CS380: Computer Graphics
Triangle Rasterization

Sung-Eui Yoon
(윤성의)

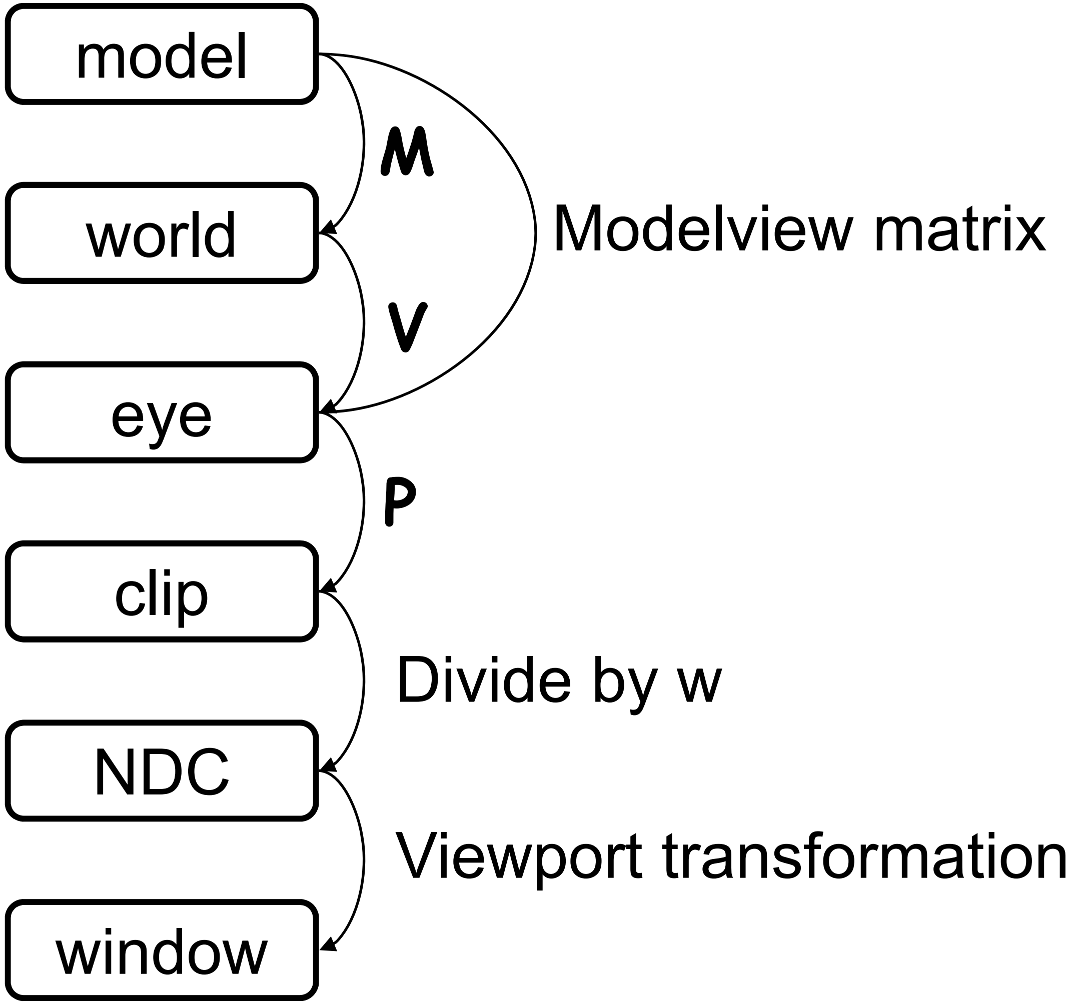
Course URL:
http://sglab.kaist.ac.kr/~sungeui/CG/

KAIST

Class Objectives (Ch. 8)

- Understand triangle rasterization using edge-equations
- Understand mechanics for parameter interpolations
- Realize benefits of incremental algorithms

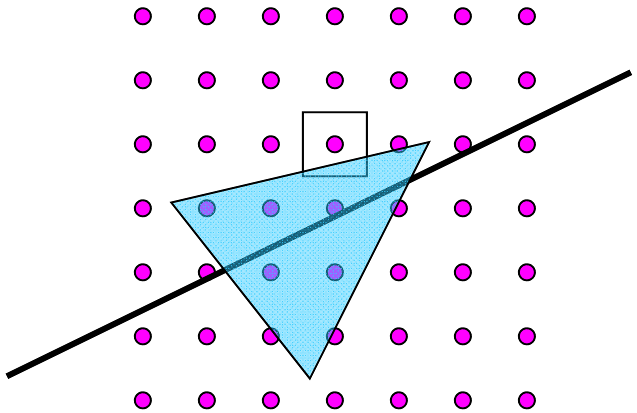
Coordinate Systems



KAIST

Primitive Rasterization

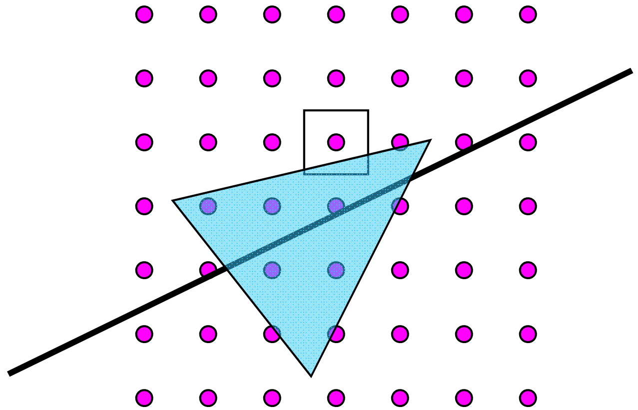
- **Rasterization converts vertex representation to pixel representation**



- **Coverage determination**
  - **Computes which pixels (samples) belong to a primitive**
- **Parameter interpolation**
  - **Computes parameters at covered pixels from parameters associated with primitive vertices**

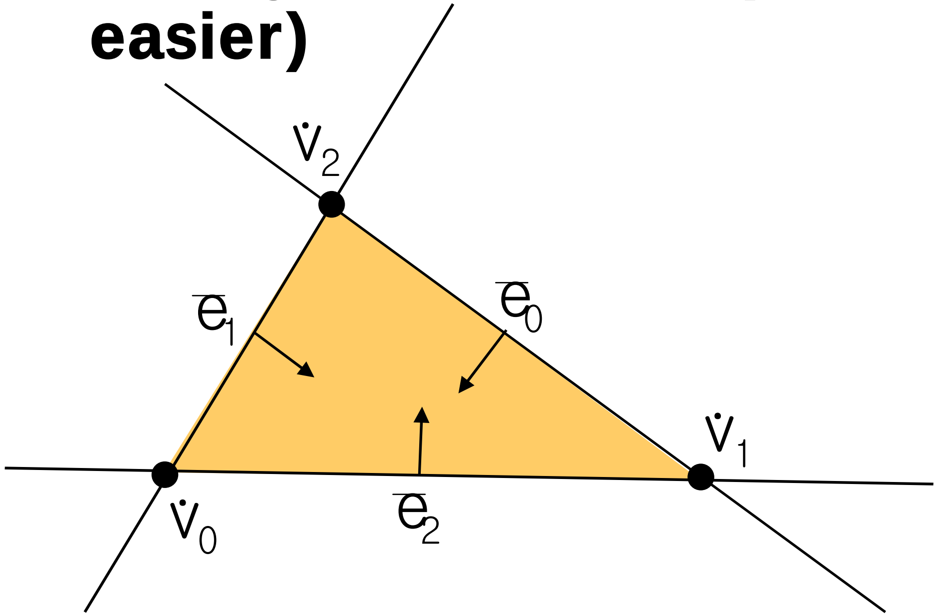
Coverage Determination

- Coverage is a 2D sampling problem
- Possible coverage criteria:
  - Distance of the primitive to sample point (often used with lines)
  - Percent coverage of a pixel (used to be popular)
  - Sample is inside the primitive (assuming it is closed)



Why Triangles?

