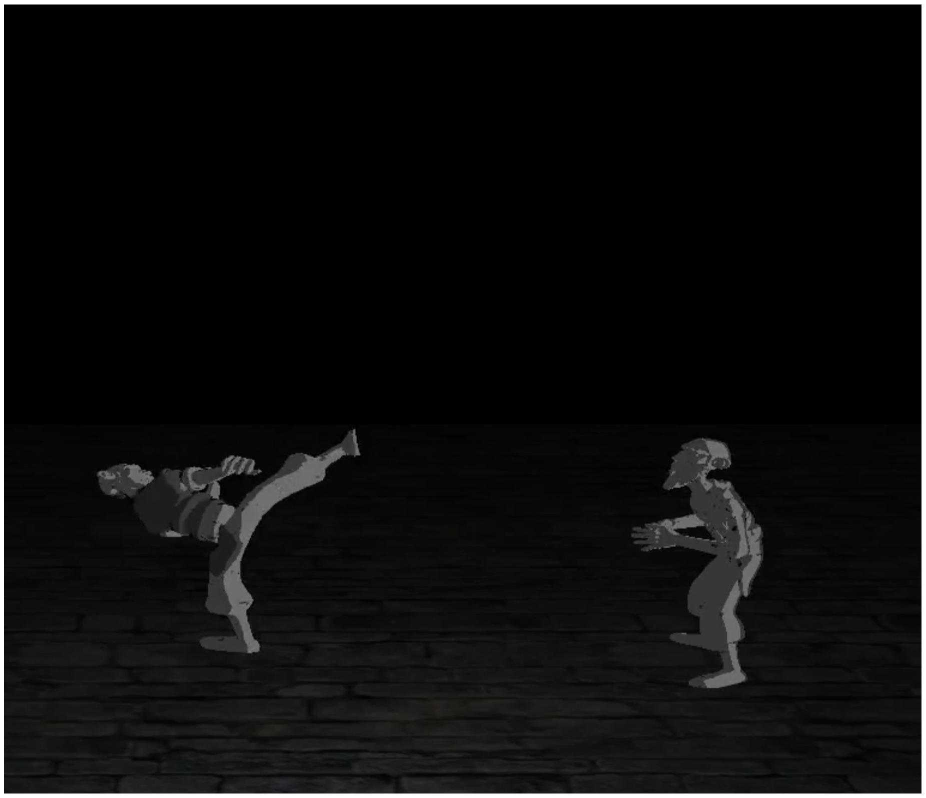
D7049E - Game Engine

Alexandre Chapin - achapin@insa-rennes.fr
Albernn Vedin - albved-7@student.ltu.se
Olof Bourghardt - olobou-7@student.ltu.se

https://github.com/olbo98/D7049E-Game-Engine

June 13, 2021



Contents

1 Introduction

The idea was to create a game engine that would contain all features necessary to be able to create 3D fighting games in the likes of Street Fighter, Mortal Kombat and Super Smahs Bros and explore how a game engine can be created and optimized with the help of a data-oriented design approach.

The engine was created in C++ together with the Ogre3D graphics library and the initial goal was for the engine to incorporate a multitude of systems including an animation system, entity component system, sound system and more.

