The 3D Rasterization Pipeline

COS 426
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 photorealismism, 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)

OpenGL executes steps of 3D rendering pipeline for each polygon.

```gl
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

Transform into 3D world coordinate system

Lighting

Viewing Transformation

Projection Transformation

Clipping

Viewport Transformation

Scan Conversion

Image
3D Rendering Pipeline (for direct illumination)

3D Primitives

Modeling Transformation

Lighting

Transform into 3D world coordinate system

Illuminate according to lighting and reflectance

Viewing Transformation

Projection Transformation

Clipping

Viewport Transformation

Scan Conversion

Image
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

- Clipping

- Viewport Transformation

- Scan Conversion
  - Image
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
- Viewport Transformation
- Scan Conversion
- Image
3D Rendering Pipeline (for direct illumination)

- **3D Primitives**
  - Transform into 3D world coordinate system
- **Modeling Transformation**
- **Lighting**
  - Illuminate according to lighting and reflectance
- **Viewing Transformation**
- **Projection Transformation**
  - Transform into 3D 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
  - 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
  - Transform into image coordinate system

- Scan Conversion
  - Image
3D Rendering Pipeline (for direct illumination)

1. **3D Primitives**
2. **Modeling Transformation**
3. **Lighting**
   - Illuminate according to lighting and reflectance
4. **Viewing Transformation**
5. **Projection Transformation**
   - Transform into 3D camera coordinate system
6. **Clipping**
   - Clip primitives outside camera’s view
7. **Viewport Transformation**
8. **Scan Conversion**
   - Transform into image coordinate system
9. **Draw pixels (includes texturing, hidden surface, ...)**

Image
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**
  - Transform into image coordinate system

- **Scan Conversion**
  - Draw pixels (includes texturing, hidden surface, ...)

**Image**
Transformations map points from one coordinate system to another.

Transformations

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

1. Modeling Transformation
   - 3D Object Coordinates

2. Viewing Transformation
   - 3D World Coordinates

3. Projection Transformation
   - 3D Camera Coordinates

4. Viewport Transformation
   - 2D Screen Coordinates

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

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

3D Camera
Coordinates

3D Object
Coordinates

3D World
Coordinates

2D Image Coordinates

2D Screen Coordinates

3D Camera
Coordinates

3D Object
Coordinates

3D World
Coordinates
Viewing Transformations

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

\[ \rightarrow \quad 3D \ Object \ Coordinates \]

Modeling Transformation

\[ \rightarrow \quad 3D \ World \ Coordinates \]

Viewing Transformation

\[ \rightarrow \quad 3D \ Camera \ Coordinates \]

Projection Transformation

\[ \rightarrow \quad 2D \ Screen \ Coordinates \]

Viewport Transformation

\[ \rightarrow \quad 2D \ Image \ Coordinates \]

\[ p'(x', y') \]
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 up vector maps to Y axis
Camera back vector maps to Z axis (pointing out of page)
Camera right vector maps to X axis
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

\[ p(x, y, z) \rightarrow 3D \text{ Object Coordinates} \]

- **Modeling Transformation**
  \[ 3D \text{ World Coordinates} \]
- **Viewing Transformation**
  \[ 3D \text{ Camera Coordinates} \]
- **Projection Transformation**
  \[ 2D \text{ Screen Coordinates} \]
- **Viewport Transformation**
  \[ 2D \text{ Image Coordinates} \]

\[ p'(x', y') \]
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
    Axonometric
      Side elevation
    Isometric
      Other
  Oblique
    Cabinet
      Cavalier
    Other
  Perspective
    One-point
    Two-point
    Three-point
Taxonomy of Projections

Planar geometric projections

Parallel

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

Axonometric
  - Isometric

Oblique
  - Cabinet
  - Cavalier

One-point

Perspective
  - Two-point
  - Three-point

Other
Parallel Projection

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

Angel Figure 5.4
Orthographic Projections

- DOP perpendicular to view plane
Parallel Projection Matrix

- General parallel projection transformation:

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

H&B Figure 12.30
Taxonomy of Projections

Planar geometric projections

Parallel

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

Axonometric
  - Isometric

Oblique
  - Cabinet
  - Cavalier

One-point

Two-point

Three-point

Perspective

Other

Other

FVFHP Figure 6.10
Perspective Projection

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

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

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
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?

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

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

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

- 4x4 matrix representation?

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

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

- In practice, want to compute a value related to depth to include in z-buffer

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

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

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

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

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

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

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

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

- **2D Screen Coordinates**

- **2D Image Coordinates**

Transformations map points from one coordinate system to another.

- **3D Camera Coordinates**

- **3D World Coordinates**

- **3D Object Coordinates**

- **3D Screen Coordinates**

- **3D Image Coordinates**
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

\[ p(x,y,z) \rightarrow \text{Modeling Transformation} \rightarrow 3D \text{ Object Coordinates} \]

\[ \rightarrow \text{Viewing Transformation} \rightarrow 3D \text{ World Coordinates} \]

\[ \rightarrow \text{Projection Transformation} \rightarrow 3D \text{ Camera Coordinates} \]

\[ \rightarrow \text{Viewport Transformation} \rightarrow 2D \text{ Screen Coordinates} \]

\[ \rightarrow 2D \text{ Image Coordinates} \rightarrow p'(x',y') \]

Modeling transformation

Viewing transformations

Viewport transformation
3D Rendering Pipeline (for direct illumination)

3D Primitives

Modeling Transformation

Lighting

Viewing Transformation

Projection Transformation

Clipping

Viewport Transformation

Scan Conversion

Image

3D Modeling Coordinates

3D World Coordinates

3D World Coordinates

3D Camera Coordinates

2D Screen Coordinates

2D Screen Coordinates

2D Screen Coordinates

2D Image Coordinates

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?

Before Clipping
Polygon Clipping

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

Window Boundary

- $P_1$
- $P_2$
- $P_3$
- $P_4$
- $P_5$

Inside

Outside

$P'$
Clipping to a Boundary

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

- Do **inside** test for each point in sequence,
- **Insert new points** when cross window boundary,
- **Remove points** outside window boundary
3D Rendering Pipeline (for direct illumination)

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

3D Modeling Coordinates
3D World Coordinates
3D World Coordinates
3D Camera Coordinates
2D Screen Coordinates
2D Screen Coordinates
2D Image Coordinates
2D Image Coordinates

Viewing Window
3D Rendering Pipeline (for direct illumination)

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

Image

Standard (aliased) Scan Conversion

P, P₁, P₃
3D Rendering Pipeline (for direct illumination)

3D Primitives → 3D Modeling Coordinates

Modeling Transformation

Lighting

Viewing Transformation

Projection Transformation

Clipping

Viewport Transformation

Scan Conversion

Image

3D World Coordinates

3D World Coordinates

3D Camera Coordinates

2D Screen Coordinates

2D Screen Coordinates

2D Image Coordinates

2D Image Coordinates

Antialiased Scan Conversion

P₁

P₃

P

2D Image Coordinates
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 $dx_L/dy_L$ and $dx_R/dy_R$;
        for each scanline at $y$
            for (int $x = x_L$; $x <= x_R$; $x++$)
                SetPixel($x$, $y$, rgba);
        $x_L += dx_L/dy_L$;
        $x_R += dx_R/dy_R$;
    }
}
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