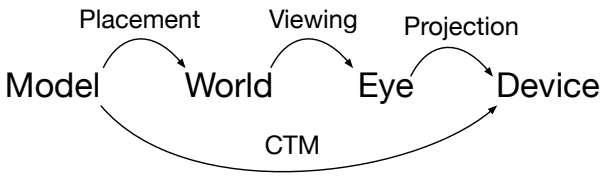
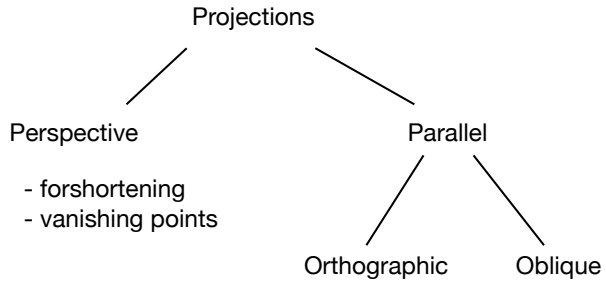
Changing Coord Sys



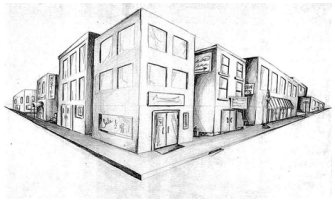
Viewing and Projections

- viewing: volume -> screen
- in 2-D it was easy: window: (xw_{min}, yw_{min}) (xw_{max}, yw_{max})
- in 3-D more complicated
- projection: R^n -> R^m, m < n
  - in graphics 3-D-> 2-D, we project onto a plane

Projections

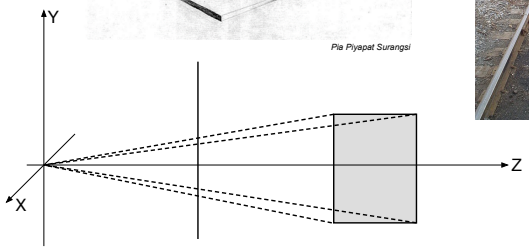


Perspective Projection

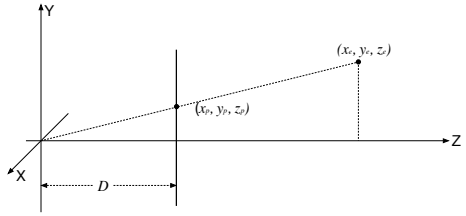


Pia Piyapat Surangsi

Wikimedia



Perspective Projection



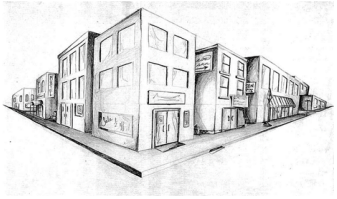
$$\frac{y_p}{D} = \frac{y_e}{z_e} \longrightarrow \quad x_p = \frac{D \cdot x_e}{z_e} \qquad y_p = \frac{D \cdot y_e}{z_e}$$

What if z_e is < 0 or = 0?

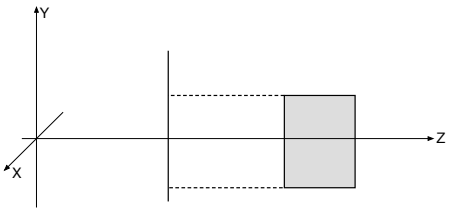
Perspective Proj Matrix

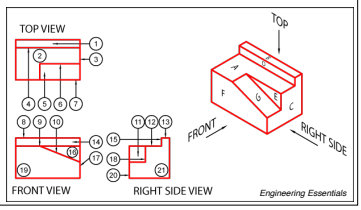
$$\begin{bmatrix} 1 & & & \\ & 1 & & \\ & & 1 & \\ & & 1/D & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z/D \end{bmatrix} \rightarrow \begin{bmatrix} D \cdot x/z \\ D \cdot y/z \\ D \\ 1 \end{bmatrix}$$

