Run! Starfish!

Team name: Starfish

黄思彧 刘海洋 金晨忻 孙泽宇

1 Introduction

1.1 The background of platform games

A platform game, or platformer, is a video game genre and subgenre of action game. In a platformer the player controls a character or avatar to jump between suspended platforms and avoid obstacles. Environments often feature uneven terrain requiring jumping and climbing in order to traverse them. The player often has some control over the height and distance of jumps to avoid letting their character fall to their death or miss necessary jumps. The most common unifying element of games of this genre is the jump button. Other acrobatic maneuvers may factor into the gameplay as well, such as swinging from objects like vines or grappling hooks, or bouncing from springboards or trampolines. These mechanics, even in the context of other genres, are commonly called platforming, a verbification of platform.

At one point, platform games were the most popular genre of video game. At the peak of their popularity, it is estimated that between one-quarter and one-third of console games were platformers. No genre either before or since has been able to achieve a similar market share. As of 2006, the genre had become far less dominant, representing a two percent market share as compared to fifteen percent in 1998, but is still commercially viable, with a number of games selling in the millions of units. Since 2010, a variety of endless running platformers for mobile devices have brought renewed popularity to the genre.

"Endless running" or "infinite running" games are platform games in which the player character is continuously moving forward through a usually procedurally generated, theoretically endless game world. Game controls are limited to making the character jump, attack, or perform special actions. The object of these games is to get as far as possible before the character dies. Endless running games have found particular success on mobile platforms. They are well-suited to the small set of controls these games require, often limited to a single screen tap for jumping. Games with similar mechanics with automatic forward movement, but where levels have been pre-designed, or procedurally generated to have a set finish line, are often called "auto-runners" to distinguish them from endless runners.

1.2 The libraries we use

1.2.1 Cocos2D

Cocos2d is an open source software framework. It can be used to build games, apps and other cross platform GUI based interactive programs. Cocos2d contains many branches with the best-known being Cocos2d-objc, Cocos2d-x, Cocos2d-html5 and Cocos2d-XNA. There are some independent editors in the cocos2d community, such as those contributing in the areas of SpriteSheet editing, particle editing, font editing and Tilemap editing as well as world editors including SpriteBuilder and CocoStudio.

All versions of Cocos2d work using the basic primitive known as a sprite. A sprite can be thought of as a simple 2D image, but can also be a container for other sprites. In Cocos2D, sprites are arranged together to form a scene, like a game level or a menu. Sprites can be manipulated in code based on events or actions or as part of animations. The sprites can be moved, rotated, scaled, have their image changed, etc.

The basic elements of Cocos2d include scene graphs, scene and layers, sprites and director.

To combine every part of the game, Cocos2d uses a technique called Scene Graph that keeps track of each component through a simple hierarchy.

With the method of scene and layers you can group objects using layers and define the order in which these layers must be drawn. The highest layer is on top, while the innermost is overlapped by the other layers. These layers are going to compose a scene.

Sprites are elements mainly based on images such as a spaceship and some asteroids.

The director, as the name says, is the responsible of the scene sequence. Through this element we can detect which is the current scene and move to the next one. The director is also useful to manage other information, like screen data and frame rate.

1.2.2 Pyaudio

PyAudio provides Python bindings for PortAudio, the cross-platform audio I/O library. With PyAudio, you can easily use Python to play and record audio on a variety of platforms, such as GNU/Linux, Microsoft Windows, and Apple Mac OS X / macOS.

To use PyAudio, first instantiate PyAudio using pyaudio.PyAudio(), which sets up the portaudio system.

To record or play audio, open a stream on the desired device with the desired audio parameters using pyaudio.PyAudio.open() (2). This sets up a pyaudio.Stream to play or record audio.

Play audio by writing audio data to the stream using pyaudio.Stream.write(), or read audio data from the stream using pyaudio.Stream.read().

Use pyaudio.Stream.stop_stream() to pause playing/recording, and pyaudio.Stream.close() to terminate the stream.

