Announcements

- Project 1 due Friday October 9
- Project 2 due Friday October 16
  - Homework introductions:
    - Daniel: Thu Oct 8, 11am, lab 250
    - Jason: Mon Oct 12, 2pm, lab 250
- TA hours:
  - Jason: Mon, 2-4pm (lab 250); Wed, 2-5pm (lab 270)
  - Daniel: Tue/Thu 11am-1:45pm (lab 250)
- Visual C++ should be installed on all Windows machines in the lab (if not try reboot)
- Gradesource IDs have been sent to everybody’s email addresses
Today

- Viewport transformation
- Barycentric coordinates
- Culling, clipping
- Rasterization
- Visibility
View volumes

• Define 3D volume seen by camera

Perspective view volume

Camera coordinates

Orthographic view volume

Camera coordinates

World coordinates

World coordinates
• Projection matrix is such that
  - User defined view volume is transformed into canonical view volume: cube [-1,1]x[-1,1]x[-1,1]
  - Multiplying vertices of view volume by projection matrix and performing homogeneous divide yields canonical view volume

• Perspective and orthographic projection are treated exactly the same way

• Canonical view volume is last stage in which coordinates are in 3D; next step is projection to 2D frame buffer
Projection matrix

Camera coordinates

Projection matrix

Canonical view volume

Clipping
Perspective projection matrix

- General view frustum with 6 parameters

\[
P_{\text{persp}}(left, right, top, bottom, near, far) = \\
\begin{bmatrix}
\frac{2\text{near}}{right-left} & 0 & \frac{right+left}{right-left} & 0 \\
0 & \frac{2\text{near}}{top-bottom} & \frac{top+bottom}{top-bottom} & 0 \\
0 & 0 & \frac{-(far+near)}{far-near} & -2\text{far}\cdot\text{near} \\
0 & 0 & \frac{1}{far-near} & 0 \\
\end{bmatrix}
\]
Perspective projection matrix

- Symmetrical view frustum with 4 parameters

\[
\mathbf{P}_{\text{persp}}(\text{FOV}, \text{aspect}, \text{near}, \text{far}) =
\begin{bmatrix}
0 & 0 & 0 & 0 \\
1 & 0 & 0 & 0 \\
0 & \text{tan}(\text{FOV}/2) & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
0 & 0 & \text{near + far} & 2 \cdot \text{near} \cdot \text{far} \\
0 & 0 & \text{near} \cdot \text{far} & \text{near} \cdot \text{far} \\
0 & 0 & -1 & 0
\end{bmatrix}
\]
Orthographic projection matrix

Camera coordinates

$$P_{\text{ortho}}(\text{right, left, top, bottom, near, far}) = \begin{bmatrix}
\frac{2}{\text{right} - \text{left}} & 0 & 0 & -\frac{\text{right} + \text{left}}{\text{right} - \text{left}} \\
0 & \frac{2}{\text{top} - \text{bottom}} & 0 & -\frac{\text{top} + \text{bottom}}{\text{top} - \text{bottom}} \\
0 & 0 & \frac{2}{\text{far} - \text{near}} & -\frac{\text{far} + \text{near}}{\text{far} - \text{near}} \\
0 & 0 & 0 & 1
\end{bmatrix}$$

$$P_{\text{ortho}}(\text{width, height, near, far}) = \begin{bmatrix}
\frac{2}{\text{width}} & 0 & 0 & 0 \\
0 & \frac{2}{\text{height}} & 0 & 0 \\
0 & 0 & \frac{2}{\text{far} - \text{near}} & \frac{\text{far} + \text{near}}{\text{far} - \text{near}} \\
0 & 0 & 0 & 1
\end{bmatrix}$$
Viewport transformation

- After applying projection matrix, scene points are in *normalized viewing coordinates*
  - Per definition range $[-1..1] \times [-1..1] \times [-1..1]$

