GPU Programming for Video Games

3D to 2D Projection

Prof. Aaron Lanterman
(Based on slides by Prof. Hsien-Hsin Sean Lee)
School of Electrical and Computer Engineering
Georgia Institute of Technology
Projection from 3D space

Much discussion adapted from Joe Farrell’s article: http://www.codeguru.com/cpp/misc/misc/math/article.php/c10123__1/ )
Canonical view volume

- Projection transforms your geometry into a canonical view volume in normalized device coordinates ("clip space")
- Only X- and Y-coordinates will be mapped onto the screen
- Z will be almost useless, but used for depth test

Canonical view volume (LHS)
(-1, -1, 0) to (1,1,1) used by Direct3D
Strange “conventions”

Canonical “clips space” view volume (LHS)
(-1, -1, 0) to (1,1,1) used by D3D/XNA
(remember unnormalized eye-space coordinates in Direct3D are in a LHS, but in XNA are in a RHS!!!)

Canonical “clip space” view volume (LHS)
(-1, -1, 1) to (1,1,1) used by OpenGL/Unity
(remember unnormalized eye-space coordinates in OpenGL are in a RHS, but in Unity are in a LHS!!)
Orthographic (or parallel) projection

- Project from 3D space to the viewer’s 2D space
Style of orthographic projection

- Same size in 2D and 3D
- No sense of distance
- Parallel lines remain parallel
- Good for tile-based games where camera is in fixed location (e.g., Mahjong or 3D Tetris)

Direct3D orthographic projection

View Volume
(an axis-aligned box)

Canonical view volume (Direct3D)

(r, t, f)

(l, b, n)

(1, 1, 1)

(0, -1, -1)
General orthographic math

• Derive $x'$ and $y'$

$$x \in [l, r] \quad x' \in [-1, 1]$$

$$-1 \leq \frac{2(x-l)}{r-l} \leq -1 \leq 1$$

$$l \leq x \leq r$$

$$-1 \leq \frac{2x-2l-r+l}{r-l} \leq 1$$

$$0 \leq x - l \leq r - l$$

$$-1 \leq \frac{2x}{r-l} - \frac{r+l}{r-l} \leq 1$$

$$0 \leq \frac{x-l}{r-l} \leq 1$$

$$0 \leq \frac{2(x-l)}{r-l} \leq 2$$

$$0 \leq \frac{2x}{r-l} - \frac{r+l}{r-l} \leq 2$$

\[\therefore \quad x' = \frac{2x}{r-l} - \frac{r+l}{r-l}\]

D3D orthographic math for Z (LHS default)

• Derive \( z' \)

\[
\begin{align*}
      z \in [n, f] & \quad z' \in [0, 1] \\
      n \leq z \leq f & \\
      0 \leq z - n \leq f - n & \\
      0 \leq \frac{z - n}{f - n} & \leq 1
\end{align*}
\]

\[
\begin{align*}
      0 \leq \frac{z}{f - n} - \frac{n}{f - n} & \leq 1 \\
      \therefore \quad z' = \frac{z}{f - n} - \frac{n}{f - n}
\end{align*}
\]
D3D orthographic results (LHS)

\[ x' = \frac{2x}{r-l} - \frac{r+l}{r-l} \]

\[ y' = \frac{2y}{t-b} - \frac{t+b}{t-b} \]

\[ z' = \frac{z}{f-n} - \frac{n}{f-n} \]
D3D orthographic matrix (LHS default)

• Direct3D primarily uses LHS, z from 0 to 1, row vectors

\[
[x', y', z', 1] = [x, y, z, 1]P \quad \text{where } P = \begin{bmatrix}
\frac{2}{r-l} & 0 & 0 & 0 \\
0 & \frac{2}{t-b} & 0 & 0 \\
0 & 0 & \frac{1}{f-n} & 0 \\
\frac{l+r}{l-r} & \frac{t+b}{b-t} & \frac{n}{n-f} & 1
\end{bmatrix}
\]

