Outline for Today

- Rendering intro
- Culling & clipping
- Scan conversion
- Midterm review
Rendering

- Fancier term: *Image Synthesis*
- Synthesis of a 2D image from a 3D scene description
- Result is a 2D array of pixels
  - Red, Green, Blue values (range 0-255 or 0.0-1.0)
  - Can also have: opacity (“alpha”), depth (“Z”), …
- *Rasterization* = determining which pixels are drawn by a given object
Hardware vs. Software Rendering

- Highest quality rendering is done by software
  - Algorithms such as “ray tracing”, “photon maps”, etc...
  - Fanciest lighting, shadowing, surface shading, smoke & fog, etc.
  - Can take minutes or hours to compute an image
  - RenderMan (Pixar), Dali (Henrik Wann Jensen), RADIANCE, POVRay, ...

- Modern computers often have special-purpose 3D rendering hardware.
  - “GPU” == Graphics Processing Unit. (Nvidia, ATI)
  - Hardware implements the traditional $3D$ graphics rendering pipeline
  - Very fast, but relatively simple algorithm:
    - Limits ability to get subtle shadows, reflections, etc.
    - Limits on complexity of surface shading description, etc.
  - Continually improving, driven by games industry.

- (Modern graphics hardware is programmable, blurring the distinction between hardware & software rendering.)

- We will start with algorithms that are used by GPUs, but we’ll do them in software.
3-D Graphics Rendering Pipeline

1. Primitives
2. Modeling Transformation
3. Viewing Transformation
4. Culling
5. Lighting & Shading
6. Clipping
7. Projection
8. Scan conversion, Hiding
9. Image

Spaces:
- Object space
- World space
- Camera space
- Normalized view space
- Image space, Device coordinates

(I added this step to the diagram)

Today
Rendering Triangle Sets

- Will focus on triangles for now
  - Most basic and useful
  - Algorithms also for lines, points, polygons, circles, ellipses, …

- Assume we have colors
  - I.e., colors assigned per-vertex
  - Next week we’ll look at lighting
We’ve already done culling

- Assume we’ve already culled to the view volume:
  - We’ve tossed out objects that we know are outside the view
  - Does that mean everything that remains will be drawn…?
More culling, and clipping

- The view volume culling may have been coarse
  ⇒ per-triangle *view volume culling*
- Some triangles may intersect the edge of the view volume
  ⇒ *clipping*
- Some triangles may be on the back sides of objects
  ⇒ *backface culling*
- Some triangles may be obscured by other triangles
  ⇒ *hidden surface elimination*, AKA *hiding*
- Some triangles may be degenerate
  ⇒ *degenerate culling*
- We will do culling/clipping before we rasterize triangles
  - We will do hidden-surface elimination at the last step
Outline for Today

- Rendering intro
- *Culling & clipping*
- Scan conversion
- Midterm review
Culling

- The sooner we can detect that a triangle is not going to be visible, the less time has to be spent processing it.
- We did object-level frustum culling.
  - Now we cull individual triangles.
- There are three common reasons to cull a triangle:
  - If it doesn’t lie within the view volume (view frustum culling)
  - If it is facing ‘away’ from the viewer (backface culling)
  - If it is degenerate (area=0)
- We’ll get to the first case later, when we do clipping.
Backface Culling

- Commonly use triangles to model the *surface* of a solid object
- If the object has no holes, the triangles will only be seen from the outside.
  - Consider the triangles as “one-sided”, i.e. only visible from the ‘front’
  - If the ‘back’ of the triangle is facing the camera, it can be culled
- Back facing triangles should be culled as early as possible
  - Expect roughly 50% of triangles in a scene to be back facing.
- Usually, backface culling is done before clipping:
  - a very quick operation
  - affects a much larger percentage of triangles than clipping
Backface Culling

- By convention, the front is the side where the vertices are ordered counterclockwise:

- Why not backface cull based on specified normals?
  - Normals not always specified
  - Per-vertex normals might not agree
  - Not used in same section of hardware:
    - normal vector used for lighting/shading only,
    - not necessarily available to clipping/culling/rasterizing units

- Most renderers allow triangles to be defined as one- or two-sided.
  - Two-sided triangles not backface culled
  - Used for thin objects, non-closed objects

- Many renderers also allow:
  - specifying whether vertices are ordered clockwise or counterclockwise
  - specifying whether to cull front or back faces
  - instead of culling, draw in a different color
Backface Culling

- Can cull in any convenient space
  - Usually backface cull in camera space
  - Can also backface cull in object space
    - transform camera position into object coords
    - don’t have to transform triangles at all before culling

