

Project6 – Lighting – per-pixel shading and optimizations for Direct3D 8

Author: Michal Valient (valient@dimension3.sk)

Conversion (with some corrections) from the HTML article written in November 2002

6.0 Abstract (2004 update)

This article starts with description of the Phong lighting equation. It continues with various implementations of this equation on the programmable graphics hardware. Some parts of this article are available in my master thesis. Also when we refer to the vertex shader we refer to the version 1.1. When we refer to the pixel shader, we refer to the version 1.4

6.1 Introduction

Light is a very important effect in the computer games. Together with the shadows it “describes” the visual atmosphere of a scene and gives us the feel of third dimension on a monitor. Computer generated 3D images will be flat, hard to orient and hard to believe-in without the lights (computed in any way and rendered in any form). Success of the computer game relies on the user's believe to what he sees. This makes lighting is very important. High quality dynamic lighting of the scenes is even more crucial with new hardware.

This article presents lighting and shading methods on the modern hardware.

Let us start with some definitions:

Material is a set of properties that describe visual appearance of a surface. The properties can include a texture, various light coefficients or a color.

We use the term **Lighting** to describe the interaction of a light and a surface (with the material applied onto it). Lighting models have to be simple enough to be computed fast in the real-time graphics and should still produce acceptable results. Phong lighting model described in this text can be divided into the three parts: ambient, diffuse and specular. Some other models are for example Cook-Torrance's model [6.2] (can be used for metallic surfaces), He's model [6.3], Oren-Nayar's model [6.4].

Shading is a process of performing the lighting computations and determining the colors of the pixels. We know three major types of shading: **flat**, **Gouraud** and **Phong**. The flat shading computes light intensity once per triangle. Intensity is then applied onto the whole triangle. This type of shading cannot produce very good results for low polygon models. On the other hand, it's fast and sometimes used in games for objects with sharp edges and flat faces. Gouraud shading (also called per-vertex shading) and Phong shading (in fact its extension used in the computer games is named per-pixel shading) will be described later in text.

6.2 Light model equation

The lighting equation (commonly used in real-time graphics) is combined from three parts - ambient, diffuse and specular in this fashion:

$$\mathbf{i}_{\text{total}} = \mathbf{i}_{\text{ambient}} + \mathbf{i}_{\text{diffuse}} + \mathbf{i}_{\text{specular}} \quad (6.1)$$

Where $\mathbf{i}_{\text{total}}$ is the final color value, $\mathbf{i}_{\text{ambient}}$ is the ambient color, $\mathbf{i}_{\text{diffuse}}$ is the diffuse color and $\mathbf{i}_{\text{specular}}$ is the specular color for given pixel

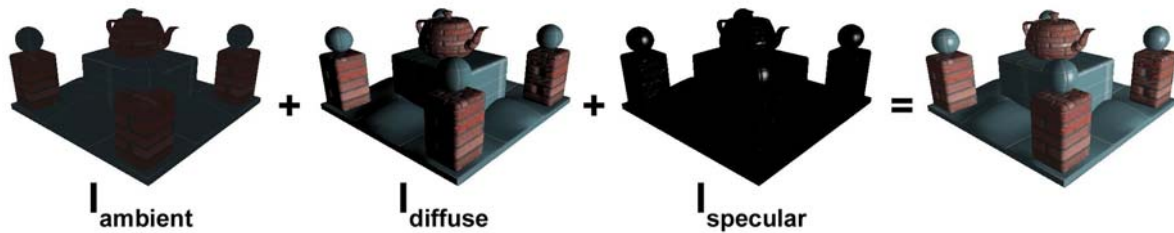


Image shows the visualization of a lighting computation.

6.2.1 Diffuse component

The diffuse component is based on a law of physics called **Lambert's law** and has the biggest base in the real world (and the interaction of photons and surfaces). The Lambert's law says that the reflected light intensity is determined by cosine of the angle \underline{a} between the surface normal \mathbf{n} and the light vector \mathbf{l} (heading from a surface point to the light) for totally matte (diffuse) surfaces. Both vectors are normalized. This lighting component is independent of viewer's position, because of the fact that for a perfectly diffuse surface the probability of a new reflection direction is equal for every direction.

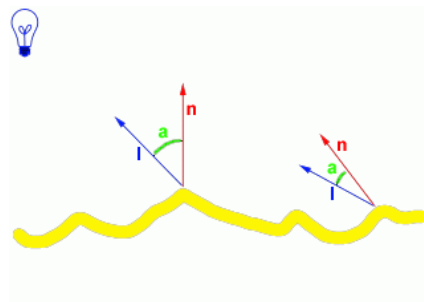


Image shows diffuse lighting intensity components.

The diffuse light intensity (i_{diffuse}) computation has the form:

$$i_{\text{diffuse}} = \cos(\underline{a}) = \mathbf{n} \cdot \mathbf{l} \quad (6.2)$$

We extend the equation 6.2 to take into account the surface color $\mathbf{m}_{\text{diffuse}}$ at current pixel, light color $\mathbf{l}_{\text{diffuse}}$ and the fact, that the angle between light and normal has to be less than $\pi/2$ to this form (normal is faced away from the light for greater angles and the pixel is not being lit and a dot product is less than zero in this case):

$$\mathbf{i}_{\text{diffuse}} = \max(\mathbf{n} \cdot \mathbf{l}, 0) * \mathbf{m}_{\text{diffuse}} * \mathbf{l}_{\text{diffuse}} \quad (6.3)$$

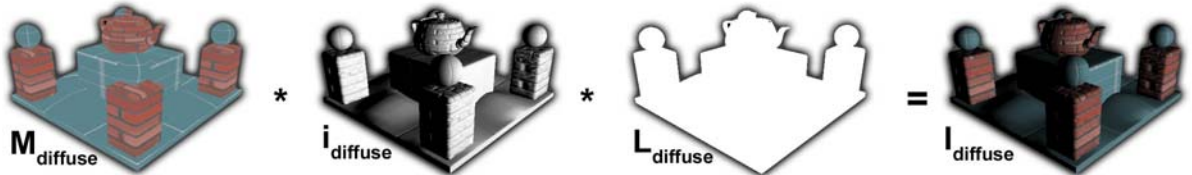
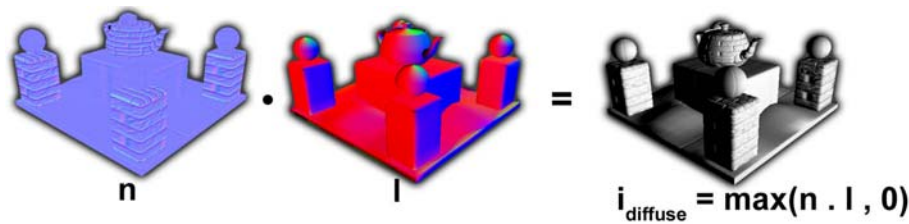


Image shows the diffuse lighting computation. The normal and light vectors are visualized with x axis assigned to the red channel y to the green channel and z to the blue channel.

Very good explanation of diffuse lighting (with a connection to the Lambert's law) can be found in [6.1] at page 71 and 72 or in [6.5] in the chapter two.

6.2.2 Specular component

We use the specular part of lighting equation to simulate shiny surfaces with the highlights. A very popular model is called the Phong reflection model [6.6] and it computes light intensity as a cosine of the angle α between the normalized light vector reflected around the point normal - \mathbf{r} - and normalized vector from the point to the position of a viewer - \mathbf{v} - powered by the *shininess* value that specifies amount of reflection. This equation has little to do with the real world illumination but it looks good under most conditions and is easy to compute.

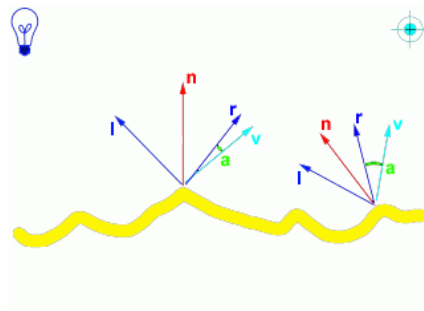


Image shows specular lighting intensity components.

Phong's specular equation then is:

$$\mathbf{r} = 2 * (\mathbf{n} \cdot \mathbf{l}) * \mathbf{n} - \mathbf{l} \tag{6.4}$$

$$i_{specular} = \cos(\underline{\alpha})^{shininess} = (\mathbf{r} \cdot \mathbf{v})^{shininess} \tag{6.5}$$

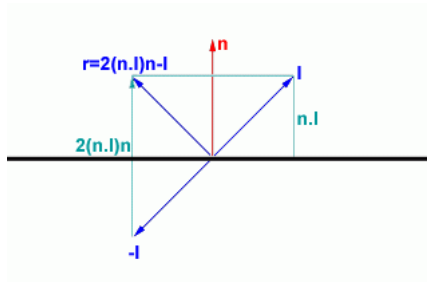


Image shows geometric interpretation of the equation 6.4.

If a dot product of \mathbf{n} and \mathbf{v} is less than zero (angle is greater than $\pi/2$) then a pixel is facing away from the light and lighting should not be computed.

We extend the equation 6.5 to take into account the material specular intensity modifier $\mathbf{m}_{\text{specular}}$ (in real-time graphics commonly called the gloss map) and the light color $\mathbf{l}_{\text{specular}}$ to this form:

$$\mathbf{i}_{\text{specular}} = (\mathbf{r} \cdot \mathbf{v})^{\text{shininess}} * \mathbf{m}_{\text{specular}} * \mathbf{l}_{\text{specular}} \quad (6.6)$$

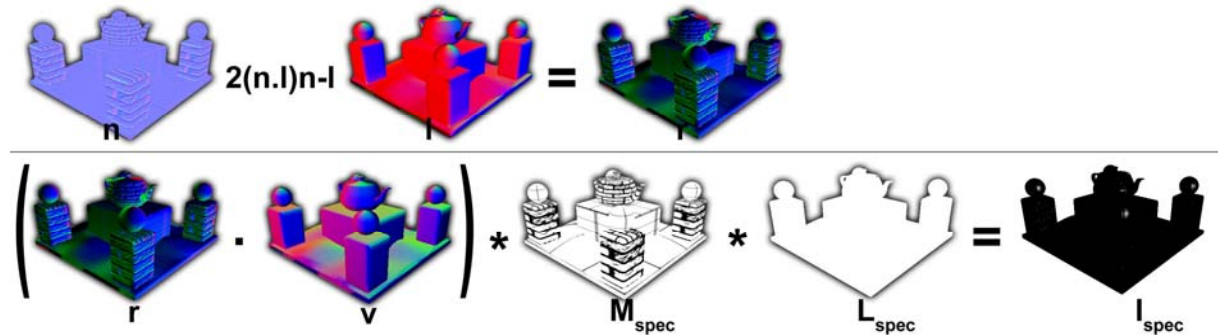


Image shows specular lighting computation. Normal, light, reflection and view vectors are visualized with x axis assigned to the red channel y to the green channel and z to the blue channel.

