CMPT 371 – TEAM 3 DESIGN DOCUMENT

Virtual Reality Medical Imaging Software with Luxsonic Technologies Inc.

April 6, 2017

1 Table of Contents

1 Purpose

The purpose of this design document is to present our architecture's description, plan the classes and their interactions. The architecture section will state why we chose to implement our architecture, the advantages and disadvantages of its implementation, and why we chose it over other architectures that we considered. In the Unified-Modeling-Language (UML) section, we will describe what each class should do, what information they will contain, and what information they will send to other classes. It will not contain any code specifications, just how the different classes are connected. Unified-Modeling-Language diagrams are provided with detailed descriptions. This document will describe the classes that will later be implemented, and depict how they will interact with each other.

2 Definitions and Acronyms

Digital Imaging and Communication in Medicine (DICOM): This is the primary file format used to store a series of medical images such as x-rays, ultrasounds, MRIs, and other images used in medicine.

Model-View-Controller (MVC): An architecture design, which implements the idea that classes that interact with the user (view), will send information to the controller that manipulates the information set of classes (model).

Graphical User Interface (GUI): This is the visual depiction of an interface. This interface is one that the user will be able to see, interact with, and affect.

Unified Modeling Language (UML): A general purpose, developmental modeling language that is used as a standard for visualizing the design of a software system.

3 Architecture Description

The architecture that will be used for this project will be independent components. This architecture breaks everything down into functional components that are used to create well-defined communication interfaces that contain different methods, properties, and events. This allows for greater abstraction without having as much concern placed on rigid communication protocols. The various events executed will involve the use of only one or two classes without the need for other classes being involved or affected. This means dependencies for each class are reduced and each class performs a very specific function. Having this as our design choice will provide us a significant advantage moving forward with the project. These advantages are described in the following section.

4 Architecture Justification

Having an independent-components-based architecture is important for a number of reasons. The primary reason for using this architecture is that it is the easiest to implement for the software being

used. Since the Unity environment relies heavily on object creation, everything created in Unity is an object. All objects in Unity have their default components such as Transforms (positions) attached to them. Because these objects are all independent of each other, it would be difficult to use an architecture with a rigid structure. In relation to the Model-View-Controller (MVC), having one class that acts as a controller would be difficult to implement. In addition, the team fully expects to create additional functions and potential classes not currently shown in the UML diagram. Since the technology and hardware is new to many members of the team, changes during implementation will occur. Minimizing these changes are ideal, but having an independent components architecture provides us the flexibility to make changes more easily during implementation.

Another significant advantage of using independent components is reusability. Since several of the classes are designed for a specific task, they can easily be reused in situations that require similar functionality. Independence of classes allows them to be modified, removed, and added with minimal impact to the rest of the system. This reduces their dependencies and coupling. The reduction in dependencies allows us to work on the project more effectively. The team can work separately on different scripts without requiring other scripts to be completed. There will also be less concern that manipulation of different scripts would have unforeseen effects on other scripts.

5 UML Diagram

The following section shows the UML diagram for all classes used in the program. Additional functions and classes will be implemented in future deliverables. The UML will visually show the interaction and relationships between all classes implemented. To reduce confusion, only script objects are shown.

