Project Report - Admirals

Oskar Gådin, Casper Norrbin, Joel Sikström, Jakob Wallén

January 12, 2024

Contents

1 Introduction **4**

2 Goals of the engine **4**

3 Related Work **6**

4 The Game Engine **6**
4.1 Design 6
4.2 Interfaces and Classes 7
4.2.1 Top-Level Interfaces 7
4.2.2 Data Classes 8
4.2.3 Static Classes 9
4.3 Objects, Elements, and Containers 9
4.3.1 User Interface Elements 9
4.3.2 Game Objects 10
4.3.3 Containers 10
4.4 Event System 10
4.4.1 Game Loop Events 10
4.4.2 Subscription Events 10
4.4.3 Deferred Actions 11
4.5 Graphical User Interface 11
4.5.1 User Interface Menus 11
4.5.2 Hovering Effects 12
4.6 Collision Detection using Quadtrees 13
4.6.1 Quadtree Examples 14
4.7 Pathfinding and Nav-meshes 14
4.8 Networking 15

5 Minimum Viable Product **15**
5.1 Engine Interactions 16
5.2 Managers 16
5.3 Window Scaling 17
5.4 Pathfinding 18
5.5 Ship Combat 18
5.6 Sprite Animations 18
5.7 Networking 19

6 Project Management **19**
6.1 Version Control 20
6.2 Continuous Integration 20

7 Tools **21**
7.1 Formatting 21
7.2 Linting 21

8 Results **22**

8.1 Profiling 22
8.2 Features 23

9 Discussion **24**
9.1 Performance 24
9.2 Application Testing 24
9.3 Division of Work 25
9.4 Ease of Use 26
9.5 Engine Dependencies 26
9.6 Open Source Contribution 26

10 Future Work **27**
10.1 Usage Examples and Guides 27
10.2 Performance 27
10.3 Graphical User Interface 27
10.4 Additional Features 28

11 Lessons Learned **28**

1 Introduction

Admirals is a game engine primarily targeted for developing 2D, real-time strategy (RTS), multiplayer games in C++ for Linux and Windows. The work done towards developing the engine is part of the course Software Engineering Project (1DL650), which has taken place during period 2 of HT2023.

Although game engines as a concept are not something unique, many existing game engines offer a full and comprehensive suite of functionality and possibilities for creating games that can easily feel overwhelming. A consequence of this is that, as a developer, it can take significant time to get used to creating games with the great number of tools at your disposal.

To make it easier for developers to create games, we aim to develop a game engine focusing on select key features we find most valuable for 2D, real-time strategy (RTS), multiplayer games. By focusing on a subset of features that are most relevant to these specific types of games, our engine can be more portable and lightweight, making it easier to run and use compared to other game engines. This makes using our engine worthwhile for developers seeking an easy-to-use, small, and portable game engine to develop their game(s) in.

The main goal of this project is to develop a game engine to allow users to more easily create games, which we demonstrate by designing a game that we imagine the engine should be able to implement, also known as a Minimum Viable Product (MVP). We use the requirements of the game to decide what functionality the game engine should implement in its exposed API to a game developer. This process of working "backward" from the game to the game engine allows us to only focus on parts of the engine that are required for us to create the MVP. Features that are not needed for the MVP have for the most part been skipped. However, some features that allow other features to be more easily implemented have been implemented for the sake of code readability and ease of use.

The project is in equal parts authored by Oskar Gådin, Casper Norrbin, Joel Sikström, and Jakob Wallén.

2 Goals of the engine

To reiterate, the goal of the engine is for it to be able to create 2D, real-time strategy (RTS), multiplayer games in C++ for Linux and Windows. Although this goal includes several key features, it does not highlight the direction in which to develop the game engine from the start. To have a more concrete plan for developing the engine and envisioning what it should be able to do, we created a concept image of a minimum viable product (MVP) game, as shown in Figure 1. From this image, we have extracted both functional and non-functional requirements for the engine that align with developing the illustrated MVP.

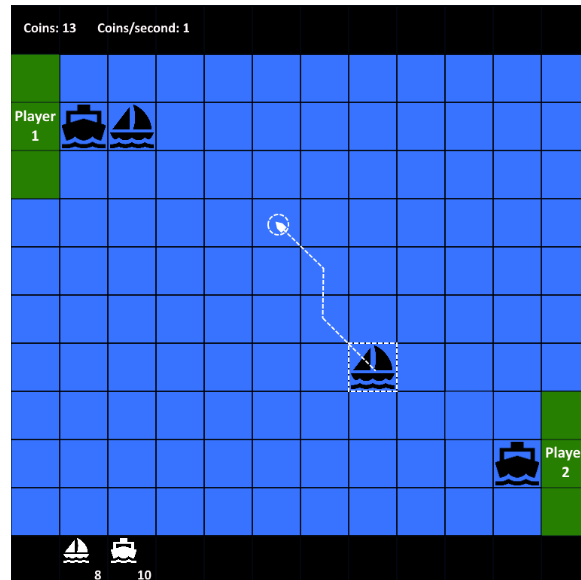


Figure 1: Initial design for the MVP.

The functional requirements extracted from the MVP concept image are as follows:

- The engine should be able to create and store game objects (objects inside a game).
- The engine should be able to create and store scenes, which hold an arbitrary number of game objects.
- The engine should be able to swap which scene is active at any single time.
- The engine should be able to create different UI Elements, like text, buttons, and input fields.
- The engine should be able to create and store DisplayLayouts, which store a collection of UI Elements.
- The engine should be able to swap which DisplayLayout(s) are active at any single time.
- The engine should be able to render both textures and simple shapes.
- The engine should be able to receive mouse and keyboard inputs from the user and handle them.
- The engine should provide a multiplayer framework that a user can build their implementation on.

The non-functional requirements are:

- The engine should be easy to use for someone experienced with the C++ language.
- The engine should be well documented with instructions on how to build and use it.
- Engine functions should be well-defined, with clear names and extra documentation when needed.
- The engine should be well tested, with tests covering all major functionality.

3 Related Work

There exist many popular game engines that are widely used today. Unity [1] and Unreal Engine [2] are two very popular engines that are fully fledged and used by large game studios to create complex games of any type. They have many features and come bundled with an interactive graphical user interface editor to make the games. When used fully, both Unity and Unreal are powerful, but for most developers, many of the features they provide are not needed and remain unused. For developers unfamiliar with their features, developing a game can be a daunting experience. Instead, developers might want an easier alternative that allows them to focus on developing their game rather than spending too much time figuring out available features that they do not need.

One of these alternatives is Pygame [3], which is a Python library to help with creating simple games. It contains functionality to do things like reading user input, drawing shapes and rendering sprites, and playing sounds. It is simple to use, but only contains a limited set of features and does not have an editor. A user includes Pygame in their project and uses the available library functions in their code.

The complexity of Admirals is somewhere in between these ends. It has a thorough set of features, more than Pygame, but does not have the whole suite of features that Unity or Unreal Engine provide. Admirals does not have an editor and is used as a library similar to Pygame.

4 The Game Engine

Developing a game engine is a balancing act between what should be provided by the engine and what is left to the game developer using the engine. The purpose of the game engine is to provide functionality for performing commonly done actions, creating and managing stateful (UI) objects, and handling events, among other things. On the one hand, the engine should be general enough for game developers to be able to create the game they want, without being hindered by certain limitations inside the engine. However, adding too much functionality will make the engine bloated and might decrease performance unnecessarily.

With the previous goals in mind, the following sections describe the functionality that we have chosen to focus on in the game engine.

4.1 Design

As the development of the engine progressed, the structure and design were modified and improved many times. The final structure can be found in Figure 2 which shows the modules and their dependencies. The final structure shows all dependencies and relations except for the EngineContext and DataClasses since their relations are too large to show coherently in the graph. EngineContext and DataClasses are used by almost all other modules and would be placed at the bottom of the dependency graph. EngineContext contains the context used by the engine, and DataClasses contains generic data classes with supporting functions.

