Hogeschool van Amsterdam

The Rendering Pipeline

Advanced Graphics Programming

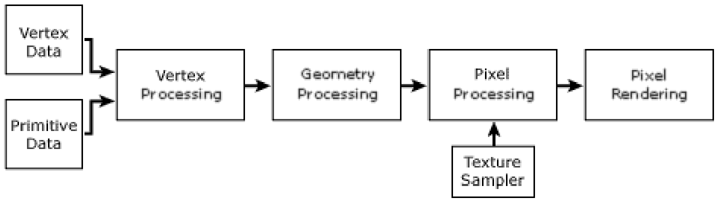
Alexander Bonnee,
Richard de Koning,
Reggie Schildmeijer

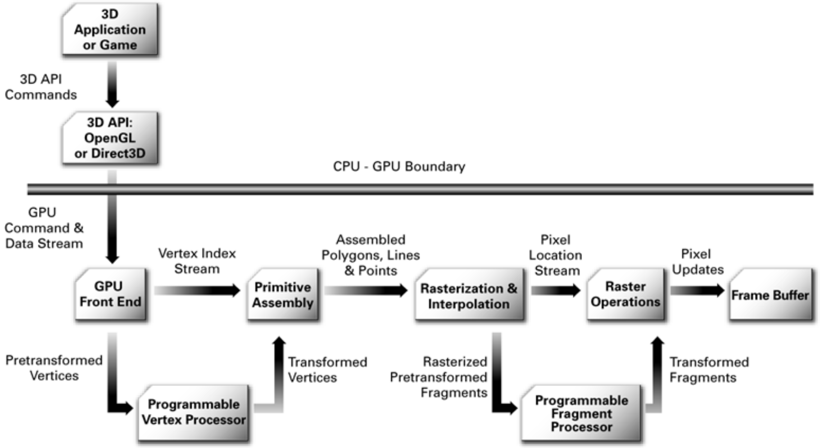
CREATING TOMORROW

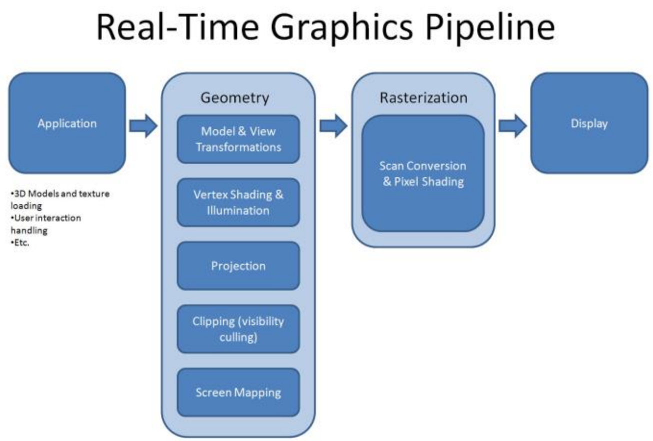
The Rendering Pipeline

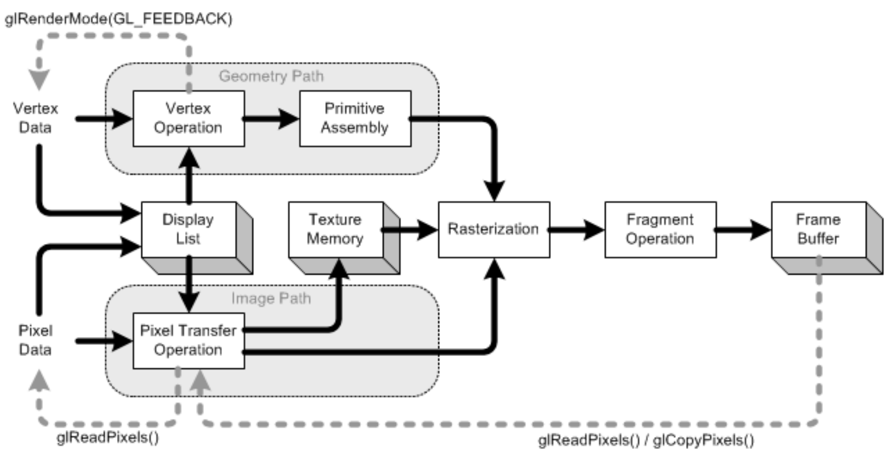
- What Rendering Pipeline?
- Level of Abstraction
- Fixed vs. Programmable
- DirectX 11 Graphics Pipeline
- Vertex and Pixel Shaders
- Unified Pipeline
- Hardware, Shader Model and Api version
- Example Shaders
- Game engine vs Render Engine

What rendering pipeline?





