Joshua Berman 320856
Terry Bugai 364017

ELEN3009: Software Development 2

Project 2013: Galaxian Game

This document contains:

Project Report

Test Report

User Manual

ELEN3009: Software Development II Project – Galaxian Game

Joshua Berman (320856)
Terry Bugai (364017)
September 30, 2013

School of Electrical & Information Engineering, University of the Witwatersrand, Private Bag 3, 2050, Johannesburg, South Africa

Abstract

The goal of this project is to design a functional Galaxian game for the PC using C++ and the SFML 2.1 library. The domain is broken down into three different parts, each representing a role that is implemented in its own class and sub-classess. The logic layer consists of the manipulation of all the visible game objects and their implementation classes. An input/output class is used to interface with the data layer consisting of a text file containing the game high-score. The visual graphics class is used to draw game objects to the visual interface of the game. An object-orientated solution using these classes creates an easily adaptable framework. The main criticism of this solution is that an inheritance-based design is only used in the graphical layer. Future improvements to the game are additional spaceship lives, a moving background, extra levels and more powerUps. A smooth running adaptation of the original Galaxian game is created that functions with two minor bugs; one being that the game slows down when a collision occurs and the other which causes the brains to jump when changing direction.

Key words: Software Project, Galaxian, C++, SFML 2.1, Object Orientation

1. Introduction

Brain-axian is an adaptation of the original space shooter game, Galaxian. This version is the story of a space cadet who encounters a fleet of enemy Brain ships while flying through space in his spaceship. The cadet must destroy all enemy Brains in order to continue on his voyage. The only way to do this is to fire neurons at the Brains; the very substance by which they are made. This project outlines an overview of the development and testing of this game. An object-orientated solution in C++ is explored through abstracting the different parts of the game into smaller, workable modules. The game and the design process are critiqued and improvements to the project are discussed.

2. Project Framework

2.1. The Original Game

The original version of Galaxian is an arcade game released in 1979 by Namco; Galaxian expanded on the concept created by *Space Invaders* [1]. As in the earlier game, Galaxian features a horde of attacking aliens firing on the player's ship. This version is a modified and more simplified version of the original game however; the overall gameplay mirrors that of the original 1979 Namco version.

2.2. Categorisation of Features

The game is broken down into different blocks of functionality, that is, basic functionality, minor enhancements and major enhancements. The basic features which are required include: the existence of a SpaceShip and its ability to fire Projectiles; Brain-Aliens and their respective Neuron-projectiles; the Ship is required to moves left, right, up and down on command; the Brains are required to move left and right in formation firing neurons and individual Brains are required to dive bomb the player by moving vertically down the screen; a Brain is destroyed when it is hit by a projectile of the SpaceShip and the SpaceShip is destroyed when it is hit by a neuron of a Brain; both the Brain and SpaceShip are destroyed when they collide; the game ends if a collision occurs between the SpaceShip and another object or when all the Brains are destroyed.

Minor features included in this version of Galaxian are: the visually aesthetic graphics; a scoring system that saves as well as displays high scores and evolved Brains which dive bomb the SpaceShip while firing upon it.

The major features included in this version are: the Brain-Freeze aliens who dive bomb the player in *swooping arcs* and then rejoin formation, as well as more powerful weapons (capable of destroying more than one Brain at a time) which become available as the player collects the PowerUp sphere floating across the screen.

Additional features which increase the overall appeal of the game include: a shield which protects the SpaceShip from Brains and their neurons and a wraparound powerUp which allows the SpaceShip to move past the left or right end of the screen and reappear on the opposite end. The wraparound feature allows the ship to tactically evade neurons. An additional feature is that only one bullet may be fired by the SpaceShip at a time as a delay is set between shots. This limits the firing rate and slows down gameplay in order to make it more challenging.

2.3. Assumptions and Constraints

The game is to be programmed in standard ANSI/ISO C++. The SFML 2.1 graphics library is to be used, but no additional libraries built on top of SFML are allowed. Loaded images are permitted in place of primitive SFML drawn objects. The game must run on the Windows platform and should have a good object-oriented design [2].