• In Direct3D: D3DXMatrixOrthoOffCenterLH(*o,l,r,b,t,n,f)

D3D orthographic math for Z (RHS weird)

- For RHS, in most API calls z clip parameters are positive, and clip space switches to using a LHS
- Derive z'

\[-z \in [n, f] \quad z' \in [0, 1]\]

\[0 \leq \frac{-z}{f-n} - \frac{n}{f-n} \leq 1\]

\[n \leq -z \leq f\]

\[0 \leq -z - n \leq f - n\]

\[0 \leq \frac{-z - n}{f - n} \leq 1\]

D3D orthographic matrix (RHS weird)

In Direct3D: D3DXMatrixOrthoOffCenterRH(*o,l,r,b,t,n,f)

In XNA: Matrix.CreateOrthographicOffCenter(l,r,b,t,n,f)

http://www.cs.utk.edu/~vose/c-stuff/opengl/gIOrtho.html
Simpler D3D ortho matrix (LHS default)

- Most orthographic projection setups
  - Z-axis passes through the center of your view volume
  - Field of view (FOV) extends equally far
    - To the left as to the right (i.e., $r = -l$)
    - To the top as to the below (i.e., $t = -b$)

$$[x', y', z', 1] = [x, y, z, 1]P \text{ where } P = \begin{bmatrix} \frac{2}{w} & 0 & 0 & 0 \\ 0 & \frac{2}{h} & 0 & 0 \\ 0 & 0 & \frac{1}{f-n} & 0 \\ 0 & 0 & \frac{n}{n-f} & 1 \end{bmatrix}$$

- In Direct3D: D3DXMatrixOrthoLH(*o,w,h,n,f)

Simpler D3D ortho matrix (RHS weird)

- For RHS, in most API calls z input parameters are positive, and clip space switches to using a LHS

\[
[x', y', z', 1] = [x, y, z, 1]P \quad \text{where } P =
\begin{bmatrix}
\frac{2}{w} & 0 & 0 & 0 \\
0 & \frac{2}{h} & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & \frac{n}{n-f} & 1
\end{bmatrix}
\]

- In Direct3D: D3DXMatrixOrthoRH(*o,w,h,n,f)
- In XNA: Matrix.CreateOrthographic(w,h,n,f)

OpenGL orthographic projection

View Volume
(an axis-aligned box)

Canonical view volume (OpenGL)

(-1, -1, -1)

(1, 1, 1)

(x, y, z)

(l, b, n)

(r, t, f)
OpenGL orthographic math for Z (RHS)

• Derive $z'$

$$-z \in [n, f] \quad z' \in [-1, 1] \quad -1 \leq \frac{2(-z - n)}{f - n} \leq 1$$

$$n \leq -z \leq f$$

$$0 \leq -z - n \leq f - n$$

$$0 \leq \frac{-z - n}{f - n} \leq 1$$

$$0 \leq \frac{2(-z - n)}{f - n} \leq 2$$

$$-1 \leq \frac{-2z - 2n - f + n}{f - n} \leq 1$$

$$-1 \leq \frac{-2z}{f - n} - \frac{f + n}{f - n} \leq 1$$

$$\therefore \quad z' = \frac{-2z}{f - n} - \frac{f + n}{f - n}$$

OpenGL orthographic results

\[ x' = \frac{2x}{r-l} - \frac{r+l}{r-l} \]

\[ y' = \frac{2y}{t-b} - \frac{t+b}{t-b} \]

\[ z' = \frac{-2z}{f-n} - \frac{f+n}{f-n} \]
Ortho proj matrix (OpenGL/Unity)

- For RHS, in most API calls z input parameters are positive, and clip space switches to using a LHS

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
1'
\end{bmatrix}
= P
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]
where
\[
P =
\begin{bmatrix}
\frac{2}{r-l} & 0 & 0 & -\frac{r+l}{r-l} \\
0 & \frac{2}{t-b} & 0 & -\frac{t+b}{t-b} \\
0 & 0 & \frac{-2}{f-n} & -\frac{f+n}{f-n} \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

