Perspective Projection and Texture Mapping

Computer Graphics
CMU 15-462/15-662, Fall 2016
Transforms: moving to 3D (and 3D-H)

Represent 3D transforms as 3x3 matrices and 3D-H transforms as 4x4 matrices

Scale:

\[
S_s = \begin{bmatrix}
S_x & 0 & 0 \\
0 & S_y & 0 \\
0 & 0 & S_z \\
\end{bmatrix}
\]

\[
S_s = \begin{bmatrix}
S_x & 0 & 0 & 0 \\
0 & S_y & 0 & 0 \\
0 & 0 & S_z & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Shear (in x, based on y, z position):

\[
H_{x,d} = \begin{bmatrix}
1 & d_y & d_z \\
0 & 1 & 0 \\
0 & 0 & 1 \\
\end{bmatrix}
\]

\[
H_{x,d} = \begin{bmatrix}
1 & d_y & d_z & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

Translate:

\[
T_b = \begin{bmatrix}
1 & 0 & 0 & b_x \\
0 & 1 & 0 & b_y \\
0 & 0 & 1 & b_z \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]
Rotations in 3D

Rotation about x axis:

\[ \mathbf{R}_{x,\theta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos \theta & -\sin \theta \\ 0 & \sin \theta & \cos \theta \end{bmatrix} \]

Rotation about y axis:

\[ \mathbf{R}_{y,\theta} = \begin{bmatrix} \cos \theta & 0 & \sin \theta \\ 0 & 1 & 0 \\ -\sin \theta & 0 & \cos \theta \end{bmatrix} \]

Rotation about z axis:

\[ \mathbf{R}_{z,\theta} = \begin{bmatrix} \cos \theta & -\sin \theta & 0 \\ \sin \theta & \cos \theta & 0 \\ 0 & 0 & 1 \end{bmatrix} \]
Rotation about an arbitrary axis

Q: Do you know how to derive it?

\[
\begin{bmatrix}
\cos \theta + u_x^2 (1 - \cos \theta) & u_x u_y (1 - \cos \theta) - u_z \sin \theta & u_x u_z (1 - \cos \theta) + u_y \sin \theta \\
u_y u_x (1 - \cos \theta) + u_z \sin \theta & \cos \theta + u_y^2 (1 - \cos \theta) & u_y u_z (1 - \cos \theta) - u_x \sin \theta \\
u_z u_x (1 - \cos \theta) - u_y \sin \theta & u_z u_y (1 - \cos \theta) + u_x \sin \theta & \cos \theta + u_z^2 (1 - \cos \theta)
\end{bmatrix}
\]

Just memorize this matrix! :-)
Review: simple camera transform

- Consider object in world at (10, 2, 0)
- Consider camera at (4, 2, 0), looking down x axis

How do you compute the transform from world to the camera coordinate system* (camera at the origin, looking down z axis)?
Translating object vertex positions by (-4, -2, 0) yields position relative to camera.
Rotation about $y$ axis by $\pi/2$ gives position of object in coordinate system where camera’s view direction is aligned with the -$z$ axis

* The convenience of the camera coordinate system will become clear soon!
Camera with arbitrary orientation

Consider camera looking in direction $w$
What transform places the object in the camera coordinate system?

Form orthonormal basis around $w$: (see $u$ and $v$)
Compute transformation that maps $u$ to x-axis, $v$ to y-axis, and $w$ to -z axis

Answer: Consider rotation matrix:

$$
R = \begin{bmatrix}
u_x & v_x & -w_x \\
u_y & v_y & -w_y \\
u_z & v_z & -w_z
\end{bmatrix}
$$

$$
R^T u = [u \cdot u \ v \cdot u \ -w \cdot u]^T = [1 \ 0 \ 0]^T
$$

$$
R^T v = [u \cdot v \ v \cdot v \ -w \cdot v]^T = [0 \ 1 \ 0]^T
$$

$$
R^T w = [u \cdot w \ v \cdot w \ -w \cdot w]^T = [0 \ 0 \ -1]^T
$$
Perspective projection
Early paintings: incorrect perspective

