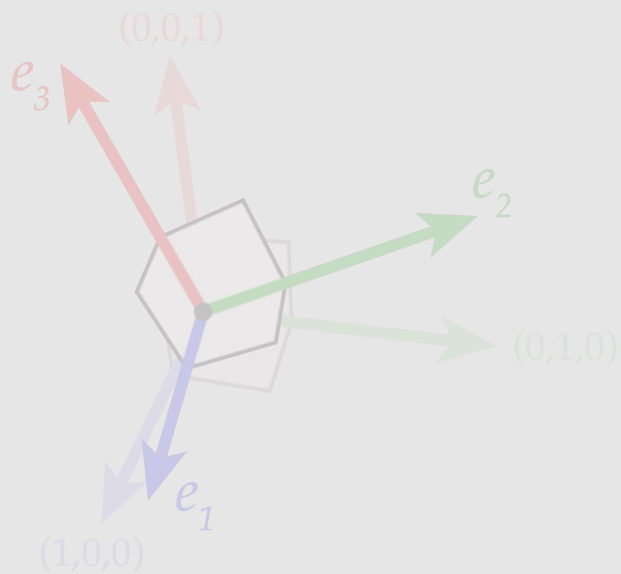


Midterm Review

Midterm Overview

- 80 minutes, during class next Thursday
 - Basic mathematical questions, no intense calculation
 - Know your definitions and be able to apply them!
 - No pseudocode
 - Review slides are a good hint as to what might be on the exam :)
- 5 questions
 - Total 125 points available
 - Graded out of 100
 - Q1-4 from these slides
- Cheat sheet: one 3x3 inch note (about the size of a post it note) front and back
- **Please bring a blue/black pen to write your solutions**

3D Inverse Rotations



$$R^T = \begin{bmatrix} \text{---} e_1^T \text{---} \\ \text{---} e_2^T \text{---} \\ \text{---} e_3^T \text{---} \end{bmatrix} \quad R = \begin{bmatrix} | & | & | \\ e_1 & e_2 & e_3 \\ | & | & | \end{bmatrix}$$

$$= \begin{bmatrix} \text{diagrams of dot products} \end{bmatrix} = \begin{bmatrix} e_1^T e_1 & e_1^T e_2 & e_1^T e_3 \\ e_2^T e_1 & e_2^T e_2 & e_2^T e_3 \\ e_3^T e_1 & e_3^T e_2 & e_3^T e_3 \end{bmatrix}$$

$$R^T R = I \Rightarrow R^T = R^{-1}$$

$$= \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

If you need to review any slides more in depth, look here for which lecture it came from

- Transformations Review
- Rasterization Review
- Geometry Review
- Spatial Data Structures Review

Transformations

- Homogeneous coordinates
- 3D Translation
- 3D Scale
- 3D Rotation
 - Axis-Aligned rotation
 - Axis-Angle rotation
 - Rotations from orthonormal bases

3D Transforms in Homogeneous Coordinate

$$\begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

[point in 3D]

Matrix representations of 3D linear transformations just
get an additional identity row/column:

$$\begin{bmatrix} \cos \theta & 0 & \sin \theta & 0 \\ 0 & 1 & 0 & 0 \\ -\sin \theta & 0 & \cos \theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & s & 0 \\ 0 & 1 & t & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & c & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & u \\ 0 & 1 & 0 & v \\ 0 & 0 & 1 & w \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

[rotate around y by θ]

[shear by z in (s,t) direction]

[scale by a,b,c]

[translate by (u,v,w)]

Translation in Homogeneous Coordinates

- A 2D translation is similar to a 3D shear
 - Moving a slice up/down the shear moves the shape

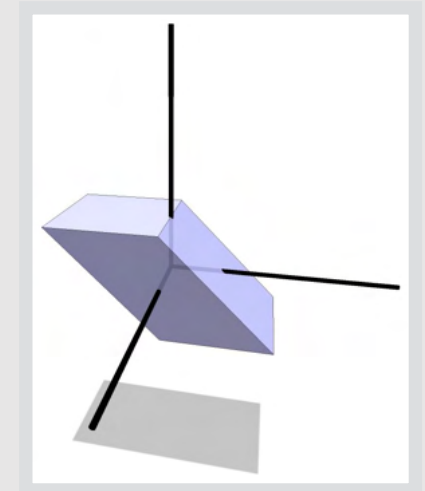
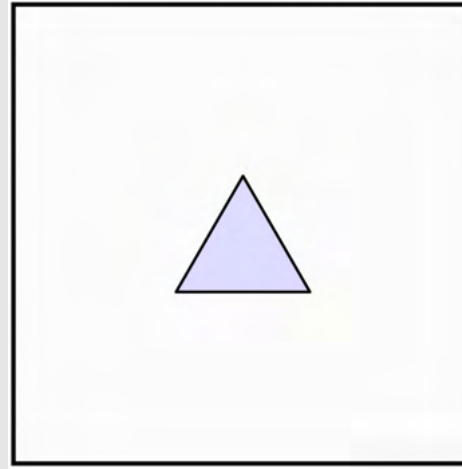
- Recall shear is written as:

$$f_{\mathbf{u},\mathbf{v}}(\mathbf{x}) = \mathbf{x} + \langle \mathbf{v}, \mathbf{x} \rangle \mathbf{u}$$

$$f_{\mathbf{u},\mathbf{v}}(\mathbf{x}) = (I + \mathbf{u}\mathbf{v}^T)\mathbf{x}$$

- In our case, $\mathbf{v} = (0, 0, 1)$, so**

$$\begin{bmatrix} 1 & 0 & u_1 \\ 0 & 1 & u_2 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} cp_1 \\ cp_2 \\ c \end{bmatrix} = \begin{bmatrix} c(p_1 + u_1) \\ c(p_2 + u_2) \\ c \end{bmatrix} \xrightarrow{1/c} \begin{bmatrix} p_1 + u_1 \\ p_2 + u_2 \end{bmatrix}$$



**most often in this class we will also use $c = 1$

Non-Uniform Scaling

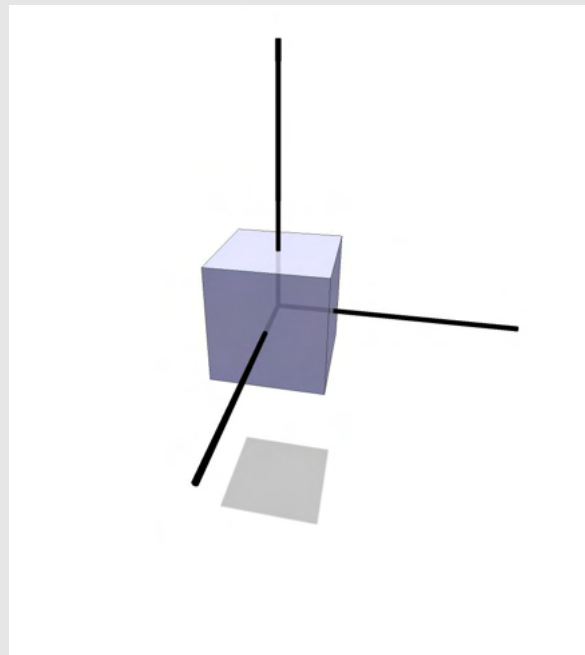
- To scale a vector u by a non-uniform amount (a, b, c) :

$$\begin{bmatrix} a & 0 & 0 \\ 0 & b & 0 \\ 0 & 0 & c \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} au_1 \\ bu_2 \\ cu_3 \end{bmatrix}$$

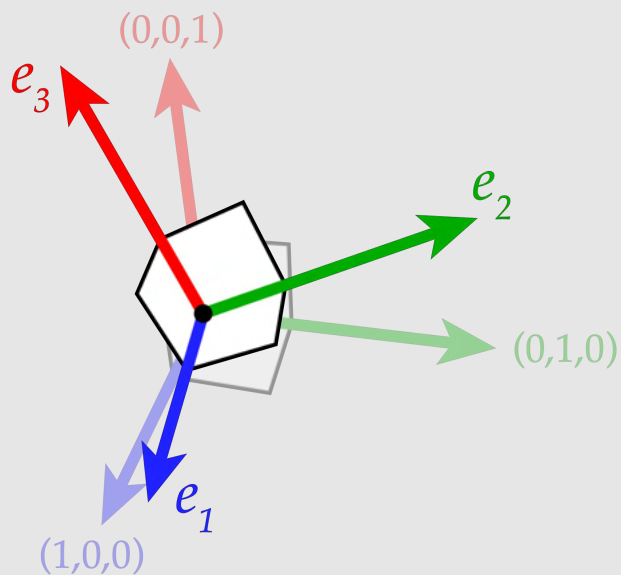
- The above works only if scaling is axis-aligned. What if it isn't?
- Idea:
 - Rotate to a new axis R
 - Perform axis-aligned scaling D
 - Rotate back to original axis R^T

$$A := R^T D R$$

- Resulting transform A is a symmetric matrix



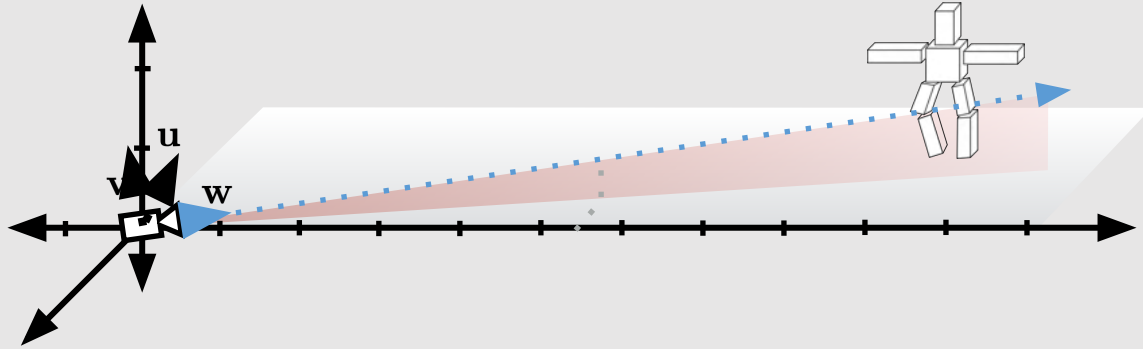
3D Inverse Rotations



$$R^T R = I \Rightarrow R^T = R^{-1}$$

$$\begin{aligned}
 & \begin{matrix} R^T & R \end{matrix} \\
 & \begin{bmatrix} \text{---} e_1^T \text{---} \\ \text{---} e_2^T \text{---} \\ \text{---} e_3^T \text{---} \end{bmatrix} \begin{bmatrix} | & | & | \\ e_1 & e_2 & e_3 \\ | & | & | \end{bmatrix} \\
 & = \begin{bmatrix} \text{diagrams of dot products} \end{bmatrix} = \begin{bmatrix} e_1^T e_1 & e_1^T e_2 & e_1^T e_3 \\ e_2^T e_1 & e_2^T e_2 & e_2^T e_3 \\ e_3^T e_1 & e_3^T e_2 & e_3^T e_3 \end{bmatrix} \\
 & = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}
 \end{aligned}$$

Rotations From Orthonormal Bases



$$R = \begin{bmatrix} -u_x & v_x & -w_x \\ -u_y & v_y & -w_y \\ -u_z & v_z & -w_z \end{bmatrix}$$

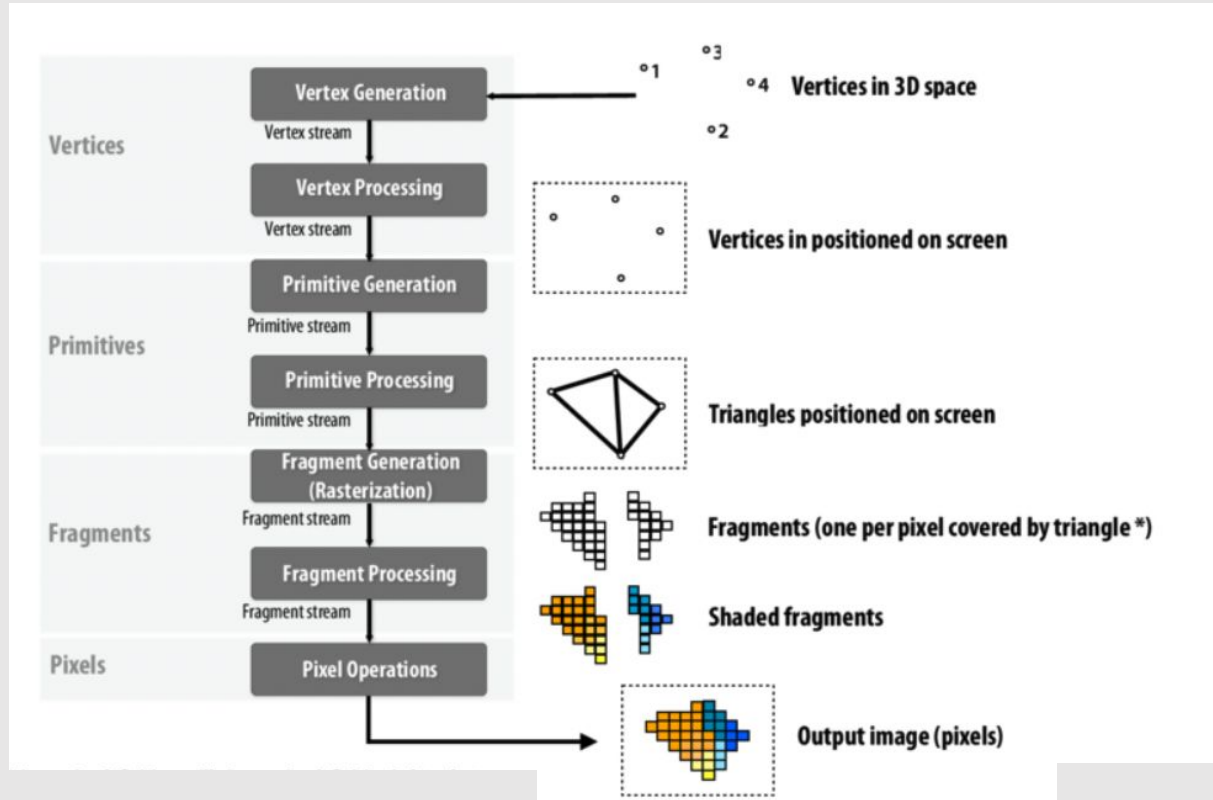
$$R^{-1} = \begin{bmatrix} -u_x & -u_y & -u_z \\ v_x & v_y & v_z \\ -w_x & -w_y & -w_z \end{bmatrix}$$

- Transformations Review
- Rasterization Review
- Geometry Review
- Spatial Data Structures Review

