ShadowVolume Sample

Collapse All

The sample demonstrates one common technique for rendering real-time shadows called shadow volumes. The shadows in the sample work by extruding faces of the occluding geometry (that are facing away from light) to form a volume that represents the shadowed area in 3D space. The stencil buffer is used as a mask for rendering additional geometry, and is updated as geometry is rendered.

Rendering real-time shadows has been an advanced topic in 3D computer graphics. This sample implements the shadow volume technique using the stencil buffer and a vertex shader. First, the occluding geometry is examined, and a different geometry (that represents the shadow volume) is created. When rendering the scene, the shadow volume geometry is rendered with a vertex-extruding vertex shader using the depth-fail stencil technique. The stencil buffer is updated during this process. Using the stencil buffer as a mask, pixels that lie in the shadow volume do not receive lighting. The depth-fail shadow technique requires that the shadow volume must be a closed volume.

The sample supports up to 6 lights and can render the scene three ways:

- With shadows as you would normally view the scene.
- With shadows and the shadow volume.
- With the shadow volume only, but no shadows. This option renders a color code for each pixel based on the complexity of the shadow volume at each pixel (i.e., how many times a pixel is rendered with the shadow volume).

Path

Source	*SDK root*\Samples\C++\Direct3D\ShadowVolume
Executable	*SDK root*\Samples\C++\Direct3D\Bin*x86 or x64*\ShadowVolume.exe

How the Sample Works

A shadow volume of an object is the region in the scene that is covered by the shadow of the object caused by a particular light source. When rendering the scene, all geometry that lies within the shadow volume should not be lit by the particular light source. A closed shadow volume consists of three parts: a front cap, a back cap, and the side. The front and back caps can be created from the shadow-casting geometry: the front cap from light-facing faces and the back cap from faces facing away from light. In addition, the back cap is translated a large distance along the direction of light, to make the shadow volume long enough to cover enough geometry in the scene. The side is usually created by first determining the silhouette edges of the shadow-casting geometry then generating faces that represent the silhouette edges extruded for a large distance along the light direction. Figure 1 shows different parts of a shadow volume.

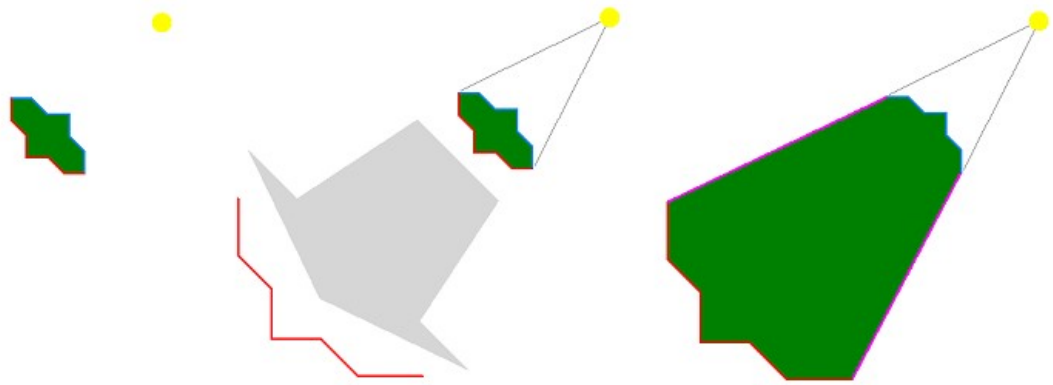

Figure 1: Creation of a shadow volume. The front cap (blue) and back cap (red) are created from the occluder's geometry. The back cap is translated to prolong the shadow volume, and the side faces (purple) are generated to enclose the shadow volume.

This sample demonstrates a specific implementation of shadow volumes. Instead of determining the silhouette and generating the shadow volume geometry on the CPU, the sample first generates a mesh that can represent the shadow volume of the occluding geometry regardless of light direction and uses a vertex shader to perform vertex extrusion as the shadow volume mesh is rendered. The underlying idea is that for triangles that face the light, we can use them as-is for the front cap of the shadow volume. For triangles that face away from the light, their vertices are translated a large distance along the light direction at each vertex, then they can be used as the back cap. However, a problem occurs at silhouette edges where one triangle faces the light and its neighbor faces away from the light. In this situation, each triangle causes its vertices to be processed differently, and it must be determined how the vertices that are shared by the two triangles should be handled.

