View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Goldsmiths Research Online

# CaveUDK: A VR Game Engine Middleware

Jean-Luc Lugrin†, Fred Charles, Marc Cavazza,
Teesside University,
School of Computing,
Middlesbrough, TS1 3BA
j-l.lugrin@tees.ac.uk

Marc Le Renard
ESIEA
38 rue des docteurs Calmette et Guérin, Parc Universitaire,
Laval, 53000
marc.lerenard@esiea-ouest.fr

Jonathan Freeman, Jane Lessiter
Psychology Department, Goldsmiths,
University of London,
London, SE14 6NW
j.freeman@gold.ac.uk

## ABSTRACT
Previous attempts at developing immersive versions of game engines have faced difficulties in achieving both overall high performance and preserving reusability of software developments. In this paper, we present a high-level VR middleware based on one of the most successful commercial game engines: the Unreal® Engine 3.0 (UE3). We describe a VR framework implemented as an extension to the Unreal® Development Kit (UDK) supporting CAVE™-like installations. Our approach relies on a distributed architecture reinforced by specific replication patterns to synchronize the user's point of view and interactions within a multi-screen installation. Our performance benchmarks indicated that our immersive port does not affect the game engine performance, even with complex real-time applications, such as fast-paced multiplayer First Person Shooter (FPS) games or high-resolution graphical environments with 2M+ polygons. A user study also demonstrated the capacity of our VR middleware to elicit high spatial presence while maintaining low cybersickness effects. With free distribution, we believe such a platform can support future entertainment and VR research.

## Categories and Subject Descriptors
H.5.1 [**Multimedia Information Systems**]: Artificial, Augmented and Virtual Reality - Virtual Reality for Art and Entertainment

## General Terms
Algorithms, Design, Performance.

## Keywords
Immersive Display, Virtual Reality, Game Engine, Framework.

## 1. INTRODUCTION AND RATIONALE
Game engines have emerged as a unique platform providing increased interactivity and compelling graphics performance [10] [24] [28] [31]. This situation had led several researchers to explore the use of game engines to support high-end VR, in particular immersive displays such as CAVE™-like [6] systems. For instance, Paul Rajlich's *CAVE Quake II*, developed at NCSA, is probably the first immersive implementation of a popular computer game. It has been followed by *CAVE Quake III Arena*, based on the open source Aftershock engine. Juarez et al. [19] have ported the CryEngine2 game engine to a CAVE™-like installation, through their *CryVE* system. However, they have reported average frame rates < 20 fps, which may not be sufficient to support a comfortable viewing and interaction experience. *CAVEUT* was originally developed at the University of Pittsburgh [15] and later extended to include stereoscopy [16], but its version of the Unreal® Engine is now out of date. *BlenderCave* [7] is the VR extension of the open-source Blender engine but its VR version demonstrated limited rendering performances, without support for dedicated I/O VR peripherals. *MiddleVR* [27] offers a VR port of the Unity Game Engine [37], but its visual performances do not reach that of the most advanced commercial game engines like the CryEngine [5] or the Unreal® Engine [38]. Performances achieved by game engines are the results of extremely complex and optimised architectures. One major challenge faced by this endeavor is the preservation of game engine performances and content synchronization within a multi-screen stereoscopic displays, since game engine optimizations have not been developed with multi-screen displays in mind. Another important aspect of high-end multi-screen VR systems, not originally part of game engines, is the integration of a mechanism for the inclusion of tracker input and the configuration of individual screens. VR systems typically propose large field of view (multiple surrounding screens), accurate motion tracking (tracker devices) and depth perception (3D rendering and head motion parallax). One important technical requirement of a multi-screen system is to preserve visualisation and interaction consistency in between screens while delivering a comfortable refresh rate and low end-to-end tracking latency. Multi-screen consistency mostly represents the preservation of the virtual object alignment when visualised over different focal planes (i.e. screens). This is especially important when the user moves an object across multiple screen borders via a virtual hand (such as a virtual weapon held by the user in a typical FPS game). Traditional VR frameworks implement asymmetric frustum, homography correction and accurate distributed object synchronization protocols to ensure the best visualisation coherence and interactivity over different screens. However, game engines typically do not include such features. Their complex architecture and implementation, as well as their proprietary source code and optimization, often make them very opaque to such transformation. In this paper, we present CaveUDK as a high-level VR middleware for CAVE™-like platforms developed on top of a state-of-the-art game engine, Unreal® Development Kit (UDK) [38]. It constitutes the natural follow-up of CaveUT [16], while providing more advanced and generic UnrealScript VR Class framework and set of software tools for multi-screen visulisation, interaction, conversion, calibration and deployment (Figure 1). Here, we discuss its implementation, performances and reusability, where *reusability* refers to its extensible high-level class framework and the presence of conversion, calibration and deployment systems.

†*Current Affiliation: Universität Würzburg, HCI, Germany*

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
*VRST'12*, December 10–12, 2012, Toronto, Ontario, Canada.
Copyright 2012 ACM 978-1-4503-1469-5/12/12...$15.00.

