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

Project Report: DarkMatter

Team A2:
Jeremiah Via
Yukun Wang
Charles Horrell
Joss Greenaway

March 31, 2011

Work Breakdown

Coding	**Joss (Team Leader)**	**Jeremiah**	**Yukun**	**Charles**
Documentation	Contribution: 5%	85%	5%	5%
Unit Tests	10%	30%	30%	30%
Networking	0%	95%	5%	0%
Menu	0%	70%	30%	0%
Animation	0%	10%	90%	0%
Music	90%	10%	0%	0%
AI Player	30%	0%	30%	40%
Human Player	0%	50%	50%	0%
Level Loader	0%	90%	10%	0%
Level Editor	0%	10%	95%	0%
Level Design	0%	33%	33%	33%
Menu Art	50%	0%	50%	0%
Game Art	0%	100%	0%	0%
Physics	25%	25%	25%	25%

Report	**Joss (Team Leader)**	**Jeremiah**	**Yukun**	**Charles**
Introduction	Contribution: 0%	80%	0%	20%
Abstract	0%	100%	0%	0%
Requirements	0%	50%	0%	50%
Design	0%	100%	0%	0%
Validation	0%	100%	0%	0%
Management	0%	40%	0%	60%
Discussion	0%	33%	33%	33%
Conclusion	0%	100%	0%	0%
User Stories	0%	0%	0%	100%
Minutes	0%	0%	0%	100%
LaTeX	0%	100%	0%	0%

Contents

1 Introduction **1**
1.1 Initial Ideas . . . 2

2 Requirements **3**
2.1 Functionality . . . 3
2.2 GUI Design . . . 4
2.3 Software Engineering . . . 4
2.4 Networking . . . 5

3 Design **6**
3.1 Overall Architecture . . . 6
3.1.1 Players . . . 6
3.1.2 Levels . . . 7
3.1.3 Networking . . . 8
3.1.4 Polish . . . 10
3.2 Division of Work . . . 11

4 Validation and Testing **12**
4.1 Testing . . . 12
4.2 Validation . . . 13

5 Project Management **14**
5.1 Subversion . . . 15
5.2 Maven . . . 16
5.3 Software Engineering Approach and Principles . . . 17
5.3.1 Other Methodologies . . . 18
5.4 Release Plan . . . 18

6 Discussion **20**
6.1 Team Mistakes . . . 20
6.2 Individual Mistakes . . . 21
6.2.1 Charles . . . 21
6.2.2 Jeremiah . . . 22
6.2.3 Yukun . . . 22

7 Conclusions **24**

A User Stories **25**

B Meeting Minutes **26**
B.1 Group Meetings 26
B.2 Demonstrator Meetings 32

Abstract

Games are an interesting experience for a software developer. They require vast amounts of effort to implement even the most trivial features. They're also unique because we can actually show our work to friends and family and get more than "So?" as a response. In fact, if done correctly, games can inspire excitement in others which makes them rewarding to work on.

Our aim for the module was to extend the premise of an enjoyable physics-based game. The inspiration for our game was Osmos by Hemisphere Games (Hemisphere Games 2009). Osmos is a game with a simple idea: absorb motes smaller than you while avoiding motes large than you. This game is a casual game that is more about solving puzzles than being the quickest.

We had the idea to change this casual gameplay into something more exciting. By making the game multiplayer, the time to carefully consider your options is gone. You must now absorb as much matter as you can before your opponent does. This game is also unique among many multiplayer games because the loss of one player does not entail the victory of the other. It is possible for everyone to lose!

Completing the transformation is in the details. We threw away the ability to change the speed of time to focus on the competitive nature of the game. The goal of the game is to become huge. We wanted to create an experience that pressured players to accomplish this goal as fast as they could. As a final detail, we added some upbeat music instead of the slow, ethereal music of the original.

Chapter 1

Introduction

This report aims to document how our team was able to create a game. We had roughly 10 weeks to create a game from start to finish. This was a new experience for all of us and we learned many lessons from it. We are all better programmers now and we have a better idea of what it means to create software with a team.

We will begin by describing the final specification of our game. It changed from our initial specification as we learned more about game programming. This was a new domain for all of us so our initial ideas of what it meant to make a game morphed as we gained experience in the field. We will also explain why we chose the features we did and how those features positively affect the user experience.

After going over the specification, we will go into the design of our game. This section will go into the vital details of our game architecture. You will learn about problems we encountered when implementing features and how we solved them. We want to not only tell you what architectural choices we made but why we made them. This will give you insight into our problem solving abilities as a team.

We will then tell you about how we tested our game and validated its user experience. This is an important section and we aim to highlight how extreme programming methodologies allowed us to create good software. We will show you how unit testing allowed us to assert our intended uses of the code and ensure future feature enhancements would be prevented from breaking old, stable code.

With testing and validation out of the way, we will talk about how we managed our project. Because this was our first experience writing code as a team, it presented us with new opportunities for learning. We will discuss how extreme programming helped us maintain forward momentum on our game and how utilising pair programming ensured that everyone understood the code to a sufficient level.

We would like to wrap our report up with a general discussion of how the module was for us. Programming a game from start to finish as a team was a novel experience. As a result of this we made mistakes. We will talk about the mistakes we made as a team and individually and the lessons we learned

from them. Because of this module we have an increased maturity in our programming, meaning that we can recognise the long-term issues with the choices we make.

1.1 Initial Ideas

At our first meeting we considered various games and discussed many ideas. We debated the merits of various games and ended up deciding on criterion for a good game. The game we would make would need to lend itself to an iterative release schedule. This was important so that we could add new features each week. We wanted to avoid anything that would require multiple weeks to implement. We also wanted a game that would be fun to play. This was a challenge because many entertaining games have a lot of complex features. These two criterion constrained our search for a game to make.

Ideas for games that were ultimately rejected included:

Fruit Ninja A game which involved the slicing of fruit with a blade controlled by the mouse. This game was rejected because we could add nothing original to it or come up with any particularly fun or innovating multiplayer modes other than a who can post the highest score style game.

Tower Defence A game involving the placing of towers and weapons to destroy enemies that work their way through a level. This could possibly be made multiplayer in a cooperative mode.

Zombie Tower Defence As above but involving zombies. We rejected both the tower defence games as we felt they had been done many times before and lacked originality. There were also free versions available of the game online that were very playable.

Space Racing Game A racing game which takes place in space where the players must race while avoiding planets and asteroids. This was rejected again for lack of originality.

The game we finally settled on was a multiplayer version of Osmos developed by Hemisphere games (Hemisphere Games 2009). We called our game Darkmatter as we would be setting it in space with the concept of stars rather than on the cellular level like Osmos.

All of us were happy with this selection as it included features that played to the strengths of all members of the team. Joss with the potential to produce exciting graphics and audio effects, Charlie with the chance to use his physics knowledge and Jeremiah and Yukun to utilise their strong programming skills.

Chapter 2

Requirements

2.1 Functionality

The game will be a multiplayer game with simple mouse-based controls. Players move about the environment by expelling a proportion of their own mass in order to gain momentum. Gameplay consists of absorbing smaller objects and the avoidance of absorption from larger objects. Combining these two constraints together force players to create a balance between their speed and their mass.

To win the game, a play must absorb 60% percent of the total object area on the map. Once this requirement is achieved, a player cannot lose because no challenger could muster enough mass to attack. The loss of one human player does not imply the victory of another. It is possible for all human players to lose the game. We feel this keeps the game exciting.

In order for the game to look and act correctly, physics has to be implemented as realistically as possible. The key physical properties we need to implement are the conservation of momentum and a modified version of inelastic collisions. It needs to be modified because it must take into account that some energy is lost in the absorption of a matter object by another.