- In OpenGL: glOrtho(l,r,b,t,n,f)
- In Unity: Matrix4x4.Ortho(l,r,b,t,n,f)
Perspective projection
Viewing frustum

Think about looking through a window in a dark room
Viewing frustum with furniture

Near plane

Far plane

Viewer’s position
Direct3D perspective projection

Canonical view volume (Direct3D)

View Frustum
(a truncated pyramid)

(l, b, n) (r, t, f)

(l, b, n)

(1, 1, 1)

(-1, -1, 0)
D3D perspective proj mapping

- Given a point \((x,y,z)\) within the view frustum, project it onto the near plane \(z=n\)
- We will map \(x\) from \([l,r]\) to \([-1,1]\) and \(y\) from \([b,t]\) to \([-1,1]\)

D3D perspective math (1)

To calculate new coordinates of $x'$ and $y'$

\[
\frac{x'}{x} = \frac{n}{z} \Rightarrow x' = \frac{nx}{z}
\]

\[
\frac{y'}{x'} = \frac{y}{x} \Rightarrow y' = \frac{yx'}{x} = \frac{y \cdot nx}{x} = \frac{ny}{z}
\]

Next apply our orthographic projection formulas

D3D perspective math (2)

Now let’s tackle the z’ component

D3D perspective math (3)

\[ x'z = \frac{2n}{r-l} x - \frac{r+l}{r-l} z \]
\[ y'z = \frac{2n}{t-b} y - \frac{t+b}{t-b} z \]
\[ z'z = pz + q \quad \text{where } p \text{ and } q \text{ are constants} \]

• We know \( z \) (depth) transformation has nothing to do with \( x \) and \( y \)

D3D perspective math (4)

\[ z'z = pz + q \quad \text{where } p \text{ and } q \text{ are constants} \]

\[ \begin{align*}
0 &= pn + q \\
f &= pf + q
\end{align*} \]

\[ \therefore p = \frac{f}{f-n} \quad \text{and} \quad q = -\frac{fn}{f-n} \]

\[ z'z = \frac{f}{f-n}z - \frac{fn}{f-n} \]

• We know (boxed equations above)
  - \( z' = 0 \) when \( z=n \) (near plane)
  - \( z' = 1 \) when \( z=f \) (far plane)

General D3D perspective matrix

\[\begin{align*}
x'z &= \frac{2n}{r-l} x - \frac{r+l}{r-l} z \\
y'z &= \frac{2n}{t-b} y - \frac{t+b}{t-b} z \\
z'z &= \frac{f}{f-n} z - \frac{fn}{f-n} \\
w'z &= z
\end{align*}\]

\[[x', y', z', w'] = [x, y, z, 1]P \quad \text{where } P = \begin{bmatrix}
\frac{2n}{r-l} & 0 & 0 & 0 \\
0 & \frac{2n}{t-b} & 0 & 0 \\
-\frac{r+l}{r-l} & -\frac{t+b}{t-b} & \frac{f}{f-n} & 1 \\
0 & 0 & -\frac{fn}{f-n} & 0
\end{bmatrix}\]

Simpler D3D perspective matrix

- Similar to orthographic projection, if \( l=-r \) and \( t=-b \), we can simplify to

\[
[x', y', z', w'] = [x, y, z, 1]P \quad \text{where } P = \begin{bmatrix}
\frac{2n}{w} & 0 & 0 & 0 \\
0 & \frac{2n}{h} & 0 & 0 \\
0 & 0 & \frac{f}{f-n} & 1 \\
0 & 0 & -\frac{fn}{f-n} & 0
\end{bmatrix}
\]

- In any case, we will have to divide by \( z \) to obtain \([x', y', z', w']\)
  - Implemented by dividing by the fourth (\( w'z \)) coordinate

