# 5.3

# Dynamic Per-Pixel Lighting Techniques


## *Dan Ginsburg and Dave Gosselin, ATI Research*

**ginsburg@alum.wpi.edu and gosselin@ati.com**


A common method of performing dynamic lighting effects in games is to use vertex-based lighting techniques. A significant shortcoming of vertex-based lighting is that the geometry must be highly tessellated in order to even approach the quality of per-pixel lighting. This article presents several techniques that can be used to perform dynamic lighting effects on a per-pixel basis. These methods have the advantage that they don't require highly tessellated geometry, and can often be performed at little performance cost on multitexturing graphics hardware.

## 3D Textures for Dynamic Lightmapping

The first method of dynamic lighting we will examine is using a 3D texture for dynamic lightmapping. 3D textures can best be explained through their relation to 2D textures. In traditional 2D texturing, two texture coordinates are present in each vertex and are interpolated across the face. Each of the two texture coordinates (s, t) refers to the distance along one of the texture map dimensions *(width* and *height).* 3D textures expand upon 2D textures by adding a third texture coordinate (r) that refers to the depth of the texture. A 3D texture can be thought of as *depth* number of slices of 2D textures (see Figure 5.3.1). The texture coordinate *r* is used to select which

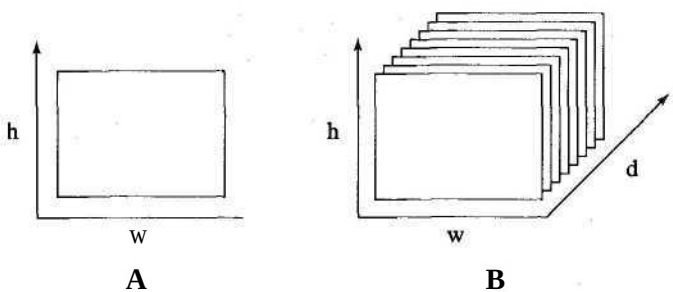


FIGURE **5.3.1** *A) 2D texture. B) 3D texture.*

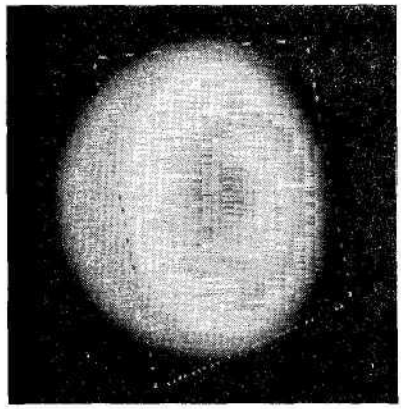

**FIGURE 5.3.2** *Cross-section of the 3D lightmap. This actual lightmap is monochrome; this picture rises color-coding to clearly show the mapfalloff. Red is highest intensity and blue is least. See color version in insert.*

of the 2D maps to access, and then coordinates *(s, t)* are used as normally in a 2D texture.

A natural use for 3D textures is in dynamic lightmapping. A typical application of dynamic lightmapping in games is having a moving light source that illuminates the geometry in the scene. For example, a rocket fired from the player would dynamically light the static game world geometry. A light of any shape can be created in a 3D texture. A simple case would be a sphere light with linear falloff from the center. In our example, a sphere light with quadratic falloff (Figure $.3.2) is used.

The 3D lightmap was generated with the following code:

## Listing 5.3.1 Code to generate 3D lightmap with quadratic falloff

```
for(r = 0; r < MAP_SIZE; r++)
      for(t = 0; t < MAP_SIZE; t++)
            for(s = 0; s < MAP_SIZE; s++)
            {
                  float DistSq = s * s + t * t + r * r ;

                  if(DiStSq < RADIUS_SQ)
                  {
                        float FallOff = (RADIUS_SQ - DistSq) /
                                        RADIUS_SQ;

                        FallOff *= FallOff;

                        LightMapfr * MAP_SIZE * MAP_SIZE + t *
                             MAP_SIZE + S] =
                              255.0f * FallOff;
```

```
            }
            else
            {
                LightMap[r * MAP_SIZE * MAP_SI2E + t *
                    MAP_SIZE + s] = 0;
            }
    }
```

