Super Asteroids

Contents

Acknowledgements....2
Introduction....3
Super Asteroids: A Quick Overview....4
Source....5
Design Document....6
Tasks....7
1. Design the Database....7
2. Create the Model Classes....7
3. Create the Data Access Classes....7
4. Test the Data Access Classes....8
5. Code the Data Importer....8
6. Load Model from the Database....8
7. Test Data Importer and Model Loading....8
8. Complete the Ship Builder....9
loadContent()....9
onPartSelected()....9
unloadContent()....9
update()....9
draw()....9
onStartGamePressed()....9
onViewLoaded()....10
onSlideView()....10
9. Implement the Quick Play Button....10
10. Make the game....11
a. Design the game viewport....11
b. Draw the background....11
c. Draw the background objects....11
d. Draw the ship....11
e. Move the ship....12
f. Fire projectiles....12
g. Create the asteroids....12
h. Add the Mini Map....12
i. (Optional) Code and test the QuadTree....13

j. Collision detection....13
k. Transition to the next level....13
l. (Optional) Create a level transition scene....13
Project Tips....13
Source Code Evaluation....14

Acknowledgements

The Super Asteroids project was created by Tyler Monson. Thanks to Tyler for this significant contribution.

Introduction

The arcade game *Asteroids* was a big hit when it was released back in 1979. In the game, the player maneuvers a spaceship to pulverize asteroids and the occasional flying saucer.

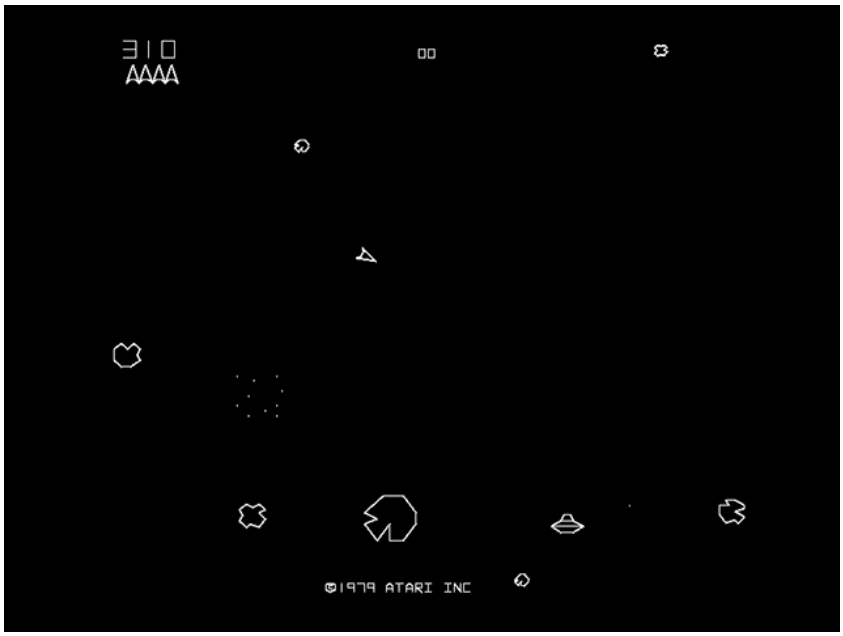


Figure 1 Asteroids 1979 - Source: http://upload.wikimedia.org/wikipedia/en/thumb/1/13/Asteroi1.png/220px-Asteroi1.png

A version of the game can be played here:

In this project, you will create *Super Asteroids*, a variation on the original *Asteroids* game. Your *Super Asteroids* game will be different from the original game in the following ways:

1. The game will be played on an Android device.
2. The player can customize its ship before starting to play the game.
3. Content for the game will be loaded from a database, making the game somewhat customizable.
4. The game "world" extends beyond what the player can see on the game view at any given time.
5. Graphics and sound effects will be greatly improved.

