Introduction to shader programming

By Taco Cohen

Last updated October 14, 2006

Version 1.01

Introduction

In recent years, more and more emphasis has been put on the graphical aspect of video games. This has led to the advent of programmable vertex and pixel shaders in 2001. Before that, rendering of 3D graphics was handled by a number of fixed graphics algorithms collectively called the fixed function pipeline.

In the fixed function pipeline, the only way to influence the rendering of 3D objects is by setting a number of parameters. Vertex and pixel shaders, on the other hand, give the programmer full control over how the objects in the 3D world are rendered, making it possible to create a visually unique game. Some examples of popular shading effects are bump/normal mapping, reflective/refractive water surfaces, toon shading and several post-processing effects like depth of field and bloom.

In this tutorial we will take a look at how vertex and pixel shaders work. We will start by examining the rendering pipeline and see where the vertex and pixel shaders come into play. After that we will build a simple lighting model in Microsoft's higher level shading language (HLSL). We will start with simple ambient lighting and later add diffuse light, specular highlights and normal mapping. I will explain the syntax details as we go. Fortunately, HLSL syntax is very similar to C/C++ and to a lesser extent to 3DGamestudio's C-Script. Finally we will learn how to incorporate the shader in a project in 3DGamestudio.

This tutorial is geared towards total novices to shader programming. However, a basic knowledge of calculus and trigonometry is required. Knowledge of vectors is also required, but there is an overview of vectors, matrices and transformations in appendix A. If you are not familiar with vector math or need to be refreshed, I strongly suggest you take a look at it first. Although I will explain the HLSL language peculiarities as we go, you will need some programming experience in any c-like language. Lastly, you need to be familiar with some of the terminology used in game development.

Let's get started!

Section I

1.1 D3D Pipeline

Before we take a look into shaders, we need to understand how the input data is manipulated to create the illusion of 3D graphics.

The application (the 3D engine) sends vertex data to the rendering API. Most Windows based 3D engines use Microsoft's Direct3D (D3D) API and so does A6.

D3D performs a series of operations on its input data that result in the final rendered image. **Figure 1.1.1** shows these operations in the order they are performed. As you can see, executing the vertex and pixel shader program is just another part of the rendering pipeline.

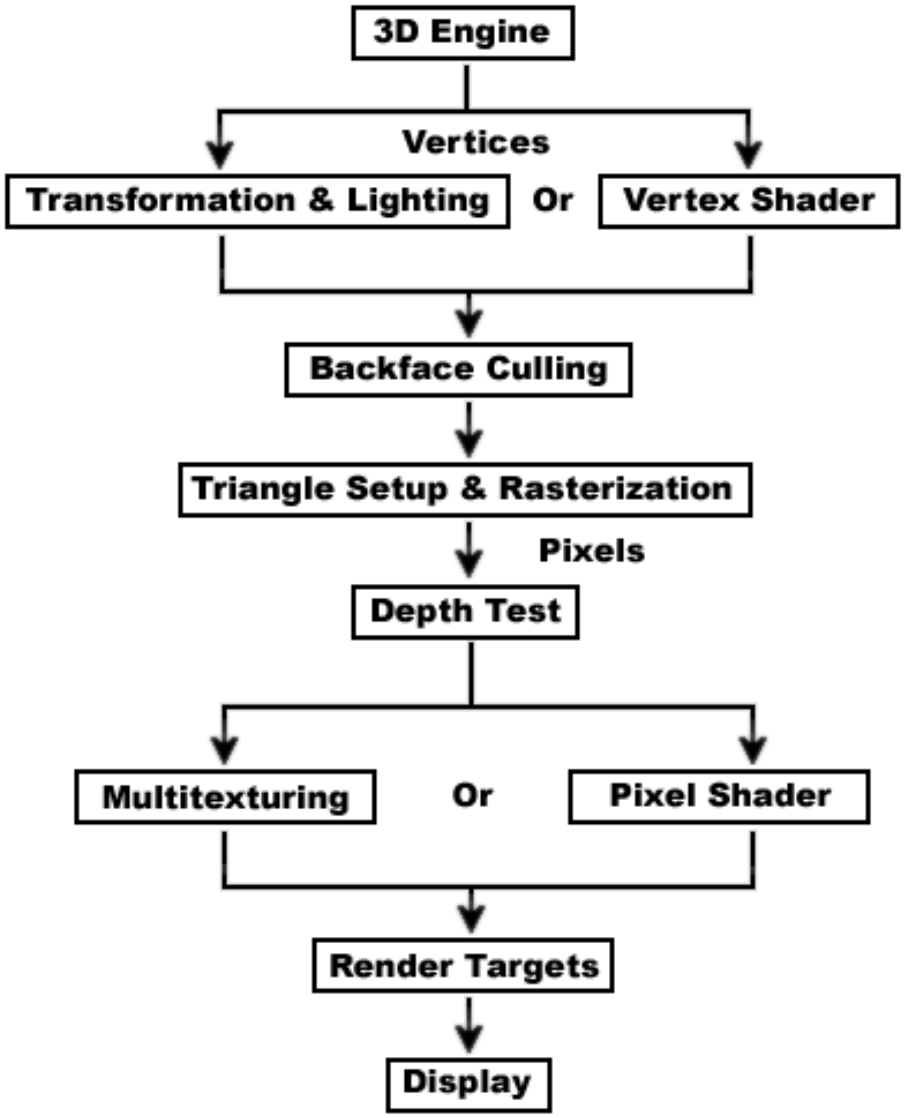


Figure 1.1.1. Simplified version of the rendering pipeline in D3D.

We will now discuss each step of the rendering pipeline separately.

Vertices

The application sends vertex data to the fixed function pipeline or the vertex shader. Typically, the vertex data consist of the vertex position, normal and UV coordinates.

The vertex position is a vector in object space, meaning that the x, y and z element of the vector are relative to the objects' centre and orientation (In contrast to world-space where the position is relative to the origin of the world and its axes).

The vertex normal is a vector describing the direction of the vertex. It is often used for calculating the lighting of the vertex.

The texture coordinates (also known as UV coordinates) give the position of the vertex on the UV map. It is used in the pixel shader to lookup the pixel color from the texture.

Other data can also be passed to the vertex shader like vertex colors, tangents, tessellation factors and more. They will be explained when needed.

Transformation & Lighting

The next step is transformation & lighting (T&L). This is an important step that is performed by the fixed function pipeline or, if you need more freedom, programmed into the vertex shader.

The input vertex position is in object space, so the position is represented by a vector whose elements are on the objects' local axes. Because the position of the vertex is relative to the objects' origin, it doesn't say anything about the position of the vertex on the screen. We must perform a series of transformations in order to get the screen coordinates of the vertex.

First the vertex position is transformed to world space. In world space, the vertex position is relative to the world origin. It is then transformed to camera space. In camera space, the x coordinate goes from left to right on the screen, y goes up and z goes into the screen. We are getting close now, but we don't have screen coordinates just yet. The view is still *orthogonal*: objects in the distance have the same size as objects that are close, there is no perspective.

The area of the world that is visible to the camera has the shape of a pyramid with the top cut off; this is called the viewing frustum. To create the illusion of perspective, the viewing frustum is 'squeezed' into a cube. This last transformation is called the perspective transformation and the resulting coordinates are in the so called clip space.

Because the far away vertices are moved more in order to fit in the cube, the far away objects will appear smaller in clip space. This may be somewhat hard to grasp, but trying to imagine the viewing frustum in your head with vertices in it may help.

If you don't fully understand what is meant by transformations just yet, don't worry. If you use the fixed function pipeline you don't have to do anything for it. If you write a vertex shader, all these transformations can be done in a single line of code which can easily be memorized.

Next is lighting. There are a number of lighting algorithms fixed into the graphics card. These algorithms assign a lightness value to every vertex dependant on the positions of the lights in the scene. The lightness value of a pixel is then calculated by linear interpolation of the lightness values of the vertices forming the triangle.

Vertex Shader

When writing advanced graphics effects you will often find that the T&L provided by the fixed function pipeline is far too limiting. If this is the case, you can write a vertex shader program to replace the T&L stage for a certain object in your scene.

In the vertex shader, you are free to do whatever you like. You can move the vertex position or texture coordinates, perform calculations and pass the results to the following stages of the rendering pipeline.

Backface Culling

Backface culling is the process of removing all triangles that face away from the viewer. The backside of triangles can't be seen anyway so this step helps reduce rendering time.
On average, half of the triangles are facing away from the viewer so this step has a significant effect on performance.