- **Triangles are simple**
  - **Simple representation for a surface element (3 points or 3 edge equations)**
  - **Triangles are linear (makes computations easier)**

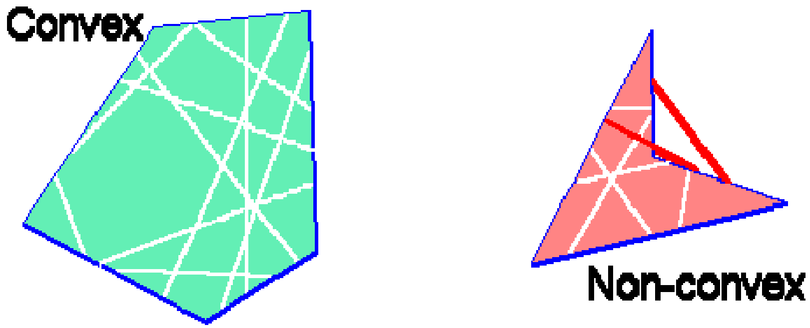


$$T = (\dot{v}_0, \dot{v}_1, \dot{v}_2)$$

$$T = (\bar{e}_0, \bar{e}_1, \bar{e}_2)$$

Why Triangles?

- Triangles are convex
- What does it mean to be a convex?



An object is convex if and only if any line segment connecting two points on ts boundary is contained entirely within the object or one of its boundaries

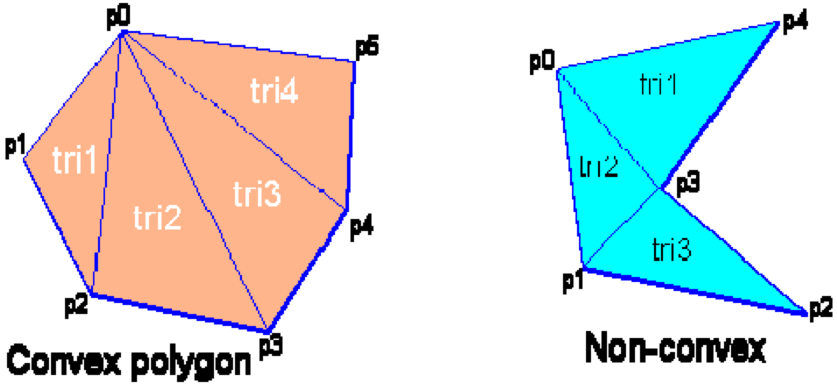
Why Triangles?

- **Triangles are convex**

- **Why is convexity important?**
  - **Regardless of a triangle's orientation on the screen a given scan line will contain only a single segment or *span* of that triangle**
  - **Simplify rasterization processes**

Why Triangles?

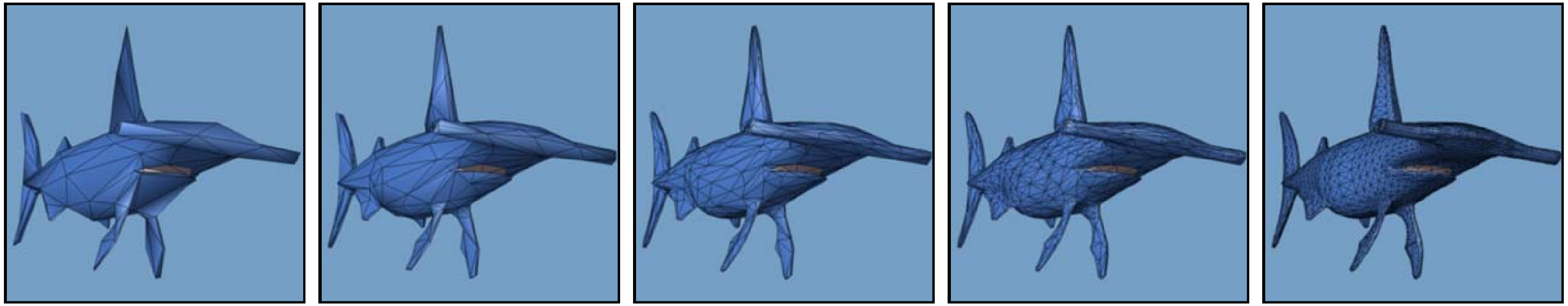
- **Arbitrary polygons can be decomposed into triangles**



- **Decomposing a convex n-sided polygon is trivial**
  - **Suppose the polygon has ordered vertices $\{v_0, v_1, \ldots v_n\}$**
  - **It can be decomposed into triangles $\{(v_0,v_1,v_2), \{v_0,v_2,v_3), (v_0,v_i,v_{i+1}), \ldots (v_0,v_{n-1},v_n)\}$**
- **Decomposing a non-convex polygon is non-trivial**
  - **Sometimes have to introduce new vertices**

Why Triangles?

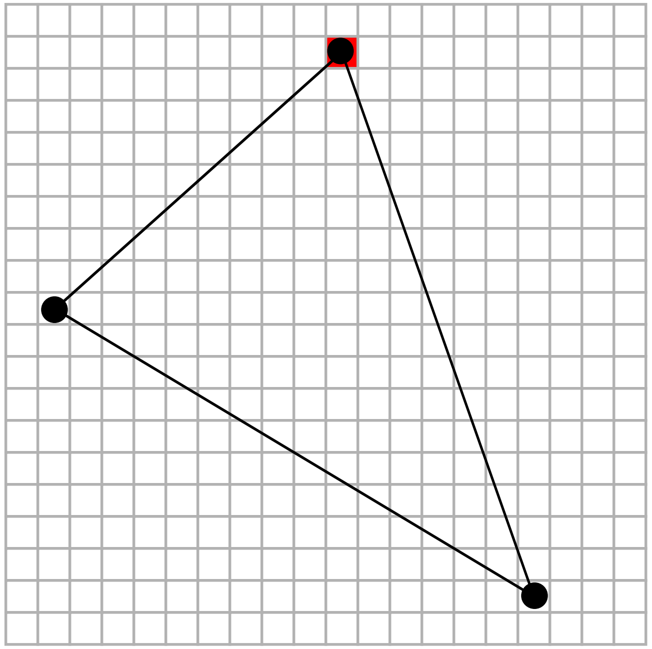
- **Triangles can approximate any 2-dimensional shape (or 3D surface)**
  - **Polygons are a locally linear (planar) approximation**
- **Improve the quality of fit by increasing the number edges or faces**



KAIST

Scanline Triangle Rasterizer

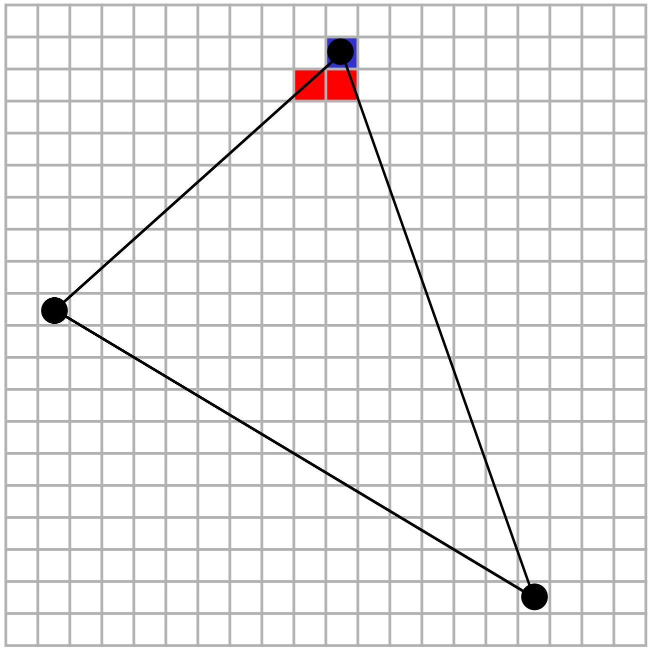
- **Walk along edges and process one scanline at a time; also called edge walk method**
- **Rasterize spans between edges**



