Graphics Pipeline

CS535 Fall 2016

Daniel G. Aliaga
Department of Computer Science
Purdue University
Ray-tracing – Inverse mapping

for every pixel
   construct a ray from the eye
for every object in the scene
   intersect ray with object
   find closest intersection with the ray
   compute normal at point of intersection
   compute color for pixel
shOOT secondary rays
Pipeline – Forward mapping

Start from the geometric primitives to find the values of the pixels
Fixed Pipeline

Programmable Pipeline (High-level Shading Language)
Alternate Pipelines?

Standard pipeline....
A Sorting Classification of (Parallel) Graphics Pipelines

- Sort-first
- Sort-middle
- Sort-last
Sort First

Graphics database (arbitrarily partitioned)

Redistribute "raw" primitives

(Pre-transform)

Geometry processing

Rasterization

Display
Sort Middle

Graphics database (arbitrarily partitioned)

Redistribute screen-space primitives

Geometry processing

Rasterization

Display
Sort Last

Graphics database (arbitrarily partitioned)

Geometry processing
Rasterization

Redistribute pixels, samples, or fragments

(Compositing)

Display
Sort First

• **Advantages:**
  - Low communication requirements when the tessellation ratio and the degree of oversampling are high, or when frame-to-frame coherence can be exploited.
  - Processors implement entire rendering pipeline for a portion of the screen.

• **Disadvantages:**
  - Susceptible to load imbalance. Primitives may clump into regions, concentrating the work on a few renderers.
  - To take advantage of frame-to-frame coherence, retained mode and complex data handling code are necessary.
Sort Middle

• **Advantages:**
  – General and straightforward; redistribution occurs at a natural place in the pipeline.

• **Disadvantages:**
  – High communication costs if tessellation ratio is high.
  – Susceptible to load imbalance between rasterizers when primitives are distributed unevenly over the screen.
Sort Last

• **Advantages:**
  - Renderers implement the full rendering pipeline and are independent until pixel merging.
  - Less prone to load imbalance.
  - SL-full merging can be embedded in a linear network, making it linearly scalable.

• **Disadvantage:**
  - Pixel traffic may be extremely high, particularly when oversampling.
Graphics Rendering History

- IKONAS Graphics System (1978)
  - (basically a dedicated CPU)
Graphics Rendering History

- TAAC Graphics Accelerator Board (a GPU...)

![Image of TAAC programmable graphics processor - 1987](image_url)
Graphics Rendering History

• Silicon Graphics Personal Iris (1986)
  – CPU and GPU (a sort first architecture)
  – “gl” appeared
  – HP had “starbase”
  – (OpenGL appeared in 1991)
Graphics Rendering History

- Renderman: a programmable shading language (1986)
Graphics Rendering History

- PixelPlanes 1, 2, 3, 4, 5
  - (a sort middle architecture)
Graphics Rendering History

- **PixelFlow**
  - A sort-last architecture
  - Formally supports programmable shading
Graphics Rendering History

- PixelFlow -> HP
Graphics Rendering History

• PixelFlow → HP
Graphics Rendering History

• PixelFlow -> NVIDIA
Graphics Rendering History
“Standard” Graphics Pipeline

- **Geometry**
  - Modeling Transformation: Transform into 3D *world* coordinate system
  - Lighting: Simulate illumination and reflectance
  - Viewing Transformation: Transform into 3D *camera* coordinate system
  - Clipping: Clip primitives outside camera’s view
  - Projection: Transform into 2D camera coordinate system
  - Scan Conversion: Draw pixels (incl. texturing, hidden surface…)

→ Image
"Standard" Graphics Pipeline

- **Geometry**
  - Modeling Transformation: Transform into 3D *world* coordinate system
  - Lighting: Simulate illumination and reflectance
  - Viewing Transformation: Transform into 3D *camera* coordinate system
  - Clipping: Clip primitives outside camera’s view
  - Projection: Transform into 2D camera coordinate system
  - Scan Conversion: Draw pixels (incl. texturing, hidden surface…)

Image
Modeling Transformations

• Most popular transformations in graphics
  – Translation
  – Rotation
  – Scale
  – Projection

• In order to use a single matrix for all, we use homogeneous coordinates...
Modeling Transformations

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

Identity

\[
\begin{bmatrix}
    x' \\
    y' \\
    z' \\
    w
\end{bmatrix} = \begin{bmatrix}
    sx & 0 & 0 & 0 \\
    0 & sy & 0 & 0 \\
    0 & 0 & sz & 0 \\
    0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
    x \\
    y \\
    z \\
    w
\end{bmatrix}
\]

Scale

\[
\begin{bmatrix}
    x' \\
    y' \\
    z' \\
    w
\end{bmatrix} = \begin{bmatrix}
    1 & 0 & 0 & tx \\
    0 & 1 & 0 & ty \\
    0 & 0 & 1 & tz \\
    0 & 0 & 0 & 1
\end{bmatrix} \begin{bmatrix}
    x \\
    y \\
    z \\
    w
\end{bmatrix}
\]