Levels will vary in style to force players to think of creative ways to solve them. We aim to include levels that require expert movement through a maze of larger matter objects and others that reward playing as fast as possible. This will keep the game fresh and exciting for players.

To allow players to extend the game, we want to create a level editor if time allows. This lets users utilise their creativity to create new levels which we may never have thought of. This extends the value of our game by allowing users to get more use from it.

Another advanced feature we want to implement, if time allows, is power-ups. We think this would add an entirely new level of game play. After much thought, we decided that defensive power-ups should be preferred. This would allow players to escape death rather than being able to dominate other players.

2.2 GUI Design

We want to design a beautiful graphic experience for our users. To do this we want to take advantage built-in Java capabilities. We had considered using the Lightweight Java Games Library (Rychlik-Prince & Matzon 2011) for this task but given the time constraints we feel it would be too much added scope to our project.

To create an immersive experience, we will make the game full-screen. But in order to prevent those with larger screens from having an unfair advantage, we decided to fix the size of the game area. Anything larger than the game area is rendered as an attractive, dark background. This lets the user focus on the actual game play.

Because it can be difficult to distinguish the difference in area between two objects that are nearly the same size, we wanted to create a visual way to easily glean this information fast. Our solution is to use colour. By using warmer colours to indicate that an enemy object is larger and cooler colours to indicate that an enemy object is smaller, users can quickly and intuitively reason about how they should move about the map.

Adding music to the game was the final touch we feel is needed in order to create an excellent game play experience. Since our game forces players to calculate their mass and speed quickly, we want to add some up-tempo music to increase their stress levels. We also want music that would create that feeling of drifting in space.

The matter objects are constantly varying in size, so we feel that giving the player the ability to zoom in and out would increase playability. Since this is no easy task, we have decided to make it an extended feature we would implement once the core of the game was finished.

2.3 Software Engineering

As mentioned earlier, we will be using the extreme programming methodology for this project. We all feel it will be a good fit for this project and our team. We hope to code at a consistent pace so at the end we are not forced to work feverishly just to meet the deadline.

We chose extreme programming for a number of reasons. The most important one is the principle of releasing early and releasing often. Because we have a short time in which to create our game, we need an impetus for adding functionality each week. Weekly releases provide this very need. By defining a set of features we want to have for each weekly release, we can keep forward momentum on our project.

Another important aspect of extreme programming we plan to adopt is collective code ownership. We feel this is important because we all need to be able to work on the same code. This also creates an environment of trust where we all depend on each other to ensure the game is working well. We never have to depend on one person to fix a bug.

We plan on doing a lot of pair programming for the project. This should

help everyone have an understanding of the code base. It will also reinforce the idea of collective code ownership because we will be working on classes together. Currently, we meet up three times a week for our pair programming sessions. That way everyone spends time coding with everyone else.

We plan on creating a lot of unit tests to ensure our code is always working. This will free us from worrying if any new changes have broke any exiting features. As long as we have good, specific tests we can rest assure that our product is functioning as intended.

2.4 Networking

In order to incorporate networking into our game, we will adopt the traditional client-server model. Our reasoning was that, while less sophisticated, we could successfully implement this model given the time constraints.

We plan on having a server which runs on its own. It will control all the game logic as well as handle requests from clients. We want to make the networking architecture scalable so that there are no limits to the number of players that can join a game. This will require careful planning and implementation but it will allow many people to play the game together, increasing the fun to be had from the game.

An extended feature we want to add would be to allow the server to load variations in the level. So, for example, the server could load a cooperative version of a level and force players to work together to accomplish a goal. This was seen as a difficult feature and kept as possible extension for the end.

Chapter 3

Design

3.1 Overall Architecture

When deciding how to structure our program, we wanted to make our code clean and minimal. By creating code that was as self-contained as possible, we could ensure that unrelated parts of the program could not affect each other (Hunt & Thomas 1999). This was an important requirement because we all needed to be able to work on the code simultaneously and minimise how much we interfered with each other.

We also ruthlessly refactored our code. This might have been a difficult task but our functionality was defined in small, easily-modifiable chunks. For example, when it came time to move the game loop to the server, the only refactoring needed was to change some variable names.

Our code was split into a number of packages. In addition to reducing clutter, keeping classes in packages based on their functionality provided a quick idea of what the class dealt with (Horstmann & Cornell 2008*b*). For example, if someone was looking through the code and came across the `VelocityVector` class, they could quickly get an idea that it had something to do with how players were defined.

Overall, we feel our code is clear and concise. We believe that it would be possible to understand nearly all of the game mechanics with a quick read. This was not an easy goal to achieve. It meant delaying the implementation of features until we really needed it and knew how it should fit with the rest of the code (Jeffries 1997).

3.1.1 Players

All players in the game were instances of the class `Matter` or one of its subclasses. In the version of the game we created, all of the players were simply perfect circles which moved through a two-dimensional space.

We also favoured composition over inheritance in the implementation of the `Matter` class. The players were essentially circles with some added features. It made sense to use `Ellipse2D` to define most of the behaviour. Rather than extending `Ellipse2D` and adding our features, we chose to create an `Ellipse2D` object as a member field of the class and implement the `Shape` interface instead.

This may seem a verbose way of achieving the same goal but it allows more flexibility in the future. We could easily create a new subclass of `Matter` which used a different shape as its underlying structure (Bloch 2008).

The subclasses of `Matter` only needed to override a single method to create new behaviour. During game play, all `Matter` objects are given an opportunity to adjust their movement. By modifying the method which creates this change, different types of `Matter` objects could be created (Bloch 2008). For our game, we successfully implemented the human player and a random player which did not make it into the final game. Unfortunately, an intelligent player was never successfully created.

3.1.2 Levels

Our aim was to make levels easy to create and maintain. This meant storing our level data in as simple a format as possible. We had considered storing the information in a YAML file but due to the simplistic nature of our levels we felt it was unnecessary.

```
# Player one
 200.0,   200.0,   20.0,   0.0,   0.0
# Player two
1263.0,   694.0,   20.0,   0.0,   0.0
```

Figure 3.1: Defining a level

To create a level, we simply list the locations, sizes and velocities of all objects in a plain-text file. An example can be seen in Figure 3.1. This was an easy format to implement. Comments were added to the parser to make it easier for us to leave some meaningful remarks about the level we were trying to create.

Had there been time, a YAML file would have been the preferred approach to create an extensible level structure. This would have given us more flexibility in defining levels. We could dynamically load different backgrounds, enemy types and level styles. While this would all be possible in a plain-text file, it begins to get difficult to keep extending it.

Because creating a level by hand was very tedious, Yukun took it upon himself to create a level editor. He was able to do this by refactoring code from an assignment in the first year. Yukun modified this class so that it could read and write level files in the format the level loader could understand.

The level loader needed the added ability of being able to load a specific type of player into the game. The way it worked was when a player connected to the server, the player would request their player number. By doing this, the player's handler thread would load the player into a specific position in the game and send the number back to the player. The level loader itself was a singleton object. In order to enforce this constraint we simply made its constructor private (Bloch 2008).

