Week3

Hooman Salamat

Objectives

1. To discover several key signals used to convey a realistic sense of volume and spatial depth in a 2D image.

2. To find out how we represent 3D objects in Direct3D.

3. To learn how we model the virtual camera.

4. To understand the rendering pipeline—the process of taking a geometric description of a 3D scene and generating a 2D image from it.

Rendering Pipeline

The rendering pipeline refers to the entire sequence of steps necessary to generate a 2D image based on what the virtual camera sees.

The left image shows a side view of some objects setup in the 3D world with a camera positioned and aimed.

The middle image shows the same scene, but from a top-down view. The "pyramid" volume specifies the volume of space that the viewer can see; objects (and parts of objects) outside this volume are not seen.

The image on the right shows the 2D image created based on what the camera "sees."
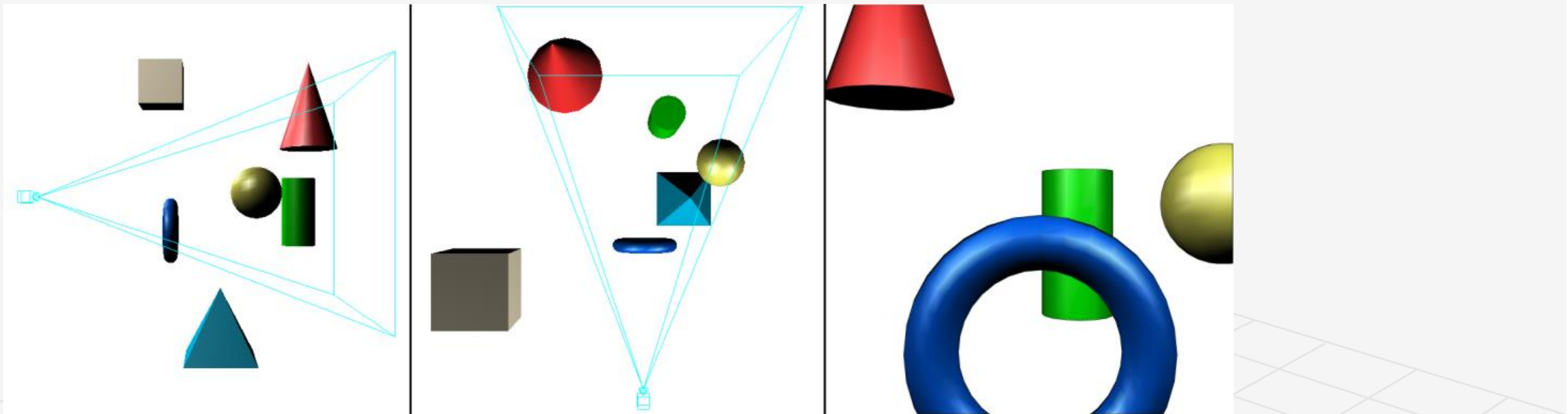
Linear perspective

The railroad rails remain parallel to each other for all time, but if you stand on the railroad and look down its path, you will observe that the two railroad rails get closer and closer together as their distance from you increases, and eventually they converge at an infinite distance. This is one observation that characterizes our human viewing system: parallel lines of vision converge to a *vanishing point*. Artists sometimes call this *linear perspective*.
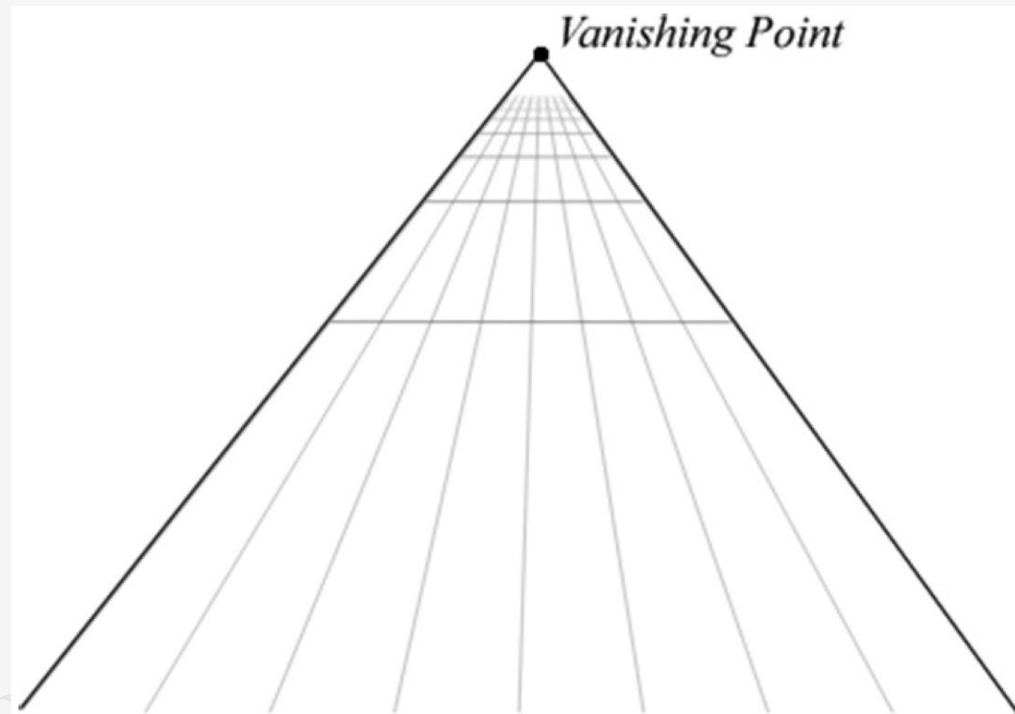
Linear perspective example

Another simple observation of how humans see things is that the size of an object appears to diminish with depth; that is, objects near us look bigger than objects far away. The following figure shows a simple scene where parallel rows of columns are placed behind each other, one after another. The columns are actually all the same size, but as their depths increase from the viewer, they get smaller and smaller. Also notice how the columns are converging to the vanishing point at the horizon.

Depth overlap

We all experience *object overlap* which refers to the fact that opaque objects obscure parts (or all) of the objects behind them. This is an important perception, as it conveys the depth ordering relationship of the objects in the scene. Direct3D uses a depth buffer to figure out which pixels are being obscured and thus should not be drawn.
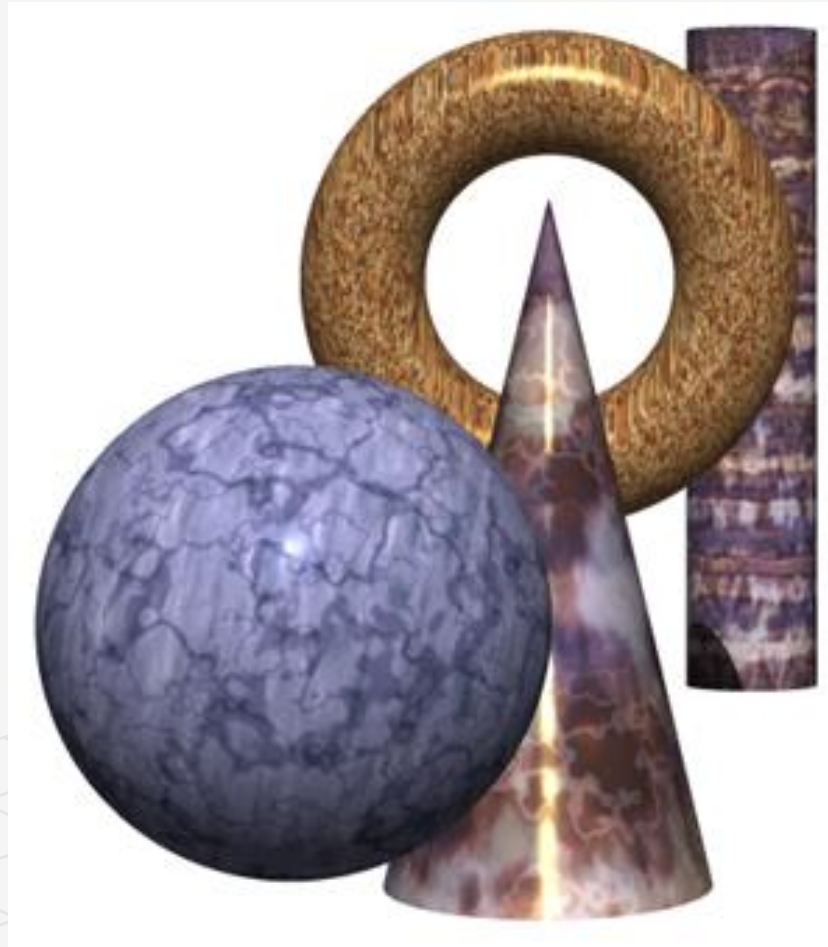
Lighting and shading on a sphere

On the left we have an unlit sphere, and on the right, we have a lit sphere. As you can see, the sphere on the left looks rather flat—maybe it is not even a sphere at all, but just a textured 2D circle! Thus, lighting and shading play a very important role in depicting the solid form and volume of 3D objects.

(a)   (b)
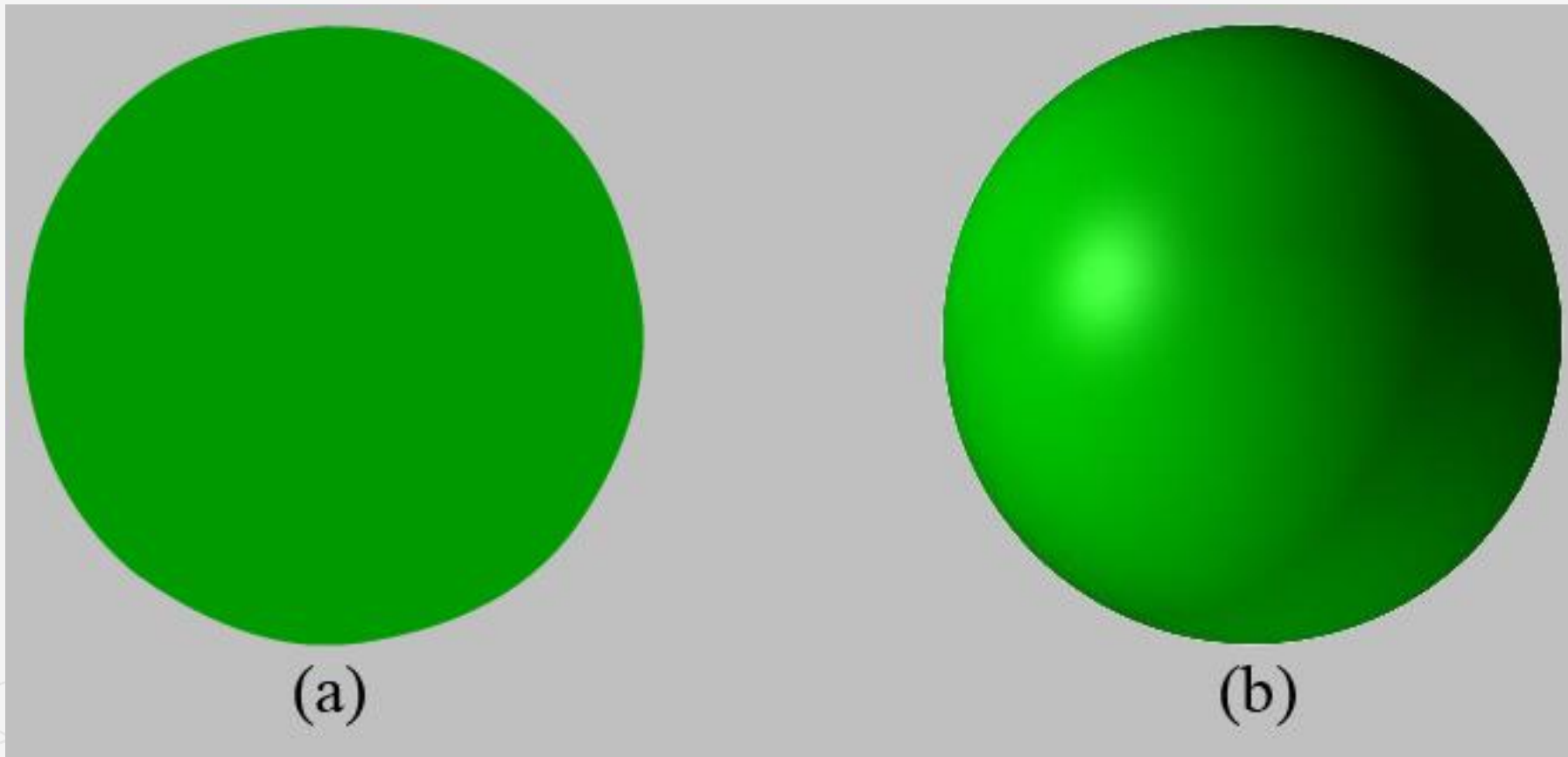
Shadows

The following shows a spaceship and its shadow. The shadow serves two key purposes. First, it tells us the origin of the light source in the scene. And secondly, it provides us with a rough idea of how high off the ground the spaceship is.

