Software Architecture and Design Assignment 2

Software Requirement (FPT University)

Software Architecture and Design

ASSIGNMENT 2

GROUP 3

Vu Sy Thanh Nam - HE140717

Nguyen Giang Nam - HE141508

Bui Thi Thao - HE141701

Luu Van Nam - HE140788

This document is about our subject Software Architecture and Design. It brings our knowledge and insight into Game engines. In this document we will answer questions about game engine, it's design, besides introducing multi thread game engine, 2 model MVC and PAC in game engine design. Also we have an idea for a game and we used an Object Oriented Design process and game engine to design our game.

I. Overview

1. What is a game engine?

For every game, game engine plays a major role since the game engine helps the game designers to bring characters of the game to life, by helping in scenes, characters and graphic generation, sound, artificial intelligence, scripting animation, networking etc.

A game engine is a software framework designed for the creation and development of video games for consoles, mobile devices and personal computers. It turns the complex task of game development simple, by providing an abstraction layer, which makes a lot of big tasks look very easy, while the game engine does all the hard work in the background. In other words, it is a framework that is designed specifically for the construction and development of video games.

So, developers or game lovers who find interest in diving deep into game development, for exploring new technologies and showcasing your imagination in the form of game or if someone is just curious to know what's happening behind game engine

technology, here's a detailed explanation of the components that constitutes a game engine.

One of the most important features of a game engine is platform abstraction, which guarantees that games are able to ship to multiple platforms with a minimal change of source code.

2. Components of Game Engine

a. Input

A game is nothing if it cannot be played, the game engine provides with support for an array of input devices like mouse, gamepad, touch etc while also providing support for devices like gamepad, joysticks etc.

b. Graphics

Game engine provides a lot of features like lighting effects, shadow, bump maps, blending animation etc to make the imported asset look real.

c. Physics

There is a sub-component of the game engine, which is known as Physics Engine. Physics engines are software which allows performing fairly accurate simulation of most of the real-life physical systems like the movement of rigid body (we will perform that practically in later chapter using Unity 3D), soft body mass and velocity alteration and fluid dynamics, bounciness etc.

d. Artificial Intelligence

Now-a-days, Artificial Intelligence is playing a significant role within the game development. Knowing the kind of weapons the player will be using based on the situation or the behavior of the player gets recorded and actions are performed accordingly, can be done using specialized software embedded into the games. The implementation of AI in games is usually done using readymade scripts that are designed and written by software engineers who are specialized in AI.

e. Sound

Audio and Rendering Engines are a sub-part of the Game engine which are used to control the sound effects and generate 3D animated graphics in your 2D screen. They provide a software abstraction of GPU using the multi-rendering API's like Direct3D or OpenGL for video rendering and API's such as Open-AL, SDL audio, X-Audio 2, Web Audio for audio.

f. Networking

Since a decade now, games support online multiplayer and social gaming, which connects your gaming adventures with your friends. Most of the gaming engines, provide

complete support and scripts for such requirements, so you do not have to worry about TCP/UDP traffic, social API integrations etc.

3. **Some game engine design**

Popular game engines are Unity, Unreal Engine, and Game Maker Studio and they each have online communities, abundant documentation, and resources to help you learn.

3.1. Unity

Unity is an all purpose game engine that supports 2D and 3D graphics, drag and drop functionality and scripting through C#. Unity is particularly popular for mobile game development and much of their focus is on mobile platforms.

3.2. Unreal Engine

Unreal Engine is a favorite for developing the most graphic-intensive games. It allows you to create 2D, 3D, VR, AR, cross-platform, single-player, or multiplayer games. It is a state-of-the-art engine and editor with photorealistic rendering, dynamic physics and effects, lifelike animation, robust data translation, and much more-on an open, extensible platform that won't tie you down.

3.3. Game Maker Studio

GameMaker Studio is one of the most recommended engines for beginning game developers. It is primarily used for making 2D games, but it is possible to make 3D games. You can export your games to multiple platforms if you have purchased the correct license. GameMaker Studio allows for drag-and-drop programming, which is a great option for beginners. They also have a custom Game Maker Language (GML) for more advanced programming.

4. **Unity 3D Engine**

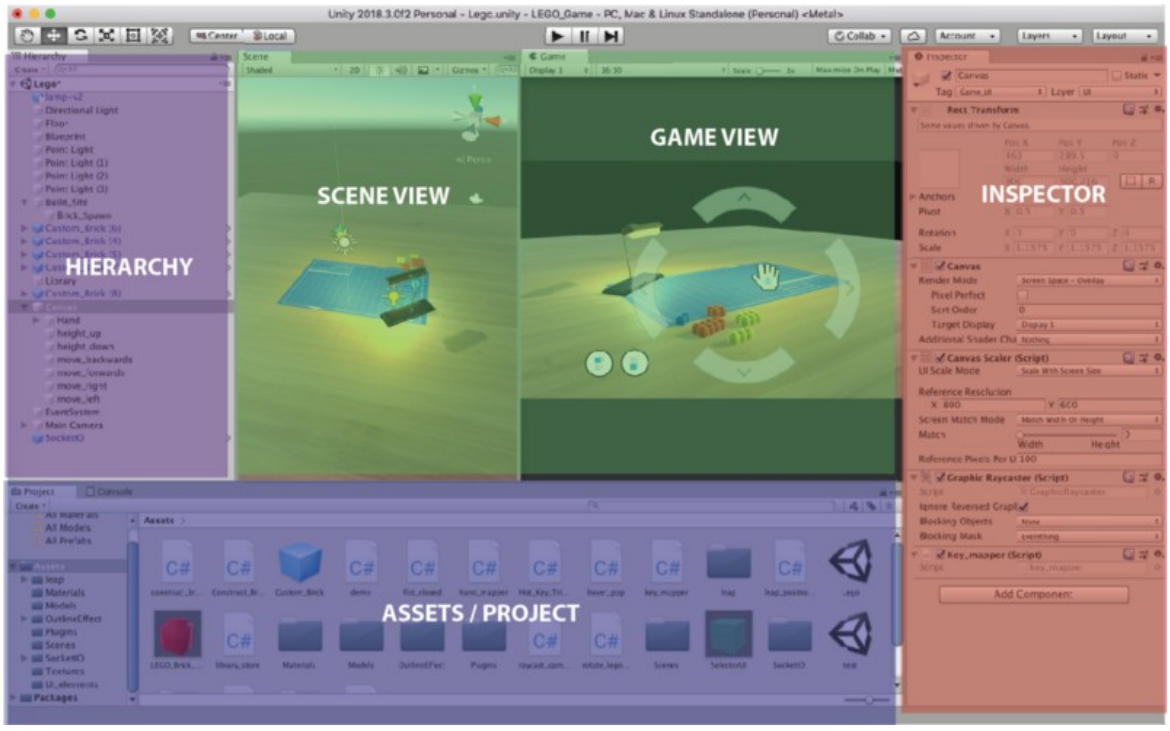
The Unity engine is C++ based, and scripting in Unity uses either C# or JavaScript. The user-generated code runs on Mono version 3.5 or the Microsoft .NET Framework 3.5. Unity allows testing of a game in development without the need to export or build. Debugging is done with MonoDevelop which can be launched within Unity.