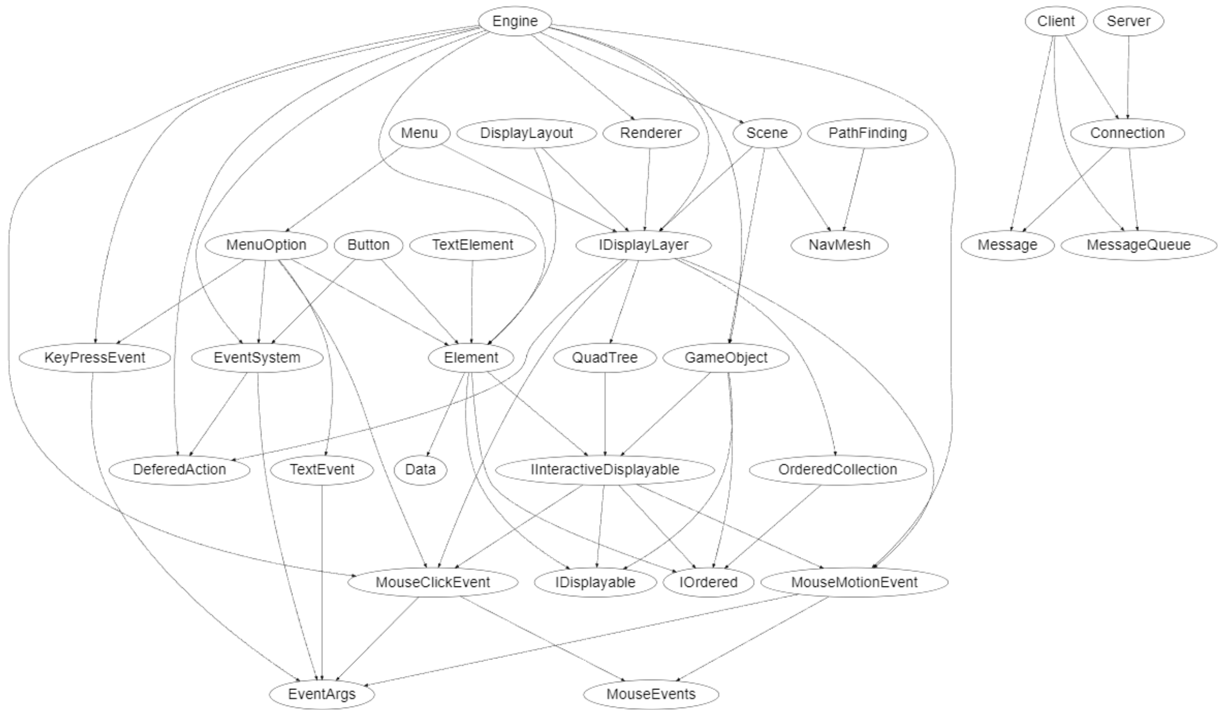


Figure 2: Final structure of the modules and dependencies of the Admirals source code. The graph shows all dependencies and relations except for the EngineContext and DataClasses classes since the relations involving them are too large to show.

As can be seen, there are many modules with many dependencies. At the bottom of the graph, we find various events, interfaces, and data structures. These are then used by other modules to implement something concrete, like a UI `Element` or a `GameObject`. These modules are then used by even higher-level modules, like menus, and scenes. Finally, at the highest level is the engine itself, which is what is exposed to the user.

The networking implementation is completely separate from the rest of the engine. This makes it more generic, and in theory, would allow it to be used for other projects. With the networking being separate, it is possible to use it without using the rest of the engine. This is useful if one wants to create a dedicated server for a game. This server should not need to run the game, and should instead only be concerned about its own function.

4.2 Interfaces and Classes

While interfaces do not exist in C++, we use the term to refer to (mostly) abstract classes that are inherited by classes that implement or support some specific behavior. These classes have a name starting with an uppercase `I`, like `IDisplayable` for example.

4.2.1 Top-Level Interfaces

The Engine makes use of a small set of top-level interfaces that guarantee some functionality of objects contained in certain collections.

The `IDisplayable` interface defines methods to render a displayable and access methods related to its bounding box. `IDisplayable` is inherited by the `IInteractiveDisplayable` interface that is implemented by `GameObject`s and UI `Element`s.

The `IOrdered` interface defines methods to access an order (or depth) and a unique identifier. It is used in `OrderedCollection`s, as discussed in section 4.3.3. It is inherited by the `IInteractiveDisplayable` interface, which is implemented by both `GameObject`s and UI `Element`s.

The `IInteractiveDisplayable` interface implements the `IDisplayable` and `IOrdered` interfaces as mentioned above, and also defines methods for handling engine loop events. It is implemented/inherited by `GameObject`s and UI `Element`s. It is used in both `QuadTree`s, and `IDisplayLayer`s.

The `IDisplayLayer` interface defines methods for a type of collection that can propagate engine events to its contained `IInteractiveDisplayable`s (displayables) as well as methods to modify what displayables it contains. It is implemented/inherited by `Scene`, and `DisplayLayout` and used by the `Renderer` and `Engine` modules.

4.2.2 Data Classes

To simplify calculations for positions, directions, colors, and other commonly used data types, we have created classes that can abstract some of these calculations.

The simplest example is the `VectorN` classes where `N` is the dimension of the vector, where the engine currently supports vectors of dimensions $N \in 2, 3, 4$. The `VectorN` classes contain `N` floating-point components and support arithmetic operations such as addition, subtraction, multiplication, and division with floating-point values or other `VectorN` objects. The `VectorN` classes also provide methods to obtain the vector magnitude and the distance to another vector/point.

The `Color` class represents a 4-dimensional RGBA color that extends the `Vector4` class and adds additional methods for converting colors from hexadecimal notation and 256-bit unsigned integer components to floating point values between 0 and 1. Converting to a canonical representation of floating point values between 0 and 1 is necessary since the Vulkan2D dependency relies on that specific representation for its colors.

The `Rect` can be used to represent areas and consists of four floating point components, two representing the width and height of the area, and two representing the 2D position of the area. The main use of this class is to group these two components in a single object and to provide methods to check if the area overlaps a point or another area (`Rect`).

The `Texture` class serves as a wrapper for texture objects from the Vulkan2D library. The main use of this wrapper is to provide a static method for loading textures from a file path and for unloading the texture when the object is deconstructed.

The `NavMesh` class is used to optimize navigation by reducing the need to calculate whether or not a move would result in a collision with another object. This is achieved by storing this information to enable re-use between multiple pathfinding calculations. It also allows further customization of the cost of each path (weighting specific moves over others).

4.2.3 Static Classes

Many classes provide static methods, but in this section, we will discuss the classes that will mainly be used statically by users of the engine.

The first such class is the `Renderer` class, commonly used in the `Render` method of classes that implement the `IDisplayable` interface. The `Renderer` class provides static methods to draw shapes, text, and textures on the screen. However, it is also used internally in the engine to initialize the rendering frameworks.

Another such class is the `Pathfinding` class, this class's only purpose is to find paths on a provided `NavMesh`. It has three public methods, one to find the aforementioned paths, and two to convert between different coordinate systems.

4.3 Objects, Elements, and Containers

The game engine has two main types of objects, user interface (UI) `Element`s and `GameObject`s, along with their respective containers.

UI `Element`s along with the `DisplayLayout` container are responsible for traditional user interface functionality such as displaying text and rendering clickable buttons, among other things. `GameObjects` and the `Scene` container are generally responsible for user-defined objects that can receive dynamic updates for each frame.

Both `Elements` and `GameObjects` are defined on the screen by their bounds stored as a `Rect`. `Element`s are defined to always be rendered on the screen, but Game Objects do not always have to be rendered. For example, one might define a Game Object that manages some functionality in the background, without needing to be rendered in any way. In the MVP, this concept is referred to as managers.

More detailed descriptions on `Element`, `GameObject`, DisplayLayout, and Scenes will be given in the sections below.

