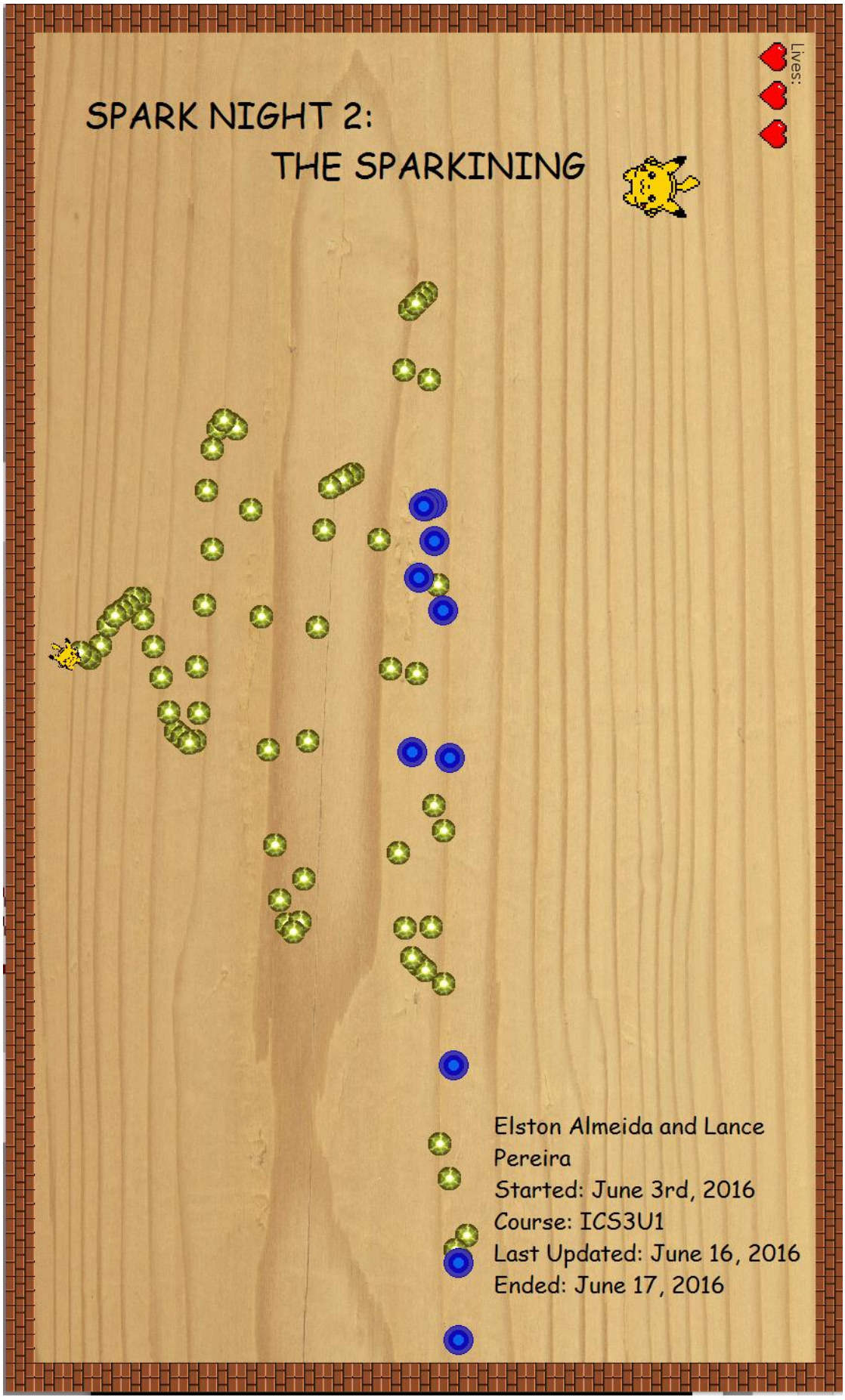

SPARK NIGHT 2:
THE SPARKINING
Lives:
Elston Almeida and Lance Pereira
Started: June 3rd, 2016
Course: ICS3U1
Last Updated: June 16, 2016
Ended: June 17, 2016

z

2. Customer Requirements:

<u>Client: Mr. Reid (Boss)</u>

- Client requires we use pygame library to develop game
- Spark theme

ICS3U – Python Final Project Rubric

Design (10 Think)

_ Written explanation of design provided (Phase 1)
_ Correct use of UMLs or Stepwise documenting high level design
_ Correct use of Flowcharts for low level design
_ Flowcharts for all complex methods
_ Designed use of Variables
_ Designed use of Selection
_ Designed use of Repetition
_ Designed use of Procedural paradigm
_ Designed use of Components (not just data holders)
_ Creative design which has unique characteristics

Documentation (10 Comm)

_ Author's Name(s)
_ Date/Purpose/Course in all files
_ Headers for each subprogram/class
_ Proper comments all selection statements
_ Proper comments explaining looping statements
_ Use of blank lines to separate important sections of code
_ Proper indentation throughout
_ Completed project report (all sections).
_ Project report is well written and easy to follow
_ Project Management tools use effectively to plan and collaborate

Implementation (10 Know)

_ Implements all aspects of the design
_ Free from syntax errors (It runs)
_ Good use of variables (including names)
_ Good use of sequence
_ Good use of selection
_ Good use of repetition
_ Good use of subprograms (including names)
_ Good use of classes and objects
_ Good use of lists
_ Efficient designs (no useless/repetitive coding)
_

Testing & Deployment (10 App)

_ Whole program works without error (any!!)
_ Problem accomplishes designed task
_ Testing section in report describing how code was tested
_ Testing show sufficient range of test cases
_ Team allowed enough time to test in class (post implementation)
_ All file(s) are appropriately named
_ Files and documents are organized and in appropriate folders
_ Software was portable (able to move and recompile without changes)
_ Design documents updated indicating what was changed
_ Task list included of who was responsible for what

Marking

Each project will receive a project mark based on the above checklist. The level will be calculated as follows:

Category	Level 0	Level 1	Level 2	Level 3	Level 4-	Level 4	Level 4+
Communication	Less than 5	5 checks	6 checks	7 checks	8 checks	9 checks	10 Checks
Thinking	Less than 5	5 checks	6 checks	7 checks	8 checks	9 checks	10 Checks
Knowledge	Less than 5	5 checks	6 checks	7 checks	8 checks	9 checks	10 Checks
Application	Less than 5	5 checks	6 checks	7 checks	8 checks	9 checks	10 Checks

Adjustments may be made based each individuals persons mark based on their contributions to the project (both stated and observed by the teacher). For examples marks may be deducted for days wasted not working on your project. It is expected that this project represents your own work. Any assistance gained from an outside source should be clearly documented. Work that is not believed to have been produced by the student will be given a mark of zero. Oral questioning may take place if concerns arise as to the authorship of the work handed in but it not required. Use of objects and components not learned in this class is allowed but will almost certainly lead to inspection of the project unless brought to the attention of the teacher earlier. In short, make sure you understand every aspect of your own project.

