Shader Programming:
An Introduction Using the Effect Framework

Jörn Loviscach [1]

Abstract: Current commodity graphics cards offer programmability through vertex shaders and pixel shaders to create special effects by deformation, lighting, texturing, etc. The Effect framework introduced by Microsoft allows to store shader program code, settings, and a limited graphical user interface within a single .fx text file. This supports a division of labor between programmers writing the code and designers using the GUI elements to control settings. Furthermore, the Effect framework proves to be ideal for experimenting with shader programming—be it for learning purposes or for rapid prototyping. In this tutorial, we employ the Effect framework for an exploratory, hands-on approach, introducing first principles only as needed, not in advance. Simple shader programs are used to review basic 3D techniques such as homogeneous coordinates and the Phong shading model. Then we turn to basic deformation effects employing vertex shaders and the use of texture maps as decals or reflected environments inside pixel shaders. To create bump mapping and related effects, tangent space coordinates and normal maps are introduced. Finally, we treat more complex effects such as anisotropic specular highlights.

Keywords: Pixel shader, Vertex shader, HLSL, Effect framework.

1 A First Glance at Shader Programming

Vertex shaders and pixel shaders of current 3D graphics chips offer a wide programmability to create stunning effects or simply to offload time-consuming tasks from the central processing unit. The effects range from creating the "look" of, for instance, velvet, marble or even human skin [1] to stylized outlines [2], genetically designed materials [3] or physical simulations of clouds [4].

While current graphics chips are programmable to some extent, their programming model remains restricted enough to allow parallel processing by a high number of functional units on the graphics chip. This explains much of the rapid growth of the computing performance of graphics chips in comparison to the—already impressive—performance growth of general-purpose microprocessors.

For instance, both the ATI Radeon X800 XT [5] and the Nvidia GeForce 6800 [6]

[1]Fachbereich Elektrotechnik und Informatik, Hochschule Bremen, 28199 Bremen, Germany
{jlovisca@informatik.hs-bremen.de}

contain 16 pixel pipelines, so that 16 pixels can be processed at the same time. Each of the pipelines contains two arithmetic logic units, which often work in parallel. These units process four-component floating point vectors, so that in the optimum case $16 \times 2 \times 4 = 128$ floating point computations happen at the same time in the pixel pipelines.

This tremendous degree of parallel processing requires—and justifies—restrictions in the programmability. Currently, attempts to cleverly circumvent these restrictions form a major source for algorithmic inventions in the computer graphics arena. While the programming model of the graphics hardware is steadily being extended, the need to allow parallel processing will continue to place severe restrictions.

When they first appeared, shaders had to be programmed in assembler code, which not only was difficult to read but also depended strongly on the specific hardware being used. In the meantime, high level shading languages such as Microsoft's HLSL ("High Level Shading Language") [7], Nvidia's Cg ("C for Graphics") [8], and the OpenGL SLang ("Shading Language") [9] have been introduced. Corresponding compliers can address a range of graphics hardware, using more complex instructions where available. This tutorial is based on HLSL, which to a large extent is identical to Cg.

Typically, vertex and pixel shaders are employed in conjunction with each other, often using several render passes with different settings for each pass. The Effect framework [7] introduced by Microsoft allows to store shader program code, settings, and a basic graphical user interface within a single .fx text file.

A programmer can develop the look of a class of materials as an .fx file, while a designer can adjust the material using the sliders, color controls, etc. described in the .fx file as user interface, see Fig. 1. Major 3D content creation software such as Alias Maya, discreet 3ds max, and Softimage XSI support a real-time preview of .fx shaders, mainly through special plug-ins. The final result will typically be used in real-time settings such as games equipped with .fx loaders. Microsoft has integrated an .fx loader into DirectX 9.0; Nvidia provides a free library called CgFX [8] with support for both OpenGL and DirectX.

2 Review of Basic 3D Techniques Using .fx

In addition to offering an improved work flow, the Effect framework allows effortless hands-on experiments. In this section we use the Effect framework to explore and review basic 3D techniques [10]. For the experiments, open a standard .fx file with Microsoft EffectEdit, which is contained in the DirectX SDK [7], or with Nvida FX Composer [11].

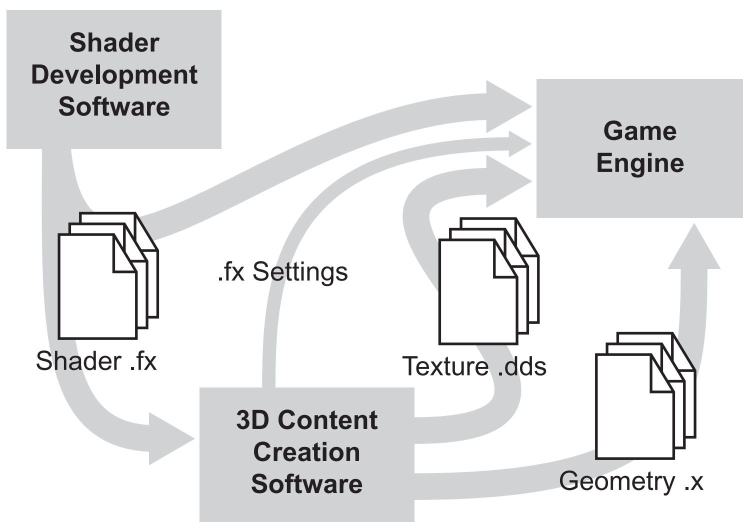

Figure 1. The Effect framework addresses the collaboration of programmers and designers.

2.1 Four-Component Vectors

A major part of all 3D computations can be unified in a scheme employing four-component vectors. In this tutorial, we follow Microsoft's programming model, which treats vectors as rows, not—as is typical in mathematics and in OpenGL—as columns. Thus, the product of a vector $\mathbf{x}$ and a matrix A is formed $\mathbf{x}A$ instead of $A\mathbf{x}$.

A four-component vector can be built both from a point in space $(x, y, z, 1)$ and from a direction $(n_x, n_y, n_z, 0)$. The value of 1 or 0 in the last component (called w) is not only a formal distinction, but later also serves to keep directions from being transformed by translations, see subsection 2.2: Under a translation, points should move, but not directions. In the context of perspective transformations (see subsection 2.2), the w component will play a more fundamental role. Note that in the context of DirectX, the z axis points inside the screen, so that its coordinate frame is left-handed, in contrast to the behavior of OpenGL before projection is applied.

RGB color triplets are cast into four-component vectors (r, g, b, a) by augmenting them with a fourth value, where a denotes alpha, the opacity. The value $a = 0$ denotes full transparency, $a = 1$ denotes complete opacity. In many computations a is ignored or simply set to 1. The values of r, g, b, too, range from 0 to 1, so that for instance the RGB triplet for

green $(0, 255, 0)$ is represented as $(0, 1, 0, 1)$. Since we are using floating point values, we no longer need to hardwire the resolution such as 256 steps into the range of the representation.

The basic data types of shaders are four-component vectors of either 16 or 32 bit floating point precision per component, called `half4` and `float4`, respectively. Pixel shaders on current ATI graphics chips will use 24 bits instead of 32. In addition, there are scalar or smaller vector types such as `float` or `half2`. Single components of vectors can be addressed via an expression like `foo.x = 4 * bar.g`. The suffixes `x`, `y`, `z`, `w` and `r`, `g`, `b`, `a` can be used interchangeably. The components of a vector may be "swizzled" on read: `float3 foo = bar.xzz`. When writing into a vector, some components may be left unchanged, such as `foo.z` in `foo.xyw = 3 * bar`.