Workflow within Unity adopted the classic Object-oriented design, where every entity in game is an object; but it also enforces a component-based architecture, where properties are defined by components and a game object is merely a container of different combinations of components. Each component is a self-contained function that performs a specific task. Components can be built-in components or generated from user-created scripts. Component-based architecture allows functions in different domains (e.g., physics, rendering, sound, and AI), and to achieve decoupling of code while increasing reusability of code.

The engine targets the following **graphics APIs**: Direct3D on Windows and Xbox One; OpenGL on Linux, macOS, and Windows; OpenGL ES on Android and iOS; WebGL on the web; and proprietary APIs on the video game consoles. Unity also offers **services** to developers, these are: Unity Ads, Unity Analytics, Unity Certification, Unity Cloud Build, Unity Everyplay, Unity IAP, Unity Multiplayer, Unity Performance Reporting and Unity Collaborate.

Multi-thread is supported in Unity script, while Unity APIs, because they are not thread-safe, can only be called from the main thread. When expensive or long-term operations are being computed in Unity, threads can still be useful (for example, AI, pathfinding, network communication, and file operations).

Unity Interface:



Scene View: This is where you will be creating level for your game, scene or 3D project. All of your Game Objects will be placed and manipulated right here.

Game View: This is where you will see your results, how your level or scene looks like. You need to have a Camera on the scene to see how it looks like. Sometimes its called Camera View.

Hierarchy: This window will display all Game Objects placed directly on the scene. Basically everything that you see in Game View, needs to be listed here. This will include non-visual and visual game objects.

Project: This is your project window. Basically it show what's inside Assets folder on your disk. Everything from Game Objects, Scripts, Textures, Folders, Models, Audio, Video and etc… will be accessible from this window.

Inspector: This panel will display different attributes and properties of selected Game Objects. Depending on the selection, the appropriate attributes and components will be listed.

5. **Multithreaded game engine**

Multi-threading means that the program is parallel, or that it has to perform multiple independent actions at the same time. If the actions are not independent then the execution will not really be parallel. In the worst case that the dependencies enforce sequential execution, performance will be much worse than the sequential version.

The first and most classic way of multithreading a game engine is to make multiple threads, and have each of them perform their own task. For example: you have a Game Thread, which runs all of the gameplay logic and AI. Then you have a Render Thread that handles all the code that deals with rendering, preparing objects to draw and executing graphics commands. Thread are synced at each frame, so if either of them is slow, both will be slowed as the run at the same time.

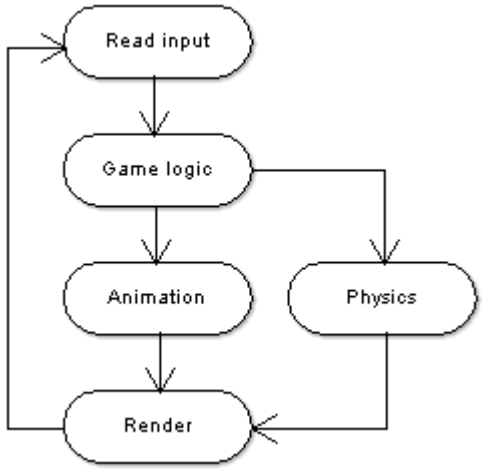
Unreal Engine 4 or **Unity 3D** has a Game Thread and a Render Thread as main, and then a few others for things such as helpers, audio, or loading. The Game Thread in Unreal Engine runs all of the gameplay logic that developers write in Blueprints and Cpp, and at the end of each frame, it will synchronize the positions and state of the objects in the world with the Render Thread, which will do all of the rendering logic and make sure to display them.

Multithreaded Game Engine Architectures

There are two main ways to break down a program to concurrent parts: function parallelism, which divides the program to concurrent tasks, and data parallelism, which tries to find some set of data for which to perform the same tasks in parallel. Of the three compared models, two will be function parallel, and one data parallel.

