TECHNICAL OVERVIEW

PREPARED FOR

Jason Strohm

Two Sisters in the Wild, LLC

PREPARED BY

Keith Reis

CIS 411 Project Group - PennWest University

SUMMARY

This document contains important information about the product: Edge of Extinction - Virtual Card Game version 2.0. You can find brief overviews regarding purpose, functions, and a deep-dive into the framework and architecture used to create the game. With the hope of instilling a basic understanding and knowledge of how the game was designed.

Table of Contents

PROJECT OVERVIEW

Our project is a game based on the educational card-game titled ‘Edge of Extinction’. The focus of the game is to provide players with a fun yet strategically challenging experience. While also being an educational learning platform that teaches players about different types of animals, plants, geographical locations, and other facets of nature and wildlife. Therefore, our game’s target audience is centred around being used in an educational setting by teachers and students. We wanted to make the game as easily accessible to download and play as possible. As well as having a clean UI that is intuitive enough to quickly learn and begin playing.

As this project has already been previously worked on and attempted by 2 development teams before us, we chose to continue development of it in the Unity Engine. The editor version we used is *2019.3.12f1*. We have exported the game to an executable program that will be hosted on the card-game’s home website. We continued following and adhering to the past group’s coding architecture and design. Mainly their use of OOP(Object-Oriented Programming) methodology. As we also believe the use of objects and inheritance was the best and most efficient means of coding and implementing the necessary components for the game.

GAME ENGINE

Our game engine used for development was Unity - a popular choice among both independent and industry-level game developers. The IDE is developed, updated, and distributed by Unity Technologies™. Additionally, Unity offers a fully supported API that allows users to create custom editing tools & scripts. We collectively decided it would be best to initially continue to expand and fix issues with the game using the past development team's editor version: *2019.3.12f1*. We decided to test out porting the game to a newer editor version - *2022.2.14f1*. This version is not considered LTS (Long-Term Supportive) by Unity. Though our group believes this will change in the near future and it will most likely become LTS.

GRAPHICS ENGINE

We continued usage of Unity's 3D game engine for development of the card game. The 3D engine is adaptable and versatile enough to produce all of the scenes and objects needed for virtualization of Edge of Extinction. It also boasts built-in connectivity with Visual Studio and C# scripting.

The 3D engine community of Unity is also massively popular and active online. This was a great help to us, as many issues or questions we've run into while testing or debugging were most likely already solved on the Unity Forums. Therefore, our development team was able to rely on passionate members of the community for assistance in solving various problems.

CODE ARCHITECTURE

Our development team's focus was looking into the previous team's already existing coding architecture - and resolving any and all issues found within it. Extensive testing and planning was required in order for us to define problem areas, and then designate tasks based on who in our team understood what solutions could be implemented. Some script files had to be completely scrapped in order for us to create and apply our own solutions. Though we still closely followed and adhered to the past group's usage of OOP (Object-Oriented Programming) principles when creating new classes.

OOP PRINCIPLES

We used an object-oriented approach for development. Not only is OOP methodology an industry standard in game development, but also a

With our project goal being the creation of a virtual card game, it makes sense to use industry-standard for designing games, it's also also an important aspect for shortening strain on the resources and time allotted to our team for completion. Many aspects of the game can be reused as objects. Rather than having to create new implementations of certain game aspects that are recurring throughout gameplay. Examples of this are as follows:

- **Card Details** (ID, Name, Type, pointValue, etc.)
- **Card Effects**
- **Card Descriptions** (Sprite, Name, Details & Effects, Action Buttons)
- **Player Class** (Contains inherited subclasses: 'Human' and 'Computer').

TESTING

Instead of using an external automated testing software, human testing was our approach to discovering and eliminating any and all bugs. Due to the fact that we're creating a game played by humans, it's ideal to use the human testing approach. As we want to account for ways in which a human may approach certain aspects of our game and unintentionally cause errors or bugs to arise during gameplay.

We also made use of a bug-tracking tool via the Github repository page. The 'Issues' tab, allowed us to create individual issues we had found during development - and categorise them accordingly based on priority levels.

Our group used two specific types of software testing methods that best suited our current resources and timeframe for completing the game.

