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
- main course deliverable: AI implementation

3) Prepare for 'Advanced Graphics I & II'

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

Course overview structure

Basic Graphics Revision

Data-driven engine design

Meshes, Materials and Geometry

Project - OBJ importer

Camera and Light (matrices revision)

Input System and Debug drawing (OpenGL buffer revision)

Quarternions, loading levels

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**

Advanced Graphics II

Animation timeline, fixed animations

Drawing and moving bones

Mesh skinning

Morph target Animation

Final project - putting it all together

Particles, Transform feedback


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

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

Resources

Output

Logic

Hardware & OS Interop

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

Mesh

Camera

Light

Audio Source

Environment

Particle Emitter

GUI Element

Trigger

Object properties

Every object should have:

- a position
- a rotation
- a scale
- a name

$\left.\begin{array}{l}\text{a position}\\ \text{a rotation}\\ \text{a scale}\end{array}\right\} \longrightarrow$ **Model matrix**

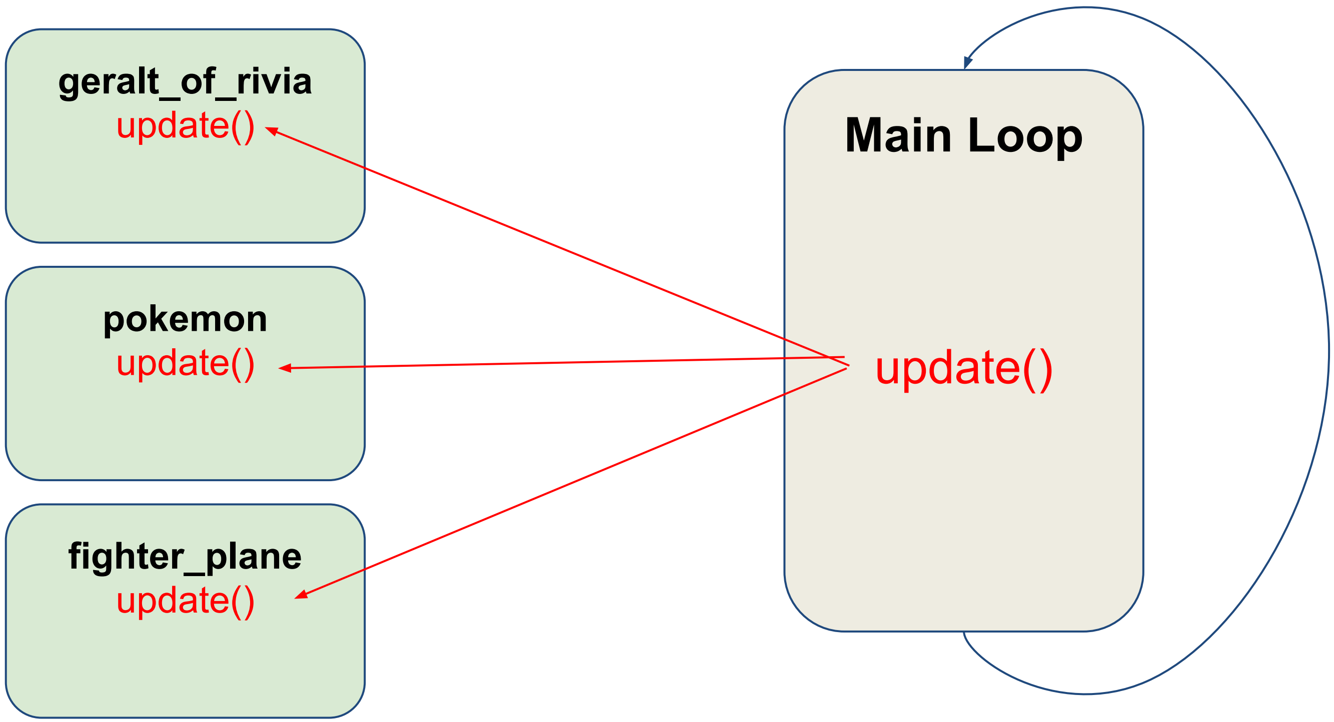
and possibly:

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

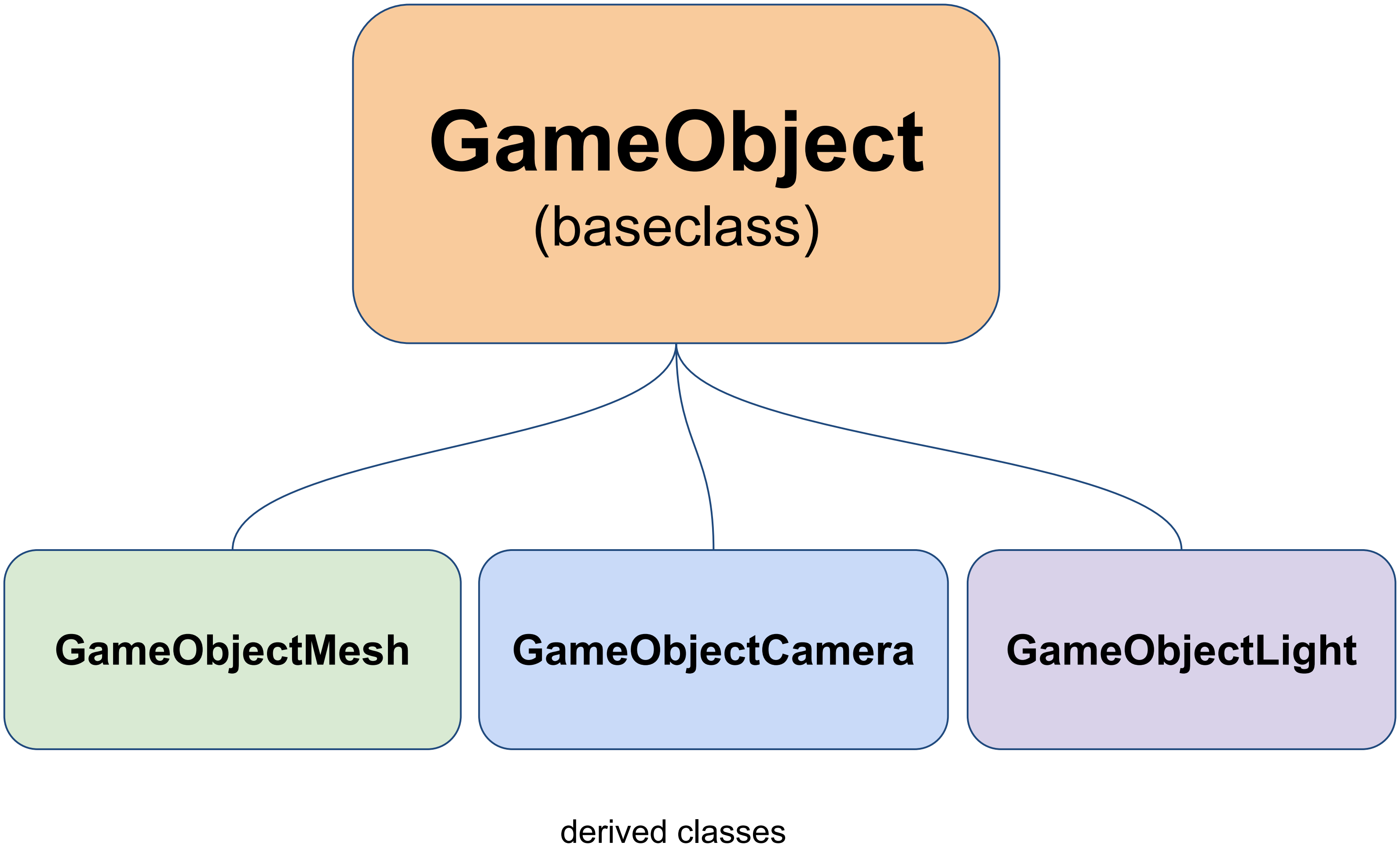
Alun Evans – alunthomas.evans@salle.url.edu