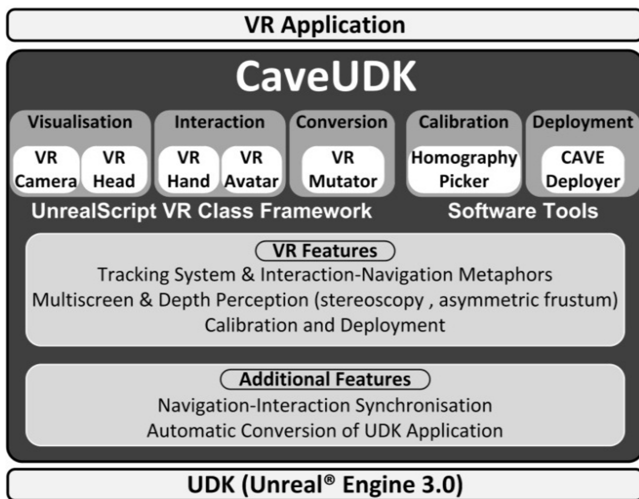


**Figure 1. The CaveUDK Framework (a layer of VR objects and tools in-between the VR hardware and the application).**

# 2. SYSTEM OVERVIEW

## 2.1 Choice of Game Engine

UDK is a free professional authoring toolset based on UE3, which is now adopted by a large development and research community. It has been installed on more than 1.4 million unique machines, and showcases numerous applications in training simulation, games, architecture and construction simulations [39]. Epic Games has allowed us to access the game engine's C++ source code to gain necessary knowledge and efficiently integrate VR features at a low level. However, one key aspect of our approach was to maximize the flexibility and extensibility of the system by keeping a major part of the project open source via a DLL plug-in architecture (for tracker integration and screen calibration) and an open-source UnrealScript classes framework (for rapid development over existing or new Unreal®-based application). CaveUDK exposes the VR system features without requesting the acquisition and recompilation of the game engine source code.

## 2.2 Target Hardware Platforms

Our target platforms for CaveUDK are multi-screen, immersive display installations such as CAVE™. Our own CAVE™ platform is a four-screen cubic-shape structure of 3.0×3.0×2.25 meters (Figure 2). Our PC cluster is composed of 5 image generators running Windows 7 Ultimate 64 bit, 2×Xeon E5606 Processor *(Quad Core, 2.13 GHz, 4MB Cache, 4.80 GT/s Intel®)*, 12Gb DDR3 RAM 1333Mhz and 2.5GB GDDR5 NVIDIA Quadro 5000 with G-Sync. One of the image generators is used as the game server; while each of the four others are connected to a projector (*Christie Digital Mirage S+6K SXGA+ DLP 3D*) generating 1400×1050×100Hz. For 3D vision, the user wears stereoactive shutter glasses (*NVIDIA Vision Pro Kit [29]*), while stereo signals and rendering synchronisation are handled at 50 Hz by the NVIDIA cards and drivers). Real-time tracking is operated by an Intersense™ IS900 system [14] for both head and wand tracking, using a VRPN *(Virtual Reality Peripheral Network [40])* server.

# 3. FRAMEWORK AND FEATURES

In order to integrate our middleware on top of the Unreal® Game engine, we modeled our architecture using a distributed approach, whereby each VR Client (game client) represents a different screen synchronised with the VR Server (game server) connected to a Tracker Server (Figure 2). Our VR framework, proposes a set of UnrealScript classes, called VR Objects, supporting consistent multi-screen visualisation (*VR-Camera* and *VR-Head* classes) and interactions (*VR-Hand* and *VR-Avatar* classes) in a multi-screen context. These classes are internally connected to VR Trackers through a module, called *VR Interface* (C++ DLL) directly integrated to the game engine source code. They also implement a specific replication process to synchronize user's point of view and interactions across all VR Clients (i.e. on all screens). We rely on a customized Master-Proxy object replication approach [25] which benefitted from the fast Unreal® network system to ensure position and states synchronicity.

