Lecture 3:
High Quality 2D Rasterization

Image Synthesis
Stanford CS348b, Spring 2013
The Reyes Pipeline

Geometric primitives

Bound

Cull?

Yes

No

Can Dice?

No

Yes

Dice

Shade

Rasterize

Grids of micropolygons

Micropolygons

Micropolygons

Grids of micropolygons

Micropolygons
Show pixel-sized upolys

(One pixel)

Micropolygons
In camera space, the camera is at the origin, looking down the z axis.
Rasterization Review

- In camera space, the camera is at the origin, looking down the z axis
- The (perspective) projection matrix projects objects onto the 2D image plane
- In the Reyes context, we’re only rasterizing pixel-sized micropolygons
How Do We Compute The Color of A Pixel?

- Which pixels does each primitive cover?
- What is the color of the first visible primitive at each pixel?
- Z-buffer: store depth of closest surface at each pixel, record its color.
What is a pixel?

Physical Display Pixels

2D Array of Point Samples

http://upload.wikimedia.org/wikipedia/commons/4/4d/Pixel_geometry_01_Pengo.jpg
Computing Pixel Values as Integration

- Consider the area around a pixel as a continuous 2D area
  - A reasonable way to compute the value for the pixel is as the average color over this area

\[
\int f(x, y) \, dx \, dy \approx \frac{1}{N} \sum_{i} f(x_i, y_i)
\]

This integral can be computed as e.g. a Riemann sum.
To Avoid Artifacts, Multiple Samples / Pixel
To Avoid Artifacts, Multiple Samples / Pixel
To Avoid Artifacts, Multiple Samples / Pixel

How many samples, where?
Two entire upcoming lectures on this....
To Avoid Artifacts, Multiple Samples / Pixel
To Avoid Artifacts, Multiple Samples / Pixel

1 sample/pixel

64 samples/pixel
A Triangle as the Interior of Three Edges
A Triangle as the Interior of Three Edges
A Triangle as the Interior of Three Edges
A Triangle as the Interior of Three Edges
A Triangle as the Interior of Three Edges
A Triangle as the Interior of Three Edges
Geometric Building-Blocks

- The signed area of the parallelogram given by the vectors $v_1 = (x_1, y_1)$ and $v_2 = (x_2, y_2)$ is given by

$$\begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} = (x_1 y_2) - (x_2 y_1)$$
Geometric Building-Blocks

- The signed area of the parallelogram given by the vectors $v_1 = (x_1, y_1)$ and $v_2 = (x_2, y_2)$ is given by

$$\begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} = (x_1 y_2) - (x_2 y_1)$$
Geometric Building-Blocks

- The signed area of the parallelogram given by the vectors \( v_1 = (x_1, y_1) \) and \( v_2 = (x_2, y_2) \) is given by

\[
\begin{vmatrix}
  x_1 & x_2 \\
  y_1 & y_2 \\
\end{vmatrix} = (x_1 y_2) - (x_2 y_1)
\]

- Half of this area is the area of the triangle they specify
Geometric Building-Blocks

- The signed area of the parallelogram given by the vectors $v_1=(x_1,y_1)$ and $v_2=(x_2,y_2)$ is given by
  $$\begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} = (x_1 y_2) - (x_2 y_1)$$

- Half of this area is the area of the triangle they specify

- The area of a triangle with vertices $p_0=(x_0,y_0)$, $p_1=(x_1,y_1)$, and $p_2=(x_2,y_2)$ is
  $$\begin{vmatrix} x_1-x_0 & x_2-x_0 \\ y_1-y_0 & y_2-y_0 \end{vmatrix} = 0.5 \left( (x_1-x_0)(y_2-y_0) - (x_2-x_0)(y_1-y_0) \right)$$
Edge Functions as Triangle Area

- Given an edge through vertices $p_0$ and $p_1$, and a point $p=(x,y)$, the area of the triangle they define is
  \[0.5 \cdot (x_1 - x_0)(y - y_0) - (x - x_0)(y_1 - y_0)\]

- Ignore the $0.5$ for now, and define
  \[e(x,y) = (x_1 - x_0)(y - y_0) - (x - x_0)(y_1 - y_0)\]

- Or, $e(x,y) = ax + by + c$, where
  - $a = -(y_1 - y_0)$, $b = (x_1 - x_0)$, $c = (y_1 - y_0)x_0 + (x_1 - x_0)y_0$

- If $e(x,y) > 0$, $(x,y)$ is inside the edge
Some Terminology, and a Note

- $e_0$ is the edge from $(x_1,y_1)$ to $(x_2,y_2)$, $e_1$ is the edge from $(x_2,y_2)$ to $(x_0,y_0)$, and $e_2$ is the edge from $(x_0,y_0)$ to $(x_1,y_1)$.

