Virtual Reality Capabilities of Graphics Engines

Edward Peek
University of Auckland, NZ
epee004@aucklanduni.ac.nz

Burkhard Wünsche
University of Auckland, NZ
burkhard@cs.auckland.ac.nz

Christof Lutteroth
University of Auckland, NZ
lutteroth@cs.auckland.ac.nz

ABSTRACT

Desktop virtual reality has traditionally been the dominant display technology for consumer-level 3D computer graphics. Recently more sophisticated technologies such as stereoscopy and head-mounted displays have become more widely available. However, most 3D software is still only designed to support desktop VR, and must be modified to both technically support these displays and also to follow the best practises for their use. In this paper we evaluate modern 3D game/graphics engines and identify the degree to which they accommodate output to different types of affordable VR displays. We show that stereoscopy is widely supported, either natively or through existing adaptions. Other VR technologies such as head-mounted displays, head-coupled perspective (and consequentially fish-tank VR) are rarely natively supported. However, we identify and describe some methods, such as re-engineering, by which support for these display technologies can be added.

Keywords: virtual reality, graphics engine, head-coupled perspective, head-mounted display, stereoscopy

1 INTRODUCTION

A wide range of computer applications employ virtual reality (VR) concepts, including the general consumer applications that involve some sort of 3D virtual environment. Common examples of such applications are 3D modelling, computer aided design (CAD), video games, data visualisation, television and movies.

Recent commercial advances in consumer-level VR have lead to certain types of VR technology becoming cheap and of high enough quality to begin displacing the entrenched traditional technologies. Some examples of new devices that employ these novel VR technologies include haptic input methods such as Nintendo Wii Remote, Microsoft Kinect and Leap Motion Controller; head-mounted displays such as the Sony Personal 3D Viewer and Oculus Rift; and stereoscopic television sets, computer displays and projectors of which there are too many to name.

While attention and interest towards these technologies is slowly growing, support for them by VR applications is still limited. In the case of haptic inputs this is understandable since implementing natural user interfaces is a substantial departure from mouse/keyboard/controller based input systems. On the other hand, support for new VR display technologies is much less invasive and in some instances can even be achieved with no modification to the original software [10].

WSCG 2013 conference proceedings, ISBN xx-xxxxxx-x-x
WSCG'2013, June 24–27, 2013
Plzen, Czech Republic.

This work presents an investigation into modern software applications with the objective of determining what types of new VR display (not input) technologies are supported by these applications. We specifically look at graphics engines: reusable software components which handle output to VR displays and are shared by many applications. This allows a large number of applications to be covered with only the need to evaluate a few specific graphics engines. The following research questions embodies the objective of this study.

How far do modern graphics engines support consumer-level VR display technologies? How easily can support be added where they do not?

In answering these questions, we also make the following contributions.

- To provide a resource useful for determining which graphics engines are suitable for future application development and research in virtual reality.
- To identify common practises, shortcuts and interaction methods in engine design that makes them, in their current state, unsuitable for VR.
- To determine a general sense of how much attention is being paid to VR issues in consumer graphics engines.

In this paper we first give some background information about graphics engines and VR display technologies in Section 2, and describe some related work in Section 3. We then describe our methodology to evaluating the graphics engines in Section 4 and discuss our results in Section 5.

2 BACKGROUND

Graphics and Game Engines

A graphics engine is a reusable software component designed to render a 3D virtual environment. Graph-

ics engines can be distributed as standalone pieces of software or as part of larger systems, notably, but not limited to, game engines. This involves taking the current state of the simulated environment as input and rendering an image based on the lighting and shading model of the simulation. Real-time graphics engines are those that are capable of performing this process quickly enough to appear seamless to a user (typically around 30–60 rendered frames per second). real-time engines allow the simulation to be interactive and react to inputs from human users; a requirement of VR systems. In order to achieve real-time speeds, graphics engines normally delegate rendering to dedicated hardware and use algorithms and models that favour fast computation over physical accuracy.

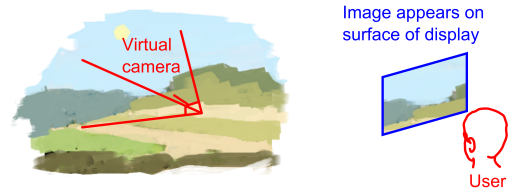
VR Display Technologies

Virtual reality display technologies (also known as 3D displays) are the VR technologies that specifically deal with *visually presenting* a virtual environment to its user. These are used in addition to other VR technologies such as input systems and audio output, as well as the software that simulates the virtual environment. Within the context of this research, we do not consider the graphical rendering algorithms (such as raserising polygons, lighting, shading and post-processing) to be part of a VR display technology, but rather part of the simulation logic. In this sense a VR display technology is only the hardware and software that *requests* graphical views from the environment simulation and *presents* them to the user.