2.2 Transformations, Homogeneous Coordinates

A linear transformation such as a rotation, a scaling, or a reflection can easily be cast into the four-component framework by appending 0's and a 1 to its 3×3 matrix:

$$\begin{pmatrix} \cdot & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & 0 \\ \cdot & \cdot & \cdot & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

However, we have to use the transposed form of the usual 3×3 matrices, because vectors will be multiplied with the above matrix as rows, i. e., from the left, not from the right.

In contrast to rotations etc., perspective projections cannot be written using matrices in the usual sense: Otherwise, they would be linear transformations, which would imply that they leave parallel lines parallel. But under a perspective projection parallel lines are transformed to rays aiming at a vanishing point. Only parallel lines that also are parallel to the projection plane will stay parallel, see Fig. 2.

The main reason to introduce four-component vectors is that also perspective transformations can be written as matrices—if one adopts the rule that as a final step the vector (x, y, z, w) has the to be converted back to 3D by division: $(x/w, y/w, z/w)$. This step justifies calling (x, y, z, w) homogeneous coordinates because a common non-zero factor will cancel. Not going into mathematical details let us remark how the division allows perspective foreshortening: w will grow with the distance from the viewer, thus reducing the size of distant objects through the division.

On the graphics card, the division by w happens outside the program code of vertex shaders and pixel shaders, see section 2.4. We can see that by using one of the example .fx files and multiplying the incoming 4D vertex position with a non-zero constant: Nothing is

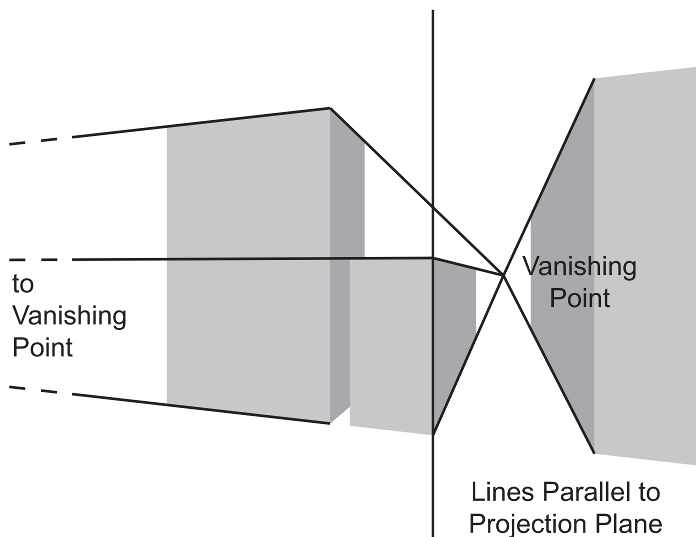

Figure 2. Linear perspective maps parallel lines that are not parallel to the projection plane to rays aiming at a vanishing point.

changed, because the division will cancel the common factor of (x, y, z) and w. To change the size of a 3D object, one has to multiply (x, y, z) alone or to divide w.

Using homogeneous coordinates, also translations can be written using 4×4 matrices:

$$(x, y, z, 1) \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & c & 1 \end{pmatrix} = (x + a, y + b, z + c, 1).$$

A direction $(n_x, n_y, n_z, 0)$, however, will be left unchanged by this matrix—which is excatly the behavior one expects of a translation.

Thus, transformations such as rotations, scalings, translations, and perspective projections, may now be written simply through matrices. Applying several transformations one after the other just requires multiplication with the product of the single matrices: Instead of computing the left hand side of

$$((\mathbf{x}A)B)C = \mathbf{x}(ABC),$$

for each point $\mathbf{x}$, one uses the right hand side of this equation, computes ABC once and thus saves two products of the type vector times matrix per point. The matrices are applied in the

written order: first A, then B, then C. Note that when using column vectors, the order of computation is reversed, because $\mathbf{x}$ will have to occur on the right: $(ABC)\mathbf{x} = A(B(C\mathbf{x}))$.

DirectX uses three matrices to control how object are transformed to the screen:

- `World` converts object coordinates to world coordinates: How is an object placed in the world? How is it oriented and scaled?
- `View` converts world coordinates to camera coordinates: How do we look at the world?
- `Projection` converts camera coordinates to normalized screen coordinates: Which kind of lens do we use?

Each point is subjected to all three matrices in series. In contrast, OpenGL uses only two matrices, combining `World` and `View` into `MODELVIEW`.

2.3 Back Face Culling, z-Buffer

Current graphics cards receive 3D objects as a soup of polygons—or in the case of DirectX even only triangles. These are painted into the frame buffer one after the other in the order given. This requires specific precautions such that for instance the back side of a box is not drawn over the front side. We can easily show what would happen by loading a simple .fx file, locating the relevant `technique` and the relevant `pass` (more on that in subsection 2.4), and changing its render settings to `ZEnable = false; CullMode = None;`. On rotating the 3D object in the view, back faces will appear over front faces.

One idea to stop this is to draw no back faces at all by setting `CullMode = CCW;`. This "back face culling" uses the winding direction of the polygons to decide if we look at a polygon from its front (counterclockwise winding) or from its back (clockwise winding). Of course the 3D model has to be built accordingly. You may also set `CullMode = CW;` in order to draw only the back faces, which can be helpful for complex shader techniques.

Depending on the 3D objects to be displayed, back face culling may or may not fully solve the visibility problem: Typical objects are non-convex, so that several front-facing polygons appear behind each other. The typical real-time solution that can cope with every kind of geometry is the z-buffer, invoked by `ZEnable = true;`. One may call it the most important functional unit of virtually any 3D graphics card built up to now. The main idea behind the z-buffer is to paint polygon after polygon, writing the depth per pixel into a non-displayed buffer (the z-buffer) having the same dimensions as the window on the screen. If the graphics card finds it will draw a pixel behind the depth current stored for it, this pixel will be left unchanged, see Figure 3. On typical hardware, the z-buffer will contain not the actual depth data, but values in the range $[0, 1]$ corresponding monotonously to the depth.

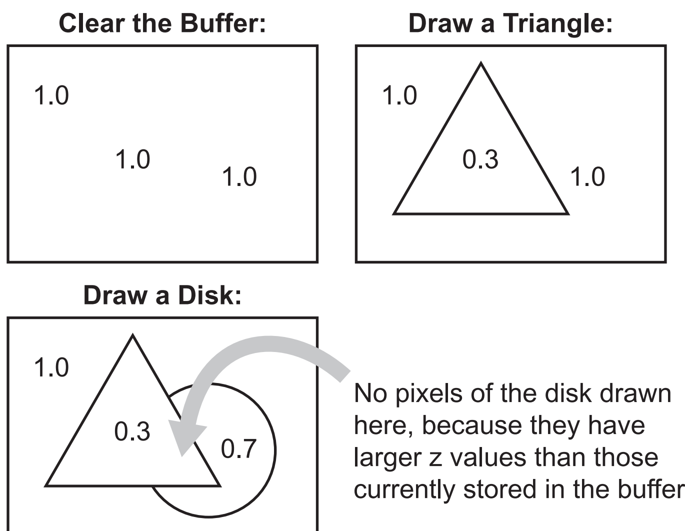

Figure 3. The z-buffer stores information about the depth of each pixel (values given in the drawing). This can be used to supress the drawing of geometry hidden beneath.

