Advanced Programming: Cycle #3

By Team 6: Robin van Emmerloot~~, Laura Kester~~, Aart Odding and Dennis Vinke

Table of Contents

Project goal game:
For the project, we want to make a game inspired by Geometry Wars: Retro Evolved. Geometry Wars is a minimalistic space shooter with vector graphics. More about the game can be found here: https://store.steampowered.com/app/8400/Geometry_Wars_Retro_Evolved/ . We want to create a game in which we will try to recreate the feel of the game and add our own twist as well.

Design:
To realise the project we want to use an Entity Component System (ECS) to create a game were we could change the structure of game components on the fly without gaining a giant inheritance tree for every possible combination of special actions a game object can use. The challenge here is to divide all the different components in such a way that it could be used by multiple objects. Because we want to include a lot of small differentiations for the base game, having a fast and solid way to manipulate the game logic is a must. By switching different components on and off during runtime, it means we can change the general feel of a game mode without cluttering the code with a gigantic if-statement or switch structure. To realise an Entity Component System, we want to build a small scale game engine in which we will make our game in.

Entity Component System:
An entity is in a sense the same as an instantiated object from a class. The biggest difference between an entity and a object however is that the behaviour of a is defined by the class it is initialized from. With the ECS we abstract the logic from an object and put it in their own component. This way the entities and components become data, while the system interacts with the data to perform the logic of an entity. The behaviour of an entity is based on the components that are linked to it. In other words: the behaviour of an entity can be changed on runtime, while the behaviour of an object is fixed during this time.

Why we prefer the ECS approach over the a more traditional object oriented approach for making a game is best explained with an example. If you want to include a playership that can move, shoot and get hit, but also have an enemyship that can do the same. The most logical way to implement this in a traditional Object Oriented Programming way is to have a base class Playership which would contain the bare minimum of what a spaceship should contain. In the derived classes the class specific data like on how it handles input for the playership and the logic how to act for the enemy ships will be inserted. If we want to make an unbeatable enemy that does not get destroyed, there would be a lot if different ways to implement it in a OOP solution. One of those solutions could be to create a new enemy ship class derived from the enemy ship class that has this different kind of behaviour implemented. Another way to include the invincibility mode is by adding an extra boolean state and functions to set and call this state or make the hitpoints of an enemy extremely

high. Making the hitpoints large needs also needs update function to regain health, else it is technically possible to kill the enemy or else some undefined behaviour can take place.

If there are more enemies that can be invincible but not all, this would mean we have duplicate code in some of the functions or redundant code in others because we decide to implement the invincibility state to the base class, or because we decided to add the functionality only to the derived classes that need it. The hierarchy tree becomes even bigger if you have spaceships that could be influenced by gravity and ships that are not.

To solve this “flaw” the inheritance paradigm introduces to making a game, we decided to use the Entity Component System paradigm. Instead of having to deal with the above problem when we want to add invincibility to an object, we will not give a health component to the entity. If an entity needs to be able to be hit after a certain amount of time or hits, we could simply add the health component during runtime when it is needed. Logic is not simply being restricted to one branch within a giant gameobject hierarchy, but decoupled over different component pieces. We think this is a more elegant way to include different types of the same object type. We are not planning on using a fully implemented ECS but we will try to incorporate the principles of ECS in a OOP way. One of the ways we think we can manage this is by having the entities also manage their own components. An entity in a ECS would have their own update function while also maintaining if they do have a certain component attached to them. This is more than an entity in a ECS would do. This way we take the best of both worlds as our data is bundled in one object that can be created and destructed when needed while also having the data behaviour linked to data. A downside of this design is the data dependability we create for components to work in our system.

Requirements:

The game functions as follows:

The player sees a 2D world from a top-down view on which it can move around a ship.
A player can move the ship in the four most basic directions, namely up, down, left and right. It is also possible to combine movement in both horizontal and vertical axis, to make it possible to move diagonally. The player can move till they hit a level boundary. Once the boundary is hit, the ship will bounce back into the field so it can move on.

When the shoot button is pressed, the ship the player is controlling will shoot from the front of the ship. The player will be able to pick up power-ups to change the way the shoot action is being visualized on the screen. For instance, it should be possible to shoot two bullets instead of one when the specific power up-for this enhancement is picked up. In order to pick up a power-up, the player moves his or her ship over the power-up.

On the screen there will also be ships that cannot be controlled by the player. These ships are enemies. The enemies can move just like the player, but do not require any input from a human. The input will be done by the system. The enemy ships can also shoot projectiles. All the projectiles (bullets) in the game will disappear after 5 seconds after they are created. The bullets will also disappear once they hit the boundary of the level or one of the ships on the field. If the ship the player controls is hit, the player will lose a live. The player loses if all their lives are used up. The player needs to kill all the enemies to win the game.

Use Case Diagram:

As we want to use an Entity Component System, the different kind of game objects that should be in the game can be represented as an entity that knows how to behave with the help of its components. This way the system (in the form of an AI) could also be represented as an actor. For simplicity sake we decided to only show how the player acts with the system. A generalisation could be used for the Player actor to show the AI could use most, if not all of the actions a player can do.

How the system reacts can be seen in figure 1. The player can do different options based on which input is done based on an input controller. While it may look more obvious to have the interaction be done based on the input controller, we decided to design the system to work the other way around. Because we work with different entities we decided to let every actor try every interaction and only initiate an action if the correct input is given.