MODEL REPRESENTATION

A solid 3D *object* is represented by a *triangle mesh* approximation, and consequently, triangles form the basic building blocks of the objects we model. An object will have more finer detail with the more triangles you use to build it, but the more triangles we use, the more processing power is needed.

In addition to triangles, it is sometimes useful to draw lines or points. For example, a curve could be graphically drawn by a sequence of short line segments a pixel thick.
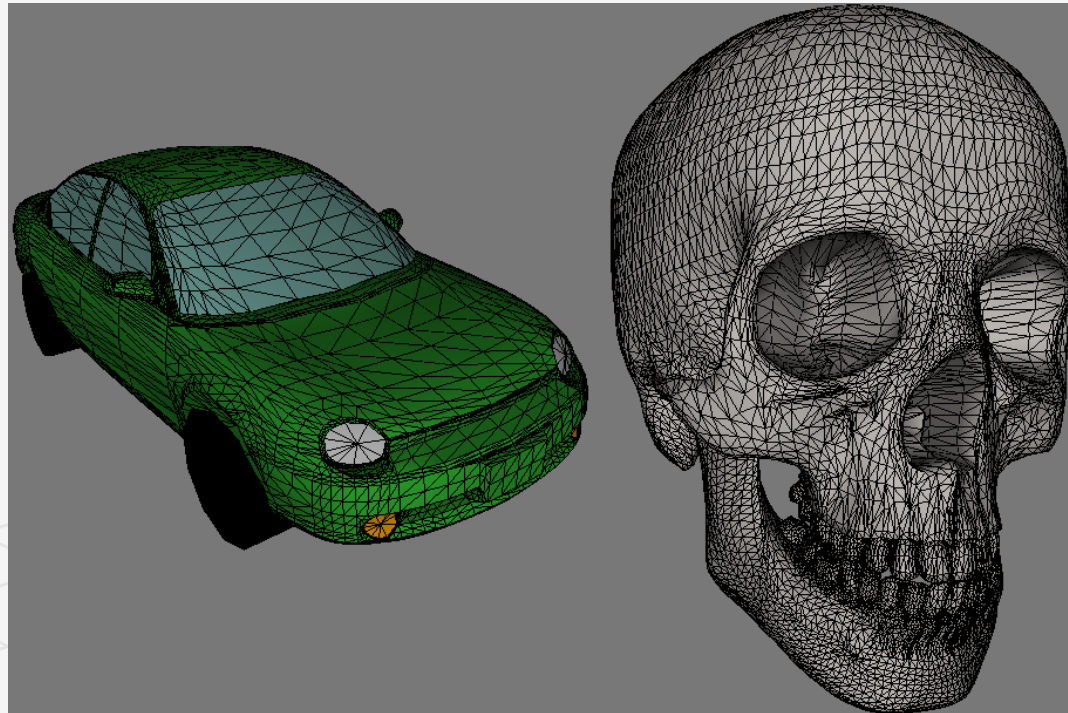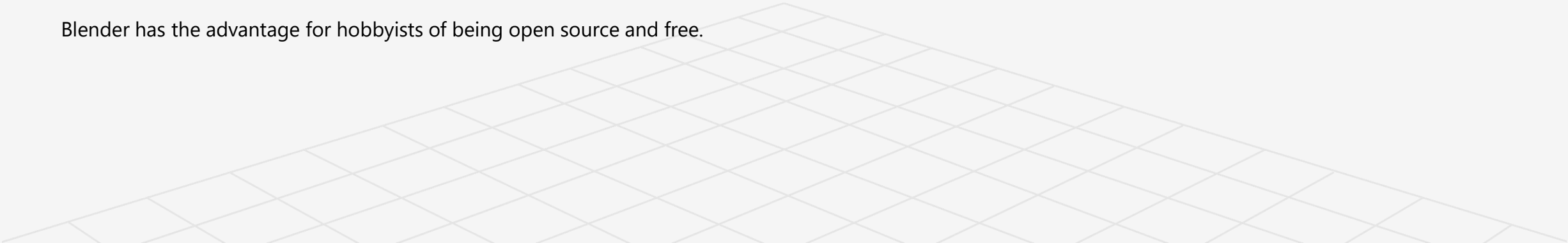
Modeler

3D modelers are used to generate and manipulate 3D objects. These modelers allow the user to build complex and realistic meshes in a visual and interactive environment with a rich tool set, thereby making the entire modeling process much easier.

Examples of popular modelers used for game development are:

1. 3D Studio Max (*http://usa.autodesk.com/3ds-max/*)

2. LightWave 3D (*https://www.lightwave3d.com/*)

3. Maya (*http://usa.autodesk.com/maya/*)

4. Softimage|XSI (www.softimage.com)

5. Blender (*www.blender.org/*)

Blender has the advantage for hobbyists of being open source and free.

BASIC COMPUTER COLOR

Computer monitors emit a mixture of red, green, and blue light through each pixel.

When the light mixture enters the eye and strikes an area of the retina, cone receptor cells

are stimulated and neural impulses are sent down the optic nerve toward the brain.

The brain interprets the signal and generates a color. As the light mixture varies,

the cells are stimulated differently, which in turn generates a different color in the mind.
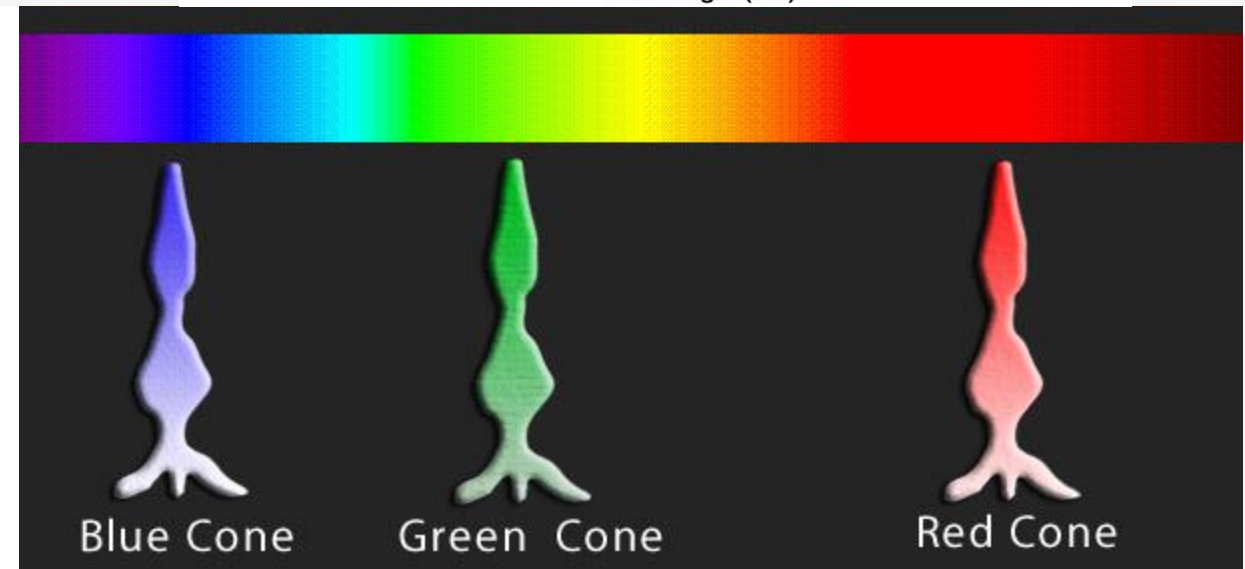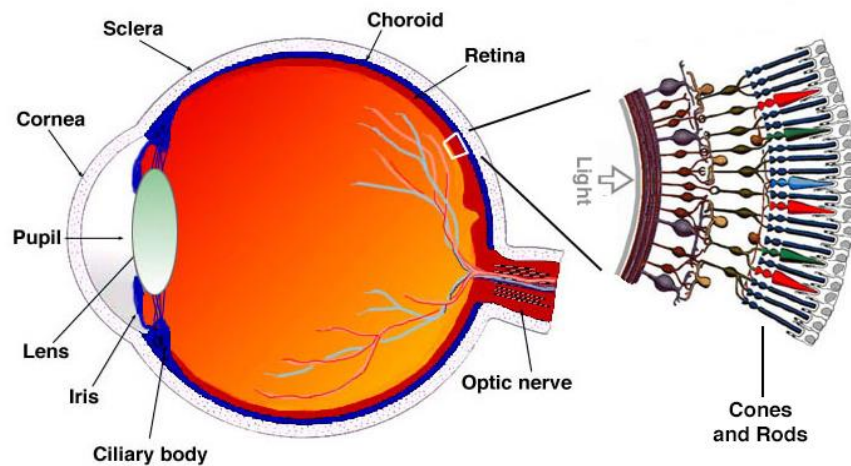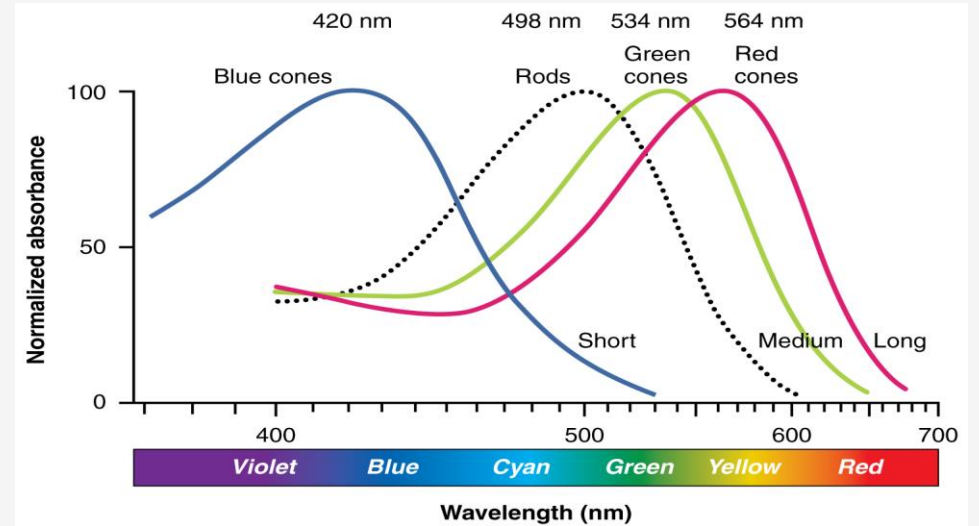
3D color vector

A monitor has a maximum intensity of red, green, and blue light it can emit. To describe the intensities of light, it is useful to use a normalized range from 0 to 1. 0 denotes no intensity and 1 denotes the full intensity. Intermediate values denote intermediate intensities.

