The 3D Rasterization Pipeline

COS 426, Fall 2022
3D Rendering Scenarios

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

  ➢ Interactive
  ◦ Images generated dynamically, in fraction of a second (e.g., 1/30) as user controls rendering parameters (e.g., camera)
    » Achieve highest quality possible in given time
    » Visualization, games, etc.
3D Polygon Rendering

• Many applications render 3D polygons with direct illumination

Valve
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 Rasterization

• We can render polygons faster if we take advantage of spatial coherence
3D Polygon Rasterization

• How?
3D Polygon Rasterization

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

3D Primitives

Modeling Transformation

Lighting

Viewing Transformation

Projection Transformation

Clipping

Viewport Transformation

Scan Conversion

Image

Vertex coordinates
Rasterization 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
Rasterization Pipeline (for direct illumination)

3D Primitives

Modeling Transformation
Transform into 3D world coordinate system

Lighting
Illuminate according to lighting and reflectance

Viewing Transformation

Projection Transformation

Clipping

Viewport Transformation

Scan Conversion

Image
Rasterization 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
Rasterization 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
Rasterization 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
Rasterization Pipeline (for direct illumination)

1. 3D Primitives
2. Modeling Transformation
3. Lighting
4. Viewing Transformation
5. Projection Transformation
6. Clipping
7. Viewport Transformation
8. Scan Conversion
9. 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 ... in clip space
Rasterization 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 image coordinate system
Rasterization 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

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

**Viewing Transformation**

- Transform into 3D camera coordinate system
- Clip primitives outside camera's view
- Transform into 2D camera coordinate system
- Draw pixels (includes texturing, hidden surface, ...)

**Projection Transformation**

- Transform into 3D world coordinate system
Rasterization 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.

- **3D Object Coordinates**
- **3D World Coordinates**
- **3D Camera Coordinates**
- **2D Screen Coordinates**
- **2D Image Coordinates**

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

**Modeling Transformation**

$p(x,y,z) \rightarrow \text{3D World Coordinates}$

**Viewing Transformation**

$p(x,y,z) \rightarrow \text{3D Camera Coordinates}$

**Projection Transformation**

$p(x,y,z) \rightarrow \text{2D Screen Coordinates}$

**Viewport Transformation**

$p(x,y,z) \rightarrow \text{2D Image Coordinates}$

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

- **p(x,y,z)**
- **3D Object Coordinates**
- **Modeling Transformation**
  - **3D World Coordinates**
  - **Viewing Transformation**
    - **3D Camera Coordinates**
    - **Projection Transformation**
      - **2D Screen Coordinates**
      - **Viewport Transformation**
        - **2D Image Coordinates**
        - **p'(x',y')**
Review: Viewing Transformation

- Mapping from world to camera coordinates
  - Eye position maps to origin
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
  \[ p^c = T \cdot p^w \]
- Trick: find $T^{-1}$ taking objects in camera to world
  \[ p^w = T^{-1} \cdot p^c \]
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
  - 3D World Coordinates
  - Viewing Transformation
    - 3D Camera Coordinates
    - Projection Transformation
      - 2D Screen Coordinates
      - Viewport Transformation
        - 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
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
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
Taxonomy of Projections

Planar geometric projections

Parallel

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

Oblique
- Cabinet

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:

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

FVFHP Figure 6.10
Return to 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

\[(x, y, z) \rightarrow (x_D/z, y_D/z)\]
Perspective Projection Matrix

- 4x4 matrix representation?

\[
x_s = x_c \frac{D}{z_c} \\
y_s = y_c \frac{D}{z_c} \\
z_s = D \\
w_s = 1
\]

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

- 4x4 matrix representation?

\[ x_s = x_c D / z_c \quad x_s = x' / w' \quad x' = x_c \]
\[ y_s = y_c D / z_c \quad y_s = y' / w' \quad y' = y_c \]
\[ z_s = D \quad z_s = z' / w' \quad z' = z_c \]
\[ w_s = 1 \quad w' = z_c / D \]
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 z-buffer