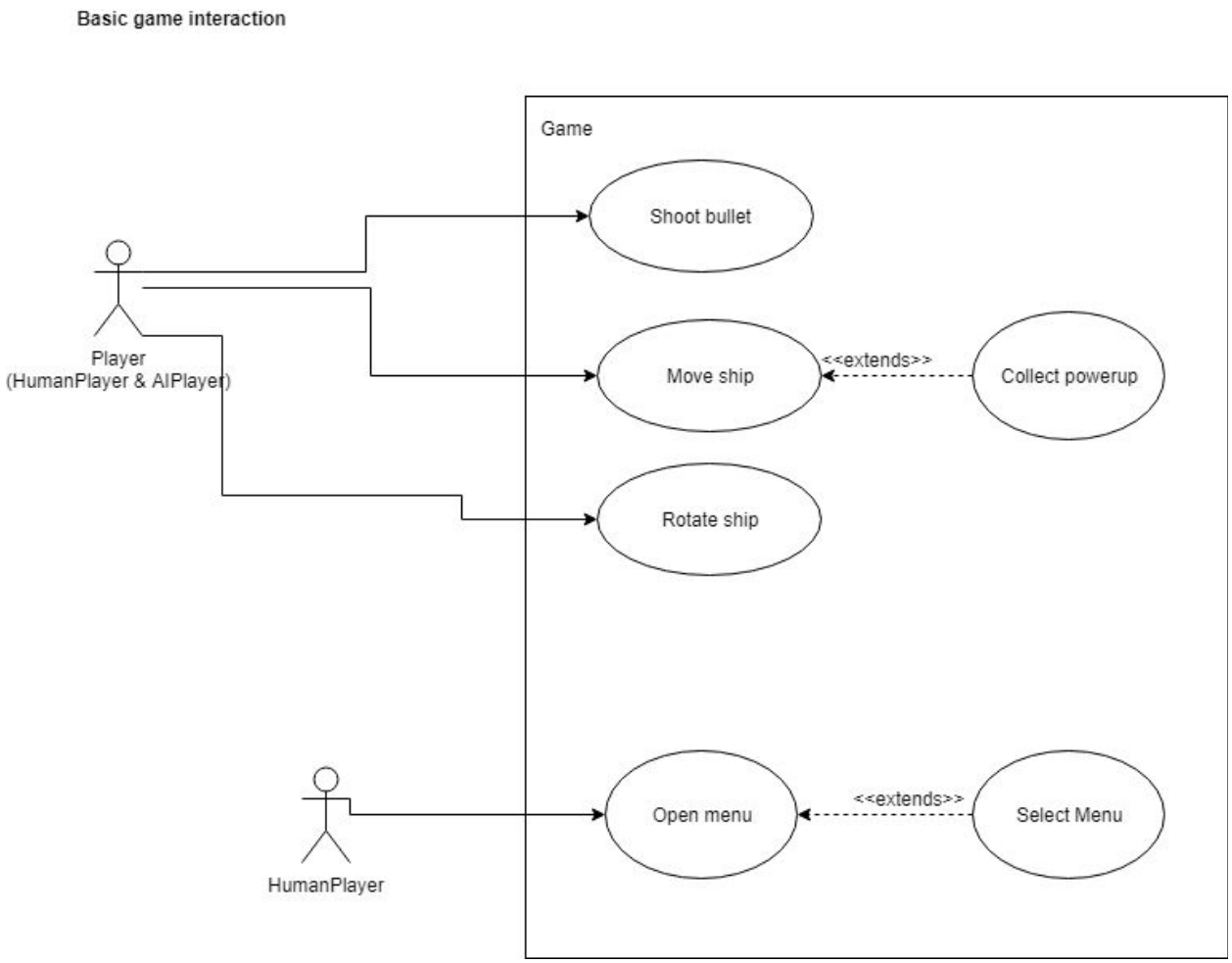


Figure 1: Player interaction Use Case Diagram

A use case for the above behaviour could be a game mode or power-up type in which two objects can be controlled by the same input controller. Instead of linking two different controlling patterns to one object type we can now recycle the same object type to one input controller.

When in the future more components or player-interactions are implemented, the same structure will be used as how the interactions are displayed above. The player can do an action after which other components will check if the interaction is valid and actually initiate the action if needed.

Class overview:

Entity Component System

Like mentioned earlier, the game we want to create is based on an Entity Component System (ECS). This means that we need to implement a structure that can handle multiple Entities and Components that manage the game. We also need to handle the renderer logic into our ECS. How a basic ECS is designed can be seen in figure 2. In our design an Entity is an object that can include different behaviour in the game. The logic itself is stored in components. The render will be responsible to render all the entities based on the information stored in a RenderComponent. The Renderer will be responsible to order all the Entities based on a z-layer and shader call to make the rendering of the game as efficient as possible, while also giving the ability to play around with different depth layers (z-layer). This entity can be seen as a container for all the different logic it can handle. A component can be one of several options. An example of this is the HealthComponent. This component is responsible for all the logic concerning an entity's health. Components also need to work together. For example, if a collision has taken place, the collision will be responsible to notify the Health Component of that entity that it should take damage.

We are still doubting if we should add an extra layer GameObject which would be a base object for all the more concrete objects created out of an entity like a player ship or enemy ship. But because they would be nothing more than a container for an entity so it knows which components it should incorporate, we did not include it yet. If in the next iteration it will be deemed more useful, we can still decide to incorporate that idea.

