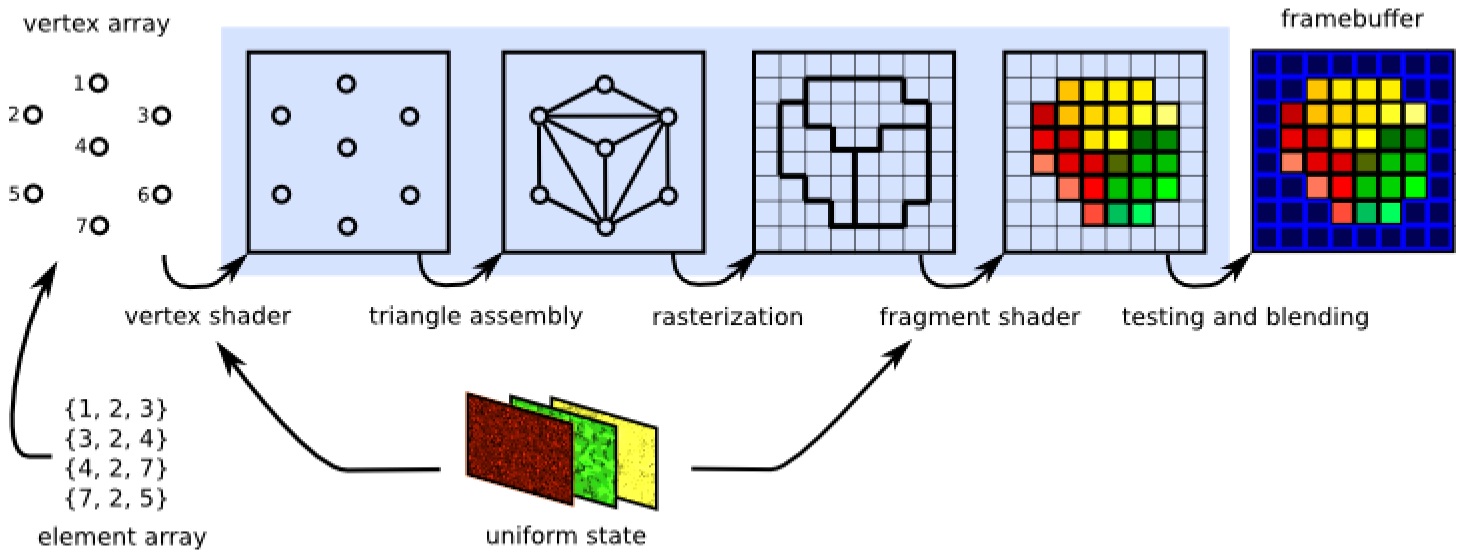
Computer Graphics: Rendering Pipeline

Dept. of Game Software
Yejin Kim

Overview

- Rendering Pipeline
- Shaders
- Tutorials



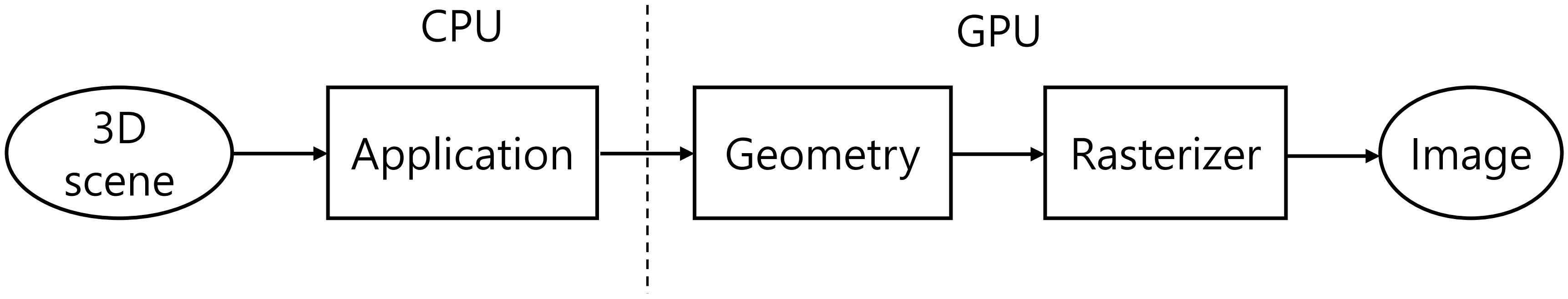
Rendering Pipeline

- Rendering (Image Synthesis)
  - The automatic process of generating a photorealistic or non-photorealistic image from a 2D or 3D model (or models in a scene file) by means of computer programs

Rendering Pipeline

- Pipeline
  - A process that creates images from 2D/3D scenes (models)

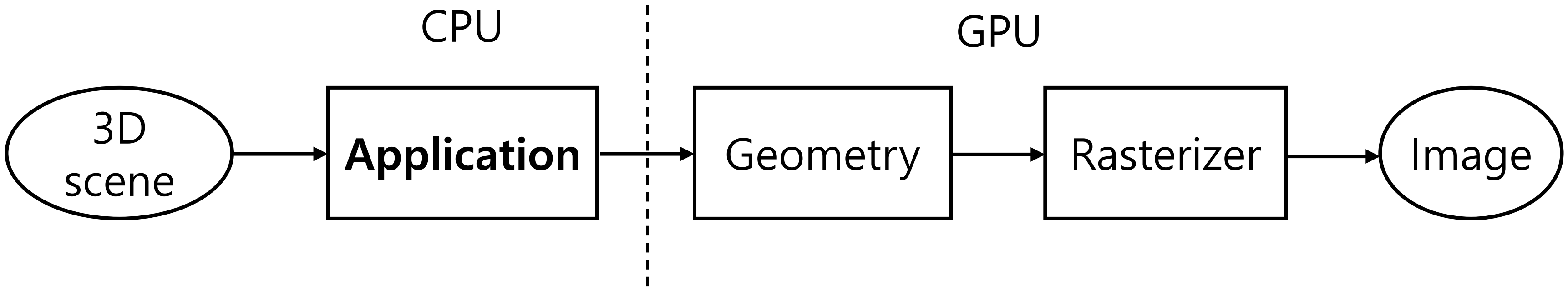
- Three conceptual stages



  - **Application**: loads graphical data to be rendered (using API calls)
  - **Geometry**: performs per-vertex operations on data
  - **Rasterizer**: performs per-pixel operations on data

Rendering Pipeline

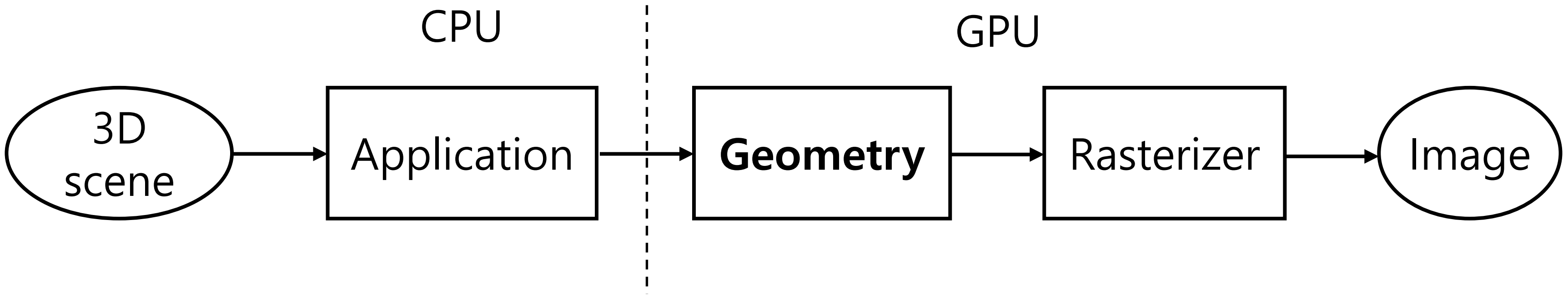
- Application stage



  - Send rendering primitives (i.e. triangles) to the graphics hardware
  - Operations (the programmer decides what to do):
    - Input controls
    - Collision detection
    - Speed-up techniques
    - Animation
    - etc.

