COMP30019 Graphics & Interaction

Lecture 3

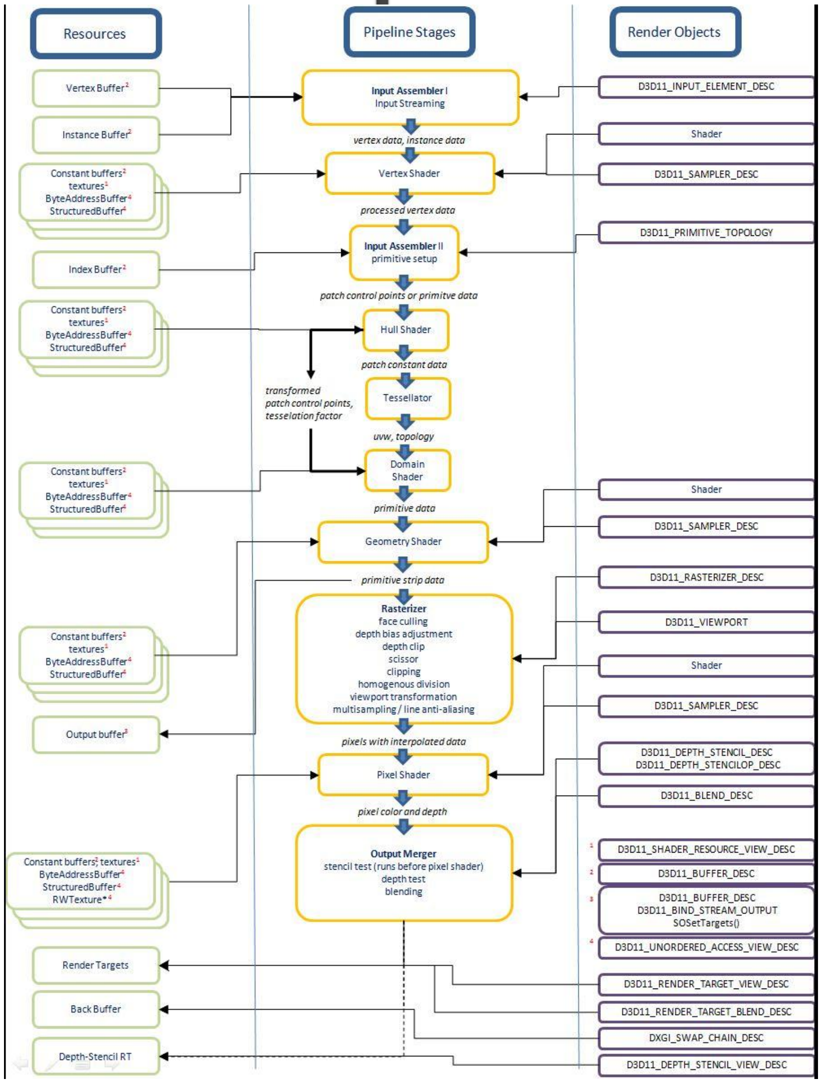
Graphics Programming

- Game engines, and other graphics programs, generally use either Direct3D (Windows) or OpenGL (most other platforms)
- Modern PC graphics cards will support some version of both APIs
- Game engines (like Unity) build upon these APIs to make development easier

Pipelining

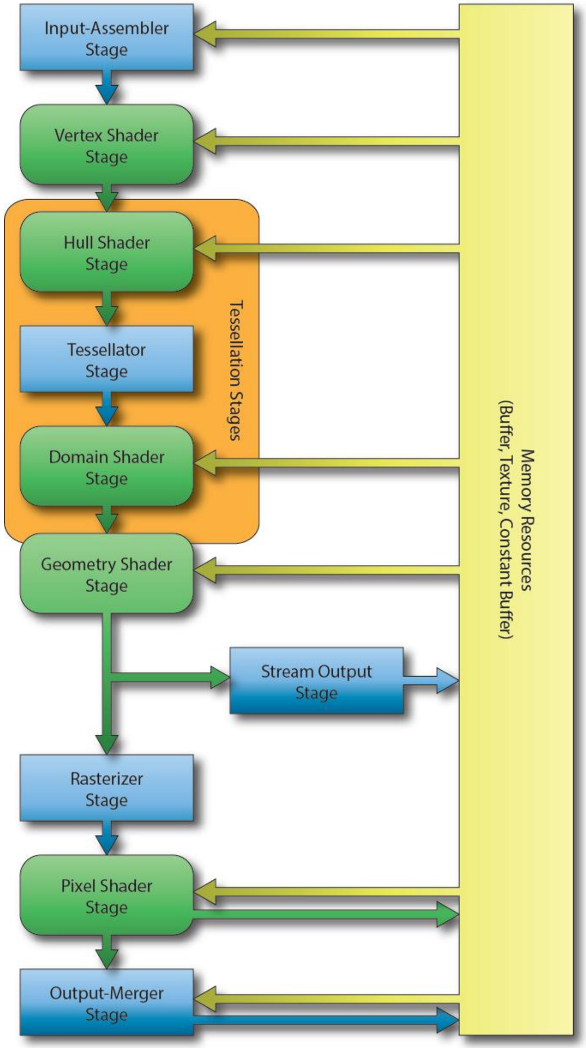
- Both OpenGL and Direct3D operate a *pipeline*, consisting of several different stages
- This allows the programmer to perform a number of different operations on the input data, and provides greater efficiency
- There are some differences between the OpenGL and Direct3D pipelines
- Will focus mainly on Direct3D pipeline

