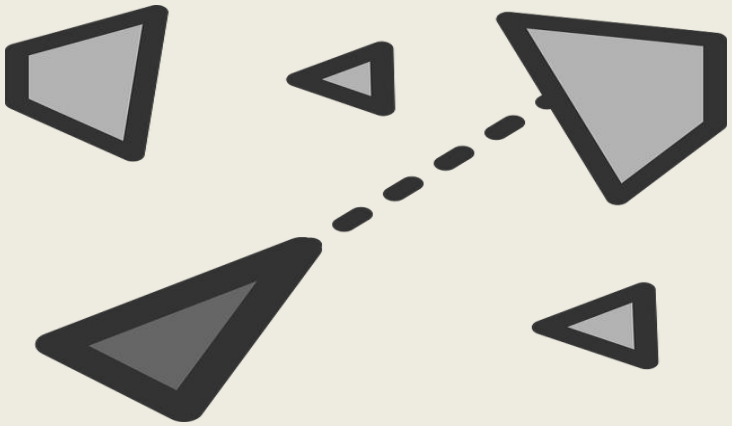
Asteroids

Software Requirements Specification



Team 4 - ISEtroids

Damien Girard
Siddhart Ghandi
Thuy-Anh Le
Genevieve Nantel
Vincent Petrella
Chi-Wing Sit

2/10/2013

TABLE OF CONTENTS

1. SYSTEM OVERVIEW

This document provides an overview of the system architecture developed by Team Astronoids as an implementation of the well-known game Asteroids for the group project assigned by Professor Zeng. The requirements outlined in the Software Requirements Specification document served as the foundations upon which the design and architecture decisions were based. The skeleton and specifications elaborated in this document are designed to support a multi-platform game application with a framework which allows further development, flexibility, and reliability.

The game interfacing is based on a centralized structure using an abstract class: 'GameObject'. The GameObject class contains basic attributes shared by all other classes in the game. Other classes which make up the system structure all extend this class. The architecture includes a database responsible for storing user high scores, statistics, and global high scores. All communication which writes to the database (storing high scores) is independent of the user and is done automatically based on the positive evaluation of a given set of conditions.

The GameObject root class structure provides flexibility for simple modification and improvement operations. As all other objects such as menus extend this class, new features can be added simply because these will inherit all GameObject qualities, which will allow them to interact with other objects using existing methods. The GameObject class encompasses all elements of the system, allowing for modularity which makes the system easy to understand, efficient, and easily amendable.

