



Shader Programming and Graphics Hardware

Marries van de Hoef



Universiteit Utrecht

Practicals

- The first assignment was about the basics
- What is going on behind the XNA functions?
- The second assignment requires that knowledge



Goals

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



High Level Overview

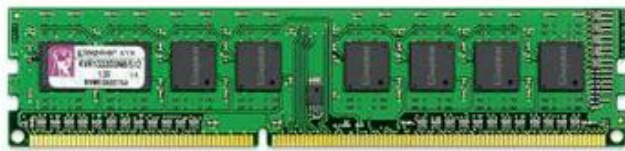
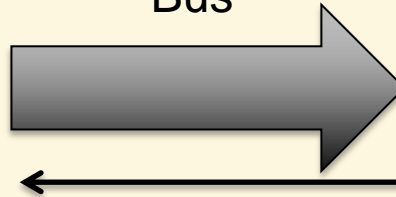
CPU



GPU



Bus



High Level Overview

CPU

- Very general
 - Suited to run normal application code.
- Instructs GPU

GPU

- Highly specialized
 - Data flow
 - Vector calculations
- Massively parallel
- Waits for CPU commands



Topics

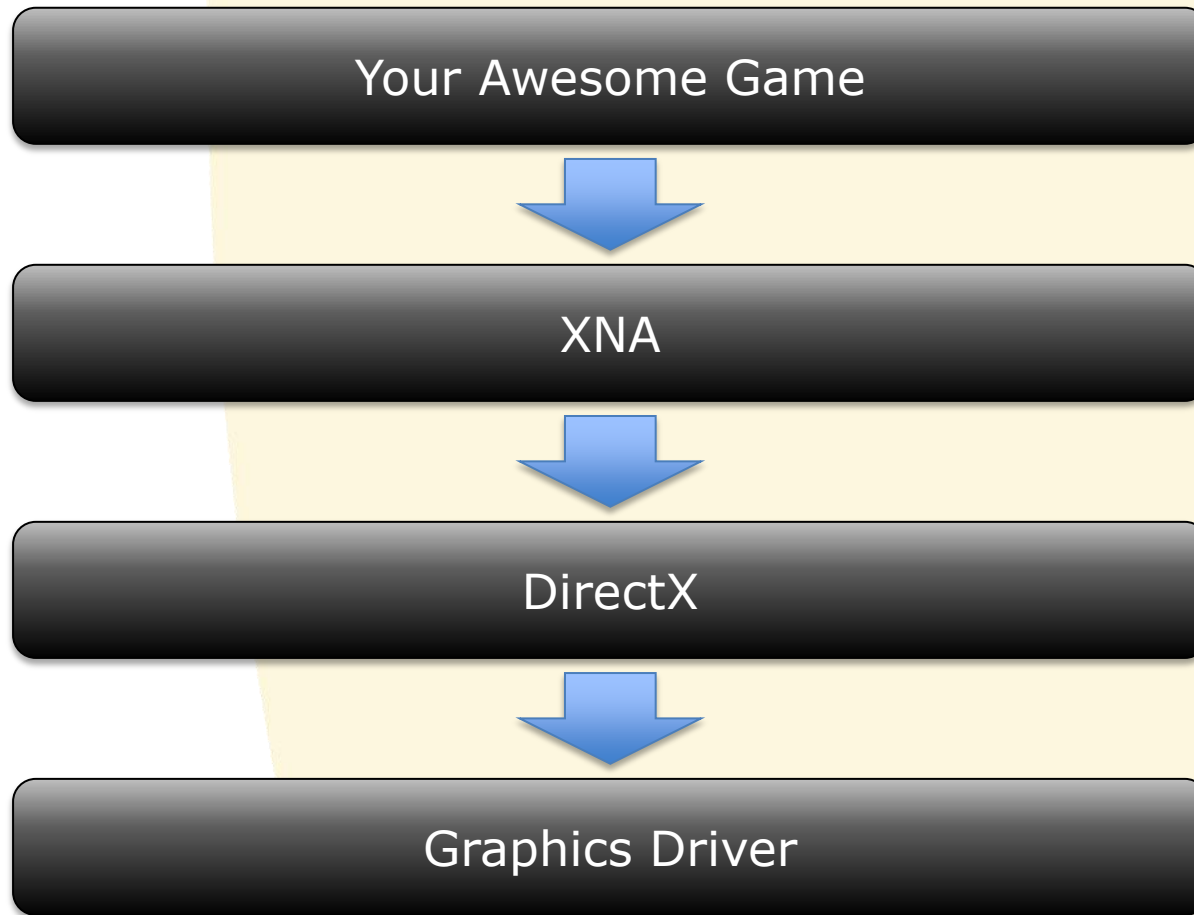
- 1. CPU side**
- 2. GPU side**
- 3. GPU Hardware**
- 4. CPU-GPU Communication**
- 5. Shader Programming**