Updating obejcts

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



Mapping this to code, using inheritance



Classic Model

LOGICAL: every game object looks after itself

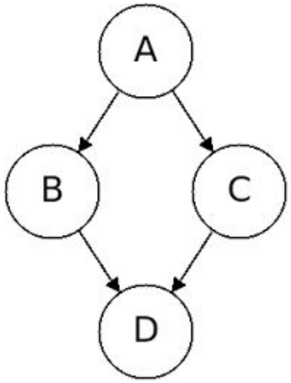
EASY TO UNDERSTAND: and easy to code

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”

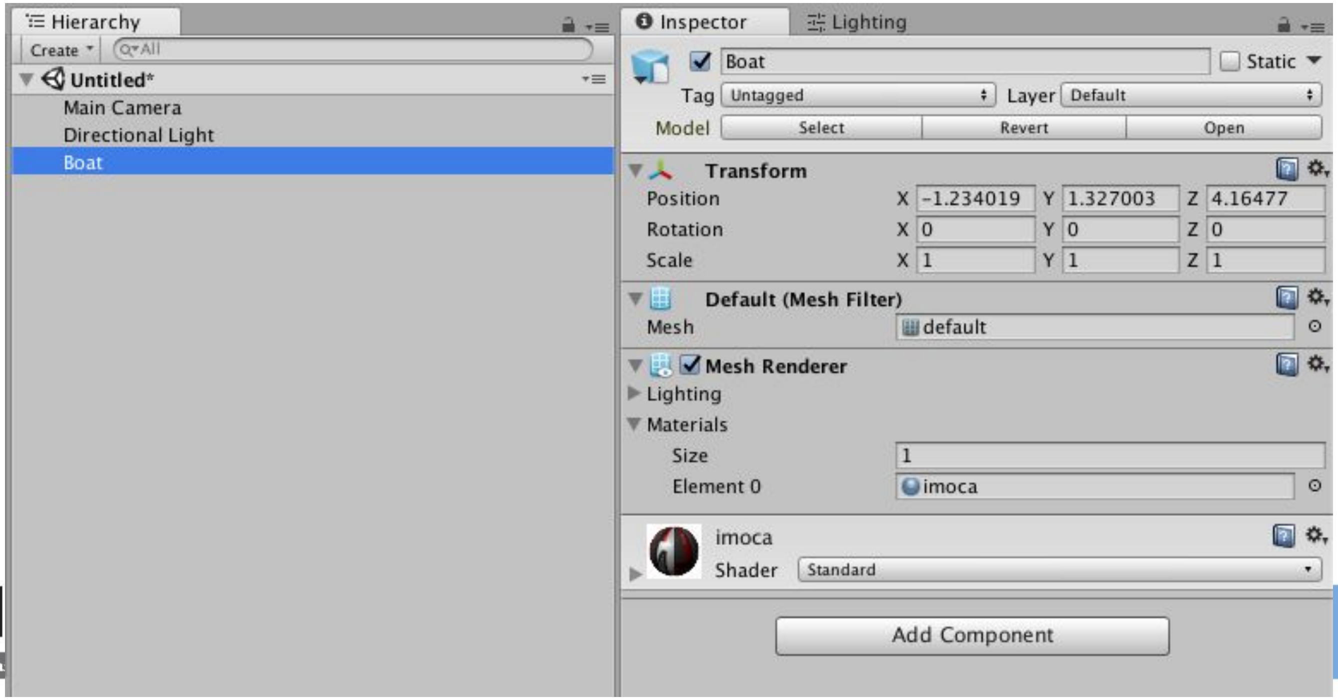


DEPENDENCIES: must wait to update

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

