BlenderCAVE: Easy VR Authoring for Multi-Screen Displays

Jorge Gascón José M. Bayona José M. Espadero and Miguel A. Otaduy

URJC Madrid, Spain

http://www.gmrv.es/~jgascon/BlenderCave/index.html

ABSTRACT

BlenderCave is a framework for the fast creation of virtual reality applications for multi-screen display systems. It consists of a set of extensions to the open-source Blender Game Engine (BGE), leveraging the scene creation and real-time rendering capabilities of BGE, and augmenting them with easy-to-setup support for CAVE or Powerwall-type displays. The framework is implemented in a distributed manner, and it contains a virtual camera setup to control the display output, and a lightweight network communication protocol to manage events and synchronize the application state. We demonstrate that, with BlenderCAVE, 3D applications can easily be written in Blender and displayed on multi-screen displays.

1. Introduction

Today, virtual reality (VR) is present in many and very diverse application domains. VR developers are expected to have expertise not only in computer graphics (CG), but also in the problem domain and in the development of sophisticated software systems. Such systems may require handling multiple processes, message passing, dynamic memory management, and a variety of process synchronization techniques. But, due to the hardware aspects of VR, developers may also be concerned with low-level issues such as device drivers for particular I/O devices or techniques to generate multiple stereoscopic views. In addition, VR applications involve an artistic component, specially in order to improve the realism and immersion of the virtual worlds generated. Therefore, they also need the contribution of artists who model, texture and animate virtual characters and objects. To ease the job of VR developers, it is desirable to integrate content creation tools, CG engines, and hardware abstraction layers that make the development independent of specific I/O devices, and also allow preliminary testing on devices different than the ones of the final product.

The needs of VR developers have already been answered to a large extent by existing tools. Multiple rendering engines [OGR, OSG, Epi, Cryb] allow the creation of highly realistic yet real-time visualization applications without low-level knowledge of the most advanced CG algorithms. VR device abstraction frameworks [BJH*01, THS*01, Ant09] ease I/O services, communications and process management aspects. Finally, virtual sandbox tools allow content creation and the definition of the application's logic in easy-to-use visual editors. However, developers still miss tools that close the complete creation pipeline, allowing an automatic deployment of an application designed on a sandbox onto the final VR hardware.

Figure 1: *Two students play a collaborative game on a CAVE. The video game (both art and logic) was quickly created using the visual editing tools of Blender. Then, our easy-to-integrate BlenderCAVE framework manages a distributed rendering architecture based on the Blender Game Engine that generates the video output for all screens in the CAVE.*

This work introduces a framework, which we refer to as BlenderCAVE, for the easy development of multi-screen VR applications on a virtual sandbox. After an evaluation of existing content editing tools and CG render engines (detailed in Section 3), we have opted for the Blender Game Engine (BGE) [Blea] as the base engine for our framework. Then, the framework includes two major components. First, a simple virtual camera setup, described in Section 4, defines the content to be output on each screen. Second, as described in Section 5, a distributed architecture and a lightweight com-

munication protocol manage the application state and synchronize the output on the various screens.

We demonstrate BlenderCAVE through several applications displayed on an immersive CAVE projection system [CNSD93]. The highlight of the applications is a multiplayer videogame (shown in Fig. 1), which was quick-to-design and easily ported from a desktop environment to the CAVE.

2. Related Work

Large-area displays provide a high degree of immersion in VR applications, often making the VR experience more compelling to the user. Large-area displays can typically be achieved in two ways. One way is to project the image onto a large passive screen using a powerful projector, such as in a PowerWall, a CurvedScreen, a Workbench [KBF*95] or subsequent similar designs. Another way is to tile multiple smaller displays. In the first way, the difficulty is to achieve high resolution images, while in the second way the difficulties are due to color and brightness consistency [Sto01]. Tiled displays are also more expensive and require more computational resources. A CAVE system [CNSD93] shares some similarities with both approaches, as it uses multiple projectors to project images on several passive screens and give the user a sense of full immersion in a virtual environment.

