II. Direct3D Foundations
5. The Rendering Pipeline

Game Graphic Programming

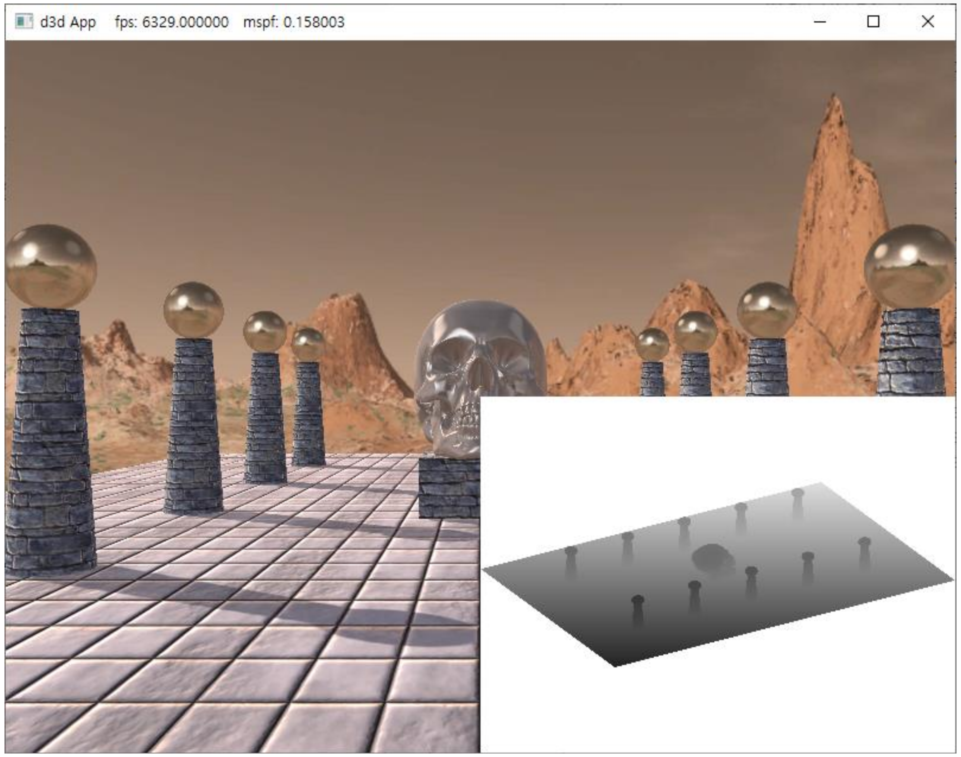
Kyung Hee University

Software Convergence

Prof. Daeho Lee

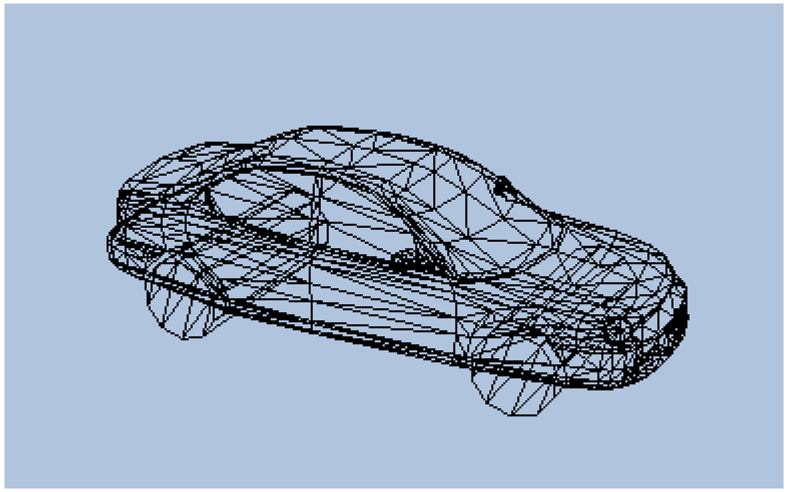
Drawing on a 2D Plane

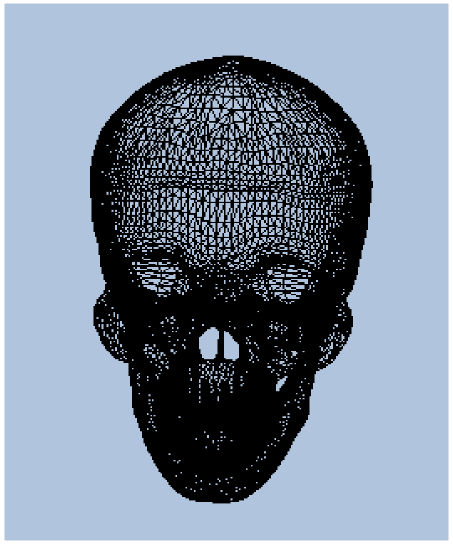
- Human viewing system
  - Parallel lines of vision converge to a vanishing point.
- 3D effects
  - Lighting/shading
  - shadow



