Image synthesis

Romain Vergne – 2014/2015
What is it?

Geometry
BRDF
Light sources
Scene org.
Etc.

Image

shape
material
lighting
space

Physics
Optics
Vis. attention
Vis. perception
A 3D world

3D Models

Lights

Cameras

Materials

Rendering
Mathematical tools

- Vector norm
  \[ |\mathbf{x}| = \sqrt{x_1^2 + x_2^2 + \cdots + x_n^2} \]

- Normalization
  \[ \hat{\mathbf{x}} = \frac{\mathbf{x}}{|\mathbf{x}|} = \left( \frac{x_1}{|\mathbf{x}|}, \frac{x_2}{|\mathbf{x}|}, \ldots, \frac{x_n}{|\mathbf{x}|} \right)^T \]

- Scalar product
  \[ \mathbf{a} \cdot \mathbf{b} = |\mathbf{a}||\mathbf{b}| \cos(\theta) \]
  OR
  \[ \mathbf{a} \cdot \mathbf{b} = a_x b_x + a_y b_y + a_z b_z \]

- Cross product
  \[ \mathbf{a} \times \mathbf{b} = |\mathbf{a}||\mathbf{b}| \sin(\theta) \mathbf{n}, \text{ with } \mathbf{n}: \text{a unit perpendicular vector to the plane} \]
  OR
  \[ \mathbf{a} \times \mathbf{b} = \begin{vmatrix} \mathbf{e}_1 & \mathbf{e}_2 & \mathbf{e}_3 \\ a_x & a_y & a_z \\ b_x & b_y & b_z \end{vmatrix} = \begin{pmatrix} a_y b_z - a_z b_y \\ a_z b_x - a_x b_z \\ a_x b_y - a_y b_x \end{pmatrix} \]
3D transformations

- Exactly the same... with 4D matrices

Scaling

\[ S = \begin{pmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \]

Translation

\[ T = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \]

Rotations – along the 3 different axis

- Invariant along the \( x \) axis
- Invariant along the \( y \) axis
- Invariant along the \( z \) axis
Hierarchical transformations

- Complex objects
  - \( M \): initial matrix
  - \text{pushmatrix()}
  - \( M = M R_\alpha T \)
  - draw A
  - \( M = M R_\beta T \)
  - draw B
  - \text{pushmatrix()}
  - \( M = M R_\gamma T \)
  - draw C1
  - \text{popmatrix()}
  - \text{pushmatrix()}
  - \( M = M R_\gamma T \)
  - draw C2
  - \text{popmatrix()}
  - \text{pushmatrix()}
  - \( M = M R_\gamma T \)
  - draw C3
  - \text{popmatrix()}
  - \text{popmatrix()}

For each triangle
  - For each pixel
    - Does triangle cover pixel?
    - Kip closest hit

"Forward-Mapping" approach to Computer Graphics
Rasterization pipeline

1: Project vertices to 2D
2: Rasterize triangle
3: Compute per-pixel color
4: Test visibility
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

Used to define a camera and apply transformations to objects

How can we define a new camera position and orientation?
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

Usually defined with:
- A camera position \( e \)
- The viewing point \( c \)
- An up-vector \( u' \)

How can we find \( r, u, v, e \) from \( e, c \) and \( u' \)?
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

\[
\text{viewMatrix} \times \text{modelMatrix} \text{ gives you the modelview matrix}
\]

From world to eye coordinates
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

viewMatrix*modelMatrix gives you the modelview matrix

From world to eye coordinates

How can we compute the distance of any point p to the camera?
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

How can we transform eye coordinates into viewport coordinates?
**Rasterization pipeline**

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

**Orthographic projection**
- Object sizes do not change
- Parallel lines are kept parallel

**Perspective projection**
- Far objects are smaller
- Similar to the human eye
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

Define the volume of your scene (in camera space)
- left/right, top/bottom, near/far boundaries

Orthographic projection
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

And remap this volume into $[-1,1]^3$
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

Orthographic projection

\[
P = \begin{pmatrix}
\frac{2 \text{right-left}}{\text{right-left}} & 0 & 0 & \frac{-\text{right-left}}{\text{right-left}} \\
0 & \frac{2 \text{top-bottom}}{\text{top-bottom}} & 0 & \frac{-\text{top-bottom}}{\text{top-bottom}} \\
0 & 0 & -\frac{2 \text{far-near}}{\text{far-near}} & \frac{-\text{far-near}}{\text{far-near}} \\
0 & 0 & 0 & 1
\end{pmatrix}
\]
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

Define the volume of your scene (in camera space)
  - left/right, top/bottom, near/far boundaries

And remap this volume into $[-1,1]^3$
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

Perspective projection (same!!)
**Rasterization pipeline**

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

More intuitive with aspect, fovy, near and far parameters

\[
P = \begin{pmatrix}
\frac{f}{\text{aspect}} & 0 & 0 & 0 \\
0 & f & 0 & 0 \\
0 & 0 & \frac{\text{far}+\text{near}}{\text{near}−\text{far}} & \frac{2\text{far}\text{near}}{\text{near}−\text{far}} \\
0 & 0 & -1 & 0 \\
\end{pmatrix}
\]

where \( f = \frac{1}{\tan \left( \frac{\text{fovy}}{2} \right)} \) and \( \text{aspect} = \frac{w}{h} \)
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

- Can be concatenated

Diagram shows the process of transforming vertex data through various matrices and coordinate systems:

- Vertex data
- ModelView matrix: world to eye coordinates
- Projection matrix: eye to viewport coordinates
- Divide by w
- Viewport transform: window coordinates

ModelviewMatrix and ProjectionMatrix can be concatenated for efficiency.
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

```
Remember we use
homogeneous coordinates
```

```
Can be concatenated
```
**Rasterization pipeline**

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

---

**Homogeneous coordinates**

Can be concatenated

**Equation**

\[
(x_w, y_w) = \left( x_0 + \frac{x_x + 1}{2}w, y_0 + \frac{y_y + 1}{2}h \right)
\]
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

What about the projected z coordinate? Do you think it is useful?
What if we project surfaces (i.e. triangles) instead of points?
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

- For each pixel
  - Test 3 edge equations
  - If all pass, draw
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

\[
P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c
\]
\[
\alpha + \beta + \gamma = 1, \quad \alpha, \beta, \gamma \geq 0
\]
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

\[ P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c \]
\[ \alpha + \beta + \gamma = 1, \quad \alpha, \beta, \gamma \geq 0 \]

projective equivalence (up to scale)  
2D homogenous coordinates

\[
\begin{pmatrix}
  x' \\
  y' \\
  1
\end{pmatrix}
\sim
\begin{pmatrix}
  P'_x \\
  P'_y \\
  P'_w
\end{pmatrix}
= \begin{pmatrix}
  a'_x \\
  a'_y \\
  a'_z \\
  b'_x \\
  b'_y \\
  b'_z \\
  c'_x \\
  c'_y \\
  c'_z
\end{pmatrix}
\begin{pmatrix}
  \alpha \\
  \beta \\
  \gamma
\end{pmatrix}
\]

\(a', b', c'\) are the projected homogeneous coordinates
Rasterization pipeline

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

\[ P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c \]
\[ \alpha + \beta + \gamma = 1, \quad \alpha, \beta, \gamma \geq 0 \]

\[
\begin{pmatrix}
\alpha \\
\beta \\
\gamma
\end{pmatrix}
\sim
\begin{pmatrix}
a'_{x} & b'_{x} & c'_{x} \\
a'_{y} & b'_{y} & c'_{y} \\
a'_{w} & b'_{w} & c'_{w}
\end{pmatrix}^{-1}
\begin{pmatrix}
x \\
y \\
1
\end{pmatrix}
\]

\text{a', b', c'} \text{ are the projected homogeneous coordinates}
**Rasterization pipeline**

- Project vertices to 2D
- Rasterize triangle
- Compute per-pixel color
- Test visibility

- Store minimum distance to camera for each pixel in z-buffer
- If new\_z < zbuffer[x,y]
  - Zbuffer[x,y] = new\_z
  - Framebuffer[x,y] = new\_color
For each triangle
  - For each pixel
    - Does triangle cover pixel?
    - Kip closest hit

For each triangle
  - Compute projection
  - Compute interpolation matrix
  - Compute Bbox, clip bbox to screen limits
  - For each pixel x,y in bbox
    - Test edge functions
    - If all Ei > 0
      - Compute barycentrics
      - Interpolate z from vertices
      - If z < zbuffer[x,y]
        - Interpolate attributes (color, normal)
        - Framebuffer[x,y] = resulting color
For each triangle
  • For each pixel
    • Does triangle cover pixel?
    • Kip closest hit

- Done automatically (OpenGL)
- Todo yourself

For each triangle
- Compute projection (vertex processing)
- Compute interpolation matrix
- Compute Bbox, clip bbox to screen limits
- For each pixel x,y in bbox
  • Test edge functions
  • If all Ei > 0
    • Compute barycentrics
    • Interpolate z from vertices
    • If z < zbuffer[x,y]
      • Interpolate attributes (color, normal)
      • Framebuffer[x,y] = resulting color (fragment processing)

• Done automatically (OpenGL)
• Todo yourself
Rasterization advantages

- Modern scenes more complicated than images
  - 1920x1080 frame (1080p)
  - 64-bit color and 32-bit depth
  - 24 Mb memory

- Rasterization can stream over triangles
  - One triangle at a time
  - Parallelism
  - Memory optimization
Rasterization limitations

- Restricted to scan-convertible primitives (triangles)

- No unified handling of
  - Shadows
  - Reflection
  - Transparency

- Potential problem of overdraw
  - Depth complexity
  - Each pixel touched many times