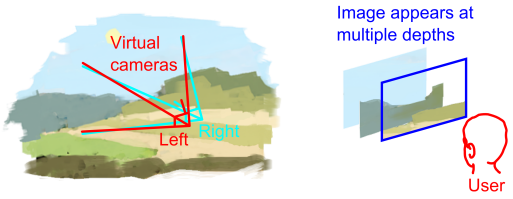
Over time many different display technologies have been developed to satisfy this role. Nearly all of these operate on some variant of a camera metaphor; i.e. a virtual pinhole camera exists in the environment and regularly takes 2D snapshots which are then displayed on a physical display surface (such as a computer monitor). The components that make up such a display technology are the software that models the virtual camera, the hardware that displays images taken by the virtual camera, and the software interface that passes these images in the correct format to the display hardware.

There are several systems [4, 11] for classifying different VR display technologies based on different properties and generalisations. We utilise an alternative system that is based on software implementation requirements. In this paper we focus on consumer-level VR display technologies; specifically *desktop VR*, *stereoscopy*, *head-coupled perspective* and *head-mounted displays*.

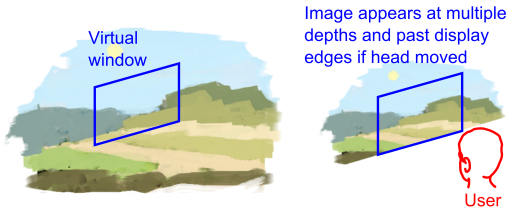
The display properties most important to this study are how they are interfaced with from software, and how the rendering pipeline must be adapted to correctly reflect their perception model. What follows is a brief description of each of these display technologies, the intent of which is to define the specific implementation requirements we use for this study.



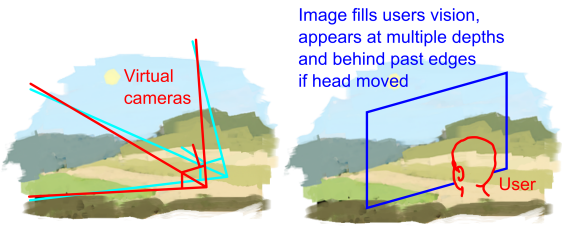
(a) Desktop VR



(b) Stereoscopy



(c) Head-coupled perspective



(d) Head-mounted display

Figure 1: Depictions of differences between the VR display technologies in their simulation models and user's perception.

Desktop VR has been the dominant form of presenting 3D virtual environments to their users since the advent of computer graphics. Desktop VR operates on a pinhole camera model, with a virtual camera controlled entirely by the simulation and a display capable of showing only a single image from this camera at a time. As the simplest form of VR it avoids many issues such as eye strain, increased computation cost and poor image quality that have hampered the use of more sophisticated technologies.

Because desktop VR is ubiquitously supported as the default output mode of virtually every graphics engine available today, we don't discuss it any further in this paper.

Stereoscopy is an extension of the desktop VR paradigm adapted for binocular vision. Stereoscopy achieves this by rendering the scene twice, once for each eye, then encoding and filtering the images in such a way that each image is seen by only one of the users' eyes. This filtering is most easily achieved through special eye glasses, the lenses of which are designed to selectively pass one of the two encodings produced by the matching display. Current methods of encoding are by colour spectrum, polarisation, temporally or spatially. These encoding methods are frequently categorised as *passive*, *active* or *autostereoscopic*. The difference between passive and active encoding is determined by whether or not the glasses are electrically actively or not: passive encoding systems are therefore colour and polarisation while the only active encoding is temporal. Autostereoscopic displays are those that do not require glasses because they encode spatially, meaning that the physical distance between the eyes is sufficient to filter the images.

Consumer stereoscopic displays interface with computers in the same way as desktop VR displays (via video interfaces such as VGA or DVI). Since most of these interfaces do not have special modes for stereoscopy, the two stereo images are packed into a single image in a format recognised by the display hardware. Such *frame packing formats* include *interlaced*, *above-below*, *side-by-side*, *2D+depth* and *interleaved*.

Because these standarised interfaces are how the software passes rendered images to the display hardware, software applications are not required to know or adapt to the encoding system of the display hardware. Instead, all that is required for stereoscopy to be supported by a graphics engine is that it is able to render two images of the same simulation state from different virtual camera positions and combine them in a frame packing format supported by the display.

Head-coupled perspective (HCP) operates on a slightly different principle than desktop VR and stereoscopy. A virtual window is defined instead of a virtual camera, with the boundary of the virtual window mapped to the edges of the user's display. Thus, the image on the display depends on the relative position of the user's head, as objects from the virtual environment are projected onto the display in the direction of the user's eyes. This projection can be done using a off-axis version of the projection mathematics used in desktop VR.

