Graphics Pipeline & Rasterization

Image removed due to copyright restrictions.
How Do We Render Interactively?

• Use graphics hardware, via [OpenGL](https://www.khronos.org/opengl/) or [DirectX](https://www.microsoft.com/en-us/software-download/directx)
  – OpenGL is multi-platform, DirectX is MS only

© Khronos Group. All rights reserved. This content is excluded from our Creative Commons license. For more information, see [http://ocw.mit.edu/help/faq-fair-use/](http://ocw.mit.edu/help/faq-fair-use/).
How Do We Render Interactively?

- Use graphics hardware, via **OpenGL** or **DirectX**
  - OpenGL is multi-platform, DirectX is MS only

- Most global effects available in ray tracing will be sacrificed for speed, but some can be approximated
Ray Casting vs. GPUs for Triangles

Ray Casting

For each pixel (ray)
  For each triangle
    Does ray hit triangle?
    Keep closest hit
Ray Casting vs. GPUs for Triangles

**Ray Casting**

For each pixel (ray)

For each triangle

Does ray hit triangle?

Keep closest hit

**GPU**

For each triangle

For each pixel

Does triangle cover pixel?

Keep closest hit
Ray Casting vs. GPUs for Triangles

Ray Casting
For each pixel (ray)
   For each triangle
      Does ray hit triangle?
      Keep closest hit

GPU
For each triangle
   For each pixel
      Does triangle cover pixel?
      Keep closest hit

It’s just a different order of the loops!
GPUs do Rasterization

- The process of taking a triangle and figuring out which pixels it covers is called **rasterization**.
GPUs do Rasterization

- The process of taking a triangle and figuring out which pixels it covers is called **rasterization**
- We’ve seen acceleration structures for ray tracing; rasterization is not stupid either
  - We’re not actually going to test *all* pixels for each triangle
Rasterization (“Scan Conversion”)

- Given a triangle’s vertices & extra info for shading, figure out which pixels to "turn on" to render the primitive
- Compute illumination values to "fill in" the pixels within the primitive
- At each pixel, keep track of the closest primitive (z-buffer)
  - Only overwrite if triangle being drawn is closer than the previous triangle in that pixel

```c
glBegin(GL_TRIANGLES)
  glNormal3f(...)
  glVertex3f(...)
  glVertex3f(...)
  glVertex3f(...)
glEnd();
```
What are the Main Differences?

<table>
<thead>
<tr>
<th>Ray Casting</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>For each pixel (ray)</td>
<td>For each triangle</td>
</tr>
<tr>
<td>For each triangle</td>
<td>For each pixel</td>
</tr>
<tr>
<td>Does ray hit triangle?</td>
<td>Does triangle cover pixel?</td>
</tr>
<tr>
<td>Keep closest hit</td>
<td>Keep closest hit</td>
</tr>
</tbody>
</table>

| Ray-centric | Triangle-centric |

- What needs to be stored in memory in each case?
## What are the Main Differences?

<table>
<thead>
<tr>
<th>Ray Casting</th>
<th>GPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>For each pixel (ray)</td>
<td>For each triangle</td>
</tr>
<tr>
<td>For each triangle</td>
<td>For each pixel</td>
</tr>
<tr>
<td>Does ray hit triangle?</td>
<td>Does triangle cover pixel?</td>
</tr>
<tr>
<td>Keep closest hit</td>
<td>Keep closest hit</td>
</tr>
</tbody>
</table>

- **Ray-centric**
- **Triangle-centric**

- In this basic form, ray tracing needs the entire scene description in memory at once
  - Then, can sample the image completely freely
- The rasterizer only needs one triangle at a time, *plus* the entire image and associated depth information for all pixels
Rasterization Advantages

• Modern scenes are more complicated than images
  – A 1920x1080 frame at 64-bit color and 32-bit depth per pixel is 24MB (not that much)
    • Of course, if we have more than one sample per pixel this gets larger, but e.g. 4x supersampling is still a relatively comfortable \( \sim 100\)MB
  – Our scenes are routinely larger than this
    • This wasn’t always true
Rasterization Advantages