2.4. Success Criteria

A complete solution requires a user friendly Galaxian game that is both fun and easy to play. The game is to have all basic functionality as well as two major features and two minor features; this is required to achieve an "exellent" rating. A complete solution has been built up in small stages. The first stage includes the basic functionality and one minor feature. The second stage includes one major feature to satisfy an "exellent" rating. Seeing that the basic functionality was buggy and dive bombing Brains did not collide correctly with the SpaceShip, the next step was to improve basic functionality as well as add one major and one minor feature. The graphics of Galaxian is not the main purpose of the game however, an easily adaptable framework is important. In order to implement an adaptable framework, the idea of basic inheritance (with the purpose of role modelling) is to be explored and the display and logic levels should not be mixed. Sufficient unit testing is required to ensure a bug-free solution.

3. System Design Aspects

The conceptual domain of the game is broken down into thirteen distinct parts, with each part being modelled by its own class in the games implementation. These thirteen distinct game features include: the central Galaxian class which manages all major events of the game; a MainMenu class which displays the menu of the game and sets its 'clickable' regions; a SplashScreen class which solely displays the splash screen and handles its mouse events; a SpaceShip class which creates a controllable ship; a Brain class which creates all Brain objects and sets the functionality of all the alien Brains; Projectile classes for both the SpaceShip and Brains which create bullets for each respectively; a PowerUp class which controls the functionality of the PowerUp weapons as well as the shield and wrapAround function; a StopWatch class which adds advanced timing for processes; a Scoring class and a VisibleGameObject class which controls the manipulations of objects on the screen.

The player's goal is to shoot every Galaxian on the screen and reach a new high score; in order to do this he needs to avoid getting killed by the bullets of the Galaxians as well as the Galaxians themselves.

3.1. Main Menu And Splash Screen

Like many games now days, Brain-axian contains a startup screen and a main menu; each requiring their own class. These classes are completely isolated and do not use or lend any functions to any other classes. This gives the programmer the option of adding additional features without affecting the rest of the program. An additional feature could be 'controller-setup' which allows the user to choses the control keys.

3.2. Brain-axian

The layout of the displayed sprites and sprite textures are positioned on an x-y plane using standard coordinates. The main screen is set to 800x800 pixels and each pixel represents an x-y coordinate. The SFML library contains some easy to use functions, such as 'setPosition()' which allows the programmer to use these x-y coordinates to represent the x-y plane.

3.3. SpaceShip and Brains

In order to display an object on the screen one can use the Sprite class, create a sprite object and use its member functions. Sprite objects themselves are obviously not displayed but rather 'Textures' are displayed on the x-y plane.

In this version of Galaxian, textures are created and images loaded into them. These images make up that of the SpaceShip, the Brains, PowerUp sphere and all projectiles.

3.4. Visible Game Objects

The VisibleGameObject class is created in order to manipulate the game objects that are to be displayed on the screen. The class is made up of functions that can be called via objects of its inheritor classes such as the ones listed in Figure 1 below.

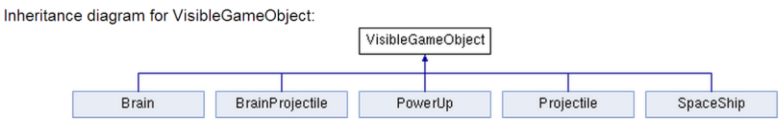


Figure 1: Inheritance Diagram, Doxygen

These functions control the movements and updates of the objects and Sprite objects such as setting the position, advancing the formation and setting imaginary boundries of the textures displayed. In this way, objects can 'collide' when their *x-y* points overlap and an event can be triggered such as a texture overlay or a position reset.

Boundaries are imaginary squares around sprites that enlarge their footprint. A sprite is passed to a member function with its *x-y* coordinates. The upper and lower boundries are set by subtracting and adding *y* values. The right and left side boundries are set by adding and subtracting *x* values.
For example: *_sprite.setBoundary(x,y)*

where *x* and *y* both equal 100. The sprite's upper and lower boundaries would be 80 and 120 and the same for the right and left boundaries. Here the boundary value is set at 40 pixels wide.

