Advanced Programming: Cycle #2

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.

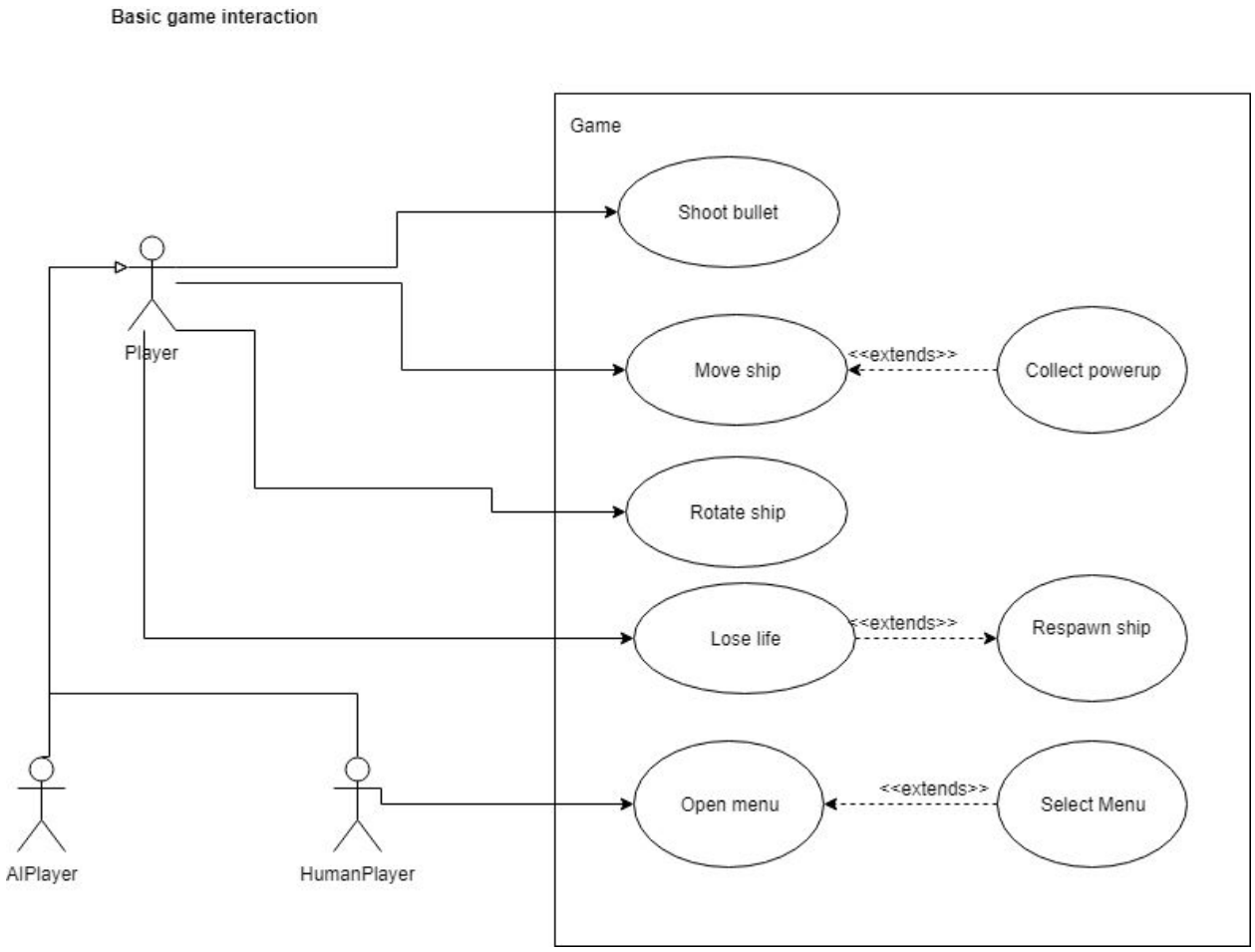
When the shoot button is pressed, the ship the player is controlling will shoot from the front of the ship. The player will be able to pick up power-ups to change the way the shoot action is being visualized on the screen. For instance, it should be possible to shoot two bullets instead of one when the 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 UML1. 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.



