REVEREX:DX Menus

COSC 3P99 - Final Technical Report

Supervisor: Prof. Renata Dividino

Nathan Fan
Department of Computer Science
Brock University
St. Catharines, Ontario
nf21dl@brocku.ca | 7267677

Preface—The systems used and discussed within this report were created in the long-term support version of Unity: 2022.3.39f1 and the Visual Studio IDE version 17.12.2 for all code writing. The code and system references displayed are expected to change and may not work for all versions of Unity/Visual Studio.

This report also assumes basic knowledge on Unity objects. As such, it will only give brief descriptions to give a surface level understanding when each is mentioned.

***Abstract*—Systems programming is the basis for all games, from game-to-game interactions to player to game interactions, these system types create the foundation of all projects. In Reverex:DX players are asked to interface with and understand underlying game systems to progress properly through both menus and in-game levels. This report will touch on and expand the first player-to-game system any player is required to interface with the menus.**

I. Introduction

The original goal when starting this project was to incorporate a more comprehensive UI base to teach players game mechanics and concepts throughout the game. But through breaking down the systems that existed beforehand in the demo; REVEREX, it was clear more work had to be put into developing more concise and clear code and a generic template for how menus should be laid out. Because of this, the focus now became to recreate menus from basic scene setups to the code itself.

Assuming a single player game, there are 2 major interactions that take place. The first is a player to game interaction, and the second is the game-to-game interaction. This report will show the application of a player-to-game interaction system found within REVEREX:DX's menus.

The aim of this report is to provide a guideline to follow when coding a menu structure. This includes:

- Audio framework
- Navigation with controllers
- Network support

And then break down the pros and cons for developers and players. By the end of this paper, you should understand specifically how REVEREX menus were set up, the upsides of using a similar template structure and how to expand, use and reuse the ideas presented within.

The remainder of this report is organized as follows, Section II provides a background overview of the generation and creations of menus both generically and within Unity. Section III will provide the methodology and thought behind each decision. Section IV will show the code and layout present within Unity, then analyze the apparent results. Section V will present a post-mortem dive into concerns and contemplations.

II. Background

REVEREX:DX is a two player cooperative game in which players take the role of medical technicians who operate an experimental machine, aptly named the REVEREX. The REVEREX is used to help bring a comatose patient back from a comatose state by entering the deep subconscious mind and bringing them back to reality. One player will play as the Navigator, who will use quick gravity defying platforming to traverse the inside of the mind. The other will play as the Vitalist, who will play short microgames to administer small doses of medication, affecting the experience of the Navigator by speeding them up or down, making them jump higher, or prevent them from having a heart attack. The two players must work together to reach the final area of the mind and save the patient, before the timer

runs out and the experimental elements of the REVEREX machine kill the patient.

Towards the development of REVEREX:DX, several concepts must be developed involving the following:

- Unity Singleton Initialization
- Unity UI & Event handling systems
- Unity additive scene loading
- Object pools

Unity Singletons are a concept derived from the concept of Singletons in Object-Oriented Programming where a piece of code and its public methods/functions/variables are all accessible from any other code that simply references it by name. In Unity, we can take this concept and apply it to Game Objects present within a scene. If a Game Object contains a singleton script, any other object's script will be able to access it.

An example of a singleton call would be:

```
GameManager.Instance.IsPaused = true;
```

Figure 1.1: Example of a singleton variable reference

Where GameManager is the script itself, Instance is the singleton reference, IsPaused is a public accessor for a Boolean variable, and we are assigning its value to true. Thus, from this point on any script that references the GameManagers Instance and gets the IsPaused variable, will see the value as true unless otherwise changed.

Unity has built in UI and Event handling systems pre-defined, but they are severely lacking and often require additional coding to get the behaviors you want.

How this project has used the event system is to handle explicit button transitions when given an input (Pressing down tilt on a controller will navigate to the button directly under the one currently selected), basic required elements such as buttons, dropdowns, images, UI displays (canvases), etc.

Unity Additive scene loading is the general concept of loading a scene on top of a preexisting scene. This has many applications but is one of the foundational building blocks for not performance taxing systems.

An additive scene is a scene which is loaded directly on top of another scene without removing the original scene. When a scene is loaded additively, it exists at the same time and place as the other scene unless specified on the position component.

The menus that are loaded as a single unit are:

- Main Menu
- Waiting Local menu
- Waiting net menu

The menus that are loaded additively are:

- Credits menu
- Options menu

Realistically all scenes are loaded additively on top of the persistent scene, but the menus loaded as a single unit are a placeholder for the main scene.

