International Conference on Artificial Reality and Telexistence
Eurographics Symposium on Virtual Environments (2015)
M. Imura, P. Figueroa, and B. Mohler (Editors)

Ythor: A Proposal and Initial Prototype of a Virtual Reality Content Sharing System

A. Steed[1]

[1]Department of Computer Science, University College London

Abstract

There are many virtual reality development tools, but we argue that none has solved the disparate problems of ease of content creation, openness of access for developers and broad support for a variety of immersive and non-immersive virtual reality systems. Ythor is a proposal and initial prototype of a platform for sharing immersive and non-immersive virtual environments. Ythor doesn't propose a new content standard, but builds upon the Unity system. The first contribution of Ythor is a clear separation of concerns between development of content (zones) and support for specific hardware. The second contribution is a set of abstractions that allow developer to author interaction techniques and behaviours without specific reference to the interaction devices that any particular runtime instance has. The third contribution is a mechanism within the Unity development environment for publishing content so that it can be made available for testing and sharing very easily. The current implementation of Ythor is a prototype but it already demonstrates capabilities that distinguish it from similar systems.

Categories and Subject Descriptors (according to ACM CCS): INFORMATION INTERFACES AND PRESENTATION [H.5.1]: Multimedia Information Systems—Artificial, augmented, and virtual realities

1. Introduction

The sharing of content between virtual environments systems is a complex technical problem. There are several issues: the variety of content that needs to be supported, the different architectures of the display systems, and the range of potential interface devices and interaction techniques. The need for common platforms for sharing content is given new impetus by the growing interest in consumer virtual reality. Larger content developers will be able to develop support specifically for each consumer platform, but there may be several consumer platforms, and within each consumer platform there may be different configurations of input devices. This may restrict the ability of small developers and hobbyists to access a large population of users

At the current time there is no roadmap, nor obvious stepping stones, from current practice to a virtual reality standard that supports simple content development. Some key features that such a standard would have would be openness of access to users across different hardware platforms, ease of access for content developers to create and publish content, portability of content across different hardware systems that users and developers have, and freedom to extend the platform.

In this paper we describe and justify the design of Ythor, which is a modest attempt to create a framework to facilitate sharing of certain types of virtual reality content. Ythor, as currently defined and implemented, is not an attempt to create a standard, but is offered as an example of how key abstractions can simplify the sharing of content. Ythor is also open, in that various parts of the Ythor system will be made open source. Anyone will be able to use Ythor for free to create and share content, and that content should be usable by users that have a broad range of virtual reality systems. Those users will not have to download multiple executables. In effect, from a user's point of view, they will experience a "cyberspace"-like virtual environment, where different virtual environments can be accessed by teleporting to named locations, or by traversing portals in the environment.

As described in more detail in Section 3, Ythor allows a content developer to work in the Unity system to create a *Zone*. A Zone can be a whole virtual environment, or simply a piece of one. By imposing certain constraints and conventions on the content and code that animates the content, a

DOI: 10.2312/egve.20151323

Zone can be made that is independent of the specific input devices or hardware that the developer, or any of their potential users may have. The content developer can then publish the content and upload it to a content hosting service. Each user will have a *Deck*, which is an executable that can browse Zones. The user will chose a Deck based on the hardware that they have, and new Decks can be programmed, also in Unity, for specific hardware. Thus a member of the general public might just download a Deck designed for the consumer device they own, whereas a university lab might develop their own Deck for their specific hardware or a Deck that embodies a particular type of new interaction technique.

From a practical point of view Yther provides two pieces of infrastructure. First: plugins and example assets for the Unity system that facilitate the creation of Yther Zones in a away the decouples Zone from Deck (see Section 4). Second: example Decks and source in Unity demonstrating how these Decks were constructed (see Section 5). The key feature that distinguishes Yther from similar systems is the set of conventions that decouple virtual environment content from the specific hardware.

2. Background

2.1. Virtual Reality Systems

The set of requirements for software systems that support a broad range of virtual reality application is very large [CSW09]. A system needs to generate high-fidelity real-time output and respond quickly to data from a variety of input devices. Furthermore virtual reality systems may need to support many display devices and thus they may be distributed across multiple physical machines that need to be synchronized.