Weiler, Atherton 1977
Rasterization Advantages

- Modern scenes are more complicated than images
  - A 1920x1080 frame (1080p) at 64-bit color and 32-bit depth per pixel is 24MB (not that much)
    - Of course, if we have more than one sample per pixel (later) this gets larger, but e.g. 4x supersampling is still a relatively comfortable ~100MB
  - Our scenes are routinely larger than this
    - This wasn’t always true

- A rasterization-based renderer can stream over the triangles, no need to keep entire dataset around
  - Allows parallelism and optimization of memory systems
Rasterization Limitations

- Restricted to scan-convertible primitives
  - Pretty much: triangles
- Faceting, shading artifacts
  - This is largely going away with programmable per-pixel shading, though
- No unified handling of shadows, reflection, transparency
- Potential problem of overdraw (high depth complexity)
  - Each pixel touched many times
Ray Casting / Tracing

• Advantages
  – Generality: can render anything that can be intersected with a ray
  – Easily allows recursion (shadows, reflections, etc.)

• Disadvantages
  – Hard to implement in hardware (lacks computation coherence, must fit entire scene in memory, bad memory behavior)
    • Not such a big point any more given general purpose GPUs
  – Has traditionally been too slow for interactive applications
  – Both of the above are changing rather rapidly right now!
Image removed due to copyright restrictions.
Modern Graphics Pipeline

• Input
  – Geometric model
    • Triangle vertices, vertex normals, texture coordinates
  – Lighting/material model (shader)
    • Light source positions, colors, intensities, etc.
    • Texture maps, specular/diffuse coefficients, etc.
  – Viewpoint + projection plane

• Output
  – Color (+depth) per pixel
Modern Graphics Pipeline

• Project vertices to 2D (image)

• Rasterize triangle: find which pixels should be lit

• Test visibility (Z-buffer), update frame buffer color

• Compute per-pixel color
Modern Graphics Pipeline

• Project vertices to 2D (image)

• Rasterize triangle: find which pixels should be lit
  – For each pixel, test 3 edge equations
    • if all pass, draw pixel

• Compute per-pixel color
• Test visibility (Z-buffer), update frame buffer color
• Perform projection of vertices
• Rasterize triangle: find which pixels should be lit
• Compute per-pixel color
• Test visibility, update frame buffer color
  – Store minimum distance to camera for each pixel in “Z-buffer”
    • ~same as \( t_{\text{min}} \) in ray casting!
  – if \( \text{newz} < \text{zbuffer}[x,y] \)
    \[
    \text{zbuffer}[x,y] = \text{new}_z \\
    \text{framebuffer}[x,y] = \text{new}_\text{color}
    \]
For each triangle
transform into eye space
(perform projection)
setup 3 edge equations
for each pixel x,y
if passes all edge equations
compute z
if z<zbuffer[x,y]
   zbuffer[x,y]=z
framebuffer[x,y]=shade()
Modern Graphics Pipeline

For each triangle
transform into eye space
(perform projection)
setup 3 edge equations
for each pixel x,y
if passes all edge equations
compute z
if z<zbuffer[x,y]
   zbuffer[x,y]=z
   framebuffer[x,y]=shade()

Questions?
Modern Graphics Pipeline

• Project vertices to 2D (image)

• Rasterize triangle: find which pixels should be lit

• Compute per-pixel color

• Test visibility (Z-buffer), update frame buffer
Projection

• Project vertices to 2D (image)

• Rasterize triangle: find which pixels should be lit

• Compute per-pixel color

• Test visibility (Z-buffer), update frame buffer
Orthographic vs. Perspective

- Orthographic

- Perspective
Perspective in 2D

This image is in the public domain. Source: openclipart
Perspective in 2D

The projected point in homogeneous coordinates (we just added $w=1$):

$$p' = \begin{pmatrix} x/z \\ 1 \\ 1 \end{pmatrix}$$
Perspective in 2D

\[ p' = \begin{pmatrix} x/z \\ 1 \\ 1 \end{pmatrix} \propto \begin{pmatrix} x \\ z \\ z \end{pmatrix} \]