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

\[
\begin{bmatrix}
x_s \\
y_s \\
z_s \\
w_s
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 0 & -1 \\
0 & 0 & 1/D & 0
\end{bmatrix}
\begin{bmatrix}
x_c \\
y_c \\
z_c \\
1
\end{bmatrix}
\]
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**
  - 2D Screen Coordinates

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

\[ p(x,y,z) \rightarrow 3D \text{ Object Coordinates} \rightarrow 3D \text{ World Coordinates} \rightarrow 3D \text{ Camera Coordinates} \rightarrow 2D \text{ Screen Coordinates} \rightarrow 2D \text{ Image Coordinates} \rightarrow p'(x',y') \]
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

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

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

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

- **Modeling transformation**

- **Viewing transformations**

- **Viewport transformation**
Rasterization 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
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?
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 crossing window boundary
  - Remove points outside window boundary
Clipping to a Boundary

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

![Diagram showing points P1, P2, P3, P4, P5 with inside and outside regions labeled.](image-url)
Clipping to a Boundary

• Do inside test for each point in sequence
  ◦ Insert new points when crossing window boundary
  ◦ Remove points outside window boundary
Clipping to a Boundary

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

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

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

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

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

• Do **inside** test for each point in sequence
  ○ Insert new points when crossing window boundary
  ○ Remove points outside window boundary
Sutherland Hodgeman Failure

• Concave Polygons
• Concave Polygons
Rasterization Pipeline (for direct illumination)

3D Primitives

Modeling Transformation

Lighting

Viewing Transformation

Projection Transformation

Clipping

Viewport Transformation

Scan Conversion

Image

Viewport Window

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

2D Image Coordinates
Rasterization 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

Standard (aliased)
Scan Conversion

\[ P_1 \]
\[ P \]
\[ P_3 \]
Rasterization 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

Antialiased Scan Conversion
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
Triangle Sweep-Line Algorithm

```c
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;
    }
}
```
Triangle Sweep-Line Algorithm

```c
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_R/dy_L;
        x_R += dx_L/dy_L;
    }
}
```

Minimize computation in inner loops
GPU Architecture

NVIDIA architecture based on Fermi logical pipeline

When tessellation is not used, two principle phases are sufficient. Work is redistributed across entire GPU after each phase.

Work Distribution Crossbar sends triangle to raster engine(s) based on screen rectangle

Multiple GPCs with their SMs can be shading the pixels of one triangle.

GF 100 Memory Hierarchy
Uniform cache not shown, can cause warp-serialized access on divergent loads

latencies

tens of cycles

several hundred cycles

SM organizes threads in groups of 32 called warp. The threads within are processed in lock-step.

Each warp gets subset of register file. If a shader needs many registers -> less warps resident, less latency hiding

A given warp is processed in-order and it may take several executions until an instruction is advanced (depends on hardware generation and type of instruction).
The scheduler switches between warps to avoid waiting for instructions that take longer (memory fetches...).

Divergent behavior between threads within warp (if/else block, loops with varying iterations...) can increase computation time for all because of lock-step processing and may risk under utilizing cores.
GPU Architecture

Fermi, Kepler, Maxwell Evolution

Kepler and Maxwell work in principle similar to Fermi. The most obvious changes are typically in the SM design or number of NOPs. The overall design can be scaled from high-end desktop to mobile by varying the number of modules.

http://nvidia.com/pdf/GeForce_GTX_T90_Whitepaper_FINAL.PDF
GPU Architecture

NVIDIA architecture based on Fermi logical pipeline

When tessellation is not used, two principle phases are sufficient. Work is redistributed across entire GPU after each phase.

Work Distribution Crossbar sends triangle to raster engine(s) based on screen rectangle.

Multiple GPCs with their SMs can be shading the pixels of one triangle.

GF 100 Memory Hierarchy
Uniform cache not shown, can cause warp-serialized access on divergent loads

~ latencies
tens of cycles
several hundred cycles