Virtual Reality Interaction Library

Installation and In-Depth Description of VRIL

Authors:
Andreas Roither, Christoph Pargfrieder

May 5, 2020

Contents

1 Installation

VRIL can be integrated by either importing the package from the Asset Store or by manually downloading the latest version from Github VRIL - Github and placing it in the Asset folder in your project.

2 Quick Start

Example scenes
To start as soon as possible please look at the included example scenes under *VRIL/ExampleScenes*. In this folder scenes with the implemented techniques have been set up. There are two different scene examples: Scenes for keyboard and mouse, and SteamVR. To use the SteamVR example scenes please install the SteamVR package from the Unity Asset Store first and then generate default actions via the SteamVR Input window.

Custom scenes
To set up a new scene by yourself, please attach the *VRIL_Manager* script to any game object you want in the scene. Afterwards assign game objects as controllers. You can assign interaction or navigation techniques to these controllers, assigning multiple techniques is possible. Please attach any technique that you want to use in your scene to a game object in the scene and add an interaction or navigation technique script to a specific controller game object at the manager. A technique script has to be derived from the *VRIL_InteractionTechniqueBase* or *VRIL_NavigationTechniqueBase*, depending on what kind of technique it is. Techniques have button mapping available to them, which has to be set if a technique has been newly added to the scene. Lastly, if an object should be interactable, the *VRIL_Interactable* script has to be attached to the game object. To enable navigation with selection-based techniques, the *VRIL_Navigable* has to be attached to the game object (e.g. attach it to the ground plane to enable your floor for navigation). In **Figure 1** below you can see an example how registered controllers and their techniques at the manager look like in the Unity Inspector. In **Figure 2** the example technique iSith shows how a technique can be set up.

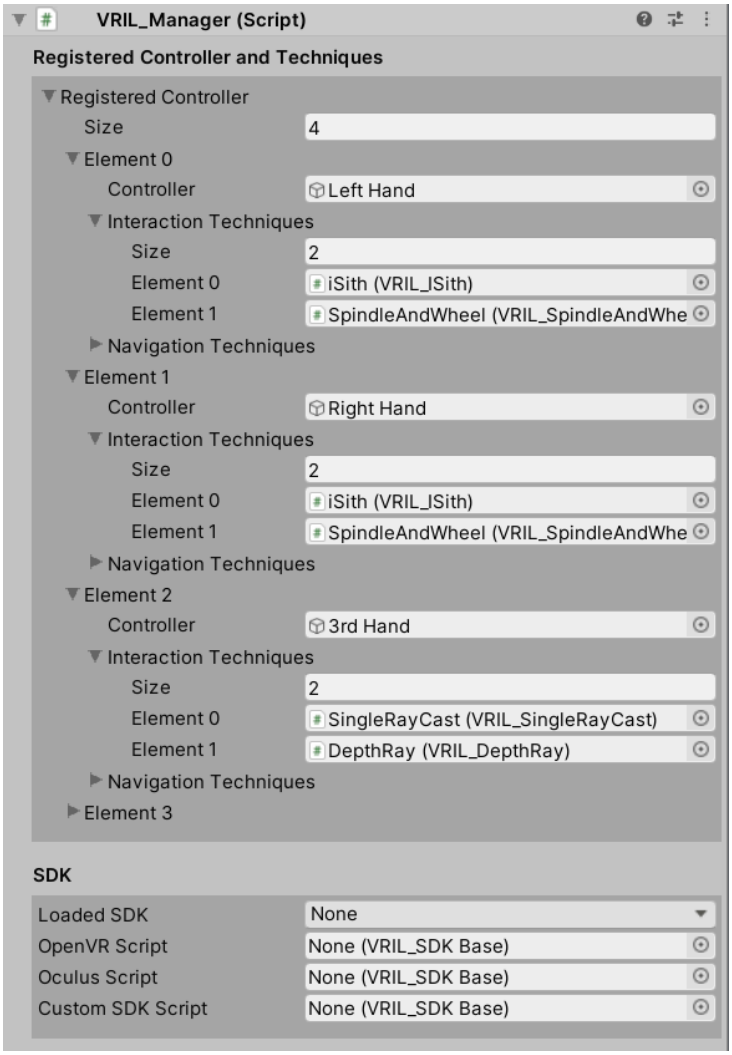

Figure 1: VRIL Manager

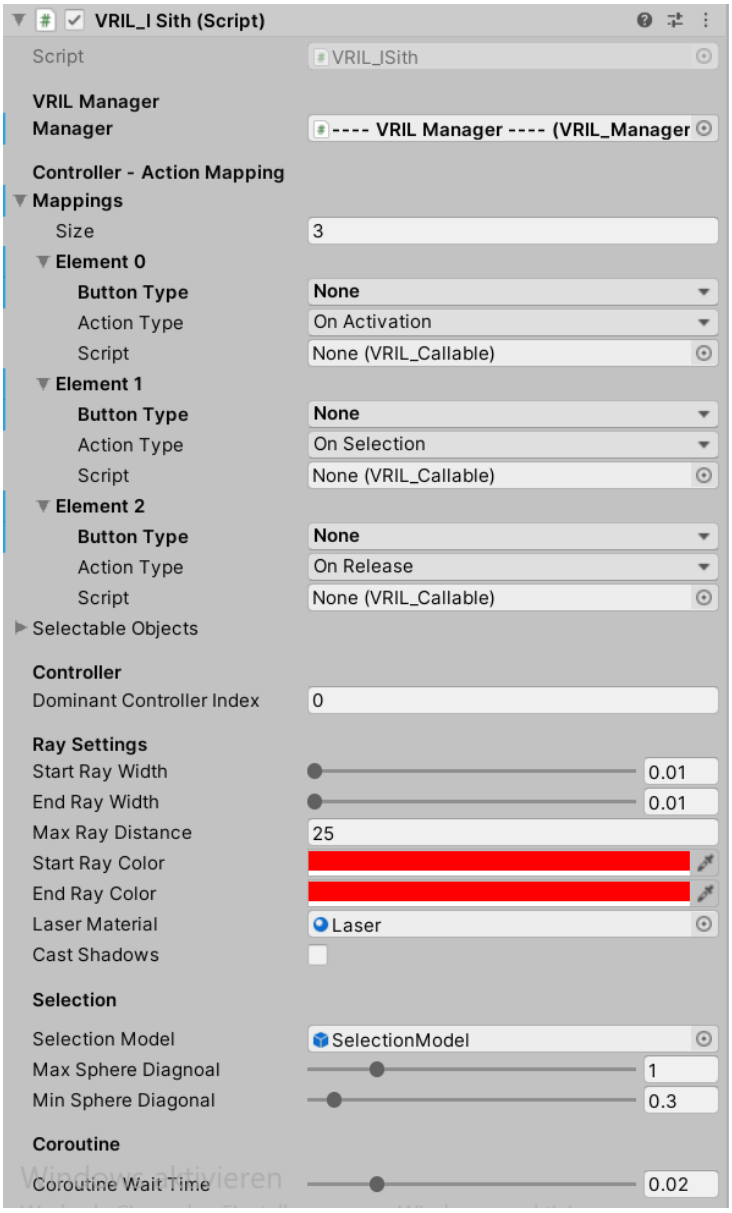

Figure 2: iSith Technique

3 In-Depth

For a more detailed explanation how the components work and UML diagrams please read this chapter.

3.1 Concept Design

Before you can use the library by your own you have to understand how the library works first. **Figure 3** shows the conceptual design of the library.

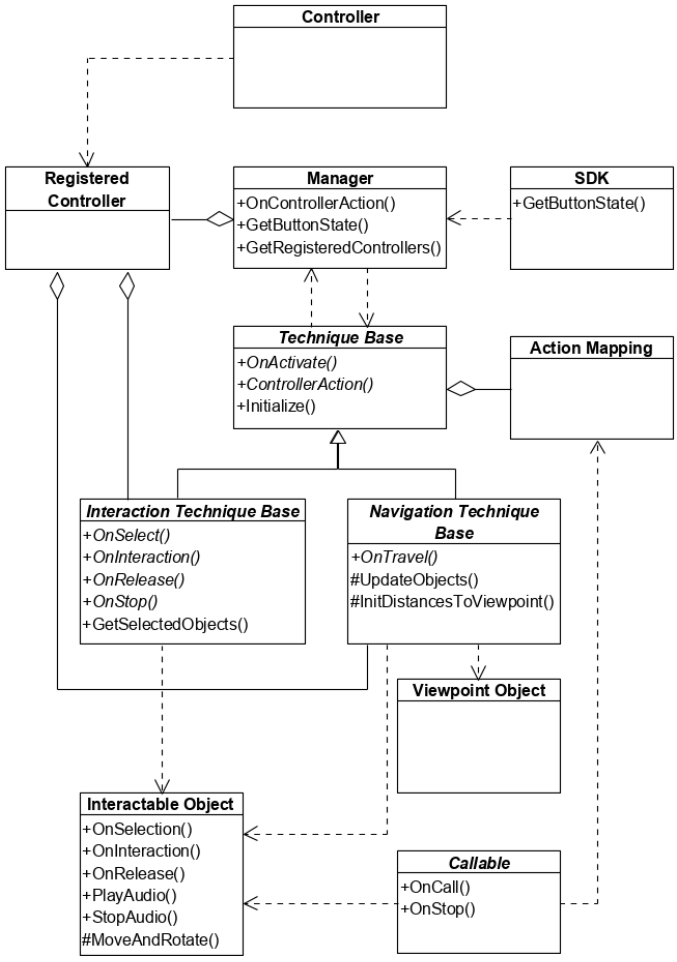

Figure 3: Concept Design

3.2 Architecture

The manager has the lists of registered interaction and navigation techniques for each controller. A controller is mapped to a specific technique which allows for diverse use of multiple controllers with multiple techniques. Additionally, it also contains a list of SDKs with the possibility to request a button state of the registered controller if additional input for a technique is ever needed. A registered controller can be any desired game object, the registered technique for the controller has to be derived from the interaction or navigation technique base. A registered controller does no literally have to be a controller, it is just the game object that is used by the technique to accomplish its task. For example, the camera can be used for a gaze-based system as a controller as well.

