Shader Programming and Graphics Hardware

by Paul Scharf, original slides by Marries van de Hoef

Some Questions

Who has

- already finished P1?
- looked at/started on P2?

Who knows

- what a shader (program) is?
- how to write a shader program?

Practicals

- **The first assignment was about the basics**

- **What is going on behind the XNA functions?**
- **How does the graphics hardware work?**
  - The second and third assignment require that knowledge

Goals

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

What is a shader?

High Level Overview

CPU

Bus

GPU

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

Architecture

ATI Radeon HD 7900 series

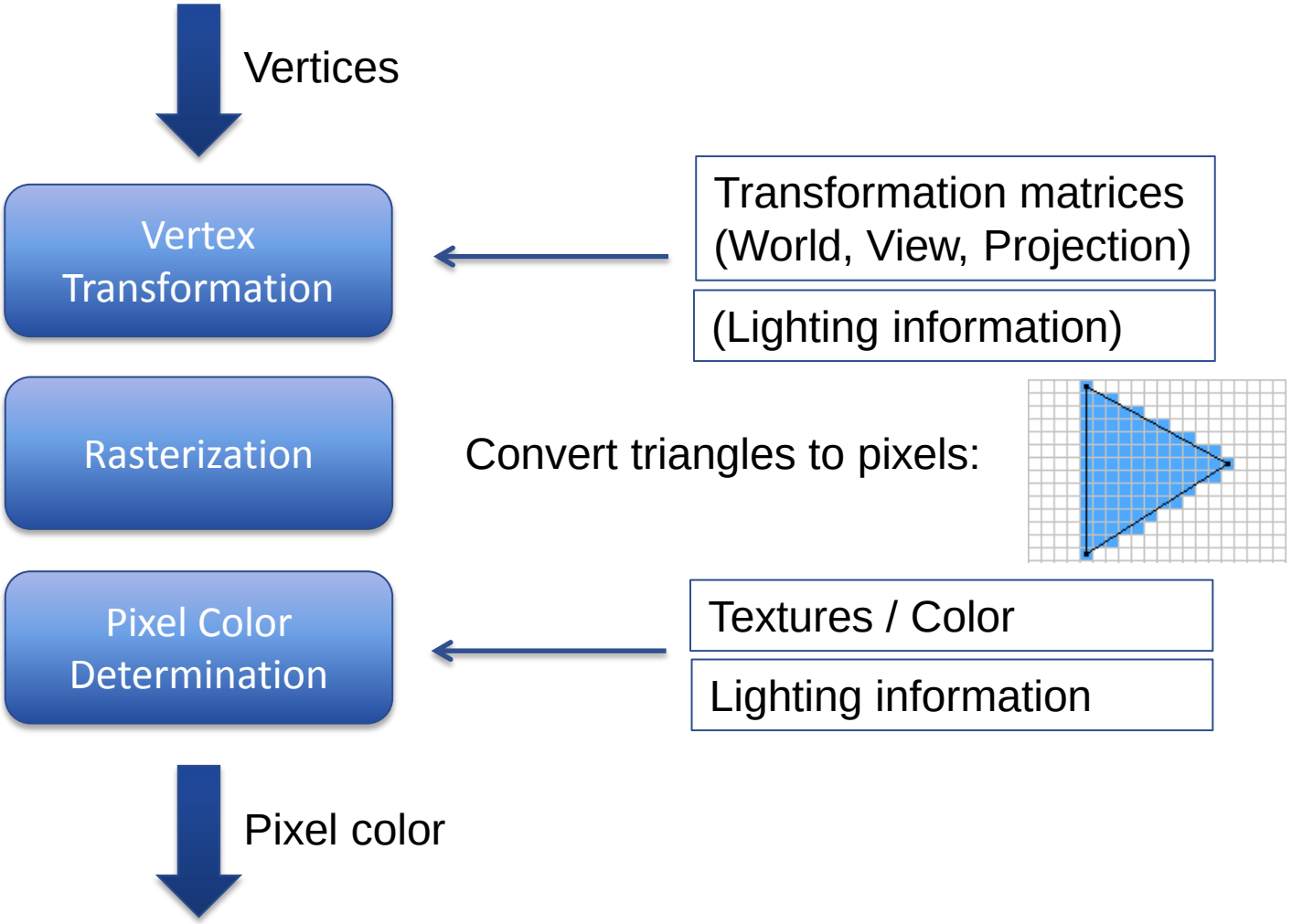
Each ‘GCN’ block: *SIMD unit* (Single Instruction, Multiple Data)

Topics

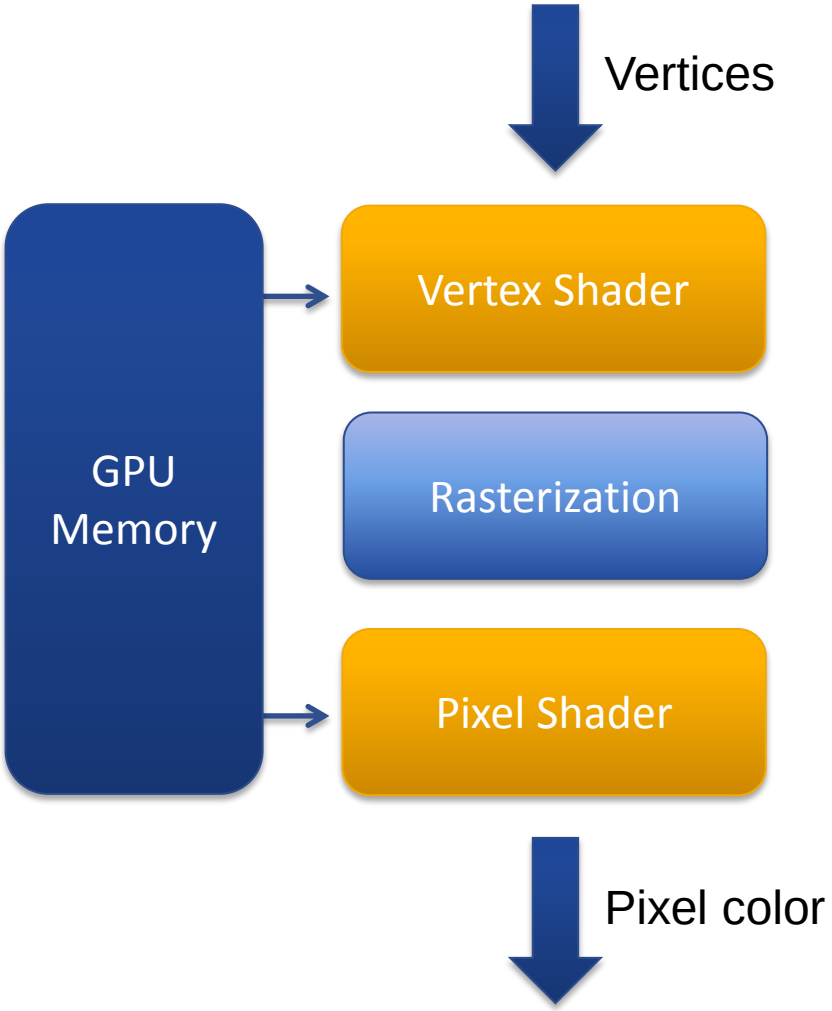
1. **Graphics Pipeline**
2. **Pipeline example**
3. **Instructing the GPU**
4. **Shader programming**

1. Graphics Pipeline

Fixed-function pipeline



Programmable pipeline



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

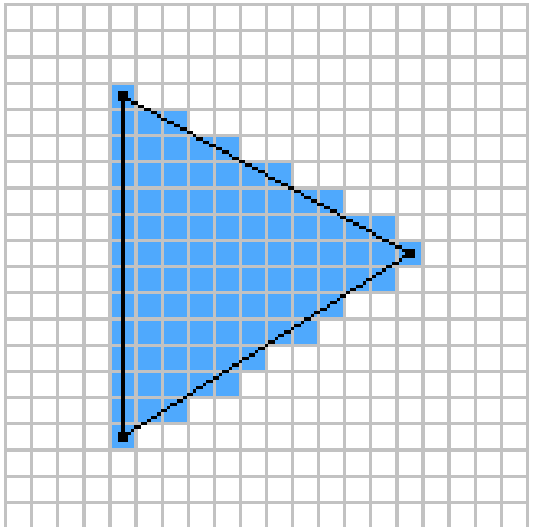
Vertex Shader

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

- **Vertex is transformed to *screen-space* (2D)**
- **Modify/Add your own vertex attributes**
  - Normal
  - Color
  - Texture coordinates
  - Lighting information

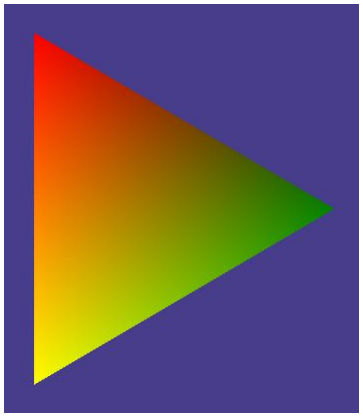
    …

Rasterizer



- **Input: 3 Vertices**
- **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: 1 Pixel (interpolated attributes)**
- **Output: 1 Pixel color**

- **Determines the final color of this pixel**
  - Retrieve a color from a texture
  - Calculate lighting
  - Normal mapping