- We can also see $e(x,y) = ax + by + c$ as $e(x,y) = n \cdot (x,y) + c$, where $n=(a,b)$ is the normal vector to the edge.

- Then $e(x,y) > 0$ means $(x,y)$ is on the same side of the edge as the normal is pointing, etc.
Basic 2D Rasterization

- For each triangle, compute 3 edge functions $e_0, e_1, e_2$
  - For each candidate sample $(x, y)$, see if all three edge functions are $\geq 0$
  - If so, that sample is inside the triangle
    - Interpolate depth, update z-buffer
Edge Edge Cases

- What if $e(x, y) = 0$?
  - Want to report a hit for exactly one of two triangles sharing and edge if this happens

- Tiebreaker rule:
  - If $e(x, y) = 0$, report hit for points on the “left” and “below” the triangle
  - Recall the line normal $(a, b)$:
    - $a < 0$: the edge is on the right
    - $a > 0$: the edge is on the left
Edge Case Handling

```c
bool insideEdge(float a, float b, float c, float x, float y) {
    float e = a * x + b * y + c;
    if (e > 0) return true;
    if (e < 0) return false;
    if (a > 0) return true;
    if (a < 0) return false;
    if (b > 0) return true;
    return false;
}
```
Edge Case Handling

```c
bool insideEdge(float a, float b, float c, float x, float y) {
    float e = a * x + b * y + c;
    if (e > 0) return true;
    if (e < 0) return false;
    if (a > 0) return true;
    if (a < 0) return false;
    if (b > 0) return true;
    return false;
}
```

Basic

```c
// in setup
bool inc = (a > 0) || (a == 0 && b > 0);
...
bool insideEdge(float a, float b, float c, float x, float y, bool inc) {
    float e = a * x + b * y + c;
    if (e > 0) return true;
    if (e < 0) return false;
    else return inc;
}
```

More Efficient
### Barycentric Interpolation

- \( e_0(x, y) = 2 \text{Area}(p_1, p_2, p) \)
- \( e_1(x, y) = 2 \text{Area}(p_2, p_0, p) \)
- \( e_2(x, y) = 2 \text{Area}(p_0, p_1, p) \)

- \( e_0 + e_1 + e_2 = 2 \text{Area}(p_0, p_1, p_2) \)

- Define \( w_i = e_i / (2 \text{Area}(p_0, p_1, p_2)) \)

- Can interpolate per-vertex values (depth, color, etc.) by
  \[
  w_0 v_0 + w_1 v_1 + w_2 v_2
  \]

- Note \( w_i \geq 0 \) and \( w_0 + w_1 + w_2 = 1 \)
Barycentric Interpolation

- $e_0(x,y) = 2 \text{Area}(p_1, p_2, p)$
- $e_1(x,y) = 2 \text{Area}(p_2, p_0, p)$
- $e_2(x,y) = 2 \text{Area}(p_0, p_1, p)$

- $e_0 + e_1 + e_2 = 2 \text{Area}(p_0, p_1, p_2)$

- Define $w_i = e_i / (2 \text{Area}(p_0, p_1, p_2))$

- Can interpolate per-vertex values (depth, color, etc.) by $w_0 v_0 + w_1 v_1 + w_2 v_2$

- Note $w_i \geq 0$ and $w_0 + w_1 + w_2 = 1$
Rasterization 1.0

- In setup (once per triangle), compute:
  - $1/(2 \times \text{Triangle Area})$
  - If signed area $< 0$, backface cull
  - Edge function coefficients
  - Triangle bounding box