An object pool is a form of saving elements that circumvents the garbage collector's cost when creating and deleting objects. This is done by reusing preexisting objects, instead of continuously creating and deleting the same object over and over, we can set up a separate script that enables/disables objects in a way that we can reuse the same object for different tasks.

The main way this was used in REVEREX:DX was to supplement the AudioManager where many audio sources are required during play-to-play sound from background music to menu sound-fx, jumping, buffs/debuffs, etc. Therefore, we created an object pool to dynamically reuse audio sources which already existed in the scene.

These concepts allow us to build and populate the menus of REVEREX:DX properly and without having to build these concepts from the ground up.

III. Methodology

Menus within Unity must be arranged within a Canvas Object present within the scene. The Canvas displays UI, which can be overlaid in game on top of a camera or the screen. For our menus we specifically used *overlay on top of the screen* option as the game is working with multiple cameras with different post-processing effects.

For organizational and ease of access purposes we will be formatting Unity Objects in the following order:

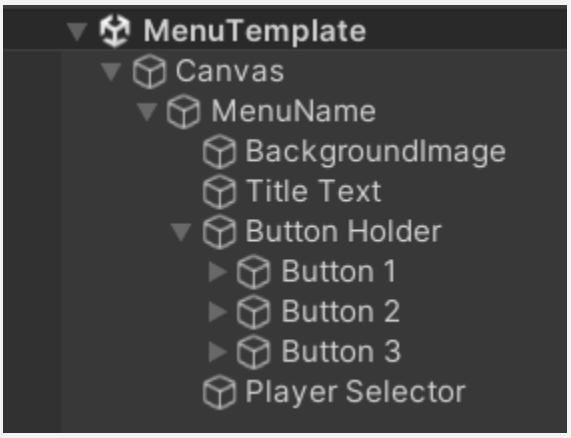

Figure 3.1.1: Unity GameObject Hierarchy

The Canvas object holds all the display and functional settings for the Menus, it would be unintuitive to place more components on the Canvas object itself. To remedy this, a menu object can be created which will hold all components for the menu and also contain the menu controller script. This allows multiple menus to exist under a single Canvas and all elements of a single canvas to exist in one place.

Because Unity renders canvas elements top-down within the object hierarchy, we must place the Background Image object first/at the top as it should be rendered first and behind all the other elements. The title is rendered next as it should not be on-top of any other element other than the Background Image object, the Button Holder object will contain all intractable elements (most of the time being buttons). This is so we can attach a vertical or horizontal Canvas Group element holder which will format the GameObjects contained within it to consistent spacing.

Now that we have a template scene setup we can work from, we will move onto scene dependent on Canvas setup and then the scripting for each.

Scene/Menu layering will be extremely important for this project. There will be 3 main layers, each usually only containing a single scene/menu within. The bottom layer is at the top:

1. Persistent scene (Never unloaded)
2. Main scene (Main scene being switched)
3. Additive scenes (handled by the main scene)
    a. Can have more than one loaded

Most games will contain the following menus:

- Main Menu
- Options/Settings Menu
- Credits Menu
- Pause Menu
- Game Mode/Level Select Menu

Although each of them can use the same framework provided in Figure 3.1.1, they will all require separate scene setups.

For REVEREX:DX we have 6 menus, with the only duplicate being a Local role select menu and Networked role select menu.

For the main menu, the hierarchy is almost identical, with an example Canvas, and since there is no need for this menu to be networked, therefore all button selections and highlights/displays will not need extra scripting.

The same as the menu canvas template, there is a main button holder that contains all the buttons that can be selected within the menu. We have 2 ways to play the game; local and online, therefore the 2 buttons *LocalMainMenuButton* and *OnlineMainMenuButton* will direct you to Local role select scene and Online role select scene respectively.

The Options menu and Credits Menu will be an additive scene load, and the replay button allows you to open a “replay file” that is saved after each run, which loads into a non-intractable replay of that save data.

The Options Menu requires a few more components than accounted for with the template, the main addition being a confirmation popup and the option object holders.

The confirmation popup checks for any changes made within the menu, if there is and the player exits without saving, the popup will trigger and make sure the player intended to exit without saving.

The option object holders contain the elements for each tab within the menu:

1. Audio
2. Game
3. Video

Each tab has different settings and objects that should be displayed when it is selected, therefore we can create separate parent objects that can be toggled on and off.

