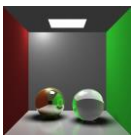


2015, 4th quarter

INFOGR: Graphics

Practical Assignment 2: Basic Shader Programming

The assignment:

The goal of the second practical assignment is for you to learn basic shader programming. As preliminary knowledge we expect you to have read and completed the previous tutorial assignment, and understand the workings and purpose of the things you implemented. Also we highly recommend you attend the presentation on *Shader Programming and Graphics Hardware* and use the slides as a reference (this talk will be given by the TAs on Thu, May 01 during the lecture). This will save you a lot of speculation and research on the workings of shaders. We expect that you know what shaders are on a high level and what a vertex and pixel shader is (what they get as input and what they have as output) and what their relationship is. If you don't know this, definitely attend the presentation and do some research of your own.

For this assignment, you will create a program that is much like the one in the previous assignment, with some differences. In the previous assignment you spent most of your time writing C# code and using XNA functionality, while using the `BasicEffect` object to influence the rendering process. This time the emphasis is on writing shader code using the *High Level Shader Language*. Using HLSL you will create your own effect to replace `BasicEffect` in XNA.

Like the previous assignment, you have to hand in your complete end result. This means all your source code, project files, content files, a readme (see below) and in addition, we also want *screenshots* of intermediate results. See the related text in the assignments that specifies what should be visible on the screenshot and how you should name the file (missing screenshots or violations of the naming conventions will have a negative effect on your grading). The screenshots have to be in the PNG image format.

The following rules from the previous assignment still apply:

- Your code has to compile and run on the machines in BBL 175, so if you work on other computers make sure to do a quick check there before you submit it. If this requirement isn't met, your work cannot be graded and your grade will default to 0.
- Please *clean* your solution before submitting (i.e. remove all the compiled files), see *Appendix 1A* of the Tutorial for more information. After this you put the solution directory in the zip file.
- When grading, we want to get the impression that you really understand what is happening in your code, so your source files should also contain comments to explain it. (Besides, every good program should be properly documented!).
- Finally, we also want to see a consistent and well readable coding style. Use indentation to indicate structure in the code for example. Don't worry about this too much though. If it is readable and consistent throughout the whole project, you should be fine.

Grading:

If you do all the assignments and all the above perfectly, you get an 8. You can earn the additional 2 points by doing the two bonus assignments to get the perfect grade 10. Each bonus assignment is worth 1 point. Note that these are considerably harder than the other assignments and require you to do some research of your own.

Reference material:

The above mentioned introduction presentation and related slides should give you a good initial start. But, of course, further references are needed if you want to become a good graphics programmer. The HLSL reference guide in the *Microsoft MSDN library* gives very useful information about data types and built in functions and provides several examples for good HLSL programming.
References about the theoretical background are listed throughout the assignment. But notice that we also expect you to do your own research if you feel that this is necessary for you to solve a particular task. As a hint, *Wikipedia* often proves to be quite helpful (e.g. you could look for *List of common shading algorithms* when working on Part 2 of the assignment).

Deliverables:
A ZIP-file containing:

1. **The contents of your (cleaned) solution directory**
2. **The read-me (in the .txt file format)**
3. **A directory with several screenshots**

The contents of your solution directory should contain:

(a) Your **solution file** (GraphicsPractical2.sln)
(b) All your **source code**
(c) All your **project** and **content files**

The readme file should contain:

(a) **The names and student IDs of your team.**
2-3 students; penalties for submitting with less or more team members will apply
(b) **A short statement specifying how you split the work among the team members.**
Briefly explain how you worked together and/or split the work. No long explanation needed, just a short description, such as "we regularly met and implemented everything together" or "Tom implemented A and B, Paul did C and D, and the rest we did together" or whatever you did. Obviously, we hope not to find any statements such as "Tom and Paul implemented everything and Frank just put his name on the delivery". Be aware that we will randomly select a few groups for a short verbal interview in order to verify that your statements are correct.
(c) **Short (!) comments on your code related to the mandatory assignments.**
Some assignments (like Lambertian shading and Phong shading) have requirements of what should be in the comments.
(d) **A statement about what bonus assignments you have implemented (if any) and related information that is needed to grade them, including detailed information on your implementation.**
We will not make any wild guesses about what you might have implemented nor will we spend long times searching for special features in your code. If we can't find and understand them easily, they will not be graded, so make sure your description and/or comments are clear.

The screenshots directory should contain:

All screenshots as specified in the assignment (cf. references in the actual assignment text)

Just to make sure, the contents of the zip file should look like this:

```
/GraphicsPractical2.sln
/GraphicsPractical2
    /GraphicsPractical2
    /GraphicsPractical2Content
/screenshots
    /normalcolors.png
    etc.
/readme.txt
```

So put the contents of your solution directory and the readme file (in the .txt file format) in the **root** of the zip file and put all the screenshots in a directory named "screenshots". Note that any violation to these rules will have negative consequences for your grading. Also notice that the readme file should be well readable. It is part of the program that you are producing, so the rules about "consistent and well readable coding style" apply to it as well.

Mode of submission:

- Upload your zip file before the deadline via the SUBMIT system at http://www.cs.uu.nl/docs/submit/
- Make sure to upload it to the correct entry, i.e. not the ones for late delivery if you are submitting on time
- Note that we only grade the latest submitted version of your assignment, so if you upload to the late delivery entries your earlier submission will be discarded.