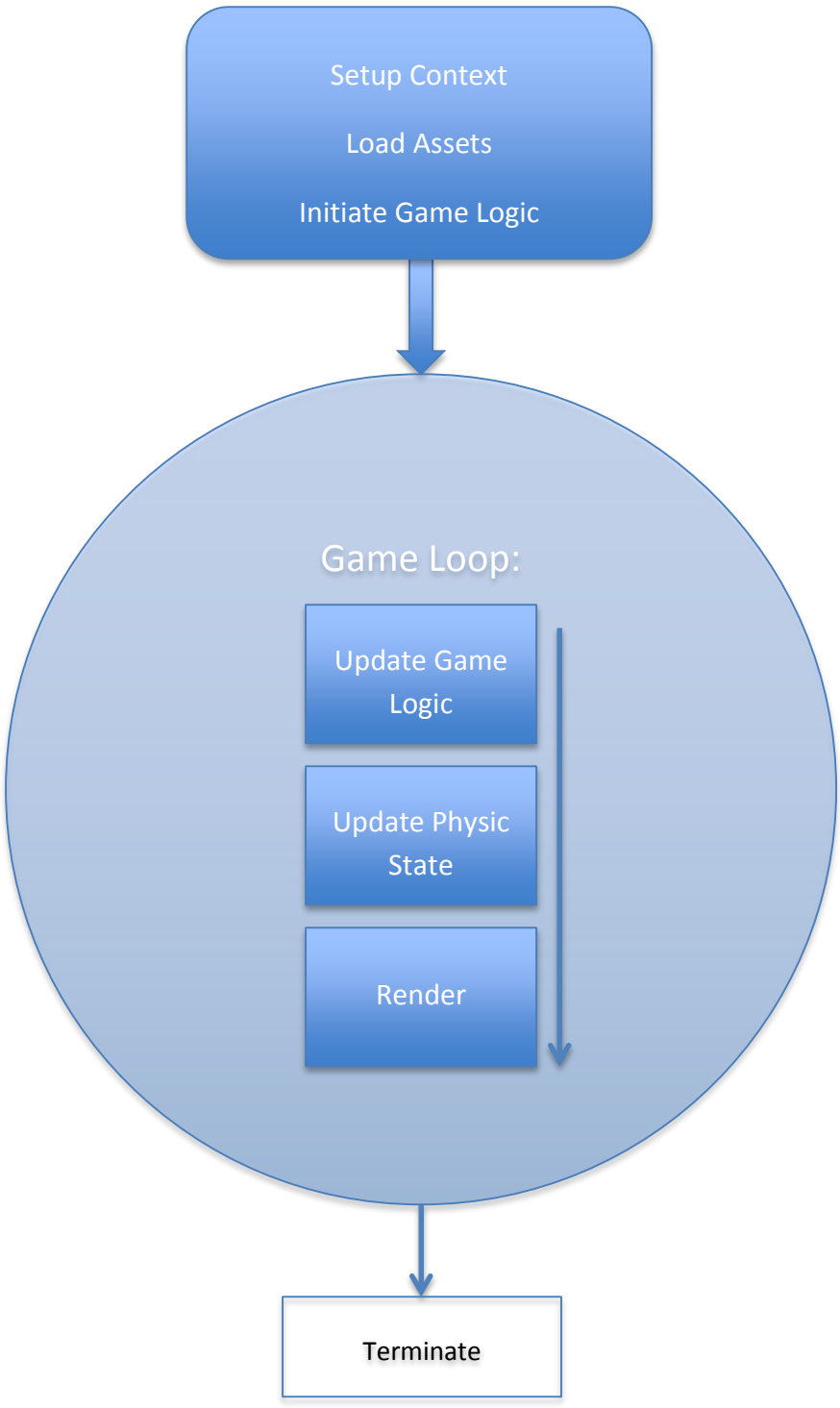


Figure 1- Basic System Architecture

2. VIEWS

2.1 USE CASE DIAGRAMS

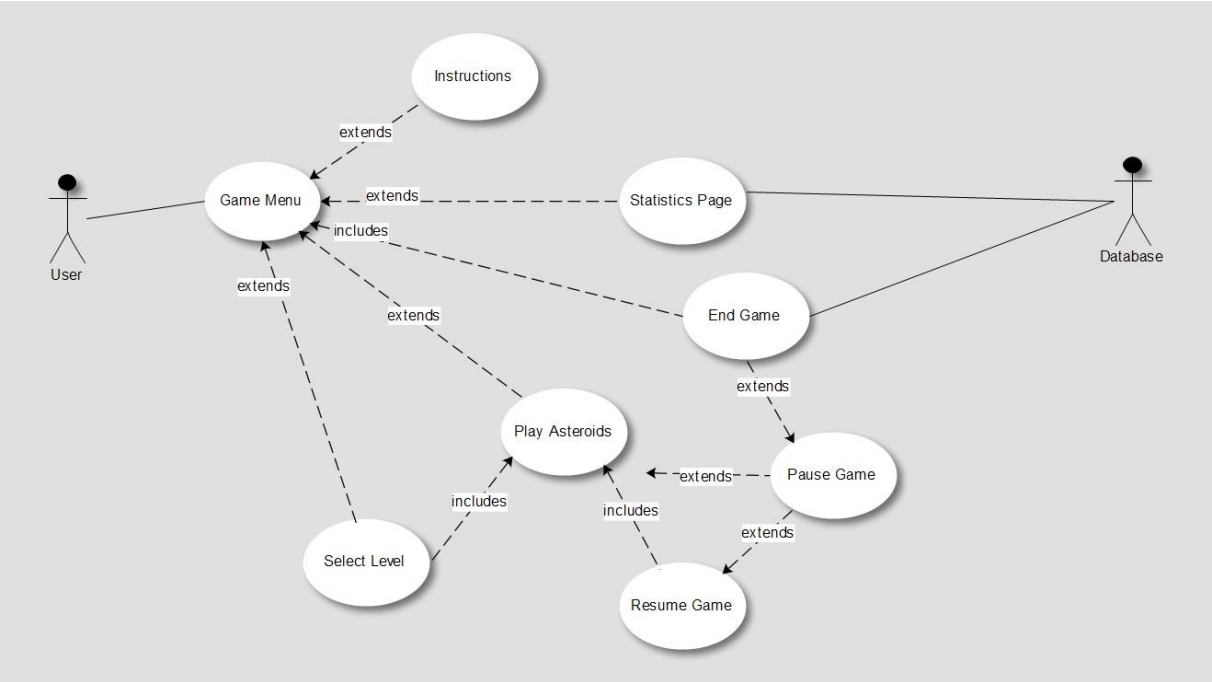


Figure 2- Game Menu

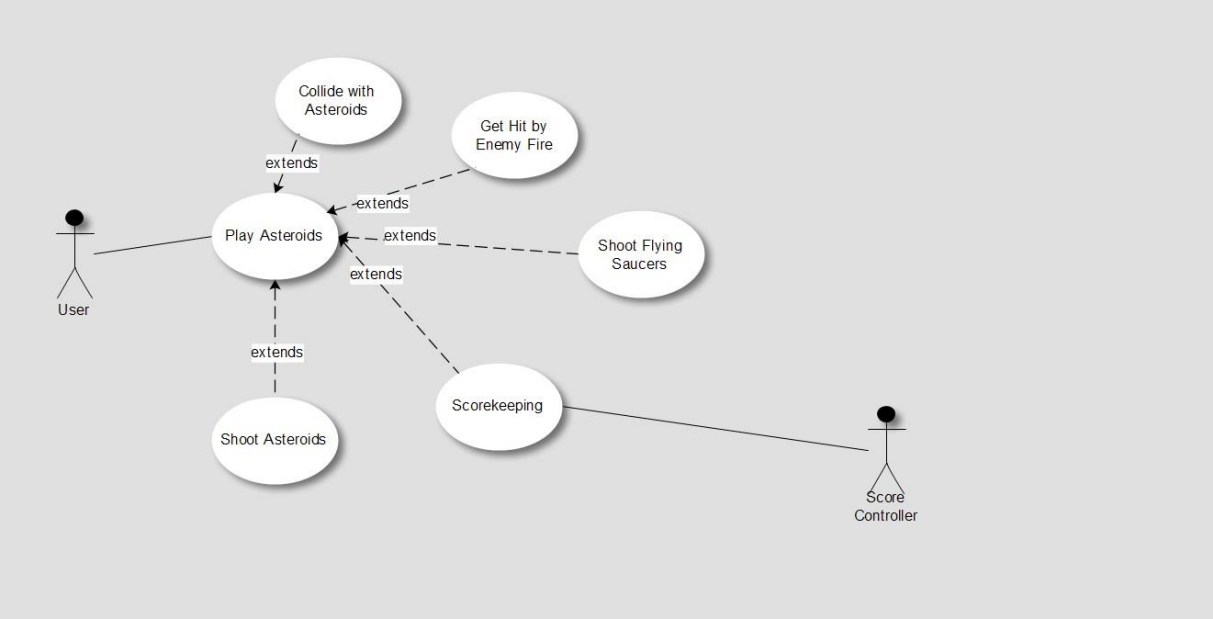


Figure 3- Play Asteroids

2.2 UML CLASS DIAGRAMS

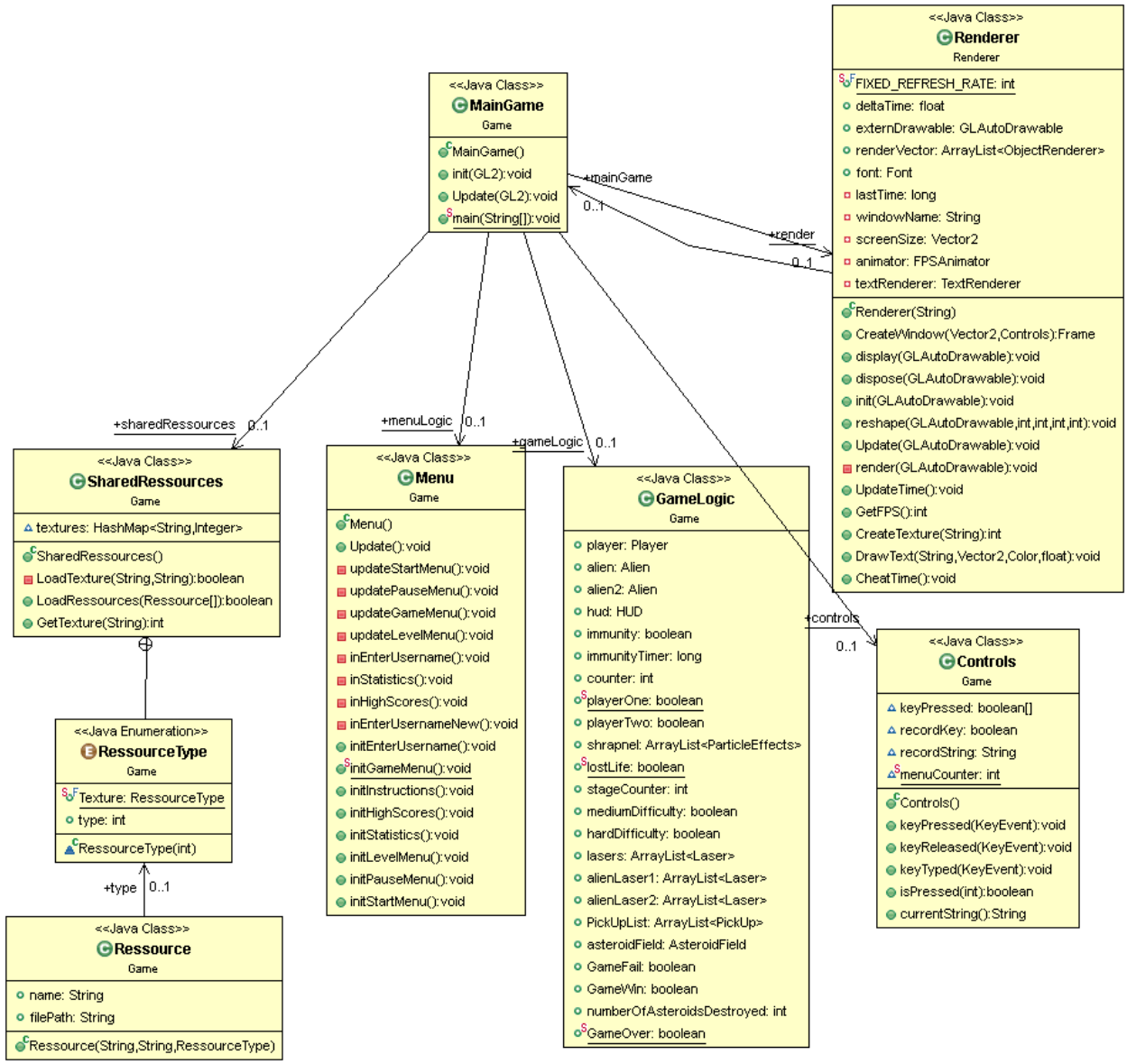


Figure 4- Main Game

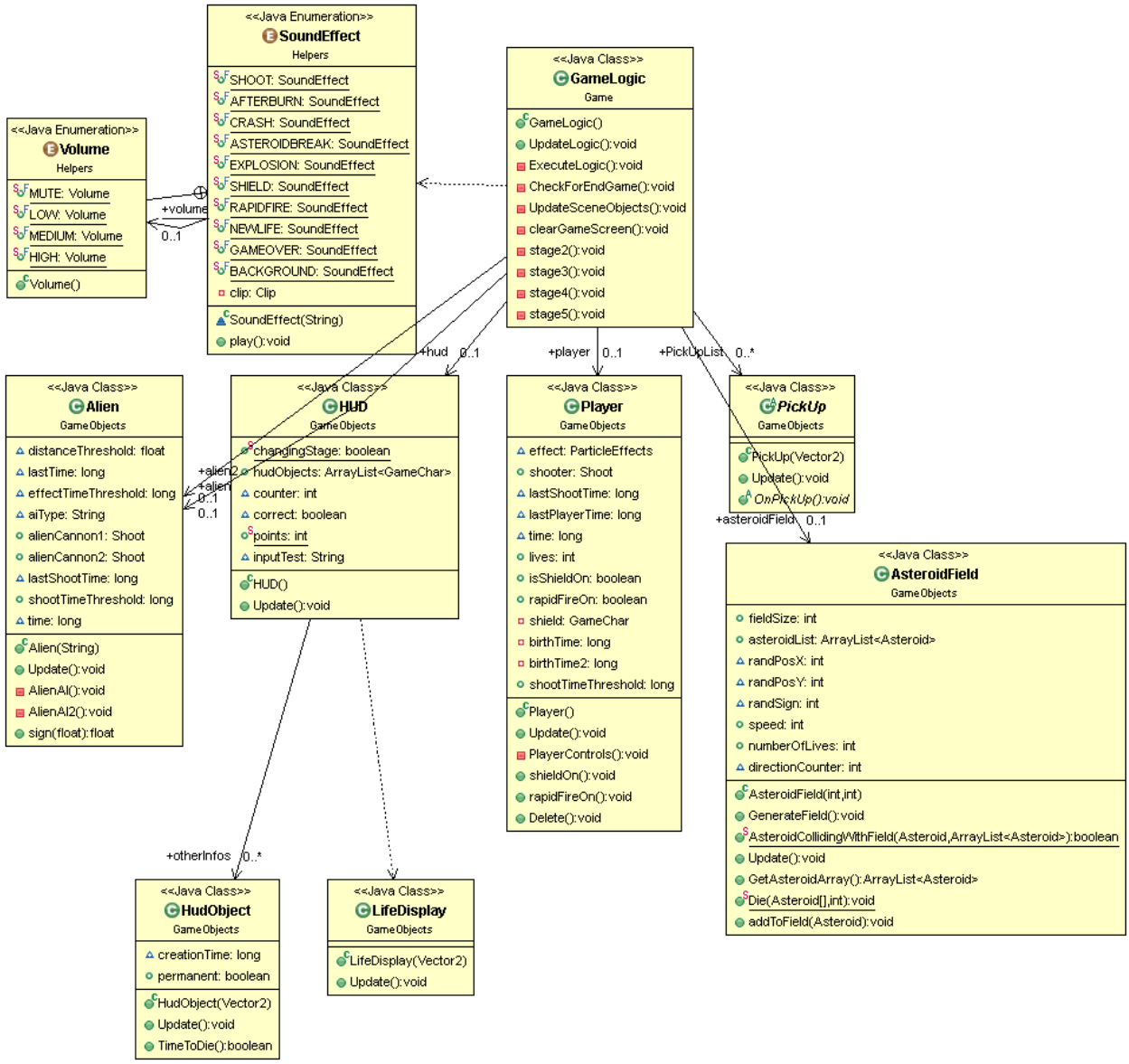


Figure 5- Game Logic

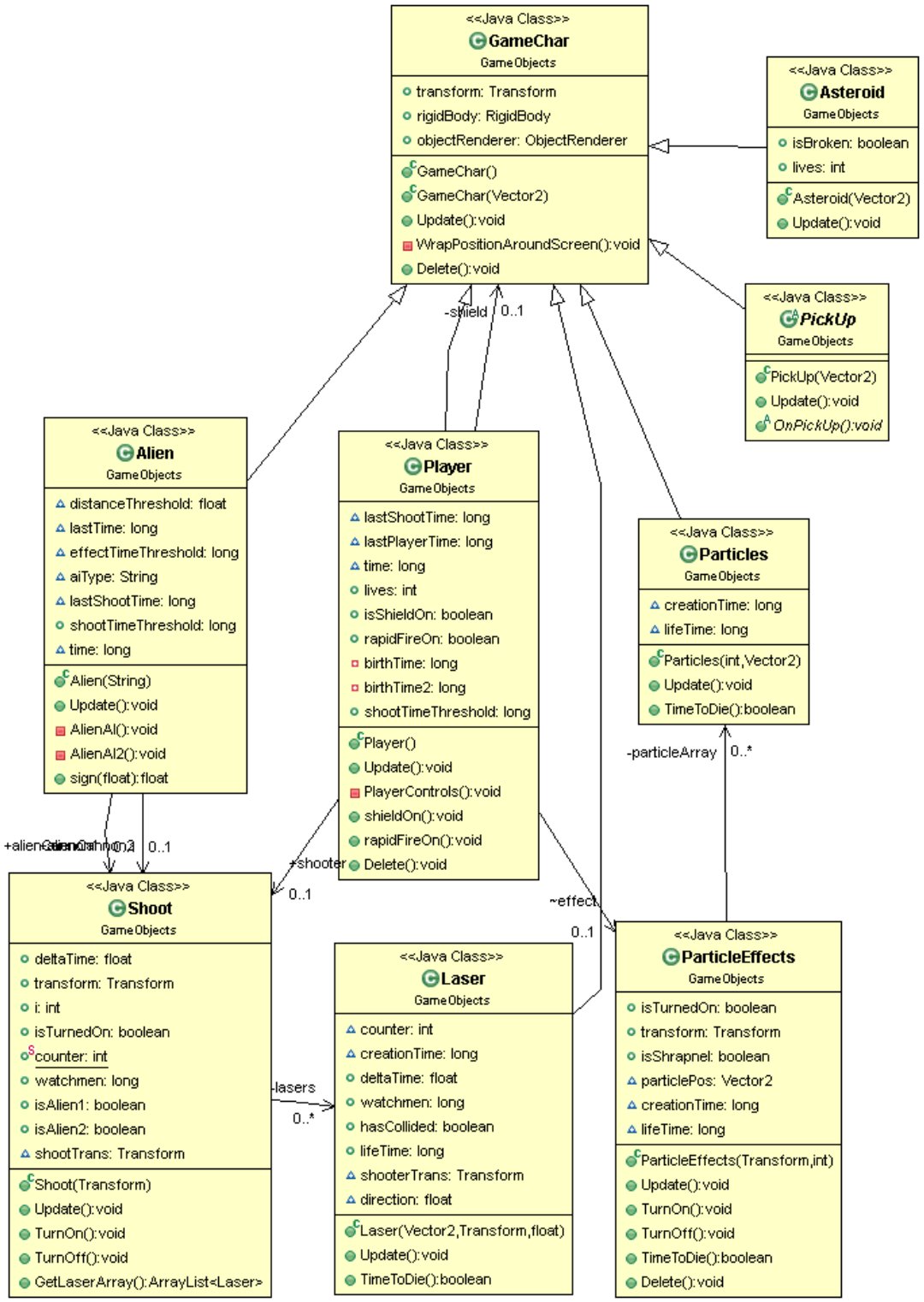


Figure 6- Game Char

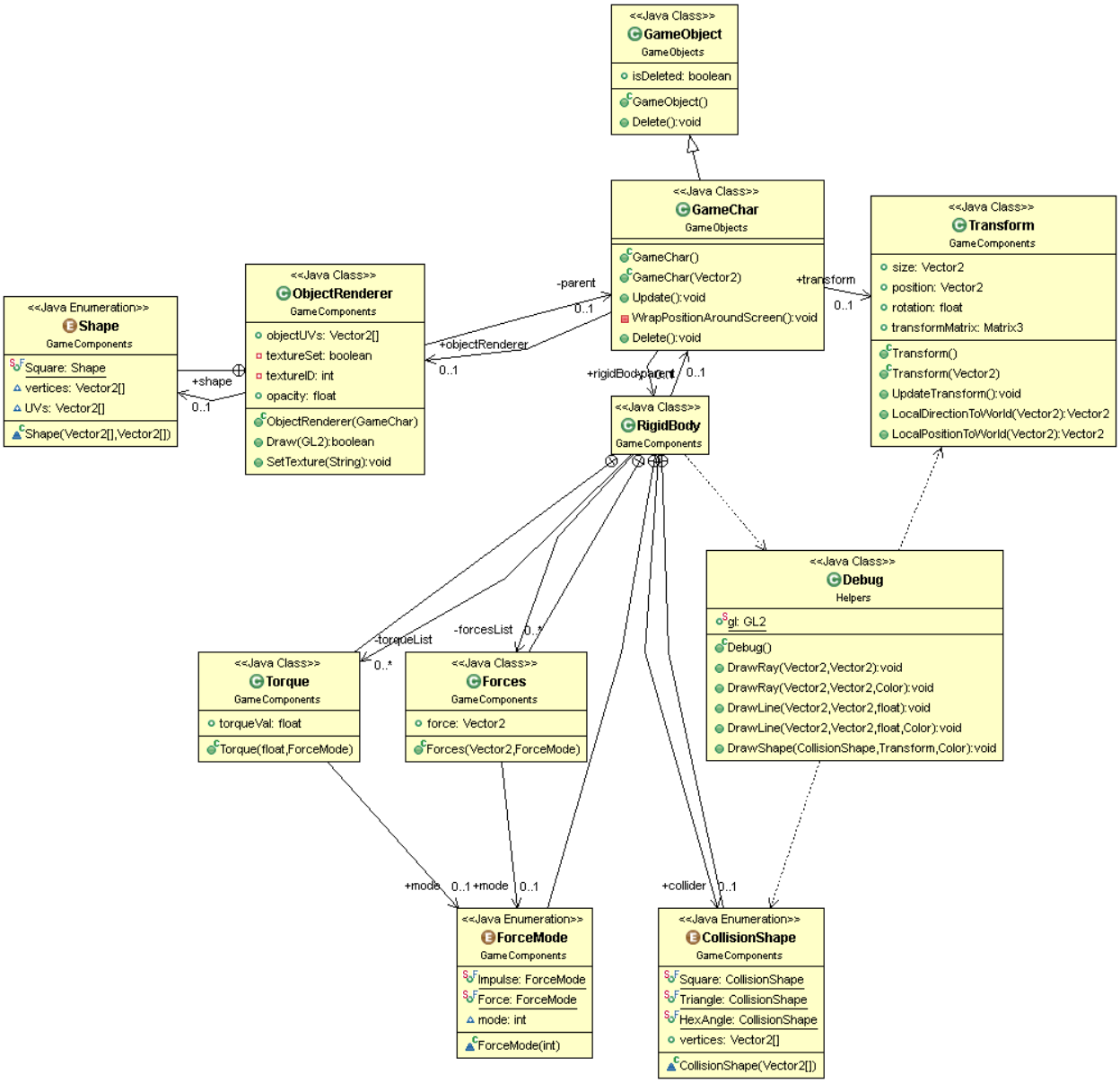


Figure 7- Game Object

Figure 8- Player

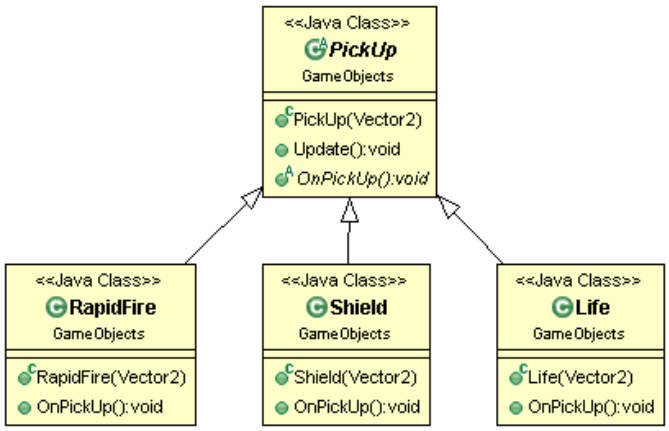


Figure 9- Pickup

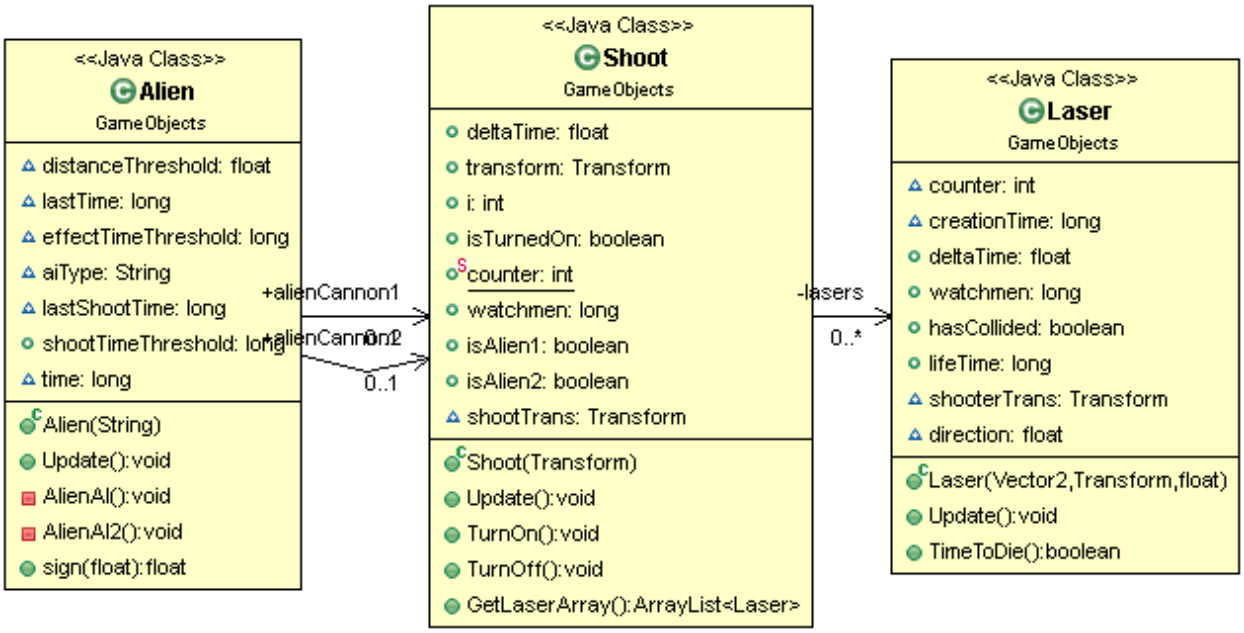


Figure 10- Alien

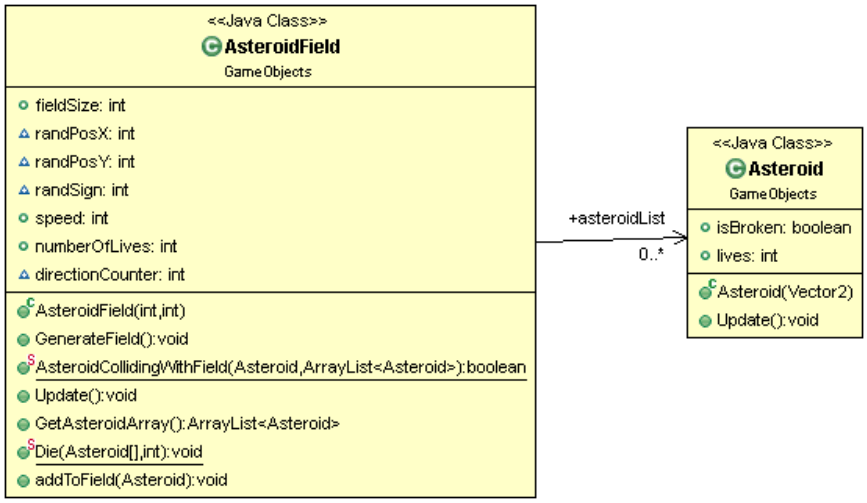


Figure 11- Asteroid Field

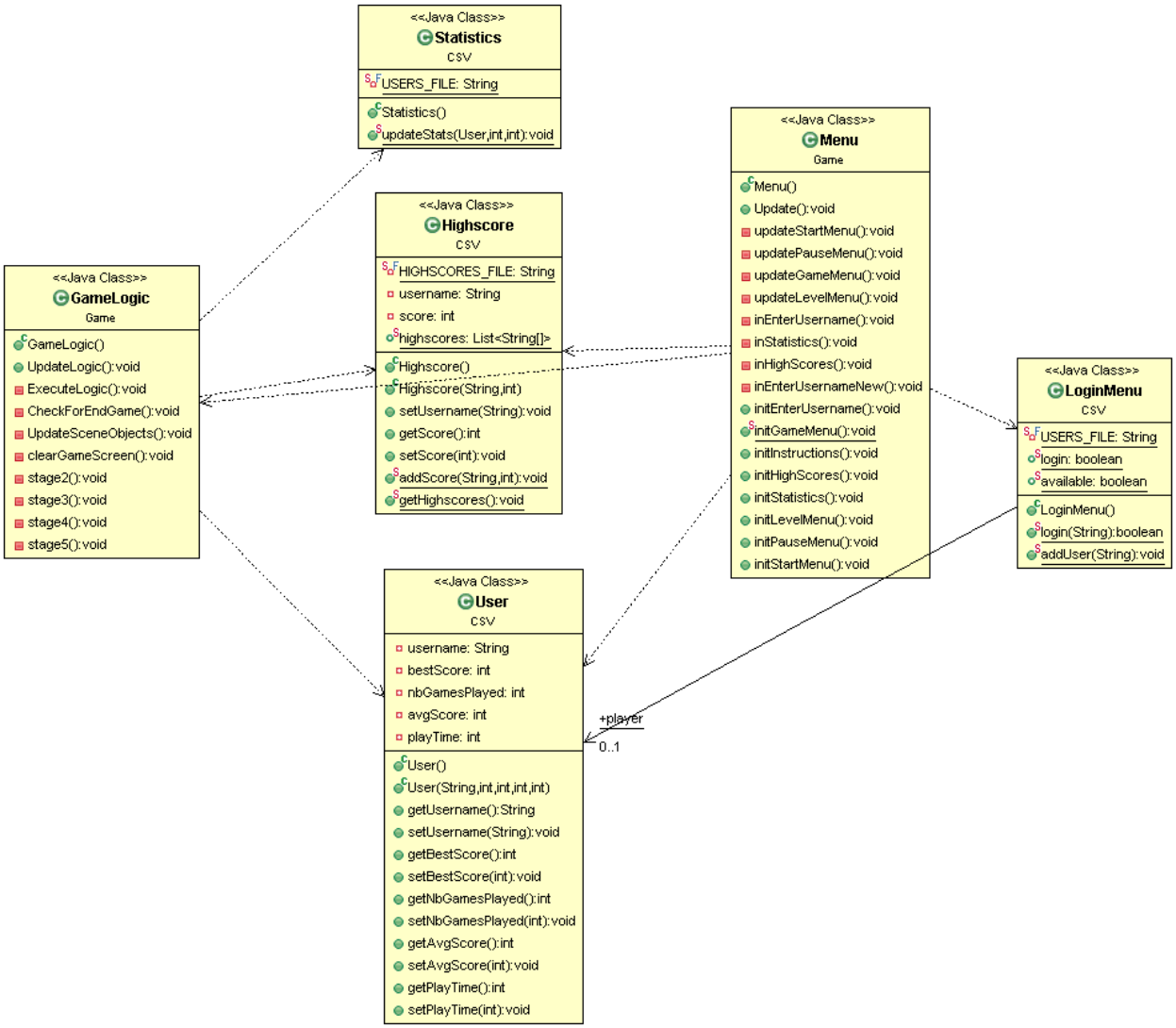


Figure 12- CSV

2.3 SEQUENCE DIAGRAM

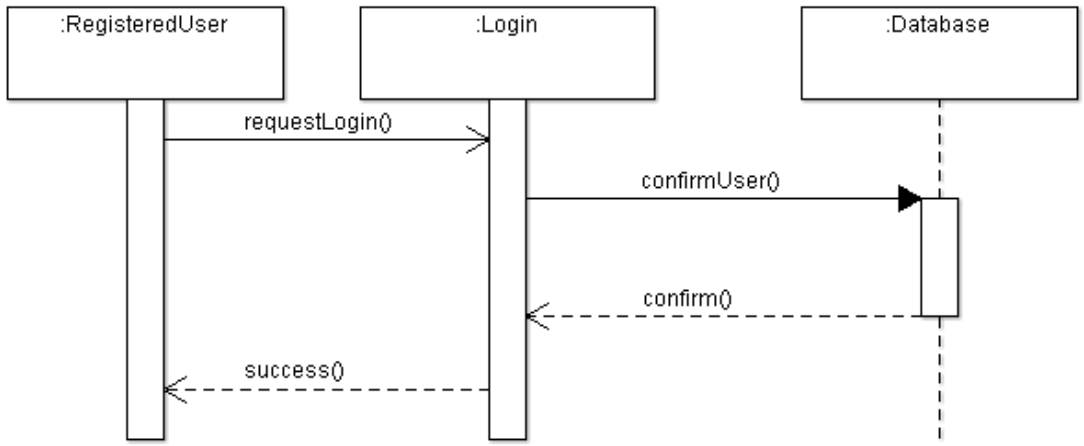


Figure 2.3.1-Login Sequence Diagram

2.4 ACTIVITY DIAGRAM

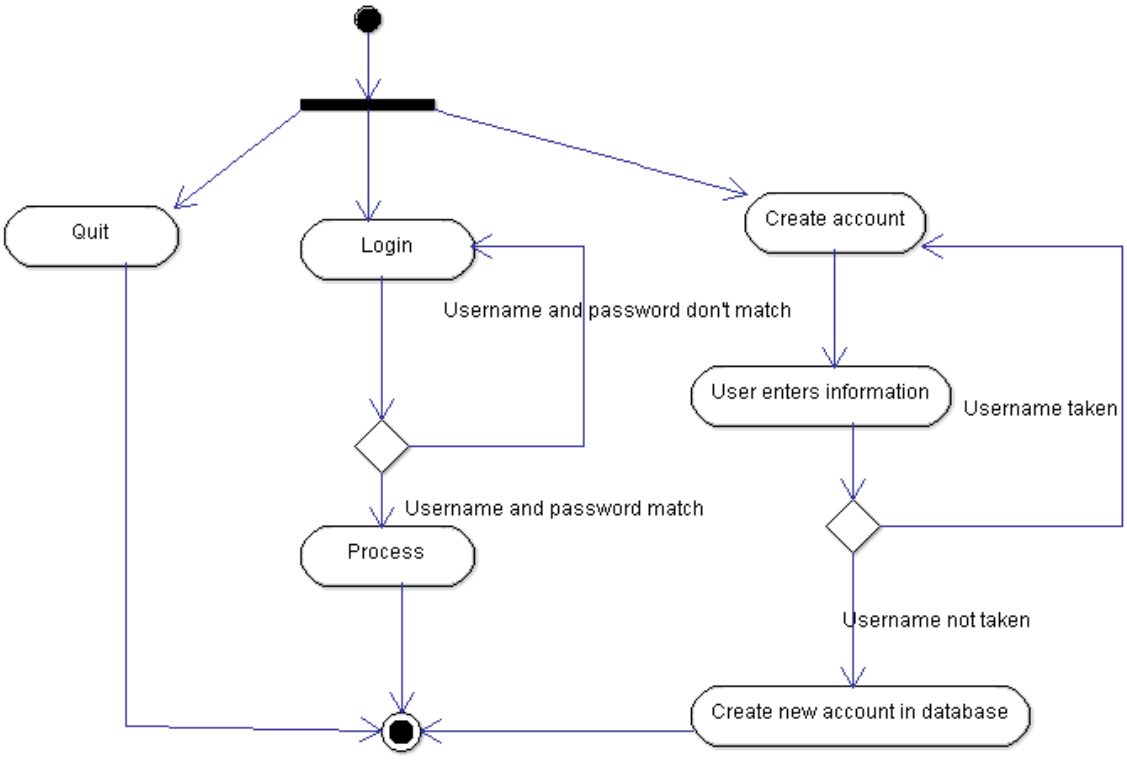


