Annex: B
ombres

En aquest annex es recullen articles i documents extrets de diferents pàgines webs on programadors i enginyers professionals exposen mètodes per a la creació d'ombres eficientment. Aquests documents són els que es van emprar per a crear l'algorisme que utilitza el joc.

- **The Theory of Stencil Shadow Volumes** de Hun Yen Kwoon.

Explica la teoria de les ombres creades a partir d'un volum creat per un objecte. Compara els algorismes *ZPass* i *ZFail*.

Extret de www.gamedev.net

- **Hardware Shadow Mapping** per Cass Everitt, Ashu Rege i Cem Cebenoyan.

Parla sobre les ombres creades a partir d'un mapa d'ombres i la seva implementació en *OpenGL* i *Direct3D*.

Extret de www.nvidia.com

- **Lightmapping - Theory and implementation** de Keshav Channa.

Introducció i teoria molt completa dels lightmaps així com un mètode d'implementació i els possibles errors durant la seva implementació i ús.

Extret de www.flipcode.com

The Theory of Stencil Shadow Volumes

by Hun Yen Kwoon

Introduction

Shadows used to be just a patch of darkened texture, usually round in shape, which is projected onto the floor below characters or objects in a game. One must be ill informed or naïve to think that we can still get away with this kind of sloppy "hacks" in future 3D games. There used to be a time where shadows are just too expensive to be rendered properly in real-time, but with the ever-increasing power of graphics hardware, failure to provide proper shadows no longer meant mediocre implementations, it borders on being guilty of criminally under-utilizing the graphics hardware available.

There are many differing shadowing techniques and approaches to implementing shadows and nailing down a "best" solution is difficult. In order to understand all the approaches and appreciate their differences, strengths and weakness, I strongly suggest reading anything and everything about doing shadows in 3D. We should not constrain ourselves to just studying shadow volume technique; any shadowing technique is worth a look. Chapter 6 of [13] has a wonderful high-level discussion on most of the known shadowing techniques. To limit the scope of this paper, we shall only discuss the theory and implementation issues of stencil shadow volumes with particular reference to using Microsoft's Direct3D API. It would also be good to understand that stencil shadow volume just isn't the "end all" shadowing technique. A discussion on the strengths of differing shadowing technique can be found at [4] with reference to a game setting. Just recently, Eric Lengyel [11] also presented a very complete article on implementing shadow volumes in OpenGL at the Gamasutra website [17]. The mathematical derivations of Lengyel's article can be found in [12]. Going back a few years, there was the famous "Carmack On Shadow Volumes" text file [6], which is nothing more that an email from John Carmack of id Software to Mark Kilgard of Nvidia about the derivation of the depth-fail shadow volume implementation. It's interesting to note that Carmack independently discovered the depth-fail method while Bill Bilodeau and Mike Songy [7] had also presented similar approach to shadow volumes. Consequently, the depth-fail method is now commonly known as "Carmack's Reverse".

Stencil Shadow Volume Concept

Frank Crow [8] first presented the idea of using shadow volumes for shadow casting in 1977. Tim Heidmann [5] of Silicon Graphics implemented Crow's shadow volume by cunningly utilizing the stencil buffer for shadow volume counting in IRIX GL. Lets take a look at how the original stencil shadow volume technique works.

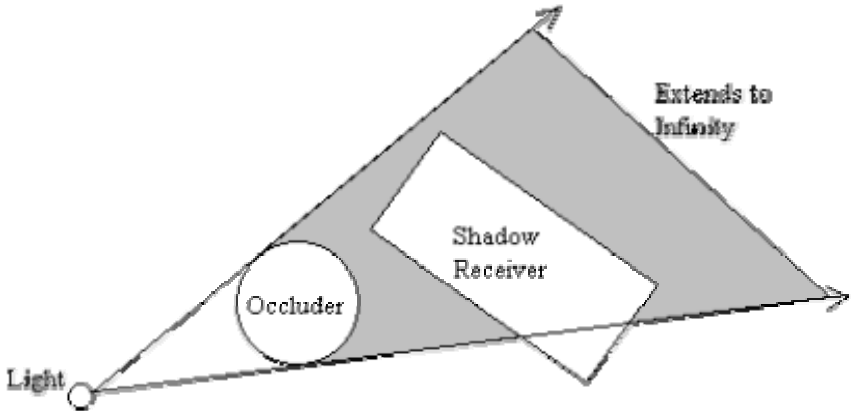


Figure 1: Occluder and shadow volume

As common convention goes, any objects in a scene that cast shadows are called **occluders**. As shown in Figure 1 above, we have a simplistic 2D view (top down) of a scene with a sphere as the occluder. The rectangle to the right of the sphere is the shadow receiver. For simplicity, we do not take into account the shadow volume created by the rectangle. The shaded region represents the shadow volume, in 2D, created by the occluder. The shadow volume is the result of extruding the silhouette edges from the point-of-view of the light source to a finite or infinite distance.

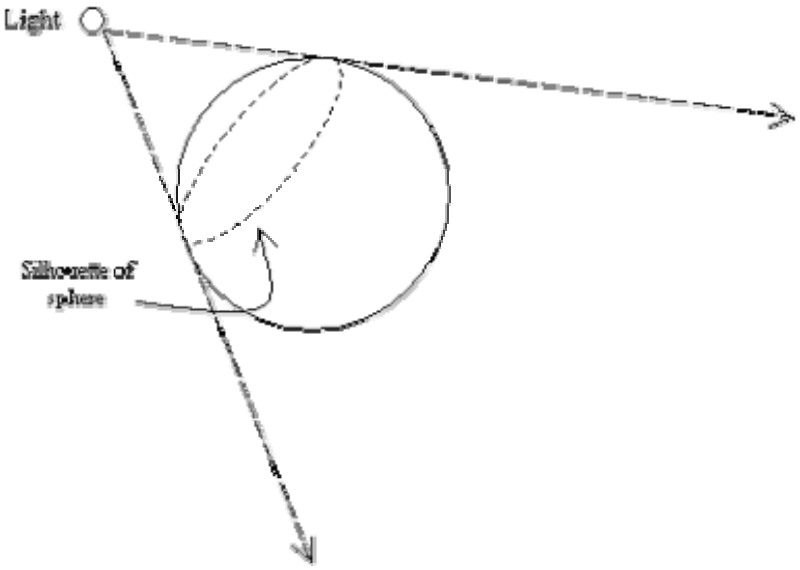


Figure 2: Silhouette of occluders

Figure 2 shows the probable silhouette of the sphere generated from the viewing position of the light source. The silhouette is simply made up of edges that consist of two vertices each. These edges are then extruded in the direction as shown by the broken arrows originating from the light source. By extruding the silhouette edges, we are effectively creating the shadow volume. It should be noted at this point in time that shadow volume extrusion differs for different light sources. For point light sources, the silhouette edges extrude exactly point for point. For infinite directional light sources, the silhouette edges extrude to a single point. We will go into the details of determining silhouette edges and the creation of the shadow volumes later. The magnitude of the extrusion can be

either finite or infinite. Thus, implementations that extrude silhouette edges to infinity are commonly known as Infinite Shadow Volumes.

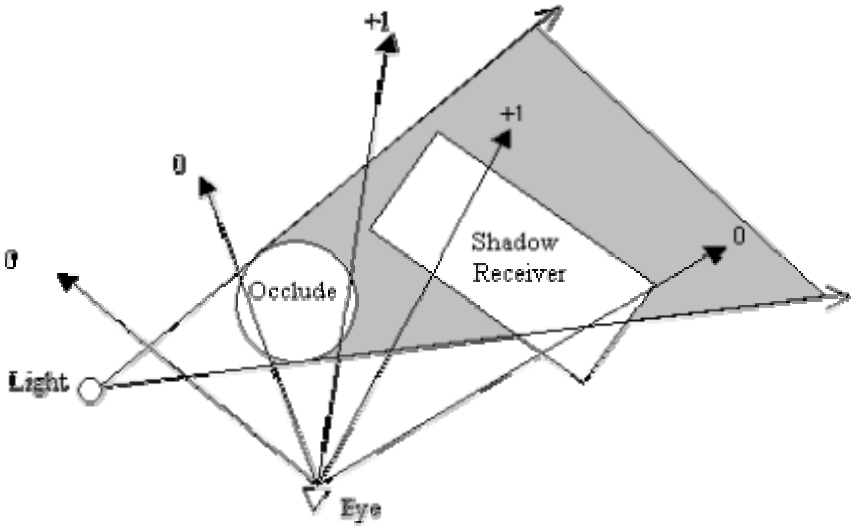


Figure 3: Depth-Pass stencil operation

Figure 3 shows the numerous possible viewing direction of a player in the scene. The numbers at the end of the arrows are the values left in the stencil buffer after rendering the shadow volume. Fragments with non-zero stencil values are considered to be in shadow. The generation of the values in the stencil buffer is the result of the following stencil operations:

1. Render **front** face of shadow volume. If depth test passes, **increment** stencil value, else does nothing. Disable draw to frame and depth buffer.
2. Render **back** face of shadow volume. If depth test passes, **decrement** stencil value, else does nothing. Disable draw to frame and depth buffer.

The above algorithm is also known as the Depth-Pass stencil shadow volume technique since we manipulate the stencil values only when depth test passes. Depth-pass is also commonly known as z-pass.

Let's assume that we had already rendered the objects onto the frame buffer prior to the above stenciling operations. This means that the depth buffer would have been set with the correct values for depth testing or z-testing if you like. The 2 leftmost ray originating from the eye position does not hit any part of the shadow volume (in gray), hence the resultant stencil values is 0, which means that the fragment represented by this two rays are not in shadow. Now lets trace the 3rd ray from the left. When we render the front face of the shadow volume, the depth test would pass and the stencil value would be incremented to 1. When we render the back face of the shadow volume, the depth test would fail since the back face of the shadow volume is behind the occluder. Thus the stencil value for the fragment represented by this ray remains at 1. This means that the fragment is in shadow since its stencil value is non-zero.

Does the shadow volume counting work for multiple shadow volumes? Yes it does.

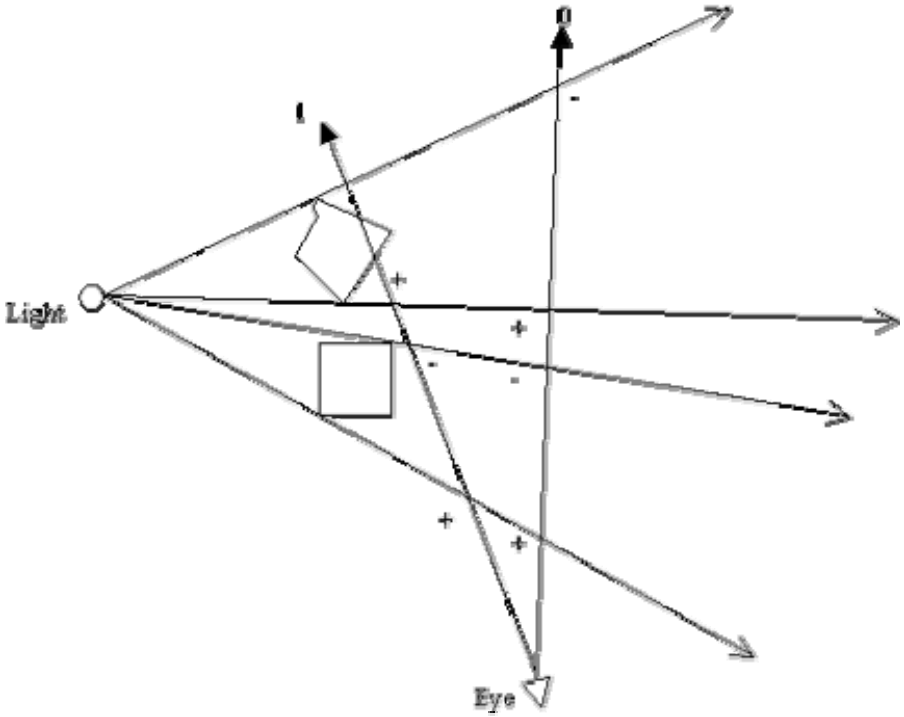


Figure 4: Multiple shadow volumes counting

Figure 4 above shows that the counting using the stencil buffer will still work even for multiple intersecting shadow volumes.

Finite Volume *vs* Infinite Volume

Referring back to Figure 1, you could see that the shadow volume is supposed to extrude to infinity. This is actually not strictly a requirement. We send the shadow volume to infinity in order to avoid the awkward situation whereby the light source is very close to an occluder.

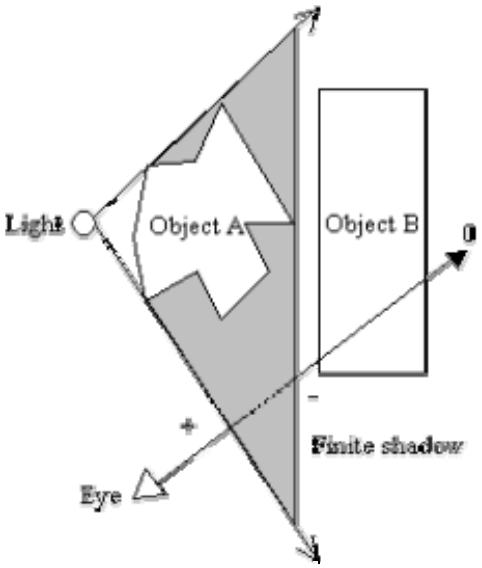


Figure 5: Finite shadow volume fails to shadow other objects

With the light close to object A, a finite shadow volume may not be enough to reach object B. The ray from the eye towards object B will end up with a fragment stencil value of 0 when in fact it should have been non-zero! An infinite shadow volume would ensure that no matter how close the object is to an occluder, the resultant shadow volume would cover all the objects in the scene. We will discuss how to extrude vertices to infinity shortly.

Carmack's Reverse

Why did John Carmack, Bill Bilodeau and Mike Songy even bother to crack their heads to come out with an alternative stencil algorithm since the depth-pass technique seems to work great? Depth-pass really works well, at least most of the time. But when the eye point enters the shadow volume, all hell break loose.

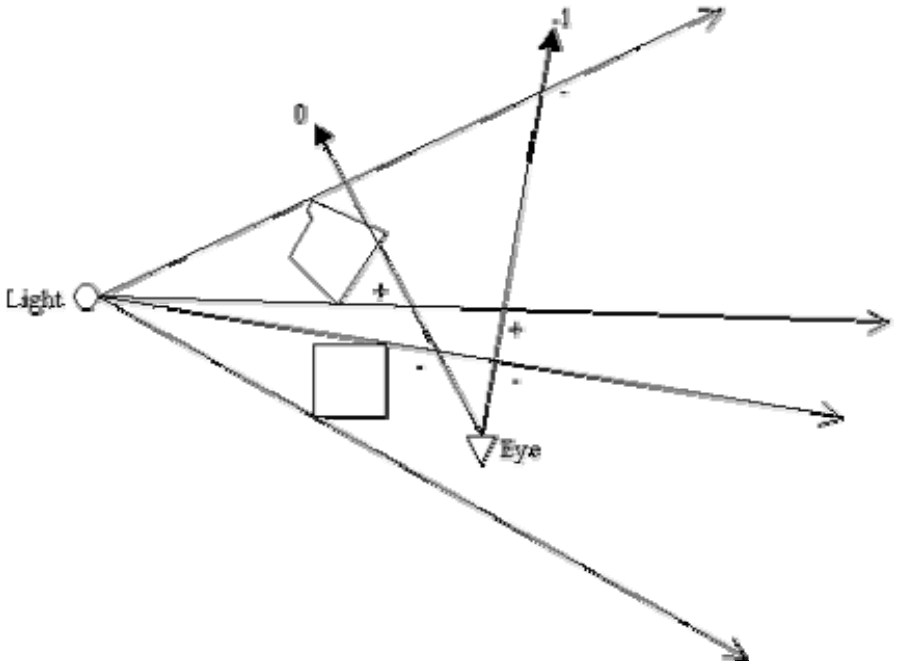


Figure 6: When eye point is within the shadow volume, depth-pass stencil operation fails

As shown in Figure 6 above, the depth-pass technique utterly fails when the eye point is within the shadow volume. This meant that we could not have that big bad horned reaper sneaking up from behind you while engulfing you in the enlarging darkness of his shadows. John Carmack would never have it this way! The following is the depth-fail (a.k.a Carmack's Reverse) algorithm:

1. Render **back** face of shadow volume. If depth test fails, **increment** stencil value, else does nothing. Disable draw to frame and depth buffer.
2. Render **front** face of shadow volume. If depth test fails, **decrement** stencil value, else does nothing. Disable draw to frame and depth buffer.

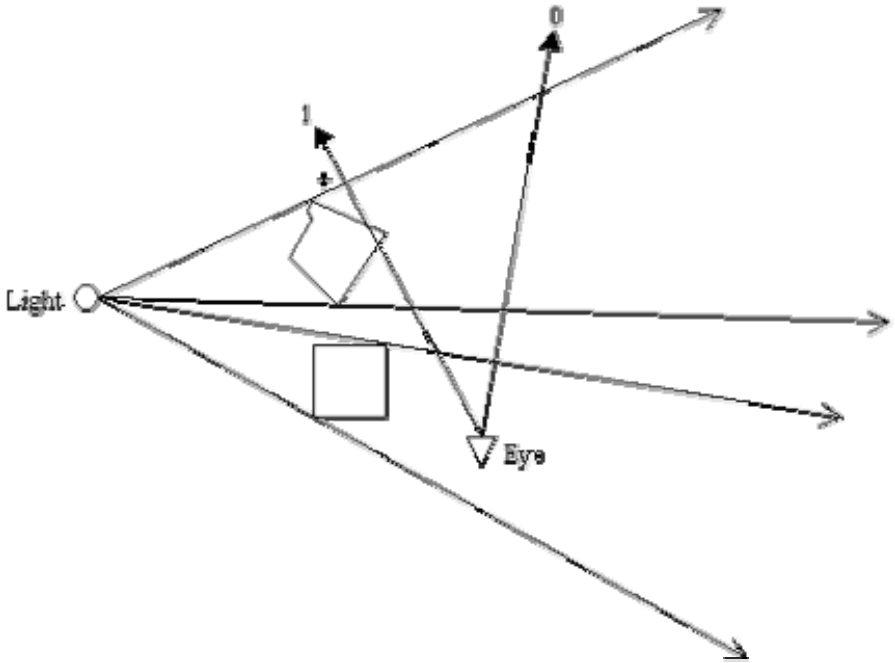


Figure 7: Depth-fail works even if eye point is in shadow

Depth-fail is also commonly referred to as z-fail. Figure 7 shows the depth-fail technique working even when the eye point is in shadow. If you think about the scenario where the eye position is outside the shadow volume, the depth-fail technique should work as well. But really, it fails in some cases. We shall discuss these scenarios soon; just remember for now that both the depth-pass and depth-fail techniques are not perfect. In fact, we would need a combination of different methods to come up with a robust solution for shadow volumes. [11] and [10] contains some very good discussion on robust stencil shadow volume solutions.

Capping For Depth-Fail

To put in non-zero values into the stencil buffer, the depth-fail technique depends on the failure to render the shadow volume's back faces with respect to the eye position. This meant that the shadow volume must be a closed volume; the shadow volume must be capped at both the front and back end (even if back end is at infinity). Without capping, the depth-fail technique would produce erroneous results. Amazing as it may sound, but yes, you can cap the shadow volume even at infinity.

