CSSE1001

Date: 17/10/08
Assignment 3 - Design Document
Name: Robert Knight
Student Number: 41777903
Project Title: Super Prin Prin Bash Excess

1 Description

The project described below, Super Prin Prin Bash Excess, is a computer game, developed for use with Windows platform. It is an arcade-platforming style game, similar in style to many such early 80s games such as Donkey Kong. The objective for this game is to collect 'Hopes' and reach the end of the level. Many enemies will try to prevent the player from doing such. The enemies are designed to work together to corner or trap the player, increasing the intensity, an important factor of arcade styled games. To defeat enemies, the player has a short-range melee weapon. After several levels the player must battle against a more difficult enemy, the boss character, before the game will finish. 8 levels with 2 boss battles was the planned length, however this version contains but 4 of those levels. Completing various objectives within the game will give the player points and as the game goes through each of its rounds, players can compete for the highest score. Various difficulties are available for the convenience of all players. The overall difficulty is intended to be about average for an arcade game however. If the player is hit by an enemy or a projectile they lose a life. Doing such three times will cause the player to return to the menu and have to begin the game from the beginning. Also featured are skip-able cutscenes to advance the game's storyline, described in section 2.

The code is written using Python, making use of the Pygame module available free online. Pygame is best implemented using a main loop function which loops continuously waiting for input and displaying changes. Many outside functions and defined classes are called within the loop. There is a class for each in-game object, most of which derive from the Pygame Rect class. See section 4.1 for a detailed description on the classes used in the game.

Prin Prin Bash is designed to be a simple but also fun game, considering the time frame available, but may easily be expanded at a later date due to the method for reading levels.

2 Story and Background

In the far off land of Skyre, a prophecy to change the fate of the world came true. Yune gave birth to the female twins of fate. The truth was, or so the prophecy said, one would be the embodiment of evil, the other good. Now, Yune, who had heard of this prophesy and very well believed in its entailments, had a predicament: she saw no way to know which was which. So, being as loving to her children as much as every mother should be, Yune raised them both together.

For four years the family was one without problem; without problem enough for Yune to completely misunderstand the twin's first argument. Mint desperately, more than anything else in the world, wanted Oregan's Butabu doll. Leaving verbal argument behind, Mint used physical violence on Oregan. Yune, still very much scared by the prophecy, decided it was best if Mint, clearly the evil child, was thrown into the river and forgot about.

Bad, bad, terribly awful mistake!

Oregan, the evil twin in truth, grew up praised as the saviour. Her power rose as with her age, until she became Queen, holding all of Skyre under her rule. She revealed her true alignment and Skyre became her plaything. She claimed the people as dolls, the land as a cardboard box. So cruel were her actions, the people, animals and even the plants eventually lost Hope. No one could stop Oregan and her most loyal minions.

The future was not lost however, for Mint, Skyre's to-be hero, still lived. Upon her descent into the river, years past, the water faeries took her in and raised her like the princess she was always meant to be. They taught her how to perform secret water faerie magic and to fight, how to act and best rule the kingdom she deserved. Unfortunately, the water faeries were known to be a little odd. Nevertheless, Mint now has the power of the Hake and Trout Magical Doodoo Stick. Therefore, with Stick in hand, at the age of sixteen, Mint heads towards Oregan's castle to bring her sister down! But she can't rule a desert, so she also must collect a lot of Hope along the way to give back to the life of Skyre. Go, Mint! Go!

3 User Interface

The level design within the game causes a need for the screen to be vertically oriented. For that reason, information about the player's progress such as score, remaining lives, remaining time, and Hopes left to collect will be displayed on the right. See figure 1 for Interface of the gameplay.

