TITLE OF PROJECT:

AUTHORS: Michael McGregor

PROJECT ABSTRACT

1. (a) Problem and Motivation:

The initial problem was how to implement a mechanic for the player to be able to change the weapon of the ship. I wanted the game to have many different unique weapons that were broken up into three categories of weapons, kinetic, laser, and missiles. I also wanted the player to be able to decide whether or not he/she wanted a specific weapon or not.

1. (b) Background:

In old arcade games, especially old shooters, it is common to implement stat changes through items that the player can pick up. These items may include health, shields, ammo, or other changes like weapons. In the old game Raiden Fighters, different colored orbs would sometimes spawn that the player could either pick up to change weapons or upgrade their current weapon. The orbs would change colors to allow the user to choose what kind of weapon they pick up or just go off the screen and disappear.

1. (c) Solution:

For our game I decided to use the same idea that Raiden Fighters used. I implemented it a little differently than they did though, because instead of having one weapon item that changes I made three kind of weapon items. Each weapon item is an item that contains a reference to an image, what kind of item it is, it’s collision box, and a velocity to which the item move every update. GameData is where all of the information is being stored and processed at. When certain events happen in the game, GameData will use a factory method to spawn in the items. The objects that are created can be predefined or randomly generated using a random number generator. This code snippet below is how each object is made.

**items.add((Item) weaponMaker.getWeapon("LASER", 400, 200));**

When the factory method creates the new item and it is rendered the object is stored in an array list within GameData. At this point the items float around freely until they are either picked up by a player or go off screen at which they are destroyed. In order for the player to pick up the items there is a check in GameData to check for collision between the item and the player ship. If there is a collision the information is saved in the ship object then the item is destroyed.

The ship object is able to process the information based on what the object is that is picked up and changes the current ship’s weapon to the appropriate weapon. Below is a code snippet of how this is done.

**//----collision: items vs friendly ships-----**

**for (int i = 0; i < this.items.size(); i++) {**

**//Rectangle[] hit = currentShip.getShipHitBox();**

**Rectangle hit = currentShip.getShipHitBox();**

**Item item = (Item) this.items.get(i);**

**if (hit.intersects(item.getRectangle1())) {**

**if (item.getItemType() == currentShip.getWeaponState()) {**

**currentShip.setLevelState(currentShip.getLevelState());**

**} else if (item.getItemType() >= 0 && item.getItemType() <= 2) {**

**currentShip.setWeaponState(item.getItemType());**

**currentShip.setLevelState(BASE\_LEVEL);**

**}**

**synchronized (items) {**

**this.items.remove(item);**

**}**

**}**

**}**

In the code above, if a ship intersects an item it first checks to see what the ship’s current weapon is. If it is the same as the weapon item picked up then the weapon level is incremented one from what the ship’s current weapon level is. If the item is different than the current ship’s weapon, the ship’s weapon type is changed to the item’s type and the weapon level is set to the base level.

1. (d) Result:

The end product is a weapon management system that operates similar to Raiden Fighters. The player is able to pick up item to either change what kind of weapon he/she has as well as upgrade the current weapon on the ship. The objects can also be lost if they go off screen.

1. (a) Problem and Motivation:

After I had the weapon types and levels mechanics working properly I needed a way to actually generate the bullets in the game. The system needed to be able to know what kind of weapons the ship had to generate the correct type of bullet. Another issue was that the system needed to be generic so it could be used by player controlled ships as well as machine controlled enemy ships. Yet another issue was that the bullets needed to be distinctly enemy or friendly so the player wouldn’t be damaged by his/her own shots.

2. (b) Background:

The idea behind having bullets is simple, shooter games need bullets or it isn’t a shooter. Without functioning weapons there wouldn’t be a game. I didn’t necessarily base my system off of any other games as much as the shooter genre itself.

2. (c) Solution:

My solution to bullet generation was to create a system that would take in data from a source and output a bullet object at a specific spot. I chose to make the bullet generation its own system to make is generic and reusable. I did this through the state design pattern which made the system loosely coupled with the type of ships firing bullets, reducing the workload for the team.