Direct3D 11 Pipeline



Source: Unity

Direct3D 11 Pipeline

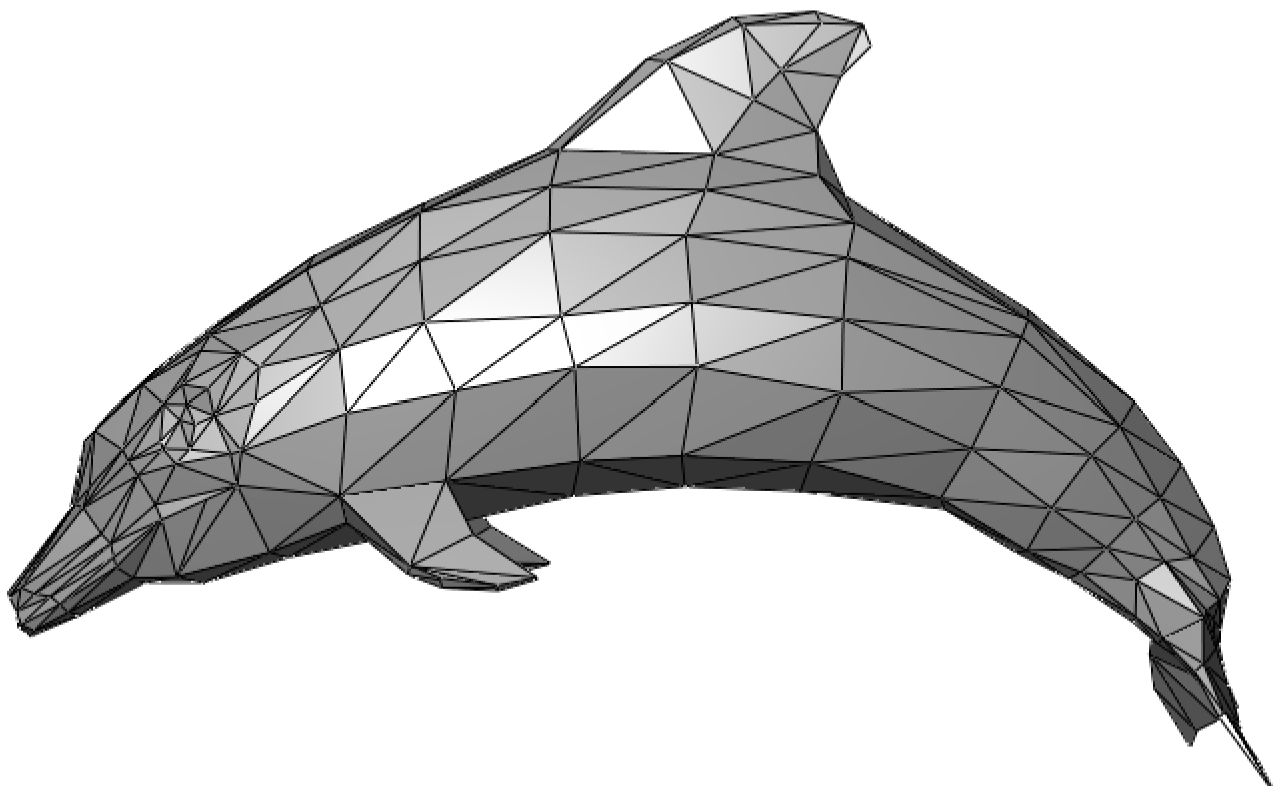


Source: 3dgep.com

Representing Objects

- For efficiency, the graphics card will render objects as triangles
- Any polyhedron can be represented by triangles
- Other 3D shapes can be approximated by triangles