The output display is just one of the hardware components involved in the design of a VR application. Typically, VR applications also involve various input devices, and possibly non-visual output. Multiple researchers have developed solutions based on abstraction layers that free the developer from dealing with implementation details about I/O devices. An early example is CAVELib [CN95], created by the inventors of the CAVE and later turned into a popular commercial tool. CAVELib's major limitation is its strong connection to specficic hardware configurations. A more recent and active VR design framework is VRJuggler [BJH*01]. It provides excellent support for I/O and it is managed at a programming level. There are many other VR development frameworks that provide abstraction layers, such as DIVERSE [KSA*03], Studierstube [SRH03], RB2 [VPL90], DIVE [CH93], dVS [Gri91], Avocado [Tra99], VRPN [THS*01], Equalizer [Equ], MR-Toolkit [SLGS92], or dVise [Ghe97]. A more recent framework, INVRS [Ant09], extends traditional abstraction capabilities to networked virtual environments.

Some VR authoring frameworks also provide easy-to-use graphical interfaces for application development. AMIRE [GHR*02] is an authoring tool for mixed reality, whose interesting aspect is that the design of the virtual environment may be performed in an intuitive manner in a mixed reality setting. EAI' Wordltoolkit [Ini97] enables novice developers to quickly prototype an application, however, it expects the application data to come from CAD packages. Finally, the commercial framework EON Studio™ [Eon], from EON Reality Inc, allows authoring through a graphical user interface. It is a complete package for a CAVE environment, however it lacks flexibility for developers.

In the creation of a VR application, modeling and rendering the virtual environment play an important role as well. There are many high-quality options for content creation and rendering, and we discuss some of the most relevant ones in detail in the next section.

3. Requirements and Selection of the Base Engine

Our approach to the design of a framework for easy development of multi-screen VR applications has been to augment one of the many existing high-quality CG render engines with functionalities to efficiently support multi-screen projection. In this section, we discuss the desired features of the base engine, we compare several high-quality engines, and we present the particular features that steered our decision toward Blender Game Engine (BGE).

From the CG point-of-view, the engine should support state-of-the-art rendering and animation features: programmable shaders, animations based on bones, simulation of rigid and deformable bodies, generation of stereoscopic video output, and extensibility through plug-ins and scripts.

On top of these features, it is desirable if the render engine includes an integrated 3D WYSIWYG scene compositor, i.e., a sandbox. We look for a solution that allows modeling, texturing and animating directly the objects that will be included in the VR application.

Other desirable features include multi-platform availability, as well as the possibility to execute the VR application on distributed heterogeneous systems. Source code access enables the possibility to implement new capabilities when necessary, and a well-stablished community of developers and artists is a good indication of further evolution of the engine, thus favoring a longer life-cycle of the VR application.

In our search for an engine that fulfills all or most of the desired features, we have evaluated in detail the following list of high-quality render engines: Unreal 3™ [Epi] from Epic Games Inc, CryEngine 2™ [Cryb] from Crytek GmbH, EON Studio™ [Eon] from EON Reality Inc, and the open source engines Ogre 3D (v. 1.6.5) [OGR], and BGE (v. 2.49) [Blea]. Other possible engines that we have considered are: proprietary game engines such as Unity™ [Unib], from Unity Technologies, Id Tech 5™ [Id] from Id Software, and Unigine™ [Unia] from Unigine Corp, or open source engines such as OpenSceneGraph [OSG], Crystal Space [Crya], Irrlich 3D Engine [Irr], and Id Software's Quake IdTech 3 [Id].

Our list is clearly not comprehensive, but we believe that it covers a set of highly representative engines. We chose Unreal, CryEngine, EON, Ogre and BGE for detailed evaluation for various reasons. Unreal and CryEngine are high-end engines used for top-class commercial video games, which is a good indication of their quality. EON, on the other hand, is an engine particularly oriented to multi-screen VR setups. And, finally, Ogre and BGE offer high-quality CG with the addition of open-source advantages. Moreover, BGE provides a content creation framework that could greatly ease application design. Table 1 compares these five engines based on CG quality features, integrated content creation and control possibilities, and further extensibility.