Translation

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

Mirror over X axis
Modeling Transformations

Rotate around Z axis:

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
w'
\end{bmatrix} =
\begin{bmatrix}
\cos \Theta & -\sin \Theta & 0 & 0 \\
\sin \Theta & \cos \Theta & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix}
\]

Rotate around Y axis:

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
w'
\end{bmatrix} =
\begin{bmatrix}
\cos \Theta & 0 & -\sin \Theta & 0 \\
0 & 1 & 0 & 0 \\
\sin \Theta & 0 & \cos \Theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix}
\]

And many more…

Rotate around X axis:

\[
\begin{bmatrix}
x' \\
y' \\
z' \\
w'
\end{bmatrix} =
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & \cos \Theta & -\sin \Theta & 0 \\
0 & \sin \Theta & \cos \Theta & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}
\begin{bmatrix}
x \\
y \\
z \\
w
\end{bmatrix}
\]
"Standard" Graphics Pipeline

Geometry

Modeling Transformation
Transform into 3D *world* coordinate system

Lighting
Simulate illumination and reflectance

Viewing Transformation
Transform into 3D *camera* coordinate system

Clipping
Clip primitives outside camera’s view

Projection
Transform into 2D camera coordinate system

Scan Conversion
Draw pixels (incl. texturing, hidden surface…)

Image
Diffuse

(mostly)
Specular++
Environment Mapping
Subsurface Scattering

(a) High-res geometry  (b) Real-time hybrid map rendering  (c) Offline SSS rendering
Others

Transparency

Radiosity

Ambient occlusion
Others
Lighting and Shading

• Light sources
  – Point light
    • Models an omnidirectional light source (e.g., a bulb)
  – Directional light
    • Models an omnidirectional light source at infinity
  – Spot light
    • Models a point light with direction

• Light model
  – Ambient light
  – Diffuse reflection
  – Specular reflection
Lighting and Shading

• Diffuse reflection
  – Lambertian model
Lighting and Shading

• Diffuse reflection
  – Lambertian model
Lighting and Shading

• Diffuse reflection
  – Lambertian model

\[ I_D = K_D (N \cdot L) I_L \]
Lighting and Shading

• Specular reflection
  – Phong model
Lighting and Shading

• Specular reflection
  – Phong model

\[ I_S = K_S (V \cdot R)^n I_L \]
Lighting and Shading

• Specular reflection
  – Phong model
Computer Graphics Pipeline

Geometry

Modeling Transformation
- Transform into 3D world coordinate system

Lighting
- Simulate illumination and reflectance

Viewing Transformation
- Transform into 3D camera coordinate system

Clipping
- Clip primitives outside camera’s view

Projection
- Transform into 2D camera coordinate system

Scan Conversion
- Draw pixels (incl. texturing, hidden surface…)

Image
Viewing Transformation

\[ \tilde{x}_c = R(\tilde{X} - C) \]
\[ \tilde{x}_c = R\tilde{X} - RC \]
\[ \tilde{x}_c = R \begin{pmatrix} X \\ Y \\ Z \end{pmatrix} + \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \]

World-to-camera matrix \( M \)

\[ R = R_x R_y R_z \]
3x3 rotation matrices

\[ t = \begin{pmatrix} t_x \\ t_y \\ t_z \end{pmatrix} \]
translation vector
“Standard” Graphics Pipeline

Geometry

Modeling Transformation
  Transform into 3D world coordinate system

Lighting
  Simulate illumination and reflectance

Viewing Transformation
  Transform into 3D camera coordinate system

Clipping
  Clip primitives outside camera’s view

Projection
  Transform into 2D camera coordinate system

Scan Conversion
  Draw pixels (incl. texturing, hidden surface…)

Image
“Standard” Graphics Pipeline

Geometry

Modeling Transformation
- Transform into 3D *world* coordinate system

Lighting
- Simulate illumination and reflectance

Viewing Transformation
- Transform into 3D *camera* coordinate system

Clipping
- Clip primitives outside camera’s view

Projection
- Transform into 2D camera coordinate system

Scan Conversion
- Draw pixels (incl. texturing, hidden surface…)

Image
Perspective projection

\[
\begin{align*}
&\frac{y}{f} = \frac{Y}{Z} \\
y = f \frac{Y}{Z} & \quad \text{and} \quad x = f \frac{X}{Z}
\end{align*}
\]
Perspective Projection

\[
\begin{pmatrix}
x \\
y
\end{pmatrix}
= \begin{pmatrix}
\frac{fX}{Z} \\
\frac{fY}{Z}
\end{pmatrix}
\]

\[
\begin{pmatrix}
fX \\
fY \\
fZ
\end{pmatrix}
= \begin{pmatrix}
X \\
Y \\
Z \\
1
\end{pmatrix}
\]
OpenGL 3D Viewing

3D Viewing in OpenGL:

Position camera;
Specify projection.
OpenGL 3D Viewing

Camera always in origin, in direction of negative $z$-axis. Convenient for 2D, but not for 3D.
OpenGL 3D Viewing

Solution for view transform: Transform your model such that you look at it in a convenient way.