The 3D lightmap itself can be specified in OpenGL using the EXT_texture3D extension (part of OpenGL 1.2.1). The 3D texture is sent to OpenGL using glTexImage3D(), which behaves much like giTexImage2D(). The only difference is that the depth of the image is specified, and the texel array contains *width* X *height* 'X *depth* texels.

The geometry in our example will be textured with a basemap *(b)* and will be modulated with a moving light source (/) represented by the 3D lightmap texture. The texture blending we wish to achieve is:

```
Result = b * 1
```

The texture environment can be configured to perform this operation through OpenGL using ARB_multitexture as follows:

```
// 3D texture lightmap on stage 0
glActiveTextureARB(GL_TEXTUREO ARB) ;
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE) ;

// Basemap to be modulated by 3D texture
glActiveTextureARB(GL_TEXTURE1_ARB);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE , GL_MODULATE) ;
```

Given the light position (/) and radius *(Ir),* the texture coordinate at each vertex *(v)* to look up into the 3D lightmap can be calculated as follows:

```
s = (v.x - 1.x) / Ir
t = (v.y - l.y) / Ir
r = (v.z - l.z) / Ir
```

Fortunately, this texture coordinate equation can be easily calculated in OpenGL using the texture matrix. First, the texture coordinate for each vertex is specified to be the same as its position *(s = v.x, t = v.y, r = v.z).* Next, each texture coordinate must be put into the space of the light by subtracting the light position. The texture matrix can be used to perform this operation by adding a translation to the texture matrix of minus the light position (— /). The texture coordinate now needs to be divided by the radius of the light. The texture matrix can be used to perform this operation by scaling the texture matrix by the reciprocal of the light radius *(1/lr)* in *s, t,* and *r.* The following block of code configures the texture matrix as described using a uniform scale.

```
glMatrixMode(GL_TEXTURE) ;
glLoadIdentityO ;
```

```
glScalef(1 / lr, 1 / lr, 1 / lr);
glTranslatef(-1.x, -l.y, -l.z);
```

Note that by using a nonuniform radius, the shape of the light could be modified to produce shapes other than a sphere without needing to modify the texture. For example, making the scale in *x* larger than in *y* and *z,* the light would be shaped as an ellipsoid.

Given this setup, each polygon is rendered with its basemap modulated by the 3D lightmap. Figure 5.3.3 shows the effect achieved by this 3D lightmapping technique.

Note that while 3D textures are a natural method for performing dynamic per-pixel point lights, 3D textures are not the most memory-efficient technique. In *Game Programming* G^—"Attenuation Maps," Dietrich presents a method for performing per-pixel point lights using only 2D and ID textures.