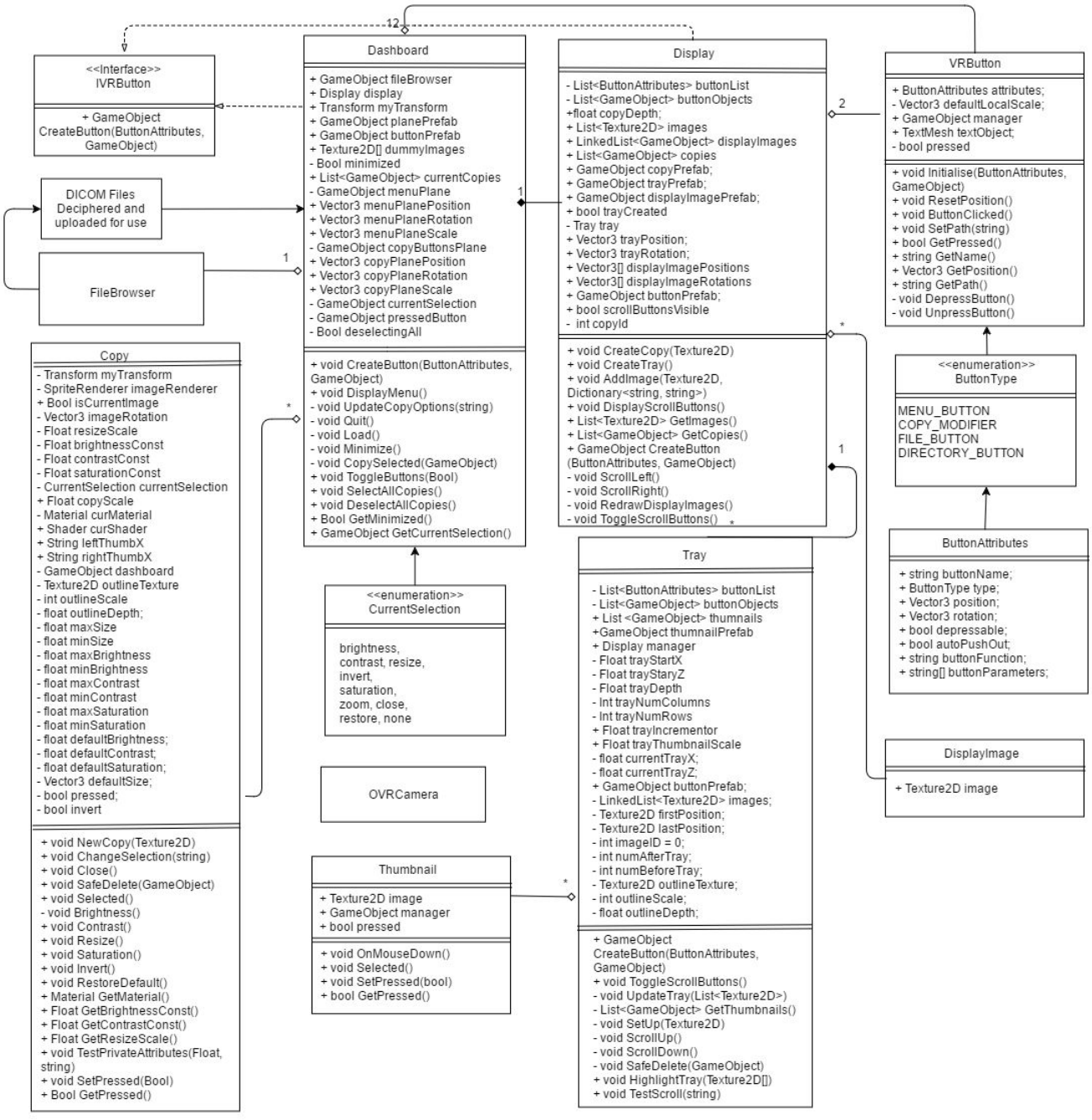

Figure 1. UML Diagram. This diagram represents the relationship, interactions, attributes, and methods involved in each class of our system.