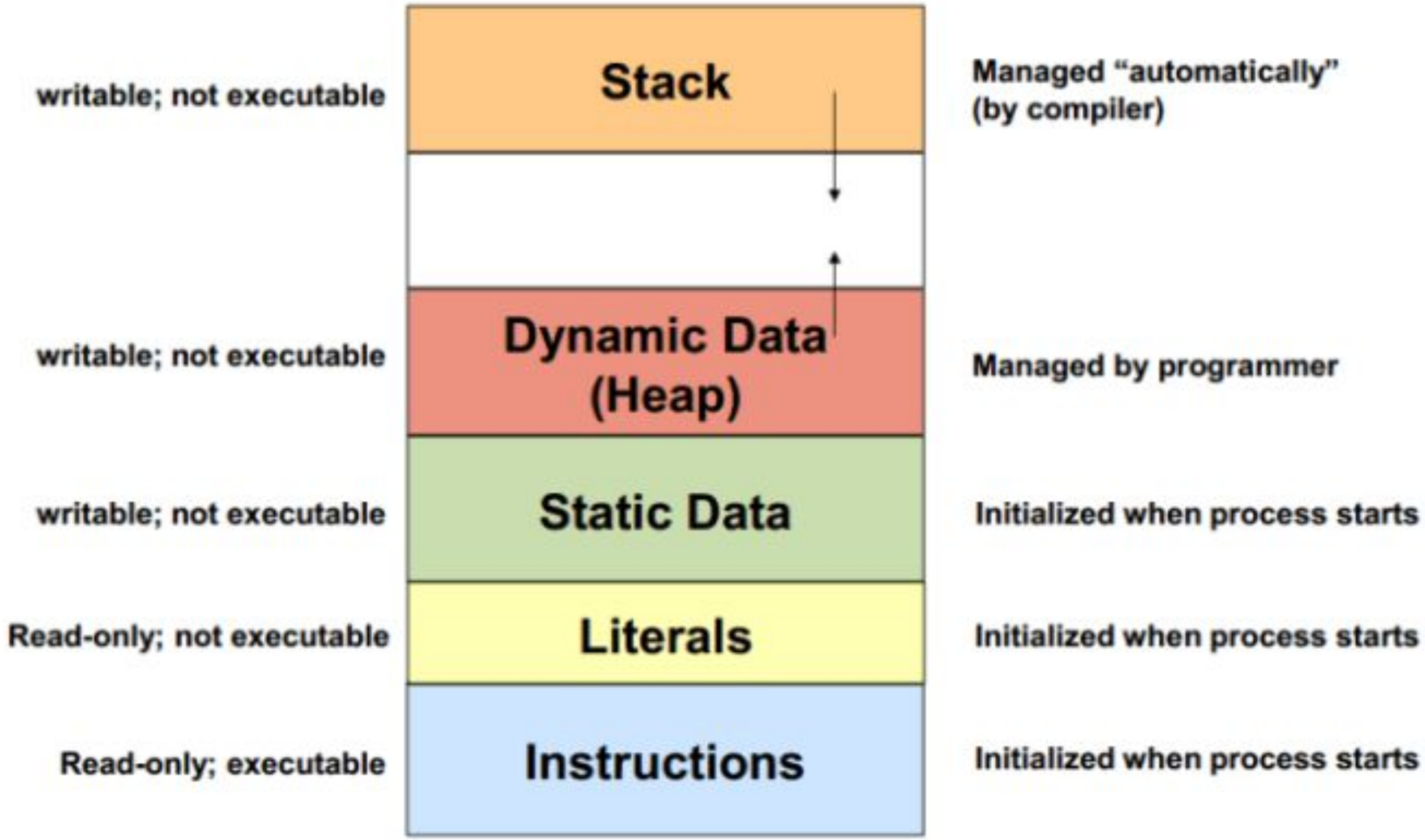
INEFFICIENT MEMORY: load all information for each GO

A break to talk about resources

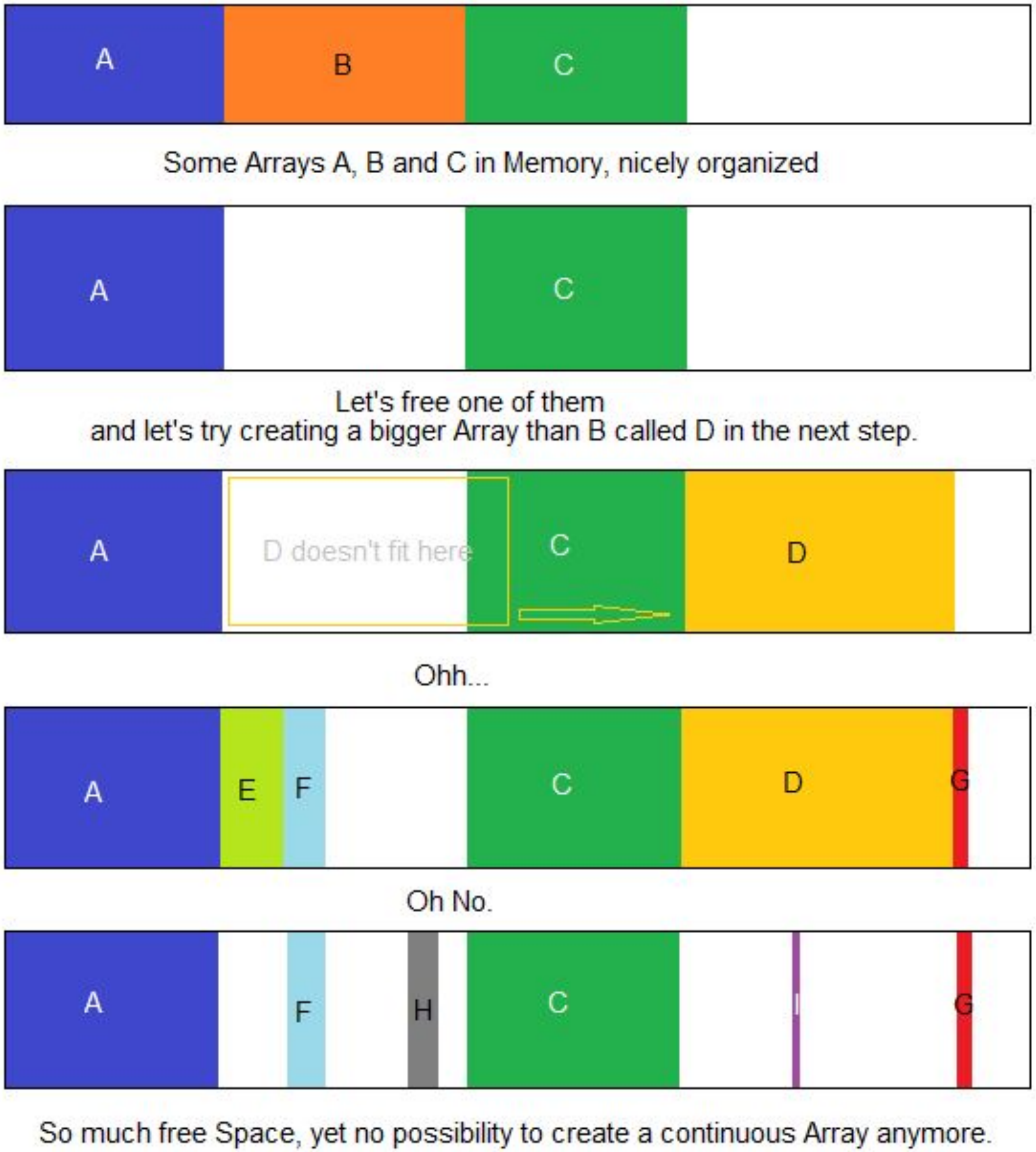
`Each component` contains pointers to mesh data, material data, textures

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

Resources: need to be loaded into memory



Resource memory allocation



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?