Approach 1: Carefully do it yourself. Apply rotations, translations, scaling, etc., before rendering the model.

Approach 2: Use `gluLookAt();`
OpenGL 3D Viewing

MatrixMode(GL_MODELVIEW);
gluLookAt(x0, y0, z0, xref, yref, zref, Vx, Vy, Vz);

x0, y0, z0: \( \mathbf{P}_0 \), viewpoint, location of camera;
xref, yref, zref: \( \mathbf{P}_{\text{ref}} \), centerpoint;
Vx, Vy, Vz: \( \mathbf{V} \), view-up vector.

Default: \( \mathbf{P}_0 = (0, 0, 0) \); \( \mathbf{P}_{\text{ref}} = (0, 0, -1) \); \( \mathbf{V} = (0, 1, 0) \).
Orthogonal projection:

MatrixMode(GL_PROJECTION);

\texttt{glOrtho}(xwmin, xwmax, ywmin, ywmax, dnear, dfar);

- \texttt{xwmin, xwmax, ywmin, ywmax:}
  - specification window

- \texttt{dnear: distance} to near clipping plane

- \texttt{dfar : distance} to far clipping plane

Select \texttt{dnear} and \texttt{dfar} right:

\texttt{dnear} < \texttt{dfar},

model fits between clipping planes.
OpenGL 3D Viewing

Perspective projection:

```c
MatrixMode(GL_PROJECTION);
glFrustum(xwmin, xwmax, ywmin, ywmax, dnear, dfar);
```

- **xwmin, xwmax, ywmin, ywmax**: specification window
- **dnear**: distance to near clipping plane
- **dfar**: distance to far clipping plane

Select **dnear** and **dfar** right:

- \( 0 < d\text{near} < d\text{far} \)
- model fits between clipping planes.

Standard projection:

- \( x\text{min} = -x\text{max}, \)
- \( y\text{min} = -y\text{max} \)
Finally, specify the viewport (just like in 2D):

```
glViewport(xvmin, yvmin, vpWidth, vpHeight);
```

- **xvmin, yvmin**: coordinates lower left corner (in pixel coordinates);
- **vpWidth, vpHeight**: width and height (in pixel coordinates);
OpenGL 2D Viewing

In short:

```
glMatrixMode(GL_PROJECTION);
glFrustum(xwmin, xwmax, ywmin, ywmax, dnear, dfar);
glViewport(xvmin, yvmin, vpWidth, vpHeight);
glMatrixMode(GL_MODELVIEW);
gluLookAt(x0, y0, z0, xref, yref, zref, Vx, Vy, Vz);
```

To prevent distortion, make sure that:

\[
\frac{ywmax - ywmin}{xwmax - xwmin} = \frac{vpWidth}{vpHeight}
\]

Make sure that you can deal with resize/reshape of the (OS) window.
“Standard” Graphics Pipeline

Geometry

Modeling Transformation
Transform into 3D world coordinate system

Lighting
Simulate illumination and reflectance

Viewing Transformation
Transform into 3D camera coordinate system

Clipping
Clip primitives outside camera’s view

Projection
Transform into 2D camera coordinate system

Scan Conversion
Draw pixels (incl. texturing, hidden surface…)

Image
Scan Conversion/Rasterization

- Determine which fragments get generated
- Interpolate parameters (colors, textures, normals, etc.)
Scan Conversion/Rasterization

- Determine which fragments get generated
- Interpolate parameters (colors, textures, normals, etc.)
Scan Conversion/Rasterization

- Determine which fragments get generated
- Interpolate parameters (colors, textures, normals, etc.)

• How?
Scan Conversion/Rasterization

- Determine which fragments get generated
- Interpolate parameters (colors, textures, normals, etc.)

- E.g., Barycentric coords (see whiteboard!)
Barycentric coordinates

\[ q = \alpha p_1 + \beta p_2 + \gamma p_3 \]

If \[ \alpha + \beta + \gamma = 1 \ and \ \{\alpha, \beta, \gamma\} \geq 0 \],
then \( q \) inside triangle \((p_1, p_2, p_3)\)

Can also write:
\[ q = \alpha p_1 + \beta p_2 + (1 - \alpha - \beta)p_3 \]
Barycentric coordinates

How to solve for $\alpha$ and $\beta$ in

$$q = \alpha p_1 + \beta p_2 + (1 - \alpha - \beta) p_3?$$

Two equations, two unknowns: use 2x2 matrix inversion...