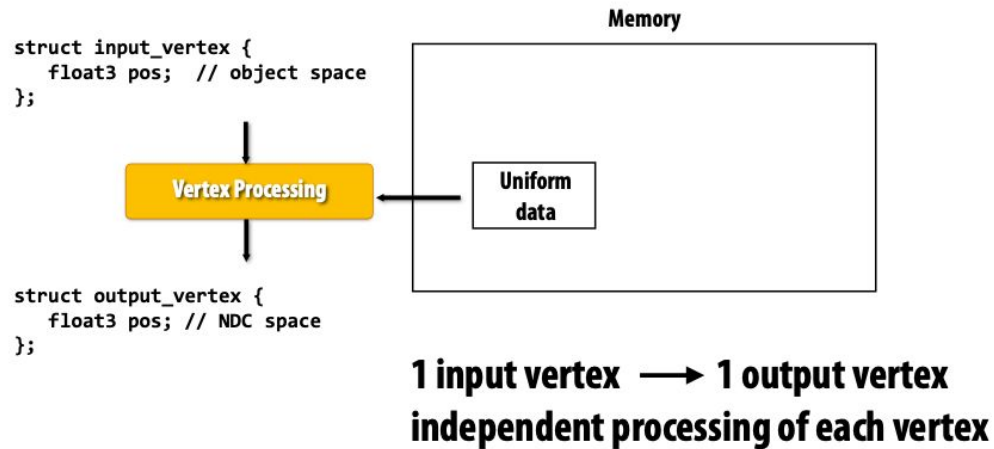
Rasterization

- The “simpler” graphics pipeline
- Scene graph
- Clipping
- Rasterization
 - Sampling
 - Point-in-triangle tests
 - Barycentric coordinates
- Textures
- Depth and Alpha blending

The Graphics Pipeline



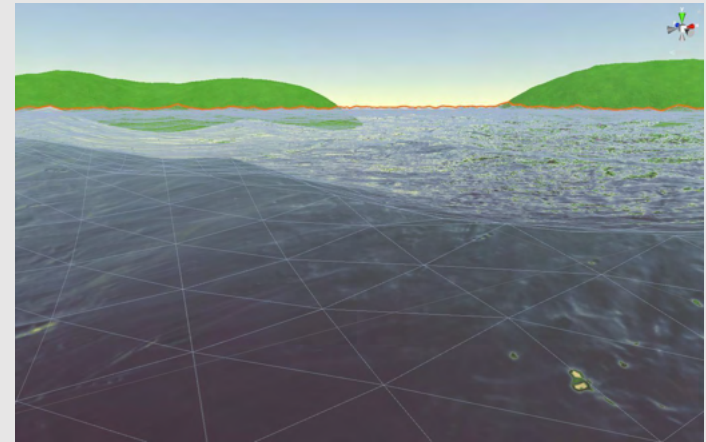
Vertex Shader



Vertex Shader Program *

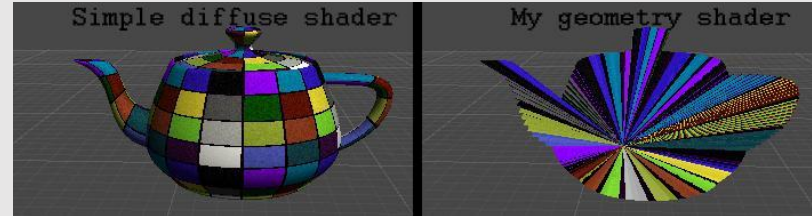
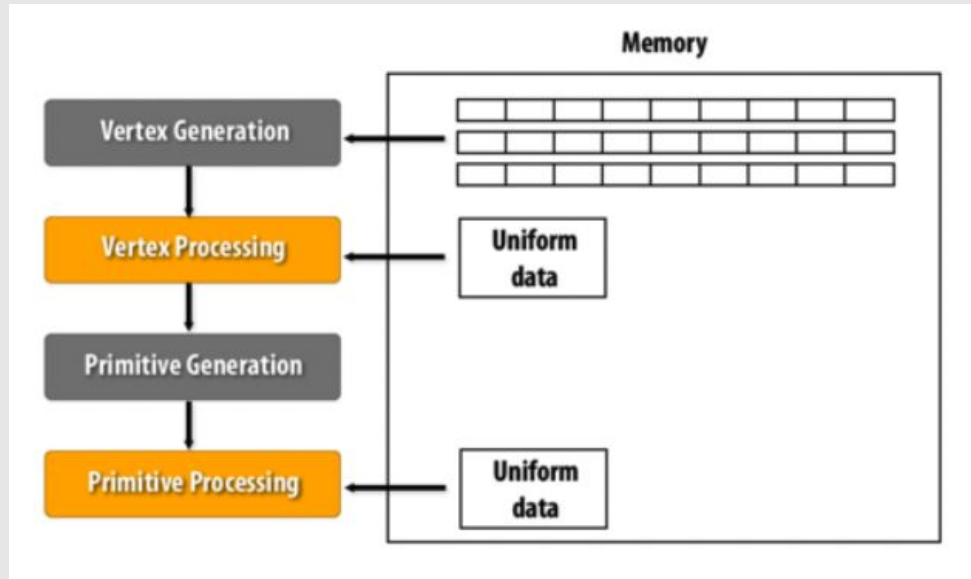
```
uniform mat4 my_transform; // P * T  
  
output_vertex my_vertex_program(input_vertex in) {  
    output_vertex out;  
    out.pos = my_transform * in.pos; // matrix-vector mult  
    return out;  
}
```

* (Note this is pseudocode, not GLSL syntax)



- Vertex shaders provide per-vertex operations that change attributes of vertices such as their positions and normal
 - Example: fluid simulation

Geometry Shader

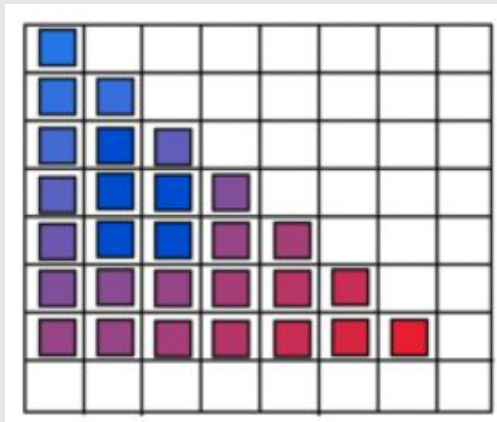
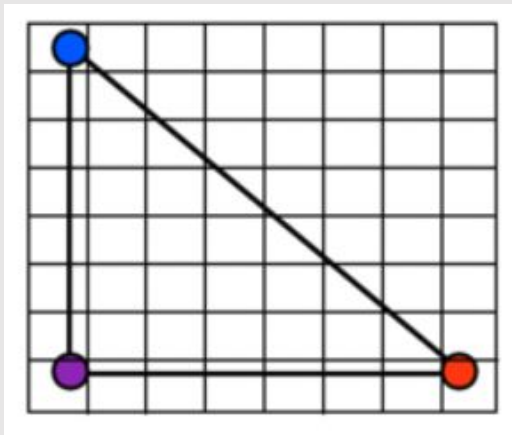
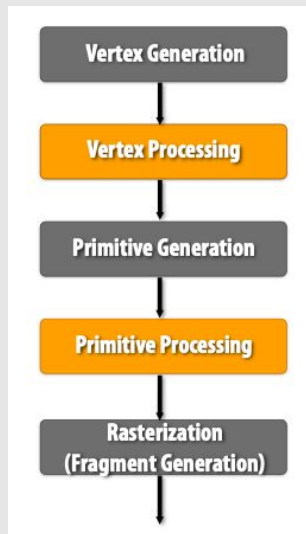


Some guy in Unity having a hard time with geometry shaders (2014)

- Recently added to OpenGL in 2007
 - As such, not many people use it
- Allows user to retarget the connectivity of their geometry by specifying tessellation operations or adding in additional geometry
 - Example: computing vertex shader on coarse geometry and then subdividing a surface in the geometry shader

* "Geometry Shader" in OpenGL/Direct3D/Metal terminology

Fragment Shader

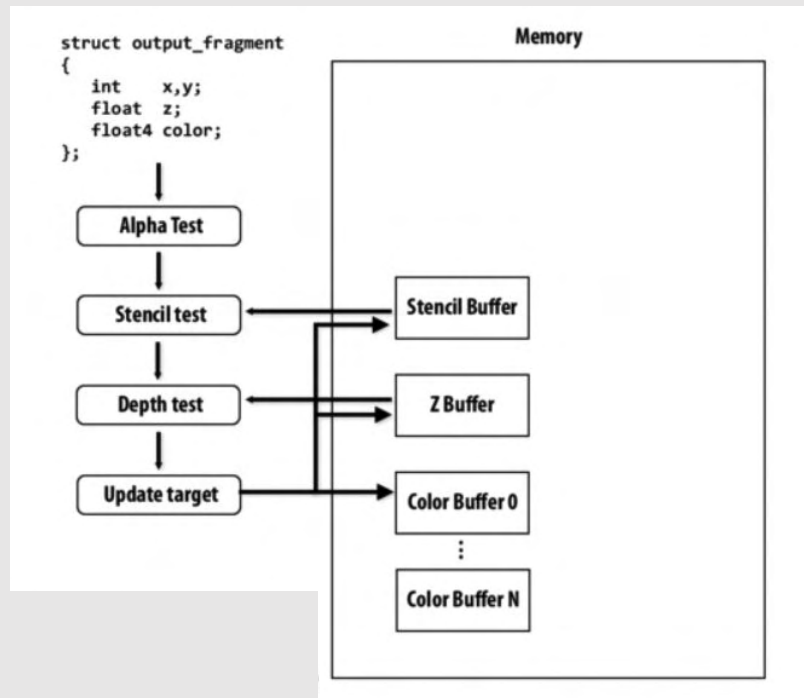


```
struct fragment    // note similarity to output_vertex from before
{
    float  x,y;    // screen pixel coordinates (sample point location)
    float  z;      // depth of triangle at sample point

    float3 normal; // interpolated application-defined attribs
    float2 texcoord; // (e.g., texture coordinates, surface normal)
}
```

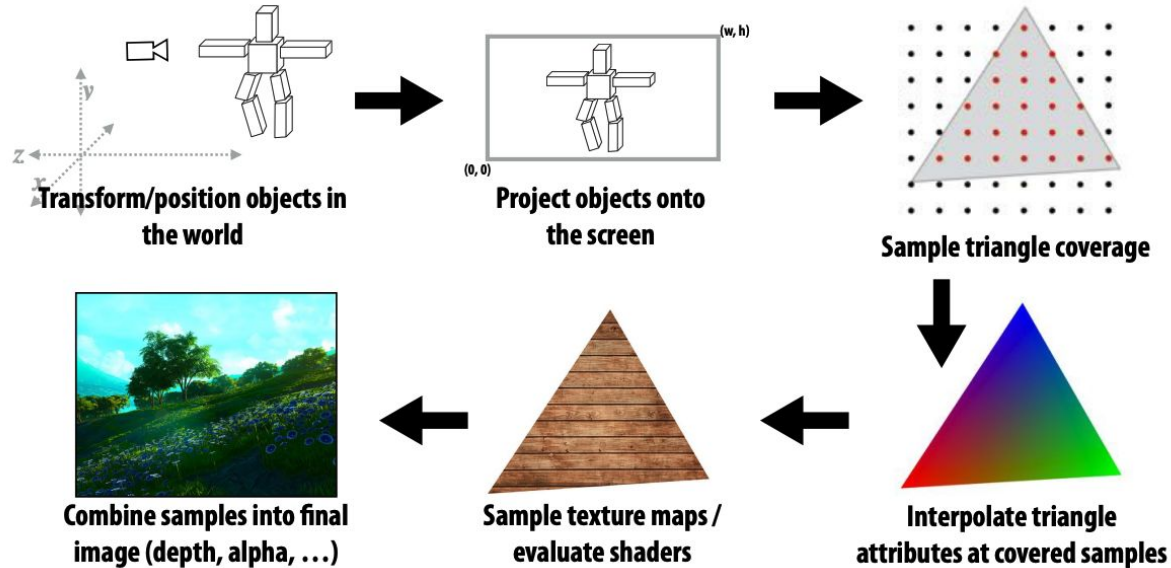
- Separate triangles into fragments
- For each fragment, compute the output RGBA

Frame-Buffer Operations



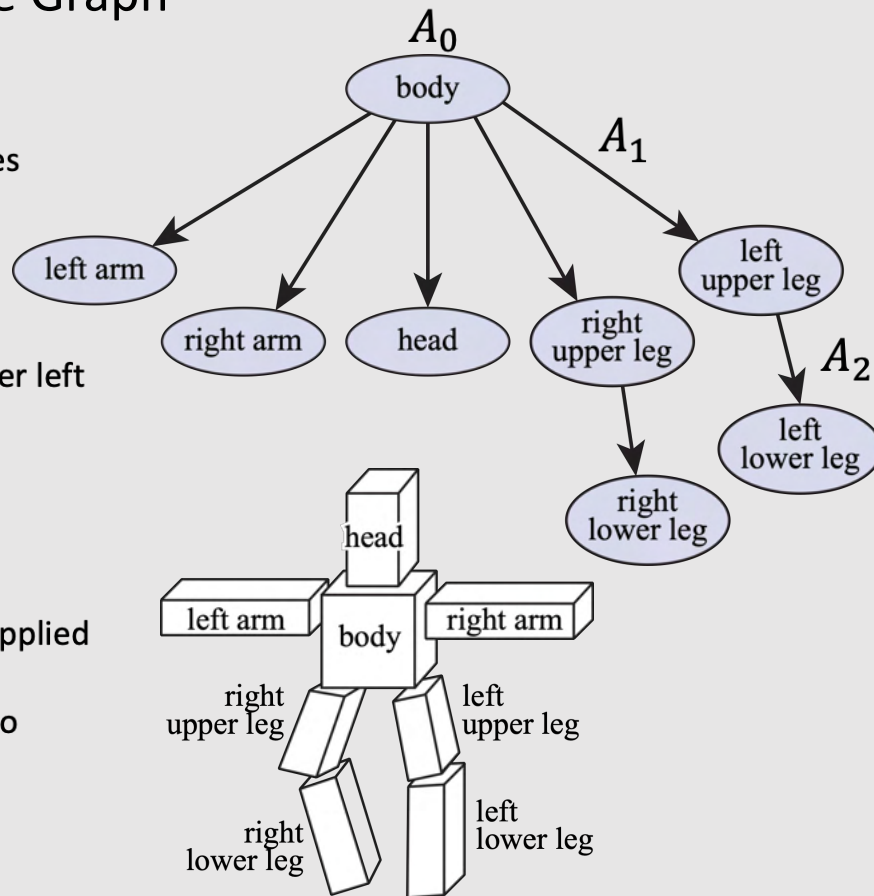
- **Alpha Test**
 - Allows fragments with alpha value greater/less than a constant specified by the user to pass
- **Stencil Test**
 - Allows fragments that pass a user-defined per-pixel function to pass
 - Stored in stencil buffer
 - Example: mattes + masking
- **Depth Test**
 - Allows fragments that are closest in depth to pass
 - Stored in Z-buffer
- **Update Target**
 - If pixel passes, modify stencil depth and color buffers
 - Reads can be done in parallel, writes require locking
 - We'll look at techniques later to accelerate this