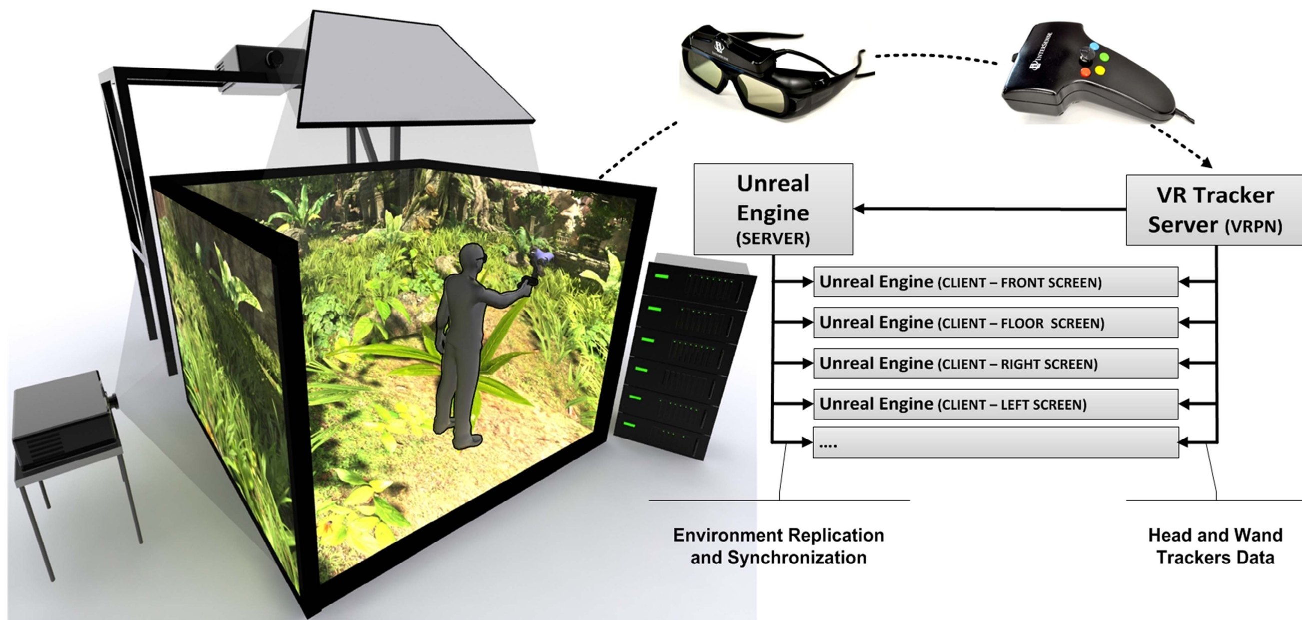


**Figure 2. CaveUDK Architecture within a CAVE™-like Platform (2M+ triangles per screen on average).**

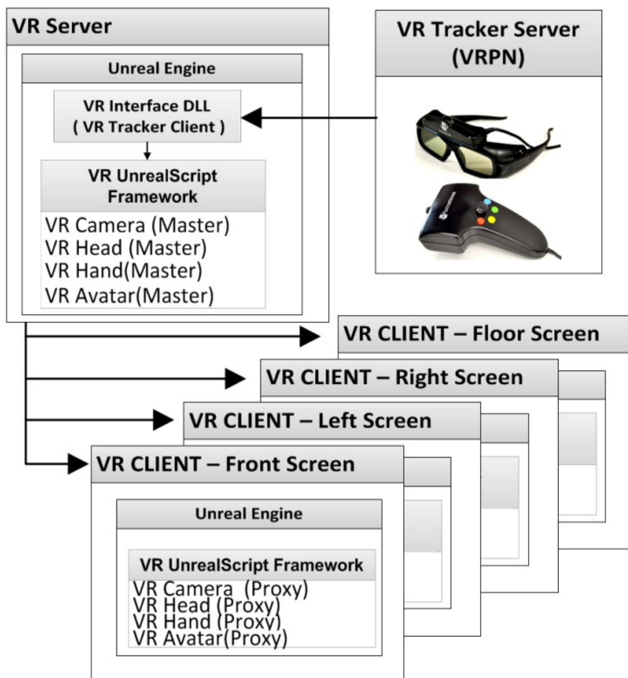


**Figure 3. VR Framework Integration and Replication.**

## 3.1 Visualization: *VR Camera* and *VR Head*

Conveying depth and layout is a key element of a VR system. In a multi–view projection system with a strong user centric paradigm such as a CAVE™, egocentric depth perception is simulated using binocular stereopsis and motion parallax. Meanwhile, creating egocentric depth perception in a multi-screen environment using a distributed architecture requires the implementation of a specific camera system. Our VR framework (Figure 3) relies on a *Virtual Camera Cluster* [7] system to map the topology and geometry of a projection system to different game clients. The location of the *Virtual Camera*s running on each game client *(i.e. VR-Camera Proxy)* are synchronised with a virtual camera (*i.e. VR-Camera Master)* controlled by the user on the game server (Figure 3). The movements of the master camera are replicated on the game client, simulating the user's motion on all screens. Meanwhile, each VR Client computes its own camera direction and viewing frustum using this origin, their respective display topology (i.e. Front, Left, Floor, Right screen) and head tracker position. Stereoscopic visualisation is achieved via specific graphics drivers synchronised with a 3D compatible projection system and glasses. We adopted the NVIDIA 3D vision system [30] which provides a generic and low cost solution, capable of adding 3D perspectives to numerous existing graphics applications. The implementation of motion parallax is also important, as it faithfully reproduces the illusion of depth by adding depth cues to artificial stereopsis [8] [17]. Motion parallax adapts each screen projection perspective to match the user's point of view as he/she moves inside the CAVE™. Therefore, the virtual world camera position and perspective matrix need to be constantly adjusted for each screen to provide the correct viewpoint. Cameras should thus compute an asymmetric viewing frustum [1] as shown on Figure 4 for each screen and for each frame using exact location of the user's head within the CAVE™ (head tracker information). A dynamic asymmetric frustum system allows the coverage of the entire visible volume in the VR scene, while preventing frustum intersections. In addition, an off-axis projection matrix allows a comfortable 3D viewing

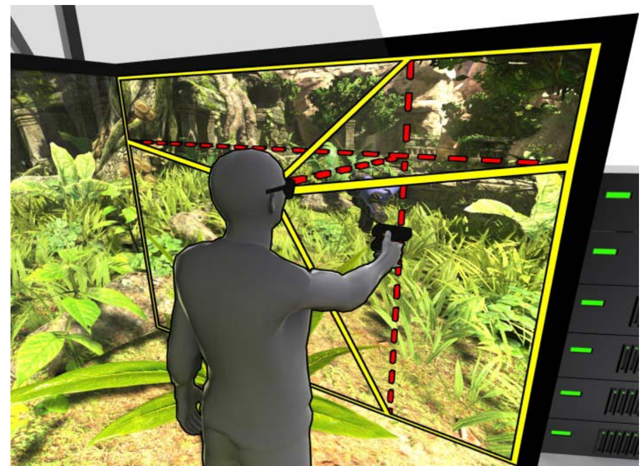

**Figure 4. VR Client viewpoint using asymmetric frustum with head tracking (horizontal /vertical FOV in dashed lines).**

experience which notably reduce the window violation effect by covering a larger part of the screen for each eye [3] [33]. Our system supports an off axis perspective implementation as the one proposed by the OpenGL wrapper library, called VRGL (by Willem de Jonge). However, we implemented such an asymmetric projection matrix computation directly at the core of the game engine rendering system[1]. The *VR-Head* object is responsible for computing the asymmetric frustum and applying user head position offset to the *VR-Camera*. As the *VR-Head* object is replicated, each client has access to the exact location of the user's head within the CAVE™, allowing them to perform their own frustum modifications. Motion parallax is then simulated by attaching the master *VR-Camera* to the head tracker with a combination of dynamic viewing frustum computed locally by each VR Client.

Therefore, multi-screen depth perception is rendered through a combination of binocular stereopsis, handled by the NVIDIA drivers and graphics cards, and motion parallax, managed by our distributed camera system. Nevertheless, with a distributed approach, it is essential to accurately synchronise camera motions on the VR Clients to achieve a smooth and consistent multi-screen visualisation when navigating in the virtual environment. The mechanisms ensuring VR Client camera synchronisation are discussed in the Navigation-Interaction synchronisation section.

