# AGE Engine

Henry Jiang

December 11, 2021

## Contents

**1 Introduction** **1**

**2 Overview** **1**
2.1 ECS . . . . . 1
2.2 Event System . . . . . 2
2.3 Application Context . . . . . 2

**3 Design** **2**
3.1 Entity Component System . . . . . 2
3.2 Application Context . . . . . 3
3.3 Event Systems . . . . . 4
3.4 ASCII Render System . . . . . 5
3.5 Scene Lifecycle . . . . . 5
3.6 User Design . . . . . 6
3.7 Other . . . . . 6

**4 Extra Features** **6**

**5 Final Question** **7**

# 1 Introduction

My implementation of the The AGE Engine is based off an ECS (Entity Component System) paradigm as the base engine with the ncurses library to implement the window management, displaying ASCII text, and keyboard user inputs. Under the ECS pattern, game entities are not of a single class type but defined by an entity with properties of components that where actions are performed upon by the systems the components are defined under.

# 2 Overview

The AGE Engine implementation can be broken down into the following 3 major components: Entity Component System, Event System, and Application Context System. Starting off with the ECS (Entity Component System),

## 2.1 ECS

Starting off with the ECS (Entity Component System), the ECS manages all game entities created throughout the games and how they behave based on Components and Systems defined for them. In our case, and Entity is a unique identifier, uint64_t in my implementation, and each entity can have different properties associated to define their behaviors. For example, each entity can have a Transfrom component to describe their position, scale, and rotation or they have a Velocity component and a Gravitation component to describe how each property of the entity should behave; and a system is

defined to modify a collection of entities that have the required components, in the case above, a Gravitation system would add the appropriate acceleration to the velocity component.

### 2.2 Event System

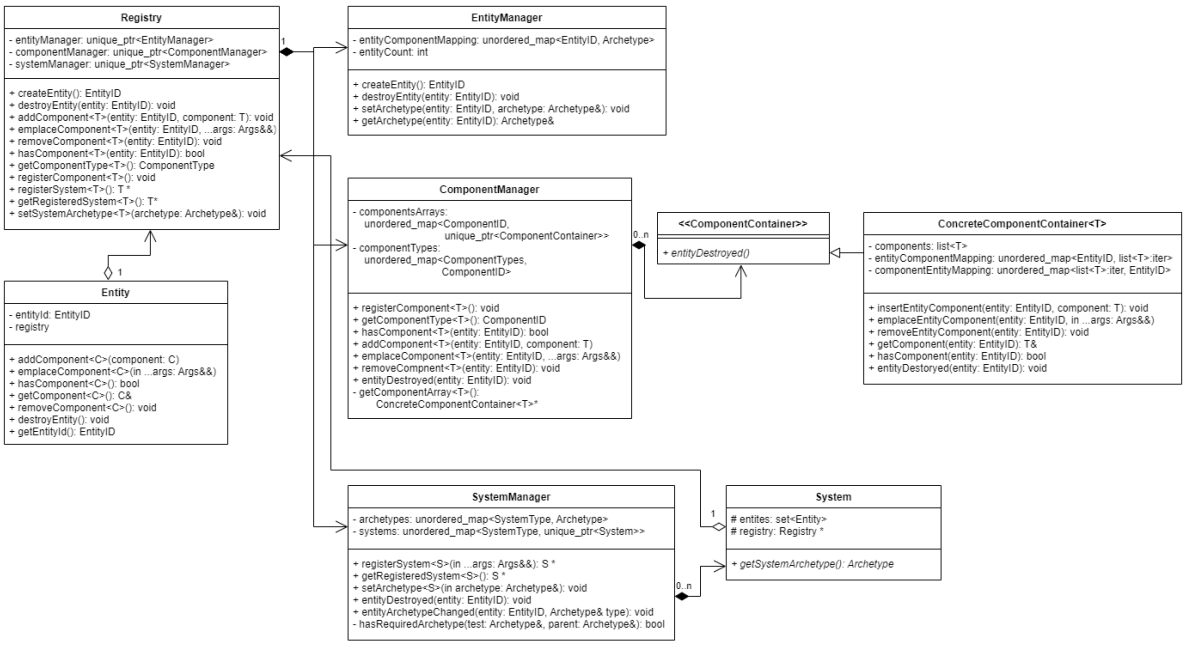
The engine needs a way of translating external events and actions that is accessible to the game and a way for different systems with in the engine to communicate with each other and the outside world. Thus the event system provides an good interface for the job. The Event System is back by the EventQueue class which is implemented as an Publisher/Subscriber pattern, where Publishers publish events to the EventQueue and Subscribers can register and unregister themselves from listening to events dispatched. Subscribers can simply pass a lambda function or class member function as a subscriber to the EventQueue. Another important distinction is that the EventQueue internally contains a queue which dispatches events all at once using the `dispatchEvents()` member function (all engine events are published per game tick).