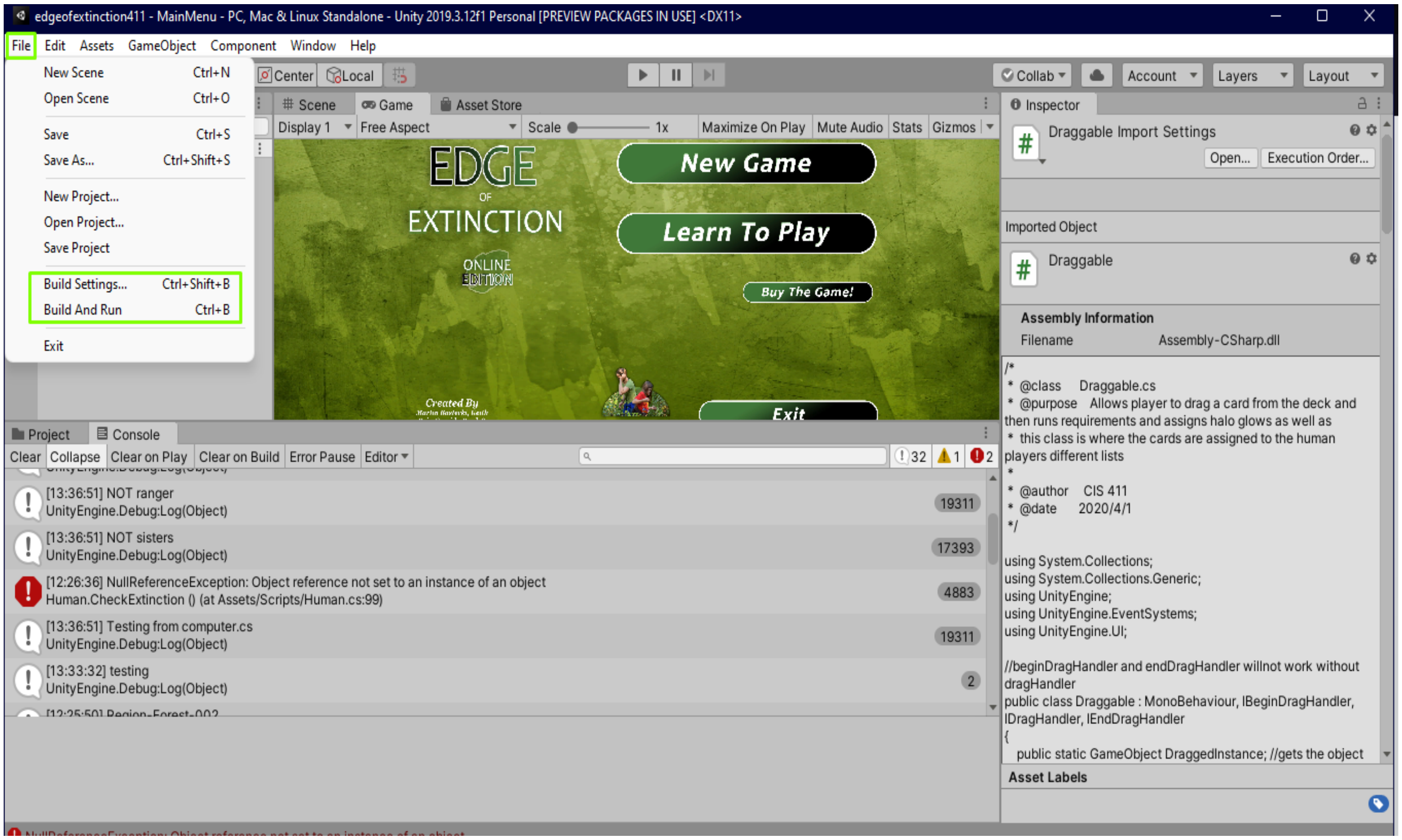
- **Unit Testing -** When we found a line(s) of code that we knew was responsible for a given bug; we tested those individual components. Rather than spending the time necessary to test the entire module that contains said code.
- **Regressive Testing -** As we made changes to the project's code, we wanted to ensure that our current versions of the game worked as intended/expected. This method of testing ensured that we didn't '*regress*' back to versions of our project that suffer from bugs or issues we've previously run into. Instead, the project was always moving forward - with each fix providing a more stable version than the last.

EXPORTING PROJECT

In order to create an executable program of Unity projects, you must build and package together the entirety of the project.

This can be done through the following steps:

1. **Select File -> Build Settings (Shortcut: Ctrl+Shift+B)**


edgeofextinction411 - MainMenu - PC, Mac & Linux Standalone - Unity 2019.3.12f1 Personal [PREVIEW PACKAGES IN USE] <DX11>
File Edit Assets GameObject Component Window Help
New Scene Ctrl+N
Open Scene Ctrl+O
Save Ctrl+S
Save As... Ctrl+Shift+S
New Project...
Open Project...
Save Project
Build Settings... Ctrl+Shift+B
Build And Run Ctrl+B
Exit
Center Local
Collab Account Layers Layout
Scene Game Asset Store
Display 1 Free Aspect Scale 1x Maximize On Play Mute Audio Stats Gizmos
EDGE OF EXTINCTION
ONLINE EDITION
New Game
Learn To Play
Buy The Game!
Exit
Created By
Inspector
Draggable Import Settings
Open... Execution Order...
Imported Object
Draggable
Project Console
Clear Collapse Clear on Play Clear on Build Error Pause Editor
32 1 2
Asset Labels

Assembly Information
Filename Assembly-CSharp.dll

```
[13:36:51] NOT ranger
UnityEngine.Debug:Log(Object)                                          19311
[13:36:51] NOT sisters
UnityEngine.Debug:Log(Object)                                          17393
[12:26:36] NullReferenceException: Object reference not set to an instance of an object
Human.CheckExtinction () (at Assets/Scripts/Human.cs:99)               4883
[13:36:51] Testing from computer.cs
UnityEngine.Debug:Log(Object)                                          19311
[13:33:32] testing
UnityEngine.Debug:Log(Object)                                          2
[12:25:50] Region-Forest-002
```

