Introduction to DirectX Programming

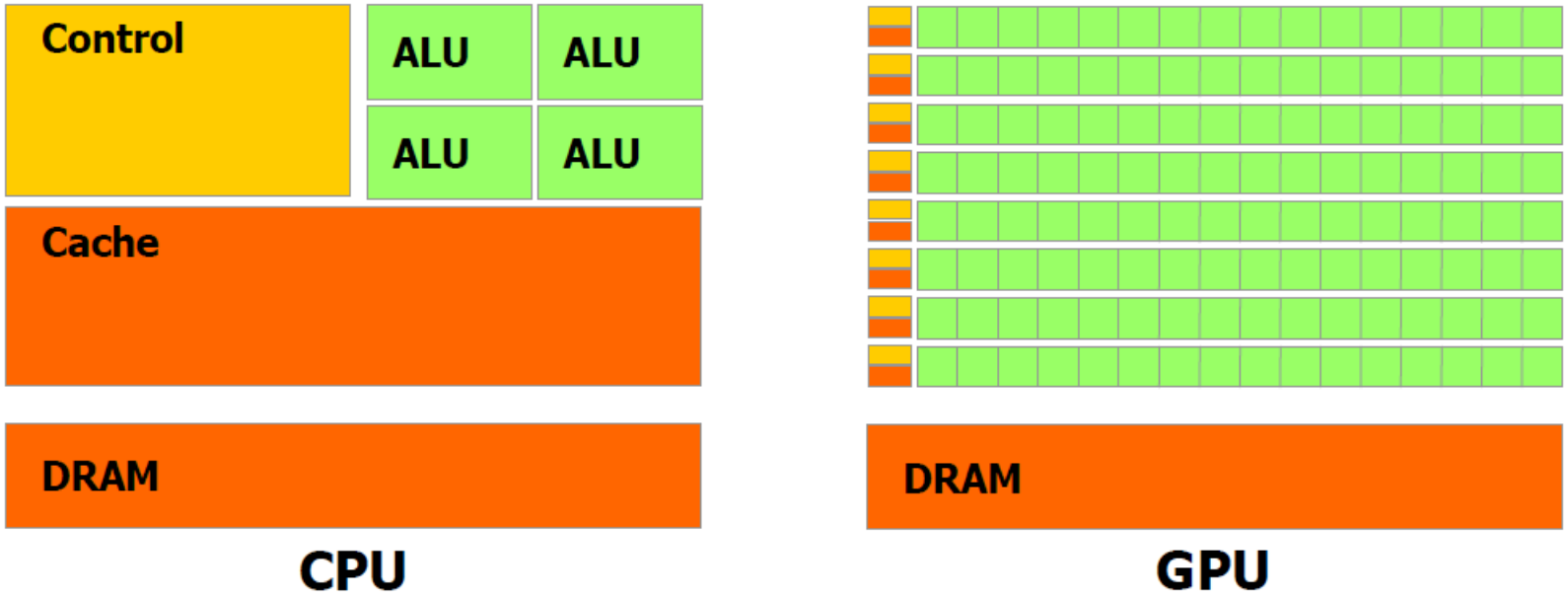
Direct3D Pipeline

Contents

- GPU programming model
- Introduction to DirectX
  - Direct3D
  - DXGI
  - HLSL
  - COM essentials
- Direct3D pipeline