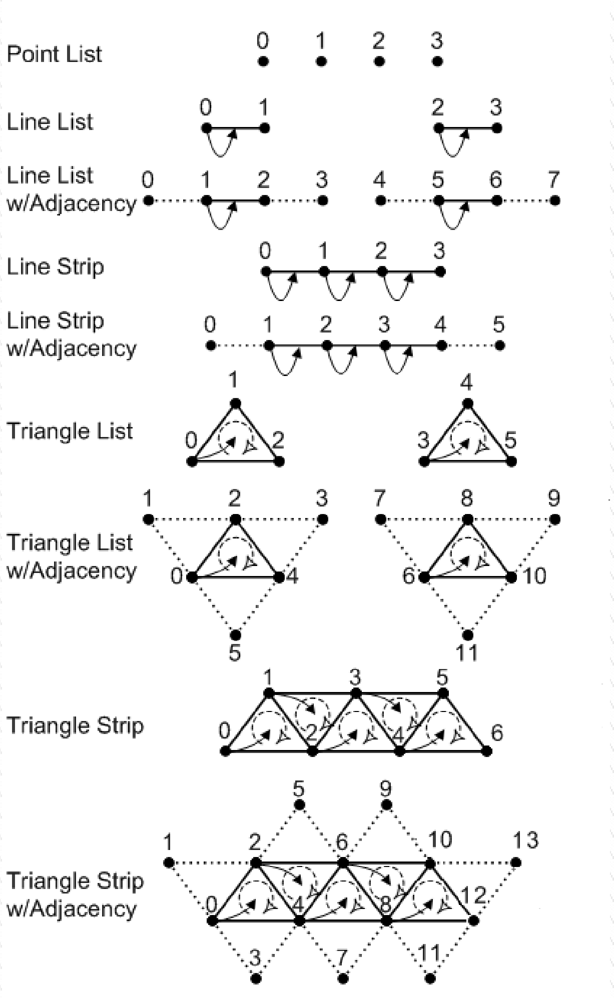
A Dolphin



Source: Wikipedia

Input Assembler

- Reads data from our buffers into a primitive format that can be used by the other stages of the pipeline
- We mainly use Triangle Lists