Deadline:

- **Tuesday, June 2, 2015, 23:59**

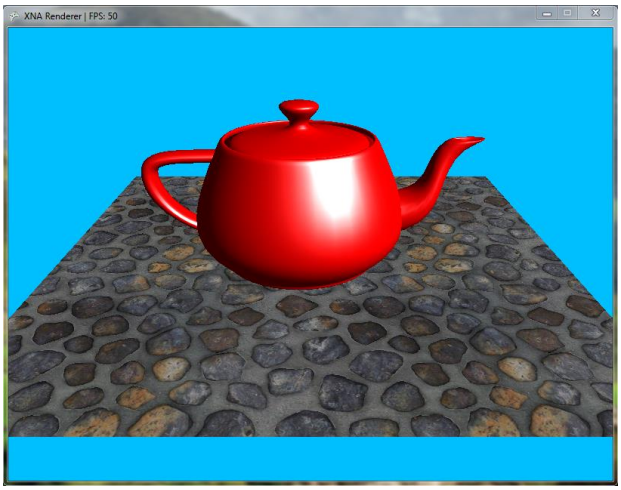
If you miss this deadline, there will be a second entry in the submit system to upload your solutions. It is open 12h longer, i.e. till **June 3, 2015, 12:00.** Uploading to this entry will result in a deduction of 0.5 in your grading.

If you miss the second deadline as well, there will be a third entry in the submit system to upload your solutions. It is open 24h longer, i.e. till **June 3, 2015, 23:59**. Uploading to this entry will result in a deduction of 1.0 in your grading.

If you the miss third deadline as well, contact the instructor of the course as soon as possible. Based on your reasons for the delay, he will decide if you get further reductions or if your assignment will be graded with a 0.

If you don't miss any deadlines, but upload your submission to the wrong entry in submit, you will get the same deduction as if you had submitted it too late, so be careful.

Content Overview



Part 0: Setting up

Part 1: Procedural texturing

1.1 Coloring using normals
1.2 Checkerboard pattern

Part 2: Lighting

2.1 Lambertian shading
2.2 Ambient shading
2.3 Phong shading
2.4 Non-uniform scaling problem

Part 3: Texturing

3.1 Texturing a quad using U,V-coordinates

Part 4: Texturing

4.1 Gamma correction
4.2 Normal mapping

Part 0: Setting up

You can use the provided minimal *framework* in the assignment files as a starting point. Explanation on the provided classes in the framework can be found as comments in the code files. If you want to start from scratch or make adjustments to the framework, you are free to do so. If you start from scratch, make sure you use the *Windows Game (4.0)* template in Visual Studio.

The provided framework has the camera's eye positioned at (0, 50, 100) and lets it look at the 3D-origin (0,0,0). The positive y-axis is "up". The window size is set to 800x600 pixels. You are free to change this, but for the *screenshots* the camera **MUST** be positioned this way, and the window size **MUST** be 800x600 pixels. *Back-face culling* is turned off.

Shaders

Probably most computer programs you have written so far are programs that run on your computer's *Central Processing Unit* or *CPU*. A *shader* is a (small) program that runs not on your CPU, but on your *Graphics Processing Unit* or *GPU*, which is a part of your video card. Shaders can be written using a *shading language*, which is a highly specialized programming language that has many built in functions and data types that are often used in (3D) graphics programming. More importantly, these functions are *hardware accelerated*, which means they are very fast!

The first big difference between a CPU and a GPU are the amount of processing units. Where a standard modern CPU has around 2 to 8 physical cores, with a small number of arithmetic units (ALUs/FPUs) in each core, a GPU can have HUNDREDS of ALUs and FPUs and tens of cores, making it capable of doing massive *parallel processing*. Technically, a GPU is an advanced version of a vector processor (SIMD-processor) that combines a uses large numbers of arithmetic units (performing actual calculations) to operate in parallel. For efficiency, several ALUs/FPUs run the same instructions at a time. This, in combination with a special memory and caching architecture ("stream processing") leads to extremely high computational throughput (a large number of computations can be done in parallel).

This is a very desirable feature for rendering 3D computer graphics. To render a 3D scene, the same calculations have to be applied to huge amounts of vertices and pixels. To illustrate this, think about modern video games, where millions of triangles can be drawn on the screen in real time, using the same operations for each vertex and pixel. The GPU makes use of this premise and is optimized to take huge batches of data in its RAM and performing the same kind of specialized operations over and over again.

The second big difference is the amount of diversity and complexity that the cores can handle. CPUs can do any kind of operation at reasonable speeds, whereas GPUs specialize in doing certain kinds of operations very fast. Optimized GPU operations are commonly used in graphics programming and consist of *vector calculations*, *matrix multiplications* and manipulating specialized data structures. It should be noted that although the GPU has much more processing power than the CPU, each individual core of the GPU is much less complex than that of the CPU. A very simple but important difference between the two is that the GPU sometimes has a very hard time dealing with *dynamic flow control* (e.g. `if/else` statements). Using flow control statements in your shader code is perfectly possible, but it can have a major impact on the speed of your program. Technically, modern GPUs are able to execute dynamic flow control – this is not the reason for the performance penalties – but it can lead to *instruction stream divergence*. This means that the parallel vector processors make different decisions on the if/else statement and cannot anylonger work in parallel. Once the massive amount of parallel units is disabled, the remaining performance is subpar in comparison to ordinary CPUs, which are optimized for general code.

Nowadays many parts of the graphics pipeline have been opened up for programmers and they can be programmed using shaders. In this assignment we will work with two kinds of shaders that will influence different parts of the graphics pipeline: the *vertex shader* and the *pixel shader*. The vertex shader is a program that works on *every single vertex* of a 3d model. This is followed later in the graphics pipeline by a pixel shader. The pixel shader is a program that works on *every single pixel* that gets produced by the graphics pipeline.

The effect file and the High Level Shading Language

In XNA we write shaders in *effect files*. You are advised to open the provided effect file (Simple.fx) and read this text alongside it.

Techniques and passes

An effect file contains one or more *techniques*. A technique consists of one or more rendering *passes*. A rendering pass

consists of a model entering the rendering pipeline, undergoing a series of manipulations inside the pipeline and when those are completed exiting the pipeline as a set of pixels ready to be drawn on the screen.

For this assignment you will only create techniques that consist of a single *pass*. Each pass has some parameters that influence the rendering process. At the very least it has its own vertex shader and pixel shader. In the pass it is specified which functions will be used as a vertex and pixel shader. A technique called `Simple` is present in the provided effect file. It has only one pass, and the vertex and pixel shader are created from the functions `SimpleVertexShader` and `SimplePixelShader`.

HLSL

DirectX uses a shading language called *High Level Shading Language* (*HLSL*), a language that strongly resembles the programming language C. It is actually very easy to use. Assuming you are familiar with C#, HLSL is far less complex. It does contain some subtleties that you should be aware of, and a few things that require some explanation.

HLSL data types

HLSL has many built in data types. Examples are the `float` data type (like in C#), but it also has for instance the `float3` data type, which represents a 3-dimensional vector, and the `float4x4` data type, which represents a 4x4 matrix of floating point numbers. You are advised to thoroughly read every section and subsection(s) in the MSDN page on Language Syntax (DirectX HLSL). A bit hard to find, but probably one of the most important subsections is the one on Per-Component Math Operations, a sub-subsection of the Data Types (DirectX HLSL) subsection. This shows how you can access the individual components of the data types. Also make sure you look into the *Texture* and *Sampler* data types, which are a bit more complex than the primitive data types (but not hard to use). Also try to experiment a bit with the syntax, sometimes you can write your code in a much more intuitive (and thus more readable) fashion than you would expect.

HLSL functions

HLSL also has many *built-in functions*, that work on the built in data types. One example is the `mul` function, which can do scalar, vector and matrix multiplications on the appropriate data types. It is already being used in the provided effect file. Familiarize yourself with the functions on the MSDN page on Intrinsic Functions (DirectX HSLS). We will *not* be using *Shader Model 4* or higher, so functions labeled as such in the *Minimum shader model* column can be omitted. Note that Shader Model "11" in the MSDN library should be read as Shader Model 1.1, so you can definitely use these functions as well!

Input/output structures and semantics

Vertex shaders have specific input and output. Look into the `VertexShaderInput` and `VertexShaderOutput` structures in the provided effect file. Notice that every member of the structure has a colon (:) behind it , followed by a word and an index. This is called a *semantic*, it helps the graphics card to understand what the data in this member represents. Common uses are *positional* data, *color* data, *surface normals* and *texture coordinates*. The `POSITION0` semantic is already being used for the position data in the structures. By using this semantic, the position data of the input vertices ends up in that member variable. You will read about semantics in the Semantics (DirectX HLSL) section of the MSDN library (a sub-subsection of the Variables (DirectX HLSL) subsection). The vertex shader output should at least contain positional data with the `POSITION0` semantic, which however you cannot use directly in the pixel shader. The pixel shader output is always a pixel color, since it is the last step in the rendering pipeline that can be influenced by shaders.

Top-level variables

The variables at the top (the `float4x4` matrices, and possibly other variables that you later add yourself) are called *top-level variables*. They are *global* variables and can be adjusted *at run-time*. More about that in the next section.

Tips and hints

1. Read the article about GPU Flow-Control Idioms on how to deal with flow control statements. After reading this article you can better decide for yourself whether it makes sense to put effort into avoiding a flow control statement.
2. The semantics in the shader code directly correspond to the last two arguments of a `VertexElement` in the `VertexDeclaration` (if you forgot, check the Tutorial assignment again), e.g. `VertexElementUsage.Position` and index `0` corresponds to the `POSITION0` semantic in the shader code.
3. You can use the data with the `POSITION0` semantic in the vertex shader, but not in the pixel shader.
4. If you want to pass data from the vertex shader to the pixel shader, you can safely use the `TEXCOORD1` to `TEXCOORD15` semantics.
5. The values that you pass into the `VertexShaderOutput` struct in the vertex shader are *not* the same values that you get as input for the pixel shader. A vertex shader has a single vertex as input, and a pixel shader works on each pixel in a triangle defined by 3 vertices. Therefor by default *all* the values in the input struct for the pixel shader, regardless of the semantic, have been linearly interpolated between these three vertices, to represent the value at the current pixel!
6. There is no overhead to calling a function in HLSL. Every function definition in HLSL implicitly has the `inline` modifier, meaning that the HLSL compiler replaces every occurrence of a function call with the body of that function. The result is a larger size of the compiled shader, but also better performance.
7. In other resources (books, websites) the *World matrix* is sometimes called the *Model matrix*, and the pixel shader is often called *fragment shader*. This terminology is also adopted by the graphics API *OpenGL* and its shading language *GLSL*.

Effects in XNA

In the tutorial assignment you worked with a class called `BasicEffect`. `BasicEffect` is a subclass of the `Effect` class. The `Effect` class, in combination with an *effect file*, enables you to use and manipulate shaders. The example effect file is loaded into an `Effect` object using the `Content` pipeline:

```
Effect effect = this.Content.Load<Effect>("Effects/Simple");
```

You can assign values to the top-level variables in your effect file at run-time using XNA, using the `Parameters` property of your `Effect` object. The `Parameters` property is a *dictionary*. A dictionary works like an array, but uses strings instead of indices to look up values. Each top-level variable can be looked up by using it's name as an index in the dictionary, and can be changed by the function `SetValue()`. Example: to pass an XNA matrix (say, the *identity matrix*) to the `float4x4 World` variable, you would write:

```
effect.Parameters["World"].SetValue(Matrix.Identity);
```

Loading a model

In the provided files of this assignment is a file containing a 3D model. Look for the file with the suffix ".fbx". We can load it into an XNA model object using:

```
this.model = this.Content.Load<Model>("Models/Teapot");
```

You can assume XNA's `Model` class is fully optimized and uses vertex- and index buffers on the graphics card to render itself and uses the appropriate vertex type so all the data from the model file is present in the vertices. The model's vertices contain *position data* and *pre-calculated surface normals*, so they don't have to be calculated manually like for the terrain. The model contains no texture information (i.e. no UV-coordinates).

