Phase III

Reilly, Terrence Justice, James Goodall, Brad
Maresh, Keefer Gleason, Meagon

February 20, 2017

Contents

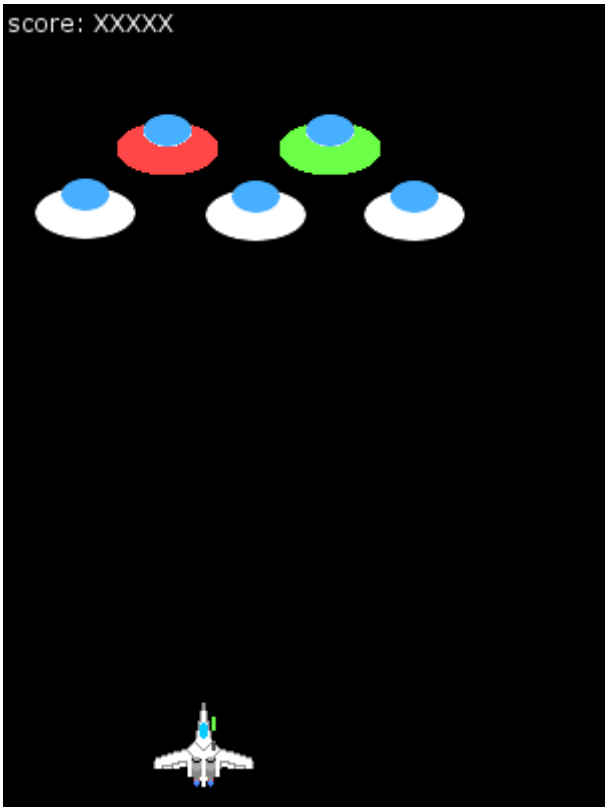
External Design Specifications

Menu

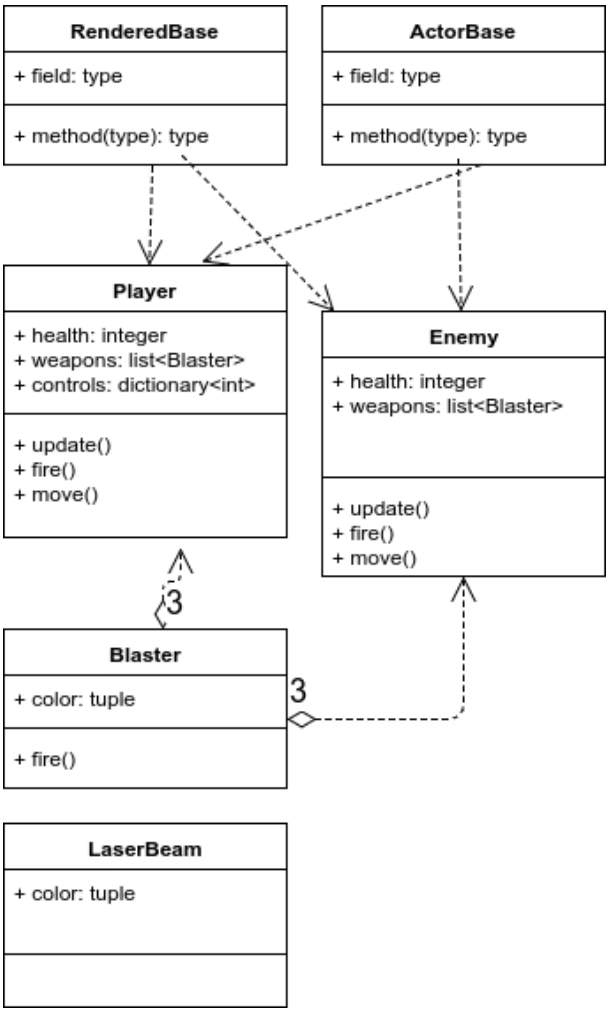
- Four buttons (Start game, Options, High Scores, Exit Game)
- Simple interface
- Potentially game title at top

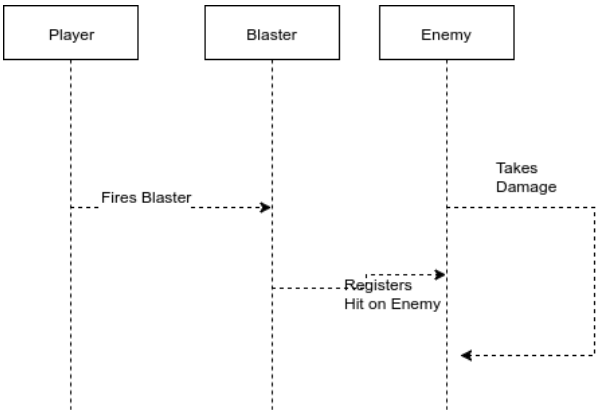
Game

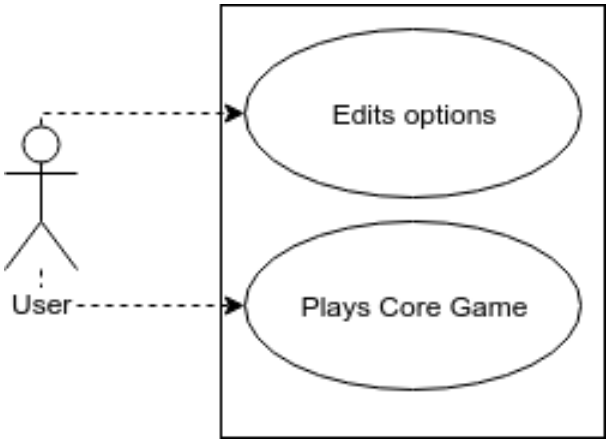
- Player sprite
  - Moves left and right, potentially up and down
  - Fires laser weapon (green pictured)
- Enemy sprites
  - Move down screen, left and right
  - White enemy vulnerable to all player weapons
  - Colored enemies only vulnerable to matching weapon blast
- Score
- Lives remaining (either number-based or icon-based) on top right (0 remaining in picture.


score: XXXXX

Software Architectural Design Specifications







Detailed Design Specifications

Interface Specifications

The classes `RenderedBase` and `ActorBase` will form a basis for the API for LazerBlast. They describe all moving objects, and all actors in the game. Among their most important functions, they will define a method to set the current action. They will also use Python's magic methods to get the next frame of their actions.

The class `MainMenu` is a basis for the User Interfaces for LazerBlast. Through it, the user can navigate through and access different parts of the game, including high scores and control options.

The GameSound class will release a sound when it receives a command telling it that a missile has been fired by the player or an enemy ship. The sound will then be transmitted through the game for the player to hear. The ShipProjectile class will contain a method which will create the missile object, another method which will assign certain characteristics to the missile object such as whether its a enemy missile or a player missile, and if a player missile then the object will be assigned a color based on a command received from the player telling the program which color the missile should be. The object will then be returned to the class which had called the ShipProjectile class in order to retrieve a missile object.

Class Definitions

- `RenderedBase` will implement the basic interface for any object which is rendered on the screen. It provides a way of getting the next image in an sequence for rendering.
- `ActorBase` will describe the interface for actors. All objects which are rendered to the screen and perform some action (the user and the enemies) will be subclasses of this class.
- `Player` will extend the ActorBase to represent the user's spaceship. The Player will check for the user's inputs when it is updated each frame.
- `Enemy` will extend the ActorBase to represent the enemy spaceships. As such, it will move on its own when updated each frame.

- `Blaster` will specify the behavior of a laser of a particular color.
- `LaserBeam` will give a graphic representation of a laser being fired.
- `GameSound` class will implement the sound aspects of the game which will provide the missile objects with sound every time they are fired and when they have made contact with the target ship.
- `ShipProjectile` class will create a new missile object when called and assign certain characteristics to that object based on information received. Once the object has been created, it will then be returned to the class which had called for it to be created.
- `MainMenu` will allow the user to access the main game and to view information such as the high scores across sessions through options on a menu, such as New Game and High Scores. Also allows the user to customize their controls through an options menu.

