Architecture Report

Overview

As can be seen from the diagrams in the report each class is linked to one another in a certain way. The architecture is divided into

- Main Class, within the BTC package
- Scene classes, within the SCN package:
    - Each different scene represent a game state/scene such as the game over scene
- Entity classes, within the CLS package:
    - These classes hold the information on the various objects in the game such as aircrafts and waypoints.
- Library classes, within the LIB and LIB.JOG packages:
    - These classes provide functionality through use of established libraries.

This report will look at the divisions and the architecture of each class, examining in detail and justifying the links between classes. The overall architecture is illustrated in appendix A.

Main:

The main class runs the game loop. It also handles initialisation of the library classes, e.g. it is responsible for initialising the window to provide an on screen GUI, and quitting and cleanly exiting the game loop when the program is ended. Scenes exist within Main, and are handled by Main's update and draw functions. Main can also close a scene and start another upon request from the scene, e.g. when a collision has occurred and the game over scene needs to be presented. Scenes are stored in a stack - as such, strictly only one scene is active at a time. Hence, communication between scenes should be carried out by passing parameters when creating a new scene. For example, a game scene can pass two crashed aircraft as a parameter to the game over scene, which can use them to draw an explosion over the aircraft's prior location.

The game loop method is shown below. It calls methods to update the scenes and their objects, passing the delta time since the previous loop, ensuring time-dependent updates function correctly. Once the updates are complete, the updated state is drawn and the loop begins again.

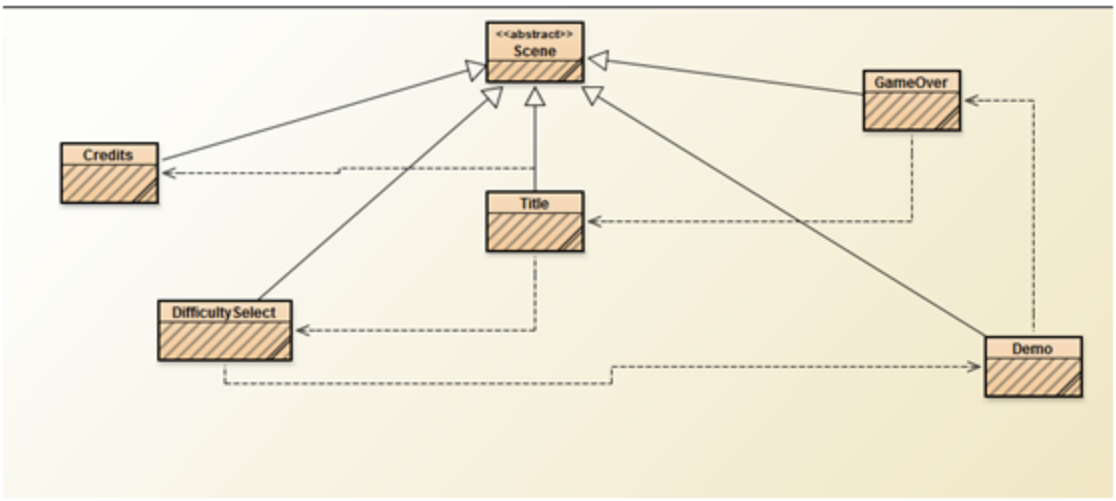
```
public Main() {
    start();
    while(!window.isClosed()) {
        dt = getDeltaTime();
        update(dt);
        draw();
    }
    quit();
}
```

The start function involves initialising the game window and graphics. It creates a scene stack, which holds the scenes in the game. The choice to use a scene stack allows us to modularise all our scene classes and allows us to use abstraction in the classes allowing us to constantly update the game scene by calling the update and draw functions. As can be seen in the class diagrams, each scene can call on another scene, when this call is made the scene is put on the scene stack. The current scene is terminated and resources released before it is removed from the stack and the next scene is drawn. This provides a clean and efficient way to track which scene is currently being used and allows new scenes to be added independently of this class. Use of a stack also allows the user to seamlessly back out of the game, e.g. to the main menu, rather than having to restart the game.

The getDeltaTime function measures the time between each frame so that we can keep track of how long it has been before each scene update. This ensures updates to time-dependent objects occur correctly. For example an aircraft with a certain speed will appear to move at the same rate regardless of how quickly updates are being performed.

