The Vehicle Pattern for Simplifying Cross-Platform Virtual Reality Development

Anthony Steed*
Department of Computer Science, University College London, United Kingdom

Abstract

Maintaining an application across different virtual reality systems can be difficult and time-consuming. Different systems require different strategies for implementing simple interactions such as locomotion and object manipulation. In this chapter, we describe a pattern that we have used to minimize the interactions between the interaction style (behaviour) with the scene elements, and the specific interaction devices. The vehicle pattern allows the programmer to ignore most of the implementation details of hardware and interaction, and focus on the scene description. They can then easily generate versions of the scene for different systems. We demonstrate the principles of the vehicle pattern with an outline implementation in Unity.

1 Overview

There is now a diverse range of consumer virtual reality hardware. Unfortunately this diverse hardware comes with a diverse variety of tools, platforms, environments and plugins for development. Developers may need to get their hands on many bits of equipment in order to test their applications in order to support a variety of platforms. Users are increasingly using a variety of add-on devices for which custom code may be required (e.g. a walking platform or a glove). Larger developers who want to support the main consumer systems can afford the time to create custom user interfaces for each platform. However, smaller developers, or professional users who want to support their own custom systems, may struggle to maintain large codebases with lots of optionality for different hardware.

While there are efforts to support device abstraction such as OSVR [OSVR, 2017], and OpenXR[Khronos Group Inc., 2017], these address relatively low-level device abstractions. Their aim is to isolate APIs for specific devices so that application code doesn't need to know the specifics of, say, which tracking devices are attached. These efforts allow, or will allow, the user to switch hardware, as long as that hardware roughly matches in functionality. To the application, it might not matter exactly what HMD is attached to the computer, or which tracking system is used. However, the application still needs to make decisions about how to implement interactions between the user and the environment, and with these device abstractions the application author might need to support various different cases such as whether or not the hand-held controllers have analogue joysticks, support finger gestures, etc.

For many application developers who want to support a variety of hardware, coding to each platform is repetitive and error prone. Each application must be developed for each platform, with its different devices and its different potential ways of interacting with the user. High-level toolkits may help (e.g. Vrui [Kreylos, 2008] or VRTK [The Stonefox, 2017]), but these don't directly address the problem of making the application as portable as possible.

We note that the building of virtual reality applications involves writing two main types of code: environment-specific code that implements behaviours (interaction styles) for the application, and code that deals with input devices and implements locomotion and object manipulation. The latter code is often quite generic. It is commonly re-used between applications and might be itself quite complex. The former is often much more specific to the application, or even a specific asset within the application. It might itself be commonly reused (e.g. code for opening and closing doors), but this code doesn't need to depend on the specific details of the virtual reality interface.

Thus we propose the *Vehicle Pattern* which is an interface and set of conventions that try to separate these code concerns so that applications can be ported very quickly between hardware platforms or customised to support uncommon hardware.We use the term "vehicle" because this conveys the idea that the user needs an interface to travel and interact over long distances. In addition, this was the term used for a similar concept in an experimental virtual reality system called DIVE [Frécon and Stenius, 1998][Frécon et al., 2001]. The term "pattern" is used across many areas of design to refer to solutions to design problems that are generic and re-useable. Many readers may have come across the term in the context of software design patterns [Gamma et al., 1995] but the concept is quite general and has been applied broadly to the design of human-computer interfaces (e.g. [Seffah, 2010]).

2 Vehicle Pattern

2.1 Design

The main idea behind the Vehicle Pattern is to make as few dependencies between environment-specific code, and interaction code as possible. We illustrate the pattern by an implementation in Unity, though we have found that similar principles using different implementation strategies are useful on other platforms.

First, we can examine how the user interaction code and environment-specific code are inter-connected. Unity uses a scene graph abstraction, where a tree structure is formed from objects called *GameObjects*. Each GameObject of the scene graph has one or more instances of *Components*. Each Component type represents a type of functionality such as 3D transformations, visual and audio renderers, meshes, collision volumes, etc. The developer extends the functionality of the scene graph by writing scripts that compile to create new types of Components that can then be added to GameObjects.

