Shadow Volume Extrusion using a Vertex Shader

Chris Brennan
3D Application Research Group
ATI Research

Introduction

The shadow volume technique of rendering real time shadows involves drawing geometry that represents the volume of space that bounds what is in the shadow cast by an object. To calculate if a pixel being drawn is in or out of a shadow, shoot a ray from the eye through the shadow volume towards the point on the rendered object and count the number of times the shadow volume is entered and exited. If the ray has entered more times than exited, the pixel being drawn is in shadow. The stencil buffer can be used to emulate this by rendering the back sides of the shadow volume triangles while incrementing the stencil buffer, followed by the front sides of the triangles, which decrement it. If the final result adds up to where it started, then you have entered and exited the shadow an equal number of times, and are therefore outside the shadow, otherwise you are inside the shadow. The next step is rendering a light pass that is masked out by a stencil test.

There are several other very different algorithms to doing shadows [Haines01]. Stencil Shadow Volumes have their benefits and drawbacks compared to other shadowing algorithms like depth buffers. The most important tradeoff is that while shadow volumes have infinite precision and no artifacts, they also have a hard edge and an uncertain render complexity depending on object shape complexity and the viewer and light positions. Previous major drawbacks to shadow volumes were the CPU power required to compute the shadow geometry and the requirement that character animation must be done on the CPU so that a proper shadow geometry could be generated, but a clever vertex shader combined with some preprocessing removes the need for all CPU computations and therefore allows the GPU to do all the character animation. A brief comparison of CPU and GPU complexity and their drawbacks can be found in Game Programming Gems II, p. 482 [Dietrich].

Another historical complexity of shadow volumes that has been solved is what to do if the viewer is inside the shadow. The problem arises that since the viewer starts in shadow, the stencil count begins off by one. Many solutions have been proposed [Haines02], and many are very computationally intensive, but a simple solution exists. Instead of incrementing and decrementing the stencil buffer with the visible portion of the shadow volume, only modify the stencil buffer when the volume is hidden by another surface by setting the depth test to fail. This sounds counter intuitive, but what it does is exactly the same thing, except it counts how many times the ray from the eye to the pixel exits and enters the shadow volume *after* the visible point of the pixel. It still tests to see if the pixel is inside or outside of the shadow volume, but it eliminates the issues with testing to see if the viewer starts in shadow. It does, however, emphasize the need to make sure that all shadow volumes are complete and closed as opposed to previous algorithms, which did not require geometry to cap the front or back of the volume.

Creating Shadow Volumes

The time consuming part of the algorithm is to detect all of the silhouette edges. These are normally found by taking a dot product with the light vector and each of the edge's two neighboring face normals. If one dot product is positive (towards the light) and one is negative (away from the light), then it is a silhouette edge. For each silhouette edge, create planes extending from the edge away from the light creating the minimum geometry needed for the shadow volume. Unfortunately, not only is it expensive to iterate across all of the edges, but also it is expensive to upload the new geometry to the video card every frame.

However, hardware vertex shaders can be used to do this work on chip. The general idea is to create geometry that can be modified by a vertex shader to properly create the shadow volume with any light position so that the geometry can reside on chip. At initialization or preprocess time, for each edge of the original object geometry, add a quad that has two sides consisting of copies of the original edge and two opposite sides of zero length. The pseudo code for this is as follows:

```
For each face
      Calculate face normal
      Create 3 new vertices for this face and face normal
      Insert the face into the draw list
      For each edge of face
            If (edge has been seen before)
                  Insert degenerate quad into draw list
                  Remove edge from checklist
            Else
                  Insert edge into a checklist
If (any edges are left in checklist)
      flag an error because the geometry is not a closed volume.
```

Figure 1 shows the geometry with the quads inserted and spread apart slightly so that they can be seen. These quads will make up the extruded edges of the shadow volume when required. The original geometry is still present and is used to cap the extruded edges on the front and back to complete the volume. After the quad insertion, each vertex neighbors only one of the original faces, and should include its face's normal. When rendering the volume, each vertex's face normal is dotted with the light vector. If the result is negative the face is facing away from the light and should therefore be pushed out to the outer extent of the light along the light vector. Otherwise, it stays exactly where the original geometry lies.

