ShadowVolume10 Sample

Collapse All

Path

Source	*SDK root*\Samples\C++\Direct3D10\ShadowVolume10
Executable	*SDK root*\Samples\C++\Direct3D10\Bin*x86 or x64*\ShadowVolume10.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 formed by translating the front facing faces a large distance away from the 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 away from the light direction. Figure 1 shows different parts of a shadow volume.

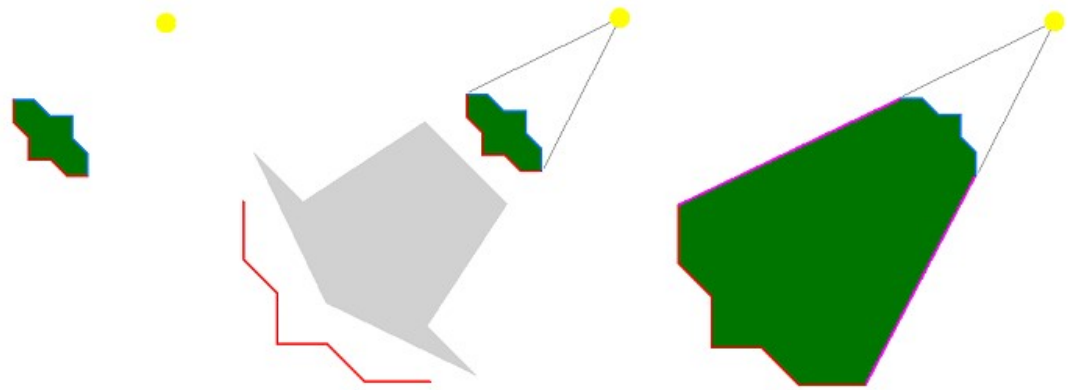
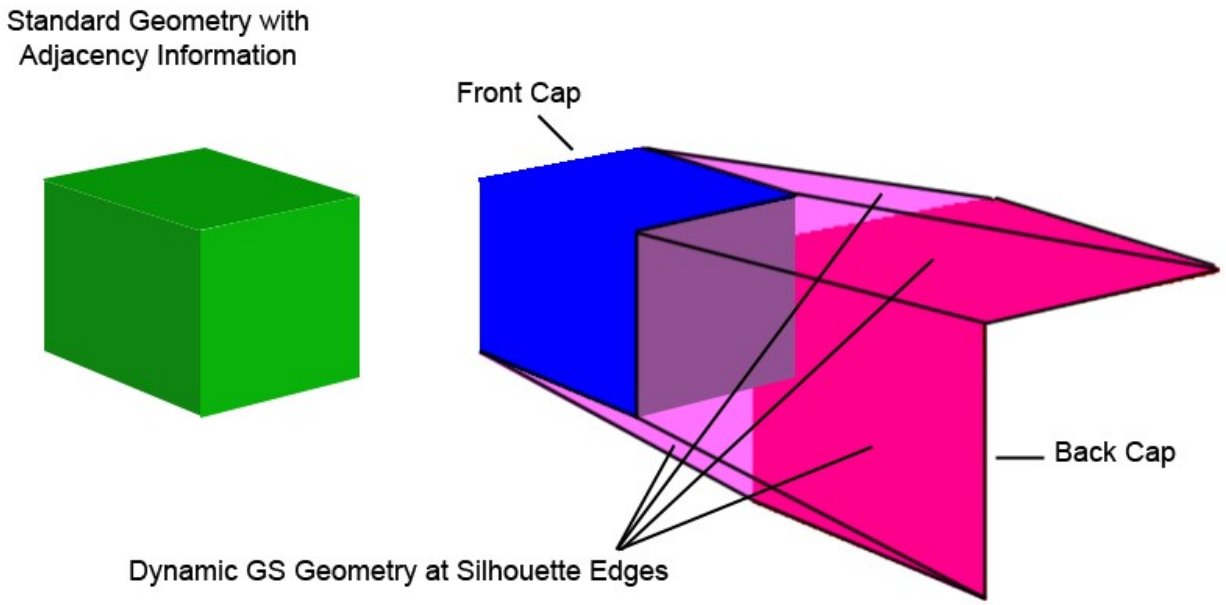


