System Design Document

Stargate: Galaxy

Peter Mitchell
Andrew Krix
Karlos White
Phil Lavender
Kane Stone

8/31/2011

Table of Contents

1 Figures

2 Revision List

Name	**Date**	**Reason For Changes**	**Version**

3 Introduction

This document discusses the implementation strategy being employed for the development of the system for Stargate: Galaxy. This document takes the information from the preceding documents and applies the requirements to the implementation strategy for within the Unity development environment. For those who have not read the earlier documentation, the game Stargate: Galaxy is a game based in the Stargate TV show universe. It involves the player by their advancing through the game till they are able to free the galaxy from the oppression of the many opposing races as seen in the show. The player will accomplish this by controlling a ship and engaging the enemy in battles. From these battles they will gain experience and resources to become stronger.

4 Document Outline

The document will cover a range of the different details that are required and be defined as part of the implementation of the deliverables. It begins with an overview of the system that is being designed. Considerations for the design including: assumptions, constraints, goals and guidelines, and the methods that will be used to develop, the architectural strategies that apply to the particular system that is being developed on, the architecture of the system and any relevant sub systems, the policies and tactics that will be used for the design, and a detailed system design with details on the design of the sub systems. This document has been designed based on the outline proposed at:

http://www.cmcrossroads.com/bradapp/docs/sdd.html.

5 System Overview

The system to be employed in this game is based off of the way in which the Unity game engine that is being used allows for development structuring. Elements are divided into various scenes that together form the structure of the gameplay. The identified scenes include: a main menu, a loading screen, a system scene that contains planets that may be travelled between within the confines of the current system. Each time the player travels in hyperspace or by Supergate to another system of planets they will be supplied with a different view of what that region of space in the galaxy is populated with. It will be appropriately populated with ships of varying numbers from different races, planets that may be controlled, and other more specific elements like a Supergate. The details of and requirements needed for each of the relevant scenes is discussed in more detail in the System Architecture and Detailed System Design sections.

6 Design Considerations

6.1 Assumptions and Dependencies

Assumptions and dependencies related to software, hardware, operating systems, end users, and changes to functionality include:

1. The end systems that will be developed for assumes a Windows environment.
2. The Unity definition for system requirements for Unity-authored content that may be found at http://unity3d.com/unity/system-requirements.html, states that Windows 2000 or later with nearly any 3D graphics card should be sufficient. For the purposes of this project testing will be performed on at least Windows 7 and perhaps Windows XP or Vista, but the primary client is intended to be Windows 7 systems. The Unity system does allow for deployment to other systems; however, only Windows will be considered for the purposes of this project.
3. There are not currently any known software requirements. There may be a minimum version of DirectX that needs to be installed to play games developed in Unity or some other requirements. These assumptions and dependencies will be further investigated.
4. The assumed end users are those of a Stargate background who would enjoy becoming involved in the themes and gameplay in the image of the show. The expected audience is not restricted just to Stargate fans. The gameplay is designed to be only roughly aligned along the characters, races, and ships that are found in the series. Anyone who is interested in futuristic space shooters involving large ships will be included in the assumed collection of interested end users. The end users are expected to be existing fans of video games systems and have a reasonable familiarity with typical control systems that this game will try to mimic. The user guide that will be provided will allow users to understand enough about what it is that the game is and how to play it.
5. As this project is for a university assessment and needs to be completed on a deadline it has been assumed that there are lots of features of the game that had to be removed. These features could be later implemented as features potentially if the game was continued post production. Although this would require further assessment of the legal rights to use of material as the backstory in particular is in use as a means of being an educational tool within a university.

6.2 Goals and Guidelines

The primary goals and guidelines of the design and project are:

- To create a product under a set of deadlines as set by the group and by formal assessment criteria.
- To perform necessary research during development and design that allows for learning of new material and the application of that material.
- The development of a game that can be played by a variety of audiences.
- To gain experience developing formal documentation.
- To make a working product.
- To keep it interesting without going overboard. (KISS principle)

6.3 Development Methods

The development method to be used is based on a general agile approach similar to the spiral model. The code and assets are to be developed in sets of content and then merged together from the various sets of code being developed by the different coders and collects/creators of assets. Initially smaller amounts of code will created to allow developers to familiarise themselves with the Unity system, and then increasing amounts as deadlines approach. The code and assets have many divisible sections allowing for parts to be worked on. The nature of Unity allows particularly for importing of scripts and assets without too much time loss during the merging. This may be done with the package method for each exporting. Although the development of the individual systems will be done separately the use of Dropbox as a coordinated file management scheme allows for rapid sharing of new content.

7 Architectural Strategies

This section covers the strategies and information associated with decisions that have been made about the general implementation approach.

7.1 Use of Products

- For this project the most important design consideration is the integration into the development platform of Unity. The reason Unity is being used is that it has been discovered to be a seemingly robust and easy to learn system. It is also recognised as a leading game engine as cited by Game Developer Magazine readers.[1]
- As part of the Unity engine there is a choice between three different languages. The functionality that can be used between the three is similar and near the same. Those languages include: C#, Javascript, and Boo. C# has been chosen due to familiarity with C# and its similar functionality with Java that the majority of the development team has worked mostly with.
- XML data storage using object serialization as a means of storing the games data is a simplified approach to data storage and retrieval. It does not secure the data in such a way that individuals could not go and just edit the files, but for the purposes of this game it allows a structured approach that allows for an engaging game without too much overhead.

7.2 Reuse of Existing Components

- Unity provides a robust framework that does include a variety of scripts; some of these may be used as part of the product. Those that are used may require modification to suit the purposes of the game, but there are likely many similar actions that may be used. During work on individual components investigation of whether there are already useful frameworks or code to use available will be investigated.
- In the case of Audio existing audio will be used from a variety of sources. The development will not have time and do not have the skills needed to create all the audio required for the project. All of this shall have been outsourced and any relevant references will be appropriately included with the game.
- The models developed for the application will be most if not all developed by the group's digital media student. If he is unable to produce enough of the models within time there have been backup options located. These options would need to be investigated for being allowed to use the models.

[1] http://www.marketwire.com/press-release/unity-named-the-1-game-engine-by-game-developer-magazine-readers-1518427.htm

7.3 Error Detection/Recovery, Memory Management, and other general System Functionality

- Errors to do with gameplay features should be detected and modified so that they shouldn’t ever need recovering.
- General error detection and recovery, along with memory management, and other core features will be expected and assumed to be handled by the game engine. Testing will be done to check that it reacts appropriately and as expected. Any features that are found to behave incorrectly will be investigated further and a work around or fix should be implemented to correct the issue/s.

8 System Architecture

As has been previously discussed the product is divided primarily into a series of “Scenes” as Unity uses to provide its development environment. A scene may be thought of as a level and can be inclusive of menus and loading screens as scenes. Each of these scenes can have a hierarchy of game objects. Game objects by themselves just provide a point in 3D space, but by adding components, scripts, meshes, and other elements to them they are given functionality to interact or be interacted with under the game play conditions. Some scenes reuse objects and particularly scripts from other scenes that involve managing the game state. These controller objects will be indicated and a definition shown of how and when they persist between scenes. Objects that don’t persist between scenes but are reused multiple times like ships for example will be pre-defined externally as what are called Prefabs that may be duplicated multiple times without having to have the item as a unique object in the scene hierarchy prior to scene initiation.

The scenes that will exist include:

1. Main Menu Scene
2. Loading Screen Scene
3. System Scene

Each of these is outlined as to what they are and the elements that would be found in the scenes over the following pages. The more detailed information on individual objects is specified in the section on detailed system design.

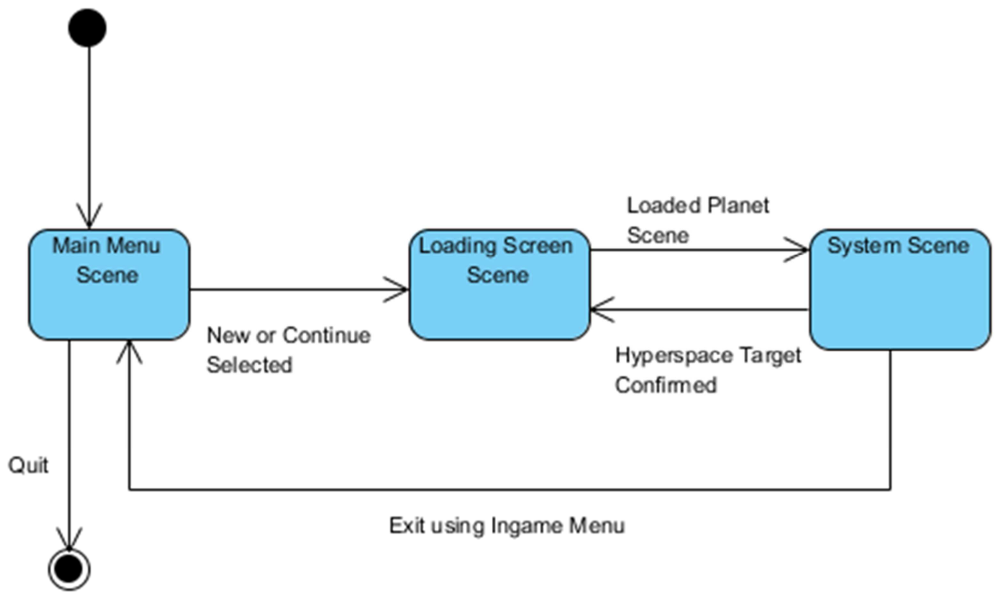

Figure 1 : Scene State Model

8.1 Main Menu Scene

The Main Menu scene is what will greet players the first time they enter the game. It will provide the means of how they will begin playing. The presentation will be simple and nicely presented to encourage players to continue on into the game.

Elements of the Main Menu will include:

1. A background image that is static depicting a Stargate themed scene.
2. Buttons for “New Game”, “Continue Game”, “Credits”, and “Quit”.
3. The modified view to show the credits on screen.
4. Background audio music.

All of this will be covered under a single Game Object:

1. Main Menu Controller

8.2 Loading Screen Scene

The Loading Screen is seen when the player has initiated one of the following conditions:

- They have started a new game.
- They are loading a game.
- They have entered Hyperspace to transition to another scene.
- They have entered a Supergate to transition to another scene.

The primary purpose is to provide a simple screen that shows something is happening. This is done by presenting a ship object representing the player’s ship and having an effect that creates the illusion of travel through Hyperspace (that may be considered as the action for Supergate travel too, just that the “entry point” is different in style. The player will not be able to interact during this transition time, but the scene to follow will be loaded and configured while the player waits.

Unique Game Objects required for this scene are:

2. Hyperspace Travel Effect

Common Game Objects required for this scene are:

3. Game Controller

8.3 Planet Scene

The Planet Scene is the scene that will be where the game play is almost entirely spent. The scene can be highly variable in contents, but the general mix contains a collection of planets and a collection of enemies and/or allies in addition to the player's ship. Some special variations include a Supergate that the Supergates allow travel between the two zones that they connect.

Unique Game Objects that are required for this scene are:

4. Ship
5. AI Controller
6. Planet Object
7. HUD Controller
8. Supergate Object
9. Projectile
10. Explosion Effect
11. Wrecked Ship
12. Camera
13. Skybox

Common Game Objects that are required for this scene are:

3. Game Controller

9 Policies and Tactics

In developing the product the individual components should be developed separately during the earlier stages of development. Communication should be maintained to ensure that when linking occurs that it will all join together correctly. The reason for developing separately is that when working with many large assets it is difficult to be continually from separate locations continually updating a single copy for Unity.

In the mid to late stages the development team should work together one large copies that contain as much as possible and export packages of the code that they have been working on. The copies of assets should be separate enough that merging code will not affect the product in a negative way.

Regular backups should be kept by all parties while working on the development so that there is no room for loss of data due to a failed merge.

10 Detailed System Design

10.1 Main Menu Controller

10.1.1 Classification

Invisible GUI controller

10.1.2 Definition

The Main Menu Controller is the object that controls the rendering and of the GUI seen. It renders the GUI elements of the buttons and background including handling of the interaction with the interface components. It will also display the credits and provide means of exiting the game. The other functions that it is capable of providing are the means for entering a new or existing game.

10.1.3 Responsibilities

This game object is responsible for:

- Providing the first visual entry point in the application.
- Provide a visually appealing interactive Interface to invite the user to play the game
- Providing an easy to use menu that allows quick starting of a new game, continue an existing game, display the credits, or quit the game.

10.1.4 Constraints

This main menu will be the first and last object to appear, it must give the user the ability to access the rest of the game from a simple interface. The object will load automatically upon start up clear itself when the game world and loaded and then reload when the user quits the main game environment.

10.1.5 Composition

The object should contain a single script file that has a the definition of rendering the GUI and a second script to control the movement of the camera.

10.1.6 Uses/Interactions

Upon initial game load this object will automatically be run and controlled by Unity using the relevant methods for handling GUI creation.

10.1.7 Resources

- GUI Menu Skin
- Skybox
- Planet earth Object
- GUI buttons
- Stargate Galaxy logo
- Game camera

10.1.8 Processing

This object main function is to provide links to all other game sections. The user selects a menu option and the object invokes the main loading of the game.

Creation

- Determine the dimensions of the screen
- Initialise all variables in relation to screen dimensions.
- Load Skybox
- Load earth

Display

- Draw main GUI menu texture
- Display all buttons and logo
- Display Credits when called

Cleanup

- Remove all elements within the scene

10.1.9 Interface/Exports

This object does not provide any interfaces for external components except for the methods required by Unity to automatically call to create the GUI for the application.

10.2 Hyperspace Travel Effect

10.2.1 Classification

Visible Effect

10.2.2 Definition

This object handles the displaying of the effect seen during the loading scene.

10.2.3 Responsibilities

Displaying an animated effect that appears like the player is travelling through hyperspace or supergate.

10.2.4 Constraints

This object should not need to communicate with any of the other scene objects. It should be destroyed at the end of the loading screen to make it simply disappear. The animation must be done with individual images. Unity Pro would be required to play movies.

10.2.5 Composition

The object should be comprised of:

- The collection of images used to animate the motion of travelling
- A script that controls the displaying and updating of images

10.2.6 Uses/Interactions

This object should not interact with any other objects.

10.2.7 Resources

The only resource required for this object is the collection of images. Importantly they must be in order.

10.2.8 Processing

When the object starts it should configure the first image to be displayed and identify the current time.