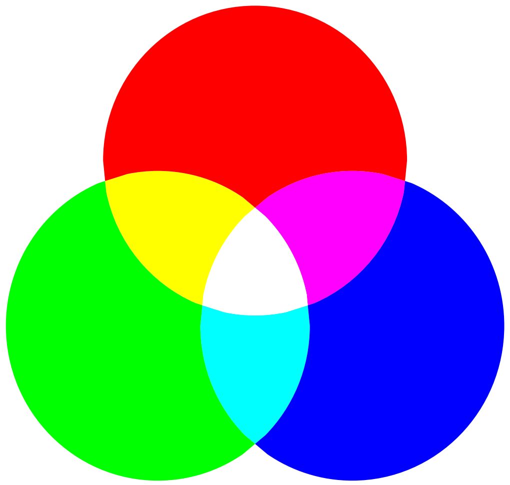
Triangle Mesh

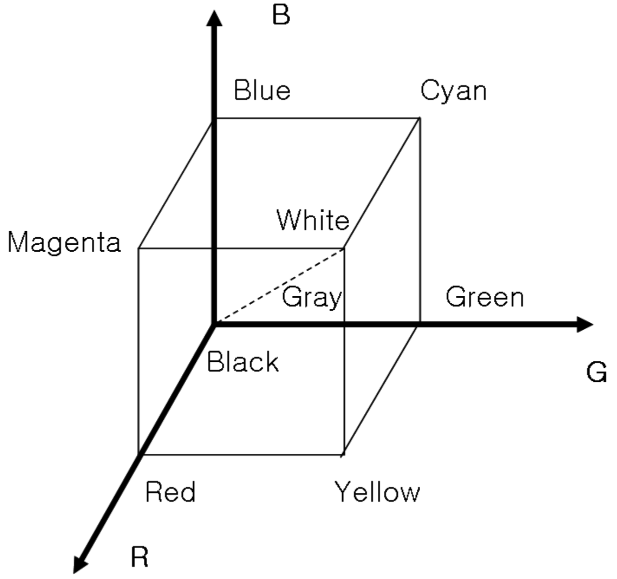
- Triangle mesh
  - It would be extremely cumbersome to manually list the triangles of a 3D model.
  - For all but the simplest models, special 3D applications called 3D modelers are used to generate and manipulate 3D objects.
  - Examples of popular modelers used for game development are 3D Studio Max (https://usa.autodesk.com/3ds-max/), LightWave 3D (https://www.lightwave3d.com/), Blender (https://www.blender.org/), …





RGB





Color Operations

- Some vector operations also apply to color vectors. For example, we can add color vectors to get new colors:

  $(0.0, 0.5, 0) + (0, 0.0, 0.25) = (0.0, 0.5, 0.25)$

- 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:

  $0.5(1, 1, 1) = (0.5, 0.5, 0.5)$

- Color vectors do get their own special color operation called modulation or componentwise multiplication. It is defined as:

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

- When doing the color operation, it is possible that your color components go outside the [0, 1] interval;

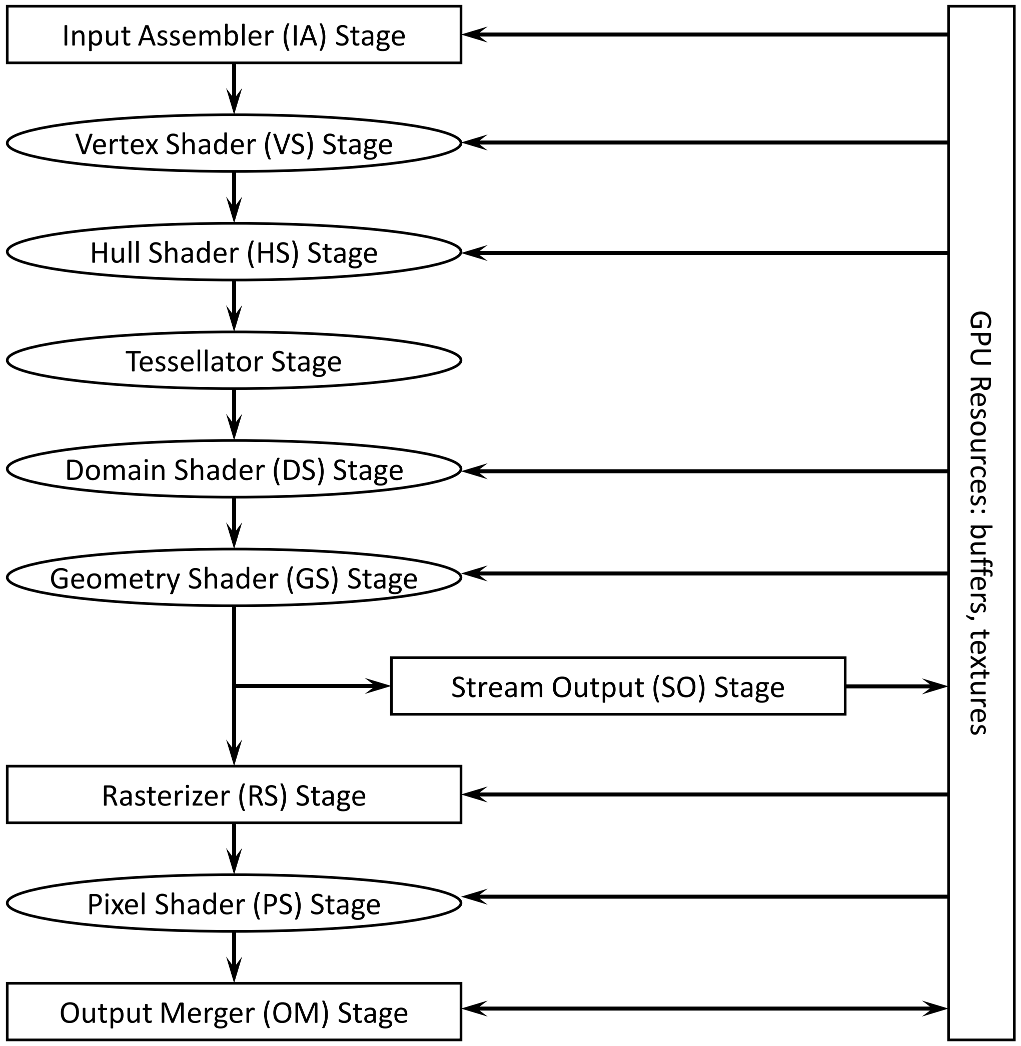
  $(1, 0.1, 0.6) + (0, 0.3, 0.5) = (1, 0.4, 1.1) \rightarrow (1, 0.4, 1)$

128-Bit/32-Bit Color

- 128-bit color
  - Alpha component
    - Opacity of color
  - 4D color vector (r, g, b, a), $0 \le r,g,b,a \le 1$
  - 32-bit (single precision floating)×4 (r,g,b,a)