### 2.3 Application Context

Another important aspect that the age engine must control is the application life cycle. The engine life cycle has 3 phase. Initialization, Active, and Shutdown. Without it, the user may use the engine in unintended ways that would be detrimental to themselves. Thus the ApplicationContext Interface the engine offers encapsulates the details of engine initialization and shutdown to the interface while leaving the user a closed but extensible way to modify application behavior, or not all by using the provided context.

## 3 Design

### 3.1 Entity Component System




My implementation for the ECS uses an array to store components contiguously. Each part of the ECS is implemented separately under their respective managers: EntityManager, ComponentManager, and SystemManager; and a common interface class to the ECS is the Registry, which contains all methods required to perform core actions, such as creating entities and adding components. The EntitySystem stores the currently alive entities in a hash map which also contains the archetype of the entity, where the archetype is simply a signature for which components are currently associated with the entity and is stored as a vector. Next, the ComponentSystem contains separate ComponentContainer

for each type of Components. Currently, the ComponentContainer is implemented as a std::list of components which does not offer good cache locality, thus a different implementation could be provided in the future for containers with more efficient access under the ComponentContainer interface. Finally, the SystemManager contains all registered system. Each System class contains an container of entities that have the appropriate archetype.

The difficult in implementing this systems is that the ECS must store different types of component class and be able to access them efficiently. Thus my design of storing the typeindex of the components offers a middle ground in terms of performance and extensibility. Instead of performing heavy dynamic casting on the stored derive class of Component, to get the correct reference, which would offer extremely poor cache locality, I went to a system that hashes typeindex of the component classes to allow static casting at the trade off that component sub-classing would not be possible. i.e. a subclass of a base Component would be considered two different components instead of the is-a relationship.

The final design decision I made was to use the Entity Class encapsulate the interface method of the Registry Class to simplify working with components. For example, with parameter packing and forwarding, we can simplify and encapsulate the process and use the Registry class as a mean to create entities.

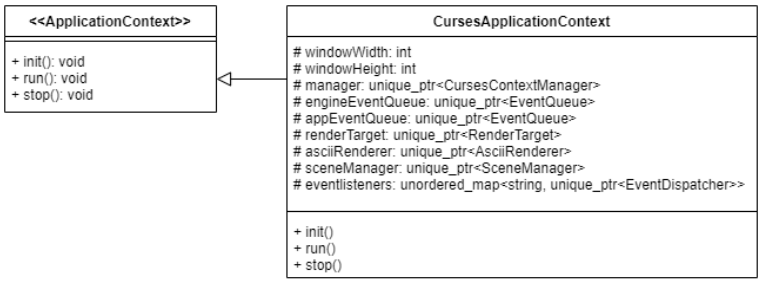
```
registry->addComponent<Component>(entityID, Component(10)); // old syntax
entity.addComponent<Component>(10); // new syntax
```

## 3.2 Application Context

The ApplicationContext provides a clean interface for the user to control the application life cycle by implementing the 3 pure virtual lifecycle, `init`, `run`, and `stop`. In order to achieve this, engine is compiled as a static library with the main function included. Thus, the engine can control the application by having restricting access to the user. But the user can still specify possible extensions to the engine by deriving from the pure virtual ApplicationContext by providing the function exposed by the engine.