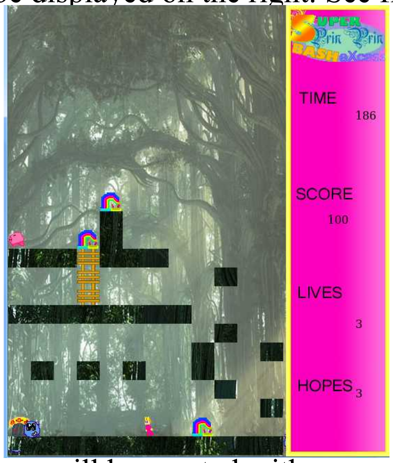

Figure 1: Gameplay screen

As an introduction the player will be greeted with a menu screen allowing the player to choose to go to an option menu, an information screen, as well as begin or exit the game. See figure 2 for initial layout design.

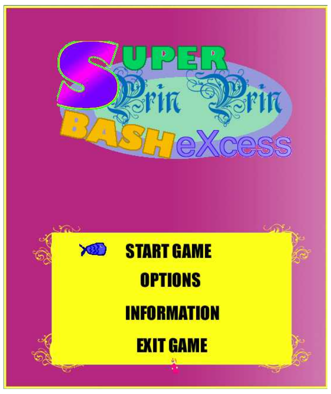

Figure 2: Menu screen

The player will interact with the game through the keyboard. Using the cursor keys and 'Z' and 'X' keys as shown in Figure 3

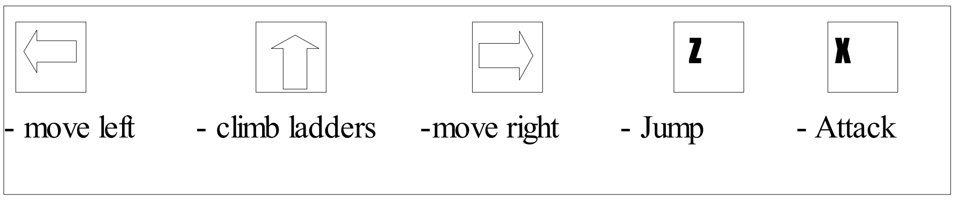

Figure 3: Default button configuration

4 Design

4.1 Classes

The game will be programmed such that each game object will be represented by a class, such that many instances of an object may be in action at a time. There are two sorts of classes: controller and Rect-based. These will be discussed in two sections below.

4.1.1 Rect-based Classes

These are classes that inherit from python.Rect. Their co-ordinates and sizes may be received by calling indexes in the list (left, top, width, height)

<u>player</u> (rect on screen for player character. More information in section 4.3)

constants:
- gravity (acceleration in pixels/game tick^2 to fall when a floor object is not within 1 pixel of the bottom of the player's rectangle)
- jumpstrength (initial speed at which to move in the -ve y-dir when jump button is pressed on ground)
- walkspeed (pixels to move left or right per game tick when the left or right button is pressed)
- swordtime (ticks to count sword collisions)

variables:
- _x (x position of player on screen)
- _y (y position of player on screen)
- _action (list of integers where each element is as in figure 4 that begins each tick all elements = 0)
- _canmove (int 1=True, 0=False) determines whether position will be changed with button inputs
- _dir direction faced (0 = left, 1 = right)
- _swordframe (int) ticks left on a single sword attack. 0 means sword is not out.
- _vspeed (distance to move vertically each tick in pixels)
- _hspeed (distance to move horizontally each tick in pixels)

functions:

```
- moveleft() Will perform the following if _canmove = 1 and _swordframe = 0
        1) If image not already the runleft image, change to it
        2) Check whether there is a wall within walkspeed of player's left and reduce
                _x by walkspeed if not
        3) Update _action[0] = 1
        Whether _canmove = 0 or not: will change _dir to 0
- moveright() (similar to moveleft())
- jump()  If _canmove = 1 and _swordframe = 0:
        1) set _canmove = 0
        2) set _vspeed to -jumpstrength
        3) set _hspeed to either 0, +walkspeed or -walkspeed depending on _dir and
        whether moving
- attack() If _swordframe = 0:
        1) set _swordframe to swordtime
        2) create instance of sword class at position passing arguments for position
        dependant on _x, _y and _dir
        3) Change player image to attacking animation
```