CPU side



CPU side



GPU side

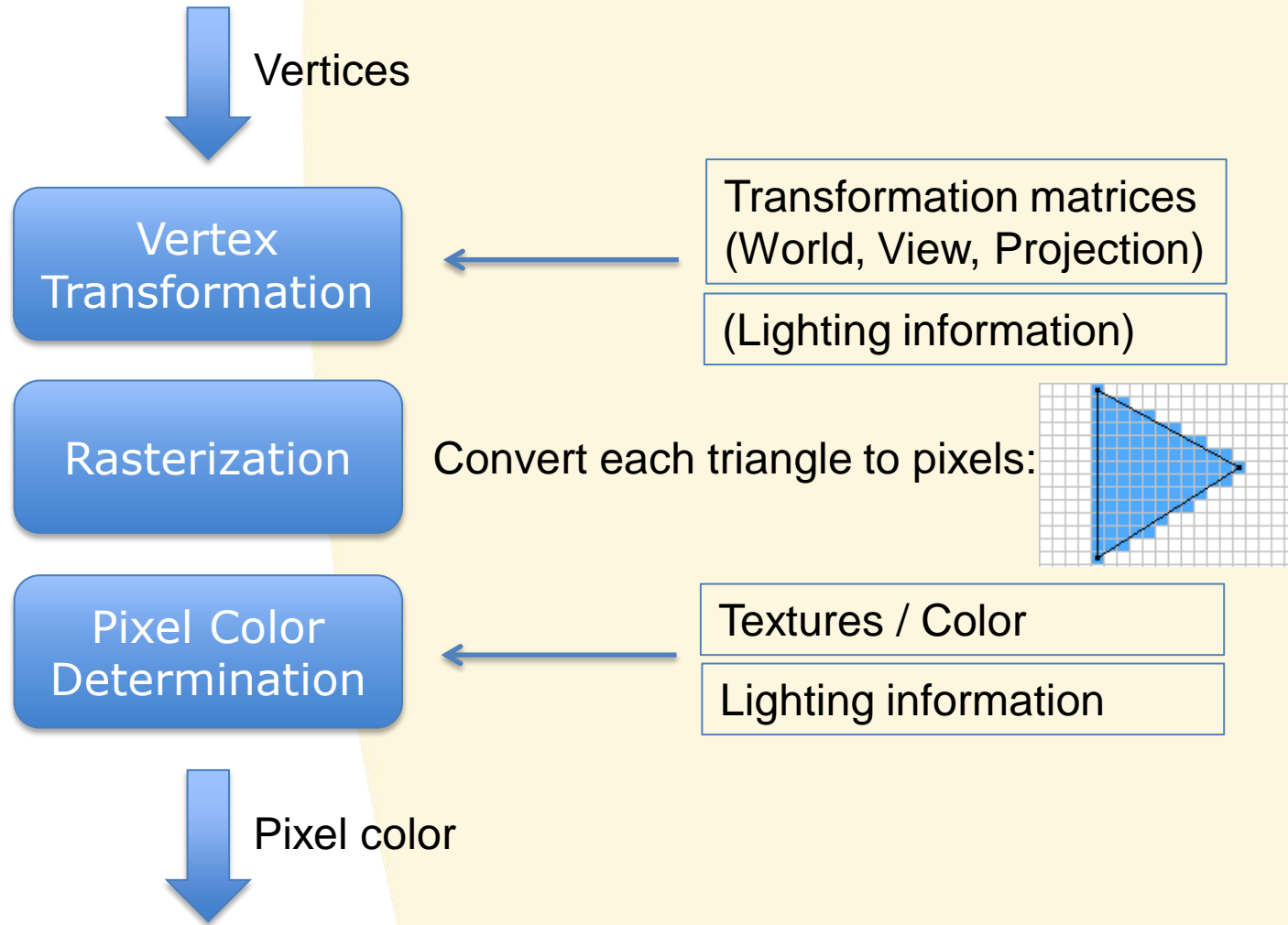


GPU side

- Graphics Pipeline
- Fixed-function pipeline vs. Programmable pipeline

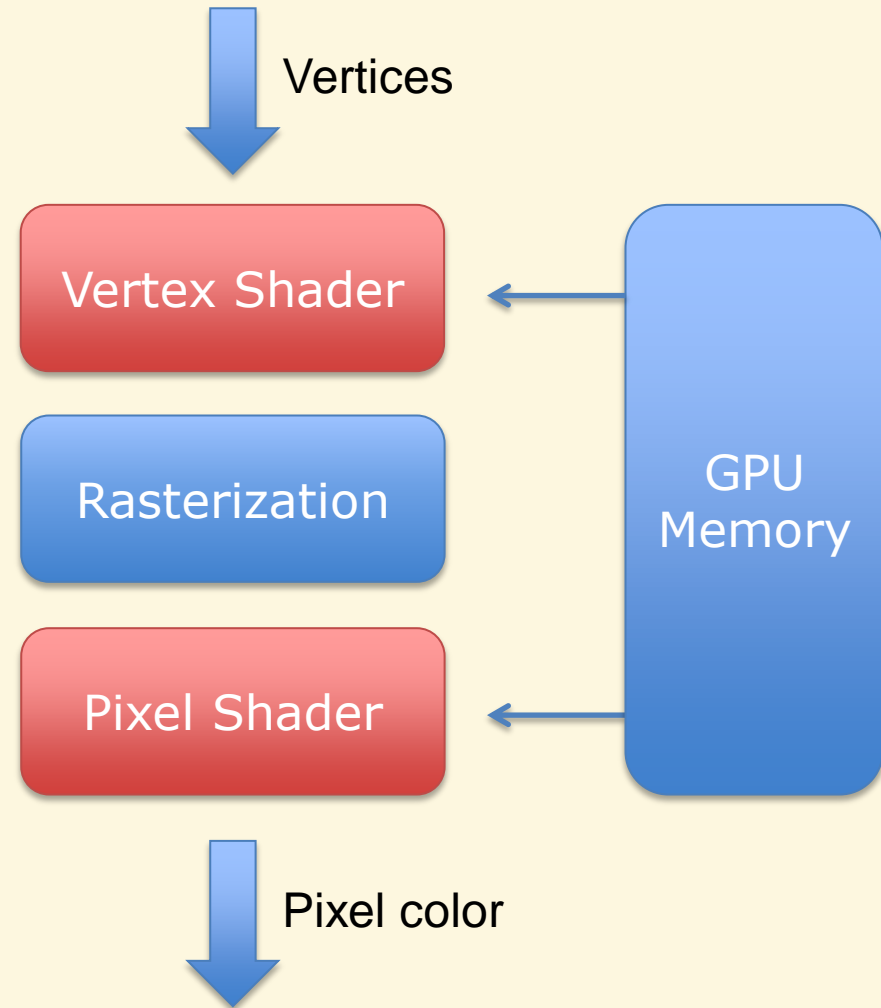


Fixed-function pipeline



Programmable pipeline

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



Vertex Shader

Vertex Shader

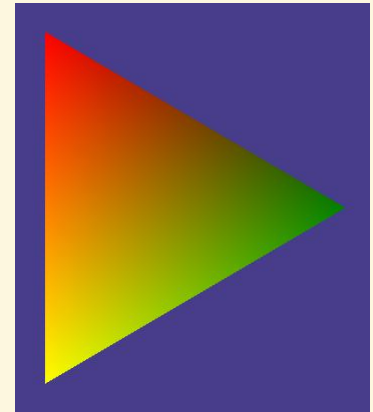
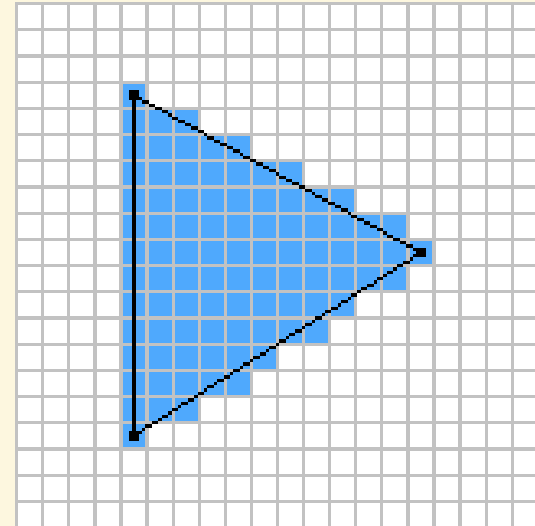
- Input: 1 Vertex
- Output: 1 Vertex
- Output position must be transformed to 2D
- Modify/add vertex attributes
 - Normal
 - Color
 - Texture coordinates
 - Lighting
 - ... anything you define



Rasterizer

Rasterization

- Input: 3 Vertex
- Output: A lot of pixels
- This stage cannot be modified
 1. Clipping
 2. Rasterize
 3. Each pixel receives all vertex attributes
 - Linearly interpolated