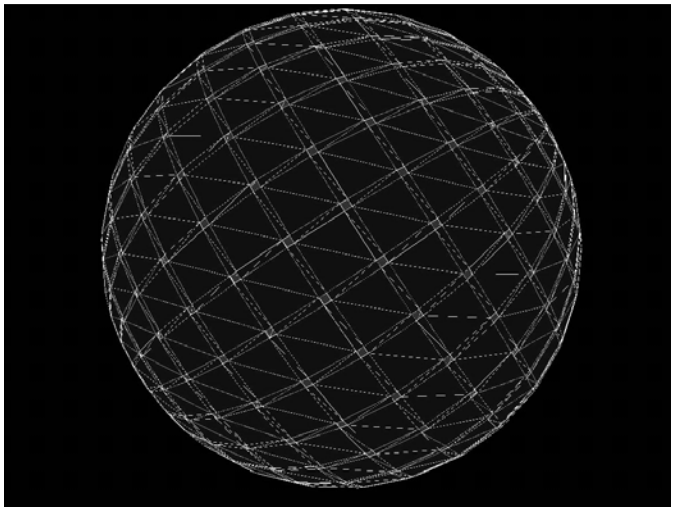

Figure 1: Illustration of invisible quads inserted into the original geometry to create a new static geometry to be used with a shadow volume vertex shader

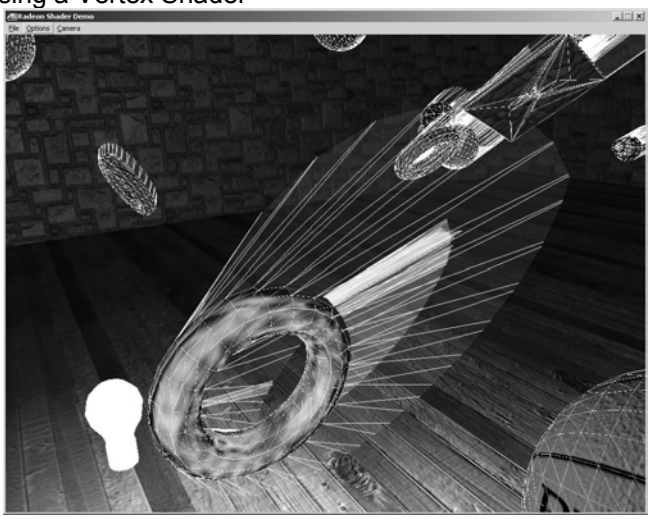

Figure 2: Shadow volumes after being extruded away from the light.

After this pass is completed, the light pass is rendered using a stencil test to knock out the pixels that are in shadow.

Figure 3: The final pass completes the effect.

The shadowed room application with source code can be found on the ATI Developer Relations website.

Effect File Code

```
matrix  mWVP;
matrix  mWV;
matrix  mP;
matrix  mWVt;
vector  cShd;
vector  pVL;

vertexshader vShd =
    decl
    {
        stream 0;
        float v0[3];  // Position
        float v1[3];  // FaceNormal
```

```
    }
    asm
    {
        ; Constants:
        ; 16..19 - Composite World*View*Proj Matrix
        ; 20..23 - Composite World*View Matrix
        ; 24..27 - Projection Matrix
        ; 28..31 - Inv Trans World*View
        ; 90     - {light range, debug visualization amount, z near, z far}
        ; 91     - View Space Light Position

        vs.1.0
        def c0, 0,0,0,1

        ; View Space
        m4x4  r0, v0, c20   ; World*View Transform of point P (pP)
        m3x3  r3, v1, c28   ; World*View Transform of normal (vN)

        sub   r1, r0, c91   ; Ray from light to the point (vLP)

        dp3  r11.x, r1, r1         ; length^2
        rsq  r11.y, r11.x          ; 1/length
        mul  r1, r1, r11.y         ; normalized

        rcp  r11.y, r11.y          ; length
        sub  r11.z, c90.x, r11.y   ; light.Range - len(vLP)
        max  r11.z, r11.z, c0.x    ; extrusion length = clamp0(light.Range - len(vLP))

        dp3  r10.z, r3, r1         ; vLP dot vN
        slt  r10.x, r10.z, c0.x    ; if (vLP.vN < 0) (is pointing away from light)

        mad   r2, r1, r11.z, r0    ; extrude along vLP

        ; Projected Space
        m4x4  r3, r2, c24   ; Projected extruded position
        m4x4  r0, v0, c16   ; World*View*Proj Transform of original position

        ; Chose final result
        sub  r10.y, c0.w, r10.x  ; !(vLP.vN >= 0)
        mul  r1, r3, r10.y
        mad  oPos, r0, r10.x, r1
    };

technique ShadowVolumes
{
    pass P0
    {
        vertexshader = <vShd>;

        VertexShaderConstant[16] = <mWVP>;
        VertexShaderConstant[20] = <mWV>;
        VertexShaderConstant[24] = <mP>;
        VertexShaderConstant[28] = <mWVt>;
        VertexShaderConstant[90] = <cShd>;
        VertexShaderConstant[91] = <pVL>;

        ColorWriteEnable = 0;
        ZFunc            = Less;
        ZWriteEnable     = False;
        StencilEnable    = True;
        StencilFunc      = Always;
        StencilMask      = 0xffffffff;
        StencilWriteMask = 0xffffffff;

        CullMode = CCW;
               StencilZFail      = IncrSat;
    }

    pass P1
    {
        CullMode = CW;
               StencilZFail      = DecrSat;
    }
}
```