3. Project Timeline:

Projected Timeline

Monday 6	Finish design for initial proposal
Tuesday 7	Implementation: Player models and movement mechanics
Wednesday 8	Implementation: Player attacks and enemies (attacks and collisions)
Thursday 9	Implementation: Map and Puzzle (Design and Model)
Friday 10	Implementation: Unit testing for important functions and bug fixing.
Monday 13 - Friday 18	Bug fixing and finishing up any problems, adding optional features to game.

Actual Timeline

Project Date	Task Done
3rd-June	Created Github repository, thought and wrote down level design with character models and movement implementation, research into movement and other pygame modules
4th-June	Implementation of mouse to face player. Got other ideas set, further experimentation with pygame.
5th-June	Mouse movement towards is thought out and worked on.
6th-June	Further work on smooth mouse movement.
7th-June	Worked on bugs with movement, and implemented pygame collisions, started work on level builder.
8th-June	Modified pygame collisions for a less buggy experience, adding more features to level design.
9th-June	Started work on attacks, made more levels.
10th-June	Implementing attacks with movement, and collisions.
11th-June	Work for better movement and collisions, and making more attacks.
12th-June	Simplifying logic, and work on attacks.
13th-June	Implementing proper images, cleaning up code.
14th-June	Implementing enemies, and adding sprites with collisions together
15th-June	Unit testing, finishing documentation, cleaning up code
16th-June	Testing final project, marking up any unfinished work

4. Design Proposal:

Initial Proposal: (Spark Night)

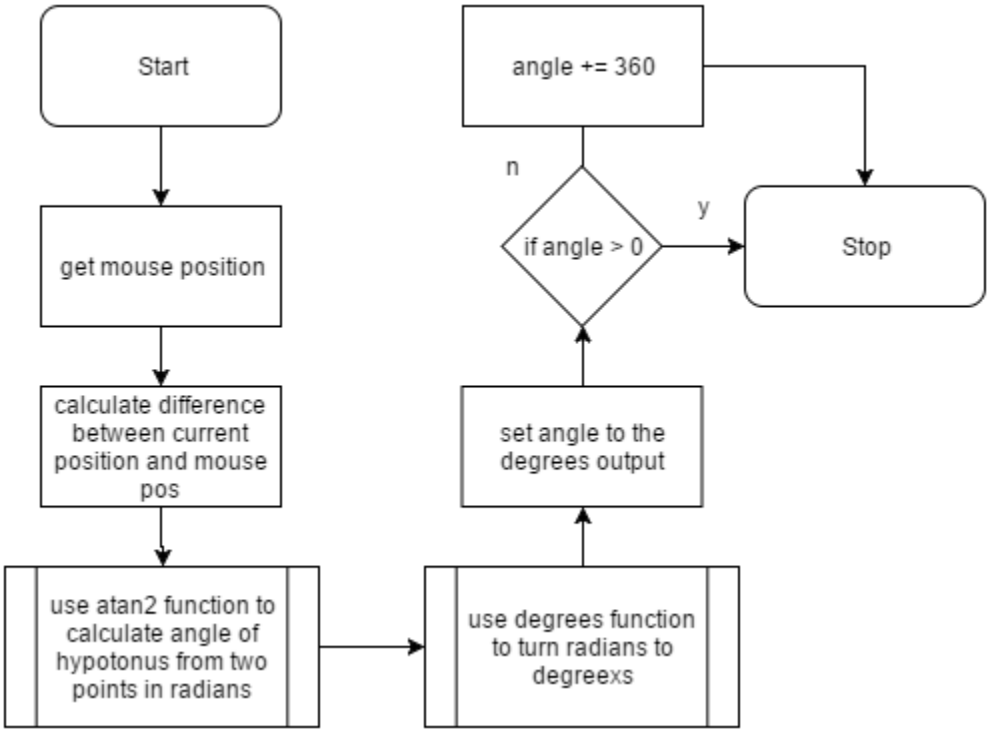
Our initial proposal consists of an arcade rpg adventure game, set in the future. The game would have a top down view similar to classic games such as Zelda, or Don't Starve. We addressed the theme of spark into the game by including lightning moves for the character. The lightning attacks are activated but the four keyboard keys Q,W,E,R, with Q being a standard lightning bullet, followed by W, and area of effect lightning attack, E a lightning shield, and R, a continuous lightning bolt that stuns the player while firing. The player in this game would always be facing the mouse and move by mouse clicks, right mouse clicks. The attacks would be projected to wherever the mouse is for the attacks that are not a shield or area of effect. We plan on implementing 2 types of monsters for the version of the game that we are submitting, 1 water type and 1 fire type, their elements determine how effective the attacks are against the enemies and how difficult the level is. We plan on also adding a boss (who we are hoping to add some AI to, so that they are fun to fight). The normal monsters will have predictable movements. We plan to create 3 maps in the version that we release, (but plan on making more if we have extra time). These levels will consist of the first spawn level, which will have tutorials, the second level with (possibly a puzzle and) some basic enemies, and lastly the final level, for now, which will have the hard enemies and the final boss at the end. (Lastly if we have time we plan on implementing a skill tree.)