Pixel Shader

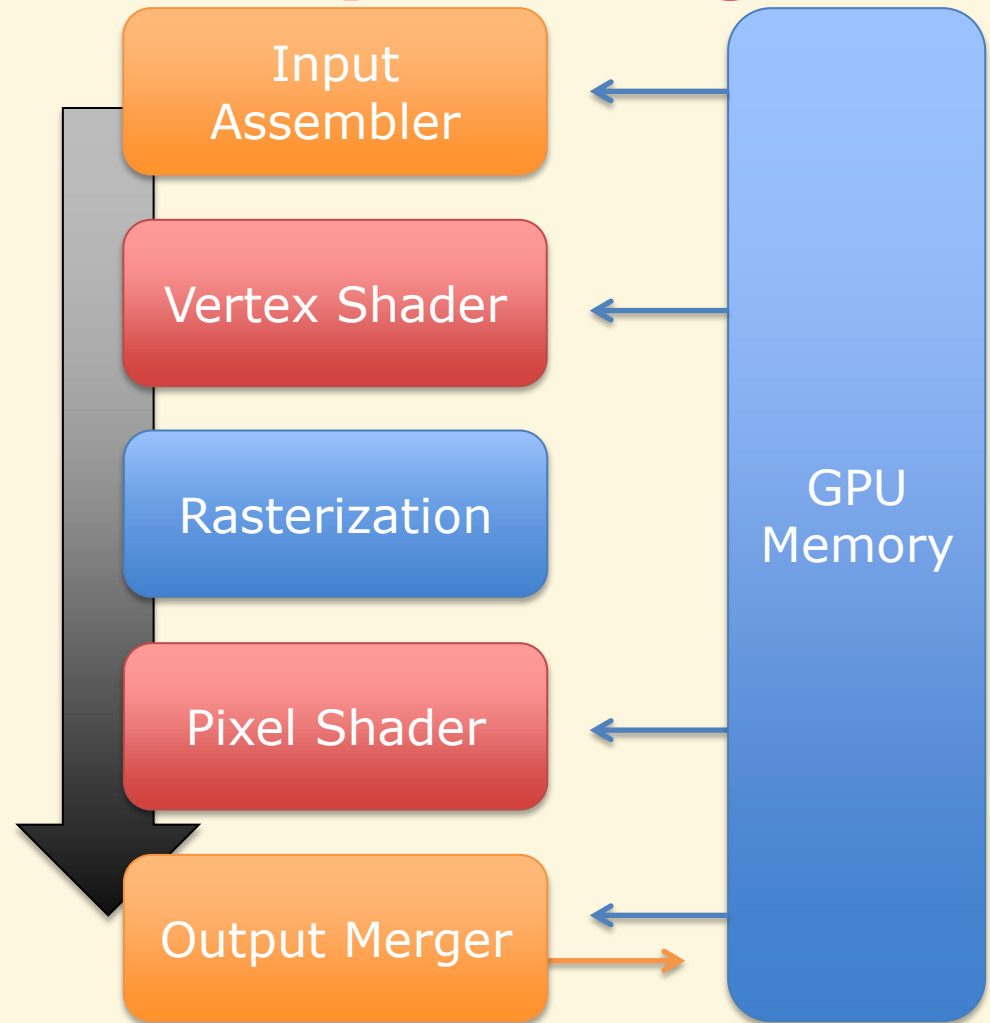
Pixel Shader

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

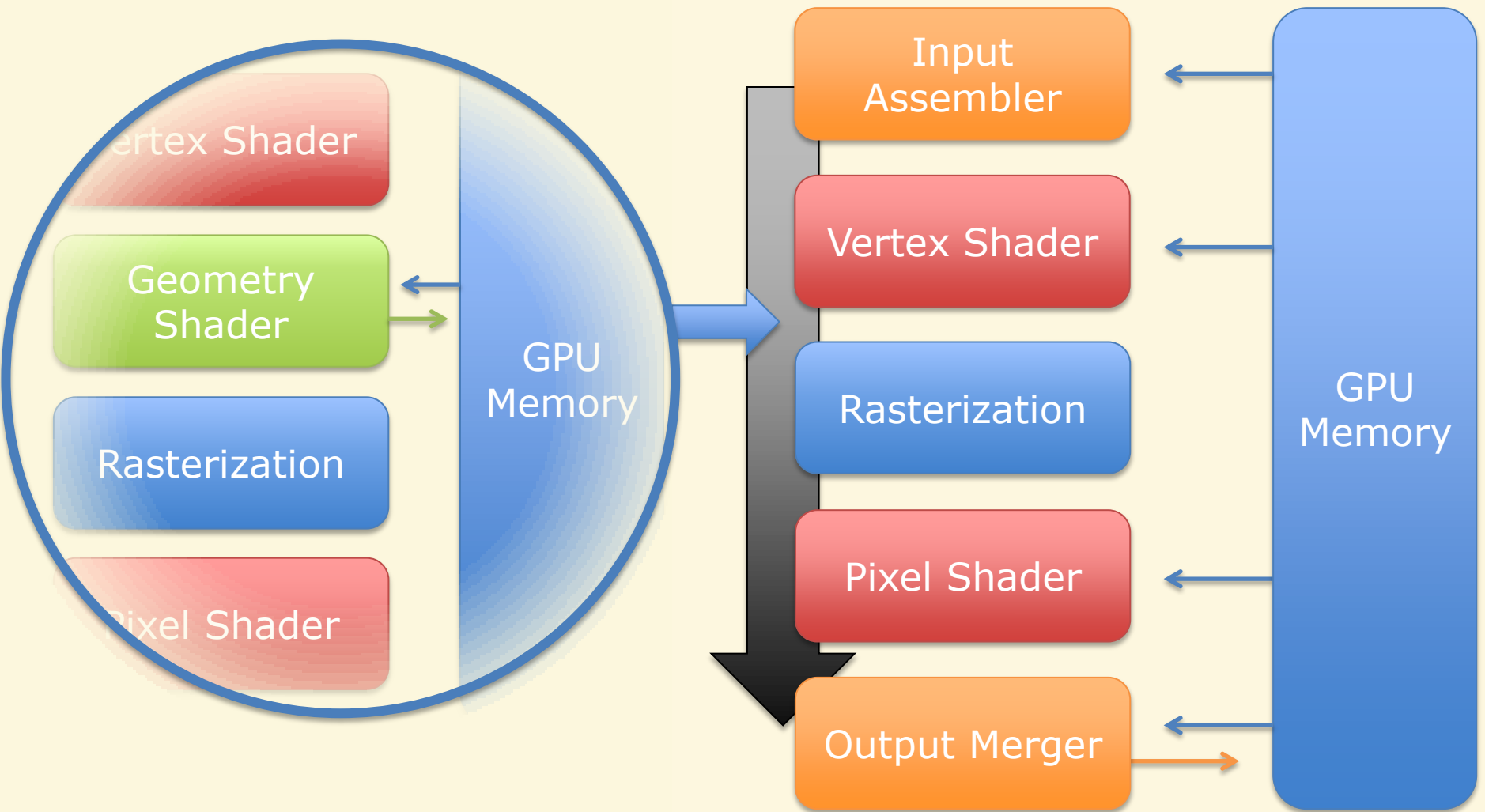


Input Assembler + Output Merger

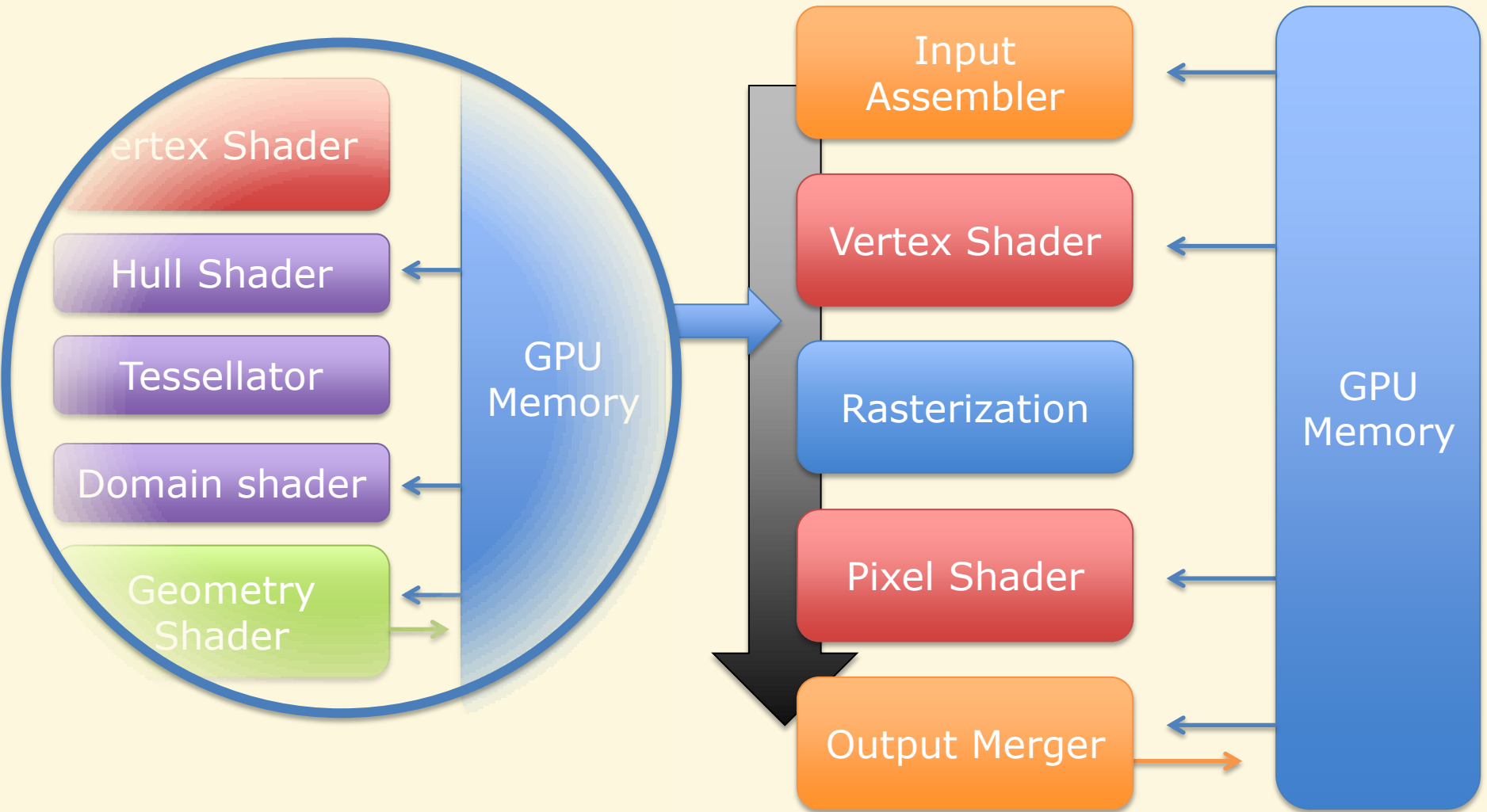
- Input Assembler
 - Before Vertex Shader
 - Assembles data:
 - Vertex Buffer
 - Index Buffer
 - PrimitiveType
- Output Merger
 - After Pixel Shader
 - Z-buffer testing
 - Blending
 - Write to render target