Since objects cannot be destroyed, two options are present when dealing with the deletion of game objects. Firstly a container can be created and these objects loaded into the container. Upon 'deletion' of the object, it is simply removed. The second option is that the position of the sprite is set beyond the boundries of the screen or main window; this rendered the object invisible to the user. The former seems to be a better option as one would prefer not to have objects drawn unnecessarily. The latter however, is far simpler as it only requires a boolean check for the object to be 'Alive' or 'Dead' and hence its position redrawn. The second method saves time and hence is used throughout the game to display objects.

Most of the game objects are created via their own class which inherits from *VisibleGameObjects* class. The SpaceShip class is one example which contains no member functions of its own but rather uses all that of *VisibleGameObjects* functions. The Projectiles and Brains work in a similar manner however, the Brain class has some of its own member functions that are specific to it and are not required by other classes; an example of this is *'formation()'* which sets the Brain layout. In this way display logic is isolated to one region of the project and can be used on multiple objects from various classes.

Major features such as the implementation of the swooping arc function can be specific to the Brain objects but if needed by the PowerUp sphere it would not have access. For this reason the *'arc()'* and *'dive()'* functions are listed in VisibleGameObjects class along with the other manipulation functions instead of in the Brain class itself. The arc function uses an overly complicated equation to update the x-y coordiantes. The equation was devised using the priciples of exponential functions and the properties of square roots:

```
x=sqrt(y)*exp(1/2)*sqrt(900-1.3*y)
```

where x is increased to a point and then quickly decreased and

```
y=y+(_elapsedTimeSinceCreated)/20000
```

where 20000 causes a delay in the movement.

3.5. Power Up

The PowerUp class is added to improve game weaponry as the user advances. A sphere is placed on the mainwindow at a radom position and slowly updates its position towards the lower end of the screen. This is done by equating the y value of the PowerUp sprite to the *rand()* function. If the SpaceShip's boundaries collide with that of the spheres, the sphere is no longer drawn and the *Select()* function is called. This function again uses the *rand()* function to generate a random number moded by 4 to select one of four PowerUps with a *switch-case*. The SpaceShip could theoreticaly never experience every PowerUp as the selection process is random. Once a PowerUp is selected, its designated boolean variable is set to 'true' and the graphical layer of the Galaxian class sets up the screen accordingly. This could be either changing the texture of the projectile or adding a 'Bubble-Shield' overlayed on the SpaceShip texture. The logic layer sets the effects of that PowerUp such that the ship cannot get destroyed while equiped with the shield. Simply put, the collision checks do not take place during that period of time.

If the projectile is changed to 'ShockTherapy', the boundry of the projectile is expanded and hence the collision area increases causing a larger area of damage. The 'Virus' works differently however, once a standard projectile collides with a Brain its position is no longer updated; rather is it set to an offscreen position as mentioned earlier. The virus projectile does not cease to update and continues to collide with every Brain in its path. This feature makes the virus the most powerfull weapon in the ships arsenal.

The wraparound effect is much simpler. When the function is implemented, the ship's *x* coordinate is changed from 800 to 0 as it reaches the end of the screen and vice versa.

3.6. Scoring

The input/output layer of Brain-axion consists of the scoring class. This gives the user the ability to track his/her overall advancement in the game as well as compare his/her score to the highest score achieved. This is done using the *fstream* library. The score saved in a textfile called 'Highscore.txt' is read in; this score is then compared to the user's current score. If the score in the text file is higher than the user's score, it remains the 'High Score'; else it is replaced with the user's current score. Currently, only one high score can be saved, but the class is easily adaptable and would allow multiple highscores in future versions.

This class also provides the text displays. It is used to notify the user upon receiving a PowerUp as well as the game state such as 'Game Over' and 'You Win'.

4. Implementation

As with many software implemented designs, the final solution is not without its flaws and shortcomings; this project is no different. In this section, the implemented design solution will be critiqued and its flaws will be identified along with possible solutions to correcting these problems.

From a programming practice or "code smells" standpoint, the implemented design is identified to have some areas that these "smells" or poor design aspects are present. There are some areas where code is duplicated, violating the DRY principle. There are one or two long functions that could be broken down into smaller segments of functionality. In some cases there is inappropriate intimacy between classes as well as variables which are available to all classes.