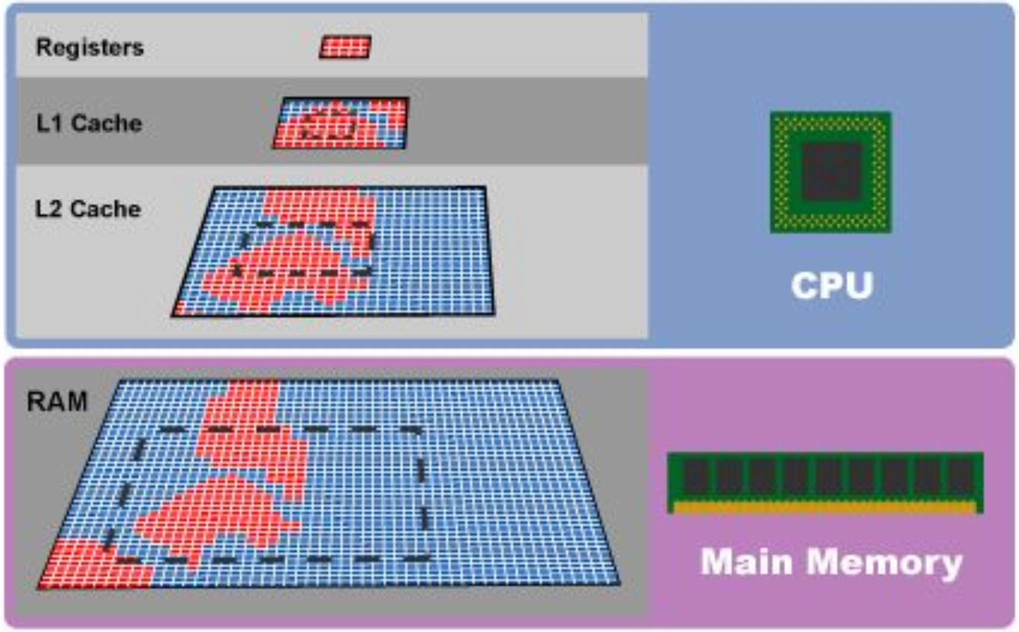
CPU REGISTERS

L1 CACHE

L2 CACHE

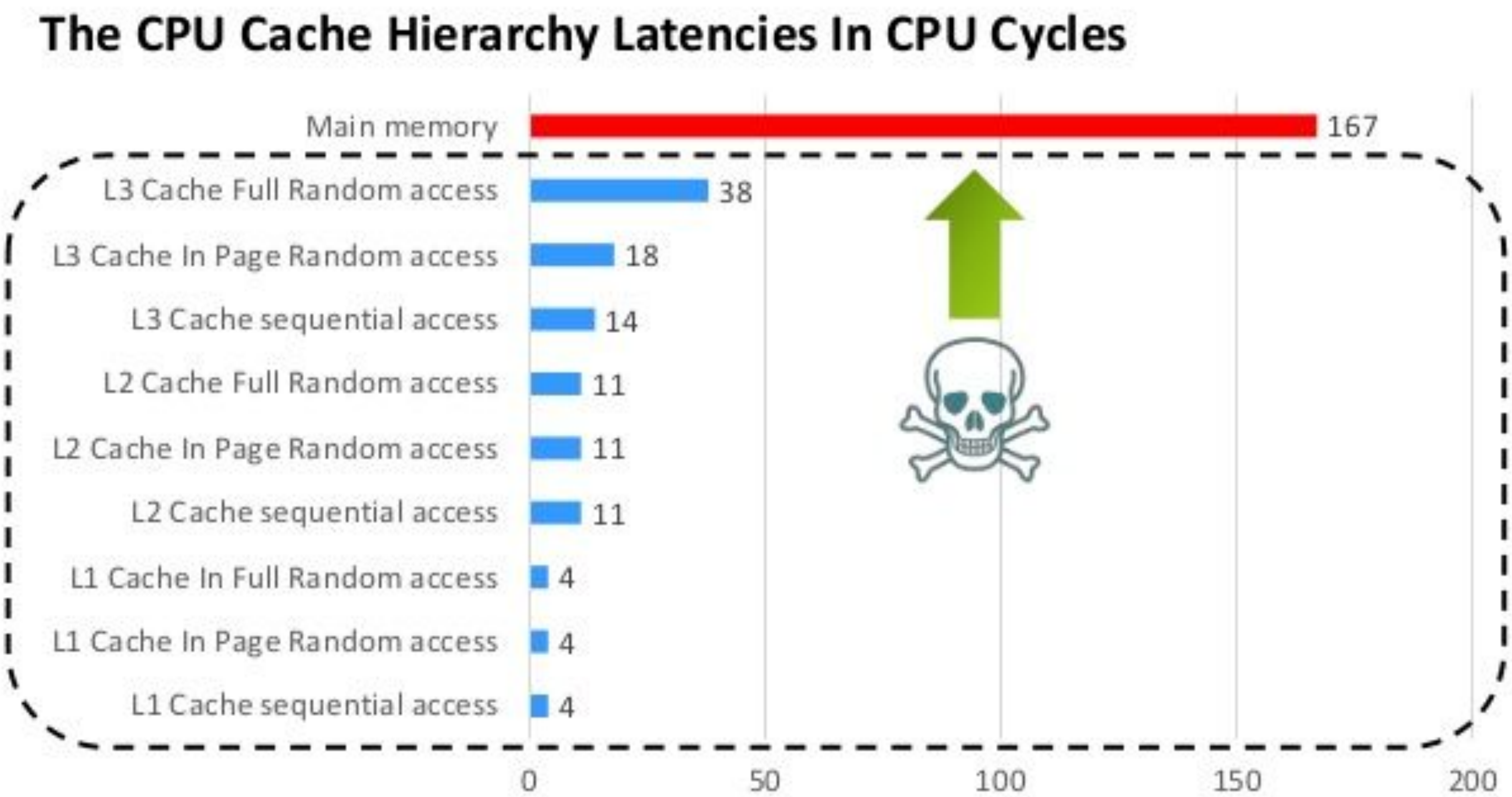
(L3 CACHE)