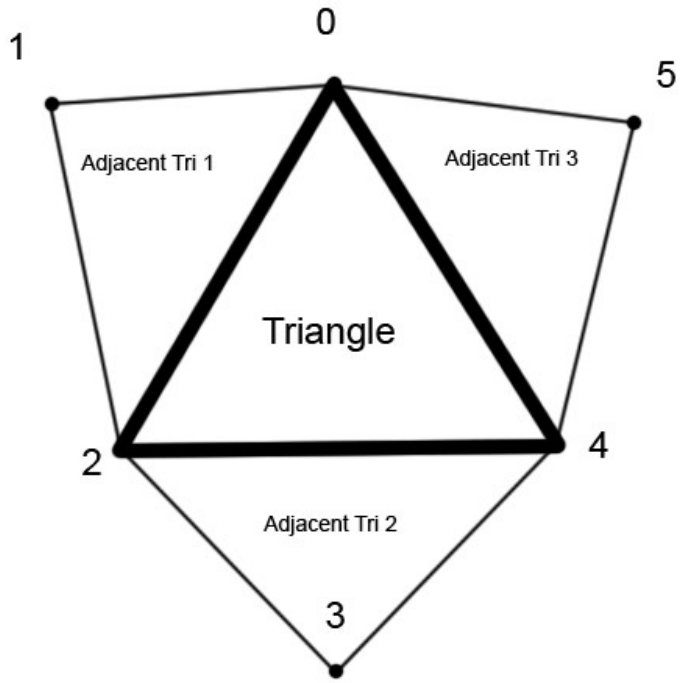
Figure 1:
Creation of a shadow volume.

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. Many traditional shadow volume approaches determine the silhouette and generate the shadow volume geometry on the CPU. This sample determines the silhouette in the geometry shader and uses the fact that the geometry shader can send a variable amount of data to the rasterizer to create new shadow volume geometry on the fly. 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. The back cap is generated from the front facing triangles 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, the geometry shader extrudes two new triangles to create a quad to match up between the front cap of the shadow volume and the back cap.



In order for the geometry shader to find silhouette edges it must know which faces are adjacent to each other. Fortunately, the geometry shader has support for a new type of input primitive, triangleadj. Triangleadj assumes that every other vertex is an adjacent vertex. The index buffer of the geometry must be modified to reflect the adjacency information of the mesh. The CDXUTMesh10::ConvertToAdjacencyIndices handles this by creating an index buffer in which every other value is the index of the adjacent vertex of the triangle that shares an edge with the current triangle. The index buffer will double in size due to the extra information being stored. The figure below demonstrates the ordering of the adjacency information.



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 the following:

- 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.

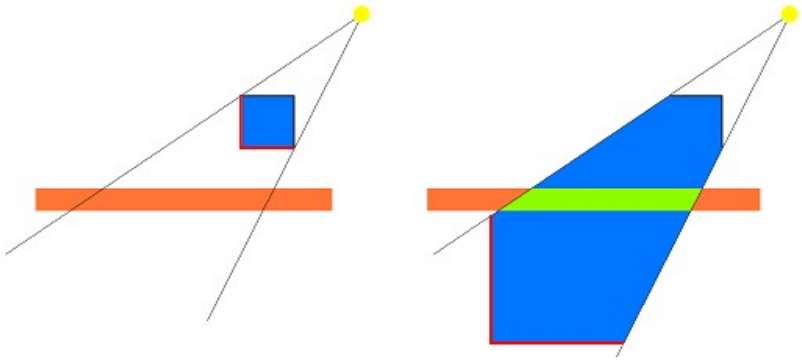


Figure 4: The faces of the shadow volume that face away from light (left, shown in red) have their vertices extruded by the vertex shader to create a volume that encloses area covered by shadows (right).

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.

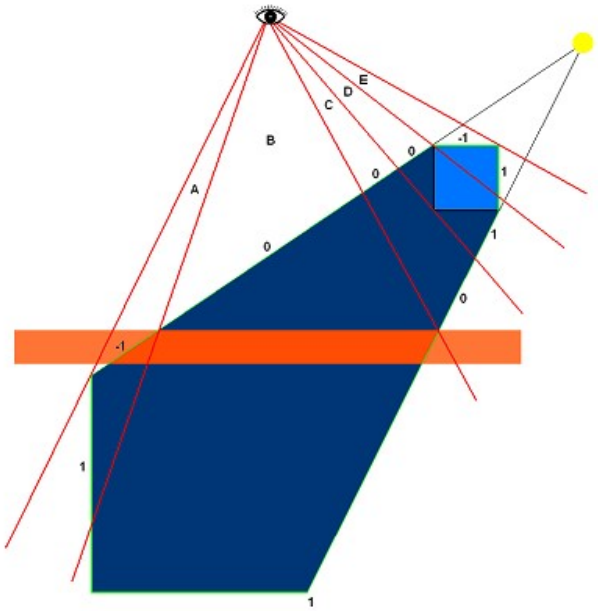


Figure 5: The depth-fail technique.

