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The laws of perspective have been repealed!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.
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

---

"Inverse-Mapping" approach

Scene primitives

"Forward-Mapping" approach

Pixel raster

Scene primitives

MIT EECS 6.837 – Durand

Tuesday, November 30, 2010
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 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 new$_z < z_{\text{buffer}}[x,y]$
    $z_{\text{buffer}}[x,y] = \text{new}_z$
    $\text{framebuffer}[x,y] = \text{new}_\text{color}$

Modern Graphics Pipeline
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 $z<\text{zbuffer}[x,y]$
      $\text{zbuffer}[x,y]=z$
      $\text{framebuffer}[x,y]=\text{shade()}$
Questions?
Interpolation in Screen Space

• How do we get that Z value for each pixel?
  – We only know z at the vertices...
  – (Remember, screen-space z is actually z’/w’)
  – Must interpolate from vertices into triangle interior

For each triangle
  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()
Interpolation in Screen Space

• Also need interpolate color, normals, texture coordinates, etc. between vertices
  – We did this with barycentrics in ray casting
    • Linear interpolation in object space
  – Is it the same as linear interpolation on the screen?
Interpolation in Screen Space

Two regions of same size in world space

X

z0

image

z1

x'
Interpolation in Screen Space

The farther region shrinks to a smaller area of the screen.

Two regions of same size in world space.
Nope, Not the Same

- Linear variation in world space does not yield linear variation in screen space due to projection
  - Think of looking at a checkerboard at a steep angle; all squares are the same size on the plane, but not on screen
Solution: Barycentrics, Again

- Barycentric coordinates for a triangle \((a, b, c)\)

\[ P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c \]

- Remember, \(\alpha + \beta + \gamma = 1\), \(\alpha, \beta, \gamma \geq 0\)
Solution: Barycentrics, Again

• Barycentric coordinates for a triangle \((a, b, c)\)

\[ P(\alpha, \beta, \gamma) = \alpha a + \beta b + \gamma c \]

– Remember, \(\alpha + \beta + \gamma = 1, \ \alpha, \beta, \gamma \geq 0\)

• Let’s project point \(P\) by projection matrix \(C\)

\[ CP = C(\alpha a + \beta b + \gamma c) \]

\[ = \alpha Ca + \beta Cb + \gamma Cc \]

\[ = \alpha a' + \beta b' + \gamma c' \]

\(a', b', c'\) are the projected homogeneous vertices before division by \(w\)
Solution: Barycentrics, Again

• From last slide:

\[ CP = \alpha a' + \beta b' + \gamma c' \]

• Seems to suggest it’s linear in screen space. But it’s homogenous coordinates
Solution: Barycentrics, Again

- From last slide:

\[ CP = \alpha a' + \beta b' + \gamma c' \]

- Seems to suggest it’s linear in screen space. But it’s homogenous coordinates

- After division by \( w \), the \((x, y)\) screen coordinates are

\[
\left( \frac{P_x}{P_w}, \frac{P_y}{P_w} \right) = \left( \frac{\alpha a'_x + \beta b'_x + \gamma c'_x}{\alpha a'_w + \beta b'_w + \gamma c'_w}, \frac{\alpha a'_y + \beta b'_y + \gamma c'_y}{\alpha a'_w + \beta b'_w + \gamma c'_w} \right)
\]

\( a', b', c' \) are the projected homogeneous vertices
Solution: Barycentrics, Again

\[ CP = \alpha a' + \beta b' + \gamma c' \]

- The (x, y) screen coordinates of P are
  
  \[
  (P_x/P_w, P_y/P_w) = \left( \frac{\alpha a'_x + \beta b'_x + \gamma c'_x}{\alpha a'_w + \beta b'_w + \gamma c'_w}, \frac{\alpha a'_y + \beta b'_y + \gamma c'_y}{\alpha a'_w + \beta b'_w + \gamma c'_w} \right)
  \]

- But our goal is, given x, y, to compute \( \alpha, \beta, \gamma \)