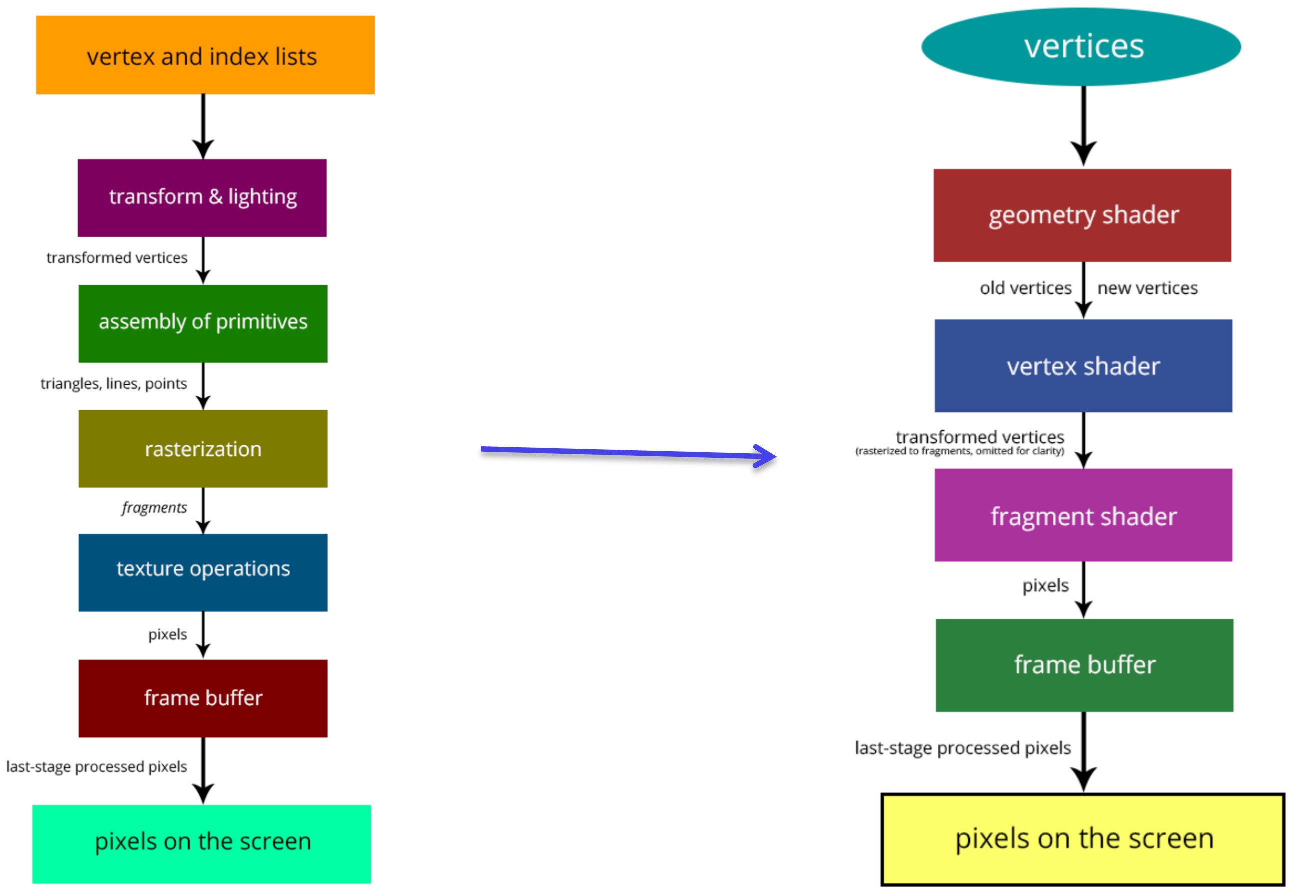




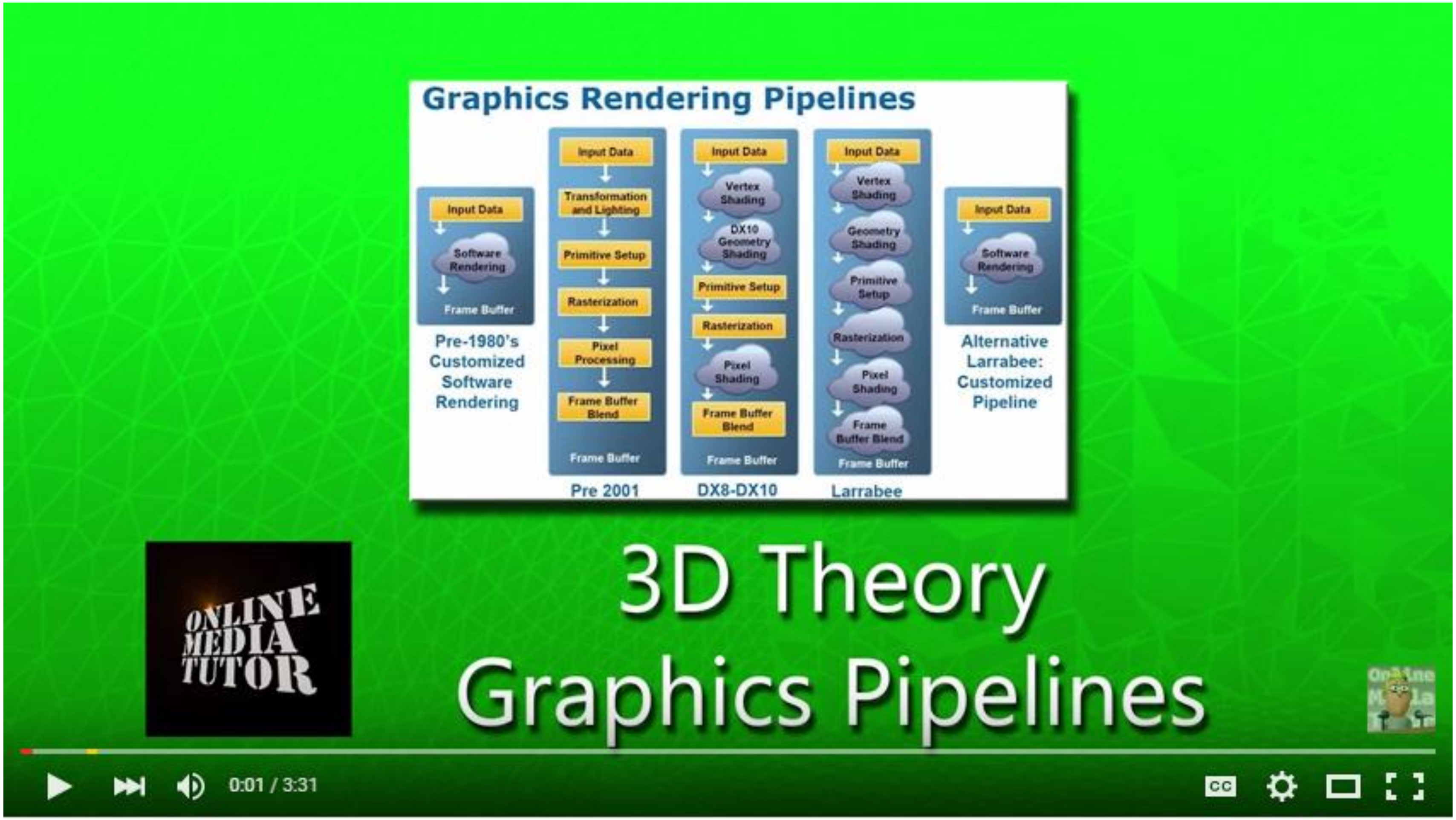
Level of abstraction

- GPU capabilities (Hardware)
  - Fixed function pipeline vs. Programmable Pipeline
- Driver vs Engines
  - Unity, Unreal rendering pipeline vs. DirectX, OpenGL

Fixed function vs Programmable

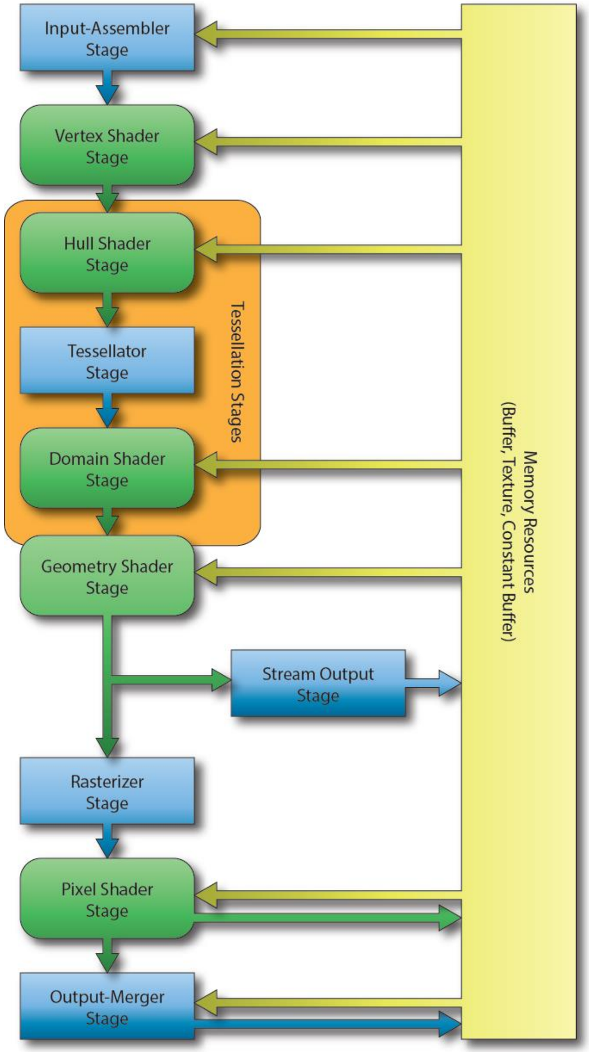


DirectX 11 Graphics Pipeline



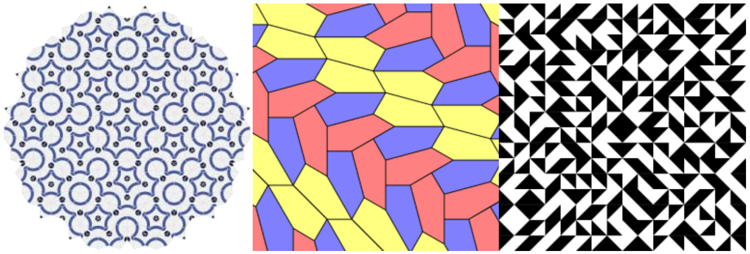
https://www.youtube.com/watch?v=bFmxMGGdBrk