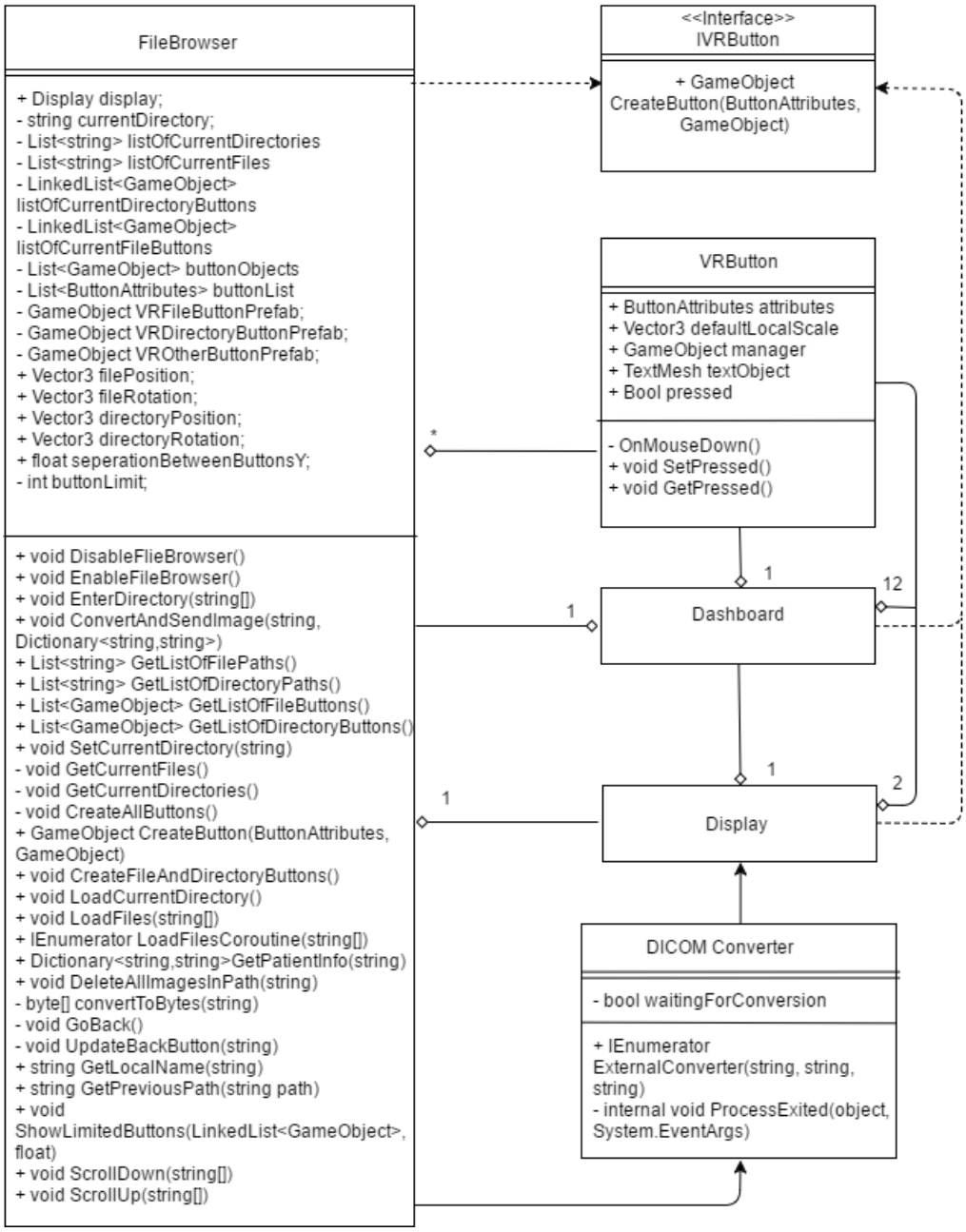

Figure 2. UML Diagram of the FileBrowser. This represents the classes that will be interacted with by the FileBrowser.

6 Class Descriptions

6.1 OVRCamera

The OVRCamera is the camera used by the Oculus Rift to see the virtual world.

6.2 IVRButton

IVRButon is an interface implemented by the Display class. It provides an `CreateButton()` function that is required for buttons to be created on an object. It takes a ButtonAttributes to describe the aspects of the button to be created, and takes in a GameObject as a prefab to make the button out of.

6.3 VRButton

VRButton represents the script associated with an intractable 3D button in the virtual world. In order for a class to create and use this class, it must also implement the IVRButton interface. This class contains a ButtonAttributes variable named `attributes`, a Vector3 called `defaultLocalScale`, a GameObject called `manager`, and a boolean called `pressed`. `attributes` is a reference to all the aspects the button was set to have. `defaultLocalScale` is a reference to what the local scale of the object should be when not pressed. `manager` is a reference to the manager of the button (i.e. dashboard), for message passing purposes. `pressed` holds the state of the button, whether it is pressed or not.