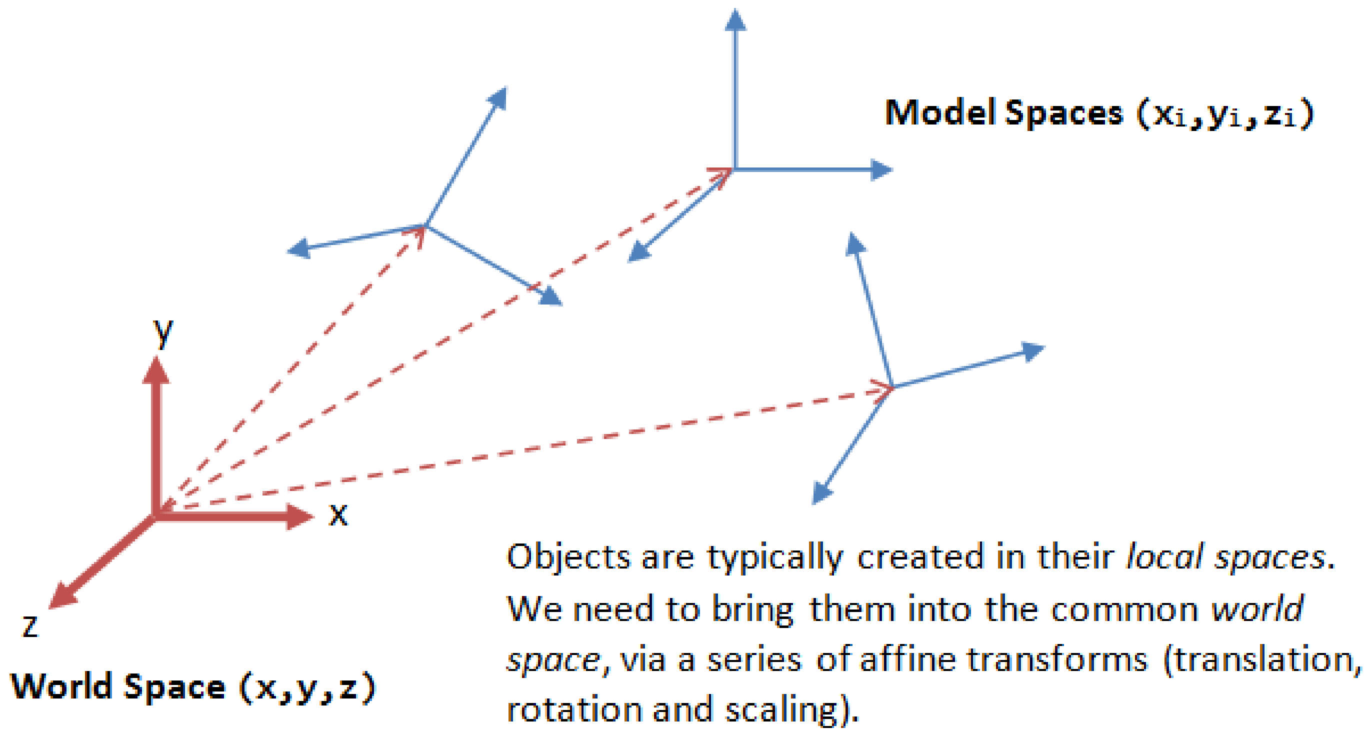


D3D11 Primitive Types
Source: Microsoft

Vertex Shader

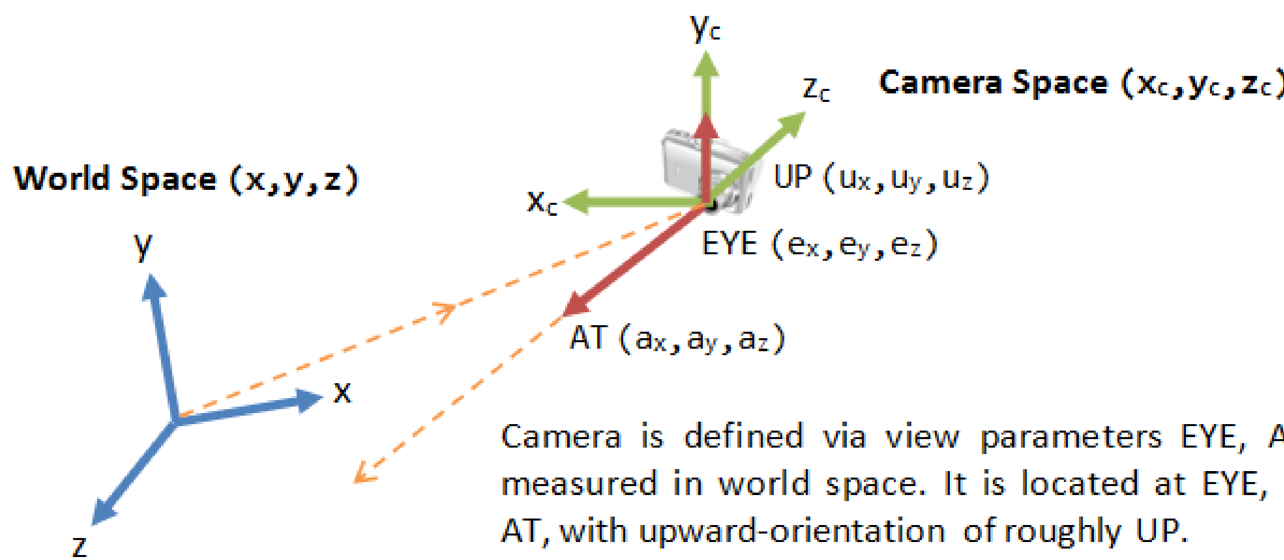
- Performs operations on individual vertices received from the Input Assembler stage
- This will typically include transformations
- May also include per-vertex lighting