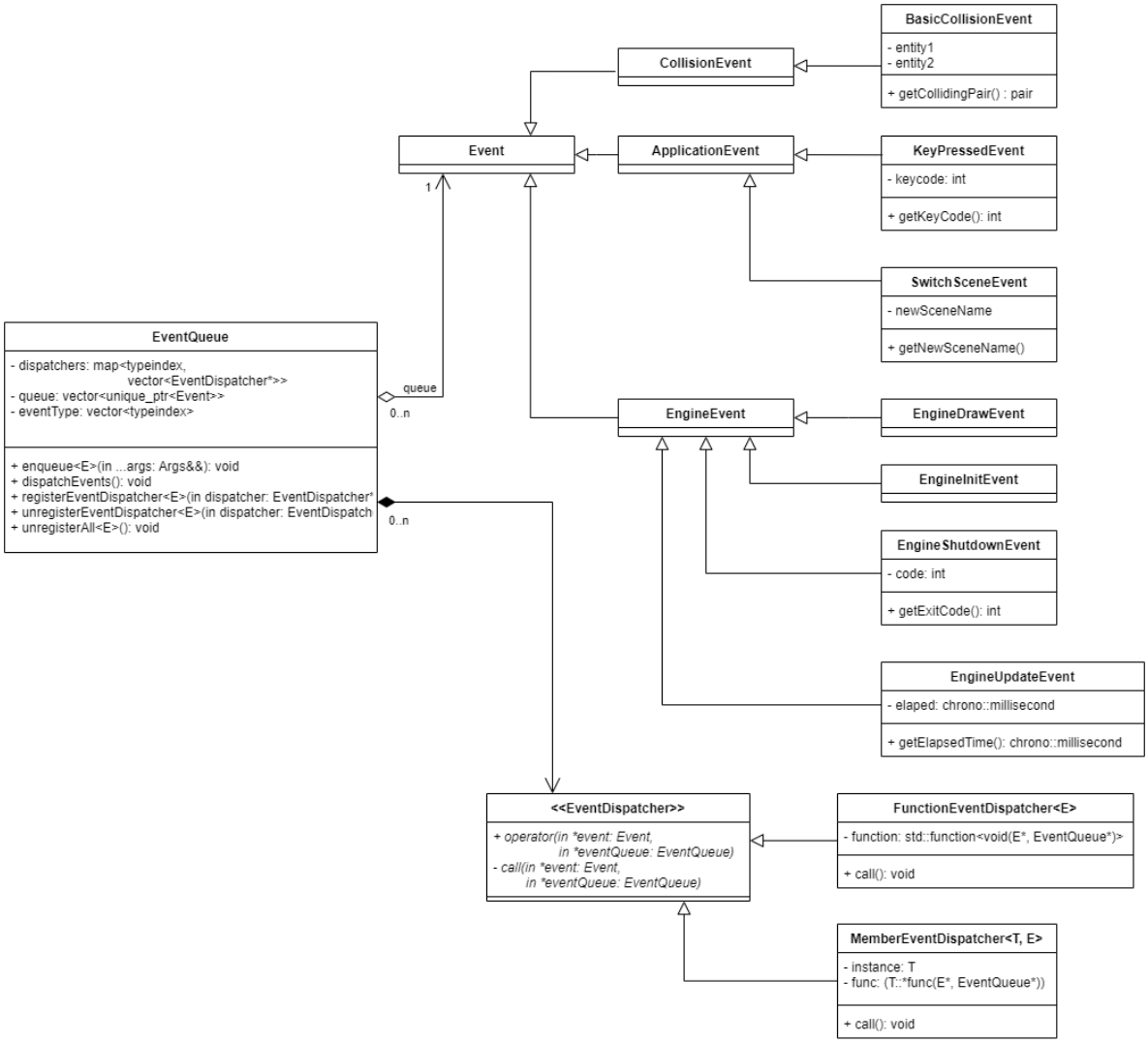
```
extern std::unique_ptr<AGE::ApplicationContext> gameEntryPoint(int argc, char *argv[]);
```

As a result, if no additional features of the engine is required, then the default CursesApplication-Context will be used which provides a basic game loop that dispatches EngineDraw and EngineUpdate events every game tick of 500ms, and a basic ncurses window at a fixed dimension (80, 25). This is the best approach to have an closed but extensible application life cycle. For the sake of this assignment, the default would be the CursesApplicationContext which contains the basic requirement of the project as seen in the UML.