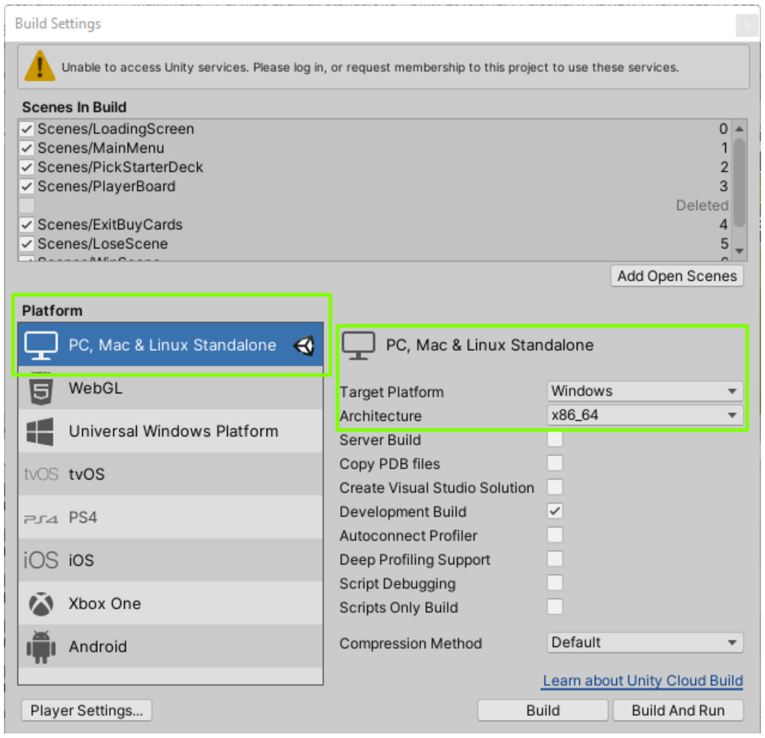
```
/*
 * @class    Draggable.cs
 * @purpose  Allows player to drag a card from the deck and then runs requirements and assigns halo glows as well as
 * this class is where the cards are assigned to the human players different lists
 *
 * @author   CIS 411
 * @date     2020/4/1
 */

using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using UnityEngine.EventSystems;
using UnityEngine.UI;

//beginDragHandler and endDragHandler willnot work without dragHandler
public class Draggable : MonoBehaviour, IBeginDragHandler, IDragHandler, IEndDragHandler
{
    public static GameObject DraggedInstance; //gets the object
```

NullReferenceException: Object reference not set to an instance of an object

2. Select Platform -> (PC, Mac, and Linux Standalone)


Build Settings

Unable to access Unity services. Please log in, or request membership to this project to use these services.

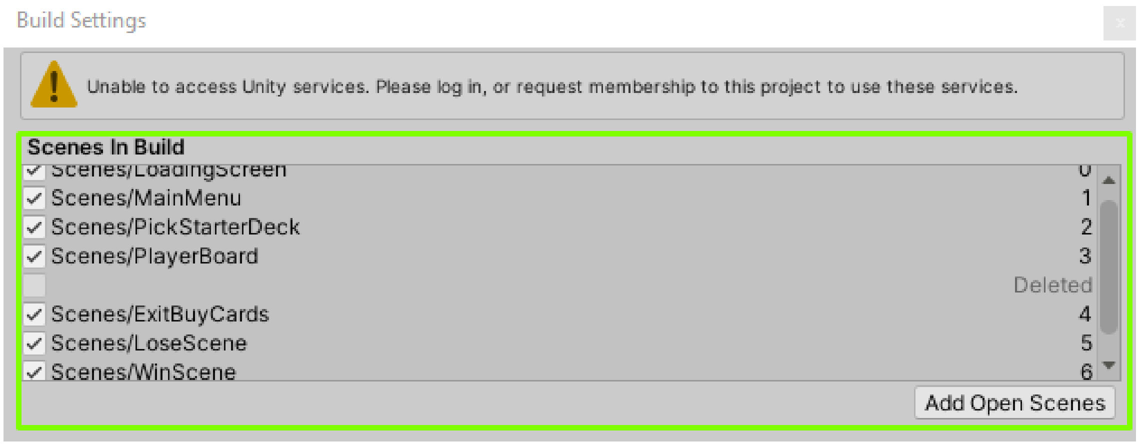
Scenes In Build

✓	Scenes/LoadingScreen	0
✓	Scenes/MainMenu	1
✓	Scenes/PickStarterDeck	2
✓	Scenes/PlayerBoard	3
		Deleted
✓	Scenes/ExitBuyCards	4
✓	Scenes/LoseScene	5

PC, Mac & Linux Standalone

Target Platform	Windows
Architecture	x86_64
Server Build	
Copy PDB files	
Create Visual Studio Solution	
Development Build	✓
Autoconnect Profiler	
Deep Profiling Support	
Script Debugging	
Scripts Only Build	
Compression Method	Default

3. Select Scenes to Include in Build

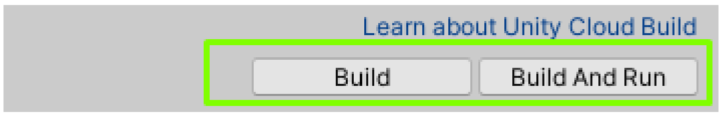

Build Settings

Unable to access Unity services. Please log in, or request membership to this project to use these services.

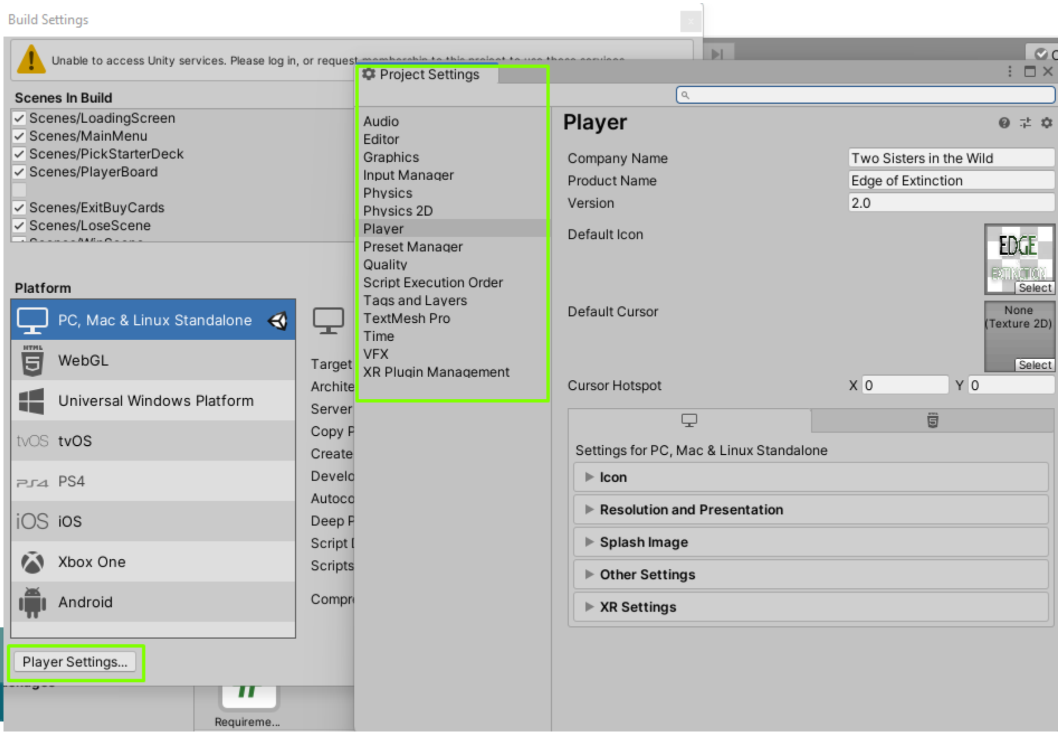
Scenes In Build

✓	Scenes/LoadingScreen	0
✓	Scenes/MainMenu	1
✓	Scenes/PickStarterDeck	2
✓	Scenes/PlayerBoard	3
		Deleted
✓	Scenes/ExitBuyCards	4
✓	Scenes/LoseScene	5
✓	Scenes/WinScene	6

4. **Press 'Build' or 'Build And Run'**



Advanced Settings: These settings are used for building the project with specific parameters or options in mind. For example, you may want to select '**Development Build**' if you wish to get an executable build that can be used for testing and debugging. By selecting **'Player Settings',** you can set various options for the final game built by Unity. A list of such options are highlighted in the reference image shown below.


Build Settings

Unable to access Unity services. Please log in, or request

Scenes In Build
- ✓ Scenes/LoadingScreen
- ✓ Scenes/MainMenu
- ✓ Scenes/PickStarterDeck
- ✓ Scenes/PlayerBoard
- ✓ Scenes/ExitBuyCards
- ✓ Scenes/LoseScene

Platform
- PC, Mac & Linux Standalone
- WebGL
- Universal Windows Platform
- tvOS
- PS4
- iOS
- Xbox One
- Android

Player Settings...

Project Settings
- Audio
- Editor
- Graphics
- Input Manager
- Physics
- Physics 2D
- Player
- Preset Manager
- Quality
- Script Execution Order
- Tags and Layers
- TextMesh Pro
- Time
- VFX
- XR Plugin Management

Player

Company Name	Two Sisters in the Wild
Product Name	Edge of Extinction
Version	2.0
Default Icon	
Default Cursor	None (Texture 2D)
Cursor Hotspot	X 0 Y 0

Settings for PC, Mac & Linux Standalone
- Icon
- Resolution and Presentation
- Splash Image
- Other Settings
- XR Settings

These sections provide a brief description and technical insight of the various scripting files contained within the subdirectory ‘Scripts’. Listed below are important class files and their contained functions, with explanations regarding their purpose, inputs, outputs, and overall functionality.

We implemented functions as a way to perform specific tasks or calculations within our game. GIving us the ability to implement many core features and aspects of the game that require usage of input/output parameters, what data is returned (if any), and special conditions and constraints

Classes, on the other hand, were implemented and used to define game objects that have specific attributes and behaviours. The classes section provides details on the attributes and methods associated with each class, as well as any inheritance relationships or interfaces that may be relevant.

Overall, the main goal of this critical section of the technical overview is to provide readers with a comprehensive understanding of our program’s capabilities. As well as how to use them effectively in any future instances of development.

GAMEMANAGER

The *GameManager.cs* class within the Scripts directory is a vital component of the overall program. Acting as the central hub for the entirety of the project. The class itself contains a wide variety of critical game object functionalities and operations. While also acting as the foundation for our OOP methodology implementation. A brief list of said functions and game objects are as follow:

- Creation and instantiation of *Player* subclass instances
  - Conversion of aforementioned instances into Unity *GameObject* instances. This is done to facilitate accessibility, persistence, and graphical rendering.
- Creation, instantiation, and loading of *Deck* and *Card* instances via *CSVParser*.
  - *CSVParser* breaks down components of *Deck* and *Card* and populates in-game the player's chosen deck. Populates *actionIDs* and *requirementIDs* for each card in temp deck object also.
  - Attaching *Deck* and *Card* instances to *Player* game objects.
- Game controls via primary loop and supporting functions
- Management of Unity-created game objects and background functions required for normal Unity program execution.