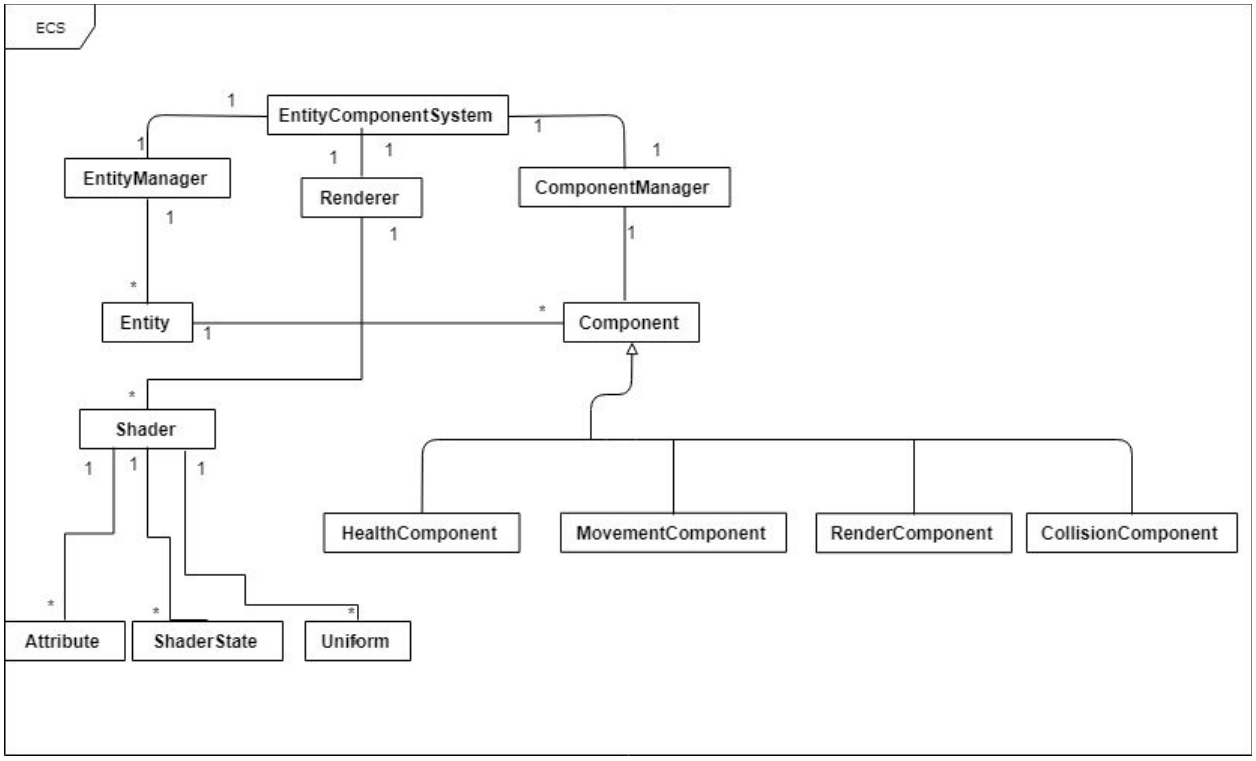


Figure 2: Entity Component Class Diagram overview

In figure 3 a more in depth overview of our design is given. Here we can see that the EntityComponentSystem class will be responsible to initialize and update the EntityManager, Renderer and ComponentManager classes. These classes are responsible to handle all the Entities, render pipeline and Components. The entities and components are created by their respective manager. The two manager classes make sure their lists of components and entities are updated when needed and in the case of the entity manager that all the entity's corresponding logic is executed.

All Components have the init() and execute() function to initialize and execute a component. The specific logic for a child component can be called in the override execution function if it needs to be executed every frame or by its member function if special criteria is met to activate it.

