Generation of Shadows in Scene Graph based VR

Bjoern Kuehl
University of Hamburg, Germany
9kuehl@informatik.uni-hamburg.de

Kristopher J. Blom
University of Hamburg, Germany
blom@informatik.uni-hamburg.de

Steffi Beckhaus
University of Hamburg, Germany
steffi.beckhaus@uni-hamburg.de

Abstract

In this paper, we present our experience implementing shadows in Scene Graph based Virtual Reality systems. Shadows are an important part of the human perception of both shape and depth and, yet, are a largely missing component in Virtual Environments. In this work, we investigate extending standard Scene Graphs to automatically produce shadowed scenes. This paper presents our experience embedding two popular real-time shadow methods, Shadow Mapping and Stencil Shadow Volumes, in a popular Scene Graph system, OpenGL Performer. Our experience has shown both ways in which they can be used and also a number of weakness in the current state of both Scene Graph systems and shadow methods. Based on our experiences, we present suggestions for the user desiring to include shadows in their Virtual Environment and highlight areas, where further development would benefit users significantly.

Keywords: Scene Graph, Virtual Reality, Stencil Shadow Volumes, Shadow Mapping

1 INTRODUCTION

A well known German figure of speech says: "There is shadow where there is light." This omnipresent effect of light occlusion makes "the shape and relative position of objects in such scenes more comprehensible" [11] and helps us to identify the distribution of light sources. However, in the realm of computer graphics, this figure of speech does not necessarily apply, because this familiar effect doesn't appear automatically, but must be imitated. The extra effort is useful, because the usual uniform illumination of computer graphics is easy to identify as artificial, amongst other things due to "flying" objects.

The perceptual cues provided by shadows are presumably even more important in the field of Virtual Reality (VR). VR focuses on both immersive environments and providing the illusion of depth; Shadows deliver important information about depth to the user. Given these foci, it is remarkable that shadows are unusual to find in the most VR environments. Perhaps, the biggest reason for this is the lack of integrated support in the graphics generation tools that VR systems typically use, Scene Graphs; Scene Graph(SG) systems used have only just begun to support shadows.

In recent years, two techniques have emerged as standards in the real-time computer graphics community for shadows generation. The first, used in the majority of cases, is **Shadow Mapping**. [11] Shadow Mapping uses a texture of the scene generated from the lights point of view, in combination with a special algorithm, to determine the portions of the scene that are not visible from the light and, therefore, in the shadow. The second shadow technique is **Stencil Shadow Volumes**. [2] This method marks objects as lying within the shadow if they are contained within a volume produced by extending the silhouette of a shadow caster with respect to a light.

In this paper we present the results of our investigation of integrating automatic shadow generation into Scene Graph based VR. In our work we have focused on SGI's OpenGL Performer^{TM} [6] - a popular SG, used in numerous VR systems - and the VR system, AVANGO. [10] We present here how shadows, both Shadow Mapping and Stencil Shadow Volumes, are integrated into AVANGO and describe the implementation of them briefly. However, a large portion of this work's importance lies rather in the experience collected during this exploration. We present much of that experience here, in the form of a number of suggestions for the user of such a system. In this discussion we also bring to the forefront many of the shortcomings of the shadow techniques for general usage and describe a number of points, where improvements to the techniques would be advantageous.

In the next sections, we present background information on Virtual Reality and Scene Graphs, followed by a small overview of the Shadow Mapping and Stencil Shadow Volume methods. In Section 4 we present details on the implementation of both shadow techniques in a SG-based VR system. In Section 5, we present the outcomes of this work, including giving advise how to use automatic generation of shadowing and discussion of the shortcomings of both the SG implementation and the shadow techniques. Section 6 discusses future di-

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.
Copyright UNION Agency - Science Press, Plzen, Czech Republic.

rections for research we have identified that would improve shadow usage in the VR context.

2 VIRTUAL REALITY

In this section we provide some necessary background information on Virtual Reality, by contrasting it with other, more standard computer graphic areas. A special focus is placed on the generation of graphics in VR systems, performed by underlying Scene Graph (SG) systems. Finally, we give a brief overview of the related work in VR and similar areas on shadows.