The “Simpler” Graphics Pipeline



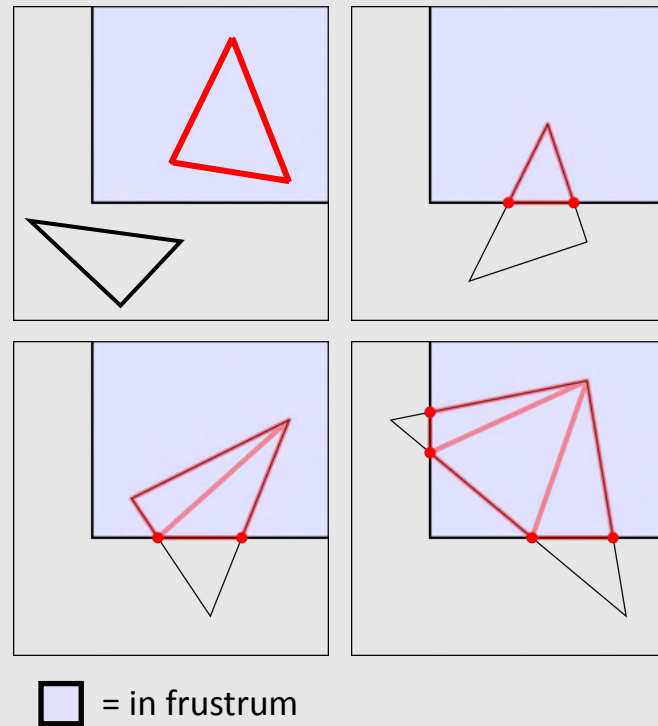
Scene Graph

- Suppose we want to build a skeleton out of cubes
- **Idea:** transform cubes in world space
 - Store transform of each cube
- **Problem:** If we rotate the left upper leg, the lower left leg won't track with it
 - **Better Idea:** store a hierarchy of transforms
 - Known as a **scene graph**
 - Each edge (+root) stores a linear transformation
 - Composition of transformations gets applied to nodes
 - Keep transformations on a stack to reduce redundant multiplication
- **Lower left leg transform:** $A_2A_1A_0$



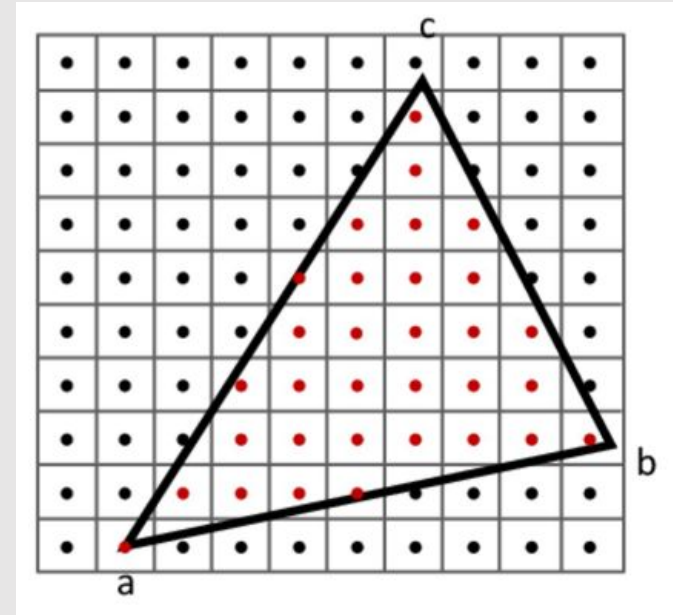
Clipping

- **Clipping** eliminates triangles not visible to the camera (not in view frustum)
 - Don't waste time rasterizing primitives you can't see!
 - Discarding individual fragments is expensive
 - “Fine granularity”
 - Makes more sense to toss out whole primitives
 - “Coarse granularity”
- What if a primitive is **partially clipped**?
 - Partially enclosed primitives are triangulated into non-overlapping smaller triangles that fit in the frustum
- If part of a triangle is outside the frustum, it means at least one of its vertices are outside the frustum
 - **Idea:** check which side of halfspaces the vertices are at

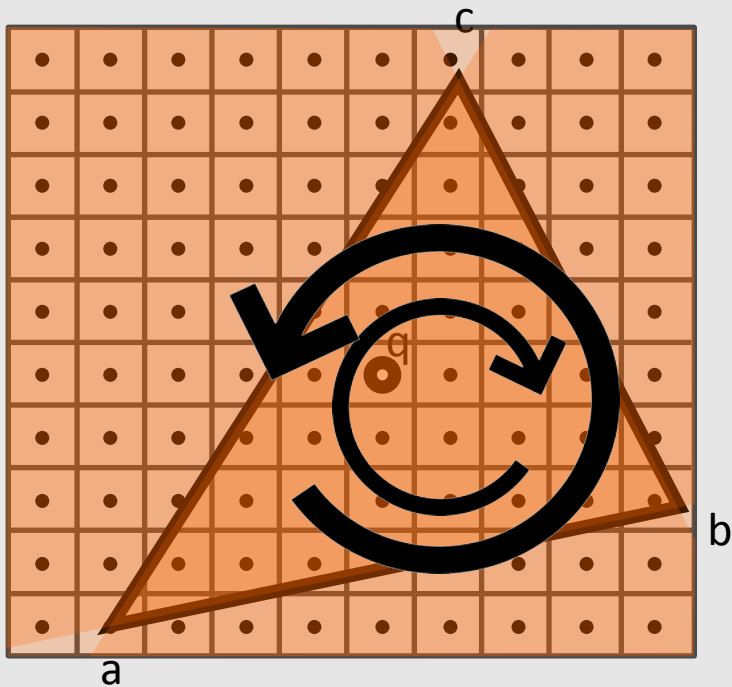


Rasterization

- Triangle
 - Bounding box
 - Incremental triangle traversal
 - Hierarchical coverage
- For each **Primitive** (Triangle):
 - For each **Pixel**:
 - If **Pixel** in **Primitive**:
 - Pixel color = Interpolated triangle color



Point-In-Triangle Test



- **Measurements must all either be positive or negative** for point to be in triangle

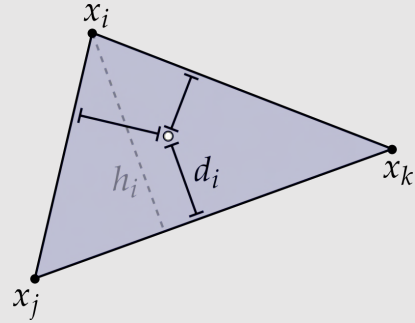
$$\begin{aligned}(\vec{ac} \times \vec{ab}) \cdot (\vec{ac} \times \vec{aq}) &> 0 \ \&\& \\ (\vec{cb} \times \vec{ca}) \cdot (\vec{cb} \times \vec{cq}) &> 0 \ \&\& \\ (\vec{ba} \times \vec{bc}) \cdot (\vec{ba} \times \vec{bq}) &> 0\end{aligned}$$

OR

$$\begin{aligned}(\vec{ab} \times \vec{ac}) \cdot (\vec{ac} \times \vec{aq}) &< 0 \ \&\& \\ (\vec{ca} \times \vec{cb}) \cdot (\vec{cb} \times \vec{cq}) &< 0 \ \&\& \\ (\vec{bc} \times \vec{ba}) \cdot (\vec{ba} \times \vec{bq}) &< 0\end{aligned}$$

- Orientation no longer matters
 - Just be consistent!

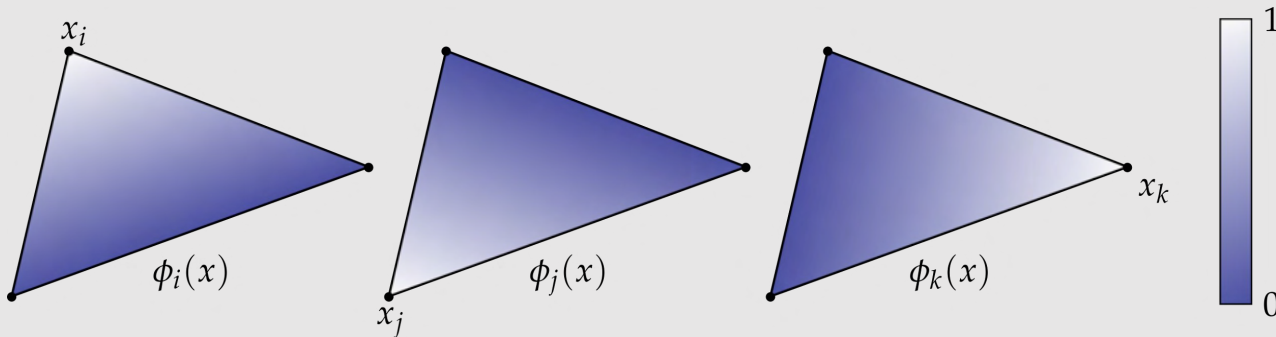
Barycentric Coordinates



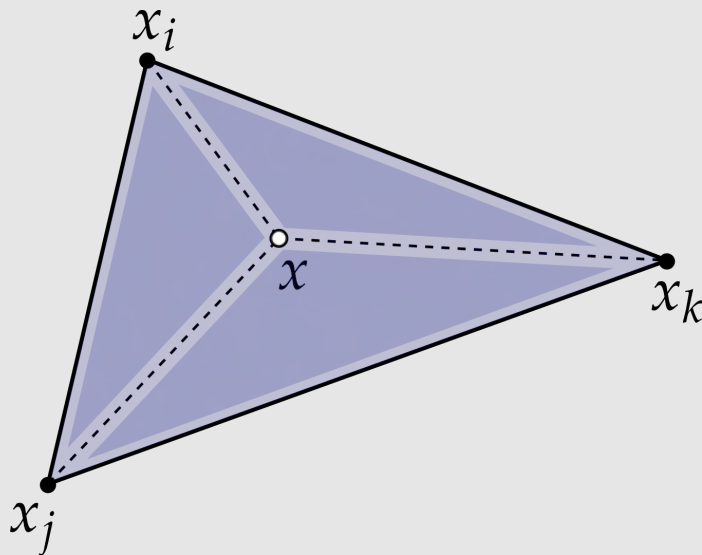
- Inversely proportional to the signed distance between the target point and a point within the triangle
- Can be computed as:

$$\phi_i(x) = d_i(x) / h_i$$

- How would you compute h_i ? $d_i(x)$?



Barycentric Coordinates [Another Way]



- Directly proportional to the signed area created by the triangle composed of the other two target points and a point within the triangle
- Can be computed as:

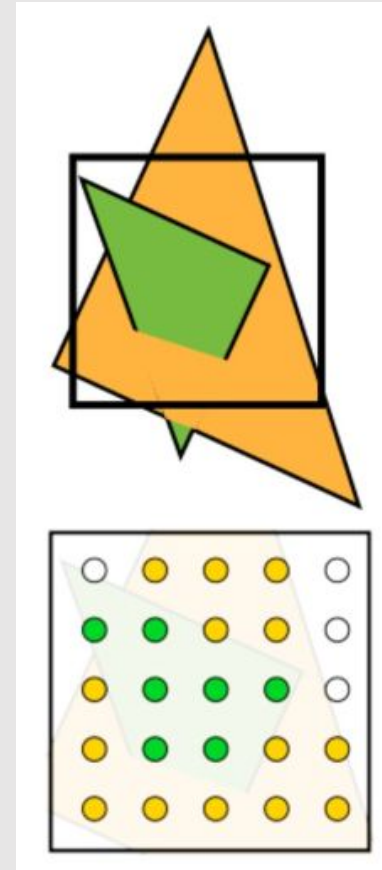
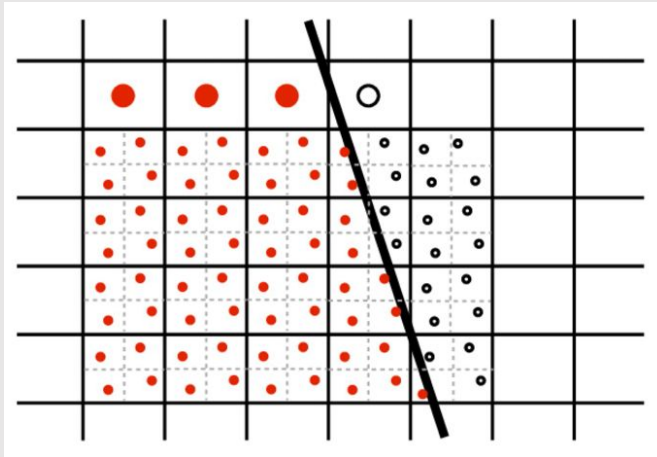
$$\phi_i(x) = \frac{\text{area}(x, x_j, x_k)}{\text{area}(x_i, x_j, x_k)}$$

- Note that signed distance / area implies barycentric coordinates can be negative, but they will still sum to 1! (if on the same plane, otherwise we project point to the plane containing our triangle)

** Interesting read of barycentric coordinates for n-gons:
<https://www.inf.usi.ch/hormann/barycentric/>

Coverage via Samples

- Sample : Discrete measurement of a signal
 - Multisampling vs Supersampling
- Approximate the coverage of the area of a pixel by taking n samples
 - Per sample coverage & depth test + texture lookup + alpha blending

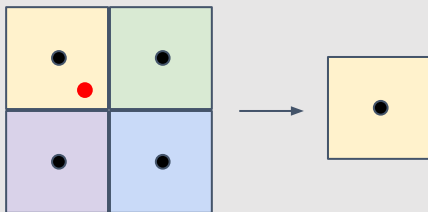


Nearest Neighbor Sampling

- **Idea:** Grab texel nearest to requested location in texture
- **Requires:**
 - 1 memory lookup
 - 0 linear interpolations

