Dot Product Texture Blending and Per-Pixel Lighting

Sim Dietrich
Please send me comments/questions/suggestions
Sim.dietrich@nvidia.com

Problem Statement

Traditional per-vertex lighting requires a high-level of tessellation to achieve a smooth look. Textures are often used to simulate additional detail rather than adding more triangles to a scene or model. For instance, a polygonal character might have a smooth round head with texture-mapped facial features. Effectively the artist is attempting to give the impression of more detail than is present in the geometry. This discontinuity becomes apparent when the face is not lit properly with respect to the environment – it can't be lit properly, because lighting is applied per-vertex and the face only exists as texels from a texture map.

Textures typically are employed as lighting look-up tables. The lighting solution is pre-calculated at level build time and stored in the texture map (ie lightmap). Because the lighting equation is pre-computed, this makes it difficult to combine correctly with dynamic lights calculated at run-time.

The ideal lighting environment would involve a surface description at maximum detail which contained enough information to recalculate the lighting at runtime.

What would such a description look like? Well, dynamic lighting is typically defined per-vertex and interpolated across a triangle, so a triangle-based lighting system seems the most logical place to start.

If we are using triangles, how big should they be? Well, for maximum detail, they should be pixel-sized. Any bigger and we have sacrificed potential detail, and any smaller and we are wasting resources.

Pixel-sized Triangles

So, suppose that we could have pixel-sized triangles. Each pixel would contain an x,y,z position, diffuse and specular color material properties, one or more sets of texture coordinates and a surface normal. This would be enough information to properly compute the lighting on a per-pixel basis.

But with the traditional approach of specifying triangle vertices, it seems impossible to guarantee that a triangle was exactly pixel sized, and not bigger or smaller, so how can this be achieved? We need a greater level of detail than per-vertex, something that scales one to one with pixels. It turns out that texels meet this requirement, if they are fetched from mip-mapped textures of a high enough resolution. Each textured pixel on the screen corresponds to a small set of filtered texels. If the maximum texture resolution meets or exceeds that of the maximum screen resolution, we can achieve the goal of one pixel to one texel.

So, assuming that our graphics hardware target allows for 2048x2048 textures, we can meet this requirement for any pixel resolution up to 2048x2048 in size.

Now let's see how we can achieve a per-pixel surface description. Since we are utilizing textures for our surface description, we can use any value looked up in a texture or interpolated across a triangle. Basically, any value available in the texture combiners is fair game. For DX7, these include D3DTA_DIFFUSE, D3DTA_SPECULAR, D3DTA_TEXTURE, and D3DTA_TFACTOR. Additionally, we can complement and alpha replicate as necessary for more control.

We could use D3DTA_DIFFUSE and D3DTA_SPECULAR for the diffuse and specular material properties at this pixel, or we could look them up from a texture instead. We could use D3DTA_TFACTOR to represent ambient light intensity, or some other constant value. The position is implicitly calculated per-pixel and is not explicitly available per-pixel. Also note that there is no access to the per-vertex surface normal either. This is the key limitation we need to overcome to compute detailed lighting, and D3DTOP_DOTPRODUCT3 is the answer.

The innovation of D3DTOP_DOTPRODUCT3 is to treat a RGBA value as a 3D vector representing X, Y and Z. If we set the combiners up like so :

```
SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_DIFFUSE );
SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_DOTPRODUCT3);
SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_SPECULAR);
```