There are eleven functions associated with buttons: `Initialise()`, `ResetPosition()`, `ButtonClicked()`, `GetPressed()`, `GetName()`, `GetPosition()`, `GetPath()`, `SetPath()`, `DepressButton()`, `UnpressButton()`, and `TimedUnpress()`. `Initialise()` take in a ButtonAttributes and a GameObject, and assigns the ButtonAttributes to the `attributes` field, and the GameObject to the `manager` field in the button. `ResetPosition()` sets the position of the object to be the position described in `attributes`, and sets the `defaultLocalScale` to be the current scale of the object. `ButtonClicked()` handles calling of the button's function, depressing and auto unpressing the button button as described in the `attributes`. `GetPressed()` returns the value of `pressed`. `GetName()` returns the name of the button as defined in `attributes`. `GetPosition()` returns the position defined in `attributes`. `GetPath()` checks the type of the button defined in `attributes`, and returns the first value in the button parameters if it's a file button, null otherwise. `SetPath()` sets the first value in the button parameters to passed in string `path`. `DepressButton()` reduces the scale of the button's z axis in half. `UnpressButton()` resets the button's local scale to the `defaultLocalScale`. `TimedUnpress()` invokes `UnpressButton()` after 0.5 seconds.

6.4 Display

The Display script will initially take all the deciphered DICOM images and assemble them into a list of Texture2Ds called `images`. This list will represent the images in the workspace that the user has loaded in. Another list, called `copies`, will store a reference to each Copy created. The last list, `displayImages`, is a doubly linked list that contains the GameObjects which each image is

associated with. These are the images shown in the display along the back wall of the room. The user will be able to scroll through these images. Additionally, the Display will contain an attribute called `tray`, which will be a reference to the Tray.

There are three GameObject attributes contained within the Display class for storing prefab objects. They are `copyPrefab`, `trayPrefab`, and `displayImagePrefab`. The `copyPrefab` will contain a prefab of the GameObject associated with the Copy class, the `trayPrefab` will be a prefab of the Tray class, and the `displayImagePrefab` will contain the prefab used to create the GameObjects in the `displayImages` list. These will be used to instantiate the corresponding objects when needed.

Two Boolean attributes are also present. The `trayCreated` attribute will initially be set to false, and is set to true once an image is added and the tray is created. The next Boolean attribute is `scrollButtonsVisable` which is used to determine if there is enough images in the `displayImages` list for the user to actually scroll through.

A float attribute is also present. The copyDepth attribute dictates the depth at which the copy is placed in front of the user.

For the button implementation, there is a list of ButtonAttributes instances called `buttonList`, holding the parameters for instantiating all buttons in the Dashboard. There is also a list of GameObject instances called `buttonObjects`, holding the instantiated buttons in the Dashboard.

To make sure everything generates in the right positions, there are two Vector3 attributes that are contained within the Display class. They are `trayPosition`, and `trayRotation`. These values determine the position and rotation of the Tray class. To make sure that the images in `displayImages` are in the correct position, we have a Vector3 array called `displayImagePositions`. When the user is scrolling through the display, the display images will move into one of these positions to be displayed. All images that do not fit in the array will be invisible the user until they are scrolled into the array.

The last attribute in the Display class is `buttonPrefab`, which is of type GameObject.

`CreateButton()` is an implementation of the IVRButton interface, made to create a button from an instance of ButtonAttributes and a prefab. When the user selects an image to manipulate a Copy class of the image will be created. This will be done by the `CreateCopy()` function, which takes in a Texture2D as an argument to produce the Copy from. The Display will also contain a `CreateTray()` function which will create the Tray from the list of `images` the Display contains. The `AddImage()` function will add a new Texture2D to the `images` list given to it from the DICOM Decoder. It will create a new GameObject in the `displayImages` list containing that image. If this is the first image added in the program, then it will create the Tray with the `CreateTray()` function. There are also two functions called `GetImages()` and `GetCopies()` which return a list of Texture2Ds and Copy GameObjects within the system (respectively). The function `ToggleScrollButtons()` will Activate/Deactivate both the left and right scroll buttons to browse through the list of images in the `displayImages` list. The `scrollLeft()` and `scrollRight()` functions will move and change the position of the images in the `displayImages` list as to which images are displayed to the user. The last function, `redrawDisplayImages()` will set which GameObjects in the

displayImages list are visible to the user.