Define viewing frustum

Parameters:
- FOV: Field of View
- Aspect ratio = Width/Height
- Near z
- Far z

Reparameterized D3D matrix

\[
P = \begin{bmatrix}
\frac{2n}{w} & 0 & 0 & 0 \\
0 & \frac{2n}{h} & 0 & 0 \\
0 & 0 & \frac{f}{f-n} & 1 \\
0 & 0 & -\frac{fn}{f-n} & 0 \\
\end{bmatrix}
\]

\[
cot\left(\frac{a}{2}\right) = \frac{2n}{h}
\]

\[
r = \frac{w}{h}
\]

\[
\frac{2n}{w} = \frac{2n}{rh} = \frac{2n}{r} = \frac{1}{r} \cot\left(\frac{a}{2}\right)
\]

Direct3D style

Need to replace \(w\) and \(h\) with FOV and aspect ratio

D3D perspective matrix (LHS default)

\[
[x', z, y', z', w'] = [x, y, z, 1]P \quad \text{where } P = \begin{bmatrix}
\frac{1}{r} \cdot \cot\left(\frac{a}{2}\right) & 0 & 0 & 0 \\
0 & \cot\left(\frac{a}{2}\right) & 0 & 0 \\
0 & 0 & \frac{f}{f - n} & 1 \\
0 & 0 & -\frac{fn}{f - n} & 0
\end{bmatrix}
\]

- \text{a: Field of View (FOV)}
- \text{r: aspect ratio } = \frac{\text{width}}{\text{height}}
- \text{n: near plan}
- \text{f: far plane}

- In Direct3D: \text{D3DXMatrixPerspectiveFovLH(*o,a,r,n,f)}
D3D perspective matrix (RHS weird)

\[
[x', y', z', w'] = [x, y, z, 1]P \quad \text{where } P =
\begin{bmatrix}
\frac{1}{r} \cot(\frac{a}{2}) & 0 & 0 & 0 \\
0 & \cot(\frac{a}{2}) & 0 & 0 \\
0 & 0 & \frac{f}{n-f} & -1 \\
0 & 0 & \frac{fn}{n-f} & 0
\end{bmatrix}
\]

- In Direct3D: D3DXMatrixPerspectiveFovRH(*o,a,r,n,f)
- In XNA: Matrix.CreatePerspectiveFieldOfView(a,r,n,f)

a: Field of View (FOV)  r: aspect ratio = width \over height  n: near plan  f: far plane

OpenGL/LHS perspective projection

Canonical view volume (Unity)

View Frustum (a truncated pyramid)

OpenGL/LHS perspective proj mapping

- Given a point \((x, y, z)\) within the view frustum, project it onto the near plane \(z=n\)
- We will map \(x\) from \([l, r]\) to \([-1, 1]\) and \(y\) from \([b, t]\) to \([-1, 1]\)

Canonical view volume (Unity)

OpenGL/LHS perspective projection math

\[ z'z = pz + q \quad \text{where } p \text{ and } q \text{ are constants} \]

\[ \begin{align*}
-n &= pn + q \\
f &= pf + q
\end{align*} \]

∴ \[ p = \frac{f + n}{f - n} \quad \text{and} \quad q = -\frac{2fn}{f - n} \]

\[ z'z = \frac{f + n}{f - n}z - \frac{2fn}{f - n} \]

• We know (boxed equations above)
  - \( z' = -1 \) when \( z = n \) (near plane)
  - \( z' = 1 \) when \( z = f \) (far plane)