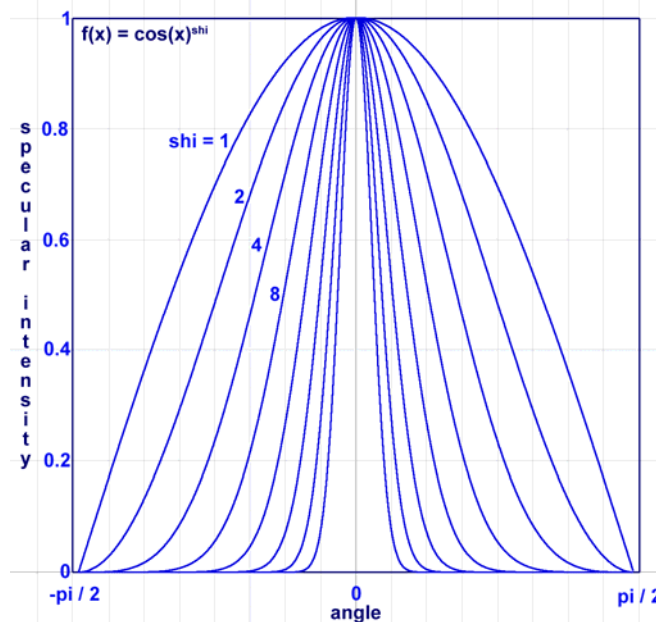


Image shows effect of the shininess parameter. The greater it is, the narrower is the area of highlight. Shininess parameter on image has values 1,2,4,8,16,32,64,128

There are several variations of Phong's equation. Commonly used is the Blinn's equation [6.7]. It uses normalized half vector \mathbf{h} between \mathbf{l} and \mathbf{v} . Resulting intensity is then a dot product of \mathbf{h} and \mathbf{n} . In general it is faster to compute it in real-time.

$$\mathbf{h} = (\mathbf{l} + \mathbf{v}) / \|\mathbf{l} + \mathbf{v}\| \quad (6.7)$$

$$i_{\text{specular}} = (\mathbf{h} \cdot \mathbf{n})^{\text{shininess}} \quad (6.8)$$

Relation between Phong's and Blinn's equation:

$$(\mathbf{h} \cdot \mathbf{n})^{4 \cdot \text{shininess}} \text{ is approximately equal to } (\mathbf{r} \cdot \mathbf{v})^{\text{shininess}} \quad (6.9)$$

Good explanation of specular lighting can be found in [6.1] at page 73-77 (includes also description of some alternative equations) or in [6.5] in chapter two.

6.2.3 Ambient component

The ambient component simulates indirect light contribution (for example a light that bounced off a wall and then reached the object) and emission of a light by the surface itself. This type of light interaction is not computed in diffuse or specular part. This component ensures that every pixel receives at least some minimum amount of color and will not remain black during the rendering.

$$\mathbf{i}_{\text{ambient}} = (\mathbf{m}_{\text{ambient}} * \mathbf{l}_{\text{ambient}}) + \mathbf{m}_{\text{emissive}} \quad (6.10)$$

Where $\mathbf{m}_{\text{ambient}}$ defines the ambient part of material properties (color or a texture value at current pixel), $\mathbf{l}_{\text{ambient}}$ is the ambient light color of a scene and $\mathbf{m}_{\text{emissive}}$ specifies color of light that surface emits.

It can be clearly seen that groups of pixel lit only by ambient lighting component will likely lose the effect of 3D (due to absence of shades) because the light contribution for this part is constant. A cure for this is to ensure that every place in scene receives at least minimum light amount from a light source (i.e. we can use some type of minor headlight light). Another method is to pre-compute offline the ambient (and static diffuse) parts of scene lighting using for example the radiosity method and then use results as textures.

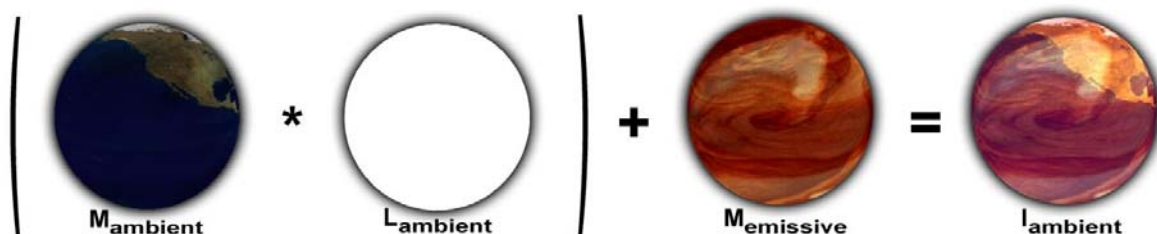


Image shows ambient (and emissive) lighting computation.

6.3 Per-vertex shading

Per-vertex shading (or widely known as the Gouraud shading) was for a long time the only choice for real-time graphics. It's quite simple to implement and runs at reasonable speed on most graphics hardware.

Gouraud in [6.8] described shading method that in the first step computes lighting intensity for every vertex in the scene (using any lighting model we choose, i.e. Phong). Lighting intensity for every pixel of a polygon is then computed as a linear interpolation of intensities on the polygon vertices. For the objects with very high tessellation we can receive very good results comparable with per-pixel shading (this style was used in the movie Toy Story - polygons were subdivided to the pixel size level and then simple Gouraud shading was used). For low tessellation (a common case in games) there is a lack of highlights on large polygons. Another disadvantage is that we cannot use bump mapping.

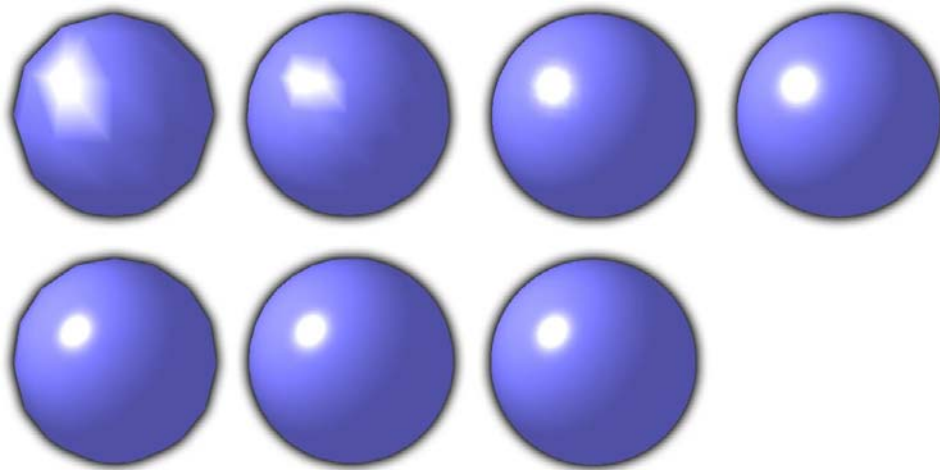


Image shows comparison of per-vertex (top line) and per-pixel (bottom line) shading. Tessellation of the sphere is increasing from left to right.

6.3.1 The vertex shader

The vertex shader that we can use to render scenes using per-vertex shading is shown on next lines. We continue with explanation of every step and a sample setup of Direct3D rasterizer. Because of the fact that basic knowledge of the vertex shader assembly is necessary to understand following code, there will be description of some shader commands in the text. For detailed information read the DirectX 8.1 SDK (online on <http://msdn.microsoft.com>)

```
;-----  
; Constant registers  
; c0-c3      - world * view * proj matrix transposed  
; c4-c7      - world matrix inverted  
; c8         - world matrix transposed  
; c12        - Light position  
; c13        - Eye position  
; c14        - Material diffuse color  
; c15        - Light diffuse color  
; c16        - Light Attenuation parameters  
; c17        - Specular power (shininess)  
;
```

```

; Input registers
; v0 - Position
; v3 - Normal
; v7 - texture coordinates
;
; Output
; oT0 - texcoord
; oD0 - diffuse color
; oD1 - specular color
;-----

vs.1.1.1          ;shader version specification

m4x4 oPos, v0, c0    ;transform vertex to clip space.
                    ;c0 holds transposed clip matrix
mov oT0, v7         ;output texture coordinates

m3x3 r1, v3, c4     ;transform normal. c4-c7 holds
                    ;inverse world transformation matrix.

dp3 r1.w, r1, r1    ;normalize normal
rsq r1.w, r1.w
mul r1, r1, r1.w

m4x4 r7, v0, c8     ;transform vertex to world space.
                    ;c8-c11 holds transposed world
                    ;transformation matrix

add r10.xyz, c12, -r7.xyz ;calculate vector from vertex position to
                    ;light position (both in world space)

dp3 r10.w, r10, r10 ;normalize light vector
rsq r11.w, r10.w
mul r11, r10, r11.w

dst r4, r10.w, r11.w ;compute the distance attenuation
dp3 r7.x, r4, c16    ;in c16 we have modifiers of attenuation
rcp r7.x, r7.x       ;reciprocal of attenuation

add r2, c13, r11    ;compute half vector H = L + V

dp3 r2.w, r2, r2    ;normalize H
rsq r2.w, r2.w
mul r2, r2, r2.w

dp3 r0.y, r1, r2    ;N dot H
dp3 r0.x, r1, r11   ;N dot L
mov r0.w, c17.x     ;get shining strength from c17
lit r4, r0

mul r4, r4, r7.xxxx ;Scale the diffuse and specular intensity
                    ;by the attenuation
mul oD0, c14, r4.y  ;diffuse color * diffuse intensity (r4.y)
mul oD1, c15, r4.z  ;specular color * specular intensity (r4.z)