$$x_p = \frac{D \cdot x_e}{z_e} \qquad y_p = \frac{D \cdot y_e}{z_e}$$

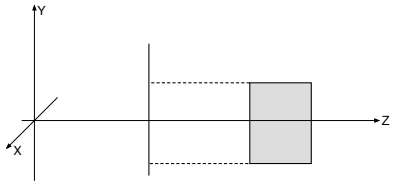


Orthographic Projection



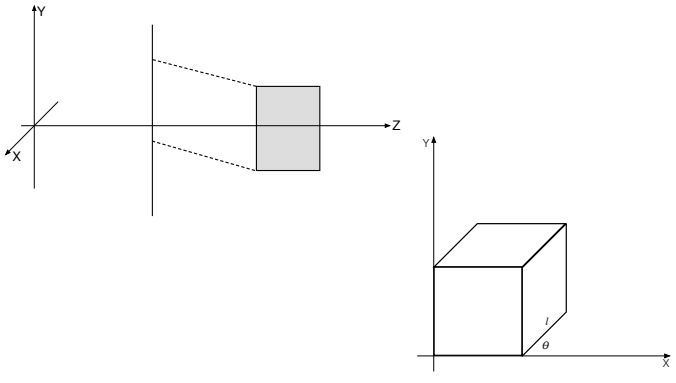


Orthographic Projection

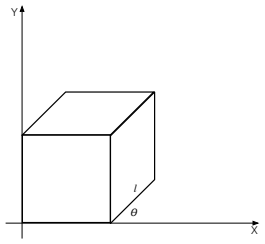


$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & D \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Oblique Projection

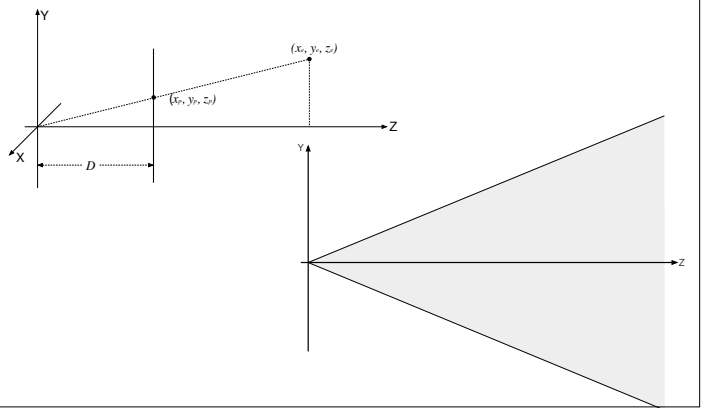


Oblique Projection

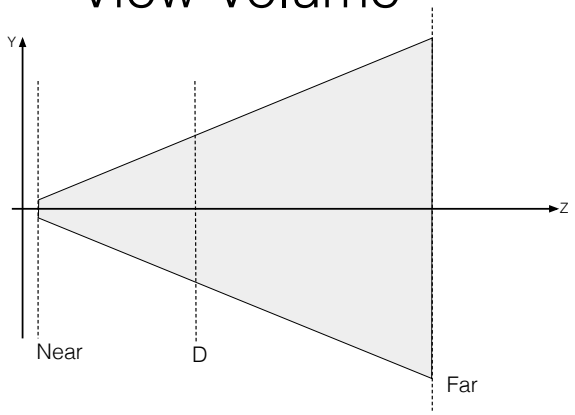


$$\begin{bmatrix} 1 & 0 & l\cos\theta & 0 \\ 0 & 1 & l\sin\theta & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective Projection vs Transformation

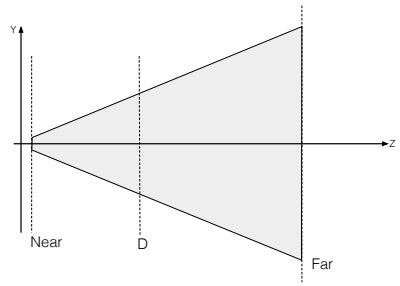


View Volume



Specifying View Volume

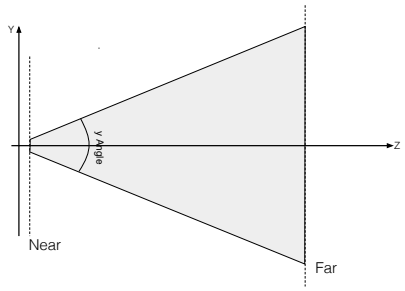
projection screen: D, width, height
distance limits: near, far



Frustum:
truncated pyramid

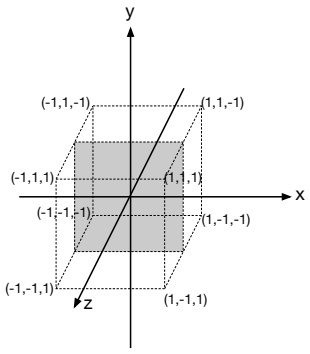
Specifying View Volume

yAngle, aspectRatio
distance limits: near, far



WebGL Default: Orthographic View Volume

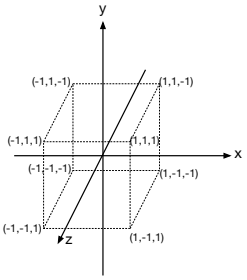
- eye at origin, looking along -z
- everything inside is visible
- everything outside is clipped away