6.5 DASHBOARD

The Dashboard is a script that will be responsible for showing the Display and Tray class. It also contains three buttons that user can use to quit the program, load a new DICOM file, or minimize the Dashboard. The Dashboard will have a several attributes. The first is a Transform called myTransform which will store the position of the Dashboard. The Dashboard will also contain attributes which will reference the Display and the GameObject associated with the FileBrowser called display and filebrowser (respectively). There is a list of ButtonAttributes instances called buttonList, holding the parameters for instantiating all buttons in the Dashboard. There is a list of GameObject instances called buttonObjects, holding the instantiated buttons in the Dashboard. A Boolean attribute, minimized, will represent whether or not the Display and Dashboard are visible to the user.

The DisplayMenu() function will be called on start to call CreateButton() for each item defined in buttonList. CreateButton() is an implementation of the IVRButton interface, made to create a button from an instance of ButtonAttributes and a prefab. When selecting the minimize button generated by the DisplayMenu() function, the other buttons (exit, and load) will become invisible to the user through the use of the Minimize() function. The Load button will bring up a new FileBrowser class (once implemented) through the Load() function. This will open the File Browser for the user to load a DICOM file from. Selecting quit will result in the program terminating by way of the Quit() function.

The buttons in the scene communicate with the dashboard via the UdpateCopyOptions() function. This tells the dashboard to send the most recently pressed button option to each of the currently selected copies. There is a CleanUpCopies() function that will remove all copies that have been flagged for deletion in the pendingDeletion list. The DeleteCopy() function will tell the Display to remove the copy from its list of copies.

The function UpdateCurrentSelection() is used to change the current selection of the Display to the button that is passed into it. The CopySelected() function is called when a copy is selected to add or remove that copy from the Display's list of current copies.

The Display contains the functions Quit(), Load(), and Minimize(). These functions are called when the corresponding buttons in the Dashboard are called. Quit() will terminate the application, Load() will start the File Browser to load a new DICOM file into the workspace, and Minimize() will minimize the Dashboard and Display from view. If the Dashboard and Display are already minimized, they will be maximized instead.

ToggleButtons() is used to toggle the visibility of the buttons in the dashboard. This is called as a helper for the Minimize() function. The SelectAllCopies() and DeselectAllCopies() functions are used to select and deselect all copies currently in the workspace, respectively. These functions will search through all copies and call methods to select each one.

The dashboard also contains the Copy manipulation buttons such as contrast and brightness, as well as a list of all currently selected Copies. When one of these buttons are clicked, it will send a message to all selected copies representing which manipulation to perform on them. The Copy will then make the

manipulation selected on the image attached to it.

Finally, the Dashboard contains test hook functions to allow easy unit testing of the class.

6.6 COPY

The Copy script will be attached to the actual objects that will be displayed to the user (the screens with an image). Each Copy will represent a 'copy' of a selected image that the user can clone and manipulate. The Copy class will have a Transform to indicate its position. They will also have a SpriteRenderer attached to them, which will hold the image that the Copy contains. There will also be a Boolean variable called `isCurrentImage`, which will indicate whether or not the image has been selected for manipulation.

The copy class will contain attributes to hold the min and max values for each manipulation. These will allow the manipulations to be constrained between these values to prevent undesired behaviour.

There are several functions that the copy class will contain. The first is `NewCopy()` which serves as a constructor for the class which will take in a Texture2D. Functions `Resize()`, `Contrast()`,`Saturation()`, `Invert()` and `Brightness()` will adjust the corresponding attribute of the Copy. `GetBrightnessConst()`, `GetContrastConst()` and `GetResizeScale()` are all getter functions to get those values from the copies. `RestoreDefault()` will revert the copy back to it's original settings.