```

First of all we transform the incoming vertex into the clipping space and output this new position in oPos register. We also assign the texture coordinates to this vertex and output them in oT0 register. It is a good practice to write oPos register as soon as possible, because graphics hardware can reject entire triangle if it is not visible even if shader execution did not complete yet. This is done by the following lines:

```

m4x4 oPos, v0, c0          ;transform vertex to clip space.
                           ;c0 holds transposed clip matrix
mov oT0, v7                ;output texture coordinates

```

Constant registers c0, c1, c2 and c3 holds a transposed matrix that transforms vertex from object's local space into the clipping space (that is $M_{\text{ObjectWorldTransform}} * M_{\text{CameraView}} * M_{\text{CameraProjection}}$). Input register v0 holds vertex position in the object's local space. The m4x4 instruction has following definition (from which it will be clear, why the matrix has to be transposed):

```
m4x4 dest, src0, src1
```

```

dest is destination register
src0 and src1 are source registers

```

m4x4 performs following operations:

```

dest.x = (src0.x * src1.x) + (src0.y * src1.y) + (src0.z * src1.z);
dest.y = (src0.x * src2.x) + (src0.y * src2.y) + (src0.z * src2.z);
dest.z = (src0.x * src3.x) + (src0.y * src3.y) + (src0.z * src3.z);
dest.w = (src0.x * src4.x) + (src0.y * src4.y) + (src0.z * src4.z);

```

For example:

```
m4x4 oPos, v0, c0
```

which is performed as four dot products:

```

dp4  oPos.x, v0, c0
dp4  oPos.y, v0, c1
dp4  oPos.z, v0, c2
dp4  oPos.w, v0, c3

```

In the next step we need to transform normal vector into the world space (and normalize it again).

```

m3x3 r1, v3, c4          ;transform normal. c4-c7 holds
                           ;inverse world transformation matrix.

dp3 r1.w, r1, r1        ;normalize normal
rsq r1.w, r1.w
mul r1, r1, r1.w

```

Constant registers c4, c5 and c6 hold inverse object's world transformation matrix. A matrix 3x3 is enough, because we will transform vector and therefore we do not need the fourth - translation part. Input register v3 holds the vertex normal vector in object local space. The m3x3 instruction has similar definition and effect as m4x4. The only difference is that it does not compute dest.w component. The proof of why using inverse transformation for normals is correct can be found in [6.12].

Normalization of a vector is done by the division with the length. Length of a vector is defined as a square root of the dot product of a vector with itself. Our code simply does this by computing a dot product using the transformed normal vector as both input arguments. Then it computes reciprocal of a square root of this dot product (one instruction) and finally it multiplies original vector with the result. This piece of code is very frequently used in the vertex shaders. Here is the brief description of used instructions:

```
dp3 dest, src0, src1
Computes the three-component dot product of the source registers
```

The following code fragment shows the operations performed

```
dest.x = (src0.x * src1.x) + (src0.y * src1.y) + (src0.z * src1.z);
dest.y = dest.z = dest.w = dest.x;
```

```
rsq dest, src0
Computes the reciprocal square root of the source scalar
```

The following code fragment shows the operations performed

```
float v = fabs(src.x);
if(v == 1.0f)
    dest.x = dest.y = dest.z = dest.w = 1.0f;
else if(v == 0)
    dest.x = dest.y = dest.z = dest.w = PLUS_INFINITY();
else
    dest.x = dest.y = dest.z = dest.w = 1.0f / sqrt(v);
```

```
mul dest, src0, src1
Multiplies sources into the destination.
```

The following code fragment shows the operations performed

```
dest.x = src0.x * src1.x;
dest.y = src0.y * src1.y;
dest.z = src0.z * src1.z;
dest.w = src0.w * src1.w;
```

On the next lines we calculate the light vector. For our needs (future usage in the lighting equation) we compute as a vector from the vertex position to the light position. We need vertex position to be in the same space as light position - the world space. We use add instruction to compute the vector from two points with inversion of last parameter (so it behaves like subtraction). Add instruction is similar to the mul instruction, but it uses piecewise addition instead of multiplication. After this, normalization of a vector is computed (it is not mentioned in the code fragment below).

```
m4x4 r7, v0, c8           ;transform vertex to world space.
                          ;c8-c11 holds transposed world
                          ;transformation matrix

add r10.xyz, c12, -r7.xyz  ;calculate vector from vertex position to
                          ;light position (both in world space)
```

To make the shading look better we also compute distance attenuation of the light. Commonly used attenuation multiplier has the following form:

$$d = 1 / (a_0 * 1 + a_1 * ||\mathbf{l}|| + a_2 * ||\mathbf{l}||^2) \quad (6.11)$$

Where a_0 , a_1 and a_2 specify distance attenuation coefficients and \mathbf{l} is a vector from vertex to light position (in the same space).

To compute the equation 6.11 in a shader we use dst instruction. This instruction uses following input parameters - **src0** = (ignored, $||\mathbf{l}||^2$, $||\mathbf{l}||^2$, ignored) and **src1** = (ignored, $1/||\mathbf{l}||$, ignored, $1/||\mathbf{l}||$) and computes the output vector in a form **dest** = (1, $||\mathbf{l}||$, $||\mathbf{l}||^2$, $1/||\mathbf{l}||$). It can be seen from equation 6.11, that the

first three components of this vector are parameters of the equation. Using a dot product with this vector and the second parameter in form (a0, a1, a2, 0) is a plain step. The rcp instruction then computes reciprocal of this value and the final attenuation.

```
dst r4, r10.w, r11.w           ;compute the distance attenuation
dp3 r7.x, r4, c16             ;in c16 we have modifiers of attenuation
rcp r7.x, r7.x                ;reciprocal of attenuation
```

In the next step we will use Blinn's lighting equation. We compute half vector **h**, then normalize it (Equation 6.7) and then use dot product with the vertex normal to compute specular intensity (Equation 6.8). Diffuse equation is then computed according to the Equation 6.2 with following dot product. The most interesting instruction used in this code fragment is the lit instruction. It takes the input vector in form

src0 = (**n.l**, **n.h**, **ignored**, **shininess**) and computes the output vector **dest** = (ignored, *i_{diffuse}*, *i_{specular}*, ignored) with lighting equation components. Note that this instruction returns expected values when **n.l** is less than zero – **dest** = (ignored, 0, 0, ignored). Also note that instead of Blinn's specular equation we can use Phong's equation.

```
add r2, c13, r11              ;compute half vector H = L + V

dp3 r2.w, r2, r2              ;normalize H
rsq r2.w, r2.w
mul r2, r2, r2.w

dp3 r0.y, r1, r2              ;N dot H
dp3 r0.x, r1, r11            ;N dot L
mov r0.w, c17.x               ;get shining strength from c17
lit r4, r0
```

The last vertex shader instructions block is multiplying the output from a lit instruction with the recently computed attenuation factor. Then this result is then modulated using the diffuse light color and specular light color.

```
mul r4, r4, r7.xxxx           ;Scale the diffuse and specular intensity
                                ;by the attenuation
mul oD0, c14, r4.y            ;diffuse color * diffuse intensity (r4.y)
mul oD1, c15, r4.z           ;specular color * specular intensity (r4.z)
```

6.3.2 Rasterizer setup

Rasterization setup specific to this shader includes:

Turning off the DirectX3D lighting, because we are providing our own shader for this.

```
Direct3DDevice->SetRenderState(D3DRS_LIGHTING, FALSE);
```

Enabling usage of the second color (specular intensity is in oD1):

```
Direct3DDevice->SetRenderState(D3DRS_SPECULARENABLE, TRUE);
```

Enabling modulation of texture stage 0 (diffuse texture) with diffuse intensity and then add the specular intensity:

```
Direct3DDevice->SetTextureStageState(0, D3DTSS_COLOROP, D3DTOP_MODULATE);
Direct3DDevice->SetTextureStageState(0, D3DTSS_COLORARG1, D3DTA_TEXTURE);
Direct3DDevice->SetTextureStageState(0, D3DTSS_COLORARG2, D3DTA_DIFFUSE);
```

6.3.3 Conclusion

Previous shader example handles only one light, but the length of a vertex shader allows us to use more lights in one rendering pass. The other possibility (used with greater count of lights) is to use the multi-pass rendering.

Per-vertex shading presented here was written in the vertex shader, which is accelerated only on DirectX 8 compliant hardware. On the older cards we can use fixed function rendering pipeline if accelerated (NVIDIA GeForce, GeForce2 and ATI Radeon 7500). On the oldest cards we can compute the vertex shader on a host CPU (NVIDIA Riva TNT, ATI Rage and Matrox G400). The simplicity of this shader makes it is faster than per-pixel shading on every current hardware, so it will be still used in the games if not as major, then as a fallback feature.



Image shows scene lit with diffuse shading

6.4 Per-pixel shading

First of all let's describe the per-pixel shading. It differs from the per-vertex shading in the fact that final intensity is not interpolated across the polygon, but rather the intensity is computed for each pixel. Produced results are more realistic. The main advantage for the real-time graphics is that we specify the normals using a special texture (called normal-texture). In the following text we step through various parts of the per-pixel shading shaders and introduce a spotlight term. At the end we will give a brief overview of some more improvements.

