Visibility

- What scene geometry is visible within each screen pixel?
  - What geometry projects into a screen pixel? (screen coverage)
  - Which of this geometry is visible from the camera at that pixel? (occlusion)
Visibility on GPU: rasterization + Z-buffering

- The rasterizer converts a primitives (triangles) into fragments
  - Computes covered pixels (selection: what fragments get generated?)
  - Computes triangle attributes for fragment (attribute assignment: how is surface data is associated with the fragment?)

- Recall: frame-buffer operations stage handles occlusion using the Z-buffer algorithm
  - Although there are many optimizations (we will discuss some today)

```
struct fragment {
    float3 normal;     // interpolated application-defined attrs
    float2 texcoord1;  // (e.g., texture coordinates, surface normal)
    float2 texcoord2;
    int x, y;          // pixel position corresponding to fragment
    float depth;       // triangle depth for fragment
}
```
Fragment selection: What does it mean for a pixel to be covered by a triangle?
Integrate pixel coverage analytically

(A fragment is an area sample)
Analytical schemes get tricky when considering occlusion

Two regions of [1] contribute to pixel. One of these regions is not convex.

Interpenetration: even worse

Note: unbounded storage per pixel.
Modern GPU fragment selection: point sample triangle-pixel coverage

Pixel \((x, y)\)

Example: Coverage sample point at pixel center

\(\uparrow\) = triangle covers sample, fragment generated for pixel

\(\downarrow\) = triangle does not cover sample, no fragment generated
Edge cases (literally)

Is fragment generated for triangle 1? for triangle 2?
Edge rules

- Direct3D rules: when edge falls directly on sample, sample classified as within triangle if the edge is a “top edge” or “left edge”
  - Top edge: horizontal edge that is above all other edges
  - Left edge: an edge that is not exactly horizontal and is on the left side of the triangle. (triangle can have one or two left edges)

Source: Direct3D Programming Guide, Microsoft
Super-sampling to anti-alias edges

(will discuss next time)
Point-in-triangle test

\[ P_i = \left( \frac{x_i}{w_i}, \frac{y_i}{w_i}, \frac{z_i}{w_i} \right) = (X_i, Y_i, Z_i) \]

\[ dX_i = X_{i+1} - X_i \]
\[ dY_i = Y_{i+1} - Y_i \]

\[ E_i(x, y) = (x - X_i) dY_i - (y - Y_i) dY_i \]
\[ = A_i x + B_i y + C_i \]

\[ E_i(x, y) = 0 : \text{point on edge} \]
\[ > 0 : \text{outside edge} \]
\[ < 0 : \text{inside edge} \]
Point-in-triangle test

\[ P_i = (x_i/w_i, y_i/w_i, z_i/w_i) = (X_i, Y_i, Z_i) \]

\[ dX_i = X_{i+1} - X_i \]

\[ dY_i = Y_{i+1} - Y_i \]

\[ E_i(x, y) = (x-X_i) \, dY_i - (y-Y_i) \, dY_i = A_i \, x + B_i \, y + C_i \]

\[ E_i(x, y) = 0 \quad : \text{point on edge} \]

\[ > 0 \quad : \text{outside edge} \]

\[ < 0 \quad : \text{inside edge} \]
Point-in-triangle test

\[ P_i = (x_i/w_i, y_i/w_i, z_i/w_i) = (X_i, Y_i, Z_i) \]

\[ dX_i = X_{i+1} - X_i \]
\[ dY_i = Y_{i+1} - Y_i \]

\[ E_i(x,y) = (x-X_i)\ dY_i - (y-Y_i)\ dY_i = A_i\ x + B_i\ y + C_i \]

\[ E_i(x,y) = 0 \text{ : point on edge} \]
\[ > 0 \text{ : outside edge} \]
\[ < 0 \text{ : inside edge} \]
Point-in-triangle test

\( P_i = (x_i/w_i, y_i/w_i, z_i/w_i) = (X_i, Y_i, Z_i) \)

\[
\begin{align*}
    dX_i &= X_{i+1} - X_i \\
    dY_i &= Y_{i+1} - Y_i
\end{align*}
\]

\[
E_i(x,y) = (x-X_i) dY_i - (y-Y_i) dY_i = A_i x + B_i y + C_i
\]