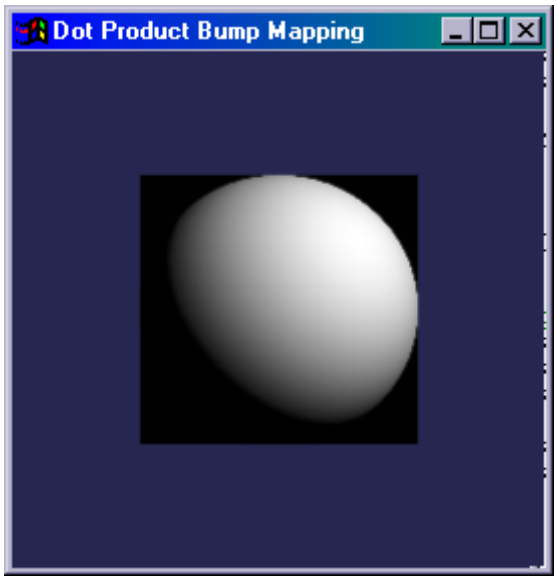
We achieve a per-pixel dot product. This can be used as part of a lighting equation. The vectors are defined per-vertex in the .color and .specular portion of the D3DVERTEX and linearly interpolated across the triangle on a per-pixel basis.

This has its uses, but fails to achieve our goal of a per-pixel surface description because we can only have surface normal discontinuities between vertices, and not on a per-pixel basis. Here is a better way to set up the combiners to achieve this end :

```
SetTextureStageState( 0, D3DTSS_COLORARG1, D3DTA_TFACTOR );
SetTextureStageState( 0, D3DTSS_COLOROP,   D3DTOP_DOTPRODUCT3);
SetTextureStageState( 0, D3DTSS_COLORARG2, D3DTA_TEXTURE);
```

Here we have asked the combiners to perform a dot product between the constant D3DTA_TFACTOR and the filtered texel for each pixel across the triangle. If we put a light direction vector into D3DTA_TFACTOR and a surface normal in the texture referenced by D3DTA_TEXTURE, then we can compute the correct per-pixel directional lighting intensity.

Here is an example of per-pixel diffuse directional lighting (note that this is only two triangles, as evidenced by the wireframe version on the right) :



What is Dot Product Texture Blending?

The D3DTOP_DOTPRODUCT3 texture blending mode was introduced in DirectX 6. The GeForce 256 supports this mode under Dx6 or Dx7 for D3D, and under OpenGL with the NV_register_combiners extension.

The D3DTOP_DOTPRODUCT3 texture blending mode is often considered to be a bump mapping technique. It certainly *can* be used for bump mapping, but more generally it can compute per-pixel lighting. This blend mode can be used to compute a per-pixel dot product of normalized vectors, which is the fundamental operation behind both the diffuse and specular lighting calculations. With some clever programming using DOT3, we can perform per-pixel lighting. Because lighting is computed on a per-pixel basis, we can achieve bump mapping as well, by brightening or darkening a pixel depending on how nearly the pixel is facing the light direction. This is an exact analogue of flat shading a triangle. A flat shaded triangle is shaded like so :

Intensity = SurfaceNormal DOT VectorTowardLight

To simulate a per-pixel flat-shaded surface, each pixel of the screen maps to a small set of texels (if mip-mapped and of a high enough resolution) that when point-sampled or averaged together yield not a color, but a normalized vector in RGB form.

What is the math behind Dot Product Texture Blending?

The DOTPRODUCT3 blend mode takes the R, G and B portions of each color argument, ranging from 0 to 255, and scales these values to lie from –1 to 1. Next, a dot product is performed, producing a scalar value. This resultant value is clamped to the range [0,1], so negative dot products map to 0. Next the value is replicated into the R,G,B and A channels. Note that clamping negative scalars to 0 is exactly what we want for computing lighting, because we don't want surfaces facing away from a light to be lit.