There are explicit mechanisms and implicit mechanisms by which the functions of different GameObjects become inter-connected. Explicit mechanisms include Components holding references to other Components or GameObjects. It is very common for a script Component to

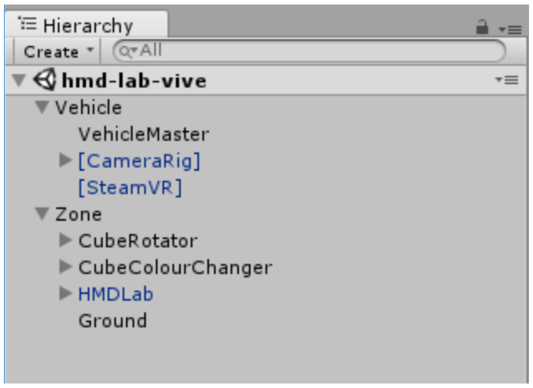
*e-mail:A.Steed@ucl.ac.uk

have a public variable which is a reference to another GameObject. The developer can assign this reference within the Unity development environment by dragging a GameObject to this variable. This makes a tight connection between the two. It is also common for scripts to look up other components and game objects based on name, type, tags or layers (see the Unity documentation for description of these). Sometimes the script will look these up dynamically, sometimes they are looked up once and then considered constant. These types of explicit connections cause problems when scene-graphs are rearranged

There are various implicit mechanisms that create relationships between objects. Proximity between objects might cause collision events in the scene, objects might interact through ray-casting, or they might even cause visual effects such as shadowing. We describe the impact of some of these in the following sections.

2.2 Vehicle and Zone

The main part of the patten is to create two isolated sub-graphs in the scene named *Vehicle* and *Zone*. It isn't strictly necessary to create two separate graphs, but it makes the different roles of the two sub-graphs very obvious and it facilitates easy deletion and replacement of the Vehicle. See Figure 1. The Vehicle contains GameObjects that isolate the device-specific and interaction-specific code. The Zone is the environment-specific code. These two sub-graphs interact implicitly: the renderers in the Vehicle will "see" the Zone objects, the collision volumes in both will interact, and physics engine will consider both sets of objects ensemble. However, we want to minimize explicit code interactions between the two or at least provide a specific point of code interaction.



```
hmd-lab-vive
  Vehicle
    VehicleMaster
    [CameraRig]
    [SteamVR]
  Zone
    CubeRotator
    CubeColourChanger
    HMDLab
    Ground
```

Figure 1: A fragment of a screen capture of the Unity Editor showing an example scene comprising a Vehicle and a Zone

If we have set up the Vehicle correctly, it should be interchangeable for any other Vehicle. In our demonstration in Unity, the Vehicles are relatively straightforward: they provide for locomotion about the environment, selection of objects and grabbing of objects. We will discus two example Vehicles that we use commonly in our own testing: one that supports the HTC Vive and one that supports interaction with the mouse and keyboard. We want to support the latter not only because it is a still a common control system, but also because during iterative development is it often convenient not to have to put on an HMD and step away from the desk in order to do a quick test. We have commonly had two or more Vehicles embedded in the Unity scene, but with all but one disabled, so that the developer can very rapidly switch between platforms.

We discuss implementation specific details in Section 2.5, but first we describe how scripts in the Vehicle and Zone can interact.

2.3 VehicleMaster Singleton

Many behaviours are initiated by input from the user. In Unity, typically a script is developed that has various callbacks that respond to different events. These include a function called once a frame (*Update*), but also functions on collision between objects (e.g. *OnTriggerEntry*), amongst others. We would like to enable a similarly simple programming model. We also want to allow the support of functions that go the other way, where objects in the scene change the vehicle behaviour. This is not discussed in this Chapter, but is an easy extension (see online materials).