Triangle Setup & Rasterization

At triangle setup the life of vertices ends and the life of pixels begins. Rasterization is the process of determining which screen pixels belong to a given triangle.

Depth Test

The depth test is used to determine the visibility of a pixel. This is done by comparing the depth of the pixel to the stored depth value at that pixels' position. If the new pixel is closer to the camera than the stored one, it is drawn and the depth value is updated. If the new pixel is behind the old pixel, it is not drawn and the old pixels' depth value remains stored.

Multitexturing

If no pixel shader is used, the default multitexturing stage from the fixed function pipeline is used. This simply draws the pixel with the color of the texture(s) at the given texture coordinates and lights it.

Pixel Shader

For anything more than simple diffuse lighting you will need a pixel shader. The pixel shader is a function that takes a number of parameters like texture coordinates and light values and returns a red, green, blue, alpha (RGBA) color vector. The pixel shader function may contain any kind of logic. The only requirement is that it returns a color vector.

Render Targets

Finally, the output is written into the render target. The render target is just a square grid of pixels. Usually, the render target is sent to the monitor so it can be displayed. However, it is also possible to reuse the rendered image, for example for doing a post-processing effect or showing it on a surface in the level, like a mirror or water surface.

Section II

In this section, we will build a simple lighting algorithm from the ground up. We will start with the most basic lighting, ambient lighting. Then we will add a diffuse lighting term and a specular lighting term. Finally we will add normal mapping for creating the illusion of surface detail.

For every term in the shading algorithm, we will first take a look at the theory behind it, followed by the code and finally a step by step explanation of the code.

I am aware that these shaders are nothing new and that they are very inefficient. I have kept things as simple as possible to make it easier to understand. The purpose of these shaders is purely educational and they should not be used in a project (it's not forbidden, though).

This tutorial does not provide an overview of the whole HLSL syntax, for that I refer you to the Microsoft Developer Network (MSDN, see the "Further Reading" section of this tutorial).

2.1 Ambient lighting

Theory

In the real world all the light has a source and direction. In computer graphics, however, we cannot calculate the path of every light beam because that would be way too computationally intensive for current hardware to keep an interactive frame rate.

Most of the things we see around us are not lit by a light source directly, instead, the light is scattered and reflected several times before entering our eye. We can use this to our advantage by discarding the source and direction information and giving all the objects in a scene a base luminance based solely on an intensity and color value. In computer graphics, we call this kind of lighting ambient.

The ambient lighting component is described by the following formula:
Ambient Light = Ambient Intensity * Ambient Color

Where "Ambient Light" is a vector containing the amount of red, green and blue ambient light on the pixel, "Ambient Color" is a vector containing the color of the ambient light and "Ambient Intensity" is a value for the intensity of the ambient light.

Implementation

```
/*************************************************************************
/       Global Variables:
/*************************************************************************/
// Tweakables:
static const float AmbientIntensity   = 1.0f;         // The intensity of the ambient light.

// Application fed data:
const float4x4 matWorldViewProj;                      // World*view*projection matrix.
const float4 vecAmbient;                              // Ambient color, passed by the engine.

/*************************************************************************
/       Vertex Shader:
/*************************************************************************/
void AmbientVS(     in float4 InPos         : POSITION,
                    out float4 OutPos       : POSITION)
{
        // Transform the vertex from object space to clip space:
        OutPos = mul(InPos, matWorldViewProj);
}

/*************************************************************************
/       Pixel Shader:
/*************************************************************************/
float4 AmbientPS() : COLOR
{
        return AmbientIntensity * vecAmbient;
}

/*************************************************************************
/       Technique:
/*************************************************************************/
technique AmbientTechnique
{
        pass P0
        {
                VertexShader = compile vs_1_1 AmbientVS();
                PixelShader  = compile ps_1_1 AmbientPS();
        }
}
```

That may look scary but don't run away just yet. We will now discuss every line in detail.

Let's first take a look at the variable definitions at the beginning of the code.

```
// Tweakables:
static const float AmbientIntensity   = 1.0f;         // The intensity of the ambient light.

// Application fed data:
const float4x4 matWorldViewProj;                      // World*view*projection matrix.
const float4 vecAmbient;                              // Ambient color, passed by the engine.
```

We first define a variable called "AmbientIntensity" and assign it a value of 1.0. This variable is a *floating point* (float) variable, meaning it can contain decimal values like 3.1415. For assigning a number to a floating point variable, an 'f' must be added at the end of it: "1.0f". The "static" keyword tells the compiler that the variable may not be changed by the application (the 3D engine). The "const" keyword tells the compiler that the variable cannot be changed from within the shader. A "static const" variable can never change (except by the programmer before runtime).

Next is the world-view-projection matrix which is used to transform the vertex to clip space (see "Transformation & Lighting" in section I of this tutorial and "Coordinate Spaces" in Appendix A). Because it is a 4x4 matrix, we use the "float4x4" data type. We use "const" to tell the compiler that the variable may not be changed from within the shader. We don't use "static", nor do we assign any initial values because the matrix is set once per frame by the engine for every mesh that uses this shader.

The matrix MUST be named "matWorldViewProj" because otherwise the engine doesn't know it has to be set. The same goes for the "vecAmbient". These variables are called effect variables in the 3DGS manual. They are assigned a value by the engine before the shader is executed. You can find a list of all such effect variables in the 3DGS manual: C-Script > Predefined Objects > Materials and Shaders > Effect Variables.

Lastly, we define a color vector vecAmbient. We use a "float4" data type so we can store the 4 values that make up the color (red, green, blue, alpha). Like the world-view-projection matrix, this variable can only be changed by the engine, not in the shader itself. More precisely, the engine will pass the ambient color defined in the material in C-Script.

On to the vertex shader:

```
void AmbientVS(     in float4 InPos      : POSITION,
                    out float4 OutPos    : POSITION)
{
        // Transform the vertex from object space to clip space:
        OutPos = mul(InPos, matWorldViewProj);
}
```

The vertex shader is just a function. The function returns a "void", meaning that the data that is returned has no specified type. In this case we could put float4 there because this function only returns a float4 but in many cases you will want to return more than one value. For c/c++ programmers: you can also use a structure to define the output data in which case you don't have to specify any output variables with the 'out' keyword.

The function is called "AmbientVS" and has 1 input (in) variable called InPos and 1 output variable (out), called OutPos. Both variables are marked with the POSITION *semantic*. A semantic is a keyword that tells the compiler what kind of data is stored in this variable. This is important, because the subsequent graphics pipeline stages must know what to do with the data.

The content of the function is only one command which transforms the vertex position to clip space. This is done by multiplying the input position vector by the world-view-projection matrix. For multiplying matrices and vectors you must use the "mul" intrinsic function.

Next is the pixel shader:

```
float4 AmbientPS() : COLOR
{
	return AmbientIntensity * vecAmbient;
}
```

Once again, we define a function. We call it AmbientPS. This function returns a "float4" which is marked by a COLOR semantic. No input variables are needed for this function.

We return a color vector which is the product of the ambient intensity and the ambient color vector. It is important to note that we are multiplying a vector with a single value. This is possible because HLSL natively supports vectors. We can perform operations on vectors in a single instruction that would, in normal C/C++ require multiple instructions:

```
// HLSL:
VectorB = VectorA * 3;

VectorC = VectorD * VectorE;

// C/C++:
VectorB.x = VectorA.x * 3;
VectorB.y = VectorA.y * 3;
VectorB.z = VectorA.z * 3;

VectorC.x = VectorD.x * VectorE.x;
VectorC.y = VectorD.y * VectorE.y;
VectorC.z = VectorD.z * VectorE.z;
```

So, since we are writing HLSL, each component (R,G,B and A) gets multiplied with the value of AmbientIntensity (see 'Scaling a Vector' in Appendix A) before it is returned.

Finally we wrap it up in a "technique":

```
technique AmbientTechnique
{
	pass P0
	{
		VertexShader = compile vs_1_1 AmbientVS();
		PixelShader  = compile ps_1_1 AmbientPS();
	}
}
```

The technique is the "heart" of the shader, here we tie everything together.

A technique consists of one or more *passes*. In a pass, the model gets rendered once with a certain vertex and pixel shader. You may also set any parameters of the FFP but we don't need that here. Sometimes, you will need multiple passes, for example when rendering the outline and insides of a mesh for a toon shader seperately. The second pass may render the model in a different way and the results of both passes can be mixed. However, that is beyond the scope of this tutorial.

We define the technique and call it AmbientTechnique. In the pass, which we call "P0", we specify a vertex and pixel shader function which are both compiled to the 1.1 *rendertarget*. Choosing the rendertarget 1.1 allows this shader to run even on old, first-generation shader

hardware but it also limits the possibilities of the programmer. Because this is a very simple shader we don’t need any special instructions and VS/PS 1.1 will do.

