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. Obviously these bullets had to be shot at and towards something so a ship class had to be built. The Ship class had to be generic enough to use for all the ship types in the game which created a challenge. 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 issue 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.

(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 objects 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.

To relay information to users we implemented a hover over feature. So what is hover over? Hover over events can be seen when a user 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. This is a common feature of today’s application, on web page, in game, and so on. This feature is more prevalent in modern application rather than older ones. We added this to our application because it is easy to use and understand.

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.

As the number of items in a game begin to increase the optimization issues become a large hindrance in player interaction and playability of a game. Enemies spawned onto the level can create a large amount of memory usage as more are in the world, using the flyweight pattern helps curb this. As the enemies are rendered on the screen they are referencing a single instance for their image and are only needing to keep their own reference of position and other details. Helping to reduce the memory usage, this was also implemented for the bullets as well as the player and enemy ships shoot more the images do not contribute significantly to the resources used by the computer.

(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. 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. Figure 3 displays a code snippet of how this is done.

When 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 by 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.

In order to make a reusable system to generate bullets we created a system that would take in data from a source and output a bullet object at a specific spot. 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, refer to figure 4 for a code snippet of how this works.

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.

Our group initially planned to have each member of our group to create one unique ship design. This turned out to be problematic because it took more time to design a cool ship design than we had anticipated. We also used a combination of the abstract factory pattern and the flyweight pattern to render the ships. It allowed us to combine six Ship class implementations into one NewShip class. We did this by using one implementation of the Ship class that uses a static EnemyFlyweightItems object, which contains all images needed for the Ship class. Refer to Appendix item 6 and 7 to see an example of the image selection algorithm. The instance of the NewShip class references the static instance of EnemyFlyweightItems method setShipImage() method. The following methods, isShipDamaged() and render() show how the NewShip class uses the EnemyFlyweightItems to display the correct ship image.

The image displayed is determined by the the context of the String variable shipType, current shipState and state.  For example, if the shipType is “defaultship”, shipState is 13 and  state equals STATE\_TURNING\_LEFT then the ship image displayed will be the Yellow default ship with left wing down.

To add sounds to the game we decided that using java sound applet. 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 tells 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.

The flyweight pattern was created with an interface allowing the calling entity to determine what image it is supposed to receive. This is done by verifying the type of object passed into the flyweight then checking the state of the object and returning the image file for the requesting object. The UML diagram for the flyweight can be found in appendix 5. The object is passed to the FlyweightItem which handles the object using switch statements to handle the object and assign the static images that are instantiated when the EnemyFlyweightFactory produces the EnemyFlyweight items for use in the GameData.

(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.

In the end, we were able to create multiple ship types.  The individual ships each have a different look and feel which our group was planning to do.  We were also able to do this with minimal system load.

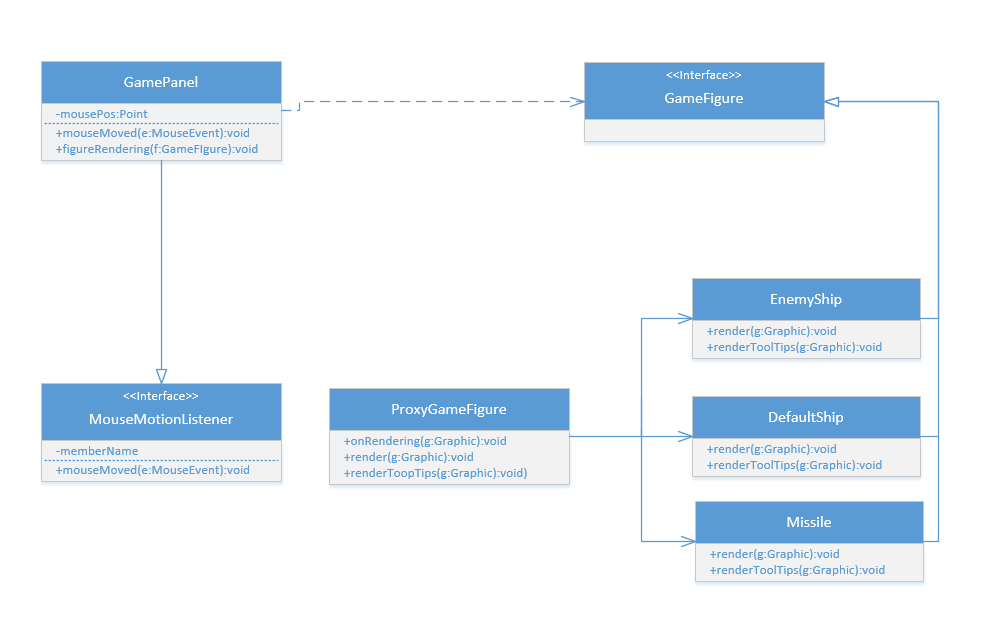
The result was pretty good. All sound preforms as you would expect them too and the background music really fits the game.

