

Hello Reaper

TDD

GAM150S17KR Spring 2017

Hello
Reaper



Producer
Cho, Yong Won

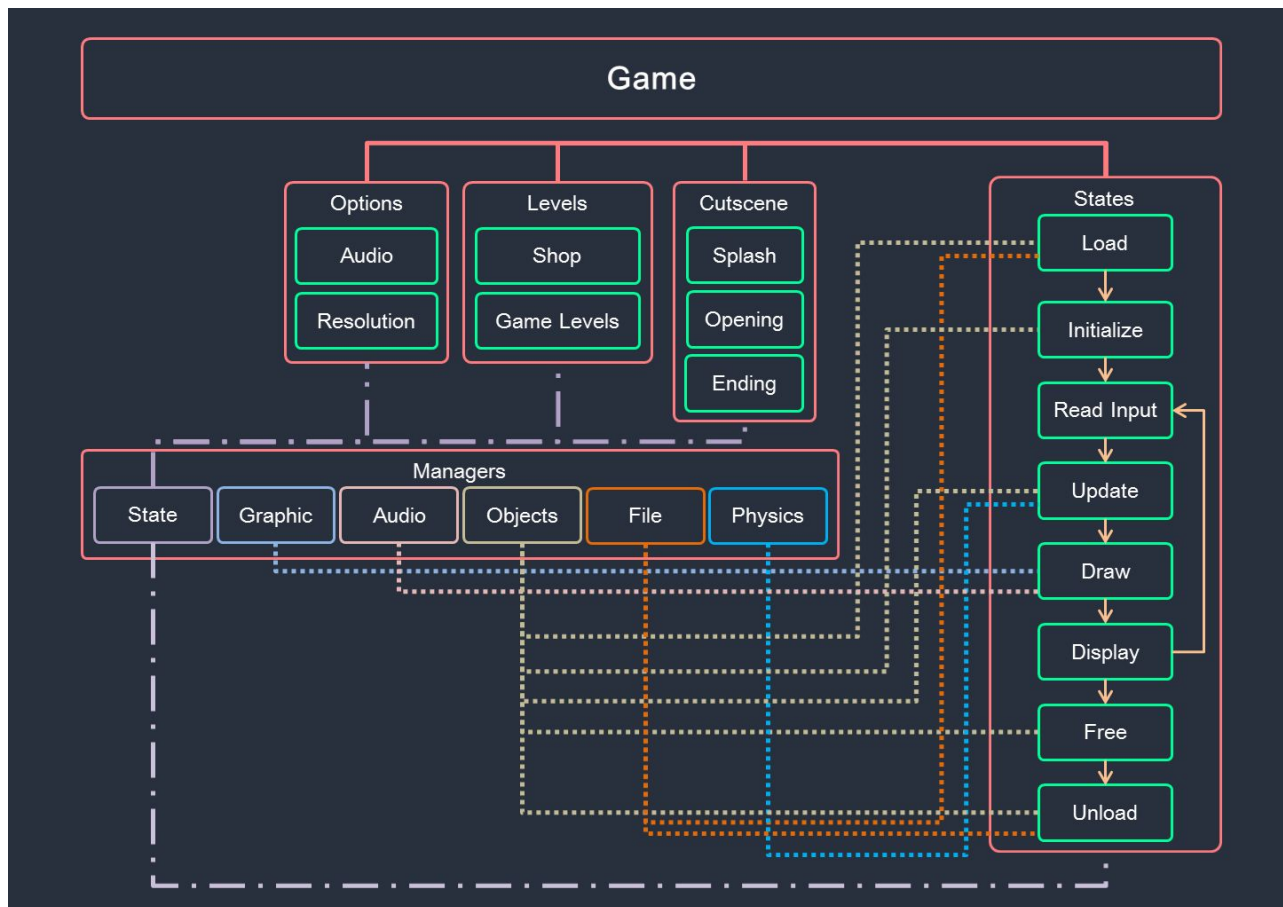
Lead Designer
Lee, Gyu Hyeon

Technical Director
Hwang, Chan Ill

TABLE OF CONTENTS

Architecture Overview	3
Graphics Implementation	3
Physics Implementation	4
Player Control Implementation	5
Behavior Implementation	6
Coding Method	6
Debugging	8
Tools	8
Scripting Languages	9
Technical Risks	9
Appendices	10