KAIST

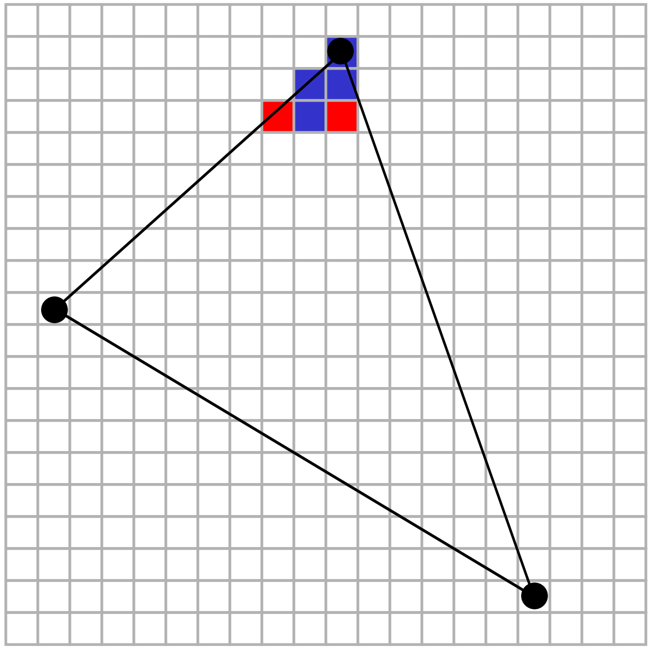
Scanline Triangle Rasterizer

- **Walk along edges and process one scanline at a time**
- **Rasterize spans between edges**



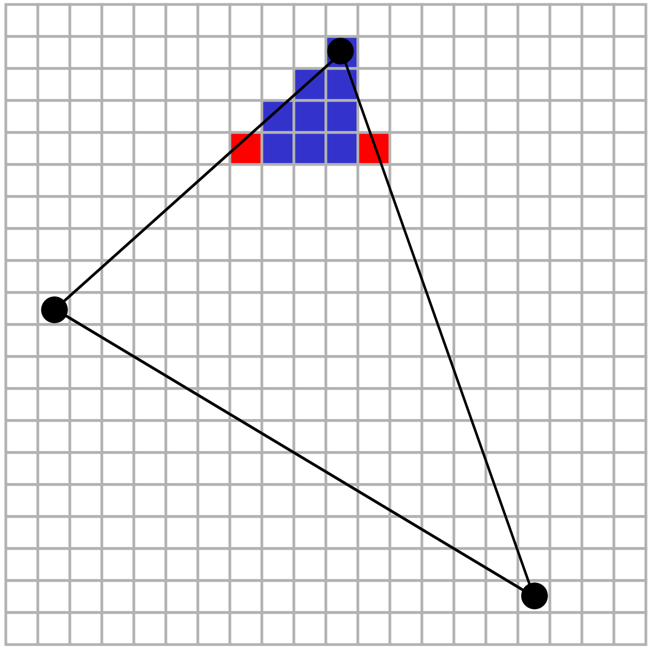
Scanline Triangle Rasterizer

- **Walk along edges and process one scanline at a time**
- **Rasterize spans between edges**



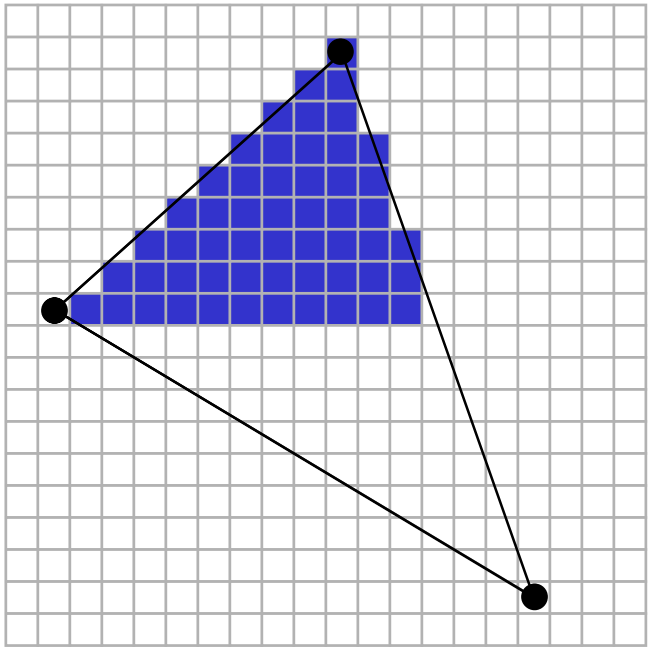
Scanline Triangle Rasterizer

- Walk along edges and process one scanline at a time
- Rasterize spans between edges



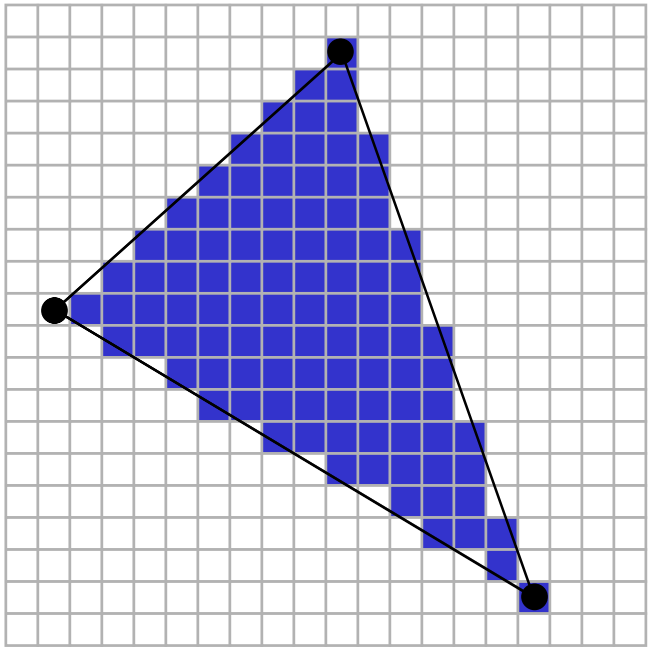
Scanline Triangle Rasterizer

- Walk along edges and process one scanline at a time
- Rasterize spans between edges



Scanline Triangle Rasterizer

- **Walk along edges and process one scanline at a time**
- **Rasterize spans between edges**

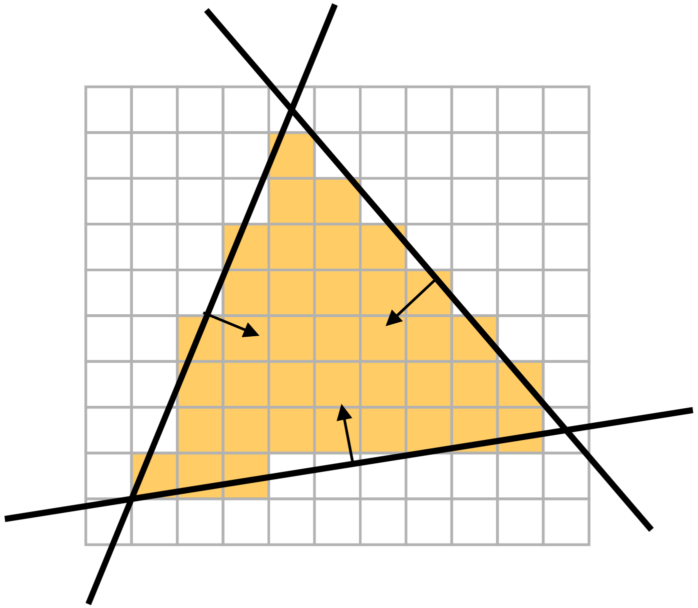


Scanline Rasterization

- **Advantages:**
  - **Can be made quite fast**
  - **Low memory usage for small scenes**
  - **Do not need full 2D z-buffer (can use 1D z-buffer on the scanline)**
- **Disadvantages:**
  - **Does not scale well to large scenes**
  - **Lots of special cases**