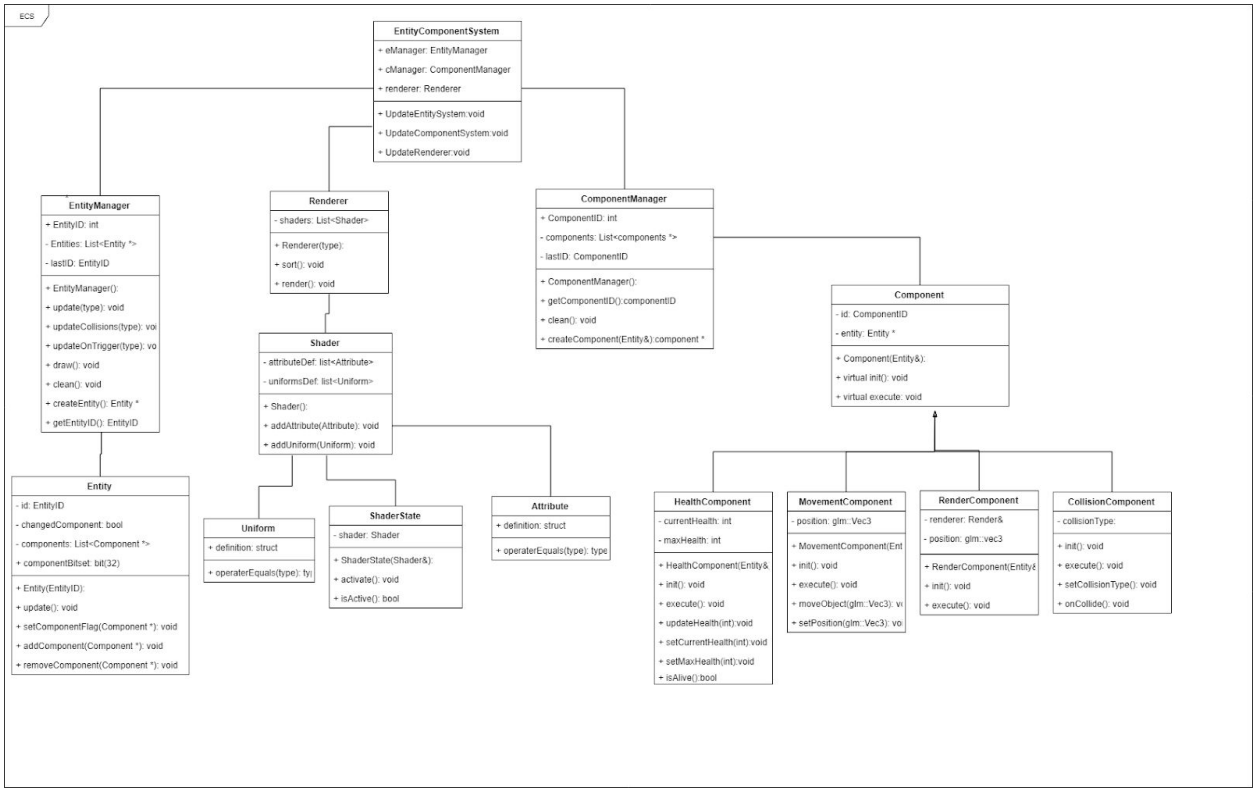


Figure 3: A more in depth look at the Entity Component Class Diagram

Main
The main class is responsible of creating an application window to play the game in and setup the openGL rendering context. An instance of the game is made and the global application time is managed here.

Game
The Game class uses the ECS to generate a playable game. A game object contains an EntityManager, InputManager, CollisionManager, SoundManager and Renderer to manage and create the playable game. It also contains a (very) rough implementation of a state machine to handle different game states. Furthermore it makes sure that the all the components update in a certain way to make sure all the behaviour in the game is consistent during run-time. An example of this is to make sure all inputs are handled before starting to see if there are collisions. The Game class also contains a GameObjectSpawner to instantiate GameObjects when needed.

GameObjectSpawner
The GameObjectSpawner translates simple entities to actual objects that interact with the game. This is done by having a function to create a player, enemy and bullets. The methods for creating these game objects are kept as abstract as possible and is nothing more than creating an Entity and adding certain components to them. The information to make the entities and components act differently is purely done by components and the data and interaction that is included in the components.

EntityManager:
The EntityManager manages all the entities in the game. This class keeps track of all the entities in the game and gives a call to all entities to let their components do their work. The manager also signals the entities so the entities clean up their own components. The EntityManager keeps track of the last unique ID generated for an entity and gives a unique ID to a newly created Entity. This manager is also responsible for deleting entities that are not in use anymore.

CollisionManager:
As the name suggests this class manages all the collisions that happen in the game. The CollisionManager keeps track of all the entities that contain a CollideComponent, looks every frame if those entities made a collision and handles the collisions accordingly. It was chosen to first find all the collisions in a frame before handling the collisions to make the overall behaviour of the game consistent. To keep track of all the collisions in a frame, the CollisionManager has a list of something we called CollisionEvents. CollisionEvent is a struct that registers the two entities which collided. Once all the needed entities have been checked for collision, the Entity Manager will loop through all the CollisionEvents and

handle the collisions accordingly. What an entity should do when it collides is handled further in the CollideComponent.

A player should not be hit by their own bullets and an enemy should not be able to pick-up powerups. To determine which type of components can collide, we use a simple lookup table and an identifier for every collidable object. Currently we have 5 different collision identifiers, namely player, enemy, bullet, enemy_bullet and power_up.

The initial idea was to incorporate some spatial sorting algorithm, but due to time constraints and overall performance of the game we decided this was not a priority for the third iteration.
The actual collision detection is handled by the collision components so the manager does not need to know what kind of collision took place, only that a collision took place. At the moment the game only supports circle collisions, but this can be build upon further in the future. The CollisionManager itself does not have to be changed to incorporate additional ways to find collision.

InputManager:
The InputManager's main duty at the moment is to keep track of all the keyboard and mouse interaction done with the game and make sure entities linked to a InputComponent act accordingly.

Abstraction of input and the ActionController
To let input work with the desired ECS it has to be handled in such a way that keys are not limited to one character, entity or action. We also felt obligated to include make the controls as dynamic and customizable as possible to fit with the ECS theme. To achieve this goal we abstracted the SDL input to an extra layer. Instead of linking button presses to functions, we added an ActionController in between. Every ActionController is linked to a keyboard key or mouse key. This way one key can be used for more actions, while actions are linked to a single entity. An example of this is that one key can trigger more than one ShootComponent without triggering all the ShootComponents in existence.

The input generated with Window Events of SDL is caught by the InputManager and if a pair of an event and ActionController exists, this will be updated. This works for every time a key is pressed, released, or when the mouse is moved. This also means that an action can be linked to more button or mouse actions.

Furthermore the InputManager is also responsible to manage every InputComponent. This includes the registration, deregistration and urging those components to update. In short the entity and their components are influenced by ActionControllers and not directly by buttons so they do not know and care which buttons are connected to their ActionControllers. This

abstraction also means that an ActionController can switch their trigger without Entities noticing.