Architecture Overview



Graphics Implementation

Before using the Warp Engine

The code had DrawRectangle Function that draws a rectangle at the certain location. However, we found out that developing the graphic engine takes a lot of time and is beyond our skills. Therefore, we decided to use the Warp Engine only for its graphic renderer.

After using the Warp Engine

Draw function is a good way to draw a object on the intended place. It does same thing as our DrawRectangle function to draw a rectangle and place it. However, the biggest difference between our engine and the warp engine is texture coords function. It can load image files and make animation with any files and this is the part we could not to in our “engine”.

Physics Implementation

General

Every movement in the game 'Hello Reaper' is time based, which means the objects move same amount of distance for the same amount of time despite the change of the frame rate. Furthermore, as the game is 2D top down shooter, there is no need for gravity. However, there are some devices to fake the player that there is gravity.

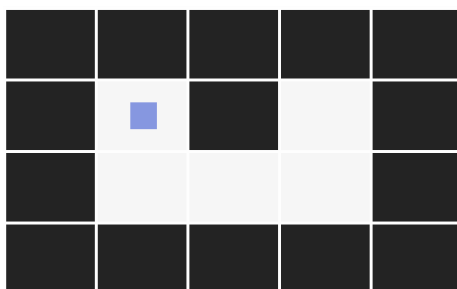
Player

- **Movement**

The player moves tile based, so the game have to know the advanced tile and figure out whether the certain tile is empty or not. In this case, point based calculation is being used to figure out this problem. For example, think about a tile map with 4 * 4 tiles(Black ones are the walls, white one is the tile map, and the blue one is the player). As the basic move length is 2 * tile size, the game will set a point at the destination (Player.Pos + 2 * Tile Size) and check whether the tile is available or not. However, there's a problem if the game checks only the destination because the player might jump over the obstacles. Therefore, the game checks the first tile(Player.Pos + Tile Size) and then check the next tile(Player.Pos + 2 * Tile Size). If the both check succeeds, the player will 'accept' the new player position.

- **Attack**

The game is currently using the basic version of ray cast - also know as the hit scan. Just like the player movement, whenever the player attacks, the game sets a point at the destination and draw a line that connects the player position and the destination. Then, the game will check how far the enemy is. If the distance is smaller than the enemy's 0.5 * size, that means the enemy is colliding. This method is simple, efficient, and instant because the game will not check the next enemy if the enemy is far away then the previous enemy. That means the game is not checking the whole objects whether they are colliding or not. Second, as the game is calculating this for one frame, the result will be applied one the next frame.



Enemy

Unlike the player, enemies' movement is not fixed on tiles. They attack and move, completely ignoring the tile maps.

- **Chaser**

Chaser type enemy has two movements. If the player is too far away, the chaser moves constant amount of distance per time. If the player is close enough, the chaser dashes towards the player.

- **Turret**

Turret type enemy does not move, so there is no movement physics for this type of enemy. For the attack, just like the previous the player attack, the turret will make a bullet, giving it a velocity so that the bullet can go towards the player.

Collision

- **ID**

Each object has own ID, which can be used to distinguish whether it should collide with something else or not. Centering the 0, the positive numbers are used for the player and negative numbers are used for enemies. The game can multiply the two IDs and make them collide only if the result is negative. For example, bullets spawned by the enemies will have same IDs as the creator so they will not collide because the result is positive ($-1 * -1 = 1$).

- **Circle to Circle Collision**

Used for common object to object colliding such as bullet and the player or enemy and the player.

- **Rectangle to Rectangle or Circle to Rectangle Collision**

Used for object to wall colliding. For example, enemy should stop or reflect when it hits the wall.