Finally, terminate the portaudio session using pyaudio.PyAudio.terminate()

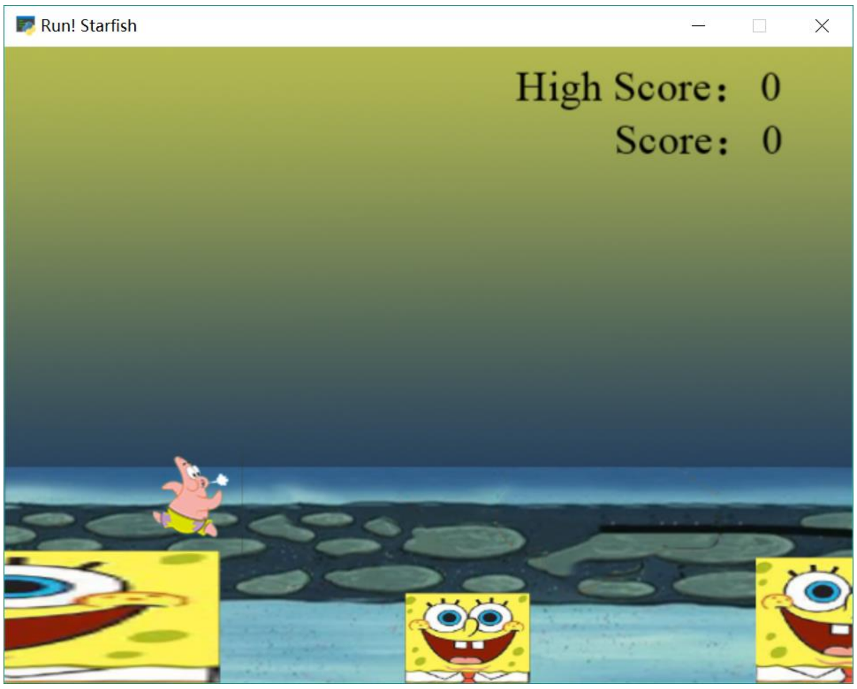
2 The rules and the objective

When the game starts, a starfish falls onto the first sponge.

The player makes sounds to control the starfish. There are two volume levels set by the developer. When the sound reaches lever 1 (the lower lever), the starfish will move forward and the moving distance is decided by the sound volume. When the sound reaches level 2 (the higher level), the starfish will jump up and then drop down and the jumping height is also decided by the sound volume. When the starfish is in the air, the player cannot make sounds to make it jump higher again, but making sounds above level 1 can keep it moving forward.

When the starfish successfully lands on the next sponge, the player will get one score. But when it falls off the sponge, the game is over and the player should restart the game at the first sponge as the score returns to zero.

The objective of the game is to get as many scores as possible. The high score will be recorded on the interface.