VR, like computer games, builds highly upon the real-time computer graphics community. VR and computer games share many similarities; However, VR differentiates itself on a number of fronts. VR is highly focused on immersion, roughly the illusion of being present in the virtual environment. This is achieved through various means. For this feeling of presence, one factor is a high quality image. The "connectedness, continuity, consistency, and meaningfulness of the perceptual stimuli presented" [5] is denoted as "Pictorial Realism" and this realism can be increased by shadows.

The hardware normally used in VR can also increase this feeling of presence. Common components include surrounding (immersive) displays, such as large projection systems or Head Mounted Displays (HMDs), stereoscopic imagery. Another component that differentiates VR from computer games is that the user's position is usually tracked, having the user's physical movement affect the display of virtual environment. The resulting motion parallax effect "'provide important information about the depth of objects in the field." [5] Stereoscopic imagery and the paralax effect enabled with head tracking have significant effects on depth perception. These also have impacts on VR systems' structure. Large scale projection systems require multiple projections, implemented either with special multi-piped systems or implemented using distributed systems. Head-tracking leads most VR to use a "the world moves" metaphor instead of the common camera movement metaphor.

In contrast to computer games, VR exhibits two big disadvantages in regards to the modeled 3D environment. Among the VR community, it is uncommon to have dedicated modelers. Commonly, available models and models created by programmers and students with little or no experience must be used. A further factor is that many games, particularly those with shadows, take place within closed environments, such as buildings with small rooms. Conversely, the environments of VR are often very large and quite often consist of open spaces, both of which become critical factors when dealing with shadows.

2.1 Scene Graph Based Systems

A Scene Graph is used by most VR systems for organizing the scene. A Scene Graph (SG) is a directed, acyclic graph consisting of a hierarchical structure of nodes designed for the simplification of the display and management of graphical environments. SGs are typically tree structures, where the root node is a "scene node," on which the rest of the scene is hung. The nodes of the graph specify information about the scene and its properties, such as transformations, state properties, material properties and geometrical primitives. In this work we have focused specifically on OpenGL Performer™, referred to throughout simply as Performer. However, the work we present here is largely applicable, outside of implementation details, to the two other popular SGs in the VR community. A large majority of the VR systems use one of the three. OpenSceneGraph (OSG: http://www.openscenegraph.org/) is a large community project and has an API very similar to OpenGL Performer. OpenSceneGraph (OpenSG: http://opensg.vrsource.org/trac) is built on the same principles, but presents a highly different API.

To render the scene, the scene graph is traversed, rendering all visible objects. A **traverser** traverses the scene graph, typically originating from the scene node and proceeding to the leaf nodes, which contain the actual geometry. In between the root node and the leaf nodes are nodes that transform and change the scene layout. In Performer, the leaf nodes are of type **pfGeode** and contain one or more **pfGeoSets**, which hold the actual geometry primitives. To be able to position the geometry in the scene, the pfGeode is hung on a transform node, the **pfDCS** node. The final major component, **pfGeoState**, is responsible for influencing the graphical state of an object and is attached to individual pfGeoSet's.

The AVANGO VR system is built as a layer on top of Performer. AVANGO is an object oriented framework for creating distributed, interactive VR applications, with an interface that is similar to Open Inventor. [10] It supports using many different input devices and also supports various types of displays. As with most VR systems, AVANGO handles this by abstracting from the underlying devices. This abstraction is a boon to application writers, but also makes implementation of some advanced concepts difficult as access to necessary components may be denied.

The main components of AVANGO are the scene graph and an orthogonal dataflow *Field Container* concept. The state of a node or object is encapsulated in *Fields*. Fields can be connected to other fields to synchronize values between them. In this manner a dataflow graph is constructed, combining Performer derived nodes with pure AVANGO Field Containers.

To get the best performance possible, all classes in AVANGO are coded in C++. To achieve increased flexibility and run-time coding capabilities, a binding to **scheme** is built in, including an interpreter for run-time command execution.

2.2 Related Work

Literature in regards to shadows in VR is very limited. The only paper directly related to our work is early work from Slater et al. [7] Their research was based on a custom Shadow Volume system and was focused on presence research. Unfortunately, the results were discouraging, but presumably are due to the very primitive technology they used.