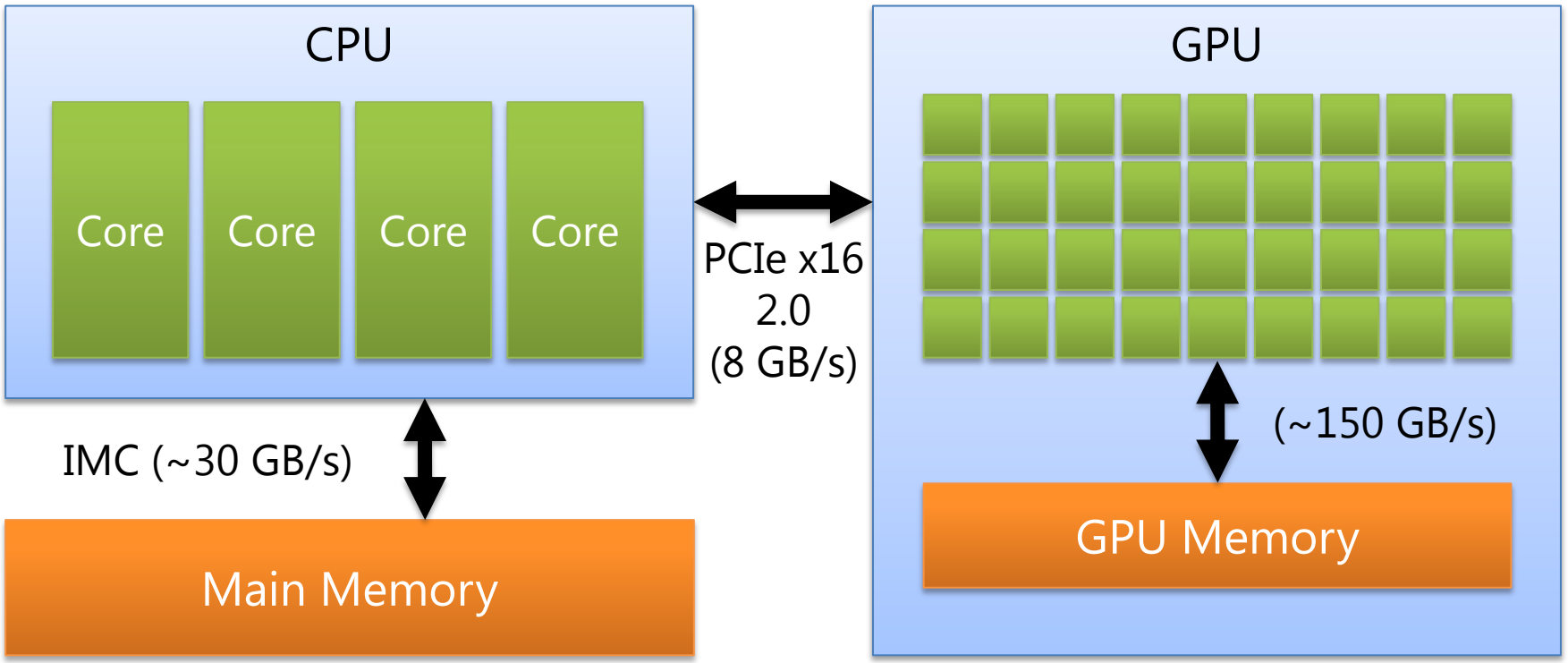
GPU Architecture

- Well-suited for data-parallel computations
- Lower requirement for sophisticated flow control