Other

As mentioned in GENERAL part, there is no gravity for the entire game. However, here are some devices to fake the player and make him or her believe that there is one. First, rounds fired by the player will remain and fall to the ground. Second, objects destroyed will also fall to the ground.

Player Control Implementation

Device for Input

The game support both the Keyboard and the Game Controller for the input. If the controller is plugged out, the game will not stop but wait for the new controller / keyboard.

Tech for Input

In the game, the player has limited resources for movement, so the game should prevent the player moving as much as he or she want with only one input. Therefore, the game uses input lock to prevent this. If the player presses the key, its state is changed from unlocked to locked. The locked key will never respond to input. For example, after the player presses 'W' key to move to upper direction, the 'W' key will be locked. The player can move to the upper direction only if he or she releases the key and re-press it.

Actual Keys

Basic player movement will use WASD keys. Combined with SHIFT key, the player can move one tile ahead. (Detailed explanation of movement is explained in physics implementation)

Attacking will use direction keys. Same as the movement, the player can perform special skills if he or she combine them with SHIFT key.

The player can change his or her weapons using numpads.

Mouse will not be used in any circumstance.

Behavior Implementation

These are the behaviors of the enemies.

- **Finding**

All enemies is processed after the player process - which means all enemies will be receiving the position of the player in a form of 2D Vector.

- **Chasing**

Chasing the player is a basic behavior of all melee type enemies. These enemies use the player position and their own positions to calculate the vector between them as subtracting the X and Y values and dividing them with the distance will give the unit vector to the player. After the vector is calculated and multiplied it with speed, they add the vector to their position for each frame - which make them move in constant speed. Same as the player, they will only accept their new positions only if the new positions are available. For example, they will not move if the wall is in front of them because they cannot accept the new positions.

- **Shooting(Attacking) and CoolDown**

Shooting includes spawning every kind of bullet even if the spawned bullet does not move. The melee type enemies spawn bullets at their position and move their bullets as they move by equalizing the position after they move. Range type enemies spawn bullets and give them initial velocity. Every enemy has attack cooldown which is calculated using seconds per frame.

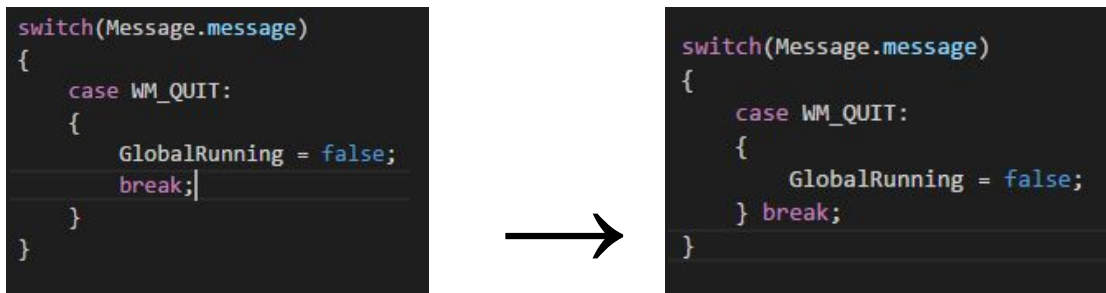
- **Dashing**

Dashing is a special behavior of the melee type enemies. These enemies suddenly move really fast if the player is close enough. The basic method is same as the method used for chasing, but in this case, the vector is not being changed until the enemies reach the destination.

Coding Method

- **Switch Statement**

Most people write 'break' inside the case statement. However, we found out that it is more convenient and easier to look when we pull this out and write outside the statement. Even better, the overall length of each statement is shorter!

