Fragment level Phong illumination

Introduction

Phong illumination really isn't something new. The Phong illumination model has been around for almost three decades now. First introduced by Phong Bui-Tuong in 1975 this model is still frequently used both in the offline rendering world and the real-time graphics world. Due to the complex math behind the model it has up until recently only been used for vertex lighting in the real-time rendering world. Both the Direct3D and OpenGL illumination models closely follow the Phong model with some small variation. Doing it on a vertex level often causes visible artifacts and less than convincing look unless you use a very high tessellation. With advances like the dot3 operation in the fixed function pipeline we got a step closer to getting lighting on a per-pixel level. Unfortunately, the limitations of the fragment processing pipeline meant a lot of compromises had to be done, even in DirectX 8 level pixel shaders. With limited range of [-1,1], or the [-8,8] in PS 1.4, and with the limited precision the DirectX 8 level graphic cards offers much of the required math is simply not possible to do. Further, the fact that there's no advanced math instruction in these graphics solution is another obstacle on our way towards advanced lighting, not to mention the instruction limit. For these reasons, tricks like packing attenuation into a 3d texture, using cubemaps for normalization and using textures as lookup tables for exponentiation of the specular component has been the norm for the past generation.

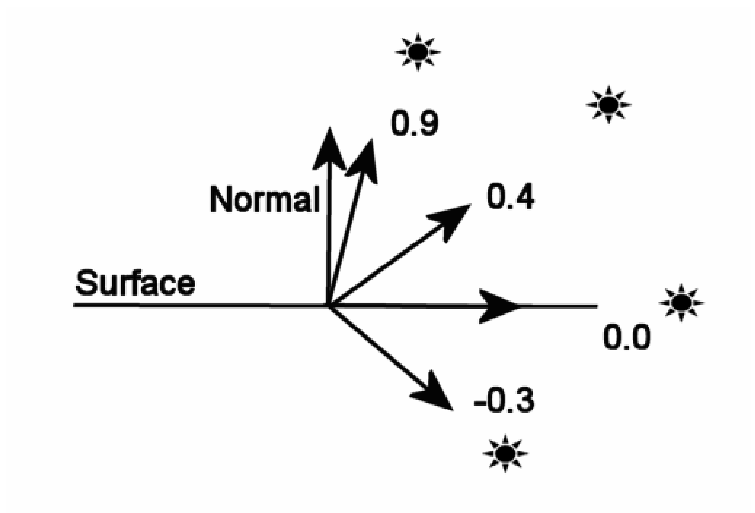
Fortunately, this will sooner or later be nothing but a bad memory of the past. With DirectX 9 level hardware we not only have the close to infinite range of floating point components and much higher precision, we are also able to do advanced math and have a whole lot of more instructions to play with before reaching the hardware limits. This means that we for the first time ever are able to truly evaluate the Phong illumination model for each pixel completely in a pixel shader. I will state however at this point that even though we are finally able to evaluate the whole Phong illumination model in the pixel shader there are still considerations and limitation that need to be addressed. The number one consideration one need to take into account is of course performance. Even with the top high-end graphics cards of today the full equation can be quite demanding on the fragment pipeline and if care is not taken performance will suffer. We'll address some of these issues later on in this article.

The Phong illumination model

Let me start by introducing the Phong illumination model:

$$I = A_{coeff} A_{color} D_{color} + \sum_i \left(Att \cdot L_{color} \left(D_{coeff} D_{color} \left(N \bullet L_i \right) + S_{coeff} S_{color} \left(R \bullet V \right)^{S_{\exp}} \right) \right)$$

So what does all this do? Let's consider every component and their purpose. The first component, I, is of course the resulting color or intensity. The other components, A, D and S represent three different attributes of light and are called ambient, diffuse and specular. We'll begin with diffuse as it's the most intuitive (though not the simplest) of these. To understand what diffuse lighting is, take a piece of paper and point a light towards it (or just imagine it in your head). The paper may represent a polygon in our little world. When the paper faces the light it'll receive a lot of light and will look bright white. Now slowly turn the paper around until the edge faces the light instead. As you can see it fades with the angle as the paper face away from the light. This phenomenon is what diffuse lighting represents. The actual math behind this is what we see in the middle of the equation above, $N \bullet L_i$. N is the normal of the surface and L_i is the light vector. The light vector is a vector that points from the point we're lighting towards the light. The light vector should be normalized, that is, being of length 1. The same should of course be true for the normal too. The dot product factor will thus be a number between -1 and 1. We don't want negative light contribution, so all dot products in this article are assumed to be clamped to the [0...1] range. Why does this expression give us the desired result? Let's illustrate it with an image:



A dot product between two perpendicular vectors will return 0, that's the case with light lying in the surface plane in the illustration above. Anything behind the surface will return a negative number and thus be clamped to 0. A light shining perpendicularly towards the surface from above will return 1 and anything lighting the surface from an angle will get a higher contribution as the light vector approaches the surface vector. Quite intuitive, but this is of course no proof of correctness. At this time it's better to spill the beans, the Phong model isn't correct. It's just an approximation of how light tend to behave but nowhere near acceptable for studying optics. However, in graphics we don't need correctness; our main concern is to please the eyes of human beings. Thus the motto is: if it looks good, then is good. Phong illumination looks good and consequently is good. So that it can't predict how photons interact with matter is not going to concerns us a whole lot.