The update function handles audio, input, the current scene and the window. The audio, input and window update events, simply update any audio which has been passed from the game to be played, whilst the input passes mouse and keyboard input from the user to any listening handlers. All of the objects in the scene are updated, e.g. writing text in the difficulty selection scene and updating aircraft position and collisions in the game scenes.

The draw function works in a similar fashion. When it is called in the main method, the current scene (residing on the top of the scene stack) has it’s draw function called. Thus the draw function for that scene is run and can be used to draw aircraft or end game scenarios depending on the scene. As mentioned before, this has been done in an effort to make it easier for future coding to be as easily understood and implemented - scenes handle themselves, and the main class modularly handle the current scene. The hope is that due to the abstraction of our update and draw functions, other classes can be added or edited without major change required elsewhere in the program.



Scenes

The scenes provide the ‘game’ - entities such as aircraft exist within them, which the scene is responsible for updating and drawing when requested to by the Main class’ update and draw. Furthermore, input events are passed to scenes for action by the Main class. Scenes can call up to the main class to close themselves and create other scenes with parameters if needed, but otherwise may not interact. Scenes exist within a stack in the main class where only the top scene is active at any time. A chart describing how the player may move between scenes is presented in Appendix B.

Scene:
This class holds abstract methods such as update() and draw(). Each different scene extends this class and implements the abstract methods so that the main method can call upon the functions when updating the current scene. The abstract methods are start, update, draw, close and playSound. Generalising across all the individual scenes, start initialises the scene, including data structures and any objects the scene needs to begin with. Update, as described above, updates the scene and it’s objects and data. Draw, calls methods through the graphics library to draw the state of the scene to the game window, for example text and graphically represented objects such as waypoints. The playSound function allows the scene to play a requested sound, such as collision warning beeps. It is important to note that this function is not intended for music, which is instead handled by the main scene to prevent a scene’s music continuing beyond the lifetime of the scene itself. Finally, close cleanly exits the scene - for example, stopping any music that the scene was playing. The start, update and draw functions can be greatly different between scenes, and will therefore be discussed. Additionally, scenes may implement an input handler to receive mouse and keyboard events from the user (which are passed by the main class to the listening scene.) Not all scenes implement this - for example, the title and difficulty scene instead use buttons which do not require the implementation of an input handler. There are currently five interactive scenes implemented that the user will meet

while playing the game.

1. Title

The title scene is the opening scene of the game. From this screen the user can view credits, help on how to play, and start the game. Starting the game leads to the difficulty selection scene. Thus the title screen has four buttons, which are initialised upon the scene start.

1. The credits button leads the scene to be updated to the Credits scene.
2. The demo button leads the scene to be updated to the Difficulty scene.
3. The Full game button does not cause any update as we have decided to leave it in case of any expansion in the future
4. The exit button quits the game.

The update function handles the regular ‘beep’ of the radar and the angle of it’s sweep, which is drawn by the draw function.

2. Difficulty

This scene is where the user chooses the difficulty. We have chosen to change velocity, spawn rates and separation rules based on which difficulty has been selected, increasing the speed of aircraft in the airspace, increasing the strictness regarding the separation of aircraft, and decreasing the time between new aircraft entering the game’s airspace. Thus when the respective difficulty button is pressed, the DifficultySelect class creates a new Demo object passing the chosen difficulty as a parameter. The difficulty is stored by the Demo scene to be used when generating new flights.

Similarly to the title scene, the start function initialises the buttons to continue into the game, and the text to be written to the text box (which is implemented as an extended ordersBox). The update function handles the updates of the textBox used to create the faux-teletype text, which is printed to the screen by the draw function.

3. Demo

The demo scene is a main game scene and is the scene which runs when the game is being played. This class contains all the interactions the user has when playing the game and interacts with all the object classes to create aircrafts, waypoints and check collisions.

Waypoints are stored as two arrays - a set which may be used by an aircraft to enter and exit the airspace (a group of 4 corner waypoints) and the set of all the waypoints in the airspace, inclusive of the first set (10 further waypoints, for a total of 14 including the first set) . Waypoints which can be used to enter and exit the airspace are ‘location’ waypoints and have a labeled city name. Other waypoints are considered ‘airspace’ waypoints. Thus when aircraft objects interact with these waypoints, we can determine if an aircraft is arriving, is in the airspace or is leaving.