The Credits Menu does not require intensive behaviors or extensive coding, it only needs to fade between any number of

screens (created by the art team). The canvas setup should look like the following:

Some considerations, there is only one button to select here, therefore there are 2 options:

1. Any key press exits the menu
2. A button is visible that exits the menu

REVEREX:DX uses option two but depending on the situation option one can be more immersive.

The Pause Menu is a menu, but it is handled quite differently from the rest of the systems. It exists on the persistent scene layer and is toggled on and off. This is done because of the frequency a player may need/want to pause, the constant loading/deleting will be extremely taxing on the garbage collector, therefore it is better to slot it within the persistent scene. Additionally, because it interfaces with the pausing of the game, it should always be present during gameplay. The canvas setup within the persistent scene looks like such:

The main consideration has already been touched on with the frequency of use of a pause menu, but additionally this menu must be semi-networked. Due to the nature of the game: competitive speed running parkour, it is expected that if one player pauses, it will pause on both ends.

Additional features were added to support this from an unpause timer; countdown after unpausing before it fully unpauses, but separate controls. Therefore, both players will be able to pause and unpause, but they can't and shouldn't see each other's selections/inputs.

The Local role select menu is the first menu that requires input separation from both players. As everything is in a local game (both players on the same machine) there are no worries about over-complication through network calls.

Because of the dual input and the way input is read and handled in the REVEREX:DX backend, it does not interface well with Unity's multiplayer input event handler. To remedy this, a custom selector was created which is player agnostic, it only needs to be assigned to either player to be able to select within the menu.

The Networked role select menu is the trickiest menu to set up, as it is the only menu that requires networked input visualization and confirmation through steam.

There are 2 stages within the Networked role select menu:

1. Create a lobby
2. Role select

The host-player will have access to the lobby creation menu where they will see a list of steam friends currently online. Both players could access this menu at the same time as no lobby has been created yet, but the moment one player invites another, a lobby is automatically created and assigned to the host-player.

An invitation will be sent to the invited player which can be accepted through steam or visually in REVEREX:DX in the form of a popup.

Only on accept, both players will be bound to a lobby and transition to a role select screen. This screen will display the roles each player is assigned to.

Unfortunately, due to how the framework is set up, a selector similar to local role select will take too long to set up and debug, therefore we opted for preselected roles and a swap button to change roles.

Using this framework players will be able to request a swap or to play the game (where requesting requires both players to press the swap button to go through). This is not an ideal solution or display method for the menu, but unfortunately time was running short and thus corners had to be cut to meet the deadline.

Now that a clearly established framework for each menu has been established, we will touch on coding methods and ideas presented for menu controllers and specific adjustments/contemplations faced with each decision.

First and foremost, audio for all menus will be run through a persistent singleton manager called the AudioManager. Using a singleton, we are able to have a list of saved audio sound effects and an object pool of audio sources from which to play the audio tracks from. This allows us to completely remove how to link to audio sources and play sounds, we can reference the audio manager like such:

```
int audioTrack = 0;
AudioManager.Instance.PlaySoundFXOnline(audio Track);
```

Figure 3.2.1: Example of playing a sound effect

Thus, from any script, as long as during play, if the AudioManager is loaded into an active scene, you will be able to play a sound effect.

As such, audio support for all menu sounds will be handled by the audio manager, with the only additional audio coding we need to do being the play sound call itself.

Within REVEREX:DX's menu scripting, we added audio calls to each button press, so each time a menu was selected or pressed, a menu sound fx would play.

This is about the extent of the audio system utilized by the menus.

The second major consideration is menu navigation for both controllers, keyboard and mouse. There is no need to rewrite what has already been proven to work, therefore unless faced with an unprecedented situation where additional coding is necessary (multiple inputs for selection/networking), we will be using the default button navigation systems built into unity.

There are a couple concepts that unity has discontinued or has not implemented yet, one of them being the last selected button and the default button selected by the system.

The first is a last selected element saved variable built into an event system. It was never specified why it was removed, but we can simply re-add this feature to the menus by adding a variable to our menu controller script:

```
private GameObject lastSelectedObject;
private GameObject firstElementToSelect;
private GameObject lastMenusSelectedObject;
```

Figure 3.2.2: Necessary variables for selection

A last selected variable must exist in the case the player left clicks with a mouse. Due to the difference in how a mouse interacts with menus compared to a controller, the moment a mouse left clicks anywhere on the screen it will immediately deselect the current element. To counter this, we added the lastSelectedObject variable, so in the case where a left click would deselect a menu item, it will default to the lastSelectedObject.

Additionally, to add to menu navigation cohesion, we will continue to set the lastSelectedButton to the current selected GameObject if it is not null.