Drawing the model

We want to draw our model using our own effect. The `Model` class has a `Draw()` method, but that doesn't let us specify our own custom effect for drawing. We have to take a slightly different approach. A model consists of one or several *meshes* (sub-models) and a mesh consists of one or several *mesh-parts*. Each mesh-part can have it's own effect (i.e. it

has a property called `Effect`). The model that was provided as part of the assignment has only one mesh, consisting of only one mesh-part. We want our model to use our own effect, so we write:

```
this.model.Meshes[0].MeshParts[0].Effect = effect;
```

In the `Draw()` method of our `Game` class, we have to specify which technique our effect should use. The effect has a property called `Techniques`, which is also a dictionary (see above). The `CurrentTechnique` property represents the technique that is currently being used by the effect. We set it to our `Simple` technique:

```
effect.CurrentTechnique = effect.Techniques["Simple"];
```

And we pass appropriate values to the top-level variables.

Finally, we draw the model. To draw the model, we have to draw each of the model's meshes (fortunately only one in our case) separately. This mesh now knows which effect it has to use, so you don't have to call `pass.Apply()` for all passes in the `CurrentTechnique` like in the tutorial, the `ModelMesh` object does this for you when you call it's `Draw()` method:

```
ModelMesh mesh = this.model.Meshes[0];
mesh.Draw();
```

Running the program should show you the silhouette of the model in a solid red color!

Part 1: Procedural texturing

The following exercises have to be implemented mostly in your effect file.

1.1 Coloring using normals

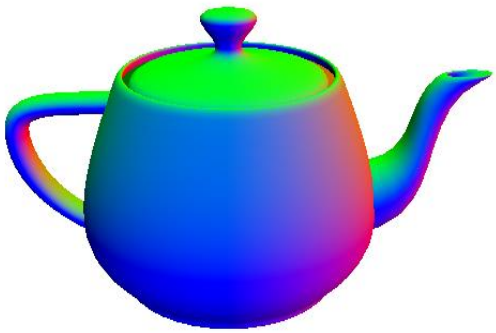
The model would look a lot better if it had some nice colors, instead of being solid red. You can add some color by using the xyz value of the normals (which lie in the [-1, 1] range) as RGB color values (in the [0, 1] range). To do this, you need to let the shader input accept the normal data from the model (just like the position data), and output some extra data to the pixel shader, so it can apply the right color.

Exercise 1.1: Implement this in the `NormalColor()` function in the effect file and use the function in your pixel shader.

Note 1: If a struct has more than one member variable, you have to initialize it before you can use it. For the vertex shader output structure, this is done like so: `VertexShaderOutput output = (VertexShaderOutput)0;` This sets all it's members to `0`. This is already done in the provided effect file.
Note 2: Do you have to clamp the normals so they lie in the [0, 1] region? Just try!
Note 3: In the next assignment, the surface color is replaced by a checkerboard pattern, however make sure we can still use the `NormalColor()` method, or this assignment cannot be graded!

Hint 1: You can use the swizzle shortcuts of `float3` and `float4`, e.g. `normal.xyz`.
Hint 2: Look into the `NORMAL0` and `COLOR0` semantic.

Screenshot description:
The model with a smooth, varying surface color depending on the normals of the vertices.
The file name should be “normalcolors.png”.



Normals as RGB colors

1.2 Checkerboard pattern

You can apply a simple texture pattern, without the need of a texture map image. By using just the x-value of the pixels, you can create a black and white stripe pattern (see slides on procedural texturing and the book for pseudo-code on the stripe pattern). A very simple extension to this idea is the checkerboard pattern, which you can create by also using the y-value.

But how about this: instead of creating a black and white checkerboard pattern, let the "white" squares use the normals for coloring, and for the "black" squares you use also the normals, but in the opposite direction! Think carefully about what you need to do here. If you do all of this in the vertex shader, you only set the colors of the vertices and they get linearly interpolated when they get to the pixel shader. Thus, you will get a smoothly shaded surface, and not a checkerboard, and for this assignment it has to be a checkerboard pattern.

When implementing it in the pixel shader, you need the 3D position of every pixel. However the position data (with the `POSITION0` semantic) that the pixel shader gets as input contains 2D screen positions (besides the fact that you can't even use it!).

Exercise 1.2: Implement this in the Procedural`Color()` function in the effect file and use this function in your pixel shader.

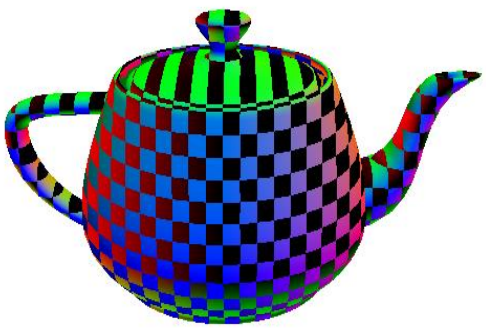
Note 1: The lecture on texturing is given quite late (i.e. closer to the deadline w.r.t. other assignments). If you want to wait for the lecture before implementing this assignment, you are advised to do the other assignments first!
Note 2: Use only the x- and y-values of the pixels. If you also use the z-value, you will probably see a weird circular pattern on the model's surface.
Note 3: In the next assignment, the checkerboard surface pattern is replaced by a single color influenced by lighting calculations, however make sure we can still use the `ProceduralColor()` method, or this assignment cannot be graded!

Hint: Each component of the vertex shader output is linearly interpolated to find the value for each pixel. 3D positions are no different. Remember that the model is made up of triangles.

Reference material for the theory:

- Slides from lecture 6 on *Texture mapping* (the part on procedural texturing contains pseudo-code)
- 3rd and 2nd edition of the book: Chapter 11.1

Screenshot description:
The model, with its surface set to a checkerboard pattern. What would be the white cell should have a surface color that depends on the normal in that point, what would be the black cell should have a surface color that depends on the opposite of the normal in that point.
The file name should be "checkerboard.png".