## 3.2 Interaction: *VR Avatar* and *VR Hand*

User interaction in CAVE™-like environments often resorts to the explicit visualization of a virtual hand supporting interaction-navigation metaphors [33] [34].Our system supports user virtual representation through the *VR-Hand* and *VR-Avatar* objects. The *VR-Hand* represents the wand tracker, which is constantly converting the tracker physical position to its position transposed within the virtual world. It continuously receives tracker data from the *VR Interface* and triggers associated tracker events (button, analog, position and orientations updates). The *VR-Hand* facilitates high-level programming of navigation and interaction aspects by redefining the tracker events responses (Figure 5) and by using its positional data as the actual wand tracker position. It

[1] For confidentiality reasons, the integration of the dynamic frustum computations inside the game engine could not be discussed here.

thus enables developers to easily attach virtual objects to the wand tracker and program interaction while automatically ensuring synchronization across all screens. The synchronization of virtual hand position and orientation is explained in the following section.

```
event TrackerInputs(TrackerInput _Input)
{
  switch(_Input.buttonId)
  {
  case 1 :  if(_Input.action == ePressed)
               GetALocalPlayerController().startfire(0);
            if(_Input.action == eReleased)
              GetALocalPlayerController().stopfire(0);
            break;
  //...
```

**Figure 5. Example of Tracker Event Overriding in *VR-Hand.***

The *VR-Hand* class also proposes a default mapping tracker event to keyboard/mouse inputs for fast conversion between desktop and immersive settings. The default navigation-interaction paradigm supported by our system is the "moving-by-pointing" navigation model [4] and the Virtual Hand/Ray-Casting [22] for interaction model. The user navigates in the virtual environment by pointing the wand tracker to a direction and moving the wand analogue stick. Consequently, the user's avatar *(VR-Avatar*) on the game server is moving in the specified direction. The avatar motion is then applied to the master *VR-Camera* also running on the server. The movements of the master camera are automatically replicated to all game clients, simulating the user's motion across all CAVE™ screens (Figure 3). To simulate user collision on clients, a collision cylinder is attached to the *VR-Avatar* proxies running on all individual clients. The *VR-Avatar* proxies are invisible to prevent them to appear in the CAVE™ and occlude the user's vision. As previously mentioned, for motion parallax simulation, when the user is moving inside the CAVE™ (i.e. walking), the master VR-Avatar position is automatically adjusted to match the offset detected by the VR-Head.

## 3.3 Synchronising Navigation-Interaction

As observed in our previous versions of CaveUT [6], the native Unreal® replication system would generate perceptible latency/ jitter during motion, and object misalignment in-between screens. Latency and jitter have been proven to diminish performance and increase simulator sickness [32]. Therefore, in order to considerably reduce the translation and rotation latency/jitters and synchronize interaction, we developed our own Navigation-Interaction replication systems i) removing native replicated variable aggregation and ii) forcing replication updates for certain data types.

The Unreal® Engine provides a high level networking and replication system [36] which relies on a very flexible and rapid distributed object approach. It maximises the responsiveness and consistency of the shared virtual environment using an optimised UDP-based network protocol, in combination with lock-step prediction/correction algorithms (e.g. dead-reckoning), area of interest management (AOI) and data quantization (compression). The location/acceleration vectors and quaternions are quantized to single 32-bit variables before being replicated. Therefore, the data loss from the compression or inaccuracy from the prediction could lead to slight differences in terms of object spatial or animation state between clients. Such a lack of accuracy or delay is acceptable in multiplayer games, where users do not share the same screen. However in our multi-screen context, they are perceptible and so could generate multi-screen object misalignments, breaking multi-screen consistency and so compromising the immersive experience. One crucial point in avoiding such misalignment is to first provide accurate virtual camera synchronization for each screen. Subsequently, all virtual objects manipulated by the user should also support accurate and low latency replication. This is especially important when the user moves an object across multiple screen borders via the Virtual Hand (such as a virtual weapon held by the user in a FPS game). A slight difference in position from one screen to another could result in the user perceiving two or three different virtual objects instead of a single one.

Therefore, our VR Object replication implementation takes advantage of the fast Unreal® network layer to synchronize client views, using a customized high-frequency network replication pattern for our *VR-Camera*, *VR-Head* and *VR-Hand* objects. The UnrealScript code samples (Figures 6-7-8) illustrate our replication pattern based on *asynchronous unreliable remote function calls* to prevent data quantization and reduce replication latency.

```
simulated state VIRTUAL_MASTER_CAMERA_UPDATE
{
  simulated event Tick (float deltatime)
    {
    //...
    CaveUDkPlayerController(CavePlayerController)
    .ServerReplicatePosition(newCamLoc,  newCamRot,
        VirtualHand.location,  VirtualHand.rotation,
        VirtualHead.location,  VirtualHead.rotation );
     //...
```

**Figure 6. *VR-Object* replication process via an asynchronous unreliable remote function call (in *VR-Camera* class).**

The replication process is handled by the *VR-Camera* object running on the server (Figure 6). For every frame (aka game tick), the *VR-Camera* evaluates if the position and rotation of the virtual camera, head and hand need to be updated on VR Clients (e.g. when the user moves the virtual hand using the wand). In case of discrepancies, the *ServerReplicatePosition* function is executed on the server, hence the keyword *server*.

```
unreliable server function ServerReplicatePosition(
vector cam_v,rotator cam_r,vector hand_v,rotator hand_r,
vector head_v ,rotator head_r)
{
 local CaveUDKSpectatorController PC;
 foreach AllControllers(class'CaveUDKSpectatorController', PC)
 {
  PC.ClientReceivedPosition (cam_v.X,cam_v.Y,cam_v.Z,
                             cam_r.Pitch,cam_r.Yaw,cam_r.Roll,
                             hand_v.X,hand_v.Y,hand_v.Z,
                             hand_r.Pitch,hand_r.Yaw,hand_r.Roll,
                             head_v.X,head_v.Y,head_v.Z,
                             head_r.Pitch,head_r.Yaw,head_r.Roll);
 //...
```

**Figure 7. *Server* function accessing game clients and triggering an instantaneous remote function call on clients (Note the uncompressed basic variables as parameter).**