To load and maintain game sprites we created a `SpriteFactory`. Our design was inspired by the tutorial created by (Glass n.d.) and by (Gamma, Helm, Johnson & Vlissides 1994). Images would be fetched from disk only once and from then on maintained in a hash table. All future access would be instant and would speed up the loading of images in the game. This feature did not

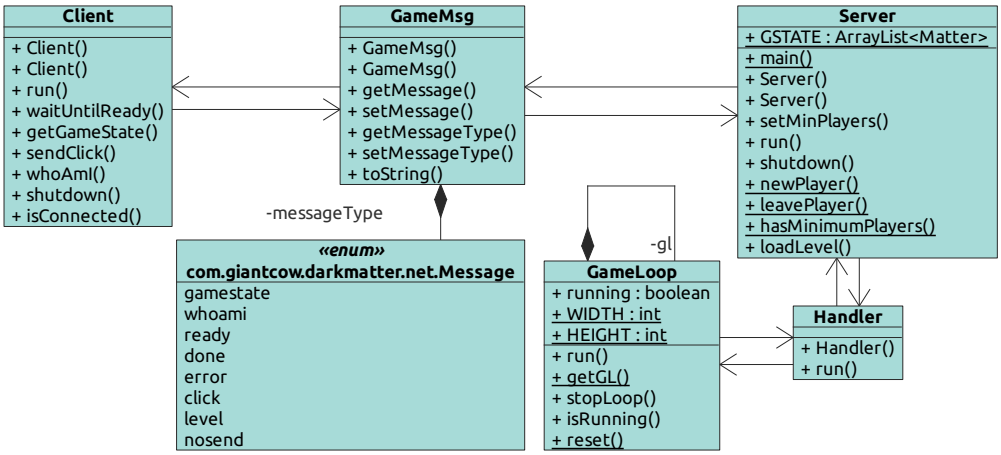

Figure 3.2: Networking Architecture

get much use in our game because we never got to create sprite images for the entities in our game.

3.1.3 Networking

The networking architecture is based on the client-server model (Davison 2008). This was a relatively straightforward system to design and implement. Our networking code places the game on the server side of the networking stack and clients connect to it. We chose this design, as opposed to a peer-to-peer design, because it freed us from having to deal with synchronisation between multiple peers.

The system we designed ended up being very flexible. There is no limit from the server to the number of players (the max tested was six) that can join a game. The only limiting factors are map size as well as system resources.

Our networking architecture can be see in Figure 3.2. The client and server can only communicate to each other through a predefined message object. Ensuring that the two halves of the networking stack interact in a sanitised and safe way helped us prevent the subtle errors that arise when sending message as Strings (Bloch 2008). For example, if we had encoded a message type as a String, a simple misspelling would have caused the system to throw and error instead of responding in the desired way.

Messages

The `GameMsg` class was a simple wrapper class. It contained data (if needed) and a message header. The header was an `enum` which simplified the processing code on the server side. Based on the message header and possible data, the client or server can perform a requested action. Figure 3.3 shows the possible messages that could be sent.

The client begins a session by sending the `whoami` message. This causes the server to load a `HumanMatter` object into a game and return to the user a player

number.

Clients implement a blocking wait. This means that the game would wait and not allow anymore execution of the client-side code until there are enough players. Accomplishing this is done with the `ready` message. Clients simply keeping sending the message and as soon as the server sends `true` back, client-side execution can begin again.

The `gametstate` message allowed the client end to query the server for the current state of the game. The server would then respond in a thread-safe manner with the current state of the game.

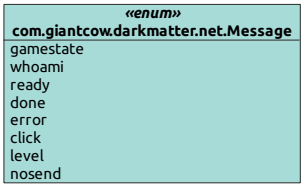

Figure 3.3: The Message Protocol

Clients can affect the game state through the `click` message. This is a message which contains the location of the user's last click. The server can then use these clicks to affect how the corresponding `Matter` objects move in the game loop.

The `level` message would request that a specified level be loaded for game play. This was to be used to allow multiple players to negotiate a level but due to time this feature was not implemented.

Finally the `done`, `error` and `nosend` messages were special messages that would allow either the client or server to deal with errors or prevent a message from being sent. The use for these was dealing with broken connections and the end game state.

Server

All game processing happened on the server side of the networking stack (Davison 2008). We wanted the server to not be tied down to any number of players so we decided to use threads to spawn special handlers for each client that connected. This also benefited our system architecture because it kept separate the code to run the server and the code to deal with client requests.

The server hosts a global variable which is shared between all client handlers and the game loop. Allowing everyone to use the same data structure ensured that all players were playing the same game. We simply had to ensure that all access to the shared global variable was thread-safe.

Additionally, the game loop ran on the server. Whenever the minimum number of players for a type of game had been satisfied, one of the client handlers would ensure the level was loaded into the game loop and the begin the level. We based our game loop on the singleton pattern (Gamma et al. 1994). This was a natural fit because we wanted exactly one copy of the game loop to run on the server at a time (Bloch 2008).

Rather than duplicate our code (Hunt & Thomas 1999) and have two copies of the game loop for different modes of game play, we opted for the more minimal approach. When the user wants to play a single player game, a server on `localhost` was implicitly created. The user was essentially playing a networked game for one on their own local network. This saved us from having to maintain two branches of identical code.

Client

Our `Client` class handles all details for the user. This class implements the entire protocol of messages specified in Figure 3.3. Our aim with this class was to create a single point of contact with the server so as to reduce its impact on the rest of the code.

During our initial attempts at running the game in a networked environment we were plagued with latency. The client side would query the server for the state of the world whenever the client needed to repaint the screen. Due to the time it took for a request to be sent and a response to return, there was a noticeable delay in the game (even on `localhost`).

To solve this problem we modified client to run in its own thread (Horstmann & Cornell 2008*a*). While running in its thread it would be querying the server for the game state constantly. This solved the problem of getting the state of the game on time but it created a new one. Our code at the time would send a message containing the location of a click immediately after it happened. Because of this, our client code would try and send a message while it was sending a message. Our solution to this was to modify how clicks were sent. A user's click were added to a queue in the client. By adding a user's click to a queue and sending all stored clicks at a regular time, we were able to overcome the problem of corrupting the I/O streams. Our final solution to the problem looked roughly like Algorithm 1.

Algorithm 1 Client networking loop

```
done ⇐ false
while ¬done do
  {Send all clicks added since last send}
  sendClicks()
  {Get the new game state}
  update()
end while
```

3.1.4 Polish

Creating the *experience* of the game was also an important consideration. Because of this, work to create a music player began at the start of our development cycle. The music player was designed to be completely decoupled from the rest of the system. Because it could run in its own thread, we could start the music player at any point in the code which allowed us to experiment with it while we were developing our code. It was important to have music that gave the right feeling and after some searching, Joss found (DJ Sasha, Orbital & The Chemical Brothers n.d.) and (MacLeod n.d.).

The visuals were also important for the game. Because of this we spent a lot of time working on code to allow the user to zoom in and out on their player. This made it so the user could navigate around cluttered areas with ease. Since this bit of the game had no impact on the logic of the game, it was implemented on the client's side. Each user would want to have their own level of zoom so

this makes sense. As a result of our design decisions, the client was simply a listener for the game state which would paint any updates that happened. The most clients can interact is by sending a message containing the location of a click or a request for a level to be loaded.

3.2 Division of Work

Our team had an informal approach to assigning work. As a team, we all had our own aptitudes and desires. Fortunately, we were able to largely assign work this way. There were only a few occasions when multiple people wanted to work on the same component and that was resolved quite easily.

In order to get an idea of how big everyone's contributions were, we played the planning game. This was a useful exercise. What would happen was that all the features for an iteration were placed on a table. We each then estimated how long it would take us to complete that particular feature. This was not based on an actual time value but on the time it would take to implement a trivial feature in Java. Our estimates would then be averaged for a final expected time. This meant that those doing smaller features were expected to finish earlier and then begin contributing a new feature to the code base.

Chapter 4

Validation and Testing

Testing and validation are important aspects of extreme programming and we did our best to remain faithful to the traditional approach. Our testing strategy never became very complicated; there was not enough time to develop a more comprehensive approach. Had there been time, we would have liked to add integration tests to our project.

4.1 Testing

Because our team adopted the extreme programming methodoloy, testing was an important part of our development process. Our pair programming sessions usually consisted of three stages, the second one being the creation of unit tests for the code we just wrote. We took advantage of JUnit in order to create a series of consitent tests. Our aim was to have as much test coverage as possible. It was not about simply creating a test for each method but instead ceating a test for each behavior we intended the method to have. It is for this reason that we have multiple tests for some methods.

Our testing strategy was quite simple. We created test suites at the package level and tested all the behaviour we wanted our code to be able to deal with. This meant identifying potential weak spots in the code and testing for it. Our process fell short of the test-driven approach of writing tests first but we still did a good job of creating tests to make our code fail. We envisioned every way in which our code could potentially fail and wrote a test for it. This meant having a lot of failling code at first but as we refined our methods and assumptions we gradually passed more tests until we passed them all.

Testing the whole of our project was done in a visual and interactive way. Part of this was because we did not create integration tests to test the behaviour of multiple subsystems working together. Integration tests were outside the scope of our assignment but we still regret not exploring and implementing them. The other reason we had to test our game interactively was simply due to the domain in which we were programming. Games present a challenge for automated testing. While it would be possible to test some behaviour with Java's GUI robots, it is a lot harder to put a test into numbers when we are aiming to test the *feeling* of the game. For these kinds of tests we simply had to

play the game a lot. This was not a problem, though, as we all enjoyed playing our game. The added benefit of play-testing the game was that it would excite us to add or enhance game features.

4.2 Validation

Validation was a very informal process for our team. Since we were the kinds of users our game was intended for, we were able to determine if the software was headed in the right direction. Because of our weekly release process, we could review the current state of the game at each meeting. By each meeting we had coded our respective parts and were now all together, we could voice opinions on the way the software worked. This was done multiple times throughout the development of our game.

Weekly validation was critical to make the zooming and scrooling features of the game work well. When discussing the feature and how it worked, we undoubtedly envisioned subtlely different variations as to how it would look. After a prototype was created we all talked about it. This input allowed a refinement of the behaviour that we all felt worked well.

A similar situation happened with the absorption of matter objects. It took many iterations to get the absroption behaviour exactly correct. There were many details to consider, such as absorption rate and change in momentum, whch made working out the exact behaviour we wanted take a long time. Each week we were able to see a slightly modified absorption behaviour and by voicing our opinions it reached its final, excellent state.

Chapter 5

Project Management

At our first meeting we discussed the required roles needed for a successful team. We decided that our team needed a *Team Leader* to guide the project and make executive decisions when required. Additionally we decided we needed a *Team Secretary* to keep the administration work up-to-date and to organise meetings and liaise with demonstrators and module coordinators. All members of the team were also designated as *Software Developers* to emphasise that we were all there to write code together. Our team structure was: *a*) Joss Greenaway as Team Leader, *b*) Charles Horrell as Team Seceretary, *c*) Yukun Wang as Software Developer and *d*) Jeremiah Via as Software Deveoper .

We decided to have one meeting a week in addition to our meeting with our demonstrator, Katrina Samperi. This was deemed necessary to keep the team in constant contact and decide on weekly and long-term goals. We made ourselves a Facebook group to maintain contact between the team.

Minutes of all meetings were uploaded to Subversion to keep the team abreast of weekly goals. Weekly reports were produced containing our weekly goals as well as a review of the past week. These were extremely useful in assessing and evaluating our own performance over the previous week and scheduling work for the week ahead.

We also scheduled pair programming sessions three times a week as shown in Figure 5.1. This was one of our favourite aspects of the agile methodology. Because of pair programming sessions, we were all aware of design decisions and how they impacted code. We were also able to find bugs quickly and create better software overall. Our pair programming sessions allowed each of us to program with everyone on the team. This was great because it gave us the chance to work with others of differing abilities and to be exposed to different ways of problem solving. Our three sessions were organized as follows:

	Tuesday	Thursday
10:00	Team Meeting	
11:00	Sessions One	
12:00		Session Two
13:00		
14:00		Session Three
15:00		Demonstrator Meeting

Figure 5.1: Weekly Meetings

1. Joss and Jeremiah
   Charlie and Yukun

2. Charlie and Jeremiah
   Joss and Yukun

3. Joss and Charlie
   Yukun and Jeremiah

The Tuesday morning meeting was mainly used for deciding on goals for the week ahead and reflection. There were many features which we wanted to get into the game. It is during this time that we would decide which features were the most critical. By defining the critical goals we were one step closer to actually getting a working game. We would also use this time to reflect on our performance over the past week and discuss any issues that had arisen in the code. Additionally, we used this time to give feedback on other's code and suggest possible improvements or altenatives. Charlie would take minutes of our meetings which would then be available for reference throughout the week. This was useful for ensuring that we were all working towards the game goal.

It was useful having a pair programming session immediately after our meeting. The meeting would get us excited to add new features to our game. It was often after a meeting that a state of "feature clairty" would posses us and cause us to become extremely productive. During these sessions we traditionally wrote a class file, as many tests to make it fail as we could think of, revisions to make the class pass those tests and then the documentation necessary. This pattern caused to implement the bulk of our code and make sure that everyone worked together to make it.

Our meeting with Katrina was used for evaluation and feedback from the previous week. These were meetings where we would show Katrina the current state of our game and solicit feedback from her. She would also give us ideas on how to plan and when to begin woking on aspects of our project. Having her guidance was immensely helpful because it functioned as a verification of our process. During one week in particular, we all fell behind in our game due to other deadlines. Katrina was quick to remind us that we have a finite amount of time to finish this game and that we must always be spending a set amount of time on it. With her input we were able to maintain an overall forward momentum on our game and anticipate problems that would arise in our software.

5.1 Subversion

We used Subversion a lot and considered it a tremendous asset to our development process. Our repository was organised the traditional way with three folders for the `trunk`, `tags` and `branches` of our code. We never actually made use of the `branches` directory due to one team member's bad experience merging different branches of code. Our goal was to create a tagged release of our code each week but we fell bhind considerably on this goal. It was hard to remember to tag a branch instead of continuing in the development process.

We all had different styles of using Subversion. Some committed to the repository every time they had a small but functioning block of code whereas others would wait until they had a more substantial amount of code to commit. Fortunately, this was not hard to work around. All of our IDEs had built-in tools to resolve the conflicts that arose so our differnt styles never created a real impedence to our development.

Subversion can become complicated when all the team members begin committing their IDE specific files and the output of the buld process. Fortunately, we were able to use Subversion's `svn:ignore` property to prevent these kinds of files from being added. The basic criteria for adding a file to the ignore pattern was if it was automatcally generated from a tool. This ensured our directories were clean and prevented the sorts of conflicts caused by build files.

Subversion was used to host all of our documentation as well. This meant that meeting minutes, schedules, mock-ups and other miscellany we kept in one conveninent location as reference for the whole team.

In order to make our lives easier, we imposed some of the rules on our repository suggested at the beginning of the module. First, we made it a rule that no code could be submiteed to the repository which failed any unit tests. Lastly, all code commits needed a message explaining *why* these changes had been made. Combined, these two rules ensured that we had no trouble using Subversion over the course of our project.

5.2 Maven

Another critical component of our build process was Apache Maven. Maven is a build tool for Java which works at a higher level than Ant (Apache Software Foundation n.d.) . It also imposed a set of best practices on the development process which aided in our development. By doing things the Maven way, we were able to get a lot for free.