After the evaluation, we decided to select BGE as our base engine. The main reason is its interesting balance of high-quality render engine with integrated content creation and

	Unreal 3	CryEngine 2	EON Studio	Ogre3D 1.6.5	BGE 2.49
Stereoscopy	yes	no	yes	experimental	yes
Multi Screen	no	no	yes	no	tiled window
Shaders	Cg	Cg	no	Cg / GLSL	GLSL
Animation Support	external	external	external	external	included
Physics	PhysX	PhysX	no	several	Bullet
Logic Graphs	KissMet	Lua	yes	no	Python / bricks
Scripting	KissMet	Lua	VBScript / JavaScript	no	Python
Modeller	external	external	external	external	included
Scene Compositor	UnrealEd	CryEngine Sandbox	external / graph	external / code	included
Plugins	yes	yes	yes	yes	yes
Source Code	no	no	no	yes	yes
License	proprietary	proprietary	proprietary	GPL	GPL

Table 1: *Comparison of render engines based on their capabilities.*

sandbox. Another positive feature is its ease of extensibility. It contains a large API based on Python, and it allows the connection of specific device drivers by implementing a binding between the driver's library (as long as it is written in C/C++) and a Python class. Moreover, source code access allows the implementation of additional capabilities.

Given our target application, i.e., creation of VR applications for multi-screen displays, we also pay special attention to the capabilities of BGE in terms of stereoscopic and multiple video output. In its standard version, BGE supports five built-in stereoscopy modes (Pageflip, Syncdouble, Anaglyph, Side-by-Side and VInterlace), which can be toggled using a GUI control [MSO*09]. BGE has single-window output, but it is possible to switch a built-in mode and draw many windows on the same screen in a tiled manner, or direct each window to a different screen on a multi-screen system. At first, this feature seemed attractive for our application, but it does not scale well as the number of windows increases. As it will be described later in Section 5 we discarded BGE's built-in multi-screen functionality, and we designed a distributed architecture.

4. Virtual Camera Setup

Our BlenderCAVE framework includes two main components, a virtual camera setup that defines the output for each display, and a distributed architecture to manage the application. This section describes the virtual camera setup, including the definition of camera frustums, and a master-slave navigation approach.

4.1. Configuration of Camera Frustums

Given a VR scene and a target multi-screen display, we define a *Virtual Camera Cluster* (VCC) that associates one virtual camera to each screen. The correct compositing of the images on the multi-screen display requires a careful selection of parameters for each camera and a synchronized transformation of all cameras as the user navigates through the scene. Our prototype implementation is limited to planar screens, and then the configuration of each camera reduces to adjusting the values of ModelView, Projection and Viewport transformations. BGE includes five additional video output modes for dome-shaped screens [Bleb], which would allow supporting also dome-shaped multi-screen displays.

The VCC maps the geometry and topology of the projection system to the camera frustums on BGE. All the cameras of the VCC are located at the same point, called *local origin*, which corresponds to the position of the observer in the VR scene. If the VR installation includes a tracker of the observer, its position is mapped to the local origin. The four corners of the near planes of the various cameras are configured (up to a scale factor) with positions and orientations that respect the relative transformation between the observer and the screens in the real world. Then, the local origin and the corners of the near planes define the perspective angles of the cameras frustums. With this approach, the images captured by the various cameras correctly match at the borders of the screens, the union of the frustums covers all the visible volume in the VR scene, and the frustums do not intersect. Fig. 2 shows three possible configurations of the VCC for the particular type of configurable 4-wall CAVE used in our experiments.

4.2. Master-Slave Navigation

We assume that the physical screen setup remains invariant while the VR application is in use, therefore, the relative transformation between the various camera frustums depends only on the local origin, i.e., the observer's position w.r.t. the physical setup. Based on this observation, we perform camera navigation in a master-slave manner, computing the transformation of a master camera based on the user's navigation, and then computing the transformation of the slave cameras relative to the master. In our experiments, we have selected the frontal camera as master camera. In BGE, the master-slave VCC is programmed as a hierarchy, represented schematically in Fig. 3, which includes the user, the master camera, and the slave cameras.

In our prototype implementation, we use a first-person navigation mode. In the VR scene, the user is represented as an invisible human-height character contained on a simple bounding box. This bounding box constitutes the *user entity* in BGE. Each camera in the master-slave VCC has an additional entity, and they are all connected in a hierarchical manner to the user entity.