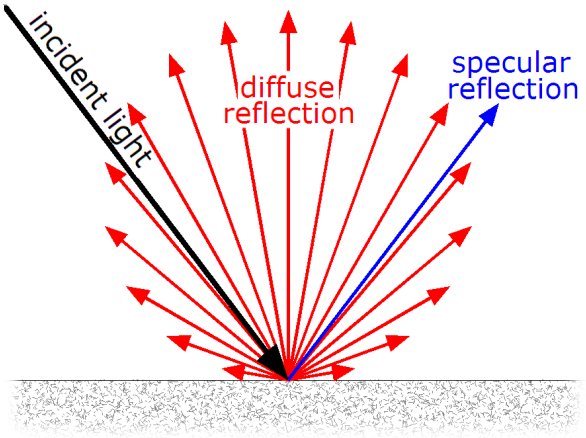


Checkerboard pattern

Part 2: Lighting

2.1 Lambertian shading

For this assignment you will implement a *light reflectance model* to make your 3D object look more realistic. This particular lighting model is called the *Lambertian reflectance model*. In computer graphics it is often referred to as *n dot l lighting*.



It is based on *Lambert's cosine law*, which states that the surface color *c* is proportional to the angle between the surface normal and the direction to the light source: $c \propto \cos \Theta$, or in vector form: $c \sim \vec{n} \cdot \vec{l}$. (The symbol ~ denotes "is proportional to", · is the dot product of two vectors.)

When also taking into account the diffuse color of the surface c_r, the intensity of the light c_l and the fact that surfaces pointing away from the light should not receive any light, the equation becomes: $c \sim c_r \cdot c_l \cdot \max(\vec{0}, \vec{n} \cdot \vec{l})$.

Lambert's model is a *diffuse* reflectance model, which represents surfaces that are completely *matte*. As illustrated in the figure on the right, this diffuse model assumes that the incident light is reflected into many directions. This is in contrast with a specular reflection model, c.f *assignment 2.3 Phong Shading*, where the light is reflected in a single direction. Unlike in the image, we will use an idealization where the light is reflected in all directions with equal intensity in the hemisphere surrounding the surface (Wikipedia), i.e. the red arrows should all have equal length.

Exercise 2.1: Implement *Lambertian* (*n dot l*) *lighting* in your shader code.

Note: Think carefully where and how you want to implement this. You can decide for yourself whether you want to use a *point light* or a *directional light.* Please state your choice in the *read-me file*. If you decide to use a point light, you *do not* have to take the decrease of light intensity into account (i.e. *light attenuation*), that happens in the real world when moving further away from the light's origin.

Hint 1: You can use the `DiffuseColor` component of the `Material` struct, which represents the color of the surface.
Hint 2: You can pass colors as `Vector4` to the shader code. Use `Color.ToVector4()`. A `Vector4` in XNA corresponds directly to a `float4` in the shader code.
Hint 3: You need to define a light source (just it's position is enough) in your XNA code and pass it to the shader for calculations. Best use a top-level variable for this.
Hint 4: To do correct lighting calculations using the normals, you have to take the World matrix into account. If the model gets rotated, so must the normals (we will test this when reviewing your code). One way to do this is extract the top-left 3x3 matrix out of the World matrix, which holds the rotation and scaling part of the World transformation, apply that to the normals and finally *normalize* them so that any scaling is removed. The extraction of the top-left 3x3 matrix of the world matrix can be done by explicitly casting it to a `float3x3`:

```
float3x3 rotationAndScale = (float3x3) World;
```

Reference material for the theory:

- Slides from lecture 3 on *Curves, surfaces and shading*
- 3rd edition of the book: Chapter 4.5.1 and 10.1.1
- 2nd edition of the book: Chapter 4.5.1 is not in here, but 10.1.1 can be found in 9.1.1.

Screenshot description:
The model, lit by either a *directional light* in direction (-1, -1, -1) or by a *point light* on position (50, 50, 50). The `DiffuseColor` property of the material should be set to XNA's `Color.Red`. The normal color from the previous assignment is replaced by this `DiffuseColor`. The effect of the light should be clearly visible.
The file name should be "lambert.png".

Lambertian shading

2.2 Ambient shading

You may have noticed that the shadows on the 3D object are completely black. In the real world, small amounts of light are reflected of every surface, so even shadows are never totally absent of light. Precisely calculating every small reflection of every possible surface is way too computationally expensive. A cheap approximation is to use *ambient lighting*, i.e. add a small constant amount of light to the surface.

Exercise 2.2: Implement *ambient shading* as an addition to Lambertian shading to avoid the completely black shadows.

Hint: You can use the `AmbientColor` and `AmbientIntensity` properties of the material.

Reference material for the theory:

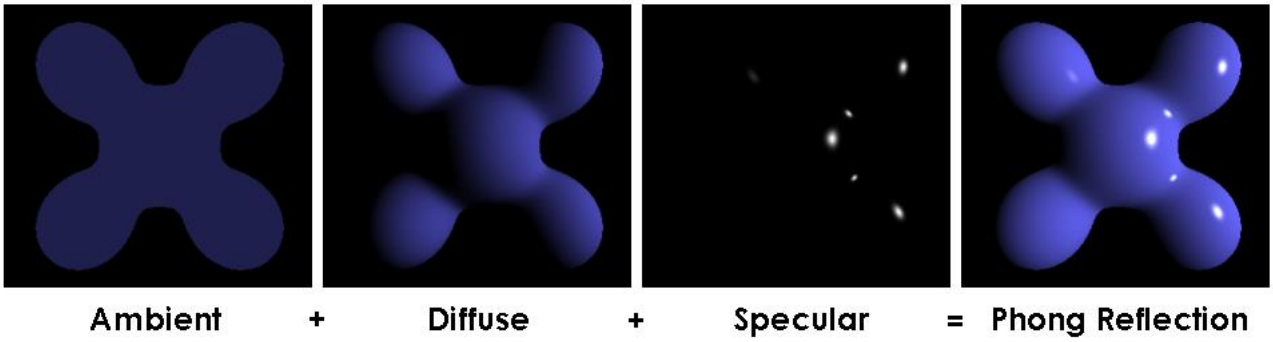
- Slides from lecture 3 on *Curves, surfaces and shading*
- 3rd edition of the book: Chapter 4.5.3
- 2nd edition of the book: Chapter 9.1.2

Screenshot description:
Use the same settings as in the above assignment. Use `Color.Red` for the `AmbientColor` and `0.2f` for the `AmbientIntensity`.
The file name should be “ambient.png”.