Results

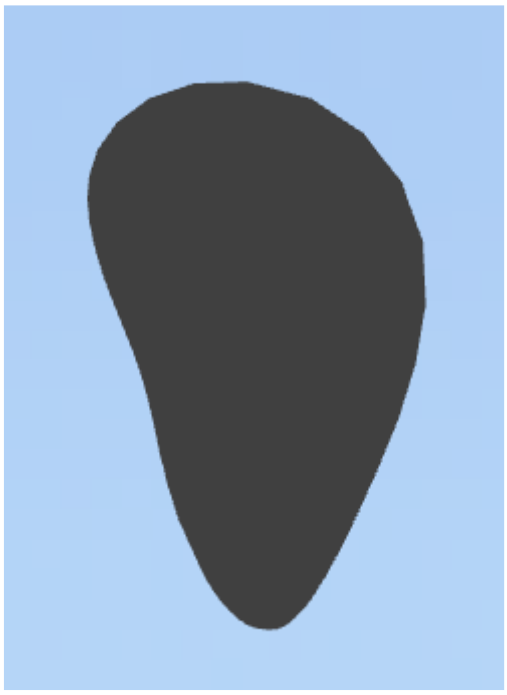


Figure 2.1.1. Ambient lighting.

As you can see in figure 2.1.1, the ambient lighting algorithm paints the whole object with exactly the same color. Because of this, no contour information is visible (except for the outline of the object).

2.2 Diffuse lighting

Theory

To overcome some of the limitations of the ambient lighting model we will now add a diffuse lighting term to the shader. The diffuse lighting model will take the light direction and the surface orientation into account to calculate the amount of light that is reflected from the surface into all directions. Because the light is reflected equally into all directions the diffuse lighting is not dependent on the view position. This makes it suitable for simulating matte surfaces like paper.

From now on, **L** is the light direction vector and **N** is the surface normal:

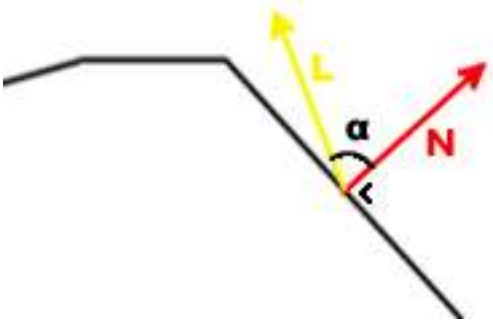


Figure 2.2.1. The light and normal vectors.

The amount of light that is reflected is directly proportional to the angle between the light direction and the surface normal. When **N** and **L** are aligned (in other words: when the light beam is perpendicular to the surface) the reflection is at its peak, when **N** and **L** are perpendicular, no diffuse light is reflected.

If we take the cosine of the angle between **N** and **L** we get 1 when they are aligned and 0 when they are perpendicular. We don't know the angle between **N** and **L** but we can use the following property of the dot product to calculate it:
$\mathbf{N} \cdot \mathbf{L} = ||\mathbf{N}|| * ||\mathbf{L}|| * \cos(\alpha)$

If we use normalized vectors we can simplify this to:
$\mathbf{N} \cdot \mathbf{L} = 1 * 1 * \cos(\alpha) = \cos(\alpha)$

The dot product equals cos(α) if **N** and **L** are unit vectors. The diffuse equation is:
Diffuse Light = Diffuse Intensity * $\mathbf{N} \cdot \mathbf{L}$

Our entire equation is now:
Final Color = (Diffuse Light + Ambient Light) * Diffuse Color

The diffuse color will be read from the texture map.

Implementation

```
/*************************************************************************
/       Global Variables:
/*************************************************************************/
// Tweakables:
static const float AmbientIntensity   = 1.0f;          // The intensity of the ambient light.
static const float DiffuseIntensity = 1.0f;            // The intensity of the diffuse light.
static const float4 SunColor = {0.9f, 0.9f, 0.5f, 1.0f};// Color vector of the sunlight.

// Application fed data:
const float4x4 matWorldViewProj;          // World*view*projection matrix.
const float4x4 matWorld;                  // World matrix.
const float4 vecAmbient;                  // Ambient color, passed by the engine.
const float4 vecSunDir;                   // The sun direction vector.

texture entSkin1;                                 // Color map.
sampler ColorMapSampler = sampler_state           // Color map sampler.
{
        Texture = <entSkin1>;
        AddressU  = Clamp;
        AddressV  = Clamp;
};

/*************************************************************************
/       Vertex Shader:
/*************************************************************************/
void DiffuseVS(       in float4 InPos         : POSITION,
                      in float3 InNormal      : NORMAL,
                      in float2 InTex         : TEXCOORD0,

                      out float4 OutPos       : POSITION,
                      out float2 OutTex       : TEXCOORD0,
                      out float3 OutNormal: TEXCOORD1)
{
        // Transform the vertex from object space to clip space:
        OutPos = mul(InPos, matWorldViewProj);

        // Transform the normal from object space to world space:
        OutNormal = normalize(mul(InNormal, matWorld));;

        // Pass the texture coordinate to the pixel shader:
        OutTex = InTex;
}


/*************************************************************************
/       Pixel Shader:
/*************************************************************************/
float4 DiffusePS(     in float2 InTex         : TEXCOORD0,
                      in float3 InNormal      : TEXCOORD1) : COLOR
```

```
{
	// Calculate the ambient term:
	float4 Ambient = AmbientIntensity * vecAmbient;

	// Calculate the diffuse term:
	float4 Diffuse = DiffuseIntensity * saturate(dot(vecSunDir, normalize(InNormal)));
	Diffuse *= SunColor;

	// Fetch the pixel color from the color map:
	float4 Color = tex2D(ColorMapSampler, InTex);

	// Calculate final color:
	return (Ambient + Diffuse) * Color;
}

/**************************************************************************
/	Technique:
/**************************************************************************/
technique DiffuseTechnique
{
	pass P0
	{
		VertexShader = compile vs_2_0 DiffuseVS();
		PixelShader  = compile ps_2_0 DiffusePS();
	}
}
```

We will first take a look at the variable definitions because some have been added:

```
// Tweakables:
static const float AmbientIntensity   = 1.0f;          // The intensity of the ambient light.
static const float DiffuseIntensity = 1.0f;            // The intensity of the diffuse light.
static const float4 SunColor = {0.2f, 0.2f, 0.2f, 1.0f};// Color vector of the sunlight.

// Application fed data:
const float4x4 matWorldViewProj;          // World*view*projection matrix.
const float4 vecAmbient;                  // Ambient color, passed by the engine.
const float4 vecSunDir;                   // The sun direction vector.

texture entSkin1;                                      // Color map.
sampler ColorMapSampler = sampler_state                // Color map sampler.
{
	Texture = <entSkin1>;
	AddressU  = Clamp;
	AddressV  = Clamp;
};
```

As you can see, two new variables have been added under tweakables. One is the diffuse intensity and the other is the color of the sunlight. Feel free to experiment with different intensity and color values.

Next we define a second matrix for transforming the vertex normal from object space to world space. We have also added a new vector that will be set to the suns' direction by the engine.

Next we define a texture. The texture is also sent from the engine and thus must be named according to the list of effect variables from the manual: 'entSkin1' for the first model skin. This shader will expect that the color texture is stored in entSkin1 (you would have to set this in MED).

Next we define a sampler. A sampler is an object that defines how a texture should be read. All we specify for this sampler, called ColorMapSampler is that if the texture coordinates are outside the texture size, the last pixel of the texture is returned. You can also choose to 'Wrap' the coordinates which is used for repeating textures. For a full list of sampler_state settings, check MSDN (see Further Reading section of this tutorial).

The vertex shader for our diffuse lighting tutorial is only slightly more complex than that of the ambient lighting shader:

```
void DiffuseVS(     in float4 InPos      : POSITION,
                    in float3 InNormal   : NORMAL,
                    in float2 InTex      : TEXCOORD0,

                    out float4 OutPos    : POSITION,
                    out float2 OutTex    : TEXCOORD0,
                    out float3 OutNormal: TEXCOORD1)
{
    // Transform the vertex from object space to clip space:
    OutPos = mul(InPos, matWorldViewProj);

    // Transform the normal from object space to world space:
    OutNormal = normalize(mul(InNormal, matWorld));;

    // Pass the texture coordinate to the pixel shader:
    OutTex = InTex;
}
```

We define the function DiffuseVS which takes three input vectors and also returns three vectors. The input vectors are automatically set to the POSITION, NORMAL and first texture coordinate (TEXCOORD1) of the vertex by DirectX (which in turn receives it from the engine, see chapter: "D3D Pipeline") because they are marked with the corresponding input semantic.

