Calvin and Hobbes

AHHHH...

Uh-oh, something is seriously wrong here.

The laws of perspective have been repealed!

Objects no longer diminish in size with distance.

Lines do not converge toward any point on the horizon!

All spatial relationships are lost. It's impossible to judge where anything is! Oh no!

Calvin, quit running around and crashing into things, or I'll sell you to the monkey house!

...And now she's lost perspective.
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

MIT EECS 6.837, Durand
Scan Conversion / Rasterization

- Given a triangle’s vertices & extra info for shading:
- Figure out which pixels to "turn on" to render the primitive
- Compute the illumination values to "fill in" the primitive
- At each pixel, keep track of the closest primitive (z-buffer)

```c
glBegin(GL_TRIANGLES)
glNormal3f(...)
glVertex3f(...)
glVertex3f(...)
glVertex3f(...)
glEnd();
```
Ray Casting vs. Rendering Pipeline

Ray Casting
For each pixel
For each object
Send pixels to the scene

Rendering Pipeline
For each triangle
For each pixel
Project scene to the pixels

"Inverse-Mapping" approach
For each pixel on the screen go through the display list

"Forward-Mapping" approach to Computer Graphics

MIT EECS 6.837, Durand
Modern Graphics Pipeline

- **Input:**
  - Geometric model
    - Triangle vertices
  - Lighting model
    - Light source position, color,
    - Normals, colors, specular coefficient, texture coordinates, etc.
  - Viewpoint + projection plane

- **Output**
  - Color per pixel
Modern Graphics Pipeline

• Perform projection of vertices

• Rasterize triangle: find which pixels should be lit

• Compute per-pixel color

• Test visibility, update frame buffer color
Modern Graphics Pipeline

• Perform projection of vertices

• 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, update frame buffer color
Modern Graphics Pipeline

- Perform projection of vertices
- Rasterize triangle: find which pixels should be lit
- Compute per-pixel color
- Test visibility, update frame buffer color
  - store min distance to camera for each pixel in z buffer
    - ~same as tmin in ray casting
  - if new \(_z\) < zbuffer\([x,y]\)
    - zbuffer\([x,y]\) = new \(_z\)
    - framebuffer\([x,y]\) = new_color
Modern Graphics Pipeline

For each triangle
  transform into eye space
  (perform projection)
  setup 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?
Ray Casting vs. Rendering Pipeline

Ray Casting

For each pixel
  For each triangle
• Ray-centric
• Needs to store scene in memory

Rendering Pipeline

For each triangle
  For each pixel
• Triangle centric
• Needs to store image (and depth) into memory
Ray Casting / Tracing

• Advantages?
  – Generality: can render anything that can be intersected with a ray
  – Atomic operation, allows recursion (shadow, reflection, etc.)

• Disadvantages?
  – Hard to implement in hardware (lacks computation coherence, must fit entire scene in memory, bad memory behavior)
  – Usually too slow for interactive applications
    • But some real-time implementations exist
Limitations of Scan Conversion

- Restricted to scan-convertible primitives
  - Pretty much triangles
- Faceting, shading artifacts
- No direct handling of shadows, reflection, transparency
- Potential problem of overdraw (high depth complexity)
- What if there are many more triangles than pixels?
Questions?
Modern vs. older graphics pipeline

- The core graphics pipeline has been simplified
  - more resources, different tradeoffs
  - in particular smaller triangles, more of the computation/memory cost goes into shading

- I will describe some of the older ways to motivate the current techniques
Modern Graphics Pipeline

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

• Perform projection of vertices

• Rasterize triangle: find which pixels should be lit

• Compute per-pixel color

• Test visibility, update frame buffer
Simple Perspective Projection

• Project all points to the $z = d$ plane, eyepoint at the origin:

$$x_p = \frac{d \cdot x}{z} = \frac{x}{z/d}$$

$$y_p = \frac{d \cdot y}{z} = \frac{y}{z/d}$$

$$z_p = d$$

$$\begin{pmatrix}
  x \\
  y \\
  z \\
  1
\end{pmatrix}
\begin{pmatrix}
  d \\
  z
\end{pmatrix}
\begin{pmatrix}
  1 & 0 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 & 0 \\
  0 & 0 & 1 & 0 & 0 \\
  0 & 0 & \frac{1}{d} & 0 & 0 \\
  0 & 0 & 0 & 1 & 1
\end{pmatrix}
\begin{pmatrix}
  x \\
  y \\
  z \\
  1
\end{pmatrix}$$
In the limit, as $d \to \infty$