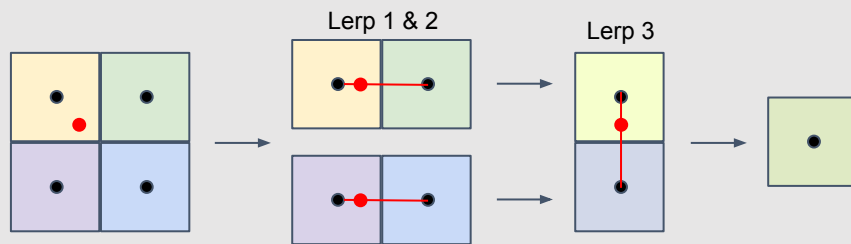
$$x' \leftarrow \text{round}(x - 0.5), \quad y' \leftarrow \text{round}(y) - 0.5$$

$$t \leftarrow \text{tex.lookup}(x', y')$$



Bilinear Interpolation Sampling

- **Idea:** Grab nearest 4 texels and blend them together based on their inverse distance from the requested location
 - Blend two sets of pixels along one axis, then blend the remaining pixels
- **Requires:**
 - 4 memory lookup
 - 3 linear interpolations



$$x' \leftarrow \text{floor}(x - 0.5), \quad y' \leftarrow \text{floor}(y - 0.5)$$

$$\Delta x \leftarrow (x - 0.5) - x'$$
$$\Delta y \leftarrow (y - 0.5) - y'$$

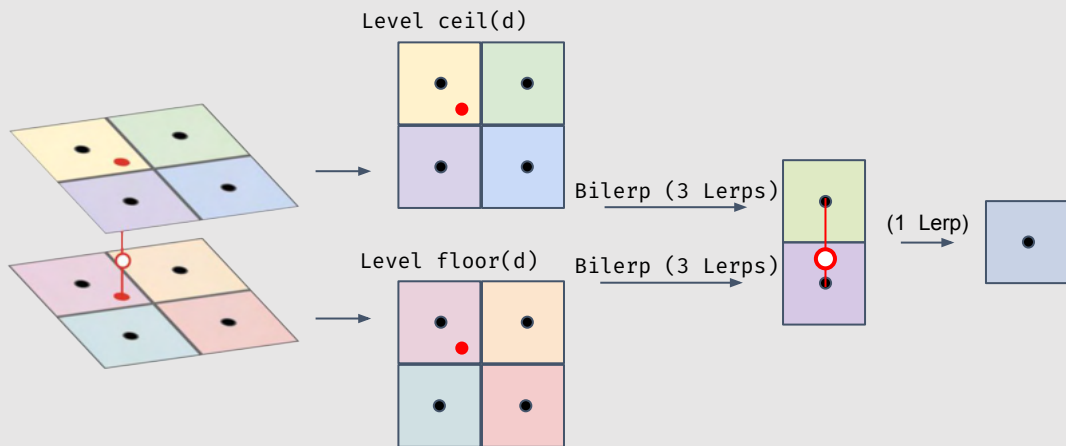
$$t_{(x,y)} \leftarrow \text{tex.lookup}(x', y')$$
$$t_{(x+1,y)} \leftarrow \text{tex.lookup}(x' + 1, y')$$
$$t_{(x,y+1)} \leftarrow \text{tex.lookup}(x', y' + 1)$$
$$t_{(x+1,y+1)} \leftarrow \text{tex.lookup}(x' + 1, y' + 1)$$

$$t_x \leftarrow (1 - \Delta x) * t_{(x,y)} + \Delta x * t_{(x+1,y)}$$
$$t_y \leftarrow (1 - \Delta x) * t_{(x,y+1)} + \Delta x * t_{(x+1,y+1)}$$

$$t \leftarrow (1 - \Delta y) * t_x + \Delta y * t_y$$

Trilinear Interpolation Sampling

- **Idea:** Perform bilinear interpolation on two layers of the mip-map that represents proper minification/magnification, blending the results together
- **Requires:**
 - 8 memory lookup
 - 7 linear interpolations



$$L_x^2 \leftarrow \frac{du^2}{dx} + \frac{dv^2}{dx}$$

$$L_y^2 \leftarrow \frac{du^2}{dy} + \frac{dv^2}{dy}$$

$$L \leftarrow \sqrt{\max(L_x^2, L_y^2)}$$

$$d \leftarrow \log_2 L$$

$$d' \leftarrow \text{floor}(d)$$

$$\Delta d \leftarrow d - d'$$

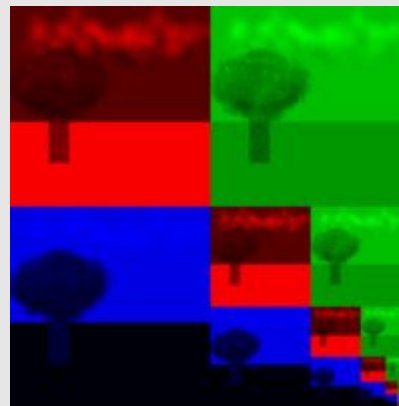
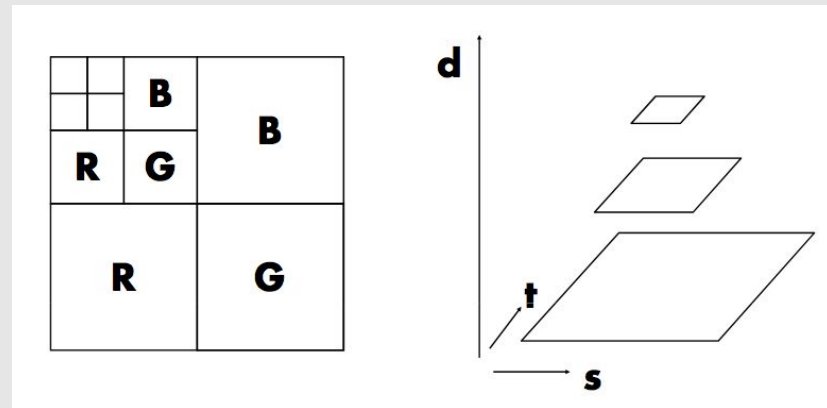
$$t_d \leftarrow \text{tex}[d']. \text{bilinear}(x, y)$$

$$t_{d+1} \leftarrow \text{tex}[d' + 1]. \text{bilinear}(x, y)$$

$$t \leftarrow (1 - \Delta d) * t_d + \Delta d * t_{d+1}$$

Mip-Map [L. Williams '83]

- Storing an RGB Mip-Map can be fit into an image twice the width and twice the height of the original image
 - See diagram for proof :)
 - Does not work as nicely for RGBA!
- **Issue:** bad spatial locality
 - Requesting a texel requires lookup in 3 very different regions of an image



Anisotropic Filtering

- **Anisotropic** filtering is dependent on direction
 - *an* – not, *iso* – same, *tropic* – direction
- **Idea:** create a new texture map that downsamples the x and y axis by 2 separately
 - Instead of taking the max, use each coordinate to index into correct location in map

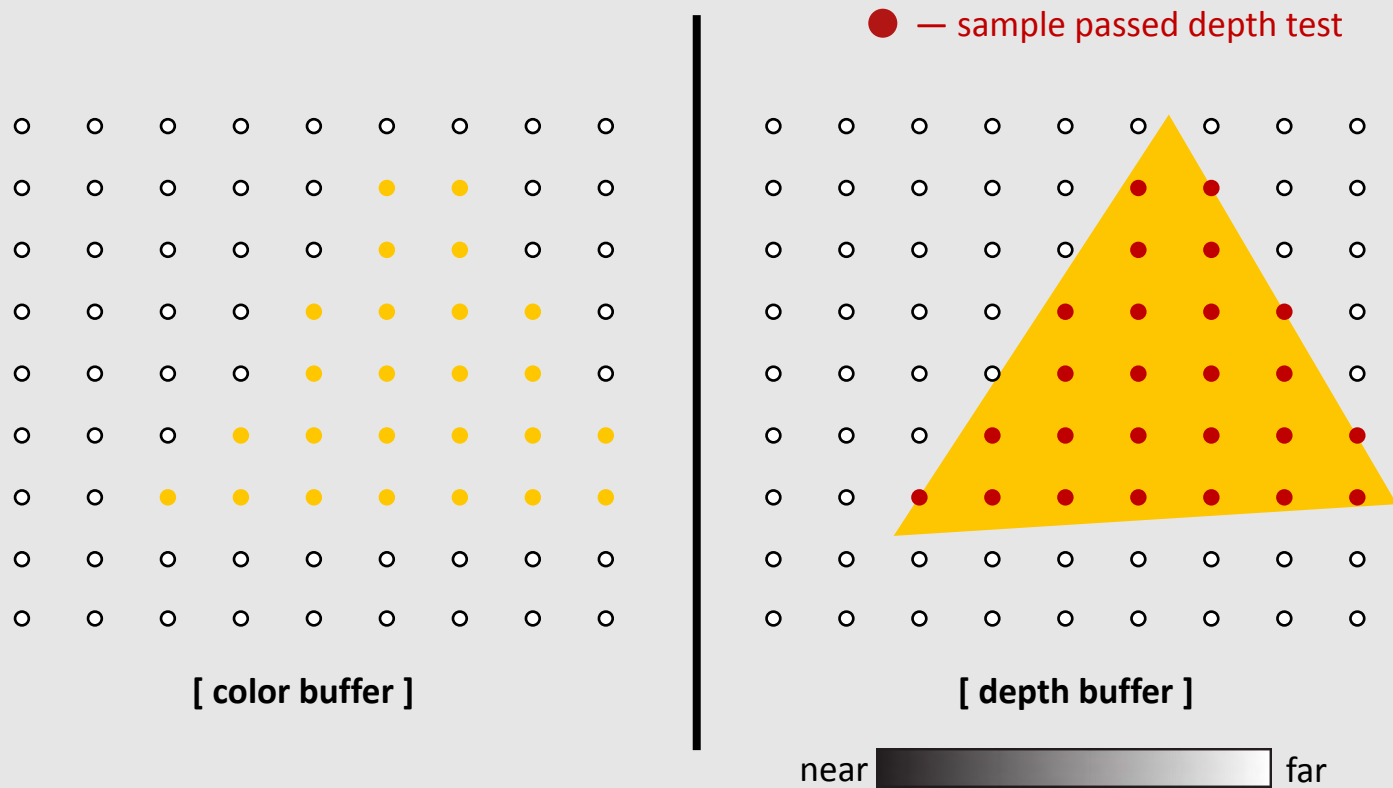
$$L = \sqrt{\log_2(L_x^2, L_y^2)}$$

$$(d_x, d_y) = (\log_2 \sqrt{L_x^2}, \log_2 \sqrt{L_y^2})$$

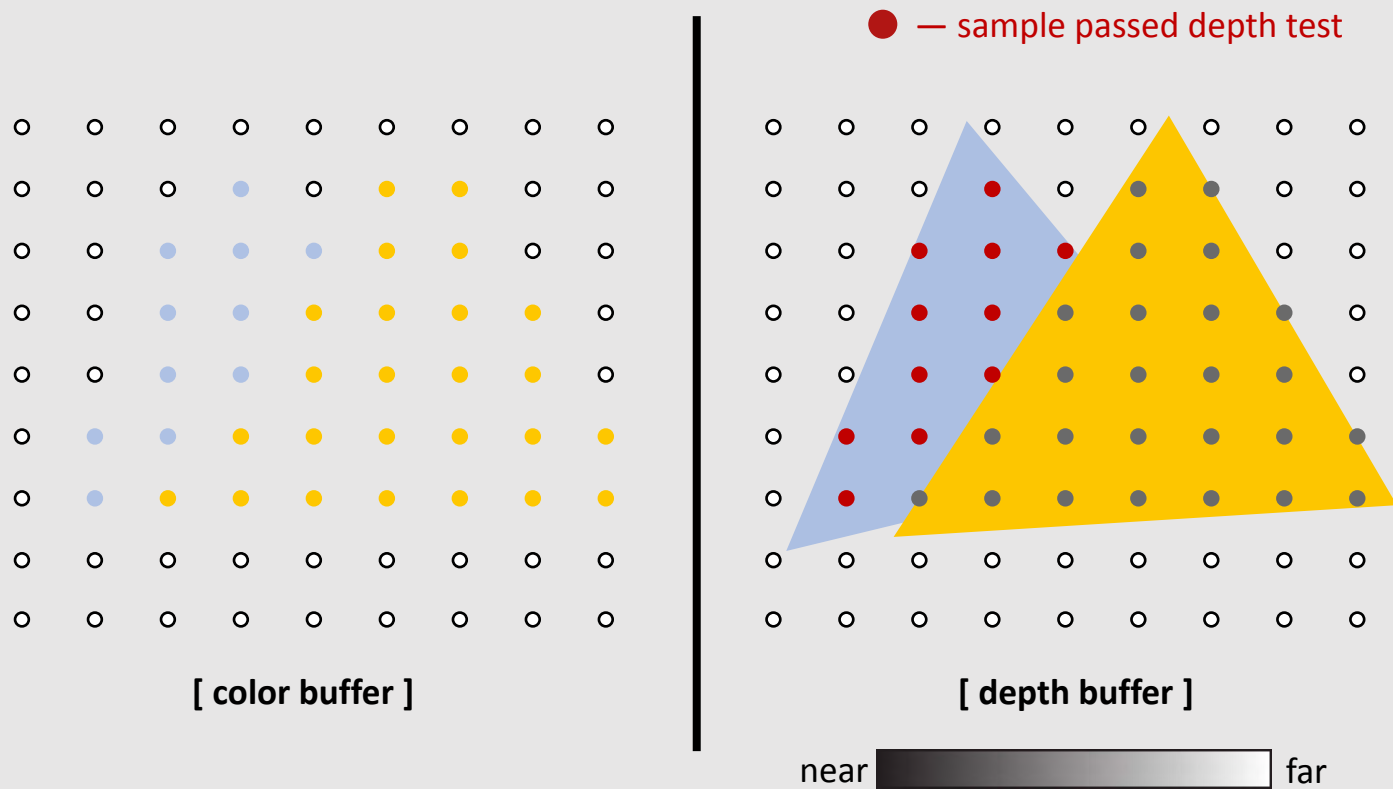
- Texture map is now a grid of downsampled textures
 - Known as a RipMap