In the shader, we once again transform the vertex. After that, the normal is transformed to world space. Since vecSunDir is in world space, the vertex normal must also be in world space in order to compare it to the sun direction vector in the pixel shader. We then proceed to pass the input texture coordinates to the output variables. We don't need to change the texture coordinate for this shader.

Almost the whole theory we have discussed is executed in the pixelshader:

```
float4 DiffusePS(   in float2 InTex      : TEXCOORD0,
                    in float3 InNormal   : TEXCOORD1) : COLOR
{
```

```
        // Calculate the ambient term:
        float4 Ambient = AmbientIntensity * vecAmbient;

        // Calculate the diffuse term:
        float4 Diffuse = DiffuseIntensity * saturate(dot(vecSunDir, normalize(InNormal)));
        Diffuse *= SunColor;

        // Fetch the pixel color from the color map:
        float4 Color = tex2D(ColorMapSampler, InTex);

        // Calculate final color:
        return (Ambient + Diffuse) * Color;
}
```

First we define the function DiffusePS which takes two vectors as input and returns a color vector. Remember we marked the output variables of the vertex shader with TEXCOORD0 and TEXCOORD1? The data that we put in it in the vertex shader (a texture coordinate and a normal) has been interpolated between the three vertices that form the triangle on which this pixel is situated. The interpolated result is put back in the TEXCOORD0 and TEXCOORD1 registers. So now that we have the texture coordinate that is somewhere in between the texture coordinates of the three vertices forming the triangle this pixel is on, we can use it to do a lookup from the texture.

First we calculate the ambient component just like in the last chapter. Only this time, we store it in a new vector called "Ambient" that we can use later on to calculate the final color.

We then continue by calculating the diffuse component. Remember the formula?
Here it is again:
Diffuse Light = Diffuse Intensity * $\mathbf{N} \cdot \mathbf{L}$

Let's take the next line apart.

```
float4 Diffuse = DiffuseIntensity * saturate(dot(vecSunDir, normalize(InNormal)));
Diffuse *= SunColor;
```

First we normalize the vector InNormal. This is necessary because it may have been "denormalized" by the interpolation. If you use a non normalized vector to calculate the dot product the result will be different because the following rule would **not** apply anymore:
$\mathbf{N} \cdot \mathbf{L} = ||\mathbf{N}|| * ||\mathbf{L}|| * \cos(\alpha) = 1 * 1 * \cos(\alpha) = \cos(\alpha)$

So we take the dot product of the sun direction (which was passed by the engine) and the normalized vertex normal. Because both are unit vectors, the result of the dot product is the cosine of the angle between those vectors.

We then use the saturate intrinsic function to clamp the value of the dot product between 0 and 1. We don't want to assign 'negative light' to surfaces that are facing away from the light!

Next we multiply the calculated color by the color of the sunlight. What happens here is that the R, G and B components of the original color get increased or decreased individually, making the resulting color more red, green or blue depending on the SunColor RGB values.

We then read the color from the color map. We use the tex2D intrinsic function to read a color vector from the ColorMapSampler at the coordinates given by InTex.

Finally, we return the total amount of light (Ambient + Diffuse), multiplied with the color from the texture map.

Results

Figure 2.2.2. Diffuse lighting.

We can now see the contours of the object and we can tell where the light is coming from. We can also see the texture that we have assigned to the model. However, the lighting is the independent of the viewers' position. This is ok for matte surfaces but shiny surfaces do not reflect light equally into every direction. In the next paragraph we will enhance the lighting algorithm to make it suitable for shiny surfaces.

2.3 Specular lighting

Theory

For simulating shiny/polished surfaces the algorithm must also take into account the location of the viewer.

There are several lighting models for simulating specular reflectance. We will use the Phong specular lighting model developed by Bui Tong Phong.

We will use a reflection vector that contains the direction of the light after it has bounced off the surface. This vector is calculated from the light direction vector and the normal of the surface, see figure 2.3.1.

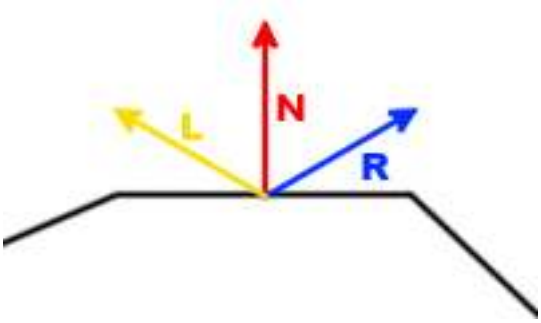


Figure 2.3.1. The reflection vector.

This reflection vector is compared to the view direction vector. If the angles are similar, the surface will be lit brightly, if they are far off, no specular light is reflected from the surface into the camera and only the diffuse and ambient light will be visible on this pixel.

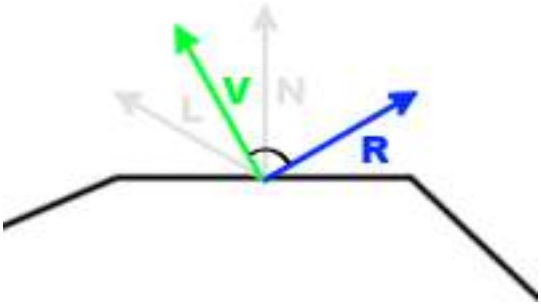


Figure 2.3.2. The angle between the reflection vector and view vector.

The reflection vector is calculated with the following formula:
$\mathbf{R} = 2 * (\mathbf{N} \cdot \mathbf{L}) * \mathbf{N} - \mathbf{L}$

The view vector is calculated by subtracting the vector world position from the camera position:
V = Camera Position - Vertex Position

This results in a vector pointing from the vertex to the camera (see “Subtracting Vectors” in Appendix A)

As with diffuse lighting, we will use the dot product to calculate the cosine of the angle difference between the two vectors. This time we need the difference between the view vector and the reflection vector. We add an exponent which makes the highlight harder or softer.
Specular Light = $(\mathbf{R} \cdot \mathbf{V})^n$

Our entire equation is now:
Final Color = (Diffuse Light + Ambient Light + Specular Light) * Diffuse Color

Implementation

```
/**************************************************************************
/       Global Variables:
/**************************************************************************/
// Tweakables:
static const float AmbientIntensity   = 1.0f;          // The intensity of the ambient light.
static const float DiffuseIntensity = 1.0f;            // The intensity of the diffuse light.
static const float SpecularIntensity = 2.0f;           // The intensity of the specular light.
static const float SpecularPower = 8.0f;               // The specular power. Used as
'glossiness' factor.
static const float4 SunColor = {0.9f, 0.9f, 0.5f, 1.0f};       // Color vector of the sunlight.

// Application fed data:
const float4x4 matWorldViewProj;           // World*view*projection matrix.
const float4x4 matWorld;                   // World matrix.
const float4 vecAmbient;                   // Ambient color.
const float4 vecSunDir;                    // The sun direction vector.
const float4 vecViewPos;                   // View position.

texture entSkin1;                          // Color map.
sampler ColorMapSampler = sampler_state // Color map sampler.
{
        Texture = <entSkin1>;
        AddressU  = Clamp;
        AddressV  = Clamp;
};

/**************************************************************************
/       Vertex Shader:
/**************************************************************************/
void SpecularVS(      in float4 InPos         : POSITION,
                      in float3 InNormal      : NORMAL,
                      in float2 InTex         : TEXCOORD0,

                      out float4 OutPos       : POSITION,
                      out float2 OutTex       : TEXCOORD0,
                      out float3 OutNormal : TEXCOORD1,
                      out float3 OutViewDir: TEXCOORD2)
{
        // Transform the vertex from object space to clip space:
        OutPos = mul(InPos, matWorldViewProj);

        // Transform the normal from object space to world space:
        OutNormal = normalize(mul(InNormal, matWorld));;

        // Pass the texture coordinate to the pixel shader:
        OutTex = InTex;

        // Calculate a vector from the vertex to the view:
        OutViewDir = vecViewPos - mul(InPos, matWorld);
```

```
}


/*************************************************************************
/       Pixel Shader:
/*************************************************************************/
float4 SpecularPS(    in float2 InTex        : TEXCOORD0,
                      in float3 InNormal     : TEXCOORD1,
                      in float4 InViewDir    : TEXCOORD2) : COLOR
{
        // Calculate the ambient term:
        float4 Ambient = AmbientIntensity * vecAmbient;

        // Calculate the diffuse term:
        InNormal = normalize(InNormal);
        float4 Diffuse = DiffuseIntensity * saturate(dot(vecSunDir, InNormal));
        Diffuse *= SunColor;

        // Fetch the pixel color from the color map:
        float4 Color = tex2D(ColorMapSampler, InTex);

        // Calculate the reflection vector:
        float3 R = normalize(2 * dot(InNormal, vecSunDir) * InNormal - vecSunDir);

        // Calculate the speculate component:
        float Specular = pow(saturate(dot(R, normalize(InViewDir))), SpecularPower) *
SpecularIntensity;

        // Calculate final color:
        return (Ambient + Diffuse + Specular) * Color;
}

/*************************************************************************
/       Technique:
/*************************************************************************/
technique SpecularTechnique
{
        pass P0
        {
                VertexShader = compile vs_2_0 SpecularVS();
                PixelShader  = compile ps_2_0 SpecularPS();
        }
}
```

We have added two new variables: SpecularIntensity and SpecularPower. These will be used in the pixel shader as an intensity and “glossiness” factor.