Rendering Pipeline

- Geometry stage

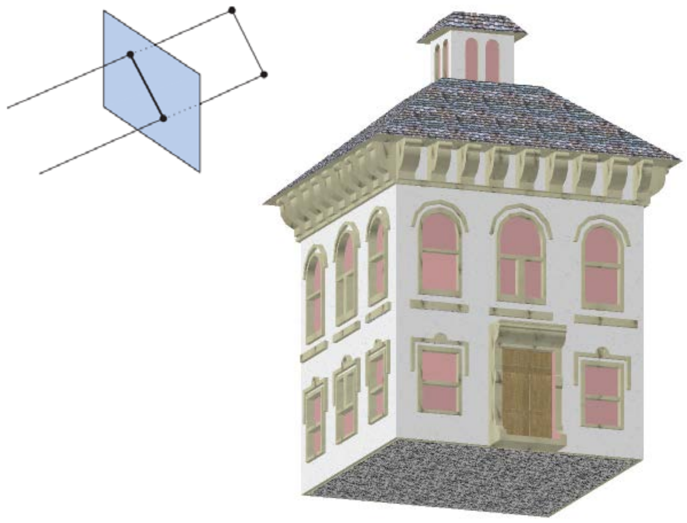
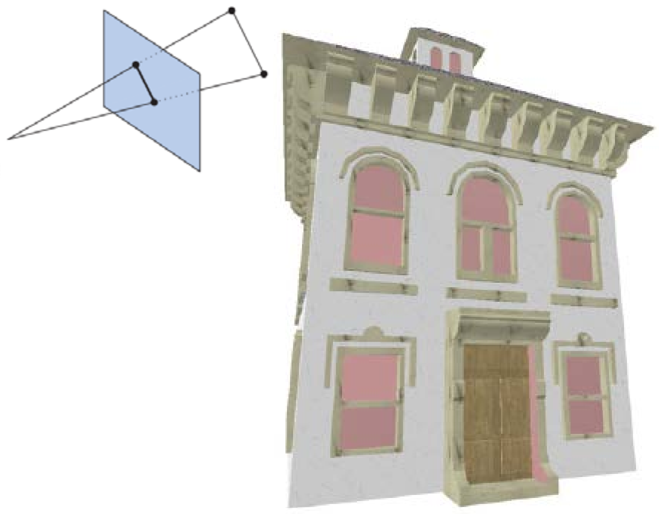
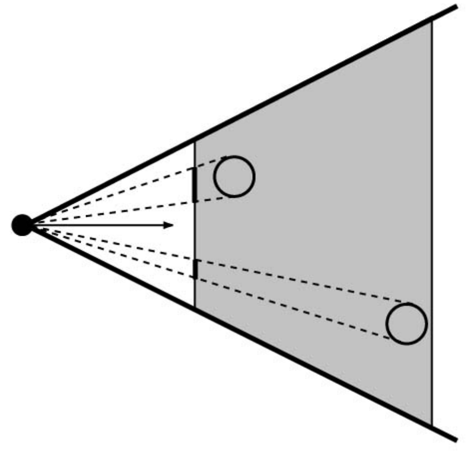


  - Geometrical operations on the input data (i.e. triangles)
  - Per-vertex operations:
    - Move objects and camera
    - Compute lighting at vertices of triangle
    - Project onto screen (3D to 2D)
    - Clipping (avoid triangles outside screen)
    - Map to window

Rendering Pipeline

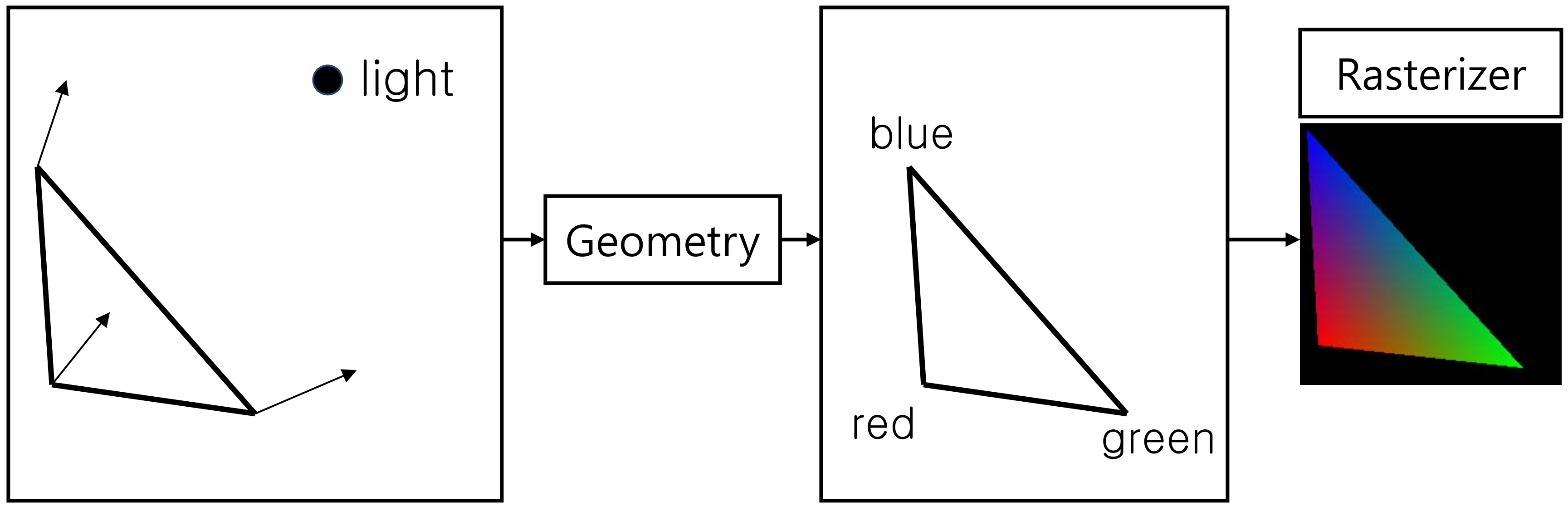
- Geometry stage: Projection
  - Maps points in 3D onto a 2D plane
  - Orthogonal (Parallel)
    - the lines of sight from the object to the projection plane are parallel
  - Perspective (Graphical)
    - 3D objects are projected on a picture plane, which has the effect that distant objects appear smaller than nearer objects

  → D3DClass::Initialize()

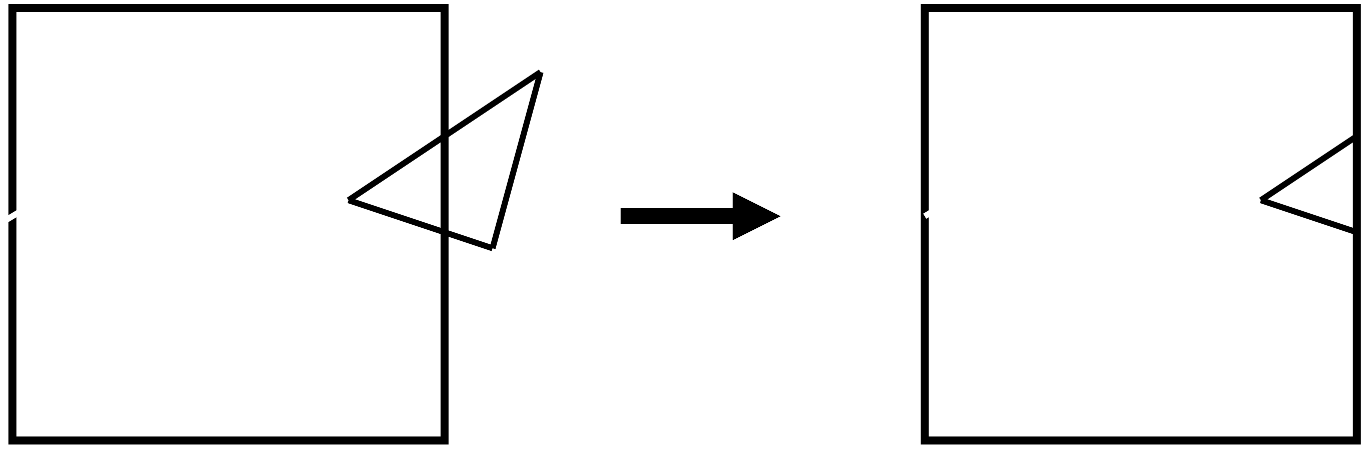
Rendering Pipeline

- Geometry stage: Lighting
  - Compute lighting effects at vertices by simulating the light interaction in nature



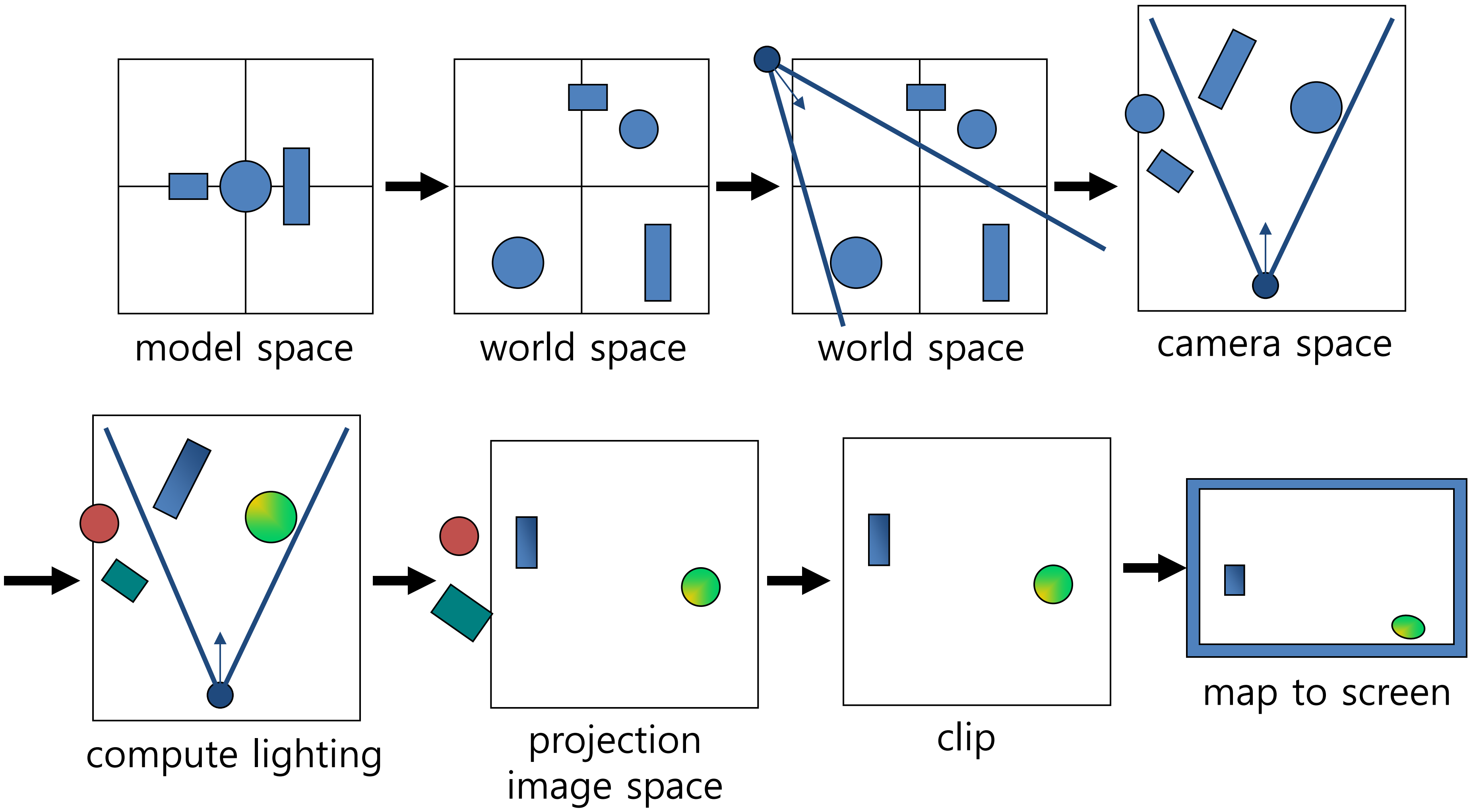
Rendering Pipeline

- Geometry stage: Clipping
  - Clip primitives to square (cube) after projection
  - Screen mapping, scales and translates square so that it ends up in a rendering window
  - These screen space coordinates together with Z (depth) are sent to the rasterizer stage



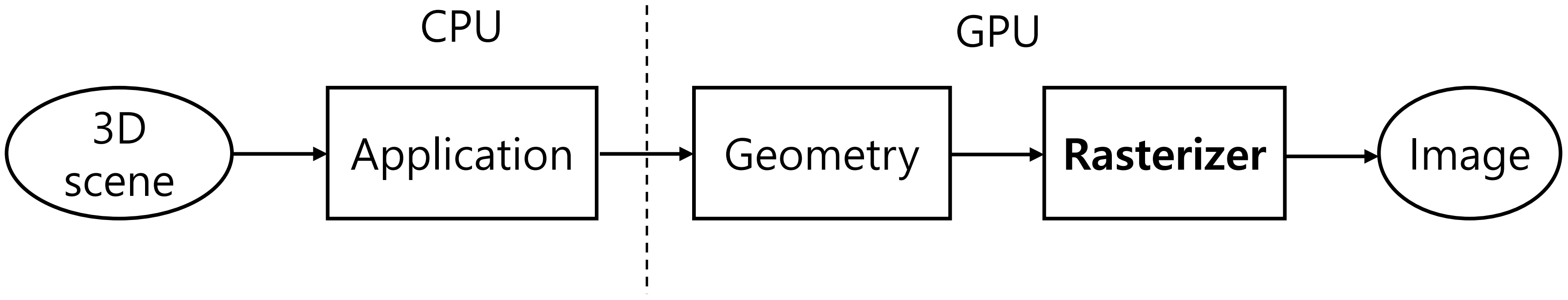
Rendering Pipeline

- Geometry stages: Summary

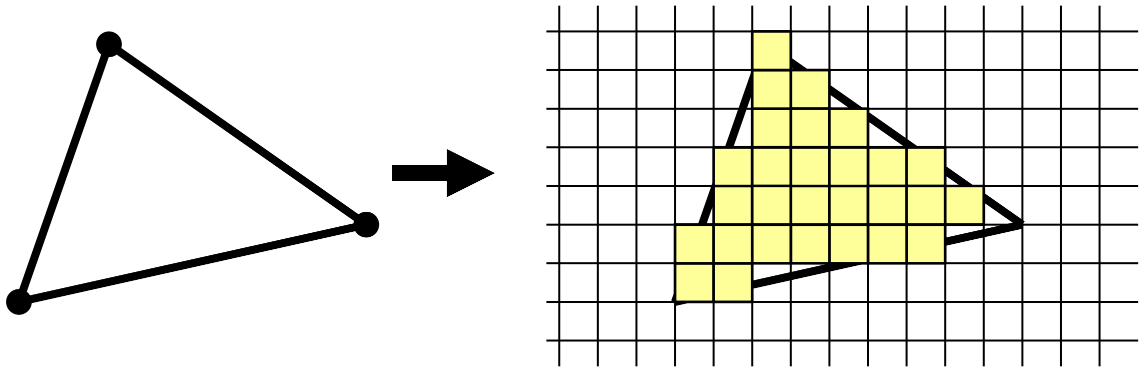


Rendering Pipeline

- Rasterizer stage



  - Input: triangle vertices from geometry
  - Find pixels inside the triangle (or on a line, or on a point)
  - Per-pixel operations:
    - Interpolation
    - Texturing
    - Z-buffering
    - etc.

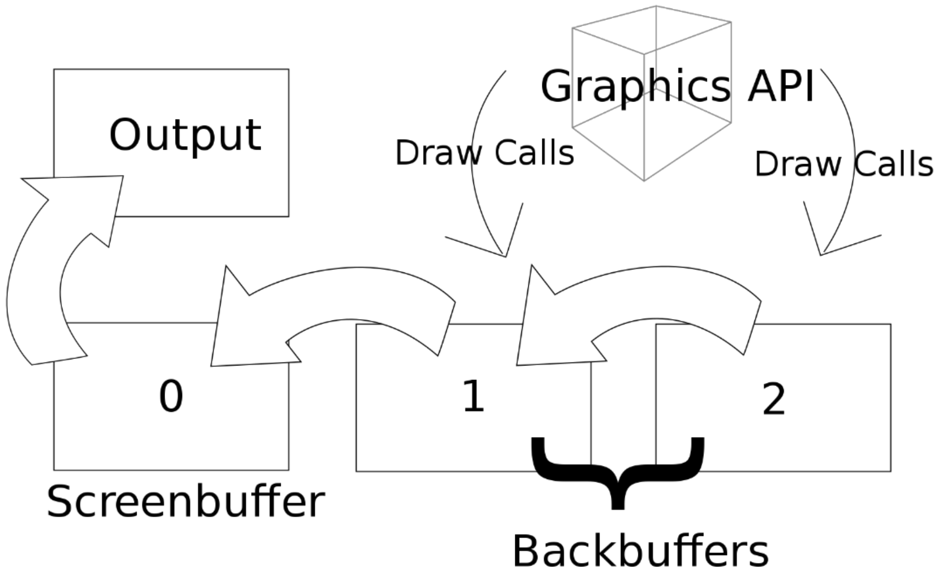


Rendering Pipeline

- Rasterizer stage: Double buffering (Swap chain)
  - Use two frame buffers: one front(screen) and one back
  - The front buffer is displayed while the back buffer is rendered to
  - When new image has been created in back buffer, swap front and back buffers

  → D3DClass::InitializeSwapBuffer()

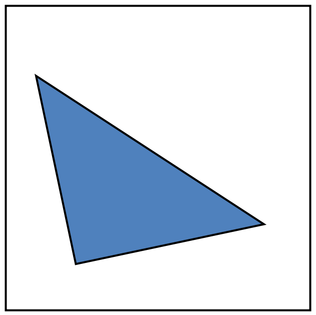
  → ID3D11RenderTargetView* m_renderTargetView