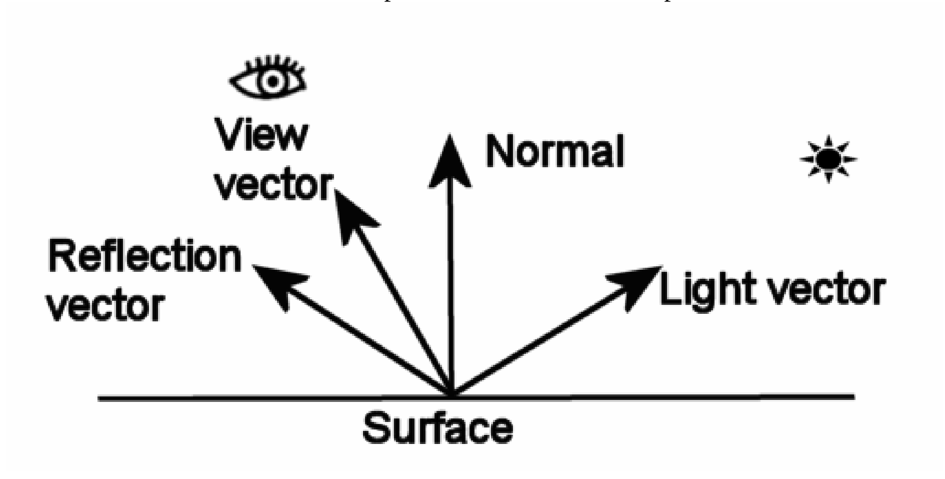
If we go back to the equation again you can see that the diffuse contribution is multiplied with two other variables, D_{coeff} and D_{color}. D_{color} is the color of the material of the surface, commonly represented by a texture or a constant color. We will use a texture and this texture is the normal base material texture as used in many applications and games and should not need any further introduction. D_{coeff} is simply a variable telling how much the diffuse component is going to contribute in whole lighting equation, you'll notice that there's also a A_{coeff} and a S_{coeff} variable which controls how much ambient and specular we want. For performance one do not necessarily need to care about all of these. In fact, it can be beneficial to just bake the D_{coeff} into the base texture. If you want less diffuse contribution you can just use a darker texture, similarly for ambient and specular. So we can do the exact same thing with that in mind and a consequently somewhat simpler expression. Here the components A, D and S have their coefficients and colors pre-baked into single entities.

$$I = AD + \sum_i \left(Att \cdot L_{color} \left(D\left(N \bullet L_i\right) + S\left(R \bullet V\right)^{S_{\exp}} \right) \right)$$

The specular component

So far we have discussed diffuse lighting only. Diffuse lighting works well for materials like wood, stone, fabric etc. But it won't work that well for materials like plastic, metal and porcelain. Why not? These materials have a property that for instance rough wood lacks, they are shiny. Shininess is the property that the specular component tries to resemble. For rough wood you could do without any shininess and it would looks pretty good. But even for rough wood a small specular component can enhance the image. There's this saying in graphics, "if you can't make it good, make it shiny". But be careful though, with Phong illumination you can make it good so you shouldn't need to resort to making it overly shiny. Unless you're trying to make it look like polished wood you should use a low specular component. The best images are created by carefully balancing specular and diffuse properly according to the properties of these materials in real life.

So how is the specular component calculated? The idea is similar to that of the diffuse component. To begin with, compute the dot product between the reflection vector R and the view vector V. The view vector is similar to the light vector, except that the view vector is the vector from the camera to the lit point rather than from the light to that said point. This reveals a significant property of specular lighting. While diffuse lighting is viewpoint independent specular is by its pure nature very viewpoint dependent. If you navigate around in your little computer graphics world looking it doesn't matter from where you observe a piece of rough wood, it'll look the same regardless where you view it from. That's not true for materials like plastics though, as you move around you'll see the reflection of the light in the surface and as the viewpoint changes the reflection will move around. To illustrate the behavior of specular take a look at this picture.



You of course get maximum reflected light if you view the surface from somewhere along the reflection vector. As you move away from this vector you see less and less reflected light. If you were to use the dot product of the view vector and reflection vector the surface still wouldn't look particularly shiny, rather it would just look bleach. Why is that? Think of a perfectly reflecting surface, a mirror in other words. A mirror will only reflect light in the exact direction of the reflection vector. That is, if you viewed it at a slight angle off from the ideal reflection angle as in the picture above you wouldn't see any reflected light at all. Thus in that case the dot product alone obviously doesn't work. Also think of a dull surface. It will reflect light in all possible directions so reflections should be visible from pretty much everywhere, even though you don't see a sharp reflection but rather just a uniformly lit surface. The difference is of course the spread. The mirror doesn't have any spread while the dull material has a significant spread. In other words, the more reflecting the material is the faster the light falls off as you move away from the reflection vector. Enter the specular exponent. As you can see in the Phong equation the dot product of the view vector and the reflection vector is raised to a power. This exponent represents the shininess of the material. The higher the exponent the shinier the material is. A specular exponent of infinity is mirror and a specular exponent of 0 is completely dull surface where light is spread equally in all directions. If you didn't raise the specular to a power, basically using a specular exponent of 1, you still have a pretty dull surface. Normal values of the specular exponent tend to be around 8 – 64. We will use a constant specular exponent of 24, something I choose because it looks pretty good, remember, it if looks good then it is good. With pixel shaders v2.0 nothing really prevents us from changing the shininess of the surface by storing the exponent in a texture and use that as a lookup table for specular exponents for each pixel. This can be used to let rusty parts of metal be non-shining while letting the intact parts shine as appropriate. A dark region in this texture represents a non-shiny area while bright regions are those that are shiny. I'll leave this as an exercise for the interested however and instead focus on a more important part, by which we can create a quite similar effect, namely gloss.

Gloss is basically just another word for the specular coefficient. As you probably remember we baked the coefficients together with the colors for each of the components, ambient, diffuse and specular. One often leaves the specular color as white which basically reduces the S component to be nothing but the specular coefficient or the gloss. This is because most shiny materials don't significantly change the color of the light as it reflects off the surface. Some material does though and if you're going to simulate this behavior you should of course keep the specular color component. Gloss is an important part of the equation however and should generally be left in the equation. It often gives better results to just alter the gloss instead of the specular component across a surface to do effects like rusty parts of a metal surface as mentioned above. So we will use a texture containing the gloss, a so called gloss map. If you want to use a specular color you can bake it into the gloss map, but in our case we will take advantage of the fact that we only have a single property to take care of and use a single channel texture to store our gloss map, which reduces the bandwidth need.

Attenuation

In real life light will fade as the lit surface get farther from the light. The falloff is a roughly a $1 / r^2$ function (think of the area of a sphere with the light in its center). In real life light sources aren't really a dimensionless point in space either. A light bulb for instance is while not particularly large still not exactly infinitesimal either.

So if we applied an attenuation factor of $1 / r^2$ we wouldn't get very realistic results. To better capture the behavior of light a slightly more complex function is commonly used.

$$\mathrm{Att} = \frac{1}{c + l \cdot r + q \cdot r^2}$$

We have constant, linear and quadratic attenuation, that's c, l and q in the formula above. It's not necessary to use all components; instead I usually drop the linear component since it doesn't add a whole lot and is the one that places the heaviest load on the fragment pipeline since it requires a square root. Usually it's enough to just offset the inverse square function with a constant and usually setting this constant to 1 will suit us well. So the attenuation function we will use is

$$\mathrm{Att} = \frac{1}{1 + q \cdot r^2}$$

Ambient

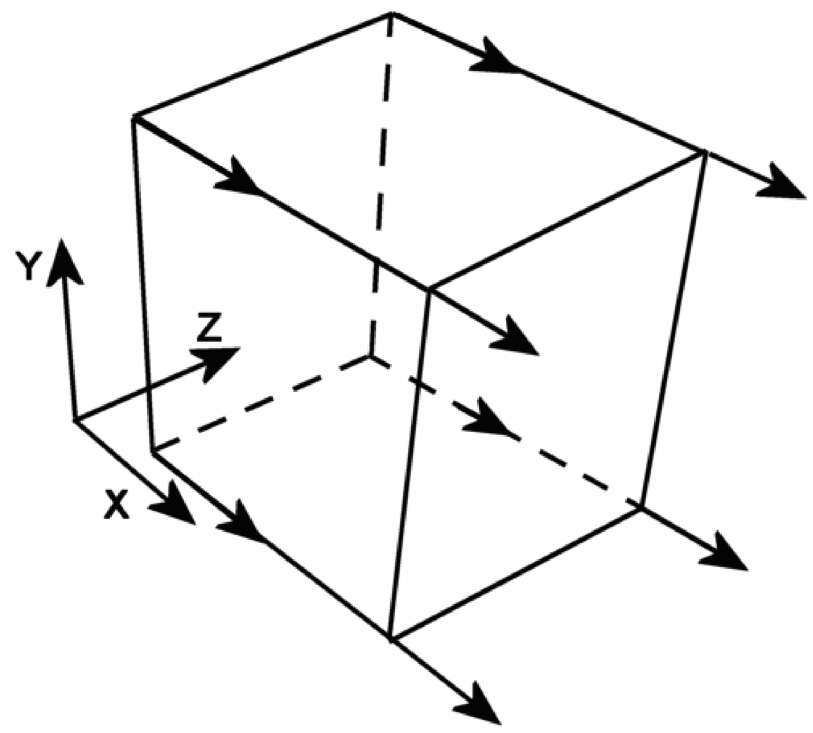
If you were to implement the lighting equation as we have gone through so far we would get quite good results. However, there's still something that will hurt the impression of reality. Polygons in our little virtual world that faces away from all our light will be black. This may sound natural as no light would hit it. However, our experience tells us otherwise. If you're in a decently lit room you'll have a hard time finding a surface that's so dark that you can't see its details and texture. Nothing really gets black. Why is that? When light hits a surface some of it scatters back. Some of that light hits our eyes which is the sole reason we can see anything at all. Not every photon scattering off from a surface will hit the eyes of the viewer though, some will bounce away and hit other surfaces. Some of that light will then once again scatter back into the scene. This is called indirect lighting and is something that unfortunately our Phong model doesn't take care of. Fortunately, there's a very cheap way to fake it though. Enter the ambient component. While none of the components of Phong illumination is particularly real or physically correct the ambient is the most fake of them all. In fact, it clearly goes against all our knowledge about light. But as always, if it looks good then it is good. Ambient gets rid of the blackness of unlit surfaces and gives a decent impression that indirect light is present in the scene. This alone is a noble enough goal to motivate its place in the Phong model and given how cheap it is to implement one would really rather instead have to motivate your stance if you were not going to use ambient.

So what is ambient then? Basically it's nothing but a constant light that hits every surface. One assumes that the light scattered off from the surfaces in the scene is uniformly distributed in all directions and all places. This is hardly close to the truth though, but works reasonably well for most normal scenes. With light hitting a surface uniformly from all directions you get no reflective kind of behavior ala specular. It's also completely angle independent so anything like diffuse is out of the window too. Basically you end up just the texture color multiplied with a constant of how much ambient you want in the scene; very simple but quite effective. For being so effective and yet so cheap it's easily the most worthwhile calculation your fragment shader can do.

Fragment level evaluation

In real life few surfaces are really flat; this is a painful truth for the graphics artist as it becomes so much harder to create realistic environments given the base primitive of 3D graphics. However, there are solutions and Phong illumination on a fragment level gives you opportunities to ease the burden on the artist without the need for zillions of tiny triangles to simulate rough surfaces. Also, it would be wasteful to do all this work on every pixel without taking advantage of the possibilities this gives you. One could for instance just interpolate the normals and evaluate the Phong equation on each pixel. While this would certainly look better than normal per vertex lighting it would still look flat. Fortunately, the Phong illumination model still has headroom to improve this significantly. The solution is of course instead of just interpolating the normals to store them in a texture and look them up on a per pixel level. This is what's commonly called a normal map, or bump map. This will let you give surfaces properties that real surfaces tend to have like roughness, bumpiness and fine details. However, this introduces some important issues and the full concept can be a significant threshold for many people to get over. We'll take it from the beginning though and will study the issues it raises in detail.