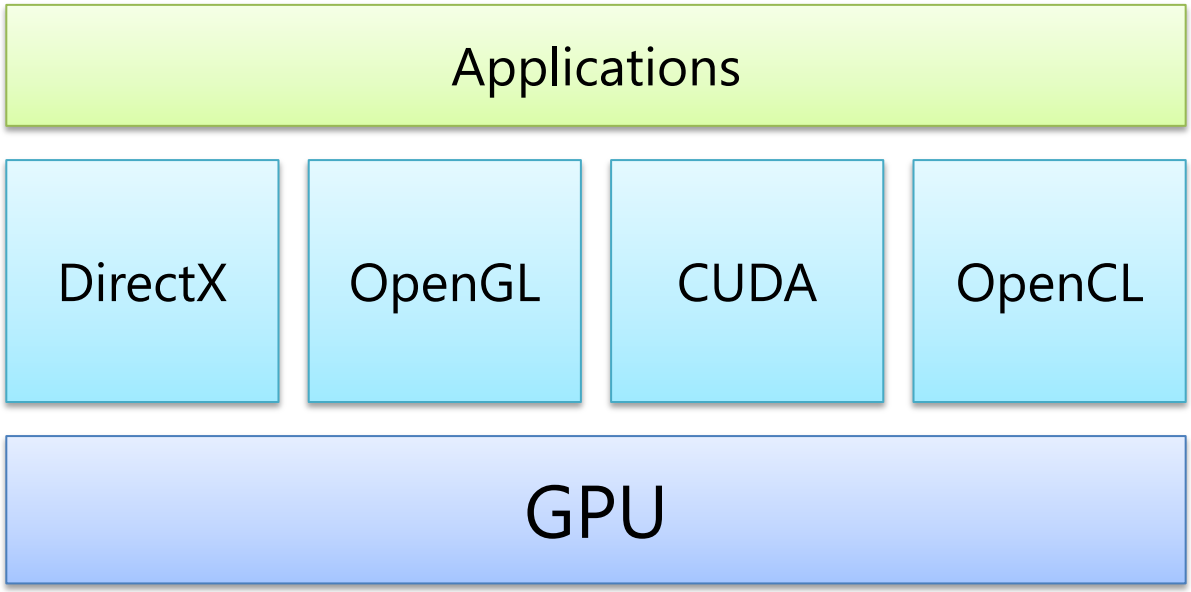
CPU-GPU Relationship

- GPU: coprocessor with its own memory



Programming with GPU

- Various APIs and technologies
  - Graphics programming
  - General-purpose GPU programming



Introduction to DirectX

- Set of low-level APIs for creating high-performance multimedia applications, on Microsoft platforms (Windows, Xbox, …)
- Components

2-D and 3-D graphics	Direct3D
Sound	XAudio2, X3DAudio, XACT
Input	XInput

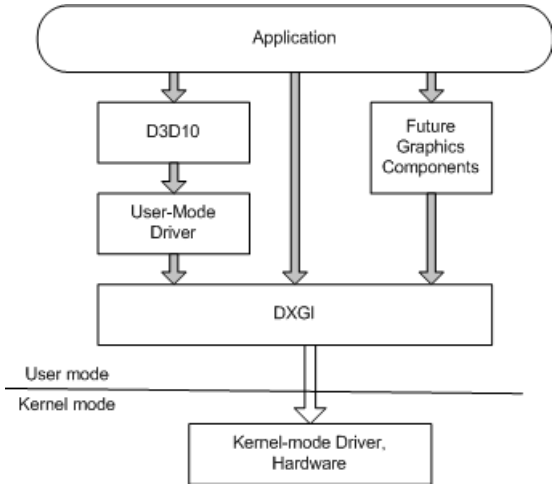
- DirectX SDK (Software Development Kit)
  - Libraries, header files, utilities, sample codes and docs

Direct3D

- API for rendering 2-D and 3-D graphics
- Hardware acceleration if available
- Advanced graphics capabilities
  - Graphics pipeline, z-buffering, anti-aliasing, alpha blending, mipmapping, …
- Abstractions
  - Pipeline stages, device, resource, swap chain, …
- API as set of COM-compliant objects

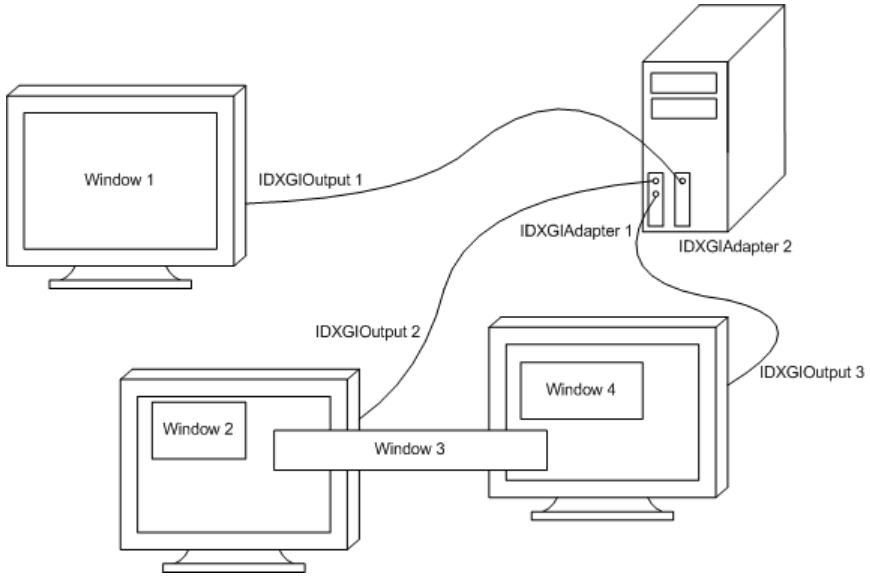
DXGI

- DirectX Graphics Infrastructure
  - Manages low-level tasks independent of DirectX graphics runtime
  - Common framework for future graphics components
- Supported tasks
  - Enumerating adapters
  - Managing swap chain
  - Handling window resizing
  - Choosing DXGI output and size
  - Debugging in full-screen mode



