Contents

Instruction Manual....2
Content Management....3
Control of game character....3
Collision detection....4
Moving and animated game element....5
Configurable game world....6
Collision response-based removal and separation of collision detection....7
Scoring System....9
High score table....9
Scenes....10
Power up (Collectable Item)....11
NPC (Enemy) control by FSM....12
Overall game play....12
Widgets....13
Audio....13
Use of Profiling....13
References....14

Instruction Manual

This is a side scrolling 2.5D shooter game. Upon entering the level, the game will begin immediately and start shooting automatically. Player can also perform a special attack manually that instant kill an enemy with the corresponded number. For example, when player press the number pad 0 key, the closest enemy with code 0 will get killed right away (code is written on the enemy body), and this special attack and be used 3 times in total and recharging 1 attack every 5 second. Aside from attacking, player can also move around freely inside the view zone.

The gameplay is to avoid getting hit by enemies, at the same time collect items and destroy enemies to achieve high score. Player earn score by destroying enemy and collecting power up item. There are 2 types of power up item can be collected in the game, the first one allows player to shoot an extra bullet, and the second power up allows player to shoot more frequently.

The victory condition is to survive from 10 waves enemy attack and archiving the highest score. On the other hand, defeat condition is when player get hit by either an enemy or an enemy's projectile.

The game has only keyboard control and is listing as the below table

W	Move up
A	Move Left
S	Move Down
D	Move Right
Enter	Go to next scene
Esc	Exit Application
Number pad 0	Special attack that attack enemy with code '0'
Number pad 1	Special attack that attack enemy with code '1'
Number pad 2	Special attack that attack enemy with code '2'
Number pad 3	Special attack that attack enemy with code '3'
Number pad 4	Special attack that attack enemy with code '4'
Number pad 5	Special attack that attack enemy with code '5'
Number pad 6	Special attack that attack enemy with code '6'
Number pad 7	Special attack that attack enemy with code '7'
Number pad 8	Special attack that attack enemy with code '8'
Number pad 9	Special attack that attack enemy with code '9'

Content Management

This application has used a resource management strategy. Meaning only appropriate assets will be loaded and will be unloaded when they are no longer needed (exit the scene).

The management is done by create a separate content manager for each scene. For each scene, the corresponding content manager will load the assets before the scene is being initialise and unload the assets before exit the scene.

The reason of separating content manager for each scene is because each manager can contain different member variable (models and textures), also by separating the managers, they can adapter to different situation such as load asset depends on the current scene player is in.

Control of game character

The game character is controlled by key press as shown in the table of Instruction Manual.

This is implemented using an event driven architecture and polling technique. A command manager is created to handle the polling and key binding serving as game-engine code. The polling listens for any changes on key state and fire the corresponded keyboard event (key up, key down, key press). The event then takes the key and key state along to invoke any registered functions. Since there are numbers of key can be listen, it will too expensive to call if listening to every key on the keyboard. Therefore, the command manager defines a dictionary to store the key that it should listen to, as well as mapping a game action to the key, so the game action can be executed along the key state changes.

The keys and game actions that control the character is implemented in the game class (ChaseCameraGame.cs) as game code. User can bind any game actions with any keys easily by calling the AddKeyBinding function with the key and game action as parameter, they will then be added to the list and being listened by the command manager.

Another layer has been added for the control of game character. Upon key state changes, instead of performing the action immediately, it fires another event with key state and amount (float) as parameter. The amount can be used to indicate the distance of the player translate, or the special attack of the player performs. The action will then be calculated in the event listener in the player class using a switch statement for player event type.

The reason of adding this layer is to allow movement being handled in a single function to serve as game-engine code. By adding this, engine user can find and make modify the actions in a single function in the player class (Player.cs).

In addition, each scene has a separated keyboard control mode, menu, level, and end screen. By adding different control mode, it prevents the system to listen for irrelevant key press to the scene (i.e moving character in menu) as well as allows user to bind a different game action to the same key.