Projectively equivalent

This image is in the public domain. Source: openclipart
Perspective in 2D

We’ll just copy z to w, and get the projected point after homogenization!
Extension to 3D

- Trivial: Just add another dimension $y$ and treat it like $x$
- Different fields of view and non-square image aspect ratios can be accomplished by simple scaling of the $x$ and $y$ axes.

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

- These projections matrices work perfectly in the sense that you get the proper 2D projections of 3D points.
- However, since we are flattening the scene onto the $z=1$ plane, we’ve lost all information about the distance to camera.
  - We need the distance for Z buffering, i.e., figuring out what is in front of what!
Basic Idea: store $1/z$

\[
\begin{pmatrix}
    x' \\ y' \\ z' \\ w'
\end{pmatrix} = \begin{pmatrix}
    1 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 0 & 0 & 1 \\
    0 & 0 & 1 & 0
\end{pmatrix} \begin{pmatrix}
    x \\
    y \\
    z \\
    1
\end{pmatrix}
\]
Basic Idea: store $1/z$

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

$$\begin{pmatrix} x' \\ y' \\ z' \\ w' \end{pmatrix} = \begin{pmatrix} x \\ y \\ 1 \\ z \end{pmatrix}$$

- $z' = 1$ before homogenization
- $z' = 1/z$ after homogenization
Full Idea: Remap the View Frustum

- We can transform the frustum by a modified projection in a way that makes it a square (cube in 3D) after division by $w'$. 
The View Frustum in 2D

- We can transform the frustum by a modified projection in a way that makes it a square (cube in 3D) after division by $w'$. 

The final image is obtained by merely dropping the $z$ coordinate after projection (orthogonal projection).
The View Frustum in 2D

- (In 3D this is a truncated pyramid.)
The View Frustum in 2D

- Far and near are kind of arbitrary
- They bound the depth storage precision

\[ \text{image } \texttt{xmin} \quad \text{image } \texttt{xmax} \]
The Canonical View Volume

- Point of the exercise: This gives screen coordinates and depth values for Z-buffering with unified math
  - Caveat: OpenGL and DirectX define Z differently [0,1] vs. [-1,1]
OpenGL Form of the Projection

\[
\begin{pmatrix}
    x' \\
    y' \\
    z' \\
    w'
\end{pmatrix} =
\begin{pmatrix}
    1 & 0 & 0 & 0 \\
    0 & 1 & 0 & 0 \\
    0 & 0 & \frac{\text{far}+\text{near}}{\text{far}-\text{near}} & 0 \\
    0 & 0 & \frac{2*\text{far}*\text{near}}{\text{far}-\text{near}} & 1
\end{pmatrix}
\begin{pmatrix}
    x \\
    y \\
    z \\
    1
\end{pmatrix}
\]

Homogeneous coordinates within canonical view volume

Input point in view coordinates
OpenGL Form of the Projection

\[
\begin{pmatrix}
  x' \\
y' \\
z' \\
w'
\end{pmatrix} =
\begin{pmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & \frac{\text{far} + \text{near}}{\text{far} - \text{near}} & \frac{2 \times \text{far} \times \text{near}}{\text{far} - \text{near}} \\
  0 & 0 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
  x \\
y \\
z \\
1
\end{pmatrix}
\]

- \( z' = \frac{(az+b)}{z} = a + \frac{b}{z} \)
  - where \( a \) & \( b \) depend on near & far