Overall, you can think of the GameManager as a system warehouse that stores important game objects and functionalities for core components of the project.

SCENES AND LOADING

(ChangeScenes.cs - HideShowBoards.cs - LevelLoader.cs)

These classes provide the necessary background operations of the game. Facilitating functionalities such as:

- Movement between engine-defined Scenes
- Displaying *Canvas* and *CanvasGroup* game objects.
- Loading persistent information across these

A more in-depth look at each class is as follows:

ChangeScene.cs: Changes active game scene to appropriate next scene using the *ChangeScene()* function; which takes in a string parameter called *sceneName*.

HideShowBoards.cs: Brings either Canvas or Game objects to the forefront to display to user. Objects are initialised and set to their appropriate canvas groups/individual canvases. Following this, *Show**canvas name* functions are used to display desired boards to the player. This is done by first disabling all other active canvases, and then setting factors necessary for the desired canvas to be displayed to true.

LevelLoader.cs: Implements a loading screen between MainMenu and PickStarterDeck scenes. This is checked by implementing a boolean value called *returnToMenu*, and setting it to true or false depending on whether or not the player is accessing the menu screen. There is also a function called *LoadLevel* which calls a routine to load MainMenu as well as the *LoadAsynchronously* function to create a timer counter and cause the loading slider to increase based on said counter.

GAME BOARD

(CardRetrievalFromDeck.cs - Draggable.cs - DoubleClickDescription.cs - HoverClass.cs)

These classes provide the necessary functionalities of moving and inspecting cards on the game board, such as:

- Ability to click and drag cards to designated areas on the board.
- Double click on a card to pull up a detailed description scene (as well as providing a button for activating special effect/action if applicable)
- Enable graphical functionality when hovering over cards with the mouse by altering card sizes.
- Drawing cards and displaying the pulled cards to the player’s hand.
- Destroying game objects

A more in-depth look at each class is as follows:

CardRetrievalFromDeck.cs: Overall, this script handles retrieving cards from the deck, storing related information about cards drawn, and initialising their relative sprites. This script facilitates the Draw() function, used in the Player class and it’s derived subclasses. The function CardDrawRandomizer() is also stored within this script. Which takes in the parameters for either: Human or Computer, followed by the current turn classification of pCurrentPlayer. It should also be noted that it is ensured that a Region type card is always received in the initial first round draw by both the Human and Computer. This script also contains the setSprite() function, taking in SpriteRenderer as a parameter. Based on the name of the card, a sprite will be created with the appropriate texture applied to it.

Draggable.cs: This script enables the Human player to click and drag cards. Placing said cards in their desired and appropriate placements on the game board. A game object called DraggedInstance is instantiated, which is used to track the object being selected and dragged. Once the player begins to drag a card, the requirements for the selected card are checked and a halo glow is assigned to show the player which panels the card can be dropped in. The functions onDrag() and onEndDrag() keep track of when a card is picked up, and when it is set down. While the function outOfCards() is checked at the end of the onEndDrag function. outOfCards checks to see if the CardDiscarded value is set to false, as well as if you've played your last remaining card in your hand. If these requirements are met, the player's turn will end and the computer's turn will begin.

DoubleClickDescription.cs: The functionality of this script pertains to displaying information of individual cards when selected by the Player. By double clicking a card on the game board, a new canvas/scene containing that card's relative details will be displayed. It contains the DestroyGameObject() function, which when called will remove the desired object (such as a card) from the game board if needed. The OnPointerClick() function will check if a card was clicked twice. If so, it will set the name, image, and description of the selected card. By using a wide variety of if/else statements to check what text the game object contains. For example, if (nameholder.Contains("Plant")), it will reset the description text in case there exists a Plant card with no description. Then it will go through an array of the PlantPlacement section on the board and match the name to one of the cards placed by the Player or Computer in said section. ImageOfCard.sprite = gameObject.GetComponent<SpriteRenderer>().sprite is placed at the end of this function in in order to place an image of the selected card within the description scene. At the end of the script is the Description() function. As given by the name, this function is used for adding in the card description based on if statements that check the type of the selected card.

HoverClass.cs: The main function of this script is for increasing the card size based on the position of the player's mouse cursor. For example, the OnMouseEnter() checks if the cursor is hovering over a card. If it is, then the sorting order of the game objects is stored - then set to max so that the hovered card always appears on top when rendered. Finally, the render is transformed to become bigger by using the following code: Rend.transform.localScale += new Vector3(0.4F, 0.4F); Then, the OnMouseExit() function will be used once the cursor has stopped hovering over a card. Resetting the sorting order of the render back to its original value, and using a negative version of the same render transformation code instead of the aforementioned positive version.

CARD UTILITIES

(Deck.cs - Card.cs - PickDeck.cs)

These classes handle various card-related utilities throughout the game. Both the Deck and Card scripts provide access to the vast majority of core game information. These scripts are arguably the most important ones out of the entire project in regards to providing the core of gameplay.

