Shader Programming and Graphics Hardware

Marries van de Hoef

Graphics 2012/2013, 4th quarter
Practicals

• The first assignment was about the basics
• What is going on behind the XNA functions?
• The second and third assignment require that knowledge

Could you make the XNA functions yourself?

For the second and especially the third assignment, you need to globally understand what’s going on inside the graphics hardware. You will write shaders, which interact (directly or indirectly) with all important parts of the graphics hardware.
Goals

- Get some intuition about graphics hardware
- Learn the role of shaders
- Shader programming basics

We will look at the entire process of using graphics hardware.

The most important part is how shaders interact in this process.

What are shaders anyway?
Shaders are small applications running on the GPU. They are your most important way of influencing what happens on the GPU, because they give total control over their stage.

We will also take a look at a simple shader to understand how it works.
The CPU communicates with the GPU over the Bus (you learned that at Computerarchitectuur en Netwerken). (The newest CPU’s don’t use the “normal” bus anymore, it’s too slow. Instead, a more direct connection is used.)

Both have their own memory, and cannot access each other’s memory. If the GPU wants to read something from the CPU, it has to be copied to GPU memory first.

The communication is nearly one-way: the CPU transports a lot of data to the GPU, while the other way around is rare.
The CPU can run any type of program. It is designed to be very flexible to deal with every application.

The GPU can only run specific programs which conform to the GPU’s specific data flow. The instructions are specialized for vector calculations. This specialization is required to make the GPU massively parallel. The CPU has around 4 cores, whereas the GPU can have over 2000 cores.

The specialism of the GPU has as consequence that it is useless by itself, it needs a smarter CPU to give him commands.
The first part of the lecture will explain the graphics pipeline, in which all processing is done.

The second part of the lecture will discuss how we can interact with the graphics pipeline to configure and program it to our needs.
Graphics Pipeline
The GPU is structured as a pipeline. Years ago, games used the fixed-function pipeline to render (before 2000). This is barely used anymore because the programmable pipeline gives much more control.

The BasicEffect you used in the first practical assignment worked similar to a fixed-function pipeline: you just set some settings, and it all works!

The first Vertex Transformation stage transforms the vertices from 3D coordinates to 2D screen positions. In the BasicEffect, you had to set the View and Projection matrices to control the transformation.

Now that the vertices are in 2D, the Rasterization stage determines which pixels are occupied by the triangle.

For each pixel, the final color is determined in the Pixel Color Determination stage. In the BasicEffect, you could for example set a Texture to control this stage, and you could specify light sources.

Note that this entire pipeline is executed for each draw call.

A draw call is one iteration through the pipeline, which is initiated through one of the GraphicsDevice draw functions such as DrawIndexedPrimitives. In the draw call, all vertices (specified as arguments of the draw function) are processed using the current settings.
The programmable pipeline replaces some stages with fully programmable shader stages to give more control.

The vertex transformation stage is replaced by the vertex shader, and the pixel color determination stage is replaced by the Pixel Shader. Note that the Pixel Shader is sometimes called the Fragment shader. Fragment shader is the term used in OpenGL, whereas Pixel Shader is used by DirectX.

There is one disadvantage: all the things the replaced stages already did, aren't done anymore! We have to program them manually in the shaders. The shaders have access to the GPU memory, so they can also access matrices and lighting information. But you can also pass different things to your shaders, for example other settings to use in your shader.
The vertex shader is executed for each vertex separately. It cannot access other vertices. The vertex shader is the same for all vertices in one draw call.

Its most important task is to transform the vertex positions from 3D to 2D, just like the “Vertex Transformation” stage did.

A vertex does not only have a position. It can have many more attributes. These can be modified in the vertex shader, and new attributes can be added, or others can be removed. The only required output attribute is the position, because it is required by the rasterizer. All other output attributes can be used later in the pixel shader. Storing something in an output attribute is the only way a vertex shader can output data.

The processing of vertex attributes can be influenced by using shader variables from the GPU memory. For example, you could calculate the lighting in the vertex shader. You would use the normal from the input vertex attributes, and the lighting direction would be stored in the shader variables. The calculated lighting would be stored in a new output attribute. (In practice, lighting is commonly calculated in the pixel shader for more detail.)
The rasterizer executes for each triangle, and it’s main function is to determine which pixels are occupied by that triangle.