Vertex Shader Transformations



Source: ntu.edu.sg

Vertex Shader Transformations

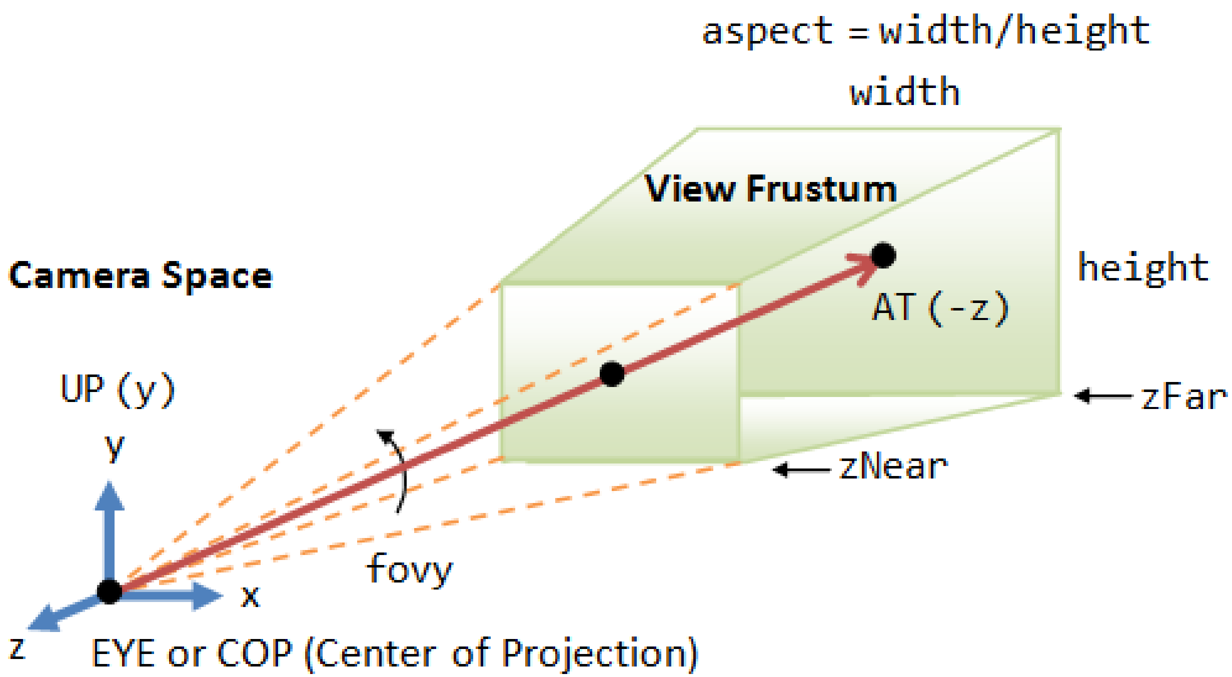


Camera is defined via view parameters EYE, AT and UP, measured in world space. It is located at EYE, pointing at AT, with upward-orientation of roughly UP.

In the Camera space, camera is located at origin, pointing at $-z_c$, with upward-orientation of y_c.
z_c is opposite of AT, y_c is roughly UP.

Source: ntu.edu.sg

Vertex Shader Transformations

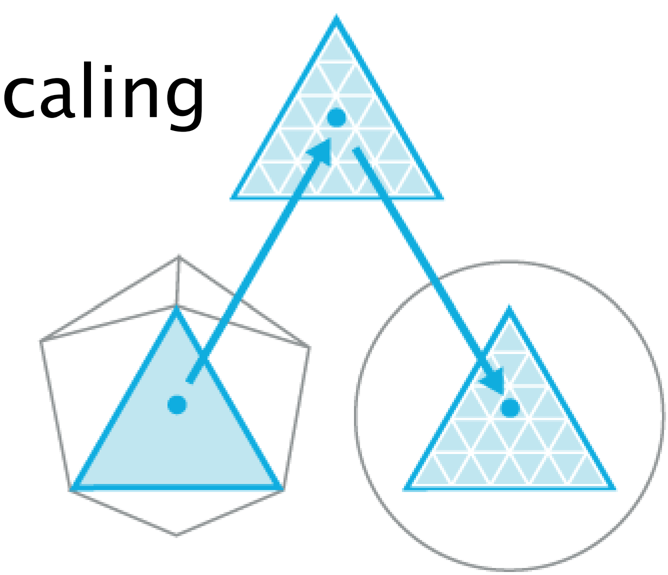


Perspective Projection: The camera's view frustum is specified via 4 view parameters: fovy, aspect, zNear and zFar.

Source: ntu.edu.sg

Tessellation Stages

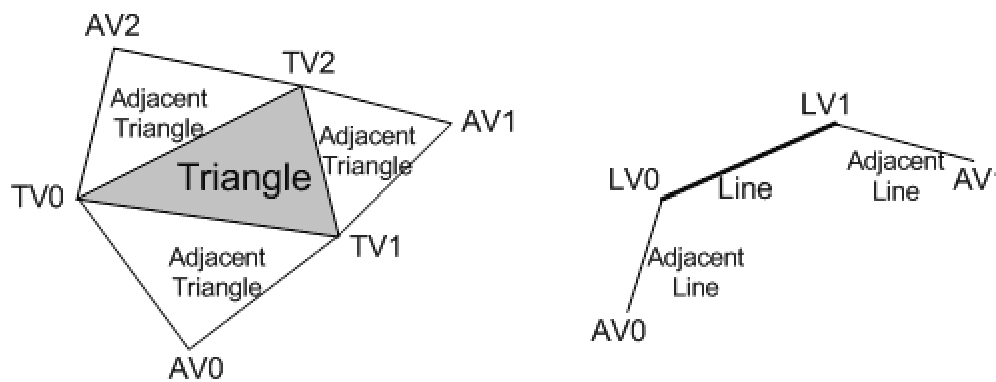
- Optional Stages, added with Direct3D 11
- These stages allow us to generate additional vertices within the GPU
- Can take a lower detail model and render in higher detail
- Can perform level of detail scaling