4.3.1 User Interface Elements

The engine provides a minimal set of features to be able to create practically any UI environment. The basis of this starts with what we call UI `Elements` (or just `Elements`), which are defined from the base class `Element` that define the functionality that all elements commonly implement. We have decided to separate elements from game objects since elements generally do not need some of the extra functionality that game objects have, such as performing updates every frame. Elements are also easily arranged on the screen, which is an added benefit since UI features are often placed in common locations, such as corners, or aligned in certain fashions. The `DisplayLayout` container therefore overrides the absolute position of individual elements to align them to specific locations defined by the user.

The implementation of specific elements is generally left to the user. However, the engine provides some elements out of the box: `TextElement` for rendering text, `Button` for rendering clickable buttons, and `Input` for text input. We feel that these specific implementations are often used in many contexts and should therefore be available out of the box.

4.3.2 Game Objects

Game objects are the type of objects that populate game scenes, they are positioned using absolute coordinates and have an update method that is called once each frame. This makes the game object highly dynamic, capable of changing its state over time as opposed to only in response to events. The implementation of game objects is entirely left to the user.

4.3.3 Containers

Containers are capable of holding a collection of displayables, such as UI `Element`s or `GameObject`s. The reason containers exist is to allow the user to easily group, apply changes, and propagate events. Each container implementation stores displayables in what is called an `OrderedCollection`. The collection orders elements according to a depth-order value that is defined for each object by the user. To solve the problem of what order objects should be rendered, the `OrderedCollection` sorts the contained displayables from lowest order to highest depth-order value. This makes rendering simple, as the objects are sorted in the same order as they should be rendered. The reason for having different container implementations (such as `DisplayLayout` and `Scene`), is because they handle engine event propagation differently and `DisplayLayout` overwrites the position of its contained elements.

A container propagates engine events to all displayables in its collection. For example, to facilitate handling mouse click events, all containers interested in mouse clicks use a quadtree to find objects at a given click position, which is described in detail in section 4.6.

Apart from the previously mentioned containers, the engine also defines a third container: `Menu`, which contains `MenuOptions`. A `Menu` is a more specific implementation of a `DisplayLayout` intended for rendering menus in a very specific way. `Menus` will be described in more detail in Section 4.5.1.

4.4 Event System

Two different types of events are defined and widely used inside the engine, Game Loop Events and Subscription Events, which we will go into more detail about in the following sections.

4.4.1 Game Loop Events

The first type of event is implemented as a core part of the game loop and will be referred to as "game loop events". Game-loop events are propagated from the engine through the display layers to the displayables they contain. These event methods receive a reference to the current engine context as an argument, and propagation cannot be stopped from the event methods. Since display layers can be toggled on/off, events are propagated only through the active layers. An example of a game loop event is the `OnUpdate` event, which is propagated every frame to allow game objects to dynamically change their state during the game loop. This event may commonly make use of the delta-time value in the engine context.

4.4.2 Subscription Events

The second type of events will be referred to as "subscription events" and works by allowing methods to dynamically subscribe and unsubscribe to an `EventSystem` object. The event system stores references to subscribed methods (event handlers) so that when the event is invoked, the

event handlers can be called. This type of event passes around a reference to the object invoking the event and an event argument object that can be modified by the event handlers. The event argument contains a flag that can be set to prevent the event from propagating to other event handlers.

An example of a subscription event is the UI button `onClick` event system class member. The button also has an engine loop event `OnClick`. The engine loop event notifies the button that it has been clicked on, the button can then invoke the subscription event to allow external methods to handle the click on the button. This allows users to have a single button class that can perform different actions depending on what event handlers are bound to the `onClick EventSystem`.

4.4.3 Deferred Actions

During the implementation of the MVP, we encountered issues such as seemingly random segfaults sometimes when changing active layers in the engine or unsubscribing event handlers from other event handlers. The issue was related to how the user of the engine could alter certain collections in the object while those collections were being iterated over as a part of handling events. For example, when iterating over currently active layers, a layer that is currently active could be made inactive, making the iterator behave in an undefined way. To fix this issue, we introduced what we call deferred actions.

Deferred actions mean that instead of immediately acting when the user requests it, the action is instead added to a list of actions that will be handled at a later time.

In the case of event handlers, the next time the event system is invoked, all deferred actions are handled before iterating the event handler collection. In the example of a deferred remove action, the actual unsubscription would occur the next time that event is invoked (before iterating subscribed event handlers). Similarly, in the case of layers, all deferred actions are handled only once in each frame. This allows the user to perform as many changes as they want during the current frame, but the updates will only take effect in the next frame.

Utilizing deferred actions in this way not only solves the problem with segfaults but also makes the engine behave more predictably, making its behavior less complex and easier to understand, which aligns well with the goals we have for the engine.

4.5 Graphical User Interface

4.5.1 User Interface Menus

As previously described, the engine provides a minimal set of features to create any type of user interface (UI) environment. The main building blocks for a UI are `Text`, `Button`, and `Input` elements. In addition to this, the engine also defines a system for managing and creating menus that can hold any number of the `MenuOption` element.

A menu is considered a container and works the same way as a Scene and DisplayLayout do. The only difference is that a `Menu` is rendered in a specific way, with the possibility of being altered by the user and having a title that is displayed above the menu. The specific way that a menu is rendered by default is by aligning options to the middle of the screen.

The engine provides several pre-defined `MenuOptions` that we feel are most relevant for creating basic menus. The options provided are: `TextOption`, `ClickOption`, `ToggleOption`, `CycleOption` and `InputOption`. These options can be extended or overridden by the user to create even more specific options.

Figure 3 shows an example of what a menu with four MenuOptions and a title looks like.

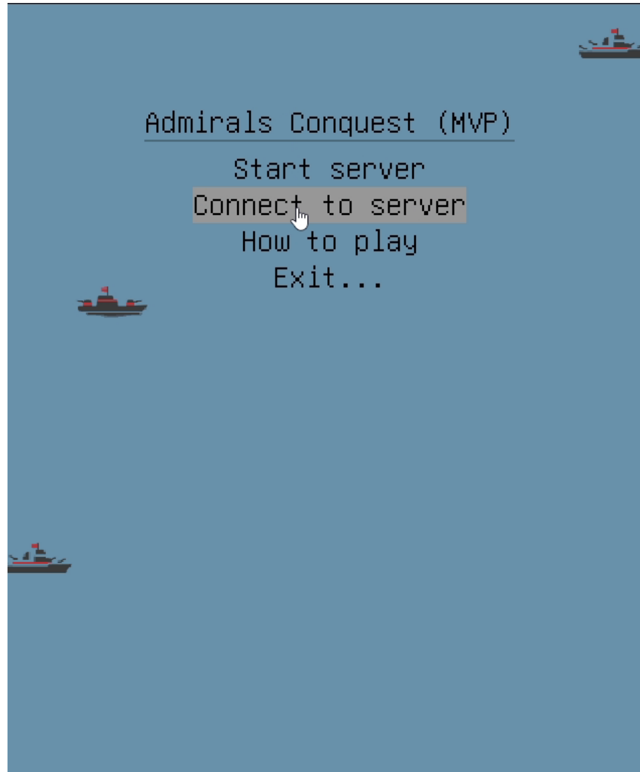


Figure 3: Menu with four different MenuOptions and title “Admirals Conquest (MVP)”

4.5.2 Hovering Effects

We define hovering over UI `Elements` as the mouse position currently being inside the element’s bounds. If multiple elements contain the mouse position, only the element on top with regards to ordering is considered to be hovered on. It turns out that supporting hovering effects is a non-trivial problem to solve, especially if the cursor has multiple ways of “unhovering” an element. It took us many iterations of implementation to get to a version that feels natural enough to be used in practice, which is what we’ll describe in this section.

We start by defining a set of events: `OnMouseEnter`, `OnMouseLeave`, `OnMouseMove`, which are called when the mouse enters an element, leaves an element, and moves inside an element respectively. These events are called with regard to differences between a list of elements that were hovered on previously and another list of what elements are currently being hovered on. For example, if an element was not previously hovered on, but is now, then the `OnMouseEnter` event is invoked.