2 Architecture

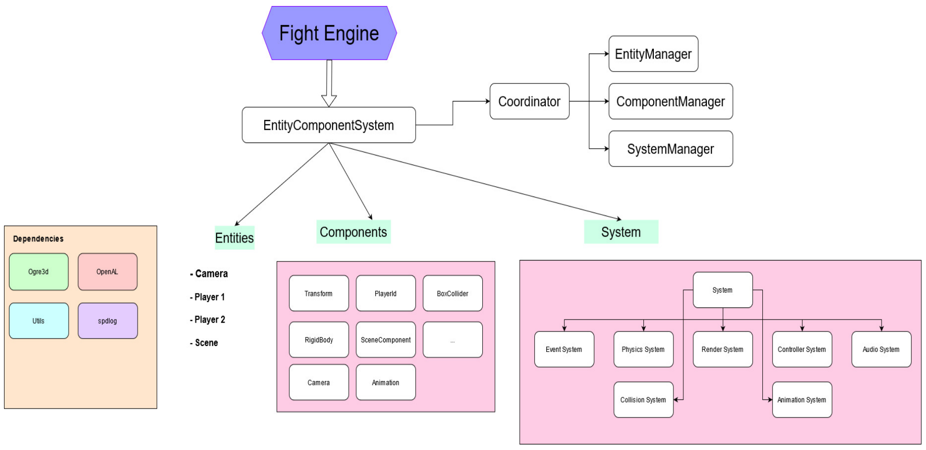


Figure 1: Game Architecture

Figure 1 shows the game engines architecture which has an entity component system consisting of coordinator that acts as a mediator between the entity manager, component manager and the system manager. These manager holds different entities, components as well as different systems that are required to make the engine work. Further the engine consists of some dependencies such as Ogre3d which is used as the render framework, openAL as a library for the audio system, spdlog for the logging system and also some utilities for the event system. In the following sections, the most important part of the architecture will be presented. First the Entity Component System, followed by the controller and then the Animation system.

2.1 Entity Component System

For our project, we decided to follow a data-oriented design. To do so, the engine should follow an Entity Component System architecture. Several ways of creating this architecture were found online but it was finally decided to use the Austin Morlan's implementation [1]. This implementation seemed to be easy to apply and also very efficient.

Several important concepts should be first presented to understand this architecture. First of all, we will use Entities that only consists of a unique ID :

```
using Entity = std::uint32_t;
```

The second very important elements are Components, and they are also very simple. Components are concretly just C++ struct containing datas, for example, our Transform Component looks like this :

```
struct Transform {
    // A SceneNode contains a position and
    // an orientation into the Ogre Scene
    Ogre::SceneNode* node;
}
```

Like Entities, Components have a unique ID :

```
using ComponentType = std::uint8_t;
```

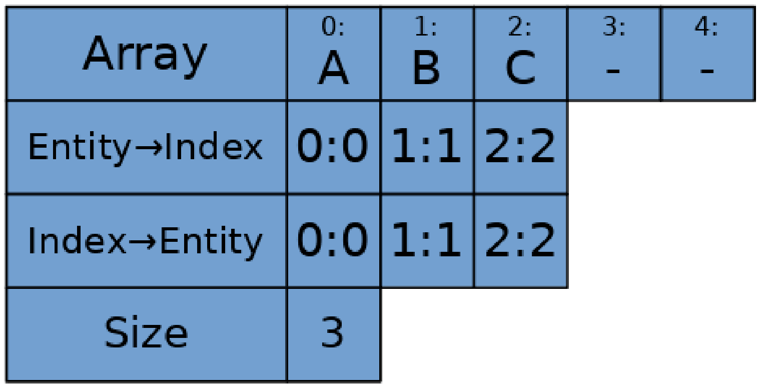
The third important concept for this implementation is the notion of Signature. Since an Entity is simply an ID, we need a way to track which components an Entity "has" but we also need a way to track which Components a System cares about. The approach chosen to do so is the usage of a **std::bitset**. Each Component has indeed a unique ID (starting from 0), which is used to represent a bit in the Signature. As an example, if Transform has type 0, RigidBody has type 1, and Gravity has type 2, an entity that "has" those three components would have a signature of 0b111 (bits 0, 1, and 2 are set). Here is what the Signature looks like :

```
using Signature = std::bitset<MAX_COMPONENTS>;
```

As presented at the beginning of the section, the Entity Component System architecture we use consists of a Coordinator that acts as a mediator between 3 classes : EntityManager, ComponentManager and SystemManager. Each one of these class will be now be quickly presented.

First, the EntityManager is in charge of giving Entity ID's and also to keep track of which IDs are in use or not. To do so, a **std::queue** is used : at the beginning it contains all the valid IDs, when an Entity is created it take an ID at the front of the queue; when an Entity is destroyed, it puts the ID at the back of the queue.

Before going through the ComponentManager, a specific structure used in the implementation should be presented : the ComponentArray, visible in 2. We indeed need to use a packed array (an array without holes inside) for each Component to contains its iterations. Since an entity is just an index, it's pretty straightforward to find its linked component. But a problem occurs when an entity is destroyed : its ID is no longer valid. Since we need to keep the data packed in memory (to go through the array without any ”if” statement) we need to be able to delete the Entity ID and its Component without letting any holes inside of the array. To do so, the structure use mapping from Entity IDs to array indices. Then, when one wants to access the array, it uses the Entity ID to look up the actual array index. When an Entity is destroyed, the last valid element in the array is taken and put into the deleted Entity’s spot and the map is updated to point to the correct position. There is also a map from the array index to an Entity ID so that when moving the last array element it knows which entity was using that index and can update its map. In order to better understand how this structure works you should look at the actual article about it that explains it with very good illustrations : [1]