RAM



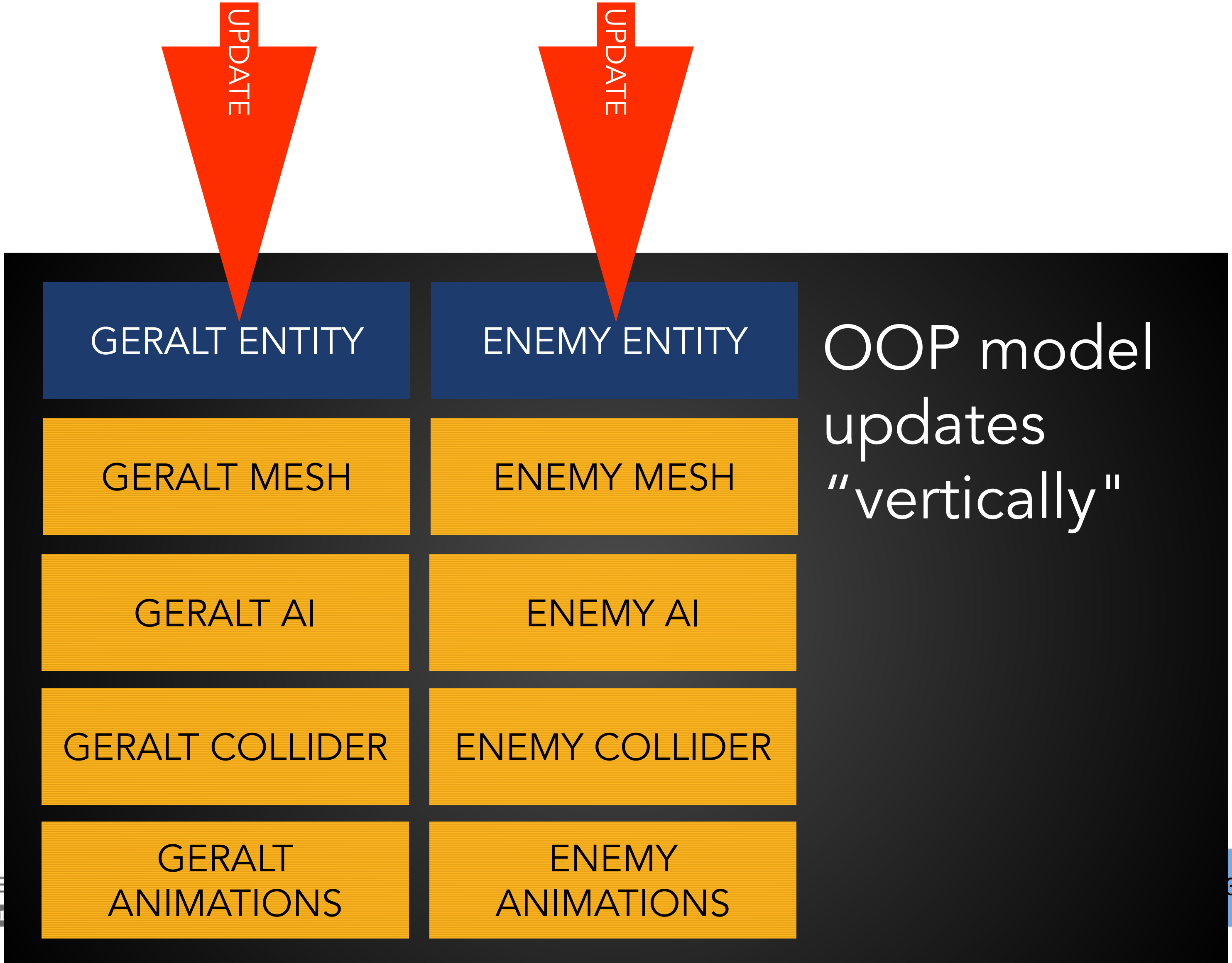
HARD-DISK SWAP FILE

Data Access time



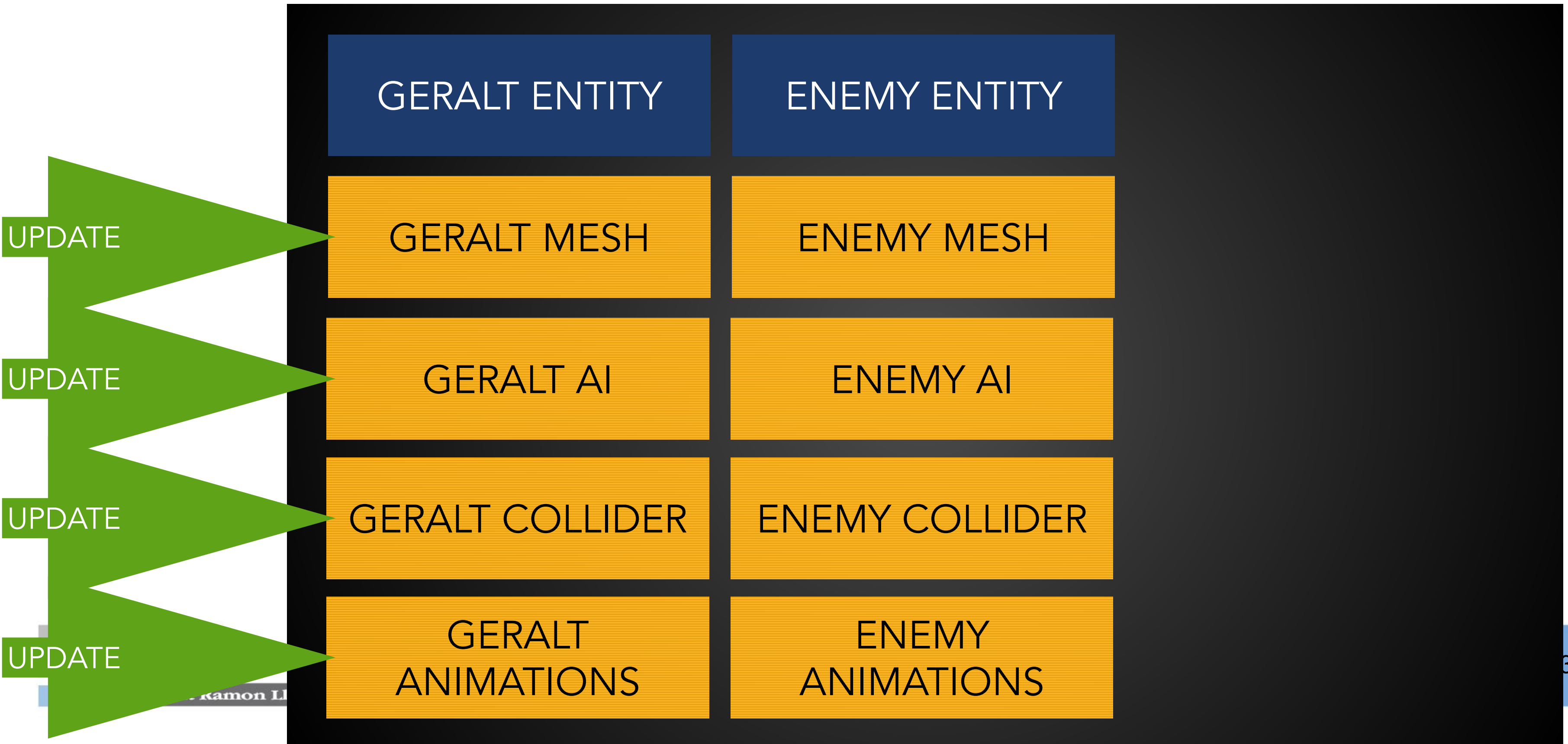
HDD SWAP FILE = OFF THE CHART!