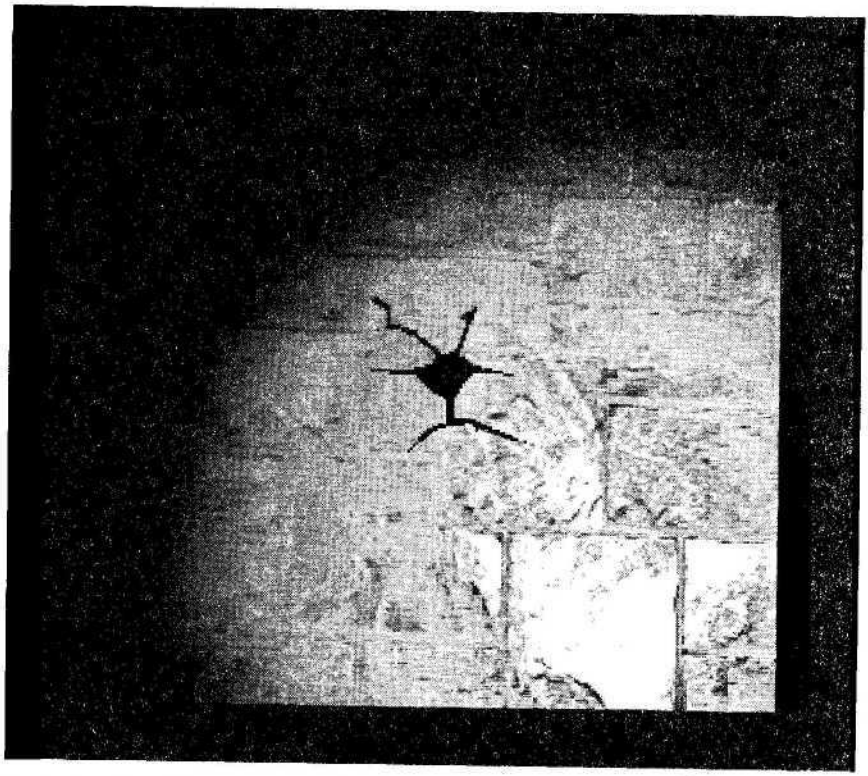

**FIGURE 5.3.3** *Basemap modulated by the 3D lightmap. See color version in insert.*

## Dot3 Bump Mapping

The effect presented in the previous section using a 3D texture as a dynamic lightmap will now be improved by adding per-pixel light perturbations to the surface. Several available graphics cards provide this functionality through Dot3 bump mapping. In OpenGL, Dot3 bump mapping is exposed through the EXT_texture_env__dot3 extension. This extension simply adds a new blend equation to the blend modes supported by EXT_texture_env_combine (GL_MODULATE, GLJNTERPOLATE, etc.). The GL_DOT3_RGB equation performs the following computation:

```
Dest.r = Dest.g = Dest.b =4 * ((AO.r - 0.5) * (A1.r - 0.5) +
                               (AO.g - 0.5) * (A1.g - 0.5) +
                               (AO.b - 0.5) * (A1.b - 0.5))
```

This blend equation performs a dot product between the (r, g, b) vectors of any two color sources.

In order to allow for color values to represent vector components in the range [-1, 1], the color values are made signed by subtracting .5 from each component. However, by performing this subtraction and multiplying each component, the outgoing color value is not scaled properly. In order to place the result of the dot product in the proper scale, it is multiplied by 4.

Using this equation, the normal vector *(N)* at each texel can be encoded into the (r, g, b) values of a texture. The light vector *(L)* in texture space can be encoded into the (r, g, b) values of another source. The resultant dot product performed at blend time yields the equation TV * *L* at each texel. This is the fundamental calculation that is needed for per-pixel bump mapping.

The texture map used for bump mapping will contain normals that will perturb the light vector over the surface. In order to generate the bump map, a grayscale image representing a heightfield of the original base map is authored. From this heightfield, the bump map can be generated by calculating the change in texel height in each direction.

The height differences can be calculated using a Sobel filter. The Sobel filter provides two image filters that detect vertical and horizontal change.

$$dx = \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \quad dy = \begin{bmatrix} 1 & 0 & -1 \\ -2 & 0 & 2 \\ 1 & 0 & -1 \end{bmatrix}$$

The normal at each texel is then equal to (-dx, -dy, 1). This vector is normalized and placed into the color components at the texel in the bump map. Figure 5.3.4 shows the bump map generated in the example.

