AE2CPP

INTRODUCTION REPORT FOR THE GAME: ANGRY BIRDS & PLANTS VS ZOMBIES

30th April 2016

Name: Yichen PAN
Student ID: 6514897

University of Nottingham Ningbo China
School of Computer Science

Contents

1 INTRODUCTION

1.1 Game Story

The title of the game is Angry Birds & Plants VS. Zombies. The game background is that there has been a war between zombies and plants lasting for hundreds of years. As time goes by, zombies gradually became immune from normal attacking from plants and they killed all sunflowers. As a result, the number of plants became smaller and smaller, and finally there was only one pea shooter left. In order to protect the garden, this final plant, pea shooter, asked help from angry birds. To fight against the vicious zombies, angry birds agreed to help the pea shooter, and thus the war between plants, angry birds and zombies starts.

1.2 Game Play

The players controls the pea shooter, which is set at the middle of the bottom of the screen. It could send out angry birds, like bullets, to kill the zombies that come from the right side of the screen and walk to the left side.

2 GAME MECHANICS

2.1 Game features and basic logics

In order to win the game, the player needs to get more than specified points in the limited time. The only way for the player to get points is to kill zombies and prevent zombies arriving at the left side of the screen because points will be deducted as punishment if the zombies come across the left side.

Moreover, different kinds of angry bird bullets have different cost, which prevents the player from shooting and wasting bullets unlimitedly. In this case, the player has to keep a balance between the cost and power of bullets. Unlike normal games, there are no common games attributes such as money, health and energy;instead, there is only one attribute, which playes the roles of all of them. This simplicity makes the game easy to play but challenging to win.

2.2 Difficulty level

In total, there are three difficulty levels, including easy, medium and difficult level. The difference between levels is the type of zombies (differ in the capability, points and velocity), the time and points required to win the game. The summary of the difference is shown as following:

Column1	Easy	Medium	Difficult
Points required	20000	35000	100000
Time	90s	75s	60s
Number of zombies on the stage	10	15	20
Game background scene	Ancient Egypt	Pirate Sea	Wild West

Figure 1: Comparison among different game levels

2.3 Zombie

Different types of zombies have different HP, velocity and rewarded points after being killed. There are in total eleven kinds of zombies and the comparison between them is shown as following:

Name	Picture	Level	HP	Velocity	Points
Fly zombie		easy	100	2	100
Balloon zombie		easy	200	5	200
Small zombie		easy	300	2	300
Evil zombie		easy	400	2	400
Newspaper zombie		medium	500	4	500
Stairs zombie		medium	600	5	600

Figure 2: List of zombies

Name	Picture	Level	HP	Velocity	Points
Polejump zombie		difficult	700	6	700
Diving zombie		difficult	800	4	800
Car zombie		difficult	900	6	900
Running zombie		difficult	1000	8	1000
Boss zombie		difficult	3000	3	5000

Figure 3: List of zombies

2.4 Angry bird bullet

Different types of angry bird bullets have different demage, cost, velocity and attribute. There are in total three kinds of bullets and the comparison between them is shown as following.

Name	Picture	Damage	Cost	Velocity	Attribute
Red bird bullet		40	0	30	none
Yellow bird bullet		20	4	40	Five bursts at a time
Black bird bullet		200	30	30	Make zombies move slowly

Figure 4: List of angry bird bullets

2.5 Game interfaces

In total, there are five different kinds of interfaces, including the initial welcome interface at the beginning, the level selection interface, the main interface for playing games, pause interface and the lose or win interface.

2.5.1 Initial welcome interface

This interface is shown right after the player runs the game program. The interface is specially designed together with the logo (Angry Birds & Plants VS Zombies). By pressing any key, the player could get into the next interface as often the case.

Figure 5: The initial welcome interface

2.5.2 Level selection interface

This interface is shown right after the player press the key in the initial welcome interface. There are three levels provided for the player to choose. At the top-right corner, there is a panel showing the history highest mark, which may encourage the player to play hard.

Figure 6: The level selection interface

In each different level, the game background is different (shown below). The type and power of zombies also differ from different levels.

Figure 7: Screenshoot of the easy level game

Figure 8: Screenshoot of the medium level game

Figure 9: Screenshoot of the difficult level game

2.5.3 Game interface

In the game, the time left is shown in the middle-up of the screen and points earned so far are shown in the right-up corner (shown in Figure 7, 8 and 9).

2.5.4 Pause interface

During the game, the player could pause the game by pressing P of the keyboard. One menu with three buttons will appear as shown in Figure 10 below. The player could continue the game by clicking Continue button, return back to the main menu (the initial welcome interface) through the middle button and exit the game directly using Exit button.

Figure 10: The pause interface

2.5.5 Win/Lose interface

After the time is up, if the player gets more points than those reuqired by the game level, the player wins the game; otherwise, the player loses the game. The two similar interfaces are shown below. The player could go to the level selection interface by clicking the OK button.

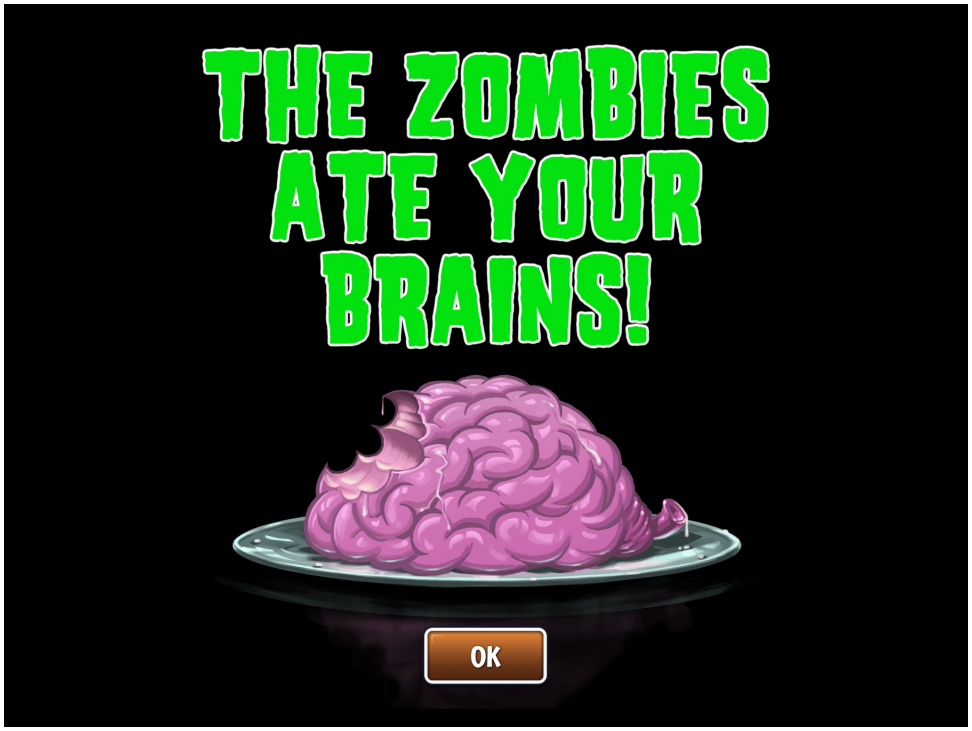

Figure 11: The game lose interface

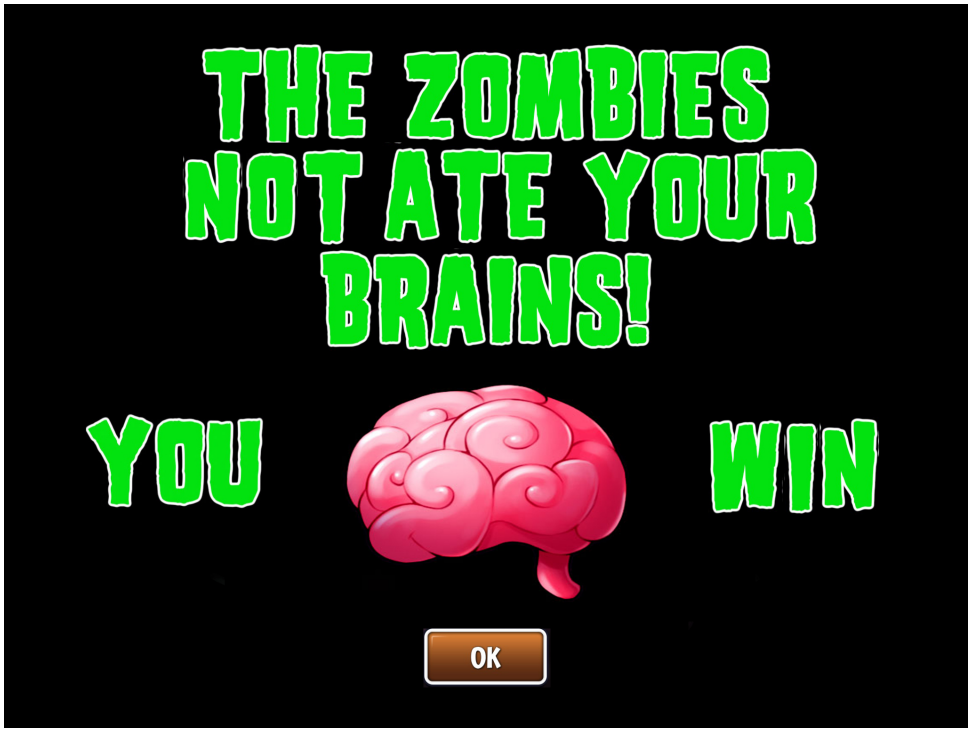

Figure 12: The game win interface

Particularly, the brains are different in two interfaces: one is complete and fresh which indicates the winning of the player and the other one is fragmentary and corrupted.

2.6 User manual

Two different sets of control are particularly designed for players using the mouse and those using the touch pad of the laptop. The player could shoot by clicking the left button of the mouse or pressing space of the keyboard. By pressing A and D or Left and Right, the player could switch between different bird bullets.

During the game, the player could pause the game by pressing P of the keyboard and exit the game directly through pressing Esc.

3 GAME DESIGN AND IMPLEMENTATION

