Scene Construction and Projection

Frederick Li

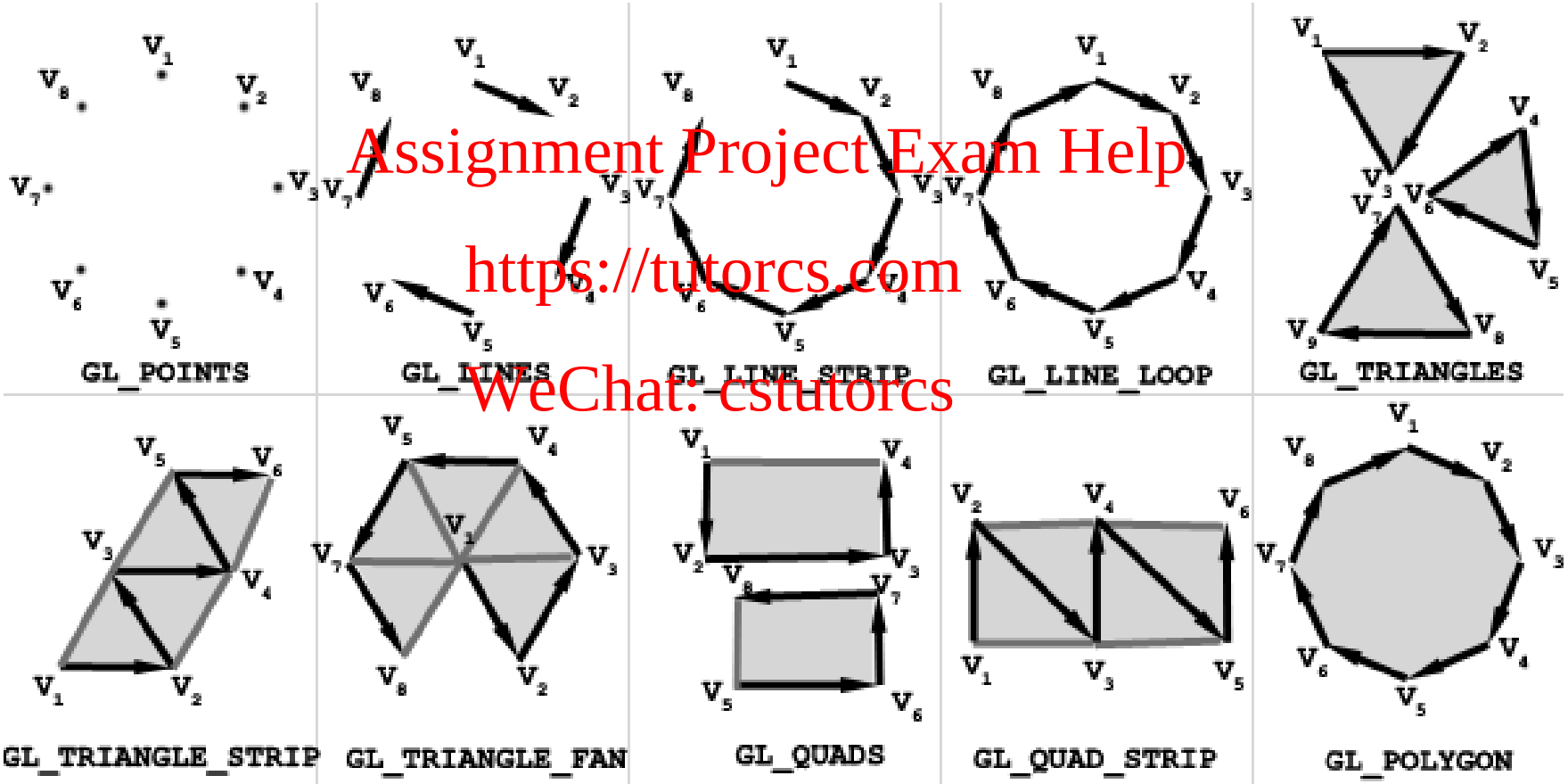
Durham University

This Lesson

- Scene construction and projection
- Concept of applying transformation operations to define how a scene is rendered
- WebGL implementation

