Scanline Rendering
Human Eye

- Think about rendering from the viewpoint of what/how you see
- Light is emitted from light sources, absorbed and reflected off of objects, and enters the pupil of the eye
• Without the pupil light scatters everywhere and does not make a coherent image
• With a pinhole pupil, light makes a coherent image
• Similarly, cameras have an aperture
Cameras

- Cameras work very much like the eye
  - Light from the environment is bent by the lens array to make an image on the film
  - Digital cameras record this image in RGB values
  - Our scanline renderer will use a “virtual” camera
Pinhole Camera

- Simplified theoretical construct, similar to the eye/camera
- Eyes/cameras can’t have VERY small holes because that limits the amount of entering light
  - and diffracts/bends the light
Pinhole Camera

- Assumptions
  - Light leaving an object travels in straight lines
  - We only care about the lines that hit a single point (the pinhole)
  - Infinite depth of field - which means everything is in focus (no blur)
  - Upside down image is formed by the intersection of these lines with an image plane
  - More distant objects subtend smaller visual angles and appear smaller
  - Objects occlude the objects behind them
OpenGL Camera

- OpenGL uses a pinhole camera.
- However, the image plane is placed in front of the focal point, so that the image is not upside down.
- The frustum is the volume of our view (shown in blue below).
  - Notice the front and back clipping planes.
- The image plane is the clipping plane nearest to the camera.
Triangles

Stanford Bunny
69,451 triangles

David, Digital Michelangelo Project
28,184,526 vertices, 56,230,343 triangles
Why Does OpenGL use Triangles?

• It’s easy to break convex polygons into triangles
• Can focus on optimizing the hardware/software implementation of only one primitive (i.e. only triangles)
• Triangles have some nice properties
  ■ Guaranteed to be planar (unlike quadrilaterals)
  ■ Guaranteed to have a well-defined interior
  ■ There exists a well-defined method for interpolating values in the triangle interior (barycentric interpolation)
  ■ Etc…
Transformations

Act on points:

\[(x', y', z') = T(x, y, z)\]

\[P' = T(P)\]

Use in Modeling

- Can create objects in convenient coordinates
- Can have multiple instances of the same object or shape
- Kinematics of linkages/skeletons for characters/robots

Use in Viewing

- Virtual camera: perspective projection
- Mapping window coordinates to framebuffer coordinates
Matrix Representation

\[ x' = m_{xx} x + m_{xy} y \]
\[ y' = m_{yx} x + m_{yy} y \]

\[
\begin{bmatrix}
  x' \\
  y'
\end{bmatrix}
= \begin{bmatrix}
  m_{xx} & m_{xy} \\
  m_{yx} & m_{yy}
\end{bmatrix}
\begin{bmatrix}
  x \\
  y
\end{bmatrix}
\]

\[ x' = M x \]

Combine a sequence of transforms into a single transform

\[ P' = A ( B ( C ( D ( P ) ) ) ) = ( A B C D ) P = M P \]

- Matrix multiplication is associative
- Compute the composite matrix M once and then apply it to many points
Lines Map to Lines

Parametric line: \( \mathbf{x} = (1 - \alpha)\mathbf{p} + \alpha \mathbf{q} \)

Transform all the points on the line

\[
\mathbf{x}' = \mathbf{M} \mathbf{x} = (1 - \alpha)\mathbf{M} \mathbf{p} + \alpha \mathbf{M} \mathbf{q} = (1 - \alpha)\mathbf{p}' + \alpha \mathbf{q}'
\]

Get a new line connecting the transformed points
Rotation

$$x' = Mx$$

glm::angleAxis(angle, vec3(ax, ay, ax))
2D Rotation Matrix

The columns of the matrix are the new locations of the x and y axes

\[
\begin{bmatrix}
  m_{xx} \\
  m_{yx}
\end{bmatrix} =
\begin{bmatrix}
  m_{xx} & m_{xy} \\
  m_{yx} & m_{yy}
\end{bmatrix}
\begin{bmatrix}
  1 \\
  0
\end{bmatrix}
\]

\[
\begin{bmatrix}
  m_{xy} \\
  m_{yy}
\end{bmatrix} =
\begin{bmatrix}
  m_{xx} & m_{xy} \\
  m_{yx} & m_{yy}
\end{bmatrix}
\begin{bmatrix}
  0 \\
  1
\end{bmatrix}
\]