For example, the values (0.25, 0.67, 1.0) mean the light mixture consists of 25% intensity of red light, 67% intensity of green light, and 100% intensity of blue light.

We represent a color by a 3D color vector (r, g, b), where $0 \leq r, g, b \leq 1$, and each color component describes the intensity of red, green, and blue light.

we can add color vectors to get new colors: (0.0, 0.5, 0) + (0, 0.0, 0.25) = (0.0, 0.5, 0.25)

By combining a medium intensity green color with a low intensity blue color, we get a dark-green color.

Colors can also be subtracted to get new colors: (1, 1, 1) – (1, 1, 0) = (0, 0, 1)

Scalar multiplication also makes sense. Consider the following: 0.5(1, 1, 1) = (0.5, 0.5, 0.5)

XMColorModulate

Color vectors get their own special color operation called *modulation* or *component-wise* multiplication. It is defined as:

$$(c_r, c_g, c_b) \otimes (k_r, k_g, k_b) = (c_r k_r, c_g k_g, c_b k_b)$$

$$(r, g, b) \otimes (0.5, 0.75, 0.25) = (0.5r, 0.75g, 0.25b)$$

```cpp
inline XMVECTOR XM_CALLCONV XMColorModulate
(
    FXMVECTOR C1,
    FXMVECTOR C2
)
{
    return XMVectorMultiply(C1, C2);
}


XMVECTOR color1 = XMVectorSet(1.0f, 0.0f, 0.0f, 1.0f);
XMVECTOR color2 = XMVectorSet(0.5f, 0.75f, 0.25f, 1.0f);
cout << "colorModulation = color1 * color2  = " << XMColorModulate(color1,color2) << endl;
```

DirectX::PackedVector namespace

The DirectX Math library (*#include <DirectXPackedVector.h>*) provides the following structure, in the DirectX::PackedVector namespace, for storing a 32-bit color:

```cpp
namespace DirectX
{

namespace PackedVector
{
struct XMCOLOR
{
    union
    {
        struct
        {
            uint8_t b;  // Blue:    0/255 to 255/255
            uint8_t g;  // Green:   0/255 to 255/255
            uint8_t r;  // Red:     0/255 to 255/255
            uint8_t a;  // Alpha:   0/255 to 255/255
        };
        uint32_t c;
    };

    XMCOLOR() = default;

    XMCOLOR(const XMCOLOR&) = default;
    XMCOLOR& operator=(const XMCOLOR&) = default;

    XMCOLOR(XMCOLOR&&) = default;
    XMCOLOR& operator=(XMCOLOR&&) = default;

    XM_CONSTEXPR XMCOLOR(uint32_t Color) : c(Color) {}
    XMCOLOR(float _r, float _g, float _b, float _a);
    explicit XMCOLOR(_In_reads_(4) const float *pArray);

    operator uint32_t () const { return c; }

    XMCOLOR& operator= (const uint32_t Color) { c = Color; return *this; }
};
```

PackedVector::XMLoadColor

To convert a 32-bit color (XMCOLOR) to a 128-bit color (XMVECTOR) and conversely, the DirectXMath library defines the following functions:

```cpp
XMVECTOR    XM_CALLCONV     XMLoadColor(const XMCOLOR* pSource);

void    XM_CALLCONV     XMStoreColor(XMCOLOR* pDestination, _In_ FXMVECTOR V);
```

Typically, 128-bit colors values are used where high precision color operations are needed (e.g., in a pixel shader); in this way, we have many bits of accuracy for the calculations so arithmetic error does not accumulate too much. The final pixel color, however, is usually stored in a 32-bit color value in the back buffer;

Rendering Pipeline

Most stages do not write to GPU resources Instead, their output is just fed in as input to the next stage

- Input Assembler Stage (IA)
- Vertex Shader Stage (VS)
- Hull Shader Stage (HS)
- Tessellator Stage (TS)
- Domain Shader Stage (DS)
- Geometry Shader Stage (GS)
- Rasterizer Stage (RS)
- Pixel Shader Stage (PS)
- Output Merger Stage (OM)
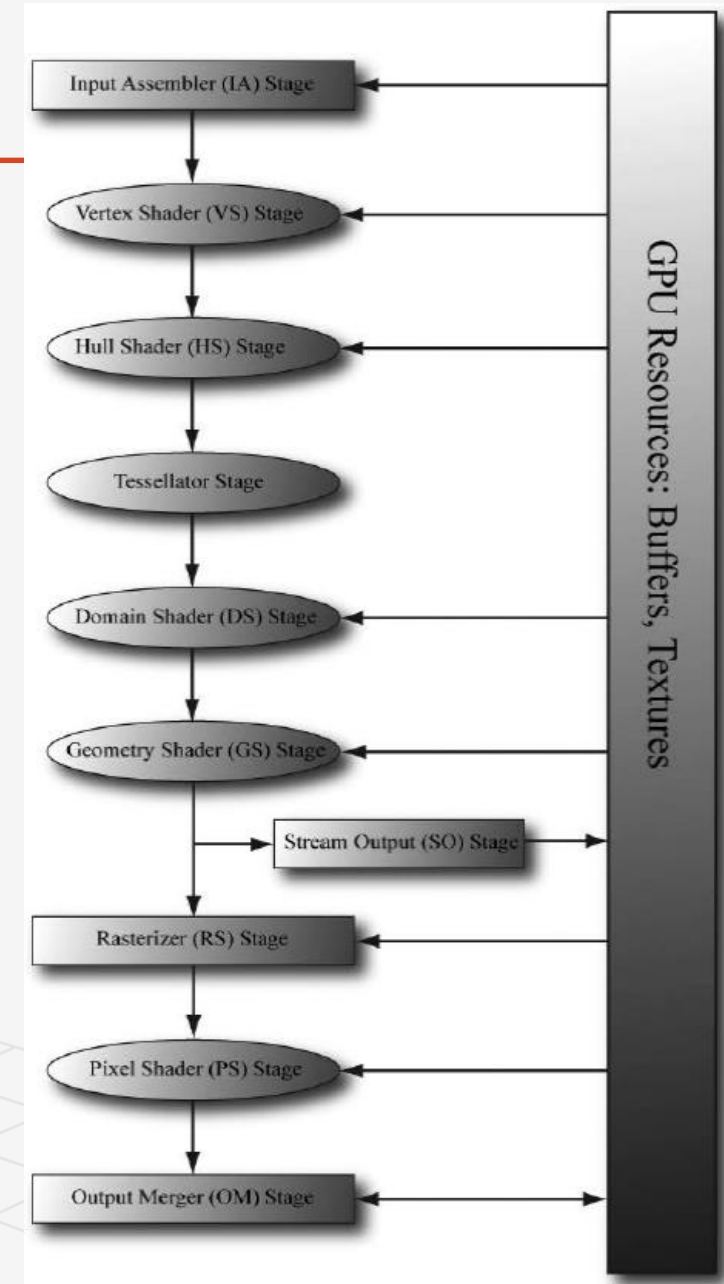
Direct3D 10 Pipeline

IASetVertexBuffers/SetIndexBuffer

IASetPrimitiveTopology

{VS|GS|PS}SetShader

{VS|GS|PS}SetShaderResources

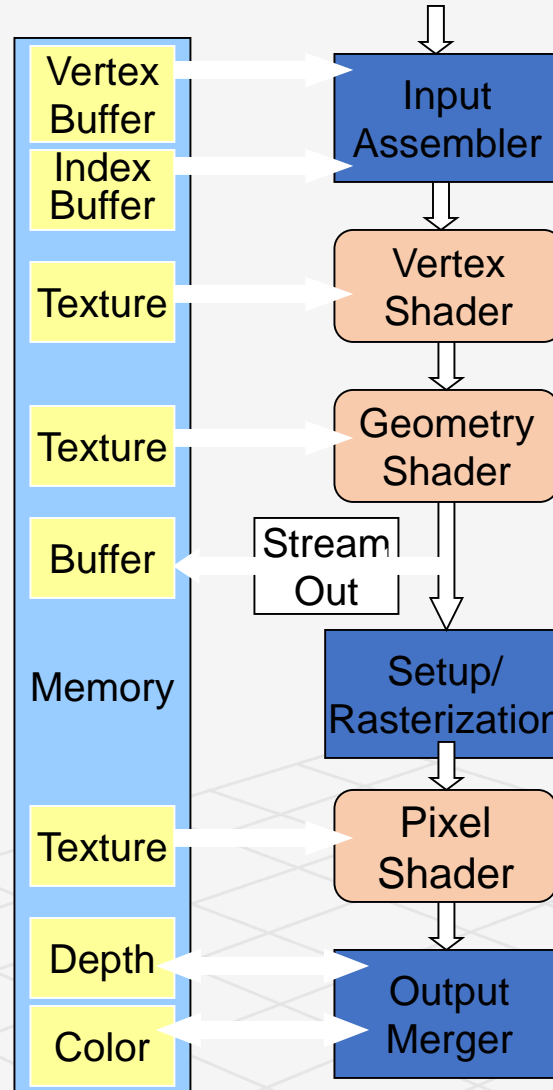{VS|GS|PS}SetConstantBuffers

{VS|GS|PS}SetSamplers

SOSetTargets

RSSetState

RSSetViewports/ScissorRects

OMSetRenderTargets

OMSetBlendState

OMSetDepthStencilState

Memory

Vertex Buffer	Input Assembler
Index Buffer	
Texture	Vertex Shader
Texture	Geometry Shader
Buffer	Stream Out
	Setup/ Rasterization
Texture	Pixel Shader
Depth	Output Merger
Color	

Input Assembler Stage

. Reads geometric data (vertices and indices) from memory and uses it to assemble geometric primitives (e.g., triangles, lines)

▪ Vertices are bound to the rendering pipeline in a special Direct3D data structure called a vertex buffer

▪ A vertex buffer just stores a list of vertices in contiguous memory.