2.4 Real-Time Render Pipeline

To allow parallel processing like on an assembly line, the tasks of the 3D chip are broken down into several stages supported by corresponding subunits. The input typically consists of vertices that determine polygons; the output are colored pixels on the screen (i. e., in the frame buffer) and possibly values in additional buffers such as the z-buffer. Speaking very broadly, one can give the following sequence of tasks in the graphics chip:

1. Transform and Lighting: Transform the vertices from their object space into normalized screen and depth coordinates. Compute the lighting per vertex.
2. Perspective Divide: Convert from homogeneous coordinates to 2D plus depth.
3. Triangle Setup, Rasterization: Build triangles from the vertices and convert these to pixels.
4. Shading and Texturing: Compute the color per pixel using textures and interpolated lighting.
5. Depth Test: Compare the depth of the current pixel to that stored in the z-buffer. If the new depth is equal or nearer, store the pixel.

6. Alpha Blending: Mix the new pixel with the pixel currently present in the buffer.

This simplified representation of the render pipeline leaves out for instance back face culling and the clipping of triangles that otherwise would overlap the borders of the view port.

On most current graphics chips, the "configurable pipeline" may be switched to a "programmable pipeline":

1. **Vertex Shader:** For every vertex a program is executed.
2. Perspective Divide: Convert from homogeneous coordinates to 2D plus depth.
3. Triangle Setup, Rasterization: Build triangles from the vertices and convert these to pixels.
4. **Pixel Shader:** For every pixel a program is executed.
5. Depth Test: Compare the depth of the current pixel to that stored in the z-buffer. If the depth is equal or nearer, store the pixel.
6. Alpha Blending: Blend the new pixel with the pixel currently present in the buffer.

Thus, the vertex shader is responsible for transform and lighting. It can introduce deformations, compute lighting, colors or auxiliary parameters per vertex. The pixel shader is responsible for retrieving textures and for computing the pixel's color, which may be based on more or less complex reflection models etc.

To allow parallel processing, a vertex shader cannot access other vertices than the one for which it is invoked; a pixel shader cannot access other pixels. A vertex shader can neither create vertices nor delete them. (However, it can move them off screen or deform polygons so that they have zero area.) A pixel shader cannot create pixels or change the (x, y) position on the screen. However, it may discard the pixel.

Typical methods to overcome these and other restrictions are to add data to each vertex and/or to render intermediate results in off-screen buffers, which can later be read back as textures. A classic way to cope with restrictions of graphics cards is to render a 3D object several times but with different settings (and possibly different shaders), which mostly requires alpha blending, see subsection 6.2. In the .fx framework, passes can be invoked simply by adding further `pass` blocks inside a `technique` block. A `technique` is a complete single- or multipass shader. An .fx file can contain several `techniques` to store different materials or—rather—different implementations of the same material.

Here is a starting point for experiments with the .fx pipeline: In a typical vertex shader, the original 3D vertex position is converted into homogeneous coordinates and subjected to

the product of `World`, `View`, and `Projection` matrix. This may look similar to the following: On the topmost level of the .fx file the matrix is declared as to be delivered by the application using the .fx file: `float4x4 WVP : WorldViewProjection;`. The host application can use the supplied "semantic" `WorldViewProjection` to identify how to set this matrix. In the vertex shader the following line or an equivalent expression will appear: `float4 HPosition = mul(float4(IN.Position, 1.0), WVP);`, where `mul` is the built-in multiplication function. For an experiment, one may for instance use the `Projection` matrix alone, thus overriding rotations etc. applied in the viewer.

3 Phong Illumination Model, Phong Interpolation

The first step to create own shaders is to reproduce the operations executed by the classic, configurable render pipeline. These operations form the basis of most shaders.

3.1 Normal Vectors

A vertex shader *must* set the final position, which is expressed in homogeneous coordinates, as seen in the previous section. In addition, a vertex shader *may* compute lighting per vertex—which is exactly what would happen in the configurable render pipeline. To allow a computation of the lighting, each vertex typically does not only contain the data of the point $\mathbf{x}$, but also another vector-valued attribute: the outward-pointing surface normal vector, typically normalized to unit length.

In HLSL, the input to the vertex shader may be declared analogously to the following:

```
1. struct AppData
2. {
3.     float3 P : POSITION;
4.     float3 N : NORMAL;
5. };
6. VertexOutput myVertexShader(AppData IN)
7. {
8.    //...
9. }
```

The semantics `POSITION` and `NORMAL` signal the shader compiler which data provided by the application is to be used.

The per-vertex normal vector is given in object coordinates, whereas most computations happen in world coordinates. Thus, the normal vector has to be converted to the world

frame. If the `World` matrix simply consists of a rotation and a translation, it can directly be used for this transformation: The normal vector will be rotated in the same way as a point, and it will keep unit length. Setting its w component to zero will protect the normal vector from being influenced by the translation.

If, however, the `World` matrix is of different type, for instance a non-uniform scaling, it cannot directly be applied to the normal vector. To find the general rule of how the normal vector $\mathbf{n}$ has to be transformed, note that it is parallel to the vector product of two linearly independent vectors $\mathbf{a}$ and $\mathbf{b}$ that are tangent to the surface. If the surface is subjected to a linear mapping M, the normal vector of the resulting surface is parallel to $(M\mathbf{a})\times(M\mathbf{b})$. (We're using column vectors here to stick to usual mathematical notation.) A little linear algebra shows that this equals $\det(M)\,(M^{-1})^T(\mathbf{a}\times\mathbf{b})$, which is a multiple of $(M^{-1})^T(\mathbf{n})$. Thus, the normal vector has to be transformed by the inverse transposed world matrix and to be normalized again. (If M does not preserve orientation, the normal vector will also have to be inverted.) So we request from the application: `float4x4 WIT : WorldInverseTranspose;` and use this matrix to transform the normal:

```
float3 NWorld = normalize(mul(float4(IN.N, 0.0), WIT).xyz);
```

3.2 Parameters

To compute the lighting, we need to provide data about the material and the light source. For brevity we stick to a small number of parameters. In the .fx file we're describing a graphical user interface consisting of a color swatch, a slider, and four number fields for a vector. This information and additional descriptions are found in the angle brackets as a collection of "annotations", i. e., descriptive data to be retrieved through the Effect framework:

```
 1. float4 DiffuseColor : Diffuse
 2. <   string UIName = "Diffuse Color";
 3. > = {0.6, 0.9, 0.6, 1.0};
 4. float SpecularPower : Power
 5. <   string UIWidget = "Slider";
 7.     float  UIMin = 1.0;
 8.     float  UIMax = 128.0;
 9.     float  UIStep = 1.0;
10.     string UIName = "Specular Power";
11. > = 30.0;
12. float4 LightPosition : Position
13. <   string Object = "PointLight";
14.     string Space = "World";
15. > = {-1.0, 2.0, 1.0, 1.0};
```

A default value is assigned to every of these parameters, so that the application need not supply data—-but may do so. Typical .fx viewers can for instance connect the position `LightPosition` to an object in the 3D scene.

Note that not all currently available software conforms to the DirectX Standard Annotations and Semantics (DXSAS). Therefore, there still may be variations in the names to be used.