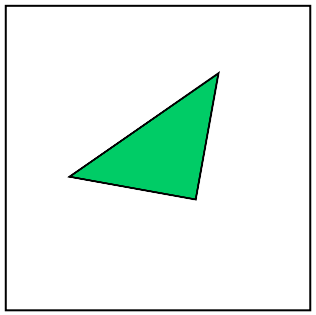


Rendering Pipeline

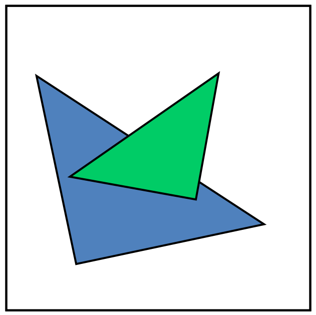
- Rasterizer stage: Z-buffering (Depth buffering)
  - Store z (depth) at each pixel
  - When scan-converting a triangle, compute z at each pixel on triangle
  - Compare triangle's z to Z-buffer z-value
  - If triangle's z is smaller, then replace Z-buffer and color buffer
  - Else do nothing

  → D3DClass::InitializeDepthStencilBuffer()

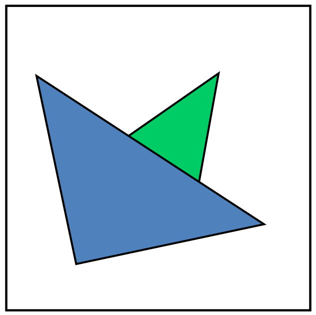

Triangle 1


Triangle 2

incorrect

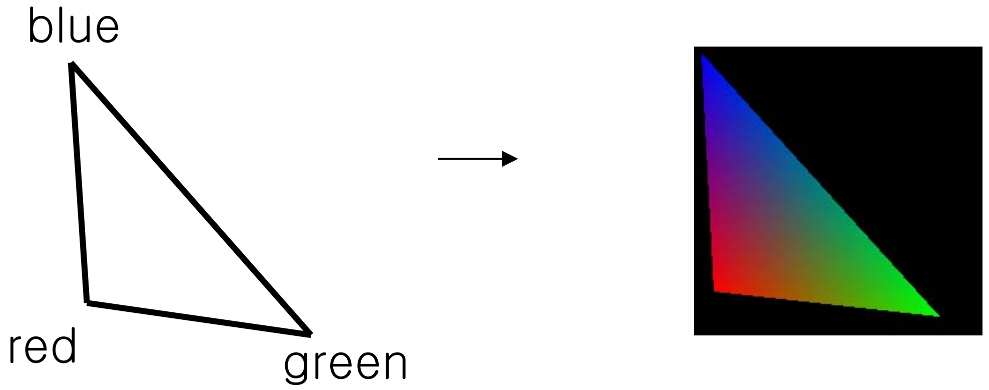

Draw 1 then 2

correct

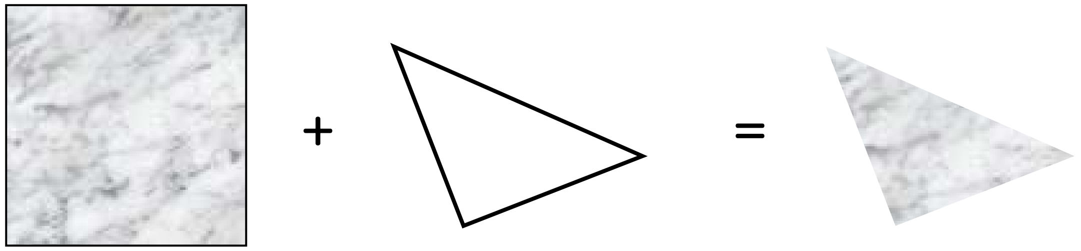

Draw 2 then 1

Rendering Pipeline

- Rasterizer stage: Coloring & texturing
  - (Gouraud) interpolation: estimate colors over the triangle



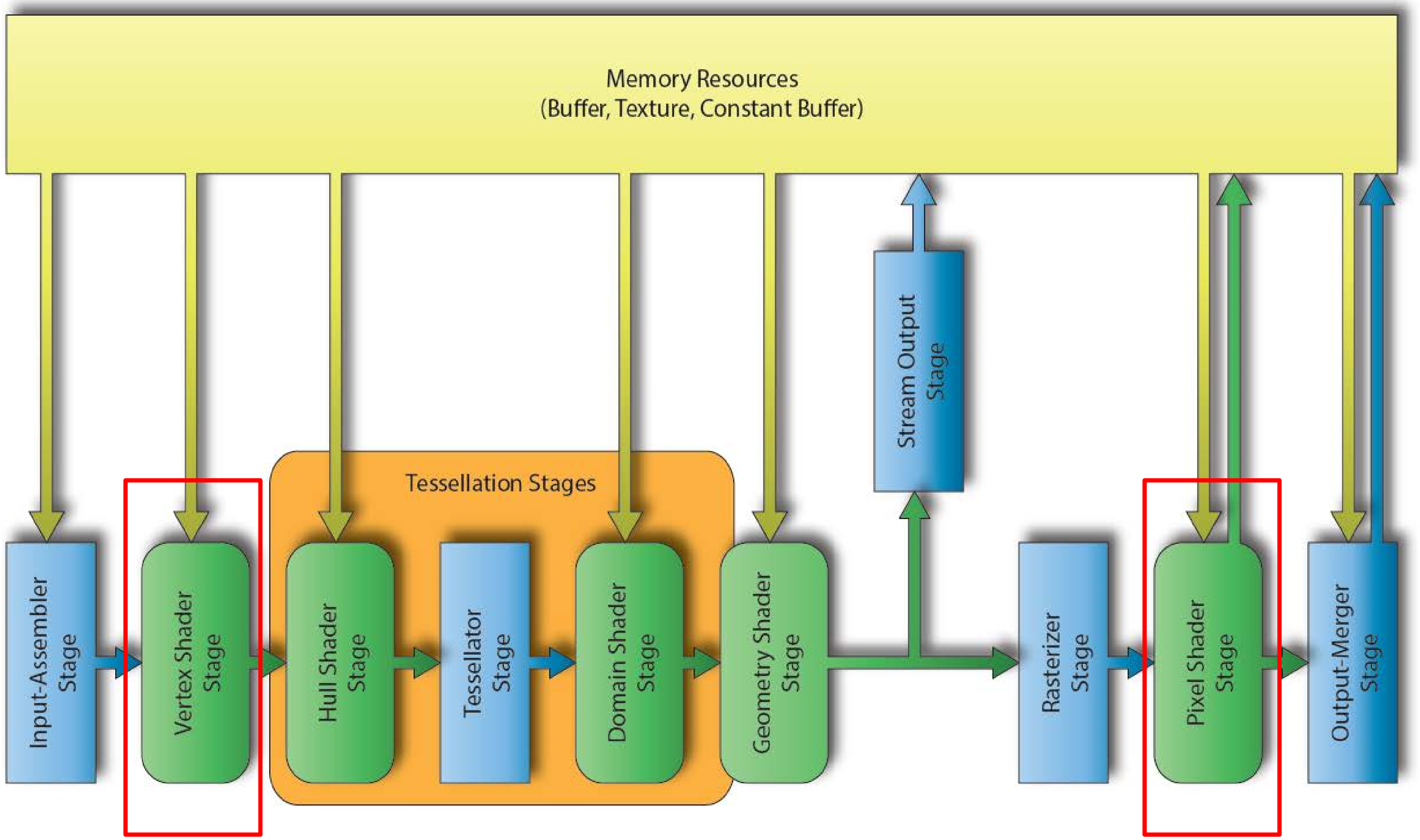
  - Texturing: map images onto geometrical object



    - More realism: bump mapping, pseudo reflections, store lighting, etc.