Using Shadow Volumes with Character Animation

The technique as described is for statically shaped objects, and does not include characters that are skinned or tweened. The biggest advantage of doing shadow volume extrusion in a vertex shader is that the volumes can exist as static vertex buffers on the GPU and the updated geometry does not have to be uploaded every frame. Therefore this technique needs to be extended to work on animated characters as well. Otherwise if the animation of the shadow volume were to be done on the CPU it would possible to do the optimized shadow volume creation at the same time.

The most straightforward approach is to copy the vertex animation data to the shadow volume geometry and skin and/or tween the face normal as you would the vertex normal. Unfortunately the face normals need to be very accurate and consistent across all vertices of a face, otherwise objects will be split and extruded in inappropriate places resulting in incorrect shadow volumes.

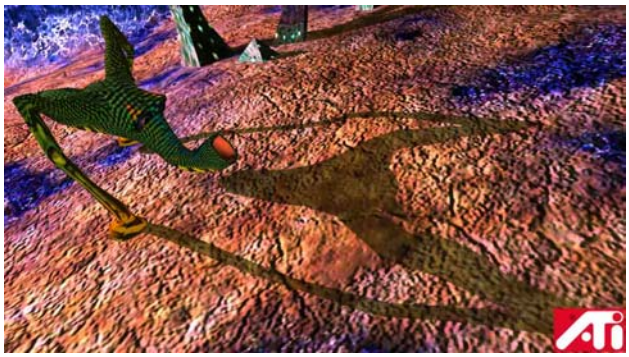

Figure 4: Artifacts caused by skinned face normals. Notice the notch in the shoulder.

These artifacts are the result of the extrusion happening across a face of the original geometry as opposed to the inserted quads along the face edges. This is caused by the fact that each vertex has different weights for skinning which yield a different face normal for each of the three vertices of a face. When that face becomes close to being a silhouette edge, it may have one or two vertices of the face moved away from the light, while the other one stays behind.

The ideal solution is to animate the positions and regenerate the face normals. However, generating face normals requires vertex neighbor information that is not normally available in a vertex shader. One possible solution is to make each vertex contain its two neighbor's position information, animate the vertex position as well at its two neighbors, and recalculate the face normal with a cross product. Three times the regular vertex position data would need to be stored and animated. This can sometimes be very expensive depending on the animation scheme and the size of the models.

An inexpensive way to fix the variation in face normals across a face is to calculate skinning weights per face in addition to per vertex and use the face weights for the face normal. This can be done by averaging all of the vertex weights, or by extracting them directly from the original art. Using the same weight for each vertex of a face guarantees that the shadow volume can only be extruded along the edge quads.

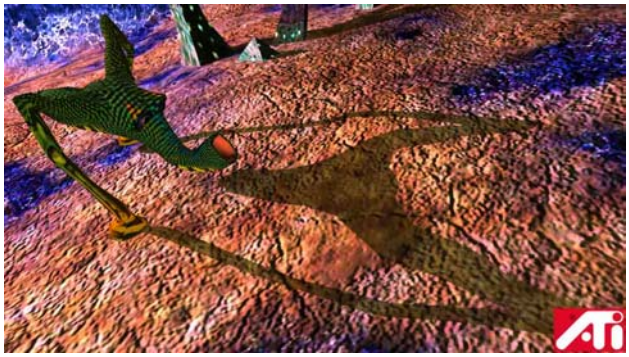

Figure 5: Animated shadow volumes with face normals skinned by face weights.

By using face weights for the face normals, the previously seen artifacts are not visible. This technique can be seen in the included ATI Island Demos.

References

[Dietrich] Dietrich, Sim. "Practical Priority Buffer Shadows," *Game Programming Gems II,* ed Mark DeLoura, Charles River Media, p. 482, 2001.

[Haines01] Haines, Eric and Möller, Tomas. "Real-Time Shadows," GDC 2001 Proceedings.

[Haines02] Haines, Eric and Akenine-Möller, Tomas, *Real-Time Rendering, 2nd edition*, A.K. Peters Ltd, 2002.