A broad commercial virtual reality software system such as Vizard [Wor15] attempts to provide a uniform development environment despite differing underlying hardware and software. However, such a package cannot fulfill all the needs of the potential user base. Thus a variety of other standards, libraries and frameworks have emerged.

To model the appearance of the virtual environment, the use of some sort of scene-graph based representation is almost ubiquitous [Sow00]. Two competing requirements in scene-graph choice are the efficiency and expressiveness of the run-time system, and the support for import of assets (geometry, textures, sounds, etc.) from external sources. Combined with the increasing use of Internet-based technologies, this had led to a few standards for scene representation, including X3D [Web15], Collada [Khr15a] and glTF [Khr15b]. These have quite different aims: sharing of interactive 3D scenes to end-users' browsers, support of interchange of assets in production of content and creation of fixed assets for OpenGL-based browsers. Given the explosion of interest in consumer virtual reality, several interesting prototypes of online content storage have emerged. A notable example is the JanusVR systems [McC15] which embeds 3D scene description within HTML.

The scene-graph is only one component that is necessary in a run-time virtual reality system. Virtual reality systems are characterized by the range of different devices that they need to support. VRPN is a commonly-used library that supports a variety of virtual reality peripherals [Uni15b]. The Diverse [KASK02] and VRJuggler [BJH*01] toolkits provide similar device support, but they also provide higher-level frameworks for developing applications. However, these toolkits focus on supporting the programmer, not the general virtual reality developer, who might expect to focus on scripting and asset creation.

Some toolkits have extended X3D with the functionality needed to describe the novel interfaces and display configurations. For example, Behr et al. [BDR04] and Figueroa et al. [FMJ*05] have proposed extensions to X3D to support new input devices. While these proposals do allow different platforms to be targeted within customized X3D-browers, we argue that a separation of concerns should be made, and that the scene should be described independently of specific interface devices (see Section 4). Our proposal is to clearly separate a browser-like piece of software from the scene description. Efforts such as X3Dom [BEJZ09] and XML3D [SKR*10] that embed X3D within standard web browsers such as Chrome and Firefox, are a step in a similar direction. Importantly though we also note that there is not likely to be a single standard "browser" for virtual reality scenes because of the variety of hardware. Thus to break the potentially circular requirements, Yther targets the programming of both browsers and scenes.

A complementary issue in virtual reality system design is the programming language used. It is common to support a high-level scripting language, with the aim of simplifying application description by exposing an interface to a framework that hides underlying implementation complexity. Thus the scripting language might be concerned with behaviours of the objects in the scene, rather than low-level message passing or display context creation. The variety of languages proposed to be used is very wide including TCL (in the DIVE system [FS98]), Java (OpenWonderland [Ope15]), Python (various, including [Wor15]) and JavaScript (various, including [Fra15]).

Such high-level languages are very powerful, but the framework that they address is arguably more important than the language itself. There is no common agreement on frameworks for virtual reality systems. One proposed standard is Open Source VR (OSVR) [OSV15] but this currently targets an SDK, not a flexible architecture that is composable at run-time. Long-term sustainability of frameworks has previously been addressed by run-time composable plugin systems (e.g. NPSNET-V [CMBZ00]), though such systems have not been widely deployed. A plugin system is currently considered to be a mid-term need for Yther, however, Yther

is based on a mature and extensive game engine, so a lot of content will not need to access such a plugin system.

2.2. CVEs

CVEs are an increasingly important application of virtual reality systems. The potential requirements for CVE software are very broad [OJP*09]. Two reference books give an overview of the field [SZ99], [SO09].

Ythor is not currently targeting the development of a standard large scale CVE in the short-term but it is a basis for investigation of such systems. It uses a notion of individual regions (Zones) that are not laid out in a unified coordinate system, but are linked together. Perhaps the first system to use this approach was DiamondPark [BWA96].