So, set the columns to the desired new locations of the x and y axes

\((- \sin \theta, \cos \theta)\)

Rotation matrix:

\[
\begin{bmatrix}
  \cos \theta & -\sin \theta \\
  \sin \theta & \cos \theta
\end{bmatrix}
\]

3D is similar...
Translation

What does a translation matrix look like?… use homogeneous coordinates

$$x' = x + t_x$$

$$y' = y + t_y$$

\[glm::translate(currentMatrix, vec3(tx, ty, tz))\]
Homogeneous Coordinates

- The homogeneous coordinates of a 3D point \((x, y, z)\) are 
  \((xw, yw, zw, w)\)

- Conversely, the position \((x, y, z)\) can be calculated from homogeneous coordinates by dividing by \(w\)
  \((xw/w, yw/w, zw/w)\)

- Setting \(w=1\), gives the homogeneous coordinates used for translations

- A 3D point \((x, y, z)\) becomes \((x, y, z, 1)\)
2D Rotation Matrix

- A 2D point becomes \((x, y, 1)\), so we need a 3x3 matrix in order to represent a 2D rotation
- This is done by adding an extra row and column of \((0, 0, 1)\)

\[
\begin{bmatrix}
  \cos \theta & \sin \theta & 0 \\
  -\sin \theta & \cos \theta & 0 \\
  0 & 0 & 1
\end{bmatrix}
\]

- Similarly, 3D rotation matrices are expressed via a 4x4 matrix
2D Translation Matrix

\[
\begin{bmatrix}
{x'} \\
{y'} \\
1
\end{bmatrix}
= \begin{bmatrix}
1 & 0 & {t_x} \\
0 & 1 & {t_y} \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
1
\end{bmatrix}
\]

3D is similar...
Vectors

• In homogeneous coordinates, vectors are represented using $w=0$ instead of $w=1$

• Thus translation does not affect vectors, as desired

\[
\begin{bmatrix}
1 & 0 & t_x \\
0 & 1 & t_y \\
0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
0
\end{bmatrix}
= 
\begin{bmatrix}
x \\
y \\
0
\end{bmatrix}
\]

• The vectors still rotate properly though…
Rotate 45 degrees about the Center (1,1)

$T(1,1) \ R(45) \ T(-1,-1)$
Composing Transforms

- Order matters
- Matrices don’t commute
- The rightmost transform is applied to the points first
Hierarchical Transforms

- $M_1$ transforms the teapot from its model space to the tables model space
- $M_2$ transforms the table from its model space to world space
- $M_2M_1$ transforms the teapot from its model space to world space
Graphics Coordinate Frames

Object Space (model space)
- Raw values as provided by `glBufferData` to a vertex buffer object (e.g. teapot centered at origin)

World Space
- Object at final location in the environment (e.g. teapot on top of a table on the ground off to the left side of the screen)

Screen Space
- Object splatted into pixels onto a two dimensional screen position

```cpp
uniform mat4 modelMatrix, viewMatrix, projectionMatrix;
```
- Transformation performed in the vertex shader.
- `viewMatrix*modelMatrix`: object to world transform
- `projectionMatrix`: world to screen transform
- `CTM = projectionMatrix*viewMatrix*modelMatrix`
World to Screen Transform

\[
\begin{align*}
\frac{x}{z} &= \frac{x'}{h} & x' &= h \frac{x}{z} \\
\frac{y}{z} &= \frac{y'}{h} & y' &= h \frac{y}{z}
\end{align*}
\]

glm::perspective creates a matrix to perform perspective projection to transform objects from world space to screen space (film plane)
Perspective Projection

\[ x' = h \frac{x}{z} \quad y' = h \frac{y}{z} \]

- Using homogeneous coordinates and setting \( w' = z \) allows us to have a linear model (4X4 matrix) for a nonlinear function \( 1/z \)

\[
\begin{bmatrix}
x'w' \\
y'w' \\
z'w' \\
w'
\end{bmatrix}
= \begin{bmatrix}
h & 0 & 0 & 0 \\
0 & h & 0 & 0 \\
0 & 0 & a & b \\
0 & 0 & 1 & 0
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
1
\end{bmatrix}
\]