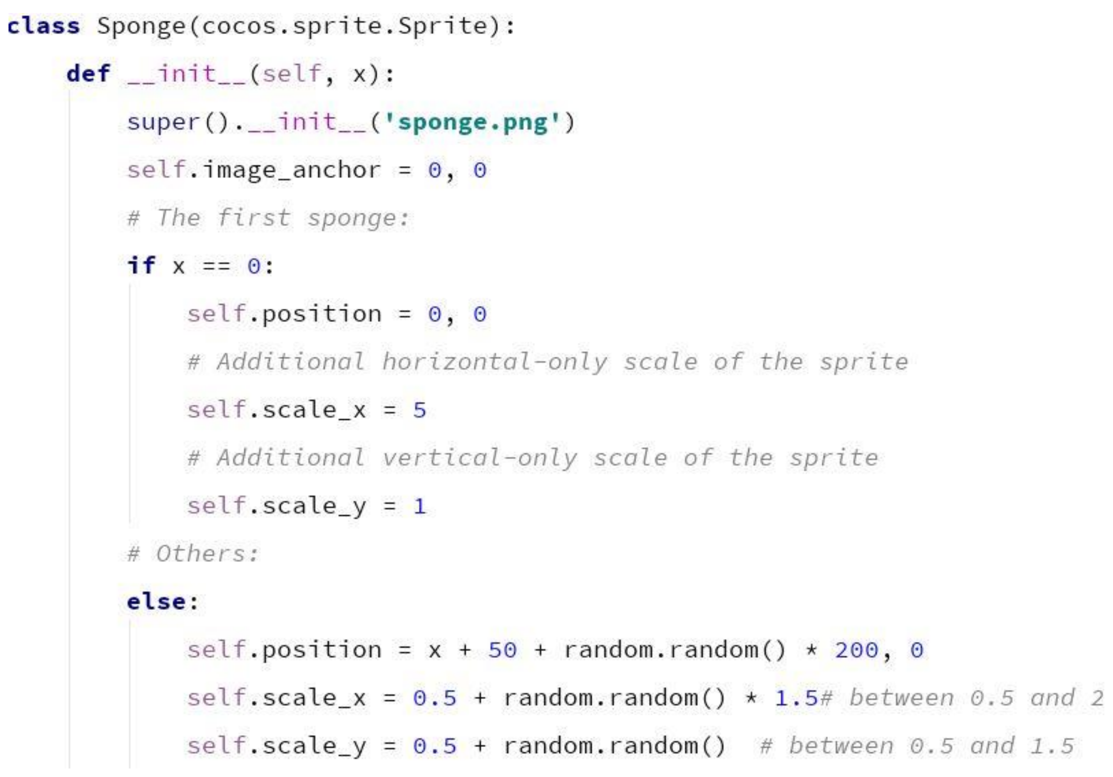
3 The development process

Some basic attributes:

Windows size = 640*480(default)
Original size of images
Background :640*480
Volume bar :100*100
Sponge: 100*100
Starfish: 80*80

Position: the left-bottom coordinate of the image.
Anchor : the point from which image will be positioned, rotated and scaled.
The range of values is (0,0) to (1,1) , while (0,0) represents the left-bottom of the image, (0.5, 0.5) represents the center of the image and (1,1) represents the right-top of the image.
scale_x & scale_y : Additional horizontal-only &vertical-only scale of the sprite.
e.g. scale_x = 2 means 2 times of the horizontal scale and scale_y =0.5 means half of the vertical scale.

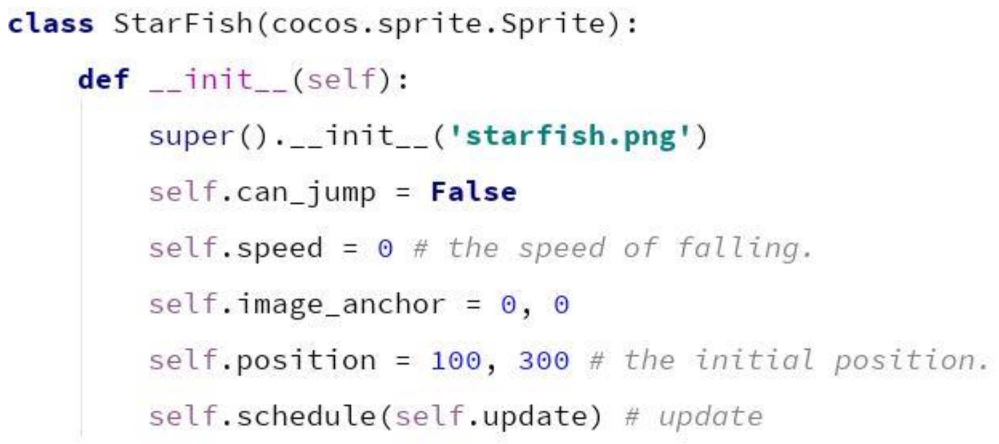
3.1 Description of Sponge



```python
class Sponge(cocos.sprite.Sprite):
    def __init__(self, x):
        super().__init__('sponge.png')
        self.image_anchor = 0, 0
        # The first sponge:
        if x == 0:
            self.position = 0, 0
            # Additional horizontal-only scale of the sprite
            self.scale_x = 5
            # Additional vertical-only scale of the sprite
            self.scale_y = 1
        # Others:
        else:
            self.position = x + 50 + random.random() * 200, 0
            self.scale_x = 0.5 + random.random() * 1.5# between 0.5 and 2
            self.scale_y = 0.5 + random.random()  # between 0.5 and 1.5
```

First, we subclass Sprite as *Sponge*, on which the starfish need to land. We set the parameter x to get the x-coordinate of the right side of the last sponge(it will be discussed in detail later).Then we set the anchor (left-bottom)of the sponge. Finally, we set position and scales for the first sponge and others(randomly), respectively.

3.2 Create the Starfish and define some functions for it



```python
class StarFish(cocos.sprite.Sprite):
    def __init__(self):
        super().__init__('starfish.png')
        self.can_jump = False
        self.speed = 0 # the speed of falling.
        self.image_anchor = 0, 0
        self.position = 100, 300 # the initial position.
        self.schedule(self.update) # update
```

Similarly, we subclass *Sprite* as *StarFish*, the character player needs to manipulate in the game. The variable *Self.can_jump* takes a Boolean value indicating whether the starfish can jump. And we set the initial speed of falling to 0. Then starfish is set to automatically update per frame.

```python
def jump(self, h):
    if self.can_jump:
        self.y += 0.5
        self.speed -=  min(h, 8)
        self.can_jump = False

def land(self, y):
    if self.y > y - 30:
        self.can_jump = True
        self.speed = 0
        self.y = y
```