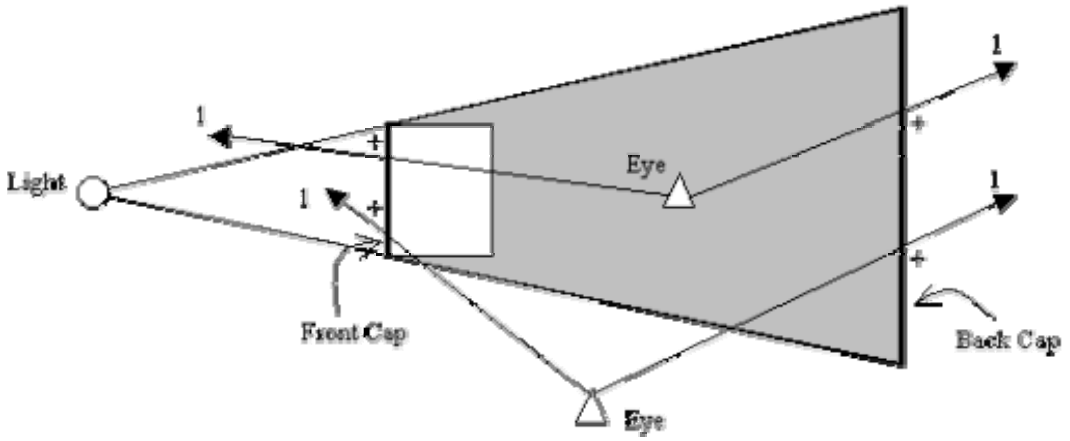


Figure 8: Capping for shadow volume

As shown in Figure 8, the front and back cap (bold lines) creates a closed shadow volume. Both the front and back caps are considered back face from the two eye positions. With depth-fail stenciling operations, the capping will create correct non-zero stencil values. There are a few ways to create the front and back capping. Mark Kilgard [2] described a non-trivial method of creating the front cap. The method basically involves the projection of the occluder's back facing geometries onto the near clip plane and uses these geometries as the front cap. Alternatively, we can build the front cap by reusing the front facing triangles with respect to the light source. The geometries used in the front cap can then be extruded, with their ordering reversed, to create the back cap. Reversing the ordering is to ensure that the back cap face outward from the shadow volume. In fact, we must always ensure that the primitives, in our case triangles, that define the entire shadow volume are outward facing as shown in Figure 9. It must be noted that rendering closed shadow volumes are somewhat more expensive than using depth-pass without shadow volume capping. Besides a larger primitive count for the shadow volume, additional computational resource are also needed to compute the front and back capping. We will go into the details of capping shadow volumes shortly.

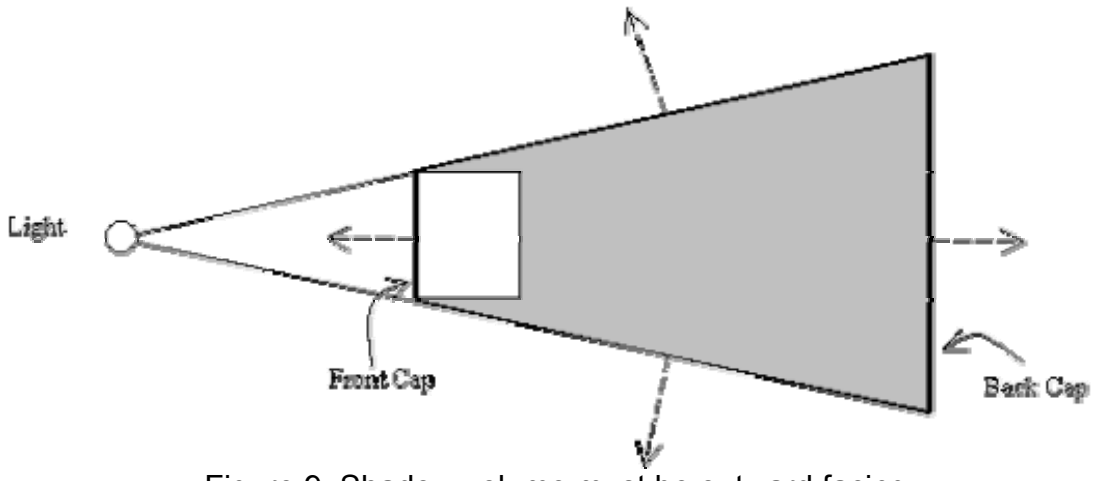


Figure 9: Shadow volume must be outward facing

Putting It Together

Let's collate what we have learned and try to come up with all the require steps to do stencil shadow volumes before we tackle all the deficiencies of the techniques discussed. A general list of steps to implement stencil shadow volumes would be:

1. Render all the objects using only ambient lighting and any other surface-shading attribute. Rendering should not depend on any particular light source. Make sure depth buffer is written.
2. Starting with a light source, clear the stencil buffer and calculate the silhouette of all the occluders with respect to the light source.
3. Extrude the silhouette away from the light source to a finite or infinite distance to form the shadow volumes and generate the capping if depth-fail technique was used. (Infinite shadow volume extrusion is not really compulsory)
4. Render the shadow volumes using the selected technique. Depth-pass or depth-fail.
5. Using the updated stencil buffer, do a lighting pass to shade (make it a tone darker) the fragments that corresponds to non-zero stencil values.
6. Repeat step 2 to 5 for all the lights in the scene.

From the above list of steps, it should be quite obvious that having more lights means having more passes, which can burn a nice hole in your frame rate pocket. In fact, we have to be very selective when deciding which lights should be used for casting shadows. The article [4] has a nice discussion on selecting shadow casting lights within a scene lit by multiple light sources. Imagine your game character standing in the middle of a stadium with four gigantic batteries of floodlights shining down the field. There should be at least 4 shadows of you game character on the floor forming a cross due to the shadow casting from 4 different directions. Selecting only 1 light source here is going to make the scene look weird. Having multiple lights allows you to get nice realistic soft shadows but there are other ways to fake it without resorting to multiple light sources. Soft shadow is a huge topic and is not within the scope of this paper, so let's just drop it from here. Rule of thumb: Always select the dominant light sources in the scene. Using the viewing frustum to select light sources can be very dangerous, since you may have a nice giant 1000-mega watt photon busting spot light right behind the top of your head. It's not in your view frustum, but it's going to be responsible for the most distinct shadows you would see in the scene. Just remember, the fewer the number of lights, the more cycles and rendering passes you can save for other visually more important effects. So choose with care!

From the released screen shots of the upcoming Doom3 engine, I estimate that id Software would have to limit the number of shadow casting lights in any scene to a maximum of say 4 or 5. Well, we will know when Doom3 hits the shelves next year.

Silhouette Determination

The very first step to constructing a shadow volume is to determine the silhouette of the occluder. The stencil shadow algorithm requires that the occluders be closed triangle meshes. This meant that every edge in the model must only be shared by 2 triangles thus disallowing any holes that would expose the interior of the model. We are only interested in the edges shared by a triangle that faces the light source and another triangle that face away from the light source. There are many ways to calculate the silhouette edges and every single one of these methods are CPU cycles hungry. Lets assume we are working with an indexed triangle mesh.

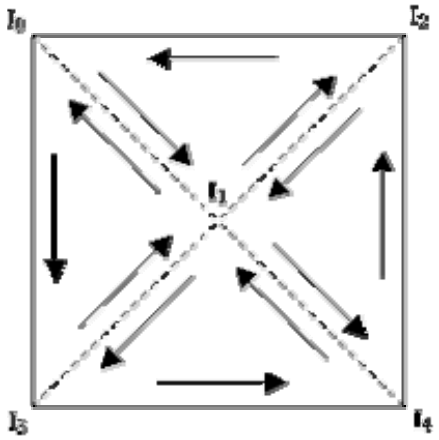


Figure 10: Edge elimination for silhouette determination

Figure 10 shows one side of a box that is made up of four triangles with a consistent counter-clockwise winding. The broken lines indicate the redundant internal edges since we are only interested in the solid line that forms the outline of the box. The redundant internal edges are indexed twice as they are shared by two triangles. We take advantage of this property to come up with a simple method to determine the silhouette edges.

1. Loop through all the model's triangles
2. If triangle faces the light source (dot product > 0)
3. Insert the three edges (pair of vertices), into an edge stack
4. Check for previous occurrence of each edges or it's reverse in the stack
5. If an edge or its reverse is found in the stack, remove both edges
6. Start with new triangle

The above algorithm will ensure that the internal edges would be eventually removed from the stack since they are indexed by more than one triangle.

Eric Lengyel [11] presented another silhouette determination algorithm that makes use of the consistent winding (counterclockwise) of vertices. The method requires 2 passes on all the triangles of the model to filter in all the edges shared by pairs of triangles. The resultant edges list then undergo the dot product operations to get the edges that are shared by a light facing triangle and a non light facing triangle.

It is important to note that silhouette determination is one of the two most expensive operations in stencil shadow volume implementation. The other is the shadow volume rendering passes to update the stencil buffer. These two areas are prime candidates for aggressive optimizations, which we will discuss in detail at the concluding sections of this paper.

Generating Shadow Volume Capping

Remember that shadow volume capping is only necessary for the depth-fail technique. The purpose of doing shadow volume capping is to ensure that our shadow volume is closed, and it must be closed even at infinity. Interestingly, the extrusion of geometries for point light sources and infinite directional light sources are different. Point light sources would extrude the silhouette edges exactly point for point while infinite directional light sources would extrude all silhouette edges to a single point at infinity. This would meant that the shadow volume's back capping would be redundant for infinite directional light sources as it is already closed.

The ideal time to generate the front and back capping would be during the silhouette generation since we are already generating the angles between the light vector and the edges. For the front cap, we just need to duplicate all front facing geometries and use these geometries for extrusion to form the back capping as well. Note that the back cap is only necessary for point light sources.

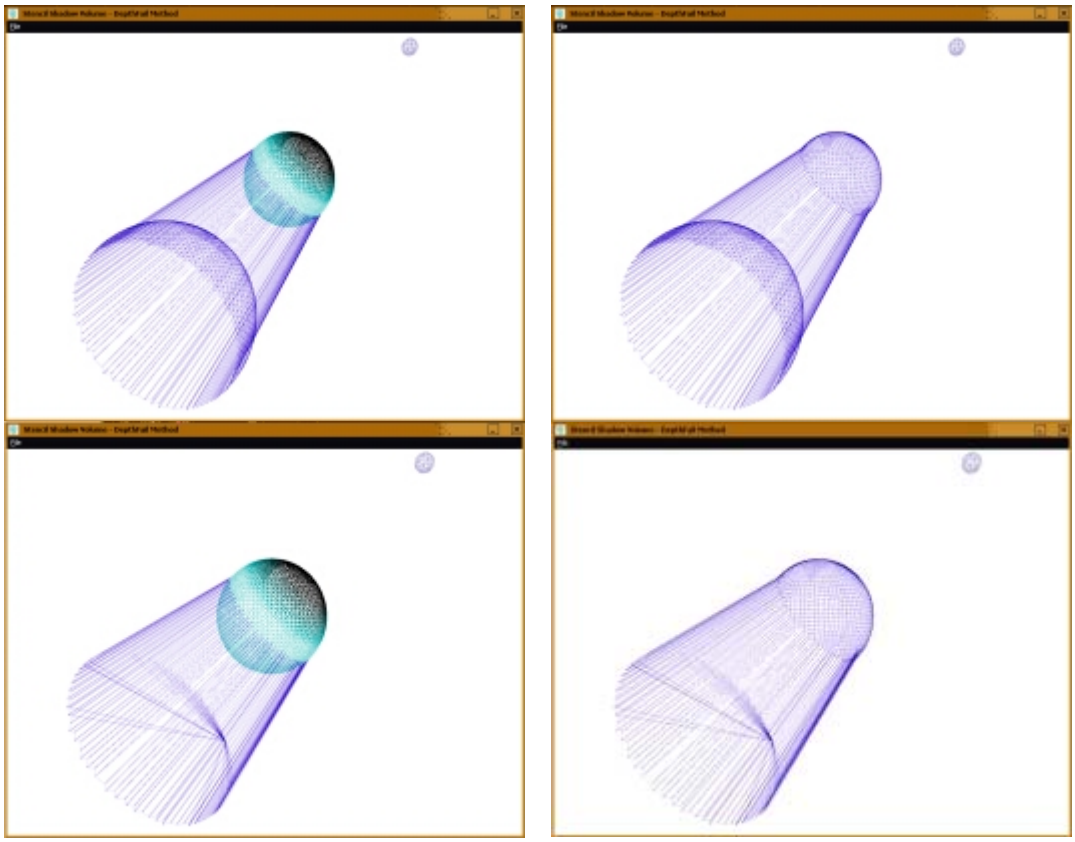


Figure 11: Closed shadow volume with point light source

Figure 11 shows two sets of images employing different geometries to close the shadow volume. The first row depicts a closed shadow volume formed by a front and back capping reusing light facing geometries. The second row shows a closed shadow volume with a front cap that reuses light facing geometries of the occluder and a triangle-fan back cap constructed from extruded silhouette edges. The triangle-fan back cap should be used as it results in less geometry and hence requires less memory and rendering time. When reusing the front facing geometries of the occluder, we should be extremely careful with regards to rendering the shadow volume since the shadow volume's front capping geometries are physically coplanar with the occluder's front facing geometries. Most often than not, precision problems will cause the front capping geometries of the shadow volume to be rendered in front of the occluder's front facing geometries causing the entire occluder to be engulfed in its own shadow volume. We can make use of the D3DRS_ZBIAS flag in Direct3D's D3DRENDERSTATETYPE to force the occluder's front facing geometries to be rendered in front of its shadow volume front cap. Simply use the D3DRS_ZBIAS flag when setting the render state (e.g. pd3dDevice->SetRenderState(D3DRS_ZBIAS, value)). We set the flag value to a higher value for the occluder's geometries and a lower value for its shadow volume. This will ensure that the front cap of the shadow volume is rendered behind the occluder's front facing geometries.

Extruding Geometries To Infinity

As discussed previously, we need to extrude the silhouette edges to infinity to avoid the situation shown in Figure 5 where a finite shadow volume extrusion fails to cover all the shadow receivers in a scene. However, it is not compulsory to extrude the silhouette edges to infinity if we can ensure that the situation in Figure 5 never happens in our scene. In practical cases, a large value would normally be more than adequate.

Mark Kilgard [2] introduced the trick of using the w value of homogenous coordinates to render semi-infinite vertices. In 4D homogenous coordinates, we represent a point or vector as (x, y, z, w) with w being the 4^{th} coordinate. For points, w is equal to 1.0. For vectors, w is equal to 0.0. The homogeneous notation is extremely useful for transforming both points and vectors. Since translation is only meaningful to points and not vectors, the value of w plays an important role in transforming only points and not vertices. This can be easily deduced since the translation values of a transformation matrix are on either the 4^{th} column or the 4^{th} row depending on the matrix convention. By setting the w value of the infinity-bound vertices to 0.0, we change the homogenous representation from that of a 3D point to a 3D vector. The rendering of a vector (w = 0.0) in clip space would be semi-infinite. It is important to note that we should only set the w values to 0.0 after transformation to clip space. In Direct3D, this would mean the combined transformation of the world, view and projection matrices. This is because when we set the flexible vertex format to D3DFVF_XYZRHW, we are bypassing Direct3D's transformation and lighting pipeline. Direct3D assumes that we had already transformed and lit the vertices. Ideally, the extrusion of geometries should be done in a vertex program since

we are already working in clip space in a vertex shader. In fact, vertex shaders and stencil shadow volumes is a match made in heaven. We will discuss the benefits of doing shadow volumes in a vertex program at the end of this paper.

While extruding geometries by a huge distance or to infinity helps to avoid the problem of finite shadow volume cover, it also generates another problem. Imagine two players in a dungeon First-Person-Shooter (FPS) game, roaming in adjacent rooms separated by a solid brick wall. The table lamp in one of the room causes one of the players to cast a shadow onto the brick separating the rooms. The player on the other room would see the shadow cast by the table lamp since the shadow volume extrudes out to infinity. The solid brick wall suddenly becomes like a thin piece of paper with a "ghost" shadow on it. Luckily, we can avoid this kind of situation in the very first place by culling away the shadow casting player's avatars using occlusion-culling techniques. Figure 12 shows a more awkward situation whereby the camera sees both the occluder and the occluder's ghost shadow on the other side of the terrain. This scenario is very possible especially for flight simulations or aerial combat games. The only possible solution to avoid both the finite shadow volume cover (Figure 5) and ghost shadow (Figure 12) is to impose limitations on the placing of light sources and occluders in a scene. If we can be sure that an occluder can never get closer than a certain distance of a shadow casting light source, then we can safely estimate the largest distance we would need to extrude the shadow volume in order to provide adequate shadow cover while not causing ghost shadows.

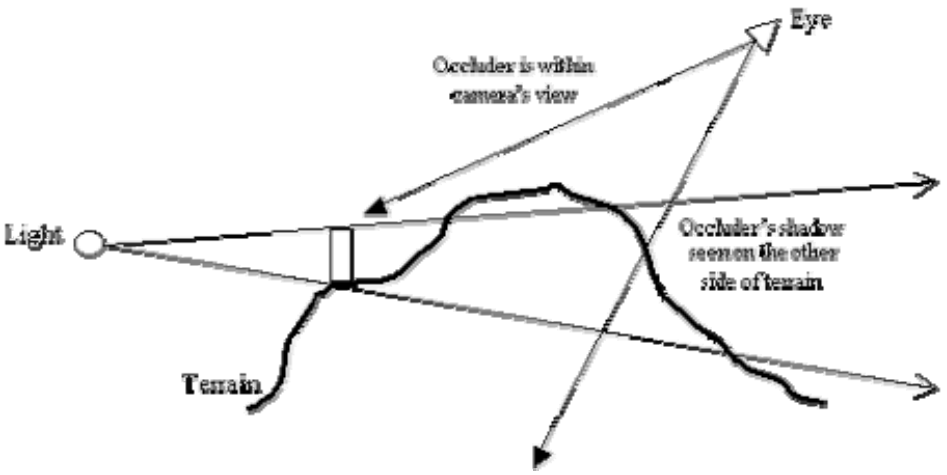


Figure 12: Ghost shadow effect due to large extrusion distance

View Frustum Clipping – The Ultimate Evil

It is time to confront the greatest evil in stencil shadow volumes: View frustum clipping. Clipping is a potential problem to any 3D rendering technique because we rely on a perspective projection view of our 3D worlds. The view frustum requires a near clipping distance and a far clipping distance, for the creation of a near clip plane and a far clip plane. Both the depth-pass and depth-fail

techniques suffer from view frustum clipping problem. Depth-pass technique suffers from errors when the shadow volume gets clipped after intersecting the near clip plane as shown in Figure 13. The red arrow represents one case whereby the stencil values for the associated fragment will be wrong due to the clipping of the shadow volume's front face.

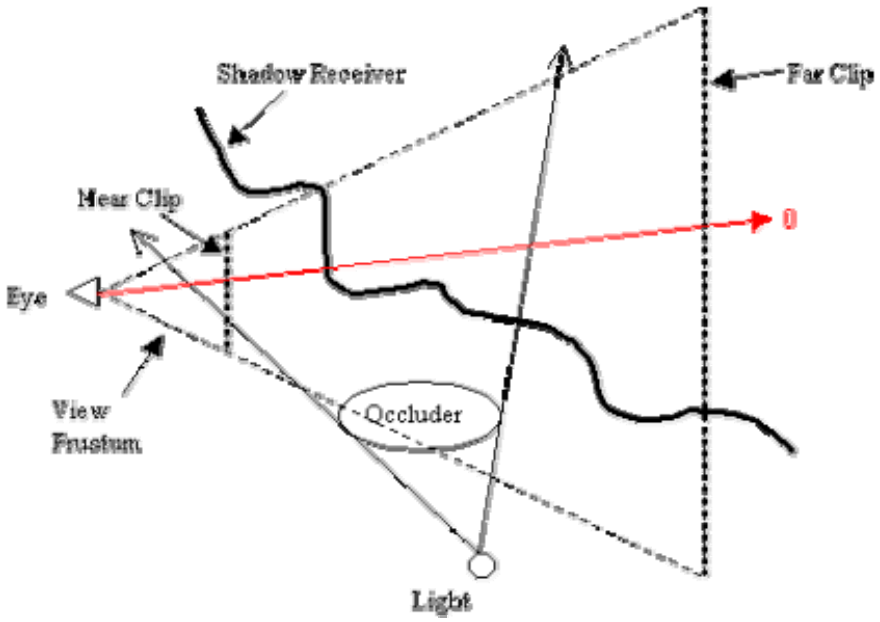


Figure 13: Shadow volume clipped at near clip plane causing depth-pass errors

On the other hand, depth-fail technique suffers from errors arising due to the clipping of the shadow volume with the far clip plane. Since the far clip plane is at a finite distance from the eye position, the depth-fail technique will almost certainly produce the wrong result when the shadow volume gets clipped at the far plane. The red arrow in Figure 14 represents a case whereby the depth-fail technique will generate errors since the back face of the shadow volume had been clipped at the far plane.