We have also added a new constant that is passed by the engine. vecViewPos is the position of the camera in the world.

The vertex shader:

```
void SpecularVS(    in float4 InPos        : POSITION,
                    in float3 InNormal     : NORMAL,
                    in float2 InTex        : TEXCOORD0,

                    out float4 OutPos      : POSITION,
                    out float2 OutTex      : TEXCOORD0,
                    out float3 OutNormal : TEXCOORD1,
                    out float3 OutViewDir: TEXCOORD2)
{
        // Transform the vertex from object space to clip space:
        OutPos = mul(InPos, matWorldViewProj);

        // Transform the normal from object space to world space:
        OutNormal = normalize(mul(InNormal, matWorld));;

        // Pass the texture coordinate to the pixel shader:
        OutTex = InTex;

        // Calculate a vector from the vertex to the view:
        OutViewDir = vecViewPos - mul(InPos, matWorld);
}
```

We have added one output parameter which will contain a vector pointing from the vertex to the view; we call it “OutViewDir”.

Like before, we transform the vertex and normal and pass the texture coordinate.

The specular highlights are dependent on the view direction so we have to calculate a vector that gives the view direction. We know the world position of the view (vecViewPos) and we know the objectspace position of the vertex. To calculate the view direction vector we need the worldspace position of the vertex so we transform the vertex position by multiplying it with the world matrix. Subtracting the vertex world position vector from the view position vector results in a direction vector pointing to the view from the vertex (see “Subtracting Vectors” in Appendix A).

The pixel shader:

```
float4 SpecularPS(  in float2 InTex        : TEXCOORD0,
                    in float3 InNormal     : TEXCOORD1,
                    in float4 InViewDir    : TEXCOORD2) : COLOR
{
        // Calculate the ambient term:
        float4 Ambient = AmbientIntensity * vecAmbient;

        // Calculate the diffuse term:
        InNormal = normalize(InNormal);
        float4 Diffuse = DiffuseIntensity * saturate(dot(vecSunDir, InNormal));
        Diffuse *= SunColor;

        // Calculate the reflection vector:
        float3 R = normalize(2 * dot(InNormal, vecSunDir) * InNormal - vecSunDir);
```

```
        // Calculate the speculate component:
        InViewDir = normalize(InViewDir);
        float Specular = pow(saturate(dot(R, InViewDir)), SpecularPower) *
SpecularIntensity;

        // Fetch the pixel color from the color map:
        float4 Color = tex2D(ColorMapSampler, InTex);

        // Calculate final color:
        return (Ambient + Diffuse + Specular) * Color;
}
```

We receive the view direction vector in TEXCOORD2 where we put it in the pixel shader.

The ambient and diffuse components are calculated like before.

We calculate the reflection vector **R** with the formula discussed before. The specular component is calculated by taking the dot product of the reflection vector and the view direction, raised to the power given by the SpecularPower variable making the highlight harder or softer. The result is multiplied by SpecularIntensity to make the highlight brighter or darker.

We once again add up all the light, now including specular and multiply it by the color which is read from the color map.

Results

Figure 2.3.3. Specular highlights.

By adding specular highlights, we see a view dependent highlight making the material look shiny and polished.

2.4 Normal mapping

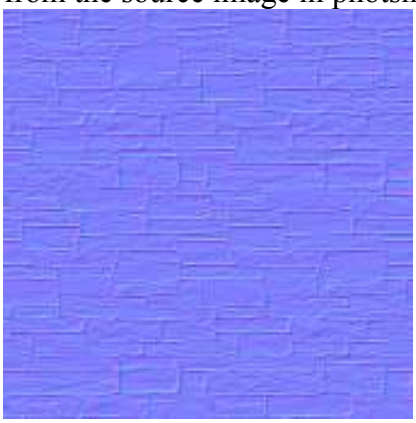
Theory

As a bonus we will add normal mapping to our lighting algorithm. Normal mapping, also known as bump mapping, creates the illusion of small surface details that are not present in the actual geometry. This is done by storing a normal vector in a secondary texture map. This normal is used in the lighting algorithms instead of the vertex normal.

Because of the way normal mapping works, only the lightness of pixels is affected because the lighting is dependant on the normal. The actual geometry is not altered. This can be noticeable when looking at the outline of a mesh or when viewing a normal mapped surface from an odd angle.

We will store the perturbed normals in a normal map. This is just a RGB image storing the x, y and z component of the normal in the red, green and blue channel of the image. Normal maps can be generated from a high-res mesh in high-end 3D modelling packages like 3D Studio Max or drawn in Photoshop with the normal map plug-in.

Here is a scaled down version of the normal map we will be using, generated in a few clicks from the source image in photshop:



You can see the full size normal map in the model “Blob.mdl” in the normal mapping demo.

Because the RGB values in images can range from 0 to 1 (often represented as 0 to 255 in image manipulation programs), the normals are “compressed” from the [-1..1] range into the [0..1] range when the normalmap is generated. In the shader we must extract the original values by multiplying it by 2 and then subtracting 1.

The normals in a normalmap are stored in the so called “tangent space” (also known as “texture space”). In the tangent space coordinate system, the W axis is the normal of the surface, the U axis is the tangent that is passed from the engine and V is perpendicular to W and U.

Because the normals are stored in tangent space, we need to convert the light and view direction vectors to tangent space as well to be able to compare these vectors. We calculate a matrix to transform the vectors from world space to tangent space and use it in the vertex shader to convert the view and light vectors to tangent space. These vectors are then passed to the pixel shader where the perturbed normal is read from the normalmap. The perturbed normal, the light vector and the view vector are then used in the lighting algorithms we have discussed in the previous chapters.