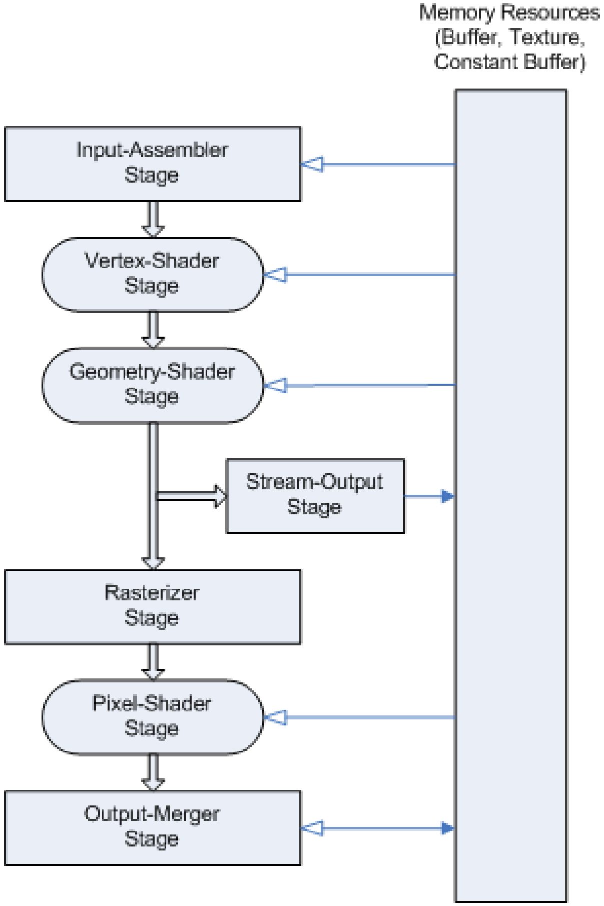
DIRECTX 11 GRAPHICS PIPELINE



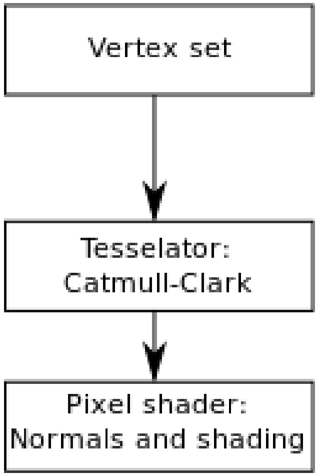
- 10 stages
- Square boxes = fixed function stages
- Round boxes = shaders
- Yellow box = bound resources

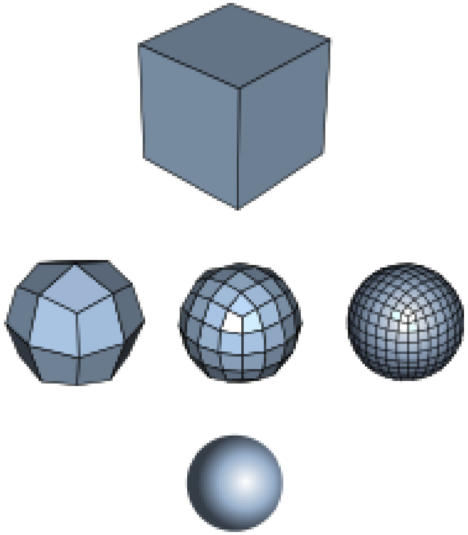
SIMPLIFIED PIPELINE





- Removed tesselation
- Tiling of a plane
- Dynamic LOD, terrain rendering
- Mesh generating algorithm
- Tiling of a plane, shape





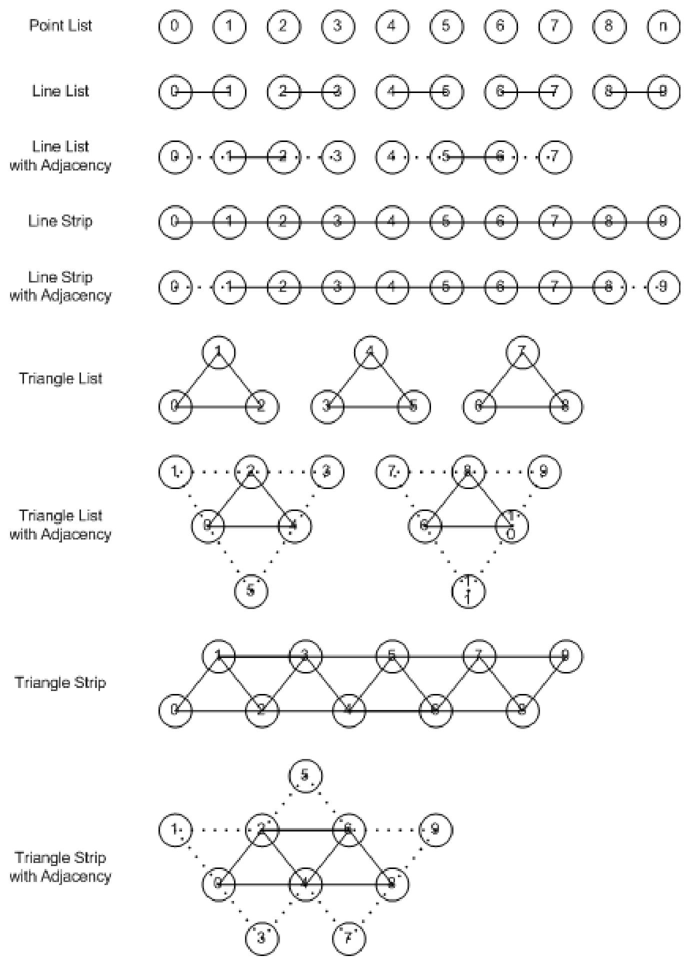
LETS PLAY PIPELINE ☺

- Suppose the following databuffer of primitive data is bound to the Input Assembler

- (0,0)
- (1,1)
- (2,2)

What will be drawn?

INPUT ASSEMBLER



<u>Input-Assembler Stage</u> - The input-assembler stage is responsible for supplying data such as triangles, lines and points to the pipeline.

The purpose of the input-assembler stage is

- to read primitive data such as
- points, lines and/or triangles
- from user-filled buffers and
- assemble the data into primitives
- that will be used by the other pipeline stages.

Line list with adjacency or a triangle list with adjacency have been added to support the geometry shader.

