Perspective Projection Transformations, Geometry and Texture Mapping
The view frustum

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

\[
P = \begin{bmatrix}
\frac{f}{\text{aspect}} & 0 & 0 & 0 \\
0 & f & 0 & 0 \\
0 & 0 & z_{\text{far}} + z_{\text{near}} & 2 \times z_{\text{far}} \times z_{\text{near}} \\
0 & 0 & \frac{z_{\text{near}} - z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} & -1 \\
0 & 0 & \frac{z_{\text{near}} - z_{\text{far}}}{z_{\text{near}} - z_{\text{far}}} & 0
\end{bmatrix}
\]
How do we go beyond cube-man?
“Implicit” Representations of Geometry

• Points aren’t known directly, but satisfy some relationship
  • unit sphere: all points x such that \( x^2 + y^2 + z^2 = 1 \)
  • More generally, \( f(x, y, z) = 0 \)
Bobby Surfaces (Implicit)

- Instead of Booleans, gradually blend surfaces together:

  ![Blobby Surfaces](image)

- Easier to understand in 2D:

  \[ \phi_p(x) := e^{\mid x - p \mid^2} \]  
  \[ f := \phi_p + \phi_q \]  
  (Gaussian centered at p)
  (Sum of Gaussians centered at different points)

  \[ f = \frac{1}{2} \]
Implicit surfaces

An example...
 Implicit Representations - Pros & Cons

• Pros:
  - description can be very compact
  - easy to determine if a point is inside/outside (just plug it in!)
  - other queries may also be easy (e.g., distance to surface)
  - easy to handle changes in topology (e.g., fluid merging)

• Cons:
  - expensive to find all points in the shape (e.g., for drawing)
  - May be difficult to model complex shapes
What about explicit representations?
Point Cloud (Explicit)

- Simplest representation: list of points \((x,y,z)\)
- Often augmented with *normals*
- Easily represent any kind of geometry
- Useful for LARGE datasets (>>1 point/pixel)
- Difficult to draw in undersampled regions
- Hard to do processing / simulation

![Point Cloud Examples](image)
Polygonal Mesh (Explicit)

- Store vertices *and* polygons (most often triangles or quads)
- Easier to do processing/simulation, adaptive sampling
- More complicated data structures
- Perhaps most common representation in graphics
Polygonal Mesh

• What is a polygon

- Vertices
  \( v_0, v_1, \ldots, v_{n-1} \)

- Edges
  \( \{(v_0, v_1), \ldots, (v_{n-2}, v_{n-1})\} \)

- Planar and non-self-intersecting
Polygonal Mesh

- Set of connected polygons

\[ M = \langle V, E, F \rangle \]

vertices \quad edges \quad faces
Polygonal Mesh

- Properties

  • Every edge belongs to at least one polygon
  • The intersection of two polygons in $M$ is either empty, a vertex, or an edge
Polygonal Mesh

- Definitions
  - Vertex degree (Valence): Number of edges incident to a vertex
Polygonal Mesh

- Definitions

- **Vertex degree (valence):** Number of edges incident to a vertex
- **Boundary:** the set of all edges that belong to only one polygon
Manifolds

- Surface locally homeomorphic to a disk
- Closed manifolds divides space into two
Manifold Mesh

- In a manifold mesh, not allowed:
Mesh Data Structures

- Store geometry & topology
  - **Geometry**: vertex locations
  - **Topology**: how vertices are connected (edges/faces)
  - **Attributes**: Normal, color, etc.
Mesh Data Structures

- Operations to be supported
  - Rendering
Mesh Data Structures

- Operations to be supported
  - Rendering
  - Geometry queries

Example:
What are the vertices of face $F$?
Mesh Data Structures

- Operations to be supported
  - Rendering
  - Geometry queries
  - Modifications

  Example:
  Remove vertex $v$. 
Mesh Data Structures

- Triangle List

<table>
<thead>
<tr>
<th>Triangles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Coord.</td>
</tr>
<tr>
<td>((x_0, y_0, z_0))</td>
</tr>
<tr>
<td>((x_3, y_3, z_3))</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

- Simple
- No connectivity
- Redundant
- STL file format
## Mesh Data Structures

- **Indexed Face Set**

<table>
<thead>
<tr>
<th></th>
<th>Triangles</th>
<th>Vertices</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex Index</td>
<td>Vertex Index</td>
<td>Vertex Index</td>
</tr>
<tr>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Index</th>
<th>Coord.</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>((x_0, y_0, z_0))</td>
</tr>
<tr>
<td>1</td>
<td>((x_1, y_1, z_1))</td>
</tr>
<tr>
<td>2</td>
<td>((x_2, y_2, z_2))</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Mesh Data Structures

- Indexed Face Set
  - Avoids redundancy
  - OBJ, OFF, WRL file formats
  - Stores connectivity, but still
    - Costly geometric queries
    - Costly mesh modifications
Where is all this data coming from?
Sources of geometry

- Acquired real-world objects via 3D Scanning
Sources of geometry

- Acquired real-world objects via 3D Scanning
Sources of geometry

- Digital 3D modeling
Sources of geometry

- Digital 3D modeling
Subdivision Surfaces (Explicit)

- Smooth out a control curve
  - Insert new vertex at each edge midpoint
  - Update vertex positions according to fixed rule
  - For careful choice of averaging rule, yields smooth curve
    - E.g. average with “next” neighbor (Chaikin)
Subdivision Surfaces (Explicit)

- Start with coarse polygon mesh ("control cage")
- Subdivide each element
- Update vertices via local averaging
- Many possible rule:
  - Catmull-Clark (quads)
  - Loop (triangles)
  - ...
- Common issues:
  - interpolating or approximating?
  - continuity at vertices?
Subdivision in Action (Pixar’s “Geri’s Game”)

So, that’s where triangles come from...
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

**Diagram:**
- **a** (red [0,0,1])
- **b** (green [0,1,0])
- **c** (blue [0,0,1])

**Text:**

- **b-a** and **c-a** form a non-orthogonal basis for points in triangle.