At each vertex, the light vector will be stored in the primary color component. With smooth shading on, this vector will be linearly interpolated across the face. The interpolated vector will be used in each per-pixel dot product. This vector must be

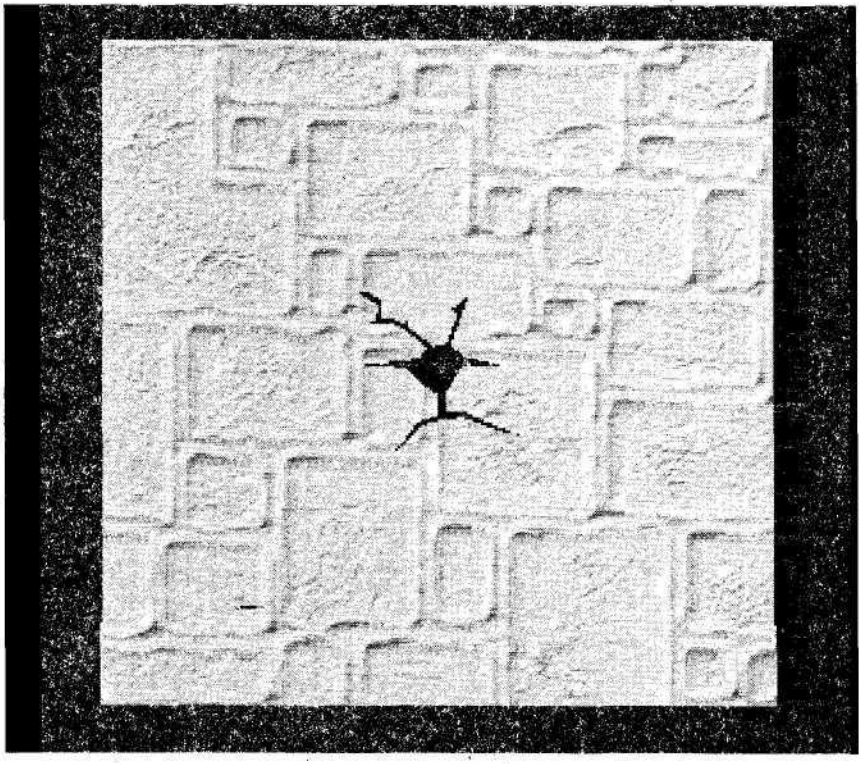

**FIGURE 5.3.4** *Dot3 bump map texture generated from the beightfield using the Sobel filter. See color version in insert.*

transformed into the space of the bump map texture on each polygon before it can be used in the dot product. For point lights, the world space light vector at each vertex is calculated by subtracting each vertex position from the center position of the 3D light. In our case, anytime the 3D lightmap or the geometry moves, the tangent space light vector must be recalculated. Note that to use Dot3 bump mapping with a static directional light, the tangent space light vector can be precalculated because the per-vertex light vector doesn't change.

In order to rotate the light vector into the space of the texture (known as tangent space), a local coordinate system at each vertex is constructed. Three vectors define this coordinate system: the vertex normal *(N),* a tangent to the surface *(T),* and the binormal *(B).* The vertex normal is determined by taking the average of all the normals to the faces that contain the vertex. The tangent vector is calculated by determining the direction of increasing *s* or *t* texture coordinates along the face in the object's coordinate system. The binormal is then calculated by taking the cross

product of the vertex normal and tangent vector. Given this coordinate basis, the light vector can be rotated into the space of the bump map using the following rotation matrix:

```
[ T.x T.y T.z (T|
  B.X B.y B.z 0 i
  N.X N.y N.Z 0 |
  0 0   0   1 J
```

Finally, this tangent space light vector is stored in the primary color component of the vertex.

Given the Dot3 texture map *(dotmap)*, the tangent space light vector *(tl)*, the basemap *(b)*, and the 3D lightmap (/), we wish to perform the following texture blending:

```
Result = (dotmap DOT3 tl) * b * 1
```