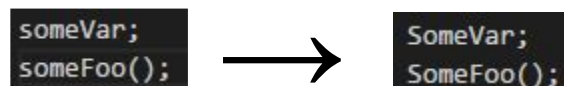


```
switch(Message.message)
{
    case WM_QUIT:
    {
        GlobalRunning = false;
        break;
    }
}
```

```
switch(Message.message)
{
    case WM_QUIT:
    {
        GlobalRunning = false;
    } break;
}
```

- **Start with Capital Letter for the functions and variables**

This might be really unfamiliar and shocking for others. However, we found out that if we start with capital letters, it is much easier to see and distinguish with pre-defined variables and functions such as windows api names.



```
someVar;
someFoo();
```

```
SomeVar;
SomeFoo();
```

- **Make sure to separate singular nouns and plural nouns while naming**

While naming the variables and even functions, we found out that it can be really confusing if we don't separate and distinguish them well. For example, BytesToLock and ByteToLock is significantly different. The first one means how many bytes we should lock and the second one means where the lock should start. Just like this, we decided to separate and name them well.

```
DWORD BytesToLock;
int32 ByteToLock;
```

- **Comments**

We have two types of comments : TODO and NOTE(or just blank).

Todo comments indicates what the author is aiming for the next step. For example, if the graphics engineer has finished loading basic images and now is aiming for the animations, he or she should write TODO(name) : Animations

Normal or Note comments are for the common informations - what this function or variable does, how this code is flowing and working.

```
// TODO(name): Fix once we have vectors :)
// NOTE(name): Be sure to mind the orders!
```

- **Class and Struct Naming**

To distinguish the variable and class name, we decided to name the class with NO capital letters and only underbars.

```
class game_state
{
};

game_state *GameState;
```

- **No Too Long Lines**

As most of us are using divided monitors, it is hard to see in one screen if the line is too long. We decided to make sure that each line is short enough to see in one screen.

```
for(int Index = 0;
    Index < EnemyManager->EnemyTurretNumber;
    Index++)
{
}
```

Debugging

We are currently using build.bat file to debug just as we want.

- **Conditions**

With a maximum warning level(w4), we decided to ignore some of the warnings that do not impact the project **at all**.

```
-WX -W4 -wd4201 -wd4100 -wd4189 -wd4505
```

- **Assert**

To safely stop the game if there is bad memory access or value, there is assert system.

```
#if HREAPER_SLOW
#define Assert(Expression) if(!(Expression)) {*(int *)0 = 0;}
#else
#define Assert(Expression)
#endif
```

- **Separated Directory**

To keep our code directory clean, we are separating our build directory so that all build related files will go into the certain folder.

```
IF NOT EXIST ..\..\build mkdir ..\..\build
pushd ..\..\build
```

- **Pause Button**

Press 'P' to make the seconds per frame 0 and see what is going on.

- **Debug Console**

We are currently only displaying the frame rate as it is the most important thing in our game.

Tools

- **Programming**
 - Visual Studio 2013 & 2015 for the compiler and debugger.
 - Visual Code for the code editing.
- **Art**
 - Pyxel Edit for the pixel arts.
 - Clip Studio for the concept arts.
- **Music**
 - Auxe for the initial prototype.
 - Logic Pro X for the complete soundtracks.
- **Game Design**
 - Articy Draft 2 for designing the game flow and the levels.

Scripting Languages

We are not using any kind of scripting language.

Technical Risks

Implementing to the Warp Engine

- **Risk**

We were not using the warp engine but using our own made “engine” to test our ideas. So far, this is a major risk until we finish implement what we have done so far to the Warp Engine
- **Mitigation**

Thankfully, we divided the game logic and the windows platform layer precisely. Therefore, the only thing we need to change is the windows platform layer part - which is graphics rendering and input processing.

Animation Rendering

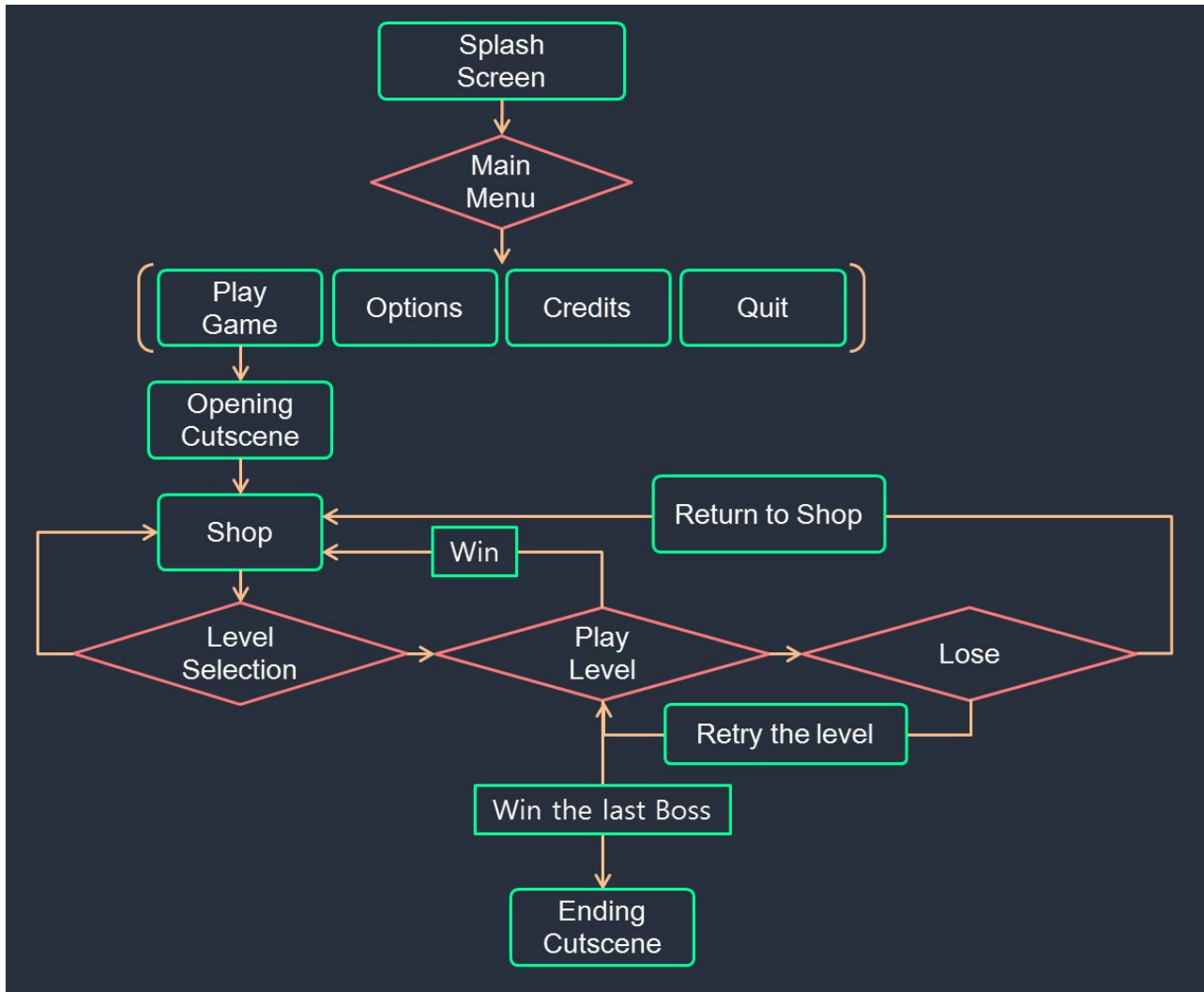
- **Risk**

It is related to our main reason of deciding that we should use the warp engine. We could not figure out how to split an image file so that we can make an animation.
- **Mitigation**

We did not have graphic manager, so we can just make a new one in the warp engine. Also, rendering is really interesting!!

Appendices

Flow Chart



Mockups

- **Splash Screen**
Where the player can see the proud digipen logo.
- **Main Menu**
Where the player can select the menu.
- **Option**
Where the player can adjust volume and resolution
- **Shop**
Where the player can select levels while seeing the structures of the levels, difficulties, and rewards. The player can also buy items to prepare for the level.

Asset Requirements

Naming Convention

All assets will be named this way.

[owner]_[part]_numbering.[file extension]

EX) Ar_MovingLeft_1.tga