Entity:
The entity class is the backbone of all instances of objects that are instantiated in the game. While an entity in an ECS normally would be limited to just being a collection of data, we included some logic into the class to manage components. A component has an ID, a bitset to indicate which components it has and an array to have a fast way to call single components with the help of an index.

As mentioned before an entity manages its own list of components. The components are added directly to an entity and also need to be removed from an entity directly. All components are updated one by one per entity. An entity also contains a get and set function for components. When a call is done to removeComponent, a flag is set to indicate the component needs to be removed. The component is not removed immediately because it can be possible that the component is still needed in the update of another component. All the component-based functions are made as template functions so every component is useable without recreating the same functions for the specific type.

Component:
Component is the base class of all the different components that are made for the game. The component class is an abstract class which has 3 virtual functions, namely init(), execute() and toString(). The init and execute functions are needed for every component to work in the system created. The toString function is included for debug purposes. The base component class also contains a pointer to the entity it is linked to. At the moment we have two kinds of components, namely a data component and a logic component. Data components contain pure data, while logic components manipulate the data which is encapsulated in a data component. During the development of the game it was proven to be a lot harder to separate the components in being purely data or purely logic based with the system we had designed. For some components we noticed that the system

TransformationComponent:
The TransformationComponent is a derived class of Component and acts as a data component. This component stores the location and transformation of an entity. It also contains a get and set function to get the location and the transformation of an entity and to manipulate it. It is also possible to scale and rotate a TransformationComponent.

MovementComponent:
The MovementComponent is a derived class of the component class and acts as a logic component. The MovementComponent contains the logic to move a component. The

MovementComponent can move an entity every frame by initialising a constant movement variable, or do single call updates received from other components in the game. The MoveComponent is dependable on the TransformationComponent.

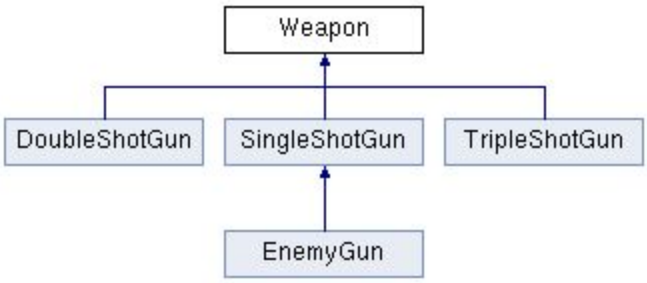
InputComponent:
The InputComponent is a derived class of the component class and acts as a logic component. The InputComponent is responsible for delegating the input done by a user to the affected components. If the component does not contain the component needed for an input, that specific input will be dropped.

HealthComponent:
The Health Component is a derived class of Component and acts as a data component. This component gives hitpoints to a keeps track of when an entity needs to be removed. If an entities hitpoints are below zero. It also contains functions which can be called i.e. during a collision to lower or higher the hitpoints.

RenderComponent:
The RenderComponent is a derived class of the component class and acts as a logic and data component. This component makes sure a shape that has to be rendered is put in the correct place in the world after which it queues itself to the renderer. The renderer will do a call to the RenderComponent when it is it's time to render. The data on how an entity should render is encapsulated further in the Shape property.

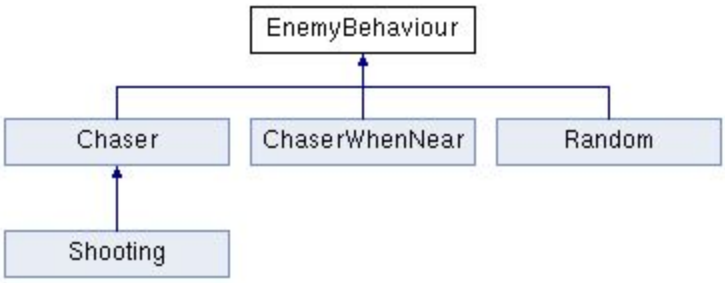
Shape
Shape class is the basis for simple polygon drawing on the graphics card. Rendering of most Renderable game objects will be implemented using this class. How an entity should be drawn and what it needs to do is all contained in this class.

ShootComponent
The ShootComponent is a derived class of the component class and acts as data and logic component. While the component itself is solely a data component, extra data this component needs is encapsulated in a derived class of Weapon. The ShootComponent is called by the InputComponent if an entity has both components. There are some enemy behaviours that can call the ShootComponent to shoot.



Weapon
Weapon is a base class of all different weapons that are available in the game. The weapon class acts as an interface on what a weapon implementation needs to do. Weapons are used to determine what kind of bullets have to be spawned in the game and how those bullets behave.

EnemyBehaviourComponent
The EnemyBehaviourComponent is a derived class of the component class and acts as data and logic component. The logic on how to update the entity attached to the EnemyBehaviourComponent is encapsulated in an EnemyBehaviour struct. The EnemyBehaviourComponent contains all the data needed for an enemy component to move and interact with the world.



Behaviours
The EnemyBehaviour struct can be seen as the actual AI component to the EnemyBehaviourComponent. The current behaviours included are:

- Chasers: Chase one enemy
- RandomMovers: Move based on a random position
- ChaseWhenNear: Only chase an entity when it is close to the entity with this behaviour
- Shoot: Stay in place and shoot an entity
- ShootAndChase: Shoot and chase an entity.