Adapter

- Abstraction of hardware/software capability used by graphics applications
  - Hardware: video card, …
  - Software: Direct3D reference device, …



Swap Chain

- Encapsulates two or more buffers used for rendering and display
- Front buffer
  - Presented to display device
- Back buffer
  - Render target
- Presents back buffer by swapping two buffers
- Properties
  - Size of render area, display refresh rate, display mode, surface format, …

HLSL

- High Level Shading Language for DirectX
- For writing codes for programmable shaders in Direct3D pipeline
- Implements series of shader models

Shader Model	Shader Profiles	Direct3D support
Shader model 1	vs_1_1	Direct3D 9
Shader model 2	ps_2_0, ps_2_x, vs_2_0, vs_2_x	
Shader model 3	ps_3_0, vs_3_0	
Shader model 4	gs_4_0, ps_4_0, vs_4_0, gs_4_1, ps_4_1, vs_4_1	Direct3D 10
Shader model 5	cs_4_0, cs_4_1, cs_5_0, ds_5_0, gs_5_0, hs_5_0, ps_5_0, vs_5_0	Direct3D 11

Compiling HLSL

- Shader profile
  - Target for compiling shader

a. Compile at runtime
   - D3DX11CompileFromFile function to compile from HLSL file
   - D3DCompile function to compile HLSL code in memory

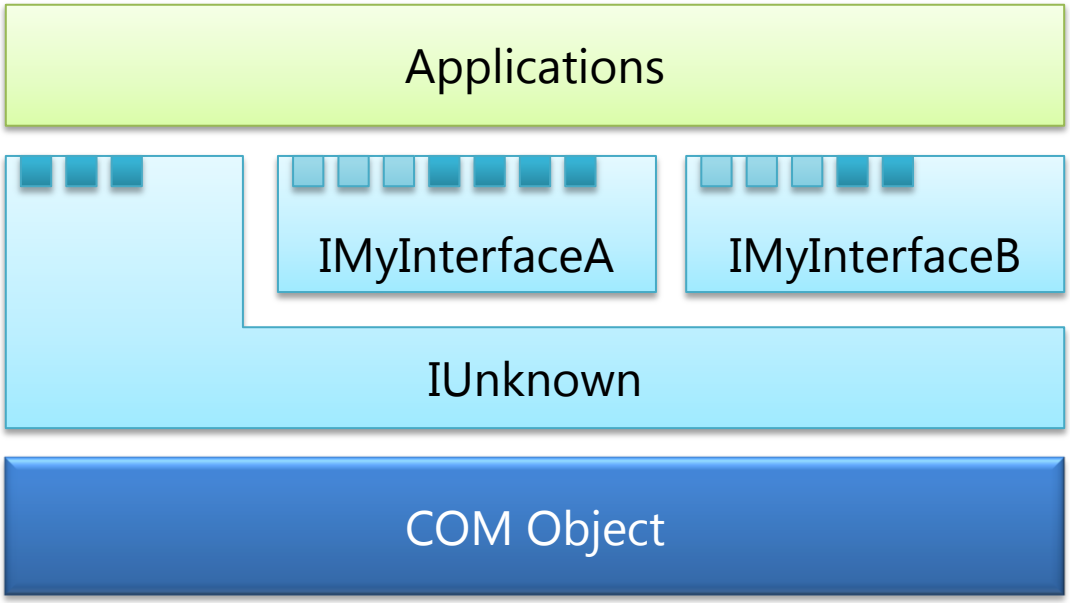
b. Compile offline
   - FXC: shader-compiler tool (effect-compiler tool)

COM Essentials

- Component Object Model
  - Binary standard
  - Platform-independent
  - Distributed
  - Object-oriented
- COM-based technologies
  - OLE
  - ActiveX
  - Many components in Windows

COM Interfaces

- Object exposes set of interfaces
- Interface consists of group of related methods
- Application (client) use object through interfaces