Here is C++ code that performs the dot product blending operation :

```
unsigned char red1 = 0x20;
unsigned char green1 = 0x60;
unsigned char blue1 = 0x20;

unsigned char red2 = 0x00;
unsigned char green2 = 0x00;
unsigned char blue2 = 0x80;

// Scale from [0, 255] to [–1.0f, 1.0f]
float theScaledRed1 = ((float)red1 – 127.5f ) / 127.5f;
float theScaledBlue1 = ((float)blue1 – 127.5f ) / 127.5f;
float theScaledGreen1 = ((float)green1 – 127.5f ) / 127.5f;

float theScaledRed2 = ((float)red2 – 127.5f ) / 127.5f;
float theScaledBlue2 = ((float)blue2 – 127.5f ) / 127.5f;
float theScaledGreen2 = ((float)green2 – 127.5f ) / 127.5f;

float theDotProduct = theScaledRed1 * theScaledRed2 +
                      theScaledGreen1 * theScaledGreen2 +
                      theScaledBlue1 * theScaledBlue2;
// Clamp <= 0.0f to 0.0f and convert to unsigned char
unsigned char temp = (unsigned char)
                         max( 0.0f,theDotProduct*255.0f));

DWORD theFinalResult = ( temp << 24 ) |( temp << 16 )|(
temp << 8 ) | temp;
```

As you can see from the sample code above, this is a significant calculation that the texture blending units in the GPU can offload from the host processor. The final scalar value is replicated into R,G, B and Alpha of the result. This scalar value represents a grayscale value that can be output to the frame buffer or used in additional texture blending calculations.

For the purposes of this document, we will simply compute this grayscale value, ignoring light or material colors. These can be easily factored in through additional textures or render passes. We will also assume directional (infinite) lights.

What Does Dot Product Texture Blending do for me?

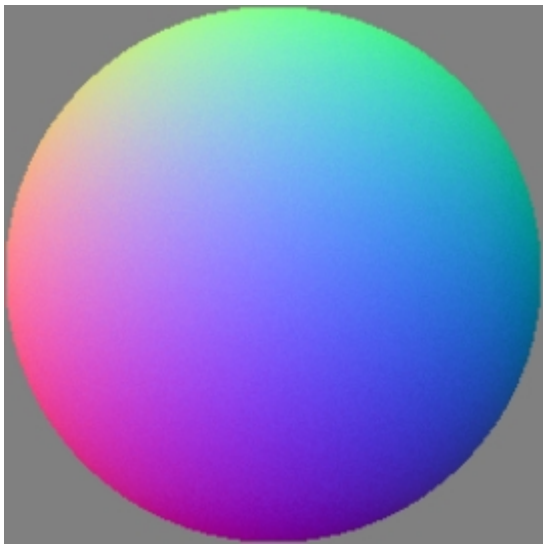
In addition to per-pixel lighting, the DOTPRODUCT3 Texture Blending can be used for bump mapping.

Many readers will be familiar with the Embossing bump mapping technique. All modern hardware supports this in one flavor or another. The downside to embossing lies in the significant per-vertex CPU cost in calculating U,V coordinates, as well as the varied vendor support for single-pass embossing techniques.

Dot product bump mapping does not require any per-vertex CPU calculations for diffuse lighting, and in many cases the specular bump mapping techniques can reduce the per-vertex CPU cost to simply a vector subtraction. Thus, dot product bump mapping can produce high-quality detail at a minimal CPU cost.

Computing Normal Maps

Dot products operate on XYZ vectors, and textures and per-vertex colors are specified with RGBA colors, so one key to the technique lies in the mapping of RGB colors to XYZ triplets described above. We could put a RGB-encoded vector at each texel in a texture, and thus give the hardware a new vector at each texture look up. Here is a procedurally generated normal map:



This particular map represents a hemisphere. Each texel on the map is an RGB-encoded vector that corresponds to a unit vector from the origin. Note the corners of the square are grey (0x00808080) which is the RGBA encoded version of the zero vector < 0,0,0 >. There is a small amount of noise generated for visual interest.

Normal maps are easy (though not especially cheap) to procedurally generate, but it would be very difficult to author them directly by painting RGB colors onto a mesh. A

better approach for authoring normal maps involves using an alpha-height field, as one would for embossing.

Each alpha value represents a height along the normal of the surface on which the normal map texture is applied. One can consider the alpha value to be a measure of displacement of the surface
Simply sample an alpha value from 3 adjacent texels in a triangular pattern. Then use the surface orientation to calculate an offset from the origin for each sample based on the alpha height. Next find the normal vector of the triangle formed by the three texel samples via a cross product. Normalize the cross product, and store as an RGB triplet back into the texture with the alpha texels. We now have an RGB-encoded normal map.