- Similar enough to our basic idea:
  - \( z' = \frac{1}{z} \)

\[
\begin{pmatrix}
x' \\
y' \\
z' \\
w'
\end{pmatrix} =
\begin{pmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & 0 & 1 \\
  0 & 0 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
x \\
y \\
z \\
1
\end{pmatrix}
\]
OpenGL Form of the Projection

\[
\begin{pmatrix}
  x' \\
  y' \\
  z' \\
  w'
\end{pmatrix}
= \begin{pmatrix}
  1 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 \\
  0 & 0 & \frac{\text{far}+\text{near}}{\text{far}−\text{near}} & \frac{-2*\text{far}*\text{near}}{\text{far}−\text{near}} \\
  0 & 0 & 1 & 0
\end{pmatrix}
\begin{pmatrix}
  x \\
  y \\
  z \\
  1
\end{pmatrix}
\]

- Details/more intuition in handout
  - “Understanding Projections and Homogenous Coordinates”
Recap: Projection

- Perform rotation/translation/other transforms to put viewpoint at origin and view direction along z axis
  - This is the OpenGL “modelview” matrix

- Combine with projection matrix (perspective or orthographic)
  - Homogenization achieves foreshortening
  - This is the OpenGL “projection” matrix

- **Corollary**: The entire transform from object space to canonical view volume $[-1,1]^3$ is a single matrix
Recap: Projection

• Perform rotation/translation/other transforms to put viewpoint at origin and view direction along z axis
  – This is the OpenGL “modelview” matrix

• Combine with projection matrix (perspective or orthographic)
  – Homogenization achieves foreshortening
  – This is the OpenGL “projection” matrix

• **Corollary:** The entire transform from object space to canonical view volume \([-1,1]^3\) is a single matrix
Modern Graphics Pipeline

- Project vertices to 2D (image)
  - We now have screen coordinates
- Rasterize triangle: find which pixels should be lit
- Compute per-pixel color
- Test visibility (Z-buffer), update frame buffer
2D Scan Conversion

- Primitives are “continuous” geometric objects; screen is discrete (pixels)
2D Scan Conversion

- Primitives are “continuous” geometric objects; screen is discrete (pixels)
- Rasterization computes a discrete approximation in terms of pixels (how?)
Edge Functions

- The triangle’s 3D edges project to line segments in the image (thanks to planar perspective)
  - Lines map to lines, not curves
Edge Functions

- The triangle’s 3D edges project to line segments in the image (thanks to planar perspective)
- The interior of the triangle is the set of points that is inside all three halfspaces defined by these lines
Edge Functions

- The triangle’s 3D edges project to line segments in the image (thanks to planar perspective)
- The interior of the triangle is the set of points that is inside all three halfspaces defined by these lines

\[
E_i(x, y) = a_i x + b_i y + c_i
\]

\[(x, y) \text{ within triangle } \iff E_i(x, y) \geq 0, \quad \forall i = 1, 2, 3\]
Brute Force Rasterizer

- Compute $E_1$, $E_2$, $E_3$ coefficients from projected vertices
  - Called “triangle setup”, yields $a_i$, $b_i$, $c_i$ for $i=1,2,3$
Brute Force Rasterizer

• Compute $E_1, E_2, E_3$ coefficients from projected vertices

• For each pixel $(x, y)$
  – Evaluate edge functions at pixel center
  – If all non-negative, pixel is in!

Problem?
Brute Force Rasterizer

- Compute $E_1, E_2, E_3$ coefficients from projected vertices
- For each pixel $(x, y)$
  - Evaluate edge functions at pixel center
  - If all non-negative, pixel is in!

If the triangle is small, lots of useless computation if we really test all pixels
Easy Optimization

• Improvement: Scan over only the pixels that overlap the *screen bounding box* of the triangle

• How do we get such a bounding box?
  – $X_{\text{min}}, X_{\text{max}}, Y_{\text{min}}, Y_{\text{max}}$ of the projected triangle vertices
Rasterization Pseudocode

For every triangle
  Compute projection for vertices, compute the $E_i$
  Compute bbox, clip bbox to screen limits
For all pixels in bbox
  Evaluate edge functions $E_i$
  If all $> 0$
    $\text{Framebuffer}[x, y] = \text{triangleColor}$

Bounding box clipping is easy, just clamp the coordinates to the screen rectangle.

