

ETHERIA

*TECHNICAL DESIGN DOCUMENT*

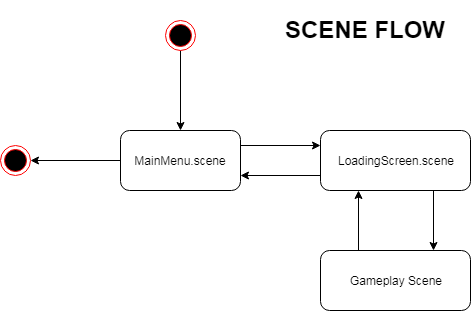
# FEATURES

* Real time strategy camera controls
* Base / depot / unit building
* Economy upgrades
* Point & click unit commands (w/ hotkey abilities for different behaviours)
* Mini-Map
* Dynamic user interface displaying player stats (resource counts, unit squads) as well as displaying
* Fog of war
* Unit boids/flocking behaviours

# TECHNICAL RISKS

The largest technical risk/challenge would optimization. Individually, all the mechanics mentioned are achievable within the timeframe allowed, however when all the features come together to make the core loop of the game, hardware resource demand will undoubtedly fill up very quickly. In return, having a game with features like containing many units on screen will need heavy focus on optimization – otherwise the game will not be able to run in a stable state and will ultimately flop in its experience.

# GAME FLOW



# Scenes:

* MainMenu
* LoadingScreen
* Level(s)

# GAME OBJECTS, SCRIPTS AND SYSTEMS

## InstanceManager

A singleton class that stores data that is to be persistent throughout the entire game session. Information such as user settings, match result data & any other required information will be stored here. (This is essentially our version of the GameInstance class from Unreal Engine 4).

## ObjectPooling

A singleton class which manages all GameObjects that will dynamically change in count. The main premise of this class is that any objects that are wrapped to this class are loaded at level start and disabled until needed. This is to ensure that objects are recycled when needed for spawning instead of using valuable hardware resources to create (or destroy) these object in memory. It is a method of optimization.

## Settings

A static class that stores game related, arbitrary, static variables such as invalid vector positions (used for object clicking), camera movement settings and other mechanic defaults.

## GameManager

A singleton class which persists for the duration of a gameplay match. It stores match related information such as the GUI widget references, as well as storing player starting values. When the match is complete, the information that passes through the *Game Manager* will be sent to either the *Instance Manager* or any of the other related classes as required.

## Player

A class that represents a player controller entity within the duration of a match. References to the player’s score, selected & grouped units and other general player specific functionality will be stored here.

## Input

A class that is attached to the player that manages the player’s controller. Functionality such as gamepad schemes or key bindings will be stored within this class and any input actions will pass through this class as a result.

## FogOfWar

This class stores all the functionality related to the fog of war mechanic within the game. It will be attached to all world objects and will show a sight line/bubble around the object. Casting a light through the “smoke” that hides everything in the game world. As an object traverses around the level – the fog of war will reveal a path throughout the map. Giving the player a need to search throughout the level to figure out the placement of points of interest.

## Abstraction

All Etheria specific GameObjects (units, buildings, upgrades, etc…) all inherit from this base class.   
The reason being is so that the selection GUI widget can correctly display the various items which represent the button to build these items. These must all essentially be the same class for the system to work – which is the main purpose of this class (aside from storing common data such as the build logo, name & descriptions).

## Selectable

This class inherits from *Abstraction* and any GameObject that can be clicked/highlight by the player will derive from this class.

## WorldObject

The *WorldObject* which inherits from *Selectable* is a base class that represents the visual representation of any of the Etheria specific GameObjects. Any props that units can interact with (such as attacking or garrisoning) must inherit from this class at some point.

## Core

The core is the central GameObject that the player will be defending throughout their gameplay sessions. The enemy units that the player fights throughout their gameplay sessions will be attempting to destroy this object as well. If the enemy units successfully destroy the object – the gameplay session will be complete, and the player will lose the match.

## BuildingSlot

This class represents the GameObject that buildings are to be placed on. There are multiple variations of the building slot (ones that are specific for turrets, bases & other general buildings) but they all contain the same functionality. It stores information such as the building list which determines what type of building slot these GameObjects are.

