

Software Workshop Team Java (06-08165) 2010/11, Dr. E. Thompson

Project Report: Space Runner

Team B3:
Daniel Cecil
Jere Ketonen
David Saunders
Michal Staniaszek

March 31, 2011

Contents

1	Introduction	1
1.1	Introduction to the Game	1
1.1.1	Uniqueness	1
1.2	Report	1
2	Requirements	3
2.1	Functional User Requirements	3
2.1.1	Attributes	5
2.2	Non-functional User Requirements	5
2.3	System Requirements	6
2.4	Future Requirements	7
3	Design	8
3.1	Release Plan	8
3.2	Class Design	8
3.3	Implementation	10
3.3.1	GUI and Controls	10
3.3.2	Menu	10
3.3.3	Controls	10
3.3.4	Game Logic	10
3.3.5	Game Entities	11
3.3.6	Network	12
3.3.7	Graphics	13
3.4	GUI Design	13
4	Validation and Testing	15
4.1	Testing	15
4.2	Validation	16
4.2.1	Functional	16
4.2.2	Non-functional	17
5	Project Management	19
5.1	Development Methods	19
5.1.1	Weekly Meetings	19
5.1.2	Incremental Stages	19
5.2	Planning	20
5.3	Testing	20
5.4	Time Management	20

6	Conclusions	22
6.1	Strengths	22
6.2	Limitations	22
6.3	Continuation	23
6.4	Future Projects	23
7	Appendices	25
7.1	Facebook	25

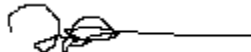
Work Breakdown

Coding	Daniel	Jere	David	Michal
Game Logic	5%	15%	0%	80%
Networking	0%	0%	0%	100%
Controls	0%	0%	95%	5%
Graphics	0%	100%	0%	0%
User Interface	0%	35%	55%	10%
Units	0%	20%	30%	50%
Pathing	0%	0%	10%	90%
Weapons	0%	10%	10%	80%
Spawns	95%	0%	0%	5%
Testing	25%	25%	25%	25%

Report	Daniel	Jere	David	Michal
Introduction	Chapter 1: 100%	0%	0%	0%
Requirements	Chapter 2: 85%	5%	5%	5%
Design	Chapter 3: 25%	0%	0%	75%
Validation and Testing	Chapter 4: 30%	0%	0%	70%
Management	Chapter 5: 0%	100%	0%	0%
Conclusions	Chapter 6: 100%	0%	0%	0%
Appendices	Chapter 7: 0%	0%	50%	50%

Daniel Cecil

Jere Ketonen



David Saunders

Michal Staniaszek

Abstract

In this project, we have created a top-down scrolling space shooter game with networked functionality. As far as we are aware, in this genre there are few examples of games which have capability for networked play, and therefore it could be considered that a networked game is rather novel in this particular area. We have attempted to make the game as modular as possible to allow for quick addition of new types of enemies, and have also used a method which means that both enemy ships and all projectiles can be made to follow paths which can use mathematical equations to specify the next position for a given moving object.

Chapter 1

Introduction

This aim of this report is to give a overview of the team Java work we did for 11 weeks on a Space Shooter game and how it progressed from an idea on paper to a functional program. It will be broken into sections covering important details on the design process, the way we worked and put the project together, the problems we faced, how we over came those problems and how we could improve the project further with more time and resources.

The first task when the team has first been introduced was to decide what type of game were to be produced, this is where the team discussed their own strengths and weaknesses so the plan could be outlined once the subject of the game had been chosen.

1.1 Introduction to the Game

The module outlined the core necessities for the project which was to produce a working multi-player game based on the Java platform which includes various aspects of sockets programming to facilitate the multi-player functionality. The game also had to have a single player mode.

With this brief the team started to brainstorm ideas in the first week of the project to see what interests each team member had. It was finally, and rather quickly, decided that the team's project would be to implement a side-scrolling space shooter game similar to the retro games from the Amiga/DOS times.

The team comprises of four students with a deadline for the project in 11 weeks time (end of 2nd semester) with a demonstration in the 10th week.

1.1.1 Uniqueness

It has been mentioned that the game is similar to those of Space Invaders and Asteroids so rather than to simply 'reinvent the wheel' the team had to ensure the game had a unique feature.

The specifications were drawn up by each member of the team and the unique point was highlighted. The game would be able to be multi-player through a network. The game would also feature extra additions such as power ups, boss fights, background music and different weapons which are also not present in current games of this type.

1.2 Report

This report contains the steps taken to produce the game, it's structure is outlined below.

- **Requirements:** This section outlines what is required from the game and features which must be implemented to be tested against. It is important this section is fully detailed before coding begins on the game to ensure everyone has a decent understanding of the game's functionality. Failure to properly understand the game can lead to delays in the game's production which could also induce bugs as well as work being undone to correct it.
- **Design:** This section details how the game is structured. In the first week the look of the game was roughly sketched and can be found in this section. The section also includes descriptions of how each class would work and interact with each other, again this is important to detail before coding commences to aid the understanding of the internal components of the game.
- **Validation and Testing:** After several components of the game were released they needed to be tested in order to confirm that each section of the game works as they should. The module introduced the JUnit package which was used to test the integrity of the game but other methods were needed to ensure the non-functional requirements were satisfied. The validation section details the steps taken to ensure the game was built correctly.
- **Software Engineering:** Any project has to have some management behind it and each team member had various levels of expertise; whether it be coding, design or management. This section details what approaches were taken to produce the project successfully. A time plan had to be organised to avoid the project being delayed.
- **Conclusions:** In this section, details explaining what was learnt from the project are described. If the project were to be run again several things might have to be changed which is part of the learning process. It also details what may have gone wrong in the project and where the difficulties lie.

Chapter 2

Requirements

2.1 Functional User Requirements

This section outlines the functional requirements which the system will be tested against. Functional requirements are what the system is expected to do and how to user interacts with the software. Each requirement is split into sub-requirements for ease of understanding and clarity.

1. The human player is able to control one's spaceship

This is a core requirement needed to be able to play the game successfully in at least a single player mode without any crashes or bugs.

- (a) The user is able to use either a mouse or the keyboard's arrow keys to move the spaceship.
- (b) The user's spaceship is able to move freely along the x and y coordinates but not leave the frame's boundaries.
- (c) The user will be able to hold more than one key for diagonal movement where the movement speed much be normalised.
- (d) The user's spaceship will be able to be represented graphically on the screen.