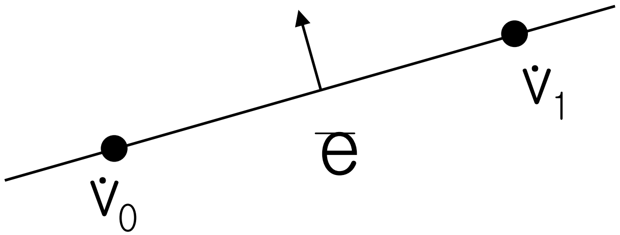
Rasterizing with Edge Equations

- Compute edge equations from vertices
- Compute interpolation equations from vertex parameters
- Traverse pixels evaluating the edge equations
- Draw pixels for which all edge equations are positive
- Interpolate parameters at pixels



Edge Equation Coefficients

- **The cross product between 2 homogeneous points generates the line between them**



$$\bar{e} = \dot{v}_0 \times \dot{v}_1$$
$$= [x_0 \quad y_0 \quad 1]^t \times [x_1 \quad y_1 \quad 1]^t$$
$$= [(y_0 - y_1) \quad (x_1 - x_0) \quad (x_0 y_1 - x_1 y_0)]$$

A B C

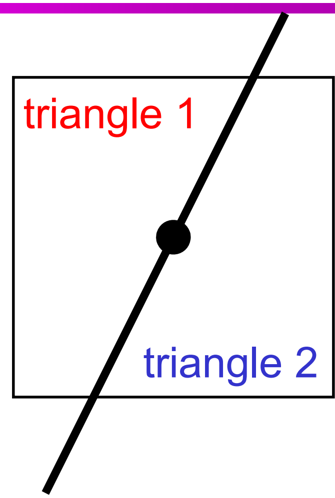
$$E(x,y) = Ax + By + C$$

- **A pixel at (x,y) is "inside" an edge if E(x,y) > 0**

KAIST

Shared Edges

- **Suppose two triangles share an edge. Which covers the pixel when the edge passes through the sample (E(x,y)=0)?**

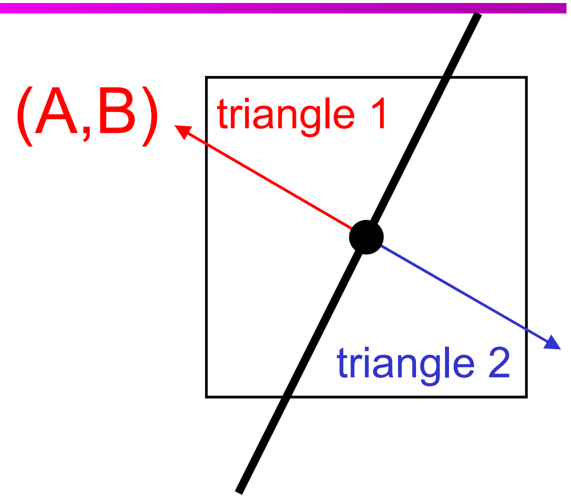


- **Both**
  - **Pixel color becomes dependent on order of triangle rendering**
  - **Creates problems when rendering transparent objects - "double hitting"**
- **Neither**
  - **Missing pixels create holes in otherwise solid surface**
- **We need a consistent tie-breaker!**

Shared Edges

- **A common tie-breaker:**

$$\text{bool } t = \begin{cases} A > 0 & \text{if } A \neq 0 \\ B > 0 & \text{otherwise} \end{cases}$$

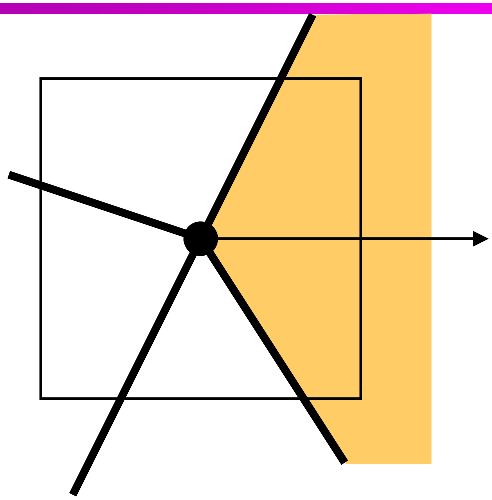


- **Coverage determination becomes**

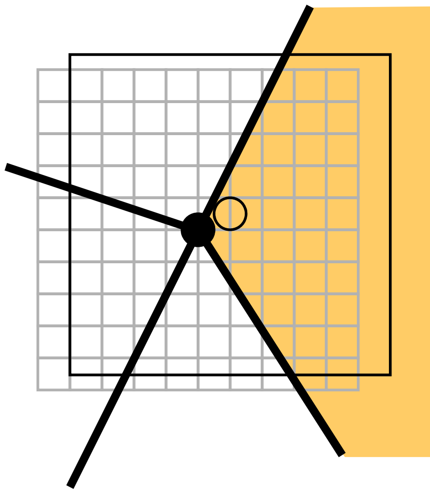
  if(E(x,y) >0 || (E(x,y)==0 && t))

  pixel is covered

Shared Vertices



- Use "inclusion direction" as a tie breaker
- Any direction can be used

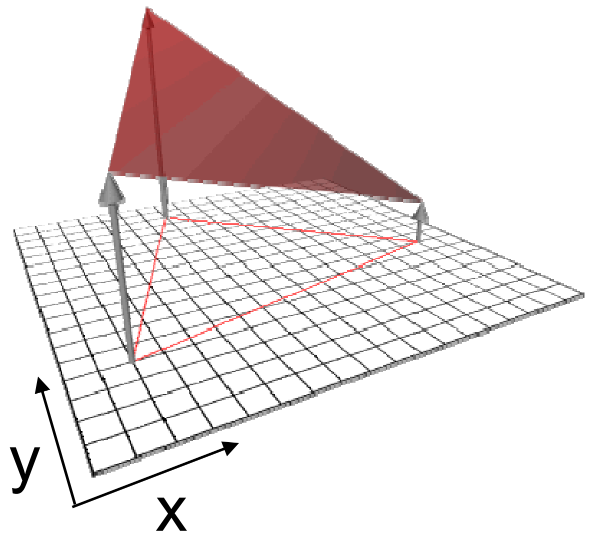


- Snap vertices to subpixel grid and displace so that no vertex can be at the pixel center

KAIST

Interpolating Parameters

- **Specify a parameter, say redness (r) at each vertex of the triangle**
  - **Linear interpolation creates a planar function**



$$r(x,y) = Ax + By + C$$

Solving for Linear Interpolation Equations

- **Given the redness of the three vertices, we can set up the following linear system:**

$$\begin{bmatrix} r_0 & r_1 & r_2 \end{bmatrix} = \begin{bmatrix} A_r & B_r & C_r \end{bmatrix} \begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{bmatrix}$$

with the solution:

$$\begin{bmatrix} A_r & B_r & C_r \end{bmatrix} = \begin{bmatrix} r_0 & r_1 & r_2 \end{bmatrix} \frac{\begin{bmatrix} (y_1 - y_2) & (x_2 - x_1) & (x_1 y_2 - x_2 y_1) \\ (y_0 - y_2) & (x_2 - x_0) & (x_0 y_2 - x_2 y_0) \\ (y_0 - y_1) & (x_1 - x_0) & (x_0 y_1 - x_1 y_0) \end{bmatrix}}{\det \begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{bmatrix}}$$

Triangle Area

$$\begin{aligned}\text{Area} &= \tfrac{1}{2}\det\begin{bmatrix} x_0 & x_1 & x_2 \\ y_0 & y_1 & y_2 \\ 1 & 1 & 1 \end{bmatrix} \\ &= \tfrac{1}{2}\left((x_1y_2 - x_2y_1) - (x_0y_2 - x_2y_0) + (x_0y_1 - x_1y_0)\right) \\ &= \tfrac{1}{2}(C_0 + C_1 + C_2)\end{aligned}$$