On script initial load, we would want to initialize and save each variable:

```
private void Start()
{
        lastmenuSelectedButton = EventSystem.current.cur-
        rentSelectedGameObject;
         EventSystem.current.SetSelectedGameObject(first-
        ElementToSelect);
}
```

Figure 3.2.3: Initializing variables for selection

And then run the following in an update loop, this one is run every frame (although it may not need to)

```
private void Update()
{
        If(EventSystem.current.currentSelectedGameObject
        == null) {
                EventSystem.current.SetSelectedGameOb-
                ject(lastSelectedButton);
        }
        else {
                lastSelectedButton = EventSystem.cur-
                rent.currentSelectedGameObject;
        }
}
```

Figure 3.2.3: Update loop implementing default selection

Through this, a controller/keyboard will always have an element to select by default, and a mouse click will never deselect a menu button, making controller/keyboard navigation impossible.

If the menu is additive, we will need another variable which will save the last menu's selected object. This is so when the additive menu is deleted, it will first select the last button interacted with in the previous menu.

```
private void OnDisable()
{
        EventSystem.current.SetSelectedGameObject(lastmen-
        uSelectedButton);
}
```

Figure 3.2.4: On script disable select last menu object

A first element to select is also required, so the menu has a first object to select for navigation purposes when an example can be found with any additive menu, for example we take the main menu as the base menu scene, and load the options menu on top of it additively.

Therefore, through this implementation an omni-navigational menu has been implemented.

Network support is an inevitable concern and consideration when making a multiplayer game. It depends on what point within the menu navigation the controls split, which requires additional scripting, where network support will have to integrate into both basic navigation and menu behaviors.

IV. Results & Analysis

The result of this setup and code structure resulted in a working menu system. Through development and now release, there

were clear upsides and downsides for developers with little impact on players.

For developers, the consistent and easily broken-down setup benefited navigation and ease of access for designers, programmers, and artists (who were implementing assets).

This led to a significant boost of efficiency compared to previous systems where there was little to no hierarchy unless mandated by a canvas object or canvas grouping element.

The way the elements were laid out were easily grouped with similar elements, such as buttons all sharing a common parent object, menu management scripts all existing in the same root objects. This made it easy to comprehend and document.

A clear downside to this system is the heavy dependency load that is put on a single controller script. Because almost all elements are routed through a single controller script that exists on the root object, the dependencies that exist in each menu are specific for only that menu. Only in the case that a menu would be directly copy-pasted (for only 1 case in REVEREX: Options menu vs. In-Game Options menu), that script will be specialized and usable in that single case only. For example, the controller script will often hold all the button behaviors; the main menu credits button will load the credits scene on top of the current main menu scene. This groups up scripting behaviors all in one place, but once again requires the existence of that script within the scene for the button(s) to function.

For players, no sizable benefit could be grasped, but the main point of menus is to be as unobtrusive and straightforward as possible; the best menu is one that no one has comments for. A unique scene hierarchy structure does not heavily change the player experience, then implementing and using a scene template has little to no downsides.

V. Postmortem Concerns & Discussion

The concerns that were brought up at the start of the project were the confusing layout and programming within the menu systems. To rectify this, a system and template were introduced to the process of creating menus, leading to the layout we have within REVEREX:DX.

From this system, the clear limitations and hindrances were the dependencies required for each menu.

It is oftentimes inefficient to code multiple lines of code that do the same thing but exist in different scenes. Therefore, perhaps a more dynamic system which integrates a system that has a button controller script, separate from the menu controller script which interfaces with each other using events rather than direct dependencies.

Another limitation we found that could not be avoided with this project was the time frame/limit presented by the requirements of the project. Because REVEREX:DX was made for a course, the deadline is set at the end of the semester, therefore in the cases where things could have been done better, it was easier to find a time saving replacement or shorthand of the correct method.

Finally, there were no major issues with the template system, it worked for what it was built to do, but it was still not nearly as efficient as it could be. Through more experimentation on removing dependencies, more event usage and a more dynamically adjustable template, an all-encompassing comprehensive template could be created.

VI. Conclusion

This report has proposed a method to menu creation and implementation within Unity. Each game's case will be different on menu requirements and visuals/behaviors and the menus created for REVEREX:DX was no exception. What was created for REVEREX:DX was sufficient for the scale, scope and idea we were trying to present, but it does not apply to all menus in every case.

This study was a very particular case which always involved 2 players playing the game, which is extremely rare for any game from most genres. Therefore, as future work directions, a generic menu setup and flow can and should be explored for multiplayer games.