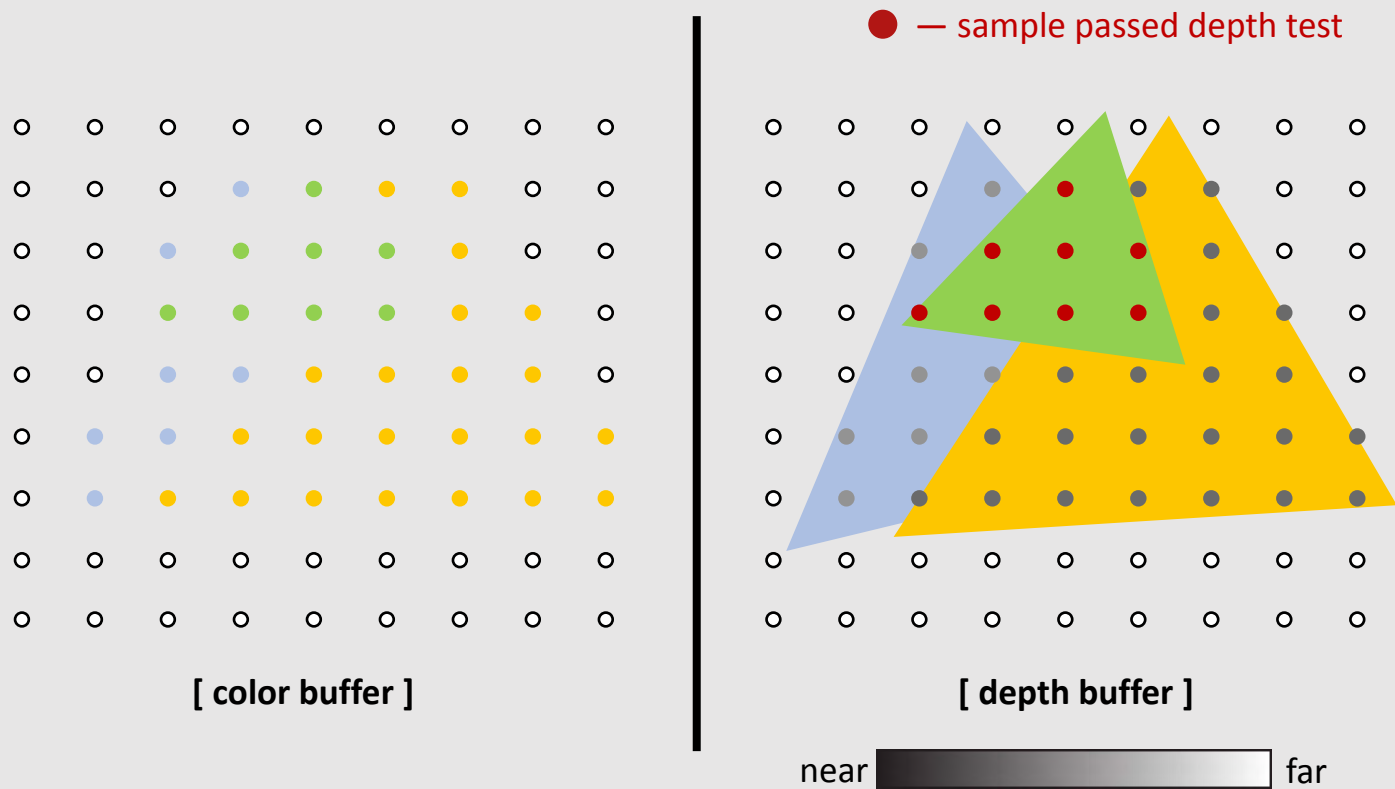
Depth Buffer (Z-buffer)



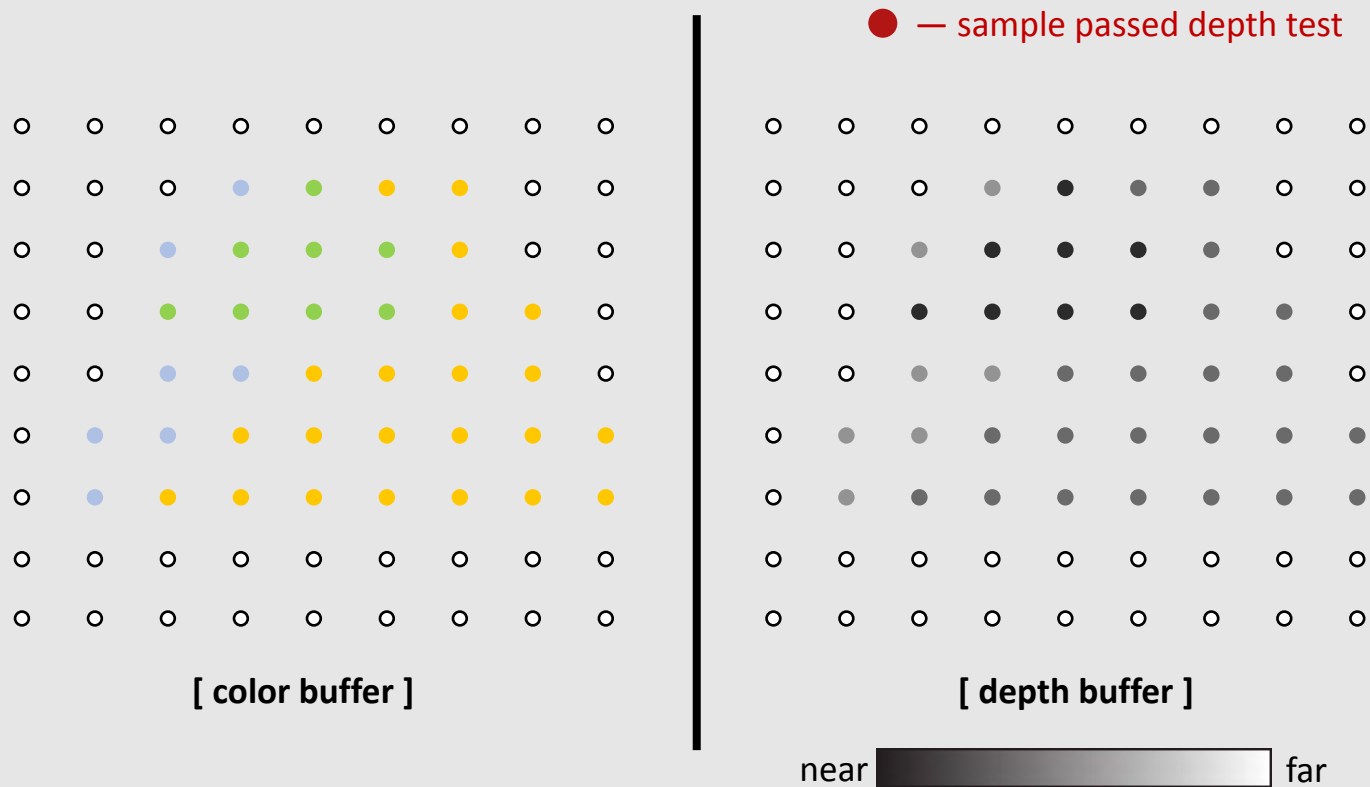
Depth Buffer (Z-buffer)



Depth Buffer (Z-buffer)



Depth Buffer (Z-buffer)



Alpha Values

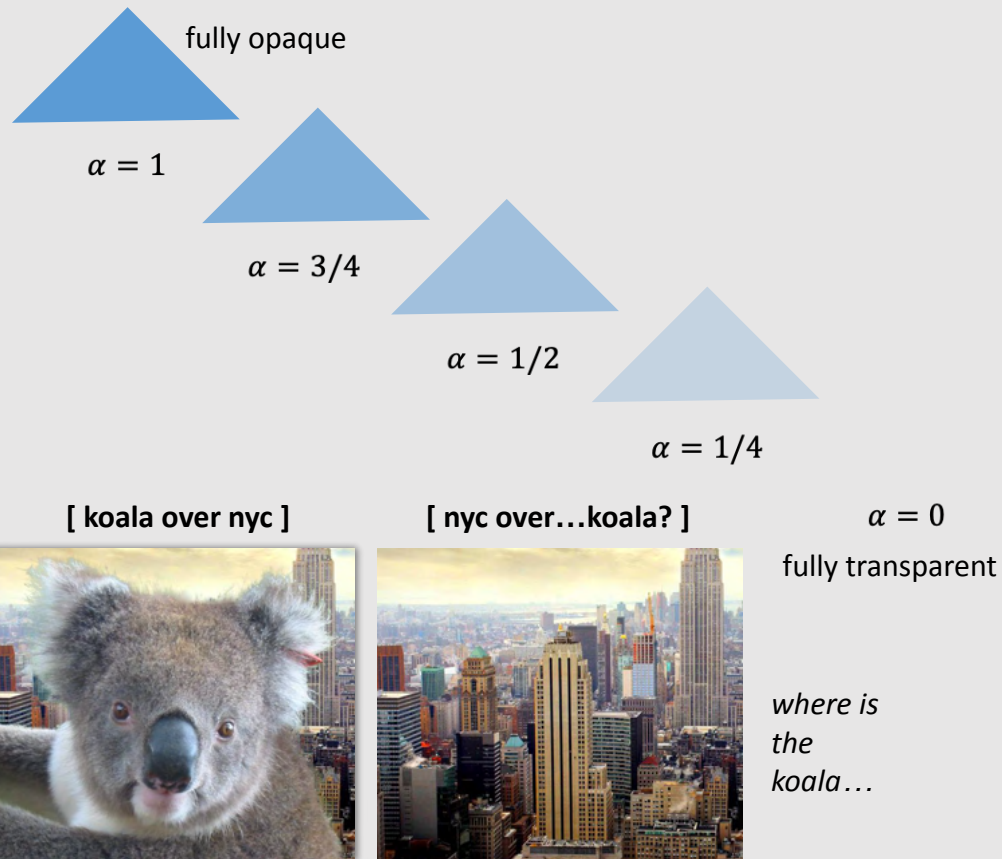
- Common image format: RGBA
 - Alpha channel specifies 'opacity'/transparency of object
 - Most common encoding is 8-bits per channel

- Compositing A over B $\neq$ B over A
 - Consider the extreme case of two opaque objects...

- Non-premultiplied alpha vs Premultiplied alpha

$$C = \alpha_B B + (1 - \alpha_B) \alpha_A A$$

$$C' = B' + (1 - \alpha_B) A'$$



Deferred Shading

- Popular algorithm for rendering in modern games
- **Idea:** restructure the rendering pipeline to perform shading **after** all occlusions have been resolved
- Not a new idea. Implemented in several classic graphics systems, but not directly supported in most high-end GPUs
 - But modern graphics pipeline provides mechanism to allow applications to implement deferred shading efficiently

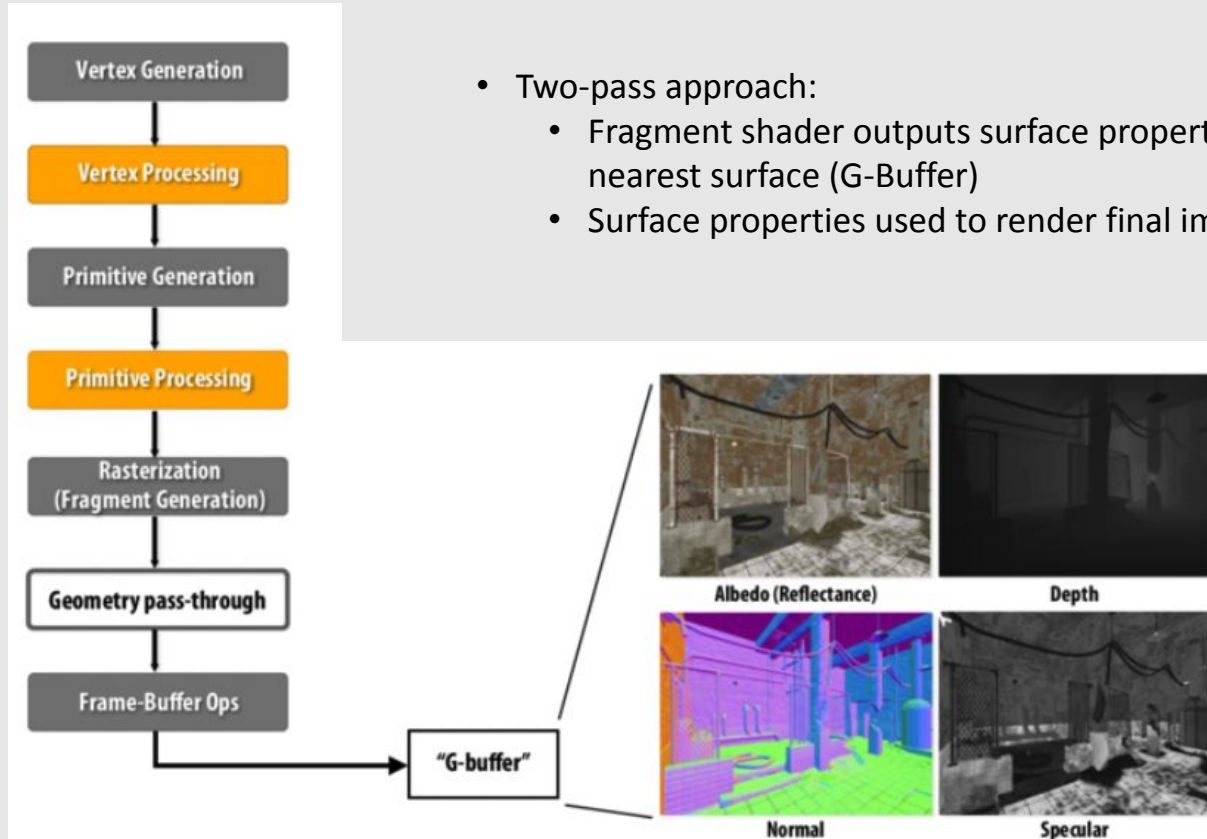


Assassin's Creed III (2012) Ubisoft



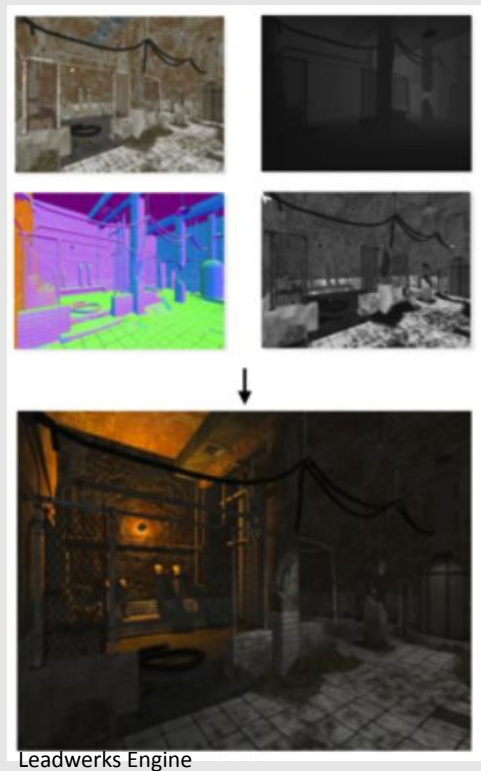
Shrek (2001) Digital Illusions Canada

Deferred Shading



Two-Pass Algorithm

- **Pass 1: Geometry Pass**
 - Render scene geometry using traditional pipeline
 - Write visible geometry information to G-Buffer
- **Pass 2: Shading Pass**
 - For each G-Buffer sample, compute shading
 - Read G-Buffer data for current sample
 - Accumulate contribution of all lights
 - Output final surface color for sample