During navigation, the user moves and orients the virtual workspace through the VR scene, and these transformations are applied to the user entity. Then, the camera transformations are computed automatically based on the tracked local

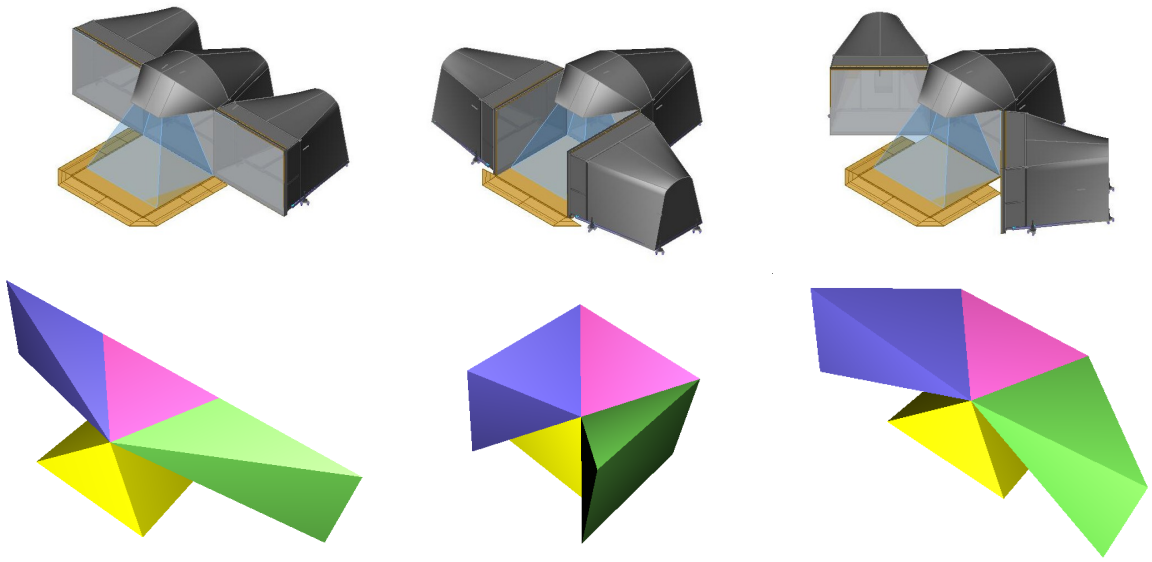

Figure 2: *From left to right, three possible setups for our reconfigurable 4-wall CAVE: wall, cube, and amphitheatre. The top row shows the screens and the mirror-based projection system. The bottom row shows frustum setups for an observer located at the center of the CAVE. If the observer's position is tracked, all four frustums need to be dynamically reconfigured, otherwise only the right and left frustums need to adapt to the CAVE's configuration.*

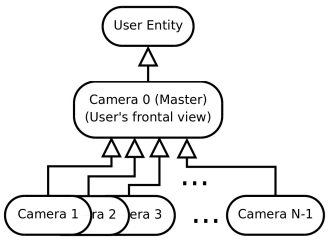

Figure 3: *BGE model of the master-slave camera hierarchy. The master camera is connected to the user entity, and all other cameras are defined relative to the master camera.*

origin. User navigation could be controlled in various ways: using a mouse, a keyboard, a wiimote, etc. Additional controls or keys can be assigned to gear other motions of the virtual character, such as jumping, crouching, or leaning. BGE also handles reactive collisions with the environment during camera navigation, using the bounding box of the user as collision primitive.

5. Communication System Architecture

Given a VR application designed on BGE, the target multi-screen display system, and the VCC that defines the camera-screen correspondence, we have designed a distributed rendering architecture that controls the video output of each screen. In this section, we describe the elements that conform this architecture and their main features, paying special attention to the following issues: maintaining a consistent application state across all elements, synchronizing the video output on all screens, and responding to external inputs (i.e., managing input peripherals).

5.1. Master-Slave Distributed Architecture

To compute the video output for the multiple screens, we have designed a (possibly heterogeneous) distributed architecture, with one PC per screen. On each PC, we execute one instance of BGE, which computes the image as seen from one of the cameras in the VCC, and outputs it to the corresponding screen.