6.4.1 Ambient component

The ambient component is the one simplest in whole shading pipeline. It only uses one constant multiplier to multiply the ambient texture. The vertex shader is very simple. It just transforms each vertex into the clipping space and passes the texture coordinates to a pixel shader.

```
;-----  
; Constant registers  
; c4-c7 world * view * proj  
;  
; Used input registers
```

```

; v0 - Position
;
; Output
; oT1 - ambient (and emissive) color multiplier
; oT0 - texture mapping coordinate
;-----

vs.1.0

m4x4 oPos, v0, c4          ;vertex clip position
mov oT0.xy, v2.xy         ;Texture coordinates for color texture

```

The pixel shader is very simple too, just the multiplication of a texture sampler with the constant ambient multiplier.

```

;-----
; Constant registers
; c0 - ambient multiplier
;
; Used input registers
; t0 - Texture coordinate
;
; Used input texture stages
; stage0 - ambient texture
;
; Output
; r0 - output color
;-----

ps.1.4
texld r0, t0              ;color map

mul r0, r0, c0           ;create ambient part

```

Arithmetic instructions are very similar to those in the vertex shaders. Pixel shader specific instruction is `texld`, which uses passed texture coordinated to load filtered texture data into a register. Pixel shader has only one output register - `r0`. The color that is written into this register will be stored in the render target (a screen in common).

Previous pixel shader can be modified to be more robust and to be more similar to the Equation 6.10. We introduce the emissive multiplier, emissive texture and one additional constant modifier. We use one new instruction - `mad`. It uses three source registers **`src0`**, **`src1`**, **`src2`** and produces the result **`dest = src0 * src1 + src2`**.

```

;-----
; Constant registers
; c0 - ambient multiplier
; c1 - emissive multiplier
; c2 - constant term
;
; Used input registers
; t0 - Texture coordinate
;
; Used input texture stages
; stage0 - ambient texture
; stage1 - emissive texture
;

```

```

; Output
; r0 - output color
;-----

ps.1.4
texld r0, t0          ;color map
texld r1, t0          ;emissive map

mad r0, r0, c0, c2    ;compute ambient part and add constant term
mad r0, r1, c1, r0    ;compute emissive part and add ambient

```

With this shader we can handle ambient and emissive model very similar to the one in the 3D Studio Max 4, which can be expressed like this:

$$\mathbf{i}_{\text{ambient_emissive}} = \mathbf{i}_{\text{ambient}} + \mathbf{i}_{\text{emissive}} \quad (6.12)$$

$$\mathbf{i}_{\text{ambient}} = \mathbf{l}_{\text{amb}} * [a_{\text{coeff}} * \mathbf{a}_{\text{tex}} + (1 - a_{\text{coeff}}) * \mathbf{a}_{\text{col}}] \quad (6.13)$$

$$\mathbf{i}_{\text{emissive}} = e_{\text{on}} * \mathbf{e}_{\text{col}} + (1 - e_{\text{on}}) * e_{\text{val}} * [d_{\text{coeff}} * \mathbf{d}_{\text{tex}} + (1 - d_{\text{coef}}) * \mathbf{d}_{\text{col}}] \quad (6.14)$$

Where:

- $\mathbf{i}_{\text{ambient}}$ is the overall material ambient intensity.
- $\mathbf{i}_{\text{emissive}}$ is the overall material emissive intensity (in 3D Studio Max is it called self illumination)
- \mathbf{l}_{amb} is the ambient light color for the scene
- a_{coeff} is the coefficient of ambient texture (from 0.0 to 1.0)
- \mathbf{a}_{tex} is the ambient texture of material
- \mathbf{a}_{col} is the ambient color of material.
- e_{on} can have value 1.0 and then emissive color is used, or 0.0 and then diffuse color is used (it is a kind of flow control)
- \mathbf{e}_{col} is the emissive color
- e_{val} is the value of emission of a diffuse texture (from 0.0 to 1.0)
- d_{coeff} is the coefficient of a diffuse texture map (from 0.0 to 1.0)
- \mathbf{d}_{tex} is the diffuse texture of material
- \mathbf{d}_{col} is the diffuse color of material

Previous equation can be rewritten into a form that is compatible with our enhanced emissive pixel shader (the diffuse texture is used as emissive):

$$\mathbf{i}_{\text{ambient_emissive}} = \mathbf{c}_0 * \mathbf{a}_{\text{tex}} + \mathbf{c}_1 * \mathbf{d}_{\text{tex}} + \mathbf{c}_2 \quad (6.15)$$

$$\mathbf{c}_0 = \mathbf{l}_{\text{amb}} * a_{\text{coeff}} \quad (6.16)$$

$$\mathbf{c}_1 = (1 - e_{\text{on}}) * e_{\text{val}} * d_{\text{coeff}} * (1.0, 1.0, 1.0) \quad (6.17)$$

$$\mathbf{c}_2 = \mathbf{l}_{\text{amb}} * (1 - a_{\text{coeff}}) * \mathbf{a}_{\text{col}} + e_{\text{on}} * \mathbf{e}_{\text{col}} + (1 - e_{\text{on}}) * e_{\text{val}} * (1 - d_{\text{coef}}) * \mathbf{d}_{\text{col}} \quad (6.18)$$

The terms \mathbf{c}_0 , \mathbf{c}_1 and \mathbf{c}_2 can be computed outside the pixel shader and they would be computed only once if we do not plan to animate the material.

Another modification that can be done is to get the multipliers not from the pixel shader constants, but to pass them from the vertex shader as a texture coordinate (texture coordinates are also called interpolators, because they are interpolated across the vertices. We use this technique later in the diffuse and specular components). This extension is necessary if

the precision and range of pixel shader constants (-1.0, 1.0) is not enough. Texture coordinates from the vertex shader have greater precision and support the range (-2048, 2048). We use the texcrd instruction to load the texture coordinate value into a register (only a register can be used in the arithmetic operations in pixel shader 1.4) and the ranges we actually receive are clamped into the range (-8.0, 8.0) - the range of a pixel shader register.

```

;vertex shader
vs 1.0
m4x4 oPos, v0, c4           ;vertex clip position
mov oT0.xy, v2.xy          ;Texture coordinates for color texture
mov oT1, c0                 ;Ambient color multiplier is in c0
constant
mov oT2, c1                 ;emissive color multiplier is in c1
constant
mov oT3, c2                 ;Constant term is in c2 constant

;pixel shader
ps.1.4
texld r0, t0                ;color map
texld r1, t0                ;emissive map
texcrd r2.rgb, t1.xyz       ;ambient color multiplier
texcrd r3.rgb, t2.xyz       ;emissive color multiplier
texcrd r4.rgb, t3.xyz       ;constant color term

mad r0, r0, r2, r4          ;compute ambient part and add constant
term
mad r0, r1, r3, r0          ;compute emissive part and add ambient

```

The reason, why we spent so much time with such a simple task as the ambient component is that it serves us as an introduction to some of the concepts used later in the shaders – extending the constant value precision in a pixel shader using interpolators and transfer of expensive constant calculations out of shader. It is a good practice to put the ambient part into the separate render pass. It is simple, fast and it fills the z-buffer. In the subsequent passes hardware can reject early the invisible pixels and costly pixel shader computations will be computed only for visible pixels.



Image shows scene lit with ambient shading

6.4.2 Diffuse component

The diffuse part of lighting is a bit more difficult. As we know from the Equation 6.2 we need to compute a dot product of the normal and the light vector. These two vectors need to be stored in the same space to receive correct results. Normals are stored in the tangent (or also called texture) space. Light vector is computed for each vertex in the world space and then interpolated across a polygon and it remains in the world space. Common practice is to transform the light vector into the texture space because it can be done in the vertex shader.

Let us describe what the normal maps are. Software such as 3D Studio Max uses height textures to perform bump mapping effects. Intensity of a pixel in the texture specifies height of that pixel. Normal maps are different. Each pixel of a normal map represents direction of a normalized surface normal and x coordinate of the normal is encoded in r component of pixel, y in g and z in b . Because of the fact that only unsigned numbers can be stored in the texture we have to map the range $[-1, 1]$ to range $[0, 1]$ for each component. The normal map can be generated directly from a height map and DirectX Library provides the `D3DXComputeNormalMap` function to perform this action.

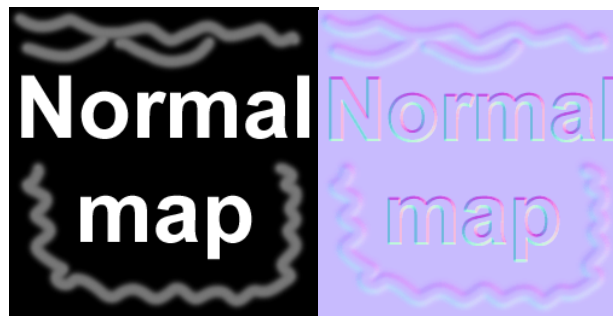


Image on the left shows height texture, on the right is converted to normal map

Now we will describe what the tangent space is. Tangent space has 3 perpendicular axes, \mathbf{t} , \mathbf{b} and \mathbf{n} . Vector \mathbf{t} , the tangent, is parallel to the direction of increasing s or t of a surface. Vector \mathbf{n} , the normal, is perpendicular to the local surface. Vector \mathbf{b} , the binormal, is perpendicular to both \mathbf{t} and \mathbf{n} . For a triangle we can consider increasing texture coordinate u as the s parameter of a surface. Similarly texture coordinate v can play the role of a t parameter. Then at each vertex we can define the tangent space with \mathbf{t} vector in the direction of incremented u , \mathbf{n} vector as the vertex normal and \mathbf{b} computed to be perpendicular to both previous vectors. Picture below shows this situation for one triangle. The red arrow is a vector in u direction, blue one is a vertex normal and green one is a binormal.

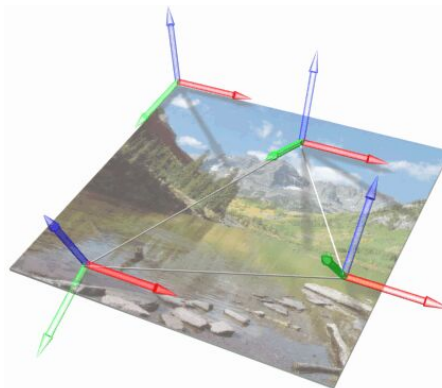


Image shows tangent space basis for each vertex of triangle. Red arrow is vector in u direction, blue one is vertex normal and green one is binormal.