\( a', b', c' \) are the projected homogeneous vertices
Solution: Barycentrics, Again

\[
\begin{pmatrix}
\frac{P_x}{P_w} \\
\frac{P_y}{P_w} \\
1
\end{pmatrix}
\sim
\begin{pmatrix}
P_x \\
P_y \\
P_w
\end{pmatrix}
= \begin{pmatrix}
a'_x \\
a'_y \\
a'_w
\end{pmatrix}
\begin{pmatrix}
b'_x \\
b'_y \\
b'_w
\end{pmatrix}
\begin{pmatrix}
c'_x \\
c'_y \\
c'_w
\end{pmatrix}
\begin{pmatrix}
\alpha \\
\beta \\
\gamma
\end{pmatrix}
\]

- It’s a projective mapping from the barycentrics onto screen coordinates!
  - Represented by a 3x3 matrix
- The inverse of a projection is a projection...
  - We’ll just take the inverse mapping to get from \((x, y, 1)\) to the barycentrics!
From Screen to Barycentrics

Recipe

• Compute projected homogeneous coordinates $a'$, $b'$, $c'$
• Put them in the columns of a matrix, invert it
• Multiply screen coordinates $(x, y, 1)$ by inverse matrix
• Then divide by the sum of the resulting coordinates
  • This ensures the result is sums to one like barycentrics should
• Then interpolate value (e.g. $Z$) from vertices using them!
From Screen to Barycentrics

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

- Notes:
  - matrix is inverted once per triangle
  - can be used to interpolate z, color, texture coordinates, etc.
Pseudocode – Rasterization

For every triangle

ComputeProjection

Compute interpolation matrix (last time)

Compute bbox, clip bbox to screen limits

For all pixels x,y in bbox

Test edge functions

If all $E_i > 0$

compute barycentrics (last time)

interpolate z from vertices

if $z < zbuffer[x,y]$

interpolate UV coordinates from vertices

look up texture color $k_d$

Framebuffer[x,y] = $k_d$  //or more complex shader
Questions?

---

Above & left:
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 unfamiliar “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
Questions?
Supersampling

- Trivial to do with rasterization as well
- Often rates of 2x to 8x
- Requires to compute per-pixel average at the end
- Most effective against edge jaggies
- Usually with jittered sampling
  – pre-computed pattern for a big block of pixels

Tuesday, November 30, 2010
1 Sample / Pixel
4 Samples / Pixel
16 Samples / Pixel
Even this sampling rate cannot get rid of all aliasing artifacts!

We are really only pushing the problem farther.
Related Idea: Multisampling

• Problem
  – Shading is very expensive today (complicated shaders)
  – Full supersampling has linear cost in $k \times k$

• Goal: High-quality edge antialiasing at lower cost

• Solution
  – Compute shading only once per pixel for each primitive, but resolve visibility at “sub-pixel” level
    • Store $(k \times \text{width}, k \times \text{height})$ frame and z buffers, but share shading results between sub-pixels within a real pixel
    • When visibility samples within a pixel hit different primitives, we get an average of their colors
      • Edges get antialiased without large shading cost
Multisampling, Visually

○ = sub-pixel visibility sample
Multisampling, Visually

○ = sub-pixel visibility sample
Multisampling, Visually

O = sub-pixel visibility sample

The color is only computed once per pixel per triangle and reused for all the visibility samples that are covered by the triangle.
Supersampling, Visually

O = sub-pixel visibility sample

When supersampling, we compute colors independently for all the visibility samples.
Multisampling Pseudocode

For each triangle
  For each pixel
    if pixel overlaps triangle
      color=shade()  // only once per pixel!
      for each sub-pixel sample
        compute edge equations & z
        if subsample passes edge equations & z < zbuffer[subsample]
        zbuffer[subsample]=z
        framebuffer[subsample]=color
Multisampling Pseudocode