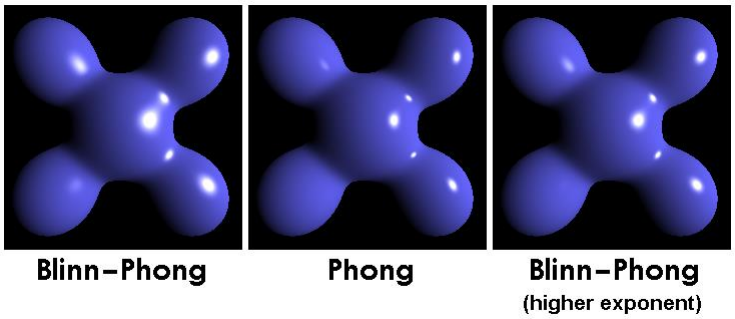
Without ambient

2.3 Phong shading

Sometimes the simple Lambertian reflectance model is not enough to correctly represent a surface. Consider for instance surfaces that reflect a large amount of light without changing it's color, like metal. To this end another model has been developed as an extension to Lambert's model, called the *Phong reflectance model*. It is used to get a *specular*



highlight on a surface.
Phong shading has an optimization called *Blinn-Phong shading*. It is faster and somewhat easier to implement, and the



result looks virtually the same:

Exercise 2.3: Implement *Phong* or *Blinn-Phong shading* (whichever you prefer, but put in the read-me which one you choose, no motivation is needed). Implement it as an addition to the Lambertian and ambient shading.

Note: Again think carefully about how and where in your shader code you will implement this, and show us your motivation in the read-me.

Hint 1: The specular highlight depends on the camera's eye position, so you also need that in your shader code.
Hint 2: You can use the `SpecularColor`, `SpecularIntensity` and `SpecularPower` properties of the `Material` struct.

Reference material for the theory:

- Slides from lecture 3 on *Curves, surfaces and shading*
- 3rd edition of the book: Chapter 4.5.2 (and 10.2.1 for further details)
- 2nd edition of the book: Chapter 9.2.1 (same as 10.2.1 in 3rd edition; no chapter 4.5.2 though)
- Look for “List of common shading algorithms” on Wikipedia

Screenshot description:
The model, lit by either a *directional light* in direction (-1, -1, -1) or by a *point light* on position (50, 50, 50). The specular highlight should be clearly visible. The `SpecularColor` should be set to `Color.White`, the `SpecularIntensity` to `2.0f` and the `SpecularPower` to `25.0f`. All other settings should be the same as in the previous assignments.
The file name should be “phong.png”.

Phong shading

2.4 Non-uniform scaling problem

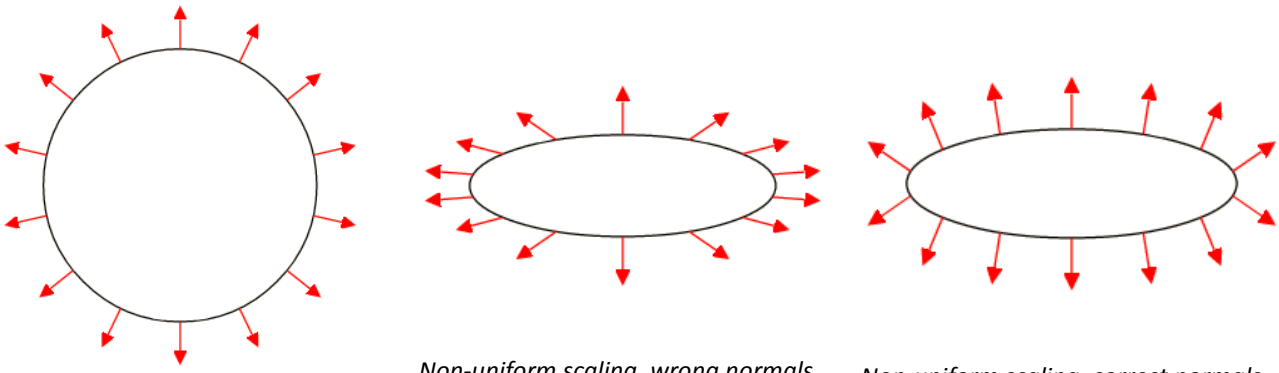
So far we have only assumed the use of a *uniform scaling matrix*. However a *non-uniform scaling matrix* is also often used in computer graphics. A non-uniform scaling matrix is a scaling matrix with different individual scaling in the x-, y- and z-direction. When it is applied to a 3D model, using the top-left 3x3 part of the World matrix to transform the normals (even when normalized) produces wrong normals.

The three images below are schematics of a sphere transformed by a scaling matrix. The red arrows represent the surface normals.

The sphere in the *left image* is transformed with a uniform scaling matrix. The normals correctly represent the surface orientation because they are perpendicular to the surface.

In the *middle image*, the sphere is transformed with a non-uniform scaling matrix. The normals no longer accurately represent the surface's orientation as they are not perpendicular to the surface. This is what happens when using the top-left 3x3 part of the World matrix to transform the normals.

This problem is solved by using the *inverse-transposed* of the World matrix instead of the top-left 3x3 "trick". The normals in the *right image* are correct for the non-linear scaling, because the inverse-transposed of the World matrix is used to transform the normals.



Uniform scaling, correct normals

Non-uniform scaling, wrong normals

Non-uniform scaling, correct normals

Exercise 2.4: Fix the non-uniform scaling problem by transforming the normals with the *inverse-transposed* of the World matrix. Calculate this matrix from the World matrix (in XNA) and pass it to your shader code to replace the top-left 3x3 matrix method.

