# Realtime Pixel Lighting Using Fragment Programs

**Steven Hales**
**Lockheed Martin**
**Orlando, FL**
**steven.ha1es@lmco.com**

## ABSTRACT

This paper will describe techniques to implement the lighting algorithms for point and spot light sources at a pixel level at realtime rates using fragment program capabilities on current graphics cards.

Visual simulation has long been required to simulate point and spot light sources such as headlights, search lights, flares, and steerable landing lights. To calculate lighting at a pixel level from these sources has traditional been difficult to run at realtime rates. Today's commercial graphics cards now have the ability to do pixel lighting at realtime rates using fragment programs (pixel shaders). The complex calculations for range and angle attenuation of these lights sources can be imbedded into texture maps and "looked-up". Two or multi-pass algorithms per polygon of the past are no longer necessary.

The generation of texture coordinates and generation of the texture maps from the attenuation calculations will be described. Lighting from directional light sources (e.g., sun or moon) and illumination of fog will also be addressed.

The pixel lighting performance of current commercial graphics cards (NVIDIA/ATI) will be discussed.

## ABOUT THE AUTHOR

**Steven Hales** is a senior staff systems engineer at Lockheed Martin. Steve graduated from the University of South Florida in 1976 with a degree in electrical engineering. He worked for 10 years at (then) Sperry Univac (now Unisys) in CPU design of mainframe computers. Steve has worked for Lockheed Martin (Martin Marietta/GE) since 1987 in visual simulation image generation.

# Realtime Pixel Lighting using Fragment Programs

**Steven Hales**
**Lockheed Martin**
**Orlando, FL**
**steven.ha1es@lmco.com**

## OVERVIEW

Graphics 3D APIs such as OpenGL and Direct3D specify lighting algorithms that are applied to the vertices of a polygon. Gourard shading is then applied such that the color at a pixel is interpolated from the color of the vertices. While this may produce adequate results for directional (infinite) light sources, such as the sun, it does not generally produce adequate results for point or spot light sources. Range attenuation may be inadequate, especially with large polygons. The lobe effect of a spotlight may be greatly distorted or missing. A classic example is a flashlight shining on the center of a large polygon. There would be no lighting since the vertices are all outside the cone angle of the light. The only solution is to tessellate the polygon until the desired effect is achieved. However, tessellation is an expensive solution.

With the advent of fragment programs (pixel shaders), the ability to do lighting algorithms in the pixel pipeline is now available. However, the calculations to do the lighting algorithms for a pixel in the same manner as a vertex are expensive. The current graphics cards do not have the performance to do these calculations and meet the realtime requirements for visual simulation in military ground and flight simulators. However, most of these calculations can be embedded in texture maps and "looked up", making realtime operation achievable.

## COMPUTE LIGHTING USING TEXTURE MAPS

### Lighting Algorithms

The lighting algorithms applied to a vertex for spot light sources can be broken into two parts: range attenuation and angle attenuation. Point lights sources, such as flares, only require range attenuation. Lets define **V** to be the position of the vertex and **L** the position of the light. Lets define **P** to be the vector from the light to the vertex, R2 as the square of the range to the light, and R as the range to the vertex from the light. Both range and angle attenuation require the following equations:

**P = V – L**
R2 = **P** * **P**
R = square root (R2)

Range attenuation is controlled by three parameters, the constant (C), linear (L), and quadratic (Q) attenuation factors. The equation for range attenuation is:

Attenuation = 1 / (C + L * R + Q * R2)

For spotlight angle attenuation, lets define **l** to be the direction of the light. Two parameters, the cone half angle (HA) and the spot exponent (S) control angle attenuation. The cosine of the angle between the direction of the light and the direction from the light to the vertex is defined as COSA. The equations for angle attenuation are:

COSA = **l** * **P** / R
If COSA > cosine (HA),
  Attenuation = COSA**S
Else
  Attenuation = 0

### Embedding Range Attenuation in a Texture Map

In general, texture coordinates must be between zero and one when looking up a texture map value (texel). Texture coordinates can be clamped to one if greater than one and to zero if less than zero. If we use a range extent of the light, we can keep the texture coordinates between zero and one while within the extent of the light, and clamped when outside the extent of the light. The range attenuation equation only approaches zero, but a small attenuation value, a (e.g. 0.05), can be chosen at which we will force the attenuation to zero.