Figure 13-Registration/Login Menu

3. SOFTWARE SUBSYSTEMS / MODULES

3.1 GAME OBJECTS

<table>
<tr><td></td><td>Game Objects:</td></tr>
<tr><td>GameObject</td><td>Abstract class that acts as the model for all active objects in the program(asteroids, player, spaceship, Menu items, etc.). Contains basic shared attributes by all Game objects</td></tr>
<tr><td>GameChar</td><td>Abstract Class that extends GameObject. Base object for all Game character units. Shares Render, transform, and Rigid Body attributes</td></tr>
<tr><td>Player</td><td>Extends GameChar. Specific Player attributes: Render, lives, damages…</td></tr>
<tr><td>Asteroid</td><td>Extends GameChar. Specific Asteroid attributes: Render, size.</td></tr>
<tr><td>Alien</td><td>Extends GameChar. Specific Alien attributes: AIienAI, one of the specific type :</td></tr>
<tr><td>Laser</td><td>Extends GameChar. Specific Bullet Attributes: Render</td></tr>
<tr><td>PickUp</td><td>Extends GameChar. Holds a specific type picked up in an enum Type :

<table><tr><td>Life</td><td>Adds life to player when picked up. Characterized by coords, duration, upgrade amount.</td></tr><tr><td>Shield</td><td>Shields player for a duration when picked up. Characterized by shield strength, shield type, duration.</td></tr><tr><td>RapidFire</td><td>Increases player fire rate when picked up. Characterized by duration, rate increase.</td></tr></table></td></tr>
</table>

3.2 OBJECT COMPONENTS

Transform	A class holding game object space transform information (Ex: Vector2 position, size, position) ...
RigidBody	A class for keeping track and updating the physic states of game object (updating velocity, acceleration, collisions…)
ObjectRenderer	A class holding all specific information about the object rendering attribute such as textureInfos, its visibility. Will be interpreted by the Render() function of the renderer class.

3.3 MAIN GAME CLASSES

	Main game Classes