- Normalized viewing coordinates can be mapped to image (=pixel=frame buffer) coordinates
  - Range depends on window (view port) size: $[x_0...x_1] \times [y_0...y_1]$

- Scale and translation required:

$$D(x_0, x_1, y_0, y_1) = \begin{bmatrix}
\frac{(x_1 - x_0)}{2} & 0 & 0 & \frac{(x_0 + x_1)}{2} \\
0 & \frac{(y_1 - y_0)}{2} & 0 & \frac{(y_0 + y_1)}{2} \\
0 & 0 & \frac{1}{2} & \frac{1}{2} \\
0 & 0 & 0 & 1
\end{bmatrix}$$
The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates

- Object-to-world matrix $M$, camera matrix $C$, projection matrix $P$, viewport matrix $D$

$$p' = \mathbf{DPC}^{-1}\mathbf{M} p$$

Object space
The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates

- Object-to-world matrix $M$, camera matrix $C$, projection matrix $P$, viewport matrix $D$

\[ p' = DPC^{-1}M p \]

Object space
World space
The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates

- Object-to-world matrix \( M \), camera matrix \( C \), projection matrix \( P \), viewport matrix \( D \)

\[
p' = DPC^{-1}M_p
\]

- Object space
- World space
- Camera space
The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates

- Object-to-world matrix $M$, camera matrix $C$, projection matrix $P$, viewport matrix $D$

$$p' = DPC^{-1}M_p$$

Object space

World space

Camera space

Canonical view volume
The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates

- Object-to-world matrix $M$, camera matrix $C$, projection matrix $P$, viewport matrix $D$

\[
p' = DPC^{-1}Mp\]

Object space
World space
Camera space
Canonical view volume
Image space
The complete transform

- Mapping a 3D point in object coordinates to pixel coordinates
- Object-to-world matrix $M$, camera matrix $C$, projection matrix $P$, viewport matrix $D$

$$p' = DPC^{-1}Mp$$

$$p' = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix}$$

Pixel coordinates: $x'/w', y'/w'$
OpenGL

- Object-to-world matrix $M$, camera matrix $C$, projection matrix $P$, viewport matrix $D$

$\text{OpenGL \ GL\_MODELVIEW \ matrix}$

$p' = DPC^{-1}Mp$

$\text{OpenGL \ GL\_PROJECTION \ matrix}$
OpenGL

• **GL_MODELVIEW, $C^{-1}M$**
  
  - Up to you to define

• **GL_PROJECTION, $P$**
  
  - Utility routines to set it by specifying view volume: glFrustum(), glPerspective(), glOrtho()
  
  - Do not use utility functions in homework project 2
  
  - You will implement a software renderer in project 3, which will not use OpenGL

• **Viewport, $D$**
  
  - Specify implicitly via glViewport()
  
  - No direct access with equivalent to GL_MODELVIEW or GL_PROJECTION
Today

- Viewport transformation
- Barycentric coordinates
- Culling, clipping
- Rasterization
- Visibility
Rendering pipeline

Scene data → Modeling and viewing transformation → Shading → Projection → Rasterization, visibility → Image

Lectures 2 and 3
Lectures 6-8
Lecture 4
Lecture 5 (today!)
Implicit 2D lines

- Given two 2D points $a, b$
- Define function $f_{ab}(p)$ such that $f_{ab}(p) = 0$ if $p$ lies on line defined by $a, b$
Implicit 2D Lines

• Point \( \mathbf{p} \) lies on the line, if \( \mathbf{p} - \mathbf{a} \) is perpendicular to the normal of the line

\[
(a_y - b_y, b_x - a_x)
\]

• Use dot product to determine on which side of the line \( \mathbf{p} \) lies. If \( f(\mathbf{p}) > 0 \), \( \mathbf{p} \) is on same side as normal, if \( f(\mathbf{p}) < 0 \) \( \mathbf{p} \) is on opposite side. If dot product is 0, \( \mathbf{p} \) lies on the line.