These three vectors provide a rotation matrix for transformation of a vector into tangent space. It looks like:

$$\begin{bmatrix} t_x & b_x & n_x & 0 \\ t_y & b_y & n_y & 0 \\ t_z & b_z & n_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Computation of the basis is described in [6.10] or in [6.1] in the chapter about bump mapping. DirectX 8.1 Library provides a method **D3DXComputeTangent** that can be used. Following short description of the tangent space is taken from [6.9] (Vectors, scalars and code was changed to follow our notation):

*Computing a tangent space vector is just like computing a normal. Remember that **n** is defined as the normal, so we should already have that piece of information. Now, we need to generate **t** and **b**. To generate the tangent space basis vectors (**t** and **b**), use the following equation:*

```
Vec1 = Vertex3 - Vertex2
Vec2 = Vertex1 - Vertex2
DeltaU1 = Vertex3.u - Vertex2.u
DeltaU2 = Vertex1.u - Vertex2.u
DirectionV = normalize( DeltaU2*Vec1 - DeltaU1*Vec2 )
DirectionU = normalize( cross( DirectionV, Vertex.n ) )
DirectionW = normalize( cross( DirectionU, DirectionV ) )
```

(Note by author: In previous equation, **Vertex1**, **Vertex2** and **Vertex3** are the vertices of the triangle, member of which current vertex (**Vertex**) is. *VertexN.u* and *VertexN.v* means texture coordinates *u* and *v* for given vertex. **Vertex.n** is current vertex normal.)

And the citation follows:

*Usually, tangents and normals are calculated during the authoring process, and the binormal is computed in the shader (as a cross product). So, the only field we are adding to our vertex format is an extra **t** vector. Additionally, if we assume the basis is orthonormal, we don't need to store **b** either, since it is just **t** cross **n**.*

*A couple of points to keep in mind: Tangents need to be averaged—and be careful about texture wrapping (since it modifies *u* and *v* values).*

Now we show and describe the vertex and pixel shaders used to compute per-pixel diffuse lighting. The vertex shader is quite simple, it only computes the tangent space basis and transforms normalized light vector into the tangent space (note that tangent space matrix is stored in transposed form, because this is required for matrix-to-matrix multiplication in shaders which is done by dot products).

```
;-----
; Constant registers
; c0-c3      - world space transposed
```

```

; c4-c7      - world * view * proj
; c8         - Light position (in world space)
;
; Input registers
; v0 - Position
; v1 - Normal
; v2 - Texture
; v3 - Tangent T
;
; Fixed temporary registers
; r9, r10, r11 - tangent space basis
; r8 - vertex world position
; r7 - light vector
;
; Output
; oT0 - tex coord
; oT1 - Light vector (in tangent space)
;-----
vs.1.1

;-----
;Following code output position and texture coordinates
;-----
m4x4 oPos, v0, c4           ;vertex clip position
mov oT0.xy, v2.xy          ;Texture coordinates for color texture

;-----
;Following code generates Tangent space base vectors
;-----
m3x3 r11, v1, c0           ;Transform normal to world space
m3x3 r9, v3, c0            ;Transform tangent to world space
mul r0,r9.zxyw,r11.yzxw    ;These two instructions compute cross
mad r10,r9.yzxw,r11.zxyw,-r0 ;product to compute binormal NxT

;-----
;Following code computes light vector and transforms it to texture space
;-----
m4x4 r8, v0, c0           ;Transform vertex into world space
add r0, c8, -r8           ;Build the light vector from light
source to vertex

dp3 r1.w, r0, r0           ;normalization
rsq r1.w, r1.w
mul r7, r0, r1.w

dp3 oT1.x, r9, r7         ;transform the light vector with n, t,
nxt
dp3 oT1.y, r10, r7        ;into texture coordinate space
dp3 oT1.z, r11, r7        ;for per pixel diffuse lighting

```

Entire lighting is done in the pixel shader. We will try to use somewhat complex diffuse lighting equation very similar to the one used in 3D Studio Max. Equation 6.19 is used to compute diffuse color contribution, equation 6.20 is the one used during lighting.

$$\mathbf{d}_{col} = [d_{coeff} * \mathbf{d}_{tex} + (1 - d_{coef}) * \mathbf{d}_{col}] \quad (6.19)$$

$$i_{diffuse} = (\mathbf{n} \cdot \mathbf{l}) * \mathbf{d}_{col} * \mathbf{l}_{diffuse} \quad (6.20)$$

Where:

- \mathbf{d}_{col} is overall diffuse color contribution
- d_{coeff} is the coefficient of diffuse texture map (from 0.0 to 1.0)
- \mathbf{d}_{tex} is the diffuse texture of material
- \mathbf{d}_{col} is the diffuse color of material
- $\mathbf{l}_{diffuse}$ is the diffuse light color intensity
- \mathbf{n} and \mathbf{l} are the normalized normal and light vectors

The equation 6.19 and 6.20 can be transformed into a form with only two constants that is more efficient for the pixel shader:

$$\mathbf{i}_{diffuse} = (\mathbf{n} \cdot \mathbf{l}) * (\mathbf{c}_0 * \mathbf{d}_{tex} + \mathbf{c}_1) \quad (6.21)$$

$$\mathbf{c}_0 = d_{coeff} + \mathbf{l}_{diffuse} \quad (6.22)$$

$$\mathbf{c}_1 = (1 - d_{coeff}) * \mathbf{d}_{col} + \mathbf{l}_{diffuse} \quad (6.23)$$

The pixel shader is now very simple. We just load the diffuse color and the normal vector for given pixel and with texcrd we load interpolated light vector (the oT1 register from vertex shader). Then we compute a dot product as specified in the equation 6.2. This instruction uses `_sat` modifier, which clamps the result into range [0.0, 1.0]. Because the normals are stored in unsigned form (as specified in the beginning of this paragraph) the `_bx2` modifier must be used on this register. It subtracts 0.5 from each channel and scales the result by 2.0, which results in the expansion of data from range [0.0 to 1.0] to [-1.0 to 1.0]. The `mad` instruction is then used to compute diffuse color contribution as specified in the equation 6.21.

```

;-----
; Constant registers
; c0 - diffuse texture multiplier (premultiplied with light color)
; c1 - additonal diffuse constant (premultiplied with light color)
;
; Used input registers
; t0 - color / bump coordinates
; t1 - light vector in tangent space
;
; Used input texture stages
; stage0 - ambient texture
; stage1 - normal texture
;
; Output
; r0 - output color
;-----

```

ps.1.4

```

texld r0, t0           ;diffuse color (t)
texld r1, t0           ;normal vector (n)
texcrd r2.rgb, t1.xyz  ;Light vector (l)

dp3_sat r1, r1_bx2, r2 ;r1 = dot(normal, light)
mad r0, r0, c0, c1     ;compute diffuse color
mul r0, r0, r1         ;modulate with dot product

```

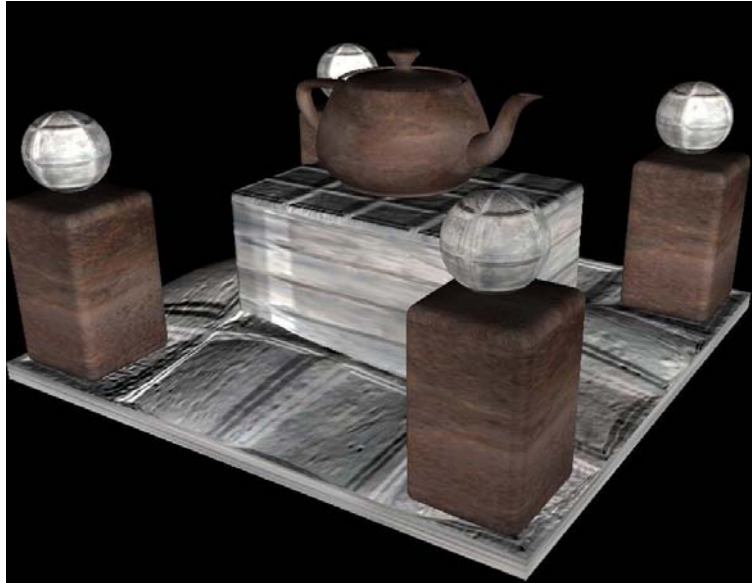


Image shows scene lit with diffuse shading

6.4.3 Specular component

Specular component is really simple after we done the diffuse part. We just have to decide, if we want to use Blinn's (6.8) or Phong's (6.5) equation. The first method computes half vector \mathbf{h} per-vertex and then computes specular term dot product in the pixel shader. The second one is a bit more complex. The eye vector (from vertex to viewer's position) is computed per-vertex and the reflection vector and a dot product is computed in the pixel shader. We will show second approach as it produces more precise results.

The only change in the vertex shader is addition of following code at the end of shader presented in previous section. Code assumes that position of the viewer is in $\mathbf{c9}$ constant of a shader and eye vector is passed to the pixel shader in tangent space in register $\mathbf{oT2}$.

```

;-----
;Following code computes specular lighting parts
;for pixel shader (Eye vector)
;-----
;
add r0, c9, -r8           ;build the eye vector from vertex to
camera source

dp3 r1.w, r0, r0           ;normalize the eye vector
rsq r1.w, r1.w
mul r6, r0, r1.w

dp3 oT2.x, r9, r6         ;transform the eye vector with N, T, NxT
dp3 oT2.y, r10, r6
dp3 oT2.z, r11, r6

```

Pixel shader is on the other hand very different. We added two new constants and one new texture, so first we take a look at the math and then at the shader. We want to add specular contribution to diffuse as shown in the equation 6.1 (let us forget about ambient part for now). We can specify specular color $\mathbf{i}_{\text{specular}}$ equation by extending the equation 6.6. We introduce the specular intensity component s_{strength} (which in pixel shader will be alpha channel of diffuse texture) and color of specular reflection $\mathbf{s}_{\text{color}}$.

$$i_{\text{specular}} = (\mathbf{r} \cdot \mathbf{v})^{\text{shininess}} * S_{\text{strength}} * \mathbf{s}_{\text{color}} * I_{\text{specular}} \quad (6.24)$$

The new constants are c2 where we store $\mathbf{s}_{\text{color}} * I_{\text{specular}}$ and c3 with *shininess* parameter. Pixel shaders do not support power function and we have to use texture as a look-up table to emulate this functionality. Each pixel at coordinates (x, y) will store the result of function $\text{pow}(x, y) = x^{(y*100.0)}$. Parameters x and y are in the range [0.0 to 1.0]. We will use this texture to compute the value of $(\mathbf{r} \cdot \mathbf{v})^{\text{shininess}}$.