- For each sample, compute:
  - Edge function values
    - Interpolate depth, color, ..., if inside triangle
  - Use interpolated values to update z-buffer
Rasterizer 1.0

// setup
float a0 = -(y2 - y1), b0 = x2 - x1, c0 = a0 * -x1 + b0 * -y1;
bool inc0 = (a0 > 0) || (a0 == 0 && b0 > 0);

// a1, b1, c1, inc1, a2, b2, c2, inc2...
float area = 0.5f * ((x1 - x0) * (y2 - y0) - (y1 - y0) * (x2 - x0));
float inv2Area = 1.f / (2.f * area);
if (area <= 0.) return; // backfacing

// compute sample bounds (x0,y0) to (x1,y1)...

// rasterize
for (float y = y0; y < y1; ++y) {
    float (float x = x0; x < x1; ++x) {
        float e0 = a0 * x + b0 * y + c0;
        if (e0 < 0. || (e0 == 0. && !inc0) continue;
        // compute and check e1 and e2

        float w0 = e0 * inv2Area, w1 = e1 * inv2Area, w2 = e2 * inv2Area;
        float z = w0 * z0 + w1 * z1 + w2 * z2;
        if (z < depthBuffer[x][y]) {
            depthBuffer[x][y] = z;
            // interpolate r, g, b
            // update r, g, b in framebuffer
        }
    }
}

Incremental Evaluation of Edge Functions

Perform incremental evaluation of the edge functions.

- Recall \( e(x,y) = ax + by + c = 0 \)
- If we have \( e(x,y) \) and want \( e(x+dx,y+dy) \):

\[
e(x+dx,y+dy) - e(x,y) = a(x+dx) + b(y+dy) + c - (ax + by + c)
= a\, dx + b\, dy
\]

\[
e(x+dx,y+dy) = e(x, y) + a\, dx + b\, dy
\]
Incremental Evaluation Rasterizer

// compute sample bounds (x0,y0) to (x1,y1)...
// setup as before...
float e0 = a0 * x0 + b0 * y0 + c0;
float e0y = e0;
// e1, e2...

for (float y = y0; y < y1; ++y) {
    float (float x = x0; x < x1; ++x, e0 += a0, e1 += a1, e2 += a2) {
        if (e0 < 0. || (e0 == 0. && !inc0) continue;
        // check e1 and e2
        float w0 = e0 * inv2Area, w1 = e1 * inv2Area, w2 = e2 * inv2Area;
        float z = w0 * z0 + w1 * z1 + w2 * z2;
        if (z < depthBuffer[x][y]) {
            depthBuffer[x][y] = z;
            // interpolate r, g, b
            // update r,g, b in framebuffer
        }
    }
    e0 = (e0y += b0);
    e1 = (e1y += b1);
    e2 = (e2y += b2);
}
Tile Culling for 2D Rasterization

- Given edge with normal \((n_x, n_y)\), we can classify a single corner of the tile w.r.t. the line to see if the entire tile is "outside"

- Determine which corner to check:
  \[ p = (x, y) = (n_x \geq 0 \, ? \, x_1 : x_0, \quad n_y \geq 0 \, ? \, y_1 : y_0) \]

- If \( e(x,y) < 0 \), all samples in the tile are culled

- Can also detect tiles that are completely inside the triangle...
// compute sample bounds (x0,y0) to (x1,y1)...
int tx0 = x0 >> logTileSize, tx1 = (x1 + tileSize - 1) >> logTileSize;
int ty0 = y0 >> logTileSize, ty1 = (y1 + tileSize - 1) >> logTileSize;
// setup as before...
// e0, e0y, e1, e1y, e2y, ...
float e0tile = a0 * (tx0 * tileSize + (a0 > 0 ? tileSize : 0) + b0 * (ty0 * tileSize + (b0 > 0 ? tileSize : 0) + c0;
float e0tiley = e0tile;
// e1tile, e2tile

for (float ty = ty0; ty < ty1; ++ty) {
    for (float tx = tx0; tx < tx1; ++tx, e0tile += a0*tileSize, e1tile += a1*tileSize, e2tile += a2*tileSize) {
        if (e1tile < 0. || e1tile < 0. || e2tile < 0.)
            continue;
        else { /* rasterize samples inside tile */ }
    }
    e0tile = (e0tiley += b0 * tileSize);
    e1tile = (e1tiley += b1 * tileSize);
    e2tile = (e2tiley += b2 * tileSize);
}
Hierarchical Tile Culling