▪ We tell Direct3D how to form geometric primitives from the vertex data by specifying the primitive topology

```
mCommandList->IASetPrimitiveTopology(D3D11_PRIMITIVE_TOPOLOGY_TRIANGLELIST);
```

```
"d3dcommon.h" lists all of existing topologies:

typedef enum D3D_PRIMITIVE_TOPOLOGY
    {
        D3D_PRIMITIVE_TOPOLOGY_UNDEFINED= 0,
        D3D_PRIMITIVE_TOPOLOGY_POINTLIST= 1,
        D3D_PRIMITIVE_TOPOLOGY_LINELIST= 2,
        D3D_PRIMITIVE_TOPOLOGY_LINESTRIP= 3,
        D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST= 4,
        D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP= 5,
```
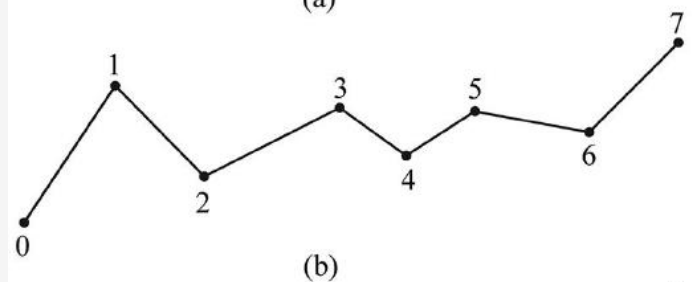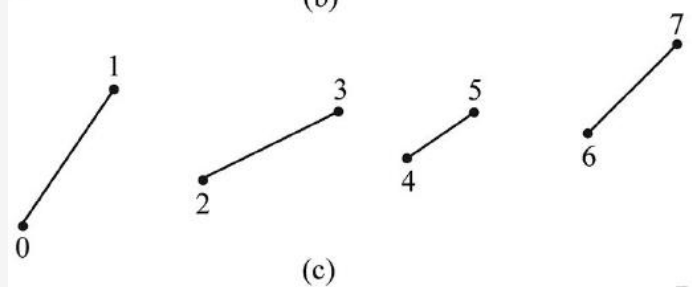
Primitive topologies
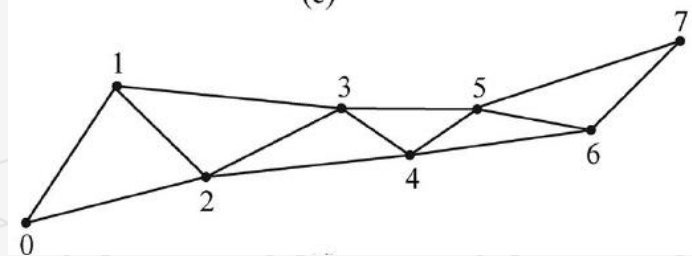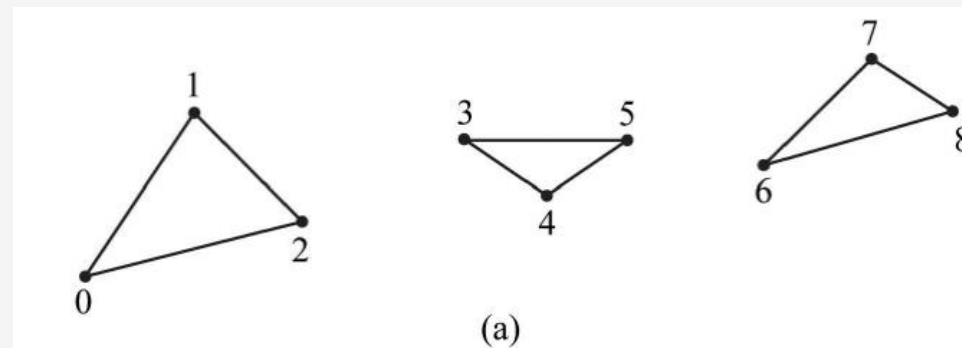

a) Point list
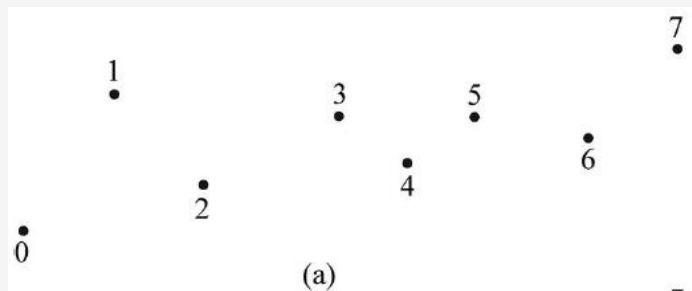
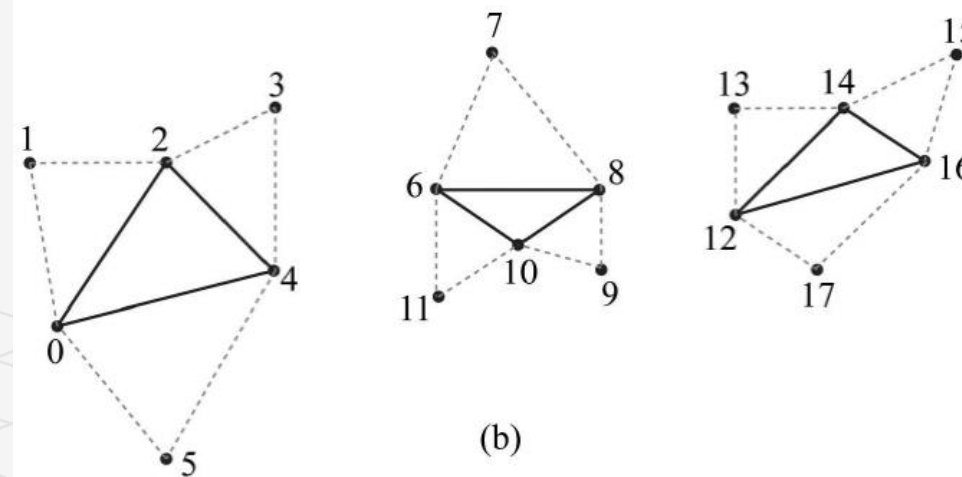b) Line strip

c) Line lists

d) Triangle strip

a) Triangle list

b) Triangle list

with adjacency

Primitive Adjacency

All Direct3D 10 and higher primitive types (except the point list) are available in two versions: one primitive type with adjacency and one primitive type without adjacency. Primitives with adjacency contain some of the surrounding vertices, while primitives without adjacency contain only the vertices of the target primitive. For example, the line list primitive (represented by the **D3D_PRIMITIVE_TOPOLOGY_LINELIST** value) has a corresponding line list primitive that includes adjacency (represented by the **D3D_PRIMITIVE_TOPOLOGY_LINELIST_ADJ** value.)

```
typedef enum D3D_PRIMITIVE_TOPOLOGY

    {
        D3D_PRIMITIVE_TOPOLOGY_UNDEFINED= 0,

        D3D_PRIMITIVE_TOPOLOGY_POINTLIST= 1,

        D3D_PRIMITIVE_TOPOLOGY_LINELIST= 2,

        D3D_PRIMITIVE_TOPOLOGY_LINESTRIP= 3,

        D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST= 4,

        D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP= 5,

        D3D_PRIMITIVE_TOPOLOGY_LINELIST_ADJ= 10,

        D3D_PRIMITIVE_TOPOLOGY_LINESTRIP_ADJ= 11,

        D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ= 12,

        D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP_ADJ= 13,
```
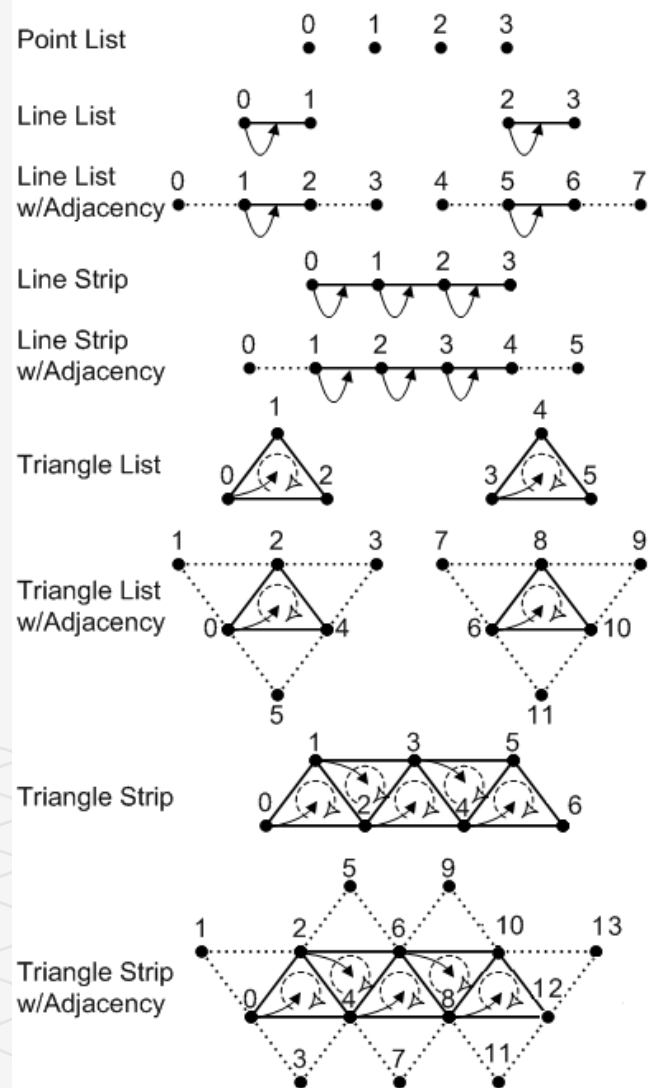
Control Point Patch List

The D3D_PRIMITIVE_TOPOLOGY_*N*_CONTROL_POINT_PATCHLIST topology type indicates that the vertex data should be interpreted as a patch lists with *N* control points. These are used in the (optional) tessellation stage of the rendering pipeline, and therefore, we will postpone a discussion of them

```
typedef enum D3D_PRIMITIVE_TOPOLOGY
    {
        D3D_PRIMITIVE_TOPOLOGY_UNDEFINED= 0,
        D3D_PRIMITIVE_TOPOLOGY_POINTLIST= 1,
        D3D_PRIMITIVE_TOPOLOGY_LINELIST= 2,
        D3D_PRIMITIVE_TOPOLOGY_LINESTRIP= 3,
        D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST= 4,
        D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP= 5,
        D3D_PRIMITIVE_TOPOLOGY_LINELIST_ADJ= 10,
        D3D_PRIMITIVE_TOPOLOGY_LINESTRIP_ADJ= 11,
        D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ= 12,
        D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP_ADJ= 13,
        D3D_PRIMITIVE_TOPOLOGY_1_CONTROL_POINT_PATCHLIST= 33,
        D3D_PRIMITIVE_TOPOLOGY_2_CONTROL_POINT_PATCHLIST= 34,
        D3D_PRIMITIVE_TOPOLOGY_3_CONTROL_POINT_PATCHLIST= 35,
```

Winding Order

Triangles are the basic building blocks for solid 3D objects. The following code shows the vertex arrays used to construct a quad and octagon using triangle lists (i.e., every three vertices form a triangle). The order in which you specify the vertices of a triangle is important and is called the *winding order*

```
Vertex quad[6] = {
v0, v1, v2, // Triangle 0
v0, v2, v3, // Triangle 1
};
Vertex octagon[24] = {
v0, v1, v2, // Triangle 0
v0, v2, v3, // Triangle 1
v0, v3, v4, // Triangle 2
v0, v4, v5, // Triangle 3
v0, v5, v6, // Triangle 4
v0, v6, v7, // Triangle 5
v0, v7, v8, // Triangle 6
v0, v8, v1 // Triangle 7
};
```
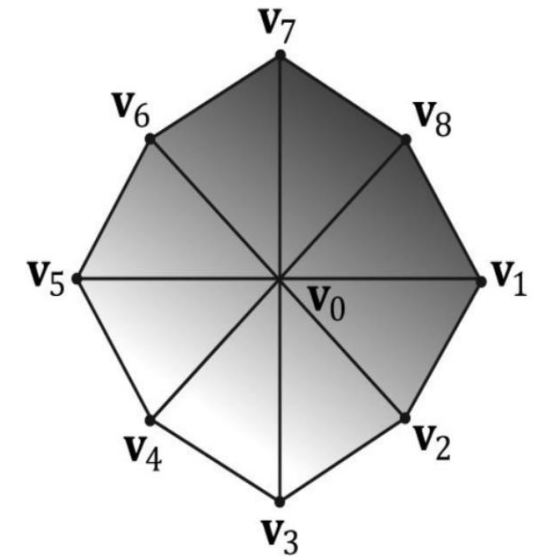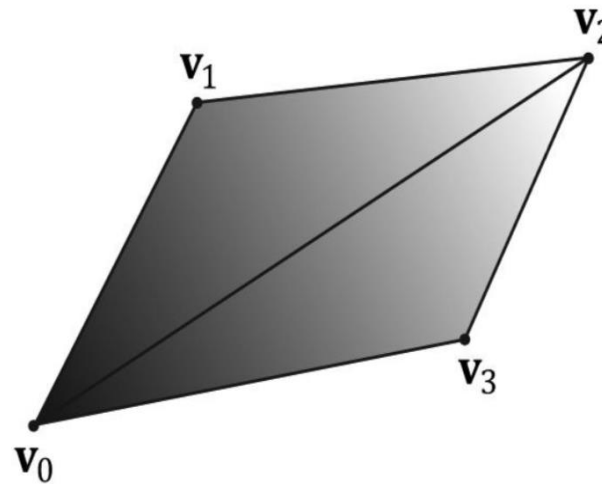
Indices

TOO MANY DUPLICATES!!!
Why is that bad?
1- Increased memory requirements
2- Increased processing by the graphics hardware

1- Indices are simply integers and do not take as much memory as a full vertex structure.
2- With good vertex cache ordering, the graphics hardware won't have to process duplicate vertices.

```
Vertex v[9] = { v0, v1, v2, v3, v4, v5, v6, v7, v8 };
UINT indexList[24] = {
0, 1, 2, // Triangle 0
0, 2, 3, // Triangle 1
0, 3, 4, // Triangle 2
0, 4, 5, // Triangle 3
0, 5, 6, // Triangle 4
0, 6, 7, // Triangle 5
0, 7, 8, // Triangle 6
0, 8, 1 // Triangle 7
};
```

Vertex Shader Stage

▪ After the primitives have been assembled, the vertices are fed into the vertex shader stage

▪ The vertex shader can be thought of as a function that inputs a vertex and outputs a vertex

▪ Many special effects can be done in the vertex shader such as transformations, lighting, and displacement mapping

▪ We have access to the input vertex data and we also can access textures and other data stored in GPU memory such as transformation matrices, and scene lights

▪ Once the vertices of the 3D model have been defined in local space, it is placed in the global scene

▪ We specify where we want the origin and axes of the local space coordinate system relative to the global scene coordinate system, then execute a change of coordinate transformation

▪ This process of changing coordinates relative to a local coordinate system into the global scene coordinate system is called the *world transform*, and the corresponding matrix is called the *world matrix*

▪ If you want to define an object directly in the world space, then you can supply an identity world matrix.

World System
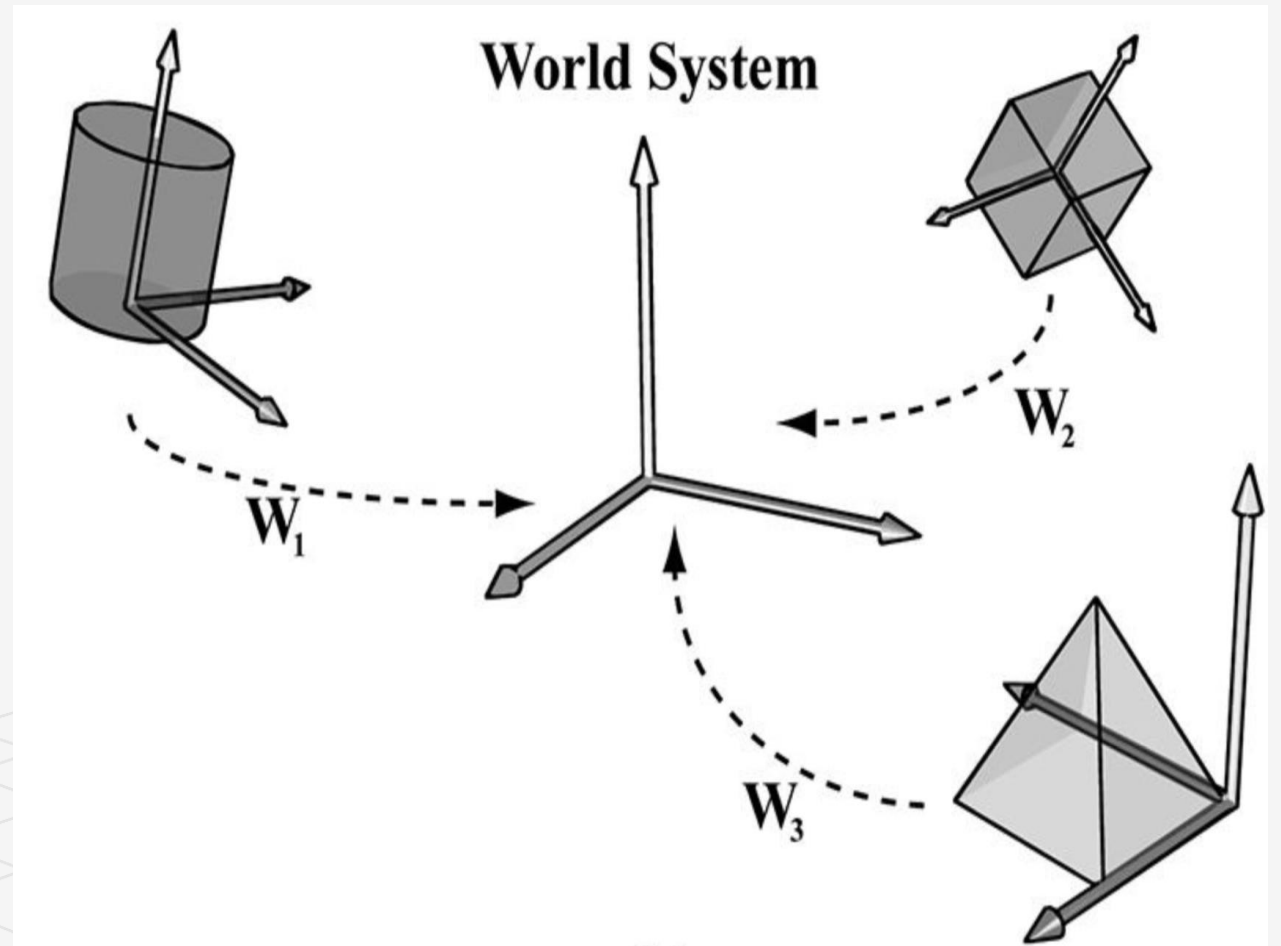
World Matrix

Change of Coordinate from local space to world space (AKA. World Matrix). A more common approach is to define **W** as a sequence of transformations, say **W** = **SRT**, the product of a scaling matrix **S** to scale the object into the world, followed by a rotation matrix **R** to define the orientation of the local space relative to the world space, followed by a translation matrix **T** to define the origin of the local space relative to the world space.

$$
\mathbf{W} = \begin{bmatrix} u_x & u_y & u_z & 0 \\ v_x & v_y & v_z & 0 \\ w_x & w_y & w_z & 0 \\ Q_x & Q_y & Q_z & 1 \end{bmatrix}
$$

Example

Suppose we have a unit square defined relative to some local space with minimum and maximum points (−0.5, 0, −0.5) and (0.5, 0, 0.5), respectively. Find the world matrix such that the square has a length of 2 in world space, the square is rotated 45° clockwise in the *xz*-plane of the world space, and the square is positioned at (10, 0, 10) in world space. We construct **S**, **R**, **T**, and **W** as follows:

$$
\mathbf{S} = \begin{bmatrix} 2 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\quad
\mathbf{R} = \begin{bmatrix} \sqrt{2}/2 & 0 & -\sqrt{2}/2 & 0 \\ 0 & 1 & 0 & 0 \\ \sqrt{2}/2 & 0 & \sqrt{2}/2 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}
\quad
\mathbf{T} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 10 & 0 & 10 & 1 \end{bmatrix}
$$

$$
\mathbf{W} = \mathbf{SRT} = \begin{bmatrix} \sqrt{2} & 0 & -\sqrt{2} & 0 \\ 0 & 1 & 0 & 0 \\ \sqrt{2} & 0 & \sqrt{2} & 0 \\ 10 & 0 & 10 & 1 \end{bmatrix}
$$
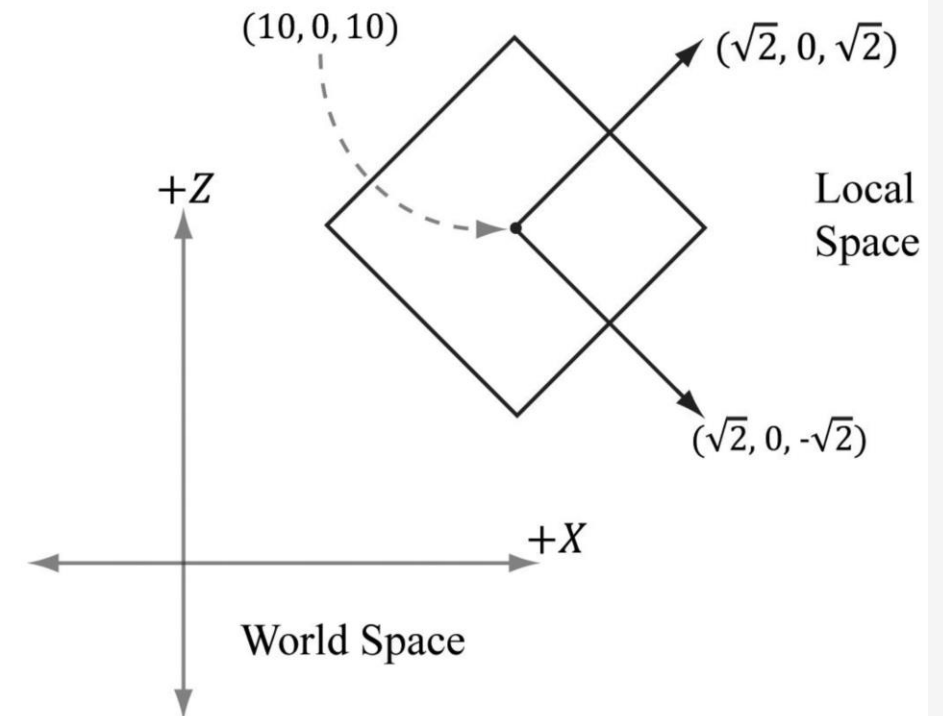
View Space

In order to form a 2D image of the scene, we must place a virtual camera in the scene. The camera specifies what volume of the world the viewer can see and thus what volume of the world we need to generate a 2D image of.