The demo start function initialises several important features. Music to be played during the scene is initialised and started, and buttons used to control aircraft on screen are initialised and prepared to receive input. Several data structures are initialised and prepared; for example, the flight generation timer is started.

The update function in this scene handles many objects, such as the aircraft in the airspace and the altimeter. During the update function, the scene calls out to other functions such as

checkCollision and generateFlight to perform other functions. Crucially, these functions are only called by the update function or as a result of a function called during the update (for example, checkCollision may call gameOver to end the game if two planes have collided.) This makes increasing the complexity of the scene easy - new functions need only be called through the update function. An example of this is the generateFlight function, which, after a certain period (dependent on the selected difficulty, tracked by the scene and timed by the update function) has elapsed, constructs a new aircraft to enter the airspace.
The draw function handles a great deal in this scene - for example, the drawing of the game GUI, the printing of text in the orders box, and the drawing of waypoints and aircraft. In order to improve readability, these functions are split into several functions which are called by the draw function - for example, the drawPlaneInfo function handles the printing of details such as speed, destination and altitude when a plane is selected by the player.
This scene implements an input handler to recieve mouse and keyboard input. This allows user input to carry out tasks such as ordering a plane to change altitude or altering the course of the selected plane. As these functions occur upon input being recieved, they represent the only functions which can be accessed externally to the start, update, draw, close, and playsound functions.

As part of the requirements, the game must check on a regular basis if there are any imminent collisions. This is done by the demo class regularly running the update function which involves checking if there are any collisions and re-drawing events dependant on actions performed by the user.