Rendering Pipeline

- DirectX 11 rendering pipeline



Rendering Pipeline

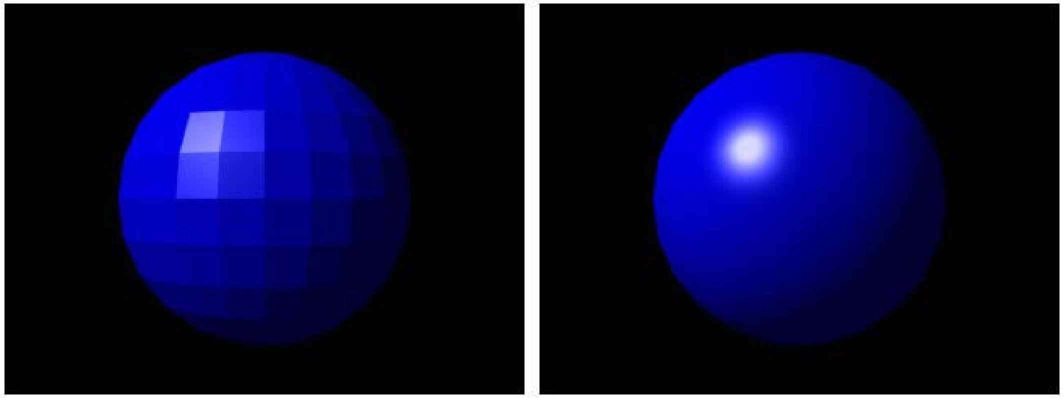
- DirectX 11 rendering pipeline stages:
  - Input-assembler: specifies the geometry and configures the layout of the data which is expected by the vertex shader
  - **Vertex shader**: transforms the vertex position from object space into clip space and performs skinning of skeletal animated meshes or per-vertex lighting
  - Hull shader (optional): determines how much an input control patch should be tessellated by the tessellation stage
  - Tessellator (fixed): subdivides a patch primitive into smaller primitives according to the tessellation factors specified by the hull shader stage
  - Domain shader (optional): computes the final vertex attributes based on the output control points from the hull shader and the interpolation coordinates from the tessellator stage

Rendering Pipeline

- DirectX 11 rendering pipeline stages:
  - Geometry shader (optional): takes a single geometric primitive as input and can either discard the primitive, transform the primitive into another primitive type (for example a point to a quad) or generate additional primitives
  - Stream output (optional, fixed): feeds primitive data back into GPU memory, and the data can be recirculated back to the rendering pipeline to be processed by another set of shaders (useful for spawning or terminating particles in a particle effect)
  - Rasterizer (fixed): clips primitives into the view frustum, performs primitive culling if either front/back-face culling is enabled, and interpolates the per-vertex attributes across the face of each primitive
  - **Pixel Shader**: takes the interpolated per-vertex values from the rasterizer stage and produces one (or more) per-pixel color values
  - Output-Merger:combines the various types of output data (pixel shader output values, depth values, and stencil information) together with the contents of the currently bound render targets to produce the final pipeline result

Shaders

- Shader
  - A program that runs on GPU and calculates the appropriate levels of light, darkness, and color during the rendering of a 3D scene
  - Uses a shading language to program the GPU's rendering pipeline
    - Position and color (hue, saturation, brightness, and contrast)
  - Needs a main program and shading algorithms
    - Using algorithms defined in a shader to build a final rendered image
    - Modifying a final image by external variables or textures introduced by the main program calling the shader.
  - Type: HLSL (DirectX), GLSL (OpenGL), Cg (NVIDIA)


Gouraud vs Phong shading

Shaders

- High-level shader language (HLSL)
  - A proprietary shading language developed by Microsoft for GPU in DirectX 9 and higher
  - Supports the shader construction with C-like syntax, types, expressions, statements, and functions
  - Vertex shader
    - Transforms and manipulates positions, color and texture coordinates
    - Controls over the geometric details of a 3D model
  - Pixel (Fragment) shader
    - Computes color and other attributes of each output pixel (fragment)
    - Controls over the postprocessing effects
      - e.g. Blur, cartoon shading, lighting, etc.

Shaders

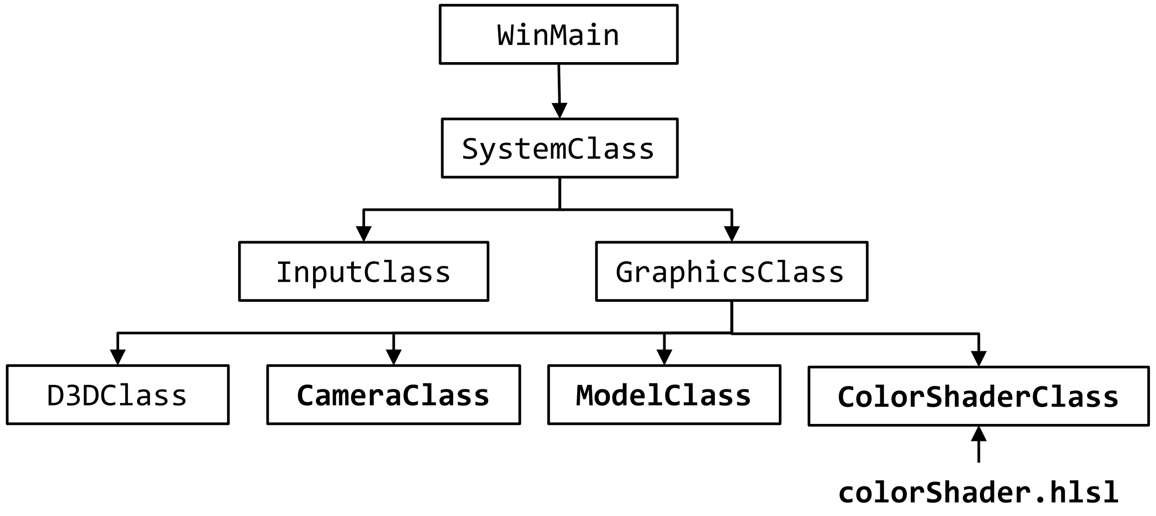
- DirectX shader models
  - DirectX 1~7 (1995~1999): fixed-function rendering pipeline
  - DirectX 8 (2000): Shader Model 1
    - vertex shader
  - DirectX 9 (2002): Shader Model 2
    - vertex, pixel shaders
  - DirectX 9c (2004): Shader Model 3
    - extended vertex, pixel shaders
  - DirectX 10 (2006): Shader Model 4
    - vertex, pixel, geometry shaders
    - effect file: .fx
  - DirectX 11 (2009): **Shader Model 5**
    - vertex, pixel, geometry, tessellation shaders
  - DirectX 12 (2014): Shader Model 6
    - extended vertex, pixel, geometry, tessellation shaders
    - raytracing

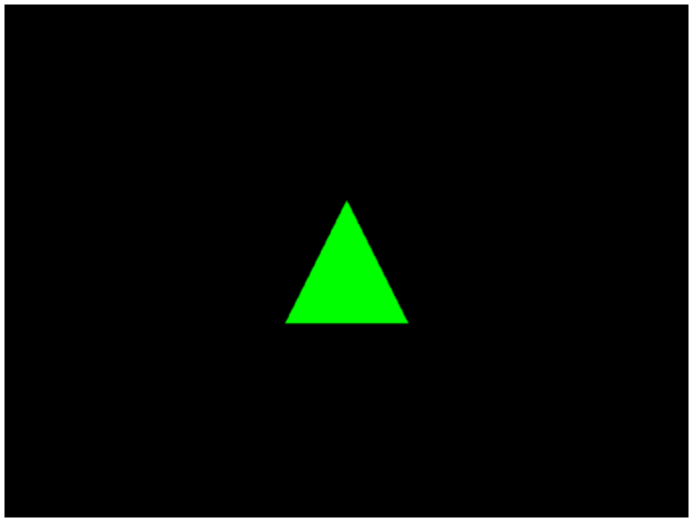
Tutorials

- Shaders
- 3D Model (OBJ)
- Instancing
- 3D Model (FBX)

2-1 Shaders

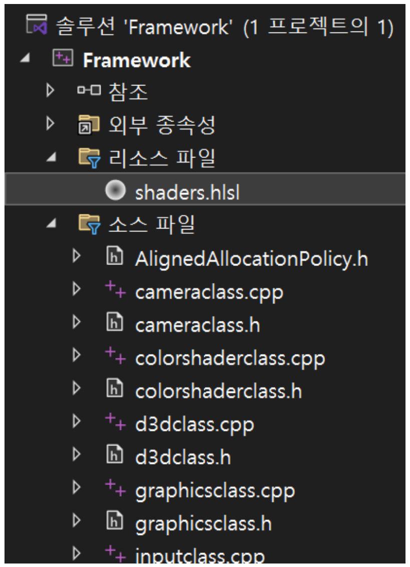
- Adding rendering classes to the Framework
  - AlignedAllocationPolicy: allocate a memory in 16-bit alignment
    - Necessary for GPU programming which processes data in 16-bits
  - CameraClass: handle the camera in the 3D space
  - ModelClass: handle the 3D models
  - ColorShaderClass: render the model using HLSL (shaders.hlsl)



