# How was the program designed?

## Worlds

The Worlds act as different scenes within our program, each having its own OpenGL callbacks for:

* Special Function – Movement Keys
* Special Up Function – Release Key
* Keyboard Up Function – Release Keys
* Keyboard Function – Keyboard
* Display Function – Display
* Idle Function – Idle
* Mouse Function – Mouse
* Motion Function – Mouse Move
* Reshape Function – Reshape
* Passive Motion Function – Mouse Move

The *main.cpp* file sets up these callbacks and allows for changing between worlds. The main function has a member variable of type World\* which keeps track of the currently activated world. They can be switched between using the variable *currWorld* variable available in the World abstract class which records which World to change to.

Our program has 3 worlds:

1. Menu world – Handles the menu system which allows the user to choose between Shays world and the Game world or to exit.
2. Shays world – Handles all of Shays simulation
   1. Shays world was transformed into its own class and made to inherit from the abstract class World. This way we were able to keep is separate from the Game world.
3. Game world – Handles all the Game world actions.

### GameWorld

The GameWorld uses game objects (GameObj) as the main interactive components in this world thus it has its own objects for the island, player and the collision sphere around the mountain on the island. It also manages the output for the GUI. When the AI system was first written, the GameWorld class was also responsible for managing the spawning of AI however this was since moved.

When the GameWorld is initialised it sets up its lighting, sound controller and game objects. Each display loop it calls the display functions for the game objects it manages after setting their positions, angle and scale.

We did toy with the idea of using the factory design pattern to create game objects however there was too many problems involved in the development of this system for it to handle just 3 objects in the game world so this was abandoned.

However a variation of the factory design pattern is in use in the projectile manager and AI manager classes. Both of these variations keeps track of the objects that have been added to them, themselves rather than returning the objects they create. This was done to keep all the objects of the same type to their own manager class.

### Player Object

### Artificial Intelligence

The artificial intelligence was initially going to be based on a finite state machine, each AI would have an instance of a finite state machine that would cycle through a series of behaviours relevant to that AI. However due to time constraints and functionality issue each AI uses only one behaviour each, with the seek behaviour being shared between two AI. This effectively defeats the purpose of the finite state machine as there are no state changes, however it does leave room for expansion through the addition of new states for new behaviours. The states use a singleton design pattern so there is only ever one instance of each and all variables that the behaviours rely on are stored within the AI objects themselves.

The AI objects are stored in a series of linked lists in an AIManager class, this class constantly iterates through each of these lists updating the AI’s with the player position, displaying the AI and deleting any AI that have dropped to zero health.

### Menu World

### Model Importer

### Collision Detection

### Game Camera

# What issues were encountered and how were they resolved?

* Model importing
  + One of our biggest issues was to do with importing models, everytime a model was required it would be read into code at the time it was needed, in the area of AI this caused a slight hang each time an AI was spawned, this issue was not resolved, if we had time we would have liked to have set up a model manager class that read in all required models at compile.
* GitHub
  + We had some major issues with our version control program git hub some changes took huge amounts of time to upload, updates were lost or only half uploaded and errors would appear between uploading and downloading a program version. This issue was never really fixed and had to be simply lived with
* Error with ATEXIT\_HACK
  + 3 of the group members encountered problems with freeglut with regards to ATEXIT\_HACK. This was found to be because of older versions of freeglut and was resolved by adding the line ‘#define GLUT\_DISABLE\_ATEXIT\_HACK’ at the top of every file that uses freeglut.

# What special features are in your program?

* Artificial Intelligence
  + The artificial intelligence is something we are really proud of. Building on our knowledge from ICT219 we were able to use finite state machines to good effect to get some interesting behaviours from the different AI types.
* GUI
  + The GUI was quite a struggle to build through the use of subwindows however we feel as though it was a valuable learning experience and is implemented relatively well.
* Use of self-contained factories
  + The use of the factory design pattern more or less came about by accident. We figured we needed a class to manage the game objects and that it would be useful if we had one that had a list of all the game objects and allowed other classes to add to this list and then they would be updated. While we didn’t implement it for all game objects, it did work for projectile objects and contains functions for adding projectiles, updating them and deleting them. We are aware that it isn’t a factory in the strictest sense of how it is defined however it’s something we are proud of.