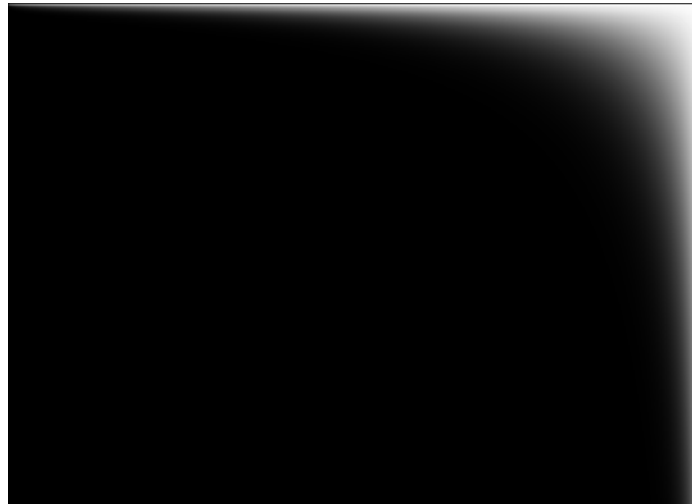


Image shows part of a look up texture for power function.

The pixel shader is divided into two phases (special phase instruction). In the first phase we compute the reflectance vector with first mad instruction (`_x2` register modifier means multiplication by two) then determine coordinates into look up texture. In the second phase we load specular intensity is loaded from look-up texture using the coordinates from first phase (this is also called dependent read). Then compute equation 6.24 and add diffuse component.

```

;-----
; Constant registers
; c0 - diffuse texture multiplier (premultiplied with light color)
; c1 - additional diffuse constant (premultiplied with light color)
; c2 - specular texture multiplier (premultiplied with light color)
; c3 - shininess
;
; Used input registers
; t0 - color / bump coordinates
; t1 - light vector in tangent space
; t2 - eye vector in tangent space
;
; Used input texture stages
; stage0 - ambient texture
; stage1 - normal texture
; stage2 - specular light map
;
; Output
; r0 - output color
;-----

```

ps.1.4

```
texld r1, t0 ;normal vector (n)
```

```

texcrd r2.rgb, t1.xyz           ;Light vector (l)
texcrd r3.rgb, t2.xyz           ;eye vector (v)

dp3 r4, r1_bx2, r2              ;r4 = dot(normal, light)
mad r5.rgb, r4_x2, r1_bx2, -r2  ;compute reflectance vector r5=2(n.l)n-l

;compute power look up coordinates
dp3 r5, r5, r3                  ;r5.x = dot(reflect, eye)
mov r5.y, c3                    ;r5.y = shininess

phase
texld r0, t0                    ;load specular texture (gloss map is in
alpha channel)
texld r2, r5                    ;specular light map

mad r0.rgb, r0, c0, c1          ;compute diffuse color into r0
mul r4.rgb, r0, r4              ;modulate with diffuse dot product

mul r0.rgb, r0.a, r2            ;modulate gloss map with (r.v)^shininess
mad r0.rgb, r0, c2, r4          ;modulate specular term with
;specular constant and add diffuse

```

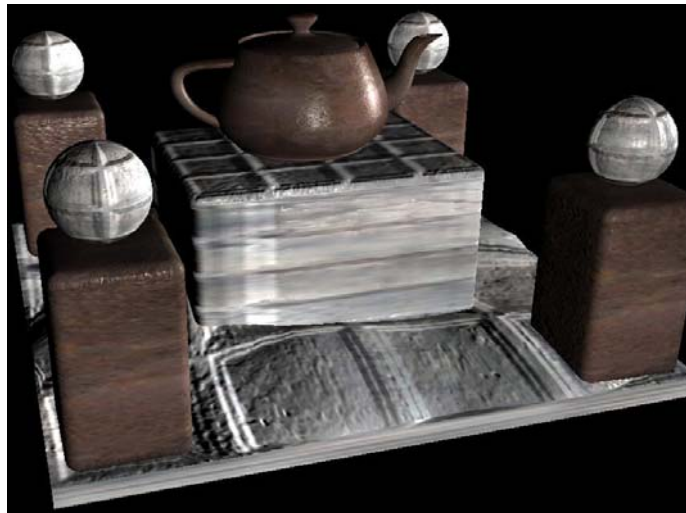


Image shows scene with diffuse and specular shading

6.4.4 Spotlight

All previous computations were done for omni directional lights only. If we want to simulate a spotlight we would use one additional texture that would be projected in the direction of the spotlight on every object. Texture will contain light color and spotlight's cone and of course can be used as a projector.

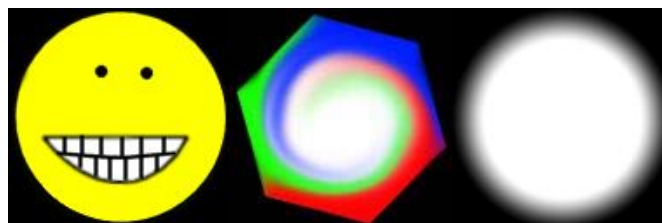


Image shows three samples of spotlight textures.

Preparation of the spotlight is very simple. Due to the fact that it acts like a camera (we can look from spotlight's origin in its direction and visible objects are lit), we can use the same popular methods as with cameras - **D3DXMatrixLookAtLH** (to build transformation into space, where light's point is origin and vector from light source to light's target is mapped to Z axis) and **D3DXMatrixPerspectiveFovLH** (to build perspective projection matrix. Light's falloff angle is used as FOV parameter) to build clipping matrix. If we transform vertex (in world space) with this matrix then x and y components contain texture coordinates into the spotlight texture after the projective divide with w . Clipping space coordinates are in range $[-1.0$ to $1.0]$ and texture has ranges $[0.0$ to $1.0]$ so we need to multiply the clipping matrix with another one, which does this transformation. We build matrix that scales by 0.5 and add 0.5 to x and y components. Note that scale of y parameter is inverted, because the texture has y value of 0.0 at first (top) row and 1.0 at last (bottom) row and the clipping matrix transforms y value of 0.0 to center and 1.0 to "top of space".

$$\begin{pmatrix} 0.5 & 0 & 0 & 0 \\ 0 & -0.5 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0.5 & 0.5 & 0 & 1 \end{pmatrix}$$

The final matrix that transforms vertex in from the object space to the projective texture coordinates is the following equation:

$$\mathbf{M}_{\text{spot}} = \mathbf{M}_{\text{ObjectWorldTransform}} * \mathbf{M}_{\text{SpotView}} * \mathbf{M}_{\text{SpotProjection}} * \mathbf{M}_{\text{Texturescale}} \quad (6.25)$$

Changes to the vertex shader are quite simple, we just transform vertex position with transposed \mathbf{M}_{spot} matrix stored in constants $c10$, $c11$, $c12$, $c13$ and output it as oT3 texture coordinate interpolator.

```
m4x4 oT3.xyw, v0, c10 ;vertex transform to texture coordinates
```

Pixel shader change is easy too. We load texture sample to the register (note that **_dw** modifier used ensures perspective divide with w parameter and the result is then perspective correct) and modulate the lighting result with this sample as the last instruction.

```
Phase
texld r4, t3_dw.xyw ;projector map
.
.
mul r0.rgb, r0, r4 ;modulate with spotlight texture
```

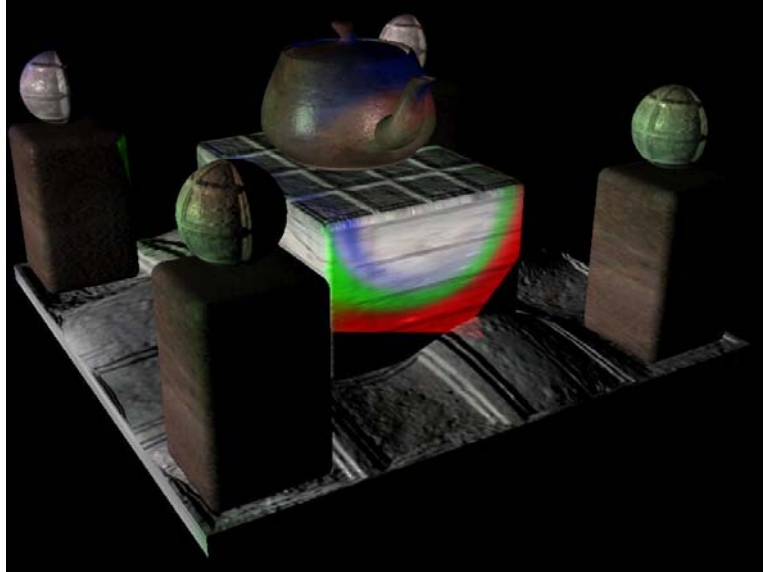


Image shows scene lit with spotlight projectors

6.4.5 Further improvements

There are several improvements possible (and commonly done) to improve the quality (or the performance) of per-pixel shading.

- **Distance attenuation** was mentioned in per-vertex shading section and not mentioned in the per-pixel part. There are two methods how to do this in general:
 1. Compute attenuation in the vertex shader and store it in \mathbf{oD}_n register (very similar to a part of shader in the section 6.3.1) and in pixel shader add another modulation of lighting term with input register \mathbf{v}_n
 2. We can use a 1D texture where we store the attenuation distance. Pixel at coordinate x contains attenuation at distance of x from light (x should be actually in range $[0.0 \text{ to } 1.0]$ for example in the clipping space of the spotlight). We can compute the distance in the vertex shader and store it in x component of \mathbf{oT}_n register. We sample the texture in the pixel shader and modulate the final color contribution. This allows us to use very exotic functions of attenuation.
- **Normalization of light vector** in pixel shader is sometimes required if the light is too close to a surface. Light vector is normalized in the vertex shader, but it is getting shorter as it is interpolated across surface. This is a common case when we use large polygons. Pixels closer to light source are darker than those on the polygon boundary. Look at next picture to see this case.