    …

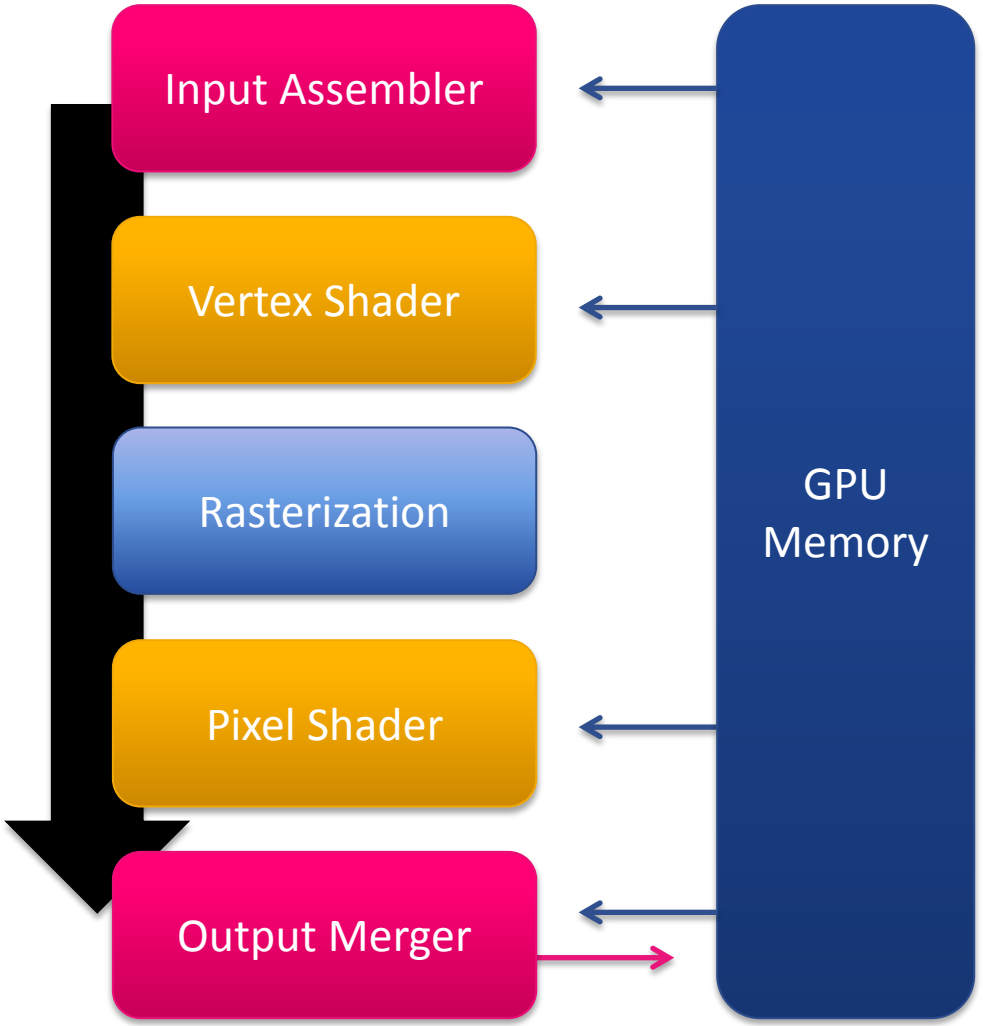
Input Assembler + Output Merger

Input Assembler

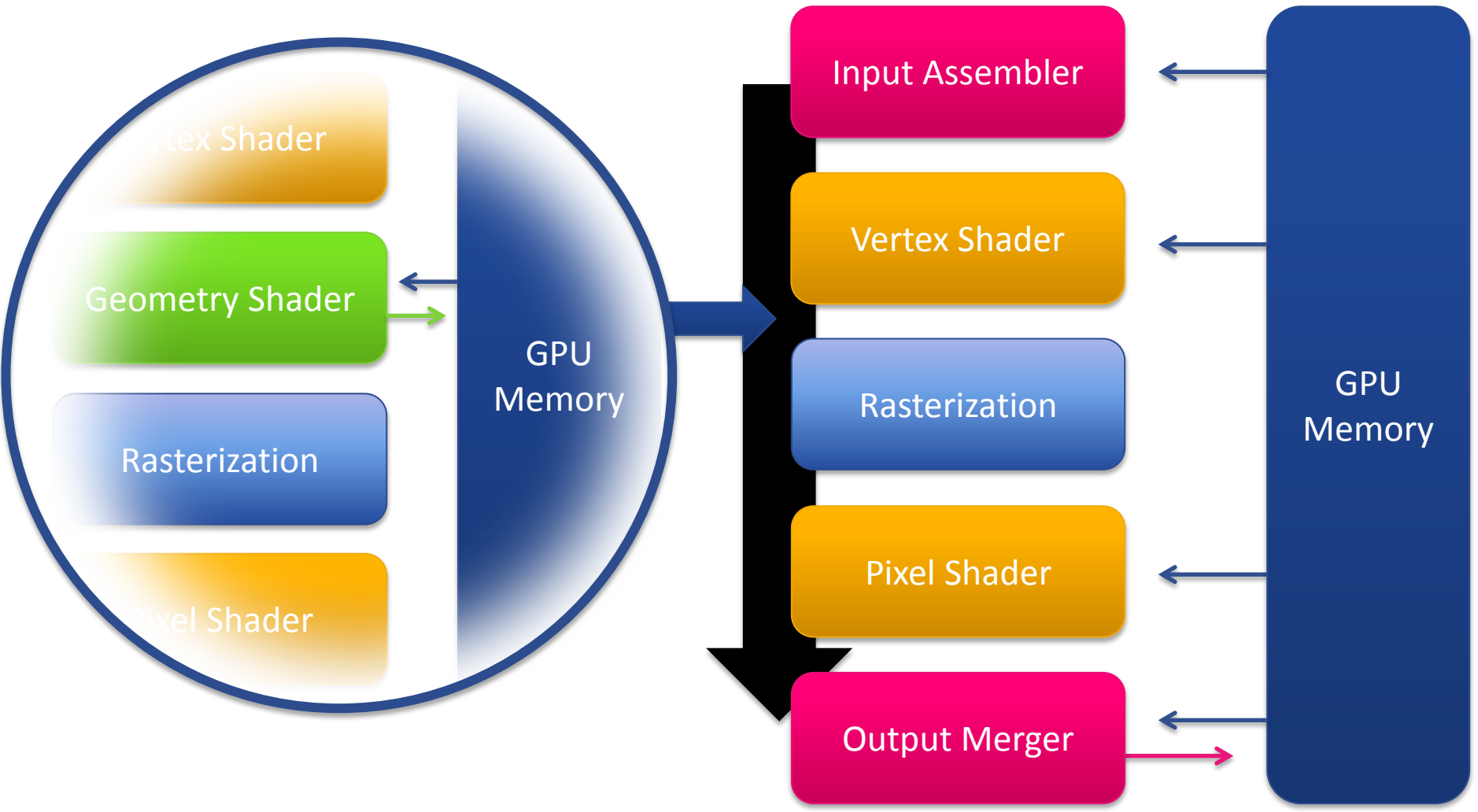
- Before Vertex Shader
- Assembles data:
  - Vertex Buffer
  - Index Buffer
  - PrimitiveType

Output Merger

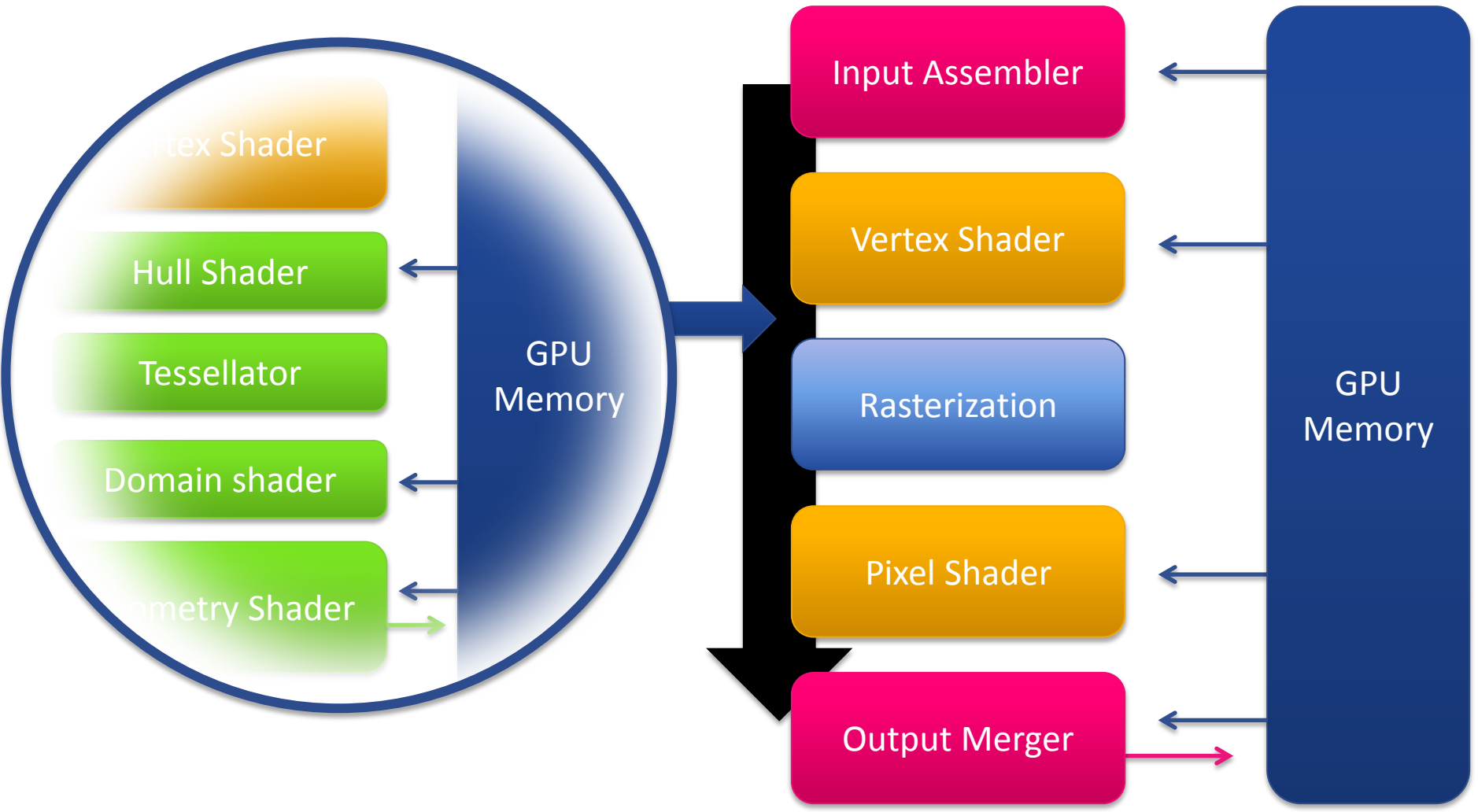
- After Pixel Shader
- Z-buffer testing
- Blending
- Write to render target



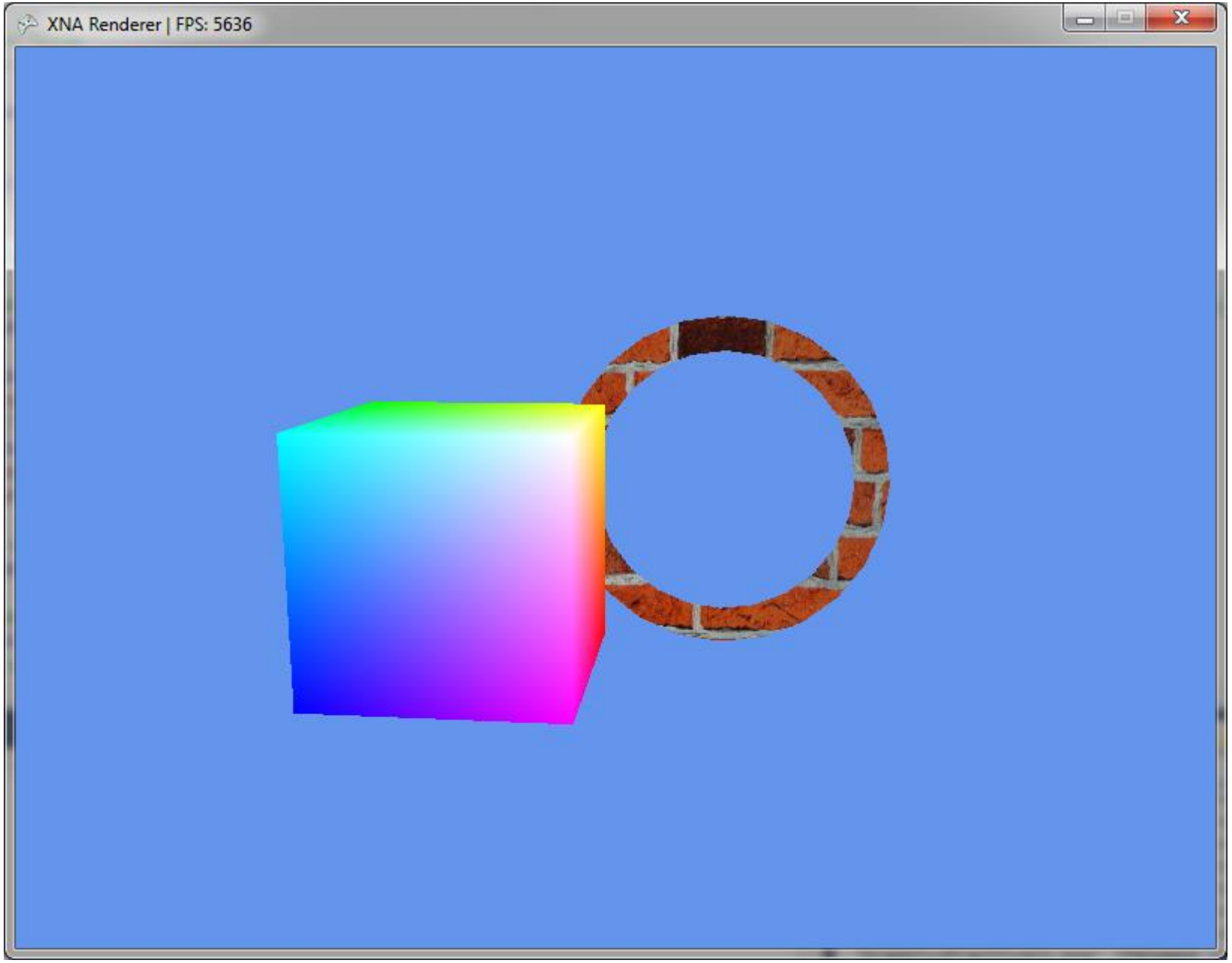
DirectX 10


Geometry Shader
Rasterization
GPU Memory
Input Assembler
Vertex Shader
Rasterization
Pixel Shader
Output Merger
GPU Memory

DirectX 11


tex Shader
Hull Shader
Tessellator
Domain shader
metry Shader
GPU Memory
Input Assembler
Vertex Shader
Rasterization
Pixel Shader
Output Merger
GPU Memory