# What issues, if any, arose when you were doing the maintenance part of the assignment?

The game world from the first assignment was largely scrapped since it had no useful purpose and was not implemented in a way that we could build a game around it. The way it currently is, is still to sufficient to build a game around however it is better than what it was.

The biggest change from assignment 1 was transforming Shays code into its own class. This was somewhat of a hassle since each of the #defines had to be transformed into enumerated types. One problem we had with changing Shays code into its own class was that the planes in Shays world stopped working at some point, possibly through fault of our own. It was identified that the program was setting the number of plains used to a negative value so this value had to be hard coded.

# Testing Details

Testing was almost non-existent throughout the writing of this program. Since this was our first game we didn’t have much idea how to test something which is mostly a visual experience however some obvious candidates for testing are Vector3 and the Score classes. We should have done a better job creating components of the overall system outside of the main program and then added them once we were sure they worked. However the times we did try this, things that worked outside of the main program did not necessarily work inside the game program. We feel like this may be in part due to the GLM model importer.

# Suggestions for Improvement

The biggest improvement that could be made is not related to the actual construction of the program, but is instead to do with time management. The major working periods on this program were upon initially receiving the assignment and during the few weeks leading up to due date, this was due to underestimating the time required to do tasks and the number of tasks involved. Doing it differently we would ensure progress is made each week thus enabling us to better judge what exactly is required to be added and the sort of time required to complete those tasks.

One thing that could have helped us with our time management was the use of an Agile development methodology. We feel like SCRUM is a bit difficult in an environment where people are also working on alternate projects (i.e. other units) at the same time. Perhaps a system such as Kanban would have helped more since having a common list of tasks could have helped more than daily meetings that failed to happen throughout the development.

There were of course many program related areas in which we could greatly improve also, the largest of these would be memory management, as we were low on time much our code is functional rather than well written and set out, therefore many objects are not deleted after their use is fulfilled so more memory is continually taken up as the program runs. The biggest memory sink was related to reading in models using GLM, every time a model was required it would be read in by GLM then displayed, even if the same model had already been used before. This was a huge waste of resources as nearly every in game model is used very often. What we would have liked to have had was a class that managed the models so that one model was only read in once and then could be used anywhere.

Multiple classes were also poorly written, with some instances of public variables being used simply to get things working. Had we the time we would very much like to overhaul most of the classes to get them running in compliance with low coupling and high cohesion.

The Finite State Machine AI also needs improvement since the way it is currently is that each AI only uses one behaviour. This behaviour handles everything that AI does instead of transitioning between multiple behaviours. Splitting these single behaviours would have made the code much more readable and understandable. This also currently defeats the purpose of having the finite state machine class.

Given more time we would also increase the current scope of our game, by adding more AI and more abilities so that the game play could be mixed up more to keep it fresh, currently the only change is difficulty as the AI come out at a faster and faster rate, this means the game gets stale very quickly.

If we had our time again, we would try to manage our time better throughout the entire development process so we don’t have to crunch so much at the end. We would also spend the time on trying to create our own model importer since the current one doesn’t provide us with the functionality for animations and uses more of a functional C style rather than the object oriented C++. We feel as though the entire program needs to be redesigned and rewritten since in its current state, it feels like too much spaghetti code.

# How is code reuse taken care of?

Code Reuse was something done reasonably well in this program, the major way was through inheritance, the majority of the class in the game inherit from the game object class, that contains variables and functions that get used in all of the children, the same also occurs with the projectile class, where projectiles of different types inherit from a parent projectile. It can also been seen in the AI where multiple AI can use the same state, in the case of our game the seek behaviour is used by two different AI’s who simply set their own speed to differ from each other.

Despite code reuse playing an important role, inheritance is not used in the strictest sense of how it should be since in all cases not all of the functions/variables in the parent are applicable in the child class.

# Can the assets/objects/items be reused easily in other programs?

The classes in this program were designed with use in other programs in mind however late in development; it came down to changing things within those classes just so that our program would work. The only classes that can easily be used in other programs are quite possibly the Vector3 class and the AI state machine classes. Most of the functions/variables in the GameObject and World classes could be used in other programs but may require some minor changes to suit those programs. The sound class could also be reused with some minor tweaks.