The related areas of Augmented Reality and Mixed Reality have been more focused on shadows, particularly dealing with mixed real and virtual content casting shadows. Both shadow generating method were used to in the related literature. For example, in [3] Stencil Shadow Volumes are used, while in [9] Shadow Mapping is used to create the shadows. These works focus highly on the underlying graphical manipulations. Prior work could not be found, in which shadows were integrated into scene graph systems. However, all three of the major SG systems have created some implementation of Shadow Mapping.

The gaming community is the one area that uses both Shadow Mapping and Stencil Shadow Volumes. Examples are commercial game engines and also state-of-the-art graphic engines, for example the popular **Ogre** (http://www.ogre3d.org/). Unfortunately, these engines are difficult to adapted to usage in VR systems, most notably because of multi-threading issues.

3 SHADOW TECHNIQUES

In this section, we briefly present the theoretical foundations of the two main shadow techniques used in real-time graphics, Shadow Mapping and Stencil Shadow Volumes. This overview is purposely general and implementation details will be handled as required in Section 4. Where appropriate, references to more thorough resources are provided.

3.1 Shadow Mapping

The most popular shadow technique, used by a large number of today's games and simulation applications, is Shadow Mapping. [11] It is based on the idea that all geometry visible from the light must be lit, and all geometry that can not be seen from the light's point of view lies in the shadow. Shadow Mapping is capable of casting shadows on arbitrary surfaces and also supports self-shadowing and inter-object shadows.

For Shadow Mapping, a two-pass algorithm must be used. In the first pass, the scene is rendered from the light's point of view. The depth values are then written to an already prepared depth texture. To use this texture in the second pass, which is from the view of the camera, it must be projected onto the scene from the light's point of view by calculating appropriate texture coordinates.

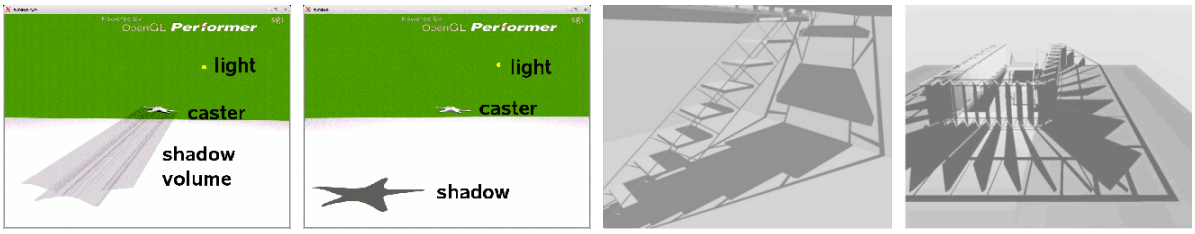
Using the generated texture coordinates, created using OpenGL's automatic texture coordinate generation, the distance between the fragment and the light source can then be compared with depth stored in the r component of the corresponding texel(s) of the shadow map. If both values are equal, then the corresponding fragment is the same as the one seen from the camera's point of view. However, if the r component of the fragment's texture coordinate is greater than the corresponding texel, then the fragment can not be seen from the lights point of view. This indicates that there is another object between the fragment and the light source. Therefore, this fragment lies in a shadow.

3.2 Stencil Shadow Volumes

The idea to use volumes to calculate shadows was introduced by F. Crow. [2] The volume that the shadow encloses is calculated for each light and object combination. It can then be used to determine what geometry is located inside the shadow. The silhouette of an object is formed from precisely the edges, for which one adjacent plane is facing the light and the other is facing away. To check if a plane is facing the light, one has to calculate the angle between the faces normal and the incoming light ray. If this angle is $> 90°$, the plane is facing toward the light source. To create a shadow volume, as seen in Figure 1(a), he extended the recovered silhouette by displacing copies of each silhouette's vertices in the direction opposite to the incoming light ray.

In order to use the later introduced GPU's inherit vertex processing power in calculating shadow volumes a new method had to be developed. Since vertex shaders are not capable of producing new vertices, objects have to be enhanced with new vertices, which can later be displaced by the vertex shader. To do that, all the object's edges are replaced by newly created quads. Two of the quad's vertices receive the normal of one adjacent face and the other two vertices the normal of other face. These quads are called "degenerate quads," since they don't contain any surface area at creation time. To create a shadow volume from the extended object, we use the vertex shader to displace all vertices that aren't facing the light source. Through the displacing process the degenerate quads are stretched so they get their own surface area. The created shadow volume consists of the newly emerged quads together with the displaced and the non-displaced triangles capping it.