For example, a Maven project is created with JUnit as a depenedency. This means that any of us can immediately begin working with the code and run unit tests. We never had to worry about downloading the right version of JUnit and adding it to our project manually. Maven installs all dependencies to a local repository on each of our machines. This means that we can keep `jar`s out of the repository which saves space. In fact, through our whole development process we never had a `jar` archive in our repository. There was simply no need because at any time we could run `mvn package` and one would be generate for us after passing all of our unit tests.

Rather than deal with the incompatible build files for a number of different IDEs, we were able to simply run the Maven project as-is. All IDEs have the ability to work with Maven projects, either built-in or through an official plug-in. This meant that we could develop according to the Maven life-cycle in any IDE. This freed us from fighting with IDE-specific build files and meant that we could all have our own settings.

Maven can generate site documentation and Java documentation as well. This was very useful. We always had a version of our site and Java documenta-

tion hosted on the internet. Because of this we always had useful information about our project. Maven's site documentation includes useful components like a page to check the code style against that of Sun's Java Guidelines. It would also give statistics on the unit tests such as coverage and how many were passing.

Overall, Maven made our lives a whole lot easier and we heartily recommend teaching it next year the same way JUnit was taught this year.

5.3 Software Engineering Approach and Principles

We decided that our main software engineering approach would be *extreme programming* which is one of the agile programming methodologies. This methodology focuses on using many short release cycles. These cycles include producing a small feature set during each iteration, producing the relevant unit tests and documenting it for our own future reference. It also places an emphasis on pair programming and other strongly cohesive team activities.

The benefits of this methodology to us were numerous. For example, the short release cycles favoured by extreme programming were convenient for us because we only had nine weeks to produce the project. This gave us nine iterations in which to plan features and distribute the work required for the game. It also favoured our approach of weekly meetings and progress reports. This meant that for the Thursday meetings with our demonstrator, we had a new iteration of working code to show. It also suited Subversion because we could have a lot of smaller commits.

By breaking the project up into these weekly cycles it also allowed us to spread the work as evenly over the term as possible. This was better than having to rush to finish and test the game at the end. By constantly testing our game and using unit tests throughout our development, we could be sure that the code we produced was working as we expected. This was better for our small team than using a non-agile method such as the Cleanroom method which only would have slowed us down. This is because we would not be able to guarantee until the final stages of testing that our game will worked how we wanted. Our agile methodology meant that our project was low risk.

Another advantage was the fact that extreme programming builds from the bottom up. We were able to quickly get a simple prototype of the game working early on in its simplest form. Each week we could add features and polish to the game. This fitted in with our schedule and short deadlines as it ensured that we built on the game, week on week, rather than trying to develop some extremely complicated features that would in the end not be implemented. It meant that if we did not meet all our aims and implement all the desired features our game would still be working to a high enough standard.

We also liked the integrated teamwork associated with extreme programming. Our team wholeheartedly took up the practise of pair programming. We met for sessions at least three times a week as detailed in 5.1 above. These were useful to the team as it allowed everyone in the team to contribute code and work with every other member of the team. This ensured that everyone

kept in touch with one another and was aware of all aspects of the project. It also allowed for our stronger coders to help our weaker ones. It gave everyone a chance to work on each section of the code and created bug-catching redundancy because there were more eyes on the code. With the exception of networking, no single person was assigned an entire task to complete on their own so there was always someone else ready to help with coding or solving a problem.

Testing was an important part of our software engineering philosophy, we aimed to cover as much of our functionality with unit tests as we could. We made it a criteria that everyone submit unit tests with the code they created. For this reason, we had one unit test for every 80 lines of code (including comments). We feel confident in our code because were able to ensure our code worked consistently for all edge cases we could think of.

5.3.1 Other Methodologies

We considered other software engineering methodologies early on in the development cycle. After some research and debate, we settled on extreme programming. Here were our evaluations of other techniques which we decided not to use.

Initially we had considered Scrum (Sutherland 2004) but decided it was not appropriate for our circumstances. We did not have a product owner which made using the technique difficult (although it would have been possible to use Katrina for this role). Also, our team was a smaller size than the traditional Scrum team and we wanted to have a shorter release cycle. A key reason this approach would not work for us was the daily meeting. Since we all had other modules we could not work out a way to meet everyday. These reasons caused us to abandon Scrum as a contender for our development process.

We also discarded Waterfall because of the uncertainty involved and the need for a long design phase at the start which went against our natural desire to start coding as soon as possible. There was also the danger of not being able to test our code until the end of the project and the fears that it would not work. We were also advised against this methodology by many guest lecturers for the course. We were also not entirely certain at the start of the project what we needed to do. Because we had to do some exploratory programming, the Waterfall approach of big design up front would have been ill-suited to us. Having to stick to our initial conceptions of what the project should be like would be extremely constricting whereas an agile development process would allow us more freedom and flexibility.

5.4 Release Plan

We adopted a modified extreme programming cycle; we would have a new release each week. At the beginning of each week, we would have a few features that we wanted to add. That entire week would be spent on coding them and making them robust and well documented. At the beginning of the next week, we would review what we did well and poorly over the last week and see how

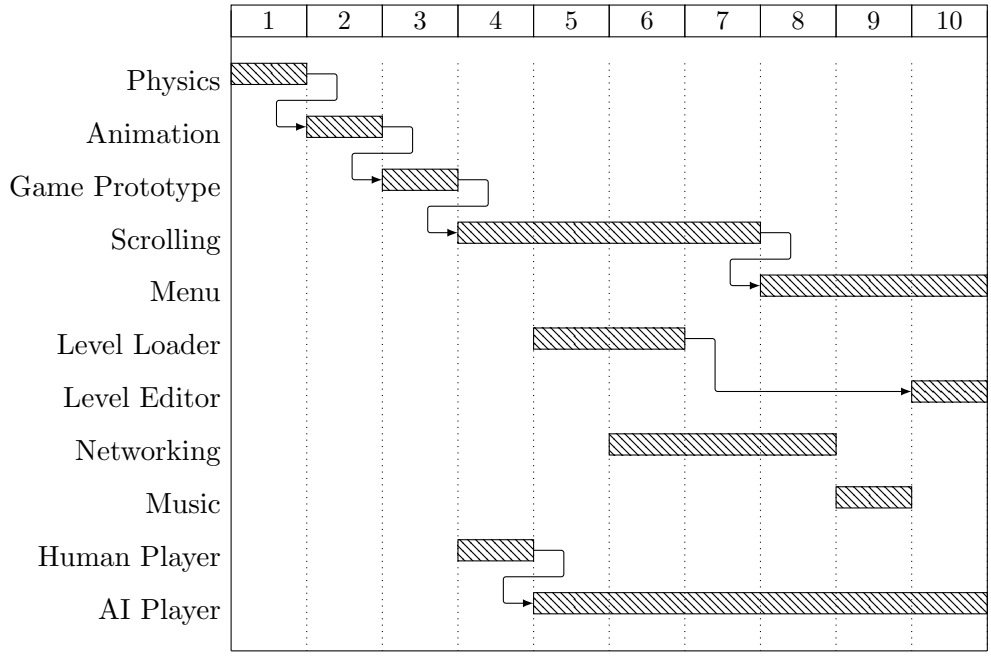

Figure 5.2: A Retrospective Project Schedule

we can improve our processes. Our release cycle worked like the one shown in Figure 5.2.

This approach allowed us to have a working and playable game at every stage of development. By gradually adding features on top a well-tested and documented core, we could be more productive. Less time needed to be spent fixing broken code. Our aim was to have a stable, minimal working product every week. We also wanted to create a tag each week with what we accomplished and use this as a frame of reference to see how our code is progressing. Unfortunately, we did not quite live up to that desire.