This stage is not programmable, however it can be configured slightly using the RasterizerState.

First, the rasterizer checks if the triangle is visible at all. If it’s not visible, the processing of this triangle stops here (the triangle is culled). The rasterizer may also perform backface culling, which removes the triangles at the backside of a model because these triangles are not visible anyway. Then the rasterizer checks which pixels are occupied by this triangle. The pixel shader will be executed for each pixel separately.

The pixel shader will use the vertex attributes outputted from the vertex shader. The attribute values are interpolated over the triangle, depending on the distance to the vertex. You have already seen in the first assignment what happens if color is interpolated over a triangle. That results in a nice gradient, which is actually just the interpolated color. Any other attributes also get interpolated, so be aware of this!
Pixel Shader

• Input: Vertex attributes for this pixel
• Output: 1 Pixel
• Determines the final color of this pixel
• Retrieve a color from a texture
• Calculate lighting
• Normal mapping
• ...

The pixel shader gives the most freedom of the stages. It receives interpolated vertex data from the rasterizer, and only needs to output a color.

Using data from the GPU-memory, we are free to do whatever we want:
- using the texture coordinates we got from the vertices, we can lookup a color from a texture
- we can also calculate the lighting at this point
And a whole bunch of other techniques.
There are two additional stages which concern the input and output of the pipeline.

The input Assembler constructs the vertices and determines the corresponding triangles for rasterization. It also formats the vertex data in such a way that the Vertex Shader can access it.

The output merger is the final stage in the pipeline, and is the only stage which outputs to GPU memory. It does z-buffer testing. This makes sure that pixels closer to the camera overlap pixels further away.

The output merger can also blend the resulting color on top of a previous image. This is for example used when drawing transparent objects.

Finally, the output merger writes to an output image, more specifically a render target. This render target can be the back buffer (display) but the output merger can also write to a texture. That image can be used in a later draw call as input for the pixel shader.
DirectX 10 has introduced new stages in the Graphics Pipeline. XNA uses DirectX 9, so you cannot use this. You don’t need to remember this, it’s just to show some of the cool recent developments.

Remember that the Vertex Shader only works on isolated vertices? The Geometry Shader works on triangles, so actually multiple vertices at the same time. It can also access neighboring triangles. It can also create new triangles, and even store them in GPU memory for later use. This makes new graphics technique possible in real time. The most used feature is the creation of new triangles, which can be used for example for fur generation.
DirectX 11 also introduced new stages in the Graphics Pipeline.

These new stages can subdivide existing triangles into more triangles, called tessellation. This adds a lot dynamic detail to the world. With this technique it is less noticeable that the world actually consists of triangles.
This is an overview of the Ati Radeon HD 7900 series, currently one of the fastest graphics cards.

In the center area, one block with GCN in it is actually a group of 64 processing units. There are over 2000 processing units in this GPU. You can clearly see the massive parallelism here.

Each processing unit can execute either vertex shader, or pixel shader code. In the past, there were separate units for vertex shaders and pixel shaders. These more general unified shaders are better because if the vertex shader is very short and the pixel shader is very long, the processing power can be divided accordingly.

The non-shader stages are not handled by these unified shaders, but by dedicated hardware. For example the Rasterizer and Vertex Assembler are shown on top.

Each GCN block, is one SIMD unit. SIMD means: Single Instruction, Multiple Data. SIMD is the principle used to keep the parallelism manageable. This means that groups of processing units are executing the same code, but on different data. So in practice, this means that the same shader is being executed simultaneously for many different vertices/pixels.
Pipeline example
We’re going to see how this image is constructed in the graphics pipeline.

There will be two draw calls, one for the box and one for the torus (donut).
We start with raw input vertices. These cannot be drawn on the screen because they have no position on the screen yet.

The input assembler stage gives these vertices to the vertex shader for processing.
The vertex shader transformed the 3D positions to 2D positions on the screen. (Note that the displays the lines between the vertices. The actual vertices lie on the intersection points of the lines.)
The rasterizer first tries to remove any triangles which are not on screen (none are removed because all triangles are within the bounds of the screen). Then the rasterizer performs backface culling, so the triangles at the backside of the box are removed.
The rasterizer performs the actual rasterization: For each triangle, it determines which pixels are covered.
The pixel shader is executed for each covered pixel, and it calculates the final color.
The output merger does the depth testing using the z buffer. Each pixel in the z buffer stores the distance to that pixel. If a new pixel has a depth value which is larger than the existing depth value for that pixel, the new is further away than something that was previously rendered so the new color value is not saved.