In order to do this, the position of the users head relative to the display must be tracked accurately in real-time. Tracking systems that have been used for this purpose include armatures [19], electromagnetic/ultrasound trackers [18] and image-based tracking [12]. A limitation of HCP is that since the displayed image depends on the position of a user, any other users looking at the same display will perceive a distorted image since they will not be viewing from the correct position.

Head-mounted displays are another type of single-user VR technology. HMDs combine the enhancements of stereoscopy with a large field-of-view and head-coupling similar to HCP. The perceptual model behind HMDs is to completely override the visual input to the users eyes and replace it with an encompassing view of the virtual environment. This is accomplished by mounting one or two small displays very close in front of the user's eyes with a lens system to allow for more natural focus. Since the displays are so close to the user's eyes, any part a display is only visible to one eye, making the system autostereoscopic.

An orientation tracker is also embedded in the head-gear, allowing for rotation of the user's head to be tracked. This allows the user to look around the virtual environment using natural head motion by binding the orientation of the virtual camera to the orientation of the user's head. This differs from HCP where it is the position, not orientation, that is tracked.

The software requirements to support HMDs are the same as stereoscopy, with the additional requirements that the orientation of the HMD must be considered by the graphics engine, as well as any distortion caused by the lens system to be corrected for.

In addition to these four technologies, there are numerous other types of VR displays that we do not adderess in this study. *Fish-tank VR* is not discussed because it is simply a combination of head-coupled perspective and stereoscopy. Furthermore, we do not consider more sophisticated VR technologies such as *multi-view displays*, *gaze-dependent depth of field*, *volumetric displays*, and *cave automatic virtual environments (CAVEs)* as they do not match our image of *consumer-level*. This is largely due to them being significantly more expensive (upwards of $1000 USD), difficult to construct from off-the-shelf components or impractical to set up in many environments (CAVEs are an example of this).

3 RELATED WORK

General purpose graphics/game engines and virtual reality research are intrinsically linked, sharing several common goals. Both are highly dependent on realistic real-time 3D graphics and simulations, and both aim to generate a high degree of immersion and engagement. Because of this game engines provide many features that make them useful tools in scientific VR research. Correspondingly, advances in VR research often end up in graphics engines when they prove to be useful enhancements.

Lewis and Jacobson [8] explore the use of game engines for scientific simulation. The networking, graph-

ical and 3D scene management capabilities of the engines are noted as factors that make them useful for the variety of sample research applications they have been used for. Two of the engines mentioned in this article — the id Tech engine and the Unreal Engine — are investigated in our research, albeit using more recent versions. The authors do note however that for applications that require more sophisticated forms of VR, the base capabilities of the engines in question are not sufficient.

A more recent report by Trenholme and Smith [16] specifically evaluated common game engines for first-person virtual environments, building upon the work of Lewis and Jacobson. This work provides generic descriptions of the advantages and disadvantages of 6 reasonably modern (1–2 major versions behind what is current now) game engines for use in simulating virtual environments. However, this comparison does not consider the engines from a VR standpoint, so it misses out on recent trends. In addition to this, the capabilities of game engines advance at an extremely rapid pace and comparisons between previous generation technologies are not accurate for the current state of the art.

Where the capabilities of an engine are not sufficient for it to be used as-is for VR applications, but close enough to make it desirable, adaptions can be made to the engine to allow for its use. Lugrin et al. [9] describe how the Unreal Engine 3 (again included in our research) can be adapted to support rendering in a CAVE system and accept input from a 3D tracked wand held by the user. This adaption was implemented as C++ plug-ins to incorporate the different forms of head and wand tracking, split across 6 networked clients to render the different sides of the CAVE with NVIDIA 3D vision to provide stereoscopy. Similar adaptions have been make to other engines to support more sophisticated VR such as with the Unity Engine and CryENGINE.

As well as game engines contributing to VR research, benefits also flow in the opposite direction, I.E. some VR technologies originally used for research have now become available in game engines. Litwiller and LaViola [6] discuss the implications of one such technology (stereoscopy) for gaming. They find that while there is no actual or perceived performance difference of the users' game scores when using stereoscopic 3D, the users did express a preference towards using stereoscopy over desktop VR. Sko and Gardner [14] investigate different technologies through implementing various uses of head tracking in games, while Andersen et al. [1] combine stereoscopy and head-coupled perspective (called fish-tank VR) in a first-person shooter game.

Despite the wealth of research into implementing VR with game engines, there is little general information on how well game engines support VR. This may be a result of the very specialised nature of many VR research projects, and the tendency to focus on a single graphics engine or VR technology. By contrast, we discuss how far several current graphics engines can go to support various VR display technologies.

4 METHODOLOGY

Given enough time and effort, any graphics engine can be made to support almost any VR display technology. Different methods are available to do this, with a different amount of intrusiveness needed depending on how the software is designed and constructed.

Because measuring the amount of effort required to implement VR in a graphics engine is a difficult and inexact task, we have instead determined the *level of suport* each graphics engine has for each of the VR display technologies. Additionally, quality factors are considered where applicable, as well as several generic properties of the engines that influence the implementation of these technologies.

Level of Support

With the flexibility of modern graphics engines it is not particularly meaningful to note features (particularly VR support) as *supported* or *not-supported*, since almost any feature can be made supported with reasonable effort. The addition of such non-native features is either facilitated through extension mechanisms built into the engine itself, built into the platform the engine runs on, or by re-engineering either of these two components. Some of the most common extension mechanisms built into graphics engines are node graphs, scripting, plug-ins and source modification.

In addition to these built-in extension mechanisms, it is also possible to add or modify functionality via re-engineering. This is required when the built-in extension points do not provide enough flexibility to implement the desired functionality. Re-engineering involves modifying the behaviour of a program by overriding portions of a program's original code or by replacing linked code libraries with modified variants. This will be described in detail along with the other extension mechanisms at the end of this section.

Level of support is measured by determining which extension mechanisms can be used to implement a desired VR display technology. Extension mechanisms with negligible differences have been combined (such as scripting and plug-ins), with two additional levels introduced for no extension needed (native support) and no in-engine support possible (re-engineering). Extension mechanisms are ordered by the proportion of engine code relative to non-engine code that implements

the VR support. The resulting levels of support and their ordering follows.

5. Natively supported
4. Via in-engine graphical customisation (including node graphs)
3. Via in-engine coding (scripting or plug-ins)
2. Via engine source code modification
1. Via re-engineering

This helps to answer our major research question and gives a sense of *engine support* and *engine flexibility* where high values indicate good VR support or flexibility, and low levels indicate poor VR support and low flexibility. It is important to note that this ordering is not a measure of the effort required to implement VR, but rather a measure of how well the engine assists this task.

We only report the highest level of support attained, as subsequently lower levels are practically always supported as well. In addition to presenting the highest level of support for each VR technology, we also indicate where third parties have demonstrated working implementations of the technology.

A brief description of each level of support follows.

Native In engines that natively support a VR technology, the developers of the engine have intentionally written the rendering pipeline in such a way that minimal effort is required by the user to enable VR rendering. All that is required is to check an option in the developer tools or set a variable in the engine's scripting environment. In addition to easily enabling the technology, the engines are also designed to avoid common optimisations and shortcuts that are not noticeable with desktop VR displays, but become noticeable with more sophisticated technologies. A common example of this is rendering objects with correct occlusion but at an incorrect depth [5], which causes depth cue conflicts under stereoscopy.

Graphical customisation Some engines are designed in such a way that the rendering process can be altered using custom tools with a graphical interface. One approach to this is via node graphs, where different components of the rendering pipeline can be rearranged, modified and reconnected in multiple configurations. Depending on what types of nodes are supported, it is sometimes possible to configure the nodes in such a way as to produce the effect of certain VR technologies. An example is shown in Figure 2, which depicts the Unreal Engine's material editing interface configured to render red-cyan anaglyph stereo as a post-processing effect.

Engine coding Practically every engine can be extended with custom code, using well-defined, but restricted, extension points. The two common forms of this are scripting, where the engine runs small programs/scripts in a restricted environment, and plug-ins, where the engine loads and runs externally compiled code. Both forms have access to a subset of engine features; however, plug-ins also have access to external APIs while scripts do not. Since this is the mechanism through which application-specific functionality is normally implemented, the engine features available to the custom code may be targeted more towards artificial intelligence, game logic and event sequencing, rather than controlling the exact rendering process.

Engine source code modification In addition to free open-source engines, some commercial engines make their complete source code available to users with the appropriate licence agreement. With access to the full source code any VR technology can be implemented, although the amount of modification required could be significant.

Re-engineering For engines that do not provide any of the above entry points for customisation, some amount of change is still possible through re-engineering. Re-engineering is a form of reverse-engineering where in addition to learning some of the workings of the program, some of its functionality is modified as well. The effort needed to fully reverse-engineer a rendering pipeline can be significant, so more minimally invasive forms of re-engineering are preferable. One of these approaches is function hooking, which is where the invocation of an internal or library function is intercepted and replaced with custom behaviour. Since a very large fraction of real-time graphics engines use the OpenGL or Direct3D libraries for hardware graphics acceleration, these libraries make reliable entry points for implementing visual-only VR technologies through function hooking. This approach has proved to be effective for adding stereoscopy to 3D games [10, 17]. We have also shown that it is also possible to implement head-coupled perspective in this manner [**?**], by hooking the OpenGL functions that load projection matrices (glFrustum and glLoadMatrix) and replacing the fixed-perspective matrices provided by the original program with head-coupled matrices.