To get shadows from the calculated shadow volumes, as seen in Figure 1, the counting capability of the stencil buffer is used to find these objects. [4] Heidmann



(a) A scene with a visible shadow volume (b) the shadow is visible, where the volume intersects geometry (c) stencil shadows in a real environment (d) stencil shadows in a real environment

Figure 1: Stencil Shadow Volumes

followed Crow's method for determining which objects are shadowed, by counting the order of the objects and volumes surfaces, today known as **z-Pass** or depth pass. It is a method, in which the front and back faces are successively rendered. If a fragment of the front faces can be drawn, the associated stencil value is incremented. Afterwards, the back faces are rendered and, for every fragment that isn't covered by another object, the associated stencil value is decremented. Fragments with an associated stencil value > 0 after all faces are rendered indicate that there is an object within the volume; Therefore, it lies in the shadow. After completing this step for all fragments, the stencil buffer contains values that divide the scene into lit and shadowed regions.

Unfortunately, z-Pass inverts the lit and shadowed regions if the camera is located inside the shadow volume. To avoid this problem, the order of operations can be reversed, known as **Carmack's reverse** [1] or simply as **z-Fail**. Using this method, the stencil buffer must be incremented, if it was not possible to draw the volume's back face, and decremented, if the front face cannot be rendered. In contrast to z-Pass, it is necessary to cap the volume, making z-Fail a bit slower. The results are almost the same, but enables entering a shadow volume, without producing inverted shadows.

There are two ways in which the stencil buffer's mask can be used to render shadows. One can use either "modulative shadows" or "additive light masking." Using modulative shadows, we overlay the scene with a quad, colored with a semi-transparent shadow color. Doing this exclusively in the regions masked in the stencil buffer, the scene's color is darkened inside the shadowed region. This method is comparatively fast and easy to use, but can produce errors within the shadow if there is specular highlighting inside the shadowed region. The "additive light masking" approach produces more realistic shadows and handles highlights the correct way, but it is noticeably slower, since the image must be drawn twice. In the first pass, the scene is rendered only with weak ambient light. After determining the shadowed regions, the scene is rendered again to the regions outside the shadowed regions with fully enabled lighting.

4 IMPLEMENTATION

The goal of our work was to make standard shadow methods available in the scene graph based VR system AVANGO, such that they were automatically created throughout the entire scene. This goal was based on an outsider's view that the shadow methods were, at this point "plug 'n play," and it should be just a simple matter of adapting the processes to work with SG based systems. In this section we present the implementation details of how both Shadow Mapping and Stencil Shadow Volumes can be implemented into SG based VR systems, specifically OpenGL Performer and AVANGO. In the following section we discuss the results of our implementations.

4.1 Shadow Mapping

Integrating Shadow Mapping into a VR system isn't necessarily easy, due to it being implemented as a two pass algorithm. While the first pass is only necessary when the light sources moves, VR systems use a "the world moves" metaphor instead of moving the camera, which means the light sources move every frame.

In Performer multi-pass algorithms can be implemented using two or more **channels**, conceptually a view of the scene that is part of the underlying "pipe." In many VR systems, including AVANGO, the underlying windowing, Performer's pipes/channels, is hidden from the user. This makes it difficult to create a second perspective needed for the Shadow Mapping method, as the underlying VR system must be modified.

Fortunately, it turned out that creating our own Shadow Mapping implementation wasn't necessary. Our initial investigations on integrating Shadow Mapping lead us to look at the Performer's own implementation, which was supported officially only under Windows and AVANGO functions only under Linux, to see how it was dealt with there. This investigation showed that the Performer method, supported through the class **pfLightSource**, did indeed function under Linux as well. Instead of reimplementing Shadow Mapping ourselves, we proceeded with the version available and describe here how it functions and how it is then integrated into AVANGO.