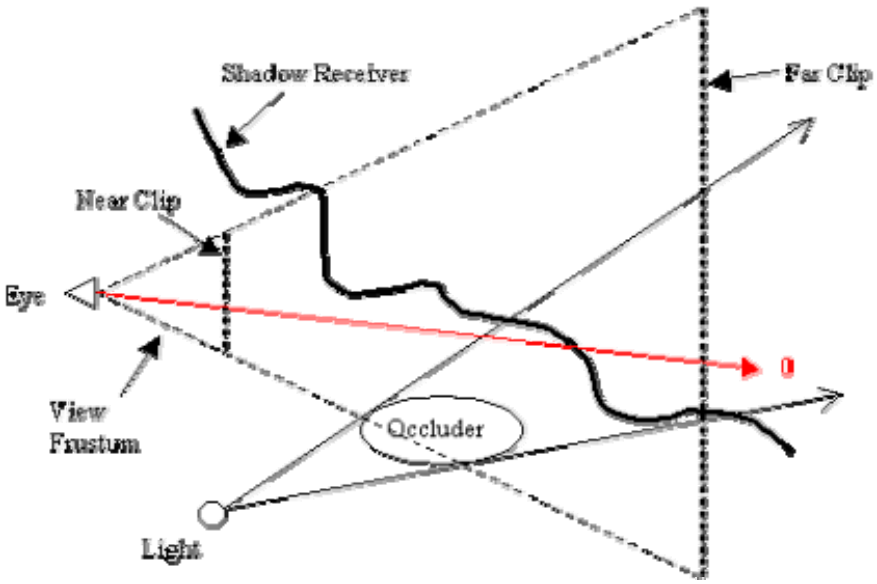


Figure 14: Shadow volume clipped at far clip plane causing depth-fail errors

We can solve the clipping problems by adjusting the clipping planes, but it is not always advisable to do so. For example, moving the near clip plane will greatly affect the depth precision and may have negative impacts on other operations that uses the depth buffer.

Mark Kilgard [2] presented an interesting idea of handling the two possible scenarios when shadow volumes intersect the near clip plane. The idea was to "cap" the shadow volume at the near clip plane, so that the previously clipped front facing geometries can now be rendered at the near clip plane. The first scenario is when all the vertices of the occluder's silhouette projects to the near clip plane. In this case, a quad strip loop is generated from all front facing vertices within the silhouette of the occluder. The quad strip loop is then projected onto the near clip plane thus forming a capping for the shadow volume.

The second scenario occurs when only part of the shadow volume projects onto the near clip plane. This proves to be very much more difficult to handle than the previous scenario. To his credit, Kilgard devised an elaborate system to filter out the vertices of triangles (facing away from the light) that should be projected onto the near clip plane in order to cap the shadow volume. The capping of shadow volumes at the near clip plane gave rise to another problem: depth precision. Rendering geometries at the near clip plane is analogous to rolling a coin along a razor's edge; the coin can drop down both sides easily. What this means is that the near plane may still clip the vertices that were meant to cap the shadow volume. To overcome this, Kilgard devised yet another method that builds a depth range "ledge" from the eye point to the near plane. The idea is to render the shadow volume from a depth range of 0.0 to 1.0, while normal scene rendering occurs within a depth range of 0.1 to 1.0. The ledge could be build into the view frustum by manipulating the perspective projection matrix. Once in place, the near clip plane capping of shadow volumes is done at a depth value of 0.05, which is half of the ledge. This idea is indeed original but it does not solve the problem totally. Cracks or "holes" in the near plane shadow cap occurs very frequently resulting in erroneous results. The conclusion with the near clip plane problem is that there are really no trivial solutions. At least, there is no known foolproof solution to the problem at the time of this writing. This makes the depth-pass technique very undesirable.

Fortunately, there is an elegant solution to the far plane clipping problem that plagues the depth-fail technique. The antidote to the problem is simply to use an infinite perspective view projection or simply an infinite view frustum. By projecting a far plane all the way to infinity, there is no mathematical chance of the shadow volume being clipped by the far plane when we are rendering the shadow volume. Even if the shadow volume were extruded to infinity, the far plane at infinity would still not clip it! Eric Lengyel presented the mathematic derivation for OpenGL perspective projection matrix in [11]. We are going to deal with Direct3D perspective projection matrix here. Lets start by looking at a standard left-handed perspective projection matrix in Direct3D:

$$P = \begin{bmatrix} \cot\left(\frac{fov_w}{2}\right) & 0 & 0 & 0 \\ 0 & \cot\left(\frac{fov_h}{2}\right) & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & \frac{-fn}{f-n} & 0 \end{bmatrix} \quad (1)$$

Variables:
n: near plane distance
f: far plane distance
fov_w: horizontal field of view in radians
fov_h: vertical field of view in radians

A far plane at infinity means that the far plane distance needs to approach ∞. Hence, we get the following perspective projection matrix when the far plane distance goes towards the infinity limit:

$$P_\infty = \lim_{f \to \infty} P = \begin{bmatrix} \cot\left(\frac{fov_w}{2}\right) & 0 & 0 & 0 \\ 0 & \cot\left(\frac{fov_h}{2}\right) & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & -n & 0 \end{bmatrix} \quad (2)$$

Equation (2) defines a perspective projection view that extends from the near plane to a far plane at infinity. But, are we absolutely sure that the vertices that we extruded to infinity using the 4D homogeneous vector does not get clipped at infinity? Sadly, we cannot be 100% sure of this due to limited hardware precision. In reality, graphics hardware sometimes produces points with a normalized z-coordinate marginally greater than 1. These values are then converted into integers for use in the depth buffer. This is going to weak havoc since our stencil operations depends wholly on the depth value testing. Fortunately, there is workaround for this problem. The solution is to map the z-coordinate values of our normalized device coordinates from a range of [0, 1] to [0, 1-e], where e is a small positive constant. What this means is that we are trying to map the z coordinate of a point at infinity to a value that is slightly less than 1.0 in normalized device coordinates. Let $\boldsymbol{D_z}$ be the original z-coordinate value and $\boldsymbol{D\varepsilon_z}$ be the mapped z-coordinate. The mapping can be achieved using equation (3) shown below:

$$D'_z = D_z(1-\varepsilon) \quad (3)$$

Now, let's make use of equation (2) to transform a point ***A*** from camera space (***A*****cam**) to clip space (***A*****clip**). Note that camera space is also commonly referred to as eye space.

$$A_{clip} = P_\infty A_{cam} = \begin{bmatrix} A_x & A_y & A_z & A_w \end{bmatrix} \begin{bmatrix} \cot\left(\frac{fov_w}{2}\right) & 0 & 0 & 0 \\ 0 & \cot\left(\frac{fov_h}{2}\right) & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & -n & 0 \end{bmatrix}$$

Which would gives us:

$$A_{clip} = \begin{bmatrix} A_x \cot\left(\frac{fov_w}{2}\right) \\ A_y \cot\left(\frac{fov_h}{2}\right) \\ A_z - nA_w \\ A_z \end{bmatrix} \quad (4)$$

Let's factor the desired range mapping into equation (3) by replacing D_z with $\frac{(A_{clip})_z}{(A_{clip})_w}$ and $D\varepsilon_z$ with $\frac{(A'_{clip})_z}{(A_{clip})_w}$:

$$\frac{(A'_{clip})_z}{(A_{clip})_w} = \frac{(A_{clip})_z}{(A_{clip})_w}(1-\varepsilon) \quad (5)$$

Simplifying equation (5) by using the values given by equation (4), we get:

$$(A'_{clip})_z = A_z(1-\varepsilon) + nA_w(\varepsilon - 1) \quad (6)$$

Using equation (6), we can enforce our range mapping into the projection matrix P_∞ given by equation (2) to get the following:

$$P'_\infty = \begin{bmatrix} \cot\left(\frac{fov_w}{2}\right) & 0 & 0 & 0 \\ 0 & \cot\left(\frac{fov_h}{2}\right) & 0 & 0 \\ 0 & 0 & (1-\varepsilon) & 1 \\ 0 & 0 & n(\varepsilon - 1) & 0 \end{bmatrix} \quad (7)$$

Thus, we can use the perspective projection matrix given in equation (7) without fear of far plane clipping of shadow volumes occurring at infinity! You might

wonder whether stretching the view frustum volume all the way to infinity would impact the depth buffer precision. The answer is, yes it does affect precision, but the loss of precision is really negligible. The amount of numerical range lost when extending the far plane out to infinity is only $\frac{n}{f}$. Say our original near clip plane is at 0.1 meter and far clip plane is at 100 meters. This range corresponds to a depth range of [-1.0, 1.0]. We then extend the far plane distance to infinity. The range from 0.1 meter to 100 meters would now correspond to a depth range of [-1, 0.999]. The range from 100 meters to infinity would correspond to a depth range of [0.999, 1.0]. The loss in depth buffer precision is really not a big impact at all. The larger the difference between the *n* and *f* values, the smaller the loss in depth buffer precision. You can find the above derivations and many other related mathematical derivations in Eric Lengyel's book [12]. It should be noted that using an infinite view frustum meant that we have to draw more geometries. This may pose a potential performance problem.

The infinite view frustum projection is really just a software solution to the far plane clipping problem. Mark Kilgard and Cass Everitt [10] presented a hardware solution to the problem instead of using an infinite view frustum. Newer graphics hardware now supports a technique called "depth-clamping". In fact, the depth-clamping extension, NV_depth_clamp, was specifically added to Nvidia's GeForce3 and above graphics cards to solve the far plane clipping problem for shadow volumes. When active, depth-clamping would force all the objects beyond the far clip plane to be drawn at the far clip plane with the maximum depth value. This meant that we can project the closed shadow volume to any arbitrary distance without fear of it being clipped by the far plane as the hardware will handle the drawing properly. With such automatic support from graphics hardware, depth-fail shadow volumes become very easy to implement. We can extend the shadow volume to infinity while rendering with our finite view frustum and still get correct depth-fail stencil values! Well, the tradeoff is hardware dependence. If we want the depth-fail shadow volume to work for any graphics card (with stenciling support), we will have to use the infinite view frustum projection instead of the depth-clamping extension.

Depth-Pass or Depth-Fail

We had run through most of the method and implementation issues of both the depth-pass and depth-fail techniques for doing stencil shadow volumes. So which method should we use in our games? Lets take stock of the pros and cons of both techniques.

Depth-pass

- Advantages
  - Does not require capping for shadow volumes
  - Less geometry to render
  - Faster of the two techniques
  - Easier to implement if we ignore the near plane clipping problem

    - Does not require an infinite perspective projection
- Disadvantages
    - Not robust due to unsolvable near plane clipping problem

Depth-fail

- Advantages
    - Robust solution since far plane clipping problem can be solved elegantly
- Disadvantages
    - Requires capping to form closed shadow volumes
    - More geometry to render due to capping
    - Slower of the two techniques
    - Slightly more difficult to implement
    - Requires an infinite perspective projection

It seems that depth-pass is the better technique of the two, but we must remember that it will totally fail when our camera enters a shadow volume. Until there is a feasible solution for the near plane clipping problem, the depth-fail technique is still required if a robust implementation is desired. The selection between the two techniques depends heavily on the constraints of the games that we are developing. If shadow casting is required for a top down or isometric view game such as Diablo, the depth-pass technique will suffice. On the other hand, for FPS games, it would be almost impossible to avoid the situation of camera entering a shadow volume. In this case, depth-fail technique is the only feasible solution. Of course, we must also not forget about other shadowing techniques such as shadow mapping. In certain situations where the shadow casters in a scene are too small for any self-shadowing to be visible, it would be wiser to just use projective shadow mapping. For realistic soft shadows, it may also be done more cheaply using shadow maps.

On the whole, it is beneficial to combine other techniques with shadow volumes to achieve better quality shadows. One example of such hybrid implementation is the Power Render X [16] game engine that generates shadows using shadow volumes and then fade out the shadows with respect to distance from the occluder by using projective textures.

The Buzzwords: Robustness and Efficiency

Doing realistic and accurate shadows in games is no longer enough as the complexity of games had skyrocketed during the past 10 years. We need to provide robust and yet efficient implementations of stencil shadow volumes. In the case of robustness, using the depth-fail technique should suffice for almost any situations imaginable. However, hardware limitations and poor frame rates will sometimes push the depth-fail technique beyond our computation budget. There are many ways to optimize our shadow volume implementation so as to

create nice looking shadows and yet hold the frame rate above that all-important 20fps benchmark.

The real bottlenecks in a stencil shadow volume implementation are silhouette determination and shadow volume rendering. The former requires a huge amount of CPU cycles and it worsens if the occluders had high polygon counts. The latter is a huge consumer of invisible fill rate. One obvious way to alleviate the CPU crunch during silhouette determination is to use a lower polygon model of the occluder. Another effective way is to determine a new silhouette only every 2-4 frames. This is based on the assumption that the light's position or the occluder's position does not change very drastically within 2-4 frames. This assumption turns out to be pretty good for most cases.

Remember that the extra capping geometries used to form a closed shadow volume in the depth-fail technique contributed to depth-fail being a more expensive method? We can drastically reduce the capping geometries for occluders that have relatively little detail on the surfaces that frequently face the light. Little detail here means fewer geometric details, which implies that the surface is rather flat and would usually produce near or fully convex silhouette hulls. If that is the case, we can often create a triangle strip to be used as the front cap to close the shadow volume. We should note that this is an approximation and hence would result in shadows that are not correct at certain angles. However this approximation should work very well for small objects.

For Direct3D implementations, it is also advisable to use "welded" meshes. A welded mesh simply means that there are no duplicated vertices representing the exact same point. To see an example of an "unwelded" mesh, open the mesh viewer tool and create a cube. Look at the vertices information of the cube and you will see that there are 24 instead of just 8 vertices. This is unavoidable since Direct3D's version of a vertex contains color and normal information that cannot be shared by different faces referring to the same point; hence extra vertices are generated for different faces. The extra vertices are redundant but could not be removed during the silhouette calculation without considerable amount of comparison work. It is therefore wiser to use welded meshes for silhouette determination. The Direct3D mesh viewer utility provides a nifty option to do just that. Click MeshOps then Weld Vertices, check Remove Back To Back Triangles, Regenerate Adjacency and Weld All Vertices before welding. Alternatively, we can also make use of the mesh function D3DXWeldVertices to weld the mesh ourselves.

For Direct3D implementations, it is also advisable to use "welded" meshes. A welded mesh simply means that there are no duplicated vertices representing the exact same point. To see an example of an "unwelded" mesh, open the mesh viewer tool and create a cube. Look at the vertex information for the cube and you will see that there are 24 instead of just 8 vertices. This is unavoidable since Direct3D's version of a vertex contains color and normal information that cannot be shared by different faces referring to the same point; hence extra vertices are generated for different faces. The extra vertices are redundant but could not be removed during the silhouette calculation without considerable amount of comparison work. It is therefore wiser to use welded meshes for

silhouette determination. The Direct3D mesh viewer utility provides a nifty option to do just that. Click MeshOps then Weld Vertices, check Remove Back To Back Triangles, Regenerate Adjacency and Weld All Vertices before welding. Alternatively, we can also make use of the mesh function D3DXWeldVertices to weld the mesh ourselves.

Regarding the invisible fill rate, they are really unavoidable. However, we could probably lessen the impact by setting the D3DRS_COLORWRITEENABLE render state in Direct3D before rendering the shadow volume. We can use it to turn off the red, green, blue and alpha channel drawing since we are only interested in filling the stencil buffer.

Another area that we should take note of is the management of shadow casting lights in our 3D scene. Good management of light sources will invariantly benefit the shadow volume generation process. The rule of thumb is to keep the number of shadow casting light sources below a maximum of 4 at any one time. Future hardware or improved algorithms would nullify the previous statement, but for now it serves as a good guideline and would probably remain so for the next 2 years at least. The important aspect of light source management is the method used for selecting which light sources should be included in the shadow volume generation process. The main parameters that should be taken into considerations could be intensity, distance from viewer, relevance to current game play and lastly visual importance. Take a look at the excellent article [4] by Charles Bloom regarding the selection of light sources for shadow casting.

Let's discuss some high level optimization that we can employ to speed up our shadow volume enabled games further. We can actually make use of the depth-pass technique when we are sure that the camera is not within any shadow volumes. This can be done rather easily by forming a near-clip volume. The light source's position and the four sides of the near plane are used to define a pyramid. The near plane closes the pyramid and thus forms the near-clip volume. If an occluder lies completely outside this volume, we can safely employ the depth-pass technique since the occluder's shadow volume has no chance of intersecting the near plane. Eric Lengyel also described utilizing OpenGL scissor rectangle support to cut down the fill rate penalty for rendering the shadow volumes and the illuminated fragments. However, comprehensive high-level scissor rectangle support is not yet available in DirectX 8.1. For details of these two optimizations, please refer to [11].

Lastly, we should aggressively utilize any hardware supports that are available. Future GPUs would be expected to support two-sided stencil testing, which will allow us to render the front and back faces of shadow volumes together. This would provide great savings when rendering the shadow volume by halving the geometry setup cost, vertex transformation cost and geometry transfer cost since we would only need to push the shadow volume geometries through the pipeline once. The hardware will take care of the front face and back face culling automatically while doing 2 stenciling pass on the same set of geometries. Hardware depth-clamping support should also be used to clip the shadow volume geometries to the far plane at no extra costs. Finally, lets look

at one of the most important modern graphics hardware capability that we should take full advantage of: **Vertex Shaders**.

Shadow Volumes Powered By Vertex Shaders

Among the whole host of improvements to commercial graphics hardware, the introduction of programmable vertex processing pipeline (vertex shaders) is perhaps the best thing that can happen to anyone implementing shadow volumes. The biggest advantage of doing shadow volumes in a vertex program is that we do not need to upload the shadow volume geometries whenever it is generated. The entire shadow volume could reside on hardware memory as static vertex buffers. The data bandwidth saved can be quite substantial. Furthermore, floating point operations done in programmable vertex hardware are incredibly fast. However, we need to note here that implementing shadow volume fully using vertex program may actually degrade performance in certain circumstances. We will go into this at the end of this section.

To leverage the power of vertex shaders, we need to preprocess our occluder's geometry first. Current vertex shader hardware does not have the capability of generating new vertices on the fly. It is strictly a 1 vertex in and 1 vertex out pipeline. This poses a problem since we need to create new vertices from the silhouette edges in order to form a shadow volume. The solution is to create all the additional vertices that are needed during preprocessing. Once in the vertex shader, we generate the shadow volume using these additional vertices. Lets look at how this is done.

We need to create a quad for every edge (2 vertices) that is shared by exactly 2 faces. The quad can be viewed as a "degenerate" quad formed by the original edge shared by 2 different faces. Both the faces contribute the same edge to the degenerate quad. Since the edges from both faces are similar, positional wise, the degenerate quad is "zero length". The only difference is that the edges hold the normal information of their respective face. Once in the vertex program, we dot the light vector and the vertex normal. If the result is positive, the vertices pass through the vertex program untouched. If the result is negative, we extrude it in the direction of the light vector. This technique would elegantly generate a closed shadow volume as light facing geometries are left untouched to form the front capping while geometries that faces away from the light are extruded to form the sides of the shadow volume and the back capping.

If you are unsure about how it works, try this example. Imagine a sphere mesh with a point light source to its left. The entire left hemisphere faces the light and hence all the geometries that define the left hemisphere are left untouched to form the front capping. The entire right hemisphere however faces away from the light. Hence, all the geometries that define the right hemisphere are

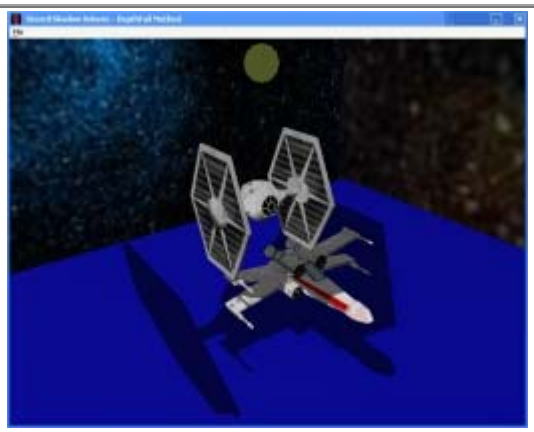
extruded to form the back capping. The sides of the shadow volume are formed auto-magically by the degenerated quads residing along the silhouette edges. In this case, the silhouette edges forms exactly a vertical line down the middle of the sphere. This works because exactly 1 edge per degenerate quad from the silhouette edges is extruded. The previously degenerate quad now becomes a normal quad that defines the shadow volume's sides. Chris Brennan presented a short article in [15] on implementing the extrusion of shadow volume in a vertex program.