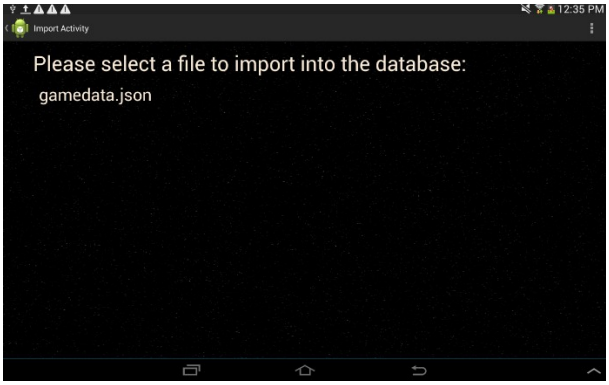
Throughout the course of this project, you will gain knowledge and experience with the following concepts, skills, and technologies:

- Design, implementation, and testing of a relatively large, multi-faceted Java program
- Object-oriented design
- Relational databases and Android's SQLite framework

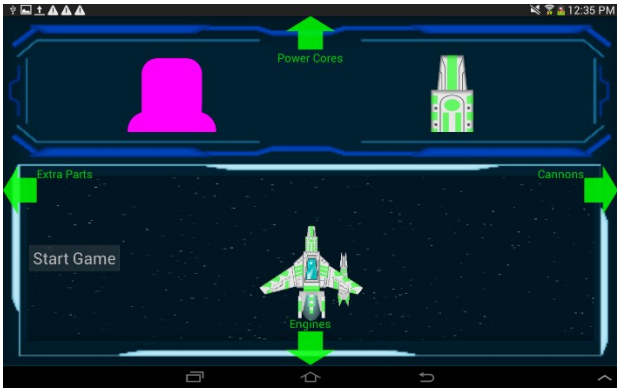
- JSON parsing
- Simple game design and testing
- The Quad Tree data structure
- Android development: Android Studio, unit testing, and debugging

Super Asteroids: A Quick Overview

Super Asteroids starts out with a title screen. Clicking “Import Data” leads to the import screen.



Once the player clicks the “Start Game” button, the ship building screen appears.



On this screen, the player is able to customize their ship. Customizing is done by choosing a “Main Body” ship part, then adding a “Cannon”, “Engine”, “Power Core”, and “Extra Part” to it. Labeled green “helper arrows” assist the player in traversing the structure of the ship building interface. Once the ship is fully assembled, the “Play Game” button is enabled. The game screen appears once the button is pressed.

The “Quick Play” button can be used to skip the ship building screen and go straight to the game screen. A random ship is selected when using the “Quick Play” button.

The game screen is where all the action takes place. Upon entering this screen, the first game level is loaded and the player can start blasting asteroids. The player guides the ship and fires its cannon through touch input. The ship will turn and accelerate towards wherever the player is touching the screen. The ship will quickly come to a stop if there is no touch input detected. To fire, the player need only tap the screen.

To get a better feel for the game, make sure to check out the demo. Information on how to obtain and run the demo can be found in the following “Source” section.

Source

The source code for this project can be downloaded from the CS 240 website. Once the code is downloaded, extract it to a folder of your choice. Next, run Android Studio and select “Open an existing Android Studio project” from the Android Studio welcome screen (or, select File >> Open menu within Android Studio). A dialogue will appear, prompting you to choose the project to open. Navigate to and select the previously extracted “SuperAsteroids” folder. Push OK. This should open the *Super Asteroids* project.

Once the project is open, take some time to explore the code in Android Studio. If you cannot see the project explorer in Android Studio, select the “Project” tab found on the left side of Android Studio. All of the Java code is located in the app >> src >> main >> java folder. Android view layouts and other files are found in the res and assets folders. Even though you may not know exactly what all of the source code is doing, please take some time to review it.

At this point, another great thing to do is review your source code TODO items. To view all project TODO items, simply click on the “TODO” tab found at the bottom left corner of Android Studio. These TODO items do not need to be completed now, but will need to be done at certain points in the project.

The source code comes with a demo of *Super Asteroids*. This demo should be used to get familiar with the game’s gameplay and features. Playing the demo should also give you a jumpstart on thinking about how you are going to design and code the game.

In order to install and run the demo, the “adb” (Android Debug Bridge) program must be on your PATH environment variable. You must also be running an Android emulator or have an Android hardware device connected to your USB port. Assuming adb is on your PATH, and you have a device or emulator available, you should be able to successfully install and run the demo using the command line. You can use the Android Studio terminal, or another terminal, to do this.

Android Studio Terminal: Select the “Terminal” tab near the bottom of Android Studio. In the terminal, type **./gradlew app:demo** and press enter. If a device is connected and on, or an emulator is running, the demo will install and run.

Another Terminal: Open a terminal of your choice and navigate to your *Super Asteroids* project directory. Execute the command **./gradlew app:demo.** If a device is connected and on, or an emulator is running, the demo will install and run.