This chosen attenuation value can be used to compute the range extent, E, of the light by solving the quadratic equation for R, where Q * R2 + L * R + (C – 1/a) = 0. Then, 3D texture coordinates s, t, and r can be set as:

s = 0.5 + k * **Px**
t = 0.5 + k * **Py**
r = 0.5 + k * **Pz**
where the constant k = 0.5 / E.

When a component of the vector to light equals –E, the texture coordinate is zero, and when the component equals E, the texture coordinate is one. At any texel in the map, the vector from the light to the texel can be determined by:

**Px** = (s – 0.5) / k
**Py** = (t – 0.5) / k
**Pz** = (r – 0.5) / k

The range to the point is then calculated and the range attenuation equation applied to determine the value of the texel. The boundaries of the map should have the texel value set to zero. In effect, the texels of the texture map are defining the attenuation at discrete points, but linear texture filtering can be used to provide smoothing.

Specifically, for a given width, height, and depth of the 3D texture map, the texture map data is computed as follows:

```
For i = 0 through (depth – 1)
  r = i / (depth – 1)
  z = (r – 0.5) / k
  For j = 0 through (height – 1)
    t = j / (height – 1)
    y = (t – 0.5) / k
    For k = 0 through (width – 1)
      s = k / (width – 1)
      x = (s – 0.5) / k
      R2 = x * x + y * y + z * z
      If (R2 > E) texel = 0
      else
      R = square root (R2)
      attenuation = 1 / (C + L * R + Q * R2)
      If attenuation > 1.0, attenuation = 1.0
      texel = attenuation converted to proper data format
```

An issue of using a 3D texture map is the amount of texture memory required. However, a 128x128x128 map normally gives adequate resolution.

If the resolution versus texture memory issue makes using a 3D map unfeasible, a slightly slower and complex method could be used. Use s and t with a 2D map to look up a texture coordinate, q, where

q = 0.5 + k * |**Px,Py**|

Then, q and r can be used to look up the range attenuation with a 2D texture map. The resolution of these two 2D texture maps could be increased greatly over the 3D map. However, most current cards are limited to four texture maps (the Radeon 9800 and X800 have eight). If a graphics card only has four texture maps and a polygon has two textures, only one would be available for range attenuation, given one is required for angle attenuation. In this case, a 3D texture map would have to be used.

**Embedding Angle Attenuation in a Texture Map**

A texture cube map will be used to look up angle attenuation. The texture coordinates is determined by first rotating the vertex into light coordinates, where the x-coordinate is in the direction of the light. Typically, a spot light may be defined relative to a viewpoint, with an azimuth and elevation. The viewpoint also may have heading, pitch, and roll. The vertices may be relative to a world coordinate set.

Once the vertex is in light coordinates, then y and z components are scaled by tangent of (90-half angle), such that the entire cube map face covers the cone angle. The positive x face (|x| > |y| and |x| > |z| and x > 0) of the cube map will have the lobe pattern. The other five faces of the cube map are zero. The texture coordinates sent with the vertex will be

s = x
t = a * y
r = a * z

where the constant a = tangent (90 – HA).

Internally, the texture coordinates to the positive x cube map will be:

s = 0.5 + 0.5 * t / s
t = 0.5 + 0.5 * r / s

So, given the s and t, COSA is computed as follows:

y/x = b * (s – 0.5)
z/x = b * (t – 0.5)
COSA = 1 / square root (y/x * y/x + z/x * z/x + 1)

where the constant b = 2 / a. The rest of the angle attenuation equation can be applied to determine the texel value.

Specifically for a given image size of the cube map, the texture map data is computed as follows:

```
For i = 0 through (size – 1)
  t = i / (size – 1)
  z/x  = b * (t – 0.5)
  For j = 0 – (size – 1)
    s = j / (size – 1)
    y/x = b * (s – 0.5)
   COSA = 1 / square root (y/x * y/x + z/x * z/x + 1)
    If COSA > cosine (HA),
      Attenuation = COSA**S
   Else
     Attenuation = 0
   texel = attenuation converted to proper data format
```

## OTHER LIGHTING ISSUES

### Directional Light Source

Since the fragment program requires the fragment color before lighting, lighting must be disabled. This requires that directional light sources must be handled in the fragment program.

