5.4

Attenuation Maps

Sim Dietrich

Vertex lighting is good for many applications and is well known and understood. It has many benefits, including the benefit of properly handling surfaces not facing the light, but vertex lighting can have artifacts when the size of the triangle is large with respect to the range of a point or spotlight.

Light maps are another approach to calculating lighting that can avoid these triangle tessellation-related artifacts, but they require expensive CPU operations to update for dynamic lights and require potentially slow upload to the video card. Still, light maps are a good solution for static lighting and shadows.

This article introduces a novel technique known as *attenuation maps*. This technique can be used to implement dynamic point lights with proper quadratic attenuation using multitexture operations. In addition, the technique can be used for spherical, ellipsoidal, cylindrical, and rectangular lighting or CSG operations, accurate to a per-pixel level, without using the stencil buffer.

Explanation

The attenuation function for lighting is typically like so:

```
X = lightPosition.X - vertexPosition.X;
Y = lightPosition.Y - vertexPosition.Y;
Z = lightPosition.Z - vertexPosition.Z;

D = sqrt(X*X + Y*Y + Z*Z);

Att = 1 / (C0 + C1*D + C2*D*D);
```

For our purposes, we assume that we want only quadratic attenuation, so assume that *C0* is 1 and *C1* is 0, giving:

```
Att = 1 / (1 + C2*D*D);
```

3D textures provide a simple method of encoding this function. Directly store the function in a 3D texture as a function of *X*, *Y*, and *Z* for the three texture coordinates. Then, simply set up texture coordinate generation to calculate *dX*, *dY*, and *dZ* relative to the light position, use the texture matrix to scale each *dX*, *dY*, and *dZ* by 1 over the light's range, and scale and bias so the center of the texture corresponds to (0,0,0).

The equation that the texture matrix computes in this case is:

```
S = ((Light.X - Vertex.X) / LightRange) / 2.0f) + 0.5f;
T = ((Light.Y - Vertex.Y) / LightRange) / 2.0f) + 0.5f;
R = ((Light.Z - Vertex.Z) / LightRange) / 2.0f) + 0.5f;
```

However, 3D textures aren't yet widely available, so we must make do with 2D and 1D textures. Even were 3D textures available, it would be advantageous to find a less texture memory-intensive method of calculating attenuation.

Well, first, since we have only 2D and 1D textures available, that means that we can't compute the function with only a single texture because we can't use all three coordinates *X*, *Y*, and *Z* at once. This means that we have to break the function into two or more parts.

How can we express the attenuation function in such a way that we can implement it using 2D and 1D textures? Let's start by breaking *X* and *Y* into a 2D texture and *Z* into its own 1D texture.

If we break *X* and *Y* into one texture and *Z* into another, that means that the result of the function of *X* and *Y*, which we call `f(X,Y)`, and the result of the function based on *Z*, which we call `g(Z)`, must be expressible as colors.

In other words, if we are storing the function as some combination of two textures, we must express the final function as a sum or multiplication of two colors. Since colors can hold only positive values from 0 to 1, this affects which form of attenuation function we can choose.

The previous attenuation function is:

```
Att = 1 / (1 + c2*D*D);
```

Some samples of this function follow:

Att(0) == 1
Att(2) == 0.5
Att(Large D) approaches 0

As *D* gets larger, the attenuation approaches 0.

Let's try to encode this function in two textures. First, we expand *D*D* into its components:

```
D = sqrt(X*X + Y*Y + Z*Z)

D*D = (X*X + Y*Y + Z*Z)
```

Now we restate the attenuation function in terms of *X*, *Y*, and *Z*:

```
Att = 1 / (1 + C2*(X*X + Y*Y + Z*Z))
```

Now it seems we are stuck because we can't express the Att function as the sum or product of the two functions `f(X,Y)` and `g(Z)` due to the fact that *X, Y*, and *Z* are all in the denominator. We have to find another function that we can separate.

We don't have any way of summing two colors and then taking a reciprocal, so we have to find a function that has the same effect but doesn't require a reciprocal.

Squaring numbers greater than one produces larger numbers, whereas squaring a number in the range (0..1) produces a smaller number. For instance, 0.5 * 0.5 equals 0.25. This is why we had to set the constant in our attenuation function, *C0*, to 1: It prevents really close lights from becoming brighter than they should.

Remember that colors are always expressed in the range [0..1]. That means that the result of `f(X,Y)` and `g(Z)` must produce results in the same range [0..1].

One function that both does not require reciprocals and produces results within the range [0..1] is:

```
Att = 1 - D*D
Att = 1 - (X*X + Y*Y + Z*Z)
```

We can encode `f(X,Y)` in a 2D texture as *(X*X + Y*Y)* and `g(Z)` in a 1D texture as simply *Z*Z*. Figures 5.4.1 and 5.4.2 give examples of these two textures.