UML1: 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.

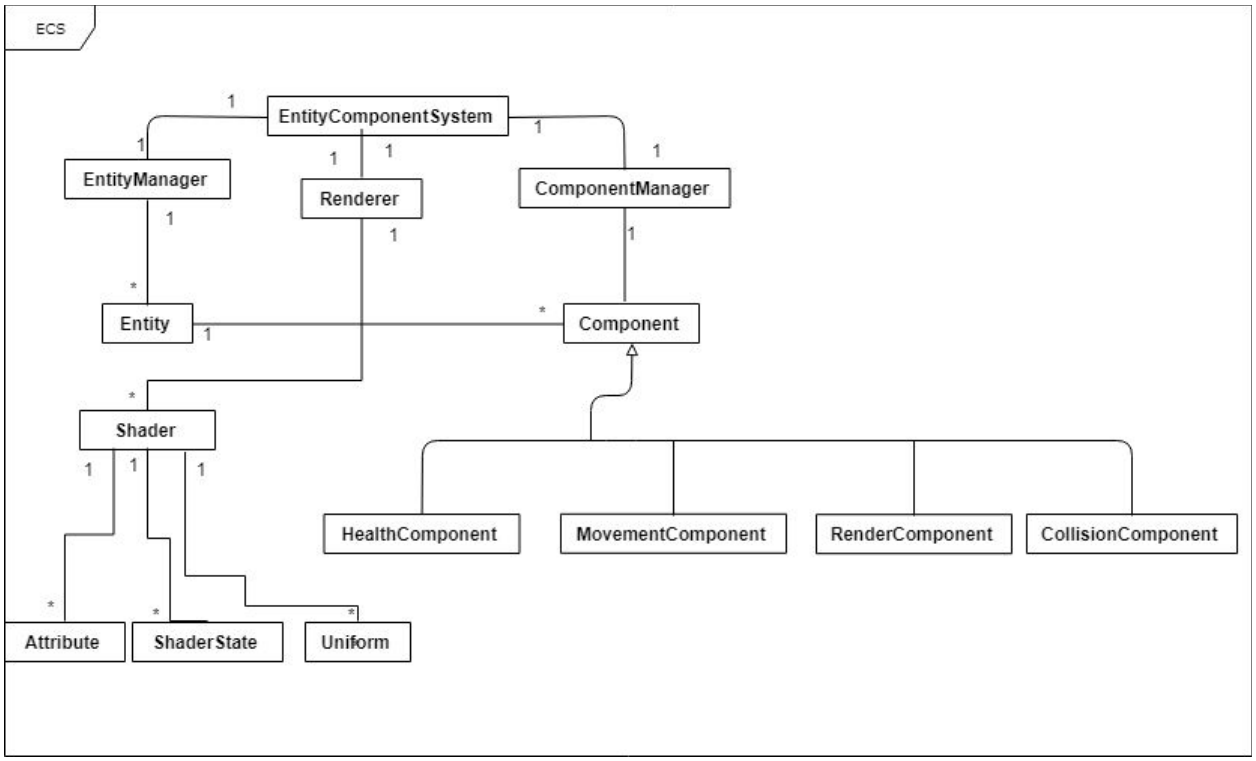
When in the future more components or player-interactions are implemented, the same structure will be used as how the interactions are displayed above. The player can do an action after which other components will check if the interaction is valid and actually initiate the action if needed.

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 we designed the ECS can be seen in UML2. 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.