This function accesses the list of player spectators (i.e. a VR Client registered as non-playing game client) connected to the server, and requests them to execute the *ClientReceivePosition* function using the master *VR-Camera*, *VR-Head* and *VR-Hand* positions as parameters (Figure 7). The *ClientReceivePosition* function is declared as a *Client* function, which will force the function to run on the game clients. Function replication in Unreal® is asynchronous, meaning that remote function calls are executed immediately rather than at the end of the game tick as

for replicated class variables. In addition, by passing floating values instead of vector and rotator variables we avoid the engine native data compression associated with such structures. Consequently, the *ClientReceivePosition* function is executed on the client as fast as possible, and maps its local copy of the *VR-Camera*, *VR-Head* and *VR-Hand* to an exact match of the positions present on the server (Figure 8).

```
unreliable client function ClientReceivedPosition(
float c_x ,float c_y ,float c_z ,int c_p ,int c_yaw , int c_r,
float ha_x,float ha_y,float ha_z,int ha_p,int ha_yaw, int ha_r,
float he_x,float he_y,float he_z,int he_p,int he_yaw, int he_r)
{
  if( LocalPlayer(Player).CaveCamera !=none)
  {
     LocalPlayer(Player).CaveCamera.setBaseActorCameraPos(
        c_x ,c_y ,c_z ,c_p,c_yaw,c_r,
        ha_x,ha_y,ha_z,ha_p,ha_yaw,ha_r,
        he_x,he_y,he_z,he_p,he_yaw,he_r);
//...
```

**Figure 8. *Client* function executed by a game client in order to update their own copy of the user's camera, head and hand.**

It is important to understand that an *unreliable* function replication makes no guarantee about the ordering of the remote function call (similar to a UDP protocol approach). Within our context, where the user's camera location and rotation need to be replicated at a high frequently to preserve multiscreen consistency and smooth navigation, using a *reliable* function would overload the network and cause latency. The *reliable* function guarantees delivery and ordering, but creates a delay due to acknowledgement waiting-time. Missing a replication update is not critical with a high refresh rate (50Hz) on a LAN. However, to avoid network congestion and to adapt to different hardware capacity, the replication frequency is adjustable from a configuration file (10 ms in our installation).This replication pattern is thus able to efficiently achieve smooth and accurate camera transition among multiple screens with no latency or jitter perceptible. It also permits the system to have a perfect multi-screen motion synchronisation for virtual objects attached to the wand tracker, such as virtual hand or weapon mesh (as shown in Figure 12 during in our immersive game usability study).

# 4. VR DEVELOPMENT WITH CAVEUDK

CaveUDK provides a high-level framework of VR Object and software tools for rapid development of VR application on the top of UDK. The following section describes its features and overall approach to hardware integration and software development.

## 4.1 Hardware Integration and Calibration

Because most multi-screen installations are custom-built, a VR Middleware should be able to adapt to different screens and tracker configurations. Our VRPN Client approach, directly integrated at the game engine level, enable to easily plug most of the VR devices available, while our simple screen configuration and homography correction supports different screen layout. The software calibration of the screen alignment is also essential to rapidly adapt the system to particular screen and projector configurations. The screen spatial configuration is easily handled via a text file specifying the coordinates of the screen corners in the tracker referential (Figure 9). However, to facilitate screen matching alignment, the system also includes a homography correction [2], providing a projective distortion matrix to perfectly match a particular screen projection geometry without manually adjusting the projector calibration settings and physical position for a perfect multi-screen alignment.

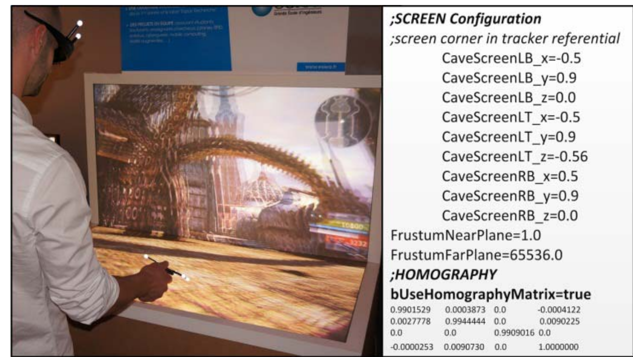


**Figure 9. Screen calibration on Workbench (homography).**

The tool delivered with the system is *Homography Picker* [12], and the homography matrix is also configured via a text file, read by the *VR Interface*. Figure 9 illustrates the porting of CaveUDK under Workbench-like screen configuration using an Optitrack V120: Trio tracker system [30]. Theoretically, CaveUDK supports up to 16-32 screens (Unreal Server maximum capacity), depending on application complexity. However, further evaluation should be carried to identify the actual screen limitation preserving application responsiveness and consistency.

## 4.2 Software Development and Deployment

### *4.2.1 Interaction Programming*

The reusability of the system mostly relies on the ease of extension of the VR framework classes and its associated tracker-events and functions overriding. As previously demonstrated (Figure 5), custom interactions could be implemented by simply extending the *VR-Hand* or *VR-Head* classes and redefining their initial properties or tracker events. The redefinition of those classes is easily specified in the *VR-Camera* Class as shown on Figure 10. In addition, the inclusion of a high-level event system on the top of a high-level OO scripting language (i.e. UnrealScript) also constitutes a critical feature for the VR community in terms of rapid development and interaction programming. Developer can redefine their own VR object classes and apply them to a UDK environment using our Conversion System as discussed in the next section.

```
class CaveUDKTCaveCamera extends CaveUDKCaveCamera;
//...
DefaultProperties
{
  //...
  VirtualHandClass=class'CaveUDKVirtualHand'
  VirtualHeadClass=class'CaveUDKVirtualHead'
  //...
```

**Figure 10. Redefining Custom Classes within CaveUDK.**

### *4.2.2 Conversion and Integration: VR-Mutator*

Adapting the game engine ensures that the approach is generic and compatible with all other applications developed with the same game engine. An application or game developed with UDK should be portable with minimum effort to an immersive context using our approach. Consequently, we designed a non-intrusive conversion system making our system reusable for many existing VR applications without any modification. The integration of the VR system to an Unreal® application requires the simple addition of an Unreal® *Mutator* to the game engine launching process. The Unreal® *Mutator* system allows the modification of certain classes or events in an existing Unreal® game level without modifying the

**Figure 11. Automatic Desktop (top)-Immersive (bottom) conversion using the VR-Mutator System.**

actual source code. Our specific CAVE™ *Mutator* automatically instantiates and activates our VR Objects within a UDK environment at run-time. This activation overrides the default camera system, activates the frustum modification and tracker event generation. Figure 11 illustrates the automatic porting of a compelling complex graphical environment (DirectX 11.0) [29] into an immersive version using our CAVE™ *Mutator* and its default control setting.