We achieve these goals by adding a singleton class that represents the Vehicle's functionality. The following code from the *VehicleMaster* class, shows the creation of singleton class. A singleton class is a common software design pattern: exactly one of these objects is created at run-time, and this single object can be found by a static reference at run-time.

```
// VebicleMaster is the main interface class between the Vehicle and Zone

public class VehicleMaster : MonoBehaviour {

    private static VehicleMaster instance;

    // Construct
    private VehicleMaster() { }

    // Instance
    public static VehicleMaster Instance
    {
        get
        {
            if (instance == null)
                instance = GameObject.FindObjectOfType(typeof(VehicleMaster)) as
                        VehicleMaster;
            return instance;
        }
    }
```
Listing 1: Part of the VehicleMaster class showing the implementation of a singleton.

In that code, the `get` function represents the accessor on the public static `Instance` variable. Another script can then just access the variable `VehicleMaster.Instance` to point to the singleton object. Thus, later we will see lines of code such as the following:

```
        VehicleMaster.Instance.OnGrabStartThis += OnTouchOrGrab;
```
Listing 2: Registering a callback delegate with the VehicleMaster singleton.

where a script on an object under the Zone sub-graph uses this singleton to register a callback.

2.4 Event Delegates

The main function of the singleton is to provide a single point for registering functions to be called by user interaction. The goal is to hide any device-specific or device-ensemble configuration from the Zone. We introduce four generic events that objects can register for:

- *TouchStart*: the user reaches out and makes contact with an object
- *TouchEnd*: the user stops touching a touchable object
- *GrabStart*: the user grabs and tries to manipulate an object
- *GrabEnd*: the user drops an object that they were able to grab

Of these, the intentions GrabStart/GrabEnd are obvious: the user wants to pick up a scene object and will drop it later. Touch is less obvious. It makes sense with a device with tracked input: some proxy of the user's hand or fingers collides with the object. It is not immediately obvious with a mouse and keyboard or a rotation only device with a single button such as Google Cardboard- and Daydream-based devices. However, we think the meaning can be clear. The "user is reaching out" has an analogy to clicking on something with a mouse, or dwelling on a fixed target in the case of head gaze-based interaction. These types of interaction are very common in virtual reality systems and they are logically different to grabbing an object, which is usually triggered by holding a button.

We added a negotiation step, where the Vehicle essentially asks whether the object can be touched or grabbed. We also support a script registering interest in interactions with just its own GameObject, or in any attempt to touch or grab an object. This can support behaviours specific to a particular object (see the example in Section 2.6 of a cube that changes colour when picked up), or are general to any interaction (e.g. playing a sound when an object is dropped).

The following code excerpt (Listing 3) shows the implementation of `DoTouchStart`, which is the function a vehicle implementation would call inside VehicleMaster when it wishes to announce that the user has started to touch an object. The function takes the target of the touch, but also the object that touched it. At the moment, this source object is vehicle implementation dependent, but as discussed later, a future implementation may try to implement an abstract avatar representation so that the receiving script can tell which body part touched the object.

```
    //
    // Event Handling
    //
    // Touch events

    public delegate bool OnTouchStartEvent(GameObject target, GameObject source);
    public event OnTouchStartEvent OnTouchStartAny;
    public event OnTouchStartEvent OnTouchStartThis;

    // DoTouchStart() method calls delegate functions to notify them that the vehicle
    //     implementation is attempting to touch the object.
    //     returns true when it succesfully touches the object; false when the touch fails.
    public bool DoTouchStart(GameObject target, GameObject source)
    {
        // Trigger those delegates that registered for all callbacks
        if (OnTouchStartAny != null && OnTouchStartAny.GetInvocationList() != null &&
                OnTouchStartAny.GetInvocationList().Length > 0)
            OnTouchStartAny(target, source);

        // Trigger those delegates that registered for their individual callback
        if (OnTouchStartThis != null && OnTouchStartThis.GetInvocationList() != null &&
                OnTouchStartThis.GetInvocationList().Length > 0)
        {

            System.Delegate[] handlers = OnTouchStartThis.GetInvocationList();
            bool success = true;

            foreach (var item in handlers)
            {
                MonoBehaviour interestedObject = (MonoBehaviour)(item.Target);
                if (interestedObject.gameObject.Equals(target))
                {
                    object[] parameters = new object[] { target, source };
                    success &= (bool)item.Method.Invoke(interestedObject, parameters);
                }
            }
            return success;
        }
        return true;
    }
```

Listing 3: A fragment from the VehicleMaster Implementation showing how the touch event is processesd.

The code uses the C# event and delegate mechanism. An external script will define a function of type `OnTouchStartEvent`. Note that the external script can register on two types of events: `OnTouchStartAny` or `OnTouchStartThis`.

Within the `OnTouchStartEvent` function, there are two main blocks of code. The first calls the delegate functions of all the scripts that have registered interest in any object being touched. The second block checks that the target object matches the script that the delegate is registered to (through the cast to `MonoBehaviour` which is the base class for all script components). Note that the invocation of the delegate (`item.Method.Invoke`) returns a boolean. This is then combined with any other flags from other delegates to return to the vehicle implementation a success or failure. For example, the vehicle should stop any response to touching this object. This is useful for ignoring objects that cannot currently be touched.

2.5 Vehicle Implementations

We have developed various vehicles for different platforms. We have various custom virtual reality systems in the lab, so our vehicles tend to be quite specific. However, we illustrate the principles with two simple vehicles, one for keyboard and mouse, and one for the HTC Vive. We list short excerpts from the vehicle implementations as these are simple variants of example code.

2.5.1 Vive Vehicle

The Vive vehicle is based on the standard SteamVR Unity plugin. Our vehicle is based on code from an online tutorial that we have used in student projects [de Kerckhove, 2017].

We start with the default scene graph for a SteamVR application. This adds GameObjects for the camera system, the camera and the two main controllers. To each of the controllers we add a small sphere collider and set it to be a collision trigger. In a script attached to each controller we then add a script (`ViveController.cs`). The following code (Listing 4) shows how touch is then implemented very easily, based on the collider on the controller game object hitting other colliders in the scene. `OnTriggerEvent` is a standard Unity callback from the collision system.

```
    // Use the Unity OnTriggerEnter() method to implementation implement touch for
    // the Vive Vehicle.
    public void OnTriggerEnter(Collider other)
    {
        if (!VehicleMaster.Instance.DoTouchStart(other.gameObject, this.gameObject))
        {
            return;
        }
        SetCollidingObject(other);
    }
```

Listing 4: Part of the Vive Vehicle implementation showing how the touch event is implemented.

Note that the call through to `DoTouchStart` function can be rejected, indicating that this object is not touchable. See the online material for the detail of implementation of touch ending, and grabbing.

This demonstration implements locomotion through a simple teleport technique. This involves an implicit interaction between the Vehicle and Zone, which is that the teleport technique can only effect travel to points on objects in a layer labelled “Ground”. However, this is a simple constraint to enforce, and we can use the same labelled objects in the Mouse and Keyboard Vehicle.

2.5.2 Mouse and Keyboard Vehicle

This demonstration is based on Unity's Standard Asset package. It modifies two scripts: `RigidbodyFirstPersonController` and `DragRigidbody`.

The first person locomotion controller has been modified with a simple switch between moving mode and manipulation mode, and some code that we don't discuss in this chapter that constrains the walkable region to the same region that the Vive vehicle's teleport functionality can reach.

We illustrate the grabbing functionality in the following excerpt (Listing 5). When the user presses a mouse button, a ray is cast into the scene. The object that is hit should have a `RigidBody` component. We then call through the VehicleMaster to check whether the object can be picked up or not (`VehicleMaster.Instance.DoGrabStart`).

```
    // We need to actually hit an object
    RaycastHit hit = new RaycastHit();
    if (!Physics.Raycast(mainCamera.ScreenPointToRay(Input.mousePosition).origin,
                         mainCamera.ScreenPointToRay(Input.mousePosition).direction,
                         out hit, 100, Physics.DefaultRaycastLayers))
    {
        return;
    }

    // We need to hit a rigidbody
    if (!hit.rigidbody) return;

    if (!VehicleMaster.Instance.DoGrabStart(hit.rigidbody.gameObject, this.gameObject))
        return;
```

Listing 5: Part of the Update function for the Mouse and Keyboard vehicle that uses the Unity Physics engine to send a ray attached to the mouse into the screen.