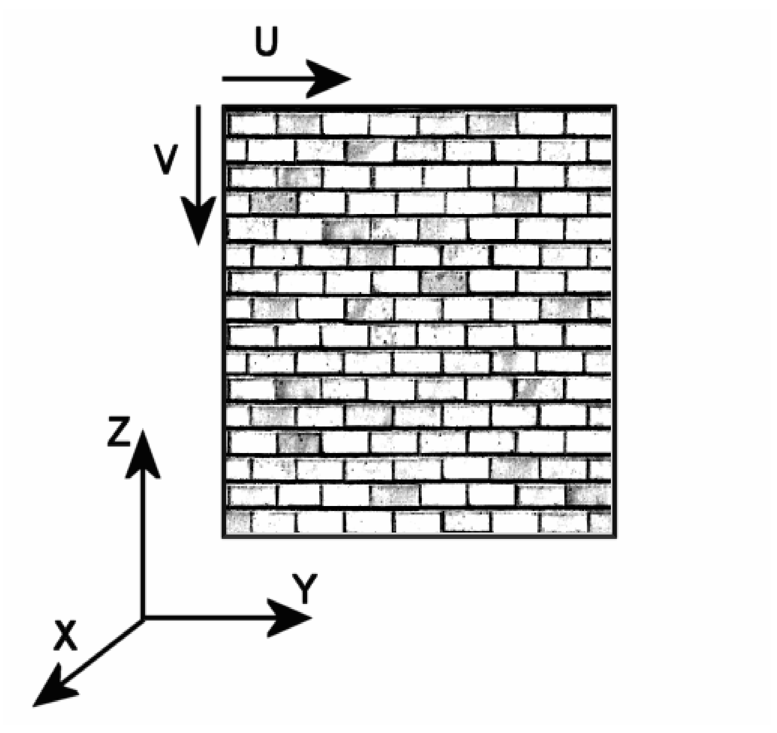
So let's assume that we have a texture with the normals stored. We sample this texture in our pixel shader and do the entire math. Will this work? Those who have tried (for instance me before I understood these issues) can assure you that it'll look very odd and incorrect. It'll look ok at some spots but wrong in most other. Well, if the normal map was created exactly for the given direction of a polygon it would work. We can't create a separate normal map for every direction a texture may be located in our scene. Not only would our dear artist refuse to take on this tremendous job, but even if he did we would bring the graphic card to its knees due to the extreme memory requirements. So this is obviously not an option. Ideally we would want a base texture, a normal map and a gloss map to go together for each material. This is the solution I'll come to in the end, so why do I insist that just sampling a texture and do the math requires a separate texture for each given possible direction? Consider a simplest example possible: You are inside a cube. All six faces use the same texture and the same normal map. Now assume we want them all to look flat, so we store a constant normal, say for instance (1, 0, 0) in the normal map. Now applying this to all six faces we'll get something like in the illustration. Of course you'd want the normals to into the box, the faces of the cube obviously have different normals and in this case only one face has correct normals. It may seam impossible at first that the faces can share the same normal map given that they are oriented differently and have very different normals. Using a separate normal map may seam to be the only solution. Fortunately, there's a better solution.



Tangent space

To solve the problem we need to introduce the concept of a vector space. Imagine that we removed the axis pointers in the picture above. How would we know which direction is the X direction? We just wouldn't know! Why? Because the direction of X is nothing but an arbitrary choice we have made. There's no fundamental truth behind this choice. It's just a choice as good as any. Imagine that we put X into Y's position and vice versa. Suddenly the (1, 0, 0) normal would be incorrect for face that it was correct for before. Not only that, but suddenly it's correct for the face in the bottom of the cube. Now imagine that we used different meanings of X, Y and Z for each face. What would that imply? (1, 0, 0) can be the correct normal for every face, we only need to adjust our coordinate system to suit the normals. This may seam backwards but it is an extremely handy thing to do in graphics.

A vector space is basically just a coordinate system. You have three vectors defining the direction of each major axis. These are the vectors pointing in X, Y and Z direction as defined by that vector space. There are two vector spaces that are important to us right now. First, the standard vector space we place all our objects into. This is called world space. This is the vector space you've been using even though you may not have realized that. As we place our objects in absolute coordinates the world space is defined by the vectors (1, 0, 0), (0, 1, 0), (0, 0, 1). The other space that's important to us is the so called tangent space. It is defined by the tangent vectors and the surface normal. Note that we still need the surface normal even though we have normals stored in the normal map. The difference though is that the surface normal is a normal normal (no pun intended), i.e. it's defined in world space. The normal map however contains normals in tangent space. To better understand the concept of tangent spaces try to think of a texture quad. The vectors that define this vector space are the ones that point in the direction of the u and v texture coordinate in world space. The normal points perpendicularly right up from the surface as usual. The picture at the side may help you understand the concept. The tangent space in this picture it thus defined by (0, 1, 0), (0, 0, -1) and (1, 0, 0), since the direction of the U texture coordinate points in the Y direction and V points in the -Z direction and the normal points in X direction.



Alright, now that we have our tangent space, what's up next? Well, we'll need to store the tangent space for each vertex in our geometry and pass that along with the vertex and texture coordinates to the vertex shader. The vertex shader needs to perform transform the light vector and reflection vector into tangent space and pass that along to the pixel shader. The pixel shader can then work as usual and use the normal from the normal map.