Assuming hardware that has three orthogonal texture units, this can be achieved by configuring the texture environment as follows (in practice, this may need to be done on multiple passes on many graphics cards):

```
// dotmap DOTS tl on stage 0
glActiveTextureARB(GL_TEXTUREO_ARB);
g!TexEnvi(GL_TEXTURE_ENV, GL_TEXTURE ENV_MODE, GL_COMBINE_EXT);
glTexEnvi(GL_TEXTURE_ENV, GL_COMBINEJ1GB_EXT, GL_DOT3_RGB_EXT);
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCEO_RGB_EXT,
GL_PRIMARY_COLOR_EXT);
glTexEnvi(GL_TEXTURE_ENV, GL_OPERANDO_RGB_EXT, GL_SRC_COLOR);
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE1_RGB_EXT, GL_TEXTURE);
glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND1_RGB_EXT, GL_SRC_COLOR);

// previous * basemap on stage 1
glActiveTextureARB(GL_TEXTURE1_ARB);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_MODULATE);
g!TexEnvi(GL_TEXTURE_ENV, GL_SOURCEO_RGB_EXT, GL_PREVIOUS_EXT);
glTexEnvi(GL_TEXTURE_ENV, GL_OPERANDO_RGB_EXT, GL_SRC_COLOR);
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE1_RGB_EXT, GL_TEXTURE);
glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND1_RGB_EXT, GL_SRC_COLOR);

// previous * lightmap on stage 2
glActiveTextureARB(GL_TEXTURE2_ARB);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_COMBINE_EXT);
glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_MODULATE);
g!TexEnvi(GL_TEXTURE_ENV, GL_SOURCEO_RGB_EXT, GL_PREVIOUS_EXT);
glTexEnvi(GL_TEXTURE_ENV, GL_OPERANDO_RGB_EXT, GL_SRC_COLOR);
glTexEnvi(GL_TEXTURE_ENV, GL_SOURCE1_RGB_EXT, GL_TEXTURE);
glTexEnvi(GL_TEXTURE_ENV, GL_OPERAND1_RGB_EXT, GL_SRC_COLOR);
```

Figure 5.3.5 shows the results of this technique.