To solve this, the two triangles are split by duplicating the shared vertices so that each triangle has its own unique three vertices. When the common edge between the triangles becomes a silhouette edge, one triangle stays where it is and the other moves along the light direction. This, however, creates a gap between the two triangles, but a closed shadow volume cannot have any gap or hole. This can be fixed by adding a quad to the shadow volume mesh between the two triangles. The edge that is shared by the two triangles gets split, and then the four vertices define the quad. Before extrusion, the quad is degenerate because the triangles are next to each other. However, when the triangles are far apart, the quad is stretched and automatically forms the side of the shadow volume. Figure 2 illustrates this process.

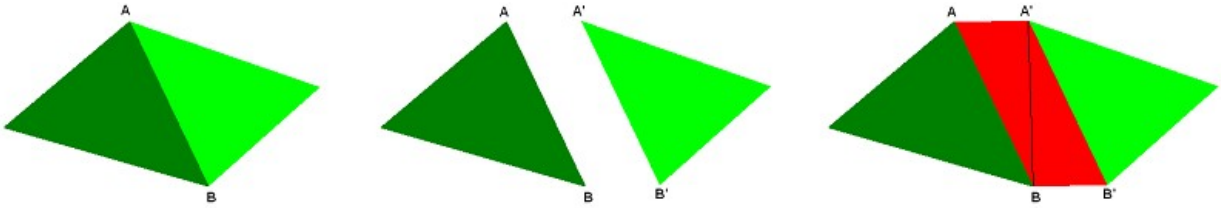

Figure 2: The process of splitting two faces. Left: The two faces share vertices A and B. Center: Generate vertices A' (duplicate of A) and B' (duplicate of B) and then let one of the faces reference them. Right: Generate a quad using vertices A, A', B, and B'. For illustration purpose, A' and B' are drawn at a distance from A and B. In actuality, A and A' are coincident in the mesh's vertex buffer, and so are B and B'. The newly created quad would therefore be degenerate.

Generating the Shadow Volume Mesh

The biggest advantage of generating a static mesh for the shadow volume and having a vertex shader extrude its vertices is that very few CPU cycles are required to render shadows. The shadow volume mesh, once generated, does not need to change no matter where the light position is because the vertex shader can extrude vertices in the correct direction as it receives the vertices from the sample. The function that generates shadow volumes is called GenerateShadowMesh. This function takes an input mesh and outputs a different mesh that represents the shadow volume for the input mesh. There are several things that this function does in order to generate the proper shadow volume for the input mesh.

For every edge in the input mesh, the function must split it up into two edges, effectively separating the two faces that share the edge. Then, it creates a quad (or two triangles) that connects the two split edges. Figure 3 visualizes this process. By default, these quads are degenerate, because the split edges are co-linear. However, when one face is extruded and the other is not, the quad between them gets stretched and forms the side of the shadow volume.

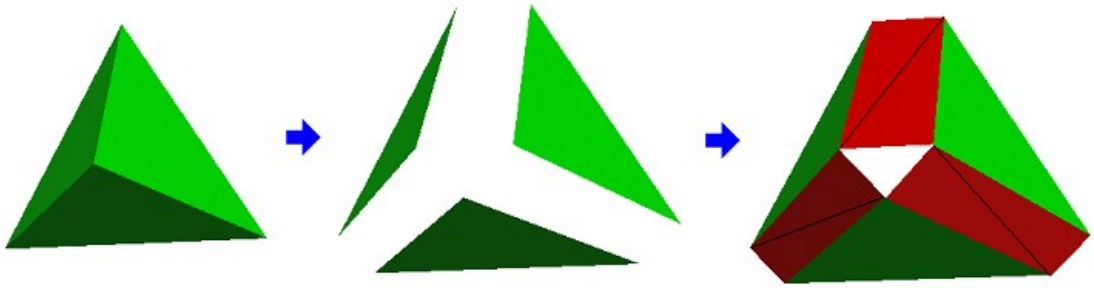

Figure 3: Mesh faces (green) are split, and then quads (red) are inserted.

The algorithm that creates the shadow volume mesh works by iterating through the faces in the input mesh. For each face iterated, three things happen.