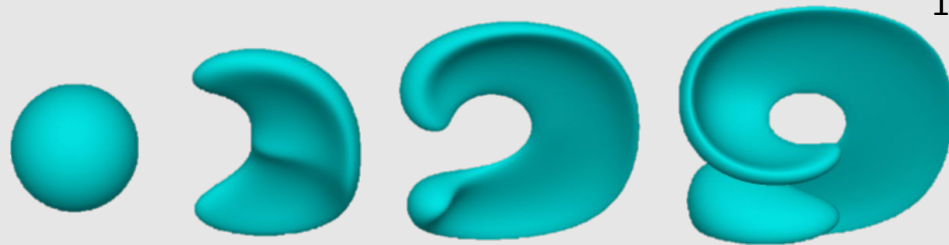
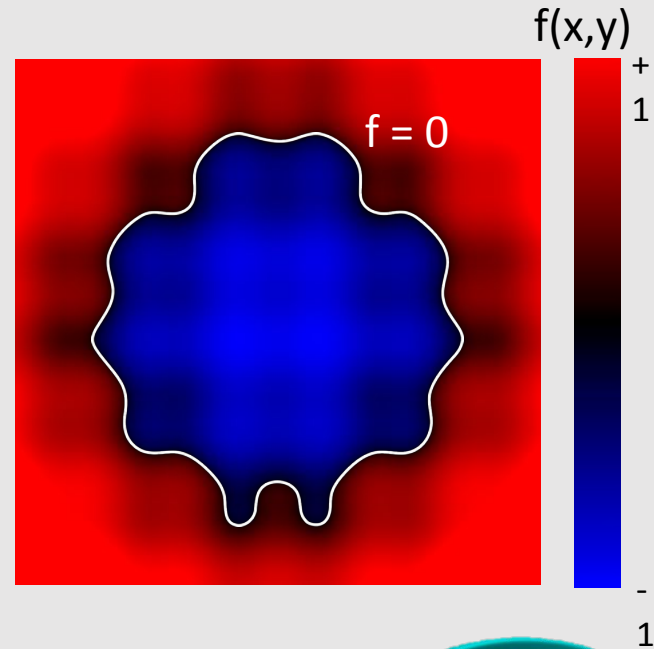


Leadwerks Engine

- Transformations Review
- Rasterization Review
- Geometry Review
- Spatial Data Structures Review

Implicit Geometry

- Points aren't known directly, but satisfy some relationship
 - Example: unit sphere is all points such that $x^2+y^2+z^2=1$
- More generally, in the form $f(x,y,z) = 0$
- Finding example points is **hard**
 - Requires solving equation
- Checking if points are inside/outside is **easy**
 - Just evaluate the function with a given point



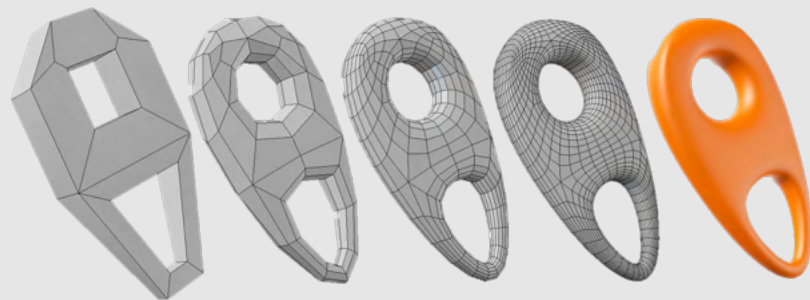
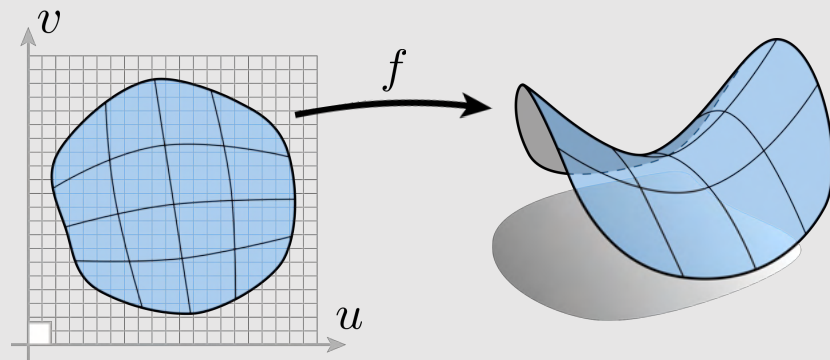
Explicit Geometry

- All points are given directly
 - The polygons we were given during rasterization is an example of explicit geometry

- More generally:

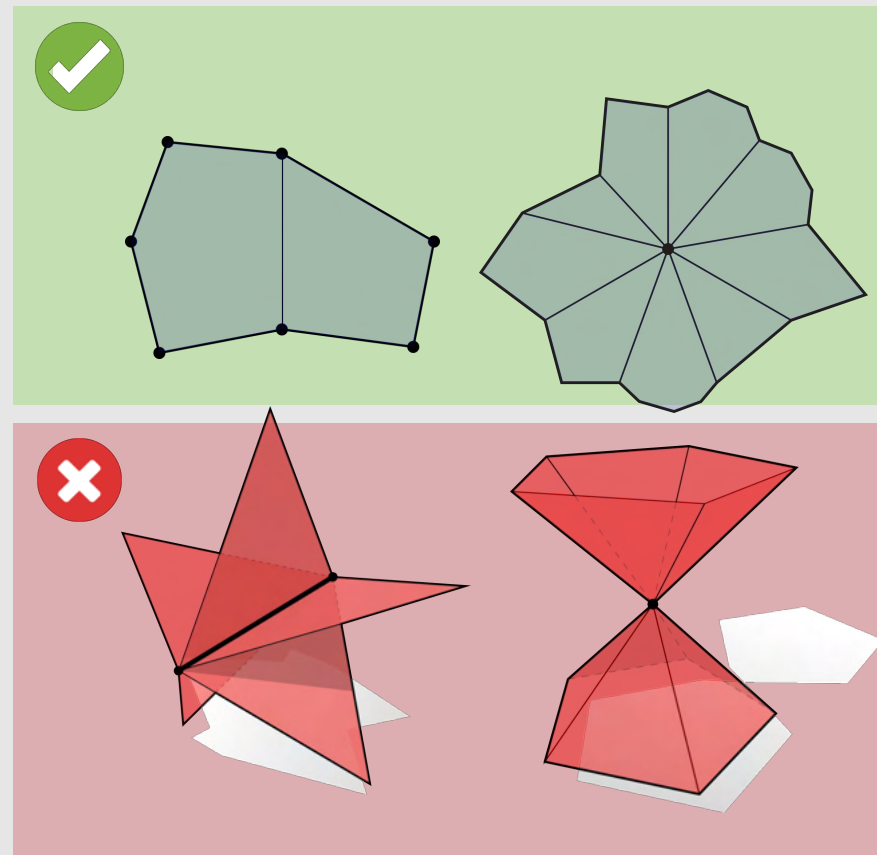
$$f : \mathbb{R}^2 \rightarrow \mathbb{R}^3; (u, v) \mapsto (x, y, z)$$

- Given any (u, v) , we can find a point on the surface
 - Can limit (u, v) to some range
 - Example: triangle with barycentric coordinates
-
- Finding example points is **easy**
 - We are given them for free
 - Checking if points are inside/outside is **hard**
 - We are given the output values and need to find input values that satisfy the geometry



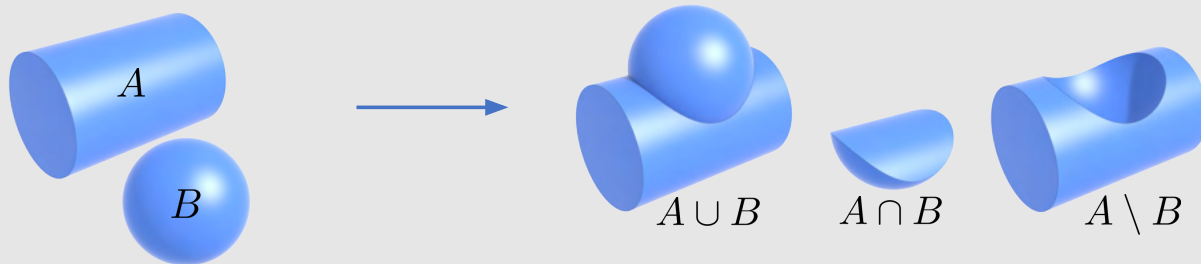
Manifold Properties

- For polygonal surfaces, we will check for “fins” and “fans”
- Every edge is contained in only two polygons (no “fins”)
 - The extra 3rd or 4th or 5th or so forth polygon is the fin of a fish
- The polygons containing each vertex make a single “fan”
 - We should be able to loop around the faces around a vertex in a clear way



Geometry Types

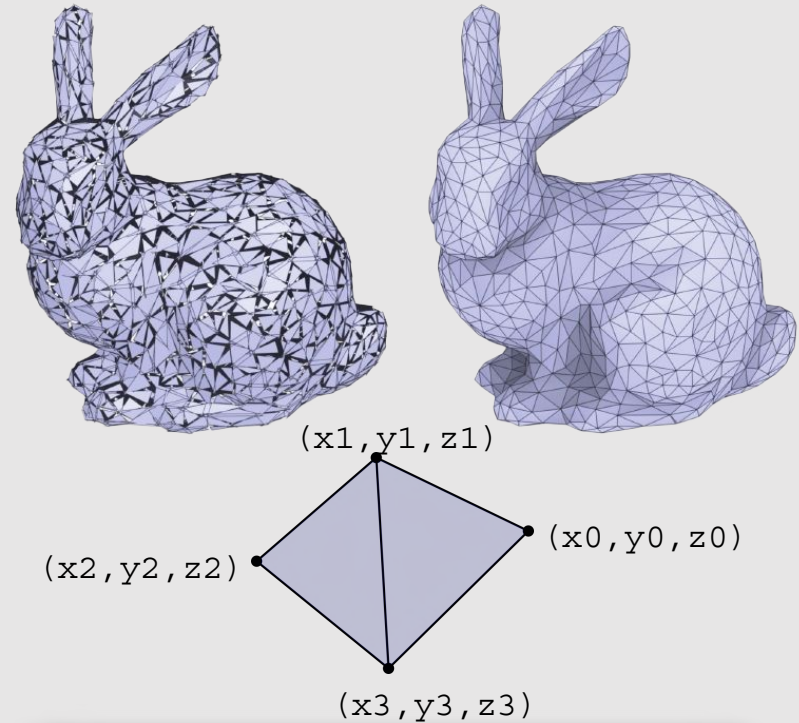
- What is implicit geometry
 - Algebraic surfaces
 - Constructive solid geometry
 - Signed distance fields
- What is explicit geometry
 - Point clouds
 - Triangle meshes
- Be able to compare the pros and cons of implicit and explicit geometry
- Manifold mesh requirements



What are some ways to describe the connectivity of geometry?

Polygon Soup

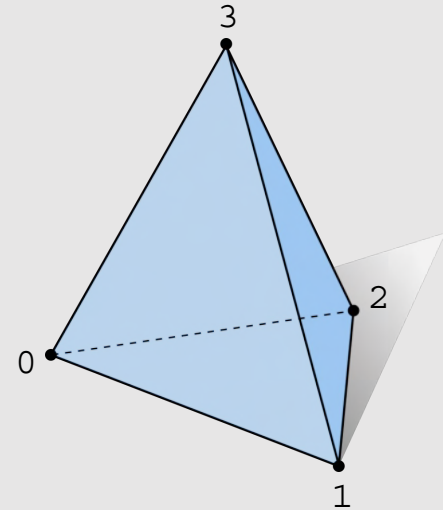
- Most basic idea imaginable:
 - For each triangle, just store three coordinates
 - No other information about connectivity
 - Not much different from point cloud
 - A “Triangle cloud”?
- **Pros:**
 - [+] Really stupid simple
- **Cons:**
 - [-] Really stupid
 - [-] Redundant storage of vertices
 - [-] Very difficult to find neighboring polygons



x_0, y_0, z_0	x_1, y_1, z_1	x_3, y_3, z_3
x_1, y_1, z_1	x_2, y_2, z_2	x_3, y_3, z_3

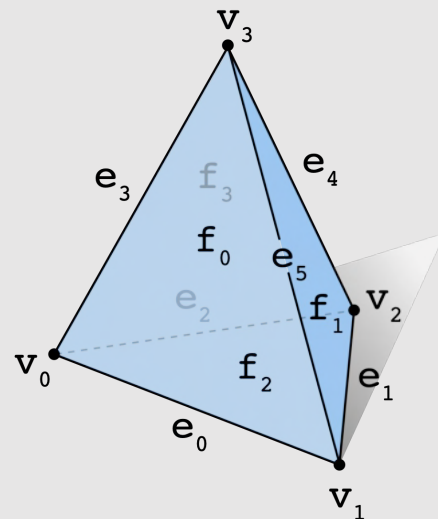
Adjacency List

- A little more complicated:
 - Store triples of coordinates (x,y,z)
 - Store tuples of indices referencing the coordinates needed to build each triangle
- **Pros:**
 - [+] No duplicate coordinates
 - [+] Lower memory footprint
 - [+] Easy to keep geometry manifold
 - [+] Supports nonmanifold geometry
 - [+] Easy to change connectivity of geometry
- **Cons:**
 - [-] Very difficult to find neighboring polygons
 - [-] Difficult to add/remove mesh elements



Incidence Matrices

- If we want to know our neighbors, let's store them:
 - Store triples of coordinates (x,y,z)
 - Store incidence matrix between vertices + edges, and edges + faces
 - 1 means touch, 0 means no touch
 - Store as sparse matrix
- **Pros:**
 - [+] No duplicate coordinates
 - [+] Finding neighbors is $O(1)$
 - [+] Easy to keep geometry manifold
 - [+] Supports nonmanifold geometry
- **Cons:**
 - [-] Larger memory footprint
 - [-] Hard to change connectivity with fixed indices
 - [-] Difficult to add/remove mesh elements



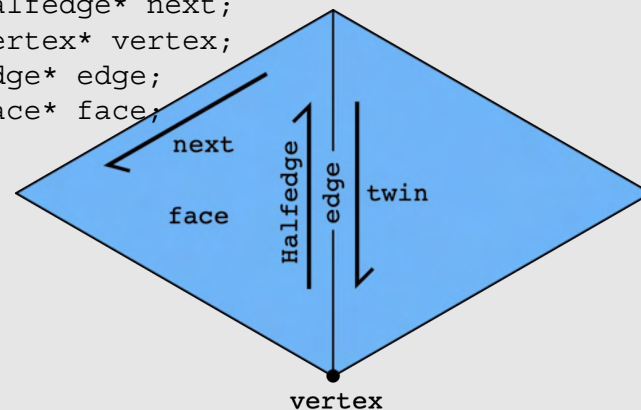
Halfedge Mesh

- What are the components of a halfedge mesh?
- How to traverse around a vertex? A face? An edge?
- Why can we not represent a non-manifold mesh using halfedge geometry?
- What makes a good mesh?