YO DAWG I HEARD YOU LIKE TILES
SO I PUT TILES IN YOUR TILES SO YOU CAN CULL MORE EFFECTIVELY
Reyes Implications for Rasterization

- Recall that we’re rasterizing grids of quads / triangles
  - Each edge is shared by 2 triangles (just opposite sign)
  - Rasterize pairs of triangles from grids, ...
- Don’t have to worry about clipping
  - Split / cull ensures no triangles spanning \( z=0 \)
- Triangles are generally all the same size
- Triangles are generally well-proportioned
Occlusion Culling

- Maintain conservative z-buffer during rasterization
- Support the query: given a bbox in pixel coordinates with \( z_{\min} \) nearest depth, is that box definitely hidden?
- Use to cull primitives during split/dice
- Can also cull grids, triangles during rasterization
Occlusion Culling: Hierarchical Z-Buffer

Basic Per-Sample Z-Buffer

compute maximum of z depths
Occlusion Culling: Hierarchical Z-Buffer

Basic Per-Sample Z-Buffer
Occlusion Culling: Hierarchical Z-Buffer

![Basic Per-Sample Z-Buffer](image)

Stanford CS348b, Spring 2013
Occlusion Culling: Hierarchical Z-Buffer

Basic Per-Sample Z-Buffer
Occlusion Culling: Hierarchical Z-Buffer

Basic Per-Sample Z-Buffer

If triangle $z_{\text{min}} > \text{aggregate } z_{\text{max}}$ values for overlapping hierarchical z-buffer regions, can cull
Transparency

- Transparency is a key visual effect: glass, smoke/fog, ...
  - Also, helps with anti-aliasing when rendering hair
- Need first opaque object and transparent layers
- Requires sorting visible points by depth

\[
\text{Final color} = \text{color()} + \text{transmittance()} \times \\
(\text{color()} + \text{transmittance()} \times \\
(\text{color()} + \text{transmittance()} \times \\
(\text{color()} + \text{transmittance()} \times \\
\text{color()})))
\]
A-Buffer Transparency

- Store a linked-list of transparent fragments at each sample
- Each fragment stores color, transmittance, depth
- When done rasterizing, sort each list, apply compositing equation
- +s: it gives the correct result!
- -s: unbounded memory requirement

```c
struct PixelSample {
  Color rgb;
  float z;
  Fragment *head;
};

struct Fragment {
  Color rgb, tr;
  float z;
  Fragment *next;
};
```
Adaptive Order-Independent Transparency

Final color = \text{color(\() +
\text{transparency(\() \times \text{color(\() +
\text{transparency(\() \times \text{color(\() +
\text{transparency(\() \times \text{color(\() +
\text{transparency(\() \times \text{color(\()
Adaptive Order-Independent Transparency

#define N 8

struct AOITNode {
    Color rgb, T;
    float z;
};

struct PixelSample {
    Color rgb;
    float z;
    AOITNode nodes[N];
};

Final color = color() +
transparency() * color() +
transparency() * color() +
transparency() * color() +
transparency() * color() +
Reyes A-buffer

- Many visibility samples per pixel (recall: 64-128)
- Many visible points per sample (under conditions of significant transparency)

1920x1080 rendering (1080p)
64 visibility samples per pixel
4 visible points per sample (rgb,a,z)

~10 GB A-buffer !!!
Reyes Implementations use Bucketing

- **Image is split into buckets of ~32x32 pixels**
- **Goal is to keep the framebuffer for just one bucket in memory**

  for each primitive, place in screen bucket
  for each bucket
    allocate framebuffer for bucket
    for each primitive
      split-dice to create grids // each split, cull primitives falling outside of bucket
      shade + rasterize grids

- **Reduces memory for image to a single bucket’s worth**
- **Increases memory needs for primitives, grids, micropolygons**
  - Need to hold on to ones that cover multiple buckets
Acknowledgement

- Slides 3, 33, and 34 were provided by Kayvon Fatahalian.