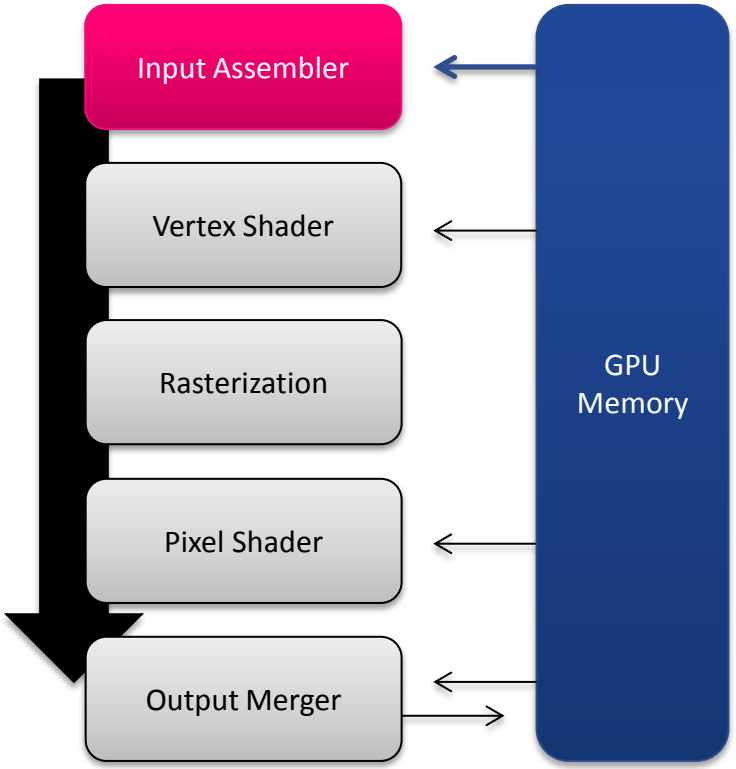
2. Pipeline example

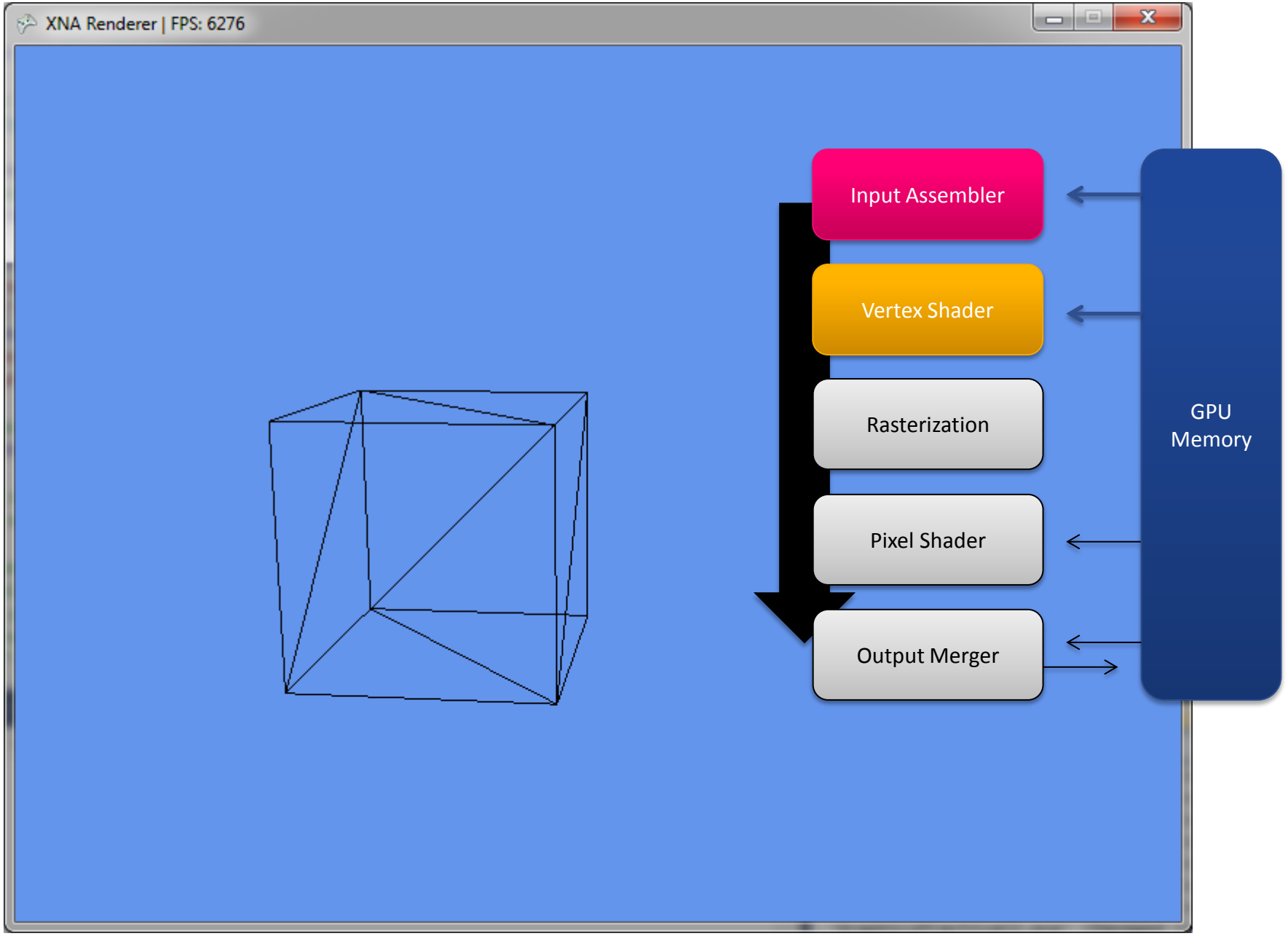

XNA Renderer | FPS: 5636

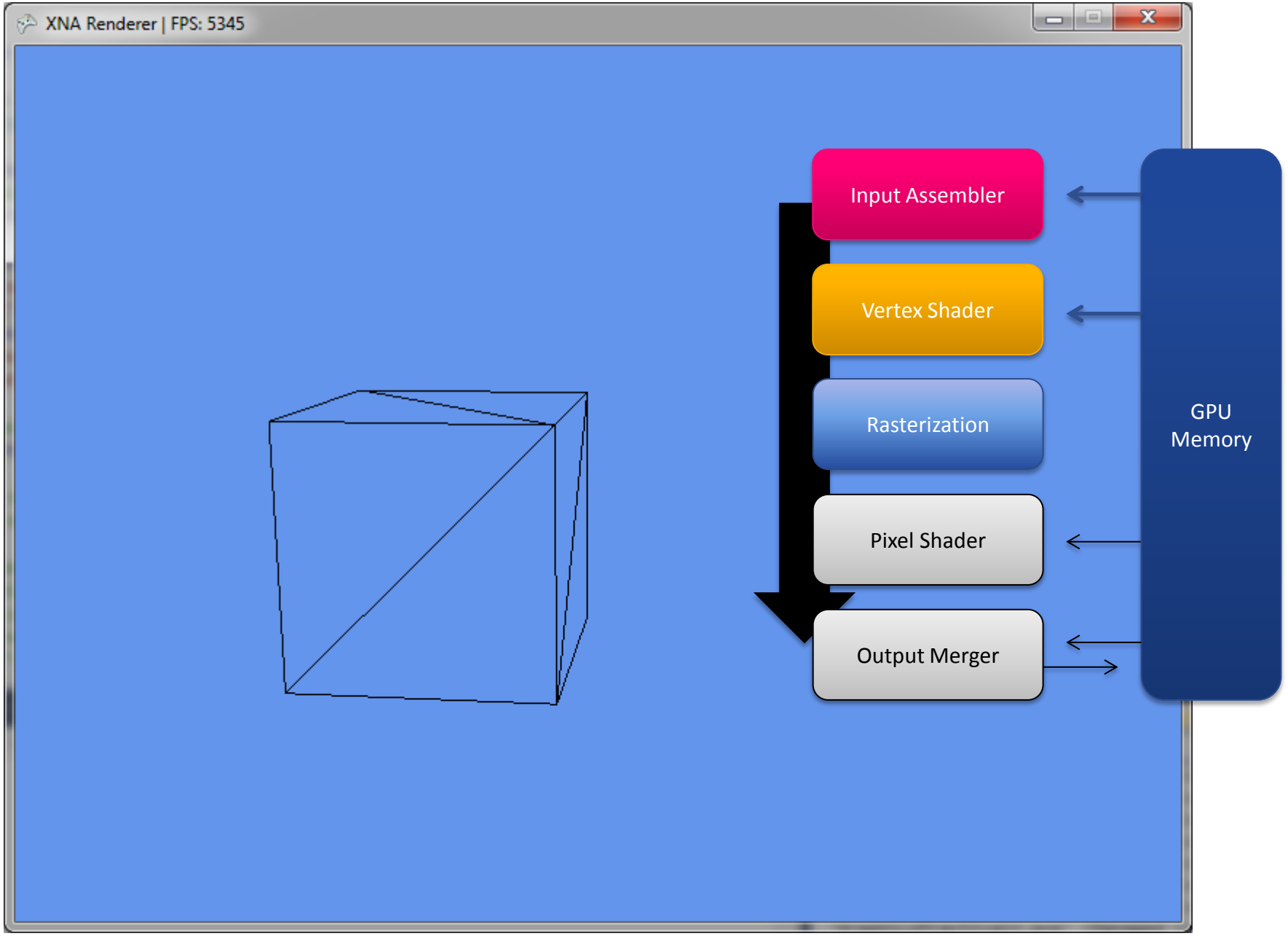
Box

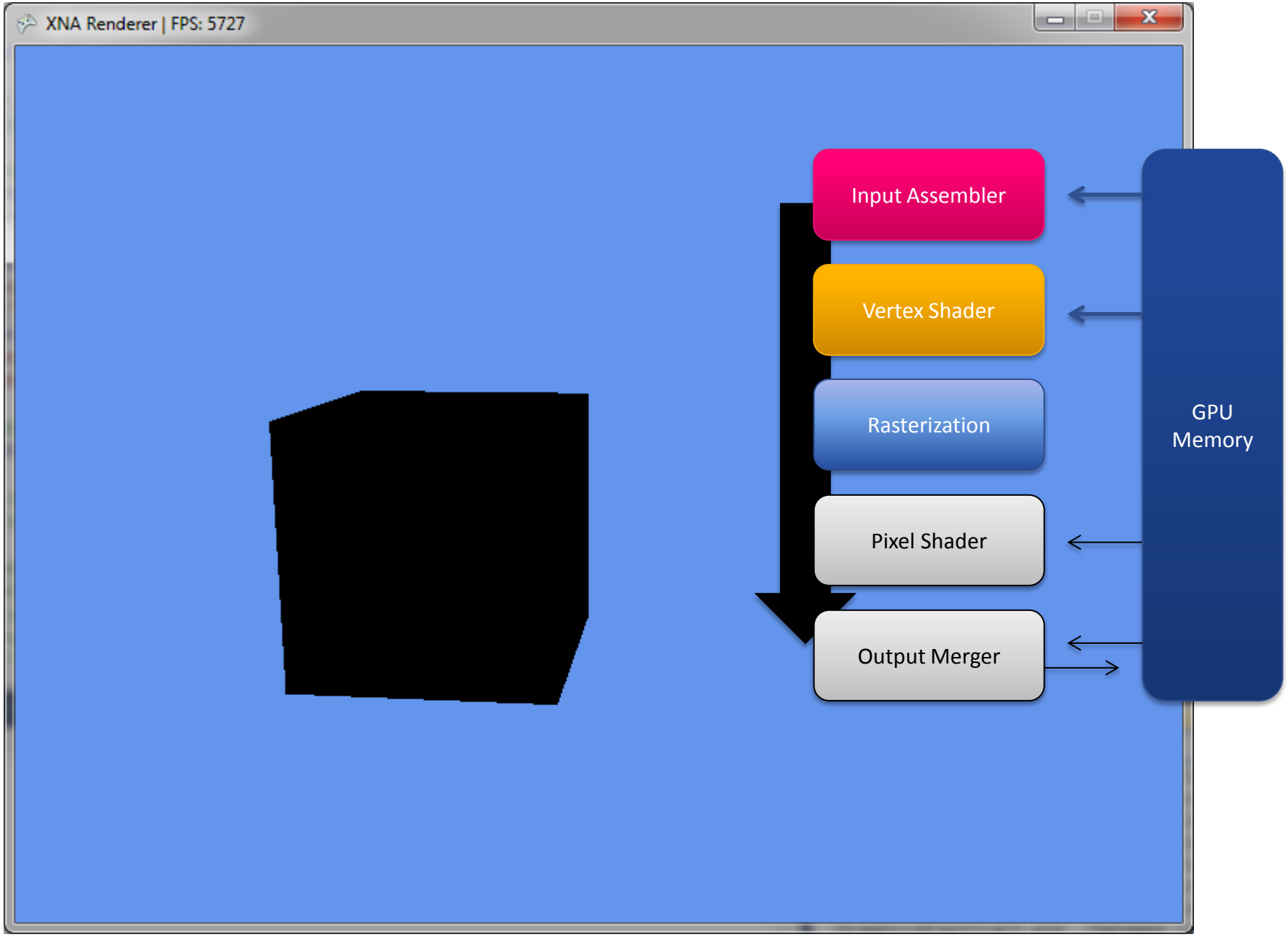
Input vertices

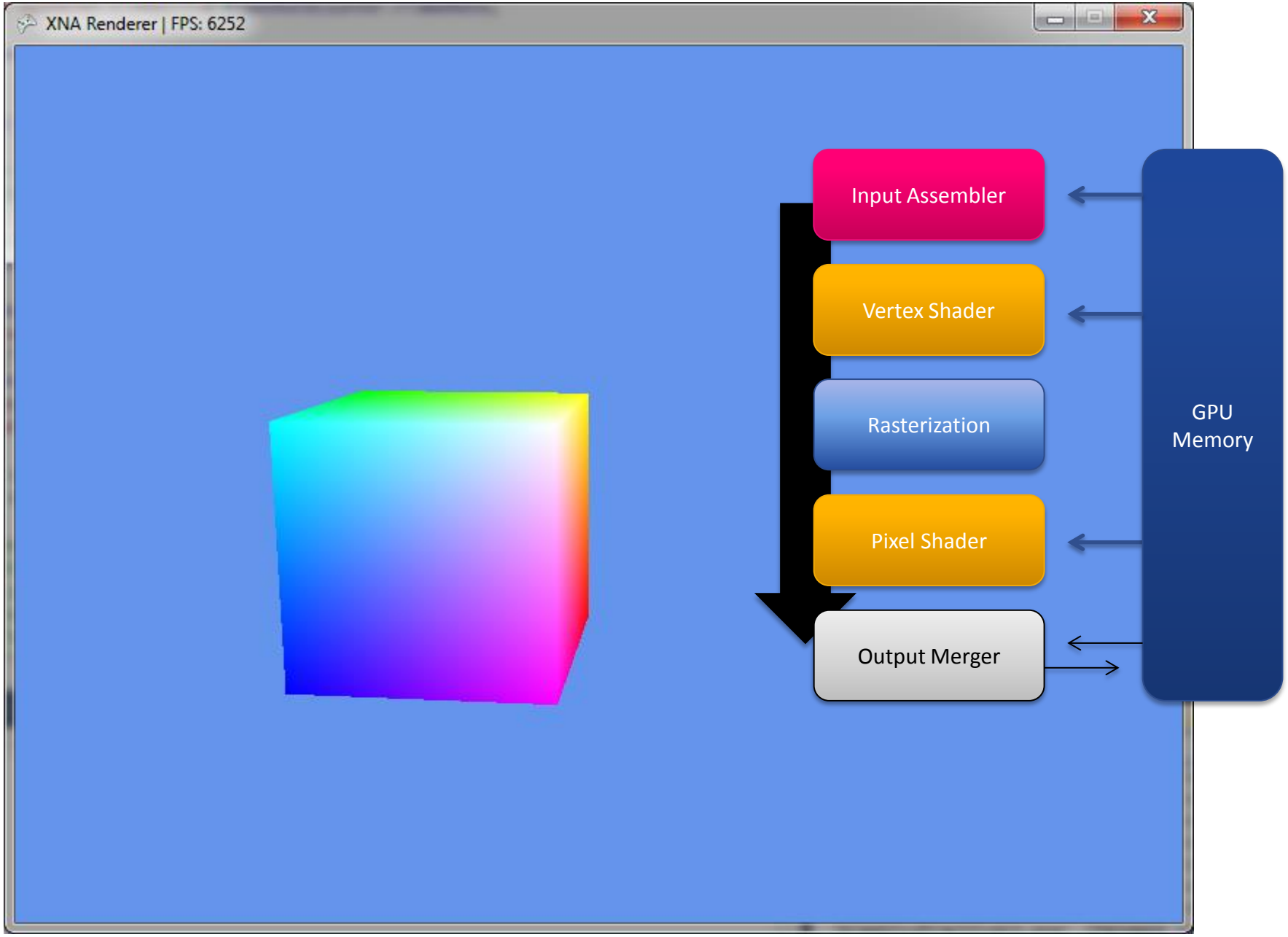
- `new Vector3(1.0f, 1.0f, 1.0f)`
- `new Vector3(1.0f, 1.0f, -1.0f)`
- `new Vector3(1.0f, -1.0f, 1.0f)`