This perspective projection matrix...

\[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1/d & 1 & 0
\end{pmatrix}
\]

\[\rightarrow\]

...is simply an orthographic projection

\[
\begin{pmatrix}
1 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0
\end{pmatrix}
\]
Recap: projection

- Perform rotation/translation to put viewpoint at origin and view direction along z axis
- Combine with simple projection matrix (perspective or orthographic)
  - Homogenization achieves foreshortening
  - This step can be optional/virtual, see later.
Questions?
Modern Graphics Pipeline

• Perform projection of vertices
  – We now have screen coord

• Rasterize triangle: find which pixels should be lit

• Compute per-pixel color

• Test visibility, update frame buffer
2D Scan Conversion

- Primitives are continuous; screen is discrete
  - Well, triangles are described by a discrete set of vertices
  - But they describe a continuous area on screen
2D Scan Conversion

- Solution: compute discrete approximation
- Scan Conversion: algorithms for efficient generation of the samples comprising this approximation
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 \textit{screen bounding box} of the triangle

• How do we get such a bounding box?
  – Xmin, Xmax, Ymin, Ymax of the triangle vertices
Questions?
For modern graphics cards

• 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
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

• Note that Bbox clipping is trivial, unlike triangle clipping
Can we do better?

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
Can we do better?

For every triangle

ComputeProjection

Compute bbox, clip bbox to screen limits

For all pixels in bbox

**Compute line equations** $ax+by+c$

If all line equations $>0$  //pixel [x,y] in triangle

Framebuffer[x,y] = triangleColor

- We don’t need to recompute line equation from scratch
Can we do better?

For every triangle
  ComputeProjection
  Compute bbox, clip bbox to screen limits
  Setup line eq
    compute $a_i dx, b_i dy$ for the 3 lines
  Initialize line eq, values for bbox corner
    $L_i = a_i x_0 + b_i y + c_i$
  For all scanline $y$ in bbox
    For 3 lines, update $L_i$
      For all $x$ in bbox
        Increment line equations: $L_i += adx$
        If all $L_i > 0$ \(/\text{pixel } [x,y] \text{ in triangle}\)
          Framebuffer$[x,y] = \text{triangleColor}$

• We save one multiplication per pixel
Can we do better?

- We compute the line equation for many useless pixels
- What could we do?
Can we do better? Kind of!

- Hierarchical rasterization
  - Usually two-level
  - Then it’s just a question of finding the right granularity
In the modern hardware

• Edge eq. in homogeneous coordinates \([x, y, w]\)
• Tiles to add a mid-level granularity
  – Early rejection of tiles
  – Memory access coherence
Ref


- Triangle Scan Conversion using 2D Homogeneous Coordinates, Marc Olano Trey Greer
Old-style rasterization

- Compute the boundary pixels (line rasterization)

Shirley page 55
Scan-line Rasterization

- Compute the boundary pixels
- Fill the spans

Shirley page 55
Scan-line Rasterization

- Requires some initial setup to prepare

Shirley page 55
Take-home message

• The appropriate algorithm depends on
  – Balance between various resources (CPU, memory, bandwidth)
  – The input (size of triangles, etc.)

• Smart algorithms often have initial preprocess
  – Assess whether it is worth it

• To save time, identify redundant computation
  – Put outside the loop and interpolate if needed
What if the $p_z$ is $> \text{eye}_z$?
What if the $p_z$ is $< \text{eye}_z$?
What if the $p_z = \text{eye}_z$?
Old solution: Clipping

"clip" geometry to view frustum

(e_{\text{eye}, x}, e_{\text{eye}, y}, e_{\text{eye}, z})

z axis

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
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
  - This one is still valid: “view frustum culling”
    - but a little different: not a strict requirement to eliminate everything behind the eye
    - makes life easier
    - use usual bounding volume hierarchy kind of trick
Questions?
Homogeneous Rasterization

- Motivation: clipping is annoying
- Idea: avoid projection (and division by zero) by performing rasterization in 3D
  - or equivalently in 2D homogenous coordinates
Homogenous rasterization

- Replace 2D edge equation by 3D plane equation
- Plane going through 3D edge and viewpoint
  => no need to project to 2D
  => No need to clip
Homogenous 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
Homogenous Rasterization

- Works for triangles behind eye!
Recap homogenous rasterization

- Rasterizes with plane tests instead of edge tests
- Alleviates 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