Actual Proposal: (Spark Night 2: The Sparking)

We planned on creating a top down arcade rpg game with the main character being an electric welding hero. We started by importing the pygame libraries to build the game from. We then designed the player, the player had a lot of attributes so we decided to make it an object with the Player class which inherited from the sprite class:

Player Class	
Attributes:	
width	int
height	int
imageMaster	Sprite object
image	Sprite object
rect	tuple
rect.x	int
rect.y	int
angle	int
mouseMovePos	int
movedy	int
movedx	int
xvelocity	int
yvelocity	int
remainderxvelocity	int
remainderyvelocity	int
moveTimer	int
remainderMoveTimer	int
tracexvelocity	int
traceyvelocity	int
traceremainderxvelocity	int
traceremainderyvelocity	int
moveFactor	int
map_number	int
wallCollision	boolean
obstacle	list of ints
imageMasterSprite	Sprite object
imageSprite	Sprite object
Methods:	
__init__(PlayerWidth,PlayerHeight)	(int,int) -> None
get_pos()	None
move()	None
changeVelocityAfterCollision()	None
moveUpdate()	None
attack_Q()	None
attack_W()	None
update()	None

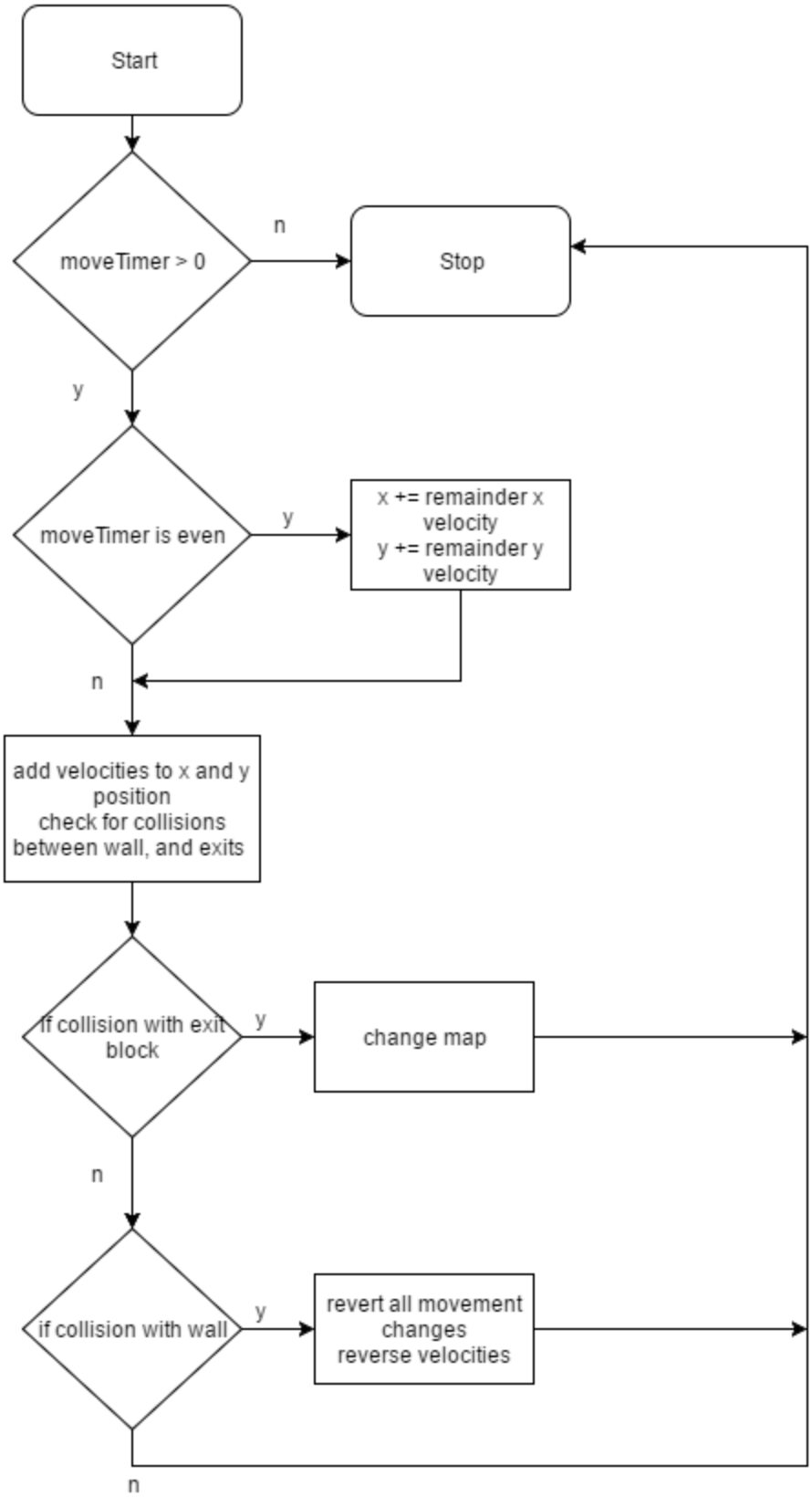
The complex functions in Player class included get_pos(), move(),moveUpdate(), attack_Q, and attack_W. The get_pos() function would be used to find the position of the mouse and the position of the player, it would then use these points to calculate the angle at which the mouse is facing at and set the angle variable to how much the object needs to rotate to face the mouse.



The move() function is designed to be called each time the user clicks the right side of the mouse, when it is called, it determines the line that it needs to take to reach the mouse and divides it by the preassigned moveFactor, so that after moveFactor amount of loops of the main function, the player is at the mouse click.

```
def move(self):
    '''
    Updates the velocities of the player after detecting a mouse click
    '''
    #determines how many increment to move the object by, say the
    #difference in x was 80, this would divide that by say 40 and get 2,
    # so each update would add 2 to posx
    self.wallCollision = pygame.sprite.spritecollide(self, self.obstacle, False)

    #Gets position of mouse and finds difference in x and y cords of
    #both points
    self.mouseMovePos = pygame.mouse.get_pos()
    self.movedx = self.mouseMovePos[0] - self.rect.center[0]
    self.movedy = self.mouseMovePos[1] - self.rect.center[1]

    #Divides difference of points by factor that determines how fast the
    #character moves
    self.xvelocity = self.movedx / self.moveFactor
    self.yvelocity = self.movedy / self.moveFactor

    #Since pygame is not perfect, when dividing, there are remainders
    #that are left, and these values store them so they can be
    # added in between big velocity movements
    self.remainderxvelocity = (self.movedx % self.moveFactor) /\
    (self.moveFactor / self.remainderMoveTimer)
    self.remainderyvelocity = (self.movedy % self.moveFactor) /\
     (self.moveFactor / self.remainderMoveTimer)

    #this variable basically tells the main loop, how many times to
    #update player pos before it reaches destination
    self.moveTimer = self.moveFactor
```

The move update function is called to increment the position of the object by the velocity assigned in the move function


Start
moveTimer > 0
n
Stop
y
moveTimer is even
y
x += remainder x velocity
y += remainder y velocity
n
add velocities to x and y position
check for collisions between wall, and exits
if collision with exit block
y
change map
n
if collision with wall
y
revert all movement changes
reverse velocities
n

The Q and W attack function create object in child classes of the player called Electricity Ball, and Field Effect for the W function. Both classes inherited from the Player class and did not need to have any more functions, just needed slight additions to pre existing functions mentioned before.

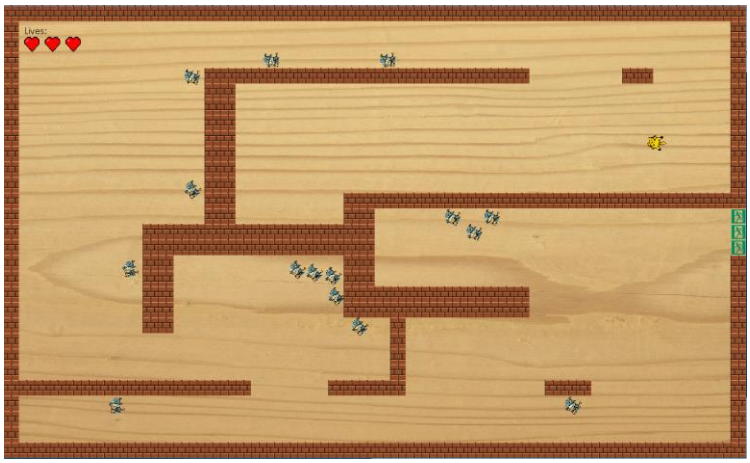
ElectricBall Class	
Attributes:	
width	int
height	int
imageMaster	Sprite object
image	Sprite object
rect	tuple
rect.x	int
rect.y	int
angle	int
mouseMovePos	int
movedy	int
movedx	int
xvelocity	int
yvelocity	int
remainderxvelocity	int
remainderyvelocity	int
moveTimer	int
remainderMoveTimer	int
tracexvelocity	int
traceyvelocity	int
traceremainderxvelocity	int
traceremainderyvelocity	int
moveFactor	int
map_number	int
wallCollision	boolean
obstacle	list of ints
isBoss	boolean
exploded	boolean
orbImage	image
orbExplosionImage	image
Methods:	
__init__(PlayerWidth,PlayerHeight)	(int,int) -> None
move()	None
changeVelocityAfterCollision()	None
moveUpdate()	None
update()	None

After the Player object was created along with its children classes, it's attacks, the objective moved towards the classes that would make the levels and the environment in them. The Level class

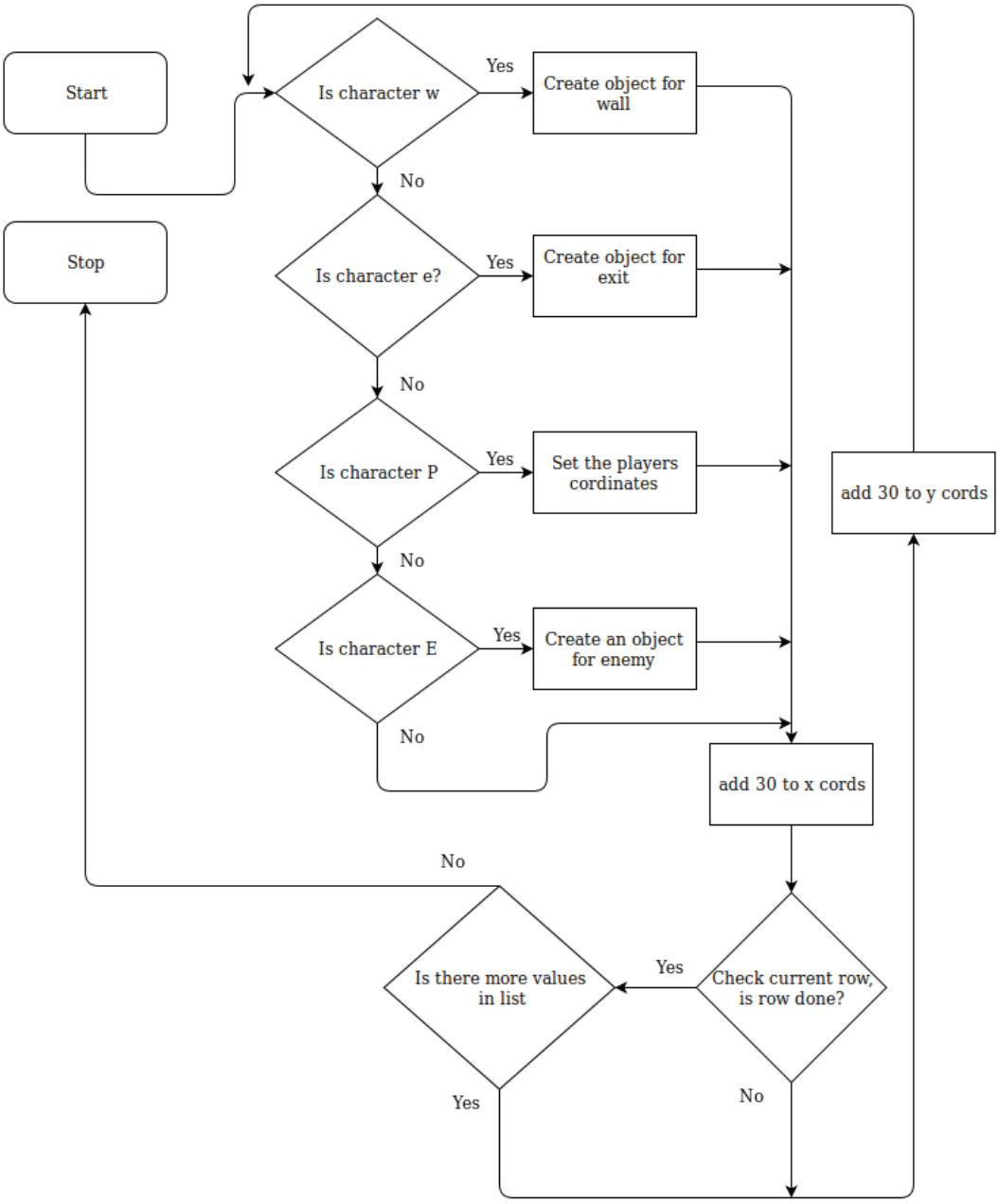
Level Class	
Attributes:	
level	int
x	int
y	int
exec_var	string
Methods:	
__init__(PlayerWidth,PlayerHeight)	(int) -> None
make_level()	None