Array	0: A	1: B	2: C	3: -	4: -
Entity→Index	0:0	1:1	2:2		
Index→Entity	0:0	1:1	2:2		
Size	3				

Figure 2: Example of a Component Array with 3 iterations of the same Component : A, B and C

Now that we have seen how the Component Array works, the Component Manager will be pretty straightforward to explain. It is in charge to talk to the different Component Arrays when a Component needs to be added or removed. As mention earlier, we need a unique ID for every type of Component so that it can have a bit in a Signature. In order to make it work correctly, the Component Manager has a ComponentType variable that increments by one with every Component type that is registered.

Finally, one of the most important element of the architecture is the System. A System represent any functionality that iterates upon a list of entities with a certain Signature of Components. Every Systems will then need a list of Entities to keep track of with which Entities it will interact with. Here is the code for the original System class :

```
class System
{
public:
        std::set<Entity> m_entities;
};
```

Each System we want to create has then to inherit from this class, this permits to have in the System Manager a list of pointers of System to keep track of all Systems. The System Manager is finally in charge of maintaining a record of registered systems and their Signatures.

2.2 Controller

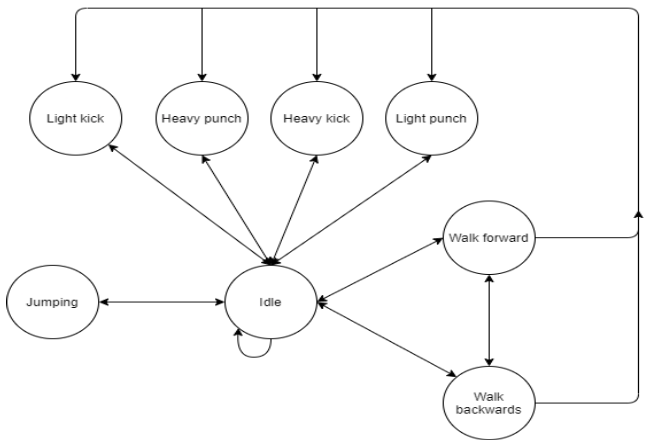


Figure 3: Player state machine

The controller is in charge of listening for different keyboard events and applying them to the player entities. When the controller receives an input it will examine the players current state to check whether the player is able to transition from the current state to the new state. There are currently eight states incorporated into the controller, walking forward, walking backwards, jumping, idle, light punch, heavy punch, light kick, heavy kick. If the state machine allows a state transition to be made the controller system will then apply any calculations necessary to the player entity such as moving the player in the right direction and send out a message that will eventually be handled by the animation system to apply a new animation.

2.3 Animation

The animation system handles all animation components in the game and updates them every frame. The animation components themselves only contains an Ogre AnimationState and can be changed to a new animation state when necessary. Currently the engine supports four different animations for movement including walking forward, walking backwards, jumping and standing still, and four different animations for various combat moves such as a light punch, heavy punch, light kick and a heavy kick.

To be able to handle the use of animations for the player characters and their different state the system needs to communicate with the controller system. The animation system receives information when to change the animation for a player entity from the controller system. The animation system will check if an animation that isn't looping has ended each frame and will notify the controller that a state change back to Idle has to be made.

3 Benchmarks

These are the results from the benchmark test using 2 different meshes to render. A total of 4 different test were conducted where each mesh was tested with and without the collision system. The tests concluded number of entities to render, delta time that shows the time period between the last frame and the current. The frame rate is basically the amount of frames that renders per second. The tests were conducted on a computer with the following configuration :

- RAM : 16Go
- CPU : Intel Core i7 10875H, 2.30Ghz
- GPU : Rtx 2070 Super with Max Q Design

The first tests are shown in table 1 and figure 1 where a conclusion of both the delta time and the frame rate is growing respectively decreasing exponentially when depending to the amount of entities rendered.