6.7 TRAY

The Tray class will hold a list of images and display a thumbnail of each image on its surface. Selecting a thumbnail on the Tray will create a copy of the image to display to the user (the same as selecting an image in the Display class). The Tray class will have a number of attributes. The first attribute will be a list of GameObject prefabs that will represent each thumbnail stored in the tray called `thumbnails`. In order to create this list, the Tray will need a reference to the thumbnail prefab GameObject which will be called `thumbnail`. It also contains a reference will contain three float attributes called `trayStartX`, `trayStartY`, and `trayDepth` which will represent the position of the Tray. It will also contain two integer attributes called `trayNumColumns` and `trayNumRows` which will represent the number of columns and rows of image thumbnails in the Tray. Iit will also have a reference to the Display called `manager`. The tray will have a list of textures that it will use as its images for thumbnails. It will have references to the textures that are in the first and last positions of the tray called `firstPosition` and `lastPosition`, respectively. The tray will have variables `numBeforeTray`, `numAfterTray` to keep track of the number of images that are before and after the tray. The tray has a list of ButtonAttributes instances called `buttonList`, holding the parameters for instantiating all buttons in the Dashboard. There is a list of GameObject instances called `buttonObjects`, holding the instantiated buttons in the Dashboard.

The Tray class contains the functions `UpdateTray()`, `CreateButton()`, `GetThumbnails()`, `Setup()`, `ScrollUp()`, `ScrollDown()`, `SafeDelete()`,and `HighlightTray()`. `UpdateTray()` takes in a Texture2D to add to the tray. It will add the new image to the next available position in the tray if there is room. If there is not room, the number of images after the tray will be incremented. `CreateButton()` is an implementation of the IVRButton interface, made to create a button from an instance of ButtonAttributes and a prefab. These buttons will

be initialized and used to scroll through the images in the tray. `GetThumbnails()` will return the list of thumbnails currently in the tray. `Setup()` must be called the first time the tray is used. It will initialize the tray and add the first image to the first available position. `ScrollUp()` and `ScrollDown()` are called by the scrolling buttons to scroll the images in the tray up or down. `HighlightTray()` is used by the Display class to highlight the images in the tray that are currently being shown in the Display.

The Tray class also contains the function `TestScroll()` that is used as a test hook by unit tests for the Tray class. This function performs tests on the scrolling functionality to ensure that the thumbnails are scrolled in the correct order and displayed properly after scrolling.

6.8 THUMBNAIL

The thumbnail class represents the images that will be present in the tray. It contains the attributes: Texture2D called `image`, a GameObject called `manager`, and a boolean `pressed`. The `image` will contain the image we want the thumbnail to display. The `manager` will be a reference to the Tray class. `OnMouseDown()` activates when clicked and `Selected()` will activate when the thumbnail is touched. `SetPressed()` is used by the TouchAndGrab class to set the Thumbnail when is has been interacted with.

6.9 FILEBROWSER

The FileBrowser will be a class that the user can call upon to search for DICOM files to load into the system. We want the FileBrowser to be at a fixed position to the user, so it will have to adjust to the Oculus Camera's position. In order to do that, it will have to have a reference to the camera's Transform position with the `cameraPosition` attribute. It will also have a reference to the Display class through the `display` attribute. The file browser will have the string attribute `currentDirectory` to store the path of the current directory. The attributes `listofCurrentDirectories` and `listOfCurrentFiles` will store a list of pathnames to the directories that are in the current directory along with the files that are present. It will also contain two lists of GameObjects called `listOfCurrentFileButtons` and `listOfCurrentDirectoryButtons`. These will store the file and directory buttons created. In order to create the buttons needed in the FileBrowser, the class will need a reference to three different GameObject prefabs. These are represented by the attributes `VRFileButtonPrefab`, `VRDirectoryButtonPrefab`, and `VROtherButtonPrefab`. In order to store the attributes for each button and manipulate them in the inspector, there is a list of ButtonAttributes called `buttonList`. There will also be several Vector3 attributes that will hold positions and rotations of the various buttons created. These include: `filePosition, fileRotation, directoryPosition, and directoryRotation`. Since the back and cancel buttons are static, there will be an attribute referencing both called `backButton` and `cancelButton`. The last two attributes included in the FileBrowser will be a float called `separationBetweenButtons` which represents the space between the directory and file buttons and an integer called `buttonLimit` which indicates the limit on how many buttons are allowed to be seen (since a directory could have numerous files).