The camera sits at the origin looking down the positive z-axis, the x-axis aims to the right of the camera, and the y-axis aims above the camera. Instead of describing our scene vertices relative to the world space, it is convenient for later stages of the rendering pipeline to describe them relative to the camera coordinate system. The change of coordinate transformation from world space to view space is called the *view transform*, and the corresponding matrix is called the *view matrix*.
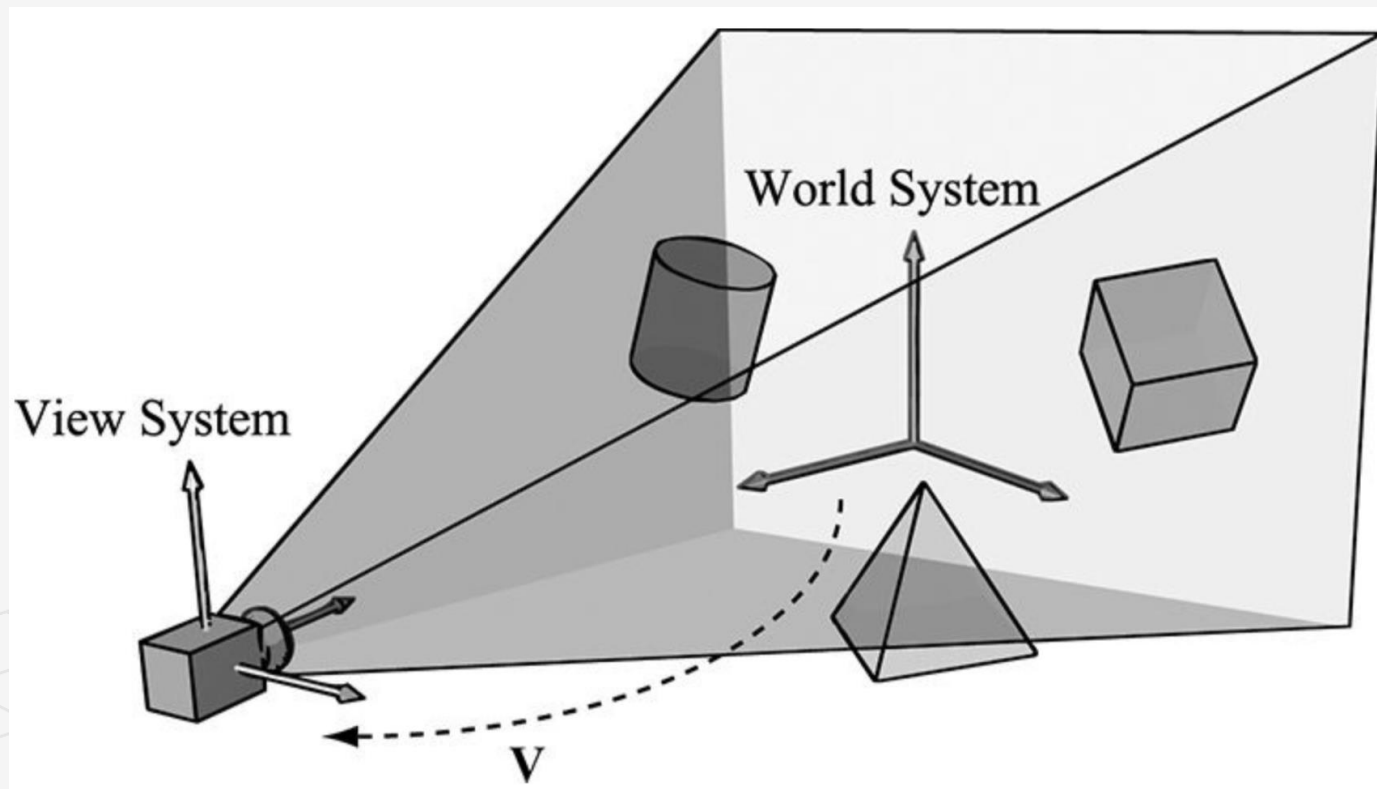
View Matrix

We want the reverse transformation from world space to view space. That reverse transformation is just given by the inverse. Thus **W**−1 transforms from world space to view space. The world coordinate system and view coordinate system generally differ by position and orientation only, so it makes intuitive sense that **W** = **RT** (i.e., the world matrix can be decomposed into a rotation followed by a translation). This form makes the inverse easier to compute:

$$V = W^{-1} = (RT)^{-1} = T^{-1}R^{-1} = T^{-1}R^{T}$$

$$= \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ -Q_x & -Q_y & -Q_z & 1 \end{bmatrix} \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} u_x & v_x & w_x & 0 \\ u_y & v_y & w_y & 0 \\ u_z & v_z & w_z & 0 \\ -Q \cdot u & -Q \cdot v & -Q \cdot w & 1 \end{bmatrix}$$

Camera Local Coordinates

Let **Q** be the position of the camera and let **T** be the target point the camera is aimed at. Furthermore, let **j** be the unit vector that describes the "up" direction of the world space. The direction the camera is looking is given by:

$$\mathbf{w} = \frac{\mathbf{T} - \mathbf{Q}}{\|\mathbf{T} - \mathbf{Q}\|}$$

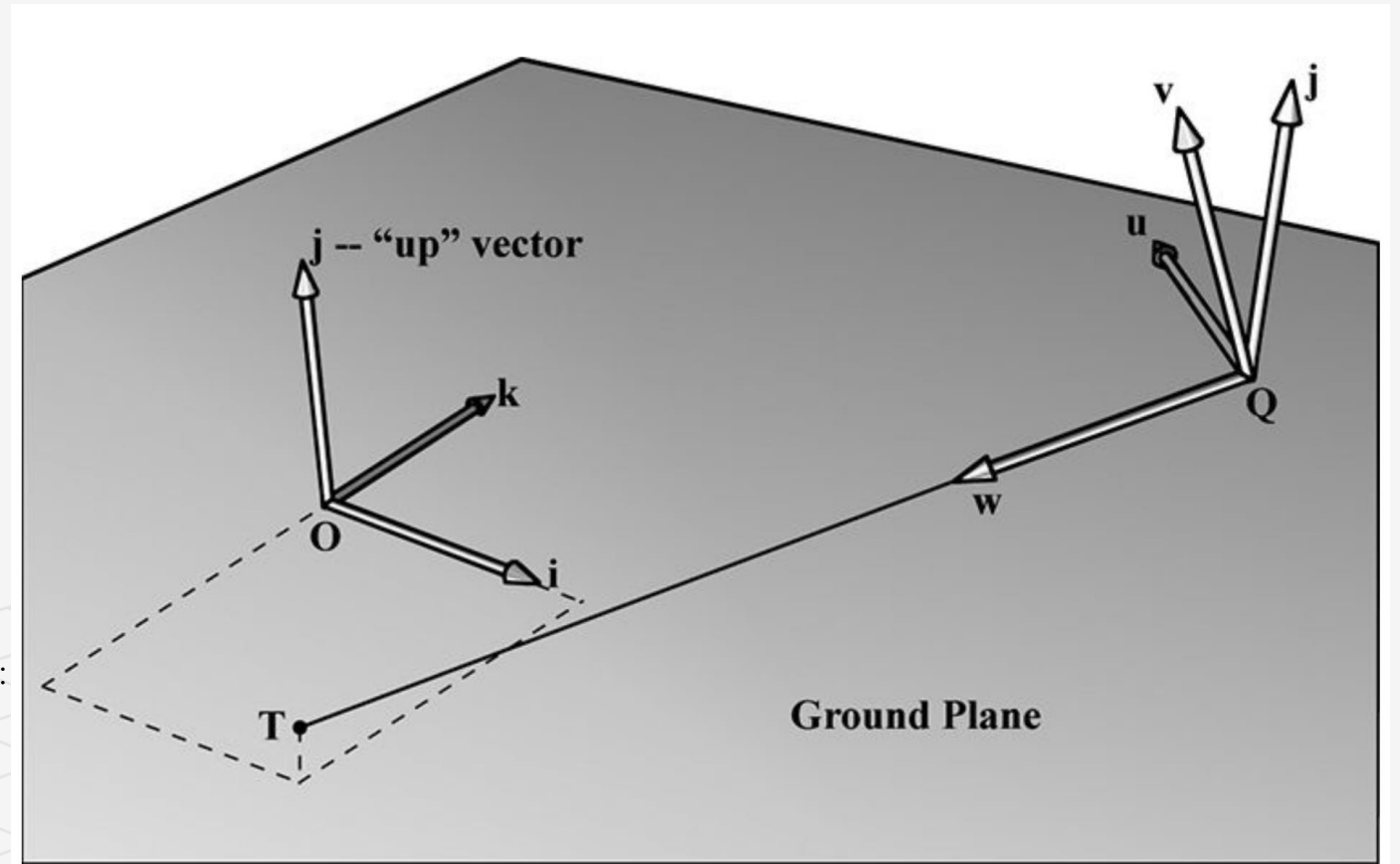This vector describes the local z-axis of the camera.

A unit vector that aims to the "right" of **w** is given by:

$$\mathbf{u} = \frac{\mathbf{j} \times \mathbf{w}}{\|\mathbf{j} \times \mathbf{w}\|}$$

This vector describes the local x-axis of the camera.

A vector that describes the local y-axis of the camera is given by:

$$\mathbf{v} = \mathbf{w} \times \mathbf{u}$$

XMMatrixLookAtLH

The DirectXMath library provides the following function for computing the view matrix based on the just described process:

```
XMMATRIX XM_CALLCONV XMMatrixLookAtLH( // Outputs view matrix V

FXMVECTOR EyePosition, // Input camera position Q

FXMVECTOR FocusPosition, // Input target point T

FXMVECTOR UpDirection); // Input world up direction j

// Build the view matrix. Look at: void BoxApp::Update(const GameTimer& gt)

    XMVECTOR pos = XMVectorSet(x, y, z, 1.0f); // camera position

    XMVECTOR target = XMVectorZero(); // target position (world centre)

    XMVECTOR up = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f); //(y-axis)

    XMMATRIX view = XMMatrixLookAtLH(pos, target, up);
```
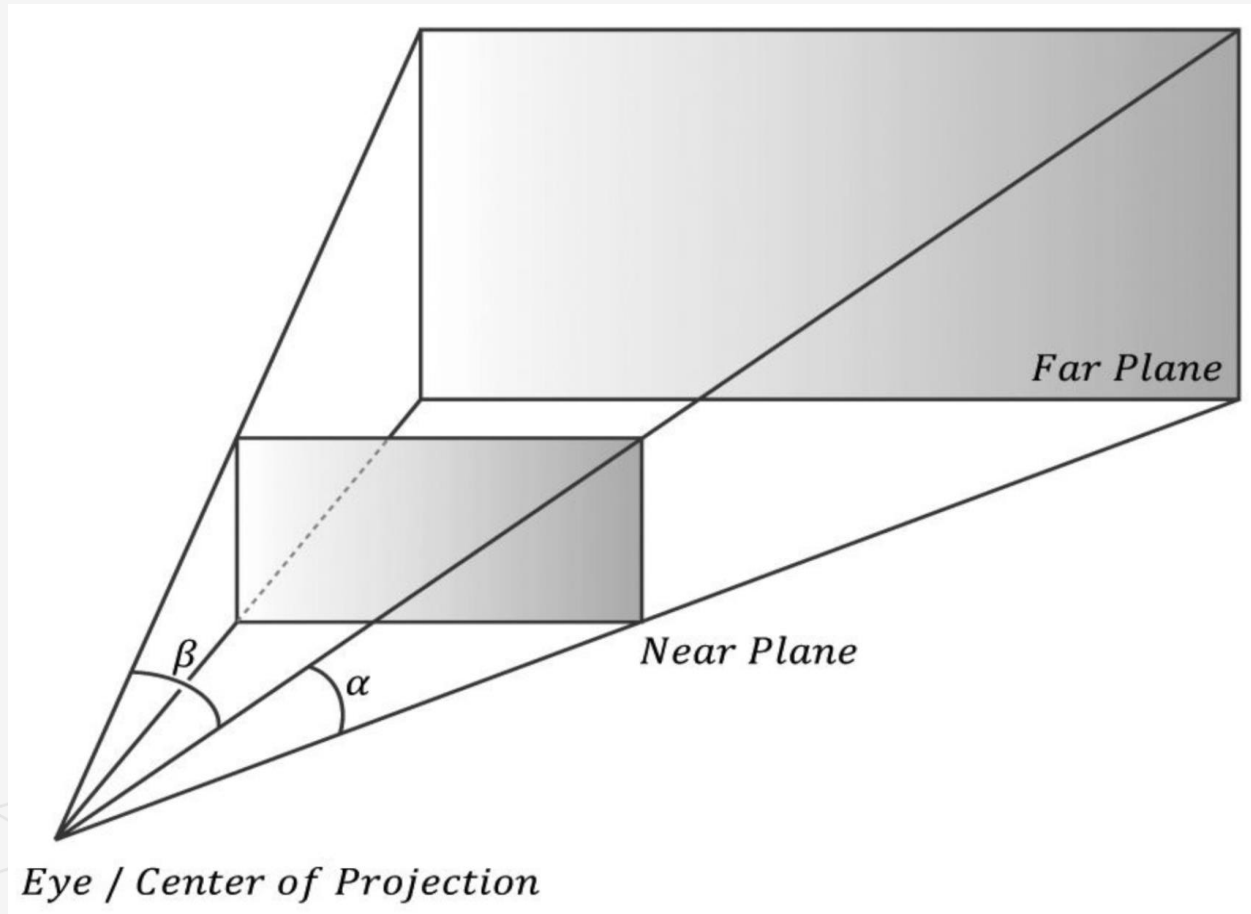
# Projection and Homogeneous Clip Space