Synchronous function parallel model

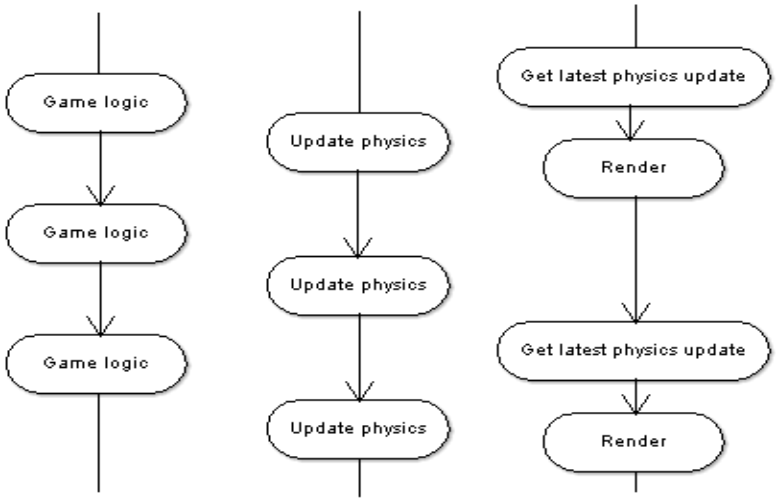
One way to include parallelism to a game loop is to find parallel tasks from an existing loop. To reduce the need for communication between parallel tasks, the tasks should preferably be truly independent of each other. An example of this could be doing a physics task while calculating an animation.



A game loop parallelized using the synchronous function parallel model. The animation and the physics tasks can be executed in parallel.

Asynchronous function parallel model

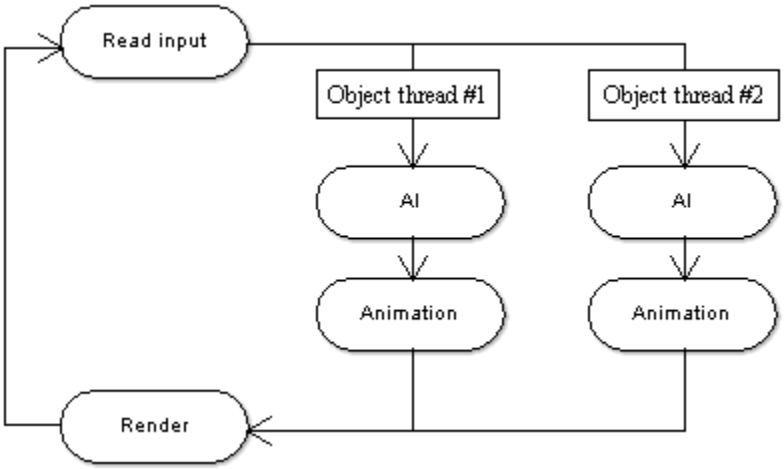
The difference is that this model doesn't contain a game loop. Instead, the tasks that drive the game forward update at their own pace, using the latest information available. For example the rendering task might not wait for the physics task to finish, but would just use the latest completed physics update. By using this method it is possible to efficiently parallelize tasks that are interdependent.



The asynchronous function parallel model enables interdependent tasks to run in parallel.

Data parallel model

In addition to finding parallel tasks, it is possible to find some set of similar data for which to perform the same tasks in parallel. With game engines, such parallel data would be the objects in the game. For example, in a flying simulation, one might decide to divide all of the planes into two threads. Each thread would handle the simulation of half of the planes



A game loop using the data parallel model. Each object thread simulates a part of the game objects

6. **The differences between a single-threaded game engine and multi-threaded game engine**

Single Thread	Multi thread
Refers to executing an entire project from beginning to end without interruption by a thread	Refers to allowing multiple threads within a process such that they execute dependently but share their resources
Must wait for the previous task to complete to respond	Multithreading enables programs to respond more quickly
A thread crash can cause the entire process to crash	If a thread crashes, the other threads can still run normally with no effect

- Most games were single threaded in the past. Developing multi-threaded games is difficult, so from that perspective it makes sense why so many games are still

primarily single-threaded. Almost all modern games use multiple threads now. Older games like CSGO or Dota 2 (2011/2014) which are esport games mainly known for CPU usage use up to 4 threads and ideally run better on at least 4core CPUs. Modern games such as Battlefield 5 (launched nov 2018) uses as many cores as possible.

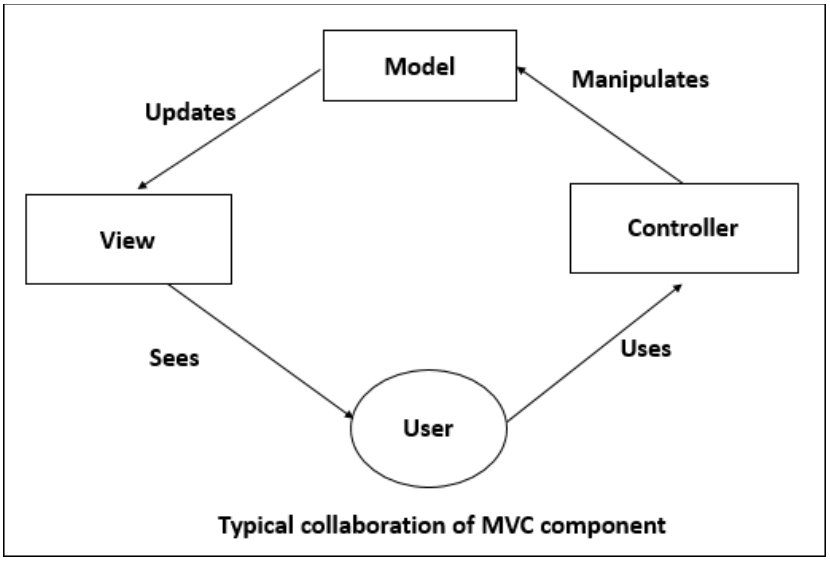
- Multithreaded doesn't automatically mean more responsive. Usually it means more throughput, but something that always reacts within a single frame is always more responsive than something which acts on some computed result that comes in 2 or 3 frames later. You probably meant that you can do more complex simulations by saturating the CPU more. This is true, but remember there are plenty of other systems in the game vying for core time.
- You can use Multithreading in games, but it does not mean that your code will run faster if your game is too simple to begin with. Do not fall into the trap of thinking that you can only have as many threads as the machine has. The operating system does a really good job at splitting up processing time.