Halfedge Data Structure

- Let's store a little, but not a lot, about our neighbors:
 - Halfedge data structure added to our geometry
 - Each edge gets 2 halfedges
 - Each halfedge "glues" an edge to a face
- **Pros:**
 - [+] No duplicate coordinates
 - [+] Finding neighbors is $O(1)$
 - [+] Easy to traverse geometry
 - [+] Easy to change mesh connectivity
 - [+] Easy to add/remove mesh elements
 - [+] Easy to keep geometry manifold
- **Cons:**
 - [-] Does not support nonmanifold geometry

```
struct Halfedge
{
    Halfedge* twin;
    Halfedge* next;
    Vertex* vertex;
    Edge* edge;
    Face* face;
};
```



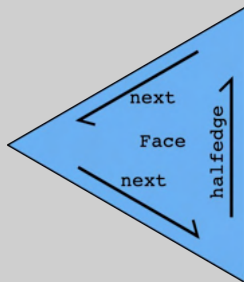
Halfedge Data Structure

- Makes mesh traversal easy
 - Use “twin” and “next” pointers to move around the mesh
 - Use “vertex”, “edge”, and “face” pointers to grab element

```
struct Halfedge
{
    Halfedge* twin;
    Halfedge* next;
    Vertex* vertex;
    Edge* edge;
    Face* face;
};
```

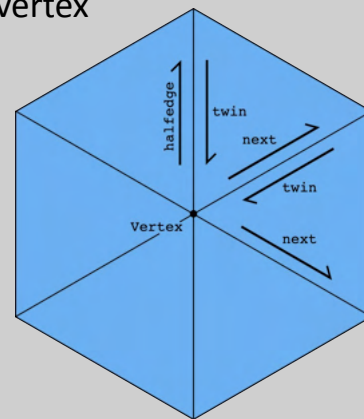
Example: visit all vertices in a face

```
Halfedge* h = f->halfedge;
do {
    h = h->next;
    // do something w/ h->vertex
}
while( h != f->halfedge );
```



Example: visit all neighbors of a vertex

```
Halfedge* h = v->halfedge;
do {
    h = h->twin->next;
}
while( h != v->halfedge );
```



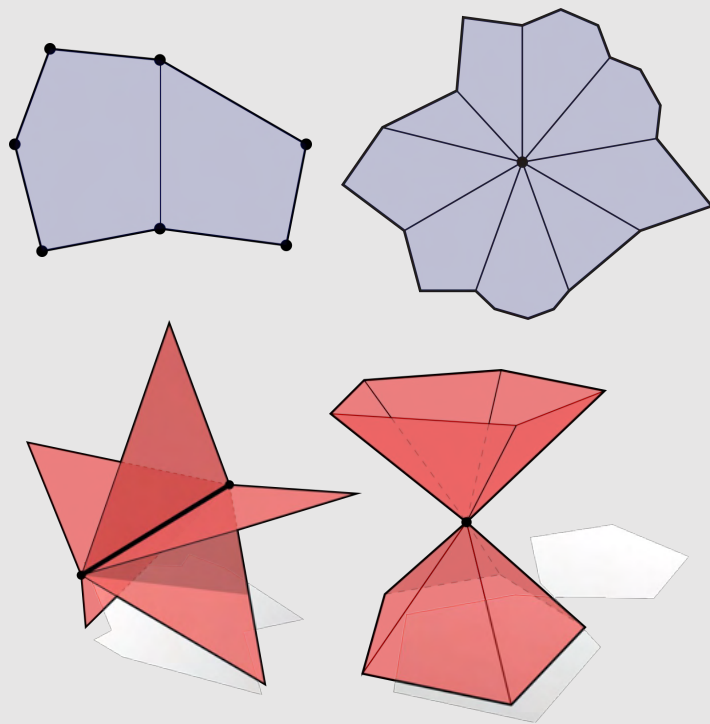
Note: only makes sense if mesh is manifold!

Halfedge Data Structure

- Halfedge meshes are always manifold!
- Halfedge data structures have the following constraints:

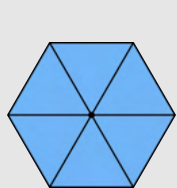
```
h->twin->twin == h // my twin's twin is me  
h->twin != h // I am not my own twin  
h2->next = h // every h's is someone's "next"
```

- Keep following **next** and you'll traverse a face
- Keep following **twin** and you'll traverse an edge
- Keep following **next->twin** and you'll traverse a vertex
- **Q: Why, therefore, is it impossible to encode the red figures?**
 - First shape violates first 2 conditions
 - Second shape violates 3rd condition

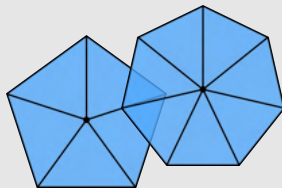


A Good Mesh Has...

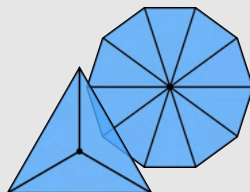
- Good approximation of original shape
 - Keep only elements that contribute information about shape
 - More elements where curvature is high
- Regular vertex degree
 - Degree 6 for triangle mesh, 4 for quad mesh
 - Better polygon shape
 - More regular computation
 - Smoother subdivision



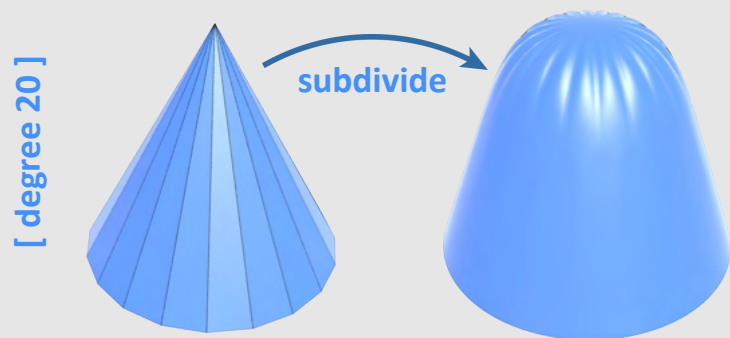
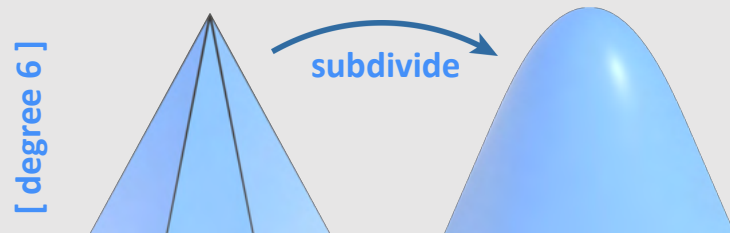
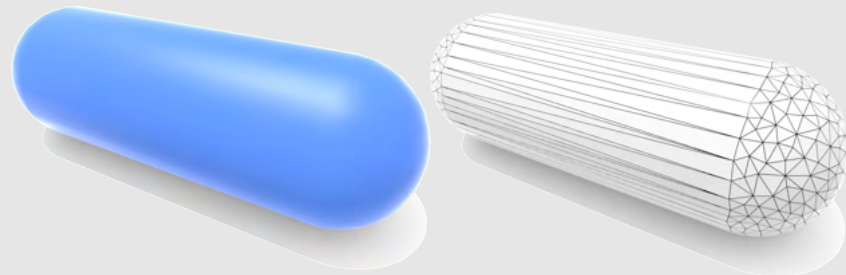
[good]



[okay]

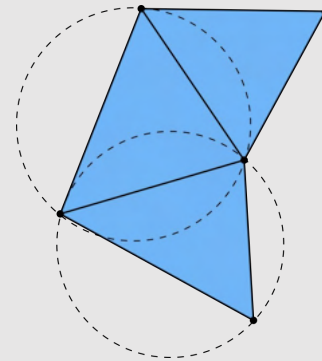
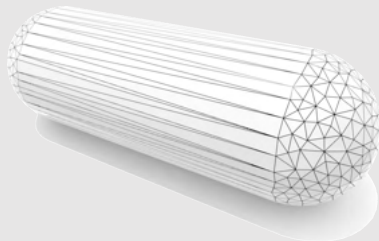


[bad]

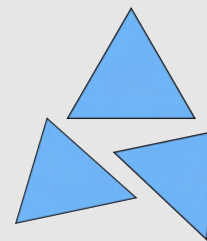


A Good Mesh Has...

- Good triangle shape
 - All angles close to 60 degrees
- More sophisticated condition: **Delaunay**
 - For every triangle, the unique circumcircle (circle passing through all vertices of the triangle) does not encase any other vertices
 - Many nice properties:
 - Maximizes minimum angle
 - Smoothest interpolation
- **Tradeoff:** sometimes a mesh can be approximated best with long & skinny triangles
 - Doesn't make the mesh Delaunay anymore
 - Example: cylinder



[delaunay]



[good]

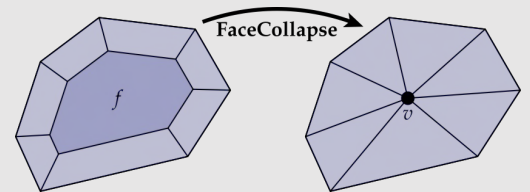
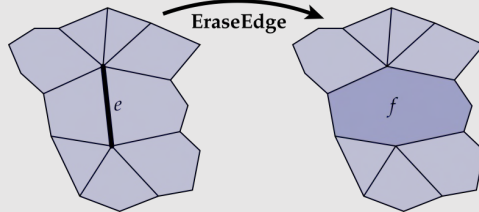
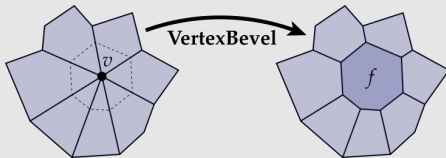
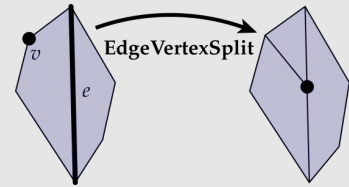
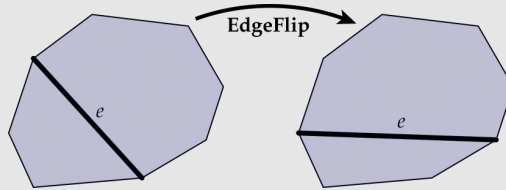
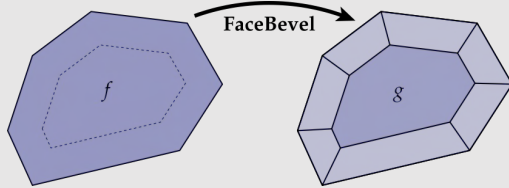
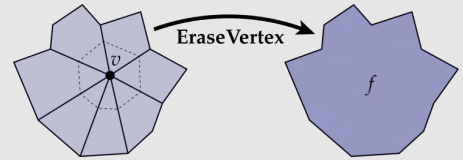
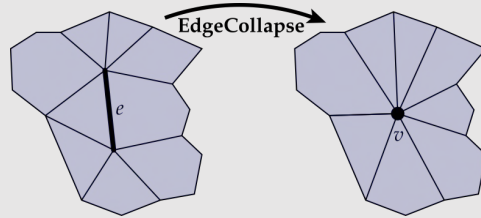
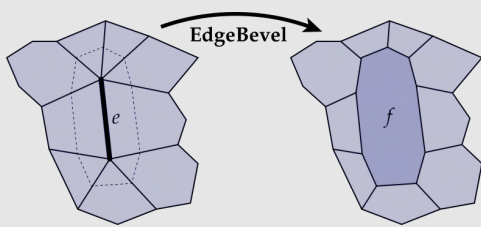


[bad]

Halfedge Mesh Operations

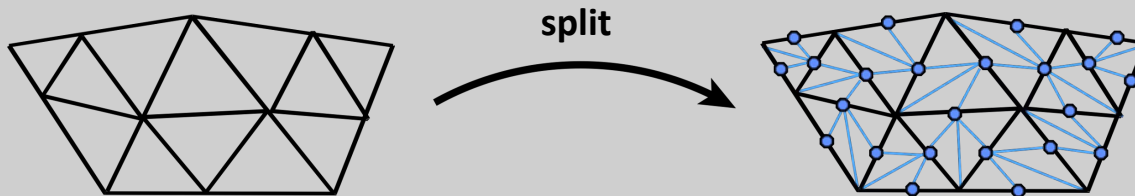
- Local Operations
 - EdgeBevel
 - EdgeCollapse
 - EraseVertex
 - FaceBevel
 - EdgeFlit
 - EdgeVertexSplit
 - VertexBevel
 - EraseEdge
 - FaceCollapse
 - ...
- Global Operations
 - Loop Subdivision
 - Isotropic Remeshing
 - Simplification
 - ...

Local Operations

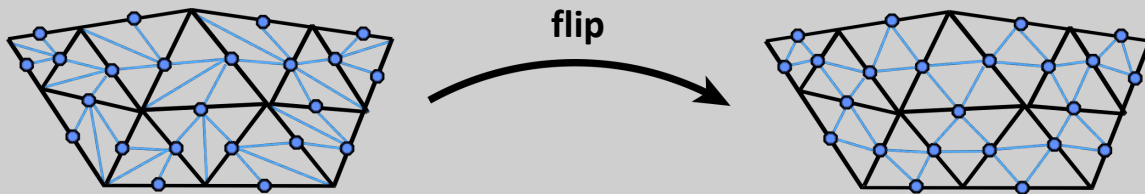


Loop Subdivision Using Local Ops

Step 1:
Split all edges in any order



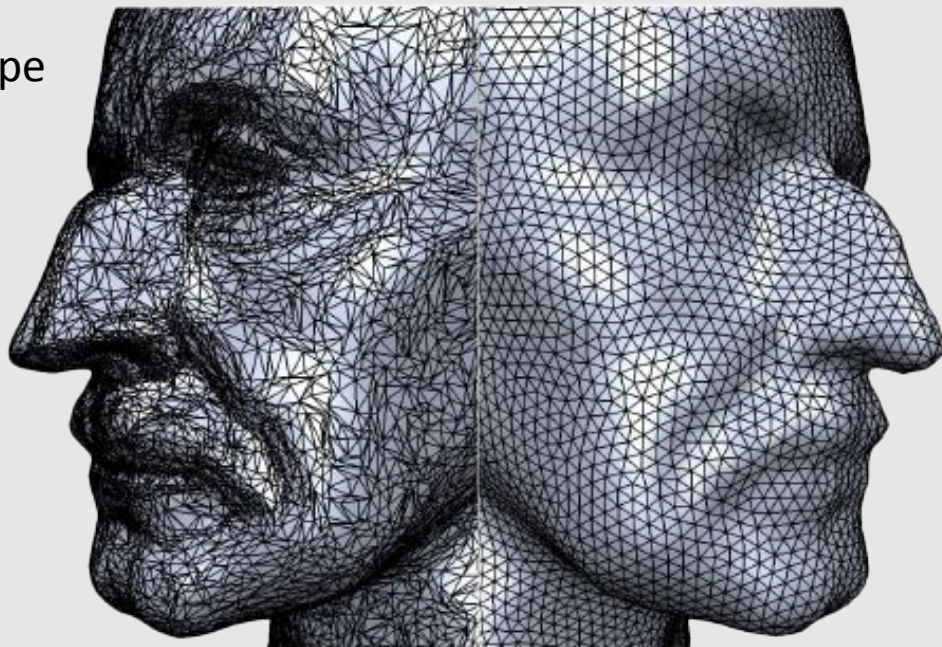
Step 2:
Flip new edges until they touch two new vertices