In this case, the z-buffer was empty so all pixels are stored. The image shows the depth values of the box.
All pixels are stored in the render target. This draw call is finished now.
The next draw call draws the torus.
First the vertices are transformed to the screen position by the vertex shader.
The rasterizer culls the backside of the torus.
The rasterizer determines the covered pixels.
The pixel shader determines the color for each covered pixel.

However, there’s a problem because the torus is behind the box, so a part of it should not be displayed. The depth testing in the output merger is going to take care of that.
For each pixel of the torus, it compares the depth to the distance stored in the z-buffer. The color of the new pixel is only stored if the distance of the new pixel is closer than the existing distance in the z-buffer. This ensures that the nearest pixel remains visible.
Finally the output merger outputs the visible torus pixels to form the final image.
Break.
Instructing the GPU
API’s

Your Awesome Game

XNA

Direct3D (graphics part of DirectX)

Graphics Driver

So, what happens on the CPU-side?

Your Game tells XNA what to do.

XNA uses DirectX to accomplish what you want. XNA is similar to DirectX, but it removes a lot of the hassle that is required to deal with DirectX. And XNA is very nicely object oriented. A disadvantage is that a few advanced features of DirectX are missing in XNA, but it’s not likely that you will need them.

Direct3D is very important here. It is called a Graphics API and it forces all drivers to behave similarly (if they want to support Direct3D). If it wasn’t for graphics API’s, you would have to make a lot of exceptions for specific drivers in your code. (Actually, DirectX is more than just a graphics API. The graphics API part is called Direct3D.) OpenGL is another graphics API. The functionality is sort of the same as DirectX, but it is structured differently. DirectX is slightly object oriented, but OpenGL is not object oriented at all. OpenGL is cross-platform while DirectX is Windows specific.

In the future you might interface with a graphics API directly, without the nice interface of XNA.

And finally the graphics driver which actually sends the data and instructions over to the GPU.
We look at how you can influence each stage in XNA. First let’s look at the non-shader stages.

First the Input Assembler stage.

You have to bind the vertex and index buffer it will use. The VertexDeclaration describes what is present in the vertex buffer. It describes the vertex attributes and their bytelocation. In XNA, the vertex declaration is defined in the Vertex you use to fill the vertex buffer. In the first practical you had to create the VertexPositionColorNormal, in which you also defined a vertex declaration.

The primitive type describes how the vertices are ordered to form primitives (triangles or lines). You have to pass this in the GraphicsDevice draw function (for example the DrawIndexedPrimitives function). Most commonly, the TriangleList is used.
We’ll skip the vertex shader stage for now, so we get to the rasterizer stage.

The rasterizer stage can be configured through the RasterizerState in XNA. There are several interesting settings, but the most important one is the culling setting. You can enable or disable the backface-culling here.

There are several other less important settings, if you’d like you can check the MSDN documentation to find out what they do.
Output Merger Stage

- State
  - DepthStencilState
    - `GraphicsDevice.DepthStencilState = ...`
    - Z-buffer settings
    - Stencil buffer settings
  - BlendState
    - `GraphicsDevice.BlendState = ...`
    - Alpha blending (for transparency)
  - RenderTarget2D
    - `GraphicsDevice.SetRenderTarget(...)`
    - `RenderTarget2D` can later be used as a `Texture2D`

We’ll also skip the Pixel Shader, so we arrive at the Output Merger stage.

The depth testing can be configured through the DepthStencilState of XNA. The most common use is to either enable or disable depth testing.

The blendstate can be configured, which is mainly used to enable or disabled alpha blending. Alpha blending blends a transparent surface on the existing image.

The render target can be changed here. By default it is set to null, which means that it will draw on the back buffer (display). However, you can also specify a `RenderTarget2D`. The `RenderTarget2D` is a subclass of `Texture2D`, so you are basically rendering on a normal texture. This texture can be used as a texture in the pixel shader of a later draw call.
Pixel Shader/Vertex Shader Stage

- **Shader**
- **Shader Variables**
  - `Effect.Parameters[“name”].SetValue(...)`
- **Textures**
  - `Effect.Parameters[“name”].SetValue(...)`
- **Apply()** after you set the values.