, were designed to meet these specifications. The Level class was designed with the purpose of switching between the 5 levels stored in the map_list file, when called upon. The only unique function of the Level class was it's "make_level()" method, which read a string, shaped into a matrix. Each letter in the string :

```
            ]
        self.map2 = [

"wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww",
"w                                                        w",
"w           E        E                                  w",
"w   E                                                    w",
"w            wwwwwwwwwwwwwwwwwwwwww      ww             w",
"w            ww                                        w",
"w            ww                                        w",
"w            ww                                        w",
"w            ww                              P         w",
"w            ww                                        w",
"w            ww                                        w",
"w            ww                                        w",
"w    E       ww       wwwwwwwwwwwwwwwwwwwwwwwwwwwwwww",
"w            ww       ww                               e",
"w        wwwwwwwwwwwwwwww  E  E                         e",
"w        wwwwwwwwwwwwwwww    E                          e",
"w        ww           ww                               w",
"w    E   ww           ww                               w",
"w        ww   E E E   wwwwwwwwwwwww                    w",
"w        ww           wwwwwwwwwwwww                    w",
"w        ww              w                             w",
"w                    E   w                             w",
"w            E           w                             w",
"w                        w                             w",
"wwwwwwwwwwwwwwwww      wwwww         www               w",
"w                                                        w",
"w    E                            E                     w",
"w                                                        w",
"wwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwwww",
            ]

        self.map3 = [
```



was parsed through by the function and analyzed by the selection to see wether to leave it empty or to create an object and draw it at the position analyzed.

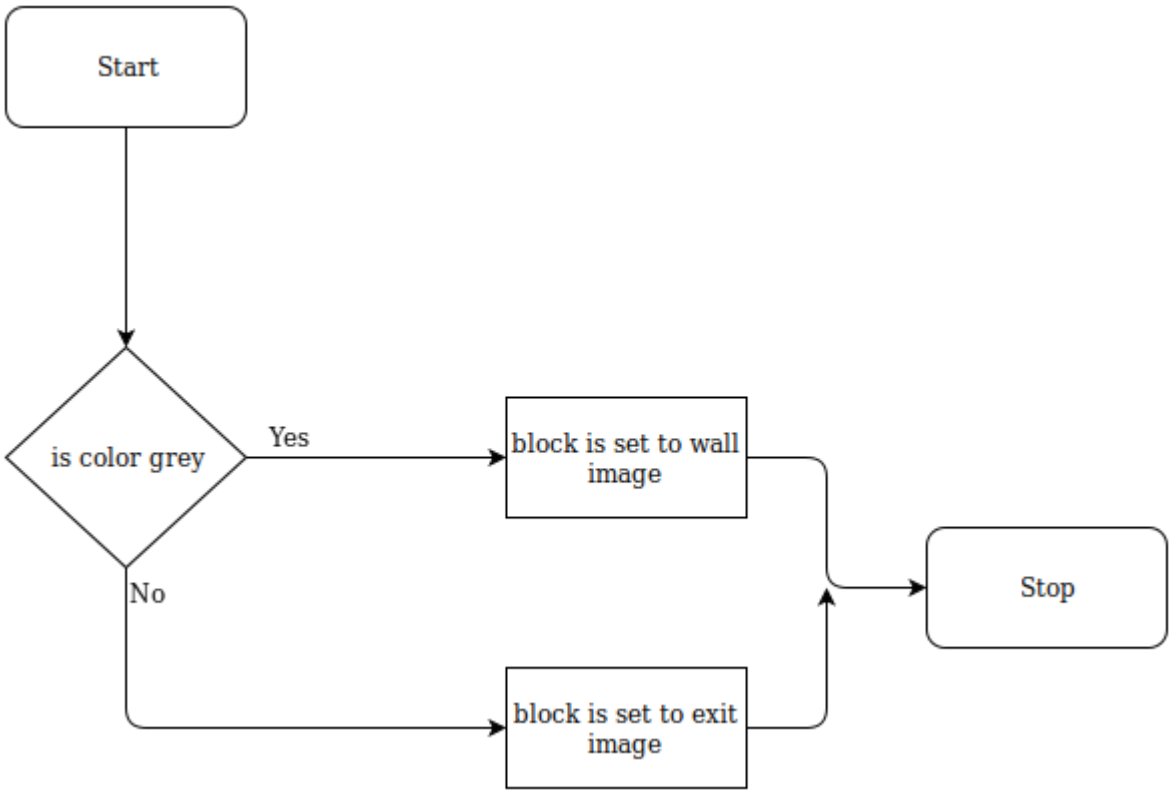


Flowchart for make_level()

The objects that were drawn were members of the Wall and Enemy class. “W”’s in the string represented wall objects to create, “E”’s indicated enemies, the “p” was the player spawn point, and “e” were special wall objects called exits that called the change map function of the Level class. The wall class as mentioned before created an object that’s sole purpose is to be a barrier to stop the player from exiting the map

Wall Class
Attributes
X, int Y, int Rect.x, int Rect.y int Color, tuple
Behaviours
checkBlockSprite()

The only complex function in this class was the “checkBlockSprite()” method, which checked to see



if the wall created was a normal block or an exit block that allows the player to update the map.

After the Wall class, we designed the enemy class. The enemy class would inherit from the Player class and have everything that the Player class has except the attack functions, and carries a modified move function. The move function is modified so that it tracks the player rather than the mouse and is constantly moving towards the player.

Enemy Class	
Attributes:	
width	int
height	int
imageMaster	Sprite object
image	Sprite object
rect	tuple
rect.x	int
rect.y	int
angle	int
mouseMovePos	int
movedy	int
movedx	int
xvelocity	int
yvelocity	int
remainderxvelocity	int
remainderyvelocity	int
moveTimer	int
remainderMoveTimer	int
tracexvelocity	int
traceyvelocity	int
traceremainderxvelocity	int
traceremainderyvelocity	int
moveFactor	int
map_number	int
wallCollision	boolean
obstacle	list of ints
isBoss	boolean
imageMasterSprite	image
imageSprite	image
Methods:	
__init__(PlayerWidth,PlayerHeight)	(int,int) -> None
move()	None
changeVelocityAfterCollision()	None
moveUpdate()	None
update()	None

After all the main features were designed we decided to design class that would handle drawing all the extra features to the game that made it more comfortable to use for the player such as a pause

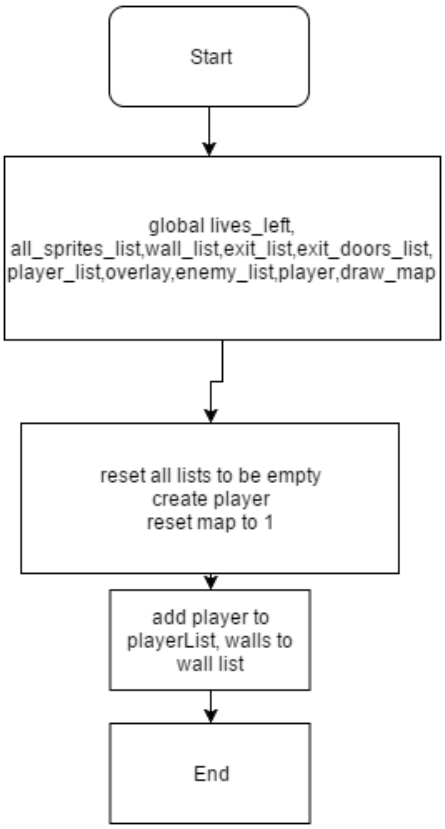
menu, mute function for the music, mouse button toggle, and a keyboard shortcut to quit. Along with these functions this class would also be in charge of displaying the characters lives.

Overlay Class
Attributes
Font types (strings, int) isPaused (Boolean) image (tuple) Colorkey (tuple)
Behaviours
update() main_menu() blurSurf(surface, amount) returns surface

The only complex function in the Overlay class was the bur Surf function which blurred the screen while the game was paused

After all the classes were designed, we designed the change_map function, which as it's name implies, changes the map and clears the global variables of the map, wall, enemy, and exit list.

Lastly we designed the restart function for the game, which had the sole purpose of resetting all the global variables in the file to the point they were when the file was first launched, basically recalling the main function.



Lastly we designed the 'main function' that we decided not to put in a function due to the large amount of global variables being passed through it.

Main Loop flow chart: http://i.imgur.com/oPA3ro2.png (too big)

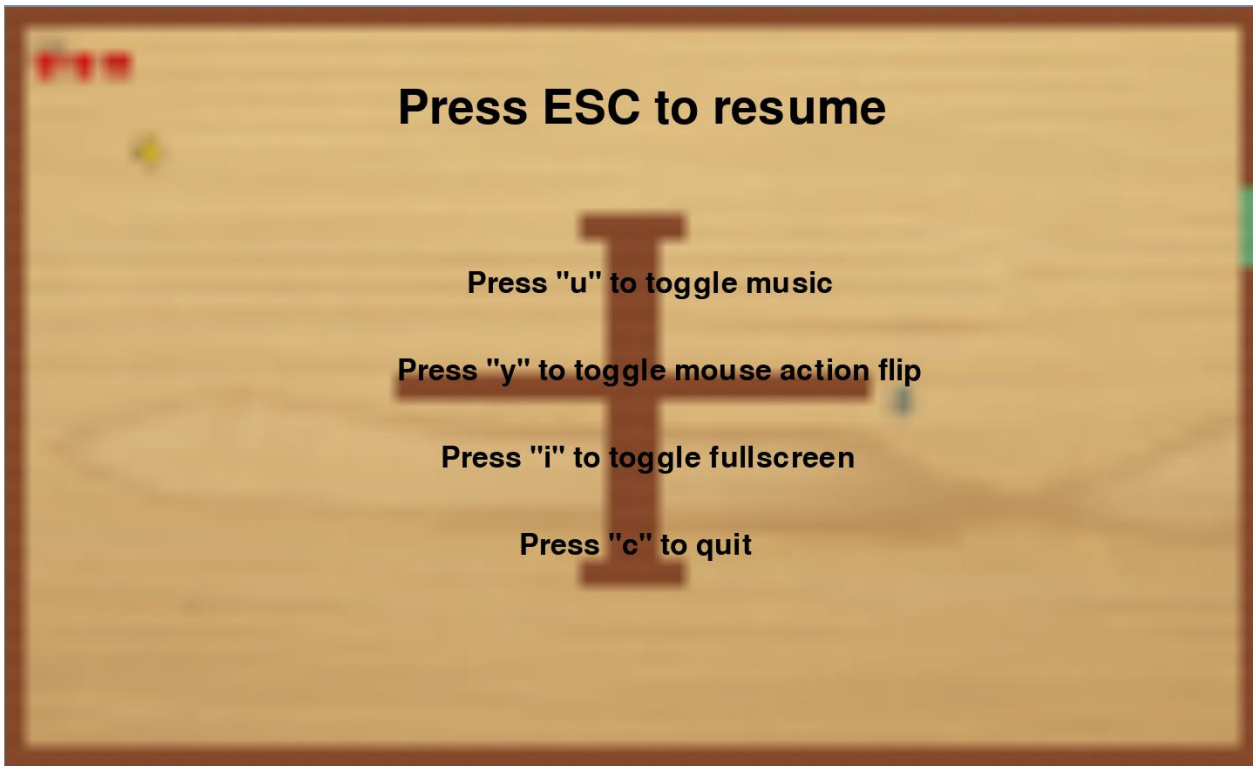
Menu loop: http://i.imgur.com/ZuIgxcw.png (too big)