Suffitient 'Structs' are not put in place where it would seem neccesary. This is a result of unplanned features. These 'structs' could be used in the setup of the formation of the Brains. In that way, if new levels are created a slight change to that struct would render an entire new level. Since levels are not required for this version, this is not entirely necessary but would add to the overall design structure.

The DRY principle is violated by the brainArmy() function in the galaxion class. The function should be setup in a way that allows all Brain objects to be passed in and setup accordingly (similar to that of the *crash(*) function) unlike the current setup where each Brain array is setup individualy. Instead of repeating the same body of code in each function, the code should be placed in it own function or class and simply called by the Brain object as required.

The *brainArmy()* function that creates the Brain layout, as well as sets their position and checks their status, is a long function that could be broken down into smaller segments of functionality. The status checks could be performed in its own function and simply called by the *brainArmy()* function when needed. The position setting of the Brains could also be placed in its own function. The "current event" function that checks for keyboard input from the user is also a long function. This could be improved by creating a separate function for each movement as well as a function which controls the wraparound feature.

Class invariants are maintained well throughout the program but the degree of information hiding or intimacy between classes is poor. Additionally, the use of global variables should be avoided since it allows access of these variables to all functions.

In many cases it is unavoidable to provide get and set functions for some data members of the class; an example of this would be the *getPosition()* function which would give the programmer the ability to change the position of the ship from outside the class. However in other cases a different approach to object communication can solve this problem. Intermediate classes, that facilitate object conversations, could be introduced which could hide the information of the interacting classes from one another.

The data, logic and display layers of the code are not completely separated; this causes difficulty when changing the means of display libraries. For example, if Allegro were to be used instead of SFML, it would not be an easy task to adapt the classes.

The current code solution contains two minor bugs. The game slows down when the ship collides with a bullet or a Brain; this problem is most likely caused by an unforeseen logic error involving the collision. The second bug occurs when the Brains change direction while in motion; this causes them to jump before continuing in the new direction. This could be due to the update function where the reverse motion occurs. These two bugs could be further explored and corrected in future game versions.

An overall critique on the code implemented is that inheritance could be used more since it encourages greater game abstraction which reduces bad coding practices such as violating the DRY principle, having longer functions and object type data members. As a result the success criteria of implementing an inheritance-based, highly abstracted solution are only partially met. The data, logic and display layers of the code could be separated more in order to allow a simpler testing framework.

5. Improvements

Above and beyond the game's current functionality, many additional features and code improvements can still be implemented to further increase the quality of the final object-orientated solution.

From a code standpoint, changes could be made that would produce a sleeker, more efficient and soundly structured solution. The conceptual model could be abstracted even further to more accurately model the domain. To aid such abstraction, a solution is preferred which is solely based on inheritance. This would assist the program in adhering to the DRY principle.

Classes and class interactions could be improved by adding classes that model and facilitate these interactions. This would improve the degree of information hiding in the solution. One such class could be a collision class that manages the collisions between various game objects. The display class could consist of overloaded versions of the same draw function so, depending on what the draw functions are sent, depends on what is drawn. This makes the interface of the display class more intuitive. Increasingly sophisticated player tracking algorithms could be implemented to improve the ability of the enemy to find and shoot the SpaceShip, making the game more challenging overall.

There is also a vast amount of additional functionality that could be included into this game if time permitted. A file controller class could be made to store player names, as well as dates with the scores, making the high scores unique to the player that achieves them. Different levels could be made which become increasingly difficult as one advances in the game. An additional feature could be implemented where the enemy needs to be shot several times before it is destroyed; the colour of the enemy could change after each shot in order to tell the player its status. The game design could be extended to allow for a second player to join the action as well.

From a graphical point of view, the background of the game could change as a player advances in a game. The Galaxip could also be given flying animations when it moves around the screen.

Furthermore audio effects can be added to aid the game in captivating the user. The sound of an explosion when the ship is destroyed or a theme song would truly take Brain-axian to the next level.