Display Technology Support Criteria

For an engine to be labelled as supporting a specific VR display technology group, it must be able to satisfy the technical requirements of at least one actual display technology in that group (e.g. support for anaglyph stereoscopy indicates general stereoscopy support). Support can be achieved at any of the levels described previously, in which case all the technical requirements of the display technology must be implemented at that

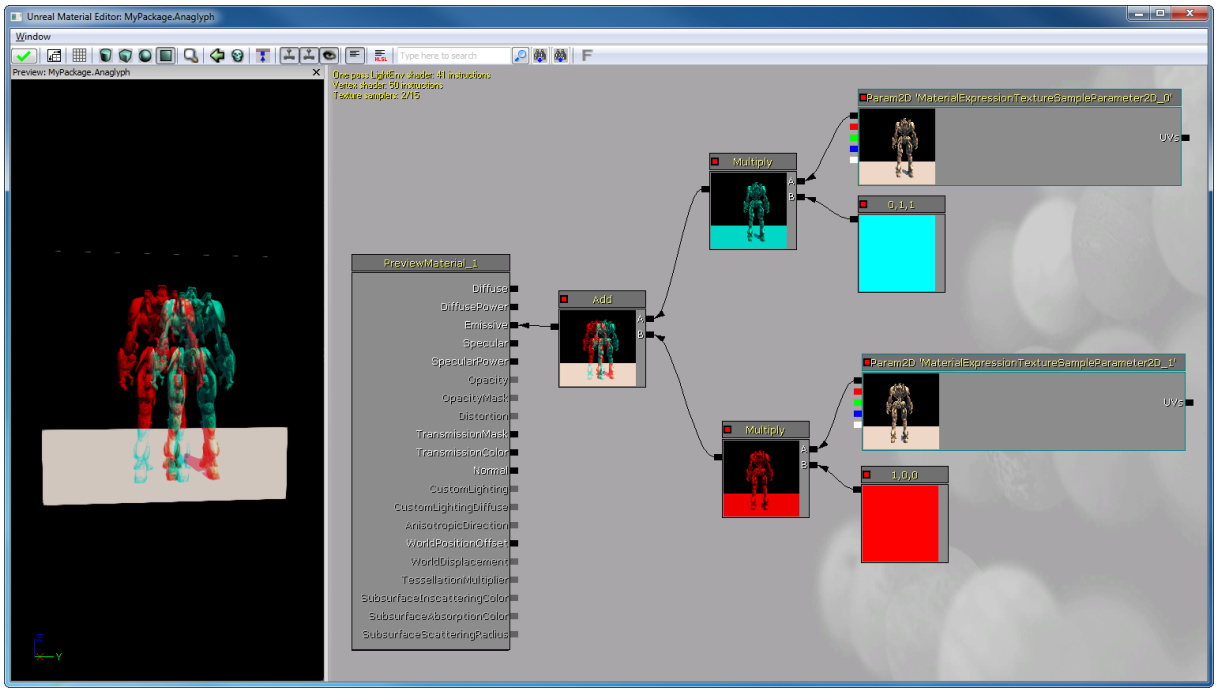


Figure 2: Configuration of the Unreal Engine to support red-cyan anaglyph stereoscopy, using the Material Editor. Adapted from [3]. Other stereo encodings can be supported in this manner, E.G. by interlacing the images for polarised stereo displays.

level or higher. The technical requirements of each display technology are the same as those outlined in Section 2.

VR Quality Factors

In addition to the technical challenge of implementing the VR display technologies just discussed, there are many secondary quality factors that affect a user's perceived quality of the VR experience. These factors arise because the implementations of the display technologies can not perfectly replicate the physical phenomenon they model. Since the differences are usually subtle, the user is frequently not consciously aware of them, but may instead experience some amount of eye strain, headaches or nausea. There can also be many different ways to implement any particular display technology, each of which balances different quality factors with other factors such as implementation cost. A prime example of this is stereoscopy, where at least ten different mechanisms to split images between the eyes have been used recently.

While quality factors are most inherently linked to the display hardware, appropriate software design can mitigate these issues, while careless design can introduce new issues. Because this study deals with the software implementation of VR display technologies, these software issue are of interest to us.

Examples of hardware quality factors that can be mitigated through software are crosstalk (stereoscopy), A/C breakdown (stereoscopy) and tracking latency (HCP and HMDs). Since these factors are well established for their respective display technologies, there are well-known techniques to minimise issues they cause. The solutions are respectively reducing scene contrast, reducing parallax and minimising rendering delays. In most cases the engines this paper evaluates have non-native support for the display technologies associated with these quality factors, and subsequently do not follow these practices.