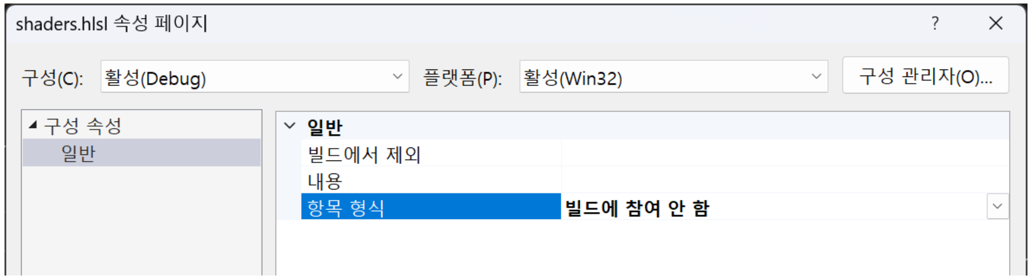


2-1 Shaders

- Shader file: *.hlsl
  - A shader file can be defined with any extension name
    - e.g. effects.fx, colorShader.hlsl, colors.shader, etc.
  - A vertex and a pixel shader is defined as a function respectively in the shader file
    - These functions can be defined in a separate file: shaders.vs, shaders.ps
  - A shader file should be excluded from project build
    - Project → shaders.hlsl → 속성 → 항목 양식: 빌드에 참여 안 함



솔루션 'Framework' (1 프로젝트의 1)
Framework
참조
외부 종속성
리소스 파일
shaders.hlsl
소스 파일
AlignedAllocationPolicy.h
cameraclass.cpp
cameraclass.h
colorshaderclass.cpp
colorshaderclass.h
d3dclass.cpp
d3dclass.h
graphicsclass.cpp
graphicsclass.h
inputclass.cpp



shaders.hlsl 속성 페이지

구성(C): 활성(Debug)　플랫폼(P): 활성(Win32)　구성 관리자(O)...

구성 속성
일반

일반	
빌드에서 제외	
내용	
항목 형식	빌드에 참여 안 함

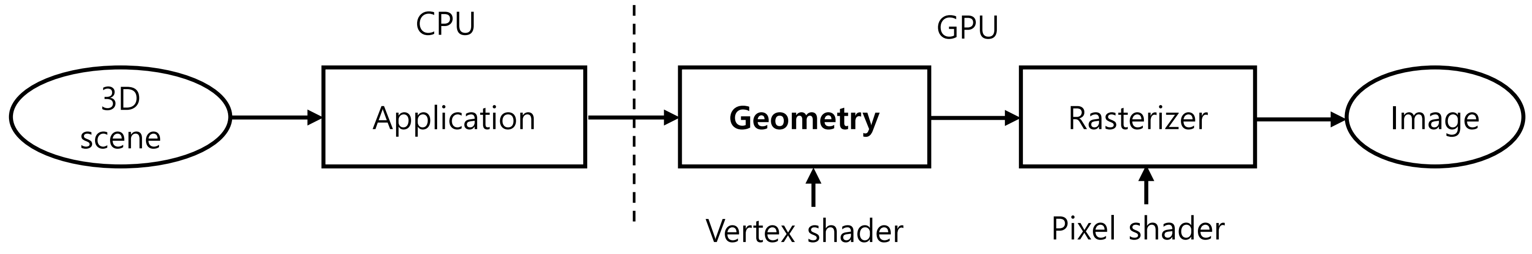
2-1 Shaders

- Geometry data buffers
  - Vertex: a data array for a vertex list
  - Index: a data array to find a vertex in the vertex buffer
  - ID3D11Device::CreateBuffer()

```
// Set up the description of the static vertex(or index) buffer
D3D11_BUFFER_DESC BufferDesc;
BufferDesc.Usage = D3D11_USAGE_DEFAULT;
BufferDesc.ByteWidth = sizeof(VertexType) * object[0].m_vertexCount;
BufferDesc.BindFlags = D3D11_BIND_VERTEX_BUFFER;
BufferDesc.CPUAccessFlags = 0;
BufferDesc.MiscFlags = 0;
BufferDesc.StructureByteStride = 0;

// Give the subresource structure a pointer to the vertex(or index) data
D3D11_SUBRESOURCE_DATA Data;
Data.pSysMem = object[0].vertices;
Data.SysMemPitch = 0;
Data.SysMemSlicePitch = 0;

// Create the vertex(or index) buffer
ID3D11Buffer Buffer;
ID3D11Device::CreateBuffer(&BufferDesc, &Data, &Buffer);
```

2-1 Shaders

- Geometry stage

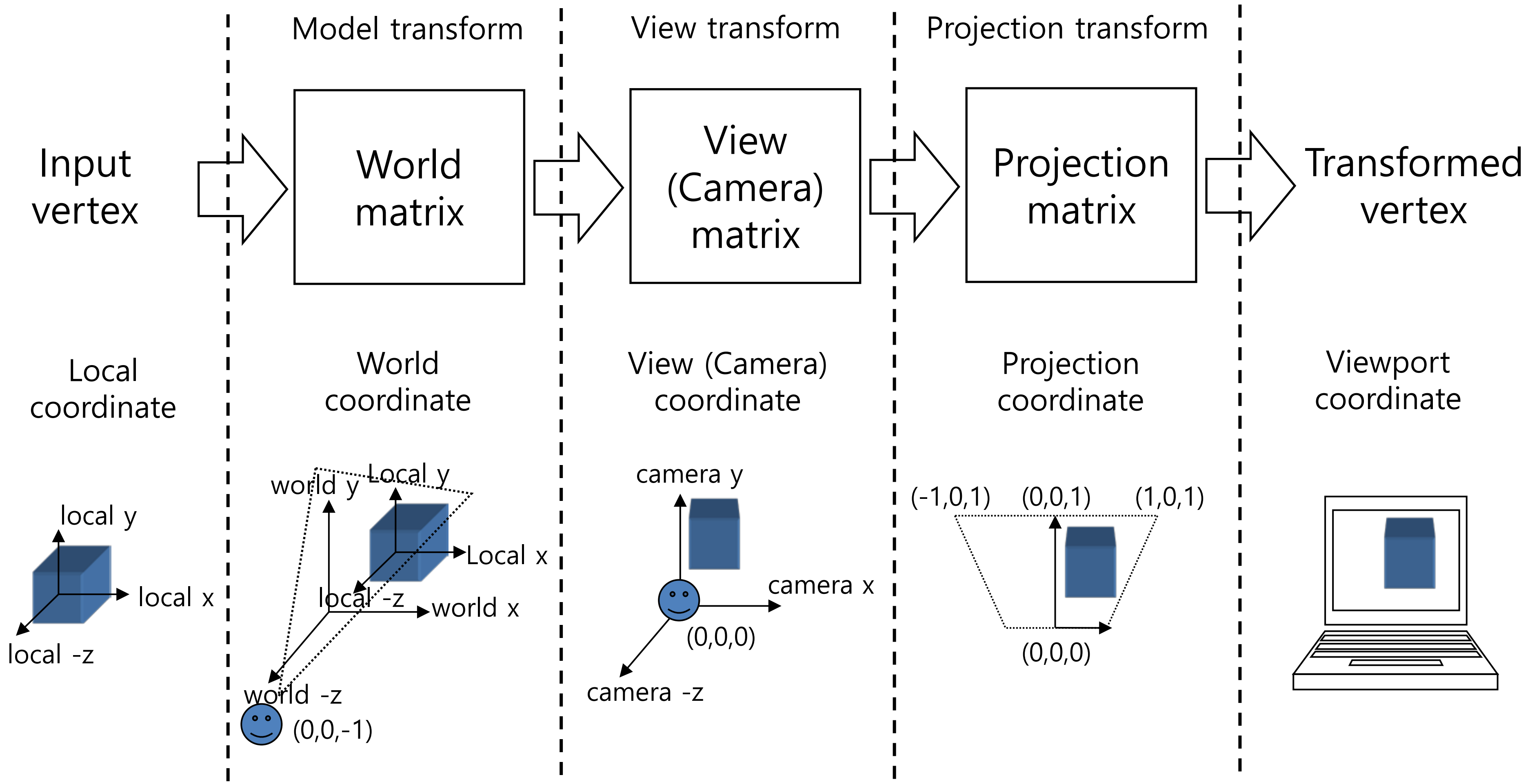


  - Vertex: handles the processing of individual vertices
  - Pixel: colors the polygons

- Vertex transformation matrices
  - World matrix
  - Camera (View) matrix
  - Projection matrix
    - Perspective projection
    - Orthogonal (parallel) projection

2-1 Shaders

- Geometry stage: Vertex transformation matrices
  - Maps an input vertex on a screen