CVE platforms do commonly solve a related systems issue: distribution of code and behaviour descriptions to local clients in order that certain reactions (e.g. physics and direct manipulation of objects) can be simulated at the rendering rate and at low latency. Thus many CVEs split behaviour execution responsibilities between a server and a client. For game engines, a typical strategy is frequent patching of the game client to make sure behaviours are up to date. An alternative is to distribute code within scene descriptions. A variety of options have been explored for this: DIVE [FS98] shared TCL code snippets that used a message-passing mechanism to share state; Ygdrasil [PADD03] used a scene description language to instantiate classes described by plugins.

In the current prototype, Ythor does not propose a common infrastructure for CVEs, but a CVE is one application of Ythor.

2.3. Unity

Unity [Uni15a] is a modern game engine that is very popular with small and medium-sized developers. This attention has been spurred by the availability of a package for Unity that supports the Oculus Rift [Ocu15].

Unity provides an integrated development environment (IDE) that combines scene editing and scripting. The developer can export a run-time package for one of several platforms, including Windows, Mac OSX, Android, iOS, Google Native Client, PS4 and XBox One.

We will describe only a few key features of Unity that are important for later discussion. The developer works on a project that has one or more scenes. Unity eventually outputs executable code along with content packages. Each scene is described as a scene-graph comprised of nodes that represent the visual and audio representation, as well as behaviour. Behaviors can be scripted in UnityScript (a minor variant of JavaScript), C# and Boo. Behaviors can access a run-time API for the main engine and modify other nodes in the scene. Native code can be incorporated through a plugin interface. Through the run-time API and standard language features and plugins, behaviours can include interacting with network services. Aside from the run-time API, Unity provides an API for scripts to alter the scene editor component of the IDE. Under the hood, Unity manages its code using Microsoft .Net assemblies. It uses the Mono framework to support .Net functionality on some non-Microsoft platforms. On platforms that do not allow dynamically loaded code, including iOS, Unity reverts to static compilation. Thus at the current time, Ythor would not be able load new scripts on iOS and we have not investigated this platform. The lack of dynamically-loaded code is an Apple policy, not a limitation of Unity, and thus any similar effort would also suffer the same problems.

3. Architecture

The over-arching goal of Ythor is to facilitate the sharing of virtual reality content between developers and users. Ythor is built upon the Unity system, which has gained a broad use in the virtual reality community. However, even within Unity, we have personally found it hard to share content between different virtual reality systems. Unity's developement model comes from the game development community. It targets the generation of an executable and a package of content that can be distributed via physical media, or online. However, the current default model of development is that if the developer wants to support a particular piece of hardware, say a HMD, they download a plugin or package of content that supports that HMD. Then they include certain assets and code within their scenes and build an executable and package that supports that hardware. If multiple devices (e.g. a HMD and a 3D input device) are to be supported, then the plugins and packages for both are included. Unity provides mechanisms to facilitate sharing of content between scenes during production, but supporting various hardware generally means compiling different executables.

This model can be contrasted with the general model of the web, where the user has a standard web browser that they use for most of their interactions with the web, with the possible exception of some websites or web services, that by accident or design only work in certain browsers. This interoperability between browsers is facilitated by content and scripting standards, such as HTML and JavaScript, though these standards are slowly changing.

3.1. Zone and Deck

Ythor's first architectural design decision is to make an analogy to the web, by defining a *Deck*† analogous to a browser, and a *Zone* analogous to a page. A user will usually use one

† The term 'Deck' is borrowed from William Gibson's *Neuromancer* novel and its sequels.

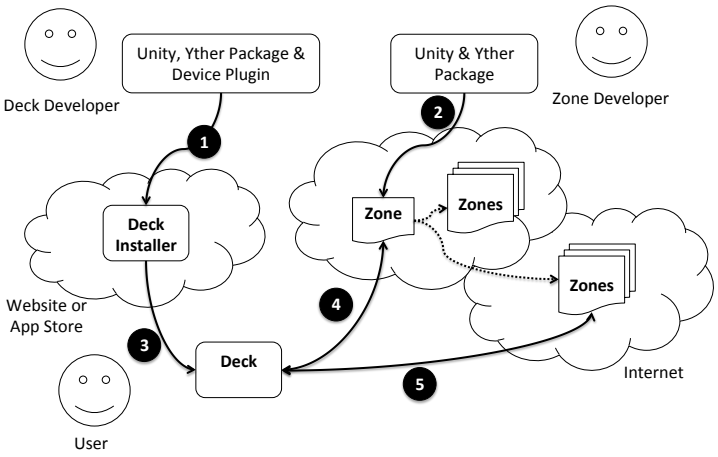

Figure 1: *The interaction between Ythor User, Deck Developer and Zone Developer*

Deck for each hardware configuration that they have. Each Deck should be able to load all the Zones. Zones are stored on content services on the web, and users download Zones to the Deck in much the same way that browsers download pages. Zones can link to each other.