INPUT ASSEMBLER

```
struct SimpleVertex
{
    D3DXVECTOR3 Position;
    D3DXVECTOR3 Color;
};

    D3D10_BUFFER_DESC bufferDesc;
    bufferDesc.Usage            = D3D10_USAGE_DEFAULT;
    bufferDesc.ByteWidth        = sizeof( SimpleVertex ) * 3;
    bufferDesc.BindFlags        = D3D10_BIND_VERTEX_BUFFER;
    bufferDesc.CPUAccessFlags   = 0;
    bufferDesc.MiscFlags        = 0;
```

1

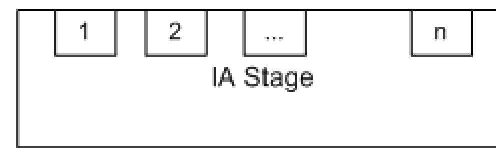
Step	Description
Create Input Buffers	Create and initialize input buffers with input vertex data.
Create the Input-Layout Object	Define how the vertex buffer data will be streamed into the IA stage by using an input-layout object.
Bind Objects to the Input-Assembler Stage	Bind the created objects (input buffers and the input-layout object) to the IA stage.
Specify the Primitive Type	Identify how the vertices will be assembled into primitives.
Call Draw Methods	Send the data bound to the IA stage through the pipeline.

eventually

Draw Methods	Description
ID3D11DeviceContext::Draw	Draw non-indexed, non-instanced primitives.
ID3D11DeviceContext::DrawInstanced	Draw non-indexed, instanced primitives.
ID3D11DeviceContext::DrawIndexed	Draw indexed, non-instanced primitives.
ID3D11DeviceContext::DrawIndexedInstanced	Draw indexed, instanced primitives.
ID3D11DeviceContext::DrawAuto	Draw non-indexed, non-instanced primitives from input data that comes from the streaming-output stage.

2

```
D3D11_INPUT_ELEMENT_DESC layout[] =
{
    { L"POSITION", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 0,
          D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { L"TEXCOORD", 0, DXGI_FORMAT_R32G32_FLOAT, 0, 12,
          D3D11_INPUT_PER_VERTEX_DATA, 0 },
    { L"NORMAL", 0, DXGI_FORMAT_R32G32B32_FLOAT, 0, 20,
          D3D11_INPUT_PER_VERTEX_DATA, 0 },
};
```



3

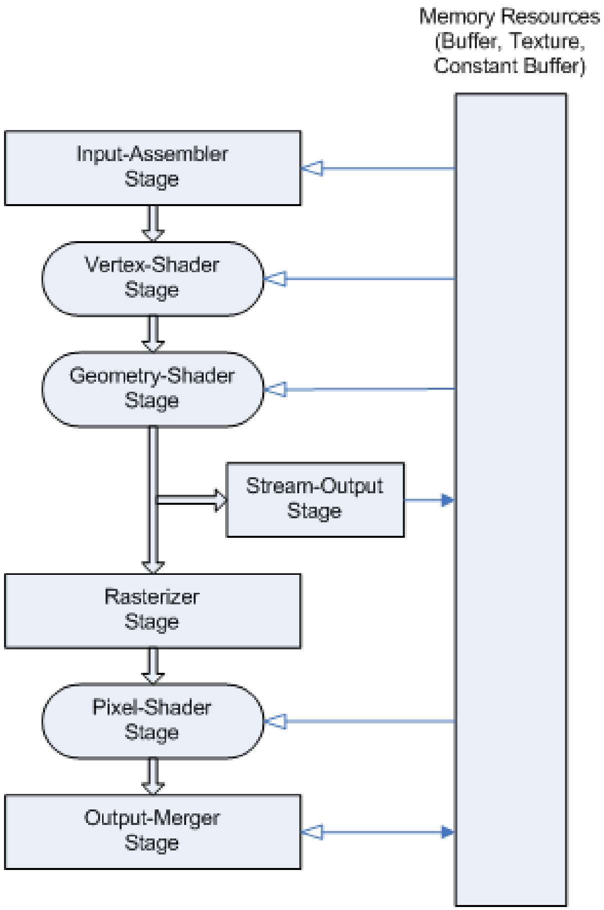
```
UINT stride = sizeof( SimpleVertex );
UINT offset = 0;
g_pd3dDevice->IASetVertexBuffers(
    0,                 // the first input slot for binding
    1,                 // the number of buffers in the array
    &g_pVertexBuffer,  // the array of vertex buffers
    &stride,           // array of stride values, one for each buffer
    &offset );         // array of offset values, one for each buffer

// Set the input layout
g_pd3dDevice->IASetInputLayout( g_pVertexLayout );
```

Vertex and Pixel Shaders

https://www.youtube.com/watch?v=TDZMSozKZ20&list=PLsPHRLf6UN4kkES0INlCfxDuf5abZW0OH

VERTEX SHADER



<u>Vertex-Shader Stage</u> - The vertex-shader stage processes vertices, typically performing operations such as transformations, skinning, and lighting.

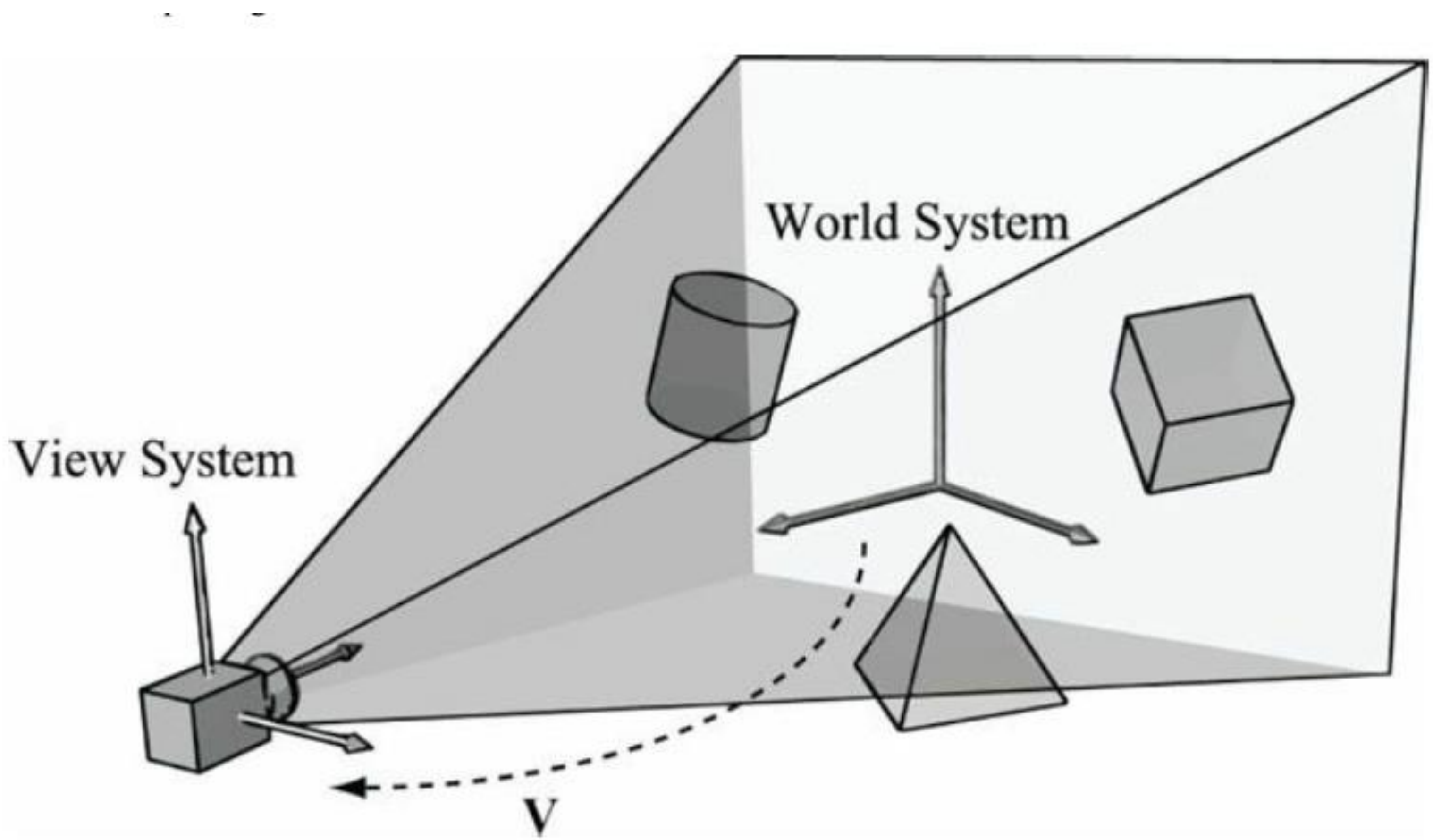
A vertex shader always takes a single input vertex and produces a single output vertex.

```
for(UINT i = 0; i < numVertices; ++i)
    outputVertex[i] = VertexShader (inputVertex[i]);
```