The technique base has a list of registered controllers that is provided by the manager since many interaction and navigation techniques depend on the position and rotation of the controller. Additionally, the base class has a list of action mappings. This allows the user to map a specific controller action to one of the available functions of the base class or a custom script.

The interactable object has a variety of options should it be selected, released or manipulated. These options include playing audio, using a custom script or moving to a desired position like the controller position.

The viewpoint object is used by the navigation techniques for travel. Usually the *[CameraRig]* object is attached to navigation techniques in the Unity editor.

3.3 Library Components

3.3.1 Manager

The *VRIL_Manager* (UML **Figure 5**) acts as an intermediate between any SDK or custom script used and the technique. As the techniques should be reusable even if the hardware changes, the manager offers the possibility to use custom SDK scripts. Custom SDK scripts have to implement the interface *VRIL_SDKBase* which provides a function to return the current state of a button which can be requested from an interaction technique.

All used controllers or rather the representing game objects register at the manager. To further specify which controller uses which navigation or interaction technique, a list of said techniques can be added to each controller. One technique can be referenced multiple times and will work for multiple controllers (the mappings have to be adjusted however). All techniques will retrieve the registered controllers at the beginning of their lifetime and save them. Since the manager has a reference to all registered techniques, controller actions can be relayed by the manager as well. The function *OnControllerInteraction* can be used to relay the pressed button as well as information in which form the interaction happened. Possible options are:

- None
- Pressed
- NearTouch
- Touched
- PressedOnce
- TouchedOnce
- Released

Available options for a controller interaction include:

- None
- Button1
- Button2
- ButtonStart
- Grip
- Trigger
- Touchpad

If a controller action happened (function OnControllerAction was called) the manager then proceeds to pass on the specified action to every technique script attached to the specified controller.

The navigation or interaction technique can then react to the input. Techniques that are not registered at the controller will not get notified. The manager also provides a function to check a button state for a specific controller as mentioned above. This function, however, is not necessarily needed by every technique but can be used to check for additional input. Lastly, the manager implements a function that returns a list of registered controllers for a given technique since most techniques need some form of position or rotation to work with (for example iSith or ray-casting) in order to function properly.

As more and more properties are added to a script the user interface will get cluttered soon. In order to provide better usability, the manager has a custom interface with sections and boxes to ease the use of the manager as seen in **Figure 5**. The custom interface may not play a big role for the manager as the manager does not have that many properties, but the custom interface will be crucial for *VRIL_Interactable* objects.

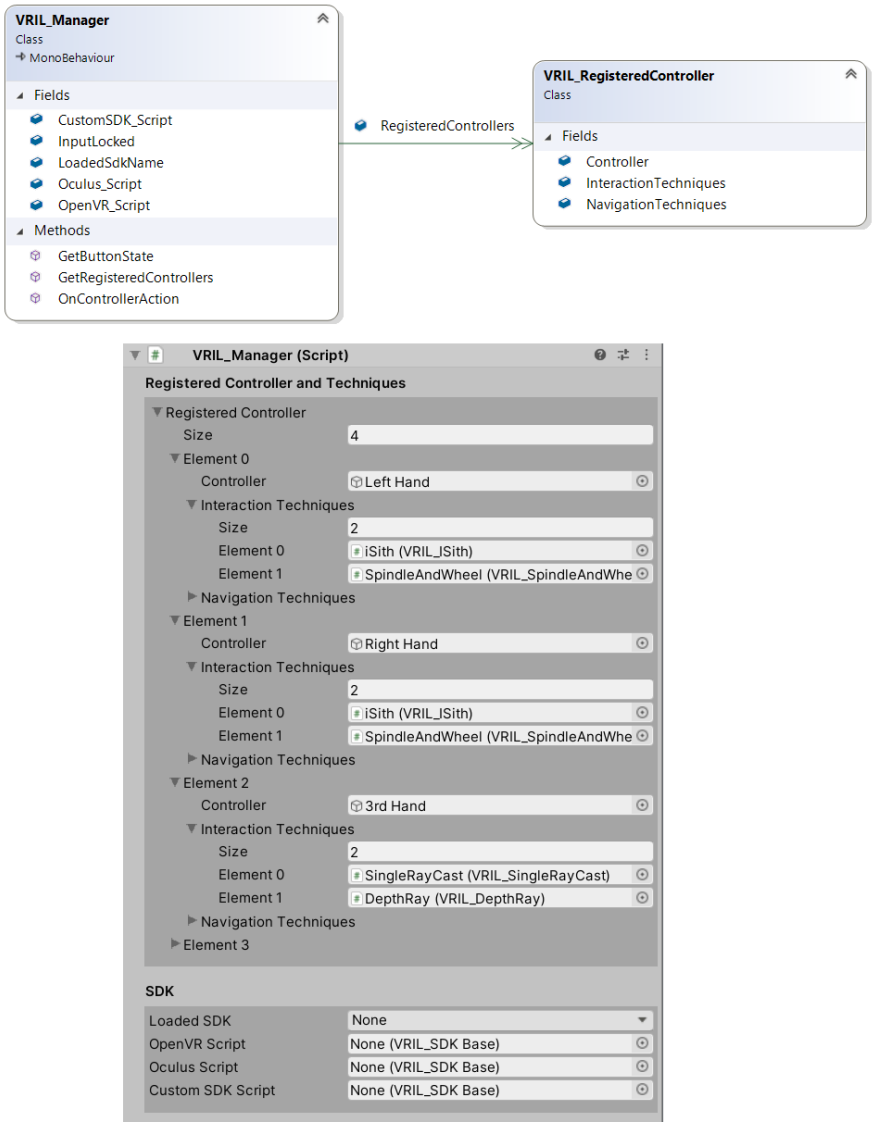

Figure 4: VRIL Manager UML Diagram

3.3.2 Technique Base

This class (UML **Figure 23**) is the common parent class for all VRIL techniques. It provides the reference to the manager instance as well as the registered controllers that will be requested from the manager when the *Initialize* function is called. Controllers that are requested from the manager

are returned in the order that they are registered. Both interaction technique base and navigation technique base are derived from this common abstract class.

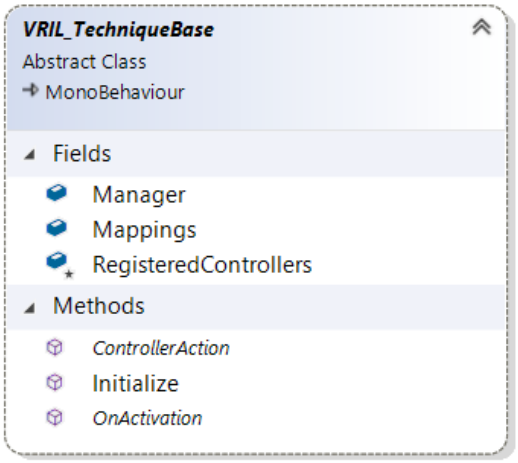

Figure 5: VRIL Technique Base Class

3.3.3 Interaction Technique Base

The interaction technique base (UML **Figure 23**) is the base class for all other interaction techniques and implements basic functionality to communicate with the registered manager and interactable objects. It is derived from the common technique class. All interaction techniques that derive from this class should call the *Initialize* function at the start. As already mentioned, the order of the controllers are saved in the order that they are registered by the abstract technique base class. This has to be kept in mind when working with the *DominantControllerIndex* property that specifies which controller is the primarily used controller (left or right handed user).

Any controller action relayed from the manager can be mapped to the functions of the base class or even a custom script.
These functions include:

- OnActivation
  is used to start or enable an interaction technique and should not be used to initialize a technique

- OnSelection
  is used to select interactable objects that are available for selection
- OnInteraction
  is used to interact with an object
- OnRelease
  is used to release all selected objects
- OnStop
  is used to stop or halt an interaction technique

The mapping is implemented by another class, the *ActionMapping* class. Mappings are saved in a list and are automatically handled by the base class. Mapping an action twice will not result in an action to be ignored which creates the possibility to call a base class function and a custom script at the same time. This allows for easier development where the base functionality of the desired mapping can be left to the base class while still adding additional options in form of the custom script. The custom script has to implement the *VRIL_Callable* interface in order for the base class to invoke the *OnCall* function.

The interaction technique base also provides basic functionality for notifying an interactable object upon selection, release or interaction. Any of these functions can be overwritten by a derived class if different actions are intended for an interaction technique. Interactable objects are game objects that have the *VRIL_Interactable* script attached to them in the Unity editor. All interactable objects can decide, based on their settings, what action should be taken for either of the previously mentioned cases should it be notified.

3.3.4 ActionMapping

The action mapping class (UML **Figure 6**) is used by the base classes of interaction and navigation to map a specific controller action to a function of the technique base or a custom script. If a custom script is used the function *OnCall* will be invoked. *ActionType* includes the following options:

- None

- OnActivation
- OnSelection
- OnInteraction
- OnRelease
- OnStop
- OnTravel
- CustomScript

ButtonType includes the following options:

- None
- Button1
- Button2
- ButtonStart
- Grip
- Trigger
- Touchpad

Mapping the buttons in the right way is entirely up to the user. Since the manager just relays any controller action to the registered techniques without any checks if the right button was pressed, changing button assignments can be done at either the techniques or at the SDK that registers controller actions and sends it to the manager.

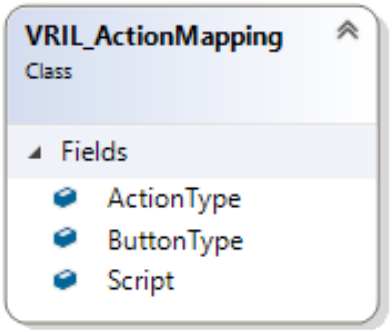

Figure 6: VRIL_ActionMapping UML diagram

3.3.5 Callable

All custom scripts that can be used by a technique or an interactable object must implement the *VRIL_Callable* interface (UML **Figure 7**). The *OnCall* method will be the function that will be invoked if the custom script should be executed. Additionally, a *OnStop* method is provided if the custom script needs to cancel ongoing processes. The *OnStop* method though will only ever be called if it's mapped correctly at the technique. Calling the method in any other function is possible as well.