Incorrect software implementations can also influence the quality of the VR effect, which can occur due to carelessness, or as a result of optimisation for desktop VR. An example of this is special layers (such as the sky, shadows and first person player's body) at arbitrary depths in different passes. While this produces correct occlusion in desktop VR, the addition of the binocular parallax cue under stereoscopy reveals the incorrect depth, and creates a conflict between these two depth cues. This is not an uncommon issue due to the dominant nature of desktop VR, and serves as another example of where a naive third party implementation may not be as good as native VR support.

From these points it should be noted that while non-native VR implementations might meet the necessary technical requirements, other factors must be taken into account as well. Where possible we have pointed out these quality issues, but due to their dependence on a specific implementation and application it is difficult to make generalisations for a single graphics engine.

General Engine Properties

In addition to VR capabilities, this paper also outlines several general properties of graphics engines. These properties are chosen to assist researchers and developers in the selection of engines, to help identify trends of VR support and to classify the engines. What follows is a list of the general properties we considered useful. We do not elaborate on these properties as, being so general, they are largely self-descriptive.

- Developer interface
- Licences
- Programming languages
- Target platforms
- Version evaluated

Graphics Engines

The graphics engines of interest to us are those that are currently being used to render real-time 3D environments for research, commercial and other applications, and will likely continue to be used in the near future. We selected a representative sample of the most popular engines for this evaluation. The total number of graphics engines is greatly inflated by the number of graphics engines that are custom built for a select few applications. A secondary limiting factor is access to engines, as many are not made available to 3rd-party developers, only made available to established companies, or have prohibitively high licencing costs (in the order of $100k+ USD). This has effectively restricted our investigation to graphics engines that are open-source or have free versions available with restricted access. Fortunately many normally expensive engines provide such versions, and so we are still able to cover a good range.

In addition to these restrictions, investigation of specific engines that are available to us have been prioritised according to the following factors.

- Engines should be in active development.
- An engine should have good community support, and be used in several applications.
- An engine should additionally have been considered in previous VR research.
- Engines designed for gaming should also have been used in non-gaming applications.
- Engines should focus on realistic and immersive graphics, and cutting edge technology.

The engines we evaluated can be put into 4 groups based on their licencing model, which also serves as a reasonably good overview of the general types of engine available.

Premium commercial engines (CryENGINE and Unreal Engine) are the most expensive and have the most comprehensive set of features. These are targeted towards large development studios that can afford the very high licencing costs to use the engine. These engines provide graphical tools to allow artists and game designers to use, while also allowing modification and extension of their source to implement application-specific behaviour. A recent trend has been for free versions of these engines to be released with specific restrictions, notably no source-code access and for non-commercial use only.

Commercial engines (Unity) are similar to premium engines but at significantly lower costs. They typically have slightly smaller feature sets or be intentionally simple and lightweight. Their main target audience is smaller (particularly indie) studios, individuals and hobbyists. Like premium engines, they typically provide graphical development interfaces to allow non-technical users to use them.

Previously commercial engines (Torque3D) are commercial engines that have at some point been made open-source. Reasons for this might be because newer versions of the same engine are now sold commercially, alternative revenue sources are being followed, because the engine is no longer competitive or to attract a larger user-base.

Open-source (OGRE and Irrlicht) are engines that are available for free under open-source licencing. They are frequently community developed, but sometimes also have backing by a commercial organisation. The quality and feature-sets of these engines varies dramatically, but usually falls short of commercial engines. These engines are typically fully code based, and do not provide graphical tools for development.

In addition to the engine categories included in this study, another major one is proprietary engines. These are those engines developed in-house for a specific application. None of these engines are included in this evaluation because they, by very nature, are not made available to third parties for development.

5 RESULTS AND DISCUSSION

The results of our evaluation can be found in Tables 1 and 2 with a discussion to follow.

The most obvious result from this evaluation is that almost none of the graphics engines evaluated support

VR technology	Stereoscopy	Head-coupled perspective	Head-mounted display
CryENGINE	**5: Native** [10] Support for both dual rendering and retargeting. Supports both manual and GPU driver frame packing.	**3: Coding** Access to camera matrices through C++ interface. C++ sufficient to access any head tracking method.	**3: Coding** [2] Stereoscopy supported natively, orientation tracking can be accessed via C++ plug-in.
OGRE	**3: Coding** [7, 10] OGRE rendering can be fully controlled and customised via the C++ interface, allowing implementation of all three display technologies.		
UDK	**4: Graphical customisation** [3, 10] Dual camera rig can be created using Unreal Kismet and outputs packed using the material editor.	**1: Re-engineering*** No access to custom camera projection from engine so re-engineering is needed if your licence does not include source code access.	**3: Coding** Stereoscopy through custom implementation, head orientation can be obtained via a custom DLL and bound to camera via script.
Unity	**3: Coding** [15] Dual cameras can be created and control via script, images can be packed as post-processing filter.	**3: Coding** Scripting supports custom camera projection matrices. Tracked head position can be obtained via C++ plug-in.	**3: Coding** Stereoscopy through custom implementation, head orientation can be obtained via C++ plug-in.
Irrlicht	**3: Coding** [10] Irrlicht rendering can be fully controlled and customised via the C++ interface, allowing implementation of all three display technologies.		
Torque3D	**3: Coding** [10] Multiple passes of rendering are supported. This can be used to create the dual views and pack them in a compatible format.	**2: Source modification** Scripting interface to camera does not support off-axis projections, camera projection generation must be modified in code.	**3: Coding** [20] Head orientation can be accessed from an external tracker over TCP. Camera orientation can be updated based on this via script.