II. Compare a multithreaded game engine design with 2 style MVC and PAC

1. **Model-View-Controller (MVC)**

One useful architecture pattern in game development is the MVC (model-view-controller) pattern.

It helps separate the input logic, the game logic and the UI (rendering). The usefulness is quickly noticeable in the early stages of any game development project because it allows to change things quickly without too much rework of code in all layers of the application.



Typical collaboration of MVC component

In MVC, as the pretty picture at the other end of that links shows, the Model holds data, the View is the part the player sees, and the Controller is an intermediary for game logic.

In MVC, the View component has direct access to the Model. The Controller itself doesn't enter the picture unless there is actual change to the data. Simply reading and displaying data is done entirely by the View component itself. As a result, a system can have many View components active at once, all reading and displaying data in a variety of different ways, even on different systems or in different languages or different modes (GUI vs. textual vs. web). On the flipside, however, that means the View component has to have some rather complex logic in it. It needs to know how to pull data out of the Model, which means it needs to know what the data structure is (or at least a rich API in front of the Model). It needs to be able to handle user interaction itself, with its own event loop.

Advantages:

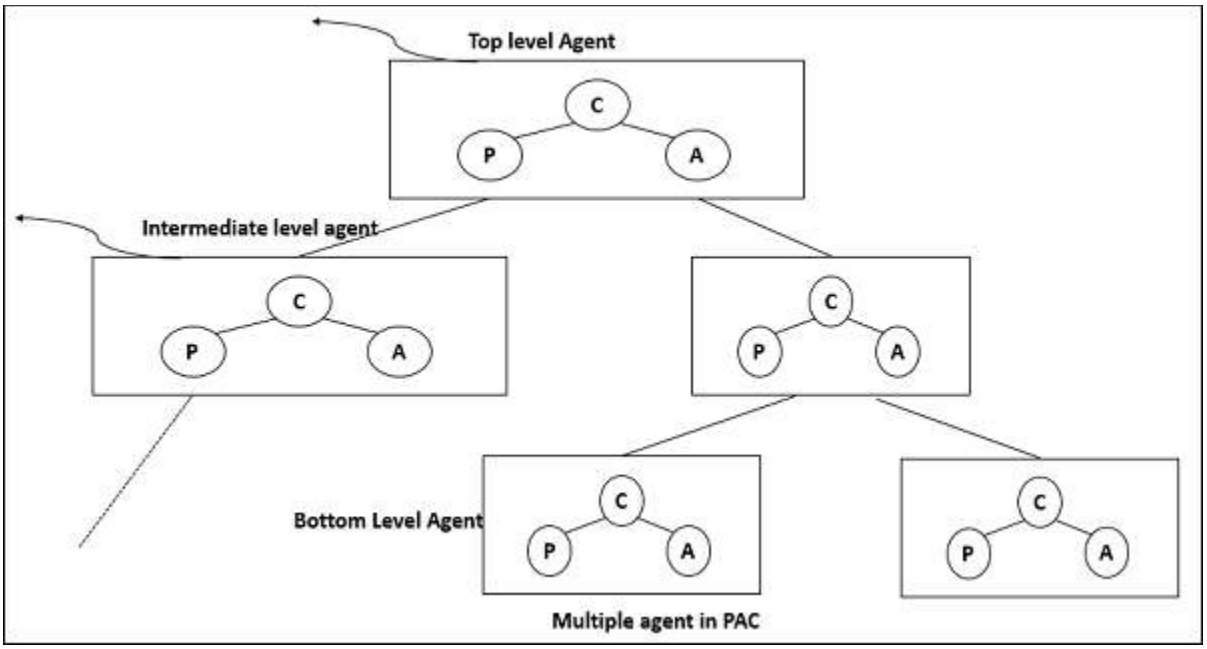
- There are many MVC vendor framework toolkits available.
- Multiple views synchronized with same data model.
- Easy to plug-in new or replace interface views.

Disadvantages:

- Multiple pairs of controllers and views based on the same data model make any data model change expensive.
- The division between the View and the Controller is not clear in some cases.

2. Presentation-Abstraction-Control (PAC)

Presentation–abstraction–control (PAC) is a software architectural pattern. It is an interaction-oriented software architecture, and is somewhat similar to model–view–controller (MVC) in that it separates an interactive system into three types of components responsible for specific aspects of the application's functionality. The abstraction component retrieves and processes the data, the presentation component formats the visual and audio presentation of data, and the control component handles things such as the flow of control and communication between the other two components.



In contrast to MVC, PAC is used as a hierarchical structure of agents, each consisting of a triad of presentation, abstraction and control parts. The agents (or triads) communicate with each other only through the control part of each triad. It also differs from MVC in that within each triad, it completely insulates the presentation (view in MVC) and the abstraction (model in MVC). This provides the option to separately multithread the model and view which can give the user experience of very short program start times, as the user interface (presentation) can be shown before the abstraction has fully initialized.