ENTITIES HAVE LOTS OF DIFFERENT COMPONENTS



WHY DON'T WE….

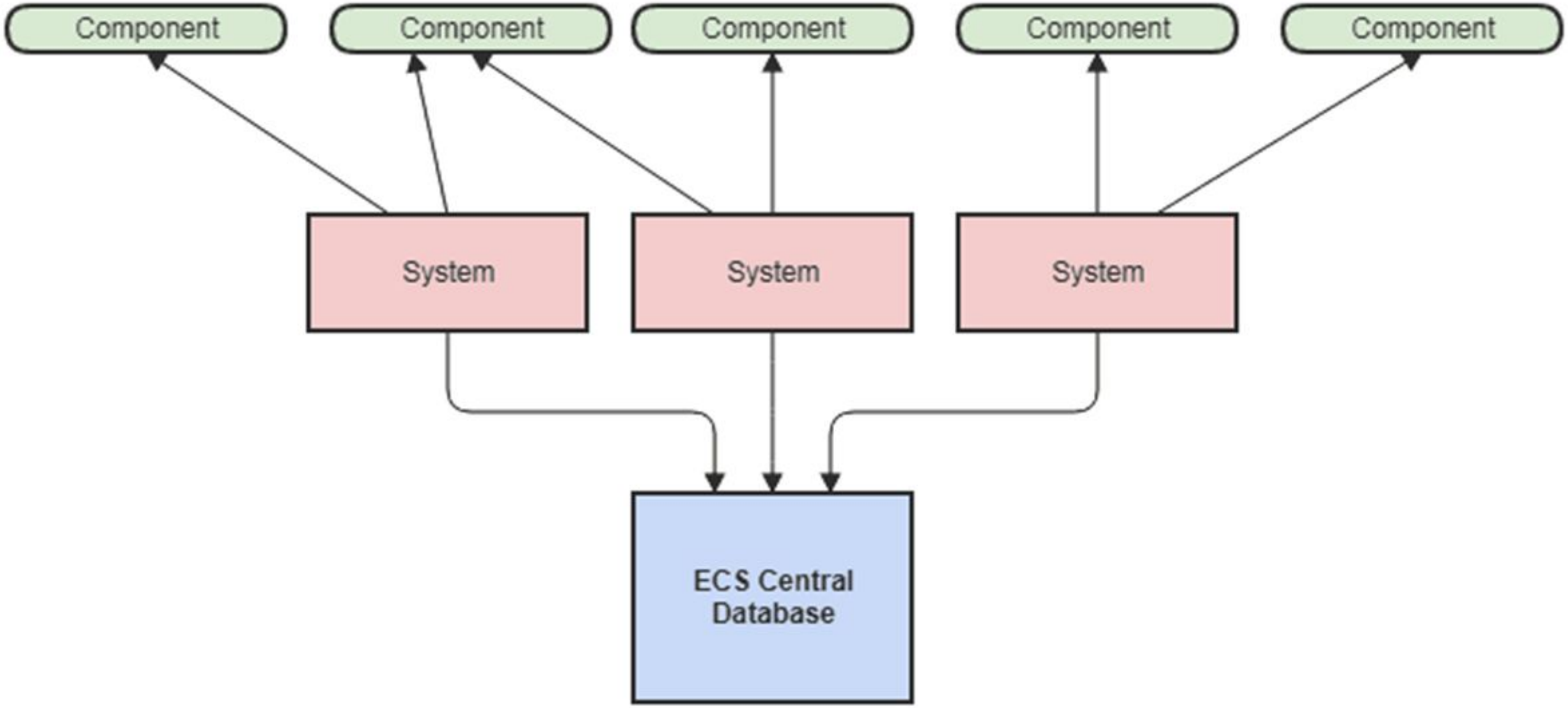
Update
"horizontally"?