See the online materials for the rest of the implementation of this example vehicle.

2.6 Demonstrations

A very simple demonstration is shown in Figure 2. This is a distilled version of a basic environment we use in various experiments at UCL where the user is sat in a virtual version of the lab where they are physically sat. The online materials include a second simple demonstration which is a recreation of the virtual pit demonstration [Usoh et al., 1999]; this is a common demonstration that we use with new users of virtual reality.

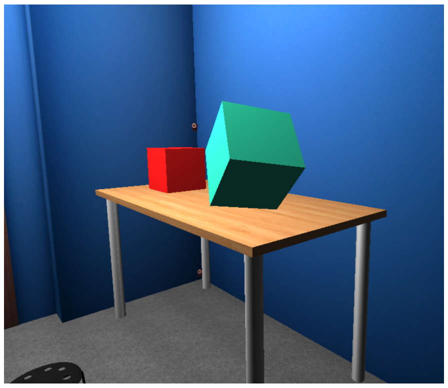


Figure 2: A simple scene modelled on our of our labs at UCL. The left cube is not manipulable and will spin when the user attempts to interact with it. The right cube is being held by the user and is changing colour.

There are two interactive objects in this environment that illustrate the main principles of the Vehicle Pattern. We show the whole scripts to emphasise how similar these are to standard script structures. The first interactive object is the cube on the left of the table in Figure 2. It is called `CubeRotator` in the scene graph in Figure 1). In Listing 6 you can see how the delegate is registered with the lines in the `Awake` callback. Note that the `OnTouchOrGrab` function rejects the touch or grab and then starts a co-routine to spin the object.

```
// BoxSpin.cs
// A behaviour that causes an object to spin when the object is grabbed.
public class BoxSpin : MonoBehaviour
{
    private bool moving;
    // register the OnTouchOrGrab script (delegate) to OTST & OGST
    void Awake()
    {
        moving = false;
        VehicleMaster.Instance.OnTouchStartThis += OnTouchOrGrab;
        VehicleMaster.Instance.OnGrabStartThis += OnTouchOrGrab;
    }

    bool OnTouchOrGrab(GameObject source, GameObject target)
    {
        if (!moving)
        {
            StartCoroutine("MoveAndWait");
        }
        return false; // Reject the touch event
    }

    // A coroutine that rotates "this" object once
    IEnumerator MoveAndWait()
    {
        float angle = 0;
        moving = true;
        while (angle <= 360.0)
        {
            transform.eulerAngles = new Vector3(0, angle, 0);
            angle = angle + 3.0f;
            yield return new WaitForFixedUpdate();
        }
        transform.eulerAngles = new Vector3(0, 0, 0);
        StopCoroutine("MoveAndWait");
        moving = false;
    }
}
```

Listing 6: The BoxSpin class spins the object when it is grabbed.

The code for the second box is very similar, see Listing 7. It registers two different delegates `OnGrabStart` and `OnGrabEnd`. The former accepts the grab event and thus the vehicle is now free to manipulate the object. It also starts to change the colour of the object. The colour changing is stopped when the latter delegate is called.

```
// ColourChange.cs
// The ColourChange behaviour causes an object to change colour repeatedly
//      when the user is holding the object.
public class ColourChange : MonoBehaviour
{
    private bool changing;
    Renderer rend;              // Store "this" object's renderer component

    // When "game" starts (world is initialized)
    void Awake()
    {
        rend = GetComponent<Renderer>();
        changing = false;
        VehicleMaster.Instance.OnGrabStartThis += OnGrabStart;
        VehicleMaster.Instance.OnGrabEndThis += OnGrabEnd;
    }

    // While "this" object is grabbed change colour.
    bool OnGrabStart(GameObject source, GameObject target)
    {
        if (!changing)
        {
            StartCoroutine("ChangeColour");
        }
        return true; // Accept the Grab event
    }
    // Disable the colour change when the grab is ended
    bool OnGrabEnd(GameObject source, GameObject target)
    {
        if (changing)
        {
            StopCoroutine("ChangeColour");
            changing = false;
        }
        return true;
    }

    // Change colour of "this" object for every .5 seconds until the co-routine is cancelled.
    IEnumerator ChangeColour()
    {
        changing = true;
        while (true)
        {
            rend.material.SetColor("_Color", new Color(Random.Range(0F, 1F),
                Random.Range(0, 1F), Random.Range(0, 1F)));
            yield return new WaitForSeconds(0.5f);
        }
    }
}
```

Listing 7: The ColourChange class behaviours causes an object to change colour repeatedly when held.

