The `AGE`-Engine Documentation

Introduction

The `AGE`-engine is a simple 2D console game engine which runs in UNIX operating system. It provides a considerable amount of C++ APIs for clients to build and create a 2D console-based game that runs in UNIX.

The engine not just providing a fix plan for a window layout, it also gives abilities for clients to create their nested windows due to the solid design of the engine itself.

The engine is following a software architectural design named Model-View-Controller (MVC). The engine's APIs can be easily grouped into three different categories where each of them serves some certain functionalities.

The engine is also built under a software architectural pattern named Entity-Component System (ECS). Clients can create their entities with different functionalities during the runtime. An entity is a composition of different components and by the meaning of a component, is just a bunch of raw data that describes certain abilities. Moreover, it gives possibilities for clients to enable and disable any functionalities during the runtime which gives the maximum flexibility.

The next section will discuss and explore on the architecture of the `AGE`-engine in more details.

Overview on Architecture

This section will explan the two core ideas to this engine, one is MVC, and the other one is ECS where both of them are described in the introduction section.

What is a Window?

The engine has its own definition about what a window is. A window is simply defined by four measurements: An ID, A position, dimension and subwindows. An ID is a UUID which is namely Universally Unique Identifier which describes the window uniqueness. A position describes the left-top point of the window itself. A dimension describes the width and the height of the window. The subwindows is a list of its child windows that are nested inside the current window.

The engine provides a default game layout. Except that, a user may also define their own custom window layout. Moreover, the window layout may even be changed or detached a subwindow during the runtime which gives flexibility for the user to create more possibilities.

Moreover, the engine provides two types of window: A window with controller and a window with camera.

Window with Controller

Note that, a window does not necessarily needs to have a display view functionality, instead, a base window is more like an abstract idea. By understanding that, the fist type of window can by easily understand: a window which has functionality that takes the user input.

Window with Camera

This is the actual general type of window that shows the actual view of everything. This type of window renders and also a wrapper that handles all the game logic. A window with Camera has a concept called `Scene` which will be discussed later in the More-Deeper-into-the-AGE-Engine section and it is the core wrapper to stores and handles the logic.

One thing that needs to be clarified is that, a window with camera is not the same concept as the view part that would apear in the MVC pattern. The actual view part is done by a system called render system that will be discussed in the Entity-Component System section.

Model-View-Controller

Controller

The `AGE`-engine gets the raw input from the user through a controller during the game loop and a user does not has ability to override the behaviour of a controller. However, an entity may not have two same type of component at the same type and components can the user can definenot be re-assgined to a different entity.their own behaviour during each game frame according to the input. One thing that needs to be clarified is that, the `AGE`-engine does not handle the user input itself. All the input are defined by the users. The exception is `ctrl+c` which will let the user to quit the program immediately.

Model

The concept about a model in the MVC can be found by a lot places in the engine itself. A window can be treated as a model (stores data), the whole Entity-Component System can be also treated as a model (stores data and handles logic), the concept of system that will be discussed in the next section will also be considered as a model.

View

To understand what is the actual view part in this engine is kind of tricky here. Like what was mentioned in the previous section, a system called render system is the actual view: takes data, renders them, done. Render system only handles logic. Moreover, not just render system, but all the systems are only responsible for handling logic but not data.

Moreover, the `AGE`-engine provides a window buffer that wraps the third library `ncurses` APIs. It is considered as a part of the view and the render system will send the output directly to the window buffer.

Entity-Component System

The highlight of the whole `AGE`-engine is the Entity-Component System that I implemented. This section will discuss in details about the three concepts on entity, component and system.

Note that, the efficiency may not be as fast as a typical ECS will do. But fast enough in this case. More details on the efficiency issues will be discussed in the More-Deeper-into-the-AGE-Engine section.

Component

A component is the raw data, thats it. It is the basic unit of everything. It provides data for some certain functionalities but not directly handles the logic. The clients may implement their own components very easily for extension purpose.

Entity

An entity is essentially an ID. It is an ID that describes an entity which is a composition of different unique components. You may treat an entity as a list of components as well. In fact, every Entity will be assigned with a UUID (Universally Unique Identifer) once it has been created. The client may add or remove component during the runtime. However, an entity may not have two same type of components at the same time and components cannot be re-assgined to a different entity due to the time limitations.

Since an entity is just an abstract representation of an ID. As a result, an entity does not have inheritance hierarchy.