To synchronize the rendered output, we set a common refresh rate on all BGE instances. The refresh rate is maintained both at the application level, i.e., for logic and physics updates, and at the GPU render level. Even though our architecture supports the use of heterogeneous PCs for the various screens, in our prototype implementation the refresh rate is limited by the slowest machine. For future versions of the framework, one could consider balancing the rendering load among the various PCs in a more efficient way.

We manage the application state and handle peripheral inputs following a master-slave approach. The PC in charge of rendering the master camera in the VCC plays the role of master in our architecture, and it communicates state changes and user input to the other PCs. BGE provides simple tools to program the application logic as a state machine. In particular, it provides logic bricks that react to events, and these logic bricks act on the application state. Events may be produced by internal logic conditions or by input devices.

5.2. Communication Protocol

We consider two different communication modes in our master-slave architecture: normal operation and initialization. During normal operation, the master handles events produced internally by its own application logic as well as events produced by input peripherals. If an event is triggered, the master communicates this event simultaneously to all slaves. When a slave receives a packet, it updates its local

version of the user entity in the VCC, computes the new position and orientation of its render camera, and a script routine triggers the event and executes its corresponding logic bricks. All slaves execute the same application logic, therefore by reacting to the same event as the master, and given that they maintain the same refresh rate, all local copies of the application state remain synchronized.

In our experiments, we used as peripherals standard keyboards and mice. Then, the information to be communicated by the master consists of: the position and orientation of the user entity, the orientation of the master camera, the keys that have been pressed or released, and the current position of the mouse. This information can be coded in very short messages. In our CAVE, the PCs are connected by less-than-half-meter-long gigabyte ethernet cable, which allowed us to send messages using UDP Multicast without package loss. The combination of small messages, fast network, and little protocol overhead, produce negligible system latency, as discussed in detail in the next section. The communication protocol could be extended to handle other types of events, such as random events or large state modifications due to physically based simulations. Given the small message size and negligible latency in the current prototype, there is plenty of room for additional communication payload.

The initialization mode is executed when a new slave PC joins the system. In this situation, the slave PC sends a request for a full-state update to the master, and the master responds with a standard UDP message containing the full state information. In addition to camera settings and input events, the full state may contain information such as the configurations of all moving objects in the scene, clip and frame numbers for animated characters, internal attributes and flags, etc. Moreover, at initialization, a slave needs to identify the particular camera in the VCC that it should render. We solve this issue by assigning to each slave camera in the VCC an *id* corresponding to the network address of the slave PC in charge. Then, a slave can discriminate its camera information simply by comparing the camera *id* with its own network address.

In our experiments, discussed in detail in the next section, the application did not suffer synchronization issues. In applications with a complex logic, however, it might be convenient to execute periodic full-state synchronizations.

The communication protocol is executed by BGE as a Python script. This guarantees a complete transparency of the communications across platforms, and enables the use of heterogeneous machines, with different operating systems. In our tests we have combined nodes running Microsoft Windows and Ubuntu Linux with no problems. All the packages are coded using a Python dictionary format, and we use the cPickle library to serialize Python objects to plain text and viceversa.

6. Implementation and Experiments

In this section, we describe first our CAVE-like installation and other hardware details. Then, we discuss the process for setting up a VR application using BlenderCAVE. Finally, we discuss the test applications we have implemented, as well as performance results.

Figure 4: *Mountain scene (*15727 *triangles) displayed on a CAVE. The scene is rendered with shadow mapping, multi-texturing (9 layers), 4 normal mapping passes, and screen-space ambient occlusion.*

6.1. Our Visualization System

The visualization system used to test BlenderCAVE is a RAVE II (Reconfigurable Advanced Visualization Environment), a CAVE-alike system with four screens designed by Fakespace Inc. The main difference with a conventional CAVE system is that the side screens of the RAVE can be reoriented to create different configurations of the immersive space (see Fig. 2). Each display module has a screen of dimensions 3.75 x 3.12 meters.

The displays use active stereo projectors and CrystalEyes shutter stereoscopic glasses. The projectors are driven by a cluster of 4 PCs with NVIDIA Quadro FX 4500 graphics cards, and connected through a Gigabyte Ethernet network. The PCs also carry GSync hardware to synchronize the pageflip on all the graphic cards. The cluster can execute Windows XP or Ubuntu Linux, both of which have been used on BlenderCAVE tests.