3 Discussion

3.1 Constraints and Limitations

The separation between Vehicle and Zone relies on several implicit assumptions. For example: a natural human scale of objects; an understanding that the Vehicle will not create collision volumes that are too large; and that the camera will move at a certain range of speeds (so that collision detection works), etc. The Vehicle Pattern as described is sufficient for basic applications that do not have highly customised needs for interaction. The pattern is easy to extend to fit specific needs, and we continue to develop our example implementation.

For example, an obvious extension would be to develop a standard representation of the user's avatar so that environment-specific interaction could start to address the avatar's representation. This would usefully include scene graph objects that components can discover which indicate the user's head position, hand positions, standing position, etc., so that even environment-specific scripts can reference them. For example, a script might want to animate an avatar so that it looks at a user. It would also be useful to have a more refined collision volume associated with the user so that fine-scale collision detection can be done. For a Vehicle based on a mouse and keyboard, or other interface without full 3D tracking, some of these avatar object positions would need to be hypothesised based on the camera position and user interaction.

When we get to the area of dynamic user interfaces that construct visual representations inside the scene such as menus, the decisions become more difficult. The separation of touch and grab works in many situations, but there is currently no fall-back for an environment behaviour that is triggered by a specific button on a controller. If this was needed the programmer would have to customise the Vehicle itself. However, in our opinion abstractions such as the Vehicle can be extended to cover a very wide range of application needs.

The Vehicle Pattern doesn't deal with porting of visual and audio assets between different platforms. Some platforms are significantly less powerful than others. When producing a game in Unity it is common to simplify assets for low-end platforms and keep different versions of scenes. How assets can be managed in real-time to support low-end platforms remains a research question.

3.2 Related Work

The Vehicle Pattern is strongly influenced by prior work in the area that uses similar principles. In particular models of web-browsing where the user has a lot of control over their web browser's behaviour. Although a lot of functionality is fixed, users can customise their web browser with various extensions. Presently, most VR development is not supporting this type of customisability for the user. Our Vehicle Pattern highlights the fact that by having developers begin the development process with device abstractions, interaction techniques are no longer customisable by the end user. The Vehicle Pattern is part of a skeleton implementation by the author, called Yther, that proposes one type of solution to this dilemma [Steed, 2015].

Previous efforts to support a broad range of 3D interfaces have acknowledged similar problems. The VRML97 and then X3D standards ([VRM, 1997], [X3D, 2013]) had specific methods to support device independence. For example, the `NavigationInfo` node had a recommended type of locomotion, such as " Walk" or "Fly" that the browser should support. In addition manipulation of objects was done in a way that could imply motion constraints. For example the `PlaneSensor` node supported objects that could be dragged along a surface. It was up to the VRML or X3D browser to determine how to implement the dragging within the user interface.

Our pattern is most strongly related to the vehicle concept from an older research platform called DIVE [Frécon and Stenius, 1998][Frécon et al., 2001]. This was browser-centric in the sense that scenes were developed independent of user interaction specification, and each user installed and ran a browser that could implement interaction in various ways. In fact, the browsers themselves were highly customisable (they were written in the TCL language), so a browser could dynamically switch its interaction style, or in the DIVE terminology switch "Vehicle". The system was event-based, with decoupled message passing between the browser and in-scene objects. Some of message types are similar to the delegate types we propose: they included grasp, select and move events. They also included various events for multi-user interaction, and other application-level events such as loading of sub-scenes. Although we have previously argued for asynchronous message passing as a mechanism to decouple user-interaction and environment-specific code, delegate-based calling seems more appropriate in modern platforms where latency is paramount [Steed, 2008]. Asynchronous message passing has its advantages, but especially in Unity there are significant downsides. For example, developers often rely on knowing the rough ordering or lack of ordering of different functions (e.g. callbacks from different internal functions such as fixed-update callbacks, the order of script evaluation in a scene graph, .Net 2.0 yield functionality, etc.), which can match poorly with asynchronous message-passing.