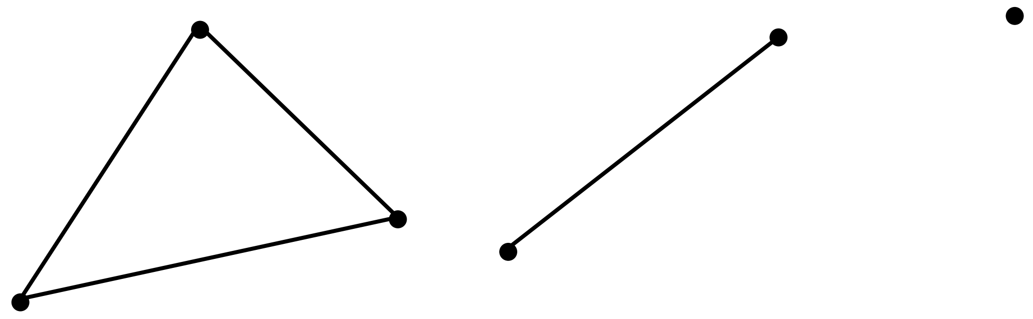
- 32-bit color
  - [0,255]×4
  - 8-bit×4 (r,g,b,a)
  -
```
struct XMCOLOR {
  union  {
    struct {
      uint8_t b; // Blue
      uint8_t g; // Green
      uint8_t r; // Red
      uint8_t a; // Alpha, msb (most significant bit)
      // [31] aaaaaaaa rrrrrrrr gggggggg bbbbbbbb [0]
    };
    uint32_t c;
  };
```

Overview of the Rendering Pipeline



Vertices

- Input assembler stage
  - The input assembler (IA) stage reads geometric data (vertices and indices) from memory and uses it to assemble geometric primitives (e.g., triangles, lines).
  - Indices are covered in a later subsection, but briefly, they define how the vertices should be put together to form the primitives.
- The vertices of a triangle are where two edges meet; the vertices of a line are the endpoints; for a single point, the point itself is the vertex.



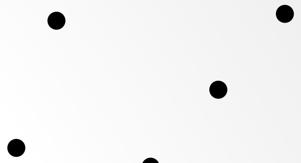
Primitive Topology (1)

- Vertices are bound to the rendering pipeline in a special Direct3D data structure called a **vertex buffer**.
- **`void IASetPrimitiveTopology( [in] D3D12_PRIMITIVE_TOPOLOGY PrimitiveTopology);`**
  - It binds information about the primitive type, and data order that describes input data for the input assembler stage.

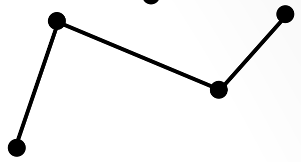
```
typedef enum D3D_PRIMITIVE_TOPOLOGY   {
  D3D_PRIMITIVE_TOPOLOGY_UNDEFINED   = 0,
  D3D_PRIMITIVE_TOPOLOGY_POINTLIST   = 1,
  D3D_PRIMITIVE_TOPOLOGY_LINELIST    = 2,
  D3D_PRIMITIVE_TOPOLOGY_LINESTRIP   = 3,
  D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST        = 4,
  D3D_PRIMITIVE_TOPOLOGY_TRIANGLESTRIP       = 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_32_CONTROL_POINT_PATCHLIST = 64,
}     D3D_PRIMITIVE_TOPOLOGY;
```

Primitive Topology (2)

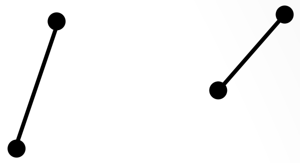
- Point list
  - Every vertex in the draw call is drawn as an individual point.



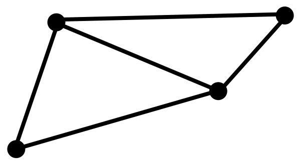
- Line strip
  - The vertices in the draw call are connected to form lines.



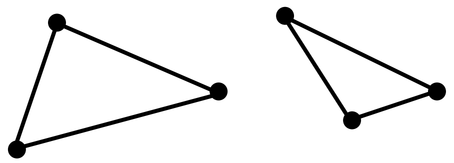
- Line list
  - Every two vertices in the draw call forms an individual line.



- Triangle strip
  - Vertices are shared between adjacent triangles, and n vertices induce n-2 triangles.

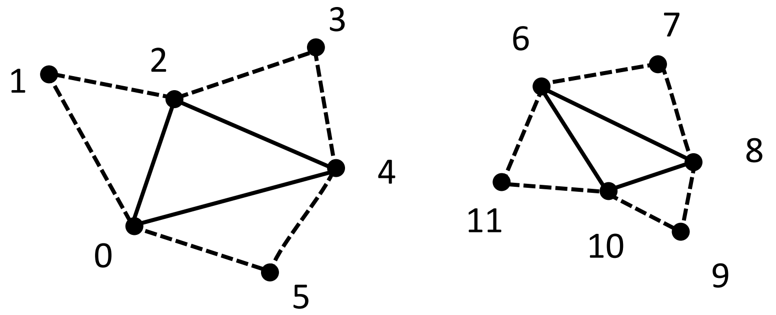


- Triangle list
  - Every three vertices in the draw call forms an individual triangle.



Primitive Topology (3)

- Primitives with adjacency
  - A triangle list with adjacency is where, for each triangle, you also include its three neighboring triangles called adjacent triangles.
  - This is used for the geometry shader.
    - **`D3D_PRIMITIVE_TOPOLOGY_TRIANGLELIST_ADJ`**



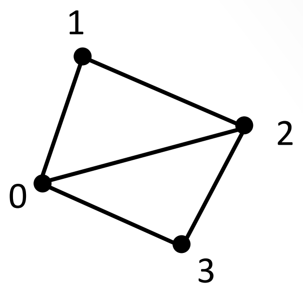
- Control point patch list
  - **`D3D_PRIMITIVE_TOPOLOGY_N_CONTROL_POINT_PATCHLIST`**
    - The vertex data should be interpreted as a patch list with N control points.
    - This is used in the (optional) tessellation stage of the rendering pipeline.

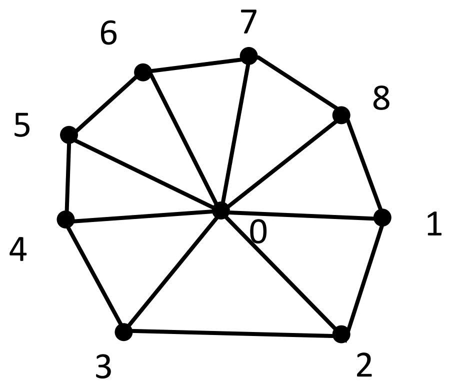
Indices (1)

- Vertex index

  -
    ```
    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
    };
    ```
  - There are two reasons why we do not want to duplicate vertices:
    - Increased memory requirements. (Why store the same vertex data more than once?)
    - Increased processing by the graphics hardware. (Why process the same vertex data more than once?)





Indices (2)

-
  ```
  Vertex v[4] = {v0, v1, v2, v3};
  UINT indexList[6] = {
     0, 1, 2, // Triangle 0
     0, 2, 3  // Triangle 1
  };
  ```

-
  ```
  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
  };
  ```

Indices (3)

- /Chapter 8 Lighting/LitColumns/Models
  - car.txt

```
VertexCount: 1860
TriangleCount: 1850
VertexList (pos, normal)
{
        -1.65064 -0.796365 -5.5759 -0.435406 0.229521 -0.870484
        -1.78981 -0.753561 -5.4874 -0.480737 0.260316 -0.837334
        -1.58193 -0.589092 -5.55562 -0.396697 0.440842 -0.805164
…
}
TriangleList
{
        0 1 2
        1 3 2
        3 4 2
…
}
```

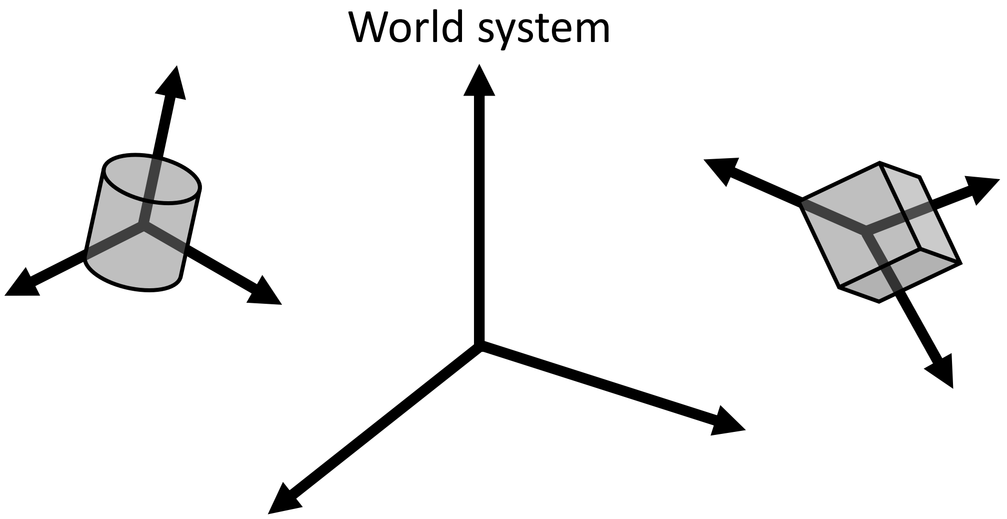
```
struct Vertex{
  DirectX::XMFLOAT3 Pos;
  DirectX::XMFLOAT3 Normal;
};
vertices[i].Pos.x vertices[i].Pos.y vertices[i].Pos.z
vertices[i].Normal.x vertices[i].Normal.y vertices[i].Normal.z
```

The Vertex Shader Stage

- The vertex-shader (VS) stage processes vertices from the input assembler, performing per-vertex operations such as transformations, skinning, morphing, and per-vertex lighting.

Local Space and World Space (1)

- The vertices of each object are defined with coordinates relative to their own **local space (local coordinate system)**.
- We define the position and orientation of each local coordinate system relative to the **world space (global scene coordinate system)** based on where we want the object in the scene.



Local Space and World Space (2)

- Defining each model relative to its own local coordinate system has several advantages:
  - 1. It is easier. For instance, usually in local space the object will be centered at the origin and symmetrical with respect to one of the major axes.
  - 2. The object may be reused across multiple scenes, in which case it makes no sense to hardcode the object's coordinates relative to a particular scene.