We can therefore write:

\[ x = a + \beta(b - a) + \gamma(c - a) \]
\[ = (1 - \beta - \gamma)a + \beta b + \gamma c \]
\[ = a_\alpha + b_\beta + c_\gamma \]

with

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

Color at **x** is **affine** combination of color at three triangle vertices.

\[ x_{\text{color}} = \alpha a_{\text{color}} + \beta b_{\text{color}} + \gamma c_{\text{color}} \]
In the context of a triangle, Barycentric coordinates are also called areal coordinates.

Barycentric coordinates as ratio of areas

\[ \alpha = \frac{A_A}{A} \]
\[ \beta = \frac{A_B}{A} \]
\[ \gamma = \frac{A_C}{A} \]

Q: Why must coordinates sum to 1?
Q: Why must coordinates be between 0 and 1?
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

\[ f_x = Ax_x + Bx_y + C \]

These constants “bake in” attribute values and barycentric coordinates.
Direct evaluation of surface attributes

For any surface attribute, value at $x$ is

$$f_x = A x_x + B x_y + C$$

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

$$f_a = A a_x + B a_y + C$$
$$f_b = A b_x + B b_y + C$$
$$f_c = A c_x + B c_y + C$$

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 perspective projection (homogenous divide), barycentric interpolation of values in screen XY coordinates does not correspond to values that vary linearly on original triangle!

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

Attribute values \( f \) vary linearly across triangle in 3D. Due to perspective projection, we \( f/z \) varies linearly in screen coordinates, and not \( f \) directly.

Basic recipe:
- Compute depth \( z \) at each vertex
- Evaluate \( Z := 1/z \), \( P := f/z \) at each vertex
- Interpolate \( Z \) and \( P \) using standard (2D) barycentric coordinates
- At each fragment, divide interpolated \( P \) by interpolated \( Z \) to get \( f \)

Works for any surface attribute \( f \) that varies linearly across triangle: e.g., color, depth, texture coordinates

For complete derivation, see Low, “Perspective-correct Interpolation”
What do we still need to create a scene like this?
Texture mapping
Many uses of texture mapping

Define variation in surface reflectance

Pattern on ball

Wood grain on floor
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

Original model  With ambient occlusion  Extracted ambient occlusion map

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 : \([0,1]^2 \rightarrow \text{float3}\) (represented by 2048x2048 image)

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

Location of highlighted triangle in texture space shown in red.

Final rendered result (entire cube shown).

Location of triangle after projection onto screen shown in red.

Surface-to-texture space mapping is provided as per vertex attributes.
Visualization of texture coordinates

Texture coordinates linearly interpolated over triangle
More complex mapping

Visualization of texture coordinates

Triangle vertices in texture space

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) = evaluate texcoord value at (x,y)
  float3 texcolor = texture.sample(u,v); ← “just” an image lookup…
  set sample’s color to texcolor;
Texture mapping adds detail

Rendered result

Triangle vertices in texture
Texture mapping adds detail

rendering without 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.

1D example

$1D$ example

2D examples: Moiré patterns,
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)

No pre-filtering of texture data

Rendering using pre-filtered texture data
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
Filtering textures

Actual texture: 700x700 image (only a crop is shown)

Actual texture: 64x64 image

Texture minification

Texture magnification
Mipmap (L. Williams 83)

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
Mipmap (L. Williams 83)

Williams’ original proposed mip-map layout

What is the storage overhead of a mipmap?

“Mip hierarchy” level $= d$

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

$$\frac{du}{dx} = u_{10} - u_{00}, \quad \frac{dv}{dx} = v_{10} - v_{00}$$
$$\frac{du}{dy} = u_{01} - u_{00}, \quad \frac{dv}{dy} = v_{01} - v_{00}$$

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

$mip-map \ d = \log_2 L$
Computing $d$

Compute differences between texture coordinate values of neighboring screen samples

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

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

\[
mip\text{-}map \ d = \log_2 L
\]
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)
Visualization of mip-map level
(trilinear filtering: visualization of continuous $d$)
“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 texel reads
- 7 lerps (7 mul + 14 add)

Figure credit: Akeley and Hanrahan
Pixel area may not map to isotropic region in texture space.
Proper filtering requires anisotropic filter footprint.

Texture space: viewed from camera with perspective.

Overblurring in $u$ direction.

Modern solution: Combine multiple mipmap samples.
Summary: texture filtering using the mip map

- Small storage overhead (33%)
  - Mipmap is 4/3 the size of original texture image

- For each isotropically-filtered sampling operation
  - Constant filtering cost (independent of $d$)
  - Constant number of texels accessed (independent of $d$)

- Combat aliasing with prefiltering, rather than supersampling
  - Recall: we used supersampling to address aliasing problem when sampling coverage

- Bilinear/trilinear filtering is isotropic and thus will “overblur” to avoid aliasing
  - Anisotropic texture filtering provides higher image quality at higher compute and memory bandwidth cost
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 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
What you know how to do (at this point in the course)

Position objects and the camera in the world

Determine the position of objects relative to the camera

Project objects onto the screen

Sample triangle coverage

Compute triangle attribute values at covered sample points

Sample texture maps