6. Conclusion

The final solution implements an object-oriented approach using stand alone classes to model the game's domain and functionality. The responsibilities of the classes are outlined by the brief description of how the classes operate. The solution contains two major and two minor features and several additional features beyond the basic functionality as specified in the project framework. The features include Brain-Aliens which dive bomb the player in *swooping arcs* and then rejoin the formation, more advanced weapons which become available as the player advances through the game, good graphics, a scoring system, different coloured Brains, a shield which protects the SpaceShip and a wrap around power. The design process and resulting product is critiqued, with the major shortcoming being the undivided logic and graphic layer and that minor bugs still exists in the game's functionality. Improvements to the design, such as using better coding practise and aditional features are explored. In conclusion, an enjoyable smooth running Galaxian based game is created that functions with two minor flaws.

References

[1] M. Amis. *Invasion of the Space Invaders.* Hutchinson, 1982.

[2] Software Project Brief-2013, https://cle.wits.ac.za/portal/site/ELEN3009_2013_Software%20Development%20II%202013

Appendix A

SpaceShip

Figure 1: SpaceShip

Power Ups:

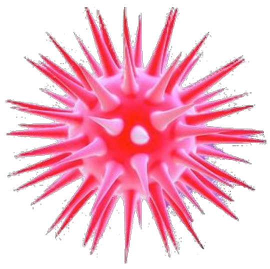


Figure 2: Virus

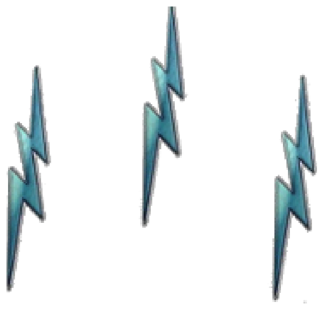


Figure 3: Shock-Therapy

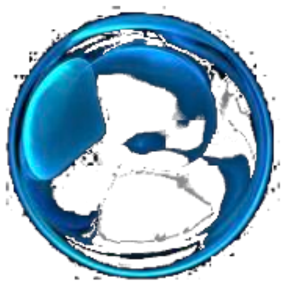


Figure 4: Bubble-Shield

Reference

[1] SpaceShip: http://www.freedigitalphotos.net/images/search.php?search=spaceship

[2] Virus: http://www.freedigitalphotos.net/images/search.php?search=virus&cat=

[3] Shock-Therapy: http://www.freedigitalphotos.net/images/Energy_and_Environme_g160.html

[4] Bubble:http://www.freedigitalphotos.net/images/search.php?search=blue+bubble&cat=&gid_search=&photogid=0&page=4

Appendix B

“Ideal” Class Hierarchy Diagram

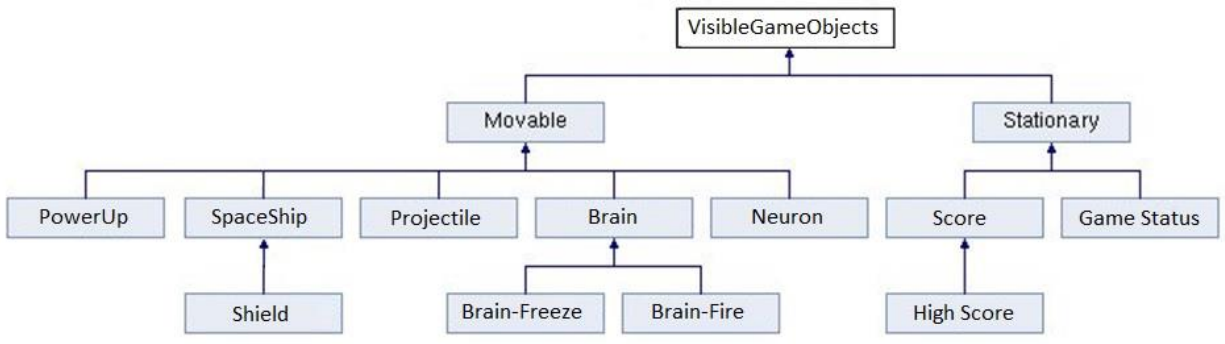


Figure 1: UML class diagram, “ideal layout”, Doxygen adaptation.