A similar distinction can be found, say, the analogy between an X3D browser (e.g. a browser with a suitable plugin) and X3D content, or between a Second Life browser and an island in Second Life. However, one current intention of Ythor is to allow experimentation with the implementation of Decks very easy: this is because consumer hardware is changing quickly, but also, it is easy to foresee that there will be a lot of innovation around input devices in the short to medium term. Thus there will need to be flexibility to reprogram Decks.

Thus the second main decision is to base Ythor on Unity, with the intention that both Decks and Zones are constructed within Unity. That is, the executable process that is the Deck is programmed in Unity using whatever plugins, packages and scripts are necessary to interface to the hardware; but also Zones are packages of content that can be loaded by a Deck. Thus the standard Unity process of compiling a set of scenes into a single executable and packages of content that can be distributed, is split into two parts that might be done by two different programmers: compiling an executable (the Deck) and producing packages of content (Zones).

We thus distinguish two categories of developers, both of whom are Unity users: Deck Developer and Zone Developer. It is intended that Zone development be relatively easy because a Zone Developer is mostly concerned with visual assets and object behaviours, rather than dealing with specifics of input. In making the separation though, we must identify how the Deck and Zone communicate. This is where a set of conventions and constraints are imposed. These conventions and constraints will be the subject of Section 4, but critically, because we are using Unity to program the Deck, the developers of a Zone can rely, either directly or indirectly, on the Unity frameworks that are provided such as the various APIs at compile time and the associated run-time facilities. For example, physics interaction will be provided by the standard Unity frameworks. On this we provide abstractions for representing the user and a publish-subscribe mechanism that means that scripts in the Zone do not need to be bound to their callers at a compile stage.

A Zone can be packaged within Unity and hosted on the web. Zones can be linked together by *Portals* that trigger the loading of new Zones.

Figure 1 gives a summary diagram of the interactions between the Ythor User, Deck and Zone Developer, which can be described as follows:

1. Deck Developer using the Ythor Package and packages for other hardware specific support within Unity, publishes an installer for a Deck, onto a website, or some app store.
2. Zone Developer using the Ythor Package within Unity publishes a Zone on to the web.
3. Ythor User selects and installs a Deck for the hardware that they have.
4. Ythor User uses their Deck to fetch and browser a Zone.
5. Ythor User can follow links through Portals to other Zones.

3.2. Vehicle and Avatar

To facilitate inter-operability of content, Ythor attempts to decouple world behaviour from interface description. A key part of this is to define a standard avatar skeleton that the user controls. However, this control may be direct or indirect, and the avatar may or may not be rendered. The key role for the avatar is to represent, to the world, the user's presence and behaviour. Thus a representation of the user is available for behaviours to respond to. A more complete discussion of the issues in building avatar can be found in Section 4.4. This representation might be controlled by a mouse, gesture interface or motion-capture system.

This avatar may or may not be visible. With Ythor's having been developed initially for sharing research demonstrations, self-representation is a key issue in virtual reality, with self-avatars being difficult to control accurately. Thus a self-representation for a 1st person system is a complex behaviour. Despite the availability of standard skeleton avatars, our experience is that these can be difficult to rig for immersive use and inevitably require some tweaking (e.g. to set offsets for trackers, correct joint placement, etc.). Thus, for the moment, we leave self-representation as a role of the Deck until an appropriate standard for rigging avatars emerges. It may be that this becomes a common part of a module hierarchy to build Decks.

3.3. Unity Support

Ythor includes a Unity package that provides assets for standard purposes (e.g. Portals as described above, and also DeckMaster see Section 4) as well as editor scripts to support publishing and management of content. This allows a developer to create a Zone. To support the creation of new Decks, various different example Decks are provided, see Section 5.