For each triangle
  For each pixel
    if pixel overlaps triangle
      color = shade() // only once per pixel!
      for each sub-pixel sample
        compute edge equations & z
        if subsample passes edge equations
          && z < zbuffer[subsample]
          zbuffer[subsample] = z
          framebuffer[subsample] = color
  At display time: //this is called “resolving”
    For each pixel
      color = average of subsamples
Multisampling vs. Supersampling

• **Supersampling**
  – Compute an entire image at a higher resolution, then downsample (blur + resample at lower res)

• **Multisampling**
  – Supersample visibility, compute expensive shading only once per pixel, reuse shading across visibility samples

• **But Why?**
  – Visibility edges are where supersampling really works
  – Shading can be prefiltered more easily than visibility

• **This is how GPUs perform antialiasing these days**
Questions?
Examples of Texture Aliasing

Magnification

Minification
Texture Filtering

• Problem: Prefiltering is impossible when you can only take point samples
  – This is why visibility (edges) need supersampling
• Texture mapping is simpler
  – Imagine again we are looking at an infinite textured plane

\textit{textured plane}
Texture Filtering

- We should pre-filter image function \textit{before sampling}
  - That means blurring the image function with a low-pass filter (convolution of image function and filter)
Texture Filtering

- We can combine low-pass and sampling
  - The value of a sample is the integral of the product of the image $f$ and the filter $h$ centered at the sample location
  - “A local average of the image $f$ weighted by the filter $h$”

$$\hat{f}_i = \int f(x) h(x) \, dx$$

image
textured plane

Low-pass filter
Texture Filtering

• Well, we can just as well change variables and compute this integral on the textured plane instead
  – In effect, we are projecting the pre-filter onto the plane
Texture Filtering

- Well, we can just as well change variables and compute this integral on the textured plane instead.
  - In effect, we are projecting the pre-filter onto the plane.
  - It’s still a weighted average of the texture under filter.

\[ \hat{f}_i = \int_{\text{plane}} f(x') h(x') |J(x, x')| \, dx' \]
Texture Pre-Filtering, Visually

- Must still integrate product of projected filter and texture – That doesn’t sound any easier...
Solution: Precomputation

- We’ll precompute and store a set of prefiltered results from each texture with different sizes of prefilters
Solution: Precomputation

• We’ll precompute and store a set of prefiltered results from each texture with different sizes of prefilters
Solution: Precomputation

• We’ll precompute and store a set of prefiltered results from each texture with different sizes of prefilters
  – Because it’s low-passed, we can also subsample
Solution: Precomputation

- We’ll precompute and store a set of prefiltered results from each texture with different sizes of prefilters
  - Because it’s low-passed, we can also subsample
This is Called “MIP-Mapping”

- Construct a pyramid of images that are pre-filtered and re-sampled at 1/2, 1/4, 1/8, etc., of the original image's sampling.
- During rasterization we compute the index of the decimated image that is sampled at a rate closest to the density of our desired sampling rate.
- MIP stands for *multum in parvo* which means *many in a small place*.
MIP-Mapping

- When a pixel wants an integral of the pre-filtered texture, we must find the “closest” results from the precomputed MIP-map pyramid
  - Must compute the “size” of the projected pre-filter in the texture UV domain
MIP-Mapping

- Simplest method: Pick the scale closest, then do usual reconstruction on that level (e.g. bilinear between 4 closest texture pixels)
MIP-Mapping

- Simplest method: Pick the scale closest, then do usual reconstruction on that level (e.g. bilinear between 4 closest texture pixels)
- Problem: discontinuity when switching scale

![Projected pre-filter](image)
Tri-Linear MIP-Mapping

- Use **two** closest scales, compute reconstruction results from both, and linearly interpolate between them.
Tri-Linear MIP-Mapping

- Use **two** closest scales, compute reconstruction results from both, and linearly interpolate between them
- Problem: our filter might not be circular, because of foreshortening