General OpenGL/LHS perspective matrix

\[
x'z = \frac{2n}{r-l} x - \frac{r+l}{r-l} z \\
y'z = \frac{2n}{t-b} y - \frac{t+b}{t-b} z \\
z'z = \frac{f+n}{f-n} z - \frac{2fn}{f-n} \\
w'z = z
\]

\[
\begin{bmatrix}
x'z \\
y'z \\
z'z \\
w'z
\end{bmatrix} = P
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]

where \( P = \)

\[
\begin{bmatrix}
\frac{2n}{r-l} & 0 & -\frac{r+l}{r-l} & 0 \\
0 & \frac{2n}{t-b} & -\frac{t+b}{t-b} & 0 \\
0 & 0 & \frac{f+n}{f-n} & -\frac{2fn}{f-n} \\
0 & 0 & 1 & 0
\end{bmatrix}
\]

Simpler OpenGL/LHS perspective matrix

- Similar to orthographic projection, if \( l=-r \) and \( t=-b \), we can simplify to

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
w'
\end{bmatrix} = P
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]

where

\[
P = \begin{bmatrix}
\frac{2n}{w} & 0 & 0 & 0 \\
0 & \frac{2n}{h} & 0 & 0 \\
0 & 0 & \frac{f + n}{f - n} & -\frac{2fn}{f - n} \\
0 & 0 & 1 & 0
\end{bmatrix}
\]

- In any case, we will have to divide by \( z \) to obtain \([x', y', z', w']\)
  - Implemented by dividing by the fourth (w'z) coordinate

Simpler OpenGL/RHS perspective matrix

\[
\begin{bmatrix}
  x' \\
  y' \\
  z' \\
  w'
\end{bmatrix}
= P
\begin{bmatrix}
  x \\
  y \\
  z \\
  1
\end{bmatrix}
\]

where \( P = \) 
\[
\begin{bmatrix}
  \frac{1}{\cot\left(\frac{a}{2}\right)} & 0 & 0 & 0 \\
  \frac{r}{2} & \cot\left(\frac{a}{2}\right) & 0 & 0 \\
  0 & 0 & \frac{f+n}{n-f} & \frac{2fn}{n-f} \\
  0 & 0 & -1 & 0
\end{bmatrix}
\]

a: Field of View (FOV)  \( r \): aspect ratio = \( \frac{\text{width}}{\text{height}} \)  n: near plan  f: far plane

- In OpenGL: \texttt{gluPerspective}(a,r,n,f)
Unity perspective matrix

\[
\begin{bmatrix}
    x'z \\
    y'z \\
    z'z \\
    w'z
\end{bmatrix}
\]
\(= P \times \begin{bmatrix}
    x \\
    y \\
    z \\
    1
\end{bmatrix}
\]

where \(P = \begin{bmatrix}
    \frac{1}{r} \cot \left( \frac{a}{2} \right) & 0 & 0 & 0 \\
    0 & \cot \left( \frac{a}{2} \right) & 0 & 0 \\
    0 & 0 & \frac{f + n}{f - n} \begin{bmatrix}
        \frac{2fn}{f-n}
    \end{bmatrix} & \frac{2fn}{f-n} \\
    0 & 0 & -1 & 0
\end{bmatrix}\)

a: Field of View (FOV)    r: aspect ratio \(= \frac{\text{width}}{\text{height}}\)    n: near plane    f: far plane

- In Unity: Matrix4x4.Perspective(a,r,n,f)
Custom projections in Unity

• From Camera.projectionMatrix documentation:

“Use a custom projection only if you really need a non-standard projection.

This property is used by Unity's water rendering to setup an oblique projection matrix.

Using custom projections requires good knowledge of transformation and projection matrices.”
Unity’s 2-D coordinate systems

- **Viewport space:**
  - (0,0) is bottom-left
  - (1,1) is top-right

- **Screen space coordinates:**
  - z “is in world units from the camera”
  - (0,0) is bottom-left
  - (Camera.pixelWidth, Camera.pixelHeight) is top-right

- **GUI space coordinates:**
  - (0,0) is upper-left
  - (Camera.pixelWidth, Camera.pixelHeight) is bottom-right
Viewport transformation

- The actual 2D projection to the viewer
- Copy to your back buffer (frame buffer)
- Can be programmed, scaled, ...
Backface culling