Entities	**Delta time (ms)**	**Frame rate (s)**
5000	2000 - 3000	0.34
1000	80-90	12
500	20	45 - 50
100	1	500 - 1000
10	0	∞

Table 1: Benchmark tests - Mesh 1 with Collision

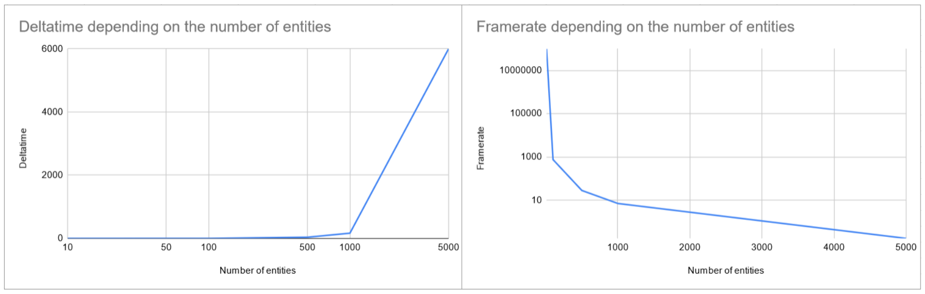


Figure 4: Graph from the results for Table 1

The second test conducted with a different mesh along with its animations and the same conclusion can be done with these results where the delta time and the frame rate grows and decreases exponentially.

Entities	**Delta time (ms)**	**Frame rate (s)**
5000	3000 - 4000	0.2 - 0.3
1000	80 - 100	8 - 10
500	18 - 20	47 - 50
100	1	1000 - ∞
10	0	∞

Table 2: Benchmark tests - Mesh 2 with Collision

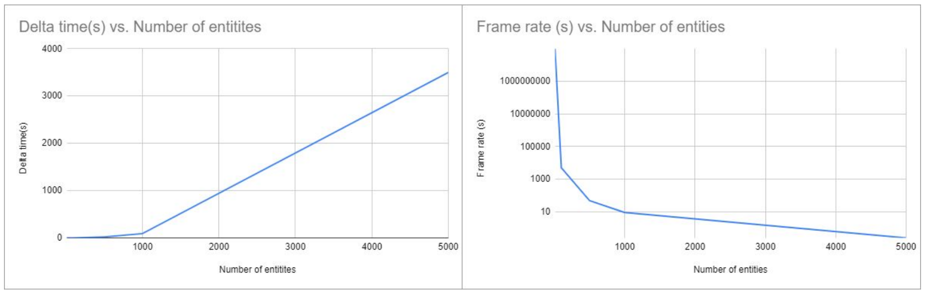


Figure 5: Graph from the results for Table 2

The tests also concluded without the collision for the mesh and the reason for that is the collision system iterates every entities twice checking if each entity collides with the other entities. This takes a lot of time and it was interesting to see how much better the program got.

The figures 6 and 7 essentially shows that it achieved much better results where the computer actually managed to render up to 60000 entities. However the same conclusion goes for the results where the delta time and the frame rate grows and decreases exponentially.

Entities	**Delta time (ms)**	**Frame rate (s)**
60000	50 - 60	15 - 17
5000	9	124 - 160
1000	1	1000
500	0	∞
100	0	∞
10	0	∞

Table 3: Benchmark tests - Mesh 1 without Collision

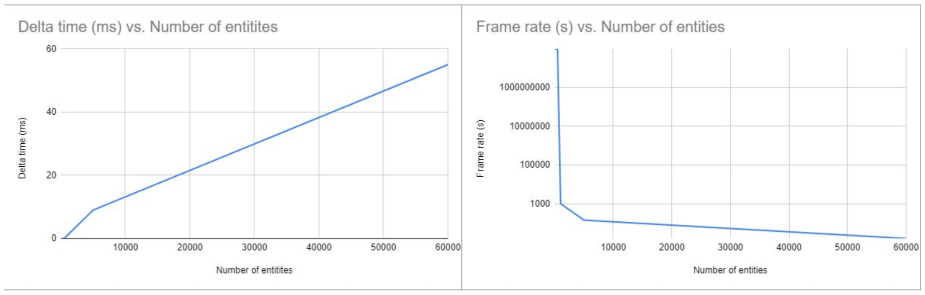


Figure 6: Graph from the results for Table 3

Entities	**Delta time (ms)**	**Frame rate (s)**
60000	50 - 60	15 - 17
5000	7	140 - 250
1000	1	1000 - ∞
500	0	∞
100	0	∞
10	0	∞