Collision detection

The collision detection is implemented using a polling technique and a brute force technique and handled by a collision manager.

Every time a collidable game object is created, it will be added to the list for the manager to handle. Each collidable game object will have a bounding box, and the manager will compare the bounding box on every frame, and this is the polling technique used. In addition, the manager compares every collidable game object in the world, and this is the brute force technique.

After detecting a collision, it will be store in a list in the manager and resolved after finish comparing every collidable object. The collisions are resolved by calling the OnCollision function of the game object, this also allows engine user to implement collision responses for individual class.

Moving and animated game element

Several game objects are animated: camera, background, player character, enemy, and projectile. Some of the game objects are also moving by several factors and they are all implemented frame rate independently.

The camera is set to constantly scrolling left, at the same time player is set to constantly moving to right with the same amount of velocity. This create an impression of game objects coming from the right side to the left, and player moving from left to right in the scene.

The game objects also have their own velocity. Enemies have velocity moving to left, up, and down based on the wave setting and life time. Power up items are resting but also look like moving to left because of the camera and player setting. The projectiles are moving at a constantly velocity with player projectiles to the right, and enemy projectile to the left. Player character can move freely in the view zone, applying velocity with WASD as mentioned in the previous sections.

The background scrolling effect is done by the camera, therefore the background is resting at all time unless the background has gone out of the view zone. The background is made of three planes with the same seamless texture. Upon the left plane leaves the view zone, the right plane will be placed on the left of the left plane to extend the background.

The frame rate independent animation is controlled by game loop. A game loop is executed every frame, the different in time between each frame is known as elapsed time. by multiplying the velocity to the elapsed time, it guarantees that the object moves the right amount of distance in every second without concerning the frame rate.

Configurable game world

The game world can be configured using the data driven approach. By modifying different values such as the world boundary, camera scrolling speed, and enemy information etc. the configuration can be done by an external text file.

In this application, the configurable game information is stored in the XML files. The information is separated in two types, environment information and wave information. The environment information handles the values that is relevant and affecting the game world such as background scrolling speed and world boundary etc. The wave information contains the values for the number of power up item and enemy in the wave, the starting position, and the types of them. There are values that configure the attack of the enemy, and movement that is based on the trigonometry functions. These values are read from the file whenever the initializing the game world. The values will then be parsed and configure the world. For the wave information, the wave pattern is consisted of the 7 lines of characters and dots. Each character represents a different game object, and the dot represent the distance between the game objects. By reading in the character, different spawning function is called to create the corresponded object. For example, the capital letter A represents an old plane that moves in a trigonometry wave form, and a small letter a represent an old plane that moves in one direction. The trigonometry wave form can be configured in the XML file(waveInfo.xml) as the wave setting. Engine user can add different trigonometry function inside the Movements tag.

```
<Movements>
    <string>Cos</string>
    <string>Sin</string>
</Movements>
```

When reading the value, system will parse the string and register the correspond trigonometry function to a delegate and pass it to the enemy setting.

Collision response-based removal and separation of collision detection

The collision detection is separated for different classes, as mentioned in the previous section Collision Detection, the collision is resolved separately for different class by calling the OnCollision function. The collision responses are also separated by comparing the type of the object as shown in the code snippet below:

```
public override void OnCollision(Collidable obj)
{
    //if player is already destroy, return
    if (!active) return;

    //if collide with an enemy, destroy player
    if (obj.GetType() == typeof(Enemy))
    {
        this.active = false;
        FireDestroyEvent(new RemoveGameObjectEventArgs(0));
    }
    //if collide with a projectile from enemy, destory player
    if (obj.GetType() == typeof(EnemyProjectile))
    {
        this.active = false;
        FireDestroyEvent(new RemoveGameObjectEventArgs(0));
    }
    //if ocllide with a power up, remove the object from the world,
    //event will be sent to here from powerup manager
    if (obj.GetType() == typeof(PowerUp))
    {
        PowerUp pu = obj as PowerUp;    //if cast succeessed
        if (pu != null)
        {
            pu.Active = false;
        }
    }
}
```

The code snippet is taken from the Player class (Player.cs). The function performs different action depends on the type of the collided object. By colliding to power up item, the power up item will be set to inactive meaning it will get destroyed by the observer (Power up manager). On the other hand, by colliding to an enemy or enemy projectile, it will set the player to inactive meaning the game is over.