2. The human player will be able to shoot

The aim of the game is to enable the user to destroy enemy ships and therefore shooting is a must-have requirement of the game.

- (a) The user will be able to shoot by tapping or holding the spacebar or the mouse button (left click).
- (b) The user's spaceship will have a type of weapon to use, this can be changed during the game (if implemented - dependant on future requirement).
- (c) The user's shot will follow a set path forwards (negative y-coordinate movement).
- (d) Shots which leave the frame's boundary will be removed from the game state.

3. Enemies will be created to be destroyed by the user

This is a core functional requirement that needs to be implemented to enable the user to progress through the game by shooting down opposing units.

- (a) Enemies will be spawned at set locations on the screen.
- (b) Enemy units are to have a set health limit.
- (c) Enemies will be able to be shot by any user spaceship.
- (d) Enemies are to be distinguishable from friendly user spaceships by using different shapes or graphics.

- (e) The enemy unit's health will be decreased when a user's shot collides with the enemy.
 - (f) Enemy's will 'die' once all their health have been depleted.
4. **Enemies are to be able to return a level of resistance**
 This requirement is needed to make the game more interesting by introducing the possibility of a player 'death'
- (a) Enemies are able to return shots towards the human players with the use of different weapons.
 - (b) Enemies are able to move in certain paths (zig-zag, diagonal, straight, side-to-side).
 - (c) If the player collides with an enemy the player will 'die'
 - (d) 'Boss' enemies are to be introduced which fire more shots and have more health.
5. **The game is to run continuously with set events occurring at regular intervals**
 This requirement ensures the game runs smoothly and that something will always happen. For example, to stop the incidence of no more enemies being spawned (so the game is playable).
- (a) The game is to implement a Timer class.
 - (b) Enemies will always be spawned at set intervals during the game. These can be changed to be more or less frequent (if future requirement is implemented).
 - (c) Each tick of the timer will move enemies, player units (depending on user input) and projectiles.
 - (d) The game panel will be redrawn at every tick of the timer.
6. **The game will be able to be multiplayer across the network**
 This requirement is necessary to fulfil the assessment criteria allowing for a second human user to play in co-operative mode with each other against the computer enemies.
- (a) Each human user will be able to select whether they will act as the host or the client PC.
 - (b) Clients will be able to enter the Host's IP address to connect.
 - (c) The host's game will start immediately after selection with clients dropping into the game at a set spawn point.
 - (d) The game must be able to support at least two human players and a maximum of eight players (7 clients).
 - (e) All users must be connected to the same LAN network.
 - (f) All users must have similar game information on their screens (player, enemy and projectile positions).
 - (g) With each tick of the timer (requirement 5) each player's screen will be updated with network data from the host.
7. **The game must have a terminating clause**
 This requirement ensures that the game will end at some point.
- (a) Once a player's health has been depleted, that unit will 'die' and be removed from the game allowing other player's to carry on playing.
 - (b) If a player collides with an enemy unit they will also 'die'
 - (c) The player's score will be displayed on termination and if high enough will be recorded in a high scores table.
8. **The game will include a Graphical User Interface (GUI)**
 This allows all users to be able to start the necessary game type as well as actually play the game with the information displayed on the screen.

- (a) The game will run from a single frame.
- (b) Panels are to be added to the frame: Menu, Game, Gameover.
- (c) The Menu Panel is to feature buttons corresponding to various game types and options.
- (d) The Menu Panel is to be accessible from within the game (Esc key).
- (e) The Game is to be able to be paused using the 'P' key.
- (f) The window is to be resizable allowing for full-screen play.
- (g) The Game Panel is to feature a scrolling 'star-like' background (black with white stars).
- (h) The game objects (players, enemies and projectiles) are to be represented by shapes or sprites (graphics).

2.1.1 Attributes

Attribute	Requirement No.	Comment
Status	All	Approved - development started
Priority	1, 2, 3, 4abc, 5, 6, 7, 8abch - Mandatory. 4d, 8defg - Important	Mandatory requirements must be implemented.
Effort	All	Deadline for code: 22/03/11 (10 weeks). Estimated 40 person-weeks for first release.
Risk	All	Medium probability of risk occurring. Large impact if assessment is not complete. High risk with networking code due to lack of experience.
Target Release	1, 8a, 8h 2 3b-f 4, 5, 7 3a, 8b-g 6	v0.1 v0.2 v0.3 v0.4 v0.5 v0.6
Assigned To	All	4 x Team Members. Requirements and tasks to be distributed at weekly meetings.

2.2 Non-functional User Requirements

This section outlines the non-functional requirements. These requirements relate to the quality of the product and testing requires opinions and qualitative methods rather than quantitative feedback. The project has been split into different categories of requirement.

Usability

9. To provide a simple, easy to use system in order to play the game

- (a) A novice to the game should be able to gain understanding and play the game within 10 minutes of first playing.
- (b) Expert gamers should be able to grasp game concept within 1-2 minutes of playing.
- (c) The game should have a clean look and feel.

- (d) The user should feel in control of their spaceship with smooth movement and quick reactions.
- (e) The menu should have a standard, organised layout with minimal pages
- (f) The game should have a professional look

A user manual or help pages are not required due to the limitations on time for the assessment and the game itself is believed to be simple enough for most people to be able to understand.

Efficiency

10. The game should be constantly quick to respond

- (a) The game should run at a constant quick speed without any lag.
- (b) The user's input should have an almost instant effect on the game.
- (c) Network play should be stable for 95% of the time.

Dependability

- 11. The game should run first time, all of the time as single player or host.
- 12. Network clients should be able to drop-into the host's game within 5 seconds.

Environmental

- 13. The game should run on the platforms detailed in the below section (system requirements).

Development

- 14. The project should be developed using appropriate software engineering practises within the time frame for the assessment.
- 15. The project must be written in the Java programming language.

Operational

- 16. The multiplayer game must be able to run on any LAN with the IP addresses given.
- 17. High scores will remain stored in each copy of the game until they are overridden by a higher score.

2.3 System Requirements

This section details the requirements and physical devices needed to play the game successfully.