- What about the third row? What are \( a \) and \( b \)?
Perspective Projection

\[ z'z = az + b \]

- If we map \( z=n \) to \( z'=n \) and map \( z=f \) to \( z'=f \), we get
  \[ n^2 = an + b \quad f^2 = af + b \]
- Two equations, two unknowns. The solution is
  \[ a = n + f \quad b = -fn \]

- \( n \) and \( f \) are still the near and far clipping planes
- Transforms the 3D frustum in world space into an orthographic volume in screen space
- Can use \( z' \) for occlusion
Rasterization

- Rasterize 2D triangles that have been transformed into screen space
- Color pixels inside the triangle with the RGB-color of the triangle
Normal to a Line

\[ p_0 = (x_0, y_0) \]

\[ t = p_1 - p_0 = (x_1 - x_0, y_1 - y_0) \]

\[ n = (y_1 - y_0, x_0 - x_1) \]

normalize, if desired
Implicit Equation for a Line

\[ t \cdot n = 0 \]

For all points \( p \) on the line

\[ (p - p_0) \cdot n = 0 \] for all points \( p \) on the line
Implicit Equation for a Ray

- Outward normal points to the right of the ray
- “Interior” points are to the left of the ray, and have negative $(p - p_o) \cdot n$ values
makeline( vert& v0, vert& v1, line& l )
{
    l.a = v1.y - v0.y;
    l.b = v0.x - v1.x;
    l.c = -(l.a * v0.x + l.b * v0.y);
}

Decide if a point(x,y) is inside a line:
e = l.a * x + l.b * y + l.c
\{e \leq 0 \text{ inside(on the left)}
\{e > 0 \text{ outside(on the right)}
Point Inside Triangle Test

- Inside a triangle, if inside (to the left of) all 3 rays
- Back facing triangles are not rendered, since no points are to the left of all three rays

Counter Clockwise
(Facing Camera)

Clockwise
(Facing Away from Camera)
Point (pixel center) Inside Triangle Test

rasterize( vert v[3] )
{
    line l0, l1, l2;
    makeline(v[0], v[1], l2);
    makeline(v[1], v[2], l0);
    makeline(v[2], v[0], l1);
    for( y=0; y<YRES; y++ ) for( x=0; x<XRES; x++ ) {
        e0 = l0.a * x + l0.b * y + l0.c;
        e1 = l1.a * x + l1.b * y + l1.c;
        e2 = l2.a * x + l2.b * y + l2.c;
        if( e0<=0 && e1<=0 && e2<=0 )
            fragment(x,y);
    }
}
Indeterminate Cases

- Edges that exactly touch pixels ($e = 0$)
- Pixels on a shared edge between two triangles are flagged by both triangles
  - Wasted effort drawing duplicate fragments
  - Problems for transparent objects
- Not including these pixels (i.e. using $e < 0$) causes gaps:
Indeterminate Cases

- Don’t draw edges to the right and above each triangle/polygon (omit bold face segments and hollow points in these figures)

```
int shadow( line l ) {
    return (l.a>0) || (l.a == 0 && l.b > 0);
} // normal points right || vertical normal pointing up
```

```
int inside( value e, line l ) {
    return (e == 0) ? !shadow(l) : (e < 0);
} // if e=0, don’t shade shadow line
```
Point (pixel center) Inside Triangle Test

rasterize( vert v[3] )
{
    line l0, l1, l2;
    makeline(v[0], v[1], l2);
    makeline(v[1], v[2], l0);
    makeline(v[2], v[0], l1);
    for( y=0; y<YRES; y++ ) for( x=0; x<XRES; x++ ) {
        e0 = l0.a * x + l0.b * y + l0.c;
        e1 = l1.a * x + l1.b * y + l1.c;
        e2 = l2.a * x + l2.b * y + l2.c;
        if( inside(e0, l0)&inside(e1, l1)&inside(e2, l2) )
            fragment(x,y);
    }
}

Point (pixel center) Inside Triangle Test

v0
v1
v2
l0
l1
l2
Bounding Box (rectangle)