Appendix C

StartUp Screens

Figure[1]: Galaxian Splash Screen

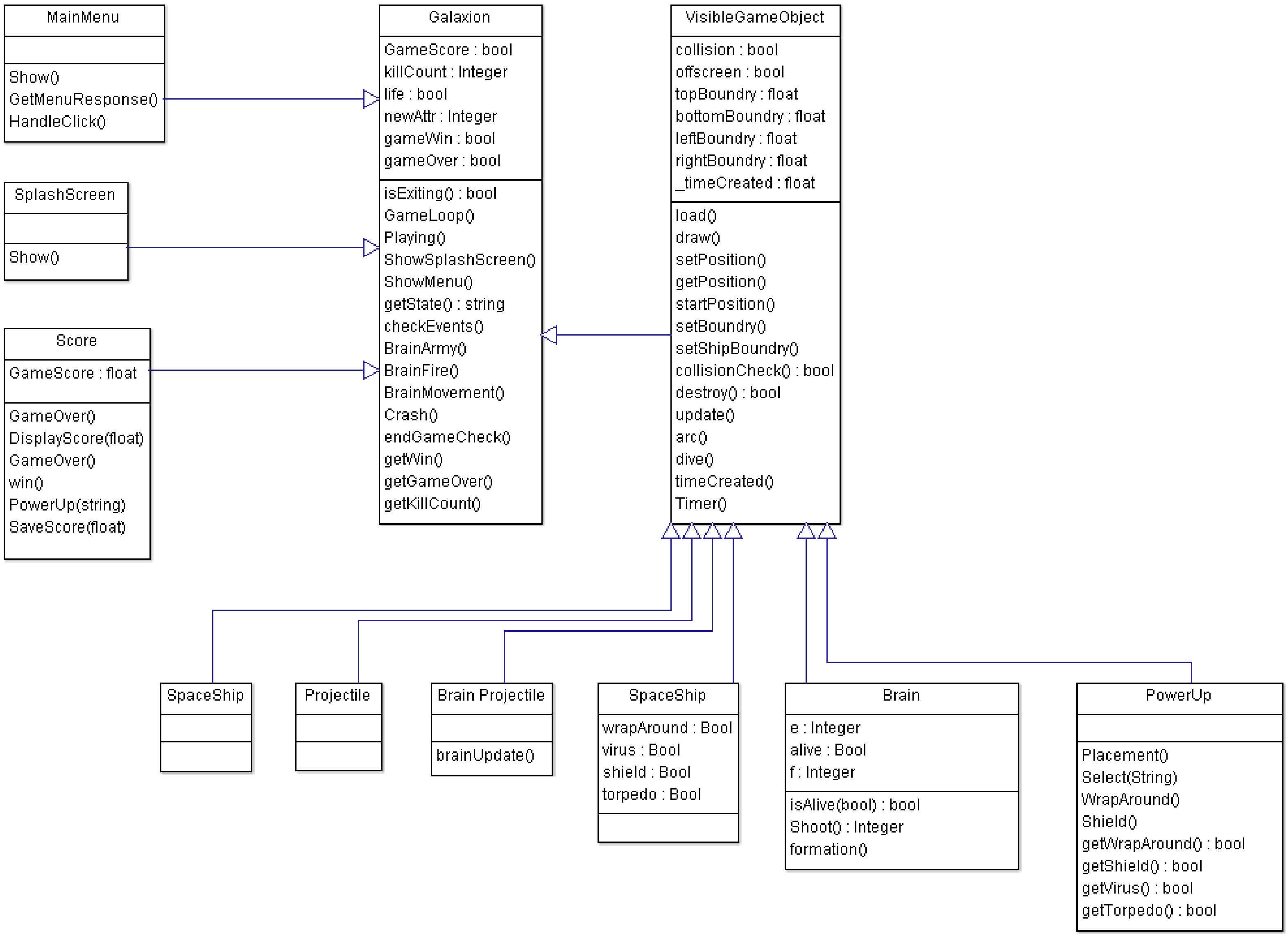
Figure[2]: Main Menu

References

[1] SplashScreen, http://mixxi-spacesynth-4ever.blogspot.com/2010/11/galaxion-around-you.html