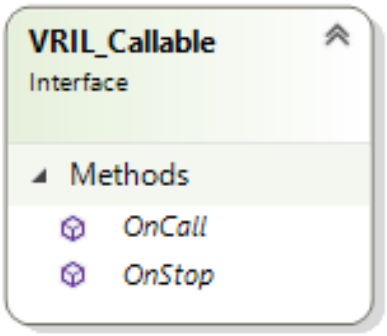

Figure 7: VRIL_Callable UML diagram

3.3.6 Interactable Object

The interactable object class (UML **Figure 24**) provides an extensive list of options for every category since an interactable object should mostly manage

itself. Each category has it's own audio source, audio clip or custom script that will be played or started if the object is notified from the interacting technique. Additionally, a general audio source is provided as well but has to be selected explicitly in order for the audio source to be used. If this option is selected only one audio source will be used by the object in case any music should be played. Since an interactable object is highly customizable with many different options a new user interface in Unity is needed in order to provide better usability. The interface can be seen in **Figure** 8. All these options mentioned above belong to one of the following categories:

Selection
The selection category provides options if the object is selectable, uses selection feedback, is swappable between controllers or if the object should attach itself to a specific location. Moreover, the speed at which the object moves to its final location can be adjusted as well.

Interaction
The interaction category provides the same options as the selection category but with additional options specific to interactions. These additional options include the ability to select if the object's position or rotation can be changed. The object itself does not check at runtime if it has been moved as these options should be used by the interaction technique that manipulates the object.

Release
The release category provides the same options as the selection category. If an interactable object should be selected, the script provides functions that move and rotate the object to the desired position (new position or initial position) and play or stop the music if any audio source or audio clip is attached and the feedback option is set.

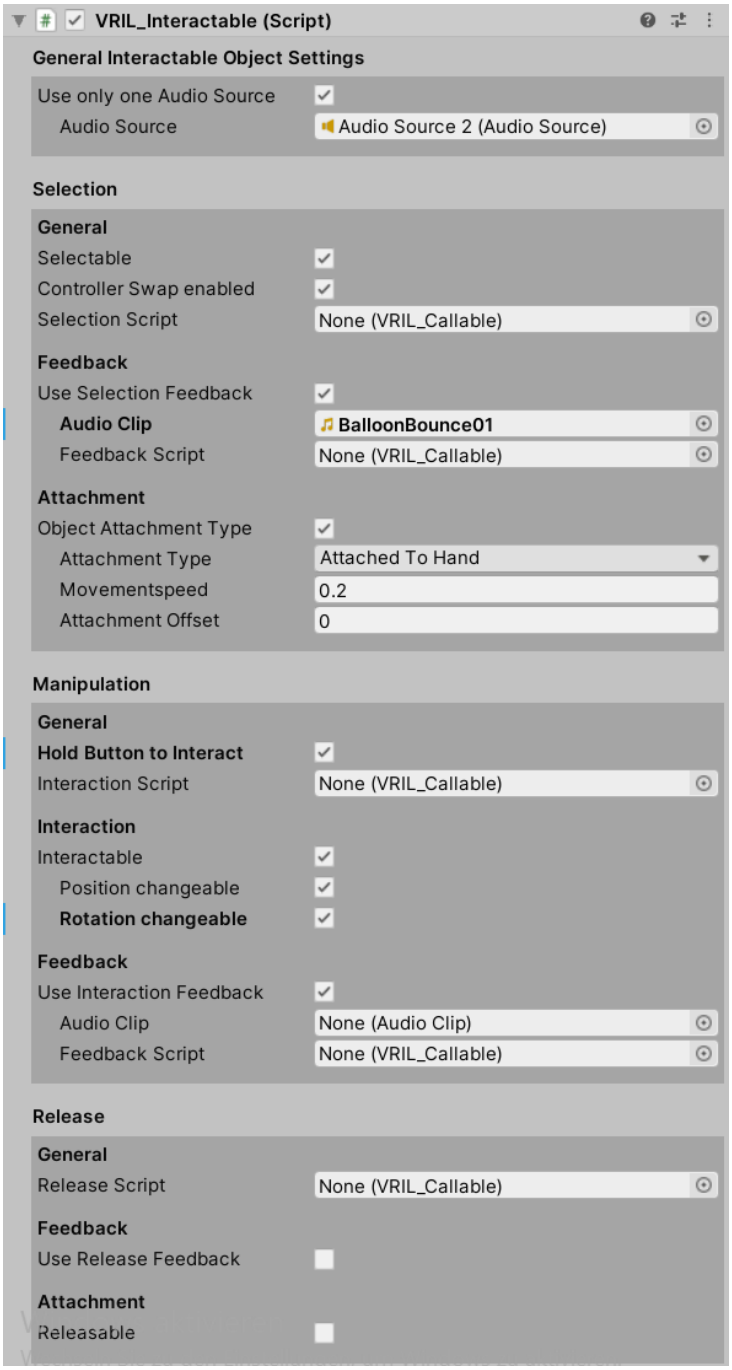

Figure 8: VRIL_Interactable object Unity interface

3.3.7 Navigation Technique Base

The abstract class *VRIL_NavigationTechniqueBase* (**Figure 25** located in the image appendix shows the UML diagram) is the foundation of navigation techniques in the library and derives from the previously described technique base class. This class provides the functionality and properties that are used by all navigation techniques such as the viewpoint object, the desired target position or the implementation of the function *ControllerAction*. The latter decides whether a mapping to an action type is set for the given input when it is called from the manager class. This works analogous to interaction techniques where the function checks the list of action mappings (class *ActionMapping*) whether it contains an entry for the given button type and what kind of action should be triggered then.

The action types *OnActivatation* and *OnTravel* result in calls of the appropriate technique functions. In case no mapping is defined for *OnTravel*, the travel task is triggered by releasing the button that is set for *OnActivatation*. The action type *CustomScript* invokes the *OnCall* function of the custom script which is attached to the mapping entry in the Unity editor (as described these scripts have to implement the *VRIL_Callable* interface).

Sometimes it is not sufficient to transfer only the viewpoint object to a target position. This is the case when the user is manipulating an object. Here, it is expected that such objects are also transferred along with the viewpoint. This class provides a function that collects the distances of selected objects to the viewpoint in a dictionary. These distances are then used in the function *TransferSelectedObjects* which places all selected objects at the same position relative to the viewpoint after the travel task has been completed. By default, these objects are also moved through other objects without collision during travel. This behaviour can be changed by using ridibody components (object behaves according to Unity's physics engine) or attaching custom scripts to the interactable component, which for example invoke the release function.

However, moving controllers separately is usually not necessary. In many VR environments, there is the notion of a "container" in which the camera and controller objects are held. The SteamVR plugin[1] provides such a container as a prefab called "Camera Rig". This object is usually set as the

[1] https://assetstore.unity.com/packages/tools/integration/steamvr-plugin-32647

viewpoint for a navigation technique in the Unity editor. When travel is performed, it is this container that moves through the VE.

A navigation technique can also use audio clips. For this purpose, the functions *PlayAudio* and *StopAudio* provide the possibility to start and stop an audio clip. If any audio source and audio clip is attached to the script, the clip will be played or stopped by invoking these functions. Usually, navigation techniques use this feature for the travel task (e.g. playing a short clip when teleporting).

The class *VRIL_NavigationTechniqueBase* overrides the *Initialize* function in order to set some additional properties on application startup such as the initial distance from viewpoint to ground. This is required by techniques that are restricted to only allow a user to navigate on a surface in a VE (for example a teleportation based technique). The technique always adds this distance to the target position in order to keep the distance to ground while travelling.

3.3.8 Navigable

The *VRIL_Navigable* class is only used as an identifier to mark an object available for the target position selection. This is necessary for selection based travel metaphors such as the point and teleport technique [BRKD16]. Here the selection of the target position is implemented via pointing by a ray from the controller to a position on an object surface. In order to distinguish between game objects where a user can travel to and any obstacles, the first must have the script *VRIL_Navigable* attached to them in the Unity editor. This is analogous to interaction where interactable objects have the component *VRIL_Interactable* attached. However, the navigable component does not provide its own functionality. **Figure 9** shows an example of a basic plane object which allows selecting a target position on its surface.

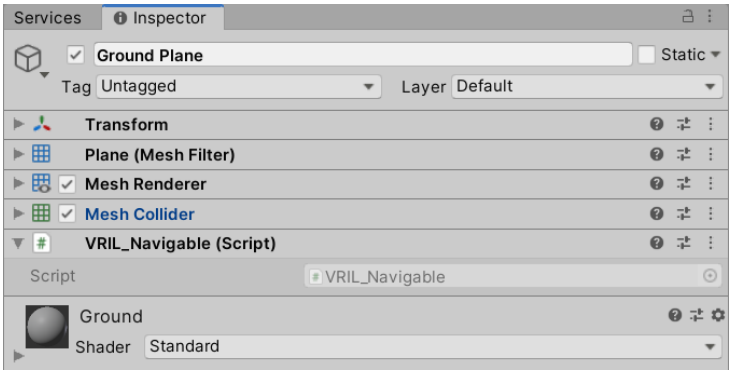

Figure 9: An example of a navigable ground plane in Unity

3.4 Example Integration

To show how the interaction library can be used, two example scripts are implemented. Scenes for each example have been set up in Unity that utilize said example scripts.

3.4.1 Keyboard and Mouse

The *VRIL_Keyboard* script allows the user to move, rotate, select and activate a technique. This script simply remaps any input from the keyboard or mouse and notifies the *VRIL_Manger*. Additionally, the script also shows which controller is currently selected which can be seen in **Figure 10** on the top left side.