The collision-based removal is always handled by the game object observer (i.e EnemyManager, PowerUpManager, ProjectileManager etc). The observer updates the game objects every frame as well as checking and setting the status of the game object. Upon detecting a game object is inactive, the observer will remove the object from the list, leaving it as inaccessible object that will be collected by the garbage collector. At the same time the observer fires an event to notify that an object is destroyed. By firing the event, other observer can also perform a relevant action to it (i.e add score, play audio etc.).

Apart from the bounding box collision detection mentioned in the previous section, a game world boundary check is also implemented. An enemy object has two states finished and onscreen. Finished means the enemy has moved out the view zone on the left, meaning that player can no longer interact with the enemy, and the object should be removed, therefore set the object to inactive. OnScreen means the enemy is not yet finished but also not yet enter the view zone. For a better game play experience and performance of the application, the enemy will not perform the combat action (shooting and moving up and down) unless the state onScreen is true.

In addition, player is also limited by the world boundary to prevent player character leaving the view zone.

Scoring System

Scoring system is implemented using the event listener and delegate. Every time the player performs an action that could earn score (i.e killing enemy), a scoring event will be fired by the observer (ScoreManager), and functions that is registered to the delegate will executed. Reason of using an event system and delegate is because the score can be added immediately after the action happen. By using a delegate, it allows another observer (i.e WidgetManager) being notified and update the object (i.e score on the screen) at the same time.

In the game, scoring is done by per frame. For example, killing an enemy gives 30 score to the player, but it is not directly added to the player. The score will be stored in a variable named "score to add". For each frame, if "score to add" is greater than 0, take 1 away and add it to the player score. This implementation create an impression of player scoring overtime.

High score table

A high score table (leaderboard) is implemented to the game and display at the end screen using the game state load/save mechanism, meaning that the game state (score) is stored in an external file. The high score table is updated every time entering the end screen by the observer (ScoreManager). The manager first read the high score table from file and deserialized it to integer, then compare the current score of player to the table. Starting from the top to bottom(highest to lowest), if the player score is higher than the score in the table, add the player score to the table, sort it from largest to smallest, and remove the 6^{th} items (smallest score). After updating the table, the manager will then serialize the table and write it to a XML file to save the table.

The deserialization is done using the loader class (Loader.cs). By given the file name, and type of the object (leaderboard). The loader deserializes the file to obtain the values, set the value for the object, and return the object.

The serialization is done using the loader class (Loader.cs), it takes in an object of a leaderboard type, serialize the object and write it to the file (leaderboard.xml).

Scenes

There are three scenes in the game: menu, level, and end screen. Player can go from menu to level, level to end screen, and end screen back to menu. The transition between scenes are handled by a finite state machine as well as using the event driven architecture. The finite state machine is implemented in the observer (SceneManager) and listen for changes of the state as the code snippet below.

```
inMenu.AddTransition(new Transition(inGame, () => bChange));
inGame.AddTransition(new Transition(inEndScreen, () => !level.Player.Active));
inEndScreen.AddTransition(new Transition(inMenu, () => bChange));
```

This means, while current state is inMenu, player will go to the game level when bChange is true, and the bChange will be true when player press the enter. While in the end screen, the observer also listen for the changes of bChange (the Enter key), and transit to menu when the condition is true.

For the game level transition. It transits to end screen when player is not active anymore, and the state is changed when player get hit by an enemy or enemy projectile.