- `new Vector3(1.0f, -1.0f, -1.0f)`
- `new Vector3(-1.0f, 1.0f, 1.0f)`
- `new Vector3(-1.0f, 1.0f, -1.0f)`
- `new Vector3(-1.0f, -1.0f, 1.0f)`
- `new Vector3(-1.0f, -1.0f, -1.0f)`

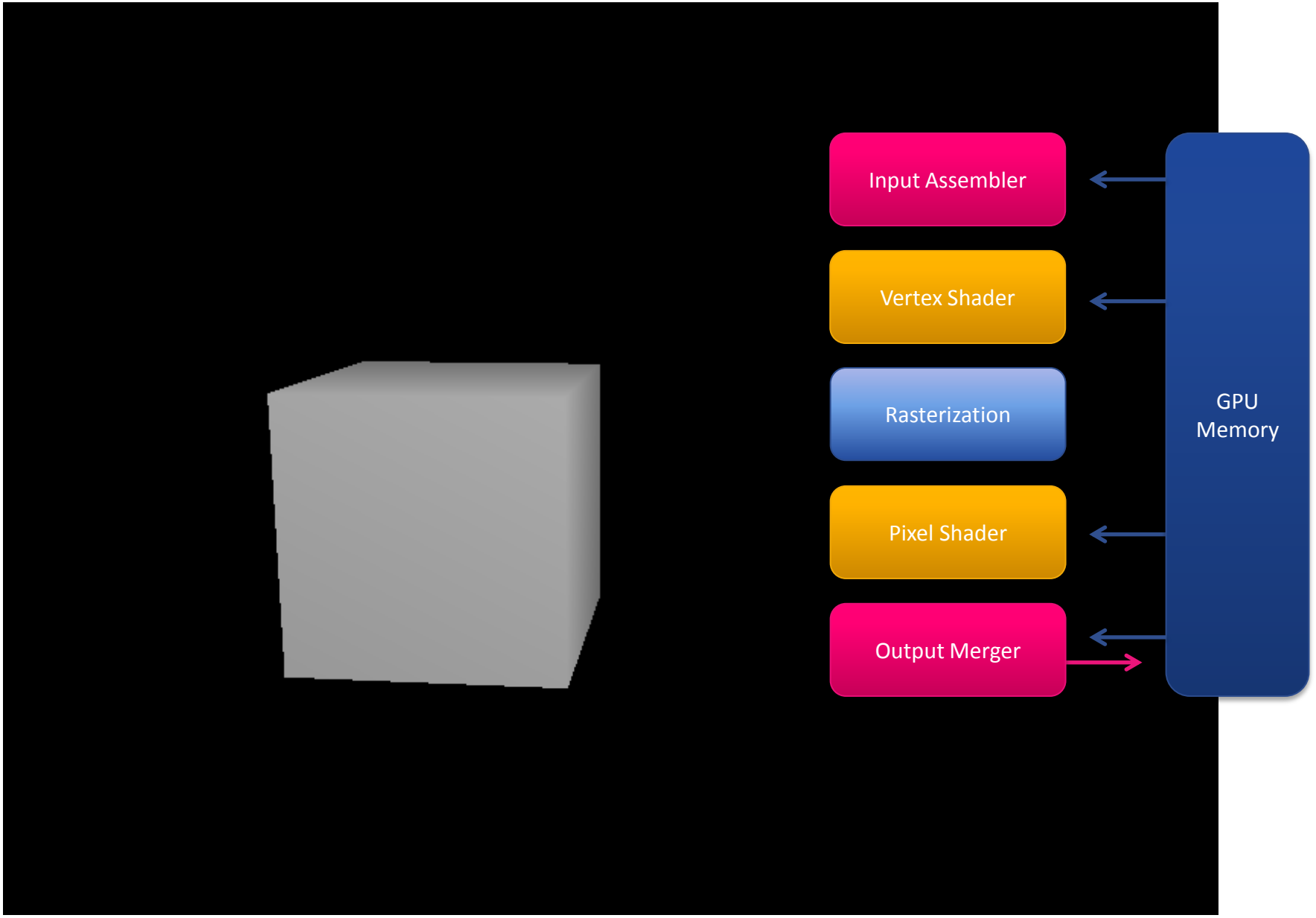


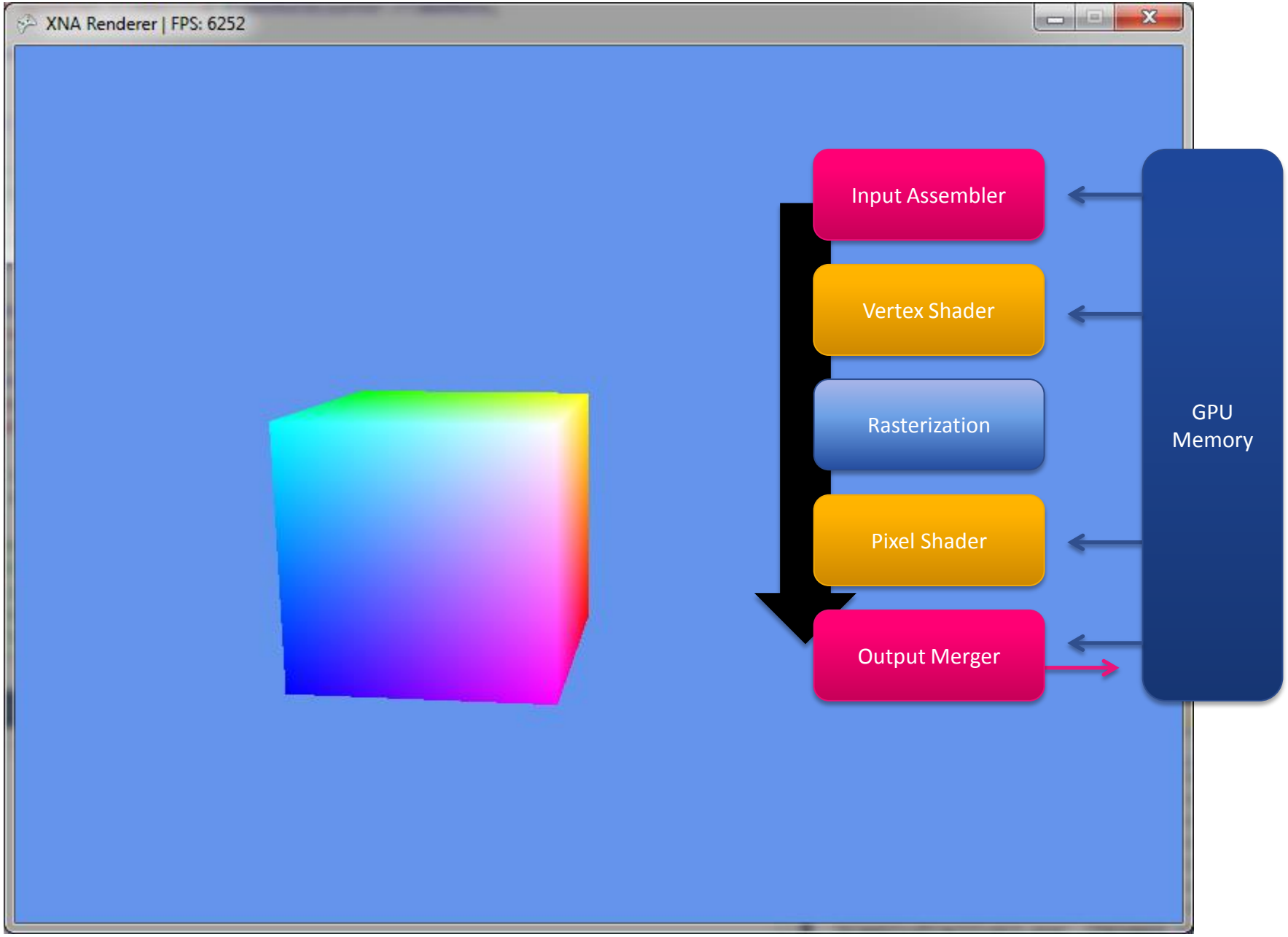

XNA Renderer | FPS: 6276
Input Assembler
Vertex Shader
Rasterization
Pixel Shader
Output Merger
GPU Memory