### *4.2.3 Deployment: Cave Launcher*

Adopting a distributed approach results in the need to control remote game client and deploy new scripts, assets and maps to a group of machines before being able to connect to a game server. For security and anti-cheating issues, Unreal® maps and asset packages are digitally signed, forcing every game client to execute the exact same copy of the UnrealScript and environment as the one running on the game server. In addition, the screen calibration files for each client also need to be deployed with minimal effort. Consequently, we developed a Java-based deployment software tool named *CaveLauncher.* It automatically transfers asset packages and screen configuration files from the server to all clients at once, while enabling to remotely control the game clients execution.

In sum, interactions programming, conversion and deployment of VR application is considerably simplified and transparent to UDK developer using CaveUDK framework and tools.

# 5. EVALUATION

## 5.1 System Performances

Low latency is a critical feature of a VR system, as it is not only a negative factor in simulator sickness, but also considerably affects interaction [35]. Consequently, VR systems should support both high rendering refresh rates (>=30Hz) and high interaction responsiveness (<=150 ms for digital games [18]). For environments of a complexity equivalent to current high-quality 3D games rendering (200k to 2M triangles per frame), our immersive system reproduces similar rendering rates (≈ 50Hz) as normal 3D desktop configurations (Table 1). Our system only presents a lower frame rate (< 20 Hz) for complex layouts such as the last UE3 tech-demo, *NightAndDay,* displaying up to 6M triangles per frame. To measure responsiveness, we also performed video-based measurements of the end-to-end input latency using a frame-counting method as described in [9], which is better adapted to direct in-game measurements, but less accurate than the pendulum method discussed in [32]. As illustrated in Table 2 our results within a FPS game context (UT3 game map over 600K and 2M polygons) demonstrated an average response time of ≈ 82 ms for user interaction, which is the average delay between a tracker event (e.g. button pressed) and its associated virtual world events (e.g. The weapon firing on all four CAVE™ screens). In term of Navigation latency, the movements inside the virtual environment triggered by the tracker analogue stick are slower on average (≈137 ms). The virtual hand motion transcribing the motion of the wand tracker inside the CAVE is also presenting a similar latency (≈126 ms), considering our Intersense IS900 tracker 4 ms latency. The difference in response time between firing and camera/virtual hand movement is due to the optimized native mechanism for firing events replication in UT3 (crucial element for an online multiplayer FPS), while our camera/virtual hand replication going through function replication has a lesser priority within the unreal network system. The difference in latency measurements between camera and virtual hand motion, despite relying on the same replication system, could be explained by the lack of accuracy of the frame counting method. Future work should include less error-prone measurement methods as discussed in [32]. However, our system responsiveness remains below the 150-200 ms threshold given by the current literature in digital games [18] and standard causal perception studies [26] [35]. Overall, the performance results of our system demonstrated the appropriateness of our VR framework implementation regarding rendering performance and end-to-end latency.

**Table 1: Comparison of desktop-immersive performances**

| Map | Triangles in Field of View | 3D Desktop FPS | 3D CAVE FPS |
|---|---|---|---|
| DM- Deck | ≈ 200-600K | ≈ 50 | ≈ 50 |
| Foliage | ≈ 1500-2500K | ≈ 50-45 | ≈ 45-40 |
| NightAndDay | ≈ 2500-6000K | ≈ 35-30 | ≈ 20-15 |

**Table 2: End-to-end Latency performances (Immersive)**

| Interactions within UT3 environments (3D/ 50Hz) | Av. Frame at 220 Hz (10 samples) | Av. in ms |
|---|---|---|
| Firing weapon (Button) | 17.9 (σ ≈2.02) | ≈82 |
| Move Camera (Analog) | 30.1 (σ ≈ 4.43) | ≈136 |
| Move Virtual Hand (wand) | 27.7 (σ ≈ 5.36) | ≈126 |

## 5.2 User Experience

Prolonged exposure to stereoscopic VR can lead to adverse effects such as cybersickness or visual discomfort, through a variety of mechanisms [11] [13] [21]. Therefore, we conducted measurements of Cybersickness in a user study comparing desktop and immersive version of the same FPS game: Unreal® Tournament 3 (UT3) as illustrated by Figure 12. Before and after each session, participants completed the Simulator Sickness Questionnaire (SSQ), developed by Kennedy et al [20]. At the end of each session, users were presented with the ITC-SOPI Presence questionnaires proposed by Lessiter et al. [23]. Amongst our 40 participants, the different SSQ scores obtained for each of the