Both the Vertex Shader and Pixel Shader stage are set through an Effect.

An Effect can have multiple techniques, which in turn can have multiple passes. A Technique describes a specific way to render. For example, you might way to use slightly different shading for metal surfaces than you would use for concrete surfaces. A Pass lists the shader functions to use. Sometimes it can be useful to split an effect over multiple passes, but this is rarely used in practice. Often an effect has just one technique and one pass.

The shaders are set when you apply the pass.

The shader variables and textures you want to use have to be set through the effect. The “name” is the variable name used in the effect file. The `Apply()` function transfers the shader variables to the GPU, so make sure that you call that function AFTER you set the values for the shader variables.
There is an important distinction between load time and run time. To minimize the traffic between the CPU and the GPU over the slow bus, you should transfer all data once at load time. At run time, you can then simply select the data you want to use.

This is the reason that the DrawUserPrimitives function is slow. It transfers all vertex data to the GPU each frame at runtime. It is much more efficient to store the vertices in a vertex buffer at load time, activate the vertex buffer by setting it at run time.

This load time/run time distinction is also important for State objects. Do not create them each frame.

Shader variables change at run time, so they have to be sent to the GPU each frame. In general, this does not have much impact because the size of this data is small.
**GPU to CPU**

- Data always goes from CPU to GPU
- GPU to CPU is uncommon, but possible (and slow)
  - Texture2D.SaveAsPng(…)
  - Texture2D.GetData(…)

All communication we discussed so far describes data sent from the CPU to the GPU. It is uncommon for data to go the other way, but it is possible (and slow).

The only output of the GPU, the final image, does not need to be transferred back to the CPU because it goes directly to the display.
Shader programming
Shaders for XNA are written in HLSL. That is the same language which is used for shader programming in DirectX.

HLSL is similar to C#, so you have the same syntax for assigning a value to a variable. However, there is a lot missing which is not relevant for shader programming. There are no classes, just functions and structs. Some other things are added, for example easy to access functions for a lot of calculations such as sinus, absolute and round. For an overview of the exact syntax and all available functions, check the MSDN documentation pages referenced in the second practical.

Writing a shader is totally different from writing normal code. There is no autocomplete, and no notification if your code is incorrect. So make sure you type the variable names correctly! There is no real debugging support. You get errors when there are problems compiling your code, but if the code compiles correctly but produces the wrong results, it is not possible to add a breakpoint and check what the shader is doing. So make sure your code is correct, or else you can spend a lot of time debugging!

A good approach is to write a shader incrementally. Do not write the entire shader first and only test the shader when it’s done, then you will have a hard time debugging. It’s better to test the shader with every change you make. So start out simple, and test with every change you make. When a change does not give the expected result, the bug has to do something with the last change you made.

- HLSL – High Level Shader Language
- Similar syntax to C#
  - Simplified
  - Specialized syntax
  - Read MSDN documentation
- Different style of writing code
  - No autocomplete
  - Hard to debug
  - Write incrementally
Global effect layout

• Global shader variables
• Textures and samplers

• Vertex attribute structs
• Vertex shader
• Pixel shader

• Techniques

XNA shaders are written in effect (fx) files. An effect file has globally this layout.

At the top there are shader variables which contain the data you set in XNA (as explained in slide 38). The shader variables cannot be changed in the middle of one draw call. They are read-only for the shader. The textures and samplers are also defined at the top. A sampler describes how texture data should be retrieved from a shader.

The vertex attributes structs describe the vertex attributes which enter and leave the vertex shader.

Then, the actual vertex/pixel shader code.

Finally at the bottom are the techniques, as discussed in slide 38.
This is the shader used to render the torus in the graphics pipeline example. This slide contains the first part of the shader, which focuses on the vertex shader. The next slide focuses on the pixel shader.