An obvious question at this point is of course, how do we create the normal map? Unfortunately there's no general method for creating a normal map from a base texture. Instead the artist needs to create the normal map along with the base texture as two separate but obviously connected entities. It's quite unintuitive to draw a normal map however; a height map is much intuitive. It's easier to think of white as high and black as low than it is to think of pink as pointing to the right and light green pointing down etc. Fortunately, there's a general way of converting a height map into a normal map which can also be done on load time. All you need to do is applying a sobel filter to every pixel. Alright, great, another technical term to learn I suppose many of you are thinking. The concept of a sobel filter is quite simple though. A sobel filter basically finds the slope of a grayscale picture. First you apply the sobel filter in X direction, then in Y direction and form the the vector (dX, dY, 1). Then normalize this vector and you're done. The filter kernels look like this:

-1	0	1
-2	0	2
-1	0	1

-1	-2	-1
0	0	0
1	2	1

If you're unfamiliar with the concept for filter kernels, just place the pixel you're filtering right now in the middle square. Then multiply each pixel that each square covers with the number that's in that square and sum it all together. The result is your filtered value. So applying the left filter will give you dX and the right one will give you dY.

Implementation

If you've read everything to this point I suppose you are getting a little tired of all the theory. So without further ado we'll dive straight into the implementation. The first thing we need to define is our vertex format. As we've concluded earlier in this text the data we need is a vertex position, a texture coordinate and our tangent space. This gives us this vertex format:

```
struct TexVertex {
	Vertex vertex;
	float s, t;
	Vertex uVec, vVec, normal;
};
```

Now we need to feed this info into the vertex shader. Feeding the vertex and texture coordinates into a vertex shader should be pretty much straightforward. It's important to note at this time though that texture coordinates no longer need to be in any way related to textures. They are really nothing but generic interpolated properties. So we will feed info into the vertex shader through texture coordinates and then pass new texture coordinates from the vertex shader into the pixel shader. So the vertex declaration will look like this:

```
D3DVERTEXELEMENT9 texVertexFormat[] = {
  { 0, 0,                                          D3DDECLTYPE_FLOAT3,
D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_POSITION, 0},
  { 0, 1 * sizeof(Vertex),                         D3DDECLTYPE_FLOAT2,
D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 0},
  { 0, 1 * sizeof(Vertex) + 2 * sizeof(float), D3DDECLTYPE_FLOAT3,
D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 1},
  { 0, 2 * sizeof(Vertex) + 2 * sizeof(float), D3DDECLTYPE_FLOAT3,
D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 2},
  { 0, 3 * sizeof(Vertex) + 2 * sizeof(float), D3DDECLTYPE_FLOAT3,
D3DDECLMETHOD_DEFAULT, D3DDECLUSAGE_TEXCOORD, 3},
  D3DDECL_END()
};
```

The vertex shader needs to compute the light vector and the view vector from the provided data. Thus we'll need to provide the vertex shader with the camera position and light position. This is best done with vertex shader constant as these attributes doesn't change with the geometry in any way. Once the view and light vectors are done we need to transform them into tangent space. The transformation is just a matrix multiplication, which by the way is nothing but a set of dot products. As these are three-dimensional properties we need only do a dp3 operation with each of uVec, tVec and the normal. The resulting vertex shader ends up as something like this:

```
  vs.2.0

  dcl_position  v0
  dcl_texcoord0 v1   // TexCoord
  dcl_texcoord1 v2   // uVec
  dcl_texcoord2 v3   // vVec
  dcl_texcoord3 v4   // normal

  // c0-c3 = mvp matrix
  // c4    = camera position
  // c5    = light position

  // Transform position
  m4x4     oPos, v0, c0

  // Output texcoord
  mov      oT0, v1

  sub      r0, c5, v0         // r0 = light vector
  dp3      oT1.x, r0, v2
  dp3      oT1.y, r0, v3
  dp3      oT1.z, r0, v4      // oT1 = light vector in tangent space

  sub      r1, c4, v0         // r1 = view vector
  dp3      oT2.x, r1, v2
  dp3      oT2.y, r1, v3
  dp3      oT2.z, r1, v4      // oT2 = view vector in tangent space
```

Alright, everything should now be properly setup for the most important piece of code of ours, the pixel shader, which will do all the tough work. As everything is now in tangent space we can just carry on all operations just as if all data, including the normal from the normal map, would have been in world space. The pixel shader will be much longer so we'll so through it step by step instead of just printing the entire code right here. We'll start with the diffuse component.

```
  ps.2.0

  dcl t0.xy
  dcl t1
```

```
dcl_2d s0
dcl_2d s1

def c0, 2.0, 1.0, 0.0, 0.0  // (2.0, 1.0, unused ...)

texld    r0, t0, s0               // r0 = base
texld    r1, t0, s1               // r1 = bump

mad      r1.xyz, r1, c0.x, -c0.y  // bump[0..1] => bump[-1..1]

dp3      r7.w, r1, r1
rsq      r7.w, r7.w
mul      r1.xyz, r1, r7.w         // r1 = post-filter normalized bumpmap

dp3      r7, t1, t1
rsq      r7.w, r7.x
mul      r3, t1, r7.w             // r3 = normalized light vector

dp3_sat  r4, r3, r1               // r4 = diffuse

mul      r4, r4, r0               // r4 = base * diffuse
mov      oC0, r4
```

Alright, this should be pretty straightforward. We begin by sampling our base texture and grabbing the normal from the bump map. We could of course have used floating point textures given that normals can have components which range from -1 to 1, but that would reduce performance without a whole lot of image quality improvement. Actually, it would reduce the image quality on current hardware since at the time of this writing no hardware is available that supports filtering on floating point textures. So, instead we take the traditional approach of packing it into a normal D3DFMT_X8R8G8B8 texture. This means we will have to unpack it in our shader though and that's the mad (as in multiply and add, not crazy) instruction right after the sampling. Note that the linear filter on the normal map isn't really that suitable for normals, so after the filtering the normal may no longer be of unit length but rather slightly shorter. This may not matter a whole lot for diffuse, but it will matter quite a lot for specular. If the length is 0.99 instead of 1.0 and you raise it to say 24 it'll not end up with the wanted 1.0 but rather something much lower, $0.99^{24} = 0.785$, which will make our specular highlights significantly less sharp. So the post-filter normalization is certainly needed, even though maybe not this early, but it doesn't hurt to use a better normal for diffuse too. The normalization process is quite simple. As you may remember from your linear algebra lectures a vector dot-multiplied with itself is the squared length of that vector. So what we do is to take the inverse square root of that squared length, which gives us the inverse of the length. Multiplying the vector with the inverse length and the normalization is done. The same is then done to the light vector. After the normalizations we can just do the dot product between these vectors, multiply that with the base texture and our diffuse is done. Note that we use dp3_sat as opposed to just dp3. This is so that all negative dot products get clamped to zero. We don't want negative light remember?