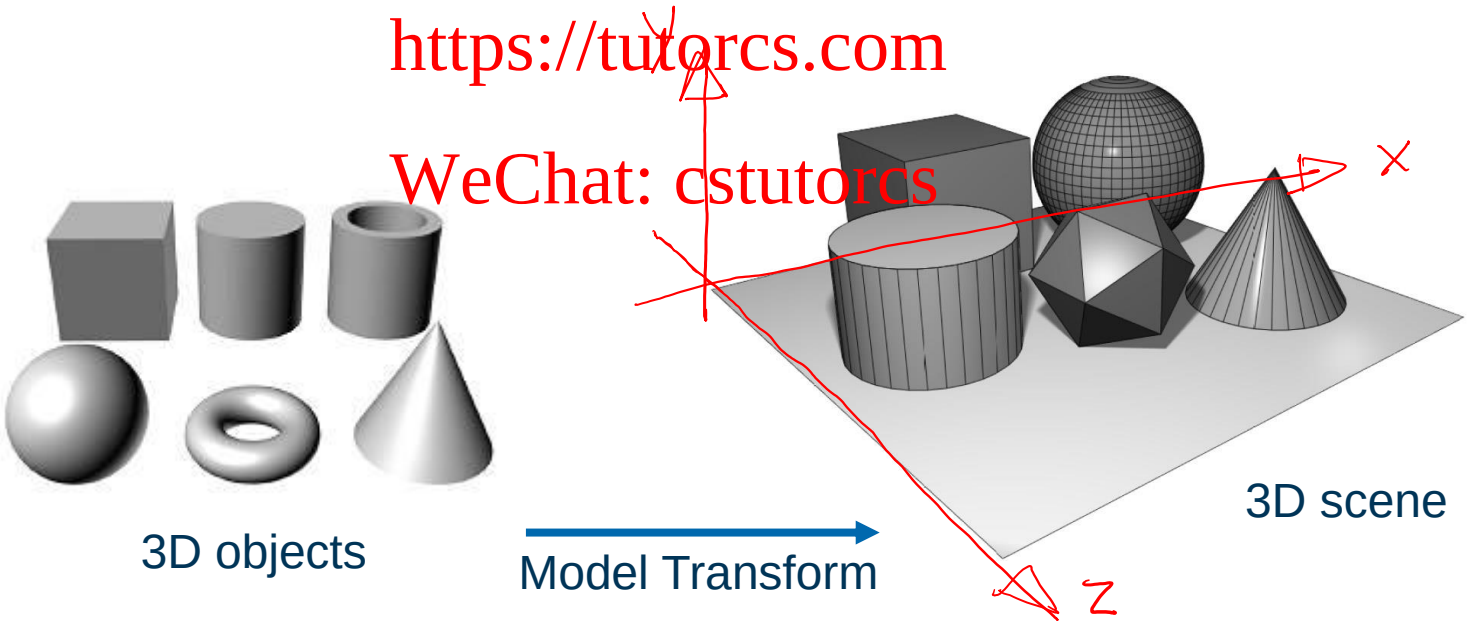
Geometric Primitives in WebGL

- Construct simple shapes from a list of vertices



3D Scene (Virtual Environment)

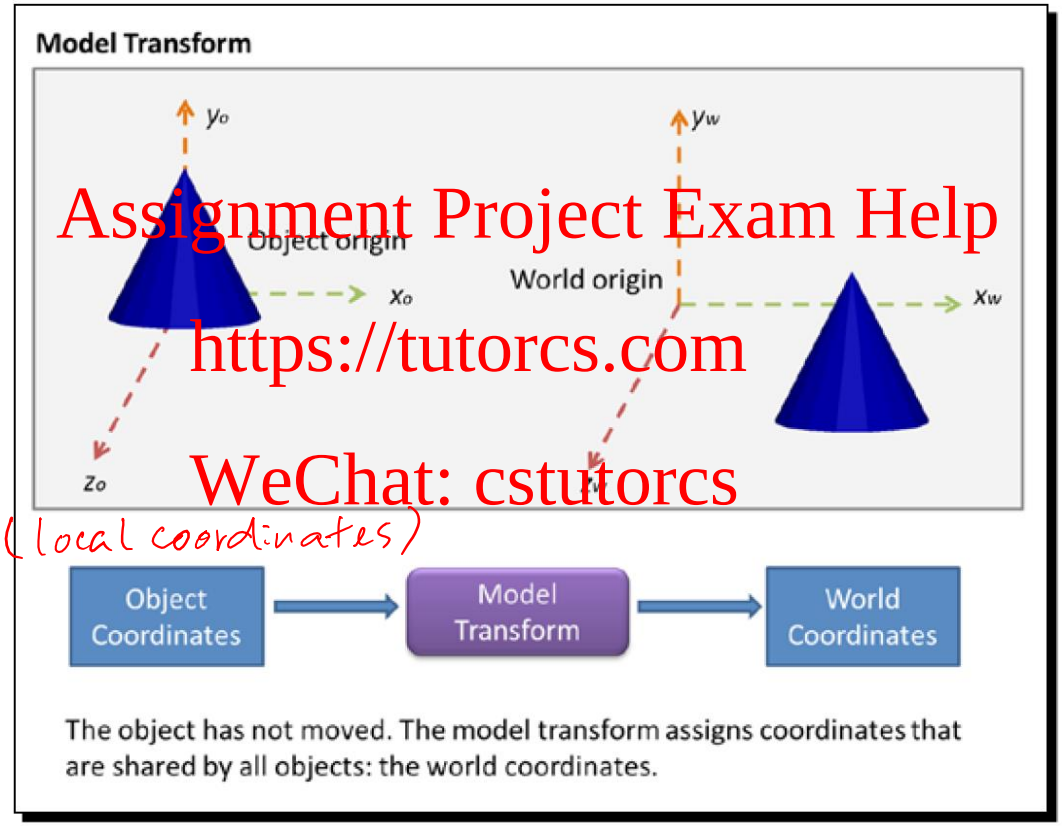
- 3D scene:
  - A space defined by a 3D coordinate system
  - Comprises 3D objects, forming an environment to support user navigation or interaction