Deck.cs: A 'Deck', should be considered as a named collection of specific *Card* objects. Each deck has the following attributes attached to it:

- Identifying string and id (*deckName* & *deckId*)
- Unique color (*deckColor*)
- List of cards contained within the deck (*cards*)
- Specified player (once selected from *PickStarterDeck* Scene)

Two players cannot use the same Deck at the same time. Thus if I choose the 'Clarion River' Deck, the CPU must choose any of the other 3 decks. Within the Deck() function, all deck attributes are set to default values. Which are changed based on the selection of the Player and Computer via the *PickStarterDeck* Scene.

PickDeck.cs: The four decks included within the game and their referred to deckNames are as follows:

- Allegheny Forest Deck (*allegheny*)
- Appalachian Homestead (*appalachain*)
- Peat Bogs (*peat*)
- Clarion River (*clarion*)

As the game currently stands, the player is restricted to choosing the 'Allegheny Forest' Deck. This is due to the fact that implementing card effects for the other 3 Decks did not meet our group's project scope. Our group did not want to include the ability to choose other decks in the game without their cards having the correct requirements and card effects. Since the player loses the ability for any strategic gameplay by choosing decks that have cards without their functioning gameplay actions.

Card.cs: A ‘Card’ is considered a digital renderization of the physical cards used in real-world Edge of Extinction matches. It contains specifications for all informational attributes the physical card would contain. The following attributes are attached to a singular card:

- Identifying ID, name, and type (*cardID*, *cardName*, *cardType*)
- Associated sprite image of the physical card
- Point value(s) gained or lost when played (*pointValue)*
- Wide variety of traits & characteristics based on the associated wildlife of the card (ex. *kingdom*, *division*, *phylum*, *family, plantType*, *animalDiet*, etc.)
- Notes or extra information about card (*cardNotes)*
- List of associated standing/special card effects (*reqID & actionID*)

A string array is passed by the CSVParser script which reads the associated text asset information of the card into the Card() function; returning the given card values. There also exists a default Card() constructor, which initially sets all card attributes to default values before reading them in from the text asset.

These Card attributes are absolutely integral to accessing information about specific cards throughout various functions and scripts of the project. Such as checking which card is in a specific placement on the board via cardName, or executing an effect on all cards of a specific cardType. Another example is checking for requirements to play a card based on what type of wildlife has been placed on the game board.

REQUIREMENTS CHECKS

(Requirements.cs)

The Requirements.cs class has been refactored multiple times by past groups as well as ours. Our main goal when restructuring the class was removing any redundancy found throughout the code. There were multiple requirements listed that had been repeated 5-6 times one after the other, and created unnecessary bloating within the class. These have since been removed, making the class much more cleaner and organised.

Overall, this class is used to specify and check that the *Card* object that the *Player* (either *Human* or *Computer*) is attempting to interact with has its requirements met prior to interaction (ex. placement on the game board).

FILE PARSING

(CSVParser.cs)

File parsing classes such as CSVParser.cs are used to read-in information from several Comma-Separated-Value document files. These files are found within the "Assets/Data" project solution directory within Unity.

How File Parsing Works in Edge of Extinction:

By parsing the included files, it populates the necessary information/attributes tied to game objects such as *Deck* and *Card*, as well as important checks for *Requirements* and *Actions* associated with all cards used throughout the game. This is functional by implementing the usage of the native C# *StreamReader* class. Which reads the included CSV files in the *Data* directory, and returns each row of the file to a function. Said function breaks down each row into numerous strings. These given strings are individually delimited with commas, and used in the creation of the aforementioned game objects and checks. Thus becoming integral information stored within the associated class instances (*Deck.cs*,*Card.cs*,*Requirements.cs*,*Actions.cs*).