<u>block</u> (takes argument from level as to whether a wall or a ladder; a rect on screen of

width and height)

constants:	- Rect attributes
	- _type (type of block 1 = floor/wall, 2 = ladder)

<u>enemy</u> (for a more detailed description of enemies see section 4.4)

constants:	- type(type of enemy as int)
	- var (varient of the type, changes slight factors or abilities)
variables:	- Rect attributes
	- _vspeed (vertical speed; updates position each game tick)
	- _hspeed (horizontal speed)
	- _count (counter for timing actions)
	- _hp (number of hits left to defeat enemy)
	- _action (integer to describe current enemy action)
functions:	- various move functions to transfer the rect at a given speed

<u>sword</u> (invisible rect taking arguments from player)

constants:	- width
	- height
variables:	- Rect attributes
	- _swordtime (Clock cycles for the sword to still be used for collision detection; this is smaller than Player._swordtime, which in turn is shorter than the surface animation)

<u>shot</u>(the bullets of Enemy._type == 4)

constants:	- _var (when 0, the bullet will move to the right, 1 for left - this is different to other cases of directions because it is inherited from the enemy._var which follows usual conventions)
functions:	- move(pass in game classes difficulty setting) - moves shot based on difficulty

<u>Ball</u>(the bullets of Enemy._type == 5)

variables:	- Rect attributes
	- _vspeed (initially random, increases with gravity)
	- _hspeed (initially random, moves to 0 with air resistance)
	- _hsdown (frames to perform an integer addition on _hspeed)
functions:	- move() moves ball

<u>Hope</u>(appear at start of level, destroyed when collected)

variables:	- Rect attributes

4.1.2 Controller Classes

<u>Game</u> (controls global variables, performs main loop and passes information to level and enemy classes. All rects and surfaces in gameplay besides Blocks have Game as their parent.)

constants:	- initlives (initial number of lives to give the player)
	- height (screen height)

- width (width of gameplay area)
- screenwidth (width of screen)
- initDifficulty (initial difficulty)

variables:
- _score (current score)
- _lives (current number of lives remaining)
- _level (current level number)
- _loop (current loop through the levels, not yet implemented)
- _diff (current difficulty - may vary from initial)
- _state(determines main loop's actions)

functions:
- initlevel(pass in current level integer) - begins level in Game class, also calls on Level's drawlevel()
- updateSideBar(count) - update the side bar with information
- eventA(event) - handle keyboard inputs
- main() - described in section 4.2

Level (takes arguments from game, creates a level and controls level variables)

constants:
- layout (2D array passed from game which contains all information about level layout)
- enGenDict (Dictionary passed from Game detailing enemy generation information)
- tilesX (number of tiles to place on screen horizontally)
- deltaX (screenwidth / tilesX)
- tilesY
- deltaY
- inittime (time in ticks to be on timer at start of level)

variables:
- _nextEn (list of next enemies to spawn in order)
- _timeEn (list of time between enemy spawns, corresponding to _nextEn)
- _time (time left on timer in ticks)
- hopes(list of hopes in level)
- block (layout array turned into a block array)
- blockRect(list of rectangles in level)
- enGen(str containing enemy Generator identities which are currently active)

functions:
- drawLevel() - takes the layout (an array of ints and strs) and turns it into Rect objects
- getSpawnPos(letter) - returns the position the specified letter is at
- makeEn(pass in enemy generator letter and the master) - checks the dictionaries and creates an enemy Rect on its master at the position of the enemy generator in question

Menu(controls main menu manipulation and animation)

constants:
- screen width and height
- cursXPos (x position of cursor)

variables:
- _screen (current screen being displayed - int)
- cursPos (position of cursor as integer 0 - 3)

functions:
- initPos() reset menu screen for animation
- main() - similar to Game.main()