We should note that the preprocessing needed creates a lot of additional geometries. In fact, only the degenerate quads along the silhouette edges are useful. The rest are simply dormant but are still being pushed through the processing pipeline. However, shadow volume generation can now be done completely on the graphics hardware and performance is generally much better than non-shader implementation in most cases.

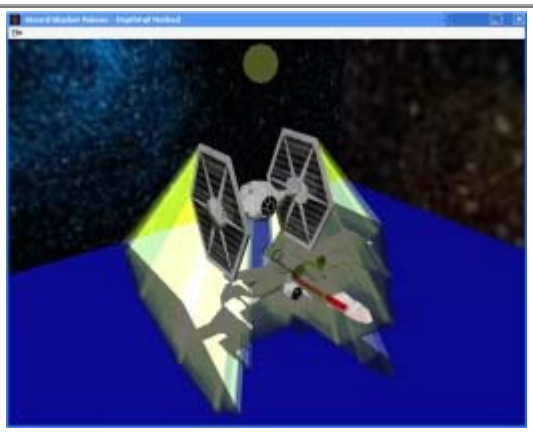
Recently Mark Kilgard pointed out that computing the silhouette edges within the vertex shader may be detrimental to performance if the occluders have high polygon counts or if there are a lot of shadows casting light sources. This assessment stern from the fact that we need to push more vertices into the pipeline and all of these have to passed through the silhouette edges testing within the vertex shader. Consequently, occluders with high polygon counts would generate large amount of wasted vertices (degenerate quads), and the cost of testing all these extra vertices may not cover the geometry upload savings we get by using vertex shaders! Having more light sources will obviously worsen such vertex shader implementation further. Hence, implementation of shadow volume on programmable vertex hardware should be thoroughly tested to ensure that we have a net performance gain over implementation utilizing the CPU. If the CPU is needed for heavy A.I. or game logic computation, a vertex shader implementation of shadow volumes may be more efficient. However, it might also be better in many cases to just use vertex shader as an assist instead of trying to do everything within the vertex shader. The moral of the story is: ***always remember to turn on everything (A.I., Physics, Sound, Input, Network, Renderer etc) in your game and benchmark, benchmark and benchmark again!***

Lastly, a more extensive and in-depth article on the stencil shadow volume technique will be available in the upcoming book ShaderX2 (www.shaderx2.com). The article in the book delves deeper into the algorithms involved in stencil shadow volume with detailed discussions of optimizations, workflow, and scene management and ‘cheats' employed in commercial 3D engines to speed up robust shadow volume implementations. There will also be 6 extensive samples that cover normal CPU, GPU implementation in assembly and GPU implementation using the new High Level Shader Language (DirectX9.0). The book is a compilation of many advance shader techniques by professionals and engineers working in the field. It will be available possibly in August 2003 and the editor is Mr Wolfgang Engel.

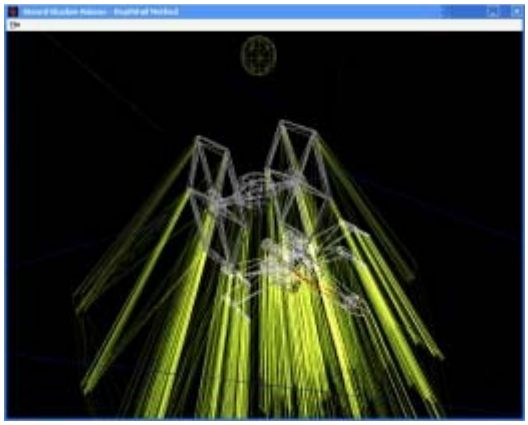
Shadow Volumes At Work



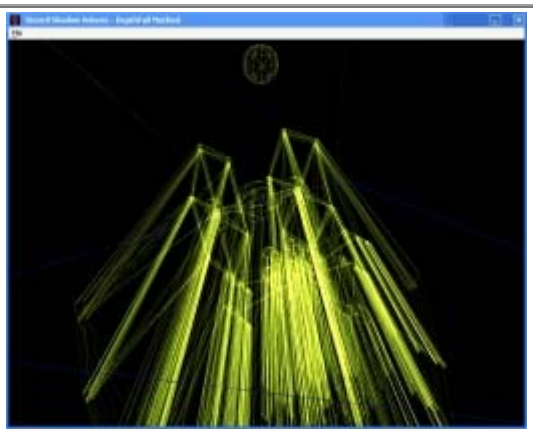
Depth-fail stencil shadow volume technique. Showcasing accurate self-shadowing and multiple occluder shadowing.



Front faces of shadow volume drawn to give a visual appreciation of the silhouette extrusion from the point light source.



Wire frame of the occluders and the shadow volume.



The occluder's geometries were omitted to show the front and back capping of the shadow volume. The light facing geometries forms the front cap.

Near plane clipping of shadow volume causes errors when the camera enters the shadow volume in the depth-pass technique.

Far plane clipping of shadow volume cause errors in the depth-fail technique.

Many thanks to Augustin Denis for providing the XWing and Tie fighter models.
(These models are fan art)

Reference

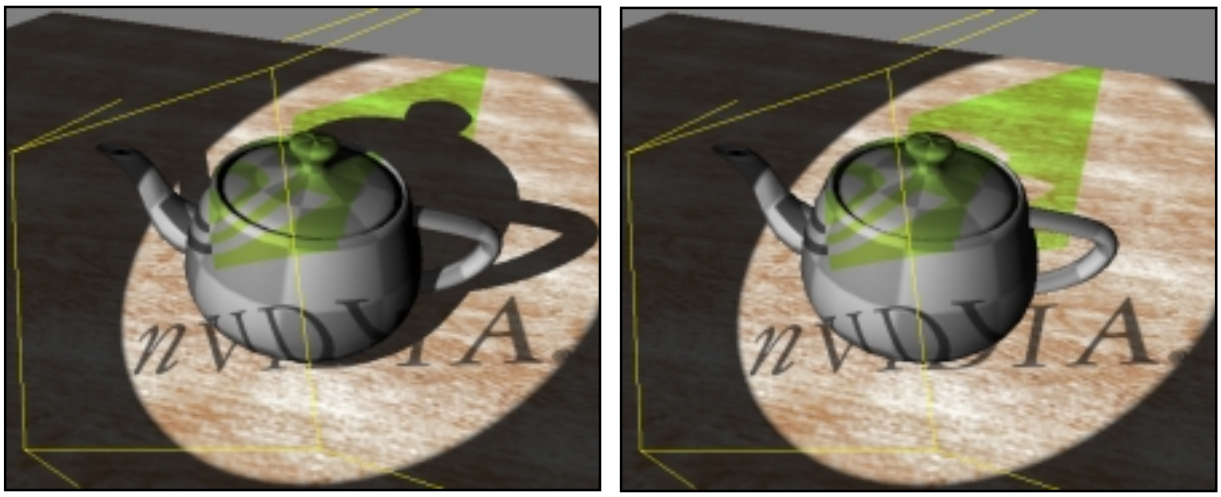
[1]: Mark Kilgard, http://developer.nvidia.com/docs/IO/1348/ATT/stencil.pdf
[2]: Mark Kilgard, http://developer.nvidia.com/docs/IO/1451/ATT/StencilShadows_CEDEC_E.pdf
[3]: http://developer.nvidia.com/view.asp?IO=inf_shadow_volumes
[4]: Charles Bloom, http://www.cbloom.com/3d/techdocs/shadow_issues.txt
[5]: Tim Heidmann, http://developer.nvidia.com/docs/IO/2585/ATT/RealShadowsRealTime.pdf
[6]: John Carmack, http://developer.nvidia.com/docs/IO/2585/ATT/CarmackOnShadowVolumes.txt
[7]: Bilodeau, Bill and Mike Songy. "Real Time Shadows", Creativity 1999, Creative Labs Inc. Sponsored game developer conferences, Los Angeles, California, and Surrey, England, May 1999.
[8]: Frank Crow. Shadows Algorithms for Computers Graphics. Computer Graphics, Vol. 11, No.3, Proceedings of SIGGRAPH 1977, July 1977.
[9]: Cass Everitt and Mark Kilgard, http://developer.nvidia.com/docs/IO/2585/ATT/RobustShadowVolumes.pdf
[10]: Cass Everitt and Mark Kilgard, http://developer.nvidia.com/docs/IO/2585/ATT/GDC2002_RobustShadowVolumes.pdf
[11]: Eric Lengyel, http://www.gamasutra.com/features/20021011/lengyel_01.htm
[12]: Eric Lengyel, "Mathematics for 3D Game Programming & Computer Graphics", Charles River Media, 2002
[13]: Tomas Moller, Eric Haines. "Realtime Rendering", 2nd Edition, A K Peters Ltd, 2002, ISBN: 1-56881-182-9.
[14]: Wolfgang F. Engel, Amir Geva and Andre LaMothe. "Beginning Direct3D Game Programming", Prima Publishing, 2001, ISBN: 0-7615-3191-2.
[15]: Wolfgang F. Engel. "Direct3D ShaderX Vertex and Pixel Shader Tips and Tricks", Wordware Publishing Inc, 2002.
[16]: Power Render X game engine. http://www.powerrender.com/prx/index.htm
[17]: Gamasutra website. http://www.gamasutra.com/

Hardware Shadow Mapping

Cass Everitt
cass@nvidia.com

Ashu Rege
arege@nvidia.com

Cem Cebenoyan
cem@nvidia.com



Introduction

Shadows make 3D computer graphics look better. Without them, scenes often feel unnatural and flat, and the relative depths of objects in the scene can be very unclear. The trouble with rendering high quality shadows is that they require a visibility test for each light source at each rasterized fragment. For ray tracers, adding an extra visibility test is trivial, but for rasterizers, it is not. Fortunately, there are a number of common cases where the light visibility test can be efficiently performed by a rasterizer. The two most common techniques for hardware accelerated complex shadowing are stenciled shadow volumes and shadow mapping. This document will focus on using shadow mapping to implement shadowing for spotlights.

Shadow mapping is an image-based shadowing technique developed by Lance Williams [8] in 1978. It is particularly amenable to hardware implementation because it makes use of existing hardware functionality – texturing and depth buffering. The only extra burden it places on hardware is the need to perform a high-precision scalar comparison for each texel fetched from the shadow map texture. Shadow maps are also attractive to application programmers because they are very easy to use, and unlike stenciled shadow volumes, they require no additional geometry processing.

Hardware accelerated shadow mapping [5] is available today on GeForce3 GPUs. It is exposed in OpenGL [4] through the SGIX_shadow and SGIX_depth_texture extensions [6], and in Direct3D 8 through a special texture format.

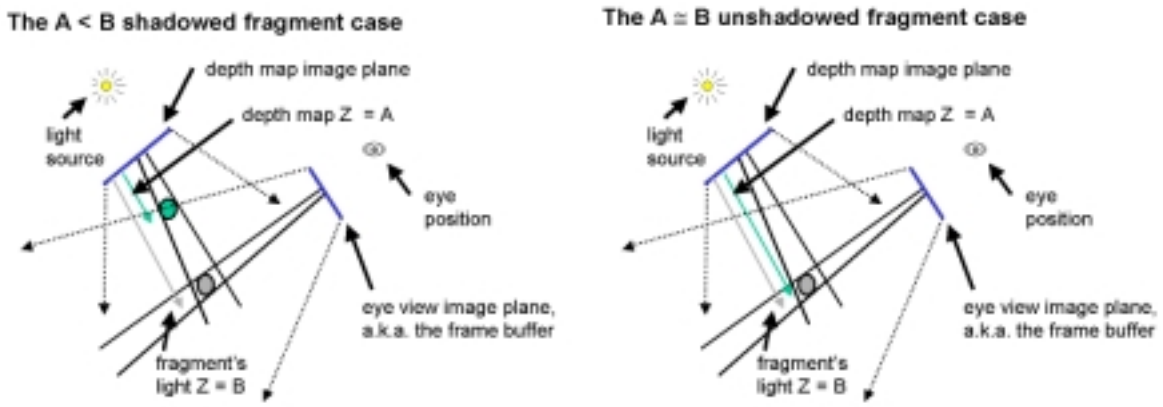


Figure 1. These diagrams were taken from Mark Kilgard's shadow mapping presentation at GDC 2001. They illustrate the shadowing comparison that occurs in shadow mapping.

How It Works

The clever insight of shadow mapping is that the depth buffer generated by rendering the scene from the light's point of view is a pre-computed light visibility test over the light's view volume. The light's depth buffer (a.k.a. the shadow map) partitions the view volume of the light into two regions: the shadowed region and the unshadowed region. The visibility test is of the form

$$p_z \leq shadow\_map(p_x, p_y)$$

where p is a point in the light's image space. Shadow mapping really happens in the texture unit, so the comparison actually looks like:

$$\frac{p_r}{p_q} \leq texture\_2D\left(\frac{p_s}{p_q}, \frac{p_t}{p_q}\right)$$

Note that this form of comparison is identical to the depth test used for visible surface determination during standard rasterization. The primary difference is that the rasterizer always generates fragments (primitive samples) on the regular grid of the eye's discretized image plane for depth test, while textures are sampled over a continuous space at irregular intervals. If we made an effort to sample the shadow map texture in the same way that we sample the depth buffer, there would be no difference at all. In fact, we can use shadow maps in this way to perform more than one depth test per fragment [2].

Figure 1 illustrates the depth comparison that takes place in shadow mapping. The eye position and image plane are shown, but they are not relevant to the visibility test because shadowing is view-independent.

Figure 2. A shadow mapped scene rendered from the eye's point of view (left), the scene as rendered from the light's point of view (center), and the corresponding depth/shadow map (right).

How To Do It

The basic steps for rendering with shadow maps are quite simple:

- render the scene from the light's point of view,
- use the light's depth buffer as a texture (shadow map),
- projectively texture the shadow map onto the scene, and
- use "texture color" (comparison result) in fragment shading.

Figure 2 shows an example scene with shadows, the same scene shown from the light's point of view, and the corresponding shadow map (or depth buffer). Note that samples that are closer to the light are darker than samples that are further away.

Since applications already have to be able to render the scene, rendering from the light's point of view is trivial. If it is available, polygon offset should be used to push fragments back slightly during this rendering pass.

Why Is Polygon Offset Needed?

If implemented literally, the light visibility test described in the previous section is prone to self-shadowing error due to it's razor's edge nature in the case of unshadowed objects. In the hypothetical "infinite resolution, infinite precision" case, surfaces that pass the visibility test would have depth *equal* to the depth value stored in the shadow map. In the real world of finite precision and finite resolution, precision and sampling issues cause problems. These problems can be solved by adding a small bias to the shadow map depths used in the comparison.

If the problem were only one of precision, a constant bias of all the shadow map depths would be sufficient, but there is also a less obvious sampling issue that affects the magnitude of bias necessary. Consider the case illustrated in Figure 3. When the geometry is rasterized from the eye's point of view, it will be sampled in different

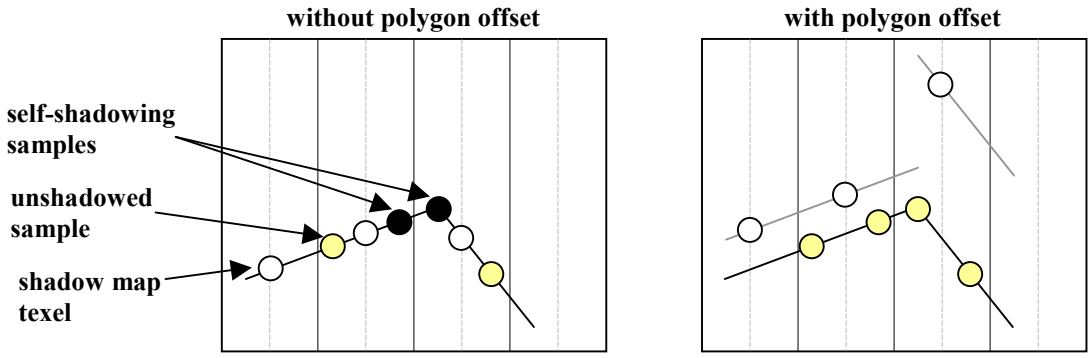


Figure 3. These figures illustrate the need for polygon offsetting to eliminate self-shadowing artifacts. The variable sampling location necessitates the use of z slope-based offset.

locations than when it was rasterized from the light's point of view. The difference in the depths of the samples is based on the slope of the polygon in light space, so in order to account for this we must supply a positive "slope factor" (typically about 1.0) to the polygon offset.

Direct3D does not expose polygon offset, so applications must provide this bias through matrix tweaks. This approach is workable, but because it fails to account for z slope, the uniform bias is generally much larger than it would otherwise need to be, which may introduce incorrectly unshadowed samples, or "light leaking".

The depth map as rendered from the light's point of view *is* the shadow map. With OpenGL, turning it into a real texture requires copying it into texture memory via glCopyTex{Sub}Image2D(). Even though the copy is card-local, it is still somewhat expensive. Direct3D's render-to-texture capability makes this copy unnecessary. You can render directly to the shadow map texture. This render-to-texture capability will also be available soon in OpenGL through extensions.

Once the shadow map texture is generated, it is projectively textured onto the scene. For shadow mapping, we compute 3D projective texture coordinates, where *r* is the sample depth in light space, and *s* and *t* index the 2D texture. Figure 4 shows these quantities, which are compared during rendering to determine light visibility.

Figure 4. A shadow mapped scene (left), the scene showing each sample's distance from the light source (center), and the scene with the shadow map shadow map projected onto it (right).

The final step in rendering shadows is to actually factor the shadow computation result into the shading equation. The result of the comparison is either 1 or 0, and it is returned as the texture color. If linear filtering is enabled, the comparison is performed at the four neighboring shadow map samples, and the results are bilinearly filtered just as if they had been colors.

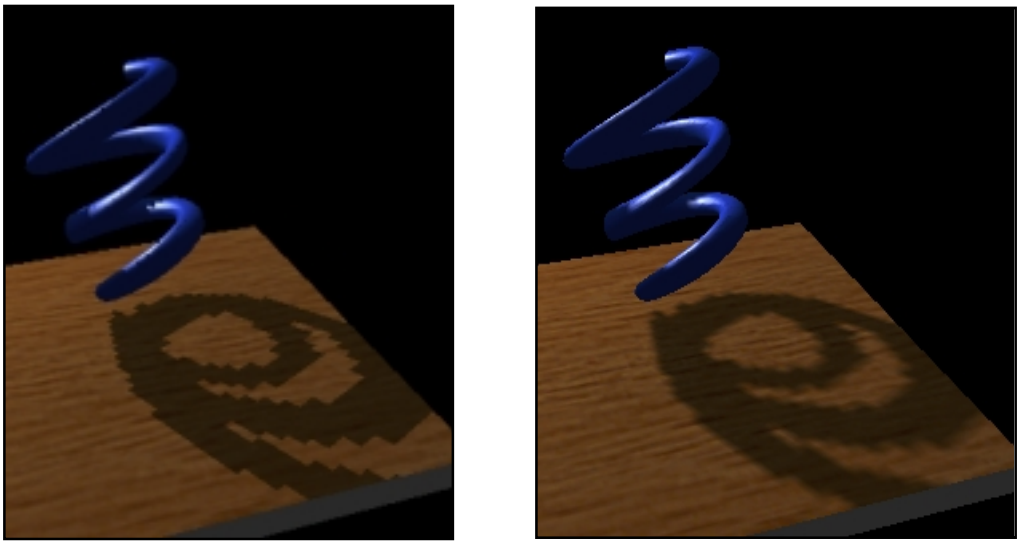


Figure 5. A very low resolution shadow map is used to demonstrate the difference between nearest (left) and linear (right) filtering for shadow maps. Credit: Mark Kilgard.

