The 3D Rasterization Pipeline

COS 426, Spring 2016
Princeton University
3D Rendering Scenarios

• Batch
  ◦ One image generated with as much quality as possible for a particular set of rendering parameters
    ▪ Take as much time as is needed (minutes)
    ▪ Useful for photorealism, movies, etc.

➢ Interactive
  ◦ Images generated in fraction of a second (<1/10) with user input, animation, varying camera, etc.
    ▪ Achieve highest quality possible in given time
    ▪ Visualization, games, etc.
3D Polygon Rendering

• Many applications use rendering of 3D polygons with direct illumination
3D Polygon Rendering

- Many applications use rendering of 3D polygons with direct illumination
Ray Casting Revisited

- For each sample ...
  - Construct ray from eye position through view plane
  - Find first surface intersected by ray through pixel
  - Compute color of sample based on illumination
3D Polygon Rendering

- We can render polygons faster if we take advantage of **spatial coherence**
3D Polygon Rendering

• How?
3D Polygon Rendering

- How?
3D Polygon Rendering

• How?
3D Rendering Pipeline  (for direct illumination)

This is a pipelined sequence of operations to draw 3D primitives into a 2D image.
3D Rendering Pipeline (for direct illumination)

- 3D Primitives
- Modeling Transformation
- Lighting
- Viewing Transformation
- Projection Transformation
- Clipping
- Viewport Transformation
- Scan Conversion
- Image

OpenGL executes steps of 3D rendering pipeline for each polygon

```c
glBegin(GL_POLYGON);
glVertex3f(0.0, 0.0, 0.0);
glVertex3f(1.0, 0.0, 0.0);
glVertex3f(0.0, 1.0, 0.0);
glEnd();
```
3D Rendering Pipeline (for direct illumination)

- 3D Primitives
- Modeling Transformation
- Lighting
- Viewing Transformation
- Projection Transformation
- Clipping
- Viewport Transformation
- Scan Conversion
- Image

Transform into 3D world coordinate system
3D Rendering Pipeline (for direct illumination)

3D Primitives

- Modeling Transformation
- Lighting
- Viewing Transformation
- Projection Transformation
- Clipping
- Viewport Transformation
- Scan Conversion

Transform into 3D world coordinate system

Illuminate according to lighting and reflectance

Image
3D Rendering Pipeline (for direct illumination)

- **3D Primitives**
- **Modeling Transformation**
- **Lighting**
- **Viewing Transformation**
- **Projection Transformation**
- **Clipping**
- **Viewport Transformation**
- **Scan Conversion**
- **Image**

Transform into 3D world coordinate system

Illuminate according to lighting and reflectance

Transform into 3D camera coordinate system
3D Rendering Pipeline (for direct illumination)

- **3D Primitives**
- **Modeling Transformation**
- **Lighting**
- **Viewing Transformation**
- **Projection Transformation**
- **Clipping**
- **Viewport Transformation**
- **Scan Conversion**
- **Image**

Transform into 3D world coordinate system

Illuminate according to lighting and reflectance

Transform into 3D camera coordinate system

Transform into 2D camera coordinate system
3D Rendering Pipeline (for direct illumination)

3D Primitives

Modeling Transformation

Transform into 3D world coordinate system

Lighting

Illuminate according to lighting and reflectance

Viewing Transformation

Transform into 3D camera coordinate system

Projection Transformation

Transform into 2D camera coordinate system

Clipping

Clip primitives outside camera’s view

Viewport Transformation

Scan Conversion

Image
3D Rendering Pipeline (for direct illumination)

- 3D Primitives
  - Modeling Transformation
  - Lighting
  - Viewing Transformation
  - Projection Transformation
  - Clipping
  - Viewport Transformation
  - Scan Conversion

Transform into 3D world coordinate system

Illuminate according to lighting and reflectance

Transform into 3D camera coordinate system

Transform into 2D camera coordinate system

Clip primitives outside camera’s view

Transform into image coordinate system

Image
3D Rendering Pipeline (for direct illumination)

3D Primitives

Modeling Transformation

Lighting

Viewing Transformation

Projection Transformation

Clipping

Viewport Transformation

Scan Conversion

Image

Transform into 3D world coordinate system

Illuminate according to lighting and reflectance

Transform into 3D camera coordinate system

Transform into 2D camera coordinate system

Clip primitives outside camera’s view

Transform into image coordinate system

Draw pixels (includes texturing, hidden surface, ...)