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.*

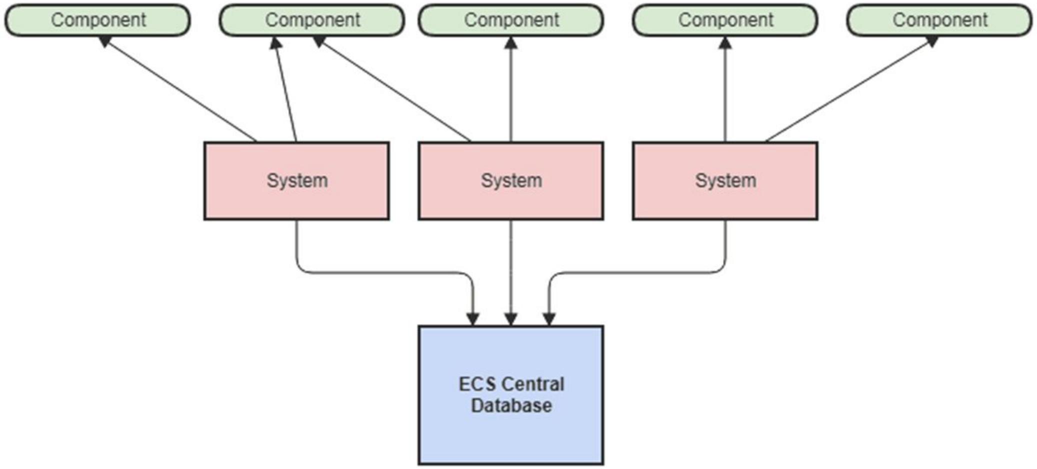


Components, Systems and behviour

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

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

In ECS, behaviour is in **SYSTEM**

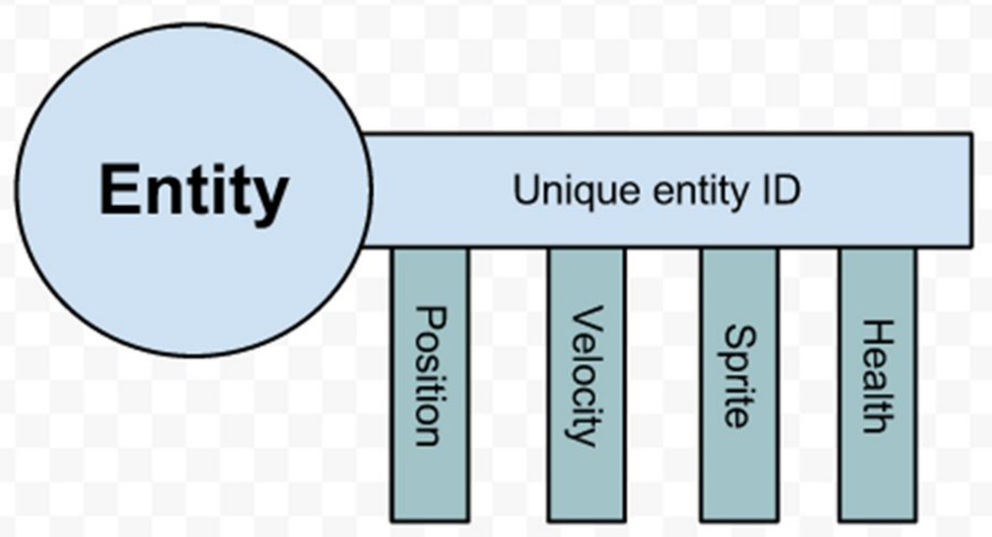


Entities are just IDs

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*.

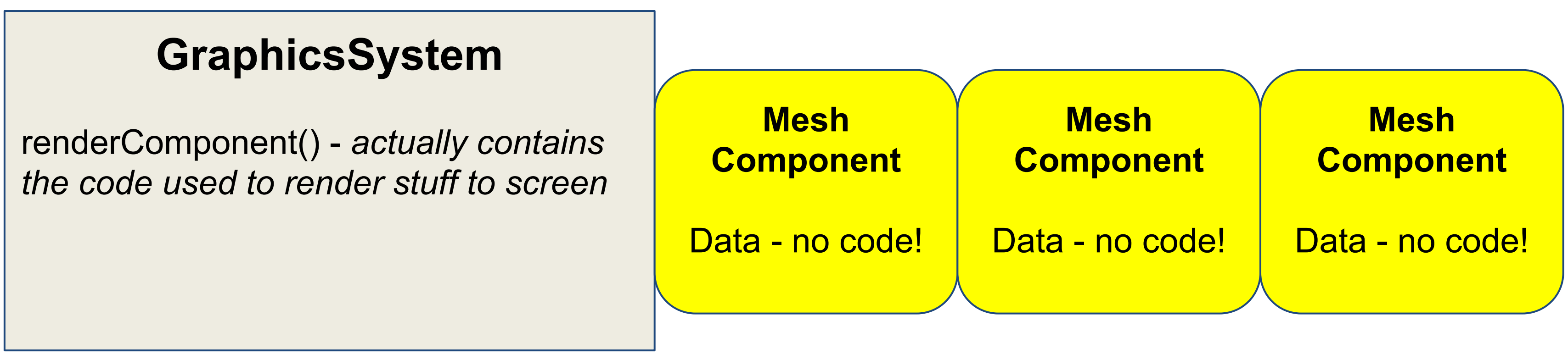
Systems access and update **only the components they need to.**

Systems example

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

Component just stores data - no implementation code!

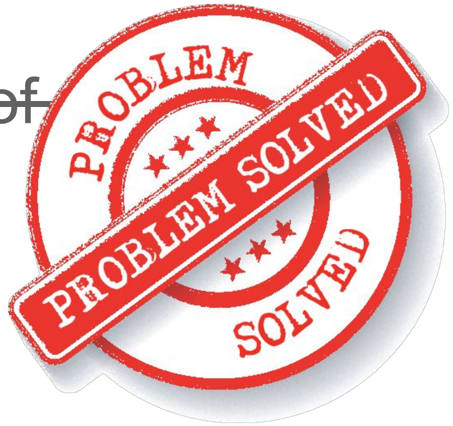
Mesh Components stored together so take advantage of cache



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 (2-3 students each group)

Based on what we've learned, use either pen-and-paper psuedocode to create a 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?
- what should each system actually update?
- 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

Creating a an ECS

A component base class only has a reference to its owner entity

```
struct Component {
    int owner;
};
```

We can extend the component base class with further components. The transform component inherits from Component and mat4:

```
struct Transform : public Component, public lm::mat4 {
    int parent = -1;
    lm::mat4 getGlobalMatrix(std::vector<Transform>& transforms) {
        if (parent != - 1){
            return transforms.at(parent).getGlobalMatrix(transforms) * *this;
        }
        else return *this;
    }
};
```

laSal Universitat R

Storing components

We an an “array of arrays” to store our components

Component Array

0	Transform Comp	0	1	2
1	Mesh Comp	0	1	
2	Light Comp	0		

Entity Class

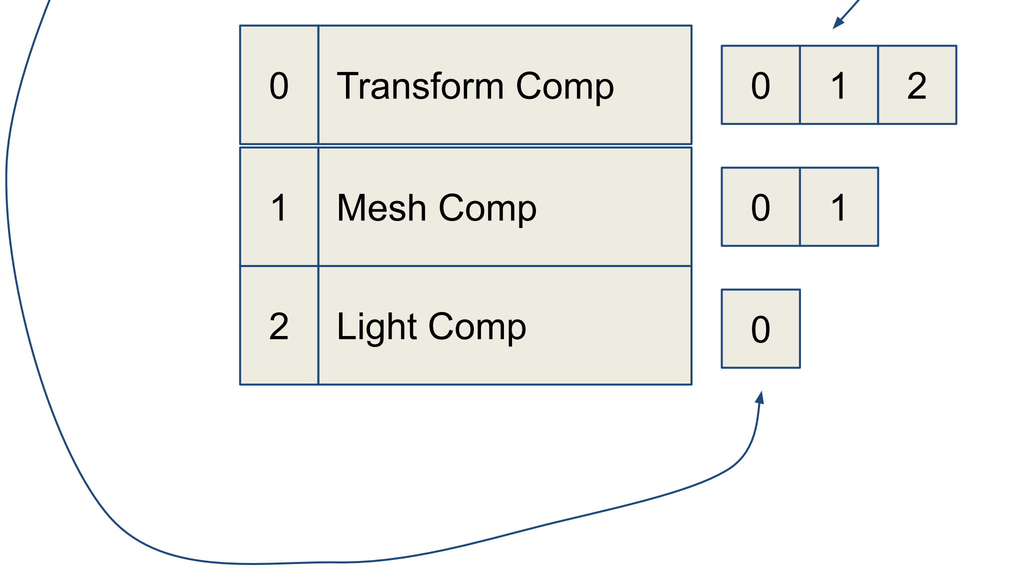
```
const int NUM_TYPE_COMPONENTS = 3; // transform, mesh, light

/*** ENTITY ****/

struct Entity {
    //name is used to store entity
    std::string name;
    //array of handles into ECM component arrays
    int components[NUM_TYPE_COMPONENTS];
```