Here is an example normal map created exactly this way (best viewed in 32 bit color):

Note the blue-gray color (0x008080FF) that corresponds to < 0, 0, 1 > over the majority of the map. This normal is the same as the surface normal, thus it generates no bumps relative to the surface.

Now that we have a normal map, how do we use it to produce lighting effects?

For a directional diffuse light, we could store the vertex normal in the D3DTA_DIFFUSE component at each vertex.

Next we can store the vector from the vertex to the light (the opposite of the light direction for a directional light) in the D3DTA_TFACTOR component.

Then the intensity at each vertex would be calculated as :

```
Intensity = VertexNormal DOT LightVector

Intensity = DIFFUSE DOT TFACTOR
```

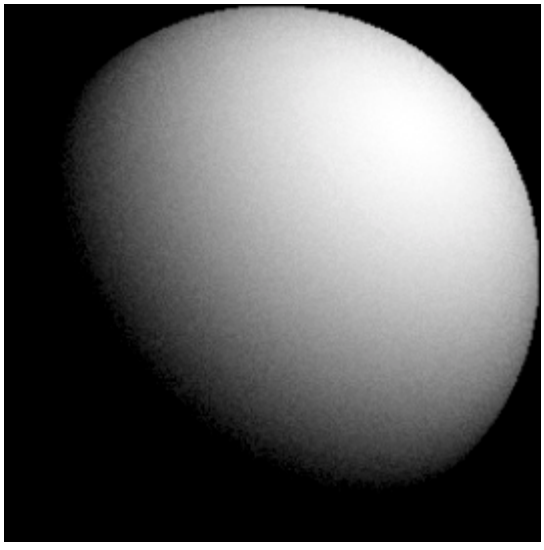
This will generate a relatively smooth shading effect when calculated per-vertex. With our normal map, however, we can calculate this equation per-pixel instead, by placing our normal map into texture unit 0.

```
Intensity = SurfaceNormal DOT LightVector

Intensity = TEXTURE0 DOT TFACTOR
```

We now have a per-pixel lighting calculation performed by the hardware.

Here is an example shot of per-pixel lighting in action. Run the DotBump example for an interactive version. Simply move the mouse to change the light direction.



What about Specular?

Specular lighting is more involved, due to the fact that it is view-dependent. This means that the direction from the vertex to the viewer must be taken into account. This is in contrast to diffuse lighting, where the view direction is irrelevant, and only the relative orientation of the surface normal and the light vector are needed.

Specular lighting can be expressed using the Blinn formulation, of :

$$Intensity = (HalfVector \bullet SurfaceNormal)^{ShinynessExponent}$$

The Half vector is the vector halfway between the vector from the vertex to the eye and the vector from the vertex to the light.

$$HalfVector = Normalize(VertexToEyeVector + LightVector)$$

For a directional light (effectively located at infinity) the light vector is constant – only the vertex to eye vector must be re-calculated per vertex.

We can perform a vector subtraction to obtain the vector from the vertex to the eye, and then use a cube map to lookup the rest of the equation. The Cube map must contain a set of 6 textures that, given a vector V, calculates H = (Normalize(V) + L). This can be generated once for each light. If the light changes direction, the cubemap can be regenerated. When cube maps are used for this purpose, they can be fairly low-res and still give good results, 32x32x6 faces works well for vector lookups, as in this example.

We would perform the vector subtraction to calculate the VertexToEyeVector, and place its X,Y,Z components in the texture coordinates U,V, Q used to index into our 32x32x6 sided cube map. The cube map texture lookup will calculate the HalfVector on a per-pixel basis. As the surface normal becomes more aligned to the HalfVector, the intensity will increase.

Alternatively, the D3DTSS_TCI_CAMERA_SPACE_NORMAL texture coordinate generation mode can be employed to generate this vector for us, thus avoiding computing even the vector subtraction on the CPU.