‘Jesus Before the Caïf’, by Giotto (1305)
Early paintings: incorrect perspective

‘Jesus Before the Caïf’, by Giotto (1305)
Geometrically correct perspective in art

Brunelleschi, elevation of Santo Spirito, 1434-83, Florence

Masaccio – The Tribute Money c.1426-27 Fresco, The Brancacci Chapel, Florence
Later... rejection of proper perspective projection
Basic perspective projection

Input: point in 3D-H
Output: perspective projected point

Assumption: Pinhole camera at (0,0) looking down z
Review: homogeneous coordinates

$wx = [wx_x \ wx_y \ w]^T$

$x = [x_x \ x_y \ 1]^T$

Many points in 2D-H correspond to same point in 2D
$x$ and $wx$ correspond to the same 2D point
(divide by $w$ to convert 2D-H back to 2D)
Basic perspective projection

Desired perspective projected result (2D point):

\[ p_{2D} = \left( \frac{x_x}{x_z}, \frac{x_y}{x_z} \right) \]

Input: point in 3D-H

Applying map to get projected point in 3D-H

Point projected to 2D-H (drop z coord)

Point in 2D (homogeneous divide)

Assumption: Pinhole camera at (0,0) looking down z
The view frustum

View frustum: region in space that may appear on the screen

How can we change the shape of the view frustum?

Want a transformation that maps view frustum to a unit cube (computing screen coordinates in that space is then trivial)
The view frustum

Pinhole Camera (0,0)

Projected on virtual screen

First step: apply perspective transform & normalize z-coord
What are the values of a and b?

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

\[ Px = (x_x, x_y, x_z, x_z) \]

\[ P_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix} \]

\[ x_1 = (x_{1x}, x_{1y}, -z_{Near}, 1); \quad Px_1 = (x_{1x}, x_{1y}, -z_{Near}, z_{Near}) \]

\[ x_7 = (x_{7x}, x_{7y}, -z_{Far}, 1); \quad Px_7 = (x_{7x}, x_{7y}, z_{Far}, z_{Far}) \]
The view frustum

Changing the shape of the view frustum:

$\theta$: field of view in y direction ($h = \tan \left( \frac{\theta}{2} \right)$)

$r = \text{aspect ratio} = \text{width} / \text{height}$

After normalizing z-component, rescale x and y components such that frustum maps to unit cube

$$f = 2\cot\left( \frac{\theta}{2} \right)$$
The view frustum

Putting it all together, transformation matrix that maps view frustum to unit cube:

\[
P = \begin{bmatrix}
\frac{f}{\text{aspect}} & 0 & 0 & 0 \\
0 & f & 0 & 0 \\
0 & 0 & \frac{z_{\text{far}}+z_{\text{near}}}{z_{\text{near}}-z_{\text{far}}} & 2 \times z_{\text{far}} \times z_{\text{near}} \\
0 & 0 & -1 & 0 \\
\end{bmatrix}
\]

Q1: What kinds of features does this transformation preserve?  
Q2: Is this transformation matrix invertible?
Transformations recap

Modeling transforms:
Position object in scene

Viewing (camera) transform:
positions objects in coordinate space relative to camera
Canonical form: camera at origin looking down \(-z\)

Projection transform + homogeneous divide:
Performs perspective projection
Canonical form: visible region of scene contained within unit cube

Screen transform:
objects now in 2D screen coordinates
Review exercise: screen transform *

Convert points in normalized coordinate space to screen pixel coordinates

Example:
All points within (-1,1) to (1,1) region are on screen
(1,1) in normalized space maps to (W,0) in screen

Step 1: reflect about x
Step 2: translate by (1,1)
Step 3: scale by (W/2,H/2)