Hint: You don't have to do the math of inverting and transposing the matrix yourself. XNA's `Matrix` class is perfectly equipped to handle these kinds of calculations for you!

Reference material for the theory:

- "Normal Transformation"

Screenshot description:
The model, scaled by `10.0f` in the x-direction, `6.5f` in the y-direction and `2.5f` in the z-direction, while still reflecting the light in the correct way. Use the same settings for the material and lighting as in the previous three exercises.
The file name should be "scaling.png".

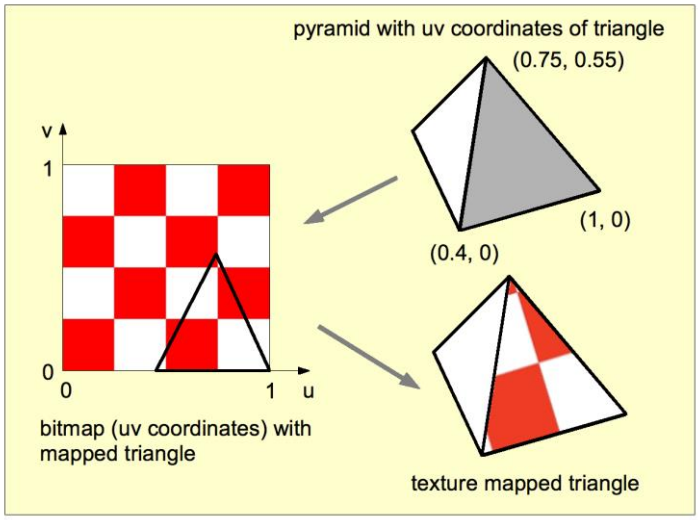
Incorrect lighting

Correct lighting

Part 3: Texturing

3.1 Texturing a quad using UV-coordinates

Some surfaces are more complex than simply a color and some shading. Take for instance a brick wall. A brick wall consists of bricks and cement, which both have complex textures and consist of many different colors (e.g. different shades of red, orange, gray etc.). To model every bump in every brick in 3D and to assign individual colors to every square millimeter of the brick is far too much work, not to mention a huge performance hit on your computer. Therefor, a more common and easier approach is to take a picture image of a brick wall and somehow stick or project it onto a 3D model that has the simplified shape of the brick wall (e.g. a 3D *quad*). We call this projected image a *texture*.



As you know, a 3D scene consists of triangles. You can apply a texture image to any triangle. This can be done by supplying the vertices in the triangle with a texture and so called *U,V-coordinates*. These are 2D coordinates in *texture space*: they describe what part of the texture is drawn onto the triangle (see image).

The provided framework has a definition for a *quad*. See the `SetupQuad()` method in the `Game` class. For this assignment you have to supply the quad with meaningful *texture coordinates* (UV-coordinates) so the provided image can be applied as a texture to the surface. You should draw it like the terrain in the tutorial assignment by using the `this.GraphicsDevice.DrawUserIndexPrimitives()` method. Note that for this quad, you do have to apply each pass in the effect explicitly before drawing, like you did for the terrain in the tutorial.
Actually applying the texture to the surface is done in the *pixel shader*. A *texture sampler* in the pixel shader can determine what the pixel color should be by using a texture image and the (automatically) interpolated U,V-coordinates.

Exercise 3.1: Apply the provided image in the assignment as a texture to a quad.

Hint 1: You can load the provided image into a `Texture` object in XNA with the content pipeline. This `Texture` object can be passed to the shader code as a top level `Texture` variable.
Hint 2: For the shader code, look into the "Texture" data type and the `TEXCOORD0` semantic in the MSDN library. Also look into the *Sampler* data type in the MSDN library (there are some examples there on how to use it). Use the function `tex2D()` in the pixel shader to sample the texture on the right location.
Hint 3: Since the 3D model does not have any UV-coordinates, applying a texture to the model isn't possible.

Reference material for the theory:

- Slides on "Texturing triangles", from the lecture on Texture Mapping.
- 3rd and 2nd edition of the book: Chapter 11.3 and 11.4

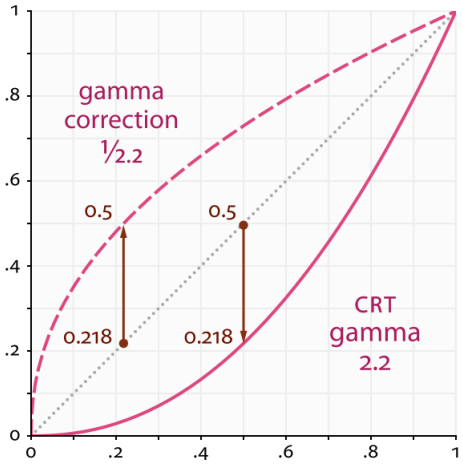
Screenshot description:
The model, "standing on" a quad that has a texture applied to it. Lighting should be turned off for this surface, i.e. the texture should be visible with it's unchanged colors, not darker, not brighter and no specular reflections. You may have to translate (move) the model so it appears to be standing on the surface. The file name should be "texture.png".

Textured quad

Part 4: Bonus assignments

4.1 Gamma correction

There is a difference between the actual physical brightness (or *luminance*) of the color channels (red, green and blue) and the color that is produced by our visual display device (e.g. computer monitor). This difference even varies for each type of display device. A measure we use for this difference is called the ***D**isplay **T**ransfer **F**unction* (also called *gamma function*). It was originally invented for CRT monitors, which had a non-linear relationship between a pixel's color intensity and voltage.



Fortunately, the DTF of a CRT monitor can be expressed quite accurately by the function $g(x) = N\,(x/N)^{1/\gamma}$, where the range of intensity values of the color channels is [0...N]. The image on the right shows the gamma functions for $\gamma = 1/2.2$ and $\gamma = 2.2$.