World and Local Coordinate Systems

Local coordinates: Each object is constructed on a dedicated coordinate system

Purpose: reduce complication for 3D scene construction



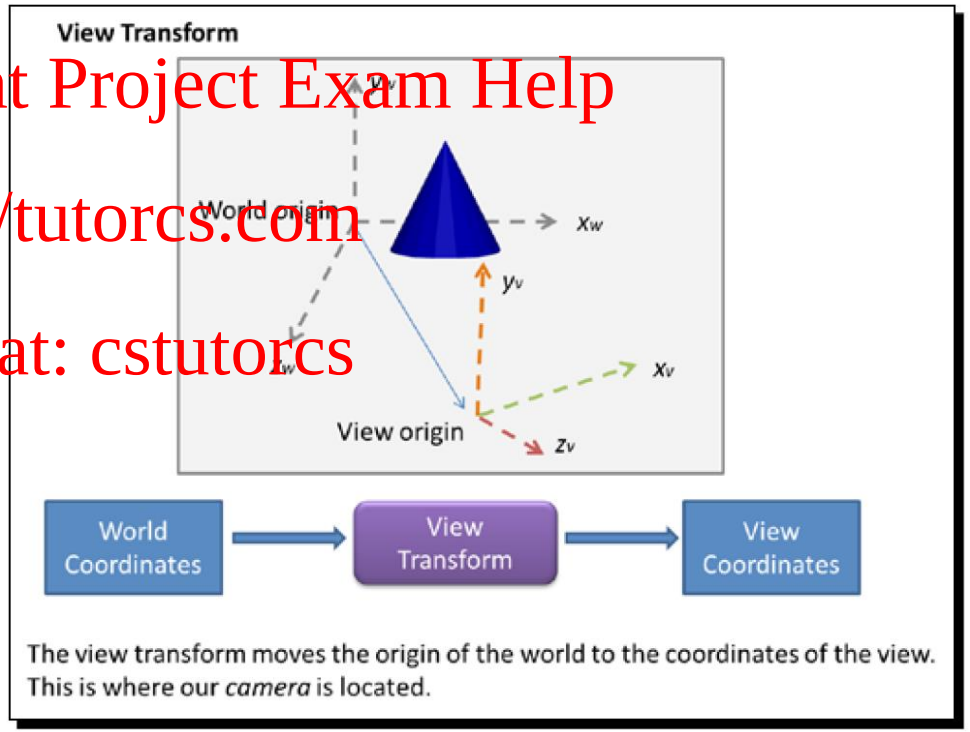
(local coordinates)

World coordinates: apply a single coordinate system to all objects globally.

View Transform

- Shift the origin of the world coordinate system to the view origin. The view origin is where our eye (or virtual camera) is located with respect to the world origin.

- **Purpose:**
  Allow a user (application) to specify how 2D rendered images of a 3D scene will be generated.

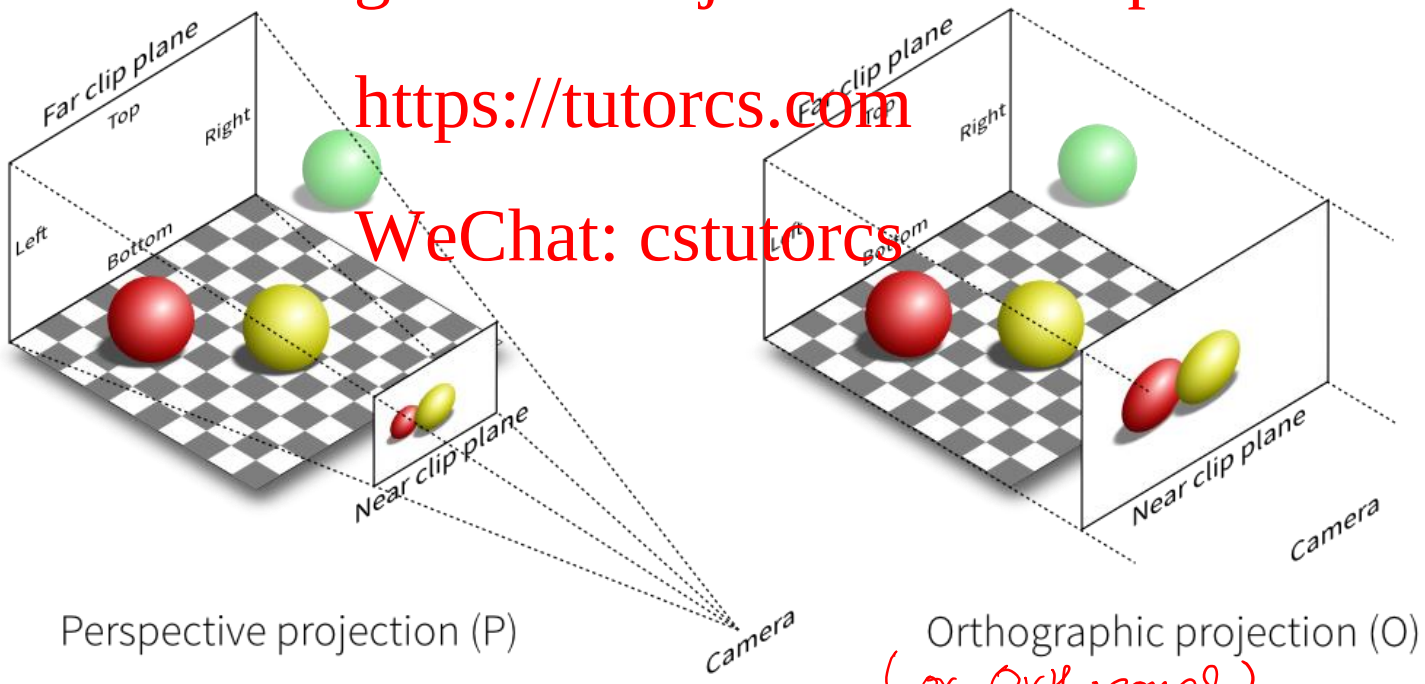


- Many CG applications implement this by **virtual camera**, which defines the current user viewpoint.

