Zaxxon

Tommy Ernsund
Game Engine Architecture, TDA572
A.A. 2020/21

1 Introduction

This paper describes the implementation of the game Zaxxon[8, 1] and the game engine based on the Entity-component-system (ECS) software architecture [7] according to [6]. The code from lab 4 was used as a starting point and then expanded upon.
Due to time constraints, a major focus had to be put on implementing features that I deemed necessary for the game engine to support, so much less time was being spent implementing the actual gameplay. As a result of this, the final game is a long way from fully representing the original from 1981, with several important parts missing.

2 Specification

In Zaxxon, the goal is for the player to get as high of a score before their ship is destroyed. I've described the game's key entities.

- **The player** controls the direction of a spaceship that is progressing towards the end level at a constant speed. Laser shots can be fired to destroy enemies and certain obstacles. Destroying an enemy will increase the player's score. The ship is destroyed after either losing all three lives or running out of fuel. Lives are lost by either getting hit by an enemy weapon or by crashing into an enemy/obstacle.
- **Enemies & Obstacles** are similar to each other in that the player will lose a life if colliding with either of them, but there are also some

key differences. Enemies can always be destroyed, but only certain obstacles can (i.e. a brick wall is indestructible). And of course, enemies will actively try to shoot the player.

- The **Laser** projectiles fired by the player will travel in the same direction as the ship is traveling at the moment. The laser will disappear after they have traveled a certain time, or when they hit an enemy/obstacle. Destroying an enemy with the laser will increase the player's score.

3 Overview

Each of the game entities in Zaxxon is essentially composed of a set of components, while their own state is kept relatively small. All of the components can access the entity's state, which for example holds its position.
Additionally, communication between different entities is implemented through messages. A message consists of a *MessageType* (for example if the player was hit or it is game over), and some *data* (what type of data varies depending on the MessageType).
While a component could be made that implement any type of behavior, this implementation just uses three basic types of components: *RenderComponent* to draw a game entity, *CollideComponent* detect whether it has collided with any other entity in a specified collection, and the *BehaviorComponent* which takes care of the game logic (i.e., updating the position of *Player* based on user input).
The game entities are in turn handled by the *Game* class, which after being created in *main()* and then updated in a loop, takes care of updating all enabled entities, checking if the lose condition has been met, among other things. To keep the game running at a constant speed, the time taken to render the previous frame is measured and used when rendering the succeeding one (this value is also used to calculate the average *FPS* if configured to).

4 Implementation

In this section I try to describe the different parts of the implemented game engine, focusing mainly on my own contributions.

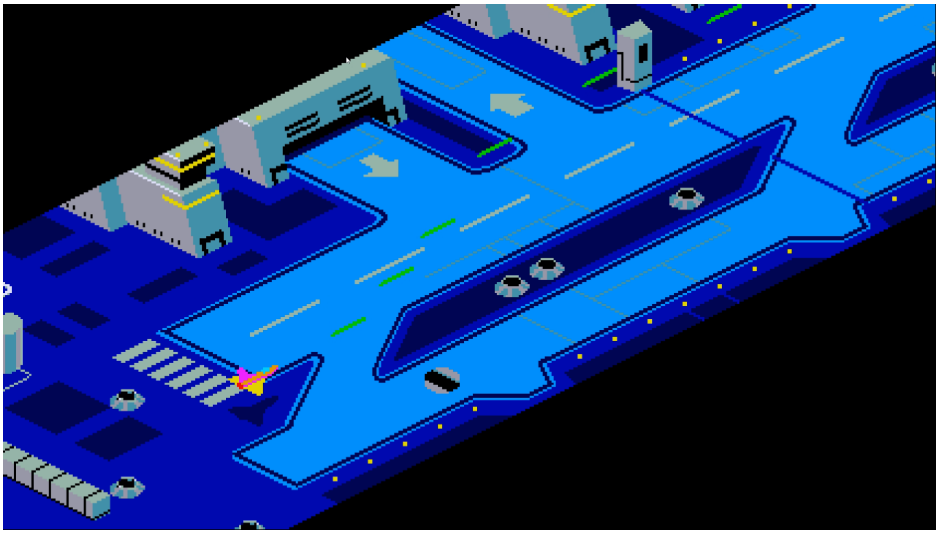

Figure 1: A screenshot from the implemented game

4.1 Object Pool

Object Pool is a type of collection with the purpose of pre-allocating storage for *Game Objects*. Then during the game loop, it can provide you with an object while avoiding memory operations that could potentially decrease performance. Zaxxonuses an *Object Pool* to store the *Enemy*, *Laser*, *Obstacle*, and *VFX* type objects.