Next we define the functions ***jump(self, h)*** and ***land(self, h)***.The first function allows starfish make one jump and it acquires an upwards velocity. Then it couldn't jump the second time before it lands on the sponge. The latter is used to make starfish land and still, which we will talk about further later.

```python
def update(self, dt):
    self.speed += 9.8 * dt
    self.y -= self.speed
    if self.y < -80:
        self.reset()

def reset(self):
    self.parent.reset()
    self.can_jump = False
    self.speed = 0
    self.position = 100, 300
```

These two functions are used to update the starfish per frame(帧).Here dt means delta time from the last frame, that is, dt = 1/ftp. And 9.8 represents gravitational acceleration (g = 9.8). If self.y < -80, meaning starfish disappear from the window, it will be reset, then its parent(the main game) will also be reset.

3.3 Create the main game

```python
class Game(cocos.layer.Layer):
    def __init__(self):
        super().__init__()

        self.bgm = Sound('bgm.wav')
        self.bgm.play(-1)

        self.bg = cocos.sprite.Sprite('background.png')
        self.bg.position = 320, 240
        self.add(self.bg)
```

We subclass Layer as the *Game* and add background image and music to it.

```python
BLACK = (0, 0, 0, 255)

# Record score
self.score = 0
# Create a Text Label to record score.
self.txt_score = cocos.text.Label(u'Score: 0',
                                  font_name='Times New Roman',
                                  font_size=24,
                                  color=BLACK)
self.txt_score.position = 460, 400
self.add(self.txt_score)
# Record the high score.
self.high_score  = 0
# Create a Text Label to record high score.
self.txt_high_score = cocos.text.Label(u'High Score: 0',
                                  font_name='Times New Roman',
                                  font_size=24,
                                  color=BLACK)
self.txt_high_score.position = 386, 440
self.add(self.txt_high_score)
```

Then we create two text labels to record the score and high score and add them to *Game*.

```
# Create a CocosNode floor.
self.floor = cocos.cocosnode.CocosNode()
self.add(self.floor)

# Create Sponge instances and add them to floor.
x = 0
for i in range(100):
    b = Sponge(x)
    self.floor.add(b)
    x = b.x + b.width
```

Here we create a *CocosNode* instance *floor* and add it to Game. Then we use a *for* loop to create a row of (100) sponges and add all of them to *floor*.

```
self.volumebar = cocos.sprite.Sprite('black.png')
self.volumebar.image_anchor = 0, 0
self.volumebar.position = 20, 450
self.volumebar.scale_y = 0.1
self.add(self.volumebar)

self.starfish = StarFish()
self.add(self.starfish)
```

Here we create a *volumebar* to show volume and a StarFish instance starfish and add both of them to Game.

```
NUM_SAMPLES = 1000
```

```
pa = PyAudio()
# store the stream input.
self.stream = pa.open(format=paInt16,
                      channels=1,
                      rate=44100,
                      input=True,
                      frames_per_buffer=NUM_SAMPLES)
# Parameters:
# format - Sampling size and format.
# channels - Number of channels
# rate - Sampling rate
# input - Specifies whether this is an input stream
# frames_per_buffer - Specifies the number of frames per buffer.

# Update the stream.
self.schedule(self.update)
```

We create a *PyAudio* instance pa and use *pa.open()* to get the voice input. Then we store the stream to self.stream and update it automatically.