Although the finite state machine handles entry and exit of the state, an event listener is implemented in the main class (ChaseCameraGame.cs) serving as game-engine code. It allows engine user to add function calls such as play audio and enable keyboard input etc. when the scene load (OnSceneLoad), or stop audio and disable keyboard input etc. when the exiting the scene (OnSceneDestroy).

The events are fired from the scene object base class (Scene.cs). Upon initialise function is being called, fire the load event. On the other hand, destroy event is fired when destroy function is being called.

The reason of using an event driven architecture and firing the event from base class is because this allow engine user to handles these situation easily in the main class. Firing the event from base class also allow user to create their own scene (i.e level2, splash screen etc.), and still firing the event by default.

Power up (Collectable Item)

The power up object is a derived class of Collidable (Collidable.cs). By default (from the base class), it updates the model view matrix every frame, as well as position of the bounding box. Since it is a collidable object, it also allows engine user to implement their own collision response (OnCollision function).

The power up action is done by event listener. When player collide with the power up object, the power up object will be set to inactive to prevent another collision. At the same time the observer (Power Up Manager) is checking the state of the power up. If the power up is inactive, the observer fire an event along with the bonus effect of the power up. The player then listen to the event and update the state based on the bonus effect type (double attack or speed attack) as the code snippet below.

```
        public void OnPowerUp(object sender, PowerUpEventArgs e)
        {
            switch (e.PowerUpEventType)
            {
                case PowerUpTypes.DOUBLE_ATTACK:
                    //increase the number of bullet fire at once, maximum 4
bullet can be fire at once
                    numberOfAttack = Math.Min(4, numberOfAttack+1);
                    break;
                case PowerUpTypes.SPEED_ATTACK:
                    //increase the number of attack can be done in 1 second
                    //maximum 6 attack can be done in 1 second
                    attackPerSecond = Math.Min(6, attackPerSecond+1);
                    fireTime = TimeSpan.FromMilliseconds(1000 / attackPerSecond);
                    break;
                default:
                    break;
            }
        }
```

NPC (Enemy) control by FSM

The enemy is controlled by finite state machine to transit between states (idle, combat, and flee.). The finite state machine listens to the changes of state every frame for every added states and transitions.

The idle state is the initial state of the enemy, and transit to combat state when enemy enter the view zone. The combat state can transit to the flee state when the HP of enemy is getting low.

During idle state, enemy will be moving in one direction only, from right to left. By entering combat mode (enter the view zone), enemy will start shooting and moving up and down (follow the setting). However, if enemy taking too much damage (HP <= 1), it will transit from combat to flee and start moving up or down (whichever is closer to the edge) to leave the view zone as soon as possible.

Overall game play

There are 10 level (waves) in the game. Starting from easy level to harder. At level 1, there will be a few enemies that cannot attack nor moving up and down. The purpose of this level is to allow player to get familiar with the control. The second level will then have enemy that shooting at the player, player will start dodging projectiles from this level. In the third level, the movement of the enemy will become a little more complex by moving up and down. After the first three levels, several types of enemy will be added in the same wave, making the level more complex and harder. At higher level, the attack rate and HP of the enemy even get higher as well as the number of enemy increase. In addition, there are power up items at the first three level can be collected easily that helps the player to build up the power for the harder level.

The wave is cleared when the last enemy of the wave is destroyed (either killed by player or moved out the scene). The system will then read and load another wave until the last wave. By completing all 10 waves, the game will be over, and leaderboard will pop up.

As mentioned before, player can kill the enemy by pressing number pad keys. Each enemy has their own code (0 to 9), by entering the number player can instant kill the enemy. The code is shown on the enemy body inside a white circle. This is done by changing the texture of the model based on the code in EnemyManager (ChangeSkin function). In addition, different type of enemy has a different model, the model is also changed by the EnemyManager (ChangeModel function). The function return a model reference based on the enemy type.