## 3.3 Event Systems

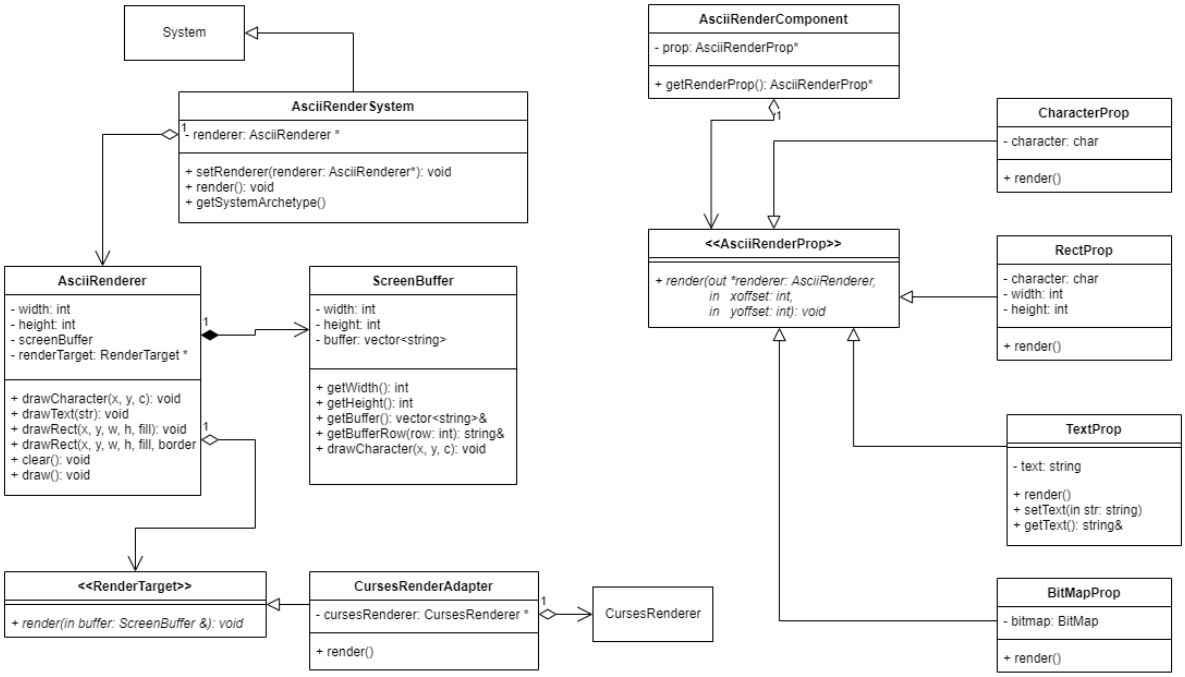
Every application needs a decoupled way to perform internal communication and handling external events. Thus the `EventQueue` class provides a way for different systems through the application to communicate with each other, and allow handling of events originating from the engine and outside.




It follows a strict Pub/Sub pattern as the Observer pattern requires the knowledge of the subject. However, in the engine environment, it doesn't matter who is dispatching the event, all the subscriber knows is that there was an event published. Similar to the ECS, `EventQueue` also uses `typeindex` to `static_cast` to the desired derived event class. Publishers can emplace events to the internal queue, where it is stored until the member function `dispatchEvents()` is called. Then, the queue iterates through all stored events, and looks up the corresponding `EventDispatcher` and calls it. Of which, the `EventDispatcher` can either be a class member function or a lambda function.

Another important job of the Event System is to decouple game logic and rendering from the game loop. Thus in the standard `CursesApplicationContext` update and render are decoupled from the game loop by dispatching the `EngineDrawEvent` and `EngineUpdateEvent`, and `CursesApplicationContext` also polls for keyboard inputs and converts them to `KeyPressedEvent` in the application event queue. Moreover, ECS Systems have access more `EngineEvent` such as `EngineShutdownEvent` to exit the application. As a result, systems can subscribe themselves to listen to these events without the game loop having knowledge of these systems, making the entire system more extensible and as decoupled as possible.

