implementation code!

Mesh Components stored together so take advantage of cache



Solved problems:

~~ATOMIC DATA: loads of code has to be reused~~

~~MULTIPLE INHERITANCE: “deadly diamond of~~

~~DEPENDENCIES: must wait to update~~

~~INEFFICIENT MEMORY: load all information for each GO~~

Group work (2-4 students each group)

Based on what we've learned, use either pen-and-paper or a Visual Studio to create a C++ class structure for an Entity-Component-System engine.

It should contain

- a central 'Game' class
- some 'typical components'
- a way of storing components, grouped by type
- Systems - think of several a game might need
  - which components does each system access?
- logic to update each system
- suggestions for std containers to use

Engine

https://github.com/AlunAlun/MVD_01_Entities

Our engine base structure

OpenGL + GLFW for platform compatibility

A linear maths library

A Entity Component System that uses Templates, std::vector (which guarantees contiguous memory), and std::tuple)

A ‘Unity style’ code syntax for creating entities and components

Code work

Components.h

- fill in MeshComponent properties (vao and num_tris)

Game.cpp

- Use functions in ECS to create entity and assign it a Mesh Component.
- Assign properties of mesh component (vao and num_tris) of created plane

General

- Reorganise code so Graphicssystem does rendering, not Game