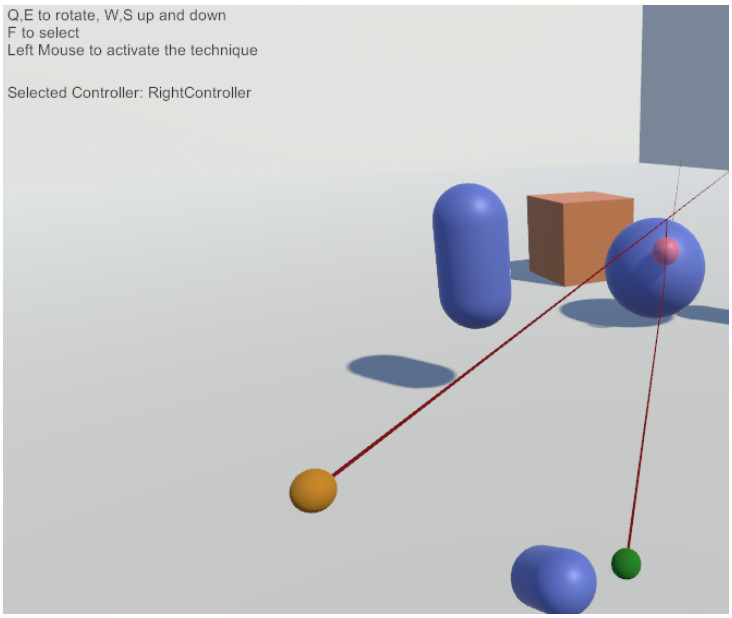

Figure 10: VRIL_Keyboard UI

3.4.2 SteamVR

The SteamVR[2] example script utilizes the default actions that are generated on start. Actions were introduced with SteamVR 2.0 and allow developers to subscribe to an action instead of checking input from the controller manually. To use said actions or create new ones, the "SteamVR Input" window in Unity offers a variety of options (package has to be installed first, as well as the SteamVR software). If these actions have been generated, they can be bound to a specific controller input action like a button or a touchpad. After actions are bound and generated, developers can subscribe or attach a new listener to an action. The example script utilizes 4 of the default actions, that are auto-generated when opening the "SteamVR Input" for the first time, and registers listeners to each one. After any input from the controller was detected, the *VRIL_Manager* will be notified. In **Figure 11**, an example for the *iSith* technique with SteamVR can be seen.

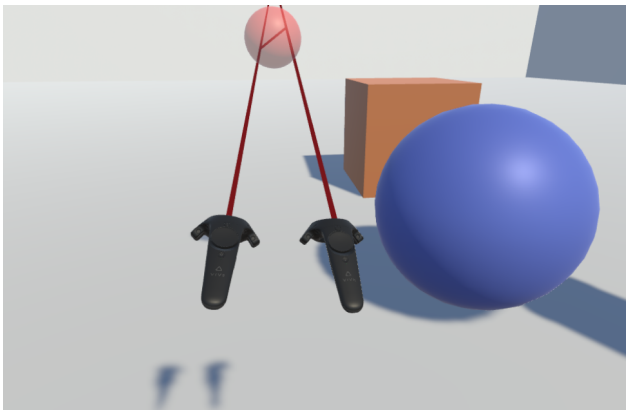

Figure 11: VRIL_SteamVR example

[2]https://steamcommunity.com/steamvr

3.5 Interaction Techniques

These are the interaction techniques that are implemented in the library. An example image illustrates the use of the technique described.

3.5.1 Ray-Cast

The Ray-Casting technique (earliest version mentioned by Bolt [Bol80], later again by Bowman and Hodges [BH97]) is an interaction technique that utilizes a single controller with a line to show the user what objects can be selected. Anything the line touches and is selectable can be selected. The ray-casting technique (UML **Figure 12**) is implemented by using a single line renderer[3]. By using a special attribute[4] from the Unity scripting API, a line renderer will automatically be attached to the parent game object of the script if no line renderer is present:

```
[RequireComponent(typeof(LineRenderer))]
```

The line renderer is used to create a single line moving forward from the first registered controller with a customizable maximum distance. Line width, starting and ending width as well as the color of the line can be adjusted in the automatically created line renderer script. Since the ray should be able to select interactable objects (objects that have the *VRIL_Interactable* component attached), a *Physics.Raycast*[5] from the Unity scripting API is used. The ray is provided with a point of origin, direction and a maximum length and any objects that collide with said ray will be selected.

An example of the Ray-Cast technique can be seen in **Figure 13**.

[3]https://docs.unity3d.com/ScriptReference/LineRenderer.html
[4]https://docs.unity3d.com/ScriptReference/RequireComponent.html
[5]https://docs.unity3d.com/ScriptReference/Physics.Raycast.html

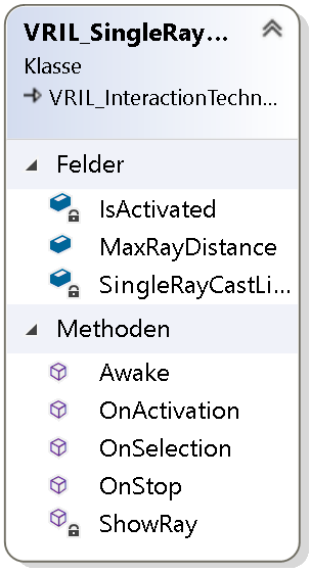

Figure 12: VRIL_SingleRayCast UML diagram

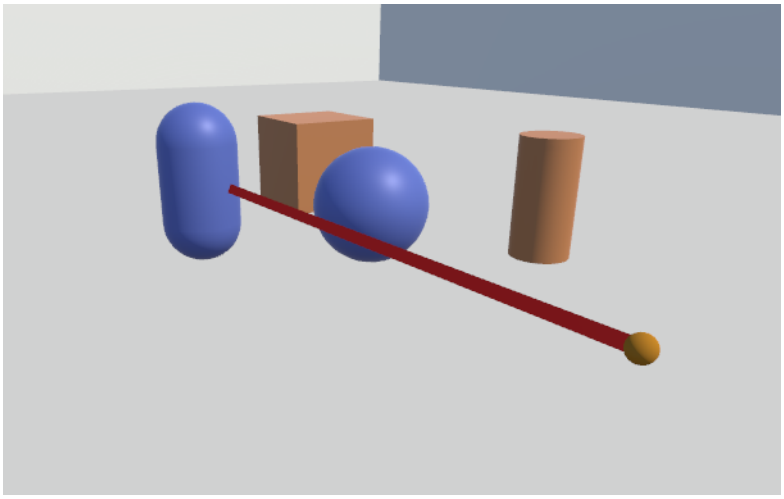

Figure 13: VRIL_SingleRayCast example

3.5.2 Depth-Ray

The Depth-Ray technique [GB06] is similar to the single Ray-Cast technique [Bol80] but instead of relying on the ray to select objects, a sphere is used to indicate the area where objects can be selected. Together with another controller, the distance between the two controllers determines the distance of the model on the ray. Linear mapping is used to determine the distance and can be adjusted freely. The greater the distance between the two registered controllers for this technique, the greater the distance of the model on the line to the controller where the ray originates. The selection model indicates the area in which an object can be selected. If another model with a different mesh is used like a quad or cylinder, the model will not accurately represent the selection area since a sphere ray cast is used to check for objects in the area. This technique provides a small set of options like the model size, initial sphere distance and linear distance mapping.

An example of this technique can be seen in **Figure 14** (UML diagram located at the image appendix **Figure 26**).

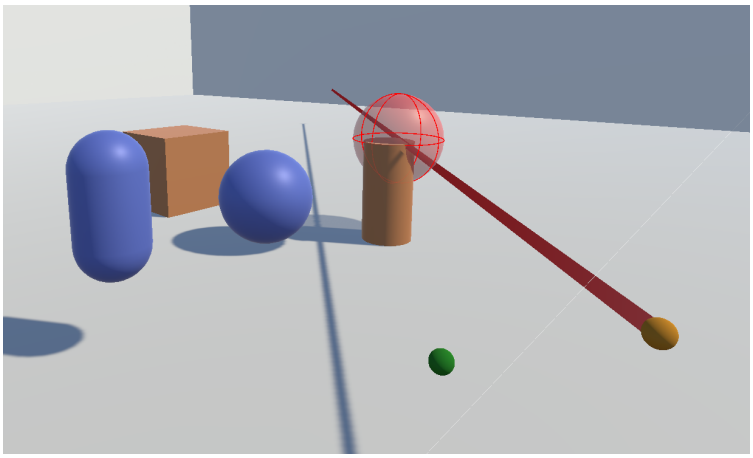

Figure 14: VRIL_DepthRay example

3.5.3 iSith

The iSith technique [WBB06] utilizes two controllers to select an object. A ray from each of the controllers is created that expands in front of the controller with a maximum distance. If those two intersect with one another a

model is created (red sphere **Figure 15**). This model can grow or shrink in size depending on the distance between the two main rays from the controllers. The model indicates the region in which an object can be selected.

The iSith technique implements three line renderers. Since the line renderers are only instantiated at runtime (there is currently no way to require 3 instances of a line renderer via class attribute), a small list of options is provided for the ray: width, color, the line material, the shadow casting mode and the maximum distance of the ray. One line is created at each controller position moving forward with the options specified. The third line from the last renderer is used to indicate the shortest distance between the two forward vectors from each controller. The line will not be displayed if one vector is not in front of the other (one controller is pointing behind the other controller). If there is a shortest distance between the two vectors, a custom model of a transparent sphere is placed in the middle between the two vectors.

The provided model will be automatically resized based on the two options *MinSphereDiagonal and MaxSphereDiagonal*. As the rays are further apart from each other the model can be resized to fit the desired size. This model can be replaced with any other model as long as no component that creates a collider[6] around the model is attached to it. The model, however, does not indicate the space in which objects can be selected if it has a different mesh than a sphere. Every frame, the interaction technique will cast a sphere, which size corresponds to the *MinSphereDiagonal and MaxSphereDiagonal* options that are used to resize the used model. Any overlapping objects that have the *VRIL_Interactable* component attached will be selected by using a sphere ray-cast. To make development easier, the *OnDrawGizmos* function from the Unity scripting API is used to show the size of the actual sphere which can be seen in the scene view in Unity if activated.