There is another component to a camera, which is the volume of space the camera sees. This volume is described by a frustum:

Perspective projection matrix

Our next task is to project the 3D geometry inside the frustum onto a 2D projection window. The projection must be done in such a way that parallel lines converge to a vanishing point, and as the 3D depth of an object increases, the size of its projection diminishes; a perspective projection does this. We call the line from a vertex to the eye point the *vertex's line of projection*. Then we define the *perspective projection transformation* as the transformation that transforms a 3D vertex **v** to the point **v'** where its line of projection intersects the 2D projection plane; we say that **v'** is the projection of **v**. The projection of a 3D object refers to the projection of all the vertices that make up the object.
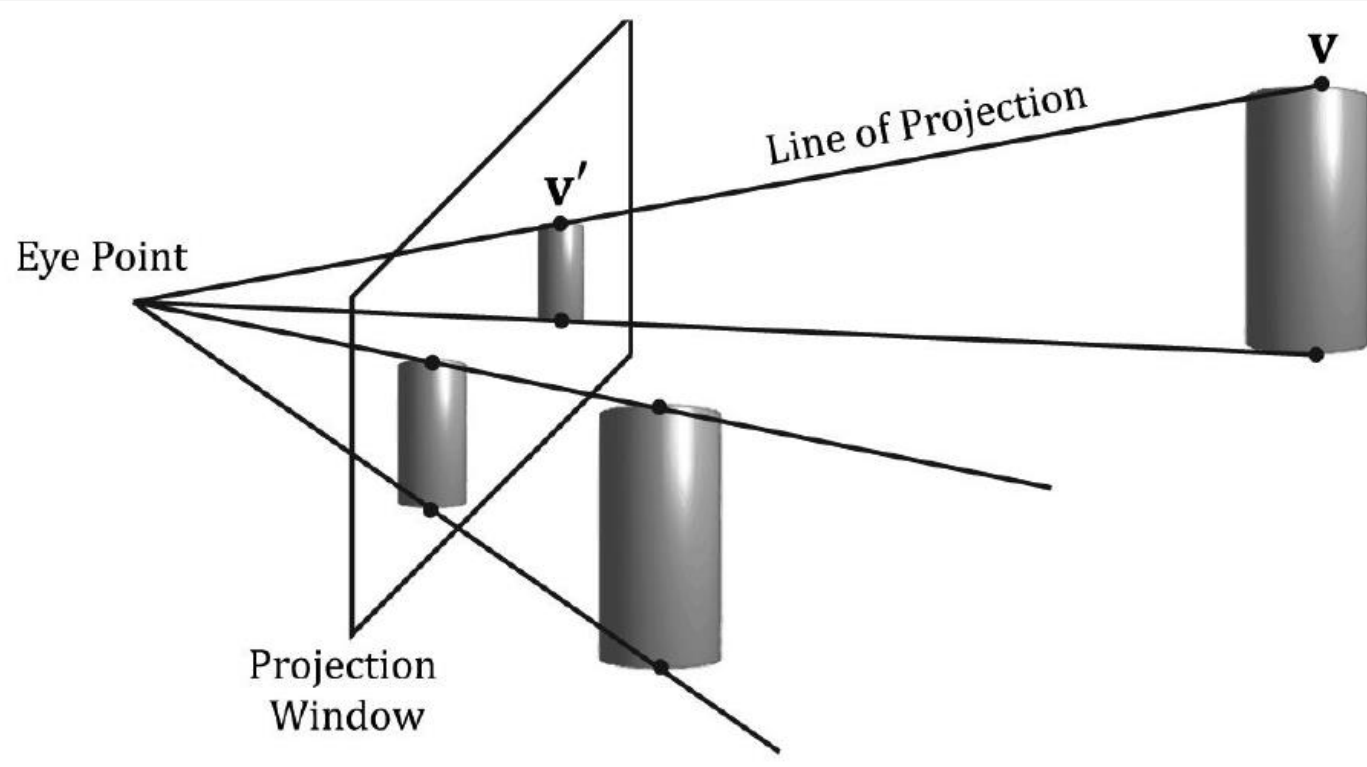
XMMatrixPerspectiveFovLH

A perspective projection matrix can be built with the following DirectX Math function:

```cpp
inline XMMATRIX XM_CALLCONV XMMatrixPerspectiveFovLH
(
    float FovAngleY,
    float AspectRatio,
    float NearZ,
    float FarZ
)
```

Field of view: 45º , Near plane at: 1 , far plane at: 1000 (these numbers are in view space)

```cpp
void BoxApp::OnResize()
{
D3DApp::OnResize();

    // The window resized, so update the aspect ratio and recompute the projection matrix.
    XMMATRIX P = XMMatrixPerspectiveFovLH(0.25f*MathHelper::Pi, AspectRatio(), 1.0f, 1000.0f);
    XMStoreFloat4x4(&mProj, P);
}
```

THE TESSELLATION STAGES

Tessellation refers to subdividing the triangles of a mesh to add new triangles. These new triangles can then be offset into new positions to create finer mesh detail.
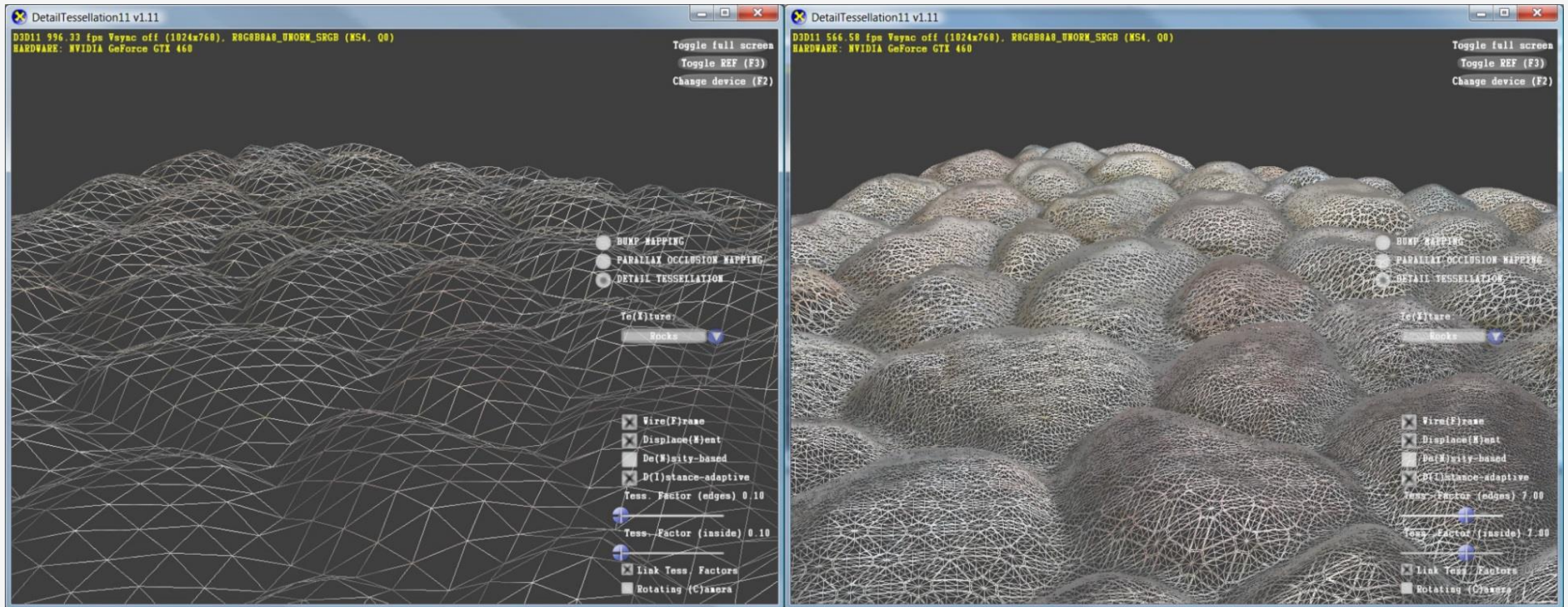
Tessellation Benefits

- We can implement a level-of-detail (LOD) mechanism, where triangles near the camera are tessellated and triangles far away from the camera are not tessellated

- We keep a simpler low-poly mesh (low triangle count) in memory, and add the extra triangles on the fly, thus saving memory

- We do operations like animation and physics on a simpler low-poly mesh, and only use the tessellated high-poly mesh for rendering

Note: The tessellation stages are new to Direct3D 11, and they provide a way to tessellate geometry on the GPU. Before Direct3D 11, if you wanted to implement a form of tessellation, it would have to be done on the CPU, and then the new tessellated geometry would have to be uploaded back to the GPU for rendering.

THE GEOMETRY SHADER STAGE

▪ The geometry shader inputs entire primitives

▪ The main advantage of the geometry shader is that it can create or destroy geometry

▪ For example, the input primitive can be expanded into one or more other primitives, or the geometry shader can choose not to output a primitive based on some condition. This is in contrast to a vertex shader, which cannot create vertices: it inputs one vertex and outputs one vertex.

▪ A common example of the geometry shader is to expand a point into a quad or to expand a line into a quad

▪ The geometry shader can stream-out vertex data into a buffer in memory, which can later be drawn. This is an advanced technique, and will be discussed later

Clipping

- *Vertex positions leaving the geometry shader must be transformed to homogeneous clip space.*

- Geometry completely outside the viewing frustum needs to be discarded, and geometry that intersects the boundary of the frustum must be clipped, so that only the interior part remains.