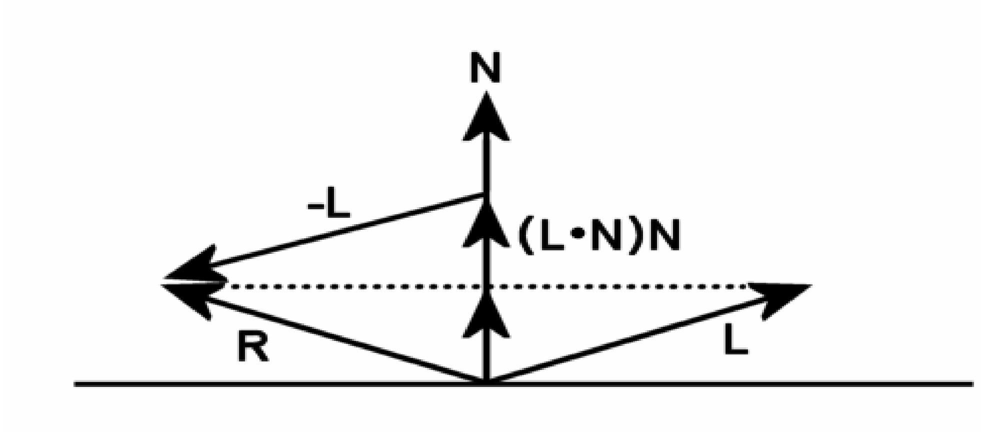
So far the output doesn't look particularly impressive. The most obvious drawback is the lack of attenuation. So far only the angle matters, not how far from the light the surface is. We'll remedy the problem right away. So we'll need this piece of code inserted right after the light vector normalization.

```
// c2 = (constant attenuation, quadratic attenuation, unused ...)
mad      r5, c2.y, r7, c2.x
rcp      r5, r5.x                 // r5 = attenuation
```

This will give us our wanted attenuation factor which we can multiply with our diffuse to get a properly attenuated light. So the last step in the shader changes as follows.

```
mul      r4, r4, r0               // r4 = base * diffuse
mul      r4, r4, r5               // r4 = base * diffuse * attenuation
mov      oC0, r4
```

Next up is our specular. To begin with we'll need to sample our gloss map. It has the same texture coordinates as the base texture and normal map so it's straightforward to add. As you may remember from our vertex shader above we get our view vector in t2. So we'll normalize as we did with the light vector. We then need to compute the reflection vector. The reflection vector is given by $2(L \bullet N)N - L$ as illustrated by the image below.



Once the reflection vector is done we basically just need to do the dot product, raise it to a power and multiply with the gloss and we're done. We'll add the specular exponent to the first constant. The code ends up something like this:

```
dcl t2
dcl_2d s2
...

def c0, 2.0, 1.0, 24.0, 0.0         // (2.0, 1.0, specular exponent, 0.0)
...

texld    r2, t0, s2                 // r2 = gloss
...

dp3      r7, t2, t2
rsq      r7.w, r7.x
mul      r6, t2, r7.w               // r6 = normalized view vector

dp3      r7, r3, r1
mul      r7, r7, c0.x
mad      r3, r7, r1, -r3            // r3 = reflection vector

dp3_sat  r3, r3, r6
pow      r3, r3.x, c0.z             // r3 = specular
mul      r3, r3, r2                 // r3 = specular * gloss
```

Given the discussion above there shouldn't be a whole lot of question marks over this code. Now we just need to combine it with the diffuse component. The last piece of code ends up as follows.

```
mad      r4, r4, r0, r3          // r4 = base * diffuse + specular * gloss
mul      r4, r4, r5              // r4 *= attenuation
mov      oC0, r4
```

The last piece of the equation that remains is now the ambient, which is also the simplest to implement. So without further ado we'll go right at the task. We'll need to pass the ambient factor to the shader. There are some unused components in our c2 constant, so we'll just use one of these. Then we only need to squeeze another instruction into the final combining code.

```
// c2 = (constant attenuation, quadratic attenuation, ambient, unused)
mad      r4, r4, r0, r3          // r4 = base * diffuse + specular * gloss
mul      r4, r4, r5              // r4 *= attenuation
mad      r4, r0, c2.z, r4        // r4 += base * ambient
mov      oC0, r4
```

Yes, that's it. The Phong model is now completed and ready for some serious action.