\( E_i(x,y) = 0 \): point on edge
\( > 0 \): outside edge
\( < 0 \): inside edge
Point-in-triangle test

\[ P_i = \left( \frac{x_i}{w_i}, \frac{y_i}{w_i}, \frac{z_i}{w_i} \right) = (X_i, Y_i, Z_i) \]

\[ dX_i = X_{i+1} - X_i \]
\[ dY_i = Y_{i+1} - Y_i \]

\[ E_i(x, y) = (x - X_i) \, dY_i - (y - Y_i) \, dY_i \]
\[ = A_i \, x + B_i \, y + C_i \]

\[ E_i(x, y) = 0 \ : \text{point on edge} \]
\[ > 0 \ : \text{outside edge} \]
\[ < 0 \ : \text{inside edge} \]
Incremental triangle traversal

\[ P_i = \left( \frac{x_i}{w_i}, \frac{y_i}{w_i}, \frac{z_i}{w_i} \right) = (X_i, Y_i, Z_i) \]

\[ dX_i = X_{i+1} - X_i \]
\[ dY_i = Y_{i+1} - Y_i \]

\[ E_i(x,y) = (x-X_i) \ dY_i - (y-Y_i) \ dY_i \]
\[ = A_i \ x + B_i \ y + C_i \]

\[ E_i(x,y) = 0 : \text{point on edge} \]
\[ > 0 : \text{outside edge} \]
\[ < 0 : \text{inside edge} \]

Note incremental update:

\[ dE_i(x+1,y) = E_i(x,y) + dY_i = E_i(x,y) + A_i \]
\[ dE_i(x,y+1) = E_i(x,y) + dX_i = E_i(x,y) + B_i \]

Incremental update saves computation:
One addition per edge, per sample test

Note: many traversals possible: backtrack, zig-zag, Hilbert/Morton curves (locality maximizing)
Modern hierarchical traversal

Traverse triangle as before, but in blocks

Test all samples in block against triangle in parallel (data-parallelism)

Can be implemented as multi-level hierarchy.

Advantages:
- Simplicity of wide parallel execution overcomes cost of extra point-in-triangle tests (recall: most triangles cover many samples, especially when super-sampling coverage)
- Can skip sample testing work (early outs): entire block not in triangle, entire block entirely within triangle
- Important for early Z cull (later in this lecture)

Another modern approach: Hierarchical Recursive Descent.
(See Mike Abrash’s Dr. Dobbs article in readings)
Attribute assignment

- How are fragment attributes (color, normal, texcoords) computed?
  - Point sample attributes as well. (e.g., at pixel center)
  - Must compute $A(x, y)$ for all attributes

Computing a plane equation for an attribute:

Attribute values at three vertices: $A_0$, $A_1$, $A_2$
Projected positions of three vertices: $(X_0, Y_0)$, $(X_1, Y_1)$, $(X_2, Y_2)$
$A(x, y) = ax + by + c$

$A_0 = aX_0 + bY_0 + c$
$A_1 = aX_1 + bY_1 + c$
$A_2 = aX_2 + bY_2 + c$

3 equations, 3 unknowns. Solve for $a, b, c$ **

** Discard zero-area triangles before getting here (recall we computed area in back-face culling)
Perspective correct interpolation

Attribute values are linear on triangle in 3D, but not linear in projected screen XY

\[
\frac{(A_0 + A_1)}{2}
\]
Perspective-correct interpolation

Linear screen interpolation of \((u,v)\)

Perspective-correct interpolation of \((u,v)\)

[images from Heckbert and Moreton 1991]
Perspective correct interpolation

Attribute values are linear on triangle in 3D, but not linear in projected screen XY
But... projected values \((A/w)\) are linear in screen XY: compute plane equations from \(A/w\)

For each generated fragment:

- evaluate \(1/w(x,y)\) (from precomputed plane equation)
- reciprocate to get \(w(x,y)\)
- for each attribute
  - evaluate \(A/w(x,y)\) (from precomputed plane equation)
  - multiply result by \(w(x,y)\) to get \(A(x,y)\)
Storage optimization:
store plane equations separate from fragments
(very useful for large triangles)

Note: can skip attribute evaluation during traversal/coverage testing (evaluate attributes as needed, on demand, during subsequent fragment processing)
Rasterization

- **Triangle setup:**
  - Transform clip space vertex positions to screen space
  - Convert positions to fixed point (Direct3D specifies 8 bits of subpixel precision**)
  - Compute edge equations
  - Compute plane equations for all vertex attributes and Z

- **Traverse**
  - Compute covered fragments using edge tests
  - Emit fragments (also emit per-triangle data as necessary)

**Note 1:** limited precision can be a good thing: can limit really acute triangles (they snap to 0 area)

**Note 2:** limited precision can be a bad thing: precision limits in (x,y) can limit precision in Z (see Akeley and Su, 2006)
Recall: z-buffer for occlusion

- Z-buffer stores depth of scene at each coverage sample
  - Each sample, not just each pixel
  - In practice, usually stores $z/w$

- Triangles are planar: each triangle has exactly one depth at each sample (consistent ordering of fragments for each sample) ** ✓