4.2 Coordinates

As the game uses isometric projection, the *IsoVector* class which uses three dimensions was added. This is used by *GameObject* to handle the in-game logic, which then converts it to a Vector2D using the following method (based on [4]):

```
Vector2D to2D(double inc = 0.5)
{
        return Vector2D(x - y, (x + y) * inc - z);
}
```

This means that a GameObject has the following variables used for positions:

```
Vector2D position;
IsoVector isoPos;
```

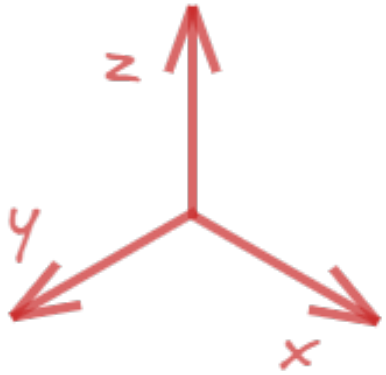

Figure 2: An isometric coordinate system

4.3 Component

Components are used to add functionality to *Game Objects*.

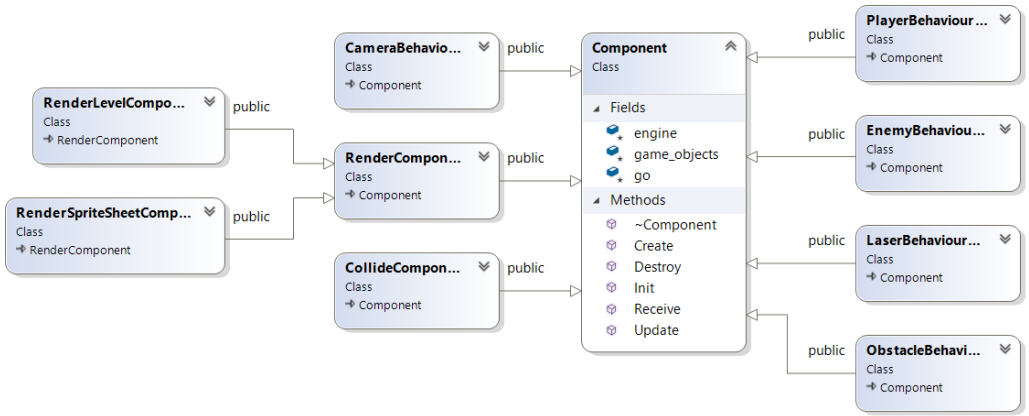

Figure 3: A class diagram of the *Component* class

4.3.1 Render Components

A lot of thought had to be put into the Render Components. While initially only having one type of Render Components, derived classes were added to better handle certain scenarios such as spritesheets and animations.

- ***RenderComponent*** will render the sprite specified in *Create(...)* at the 2D position of the *Game Object* it is part of. Although, if it is given a reference position it will use that as origin, and if *render_flat* is set to true it will use z-value from *isoPos* and subtract it from the position y-value (for example used when rendering the shadow of a spaceship).

```
Sprite* sprite;
const Vector2D* ref_pos;
bool render_flat;
```

- ***RenderSpritesheetComponent*** expands by adding support for spritesheets and animations. As all sprite in a sheet currently is the same size, select which sprite to draw is simply made using this:

```
void RenderSpriteSheetComponent::selectSprite(int index)
{
        clip.x = index * clip.w;
}
```

  Several methods was also added for animations:

```
void addAnimation(int id, const Animation& animation);
void addAnimation(int id, int* frame_sequence, int
   frame_num, double time_per_frame);
void startAnimation(int id, bool loop = false);
void stopAnimation();
bool isAnimationActive() { return is_animation_active; }
```

  To make it easier to support several different animations for a single *RenderSpritesheetComponent*, an *Animation* struct was also added.

- ***RenderLevelComponent*** is only used to render the level itself. It is similar to RenderSpritesheetComponent, but with an offset added. These values are used to the starting position to the lower-left corner of the level sprite.

With some work, *RenderSpritesheetComponent* could probably handle all rendering – the reason it does not is that support for spritesheets was added late in the project, and modifying the existing *RenderComponent* was too much work.