Figure 5 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

The current sample performs normal rendering and shadow rendering using the same mesh. Because the mesh class tracks only one index buffer at a time, adjacency information is sent to the shader even when it is not needed for shadow calculations. The shader must do extra work to remove this adjacency information in the geometry shader. To improve performance the application could keep two index buffers for each mesh. One would be the standard index buffer and would be used for non-shadow rendering. The second would contain adjacency information and only be used when extruding shadow volumes.

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 Artifacts

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 flaw 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.

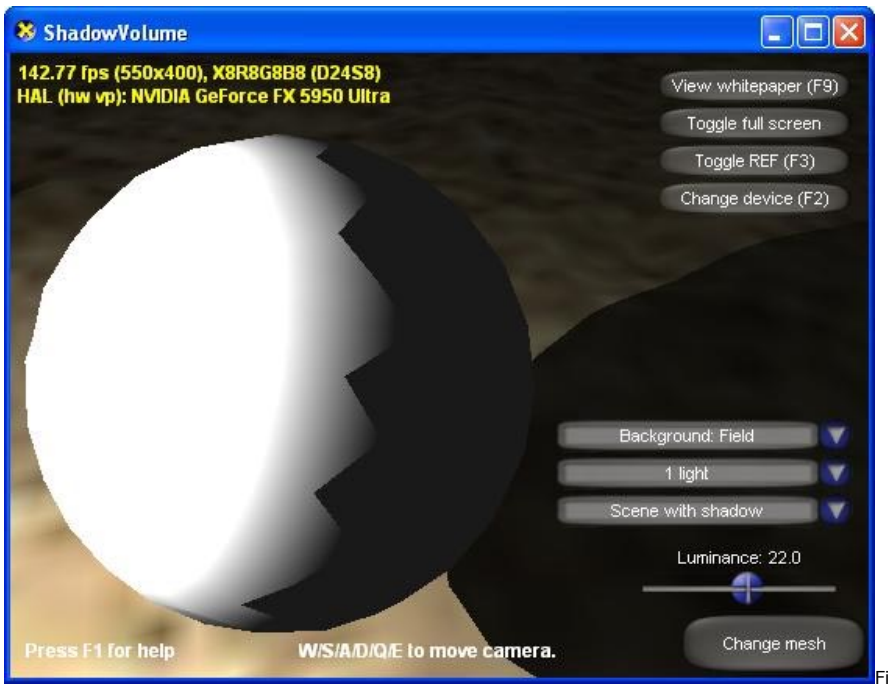


Figure 6: Shadow volume artifact near the silhouette edges.

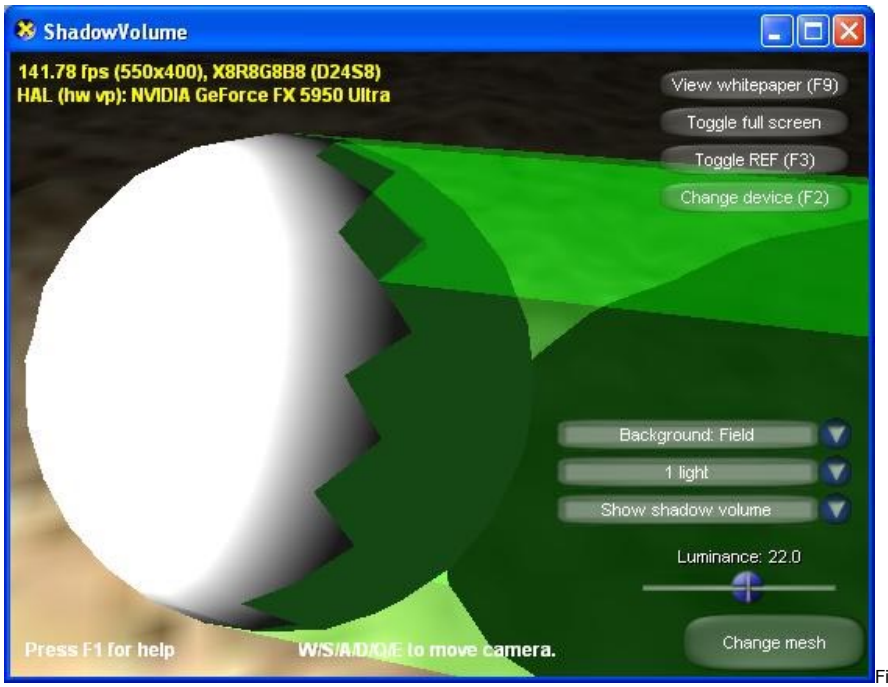


Figure 7: Displaying the shadow volume reveals that the jagged edge is caused by faces that are treated as being entirely in shadow when they are actually only partially shadowed.