OPENING URLs

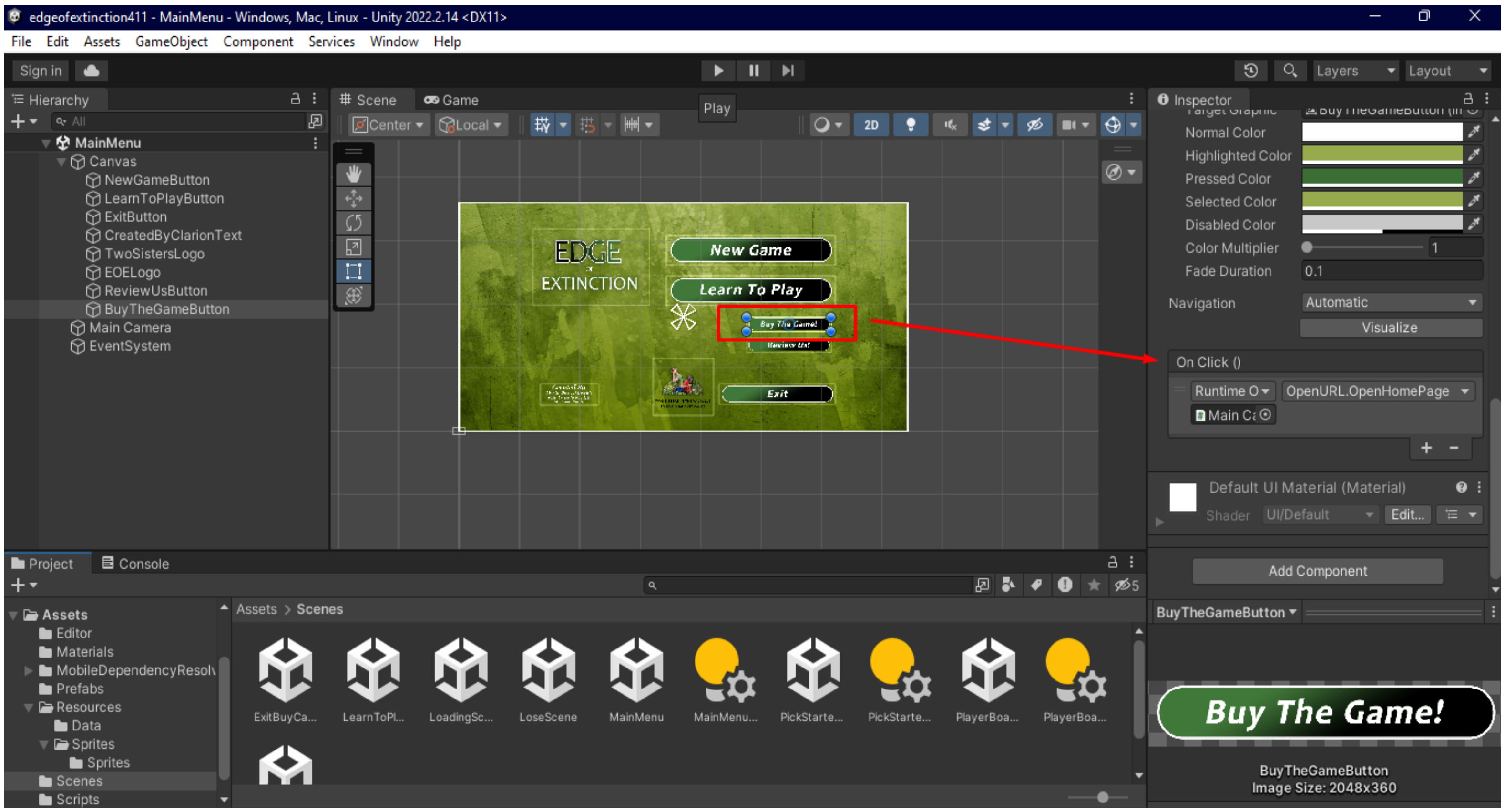
(OpenURL.cs)

This class is what facilitates the ability for players to click buttons throughout the game that will open links to client-requested websites. The user's preferred web-browser will open the link, and redirect them to whatever link is specified by the OnClick() event.

Current links to pages included in the game are as follows:

- **Home Page:** Links to *tswgames.com*
- **Purchasing Decks:** Links to *tswgames.com/products/*insert deck name here**
- **Review Us:** Links to facebook page *facebook.com/tswgamesllc*

Functions were created for each link, which can be called by implementing them into the OnClick() event of a menu button through the inspector tab of said button as shown below:



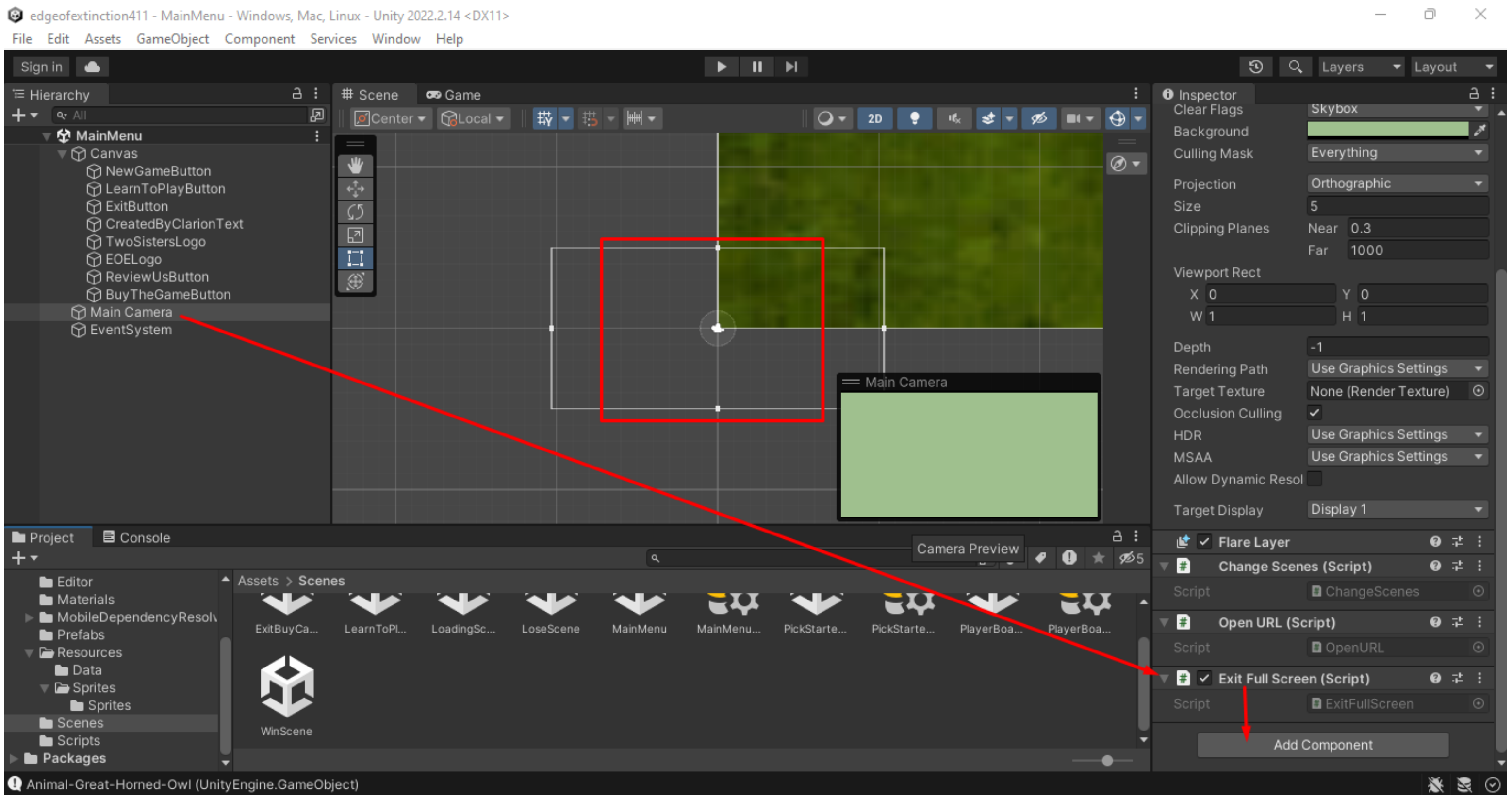
Normal Color
Highlighted Color
Pressed Color
Selected Color
Disabled Color
Color Multiplier 1
Fade Duration 0.1
Navigation Automatic
On Click ()
Runtime O | OpenURL.OpenHomePage
Main C
Default UI Material (Material)
Shader UI/Default
BuyTheGameButton
Image Size: 2048x360