## Building

This class represents any depot/base/tower(s) within the game world. There are multiple variations of this object that have slightly different characterises, but they are all static objects with no movements attached to them. Some of the behaviours that differentiate the types of *buildings* are: spawning ai units, resource generation, upgrading economy.

## Unit

The *unit* class is the GameObject that represents individual agents within the game world. Some units will be grouped into *squads* and will have slightly different movement characteristics.  
All units will have a *weapon* attached to them, to give them combat capabilities as well as weighted targeting behaviours – as certain units will be more effective against specific units & will need to prioritise their targets accordingly.

## Squad

The *squad* class is a non-visualized entity that groups a bunch of *units* to give them boid movement behaviours. *Squad*s will be a “buildable” object for the player – as they will have the ability to spawn *squad*s of weaker infantry units that will act as 1 unit when attacking or completing their objective(s) set by the player.

## Humanoids

The *humanoid* class represents any ground infantry units that are within the game world. General characteristics include movement speeds as well as strafing abilities (which will be visualized as side stepping through animation). *Humanoids* will also have slightly different death animations as they are organic entities and will not explode when dying. Faked ragdoll animations will play when these objects are destroyed/killed.

## Vehicle

This class represents any vehicle (ground or air based) within the game world. General vehicle behaviours such as acceleration, steering speeds will be stored within this class. Vehicles will not be able to strafe by default unless specified otherwise.

## AirVehicle

This class represents any air-based vehicles within the game world. Functionality such as obstacle avoidance will be located within this class. The air vehicle(s) have the bonus of more versatile movement behaviours as they will be able to fly over large gaps of terrain within the game world as they are not restricted to the general ground path, as well as strafing for the same reason.

## Weapon

This class is a non-visualized script that stores information such as magazine count, firing rate, reload/burst delay times. 1 weapon will be attached to only a single unit at any given time. Weapons can have different firing mechanics such as using hit-scan, particle-based damage or physics-based *projectiles.*

## Projectile

This class represents any physically seen *projectiles* in the world. Projectiles will have differentiating characteristics such as the ability to track a target, be affected by gravity (firing in an ark for a mortar, as an example) or simply explode when contacting another physically represented object.

# SYSTEM REQUIREMENTS

* Windows 7 SP1 OS
* Graphics Card w/ DX10 capabilities
* CPU: SSE2 Instruction Set support

*System requirements are at subject to change as the project development progresses.*

# Third Party Tools

* Unity 2017.3.0f3
* Visual Studio 2018
* X-Input (Plugin)
* EasySave3 (Plugin)
* ProBuilder 2.9.8f3
* Maya 2018
* Adobe Photoshop CC2018
* Pixologic ZBrush 4R8P2
* 3D Code GL
* XNormal

# Coding Conventions

* All functions will have the first letter in each word a capital.  
  EG: ‘public void OnWheelSelect();’
* All public variables will follow the same convention as functions.  
  EG: ‘public string ObjectName;’
* All member (protected/private) are follow the same conventions as public variables, with the bonus of an underscore prefixing the variable name.  
  EG: ‘protected float \_Hitpoints = 0;’
* All functions are to be commented with a summary, listing of all parameter names as well as specifying their return types.

At the top of each script file we’ll have the creator of the script, the date it was created, and the date it was last modified. Any people who edit the script must change the last modified date as well as the name of the person last updated the script.

# Source Control

Each person will commit to a designated folder separated by each discipline (which is located at the root of the assets folder), when the content of their commit is verified by the leads to be up to build standard, it will be integrated into the main build. This is an extra precaution to avoid conflicts when people are working on the same script/mechanic.  
Each commit message must have any major changes listed to files as well as any major additions and what they do.

# Team Members

* Daniel Marton (Lead Programmer / Project Lead)
* Angus Secomb (Programmer)
* Joshua Peake (Programmer)
* Joshua D’Agostino (Lead Design)
* Siobhan O’Connell (Producer / Designer)
* Mitchell Trytell (Lead Artist)
* James Dunlop (Artist)
* Raph Buenaventura (Artist)
* Cameron Jones (Artist)