The Traditional Graphics Pipeline

Last Time?
- Participating Media
- Measuring BRDFs
- 3D Digitizing & Scattering
- BSSRDFs
  - Monte Carlo Simulation
  - Dipole Approximation

Today
- Ray Casting / Tracing vs. Scan Conversion
- Traditional Graphics Pipeline
- Clipping
- Rasterization/Scan Conversion

Ray Casting / Tracing
- Advantages?
  - Smooth variation of normal, silhouettes
  - Generality: can render anything that can be intersected with a ray
  - Atomic operation, allows recursion
- Disadvantages?
  - Time complexity (N objects, R pixels)
  - Usually too slow for interactive applications
  - Hard to implement in hardware (lacks computation coherence, must fit entire scene in memory)

How Do We Render Interactively?
- Use graphics hardware (the graphics pipeline), via OpenGL, MesaGL, or DirectX
- Most global effects available in ray tracing will be sacrificed, but some can be approximated

Scan Conversion
- Given a primitive's vertices & the illumination at each vertex:
  ```
glBegin(GL_TRIANGLES)
glNormal3f(...)
glVertex3f(...)
glVertex3f(...)
glVertex3f(...)
glEnd();
```
- Figure out which pixels to "turn on" to render the primitive
- Interpolate the illumination values to "fill in" the primitive
- At each pixel, keep track of the closest primitive (z-buffer)
Limitations of Scan Conversion

- Restricted to scan-convertible primitives
  - Object polygonization
- Faceting, shading artifacts
- Effective resolution is hardware dependent
- No handling of shadows, reflection, transparency
- Problem of overdraw (high depth complexity)
- What if there are many more triangles than pixels?

Ray Casting vs. Rendering Pipeline

<table>
<thead>
<tr>
<th>Ray Casting</th>
<th>Rendering Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td>For each pixel</td>
<td>For each triangle</td>
</tr>
<tr>
<td>For each object</td>
<td>For each pixel</td>
</tr>
<tr>
<td>Send pixels into the scene</td>
<td>Project scene to the pixels</td>
</tr>
<tr>
<td>Discretize first</td>
<td>Discretize last</td>
</tr>
</tbody>
</table>

“Inverse-Mapping” approach vs. “Forward-Mapping” approach

Questions?

Today

- Ray Casting / Tracing vs. Scan Conversion
- Traditional Graphics Pipeline
- Clipping
- Rasterization/Scan Conversion

The Graphics Pipeline

- **Input:** Geometric model:
  - Description of all object, surface, and light source geometry and transformations
  - Lighting model:
    - Computational description of object and light properties, interaction (reflection)
- **Synthetic Viewpoint or Camera:**
  - Eye position and viewing fraction
  - **Raster Viewport:**
    - Pixel grid onto which image plane is mapped

- **Output:**
  - Colors/Intensities suitable for framebuffer display
    - (e.g. 24-bit RGB value at each pixel)
### The Graphics Pipeline

- Primitives are processed in a series of stages
- Each stage forwards its result on to the next stage
- The pipeline can be drawn and implemented in different ways
- Some stages may be in hardware, others in software
- Optimizations & additional programmability are available at some stages

### Modeling Transformations

- 3D models defined in their own coordinate system (object space)
- Modeling transforms orient the models within a common coordinate frame (world space)

### Illumination (Shading) (Lighting)

- Vertices lit (shaded) according to material properties, surface properties (normal) and light sources
- Local lighting model (Diffuse, Ambient, Phong, etc.)

### Viewing Transformation

- Maps world space to eye space
- Viewing position is transformed to origin & direction is oriented along some axis (usually $z$)

### Clipping

- Transform to Normalized Device Coordinates (NDC)
- Portions of the object outside the view volume (view frustum) are removed

### Projection

- The objects are projected to the 2D image place (screen space)
Scan Conversion (Rasterization)

- Rasterizes objects into pixels
- Interpolate values as we go (color, depth, etc.)

Visibility / Display

- Each pixel remembers the closest object (depth buffer)
- Almost every step in the graphics pipeline involves a change of coordinate system. Transformations are central to understanding 3D computer graphics.

Questions?

Today

- Ray Casting / Tracing vs. Scan Conversion
- Traditional Graphics Pipeline
- Clipping
  - Coordinate Systems
- Rasterization/Scan Conversion

Common Coordinate Systems

- Object space
  - local to each object
- World space
  - common to all objects
- Eye space / Camera space
  - derived from view frustum
- Clip space / Normalized Device Coordinates (NDC)
  - $[-1, -1, -1] \rightarrow [1, 1, 1]$
- Screen space
  - indexed according to hardware attributes

Coordinate Systems in the Pipeline

- Object space
- World space
- Eye Space / Camera Space
- Clip Space (NDC)
- Screen Space
Normalized Device Coordinates

- Clipping is more efficient in a rectangular, axis-aligned volume: \((-1, -1, -1) \rightarrow (1, 1, 1)\) OR \((0, 0, 0) \rightarrow (1, 1, 1)\)

What if the \(p_z > \text{eye}_z\)?