Aliasing

While we already get pretty good results there is still a couple of issue that needs to be addressed however. One such issue is aliasing. You probably already know the reasons why we use techniques like mipmapping. If you don't have the mathematical background you probably at least know from experience that not using mipmapping will cause severe shimmering artifacts on objects at a distance. Why is that? The mathematical explanation is that it violates the Nyquist-frequency. Now that probably sound like Greek to most people and only those with a signal processing background will be familiar with it. Basically we are stating that the frequency present in the texture is higher than half the sampling rate, which may only confuse you more, but it's actually a quite easy concept to understand even though it would take a higher degree of mathematical skills to do the reasoning from a mathematical point of view. Assume we are rendering to a resolution of 256x256, a resolution that will hardly ever be used in real life, but for this example it makes it easy to understand the issues. Assume we also have a texture of a 256x256 containing a checkerboard pattern, that is, every other pixel is black and white. Ignoring that we usually have linear filtering it would seam that mapping this texture onto the full screen will work just fine. Every other pixel gets black and white. Now assume we map it to the upper left 128x128 pixels. Only every other pixel from the texture will end up on screen (still ignoring filters), so we get only the black pixels by a seemingly unfortunate bad luck. Obviously information got lost in the process. It's hard to get something useful in this situation either way, but at least we would want all pixels in the texture to contribute to the final results producing some kind of grey. Alright you say and point out that this is exactly what a linear filter will do for us. True, in this case using a linear filter would be enough, but then consider another checkerboard texture but with each 2x2 pixels being either white or black. Mapping this to either 256x256 or 128x128 will work just fine. Now map it to 64x64 and consider the results. Now we're back in the same situation again. We will get nothing but black as nothing from the white 2x2 pixel blocks will ever be touched by the linear filter. Obviously information once again got lost. Ideally we would want every 4x4 block in the texture to contribute to each pixel. This is basically what mipmapping does. It tries to match the pixel and texel rates by using smaller down-sampled textures to better fit the spacing between where in the texture each pixel would sample. So when mapping to a 256x256 pixel area the full 256x256 mipmap would be used, while when mapping it to a 64x64 pixel area it would use a 64x64 mipmap. For anything in between it would interpolate between the two closest mipmap levels for a smooth transition. Doing this should effectively get rid of all kind of texture shimmer artifacts related to texture sampling.

Alright, so what's up with all this theory, the problem is solved, right? Well, I'd love that to be true. Unfortunately it's not. During the DirectX 7 era one could pretty much state that it was a solved problem, but with the pixel shaders of today we are basically back on square one again. Why? Well, during the DirectX 7 era textures were combined with simple arithmetic operations like modulating a base texture with a lightmap, possibly adding an environment map onto that. Simple arithmetic operations like multiplications and additions don't change the frequency properties of the texture. So as long as you use these simple operations you'll be fine. Unfortunately this is not the case with operations like dot products. It basically kicks all the assumptions from the reasoning behind mipmapping out of the window. This means that we'll once again see shimmering. And since the trend is that multisampling replaces supersampling as the preferred anti-aliasing technique we won't get any help there either. The situation is however not as horrible as it may first appear. We just need to be aware of the problem and carefully tackle it. While mipmapping may no longer perfectly match our source it certainly helps us a lot. Again, what's the reason for shimmering? There are too high frequencies in the source material. What can we do about it? Reduce the high frequency components in our textures, or in plain English, use blurrier textures. Important to note here though is that there's no need to use a blurrier base texture since it will only be part of simple arithmetic operations. Our main target is instead our normal map and to some extent the gloss map. The general advice is to avoid having sharp contrasts in the normal map. You also don't necessarily need to use the whole 0 to 1 range when creating your height map. Sharp contrasts in the gloss map is generally not desired either. Smoother transitions in the gloss map can help hiding the aliasing artifacts slightly. It's also noteworthy that a high specular exponent while giving sharper and generally better looking specular highlights also adds to the aliasing; so these two factors needs to be balanced. A good advice is to use a blurrier normal map the higher the specular exponent is. That is, a shiny surface will need a blurrier normal map. Aliasing certainly occurs from diffuse too, so you can't use too sharp normal maps for dull surfaces either. It's also important to note that the artifacts tend to occur on lower mipmap levels, so it may help to not only just down-sample the previous mipmap level when creating the mipmap chain, but also apply a soft blur filter.

If you work for a game or content creation company it's important that you make sure the artist understand these issues. Unlike many other issues that can be handled graciously by the programmer this will require awareness from the artists. The best thing the programmer can do is to educate the artist and provide good tools for previewing the material.

Shadows

There’s one thing left that seriously hurts the impression of reality, and that’s the lack of shadows. It would be wasteful to spend all this time implementing Phong illumination and leave it in this state. There are several shadowing techniques to choose from and some of them exist in several different forms. Unfortunately, they all suck in one way or another. The two most common are stencil shadows and shadow mapping. The advantage of stencil shadows is that the shadows are pixel accurate and that stenciling is widely supported. The disadvantage is that it’s slow, not particularly scalable, hard to implement, not very general and may interfere with some anti-aliasing techniques. The advantage of shadow mapping is that it’s reasonably fast, quite scaleable, easy to implement and very general. The disadvantage is that the shadows are prone to aliasing. It has enough pluses though to make it my shadow technique of choice.

The idea behind shadow mapping is simple. In the first pass you render the distance to the light into a texture from the lights point of view. Then in the second pass you check the distance to the light against what’s stored in the texture from pass 1. If the distance is larger than the stored value obviously some other object is in the same line of view that covers the light, which implies that it’s in shadow. Otherwise it’s lit. Quite simple, isn’t it? In our case we use omni-directional lights, so we’ll need to render to a cubemap instead of a normal texture. As we’re only interested in distance and not colors etc we can use a much simpler pass. No textures, just plain geometry. For that we’ll need a pair of simple shaders.

```
vs.2.0

dcl_position  v0
// c0-c3 = mvp matrix
// c5    = light position

// Transform position
m4x4     oPos, v0, c0

sub      oT0, c5, v0       // oT0 = light vector
```

It can’t be simpler, just compute the light vector. No tangent spaces or anything, just a subtraction and we’re done. The pixel shader isn’t any more complex.

```
ps.2.0

dcl t0

dp3      r0, t0, t0
mov      oC0, r0
```

The dot product with itself gives the squared length of the light vector. Normally one would compare the distances, but the squared distances works just as well and gives a significant speed boost. There is an issue though we need to take care of for this to work well. When comparing with the stored distance there will unavoidably be precision errors due to the finite resolution of our shadow map and limited number of bits. For this reason you need to bias the distance to give some headroom for precision errors. Normally you would just add a small number. However, if we’re using the squared distances this won’t work very well due to the non-linear spacing we have. It would effectively make our bias smaller and smaller with distance and artifacts would soon be visible. If we use a larger bias we would instead get problems with missing shadows close to the light. Unfortunately, there’s no optimal bias in between either, rather we could find biases which causes both artifacts. Instead we’ll take a different approach. We’ll just multiply the distance with a constant slightly less than 1. This will instead define the allowed error in terms of a certain percentage, which will work much better. Only very close up on the light will there be artifacts. If this is a case that matters much to you there’s still the option to use linear distance rather than the squared distance, but at a performance cost of course.