Depending on the position and if there is a shortest distance between the two lines, the provided model will be set active or inactive to provide better performance instead of deleting it. Setting the model inactive means it will not be visible anymore. If the model is invisible no objects can be selected even if the rays point at objects as the main usage is to select objects with the sphere instead of the rays.

[6]https://docs.unity3d.com/ScriptReference/Collider.html

An example of the iSith implementation in action can be seen in **Figure 15**.

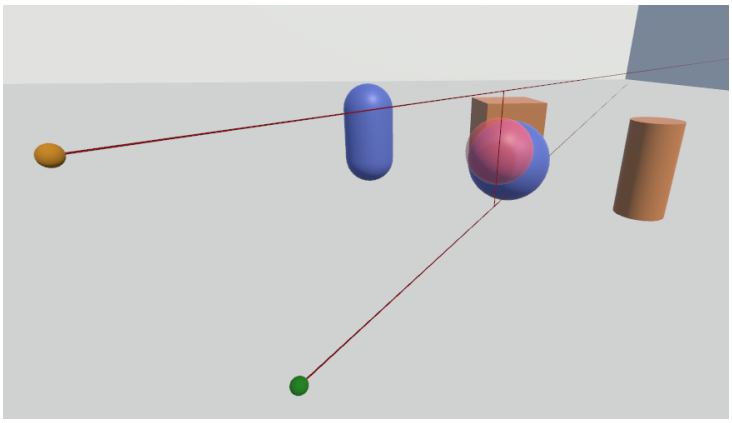

Figure 15: VRIL_iSith example

3.5.4 Spindle + Wheel technique

The Spindle + Wheel technique [CW15] is an extension of the original Spindle technique [MM95a]. While the iSith technique [WBB06] focuses on the selection part of an interaction technique by providing a unique way of selection with two rays, the Spindle + Wheel technique focuses on the manipulation part after selecting an object.

An object is selected by using both controllers and a selection model similar to the iSith technique. But instead of using two rays only one ray is instantiated. At the middle between the two controllers, a transparent model is placed to indicate the area in which objects can be selected. After an object has been selected the technique can be switched to the manipulation mode which results in the selected object being placed in between the two controllers. The manipulation mode then allows users to manipulate an object's rotation, position and the scale:

- Scaling:
  The distance between the two controllers translates to the scale of the object.

- XYZ axis translation:
  Moving both controllers allow for the placement of the object along all axes.
- YZ axis rotation:
  Asymmetric hand movement result in yz rotations on the object.
- X rotations:
  Moving the dominant-hand controller results in changed x rotations on the object that translate to the controller rotation.

An example of this technique can be seen in **Figure 16** and **Figure 17** (UML diagram **Figure 28** located at the image appendix).

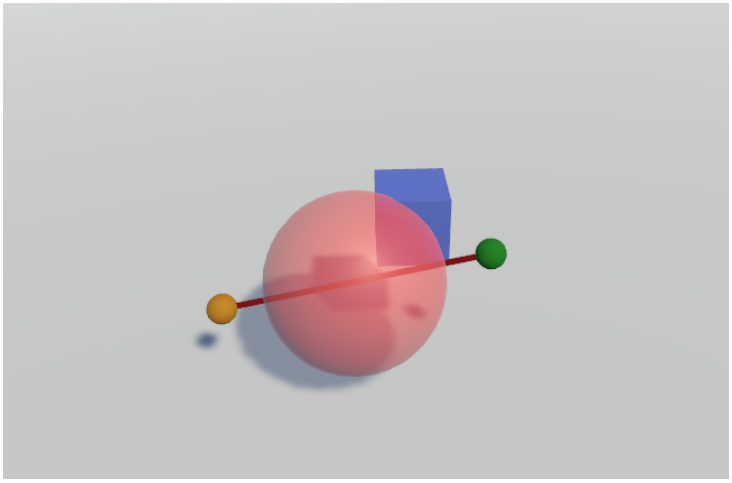

Figure 16: VRIL_SpindleAndWheel selection example

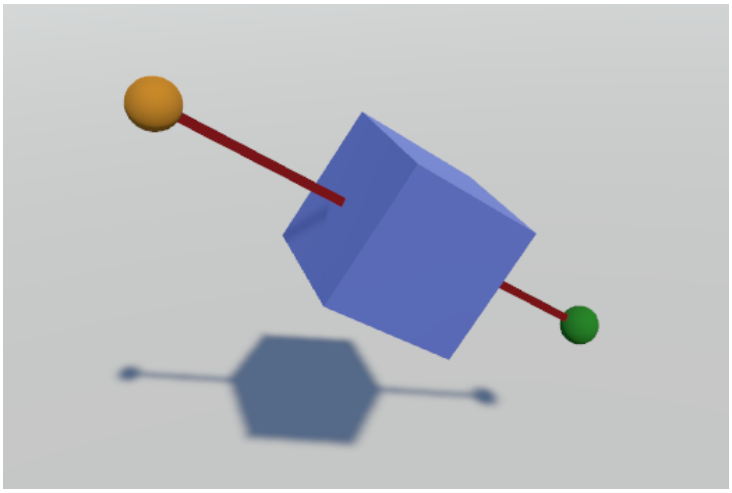

Figure 17: VRIL_SpindleAndWheel manipulation example

3.6 Navigation Techniques

3.6.1 Steering

The library provides both gaze-directed steering and hand-directed steering techniques [Min95] (UML diagram of steering class in **Figure 29** located in the image appendix). On application startup, it is identified whether the camera or controller object is used for steering, depending on the value provided in option Technique. When the steering technique is activated, the viewpoint is transferred repeatedly to a new position as long as the button is pressed. The target position is calculated by adding the object's forwards to the current position of the viewpoint. In addition, the result is calculated by the provided velocity. The steering technique provides options that must be set in advance to enable movement along the desired axes. For example, to be able to fly through the virtual space, all axes have to be enabled.

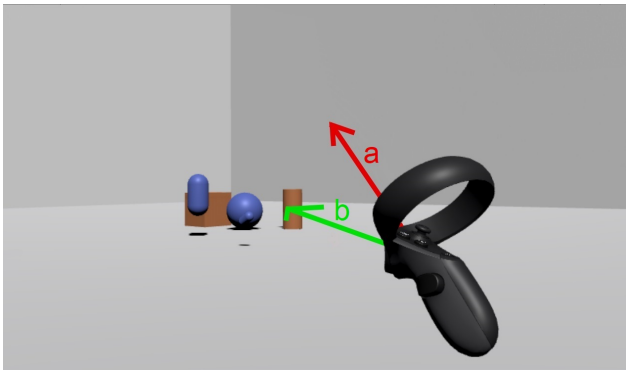

Figure 18: Hand-directed steering: (a) Direction vector (b) Calculated target position

When the hand-directed steering technique is used, an additional option provided in the Inspector window allows to enable either the pointing mode or crosshairs mode. While former simply uses the forward vector of the controller object as steering direction, the latter calculates a vector from camera to controller object. This vector is used then for the direction. An example of the hand-directed steering technique that uses the pointing mode can be seen in **Figure 18**. Movement along the Y axis is disabled here. Therefore, only X and Z of the direction vector are applied to next position.

3.6.2 Grab the Air

The grab the air technique [MM95b] (UML in **Figure 20** located in the image appendix) transfers not the viewpoint to a new position, but the whole virtual world around the user. The position of the user remains the same. To avoid moving each individual object separately, a parent world object containing all objects of the virtual world has to be attached to the technique component in the Unity editor. This world object is then used by the technique to apply position changes of the controller to the world. When the technique gets activated, a co-routine is started which continuously applies the position changes of the controller to the world object as long as the button is pressed. Each controller position change is determined by the vector from its previous position (world is not attached to the controller!). Similar to the steering techniques, options are provided to enable individual axes. To prevent the user from being affected by any position changes, the viewpoint cannot be a child object of the world, it must be separated from it.

Since many grab motions are needed for travelling larger distances within the virtual world, a scale factor is provided. The vector for the difference of the controller positions is multiplied by this movement scalor to achieve that every controller motion results in larger position changes of the world. **Figure 19** illustrates the grab the air technique by using an example scene from the Unity engine: When the grab the air technique is activated and the controller moves towards the camera (a), the virtual world comes closer to the user (b).