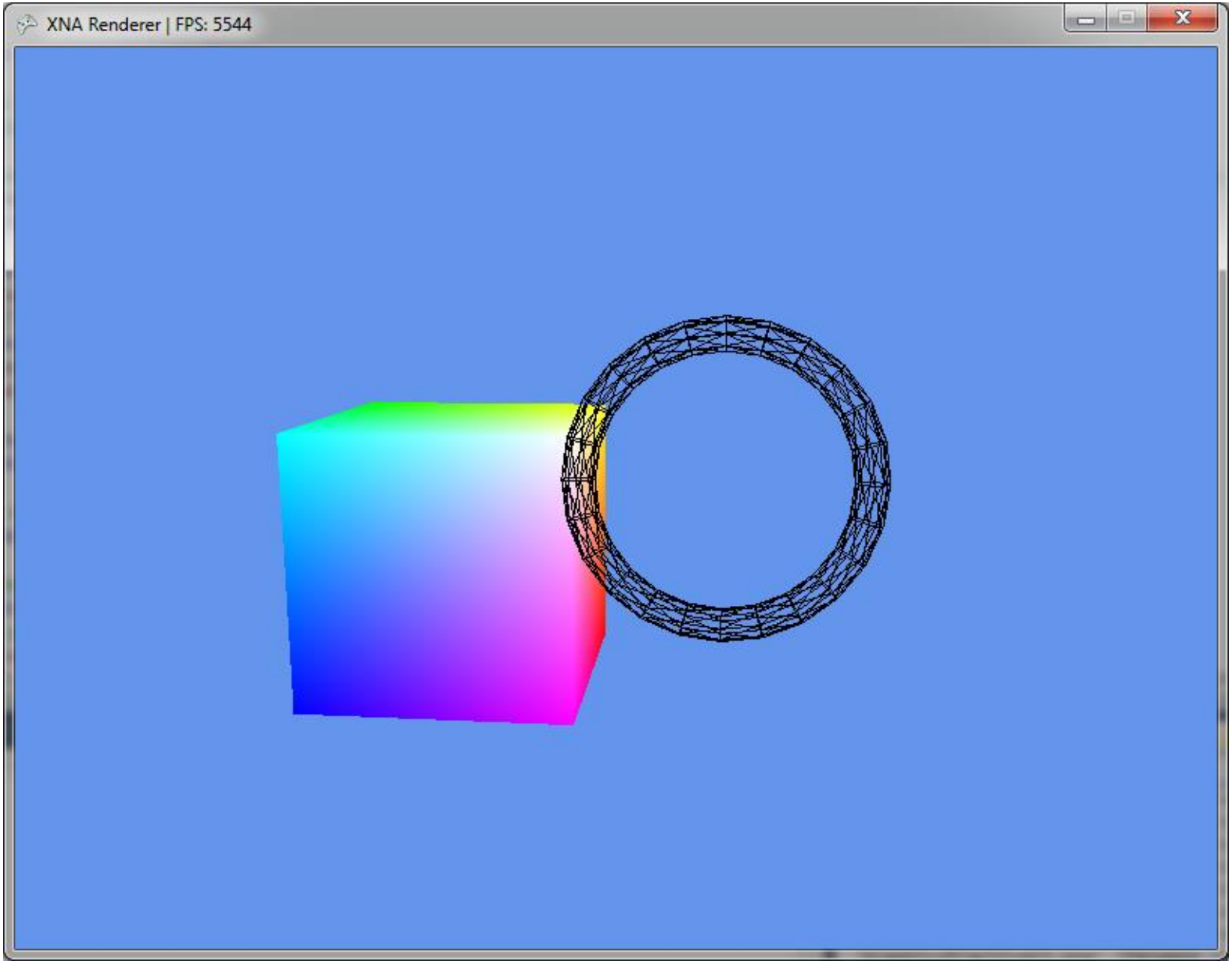

XNA Renderer | FPS: 5345
Input Assembler
Vertex Shader
Rasterization
Pixel Shader
Output Merger
GPU Memory

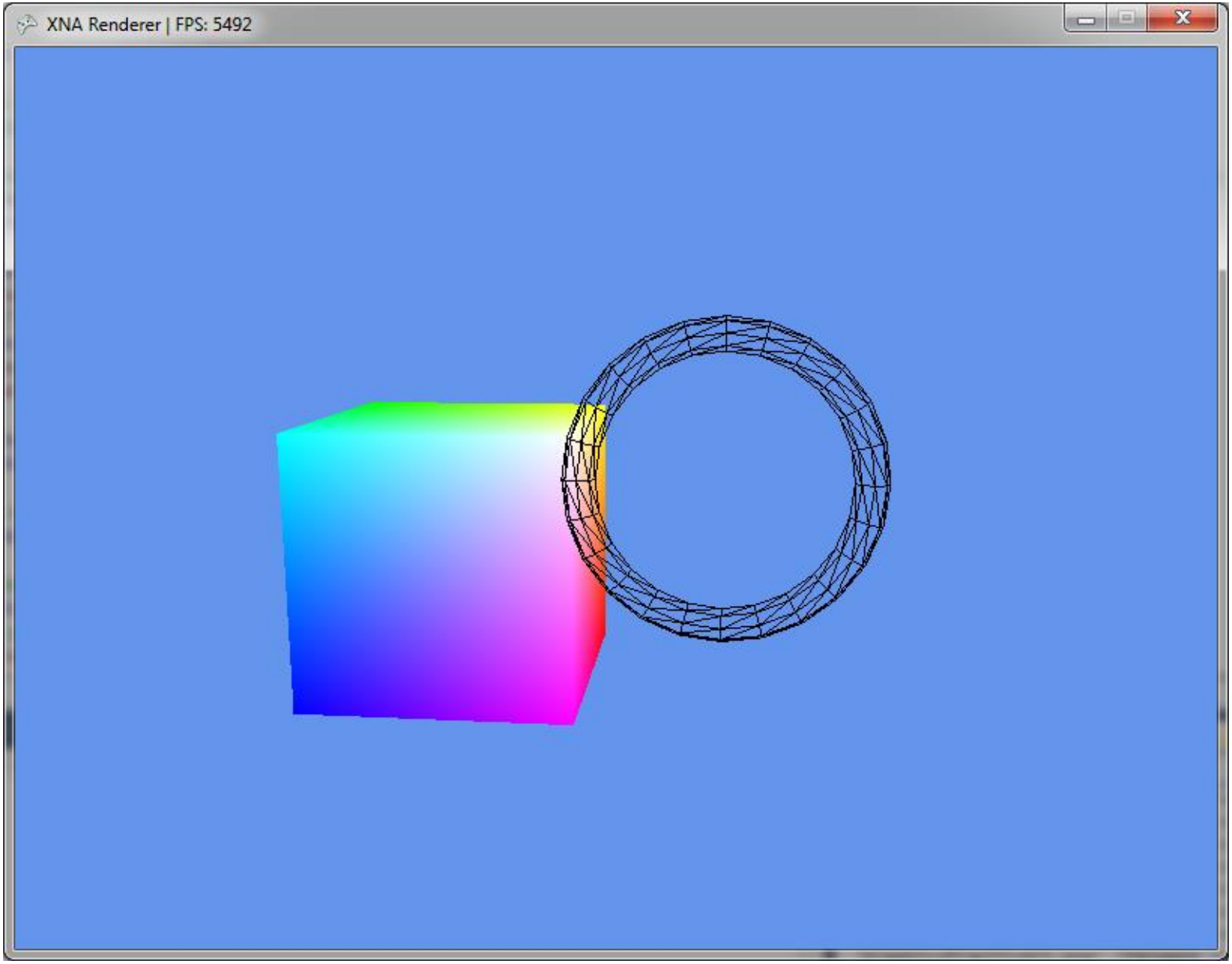

XNA Renderer | FPS: 5727
Input Assembler
Vertex Shader
Rasterization
Pixel Shader
Output Merger
GPU Memory

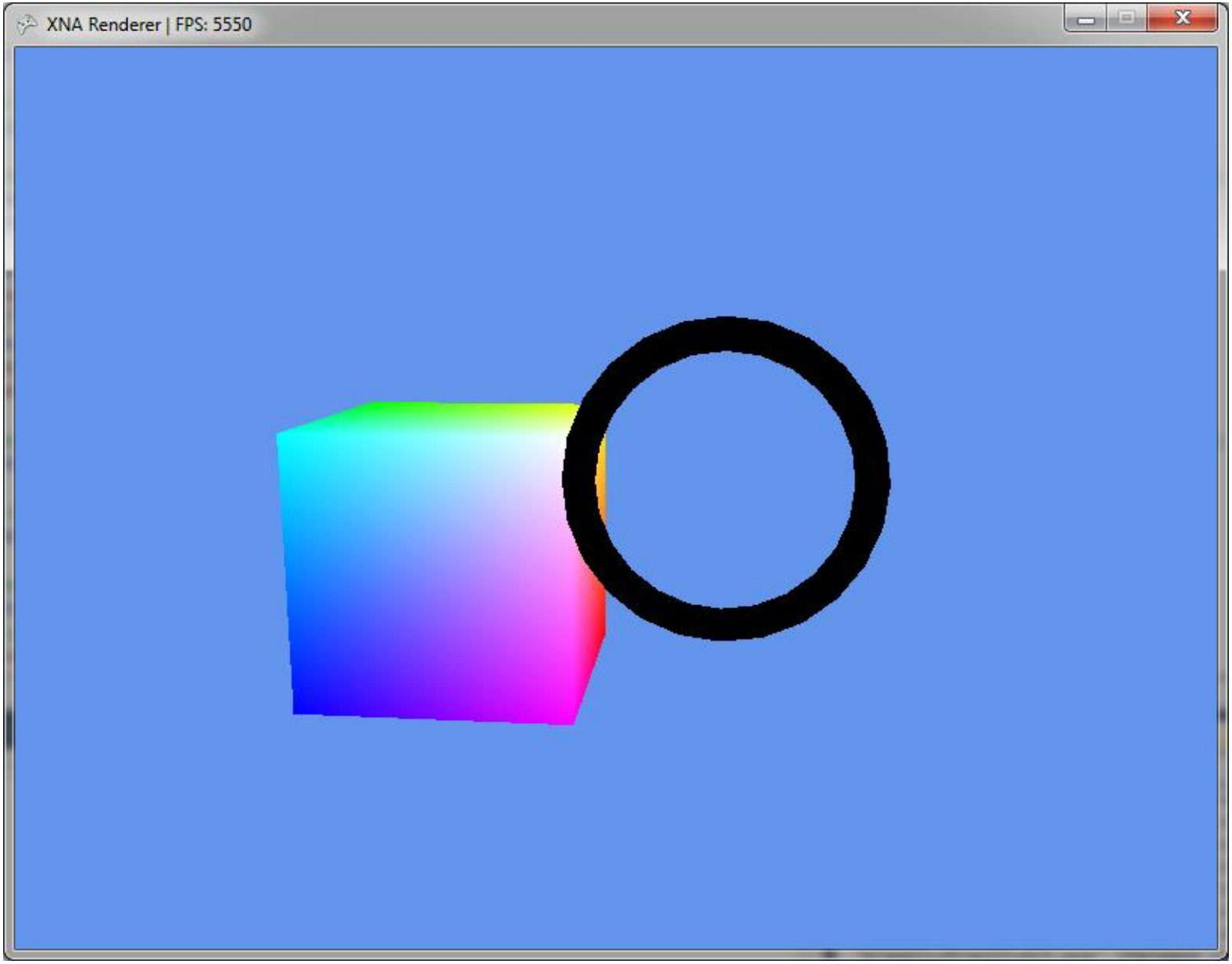

XNA Renderer | FPS: 6252
Input Assembler
Vertex Shader
Rasterization
Pixel Shader
Output Merger
GPU Memory

