Key responsibilities.

Jamie - *Audio Manager / Menu*
Responsibilities - Implementation of the Audio Manager and creating and maintaining various menus.

Sam - *Entity Component System / Renderer / Editor*
Responsibilities - Implementation and maintenance of the ECS system. Creation and maintenance of the game and editor 2D rendering systems. Creation and maintenance of the level editor tool to be used to design and build levels for the game and manage the game asset pipeline.

Steve - ***Entity Component System / Input System / Control Systems.***
Responsibilities - Implementation and maintenance of the ECS System. Implementation and creation of the Input and control systems. Creation of entity behaviours within the structure of the ECS.

Zack - *Collisions / Events*
Responsibilities - Implementation and maintenance of collision system and collision resolution system. Creation of Events and Event Bus used for communication across different systems in ECS.

Csongor-Zsolt - *Health bar / Multi-entity / Boss*
Responsibilities - Implementation of multi-entity objects, such as the health bars; feature-enriching systems for boss implementation

Discussion of class structure for the game engine.

The game engine was implemented as a static library that provides base functionality for client code to build from. This included creating a window, receiving operating system messages, and creating the entity and component data structures.

Two client codebases of the game engine library were created. The first implemented the game runtime that loaded pre-processed optimised data files to display, and performed game logic to create gameplay and game systems. The second implemented the level editor which imported artist authored files in external applications and provided tools to build a level for the game. This included a tile map editor, free entity and enemy spawn placement. The editor exported the level in optimised data file formats that could be loaded by the game runtime.

A core goal of the engine was to implement a data oriented design instead of an object oriented approach in order to reduce cache misses and improve performance. This was implemented in the form of the entity component system.

The game engine layer provided its functionality in the following classes:

UML
UML Scalable Vector Graphic

ScarleGame
The ScarleGame class is the main class that client codebases must extend to use the game engine, providing a series of virtual methods that are called during engine events such as start, update, render and on any window state changes. ScarleGame takes ownership of core engine objects such as the Direct 3D device and context, swap chain, audio engine and the ECS. Creation and destruction of these objects are managed by the class. This class is persistent, in that it is created at application startup and will not be destroyed until application shutdown. This allows client code to safely access its members during application lifetime.

Audio Manager
The Audio manager's main job was handling the creation and playing of all audio tracks needed by the game. This worked by using the DirectX Xaudio library and the DirectX toolkit. To create a new sound effect the file path to a wav file was passed to the manager which then firstly checked whether that effect had already been loaded into memory, this prevented the same effect from being unnecessarily loaded into memory again and instead returned an index to an array stored in the manager, where all sound effects are stored. Once an effect was loaded an instance could be created which allowed the effect to be played. By using instances to control the playing of sound effects rather than just playing the effect itself it allowed the same effect to be played by multiple different objects without needing to load the effect multiple times. The manager held a vector of unique pointers to all effect instances so that it could not only stop all instances when needed but also would handle the deletion of memory of each of those instances when needed.

Event Bus
The Event Bus was designed to enable communication between different Systems in the ECS, as well as the entire game project, without coupling these different systems to each other.
The structure of the Event Bus was based on the Observer/Locator programming pattern.
The implementation allows for functions across the different systems to subscribe to a list of an event of a given type.
When the Event Bus then invokes an Event it will loop through the list of subscribers of that event type and call the functions. It was decided to not add Events to a queue to be resolved at a later time so that it was in the implementers control as to when the Events would take place during execution.

Events take the form of simple structs that are used to transfer data relevant to the event type to the different functions.

FSM
This implementation of the finite state machine pattern utilises the game class as state machine. A base state is used as an interface mirroring the methods used for the main game loop, update, fixed update and render methods. It also adds a function for both initialising and ending the state that is not reliant on the constructor or destructor. The state is stored as a unique pointer within the state machine and then each pure virtual function is called from the corresponding part of the main game loop in the game class.

When creating the state, the game passes a reference to itself into the constructor, which is then stored within the state. In doing so the state can be treated as an extension of the game and have access to any public members.

Camera2D
The Camera 2D class encapsulates properties about a 2D camera. It stores the camera's position and zoom and provides methods to add offsets to these and calculate the matrices required for rendering systems.

ECS - Entity Component System
The Entity Component System was developed by using three main classes, those were the *Entity Manager*, the Component manager and the System manager.
The Entity managers responsibilities were for the creation of unique id's that act as entities. It holds the responsibility of tracking and repurposing the ID when no longer in use.
The ***Component Manager*** was implemented with two main responsibilities. The first being registering components and creating arrays to store component data. The second, the creation of individual component data and keeping track of the data position in the array while also keeping the array packed and managing a map to the entity that the component belongs to.
Component Data takes the form of simple structs, then using templating and generic coding the component manager is able to use this without the use of inheritance.
The ***System Manager*** allows for systems to be registered at runtime and uses a bitwise signature to keep track of what entities each system cares about based on the component data registered to each entity.