Projected pre-filter
Anisotropic filtering

- Approximate Elliptical filter with multiple circular ones (usually 5)
- Perform trilinear lookup at each one
- i.e. consider five times eight values
  - fair amount of computation
  - This is why graphics hardware has dedicated units to compute trilinear mipmap reconstruction
MIP Mapping Example

Nearest Neighbor

MIP Mapped (Tri-Linear)
MIP Mapping Example

nearest neighbor/point sampling

mipmaps & linear interpolation (tri-linear)
Questions
Storing MIP Maps

- Can be stored compactly: Only 1/3 more space!
Finding the MIP Level

• Often we think of the pre-filter as a box
  – What is the projection of the square pixel “window” in texture space?
Finding the MIP Level

- Often we think of the pre-filter as a box
  - What is the projection of the square pixel “window” in texture space?
  - Answer is in the partial derivatives $p_x$ and $p_y$ of $(u,v)$ w.r.t. screen $(x,y)$

$$p_x = \left(\frac{du}{dx}, \frac{dv}{dx}\right)$$
$$p_y = \left(\frac{du}{dy}, \frac{dv}{dy}\right)$$
For isotropic trilinear mipmapping

- No right answer, circular approximation
- Two most common approaches are
  - Pick level according to the length (in texels) of the longer partial
    \[ \log_2 \max \{ w|p_x|, h|p_y| \} \]
  - Pick level according to the length of their sum
    \[ \log_2 \sqrt{(w|p_x|^2 + (h|p_y|)^2} \]

\[ p_x = (du/dx, dv/dx) \]
\[ p_y = (du/dy, dv/dy) \]
Anisotropic filtering

- Pick levels according to smallest partial
  - well, actually max of the smallest and the largest/5
- Distribute circular “probes” along longest one
- Weight them by a Gaussian

\[ p_x = (\frac{du}{dx}, \frac{dv}{dx}) \]
\[ p_y = (\frac{du}{dy}, \frac{dv}{dy}) \]
How Are Partials Computed?

• You can derive closed form formulas based on the \( uv \) and \( xyw \) coordinates of the vertices...
  – This is what used to be done
• ..but shaders may compute texture coordinates programmatically, not necessarily interpolated
  – No way of getting analytic derivatives!

• In practice, use finite differences
  – GPUs process pixels in blocks of (at least) 4 anyway
    • These 2x2 blocks are called quads
trilinear mipmapping (excessive blurring)

anisotropic filtering
Further Reading

• Paul Heckbert published seminal work on texture mapping and filtering in his master’s thesis (!)
  – Including EWA
  – Highly recommended reading!

• More reading
  – Feline: Fast Elliptical Lines for Anisotropic Texture Mapping, McCormack, Perry, Farkas, Jouppi
    SIGGRAPH 1999
Questions?


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
## Ray Casting vs. Rendering Pipeline

<table>
<thead>
<tr>
<th>Ray Casting</th>
<th>Rendering Pipeline</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>For each pixel</strong></td>
<td><strong>For each triangle</strong></td>
</tr>
<tr>
<td><strong>For each object</strong></td>
<td><strong>For each pixel</strong></td>
</tr>
<tr>
<td>- Whole scene must be in memory</td>
<td>- Harder to get global illumination</td>
</tr>
<tr>
<td>- needs spatial acceleration to be efficient</td>
<td>- Needs smarter techniques to address depth complexity (overdraw)</td>
</tr>
<tr>
<td>+ Depth complexity: no computation for hidden parts</td>
<td>+ Primitives processed one at a time</td>
</tr>
<tr>
<td>+ Atomic computation</td>
<td>+ Coherence: geometric transforms for vertices only</td>
</tr>
<tr>
<td>+ More general, more flexible</td>
<td>+ Good bandwidth/computation ratio</td>
</tr>
<tr>
<td>– Primitives, lighting effects, adaptive antialiasing</td>
<td>+ Minimal state required, good memory behavior</td>
</tr>
</tbody>
</table>
Modern Graphics Hardware