Using these three events makes it possible to trigger the right actions when the mouse moves on the screen, which is handled by the display layout for elements. For example, when the

`OnMouseEnter` event is invoked for the button, the background changes to be more gray and the cursor changes to a hand instead of an arrow, as shown in Figure 3.

Implementing hovering effects is in its simplest form straightforward to do if the only way the mouse can enter and leave elements is by moving the mouse. However, this is not the case in many graphical systems, and our engine is not an exception. The main thing that complicates this is that the user can hide and show layers at their discretion. This means that elements that are hovered on can suddenly vanish and elements can also appear out of nowhere. To support this, the engine broadcasts an artificial mouse move event with the current mouse position to elements that have just been displayed so that they can take the mouse position into account, handling corresponding `OnMouseEnter`/`OnMouseLeave`/`OnMouseMove` accordingly.

Another thing that must be supported to have predictable transitions is handling `OnMouseLeave` event(s) before `OnMouseEnter` events. If this is not done correctly, transitions and, in extension, hovering effects may break. This caused us some tricky bugs because we previously handled `OnMouseLeave` events last since it was slightly more efficient.

4.6 Collision Detection using Quadtrees

Figuring out which displayables are clicked on during a mouse click on the screen is a non-trivial problem if you want a solution that scales with many objects and supports an environment where the game resolution or window might change during run-time. [1]

The engine implements a quadtree [4] that recursively divides the window into quadrants and stores data on which quadrants contain what displayables. Building a quadtree is straightforward if the data that is stored inside it are points (i.e. 2D coordinates). However, since displayables are defined from their bounds, which is a rectangle (origin, width, and height), the quadtree needs to handle this in a well-defined manner. To do this, we have added support for areas by deviating from a traditional quadtree implementation by adding the following rules when building the quadtree:

- A displayable will be included in all quadrants within which it falls, which often results in a displayable being in multiple quadrants.
- If a quadrant only contains one displayable, it will not be further divided (similar to points).
- If all displayables that are to be included in a quadrant entirely overlap it, the quadrant will not be further divided.
- Since overlapping displayables might end up in the quadtree dividing infinitely, there is a limit on the minimum width of a quadrant (which correlates to a maximum depth inside the tree).

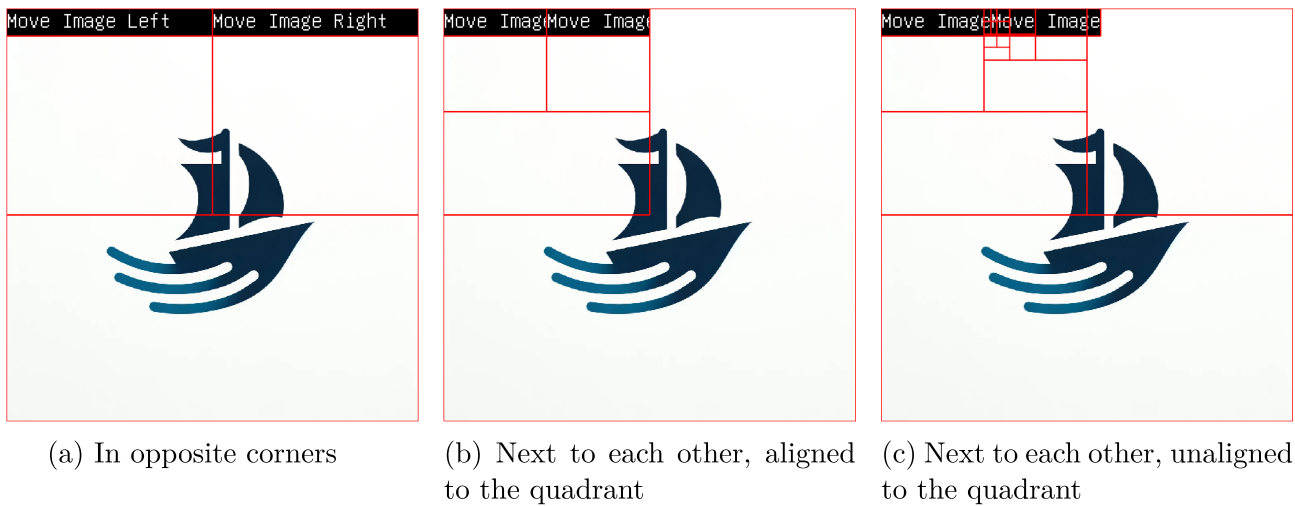
The quadtree is searched using a mouse coordinate, i.e. a `Vector2` (2D point). This always allows us to find a single quadrant in which the mouse coordinate falls and return a reference to the set of displayables that fall within it. Later on, the displayables placed inside the quadrant

[1] Initially, this problem had been solved by iterating over all displayables and checking if the mouse was within the displayable's bounds, and if it was, let the displayable handle the click from there. This is a naive solution that does not scale particularly well when the total number of displayables increases, since all displayables have to be iterated over to see if they contain the mouse's coordinates.

will have to handle the mouse click from there. Ideally, the number of returned displayables will be significantly lower than the number of total displayables for the quadtree to be useful.

4.6.1 Quadtree Examples

Figure 4 shows three examples of how the quadtree is divided into smaller quadrants depending on where the displayables are located. Figure 4a only contains two quadrants, as the two elements fit perfectly in the upper left and right quadrants, respectively. Figure 4b is divided to a depth of two, once in the upper left quadrant and again in the upper left and upper right quadrants. Figure 4c is a little more complex since the displayables are unaligned to the quadrant borders, which is usually the case since developers will not, and should not have to, account for quadtree borders. Since the displayables are unaligned in this unfortunate configuration, the quadtree will divide itself until the minimum quadrant width is reached, which in this case is 10 pixels.



(a) In opposite corners

(b) Next to each other, aligned to the quadrant

(c) Next to each other, unaligned to the quadrant

Figure 4: Two (2) displayables in different configurations showing how the quadtree is divided. Red lines indicate quadrant borders.

4.7 Pathfinding and Nav-meshes

To implement ship movement for the MVP we implemented the A* [5] pathfinding algorithm and nav-mesh objects containing data related to the cost of movement. The A* algorithm uses the nav-mesh to find a path to a given destination that minimizes the sum of the movement costs along that path. A nav-mesh is a 2D grid representation of a part of the world where each cell in that grid contains a value representing the cost of moving to that cell. The implementation is not specific to our MVP implementation and can be reused for non-grid-based games.

To create a nav-mesh we added a method to build a nav-mesh from the quadtree present in a scene and we made the cell sizes of the nav-mesh match the size of our game board grid. The nav-mesh queries each cell area in the quadtree to check if there are any overlapping objects and then uses a user-provided method to determine if that cell is navigable or not. The MVP uses the same cost for each navigable cell on the game board.

4.8 Networking

A networking implementation for a game engine needs to be broad to not limit the user in what they can create. Without making assumptions about the end game, the engine should support it. There are, however, some assumptions that need to be made regarding architecture and communication protocols.

We chose a client-server architecture, where a server is connected to all clients. This is the most versatile solution, as it allows the use of both dedicated servers and individual clients acting as a host.

Clients and the server communicate through messages defined by the user. These could be anything, from chatting to player movement. The engine provides various overrideable functions that the user can use for their implementation. These are:

1. OnClientConnect: Called for each client that connects to the server.
2. OnClientDisconnect: Called for each client that disconnects from the server.
3. OnMessage: Called for each message received to or from a client.

In addition, the server has functions to send messages to one or all clients, and the client has functions to send messages to the server.

How these messages are handled on both the server and the clients is up to the user. The engine only facilitates the message passing between these. The user simply overrides and extends the available functions, which will then be called automagically when the correct conditions are met.

Our implementation uses the ASIO C++ library [6], which is a cross-platform low-level networking library. It provides a consistent asynchronous I/O model, which allows us to handle networking concurrently with the rest of the engine. This also makes the networking platform-agnostic, allowing for example clients to be on a Windows machine, while the server is run on a Linux system.

We have chosen to use TCP as the transport protocol. This ensures that all messages are guaranteed to arrive but is slower than other alternatives like UDP. Since we are focusing on RTS games, consistency is vital. It is more important that all data arrive at the cost of speed than for most data to arrive some milliseconds faster. Desyncing clients would be a big issue, so all clients need to see the same state.

5 Minimum Viable Product

To showcase the built engine, an MVP (Minimum Viable Product) of a game was developed using it called “Admirals Conquest”. The game uses a grid-based layout with two players in opposite corners who will purchase and manage ship units to attack the other player. The goal is for the players to defeat the other player’s base of operation located on the opposite side of the map.

Figure 5: The MVP game, Admirals Conquest, in progress between two players

5.1 Engine Interactions

The MVP interacts with the engine by using the classes defined in the engine. Most importantly, it creates a `Engine` class instance which it initializes with data before calling its `StartGameLoop` method. The engine instance is initialized by adding `IDisplayLayers` containing appropriate displayables, to the engine. These initial displayables can then make changes to the engine and `IDisplayLayers` during the game loop.

5.2 Managers

A common paradigm to use when designing games is to use various managers who each handle a specific part of the game. In our game, we created the following:

Game manager: Handles the game logic and connects the other managers.

Network manager: Handles sending and receiving messages to and from the server.

Menu manager: Handles various menus and the actions performed on them.

Selection manager: Handles the selection of ships inside the game.

Animator: Handles the animations of the various displayed game objects.

This aids in the separation of concerns and allows the different managers to communicate using only higher-level functions, without caring about implementation details. For example, once a game has ended, the network manager receives the message, which then tells the game manager that the game has ended, which tells the menu manager to display the end-of-game menu.

5.3 Window Scaling

One limitation of the engine is that the game itself needs to implement how it should scale to the size of its window. For our MVP, we chose to render a 10x12 grid of cells (with the top and bottom rows being used to give room for UI `Elements`, and the 10x10 grid in the middle being used for the game board). As the window from which this board is viewed is re-scaled, the 10x12 grid will scale to fit, keeping the same aspect ratio and centering the board in the middle of the window as can be seen in Figure 6.

Figure 6: Window scaling of a wide screen in the MVP

Scaling the game board to fit the screen is a challenging issue to implement in the MVP. Engine features like quadtrees require the objects to have a bounding box with the correct position and size where they are rendered. However, the game itself is played on a 10x10 board, which means that we have to convert between two coordinate systems.

Our solution overrides the public getter for the `GameObject` bounding box to apply the correct conversion from the board coordinates each object stores as their position, to the world coordinates used for the quadtrees. Each object then uses this conversion in its render methods to draw the object where the quadtree believes the object is. To simplify this conversion for the game objects in the MVP, this is abstracted away in an abstract `GridObject` base class that the other visible game objects in the MVP inherit from. Meaning that the game objects inheriting from `GridObject` inherit the conversion between the different coordinate systems. Except then having to apply the opposite conversion when reading mouse positions or nav-path coordinates, this almost entirely obscures the conversion in the MVP.

5.4 Pathfinding

When the player wants to move a ship on the game grid they have to select the ship they want to move and the destination the ship shall travel to. How the ship moves is decided by a pathfinding algorithm that was developed in the engine and it checks which cells are occupied and which grid tile is a valid option to move to. The implementation of the pathfinding algorithm is described in more detail in Section 4.7.

The system also allows the player to select multiple ships to move to a grid tile. As multiple ships cannot be placed on the same tile the position of the ships is decided by which ship arrives first with the other ships placing themselves around the selected tile.

When a ship is set to move to its destination, data related to the move is sent to the server where its position is updated. The server performs secondary checks to see if the next grid tile for the ship is free or occupied. If the check fails the ship is not moved and the client is updated to stop the movement of the ship.

5.5 Ship Combat

For a ship to attack in the MVP, a targeting system is used. When a player wants to attack an enemy ship they have to select one of their ships and select the ship they want to target afterwards. The targeting system uses the same framework as the pathfinding and movement system to find valid targets and disallows a ship to be targeted if no path is found. As it shares the same framework it also allows the player to select multiple ships to target a ship. When a ship has a target it will move until the target is in range before initializing an attack. If the target moves away afterward the attacking ship will follow it to continue attacking.

When the ship units are attacking their action is handled on the server to keep both players in sync. The attack takes into consideration the ship's unique stats for damage dealt and the delay between attacks. First, all attacks by the ships are considered, and later the server checks if any ships are dead, in which case they are removed from the game.

To buy ships the player must accumulate enough gold. The gold is received passively from the server where it sends out a small amount each second and actively when the player places idle ships near "treasure islands".

5.6 Sprite Animations

The sprite animation in the MVP is handled by the Animator manager and is used to display an animation of game objects and background elements. The sprite animations are done by having each object to be rendered having a stack of pointers to multiple sprites. The Animator manager iterates over this list during runtime to switch between the frames of animation.

The sprites used to animate the objects in the MVP are fetched from the website OpenGameArt [7] where users can upload sprite art that can be used for free given the license is followed and credit is given to the artist.

The Sprites we have used are:

"Modified 32x32 Treasure chest" by Blarumyrran CC0: [8]

"Animated Water Tiles" by Sevarihk CC-BY 4.0: [9]

More sprite art has also been used that was created by Sevarihk and found on their website, Aurora-Sprites. [10]

5.7 Networking

The game builds on top of the networking functionality provided by the engine to implement various message types. Messages are sent by either a client or the game server and are handled according to their type.

After connecting to a game server, clients are marked as “waiting”. A client needs to send a “ready” message to be marked as ready. Once both players are ready, the server announces that the game has started.

The server broadcasts the complete state of the game many times each second. This includes the current gold of players, base health, and ship location, health, and current action. Since the game is small and simple, all this data can fit neatly into one message. For more complex games, this would not work and another way of communicating the current state would be needed.

All actions happen on the server. It is authoritative, and clients can only send which actions they want the ships to do. It is the server that decides if any action is valid and if it should be performed. This makes the clients simpler, as they do not need to handle any game logic. They simply show a visual representation of the current state and have the functionality to send actions to the server.

If a player were to disconnect in the middle of a game, the server can pause the game and wait for that player to reconnect. While a game is paused, all actions are ignored and the game state is frozen. Once the player has connected, the game is resumed from the same point. After a game has finished, the state is reset, and either the same or new players can start a new game by readying up again.

6 Project Management

All members are previously familiar with variations of the Scrum agile framework of project management and using Kanban-like tools (Trello, Jira, GitHub Projects, etc) and have used them successfully in previous collaborative efforts. Due to both familiarity and our wanting to spend as much time as possible creating a good piece of software, we have chosen to use a variation of Scrum that we feel adapts well to our goals and does not require us to spend time learning something new.

The Scrum variation that we have decided to follow consists of sprints that are one week long and without any distinct phases. The reason we have chosen to group sprints into phases is that we are working on many things in parallel and many phases would be overlapping, which would defeat their purpose. Sprints start and end on Mondays with a short standup each day that is generally about twenty minutes. This meeting has generally taken place at 13:15, except for Wednesdays when we have also met with the teachers to discuss progress, where we have instead had the standup in proximity to that meeting.

On Mondays when we have a sprint transition we have a purposefully longer meeting to reflect on what we need to do for the upcoming sprint and fill out our Kanban board on GitHub

Projects with tasks that we want to accomplish this sprint. In our GitHub Projects board, we have five (5) columns in which tasks are placed:

Backlog Tasks that are set to be completed, but not during the current sprint.

Sprint Backlog Tasks that are set to be completed during the current sprint.

In Progress Tasks that are currently being worked on.

In Review Tasks that are ready to be reviewed (most likely in the form of a pull request).

Completed All tasks that have been completed successfully.

The workflow has consisted of assigning yourself to a task that you want to work on from the Sprint Backlog and moving it to In Progress. A single member should generally not work on many things concurrently, so limiting how many cards a member is assigned to in the In Progress column is something we have adhered to.

