Basic Functionality Report for BulletHellShooter

Noah Hanson

Jamie Van Overschelde

Oliver Nigaba

John Krueger

***Abstract*— This document discusses the progress of Synergistic Solutions on their BulletHellShooter project. The document will discuss the progression of the project, success, challenges, and what is left to accomplish. The document includes screenshots of some of the code we have currently, as well as, a screenshot of in-game play.**

I. Project so Far

A. Milestones and Progress

So far, we have completed the collision detection for the game, and basic enemy movement, which includes customizable movement patterns. Both of these features were completed slightly behind schedule. We are expect to be done with the start screen and game over screen by Thursday evening, in line with our schedule. Beyond that, a large number of our tasks and milestones remain to be completed. Specifically, we have yet to complete our team milestones of having a complete game(17th of April), or having the final presentation finished (30th of April)

B. Successes

We currently have a functioning test demo that other features can be added onto as we complete them. The game engine provides a solid framework for our group to continue building our personal milestones off of. The rewrite of the movement and collision system has lent additional flexibility to our game.

C. Challenges

As a group, we have experienced multiple challenges. One challenge we continually run into is meeting. Every group member has changing schedules due to extra-curricular activities that make it difficult for everyone to meet regularly. We are addressing this challenge via “telecommuting” to meetings via discord. This has helped lend additional flexibility to when we can met. We have also run into troubles with collision detection in the game. When one enemy got hit all of the enemies were destroyed. This challenge was overcome by rewriting the whole collision detection system from scratch.

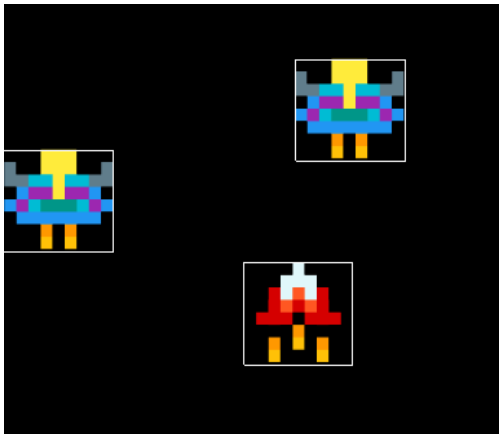
D. Still to Do

Currently, we still have to finish some code and integrating it all into one cohesive game. We also have to finish our final presentations, implement sound into the game, and finish much of the group created milestones. We had set these milestones as due after this Milestone check-in, so this does not mean we are behind.

II. Screen Shots/Images

```
@Override
public void update(float delta) {
    //enemies.get(0).update();
    for (MovementBase enemy: enemies) {
        enemy.update();
    }
    player.update();
}

@Override
public void interpolate(float alpha) {
    for (MovementBase enemy: enemies) {
        enemy.interpolate(alpha);
    }

    //enemies.get(0).interpolate(alpha);
    player.interpolate(alpha);
}

@Override
public void render(Graphics g) {
    //Sprite temp = new Sprite(texture);
    //g.drawSprite(temp);
    player.render(g);

    for (MovementBase enemy: enemies) {
        enemy.render(g);
    }
    //enemies.get(0).render(g);
    //playerShip.render(g);
}
```

- Pictured above is the core loop of the gameplay. It is split into three parts. First, is update. This method is called about every .1 seconds and updates the game state. Render draws the various sprites and GUI components on the screen. Interpolate serves as a bridge between update and render to help ensure that the game doesn’t desync do do hardware issues. Each class has its own set of these three methods to “hook” into the gameplay loop.



Identify applicable funding agency here. If none, delete this text box.

- Pictured above is a section of the screen during gameplay. The enemy in the top right moves back and forth at an angle. This behavior can be altered in the constructor to produce all kinds of serpentine behavior. The top left enemy is a basic unit which moves straight down. The player ship, in red, is controlled via the keyboard. The white rectangles are the hitboxes used for collision detection.

```
//Every enemy when they updates checks to see if they have collided with the p
if (hitbox.overlaps(MyMini2DxGame.player.GetHitBox())){
    System.out.println("The player has been hit!!!");
}
```

- Pictured above is the new and simplified collision system. Everything is abstracted to a rectangle for the sake of collision detection, and then the geometry library that comes packaged with the game engine is used to check for overlap. This simplified system has solved the plague of errors that once terrorized this fair project. All game objects implement some derivative of this system.

```
public void update(){
   //Adjusts speed to create serpantine
behavior

   //First checks it see of we should reset
steps, and reverses directions if so.
   //if it's -1, then we just pretend it
doesn't exists.
   if (currentStepsToSide == maxStepsToSide
&& maxStepsToSide != -1) {
       SetXSpeed(GetXSpeed() * -1);
       currentStepsToSide = 0;
   } else if (maxStepsToSide != -1) {
       //increments if its not zero.
       currentStepsToSide++;
   }

   if (currentStepsUpDown == maxStepsUpDown
&& maxStepsUpDown != -1) {
       SetYSpeed(GetYSpeed() * -1);
       currentStepsUpDown = 0;
   } else if (maxStepsUpDown != -1) {
       currentStepsUpDown++;
   }

   //Updates game logic, checks collisions
   super.update();
}
```

- Above is a snippet of code which enables one to use the constructor to create enemies with diverse movement patterns without the need for additional subclasses. This operates by letting the programmer control the movement logic via modifying the speed of an enemy before drawing or changing coordinates.