To incorporate the Shinyness exponent of 2, simply render an additional pass, with SRC*DEST alpha blending. This will allow the highlight to falloff more sharply towards its edges.

So, to calculate specular light with an exponent of 1, set up the texture combiners to perform :

```
Intensity = SurfaceNormal DOT HalfVector

Intensity = TEXTURE0 DOT TEXTURE1
```

Dealing with Coordinate Systems

When lighting is computed using vector dot products, the two vectors involved must be defined in the same coordinate system. So, to ensure this, we can either recompute the normal maps to be defined in the same coordinate system as the light vector, or we can move the light vector into the coordinate system of the normal map.

It is always cheaper to move the light vector into the same space as the normal map, since the normal map will contain more than one vector.

Case Study : Lighting a Ceiling with a Diffuse Direcitonal Light

To light a ceiling with dot product lighting, we generate a normal map corresponding to the ceiling's orientation, with a surface normal of <0, -1, 0 >. We can tile the normal map like any other texture.

Given a diffuse, directional light defined in world space, simply apply

```
Intensity = SurfaceNormal DOT LightVector

Intensity = TEXTURE0 DOT TFACTOR
```

That's all that's required for diffuse lighting world geometry. On the next page we'll look at specular…

Case Study : Lighting a Ceiling with a Specular Directional Light

Use the same normal map as above.

To compute the proper half vector, we create a cubemap that contains the half vectors for a given vertex to viewer vector.

First, move the light into the same space as the normal map. In this case we are using world space, so the light vector is fine as is.

Move the eye position into world space – again we should already have this in world space.

Create a 32x32 6 sided cube map texture that computes :

HalfVector = Normalize(VertexToEyeVector + LightVector)

For each vertex, subtract the EyePosition from the VertexPosition, giving VertexToEyeVector.

Pass in the X,Y,Z components of this vector into the cube-map's texture coordinates as U,V, and Q.

The normal map is TEXTURE0, and the cubemap is TEXTURE1.

The cubemap must be accessed with 3d texture coordinates.

```
Intensity = SurfaceNormal DOT HalfVector

Intensity = TEXTURE0 DOT TEXTURE1
```

Note that this assumes a local viewer (which is the most accurate model). For an infinite viewer, we can simply use the ViewDirection in place of the VertexToEyeVector.

Case Study : Lighting a Model with a Specular Directional Light

Use a normal map computed relative to **model space**.

To compute the proper half vector, we create a cubemap that contains the half vectors for a given vertex to viewer vector.

First, move the light into the same space as the normal map. In this case we are using model space, so the light vector must be moved through the model hierarchy until it lies in the space in which the normal map was generated.

Move the eye position into model space, again traversing the hierarchy if necessary.

Create a 32x32 6 sided cube map texture that computes :

HalfVector = Normalize(VertexToEyeVector + LightVector)

For each vertex, subtract the EyePosition from the VertexPosition, giving VertexToEyeVector.

Pass in the X,Y,Z components of this vector into the cube-map's texture coordinates as U,V, and Q. This can be accomplished via camera space position texgen.

The normal map is TEXTURE0, and the cubemap is TEXTURE1.

The cubemap must be accessed with 3d texture coordinates.

```
Intensity = SurfaceNormal DOT HalfVector

Intensity = TEXTURE0 DOT TEXTURE1
```

Note that this assumes a local viewer (which is the the most accurate model). For an infinite viewer, we can simply use the ViewDirection in place of the VertexToEyeVector.

Note that we either have to recompute the cubemap for a specular light for each model, or we can simply create a vector normalization cubemap, and compute

HalfVector = (Normalize(VertexToEyeVector) + LightVector))

on the CPU, and pass in the HalfVector as U,V,Q to obtain the normalized version of the HalfVector from the cubemap. For an infinite viewer, we can skip the CPU normalization, and compute :

HalfVector = (ViewDirection + LightVector))