2-1 Shaders

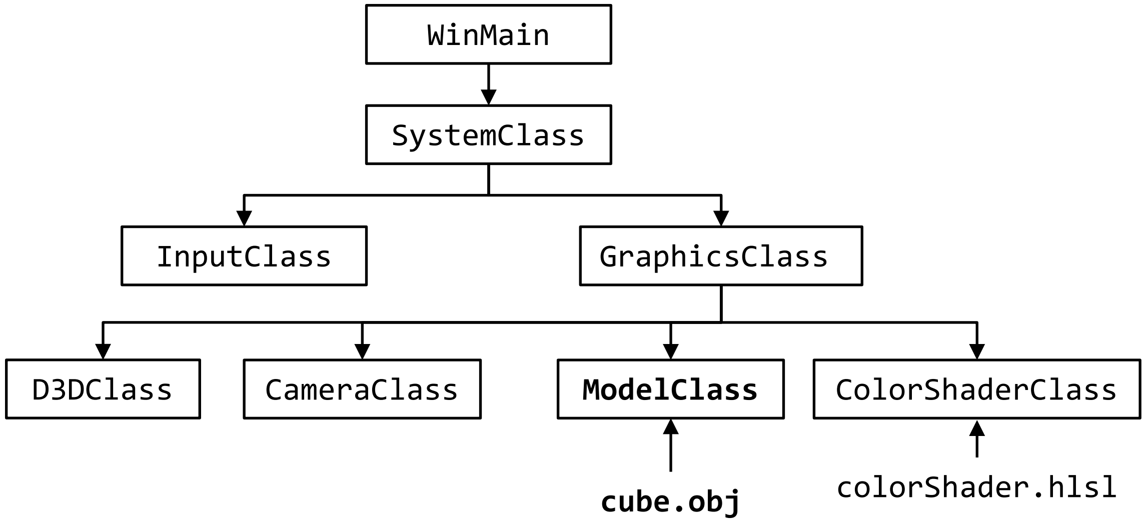
- Model translation, rotation, and scale
  - XMatrixTranslation(x, y, z)
  - XMatrixRotationX(radian)
  - XMatrixRotationY(radian)
  - XMatrixRotationZ(radian)
  - XMatrixScaling(x, y, z)

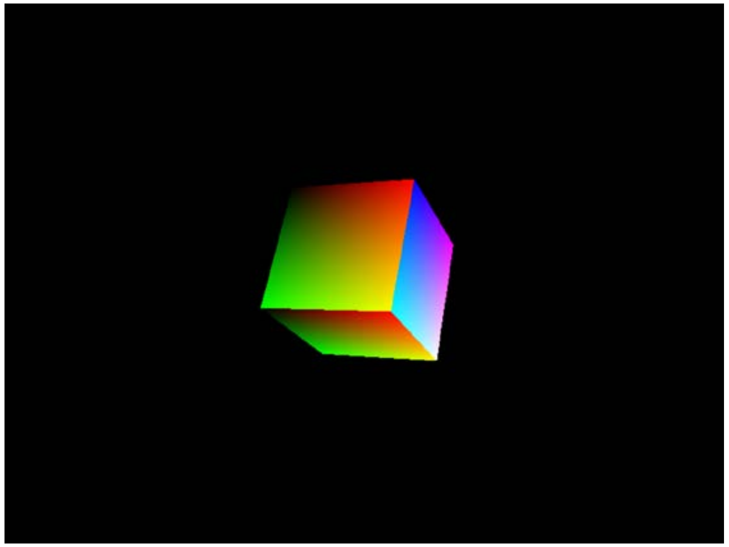
```
XMMATRIX worldMatrix;

m_D3D->GetWorldMatrix(worldMatrix);
worldMatrix *= XMMatrixScaling(0.5f, 0.5f, 0.5f);
worldMatrix *= XMMatrixRotationY(180.0f/XM_PI);        // Radian angle
worldMatrix *= XMMatrixTranslation(-2.0f, 0.5f, 0.0f);
```

2-2 3D Model (OBJ)

- Loading a 3D model from an external file
  - ModelClass: loads 3D model data from an OBJ file

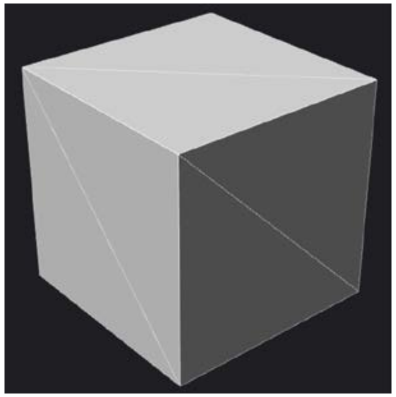
- Rotating and translating the loaded model
  - Use a world matrix





2-2 3D Model (OBJ)

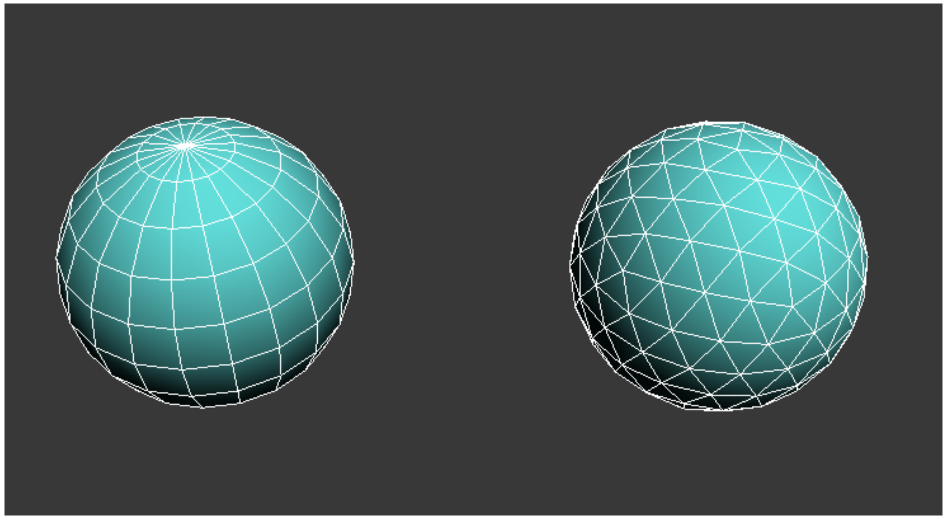
- Loading an external model: OBJ
  - Geometry: "cube.obj"
  - Material: "cube.mtl" (*Not necessary)
    - Include image file names
  - Image: use DDS
    - JPG/JPEG, PNG, GIFF, TIFF → DDS
  - Do NOT include an OBJ file in the "솔류션 탐색기" of a VS project → Build error…!



```
mtllib cube.mtl
g default
v −0.500000 −0.500000 0.500000
…

vt 0.001992 0.001992
…

vn 0.000000 0.000000 1.000000
…

s 1
g pCube1
usemtl file1SG
f 1/1/1 2/2/2 3/3/3
f 3/3/3 2/2/2 4/4/4

s 2
f 3/13/5 4/14/6 5/15/7
f 5/15/7 4/14/6 6/16/8
…
```

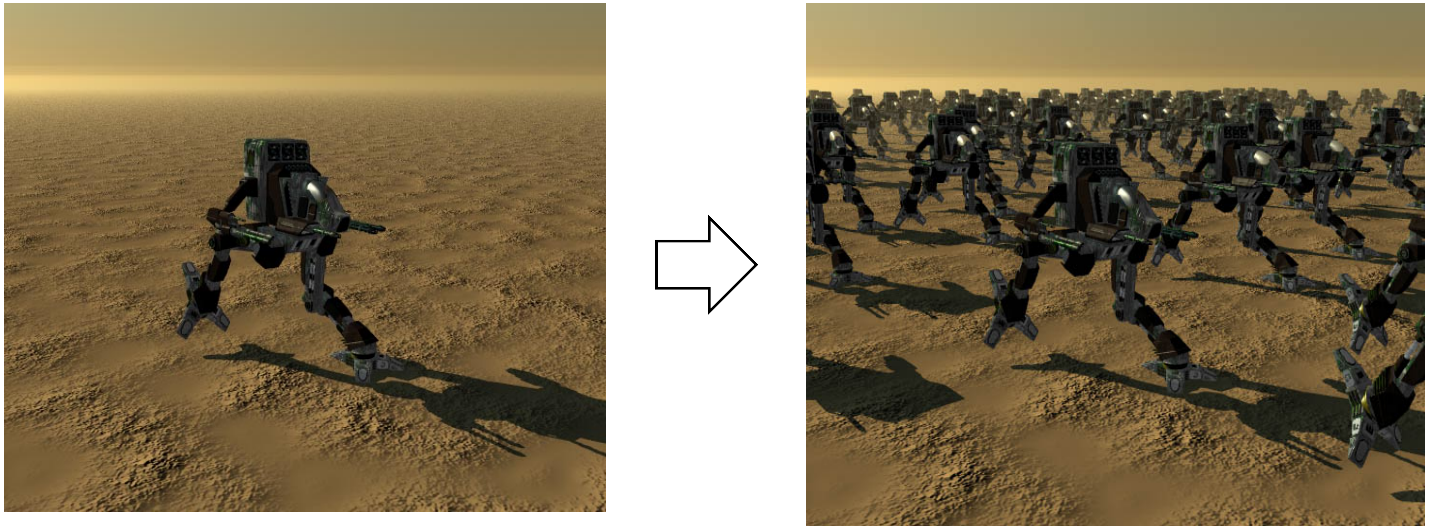
2-2 3D Model (OBJ)

- Loading an external model file
  - Using triangular mesh models
    - deviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
  - Using quadrilateral mesh models
    - deviceContext->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP);
  - Convert Quad → Triangle using 3D modeling tool
    - Blender, Max, Maya, etc.