- **Area = 0 means that the triangle is not visible**
- **Area < 0 means the triangle is back facing:**
  - **Reject triangle if performing back-face culling**
  - **Otherwise, flip edge equations by multiplying by -1**

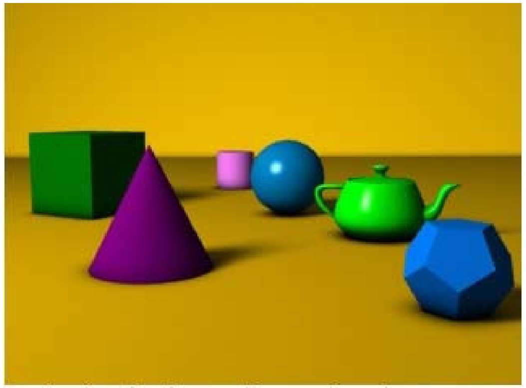
Interpolation Equation

- **The parameter plane equation is just a linear combination of the edge equations**

$$\begin{bmatrix} A_r & B_r & C_r \end{bmatrix} = \frac{1}{2 \cdot area} \begin{bmatrix} r_0 & r_1 & r_2 \end{bmatrix} \begin{bmatrix} \overline{e}_0 \\ \overline{e}_1 \\ \overline{e}_2 \end{bmatrix}$$

Z-Buffering

- When rendering multiple triangles we need to determine which triangles are visible
- Use z-buffer to resolve visibility
  - Stores the depth at each pixel
- Initialize z-buffer to 1 (far value)
  - Post-perspective z values lie between 0 and 1
- Linearly interpolate depth (z_{tri}) across triangles
- If $z_{tri}(x,y)$ < zBuffer[x][y]
  write to pixel at (x,y)
  zBuffer[x][y] = $z_{tri}(x,y)$



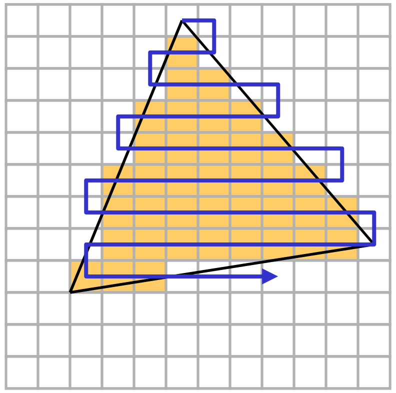
A simple three dimensional scene

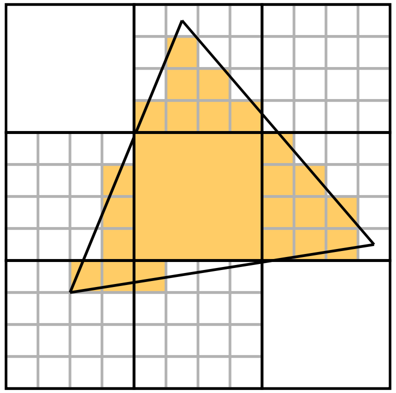
Z-buffer representation

image from wikipedia.com

Traversing Pixels

- Free to traverse pixels
  - Edge and interpolation equations can be computed at any point
- Try to minimize work
  - Restrict traversal to primitive bounding box
  - Hierarchical traversal
    - Knock out tiles of pixels (say 4x4) at a time
    - Test corners of tiles against equations
    - Test individual pixels of tiles not entirely inside or outside





Incremental Algorithms

- **Some computation can be saved by updating the edge and interpolation equations incrementally:**

$$
\begin{aligned}
E(x,y) &= Ax + By + C \\
E(x+\Delta, y) &= A(x+\Delta) + By + C \\
&= E(x,y) + A \cdot \Delta \\
E(x, y+\Delta) &= Ax + B(y+\Delta) + C \\
&= E(x,y) + B \cdot \Delta
\end{aligned}
$$

- **Equations can be updated with a single addition!**

KAIST

Triangle Setup

- Compute edge equations
  - 3 cross products
- Compute triangle area
  - A few additions
- Cull zero area and back-facing triangles and/or flip edge equations
- Compute interpolation equations
  - Matrix/vector product per parameter

KAIST

Massive Models

100,000,000 primitives

<u>1,000,000 pixels</u>

100 visible primitives/pixel

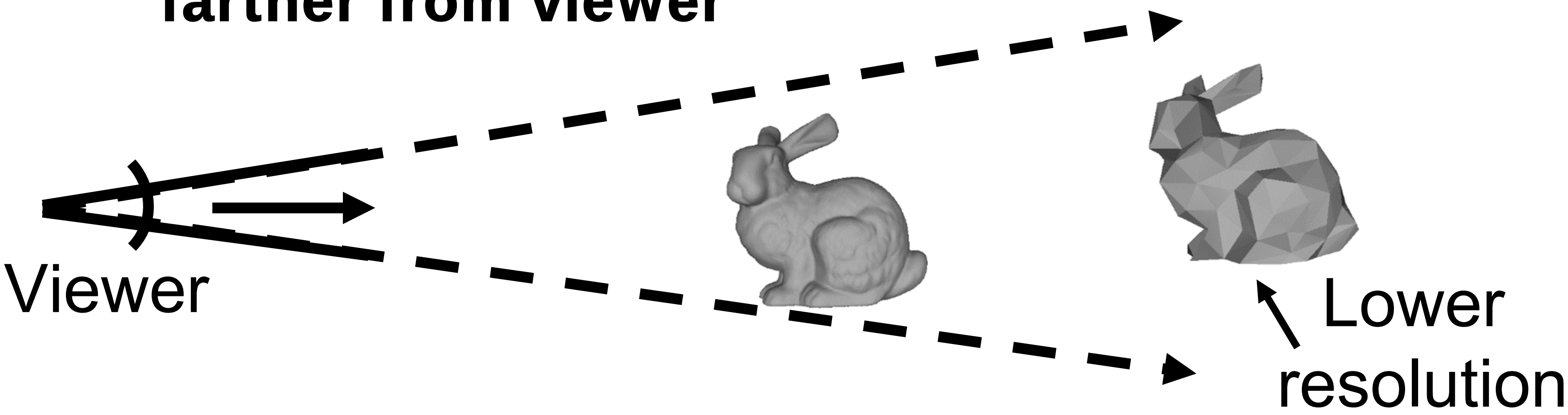
- **Cost to render a single triangle**
  - **Specify 3 vertices**
  - **Compute 3 edge equations**
  - **Evaluate equations one**

St. Mathew models consisting of about 400M triangles (Michelangelo Project)

KAIST

Multi-Resolution or Levels-of-Detail (LOD) Techniques

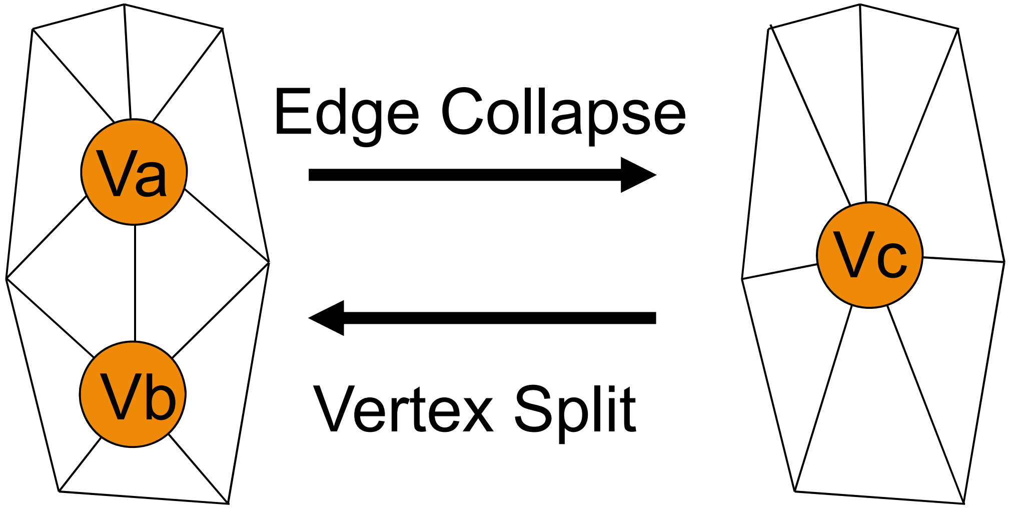
- **Basic idea**
  - **Render with fewer triangles when model is farther from viewer**



- **Methods**
  - **Polygonal simplification**

Polygonal Simplification

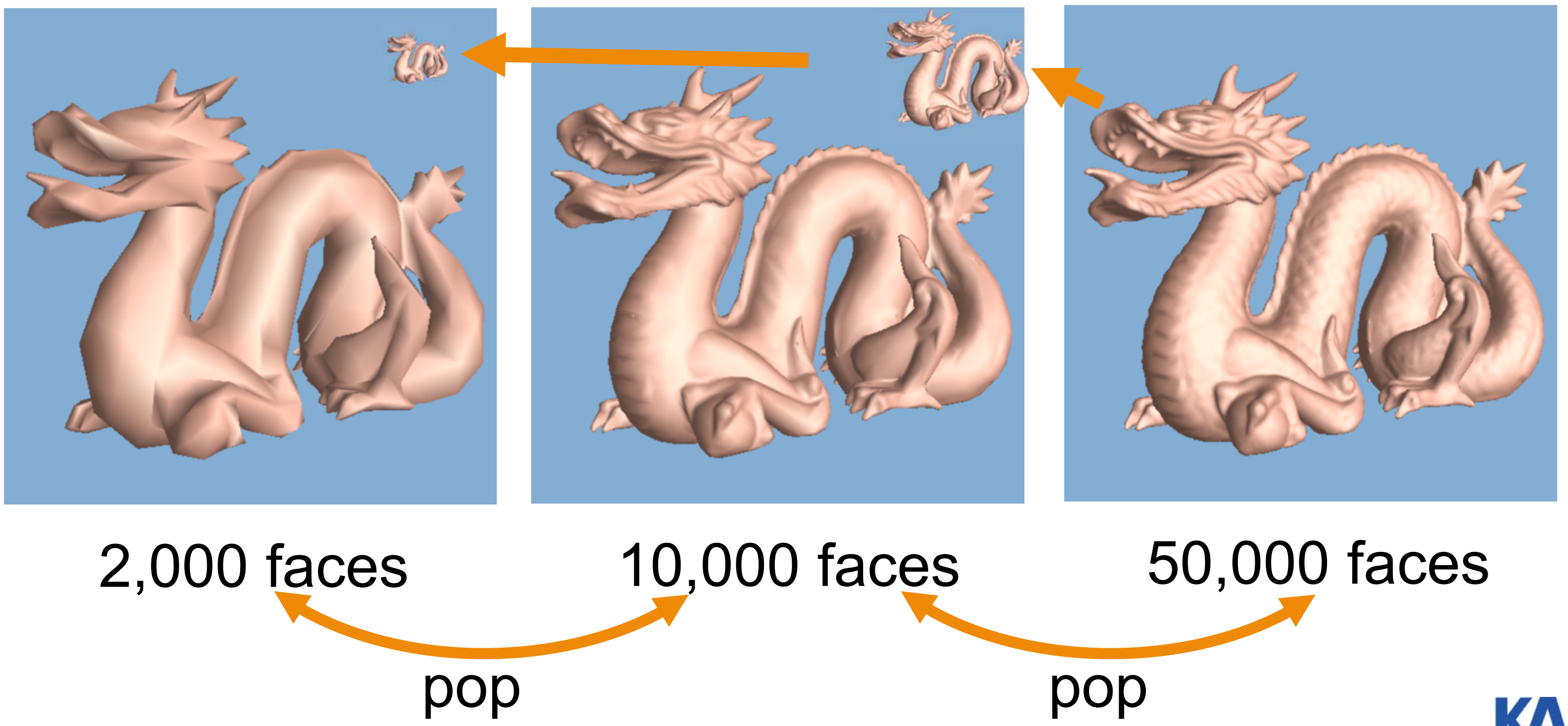
- **Method for reducing the polygon count of mesh**



KAIST

Static LODs

- Pre-compute discrete simplified meshes
  - Switch between them at runtime
  - Has very low LOD selection overhead

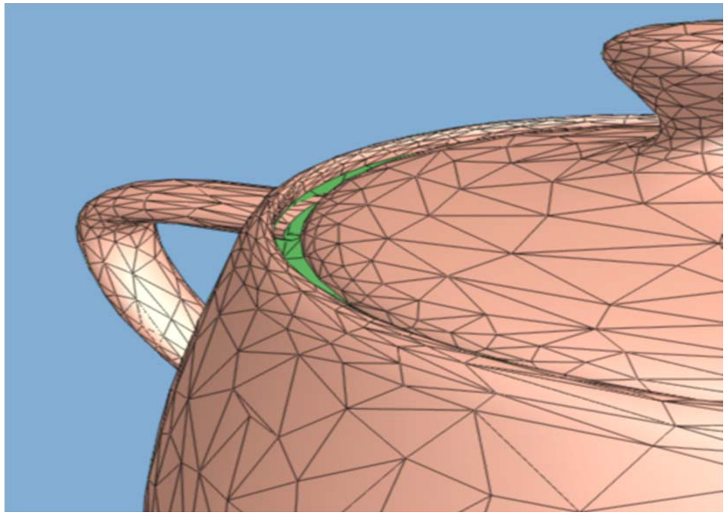


Excerpted from Hoppe's slides

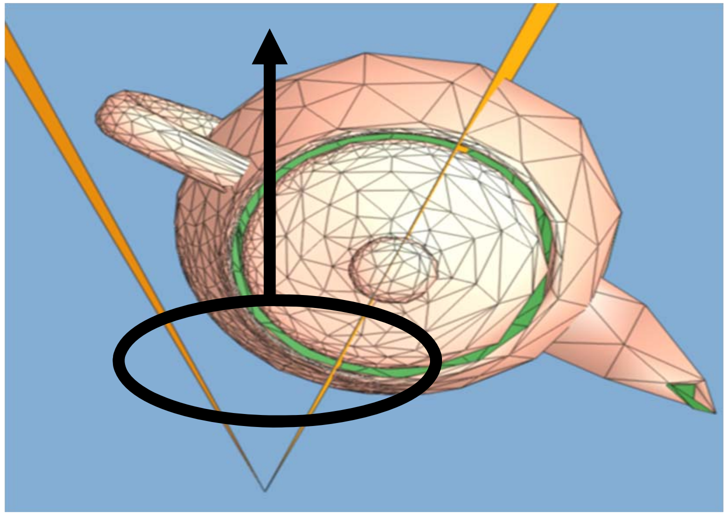
Dynamic Simplification

- **Provides smooth and varying LODs over the mesh [Hoppe 97]**

1st person's view



3rd person's view



Play video

View-Dependent Rendering [Yoon et al., SIG 05]

30 Pixels of error

Pentium 4

GeForce Go 6800 Ultra

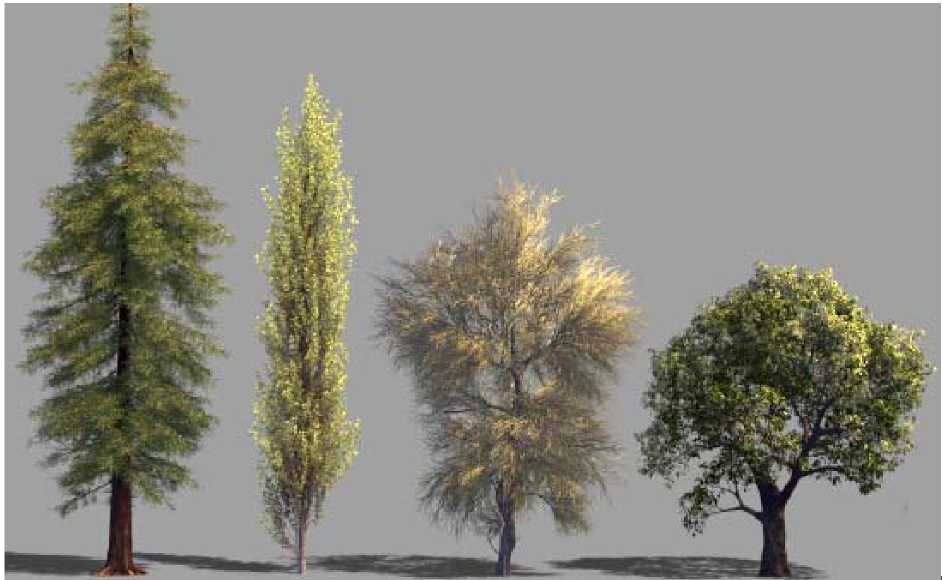
1GB RAM

KAIST

What if there are so many objects?

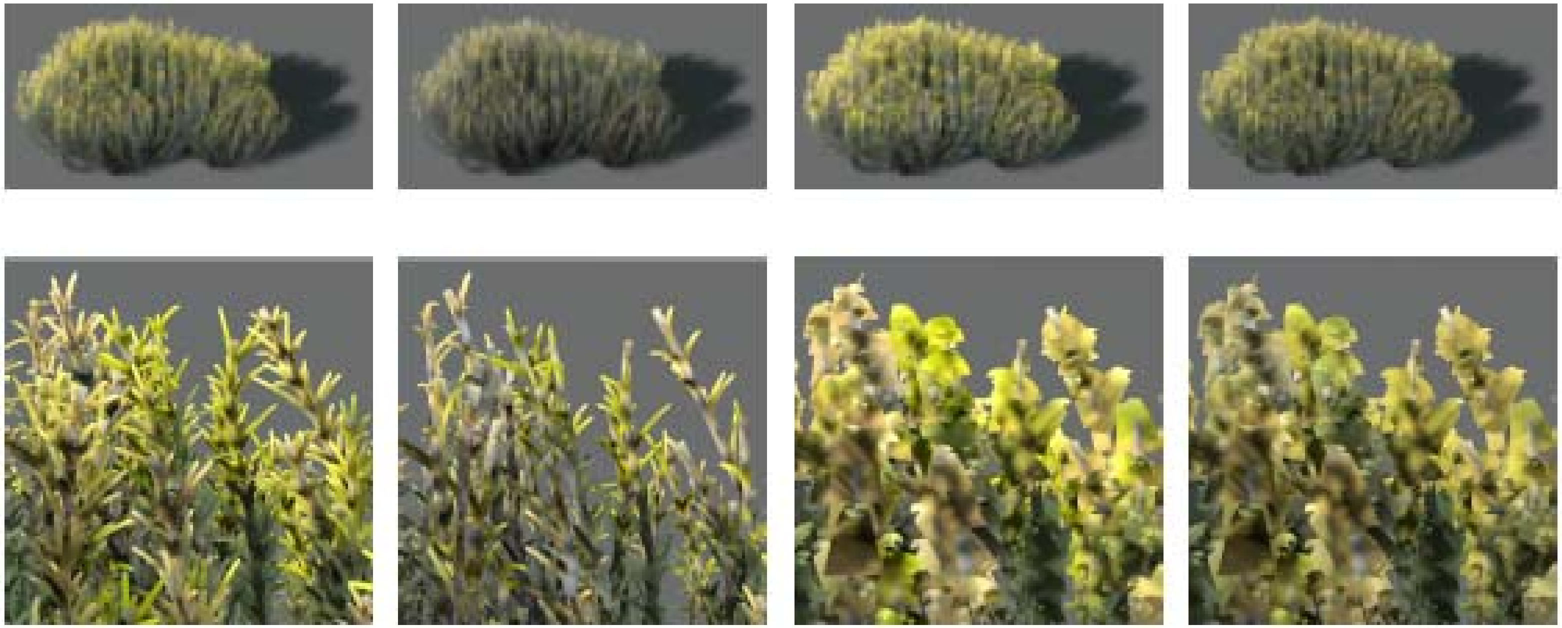
From "cars", a Pixar movie

What if there are so many objects?



From a Pixar movie

Stochastic Simplification of Aggregate Detail
Cook et al., ACM SIGGRAPH 2007

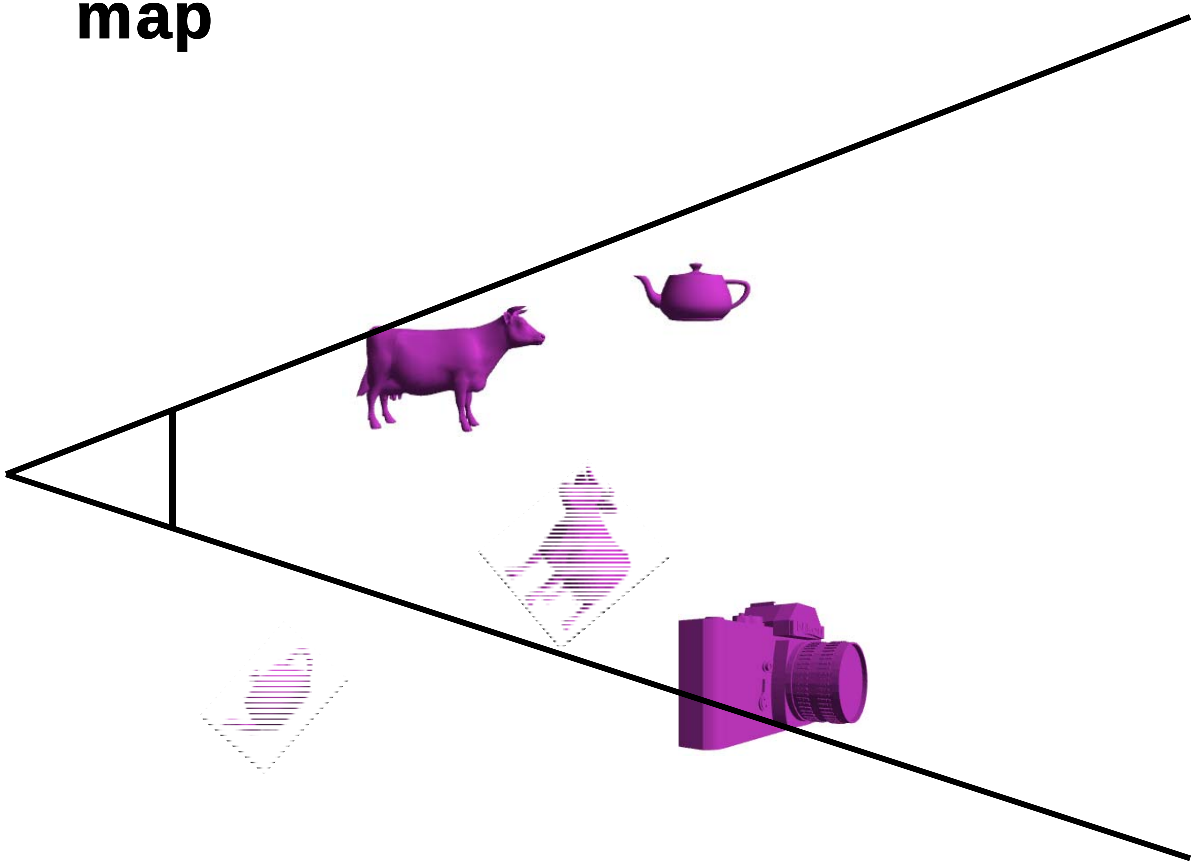


(a) (b) (c) (d)

Figure 2: *Distant views of the plant from Figure 1 with close-ups below: (a) unsimplified, (b) with 90% of its leaves excluded, (c) with area correction, (d) with area and contrast correction.*