Our entity class references the component storage

0	Transform Comp	0	1	2
1	Mesh Comp	0	1	
2	Light Comp	0		

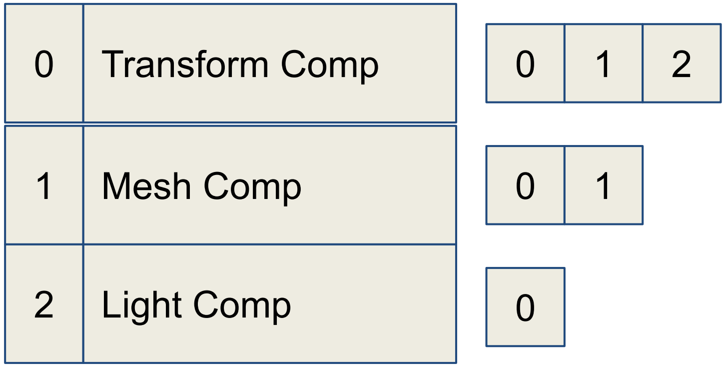
E.g. an entity with a light component

```
Entity some_light;
some_light.components[0] = 1; // Transform comp at index one
some_light.components[1] = -1; // No mesh component!
some_light.component[2] = 0; // Light component at index 0
```



Component storage - array of arrays

Difficult in C++ because of *content* of of each array will be different.



Could solve it using pointers a casting but its an ugly hack and we trying to avoid pointers

So we use a new feature in C++14 called the *std::tuple*

Standard Template Library (STL) revision

C++ defines a series of standard templates containers which give ‘high-level’ programming capability.

they are always accessed by using the ‘standard’ namespace - std::

std::vector is a resizeable array

std::map lets you map any data type to another

STL revision

They are called ‘template’ classes because they are defined as templates that can be used for any data type.

When you want to use one, you MUST state what data type it will contain e.g.

```
std::vector<float> some_float_array;

std::map<std::string, int> some_map;
```

Top tip: you can apply templates to *any* class in C++. They are very useful.

Google “C++ template tutorial” for more info.

Alun Evans – alunthomas.evans@salle.url.edu

STL vector

A resizeable array

push_back() - creates a *copy* of object in vector

emplace_back() - creates a *new instance* of object

STL includes

To use STL templates you need to #include them

#include <vector>

#include <map>

they still all under std:: namespace

The std::tuple

A new addition to STL in C++14 (#include <tuple>)

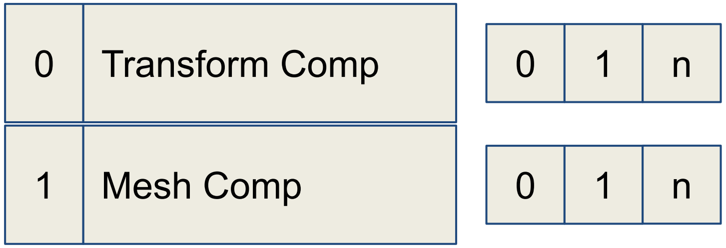
A tuple is an object that groups different variables together *and permit different types*.

```
std::tuple<int,char> foo (10,'x');
```

In old C++ this is not possible!

Component storage - tuple of vectors

```
std::tuple< std::vector<Transform>,
            std::vector<Mesh>
            > component_arrays;
```



Access content of tuple using std::get

```
std::vector<Transform>& the_transform_array = std::get<std::vector<Transform>>(component_arrays);
```

& - reference variables

remember the difference between & and *?

<u>& = reference to the memory of a variable</u>

If you don't see it, assume the variable is a copy!!

<u>* = pointer OR deference</u>

```
int foo = 25;
int* bar = &foo; //memory address of foo
int pepe = *bar; //pepe = 25
```

Getting tuple content - std::get

std::get return us the value of the tuple that corresponds to the type

```
using namespace std;

vector<Transform>& trans_array = get<vector<Transform>(components_array);
```

We will use templates later on to get in a clever way

STL and references

STL containers almost always return references

```
std::vector<Transform>& the_transform_array = std::get<std::vector<Transform>>(component_arrays);
```

You need to be careful to declare variables as references, so you can modify them

typedef

Another feature of C /C++ is the ability to **define new variable types**

```
typedef std::vector<std::string> myArrType;
myArrType foo;
foo.push_back("Alun is great");
```

Typedef our component array

```
typedef std::tuple<
std::vector<Transform>,
std::vector<Mesh>
> ComponentArrays;
```

Now instead of typing std:tuple<std:vector<Transform>, etc every time, we just type “ComponentArrays”

Easy!

So far…

Components defined

Typedeffed ComponentArrays

Entity defined

Entity Component Store (ECS)

The ECS is going to be THE place which we refer to from everywhere.

It stores the entities of our game, and the array-of-arrays with our components

```
//the entity component manager is a global struct that contains an array of
//all the entities, and an array to store each of the component types
struct EntityComponentStore {

    //vector of all entities
    vector<Entity> entities;

    ComponentArrays components; // defined at bottom of Components.h
```

ECS core functions

createEntity(string name)

createComponent<T>()

createComponentForEntity(int ent_id)

getComponentFromEntity(int ent_id)

getComponentID(int ent_id)

getAllComponents<T>()

C++ template functions

We know that STL library uses templates to store different variables

Well we can do the same with functions!

```
//creates a new component with no entity parent
template<typename T>
int createComponent(){
    // get reference to vector
    vector<T>& the_vec = get<vector<T>>(components);
    // add a new object at back of vector
    the_vec.emplace_back();
    // return index of new object in vector
    return (int)the_vec->size() - 1;
}
```

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