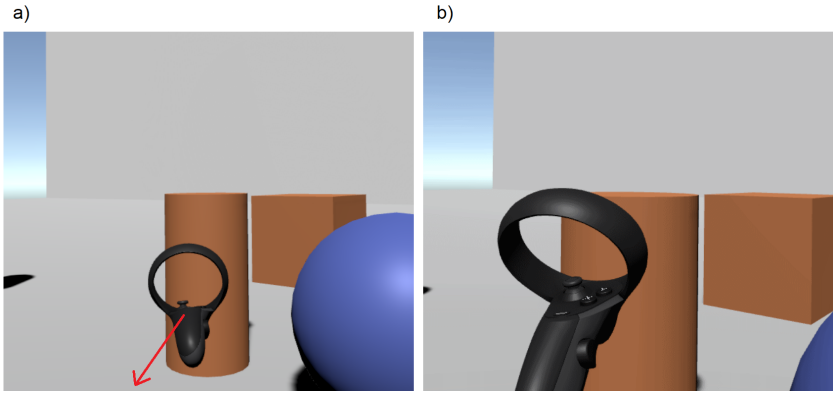

Figure 19: Unity example for the grab the air technique

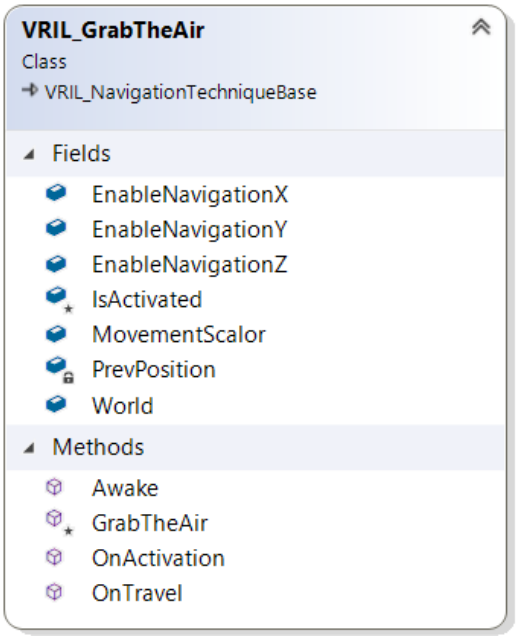

Figure 20: *VRIL_GrabTheAir* UML

3.6.3 Point and Teleport

The point and teleport technique [BRKD16] allows the user to select the desired target position in advance. The viewpoint is then transferred to this selected target position. The library provides two implementations: Blink teleport [Cre15] and dash teleport [BMF18]). Both are derived from the common base class *VRIL_TeleportBase* (UML diagram in **Figure 30** located in the image appendix) which handles the selection of the desired position. On activation, the technique enables a selection mode where a coroutine is started to display a parabola which can be controlled by the user to target to any object surfaces in the virtual space. This parabola is drawn from the controller object up to a maximum distance. When the parabola hits any object, the collision point is checked whether it can be used as a valid position to travel to. For the position detection, the target object has to be navigable and a collider[7] component must be attached to it. Also a maximum surface angle can be specified. The parabola itself is based on the projectile motion curve and is composed of multiple rays. The curve can be modified by setting

[7]https://docs.unity3d.com/ScriptReference/Collider.html

the projectile velocity in Unity editor. A *Physics.Raycast*[8] is then used that moves along these points in the parabola to check for collision within the distance between them until either a defined maximum number of segments is reached or a navigable object is hit.

For the visualization, a line renderer[9] is used, to which all points are passed. This line renderer is instantiated at runtime and can be modified in advance by the provided options for length, width, line material and the maximum number of points. Also two different colors can be set in order to visualize whether a valid target position has been selected. Depending on the decision whether the desired point can be used for travel, the color of the line renderer is updated accordingly (for example using a green color to show a successful selection and a red one for hitting obstacles or no hit). In addition, a selected position is highlighted by an object called "hit entity". This object must be attached to the script in the Unity editor and is placed at the selected position in case it is valid. **Figure 21** shows an example of the point and teleport technique.

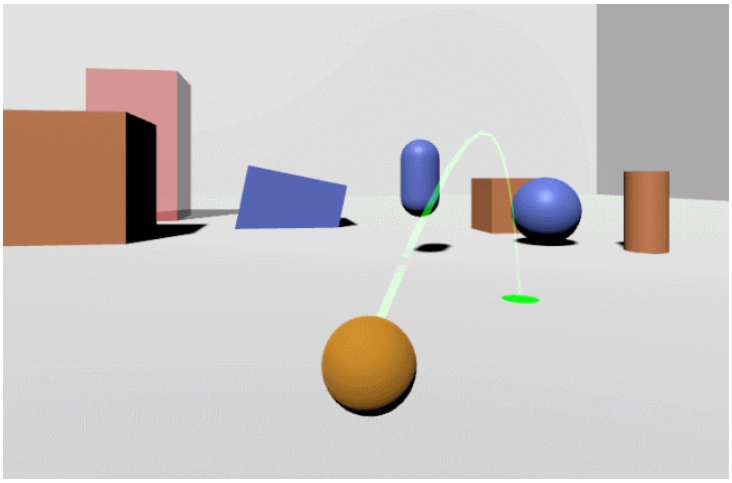

Figure 21: Point and teleport Unity example

Blink Teleport

[8]https://docs.unity3d.com/ScriptReference/Physics.Raycast.html
[9]https://docs.unity3d.com/Manual/class-LineRenderer.html

The blink teleport transfers the viewpoint instantaneously to a selected position by simply updating the viewpoint position. To achieve the effect of eye-blinking, the scene is faded out for a moment of time. Here, *SteamVR_Fade* from the Valve.VR[10] is used, whereby the scene fades out immediately and the given color is shown (invoking function with zero seconds as time to fade in). After a specified period of time, the scene fades in with the given duration:

```
SteamVR_Fade.View(Color.clear, FadeInDuration);
```

Dash Teleport
The dash teleport transfers the viewpoint continuously towards the selected position by a given velocity. For this dash movement, a co-routine is started which moves the viewpoint step by step towards the target position until it is close enough. For each step, the *MoveTowards*[11] function from the Unity scripting API is used to update the viewpoint position.

```
Viewpoint.transform.position =
↪ Vector3.MoveTowards(Viewpoint.transform.position,
↪ TargetPosition, Velocity * Time.deltaTime);
```

3.6.4 World in Miniature

The world in miniature (WIM) technique [SCP95] allows the user to select a target position by pointing with a ray on a miniature representation of the virtual world (UML in **Figure 31** located in the image appendix). The viewpoint is then transferred to the corresponding position in the large-scaled world. This technique requires two controllers. The ray hand controller activates the technique and enables the ray for the selection on the miniature world, which is attached to the second controller (WIM hand). On activation, the technique enables the ray and the miniature world is created.

First, all objects having a *MeshRenderer*[12] component attached are selected. Since not all objects are necessary in the WIM, these objects are filtered in advance. Objects having the ignore component attached, the controllers

[10] https://valvesoftware.github.io/steamvr_unity_plugin/api/Valve.VR
[11] https://docs.unity3d.com/ScriptReference/Vector3.MoveTowards.html
[12] https://docs.unity3d.com/560/Documentation/Manual/class-MeshRenderer.html

and too small objects are ignored in the WIM. For the latter, options are provided to define threshold values for X, Y and Z an object have to exceed to be included in the miniature world. The filtered objects are then cloned by invoking *Object.Instantiate*[13]. These clones are assigned to the WIM object as children, which is an empty game object. The newly created world is scaled down by simply setting the local scale of the WIM object to a provided scale factor. The WIM is then rendered on top of the controller of the WIM hand.

To avoid the miniature version becoming poorly visible in darker virtual worlds, a light source can be added (placed above the controller). It is possible to provide a layer in order the light source only illuminates the WIM by using a suitable culling mask. The WIM objects can also be updated according to their original's in the large-scaled world. When the option "Refresh WIM" is enabled, the synchronization of both worlds is performed whereby new objects get cloned and attached to the WIM and deleted ones get also deleted in the WIM.

In order to visualize the user's current position and orientation, an avatar is used for the representation in the WIM. The avatar's local Y-rotation is updated according to the rotation of the viewpoint camera and shows the position as well as the current viewing direction in the WIM. In addition to this, a second avatar is provided in order to visualize the orientation at the desired position during the selection. For this "shadow avatar" it is recommended to use the same avatar object but with a higher transparency to make both visually different. There is an option provided to specify whether the shadow avatar statically looks away from camera or its orientation can be modified by the user. For the latter, it is manipulate the shadow avatar's viewing direction by rotating the controller.

In addition to the miniature world, a ray is instantiated which is drawn from the ray hand directing to the controller's forward vector. Analogous to the point and teleport technique, only surfaces of navigable objects can be selected. Furthermore, an option is provided to define a maximum angle for a valid surface in the Unity editor. **Figure 22** illustrates an Unity example of the WIM technique.

[13]https://docs.unity3d.com/ScriptReference/Object.Instantiate.html

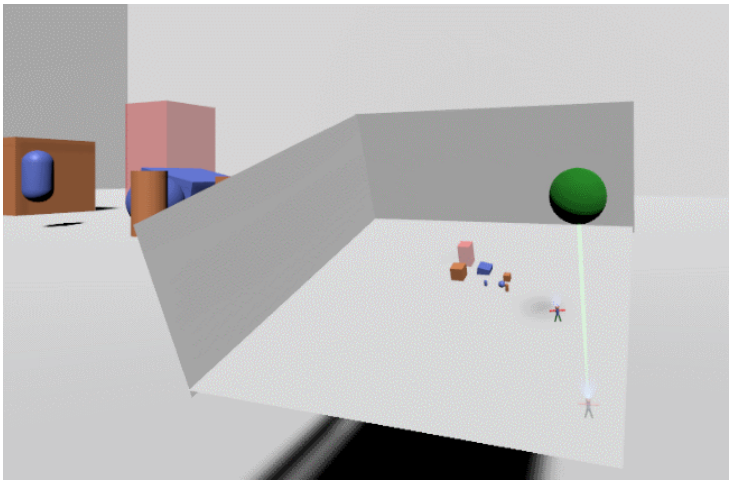

Figure 22: Unity example for the WIM technique

Teleportation Based Approach
The teleportation based approach transfers the user instantaneously by updating the viewpoint position according to the selected position on the miniature. As the target orientation is changing as well, it is necessary to rotate the viewpoint accordingly. Thereby the difference between the avatar's local Y rotation is added to the Y rotation of the viewpoint object. Since the forward direction of the camera does not always match the forward direction of the viewpoint, the local Y rotation of the camera is subtracted from the viewpoint rotation. This way the user's viewing direction remains the same even when not looking straight ahead. When the viewpoint is transferred to the target position, the hit entity is removed from the WIM object (instance is reused on next activation). Finally, the WIM object is destroyed. Both avatar objects and the hit entity are essential for travel task. However, these objects do not necessarily have to be attached in the Unity editor. Here, an empty game object is used in case no object is provided.