4 Conclusions

The excitement around consumer virtual reality, and the ready availability of a variety of new devices means that there are great opportunities to develop new user interfaces. Unfortunately, there is a dearth of high-level toolkits that simplify support for a range of devices. While development environments such as Unity or Unreal Engine enable development for many devices, porting code between different devices, or ensembles of devices is tricky.

In this chapter we have described the Vehicle Pattern, which attempts to separate the concerns of programming support for user-interaction on devices, from virtual environment-specific behaviour. This pattern has proved very useful in our lab, where students may want to develop on one machine, but quickly deploy to another that supports a specific hardware configuration. While the skeleton implementation is simple, we hope that it can be useful for others as is. We also hope that the principles can help inform future toolkits so that some conventions, or even standards, can emerge that alleviate some of the need for the developer to consider which virtual reality systems their content may run on.

References

[VRM, 1997] (1997). ISO/IEC/DIS14772, The Virtual Reality Modeling Language.

[X3D, 2013] (2013). ISO/IEC IS 19775-1, X3D Abstract : Node Definitions. `http://www.web3d.org/documents/specifications/197751/V3.3/index.html`.

[de Kerckhove, 2017] de Kerckhove, E. V. (2016 (accessed July 1, 2017)). HTC Vive Tutorial for Unity. `https://www.raywenderlich.com/149239/htc-vive-tutorial-unity/`.

[Frécon et al., 2001] Frécon, E., Smith, G., Steed, A., Stenius, M., and Ståhl, O. (2001). An overview of the coven platform. *Presence: Teleoperators and Virtual Environments*, 10(1):109–127.

[Frécon and Stenius, 1998] Frécon, E. and Stenius, M. (1998). Dive: A scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering*, 5(3):91.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[Khronos Group Inc., 2017] Khronos Group Inc. (2017 (accessed July 1, 2017)). OpenXR, cross-platform, portable, virtual reality. `https://www.khronos.org/openxr`.

[Kreylos, 2008] Kreylos, O. (2008). Environment-independent vr development. In *Advances in Visual Computing: 4th International Symposium, ISVC 2008, Las Vegas, NV, USA, December 1-3, 2008. Proceedings, Part I*, pages 901–912. Springer Berlin Heidelberg.

[OSVR, 2017] OSVR (2015 (accessed July 1, 2017)). Open source virtual reality. `http://osvr.com/software.html`.

[Seffah, 2010] Seffah, A. (2010). The evolution of design patterns in hci: from pattern languages to pattern-oriented design. In *Proceedings of the 1st International Workshop on Pattern-Driven Engineering of Interactive Computing Systems*, pages 4–9. ACM.

[Steed, 2008] Steed, A. (2008). Some useful abstractions for re-usable virtual environment platform. In *In Proceedings of the IEEE VR SEARIS Workshop*.

[Steed, 2015] Steed, A. (2015). Yther: A proposal and initial prototype of a virtual reality content sharing system. In *Proceedings of the 25th International Conference on Artificial Reality and Telexistence and 20th Eurographics Symposium on Virtual Environments*, ICAT - EGVE '15, pages 151–158, Aire-la-Ville, Switzerland, Switzerland. Eurographics Association.

[The Stonefox, 2017] The Stonefox (2017 (accessed July 1, 2017)). VRTK - Virtual Reality Toolkit. `https://vrtoolkit.readme.io/`.

[Usoh et al., 1999] Usoh, M., Arthur, K., Whitton, M. C., Bastos, R., Steed, A., Slater, M., and Brooks Jr, F. P. (1999). Walking¿ walking-in-place¿ flying, in virtual environments. In *Proceedings of the 26th annual conference on Computer graphics and interactive techniques*, pages 359–364. ACM Press/Addison-Wesley Publishing Co.