Illustration of View Transform

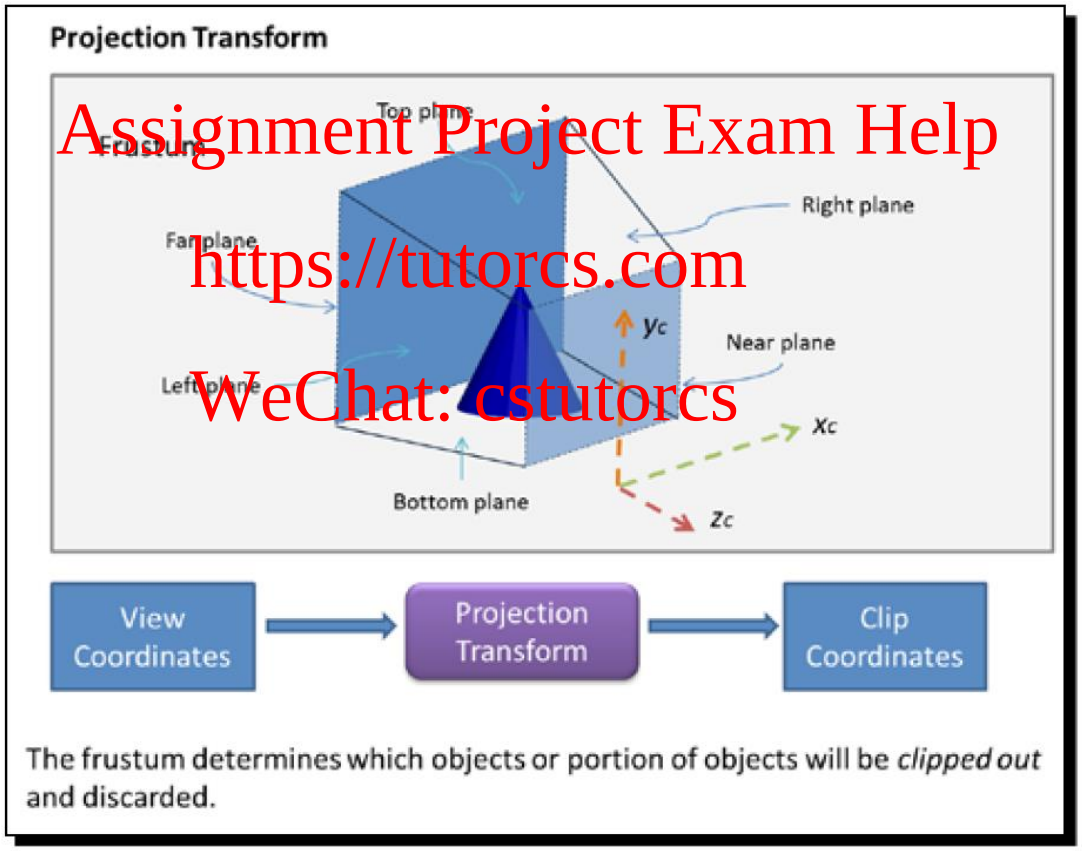
Projection Transform

- Define which part of a 3D scene will currently be visible. Projection transform defines a visible region and how this region is projected onto the screen.
- There are two popular implementations:



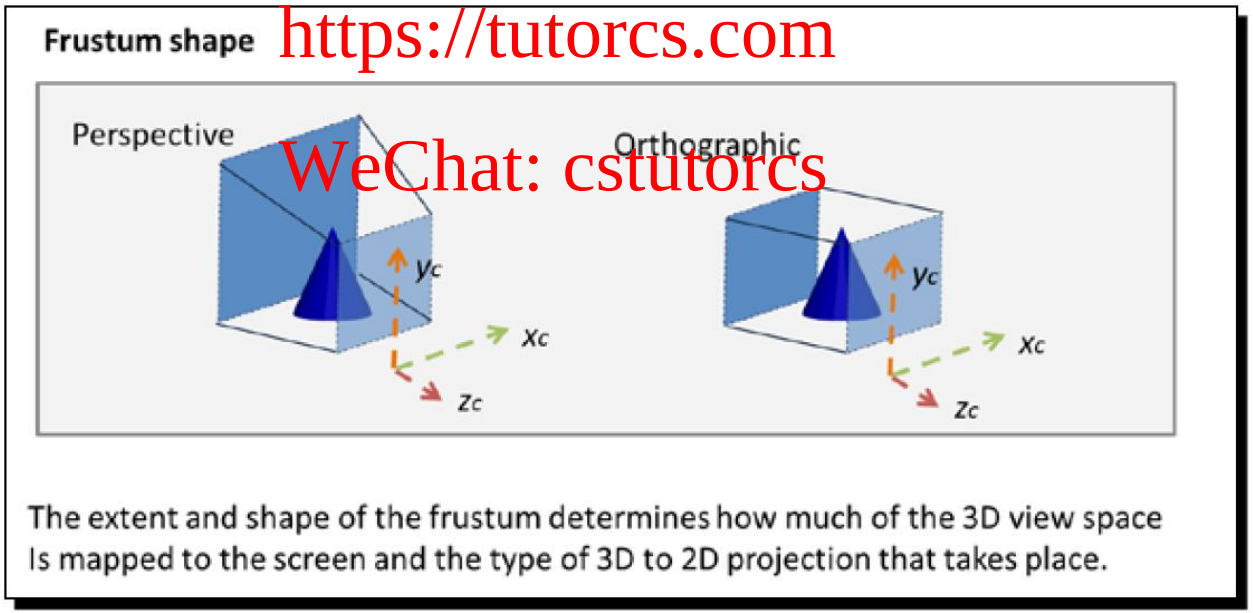
Define a View Frustum

- Projection transform is done based on a view frustum and it is defined by six planes (near, far, top, bottom, right, and left planes)



