Course Work Cover Sheet - The School of Computing

Author’s Name: Kyle Harrison

Matriculation

Numbers: 110009870

Module Code: AC31009

Assignment Title: Games Programming - Report

Date Submitted: 23/11/12

Date Due to be

Submitted: 23/11/12

Word Count

|  |  |
| --- | --- |
| Your course work may be used by The School of Computing for demonstration purposes with other classes and/or members of the public, unless you indicate otherwise. Tick the box at the right if you do not wish your work to be used in this way. |  |

I certify that this assignment represents all my own work and that no parts of it have been copied and that no collusion has taken place with any other person. I have also read and understood The School of Computing’s regulations as regards course work and I certify that this course work was performed according to those regulations.

I certify that any disks submitted with this assignment have been virus checked and have no viruses on them.

Signed: ………….…..<Name>

Ariyana Games Report

# Design patterns

### State/Strategy

In my design I had State and Strategy for both Units and Enemies behaviors alongside the current state the unit/enemy existed in. However during development I discovered that I was able to combine what I needed from both of these into one interface connecting to the context class and then separate out into multiple concrete strategy classes. The state of the enemy was replaced with a Boolean value to confirm if the unit/ enemy are alive or dead and moving through the behaviors of the concrete strategy provided the functionality needed to fulfill the remainder of the roles. By changing from one behavior to another such as moving to attacking states were no longer needed.

The reason for 2 different implementations is to reflect the AI having control over an Enemy and a user having control over a unit with each behaving differently to each given situation and requiring completely different code. The context classes can be found in the model and the interface and concrete strategy can be found in the controller.

**Context**: Enemy **Context**: Unit

**Interface** IEnemyType **Interface**: IUnitType

**Concrete** **Strategy**: Wraith **Concrete** **Strategy:**,ArmouredKnight

### Singleton

Singleton was used for the Configuration class. Initially in the design, a configuration class was not included however during development it was apparent it was needed to store and maintain a single instance of the graphics device manage as only one instance of a class is required and the instance must be available throughout all the code. Using different machines with different resolutions required the game resolution to be adjusted every time the source code was run from a different machine and problems began to arise from multiple instances of different screen resolutions that caused sprites that were generated dynamically based off the resolution to appear in the wrong locations. This solved those problems. Additionally singleton had been purposed for other classes however as they were not implemented they were not used. The Configuration class is in the model.

**Class**: Configuration

### Observer

One of the design patterns that I had planned to use was the Observer design pattern. During planning this tackled some of the issues that arose during development in regards to management and control of the moving pieces and objects on the game board however due to time constraints this design pattern was not used and instead a solution that produced a working game was used in its place. The solution was to allow access to game pieces and objects statically via the controller classes to gain access to their underlying model. If Observer was implemented the game board manager class would have a greater degree of maintainability however the way in which the game plays to the user would not be effected and would appear no different. In terms of developing the full game this would be essential and would need to be established before the game was extended further. However for a small project such as this where a simple tech demo is all that is required it is much more important to produce the functionality to the user with a game that is playable than something that is only half complete and this is why this specific design pattern was note used in order to create a working game.

## Deviations from plan

During my time management I had planned to produce a playable version each week by producing a prototype version and building upon it however I had to deviate from the plan greatly due to problems within another module with a large portion of absent team members resulting in greater time being required to spend working on other assignments. However to combat this I produced almost daily versions of the game with minor incremental changes. Often something simple such as adding movement to one type of sprite and then testing it worked correctly before moving forward with another task. Essentially this was a boon to the game as by taking incremental steps once something was coded and working it often did not require any further changes other than adding additional functionality to the game. Although this resulted in a much later playable version of the game than first anticipated it did ensure that each daily version of the game was a stable release.   
  
With a reduced amount of time available for the game I found that I had to reduce the game greatly from an entire game down to a single level of the game otherwise called a phase. This meant that classes were either removed or only practically coded to provide the minimum functionality required to make the game playable.

Additional changes to the design included some classes behaving as the model, controller and view. This change was intentional as XNA already uses a MVC framework however it was not the strict MVC design I had planned to use. The reason for this was that is reduced the number of classes in the application whilst still allowing the model, view and controller all to remain separate within the game. This greatly helped reduce on the development time and allowed for much easier changes to the code.

### Testing Strategy

The testing strategy used for Ariyana was a simple one. By taking a rapid application development approach to the game development the game went through several iterations and with each the game became slightly closer to producing the end product. Constant testing ensure that whenever new functionality was added it was tested almost immediately and then once it was proven to work correctly it did not need any further changes. This ensured any changes to the source code could quickly be identified that caused an error or exception allowing each to be altered or removed to ensure the game could handle resolve the issues and remain playable and stable at all times.  
  
Within the final days of the project some user testing was carried out by a variety of users both that had an understanding of how the game worked and those that did not. Each was asked to play the game and during their second attempt at the game they were asked to try and break it. This highlighted issues with performance such as slow to change menus or exceptions when a user pressed a button that was not supported that caused the game to crash. All fatal exceptions identified were resolved and any minor issue in the game that previously caused it to halt or come to a grinding stop was resolved. An example of this is when units could collide with each other the game would come to a grinding stop. This was an issue as some players would often attempt to place a unit on top of another. This was resolved to ensure that if a unit does collide with another that the game will continue to run and it allows the user to undo their mistake manually. The result of this strategy of daily iterations that included testing alongside some user testing ensures the game was as stable as possible by the end of the deadline.