IUnknown Interface

- Every object implements IUnknown interface

Method	Description
QueryInterface	Retrieves pointers to the supported interfaces on object
AddRef	Increments reference count for interface on object
Release	Decrements reference count for interface on object

COM in C++

Interface	Abstract class
Interface method	Pure virtual function
Interface inheritance	Class derivation – Interface inheritance – No code reuse
Interface pointer	Pointer to class

```
// unknwn.h

class IUnknown
{
public:
    virtual HRESULT QueryInterface (
        /* [in] */ REFIID riid,
        /* [out] */ void** ppvObject ) = 0;
    virtual ULONG AddRef() = 0;
    virtual ULONG Release() = 0;
};
```

Using COM Objects

Users

- Initialize COM library
- Create object and obtain pointer of interface
- Use interface methods
- Release object

Developers

- Publish interfaces
  - Documents describing behavior of interface methods
- Provide interface implementation (in binary form)

Example in C++

```
IMyInterface* my;
CreateMyObject(&my);

HRESULT hr;
hr = my->MyMethod(i);

my->Release();
```

```
bool CreateMyObject(IMyInterface** pp);

class IMyInterface
{
public:
    virtual HRESULT MyMethod(int i) = 0;
};
```

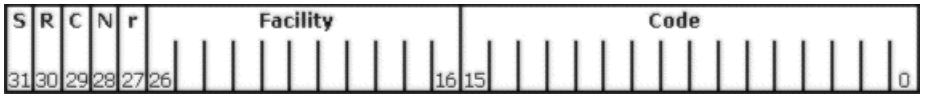
GUID

- Globally unique identifier
  - Unique reference number used as identifier
  - Implements universally unique identifier (UUID) standard
  - Represented by 32-character hexademical string (e.g., {21EC2020-3AEA-1069-A2DD-08002B30309D})
  - Usually stored as 128-bit integer
- Interface identifier (IID)
- Class identifier (CLSID)

Error Handling in COM

- HRESULT
  - 32-bit value used to describe the status after operation

Internal structure



Return value/code	Description
0x00000000 S_OK	Operation successful
0x80004001 E_NOIMPL	Not implemented
0x80004002 E_NOINTERFACE	Interface not supported
0x80004004 E_ABORT	Operation aborted
0x80004005 E_FAIL	Unspecified failure
0x80070057 E_INVALIDARG	One or more arguments are invalid

Macro	Description
SUCCEEDED(hr)	TRUE if *hr* represents success status value; otherwise, FALSE
FAILED(hr)	TRUE if *hr* represents falied status value; otherwise, FALSE

Direct3D Return Codes

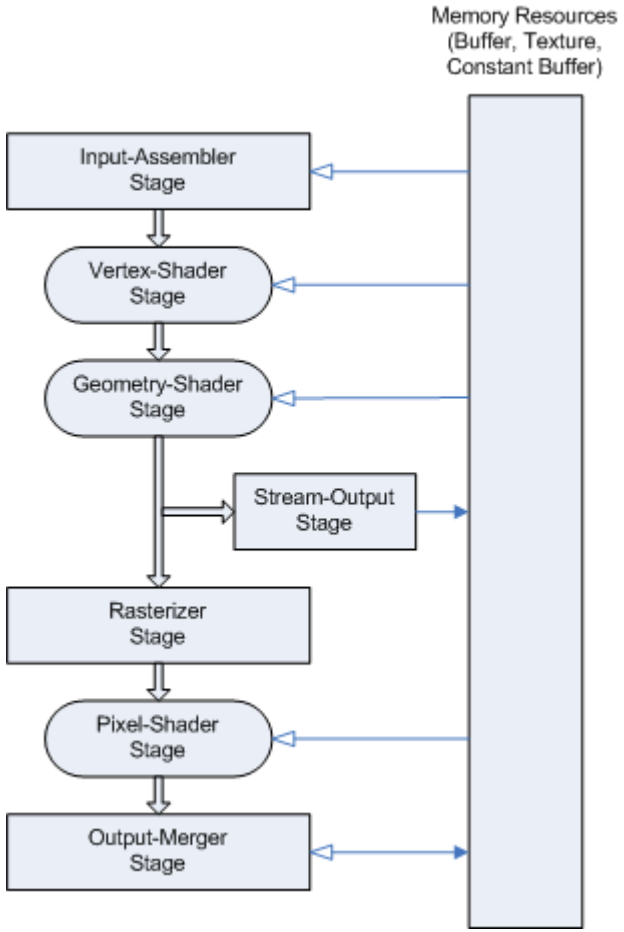
HRESULT	Description
S_OK	No error occurred.
S_FALSE	Alternate success value, indicating a successful but nonstandard completion (the precise meaning depends on context).
E_FAIL	Attempted to create a device with the debug layer enabled and the layer is not installed.
E_OUTOFMEMORY	Direct3D could not allocate sufficient memory to complete the call.
E_INVALIDARG	An invalid parameter was passed to the returning function.
D3DERR_INVALIDCALL	The method call is invalid. For example, a method's parameter may not be a valid pointer.
D3D11_ERROR_FILE_NOT_FOUND	The file was not found.
D3D11_ERROR_TOO_MANY_UNIQUE_STATE_OBJECTS	There are too many unique instances of a particular type of state object.
D3D11_ERROR_TOO_MANY_UNIQUE_VIEW_OBJECTS	There are too many unique instances of a particular type of view object.