Design Document

The first step in developing a program like *Super Asteroids* is to spend some time understanding the problem you are trying to solve. Once you understand the problem, you can start to design the program by creating classes that perform each of the functions required by the program. Each class should be documented with its responsibilities and how it interacts with other classes. This process will help you determine the classes you need to write, and will help you understand how the classes work together to produce a working program.

Once you've thought through your design the best you can without writing any code, you should make a first attempt at implementing your design. As your implementation progresses, you will probably find your design was incomplete or faulty in some respects. This is to be expected, because some insights only come from actually writing code. As you proceed, make necessary changes and corrections to your design and incorporate them into your code.

To encourage you to follow this design process, you will be required to submit a design document for your program. Your design document should include two parts:

1. The Database Schema: Create a text file containing all of the SQLite Create Table statements needed to create all of the tables in your database schema.
2. Class Documentation: Create class stubs for your **Model Classes** and **Data Access Classes**. Use the Javadoc creator to generate Javadoc documentation for your project.

Start the electronic submission of this assignment by creating a website containing the two items above. Finish the electronic submission of this assignment by sending the TAs an email containing:

1. Your name
2. The URL of your SQL file
3. The URL of your Javadoc's index.html file

The TAs will follow the provided URLs to view and grade your design documents. **It is your responsibility to ensure the URLs work properly**. Broken or misdirected URLs will leave the TAs unable to grade your work.

It might be news to you that any files placed in the **public_html** directory of your CS home directory are automatically published to the web. This directory allows you to easily create your own website. For example, the URL for the personal website of a user with the BYU CS login "fred" would be . If Fred placed a file named **database.txt** in his **public_html** directory, the file would be accessible on the web at **http://students.cs.byu.edu/~fred/database.txt.**

You may use your CS web site, or some other website, to submit your design document.

Tasks

If you have not done so already, please read the “Source” section of this document before continuing.

The following are the steps you need to take to complete this project.

1. Design the Database

Using the information found in the Asteroids Data document as a reference, design a SQLite database for the game. The database should be made up of relational tables designed to hold the data related to the various elements of the game. However, the database should not be designed to store image or sounds files. Media files such as this will be stored in the assets folder of the Android application. If a media file is related to an element of the game, the game element should store the file path to the media file. The file path to the media file starts in the assets folder. Example: If the sound file “laser.ogg” is related to the cannon game element and is located in the “sounds” subfolder of the assets folder, the database would store the string “sounds/laser.ogg” as the cannon sound effect.

After you have settled on a schema for your database, create a text file containing a “drop table” statement for each table in your schema. The “drop table” statements will be followed by the “create table” statements for each of the tables you have designed.

Firefox’s “SQLite Manager” plugin is an excellent tool for interactively designing, creating, querying, and manipulating your database.

2. Create the Model Classes

Create a package of Java classes modeling the core information manipulated by the game. The package will include classes such as Cannon, Asteroid, Level, ExtraPart, etc. Generally, the model classes will store the same information found in their respective database tables.

Model classes serve several important functions in the program. Not only are they crucial for storing and manipulating data when it is in memory, but they also help in the process of transferring data between memory and the database. Whereas the model classes are mainly data containers, they will also contain important algorithms needed to run the game.

Make sure to put these classes in their own package.

3. Create the Data Access Classes

Create one or more Java classes that encapsulate your database access logic. These classes will be used by the game to load data from the database into

model objects. They will also be used by the Data Importer to populate the database with data from JSON game files. All of the code used to directly access the SQLite database should be contained in your data access classes. These classes should provide operations for querying, inserting, updating, and deleting data in the database. These classes should be implemented using Android's SQLite framework.

Once again, make sure to put these classes in their own package.

4. Test the Data Access Classes

Throughout the development of your data access classes, you should be using JUnit to write automated unit tests for those classes. These tests should be designed to verify the correct execution of all of the operations on each of your data access classes.

Thoroughly testing your database access classes through Junit at this time will:

- Save you debugging time in the future
- Demonstrate how to isolate and verify a small portion of a program

5. Code the Data Importer

With your model classes and data access classes coded and tested, you are ready to start working on your data importer. Coding and debugging your data importer is a crucial piece of this project for several reasons:

1. It gives you the opportunity to learn how to parse data, particularly JSON data.
2. The importer allows the TAs to import content into your game, thus enabling them to test your game's ability to support dynamic content.