Renderer	The main rendering class: Implements functions for creating and updating the OpenGL context (Drawing arrays of object, resize window...). An Update() function will be called at each frame.
GameLogic	A class for holding and updating the state of the game according to the game rules. Here we will create and update our Game Objects. It is composed of a MenuLogic, and a GamePlayLogic. UpdateLogic() updates the current required logic (menu or gameplay)
MenuLogic	A class to provide Specific executions of the menu logic. Holds the current Menu element selected, decides what menu elements to show etc ...
GamePlayLogic	A class that will instantiate GameChars and manage them accordingly to the game rules…
MainGame	And here is our Main() function! It will instantiate Renderer and GameLogic singletons, and will initiate the game loop, where we will list all the stuff our main singletons will execute (Ex: renderer.Update(), gameLogic.Update()) until termination request.

RENDERING:

Our rendering pipeline is fairly straightforward: The Render class provides a method to initialize our graphic (OpenGL) context. Then each visible GameObject (namely, child class GameChar, or GuiElement) will hold a component called ObjectRenderer, which will specify and hold the rendering data and state of an object. These ObjectRender instances will be added to the Renderer renderList. At each frame, the Update() method of the Renderer will be called in the main loop and will linearly render all the elements in this list. On termination request, the renderer will CleanUp() every elements remainding in the list.

PHYSICS AND RIGIDBODY:

Each GameObject that has a RigidBody component (namely all the GameChars) will have its physic state stored in the RigidBody. The physic state means the current Velocity, Acceleration, and list of forces applied to the object. Much like the ObjectRender component and his Draw method, the RigidBody has and UpdateState(Transform transform) method, which will compute and replace the state with a new one at each call based on an iterative integration method and will modify the transform passed in accordingly. In general, the transform passed in will be the transform of the GameChar itself. The call to update rigidbody physic states will be made by the gameplay logic at each frame, on all the game characters currently instantiated.

GAMELOGIC

At initialization, the game logic instantiates a menu logic. The menu logic implements the menu program flow and executes until the play button is selected. It then instantiates a Gameplay logic. The gameplay logic executes the game rules, it generates game characters (Asteroids, Aliens, Player, Power Ups) and

update their RigidBody/object renderer according to the game rules (ex: when the “turn left” key is pressed, push the according force to the forces list of the RigidBody. When an asteroids is destroyed, change its renderer…).