Note that squared distances will return quite large numbers, certainly larger than 1 in general unless we use a very small world, so we’ll need a floating point texture to store it to. We could use a normal fixed point texture too, but then we'd need to scale it down such that we'll never get anything larger than 1. We can't allow clamping as that will destroy our shadows. Also, floating point better suits our quadratic representation of distance. So the best choice for us would to use a D3DFMT_R32F texture. Note that some pixel shader 2.0 hardware doesn't support floating point cubemaps though, but otherwise this is an ideal format as it is single channel and floating point with high precision. If you need to support such hardware you'll be better off just using the linear distance instead though.

To implement shadows we also need to change our lighting shaders. Our dear vertex shader will receive another line.

```
mov       oT3, -r0           // oT3 = shadow map
```

This line isn't obvious just by looking at it; instead you must take a look at the old vertex shader and see that r0 will since earlier computations contain the light vector, that is, the light position minus the vertex position. We want to look up in the cubemap in the direction from the light position towards the vertex position, that is, the exact opposite direction of the light vector. So that's how we come up with –r0. The pixel shader gets more extensive additions. First we need some basic setup and then we sample the shadow map.

```
dcl t3
dcl_cube s3
...

def c1, 0.97, 1.0, 0.0, 0.0 // (biasfactor, averaging factors)
...

texld     r8, t3, s3          // r8 = shadow map
```

Then right after we normalize the light vector we'll squeeze in an instruction to compute the biased distance to the light.

```
mul       r8.y, r7.x, c1.x  // r8.y = lengthSqr(light vector) * biasfactor
```

We now need to get a shadow factor, that is, 0 if we're in shadow and 1 otherwise. So we'll compare and grab a zero or one from our c1 constant depending on the outcome of the comparison.

```
sub       r8.x, r8.x, r8.y
cmp       r8.x, r8.x, c1.y, c1.z  // r8.x = shadow factor
```

Now we only need to multiply this with our diffuse and specular components. The ambient will be left alone though as we want ambient to be visible in shadowed areas too. So the component combining will changed to this.

```
mad       r4, r4, r0, r3         // r4 = base * diffuse + specular * gloss
mul       r4, r4, r5             // r4 *= attenuation
mul       r4, r4, r8.x           // r8 *= shadow factor
mad       r4, r0, c2.z, r4       // r4 += base * ambient
mov       oC0, r4
```

Tada, we have shadows! We could leave it at this and be fairly satisfied. This doesn't mean however there are no improvements left to be done. Surely enough I have another trick for you. While the shadows created with the above code looks fairly good there is a problem. If the shadow map is of low resolution, say 256x256, we will get pixilation of the shadows. The edges of the shadows have obvious stair-stepping. What can we do about it? Well, we could increase the resolution of our shadow map. This will quickly kill our performance though. Rendering to a 512x512 shadow map requires four times the fillrate of rendering to a 256x256 shadow map. Instead we'll try to anti-alias our shadows. How can we do that? By taking several samples and average them. So we'll just take the normal shadow map sampling position and add an arbitrary constant to offset it slightly and take another sample. We'll take three additional samples for a total of four to get a decent smoothing of the edges. So we'll need to provide three additional sampling positions from the vertex shader.

```
def c8,   1.0,  2.0, -1.0, 0.0
def c9,   2.0, -1.0,  1.0, 0.0
def c10, -1.0,  1.0,  2.0, 0.0
...

sub      oT4, c8,  r0
sub      oT5, c9,  r0
sub      oT6, c10, r0
```

The pixel shader gets its fair share of editions too. The changes are pretty straightforward. First we just sample at the newly provide sample positions.

```
dcl t4
dcl t5
dcl t6
...

texld    r9,  t4, s3              // r9  = shadow map
texld    r10, t5, s3              // r10 = shadow map
texld    r11, t6, s3              // r11 = shadow map
...
```

Then we'll need to revise the shadow factor calculation slightly. We'll use 0.25 instead of 1.0 for obvious reasons. We'll accumulate the results from all sample comparisons in r8.x, so the final combining code remains the same.

```
def      c1, 0.97, 0.25, 0.0, 0.0 // (biasfactor, averaging factors)
...

sub      r8.x,  r8.x,  r8.y
sub      r9.x,  r9.x,  r8.y
sub      r10.x, r10.x, r8.y
sub      r11.x, r11.x, r8.y

cmp      r8.x,  r8.x,  c1.y, c1.z
cmp      r9.x,  r9.x,  c1.y, c1.z
cmp      r10.x, r10.x, c1.y, c1.z
cmp      r11.x, r11.x, c1.y, c1.z

add      r8.x, r8.x, r9.x
add      r8.x, r8.x, r10.x
add      r8.x, r8.x, r11.x
```

And that's it. We can now view our precious creation with highest pleasure. The shadows should now look much smoother. If we make a close-up though we can still see stair-stepping, more samples would solve that. I'll leave that as an exercise for the interested.

We've come a long way, we have implemented something that at the time of this writing was hardly possible to do in real-time just a year ago. It's fascinating how far graphics technology has advanced recently, and we're still moving. As mentioned several times in this article we are still doing lots of things that are hardly real, but as technology goes forward I hope we can overcome these problems too. I hope we can join up some time in the future and implement real soft shadows, real indirect lighting and real displaced geometry instead of normal mapped simulations. See you then.