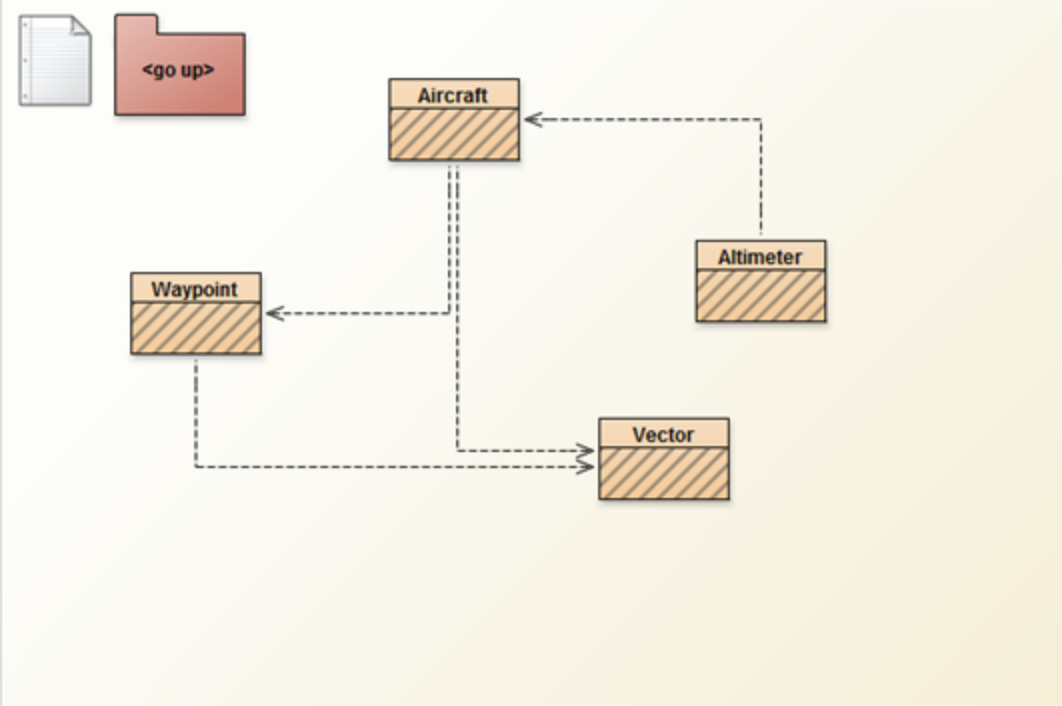
4. Game over

The game over scene is the result of two planes crashing. This scene is created as a result of a previous game scene reaching a game over state - for example, due to aircraft collision - which calls a gameOver function. When this occurs, the prior scene is popped from the scene stack and the game over scene takes it’s place. The game over scene includes information of which aircrafts crashed, drawing the crashed planes with an explosion animation. This scene also implements an event handler, allowing any input event to exit the scene and return to the main menu.
The start function of the game over scene initialises two new aircraft based on the details passed of the old ones, preventing any reliance on the previous scene. A sprite animation is also initialised to handle animation of an explosion. A timer is initialised at 0, to be used to draw the aircraft for a certain period and then draw the text box. Finally, the text box (implemented as a large ordersBox) is prepared with text to be written regarding the player’s failure to avoid the collision.
The update function tracks a timer since the start of the scene. Between time 0 and 3 seconds, the explosion sprite is updated over the crash location. The aircraft are not updated as they remain stationary so the player has an understanding of where they failed. After 3 seconds, the sprite is no longer updated and the text box is instead updated.
The draw function handles the usual GUI elements. Similarly to the update function, the draw function draws the aircraft and explosion sprite for 3 seconds after the start of the scene. After 3

seconds, the text of the text box is instead printed to the screen.
A very simple input handler is implemented, allowing the player to immediately skip back to the main menu by pressing any key.

5. Credits

The credits scene can be accessed from the title scene and holds names of the creators of the game and the teams. The start function of this scene simply initialises some data and music, whilst the update function controls the speed at which to scroll the credits. The credits themselves are drawn through the draw function as a series of text prints. A very simple input handler allows the player to increase the scroll speed of the credits by holding the left mouse button. This scene was implemented to credit the team and our use of external libraries and audio resources.



Entities:

These classes represent entities within the game - strictly, they are declared or created within a Scene, which is responsible for managing them in it's update and draw functions.

1. Aircraft

The aircrafts that are created and then controlled by the user are the integral part of the game. An aircraft object holds important information such as it's position and speed. A single class

represents all aircraft - to represent e.g. a biplane the speed attribute could be lowered and the image attribute changed to a biplane icon when the aircraft is created.

This class holds all the information on an aircraft such as velocity, radius, destination, current target etc. All these variables are stored as private, as they should not be directly changed by the scene in which they reside. Instead, an update function is provided which handles this. This is called during the scene's update for all aircraft in the scene. The separation rules for the aircraft are also initialised in this class and the values are gained from the difficulty selection scene which stores the value chosen by the user. When a user clicks on an aircraft the GUI shows a lot of information about that aircraft, which is easily, accessed using the accessor methods in the class.

When an aircraft is initialised, it obtains a random name, origin and destination. Using a greedy weighted path finding algorithm (mentioned later), a route of waypoints are assigned to the aircraft. This is the default route a plane will follow when entering the scene. Offsets have been added to the aircraft class to ensure that planes do not crash into each other on spawn points.

There is also a draw function in the class. This is provided to allow all aircraft to be easily drawn to the screen when the scene is carrying out it's own draw function.

The update and draw functions together update and graphically represent the aircraft as it travels throughout the airspace - for example, positioning is updated according to direction and speed, and the aircraft drawn at it's new location. The route is followed and updated as waypoints are reached, keeping the aircraft on track. Furthermore, the aircraft will ascend and descend over time as ordered by the player. Since there are two discrete 'levels' of altitude in our game, the plane images are shaded differently and show different labels depending on their height, dependent on the altitude the player has the game at in a game class.

The aircraft class uses a greedy route finding algorithm to initialise a default route of waypoints from origin to destination, via the waypoints in the scene. This is carried out by use of the Waypoint and Vector classes static functions, e.g. getCostBetween, to calculate a route which favours the waypoints closest to the aircraft weighted by the distance from the waypoint to the destination. This creates a realistic routing behaviour - an aircraft will never initialise with a route which heads away from it's destination. This cheaply approximates an optimal route finding algorithm for the small number of possible waypoints in the airspace. As waypoints are unchangeable by the player, as the game progresses and many aircraft are in the airspace, they player is challenged to alter this route and prevent collisions.

Other functions are provided for convenience, e.g. to allow the scene to check if an aircraft is out of bounds and to allow the scene to handle mouse input which is altering an aircraft's route.

Aircraft are created and held in ArrayLists by their parent scene, which may act upon them - e.g. during a scene's update function, all the aircraft in the list will be called to update.

1. Altimeter
   This class holds information about the plane's height in feet and the current banking of the plane, for a selected plane. An example is the Altimeter object in the Demo scene, which graphically represents the selected aircraft's behaviour as it turns towards waypoints on it's route. The altimeter can be used to request a selected aircraft to ascend and descend, and therefore implements an event handler. The update function alters the altimeter chevron as the plane banks and moves, whilst the draw function draws it graphically to the screen. Again, these functions should be called through the scene's update and draw functions.

2. OrdersBox
   When an instruction is given by the user to a certain plane, the instruction is shown as a order from the ATC to the pilot via this orders box. It includes a function to have the text print character by character in retro style. It includes an addOrder function which is called upon when a new command is issued to an aircraft. This has been implemented in an attempt to make the game feel more realistic, as the ATC gives commands to the pilot. It is used in the demo scene to provide the order's box which reads out the player's commands.

3. Vector
   The vector class is a basic class with vector operations. This class is used as a data structure, e.g. to hold the heading of an aircraft or the position of a waypoint, and also as a convenience, for example to calculate the distance between two waypoints during route finding operations. It is connected to several other classes - e.g. Aircraft have a position and velocity vector, whilst Waypoints have a position vector. Furthermore, static methods are used by other classes (such as Aircraft during route finding) to get distances between waypoints, etc.

4. Waypoint
   This class holds information about each waypoint on screen, including their position as a vector and whether or not the waypoint should be considered a point where the aircraft can enter and exit the airspace. Since waypoints do not move or change after initialisation, this class does not have an update function. A draw function is provided to allow the scene to draw it's waypoints. Since any waypoint can be an entry or exit point, functionality is available for aircraft to take off or land at runways - a waypoint on the runway represents the take off / landing point, and aircraft can be randomly initialised (as with other waypoints) to spawn there and enter the airspace, or indeed with this point as a destination to reach. Waypoints are declared in the code and stored in Arrays within the parent scene, which may be passed as needed to other classes - for example, for route finding by aircraft.

Library classes:

These classes below have been referred to in the above descriptions however not been described. The classes are mostly used for GUI purposes and used by different scenes for the purpose of writing text such as the end-game scenario. They are used following object oriented principles to provide functionality for multiple scenes with an implementation independent of the

scene; for example, ButtonText provides button functionality across multiple scenes. These classes differ from the prebuilt library classes are they were made by the team to provide functionality specific to the Bear Traffic Controller game, and are not available as imports externally.

1. ButtonText - This class is for text in the buttons on our main screen or other screens in the game. It holds the text for that button and the action that the button does when clicked. It also holds the coordinates and size of the button. It is used to provide button functionality throughout the program.

3. RandomNumber - A class to generate a random number between a predefined minimum and maximum, built upon Java.util.Random. This class is called when a random number is required in the scene, for example for a flight number or for the number of deaths in the end game scene. This was defined as a separate class as many other classes (e.g. the demo scene, for positioning a newly spawned flight) use random number generation for various purposes. Combining these functions into a single library function improved maintainability and a place for any additions to random number generation.

4. SpriteAnimation - This class is concerned with the animation for the sprites seen on screen, eg: the explosion on the game over screen. The object is initialised with an image, coordinates, fps and number of frames in the animation. Though only used for the explosion, this class provides functionality for sprite animation of any other image; for example, aircraft with an animated contrail following them.

The update function checks if all the frames have finished drawing if not it resets timer and increments and retrieves each frame till all frames have finished drawing. The draw function draws the animation frame by frame.

5. TextBox - This class is similar to the orders box, it also includes delays and a retro style printing function. However this class implements a more important update and draw function. The update function updates the timer of the textBox and adds an delays if the buffer is full. The draw function, prints the available characters to the text box. The class can be used to provide the retro teletype style printing throughout the game, e.g. in the demo scene and the difficulty scene.

Prebuilt library classes (Lib.Jog):
These classes provide functionality needed for the game through established libraries, e.g. providing audio and graphical functionality. As the libraries are established, we can be confident they are well tested and fully featured. Use of these libraries significantly reduced development time as it was not necessary to rework the wheel in order to provide vital functionality. As a result, we were able to place our effort into other areas of development. The libraries used combine functionality from both LWJGL (*Light Weight Java Gaming Library*) and Slick to achieve their purpose.