(not exhaustive)

Direct3D Pipeline

- Stages
  - Configured using API
- Shader stages
  - Programmable using HLSL

Stage	Abbr.	Description
Input-assembler	IA	Supplies data to pipeline
Vertex-shader	VS	Processes vertices
Rasterizer	RS	Converts vector data into pixels
Pixel-shader	PS	Generates per-pixel data
Output-merger	OM	Generates final pipeline result



(Note: Some stages are omitted for simplicity)

Input-Assembler Stage

- Primitive
  - Composed of one or more vertices
  - Point, line and triangle
  - Primitive types (or topologies)
    - Point list, line list, line strip, triangle list, triangle strip, …
- Reads input data from user-filled buffers and assembles data into primitives
- Attaches system-generated values
  - VertexID, PrimitiveID, InstanceID, …

Shader Stages

- Vertex/hull/domain/geometry/pixel-shader and compute-shader
- Built on common shader core
- Programmable with HLSL

Classification	**Methods**
Creating shader object	ID3D11Device::Create[Vertex]Shader
Setting shader to use	ID3D11DeviceContext::[VS]SetShader
Setting up each shader stage	ID3D11DeviceContext::[VS]SetConstantBuffers ID3D11DeviceContext::[VS]SetShaderResources ID3D11DeviceContext::[VS]SetSamplers ID3D11DeviceContext::CSSetUnorderedAccessViews

Note: [VS] means one of VS, HS, DS, GS, PS and CS, and
[Vertex] one of Vertex, Hull, Domain, Geometry, Pixel and Compute

Vertex-Shader Stage

- Processes each vertex of primitives from IA
  - Performs per-vertex operations (transformations, skinning, morphing, per-vertex lighting, …)
- Single input vertex, single output vertex
  - Each vertex data can contain up to 16 vectors
  - Always run on all vertices
- Some texture methods available
  - Load, Sample, SampleCmpLevelZero, SampleGrad, …
- Mandatory stage

Rasterizer Stage

- Converts each primitive into pixels
  - Interpolating per-vertex values across primitive
- Performed operations
  - Clipping vertices to view frustum
  - Culling out back faces
  - Performing divide by z to provide perspective (input vertex positions assumed to be in homogeneous clip-space)
  - Mapping primitives to 2-D viewport
  - Determining how to invoke pixel shader

Pixel-Shader Stage

- Enables rich shading techniques and more
  - Per-pixel lighting, post-processing, ray-casting, …
- Processes each pixel from rasterizer stage
  - Interpolated per-pixel values as input
- Generated output
  - One or more colors (to be written to render targets)
  - Optional depth value
    (replacing the one interpolated by rasterizer)
- All texture methods available

Output-Merger Stage

- Generates final rendered pixel color
- Combines relevant information
  - Pipeline state, pixel data from PS, contents of render targets, contents of depth/stencil buffers, …
- Final step in pipeline
  - Determines visibility of pixels (depth-stencil testing)
  - Blends final pixel colors