Cutscene(controls events during cutscenes)

constants:
- masterdisplay (the main surface of the Game or Menu which created the Cutscene instance)

- cutletter (the cutscene indentifier to load - single letter string)
- sceneTot (the number of screens in the cutscene)
- textSurfList (a list of lists of surfaces containing the text as parsed from the file)

variables: - scene (current screen into the cutscene)

functions: - main() - waits for button input and acts accordingly

4.2 Main loop

After the gameplay is initialised for that level, a loop will run constantly at approx 50 Hz. There are 4 states which completely change the actions. The main tasks for each state are as follows:

State: 'Play'

1) Initialise variables
2) Check button inputs by user and call whatever player functions are involved
3) Check for block collisions. These include player to block and enemy to block for some enemies.
4) Check for hope collisions, reacting accordingly
5) Move player according to the vertical and horizontal speeds
6) If the sword is out, check whether should be removed
7) Check whether player won the level
8) Complete all enemy movement functions
9) Check for enemy/player and enemy/sword collisions
10) Move all bullets and check whether they have collided with player
11) Spawn enemies or show spawning surface if need be
12) Redraw display in areas where there was change (or of entire screen if need be)

State 'Begin':

1) Initialise variables
2) Move text surface to correct position
3) Flip Display

State 'Win':

1) Change player surface to show winning animation
2) If time is right will create next level
3) Update display around player only

State 'Lose':

1) Change player surface to show losing animation
2) If time is right will reduce lives then reload current level. If lives are zero will show game over cutscene instead, then break main loop
3) Update display around player only

4.3 Player Class

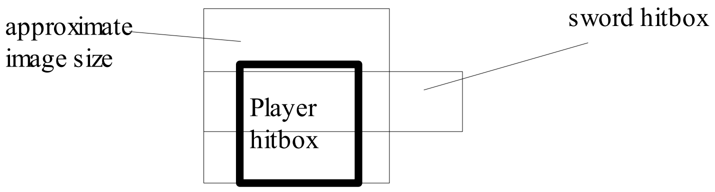

Figure 4: player rectangle sizes

During gameplay, the user interacts with the game by controlling the Player class. The Player is made up three main shapes. As shown in Figure 4, the Player class (in bold), the sword class and the player Surface (or image), make up the total Player. The size of these rectangles are a significant factor in how the game is played. Since the playfield is very narrow, the area that the sword hitbox can hit horizontally is very small. It does stick out slightly more than the player however. Getting used to the range of the sword is vital for success in the game. It should be noted that the sword box is fairly low down when compared to the sprite. This means attacks from above are more threatening. Slight ease is put on the player because of the large height of the screen but generally attacking enemies in the air can be tense for the user.

Besides enemies, the most important collision for the Player is with Blocks. There must be a block directly under the Player Surface, or gravity will cause the Player to move towards the bottom of the screen. To counter this, a player may jump. There are three ways to jump in the game. Pushing 'Z' while moving will cause the Player to do a 'long' jump, pushing 'Z' while not moving will cause the Player to do a 'standing' jump, and pushing 'Z' while not moving then immediately holding a direction will perform a 'short' jump. See Figure 5. A jump may only be performed if the Player Rect is on the ground. When moving up, left, or right, a Player cannot move through a block, and will be sent back.

Ladders act similar to blocks except the player may traverse vertically along them. A player cannot traverse horizontally on a ladder however.

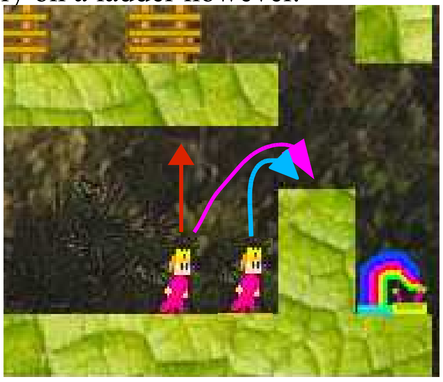

FIGURE 5: Jump Angles

4.4 Enemy Class

Unlike a Player, enemy's Surfaces and Rectangles are the same size. The reduced size of Player Rect complements this, giving an end result that looks correct. All enemies have a variety of actions which they move between, usually randomly, after set times. A short description of each enemy in the game thus far is given in the following sections.