Advantages:

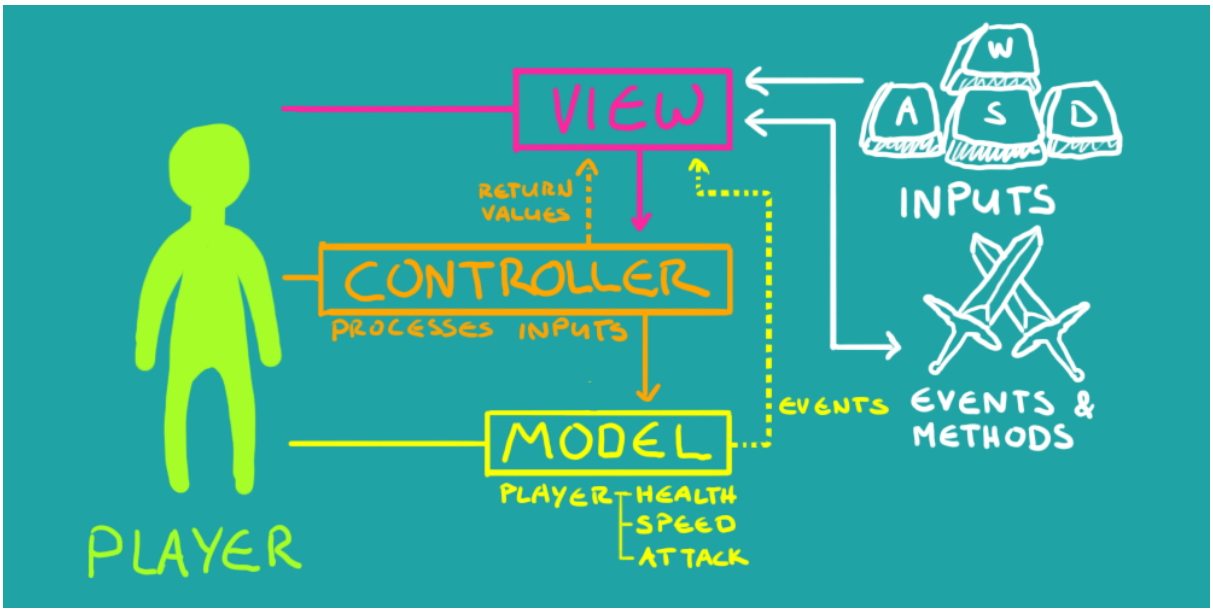
- Support for multi-tasking and multi-viewing
- Support for agent reusability and extensibility
- Easy to plug-in new agent or change an existing one
- Support for concurrency where multiple agents are running in parallel in different threads or different devices or computers

Disadvantages:

- Overhead due to the control bridge between presentation and abstraction and the communication of controls among agents.
- Difficult to determine the right number of agents because of loose coupling and high independence among agents.
- Complete separation of presentation and abstraction by control in each agent generate development complexity since communications between agents only take place between the controls of agents

3. MVC and PAC in Unity 3D Engine

MVC:



- **Model**: the player character, his/her inventory, spells, abilities, NPCs, and even things like the map and combat rules all being part of the model.
- **View**: is the UI dependent layer. It reflects the specific choice of UI you went with and will be very much dedicated to that technology. It might be responsible for reading the state of the model and drawing it in 3D, or images. It is also responsible for displaying any control mechanisms the player needs to use to interact with the game.
- **Controller**: is the glue between the two. It should never have any of the actual game logic in it, nor should it be responsible for driving the View layer. Instead it should translate actions taken in the View layer (clicking on buttons, clicking on areas of the screen, joystick actions, whatever) into actions taken on the model. For example, dropping an item, attacking an NPC, whatever. It is also responsible for gathering up data and doing any conversion or processing to make it easier for the View layer to display it.

PAC: Similar to MVC: the Presentation and Abstraction are same with View and Model but completely insulated.

- **Presentation**: creates all the basic elements of the interface; i.e. the game window, the grid layout of the interface and the size of each row or column, the creation of widgets for other elements of the interface. It also manages the impact of certain user events on the progress of the game.
- **Abstraction**: contains all the rules of the game as well as all the knowledge associated with it. In particular, it knows the location and characteristics of

all buildings and units on the playing field, information of agents, resources, etc. as well as the location of all obstacles on the field. It is also this aspect that the loading of a game takes place.

- **Control**: handles things such as the flow of control and communication between the other two components.

III. Shooting battle: age of skills (OOD)

1. Introduction

Shooting battle: age of skills is a multiplayer gunfight game with the special feature that each player has a character with unique skills with a bomb mode common in all other gunfight games. What makes this game not boring is the strategy and creativity based on the character's skill set. The game will bring players a new feeling, many game modes as well as many characters, skills, weapons. The game has beautiful simple graphics, vivid sound and smooth motion to help players have the best experience

Gameplay: 10 players in a match will divide to 2 team, one attacker and one defender with bomb mode. The defender must protect 2 site of the map, doesn't let the attacker entry site and plant the spike. They must buy weapon in store at first of each round. When 12 round played, the team will be change, and which team can reach 13 games won first will be win this match.

2. Shooting game with MVC

The MVC design pattern splits your software into three major categories: Models, Views and Controllers.

Models store values. In a game related example the model of the player would store their total and current health points, their speed, attack power and so on. Models do not perform any calculations or decisions, nor do they interact with other units or components. They just handle data.

The view is the interface of the MVC trio. It handles inputs and events and is in charge of graphical representation or objects in my concept. The view also is the only part of the MVC unit that communicates with other scripts outside the unit itself.

Finally **the Controller** is the brain of the unit. It handles information, makes calculations and decisions with it and sends it to the model or returns it to the view.

In this game, the player controls a robot the following can happen:

- 1 – User clicks/taps somewhere on the screen.
- 2 – The controller handles the click/tap and converts the event into an appropriate action. For example if the terrain is occupied by an enemy, an attack action is created, if it is empty terrain, then a move action is created

and finally if the place where the user tapped is occupied by an obstacle, do nothing.