At project pass-off time, a TA will copy a game data .json file along with image and sound files into your project's asset folder. After installing the game onto your device, the TA will import the above mentioned .json file into your database, then play the game using this data.

Your data importer must implement the **IGameDataImporter** interface found in the **importer** package of the project source code. To ensure your importer code is called when a file is selected, please complete the TODO item found in the **onCreate()** function of the **ImportActivity** class. (The **ImportActivity** class is the **importer** package.)

6. Load Model from the Database

Write the code that will load data from the database into your model objects. Of course, this code should use your data access classes to load the data. You could load all of the game's data from the database at the beginning of the game so that everything is in memory the entire time. Or, you could load data from the database dynamically as it becomes needed (e.g., load the data for a level when the user enters that level, and throw it away when the user leaves the level).

To connect your model initialization/loading code into the program, complete the TODO in the **onCreate()** function of the **MainActivity** class.

7. Test Data Importer and Model Loading

Write JUnit tests to verify that your Data Importer and model loading logic are working properly. These tests should: 1) Run the data importer to load a game into the database, 2) Run the code that loads data from the database into your model objects, and 3) Verify that the loaded model objects have the correct data in them. Once these tests are working, you will know that the data you are running your game with is correct, and that any bugs you find in your program later are probably not in the data importing or loading code.

8. Complete the Ship Builder

Code implementing the visual aspects of the ship builder has been provided. The existing code provides functionality to:

- Start the load content process
- Detect fling touch input
- Display the parts available for building the ship
- Detect ship part selection
- Animate to a new part selection view
- Show, hide, or change the text of the helper arrows
- Enabled or disable the “Start Game” button
- Start the actual gameplay

The existing code does not:

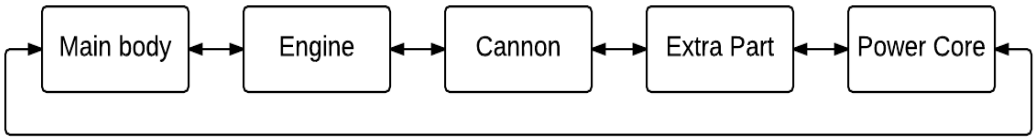
- Finish the content loading process
- Respond to fling touch input
- Have any parts to display (the parts must come from your database)
- Respond to ship part selection
- Configure the helper arrows
- Enable the “Start Game” button when all ship parts have been selected
- Respond to the “Start Game” button pressed event

To complete the ship builder, you must write a ship building controller that implements the **IShipBuildingController** interface. Thus, your controller will be required to implement all of the functions in the **IShipBuildingController** and **IGameDelegate** interfaces. All of these functions will be called by the existing code, so you will never need to worry about calling them yourself. The following are short descriptions of the functions you will be implementing in your controller:

loadContent() – Use this function to extract ship part data from your database. Then, use this data to load ship part images. Once the images are loaded, pass the image IDs to the **ShipBuildingActivity** using the **setPartViewImageList()** function.

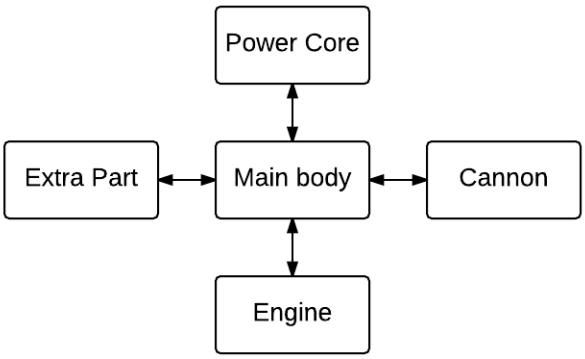
onPartSelected() – Respond to changes in ship part selection here.

unloadContent() **–** The **loadContent()** function loaded many images into memory. Yet, most of these images will not be used for the rest of the game. Use this function to unload all of those images.
update() – Leave this function empty.
draw() – Use this function to draw the ship. **Note: The document "The Ship: Putting it All Together" will help you understand how to draw the ship.**
onStartGamePressed() – Call **ShipBuildingActivity's startGame()** function, which will cause the **GameActivity** to be started.
onViewLoaded() – Use this function to configure the helper arrows for the part selection screens. Helper arrows should only be configured in this function. Doing so ensures the views for the arrows have been created. Configuring the arrows outside this function will throw a null pointer exception, crashing the app.
onSlideView() – This function is called as the player makes a fling motion on the device. The fling motion indicates the player is trying to change to another part selection view. Your implementation of this function should use the **ShipBuildingActivity's animateToView()** function to animate the transition from one part selection view to another.
The main body selection view is always the first selection view in the ship builder. Yet, with four fling directions and four other part selection views, there is room to customize the layout of the part selection views. For example, the part selection views could be lined up like this:



With this example, the fling UP and DOWN directions are meaningless. In this case, starting at the main body view and flinging from right to left on the view should animate the engine view in from the right and animate the main body view out to the left. Imagine you are pushing the current view away and pulling the new view in with your motion.