Note: No visibility
For every triangle
Compute projection for vertices, compute the $E_i$
Compute bbox, clip bbox to screen limits
For all pixels in bbox
Evaluate edge functions $E_i$
If all $> 0$
\[ \text{Framebuffer}[x,y] = \text{triangleColor} \]

Bounding box clipping is easy, just clamp the coordinates to the screen rectangle

Questions?
Can We Do Better?

For every triangle

Compute projection for vertices, compute the $E_i$
Compute bbox, clip bbox to screen limits
For all pixels in bbox

Evaluate edge functions $a_i x + b_i y + c_i$
If all $> 0$

$$\text{Framebuffer}[x, y] = \text{triangleColor}$$
Can We Do Better?

For every triangle
Compute projection for vertices, compute the $E_i$
Compute bbox, clip bbox to screen limits
For all pixels in bbox

Evaluate edge functions $a_i x + b_i y + c_i$
If all $> 0$

Framebuffer[$x, y$] = triangleColor

These are linear functions of the pixel coordinates $(x,y)$, i.e., they only change by a constant amount when we step from $x$ to $x+1$ (resp. $y$ to $y+1$)
Incremental Edge Functions

For every triangle

  ComputeProjection

  Compute bbox, clip bbox to screen limits

  For all scanlines y in bbox

    Evaluate all $E_i$'s at $(x_0, y)$: $E_i = a_i x_0 + b_i y + c_i$

    For all pixels x in bbox

      If all $E_i > 0$

        $\text{Framebuffer}[x, y] = \text{triangleColor}$

        Increment line equations: $E_i += a_i$

• We save ~two multiplications and two additions per pixel when the triangle is large
Incremental Edge Functions

For every triangle
Compute Projection
Compute bbox, clip bbox to screen limits
For all scanlines y in bbox
   Evaluate all $E_i$'s at $(x_0, y)$: $E_i = a_i x_0 + b_i y + c_i$
   For all pixels x in bbox
       If all $E_i > 0$
           Framebuffer[x, y] = triangleColor
       Increment line equations: $E_i + = a_i$

• We save ~two multiplications and two additions per pixel when the triangle is large

Can also zig-zag to avoid reinitialization per scanline, just initialize once at x0, y0
• For a really HC piece of rasterizer engineering, see the hierarchical Hilbert curve rasterizer by McCool, Wales and Moule.
  – (Hierarchical? We’ll look at that next..)
Can We Do Even Better?

- We compute the line equation for many useless pixels
- What could we do?
Indeed, We Can Be Smarter
Indeed, We Can Be Smarter

• Hierarchical rasterization!
  – Conservatively test **blocks of pixels** before going to per-pixel level (can skip large blocks at once)
  – Usually two levels

Indeed, We Can Be Smarter

• Hierarchical rasterization!
  – Conservatively test **blocks of pixels** before going to per-pixel level (can skip large blocks at once)
  – Usually two levels

Can also test if an entire block is *inside* the triangle; then, can skip edge functions tests for all pixels for even further speedups. (Must still test Z, because they might still be occluded.)
Further References


Oldschool Rasterization

- Compute the boundary pixels using line rasterization
Oldschool Rasterization

- Compute the boundary pixels using line rasterization
- Fill the spans
Oldschool Rasterization

• Compute the boundary pixels using line rasterization
• Fill the spans

More annoying to implement than edge functions
Not faster unless triangles are huge
Oldschool Rasterization

- Compute the boundary pixels using line rasterization
- Fill the spans

More annoying to implement than edge functions

Not faster unless triangles are huge

Questions?
What if the $p_z$ is $> \text{eye}_z$?
What if the $p_z$ is $< \text{eye}_z$?
What if the $p_z = \text{eye}_z$?