- First, three new vertices and one new face are generated for the shadow volume mesh. Each face must have its own unique three vertices because the faces are separated by degenerate quads.
- Then, the normal of the new vertices are computed to be the normal of the new face, as illustrated in figure 4. The reason that this is necessary is because vertex extrusion is done by a vertex shader, and vertex shaders only see vertex normals, not face normals. By setting the vertex normals to match their face normals, the vertex shader will correctly extrude vertices when the faces they belong are facing away from the light.
- Finally, the three edges of a face are added to an edge mapping table. An edge mapping entry contains one source edge, representing the edge in the input mesh, and two output edges, representing the split edges in the output mesh. Essentially, the table records the edges in the source mesh and what edges they split into in the output mesh. This information is needed when the quads are generated later. For each edge of the added face, the algorithm looks through the edge mapping table, and if it cannot find an existing entry for the source edge, it creates one and initializes the source edge and one output edge of the edge mapping entry. However, if it finds that the source edge already has an entry in the table, then it has the four vertices of the quad for this edge, so it adds the two faces for the quad to the output mesh and remove the edge mapping entry from the table.

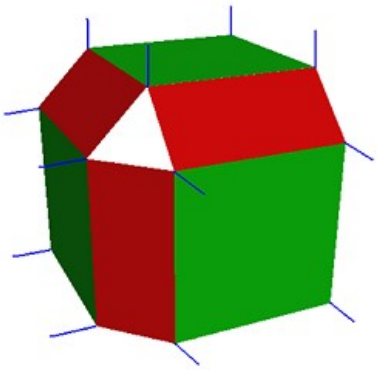

Figure 4: Vertex normals are set up to be identical as the face normals.

At this point, the output mesh contains all of the faces that are in the input mesh, and every edge in the input mesh has been converted to a quad in the output mesh. There is also a list of edges in the mapping table that represents the edges that are not shared in the input mesh. The existence of these edges implies that the input mesh has holes in it, and the holes must be patched so that the shadow mesh becomes a closed volume. This is essential for rendering the shadows with the depth-fail stencil technique. The patching algorithm looks through the mapping table and finds two edges that share a vertex in the original mesh. Then it patches the hole by generating a new face and three new vertices out of the two neighboring edges' vertices. After that, the code generates two quads to connect the patch face to the existing geometry of the output mesh. This process is illustrated in figure 5.

Figure 5: Creating a colsed volume shadow mesh.

Rendering Shadows

At the top level, the rendering steps look like the following:

- If ambient lighting is enabled, render the entire scene with ambient only.
- For each light in the scene, do these:
  - Disable depth-buffer and frame-buffer writing.
  - Prepare the stencil buffer render states for rendering the shadow volume.
  - Render the shadow volume mesh with a vertex extruding shader. This sets up the stencil buffer according to whether or not the pixels are in the shadow volume.
  - Prepare the stencil buffer render states for lighting.
  - Prepare the additive blending mode.
  - Render the scene for lighting with only the light being processed.

The lights in the scene must be processed separately because different light positions require different shadow volumes, and thus different stencil bits get updated. Here is how the code processes each light in the scene in details. First, it renders the shadow volume mesh without writing to the depth buffer and frame buffer. These buffers need to be disabled because the purpose of rendering the shadow volume is merely setting the stencil bits for pixels covered by the shadow, and the shadow volume mesh itself should not be visible in the scene. The shadow mesh is rendered using the depth-fail stencil shadow technique and a vertex extruding shader. In the shader, the vertex's normal is examined. If the normal points toward the light, the vertex is left where it is. However, if the normal points away from the light, the vertex is extruded to infinity. This is done by making the vertex's world coordinates the same as the light-to-vertex vector with a W value of 0. The effect of this operation is that all faces facing away from the light get projected to infinity along the light direction. Since faces are connected by quads, when one face gets projected and its neighbor does not, the quad between them is no longer degenerate. It is stretched to become the side of the shadow volume. Figure 6 shows this.

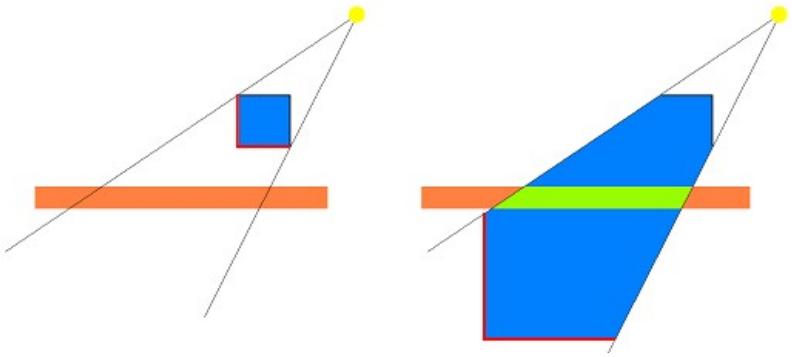

Figure 6: The faces of the shadow volume that face away from light (left, shown in red) have their vertices extruded by the vertex shader to create a volume that encloses area covered by shadows (right).

When rendering the shadow mesh with the depth-fail technique, the code first renders all back-facing triangles of the shadow mesh. If a pixel's depth value fails the depth comparison (usually this means the pixel's depth is greater than the value in the depth buffer), the stencil value is incremented for that pixel. Next, the code renders all front-facing triangles, and if a pixel's depth fails the depth comparison, the stencil value for the pixel is decremented. When the entire shadow volume mesh has been rendered in this fashion, the pixels in the scene that are covered by the shadow volume have a non-zero stencil value while all other pixels have a zero stencil. Lighting for the light being processed can then be done by rendering the entire scene and writing out pixels only if their stencil values are zero.

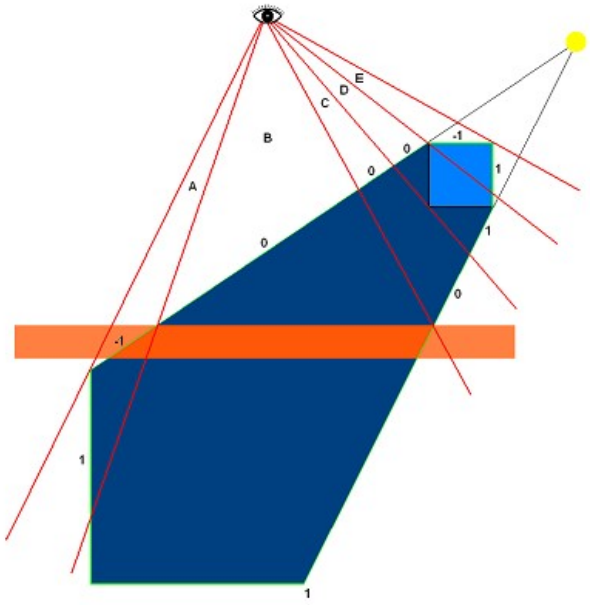

Figure 7: The depth-fail technique.

Figure 7 illustrates the depth-fail technique. The orange block represents the shadow receiver geometry. Regions A, B, C, D and E are five areas in the frame buffer where the shadow volume is rendered. The numbers indicate the stencil value changes as the front and back faces of the shadow volume are rendered. In region A and E, both the front and back faces of the shadow volume fail the depth test, and therefore both cause the stencil value to change. For region A, the orange shadow receiver is causing the depth test to fail, and for region E, the cube's geometry is failing the test. The net result is that stencil values in this region stay at 0 after all faces are rendered. In region B and D, the front faces pass the depth test while the back faces fail, so the stencil values are not changed with the front faces, and the net stencil changes are 1. In region C, both the front and back faces pass the depth test, and so neither causes the stencil values to change, and the stencil values stay at 0 in this region. When the shadow volume is completely rendered, only the stencil values in regions B and D are non-zero, which correctly indicates that regions B and D are the only shadowed areas for this particular light.

Performance Considerations

Performing vertex extrusion in a vertex shader provides some advantages. Once the shadow volume mesh is generated for a mesh geometry, the application can send the mesh to the 3D device for rendering without modifying it in any way, thus removing the need to perform silhouette determination on the host CPU. However, this technique is not without its shortcoming. The static shadow volume mesh tends to have more vertices than the visible geometry. This is because the geometry needs to have its faces separated and quads generated to produce a valid, closed shadow volume. A completely welded mesh would have a shadow mesh with three times the number of vertices. The increase in vertex count could be so significant that it actually reduces performance by processing them in the vertex shader. Applications are encouraged to implement alternative techniques, such as silhouette determination on the CPU, and then compare the performance result to reach a reasonable conclusion about which technique should be used for each shadow-casting object in the scene.

It is beneficial to use separate meshes for the shadow-casting geometry and its shadow volume. If the additional vertices and faces reside in the original mesh, the application would unnecessarily need to process the extra vertices when rendering the geometry, even when these vertices do not serve any purpose during this process. These extra vertices would also come with the vertex data that do not get used when rendering the shadow, such as texture coordinates and material colors. For the situation when less accurate shadows are adequate, a less-detailed version of the mesh can be used for shadow volume rendering.