Canonical View Volume or Clip View Volume

Orthographic Projection Transformation

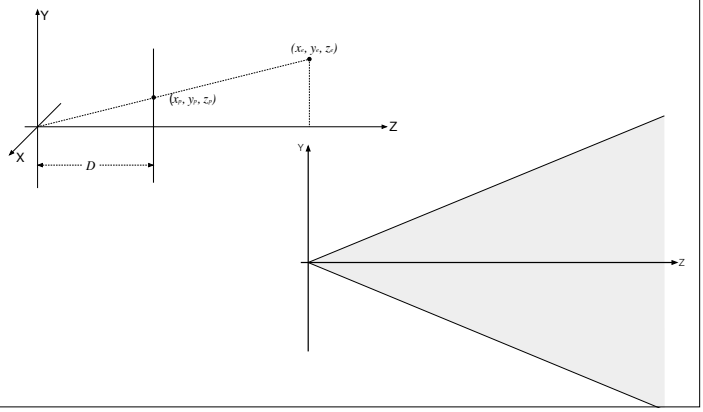
given: *left*, *right*, *bottom*, *top*, *near*, *far*



$$w = right - left$$
$$h = top - bottom$$
$$d = far - near$$

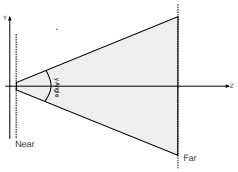
$$\begin{bmatrix} 2/w & 0 & 0 & -(left+right)/w \\ 0 & 2/h & 0 & -(top+bottom)/h \\ 0 & 0 & 2/d & -(near+far)/d \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective Projection vs Transformation

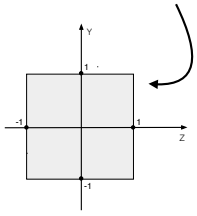


Perspective ~~Projection~~ Transformation

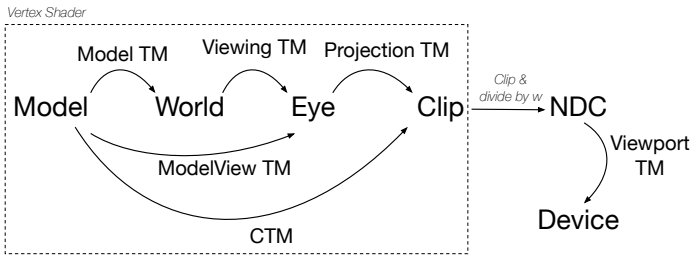
Given: *yAngle*, *aspectRatio*, *near*, *far*



$$\begin{bmatrix} \frac{1}{aspectRatio \cdot \tan(\frac{yAngle}{2})} & 0 & 0 & 0 \\ 0 & \frac{1}{\tan(\frac{yAngle}{2})} & 0 & 0 \\ 0 & 0 & \frac{near+far}{far-near} & \frac{2 \cdot far \cdot near}{far-near} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$



Changing Coord Sys

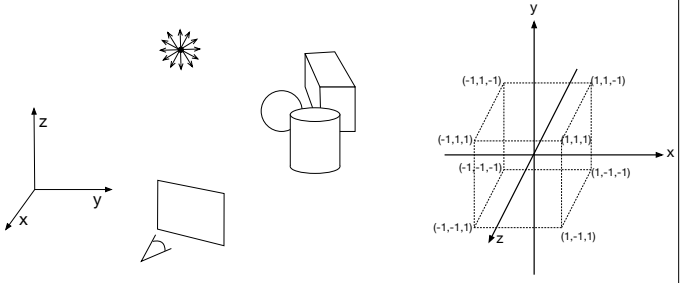


Coordinate Systems

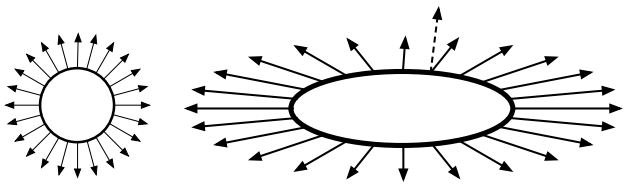
- Model: define object
- World: place objects, define eye/camera, define light positions, perform lighting operations
- Eye: define view volume, perform lighting operations
- Canonical View Volume: clip
- Screen: display

Limitations

- lighting/shading can only be done in World or Eye CS
- Why?

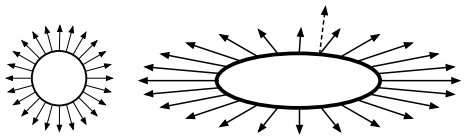


Transforming Points vs Normals



Transforming Normals

- points: TM
- normals: $(TM^{-1})^T$



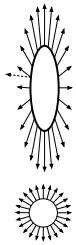
Transforming Normals

eg: TM= Scale(1,1,3)·Rotate$_z$(30°)

$$TM = \begin{bmatrix} .866 & -.5 & 0 & 0 \\ .5 & .866 & 0 & 0 \\ 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$TM^{-1} = \begin{bmatrix} .866 & .5 & 0 & 0 \\ -.5 & .866 & 0 & 0 \\ 0 & 0 & .33 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\left(TM^{-1}\right)^T = \begin{bmatrix} .866 & -.5 & 0 & 0 \\ .5 & .866 & 0 & 0 \\ 0 & 0 & .33 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$