Basically, the system works by passing the weapon type and weapon level to context so that the state of the weapon system can be set. Once the state is changed to the correct bullet type a fire method is called which will actually generate the bullet objects. The player’s ship does this in the KeyController class. As the player hits the spacebar key the system is initiated, as seen in this snippet.

**switch (mainShip.getWeaponState()) {**

**…**

**case 2: //missile**

**KeyController.bullet.setState(missile, mainShip.getLevelState());**

**//System.out.println(this.bullet.getState().toString());**

**KeyController.bullet.fire(mainShip.getX(), mainShip.getY());**

**Sound.shot2.play();**

**break;**

**…**

Bullet is an object of class Context and the state is set to the weapon type and level of the player’s ship. The fire method is called next which spawns the correct shot at the given position. Then a sound is initiated. This system can be used exactly the same for enemy ships as well. The state and position need only be set by an enemy ship instead of a player controlled ship.

The last issue is to determine if it a shot is friendly or foe, and the solution to this was to create two bullet array lists to use for collision detection. The two array lists are stored in GameData and when the bullets are created they are added to one or the other.

**switch (wls) {**

**case 0:**

**synchronized (Main.gameData.friendlyBullets) {**

**Main.gameData.friendlyBullets.add(new LaserBulletBaseLevel(x + 20, y - 40, false));**

**break;**

**}**

**…**

As seen above the array list that the bullet is added to is set beforehand. To change this from the friendly bullet array list to the enemy bullet list, the only thing that would need to be changed would be

**Main.gameData.friendlyBullets**

To

**Main.gameData.enemyBullets**

Finally each bullet has two defined behaviors, one for the enemy bullets and one for the friendly bullets. The data is passed through the third parameter for the bullet type, false is for the friendly bullet behavior and true is for the enemy bullet behavior.

2. (d) result:

What is accomplished here is exactly what we needed. The system will create the correct bullet wherever it needs to be spawned at. It is also generic which allows it to be used for enemy and friendly ships alike. The bullets that are produced with this system behave as expected and makes up one of the most essential mechanics of the game.

3,(a) Problem and Motivation:

The main problem was actually figuring out how sound file work in java programming. This wa my first time ever adding sound effects to a game. I researched my different techniques online and decided using java applets were better suited for a project like this. I wanted the music to be cool and similar to other games and I also wanted all actions to have sound.

Given the fact that we had different types of weapons in the game I need to be sure that each shot had a different sound. Without that the game would feel boring and not very detailed. The main problem was finding different sounds for the respective weapon used.

3.(b) Background:

All games have sound, without it the game just seemed incomplete. So we knew that adding sound was a major thing that we needed to implement. Sound is almost important as gameplay when making a game. So having everything silent was not an option.

3.(c) Solution:

I decided that using java sound applet would be a good solution for our game, We didn’t have that many sounds so it would be best to just have the program load them when they were needed.

public static final Sound sound1 = new Sound("/sounds//frantic.wav");

public static final Sound shot = new Sound("/sounds//shot.wav");

public static final Sound shot2 = new Sound("/sounds//laser.wav");

public static final Sound shot3 = new Sound("/sounds//teleport.wav");

public static final Sound dead = new Sound("/sounds//explosion.wav");

This code tell the program where to look for each file. When the file is needed.

Sound1.play();

Shot.play();

Shot2.play();

Shot3.play()

Dead.play();

These all tell the program whe to play each file and there are other functions like “.loop()” and “.stop()’ I used the loop function to replay the background music over and over while the game is playing.

3.(d) Result:

The result was pretty good. All sound preform as you would expect them too and the background music really fits the game.

Do not simply write what you did or accomplished, but write HOW you achieved (technical contents) in terms of novel ideas, algorithms, design patterns, architectures, etc.

Q: **Abstract Contents**

I just wanted to clarify some what the Abstract should contain.

1. Is it fine to include some diagrams in the abstract?  
For example, can ER diagrams be included for when detailing the database design.  
  
2. Do sections of the abstract have to be labeled with each member's name?

For example, if Member A worked mainly on technical feature 1 does the section in the abstract about feature 1 have to be labeled with Member A's name.

A:

1. Yes, you may include diagrams, tables, pictures, etc.

2. No, you don't have to. I will refer to the final individual demo to assess one's contribution to the project.