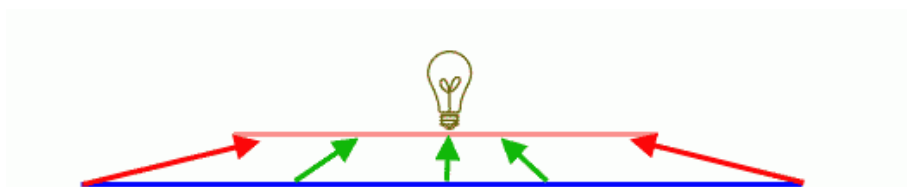


Image shows de-normalization of interpolated light vector, if light is too close to surface

There are several solutions how to solve this problem:

1. A partial solution is to use only small polygons. Light will not be near enough to the surface to cause problem in common cases then.
2. Use distance attenuation texture and in distances smaller than 1 store multiplier that will "normalize" the light vector according to the distance.
3. Use Newton-Raphson approximation in the pixel shader (an idea from Tomas Arce presentation at GDC2002):

```
mul r0, LightVector, 0.5
dp3 r1, LightVector, LightVector
mad r0, 1-r1, r0, LightVector
```

4. Use a cube map texture, where for each vector (x,y,z) we will store normalized vector in components (r,g,b), very similar to normal texture. Pixel shader will be then modified from:

```
texcrd r2.rgb, t1.xyz ;Light vector (l)
```

To:

```
texld r2, t1 ;Light vector (l) from
cube map
```

We need to use the **r2_bx2** modifier instead of simple **r2** everywhere in the pixel shader. Previous example assumes the cube map loaded into texture stage 2.



Image shows the normalization cube map texture

- **Per-pixel shininess** is a nice way to simulate different properties of a surface. In section 6.4.3 we introduced shining strength (gloss) texture in alpha channel of a diffuse texture. The shininess parameter was a pixel shader constant. Because of the fact, that we have free alpha channel in the normal map, we can store per pixel shininess there and use it with no loss of speed. This way we can simulate for example a checkerboard floor with black rectangles made from marble and white ones made from wood.
- **Reflectance** can be rendered on the per-pixel basis. We have to compute reflection vector for camera in the same way we did it for light and use it to sample cube map texture with stored reflection images. One difference is that we need to do this computation in world space (or objects space), so instead of transforming light and eye vector to tangent space in vertex shader, we pass tangent-to-world space transformation matrix to the pixel shader, transform normal vector to the world space and then compute reflection vector. We can add **Fresnel term** to compute reflection intensity to achieve more realistic results (can be added even if we do not use reflection to modulate the specular term). A good explanation, examples of the shaders and approximations can be found in [6.11].
- **Lightmaps** can be added to a simple ambient pass. We can store lighting intensity in lightmaps as we mentioned before. Due to the slowness of the computation, lightmaps

are pre-computed during design of the scene and stored in textures. Computation is correct only for static lights and static objects (and only for diffuse component). For these objects we can add modulation of diffuse texture with lightmap to the ambient part of shading. For static lights and dynamic objects, lightmaps can be stored in a 3D texture, but memory consumption is very high and therefore resolution of texture has to be low.

- To gain performance, we can compute **multiple lights in one pass**. With vertex shaders 1.1 and pixel shaders 1.4 we can do 2 per pixel specular and diffuse spotlights, 3 per pixel specular and diffuse omnidirectional lights or 4 diffuse omnidirectional lights in single pass.

Here is the example of 3 diffuse omnidirectional lights in a single pass and single phase (which is generally faster than shader with 4 diffuse lights that needs two phases)

```

;-----
; Vertex shader for optimized per pixel diffuse - 3 lights in single pass
;and single phase
;-----
;
; Constant registers
; c0-c3      - world space transposed
; c4-c7      - world * view * proj
; c8         - Light0 position (In World Space)
; c9         - Light1 position (In World Space)
; c10        - Light2 position (In World Space)
; c16        - Global attenuation parameters
;
; Input registers
; v0 - Position
; v1 - Normal
; v2 - Texture
; v3 - Tangent T
;-----

vs.1.1.1

dcl_position v0
dcl_normal v1
dcl_texcoord v2
dcl_tangent v3

;-----
;Following code output position and texture coordinates
;-----
;
m4x4 oPos, v0, c4           ;vertex clip position
mov oT0.xy, v2.xy          ;Texture coordinates for color texture

;-----
;Following code generates Tangent space base vectors
;-----
;
m3x3 r11, v1, c0           ;use normal as third parameter of
texture space
mov r11.w, v1.w
m3x3 r9, v3, c0            ;generate tangent SxN
mov r9.w, v3.w

```

```

mul r0,r9.zxyw,r11.yzxw      ;The cross product to compute binormal NxT
mad r10,r9.yzxw,r11.zxyw,-r0 ;The cross product to compute binormal NxT

m4x4 r8, v0, c0              ;Transform vertex into world position

;-----
;Following code computes light vector in texture space
;for pixel shader - LIGHT 1
;-----
;
add r0, c8, -r8              ;Build the light vector from light source
                               ;normalize the light vector (d stands for distance)
dp3 r2.w, r0, r0              ;r2.w = r0 * r0 = (x*x) + (y*y) + (z*z)
rsq r1.w, r2.w                ;r1.w = 1/d = 1/||V|| = 1/sqrt(r2.w)
mul r7, r0, r1.w             ;r7 = r0 * (1/d) = r0/d

dst r2, r2.w, r1.w           ;compute the distance attenuation
dp3 r2, r2, c16               ;in c16 we have modifiers of
attenuation
rcp oT4.x, r2.x              ;reciprocal of attenuation

dp3 oT1.x, r9, r7            ;transform the light vector with N, T, NxT
dp3 oT1.y, r10, r7           ;into texture coordinate space
dp3 oT1.z, r11, r7           ;for per pixel diffuse lighting

;-----
;Following code computes light vector in texture space
;for pixel shader - LIGHT 2
;-----
;
add r0, c9, -r8              ;Build the light vector from light
source to vertex

                               ;normalize the light vector (d stands
for distance)
dp3 r2.w, r0, r0              ;r2.w = r0 * r0 = (x*x) + (y*y) + (z*z)
rsq r1.w, r2.w                ;r1.w = 1/d = 1/||V|| = 1/sqrt(r2.w)
mul r7, r0, r1.w             ;r7 = r0 * (1/d) = r0/d

dst r2, r2.w, r1.w           ;compute the distance attenuation
dp3 r2, r2, c16               ;in c16 we have modifiers of
attenuation
rcp oT4.y, r2.x              ;reciprocal of attenuation

dp3 oT2.x, r9, r7            ;transform the light vector with N, T, NxT
dp3 oT2.y, r10, r7           ;into texture coordinate space
dp3 oT2.z, r11, r7           ;for per pixel diffuse lighting

;-----
;Following code computes light vector in texture space
;for pixel shader - LIGHT 3
;-----
;
add r0, c10, -r8             ;Build the light vector from light
source to vertex

                               ;normalize the light vector (d stands
for distance)
dp3 r2.w, r0, r0              ;r2.w = r0 * r0 = (x*x) + (y*y) + (z*z)
rsq r1.w, r2.w                ;r1.w = 1/d = 1/||V|| = 1/sqrt(r2.w)
mul r7, r0, r1.w             ;r7 = r0 * (1/d) = r0/d

dst r2, r2.w, r1.w           ;compute the distance attenuation

```

```

dp3 r2, r2, c16           ;in c16 we have modifiers of
attenuation
rcp oT4.z, r2.x           ;reciprocal of attenuation

dp3 oT3.x, r9, r7         ;transform the light vector with N, T, NxT
dp3 oT3.y, r10, r7        ;into texture coordinate space
dp3 oT3.z, r11, r7        ;for per pixel diffuse lighting

```

- And the pixel shader:

```

;-----
;Pixel shader for optimized per pixel diffuse - 3 lights in single pass
;and single phase
;-----
;
; Constant registers
; c0 - light1 color
; c1 - light2 color
; c2 - light3 color
;
; Used input registers
; t0 - color / bump coordinates
; t1 - light vector in tangent space
; t2 - light2 vector in tangent space
; t3 - light3 vector in tangent space
; t4 - attenuation of each lights in RGB (R = L1, G = L2, B = L3)
;
; Used input texture stages
; stage0 - ambient texture
; stage1 - normal texture
;
; Output
; r0 - output color
;-----

```

```

ps.1.4
texld r0, t0             ;diffuse color (t)
texld r1, t0             ;normal vector (n)
texcrd r2.rgb, t1.xyz    ;Light vector (l1)
texcrd r3.rgb, t2.xyz    ;Light vector (l2)
texcrd r4.rgb, t3.xyz    ;Light vector (l3)
texcrd r5.rgb, t4.xyz    ;attenuation

dp3_sat r2.r, r1_bx2, r2 ;r2 = dot(normal, light1)
dp3_sat r2.g, r1_bx2, r3 ;r3 = dot(normal, light2)
dp3_sat r2.b, r1_bx2, r4 ;r4 = dot(normal, light3)
mul r2.rgb, r2, r5       ;attenuation

mul r1.rgb, r2.r, c0
mad r1.rgb, r2.g, c1, r1
mad r1.rgb, r2.b, c2, r1
mul r0, r0, r1

```

- Here is the example of 2 per pixel specular and diffuse spotlights in one pass

```

;-----
;Vertex shader for 2 per pixel specular and diffuse spotlights in one pass
;-----
;