- 1. The game is to be written in the Java programming language.
- 2. All user PCs will need to follow the system requirements for Java 6
 - (a) Windows 7, Vista, XP, 2000, Server 2008, Server 2003. All 32 and 64-bit operating systems
 - (b) Mac OS X
 - (c) Most Linux distributions
 - (d) Solaris
- 3. Keyboard and Mouse are required (with Western layout)
- 4. For multiplayer, all users will need to be connected to a local area network (LAN) at least
- 5. Monitor and graphics card with at least a resolution of 800 x 600

2.4 Future Requirements

Requirements 1 to 17 are aimed at the first release which is as far as the project is aimed to go. However if the project is ahead of schedule the progress will be re-evaluated and requirements from this list will be introduced depending on complexity and remaining time and resources available.

- Power Ups
 - Increased health
 - Increased shot damage
 - Enemies freeze
 - Move quicker
- Boss Fights
 - Enemy unit with lots of health - harder to kill
 - Enemy has greater weapons
 - Enemy does not move off screen (has to be killed)
- Different Weapons
 - Greater damage
 - Single-shot
 - Multiple direction shooting
- Increasing Difficulty
 - Enemies spawn more often in larger numbers
 - Enemies have greater damage
 - Enemies have greater health
 - Enemies shoot more often
 - More boss fights
- Background music added
- Endless mode (player respawns)

Chapter 3

Design

This section is crucial. Describe the overall structure of your program at a suitably high level of abstraction. For instance, UML diagrams or informal box-and-arrow diagrams can be used to describe program structure. Be sure to describe the MVC structure used. Note that code listings or screenshots are not appropriate here. An important point is how you have divided the project into modules that different team members can work on, and how these are then integrated. For example, you could use interfaces to describe a clean boundary between modules, so that some team members use the functionality provided by the interface, while another team member implements it. Bear in mind Software Engineering principles of good design like coherence and coupling.

3.1 Release Plan

This section outlines the proposed release plan of the project. It has been decided that the project will take the incremental approach of software development with acceptance testing taking place at each stage. The six internal releases will be implemented consecutively to bring the project to the first public release within the 10 week timeframe.

- **Version 0.1:** Basic player spaceship (Java shape) on screen with controls and movement.
- **Version 0.2:** Basic shooting from the player's spaceship.
- **Version 0.3:** Static enemies on the screen, player able to shoot the enemy and score incremented.
- **Version 0.4:** Enemies have movement with the use of paths, are able to shoot back and collide with players.
- **Version 0.5:** Background scrolling, enemies are spawned at regular intervals at set points using a timer.
- **Version 0.6:** Multiplayer networking implemented.

This brings the project to the first public stable release which satisfies the requirements for the assessment. It time is available certain future requirements (section 2.4) towards the second public release.

3.2 Class Design

Below shows a diagram of how the project is to be organised in terms of classes and packages (dotted lines show packages).