Typical use:

- Local space, World space, View space

LOCAL, WORLD, VIEW



Vertex Shader

```
/************* Vertex Shader *************/

VS_OUTPUT vertex_shader(VS_INPUT IN)
{
    VS_OUTPUT OUT = (VS_OUTPUT)0;

    OUT.Position = mul(IN.ObjectPosition, WorldViewProjection);
    OUT.TextureCoordinate = get_corrected_texture_coordinate(IN.TextureCoordinate);
    OUT.Normal = normalize(mul(float4(IN.Normal, 0), World).xyz);
    OUT.LightDirection = normalize(-LightDirection);

    float3 worldPosition = mul(IN.ObjectPosition, World).xyz;
    OUT.ViewDirection = normalize(CameraPosition - worldPosition);

    return OUT;
}
```

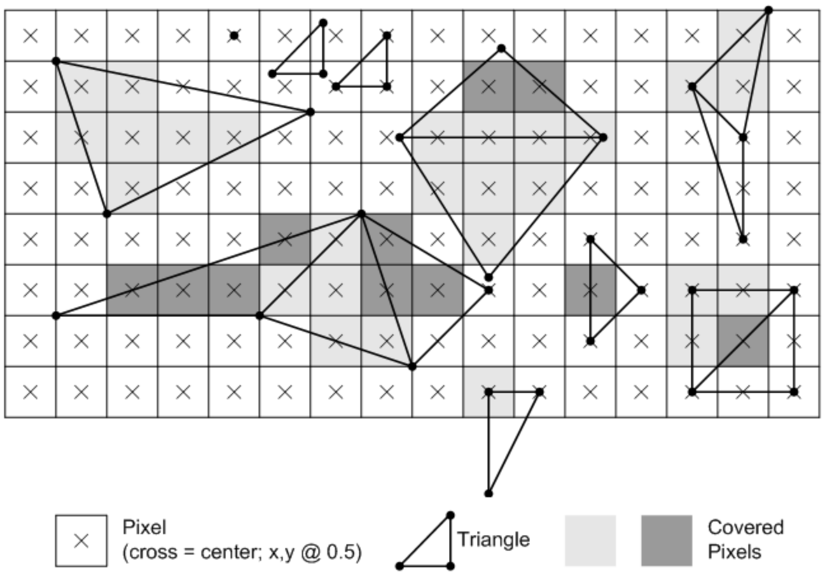
Data structures

```
/************* Data Structures *************/

struct VS_INPUT
{
    float4 ObjectPosition : POSITION;
    float2 TextureCoordinate : TEXCOORD;
    float3 Normal : NORMAL;
};

struct VS_OUTPUT
{
    float4 Position : SV_Position;
    float3 Normal : NORMAL;
    float2 TextureCoordinate : TEXCOORD0;
    float3 LightDirection : TEXCOORD1;
    float3 ViewDirection : TEXCOORD2;
};
```

RASTERIZER



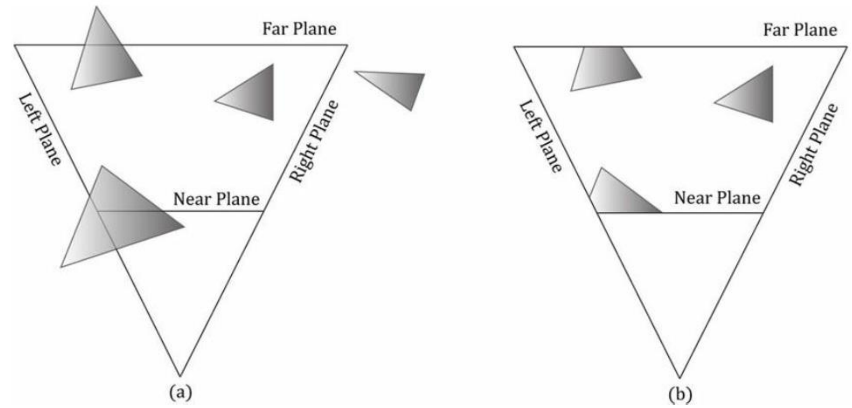
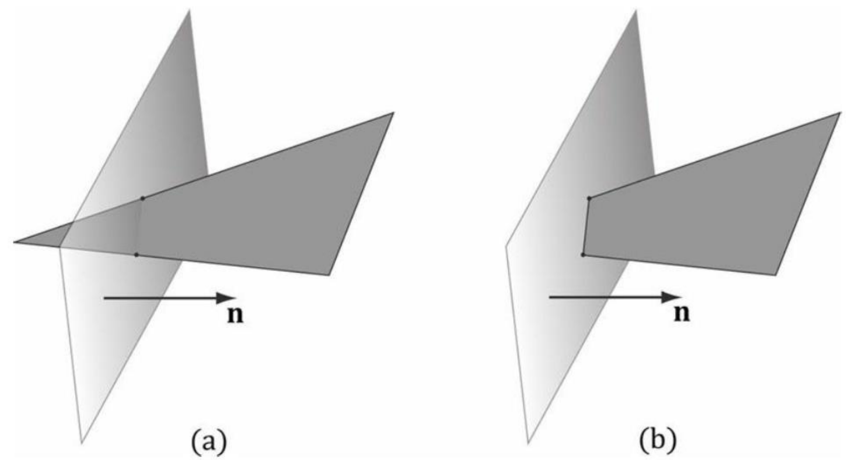


Figure 5.27. (a) Before clipping. (b) After clipping.



Pixel Shader

```
/************* Pixel Shader *************/

float4 pixel_shader(VS_OUTPUT IN) : SV_Target
{
    float4 OUT = (float4)0;

    float3 normal = normalize(IN.Normal);
    float3 lightDirection = normalize(IN.LightDirection);
    float3 viewDirection = normalize(IN.ViewDirection);
    float n_dot_l = dot(lightDirection, normal);

    float4 color = ColorTexture.Sample(ColorSampler, N.TextureCoordinate);
    float3 ambient = AmbientColor.rgb * AmbientColor.a * color.rgb;
    float3 diffuse = (float3)0;
    float3 specular = (float3)0;

    if (n_dot_l > 0)
    {
        diffuse = LightColor.rgb * LightColor.a * saturate(n_dot_l) * color.rgb;

        // R = 2 * (N.L) * N - L
        float3 reflectionVector = normalize(2 * n_dot_l * normal - lightDirection);

        // specular = R.V^n with gloss map in color texture's alpha channel
        specular = SpecularColor.rgb * SpecularColor.a * min(pow(saturate(dot(reflectionVector, viewDirection)), SpecularPower), color.w);
    }

    OUT.rgb = ambient + diffuse + specular;
    OUT.a = 1.0f;

    return OUT;
}
```

Output Merger Stage

Output Merger Stage - The output-merger (OM) stage generates the final rendered pixel color using

- a combination of pipeline state,
- the pixel data generated by the pixel shaders,
- the contents of the render targets, and
- the contents of the depth/stencil buffers.