- Define the eye vector: \( \vec{e} = \) vector from the triangle to the camera position
- Compute the triangle's normal (doesn't need to be unit length):
  \[
  \vec{n} = (\vec{p}_1 - \vec{p}_0) \times (\vec{p}_2 - \vec{p}_0)
  \]
- Check if \( \vec{n} \) is pointing in the same direction as \( \vec{e} \):
  cull if \( \vec{n} \cdot \vec{e} \leq 0 \)
Degenerate Culling

- A degenerate triangle has no area, nothing to draw
  - Can happen if all 3 vertices lie in a straight line
  - Can happen if 2 or all 3 vertices are located at the exact same place

- The backface cull test will automatically reject these, as the normal will be zero:

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

Cull if \( \vec{n} \cdot \vec{e} \leq 0 \)
Clipping

- For triangles that intersect the faces of the view volume
  - Partly on screen, partly off
  - Don’t want to rasterize the parts that are offscreen
  - Cut away the parts outside the view volume.

- Algorithm works on one triangle at a time
  - Fancier algorithms could work on strips or meshes, and share some work
Clipping planes

- The view volume is defined by 6 planes
  - Orthographic, perspective, or normalized view cube
  - Each plane defines an inside and an outside

- A single triangle might intersect any of them
  - Geometrically possible to intersect all 6 at once!
  - More common to intersect at most one or two
  - Turns out not to make a difference for the algorithm…
Clipping Algorithm

- Algorithm: test triangle against each plane
  - If triangle intersects the plane, clip it so it lies entirely inside:
    - May become one new triangle
    - May become two new triangles
- These are then tested against the remaining clipping planes
- After every plane has been tested, resulting triangle(s) are inside
Clipping to a plane

- Determine which of the triangle’s 3 vertices are on which side of the plane
  - If all 3 vertices are on the ‘inside’, then the triangle doesn’t need to be clipped
  - If all 3 vertices are on the ‘outside’, then the triangle is completely off-screen
    - Throw it away
    - This is *per-triangle culling* -- comes for free when clipping
  - If 1 vertex is inside, then clip: create two vertices on the plane, and a single new triangle
  - If 2 vertices are inside, clip: create two vertices on the plane, and two new triangles

Inside

Outside
Remember: Signed Distance to Plane

- Plane represented by normal $\mathbf{n}$ and distance $d$ from origin
- The distance has a sign:
  - positive on the side of the plane the normal points to
  - negative on the opposite side
  - (0 exactly on the plane)

$$\text{dist}(\mathbf{x}) = \mathbf{x} \cdot \mathbf{n} - d$$
Remember: view volume with oriented planes

- Normal of each plane points outside
  - “outside” means positive distance
  - “inside” means negative distance
Vertex Inside/Outside Testing

- Similar to frustum culling
- Compute signed distance to plane for each of the three vertices, $v_0$, $v_1$, and $v_2$
  - Negative (or 0): vertex is inside the view volume
  - Positive: vertex is outside the view volume

$$d_0 = \text{dist}(v_0)$$
$$d_1 = \text{dist}(v_1)$$
$$d_2 = \text{dist}(v_2)$$
Triangle clipping

- For triangle that intersects a clipping plane
  1. Determine which two edges intersect plane
     • Exactly two edges will have opposite sign for each vertex
Triangle clipping

- For triangle that intersects a clipping plane:
  1. Determine which two edges intersect plane
  2. Compute intersection of each of those two edges with plane
     - (we’ll see how in a moment)
Triangle clipping

For triangle that intersects a clipping plane:
1. Determine which two edges intersect plane
2. Compute intersection of each of those two edges with plane
3. Construct new triangle or triangles.
Edge-Plane Intersection

- Given triangle edge with \( v_a \) and \( v_b \) on opposite sides of the plane, having signed distances \( d_a \) and \( d_b \)
- Compute point \( x \) where the edge intersects the plane:

\[
x = (1 - t)v_a + tv_b
\]

where \( t = \frac{d_a}{d_a - d_b} \), the fractional position of the plane between \( v_a \) and \( v_b \)
Clipping: Spaces

- Clipping can be done in any convenient space
- Can clip in camera space
  - Ordinary 3D space
  - Clipping planes are conveniently described in this space
  - But must transform vertices to camera space, clip, then transform rest of the way
- Can clip in normalized view space
  - Clipping planes are very convenient (faces of normalized cube)
  - But have already performed divide-by-w (expensive)
- If careful, can clip after applying perspective matrix, but before divide-by-w.
Outline for Today

- Rendering intro
- Culling & clipping
- *Scan conversion*
- Midterm review
Where are we now?