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.

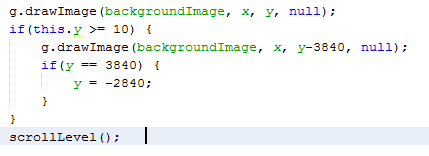
The result for the level produces an image and content that is contained for the user to see easily and without many problems. The scrolling does not detract from the items or enemies that are on the level and still allows the user to get information about what level and what it happening. This sort 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.

With the inclusion of the flyweight pattern memory usage was able to be optimized to use up to 30mb less. This allowed a smoother framerate to be produced as well as running better for debugging purposes on the different developer computers.

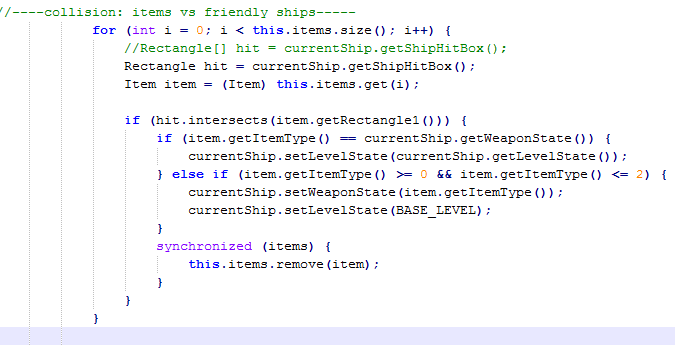
Appendix:

1.

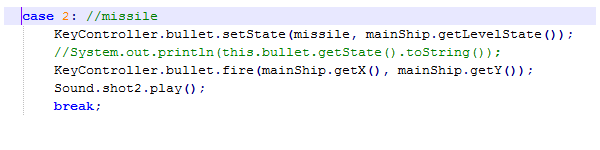
2.



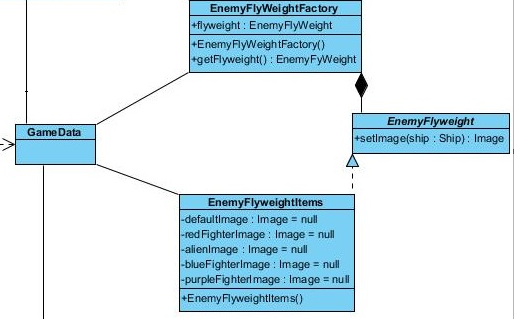
3.



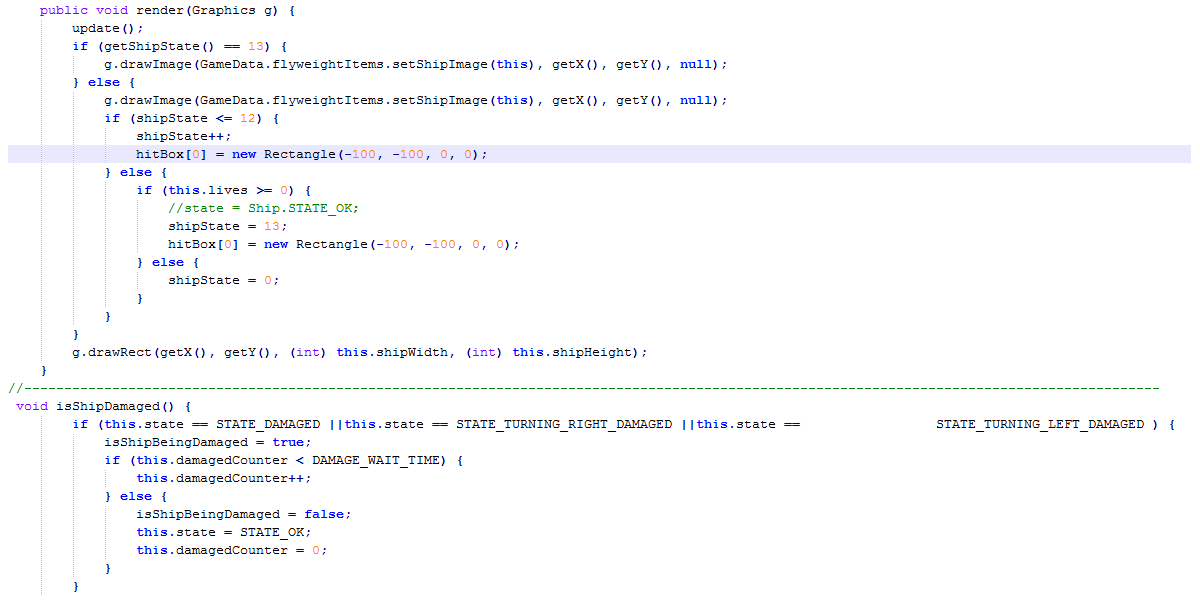
4.



5.



6.



7.