* Adopting convention that top-left of screen is (0,0) to match SVG convention in Assignment 1. Many 3D graphics systems like OpenGL place (0,0) in bottom-left. In this case what would the transform be?
Transformations recap

Modeling transforms: Position object in scene

Viewing (camera) transform: positions objects in coordinate space relative to camera
Canonical form: camera at origin looking down -z

Projection transform + homogeneous divide:
Performs perspective projection
Canonical form: visible region of scene contained within unit cube

Screen transform:
objects now in 2D screen coordinates

Compute screen coverage from 2D object position
Coverage\((x,y)\)

A few lectures ago we discussed how to sample coverage given the 2D position of the triangle’s vertices.
Consider sampling color($x,y$)

What is the triangle’s color at point $x$?
Review: interpolation in 1D

Between $x_2$ and $x_3$:

$$f_{\text{recon}}(t) = (1 - t)f(x_2) + tf(x_3)$$

where:

$$t = \frac{(x - x_2)}{x_3 - x_2}$$

(measures how far $x$ is from $x_2$)
Consider similar behavior on triangle

Color depends on distance from base

\[ \text{color at } x = (1 - t) [0 \ 0 \ 1] + t [0 \ 0 \ 0] \]

\[ t = \frac{\text{distance from } x \text{ to } b - a}{\text{distance from } c \text{ to } b - a} \]

How can we interpolate in 2D between three values?
Interpolation via barycentric coordinates

\[ \mathbf{b} - \mathbf{a} \text{ and } \mathbf{c} - \mathbf{a} \text{ form a non-orthogonal basis for points in triangle.} \]

We can therefore write:

\[
\mathbf{x} = \mathbf{a} + \beta (\mathbf{b} - \mathbf{a}) + \gamma (\mathbf{c} - \mathbf{a})
\]

\[
= (1 - \beta - \gamma) \mathbf{a} + \beta \mathbf{b} + \gamma \mathbf{c}
\]

\[
= \alpha \mathbf{a} + \beta \mathbf{b} + \gamma \mathbf{c}
\]

with

\[
\alpha + \beta + \gamma = 1
\]

Color at \( \mathbf{x} \) is **affine** combination of color at three triangle vertices.

\[
\mathbf{x}_{\text{color}} = \alpha \mathbf{a}_{\text{color}} + \beta \mathbf{b}_{\text{color}} + \gamma \mathbf{c}_{\text{color}}
\]
Barycentric coordinates as ratio of areas

In the context of a triangle, Barycentric coordinates are also called areal coordinates

\[
\alpha = \frac{A_A}{A} \\
\beta = \frac{A_B}{A} \\
\gamma = \frac{A_C}{A}
\]
Barycentric interpolation as an affine map

\[ f_x = \alpha f_a + \beta f_b + \gamma f_c \]

\[ = \begin{bmatrix} f_a & f_b & f_c \end{bmatrix} \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} \]

but

\[ \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = A \begin{bmatrix} x_x \\ x_y \\ 1 \end{bmatrix} \]

so

\[ f_x = Ax_x + Bx_y + C \]
Direct evaluation of surface attributes

For any surface attribute, value at $x$ is

$$f_x = Ax_x + Bx_y + C$$

To find, $A$, $B$ and $C$, plug in values at triangle vertices, where we know the values of the attribute $(f_a, f_b, f_c)$

$$f_a = Aa_x + Ba_y + C$$
$$f_b = Ab_x + Bb_y + C$$
$$f_c = Ac_x + Bc_y + C$$

3 equations, solve for 3 unknowns $(A, B, C)$

Note: $A$, $B$ and $C$ will be different for different attributes
But what are we interpolating again?

What are \( a, b \) and \( c \)?
Perspective-incorrect interpolation

Due to projection, affine function (barycentric interpolation) in screen XY coordinates does not correspond to values that vary linearly on a triangle with vertices at different depths.