  - 3. Finally, sometimes we draw the same object more than once in a scene, but in different positions, orientations, and scales. It would be wasteful to duplicate the object's vertex and index data for each instance. Instead, we store a single copy of the geometry (i.e., vertex and index lists) relative to its local space. Then, we draw the object several times, but each time with a different world matrix to specify the position, orientation, and scale of the instance in the world space. This is called **instancing**.

Local Space and World Space (3)

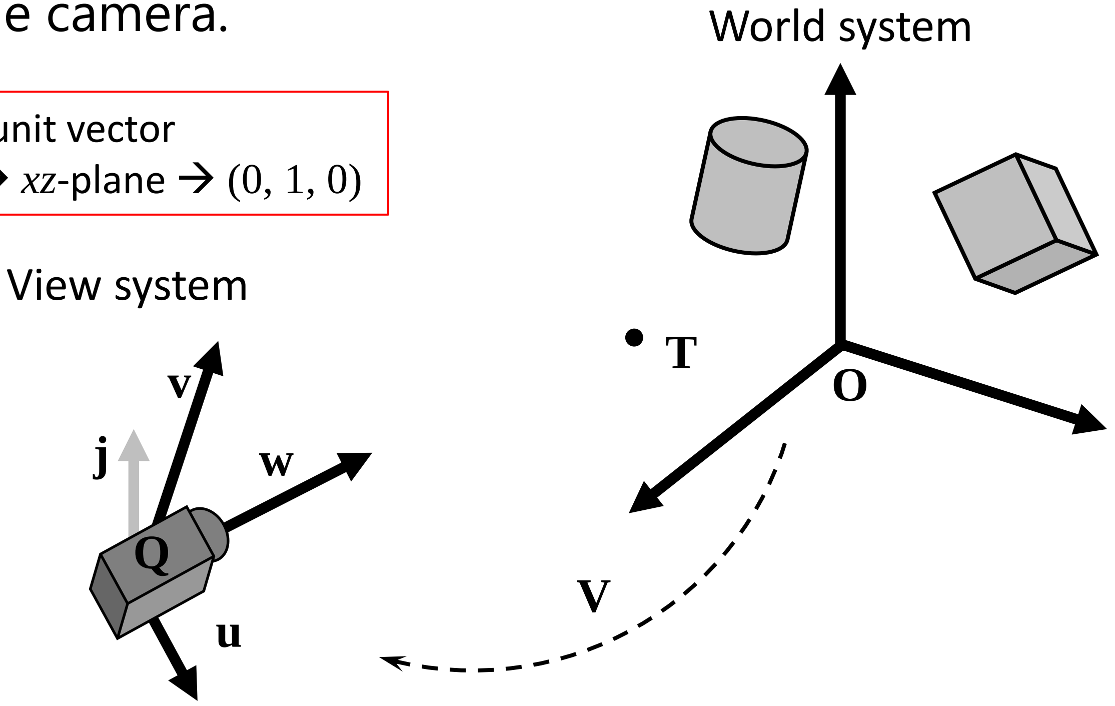
- World matrix
  - **W**: scaling (**S**), rotation (**R**), translation (**T**)

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

View Space (1)

- 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. Let us attach a local coordinate system (called view space, eye space, or camera space) to the camera.

j: up direction, unit vector
Ground plane → *xz*-plane → (0, 1, 0)



View Space (2)

- The world coordinate system and view coordinate system generally differ by position and orientation only.
  - **W**: change of coordinate matrix from view space to world space

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

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

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

$$\mathbf{w} = \frac{\mathbf{T}-\mathbf{Q}}{\|\mathbf{T}-\mathbf{Q}\|} \qquad \mathbf{u} = \frac{\mathbf{j}\times\mathbf{w}}{\|\mathbf{j}\times\mathbf{w}\|} \qquad \mathbf{v} = \mathbf{w}\times\mathbf{u}$$

View Space (3)

LH: left-handed coordinate system

- View matrix

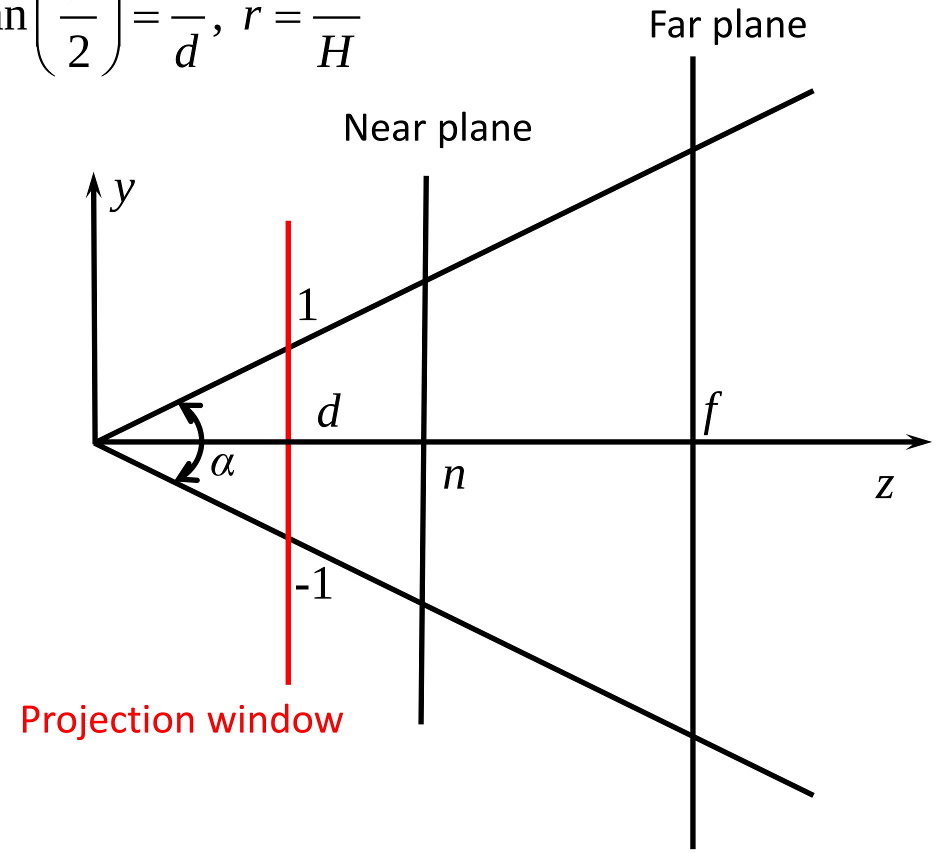
  -
    ```
    XMMATRIX XM_CALLCONV XMMatrixLookAtLH(
      FXMVECTOR EyePosition,   // Input camera position Q
      XMVECTOR FocusPosition,  // Input target point T
      FXMVECTOR UpDirection);  // Input world up direction j
    ```
    - Usually the world's y-axis corresponds to the "up" direction, so the "up" vector is usually always j = (0, 1, 0).
    - As an example, suppose we want to position the camera at the point (5, 3, −10) relative to the world space, and have the camera look at the origin of the world (0, 0, 0). We can build the view matrix by writing:

      ```
      XMVECTOR pos = XMVectorSet(5, 3, -10, 1.0f);
      XMVECTOR target = XMVectorZero();
      XMVECTOR up = XMVectorSet(0.0f, 1.0f, 0.0f, 0.0f);
      XMMATRIX V = XMMatrixLookAtLH(pos, target, up)
      ```