These light sources are SG nodes, which means that the position, from where the scene is lit, is determined

by placing them in the SG. Because Shadow Mapping doesn't support omnidirectional shadows, it is important to specify the light's direction. The field-of-view (*fov*) of the light source is an important factor affecting the quality of the shadow; A smaller *fov* can better take advantage of the texture resolution, which results in smoother borders of the shadow. For the precision of the shadows, the two clipping planes must lie as close together as possible. In AVANGO, these configurations can be changed at run-time, since all attributes of the class fpLightSource are represented as fields. The following code snippet shows the sequence of scheme scripting commands required to use Performer's Shadow Mapping in AVANGO:

```
(define newLight (make-instance-by-name "fpLightSource"))
(fp-set-value newLight 'Name "newLight")
(fp-set-value newLight 'Position (make-vec4 0 0 0 1))
(fp-set-value newLight 'SpotDir (make-vec3 0 0 -1))
(fp-set-value newLight 'SpotCone (make-vec2 0 180))

(fp-set-value newLight 'ShadowEnable 1)
// resolution of the shadow texture:
(fp-set-value newLight 'ShadowSize 512)
(fp-set-value newLight 'ShadowFrustum
          (make-vec4 -0.5 0.5 -0.5 0.5))
(fp-set-value newLight 'ShadowNearFar (make-vec2 0.01 100))
```

4.2 Stencil Shadow Volumes

As Stencil Shadow Volumes are found neither in Performer nor AVANGO, we describe here all of the implementation details from our implementation. We extended AVANGO with a new node, "fpStencilShadow," which is directly inherited from the node fpDCS. The realization of this extension, as a transform node, makes it capable of being positioned within the scene graph and also has the advantage of collecting all the required objects for this method under one node. The task of this node class is to extend loaded objects by degenerate quads and manipulate the way the underlying geometry is drawn.

To extend an object by new quads, it must be committed to the extension's most important field, "Caster." This field holds a node or complete sub-tree, allowing the inclusion of any geometry set consisting of triangles or triangle strips. When the field update mechanism finds that the Caster field has received a new object, a function is called that creates a custom recursive pfuTraverser, which traverses the object of the field, searching for geometry nodes. When a geometry node is found, the contained GeoSets are extracted to create the new quads from them. Since GeoSets can contain only one primitive type, the new quads are put into a newly created GeoSet.

To create the shadow volume, the newly created GeoSet and a copy of the original one are placed as children of two newly created geometry nodes. This is done, so that the front and back faces can obtain different Draw-Traverser functions, describing the different behaviour for passing or not passing the depth test. These two nodes are then attached to the instance of the fpStencilShadow class.

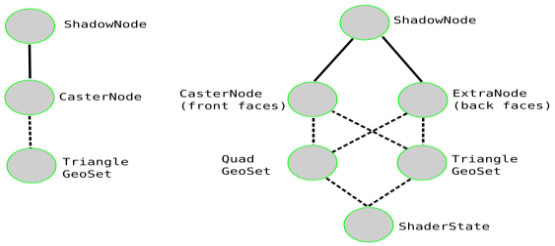


Figure 2: An object consisting of one triangle-geoset is hung under the ShadowNode. After modification performed by the traverser, another geode is created referencing both the triangle and the quad geosets.

The resulting Scene Graphs can be seen in Figure 2. In order to be able to displace the vertices of the new GeoSets later, all GeoSets have a special GeoState that is comprised a shader program that displaces vertices, dependant on the light's position. Further state requires are discussed later.

In Performer, it is possible to use OpenGL commands directly, so we can change the systems state. Unfortunately, changing the systems state this way is critical, because the changes are hidden from Performer. This means that Performer acts on the assumption that the state isn't touched, so completely unwanted effects can occur. That is why we have to save the OpenGL state on our own, so we can undo the changes later.

To be able to undo these changes, the first command stores the current state. In the Post-Draw-Function, this saved state can be loaded again. In order to perform the depth test without changing the depth entries, the depth test is enabled and the associating mask is "false." The Pre-Draw-Function shown below is designed for the front faces, so the back faces are culled away. Now, the stencil test can be enabled and configured, such that it is incremented when it is possible to render the current fragment. Using z-Pass, the Pre-Draw-Traverser for the front faces follows:

```
int fpStencilShadow::pfdPreDraw1(pfTraverser *trav, void *data)
{
        glPushAttrib( GL_ALL_ATTRIB_BITS);
        glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);
        glEnable( GL_DEPTH_TEST);
        glDepthMask(GL_FALSE);
        glEnable( GL_CULL_FACE);
        glCullFace( GL_BACK);
        glEnable(GL_STENCIL_TEST);
        glStencilFunc(  GL_ALWAYS, 0, 0xffffffff);
        glStencilOp( GL_KEEP, GL_KEEP, GL_INCR_WRAP);
   return PFTRAV_CONT;
}
```