3D Rendering Pipeline (for direct illumination)

- **3D Primitives**
  - **Modeling Transformation**
  - **Lighting**
  - **Viewing Transformation**
  - **Projection Transformation**
  - **Clipping**
  - **Viewport Transformation**
  - **Scan Conversion**

- **Transform into 3D world coordinate system**
- Illuminate according to lighting and reflectance
- **Transform into 3D camera coordinate system**
- **Transform into 2D camera coordinate system**
- Clip primitives outside camera’s view
- **Transform into image coordinate system**
- Draw pixels (includes texturing, hidden surface, ...)

Image
Transformations

$$p(x,y,z)$$

Modeling Transformation

$$3D \text{ Object Coordinates}$$

Viewing Transformation

$$3D \text{ World Coordinates}$$

Projection Transformation

$$3D \text{ Camera Coordinates}$$

Viewport Transformation

$$2D \text{ Screen Coordinates}$$

$$2D \text{ Image Coordinates}$$

$$p'(x',y')$$

Transformations map points from one coordinate system to another
Viewing Transformations

\( p(x,y,z) \)

- **Modeling Transformation**
  - 3D Object Coordinates

- **Viewing Transformation**
  - 3D World Coordinates

- **Projection Transformation**
  - 3D Camera Coordinates

- **Viewport Transformation**
  - 2D Screen Coordinates

\( p'(x',y') \)

**Viewing Transformations**
Review: Viewing Transformation

- Mapping from world to camera coordinates
  - Eye position maps to origin
  - Right vector maps to X axis
  - Up vector maps to Y axis
  - Back vector maps to Z axis
Review: Camera Coordinates

- Canonical coordinate system
  - Convention is right-handed (looking down -z axis)
  - Convenient for projection, clipping, etc.

Camera right vector maps to X axis
Camera up vector maps to Y axis
Camera back vector maps to Z axis (pointing out of page)
Finding the viewing transformation

• We have the camera (in world coordinates)

• We want $T$ taking objects from world to camera

$$p^c = T \ p^w$$

• Trick: find $T^{-1}$ taking objects in camera to world

$$p^w = T^{-1} p^c$$

\[
\begin{bmatrix}
  x' \\
  y' \\
  z' \\
  w'
\end{bmatrix} =
\begin{bmatrix}
  a & b & c & d \\
  e & f & g & h \\
  i & j & k & l \\
  m & n & o & p
\end{bmatrix}
\begin{bmatrix}
  x \\
  y \\
  z \\
  w
\end{bmatrix}
\]
Finding the Viewing Transformation

• Trick: map from camera coordinates to world
  ○ Origin maps to eye position
  ○ Z axis maps to Back vector
  ○ Y axis maps to Up vector
  ○ X axis maps to Right vector

\[
\begin{bmatrix}
  x' \\
y' \\
z' \\
w'
\end{bmatrix} =
\begin{bmatrix}
  R_x & U_x & B_x & E_x \\
  R_y & U_y & B_y & E_y \\
  R_z & U_z & B_z & E_z \\
  R_w & U_w & B_w & E_w
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix}
\]

• This matrix is $T^{-1}$ so we invert it to get $T$ … easy!
Viewing Transformations

\[ p(x,y,z) \]

- Modeling Transformation
  - 3D Object Coordinates
- Viewing Transformation
  - 3D World Coordinates
- Projection Transformation
  - 3D Camera Coordinates
- Viewport Transformation
  - 2D Screen Coordinates
  - 2D Image Coordinates
  \[ p'(x',y') \]
Projection

• General definition:
  ◦ Transform points in $n$-space to $m$-space ($m<n$)

• In computer graphics:
  ◦ Map 3D camera coordinates to 2D screen coordinates
Taxonomy of Projections

Planar geometric projections

Parallel
- Orthographic
  - Top (plan)
  - Front elevation
  - Side elevation
- Axonometric
  - Isometric
    - Other