Projection and Homogeneous Clip Space (1)

- Defining a frustum

$$\tan\left(\frac{\alpha}{2}\right)=\frac{1}{d},\ r=\frac{W}{H}$$



$$y_p : d = y : z$$

$$y_p = \frac{dy}{z} = \frac{y}{z\tan\left(\frac{\alpha}{2}\right)}$$

$$x_p : d = y : z$$

$$x_p = \frac{dx\left(H/W\right)}{z} = \frac{x}{rz\tan\left(\frac{\alpha}{2}\right)}$$

$$\mathbf{P}=\begin{pmatrix} \frac{1}{r\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 \\ 0 & 0 & A & 1 \\ 0 & 0 & B & 0 \end{pmatrix}$$

Projection and Homogeneous Clip Space (2)

$$z_p = \frac{Az+B}{z} = A + \frac{B}{z}$$

z range → [n, f] → [0, 1]

$$1 = A + \frac{B}{f},\ 0 = A + \frac{B}{n}$$

$$1 = \frac{B}{f} - \frac{B}{n} = \frac{B(n-f)}{nf}$$

$$B = \frac{-nf}{f-n}$$

$$A = -\frac{-nf}{f-n}\frac{1}{n} = \frac{f}{f-n}$$

$$\mathbf{P} = \begin{pmatrix} \frac{1}{r\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan\left(\frac{\alpha}{2}\right)} & 0 & 0 \\ 0 & 0 & \frac{f}{f-n} & 1 \\ 0 & 0 & \frac{-nf}{f-n} & 0 \end{pmatrix}$$

Projection and Homogeneous Clip Space (3)