After attaching the shadow volumes to the scene graph, they are rendered, with the color buffer disabled. This is done in order to create a mask in the stencil buffer that separates lit regions from shadowed ones. As mentioned previously, there are two ways of using this mask for creating real shadows. We have implemented "modulative shadows," in order to avoid the multi-pass issues discussed in Section 4.1. Therefore, a node containing a quad is held in the field **ShadowQuad**. The ShadowQuad functions as a semi-transparent film, through which the whole scene can be

(a) 32x32 (b) 128x128 (c) 256x256 (d) 1024x1024

Figure 3: Depiction of a typical outdoor scene, a marketplace in Germany with different sizes of depth texture.

seen, darkening the shadowed regions. For this function to behave correctly, it is necessary that the film is always positioned between the camera and the scene. For this function to behave correctly, it is necessary that the film is always positioned between the camera and the scene. For that reason, a transformation node, above the geometry node, positions the film in front of the camera.

Furthermore, if the field that contains the ShadowQuad is modified, the field update mechanism calls a function that creates a new traverser. It then traverses the geometry subtree searching for geometry nodes to attach the Draw-Traverser functions to them. These functions configure the drawing of the ShadowQuad, such that it is drawn dependent on the values in the stencil buffer. In this manner, the film is laid over the scene only on the shadowed regions. In order to avoid having the shadow quad simply covering objects behind it, the Draw-Traverser activates the blending mode, darkening the shadow regions by blending its color with a black color whose alpha value is 0.5. The code of the Draw-Traverser function is as follows:

```
int fpStencilShadow::pfdPreDraw3(pfTraverser *trav, void *data)
{
        glPushAttrib( GL_ALL_ATTRIB_BITS);
        glColorMask(GL_TRUE, GL_TRUE, GL_TRUE, GL_TRUE);
        glEnable( GL_STENCIL_TEST);
        glStencilFunc(GL_NOTEQUAL, 0,  0xffffffff);
        glStencilOp( GL_REPLACE, GL_REPLACE, GL_REPLACE);
        glColor4f(0.0f, 0.0f, 0.0f, 0.5f);
        glEnable(GL_BLEND);
        glBlendFunc( GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
        return PFTRAV_CONT;
}
```

5 RESULTS

In this section, we discusses our experience of using the implemented shadow methods. We also provide the user with much of the information that they will require in order to successfully introduce shadows into their environment. We have divided this into three parts, handling of the two method individually and a short statement on using both methods simultaneously.

Shadow Mapping The Shadow Mapping routine works much as advertised, working with arbitrary surfaces and performing self-shadowing. As seen in the code in Section 4.1, it is easy to enable in AVANGO. It is, however, not a "plug 'n play" method. As will be discussed in the following paragraphs, there are numerous points to tweak to the individual scene. One of the main points that requires careful consideration from the user is the texture resolution vs. the area to which it is applied. The second area of importance relates to the depth comparison and its inherit inaccuracy. Luckily, the AVANGO implementation assists the user highly in this endeavour, as they can modify all relevant values at runtime using the scheme interface or a GUI interface.

Depending on the resolution of the shadow texture, the shadow is, more or less, blocky. Although it is possible to assign an arbitrary value for the resolution of the shadow texture, it must be a square of a power of 2 to be accepted as a valid value. To get a correct texture mapping, the texture resolution can be no larger than the window size and the TFT monitor used had a maximum resolution of 1280x1024, so already a texture resolution of 1024x1024 could only be used in full screen mode. Naturally, larger textures take longer to generate and can cause texture swapping problems, so always selecting a large texture may not be appropriate.

As one can see in Figure 3, any resolution $<=$ 128x128 results in such a blocky shadow that it is better to leave the scene shadowless than use a shadow that can be identified as artificial on the first view. We recommend the use of the maximum resolution available, or at least 512x512 as a resolution for the shadow texture. The results are still blocky, but Performer tries to soften the edges, so it is tolerable.

The other half of the resolution problem is dependent on the chosen field-of-view and, in practice highly, influences the "blockyness" of the shadows. The smaller the field-of-view is, the better the resolution is utilized. In VR, where it is common to simulate a world size of square kilometres, it is not recommend to use a shadow field-of-view including the whole scene like it would be used for a sun to casts shadows. Instead, local light sources lighting need to be used, shadowing only a small area with each.