4. Decoupling Zone and Deck

As discussed in Section 3 Ythor attempts to decouple description of the browser (Deck) from the virtual world (Zone). Unity is thus used to develop Decks and separately Unity is used to program Zones. The explicit goal is that whoever develops a Zone need not design for specific hardware. The developer will probably have access to several Decks, but they will only need to test the content on one.

If we consider the running instance of a Deck that has loaded a Zone, from a Unity run-time point of view, there will be a scene-graph containing two sub-graphs. The first sub-graph is that was constructed and loaded by the Deck at initialization. This will include scripts and plugins that describe the main hardware interfaces for the Deck including camera views, devices interfaces, any heads-up-display required, etc.. The second sub-graph represents the Zone. One sub-graph will be loaded and unloaded (the Zone) when the user moves between Zones, but the other sub-graph will remain. In the following sub-sections we outline how the Deck and Zone interact to effect behaviours.

4.1. Surface-Based Interaction

The first observation is that by combining objects in two sub-graphs, implicit interactions would happen anyway in Unity: visually the objects would appear to interact and, more importantly, collision detection will occur. A behaviour in the Zone might anyway simply use searching through the scene graph by name, scene-graph path, component type or tag. Additionally geometric queries such as ray casting and bounding volume collisions would work. There are certain considerations that the programmer might need to be aware of (e.g. the Zone might not be a specific numbered child of the scene), but such issues should be easy to detect and warn about at the production stage.

We have called these types of interactions "surface-based interactions". This name is inspired by the scene-graph-as-bus experiment, where an abstract scene graph was used as the communication between two heterogeneous 3D applications [ZHC*00]. One application was able to interact with the other simply by finding appropriate surfaces in the shared scene graph. Surface-based interaction is sufficient for many behaviours within Zones (i.e. behaviours that are triggered and cause in-Zone behaviour such as autonomous object movement).

4.2. DeckMaster

The Zone will need to call functionality on the Deck. For example, triggering the loading of a new Zone. This can easily be achieved by having a singleton object *DeckMaster* that can be found by name or type. All scripts in the Zone can discover this; the Unity package provides a function to identify the DeckMaster that any script can use.

4.3. Message Passing

The Deck will need to be able to activate functionality on nodes in the Zone. This binding needs to be done at run-time and thus there needs to be a way of discovering necessary relationships between objects. There are various ways of achieving this. In the current version of Ythor we have opted for a lightweight publish-subscribe message passing system. This allows a script to register a callback based on certain types of event. Unity already provides callbacks, e.g. for collision events or key events, so the programming model should be familiar to developers.

Unlike the built-in Unity message-passing system, our callback interface declares the originator of the message, so that certain user interface functionality can be created by the object (e.g. creating a visual representation of a grab action,

by a visual tether between object and the hand, or other body part, that grabbed the object).

The mechanism is partly based on analysis of previous message-based systems, for example the DIVE system [FS98] or Distributed Inventor [HSFP99]. In the current version we have supported the following types of generic events that map to high-level interaction events: *Grasp*, *Drop*, *Select*, *Deselect*, *Travel* and *Stop*.

Each message type passes a success or fail as well as the source object. This would allow subsequent behaviours of nodes in the Zone to react to these events by, for example, changing their behaviour, applying constraints, highlighting, etc. Scripts can register for all events of a certain type, or only events that apply to a specific object.

A Deck is required to generate these messages and effect the associated interaction techniques. Both grasp and select are provided as grasp should be used to model grabbing and manipulation of objects, while select should be used to model touch without the intention to manipulate. It is up to the Deck how specifically to implement these. Similarly the Deck should provide a travel technique that respects collision volumes present in the Zone. The associated Travel and Stop events are used as hints to functionality that the Zone provides, such as audio effects. For complex effects (e.g. footsteps), the Zone might need closer observation of the avatar, but we consider that this already feasible under the surface-based interaction, if the script observes the avatar objects (see below).

We would note that arguably the feedback that might be generated on interactions should be generated by the Deck as the normal functioning of its interaction techniques. However, it is not clear how the negotiation of this might happen: Zones might have different visual styles, where certain interaction feedback might be inappropriate. The negotiation of this, and hints from the Zone to the Deck about what styles of feedback are allowed, is future work.