Translucence
Backlighting

Procedural texture,
Anisotropic brdf

Iridescence
Phong shader example

• Global light parameters: position, intensity, etc.
  – passed as parameter to shader
• Per pixel: light direction, view, normal, color
  – might need to be interpolated with barycentric coordinates (e.g. per vertex color, per-vertex normal)
Texture mapping needs coordinates

- Texture mapping is a crucial feature of modern graphics hardware
- When rendering a triangle, we need to interpolate uv texture coordinates from vertices per pixel
Interpolation

- We need to interpolate color, normals, texture coordinates, etc. between vertices.
- We can do linear interpolation in screen space or object space. Is it the same?

\[
[R_0, G_0, B_0] \quad \rightarrow \quad [R_1, G_1, B_1]
\]
Gouraud interpolation

- Gouraud: interpolate linearly in screen space
- Not same as object-space interpolation
- Would mean that the physical color/texture of a 3D point depends on viewpoint

\[ [R_0, G_0, B_0] \]

\[ [R_1, G_1, B_1] \]
Perspective-correct interpolation

- Quantity (e.g. R) is linear wrt e.g. z: $R = az + b$
- Interpolate R like we interpolate z
  - See in a few slides
  - hyperbolic
Recap

- Need to interpolate various values provided at vertices (color, texture coordinate, normal) across pixels
- Cannot use linear interpolation in screen space
- Hyperbolic interpolation as a function of screen coordinates
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

MIT EECS 6.837, Durand
Visibility

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

- Maintain intersection with closest object
Painter’s algorithm

• Draw back-to-front
• How do we sort objects?
• Can we always sort objects?
Painter’s algorithm

• Draw back-to-front
• How do we sort objects?
• Can we always sort objects?
  – No, there can be cycles
  – Requires to split polygons
Painter’s algorithm

• Old solution for hidden-surface removal
  – Good because ordering is useful for other operations (transparency, antialiasing)

• But
  – Ordering is tough
  – Cycles
  – Must be done by CPU

• Hardly used now

• But some sort of partial ordering is sometimes useful
  – Usually front-to-back
  – To make sure foreground is rendered first
  – For transparency
Visibility

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

- In addition to frame buffer (R, G, B)
- Store distance to camera (z-buffer)
- Pixel is updated only if new z is closer than z-buffer value
Z-buffer pseudo code

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

    Compute currentZ

    Increment currentColor

    If all line equations > 0  //pixel [x,y] in triangle
      If currentZ < zBuffer[x,y]  //pixel is visible
         Framebuffer[x,y] = currentColor
         zBuffer[x,y] = currentZ
Works for hard cases!
What exactly do we store

- Floating point distance
- Can we interpolate $z$ in screen space?
  - i.e. does $z$ vary linearly in screen space?
Z interpolation

- \( x' = \frac{x}{z} \) [and \( x = az + b \)]
- Hyperbolic variation
- \( Z \) cannot be linearly interpolated
Simple Perspective Projection

- Project all points along the $z$ axis to the $z = d$ plane, eyepoint at the origin

$$\begin{pmatrix} x \ast \frac{d}{z} \\ y \ast \frac{d}{z} \\ d \\ 1 \end{pmatrix} = \begin{pmatrix} x \\ y \\ z \\ \frac{z}{d} \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1/d & 0 & 0 \end{pmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix}$$

\text{homogenize}
Yet another Perspective Projection

- Change the $z$ component
- Compute $d/z$
- Can be linearly interpolated

\[
\begin{pmatrix}
  x \\
  y \\
  d/z \\
  1
\end{pmatrix} =
\begin{pmatrix}
  x \\
  y \\
  l \\
  z/d
\end{pmatrix} =
\begin{pmatrix}
  1 & 0 & 0 & 0 & 0 \\
  0 & 1 & 0 & 0 & 0 \\
  0 & 0 & 0 & 0 & 1 \\
  0 & 0 & 1/d & 0 & 0
\end{pmatrix}
\begin{pmatrix}
  x \\
  y \\
  z \\
  1
\end{pmatrix}
\]
Advantages of $1/z$

- Can be interpolated linearly in screen space
- Puts more precision for close objects
- Useful when using integers
  - more precision where perceptible
Old days: Integer z-buffer

- Use $1/z$ to have more precision in the foreground
- Set a near and far plane
  - $1/z$ values linearly encoded between $1/near$ and $1/far$
- Careful, test direction is reversed
Modern days: 32-bit float z buffer