Registry (Entity Manager)

The core framework in my ECS. An registry can be considered as a query system that stores all the entities and sort them for fast querying. The systems (will be discussed soon) will ask (query) for entities directly through its own registry. An entity may be destroy or add into an registry during the runtime. The client has zero controlling on a registry. This is considered as the core framework of the ECS.

A registry is not unique and it is corresponding to a scene which will be discussed soon.

System

System is the opposite concept to the component: systems only handle the logic. An example of a system is render system, it asks (querys) an registry for all the entities that has a renderer component. The system renders each entity by its own renderer component.

A system can be considered as two types, logic systems and view systems. Each type of systems are done is different stage of the game loop.

Scene

Every window with camera has a coressponding scene, a scene is just a wrapper for mutiple systems and a single registry. It gives ability for clients to handles different game logic in different game window.

User Guideline

The `AgeEngine` class

The class `AgeEngine` handles everything about the engine logic. It controls the game loop, timer, all the game logics and rendering functionalities. It is the everything about the engine. A client needs to inherit from the provided class and override for two certain methods to have controls over different functionalities: `init()` method will be invoked during the beginning of the game, user can set up the game in this function. `onEachFrame(int input)` will be invoke for every game frame, the input will be `ERR` if no inputs is receiving, client may define their own user input by overriding this function.

Moreover, the default constructor of `AgeEngine` will provide a default game layout which is exactly like the pdf shown layout, which the wrapper window is named `GameWindow` and it is a `WindowWithController` which takes the input from the user. It consists of two subwindows named `StatusWindow` and `BoardWindow`, both of the subwindows are `WindowWithCamera` where the clients have full controls over these two windows.

If the clients want to create their custom window layout, they may provide a new `WindowWithController` that has other subwindows and pass them into the `AgeEngine` constructor.

Note that, the `AgeEngine` only accepts a custom overall window with `WindowWithController` type, that is, it only support a single controller.

The `AgeEngine` has its own nested `Timer` class for controlling a fix frame per second (FPS).

Things about Controllers

A cilent has some certain controls on the behaviours of a controllers. `WindowWithConroller` provides APIs for clients to switch the current controller. Clients may create their own by inheriting from an abstract class `Controller`.

Window Construction

An abstract base class `WindowModel` has two implemented methods, `renderViews()` and `updateLogics()`. Each of them will invokes each of their subwindow method `onRenderView()` and `onUdpateLogic()` respectively. That is, it is a recursive behaviour describes how a nested window works during the render and game logic processing.

Clients may inherits from two concrete classes `WindowWithController` and `WindowWithCamera` to create their own custom window type and both of them are inherited from a base abstract class `WindowModel`.

If necessary, clients may also inherits from `WindowModel` to create custom window type and it can be also considered as a subwindow type as well.

When inheriting from `WindowModel`, clients need to override the virtual method named `onUpdateLogic()` and `onDrawView()` if necessary. These functions are invoked during `WindowModel::updateLogics()` and update `WindowModel::renderViews()` processes.

Entity-Component System (ECS)

Clients may create their on components and systems by inheriting from provided classes.

However, `Entity` class cannot be inherited since it was suppose to be considered as an ID. Inheriting an ID does not make any sense.

Details on `Registry`

A `Registry` not just handles all the entities, but moreover, when a new `Component` is added into an existed entity, since the adding process is calling through the registry's API, `Registry` will grouping entities by their different components.

For an instance, entity `e1` has two components: `c1,c2`. Entity `e2` only has one component `c1`. When systems querying from an registry for component `c1`, it was actually asking for all the entities which obtains the component `c1`. That is, it will return `e1,e2`.

To achieve the above behaviour, I decide to mark each component type with a unique ID (note that it is the type having the ID, not a concrete instance of a component). Then using mapping from component type to a list of entity pointers. Everytime an new component is adding to an existed entity, the mapping will be updated.

During my own implementation of the registry, I looked up a third library named `entt` which is a mordern C++ library implemented using `C++17`. As a consequence, I also implemented my own `Registry` using `C++17`.

Custom Components

Clients may inherits from a base class named `Component` to create a custom component. Note that a `Component` should only stores the data but not handling the logic.

`Component` may be nested using reference for fast logic handling purpose.

The `AGE`-engine also provides a concept named `GlobalComponent`. Is is a base class and not having a IS-A relationship with `Component`.

`GlobalComponent` does not bind to any specific `Entity`. As the name described, it should be considered as an global data package that every system may queries for that.

To construct and query a global component, calls `Registry::emplaceGlobal()` and `Registry::queryGlobal()`.

I implement such a feature simply for convinence purpose where registry can stores some informations as a global values.