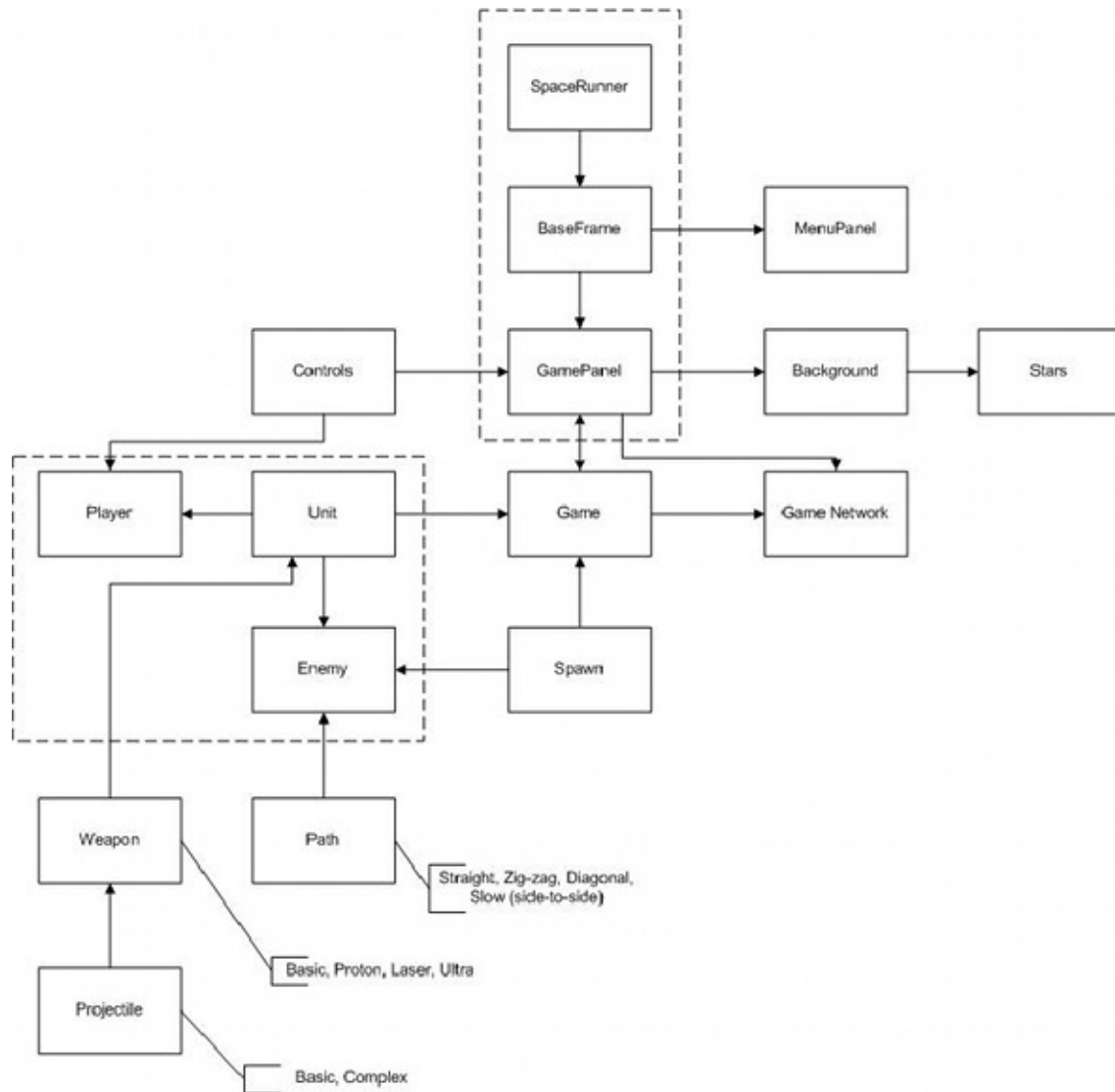


Figure 3.1: Proposed class diagram for the project

3.3 Implementation

3.3.1 GUI and Controls

The GUI was written using the standard JFrame and JPanel layout. A menu and a game panel were embedded inside the JFrame using a panel with a CardLayout. The menu and panel are part of the card layout, and can therefore be switched to very easily by referring to the name given to the CardLayout when each panel is added. There were some problems implementing a switching method to switch out the game and menu when either was required. Since the GamePanel is controlled by a timer, it is not possible to throw exceptions out. Therefore, another method was devised, by which the frame would call a method in the currently active panel to check if a 'switch' flag was set. Based on this flag, the frame would change the panel that was currently being displayed, along with performing relevant actions. For example, when the menu is accessed while playing the game, the game has to be paused, and so the frame accesses a variable in the GamePanel which controls whether the game logic is progressing or not.

3.3.2 Menu

The menu is composed of simple objects such as rectangles that switch out using card panels. These card panels allow for easy switching of panels when an action is performed such as for our game when the user clicks to start a single player game. The way the panel detects what the user wants us by using mouse listeners and checking if the mouse is within the bounds of the rectangle when the mouse button is clicked. Once clicked the panel will switch out to the corresponding panel action that was designated as the action of the "on click" of the rectangle. Another thing to note is that when escape is pressed during game play the game will pause and switch the panel out back to the menu where the user can then either start a new game or resume the current game.

3.3.3 Controls

The controls for the game were implemented by using mouse and key listeners as to allow the user the choice of input method. The main way the controls work is by first detecting if the keyboard or mouse is being used and then using the appropriate method. For the mouse it would get the current pointer location and move the ship to that location on screen as well as listening for mouse button presses to trigger the shooting method. The keyboard worked by detecting which key was being pressed and setting a boolean value to be true, this allowed the user to press multiple keys at once to move the ship diagonally while also shooting. The control class was embedded into the game panel class so it ran when ever the game was started. Using the boolean values from the control class then decided upon where the ship would move to and if it was shooting or not.

3.3.4 Game Logic

The game logic is run in a loop that is controlled by the game timer. The timer decides on the frame rate of the game which affects the speed of which objects such as projectiles can move. Also within the game logic loop major components to the game such as collision detection and array pruning are called from the logic method. This part of the game could be said to be the game core that calls all the necessary methods.

Overview

The game logic was written as a part of the program that could be used easily without any knowledge of the actual processing going on inside. The game logic is made up of a series of arrays, each consisting of objects that are part of the game; spawns, players, enemies and projectiles. These arrays are modified by the game in order to update the current state of the game. The arrays can be accessed by any class via getter methods, and objects can be added and removed if necessary. The logic can also perform removal of objects on command using certain pruning methods which are described later. The game logic also has methods to

move all objects on the screen to new positions. The objects inside the arrays are manipulated inside the game panel using the methods of the game class. After logic has been done on the arrays, the panel then draws the objects inside the arrays.

Collision detection

The collision detection is naïve, and done in two separate stages. Collisions of enemies with player projectiles is done first, and then collisions of enemy projectiles with the player, and enemies themselves with the player. This method is quick, and is not particularly computationally intensive given the type of game that we have written. Whether a collision actually occurs or not is defined by using a rectangular ‘hitbox’ which surrounds the unit or projectile. If the hitboxes overlap, then the damage that the bullet carries is transmitted to the unit, and if the unit’s health drops to zero or below as a result, the unit is removed from the array. The projectile is removed from the array when it collides with a player or enemy object. Doing this gives quite a realistic feeling—it is not usual that a projectile passes through an object. However, implementing weapons that travel through units would not be problematic either, given the structure of the code. In order to avoid concurrent modification problems, enemies and projectiles that must be removed from arrays must first be stored in a secondary arraylist, which is then used to remove objects from the original array once all of the collisions have been checked.

Array Pruning

So as not to overload the memory of the computer with a huge number of objects, the arrays in the game are pruned of projectiles and enemies that are no longer within the visible area of the screen. This is done by creating a rectangle the size of the frame window, with the same coordinate space, and then, avoiding concurrent modification, removing the objects whose locations are not within the bounds of the frame rectangle. This also means that less processing is necessary during collision detection, and there are fewer objects that must be dealt with.

3.3.5 Game Entities

Units

For the units, we defined an abstract class that would contain all methods that would be necessary to access the basic information about the unit such as health. The game contains two subclasses of the unit class, player and enemy. In each of these classes, there are some parts that are specific to the requirements of those objects. For example, enemies have a method to get the value in points for destroying them, and the player class has a method to increment the score for that player. Using an abstract class meant that the subclasses already had all the necessary methods for access, and only needed some simple additional methods that were specific to the type of unit, and this saved writing quite a lot of unnecessary code. Each enemy unit moves based upon a path. Paths consist of a single equation, along with some directional variables to determine the next position of a unit based on its current position. Each time the game logic is run, this method is called with the object’s current location, and returns the location that the object is due to move to next. This sort of implementation of paths allows for a robust method of control for both enemies and projectiles, and can be used to create a variety of interesting path types relatively easily, which allows for good extensibility. These paths also mean that the speed of units and projectiles can be controlled by using some sort of speed variable to increase the multiplier on the default additions to the coordinates, which would be very useful if there were to be difficulty levels added.

Weapons and Projectiles

The weapon and projectile classes are fundamental to the gameplay of the game. The weapon class is an abstract class which is intended to allow for quick addition of new weapon types into the game. Each weapon has a fire method which causes the weapon to create a new arraylist containing a number of projectiles

specified by the method. This means that it is simple to add new projectiles to the game logic from quite a high position based on object orientation. Once one has a reference to a unit, it is only necessary to get the reference of the weapon, and then add the return array of the fire method to the game logic when one wants to cause weapons to fire. Each weapon has a fire rate, which is used by the logic to determine whether the weapon's fire method should be called or not. This allows for precise control of the weapons firing.

Projectiles are very simple objects which move based on paths. Each projectile object contains some data about which player it was spawned from (if it was), and details about its damage.