Source: Microsoft

Geometry Shader

- Optional Stage, added with Direct3D 10
- Operates on an entire primitive (e.g. triangle)
- Can perform a number of algorithms, e.g. dynamically calculating normals, particle systems, shadow volume generation



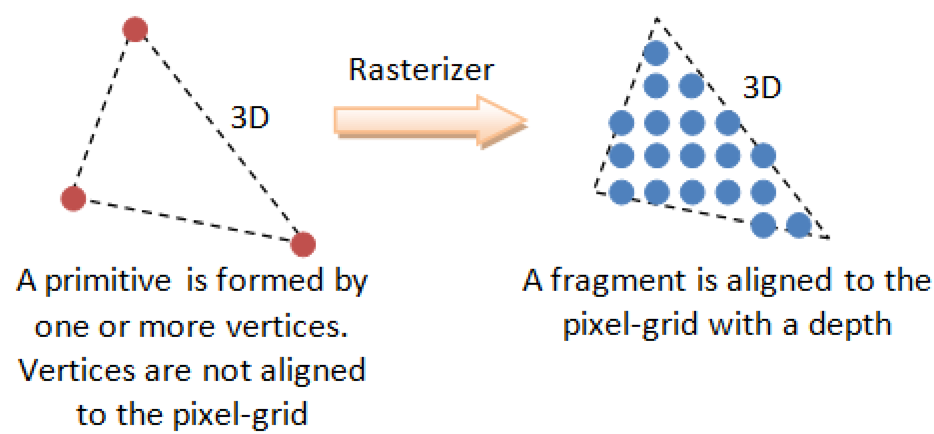
Source: Microsoft

Stream Output Stage

- Allows us to receive data (vertices or primitives) from the geometry shader or vertex shader and feed back into pipeline for processing by another set of shaders
- Useful e.g. for particle systems

Rasterizer Stage

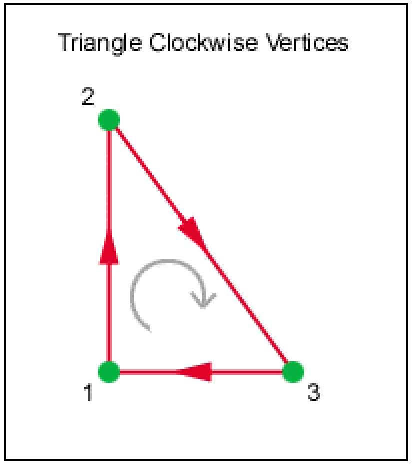
- Interpolates data between vertices to produce per-pixel data
- Clips primitives into view frustum
- Performs culling



Source: ntu.edu.sg

Culling

- In order to avoid rendering vertices that will not be displayed in the final image, DirectX performs ‘culling’
- Triangles facing away from the camera will be culled and not rendered
- By default, DirectX performs ‘Counter-Clockwise culling’
- Triangles with vertices in a counter-clockwise order are not rendered
- The order of vertices is therefore important
- Left hand rule