Chapter 6

Discussion

This was the first major programming project any of us had ever undertaken in a team. As with all new experiences, we made a number of mistakes but learnt a great deal from them. This chapter will detail some of our shortcomings, both personally and as a team, and the lessons we learned from this whole experience. We are all better programmers and teammates because of this module and we feel grateful for this opportunity. We did not follow all the best practises for working in a team but did an admirable job nonetheless. Each of us failed in some capacity but we all learnt from it. What follows is the result of a lot of self-critical reflection. This section is not meant to give the impression that we were a terrible team, in fact we all still like each other, just that we made a number of rookie mistakes which did have *some* impact. Overall we worked well as a team and we are all proud of the work we created together.

6.1 Team Mistakes

During the development of our game we made a series of missteps which hindered our development process. None of the mistake were egregious or even overt at the time but after spending some time to reflect on the module, we have been able to recognise some ways in which we could have done better. Among our mistakes were a lack of communication, ambiguity in responsibility for a particular feature and the failure of completing a task on time.

We were plagued by communication issues throughout the project. Our team used Facebook as a means of communication and in retrospect it was a mistake. Our group Facebook wall is almost entirely populated by the communication of one person. Facebook is a passive stream of consumption, i.e., users do not respond to most input they receive on it. This pattern of use does not lend itself well to a team project where input from all members is important. As a result of using Facebook, one person would post questions but never receive responses. This got in the way of good communication and hurt team morale. On the next project, alternative means of communication will be used.

Another mistake was the lack of a rigid break up of work requirements. This was in part to our agile philosophy of team code ownership and the responsibility of everyone to work on every class. It, however, resulted in a lack

of understanding about who had to produce an aspect of the software. It allowed us to be apathetic about the project as a whole at times, such as week six when the majority of the team prioritised the Software System Components assignment. In the future, stricter assignment of work would be beneficial.

It is worth noting that our team leader Joss disappeared midway through term. Obviously, losing a team member has a negative impact on a team's ability to complete work. It was also a confusion situation because we never heard anything from him on the matter. He just disappeared and we had to find out through his friends that he had become ill. The main problem from this situation was that our team was left without a team leader. No one ever took the place of team leader and so our development happened in a more ad hoc manner. This is something we should have immediately rectified by chasing him down on his university email and reassigning the position when it became clear he would not have been returning.

Our team was also negatively impacted by the failure of some members to complete their assigned tasks on time. While this was rare, it did mean that others had to pick up the slack and start working on parts of the code that they were not meant to be working on. Having to do another's work prevented some features from making their way into the game. This is an issue which we should have addressed at the beginning. We had trouble with this because it is hard to admonish people we like. We have all learnt the importance of ensuring that team members complete their task. In future projects, we will all be more forward about the failures of our teammates to complete their work.

6.2 Individual Mistakes

6.2.1 Charles

This was my first time working in a team in a software development environment. While familiar with the processes involved in large team projects academically, I was less prepared for how this would relate to a software project. I made quite a few mistakes over the period which I have managed to learn from.

My main mistake was not maintaining a constant work rate throughout the term. I prioritised some other modules above this module and particularly around the time I was doing my communication skills presentations and essays I was not able to offer the input to the team java module that I should have. This is a valuable lesson for time management and work planning. In the future I will plan for these thing and anticipate it and make it up to my team by doing more work in advance of the time I am going to be working on other projects.

Another mistake I made was not getting as involved in the production of software as either Yukun or Jeremiah. While I did write code for the `VelocityVector` class, the AI player, unit tests and produce some levels I did not really write enough code overall. This was because I took my role as team secretary too literally and perhaps put too much emphasis of my time on producing team meeting minutes, weekly reports, user stories and working on the final report than writing code. In future projects I will work on writing more code.

Another mistake I felt I made was not asking Jeremiah, the most experienced coder on our team, for help with writing more code. I was afraid that my inadequacies as a coder would show through and I was embarrassed by this. I should have utilised the pair programming sessions more fully in the later weeks of the project as in the first six weeks to help me code and learn from these two excellent coders. I learnt from this to always make use of such excellent resources in future projects.

6.2.2 Jeremiah

While I feel I worked well on a team, I did make a number of mistakes over the course of our project. These were lack of leadership, the inability to correctly deal with a teammate's missed deadline and difficult in sharing code.

In retrospect I should have taken the leadership position of our team. I had spent the previous term reading books on teamwork and wanted to apply some of what I had learnt. I had heard of Joss' reputation for partying and disregard for work and I had hoped to stem this behaviour. So I applied a technique I had read in a book by (Carnegie 2006): "Give the other person a fine reputation to live up to." So I figured by making Joss team leader he would live up to the leadership reputation. It did in fact work for a good portion of the term. But, unfortunately he disappeared and never gave us any indication of what had happened to him. I should have taken his role as soon as a week had passed. This would have prevented our team from remaining in decision paralysis for son long.

Another mistake I made was not properly dealing with other team member's uncompleted work correctly. I did not do a good enough job of chasing them down and making sure they completed their task; getting the demonstrator involved if nothing had changed. Instead, I would take it upon myself to finish their task for them. This helps no one on the team. It delays progress because my time is spent doing other people's work and it give the other person a pass when they need to understand that their contribution is important. In the future, I will make an effort to make sure all my team members contribute evenly.

I also had a hard time sharing code. When programming, I tended to be pretty controlling about the way the code would function. This made pair programming sessions frustrating because I wanted to take control and immediately begin typing what I felt was the correct way to proceed. Overall, the pair programming sessions did help me with this feeling. I am now a lot more open minded and patient when it comes to coding with others.

6.2.3 Yukun

This is my first time to do a team software project. Although the final result looks pretty good, I still made some mistake in the process.

The main thing is team work. To be honest, in this project we worked in a team, but not as a team. The team members worked separately although at beginning we planned pair programming. And when we lost the team leader,

we did not point out a temporary leader which is one of the reasons why our project was so slow.

Secondly, I did not have a good communication with the other team members. My poor English made me ashamed to talk and explain my ideas. And then since we had the whole term time to finish this project, I let it fall behind. Especially in the middle term, my schedule got bogged down which made me less enthusiastic about coding. Another mistake I made in the project was not fully utilizing the Junit test. Due to a lack of experience, I always wrote JUnit tests after programming instead of before. So when I was coding, I was confused about the target of each method.

In this project, I learned a lot things I never learned before. I learned how to user version control software by using subversion. And thanks to Jeremiah, we had Maven which made our work process convenient. Meanwhile, I learned how to use JUnit test to set goals for programs. On the other hand, team work is not as simple as I thought. I should communicate better with team members and when a problem occurs the team needs to make a decision in time.

Chapter 7

Conclusions

Over the course of the past term, our team implemented an exciting game from start to finish. Doing this exposed all of us to new facets of software development and teamwork. We are all grateful for the experience and we will be able to apply the these lessons to future software projects.

We talked about our specification. It changed as we gained more knowledge in the domain of games development. As time went on, our specification became more exacting until it turned into what it is now. Our specification talked about features we chose for the game and why we felt they were good for the user experience.

The next chapter talked about our design. This space was used to talk about the design decisions when made and *why* we made them. We talked at a high level of abstraction about important components of the overall system architecture, most notably the networking architecture.

After the design we spoke at length about our testing strategy. We had good test coverage and this helped us know our game was working correctly. It also gave us the confidence to implement new features without breaking old ones.

In the discussion, we spoke about our failings as a team and individually. None of these failings were enough to warrant demonstrator involvement nor was their impact too great a burden for us to create the great game we did. This section was an opportunity for each of us to share ways in which we failed and the lessons we learned.

This project was a great opportunity for each of us to become better programmers, problem solvers and teammates. We have learned skills ranging from version control to the planning game that will ensure that we will be more effective on our next project. Each of us are now more equipped to handle big projects in the future either alone or on a team.

Appendix A

User Stories

Instructions We should have a how to play section on the main menu as an option besides playing the game. So after the program initially starts up the user should be able to click on the how to play button and be given a set of instructions on how to play the game or possibly if we have the time to implement it a tutorial level.

Single Player We should have an option to play a randomized level with a single player and an AI player on a single machine. This could possibly be labeled as play random map.

Campaign Mode We should have an option to play through a set of pre designed levels produced by us and in un lockable order portraying the storyline we came up with in our specification and initial meetings.

Networked Multiplayer We should have the option on the main screen to play multiplayer game this should then allow the player to start up or connect to a client server run by the other player.

Multiplayer Instructions We should have a help section maybe contained or containing the how to play section that outlines exactly how to host or connect to a game.

Quitting a Game The user should be able to exit to a menu by pressing escape and then from there should have a button to exit Darkmatter.

Full-Screen Gameplay We should have a resizeable frame that scales up or down all the objects contained in it accordingly at the press of a button to allow for play in full screen and windowed mode.

Appendix B

Meeting Minutes

B.1 Group Meetings

Meeting 1

20 January 2011

Present

1. Joss Greenaway (Team Leader)
2. Charlie Horrell (Secretary)
3. Jeremiah Via
4. Yukun Wang

Summary

During this meeting we spent some time getting to know each other and confirmed the appointment of Joss and myself as officers. The main content of the meeting was talking about possible game ideas. Ideas proposed included:

- An physics game styled on Osmos with added features and originality. We decided this was our primary idea and Joss came up with the project name of "Darkmatter" for the game.
- An Osmos style racing game using the Osmos propulsion mechanics to race blobs around an obstacle course.
- A tower defense style game possibly involving zombies and a cooperative mode.
- Fruit ninja.
- Fall down: a two player game involved in a race to see which player can fall down quickest.

Once we had decided on Osmos as our primary idea and zombie tower defense as our fall back, we considered ways to put our own spin on Osmos. These included the addition of weapons to the right click, such as an antimatter weapon that would split any other dark matter stars in half or reduce size. And also a weapon to stop the balls in full flow. The main aspect of originality that we add as well as setting it in space is the addition of cooperative and competitive game play. We talked about various modes such as straight up death match style, i.e., just become the biggest player, to other game modes that would require players to work together to take down a larger enemy or maybe a mode where the players are segregated and need to complete tasks with each other to complete the level.

We also touched on the software engineering principles we would employ and this included testing methods such as JUnit and the possible use of pair programming was mentioned as a method as well as other methods such as "Scrum". Jeremiah suggested the use of Maven for our project.

Action

All Research Osmos and Subversion

Meeting 2

25 January 2011

Present

1. Joss Greenaway (Team Leader)
2. Charlie Horrell (Secretary)
3. Jeremiah Via
4. Yukun Wang

Summary

We met for the second time to discuss coding practices and the structure of our game. We decided that we would leave the start of coding till week 3 when we knew how to work sockets and networking into the code as we felt that this would impact our code at every level. We decided on a pair programming schedule where we would each code with each other member of the team for an hour a week. We also decided that we would have this meeting every week. After these discussions and some discussion of the physics engine we decided to head to the labs to get Subversion working and test it out. Jeremiah helped the team with this as he has prior experience of using subversion.

Action

All Research coding structure and Osmos. Submit progress logs to Subversion.

Charlie Produce timetable of pair programming and other Team Java commitments and upload to Subversion.

Meeting 3

1 February 2011

Present

1. Joss Greenaway (Team Leader)
2. Charlie Horrell (Secretary)
3. Jeremiah Via
4. Yukun Wang

Summary

We decided to meet to discuss the impending deadline for the specification and to make sure we were broadly on the same page regarding what to put in it. We also talked over software engineering principles that we were going to use including maven and testing. We also discussed the GUI and the possible methods of animating the dark matter. Yukun also showed the team a small model he had made with an object bouncing around a screen and interacting with the edges. We then created a sandbox in the subversion repository in which we could put experimental code to play around with. We decided on physics that would go into the game that included making the game universe frictionless to make the modeling easier and to replicate space.

Action

All Produce specification and submit to Subversion.

Meeting 4

1 February 2011

Present

1. Joss Greenaway (Team Leader)
2. Charlie Horrell (Secretary)
3. Jeremiah Via
4. Yukun Wang

Summary

We discussed how we were going to approach the pair programming sessions over the coming week. It was decided upon that for the sessions scheduled straight after the meeting that Jeremiah and Joss would work on the matter class and charlie and Yukun would work on the physics class. This work was then undertaken and submitted to subversion. It was raised that charlie and Yukun were not uploading comments when they submitted work to subversion and this should be remedied. Jeremiah and Yukun also showed of various prototypes that they had uploaded to subversion to test out aspects of the game.

Action

Joss & Jeremiah Produce Matter Class.

Charlie & Yukun Produce Physics Class.

Meeting 5

8 February 2011

Present

1. Joss Greenaway (Team Leader)
2. Charlie Horrell (Secretary)
3. Jeremiah Via
4. Yukun Wang

Summary

Firstly we discussed the progress of the previous and team members were bought up to date on the demonstrator meeting from the previous week. We discussed the points raised in that meeting and endeavored to address the issues that came up such as using more scientific English and producing a detailed weekly report. It was decided that charlie would produce this in time for the demonstrator meeting for the following weeks. Jeremiah demonstrated to us the Darkmatter class he had written and Joss showed us the draft of an opening video for the game. We then decided on tasks for the week and ranked tasks that needed to be done in order of priority. The main issues to be undertaken were programming an AI for the other player in the single player mode, start programming the networking, fixing the ejection angles of the balls of matter and introduce an egocentric scrolling view.

Action

Joss Fix ejection angles of matter.

Charlie Produce AI for other player.

Jeremiah Start the client-server networking.

Yukun Develop a scrolling system with the player always at the center.

Meeting 6

15 February 2011

Present

1. Joss Greenaway (Team Leader)
2. Charlie Horrell (Secretary)
3. Jeremiah Via
4. Yukun Wang

Summary

This was an extremely brief meeting where we just decided that we would carry on with the previous weeks goals and that Joss would work on producing the media player. Jeremiah also filled us in on how the changes to the client-server would effect the rest of the game code and helped us install the latest versions of maven on our machines.

Action

Joss Produce media player for the game.

Charlie Produce AI for other player.

Jeremiah Start the client-server networking.

Yukun Develop a scrolling system with the player always at the center.

Meeting 7

22 February 2011

Present

1. Charlie Horrell (Secretary)
2. Jeremiah Via
3. Yukun Wang

Summary

This was another brief meeting where we again decided to carry on with the previous weeks work as it was still in progress. Also tidy up and review JUnit tests for the testing vivas later on in the week.

Action

Joss Produce media player for game.

Charlie Produce AI for other player.

Yukun Develop an scrolling system with the player always at the center.

Jeremiah Start the client server networking.

All Review tests.

Meeting 8

1 March 2011

Present

1. Joss Greenaway (Team Leader)
2. Charlie Horrell (Secretary)
3. Jeremiah Via
4. Yukun Wang

Summary

This meeting we talked frankly about our disappointing progress of the previous few weeks and decided on a course of action that would propel us forward. This involved finishing off previous tasks including working on finishing networking and making AI smarter and scrolling smoother. We also decided that a level builder would be a good feature to improve the games single player mode.