```

```

; Constant registers
; c0-c3      - world space transposed
; c4-c7      - world * view * proj
; c8         - Spotlight1 position (In World Space)
; c9         - Eye position (In World Space)
; c10-c13    - Spotlight1 projection matrix
; c14        - Spotlight2 position (In World Space)
; c15-c18    - Spotlight2 projection matrix
;
; Input registers
; v0 - Position
; v1 - Normal
; v2 - Texture
; v3 - Tangent T
;
; Output
; oT0 - tex coord
; oT1 - Spotlight1 vector (in tangent space)
; oT2 - eye vector (in tangent space)
; oT3 - projective spotlight1 texture coordinates
; oT4 - Spotlight2 vector (in tangent space)
; oT5 - projective spotlight2 texture coordinates
;-----

vs.1.1

dcl_position v0
dcl_normal v1
dcl_texcoord v2
dcl_tangent v3

;-----
;Following code output position and texture coordinates
;-----
;
m4x4 oPos, v0, c4          ;vertex clip position
mov oT0.xy, v2.xy         ;Texture coordinates for color texture

;-----
;Following code generates Tangent space base vectors
;-----
;
m3x3 r11, v1, c0          ;use normal as third parameter of texture space
mov r11.w, v1.w
m3x3 r9, v3, c0           ;generate tangent SxN
mov r9.w, v3.w

mul r0,r9.zxyw,r11.yzxw   ;The cross product to compute binormal NxT
mad r10,r9.yzxw,r11.zxyw,-r0 ;The cross product to compute binormal NxT

m4x4 r8, v0, c0          ;Transform vertex into world position

;-----
;Following code computes light1 vector in texture space
;for pixel shader
;-----
;
add r0, c8, -r8          ;Build the light1 vector
                           ;normalize the light1 vector (d stands
for distance)
dp3 r1.w, r0, r0         ;r1.w = r0 * r0 = (x*x) + (y*y) + (z*z)

```

```

rsq r1.w, r1.w           ;r1.w = 1/d = 1/||V|| = 1/sqrt(r1.w)
mul r7, r0, r1.w        ;r7 = r0 * (1/d) = r0/d

dp3 oT1.x, r9, r7        ;transform the light1 vector with N, T, NxT
dp3 oT1.y, r10, r7       ;into texture coordinate space
dp3 oT1.z, r11, r7       ;for per pixel diffuse lighting

;-----
;Following code computes specular lighting parts
;for pixel shader (Eye vector)
;-----
;
add r0, c9, -r8          ;build the eye vector

dp3 r1.w, r0, r0         ;normalize the eye vector
rsq r1.w, r1.w
mul r6, r0, r1.w

dp3 oT2.x, r9, r6        ;transform the eye vector with S, T, SxT
dp3 oT2.y, r10, r6
dp3 oT2.z, r11, r6

;-----
;Following code computes light2 vector in texture space
;for pixel shader
;-----
;
add r0, c14, -r8        ;Build the light2 vector

dp3 r1.w, r0, r0         ;normalize
rsq r1.w, r1.w
mul r7, r0, r1.w

dp3 oT4.x, r9, r7        ;transform the light1 vector with N, T, NxT
dp3 oT4.y, r10, r7       ;into texture coordinate space
dp3 oT4.z, r11, r7       ;for per pixel diffuse lighting

;-----
;Following code output projective texture coordinates for spot 1
;-----
;
m4x4 oT3.xyzw, v0, c10   ;vertex transform to texture coordinates

;-----
;Following code output projective texture coordinates for spot 2
;-----
;
m4x4 oT5.xyzw, v0, c15   ;vertex transform to texture coordinates

```

- And the pixel shader:

```

;-----
;Pixel shader for 2 per pixel specular and diffuse spotlights in one pass
;-----
;
; Constant registers
; c0 - shininess (in all components)
; c1 - enable / disable light 2
;
; Used input registers

```

```

; t0 - color / bump coordinates
; t1 - light1 vector in tangent space
; t2 - eye vector in tangent space
; t3 - projector1 coordinates
; t4 - light2 vector in tangent space
; t5 - projector2 coordinates
;
; Used input texture stages
; stage0 - ambient texture
; stage1 - normal texture
; stage2 - specular light map 1
; stage3 - projector1 texture
; stage4 - specular light map 2
; stage5 - projector2 texture
;
; Output
; r0 - output color
;-----

```

ps.1.4

```

texld r1, t0                ;normal vector (n)
texcrd r2.rgb, t1.xyz       ;Light1 vector (l1)
texcrd r3.rgb, t2.xyz       ;eye vector (v)
texcrd r4.rgb, t4.xyz       ;Light2 vector (l2)

;Light 1
; r2.b - diffuse dot product
; r2 - specular lightmap coordinates
dp3 r0, r1_bx2, r2          ;r0 = dot(normal, light)
mad r2.rgb, r0_x2, r1_bx2, -r2 ;compute reflectance vector r2=2(n.l)n-1
dp3 r2.r, r2, r3            ;r2.xyz = dot(reflect, eye)
mov r2.g, r0                ;diffuse component is saved in r2.g

;Light 2
; r4.g - diffuse dot product
; r4 - specular lightmap coordinates
dp3 r0, r1_bx2, r4          ;r0 = dot(normal, light)
mad r4.rgb, r0_x2, r1_bx2, -r4 ;compute reflectance vector r4=2(n.l)n-1
dp3 r4, r4, r3              ;r2.x = dot(reflect, eye)
mov r4.g, r0                ;diffuse component is saved in r4.g

phase
;In the first phase, r2.g and r4.g components were filled with diffuse
intensities and r2.r, r4.r with specular look-up values.
;In the next phase, from look-up value, specular intensity will be
recovered. In normal case r2.g and r4.g will be rewritten, but we store
following function in the texture:
;f(r,g) = (r, g, pow(r, shi), pow(r, shi)). So we are able to "pass"
;g component across depended read.
;
;In the second phase, specular computations are done in alpha pipe
;parallel to diffuse computations in the rgb pipe (the + sign).
;This helps us to "enlarge" number of instructions to double.
;
texld r0, t0                ;load specular texture (gloss map is in
alpha channel)
texld r2, r2                ;specular light map 1.
texld r3, t3_dw.xyw        ;projector1 map
texld r4, r4                ;specular light map 2
texld r5, t5_dw.xyw        ;projector1 map

```

```

mul r2.rgb, r0, r2.g           ;diffuse 1
  +mul r2.a, r0.a, r2.a       ;specular 1
mul r4.rgb, r0, r4.g           ;diffuse 2
  +mul r4.a, r0.a, r4.a       ;specular 2
mul r4.rgba, r4, c1           ;disable / enable specular2

mul r0.rgba, r3, r2.a         ;specular1 * spotlight1
mad r0.rgba, r3, r2, r0       ;diffuse1 * spotlight1 + prev
mad r0.rgba, r5, r4.a, r0     ;specular2 * spotlight2 + prev
mad r0.rgba, r5, r4, r0       ;diffuse * spotlight2 +prev

```

6.4.6 Conclusion

Per-pixel shading is a great step ahead in the real-time computer graphics. The good thing is that it can be done on the fixed function hardware with GeForce1, GeForce2 and Radeon7500 cards to some extent (using DOT3 texture stage), so not only owners of newest hardware will see nice games.

Pixel shaders described in this chapter are very complex. For common game usage we do not need so many multiplier constants and we can save instructions and increase the overall speed. Another good approach is not to shade the entire scene per-pixel, but use the lightmaps for static parts, per-vertex shading for the distant and minor lights and for a few major and close lights we can use per pixel shading.

DirectX 8 class hardware lacks precision in pixel shader instructions and registers, so results are not as good as can be. Upcoming DirectX 9 offers larger shaders and flow control (so we can do more lights per pass) and full 32 bit floating point precision in pixel shader registers and instructions.

6.6 Literature

- [6.1] Möller T., Haines E., "Real - Time Rendering", A.K Peters, Natick, Massachusetts, 1999, <http://www.acm.org/tog/resources/rtr/> or <http://www.realtimerendering.com>
- [6.2] Cook, Robert, L., Kenneth E. Torrance, "A reflectance model for computer graphics", Computer graphics (SIGGRAPH '81 Proceedings), vol. 15, no. 3, pp. 307-316, July 1981
- [6.3] He Xiao D., Kenneth E. Torrance, Francois X. Sillion, Donald P. Greenberg, "A Comprehensive Physical Model For Light Reflection", Computer graphics (SIGGRAPH '91 Proceedings), vol. 25, no. 4, pp. 175-186, July 1991
- [6.4] Oren, Michael, Shree K. Nayar, "Generalization of Lambert's Reflectance Model", Computer graphics (SIGGRAPH '94 Proceedings), pp. 239-246, 1994. <http://www.cs.columbia.edu/~oren/>
- [6.5] Alan Watt, Mark Watt, "Advanced Animation and Rendering Techniques, Theory and Practice", ACM Press, 1992
- [6.6] Phong, Bui Tuong, "Illumination of Computer Generated Pictures", Communications of the ACM, vol. 18, no. 6, pp. 311-317, June 1975

- [6.7] Blinn, James F., "Models of Light Reflection for Computer Synthesized Pictures", ACM Computer graphics (SIGGRAPH '77), vol. 11, no. 2, pp. 192-198, July 1977
- [6.8] Gouraud, H., "Continuous shading of curved surfaces", IEEE Transactions on , vol. C-20, pp. 623-629, June 1971
- [6.9] Taylor, Philip, "Per-Pixel Lighting", MSDN Article, <http://msdn.microsoft.com/directx>, November 2001
- [6.10] Lengyel, Eric, "Mathematics for 3D Game Programming and Computer Graphics", Charles River Media Inc., pp 150 - 157, 2002
- [6.11] Wloka, Matthias, "Fresnel Reflection Technical Report", nVidia Corporation paper, <http://developer.nvidia.com>, June 2002
- [6.12] Turkowski, Ken, "Properties of Surface-Normal Transformations", in Andrew Glassner (editor), Graphics Gems, Academic Press, Inc., pp. 539-547, <http://www.worldserver.com/turk/computergraphics/index.html>, July 1990