To make sure

Right now the behaviours are limited to one behaviour per EnemyBehaviourComponent and if you want to combine different behaviours, you have to switch the behaviours or create a new EnemyBehaviour by inheriting from one of the other Behaviours. This last approach, which we used for the ShootAndChase behaviour, is more limited to changing behaviours or having an array of behaviours as you are quite limited in which behaviours you can combine or reuse. Even more so if we want to combine three different behaviours.

CollideComponent

The CollideComponent is a derived class of the component class and acts as data and logic component. The logic on how to check for collision and what to do on collision is also included in this class. The hasCollision and onCollision methods that check for collision and handle what to do after a collision are both virtual. To get different behaviours, the idea is to create a new component derived from CollideComponents where the hasCollision and onCollision have been altered. We decided to take a different approach to include the logic in this component compared to the Shoot, Render and EnemyBaviourComponent as we had these two ideas for the changing behaviour problem and wanted to know which of the two would we would prefer. We found out that having the logic component as a separate class, as has been done in the RenderComponent, EnemyBehaviourComponent and ShootComponent allows a more flexible and quicker way of implementing different behaviours needed for the structure. It also gives a more cleaner separation between actual logic and data, if done correctly.

Design of OpenGL abstraction:
For the rendering of the game we want to use modern OpenGl. OpenGL is a C API, which extensively makes use of global state. This means OpenGL can be hard to combine with modern object oriented C++ projects. To work around this problem we are programming our own rendering abstraction on top of OpenGL, This allows use to use OpenGL through a more object oriented interface.

Three important factors had to be considered: performance, usability, and safety. Wrapping OpenGL objects in C++ objects makes them a lot safer to use. This is because changing a feature of a texture in OpenGL involves first binding said texture to the global active texture, and then applying an OpenGL function that changes the active global texture. By making this function a method of the Texture class, we can bind the texture to OpenGL’s active texture, and apply the function in one step, This reduces the risk of semantic errors.

The only issue with this is a possible small overhead, when doing multiple operations on a single Texture object in a row. This would result in the Texture being bound to OpenGL’s active texture more often than necessary. We still accepted this overhead, because it does

make out code a lot safer, and assume the Graphics drivers are smart enough to do a check if a texture is already active, and return the binding function early.

The hardest part of creating the OpenGL abstraction is the way to deal with Shader programs. These are C-like programs that are executed on the graphics card, that can be used to place vertices of geometry on the screen, and calculate the colors of the pixels within these geometries.

Shader programs have two types of variable data that can be supplied from the program using OpenGL. Uniform variables and Attribute variables. The uniform variable are the same for all executions of the Shader, and can only be adjusted in between draw calls. The attributes are providing unique data to each invocation of a Shader, which generally means one set of uniform variables per vertex that is being rendered. The problem is in the fact that OpenGL stores the values of the uniform variables with the Shader, The issue is that many different C++ objects want to use the Shader with different uniform variables, to render themselves. This means that each time an Object wants to render it needs to tell OpenGL the value for every uniform variable in the Shader, which is quite a performance hit for the program.

Two major iterations were necessary to arrive at an interface we are happy with, is performant enough for a medium sized game, and has a nice object oriented interface.

In the first iteration a class could be given rendering capabilities by inheriting from a class that implements Renderable. For instance classes that would implement DefaultRenderable, would gain access to protected member variables such as set_vertex_positions(), set_color(), and set_transformation(). DefaultRenderable was implemented using a relatively basic shader, and using methods such as set_color() would directly set uniform variables in the shader's state. After a lot of discussion it was found that this method was not satisfactory for our needs for multiple reasons:

- Every shader that we wanted to use would require a specific subclass of Renderable.
- It would be impossible for an object to be rendered using two or more shaders.
- Things that need to be drawn that are not part of the game HUD, menu's, score etc, would need to inherit from a Renderable Components, which is meant for game objects.
- Shaders would end up being implemented as classes for which only one instance is made (not a nice design).
- Lastly this method is not very efficient.

The second iteration a new method was designed to improve on these points. We first looked at the anatomy of a shader in OpenGL, The state in a shader is decided by two

things: the attributes (variables unique per vertex), and the uniforms (variables unique per draw call). Furthermore we found that generally you only want one copy of a shader, in your code, even though you want to render to this shader with many different states of this shader. Because of this we decided to split the Shader into a definition part, and a State part. The state can be activated so that all variables can be stored in said state, and restored after something else was drawn.

This change meant there are now six main classes:

- Shader: Holds the immutable part of a shader program (source, attribute - and uniform definitions
- ShaderState: Holds the mutable part of a shader program (Uniforms and Attributes)
- UniformDefinition: immutable definition of the existence of a uniform variable in a shader program.
- Uniform: the variable uniform value in an OpenGl shader.
- AttributeDefinition: immutable definition of the existence of a attribute variable in a shader program.
- Attribute: the variable attribute value in an OpenGl shader.

Design of the audio system:
Playing sounds from the game means the game code needs to send the audio data to the audio driver, so the audio driver can play the sound on the speakers. This communication is operating system/ audio driver specific. Luckily SDL contains an audio abstraction with a platform independent interface. This interface is what mostly dictated the design of the audio manager in our game.

In SDL you play audio by opening an audio device, and giving this audio device a function-pointer of a function it can call to retrieve more audio samples. This means that if multiple sounds should be played at once the samples of these multiple sounds need to be mixed first before being supplied to SDL by the callback function.