The OM stage uses

- Depth and Stencil test
- Blending function

https://msdn.microsoft.com/en-us/library/windows/desktop/bb205120(v=vs.85).aspx

Shaders, shaders, shaders

- DirectX 9: Vertex Shader, Pixel Shader
- DirectX 10: Vertex Shader, Geometry Shader, Pixel Shader
- DirectX 11: Vertex Shader, Hull Shader, Domain Shader, Geometry Shader, Pixel Shader, Compute Shader

All shaders require the same basic functionality: Textures (or other data) and math operations.

HBO-ICT GAME DEVELOPMENT

SHADER MODEL VS API VERSION

Vertex shader comparison [edit]

Vertex shader version	VS 1.1[9]	VS 2.0[4][9]	VS 2.0a[4][9]	VS 3.0[5][9]	VS 4.0[6]	VS 4.1[10]	VS 5.0[8]
# of instruction slots	128	256	256	≥ 512	4096	4096	4096
Max # of instructions executed	Unknown	65536	65536	65536	65536	65536	65536
Instruction predication	No	No	Yes	Yes	Yes	Yes	Yes
Temp registers	12	12	13	32	4096	4096	4096
# constant registers	≥ 96	≥ 256	≥ 256	≥ 256	16×4096	16×4096	16×4096
Static flow control	???	Yes	Yes	Yes	Yes	Yes	Yes
Dynamic flow control	No	No	Yes	Yes	Yes	Yes	Yes
Dynamic flow control depth	No	No	24	24	Yes	Yes	Yes
Vertex texture fetch	No	No	No	Yes	Yes	Yes	Yes
# of texture samplers	N/A	N/A	N/A	4	128	128	128
Geometry instancing support	No	No	No	Yes	Yes	Yes	Yes
Bitwise operators	No	No	No	No	Yes	Yes	Yes
Native integers	No	No	No	No	Yes	Yes	Yes

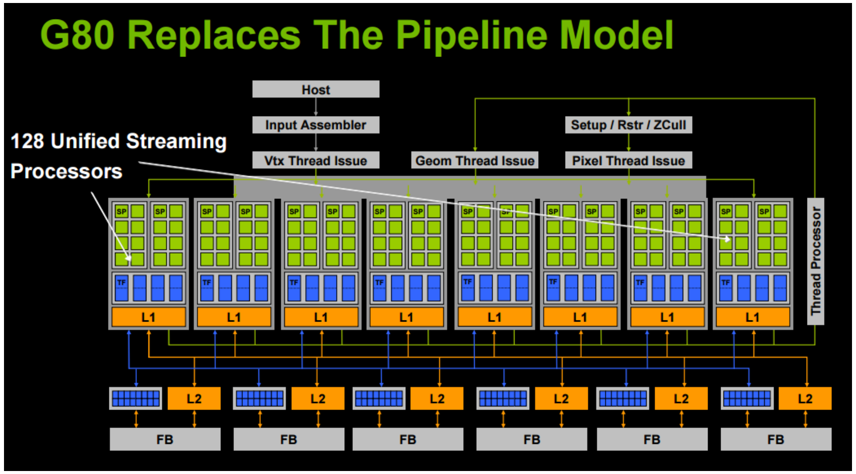
- **VS 2.0** = DirectX 9.0 original **Shader Model 2** specification.
- **VS 2.0a** = NVIDIA GeForce FX/PCX-optimized model, DirectX 9.0a.
- **VS 3.0** = **Shader Model 3.0**, DirectX 9.0c.
- **VS 4.0** = **Shader Model 4.0**, DirectX 10.
- **VS 4.1** = **Shader Model 4.1**, DirectX 10.1.
- **VS 5.0** = **Shader Model 5.0**, DirectX 11.

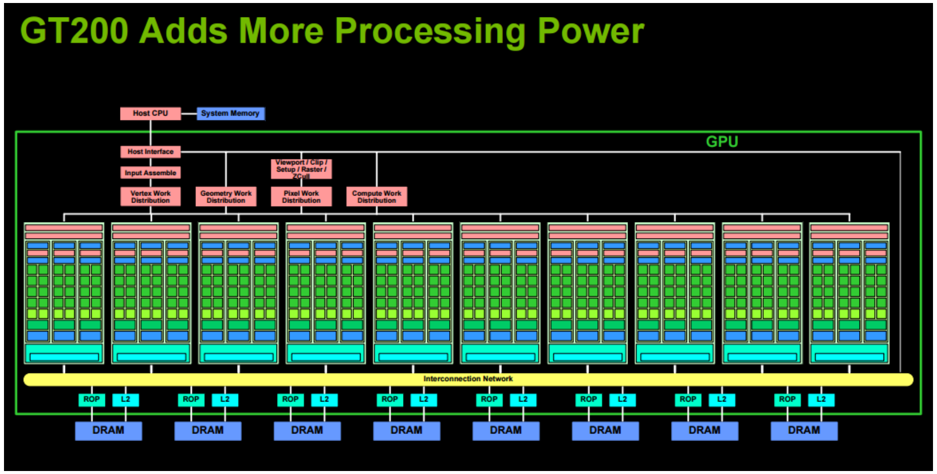
Pixel shader comparison [edit]

Pixel shader version	1.0 to 1.3[3]	1.4[3]	2.0[3][4]	2.0a[3][4]	2.0b[3][4]	3.0[3][5]	4.0[6]	4.1[7]	5.0[8]
Dependent texture limit	4	6	8	Unlimited	8	Unlimited	Unlimited	Unlimited	Unlimited
Texture instruction limit	4	6*2	32	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited	Unlimited
Position register	No	No	No	No	No	Yes	Yes	Yes	Yes
Instruction slots	8+4	8+4	32 + 64	512	512	≥ 512	≥ 65536	≥ 65536	≥ 65536
Executed instructions	8+4	6*2+8*2	32 + 64	512	512	65536	Unlimited	Unlimited	Unlimited
Texture indirections	4	4	4	Unlimited	4	Unlimited	Unlimited	Unlimited	Unlimited
Interpolated registers	2 + 8	2 + 8	2 + 8	2 + 8	2 + 8	10	32	32	32
Instruction predication	No	No	No	Yes	No	Yes	No	No	No
Index input registers	No	No	No	No	No	Yes	Yes	Yes	Yes
Temp registers	2	6	12 to 32	22	32	32	4096	4096	4096
Constant registers	8	8	32	32	32	224	16×4096	16×4096	16×4096
Arbitrary swizzling	No	No	No	Yes	No	Yes	Yes	Yes	Yes
Gradient instructions	No	No	No	Yes	No	Yes	Yes	Yes	Yes
Loop count register	No	No	No	No	No	Yes	Yes	Yes	Yes
Face register (2-sided lighting)	No	No	No	No	No	Yes	Yes	Yes	Yes
Dynamic flow control	No	No	No	No	No	24	Yes	Yes	Yes
Bitwise Operators	No	No	No	No	No	No	Yes	Yes	Yes
Native Integers	No	No	No	No	No	No	Yes	Yes	Yes

- **PS 2.0** = DirectX 9.0 original **Shader Model 2** specification.
- **PS 2.0a** = NVIDIA GeForce FX/PCX-optimized model, DirectX 9.0a.
- **PS 2.0b** = ATI Radeon X700, X800, X850, FireGL X3-256, V5000, V5100 and V7100 shader model, DirectX 9.0b.
- **PS 3.0** = **Shader Model 3.0**, DirectX 9.0c.
- **PS 4.0** = **Shader Model 4.0**, DirectX 10.
- **PS 4.1** = **Shader Model 4.1**, DirectX 10.1.
- **PS 5.0** = **Shader Model 5.0**, DirectX 11.