Flight Into the Miniature
This approach allows the user to fly into the miniature world until the position and orientation of the viewpoint is identical to the figure [PBBW95]. Instead of translating the viewpoint once, the technique starts an animation by using a co-routine to move the user towards the target position while continuously scaling up the WIM. For each step, the current local scale of the

WIM is multiplied by the scaling velocity. The WIM then is rendered with the newly calculated local scale. Furthermore, the viewpoint moves towards the hit entity position by a provided velocity (invoking Vector3.MoveTowards of the Unity scripting API). Additionally, the viewpoint rotation changes until it is identical with the viewing direction of the shadow avatar. The function *Quaternion.RotateTowards*[14] from the Unity scripting API is used which performs a rotation from a quaternion towards a provided target quaternion by an angular step:

```
Viewpoint.transform.rotation =
↪ Quaternion.RotateTowards(Viewpoint.transform.rotation,
↪ rotation, step);
```

The function is provided with the viewpoint rotation as the quaternion of origin, the rotation of the shadow avatar as target quaternion and an angular step. The latter is calculated by a provided rotation factor, the difference of rotation between viewpoint and shadow avatar and the ray length. When the viewpoint arrives at the target position, the flight animation ends and the WIM object is destroyed.

[14] https://docs.unity3d.com/ScriptReference/Quaternion.RotateTowards.html

3.7 Extendability

Since the design of the library supports extensibility, new navigation and interaction techniques can easily be created. Additionally, the mapping of controller actions to a custom script can be used to extend an existing technique even more. The manager relays controller actions, which have to be mapped into the corresponding controller actions first (provided by the library). This allows the use of interchangeable SDKs or custom input frameworks. The SKDs have to invoke the function *OnControllerAction* of the manager with a custom *VRIL_ControllerActionEventArgs* object that is used to specify what controller action was triggered.

3.8 Software used

For the scripting environment, Visual Studio[15] is used. Visual Studio provides the option to debug scripts and is integrated well into the Unity development process.