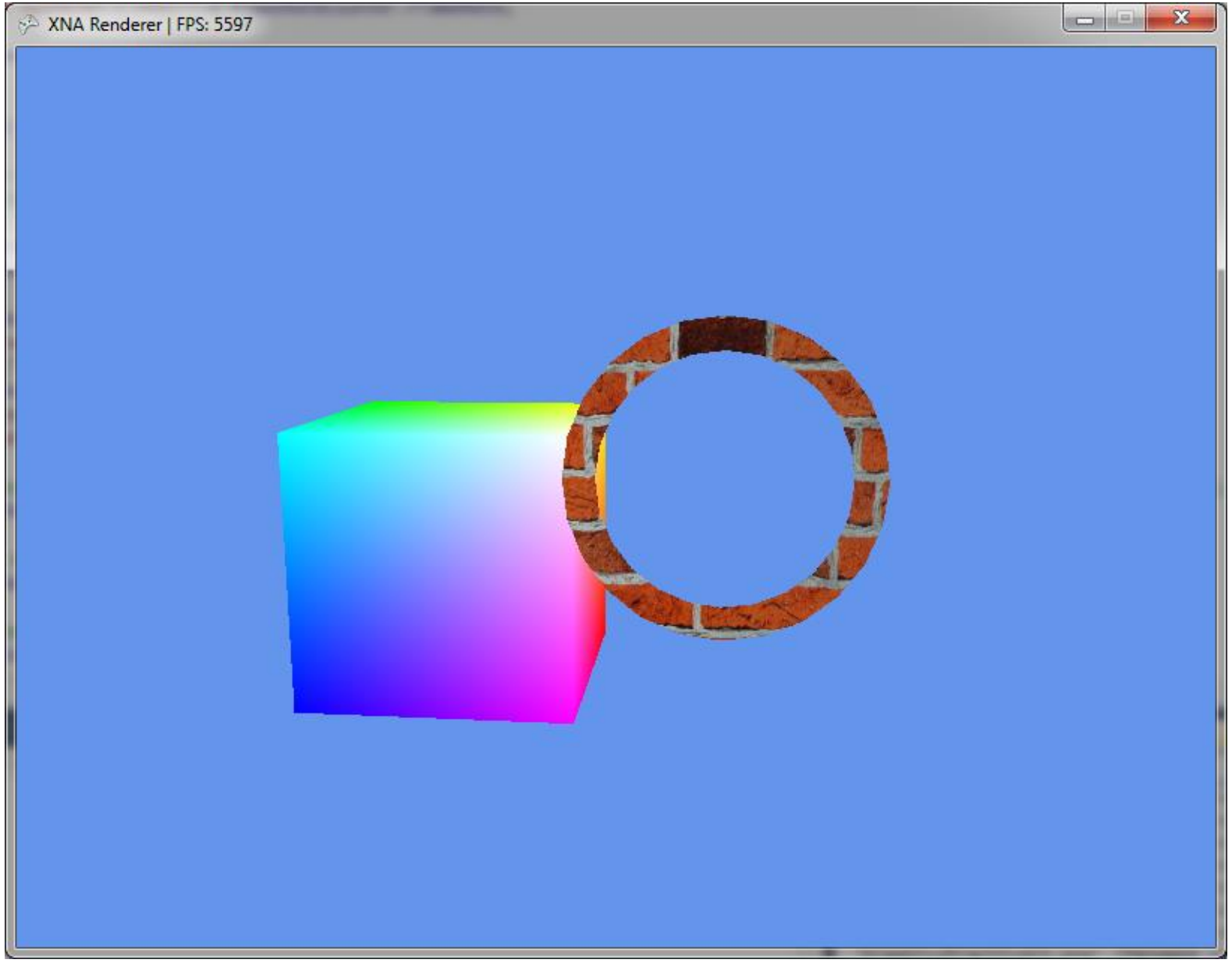

Input Assembler
Vertex Shader
Rasterization
Pixel Shader
Output Merger
GPU Memory

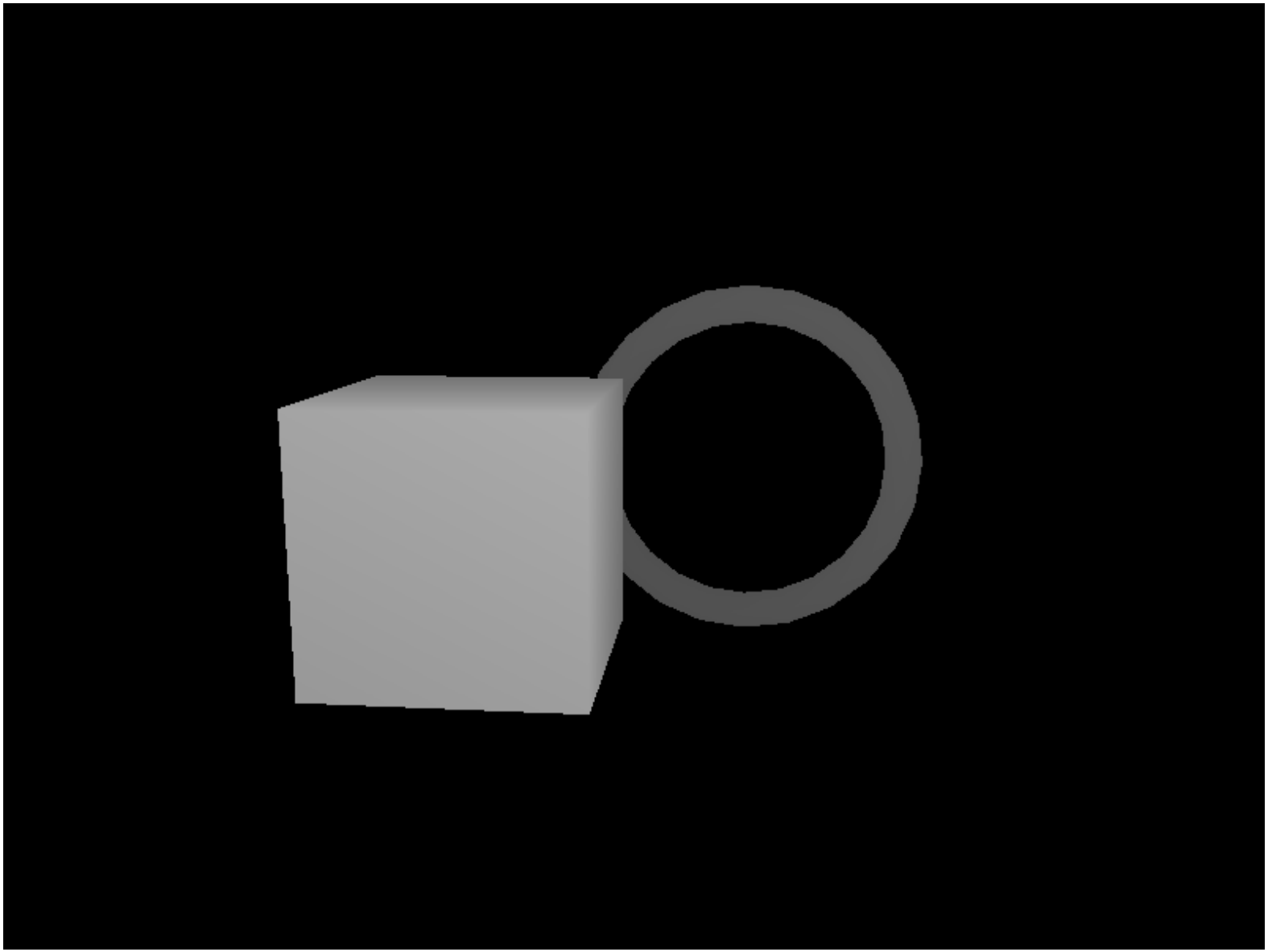

XNA Renderer | FPS: 6252
Input Assembler
Vertex Shader
Rasterization
Pixel Shader
Output Merger
GPU Memory