6.2. Setting up and Running BlenderCAVE

BlenderCAVE is programmed as a set of scripts that control the VR application logic on BGE. Given a certain VR application on BGE and a cluster of PCs that send video output to a multi-screen display, setting up BlenderCAVE to drive the multi-screen display is an extremely easy task. First, one needs to include the VCC hierarchical entity on every instance of the VR application. The BlenderCAVE scripts are associated to the VCC, hence they are automatically included. Note that the camera settings of the VCC should be adjusted to match the specific display, as described in Section 4.1. If a display installation is permanent, then the VCC may be defined only once and imported in multiple applications. The accompanying video shows a tutorial that describes the creation of a VCC template and the inclusion of the VCC template and its associated communication scripts in an application. Setting up BlenderCAVE to drive our CAVE system takes less than two minutes once the VCC is defined.

Additionally, one needs to set the camera *ids* for the var-

ious PCs, as described in Section 5.2, and adjust the basic rendering settings of BGE (i.e., fullscreen rendering and activation of the pageflip option). In our examples, we used a 1024×768 resolution for each screen and a refresh rate of 100Hz, but higher resolutions and refresh rates are supported.

6.3. Test Applications

We have tested BlenderCAVE on three different VR applications. Two of these applications, the mountain scene in Fig. 4 and the shark scene in Fig. 5, intend to demonstrate the easiness to create applications with high-quality graphics and render them on a CAVE. The third application, the game in Fig. 1, gives a glimpse of the great possibilities for CAVE-oriented application development. In all the images shown in the paper, as well as in the accompanying video, stereo output was disabled, but the system runs in full stereo mode.

The mountain scene in Fig. 4 is composed of 15727 triangles and shows dynamic shadow computation using shadow maps as the sun rises and sets. We used a high-res shadow buffer of 2048×2048 pixels. More interestingly, the mountains are render using multi-layer texturing, with 9 texture layers, 4 simultaneous normal mapping passes, and an additional pass of screen-space ambient occlusion.

The sharks in Fig. 5 are composed of 7502 and 5197 triangles each, and are animated using bones and a skinning technique. Both sharks are rendered using a diffuse texture and a pseudo environment map.

Fig. 5 also demonstrates the effectiveness of our VCC camera setup. Notice how the images of the sharks are projected onto the seams and corners of the CAVE, and there is barely any noticeable distortion. In this example, the location of the physical camera is being used as local origin for the VCC. Please see the accompanying video for a dynamic demonstration.

Our last test scene is a collaborative videogame, shown in Fig. 1, where two users fight against a group of zombie skeletons. The floor is rendered using normal mapping, and the skeletons (8609 triangles each) are animated using bones and predefined animation clips. The application maintains the target frame rate (50Hz, stereo) with up to 15 skeletons, for a total of 135553 triangles in the scene.

The videogame was initially designed as a single-player game using the Blender content creation tool, with logic bricks and Python scripting. The major result proved with this scene was that BlenderCAVE allowed extremely simple adaptation of the videogame to a multi-screen display, i.e., our CAVE. Porting the videogame to BlenderCAVE required the definition of the VCC (approximately 1 hour, including tests, but this needs to be done only once if the configuration of the CAVE is static), adding the BlenderCAVE scripts to the application (done in just 1 minute), and, of course, installing the application in all PCs in the architecture.

We have also measured the communication latency and bandwidth in our system. Thanks to the use of UDP Multicast and our event-driven protocol, the network traffic is very low. We measured the total traffic from the master to all slaves in situations with frequent camera motion and button-click events, and we reached peak packet sizes of just 12kB. To measure network latency, we timed the round trip of a packet between the master and a slave, which peaked at just 16μs. As a conclusion, with our Gigabyte Ethernet network, communication latency is not an issue.