3.3 Phong Illumination Model in the Vertex Shader

Now we are ready to rebuild the Phong illumination model, which would be executed by the rendering pipeline when programmable shading is turned off. For simplicity we assume that the light and the specular highlights are white, that the light emits uniformly in all directions, and that there is no decay with distance. We do not want to apply ambient light (which would simply be an added color), so that in the Phong model there are two contributions to the final color on the screen: a diffuse and a specular component.

The diffuse component is proportional to the amount of light received by a small area of the surface around the current vertex. Assuming perfect diffusive behavior according to Lambert, the incident light is reflected uniformly into all directions of the hemisphere outside the object. Thus, if the vertex is visible at all, the diffuse component of its color does not depend on the position of the viewer. Its intensity is determined by the angle between the normal vector and the unit vector to the light source. If the light source is in the zenith, the diffuse component will attain its maximum, see Fig. 4.

To implement the Lambert model in the shader, we first need to compute a unit vector ("light vector") pointing from the current vertex to the light source. The position of the current vertex is given in object coordinates, but the position of the light source typically is given in world coordinates, so that we need to convert between the two coordinate frames. To this end, we request the `World` matrix from the application: `float4x4 W : World;`. Using the built-in functions `normalize`, `dot` and `max` we can compute the light vector and the diffuse components as follows:

```
1. float4 PObject = float4(IN.P, 1.0);
2. float3 PWorld = mul(PObject, W).xyz;
3. float3 LightVector = normalize(LightPosition.xyz - PWorld);
4. float3 NWorld = normalize(mul(float4(IN.N, 0.0), WIT).xyz);
5. float DiffuseIntensity = max(0.0, dot(LightVector,NWorld));
6. OUT.Color = DiffuseIntensity * DiffuseColor;
```

Several remarks are in order:

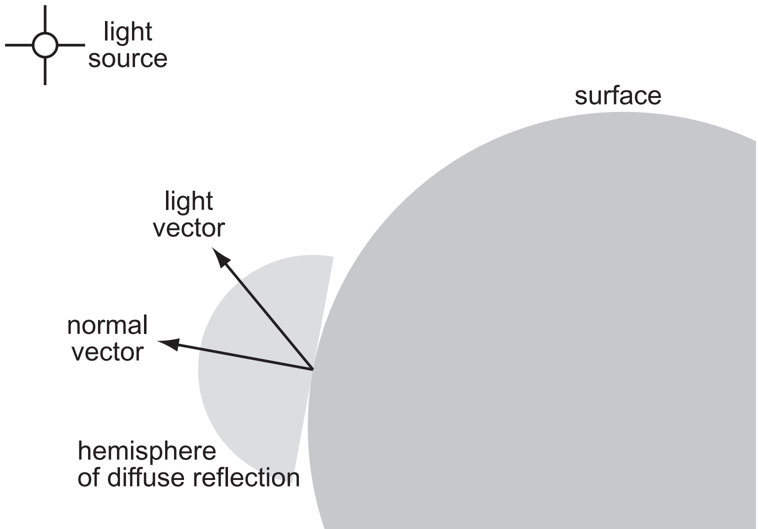

Figure 4. A perfectly diffusely reflecting surface will scatter the incident light uniformly over a hemisphere. The amount of incident light depends on the normal vector.

- The scalar product is computed by `dot`. The function `mul` applied to two vectors would compute a vector of component-wise products instead. This can for instance be used to filter a light color with a surface color.

- `max(0.0, dot(/*...*/))` clamps negative values of the scalar product to zero: If the scalar product is negative, the light source is behind the horizon of the vertex, so that its color should be zero. A negative color value would incur no problems by itself but would distort other color components such as ambient lighting applied to the same vertex in the same pass.

Now we turn to the specular component. It will form a lobe around the direction in which the light source would be perfectly mirrored in the surface. Therefore, the position of the viewer has to enter the computation. This position may be given simply as `float3` parameter. Alternatively, it can be read off from the `View` matrix: This matrix transforms the point where the viewer is located to the origin. Its inverse matrix reverses this and thus contains the position of the viewer in its last row. So we can ask for the inverse `View` matrix: `float4x4 VI : ViewInverse;` and compute a unit vector pointing from the vertex to the viewer: `float3 ViewVector = normalize(VI[3].xyz - PWorld);`.

A typical method to compute the intensity of the specular component is to introduce the “half vector”:

```
float3 HalfVector = normalize(ViewVector + LightVector);
```

If this vector points along the normal vector, we have a mirror-like situation. The more the half vector deviates from the normal, the more the viewer moves off the lobe of the specular reflection, see Fig. 5. This effect can be captured through the scalar product of both vectors. However, if you simply assign `OUT.Color = max(0.0, dot(HalfVector, NWorld));` you see a very broad light distribution. (Again, the `max` is used to cut off unwanted negative values.) So we need to conserve values near to 1 and suppress the smaller values. A simple way to do this is Phong’s exponentiation: Simply form a high power of the the scalar product:

```
OUT.Color=pow(max(0.0,dot(HalfVector,NWorld)),SpecularPower);
```

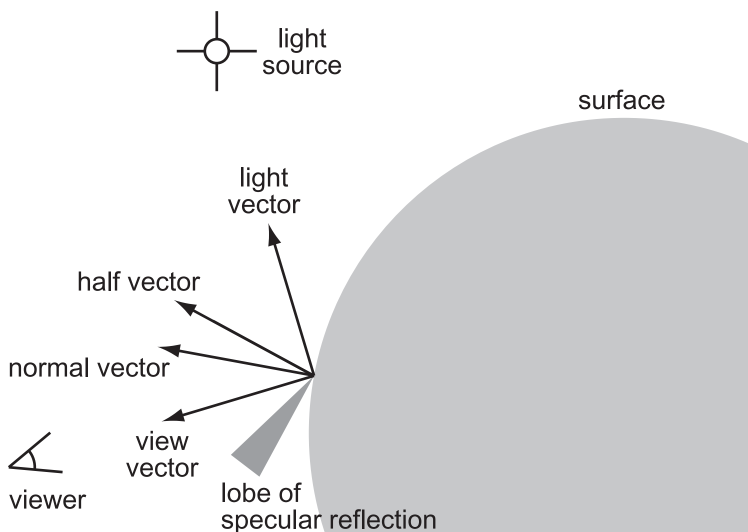

Figure 5. The more the “half vector” deviates from the normal, the more the viewing direction deviates from the direction of mirror-like reflection.

Finally, add the diffuse color component computed before instead of replacing it, by writing `OUT.Color += pow(/*...*/)`. In the highlight, some components of this sum

will be larger than 1. The graphics card deals with such color values that are above one or below zero by clamping them to one or zero, respectively.

The function `lit` of HLSL combines many of these computational steps into the following:

```
1. float4 lighting = lit(dot(LightVector, NWorld),
2.                       dot(HalfVector, NWorld), SpecularPower);
3. OUT.Color = DiffuseColor*lighting.y + lighting.z;
```

Furthermore, `lit` suppresses the specular term when the normal points away from the light. Otherwise, broad highlights may spill into the shadowed part of a surface.