The ECS uses the following component structures and systems:

Input System
The input system is responsible for the creation of the directx keyboard and tracking the keyboard state. The System, in junction with the Entity Component System back end, tracks each entity that owns an input data struct. The structure itself is a set of keys, named after game actions, that can be mapped to any key on the direct x keyboard.
In the update loop the input system then takes the entities that have input data and applies that to movement axis and actions then applying it to the relevant components.

Rendering systems
The engine provides systems for rendering sprites, animated sprites, HUD images and HUD text. Each system has a corresponding component data structure that contains the data necessary for the system to operate. Each of these systems draws the sprite or text into a supplied sprite batch which is sent to the GPU in a single draw call. They also require a pointer to the loaded texture resources.

Collision Resolution System
The system made use of the Event Bus so that the collision detection between different entities could be decoupled from the resolution details of the different collisions that occurred.

How effective was it to use/develop with?
An aspect of the engine implementation that worked well for the project was creating the engine as a library that client codebases could use. This made a lot of code reusable increasing the maintainability of the codebase. An error with a core system could be fixed in one area and this would automatically apply to client codebases.

Furthermore, separating the systems so that each one performed a specific task made working collaboratively easier. There were less conflicts encountered when merging work as each individual was working on an isolated area of the codebase. This accelerated development time as less time was spent resolving conflicting work, allowing more time to be spent creating or improving other features.

The structure of the core Scarle Game and Game classes used helped to keep low level engine objects encapsulated away from game data. This made it easier to know what data was specific to the game and to allow read only access to lower level objects when they were needed by the game. Another advantage of this class structure was that by extending the base Scarle Game class provided each sub class with a common interface. This allowed them to implement their own behaviours for each engine event such as updating, rendering and for any window environment changes. This also makes the code more reusable as unique uses of the engine can be created by extending the base Scarle Game class.

The project used an ECS (Entity Component System) architecture for development. The main advantages of this architecture is that it allowed for decoupling of the logic that was required to act on entities for the game into their own Systems. Each System had an enforced structure through the use of a base System interface. The use of the interface ensured that each member of the team was following a similar code structure for their respective parts of the project and also created a standardised workflow. However the granularity of Systems doing one specific task, whilst useful for knowing what a System was doing, added complexity to the overall structure of the project. This was due to the decoupling nature of the Systems, which made the

flow of the game difficult to follow. It also increased the amount of files in the project which made keeping track of the file structure problematic.

One problem we consistently encountered was that the filters within Visual Studio were not being handled properly during a merge. As the filters act as a virtual file system, without system directories, that meant that when it failed all of the filters were lost and all the files within the solution explorer became disorganised. This made working with the project extremely difficult and made using the solution search bar essential to be able to continue work at a reasonable rate. After several attempts to fix this we gave up as it was taking a considerable amount of time to reorganise the structure. Later in the project we switched to an on-disk file structure and that helped to alleviate the issue.

Another issue we faced with the project level structure was that it wasn't completely clear which project certain systems and components should belong to. The first time this came up was when we found the need to pass a reference to the renderer into another system. The renderer acts like an api for the render systems in the engine but is a part of the game and not the engine. Because we needed access to these parts of the api within engine level systems and the static library couldn’t get access to this, we had to come up with a solution. This was when IRenderer was implemented into the engine to allow us to have an engine level interface that could be used only for the functions that were needed outside of the renderer wrapper.
When creating a namespace factory setup for the creation of entities we found that other systems like bullet and enemy spawn systems weren’t able to access the namespace as it belonged to the executable. To solve this we revised which systems had game related logic in them and if it could be made generic or not and then moved them over to the executable.

Planning for the project took longer due to the need to separate everything into various systems and components. This also meant that when designing a new system, component or game feature you needed to figure out if something else already had the feature you would need to use or if you needed to create it yourself. Due to the nature of ECS components being small and specialized, in the beginning of the project this often meant you would need to create various new components before you could begin working on the game feature or system originally planned. However once components and systems had been created it meant that workflow was increased as the systems and components were modular and were very easy to simply include and use for the feature which was being implemented. The specialized nature of these systems also made the code easy to read and to know from looking at their name what function they served.

References

Morlan, Austin. (2019) A Simple Entity Component System (ECS) [C++]. Code Contact LinkedIn [blog]. 25 June. Available from: https://austinmorlan.com/posts/entity_component_system/ [Accessed 13 April 2021].

Acton, M. (2014). *Data Oriented Design and C++* [Presentation at CppCon14]. Bellevue, Washington, USA. 07-12 September.

Microsoft (2018) XAudio2 APIs. Available from: https://docs.microsoft.com/en-us/windows/win32/xaudio2/xaudio2-apis-portal [Accessed 13 April 2021].

Nystrom, R., author (2014), *Game programming patterns*, Genever benning, Place of publication not identified.