DirectX 10



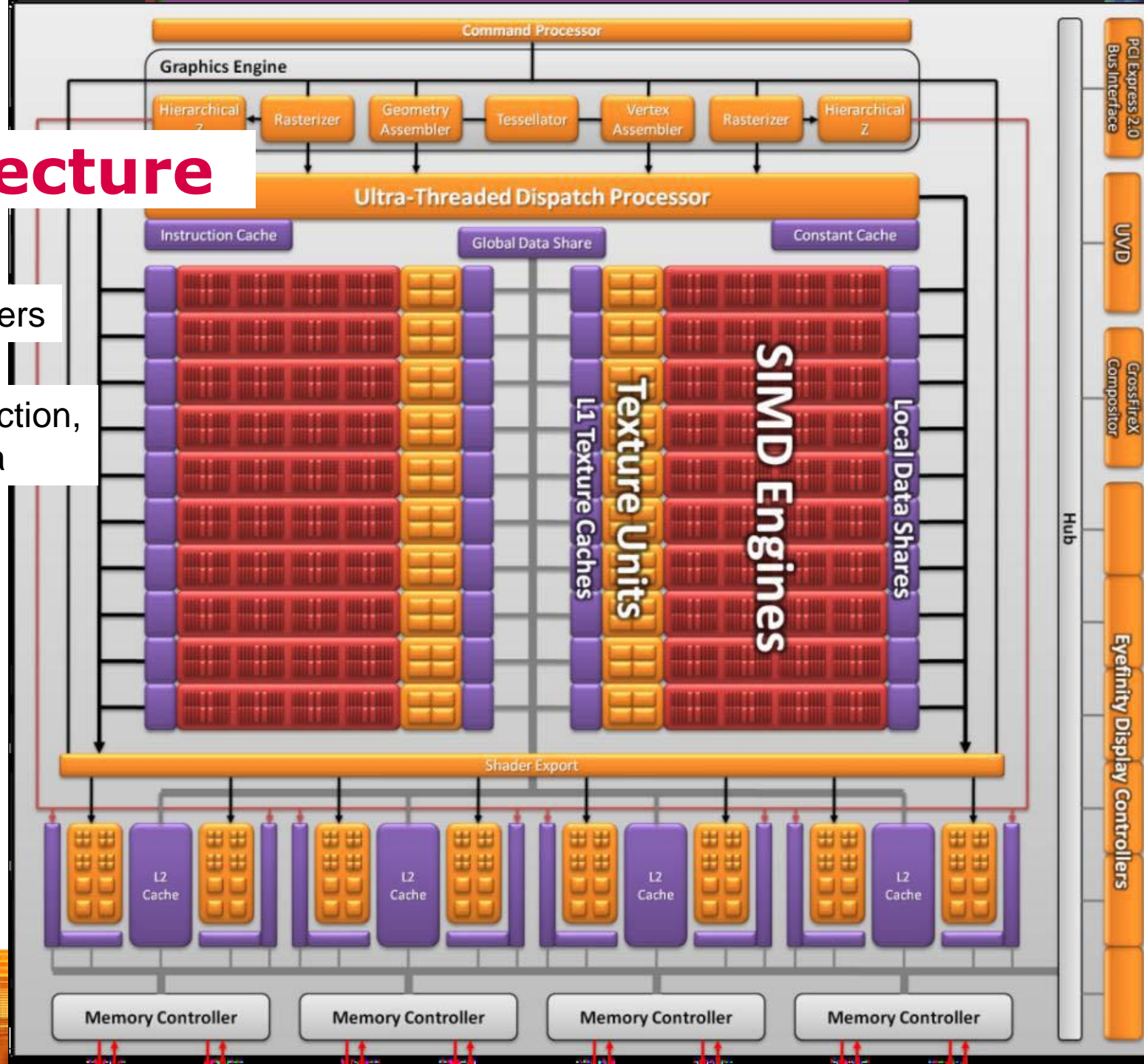
DirectX 11



GPU hardware



Architecture



- Unified shaders
- Single Instruction, Multiple Data



CPU-GPU Communication



Communication

- Data
 - Vertex Buffer
 - Index Buffer
 - Textures
- Effects
 - Shaders
 - Shader variables
- State
 - Settings for various stages



Communication

- At load time
 - Create Data/Effects/State in GPU memory
- At run time
 - Select active state
 - Copy shader variables to GPU memory
- Never ever ever recreate the same thing each frame!