[15] https://visualstudio.microsoft.com

4 Image Appendix

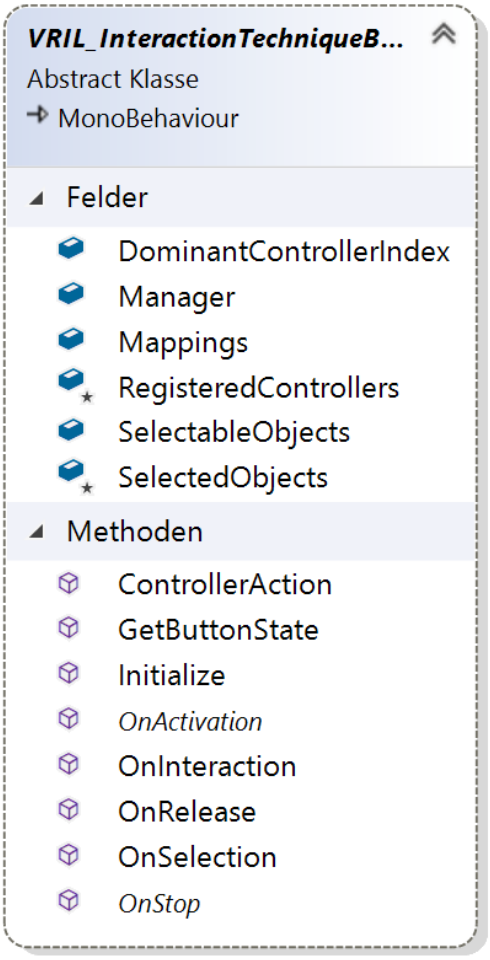

Figure 23: VRIL_InteractionTechniqueBase UML diagram

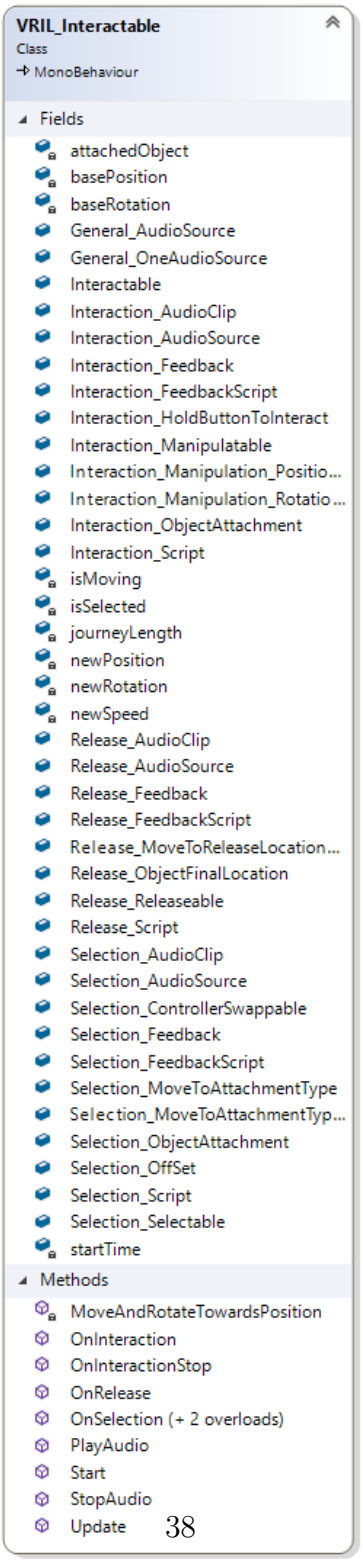

Figure 24: VRIL_Interactable UML diagram

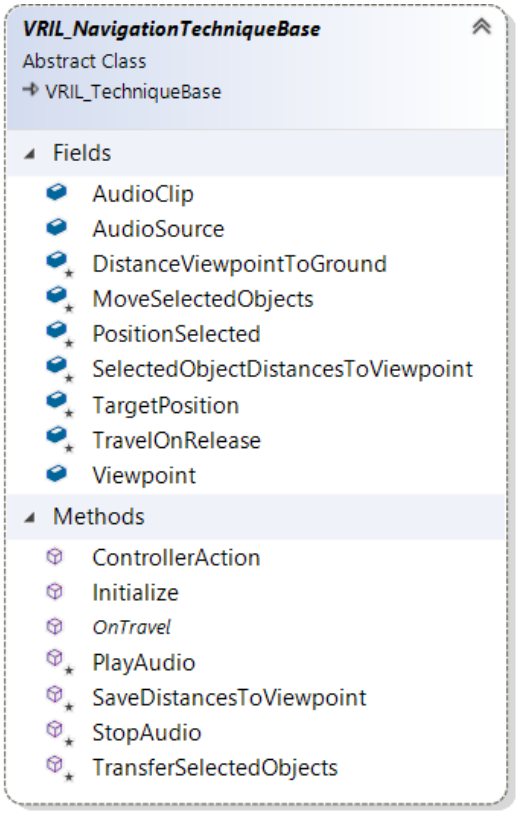

Figure 25: UML diagram of the VRIL navigation technique base class

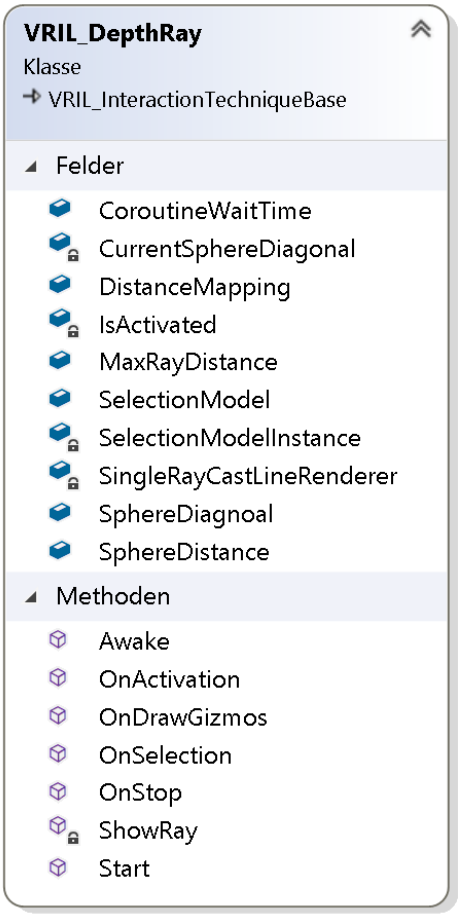

Figure 26: VRIL_DepthRay UML diagram

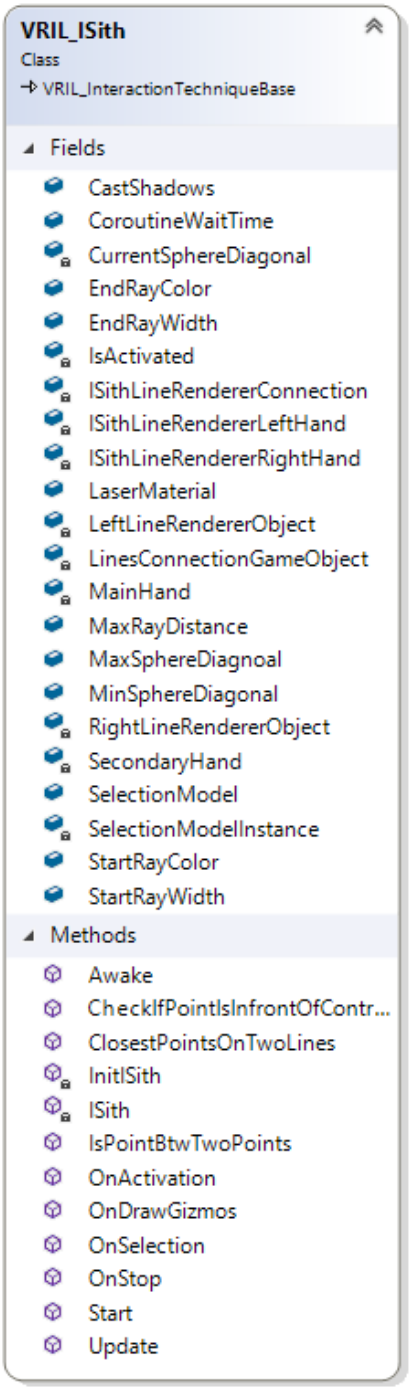

Figure 27: VRIL_iSith UML diagram

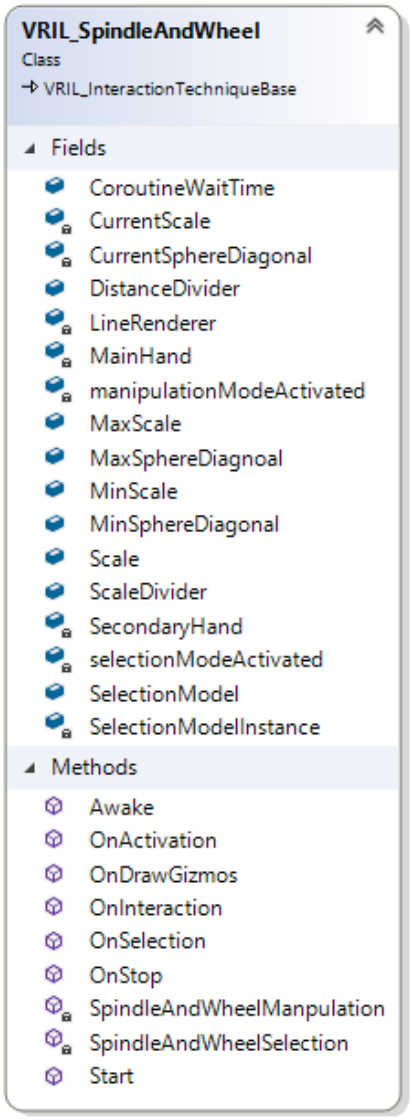

Figure 28: VRIL_SpindleAndWheel UML diagram

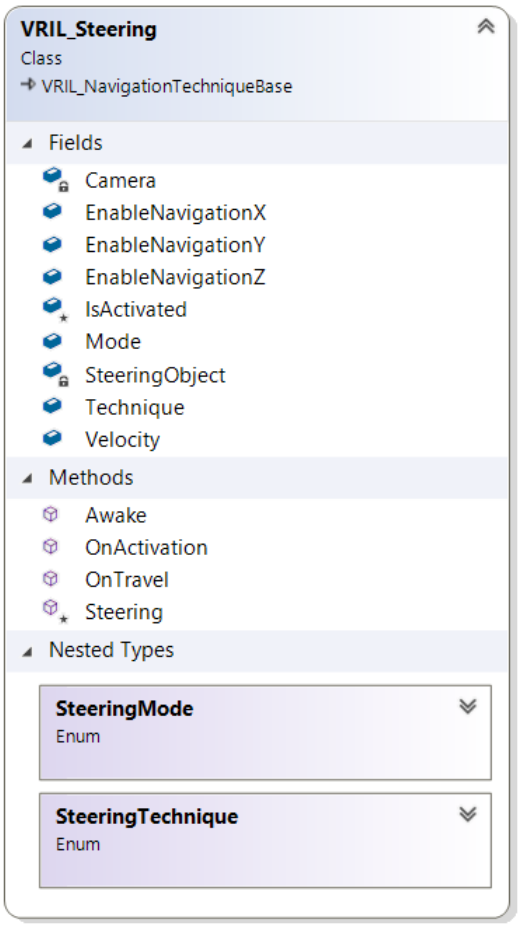

Figure 29: VRIL_Steering UML diagram

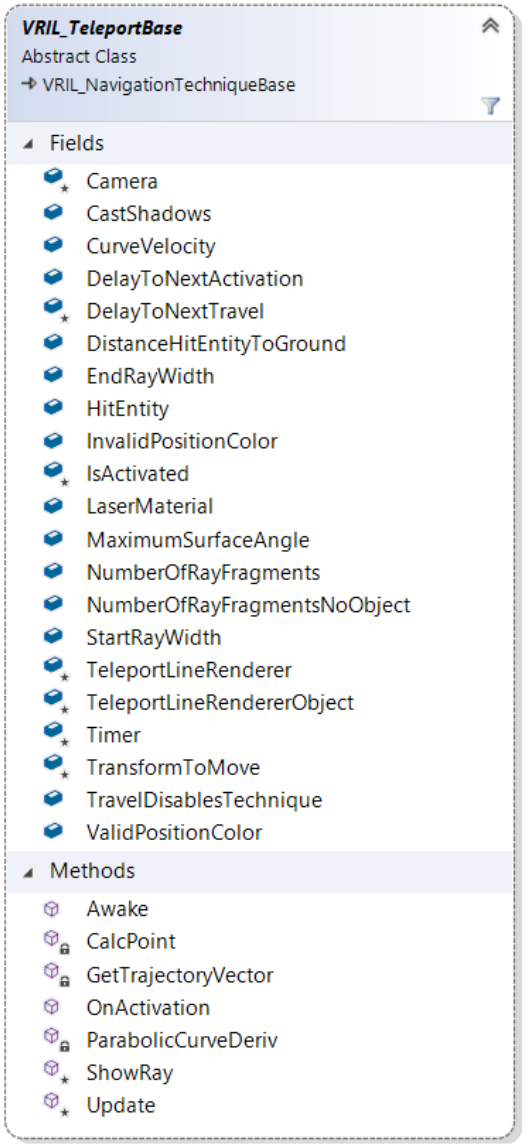

Figure 30: VRIL_TeleportBase UML diagram

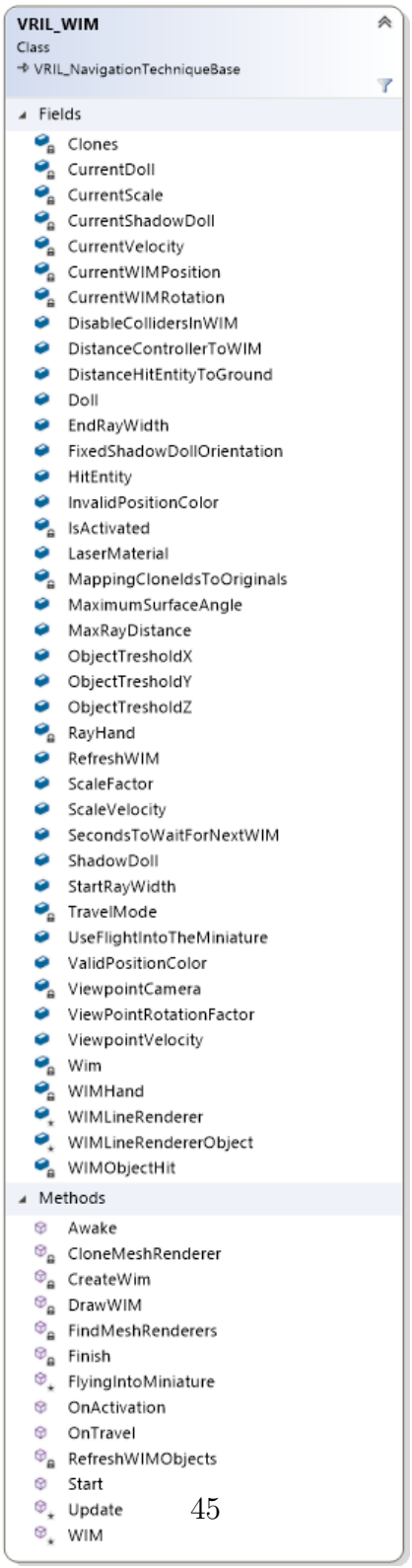

Figure 31: VRIL_WIM UML diagram

References

[BH97] Doug A. Bowman and Larry F. Hodges. An evaluation of techniques for grabbing and manipulating remote objects in immersive virtual environments. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, I3D '97, pages 35–ff., New York, NY, USA, 1997. ACM.

[BMF18] Jiwan Bhandari, Paul MacNeilage, and Eelke Folmer. Teleportation without spatial disorientation using optical flow cues. In *Proceedings of Graphics Interface*, volume 2018, 2018.

[Bol80] Richard A. Bolt. Put-that-there: Voice and gesture at the graphics interface. In *Proceedings of the 7th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '80, pages 262–270, New York, NY, USA, 1980. ACM.

[BRKD16] Evren Bozgeyikli, Andrew Raij, Srinivas Katkoori, and Rajiv Dubey. Point & teleport locomotion technique for virtual reality. In *Proceedings of the 2016 Annual Symposium on Computer-Human Interaction in Play*, pages 205–216. ACM, 2016.

[Cre15] Cloudhead Games Crew. Blink, and you'll miss us at pax prime!, 8 2015.

[CW15] I. Cho and Z. Wartell. Evaluation of a bimanual simultaneous 7dof interaction technique in virtual environments. In *2015 IEEE Symposium on 3D User Interfaces (3DUI)*, pages 133–136, March 2015.

[GB06] Tovi Grossman and Ravin Balakrishnan. The design and evaluation of selection techniques for 3d volumetric displays. In *Proceedings of the 19th Annual ACM Symposium on User Interface Software and Technology*, UIST '06, pages 3–12, New York, NY, USA, 2006. ACM.

[Min95] Mark R Mine. Virtual environment interaction techniques. *UNC Chapel Hill CS Dept*, 1995.

[MM95a] Daniel P. Mapes and J. Michael Moshell. A two-handed interface for object manipulation in virtual environments. *Presence: Teleoper. Virtual Environ.*, 4(4):403–416, January 1995.

[MM95b] Daniel P Mapes and J Michael Moshell. A two-handed interface for object manipulation in virtual environments. *Presence: Teleoperators & Virtual Environments*, 4(4):403–416, 1995.

[PBBW95] Randy Pausch, Tommy Burnette, Dan Brockway, and Michael E. Weiblen. Navigation and locomotion in virtual worlds via flight into hand-held miniatures. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, pages 399–400. ACM, 1995.

[SCP95] Richard Stoakley, Matthew J Conway, and Randy Pausch. Virtual reality on a wim: interactive worlds in miniature. In *CHI*, volume 95, pages 265–272, 1995.

[WBB06] H. P. Wyss, R. Blach, and M. Bues. isith - intersection-based spatial interaction for two hands. In *3D User Interfaces (3DUI'06)*, pages 59–61, March 2006.