With GeForce3 hardware, it is easiest to use NV_register_combiners to implement the desired per-fragment shading based on the shadow comparison. One simple approach is to use the shadowing result directly to modulate the diffuse and specular intensity. Kilgard points out [3] that leaving some fraction of diffuse intensity in helps keep shadows areas from looking too "flat".

OpenGL API Details

Support for shadow mapping in OpenGL is provided by the SGIX_shadow and SGIX_depth_texture extensions. The SGIX_shadow extension exposes the per-texel comparison as a texture parameter, and SGIX_depth_texture defines a texture internal format of DEPTH_COMPONENT, complete with various bits-per-texel choices. It also provides semantics for glCopyTex{Sub}Image*() calls to read from the depth buffer when performing a copy.

Direct3D API Details

Support for shadow mapping in Direct3D is provided by special depth texture formats exposed in drivers version 21.81 and later. Support for both 24-bit (D3DFMT_D24S8) and 16-bit (D3DFMT_D16) shadow maps is included.

Setup

The following code snippet checks for hardware shadow map support on the default adapter in 32-bit color:

```
HRESULT hr = pD3D->CheckDeviceFormat(
        D3DADAPTER_DEFAULT,         //default adapter
        D3DDEVTYPE_HAL,             //HAL device
        D3DFMT_X8R8G8B8,            //display mode
        D3DUSAGE_DEPTHSTENCIL,      //shadow map is a depth/s surface
        D3DRTYPE_TEXTURE,           //shadow map is a texture
        D3DFMT_D24S8                //format of shadow map
);
```

Note that since shadow mapping in Direct3D relies on "overloading" the meaning of an existing texture format, the above check does not guarantee hardware shadow map support, since it's feasible that a particular hardware / driver combo could one day exist that supports depth texture formats for another purpose. For this reason, it's a good idea to supplement the above check with a check that the hardware is GeForce3 or greater.

Once shadow map support has been determined, you can create the shadow map using the following call:

```
pD3DDev->CreateTexture(texWidth, texHeight, 1,
        D3DUSAGE_DEPTHSTENCIL, D3DFMT_D24S8, D3DPOOL_DEFAULT,
        &pTex);
```

Note that you **must** create a corresponding color surface to go along with your depth surface since Direct3D requires you to set a color surface / z surface pair when doing a SetRenderTarget(). If you're not using the color buffer for anything, it's best to turn off color writes when rendering to it using the D3DRS_COLORWRITEENABLE renderstate to save bandwidth.

Rendering

Rendering uses the same ideas as in OpenGL: you render from the point of view of the light to the shadow map you created, then set the shadow map texture in a texture stage and set the texture coordinates in that stage to index into the shadow map at (s / q, t / q) and use the depth value (r / q) for the comparison. There are a few Direct3D-specific idiosyncrasies to be aware of, however:

- The (z / w) value used to compare with the value in the shadow map is in the range [0..$2^{bitdepth}$-1], not [0..1], where 'bitdepth' is the bitdepth of the shadowmap (24 or 16 bits). This means you have to put an additional scale factor into your texture matrix.
- Direct3D addresses pixels and texels in different ways [1], where integral screen coordinates address pixel centers and integral texture coordinates address texel boundaries. You need to take this into account when addressing the shadow map. There are two ways to do this: either offset the viewport by half a texel when rendering the shadow map, or offset by half a texel when addressing the shadow map.
- As stated earlier, there is no native polygon offset support in Direct3D. The closest thing is D3DRS_ZBIAS, but this doesn't help us when shadow mapping since it can only be used to bias depth a constant amount *towards* the camera, not away. Instead we can get similar functionality, albeit without taking into account polygon slope, by adding a small bias amount to our texture matrix.

Here is a sample texture matrix that takes into account these limitations:

```
float fOffsetX = 0.5f + (0.5f / fTexWidth);
float fOffsetY = 0.5f + (0.5f / fTexHeight);
D3DXMATRIX texScaleBiasMat( 0.5f,     0.0f,     0.0f,      0.0f,
                            0.0f,    -0.5f,     0.0f,      0.0f,
                            0.0f,     0.0f,     fZScale,   0.0f,
                            fOffsetX, fOffsetY, fBias,     1.0f );
```

Where fZScale is the ($2^{bitdepth}$-1) and fBias is a small negative value. Note that this matrix is applied post-projection, **not** in eye space.

Once the texture coordinates have been setup properly, the hardware will automatically compare (r / q) > shadowMap[s / q, t / q] and return zero to indicate in shadow or one to indicate in light (or potentially something in between if you're on the shadow edge and using D3DTEXF_LINEAR). The following pixel shader shows a simple use of shadow mapping (but note that you don't have to use pixel shaders to use shadow maps, DirectX7-style texture stage states work as well):

```
tex t0   // normal map
tex t1   // decal texture
tex t2   // shadow map
dp3_sat r0, t0_bx2, v0_bx2  //light vector is in v0
mul r0, r0, t2   //modulate lighting contribution by shadow result
mul r0, r0, t1   //modulate lighting contribution by decal
```

Advantages and Limitations

As with any technique, shadow mapping has certain advantages and limitations to be aware of. The fact that it is image-based turns out to be both an advantage and a limitation. It's advantageous, because it doesn't require additional application geometry processing, it works well with GPU-created and GPU-altered geometry and correctly handles fragment culling like alpha test. The associated limitation is that because it's image based, it works well for spotlights, but not point light sources. One could imagine a cube map –based shadow mapping system, but they would require six 90-degree frusta, which would each need to be fairly high resolution, and five more passes over the

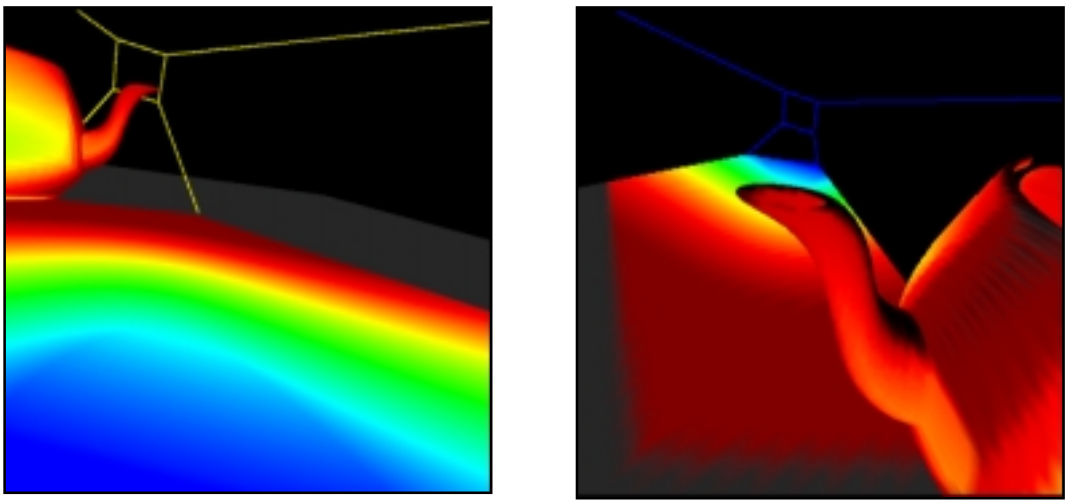


Figure 6. The "dueling frusta" problem occurs when the spotlight points toward the eye. The eye's view (left) shows the variation in sampling frequency of the shadow map, blue being the highest . The light's view (right) shows the very small portion of the light's image plane needs high frequency sampling.

geometry to generate the shadow map.

Along the same lines, the quality of shadow mapping depends on how well the shadow map sampling frequency matches the eye's sampling frequency. When the eye and light have similar location and orientation, the sampling frequencies match pretty well, but when the light and eye are looking toward each other, the sampling frequencies rarely match well. Figure 6 illustrates this "dueling frusta" situation.

Another problem that comes up with any projective texture mapping is the phantom "negative projection". This is actually pretty simple to remove at the cost of an additional texture unit, or per-vertex color. The goal is just to make sure that the shadow test always returns "shadowed" for surfaces behind the light.

Finally, the polygon offset fudge factor, while quite adequate for virtually all uses of shadow mapping, can be a bit dissatisfying. Andrew Woo [9] suggested an alternative shadow map generation that is produced from averaging the nearest and second-nearest depth layers from the light's point of view. This technique can actually be implemented as a two-pass technique on GeForce3 hardware using the depth peeling technique described in [2] and with a slight twist. In the second pass, the shadow map is used to peel away the nearest surfaces, but all depths are computed as the average of the

fragment's original depth and the nearest depth at that fragment's (x,y). The nearest surface (that is not peeled away), is then the average of the first and second nearest fragments!

Wang and Molnar introduce another technique to reduce the need for polygon offset [7]. Their technique works by rendering only back-faces into the shadow map, relying on the observation that back-face z-values and front-face z-values are likely far enough apart in z to not falsely self-shadow. This only helps front-faces, of course, but back-faces (with respect to the light) are, by definition, not in light, which helps hide artifacts. Note that this algorithm only works for closed polygonal objects.

Computing Transformations for Shadow Mapping

Computing the transformations required for shadow mapping can be somewhat tricky. This section provides details on the various transformations that need to be applied during the two render passes. While this section provides details for the OpenGL case, the transformations required for Direct3D are very similar with the main exception being that the texture coordinate generation is done directly via a matrix instead of the *texgen* planes. Also, keep in mind that the scale-bias matrix in Direct3D requires an additional offset to account for the discrepancy between pixel and texel coordinates as mentioned earlier, and that *eye linear* texgen is called D3DTSS_TCI_CAMERASPACEPOSITION.

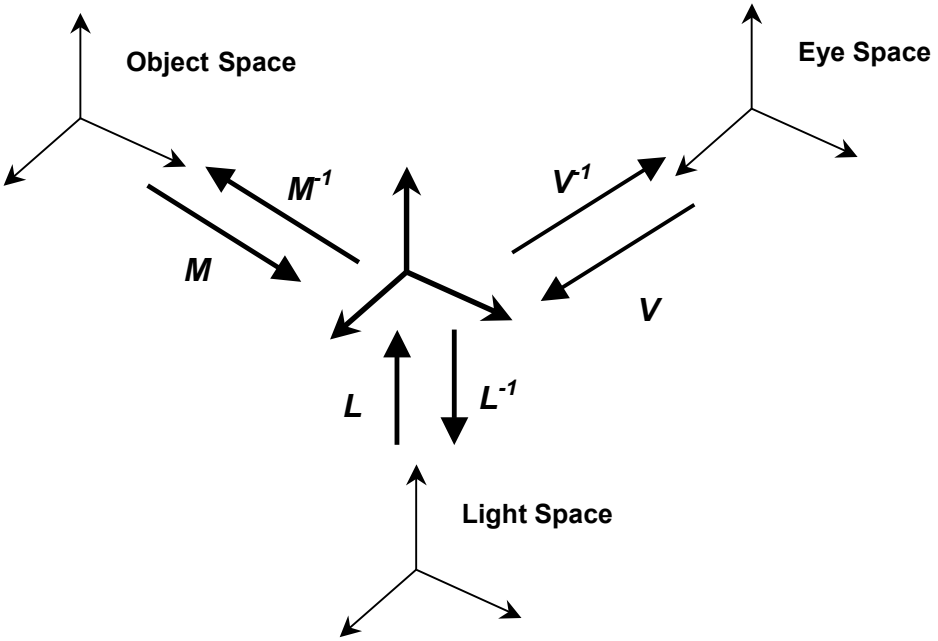


Figure 7: Schematic view of the basic transformations involved in shadow mapping.

Figure 7 shows the three primary transformations (and inverses) used in shadow mapping. Note that we use the convention of using the *forward* transforms as going *to* world coordinates. The standard 'modelview' matrix using the above notation will therefore be: $\boldsymbol{V^{1}M}$. In addition to the above transformations, we also have to account for the projections involved in the two passes – these could be different depending on the frusta for the light and eye. The projection transformation will also be applied during the texture coordinate generation phase which is depicted in Figure 8 for OpenGL. As shown

in the figure, two transformations are applied to the eye coordinates – the *texgen* planes, and the *texture matrix*. For *eye linear* texgen planes, OpenGL will automatically multiply the eye coordinates with the inverse of the modelview matrix *in effect when the planes are specified*. (See Appendix A for a more detailed explanation of the texgen planes in the eye linear case.)

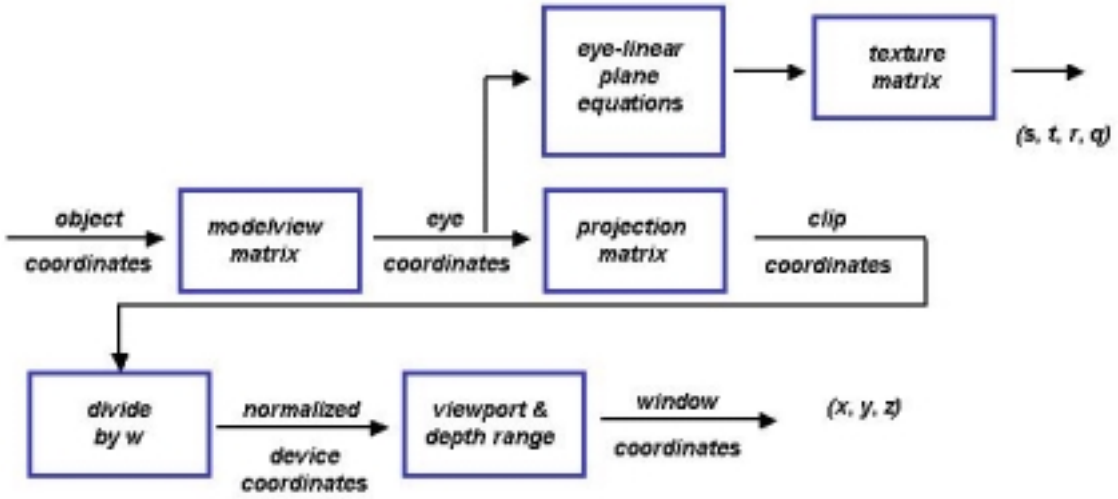


Figure 8: OpenGL Transformation Pipeline

The resulting texture coordinates are therefore computed as:

$$[x_e,y_e,z_e,w_e]^T = (\text{modelview})\ [x_o,y_o,z_o,w_o]^T$$

$$E_e = E_{po}(\text{modelview}_{po})^{-1}$$

$$[s,t,r,q]^T = T\ E_e\ [x_e,y_e,z_e,w_e]^T$$

Equation 1

Here the subscript '**o**' denotes object space coordinates, and the subscript '**e**' refers to eye space coordinates, $\textbf{modelview}_{po}$ is the modelview matrix in effect when the eye linear texgen plane equations are specified, E_{po} is the matrix composed of the eye linear plane equations *as specified to OpenGL* (i.e. in their own object space), E_e is the matrix composed of the transformed plane equations (these are the plane equations that are

actually stored by OpenGL), **T** is the texture matrix, and **modelview** is the modelview matrix when rendering the scene geometry.

Setting Up the Transformations

We want to set the transformations in Equation 1 to compute texture coordinates **(s,t,r,q)** such that **(s/q,t/q)** will be the fragment's location within the depth texture, and **r/q** will be the window-space z of the fragment relative to the light's frustum. In other words, we want to compute:

$$[s,t,r,q]^T = S\ P_{light}\ L^{-1}\ M\ [x_o,y_o,z_o,w_o]^T$$

Equation 2

Here, **S** is the scale-bias matrix, given by:

$$\begin{bmatrix} ½ & 0 & 0 & ½ \\ 0 & ½ & 0 & ½ \\ 0 & 0 & ½ & ½ \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$\mathbf{P_{light}}$ is the projection matrix for the light frustum. The "texgen matrix" ($\mathbf{E_e}$), however, is applied to *eye* coordinates $[x_e,y_e,z_e,w_e]^T$ but we want to generate the coordinates in *light* space, since that is where the depth map computation takes place. So we need to take $[x_e,y_e,z_e,w_e]^T$ back into world space by applying the transform **V**. That is, we want to compute $[s,t,r,q]^T$ as:

$$[s,t,r,q]^T = S\ P_{light}\ L^{-1}\ V\ [x_e,y_e,z_e,w_e]^T$$

Equation 3

Note that the right hand side of Equation 3 reduces to $S\ P_{light}\ L^{-1}\ M\ [x_o,y_o,z_o,w_o]$, precisely what we want. A straightforward way to compute Equation 3 is to set $\mathbf{modelview_{po}}$ to *identity* and set:

$$T\ E_{po} = S\ P_{light}\ L^{-1}\ V$$

Equation 4

The first observation is that we have two matrices **T** (the texture matrix) and $\mathbf{E_{po}}$ (the eye linear texgen planes specified to OpenGL) so we can compute Equation 4 in several

ways. Since we are going to have to set the eye linear planes in any case, the less expensive thing to do is to not set the texture matrix at all, and use the texgen matrix **G** for the entire computation[†], i.e., set

$$\mathbf{E}_{po} = \mathbf{S}\,\mathbf{P}_{light}\,\mathbf{L}^{-1}\,\mathbf{V}$$

Equation 5

This assumes that the modelview matrix, $\mathbf{modelview}_{po}$, was identity at the time the texgen planes are set. Another improvement is to make use of the fact that OpenGL automatically multiplies $[x_e,y_e,z_e,w_e]^T$ with $(\mathbf{modelview}_{po})^{-1}$ for eye linear texgen. The sole purpose of using **V** in Equation 5 is to eliminate $\mathbf{V}^{-1}$. If we set $\mathbf{modelview}_{po} = \mathbf{V}^{-1}$, then OpenGL will do the elimination for us and we can avoid having to compute **V**, the inverse of the view matrix. The steps can be summarized as follows:

First Pass (Depth Map Generation)

- Render from light's point of view. Set projection matrix to $\mathbf{P}_{light}$. Set the view portion of the modelview matrix to $\mathbf{L}^{-1}$.
- Render scene (with appropriate modeling transform(s) **M**).

Second Pass (Depth Map Comparison)

- Render from eye's point of view. Set projection matrix to be $\mathbf{P}_{eye}$. Set the view portion of the modelview matrix to be $\mathbf{V}^{-1}$.
- Set texgen to be EYE_LINEAR. Specify texgen planes as $\mathbf{E}_{po} = \mathbf{S}\,\mathbf{P}_{light}\,\mathbf{L}^{-1}$
- Render scene (with appropriate modeling transform(s) **M**)

Conclusions

Shadow mapping is an easy-to-use shadowing technique that makes 3D rendering just look better. It enjoys hardware acceleration on GeForce3 GPUs. There is example source code in the NVSDK (hw_shadowmaps_simple, hw_woo_shadowmaps) that demonstrate the technique, and the corresponding OpenGL extensions. Please direct questions or comments to cass@nvidia.com.

References

[1] Craig Duttweiler. Mapping Texels to Pixels in Direct3D. http://developer.nvidia.com/view.asp?IO=Mapping_texels_Pixels.

[†] Note that this technique of collapsing the texture and texgen matrices works in our case because we are setting all four planes, and using the same mode for all four planes. In general, each coordinate can have a different mode (eye linear, object linear, sphere map…) and the technique may not be applicable.

[2] Cass Everitt. Interactive Order-Independent Transparency. Whitepaper: http://developer.nvidia.com/view.asp?IO=Interactive_Order_Transparency.

[3] Mark Kilgard. Shadow Mapping with Today's Hardware. Technical presentation: http://developer.nvidia.com/view.asp?IO=cedec_shadowmap.

[4] Mark Segal and Kurt Akeley. The OpenGL Graphics System: A Specification (Version 1.2.1). www.opengl.org

[5] Mark Segal, et al. Fast shadows and lighting effects using texture mapping. In *Proceedings of SIGGRAPH '92*, pages 249-252, 1992.

[6] OpenGL Extension Registry. http://oss.sgi.com/projects/ogl-sample/registry/.