Assume:

- We are rendering triangles with per-vertex colors
- We’ve transformed to image space/device coordinates
  - x,y are actual pixel coordinates
  - z is -1 to 1 pseudo-distance (but we don’t care about z for now)
- We’ve culled and clipped so that every triangle lies entirely on screen
Rendered

- **Determine:**
  - Which pixels covered by each triangle (*Rasterization*)
  - What color at that pixel
  - Which triangle is frontmost, in case of overlap
Various algorithms:

- Surface subdivision
- Spatial subdivision
- Edge equations
- *Scan Conversion*
Scan Conversion

- We have 2D triangles in device coordinates
- Draw each 2D triangle on the screen (or in the image)
  - Fill in the pixels that are covered by the triangle
  - For now, just find the right pixels
  - Later on, interpolate per-vertex data such as depth, color, normal, or other info

- Operate scanline by scanline (row-by-row) in the image
  - On each scanline, find the region that the triangle occupies
  - Fill those pixels.

- Lots of variations of scan conversion algorithms.
  - We’ll look at the fundamental idea
Device coordinates

- Viewport with 800 x 600 pixels $\Leftrightarrow$ device coordinates range from 0.0 to 800.0 in x and 0.0 to 600.0 in y
- The centers of pixels are on the 1/2’s:
  - The center of the lower left pixel is 0.5, 0.5
  - The center of the upper right pixel is 799.5, 599.5
  - The first row of pixels is at $y=0.5$
  - The first column of pixels is at $x=0.5$

![Diagram showing device coordinates]

4.0, 3.0

0.0, 0.0

2.5, 0.5
Rasterization Rules

Rules:
- Fill a pixel if the center of the pixel lies within the triangle
- If a pixel center exactly touches the edge or vertex of the triangle:
  - Fill the pixel only if the triangle extends to the right
    (extends mathematically even an infinitesimal amount less than a pixel)

Need precise rules like this to ensure:
- that pixels are rendered the same way regardless of order triangles are rendered
- that if two triangles share an edge, no pixels are left out along the border
- that if two triangles share an edge, no pixels are drawn twice on the border

Important: vertex coordinates in device space are floating point values
- should *not* be rounded to integers
Triangle Rasterization
Triangle Scan Conversion

- Start at the top of the triangle
- Work our way down one scanline at a time
- For each scanline, we will fill from left to right

- This is the most common way people think of it, but
  - Different renderers may do it in which ever order they want
  - In a hardware renderer, the order might be arranged in such a way as to optimize memory access performance
Triangle Scan Conversion

- Input: three 2D vertices
  - possibly with depth, color, normal, data per vertex
- Sort the vertices into top-down order, label them:
  - \( v_0 \) is now the top (highest y value),
  - \( v_1 \) in the middle,
  - \( v_2 \) at the bottom
- This may break our counterclockwise ordering
  - don’t care: we’re past backface culling
  - don’t care: doesn’t affect per-vertex normal data
- Steps:
  - Fill the top half of the triangle: between \( v_0 \) and \( v_1 \)
  - Fill the bottom half of the triangle: between \( v_1 \) and \( v_2 \)
- Notes:
  - if \( v_0 \) and \( v_1 \) have same y value, no top half
  - if \( v_1 \) and \( v_2 \) have same y value, no bottom half
  - \( v_0 \) and \( v_1 \) and \( v_2 \) have same y value? degenerate, already culled
Triangle Rasterization
Slope Computation

- Compute the slope of each edge:
  - How much the x-value changes with each full pixel step in y

\[
\frac{dx_L}{dy} = \frac{v_{0x} - v_{1x}}{v_{0y} - v_{1y}}
\]

\[
\frac{dx_R}{dy} = \frac{v_{0x} - v_{2x}}{v_{0y} - v_{2y}}
\]
Find The First Scanline

- To start filling the top half of the triangle (from \( \mathbf{v}_0 \) to \( \mathbf{v}_1 \)):
  - determine the first scanline to start on, i.e. the \( \frac{1}{2} \)-pixel \( y \) coordinate
  - determine the \( x \) values where edges \( \mathbf{v}_0 \mathbf{v}_1 \) and \( \mathbf{v}_0 \mathbf{v}_2 \) intersect that scanline
    - We can use the computed slopes to do this quickly

\[
\begin{align*}
y &= \left[ v_{0y} + \frac{1}{2} \right] - \frac{1}{2} & \text{scanline at highest } \frac{1}{2} \text{-pixel } y \text{ value } \leq v_{0y} \\
\Delta y &= v_{0y} - y & \text{distance in } y \text{ from } \mathbf{v}_0 \text{ to scanline} \\
x_L &= v_{0x} - \Delta y \frac{dx_L}{dy} & \text{ } x \text{ value at intersection of left edge with scanline} \\
x_R &= v_{0x} - \Delta y \frac{dx_R}{dy} & \text{ } x \text{ value at intersection of right edge with scanline}
\end{align*}
\]
Filling The Span