The game is written in C++ and Simple DirectMedia Layer (SDL) 2.0, which is a cross-platform development library designed to provide low level access to audio, keyboard, mouse, joystick, and graphics hardware (Wikipedia, 2016). The complete codes are divided into several logically related components and run through the main function defined in MainFunction class.

3.1 Game engine: MyEngine and Game class

First of all, MyEngine class is designed as the generally applicable base engine for the SDL based 2-D game, which provides fundamental functions such as mouse and keyboard events handlers, functions for updating objects and rendering the screen, game logic framework and deallocation functions for freeing memory. It is a base class which only serves as a template and consists of a number of virtual functions that will be overridden by its derived classes for specific implementation. It also consists of necessary objects for the game such as SDL_Window, SDL_Renderer, timer, the current position of the mouse, the list of all displayable objects, the current game state and game level. There are three main functions responsible for handling the game logic. The first one is called mainLoop, which loops until reaching the specific exiting state. In each loop, it stores key press information and calls call-back functions for different events. It also calls two other main functions. One is called gameAction, which updates all related objects by calling update functions of each displayable objects and the other is gameRender which renders the updated objects to the screen.

Game class is the derived class of MyEngine class specially designed for this game. It gives concrete implementation of virtual functions defined in the base class and consists of all game specific objects, including the list of zombies and angry bird bullets, the textures of background, buttons and texts, audios, the timer and score. It overrides event handlers, gameAction function and gameRender function to handle game-specific logics and realize various updates of objects.