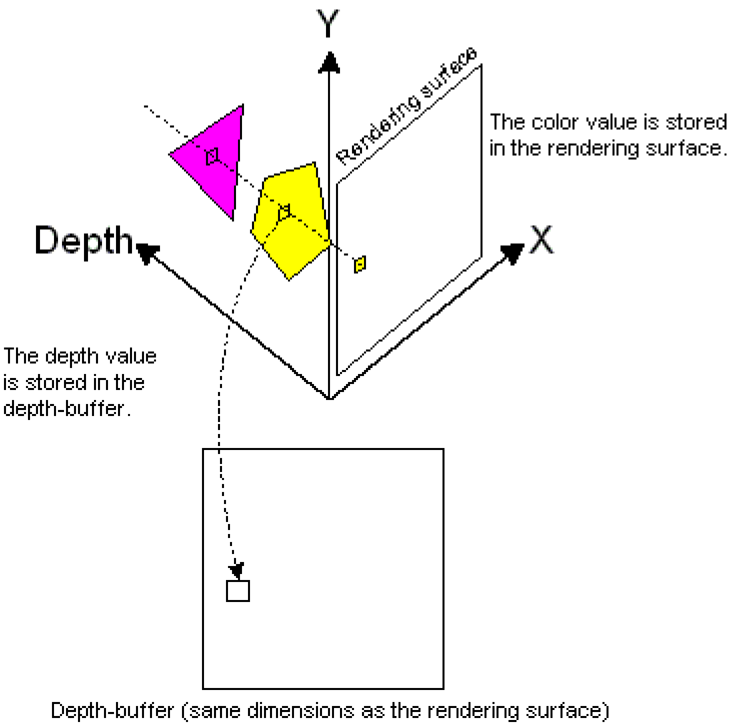


Pixel (Fragment) Shader

- Produces colour values for each interpolated pixel fragment
- Per-pixel lighting can be performed
- Can also produce depth values for depth-buffering

Output-Merger Stage

- Combines pixel shader output values to produce final image
- May also perform depth buffering

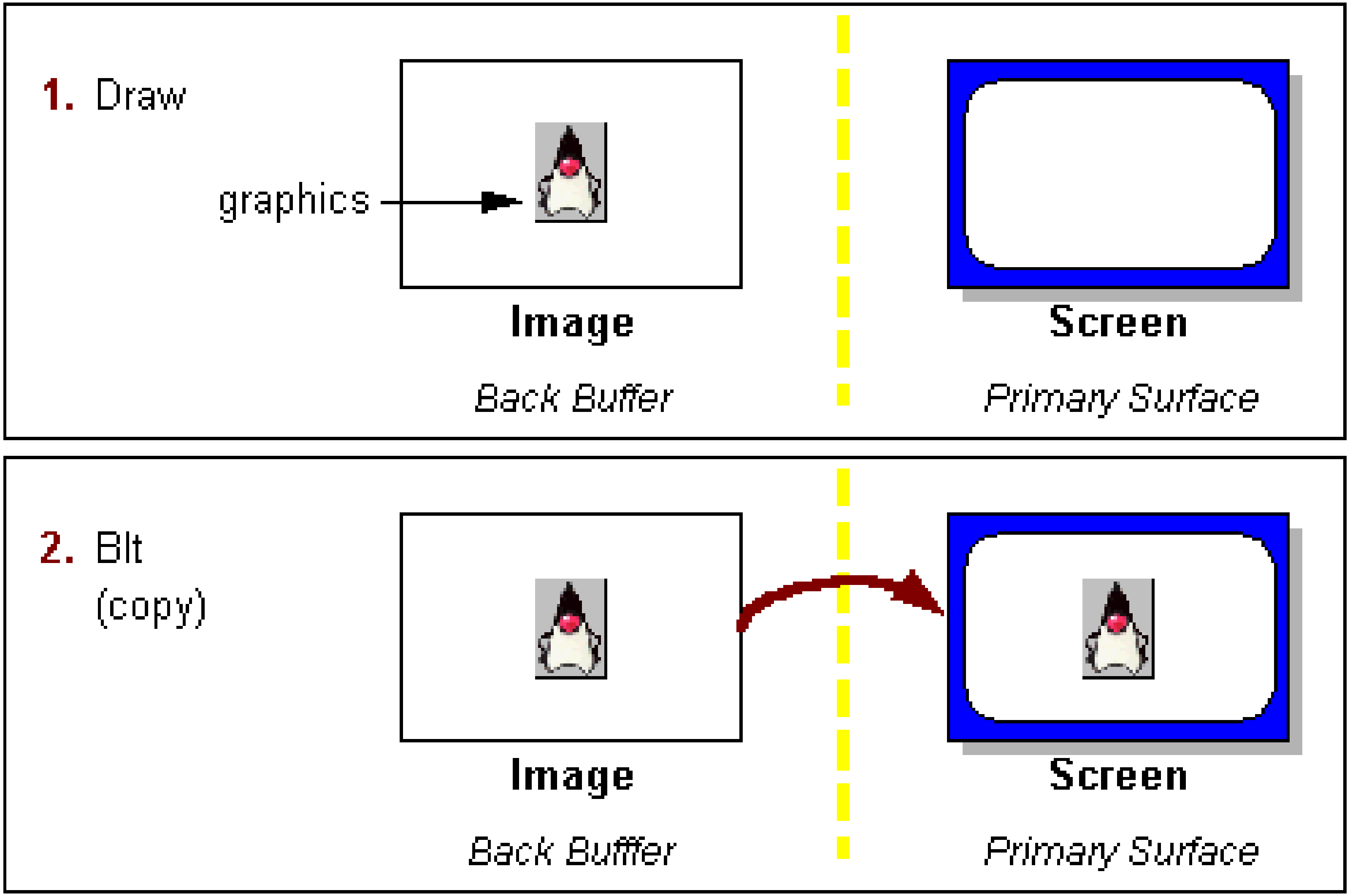


Source: Microsoft

Double Buffering

- Don't want to draw objects directly to the screen
- The screen could update before a new frame has been completely drawn
- Instead, draw next frame to a buffer and swap buffers when complete.