WebGL

- HTML5 + WebGL
- Javascript + GLSL ES 2.0 (OpenGL Shading Language)
- Vertex and Fragment Shaders

Dividing Work

- HTML5:
  - performs interactive tasks
  - event queue
  - callback functions
- WebGL:
  - 3D graphics
  - graphics pipeline

Dividing Work

- html file:
  - UI/interaction, canvas, shader descriptions
- javascript program front end:
  - model, transformation setup, main app logic
- vertex shader:
  - transformations
  - shading/lighting
- fragment shader:
  - shading/lighting, texture, draw pixels

Vertex Shader

- exposes variables and buffer objects to front end
- vertex shader program is called for every vertex, one vertex at a time
- typically passed an array but it only sees one vertex at a time
- sends result (`gl_Position`) to clipper/rasterizer (hidden to you) and then to fragment shader

Vertex Shader Variables

Storage Qualifiers

- **const**: constant
- **attribute**: per-vertex data
- **uniform**: does not change across the primitive
- **varying**: linkage between vertex and fragment shaders

Naming Convention

- a_variableName // attribute variable
- u_variableName // uniform variable

Simple Shaders (Koch)

```
<script id="vertex-shader" type="x-shader/x-vertex">

attribute vec4 vPosition;

void
main()
{
    gl_Position = vPosition;
}
</script>

<script id="fragment-shader" type="x-shader/x-fragment">
precision mediump float;

void
main()
{
    gl_FragColor = vec4( 1.0, 0.0, 0.0, 1.0 );
}
```

Communicating with Shaders

- shared variables, buffer objects

```
<script id="vertex-shader" type="x-shader/x-vertex">

attribute vec4 vPosition;

void
main()
{
    gl_Position = vPosition;
}
</script>
```

```
var vPosition = gl.getAttribLocation( program, "vPosition" );
```

Set up Array Koch

```
var bufferId = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, bufferId);
gl.bufferData(gl.ARRAY_BUFFER, bufferSizeInBytes, gl.STATIC_DRAW);
…
…
var vPosition = gl.getAttribLocation(program, "vPosition");
gl.vertexAttribPointer(vPosition, 2, gl.FLOAT, false, 0, 0);
gl.enableVertexAttribArray(vPosition);
…
…
gl.bufferSubData(gl.ARRAY_BUFFER, 0, flatten(points));
gl.clear(gl.COLOR_BUFFER_BIT);
gl.drawArrays( gl.LINE_STRIP, 0, points.length);
```

Note: `gl.ARRAY_BUFFER` *is tied to a special register in Fragment Shader*

Vertex Shader Example 2

```
<!DOCTYPE html>
<html>

<script id="vertex-shader" type="x-shader/x-vertex">

attribute vec4 vPosition;
attribute vec3 vNormal;

uniform mat4 modelViewMatrix;
uniform mat4 normalMatrix;
uniform mat4 projectionMatrix;

uniform vec4 lightPosition;
uniform vec4 lightColor;
uniform vec4 materialColor;
uniform float Ka, Kd, Ks, shininess;

varying vec4 fColor;
```

Vertex Shader Example 2

```
void main()
{
    gl_Position = projectionMatrix * modelViewMatrix * vPosition;

   // Transform vertex normal into eye coordinates
    vec3 pos = (modelViewMatrix * vPosition).xyz;
    vec3 N = normalize( (normalMatrix*vec4(vNormal,0.0)).xyz);

    // get light direction and eye direction
    vec3 L = vec3(normalize(lightPosition.xyz-pos)) ;
    vec3 E = normalize(-pos.xyz) ;

    // compute reflected light direction
    vec3 R = reflect(-L, N) ;

    // Compute terms in the illumination equation
    vec4 ambientComponent = Ka*materialColor;
    vec4 diffuseComponent = Kd*max( dot(L, N), 0.0 )*materialColor;
    vec4 specularComponent = vec4(0.0, 0.0, 0.0, 1.0);
    if( dot(L, N) > 0.0 ) {
    float Ks = pow( max(dot(R, E), 0.0), shininess );
        specularComponent = Ks * vec4(1.0, 1.0, 1.0, 1.0);
    }
    fColor = (ambientComponent + diffuseComponent + specularComponent)*lightColor;
    fColor.a = 1.0;
}
</script>
```

Fragment Shader Example 2

```
<script id="fragment-shader" type="x-shader/x-fragment">

precision mediump float;

varying vec4 fColor;

void
main()
{

    gl_FragColor = fColor;

}
</script>
```