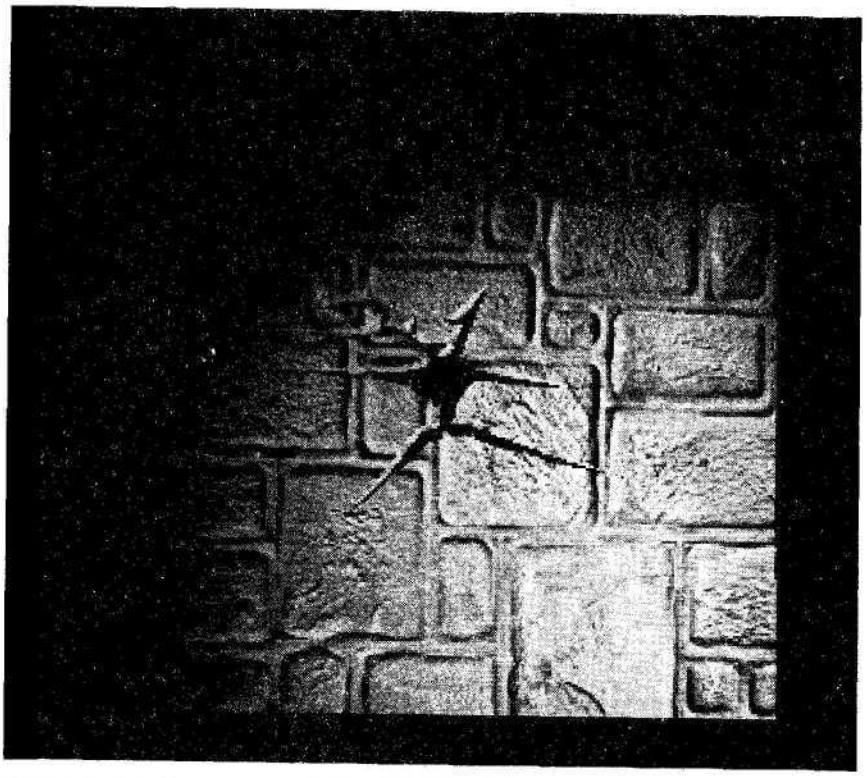

**FIGURE 5.3.5** *Base map modulated by 3D lightmap with Dot3 bump mapping. See color version in insert.*

## Cubemap Normalizer

We now have a bump-mapped object dynamically lit on a per-pixel basis. However, there is one major shortcoming with the method presented. The tangent space light vector calculated at each vertex used in the Dot3 blend stage was stored in the primary color component. By storing the light vector in the color component with smooth shading enabled, the vector is linearly interpolated across the face. Unfortunately, the linearly interpolated vector will often fail to maintain its length of 1 across the face. Figure 5.3.6 shows an example of two linearly interpolated 2D vectors. VI and V2 both have a length of 1, but when linearly interpolated, the resultant vector has a length less than 1. This failure to maintain normalization results in decreased visual quality, particularly when the geometry is not highly tessellated.

In order to correct this problem, a cubic environment map will be used to keep the light vector normalized. The contents of the cube map at each texel will be the *same* as the normalized texture coordinate used to look up into it. Instead of storing the tangent space light vector in the primary color component, it will be stored as a texture coordinate that will be used to look up into the cubemap. Since each texel in

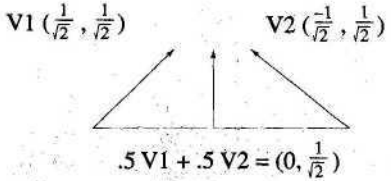


**FIGURE 5.3.6** *The linear interpolation of two normalized vectors resulting in a vector whose length is less than 1.*

the map contains the normalized vector of the texture coordinate, the light vector will stay normalized when interpolated across the face.

The Dot3 operation using the cubemap can be done using two texture units by configuring the texture environment as follows:

```
// Cubmap normalizer on stage 0, just pass it through
glActiveTextureARB(GL_TEXTUREO_ARB);
g!TexEnvi(GL_TEXTURE_ENV,  GL_TEXTURE_ENV_MODE,
            GL_COMBINE_EXT);
glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT, GL_REPLACE);
glTexEnvi(GL_TEXTURE_ENV,  GL~SOURCEO_RGB_EXT,  GL_TEXTURE);
glTexEnvi(GL_TEXTURE_ENV,  GL_OPERANDO_RGB_EXT,  GL_SRC_COLOR);

// dotmap DOT3 cubemap on stage 1
glActiveTextureARB(GL_TEXTURE1_ARB);
glTexEnvi(GL_TEXTURE_ENV,  GL_TEXTURE_ENV_MODE,
            GL_COMBINE_EXT);
glTexEnvi(GL_TEXTURE_ENV, GL_COMBINE_RGB_EXT,
            GL_DOT3_RGB_EXT);
glTexEnvi(GL_TEXTURE_ENV,  GL_SOURCEO_RGB_EXT,
            GL_PREVIOUS_EXT);
glTexEnvi(GL_TEXTURE_ENV,  GL_OPERANDO_RGB_EXT,  GL_SRC_COLOR);
glTexEnvi(GL_TEXTURE_ENV,  GL_SOURCE1_RGB_EXT,  GL_TEXTURE);
glTexEnvi(GL_TEXTURE_ENV,  GL_OPERAND1_RGB_EXT,  GL_SRC_COLOR) ;
```

Figure 5.3.7 shows the improved results by using the cubemap normalizer.