When $w' = 0$, point projects to infinity (homogenization is division by $w'$).
A Solution: Clipping

"clip" geometry to view frustum, discard outside parts

(image plane)

(eye_x, eye_y, eye_z)

z axis

z=near

z=far

image plane
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

Leonard McMillan, Computer Science at the University of North Carolina in Chapel Hill.
Why Clip?

• Avoid degeneracies
  – Don’t draw stuff behind the eye
  – Avoid division by 0 and overflow
Related Idea

• “View Frustum Culling”
  – Use bounding volumes/hierarchies to test whether any part of an object is within the view frustum
    • Need “frustum vs. bounding volume” intersection test
    • Crucial to do hierarchically when scene has *lots* of objects!
    • Early rejection (different from clipping)

Related Idea

• “View Frustum Culling”
  – Use bounding volumes/hierarchies to test whether any part of an object is within the view frustum
    • Need “frustum vs. bounding volume” intersection test
    • Crucial to do hierarchically when scene has *lots* of objects!
    • Early rejection (different from clipping)

See e.g. Optimized view frustum culling algorithms for bounding boxes, Ulf Assarsson and Tomas Möller, journal of graphics tools, 2000.
Homogeneous Rasterization

• Idea: avoid projection (and division by zero) by performing rasterization in 3D
  – Or equivalently, use 2D homogenous coordinates
    \((w'=z\) after the projection matrix, remember)

• Motivation: clipping is annoying

Homogeneous Rasterization
Homogeneous Rasterization

- Replace 2D edge equation by 3D plane equation
  - Plane going through 3D edge and viewpoint
  - Still a halfspace, just 3D

2D rasterization

3D (homogenous) rasterization
Homogeneous Rasterization

• Replace 2D edge equation by 3D plane equation
  – Treat pixels as 3D points \((x, y, 1)\) on image plane, test for containment in 3 halfspaces just like edge functions

2D rasterization

3D (homogenous) rasterization
Homogeneous Rasterization

Given 3D triangle
   setup plane equations
   (plane through viewpoint & triangle edge)
For each pixel x, y
   compute plane equations for (x, y, 1)
   if all pass, draw pixel
Homogeneous Rasterization

- Works for triangles behind eye
- Still linear, can evaluate incrementally/hierarchically like 2D
Homogeneous Rasterization Recap

- Rasterizes with plane tests instead of edge tests
- Removes the need for clipping!
Homogeneous Rasterization Recap

- Rasterizes with plane tests instead of edge tests
- Removes the need for clipping!

Questions?
Modern Graphics Pipeline

• Perform projection of vertices

• Rasterize triangle: find which pixels should be lit

• Compute per-pixel color

• Test visibility, update frame buffer
Pixel Shaders

- Modern graphics hardware enables the execution of rather complex programs to compute the color of every single pixel
- More later
Modern Graphics Pipeline

- Perform projection of vertices
- Rasterize triangle: find which pixels should be lit
- Compute per-pixel color
- Test visibility, update frame buffer
Visibility

- How do we know which parts are visible/in front?
Ray Casting

- Maintain intersection with closest object
Visibility

- In ray casting, use intersection with closest $t$
- Now we have swapped the loops (pixel, object)
- What do we do?
Z buffer

- In addition to frame buffer (R, G, B)
- Store distance to camera (z-buffer)
- Pixel is updated only if newz is closer than z-buffer value
For every triangle
  Compute Projection, color at vertices
  Setup line equations
  Compute bbox, clip bbox to screen limits
  For all pixels in bbox
    Increment line equations
    \textbf{Compute currentZ}
    Compute currentColor
    If all line equations > 0 \hspace{1em} //pixel \([x,y]\) in triangle
      \textbf{If currentZ < zBuffer}[x,y] \hspace{1em} //pixel is visible
      Framebuffer[x,y] = currentColor
      zBuffer[x,y] = currentZ
Works for hard cases!
More questions for next time

- How do we get $Z$?
- Texture Mapping?
That's All For Today!

- Next time:
  Screen-space interpolation, visibility, shading

Screenshot from the video game Uncharted 2 has been removed due to copyright restrictions.