TITLE OF PROJECT:

AUTHORS: Michael McGregor, Brian Davis, Yuchen Feng, Ryan Conyac, Will McLain

PROJECT ABSTRACT

1. (a) Problem and Motivation:

Building an old arcade style shooter posed quite a few challenges for us. One of the initial problems was how to implement a mechanic for the player to be able to change the weapon of the ship. We wanted the game to have many different unique weapons that were broken up into three categories of weapons, kinetic, laser, and missiles. We also wanted the player to be able to decide whether or not they want a specific weapon or not. With each event in the game whether it be a bullet being shot or a ship being destroyed we wanted there to be a sound attributed with it. The sounds posed problems of their own that we had to overcome. Along with the visual and audio aspects of the different bullet types, we also needed a way for user to understand what the objects were through an easy to understand interface. Another issues that arose was managing the images for all of the objects in the game. Initially every object had its own image which put a significant strain on the system that we had to deal with. The final issue was creating the background that the whole game would play on.

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

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

Given the fact that we had different types of weapons in the game we 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.

The initial problem is how to let a player know what is each item doing during the game, like ship, different weapon, and bullet and enemy ships. So the hover over feature was added in the game that when a player is playing the game, he/she can move the mouse to an item and it will show the tooltip what the item does. Since the game have different weapon, then the player can easily see what each weapon does.

The problem of creating the level was something that had to be sorted out for us in the main part of the game as having different levels is a large part of keeping user interaction and involvement up. This had to be done so it would not largely impact the players experience and not hinder their ability to see enemies on the screen and not be a large distraction from the gameplay. The level also had to contain the players save data and scoring information to be used in the game as the leaderboard needs a place to update the information.

(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.

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.

So what is hover over, a mouse hover triggers an event when a use places a mouse over a designated area, such as hyperlink on a Web page. The action of moving the mouse over the item causes event such as pop-up windows or description boxes. People can see this feature everywhere, on web page, in game, and so on. In some very old game, they may don’t have this feature, but right now there are getting more and more in the game. It is a good feature for player, so it was added in our game.

Levels have been the sole major interactive piece of games since their introduction. They can be found to communicate not just the heads up display and other key useful information. But also what enemies or challenges the player can face on the level. The main draw of using such things in at best it can allow the player to feel more immersed into the game and allow them to gain more information that just from help or from reading sources as it allows them to see what is happening and be able to react as they need to overcome the problem.

(c) Solution:

For our game we decided to use the same idea that Raiden Fighters used. We implemented it a little differently than they did though, because instead of having one weapon item that changes we 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.

We 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 when to play each file and there are other functions like “.loop()” and “.stop()’ we used the loop function to replay the background music over and over while the game is playing.

For this feature, the proxy design pattern was used in the game. First, the GamePanel class implement MouseMotionListener to get the mouse position, and we changed all figureRender method to figureRending. In the figureRendering method, it uses the proxyfigure to render the object. The right part of the diagram is the proxy design pattern, and the EnemyShip, DefaultShip and Missle is GameFigure. In the GamePanel class, when the figureRendering method is called, it will create the ProxyGameFigure instance, and the ProxyGameFigure will call the onRendering method, this method checks if the rectangle of the GameFigure contains the mouse point, if it does, it will call the renderToolTips to show the GameFigure information, and otherwise, it will only call the render method. A diagram of the classes can be found in the appendix item 1.

For the level a few items had to be worked out. The level had to contain the images used for both the first level and for the transition and for the second level that the player will see. The level would have to use the sizing of the image to determine when to loop the image back otherwise the image would slide off the screen and not update and show the user. This is an example of this code as the images for the level are constrained to certain size limits, see appendix 2 for the image. In this example a new image is drawn once a certain point is reached so the two images are placed on top of each other until the second one reaches a merge point. Once that merged point or a point where the two images have similar looks so the background seems to not jump so badly the entire image set is moved up to the top of the field of view to begin scrolling down the user’s view again. This continues until the level ends, is paused, or the game is quit.

(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.

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

In the demo, player can see the hover over feature works well, when you move to the different weapon, it show the different weapon information. When a player is control different ship, it will shoe different ship’s information.

The result for the level produces an image and content that is contained for the user to see easily and without many problems. The scrolling does not detract from the items or enemies that are on the level and still allows the user to get information about what level and what it happening. This sort of system can be easily expanded to a number of levels containing different enemies or backgrounds allowing for a larger variation for the player to experience.

1. (a) Problem and Motivation:

After we had the weapon types and levels mechanics working properly we 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 player-controlled ships could use it 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.

(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. We didn’t necessarily base my system off of any other games as much as the shooter genre itself.

(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. We chose to make the bullet generation its own system to make is generic and reusable. We 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

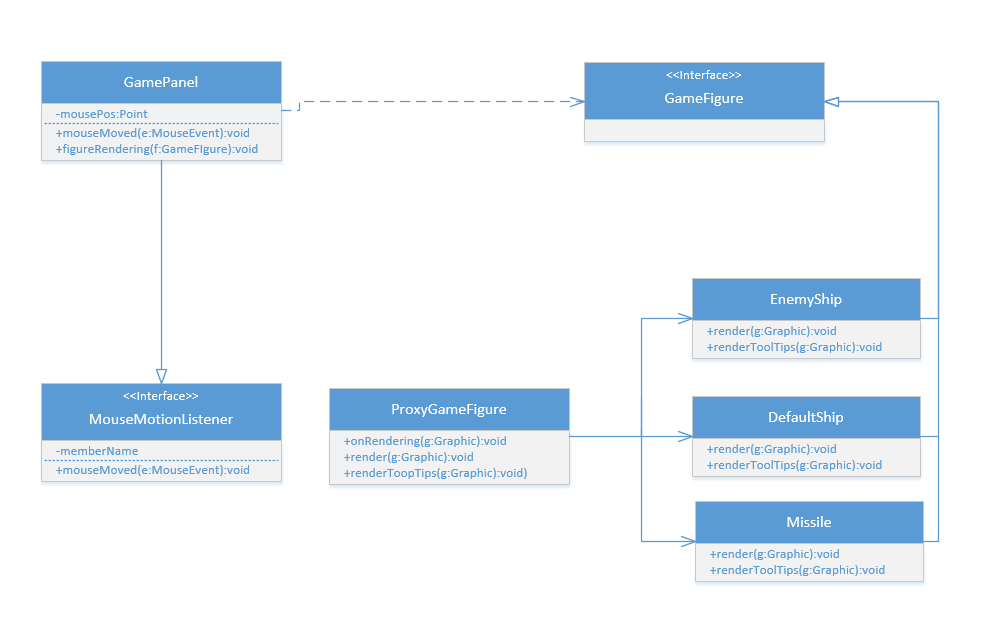
**Main.gameData.enemyBullets**

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.

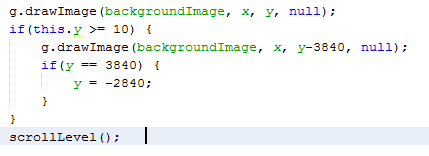
(d) Result:

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.

Appendix:

1.

2.



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.