Computer Viewing

CS 432 Interactive Computer Graphics
Prof. David E. Breen
Department of Computer Science
Positioning the Camera
Objectives

• Introduce the mathematics of projection
• Describe WebGL projection functions in MV.js
• Look at alternate viewing APIs
From the Beginning

• In the beginning:
  - fixed function pipeline
  - Model-View and Projection Transformation
  - Predefined frames: model, object, camera, clip, ndc, window

• After deprecation
  - pipeline with programmable shaders
  - no transformations
  - clip, ndc window frames

• MV.js reintroduces original capabilities
Computer Viewing

• There are three aspects of the viewing process, all of which should be / are implemented in the pipeline,
  - Positioning the camera
    • Setting the model-view matrix
  - Selecting a lens
    • Setting the projection matrix
  - Clipping
    • Setting the view volume
The WebGL Camera

• In OpenGL, initially the object and camera frames are the same
  - Default model-view matrix is an identity
• The camera is located at origin and points in the negative z direction
• OpenGL also specifies a default view volume that is a cube with sides of length 2 centered at the origin
  - Default projection matrix is an identity (almost)
Hard-wired projection is orthographic
Moving the Camera Frame

- If we want to visualize objects with both positive and negative z values we can either
  - Move the camera in the positive z direction
    - Translate the camera frame
  - Move the objects in the negative z direction
    - Translate the world frame
- Both of these views are equivalent and are determined by the model-view matrix
  - Want a translation \( \text{Translate}(0.0,0.0,-d); \)
  - \( d > 0 \)
Moving Camera back from Origin

frames after translation by \(-d\)
\[ d > 0 \]

default frames

(a) 

(b)
Moving Camera back from Origin

frames after translation by \(-d\)

default frames

World is actually being moved relative to camera frame.
Moving the Camera

• We can move the camera to any desired position by a sequence of rotations and translations

• Example: side view
  - Move camera away from origin
  - Rotate it 90 degrees
• Remember that last transformation specified is first to be applied

// Using MV.js
var t = translate(0.0, 0.0, -d);
var ry = rotateY(-90.0);
var m = mult(t, ry);
// or
var m = mult(translate (0.0, 0.0, -d),
           rotateY(-90.0));

This is code for moving world frame relative to camera frame. It is the inverse of the desired camera movement.
LookAt Function

LookAt(eye, at, up)
- eye – location of camera
- at – look-at point
- up – up vector
The LookAt Function

- The GLU library contained the function `gluLookAt` to form the required modelview matrix through a simple interface
- Note the need for setting an up direction
  - Should not be parallel to look-at direction
- Replaced by `lookAt()` in MV.js
  - Can concatenate with modeling transformations
- Example: isometric view of cube aligned with axes

```javascript
var eye = vec3(1.0, 1.0, 1.0);
var at = vec3(0.0, 0.0, 0.0);
var up = vec3(0.0, 1.0, 0.0);
var mv = lookAt(eye, at, up);
```
Other Viewing APIs

• The LookAt function is only one possible API for positioning the camera

• Others include
  
  - View reference point, view plane normal, view up (PHIGS, GKS-3D)
  - Yaw, pitch, roll
  - Elevation, azimuth, twist
  - Direction angles
Projection
Projections and Normalization

• The default projection in the eye (camera) frame is orthographic

• For points within the default view volume

\[ x_p = x \]
\[ y_p = y \]
\[ z_p = 0 \]

• Most graphics systems use view normalization
  - All other views are converted to the default view by transformations that determine the projection matrix
  - Allows use of the same pipeline for all views
Homogeneous Coordinate Representation

default orthographic projection

\[ \begin{align*}
x_p &= x \\
y_p &= y \\
z_p &= 0 \\
w_p &= 1
\end{align*} \]

\[ p_p = Mp \]

\[ M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix} \]

In practice, we can let \( M = I \) and set the \( z \) term to zero later.
Simple Perspective

- Center of projection at the origin
- Projection plane $z = d, d < 0$
Perspective Equations

Consider top and side views

\[
\begin{align*}
\frac{x_p}{z / d} &= x \\
\frac{y_p}{z / d} &= y \\
z_p &= d
\end{align*}
\]
Homogeneous Coordinate Form

consider \( q = Mp \) where \( M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1/d & 0 \end{bmatrix} \)

\[
p = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \quad \Rightarrow \quad q = \begin{bmatrix} x \\ y \\ z \\ z/d \end{bmatrix} = \begin{bmatrix} x/(z/d) \\ y/(z/d) \\ d \\ 1 \end{bmatrix}
\]
Perspective Division

• However \( w \neq 1 \), so we must divide by \( w \) to return from homogeneous coordinates

• This *perspective division* yields

\[
\begin{align*}
    x_p &= \frac{x}{z/d} \\
    y_p &= \frac{y}{z/d} \\
    z_p &= d
\end{align*}
\]

the desired perspective equations

• We will consider the corresponding clipping volume with MV.js functions that are equivalent to deprecated OpenGL functions
View Volumes

Are attached to the cameras!

i.e., view volume parameters are in camera coordinates
WebGL Orthogonal Viewing

\[ \text{ortho} \left( \text{left, right, bottom, top, near, far} \right) \]

near and far measured from camera
View volume specified in camera coordinates
WebGL Perspective

frustum(left, right, bottom, top, near, far)

Specified in \textit{camera coordinates}
Using Field of View

• With frustum it is often difficult to get the desired view

• perspective (fovy, aspect, near, far) often provides a better interface

![Diagram showing front plane with equation aspect = w/h](image)
Computing Matrices

• Compute in JS file, send to vertex shader with `gl.uniformMatrix4fv`
• Dynamic: update in `render()` or shader
var render = function()
{
    gl.clear( gl.COLOR_BUFFER_BIT | gl.DEPTH_BUFFER_BIT);
    eye = vec3(radius*Math.sin(theta)*Math.cos(phi),
               radius*Math.sin(theta)*Math.sin(phi), radius*Math.cos(theta));
    modelViewMatrix = lookAt(eye, at , up);
    projectionMatrix = perspective(fovy, aspect, near, far);
    gl.uniformMatrix4fv( modelViewMatrixLoc, false,
                         flatten(modelViewMatrix) );
    gl.uniformMatrix4fv( projectionMatrixLoc, false,
                         flatten(projectionMatrix) );
    gl.drawArrays( gl.TRIANGLES, 0, NumVertices );
}
in vec4 aPosition;
in vec4 aColor;
out vec4 vColor;
uniform mat4 modelViewMatrix;
uniform mat4 projectionMatrix;

void main() {
    gl_Position = projectionMatrix*modelViewMatrix*aPosition;
    vColor = aColor;
}
Orthogonal Projection Matrices
Objectives

• Derive the projection matrices used for standard orthogonal projections
• Introduce oblique projections
• Introduce projection normalization
Normalization

• Rather than derive a different projection matrix for each type of projection, we can convert all projections to orthogonal projections with the default view volume.
• This strategy allows us to use standard transformations in the pipeline and makes for efficient clipping.
Pipeline View

modelview transformation → projection transformation → perspective division

nonsingular

clipping → projection

against default cube

4D → 3D

vertex shader output

3D → 2D
Notes

• We stay in four-dimensional homogeneous coordinates through both the modelview and projection transformations
  - Both these transformations are nonsingular
  - Default to identity matrices (orthogonal view)

• Normalization lets us clip against simple cube regardless of type of projection

• Delay final projection until end
  - Important for hidden-surface removal to retain depth information as long as possible
Orthographic Normalization

Ortho \((left, right, bottom, top, near, far)\)

Normalization ⇒ find transformation to convert specified clipping volume to default

\((left, bottom, -near)\)

\((-1, -1, -1)\)

\((right, top, -far)\)

\((1, 1, 1)\)

left < right \hspace{1em} bottom < top \hspace{1em} 0 < near < far
Orthographic Matrix

- Two steps
  - Move center to origin
    \[ T\left(-\frac{(\text{left}+\text{right})}{2}, -\frac{(\text{bottom}+\text{top})}{2}, (\text{near}+\text{far})/2\right) \]
  - Scale to have sides of length 2
    \[ S\left(\frac{2}{(\text{left}-\text{right})}, \frac{2}{(\text{top}-\text{bottom})}, \frac{2}{(\text{near}-\text{far})}\right) \]

\[
P = ST = \begin{bmatrix}
\frac{2}{\text{right} - \text{left}} & 0 & 0 & \frac{-\text{right} + \text{left}}{2} \\
0 & \frac{2}{\text{top} - \text{bottom}} & 0 & \frac{-\text{top} + \text{bottom}}{2} \\
0 & 0 & \frac{2}{\text{near} - \text{far}} & \frac{-\text{near} + \text{far}}{2} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Final Projection

• Set $z = 0$

• Equivalent to the homogeneous coordinate transformation

$$M_{orth} = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}$$

• Hence, general orthographic projection in 4D is

$$P = M_{orth}ST$$
Oblique Projections

• The OpenGL projection functions cannot produce general parallel projections such as

• However if we look at the example of the cube it appears that the cube has been sheared

• Oblique Projection = Shear + Orthographic Projection
General Shear

Top view

Side view
Shear Matrix

\[ H(\theta, \phi) = \begin{bmatrix} 1 & 0 & -\cot \theta & 0 \\ 0 & 1 & -\cot \phi & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \]

Projection matrix

\[ P = M_{\text{orth}} H(\theta, \phi) \]

General case: \[ P = M_{\text{orth}} STH(\theta, \phi) \]
Equivalency
Effect on Clipping

- The projection matrix $P = STH$ transforms the original clipping volume to the default clipping volume.