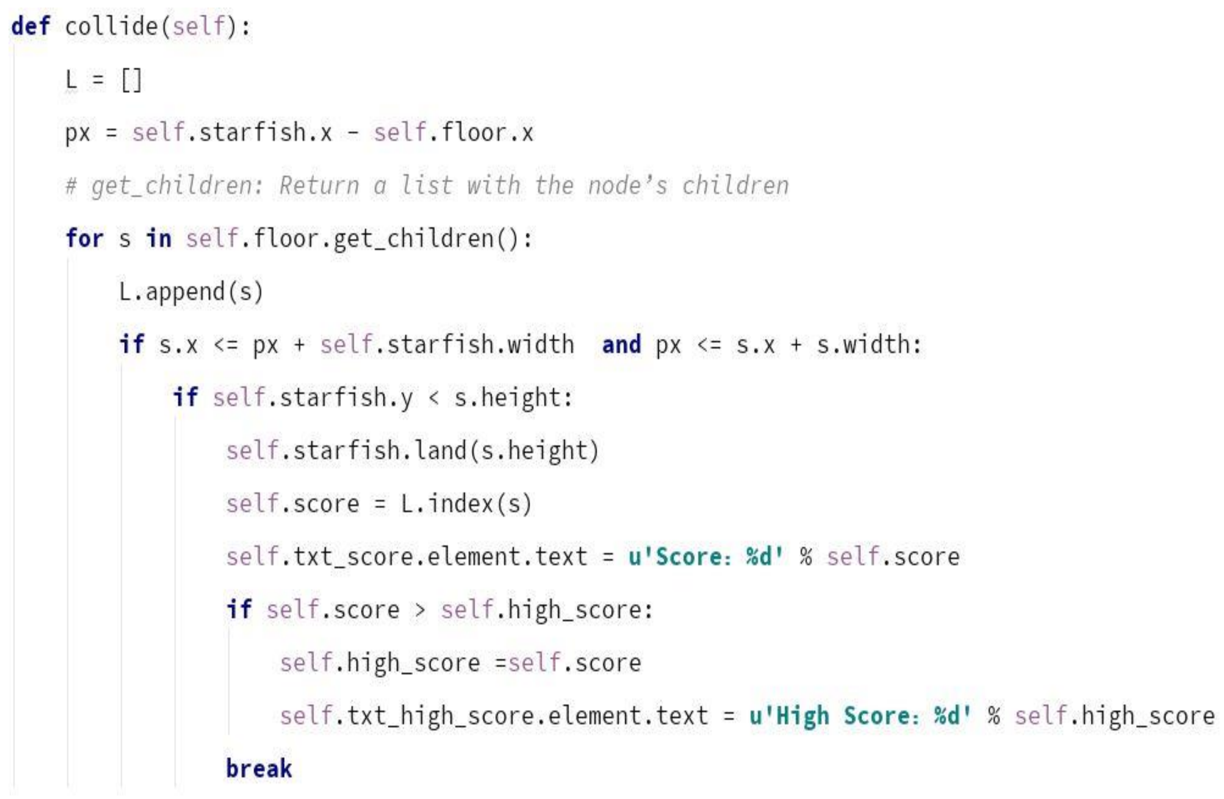
```
LEVEL_1 = 2000
LEVEL_2 = 10000

def update(self, dt): # dt=1/fps (seconds)
    # Read samples from the stream.
    # num_frames - The number of frames to read.
    string_audio_data = self.stream.read(NUM_SAMPLES)
    # struct.unpack(fmt, buffer):
    # Unpack from the buffer according to the formal string fmt,
    # Return a tuple.
    k = max(struct.unpack('1000h', string_audio_data)) #
    self.volumebar.scale_x = k / 10000.0
    # Determine moving and jumping
    if k > LEVEL_1:
        self.floor.x -= 150 * dt
    if k > LEVEL_2:
        self.starfish.jump((k - LEVEL_2) / 1000.0)
    self.collide()
```

Next we define the function update(self, dt), in which we read the stream data and unpack them to a tuple to get the max volume *k*. By comparing k with LEVEL_1 and LEVEL_2, we update the horizontal size of the volumebar and determine whether starfish will move horizontally and jump.

self.floor.x -= 150*dt means the floor moves left at a constant speed and the sponges added to the floor move accordingly as if starfish is moving forward.

The function ***self.collide()*** is shown as follows:



```python
def collide(self):
    L = []
    px = self.starfish.x - self.floor.x
    # get_children: Return a list with the node’s children
    for s in self.floor.get_children():
        L.append(s)
        if s.x <= px + self.starfish.width  and px <= s.x + s.width:
            if self.starfish.y < s.height:
                self.starfish.land(s.height)
                self.score = L.index(s)
                self.txt_score.element.text = u'Score: %d' % self.score
                if self.score > self.high_score:
                    self.high_score =self.score
                    self.txt_high_score.element.text = u'High Score: %d' % self.high_score
                break
```

First we create an empty list L and add the sponges to the list one by one, which were added to the floor before. Here ***px*** equals to the relative x-coordinate of starfish to floor, while s.x(the x-coordinate of sponge) is not changed. Then we determine if starfish has successfully landed on the sponge. If so, we update the score and high score according to the index of the sponge it has landed on and break the loop.

```
def reset(self):
    self.bgm.stop()
    game_over = Sound('over.wav')
    game_over.play()
    sleep(3) # pause for 3 seconds.
    game_over.stop()
    self.bgm.play(-1)
    self.floor.x = 0
    self.score = 0
    self.txt_score.element.text = u'Score: 0'
```

Set the function **reset(self)**. When game is over, the background music will be changed and the picture will pause for 3 seconds. After that, the music is changed back, and the x-coordinate of floor is reset to 0. And score is reset to 0, while high score keeps the same.

```
def update(self, dt):
    self.speed += 9.8 * dt
    self.y -= self.speed
    if self.y < -80:
        self.reset()

def reset(self):
    self.parent.reset()
    self.can_jump = False
    self.speed = 0
    self.position = 100, 300
```

Let’s go back to the part of starfish.py. We can see that when starfish is reset, its parent Game will be reset at the same time.

```
# Initialize the Director (which create a 640*480 window)and Mixer.
cocos.director.director.init(caption="Run! Starfish")
mixer.init()
# Then we create a Scene that contains the VoiceGame layer as a child.
main_scene = cocos.scene.Scene(Game())
# And finally we run the scene.
cocos.director.director.run(main_scene)
```

Finally, we initialize the ***director*** and ***mixer***, add Game to a scene and run the game.

4 Problems and optimization

4.1 Distance tested time and again

Through many times of trial and calculation, we finally decided the range of distances between sponges. It can neither be too large or too small, or the players may find it rather difficult to decide the commanding volume----in practice we find out that a too large distance followed by a small sponge will be very difficult for the player to decide the volume because the starfish easily flies too far to fall on the next sponge if you raise your voice; and it is the same for the too small distance – you easily miss the next sponge even if your voice are not so high. Finally, we decided that the distance should fall roughly between 50 and 250, which is relatively user-friendly, although potential of future improvements still exists, as discussed in the next part.

4.2 Never be too accurate

If the user find that he narrowly falls on the sponge—almost on the margin—and survives, he may feel very lucky. Therefore, we allowed the survival even if the starfish does not hit the sponge completely.

To realize this, we came up with a fresh idea. We did not adopt a calculation of the distance from the sponge to the landing site of the starfish, comparison with some criteria or something. Instead, we simply make the original picture of the sponge larger—by adding some blank space to the two sides of the sponge as if each sponge has an invisible part linking to it.

4.3 Details (prohibition of the second jump in the air, volume setting in different situations, etc.)

Although most details of perfection are not difficult to realize through only a couple of lines of codes, we realized that we needed them only after practice. For instance, we developed and tried the code in the dormitory, but tried different settings like classrooms and canteens to decide the volume criteria.

4.4 User-friendly interaction environment

It may sound childish but we laid great emphasis on user-friendly interaction environment. One example will be enough for you to understand this. Try our game and after a while, listen to the comforting BGM for a failure— 'It's not about winning. It's about fun!'

5 Future development

5.1 Inappropriate distance difficult to cover

Although the distance has been tested time and again, in some extreme cases, also not very possibly, the gap can reach 249.5, which will be quite difficult for the starfish to cover at a jump. This is because the distance is decided by the position as well as widths of sponges, which are both random so as to realize the flexibility. In the future a monitor of inappropriate distance should be employed so that something can be changed under that circumstance. The monitor can also make a difference if the following sponge is too small after the long distance.

5.2 Interference of noise

Noise in the room of the player can easily be recorded and lead to an unexpected jump. In the future, machine learning can be employed. Before the game starts, we record some voice as training data, let the machine learn characteristics such as tune(音调) and tone(音色), so that only the voice of the player can be a signal for action.

5.3 Level up: Interaction with others

Through access to the Internet, competition among players can be allowed.

5.4 Level up: Time limitation & Length Limitation

At present, only 100 sponges are allowed. Designers are still debating on whether to allow infinite length of game, unsure about whether infinity will bring more fun.

Alternatively, time-keeper could be adopted. This can be set as kind of limitation, or even set as part of scores.

6 Ending

Despite the time limitation in conducting the project, we still learnt a lot from numerous python libraries and succeeded in designing a game of our own, which may not be a high-end one but feasted us with not only rich sense of achievement but also a mature mindset in programing. Hopefully in the future we can go further in python, as well as in the broader world of programming.

7 Team members and work division

Name	No.	Work
黄思彧	517120910198	Introduction; Game rules; Report outline; PPT design
刘海洋	517120910200	Game rules; Future development; Ending
孙泽宇	517120910217	Problems & Optimization; Picture processing
金晨忻	517120910215	Coding development; Optimization