Additionally, it is possible that the depth value of a fragment in the depth texture differs from the one stored in the frame buffer for the same object, since the depth values of the frame buffer and the ones of the shadow texture descend from different rasters. This produces artefacts and incorrect self-shadowing when they are compared and are detected as different. The steeper the slope of the surface in depth and the lower

(a) polygon offset is: too high (b) polygon offset is: too low (c) polygon offset is: just right (d) both techniques in combination

Figure 4: Generating and tweaking shadows using the implementation.

the resolution it is, the more such errors appear. The special values PFLS_SHADOW_DISPLACE_SCALE and PFLS_SHADOW_DISPLACE_OFFSET reduce such mistakes by adding a small offset, which is dependent on the surface's depth slope, to the sampled depth value. Usual values are 1.0 for the displace scale and 0.001 for the displace offset, but they are scene dependent. Customizing them for the current scene must be done manually. The effect of an incorrect offset, either too great or too small, can be seen in Figure 4. An important thing to mention is that it often occurs that when a good value for an specific object in the scene is found, it will result in improper results for other objects. It may be the case that there is no value, with which all objects are properly shadowed!

Stencil Shadow Volumes The Stencil Shadow Volumes implementation integrated well in AVANGO and seemed to be a solution that would be highly effective for our use. The implementation process was made more complex due to Performer's handling of state. Objects in Performer that do not possess their own GeoState inherit the current GeoState. This is done to reduce OpenGL context switches, but leads to problems if extra care is not taken in the shadow programming. That is why the GeoSet of the shadow quad also gets an empty GeoState to avoid inheriting the current one. With this implementation issue taken care of, we had expected the user to get their "plug 'n play" shadows, if only for limited objects due to the processing load of the method.

While implementing the Stencil Shadow Volumes, we discovered fully unexpected problems with regard to the shadow casting objects. Although it was known that such objects must be closed, 2-manifold, and consisting only of triangles (or triangle-strips), the problems these limitations brought up were not foreseen.

Most of the objects available on the internet are not constructed carefully enough to be usable for shadow volume generation, because the modelers didn't create them for this purpose. Similarly, the models created by students and computer scientists are often inadequate. If the model is only displayed, it seems to be all right, but if further processing of the model is performed, it causes problems. Shared vertices of neighboring surfaces that have differing positions and are bordered by more than two surfaces could result in holes. This leads to incorrect results, not only for shadow volume calculation, but also for techniques like Non-Photorealistic Rendering. A single incorrect vertex is enough to get a completely wrong shadow volume, which results in visible parts of the shadow volume.

This problem is unfortunately compounded by the SG systems and, in particular, Performer. When loading objects, the Performer loader tries to optimize the model for faster processing, converting it into its own native format. This leads to a re-triangulation of the object, which, in turn, isn't implemented carefully enough to create shadow volumes from this. Many times, after this "optimization," incorrect planes are included in the object or other planes are missing, so there are holes in the object. Mostly, these faults are not visible on the object itself, but the models can no longer be used for shadow generation. To avoid such problems, it is possible to convert the object to the Performer format in advance by using the program "pfConv." This application can be configured such that the triangulation remains untouched. If the converted object is loaded by the shadow-application, the loader will never re-triangulate it again.

Combined Usage Both the Stencil Shadow Volume and Shadow Mapping techniques and implementations have strengths and weaknesses. The quality of pictures strongly depends on the environment they are used on.

Most often, Shadow Mapping's implementation is faster than Stencil Shadow Volumes, since most of the calculations can be done on the specialized GPU of modern graphic cards. The dependency on textures and their limited resolution are two limitations on Shadow Mappings ability to produce beautiful shadows. On the other hand, it is not dependent on the scene's complexity as shadow volumes are, so it can be used as a more general solution, particularly for complex scenes. The Shadow Mapping implementation is also much more robust in terms of handling arbitrary objects, but requires fine adjustments with polygon offset over all components. Adjusting the polygon offset of the scene can be performed for static scene; However, for moving objects, changing position and form, the use Sten-

cil Shadow Volumes do not require adjusting it to the current situation and scene, making potentially better suited for this usage.