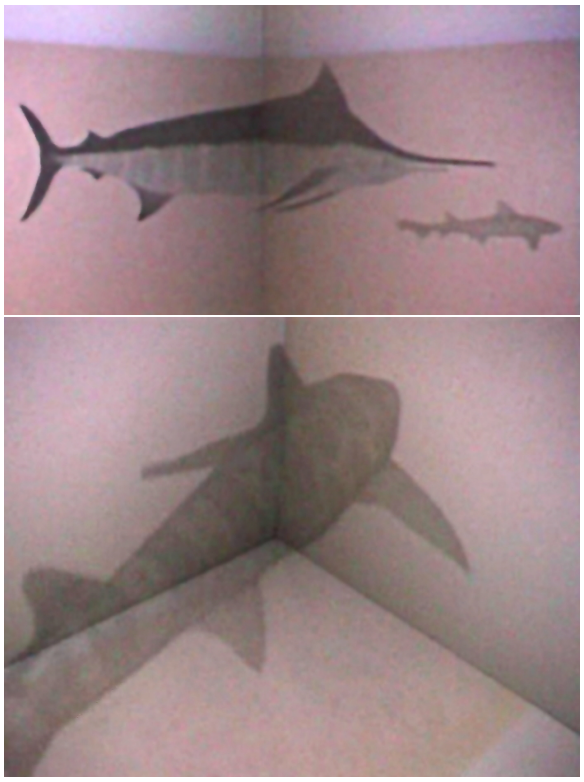

Figure 5: *Shark scene displayed on a CAVE. Notice the lack of distortion as the images of the sharks are projected onto the seams and corners of the CAVE, demonstrating the effectiveness of the VCC setup.*

7. Discussion

In the past, the creation of a multi-user VR application for a CAVE entailed the integration of I/O peripherals in the render engine, setting up and coordinating multiple instances of the render engine to drive all screens of the display, and importing art content in the rendering application. This paper shows that the BlenderCAVE framework allows a much simpler development of complex and interesting VR applications for a CAVE-like display. Using BGE as base render engine, and taking advantage of Blender's content creation possibilities, BlenderCAVE augments the engine to easily direct the video output to a multi-screen display.

There are, however, multiple directions in which the features of BlenderCAVE could be improved or extended. For more general VR applications, it will be necessary to provide support to many I/O peripherals. This can be done by integrating one of the existing VR libraries for hardware abstraction, possibly through Python-based extensions.

As discussed in the paper, the current communication protocol is particularly efficient for event-driven state changes, but applications with a complex state, such as physically based simulations, may require modifications. Under a complex state, there is a trade-off between distribution of state computations, communication of state updates, and network

bandwidth. The current framework is also limited in terms of the tight connection between BGE instances and output screens. Currently, each screen is driven by a different BGE instance, running on a different PC. For tiled displays, it might be convenient to distribute rendering load differently, perhaps with the same machine driving several displays. The critical factor should be the minimization of the computing resources, subject to fulfilling the desired refresh rate, which makes the problem application-dependent.

Last, BlenderCAVE could be extended with features that would increase the rendering quality on the CAVE. Such features include color and brightness correction for seamless image continuity across the screens.

BlenderCave source code, templates and examples can be downloaded from:

http://www.gmrv.es/~jgascon/BlenderCave/index.html

Acknowledgements

This project has been supported in part by the Spanish Ministry of Science and Innovation (project TIN2009-07942). The authors also would like to thank Martinsh Upitis ('martinsh' from BlenderArtists.org), author of the mountain scene, and the GMRV group at URJC Madrid.

References

[Ant09] ANTHES C.: *A Collaborative Interaction Framework for Networked Virtual Environments*. PhD thesis, Institute of Graphics and Parallel Processing at JKU Linz, Austria, Institute of Graphics and Parallel Processing at JKU Linz, Austria, August 2009.

[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), pp. 89 –96.

[Blea] BLENDER: Blender Game Engine Features website. http://www.blender.org/education-help/tutorials/game-engine/.

[Bleb] BLENDER: Dome mode in blender game engine. http://wiki.blender.org/index.php/Dev:Source/GameEngine/2.49/Fisheye_Dome_Camera.

[CH93] CARLSSON C., HAGSAND. O.: Dive - a platform for multi-user virtual environments. In *Computers and Graphics* (1993), pp. 663–669.

[CN95] CRUZ-NEIRA C.: *Virtual Reality Based on Multiple Projection Screens: The CAVE and its Applications to Computational Science and Engineering*. PhD thesis, University of Illinois at Chicago, Department of Electrical Engineering and Computer Science, 1995.