Data

- Vertex Buffer
 - Load: `VertexBuffer.SetData(...)`
 - Activate: `GraphicsDevice.SetVertexBuffer(...)`
- Index Buffer
 - Load: `IndexBuffer.SetData(...)`
 - Activate: `GraphicsDevice.Indices = ...`
- Textures
 - Load: `Content.Load<Texture2D>(...)`
 - Activate: Same as shader variables



Effects

- Shader
 - Load: `Content.Load<Effect>(…)`
 - Activate: `Effect.CurrentTechnique.Pass[0].Apply()`
- Shader Variables
 - Copy/Activate: `Effect.Parameters[“name”].SetValue(…)`
- `EffectParameter name = Effect.Parameters[“name”];`
- `name.SetValue(…);`



State

- Input Assembler
 - VertexDeclaration
 - Implicitly activated in VertexBuffer
 - PrimitiveType
 - Selected in Draw function
- Rasterizer
 - RasterizerState
 - GraphicsDevice.RasterizerState = ...
 - Wireframe
 - Backface culling



State

- Output Merger
 - DepthStencilState
 - GraphicsDevice.DepthStencilState = ...
 - Z-buffer settings
 - Stencil buffer settings
 - BlendState
 - GraphicsDevice.BlendState = ...
 - Alpha blending (for transparency)
 - RenderTarget2D
 - GraphicsDevice.SetRenderTarget(...)
 - Renders to Texture2D



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
- Different style of writing code
 - No autocomplete
 - Hard to debug
 - Write incrementally



```

// TODO: add effect parameters here.
float4 shaderVariable;

struct VSInput
{
    float4 Position : POSITION;
    float2 TextureCoordinates : TEXCOORD0;
};

struct VSOutput
{
    float4 Position : POSITION;
    float2 TextureCoordinates : TEXCOORD0;
};

VSOutput VS(VSInput input)
{
    VSOutput output;

    output.Position = input.Position;
    output.TextureCoordinates =
        input.TextureCoordinates;

    //TODO: add your vertex shader code here.

    return output;
}

```

```

float4 PS(VSOutput input) : COLOR0
{
    // TODO: add your pixel shader code here.

    return float4(1, 0, 0, 1);
}

technique Technique1
{
    pass Pass1
    {
        // TODO: set renderstates here.

        VertexShader = compile vs_2_0 VS();
        PixelShader = compile ps_2_0 PS();
    }
}

```



- **Scalar Types:** `bool int float half`
- **Vector Types:** `float4 int1 bool3 half2`
- **Matrix Types:** `float4x4 bool3x2 int1x4 half3x3`

- **Swizzle**
 - `float3 a = b.xyz; a = float3(b.x, b.y, b.z);`
 - `float2 c = b.zx; c = float2(b.z, b.x);`
 - `float4 d = b.wwyx;`
 - `float4 e = b.rgba;`

- **Semantics**
 - POSITION
 - NORMAL
 - COLOR0
 - TEXCOORD0, TEXCOORD1, TEXCOORD2...



Functions

- Normal operators

- + * / += *=
- == >

- Built-in Functions

- dot cross mul
- abs normalize
- min max clamp saturate
- cos sin tan
- floor ceil round

- Define your own functions



Branching

- Looping

- `for(uint i = 0; i<4; i++) {...}`

- `while(a>b) {...}`

- If/else

- `if(a>b) {...} else {...}`

- Dynamic branching

- Drop pixel

- `discard;`



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



How to start

- Read assignment 2
 - Provides the basic framework
- Start playing around
 - Change some stuff
 - Have fun