Action

Charlie Keep working on AI; begin working on final report.

Joss Continue working on media player.

Jeremiah Work on networking.

Yukun Create a level builder.

Meeting 9

8 March 2011

Present

1. Joss Greenaway (Team Leader)
2. Charlie Horrell (Secretary)
3. Jeremiah Via
4. Yukun Wang

Summary

This meeting we talked about the things we needed to do to polish off our project. This included working more on the final report, integrating the networking and producing a menu to polish up the game.

Action

Jeremiah Continue working on networking.

Charlie Work on final report and submit an iteration to Subversion.

Yukun Work on menu.

Joss Work on media player.

B.2 Demonstrator Meetings

Meeting 1

20 January 2011

Present

1. Katrina Samperi (Demonstrator)
2. Joss Greenaway (Team Leader)
3. Charlie Horrell (Secretary)
4. Jeremiah Via
5. Yukun Wang

Summary

During this meeting we introduced ourselves to Katrina and pitched our idea of Darkmatter to her. Katrina approved our idea and informed us of the basic structure of the module and the importance of logs,subversion and team work. She also prompted us into choosing a team name! After a brief discussion about Pokemon we settled on team "Giant Cow" because to quote Joss "one of the new Pokemon looks just like a giant cow." The name was catchy yet suitably absurd

and humorous so “Giant Cow Games” was born! We also asked Katrina about the specification that is due in by the end of week two and it was suggested to us that we should not go into technical detail but give a more general idea of what our game will be about and the principles used to code it. We were also warned about pitfalls previous teams had had in splitting up components of the project arbitrarily and then meeting at a later date to merge sections, as this had no fail safe and often required a lot of extra work to integrate components. We also raised the issue of possibly using games libraries or other source code in the programming of the game. Katrina said she would look into this for us and let us know. Katrina also indicated to us that pair programming was an excellent idea.

Action

All Research Osmos and Subversion. Think about content for specification.

Charlie Update Facebook group, minutes and arrange further meetings.

Meeting 2

7 January 2011

Present

1. Katrina Samperi (Demonstrator)
2. Joss Greenaway (Team Leader)
3. Charlie Horrell (Secretary)
4. Jeremiah Via
5. Yukun Wang

Summary

During this meeting we were given more precise instructions on when to submit the specification. We were told it had to be in for 12pm on subversion in PDF format. We also talked about our progress for the week, including pair programming and testing with maven. Katrina approved of these measures. We also decided to drop scrum programming partly because of the roles that different people would have to take on made the process seem convoluted. We were told that creating user stories might be a helpful part of our development process.

Action

All Produce specification and upload it to Subversion.

Meeting 3

3 February 2011

Present

1. Katrina Samperi (Demonstrator)
2. Charlie Horrell (Secretary)
3. Yukun Wang

Apologies

1. Jeremiah Via (*Ill*)
2. Joss Greenaway (*Ill*)

Summary

During this meeting we were given general feedback on our specifications. A major issue raised was the standard of scientific and formal English as well as issues with grammar. We were informed that unless this was improved in the final report it could loose us significant marks. Also raised was the lack of updating the weekly work logs. It was stressed these should be updated every week before the meeting with Katrina. Some clarification was required for Yukun on the non use of the waterfall model as we are using an agile approach. We were also told to finally decided which networking strategy we were going to use. Otherwise feedback on the specification was generally positive. We were also encouraged to produce a weekly report for all future demonstrator meetings.

I raised the issue of the lack of details on the subversion assessment. We were advised as we had well over 60 revisions already we did not need to worry about it for now.

Action

None.

Meeting 4

10 February 2011

Present

1. Katrina Samperi (Demonstrator)
2. Joss Greenaway (Team Leader)
3. Charlie Horrell (Secretary)

4. Jeremiah Via

5. Yukun Wang

Summary

During this meeting we presented to Katrina the weekly report that had been promised at the previous weeks meeting. Feedback from this was generally positive however Katrina requested that it would have been more useful if the weekly report had been submitted to subversion earlier in the day, unfortunately a large part of our timetabled man hours for the project takes place on the Thursdays and so any report submitted earlier in the day would likely be a false state of affairs. Katrina approved of our production of long term goals and organization in approaching problems.

We were also introduced to the planning game which involved different team members approximating the amount of time taken for particular tasks in arbitrary units consisting of the time taken for a simple feature such as a button to be programmed in java. From this we decided that the client server would take the majority of the weeks ahead and as such the team should pitch in to help Jeremiah with the production of this. We also rated the scrolling problem as taking a long time to solve.

Action

Jeremiah Networking.

Charlie AI player.

Joss Fix angles of expelled matter.

Yukun Implement scrolling.

Meeting 5

17 February 2011

Present

1. Katrina Samperi (Demonstrator)

2. Joss Greenaway (Team Leader)

3. Jeremiah Via

4. Yukun Wang

Apologies

1. Charlie Horrell (*Ill*)

Summary

During this meeting the testing vivas took place. Charlie was ill so his took place the following week.

Action

None.

Meeting 6

24 February 2011

Present

1. Katrina Samperi (Demonstrator)
2. Charlie Horrell (Secretary)
3. Jeremiah Via
4. Yukun Wang

Absent

1. Joss Greenaway

Summary

During this meeting we had our prototype demonstration on the project. We received largely negative feedback that our development was stagnant, our game was buggy and that we needed multiple levels in the single player game mode.

Action

All Finish work on assigned tasks.

Meeting 7

3 March 2011

Present

1. Katrina Samperi (Demonstrator)
2. Joss Greenaway (Team Leader)
3. Charlie Horrell (Secretary)
4. Jeremiah Via
5. Yukun Wang

Summary

During this meeting we showed Katrina our improved game with multiple levels fixed scrolling and better AI, it is now also less buggy and has working music. We received largely positive feedback on this point.

Action

None.

Bibliography

Apache Software Foundation (n.d.), 'Maven'. v2.2.1.

Bloch, J. (2008), *Effective Java*, The Java Series, 2 edn, Addison-Wesley.

Carnegie, D. (2006), *How to Win Friends and Influence People*, Vermillion.

Davison, A. (2008), *Killer Game Programming in Java*, O'Reilly Media.

DJ Sasha, Orbital & The Chemical Brothers (n.d.), 'Sasha xpander'.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J. (1994), *Design patterns: Elements of Reusable Object-Oriented Software*, Addison Wesley.

Glass, K. (n.d.), 'Space invaders 101: An accelerated java 2d tutorial'.
URL: *http://www.cokeandcode.com/node/6*

Hemisphere Games (2009), 'Osmos'. v1.6.0.
URL: *www.hemispheregames.com/osmos*

Horstmann, C. S. & Cornell, G. (2008*a*), *Core Java: Advanced Features*, Vol. 2, 8 edn, Prentice Hall.

Horstmann, C. S. & Cornell, G. (2008*b*), *Core Java: Fundamentals*, Vol. 1, 8 edn, Prentice Hall.

Hunt, A. & Thomas, D. (1999), *The Pragmatic Programmer: From Journeyman to Master*, The Pragmatic Bookshelf, Addison-Wesley Professional.

Jeffries, R. E. (1997), 'You're not gonna need it!'.
URL: *http://www.xprogramming.com/Practices/PracNotNeed.html*

MacLeod, K. (n.d.), 'Space figher loop'.
URL: *http://incompetech.com*

Rychlik-Prince, C. & Matzon, B. (2011), 'Lightweight java games library'.
URL: *www.lwjgl.org*

Sutherland, D. J. (2004), 'Agile development: Lessons learned from the first scrum'.