Changes to Design as Implemented:

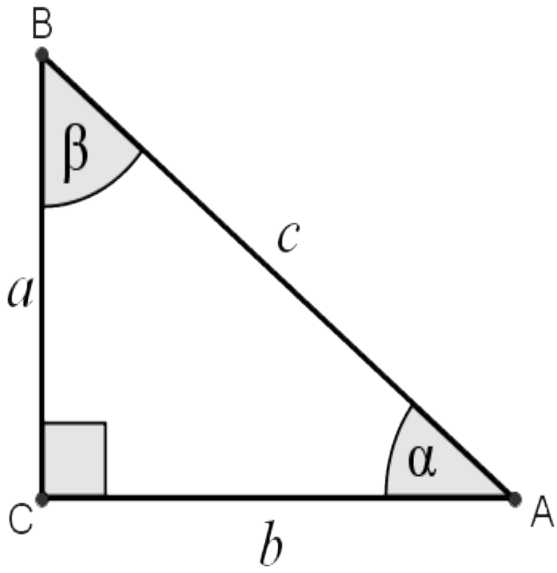
- Decided not to do ultimate attack or E attack
- Decided not to implement boss
- Everything else designed was implemented

5. Implementation Details and Deliverables

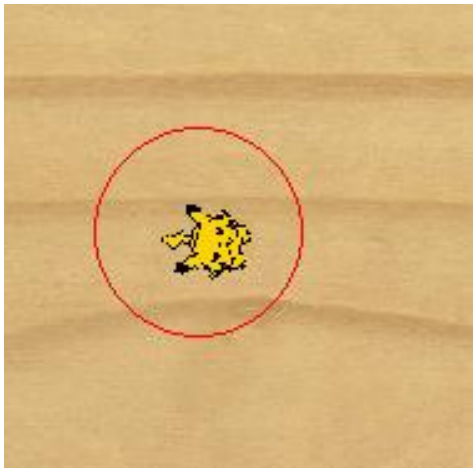
The game introduces your character as a pikachu and wizards who are the evil people, the controls include a “click to move” with your left click , the player is stacked with 3 lives and each time the player comes in contact with any of the other witches, the player will lose a life. When a player looses all three lives, the game resets back to the original state. The player has two different attacks to counter the wizards, first being the field of effect or “mag field” attack and the second is an orb of electricity, this helps the game incorporate some form of sparks. The pause menu gives you the ability to stop the game in a live session and be able to change some settings suiting your need, such as changing the mouse click to move button, toggle music, and fullscreen or normal modes. (To the right is the enemy sprite and the orb of electricity used in the game and the bottom image shows the pause menu)



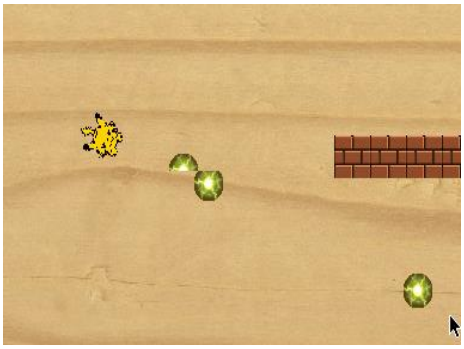
To move to the position of the mouse, we used the inbuilt pygame function to get the coordinates of the mouse in a tuple, this tuple is compared to the player's current coordinates to get the x and the y difference from the mouse to the player, additionally this was able to help us attain the angle of the player relative to the mouse. The difference can create an imaginary triangle where a clean path straight to the mouse can be drawn via using the hypotenuse of imaginary triangle. Then the player was moved over time towards the mouse by lowering the difference in the by getting a velocity from the hypotenuse, to get a smooth effect. The collisions were implemented with the built in pygame sprite checking collisions with groups (pygame.spritecollide) and our own personal method of checking for any collisions which would put a sprite in the next location and check if it collides with anything, if not the sprite would continue to move, however if it did, the velocities would be inverted and lowered accordingly. (right angle triangle to the right for reference)



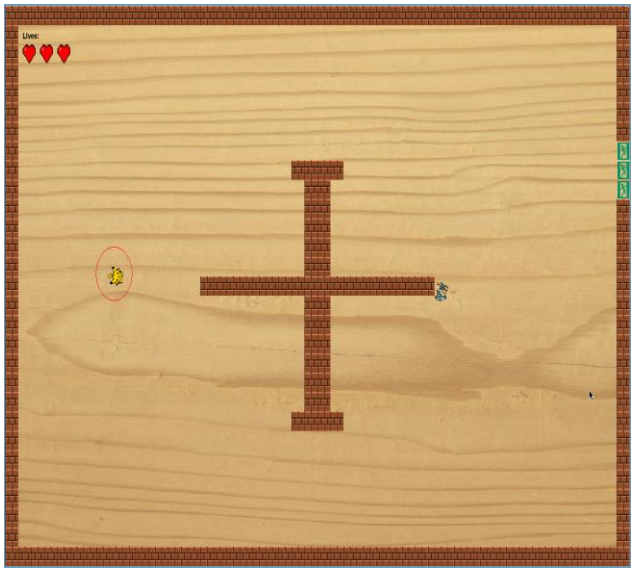
In the field of effect, the main goal is to create a visualized magnetic field around the player allowing them to entertain themselves with a close quarters combat. This was accomplished by drawing a circle around the middle of the player's coordinates and players center coordinates. This would then increase the incrementer, as the incrementer increased the radius of the circle is increased, and so does the circles collision box. This helps to provide accurate hitbox. This sprite is added to the list of attacks. (To the right is the demonstration of the attack)



The electric orb attack works by first taking inheritance from the player class, this allows it to get the mouse position and have the sprite move towards the mouse, the velocity however does not change, it does not rotate, and it is made to disappear when it collides with an object. In addition to being added to all_sprites_list, it is added to the attack list. ((screen tearing), demonstration of electric orb on right)



The enemy also uses inheritance of the player to get position of the player and gives it to the ability to constantly track the mouse. This is done by the move class, the class checks if the selected path is able to move by checking collisions with walls as it checks the wall sprites list before hand, similar to the players movement, except it has been modified to make sure that if it does come in collision in the wall, the sprite will bounce, this allows the sprite to constantly move. The boss can also be classified as an enemy too as they are all modified players. The boss has been modified to constantly shoot at the player, and track the player of the difference of x and y coordinates and the using the difference with the atan function to get an angle again.