At the top, the shader variables are defined. They are used in the vertex shader. The vertex shader has as input parameter the VertexShaderInput struct, which contains the vertex attributes it wants to receive. The vertex shader outputs the VertexShaderOutput struct. The vertex attributes in the structs have a special syntax. POSITION0 and TEXCOORD0 are called semantics and they describe what type of data this field will contain. This is important for the transportation between stages. The input assembler uses the semantics to correctly match vertex buffer data with the vertex shader input. The rasterizer uses the POSITION0 semantic to find the 2D position it has to use for the rasterization.

First, the VertexShaderOutput struct is initialized by clearing it with zero values. Unlike in C#, variables are not automatically set to their default values in HLSL. Then the 3D Position of the vertex is multiplied by the World, View and Projection matrices to transform it to a 2D position. Note that multiplying a matrix is done with the mul function. The texture coordinate attribute of the output struct is copied without processing from the input struct. Finally, the VertexShaderOutput is returned.
This slide contains the pixel shader part of the effect.

The pixel shader has the VertexShaderOutput struct as input. The values contained in this struct are interpolated by the rasterizer as explained in slide 11. The pixel shader returns one float4, which has the COLOR0 semantic. This is expected, because the task of the pixel shader is to output the color for this pixel.

The pixel shader uses the tex2D function to lookup data from a texture at the input.TexCoords location. The sampler describes how the texture should be sampled. The Sampler contains a reference to the texture it has to sample, and several settings. The filter settings describe the behavior when the texture is stretched or compressed, and the Address settings describe the behavior when it tries to sample outside of the range of the texture. For more information, check the MSDN documentation.

Finally, the effect contains the technique used. There is one technique which contains one pass. The pass describes which functions should be used as Vertex Shader and Pixel Shader. vs_2_0 and ps_2_0 defines the shader model used to compile the shaders. A lower shader model allows the shader to be used on older hardware, while a higher shader model enables more features. You will probably have to increase the shader model to 3.0 for the third practical.
This is what the final result looks like after the shaders have been executed.
Changing the pixel shader

• Freedom to change the code to whatever you want

```c
float4 SimplePixelShader(VertexShaderOutput input) : COLOR0
{
    return tex2D(TextureSampler, input.TexCoords);
}
```

```c
float4 SimplePixelShader(VertexShaderOutput input) : COLOR0
{
    return float4(input.TexCoords, 0, 1);
}
```

We can adjust the code to whatever we want. For example, we can output the texture coordinates as colors, just because we can.

Note how we convert the TexCoords (which is a float2 type) to a float4 type.
The result will change to this.
We can just do one minus the result of the \texttt{tex2D} function. Since the \texttt{tex2D} function returns a \texttt{float4}, that means that the 1 in this case is also interpreted as a \texttt{float4} (where all components are 1).
The result will change to this.
Changing the pixel shader

- Swizzling the color
  - Swap the red and green channels

```c
float4 SimplePixelShader(VertexShaderOutput input) : COLOR0
{
    return tex2D(TextureSampler, input.TexCoords);
}
```

```c
float4 SimplePixelShader(VertexShaderOutput input) : COLOR0
{
    float4 color = tex2D(TextureSampler, input.TexCoords);
    float4 swizzledColor = color.grba;
    return swizzledColor;
}
```

We can also rearrange channels easily using the swizzle syntax. The rgba channels are rearranged so that the red and green channels are swapped. This syntax is very powerful and flexible, you can try things out for yourself. This is also possible with xyzw instead of rgba.
The result will change to this.
There are several things you have to keep in mind when you are programming shaders.

The GPU is not optimized for if statements. However, very short if statements are okay.

Write your shaders incrementally, to make debugging easier. When you do have to debug, you can learn more about what your shader is doing by outputting intermediate values as the color. This way, you can find out where the error is.

The compiler in the graphics driver optimizes your code. If a calculated result is never used, the calculation is removed. This can optimize more than you would expect. It occurred a few times to me that I thought I did a great optimization, but it turned out that the compiler had already optimized that! Nevertheless, should not try to rely on the compiler for optimization.
So in short what the Graphics Pipeline does:
- The input assembler fetches data from the vertex buffer and feeds it to the vertex shader.
- The vertex shader transforms the vertex position from 3D to 2D.
- The rasterizer determines the pixels covered by each triangle.
- The pixel shader calculates the color for each pixel.
- The output merger outputs the data.

You can influence the pipeline through several State objects and through Effect files.