The UML sequence diagram below shows the underlying logic behind the game and the order of function calls:

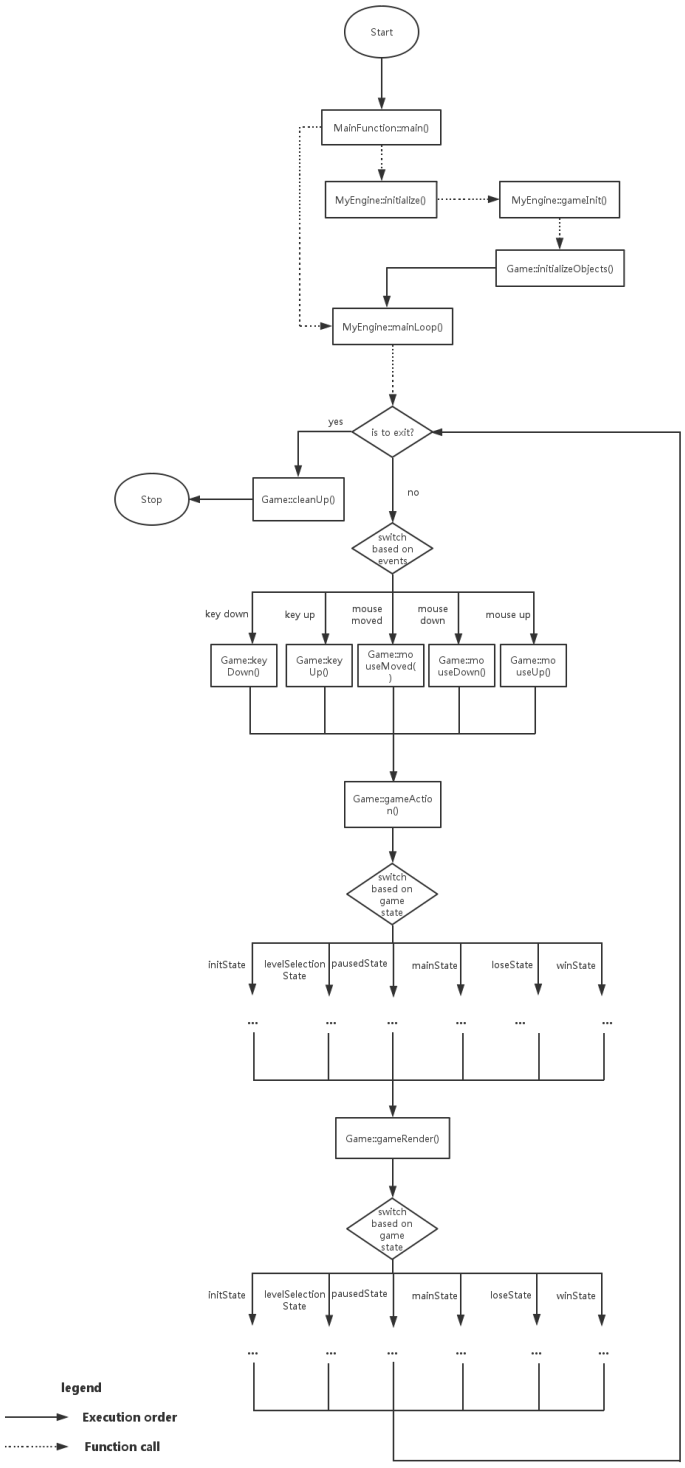

Figure 13: UML sequence diagram

3.2 MyTexture class

MyTexture is a wrapper class which encapsulates all necessary information and functions such as loading image from file, rendering the loaded image at specified position and deallocating the texture, of the native SDL_Texture class.

3.3 MyTimer class

MyTimer is a function-specific class responsible for all the basic functions such as starting, stopping, pausing and unpausing the timer.

3.4 Utility classes: struct Vector2D and Utilities class

Structure Vector2D and Utilities class are two utility classes, specially used for calculating the angle between two pints in the rectangular coordinate system. They are modified from codes provided in the AE2CPP lecture.