Perspective
- One-point
  - Cabinet
    - Cavalier
    - Other
- Two-point
  - Three-point
Taxonomy of Projections

Planar geometric projections

Parallel

Orthographic
- Top (plan)
- Front elevation
- Side elevation
- Axonometric

Oblique
- Cabinet
- Cavalier

One-point
- Two-point
- Three-point

Perspective

Other
Parallel Projection

- Center of projection is at infinity
  - Direction of projection (DOP) same for all points
Orthographic Projections

- DOP perpendicular to view plane
Parallel Projection Matrix

- General parallel projection transformation:

\[
\begin{bmatrix}
1 & 0 & L \cos \phi & 0 \\
0 & 1 & L \sin \phi & 0 \\
0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x_c \\
y_c \\
z_c \\
1
\end{bmatrix}
\]
Parallel Projection View Volume

Parallelepiped View Volume

Back Plane

Front Plane

Window

$Z_V$
Taxonomy of Projections

Planar geometric projections

Parallel

Orthographic
- Top (plan)
- Front elevation

Axonometric
- Side elevation

Oblique
- Cabinet
- Cavalier

One-point

Two-point

Three-point

Perspective

Other

FVFHP Figure 6.10
Perspective Projection

• Map points onto “view plane” along “projectors” emanating from “center of projection” (COP)
Perspective Projection View Volume

Frustum View Volume

View Plane

Back Plane

Front Plane

Window

Projection Reference Point

$z_v$
What are the coordinates of the point resulting from projection of \((x,y,z)\) onto the view plane?
Perspective Projection

• Compute 2D coordinates from 3D coordinates with similar triangles

\[(x, y, z)\]

\[(xD/z, yD/z)\]
Perspective Projection Matrix

- 4x4 matrix representation?

\[
\begin{align*}
    x_s &= x_c D / z_c \\
    y_s &= y_c D / z_c \\
    z_s &= D \\
    w_s &= 1
\end{align*}
\]

\[
\begin{bmatrix}
    x_s \\
    y_s \\
    z_s \\
    w_s
\end{bmatrix} = \begin{bmatrix}
    \? & \? & \? & \? \\
    \? & \? & \? & \? \\
    \? & \? & \? & \? \\
    \? & \? & \? & \? \\
\end{bmatrix} \begin{bmatrix}
    x_c \\
    y_c \\
    z_c \\
    1
\end{bmatrix}
\]
Perspective Projection Matrix

- 4x4 matrix representation?

\[
\begin{align*}
x_s &= x_c \frac{D}{z_c} & x_s &= x'/w' & x' &= x_c \\
y_s &= y_c \frac{D}{z_c} & y_s &= y'/w' & y' &= y_c \\
z_s &= D & z_s &= z'/w' & z' &= z_c \\
w_s &= 1 & w' &= z_c / D
\end{align*}
\]

\[
\begin{bmatrix}
x_s \\
y_s \\
z_s \\
w_s
\end{bmatrix}
= \begin{bmatrix}
? & ? & ? & ?
\end{bmatrix}
\begin{bmatrix}
x_c \\
y_c \\
z_c \\
1
\end{bmatrix}
\]
Perspective Projection Matrix

• 4x4 matrix representation?

\[
\begin{align*}
x_s &= x_c D / z_c & x_s &= x'/w' & x' &= x_c \\
y_s &= y_c D / z_c & y_s &= y'/w' & y' &= y_c \\
z_s &= D & z_s &= z'/w' & z' &= z_c \\
w_s &= 1 & w' &= z_c / D
\end{align*}
\]

\[
\begin{bmatrix}
x_s \\
y_s \\
z_s \\
w_s
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 1/D & 0
\end{bmatrix}
\begin{bmatrix}
x_c \\
y_c \\
z_c \\
1
\end{bmatrix}
\]
Perspective Projection Matrix