[CNSD93] CRUZ-NEIRA C., SANDIN D. J., DEFANTI T. A.: Surround-screen projection-based virtual reality: The design and implementation of the cave. In *T. Kajiya, editor, Computer Graphics (SIGGRAPH 93 Proceedings* (1993), pp. 135–142.

[Crya] CRYSTAL SPACE: Crystal Space website. http://www.crystalspace3d.org/main/Main_Page.

[Cryb] CRYTEK GMBH: Crytek CryEngine Sandbox. http://www.crytek.com/cryengine.

[Eon] EON REALITY INC: *Eon Studio Reference Manual*.

[Epi] EPIC GAMES INC: Epic Games Unreal Engine. http://www.unreal.com/.

[Equ] EQUALIZER: Equalizer: standard middleware to create and deploy parallel OpenGL-based applications. http://www.equalizergraphics.com.

[Ghe97] GHEE S.: Programming virtual worlds. In *ACM SIGGRAPH 97 Conference, Los Angeles* (1997).

[GHR*02] GRIMM P., HALLER M., REINHOLD S., REIMANN C., ZAUNER J.: Amire - authoring mixed reality. In *Proc. of IEEE International Augmented Reality Toolkit Workshop* (2002).

[Gri91] GRIMSDALE. C.: dvs-distributed virtual environment system. In *In Proceedings of Computer Graphics 1991 Conference* (1991).

[Id] ID SOFTWARE: Quake IdTech 3. http://www.idsoftware.com/.

[Ini97] INITION: *Sense8 WorldToolkit R8 Reference Manual*. 1997.

[Irr] IRRLICHT: Irrlicht Engine website. http://irrlicht.sourceforge.net/.

[KBF*95] KRÜGER W., BOHN C.-A., FRÖHLICH B., SCHÜTH H., STRAUSS W., WESCHE G.: The responsive workbench: A virtual work environment. In *IEEE Computer* (1995), pp. 42–48.

[KSA*03] KELSO J., SATTERFIELD S. G., ARSENAULT L. E., KETCHAN P. M., KRIZ R. D.: Diverse: A framework for building extensible and reconfigurable device-independent virtual environments and distributed asynchronous simulations. *Presence: Teleoperators and Virtual Environments 12*, 1 (2003), 19–36.

[MSO*09] MARTÍN S., SUÁREZ J., OREA R., RUBIO R., GALLEGO R.: Glsv: Graphics library stereo vision for opengl. *Virtual Reality 13* (2009), 51–57. 10.1007/s10055-008-0105-y.

[OGR] OGRE: OGRE-Open Source 3D Graphics Engine. http://www.ogre3d.org/.

[OSG] OSG: OpenSceneGraph website. http://www.openscenegraph.org/projects/osg.

[SLGS92] SHAWN C., LIANG J., GREEN M., SUN Y.: The decoupled simulation model for virtual reality systems. In *Proceedings of the ACM SIGCHI Human Factors in Computer Systems Conference* (1992), pp. pp.321–328.

[SRH03] SCHMALSTIEG D., REITMAYR G., HESINA G.: Distributed applications for collaborative three-dimensional workspaces. *Presence: Teleoperators and Virtual Environments 12*, 1 (2003), 52–67.

[Sto01] STONE M. C.: Color and brightness appearance issues in tiled displays. *IEEE Computer Graphics and Applications 21* (2001), 58–66.

[THS*01] TAYLOR II R. M., HUDSON T. C., SEEGER A., WEBER H., JULIANO J., HELSER A. T.: Vrpn: A device-independent, network-transparent vr peripheral system. In *VRST 2001 conference* (2001).

[Tra99] TRAMBEREND H.: Avocado: a distributed virtual reality framework. In *Virtual Reality, 1999. Proceedings., IEEE* (Mar. 1999), pp. 14 –21.

[Unia] UNIGINE CORP: Unigine: multi-platform real-time 3D engine website. http://unigine.com/.

[Unib] UNITY TECHNOLOGIES: Unity 3D engine website. http://unity3d.com/unity/.

[VPL90] VPL RESEARCH INC: *Reality Built for Two: RB2 Operation Manual*. 1990.