All the sprites are split up into their own pygame sprite groups that allow us to check for collisions and call specific lists to get items to manipulate inside. All the blocks created in the level class will get saved in their own private list only for walls. The exit and the normal walls are no different, except one is set to a different color which would display a different sprite image and set in the exit list and the normal walls share a common list with the exit blocks.

Testing:

In unit testing we used code traces and debugged using print statements, and 2 tests cases to see if the velocity of the players was correct. The test cases for test move and test change velocity checks to see if the x and y position of the player equals 145 after updating it with the velocities provided. The 2nd test case determines to see if the velocities are returned to 1 after a collision which this test case simulates. Both tests were ran after the last update and returned 'OK' values indicating that our two functions were working properly. The reason only 2 were done, was because these were the two functions that were buggy during implementation and needed to be tracked as we developed more code.

```
C:\Users\Pereira Family\Documents\GitHub\cpt [attack +2 ~4 -1 !]> .\testmodules.py
..
----------------------------------------------------------------------
Ran 2 tests in 0.001s

OK
C:\Users\Pereira Family\Documents\GitHub\cpt [attack +2 ~4 -1 !]>
```

```
import unittest
from main import *

class Testmain(unittest.TestCase):
    pygame.init()
    screen = pygame.display.set_mode([width, height],pygame.FULLSCREEN)
    pygame.display.set_caption("Sparknight 2: The Sparkening")
    player = Player(30,30)
    player.xvelocity = 1
    player.vyelocity = -1
    player.remainderxvelocity = 1
    player.remainderyvelocity = -1

    player.rect.x = 1
    player.rect.y = 1

    def test_move(self):
        player.move()
        self.assertEqual([player.rect.x,player.rect.y],[145,145])

    def test_changeVelocityAfterCollision(self):
        player.xvelocity = 5
        player.vyelocity = -10
        player.remainderxvelocity = 4
        player.remainderyvelocity = -3
        player.changeVelocityAfterCollision()
        self.assertEqual(player.xvelocity,-1)
        self.assertEqual(player.vyelocity,1)
        self.assertEqual(player.remainderxvelocity,-1)
        self.assertEqual(player.remainderyvelocity,1)

if __name__ == '__main__':
    unittest.main()
```

6. Maintenance

The final version of our game was, okay. It had many features that we and our game testers found enjoyable and fun but also had some features that we were deeply ashamed we could not finish. The features that we were most proud of was the player's movement which although it sounds easy was one of the hardest things to implement due to the "interesting" way that pygame moved sprites by decimal numbers. It involved us coming up with a new velocity values that would be added every other move call. This was truly impressive for us as we were the only group that used a mouse click to move, which made the game a bit more complex to work around, but made it much more satisfying when completed.

One of the things in our project that we were disappointed with and wished we had more time with was the collisions. The collisions, although functional, were not to our standards as they would sometimes cause the player to get temporarily stuck unless they clicked in the opposite direction of the wall. The collisions were by far the hardest part of the entire project due to the sheer amount of possible scenarios the player could face. In older versions of the game we had the Q attack bounce of the walls to make it more fun, but this added a new level of complexity to the collision functions which usually caused the orb to glitch through the walls. This forced us to limit the Q orb to just have an inverted movement after collision, which made the game a bit more boring than we would have like if we had the bouncy balls.

The last thing we felt could have been a lot better was the implementation of enemy AI. The enemies in our game were supposed to follow the player around and bounce of the walls in a modified version of the orb attack . Due to the failure of our collisions function at detecting which side of the sprite collided caused us to reduce the mobility of the enemies to go to the closest wall to the player. Furthermore due to the restrictions of the collisions , the enemies would get stuck in walls causing the enemies to become nothing more than suicide bombers at best and sitting ducks at worst.

Overall the greatest flaw in our project was that we did not allocate enough time for collision implementation. This problem was supplemented by the fact that we used the mouse to move and rotated the player, causing the player's get rect to change size (pygame flaw, not user created). Although the movement impaired the other features of our game, it added an amazing feature that we were both proud of.

7. Resource Allocations

Lance	Player movement, Enemy class, Electric orb attack, Half the flow charts and UML's, collisions for player and orbs, Attempted Boss, And restart function, Maintenance, Design write up, unit tests.
Elston	Levels, level builder, field of effect, main menu, UML's for overlay wall class , Implementation Details and Deliverables, time lines, change map function, flow charts for all aforementioned functions and classes and main loop, unit tests.

Refrences:

blurSurf function
http://www.akeric.com/blog/?p=720

level making and other functions
http://pygame.org

Wizard sprite
http://vignette3.wikia.nocookie.net/scribblenauts/images/f/fc/Wizard_Male.png/revision/latest?cb=20130215182314

Player sprite
http://piq.codeus.net/static/media/userpics/piq_244306_100x100.png

Walls:
https://www.google.ca/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&ved=0ahUKEwit1djG_a7NAhXK7IMKHd7nB8cQjBwIBA&url=http%3A%2F%2Fgallery.yopriceville.com%2Fvar%2Falbums%2FBackgrounds%2FBrick_Wall_Background.png%3Fm%3D1399676400&bvm=bv.124272578,d.cGc&psig=AFQjCNEv6rcMOPIM3FlidtOS9zQfEVi29g&ust=1466250134775093

Exits:
http://www.pngall.com/exit-png

Music:
Merry Christmas Mr.Lawrence - Ryuichi Sakamoto

Player Attack Q sprite:
http://s1087.photobucket.com/user/lmsvv/media/lightningball.png.html