Attribute values must be interpolated linearly in 3D object space!
An example: perspective-incorrect interpolation
Perspective-correct interpolation

Assume triangle attribute varies linearly across the triangle

Attribute’s value at 3D (non-homogeneous) point \( P = [x \ y \ z]^T \) is therefore:

\[
f(x, y, z) = ax + by + cz
\]

Project \( P \), get 2D homogeneous representation:

\[
[x_{2D-H} \ y_{2D-H} \ w]^T = [x \ y \ z]^T
\]

Rewrite attribute equation for \( f \) in terms of 2D homogeneous coordinates:

\[
f = ax_{2D-H} + by_{2D-H} + cw
\]

\[
\frac{f}{w} = a \frac{x_{2D-H}}{w} + b \frac{y_{2D-H}}{w} + c
\]

Where \( [x_{2D} \ y_{2D}]^T \) are projected screen 2D coordinates (after homogeneous divide)

So … \( \frac{f}{w} \) is affine function of 2D screen coordinates…
Perspective-correct interpolation

Attribute values vary linearly across triangle in 3D, but not in projected screen XY

Affine combinations of projected attribute values \((f/w)\) correspond to linear interpolation in 3D

To evaluate surface attribute \(f\) at every covered sample:

- Evaluate \(\frac{1}{w}(x,y)\) (from precomputed equation for \(\frac{1}{w}\))
- Reciprocate \(\frac{1}{w}(x,y)\) to get \(w(x,y)\)
- For each triangle attribute:
  - Evaluate \(\frac{f}{w}(x,y)\) (from precomputed equation for \(\frac{f}{w}\))
  - Multiply \(\frac{f}{w}(x,y)\) by \(w(x,y)\) to get \(f(x,y)\)

Works for any surface attribute \(f\) that varies linearly across triangle:
e.g., color, depth, texture coordinates
Consider the following example
Texture mapping
Many uses of texture mapping

Define variation in surface reflectance
Describe surface material properties

Multiple layers of texture maps for color, logos, scratches, etc.
Normal mapping

Use texture value to perturb surface normal to give appearance of a bumpy surface. Observe: smooth silhouette and smooth shadow boundary indicates surface geometry is not bumpy.

Rendering using high-resolution surface geometry (note bumpy silhouette and shadow boundary).
Represent precomputed lighting and shadows

Grace Cathedral environment map

Environment map used in rendering
Texture coordinates

“Texture coordinates” define a mapping from surface coordinates (points on triangle) to points in texture domain.

myTex(u,v) is a function defined on the [0,1]² domain:

myTex : [0,1]² → float3
(represented by 2048x2048 image)

Eight triangles (one face of cube) with surface parameterization provided as per-vertex texture coordinates.

Today we’ll assume surface-to-texture space mapping is provided as per vertex values (Methods for generating surface texture parameterizations will be discussed in a later lecture)
Visualization of texture coordinates

Texture coordinates linearly interpolated over triangle
More complex mapping

Each vertex has a coordinate \((u,v)\) in texture space.
(Actually coming up with these coordinates is another story!)
Simple texture mapping operation

for each covered screen sample (x,y):
  \((u,v) = \text{evaluate texcoord value at } (x,y)\)
  \begin{align*}
    \text{float3 texcolor} &= \text{texture.sample}(u,v); \quad \Leftarrow \text{“just” an image lookup…} \\
    \text{set sample’s color to texcolor;}
  \end{align*}
Texture mapping adds detail

Rendered result

Triangle vertices in texture
Texture mapping adds detail

- rendering without texture
- rendering with texture
- texture image

Each triangle “copies” a piece of the image back to the surface.
Another example: Sponza

Notice texture coordinates repeat over surface.
Textured Sponza
Example textures used in Sponza
Texture space samples

Sample positions in XY screen space

Sample positions are uniformly distributed in screen space (rasterizer samples triangle’s appearance at these locations)

Sample positions in texture space

Texture sample positions in texture space (texture function is sampled at these locations)

Q: what does it mean that equally-spaced points in screen space move further apart in texture space?
Recall: aliasing