- Now fill pixels whose centers lie between $x_L$ and $x_R$

$$x_{first} = \left\lfloor x_L + \frac{1}{2} \right\rfloor - \frac{1}{2}$$

(also, $\Delta x = x_L - x_{first}$ for use later)

$$x_{last} = \left\lfloor x_R + \frac{1}{2} \right\rfloor - \frac{1}{2}$$

for $x = x_{first}$ to $x_{last}$:

$$\text{fill}(x, y)$$
Looping in Y

- Loop through all scanlines from $v_0$ down to $v_1$
  - At each scanline, update $x_0$ and $x_1$ incrementally with the slopes

\[
\begin{align*}
y &\leftarrow y - 1 \\
x_L &\leftarrow x_L - \frac{dx_L}{dy} \\
x_R &\leftarrow x_R - \frac{dx_R}{dy}
\end{align*}
\]
Looping in Y

- We loop from $v_0$ down to $v_1$
- Then recompute the slope of the $v_1$ edge
- Filling bottom half from $v_1$ down to $v_2$ proceeds same as the top half
Scan Conversion

- There is some cost in the set-up:
  - calculate slopes, initial y and x values
  - several divisions, etc.

- The cost per scanline and per pixel is very low
  - take advantage of incremental computation

- In hardware, scan conversion is usually done with fixed point math
  - much of the work can be done with low precision fixed point
  - 16 bits is usually fine for most operations
Interpolating color

- For solid-color triangles, fill each pixel with the color
- With per-vertex colors, want to interpolate color smoothly across the triangle
  - known *Gouraud Interpolation* or *Gouraud Shading*
Bilinear Color interpolation

- Given RGB values at each vertex: \( v_{0r}, v_{0g}, v_{0b} \) etc.
- Same structure, but with interpolation along each scanline as well as from one scanline to the next:

  Compute "slope" of color changing on each edge:
  \[
  \frac{dr_L}{dy} = \frac{v_{0r} - v_{1r}}{v_{0y} - v_{1y}} \quad \frac{dg_L}{dy} = \frac{v_{0g} - v_{1g}}{v_{0y} - v_{1y}} \quad \frac{db_L}{dy} = \frac{v_{0b} - v_{1b}}{v_{0y} - v_{1y}} \quad \frac{dr_R}{dy} = \cdots \quad \frac{dg_R}{dy} = \cdots \quad \frac{db_R}{dy} = \cdots
  \]

  Compute starting and ending color on first scanline:
  \[
  r_L = v_{0r} - \Delta y \frac{dr_L}{dy} \quad g_L = v_{0g} - \Delta y \frac{dg_L}{dy} \quad b_L = v_{0b} - \Delta y \frac{db_L}{dy} \quad r_R = \cdots \quad g_R = \cdots \quad b_R = \cdots
  \]

  For each scanline:

  Compute "slope" of color changing on the scanline:
  \[
  \frac{dr}{dx} = \frac{r_R - r_L}{x_R - x_L} \quad \frac{dg}{dx} = \frac{g_R - g_L}{x_R - x_L} \quad \frac{db}{dx} = \frac{b_R - b_L}{x_R - x_L}
  \]

  Compute color of first pixel on the scanline:
  \[
  r = r_L + \Delta x \frac{dr}{dx} \quad g = g_L + \Delta x \frac{dg}{dx} \quad b = b_L + \Delta x \frac{db}{dx}
  \]

  For \( x = x_{\text{first}} \) to \( x_{\text{last}} \):

  \[
  \text{fill}(x, y, r, g, b)
  \]

  \[
  r \leftarrow r + \frac{dr}{dx} \quad g \leftarrow g + \frac{dg}{dx} \quad b \leftarrow b + \frac{db}{dx}
  \]

  Update edge colors for next scanline:
  \[
  r_L \leftarrow r_L - \frac{dr_L}{dy} \quad g_L \leftarrow g_L - \frac{dg_L}{dy} \quad b_L \leftarrow b_L - \frac{db_L}{dy} \quad r_R \leftarrow \cdots \quad g_R \leftarrow \cdots \quad b_R \leftarrow \cdots
  \]
Interpolating other data

- For lighting calculations, can interpolate per-vertex normals to each pixel (next week)
- For texture mapping, can interpolate per-vertex texture coordinates to each pixel (in a few weeks)
- Use same structure as for color interpolation
Hidden Surface Removal

- Don’t draw pixels of triangles that are blocked by others
- Intuitive approach: pre-sort triangles, and draw them back-to-front
  - Closer triangles overwrite farther triangles
  - Known as *Painter’s Algorithm*
- Problem: slow to sort all the triangles
- Problem: triangles don’t always have a well-defined order
Z-Buffer, AKA Depth Buffer