Table 1: Graphics engines' levels of support for various VR display technologies. *depends on licence

Name and Version	Interface	Licence	Code language	Platforms
CryENGINE 3.4.4	GUI Framework	Free for non-commercial use, Licence required for commercial use or source code access	C++ Lua	PC Games console
OGRE 1.8.1	Library	Open-source (MIT)	C++ Material scripts	PC Smartphone
UDK 2013/02b	GUI	Free for non-commercial use, Licence required for commercial use or source code access	C++ UnrealScript	PC Games console Smartphone
Unity 4.0.1f2	GUI	Free limited version Flat fee pro version Source code access via special licence	C# JavaScript	PC Games console Smartphone
Irrlicht 1.8	Library	Open-source (zlib)	C++	PC
Torque3D 2.0	GUI, Framework	Open-source (MIT)	TorqueScript, C++	PC

Table 2: General properties of graphics engines

a non-traditional VR display technology. The only engine that does is the CryENGINE, which natively supports stereoscopy in most of the formats used by modern stereoscopic displays. There are two explanations for this deficit. Firstly, that the developers of the engines do not believe these display technologies warrant the extra effort needed to support them. Or secondly, that they believe that the 3rd party support is good enough that native support is not necessary. It is our belief that the second point is the more likely, since all engines support stereoscopy through several 3rd party programs including NVIDIA 3D Vision.

In terms of how well the engines are designed to accommodate 3rd party VR support, most rate very highly with all but two instances having levels of support at *level 3: coding* or better. The two instances of lower support occurred when the scripting system did not provide enough control over the camera parameters. It is unknown whether the lack of access is intentional because the underlying rendering systems do not support arbitrary camera properties, or whether they were seen as unnecessary, not useful or just not thought considered.

In some cases the engine extension mechanisms do not have enough functionality to host the entire VR technology, but do provide communication functionality so that part of the technology can be offloaded to a separate process. This occurs when the scripting interface can't access the HMD or HCP head tracking values directly, but can indirectly over local TCP or UDP. Native code (e.g. C and C++) is normally needed to access the head tracking hardware. An example of this is Torque3D which does not provide any access to native code at levels of support above *level 2: source code modification*.

Of the three display technologies considered, HCP is the only for which we could not find any examples of 3rd-party implementations. Potential explanations might be that this is a less well-known technique, that it is a predominantly software technique and so is less easily commercialised, or more likely because it does not provide as good an effect as the other VR technologies.

The core point to take away from this work is that while the majority of graphics engines *do not* support most VR display technologies natively, they *almost always* provide enough flexibility such that support can be manually added.

6 CONCLUSIONS

We have described the mechanisms by which modern graphics and game engines may be extended to support non-traditional display technologies, particularly stereoscopy, head-coupled perspective and head-mounted displays. Where these engines do not have built-in extension mechanisms, or the ones that are provided are too limited, these display technologies can always be implemented through re-engineering the engine.

Most of the engines evaluated do not provide native support for any non-traditional display technologies, and stereoscopy is the only technology that has any amount of native support in current versions of these engines. However several engines have support for head-mounted displays planned for future versions.

In the many instances where an engine does not provide native support for a display technology, support can usually be attained by developing a script or plug-in to produce the effect. Often this has been proved possible by other researchers or developers, and in many cases the source for the implementation is publicly available.

7 FUTURE WORK

As previously discussed, we believe the reason that most engines do not support most of the VR technologies evaluated is that there are still too few commercial displays that use them. As more exemplar displays become available this should start to change, and this can already be seen with several game engine developers (Torque3D, UDK and Unity) announcing support for HMDs (specifically the Oculus Rift) in future versions. It will be interesting to see whether support for specific technologies such as this will bleed through to other technologies as VR sophistication becomes a more important feature.

We have also considered a very small subset of the available classes of VR display technologies. Extending this evaluation to other technologies such as CAVEs, volumetric displays, multi-view displays and gaze-dependent field of view will increase the number of applications that benefit and also expose how engines can be adapted to cope with technologies substantially different from desktop VR.

In a similar vein, we have only evaluated 6 graphics engines which represents a tiny fraction of the entire population. Our preference towards selecting high speed real-time engines that have already been used for VR applications also means we did not consider any graphics engines used for applications such as CAD or scientific visualisation, which often have pseudo-real-time engines (in the sense that they react reasonably quickly to input, but not seamlessly).

We have also only considered the display side of VR, and ignored input technologies. While in many cases this can be done with little consequence, dependencies between the two have been known to cause problems. For instance mouse pointing depends on the virtual cameras projection properties which breaks down when there are multiple projections, as with stereoscopy, or the projection changes continuously, as with the tracking from HCP and HMDs. More work is needed to determine ways in which such input systems can be

accommodated for when using these display technologies.

REFERENCES

[1] A.S. Andersen, J. Holst, and S.E. Vestergaard. The implementation of fish tank virtual reality in games.

[2] Nathan Andrews. Crysis vr - head and gun tracking mod for the oculus rift, February 2013. URL http://www.youtube.com/watch?v=TJx21yuCi7E.

[3] Christopher Berry. How to make a stereoscopic camera rig for udk, July 2011. URL http://www.thebeardedberry.com/How%20To%20Make%20A%20Stereoscopic%20Camera%20Rig%20for%20UDK.pdf.

[4] Barry Blundell. On exemplar 3d display technologies. Technical report, Auckland University of Technology, 02 2012. URL http://www.barrygblundell.com/upload/BBlundellWhitePaper.pdf.

[5] Slava Gostrenko. 3d stereoscopic game development - how to make your game look like beowulf 3d. In *NVIDIA Presentations at Game Developers Conference 2008*, San Francisco, USA, February 2008. URL http://www.nvidia.com/object/gdc-2008.html.

[6] Joseph J. LaViola, Jr. and Tad Litwiller. Evaluating the benefits of 3d stereo in modern video games. In *Proceedings of the 2011 annual conference on Human factors in computing systems*, CHI '11, pages 2345–2354, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0228-9.

[7] Mathieu Le Ber. Stereoscopy manager for ogre, January 2012. URL http://sourceforge.net/p/ogreaddons/code/2986/tree/trunk/stereoscopy/.

[8] Michael Lewis and Jeffrey Jacobson. Game engines. *Communications of the ACM*, 45(1):27, 2002.

[9] Jean-Luc Lugrin, Fred Charles, Marc Cavazza, Marc Le Renard, Jonathan Freeman, and Jane Lessiter. Caveudk: a vr game engine middleware. In *Proceedings of the 18th ACM symposium on Virtual reality software and technology*, VRST '12, pages 137–144, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1469-5.

[10] NVIDIA Corporation. Nvidia 3d vision, March 2013. URL http://www.nvidia.com/object/3d-vision-main.html.

[11] Waldir Pimenta and Luís Paulo Santos. A comprehensive taxonomy for three-dimensional displays. In *WSCG 2012 – 20th International Conference on Computer Graphics, Visualization and Computer Vision*, pages 139–146. Union Agency, 2012.

[12] J. Rekimoto. A vision-based head tracker for fish tank virtual reality-vr without head gear. In *Virtual Reality Annual International Symposium, 1995. Proceedings.*, pages 94 –100, mar 1995. doi: 10.1109/VRAIS.1995.512484.

[13] Brian Ries, Victoria Interrante, Michael Kaeding, and Lee Anderson. The effect of self-embodiment on distance perception in immersive virtual environments. In *Proceedings of the 2008 ACM symposium on Virtual reality software and technology*, VRST '08, pages 167–170, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-951-7.

[14] T. Sko and H. Gardner. Head tracking in first-person games: Interaction using a web-camera. *Human-Computer Interaction–INTERACT 2009*, pages 342–355, 2009.

[15] Stereoskopix. Stereoskopix fov2go, January 2013. URL https://www.assetstore.unity3d.com/#/content/2927.

[16] David Trenholme and ShamusP. Smith. Computer game engines for developing first-person virtual environments. *Virtual Reality*, 12:181–187, 2008. ISSN 1359-4338. doi: 10.1007/s10055-008-0092-z. URL http://dx.doi.org/10.1007/s10055-008-0092-z.

[17] TriDef. Tridef 3d, March 2013. URL http://www.tridef.com/products/pc.

[18] Colin Ware and Glenn Franck. Evaluating stereo and motion cues for visualizing information nets in three dimensions. *ACM Trans. Graph.*, 15:121–140, April 1996. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/234972.234975. URL http://doi.acm.org/10.1145/234972.234975.

[19] Colin Ware, Kevin Arthur, and Kellogg S. Booth. Fish tank virtual reality. In *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*, CHI '93, pages 37–42, New York, NY, USA, 1993. ACM. ISBN 0-89791-575-5.

[20] David Wyand. Torque 3d and oculus rift, March 2013. URL http://www.garagegames.com/community/blogs/view/22225.