## 3.4 ASCII Render System

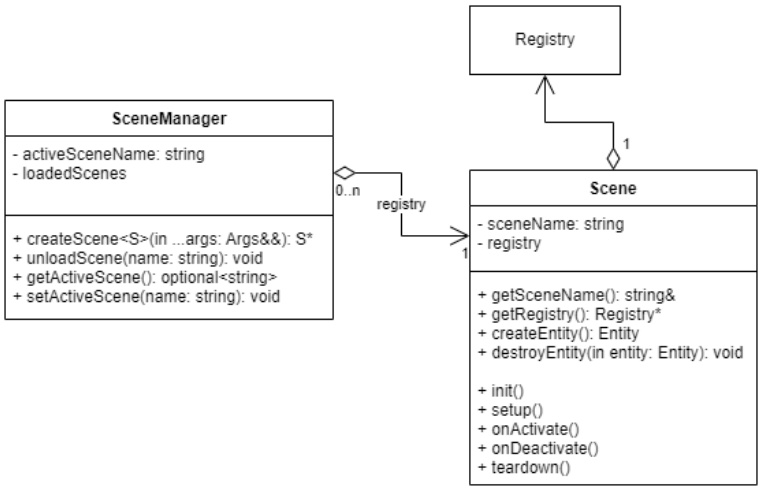



Since the engine core implements an ECS System, it is quite natural to provide the render as part of the ECS as a System that acts on entities with the `AsciiRenderComponent` component. Starting off on the `AsciiRenderer`, it implements the actual draw mechanics. It has the basic `drawCharacter`, `drawText` and `drawRect`, from which everything else can be drawn using the above interface. The renderer stores an internal screen buffer to be drawn on, which abstracts away what the renderer is drawing to. Here, the Adapter pattern is used, as the target would be an ncurses window. Thus, by implementing the `RenderTarget` interface, we can implement the target as the base terminal through `std::cout` or other means. In addition, this systems can be extended easily. For example, to convert it to use 3D renderer, the user can define an `3DRenderSystem` that acts upon entites with `MeshComponents` for example. Next, under the dependency inversion principle, `AsciiRenderProp` provides an interface to the renderer for which type of objects an entity should draw by providing an implementation to the pure virtual `void render(AsciiRenderer* renderer, int xoffset, int yoffset)` method.

## 3.5 Scene Lifecycle

After organizing game objects into entities, we have a final project to handle. How do we separate these entites and systems. The answer is the `Scene` class.

A `Scene` is the first logical division of a game (i.e. a game can be divided up to multiple scenes) is a collection of resources, entities, and logic that functions together.




As such, the `Scene` class contains all the logical elements required. The base class must contains entities, so it contains an instance of the ECS under registry, and addition resources management and event listeners can be added to derived classes of `Scene`. Next, the `Scene` life cycle is important, and there are 5 of them. `init` initializes all `Scene` object contents, `setup` places entities and game objects in the scene, `onActivate` triggers when the scene becomes active and should registers event listeners to listen to events, onDeactive gets called when scene is no longer the active scene, and finally `teardown` runs before the scene is destroyed at the end of its lifecycle.

Next, the life cycles of a scene can be difficult to manage. So the `SceneManager` have a simplified interface to create, destroy, and activate scenes, and each of the methods manages and calls the core life cycle methods on the `Scene` object.

Furthermore, Entities with the Scene have access to the `EventQueue` to the application and the engine, where it can create the `SwitchSceneEvent` under `ApplicationEvent` to trigger the `SceneManager` to switch active scenes.

### 3.6 User Design

The goal of the engine is to provide a set of classes to implement a game within a closed but extensible system. To make a game, the user first must use the default `CursesApplicationContext` to create an application context or derive an enhanced version of the application context. Next, the user must create the Scenes the game is composed of and register the appropriate components and systems belonging to each `Scene`. The user must derive from the `Scene` base class to prove their own implementation for the `setup()` method to create game entities to be placed in the `Scene`. In addition, more components and systems can be created to suit the user's needs, but an extensive base set of them are already provided by the engine implementation. Next, the user can implement addition events relevant to the game they are writing, such as `GameOverEvent` to signal a win condition.

### 3.7 Other

Another problem that the engine has to handle is managing resources by ncurses. Therefore, a wrapper is necessary to create and manage the window, and getting keyboard inputs. The class `CursesContextManager` follows RAII and follows the singleton pattern to ensure that there is only every one instance of the ncurses context, and calls the appropriate ncurses library functions to setup the window and keyboard functionalities and the appropriate function when destructing.

## 4 Extra Features

With the ECS systems, any feature is possible to implement, but due to the limited time, they are not implemented.

The class `EnhancedCursesApplicationContext` offers higher resolution windows, variable tick rate and render fps.

# 5 Final Question

What would you have done differently if you had the chance to start over?

I don't think that I would've done anything differently design wise. I believe that by having the ECS, EventQueue, and the Application Context, it is possible to implement any type of games. However, given more time, the implementation side could definitely be improved. There are different places where a better implementation could be provided, but they can all be implemented down the line due to their interfacable nature with relatively low effort. However, there are some minor design choices that can be improved. For example, the `Scene` lifecycle can be improved, and the `ApplicationContext` class has all 3 life cycle as virtual, thus making any new derived class cumbersome to override. Next, the ECS include structure could have been done better with by using more pointers and forward references to reduce compile time. However, overall there are not much that I would've done differently.