\[
f_{ab}(\mathbf{p}) = (a_y - b_y, b_x - a_x) \cdot (p_x - a_x, p_y - a_y)
\]
Barycentric coordinates

- Coordinates for 2D plane defined by triangle vertices \( a, b, c \)
- Any point \( p \) in the plane defined by \( a, b, c \) is
  \[
  p = a + \beta (b - a) + \gamma (c - a)
  = (1 - \beta - \gamma) a + \beta b + \gamma c
  \]
- We define \( \alpha = 1 - \beta - \gamma \)
  \[
  => p = \alpha a + \beta b + \gamma c
  \]
- \( \alpha, \beta, \gamma \) are called **barycentric** coordinates
- Works in 2D and in 3D
- If we imagine masses equal to \( \alpha, \beta, \gamma \) attached to the vertices of the triangle, the center of mass (the barycenter) is then \( p \). This is the origin of the term “barycentric” (introduced 1827 by Möbius)
Barycentric coordinates

\[ p = a + \beta(b - a) + \gamma(c - a) \]

- \( p \) is inside the triangle if \( 0 < \alpha, \beta, \gamma < 1 \)
Barycentric coordinates

- Problem: Given point $p$, find its barycentric coordinates
- Use equation for implicit lines

$$\beta(p) = \frac{f_{ac}(p)}{f_{ac}(b)}$$

$$\gamma(p) = \frac{f_{ab}(p)}{f_{ab}(c)}$$

$$\alpha = 1 - \beta - \gamma$$

$0 < \beta < 1$

- Division by zero if triangle is degenerate
Barycentric interpolation

- Interpolate values across triangles, e.g., colors

\[ c(p) = \alpha(p)c_a + \beta(p)c_b + \gamma(p)c_c \]

- Linear interpolation on triangles
Today

• Viewport transformation
• Barycentric coordinates
• Culling, clipping
• Rasterization
• Visibility
Primitives

Modeling and viewing transformation

Shading

Projection

Scan conversion, visibility

Image

Culling, clipping
- Discard geometry that should not be drawn
Culling

• Discard geometry that does not need to be drawn as early as possible

• Two types of culling:
  - Object-level frustum culling
    • Later in class
  - Triangle culling
    • View frustum culling (clipping): outside view frustum
    • Backface culling: facing “away” from the viewer
    • Degenerate culling: area=0
Backface culling

• Consider triangles as “one-sided”, i.e., only visible from the “front”

• Closed objects
  - If the “back” of the triangle is facing the camera, it is not visible
  - Gain efficiency by not drawing it (culling)
  - Roughly 50% of triangles in a scene are back facing
Backface culling

- Convention: front side means vertices are ordered counterclockwise

- OpenGL allows one- or two-sided triangles
  - One-sided triangles:
    \[ \text{glEnable(GL\_CULL\_FACE)}; \text{glCullFace(GL\_BACK)} \]
  - Two-sided triangles (no backface culling):
    \[ \text{glDisable(GL\_CULL\_FACE)} \]
Backface culling

• Compute triangle normal after projection (homogeneous division)

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

• Third component of \( \mathbf{n} \) negative: front-facing, otherwise back-facing
  - Remember: projection matrix is such that homogeneous division flips sign of third component
Degenerate culling

- Degenerate triangle has no area
  - Vertices lie in a straight line
  - Vertices at the exact same place
  - Normal $n=0$
View frustum culling, clipping

• Triangles that intersect the faces of the view volume
  - Partly on screen, partly off screen
  - Do not rasterize the parts that are off-screen

• Traditional clipping
  - Split triangles that lie partly inside/outside viewing volume before homogeneous division
  - Avoid problems with division by zero

• Modern GPU implementations avoid clipping
Today

- Viewport transformation
- Barycentric coordinates
- Culling, clipping
- Rasterization
- Visibility
Scan conversion and rasterization are synonyms.

One of the main operations performed by GPU.

Draw triangles, lines, points (squares).