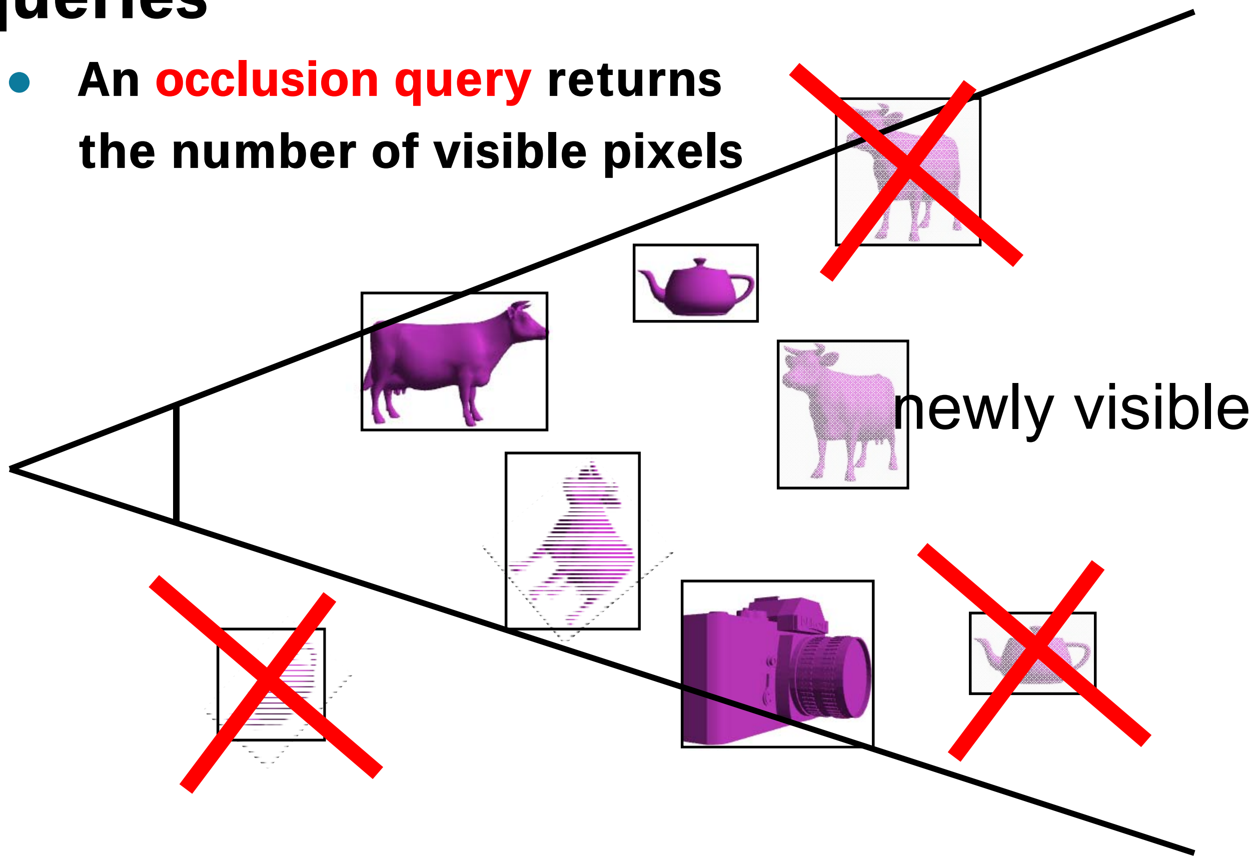
Occlusion Culling with Occlusion Queries

- Render objects visible in previous frame
  - Known as occlusion representation or occlusion map



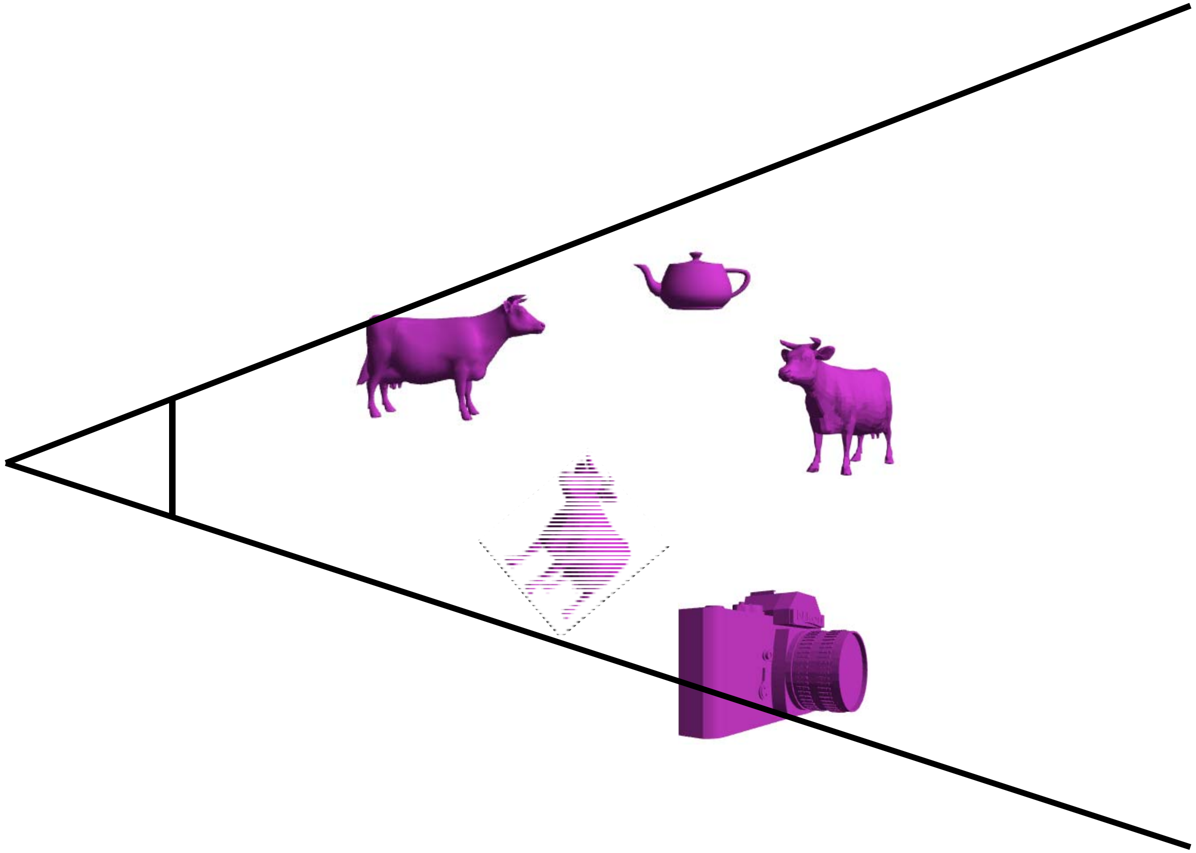
Occlusion Culling with Occlusion Queries

- **Turn off color and depth writes**
- **Render object bounding boxes with occlusion queries**
  - **An occlusion query returns the number of visible pixels**



Occlusion Culling with Occlusion Queries

- Re-enable color writes
- Render newly visible objects



Class Objectives were:

- Understand triangle rasterization using edge-equations
- Understand mechanics for parameter interpolations
- Realize benefits of incremental algorithms

Next Time

- Illumination and shading
- Texture mapping

Homework

- **Go over the next lecture slides before the class**
- **Watch 2 SIGGRAPH videos and submit your summaries before every Tue. class**
  - **Send an email to cs380ta@gmail.com**
  - **Just one paragraph for each summary**

KAIST

Any Questions?

- Come up with one question on what we have discussed in the class and submit at the end of the class
  - 1 for already answered questions
  - 2 for typical questions
  - 3 for questions with thoughts or that surprised me

- Submit at least four times during the whole semester

KAIST