If flat shading is active, this only requires that the shading value be passed as a texture coordinate to the fragment program. Assuming no specular lighting, the shading value f is calculated as follows:

f = ambient + diffuse * (**d** * **n**)

where **d** is the light direction and **n** is the face normal.

If smooth shading is active, the resultant shading at a vertex can be passed as a texture coordinate. The interpolated texture coordinate at the fragment is the shading value.

### Multiple Light Sources

In the case where multiple point or spot light sources are required, such as headlights, the range and angle attenuation texture maps may be able to be used for the additional light sources.

If the range attenuation parameters (C, L, Q) are the same, only additional texture coordinates are required. These would be generated as before, but using the new light position (**L**). If the light sources are close together, as in the case of headlights, then the same texture coordinates might be used for both sources.

If the angle attenuation parameters (HA and S) are the same, only additional texture coordinates are required. These would be generated as before, but using the new light position (**L**).

Most current graphics cards support eight texture coordinates, so up to four spot light sources could be handled in this manner (given a texture coordinate used for directional light source).

### Lighting in Fog

The fragment program must also handle lighting in fog. The effect of the fog will depend on if a fragment is inside or outside the spot light, the range attenuation of the light, and the fog density. The amount of fog increases with range, but the intensity of the light decreases. So, the issue is determine how must light is reflected back off the fog.

A method to simulate this effect is to determine a fog color to approximate the amount of reflection. A method of doing this is to take samples across the extent of the light. At each sample, calculate the fog factor and the attenuation. Multiply one minus the fog factor times the attenuation. Take the maximum value of the samples as the fog color intensity for the light. This value can be passed as a parameter to the fragment program. A fragment will compute its fog color as:

Fragment fog color = light color * angle attenuation * fog color intensity + ambient fog color

The fog factor at the fragment is then used to interpolate between color before fog and this fragment fog color.

## CREATING THE FRAGMENT PROGRAM

### Fragment Program Lighting Equations

The general equation for lighting at a face fragment (pixel) is:

Output color = (Sum of (range attenuation * angle attenuation * (ambient light color + (**l***N) * diffuse light color) for all non-directional light sources + directional ambient + directional diffuse * (**d** * **n**)) * face fragment color * face texture 1 * face texture 2 (if any).

**Sample Fragment Program**

Lets assume a two headlights case, with range attenuation and spot light parameters being the same for both lights. We will assume the lights are close enough together that the same range attenuation value can be used for both lights. We will ignore diffuse lighting for the headlights and apply the light color to the ambient light color. Let RA be range attenuation and AA1 and AA2 be angle attenuation for the two lights. Let lcol be the light color and icol the fragment color. The resultant lighting equation is:

Output color = (RA * (AA1 + AA2) * lcol + f) * icol * tex1 * tex2

Using the OpenGL ARB_fragment_program extension, the sample program is:

```
static char *arb_pixel_light =
 "!!ARBfp1.0                                        \
  OPTION   ARB_fog_exp2;                            \
  OUTPUT   output = result.color;                   \
  ATTRIB   tc_t1   = fragment.texcoord[0];   \
  ATTRIB   tc_t2   = fragment.texcoord[1];   \
  ATTRIB   tc_ra   = fragment.texcoord[2];   \
  ATTRIB   tc_aa1 = fragment.texcoord[3];   \
  ATTRIB   tc_aa2 = fragment.texcoord[4];   \
  ATTRIB   tc_f    = fragment.texcoord[5];   \
  ATTRIB   icol  = fragment.color.primary;   \
  PARAM   lcol = state.light[1].ambient;     \
  TEMP    tex1;                                     \
  TEMP    tex2;                                     \
  TEMP    ra;                                       \
  TEMP    aa1;                                      \
  TEMP    aa2;                                      \
  TEMP    aa;                                       \
  TEMP   acc;                                       \
  TXP     tex1, tc_t0, texture[0], 2D;              \
  TXP     tex2, tc_t1, texture[1], 2D;              \
  TEX     ra,   tc_ra, texture[2], 3D;              \
  TEX     aa1,  tc_aa1, texture[3], CUBE;     \
  TEX     aa2,  tc_aa2, texture[3], CUBE;     \
  ADD     aa, aa1, aa2;                             \
  MUL     acc, aa, ra;                              \
  MAD_SAT acc.xyz, acc, lcol, tc_f.x;         \
  MUL     acc, acc, tex1;                           \
  MUL     acc, acc, tex2;                           \
  MUL_SAT output, acc, icol;                        \
  END";
```

**FRAGMENT PROGRAM RESULTS**

The fragment program above was applied to a typical simulation and training database, as shown in Figure 1, using the ATI Radeon X800 Pro card to show per pixel lighting. Figure 2 shows the same scene with low visibility (fog). The fragment program was changed to implement the logic described in the "Lighting in Fog" section.