Using separate meshes may increase the memory requirement for the geometry. This is the case in the sample, because **ID3DXMesh** is used for the mesh objects. However, applications can eliminate the need for additional memory by directly utilizing vertex buffers, index buffers, and vertex streams. First, the scene mesh's vertex buffer needs to be split into multiple vertex buffers so that one vertex buffer contains the positions and normals, while other vertex buffers contain the rest of the vertex data, such as material attributes, texture coordinates, and blending parameters. When rendering the geometry, the application must set up multiple vertex streams, one for each vertex buffer. Next, after the shadow volume mesh is generated, the application must reorder its vertex buffer and fix-up its index buffer, so that all vertices that are originally in the input mesh come before the newly created ones. When this is done, the input mesh's vertex buffer for positions and normals would be identical to the first half of the shadow mesh's vertex buffer. Therefore, the input mesh's vertex buffer can be released and the mesh can simply reference the shadow mesh's vertex buffer. Note that the two meshes still require different index buffers. See figure 8 for a visualization of this buffer-sharing arrangement.

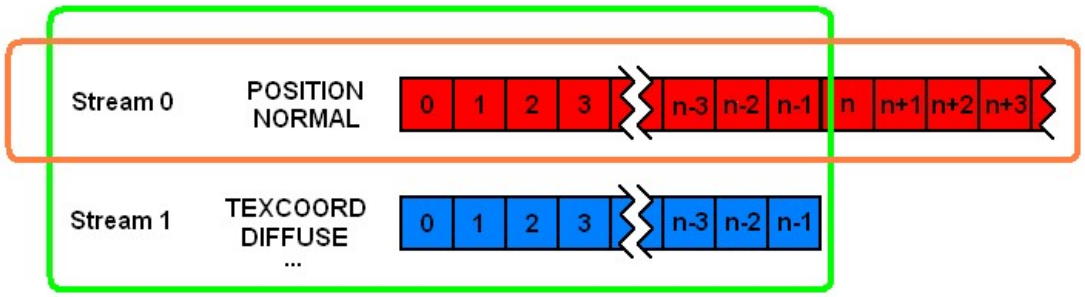

Figure 8: Sharing vertex data between the scene mesh and shadow volume mesh. The scene mesh here (enclosed by the green rectangle) has N vertices that span across two distinct vertex buffers, one for positions and normals (shown in red) and the other for everything else (shown in blue). The shadow volume mesh (enclosed by the orange rectangle) has a vertex count higher than the scene mesh. Therefore, its vertex buffer (in red) contains more than N vertices, of which the first N vertices are shared by the scene mesh.

The drawback of using this technique is that the application must manage the vertex and index buffers of the meshes on its own, and can no longer make use of the **ID3DXMesh** interface for these. Each application should weigh this trade-off based on its own needs and goals.

Finally, there is another area that could call for some performance optimization. As shown earlier, the rendering algorithm with shadow volumes requires that the scene be rendered in multiple passes (one plus the number of lights in the scene, to be precise). Every time the scene is rendered, the same vertices get sent to the device and processed by the vertex shaders. This can be avoided if the application employs deferred lighting with multiple rendertargets. With this technique, the application renders the scene once and outputs a color map, a normal map, and a position map. Then, in subsequent passes, it can retrieve the values in these maps in the pixel shader and apply lighting based on the color, normal and position data it reads. The benefit of doing this is tremendous. Each vertex in the scene only has to be processed once (during the first pass), and each pixel is processed exactly once in subsequent passes, thus ensuring that no overdraw happens in these passes.

Shadow Volume Artifact

It is worth noting that a shadow volume is not a perfect shadow technique. Aside from the high fill-rate requirement and silhouette determination, the image rendered by the technique can sometimes contain artifacts near the silhouette edges, as shown in figure 9. The prime source of this artifact lies in the fact that when a geometry is rendered to cast shadow onto itself, its faces are usually entirely in shadow or entirely lit, depending on whether the face's normal points toward the light. Lighting computation, however, uses vertex normals instead of face normals. Therefore, for a face that is near-parallel to the light direction, it will either be all lit or all shadowed, when in truth, only part of it might really be in shadow. This is an inherent issue of stencil shadow volume technique, and should be a consideration when implementing shadow support. The artifact can be reduced by increasing mesh details, at the cost of higher rendering time for the mesh. The closer to the face normal that the vertex normals get, the less apparent the artifact will be. If the application cannot reduce the artifact down to an acceptable level, it should also consider using other types of shadow technique, such as shadow mapping or pre-computed radiance transfer.