4.4.1 Bubble Enemies (Pop)

The bubble enemies are the most commonly occurred within the game. After a short time, determined by the difficulty level they will choose a direction and move that way until the next check. They are effected by gravity. They are the easiest enemies to defeat when at the same horizontal level as the player. Bubble enemies are most threatening however when changing vertical levels. In areas with little space between one line of blocks and the proceeding line above or below it the player must react quickly to their movements to get out of the way.

4.4.2 Pig Enemies (Butabu)

Next most common enemy in the game is the pig enemy. Pigs cannot move vertically, nor will be effected by gravity. In fact, they will turn away from a pit if the approach one. Normally they move as fast as the bubble enemies. However, if the Player is at the same height, the pig will increase its speed. On creation, a pig is decided to be either a 'jumping' pig or a 'ground' pig. The main danger with pigs is the player cannot be sure which sort the pig is until they manoeuvre very close to the pig. The two types require different actions from the player. A jumping pig can be walked straight under if the player does not pause in their approach to the pig; if they do such to a ground pig however they will be hit. Likewise a ground pig can easily be defeated by standing still, waiting until it is in the range of the Sword then striking; a jumping pig can go over the top of the sword if this is done and hit the player from above.

4.4.3 Heart Enemies (Luv)

Heart enemies are the hardest to hit in the game. They always home on to the player no matter the time, although 50% of the Heart enemies created will not home exactly onto the player's position. Because of their difficulty heart enemies usually only appear late in a level if the player is taking too long. One of their features is moving away from the player when the sword button is pushed. They do so rapidly enough that unless the player strikes with perfect timing, the heart will escape. Also, because of their homing, they will often be approaching the player from the vertical, which has already been proven as a weakness for the player.

4.4.4 Hornet Enemies (Gondi)

Hornet enemies move up and down the screen rapidly, using a similar random decision counting system as the bubble enemies use. In their decision making, hornets also choose a position above the player to fire. These are: where the player is standing, at a height above the player equal to the peak of a jump and at a height halfway between these positions. The point of this is that a pair of hornets can create very difficult to avoid bullet patterns. The first may shoot a bullet straight at the player just as the next shoots one at a jump spot so the player is not safe from the shots on the ground or in the air. Hornets movements are restricted to the edge of the game screen, but can travel these very fast. To help the player defeat these creatures, no collision is detected between the Player rect and the hornet Rect. So a player can pass through a hornet. However, if the player gets the timing of a sword strike wrong, a hornet can get a shoot at point blank range and hit the player.

4.4.5 Volcano Enemies (Major Lieutenant Volc)

Volcano enemies are the only standard enemies to require more than one hit to defeat. They do not move and simply fire Balls at a rate determined by the game difficulty. A volcano at close range is almost completely safe, but an unattended one (or more) situated above the player can quickly cause many Balls to descend. A ball follows an randomly determined arc so that even though the volcano stays still it can fire at different positions.

5 Support Modules

The project makes much use of Pygame, a module which is not included in the standard Python download but is available at http://www.pygame.org/download.shtml

6 References

Many ideas for code in Prin Prin Bash was influenced by articles available on the Pygame official website. No code was directly taken, however.

The majority of graphics were personally created by the game developer, except for the background and foreground image of each level (besides background for level4). Such images were obtained and used without permission from the following sources:
- http://textures.forrest.cz/
- http://wall.alphacoders.com/

All sounds and music were personally created and composed using 3xOsc in Image-Line's Fruity Loops 7.

7 Technical Support

For technical support, please email s4177790@student.uq.edu.au
We will be happy to answer any questions or queries about the project.