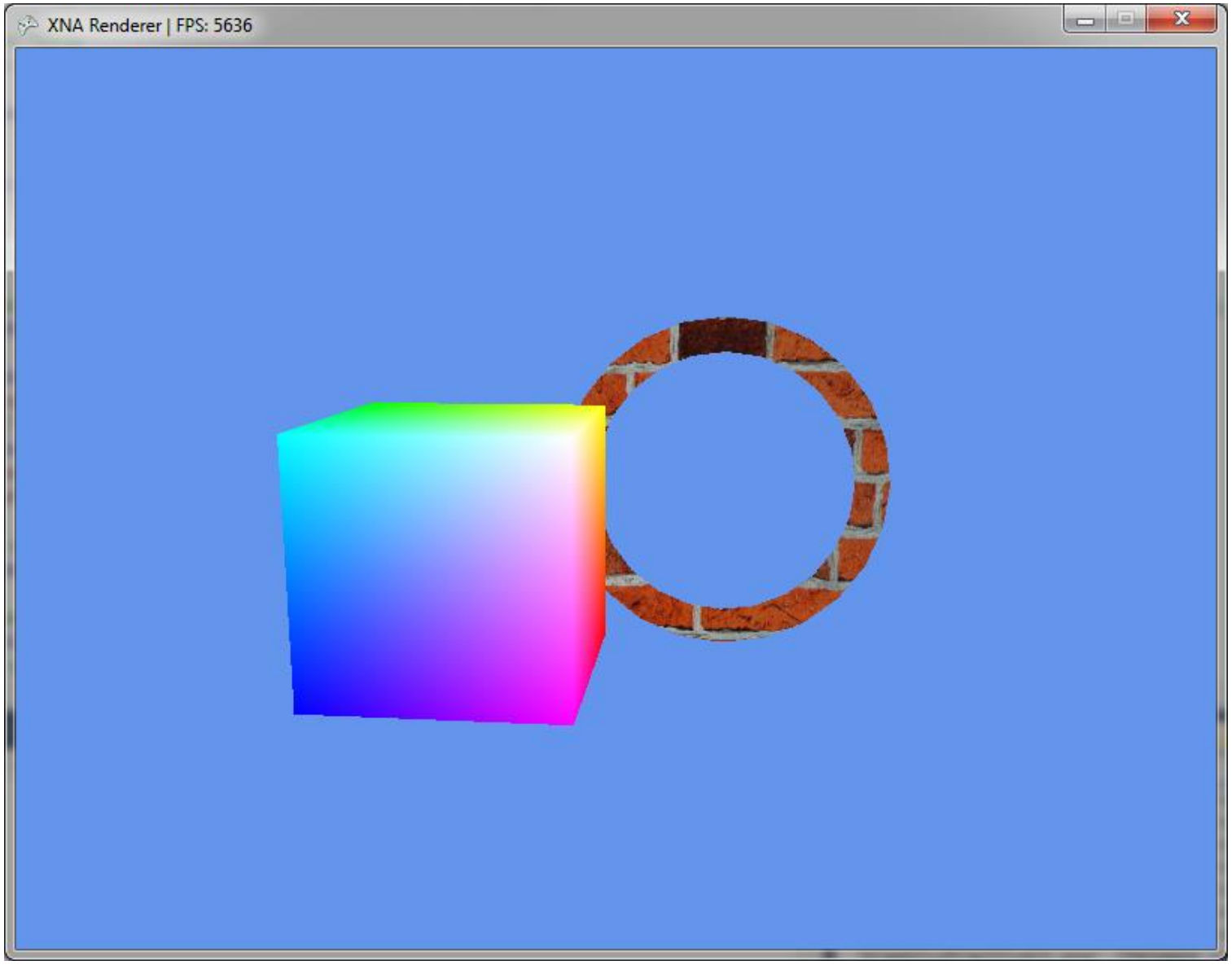

XNA Renderer | FPS: 5544


XNA Renderer | FPS: 5492


XNA Renderer | FPS: 5550

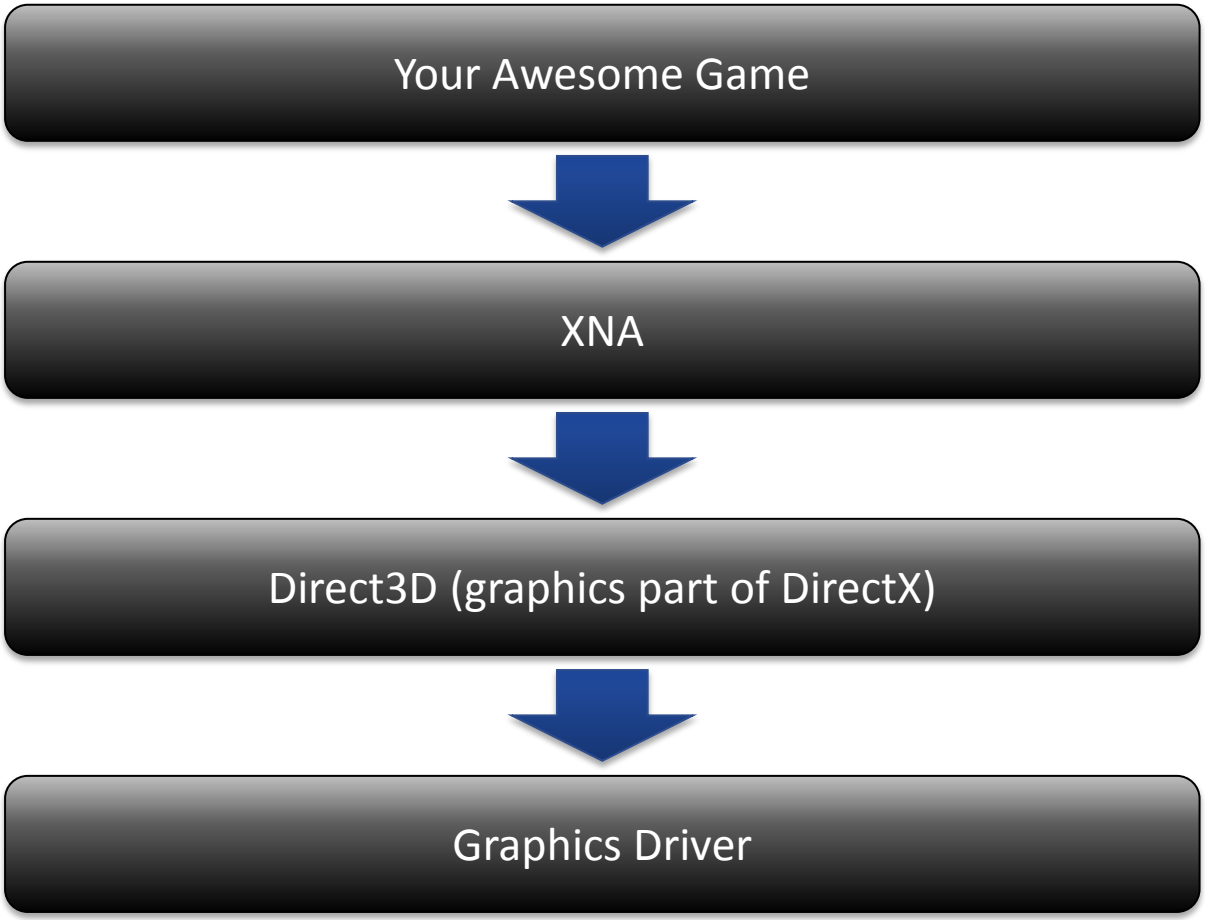

XNA Renderer | FPS: 5597




XNA Renderer | FPS: 5636

3. Instructing the GPU

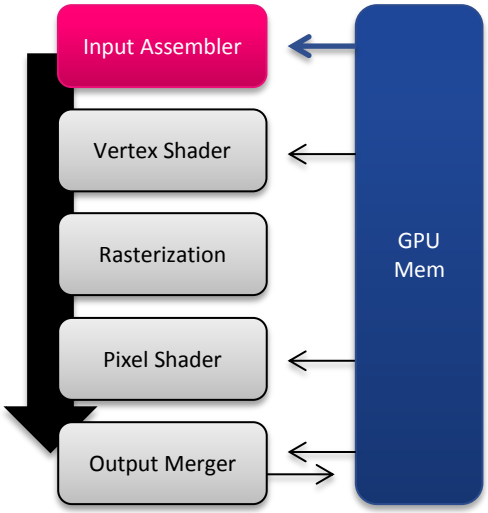
API's


Your Awesome Game
XNA
Direct3D (graphics part of DirectX)
Graphics Driver

Input Assembler Stage

Data

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



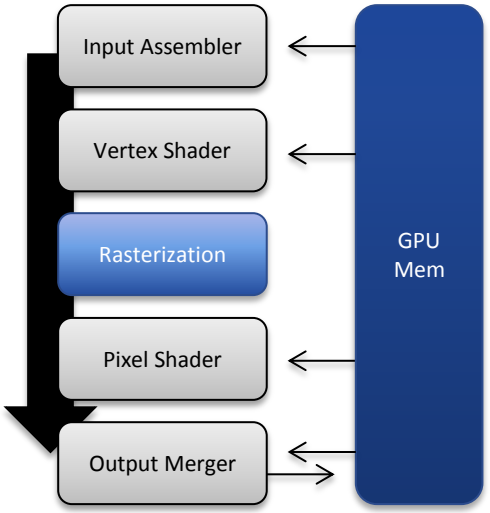
State

- VertexDeclaration
  - Implicitly activated in VertexBuffer
- PrimitiveType
  - Selected in Draw function

Rasterizer Stage

State

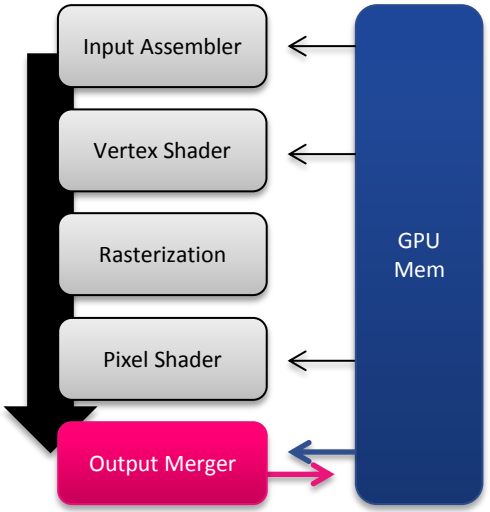
- RasterizerState
  - `GraphicsDevice.RasterizerState = …`
  - Backface culling
  - Wireframe
  - (MSAA, Scissor test, depth bias)



Output Merger Stage

State

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



Pixel Shader/Vertex Shader Stage

Shader

- Activate:
  `Effect.CurrentTechnique.Pass[0].Apply()`

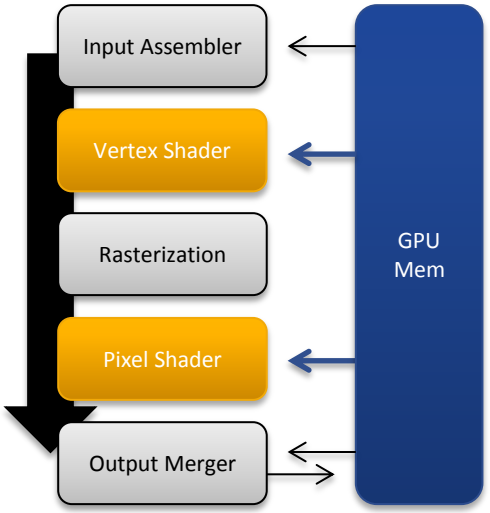
Shader Variables

- `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(…)

4. Shader programming

Shader code

HLSL – High Level Shader Language

Similar syntax to C#

- Simplified
- Specialized syntax
- Read MSDN documentation

Different style of writing code

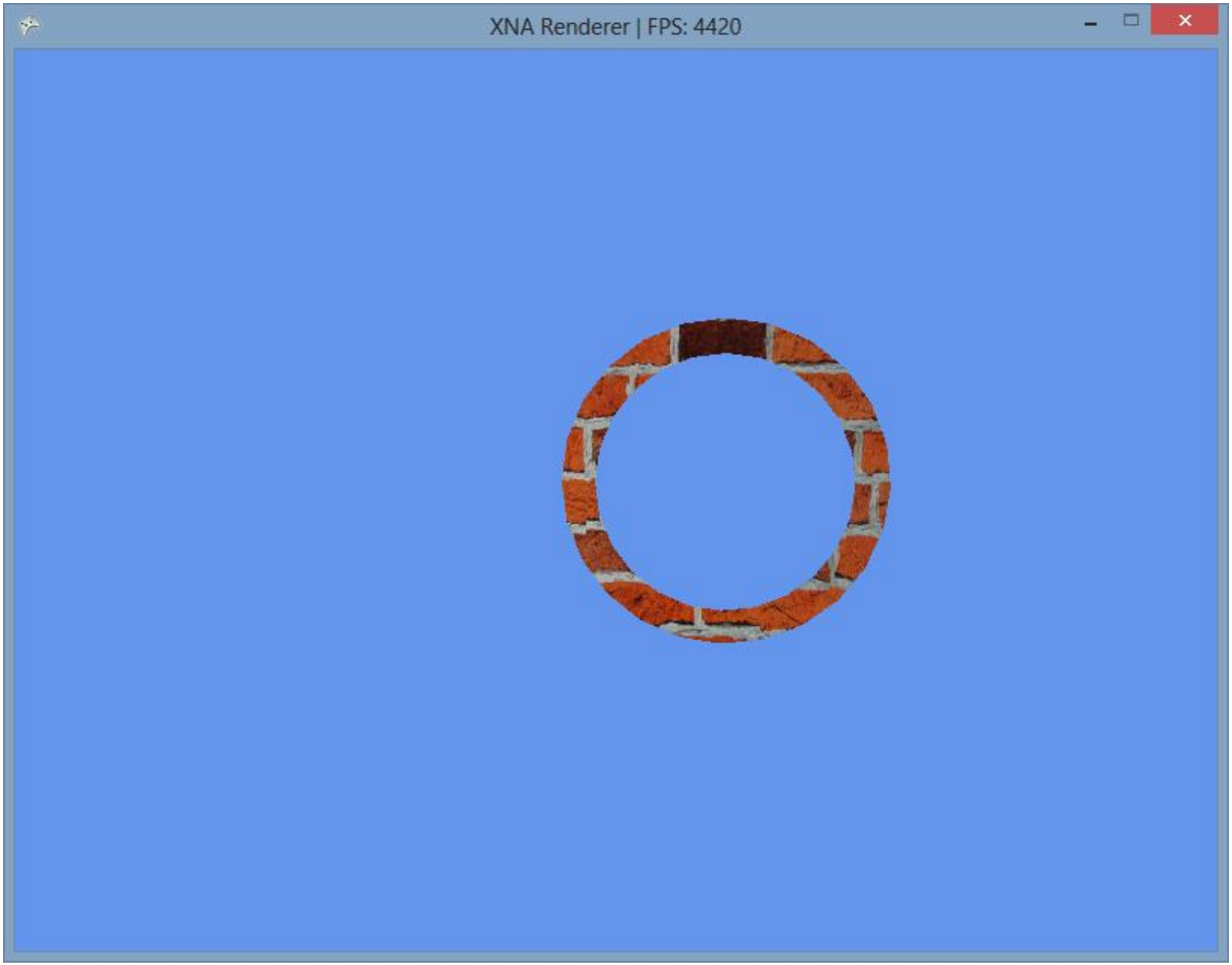
- No autocomplete
- Hard to debug
- Write incrementally

Global effect layout

- **Globals and Types**
  - 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;
}
```

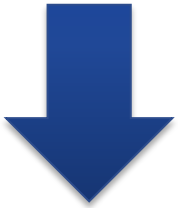
```
Texture2D BrickTexture;
SamplerState TextureSampler = sampler_state
{
	Texture = <BrickTexture>;
	MipFilter = Point;
	MinFilter = Linear;
	MagFilter = Linear;
	AddressU  = Clamp;
	AddressV  = Clamp;
};