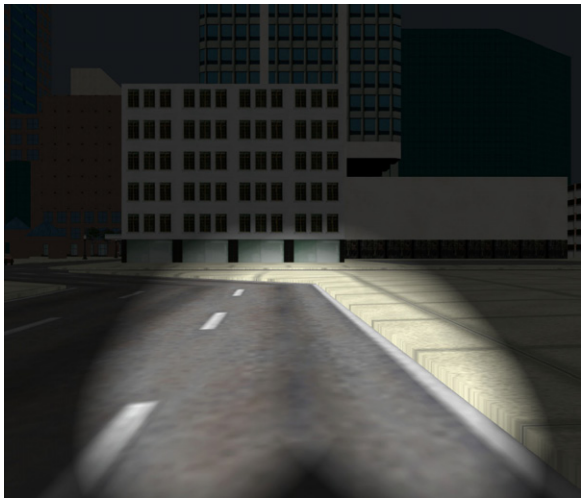

**Figure 1.** Per Pixel Lighting Without Fog

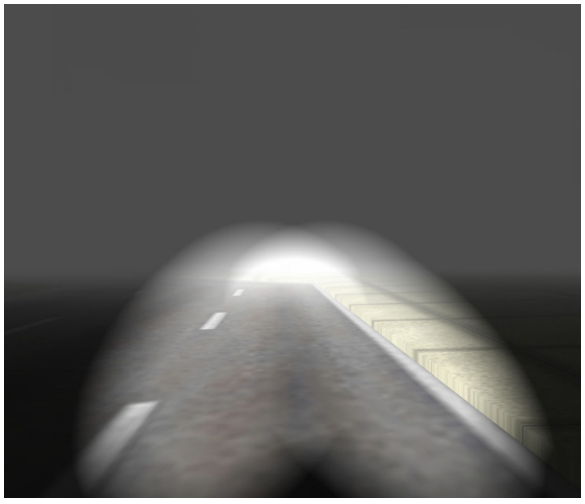

**Figure 2.** Per Pixel Lighting With Fog

**GRAPHICS CARDS PERFORMANCE**

The scene in Figure 1 was used to measure the performance of several graphics cards with (shader) and without (legacy) per pixel lighting. The results were normalized to the fastest card (ATI X800). The fragment program was executed on all polygons.

**Table 1.** Normalized Graphics Card Data

| | **Legacy** | **Shader** |
|---|---|---|
| **ATI Radeon X800 PRO** | 1.0 | 1.95 |
| **ATI Radeon 9800 PRO** | 1.130 | 2.450 |
| **Nvidia GeForce FX 5950** | 1.847 | 4.830 |

The ATI 9800 card has significantly better performance than the Nvidia 5950, with and without per pixel lighting. The X800 was just released, but the Nvidia 6800 was not available. However, there is significant improvement between the ATI X800 and 9800, with more improvement on the shader path than the legacy path.

The pixel processing time degrades significantly when the fragment program is enabled, even with texture lookup to eliminate most of the calculations. The pixel processing time at least doubles.

Even a fragment program that just mimics the legacy texture pipeline logic runs much slower than just using the legacy pipeline.

However, the polygons within the extent of the light (and within the cone angle) can be sorted from those outside the extent of the light (or outside the cone angle). The fragment program would be enabled only for those polygons within the extent (and cone angle) of the light.

## CONCLUSION

Other shading languages are available, such as Cg and OpenGL Shading Language. Direct 3D is an option besides OpenGL. But, it is unlikely that the performance issues are dependent on the platform used.

As far as how simulation and training programs will be able to leverage the rapidly changing GPU programmability and programming interfaces, the approach used here for pixel lighting (point and spot light sources), is the same approach as used for bump mapping, Phong shading, and other advanced graphics features. In general, however, the use of fragment programs may remain somewhat limited given current performance levels. Limiting the number of polygons that require the use of the fragment programs can give a trade off between performance and image quality.