- Inefficient to check every pixel on the screen
- Calculate a bounding box around the triangle, and only check pixels inside the box
- Round coordinates upward (ceil) to the nearest integer

```
bound3( vert v[3], bbox& b )
{
    b.xmin = ceil(min(v[0].x, v[1].x, v[2].x));
    b.xmax = ceil(max(v[0].x, v[1].x, v[2].x));
    b.ymin = ceil(min(v[0].y, v[1].y, v[2].y));
    b.ymax = ceil(max(v[0].y, v[1].y, v[2].y));
}
```
rasterize( vert v[3] )
{
    bbox b; bound3(v, b);
    line l0, l1, l2;
    makeline(v[0], v[1], l2);
    makeline(v[1], v[2], l0);
    makeline(v[2], v[0], l1);
    for( y=b.ymin; y<b.ymax, y++ ) for( x=b.xmin; x<b.xmax, x++ ){
        e0 = l0.A * x + l0.B * y + l0.C;
        e2 = l2.A * x + l2.B * y + l2.C;
        if( inside(e0, l0) & inside(e1, l1) & inside(e2, l2) )
            fragment(x, y);
    }
}
OpenGL has more than triangles…

Images

Bitmaps
OpenGI Pipeline

1. **Commands Processor**
   - Individual Vertices
   - Transformed Vertices
   - Primitives

2. **Per-vertex ops**
   - Primitives assembly

3. **Per-fragment ops**
   - Rasterization

4. **Framebuffer ops**
   - Texturing
     - Shaded Fragments
     - Pixels in the framebuffer

5. **Display**
Framebuffer

Example Framebuffer: 1440 x 900

The viewport is the portion of the window that can be drawn in, no pixels will appear outside the viewport.

Viewport (256x256)

Window (512 x 512)

All coordinates are integers; they refer to pixel locations in the framebuffer.
Frame Buffer Operations

Operation

- Test window ownership
- Test scissor and stencil mask
- Test alpha (transparency)
- Test depth (z-buffer)

Blending or compositing

Textured Fragments  Framebuffer Pixels
Depth Buffer (Z-Buffer)

- Initialize z-buffer to $z_{max}$
- Interpolate $z$ across the triangle
- Draw fragment if closer

```cpp
if (frag.Z < Z[frag.X][frag.Y]) {
}
```

Frame Buffering

Store image in a buffer to separate display refresh rate from drawing rate:

Single-buffer
- Draw into display buffer directly
- May see picture being drawn

Double-buffer
- Display “front” buffer
- Draw into “back” buffer (can’t see drawing)
- Swap front and back (idle while waiting for vertical sync)

Triple-buffer
- Avoid waiting for vertical sync
Graphics Processing Unit

- OpenGL commands communicate with the GPU
- The GPU is designed to rapidly manipulate and alter memory to accelerate the building of images in a framebuffer
- In contrast, our ray tracer will be implemented on the CPU (although there are ray tracers that utilize the GPU, e.g. Nvidia Optix)
What’s in a GPU?

- Shader Core
- Shader Core
- Shader Core
- Shader Core
- Tex
- Tex
- Tex
- Tex
- Primitive Assembly
- Rasterizer
- Framebuffer Ops
- Work Distributor
PC

3.0 Ghz Intel Core2 Duo
- Core 1
- Core 2
- 4MB L2 Cache

2GB main memory (DDR2)

NVIDIA GeForce 8800 GTX (575 MHz)
- (16 cores)

512MB video Memory (GDDR3)

NVIDIA 8800GTX

PCIe Bus (v1 = 4 GB/sec)
- 12.8 GB/sec
- 84 GB/sec

System board (Intel D975)
- 2GB main memory (DDR2)
- 4MB L2 Cache
- 512MB video Memory (GDDR3)
Xbox 360

3.2 Ghz PowerPC CPU
Core 1  Core 2  Core 3
L2 Cache

500 Mhz ATI GPU
48 3D Cores
Frame buffer
Video out

512 MB memory

IO Chip

controllers/ethernet/audio/DVD/etc.

Display (TV)
PS3

3.2 Ghz Cell

PPC Core

SPU0

SPU1

SPU2

SPU3

SPU4

SPU5

SPU6

SPU7

L2 Cache

IO Chip

256 MB Memory
(XDR)

256 MB video
Memory
(GDDR3)

Multiple 3D
cores

Video out

550 Mhz NVIDIA RSX GPU

controllers/ethernet/
audio/DVD/etc.

Display (TV)
Hybrid CPU-GPUs

Intel Sandybridge

Apple A5