3.4 Phong Illumination Model in the Pixel Shader, Phong Interpolation

The higher the Phong exponent, the smaller the highlights—and the more they turn into a polygonal shape revealing the underlying geometry of triangles. This effect is due to the bilinear (Gouraud) interpolation of the color values computed for the vertices: The complete lighting computation is only executed in the vertex shader; the actual colors for the single pixels are formed by blending the resulting vertex colors. A small highlight completely contained inside a triangle would not show up at all because it is not present at the vertices.

A major improvement can be achieved by moving the lighting computation into the pixel shader—at the cost of running this computation hundred or thousand times more often. Now, the vertex shader will not deliver a color to the pixel shader, but basic geometric properties like normal, view and light vector, which it typically converts from the object to the world coordinate frame.

Like it did with the color before, the graphics chip will bilinearly interpolate these data so that every pixel receives an interpolated normal etc. This “Phong interpolation” results in much improved results as compared to the interpolation of colors (“Gouraud interpolation”), see Fig. 6. Virtually all offline 3D renderers employ Phong interpolation; with pixel shaders one can achieve this quality in real time. (Note that Phong interpolation and the Phong illumination model may be used separately from each other, as is the case in the standard configurable render pipeline, which only employs Gouraud interpolation.)

Data is delivered from the vertex shader to the pixel shader using a common data structure such as `VertexOutput` in the following:

```
1. struct VertexOutput
2. {
3.    float4 HP : POSITION;  // homogeneous position
```

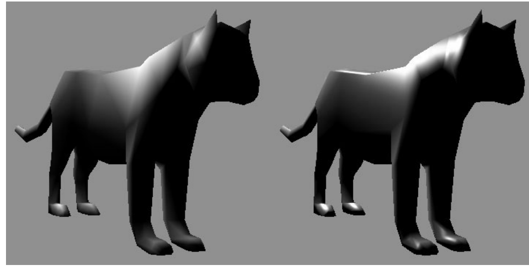

Figure 6. Computing the lighting model per pixel instead of interpolating the color computed for the vertices leads to improved highlights.

```
 4.    float3 N  : TEXCOORD0; // normal vector
 5.    float3 V  : TEXCOORD1; // view vector
 6.    float3 L  : TEXCOORD2; // light vector
 7. };
 8. struct PixelOutput
 9. {
10.    float4 Color : COLOR;
11. };
12. VertexOutput myVertexShader(AppData IN)
13. {
14.     VertexOutput OUT;
15.     //...
16.     return OUT;
17. }
18. PixelOutput myPixelShader(VertexOutput IN)
19. {
20.     PixelOutput OUT;
21.     //...
22.     return OUT;
23. }
```

Semantics like `POSITION` are used to declare how the single members are to be delivered. Here, the data fields of the classic rendering pipeline are to be used. So there is `POSITION`, which must be written in the vertex shader, but may on most current graphics cards not be read in the pixel shader. `COLOR0` and `COLOR1` can be written to and read back; however, these values may be delivered in eight bit precision to the pixel shader and may be clamped to the range $0 \ldots 1$. At this stage behind the vertex shader there is no `NORMAL`. Most

parameters end up in a set of texture coordinates `TEXCOORD`*n*.

The form of the .fx file suggests that the pixel shader receives the data the vertex shader has written. But this is not quite true: The pixel shader receives a bilinear interpolation of the values computed in the vertex shader. This can cleverly be exploited to simplify shaders.

In the vertex shader, we now compute geometric quantities:

```
 1. VertexOutput myVertexShader(AppData IN)
 2. {
 3.     VertexOutput OUT;
 4.     float4 PObject = float4(IN.P, 1.0);
 5.     OUT.HP = mul(PObject, WVP);
 6.     float3 PWorld = mul(PObject, W).xyz;
 7.     OUT.L = normalize(LightPosition - PWorld);
 8.     OUT.N = normalize(mul(float4(IN.N, 0.0), WIT).xyz);
 9.     OUT.V = normalize(VI[3].xyz - PWorld);
10.     return OUT;
11. }
```

The Phong illumination model can be evaluated from the interpolated geometric quantities in the pixel shader:

```
 1. PixelOutput myPixelShader(VertexOutput IN)
 2. {
 3.     PixelOutput OUT;
 4.     float3 N = normalize(IN.N);
 5.     float3 V = normalize(IN.V);
 6.     float3 L = normalize(IN.L);
 7.     float3 H = normalize(V + L);
 8.     float4 lighting = lit(dot(L,N),dot(H,N),SpecularPower);
 9.     OUT.Color = DiffuseColor*lighting.y + lighting.z;
10. }
```

For this to work, you have to make sure that at least Shader Model 2.0 is used: The compiler line in the pass structure of the .fx file should read:

```
PixelShader = compile ps_2_0 myPixelShader();
```

The normalization on lines 4, 5, and 6 is required because the bilinearly interpolated vectors will typically no longer possess unit length. But it is worth a try to skip the normalization.

Especially for slowly varying vectors the difference may be so small that it is virtually invisible. Another point left to optimization is to only compute the specular component inside the pixel shader, not the diffuse component.

4 Basic Shader Effects

The typical use of vertex shaders is to deform models, while pixel shaders largely work with textures. In this section we introduce these basic operations.

4.1 Deformation

Inside a vertex shader, deformation effects can simply be achieved by changing the position of the vertex before subjecting it to the `WorldViewProjection` matrix or—for a deformation occurring in world coordinates—the `ViewProjection` matrix. For instance, one may form

```
float3 p = IN.P + float3(sin(IN.P.y), 0, 0);
```

to move vertices to the left or to the right, depending on their height.

However, also the per-vertex normal vectors have to be deformed correspondingly. Otherwise the lighting will be wrong, see Fig. 7. If the deformation acts by transforming the 3D position $\mathbf{x}$ to $\mathbf{f}(\mathbf{x})$, the normals should be transformed [12] by $(J^{-1})^T$, where J is the Jacobian matrix of $\mathbf{f}$, i. e., the matrix formed by all partial derivatives of the three components of $\mathbf{f}$.

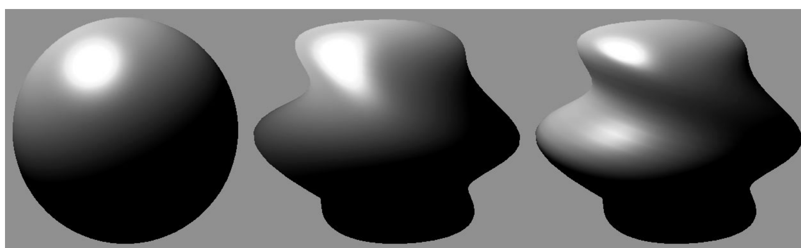

Figure 7. A deformation effect may not only change the positions of the vertices (left: original, mid: deformed) but must also adjust the normals to allow correct lighting (right).

Let us momentarily switch to column vectors in order to stick to the usual mathematical notation. Then for the example give above, we have $f(x, y, z) = (x + \sin(y), y, z)$, so

that the Jacobian matrix is

$$\begin{pmatrix} \partial f_x/\partial x & \partial f_x/\partial y & \partial f_x/\partial z \\ \partial f_y/\partial x & \partial f_y/\partial y & \partial f_y/\partial z \\ \partial f_z/\partial x & \partial f_z/\partial y & \partial f_z/\partial z \end{pmatrix} = \begin{pmatrix} 1 & \cos(y) & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$

The inverse transpose of J is

$$\begin{pmatrix} 1 & 0 & 0 \\ -\cos(y) & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix},$$

which means that the normal vector has to be transformed according to

```
float3 n = IN.N + float3(0, -cos(IN.P.y)*IN.N.x, 0);
```

and to be normalized again.

Deformers of this kind are typically used to produce twisting, tapering, melting, or similar effects on an entire 3D object. For character animation another type of deformation is important: skinning, i. e., controlling a mesh through a skeleton of bones, which exert local forces. The deformation caused by a single bone can be described similar to the `World` matrix. However, now we need a palette of matrices, one for each bone. Each vertex carries information about which bone or which blend of bones is used to deform it.

4.2 Texture maps

To use a texture inside an .fx file, first the texture and the sampler have to be configured:

```
 1. texture DiffuseTexture : Diffuse
 2. <
 3.     string ResourceName = "texture.dds";
 4. >;
 5. sampler DiffuseMap = sampler_state
 6. {
 7.     texture = <DiffuseTexture>;
 8.     MAGFILTER = LINEAR;
 9.     MINFILTER = LINEAR;
10.     MIPFILTER = LINEAR;
11. };
```

This will load a texture file named texture.dds and assign it to a texture sampler employing MIP mapping (i. e., blending multiple levels of detail to prevent artifacts). Files in the .dds format of DirectX can be prepared for instance with DxTex of the Microsoft DirectX SDK [7] or Nvidia's texture tools [13].

The most basic way to use textures is to wrap an image around 3D objects using uv texture coordinates: To every vertex the information is added, which point (u, v) of the image shall be mapped onto this vertex ($u = 0 \ldots 1$, $v = 0 \ldots 1$), see Fig. 8. As usual, values are assigned to the single pixels through bilinear interpolation of the values at the vertices.

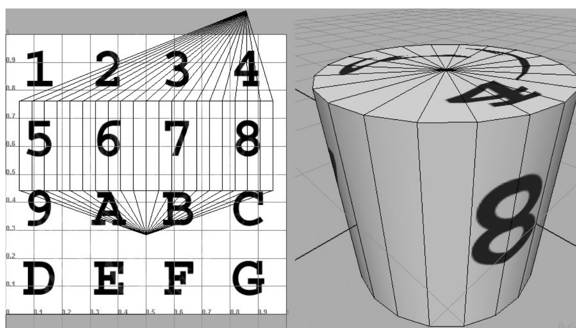

Figure 8. Texture coordinates are used to wrap a picture (left) around the surface (screenshot from Alias Maya).

We need to receive uv data in the vertex shader:

```
struct AppData
{
    //...
    float2 UV : TEXCOORD3;
};
```

The vertex shader simply hands over these data to the bilinear interpolation occurring before the pixel shader:

```
struct VertexOutput
 1. {
 2.    // ...
 3.    float2 UV : TEXCOORD3;
 4. };
```

```
 5. VertexOutput myVertexShader(AppData IN)
 6. {
 7.     VertexOutput OUT;
 8.     // ...
 9.     OUT.UV = IN.UV;
10.     return OUT;
11. }
```

Now we are ready to retrieve texture values in the pixel shader. These consist of the texture image's RGBA values converted to four-component vectors.

```
PixelOutput myPixelShader(VertexOutput IN)
{
    //...
    float4 t = tex2D(DiffuseMap, IN.UV);
    //...
}
```

A first experiment would be to use the retrieved value directly as output color. An advanced method would be to use two texture maps, one controlling the diffuse color in the Phong model and one controlling the intensity of highlights, thus adding wet spots to a model's surface.

5 Bump Mapping

One of the most visually effective uses of pixel shaders is bump mapping: Fine structures like engraved letters or wood veins are applied to a surface by deforming not its actual geometry but the per-pixel normals, see Fig. 9. This allows to depict fine-scale geometry while still working with relatively few polygons. Of course, the method fails for structures like mountains on a meteorite, where large deformations are needed for instance to let the silhouette look deformed, too.

5.1 Tangent Space Coordinates

Small deformations of the normal vector field are best described in a coordinate frame that is locally adjusted to the object's surface: The z axis points in the direction of the original normal, the x and y axes point along the tangent plane. To describe this moving coordinate frame, we equip each vertex not only with `NORMAL` data, but also with `TANGENT` and `BINORMAL` data. These form an orthogonal frame at each vertex, and are to be delivered by the host application:

Figure 9. A bump map (upper left) is converted to a normal map (lower left: separated RGB channels), which is used to deflect the per-pixel normal from its original direction (mid) to let a relief appear on the surface (right).

```
1. struct AppData
2. {
3.     float3 P  : POSITION;
4.     float3 N  : NORMAL;
5.     float3 T  : TANGENT;
6.     float3 B  : BINORMAL;
7.     float2 UV : TEXCOORD0;
8. };
```

How these additional vectors can be generated will be treated in subsection 5.3.

The vertex shader transforms these three vectors from object space to world space:

```
 1. struct VertexOutput
 2. {
 3.     float4 HP : POSITION;
 4.     float2 UV : TEXCOORD0;
 5.     float3 N  : TEXCOORD1;
 6.     float3 T  : TEXCOORD2;
 7.     float3 B  : TEXCOORD3;
 8.     float3 V  : TEXCOORD4;
 9. };
10. VertexOutput myVertexShader(AppData IN)
11. {
```

```
12.     VertexOutput OUT;
13.     float4 PObject = float4(IN.P, 1.0);
14.     OUT.H = mul(PObject, WVP);
15.     OUT.N = normalize(mul(float4(IN.N, 0.0), WIT).xyz);
16.     OUT.T = normalize(mul(float4(IN.T, 0.0), W).xyz);
17.     OUT.B = normalize(mul(float4(IN.B, 0.0), W).xyz);
18.     float4 PWorld = mul(PObject, W);
19.     OUT.V = normalize(VI[3].xyz - PWorld.xyz);
20.     OUT.UV = IN.UV;
21.     return OUT;
22. }
```

The graphics chip will interpolate the `NORMAL`, `TANGENT` and `BINORMAL` data and feed those into the pixel shader. Precise perpendicularity will be lost due to the interpolation and in the case of a non-uniform scaling by transforming the binormal vector with the `World` matrix, but typically this does not lead to visually objectionable errors.

5.2 Normal Maps and Environment Maps

Inside typical 3D offline-rendering software, bump maps are given as grey-scale images that describe the intended offset from the actual geometry: White means shift outward along the normal, black means shift inward. The deformed normal vector can be computed from the bump map, see section 5.3.

To gain speed, we need to compute the deformed normals in advance and store them as a "normal map". This is attached to the surface via texture coordinates. Technically being a regular texture, the normal map can only store values between 0 and 1; the components of a normal vector range, however, between -1 and 1, so that one has to rescale them: The normal (n_x, n_y, n_z) is stored as pseudo-color $(r, g, b) = 0.5(n_x, n_y, n_z) + (0.5, 0.5, 0.5)$ inside the normal map. Here, n_x, n_y, and n_z refer to the tangent frame, which means that a non-deformed normal is represented by $(0, 0, 1)$ and stored as the color $(0.5, 0.5, 1.0)$. Thus, typical normal maps contain lots of bluish pixels: The normal is seldom deformed strongly.