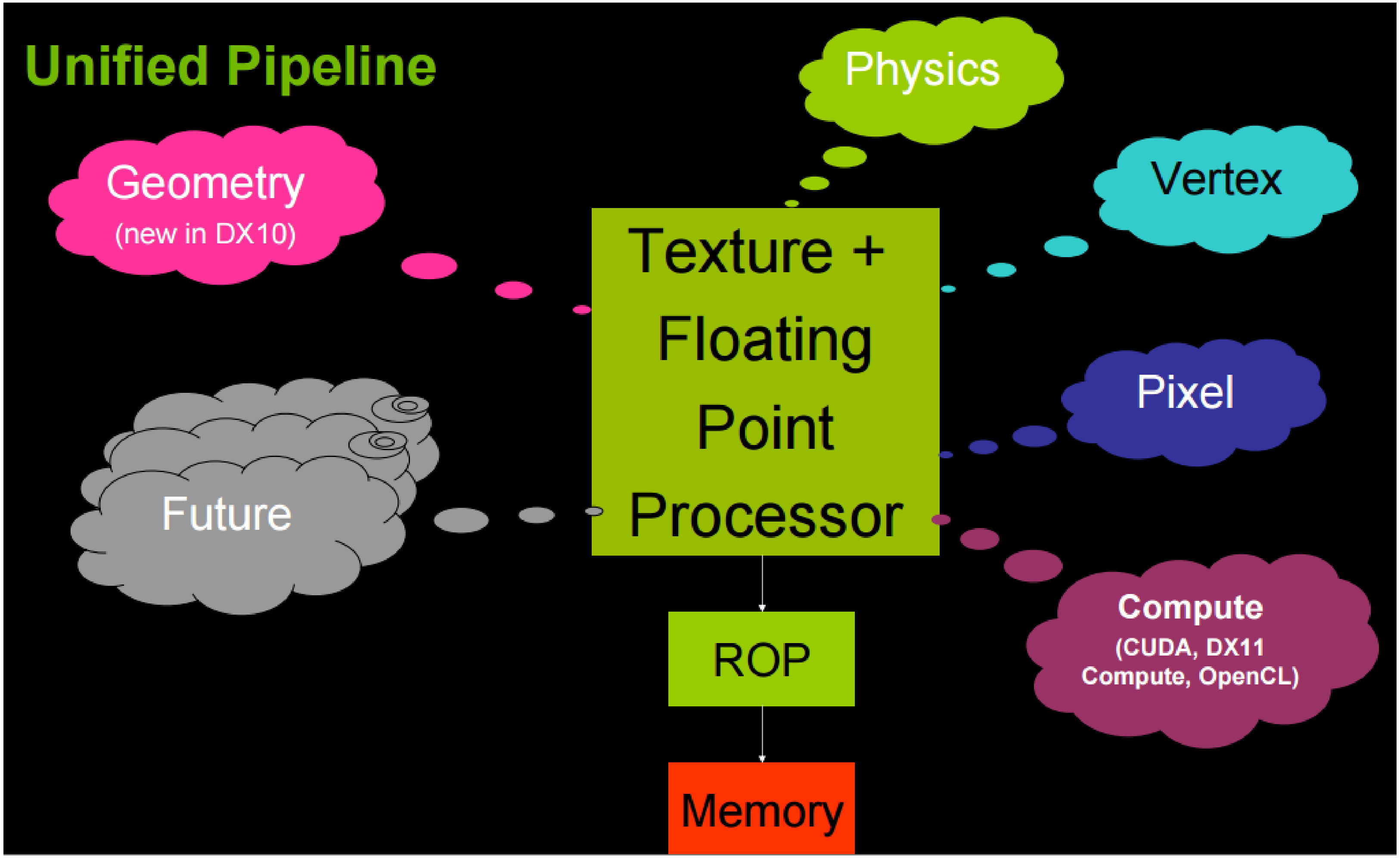
HARDWARE



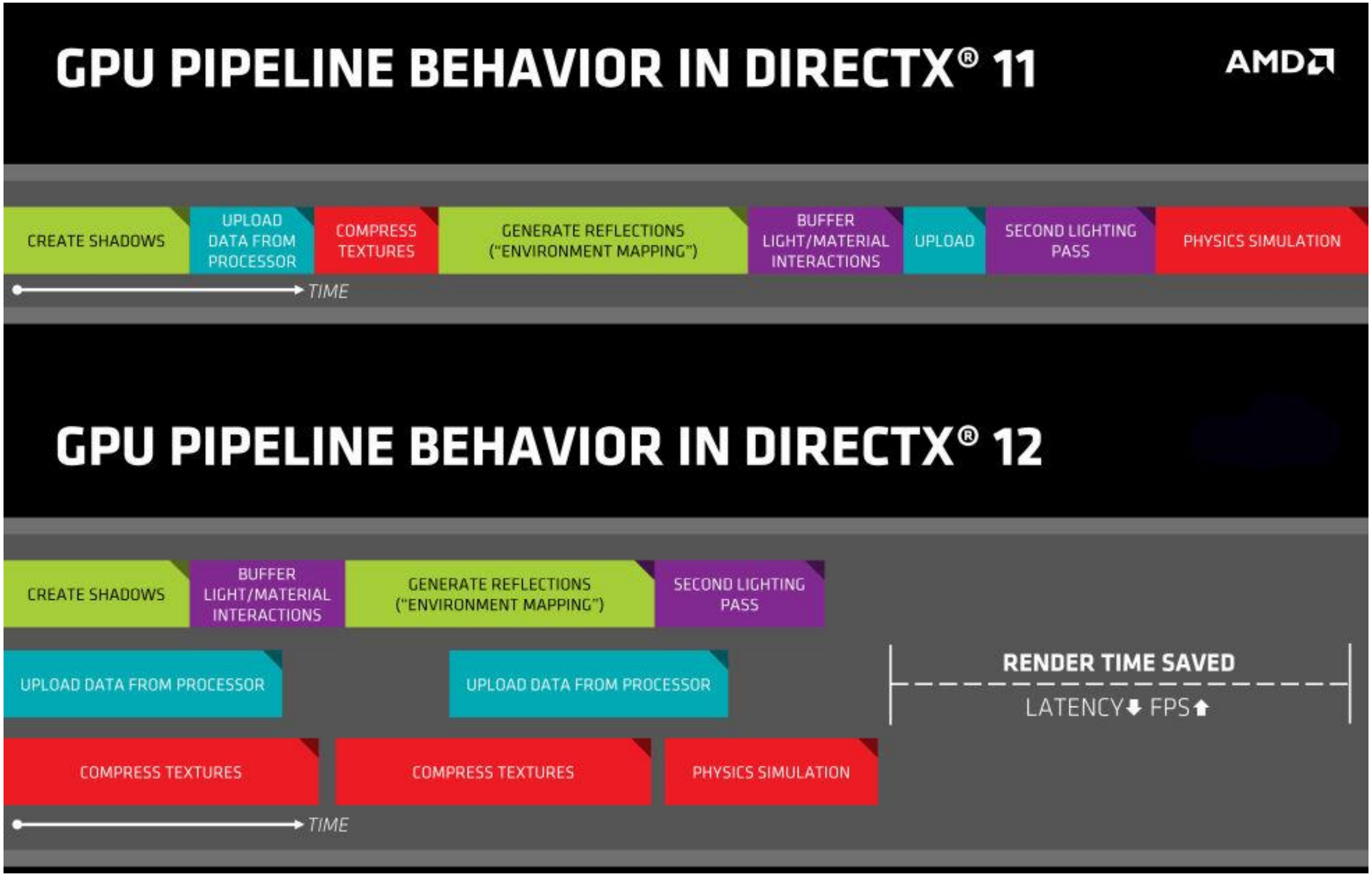


GTX TITAN Z GPU Engine Specs:	
CUDA Cores	5760
Base Clock (MHz)	705
Boost Clock (MHz)	876
Texture Fill Rate (GigaTexels/sec)	338
GTX TITAN Z Memory Specs:	
Memory Clock	7.0 Gbps
Standard Memory Config	12 GB
Memory Interface	GDDR5
Memory Interface Width	768-bit (384-bit per GPU)
Memory Bandwidth (GB/sec)	672
GTX TITAN Z Support:	
OpenGL	4.4
Bus Support	PCI Express 3.0
Certified for Windows 8, Windows 7, or Windows Vista	Yes

Unified pipeline, GPGPU



DIRECTX 12



GPU PIPELINE BEHAVIOR IN DIRECTX® 11
AMD
CREATE SHADOWS
UPLOAD DATA FROM PROCESSOR
COMPRESS TEXTURES
GENERATE REFLECTIONS ("ENVIRONMENT MAPPING")