Note how the edges of the textures shown in Figure 5.4.1 are clamped to be exactly one. By adding these two functions together via multitexturing, we can com-

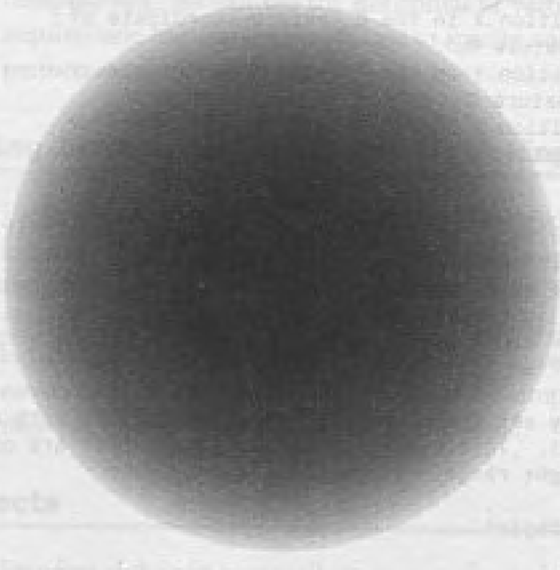

FIGURE 5.4.1. *f(X, Y) = (X*X + Y*Y).*

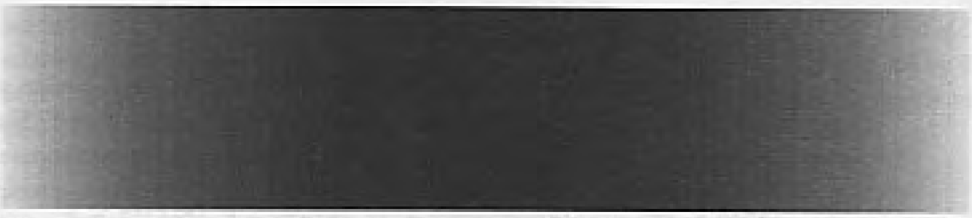
FIGURE 5.4.2. $g(Z) = Z*Z$.

pute $D*D$ on a per-pixel basis. Using inverse blending, we can use the alpha-blending unit to finish the computation and compute $1 - D*D$.

So, one procedure for implementing point lights with attenuation maps is as follows:

```
Draw ambient light and/or global illumination in the scene

For each Point Light in the scene {
        For each Object that is approximately near or within the
        Light's range {
        For each Vertex in the Object {
        Subtract the Vertex Position from the Point Light
                Position
        Scale the Position by 1 / the Point Light's Range
        Scale and Bias the result to range from 0 to 1
                for points inside the Light's Range

        Store Position.X in the S texture coordinate of
                Texture 0
        Store Position.Y in the T texture coordinate of
                Texture 0
        Store Position.Z in the S texture coordinate of
                Texture 1

        }
        Set up the multitexture hardware to choose the Light Color
        in the color unit
        Set up the multitexture hardware to compute Texture 0 +
        Texture 1 in the alpha unit
        Set up the alpha-blender to compute SrcColor * InvSrcAlpha
        + FrameBuffer

        Optionally set the alpha test to reject pixels with an
        Alpha of 1.  This avoids rendering pixels that are outside
        of the light range.

        Draw the Object
        }
}
```

Alternatively, we can use texture coordinate generation and the texture matrix to compute all per-vertex operations on the GPU:

- Set up texture coordinate generation for the first multitexture stage to give camera space position.
- Set up the texture matrix to subtract the light's *X* and *Y* position, thus giving us (*dX*, *dY*), which are stored in *S* and *T* of texture 0.
- Set up the second multitexture stage to use texture coordinate generation to give us camera space position, just like before.
- Set up the texture matrix to rotate *Z* onto the *X*-axis, and then subtract the light's *Z* position, giving us *dZ*, which is stored in *S* for texture 1.

This technique can be modified to fit a variety of light options. One option for graphics hardware or APIs (such as OpenGL) that don't allow differing color and alpha-texture blending modes is to factor the light color into the texture itself. This modification also allows other blending modes, such as multiplicative frame buffer blending, to be employed.

The attenuation function we computed is:

```
LightColor * (1 - (X*X + Y*Y + Z*Z))
```

Multiplying through by the LightColor gives:

```
LightColor - LightColor * (X*X + Y*Y + Z*Z) =
LightColor - (LightColor * (X*X + Y*Y) + LightColor * (Z*Z))
```

This implies that we need to pre-multiply the light color into both attenuation maps to get the right effect, but we really need only one map. It turns out that we can just use the center of the *LightColor * (X*X + Y*Y)* texture for the *LightColor * (Z*Z)* computation. We can use the horizontal or vertical center, but the horizontal center requires one less texture coordinate to specify and might provide better texture cache performance.

Comparing Attenuation Maps and Light Maps

Light maps are commonly employed to store static lighting data, such as shadows and light calculated through a global illumination solution. Updating light maps at run time for point lights is complicated and costly. Point lights performed with attenuation maps complement light maps nicely by taking over the chores of dynamic point lights. Instead of uploading new light maps to reflect a point light changing its color, range, or location, the nearby scene can be simply rerendered using the attenuation map textures to blend the light into the scene.

CSG Effects

By using alpha test or stencil, we can test for inclusion in perfectly spherical areas such as the falloff range of a point light. For each pixel that would be drawn with a point light, we can set the stencil to a certain value or blend into the frame buffer a constant color, thus being able to do other range-based effects.

Range-Based Fog

One application of this concept is per-pixel range-based fog. Simply render the scene as normal with no fog applied, and then render the scene with the attenuation map, treating the camera position as the "light position." Set the texture matrix to identity, and then scale the matrix by 1 over the light's range. This technique allows per-pixel, perspective-correct range fog.

The `SRC_COLOR` sent to the frame buffer blending unit should be the fog color times fog density. This gives a fog density of zero at the viewer, and at the maximum fog range the density will be one.

When rendering the fog pass, set up the alpha blender to perform `SRC_COLOR * 1 + DST_COLOR * (1 – SRC_COLOR)`.

Other Shapes

Sometimes a sphere is not what is needed; to make an oblong shape, such as a rectangle or ellipse, simply select minor and major axes and align them to the world with the texture matrix. We then have to scale the major and minor axes separately.

Conclusion

By cleverly choosing our attenuation function, we are able to perform a per-pixel spherical range calculation using two texture maps. The result of this calculation can be used for per-pixel point lights, fog, or CSG effects.