- 3 – The controller updates the robot‘s (model‘s) state accordingly. If the move action was created, then it changes the position, if the attack, then it fires.
- 4 – The renderer (view) gets notified about the state changes and renders the world’s current state.

What this all means is that the models (characters) don’t know anything about how to draw themselves, or how to change their state (position, hit points). They are dumb entities.

The controller is in charge of changing the models’ state and notify the renderer.

The renderer has to have a reference to the models (characters and any other entities) and their state, in order to draw them.

We know from the typical game architecture that the main loop acts as a super controller, which updates the states and then renders the objects onto the screen many times a second. We can put all the update and rendering into the main loop along with the characters but that would be messy.

- **The models**

The character controlled by the player

An area where the character can move

Some obstacles, terrain

Some enemies to shoot at

- **The controllers**

The main loop and the input handler

Controller to process player input

Controller to perform actions on the player’s character (move, attack, ability, skills)

- **The views**

The world renderer – to render the objects onto the screen

3. Game components and features

3.1. Character (Agent)

Each agent in the game has its own characteristics, different looks, different voice sounds, and different skill sets.

Each character will have 4 skills including: 3 basic skills that can be purchased in the store in a certain amount, and an ultimate skill obtained after each kill the enemy.

An agent with his skills

3.2. Weapon and Armor

There are 7 kinds of weapon: Sidearm, Smg, Rifle, Shotgun, Sniper, Machine gun, Melee and 2 kinds of armor: Light shield and Heavy shield. They sell in store in the match.

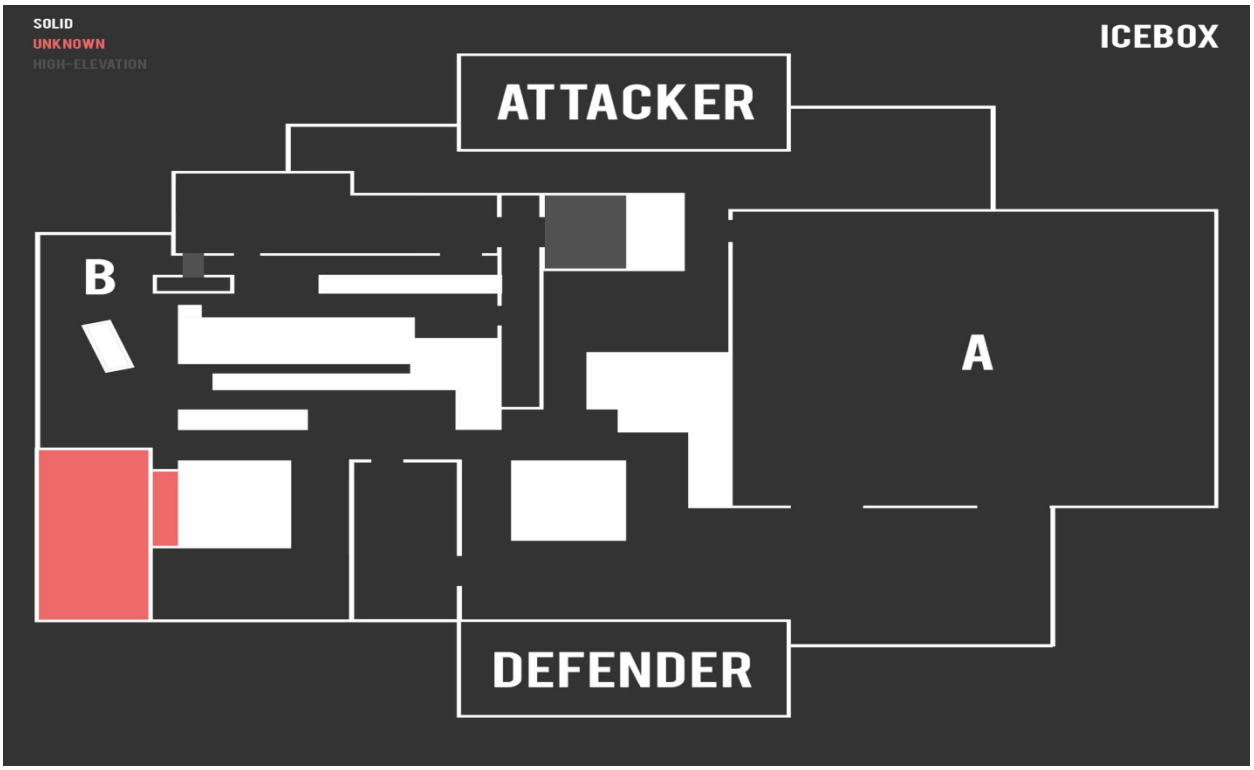
Each weapon has different damage, bullet speed, bullet path, number of bullets. They have different prices and can buy in store at first of round.