Focus on triangles in this lecture.
Rasterization
Rasterization

• How many pixels can a modern graphics processor draw per second?
Rasterization

- How many pixels can a modern graphics processor draw per second?
- Rasterization is „hard-coded“, cannot be modified by the software
- NVidia Geforce 295 GTX
  - Theoretical peak 32 billion pixels per second
  - Multiple of what the fastest CPU could do
Rasterization

- Many different algorithms
- Old style
  - Rasterize edges first
Rasterization

- Many different algorithms
- Old style
  - Rasterize edges first
  - Fill the spans (scan lines, scan conversion)
Rasterization

- Many different algorithms
- Old style
  - Rasterize edges first
  - Fill the spans (scan lines, scan conversion)
  - Requires clipping
  - Not preferred for hardware implementation today
Rasterization

- GPU rasterization today based on “homogeneous rasterization”
  
  http://www.ece.unm.edu/course/ece595/docs/olano.pdf


- Does not require full clipping, does not perform homogeneous division at vertices

- Today in class
  - Simpler algorithm based on barycentric coordinates
  - Easy to implement
  - Requires clipping
Rasterization

- Given vertices in pixel coordinates

\[
p' = \begin{bmatrix} x' \\ y' \\ z' \\ w' \end{bmatrix} \quad \text{Pixel coordinates} \quad \begin{bmatrix} x'/w' \\ y'/w' \end{bmatrix}
\]

- World space
- Camera space
- Clip space
- Image space

\[
p' = DPC^{-1}Mp
\]
Rasterization

• Simple algorithm

compute bbox
clip bbox to screen limits
for all pixels \([x,y]\) in bbox
compute barycentric coordinates alpha, beta, gamma
if \(0<alpha,beta,gamma<1\) //pixel in triangle
\[image[x,y]=triangle\text{Color}\]

• Bounding box clipping trivial
So far, we compute barycentric coordinates of many useless pixels

Improvement?
Rasterization

Hierarchy

- If block of pixels is outside triangle, no need to test individual pixels
- Can have several levels, usually two-level
- Find right granularity and size of blocks for optimal performance
2D Triangle-Rectangle Intersection

• If one of the following tests returns true, the triangle intersects the rectangle:
  - Test if any of the triangle’s vertices are inside the rectangle (e.g., by comparing the x/y coordinates to the min/max x/y coordinates of the rectangle)
  - Test if one of the quad’s vertices is inside the triangle (e.g., using barycentric coordinates)
  - Intersect all edges of the triangle with all edges of the rectangle
Where is the center of a pixel?

• Depends on conventions
• With our viewport transformation:
  - 800 x 600 pixels $\Leftrightarrow$ viewport coordinates are in $[0...800] \times [0...600]$
  - Center of lower left pixel is 0.5, 0.5
  - Center of upper right pixel is 799.5, 599.5
Rasterization

Shared edges

- Each pixel needs to be rasterized exactly once
- Resulting image is independent of drawing order
- Rule: If pixel center exactly touches an edge or vertex
  - Fill pixel only if triangle extends to the right
Today

- Viewport transformation
- Barycentric coordinates
- Culling, clipping
- Rasterization
- Visibility
• At each pixel, we need to determine which triangle is visible
Painter’s algorithm

- Paint from back to front
- Every new pixel always paints over previous pixel in frame buffer
- Need to sort geometry according to depth
- May need to split triangles if they intersect

- Old style, before memory became cheap
Z-buffering

• Store z-value for each pixel

• Depth test
  - During rasterization, compare stored value to new value
  - Update pixel only if new value is smaller

```c
setpixel(int x, int y, color c, float z)
if(z<zbuffer(x,y)) then
  zbuffer(x,y) = z
  color(x,y) = c
```

• z-buffer is dedicated memory reserved for GPU (graphics memory)

• Depth test is performed by GPU
Next Lecture

• Perspectively correct interpolation
• Color spaces