Render System

This is the actual view part of MVC. It querys for all entities which obtains `CRenderer` component from an registry. Using the Painter's Algorithm to complete each frame draw. That is, it sorts all the entities by their altitude from low to high using priority queue. Then renders each entity from low to high as well.

Since each `CRenderer` component has a reference to an abstract class `CTexture` component, which has a method named `paint()` which describes how to actually render the texture into the window buffer.

Since we are using priority queue, the time complexcity is `O(n*logn + m*n)` where `n` is the number of entities and `m` is the number of pixels to be filled. The space complexicty is `O(n+m)`.

Painter's algorithm has its own advantages and disadvantages.

It is definitely memory efficency by considering it does not requires extra information to be stored during the renderer process.

However, it is considerably taking more process power since all part of the images are rendered even if they are covered by other images.

It also have certain limitations when rendering situations like cyclic overlapping.

The render system may be improved using the Reverse Painter's algorithm and which will be discussed later in the final-question-section.

Motion System

It querys for all entities which obtains `CVelocity` and `CGravity` components respectively from an registry. Iterates all the relevant entities and simply updates them using some simple math.

Collision Sytem

Similarly, it querys for all entities which obtains `CCollidable` component from an registry. Each `CCollidable` has a reference to a `CBoundingBox`, which handles the actual collision detection logic. The engine already provides a type of bounding box named `CRectBox` which is essentially a rectangle-shape bounding box.

Clients may inherits from `CBoundingBox` to implement their own bounding box such as circle box or angled rectangle-shape box as long as the user provides their own collision detection logic.

Once an collision happens, the behaviour for each entity is defined by the component `CCollidable` using a vector of function pointer that points to different callback functions. These functions will be invoked once the collision happens. The first added callback will be invoked first. The collider's functions will be invoked before the collidee's functions.

The collision system is using the simplest algorithm to detect collision: comparing each entity. The time complexcity is `O([n(n+1)]/2) -> O(n^2)`.

I was aiming to implement quadtree at the beginning but due to the time limitation, I just simply decide to use the simplist algorithm for completion.

Moreover, the collision system is using discrete collision detection instead of continuous collision detection which may causes some weird behaviours once the entity moves to fast or the FPS is running too slow.

The `Utility` namespace provides some helper functions to consturct four invisible border surrounding the window to achieve the solid board view.

Despawn System

To achieve the despawn feature, I simply implemented a new system for convinence: Iterates all the entities which obtains the component `CDespawn` and check if they are outside of the screen, if they do, increment their tick count by one. Once the tick count reaches the maximium, we destroy that entity.

Custom Systems

Clients may inherits from abstract classes either `LogicSystem` or `ViewSystem` to create their own custom system. `LogicSystem` is processing automatically during the function `WindowWithCamera::onUpdateLogic()` and `ViewSystem` is processing automatically during the function `WindowWithCamera::onDrawView()`.

Once a custom system is written, you may call `Scene::emplaceSystem()` to construct your system into a `WindowWithCamera`. To get a reference to a scene, call `WindowWithCamera::getScene()`.

Entity Construction

The only valid way to create an `Entity` is by calling `Registry::create()` which will construct an empty entity and stores inside the `Registry` and the function returns a reference to it (registry manages the lifetime of an entity).

To add a new `Component` to the `Entity`, calls `Registry::emplace()`.

To remove a `Component` from an `Entity`, calls `Registry::remove()`.

To destroy an `Entity` from a `Registry`, calls `Registry::destroy()`. Or, marks the `Entity` as disabled and calls `Registry::refresh()` to destroys all the disabled entities at once.

Extra Credit Features

The Entity-Component System may be considered as a extra credit feature itself. It is not as easy as it is since to ahiceve a fast query enough, the inner design and algorithm should be welly considered. Moreover, I used a lot of C++17 features and features that are not mentioned in the course notes for completion which is really chellenging. However, the system still works perfectly since I strictly follows all the good design pattern and code technique that I have learned.

Moreover, I supported not just a fix window layout, but nested windows and clients may even provide their own window layouts and even detach any windows during the runtime.

As a consequnce, the window size is also not restrictly locked, it can be client-side customized.

The `AGE`-engine also provides a higher FPS (60 FPS) version of the game.

Final Question

- `quadtree` for fast collision detection. If it is too hard, I may use `Sweep and Prune` instead.
- `Continuous Collision Detection`.
- `Reverse Painter's Algorithm` which is more time efficency.

Conclusion

Everything I said is true. I just didn't have enough time to show my true power of my engine through the demo.