Weapon store

3.3. Map

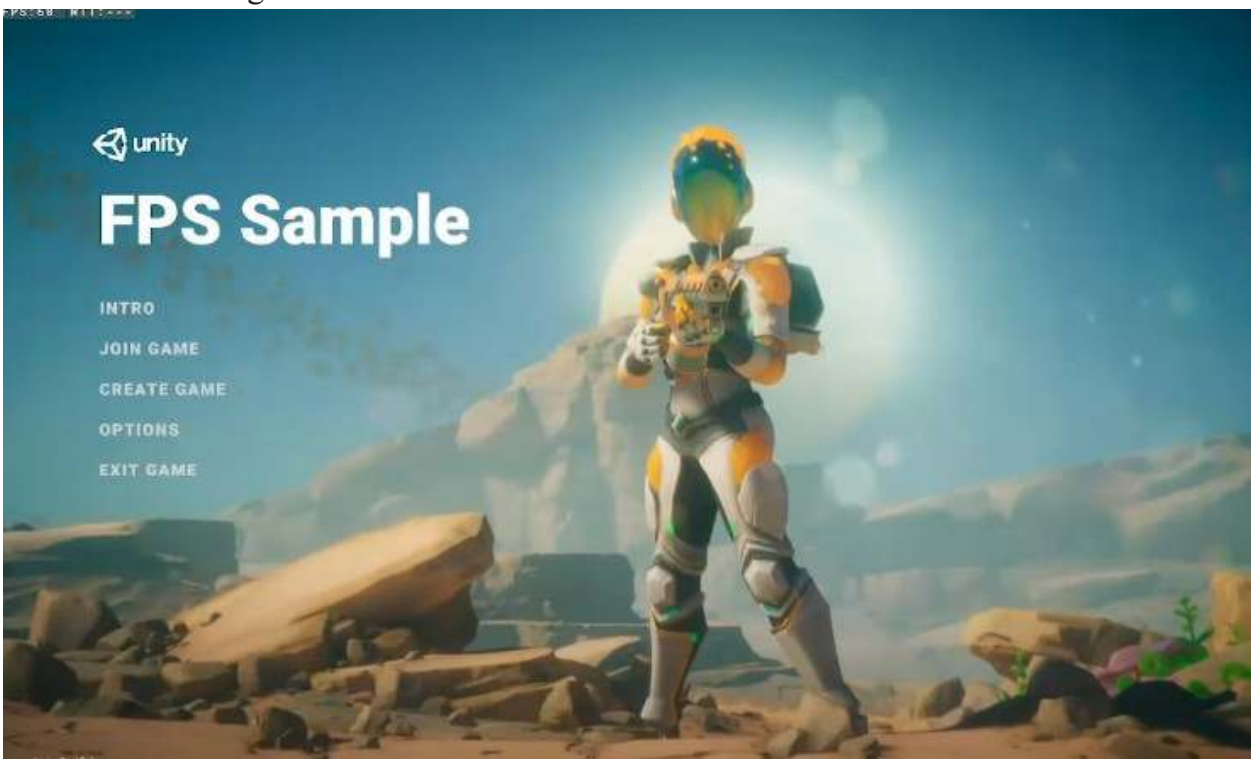
Each map has 2 site to attack, the attacker team must entry and plant spike in site, the defender team will be protect site and does not let the attacker get in.

A map in game



A map layout

3.4. Some game screens



Game menu screen

Select agent screen

Spawn base screen

In a round screen

4. CRC Card Approach

1.1. Agent

Agent	
Responsibility	**Collaborators**
- Move, jump, crouch	- Terrain, AgentController
- View	
- Detect enemies	- AgentController
- Plants, defuses spike	- AgentController
- Shoots	- WeaponController
- Use skills	- AgentController
- Buy weapons, skills	- AgentController
- Knows HP, Armor, Position	

1.2. Weapon

Weapon	
Responsibility	**Collaborators**
- Knows weapon type, prices	
- Knows weapon bullet damage	
- Shoots	- Terrain, WeaponController
- Knows bullet path	- WeaponController
- Drops Weapon	- WeaponController