- Determine “facing direction”
- Triangle order matters
- How to compute a normal vector for 2 given vectors?
  - Using **cross product** of 2 given vectors

2 Vectors

\[ \vec{V_1} = (b_1 - a_1)\hat{i} + (b_2 - a_2)\hat{j} + (b_3 - a_3)\hat{k} \]
\[ \vec{V_2} = (c_1 - a_1)\hat{i} + (c_2 - a_2)\hat{j} + (c_3 - a_3)\hat{k} \]

Cross product

\[ \vec{V_1} \times \vec{V_2} = (x_2y_3 - x_3y_2)\hat{i} + (x_3y_1 - x_1y_3)\hat{j} + (x_1y_2 - x_2y_1)\hat{k} \]
Compute the surface normal for a triangle

- Clockwise normals, LHS

\[ \vec{v}_1 = 3\mathbf{i} + 3\mathbf{j} + 0\mathbf{k} \]
\[ \vec{v}_2 = 4\mathbf{i} + 0\mathbf{j} + 0\mathbf{k} \]
Backface culling method (1)

- Check if the normal is facing the camera
- How to determine that?
  - Use Dot Product
Backface culling method (2)

- Check if the normal is facing the camera
- How to determine that?
  - Use Dot Product
Dot product method (1)

\[ \vec{A} \cdot \vec{B} = |A| |B| \cos \theta \]

\[ \vec{A} \cdot \vec{B} > 0 \quad \Rightarrow \quad -\frac{\pi}{2} < \theta < \frac{\pi}{2} \]
Dot product method (2)

\[ \vec{A} \cdot \vec{B} = |A||B| \cos \theta \]

\[ \vec{A} \cdot \vec{B} > 0 \implies -\frac{\pi}{2} < \theta < \frac{\pi}{2} \]
Dot product method (3)

\[ \vec{A} \cdot \vec{B} > 0 \Rightarrow -\frac{\pi}{2} < \theta < \frac{\pi}{2} \]
Caution!

\[ \vec{A} \cdot \vec{B} > 0 \implies -\frac{\pi}{2} < \theta < \frac{\pi}{2} \]
When to perform backface culling?

Make sure camera is in correct coordinates!
How about before you even start?

Transform camera vectors into object spaces
Or how about at the very end?

Now you can just check “winding order” in 2D

As “officially” done by OpenGL
3D clipping

- Test 6 planes if a triangle is inside, outside, or partially inside the view frustum
- Clipping creates new triangles (triangulation)
  - Interpolate new vertices info
Appendix
Clipping against a plane

- Test each vertex of a triangle
  - Outside
  - Inside
  - Partly inside

- Incurred computation overhead

- Save unnecessary computation (and bandwidth) later

- Need to know how to determine a plane

- Need to know how to determine a vertex is inside or outside a plane
Specifying a plane

- You need two things to specify a plane
  - A point on the plane \((p_0, p_1, p_2)\)
  - A vector (normal) perpendicular to the plane \((a, b, c)\)
  - Plane \(a(x - p_0) + b(y - p_1) + c(z - p_2) = 0\)
Distance calculation from a plane (1)

- Given a point R, calculate the distance
  - Distance > 0 inside the plane
  - Distance = 0 on the plane
  - Distance < 0 outside the plane

\[ d = |R - P| \cdot \cos \theta = |R - P| \cdot \frac{\vec{v} \cdot (R - P)}{|\vec{v}| \cdot |R - P|} = \frac{\vec{v} \cdot (R - P)}{|\vec{v}|} \]
Distance calculation from a plane (2)

\[ d = | R - P | \cdot \cos(180 - \theta) \]
Triangulation using interpolation

\[ s = \frac{d_1}{d_1 + d_2} \]

\[ x = a_1 + s \cdot (a_2 - a_1) \]

\[ y = b_1 + s \cdot (b_2 - b_1) \]

\[ z = c_1 + s \cdot (c_2 - c_1) \]