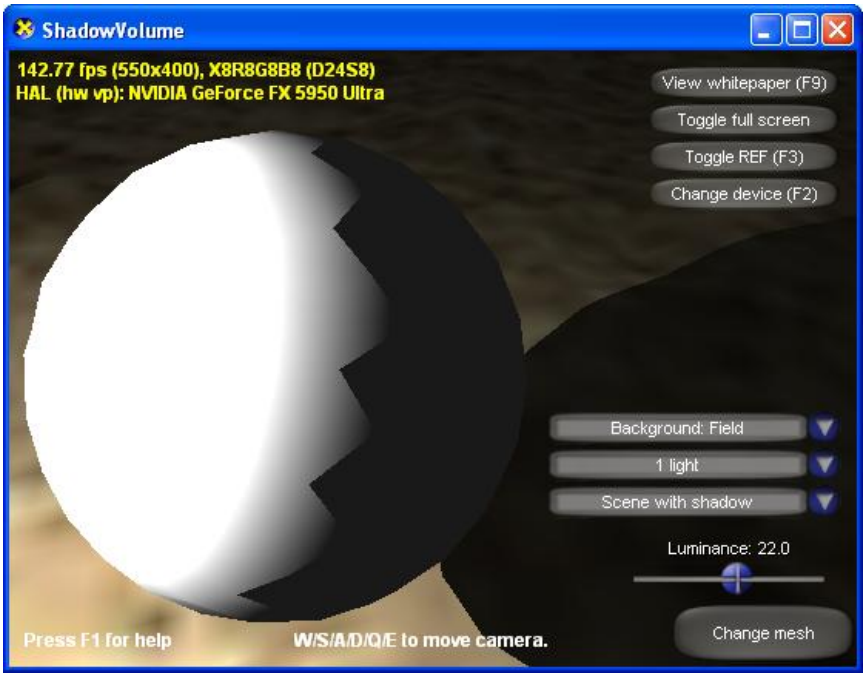

Figure 9: Shadow volume artifact near the silhouette edges.

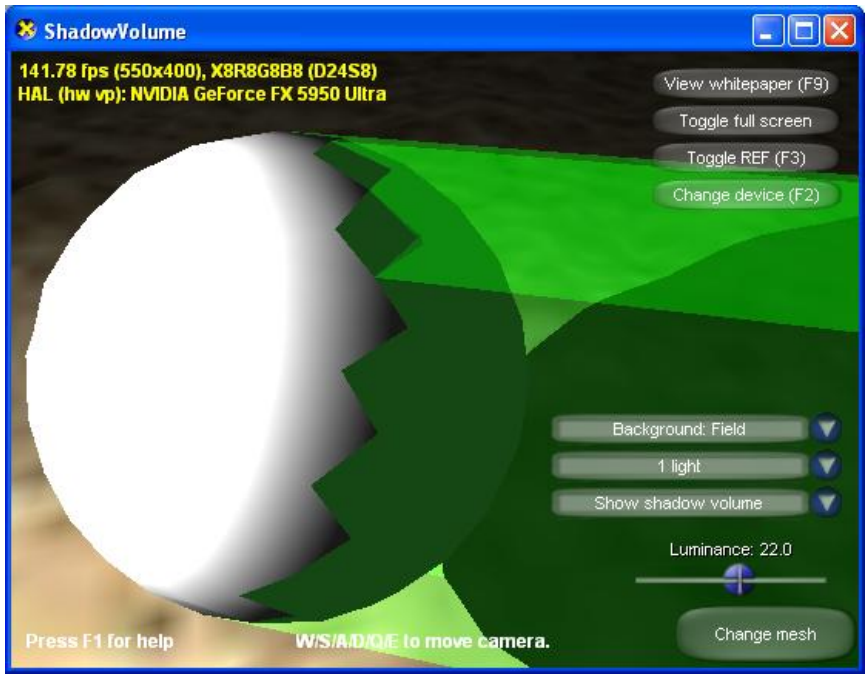

Figure 10: Displaying the shadow volume reveals that the jagged edge is caused by faces that are treated as being entirely in shadow when they are actually only partially shadowed.