Pseudocode

```
# Terrence's section
class RenderedBase(object):
    images = dict()
    sounds = dict()
    _action = None
    _action_i = -1
    box = pygame.Rect(0, 0, 0, 0)

    def set_action(self, action):
        if action not in self.images:
            raise Exception
        self._action_i = 0
        self._action = action

    def __next__(self):
        if self._action_i == -1:
            raise Exception
        self._action_i += 1
        return self.images[self._action][self._action_i - 1]

    def render(self, context):
        pygame.draw.rect(context, (255, 0, 0), self.box)


class ActorBase(object):
    def __init__(self, health=0, weapons=list()):
        self.health = health
        self.weapons = weapons
        self._weapon_i = -1 if len(self.weapons) == 0 else 0
```

```
    def add_weapon(self, weapon):
        if self._weapon_i == -1:
            self._weapon_i = 0
        self.weapons.append(weapon)

    @property
    def weapon(self):
        if not (0 <= self._weapon_i < len(self.weapons)):
            raise Exception
        return self.weapons[self._weapon_i]

    def next_weapon(self):
        self._weapon_i = (self._weapon_i + 1) % len(self.weapons)
```

```
import pygame
from base_classes import RenderedBase, ActorBase

class Player(ActorBase, RenderedBase):
    """ Player's ship """
    def __init__(self, controls=dict(), health=0, weapons=list()):
        ActorBase.__init__(health=health, weapons=weapons)
        self.controls = controls

    def update(self):
        pass # Checks for user inputs and moves or fires accordingly.

    def fire(self):
        pass # Fires the active weapon

    def move(self):
        pass # Changes position on screen.

class Enemy(ActorBase, RenderedBase):
    """ Enemy ship that combats player. """
    def __init__(self, health=0, weapons=list()):
        ActorBase.__init__(health=health, weapons=weapons)

    def update(self):
        pass # Moves downward and randomly either left or right.

    def fire(self):
        pass # Fires active weapon

    def move(self):
        pass # Changes position on screen.


class LaserBlaster(object):
    """ A weapon for ships to use. """
```

```
    def __init__(self, color=(0,0,0), charge=0, damage=0):
        self.color = color
        self.charge = charge
        self.damage = damage

class LaserBeam(RenderedBase):
    """ Graphic representation of a fired laserbeam. """
    def __init__(self, color=(0,0,0)):
        self.color = color

#Keefer's section
class MainMenu():
    def __init__(self, cursor=0):
        #set the menu type (main/high score/options)

        self.setup_graphic()
        self.setup_cursor()

    def setup_graphic(self):
        #draw title graphic/high score table

    def setup_cursor(self):
        self.cursor = cursor
        #draw cursor image

    def update_cursor(self, keys):
        if self.cursor.state == 0:
            #Point to option one (New Game/Set buttons/Return)
            if key down:
                self.cursor.state == 1
            elif key return:
                #Initialize new game/set keys/return to previous
        elif self.cursor.state == 1:
            #Point to option two (High Scores/return)
            if key down:
                self.cursor.state == 2
            elif key up:
                self.cursor.state == 0
            elif key return:
                #Initialize options menu
        elif self.cursor.state == 2:
            #Point to option three (Exit)
            if key down:
                self.cursor.state == 3
            elif key up:
                self.cursor.state == 2
            elif key return:
                #open options menu
        elif self.cursor.state == 3:
```

```
            #Point to option three (Exit)
            if key up:
                self.cursor.state == 2
            elif key return:
                #close the application
# Meagon's Section
class GameSound(string):
    sounds = dict()

    if string == 'fire':
        sounds = dict['fire']
    elif string == 'hit':
        sounds = dict['hit']
    else:
        error
    sounds = none
    sounds.play()


class ShipProjectile(stringC, stringT):

    Color = ''
    Type = ''
    JAMMED = error

    def __init__(self, color, type):
        self.color = color
        self.type = type

    def projectileColor(stringC):
        if stringC == 'blue':
            color = stringC
        elif stringC == 'red':
            color = stringC
        elif stringC == 'green':
            color = string
        else:
            color = 'white'  # default color, no hit points

    def projectileType(stringT):
        if stringT == 'enemy':
            Type = stringT
        elif stringT == 'player':
            Type = stringT
        else:
            Type = 'JAMMED'  # default, no fire of projectile

        missile = ShipProjectile(stringC, stringT)
```

```
        return missile
```

Data File Specifications

One data file for LazerBlast will consist of a pickled state of all high scores, and, possibly, of the current game state (if a user wants to continue a longer game.) The format itself is decided by Python, and will represent the internal python objects.

Another data file for LazerBlast will consist of input data received from the player in order to allow the program to determine when a sound file needs to be accessed and played, as well as when to create a missile object and what characteristics to assign to that newly created object.

Test Plan Specifications

In order to keep track of our tests we will be using a chart to track the test number, the test data, the test purpose, our expected results, and whether or not the test was successful. Below is a sample of the chart which we will be using to track our tests and their results.

Test No.	Test Data	Reason	Expected Results	Success	Comments
1	Enter temporary unlimited health for player	To test hit on enemy player without taking damage.	Enemy player will take hit and be terminated while colors match appropriately.	Yes	Test was able to be completed successfully.
2	Replace algorithm related to enemy ships.	Trying to make the path our enemy ships follow more efficient.	Enemy ships will follow same path using a more efficient algorithm.	Enemy ships getting stuck when program is running with new algorithm. Algorithm needs altered and further testing required.	

Tests relating to the syntax of our program shall be performed through the use of both the unit test framework provided through Python, as well as a code review process which will involve developers reviewing each others code and offering any advice on possible found issues.

Performance Tests

The performance tests will be documented within our test documentation sheet where we give the test data being used to test the performance, the reason which will specify what exactly relating to the performance is being tested, our expected results, whether or not the test was successful, and any additional comments which the team member who performed the tests feels should be brought to the developers and users attention. Some of the steps that will be taken prior to performance tests are to make sure that all of the correct software has been installed so that the game may perform to the best of its ability.

One of the main things which our performance tests will include is the speed and smoothness of our programs flow since python games created and ran within Pygame tend to have issues with their speed. A lot of the issues relating to the performance test are often related to the syntax used in developing the game. Steps to avoid these being issues will include using certain modules, libraries, or even just replacing the syntax with something which may help to scientifically improve the speed and overall performance of our game.

Stress Tests

The stress tests which will be performed in relation to our game will include increasing the number of enemy ships to see how many ships the program may be able to handle at once. This test will allow our developers to determine what a good number of enemy ships may be in order to provide the user with a smooth flowing game as well as a challenge. This stress test will also help developers in determining how well the path finding algorithm will work for the chosen number of enemy ships at any one time. Another test to be performed under the stress test phase will be the intensity of the boss enemy ship and what all factors may play in the fight between the player and the boss ship. Once again this test will allow developers to provide the players with a smooth flowing program as well as a challenging game.

Functional Tests

Once both the performance tests and stress tests have helped in determining syntax, libraries, modules, and any other relating factors to be used within the program, then functional tests shall be performed by going through the game and focusing on each phase and detail. Once our group has performed Alpha testing of the game and have made sure that the game is functioning as planned then the game shall be released to a select group of individuals for Beta testing so that we may receive feedback on any functionality issues which we missed. During both the Alpha and Beta testing processes, documentation shall take place in reporting each issue, the proposed solution, testing of the program with the solution in place and a recording of the results. Each reported issues shall be dealt with and the process of Alpha and Beta testing will be repeated. Once our group and those individuals chosen to participate in the Beta testing process are satisfied with the games functionality, we will consider all tests complete.

Appendix

		Name	Duration	Start	Finish	Predecessors	Resource Names
1		Menu Outline	3 days?	2/20/17 8:00 AM	2/22/17 5:00 PM		
2		Abstract Actors	1 day?	2/20/17 8:00 AM	2/20/17 5:00 PM		
3		Moving Actors	2 days?	2/21/17 8:00 AM	2/22/17 5:00 PM	2	
4		Actor Interactions	2 days	2/23/17 8:00 AM	2/24/17 5:00 PM	3	

19 Feb 17
F S S M T W T F S

LazerBlast_Phase_3 - page1