Table 4: Benchmark tests - Mesh 2 without Collision

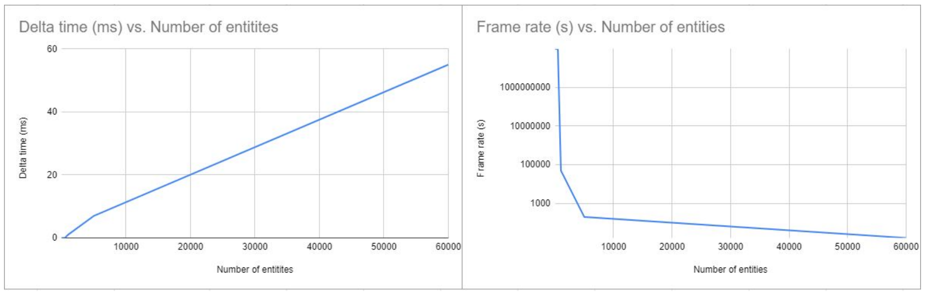


Figure 7: Graph from the results for Table 4

4 Optimizations

When developing a game, one traditionally use an inheritance approach for the implementation but it can cause problems. For instance having a Goblin inherit from a Monster which inherits from an Actor. One can also have a ShopKeeper that inherits from a Human which also inherits from an Actor. Moreover the Actor have a certain function, so for every Goblin you can call Goblin.function() and also call the function for every Builder, Builder.function(). However, this can cause some flexibility problems where it can disorganize the inheritance tree. For example in the ShopKeeper class where you have selling, bartending and so on, a Goblin can however not inherit from ShopKeeper because that would make a Goblin ShopKeeper Human.

Another problem is misusing the cache memory. In game development one usually iterates over objects multiple times and running methods every frame. For example a physic system can update position, velocity, acceleration and you would have a big object containing all this needs for the physics. Then you would call some sort of integrate function on every object that needs to be updated. This means that you would also have a function access to all its member variables meaning that if one only wants to access the position every other member variables is still pulled in the cache line which is a very big waste of cache.

This resulted that the group decided to implement an entity component system design as mentioned in section 2.1. So the traditional object is gone where we instead have an Entity which essentially is just an id. This id is then used an index into an array of different components where the components are just structs of data. Finally a system is just a logic that operates on the components.

Concerning our final project, after having run some benchmark tests we have seen that our Collision system needs a lot of resources. We've got indeed 10 times more frame per seconds without the Collision system than with it. We think that the problem is due to the fact that we go through all the entities 2 times to check if one entity collide with another. If we got N entities for example we got a complexity of $O(N^2)$ that is very huge ! We should then dig into this Collision system to find a way to optimize it.

Another system that could use some improvements is the Event System. Currently when a system receives a message it needs to take care of it immediately which could cause some overhead. All other systems have to wait for one system to finish processing their message before they get the chance to either read their messages or continue executing. Allowing the systems to instead choose when to process a message during execution would be much better. One could also implement some threads for parallel execution to allow systems to process their messages in parallel.

5 Reflections

Some things went right and some things went wrong, that goes with everything and the same goes for making a game engine. There were a lot of planning at the beginning and it was hard to come up with a plan on where to actually begin. Despite that the group manage to make a plan and bring up requirements for the engine. The group was pretty ambitious when it came to the objectives for the engine and it felt like everyone wanted a fighting engine that could make a game in the likes of street fighter. However things did not go quite that well and there were a lot of troubles along the way. First off, Ogre3ds' API was hard to navigate around, the documentation was absurdly bad and also a lot of it was outdated. Secondly the input system gave a lot of problems where it gave a delay when you wanted to hold down a button and releasing the key gave a lot of bugs. Furthermore the group spent a lot of time to use different libraries to integrate with the engine but it essentially just gave a lot of errors and ended up wasting a lot of time.

Those were a few of the problems that we got but as a conclusion the group can still be pretty happy about the results. We still managed to get an animation system working where it can change between different animation states among other things and make it at the end look like a fighting game.

However if the group had more time, we essentially wanted to implement all the requirements that we decided at the beginning, for instance making the characters hit each other where they lose health along with different sound effects for the animations. Moreover making a nice graphical interface for the characters name and health would be very convenient to have. Finally implementing different arenas where the characters can fight on as well as multiple rounds would be great as well.

Nevertheless, it was a fun experienced to implement an entire game engine where the group has learned a lot of new interesting concepts such as using a data-oriented design for the implementation, how the different systems communicate with each other and essentially how a game engine can be built.

References

[1] A Simple Entity Component System (ECS) [C++], Austin Morlan, `https://austinmorlan.com/posts/entity_component_system/`