Alternatively, you could use a configuration similar to the one seen to the right.



To get a better understanding of this, please read the documentation for the existing code. Also, take a look at the demo.

Configure the helper arrows to help the player know their way around the ship builder interface you have created.

To connect your ship building controller to the **ShipBuildingActivity**, complete the TODO found in the **onCreate()** function of the **ShipBuildingActivity** class.

9. Implement the Quick Play Button

To implement the quick play button, you must write a main menu controller that implements the **IMainMenuController** interface, including the **onQuickPlayPressed()** function. This function will be called by the existing code, so you will not need to worry about calling it yourself. Your implementation of the **onQuickPlayPressed()** function should do the following:

1. Since the user has opted to not select their own ship configuration, select some ship parts for the user (randomly or whatever you like), and configure your program to use those parts.
2. Call **MainActivity's startGame()** function, which will cause the **GameActivity** to be started.

To connect your main menu controller to the **MainActivity**, complete the TODO found in the **onCreate()** function of the **MainActivity** class.

10. Make the game

Before attempting this task, make sure you read and understand the following accompanying documents:

- **The Ship: Putting it All Together**
- **Simple Game Architecture (Make sure you complete the TODO item!)**
- **Coordinate Systems**

There's a lot to do for this task, so it will be broken down into meaningful subtasks. Instead of detailing everything you should do, the tasks provide tips and restrictions for different parts of the game. It is your job to complete these subtasks by combining your knowledge of these tips and restrictions, with information from the lectures, demo, and supporting documents.

a. Design the game viewport

Your viewport should have the same dimensions as the Android view hosting the game. These dimensions can be accessed through the **DrawingHelper**. The viewport should be made up of static variables and functions. This allows its functionality to be accessed by any game element. The viewport rectangle MUST stay in the bounds of the world/level. Give your viewport functionality to transform world coordinates into view coordinates. When possible, the viewport should stay centered on the ship position.

b. Draw the background

Use space.bmp as the background for each level. The image should be stretched to fit the dimensions of the current level. Do not create a scaled version of space.bmp. Instead, determine where the viewport intersects with the image and draw that portion of the image to the whole game view. There is a **drawImage()** method on the **DrawingHelper** allowing you to draw a

portion of an image. Feel free to use another background image if you would like.

c. Draw the background objects

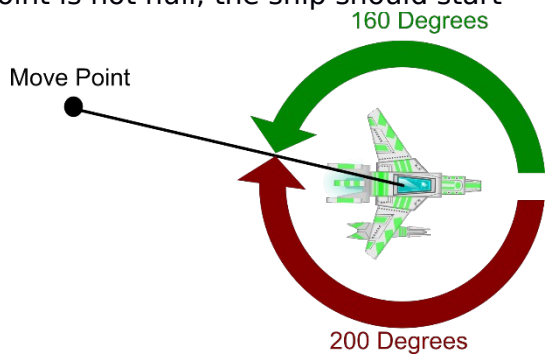
Background objects should be drawn after the background discussed above. They should be drawn in the order they are retrieved from the database, using the scale and position specified in the database. Remember, the object position retrieved from the database is in world coordinates.

d. Draw the ship

Draw the ship with all of its parts. Design the ship so it can be drawn at any rotation angle from 0 – 360 degrees. If the viewport is programmed correctly, the ship should draw to the center of the game view—unless the viewport is near the edges of the world space.

e. Move the ship

Whenever the **InputManager**'s movePoint is not null, the ship should start rotating to the point and accelerating in the direction the ship is currently facing. Please note that the **InputManager**'s movePoint is stored in view coordinates. Thus, the movePoint must be converted to world coordinates before it will be of any use. The ship must always take the shortest rotation route to the movePoint. For example: If the ship needs to rotate 200 degrees in the clockwise direction or 160 degrees in the counter clockwise direction, it should choose to rotate in the counter clockwise direction.