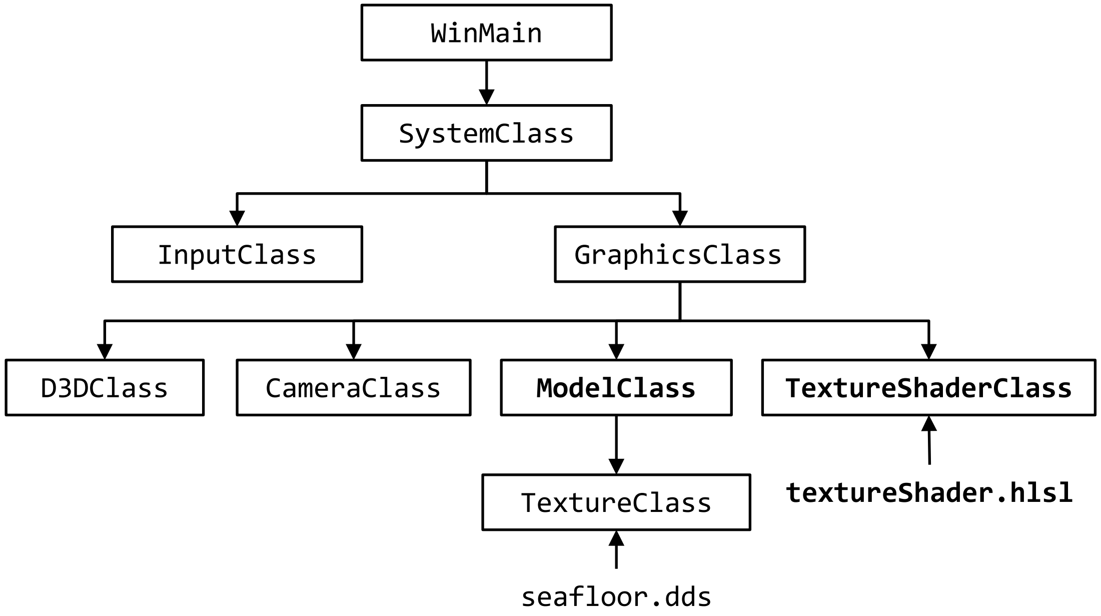
2-3 Instancing

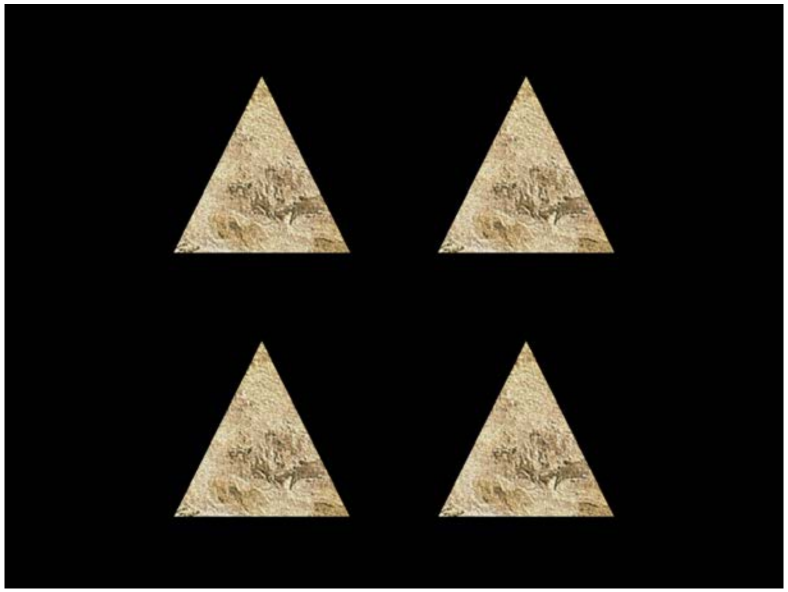
- Mesh Instances
  - Render multiple copies of the same geometry with just changes in position, scale, color, etc.
  - Single vertex buffer + **instance** buffer



2-3 Instancing

- Adding multiple instances to the Framework
  - ModelClass: handles an instance buffer
  - TextureShaderClass: handles setting up instances for the shader



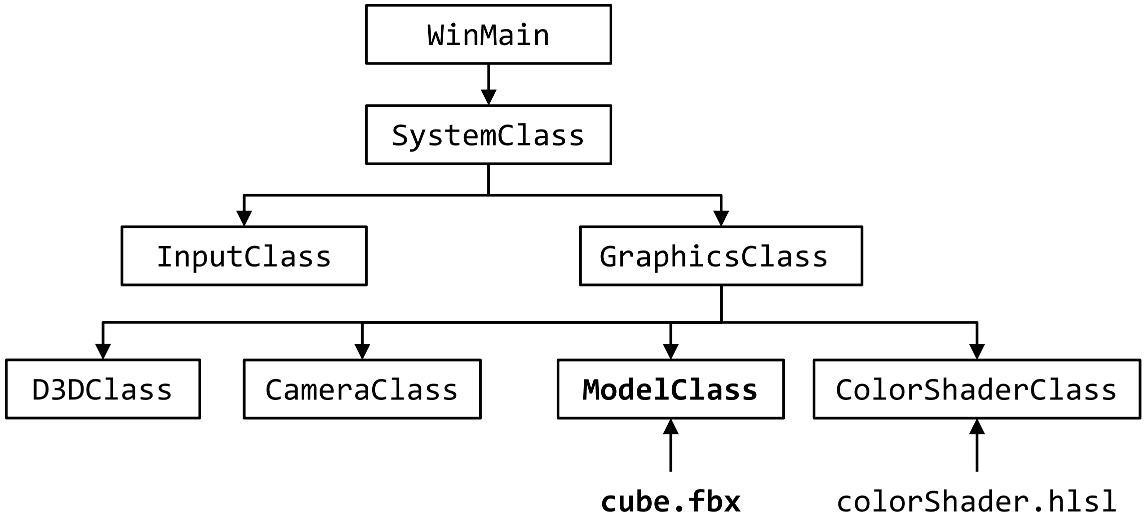


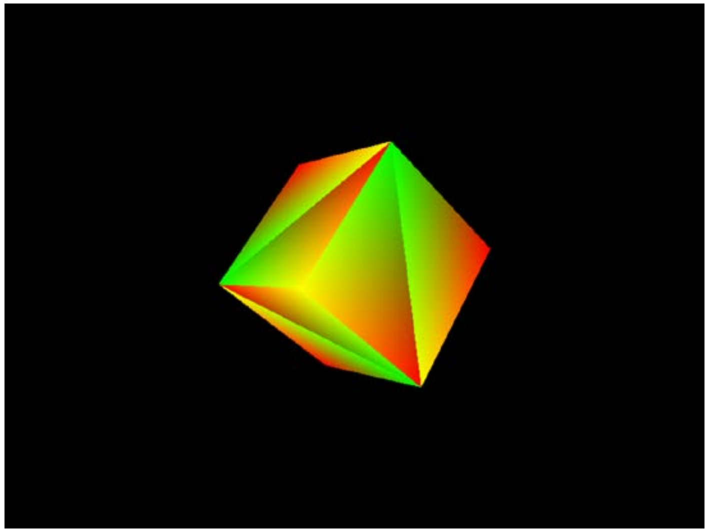
2-4 3D Model (FBX)

- Other 3D file formats: Open Asset Import Library (ASSIMP)
  - https://assimp.org/
  - Written in C++
  - Under a liberal BSD license
  - Loads all input model formats into one straightforward data structure for further processing
  - Augmented by various post processing tools, including frequently-needed operations such as computing normal and tangent vectors.
  - Supports more than 40 file formats

2-4 3D Model (FBX)

- Loading a 3D model from an external file
  - ModelClass: loads 3D model data from a FBX file
  - [Project] 속성 → C/C++ → 추가 포함 디렉터리: include
  - .₩lib₩assimp-vc142-mtd.lib: a ASSIMP library file (x86 Debug)
  - .₩include₩assimp: header files for ASSIMP





References

- Wikipedia
  - www.wikipedia.org
- Introduction to DirectX 11
  - www.3dgep.com/introduction-to-directx-11
- Raster Tek
  - www.ratertek.com
- Braynzar Soft
  - www.braynzarsoft.net)
- CS 445: Introduction to Computer Graphics *[Aaron Bloomield]*
  - www.cs.virginia.edu/~asb/teaching/cs445-fall06

Q & A