A perfect CRT has a linear DTF, i.e. a γ value of 1 (the diagonal line in the image). Such a CRT does not exist, we have to apply some pre-processing to the images, so they will appear correctly on the screen. We *pre-map* our image with the inverse of the DTF before we display it, so we get a linear response from the display device: $I_{screen} = g\,(g^{-1}\,(I_{image}))$. *Gamma correction* not only changes the intensity of the color channels, it also changes the proportions between red, green and blue.

With today's LCD monitors, gamma correction is less relevant. However you can still apply the gamma function to each color channel in each pixel to change the (aesthetic) look of your rendered images. Applying a function or algorithm to each pixel after all the rendering steps have been completed is called a *post-processing effect*.

Exercise 4.1: Apply the gamma correction function to each color channel in each pixel. Implement this as a post-processing effect in a new effect file called “PostProcessing.fx”.

Note: You could apply the gamma correction directly in your existing pixel shader function, but for this assignment it HAS to be implemented as a post-processing effect. Implementing this assignment as a post-processing effect also gives you a head start for some assignments in the next practical assignment.

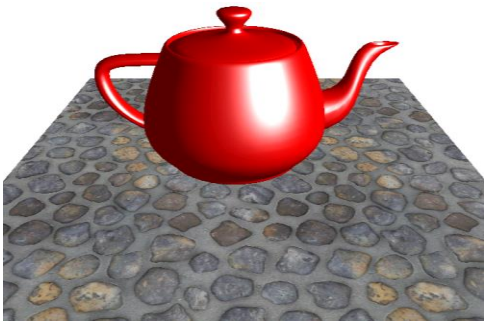
Hint 1: There are two ways to do post-processing:

1. You can let your program render to a texture instead of to the back-buffer by changing the *RenderTarget*. You can then draw a full screen quad, using the result of your render as a texture. You can then sample every pixel using the texture sampler, as you did for the previous assignment. This is the traditional way to do post-processing when using DirectX or OpenGL and requires good knowledge of the several matrix multiplications in the rendering pipeline. A post-processing effect is performed in *screen-space* instead of *world-space* (see the previous Tutorial assignment).
2. The *SpriteBatch* object has functionality that lets you draw a (full-screen) image and apply a pixel shader to it. This is a more elegant solution, but works only in XNA.

Hint 2: We can simplify the gamma function, because we use floating point color channels in the range [0...1].

Hint 3: The gamma function should be applied to each color channel individually (R, G and B). The gamma value can be the same for all channels though.

Screenshot description:
The model, with gamma correction as a post-processing effect using a γ of `1.5f`. As you can see in the reference image, if you implemented it properly the image will get brighter.
The file name should be “gamma.png”.



Gamma correction with $\gamma = 1.5$

4.2 Normal mapping

Normal mapping is a common technique for adding visual detail to a surface at relatively low cost. Instead of actually modeling the small-scale *bumpiness* of a surface by adding additional vertices, we will use shading to suggest the presence of additional geometry. By playing with the surface normals, we change the way the lighting calculations are done. When shading a point, we will not plainly use the actual normal of the geometry but we will disturb it by some small amount.

A typical way to determine this change is to do a texture look-up and add the color from the texture (RGB, which is a 3-vector) to the actual normal and re-normalize the result. Here you can see that images are not only used to be directly applied to a surface as a texture, but they can also be seen as an array of vectors, thus containing a lot of information other than just color. A texture used in this way would be called a *normal map* and for the effect to look good, you need an appropriate texture. A surface texture and it's corresponding normal map have been supplied in the assignment files.

Consider a normal map depicting the bricks of a wall. If we implement normal mapping naively, the following might occur: on one side of the wall the effect shows the bricks sticking out of the wall whereas on the other side of the wall, the same texture makes the bricks look as though they stick into the wall. This is what happens when you add the value from the normal map to the geometric normal without regard for the actual geometry, effectively treating the values in the normal map as vectors in world space.

For modeling convenience and re-use of normal maps, it is typical to consider so-called *tangent space* normal maps. Here we transform the vector from the normal map into tangent space before adding it to the geometric normal. The most important part of this is that we transform the vector such that its z-axis in the texture coincides with the geometric normal. This defines a consistent notion of "into the surface" and "out of the surface" for the normal map. We also align the x-axis of the vector coming from the normal map with the world space direction of the u-texture coordinate and correspondingly align the y-axis of the normal map vector with the v-direction. Look up the difference between a world-space normal map and a tangent-space normal map. Notice that the tangent space normal map is mostly blue, which is positive-Z, that is, along the original normal. This is ideal because it completely ties the information in the normal map to the geometry onto which it is applied and its mapping. However, it might be tricky to derive this transformation. This kind of solution would be considered portable.

Exercise 4.2: Add visual detail to the quad by disturbing the surface normals. Use the RGB values in the corresponding normal map as a 3-vector to distort the normals.

Note: The simplest way to implement this is to only take the existing flat surface (the quad) into account, and implement normal mapping only for this specific case. This will give you part of the credit for this assignment, but you will be judged on the portability of your solution, i.e. would it work in other circumstances, like if the quad is rotated or scaled.

Hint 1: We would typically like the vectors from the normal map to have components in the range [-1, 1].
Hint 2: It is probably a good idea to multiply the vector from the texture with a small value before adding it to the geometric normal so the effect can be used subtly.
Hint 3: HLSL and XNA support passing of *tangent data* for each vertex. Look into the `TANGENT0` semantic if you really want to implement it perfectly.

Reference material for the theory:

- Look for “Normal mapping” on Wikipedia

Screenshot description:
If you have done the gamma correction assignment, set gamma back to `1.0f`. Note that lighting should be enabled now for the quad, also with specular reflections. Make two screenshots! One screenshot of a surface or model with normal mapping applied to it, lit by either a *directional light* in direction (-1, -1, -1) or by a *point light* on position (50, 50, 50). The name should be “normalmapping1.png”. Another screenshot of the same model/surface, lit by the same light but with the light's position/direction mirrored in the x-axis (see the reference images below). The name should be “normalmapping2.png”.

Normal mapped surface, light from the right

Normal mapped surface, light from the left