When the object updates the current image should be cycled with even time spacing.

When the OnGUI call occurs the current image should be rendered to the screen.

10.2.9 Interface/Exports

The three methods that allow the processing are all directly called automatically by the Unity engine via the MonoBehaviour system.

10.3 Game Controller

10.3.1 Classification

Invisible Controller

10.3.2 Definition

The game controller is an object that is persistent between all scenes except the main menu. It maintains the game state and automatically makes overall updates to simulate the different groups automatically attacking. As part of this object it maintains a collection of the systems that exist with information on their current state and that of what the player would know. The object maintains a database of the ship data and weapon data that all other objects are allowed to access. Importantly also it maintains the prefab links for all planets, the supergate, every ship, every projectile, wrecked ship and explosion. These can then be referenced from other objects.

10.3.3 Responsibilities

The Game Controller is responsible for:

- Maintaining a list of all different ship prefabs
- Maintaining a list of all different planet/supergate prefabs
- Maintaining a list of all different weapon prefabs
- Maintaining a list of all different skybox materials
- Maintaining a list of all the different music audio clips
- Maintaining a prefab for the hyperspace windows
- Maintaining a prefab for the explosions
- Maintaining a prefab for the wrecked ships
- Maintaining a prefab for the AIController objects
- Playing/Stopping/Changing the audio track
- Loading all the database (for ships, weapons, system and ship player saved data)
- Changing the scene between the Loading Screen and the main Game
- Loading and configuring the scene elements during loading
- Triggering hyperspace travel animation and target system for the player
- Finding a hyperspace target for an AI ship
- Spawning hyperspace windows
- Exiting to main menu

- Simulating regular updates to the entire galaxy’s status
- Determining which region the player should be in
- Storing the player’s current ship state
- Increasing the player’s experience and levelling up
- Purchasing weapons
- Purchasing ships
- Resource Management
- Switching current ship
- Calling Allies
- Claiming planets
- Updating the vision of planets as seen on the map
- Getting all objects that may be targeted
- Saving the game
- Loading the game
- Generating the database from scratch for new games

10.3.4 Constraints

The object automatically constrains:

- Time between events like the calling of allies, how often to check where the player’s ship is, how often to simulate system wide battles
- Restrictions on when events occur like purchasing using the trading system for either ships or weapons based on the number of items or objects held and how many resources are being held
- Actions where it is possible that an object may be null are checked and taken appropriately to handle errors so that they won’t occur
- There must be only one Game Controller

There are more constraints, but they are handled by low level systems and do not need high level detailing in this document due to time based documentation constraints.

10.3.5 Composition

The GameController object composition is very simple. The prefab of the object simply contains the GameController script and the script handles everything from there. Additionally an audio source is attached to the controller for playing the music.

Attached to the script need to be:

- All ship prefabs in correct ID order so that they match the database contents
- All planet prefabs. Earth, the Supergate, and Asgard Tradeworld all need to be in specific locations, but all the rest it does not matter the order that they are in as long as they are always the same.
- All weapon prefabs in correct ID order so that they match the database contents
- All skybox materials in correct order so that systems requesting a particular skybox by ID can display the expected material
- All the music audio clips with the enemy based ones in the first 2 IDs and all others in the remaining IDs
- The hyperspace gate prefab
- The explosion prefab
- The wrecked ship prefab
- The AI Controller prefab

Not attached directly to the object, but still instantiated and controlled directly and entirely by the object is also the DataLoader script. The Data Loader script has been included under the GameController object here as it is really a sub section of the Game Controller’s database management. Another definitive feature that is used by everything including the Game Controller is the DataManager. It exists as the definition of all the data structures that the game uses and also assists with a couple of helper functions that will be identified in following sections.

The diagram on the following page shows the class definition of GameController at the top and then the struct definitions as specified inside of DataManager for the SaveData and ShipDatabase. These data structures are used heavily throughout the game. Give a lot more time if would have made sense to make them each individually a class for many of them. However a shortcut has been devised to make them structs and handle the manipulation externally. The Battle struct is defined and used within the GameController to manage the simulation of battles.

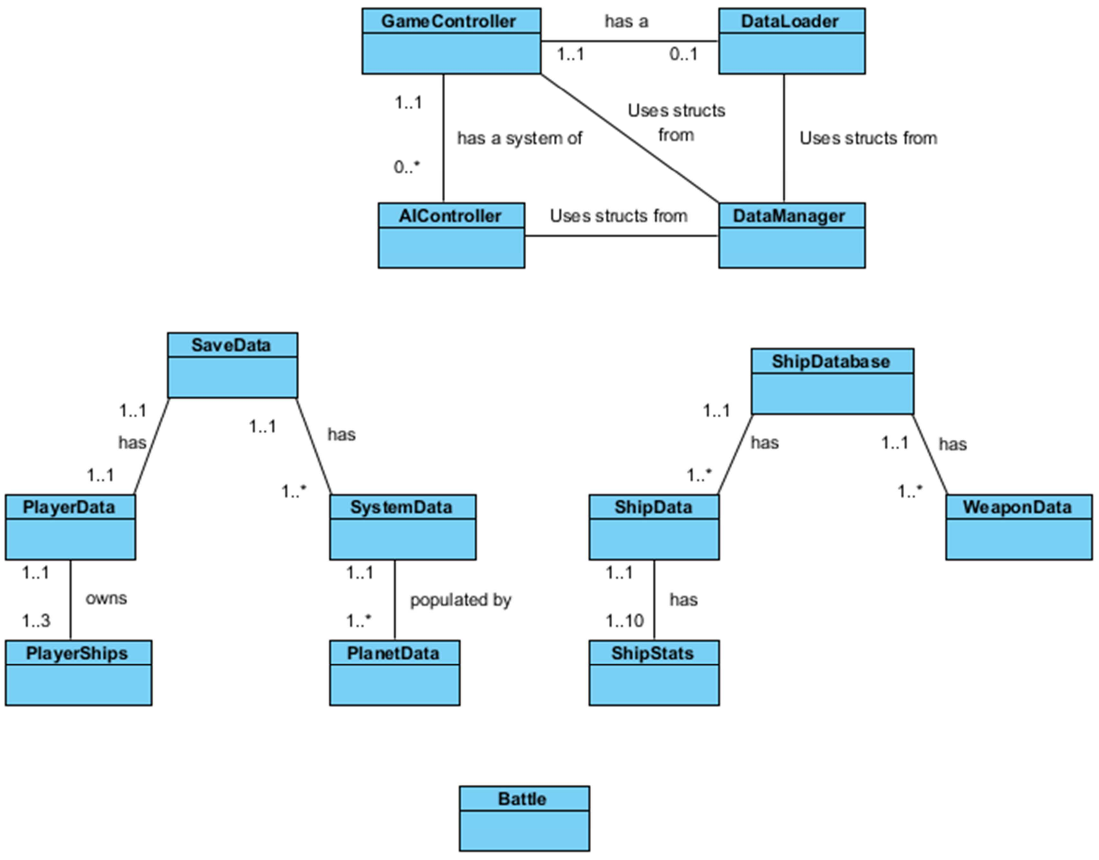

Figure 2 : Game Controller Composition

10.3.6 Uses/Interactions

The Game Controller must exist in the Loading Scene as a loaded prefab. As soon as the scene loads for the first time and the GameController spawns it will immediately begin configuring everything that the game needs to work.