[7] Yulan Wang and Steven Molnar. Second-Depth Shadow Mapping. UNC-CS Technical Report TR94-019, 1994.

[8] Lance Williams. Casting curved shadows on curved surfaces. In *Proceedings of SIGGRAPH '78*, pages 270-274, 1978.

[9] Andrew Woo, P. Poulin, and A. Fournier. "A Survey of Shadow Algorithms," IEEE Computer Graphics and Applications: vol 10(6), pages 13-32, 1990.

Appendix A: Another Way to Think about EYE_LINEAR planes in OpenGL

An unfortunate thing about EYE_LINEAR texgen in OpenGL is that the name implies that the plane equations are specified in eye space, when they are, in fact, specified in their own object space. There are two ways one can think about the planes specified in EYE_LINEAR texgen. As mentioned earlier, OpenGL will automatically multiply the planes specified with $(\mathbf{modelview}_{\mathbf{po}})^{-1}$, i.e. the inverse of the modelview matrix in effect when the planes are specified. From Equation 1 we see that the net effect is to map the vertex position in eye coordinates $[x_e,y_e,z_e,w_e]^T$ back to the 'object space' defined by $(\mathbf{modelview}_{\mathbf{po}})^{-1}$. The transformed coordinates are then evaluated at each plane in this object space to get the texture coordinates. An alternate way to think about the texgen planes is to consider the matrix $\mathbf{E}_e = \mathbf{E}_{po}(\mathbf{modelview})^{-1}$, which defines a map whose domain is *eye* space, with the planes $\mathbf{E}_{po}$ being specified in object space. $\mathbf{E}_e$ therefore defines the *transformed* planes in eye space. In either case, the planes are being *specified* in the 'object space' defined by $(\mathbf{modelview}_{\mathbf{po}})^{-1}$ and *not* in eye space.

In the shadow mapping case described earlier, the modelview matrix is set to $\mathbf{V}^{-1}$ when the texgen plane equations are specified. This is the same thing as saying that we are specifying the plane equations in *world* space. If the modelview matrix were set to identity, then we would be specifying the equations in *eye* space. The same is true if we were specifying vertex positions.

We could set the modelview matrix to $\mathbf{V}^{-1}\mathbf{L}$, and specify the plane equations in *light* space. This might be handy, because we would only need to update our plane equations if the light's projection ($\mathbf{P}_{\mathbf{light}}$) changed. We could even put the *whole* transformation into the modelview matrix as $\mathbf{V}^{-1}\mathbf{L}\mathbf{P}_{\mathbf{light}}^{-1}\mathbf{S}^{-1}$. In this case, the texgen planes are *always* just specified as identity ($\mathbf{E}_{\mathbf{po}} = \mathbf{I}$)!

Light Mapping - Theory and Implementation

by Keshav Channa (21 July 2003)

Introduction

Ever since the days of Quake, developers have extensively used light mapping as the closest they can get to realistic lighting. (Nowadays true real-time Per-Pixel-Lighting is slowly replacing light maps).

In this article I am covering a simple but quick light mapper that we developed in 2001 for use in our internal projects at Dhruva Interactive. The method is not as precise as radiosity based techniques, but nevertheless produces results that are effective and usable.

This article will be useful for people who havenï¿½t got light maps into their engine / level yet, and are looking for relative resources.

Overview

This document explains / demonstrates the process of creating light maps for use in games or any graphics application.

Objectives

The goal of this document is to explain the process of creating light maps. It describes how light map lumels are calculated and how the final pixel color is determined.

However, this document does not cover any of the latest ï¿½per-pixelï¿½ techniques that are possible with the new generation graphics chipsets.

Assumptions

It is assumed that the reader has in depth knowledge of 3D game programming and the essentials of 3D graphics, especially lighting, materials, 3D geometry, polygons, plane, textures, texture co-ordinates, etc.

Also, this document explains only the process of light map evaluation or generation ï¿½ it does not explain how light map texture co-ordinates are calculated. If you do not have light map texture co-ordinates in your mesh yet, then, Iï¿½ve got a simple method to test the light map generation.

Click here to read more on this.

If you are really eager to know the results of a light map based lighting technique before you get on with this article, then click here to go to the demo section, where you can download an interactive demo.

Lighting Basics

You've seen games which look pretty close to real life ambience (I mean, only look wise). The reason for that, is the use of lights. If the game was not "lit", then, it would look less than ordinary. It is the lighting, which makes the player, look at the game again and again. Take a

look at what difference, lighting makes:

World with light-map lighting. The white rhombus type object (on the right hand side) represents a point light source.

World without any lighting.

The results are impressive, aren't they?

Now that youï¿½ve seen the results, letï¿½s take a look at the different types lighting in practice (as far as the "static world" is concerned).

1. Vertex lighting:

For every vertex, a color value is calculated, based on how lights affect it.
The color values will be interpolated across the triangle.
Triangles / polygons cannot be very large, other wise which visual glitches will be seen.
Polygons have to be tessellated to a decent level for the output to be good.
If vertex count goes up, then, calculations also take longer ï¿½ since it is per vertex based.
Does not incur a load on texture memory (as required for light maps).
All the calculations are done in real time ï¿½ hence real-time lighting is very much possible.
Shadow(s) are not correct.
Can achieve amazing lighting effects.

2. Per-pixel lighting (Real time):

This document does not cover the per-pixel lighting methods that are in practice today, i.e., the effects that are achievable on current generation of graphics cards, like NVIDIA GEForce 3/4, ATI Radeon 8500/9700 and later.

For every pixel that is going to be drawn, calculate the color value based on how lights affect it.
Does incur a huge load on the engine.
It is not practical for a real-time game to use it.
Accurate shadows are possible (another overhead with collision detection is a certain possibility).
Can achieve the most realistic lighting possible ï¿½ but is too slow to use in real time games.

3. Per-pixel lighting (light map):

Realistic lighting can be achieved.
Dynamic lighting needs a lot more work.
Can combine with vertex lighting to achieve real-time dynamic lighting.
Every single bit of expensive lighting calculation is done during pre-process time.
Hence, no overhead during runtime.
At run-time, all calculations (color arithmetic) are done by hardware. Hence it is very fast.
Visual quality of the lighting is directly dependant on the size of the light map texture(s).
The closest we can get to per-pixel lighting, with such less overhead.
For every triangle, a diffuse texture map is applied first and then, a light map is usually modulated (multiplied) with it.

Per-pixel Lighting Using Lightmaps

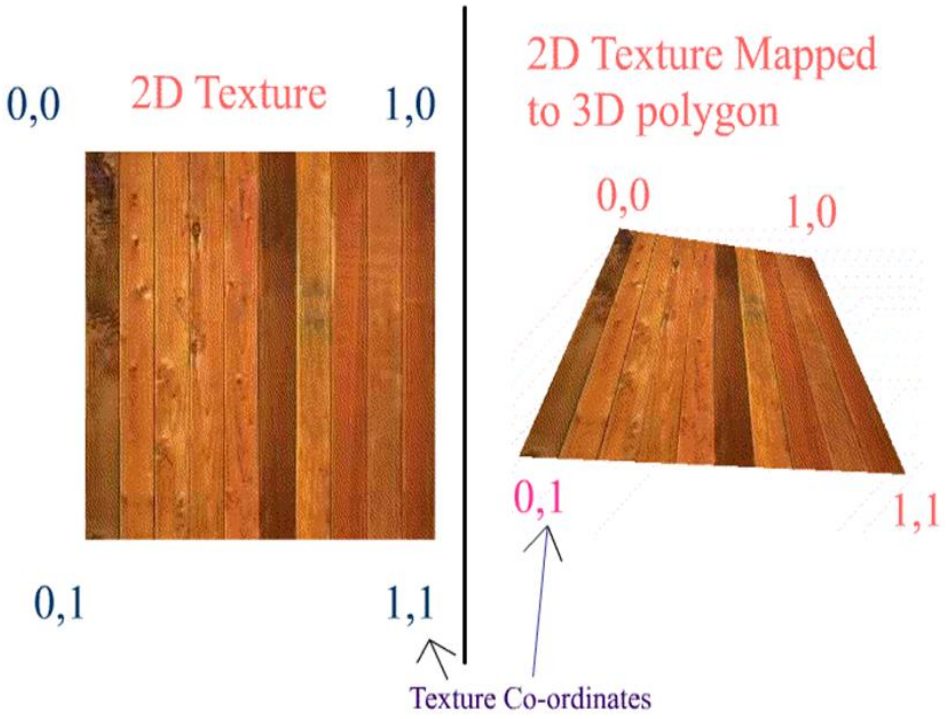
We will be discussing light-map based lighting in the remainder of the document. A light-map is nothing but another texture map. The only difference between a light map and a diffuse map is that, a diffuse map holds plain color values, whereas, a light map holds color values as a result of light(s) affecting the polygons on which this light map was applied. Everything else is the same.

Light maps textures are mapped to a triangle/polygon using a unique set of texture co-ordinates known as light map texture co-ordinates.

Resource for light map is loaded the same way as the diffuse textures.

Each pixel in a diffuse map texture is usually referred to as ï¿½texelï¿½, whereas each pixel in a light map texture is referred to as ï¿½lumelï¿½. We all want to use fancy words ï¿½ donï¿½t we???

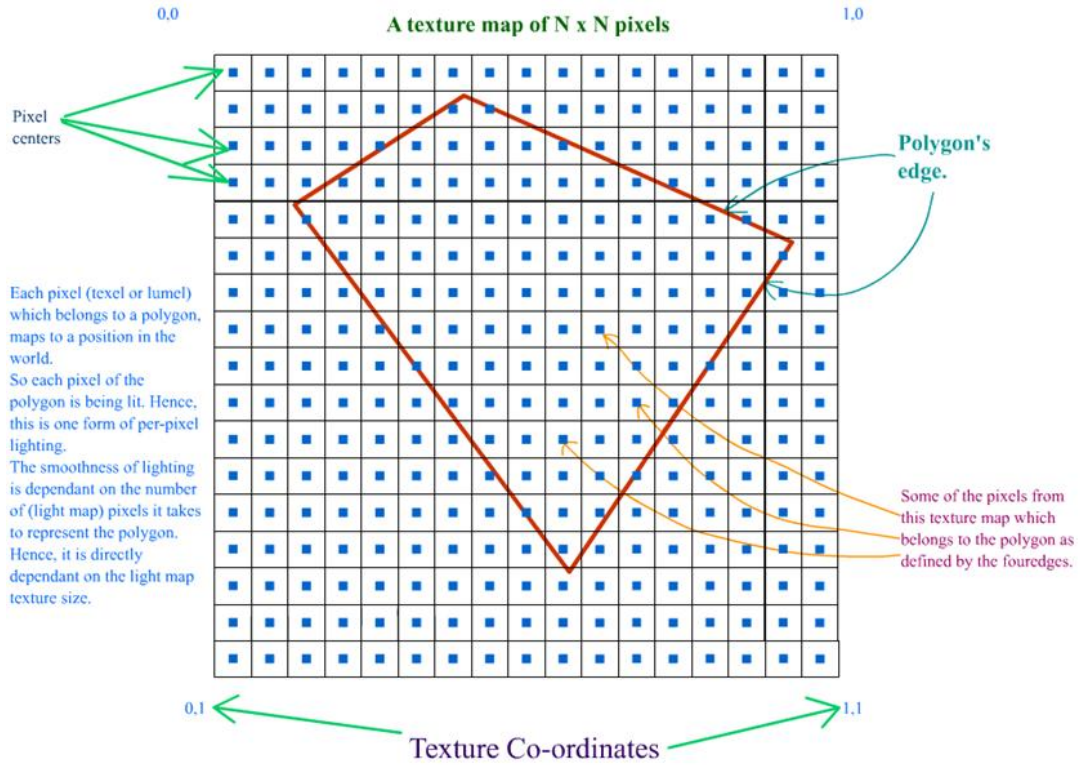
Before delving into the process of light map calculation, lets take a look at how a 2D texture is mapped on to a 3D triangle.



On the left hand side of the above diagram, is a 2D texture (as you would see in any of the image editing tools) of (N x N) dimension. The normalized dimension is also shown.

On the right hand side is a 3D polygon, as you would see in the game, except that the background is missing. The texture co-ordinates are mentioned for each vertex. As you can see, the texture is mapped to the polygon with a mapping ratio of 1:1. i.e. the whole texture is mapped to the polygon at a ratio of 1:1.

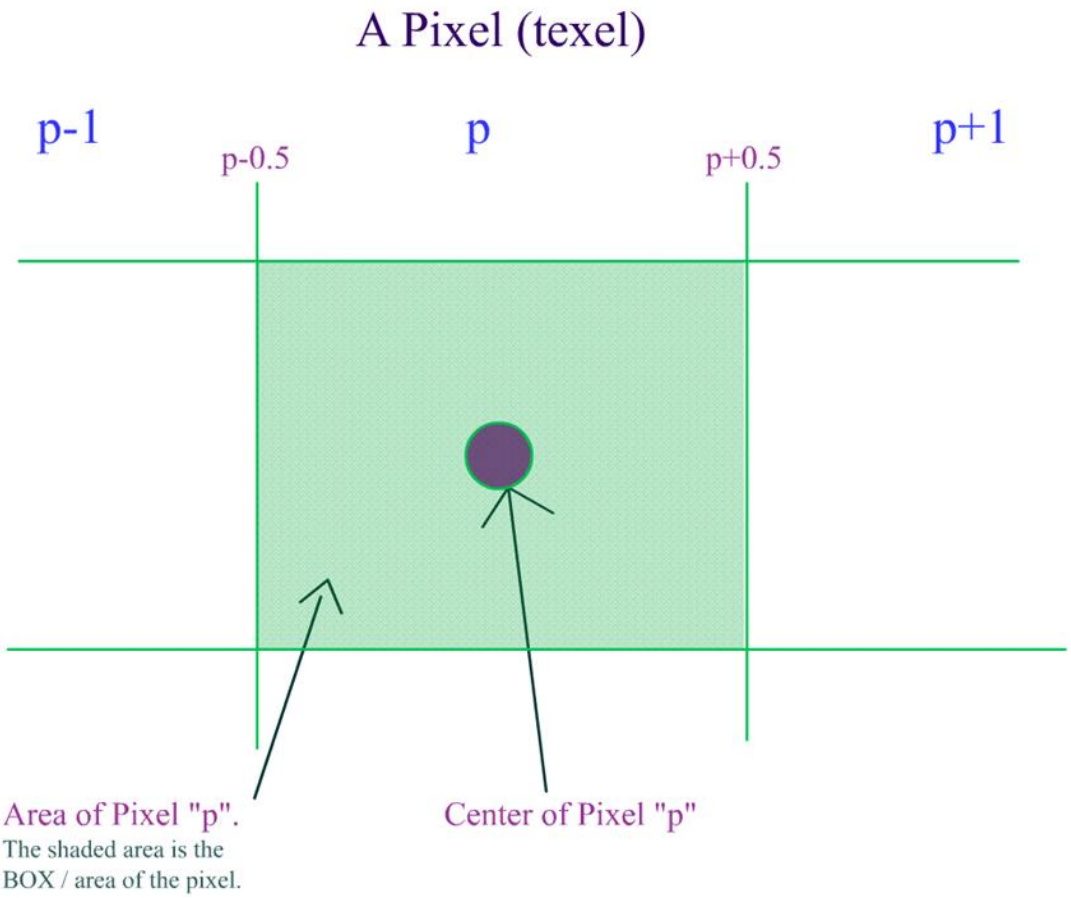
Now consider the diagram below. The diagram shows a polygon being mapped or pasted on a 2D texture. This polygon however uses only a part of the texture. So, the mapping ratio is not 1:1. We can observe that the polygon covers a only a part of the area of the complete texture. i.e. this polygon has some pixels from the texture map belonging to it. Hence, more the pixels that this polygon has, the more nicer that this polygon will look. (This is in 3D viewing, considering the camera is neither very close to the polygon nor far from the polygon. We shall ignore MIPMAPS for this discussion).



ï¿½s look at the above diagram closely taking light maps into account. A diffuse map, may may not be shared by polygons. i.e. a pixel from a diffuse map can be belong to "n" number polygons. **But, for a light map, a pixel belongs to one and only one polygon.**

ch pixel in the light map has a corresponding position in the world, with respect to the ygon that it belongs to. You have to understand this concept very well. Since every vertex of riangle has a position in the world, every light map pixel that this triangle will hold will have osition in the world ï¿½ which will vary uniformly across the length of each edge.

o, there is a concept of pixel center. Whenever we refer to a pixel, it means the pixel center. ixel is not a point, but it is a box. Please refer to the diagram below.

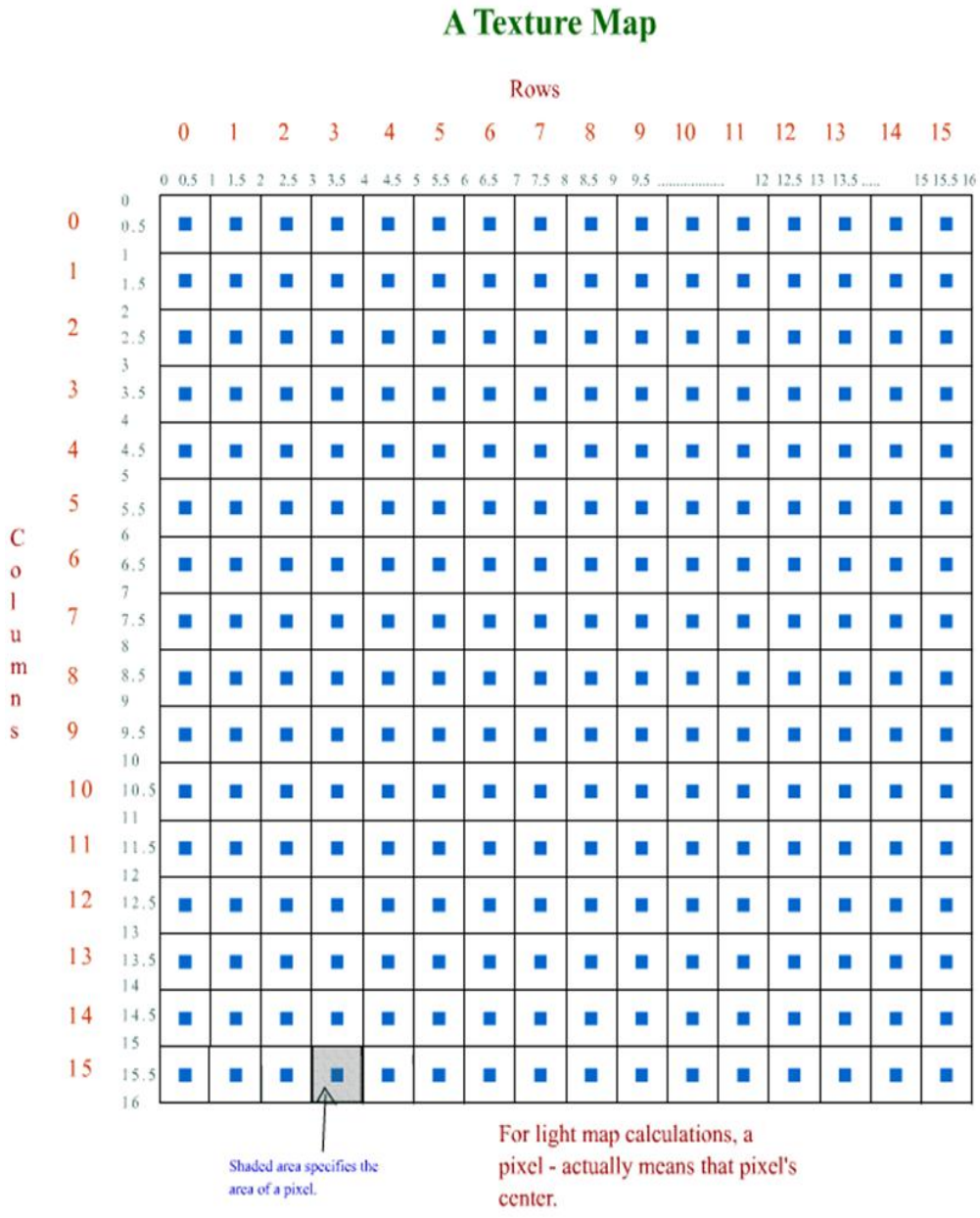


ased on the above criteria, whenever we want a UV co-ordinate for any pixel, it is calculated
is way:

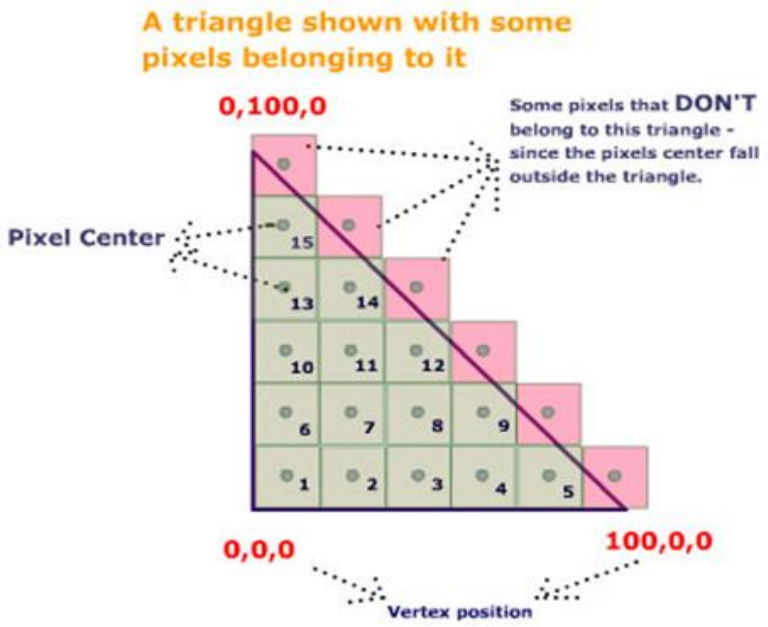
```
x = (w+0.5) / Width
y = (h+0.5) / Height
```

the above equations,
and h = offsets for the current pixel that we are calculating UV co-ordinates for.
idth = width of the light map texture.
eight = height of the light map texture.

his is what is mentioned in the diagram below.