EXITING FULLSCREEN

(ExitFullScreen.cs)

The implementation of the functionality provided by the *ExitFullScreen.cs* class was specifically requested by the client. Said functionality is that when a user presses a button, the game changes to windowed mode and allows players to move it around or exit or minimise out of the game. The chosen key for this was the 'escape'

This effect was implemented throughout all scenes of the project by adding the script component for *ExitFullScreen.cs* to the Main Camera of the game as shown below:



Clear Flags	Skybox
Background	🟩
Culling Mask	Everything
Projection	Orthographic
Size	5
Clipping Planes	Near 0.3
	Far 1000
Viewport Rect	X 0 Y 0
	W 1 H 1
Depth	-1
Rendering Path	Use Graphics Settings
Target Texture	None (Render Texture)
Occlusion Culling	✓
HDR	Use Graphics Settings
MSAA	Use Graphics Settings
Allow Dynamic Resol	
Target Display	Display 1

✓ Flare Layer

Change Scenes (Script) — Script: ChangeScenes

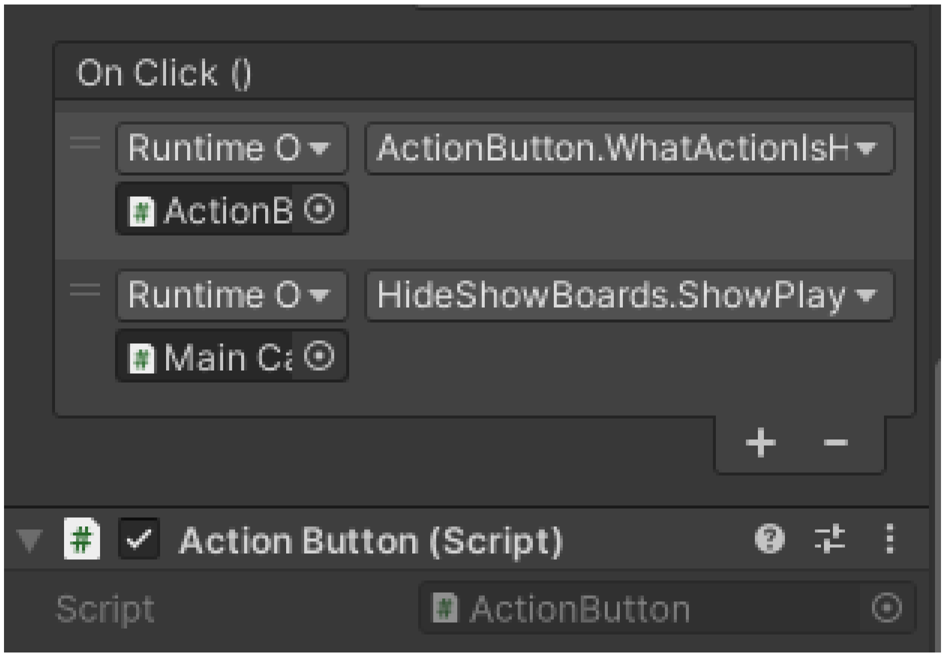
Open URL (Script) — Script: OpenURL

✓ Exit Full Screen (Script) — Script: ExitFullScreen

ACTION BUTTON

(ActionButton.cs)

This script was created by our group in order to facilitate the functionality needed for activating cards that have special effects. When the player presses the action button on a card, a function called *WhatActionIsHappening()* will be called to check the card information and determine what special ability the card holds. It will then begin the process for the effect by using the OnClick() event to execute the aforementioned function within the *ActionButton* script as shown below:


On Click ()
Runtime O | ActionButton.WhatActionIsH
ActionB
Runtime O | HideShowBoards.ShowPlay
Main Ca
Action Button (Script)
Script | ActionButton