- In practice, want to compute a value related to depth to include in visibility calculations

\[
\begin{align*}
    x_s &= x_c \frac{D}{z_c} \\
    y_s &= y_c \frac{D}{z_c} \\
    z_s &= -\frac{D}{z_c} \\
    w_s &= 1
\end{align*}
\]

\[
\begin{bmatrix}
    x_s \\
    y_s \\
    z_s \\
    w_s
\end{bmatrix}
= 
\begin{bmatrix}
    1 & 0 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 & 0 \\
    0 & 0 & 0 & 0 & -1 \\
    0 & 0 & 1/D & 0 & 0
\end{bmatrix}
\begin{bmatrix}
    x_c \\
    y_c \\
    z_c \\
    1
\end{bmatrix}
\]
Perspective vs. Parallel

• Perspective projection
  + Size varies inversely with distance - looks realistic
    – Distance and angles are not (in general) preserved
    – Parallel lines do not (in general) remain parallel

• Parallel projection
  + Good for exact measurements
  + Parallel lines remain parallel
    – Angles are not (in general) preserved
    – Less realistic looking
Transformations map points from one coordinate system to another.

- **Modeling Transformation**
  - 3D Object Coordinates

- **Viewing Transformation**
  - 3D World Coordinates
  - 3D Camera Coordinates

- **Projection Transformation**
  - 3D Camera Coordinates
  - 2D Screen Coordinates

- **Viewport Transformation**
  - 2D Screen Coordinates
  - 2D Image Coordinates

\[ p(x,y,z) \]

\[ p'(x',y') \]
Viewport Transformation

- Transform 2D geometric primitives from screen coordinate system (normalized device coordinates) to image coordinate system (pixels)
Viewport Transformation

• Window-to-viewport mapping

\[
\begin{align*}
vx &= vx1 + (wx - wx1) \times (vx2 - vx1) / (wx2 - wx1); \\
vy &= vy1 + (wy - wy1) \times (vy2 - vy1) / (wy2 - wy1);
\end{align*}
\]
Summary of Transformations

\[ p(x, y, z) \]

- **Modeling Transformation**
  - 3D Object Coordinates

- **Viewing Transformation**
  - 3D World Coordinates
  - 3D Camera Coordinates

- **Projection Transformation**
  - 2D Screen Coordinates

- **Viewport Transformation**
  - 2D Image Coordinates

\[ p'(x', y') \]

- **Modeling transformation**
- **Viewing transformations**
- **Viewport transformation**
3D Rendering Pipeline (for direct illumination)

- 3D Primitives
- Modeling Transformation
  - 3D Modeling Coordinates
  - 3D World Coordinates
  - Lighting
  - 3D World Coordinates
  - Viewing Transformation
    - 3D Camera Coordinates
    - Projection Transformation
      - 2D Screen Coordinates
      - Clipping
        - 2D Screen Coordinates
        - Viewport Transformation
          - 2D Screen Coordinates
          - Scan Conversion
            - 2D Image Coordinates
            - Image
              - 2D Image Coordinates
Clipping

• Avoid drawing parts of primitives outside window
  ◦ Window defines part of scene being viewed
  ◦ Must draw geometric primitives only inside window
Polygon Clipping

• Find the part of a polygon inside the clip window?
Polygon Clipping

• Find the part of a polygon inside the clip window?

After Clipping
Sutherland Hodgeman Clipping

- Clip to each window boundary one at a time (for convex polygons)
Sutherland Hodgeman Clipping

- Clip to each window boundary one at a time
Sutherland Hodgeman Clipping

• Clip to each window boundary one at a time
Sutherland Hodgeman Clipping

- Clip to each window boundary one at a time
Sutherland Hodgeman Clipping

- Clip to each window boundary one at a time
Clipping to a Boundary

- Do **inside** test for each point in sequence,
- Insert new points when cross window boundary,
- Remove points outside window boundary
Clipping to a Boundary

- Do **inside** test for each point in sequence.
- Insert new points when cross window boundary.
- Remove points outside window boundary.
Clipping to a Boundary

- Do **inside** test for each point in sequence,
  - Insert new points when cross window boundary,
  - Remove points outside window boundary
Clipping to a Boundary

- Do **inside** test for each point in sequence,
  Insert new points when cross window boundary,
  Remove points outside window boundary
Clipping to a Boundary

- Do **inside** test for each point in sequence,
  - **Insert new points** when cross window boundary,
  - **Remove points** outside window boundary
Clipping to a Boundary

- Do **inside** test for each point in sequence,
  Insert new points when cross window boundary,
  Remove points outside window boundary
Clipping to a Boundary