Now, Iï¿½m trying illustrate what I have said above, in the form of a diagram. First, take a look at the diagram:



In the above figure, I am hypothetically describing the relationship between the triangle and the (light map) pixels.

In the above figure,

the triangle is defined by the 3 thick black edges / lines.
the three vertices of the triangle are (0,0,0) , (0,100,0) and (100,0,0)
Since the Z co-ordinate for all the three vertices is same, we can safely ignore the Z-component for our calculations.
Each box inside (and slightly outside) the triangle identify a unique (light map) pixel.
Remember that a pixel is a box (has area) and not a point.
Green box means, that the pixel is well with in the triangle and that this light map pixel belongs to this triangle.
Pink box means that the pixel centers fall outside this triangle and hence these pixels do NOT belong to this triangle.
Every pixel that belongs to this triangle has been numbered in certain order.
This triangle contains 15 pixels.
Our exercise now is to determine the approximate theoretical world position for each pixel just by observing the triangle and the pixels.

Remember, what we are doing now is just based on eye measurements. It is in no way accurate.

This exercise is to just make you understand the relationship between vertex, lightmap texture-cordinates and light map pixels. This exercise should give you an idea of what the "world-position" for a pixel means.

Look at the bottom most line, there are 5 pixels and the width of the edge is 100 units on X axis.

That means 20 units for each pixel.
Also, the edgeï¿½s position varies only on X axis. So, Y and Z values remain constant.
Hence the first pixel would have the X value of 20 (Not accurate by any means).
First pixelï¿½s position would be (20.0, 0.0, 0.0)

The second would have a position of (40.0, 0.0, 0.0), third would have (60.0, 0.0, 0.0), fourth would have (80.0, 0.0, 0.0) and the fifth would have (100.0, 0.0, 0.0)

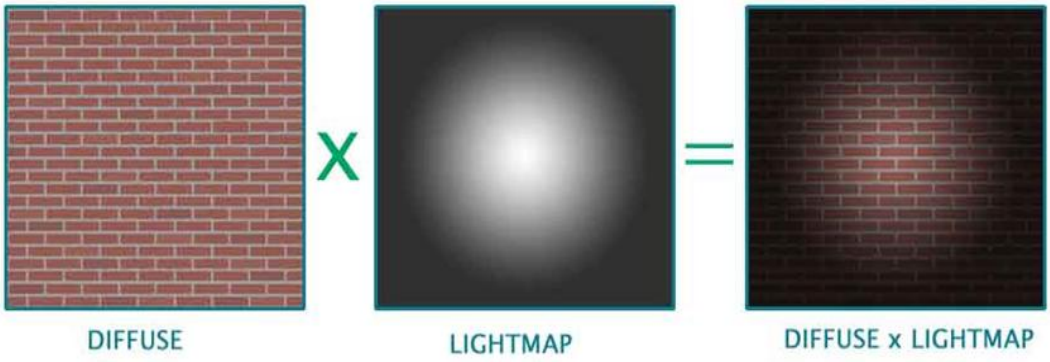
Similarly, try to arrive at the positions for the other pixels.

I'm reminding you again that the above results are NOT correct. That exercise was to make you understand what "world-position" for a pixel means.

Also, from the above diagram, you can decipher that ï¿½ the more (light map) pixels a triangle has, the lesser will be the world-position shift from pixel to pixel ï¿½ and hence results in a smoother output. Try to figure out how?

If you still havenï¿½t understood the concept of pixel center, and world-positions for a pixel, then, please go through this document again from the start. Please do not continue if you are not clear.

Hereï¿½s an image which shows the result of using a light map.



A Simple Lightmap Based Lighting Technique

Now letï¿½s get on with the actual process of light mapping. The complete process of calculating light maps is split into three parts. They are:

- 1. Calculating / retrieving light map texture co-ordinates.
  2. Calculating the world position and normal for every pixel in every light map.
- 3. Calculating the final color for every pixel.

a. Calculating / retrieving light map texture co-ordinates:

This is the very first and basic process. It involves assigning each polygon to a specific area of the light map. This topic in itself is a candidate for a lengthy article. Hence, I'm going to skip this topic and jump to the next one.

However, if you want links to articles that explain this, then, I've provided some in the links section, at the very end of this article.

Also, this is one of the most important processes, since it determines the efficiency of using texture space. This can either be done in an automated process or can be done manually in editing tools such as 3DS Maxï¿½ or Mayaï¿½.

However, if you want a quick way to generate a world to test light maps, then you can do this:

Create an empty cube, a pretty large one. (empty, hence collision detection is not required)

Assign diffuse and light map texture co-ordinates manually.

Even better, you could use one diffuse for all the faces of the cube and use six different lightmaps for each face(polygon) of the cube.

This way, you can map the light map to the polygons of the cube with a ratio of 1:1.

Create one or two lights at the extreme top ends of the cube.

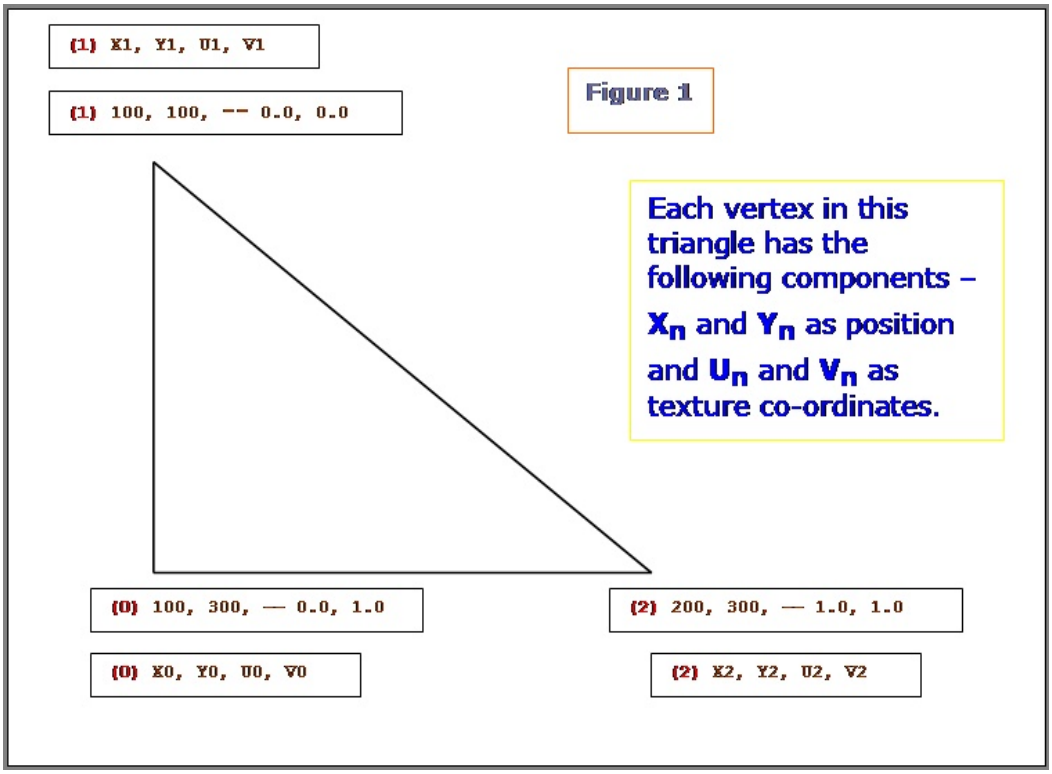
Use this setup to test your light map generation.

In the next level, to test shadow generation, you can add a box at the bottom center of the cube, and add collision detection routines.

b. Calculating the world position and normal for every pixel in every light map:

This is the pre-processing stage of the light map calculation. As you know, each pixel in the light map will map to a position in the world. This is exactly what we need to calculate. As long as the world geometry and the light map texture sizes doesnï¿½t change, this data is static, i.e. it can be calculated once and reused.

This is how it is done. Now consider a triangle like this. Why we have only 2D components for vertex position in the below triangle will be explained in later paragraphs.



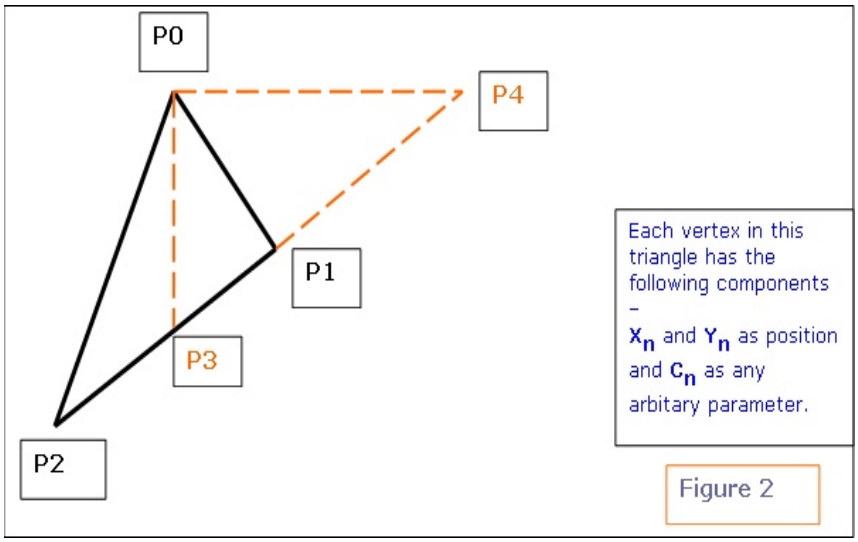
Letï¿½s see what are all the know factors here:

a. We know the end points of the triangle (2D).
b. We know the (light map) texture co-ordinates for all the 3 vertices.

What do we need to calculate here: Given a texture co-ordinate value (on or within the 2D triangle), retrieve the 2D position on or within the 2D triangle. We have to do it for every pixel of the light map that this polygon owns. (Remember, a pixel from a light map can belong to one and only one

triangle / polygon.) Let's see how we can achieve this.

NOTE: In the following few paragraphs, Ïï¿½m using certain equations, illustrations and quotes from Chris Heckerï¿½s article on Perspective Texture Mapping. Please refer to the above said article for more information.



Consider triangle P0, P1, P2 in the above diagram. Each vertex has a screen space (2D SPACE) associated with it i.e. (X, Y). But in addition, there is an arbitrary parameter, C, which is, color for Gouraud Shading, or 1/Z, u/z or v/z for perspective texture mapping. Hence ï¿½Cï¿½ is any parameter we can linearly interpolate over the surface of the two-dimensional triangle. For the derivation, two points P3 and P4 are constructed as shown in the above diagram.

Therefore,

```
x1 - x2       x4 ï¿½ x2
--------   =  --------
y1 ï¿½ y2      y4 ï¿½ y2

and

c1 ï¿½ c2         c4 ï¿½ c2
--------    =   --------
y1 ï¿½ y2         y4 ï¿½ y2
```

We know that y4 = y0 and x3 = x0.

Click here to see the derivation as to how we arrived at equation 1 and 2 mentioned below. For the triangle, we get the following two equations:

$$\frac{dc}{dx} = \frac{c4 - c0}{x4 - x0} = \frac{(c1-c2)(y0-y2) - (c0-c2)(y1-y2)}{(x1-x2)(y0-y2) - (x0-x2)(y1-y2)} \quad \text{Equation 1}$$

$$\frac{dc}{dy} = \frac{c3 - c0}{y3 - y0} = \frac{(c1-c2)(x0-x2) - (c0-c2)(x1-x2)}{(x0-x2)(y1-y2) - (x1-x2)(y0-y2)} \quad \text{Equation 2}$$

The above two equations tell how much of variable ï¿½Cï¿½ varies w.r.t X and Y. i.e. given a position (x, y) we can calculate ï¿½Cï¿½ for that position. For our lightmap solution, we need just the opposite. We know the texture co-ordinates (i.e. ï¿½Cï¿½) and we need to retrieve the position. I will be using the formulas as mentioned below. These are directly inherited from the above two equations. (Please refer to figure 1 for the variable names)

```
denominator = (v0-v2)(u1-u2) ï¿½ (v1-v2)(u0-u2)

dp      dx       (x1-x2)(v0-v2) ï¿½ (x0-x2)(v1-v2)
--- =   ---   =  ---------------------------------------
du      du              denominator

dp      dx       (x1-x2)(u0-u2) ï¿½ (x0-x2)(u1-u2)
--- =   ---   =  ---------------------------------------
dv      dv              -denominator

dq      dy       (y1-y2)(v0-v2) ï¿½ (y0-y2)(v1-v2)
--- =   ---   =  ---------------------------------------
du      du              denominator

dq      dy       (y1-y2)(u0-u2) ï¿½ (y0-y2)(u1-u2)
--- =   ---   =  ---------------------------------------
dv      dv              -denominator
```

Now get uv position relative to the first vertexï¿½s light map texture co-ordinate.

```
duv.x = uv->x ï¿½ u0
duv.y = uv->y ï¿½ v0
```

uv is the pointer to the texture co-ordinate for which the ï¿½world-positionï¿½ has to be computed. U0 and V0 are the light map texture co-ordinates for the first vertex.

Let **pos** be the pointer to final position.

```
Equation 3
```

```
pos->x = (x0) + (dpdu * duv.x) + (dpdv * duv.y)
pos->y = (y0) + (dqdu * duv.x) + (dqdv * duv.y)
```

Now we have the 2D position corresponding to the triangle and the UV coordinate.

Let's look at the figure 1 as an example, and try to retrieve the position for a given UV co-ordinate:

Let the UV co-ordinate for which the Position has to be calculated be {0.5, 1.0 } We definitely know that co-ordinates {0.5, 1.0 } fall well within (or on) the triangle.

We get the following values:

```
dxdu = dpdu = 100
dxdv = dpdv = 0
dydu = dqdu = 0
dydv = dqdv = 200

duv.x = (0.5 ï¿½ 0) = 0.5
duv.y = (1.0 ï¿½ 0) = 1.0
```

Hence the position is:

```
Pos->x = 150
Pos->y = 300
```

You can change the winding order and try ï¿½ you'll get the same result. Only the dxdu, dxdv, dydu, dydv, and duv values will change.

Now, we've retrieved the 2D position w.r.t a UV co-ordinate. But finally, we need a 3D position to do any 3D calculations. How do we do this? Here's the plane equation to the rescue. We know that a plane is represented by the equation: **Ax+By+Cz+D = 0**. Every polygon / triangle has a plane equation associated with it. Also, we can project any triangle/polygon along two of it's major axis. i.e. we are projecting a 3D triangle to 2D, by ignoring one of itï¿½s axis. This is required, because a texture is 2D whereas a triangle is in 3D space.

Which axis ï¿½ X, Y or Z to ignore? How we choose the axis to ignore is, given the plane normal, choose the two axis that are most aligned to the plane normal. If a, b, c (analogous to x, y, z axis) represent the plane normal, then, find out the maximum of the absolute of these three values, and ignore that axis.

Ex. If plane normal is (0, 1, 0), then, we would choose XZ axis and ignore the Y component. If plane normal is (0, -1, 0), then also, we would choose XZ axis.

Now we'll refer back to figure 1.

If the plane normal for the given triangle in figure 1 was (0, 1, 0), then the (Xn, Yn) values specified

the derivation consistent.

Remember, the triangle is still in 3D, but weï¿½re converting it to 2D by ignoring one component.

Look at equation 3. Now, we've the world position of the lumel in 2D co-ordinates. We've to convert it to 3D. So, use the plane equation. Depending on which axis weï¿½ve projected the triangle, we have to use the appropriate equation.

For example: The plane normal was (0.123, 0.824, 0.34).

According to what Iï¿½ve mentioned above, we ignore the Y ï¿½ component. Hence, when we get the 2D position of the lumel, it will be the X and Z components. We need to calculate the Y component.

We have Ax+By+Cz+D = 0.

By = -(Ax+Cz+D)

y = -(Ax+Cz+D) / B.

Thus we have the Y component. Similarly, we can calculate the missing component for other projections also.

First, let's look at the Lumel structure.

```
struct Lumel
{
    D3DXVECTOR3     Position ;       // lumel position in the world.
    D3DXVECTOR3     Normal ;         // lumel normal (used for
                                     // calculating N.L)
    DWORD           Color ;          // final color.
    BYTE            r, g, b, a;      // the red, green, blue and alpha
                                     // component.
    int             LegalPosition ;  // is this lumel legal.
    DWORD           polygonIndex ;   // index of the polygon that it
                                     // belongs to.
} ;
```

This structure is used for every pixel in every light map. Ex. If the dimensions of a light map are 64x64, then, the memory required to hold the lumel info Would be:

size of each Lumel structure = 40 bytes.

Total size = (64 * 64 * 40) bytes = 163840 Bytes = 160 Kbytes.

This is just a test case. You CAN reduce the memory foot print.

"LegalPosition" member of the Lumel structure, will hold information whether the particular pixel belongs to any polygon or not.