The reason we have chosen to use GitHub Projects instead of other tools such as Trello or Jira is that it integrates well with also hosting our code and issue management on GitHub. The automatic features with merging and linking issues to pull requests make the process of using the board seamless and hence the choice of using it instead of Trello or Jira uncomplicated.

6.1 Version Control

We have used Git and GitHub for version control and hosted the code on a private GitHub repository located at https://github.com/joelsiks/admirals.

The process of working with GitHub is adapted from experiences we have that have worked great in the past. Each group member develops new features, fixes, and general additions to the codebase in separate branches from the main branch. When the code they have produced is ready to be reviewed, which should be done by at least one other member, they create a pull request, which is merged after any potential feedback is fixed and it is considered ready to be merged into the existing codebase.

Since our group is only four people we have often communicated verbally about problems during meetings. With this said, we have tried our best to write down reflections and problems in GitHub using issues or comments in pull requests. Our main reason for writing down issues is that it is much easier to remember what someone has said or thinks about something that way and because we can observe each other's work more easily.

6.2 Continuous Integration

To maintain a high standard and quality of the code we produce we have created a continuous integration (CI) workflow that performs certain checks before something is about to be merged into the main branch. If any of the CI checks fail, the code cannot (and should not) be merged before they are resolved.

The CI checks that we have are:

Code Formatting Check that the code is formatted according to our pre-defined style, which makes the code look consistent and well-formatted. Makes the code easier to read and understand, which facilitates maintainability and refactoring in the future.

Compilation Checks that each target defined in our CMake file compiles successfully. All code that is on the main branch should work and therefore, compile.

Another check that we would have wanted to include in the CI, which could potentially be added in the future, is to check that the linting does not produce any errors. Linting, which we'll talk more about later, is a topic that requires us to define a specific set of rules to consider, which has been hard for us to agree upon.

7 Tools

In this section, we will describe what, why, and when we have applied certain tools to improve the quality and consistency of our code in several ways. A common feature of the tools we have used is that they all integrate well with the development environment we use, Visual Studio Code, making their use and application seamless.

7.1 Formatting

As previously stated, making the code look consistent and well-formatted makes it easier to both read and understand, which also facilitates its maintainability and refactoring potential in the future. Having a well-defined way in which the code is formatted also removes any discussion of what is right or wrong between members, given that we all have settled on an agreed-upon style.

To format our code, we have used a tool called clang-format which performs formatting using a set of rules. We have chosen to base our style on a predefined style and change only a few aspects that we just like more. In the end, this boils down to personal preference and the style we have chosen is what we think looks the best. Listing 1 shows the entire set of rules that we use.

```
BasedOnStyle: LLVM
IndentWidth: 4
# Indent "public/private" in classes by nothing (negative of IndentWidth)
AccessModifierOffset: -4
# Must be 80 characters or less!
ColumnLimit: 80
# use \n instead of \r\n
UseCRLF: false
```

Listing 1: The rules for formatting used by clang-format.

Many editors, specifically Visual Studio code, provide a feature for automatic formatting when saving a file. This has resulted in us rarely having to format the code using terminal commands, making applying formatting an uncomplicated endeavor.

7.2 Linting

Linting is a process in which the code is statically analyzed to find potential issues, errors, or inconsistencies in the code. We have decided to use a linter for two reasons in our project: first, to make sure that we follow certain naming conventions for class-member variables, and second, to root out any potential issues or errors that are easily missed.

To perform linting we have used a tool called clang-tidy which performs linting with regards to a set of existing rules from which we select a subset that we are most interested in checking. The reason that a subset is selected is that clang-tidy provides a large number of rules for many different purposes, many of which are not interesting for our codebase and should therefore not be included. Finding what rules to use and what rules not to use has been hard since it requires knowledge about what rules exist. To help us to create an initial subset of rules we have used ChatGPT, where we provided information about our code and some hints of what qualities we are most interested in. From the initial subset provided by ChatGPT, we have continuously added and removed rules as we figured out which are interesting and not. However, this process is still ongoing, which is the reason we have not decided to use this in our CI stack.

All in all, using linting through clang-tidy has increased the quality of our code and made it more consistent in ways that only using a code formatter could not.

A feature of clang-tidy that we are particularly satisfied with and want to point out is its ability to check that class-member variables are prefixed in a certain way. We use this to ensure that private class-member variables are prefixed using “m_” so that they are easier to detect when reading code. Listing 2 shows the clang-tidy rule that performs this check.

```
CheckOptions:
  - { key: readability-identifier-naming.PrivateMemberPrefix, value: m_ }
```

Listing 2: Class-member variable prefix rule for clang-tidy.

In Visual Studio Code, linting is integrated well enough that problems are shown in a separate tab for the currently open file, making it easy to detect and fix problems. An alternative would be to run the linter in the terminal, open the file in which a problem is found, fix it, and re-run the linter to check that it is resolved. In general, using clang-tidy with Visual Studio Code has worked great, apart from it being slow in some cases, which could easily be resolved by closing and re-opening the entire editor.

8 Results

In this section, we will include tables and graphs from profiling, discussing how much time is spent on certain parts of the engine: rendering, event handling, and rebuilding quadtrees. Additionally, we will also discuss what features the final version of the engine has in connection, and what goals have been fulfilled.

8.1 Profiling

The engine has been profiled through the MVP, where a short game session has been played between two players. The Callgrind tool, which is part of the Valgrind suite, has been used to measure the number of executed instructions when running the. We have chosen to perform profiling on Linux instead of Windows because Linux natively supports using Callgrind and KCachegrind for interpreting the results, which Windows does not.

Additionally, we have chosen to not include profiling results for the initialization of the engine, such as loading textures, creating a window in the operating system, and initializing Vulkan functionality. The reason for this is that the initialization will be different on every machine the engine is executed on and takes a variable amount of time depending on what else is running

on the system, the hardware configuration, and the amount of data being loaded. Instead, we will include results from the engine's game loop, which is relative to other parts being executed and thus gives a relevant and interesting result to analyze and interpret.

Figure 7 shows a table containing the profiling results of the functions called inside the engine's game loop (referred to as `Engine::StartGameLoop`).

Ir	Ir per call	Count	Callee
60.39	5 212 890	33	admirals::Scene::Update(admirals::EngineContext const&) (mvp: Scene.cpp)
26.45	2 282 931	33	admirals::Engine::PollAndHandleEvent() (mvp: Engine.cpp, ...)
10.47	903 632	33	admirals::renderer::Renderer::Render(admirals::EngineContext const&, std::vector<std::shared_ptr
1.82	235 095	22	admirals::UI::menu::Menu::Update(admirals::EngineContext const&) (mvp: Menu.cpp, ...)
0.67	57 863	33	admirals::Engine::HandleDeferredLayerActions() (mvp: Engine.cpp, ...)
0.21	49 147	12	admirals::UI::DisplayLayout::Update(admirals::EngineContext const&) (mvp: DisplayLayout.cpp, ...)
0.00	3 505	1	admirals::Scene::OnStart(admirals::EngineContext const&) (mvp: Scene.cpp, ...)
0.00	49	33	admirals::renderer::Renderer::GetWindowSize() const (mvp: Renderer.cpp, ...)

Figure 7: Profiling results of `Engine::StartGameLoop` showing the percentage of time taken, number of instructions, number of calls, and function name.

Figure 8 shows a graph containing the most time-consuming calls inside the game loop and what takes time inside the sub-calls of those calls.

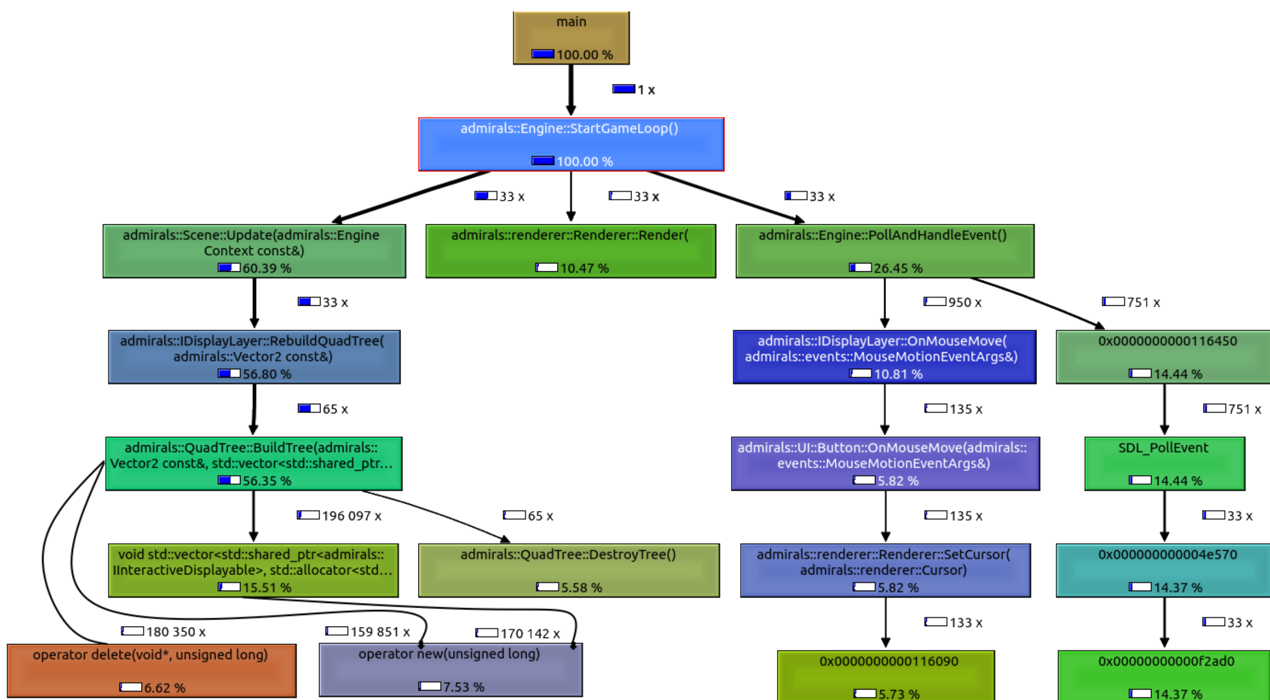


Figure 8: Profiling results of `Engine::StartGameLoop` with sub-calls included. The addresses starting with `0x0000` are shared-library translation calls and should be ignored.

8.2 Features

The final version of Admirals fulfills all the functional requirements defined in Section 2. It is possible to create different game objects and UI `Elements` and to store these in scenes and menus. The containers can be swapped at any time, allowing the user to control the flow of their

game. These are rendered by the engine according to the user's specifications. Additionally, the engine supports networking, allowing the user to send arbitrary messages to/from different endpoints.

With all the requirements/features, it is possible to implement (almost) any arbitrary game in the engine. We proved this by creating Admiral's Conquest, a two-player online RTS game, described fully in Section 5. This game uses all of the engine features in its implementation and builds on top to add additional game-unique features. This shows that Admirals is a capable engine, able to be used when creating games.

All of the non-functional requirements were also fulfilled but to various degrees. The engine is well documented, with thorough instructions on how to build and include in projects. If the user is comfortable with C++, the engine should be easy to use, but we have no way of verifying this. Some functionality has dedicated tests, but others lack them. However, Admiral's Conquest tests all functionality and acts as a large integration test to verify that all modules work together.

9 Discussion

In this section, we will discuss the results presented in the previous section as well as other areas which are interesting to note.

9.1 Performance

We have spent significant time developing and fine-tuning the way that mouse-click events are handled, which are handled using quadtrees in the final version of the engine. In the profiling results, as shown in Figure 7, we can observe that about 94% of the time inside `Scene::Update` is spent on destroying and rebuilding the quadtree so that it is always up to date with the latest changes in positions of game objects. Interesting to note is that creating and deleting shared pointers inside the quadtree (operator delete and operator new) takes up about 24% of the time inside `Scene::Update`. Overall, the performance increase of using a quadtree instead of a naive solution is still better but could use improvement.

Regarding `Engine::PollAndHandleEvent`, about 55% of the time is spent inside SDL's `SDL_PollEvent`, which we can't improve on. The rest of the time is almost exclusively spent on handling mouse-move events, which typically are called often since mouse-moves occur extremely often.

All things not listed in the profiling results in Figure 8 take up less than 2% of the time inside the game loop and are therefore not as interesting to look at. What the results have shown us, which we were already suspicious about, is the significant time spent on quadtrees. For now, this performance hit is acceptable and does not impact the way that the MVP is played. However, for the future, the quadtree should ideally take up less time, perhaps through the use of caching and more infrequent updating of quadtrees.

9.2 Application Testing

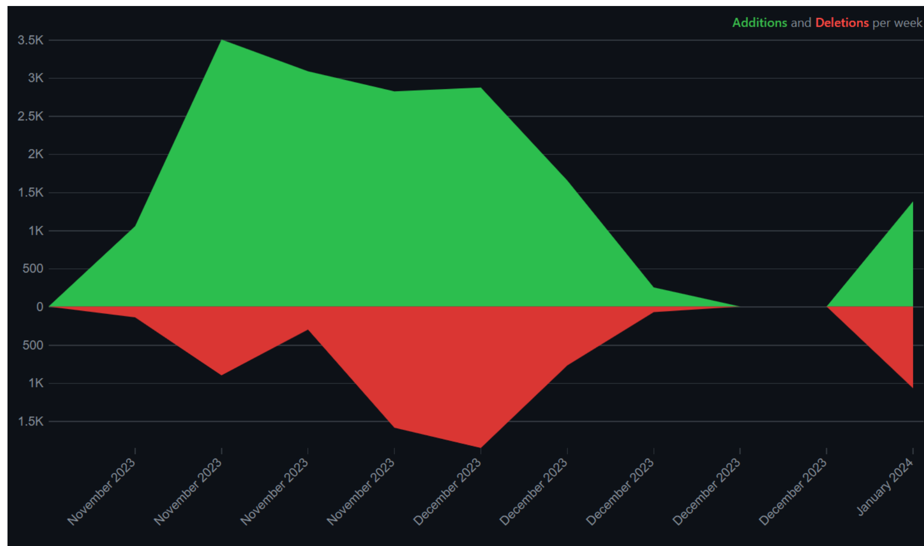
Our project is a bit different as the main deliverable is the game engine itself. With this in mind, it is through the MVP that has been developed that the engine is most easily tested.

The plan was to expose the MVP, and in turn, the engine, to be tested by someone without knowledge about the inner workings of the engine. However, this has not been done because the people responsible for this failed to respond in time. Instead, the tests performed were from our internal perspective, with knowledge and insights into the application. With this, we can draw some conclusions, but it is not as conclusive as external impartial testing can be.

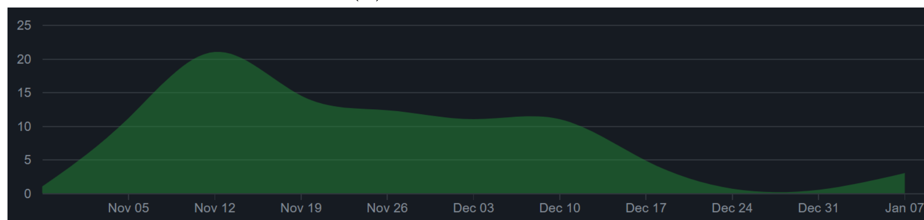
During the development of the MVP, we continuously used and tested the engine's features and discovered bugs along the way. In our regular workflow, we would implement some system in the MVP and then make changes to the engine to make that system possible or fix bugs discovered along the way.