## Per-Pixel Spotlights

In addition to per-pixel point lights as we have implemented using a 3D lightmap, another useful dynamic lighting technique is per-pixel spotlights. Instead of using a 3D lightmap, we will use 2D texture to represent the light and project it onto each face. The contents of the 2D texture will be a cross-section of the spotlight cone containing the attenuation falloff from the center. As with point lights, a tangent space light vector from the spotlight will be placed in the color component of the vertex. A falloff factor based on the distance from the tip of the cone to the vertex is placed in the alpha channel. The tangent space light vector will be Dot3'd with the vectors in

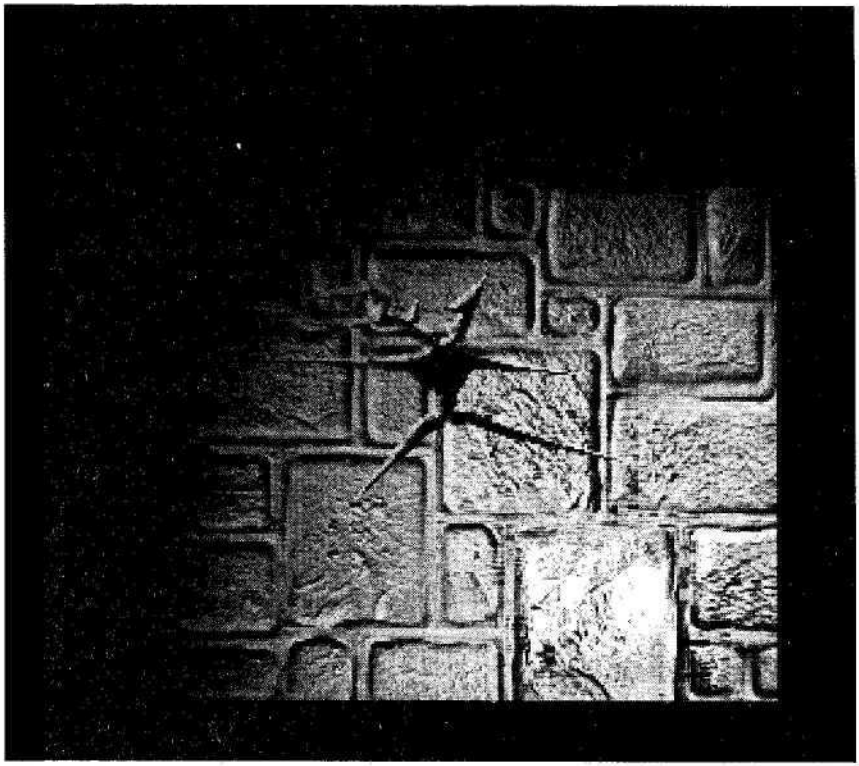

**FIGURE 5.3.7** *Base map modulated by 3D lightmap with Dot3 bump mapping and cubemap normalizes See color version in insert.*

the normal map. The interpolated alpha in the vertex color is modulated with the color to perform the distance attenuation.

The result of this Dot3 calculation now need to be modulated by the projected spotlight texture. The spotlight can be projected onto the face by configuring the texture matrix to perform the projection. The texture matrix is first set to represent the view from the spotlight by calculating the "look-at" matrix based on its position, direction, and any vector perpendicular to the direction. The "look-at" texture matrk must finally be multiplied by a matrix that represents the frustum created by the spotlight. The texture coordinates for the spotlight texture will then simply be the world coordinates of the vertex. The result of this texture look-up is modulated with the DOT3 calculation to get the final lighting.

## References

[Blythe98] Blythe, David, "Advanced Graphics Programming Techniques Using OpenGL" available online at www.sgi.com/software/opengl/advanced98/notes/notes.html

[DeLouraOO] DeLoura, Mark, *Game Programming Gems,* Charles River Media, 2000.
Dietrich, Sim, "Attenuation Maps," pp. 543-548.
Dietrich, Sim, "Hardware Bump Mapping," pp. 555-561.

[Mitchell] Mitchell, Jason, "Radeon Bump Mapping Tutorial," available online at www.ati.com/na/pages/resource_centre/dev_rel/sdk/RadeonSDK/Html/Tutorials/RadeonBumpMap.html

ARB_multitexture, EXT_texture_env_combine, and EXT_texture_env_dot3 spec available online at http://oss.sgi.com/projects/ogl-sample/registry/