Widgets

Widget is implemented in the game as a 2D element to be drawn. The widgets are always drawn after the 3D object. The background of the menu and end screen, the text of the leader board, and HUD in the level are all done by the widget. An observer (Widget Manager) is used to manage all the widgets. In the manager, engine-user can create their own widget by adding widget image and widget text. By default the text will be drawn after the image. However user can draw a single text or image by calling DrawWidgetText or DrawWidgetImage from the WidgetManager.

To create a widget, user can either create it by the default constructor, or passing an owner as the parameter, which could be used to obtain information from the owner. A widget image can be created by default constructor, or taking in a 2D texture, or taking in scale, position etc. A widget text can be created by the default constructor as an empty text, or passing in the text, font style, scale, position etc.

Audio

Different background music is used in different scene. When changing the scene, the current music will get unloaded, and next music will be loaded and played. Other sound effects are loaded along with the background music but only play when certain condition is met. i.e. the explosion sound effect is played when an enemy is killed by player, scoring sound is played when the score is adding up etc. The function that play the sound effect is called in the event listener (OnPlayerFire, OnEnemyDestroy etc.).

Use of Profiling

The game engine can be optimized using profiling software. The profiling software shows the number of times each function being called, also the time taken for execution. Meaning that developer can easily find out the heavy function by simply looking at the graph, and directly optimize the function that is affecting the performance.

For this game engine, statistical profiler could be preferred over instrument profiler. Because this is a game engine for fast pace shooter, it is important to see the performance at full speed, rather than estimate the performance by looking at the execution time. However, if the developer has already targeted functions that need to be optimized, macro profiler could be

preferred over statistical and instrument profiler. Since it allows the developer analysis only the targeted function, or even part of the function only. This avoid reducing the performance of the game engine, at the same time also giving accurate result of the execution time.

References

58pic.com.

(2018). *ÞÍ·ÞÞÕ····cTʸÞÞc·····_jpgÞÞ'_1000····_···2630 2948-ğcÞÞ*. [online] Available at: http://www.58pic.com/newpic/26302948.html [Accessed 18 Mar. 2018].

Dafont.com. (2018). *DS digi - Search - dafont.com*. [online] Available at: https://www.dafont.com/search.php?q=DS+digi [Accessed 24 Mar. 2018].

Free3d.com. (2018). *Futuristic Combat Jet Rigged Free 3D Model - .3ds .obj .dae .blend .fbx .dxf .stl - Free3D*. [online] Available at: https://free3d.com/3d-model/futuristic-combat-jet-rigged--94053.html [Accessed 18 Mar. 2018].

Free3d.com. (2018). *Mi-24A Hind Free 3D Model - .3ds .tga - Free3D*. [online] Available at: https://free3d.com/3d-model/mi-24a-hind-17508.html [Accessed 18 Mar. 2018].

Models, F., model, F. and model, F. (2018). *F4U Corsair | 3D model*. [online] CGTrader. Available at: https://www.cgtrader.com/free-3d-models/aircraft/historic/f4u-corsair-a98a641284cfd4a31d5674ed0e2e6d5f [Accessed 18 Mar. 2018].

Koenig, M. (2018). *Bullet Whizzing By Sounds | Effects | Sound Bites | Sound Clips from SoundBible.com*. [online] Soundbible.com. Available at: http://soundbible.com/1875-Bullet-Whizzing-By.html [Accessed 11 Apr. 2018].

PixelsTalk.Net. (2018). *Desktop Sky Backgrounds | PixelsTalk.Net*. [online] Available at: https://www.pixelstalk.net/desktop-sky-backgrounds/ [Accessed 18 Mar. 2018].

Wallpapercave.com. (2018). *Steam Community :: Guide :: The Most Vaporwave/Aesthetic*. [online] Available at: https://wallpapercave.com/w/wp2337010 [Accessed 18 Mar. 2018].

YouTube. (2018). *Sound effect 03 AudioClip UI score sound*. [online] Available at: https://www.youtube.com/watch?v=UvnLpM1ajlI [Accessed 11 Apr. 2018].