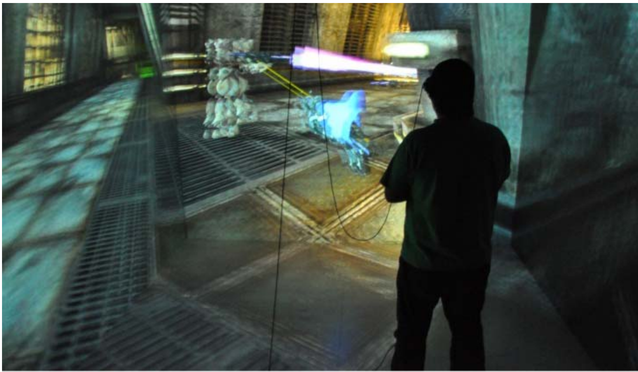

**Figure 12. Immersive 3D gaming with FPS game (weapon attached to virtual hand – in blue).**

cybersickness components (Nausea, Oculomotor and Disorientation) are low (around 5-7%), giving an overall score of 19.5% for an average exposure time of 10 min. It is important to note that the desktop game session revealed an average of 7.33 % SSQ score, creating only a 12% gap with the immersive version. All participants felt comfortable having completed the whole experiment. Meanwhile, the average of Negative Effects expressed in the ITC-SOPI questionnaire (mostly visual discomfort) increased by 23% in the immersive condition. As expected the immersive settings resulted in superior levels of Spatial Presence (+20%), Ecological Validity (+14%) and Engagement (+4%). A series of paired samples t-tests were conducted to explore whether these differences were statistically significant across the two experimental conditions (Desktop *vs.* CAVE™). Results for all four subscales revealed that significantly higher presence related ratings were given for the CAVE™ setting compared with the desktop setting: Spatial Presence ($t(38) = 9.51$, $p < 0.01$), Engagement ($t(38) = 3.59$, $p < 0.01$), Ecological Validity ($t(38) = 6.65$, $p < 0.01$), while Negative Effects remained moderate ($t(38) = 9.10$, $p < 0.01$).

In conclusion, the navigation-interaction replication pattern employed appears to provide a comfortable multi-screen visualization while having little impact on the performances (real-time interaction response time and overall frame rate close to desktop performances). CaveUDK is thus capable of supporting state-of-the-art real-world applications, providing real-time response, multi-screen consistency, and convincing 3D rendering. In addition, our preference questionnaire results also demonstrated a strong preference from users towards the immersive setting (72% of users expressed a clear preference), which is also confirmed by their self-reporting (*"It made me feel part of the world, easy to get immersed", "The immersion was so complete that towards the end of the second round, I forgot the walls were separated and just saw the environment around me").*

## 6. CONCLUSIONS

CaveUDK offers a high-level VR middleware for the rapid deployment of immersive applications developed with UDK. It brings the benefits of developing with game engines to the VR community: an unmatched performance/cost ratio for visual rendering, sophisticated built-in Physics engines and advanced mechanisms for environment and characters behavior. The reusability of our system resides in its extensible high-level class framework and the integration of an automatic conversion, calibration and deployment systems. Custom interactions can be implemented by simply extending our UnrealScript classes and redefining their initial properties or tracker events. Our performance benchmarks also indicated that our implementation does not significantly affect the baseline performance characteristics of the game engine, even with complex real-time applications such as fast-paced multiplayer FPS games and high-resolution graphic environments (2M+ polygons). The user study demonstrated the capacity of the resulting system to elicit high spatial presence without introducing significant cyber sickness effects. The free, open source distribution of CaveUDK should open interesting perspectives for VR technology, especially concerning the usability and development of future immersive entertainment, serious games or virtual training applications.

One important feature missing from the framework is the presence of a multi-screen compatible 3D head-up displays (HUD) system, which would allow a quick conversion of 2D Desktop HUD to 3D Immersive HUD and follow the user's head position while adjusting horizontal screen parallax to limit visual fatigue. This feature would be primarily needed for immersive gaming, which would constitute a natural application of our system. Further work also should investigate the factors of such visual fatigue, and explore possible solutions to reduce it such as "Dynamic Parallax Separation" with content-adaptive adjustment [13].

## 7. ACKNOWLEDGEMENTS

Epic Games (Mark Rein) and Public VR (Jeff Jacobson) are thanked for granted us access to the Unreal® engine source code. Matthew Laverick provided the visuals of Figure 1. This work has been funded (in part) by the European Commission through the CEEDS project (FP7-ICT-258749).

## 8. REFERENCES

[1] Arthur, K. W. 3D Task performance using head-coupled stereo displays. M.Sc. Thesis. University of British Columbia, 1993.

[2] Bimber, O. and Raskar, R., 2005. *Spatial Augmented Reality: Merging Real and Virtual Worlds*. AK Peters, CRC Press.

[3] Bourke, P., 1999. *Calculating Stereo Pairs*. URL: http://local.wasp.uwa.edu.au/~pbourke/miscellaneous/stereographics/stereorender/

[4] Bowman, D. A., Johnson, D. B. and Hodges, L. F., 1999. Testbed evaluation of virtual environment interaction techniques. In *Proceedings of the ACM symposium on Virtual reality software and technology (VRST '99)*. ACM, New York, NY, USA, 26-33.

[5] *CryENGINE®3*. URL: http://mycryengine.com/

[6] Cruz-Neira, C., Sandin, D. J. and DeFanti, T. A., 1993. Surround-screen projection-based virtual reality: the design and implementation of the CAVE. In *Proceedings of the 20th annual conference on Computer graphics and interactive techniques.* (Anaheim, CA, USA). ACM, New York, NY, USA, 135-142.

[7] Gascón, G., José M. Bayona, José Miguel Espadero, Miguel A. Otaduy, 2011. BlenderCAVE: Easy VR Authoring for Multi-Screen Displays. *SIACG 2011: Ibero-American Symposium in Computer Graphics* .

[8] Hassaine, D., Nicolas S. Holliman, and Simon P. Liversedge., 2010. Investigating the performance of path-searching tasks in depth on multiview displays. *ACM Trans. Appl. Percept. 8, 1, Article 8* (November 2010), 18 pages.

[9] He, D., Liu, F., Pape, D., Dawe, G. and Sandin. D., 2000. Video-Based Measurement of System Latency, *International Immersive Projection Technology Workshop*, Ames IA, USA.

[10] Herrlich, M., 2007. A Tool for Landscape Architecture Based on Computer Game Technology. In *Proceedings of the 17th International Conference on Artificial Reality and Telexistence*, IEEE, 264-268.