- Projection matrix
  -
    ```
    XMMATRIX XM_CALLCONV XMMatrixPerspectiveFovLH(
      float FovAngleY,// vertical field of view angle in radians
      float Aspect,   // aspect ratio = width / height
      float NearZ,    // distance to near plane
      float FarZ);    // distance to far plane
    ```
  - We specify a 45° vertical field of view, a near plane at z = 1 and a far plane at z = 1000 (these lengths are in view space).
    -
      ```
      XMMATRIX P = XMMatrixPerspectiveFovLH(0.25f*XM_PI, AspectRatio(),
      1.0f, 1000.0f);
      ```
    - The aspect ratio is taken to match our window aspect ratio:
    -
      ```
      float D3DApp::AspectRatio()const {
        return static_cast<float>(mClientWidth) / mClientHeight;
      }
      ```

Tessellation

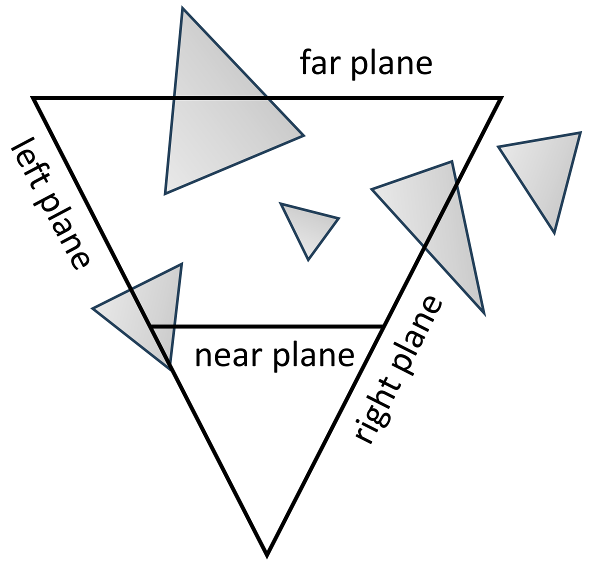
- Tessellation refers to subdividing the triangles of a mesh to add new triangles.
- There are a number of benefits to tessellations:
    - 1. We can implement a level-of-detail (LOD) mechanism, where triangles near the camera are tessellated to add more detail, and triangles far away from the camera are not tessellated. In this way, we only use more triangles where the extra detail will be noticed.
    - 2. We keep a simpler low-poly mesh (low-poly means low triangle count) in memory, and add the extra triangles on the fly, thus saving memory.
    - 3. We do operations like animation and physics on a simpler low-poly mesh, and only use the tessellated high-poly mesh for rendering.

Geometry Shader

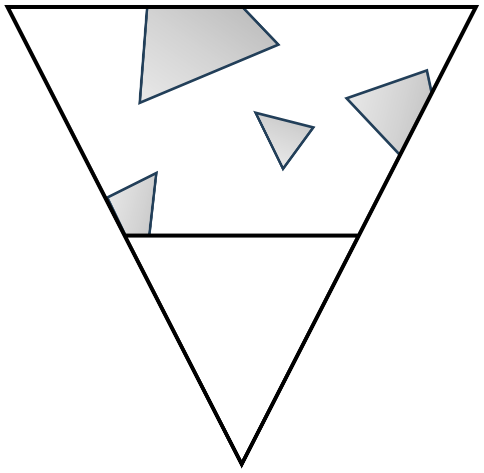
- The geometry shader (GS) stage runs application-specified shader code with vertices as input and the ability to generate vertices on output.
- Unlike vertex shaders, which operate on a single vertex, the geometry shader's inputs are the vertices for a full primitive (two vertices for lines, three vertices for triangles, or single vertex for point).
- The geometry shader stage is optional.

Clipping (1)

- 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.



before clipping

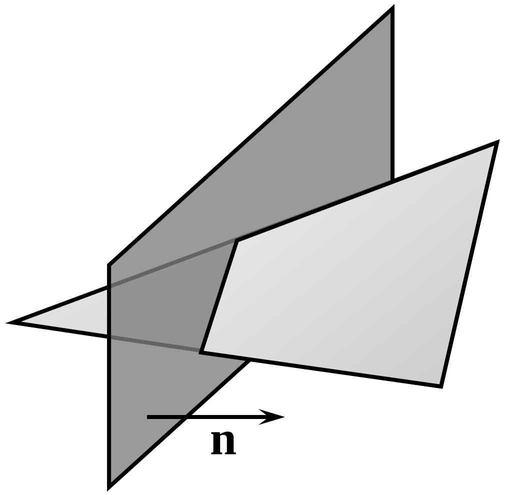


after clipping

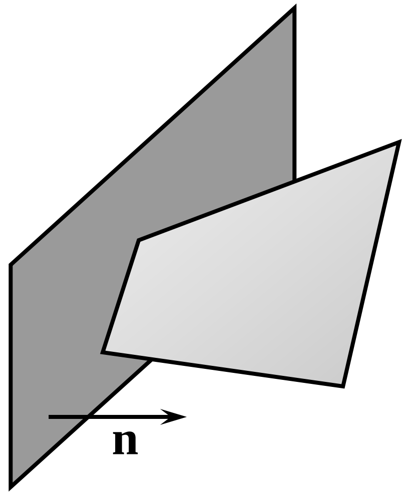
We can think of the frustum as being the region bounded by six planes: the top, bottom, left, right, near, and far planes.

Clipping (2)

- 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.



before clipping



after clipping

Viewport Transform

- 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 (1)

- A triangle has two sides. To distinguish between the two sides we use the following convention. If the triangle vertices are ordered $\mathbf{v}_0$, $\mathbf{v}_1$, $\mathbf{v}_2$ then we compute the triangle normal $\mathbf{n}$ like so:

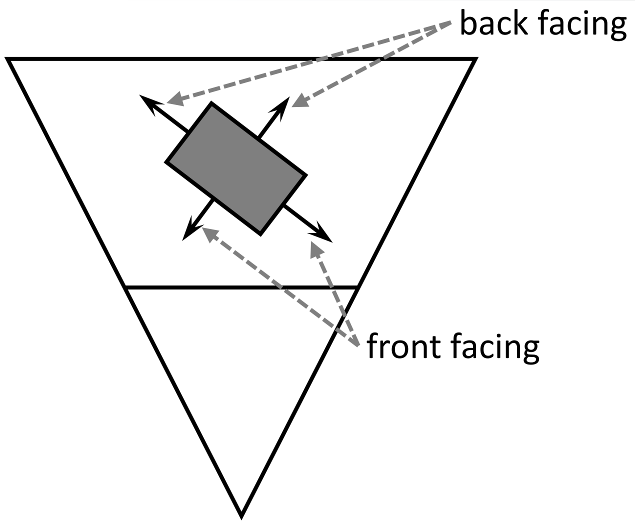
$$\mathbf{e}_0 = \mathbf{v}_1 - \mathbf{v}_0$$

$$\mathbf{e}_1 = \mathbf{v}_2 - \mathbf{v}_0$$

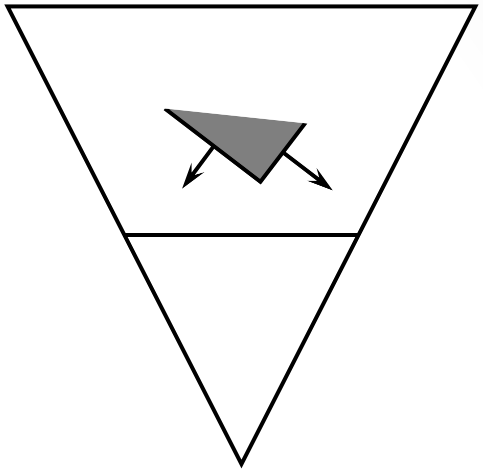
$$\mathbf{n} = \frac{\mathbf{e}_0 \times \mathbf{e}_1}{\|\mathbf{e}_0 \times \mathbf{e}_1\|}$$

- The side the normal vector emanates from is the front side and the other side is the back side.
- We say that a triangle is front-facing if the viewer sees the front side of a triangle, and we say a triangle is back-facing if the viewer sees the back side of a triangle.
- By default, Direct3D treats triangles with a clockwise winding order (with respect to the viewer) as front-facing, and triangles with a counterclockwise winding order (with respect to the viewer) as back-facing.

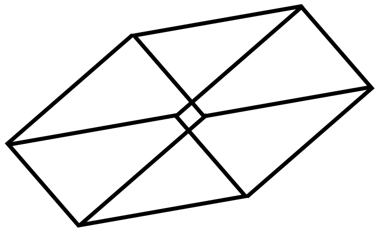
Backface Culling (2)

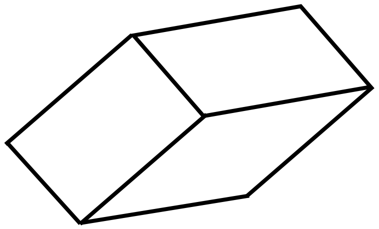


before culling



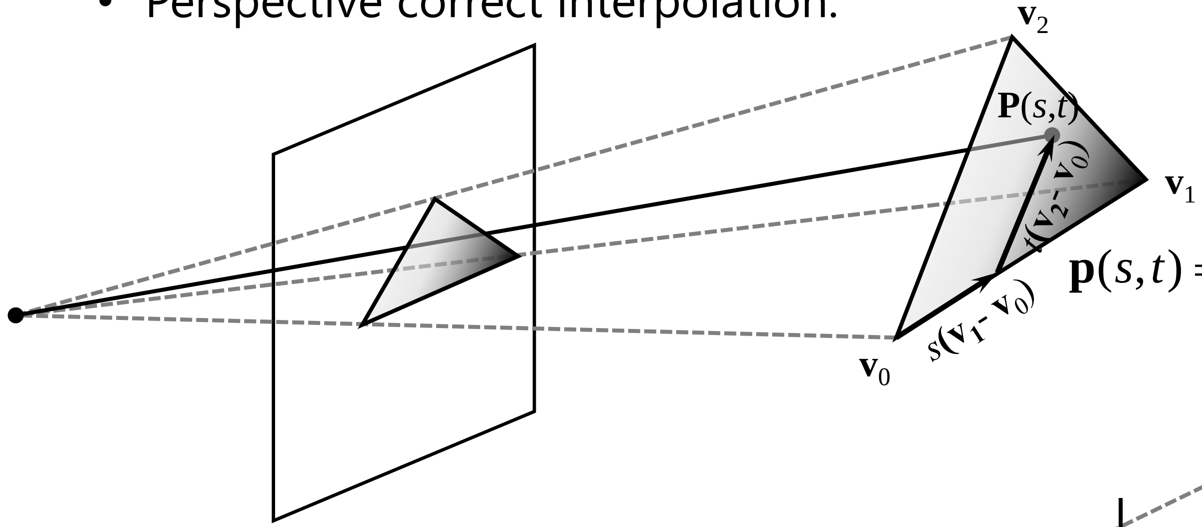
after culling



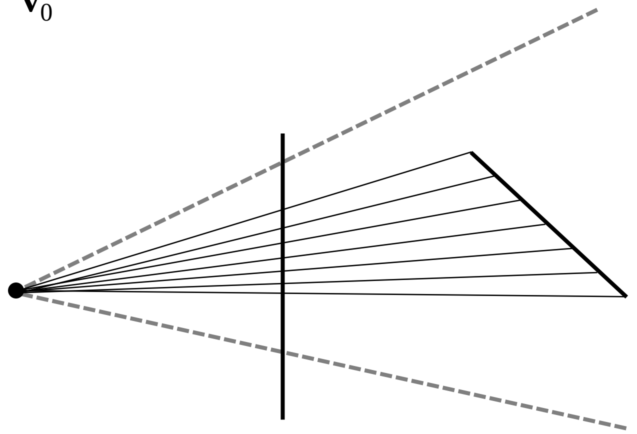


Vertex Attribute Interpolation

- After the viewport transform, these attributes need to be interpolated for each pixel covering the triangle.
  - The vertex attributes are interpolated in screen space in such a way that the attributes are interpolated linearly across the triangle in 3D space.
  - Perspective correct interpolation.


$\mathbf{v}_2$
$\mathbf{P}(s,t)$
$\mathbf{v}_1$
$t(\mathbf{v}_2 - \mathbf{v}_0)$
$s(\mathbf{v}_1 - \mathbf{v}_0)$
$\mathbf{v}_0$