The structure for a vertex displaying a light map would look something like this.

```
struct LMVertex
{
    D3DXVECTOR3  Position ;        // vertex position.
    D3DXVECTOR3  Normal ;          // vertex normal
    DWORD        Color ;           // vertex color.
```

```
    D3DXVECTOR2  t0 ;              // diffuse texture co-ordinates.
    D3DXVECTOR2  t1 ;              // light map texture co-ordinates.
} ;
```

The structure for a polygon displaying a light map would look something like this.

```
struct LMPolygon
{
    LMVertex      *vertices ;           // array of vertices.
    WORD          *indices ;            // array of indices.
    DWORD         VertexCount,          // No. of vertices in the array
                  FaceCount ;           // No. of faces to draw in this
                                        // polygon.

    DWORD         DiffuseTextureIndex ; // the index in to the diffuse
                                        // texture array

    DWORD         LMTextureIndex ;      // the index in to the light-map
                                        // texture array
} ;
```

Hereï¿½s the pseudocode for a function called BuildLumelInfo() that actually fills in the world position for every pixel in the light map:

```
BuildLumelInfo()
{
    // this function has to be called for each light map texture
    for(0 to lightmap height)
    {
        for(0 to lightmap width)
        {
            w = current width during the iteration (for loop)
            h = current height during the iteration (for loop)

            U = (w+0.5) / width
            V = (h+0.5) / height

            UV.x = U
            UV.y = V

            if (LumelGetWorldPosition(/*UV, this light map texture*/)) SUCCEEDED
            then
                // Mark this lumel as LEGAL.
            else
            {
                // Mark this lumel as illegal ï¿½ in the sense that no triangle
uses this pixel / lumel.
            }
        }

    }
}


LumelGetWorldPosition( UV, light map texture )
{
    for( number of polygons sharing this light map texture )
    {
```

```
        // do the "Bounding Box" lightmap texture co-ordinate rejection // test.
        if( /*UV co-ordinates of the light map do not fall inside the
            polygonï¿½s MAXIMUM and MINIMUM UV co-ordinates*/ )
        then
            //try next polygon ;

        // code for the above explanation

        if(uv->x < poly->minUV.x)   continue ;
        if(uv->y < poly->minUV.y)   continue ;
        if(uv->x > poly->maxUV.x)   continue ;
        if(uv->y > poly->maxUV.y)   continue ;

        for( /* number of faces in this polygon */ )
        {
            /*Get the three vertices that make up this face.
            Check if light map UV co-ordinates actually fall inside the
polygonï¿½s
            UV co-ordinates. This routine is similar to routines like
ï¿½PointInPolygonï¿½
            or ï¿½PointInTriangleï¿½.

            If YES, then call GetWorldPos to get the actual world position in 3D
            for this given light map UV co-ordinate.

            If NO, then this is not a legal pixel, i.e. this pixel does not
            belong to THIS polygon.*/
        }
    }
}


GetWorldPos(UV uv)
{
        // get uv position relative to uv0
        duv.x = uv->x - uv0->x ;
        duv.y = uv->y - uv0->y ;

        // retrieve the components of the two major axis.
        // i.e. here we are converting from 3D triangle to 2D.

        switch(PlaneProjection)
        {
                case PLANE XZ :
                        // collect X and Z components
                break ;

                case PLANE XY :
                        // collect X and Y components
                break ;

                case PLANE YZ :
                        // collect Y and Z components
                break ;
        }

        // Calculate the gradients from the equations derived above.
        // See Equation 3 above.

        Now calculate gradients.
        i.e. dp/du, dp/dv, dq/du, dq/dv, etc.

        // In the following line, I have used a, b, instead of X, Y or Z.
        // This is because, depending on the polygonï¿½s plane we
        // choose either XY or YZ or XZ components. Hence, a and b map to
        // either XY or YZ or XZ components
        pos->a = (a0) + (dpdu * duv.x) + (dpdv * duv.y)
        pos->b = (b0) + (dqdu * duv.x) + (dqdv * duv.y)

        // get the world pos in 3D
        // calculate the remaining single co-ordinate based on the polygonï¿½s
        // plane.
```

```
        switch(PlaneProjection)
        {
                case PLANE XZ :
                        // We would have got X and Z as the 2D components.
                        // calculate the Y component.
                        y = -(Ax+Cz+D) / B.
                break ;

                case PLANE XY :
                        // We would have got X and Y as the 2D components.
                        // calculate the Z component.
                        z = -(Ax+By+D) / C.
                break ;

                case PLANE YZ :
                        // We would have got Y and Z as the 2D components.
                        // calculate the X component.
                        x = -(By+Cz+D) / A.
                break ;
        }
}
```

The function to build the lumel information is as follows: Remember, as long as the geometry, light map texture co-ordinates and light map texture sizes are constant, this function can be called only once and re-used again and again. This way, if you change a property for a light, then, you don't have to build the whole data base again. All you have to call is the function BuildLightMaps. BuildLumelInfoForAllLightmaps() should be called before BuildLightMaps().

```
BuildLumelInfoForAllLightmaps()
{
    // Do initialization here.

    for (number of light maps)
    {
        // If memory for Lumels not allocated, then,
        // Allocate memory to hold the lumel info for this particular
        // light map.

        BuildLumelInfo(this light map) ;     // Calculates the
                                             // world position for all the lumels.
    }
}
```

c. Calculating the final color for every pixel:

This is the last process involved in the calculation of light maps. Here we fill out the actual pixel values in every light map. I'll first give the pseudo code here which calculates the color for every pixel in the light map.

```
BuildThisLightMap()
{
    for(0 to lightmap height)
    {
        for(0 to lightmap width)
        {
            lumel = current lumel ;
```

```
            if(lumel is not legal)
            then
                   try next lumel.

            for( number of lights )
            {
                // cos theta = N.L

                dir = lightPosition - lumel->Position
                dot = D3DXVec3Dot(&lumel->Normal, &dir) ;

                // if light is facing away from the lumel, then ignore
                // the effect of this light on this lumel.

                if( dot < 0.0 )
                    try next light ;

                distance =  distance between lumel and light source.

                if(distance > light range)
                    try next light ;

                // Check Collision of ray from light source to lumel.

                if( collision occurred )
                then
                {
                    // lumel is in shadow.
                    continue ;
                }

                // GetColorFromLightSource.
                // Write color info to lumel.
            }
        }
    }
}
```

As you can see, the pseudo code explains itself pretty well. It's the basic light calculation. I wonï¿½t spend too much time there. Let's look at the procedure which starts the process of calculating colors for all the light maps.

```
BuildLightMaps()
{
      for (number of light maps)
      {
            BuildThisLightMap () ;                    // does all the lighting
calculations.
            BlurThisMap() ;                           // blurï¿½s the light map.
            FillAllIllegalPixelsForThisLightMap() ;   // fills all the illegal

                                                      // lumels with the closest
                                                      // color ï¿½ to prevent
bleeding when
                                                      // bi-linear filtering is
used.

            WriteThisLightMapToFile() ;               // finally write the lightmap
colors
                                                      // to file.
                                                      // I write it in a 24-bit BMP
format.

      }
```

```
}
```

Lets look closely at what the two functions *BlurThisMap* and *FillAllIllegalPixelsForThisLightMap* do.

No matter whatever we do, if youï¿½ve NOT turned on any filtering, then, "pixels" can be seen in the final rendered image. This is very unrealistic and can easily annoy the player / viewer. Hence, we try to smoothen out the "pixels".

You can use any filter you want for smoothing. Iï¿½m using a BOX filter in my code. This is exactly what my BlurThisMap does.

```
BlurThisMap()
{
    for(0 to height)
    {
        for(0 to width)
        {
            w = current width during the iteration (for loop)
            h = current height during the iteration (for loop)

            current pixel = GetCurrentPixel(w,h)

            // Get neighboring 8 pixels for current pixel, ignoring the
            // illegal pixels.

            sum_color = Add color from the neighboring legal pixels.

            // calculate the average.
            final color = sum color / no. of neighboring legal pixels.

            SetCurrentPixelColor(w, h, final_color) ;
        }
    }
}
```

Actually, if youï¿½ve turned on Bi-Linear filtering in your game, then, the effect of "pixels" are reduced. But, still we smoothen out the map, to make the final image appear really smooth. If the final result (in the game) looks good without blurring the light map texture, then, you may skip the BlurThisMap procedure.

There's another problem if we use bi-linear filtering. It's the **"bleeding"** problem.

When bi-linear filtering is turned on in your game, then, whenever a pixel is chosen, the final color will not be the color from the pixel alone, but, the average of the pixels around it. (average of how many pixels - depends on the kind of filtering used.)

As you know, some of the pixels in our texture map will be illegal. It means, that particular pixel belongs to no polygon.

Usually, the color of any illegal pixel will be zero, since no color calculation is being done for that pixel.

So, while rendering, whenever a pixel is chosen, the average of the color around that pixel will be considered. In this process, even the "illegal" pixels may be chosen. This is why "bleeding" happens.

somehow got to get rid of the illegal pixels.

Actually, we can't get rid of them. What we can do is fill every illegal pixel with a color from the closest "legal pixel". This way, during filtering, it is assured that the closest and most appropriate color is chosen.

This way, most of the "bleeding" problems will be solved. This is what the procedure *FillAllIllegalPixelsForThisLightMap* does.

One way of solving "bleeding" problem, without having to fill out illegal pixels, is to set the color of all the illegal pixels to ambient color. Even though this gives decent results and is also very inexpensive, it's not the correct way of doing it. Maybe, you can consider this method for real-time light map generation.

Take a look at the figure below:

Bleeding due to bi-linear filtering being turned on.

No bleeding even if bi-linear filtering is turned on.

Showing the lightmap:

Now that youï¿½ve taken so much time to calculate the light maps, itï¿½s ï¿½display timeï¿½. Hereï¿½s the code for DirectX 8.1:

```
// set the appropriate values for the texture stages.
// here Ii¿½m assuming that the device supports two or more texture stages.

SETTEXTURESTAGE(device8, 0, D3DTSS_COLOROP, D3DTOP_SELECTARG1) ;
SETTEXTURESTAGE(device8, 0, D3DTSS_COLORARG1, D3DTA_TEXTURE) ;

// multiply
SETTEXTURESTAGE(device8, 1, D3DTSS_COLOROP, D3DTOP_MODULATE) ;

SETTEXTURESTAGE(device8, 1, D3DTSS_COLORARG2, D3DTA_TEXTURE) ;
SETTEXTURESTAGE(device8, 1, D3DTSS_COLORARG1, D3DTA_CURRENT) ;

// set the appropriate vertex shader.
SETVERTEXSHADER(device8, FVF_MYVERTEXSHADER) ;

SETTRANSFORM(device8, D3DTS_WORLD) ;     // set the world matrix

// set the texture for the first stage.
SETTEXTURE(device8, 0, diffuseTexture) ;
```

```
// set the texture for the second stage.
SETTEXTURE(device8, 1, lightMapTexture) ;

DrawPrimitive() ;                                    // draw the polygons
```

Winding up...

As I have mentioned above, you have to look into Chris Heckerï¿½s article for more explanation on some of the equations I have used.

Some more illustration of using light maps:

To assert the effectiveness of using light maps, a scene has been rendered with different properties / effects. Click on each of the link below to see the images:

- A flat scene without light maps or any lighting.
- Scene rendered with simple vertex lighting.
- A light map that has been generated for the scene.
- Scene rendered with a medium resolution light map.
- Scene rendered with a high resolution light map.
- Scene rendered with light map only, i.e. no diffuse texture.

Where can you go from here?

You can use it to light up your ï¿½un-litï¿½ level
Implement dynamic light mapped lighting .. that would be cool.
Using the knowledge gained by using the light mapping technique, one can extend it to implement radiosity lighting.

Conclusion:

With the arrival of new monster graphics cards, lighting using lightmaps may become extinct. **This article does not provide you any cutting edge technology for todayï¿½s hardware**, but it does provide some basic but useful information on the process of creating light maps.

Demo:

An interactive demo has been included with this article where you can see the practical results of this article. I suggest that you download the demo and have a look at the results your self. After you have downloaded and unzipped all the files, look into the readme.txt file for more info. Feel free to play around with the lights. In the demo, you can add and delete static and dynamic lights, change the light properties and build light maps on the fly - and see the results. It's a very good example for using light maps.
Click here to download the demo (~2.1 MB).
(* Editor's note: source code is not included with this demo.)

Credits and Acknowledgements:

Please look at Chris Heckerï¿½s article on Perspective Texture Mapping at:

http://www.d6.com/users/checker

I would also like to thank my company for permitting me to publish this article. I also acknowledge the contributions of all my team members.

Links:

Click here for some info on light map co-ordinate generation.
http://www.flipcode.com/cgi-bin/msg.cgi?showThread=06June2000-LightmapStorage&forum=askmid&id=-1

Some more links to articles or docs about light maps:

http://polygone.flipcode.com/
http://www.flipcode.com/cgi-bin/knowledge.cgi?showunit=79
http://www.flipcode.com/tutorials/tut_lightmaps.shtml

Some Information about the author:

Keshav Channa is the team lead - Engineering, at Dhruva Interactive, India's pioneer in games development. He was part of the team which ported the Infogrames title Mission: Impossible to the PC and has been an integral part of Dhruva's in-house R&D efforts. Previous articles include ï¿½Geometry Skinning / Blending and Vertex Lightingï¿½ published on flipcode.

He is currently working on a multiplayer game for the PC. You can see his work at the Portfolio section at www.dhruva.com

Keshav can be contacted at kbc at dhruva dot com.

Derivation

NOTE: The following section is extracted from *Chris Heckerï¿½s article on Perspective Texture Mapping* and the derivation elaborated further.

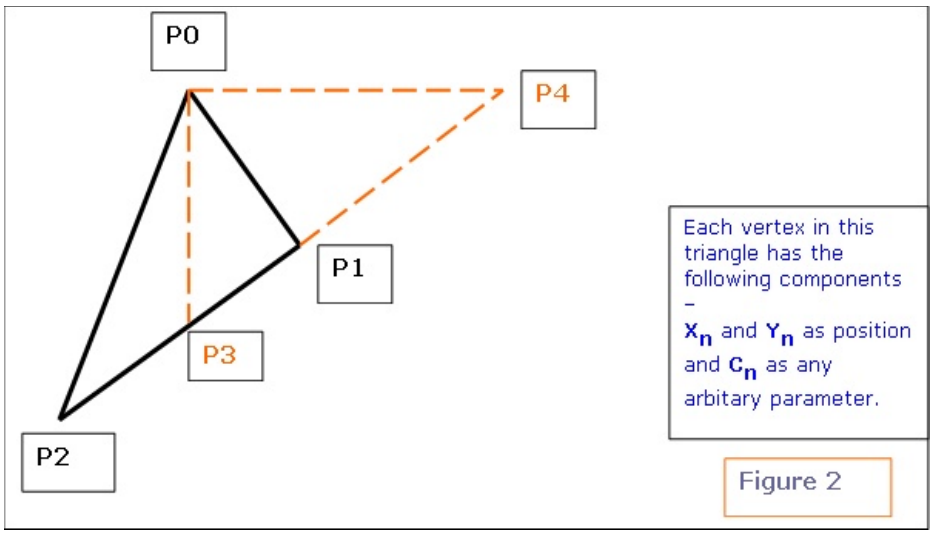


Figure 2

Given this Δ $\mathbf{P_0P_1P_2}$, let's figure out how parameter 'C' changes, if we hold 'Y' constant and step in 'X'.
We will use point P_4 in our construction.
P_3 and P_4 are both on the line P_1P_2.

It is clear that $Y_4 = Y_0$, we can derive the other co-ordinates for P4 using the line equations.

$$\frac{X_1 - X_2}{Y_1 - Y_2} = \frac{X_4 - X_2}{Y_4 - Y_2}$$

and

$$\frac{C_1 - C_2}{Y_1 - Y_2} = \frac{C_4 - C_2}{Y_4 - Y_2}$$

Substituting Y_0 for Y_4 and solving, we get,

$$X_4 = \frac{(X_1 - X_2) \bullet (Y_4 - Y_2)}{(Y_1 - Y_2)} + X_2$$

$$\Theta\,(Y_4 = Y_0) \Rightarrow$$

$$X_4 = \frac{(X_1 - X_2) \bullet (Y_0 - Y_2) + (X_2 \bullet (Y_1 - Y_2))}{(Y_1 - Y_2)} \quad \rightarrow (1)$$

Similarly,

$$C_4 = \frac{(C_1 - C_2) \bullet (Y_0 - Y_2) + (C_2 \bullet (Y_1 - Y_2))}{(Y_1 - Y_2)} \quad \rightarrow (2)$$

Now, we'll compute the difference in 'C' as it moves from P_0 to P_4.

$$\frac{dc}{dx} = \frac{(C4 - C0)}{(X4 - X0)}$$ → 2A

(The equations are from *Chris Heckerï¿½s article on Perspective Texture Mapping*. The only thing Iï¿½m doing from here onwards, is elaborating the derivation.) The derivation is very simple and involves plain substitution and some re-ordering.

Consider the numerator.
dc = (C_4 – C_0)

Substituting for ***C_4*** from ***equation 2***, we get,

$$dc = \frac{((C1 - C2) \bullet (Y0 - Y2)) + (C2 \bullet (Y1 - Y2))}{(Y1 - Y2)} - C0$$

(C_4 – C_0) =

$$\frac{((C1 - C2) \bullet (Y0 - Y2)) + (C2 \bullet (Y1 - Y2)) - (C0 \bullet (Y1 - Y2))}{(Y1 - Y2)}$$

(C_4 – C_0) = $$\frac{((C1 - C2) \bullet (Y0 - Y2)) + ((Y1 - Y2) \bullet (C2 - C0))}{(Y1 - Y2)}$$

(C_4 – C_0) = $$\frac{((C1 - C2) \bullet (Y0 - Y2)) - ((Y1 - Y2) \bullet (C0 - C2))}{(Y1 - Y2)}$$ → 3

From ***equation 2A***, we get, denominator **dx = ($X_4 - X_0$)**

Solving for ($X_4 - X_0$), by substituting for X_4 from ***equation 1***, we finally get::

$$(X_4 - X_0) = \frac{((X_1 - X_2) \bullet (Y_0 - Y_2)) - ((Y_1 - Y_2) \bullet (X_0 - X_2))}{(Y_1 - Y_2)} \quad (4)$$

Equation 3 / Equation 4 yeilds,

$$\frac{dc}{dx} = \frac{(C_4 - C_0)}{(X_4 - X_0)} = \frac{((C_1 - C_2) \bullet (Y_0 - Y_2)) - ((C_0 - C_2) \bullet (Y_1 - Y_2))}{((X_1 - X_2) \bullet (Y_0 - Y_2)) - ((X_0 - X_2) \bullet (Y_1 - Y_2))} \quad (5)$$

Similarly,

$$\frac{dc}{dy} = \frac{(C_3 - C_0)}{(Y_3 - Y_0)} = \frac{((C_1 - C_2) \bullet (X_0 - X_2)) - ((C_0 - C_2) \bullet (X_1 - X_2))}{((X_0 - X_2) \bullet (Y_1 - Y_2)) - ((X_1 - X_2) \bullet (Y_0 - Y_2))} \quad (6)$$

Let's see how, $\frac{dc}{dy}$ was derived.

We know that,

$$\frac{(X_1 - X_2)}{(Y_1 - Y_2)} = \frac{(X_3 - X_2)}{(Y_3 - Y_2)}$$

Also, $X_3 = X_0$.

$$\therefore Y_3 = \frac{(Y_1 - Y_2) \bullet (X_0 - X_2)}{(X_1 - X_2)} + Y_2 \quad \rightarrow \text{7A}$$

Similarly,

$$\frac{(X_1 - X_2)}{(C_1 - C_2)} = \frac{(X_3 - X_2)}{(C_3 - C_2)}$$

We know that $X_3 = X_0$, hence we get,

$$\therefore C_3 = \frac{(C_1 - C_2) \bullet (X_0 - X_2)}{(X_1 - X_2)} + C_2 \quad \rightarrow \text{7B}$$

Substituting for $\mathbf{Y_3}$ and $\mathbf{C_3}$ in the equation

$$\frac{dc}{dy} = \frac{(C_3 - C_0)}{(Y_3 - Y_0)}$$

yields

$$\frac{dc}{dy} = \frac{(C_3 - C_0)}{(Y_3 - Y_0)} = \frac{((C_1 - C_2) \bullet (X_0 - X_2)) - ((C_0 - C_2) \bullet (X_1 - X_2))}{((X_0 - X_2) \bullet (Y_1 - Y_2)) - ((X_1 - X_2) \bullet (Y_0 - Y_2))}$$

Finally the two equations that are of importance to us are:

$$\frac{dc}{dx} = \frac{(C_4 - C_0)}{(X_4 - X_0)} = \frac{((C_1 - C_2) \bullet (Y_0 - Y_2)) - ((C_0 - C_2) \bullet (Y_1 - Y_2))}{((X_1 - X_2) \bullet (Y_0 - Y_2)) - ((X_0 - X_2) \bullet (Y_1 - Y_2))}$$

and

$$\frac{dc}{dy} = \frac{(C_3 - C_0)}{(Y_3 - Y_0)} = \frac{((C_1 - C_2) \bullet (X_0 - X_2)) - ((C_0 - C_2) \bullet (X_1 - X_2))}{((X_0 - X_2) \bullet (Y_1 - Y_2)) - ((X_1 - X_2) \bullet (Y_0 - Y_2))}$$