Spawns

The Spawn class is used to create new enemies during the game. The game has seven different set spawn points at regular intervals along the top of the screen, these points are held inside an array within the Spawn class. The class is currently able to spawn three types of enemy units: default unit which is abstract so that future types of unit can be created, boss enemy which is harder to shoot as it has increased health and different weapons as well as moving along a different path and finally the basic enemy unit which makes up the majority of the game. The spawn methods are called on the Spawn object in the GamePanel class at regular intervals in the game cycle (timer). There are various different types of spawn options available:

- Spawn 1 unit at a given point (0.8% chance of it being a boss enemy).
- Spawn a given number of units at the same given point.
- Spawn a given number of units at the same random point.
- Spawn a given number of units at different (chance of it being the same) random point.

Each of these methods returns an ArrayList of Units which can then be drawn in the render method in the GamePanel class. The Spawn class is abstract and can be modified in the future to add new types of enemy, the frequency can be changed in the GamePanel of how often units are spawned too.

3.3.6 Network

Networking is based on a client-server principle. Upon choosing the multiplayer option in the menu, the player is given a choice of hosting or joining a game. Depending on the option chosen, the game panel is initialised with server or client object which perform different tasks. When a client connects to the server via an input IP address, the server receives a reference to the client, which it then stores in a map. This map is used to get references to clients when data has to be sent to them. With a map, it is easy to send either to all clients, or choose a specific client to send data to. Client connections are dealt with by a handler, which sits in its own thread and waits for connection attempts to the server. When a connection is received, the handler adds the connection to the server's map via the interface, and the client immediately begins receiving any data from the server. Each individual client is embedded inside a class which contains methods to perform specific actions such as sending data over the network. The client and server perform interactions by first sending a string containing some information about the message that is to be sent next, and then sending the message. This method of communication is facilitated by a listener which runs in its own thread and constantly listens for commands coming in. If a command is received, a method in the socket is called to execute the request. This well structured communication method means that all communication can be handled in the correct way for that particular command. The method by which data is communicated over the network is via game state objects, which contain all the data needed by the game to run correctly; all the arrays, and boolean variables representing whether the game is running or not, along with a few others. The game state is capable of doing some processing, in order to reduce the data that is transferred over the network. A client first receives data from the server, and then runs its logic on it, moving projectiles fired by all units, and moving the player location. Each loop, the current state of the game is made into a new game state. This game state is then compared against the game state that was received from the server at the start of the loop. All objects that are the same in both game states are ignored—only new objects and

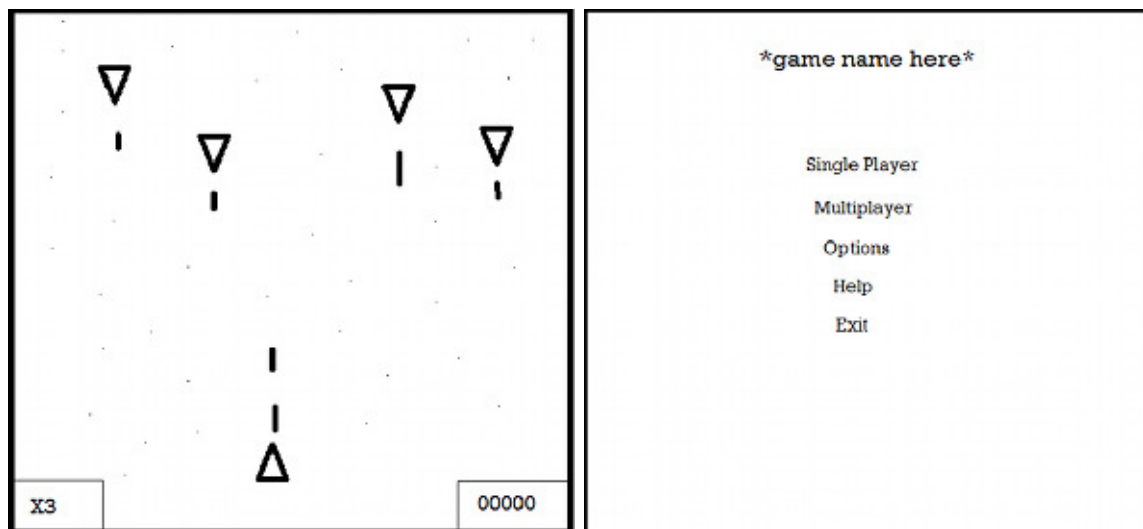
the player reference are passed to the server, which reduces load on the network and also makes it easier for the server to collate the data into a single new game state which combines data from all clients which are currently connected. The server takes the new amalgamated game state, performs logic on it, and then sends it out to all clients, and the cycle continues. This method, along with the client storage method means that clients can drop in and out of the game without any wait time. In order to facilitate the transfer of objects over the network, upon object creation, each object is given a reference which will allow checking of the objects even if they are duplicated later, which is necessary for the removal of old objects.

3.3.7 Graphics

Graphics is implemented using a map stored in the game panel class. This map is initialised upon loading of the game with images from the relevant locations. Each object which must be drawn has an image reference string, which refers to the string that the image was added to the map with. When the object is drawn, the image reference is used to retrieve the image for that particular object, which is passed to the draw method of the object, which then draws it. Using this method meant that it was not necessary to pass images over the network, which would greatly slow communication due to the relatively large size of images. However, this method could, in theory, lead to the modification of the image files which are used to draw objects. In actuality, this will have no real effect on gameplay, since the size of the images are not at all considered when processing game logic.

3.4 GUI Design

The GUI will be fairly simple and initially the on-screen components will be made up of shapes (Java objects). The initial sketches are shown below which were created in the first week of development to aim understanding of the project.



The menu screen (right) will have buttons that allow the user to select different types of game play, for example single or multi-player, as well as being able to exit the game or select different options. The menu is also accessible from within the game using the Esc key.

The game is to be implemented in a single window (frame) with different panels added (i.e. menu and game panels). The frame is to be resizable (including the layouts of ships) to allow the user to change the game from the default size.

In the Game Panel (left) the objects are represented as shapes. These are to be replaced with sprites (images) if the project allows sufficient time.