3.5 Game state

In total, there are six game states defined in State enumeration, including initState, levelSelectionState, mainState, pausedState, loseState and winState. Through these six states combing with switch statement, it is easy and flexible to control the state of the game.

4 OOP DESIGN

Object oriented concepts, such as abstraction, encapsulation, separating implementation from interface, inheritance, composition and polymorphism, are carefully taken in consideration and adopted flexibly during the development of the game.

In terms of abstraction, the main purpose is to hide data or information that is not necessary to be presented. The important idea behind data abstraction is to give a clear separation between properties of data type and the associated implementation details. For encapsulation, it is mainly focused on restricting access to some components of the object, which prevents the object being changed directly from outside sources. Abstraction together with encapsulation mechanisms provides more flexibility in approach and ensures the security of the program. As a form of software reuse, inheritance allows an object (of the derived class) to inherit properties and methods from another object of a higher or base class. The relationship between the two classes is the is-a relationship. In C++, there are two types of inheritance, single and multiple inheritance, and three specific ways to implement inheritance: public, protected and private. In this game, inheritance is efficiently adopted in the design of zombies, bird bullets among all other displayed objects in the game.

In OOP design, composition refers to a way to combine simple objects or data types into more complex ones. Each composition is an independent logic building block of related objects of other class. The relationship represented by composition is the has-a relationship, which is different from the is-a relationship in inheritance. Polymorphism is another important part of OOP, which enables having different objects with the same interface, but with different implementations. In the OOP design of this game, polymorphism is realized through hierarchical inheritances and flexible manipulation of pointers.

4.1 Separating interface from implementation

As a fundamental principle of software engineering, separation between interface and implementation could hide details of implementation from the client code. In this game, all class definition is splitted into two separate files, the header file where the class is defined and the source-code file in which the member functions of the class are defined.

4.2 Inheritance hirearchy

DisplayableObject is the base class which represents an independent object shown in the game interface. It composes all necessary information about an game object including the pointer to the texture (of MyTexture class), the pointer to the game engine (of MyEngine class), the width, height and the position of the object. Zombie, BirdBullet, Battery, Button and Text are all derived classes of DisplayableObject class because they all share common properties and methods.

Each derived class overrides the virtual function such as draw and doUpdate to realize various visual appearance and animation. The UML class diagram below shows the relationship of the hierarchical inheritance:
Based on the DisplayableObject inheritance, polymorphism could be easily achieved and thus the whole game system becomes much more flexible and easily extensive. By assigning the address of a derived class to a base-class pointer, specific functions of the derived classes could be called through base-class pointers. Based on this mechanism, a list of DisplayableObject pointers pointing to different derived classes is maintained. When updating or rendering each object, there is no need to distinguish among object types and invoke an appropriate action for a particular object at compile time; instead, a one time traversal through the whole list and calling one same function of each object could achieve the expected effect.

Moreover, new classes such as a new zombie or bird bullet can also be added with little or no modification to the program, as long as they are part of the inheritance hierarchy that the program processes generally.

4.2.1 Inheritance Example: Zombie and its derived classes

Zombie class itself serves as a base class and provides template for its derived classes, various kinds of zombies. In the game, derived classes only override the constructor to specify different texture, HP, velocity and rewarded points due to the simplicity of the game. However, there may be more methods needed to be overridden, such as die function which creates various dying animation after the zombie is killed.

Figure 14: Composition and aggregation UML diagram

4.3 Composition and aggregation

Basically, there are two kinds of compositions. The first one is the normal composition where all sub objects which belong to the owner object are destroyed whenever the owner object is destroyed whereas the other one is called aggregation where sub objects will not be destroyed. In the normal composition, subclasses are added as normal variables or pointers, existing within the class. However, in aggregation, sub objects are either pointers or references.

In the game, one example of aggregation is that each object of DisplayableObject maintains one pointer to the game engine so that each object could get or change attributes of the game such as the current position of the mouse, the game state and game level by calling getters and setters. As aggregation, when the objects of DisplayableObject class is destroyed, for example, the zombie is dead and the bullet is exploded, the game engine will not be affected and still exist.