Undersampling a high-frequency signal can result in aliasing.
Aliasing due to undersampling texture

No pre-filtering of texture data
(resulting image exhibits aliasing)

Rendering using pre-filtered texture data
Aliasing due to undersampling (zoom)
Filtering textures

▪ Minification:
  - Area of screen pixel maps to large region of texture (filtering required -- averaging)
  - One texel corresponds to far less than a pixel on screen
  - Example: when scene object is very far away
  - Texture map is too large, it contains more details than screen can display

▪ Magnification:
  - Area of screen pixel maps to tiny region of texture (interpolation required)
  - One texel maps to many screen pixels
  - Example: when camera is very close to scene object
  - Texture map is too small

Figure credit: Akeley and Hanrahan
Mipmap (L. Williams 83)

Level 0 = 128x128  
Level 1 = 64x64  
Level 2 = 32x32  
Level 3 = 16x16

Level 4 = 8x8  
Level 5 = 4x4  
Level 6 = 2x2  
Level 7 = 1x1

Idea: prefilter texture data to remove high frequencies
Texels at higher levels store integral of the texture function over a region of texture space (downsampled images)
Texels at higher levels represent low-pass filtered version of original texture signal
What is the storage overhead of a mipmap?

Slide credit: Akeley and Hanrahan
Computing $d$

Compute differences between texture coordinate values of neighboring screen samples
Computing $d$

Compute differences between texture coordinate values of neighboring screen samples

$$\begin{align*}
\frac{du}{dx} &= u_{10} - u_{00} \\
\frac{dv}{dx} &= v_{10} - v_{00} \\
\frac{du}{dy} &= u_{01} - u_{00} \\
\frac{dv}{dy} &= v_{01} - v_{00}
\end{align*}$$

$$L = \max \left( \sqrt{\left( \frac{du}{dx} \right)^2 + \left( \frac{dv}{dx} \right)^2}, \sqrt{\left( \frac{du}{dy} \right)^2 + \left( \frac{dv}{dy} \right)^2} \right)$$

$mip-map d = \log_2 L$
Sponza (bilinear resampling at level 0)
Sponza (bilinear resampling at level 2)
Sponza (bilinear resampling at level 4)
Visualization of mip-map level
(bilinear filtering only: $d$ clamped to nearest level)
“Tri-linear” filtering

\[ \text{lerp}(t, v_1, v_2) = v_1 + t(v_2 - v_1) \]

Bilinear resampling:
- four texel reads
- 3 lerps (3 mul + 6 add)

Trilinear resampling:
- eight telex reads
- 7 lerps (7 mul + 14 add)

Figure credit: Akeley and Hanrahan
Visualization of mip-map level
(trilinear filtering: visualization of continuous $d$)
Summary: a texture sampling operation

1. Compute u and v from screen sample x,y (via evaluation of attribute equations)
2. Compute du/dx, du/dy, dv/dx, dv/dy differentials from screen-adjacent samples.
3. Compute d
4. Convert normalized texture coordinate (u,v) to texture coordinates texel_u, texel_v
5. Compute required texels in window of filter
6. Load required texels (need eight texels for trilinear)
7. Perform tri-linear interpolation according to (texel_u, texel_v, d)

Takeaway: a texture sampling operation is not "just" an image pixel lookup! It involves a significant amount of math.

All modern GPUs have dedicated fixed-function hardware support for performing texture sampling operations.
Texturing summary

- Texture coordinates: define mapping between points on triangle’s surface (object coordinate space) to points in texture coordinate space

- Texture mapping is a sampling operation and is prone to aliasing
  - Solution: prefilter texture map to eliminate high frequencies in texture signal
  - Mip-map: precompute and store multiple resampled versions of the texture image (each with different amounts of low-pass filtering)
  - During rendering: dynamically select how much low-pass filtering is required based on distance between neighboring screen samples in texture space
    - Goal is to retain as much high-frequency content (detail) in the texture as possible, while avoiding aliasing