Chapter 4

Validation and Testing

4.1 Testing

During our initial discussions, we made some simple plans for how we would attempt to test each part of our program. Since each member of the team would be writing some parts of the code, it seemed to make sense that whoever wrote the code should test it as well. We decided that it would be a good idea to use JUnit tests, since these allowed for the mass testing of all classes in the program once everything was written. If we were to use another testing method, it would likely result in having to run some part of the program and check that its output was correct, or use a multitude of print statements to check that the program was running correctly, which would lead to having to trawl through a significant amount of data and pick out relevant parts, which would likely be very difficult to find, given that bugs do not necessarily occur frequently, and when they do, the signs are not obvious when looking at output. Once we had actually started the project, however, it became obvious that many parts of the game were not conducive to testing. For example, testing controls with JUnit is possible, but not particularly easy.

The easiest way of testing the controls is simply to play the game, see if there are any problems, and then attempt to fix them. As development progressed, we discovered that writing tests before or after writing code meant that our development speed slowed significantly. We would spend quite a while writing tests to make sure that relatively simple parts of the code worked correctly, when it was obvious from playing the game that everything was as expected. Eventually, we decided that using JUnit was slowing us down too much, and did not actually provide as much benefit as we had initially hoped that it would provide, and we each used testing methods that we usually used when programming. Mostly, this involved reading the output at specific parts of the program that we knew would contain problems.

Of course, testing was only done if something didn't work properly—there was no need to test something that didn't seem to be broken. Since we were running the game regularly and playing it to check if the code we had written was doing what it was supposed to, we believe that any bugs we had were spotted far more quickly this way than by laboriously writing JUnit tests and then using them to check that everything was working properly. By playing the game, we could test all the conditions that we would expect the code to work in, and if anything was broken, we would move to the output reading method, which in most cases worked without a problem. Printing out variables or arrays to see their current state allowed us to quickly see where the problem lay. There were a few times where this method did not allow us to solve the problem, and in these cases we went to the built in debug tool in netbeans. By stepping through each part of the program, or parts of the program that we thought were causing the problem we could see the state of all variables in an organised way, and to step into methods that were part of the execution process, to check if there was some kind of error we were making in the order that code was executed.

We think that the main problem with the lack of success of the JUnit tests was the fact that the code distribution between members was not particularly even. As such, some members perhaps didn't understand the workings of a certain part of the program enough to write a test for it, and so tests were left unwritten. Since towards the end of development, there were only really two of us working on the code, we did not have sufficient time to test the entire code base, as we had to concentrate on writing the parts of the code that were required to complete the program. Perhaps if we had split the work in a more appropriate way, to allow everyone to work on some part, rather than ending up with two of us working on everything, we would have had enough time to write proper tests for everything. Another way that we could have solved this problem would have been to have one member who was responsible for testing all of the code, and not write any actual game code themselves. However, this person's understanding of code would have had to have been better than that of those writing the code, in order to be able to test everything correctly. Given the skill levels of members in our team, it is unlikely that this would have been possible, as the majority of the code would probably have not been written, since the person with the best understanding of code would probably be the one who would be needed to write most of the complex code, and other team members would likely be unable to test the code that that person would have written. Overall, however, apart from the networking code, we believe that everything has been tested to a relatively satisfactory standard by using only the testing methods that we have used. No major bugs in the code have been discovered yet, and as far as we are aware, the minor bugs have no real impact on gameplay.

4.2 Validation

This section compares the game's performance against the specifications that were originally outlined in the earlier section.

4.2.1 Functional

1. The human player is able to control one's spaceship

This was validated by each member of the team. As this is such a central part of the game, controls were tested almost every time the game was run, and so this section is well checked. The controls used can either be keyboard or mouse. The spaceship of the user is able to move freely, but can leave the frame boundaries when using the mouse to control it. The diagonal movement of the ship is not normalised, and therefore diagonal travel is slower than travel along a single axis. When using a mouse, this is no longer a problem. The spaceship is represented by a graphical sprite, whose image can be easily changed.

2. The human player will be able to shoot

Shooting is again a central part of the game, and so was tested often. In initial stages, tests were done with JUnit, and then later on through playing the game. Projectiles which are no longer visible are removed from the game so that they no longer need to be processed.

3. Enemies will be created to be destroyed by the user

Enemies are created at regular intervals and can be destroyed by the user once then are hit with enough projectiles. Each enemy has a specific amount of health and is destroyed once this value reaches zero, usually after being hit by player projectiles. Boss enemies are also created with a small chance, and these have different firing patterns and health than the standard enemies. Enemies have a different sprite to the player, which can be changed easily if necessary.

4. Enemies are to be able to return a level of resistance

There is a significant amount of resistance to the player's progress. Enemies fire a large number of projectiles in a variety of different paths. In particular, boss enemies use weapons that fire rapidly and produce a large number of projectiles. Enemies have fixed paths depending on their type. These paths are different for normal and boss enemies. Boss enemies will stay on the screen until they are destroyed. Colliding with any enemy or too many enemy projectiles will cause the player to die.

5. **The game is to run continuously with set events occurring at regular intervals**

While the game is running, the game loop spawns enemies at specific time intervals. Enemies will keep spawning until the player dies. The enemies move each time the game logic is updated, which is every tick of the timer. The difficulty of the game can be controlled by increasing the rate at which the timer fires events.

6. **The game will be able to be multiplayer across the network**

It is possible to play the game with multiple players, although the game has not been tested with more than two players. Upon choosing the multiplayer option, players can choose to host a game or to find a game already running. Hosting will put you into a game, and you can wait for players to connect. A client will input the IP and port number of the server and will be connected. When the client connects, it drops into the server's game with a new player spawned that is controlled by the client user. The game can be connected over a network. During gameplay, users can see ship locations and projectiles the same as on any other screen. There are some problems with projectiles not correctly displaying, and some duplicate players appearing behind the actual position of the client player, but this is only present on the client's screen. The positions of all objects on the screen are updated each tick, when the logic is executed on the server. The execution of the logic also sends the server gamestate to clients.

7. **The game must have a terminating clause**

The game ends when the player loses all their health or collides with an enemy. The player score is displayed during the game, but not on the game over screen. The score is not stored anywhere, as there is no facility to do this.

8. **The game will include a Graphical User Interface (GUI)**

The game has a basic graphical user interface. When the game is run, the user chooses options for games by clicking buttons displayed on the screen. The game is all run in a single frame, with panels being switched out depending on the state of the game. The menu can be accessed at any time by pressing the escape button, and the game can be paused with the P key. The window can be resized, but the background does not draw correctly when this is done, and enemies do not spawn all over the screen, but remain in the top left side of the screen. Both the game panel and menus feature an aesthetically pleasing starry background, which gives the user the impression of flying through space. Each projectile type and players and enemies are represented by a graphical sprite.

4.2.2 Non-functional

9. **To provide a simple, easy to use system in order to play the game**

To validate this requirement the team enlisted the help of several people outside of the game's development environment which is an example of alpha, or acceptance, testing. The game was first given to a young novice player and instructed to play the game as if they had just downloaded it for the first time. The team observed the player while they did so and came to the following conclusions. The player was able to navigate the menu very easily and was into the actual gameplay within just a few seconds of launching the program. However, the game soon became difficult resulting in the player's unit 'dying' within around twenty seconds. The user continued to relaunch and play the game for around 15 minutes. That person agreed that the menu system was easy to use with a clean look and feel. It took about 12 minutes before the user had a good run and became used to the game. This was slightly longer than the anticipated 10 minutes but is still acceptable for use.

Similarly, the team asked an expert gamer from Computer Science to test the game and agreed with the earlier comments but became familiar with the game after just a few runs.

10. **The game should be quick to respond**

With this requirement, all members of the team ran their own tests playing the game after it had been implemented. It was noticed that the game starts to lag after around 2-3 minutes of continuous gameplay. This bug had only been first noticed with only one week until the deadline. The team set

about discovering why this way - the arrays which hold the objects to be drawn were not too large and had been pruned correctly so the bug still exists. However, at first the game is very responsive to the user's input having an almost instant affect on the gameplay. Network testing showed up no errors and worked every time the program was run (over 20 times). However, if the user is trying to do something out of what they should be it crashes. For example if the user tries to connect to a host in multi-player mode but inputs the wrong IP address or the host is not there, it crashes without any dialogue boxes.

11. The game should run first time, all the time

To test this requirement the game was run over and over again 20 times. The team added various outputs to the console (print line statements) to show what the game was doing. For example, the game would output a message when the controls were enabled, various networking messages as well as outputting which key was being pressed. Two people would observe this test; one to play the game and the other to look at the output messages. This test was successful and the game didn't crash once.

12. Network clients should be able to drop-in within 5 seconds

Again to test this requirement the game was run multiple times. The drop-in occurred almost instantly when it was being played on the same machine (localhost IP) and in around 1-2 seconds when using a LAN router. It is not possible to play over a WAN currently.

13. The game should run on the platform detailed in system requirements. This test was noticed early on; the team all have various platforms to develop on (Mac OSX, Windows 7, Windows XP and Ubuntu Linux). The game managed to run the same on all platforms with no major differences (frame theming changed due to operating system designs).

14. The project should be developed using appropriate software engineering practises within the time frame for the assessment.

The team used the incremental software life-cycle as detailed in the team management section of this report, the team also produced a Gantt chart showing the time frame for the assessment and, although there were some delays, the project was delivered on time (both the report and software).

15. The project must be written in the Java programming language.

This was a core requirement of the module, the team managed to use the Java API as well as other lecture notes to produce the software. The team was familiar with the Java language as it had been studied in previous degree modules and only certain aspects (e.g. networking) needed to be researched.

16. The multi-player game must be able to run on any LAN with the IP addresses given.

During the testing phases, the game was tested on multiple LANs and locations and all seemed to work perfectly fine. It is also possible to run two versions of the game on one machine and use the "localhost" IP address - although the game isn't really playable as the same controls are used for both copies of the game.

17. High scores will remain stored in each copy of the game until they are overridden by a higher score.

This requirement was not possible to test as the functionality had not been added to the game yet. The game did however display a score during the game in the bottom right-hand corner of the screen but no scores were recorded. This would be an easy modification to implement by writing the scores to an external file and using algorithms to check if a high score had been beaten.

Chapter 5

Project Management

5.1 Development Methods

We discussed the possible use of software engineering techniques we could utilise to make things easier and to ensure success and good planning. In a way it was hard, since we have no concrete experience about how much work a single software engineer can do and so forth.

5.1.1 Weekly Meetings

In the weekly meetings we discussed our progress so far on the whole project and any problems that had arisen on the week and how we could get around them and fix them. We went over our plans of what we would have to have done during the next week and revised if it still was a good idea and possible to implement at that stage. After deciding what we wanted to accomplish in the next week we tried to split the tasks evenly amongst us.

We usually had two meetings a week, one on Tuesday and one on Thursday with the demonstrator. After we got feedback from the demonstrator we reviewed our plans. Depending on what we discussed we usually wrote logs and drew some rough drafts about what to do. Our main communication medium has been Facebook. We formed a group there where we only have the members of our team and we have been dealing with assigning tasks over it and talking over any problems we might have. I have been quite surprised how well it has worked for a thing like this.

5.1.2 Incremental Stages

We decided to build the base game first, without anything fancy in it, as in the absolute core of the game. So we started out making the ships with just shapes and not sprites. After we got that to render properly, we started adding movement, opponents, spawning and shooting.

Basically we built the game in incremental stages, first making the core and then adding functionality on top of that. The goal was to have a working copy with more functionality at the end of every week, according to our schedule and plans. I think we managed that fairly well. At the very start we decided to try to keep the game at a bare minimum, no extra fancy stuff added or even planned very far until the core was working well, since we knew the hazards of getting stuck with planning and daydreaming without getting anything worthwhile working in the end.

5.2 Planning

We ended up with the idea of trying to document everything as well as we can and use weekly meetings to plan out the tasks for that week and where we want to be after it. So we set our iteration period to last a week and assigned tasks at the start of it. Requirements documentation and user requirements was among the things we knew are critical if you want a project to succeed in the timeline it has been given.

Trying our best to decide how much time to use on each part and how to divide the tasks in even portions, so that the contribution would be spread evenly amongst us. But as said above, we found it to be rather difficult since we had no clear concept of how hard it would be to code networking and other parts we had no experience about.

One thing that was a bit daunting was the team work itself, since no-one of us had any experience in it either, so how would it work if we had to work on a class that someone else was working on already and how to arrange it all so that it would not get too messy. In the end we drew a rough diagram on how to organise our packages and classes and a basic timeline of when we would want each version to be out and what would it entail.

5.3 Testing

Testing is obviously an important part of any software development. At the very start of the project we decided to test everything well and thoroughly, if possible. We didn't have any experience with JUnit or testing in general, apart from the things we have done to debug our code and show testing with main methods during the first year.