Appendix D

Figure 1: Class Diagram , ArgoUML



Appendix E

Game Screen Caps

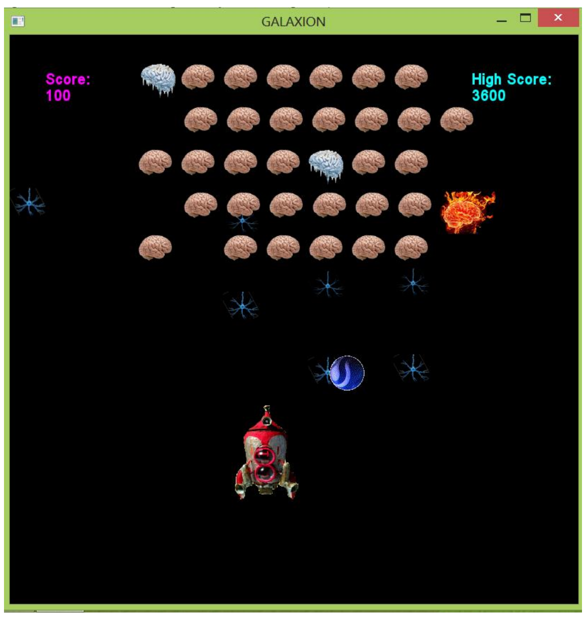


Figure 1: In game Screen

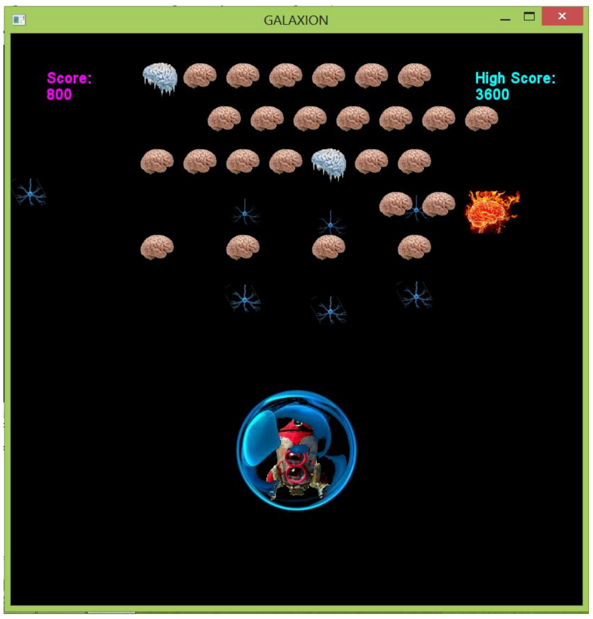


Figure 2: In Game PowerUp shield equipped

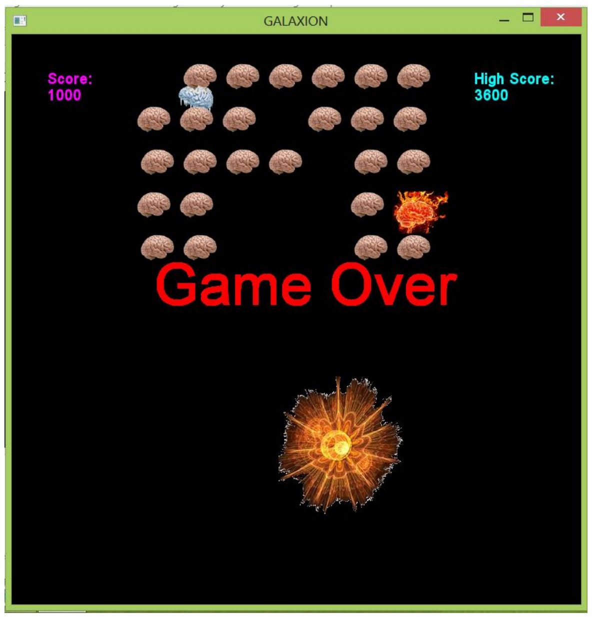


Figure 3: In game Screen collision, Game Over