[11] Hoffman, D., Girshick, A., Akeley, K., and Banks, M., 2008. Vergence–accommodation conflicts hinder visual performance and cause visual fatigue. *Journal of Vision, volume 8(3)*.

[12] *Homography Picker*. URL: https://sites.google.com/site/jeromeardouin/projects/homography-picker

[13] Hughes, J., 2011. *Automatic Dynamic Stereoscopic 3D. Game Engine Gems 2, Chapter 9*, Eric Lengyel AK Peters/CRC Press 2011, 135–149.

[14] *Intersense IS 900*. URL: http://www.intersense.com/pages/20/14

[15] Jacobson, J., 2003. Using CaveUT to build immersive displays with the unreal tournament engine and a PC cluster. In *Proceedings of the 2003 symposium on Interactive 3D graphics*. (Monterey, California, USA). ACM, New York, NY, USA, 221-222.

[16] Jacobson, J., Le Renard, M., Lugrin, J-L. and Cavazza, M., 2005. The CaveUT system: immersive entertainment based on a game engine. *In Proceedings of the 2005 ACM SIGCHI International Conference on Advances in computer entertainment technology* (ACE '05). ACM, New York, NY, USA, 184-187.

[17] Jones, J. A., Swan, J. E., Singh, G., Kolstad, E., and Ellis, S. R., 2008. The effects of virtual reality, augmented reality, and motion parallax on egocentric depth perception. In *Proceedings of the 5th symposium on Applied perception in graphics and visualization (APGV '08)*. ACM, New York, NY, USA, 9-14.

[18] Jörg, S., Normoyle, A. and Safonova, A., 2012. How responsiveness affects players' perception in digital games. *In Proceedings of the ACM Symposium on Applied Perception (SAP '12)*. ACM, New York, NY, USA, 33-38.

[19] Juarez, A., Schonenberg, W. and Bartneck, C., 2010. Implementing a low-cost CAVE system using the CryEngine2. *Entertainment Computing*, 1(3-4), (December 2010), 157–164.

[20] Kennedy, R., Lane, N., Berbaum, K. and Lilienthal, M., 1993. Simulator Sickness Questionnaire: An Enhanced Method for Quantifying Simulator Sickness. *The International Journal of Aviation Psychology*, Vol. 3, Issue 3, 203-220.

[21] Lambooij, M., Fortuin, M., IJsselsteijn, W. A., & Heynderickx, I., 2009. Measuring Visual Discomfort associated with 3D displays*.* In *Proceedings of SPIE,* 7237, San Jose, CA, USA.

[22] Lee, G. A., Kim, G. J. and Park, C., 2002. Modeling virtual object behavior within virtual environment. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology (VRST '02)*. ACM, New York, NY, 41-48.

[23] Lessiter, J., Freeman, J., Keogh, E., and Davidoff, J. A 2011. Cross-Media Presence Questionnaire: The ITC-Sense of Presence Inventory. *Presence: Teleoperators and Virtual Environments*, 10:3, 282-297.

[24] Lewis, M., and Jacobson, J., 2002. Game Engines In Scientific Research. *Communications of the ACM*, 45, 1, 2002, 27-31.

[25] Li, F.W.B., Lau, R.W.H., Kilis, D. and Li, L.W. F., 2011. Game-on-demand: An online game engine based on geometry streaming. *ACM Trans. Multimedia Comput. Commun. Appl.* 7, 3, Article 19), 22 pages.

[26] Michotte, A. (1963). *The perception of causality*. New York: Basic Books. Translated from the French by T. R. and E.Miles

[27] *MiddleVR*. URL: http://www.imin.fr/middlevr-for-unity/

[28] Noh, S. S., Hong, S. D., & Park, J. W., 2006. Using a Game Engine Technique to Produce 3D Entertainment Contents. In *Proceedings of the 16th international Conference on Artificial Reality and Telexistence--Workshops* (November 29 - December 01, 2006). IEEE Computer Society, 246-251.

[29] *NVIDIA*. URL: http://www.nvidia.com

[30] *OptiTrack V120-Trio*. URL: http://www.naturalpoint.com/optitrack/products/v120-trio/

[31] Richie, A., Lindstrom, P. and Duggan, B., 2006. Using the Source engine for Serious Games. In *Proceedings of the 9th International Conference on Computer Games: AI, Animation, Mobile, Educational & Serious Games*, November 2006, Dublin Institute of Technology, Ireland.

[32] Steed, A., 2008. A simple method for estimating the latency of interactive, real-time graphics simulations. In *Proceedings of the 2008 ACM Symposium on Virtual Reality Software and Technology (VRST '08)*. NY, 123-129.

[33] Sherstyuk, A. and State, A., 2010. Dynamic eye convergence for head-mounted displays. In *Proceedings of the 17th ACM Symposium on Virtual Reality Software and Technology (VRST '10)*. ACM, New York,, USA, 43-46.

[34] Sutcliffe, A., Gault, B., Fernando, T. and Tan, K., 2006. Investigating interaction in CAVE virtual environments. *ACM Trans. Comput.-Hum. Interact.* 13, 2, 235-267.

[35] Teather, R.J., Pavlovych, A., Stuerzlinger, W. and MacKenzie, I.S., 2009. Effects of tracking technology, latency, and spatial jitter on object movement. In *Proceedings of the 2009 IEEE Symposium on 3D User Interfaces* (3DUI '09). Washington, DC, USA, 43-50.

[36] *UDK Networking and Replication*. URL: http://udn.epicgames.com/Three/ReplicationHome.html

[37] *Unity® Game Engine.* URL: http://unity3d.com/

[38] *Unreal® Development Kit (UDK)*. URL: http://www.unrealengine.com/udk/

[39] *Unreal® Engine Technology Awards.* URL: http://www.unrealengine.com/awards_accolades/

[40] *Virtual Reality Peripheral Network (VRPN)*. URL: http://www.cs.unc.edu/Research/vrpn/