- After fragment processing (shading) ...

```c
if (fragment.depth < z_buffer[fragment.x][fragment.y])
{
    color_buffer[fragment.x][fragment.y].rgba =
    blend(color_buffer[fragment.x][fragment.y].rgba, fragment.rgba);
    z_buffer[fragment.x][fragment.y] = fragment.depth;
}
```

- Constant time occlusion test per fragment ✓
- Constant space per coverage sample ✓

** assumes edge-on triangles have been discarded
Z-buffer for occlusion

- High bandwidth requirements (particularly when super-sampling)
  - Number of Z-buffer reads/writes depends on:
    - depth complexity of the scene
    - order triangles are provided to the graphics pipeline
      (if depth test fails, don’t write Z or rgba)

- Bandwidth estimate:
  - 60 Hz * 2 MPixel image * avg. depth complexity 4 (assume replace 50%, 32-bit Z) = 2.8 GB/s
  - If super-sampling, multiply by 4 or 8x
  - 5 shadow maps per frame (1 MPixel, not super-sampled): additional 8.6 GB/s
  - Note: this does not include color buffer bandwidth

- Modern GPU implementations employ caching, compression
  - Recall sort-middle chunked: Z-buffer for current tile always on chip, can
    (sometimes) skip write of final Z values to memory (Z-buffer bandwidth = 0)
Z-buffer compression

- Modern GPUs implement some form of lossless Z-buffer compression

- Very large compression ratios possible by exploiting screen coherence in depth values
  - Store plane equation for $Z$ for an entire tile of pixels (possible when triangle covers tile)
  - Store base + low precision offsets for each sample in a tile
Early Z-culling ("early Z")

Goal: discard useless fragments from pipeline as soon as possible
Early Z-culling ("early Z")

Constraint: occlusion cannot depend on shading
e.g., pipeline alpha test enabled, fragment shader modifies Z

Note: Only provides benefit if blue triangle is rendered by application first.
Early Z

- Perform depth test after rasterization, prior to fragment shading
- Reduces fragment processing work
  - Amount of reduction dependent on triangle ordering
  - Ideal: front-to-back order
- **Does not** reduce Z-buffer bandwidth (same Z reads and writes still occur)
- Common trick: “Z-prepass”
  - Two rendering passes
    1. Render all scene geometry, with fragment processing disabled (pre-populate the Z-buffer)
    2. Re-render scene with shading enabled
  - Overhead of processing geometry twice vs. maximal early Z culling
Hierarchical early Z: "hi-Z"

Recall hierarchical traversal during rasterization

For each screen tile, compute farthest value in the z-buffer: \( z_{\text{far}} \)

During traversal, for each tile:

1. Compute closest point on triangle in tile: 
   \( \text{tri}_{\text{near}} \) (using Z plane equation)

2. If \( \text{tri}_{\text{near}} > z_{\text{far}} \), then triangle is occluded in this tile. Proceed immediately to next tile. (no fragments generated)

Note, if z-buffer also stores \( z_{\text{near}} \) for each tile and \( \text{tri}_{\text{far}} < z_{\text{near}} \), then all depth tests for triangle in tile will pass. (no need to check individual per-sample depth values later)
Hierarchical + early Z-culling

Remember: these are GPU implementation optimizations. They are not reflected in the pipeline abstraction.
Hierarchical Z

- Perform depth test at tile granularity prior to sampling coverage
  - Reduces rasterization work
  - Reduces required Z-buffer bandwidth
  - *Does not* reduce fragment processing work more than early Z (conservative optimization: will discard a subset of the fragments early Z does)
Modern research topic

- Accurate camera simulation in real-time rendering
  - Visibility algorithms discussed today simulate image formation by virtual pinhole camera, with infinite shutter
  - Real cameras have finite apertures, finite exposure duration
  - Visibility computation requires integration over time and lens aperture (high computational cost + diminished spatial coherence)

Time integration: motion blur

Lens integration: defocus blur
Readings

Rasterization Techniques:
- M. Olano and T. Greer, Triangle Scan Conversation Using 2D Homogeneous Coordinates. Graphics Hardware 97
- M. Abrash, Rasterization on Larrabee, Dr. Dobbs Portal. May 1, 2009
  http://drdobbs.com/high-performance-computing/217200602
- Take a look at source code for NVIDIA CUDA rasterizer:

Hierarchical Z-Buffering:
- N. Greene et al., Hierarchical Z-Buffer Visibility. SIGGRAPH 93
- S. Morien, ATI Radeon HyperZ Technology. Hot 3D Presentation, Graphics Hardware 2000

Z-Buffer Precision:

Recent Rasterization Topics:
- K. Fatahalian et al., Data-parallel Rasterization of Micropolygons with Motion and Defocus Blur. High Performance Graphics 2009
- S. Laine et al., Clipless Dual-Space Bounds for Faster Stochastic Rasterization. SIGGRAPH 2011

Also Highly Recommended: