INM379 Computer Games Architectures

Paraschos Moschovitis – Student ID 110002549

Contents

Introduction 3
Structure Overview 3
Class Analysis 4
Background.cs 4
Collidable.cs 4
Collision.cs 5
CollisionManager.cs 5
Enemies.cs 5
Explosion.cs 6
FileReader.cs 6
HighScore.cs 7
HUD.cs 7
ButtonState.cs 7
KeyboardEventArgs.cs 7
InputListener.cs 7
CommandManager.cs 7
Player.cs 8
Projectile.cs 9
SoundManager.cs 9
ScoreManager.cs 9
Game1.cs 9
Discussion 11
A discussion of the use of profiling software to improve the performance of your game engine 11
References 12

Introduction

The aim of this report is to provide a clear overview of the architectural principles, models and methods followed to create a 2D game on Microsoft XNA using Visual Studio 2015. The project demonstrates the use of data-driven and event-driven architectures to separate the game engine from the game code and focuses on the decoupling of separate code modules for greater usability and functionality without disrupting the core engine.

It illustrates how each element of the game was structured, the reason behind it and how it contributes to the smoother operation of the game. Therefore, it is broken down to each individual class to be analysed below

Structure Overview

The game is a 2D shooter on a landscape mode where the player controls a sprite, in our case a guitar pick, fighting enemy notes to earn points and achieve a high score. Therefore, a few classes had to be created to present that type of gameplay action while also implementing event-driven and data-driven architectures. The class descriptions and roles can be seen below and are analysed further in the next chapter:

Game1.cs: That's where the main game logic is built. This class uses instances from other classes to set up, configure and most importantly run the game. It connects all the game elements into a playable game.

Background.cs: A simple class created to regulate a moving background image

ButtonState.cs: Contains an enumerator with the keyboard button states

Collidable.cs: An important class from which the Player and Enemies classes inherit properties. Used to separate collision detection and reaction

Collision.cs: Setting up the collision action, and ensuring that two same objects will not collide

CollisionManager.cs: Is the regulator class with the methods necessary to set up a list of collidable objects with operators such as addition and removal, and check for their collision

CommandManager.cs: Containts the delegate and event assignments necessary for the event-driven design of the user input part (keyboard)

Enemies.cs: Contains all the information and methods required for enemies created within the game

Explosion.cs: Spawns time-independent animated explosions when an enemy dies

FileReader.cs: Contains methods to read data from txt files, such as enemy attributes, to implement a data-driven architecture

HighScore.cs: A simple class containing the format of a file that collects all the high scores

HUD.cs: Another simple class used to display an update a score while playing the game

InputListener.cs: Used in conjunction with the KeyboardEventArgs class to capture the user inputs and notify the CommandManager

KeyboardEventArgs.cs: A custom event argument class inheriting from EventArgs, to capture current and previous keyboard states and key pressed

Player.cs: Important class defining all the player attributes, controls and methods necessary for the game

Program.cs: Creating an instance of the game to play

Projectile.cs: Contains the specifications of a projectile spawned by player

SoundManager.cs: Loads the music and effects to be used in the game

ScoreManager.cs: Supporting an event driven design to increase the game score

Class Analysis

The following is an analysis of each of the classes used in the project describing the code and its functionality as well as how they are connected together. For the complete code please see the Games Architecture project itself.

Background.cs

The role of this class is to implement a rolling background image. It is made up of two 2D vectors that store the position of the image, img1 and img2. Img2 is set at a position on the x axis which is just outside the game window and as both images are assigned a speed moving towards the left, Img2 will eventually appear in the screen and Img1 will disappear to the left. When either of the images reach the position (-1118, 0) which is the width of the game window, they are moved to (+1118,0) to achieve the rolling background

Collidable.cs

A class where all collidable objects in the game - in our case, the player object and the enemy objects – inherit from. Collidables are all objects that have a boundary which when it collides with a boundary of another object, it can react. Therefore, this class provides a Rectangle variable which is used as a collider box.

The important part is the two functions it also specifies, which are CollisionTest and OnCollision together with the property FlaggedForRemoval. CollisionTest will constantly be checking in an update whether the current object is colliding with another object whereas OnCollision will specify what action to be taken when such objects have intersected (collided). Therefore, it provides a good way to separate *collision detection* and *collision response* and make the code more robust. Both are methods shared by all the classes that inherit from Collidable.cs and since they are virtual, they can be overridden to give different reactions to collisions accordingly.

FlaggedForRemoval marks the object if a collision occurs and its effect is to remove it from the game. It is a mark used in the Game1.ResolveRemovals() to delete the objects from the game.

Collision.cs

This class is made out of two components, the Collision class itself and a CollisionComparer class. Collision takes two objects, makes sure they are not null and that their collision hasn't already been registered. CollisionComparer takes two Collision objects and ensures they are not the same and then gets the first ones' hashcode to use it at the CollisionManager class.

Finally, Resolve method invokes the OnCollision method mention above for the two objects, drawing the overridden function from the first one.

CollisionManager.cs

This class provides a list of all the collidable objects and the collisions as well as methods to modify and update them:

AddCollidable method adds a collidable object to the list

RemoveCollidable method removes the object from the list

UpdateCollisions method firstly clears the list since we want it to retrack collisions every frame and then iterates through the collidable list to find collisions. If it does, it adds it to the hashset with the collisions.

ResolveCollisions method goes through all collisions gathered and calls the Resolve method, as mentioned in the Collidable class, on each of the collisions

The *UpdateCollisions* and *ResolveCollisions* are then grouped in an Update function to update together.

Enemies.cs

The class contains the information necessary to spawn enemy objects in the game and uses a data-driven architecture to do so. Using the function *FileReader.ReadLinesFromTextFile()* (discussed below) and a given path the class reads the lines from the file that hold the

enemy attributes and stores it on a list. Then it initialises each enemy attribute by accessing a position in that list. The assumption is made that the enemy attribute text file will always store the required information in the same order.

Adding to the challenges of the game, a difficultyLevel parameter is also used to calculate enemy health and damage and adjust it accordingly as the difficulty of the game progresses. Its value is also read from a text file.

The Enemies class inherits the properties from the Collidable class therefore, it has access to a rectangle which is used as a collider. Within the Enemies class, the methods *CollisionTest* and *OnCollision* are overridden to check for collision appropriately and then react according to the game rules. In a summary, if collision is detected with a player then mark the enemy with the *FlaggedForRemoval = true* so that it can be removed on the next Update.

As a side note, enemies have a constant speed regulated by the attributes read from the file.

Explosion.cs

As mentioned in the overview, this class is used to demonstrate frame-independent animation in the game. Whenever an enemy sprite is removed from the game, an explosion will appear.

To set it up, firstly a timer is added to the class:

a `timer += (float)gameTime.ElapsedGameTime.TotalMilliseconds;`

along with an ”interval” value which specifies how fast or slow the animation will progress. Next, a rectangle variable called sourceRec is initialized which will be positioned around the first frame of the animation sprite sheet. It is only what is currently inside the sourceRec that will be drawn on the screen.

Each time the timer surpasses the interval set, this rectangle will move to the next frame of the sprite sheet until it reaches the end. For this project, a sprite sheet with 18 frames was used therefore, the rectangle will move 18 positions before resetting.

FileReader.cs

This class is used to specify methods to read from files. It holds a Stream variable and updates it based on input when called. It is only function is the ReadLineFromTextFile() mentioned above which uses a StreamReader variable to get each line of the file read by using ReadLIne(). Each of those lines are added to a list of lines as the function goes through the file and the function itself returns that list at the end. Also, there is an exception thrown in case the file could not be read.

HighScore.cs

The HighScore class acts like a structure and stores a name and a score up to a maximum of four scores to be used by the ScoreManager.

HUD.cs

Here is where the score displayed while the game is played is set up. This class specifies its position on the window and its font. The score is always displayed; however, a further improvement could be to be able to toggle it for display only when needed by the player.

ButtonState.cs

It contains an enumerator for button states which will identify the keyboard key state later, when used in the event driven design of user input.

KeyboardEventArgs.cs

A custom class inheriting from the EventArgs class. It is used to store a key, current keyboard state and previous keyboard state. These parameters will be used to define a button that is up, down momentarily or pressed down to take the best action in game and improve the controls.

InputListener.cs

As described previously the role of this class is to “hear” and register what is coming from the keyboard/user. It holds a hashset o the keys pressed and a function *AddKey* to expand it and has three events, *OnKeyDown, OnKeyPressed* and *OnKeyUp* to capture all possible states of the keyboard keys. A *FireKeyboardEvents* function specifies which keyboard event to fire by checking the state of the keyboard before and currently and chooses the most appropriate one.

In its update function it constantly updates the keyboard states and triggers the *FireKeyboardEvents* function.

CommandManager.cs

The main class used to manage the keyboard inputs with an event-driven design. It provides a delegate, GameAction to specify the signature of the functions called when an event is triggered. In this case we are looking for a button state from the enumerator eButtonState and an amount which for simplicity is fixed to 1. The class also holds a Dictionary of keys and actions.

Using an instance of the InputListener class mentioned above, it registers the events with the input listener so the two can communicate.

The input listener waits for a keyboard key to be pressed then sends a notification to the CommandManager which has subscribed its current instance to that event, takes action.

The buttons to be added through the CommandManager are specified in the Player class mentioned below. The way to do that is through the function *AddKeyboardBinding*.

Player.cs

The design of the player class is part event-driven and part data-driven. As specified above when talking about the CommandManager and InputListener, there is a set of events that can be triggered according to which button is pressed and its state. In the Player class, we can define which functions to call for each of the buttons added to the CommandManager in the Game1 class. For example:

Moving the player upwards

```
  public void Up(eButtonState buttonState, Vector2 amount)
        {
            if(buttonState == eButtonState.DOWN)
            {
                velocity.Y -= 4.0f;

            }

        }
```

Player shooting projectiles

```
public void PlayerShoots(eButtonState buttonState, Vector2 amount)
        {
            if(buttonState == eButtonState.DOWN)
            {
                PlayerShoots();
            }
            if(buttonState ==eButtonState.UP)
            {
                projDelay = 1;
            }
        }
```

Also, the player movement is frame independent, as it uses a time variable to update the sprite's position:

```
float time = (float)gameTime.ElapsedGameTime.TotalMilliseconds;
            position += velocity / time;
```

The player in this game can shoot projectiles by calling the *PlayerShoots* function. In that function there is three if statements to regulate the projectiles spawning time using the projDelay variable. When the projDelay is back to zero, it means that we can shoot a projectile again. Each projectile shot has a collider which is a Rectangle with the same position as the projectile sprite and can collide with enemies.

Then, in the *UpdateProjectile* function the position of that particular projectile is checked and if it is out of bounds then that projectile is deleted. First it is marked as *p.exists=false* and then on a separate projectile list iteration all the projectiles that have been marked are removed. That is the same logic as the Collidable class.

Finally, since the Player is inheriting from the Collidable class, the two methods provided by that class are overridden to abide to the game rules. Namely, when a player collides with an enemy, that enemy is destroyed, and the player's health is reduced by that enemy's damage. Player's health is represented by 30x30 pixel squares which is added next to each other to appear as a bar.

Projectile.cs

This class gives texture, speed and a collider to the projectiles shot by the player. Even though the Projectile class could also be inheriting from Collidables, it was chosen not to as it was simpler to check for projectiles colliding with enemies on the Game1 class.

SoundManager.cs

We use the SoundManager to load two sounds in our game, a background music that will be on during the Playing state and a sound effect triggered when an enemy dies. The load method for the sound effect can take a wav file however, for the background music which would be of type "Song" cannot take a wav input but instead can take an mp3.

ScoreManager.cs

The score manager acts similarly to the CommandManager however, regulating how the event is increased when a certain event is triggered. In our case the score would increase when an enemy dies. However, this was not implemented fully due to time constraints and therefore, it can be an improvement point.

Game1.cs

This is where all the classes are connected to and therefore is a quite straightforward class. To start with, this is where the key bindings are specified and initialized through the *InitializeBindings* function. It uses an instance of the commandManager and the *AddKeyboardBinding* function to bind a key to an action.

Then, there is a small function *InitializeCollidableObjects*, where the player object is added to the collidables list however, since the enemy is appearing in a list, it is added to the collidables further down.

The game has three states as specified within the Update function and these are Menu, Playing and GameOver.

Menu:

This is the main menu screen and can contain instructions for controls and other functions such as muting the sound. For the moment, when the game starts the Menu screen appears and by pressing Enter the player can start the game while pressing Escape exits the game

Playing:

The game has started, and the player controls the guitar pick to shoot down enemies and collect points. During the Playing state all classes update functions are called. At the same time, here is where we check for collision between projectile and enemies. If they do collide, the projectile is marked for removal and the enemy's health drops. When it reaches zero, the enemy is also marked for removal, the "enemyDies" sound plays, the hud score is increased and an explosion object is added to the explosion list. The ManageExplosions function is responsible to update the explosion list and remove the explosion animations that have been completed.

The enemies are loaded through the *LoadEnemies()* function using a data-driven design. A random number would be generated when an enemy is about to spawn. That number is used to name the file to look for and therefore, give different attributes to different enemies as appropriate.

GameOver:

When the player's health reaches zero, the game transits to the GameOver state. Here, the player's score is saved and on hitting Enter, the game attributes reset, and the player is moved back to the main menu screen.

To save, load and display a high score table, four functions were necessary:

SaveHighScores -> Opens up the file name, saves it there using XmlSerializer and the instance of the HighScore class which is `[Serializable]` as specified in class.

LoadScores -> Needed to read the xml file by deserializing the stream and returning the high score

SaveScore -> Uses the LoadScores function to open the file and then iterates through it to rearrange the values in an ascending order taking into account the current game score and saving it as "Yours" in the high score table (if the player has beaten the score of the other 4 defaults). The function is called at the GameOver state in the Game1.Update().

DisplayHighscores -> Uses the LoadScores function to open the high scores file and then line by line it displays it on the game window. This function is called at the GameOver state in the Game1.Draw function

Discussion

A discussion of the use of profiling software to improve the performance of your game engine

A profiler can be a useful tool in improving a game performance as it breaks down the time spent running each function and illustrates where the bottleneck of the code is. Using a profiler can point out which functions take the longer to process by measuring the execution time of each of the code blocks using the CPU's hi-res timer and therefore, indicate that improvements could be made to speed it up. These improvements could be directed towards calling that function fewer times or improving the algorithm itself to perform functions faster. For example, a square root function is slow to call and often, mathematical types can be re-written by excluding it (raising to the square etc..). Since most modern game engines have their own profilers, like Unity, they are simple to use and can save time while improving performance. A few profilers that can be used with XNA are NProf, SlimTune or XNA's Performance Analysis.

However, there is still some tricks that can be done to spot slow functions in the code. For example, while the code is running, we can stop it by pressing break on the IDE and that will stop the code at the function or block currently running. Doing this a few times will most probably yield similar results as it is likely that the code will stop at the same block every time, the one taking the most time to run.

As the Pareto Principle dictates, there is an 80:20 rule or more specifically for computer science, a 90:10 rule. This means that 10% of the code amounts to 90% of the issues therefore, fixing that 10% will sort out the 90% of the problems. This signifies the importance of using a profiler to improve game performance.

References

GAMESARCHITECTURE.L02

Moodle.city.ac.uk. (2018). *GamesArchitecture.L02*. [online] Available at: https://moodle.city.ac.uk/mod/resource/view.php?id=908338 [Accessed 20 Apr. 2018].