Due to the completely different concepts used, it is possible to use both shadow methods in combination. The only thing that should be observed is the use of different light sources for each shadow method to avoid having the resulting shadows overlap. An example of the combined usage can be seen in Figure 4(d), where the brighter shadow is produced by Shadow Mapping. The darker coloring of the Stencil Shadow in the figure is purposely done to differentiate the two shadows.

6 FUTURE WORK

Our experiences have shown that the biggest problem with Stencil Shadow Volumes in the SG are objects with wrong surfaces or holes in them. To avoid these problems completely, methods to automatically check and repair the objects would be optimal. Such tools embedded in the modeling program would be of great help for modelers, particularly those amateur modelers VR often uses. Additionally, a model loader that is capable of reducing automatically the complexity of the GeoSets to accelerate the volume calculations would be desirable. In many Stencil Shadow Volume applications there is, next to the original object, a reduced version; The original is displayed in the scene and the reduced version is used for calculating the shadow volume.

Future investigations of interest would be the implementation of advanced Shadow Mapping approaches. There are many proposals that avoid or reduce the dependency on the texture resolution, for example the "Perspective Shadow Mapping." [8] Unfortunately, to use them in our setup would require customizing the available pfLightSource, which is not available in source code. Instead, like the fpStencilShadow extension, a new node or object must be created. This would require a multi-pass rendering algorithm, which needs to access the underlying channel to be implemented. Dependent on the VR system being used, this could be difficult to implement, as is the case with AVANGO.

7 CONCLUSION

In this paper we have presented how two popular shadow techniques, Shadow Mapping and Stencil Shadow Volumes, can be implemented in a Scene Graph based VR system, in pursuit of a system for automatic generation of shadows. We presented how we implemented these in the OpenGL Performer Scene Graph and the AVANGO VR system, using methods viable for other systems. Based on our experience, we have discussed both the implementation and usage of the systems. As a portion of this discussion, we have presented advice to the potential user of such a system on the different aspects that the user must be aware of, such as model issues with Stencil Shadow Volumes and tweaking Shadow Mapping parameters for typical VR environments. Finally, we have identified a number of areas of further research and development, which could help deliver the goal of truly automatic generation of shadows in SG based VR.

REFERENCES

[1] John Carmack. John carmack on shadow volumes. *http://developer.nvidia.com/object/robust-shadow-volumes.html*, 2000.

[2] Franklin Crow. Shadow algorithms for computer graphics. *Proceedings of SIGGRAPH 77*, 11(2):242–248, 1977.

[3] Michael Haller, Stephan Drab, and Werner Hartmann. A real-time shadow approach for an augmented reality application using shadow volumes. In *Proceedings of the ACM Symposium on Virtual Reality Software and Technology*, pages 56–65, 2003.

[4] Tim Heidmann. Real shadows real time. *IRIS Universe*, 18:28–31, 1991.

[5] Wallace Sadowski and Kay Stanney. *Handbook of Virtual Environments: Design, Implementation, and Applications*, chapter Presence in Virtual Environments, pages 791–796. Lawrence Erlbaum Associates Inc,US, 2002.

[6] SGI. OpenGL Performer Programmer's Guide. *http://techpubs.sgi.com/library/tpl/cgi-bin/browse.cgi?coll=0650&db=bks&cmd=toc&pth=/SGI_Developer/Perf_PG/*, 2004.

[7] M. Slater, M. Usoh, and Y. Chrysanthou. The influence of dynamic shadows on presence in immersive virtual environments. In *citeseer.ist.psu.edu/slater95influence.html*, 1995.

[8] Marc Stamminger and George Drettakis. Perspective shadow maps. In John Hughes, editor, *Proceedings of ACM SIGGRAPH 2002*. ACM Press/ACM SIGGRAPH, July 2002.

[9] Andrei State, Gentaro Hirota, David T. Chen, William F. Garrett, and Mark A. Livingston. Superior augmented reality registration by integrating landmark tracking and magnetic tracking. *Computer Graphics*, 30:429–438, 1996.

[10] Henrik Tramberend. *Avocado: A Distributed Virtual Environment Framework*. PhD thesis, Universität Bielefeld, 2003.

[11] Lance Williams. Casting curved shadows on curved surfaces. *Proceedings of SIGGRAPH 78*, 5:270–274, 1978.