10.3.7 Resources

The Game Controller requires the resources of:

- All prefabs as defined as in the composition section on the previous page
- All music tracks that are needed for playing
- The Game Controller script
- The Data Loader script
- The Data Manager script

10.3.8 Processing

This section has been broken down based on the three different scripts that exist. Most of the actual processing is done in the Game Controller script, but there is also some processing that occurs in the Data Loader and Data Manager scripts.

10.3.8.1 Game Controller script

The processing that the game controller does includes events for: when the object awakens, when the object is created, when the object needs to update, and when the level has loaded. The rest of the functionality is handled through the interfaces and shall be summarised there. Please refer to the relevant diagrams in the System Requirements specification appendix for a specific control flow of actions. This document given time would include more detail here too, but it is already a long document without inclusion of that information.

On awakening:

The object indicates to the Unity framework that it should not be destroyed on scene loading. This is how the object is maintained across all the scenes.

On start:

- Checking should occur to ensure that there are no other GameControllers found (as will occur when the loading screen is switched to on subsequent times) and if there are they should be purged.
- Instantiation of the Data Loader script and then loading of the player’s data that is saved for the system configuration, ship configuration. Then also the ship data base containing the weapon and ship information.
- If it is the first time that the player has started then they should be given control of the Prometheus automatically and have that configured for them.
- All other settings and variables should be initialised.

- Then jumping to hyperspace should occur with the target as Earth. This means the loading screen will be reloaded the first time.

On update:

- Updating the music should occur. If the player does not exist then the music should be random. If the player does exist and they are in combat the music should be set to combat music if it isn't already. If they aren't in combat and the music is combat music then it should be changed to non-combat music. And in either case if music has finished playing a new track should be switched to.
- Managing when to change out of the loading screen should be checked. If the timer has reached 0 for the countdown till change the game scene should be loaded.
- System wide battles should be simulated. See Update AI Planet Control in Figure 31 of section 11.28 of the System Requirement Specification for a high level overview.
- Finally also the current planet that the player is associated with should be updated if required. The player needs to be 20% closer to another planet than the current one for a change to occur. When the player changes system like this it should automatically update the vision of the systems.

On level loaded:

- Once the level has loaded with all the objects the spawning of the player is needed, so this should complete the hyperspace jump by spawning a hyperspace window and the player after spawning all of the planets and other ships that are required to populate the specific galaxy. Most of this code is handled externally by the AI Controllers that are responsible for overseeing each of the planets.

10.3.8.2 Other scripts

The other two scripts both are entirely based on interfaces so they shall not be summarized in functionality at all here.

10.3.9 Interface/Exports

Just like for the processing section this section shall split the information up between the three scripts.

10.3.9.1 Game Controller script

The game controllers interfaces may be considered as a set of different functions that are paired together to complete types of tasks.

Hyperspace:

- To immediately jump into hyperspace with no effect shown calling:
  void hyperspaceJumpTo(int systemID, int planetID) will switch to the loading scene and take the player to the designated system.
- To make the player jump through a hyperspace window with the effect using the void setHyperspaceTarget(int systemID, int planetID) will initiate the player ships ShipAI actions that make the animations occur.
- Once any ship AI or player is ready to jump through hyperspace (normally when visually they have reached the hyperspace window) the function :
  void performHyperspaceJump(GameObject obj, int targetSystem, int targetPlanet)
  may be called. This will move the ship around if an AI, and switch to loading if it is the player to complete the jump.
- AI ships may make use of the void setAIHyperspaceTarget(GameObject obj), and the method call will automatically choose an appropriate hyperspace target for the given object.
- For spawning hyperspace windows using the hyperspace prefab the void spawnHyperspaceWindow(Vector3 location, bool atLocation) will spawn a hyperspace window relative to the ship dependent on whether the ship is entering or exiting hyperspace and then returns the location of where the hyperspace window is.
- Used for hyperspace and other general spawning is a helper function too Vector3 getVector3InRangeOfPoint(Vector3 pos, float minDistance, float maxDistance)
  the function will determine a random point by adding a random value between the min and max to each of the numbers (positive and negative ranges. So specify min 30 and max 60 will actually add values between -60 and -30 as well as potentially 30 and 60 to the position.

Loading/Saving:

- Storing the player’s current ship state needs to be regularly done using the function void storePlayerShipState()
  It make it so that the status of the ship is retained between scenes and loading games.
- void saveGame() saves the current game data using the data loader object.
- void loadGame() loads the game data using the data loader object.

Purchasing:

- bool purchaseWeapon(int weaponID) will purchase the weapon based on the ID in the database if it is allowed and update the ship to include it. However it will return false if the purchase fails.
- bool buyShip(int shipID) will purchase a ship of the given ID and setup the default configuration for it in the save information.
- bool canBuyShip(int shipID) determines if the ship of the given ID may be purchased based on existing ownership and the resources of the player.
- bool canBuyWeapon(int weaponID) determines if the weapon of the given ID may be purchased.
- int[] getPurchasableWeapons() determines how many of each weapon ID may be purchased without considering the ability to pay for them.
- void addResources(int[] resources) increases the amount of resources held by the player that may then be used for purchasing. This should be used when looting ships.

Prefab collection:

- GameObject getPrefabShip(int shipPrefabID) gets the required prefab of a ship.
- GameObject getPrefabPlanet(int planetPrefabID) gets the required prefab of a planet.
- GameObject getPrefabWeapon(int weaponPrefabID) gets the required prefab of a weapon.
- GameObject getPrefebWreck() gets the wreck prefab.
- GameObject getPrefebExplosion() gets the explosion prefab.

General interface functions:

- void exitToMainMenu() destroys the game controller (otherwise game controller would come to main menu too) and then loads the main menu scene.
- void increasePlayerExp(int amount) will increase the player’s experience for their current ship and then automatically handle levelling up if required.
- void setCurrentShip(int playerShipID) changes the current ship to the specified ID and spawns that ship wherever it can with the correct configuration.
- void attemptCallAllies() will allow potential allies to be called in. If calling too often the function will show a decline message, and a random change will determine if they will decline the request or be on their way.

- void caimPlanet(int systemID, int planetID, DataManager.Race race) changes the controller of the specified planet to the specified race if it is allowed and then updates the player's vision of what is going on if they are allowed to know.
- void updateStatusPlayerVision(int systemID, int planetID) updates what the player sees in a given system for who is controlling , who the strongest enemy (or ally) is and the size of that force.
- void getPlanetList() gets the list of all planets in the current area of play.
- List<GameObject> getTargettableObjects() gets all the nearby objects that can be targeted.
- DataManager.ShipData getShipData(int shipID) gets the information about the specified Ship.
- DataManager.WeaponData getWeaponData(int weaponID) gets the information about the specified weapon.
- void setSkyBoxMat(int id) sets the skybox to the specified prefab id.

10.3.9.2 Data Loader script

There are four different functions for the data loader script that basically do all the saving and loading. The other two functions that are major ones generate the entire database for the ships, weapons, and initial new game system data and player data.

- DataManager.SaveData loadPlayerData(bool defaultSave)
  This function will load the initial data or populate the data with a new game depending on whether a NewGameFlag prefab exists (as would be spawned by the MainMenu). The function loads all of the data using XML deserialization and returns the package.
- DataManager.ShipDatabase loadShipData()
  Loads the ship database and creates it if it doesn't exist. Data is loaded using XML deserialization.
- void saveData(DataManager.SaveData data)
  Save the system and player data using XML serialization.
- void saveData(DataManager.ShipDatabase data)
  Save the ship database using XML serialization.

10.3.9.3 Data Manager script

This script is mostly used to define the data structures used in the game thus managing the data. The interface static helper functions that may be called at any time though are as follow.

- bool isAlly(Race r1, Race r2) checks if the races are friendly toward each other.
- ForceSize getForceSizeFromCount(int count) determines a force size appropriate to the number of enemies specified.
- int getShipCountFromForceSize(ForceSize forceSize) does the reverse of the above and gets a number that can be used for a number of ships based on a force size.

To save writing out all of the definitions of the structs they may be viewed in the source code if desired. Under Assets/Scripts/GameControllers. Mostly they will be defined under the DataManager.cs code file.

10.4 Ship

10.4.1 Classification

Visible Mesh and player/AI objects

10.4.2 Definition

The ships are all of the flying objects that populate the scene. They get information about themselves from the Game Controller and populate an AI Controller to place them around the planet. The player is give control of one of the ships by simply flagging it as the player and attaching a couple of additional scripts to ensure some additional functions work.

10.4.3 Responsibilities

Ships have the following responsibilities:

- They must manage their current state of movement
- They must appear visually to show that they are existing
- They must manage their own properties like hull, shield, and power status
- They must manage the updating of attack targets
- They must manage the AI control of the ship if the ship is an AI ship
- They must manage the firing of their weapons, taking damage, and destruction

10.4.4 Constraints

Some of the constraints that are imposed on the ship objects are:

- How often events occur like healing, firing weapons, updating AI
- How many weapons can be firing
- How the ship can move
- What the ship can fire at

10.4.5 Composition

The elements that make up every ship object are:

- A mesh detailing the ship
- A diffuse illumination material that includes attached a diffuse texture for the mesh and a texture for the illuminated ship object.
- A collider
- A rigid body
- A link the ShipAI script
- Tagged as “Player” for player ships, “Enemy” for ships owned by enemy races, and “Ally” for ships owned by allied races.

Some ships that have engines as particle effects also include engine objects with the following properties:

- A particle effect with emitter, animator and render.
- The material and textures that are used for the particle effect
- A light source

The player ships specifically also require:

- Movement Control script

The ships that part of the game are:

- Asgard Mothership (Ally)
- Goa’uld Ha’tak (Enemy)
- Goa’uld Advanced Ha’tak (Enemy) – a scaled Ha’tak model
- Goa’uld boss Anubis’s Flagship (Enemy)
- Wraith Hive Ship (Enemy)
- Wraith Cruiser (Enemy)
- Wraith Super Hive (Enemy) – a scaled hive ship
- Replicator Ship (Enemy)
- Ori Mothership (Enemy)
- Prometheus (Player)
- Daedalus (Player)
- Asgard Mothership (Player) – same model, but modified

10.4.6 Uses/Interactions

Ships are spawned and managed by the AI Controller objects. They must be configured using the appropriate interface method to provide the ships with the information they need.

10.4.7 Resources

The ships require the following resources:

- Mesh for each ship
- Materials with textures for diffuse and illumination of the meshes
- Any resources required for the particle effects if they exist
- A prefab for each ship that contains the mesh, materials, collider, rigid body, ShipAI script, particle effects if required, and the Movement Control script if required.

10.4.8 Processing

The processing for ships is completed in two different scripts; the ShipAI script and the Movement Control script. ShipAI handles almost everything except for the player's target selection and commanding. The Movement Control script uses a popup interface provided by the HUD controller to control the ShipAI methods that make the ship change targets.

10.4.8.1 ShipAI script

There is only one primary processing method that isn't handled by the interface methods that is the Update() method. The update method should complete the following processing tasks.

- If a hyperspace window needs to be spawned for travelling to a hyperspace target spawn one
- If the ship is an AI ship and can retreat initiate actions using the Game Controllers setAIHyperspaceTarget method. The definition of CanRetreat may be found in section 11.23 of the system requirements specification. The table there outlines what allows a valid retreat.
- If the ship is infected the infection is updated for the cooldown if started and infection removed if the countdown has completed.
- Updating of the attack target takes place checking if there are enemies and therefore the ship is in combat still and then whether the selected target can be attacked. If infected a random target will be chosen, and if the ship is AI controlled a new target is selected using the process as defined in section 11.25 of the system requirement specification, and 11.26.

- Updating of the AI movement is completed such that the AI will manage how and where they should be moving and making themselves stop at certain points.
- Updating of the overall ship movement should then occur handling the reaching destination points for stopping and hyperspace jumps, and updating the current movement speed and direction of motion to then transform the location using the rigid bodies velocity.
- Weapon updating of cooldowns for each active weapon should occur and once each weapon reaches no cooldown remaining they should fire by creating a projectile and configuring it to travel toward the current target. Weapons should only fire within a particular range of the enemy ship.
- Healing will then also occur first healing the hull by 5% each second and once the hull is healed the same for the shield. This will only occur though when out of combat.

10.4.8.2 Movement Control script

The Movement Control script primarily detects when elements have been selected and displays the appropriate HUD component so that an action related to the ship may be completed. The processing that is done by the Movement Control is primarily done on beginning, on updating, and on the last update.

On start:

- Configure all of the pointers to the required to the objects in the scene

On update:

- Collect the current status of button presses and update the pointer to the HUD Controller.

On last update:

- If the left mouse button has been pressed a ray cast should be sent to determine if an object in the scene has been clicked. If one has then the HUD should be called for the selection window for what the player wants to do with the target. Otherwise if it is just empty space the player should begin travelling in that direction. If they have clicked themselves then they should become stationary.

10.4.9 Interface/Exports

10.4.9.1 ShipAI Script

The ShipAI script has the following interface methods available as grouped into types of calls.

Ship Configuration:

- void configureShip(int level, DataManger.Race race, DataManager.ShipData shipData, AIController controller, GameController, game) configures the ships entire set of values with the correct values based on the ships current level and other configuration properties.

Movement:

- void initiateMove(MoveMode mode, Vector3 target) allows for waypoint based target selection or travel in a direction
- void initiateMove(MoveMode mode, GameObject target) allows for movement in relation to a specific object
- void setMoveMode(MoveMode mode) allows for simplistic movement setting such as MoveMode.Stationary. This should not be used except when the person knows what they are doing though.
- void enterSupergate() begins movement toward and entry into "hyperspace" for the supergate.

Ship power and status:

- int getShieldPercent() gets the amount of shield as a percent
- int getHullPercent() gets the amount of hull as a percent
- int getShieldPowerPercent() gets the amount of shield power as a percent
- int getWeaponPowerPercent() gets the amount of weapon power as a perecent
- int getEnginePowerPercent() gets the amount of engine power as a percent
- int getPowerUsePercent() gets the amount of power used as a percent
- bool powerIncreaseAllowed(int amount) checks whether an increase of power of the specified quantity is allowed
- void updatePowerStatus(int shieldPower, int weaponPower, int speedPower) takes the percent values of power allocation between the three different types and applies them updating the ships power use.

Weapons

- void damageShip(DataManager.WeaponData weapon, bool isPlayerShot) will damage the ship and if it is a player shot the target will be tagged so that on destruction the player gains experience. The damage done is calculated based on whether the shield is up or not, and also takes into account he quantity of shield power that has been allocated. Damage is applied to the shields first and then residual or all damage is applied to the hull as needed. When the hull falls below 0 if the ship is the player they are set to fly back to Earth, and if they are any other ship they spawn explosions and a wrecked ship using the prefabs on the game controller.

- void beginPlayerCancelInfection() if the player is infected then they will have a button to press in the interface that will begin the cleansing of the ship. That should call this method to start a random cleansing cooldown timer.
- void equipWeapon(int weaponID, int count) equips the count of the weapon ID type to the ship adding to the value if there are some already equipped or updating the existing numbers if some exist. Equipped weapons are disabled by default.
- void enableWeapon(int weaponID) searches through and finds an inactive weapon of the ID if it exists and enables one of it. If unable to enable a warning will appear.
- void disableWeapon(int weaponID) searches through and find an active weapon of the ID if it exists and disables one of it.
- int getNumberOfFiring(int weaponID) gets the number of currently active weapons of the specified ID.

Movement Control script

The movement control's callback methods allow the HUD to reply about what has been chosen as an action.

- void selectedMove() makes the ship fly at the target object.
- void selectedOrbit() makes the ship fly into an orbit around the target object.
- void selectedAttack() makes the ship fly at the target object.
- void selectedLoot() loots the selected object. Will display a message indicating too far away or if in combat that they can't loot it.
- void selectedUse() makes the enter stargate trigger if close enough.

10.5 AI Controller

10.5.1 Classification

Invisible Controller

10.5.2 Definition

The AI controller object’s purpose is to manage the population of AI that exist within a single current system that the player has entered. It will manage the existing ships and allow for additional entry of other ships. The main purpose is to coordinate the targeting system to manage which ships should be targeting each other.

10.5.3 Responsibilities

The responsibilities of this object should include:

- Management of a single planet
- Management of all the ships both AI and player that are associated with that planet in space.
- Spawning of collections of ships
- Spawning of individual ships
- Managing the collection of wrecked ships associated with the planet; including removing those objects when needed
- Managing of the capturing of the associated planet
- AI target selection and target tracking
- Updating the vision status for the planet when circumstances change

10.5.4 Constraints

The controller should only manage and allow interaction between ships in the current controller.

10.5.5 Composition

The AI Controller is just a prefab with the AI Controller Script attached to it.

10.5.6 Uses/Interactions

AI Controller instances are spawned for each planet that is spawned and are managed by the GameController for when they need to be destroyed.

10.5.7 Resources

The AI Controller script and AI Controller prefab.

10.5.8 Processing

The only event that AI Controller handles processing by itself is the update.

On update:

- Update the planet capture status. If capturing is in progress claim the planet or check if the capturer is still not in combat if they are not alone (therefore with enemies) the capturing must be stopped. If capturing has not yet begun then a check should be done to ensure that there is a single type of allied group of ships at the planet. If this is the case then capturing can begin.

10.5.9 Interface/Exports

The interfaces that are available within the AI Controller are mostly related to target acquisition and management of the collection of ships.

Targeting Interfaces:

- GameObject selectRandomTarget(DataManager.Race race) selects a random target that is not friendly with the provided race.
- GameObject selectRandomTarget(GameObject notthis) selects a random target regardless of race that isn’t the specified game object.
- int getEnemyCountOfRace(DataManager.Race race) gets the number of enemies that the specified race has.

- List<GameObject> getEnemyTargetOfFriends(DataManager.Race race) gets the targets of the specified races allies that are enemies
- List<GameObject> getAllTargettingShip(GameObject ship) gets all the ships that are currently targeting the specified ship
- List<GameObject> getEnemyTargetingFriends(DataManager.race) gets all the ships that are currently targeting friendly ships
- List<GameObject> getAllies(DataManager.race) gets all the allied ships
- List<GameObject> getEnemies(DataManager race) gets all the enemies
- GameObject getNewTarget() uses a combination of the previously specified methods to determine an optimum target to attack
- Vector3 getVector3InRangeOfPoint(Vector3 pos, float minDistance, float maxDistance) gets a random point within a cube that has a cube hole in the centre.
- List<GameObject> findShipsWithInRangeOf(Vector3 pos, float range) gets all the ships within the specified distance of the specified position.
- List<GameObject> getTargetableObjects() gets all the different objects that can be targeted by the player including enemies, wrecks, allies, and the planet.

Ship List Management:

- void spawnShip(int level, int shipID, DataManager.Race race, bool instant) spawns a ship of the specified type with a hyperspace window if required and adds the ship to the internal lists so that it may be targeted by other ships.
- void spawnShip(GameObject ship, bool instant) adds an existing ship to the AI Controllers lists using the same spawning process as previously specified.
- void addShip(GameObject ship) adds an existing ship to the lists without processing a spawning at a location.
- void removePlayerShip() removes specifically the player’s ship from the lists
- void removeShip(GameObject ship) removes the specified ship from the lists.
- Bool isPlayerShipHere() gets if the player’s ship is currently here
- DataManager.ForceSize[] getRaceForceSizes() gets the sizes of the forces in the current system and converts them to a force size.
- void setupShipCollection(DataManager.ForceSize[] forceSizes, bool instant, bool boss) spawns ships for every race that has a non-zero force size. Also spawns boss if required too.

Other:

- void updateVisionStatus() updates the vision status of the planet that the AI Controller is governing.

10.6 Planet Object

10.6.1 Classification

Visible Mesh

10.6.2 Definition

The planet object is simply a sphere mesh that will appear with a variety of different possible textures. Planets may be selected as targets for the purpose of travelling toward them, but are otherwise just aesthetic.

10.6.3 Responsibilities

Planets are responsible for providing something gigantic enough to show the player that they are where they should be. The AI ships are spawned nearby planets but planets have not functionality directly on them that handles anything. All planets though must be “selectable” by the player based on properties to be used by other objects.

Most planets are to be standard and not anything special. Exceptions to this are:

- Earth: This planet must have the specific appearance of Earth and allow interaction to provide the user with the Earth menu as defined in the HUD Controller.
- Asgard Homeworld: The Asgard’s primary planet that is designated as a place for the player to trade their collected resources for weapon upgrades and ships must have a unique appearance like Earth.

10.6.4 Constraints

There are no constraints other than that the Earth and Asgard worlds must appear unique. Due to the number of planets that have been collected the rest may be seen duplicated potentially.

10.6.5 Composition

Each planet object should include:

- Sphere mesh
- Material with diffuse, normal map (if available), and illumination textures using an illuminated bump map shader pipeline.
- A sphere collider that has a radius matching that of the object
- Tagged as "Planet" (or "Homeworld" and "Tradeworld" respectively for the other elements)

10.6.6 Uses/Interactions

The planets location may be used by the AI Controller to spawn ships relative in location to it.

The Game Controller will spawn the instances of the planet objects and indicate the spawned planet to the AI Controller.

10.6.7 Resources

Resources for this object include:

- Diffuse textures for each planet
- Normal bump maps for most planets
- Illumination textures for all planets
- Sphere mesh (provided internally by Unity)
- Material object that combines the diffuse texture, normal map, and illumination textures
- Prefab component that combines the material, sphere mesh, and collider

10.6.8 Processing

None, as the object's collider and other interfaces are handled by Unity and other scripts as required. This object is by itself completely inert.

10.6.9 Interface/Exports

None.

10.7 HUD Controller

10.7.1 Classification

Invisible HUD Controller

10.7.2 Definition

This object is intended to be how the player controls the ship they are flying. The HUD will manage nearly all of the user input and display all required information for the user to interact with the game.

10.7.3 Responsibilities

- Provide a means for the user to manage all ship and game environment functions
- Display and Manage power controls
- Display players resources
- Display current target details
- Display ship shield and hull status
- Display ship name, level and current XP
- Display help functions, Call allies and purge ship
- Display and control all equipped weapons
- Display and navigate to all available planets, ships and other game objects
- Provide an interface to handle Trade
- Display a map of the game universe and allow hyperspace travel
- Provide an interface to switch current ship, load and save game states
- Target selection menu for when game elements are selected
- conformation window for decision checks
- Display warning messages and other game text in a colour coded scrolling text method.

10.7.4 Constraints

The HUD must work with all other game scripts and provide a way for all elements to communicate both ways. Nearly all information displayed is from external scripts, the GUI assumes that all information is available upon request and must be ready to display information when externally requested.

10.7.5 Composition

There are 11 main GUI elements all requiring their own scripts and design assets. Within each element there are many other GUI elements used to provide different visual displays and interactions. These include:

- Text
- Textures
- Images
- Buttons
- Selection lists
- Scroll bars
- Windows
- Groups
- Labels

10.7.6 Uses/Interactions

Within the main Game environment there are 11 GUI elements:

- Control Panel
    - The main user control panel at the bottom of the screen, used to control energy and display target/resource/ship health details.
- Confirmation Window
    - Simple window displaying a question with yes/no buttons.
- Display Messages
    - Scrolling Screen messages colour coded to importance.
- Earth Panel
    - Full screen display allowing user to load/save game and switch current ship.
- Help Menu
    - Simple 2 button panel used to call Allies or purge ship from replicators.
- Map Panel
    - Full screen menu displaying all planets in the game universe, all details for each planet and the ability to hyperspace to a selected planet.
- Object Panel
    - Displays a selectable list of all available planets, ships and other game objects with the ability to move to attack and orbit the selected object.
- Player Info Panel
    - Simple panel to display the player's ship name, level and current XP level.
- Target Select
    - A small window that opens next to the mouse when the user selects a game object with a list of available actions such as Attack, move and orbit.

- Trade Panel
  - A full screen display that allows the user to use collected resource to purchase other ships and weapons.
- Weapons Panel
  - Displays all available weapons with the ability to enable/disable and assign the amount of each weapon to fire.

10.7.7 Resources

- Textures
  - Control Panel
  - Hull image
  - Power bar (blue)
  - Shield bar (white)
  - Main panel image
  - Resource images
  - Small panel (for help and ship details)
  - Side panels (for weapons and Object list)
  - Top Panel (for trade, earth menu and map)
  - Standard buttons
  - Weapon buttons
  - Weapon images
  - Ship images

10.7.8 Processing

Most GUI components require the use of Start() to initialize their resources, Update() to check for clicks or key presses, and OnGUI() to render content to the screen. This information has not been included in full for all the scripts due to the large amount of content. Please see the game for further details.

10.7.9 Interface/Exports

See the above too.

10.8 Supergate Object

10.8.1 Classification

Visible Effect/Mesh

10.8.2 Definition

The object is responsible for existing as a visible point of entry to the Ori galaxy and for exiting back to the main galaxy. It uses most of the same properties of the planets.

10.8.3 Responsibilities

Sit in space looking pretty. Also allow for interaction via scripts on other objects to use as a portal to travel to the other of the pair of supergates.

10.8.4 Constraints

The supergate should be stationary, but otherwise there are no other important constraints.

10.8.5 Composition

The super gate object should consist of:

- A supergate mesh
- A material with textures for the diffuse, and illumination using a self-illuminating diffuse shader
- A box collider that is used for checking if the gate has been selected.
- The tag for the object set to “Supergate” to indicate what it is.

10.8.6 Uses/Interactions

The gate itself does not provide the interaction that is all handled via other scripts and objects. In particular the player's ship objects. It should be attached to the GameController object for spawning of the prefab. In terms of spawning it is considered to be a "planet" and takes the place of a planet when the appropriate instance occurs for needing to spawn it.

10.8.7 Resources

Resources should include:

- A diffuse and illumination texture
- A material that combine the two textures
- The supergate mesh
- A prefab component that includes the material, mesh, and collider

10.8.8 Processing

No processing is required on this object.

10.8.9 Interface/Exports

There are no interfaces for this object.

10.9 Projectile

10.9.1 Classification

Particle Effect (except for the rocket which is a mesh and particle effect)

10.9.2 Definition

An effect that has a velocity in a particular direction normally that may contact with ships causing possible damage. There are a number of different damage types and effect types that apply to this object.

10.9.3 Responsibilities

The responsibilities of this object include:

- To show visually that there is a form of projectile moving through space
- To check for collisions with objects and destroy itself and damage the object if that object can be damaged (only applies to ships)
- To correctly move toward the designated target

10.9.4 Constraints

Projectiles should not collide with other projectiles and they should not collide with the ship they were fired from. The projectiles do not know how much damage they do or how they move, this information is given to them when they are spawned via the projectile data script from database information.

10.9.5 Composition

There are two different variations of projectile type. All except the rocket are just particle effects for the visuals.

All projectiles have:

- A particle animator
- A particle renderer
- A particle emitter (ellipsoid type)
- One or more textures to provide the effect detail
- One or more materials to apply the textures in the a particle systems with appropriate shader configurations to provide needed effects
- A sphere collider for detecting collisions
- Tagged as "Projectile" and on a separate layer from other objects
- A point based light source
- An audio source
- An audio clip
- A rigid body component
- The Projectile Data script to handle all the interactions

The rocket projectile in addition to all of these properties also should have:

- A rocket mesh
- A diffuse material with diffuse texture to apply to the rocket mesh

10.9.6 Uses/Interactions

There are 10 different projectile types:

- Asgard energy beam
  - Appear as a beam like effect that is long and blue-ish
- Asgard energy weapon
  - Appear as an orb of light blue energy
- Goa'uld energy weapon
  - Appear as a golden/yellow orb
- Human projectile
  - Appear as a smaller series of shots
- Large Wraith energy weapon
  - Appear as a large purple-ish orb

- Small Wraith energy weapon
    - Appear as a small purple-ish orb
- Ori energy weapon
    - Appear as a beam of yellow light
- Replicator energy weapon
    - Appear as a light blue/green tinged orb
- Replicator special attack (infection weapon)
    - Appear as a green orb
- Rocket
    - Appear as a rocket with fire trailing out the back end

Each of the prefabs for these should be attached to the Game Controller. They are spawned by the ShipAI class on the Ships when the fire weapon action triggers. The ships pass all relevant information about the target to the projectiles and they also configure the collision ignore between the firing ship and the projectile.

10.9.7 Resources

- Materials and textures for each particle effect
- Mesh, materials and textures for the rocket object
- Prefab objects for each projectile including the: particle effect creators, the material/s, the collider, the light source, the audio source, the audio clip link, the projectile data script and if required also the rocket mesh.
- The Projectile Data script

10.9.8 Processing

The internally for projectiles is all handled by the Projectile Data script. The three events that apply to this object that the script handles are that of: when the object is created, when the object needs to update, and when the object collides with another valid object. These are all handled automatically by Unity's system calls.

- On creation:
  When the object is created the object needs to find all other projectiles and configure the physics system to ignore the collisions with those objects colliders.
- On update:
  When the object updates the object's "life" needs to decrease and expire destroying the object after a period of time. Also if the object is a rocket then it should be directing itself toward the ship (for if the target has moved).

- On collision:
  When the object collides with another object it should check if the object is a not a ship and then just destroy itself if that is the case. Otherwise it should damage the ship and destroy itself.

10.9.9 Interface/Exports

Most of the interfaces are handled by Unity automatically using the methods Start(), Update(), and OnTriggerStay(). The following those is an interface that must be called as soon as possible after instantiating an instance of the projectile:

public void configureProjectile(GameObject parent, DataManager.WeaponData weaponData, GameObject target, bool firedByPlayer)

The interface allows indication of the target ship and who fired the object with the parent and target variables. The weaponData variable contains the damage and weapon type information. And firedByPlayer is used to allow tagging of ships so that when they are destroyed the player gains experience.

10.10 Explosion Effect

10.10.1 Classification

Visible Effect

10.10.2 Definition

An effect that may be created at a location that will enact an explosion erupting.

10.10.3 Responsibilities

The only responsibility is to provide an effect to enhance the appearance of the destruction of ships within the game.

10.10.4 Constraints

This object is used from the supplied explosions pack provided with Unity. It is constrained to appear for only a brief period of time as defined by the object.

10.10.5 Composition

The object’s composition includes multiple scripts, prefabs, and materials that create the effect as is to be seen in the game. For more details the importable package may be found at: http://unity3d.com/support/resources/unity-extensions/explosion-framework.html

10.10.6 Uses/Interactions

The customised explosion prefab should be attached to the GameController so that other objects may get a pointer to it and instantiate a new instance. The object controls itself entirely so simply spawning it triggers the explosion effect.

10.10.7 Resources

The resources as supplied in the detonator package with a customised prefab that enables the visual and script features required for the explosion to work without impacting on other objects.

10.10.8 Processing

Processing is not handled by our code for this object other than instantiating an instance of it. See composition.

10.10.9 Interface/Exports

None required.

10.11 Wrecked Ship

10.11.1 Classification

Visible Mesh

10.11.2 Definition

A broken ship that represents that the ship that was there was destroyed in the combat. This object may be looted for an amount of resources. It will disappear after leaving the scene.

10.11.3 Responsibilities

The responsibilities of this object include:

- To provide a visual representation of a generic destroyed object
- To provide a quantity of resources that the player may collect

10.11.4 Constraints

The object itself does not provide the interaction that triggers the collection of resources. The resources may also only be collected once. The object once spawned should remain stationary.

10.11.5 Composition

The wrecked ship should consist of:

- A mesh
- A diffuse texture and illumination texture for the mesh
- A diffuse illumination material that uses the textures
- A rigid body component
- A box collider (as it is easier to click in the area of the mesh than to click On the mesh itself for this object)
- The Lootable Ship script
- Tagged as “WreckedShip” for required components to understand what the object is

10.11.6 Uses/Interactions

This object like the others should also be placed on the Game Controller object and is instantiated by ships that are about to remove themselves due to destruction. The object requires instantiation and linking to an AI Controller. Also setting the resources on the object so that there is something to be looted must be done.

10.11.7 Resources

- A mesh for the wreck of the ship
- A material and texture resources to apply to the mesh
- A Lootable Ship Script
- A prefab that combines the textured with material mesh, a rigid body, box collider, and the lootable ship script

10.11.8 Processing

All the processing that is done is handled via the provided interfaces.

10.11.9 Interface/Exports

The interfaces that may be used should include:

public void setResources(int[] resources)

This should be used to configure the number of each resource that the object holds until the loot is claimed.

public int[] getResources()

This is an optional method for use that may be used if any object wishes to know how many resources are there without taking them.

public bool isLooted()

This method can be used to check if the object has already been looted.

public void claimLoot()

This method is the main interface method as it provides the means of claiming the contents of the loot. It automatically adds the resources to the player’s stores.

10.12 Camera

10.12.1 Classification

View Window

10.12.2 Definition

The camera provides the view of all the objects in the scene. The main changes made are the attaching of skyboxes and how the camera may be controlled by the player. This object is referring only to the main camera in the game. There are cameras that exist in the main menu and loading screens too, but they are less significant and have therefore not been detailed here at all.

10.12.3 Responsibilities

The camera is responsible for displaying everything in the scene.

10.12.4 Constraints

The camera may only zoom out to a particular distance from the player.

10.12.5 Composition

The camera includes:

- A GUILayer for displaying the interface elements
- A Flare Layer for some of the effects
- An audio listener to get relative 3d audio effects
- A skybox to customise the surroundings seen in the distance
- A camera script to control the positioning and movement of the camera

10.12.6 Uses/Interactions

The camera is self-governing of position and updating apart from the updates configured by the camera script. The skybox materials are attached by the GameController.

10.12.7 Resources

Only the camera script is required directly for use as a resource. The Skyboxes may be considered as resources too, but they are governed by the GameController.

10.12.8 Processing

The camera script only has two main functions that process information. The Update() and LastUpdate() functions. The Update() occurs during updates as normal, and LastUpdate() occurs after all other updates.

On update:

The target of the player's ship as the camera's look at position is calculated. Also scrolling with the middle mouse button to zoom in and out should be managed in this code as well.

On last update:

The use of the mouse button being held down should allow updating of the cameras rotation around the ship object.

10.12.9 Interface/Exports

There are no interfaces that are significant for this object.

10.13 Skybox

10.13.1 Classification

Background

10.13.2 Definition

A visual effect that make it appear that the distance has something further that may be obtained but won’t change even if you keep trying to go forward.

10.13.3 Responsibilities

The skybox’s responsibility is to display a variety of different scene backgrounds.

10.13.4 Constraints

There are no constraints on this object it is just a series of materials.

10.13.5 Composition

There are 6 different skyboxes that are to be used in the game. Different systems within the game use different skybox materials. Each skybox is maintained by a skybox material. Each material is made up of 6 different images for the up, down, left, right, forward and back directions. These textures together are blended to make it look good from any direction.

10.13.6 Uses/Interactions

The materials are to be stored on the GameController object and then placed into the scene by it when entering a system by attaching the material to the camera.

10.13.7 Resources

For each skybox there needs to be a material and the 6 textures that together form the skybox.

10.13.8 Processing

There is no processing required for this object.

10.13.9 Interface/Exports

There are no interfaces required for this object.

\

11 Glossary

Diffuse is a type of static texture

GUI is Graphical User Interface

HUD is Heads Up Display (another name for GUI)

Mesh is a 3d Model

12 Bibliography

http://www.cmcrossroads.com/bradapp/docs/sdd.html

http://www.marketwire.com/press-release/unity-named-the-1-game-engine-by-game-developer-magazine-readers-1518427.htm

http://unity3d.com/unity/system-requirements.html