- At each pixel, store a depth (z) value along with the RGB color
- Before starting to render a frame, set all z values to the furthest possible value
- When rasterizing, compute the z value at each pixel
  - Compare computed z value to the value stored in the buffer
  - If computed z value is farther than stored value, leave pixel as is
  - If computed z value is nearer than stored value, fill pixel and store new z value

- How to compute z value at each pixel...?
  - We chose projection to preserve straight lines: also preserves flatness of triangles
  - So we can linearly interpolate z from per-vertex z values.
  - Like color interpolation, this is bi-linear
  - $dz/dy$ and $dz/dx$ are constant on the triangle, can compute them once
Z-Buffer

- Great, simple, fast technique
- Used in all GPU renderers
- First used by Ed Catmull in his 1974 PhD thesis
  - At the time it was an extreme solution: “huge” memory cost
  - (Since then, Ed Catmull founded Pixar…)

- Doesn’t directly handle fancier features such as transparency, shadowing, smoke
  - Fancier features generally require computing effects that depend on relationships between objects.
  - Hard to do that when processing each triangle independently.
Z-buffer resolution

- Generally have fixed resolution in z, e.g. 32 bits
- Perspective mapping causes near objects to have more depth resolution than far things.
- Because of limited resolution, can have “z-fighting”:
  - If two triangles are co-planar to within the z-resolution, it’s arbitrary which one will be chosen at each pixel
  - If they have different colors, the color will flicker or have strange patterns
- To increase effective resolution:
  - Remember, projection transforms map everything between near & far planes to min,max in depth
  - The farther apart near & far planes are, the less resolution available to distinguish things between them
  - Put near & far planes as close together as you can: minimize the ratio of far/near distance
  - near clip distance of 0.1 and far clip distance of 1000 (i.e., ratio of 10,000) generally works OK for 32 bits.
Outline for Today

- Rendering intro
- Culling & clipping
- Scan conversion
- *Midterm review*
Midterm

- Covers all material through project 4:
  - Geometry & Homogeneous Coordinates
  - Modeling & Scene Graphs
  - Triangle Sets & Tessellation
  - Camera transforms, perspective
  - Culling

- Material from lectures & notes
- Mostly material you should be familiar with from doing projects
- No OpenGL.
- No essay questions.
- Closed book.
Homogeneous point transform

- Transform a point:

\[
\begin{bmatrix}
    p'_x \\
p'_y \\
p'_z \\
1
\end{bmatrix} =
\begin{bmatrix}
    m_{xx} & m_{xy} & m_{xz} & d_x \\
m_{yx} & m_{yy} & m_{yz} & d_y \\
m_{zx} & m_{zy} & m_{zz} & d_z \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
p_x \\
p_y \\
p_z \\
1
\end{bmatrix} =
\begin{bmatrix}
m_{xx}p_x + m_{xy}p_y + m_{xz}p_z + d_x \\
m_{yx}p_x + m_{yy}p_y + m_{yz}p_z + d_y \\
m_{zx}p_x + m_{zy}p_y + m_{zz}p_z + d_z \\
0 + 0 + 0 + 1
\end{bmatrix}
\]

- Top three rows are the affine transform!
- Bottom row stays 1
Homogeneous vector transform

- Transform a vector:

\[
\begin{bmatrix}
  v'_x \\
  v'_y \\
  v'_z \\
  0
\end{bmatrix} =
\begin{bmatrix}
  m_{xx} & m_{xy} & m_{xz} & d_x \\
  m_{yx} & m_{yy} & m_{yz} & d_y \\
  m_{zx} & m_{zy} & m_{zz} & d_z \\
  0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
  v_x \\
  v_y \\
  v_z \\
  0
\end{bmatrix}
= \begin{bmatrix}
  m_{xx}v_x + m_{xy}v_y + m_{xz}v_z + 0 \\
  m_{yx}v_x + m_{yy}v_y + m_{yz}v_z + 0 \\
  m_{zx}v_x + m_{zy}v_y + m_{zz}v_z + 0 \\
  0 + 0 + 0 + 0
\end{bmatrix}
\]

- Top three rows are the linear transform
  - Displacement \( d \) is properly ignored
- Bottom row stays 0
Homogeneous arithmetic

