Scanline Renderer

• Think of it as setting up a virtual environment/world that mimics the real world
  ■ but cheats a bit for efficiency, using what we know about light and our eye... e.g., from the last lecture
• Set up a virtual camera somewhere in the virtual world and point it in some direction
  ■ Put a virtual piece of film (that stores RGB values) into the camera, and take a picture
  ■ This film data is your final image
• Place some objects in front of the camera, including a floor/ground, some walls, ceiling/sky, etc.
  ■ How do we represent these virtual objects? Virtual geometry? Geometric Modeling
  ■ This is actually a two step process: (1) Make the objects (2) Place the objects in the scene
  ■ Making objects is itself a two step process: build the geometry & then paint it
• Don’t forget to put some lights into the scene, so it’s not completely dark
• Finally, snap the picture --- that is, code needs to emit light from the virtual light sources, bounce it off the geometry, and follow it into the camera onto the film
Homework

- This week’s homework and sample code (along with the lectures today and Thursday) is designed to help you learn your way around in a scanline renderer
  - We’re using OpenGL (there are other options...)
  - It provides real time feedback so you can experiment with stuff and learn by doing
  - By way of comparison, the ray tracer will be A LOT SLOWER

- Next week, we’ll focus on Geometric Modeling (in the lectures, sample code, and homework) to help you create and obtain geometry that you can put into your virtual scenes

- Make sure to keep the basic concepts from last Thursday’s lecture in mind as we progress
  - we’ll assume familiarity with that stuff, as we make various assumptions
Cameras
• Rendering a computer generated image is similar in spirit to how you see real world images
• Light is emitted from light sources, absorbed and reflected off of objects, and enters the pupil of the eye
• Without a pupil, light would scatter everywhere and would not make a coherent image on the cones of the eye
• With a pupil, light makes a coherent image on the cones
• Cameras, similarly, 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 as RGB values
  - Our scanline renderer will use a “virtual” camera

![Diagram of a camera showing light from an object being bent by the lens to form an image on the film.](image-url)
Pinhole Camera

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

- Light leaving any point travels in straight lines
- We only care about the lines that hit the pinhole (a single point)
- Infinite depth of field – i.e., everything is in focus (no blur)
- An 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 (i.e. the film) is placed in front of the pinhole, so that the image is not upside down.
- The frustum is the volume of the view (shown in blue below).
  - Notice the front and back clipping planes!
- The image plane (i.e. the film) is the clipping plane closest to the camera.
Helpful Hints

- Get very familiar with the OpenGL camera
- Do not put the camera too close to your objects (fisheye look)
  - Set it up like a real world scene
  - Significant/severe deductions for poor camera placement, fisheye, etc. (because your image will look terrible)
- Make sure your near and far clipping plane have enough space between them to contain your scene
  - Do not set the near clipping plane at the camera aperture!
- Make sure your objects are inside the viewing frustum

- Etc. Etc.
- This is a real time system, use the interactivity to your benefit...
Question 1

Where would you like to take your CG picture?


Etc.?
Geometric Models
Triangles

Stanford Bunny
69,451 triangles

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

• It’s easy to break convex polygons into triangles
  ■ As a result, can focus on optimizing the hardware/software implementation of only one primitive (i.e. optimize for 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...
GEOMETRIC MODELING

Next week...
Arranging your scene...
Transformations

Transform every point:

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

\[P' = T(P)\]

Moves triangles (and thus geometry)

- Create objects (or parts of objects) in convenient coordinates, and then transform into the scene later
- Make multiple copies (even different sizes) of a single object
- Connect kinematics of linkages/skeletons for characters/robots

http://youtube/6G3O60o5U7w
Matrix Representation

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

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

\[ x' = M x \]

Efficiency: can combine a sequence of transforms into a single transform (since matrix multiplication is associative):

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

- Compute the composite matrix M once and then apply it to many points
Hierarchical Transforms

- $M_1$ transforms the teapot from its model space to the table’s 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
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
Question 2

What sorts of objects will be in your scene?


Etc.?
Rotations
Rotation

\[
x' = Mx
\]

\[
glm::angleAxis(\text{angle}, \text{vec3(ax, ay, ax)})
\]
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...
Translations
Translation

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

$$x' = x + t_x$$
$$y' = y + t_y$$

\texttt{glm::translate(current, vec3(tx, ty, tz))}
Homogeneous Coordinates

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

• Conversely, the position \((x, y, z)\) of a 3D point 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...
Composite Transformations
Rotate 45 degrees about the point (1,1)...
Be Careful, Order Matters...

- Matrix multiplication is NOT commutative
- The rightmost transform is applied to the points first
Hint

• Place an object at the center of the target coordinate system (scene or an intermediate)

• Rotate the object into the desired orientation

• Translate the object in order to place it in the target coordinate system (scene or an intermediate)
Question 3

Looking for Artists:
3A. Name/Email
3B. Best Advice