[Diagram showing the effect of the projection matrix on the clipping volume, with labels for DOP, top view, far plane, near plane, clipping volume, object, and distorted object (projects correctly).]
Perspective Projection Matrices
Objectives

• Derive the perspective projection matrices used for standard WebGL projections
Simple Perspective

Consider a simple perspective with the COP at the origin, the near clipping plane at \( z = -1 \), and a 90 degree field of view determined by the planes \( x = \pm z, \ y = \pm z \)
Perspective Matrices

Simple projection matrix in homogeneous coordinates

\[ M = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -1 & 0 \end{bmatrix} \]

\[
\begin{align*}
    x' &= x \\
    y' &= y \\
    z' &= z \\
    w' &= -z
\end{align*}
\]

\[
\begin{align*}
    x' &= -x/z \\
    y' &= -y/z \\
    z' &= -1
\end{align*}
\]

Note that this matrix is independent of the far clipping plane
Generalization

\[
N = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & \alpha & \beta \\
0 & 0 & -1 & 0 \\
\end{bmatrix}
\]

\[
x' = x \\
y' = y \\
z' = \alpha z + \beta \\
w' = -z
\]

After perspective division, the point \((x, y, z, 1)\) goes to

\[
x'' = -\frac{x}{z} \\
y'' = -\frac{y}{z} \\
z'' = -(\alpha + \beta/z)
\]

which projects orthogonally to the desired point regardless of \(\alpha\) and \(\beta\)
Defining \( \alpha \) and \( \beta \)

If we define

\[
\alpha = \frac{\text{near} + \text{far}}{\text{near} - \text{far}}
\]

\[
\beta = \frac{2 \times \text{near} \times \text{far}}{\text{near} - \text{far}}
\]

the near plane is mapped to \( z = -1 \)

the far plane is mapped to \( z = 1 \)

and the sides are mapped to \( x = \pm 1, y = \pm 1 \)

Hence the new clipping volume is the default clipping volume

\[
z' = -(\alpha + \beta/z)
\]
**Normalization Transformation**

Original clipping volume

Original object

New clipping volume

Normalized coordinates:
- $z = -x$
- $z = x$
- $z = -\text{near}$
- $z = -\text{far}$
- $x = -1$
- $x = 1$
- $z = -1$
- $z = 1$

Distorted object projects correctly
Normalization and Hidden-Surface Removal

• Although our selection of the form of the perspective matrices may appear somewhat arbitrary, it was chosen so that if $z_1 > z_2$ in the original clipping volume then the for the transformed points $z_1' > z_2'$

• Thus hidden surface removal works if we first apply the normalization transformation

• However, the formula $z' = -(\alpha + \beta/z)$ implies that the distances are distorted by the normalization which can cause numerical problems especially if the near distance is small
WebGL Perspective

- frustum allows for an asymmetric viewing frustum (although perspective does not)

frustum(left, right, bottom, top, near, far)

\[
Z = Z_{\text{min}}
\]

\[
(x_{\text{min}}, y_{\text{min}}, z_{\text{max}}) \longrightarrow \text{COP} \quad (x_{\text{max}}, y_{\text{max}}, z_{\text{max}})
\]

left < right \quad bottom < top \quad near < far

near & far positive
WebGL Perspective

- `perspective` provides less flexible, but more intuitive perspective viewing

```javascript
perspective(fov, aspect, near, far);
```

- Field of view in degrees
- `aspect`: w/h
- `near < far`, both positive
OpenGL Perspective Matrix

- The normalization in frustum requires an initial shear to form a right viewing pyramid, followed by a scaling to get the normalized perspective volume. Finally, the perspective matrix results in needing only a final orthographic transformation

\[
P = NSH
\]

our previously defined perspective matrix  shear and scale
Why do we do it this way?