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



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.

The Output Merge Stage

- After pixel fragments have been generated by the pixel shader, they move onto the output merger (OM) stage of the rendering pipeline. 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 completely.
- Some special effects like transparency are implemented with blending.

Polymorphism (1)

- Polymorphism
  - Polymorphism is the ability of an object to take on many forms.
  - Compile-time polymorphism
    - Function overloading
    - Operator overloading
    - Templates
  - Run-time polymorphism
    - Virtual function
      - Runtime polymorphism, as the name defines, is the ability to use the derived classes through the base class pointers and references.

Polymorphism (2)

- The **`virtual`** specifier specifies that a non-static member function is virtual and supports dynamic dispatch.
  - Dynamic dispatch is the process of selecting which implementation of an operation (method or function) to call at run time.
  - Base class
    - **`virtual function-prototype`**;
  - Derived class
    - **`function-prototype override; // override is optional`**
- A **`virtual`** destructor is used to free up the memory space allocated by the derived class object or instance while deleting instances of the derived class using a base class pointer object.
- Pure **`virtual`** function
  - **`virtual member-function-declaration = 0;`**
  - A **`virtual`** function is a member function that is declared within a base class and is re-defined (overridden) by a derived class.
  - A pure **`virtual`** function is a member function of the base class whose only declaration is provided in the base class and should be defined in the derived class; otherwise, the derived class also becomes an abstract class.
  - The base class containing pure **`virtual`** function becomes an **abstract class**.

Smart Pointers (1)

- A smart pointer is a wrapper class template over a pointer with an operator like `*` and `->` overloaded.

- `std::shared_ptr` is a smart pointer that retains shared ownership of an object through a pointer.
  - `Header <memory>`
  - `template<class T> class shared_ptr;`
  - Instance
    - An instance is an object of a class.
    - `std::shared_ptr<type> identifier initializer`$_{opt}$`;`
      - `std::shared_ptr<int> p11(new int(11));`
      - `std::shared_ptr<int> p12 = std::make_shared<int>(12);`
      - `auto p13 = std::make_shared<int>(13);`
      - `std::shared_ptr<int> p14 = p12;`

Smart Pointers (2)

- **`std::unique_ptr`** is a smart pointer that owns (is responsible for) and manages another object via a pointer and subsequently disposes of that object when the **`unique_ptr`** goes out of scope.
- **`std::unique_ptr`** objects own their pointer uniquely.
  - **`Header <memory>`**
  - **`template<class T, class Deleter=std::default_delete<T>> class unique_ptr;`**
  - Instance
    - An instance is an object of a class.
    - **`std::unique_ptr<type> identifier initializer`**$_{opt}$**`;`**
      - **`std::unique_ptr<int> p21(new int(21));`**
      - **`std::unique_ptr<int> p22 = std::make_unique<Widget>(22);`**
      - **`std::unique_ptr<int> p23 = std::move(p22);`**
  - Member functions
    - **`pointer get() const noexcept;`**
      - It returns a pointer to the managed object or **`nullptr`** if no object is owned.

`std::unordered_map`

- `std::unordered_map` is an associative container that contains key-value pairs with unique keys.
  - `std::map` is a **sorted** associative container that contains key-value pairs with unique keys.
  - Members
    - `T& at(const Key& key);`
      - It returns a reference to the mapped value of the element with specified `key`. If no such element exists, an exception of type `std::out_of_range` is thrown.
    - `T& operator[](const Key& key);`
      - It returns a reference to the value that is mapped to a key equivalent to `key`, performing an insertion if such `key` does not already exist.
    - `std::pair<iterator, bool> insert(const value_type& value);`
      - `value_type: std::pair<const Key, T>`
      - `T: mapped_type`