- **Legal operations always end in 0 or 1!**

<table>
<thead>
<tr>
<th>Operation</th>
<th>Example</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector+vector</td>
<td>[\begin{bmatrix} 0 \ 0 \end{bmatrix} ] + [\begin{bmatrix} 0 \ 0 \end{bmatrix} ]</td>
<td>[\begin{bmatrix} 0 \ 0 \end{bmatrix} ]</td>
</tr>
<tr>
<td>Vector-vector</td>
<td>[\begin{bmatrix} 0 \ 0 \end{bmatrix} ] - [\begin{bmatrix} 0 \ 0 \end{bmatrix} ]</td>
<td>[\begin{bmatrix} 0 \ 0 \end{bmatrix} ]</td>
</tr>
<tr>
<td>Scalar*vector</td>
<td>[s \begin{bmatrix} 0 \end{bmatrix} ]</td>
<td>[\begin{bmatrix} 0 \end{bmatrix} ]</td>
</tr>
<tr>
<td>Point+vector</td>
<td>[\begin{bmatrix} 1 \ 1 \end{bmatrix} ] + [\begin{bmatrix} 0 \ 0 \end{bmatrix} ]</td>
<td>[\begin{bmatrix} 1 \ 0 \end{bmatrix} ]</td>
</tr>
<tr>
<td>Point-point</td>
<td>[\begin{bmatrix} 1 \ 1 \end{bmatrix} ] - [\begin{bmatrix} 1 \ 1 \end{bmatrix} ]</td>
<td>[\begin{bmatrix} 0 \ 0 \end{bmatrix} ]</td>
</tr>
<tr>
<td>Point+point</td>
<td>[\begin{bmatrix} 1 \ 1 \end{bmatrix} ] + [\begin{bmatrix} 1 \ 1 \end{bmatrix} ]</td>
<td>[\begin{bmatrix} 2 \ 2 \end{bmatrix} ]</td>
</tr>
<tr>
<td>Scalar*point</td>
<td>[s \begin{bmatrix} \vdots \ 1 \end{bmatrix} ]</td>
<td>[\begin{bmatrix} \vdots \ s \end{bmatrix} ]</td>
</tr>
</tbody>
</table>

\{ weighted average \ of points: \[\frac{1}{3} \begin{bmatrix} \vdots \\ 1 \end{bmatrix} \] + \[\frac{2}{3} \begin{bmatrix} \vdots \\ 1 \end{bmatrix} \] | \[\begin{bmatrix} \vdots \\ 1 \end{bmatrix} \] \}
Homogeneous Transforms

- Rotation, Scale, and Translation of points and vectors unified in a single matrix transformation:

\[ p' = M \cdot p \]

- Matrix has the form:
  - Last row always 0,0,0,1

\[
\begin{bmatrix}
  m_{xx} & m_{xy} & m_{xz} & d_x \\
  m_{yx} & m_{yy} & m_{yz} & d_y \\
  m_{zx} & m_{zy} & m_{zz} & d_z \\
  0 & 0 & 0 & 1
\end{bmatrix}
\]

- Transforms compose by matrix multiplication.
Order matters!

- Matrix multiplication does NOT commute:
  \[ M \cdot N \neq N \cdot M \]
  - (unless one or the other is a uniform scale)
  - Try this:
    rotate 90 degrees about x then 90 degrees about z, versus
    rotate 90 degrees about z then 90 degrees about x.

- Matrix composition works *right-to-left.*
  - Compose:
    \[ M = A \cdot B \cdot C \]
    Then apply it to a vector:
    \[ \mathbf{v}' = M \cdot \mathbf{v} \]
    \[ \mathbf{v}' = (A \cdot B \cdot C) \cdot \mathbf{v} \]
    \[ \mathbf{v}' = A \cdot (B \cdot (C \cdot \mathbf{v})) \]
    It first applies \( C \) to \( \mathbf{v} \), then applies \( B \) to the result, then applies \( A \) to the result of that.
Dot Product

\[ \mathbf{a} \cdot \mathbf{b} = \sum a_i b_i \]
\[ \mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z \]
\[ \mathbf{a} \cdot \mathbf{b} = \begin{bmatrix} a_x & a_y & a_z \end{bmatrix} \begin{bmatrix} b_x \\ b_y \\ b_z \end{bmatrix} \]
\[ \mathbf{a} \cdot \mathbf{b} = \mathbf{a}^T \mathbf{b} \]
\[ \mathbf{a} \cdot \mathbf{b} = |a| |b| \cos \theta \]
Example: Angle Between Vectors

- How do you find the angle \( \theta \) between vectors \( \mathbf{a} \) and \( \mathbf{b} \)?
Example: Angle Between Vectors

\[ \mathbf{a} \cdot \mathbf{b} = |\mathbf{a}| |\mathbf{b}| \cos \theta \]

\[ \cos \theta = \left( \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| |\mathbf{b}|} \right) \]

\[ \theta = \cos^{-1} \left( \frac{\mathbf{a} \cdot \mathbf{b}}{|\mathbf{a}| |\mathbf{b}|} \right) \]
Dot Product Properties