4.3.2 Collision Component

Due to the isometric coordinates system being used, collision detection is essentially in 3D. To make things simpler, the current implementation treats all objects as spheres instead of cubes when checking for collisions (the idea for this was found here [3]).

```
IsoVector go0_center = { go0->isoPos.x + go0_coll->size.x /
    2, go0->isoPos.y + go0_coll->size.y / 2, go0->isoPos.z +
    go0_coll->size.z / 2 };

// Calculate the distance between the objects and its
    diagonal length
double dist = go_center.distance(go0_center);
double go0_diag = go0->isoPos.distance(go0->isoPos + go0_coll
    ->size);

// Check if distance between them is less than their combined
     half-diagonal
if (dist < (go_diag + go0_diag) / 2)
{
    // Handle collision...
}
```

The Collision Component was remade somewhat compared to the lab; if for example, a Game Object wants to test collision against multiple other Game Objects it will still only have one Collision Component. Instead, a *std::vector* [**?**] is used to store all the Object Pools that it will check collisions against.

```
        std::vector<ObjectPool<GameObject>*>
            coll_objects_pools;
```

So instead of having multiple *Collision Components*, *AddCollObjects()* will be called twice (of course these objects will still be required to have a *Collision Component* themself).

```
void CollideComponent::AddCollObjects(ObjectPool<GameObject>*
    coll_objects)
{
        coll_objects_pools.push_back(coll_objects);
}
```

4.3.3 Behaviour Component

The *Behavior Component* contains the game logic for a *Game Object*. Apart from Player, the *Behavior Component* of most other entities basically only

updates their position. Player also uses it for firing the laser (which is dependant on the current energy level and time since the last shot), to change its sprite based on the ship's direction, etc.

4.4 Game Object

As mentioned earlier, a *Game Object* is mostly made up of a collection with an arbitrary number of Components that all can access the state of the Game Object. Additionally, they can communicate with each other by send and receive messages. As part of a *Game Objects* update, they will also update all of the *Components* in their collection.

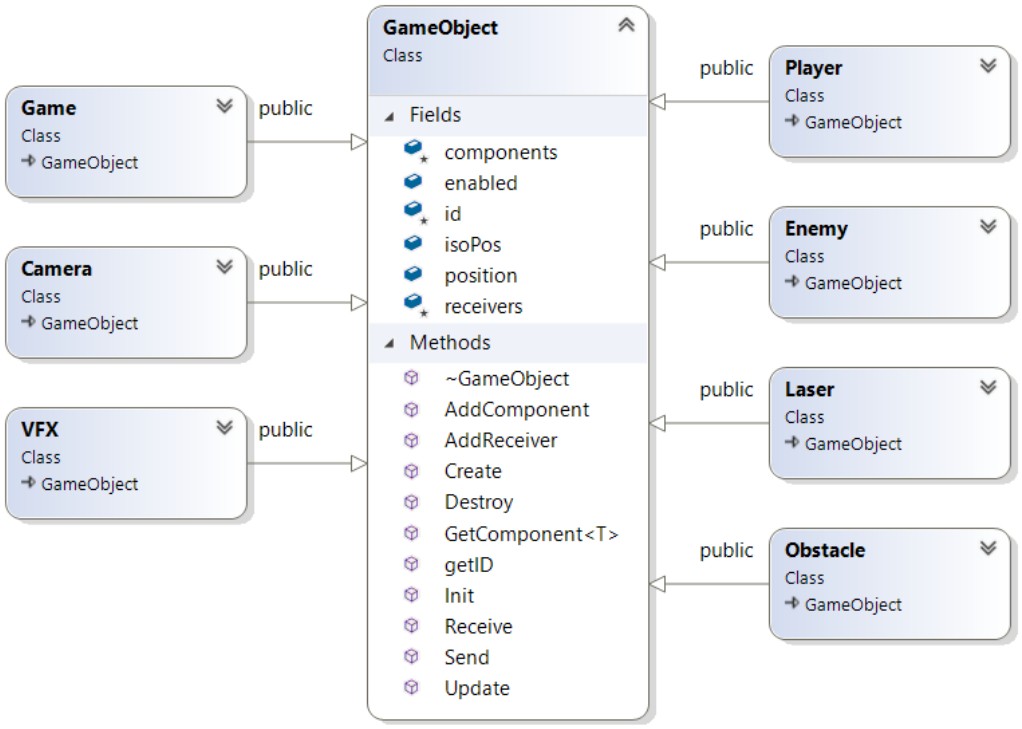

Figure 4: A class diagram of the *GameObject* class

4.5 Player

Player will move forward along the negative y-axis (according to Fig. 2) at a constant speed while the user only can control the direction it is going.
It has both a RenderComponent and a RenderSpritesheetComponent, the prior being used for its shadow.
Lastly, it has a Collision Component that will check for collisions with the Enemies and Obstacles.