This class will include several functions for generating and navigating the simulated directories. The `GetListOfFilePaths()` and `GetListOfDirectoryPaths()` will return the

listOfCurrentFiles and listOfCurrentDirectories attributes. The functions GetDirectoryButtons() and GetFileButtons() will return the listOfCurrentFileButtons and listOfCurrentDirectoryButtons attributes. Directory buttons will allow the user to enter the directory corresponding to that button.

The file buttons will send the path of the selected file to the ConvertAndSendImage(). The ConvertAndSendImage() function will get the file from the path and convert it to a Texture2D and send it to the Display for use. It will also take in a Dictionary as an argument, which represents the patient info that is associated with the image.

To keep up with good programming practices, we have setter and getter functions to retrieve different attributes. These include GetListOfFilePaths(), GetListOfOfDirectoryPath(), GetListOfFileButtons, GetListOfDirectoryButtons(), GetCurrentFiles(), GetCurrentDirectories(), and SetCurrentDirectory(string). These functions essentially perform the task associated with their names.

CreateButton() is an implementation of the IVRButton interface, made to create a button from an instance of ButtonAttributes and a prefab. It is used by the CreateFileAndDirectories() function to generate all the required file and directory buttons.

Several functions in the filebrowser were used to obtain and load the DICOM files. The first is LoadCurrentDirectory() which will load all files in the current directory to be parsed by the DICOMConverter. LoadFiles() will load all specified files into the decoder. The LoadFilesCoroutine() will send each file to be loaded on a new coroutine to deal with latency issues. The last DICOM function is the GetPatientInfo() which uses the DICOM dictionary to retrieve specific information from the path of a dicom image given to it.

The EnableFileBrowser() and DisableFileBrowser() functions will either allow the FileBrowser to be visible or invisible to the user. The EnterDirectory() function will be called to enter and change the currentDirectory to the one corresponding to the DirectoryButton selected by the user. The functions GetCurrentDirectories() and GetCurrentFiles() will search the current directory for file and directory paths and set the listofCurrentDirectories and listOfCurrentFiles to what it finds. The function GoBack() will update the current directory the user is in to the previous directory. UpdateBackButton() will update the path stored in the back button. GetLocalName() will take in a string representing a path and then return the word after the last backslash. GetPreviousPath() does something similar, but returns everything except the last word of the path.

The functions ScrollUp() and ScrollDown() will scroll the position of either the listOfCurrentFileButtons orlistOfCurrentDirectoryButtons up or down. Which buttons are currently visable to the user is determined by the ShowLimitedButtons() function. This function will check if the position of the buttons is above a value calculated by the separationBetweenButtons and buttonLimit attributes.

6.10 DICOMConverter

The DICOMConverter class is used solely to extract the images from DICOM files. It calls an external DICOM Converter program made to circumvent restrictions within the Unity Engine. The class contains

one one attribute, a boolean called `waitingForConversion`.

There are two functions in the class: `ExternalConverter()`, and `ProcessExited()`. `ExternalConverter()` is a Coroutine function that takes in 3 string arguments which are passed directly to the external program: `mode`, `targetPath`, and `destinationPath`. The Function starts the external program on a separate process, sets `waitingForConversion` to true, and waits until `waitingForConversion` is reset to false. `ProcessExited()` is an event listener for the process, and is called when the external program finishes execution. It changes `waitingForConversion` to false when called.