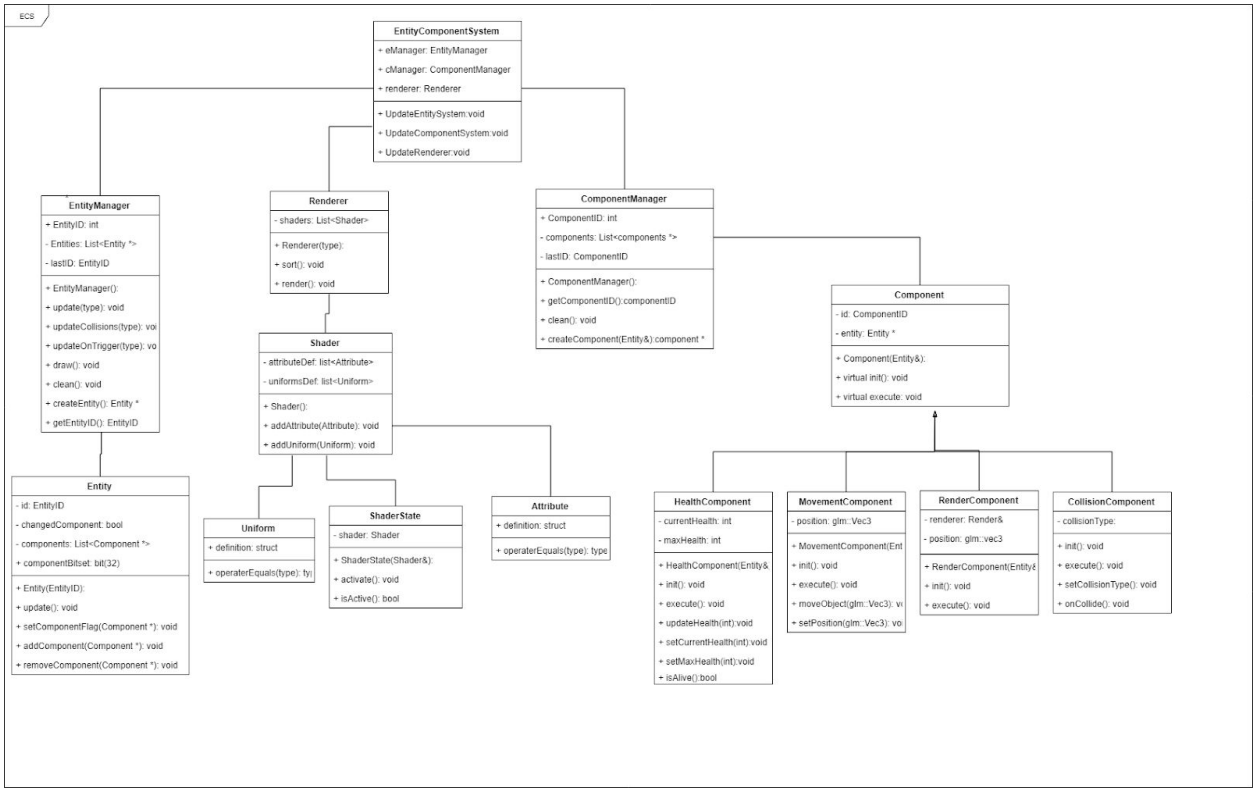
We are still doubting if we should add an extra layer GameObject which would be a base object for all the more concrete objects created out of an entity like a player ship or enemy ship. But because they would be nothing more than a container for an entity so it knows which components it should incorporate, 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.



UML2: Entity Component Class Diagram overview

In UML3 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.



UML3: A more in depth look at the Entity Component Class Diagram

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.
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.

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 InputComponent 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. Right now we combined the shape that should be rendered to the logic to let it be rendered, but we are still discussing if we should give the shape its own component.

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.

Division of work over team members and next 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/
  - /Geometry_Wars/io/
  - /Geometry_Wars/opengl/

* Make sure that the shaders 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 /Geometry_Wars/glm_include/ to the include directories, under project properties.

* glad has one c file that we have added to /Geometry_Wars/ so that should be compiled properly by following step 2. Furthermore the headers of glad need to be added by including the folder "/Geometry_Wars/glad_include" under project properties

* SDL2: add the folder /Geometry_Wars/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 /Geometry_Wars/SDL2-2.0.9/lib/x86/ or /Geometry_Wars/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.