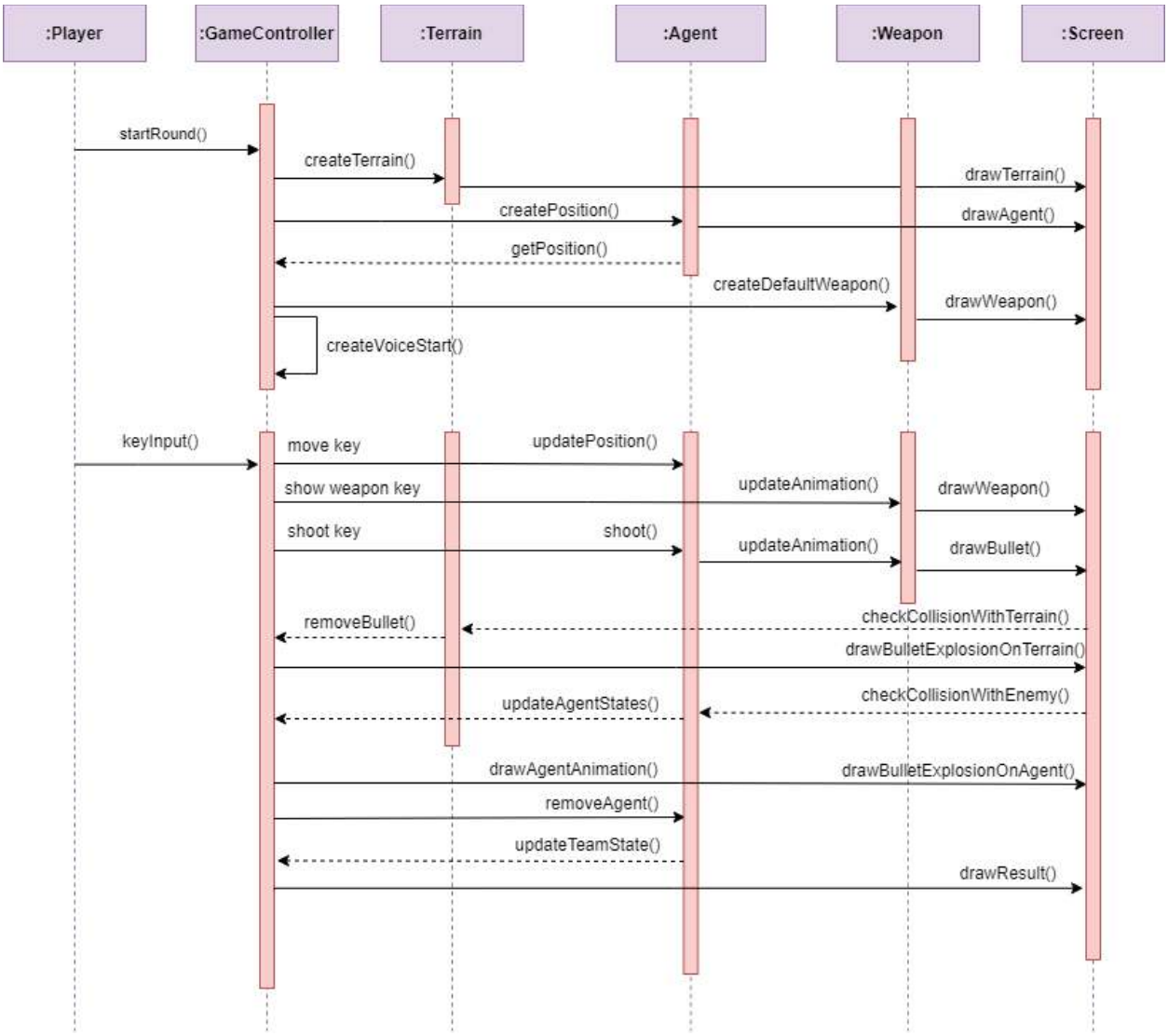
1.3. Agent Controller

AgentController	
Responsibility	**Collaborators**
- Handle agents move step	
- Handle skills cooldown	
- Handle agent properties	
- Handle money buy items	
- Handle agent states	

1.4. Weapon Controller

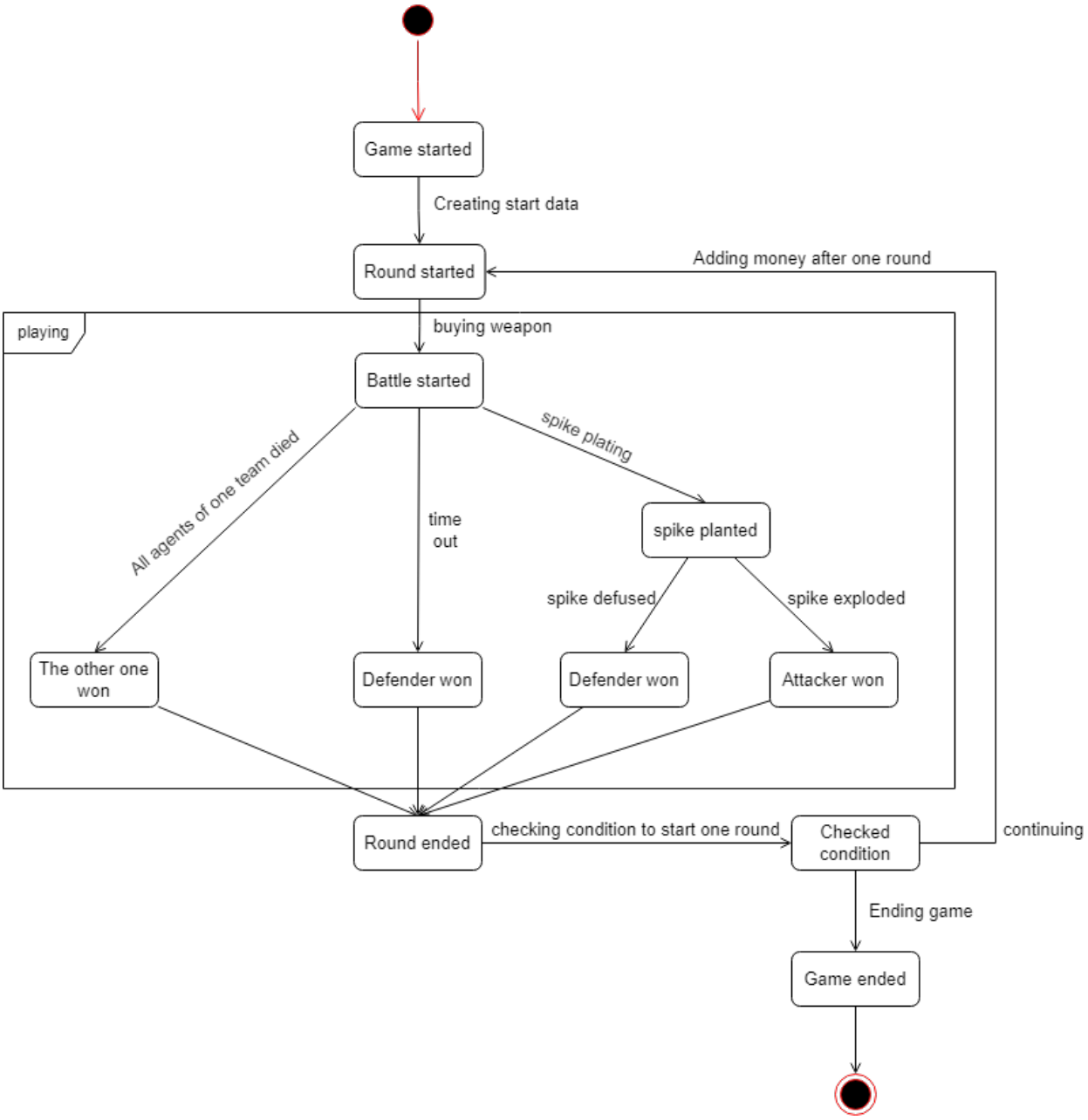
WeaponController	
Responsibility	**Collaborators**
- Handle bullet path	AgentController
- Handle buy weapon	
- Handle weapon damage	
- Handle weapon skin	

5. Construct Interaction Diagram



Animation sequence diagram

6. State Chart



State Chart of a game match

7. Class Diagram