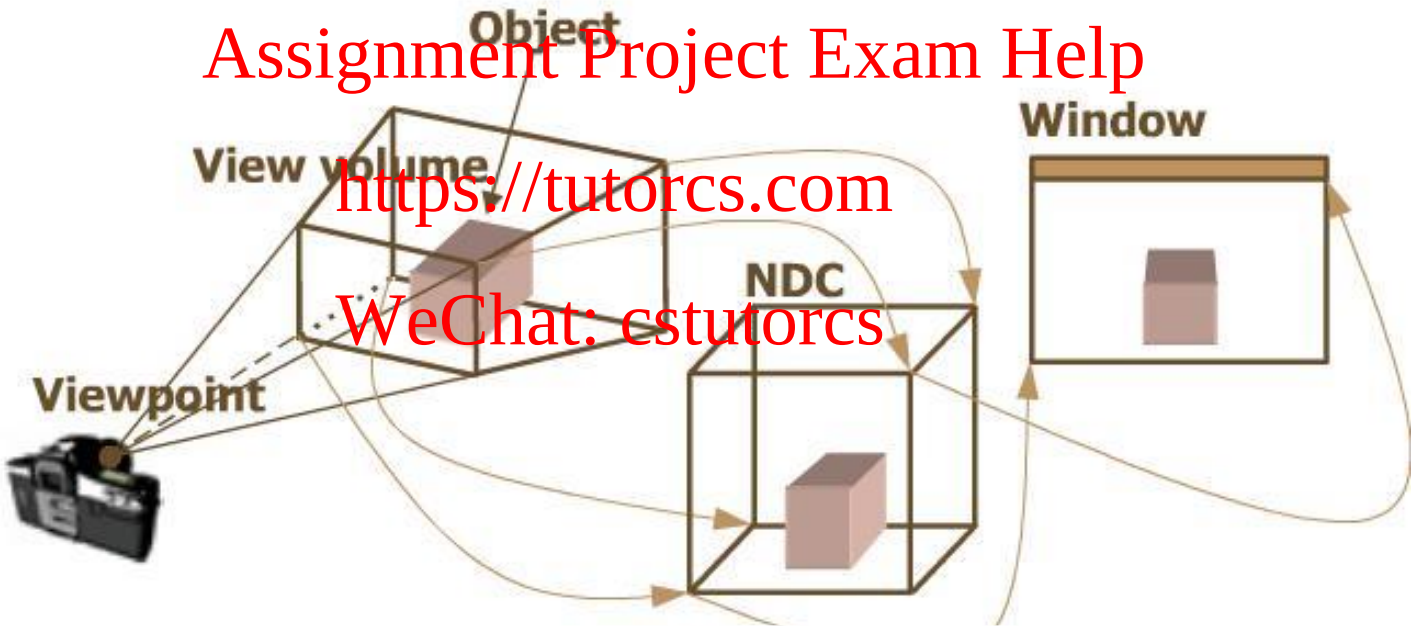
Types of View Frustum

- The shape and extent of the frustum determines the type of view projection from the 3D scene space to the 2D screen
- If the far and near planes have the same dimensions, then the frustum will determine an ***orthographic*** projection. Otherwise, it will be a ***perspective*** projection