The pixel shader retrieves the pseudo-color values from the normal map, reverses the scaling and uses the `NORMAL`, `TANGENT` and `BINORMAL` data to compute the deformed normal vector in world coordinates:

```
 1. PixelOutput myPixelShader(VertexOutput IN)
 2. {
 3.     PixelOutput OUT;
 4.     float3 N = normalize(IN.N);
```

```
 5.     float3 T = normalize(IN.T);
 6.     float3 B = normalize(IN.B);
 7.     float3 NObj = 2.0*tex2D(NormalMap, IN.UV).xyz - 1.0;
 8.     float3 NWorld = normalize(NObj.x*T+NObj.y*B+NObj.z*N);
 9.     //...
10.     return OUT;
11. }
```

To achieve an impressive visual effect, the deformed normal vector can be used to create the look of dented chrome. This requires fake reflections, since we do not want to do ray tracing on the graphics chip. The usual solution is to load a "cube map" showing an environment projected to the six faces of a cube:

```
 1. texture EnviTexture : Environment
 2. <
 3.     string ResourceName = "nvlobby_cube_mipmap.dds";
 4.     string TextureType = "Cube";
 5. >;
 6. sampler EnviMap = sampler_state
 7. {
 8.     texture = <EnviTexture>;
 9.     MINFILTER = LINEAR;
10.     MAGFILTER = LINEAR;
11.     MIPFILTER = LINEAR;
12. };
```

Then we can add the following line to the pixel shader:

```
OUT.Color = texCUBE(EnviMap, reflect(-IN.View, NWorld));
```

Here, the vector `-IN.View` from the viewer is reflected (another built-in function) on the local bumped tangent plane. The resulting direction is used to retrieve a color from the environment map. Thus, the environment map is treated like wallpaper glued to an infinitely large cubic room. Untrained observers note neither the slight error in the mirror image resulting from this approximation nor the circumstance that self-reflections are missing.

5.3 Computation of Tangent Vectors and Normal Maps

Given a smooth parameterized surface $(u, v) \mapsto \mathbf{p}(u, v)$, one can easily find a (non-normalized) tangent vector field as $\partial\mathbf{p}(u, v)/\partial u$. However, in the context of current 3D

graphics chips we deal with polyhedra, not with smooth surfaces. One can approximate the directional derivative along the local u direction using a least-square fit. From that results a approximate tangent vector at each vertex. Then, the binormal vector can for instance be found by forming the vector product of the normal vector and the tangent vector. Ready-to-use functions to compute tangent and binormal vectors are for instance available in the application programming interface of Microsoft DirectX 9.0.

Normal Maps virtually cannot be generated manually. What *can* be drawn by hand are bump maps, i. e., height fields $f(u, v)$ where the gray level of a pixel indicates its displacement along the normal. One can simply form pixel-wise differences along the u and and the v directions to find how the normal vector has to be bent: For instance, if a dark gray pixel is lying left to a medium gray pixel, the normal is inclined towards the left side. This information can be converted to pseudo-colors and stored in a normal map. Software tools [13] are available for this job. Thus, the use of tangent space coordinates allows to fake deformations without reference to the actual geometry of the object. Furthermore, the use of tangent space coordinates easily allows to scale the normal values in such a way that the normal map exhausts the full range of eight bits per channel of standard textures. This proves to be especially important with environment mapping, where the reflection tends to produce objectionable artifacts from even small steps in the normal map. This problem disappears with special texture formats that store 16 or 32 bits per channel.

6 Complex Shader Effects

6.1 Using Textures to Store Arbitrary Functions

As we have seen, textures may be used to store how the color of a material depends on the position on the surface, how the environment looks like, and how the surface is deformed on a small scale. Many more relationships cannot easily be expressed in analytical form and thus are candidates for 1D, 2D, 3D, or cube maps. In addition, many functions that *can* be expressed in analytical form are better precomputed as textures to reduce the computatioal load.

As first example of how to achieve complex results through textures as tabulated functions, consider the following extension of the per-pixel Phong lighting model. In the corresponding pixel shader, we may write:

```
1. float3 N = ... // normalized normal vector
2. float3 V = ... // normalized view vector
4. float3 L = ... // normalized light vector
5. float3 H = normalize(V + L); // normalized half vector
6. float VdotN = dot(V, N);
```

```
7. float HdotN = dot(H, N);
8. OUT.Color = tex2D(LightingModel, float2(VdotN, HdotN));
```

Here, the texture bound to `LightingModel` controls the "look" of the surface. It can be computed to exactly reproduce the Phong illumination model. Much more interesting and/or realistic behavior can be achieved by painting an image to be used for this map, see Fig. 10. Furthermore, we see that the texture saves computation. For instance, the `pow` function is no longer invoked. We even do not need to clamp negative values of `VdotN` and `HdotN`, because the texture sampler can do that automatically.

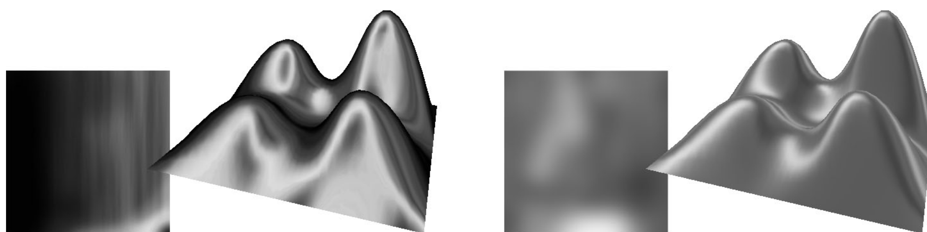

Figure 10. Using textures (square insets) to store arbitrary functions can both save computation time and add to the realism.

As another example of a not-so-obvious use of textures we present a shader for anisotropic highlights. These are present on shiny materials that possess a directional structure on a microscopic scale such as brushed aluminum, fur, or textile fabrics. The texture used as function table resembles an image of the highlight. For instance, if the texture contains in its center an horizontal ellipsis painted white on a dark background, the highlight extends along the u direction of the surface, see Fig. 11.

One way to achieve this is to compute the components of the half vector in the `TANGENT` and the `BINORMAL` direction. If both are zero, we have a mirror-like situation. Thus, these two components form a kind of local coordinate frame around the highlights.

```
1. float HdotT = dot(H, T);
2. float HdotB = dot(H, B);
3. OUT.Color = dot(L, N)* tex2D(HighlightModel,
4.              0.5*float2(HdotT, HdotB) + float2(0.5, 0.5));
```

The texture addressing is shifted in order to put the highlight at the center $(u, v) = (0.5, 0.5)$. The dot product of the light vector and the normal is introduced because otherwise also faces

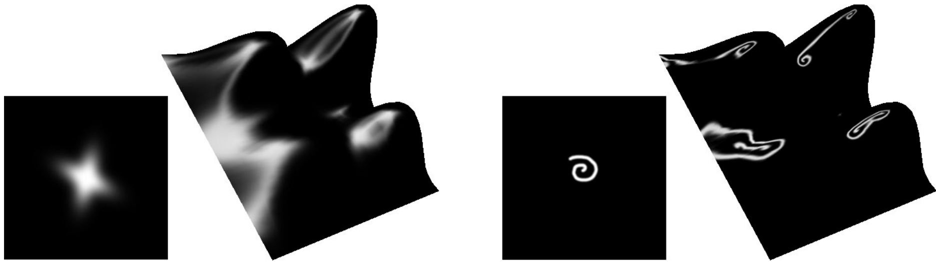

Figure 11. A texture (square insets) can be used to control the shape of anisotropic highlights.

pointing away from the light would be illuminated spuriously. In addition, this factor softly cuts off the highlight at the horizon.

6.2 Alpha Blending, Multiple Render Passes

Finally, we give meaning to alpha, the fourth component of all color vectors (r, g, b, a). It may control how strongly the currently computed color will be blended with the color already present at this pixel location. For a first experiment, try a pixel shader producing the output color `OUT.Color = float4(0, 1, 0, 0.5);` and use the following settings inside the `pass`:

```
1. AlphaBlendEnable = true;
2. SrcBlend = SrcAlpha;
3. DestBlend = InvSrcAlpha;
4. BlendOp = Add;
5. ZEnable = true;
6. ZWriteEnable = false;
7. CullMode = None;
```

Lines 2, 3, and 4 configure the operation used for blending. Here we compute a weighted sum (blend operation) of the new color (source) with the color already present (destination):

$$c_{\text{dest}} \leftarrow a_{\text{src}} c_{\text{src}} + (1 - a_{\text{src}}) c_{\text{dest}}$$

The new alpha acts as weight: If it equals one, the new color will completely replace the existing color, which corresponds to full opacity. If it equals zero, the pixel will be left unchanged, which corresponds to full transparency.

Line 5 says that we want to execute the depth test, so that no pixels are touched that lie in front of our current object. Line 6, however, switches off the writing into the z-buffer, and line 7 switches off back face culling. So all faces of the current 3D object will be rendered—if they are not hidden behind other 3D objects that have been drawn *before* `ZWriteEnable` was switched off. This is a typical setting for semi-transparency effects.

Alpha blending allows to break up complex computations into several rendering passes:

```
 1. technique t0
 2. {
 3.     pass p0
 4.     {
 5.         \\...
 6.     }
 7.     pass p1
 8.     {
 9.         \\...
10.     }
11. }
```

For instance, each pass may add the contribution of a different light source. While this would be more time-consuming than a single-pass solution, it may demand less capabilities from the hardware, especially in terms of the shader's program length.

Furthermore, some effects employ several different geometric deformations, and thus cannot be realized within a single pass. A classic example for this is the simple glow effect: First, render the 3D object as usual; second, render it in a bright color with medium alpha and using a vertex shader to expand the geometry along the normal vector.

7 Conclusion and Outlook

We have introduced the programming model of the current programmable graphics pipeline using hands-on experiments to introduce basic methods and tools of the trade. In this final section, we outline advanced methods and give directions for future work.

7.1 Advanced Features and Effects

The next major step in using the Effect framework is to use it in self-developed software. With Microsoft's DirectX 9.0 or Nvidia's CgFX toolkit this merely boils down to code like the following:

```
1. int numPasses = myEffect.Begin();
2. for(int i = 0; i < numPasses; i++)
3. {
4.     myEffect.BeginPass(i);
5.     myMesh.DrawSubset(0);
6.     myEffect.EndPass(); // added in DX 9.0c
7. }
8. myEffect.End();
```

For own experiments, Managed DirectX [14], which is a .NET wrapper of the DirectX API, may prove helpful. It allows to use virtually all advanced features of 3D graphics cards using powerful classes under C++ or C#, the Java cousin introduced by Microsoft.

Employing vertex and pixel shaders in one's own software secures more control about what is rendered into which buffer. For instance, one way to create shadows is to render the depth of the 3D scene seen from the view of the light source into an off-screen buffer. This buffer is then retrieved as texture ("shadow map") inside a rendering pass on the frame buffer. If a point is more distant from the light source than the minimum distance on this ray (as stored in the off-screen buffer), it receives no light. Other effects demanding off-screen buffers include soft glow and depth-of-field.

None of the shaders covered so far included loops such as `for` and `while`, branching such as `if ... else`, or subroutines. We did not even use integer variables. Obviously, many features of typical programming languages are not crucial to shading. Current graphics cards do not fully support loops. When branching over a piece of code, most of them will take as much time as if they actually executed every single statement in between. Even the latest graphics cards use float variables to emulate integer types.

For an overview of the zoo of "Shader Models" see the DirectX documentation [7]. The shader version also appears in the `compile` commands in an .fx file. For instance, `compile ps_2_0 MyPixelShader();` asks the Effect framework to compile the shader using the capabilities of a graphics card with pixel shader version 2.0.

7.2 Ideas for Future Work

Two main directions offer themselves for future work:

First, one can try to convert each and every classic effect into a shader program—or try to invent new effects or new shader algorithms from scratch. Given the major restrictions of the programming model, this requires some ingenuity. However, graphics hardware is a moving target. What is difficult today can become trivial with tomorrow's chips. For instance, some of the newest graphics cards allow to retrieve textures inside a vertex shader, whereas

previously textures were only accessible from pixel shaders.

Second, one can try to improve the workflow. Working with shaders will largely turn from engineering into a less technical activity, like designing and animating 3D objects did long ago. How can designers be supported best? This requires, for instance, effortless debugging, good integration of shaders into 3D content creation software, and the support of a broad range of hardware and software platforms, notably those based on OpenGL.

Inside of FX Composer, the .fx file format is already amended by instructions to render into off-screen buffers and use them as textures inside a later pass [11]. The next version of FX Composer and upcoming software of Microsoft will even take further steps towards integrating more complex effect structures [15]. However, the question arises, if .fx is becoming overburdened with these extensions, so that one should better develop a new language instead.

References

[1] C. Beeson and K. Bjorke, Skin in the “Dawn” Demo, in: R. Fernando, ed., GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics, Addison-Wesley 2004, pp. 45–62

[2] M. McGuire and J. F. Hughes, Hardware-Determined Feature Edges, Proceedings of NPAR 2004, pp. 135–147

[3] J. Meyer-Spradow and J. Loviscach, Evolutionary Design of BRDFs, Eurographics 2003 Short Paper Proceedings, pp. 301–306

[4] M. J. Harris, Fast Fluid Dynamics Simulation on the GPU, in: R. Fernando, ed., GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics, Addison-Wesley 2004, pp. 637–665

[5] ATI, Radeon X800 Architecture, http://www.ati.com/products/radeonx800/RADEONX800ArchitectureWhitePaper.pdf

[6] Nvidia, Technical Brief: The GeForce 6 Series of GPUs, http://www.nvidia.com/object/IO_12394.html

[7] Microsoft DirectX 9.0 SDK and Documentation, http://www.microsoft.com/directx

[8] Nvidia Cg, CgFX library, http://developer.nvidia.com/object/cg_toolkit.html

[9] OpenGL Shading Language, http://www.opengl.org/

[10] P. Shirley, Fundamentals of Computer Graphics, AK Peters 2002

[11] Nvidia FX Composer, http://developer.nvidia.com/object/fx_composer_home.html

[12] E. d'Eon, Deformers, in: R. Fernando, ed., GPU Gems: Programming Techniques, Tips, and Tricks for Real-Time Graphics, Addison-Wesley 2004, pp. 723–732

[13] Nvidia Normal Map Plug-in, http://developer.nvidia.com/object/nv_texture_tools.html

[14] T. Miller, Managed DirectX Kick Start: Graphics and Game Programming, Wordware 2003

[15] C. Maughan and D. Horowitz, FX Composer 1.5—Standardization, to appear in: W. Engel, ed., ShaderX^3, Wordware 2004