4.4. Avatar

The Deck provides an avatar in the scene. In the current prototype, it is not required that this include a visual representation. There should be a series of nodes that represent the positions of various limbs and end-effectors. These are primarily provided so that objects in the Zone can discover them, and thus interact with them. When the Deck effects certain interaction techniques, it sends the message with the avatar limb that is responsible (e.g. left hand, right hand, etc.). This allows feedback to be generated appropriately. Having a head node, nodes for the eyes and ears, etc. allows for other common scene behaviours such as avatar gaze at the user, sound effects that can be positioned to the sides of the users, etc.

The following nodes are required: head, left eye, right eye, pelvis, left foot, right foot, left hand, right hand, ground center (below the pelvis). The Yther website gives a detailed specification.

4.5. AirLock

Each Deck provides the core functionality for interacting with Zones. As noted, the Deck provides mechanisms for travel, object selection and object manipulation. Each Deck is free to implement these however they wish; see the examples in Section 5.1.

Each Deck will need to load a Zone on initialization. Our current examples load a test Zone that we have called an *AirLock*. The Airlock serves two purposes. First it provides a test ground for all the interaction techniques. If the Deck can operate the behaviours in the Airlock, then it should work on other Zones. The Airlock thus serves as a simple test for new Decks; the content for a standard Airlock is including in the Unity package.

5. Example Decks and Zones

5.1. Decks

Four Decks have been developed to test the examples. The version of Unity used was 5.0.3. The first is a Desktop client that is designed to be used with just a mouse for travel, selection and manipulation. This uses an on-screen widget to effect travel and mouse clicks to select and drag/manipulate objects.. The second is based on the Oculus Rift DK1 and DK2. This uses the standard Oculus SDK for Unity. Selection and manipulation is done by ray selection from the forehead. The third is a variant of the second that uses a VRPN server [Uni15b] to connect to a Polhemus Fastrak tracker. The Polhemus tracker provides hand tracking. Selection and manipulation is done by casting a ray from the hand. The MiddleVR software is used to connect to the VRPN server [i'm15]. The fourth is developed for a specific CAVE-like display. This system has three walls and a floor. The four projectors are driven by a custom-built high-end PC with an NVidia K5000 card. Tracking is provided by an Intersense IS-900 system, that is supported by a VRPN server [Uni15b]. The MiddleVR software is again used to interface to the VRPN server, and also to set up camera views with stereo rendering. Example views from these Decks are shown in Figure 2. All are shown running a Virtual Pit Zone.

5.2. Zones

As discussed in Section 4.5, the purpose of an AirLock is to be the default zone on loading a Deck. One simple example is shown in Figure 3, Top Left, where the AirLock is actually a model of the lab where one of the HMD systems is installed. Thus the idea is that the person sees the same environment when they don the HMD. This AirLock contains a table with two objects: one object spins when selected, the

Figure 2: *A Virtual Pit loaded on three different Decks. Top: screenshot of desktop client. Middle: screenshot of HMD client. Bottom: view of user inside the CAVE.*

other one changes color while it is being moved. This, and the ability to travel about the space act as a very simple test of the Deck. The door is a Portal to a hub Zone.

A simple example hub Zone is shown in Figure 3, Top Right. This model is based on a free model of Stonehenge downloaded from 3D Warehouse [Tri15]. Certain of the stone arches are Portals to other environments. For this hub, some of the Zones include the Virtual Pit shown in Figure 2, an experiment using an self-avatar, Figure 3 Bottom Left, and an "escape the room"-style puzzle which was done as student coursework Figure 3, Bottom Right.

Figure 3: *Further example Zones.*

6. Limitations and Discussion

6.1. Short-Term Development

Ythor is based on a simple set of abstractions and conventions. Because the Unity environment is so powerful, with lots of existing content and many strategies for implementation, the main current risk is that it will be perceived as being too hard, or too limiting to follow the Ythor conventions. We hope to combat this by porting, and encouraging porting of more complex demonstrations.

6.2. Zones

Currently we have a Zones and Portals abstraction to support large worlds. Portals are only an abstract concept and are not necessarily visible or door-shaped. Thus while they are a flexible mechanism the model is that the whole Zone is loaded before being inserted into the scene. Thus to support very large models, a fully streaming model may be needed.

6.3. Plugins

Virtual environment software is a very broad domain. Ythor was not designed with the requirements of application domains such as scientific visualization or application requirements such as real-time video texturing. While these might be addressed purely through scripting, some sort of native code extension is inevitable. While it may be possible in the future for Ythor to provide for a Zone to load native code, what would be more useful is a standard framework for plugin code extension.

7. Conclusions

The initial aim of Ythor was to facilitate content sharing between different virtual reality systems without having to resort to recompiling demonstrations. While our own lab had tried to standardize on various APIs and frameworks to support all the equipment, the key stumbling block was normally that the tools to create content were not very sophisticated or did not support a wide range of users. The design of Ythor thus started from a well-supported development environment with a large user community (Unity) and asked what was the minimal imposition of additional framework and constraints that could achieve this aim. The notions and implementation of Deck and Zone are flexible enough to support content sharing amongst a variety of platforms as shown in Section 5. In principle a much broader set is supportable by having developers build their own Decks.

Ythor is an experiment. The components will be made open source so that others in community can develop and contribute. However the challenge is to design a more flexible platform or to evolve Ythor into something that can fulfill the requirements of a broader set of virtual reality developers. Our main concern in developing the platform was

in sharing content between different systems within a laboratory, but the main use of Yther in the short term might well be games developers. Future analysis may indicate that Unity was not the right platform, though it has many positive properties. There are risks that Unity will change dramatically, invalidating previous content, or that the Deck paradigm provides too many constraints on developers of Zones. However, we believe that this is a useful debate to continue through exploring novel system designs.

References

[BDR04] BEHR J., DÄHNE P., ROTH M.: Utilizing x3d for immersive environments. In *Proceedings of the Ninth International Conference on 3D Web Technology* (New York, NY, USA, 2004), Web3D '04, ACM, pp. 71–78. 2

[BEJZ09] BEHR J., ESCHLER P., JUNG Y., ZÖLLNER M.: X3dom: a dom-based html5/x3d integration model. In *Proceedings of the 14th International Conference on 3D Web Technology* (2009), ACM, pp. 127–135. 2

[BJH*01] BIERBAUM A., JUST C., HARTLING P., MEINERT K., BAKER A., CRUZ-NEIRA C.: Vr juggler: A virtual platform for virtual reality application development. In *Virtual Reality, 2001. Proceedings. IEEE* (2001), IEEE, pp. 89–96. 2

[BWA96] BARRUS J., WATERS R., ANDERSON D.: Locales and beacons: efficient and precise support for large multi-user virtual environments. In *Virtual Reality Annual International Symposium, 1996., Proceedings of the IEEE 1996* (Mar 1996), pp. 204–213, 268. 3

[CMBZ00] CAPPS M., MCGREGOR D., BRUTZMAN D., ZYDA M.: Npsnet-v. a new beginning for dynamically extensible virtual environments. *Computer Graphics and Applications, IEEE 20*, 5 (2000), 12–15. 2

[CSW09] CRAIG A. B., SHERMAN W. R., WILL J. D.: *Developing virtual reality applications: Foundations of effective design*. Morgan Kaufmann, 2009. 2

[FMJ*05] FIGUEROA P., MEDINA O., JIMÉNEZ R., MARTÍNEZ J., ALBARRACÍN C.: Extensions for interactivity and retargeting in x3d. In *Proceedings of the Tenth International Conference on 3D Web Technology* (New York, NY, USA, 2005), Web3D '05, ACM, pp. 103–110. 2

[Fra15] FRAUNHOFER IGD: Instant reality release 2.4.0. http://www.instantreality.org/, 2015 (accessed July 1, 2015). 2

[FS98] FRÉCON E., STENIUS M.: Dive: A scaleable network architecture for distributed virtual environments. *Distributed Systems Engineering 5*, 3 (1998), 91. 2, 3, 6