f. Fire projectiles

Fire projectiles from the ship's cannon when the **InputManager** indicates the fire button has been pressed and the cannon is not cooling down. Projectiles should be removed from the game once they have left the bounds of the world. Projectile starting points should be calculated using the cannon's emit point.

g. Create the asteroids

At the beginning of every level, asteroids should be generated at random positions with random velocities. To keep asteroids from being generated on top of the ship, make sure there is a safe zone around its starting position (center of the level) where asteroids cannot be generated. Asteroids should not be allowed to leave the bounds of the level. Instead, they should bounce off the edges of the level as seen in the demo.

Once an asteroid loses all of its hit points, it should split into pieces. “Regular” asteroids should break into two pieces. “Octeroid” asteroids should break into eight pieces. “Growing” asteroids should break into two pieces, and should also grow larger over time (if you want, you may cap the size of growing asteroids so they don’t grow infinitely large).

When an asteroid breaks into pieces, the new asteroids start with half the parent asteroid’s original hit points, and should be drawn at 1/n the parent’s scale (where n is the number of pieces the parent broke into). Randomly set the child asteroid’s velocity as it is created. Asteroids should only split twice before they are completely destroyed.

h. Add the Mini Map

Draw the mini map to any corner of the game view. The mini map should always be the same size, no matter the dimensions of the current level. Pick a color for asteroids and another color for the ship. Draw a dot at the appropriate location on the mini map for every active asteroid and the ship.

i. (Optional) Code and test the QuadTree

Code the QuadTree using information from the following tutorial:

After coding the QuadTree, develop and run some thorough Junit tests on it. Drawing your QuadTree is not required, but is a great extension to your testing.

j. Collision detection

Test for collisions in the game. (If you implemented the QuadTree, use it here to efficiently test for object collisions. If you did not implement the QuadTree, you will need to test all combinations of objects for collisions, which will be slower than using a QuadTree.) For every collision detected, use **touch()** functions to simulate collision behavior. For example, if objects *A* and *B* collided, you would call *A*.**touch(***B***)** and *B*.**touch(***A***)**. Program the touch functions with the following collision behavior:

Collision Behavior

- Nothing happens when two asteroids collide.
- Nothing happens when two projectiles collide.
- Nothing happens when a projectile collides with the ship.
- Projectiles collide with and damage asteroids.
- If the ship is not in safe mode and collides with an asteroid, the ship loses a life. If the player still has lives, the ship enters safe mode for 5 seconds (see the demo). Also, the ship should deal 1 damage to the asteroid if the ship is not in safe mode at the moment of the collision.

k. Transition to the next level

Once a player has destroyed all of the asteroids in the level, transition to the next level.

l. (Optional) Create a level transition scene

Make a level transition cut scene if you so desire.

Project Tips

1. Thoroughly read the documentation provided with the source code.
2. Make sure to complete all of the TODO items found in the source code.
3. The **GraphicsUtils** class in the **edu.byu.cs.superasteroids.core** package provides a number of useful functions for implementing the graphics algorithms required to implement this game. It is highly recommended that you review the code for this class and read the documentation for its methods, and then use this class to help you implement the game's functionality.
4. The app >> src >> androidTest >> java directory contains a class named **GraphicsUtilsTests**. This class contains JUnit test cases for the **GraphicsUtils** class. These tests serve as a useful example of how to write JUnit tests.
5. The spec does not give you all the details. Don't be afraid to use your intuition to fill in the gaps. For instance, the spec does not specify the maximum velocity for projectiles or asteroids. You can choose these values.
6. Think through your code before deploying and testing on a device. Carefully reviewing your code can spare you waiting for the app to deploy to the device.
7. Have fun with the project! Feel free to customize the game as much as you want while still fulfilling the requirements of the spec. You can create new asteroid types and new ship parts. You can even add some alien space ships that will chase and shoot at the player. Just remember, your importer always needs to be able import the basic data.

Source Code Evaluation

After you pass off your project, your source code will be graded by a TA on how well you followed good programming practices. The following criteria will be used to evaluate your source code:

- (15%) Effective class, method, and variable names
- (20%) Effective decomposition of classes and methods
- (20%) Code layout is readable and consistent
- (15%) Effective organization of classes into Java packages
- (30%) High-quality unit test cases implemented with JUnit