1. Audio - The class contains the functions for any audio in the scene such as volume control, pausing and looping. It has been used so that we can add background music to any of our scenes and adjust position or volume dependant on what the action in the scene is. It also allows for playing sounds, e.g. the radar beep on the title scene or the explosion caused by a plane collision. This library encompasses LWJGL's OpenAL support, and Slick's Audio, Audio Loader, Sound Store and Resource Loader support to provide the audio functionality. Any class which has functionality to play sounds uses this library - this includes the scenes, and the aircraft class which can call the PlaySound of their parent scene to play the collision warning beep.

2. Graphics - This allows basic drawing of shapes to the screen as well as images, fonts and printed text. It includes methods for Quad setup which is used in the spriteAnimation class mentioned above. This library handles all drawing to the screen, and encompasses LWJGL's OpenGL support and Slick's font, colour, texture and resource loading support. Unlike OpenGL, this library considers the origin of the screen to be the top left of the window. This library is used by any class which has functions which draw to the screen - this includes all the scenes and the main class, the aircraft, altimeter and so on.

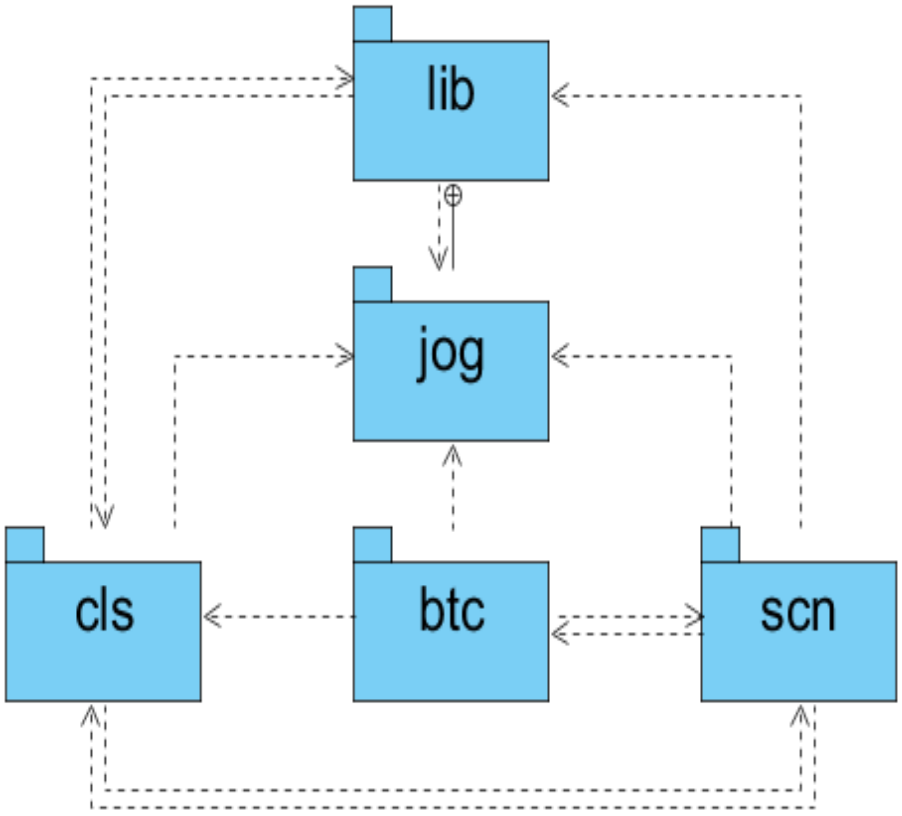
3. Input - This class is encompasses LWJGL's Keyboard and Mouse handling. This provides us with complete support for input events for any key and mouse action, and the handlers to act upon these events. This library is used by any class which must handle user events including the main class and several scenes, e.g. the demo and game over scene. Not all scenes use this directly, however - for example, where ButtonText is instead used to provide button functionality in the difficulty selection scene.

4. Window - This library provides a layer above Slick and LWJGL, using methods from both libraries to create and manage a window inside which the game is presented. Slick is used to load the icon image which is presented on the window's border. This library is used by the main class to create and manage the game window.

Missing Functionality

Though our requirements state that the score of the player should be calculated and tracked, we were asked by the client to remove this feature until a later date. Hence, this requirement is not satisfied in this version of the game.

Appendix A - Overall Architecture Diagram



Appendix B - Scene Interaction Chart