After the lecture on testing and some help with the demonstrator we set up our tests and managed to test everything we had done so far and it all seemed to be in order. At the next stages of the project, I must admit, that we did not really utilise JUnit testing until the very end. But this does not mean we did not test everything. Everyone took care that the code they committed was tested with main methods and different kind of System.out.print commands.

5.4 Time Management

When beginning the project, a time plan was created to provide detailing to ensure the project stayed on track and was completed on time. The Gantt chart for the project is shown below.

Section	Duration (weeks)	1	2	3	4	5	6	7	8	9	10	11
Meet Team / Initial Ideas	1											
Specification	1											
Design	1											
Game Logic & Controls	2											
Test Classes	1											
Spawning Enemies	2											
Background / Menu Design	1											
Graphics	1											
Networking	3											
Document Report	3											
Fix Bugs / Tidy Code	1											

The project was delayed slightly towards the end of the project which mean't it would become slightly rushed. However, the team managed to allocated appropriate resources to the sections that required more attention (networking) whilst the remainder of the team continued to work on the final report.

Overall, the project was delivered on-time and the coding introduced additional features which weren't in the initial requirements but documented in the future requirements.

Chapter 6

Conclusions

After completing the coding section of the project it has been possible to evaluate how well the whole experience went.

6.1 Strengths

First of all the project was a success; we were able to implement a multi-player space shooter type game within the time limit specified in the module.

The game features several items that we did not originally think we would be able to implement, these features are the background music which was added towards the end of the project, graphics for the units and projectiles which were implemented with a few weeks to go and also the boss enemies which add a new level of difficulty to the game making it more interesting.

The management side of the project was also successful as the team were very good at communicating ideas between each other. To do this the team organised weekly face-to-face meetings to openly discuss the week's work as well as evaluate what had taken place that week and what still needs to be done. Although the project may have seemed to be rushed towards the end, it was still delivered on time and is playable both as a single player game but also as a multi-player networked game.

During the development of the project the team faced a setback where one member of the team had to leave the UK, despite this inconvenience the team still managed to communicate effectively through email and Facebook and didn't cause too much disruption to the project as a whole. In this scenario the team decided it would be best to break the project up into larger sections so the team member could work alone without needing as much communication with the rest of the team like the start of the project had done. This was a success and the team member managed to contribute both to the coding sections of the game as well as the final report.

6.2 Limitations

Some of the game's limitations have been documented in the testing and validation section.

Although the game itself functions properly and is playable there are some slight bugs as with any software project. After playing the game for a few minutes, the number of enemies and subsequently projectiles increases. The game manages to prune the arrays containing the objects when they leave the screen but soon the game starts to slow down and become slightly unresponsive. This bug was only introduced in the last week and due to time limitations, still exists.

The project did become delayed towards the end of the project, and, although it wasn't too much of an issue there are some claims of it being 'hacked together'. No one in the team originally had any experience in network and sockets programming which was required for the game, it was decided that this part would be left towards the end of the project while we would learn more during the term through lectures and other modules. The networking section of the project does work properly, however the code is quite messy and some shortcuts have been taken. If time was not a factor this section could have been smoother and implemented better.

As previously mentioned, one of the team members had to leave the UK during the development of the project. Communication became harder as it was only possible to communicate through electronic forms such as email and Facebook rather than the preferred face-to-face meetings. It also meant that the team member was not present for the final demonstration to explain their part of the code. Despite this setback the team still managed to proceed with the project and the team member mentioned was able to contribute to all aspects of the project (coding and report) successfully.

There are also some other limitations that relate to the game play. The enemies all have the same path to move along (except for the boss enemies) which makes the game slightly predictable. There are other paths available to use but not method to call them, perhaps a random way of choosing a path should be implemented. Also, the mouse movement can be quite sensitive if the user is not used to it which makes collisions with enemies hard to avoid.

6.3 Continuation

If the project were to be continued without a time limitation then the game could be improved. The networking side of the game could be more stable and tidy and as the project is already quite abstract perhaps certain features of the current existing game can be improved, for example the enemy paths.

During the project one team member had to be away from the UK which led to certain communication issues. If the project had no time limit the team may have decided to give that person a break until a time where they could become more involved. However, in this case the team was able to work around this problem and being forced to overcome the communication difficulties which occurred with a relative degree of ease.

Outlined in the requirements of this document is a list of future requirements which could be implemented if the team were to continue the project. However, this list isn't too long which identifies that the scope of this project isn't very large. Currently the only things the player can do is move and shoot the enemies, or to do the same thing along side a network player. The game could be improved though to become more exciting but the core principal of the game isn't very large.

Currently when the player 'dies' the game ends and displays a 'Game Over' screen (including in network play). To play the game again the user has to relaunch the program, future versions of the game could see the possibility for an endless mode (as documented in future requirements) where the user is re-spawned after they 'die' as well as having the option of a 'Play Again' button to avoid relaunching the program.

6.4 Future Projects

From completing this project the team has learnt a lot about the processes of software engineering and development and would make changes to the way in which the software was developed in the future.

The software management aspect of the game seemed to run smoothly but actually there was more work

to do than what had first been anticipated. Getting to the first version of the software (v0.1) took quite long (a few weeks), this seemed to be acceptable as it would include the core basics and would see the implementation of the base cases. It was then thought that the later sections of code would be able to be introduced without changing the code too much. One of these examples is with the networking side of the program. As mentioned previously the team had relatively little experience in sockets programming and it was decided that this section would be left until last to implement. Looking back, this was perhaps the wrong decision. The thinking behind leaving the networking until later was that the team could learn more during the term from other modules as well as programming lectures. In the future if the team had little knowledge of an important aspect of the project it would be advisable to research that field from the start and try to implement it in small steps rather than as a whole when time is running out.

The scope of the project is quite small as there is little else the user can do rather than move and shoot. Although the game is large enough for the purpose of this assignment, in future projects perhaps a different game type could be suggested. Conversely, the team all enjoyed working together to produce the game and the course has benefitted each individual for future projects. The team believes the game works well despite having some small issues but all team members are happy with the output of both the software and the report.

Chapter 7

Appendices

7.1 Facebook

The team used facebook for the communication and discussion of ideas, as well as distribution of tasks. To see the logs of this activity, please see the location `/trunk/docs/logs/facebook.pdf`