[HSFP99] HESINA G., SCHMALSTIEG D., FURHMANN A., PURGATHOFER W.: Distributed open inventor: a practical approach to distributed 3d graphics. In *Proceedings of the ACM symposium on Virtual reality software and technology* (1999), ACM, pp. 74–81. 6

[i'm15] I'M IN VR: Middlevr. http://www.imin-vr.com/middlevr/, 2015 (accessed July 1, 2015). 6

[KASK02] KELSO J., ARSENAULT L., SATTERFIELD S., KRIZ R.: Diverse: a framework for building extensible and reconfigurable device independent virtual environments. In *Virtual Reality, 2002. Proceedings. IEEE* (2002), pp. 183–190. 2

[Khr15a] KHRONOS GROUP: Collada. https://collada.org/, 2015 (accessed July 1, 2015). 2

[Khr15b] KHRONOS GROUP: glTF. https://github.com/KhronosGroup/glTF, 2015 (accessed July 1, 2015). 2

[McC15] MCCRAE J.: Janus vr. http://www.dgp.toronto.edu/~mccrae/projects/firebox/site.html, 2015 (accessed July 1, 2015). 2

[Net15] NETEASE, INC.: Unitysocketio. https://github.com/NetEase/UnitySocketIO, 2015 (accessed July 1, 2015).

[Ocu15] OCULUS VR INC.: Oculus sdk v0.4.2 beta unity integration. https://developer.oculusvr.com/?action=dl, 2015 (accessed July 1, 2015). 3

[OJP*09] OLIVEIRA M., JORDAN J., PEREIRA J., JORGE J., STEED A.: Analysis domain model for shared virtual environments. *The International Journal of Virtual Reality 8*, 4 (2009), 1–30. 3

[Ope15] OPEN WONDERLAND FOUNDATION: Open wonderland. http://openwonderland.org/, 2015 (accessed July 1, 2015). 2

[OSV15] OSVR: Open source virtual reality. http://osvr.com/software.html, 2015 (accessed July 1, 2015). 2

[PADD03] PAPE D., ANSTEY J., DOLINSKY M., DAMBIK E. J.: Ygdrasil - a framework for composing shared virtual worlds. *Future Generation Computer Systems 19*, 6 (2003), 1041–1049. 3

[SKR*10] SONS K., KLEIN F., RUBINSTEIN D., BYELOZYOROV S., SLUSALLEK P.: Xml3d: interactive 3d graphics for the web. In *Proceedings of the 15th International Conference on Web 3D Technology* (2010), ACM, pp. 175–184. 2

[SO09] STEED A., OLIVEIRA M. F.: *Networked Graphics: Building Networked Games and Virtual Environments*. Elsevier, 2009. 3

[Sow00] SOWIZRAL H.: Scene graphs in the new millennium. *Computer Graphics and Applications, IEEE 20*, 1 (2000), 56–57. 2

[SZ99] SINGHAL S., ZYDA M.: *Networked Virtual Environments: Design and Implementation*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999. 3

[Tri15] TRIMBLE NAVIATION LTD.: 3d warehouse. https://3dwarehouse.sketchup.com/, 2015 (accessed July 1, 2015). 7

[Uni15a] UNITY TECHNOLOGIES: Unity. http://unity3d.com/, 2015 (accessed July 1, 2015). 3

[Uni15b] UNIVERSITY OF NORTH CAROLINA AT CHAPEL HILL: Virtual reality peripheral network. http://www.cs.unc.edu/Research/vrpn/, 2015 (accessed July 1, 2015). 2, 6

[Web15] WEB3D CONSORTIUM: X3d architecture and base components v3.3, iso/iec is 19775-1:2013. http://www.web3d.org/content/x3d-v33-abstract-specification, 2013 (accessed July 1, 2015). 2

[Wor15] WORLD VIZ: Vizard virtual reality software toolkit. http://www.worldviz.com/products/vizard, 2015 (accessed July 1, 2015). 2

[ZHC*00] ZELEZNIK B., HOLDEN L., CAPPS M., ABRAMS H., MILLER T.: Scene-graph-as-bus: Collaboration between heterogeneous stand-alone 3-d graphical applications. In *Computer Graphics Forum* (2000), vol. 19, Wiley Online Library, pp. 91–98. 5