Double Buffering



Source: Oracle

A Very Simple Unity Shader

```
Shader "UnityShaderTutorial/Tutorial1AmbientLight" {
    Properties {
        _AmbientLightColor ("Ambient Light Color", Color) = (1,1,1,1)
        _AmbientLighIntensity("Ambient Light Intensity", Range(0.0, 1.0)) = 1.0
    }
    SubShader
    {
        Pass
        {
            CGPROGRAM
            #pragma target 2.0
            #pragma vertex vertexShader
            #pragma fragment fragmentShader

            fixed4 _AmbientLightColor;
            float _AmbientLighIntensity;

            float4 vertexShader(float4 v:POSITION) : SV_POSITION
            {
                return mul(UNITY_MATRIX_MVP, v);
            }

            fixed4 fragmentShader() : SV_Target
            {
                return _AmbientLightColor * _AmbientLighIntensity;
            }

            ENDCG
        }
    }
}
```

Source: digitalerr0r.wordpress.com

The Structure

- **Shader** "UnityShaderTutorial/Tutorial1AmbientLight - The name we can use to identify it
- **Properties** {
  _AmbientLightColor ("Ambient Light Color", Color) = (1,1,1,1)
  _AmbientLighIntensity("Ambient Light Intensity", Range(0.0, 1.0)) = 1.0
  } – These can be set in the GUI and accessed in the shader
- **SubShader** – We can have more than one SubShader to operate on different hardware
- **Pass**: A subshader can be split into multiple passes, rendering the geometry more than once
- **CGPROGRAM**: This is the 'meat' of the shader – where we specify code to act at differnet levels of the pipeline. Here we specify a vertex shader and a pixel (fragment) shader. We need at least these two to render the geometry.
- **#pragma target 2.0:** This specifies the hardware required for the shader to run. 2.0 is the minimal setting, correspond to Shader Model 2.0 (DX9). See the Unity **Shader Compilation Target Levels** documentation

The Structure

- ```
  #pragma vertex vertexShader
  #pragma fragment fragmentShader
  ```

These specify the names of the functions that will be used as the vertex and fragment shaders respectively

- ```
  float4 vertexShader(float4 v:POSITION) : SV_POSITION
      {
          return mul(UNITY_MATRIX_MVP, v);
      }
  ```

Converts input vertex from object coordinates to camera coordinates. The SV_POSITION semantic indicates to the rasterizer stage that the output should be interpreted as a position value for the vertex

- ```
  fixed4 fragmentShader() : SV_Target
      {
          return _AmbientLightColor * _AmbientLighIntensity;
      }
  ```

Simply sets the colour of a particular pixel to a specific value. The SV_Target semantic instructs the Output Merger stage interpret this as a color value
  ```

What's permitted in CG/HLSL

- The CG/HLSL syntax is quite similar to C, although more restricted. There are a number of permitted datatypes (N.B. Not exhaustive):

Examples of datatypes in HSLS	
bool	true or false
int	32-bit integer
half	16bit integer
float	32bit float
double	64bit double

Source: digitalerr0r.wordpress.com

What's permitted in CG/HLSL

Examples of vectors in HSLS	
float3 vectorTest	float x 3
float vectorTest[3]	float x 3
vector vectorTest	float x 3
float2 vectorTest	float x 2
bool3 vectorTest	bool x 3

Matrices in HSLS	
float3x3	a 3×3 matrix, type float
float2x2	a 2×2 matrix, type float

Source: digitalerr0r.wordpress.com

What's permitted in CG/HLSL

- And a lot of functions

Some functions in HLSL	
cos(x)	Returns cosine of x
sin(x)	Returns sinus of x
cross(a, b)	Returns the cross product of two vectors a and b
dot(a,b)	Returns the dot product of two vectors a and b
normalize(v)	Returns a normalized vector v (v / |v|)

Source: digitalerr0r.wordpress.com

What's permitted in CG/HLSL

- Consult the MSDN documentation for a more exhaustive list:
- Functions: https://msdn.microsoft.com/en-us/library/ff471376.aspx
- Data Types: https://msdn.microsoft.com/en-us/library/bb509587(v=vs.85).aspx