Implementation

```
/*********************************************************************
/       Global Variables:
/*********************************************************************/
// Tweakables:
static const float AmbientIntensity   = 1.0f; // The intensity of the ambient light.
static const float DiffuseIntensity = 1.0f;     // The intensity of the diffuse light.
static const float SpecularIntensity = 1.0f;   // The intensity of the specular light.
static const float SpecularPower = 8.0f;       // The specular power. Used as 'glossyness' factor.
static const float4 SunColor = {0.9f, 0.9f, 0.5f, 1.0f};        // Color vector of the sunlight.

// Application fed data:
const float4x4 matWorldViewProj;            // World*view*projection matrix.
const float4x4 matWorld;                    // World matrix.
const float4 vecAmbient;                    // Ambient color.
const float4 vecSunDir;                     // The sun direction vector.
const float4 vecViewPos;                    // View position.

texture entSkin1;                                   // Color map.
sampler ColorMapSampler = sampler_state         // Color map sampler.
{
        Texture = <entSkin1>;
        AddressU  = Clamp;
        AddressV  = Clamp;
};

texture entSkin2;                                   // Normal map.
sampler NormalMapSampler = sampler_state        // Normal map sampler.
{
        Texture = <entSkin2>;
        AddressU  = Clamp;
        AddressV  = Clamp;
};


/*********************************************************************
/       Vertex Shader:
/*********************************************************************/
void NormalMapVS( in float4 InPos          : POSITION,
                  in float3 InNormal       : NORMAL,
                  in float2 InTex                  : TEXCOORD0,
                  in float3 InTangent      : TEXCOORD2,

                  out float4 OutPos        : POSITION,
                  out float2 OutTex        : TEXCOORD0,
                  out float3 OutViewDir: TEXCOORD1,
                  out float3 OutSunDir: TEXCOORD2)
{
        // Transform the vertex from object space to clip space:
        OutPos = mul(InPos, matWorldViewProj);
```

```
	// Pass the texture coordinate to the pixel shader:
	OutTex = InTex;

	// Compute 3x3 matrix to transform from world space to tangent space:
	half3x3 worldToTangentSpace;
	worldToTangentSpace[0] = mul(InTangent, matWorld);
	worldToTangentSpace[1] = mul(cross(InTangent, InNormal), matWorld);
	worldToTangentSpace[2] = mul(InNormal, matWorld);

	// Calculate the view direction vector in tangent space:
	OutViewDir = normalize(mul(worldToTangentSpace, vecViewPos - mul(InPos,
matWorld)));

	// Calculate the light direction vector in tangent space:
	OutSunDir = normalize(mul(worldToTangentSpace, -vecSunDir));
}


/**************************************************************************
/	Pixel Shader:
/**************************************************************************/
float4 NormalMapPS(in float2 InTex	: TEXCOORD0,
		   in float3 InViewDir	: TEXCOORD1,
		   in float3 InSunDir	: TEXCOORD2) : COLOR
{
	// Read the normal from the normal map and convert from [0..1] to [-1..1] range
	float3 BumpNormal = 2 * tex2D(NormalMapSampler, InTex) - 1;

	// Calculate the ambient term:
	float4 Ambient = AmbientIntensity * vecAmbient;

	// Calculate the diffuse term:
	float4 Diffuse = DiffuseIntensity * saturate(dot(InSunDir, BumpNormal));
	Diffuse *= SunColor;

	// Calculate the reflection vector:
	float3 R = normalize(2 * dot(BumpNormal, InSunDir) * BumpNormal - InSunDir);

	// Calculate the specular term:
	InViewDir = normalize(InViewDir);
	float Specular = pow(saturate(dot(R, InViewDir)), SpecularPower) * SpecularIntensity;

	// Fetch the pixel color from the color map:
	float4 Color = tex2D(ColorMapSampler, InTex);

	// Calculate final color:
	return (Ambient + Diffuse + Specular) * Color;
}
```

```
/**************************************************************************
/       Technique:
/**************************************************************************/
technique SpecularTechnique
{
        pass P0
        {
                VertexShader = compile vs_2_0 NormalMapVS();
                PixelShader  = compile ps_2_0 NormalMapPS();
        }
}
```

To use the normal mapping shader we have to tell the engine that we need it to send the tangents to the vertex shader. We do this by setting the tangent flag in the material definition in c-script. More on that in the next section.

We have added another texture (entSkin2, the 2nd skin in MED) and sampler object for the normal map.

The vertex shader calculates the world-to-tangent-space matrix and transforms the view direction vector and light direction vector to tangent space with it:

```
void NormalMapVS( in float4 InPos          : POSITION,
                  in float3 InNormal       : NORMAL,
                  in float2 InTex          : TEXCOORD0,
                  in float3 InTangent      : TEXCOORD2,

                  out float4 OutPos        : POSITION,
                  out float2 OutTex        : TEXCOORD0,
                  out float3 OutViewDir: TEXCOORD1,
                  out float3 OutSunDir: TEXCOORD2)
{
        // Transform the vertex from object space to clip space:
        OutPos = mul(InPos, matWorldViewProj);

        // Pass the texture coordinate to the pixel shader:
        OutTex = InTex;

        // Compute 3x3 matrix to transform from world space to tangent space:
        half3x3 worldToTangentSpace;
        worldToTangentSpace[0] = mul(InTangent, matWorld);
        worldToTangentSpace[1] = mul(cross(InTangent, InNormal), matWorld);
        worldToTangentSpace[2] = mul(InNormal, matWorld);

        // Calculate the view direction vector in tangent space:
        OutViewDir = normalize(mul(worldToTangentSpace, vecViewPos - mul(InPos,
matWorld)));

        // Calculate the light direction vector in tangent space:
        OutSunDir = normalize(mul(worldToTangentSpace, -vecSunDir));
}
```

First we add another input vector called InTangent. We use the input semantic TEXCOORD2 because A6 sends the tangents through this register.
We calculate a matrix for converting the view and light direction vectors to tangent space. There are two new output vectors called OutViewDir and OutSunDir where we put the tangent space view and light direction vectors.

The pixel shader:

```
float4 NormalMapPS(in float2 InTex        : TEXCOORD0,
                   in float3 InViewDir    : TEXCOORD1,
                   in float3 InSunDir     : TEXCOORD2) : COLOR
{
        // Read the normal from the normal map and convert from [0..1] to [-1..1] range
        float3 BumpNormal = 2 * tex2D(NormalMapSampler, InTex) - 1;

        // Calculate the ambient term:
        float4 Ambient = AmbientIntensity * vecAmbient;

        // Calculate the diffuse term:
        float4 Diffuse = DiffuseIntensity * saturate(dot(InSunDir, BumpNormal));
        Diffuse *= SunColor;

        // Calculate the reflection vector:
        float3 R = normalize(2 * dot(BumpNormal, InSunDir) * BumpNormal - InSunDir);

        // Calculate the specular term:
        InViewDir = normalize(InViewDir);
        float Specular = pow(saturate(dot(R, InViewDir)), SpecularPower) *
SpecularIntensity;

        // Fetch the pixel color from the color map:
        float4 Color = tex2D(ColorMapSampler, InTex);

        // Calculate final color:
        return (Ambient + Diffuse + Specular) * Color;
}
```

The pixel shader is the same as for the specular shader but this time we receive the view and light direction vectors from the vertex shader and we use the normal read from the normalmap.

We use InViewDir on TEXCOORD1 and InSunDir on TEXCOORD2 to receive the tangent space vectors from the vertex shaders.

The normal is retrieved by reading from the normalmap at the given texture coordinates and converting it from the [0..1] range to the [-1..1] range.

The rest of the shader is identical to standard specular lighting only this time we use the perturbed normal from the normalmap and the light and view vector (in tangent space) from the vertex shader.

Results

Figure 2.3.4. Normal mapping with specular highlights.

By using normal mapping we can simulate a lot of surface details that are not present in the actual geometry, which is significantly faster (both in art creation and rendering) than modelling all that detail. Notice that you can still recognize hard edges on the outline of the object because the normal mapping does not affect the polygons.

Section III

3.1 Implementing the shaders in A6

We will now take a look at how to implement our shaders in A6. If you've come this far, implementing the shaders will be a piece of cake.

Assigning the material by script

Before we can assign the shader to an entity in our scene, we must first define a material in the script. This material contains a pointer to the effect file which in turn contains the shader code. A material definition may also contain other material properties like ambient color, specular power and more. Note that if you use a shader (the effect keyword in C-Script), the other material properties will not influence the look of the model unless you have programmed your shader to use those variables.

Create a new script called materials.wdl. Include it in your main script file like this:

```
include <materials.wdl>;
```

Create a new file named NormalMapped.fx and paste all the shader code, including variable definitions, vertex and pixel shader functions and techniques in it.

Open materials.wdl and paste the following material definition:

```
material mtlNormalMapped
{
	flags = tangent;		// Make sure the engine sends tangents to the shader (only
					// needed for tangent space normal mapping)
	effect = "NormalMapped.fx"; // This is the effect file that contains the shader code.
}
```

You can assign a material to an entity by using the entity.material parameter:

```
entity.material = mtlNormalMapped;
```

An example of an action that you can assign through WED containing only the assignment of our material:

```
action AssignMtl
{
	my.material = mtlNormalMapped;
}
```

Assigning the material through MED

Since engine version 6.40.5, you can assign materials to (a part of) a model in MED. In MED, go to Edit > Manage Skins. If there is a skin already, click [Skin Settings], if there is no skin yet, click [New Skin]. Check the "Effect Setup" and "Texture" Boxes and choose an effect file (.fx) and a texture file.

Close the skin editor and save the model. Done!

Appendix A Math

In this section I will provide an overview of the mathematical concepts behind many graphics algorithms. I will not go into too much detail because this is an entry level tutorial. If you would like get a more in-depth understanding of the mathematics, please see the further reading section.

For the mathematicians out there, please note that I will sometimes present concepts in a simplified way to make it easier to understand for newbies at the cost of completeness and correctness. For example, I will not explain that a vectors' properties are independent from the coordinate system it is represented in because we only use the Cartesian system.

What is a vector?

A vector is a concept characterized by a magnitude (length) and a direction. A vector is defined like this:
V = [e_1, e_2, e_3, …, e_n]
It is best to imagine a vector as an arrow, see figure 1. As a programmer, you can also imagine a vector as an array.