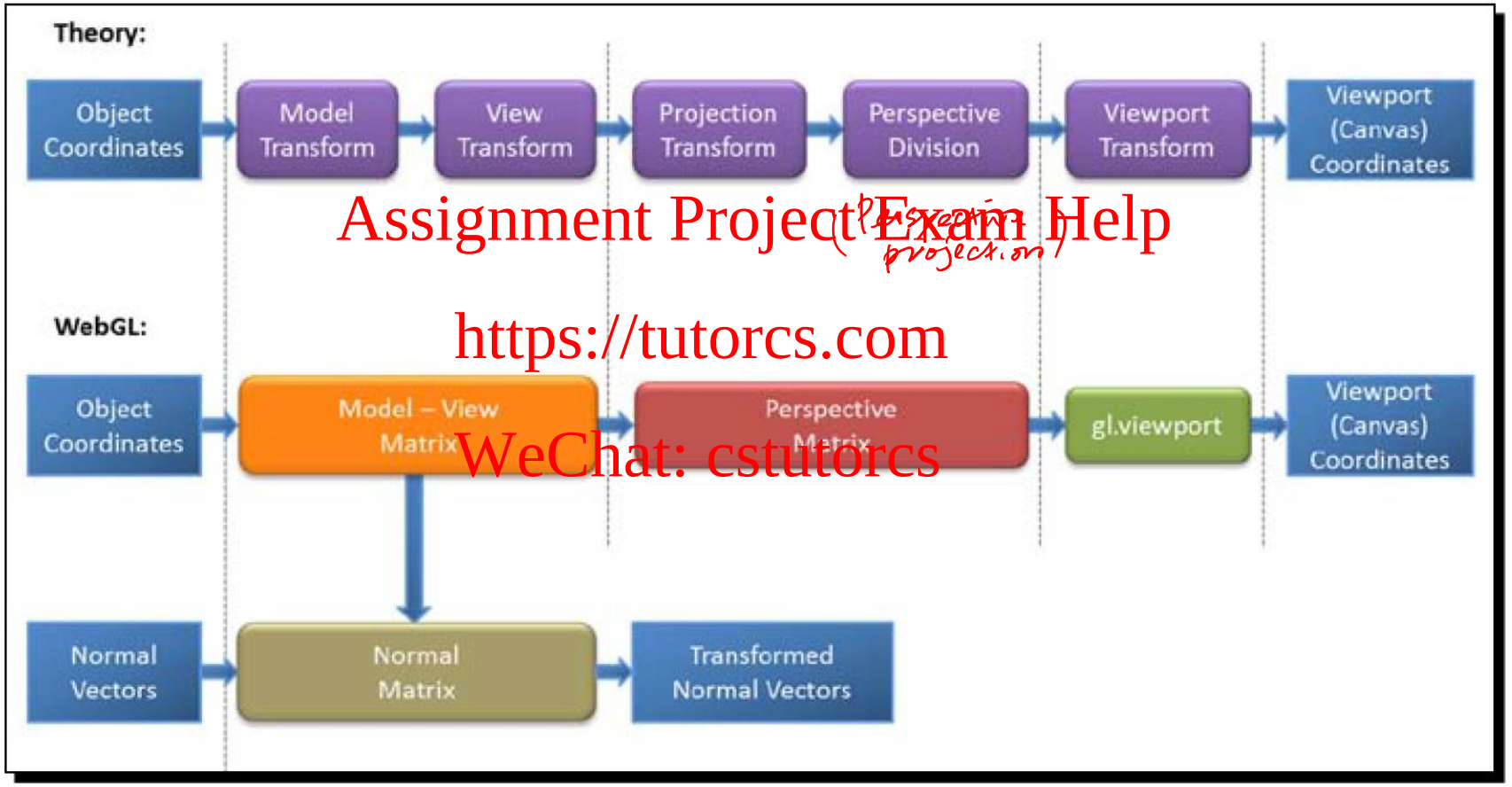
Viewport Transform

- Map projected view to the available space in the computer screen, i.e. viewport, typically referring to the **canvas**



NDC: Normalized Device Coordinates. Its x and y coordinates represent the location of your vertices on a normalized 2D screen space.

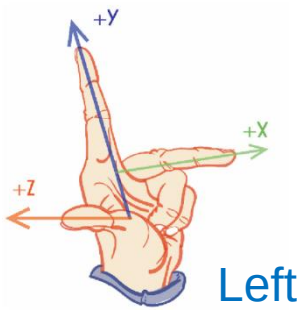
Model-View-Projection Transformation



Example – LookAtTriangles

- Virtual Camera: setLookAt()
  - Parameters: Eye Position, Look-at Position, Camera Orientation

$(x_{eye}, y_{eye}, z_{eye})$ (x_{at}, y_{at}, z_{at}) (dir_x, dir_y, dir_z)



Left hand rule


```
// Set the matrix to be used for to set the camera view
var viewMatrix = new Matrix4();
viewMatrix.setLookAt(0.20, 0.25, 0.25, 0, 0, 0, 0, 1, 0);

// Set the view matrix
gl.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix.elements);

// Draw the rectangle
gl.drawArrays(gl.TRIANGLES, 0, n);
```

(main program)

```
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'attribute vec4 a_Color;\n' +
  'uniform mat4 u_ViewMatrix;\n' +
  'varying vec4 v_Color;\n' +
  'void main() {\n' +
  '  gl_Position = u_ViewMatrix * a_Position;\n' +
  '  v_Color = a_Color;\n' +
  '}\n';
```

Example – LookAtRotatedTriangles

```
// Set the matrix to be used for to set the camera view
var viewMatrix = new Matrix4();
viewMatrix.setLookAt(0.20, 0.25, 0.25, 0, 0, 0, 0, 1, 0);

// Calculate matrix for rotate
var modelMatrix = new Matrix4();
modelMatrix.setRotate(-10, 0, 0, 1); // Rotate around z-axis

// Pass the view projection matrix and model matrix
gl.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix.elements);
gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);

// Draw the rectangle
gl.drawArrays(gl.TRIANGLES, 0, n);
```

Define a rotation matrix to manipulate 3D objects

Modified part of vertex shader

```
'void main() {\n' +
'  gl_Position = u_ViewMatrix * u_ModelMatrix * a_Position;\n' +
'  v_Color = a_Color;\n' +
'}\n';
```