9.3 Division of Work

At the beginning of the project, we decided that everyone would put in as much work as they were willing and able. This was necessary since some group members had not yet found a thesis project to work on and needed additional time for such activities. Because of some of this, the distribution of work is not as even as it otherwise could have been. Those of us who were able to work hard in the beginning compensated for the lack of time of the other members of the group. This being said, much of the work was completed during the beginning of the project, as can be seen in Figure 9. After the initial weeks, we had to decrease the amount of work to a more reasonable level, while also spending more time fixing bugs and finding issues with existing code. This resulted in a steady decline in the amount of new code written following the peak in week 45 (November 12 in the graphs).



(a) Code frequency



(b) Number of commits

Figure 9: GitHub insights data from the Admirals project.

Because of how we distributed work, some group members worked more than others. However, we managed to stay ahead of the initial time-plan throughout the project. The amount of work in a project such as this is substantial, and we were therefore unable to focus on some parts that we were initially interested in, to enable us to complete the core game engine. One such part was the multiplayer aspects which we initially thought we would dive much deeper into.

9.4 Ease of Use

One of our goals for the engine was for it to be easy to use by an experienced user in C++. We have defined easy to use as the user should not need to add long or complicated sets of code to accomplish anything. The user should only have to call a single command if they want to add or remove something and not worry about the performance of the engine during runtime. The user should neither have to know how the code of the engine is written nor have to look up how certain functions or methods work in detail.

With the current version of the engine, it is possible to use it by including the engine in the main file of the game and initializing it by creating a scene and adding game objects to that scene. While the setup of the engine could be smoother with multiple parts of the engine needing to be initialized it is easy to add game objects when it is initialized as it would only require adding a couple of lines of code.

While the process of setting up and creating a game with the engine is easy the documentation for users regarding potential help if they need it is lacking. This was mostly due to time constraints and lack of foresight.

9.5 Engine Dependencies

The game engine has three dependencies: Vulkan2D [11], SDL2 [12], and ASIO [6].

Vulkan2D is an external library by Paolo Mazzon built upon Vulkan and SDL2 used to render and display objects. Vulkan2D is built to require no prior knowledge of Vulkan and be easy to use so it was a good option for our engine as we did not want to spend much time on developing and working with rendering.

Simple DirectMedia Layer 2 (SDL2) is a cross-platform software development library that provides developers access to hardware components and manages video, audio, input devices, etc. In our engine we don't use it directly but through the library Vulkan2D. Without this, the engine wouldn't be able to listen after inputs from the user nor display the game itself.

ASIO is a cross-platform C++ library for networks and provides a consistent asynchronous model for developers. ASIO is used in the engine to establish connections between different users and set them as client and server. Without this dependency, it would not be possible to have multiplayer as we envision.

9.6 Open Source Contribution

As part of developing the engine and using the Vulkan2D library by Paolo Mazzon, we had the opportunity to contribute to it by fixing a bug. The bug was related to calculating the number of images in the current Vulkan swapchain.

Fixing the bug allowed us to continue using the Vulkan2D dependency the way we wanted to and contribute to the open-source community simultaneously. The pull request for the fix/contribution can be found at: https://github.com/PaoloMazzon/Vulkan2D/pull/10.

10 Future Work

There are a multitude of different options to be considered for future work. What we feel are the most relevant and likely to be worthwhile features will be discussed below. No future work will be considered for the MVP as it is intended to only showcase what can be accomplished using the engine.

10.1 Usage Examples and Guides

To make the engine easy to use, it is important to provide extensive documentation and examples of the various parts of the engine. Due to time constraints and a failure to include this in our initial time plan, this has been left to be implemented in the future. Especially, this includes documentation in the form of usage examples and descriptions of wide-spanning features.

10.2 Performance

The performance of the engine can be improved in several ways, the first change to consider is the parallelization of the engine game loop. This could drastically improve the performance of larger numbers of objects. This was not implemented because the MVP did not require a large number of objects, and the complexity of such a feature was too large for the available time for the project.

Ideally, as mentioned in Section 9.1, quadtrees should be made more efficient, perhaps by utilizing caching measures and figuring out ways in which rebuilding quadtrees can be done more infrequently. This would shift available resources to the game developer to perform the game logic instead of the engine being too "bulky". Another, more complex way, in which to increase performance is to make building the quadtree more efficient, perhaps through reducing the number of memory allocations. In Figure 8, we can see that the vector allocator is called almost 200 000 times during 66 rebuilds of the quadtree.

Another change to consider is to reduce the usage of shared pointers, and instead move to manual memory handling. This change can improve performance by reducing unnecessary reference counting.

10.3 Graphical User Interface

To make the engine more user-friendly, a graphical user interface similar to that of Unity or Unreal Engine should be developed. This is a substantial task that could be a project all on its own.

10.4 Additional Features

Many features could be added to the Engine, one feature we have considered is the option to change the icon of the game window. This could be easily solved by adding additional dependencies, but to keep the project lightweight, we made the choice not to add these dependencies and instead leave it to be implemented in the Engine directly in the future.

11 Lessons Learned

When developing the engine each member focused mainly on the parts they felt were most interesting, which meant that certain parts of the engine received more work than others. For example, Joel was really in developing GUI functionality, so he spent a lot of time on that, while Casper was intrigued by networking and spent most of his time around that. In the end, this worked in our favor since the parts we were interested in ended up with high-quality design and functionality. Additionally, since we did not have strictly set responsibilities, we had reoccurring discussions on what needed to be worked on at sprint transitions, which worked great for us.

Throughout the project, we have worked on things in parallel, which required us to continuously create and review pull requests on GitHub to resolve conflicts and also to ensure code quality and consistency. We feel that spending time on pull requests has worked to our advantage and has been time well-spent for us.

Not using Visual Studio when doing a C++ project on Windows poses several problems. This meant downloading Linux-based tools on Windows, performing profiling on Linux, and using a Linux-based compiler instead of MSVC. If we were to start a similar project in the future that is as Windows-orientated as this game engine was meant to be, we would consider to instead use Visual Studio instead to make setup and development much easier.

Although this project was meant to be a culmination of our studies and for us to "show off" what we have learned so far, this project turned out to be a great learning experience nonetheless. Some of us were unfamiliar with C++ and our proficiency in it improved throughout the project, learning new concepts along the way. We also learned how important a proper development environment is to speed up development.

References

[1] Unity Technologies. (2023) Unity Documentation. Accessed 2023-12-14. [Online]. Available: https://docs.unity.com/

[2] Epic Games. (2023) Unreal Engine 5.3 Documentation. Accessed 2023-12-14. [Online]. Available: https://docs.unrealengine.com/5.3/en-US/

[3] Pygame Community. (2023) Pygame. Accessed 2023-12-14. [Online]. Available: https://www.pygame.org/docs/

[4] Wikipedia. QuadTree. Accessed 2023-12-07. [Online]. Available: https://en.wikipedia.org/wiki/Quadtree

[5] Geeks for Geeks. A* Search Algorithm. Accessed 2024-01-08. [Online]. Available: https://www.geeksforgeeks.org/a-search-algorithm/

[6] C. Kohlhoff. (2023) Asio C++ Library. Accessed 2023-12-07. [Online]. Available: https://think-async.com/Asio/

[7] OpenGameArt. OpenGameArt. Accessed 2024-01-07. [Online]. Available: https://opengameart.org/

[8] Blarumyrran. Modified 32x32 Treasure chest. Accessed 2024-01-07. [Online]. Available: https://opengameart.org/content/modified-32x32-treasure-chest

[9] Sevarihk. Animated Water Tiles. Accessed 2024-01-07. [Online]. Available: https://opengameart.org/content/animated-water-tiles-0

[10] Aurora, Sevarihk. Aurora-Sprites. Accessed 2024-01-07. [Online]. Available: https://aurora-sprites.wixsite.com/main

[11] PaoloMazzon. (2023) Vulkan2D. Accessed 2023-11-03. [Online]. Available: https://github.com/PaoloMazzon/Vulkan2D/tree/master

[12] SDL community. Simple DirectMedia Layer - Homepage. Accessed 2024-01-07. [Online]. Available: https://www.libsdl.org/