What if the \(p_z < \text{eye}_z\)?

What if the \(p_z \approx \text{eye}_z\)?

Clipping

- Eliminate portions of objects outside the viewing frustum
- View Frustum
  - boundaries of the image plane projected in 3D
  - a near & far clipping plane
- User may define additional clipping planes
Why Clip?

- Avoid degeneracies
  - Don’t draw stuff behind the eye
  - Avoid division by 0 and overflow
- Efficiency
  - Don’t waste time on objects outside the image boundary
- Other graphics applications (often non-convex)
  - Hidden-surface removal, Shadows, Picking, Binning, CSG (Boolean) operations (2D & 3D)

Clipping Strategies

- Don’t clip (and hope for the best)
- Clip on-the-fly during rasterization
- Analytical clipping: alter input geometry

The Graphics Pipeline

- Former hardware relied on full clipping
- Modern hardware mostly avoids clipping
  - Only with respect to plane z=0
- In general, it is useful to learn clipping because it is similar to many geometric algorithms

Questions?

Today

- Ray Casting / Tracing vs. Scan Conversion
- Traditional Graphics Pipeline
- Clipping
  - Rasterization/Scan Conversion
    - Line Rasterization
    - Triangle Rasterization

2D Scan Conversion

- Geometric primitives (point, line, polygon, circle, polyhedron, sphere...)
- Primitives are continuous; screen is discrete
- Scan Conversion: algorithms for efficient generation of the samples comprising this approximation
Scan Converting 2D Line Segments

- Given:
  - Segment endpoints (integers \(x_1, y_1; x_2, y_2\))
- Identify:
  - Set of pixels \((x, y)\) to display for segment

Line Rasterization Requirements

- Transform **continuous** primitive into **discrete** samples
- Uniform thickness & brightness
- Continuous appearance
- No gaps
- Accuracy
- Speed

Algorithm Design Choices

- Assume:
  - \(m = \frac{dy}{dx}, \ 0 < m < 1\)
- Exactly one pixel per column
  - fewer \(\rightarrow\) disconnected, more \(\rightarrow\) too thick

Naive Line Rasterization Algorithm

- Simply compute \(y\) as a function of \(x\)
  - Conceptually: move vertical scan line from \(x_1\) to \(x_2\)
  - What is the expression of \(y\) as function of \(x\)?
  - Set pixel \((x, \text{round}(y(x)))\)

\[
y = y_1 + \frac{x - x_1}{x_2 - x_1} (y_2 - y_1) = y_1 + m(x - x_1)
\]

\[
m = \frac{dy}{dx}
\]

Efficiency

- Computing \(y\) value is expensive
  \(y = y_1 + m(x - x_1)\)
- Observe: \(y = m\) at each \(x\) step (\(m = \frac{dy}{dx}\))

Bresenham's Algorithm (DDA)

- Select pixel vertically closest to line segment
  - intuitive, efficient, pixel center always within 0.5 vertically
- Generalize to handle all eight octants using symmetry
- Can be modified to use only integer arithmetic
Line Rasterization & Grid Marching

- Can be used for ray-casting acceleration
- March a ray through a grid
- Collect all grid cells, not just 1 per column (or row)

Questions?

Brute force solution for triangles

- For each pixel
  - Compute line equations at pixel center
  - “clip” against the triangle

Problem?

Brute force solution for triangles

- For each pixel
  - Compute line equations at pixel center
  - “clip” against the triangle

Problem? If the triangle is small, a lot of useless computation

Brute force solution for triangles

- Improvement: Compute only for the screen bounding box of the triangle
- How do we get such a bounding box?
  - Xmin, Xmax, Ymin, Ymax of the triangle vertices

Can we do better? Kind of!

- We compute the line equation for many useless pixels
- What could we do?
Scan-line Rasterization

• Compute the boundary pixels
• Fill the spans
• Interpolate vertex color along the edges & spans!

But These Days…

• Triangles are usually very small
• Setup cost are becoming more troublesome
• Clipping is annoying
• Brute force is tractable

Modern Rasterization

For every triangle
ComputeProjection
Compute bbox, clip bbox to screen limits
For all pixels in bbox
Compute line equations
If all line equations > 0 //pixel [x,y] in triangle
Framebuffer[x,y] = triangleColor

Questions?

Reading for Today:

• “Ray Tracing on Programmable Graphics Hardware Purcell”, Buck, Mark, & Hanrahan SIGGRAPH 2002

Reading for Tuesday 4/5: