Shader Programming and Graphics Hardware

Marries van de Hoef

Graphics 2011/2012, 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
Goals

• Get some intuition about graphics hardware

• Learn the role of shaders

• Shader programming basics
High Level Overview

CPU

GPU

Bus
# High Level Overview

<table>
<thead>
<tr>
<th><strong>CPU</strong></th>
<th><strong>GPU</strong></th>
</tr>
</thead>
</table>
| • Very general  
  - Suited to run normal application code. | • Highly specialized  
  - Data flow  
  - Vector calculations |
| • Instructs GPU | • Massively parallel |
|                 | • Waits for CPU commands |
Topics

1. Graphics Pipeline
2. Pipeline example
3. Instructing the GPU
4. Shader programming
Graphics Pipeline
Fixed-function pipeline

- Vertices
  - Vertex Transformation
  - Rasterization
  - Pixel Color Determination

Transformations matrices (World, View, Projection)
(Lighting information)

Convert each triangle to pixels:
Textures / Color
Lighting information

Pixel color
Programmable pipeline

- Vertex Transformation stage replaced by Vertex Shader
- Pixel Color Determination replaced by Pixel Shader
- Access to GPU-memory

Diagram:
- Vertices
  - Vertex Shader
  - Rasterization
  - Pixel Shader
  - GPU Memory
  - Pixel color
**Vertex Shader**

- **Input:** 1 Vertex
- **Output:** 1 Vertex

- Output position must be transformed to 2D

- Modify/add vertex attributes
  - Normal
  - Color
  - Texture coordinates
  - Lighting
  - ... anything you define
**Rasterizer**

- Input: 3 Vertex
- Output: A lot of pixels

- This stage is not programmable

1. Culling
2. Rasterize
3. Each pixel receives all vertex attributes
   - Linearly interpolated
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
- ...
Input Assembler + Output Merger

- **Input Assembler**
  - Before Vertex Shader
  - Assembles data:
    - Vertex Buffer
    - Index Buffer
    - PrimitiveType

- **Output Merger**
  - After Pixel Shader
  - Z-buffer testing
  - Blending
  - Write to render target
DirectX 10

- Vertex Shader
- Geometry Shader
- Rasterization
- Pixel Shader
- Input Assembler
- Vertex Shader
- Rasterization
- Pixel Shader
- Output Merger
- GPU Memory
DirectX 11

- Vertex Shader
- Hull Shader
- Tessellator
- Domain shader
- Geometry Shader
- Input Assembler
- Vertex Shader
- Rasterization
- Pixel Shader
- Output Merger

GPU Memory
Architecture

- Unified shaders
- Single Instruction, Multiple Data
Pipeline example
Box

• Input vertices
  – new Vector3(1.0f, 1.0f, 1.0f)
  – new Vector3(1.0f, 1.0f, -1.0f)
  – new Vector3(1.0f, -1.0f, 1.0f)
  – new Vector3(-1.0f, 1.0f, -1.0f)
  – new Vector3(-1.0f, -1.0f, 1.0f)
  – new Vector3(-1.0f, -1.0f, -1.0f)
Instructing the GPU
API’s

Your Awesome Game

XNA

DirectX

Graphics Driver
Input Assembler Stage

- **Data**
  - Vertex Buffer
    - `GraphicsDevice.SetVertexBuffer(...)`
  - Index Buffer
    - `GraphicsDevice.Indices = ...`

- **State**
  - `VertexDeclaration`
    - Implicitly activated in VertexBuffer
  - `PrimitiveType`
    - Selected in Draw function
Rasterizer Stage

- State
  - RasterizerState
    - GraphicsDevice.RasterizerState = ...
    - Backface culling
    - Wireframe
    - (MSAA, Scissor test, depth bias)
Output Merger Stage

- State
  - DepthStencilState
    - GraphicsDevice.DepthStencilState = ...
    - Z-buffer settings
    - Stencil buffer settings
  - BlendState
    - GraphicsDevice.BlendState = ...
    - Alpha blending (for transparency)
- RenderTarget2D
  - GraphicsDevice.SetRenderTarget(...)
  - Renders to Texture2D
Pixel Shader/Vertex Shader Stage

- Shader

- Shader Variables
  - `Effect.Parameters["name"]`.SetValue(...)

- Textures
  - `Effect.Parameters["name"]`.SetValue(...)

- Apply() after you set the values.
Load time / run time

• At load time
  – Create and copy Data/Effects/State to GPU memory

• At run time
  – Select active state
  – Copy shader variables to GPU memory

• Never create the same thing each frame!
GPU to CPU

• Data always goes from CPU to GPU

• GPU to CPU is uncommon, but possible (and slow)
  – Texture2D.SaveAsPng(…)
  – Texture2D.GetData(…)

Shader programming
Shader code

- 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
float4x4 View, Projection, World;

struct VertexShaderInput
{
    float4 Position3D : POSITION0;
    float2 TexCoords : TEXCOORD0;
};

struct VertexShaderOutput
{
    float4 Position2D : POSITION0;
    float2 TexCoords : TEXCOORD0;
};

VertexShaderOutput SimpleVertexShader(VertexShaderInput input)
{
    VertexShaderOutput output = (VertexShaderOutput)0;

    float4 worldPosition = mul(input.Position3D, World);
    float4 viewPosition = mul(worldPosition, View);
    output.Position2D = mul(viewPosition, Projection);

    output.TexCoords = input.TexCoords;

    return output;
}
Texture2D BrickTexture;
SamplerState TextureSampler = sampler_state {
    Texture = <BrickTexture>;
    MipFilter = Point;
    MinFilter = Linear;
    MagFilter = Linear;
    AddressU = Clamp;
    AddressV = Clamp;
};

struct VertexShaderOutput {
    float4 Position2D : POSITION0;
    float2 TexCoords : TEXCOORD0;
};

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

technique Simple {
    pass Pass0 {
        VertexShader = compile vs_2_0 SimpleVertexShader();
        PixelShader   = compile ps_2_0 SimplePixelShader();
    }
}
Keep in mind

- Avoid dynamic branching
  - Short is ok: if(a>b) c=0 else c=1

- Think before the writing
  - Write incrementally
  - For debugging: if(a<0) return float4(1,0,0,1);

- Aggressive compiler
  - Might optimize more than you think
Summary

• Graphics Pipeline
  – Input Assembler
  – Vertex Shader
  – Rasterizer
  – Pixel Shader
  – Output Merger

• Control the pipeline through State and Effects