3.4 HELPER CLASSES

	“Helper” Classes:
Helper	Stores all the generic and quite random but handy functions we will hack during our implementation trip. (Ex: “DecimalToString(…))
Maths	Stores all the Maths related functions we could happen to need to implement ourselves (Ex: FastSqrFTW()).
Vec2	A 2D Vector class implementing 2D Vectors stuff (Ex: SqrMagnitude(), Dot/CrossProduct(…), Normalize())
Mat2	A 2D Matrix class implementing 2D Matrices stuff (Ex: Mult(…), LocalToWorld(), WorldToLocal(), Rotation(…)).
Sounds	Contains all the different sounds which are used in the game and can play them with different preset volume

4. ANALYSIS

In our System Requirement Specification document we listed many important quality requirements which were divided in different subsections. They were divided between exactly 6 subsections: performance, usability, reliability and availability, safety, maintainability and extensibility.

In terms of performance, the player needs to be able to play in real-time. There must be a minimum delay between user input and game reaction and also a general delay between two game frames. It is therefore required that the minimum frame rate is 60 fps on current mid-range hardware. Also, query and reporting times should be less than 20 seconds on current mid-range hardware. Calculations to keep score of the player up-to-date should be executed instantaneously. Finally, a high score table will appear in less than 5 seconds after the game is over. In terms of usability, it is important to mention that English will be the language used for the game. To facilitate the user's game, shortcuts on the keyboard will be implemented. Also, the game interface will be designed to appeal to users of all ages and you can learn how to play while playing the game. For reliability and availability, we will make sure that all high scores are maintained in the database. If an exception that forces the program to terminate occurs, it is necessary to save the current state of the session before termination. In terms of safety, the game will be paused if the user clicks outside the system's window. For maintainability reasons, the code will be fully documented and modular in order to permit future modifications. Finally, in terms of extensibility requirements, the game will be extensible and allow easy changes or improvements in the future.

The renderer of our software is in a main loop which will allow the high performance of our software. All the images will be displayed in 60 frames per second. However, the desired frame rate might chance during the implementation of our system. Also, we will be implementing a CSV database to make it easy to implement but also improve the security of the application. In order to satisfy our security requirement, all the player information (statistics and high scores) will be recorded in an extensible file

only accessible by the administrator. This will prevent any users to access and alter the data. The database will only be accessible to the administrator. Finally, the gameObject structure will allow for simple improvement of the game system. Since levels themselves are GameObjects, they can be added relatively easily. In addition to this, if new elements or interfaces need to be added to the game, a new GameObject can be created. This object will inherit GameObject qualities and be able to interact through already implemented methods with other GameObjects. With the implementation of this structure, anything can be added easily so long as it conforms to the GameObject structure, such as menus, asteroids, aliens, spaceships and levels. However, anything requiring major change in the gameplay logic will be taking longer to implement

5. DESIGN RATIONALE

Our Software architecture offers a fairly straightforward implementation of the main elements of the game system. Our overall strategy consists in establishing a strict class hierarchy where main objects such as Transform, RigidBody and Renderer do not need to interact directly with each other and only need to be called on by higher classes in order for their attributes to be used. If the structure is to be abstracted into a tree, this would mean that there is always a higher abstract class as the root node. Subsequently, the other functions would correspond to the lower nodes according to hierarchy.

This in turns allows for simple expansion of any of the game's features by creating objects on the desired level of hierarchy. For example, if a new level or game mode needs to be implemented, we can simply create a new instance of these objects and plug in whatever values or attributes they require. Returning to the tree structure, this translates to being able to easily add branches at whatever level of the tree.

We decided to create our own graphic layer from scratch (using only function call from the openGL specifications) to achieve as highest performance as possible, and to allow us stability and flexibility in our architecture design, rather than using the cumbersome java graphic libraries.

The highscore and player databases will be implemented with a simple CSV file. This allows for straightforward reading and writing of information to and from the file. As for issues of security, the CSV file will be encrypted with an algorithm that must still be developed.

6. WORKLOAD BREAKDOWN

Names	Responsibility
Vincent Petrella	Rendering + physics
Damien Girard	Game Logic (for game)
Xavier Sit	Sounds and game logic(for game)
Thuy-anh Le	Game logic for menu
Siddhart Gandhi	Database integration (stats+ high scores)
Genevieve Nantel	Game logic for aliens (AI)

During the implementation phase, everyone will be responsible for coding. While some are more proficient and experienced coders than others, some will be assigned to program more challenging parts than others. The people assigned with the least challenging parts will also be focussing on other important aspects of the project such as documentation.

Main Documents:

- Prototype demonstration and feature/requirements summary document
- Test documents
- Final directory:
    - User Manual file
    - Implementation notes file
    - Document recording all meeting minutes

Moreover, we will be writing an implementation/testing notes document that is maintained by individuals to describe the current state of their work during testing. Throughout the whole implementation, we will be encouraged to test ''early'' and ''often''. In terms of early, we want to start testing at the beginning of the implementation phase by testing each unit separately. This goes hand in hand with testing often meaning that members will be asked to test their parts as soon as they are implemented rather than waiting for the completion of the units they are part of. This will prevent experiencing major errors during the official testing phase. However, testing will not be done by the programmer in question but by another member in the team since the programmer is too familiar with

his/her own code. We will constantly updating a document recording our tests. There will be 3 majors parts involved in the test process; unit testing, integration testing and system testing. Also, it is important that the entire team knows where files are to be stored and from where they can be retrieved. In our case, we will be using GitHub. Finally, as a side note, we know that the implementation language is a factor that weights high. While some are more skilled with certain languages, we had to come to an agreement. According to the common degree of expertise of the team members, java was the chosen language that will be used for implementing the asteroid game.

February 2013

MONDAY	TUESDAY	WEDNESDAY	THURSDAY	FRIDAY	SATURDAY	SUNDAY
18	19	20 Architecture Design Review Meeting	21	22	23 Group meeting 2h00 -5h00	24
25	26	27	28 SAD final version done			

March2013

LUNDI	MARDI	MERCREDI	JEUDI	VENDREDI	SAMEDI	DIMANCHE
				1 Phase 2 due: Software Architecture Document 11h59 pm (20%)	2 M-B Start Implementation Phase	3 M-B
4 M-B	5 M-B	6 M-B	7 M-B	8 M-B	9	10
11	12	13	14	15 Finish implementation and start integration testing	16	17
18	19	20	21	22	23	24
25	26	27 Prototype Demonstration 5%	28	29	30	31

April 2013

LUNDI	MARDI	MERCREDI	JEUDI	VENDREDI	SAMEDI	DIMANCHE
1	2	3	4	5	6	7
8	9	10 Project Presentation (15%)	11	12	13	14
15	16Final deliverables and peer evaluation (40%)					