- Just store z, the float precision will do the rest
  - floats have more precision for small values
Buster's representation of Van Gogh's *Sunflowers* is a good example of invertism. From a purely visual perspective, the brown mark at the top of the work clearly represents the dark line which defines the edge of the table and the bottom of the vase, as shown in the photograph (left), while the blue marks represent the flowers. However, biologists interpret these blue marks as territorial and similar in function to the arrowhead paw marks cats make to demarcate their feces. In the painting these marks signify ownership of the inverted object and are thought to have the function of rendering its unfamiliarity 'safe.'
The infamous half pixel

• I refuse to teach it, but it’s an annoying issue you should know about

• Do a line drawing of a rectangle from [top, right] to [bottom, left]

• Do we actually draw the columns/rows of pixels?
The infamous half pixel

- Displace by half a pixel so that top, right, bottom, left are in the middle of pixels
- Just change the viewport transform
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 frustum
- Raster Viewport:
  - Pixel grid onto which image plane is mapped

**Output:**
- Colors/Intensities suitable for framebuffer display (For example, 24-bit RGB value at each pixel)
Modern Graphics Hardware

- Geometry (G)
- Rasterization (R)
- Texture (T)
- Fragment (F)
- Display (D)

MIT EECS 6.837, Durand
Graphics Hardware

- High performance through
  - Parallelism
  - Specialization
  - No data dependency
  - Efficient pre-fetching
Programmable Graphics Hardware

- Geometry and pixel (fragment) stage become programmable
  - Elaborate appearance
  - More and more general-purpose computation (GPU hacking)
Modern Graphics Hardware

• About 4-6 geometry units
• About 16 fragment units
• Deep pipeline (~800 stages)
• Tiling (about 4x4)
  – Early z-rejection if entire tile is occluded
• Pixels rasterized by quads (2x2 pixels)
  – Allows for derivatives
• Very efficient texture pre-fetching
  – And smart memory layout
Current GPUs

- Programmable geometry and fragment stages
- 600 million vertices/second, 6 billion texels/second
- In the range of tera operations/second
- Floating point operations only
- Very little cache
Computational Requirements

- 20 Mvert/s
- 1000 Mpix/s
- 120 Mpix/s

Flow chart with data rates:
- Application: 0.880 GB/s
- Command
- Geometry
- Rasterization: 4 GB/s
- Texture
- Fragment: 150 Gops
- Display: 0.36 GB/s
- Texture Memory
- Framebuffer: 16 GB/s

Rough estimate:
- Vertex: 5 Gops
- Fragment

[Reference: Akeley, Hanrahan]

MIT EECS 6.837, Durand
Questions?

Above:

Above:
Clyde interacts with his sister’s sculpture, allowing his whole body to become implicated in its heavily nuanced form.

Above:
Interpretive diagram by Peter Muxlow:
1. Tail form.
2. Erogenous edging.
3. Ovoidal aperture.
4. Restrictive vine forms.
“The synthetic fiber has been carefully frayed to resemble the texture and color of a cat’s tail in the upright welcoming position—inviting, yet guarding the entrance beyond. However, this controlling tail is itself compromised by restrictive vines so that the whole erogenously edged aperture hints at pleasure tinged with the possibility of entanglement.”

Ray Casting vs. Rendering Pipeline

Ray Casting

For each pixel
  For each object
    • Ray-centric
    • Needs to store scene in memory
    • (Mostly) Random access to scene

Rendering Pipeline

For each triangle
  For each pixel
    • triangle centric
    • Needs to store image (and depth) into memory
    • (Mostly) random access to frame buffer

Which is smaller? Scene or Frame?
  Frame

Which is easiest to access randomly?
  Frame because regular sampling

MIT EECS 6.837, Durand
### Ray Casting vs. Rendering Pipeline

#### Ray Casting

**For each pixel**

- Whole scene must be in memory
- Needs spatial acceleration to be efficient

**For each object**

+ Depth complexity: no computation for hidden parts
+ Atomic computation
+ More general, more flexible
  - Primitives, lighting effects, adaptive antialiasing

#### Rendering Pipeline

**For each triangle**

- Harder to get global illumination
- Needs smarter techniques to address depth complexity (overdraw)

**For each pixel**

+ Primitives processed one at a time
+ Coherence: geometric transforms for vertices only
+ Good bandwidth/computation ratio
+ Minimal state required, good memory behavior
Movies

both pipeline and ray tracing
Games
Simulation
(painter for a long time)
CAD-CAM & Design

pipeline during design, anything for final image
Architecture

ray-tracing, pipeline with preprocessing for complex lighting

MIT EECS 6.837, Durand
Virtual Reality

pipeline
Visualization

mostly pipeline, ray-tracing for high-quality eye candy, interactive ray-tracing is starting