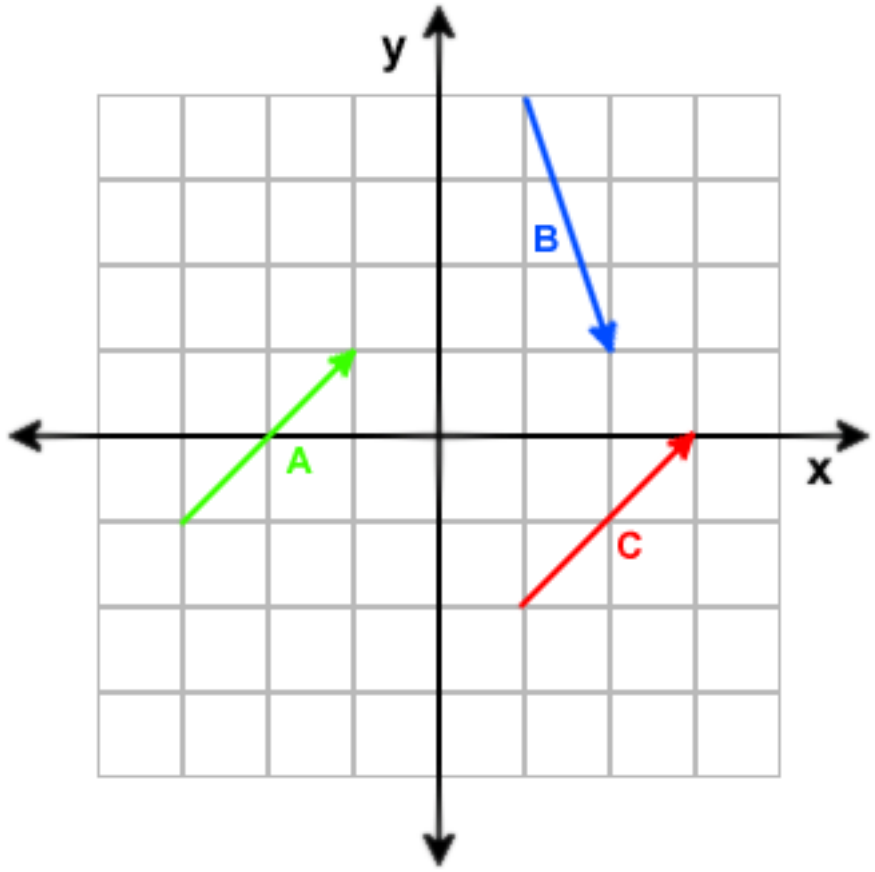


Figure 1. Graphical representation of a few 2D vectors.

A = [2, 2]
B = [-1, -3]
C = [2, 2]

Note that vector **A** and vector **C** are the same because they have the same magnitude and direction. A vector is NOT defined by its starting point; in fact it doesn't have one.

The elements of a vector are also known as its dimensions. A vector can have any number of dimensions but in graphics programming we mostly use 3D and 4D vectors. For the following examples however, I will mostly use 2D vectors because they are easier to visualize.

The vector as described above is a spatial vector. Vectors are also used to store colors (red, green, blue, [alpha]) or positions (which are basically direction vectors that start at the origin).

Vector addition

Addition of vectors is done by adding the corresponding components of each vector. Example:
A = [3, 1]
B = [0, 2]
A + **B** = [3 + 0, 1 + 2] = [3, 3]

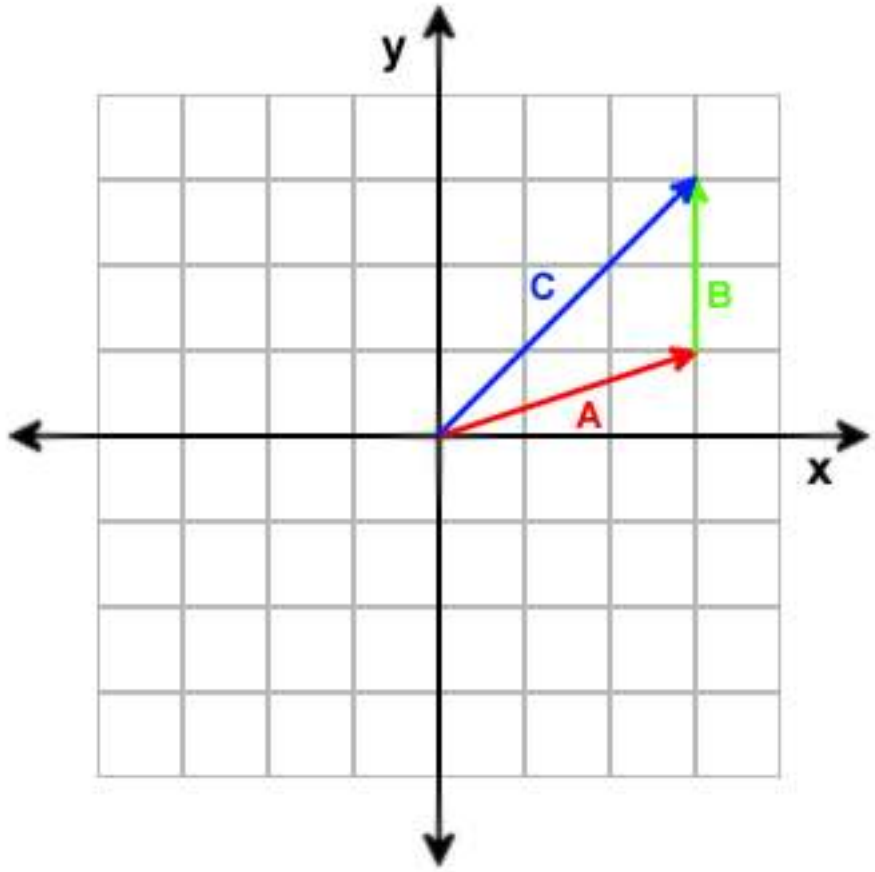


Figure 2. Addition of vectors.

Adding two vectors results in a third vector encompassing both displacements.

Vector subtraction

Subtraction of vectors is done by subtracting the corresponding components of each vector. Example:
A = [-3, 2]
B = [2, 3]
A – **B** = [(-3) - 2, 2 - 3)] = [3, -2]

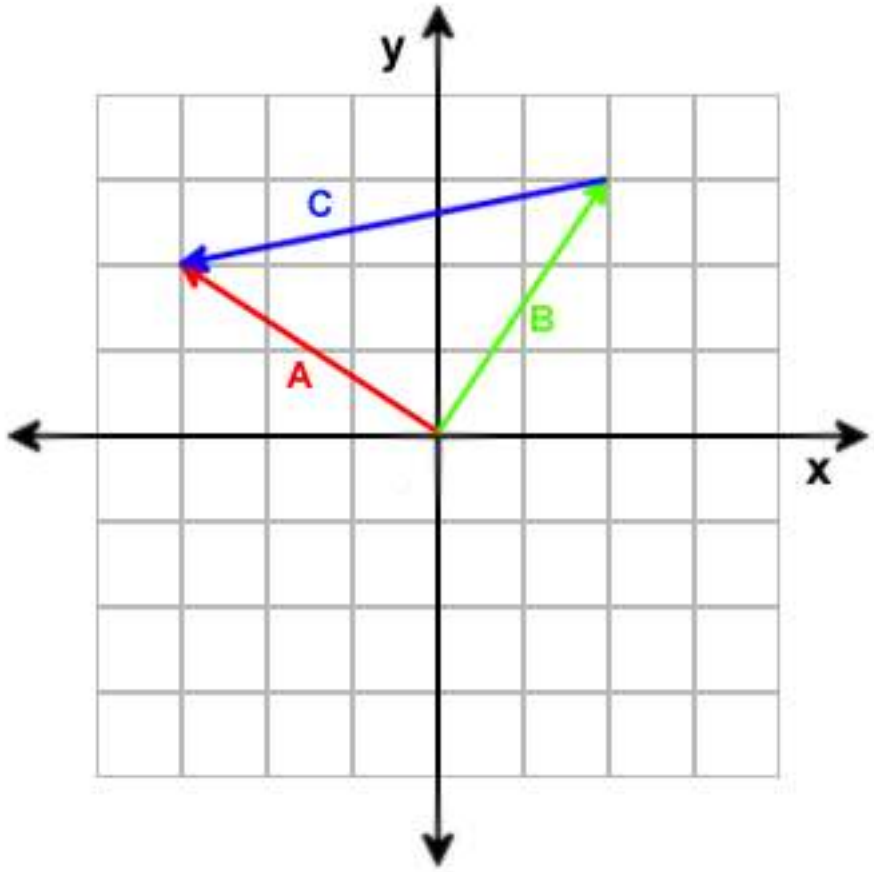


Figure 3. Subtraction of vectors.