BUFFER LIGHT/MATERIAL INTERACTIONS
UPLOAD
SECOND LIGHTING PASS
PHYSICS SIMULATION
TIME
GPU PIPELINE BEHAVIOR IN DIRECTX® 12
CREATE SHADOWS
BUFFER LIGHT/MATERIAL INTERACTIONS
GENERATE REFLECTIONS ("ENVIRONMENT MAPPING")
SECOND LIGHTING PASS
UPLOAD DATA FROM PROCESSOR
UPLOAD DATA FROM PROCESSOR
RENDER TIME SAVED
LATENCY FPS
COMPRESS TEXTURES
COMPRESS TEXTURES
PHYSICS SIMULATION
TIME

HBO-ICT GAME DEVELOPMENT

GAME ENGINE VS RENDER ENGINE

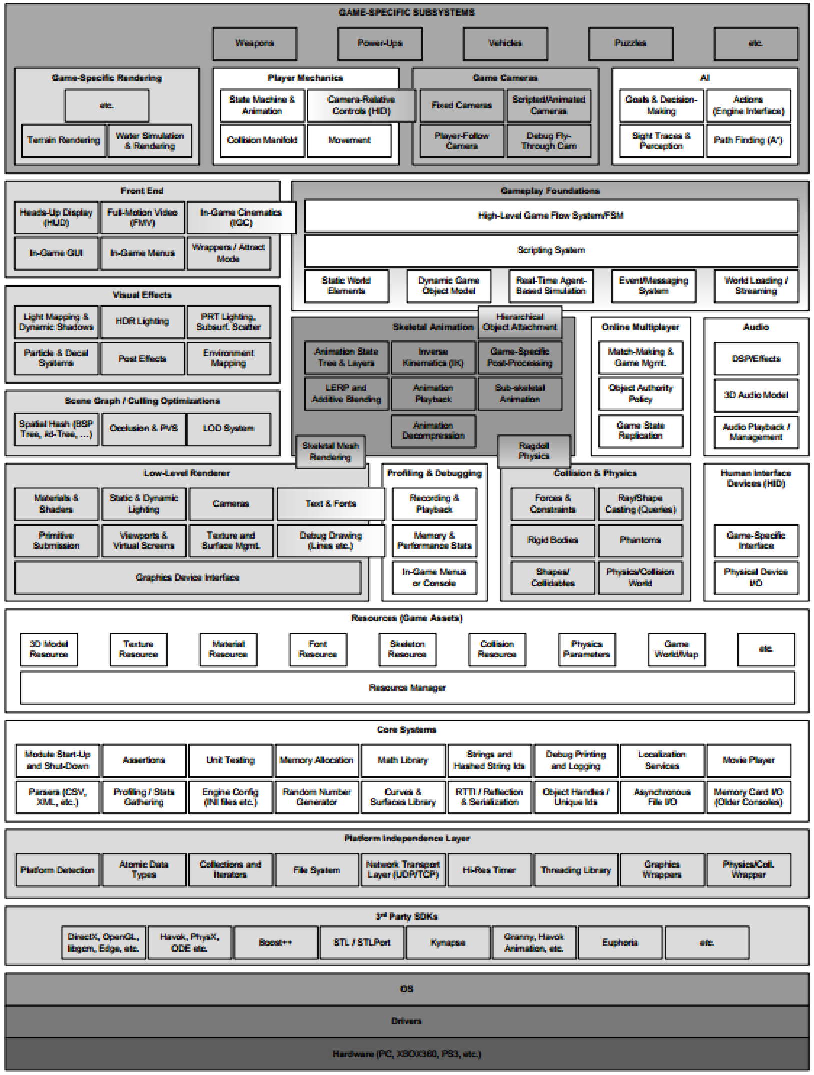


Figure 1.15. Runtime game engine architecture.

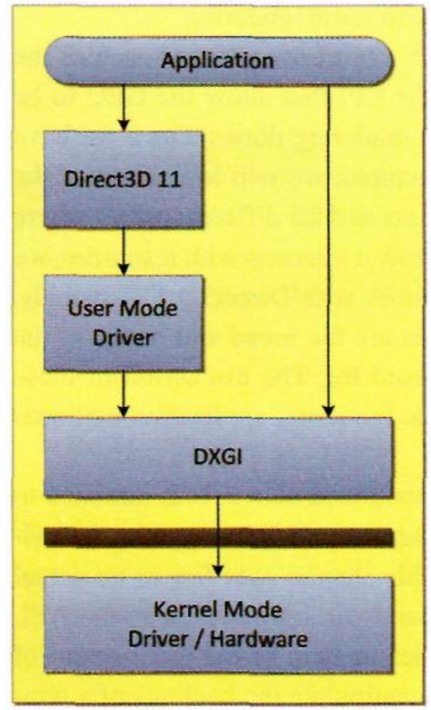


Figure 1.1. The various components of the graphics subsystems used with Direct3D

- Render engine subsystem
- DirectX 11 on windows 7/8/10 (?)
- C++