Geometry

Rasterization

Texture

Fragment

Display
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

Numbers are a little old

0.880 GB/s

- Application
- Command
- Geometry
- Rasterization

Rough estimate

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Vertex</td>
<td>5 Gops</td>
</tr>
<tr>
<td>Fragment</td>
<td>150 Gops</td>
</tr>
</tbody>
</table>

0.36 GB/s

- Texture
- Fragment

16 GB/s

- Framebuffer

4 GB/s

- Texture Memory

120 Mpix/s

- Display

1000 Mpix/s

- Rasterization

20 Mvert/s

- Command

- Application

Numbers are a little old

[Akeley, Hanrahan]
Questions?
Movies

both rasterization and ray tracing
Games

rasterization
Simulation

rasterization

(painter for a long time)
rasterization for GUI,
anything for final image
Architecture

ray-tracing, rasterization with preprocessing for complex lighting
Virtual Reality
Visualization

mostly rasterization, interactive ray-tracing is starting
Medical Imaging

same as visualization
Questions?
More issues

- Transparency
  - Difficult, pretty much unsolved!
- Alternative
  - Reyes (Pixar’s renderman)
  - deferred shading
  - pre-Z pass
  - tile-based rendering

- Shadows
  - Next time
- Reflections, global illumination
Transparency

• Triangles and pixels can have transparency (alpha)
• But the result depends on the order in which triangles are sent

• Big problem: visibility
  – There is only one depth stored per pixel/sample
  – transparent objects involve multiple depth
  – full solutions store a (variable-length) list of visible objects and depth at each pixel
    • see e.g. the A-buffer by Carpenter
      [source: http://portal.acm.org/citation.cfm?id=808585]
Deferred shading

- Avoid shading fragments that are eventually hidden
  - shading becomes more and more costly
- First pass: rasterize triangles, store information such as normals, BRDF per pixel
- Second pass: use stored information to compute shading

- Advantage: no useless shading
- Disadvantage: storage, antialiasing is difficult
Pre z pass

• Again, avoid shading hidden fragment
• First pass: rasterize triangles, update only z buffer, not color buffer
• Second pass: rasterize triangles again, but this time, do full shading

• Advantage over deferred shading: less storage, less code modification, more general shading is possible, multisampling possible
• Disadvantage: needs to rasterize twice
Tile-based rendering

- Problem: framebuffer is a lot of memory, especially with antialiasing
- Solution: render subsets of the screen at once
- For each tile of pixels
  - For each triangle
    - for each pixel

- Might need to handle a triangle in multiple tiles
  - redundant computation for projection and setup
- Used in mobile graphics cards
Reyes - Pixar’s Renderman

- Based on micropolygons
  - each primitive gets diced into polygons as small as a pixel
- Enables antialiasing motion blur, depth of field
- Shading is computed at the micropolygon level, not pixel
  - related to multisampling: shaded value will be used for multiple visibility sample
Dicing and rasterization

Figure 4a. A sphere is split into patches, and one of the patches is diced into a 8x8 grid of micropolygons.

Figure 4b. The micropolygons in the grid are transformed to screen space, where they are stochastically sampled.
Reyes - Pixar’s Renderman

- Tile-based to save memory and maximize texture coherence
- Order-independent transparency
  - stores list of fragments and depth per pixel
- Micropolygons get rasterized in space, lens and time
  - Frame buffer has multiple samples per pixel
  - each sample has lens coordinates and time value
Reyes - ignoring transparency

• For each tile of pixels
  – For each geometry
    • Dice into micropolygons adaptively
    • For each micropolygon
      – compute shaded value
      – For each sample in tile at coordinates x, y, u, v, t
        » reproject micropolygon to its position at time t, and lens position uv
        » determine if micropolygon overlaps samples
        » if yes, test visibility (z-buffer)
        » if z buffer passes, update framebuffer
REYES results

Figure 6. 1986 Pixar Christmas Card by John Lasseter and Eben Ostby.

MIT EECS 6.837 – Durand
Questions?