Subtracting two vectors results in a third vector pointing from the end point of the second vector to the end point of the first vector.

Vector multiplication

Multiplication of two vectors is done by multiplying the corresponding components.
Example:
A = [3, 5, 1]
B = [2, 2, 3]
A * **B** = [3 * 2, 5 * 2, 1 * 3] = [6, 10, 3]

Magnitude of a vector

The magnitude of a vector is denoted by two vertical stripes on either side of the vector:
$||\mathbf{V}||$

It can be calculated with Pythagoras' theorem. For those of you that don't know or forgot it:
$||\mathbf{V}|| = \text{square-root } ((\mathbf{V}_x)^2 + (\mathbf{Vy})^2 + (\mathbf{Vz})^2)$

Scaling a vector

Scaling a vector is done by multiplying each component with a scalar. If the scalar is negative, the direction of the vector is also inverted.
Example:
A = [2, 1]
B = 2 * **A** = [4, 2]

C = -0.5 * **A** = [-1, -0.5]

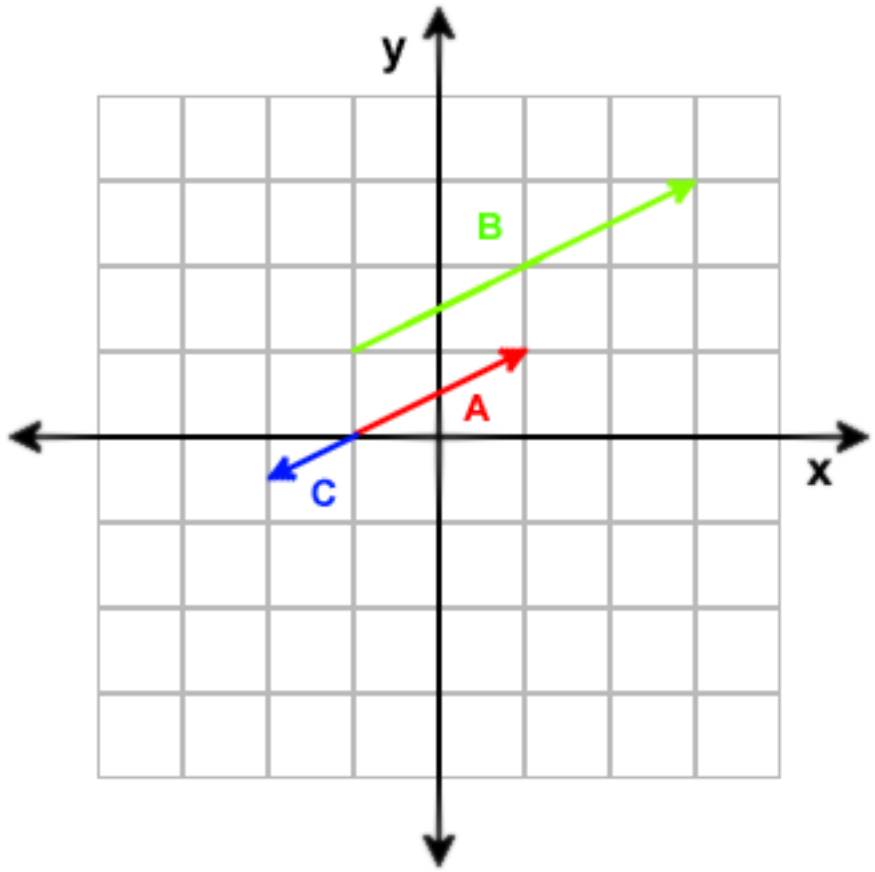


Figure 4. Scaling a vector.

Multiplying a vector with a scalar will only change the magnitude of the vector. If the scalar is negative the vector is also inverted.

Unit vectors

A unit vector is any vector of length one. This does **not** mean that all components must be 1.
Example:
A = [1, 1]
B = [0.71, 0.71]
C = [1, 0]
||**A**|| = square-root($1^2 + 1^2$) ≈ 1,41 → **A** is NOT a unit vector.
||**B**|| = square-root(0.71^2, 0.71^2) = 1 → **B** is a unit vector.
||**C**|| = square-root(1^2, 0^2) = 1 → **C** is a unit vector.

For many calculations in lighting algorithms the vectors must be unit vectors. You can convert an arbitrary vector to a unit vector, this is called *normalization*. To normalize a vector you must divide each component of the vector by its length.
Example:
V = [4, 4]
||**V**|| = square-root($4^2 + 4^2$) = square-root(32) ≈ 5.66
$\mathbf{V}_{normalized}$ = [4 / 5.66, 4 / 5.66] ≈ [0.71, 0.71]

And to prove that it's actually normalized:
|| $\mathbf{V}_{normalized}$ || = square-root($0.71^2 + 0.71^2$) ≈ 1

Dot product

The dot product is a very important operation in graphics programming. The interesting thing about it is that the result is not a vector but a single value. It can be used to find the angle between two vectors.
The dot product is defined as:
$\mathbf{A} \cdot \mathbf{B} = ||\mathbf{A}|| * ||\mathbf{B}|| * \cos\angle(\mathbf{A}, \mathbf{B})$

The dot product is the length of vector **A**, multiplied by the length of vector **B**, multiplied by the cosine of the angle between **A** and **B**. Because in shader programming we usually don't know the angle between **A** and **B**, this is of little use to us. Fortunately, there is another way to calculate the dot product:
$A \cdot \mathbf{B} = \mathbf{A}_x * \mathbf{B}_x + \mathbf{A}_y * \mathbf{B}_y + \mathbf{A}_z * \mathbf{B}_z$

So now we know how to calculate the dot product. To find the angle between **A** and **B**, we simply divide this value by the product of ||**A**|| and ||**B**||. In the case of normalized vectors the dot product is equal to the cosine of the angle between **A** and **B** because ||**A**|| and ||**B**|| are both 1.

Coordinate spaces

Another important concept to grasp is coordinate spaces. You will be dealing with it in pretty much every shader you will write. You've probably already used them without even knowing it.

In computer graphics we use the 3D Cartesian *coordinate system* to represent vectors. This means that a vector is represented by an x, y and z coordinate that give the displacement from the origin in the direction of an axis. But what does it mean when an object is at a certain point on an axis? Where is the origin and in what directions do the axes go? What values may be used to define a vector?

The answers to those questions are different in different coordinate spaces. The most common coordinate spaces used in 3D graphics are the local or object space, the world space, the camera space and the clip space.

Object space is the coordinate space in which vertex positions are stored. The origin or null vector (0,0,0) is at the origin of the model (which is the (x,y,z) position of the object in the world) and its axes go from left to right, from down to up and from the back to the front of the model. If you move or rotate an object in the world, the vertex positions in object space will not change, only the origin or axes of the object in the world change but the vertices are at the still position on those axes.

Object positions are usually stored in world space. The world space origin and axes are those that you see when you create a level in 3Dgamestudios' World Editor (WED). They are not defined by anything else.

3DGamestudio's "view entities" are in camera space. This means that the origin is at the camera and the axes point up/down, right/left and into your monitor.

Lastly there is clip space. In clip space, all coordinates range from [-1..1] on all axes. In clip space, only the objects that are visible (the objects that are within the viewing volume) are present and far away objects are smaller than close-by objects.

What is a matrix?

A matrix is a grid of values. It's basically a row (or column) of vectors. A matrix is usually represented by a multidimensional array in code. Example:

M = 1, 0, 0,
2, 0, 0,
0, 3, 0

M is a 3x3 matrix with:
M[0, 0] = 1
M[1, 0] = 2
M[2, 1] = 3
All other values of **M** are at 0.

By multiplying all vertices of an object with a matrix it is possible to move, rotate and scale objects. These are called *transformations*. When multiplying two matrices the resulting matrix will encompass both transformations.

For more information on matrices, see the "Further Reading" section.

Further Reading

More information regarding shaders can be found on these websites:

Full DirectX documentation & HLSL syntax: http://msdn2.microsoft.com
Tutorial - Math Primer: http://triplebuffer.devmaster.net/file.php?id=5&page=0
Vector Math for 3D Computer Graphics:
http://chortle.ccsu.edu/VectorLessons/vectorIndex.html
Shader samples, documentation & tools: http://www.developer.nvidia.com

Contact

Tutorial written by Taco Cohen AKA Excessus.

Comments? Suggestions? Errors? Mail me at: taco.cohen@gmail.com