- The dot product is a scalar value that tells us something about the relationship between two vectors.
  - If $\mathbf{a} \cdot \mathbf{b} > 0$ then $\theta < 90^\circ$
    - Vectors point in the same general direction
  - If $\mathbf{a} \cdot \mathbf{b} < 0$ then $\theta > 90^\circ$
    - Vectors point in opposite direction
  - If $\mathbf{a} \cdot \mathbf{b} = 0$ then $\theta = 90^\circ$
    - Vectors are perpendicular
    - (or one or both of the vectors is degenerate $(0,0,0)$)
Example: Normal of a Triangle

- The concept of a “normal” will be essential to lighting
- Find the unit length normal of the triangle defined by 3D points a, b, c
Example: Normal of a Triangle

- Find the unit length normal of the triangle defined by 3D points $a$, $b$, and $c$

$$\mathbf{n}^* = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})$$

$$\mathbf{n} = \frac{\mathbf{n}^*}{|\mathbf{n}^*|}$$
Example: Alignment to Target

- An object is at position \( \mathbf{p} \) with a \textit{unit length} heading of \( \mathbf{h} \). We want to rotate it so that the heading is facing some target at position \( \mathbf{t} \). Find a unit axis \( \mathbf{a} \) and an angle \( \theta \) to rotate around.
Example: Alignment to Target

\[ \vec{d} = \frac{\vec{t} - \vec{p}}{|\vec{t} - \vec{p}|} \]

\[ \vec{a} = \frac{\vec{h} \times \vec{d}}{|\vec{h} \times \vec{d}|} \]

\[ \theta = \cos^{-1}(\vec{h} \cdot \vec{d}) \]
Example: Distance to Plane

A plane is described by a point $p$ on the plane and a unit normal $\mathbf{n}$. Find the (perpendicular) distance from point $x$ to the plane.
Example: Distance to Plane

- The distance is the length of the projection of \( \overrightarrow{x - p} \) onto \( \overrightarrow{n} \):

\[
dist = (\overrightarrow{x - p}) \cdot \overrightarrow{n}
\]
3-D Graphics Rendering Pipeline

- Primitives
  - Modeling Transformation
  - Viewing Transformation
  - Culling
  - Lighting & Shading
  - Clipping
  - Projection
  - Scan conversion, Hiding

- Spaces:
  - Object space
  - World space
  - Camera space
  - Normalized view space
  - Image space, Device coordinates
Object and World Coordinates
Placing object coordinates in the world

- Place the coordinate frame for the object in the world
  - Don’t know or care about the shape of the object
  - World matrix columns = object’s frame in world coordinates
Transformation as coordinate frame

- Build matrix from vectors \( \mathbf{a}, \mathbf{b}, \mathbf{c}, \) point \( \mathbf{d} \)

\[
M = \begin{bmatrix}
    a_x & b_x & c_x & d_x \\
    a_y & b_y & c_y & d_y \\
    a_z & b_z & c_z & d_z \\
    0   & 0   & 0   & 1
\end{bmatrix}
\]

- Notice effect on coordinate frame:

\[
\mathbf{a} = M \begin{bmatrix} 1 \\ 0 \\ 0 \\ 0 \end{bmatrix} = M \mathbf{x} \quad \mathbf{b} = M \begin{bmatrix} 0 \\ 1 \\ 0 \\ 0 \end{bmatrix} = M \mathbf{y} \quad \mathbf{c} = M \begin{bmatrix} 0 \\ 0 \\ 1 \\ 0 \end{bmatrix} = M \mathbf{z} \quad \mathbf{d} = M \begin{bmatrix} 0 \\ 0 \\ 0 \\ 1 \end{bmatrix} = M \mathbf{o}
\]

- Any transform \( M \) describes change of frame:

\[
\langle \mathbf{d}, \mathbf{a}, \mathbf{b}, \mathbf{c} \rangle = M \langle \mathbf{o}, \mathbf{x}, \mathbf{y}, \mathbf{z} \rangle
\]

- If \( \mathbf{a}, \mathbf{b}, \mathbf{c} \) are right-handed orthonormal, transformation is *rigid*
  - Pure rotation, pure translation, or mix of rotation and translation
  - No scale
Relative Transformations

- Put the objects on the tables
  - Each table has a simple coordinate system
  - E.g. Book1 at (3.75,1,0) on Table1’s top
  - E.g. Keyboard at (3,.5,0) on Table2’s top
  - Don’t care where the tables are in order to do this part

- Put the tables in the room
  - Books etc. should end up in the right place

- Current Transformation Matrix (CTM)
  - Matrix stack: PushCTM(), PopCTM()
More detailed scene graph
Composing with inverses, pictorially