With regard to the normal composition, for example, the Game class consists of necessary objects for the game such as SDL_Window, SDL_Renderer, timer, background images (of MyTexture class) for different interfaces, audios and the list of all displayable objects. The UML diagram below illustrates the main composition and aggregation relationship.

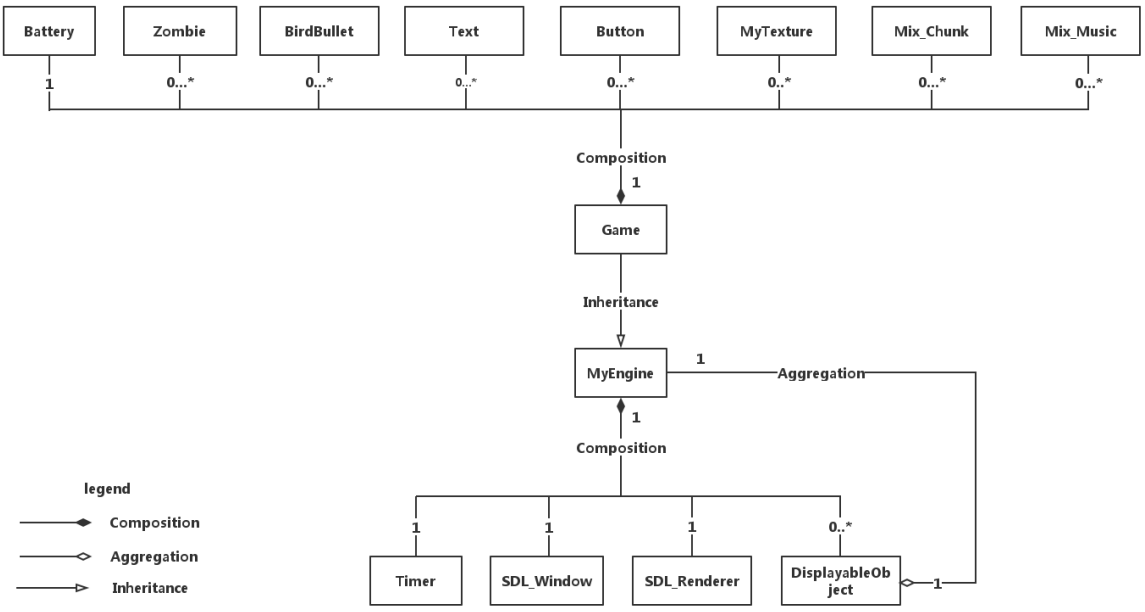

Figure 15: Composition and aggregation UML diagram

4.4 Design pattern: Observer

The relationship between the game engine and all displayable objects is by nature a one-to-many dependency, which means once the state of the engine (game itself) is changed, all its dependents should be notified and updated automatically. The observer design pattern is an ideal solution to this situation.

In the design of the Game class (game engine), it maintains a list of objects of DisplayaleObject, which are its dependents, called observers, and notifies them automatically when the state is changed through function updateAllObjects.

5 CONCLUSION

After developing a complete SDL-based game, a deeper understanding in C++ programming language and object-oriented programming is developed. Below is one reflection on OOP after implementing inheritance in pratical work.

5.1 Composition over inheritance

Composition over inheritance in OOP indicates that polymorphic behavior and code reuse should be achieved through composition instead of inheritance. This is a well-known principle of OOP, which is also stated in the Design Patterns: Favor 'object composition' over 'class inheritance' (Gamma et al, 1994).

The reason why composition is preferred comparing with interitance is that it is easier to modify the related code. With composition, it is easy to change the behavior on the fly with Dependency Injection. By comparison, inheritance is much more rigid.

REFERENCES

Wikipedia. (2016). Simple DirectMedia Layer. [online]
Available at: https:en.wikipedia.orgwikiSimple_DirectMedia_Layer [Accessed 30 Apr. 2016].

Gamma et al. 1994, p. 20.