To clip a polygon against the frustum, we clip it against each frustum plane one-by-one

- When clipping a polygon against a plane the part in the positive half-space of the plane is kept, and the part in the negative half space is discarded
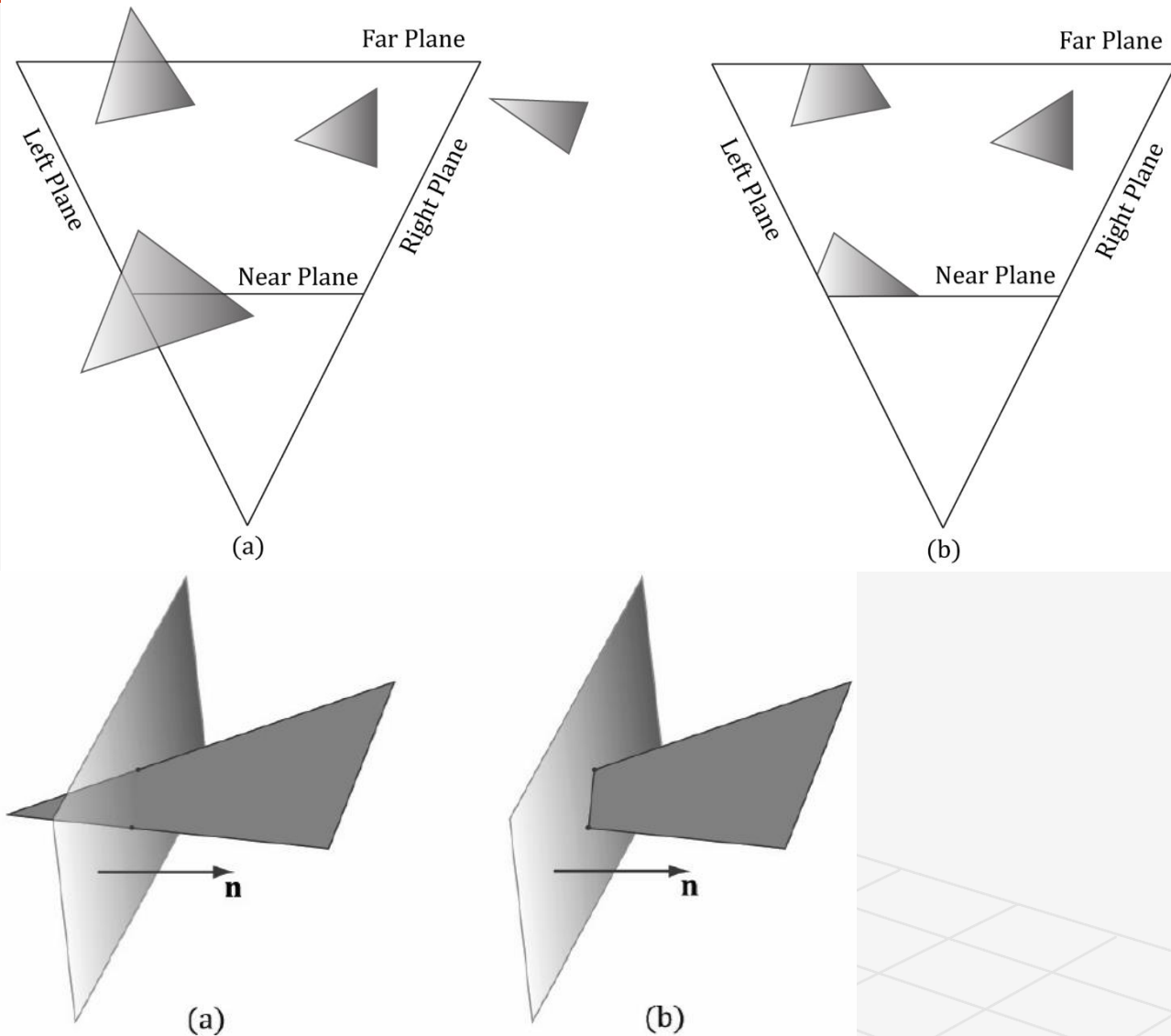
Rasterization

The main job of the rasterization stage is to compute pixel colors from the projected 3D triangles

▪ This stage can be broken down into viewport transformation, backface culling and vertex attribute interpolation.

After clipping, the hardware can do the perspective divide to transform from homogeneous clip space to normalized device coordinates (NDC). Once vertices are in NDC space, the 2D x- and y- coordinates forming the 2D image are transformed to a rectangle on the back buffer called the viewport.

After this transform, the x and y-coordinates are in units of pixels. Usually the viewport transformation does not modify the z-coordinate, as it is used for depth buffering, but it can by modifying the MinDepth and MaxDepth values of the D3D12_VIEWPORT structure. The MinDepth and MaxDepth values must be between 0 and 1.

Backface culling

A triangle has two sides. To distinguish between the two sides we use the following convention. If the triangle vertices are ordered **v**0, **v**1, **v**2 then we compute the triangle normal **n.** The side the normal vector emanates from is the *front side* and the other side is the *back side*. The left triangle is front-facing from our viewpoint, and the right triangle is back-facing from our viewpoint. *Backface culling* refers to the process of discarding backfacing triangles from the pipeline. This can potentially reduce the amount of triangles that need to be processed by half.

$$e_0 = v_1 - v_0$$

$$e_1 = v_2 - v_0$$

$$n = \frac{e_0 \times e_1}{e_0 \times e_1}$$
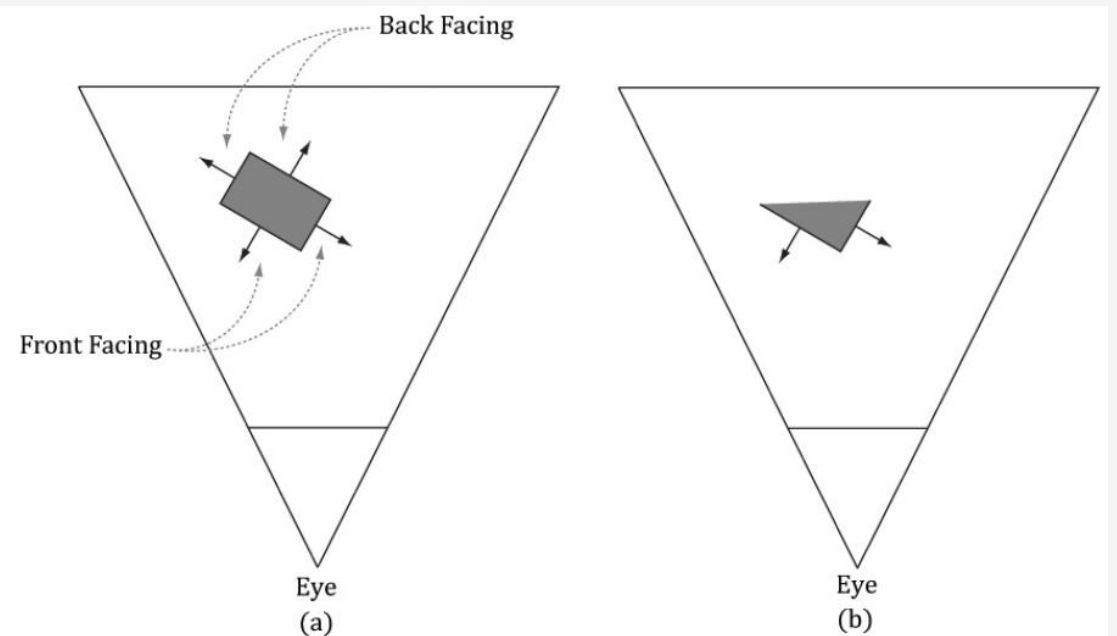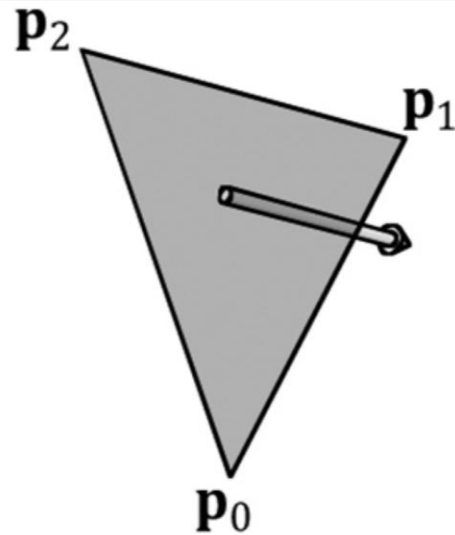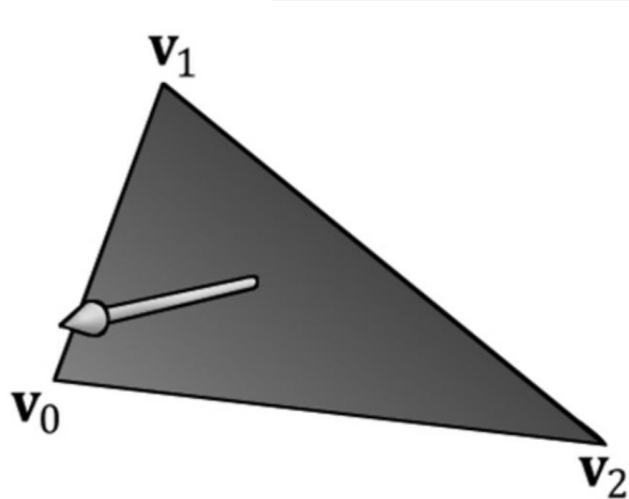
Vertex Attribute Interpolation

In addition to position, we can attach attributes to vertices such as colors, normal vectors, and texture coordinates.

After the viewport transform, these attributes need to be interpolated for each pixel covering the triangle. In addition to vertex attributes, vertex depth values need to get interpolated so that each pixel has a depth value for the depth buffering algorithm.

The vertex attributes are interpolated in screen space in such a way that the attributes are interpolated linearly across the triangle in 3D space. This requires the so called *perspective correct interpolation*. Essentially, interpolation allows us to use the vertex values to compute values for the interior pixels.

$$\mathbf{p}(s,t) = \mathbf{v}_0 + s(\mathbf{v}_1 - \mathbf{v}_0) + t(\mathbf{v}_2 - \mathbf{v}_0)$$

$$\text{for } s \geq 0, t \geq 0, s + t \leq 1$$
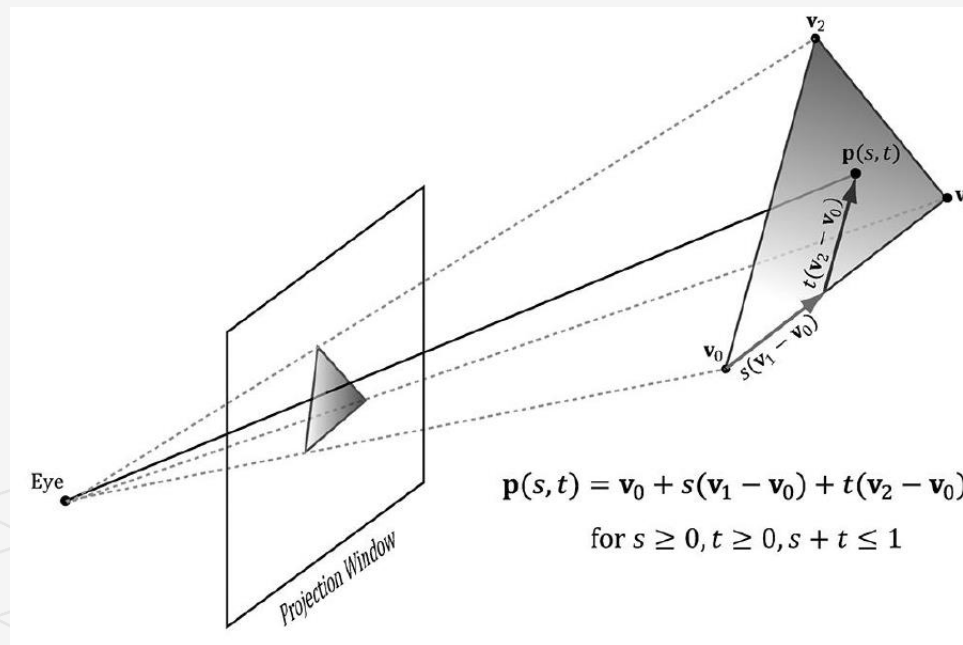
THE PIXEL SHADER STAGE

▪ Pixel shaders are programs we write that are executed on the GPU

▪ A pixel shader is executed for each pixel fragment and uses the interpolated vertex attributes as input to compute a color.

▪ A pixel shader can be as simple as returning a constant color, to doing more complicated things like per-pixel lighting, reflections and shadowing effects.

Output Merger Stage

- After pixel fragments have been generated by the pixel shader, they move onto the output merger (OM) stage

- In this stage, some pixel fragments may be rejected (e.g., from the depth or stencil buffer tests).

- Pixel fragments that are not rejected are written to the back buffer

- Blending is also done in this stage, where a pixel may be blended with the pixel currently on the back buffer instead of overriding it.

- Some special effects like transparency are implemented with blending