4.6 Laser

Laser is for now only used by *Player*, but should be easy to use for *Enemies* too as *Init()* takes a position and direction. As it initializes, it will also start an animation (making the laser shorter and shorter to give a greater sense of how fast it travels).
LaserBehaviourComponent just updates the Game Objects position using the direction and will disable itself after having traveled a certain distance.

4.7 Enemy, Obstacle

Enemy and *Obstacle* are currently as there was not enough time to fully implement them. The main difference is that *Enemies* can be destroyed, while *Obstacles* can not.

4.8 Camera

The *Camera* object basically keeps track of the current position in the level, and also renders the level itself (which maybe it shouldn't). This position is used by the *RenderComponents* in some of the other *Game Objects* as a reference position to calculate the viewspace coordinates.

4.9 VFX

VFX was made for simple animation not related to any pre-existing *Game Object*, for example, explosions. Each object in the *VFX pool* only has a *Render Component* with all of the different Animations added. To show an animation, you just need to call Init with the animations ID, position, and if you want it to loop, as seen below.

```
void Init(int animation_id, IsoVector pos, bool
    loop_animation) {...}
```

This will basically just set the *Game Objects* position and use its *RenderComponent* to start to specified animation. The *Game Object* will be disabled when the animation is no longer active.

4.10 Game

It's the *Game* class where all other *Game Objects* are created along with the related *Components* etc. All created objects are inserted into a std::set [2]. As Game update, it will update objects in the set. It will also draw all the UI elements (score, player lives, etc), keep track of the player's score, and do things like playing sound effects.

4.11 AudioManager

The *AudioManager* class was made to handle playing both short sound effects and music. It's is created, initialized in the *Game* object. When loading sound files, each must be given a unique ID which is then later used to play a certain sound effect. If a *Game Object* wants to play something, they can just send the ID via a message with *PLAY_SOUND* as the *MessageType*.i

5 Conclusions

Initially, the biggest challenge implementing this game was the fact that it uses isometric projection. The use of an ECS architecture greatly simplified things, although its structure and implementation of the different entities and components could be substantially improved if not for a limited time. Then, of course, there are the more obvious things like finishing implementing the remaining parts of the game (enemies, obstacles, etc) and change to a proper collision detection algorithm.
In the end, I found that making something like a game engine, where the complexity has no bounds and endless improvements can be made be a good learning opportunity. Having no prior knowledge of game development other than the labs meant that I had to learn, or figure out, how to implement certain features during the course of the project. This also meant that as I

made progress, I continuously realized how certain features could be implemented in a better way. "My existing code, which I was happy with before, is now feeling too constricting." [5] nicely summarizes how I feel about the end result – but maybe that's a good thing.

References

[1] What's an Entity System? http://entity-systems.wikidot.com/, 30 Nov. 2014. Accessed: 2021-7-29.

[2] std::set. https://en.cppreference.com/w/cpp/container/set, 14 Jan. 2021. Accessed: 2021-7-29.

[3] 3Dave. What is the fastest algorithm to check if two cubes intersect (where the cubes are not axis aligned)? https://gamedev.stackexchange.com/questions/130408/what-is-the-fastest-algorithm-to-check-if-two-cubes-intersect-where-the-cubes- 24 Sept. 2016. Accessed: 2021-7-2.

[4] Acuña, F. Raising the ground - Fredy Acuña - Medium. https://medium.com/@fredhii/rising-the-ground-64957937513b, 15 May 2020. Accessed: 2021-8-19.

[5] Berardi, G. State of the Art Game Objects. https://www.gbgames.com/2010/10/27/state-of-the-art-game-objects/, 27 Oct. 2010. Accessed: 2021-8-13.

[6] Nystrom, R. *Game Programming Patterns*. Genever Benning, 2 Nov. 2014.

[7] Wikipedia contributors. Entity component system. https://en.wikipedia.org/w/index.php?title=Entity_component_system&oldid=1032677550, 8 July 2021. Accessed: 2021-8-01.

[8] Wikipedia contributors. Zaxxon. https://en.wikipedia.org/w/index.php?title=Zaxxon&oldid=1033592160, 14 July 2021. Accessed: 2021-07-20.