To go from one space to another, compose along arrows
- Backwards along arrow: use inverse transform

\[
\text{Lamp in world coords} = M_{\text{table1}} M_{\text{top1}} M_{\text{lamp}}
\]
\[
\text{Plant in Tabletop1 coords} = M_{\text{top1}}^{-1} M_{\text{table1}}^{-1} M_{\text{table2}} M_{\text{top2}} M_{\text{plant}}
\]
Camera Look-At setup

- **look-from** = eye point \( e \)
- **look-at** = target point \( t \)
- **up vector** \( \mathbf{u} \)
- **view vector**

World coordinates

Camera Space

Camera Matrix \( \mathbf{C} \)
“Look-at” Matrix calculation, summary

Given: eye point \( e \), target point \( t \), and up vector \( \vec{u} \)

Construct: columns of camera matrix \( C \)

\[
\begin{align*}
d &= e \\
\vec{c} &= \frac{e - t}{|e - t|} \\
\vec{a} &= \frac{\vec{u} \times \vec{c}}{|\vec{u} \times \vec{c}|} \\
\vec{b} &= \vec{c} \times \vec{a}
\end{align*}
\]

- Note: The up vector may not end up parallel to the camera y axis
  - The projection of the up vector onto the film plane lines up with camera y

- If the up vector is parallel to the view vector, the result is undefined!
  - The up vector will project to nothing in the image
  - No matter how you spin the camera, there’s no thing to line up with the camera y
  - It’s a user error!
Perspective Projection

- Assume that we have “film” at distance $d$ from the eye
- Distant tall object projects to same height as near small object
- By similar triangles, we have:
  \[
  \frac{y'}{d} = \frac{y_1}{z_1} = \frac{y_2}{z_2}
  \]
  
  Giving the transformation relations:
  \[
  y' = d \frac{y}{z}, \quad x' = d \frac{x}{z}
  \]

- Notice: divide by $z$
  - not a linear operation!
Homogeneous Perspective Projection

- The homogeneous perspective projection matrix. Notice the last row!

\[
P = \begin{bmatrix}
d_1 & 0 & 0 & 0 \\
0 & d_2 & 0 & 0 \\
0 & 0 & A & B \\
0 & 0 & 1 & 0 \\
\end{bmatrix}
\]

- Multiply it by a homogeneous point

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
w'
\end{bmatrix} = P \begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix} = \begin{bmatrix}
d_1 x + 0 + 0 + 0 \\
0 + d_2 y + 0 + 0 \\
0 + 0 + A z + B \\
0 + 0 + z + 0
\end{bmatrix} = \begin{bmatrix}
d_1 x \\
d_2 y \\
A z + B \\
z
\end{bmatrix}
\]

- Notice that the result doesn’t have \( w=1 \). So divide by \( w \):

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
w'
\end{bmatrix} \Rightarrow \begin{bmatrix}
x' / w' \\
y' / w' \\
z' / w' \\
w' / w'
\end{bmatrix} = \begin{bmatrix}
d_1 x / z \\
d_2 x / z \\
A + B / z \\
1
\end{bmatrix}
\]
Viewport Transformation
The complete transform

- Composing the modeling matrix \( M \), the camera matrix \( C \), the projection matrix \( P \), and the viewport matrix \( D \), we have, for some point \( p \):

\[
pixel = (D \ P \ C^{-1} \ M) \ p
\]
Cube - indexed triangles

- **8 vertices:**
  - P0: (1, -1, 1)
  - P1: (1, -1, -1)
  - P2: (1, 1, -1)
  - P3: (1, 1, 1)
  - P4: (-1, -1, 1)
  - P5: (-1, -1, -1)
  - P6: (-1, 1, -1)
  - P7: (-1, 1, 1)

- **12 triangles:**
  - P4 P0 P3
  - P4 P3 P7
  - P0 P1 P2
  - P0 P2 P3
  - P1 P5 P6
  - P1 P6 P2
  - P5 P4 P7
  - P5 P7 P6
  - P7 P3 P2
  - P7 P2 P6
  - P0 P5 P1
  - P0 P4 P5

- **8 vertices** * 3 floats = 24 floats
- **12 triangles** * 3 points = 36 integers
Bounding Volume

- Simple shape that completely encloses an object
- Generally a box or sphere
- We’ll use spheres:
  - Easiest to work with
  - Though hard to get tight fits
Culling to frustum

- Defined by 6 planes
- Each plane divides space:
  - “outside”
  - “inside”
- Check each object against each plane:
  - entirely outside
  - entirely inside
  - intersecting
- If it’s completely “outside” any plane it’s outside the frustum
- If it’s completely “inside” all planes it’s inside the frustum
- Else it’s partly inside and partly out
Good luck.

- Next class: midterm
- Project 4 not due til Friday
- Project 5 will be due next Wednesday (but will be shorter)
- Next week: lighting