Example – PerspectiveView_mvp

- Support perspective projection

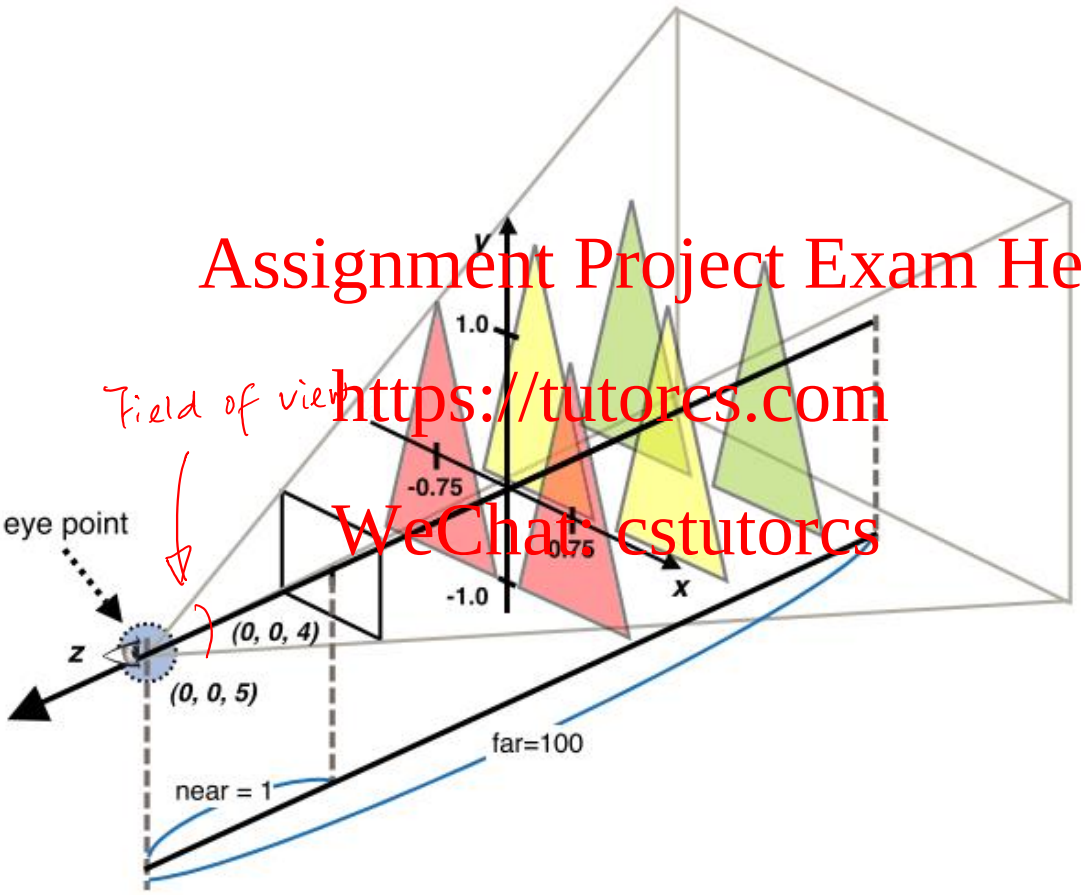
```
// Calculate the view matrix and the projection matrix
modelMatrix.setTranslate(0.75, 0, 0);  // Translate 0.75 units along the positive x-axis
viewMatrix.setLookAt(0, 0, 5, 0, 0, -100, 0, 1, 0);
projMatrix.setPerspective(30, canvas.width/canvas.height, 1, 100);
// Pass the model, view, and projection matrix to the uniform variable respectively
gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);
gl.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix.elements);
gl.uniformMatrix4fv(u_ProjMatrix, false, projMatrix.elements);

gl.drawArrays(gl.TRIANGLES, 0, n); // Draw the triangles
```


Modified part of vertex shader

```
'void main() {\n' +
'  gl_Position = u_ProjMatrix * u_ViewMatrix * u_ModelMatrix * a_Position;\n' +
'  v_Color = a_Color;\n' +
'}\n';
```

Definition of setPerspective()



Parameters:

1. Field of view (in terms of angle)
2. Aspect ratio
3. Near plane
4. Far plane

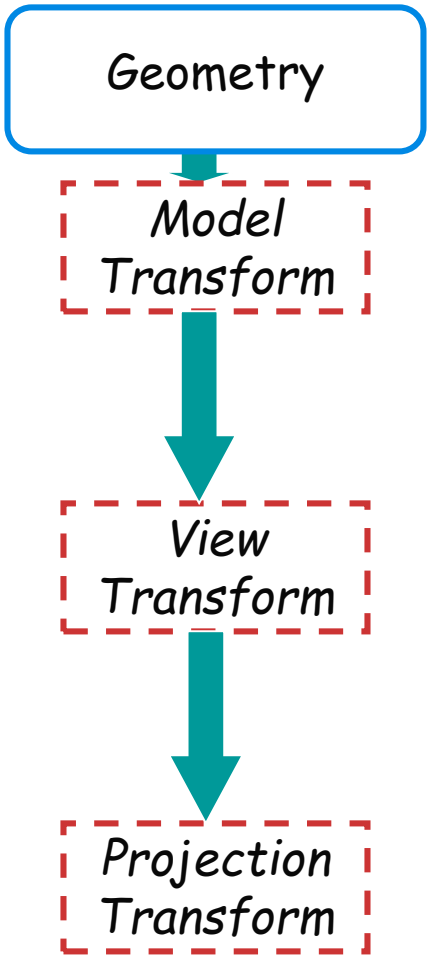
Figure 7.23 The positions of the triangles with respect to the quadrangular pyramid viewing volume

Model, View, Projection Transforms

```
// Vertex shader program
var VSHADER_SOURCE =
  'attribute vec4 a_Position;\n' +
  'attribute vec4 a_Color;\n' +
  'uniform mat4 u_ModelMatrix;\n' +
  'uniform mat4 u_ViewMatrix;\n' +
  'uniform mat4 u_ProjMatrix;\n' +
  'varying vec4 v_Color;\n' +
  'void main() {\n' +
  '  gl_Position = u_ProjMatrix * u_ViewMatrix * u_ModelMatrix * a_Position;\n' +
  '  v_Color = a_Color;\n' +
  '}\n';
```


Implementation of model, view and projection transformations can be done in the vertex shader with a REVISED ORDER of multiplications.

Reference: modelTransform.js



Create Transformation Matrices

This part is done in the main() program.

```
// Get the storage locations of u_ModelMatrix, u_ViewMatrix, and u_ProjMatrix
var u_ModelMatrix = gl.getUniformLocation(gl.program, 'u_ModelMatrix');
var u_ViewMatrix = gl.getUniformLocation(gl.program, 'u_ViewMatrix');
var u_ProjMatrix = gl.getUniformLocation(gl.program, 'u_ProjMatrix');
if (!u_ModelMatrix || !u_ViewMatrix || !u_ProjMatrix) {
  console.log('Failed to Get the storage locations of u_ModelMatrix, u_ViewMatrix, and/or u_ProjMatrix');
  return;
}

var modelMatrix = new Matrix4(); // The model matrix
var viewMatrix = new Matrix4();  // The view matrix
var projMatrix = new Matrix4();  // The projection matrix

// Calculate the view matrix and the projection matrix
viewMatrix.setLookAt(0, 0, 15, 0, 0, -100, 0, 1, 0);
projMatrix.setPerspective(30, canvas.width/canvas.height, 1, 100);
// Pass the model, view, and projection matrix to the uniform variable respectively
gl.uniformMatrix4fv(u_ViewMatrix, false, viewMatrix.elements);
gl.uniformMatrix4fv(u_ProjMatrix, false, projMatrix.elements);
```