This that we need two types of data to properly play sounds. The Sound struct, which holds the samples of a sound, the amount of samples the sound has, and the specification of the samples (mono, stereo sample frequency etc). This Sound class we don't actually want to use to keep track of sounds that are currently playing, because there is no need to copy the all the samples each time we wish to play a sound (they always stay the same). Because of this we also have a SoundInvocation struct. This struct has a pointer to the Sound struct that it belongs to, and keeps track of how many of the samples of the SoundInvocation have been played already, and how many still need to be played.

Furthermore we have defined an enum class of sound ID's, "Sounds" that has a unique value for every sound that is used in the game. This is useful when referring to sounds in the code with minimal resource usage, and clarity of the code.

When the user sends the wants to play a sound, they can invoke a function in the audio manager with a Sounds enum as an argument, this function looks up which Sound instance belongs to the enum, and creates a new SoundInvocation, which is added to a list. In the callback function all soundsInvocations are looped through, and handled.

Because we know for sure we never want to have more than one audio manager, and because it should be possible to play sounds from anywhere in the game code, we implemented the audio manager as a completely static class. This means the class has no constructor, destructor or non-static data members. This essentially makes the class a namespace, however it still allows you to hide static variables as private, which keeps the interface of the "class" clean.

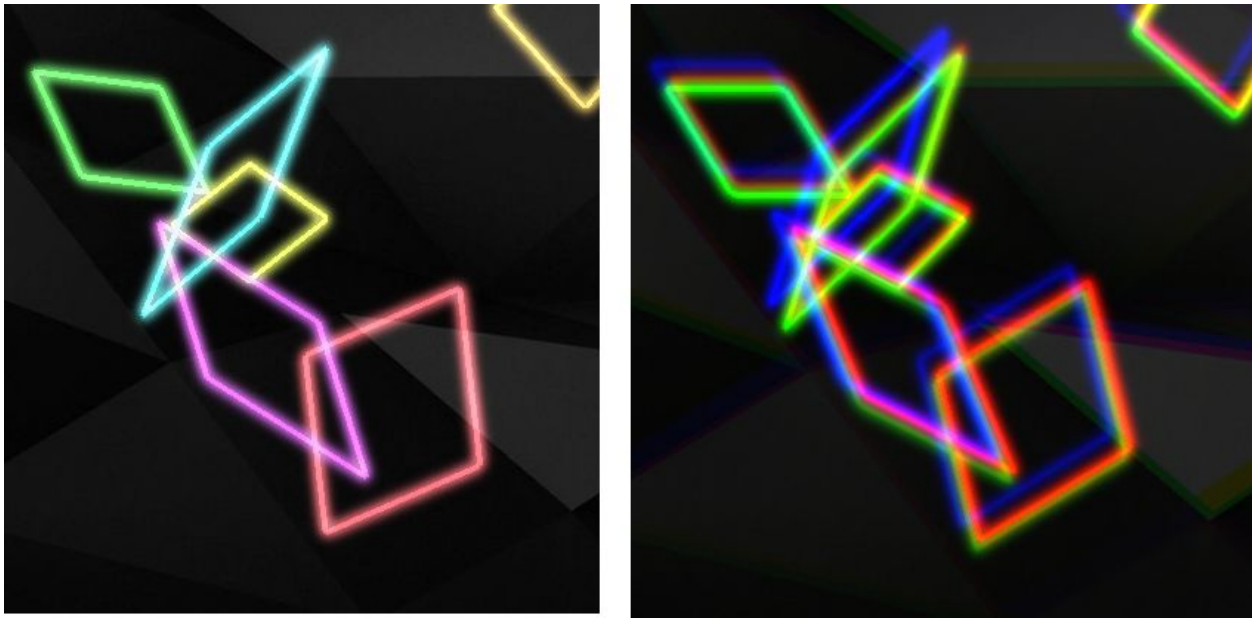
Usage of the audio manager now looks like this:
```
AudioManager::initialize();
AudioManager::play(Sounds::EXPLOSION);
AudioManager::shutdown();
```
Etc.

Description of rendering pipeline:
To render things to the screen we use OpenGL with our custom OpenGL abstraction (see Design of OpenGL abstraction), All rendering is done using Modern OpenGL 3.3+ core profile, using custom shaders. The renderer makes extensive use of framebuffers allowing us to do multi-pass rendering, which gives possibility for a lot of cool effects.

To actually render the game objects a RenderComponent class is used. This component registers itself every frame, it wishes to be drawn, at the Renderer. When all components have been updated, the render_frame() function is called in the renderer. We will now describe the Rendering process in more detail:

- The first FrameBuffer is bound as active in OpenGL, and it is cleared to uniform black color.
- All the components that have been registered to be rendered this frame are iterated and drawn to the frame buffer. For this the default shader is used. The default shader is quite minimalistic: it allows you to set the vertex values (the shape), select a color, select the draw mode OpenGL uses (GL_POINTS, GL_TRIANGLES, etc), and apply a linear transformation to each vertex. The result is uniform colored polygons on a black background.
- The first FrameBuffer is unbound so that the whole rendered image can be used as a texture in following shader passes.
- Now to add the bloom effect to the game we need to combine the initial rendered image, with a blurred version of the image, to do this we need to blur, we made a special blurring class for this purpose: GaussianBlur:
- GaussianBlur works by having a list of Framebuffers and iteratively sampling one frame buffer while writing to the next one, horizontal and vertical passes are separated to give much higher performance. Four passes of blurring are applied, and each time the target framebuffer is slightly smaller than the source framebuffer. This is both good for performance, and makes the blur area larger. A good description of the algorithm we implemented can be found here: https://youtu.be/LyoSSoYyfVU
- Next the original non blurred image, and the blurred image are taken, and they are combined together using a weighted sum function per pixel. The result of this is a image of the shapes radiating their color outwards, giving the effect also known as Bloom.
- For the next stage we need to combine the finished bloom image with the background, it is not possible to render the background first, because we do not want the background to appear blurred. The background is also rendered to its own framebuffer. The background and bloom image are combined together by looking at which pixel is the brightest.
- In this last shader pass some last effects are also applied, such as tearing apart the r, g and b channels, when the player is shot, and inverting the final image if desired.

The look of the game in the intro screen



The result of the rgb tearing, effect when player is hit with a bullet.

Evaluation, limitations, known bugs, and future work:

- This version of the game is a proof of concept, and does not yet have very many gameplay elements. We would still like to implement a score system, levels, more enemies, more types of attacks, better art for the game etc. etc.
- Currently a lot of out logic and physics is bound to the framerate, ie maximum movement speed of 5 pixels per frame, this would be nicer to be bound to the frametimes.
- Sometimes the Sound crashes, this is because of a multithreading bug that needs to be fixed with locking the sdl audio devices.
- A feature we would still like to add is the ability to use compressed file formats for loading and storing the assets, this is not supported by sdl, and was too much work to implement ourselves.
- Right now for every shader we add we have to define the layout of the shader: uniform variables, attributes etc. This is necessary for the ShaderState to function properly. It would be very cool if we could read in the shaders, do some processing on them and then deduce this information from the source code of the shaders.
- The design of the engine is a mixed bag. Adding and creating GameObjects is very fast and easy to do. We do miss having the Components and Systems separated. While it is mostly fine in the structure we achieved right now, we noticed a lot of dependencies going into the GameObjectSpawner. When using a real ECS you do not deal with these problems as you can just call the data from components through a System Manager or something likewise.

Division of work over team members and next iteration (old from second iteration)
For the second iteration we made sure that the Entity Component System and Renderer are able to work with each other. We reached our goal, so we are now going to actually implement the game for the third iteration. Right now our main concern for the game is to come up with a design for the event handling and implementing it. We hoped we could solve it purely by using components, but we did not like the outcome so we decided to include some extra time to build an event handling system that works to our liking.

Furthermore some small fixes for the existing components need to be added like the option to show the game in a fullscreen window and creating a more refined input handling. We are also missing an implementation to play sounds.

The end goal for this iteration is to have a working game. As we now have most of the components we can already start with the base of the game. The game class can be created, assets can be made and a simple level manager should be build.

What	**When**	**Who**
Create / implement event handling	Christmas holiday	Dennis
Make game fullscreen	Christmas holiday	Aart
Create first version game	Before last cycle	Everyone
Create game content	Continuously	Everyone
Implement input handling	Christmas holiday	Dennis / Robin
Sound	Christmas holiday	Aart
LevelManager	Before last cycle	Robin
Game class	Christmas holiday	Aart
Clean up code	Continuously	Everyone
Report	Continuously	Everyone
Documentation	Continuously	Each person their own code
Add proper folder structure	Soon	Robin

User manual (Steps to compile the project)

* Source code will only compile with C++17 standard.
  In visual studio this settings can be found in the project properties under
  C/C++ >> Language >> C++ Language Standard >> ISO C++ 17 (/std:c++17)

* Create a new solution and add all the .h and .cpp files found in:
  - /Geometry_Wars/
And all its subdirectories

* Make sure that the data folder is inside of the Geometry_Wars folder, and the Geometry Wars folder can be found, by moving up folders from the executable's directory:

  E.G. When the executable is in:
  C:\Users\aart\Documents\AP\AP\x64\Debug\Geometry_Wars.exe

  then having the shader folder located at:
  C:\Users\aart\Documents\AP\Geometry_Wars\shaders

  Is okay because the "Geometry_Wars" folder is visible by moving up from
  C:\Users\aart\Documents\AP\AP\x64\Debug\Geometry_Wars.exe to
  C:\Users\aart\Documents\AP\

* Three external libraries are used (with Johan's permission): GLM, glad, and SDL2. For each:

* GLM is header only and thus the only thing that needs to be done is add the folder /glm_include/ to the include directories, under project properties.

* GLAD has one c file which we have already added to the sources. The folder /glad_include/ needs to be added to the include directories.

* SDL2: add the folder /SDL2-2.0.9/include/ to the solutions include directories, SDL3 has three.lib files, that need to be added. Under project properties >> VC++ Directories >> Library Directories, add /SDL2-2.0.9/lib/x86/ or /SDL2-2.0.9/lib/x64/, and under Linker >> Input >> Additional Dependencies add SDL2.lib, SDL2main.lib, SDL2test.lib. Lastly add SDL2.dll to the executable directory. For more information see: https://www.wikihow.com/Set-Up-SDL-with-Visual-Studio-2017

* Lastly when using SDL sometimes it is necessary to specify the subsystem: in project properties, under linker >> system >> SubSystem, select Console. If you do not follow this step, you will get a runtime error when executing the application.