GAMESARCHITECTURE.L03

Moodle.city.ac.uk. (2018). *GamesArchitecture.L03*. [online] Available at: https://moodle.city.ac.uk/mod/resource/view.php?id=908345 [Accessed 20 Apr. 2018].

EVENT DRIVEN DESIGN

Moodle.city.ac.uk. (2018). *Event Driven Design*. [online] Available at: https://moodle.city.ac.uk/mod/resource/view.php?id=908350 [Accessed 20 Apr. 2018].

COLLISION DETECTION

Moodle.city.ac.uk. (2018). *Event Driven Design*. [online] Available at: https://moodle.city.ac.uk/mod/resource/view.php?id=908357 [Accessed 20 Apr. 2018].

GAMESARCHITECTURE.L04

Moodle.city.ac.uk. (2018). *GamesArchitecture.L03*. [online] Available at: https://moodle.city.ac.uk/mod/resource/view.php?id=908345 [Accessed 20 Apr. 2018].

GAMESARCHITECTURE.L06

Moodle.city.ac.uk. (2018). *GamesArchitecture.L03*. [online] Available at: https://moodle.city.ac.uk/mod/resource/view.php?id=908345 [Accessed 20 Apr. 2018].

GAMESARCHITECTURE.L08

Moodle.city.ac.uk. (2018). *GamesArchitecture.L03*. [online] Available at: https://moodle.city.ac.uk/mod/resource/view.php?id=908345 [Accessed 20 Apr. 2018].

HIGH SCORE SYSTEM - MONOGAME/XNA | DREAM.IN.CODE

Dreamincode.net. (2018). *High Score System - MonoGame/XNA | Dream.In.Code*. [online] Available at: http://www.dreamincode.net/forums/topic/180164-high-score-system/ [Accessed 14 Apr. 2018].

XNA ESSENTIALS | STORING AND RETREIVING HIGH SCORES

Xnaessentials.com. (2018). *XNA Essentials | Storing and Retreiving High Scores*. [online] Available at: http://xnaessentials.com/tutorials/highscores.aspx [Accessed 20 Apr. 2018].

DOT NET PERLS

Dotnetperls.com. (2018). [online] Available at: https://www.dotnetperls.com/xmlreader [Accessed 18 Apr. 2018].

CARTBLANCHE/MONOGAME-SAMPLES

GitHub. (2018). *CartBlanche/MonoGame-Samples*. [online] Available at: https://github.com/CartBlanche/MonoGame-Samples/blob/master/XNAPacMan/HighScores.cs [Accessed 20 Apr. 2018].

MICROSOFT SUPPORT

Support.microsoft.com. (2018). [online] Available at: https://support.microsoft.com/en-us/help/307548/how-to-read-xml-from-a-file-by-using-visual-c [Accessed 20 Apr. 2018].

STACK OVERFLOW

Adding to XML file

file, A. (2018). *Adding to XML file*. [online] Stackoverflow.com. Available at: https://stackoverflow.com/questions/9761363/adding-to-xml-file [Accessed 20 Apr. 2018].

XNA: DRAWING TEXT - COMPUTER GAMES PROGRAMMING

Cgp.wikidot.com. (2018). *XNA: Drawing Text - Computer Games Programming*. [online] Available at: http://cgp.wikidot.com/xna-tut-text [Accessed 20 Apr. 2018].

PLAYING .MP3 AND .WMA AUDIO FILES IN XNA - RB WHITAKER'S WIKI

Rbwhitaker.wikidot.com. (2018). *Playing .MP3 and .WMA Audio Files in XNA - RB Whitaker's Wiki*. [online] Available at: http://rbwhitaker.wikidot.com/another-way-to-do-audio-in-xna [Accessed 20 Apr. 2018].

EXPLOSION SPRITE

Available at: http://i.imgur.com/dTWlsr0.png

Available at: https://forums.uberent.com/threads/reference-papadump-exe-papatran-exe.48386/page-2