• Normalization allows for a single pipeline for both perspective and orthogonal viewing
• We stay in four dimensional homogeneous coordinates as long as possible to retain three-dimensional information needed for hidden-surface removal and shading
• Simplifies clipping
Perspective Matrices

**frustum**

\[ P = \begin{bmatrix}
\frac{2 \cdot \text{near}}{\text{right} - \text{left}} & 0 & \frac{\text{right} - \text{left}}{\text{right} - \text{left}} & 0 \\
0 & \frac{2 \cdot \text{near}}{\text{top} - \text{bottom}} & \frac{\text{top} + \text{bottom}}{\text{top} - \text{bottom}} & 0 \\
0 & 0 & \frac{\text{far} + \text{near}}{\text{far} - \text{near}} & \frac{-2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}} \\
0 & 0 & \frac{-1}{\text{far} - \text{near}} & 0
\end{bmatrix} \]

**Perspective**

\[ P = \begin{bmatrix}
\frac{\text{near}}{\text{right}} & 0 & 0 & 0 \\
0 & \frac{\text{near}}{\text{top}} & 0 & 0 \\
0 & 0 & \frac{-\text{far} + \text{near}}{\text{far} - \text{near}} & \frac{-2 \cdot \text{far} \cdot \text{near}}{\text{far} - \text{near}} \\
0 & 0 & \frac{-1}{\text{far} - \text{near}} & 0
\end{bmatrix} \]
Viewing WebGL Code

• It’s not too bad, thanks to Ed Angel

• In application

```javascript
model_view = lookAt(eye, at, up);
projection = ortho(left, right, bottom, top, near, far);
```

or

```javascript
projection = perspective(fov, aspect, near, far);
```

• In vertex shader

```javascript
gl_Position = projection*model_view*vPosition;
```
Go to Assignment 5
Simple Mesh Format (SMF)

- Defined by Michael Garland
  - https://mgarland.org
  - https://people.sc.fsu.edu/~jburkardt/txt/smf_format.txt
- Triangle data
- List of 3D vertices
- List of references to vertex array
  define faces (triangles)
- Vertex indices begin at 1

```#$SMF 1.0
#$vertices 5
#$faces 6
v  2.0 0.0 2.0
v  2.0 0.0 -2.0
v -2.0 0.0 -2.0
v -2.0 0.0  2.0
v  0.0 5.0  0.0
f 1 3 2
f 1 4 3
f 3 5 2
f 2 5 1
f 1 5 4
f 4 5 3```
var smf_file = loadFileAJAX(fname); // in initShaders2.js
var lines = smf_file.split('\n');

for(var line = 0; line < lines.length; line++){
    var strings = lines[line].trimRight().split(' ');
    switch(strings[0]){
        case('v'):
            // Process vertices
            break;
        case('f'):
            // Process faces
            break;
    }
}

Reading an SMF File
Normal for Triangle

Plane \( \mathbf{n} \cdot (\mathbf{p} - \mathbf{p}_0) = 0 \)

\[
\mathbf{n} = (\mathbf{p}_1 - \mathbf{p}_0) \times (\mathbf{p}_2 - \mathbf{p}_0)
\]

Normalize \( \mathbf{n} \leftarrow \mathbf{n}/|\mathbf{n}| \)

Note that right-hand rule determines outward face
Flat Shading

• Define objects with TRIANGLES
• Vertices and colors are duplicated in buffers
• The three vertices of each triangle are assigned the same color
Flat Shading

- GLSL 3.0 supports a “flat” qualifier
- Each vertex has a unique color
- Triangle’s color is set to the last vertex’s color
- In vertex shader
  
  ```glsl
  flat out vcolor;
  ```
- In fragment shader
  
  ```glsl
  flat in vcolor;
  ```
Mesh Shading

• The previous example is not general because we knew the normal at each vertex analytically.

• For polygonal models, Gouraud proposed we use the average of the normals around a mesh vertex:

\[ \mathbf{n} = \frac{\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4}{|\mathbf{n}_1 + \mathbf{n}_2 + \mathbf{n}_3 + \mathbf{n}_4|} \]
Calculating Normals

- Create vector structure (for normals) same size as vertex structure
- Initialize with [0,0,0]
- For each face
  - Calculate unit normal
  - Add to normals structure using vertex indices
- Normalize all the normal vectors
Suggestions for HW5

• Read in smf files & test with HW4
• Transform centroid of bounding box to origin
• Calculate normal for each triangle
• Use normalized normal as color for triangle
  - Be sure to take absolute value
• Each duplicated triangle vertex has its own color
  - This color is passed to the fragment shader
• Next implement LookAt feature
• Test with fixed eye point and orthographic projection
Suggestions for HW5

• Make sure that your model is in the view volume defined by your ortho() function
• Recall that your view volume is defined in camera coordinates
• Send model-view and projection matrix to the vertex shader via uniform variable
• Render with gl.TRIANGLES
• You should now be displaying an orthographic projection from an arbitrary camera location/orientation
Suggestions for HW5

• Implement perspective projection
  - Think about the parameter values you use

• Define menu that allows user to pick projection type

• Make camera move with a keyboard() function

• User should be able to change the angle, height and radius of the camera location (in cylindrical coordinates)