- Do **inside** test for each point in sequence,
- Insert new points when cross window boundary,
- Remove points outside window boundary
Clipping to a Boundary

- Do **inside** test for each point in sequence,
  **Insert new points** when cross window boundary,
- **Remove points** outside window boundary

Window Boundary

Points: $P_1, P_2, P_3, P_4, P_5, P', P''$
Clipping to a Boundary

- Do **inside** test for each point in sequence,
- **Insert new points** when cross window boundary,
- **Remove points** outside window boundary

Window Boundary

- $P'$
- $P_1$
- $P_2$
- $P''$
3D Rendering Pipeline (for direct illumination)

- **3D Primitives**
- **Modeling Transformation**
- **Lighting**
- **Viewing Transformation**
- **Projection Transformation**
- **Clipping**
- **Viewport Transformation**
- **Scan Conversion**

**Coordinate Systems:**
- 3D Modeling Coordinates
- 3D World Coordinates
- 3D Camera Coordinates
- 2D Screen Coordinates
- 2D Image Coordinates

**Image**

**Viewing Window**
3D Rendering Pipeline (for direct illumination)

- **Scan Conversion**
  - 2D Image Coordinates

- **Viewport Transformation**
  - 2D Screen Coordinates

- **Clipping**
  - 2D Screen Coordinates

- **Projection Transformation**
  - 3D Camera Coordinates

- **Viewing Transformation**
  - 3D World Coordinates

- **Lighting**
  - 3D World Coordinates

- **Modeling Transformation**
  - 3D Modeling Coordinates

- **3D Primitives**
  - 3D World Coordinates

Standard (aliased) Scan Conversion
3D Rendering Pipeline (for direct illumination)

3D Primitives

Modeling Transformation

3D Modeling Coordinates

Lighting

3D World Coordinates

Viewing Transformation

3D World Coordinates

Projection Transformation

3D Camera Coordinates

Clipping

2D Screen Coordinates

Viewport Transformation

2D Screen Coordinates

Scan Conversion

2D Image Coordinates

Image

2D Image Coordinates

Antialiased Scan Conversion

P

P₁

P₃
Scan Conversion

• Render an image of a geometric primitive by setting pixel colors

```c
void SetPixel(int x, int y, Color rgba)
```

• Example: Filling the inside of a triangle
Triangle Scan Conversion

• Properties of a good algorithm
  ○ Symmetric
  ○ Straight edges
  ○ No cracks between adjacent primitives
  ○ (Antialiased edges)
  ○ FAST!
Simple Algorithm

- Color all pixels inside triangle

```c
void ScanTriangle(Triangle T, Color rgba){
    for each pixel P in bbox(T){
        if (Inside(T, P))
            SetPixel(P.x, P.y, rgba);
    }
}
```
Triangle Sweep-Line Algorithm

• Take advantage of spatial coherence
  ◦ Compute which pixels are inside using horizontal spans
  ◦ Process horizontal spans in scan-line order

• Take advantage of edge linearity
  ◦ Use edge slopes to update coordinates incrementally
void ScanTriangle(Triangle T, Color rgba) {
    for each edge pair {
        initialize \( x_L, x_R \);
        compute \( \frac{dx_L}{dy_L} \) and \( \frac{dx_R}{dy_R} \);
        for each scanline at \( y \)
            for (int \( x = x_L; \ x \leq x_R; \ x++ \))
                SetPixel(\( x, y, \) rgba);
        \( x_L += \frac{dx_L}{dy_L}; \)
        \( x_R += \frac{dx_R}{dy_R}; \)
    }
}
void ScanTriangle(Triangle T, Color rgba) {
    for each edge pair {
        initialize $x_L$, $x_R$;
        compute $dx_L/dy_L$ and $dx_R/dy_R$;
        for each scanline at $y$
            for (int $x = x_L; x <= x_R; x++$)
                SetPixel($x, y, rgba$);
        $x_L += dx_L/dy_L$;
        $x_R += dx_R/dy_R$;
    }
}

Minimize computation in inner loops
GPU Architecture

GeForce 6 Series Architecture
GPU Architecture

GeForce 6 Series Architecture