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();
	}
}
```


XNA Renderer | FPS: 4420

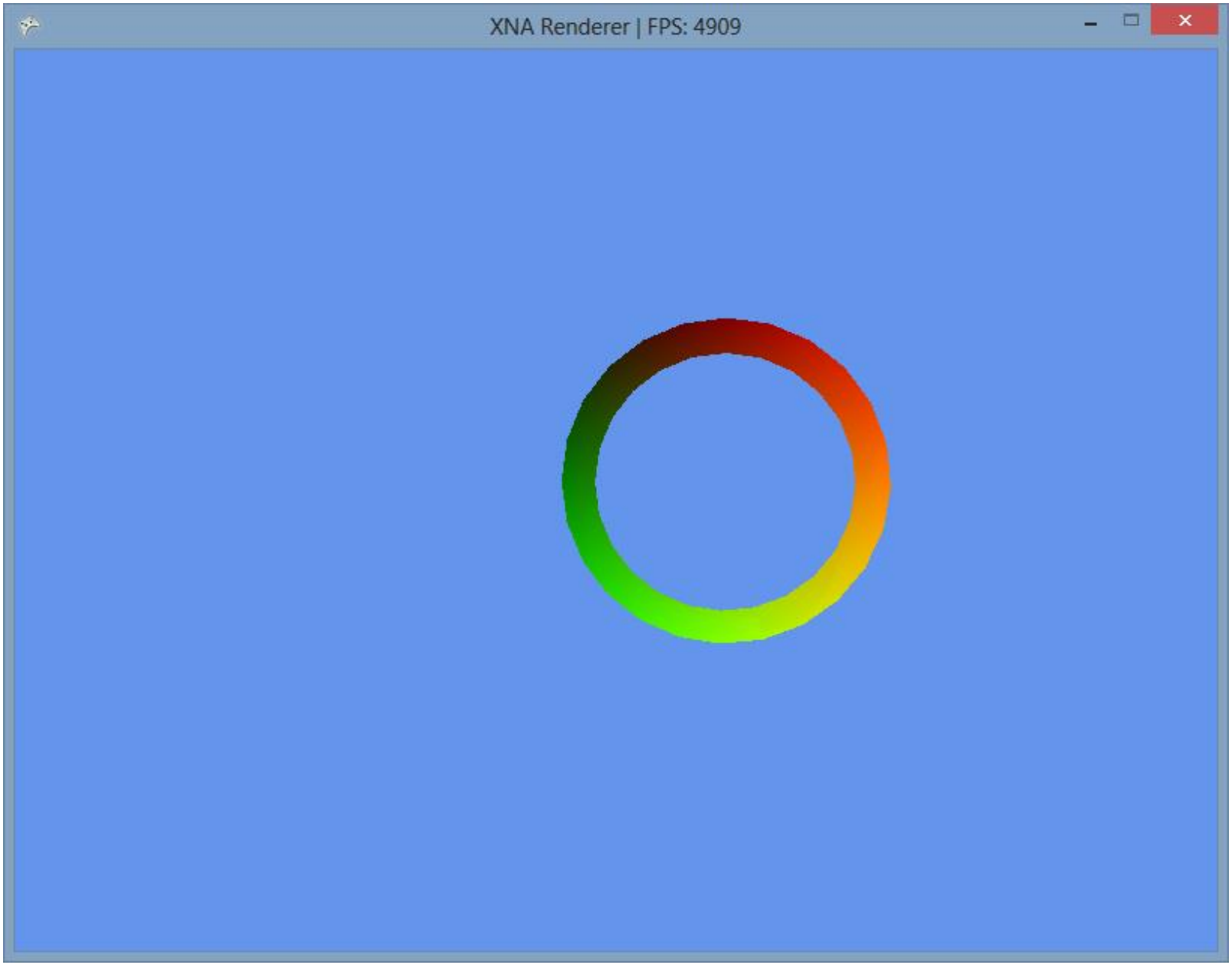
Changing the pixel shader

Freedom to change the code to whatever you want

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



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

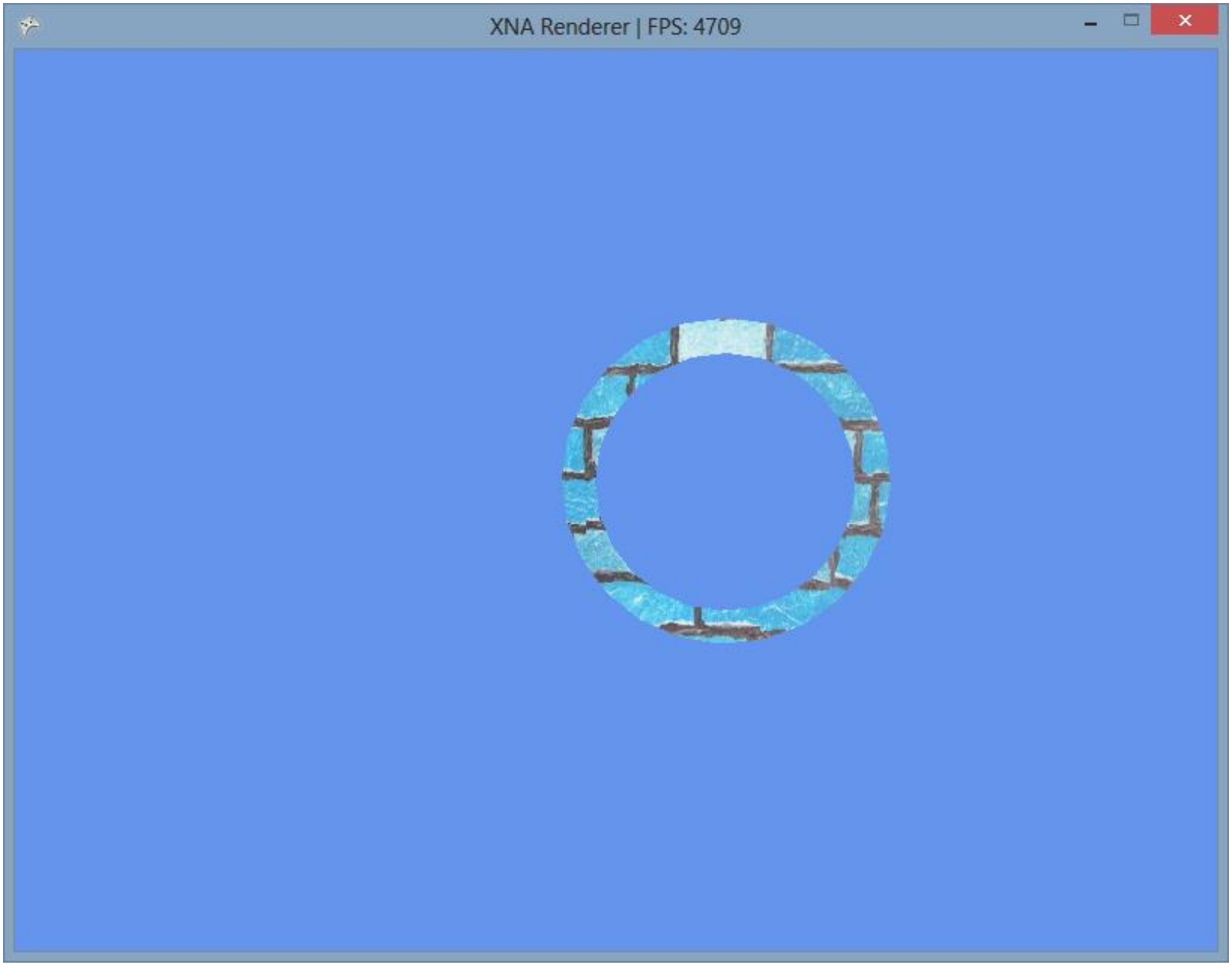

XNA Renderer | FPS: 4909

Changing the pixel shader

Inverting the color

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

```
float4 SimplePixelShader(VertexShaderOutput input) : COLOR0
{
    return 1-tex2D(TextureSampler, input.TexCoords);
}
```

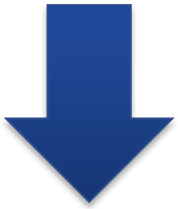

XNA Renderer | FPS: 4709

Changing the pixel shader

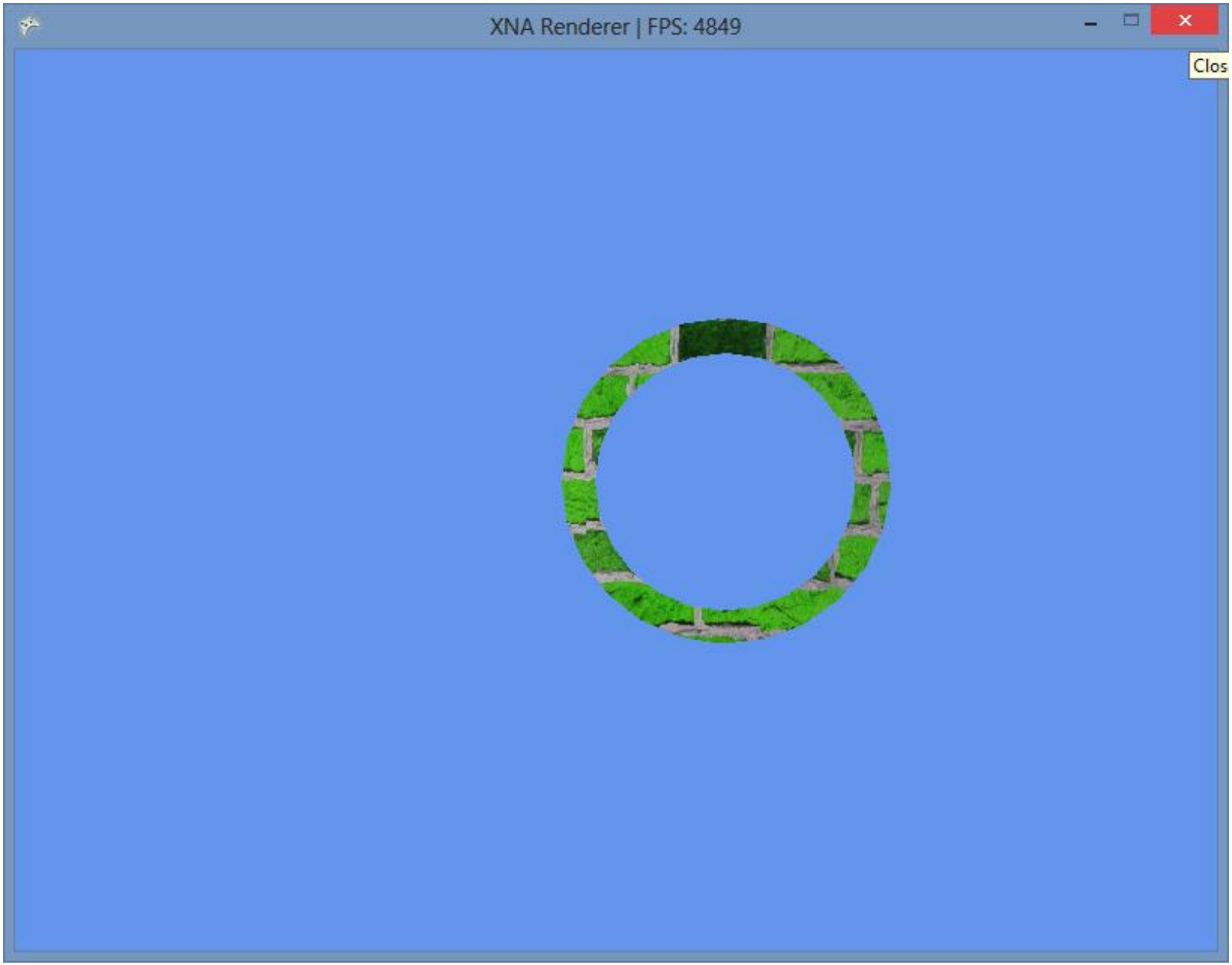
Swizzling the color

- Swap the red and green channels

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



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


XNA Renderer | FPS: 4849
Clos

Keep in mind

Avoid dynamic branching

- Short is ok:

```
if (a > b) c = 0; else c = 1;
```

Think before coding

- Write incrementally
- For debugging:

```
if (a < 0) return float4(1, 0, 0, 1);
```

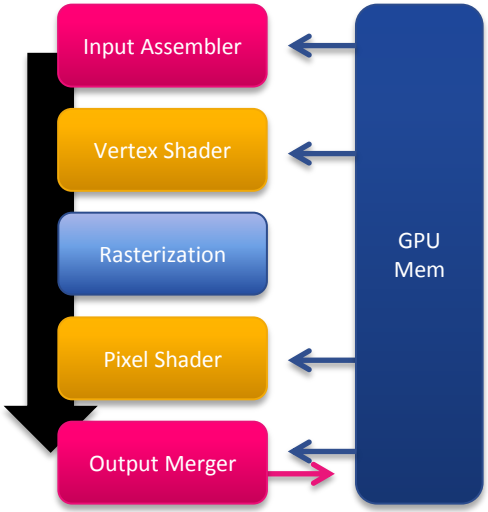
Aggressive compiler

- 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