Allocate uniform attribute address space for vertex shader

create variables to hold transformation matrices

Define view & projection matrices

Use of Model Transformations

This part is done in the main() program.

```
// Rotate, and then translate
modelMatrix.setTranslate(2, 0, 0);  // Translation
modelMatrix.rotate(30, 0, 0);       // Rotate

// Pass the model matrix to the uniform variable
gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);

// Draw the square
gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_BYTE, 0);
```

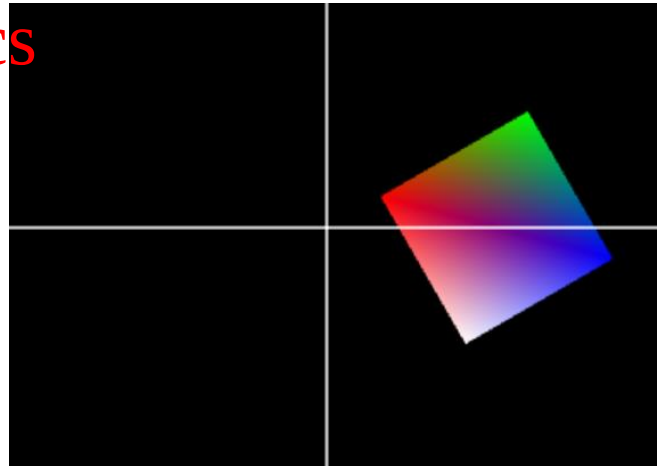
Apply model transform and draw the square

Data structure of the square


```
//  v0------v1
//  |       |
//  |       |
//  |       |
//  v3------v2

var verticesColors = new Float32Array([
  // Vertex coordinates and color (for square)
  -1.0,  1.0,   0.0,  1.0,  0.0,  0.0,  // (x,y,z), (r,g,b)
   1.0,  1.0,   0.0,  0.0,  1.0,  0.0,
   1.0, -1.0,   0.0,  0.0,  0.0,  1.0,
  -1.0, -1.0,   0.0,  1.0,  1.0,  1.0
]);

// Indices of the vertices
var indices = new Uint8Array([
  0, 1, 2,    // 1st triangle
  2, 3, 0     // 2nd triangle
]);
```



Render Different Types of Objects

```
// Set the vertex coordinates and color (for the x, y axes)
var n = initAxesVertexBuffers(gl);
if (n < 0) {
  console.log('Failed to set the vertex information');
  return;
}

// Calculate the view matrix and the projection matrix
modelMatrix.setTranslate(0, 0, 0);  // No Translation
// Pass the model matrix to the uniform variable
gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);

// Draw x and y axes
gl.drawArrays(gl.LINES, 0, n);
```

X & Y axes

```
var verticesColors = new Float32Array([
  // Vertex coordinates and color (for axes)
  -10.0,  0.0,   0.0,  1.0,  1.0,  1.0,  // (x,y,z), (r,g,b)
   10.0,  0.0,   0.0,  1.0,  1.0,  1.0,
   0.0,  10.0,   0.0,  1.0,  1.0,  1.0,
   0.0, -10.0,   0.0,  1.0,  1.0,  1.0
]);
var n = 4;
```

```
// Set the vertex coordinates and color (for the colorful square)
var n = initVertexBuffers(gl);
if (n < 0) {
  console.log('Failed to set the vertex information');
  return;
}

// Rotate, and then translate
modelMatrix.setTranslate(2, 0, 0);  // Translation
modelMatrix.rotate(30, 0, 0);       // Rotate

// Pass the model matrix to the uniform variable
gl.uniformMatrix4fv(u_ModelMatrix, false, modelMatrix.elements);

// Draw the square
gl.drawElements(gl.TRIANGLES, n, gl.UNSIGNED_BYTE, 0);
```

Colorful square

```
//  v0------v1
//  |        |
//  |        |
//  |        |
//  v3------v2

var verticesColors = new Float32Array([
  // Vertex coordinates and color (for square)
  -1.0,  1.0,   0.0,  1.0,  0.0,  0.0,  // (x,y,z), (r,g,b)
   1.0,  1.0,   0.0,  0.0,  1.0,  0.0,
   1.0, -1.0,   0.0,  0.0,  0.0,  1.0,
  -1.0, -1.0,   0.0,  1.0,  1.0,  1.0
]);

// Indices of the vertices
var indices = new Uint8Array([
  0, 1, 2,    // 1st triangle
  2, 3, 0     // 2nd triangle
]);
```

```
gl.vertexAttribPointer(a_Position, 3, gl.FLOAT, false, FSIZE * 6, 0);
gl.enableVertexAttribArray(a_Position);  // Enable the assignment of the buffer object

// Get the storage location of a_Position, assign buffer and enable
var a_Color = gl.getAttribLocation(gl.program, 'a_Color');
if(a_Color < 0) {
  console.log('Failed to get the storage location of a_Color');
  return -1;
}
gl.vertexAttribPointer(a_Color, 3, gl.FLOAT, false, FSIZE * 6, FSIZE * 3);
gl.enableVertexAttribArray(a_Color);  // Enable the assignment of the buffer object

// Unbind the buffer object
gl.bindBuffer(gl.ARRAY_BUFFER, null);

return n;
```

Buffer objects & Shader Attrib.

Reference

- WebGL Programming Guide [Ch. 7]