Isotropic Remeshing

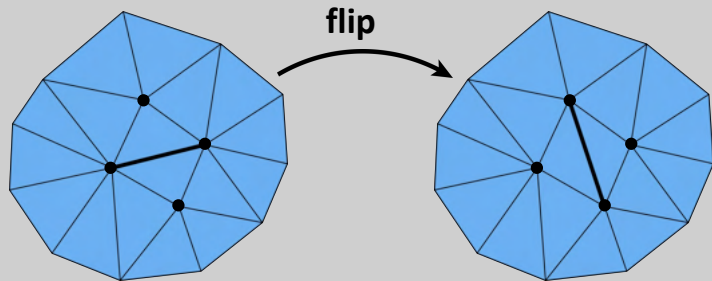
Goal: change the mesh to make triangles more uniform shape and size

- Helps achieve good mesh properties:
 - Good approximation of original shape
 - Vertex degrees close to 6
 - Angles close to 60deg
 - Delaunay triangles

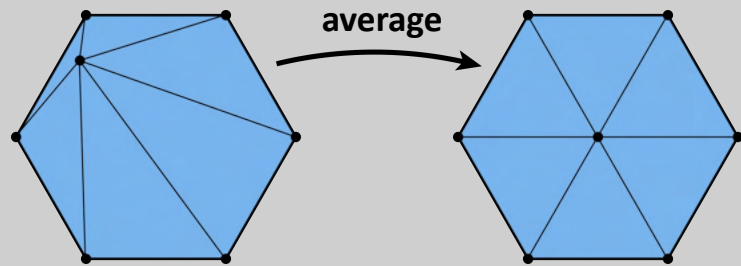


Isotropic Remeshing

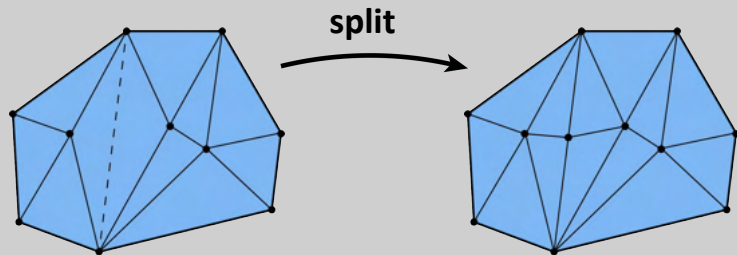
Step 1:



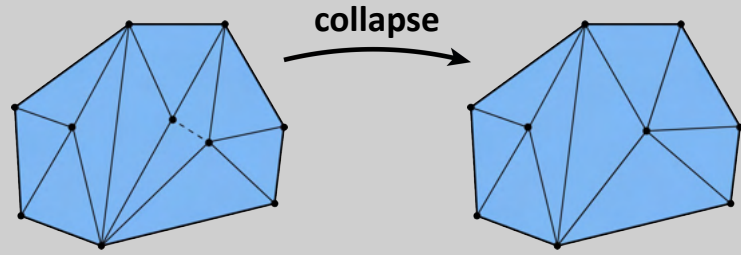
Step 2:



Step 3:

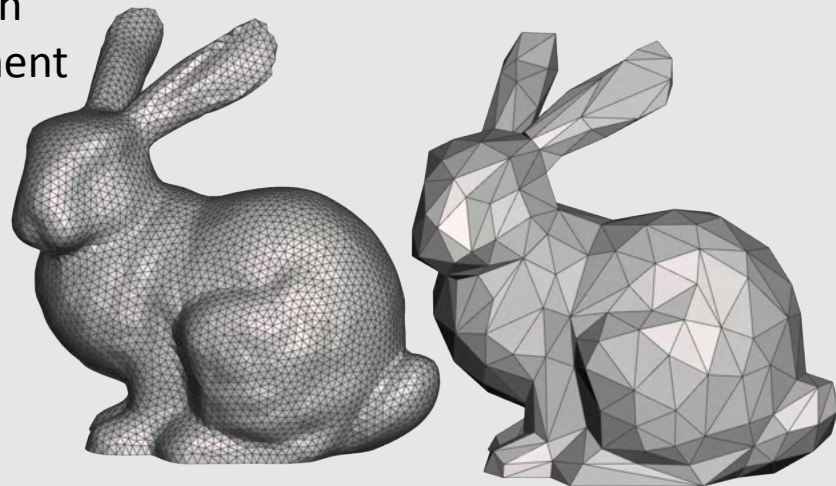


Step 4:



Simplification Algorithm Basics

- Simplification is the process of **downsampling** a mesh
 - Less Storage overhead
 - Smaller file sizes
 - Less Processing overhead
 - Less elements to iterate over
 - Larger mesh modifications
 - Instead of moving tens of smaller mesh elements, move one larger mesh element



**invented at CMU (Garland & Heckbert 1997)

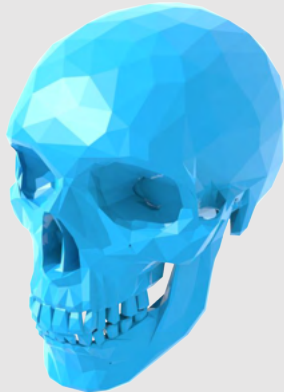
Simplification Algorithm Basics

- Greedy Algorithm:
 - Assign each edge a cost
 - Collapse edge with least cost
 - Repeat until target number of elements is reached
- Particularly effective cost function: **quadric error metric****

*No need to
remember the
specific metric!*



[30,000 triangles]



[3,000 triangles]



[300 triangles]



[30 triangles]

**invented at CMU (Garland & Heckbert 1997)

- Transformations Review
- Rasterization Review
- Geometry Review
- Spatial Data Structures Review

Spatial Data Structures

- **Primitive-partitioning** acceleration structure:
 - Partitions node's primitives into disjoint sets (but sets may overlap in space)
 - Bounding Volume Hierarchy
 - How to construct a BVH
 - How to traverse a BVH
 - Axis-aligned vs non-axis aligned BVHs
- **Space-partitioning** acceleration structures:
 - Partitions space into disjoint regions (but primitives may be contained in multiple regions)
 - K-D Trees
 - Uniform Grids
 - Quad/OctTrees

BVH Construction

For axis x, y, z :

Initialize buckets

For each primitive p in node:

$B = \text{compute_bucket}(p.\text{centroid})$

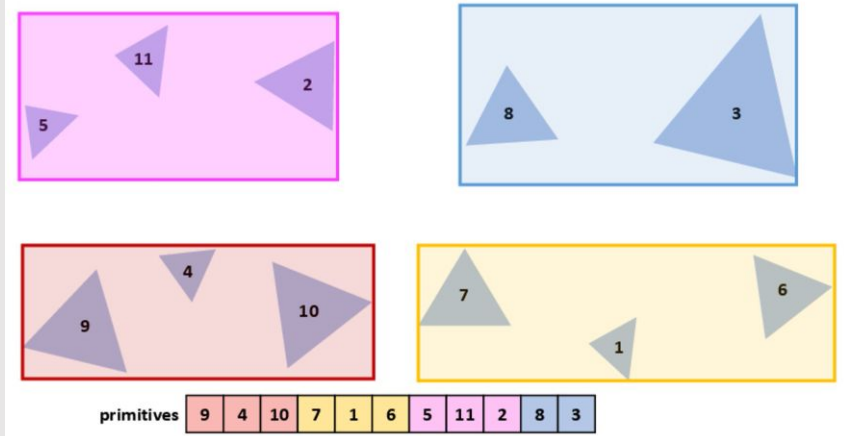
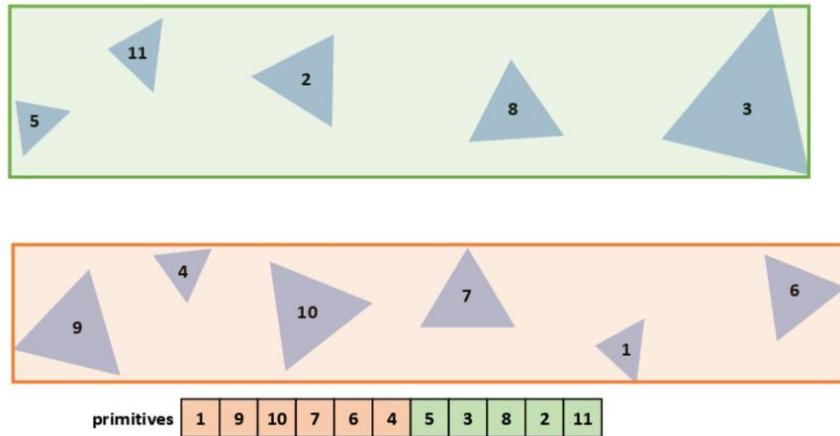
$B.\text{bbox}.\text{enclose}(p.\text{bbox})$

$B.\text{prim_count}++$

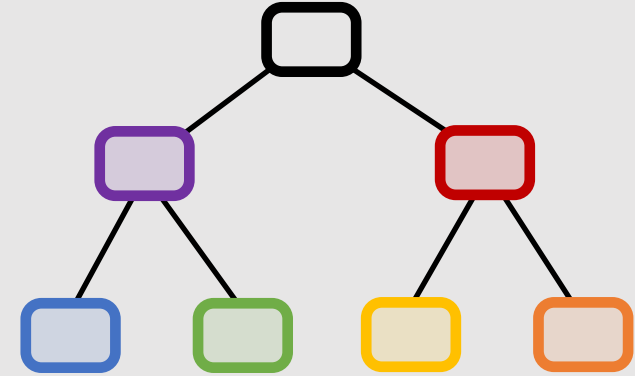
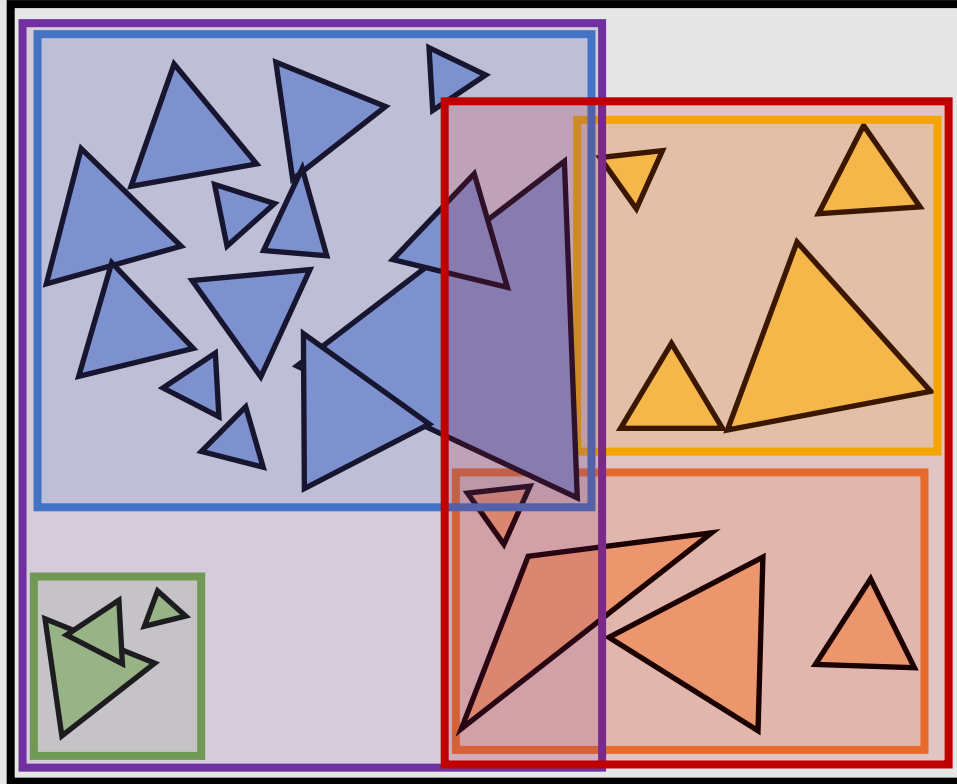
For each of $|B| - 1$ possible partitions

Evaluate cost (SAH), keep track of lowest cost partition

Recurse on lowest cost partition found (or make node leaf)



BVH Example



Bounding boxes will sometimes intersect!

Axis-Aligned BVH

- **Are non-axis-aligned BVHs actually faster?**

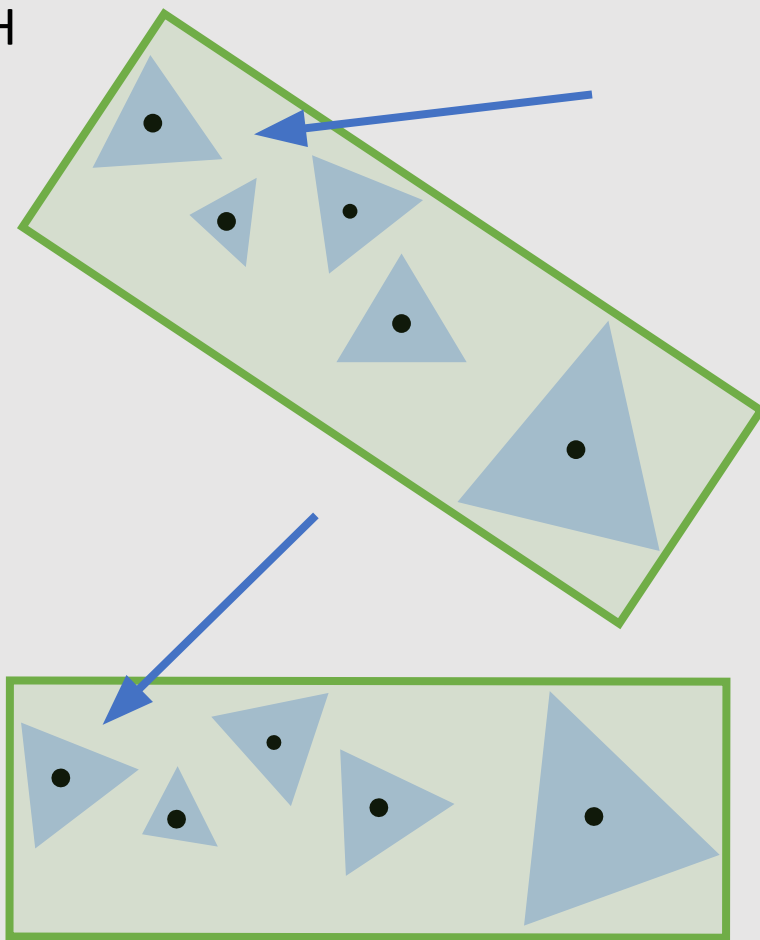
- Yes, and no.

$$C = C_{trav} + \frac{S_A}{S_C} N_A C_{tri} + \frac{S_B}{S_C} N_B C_{tri}$$

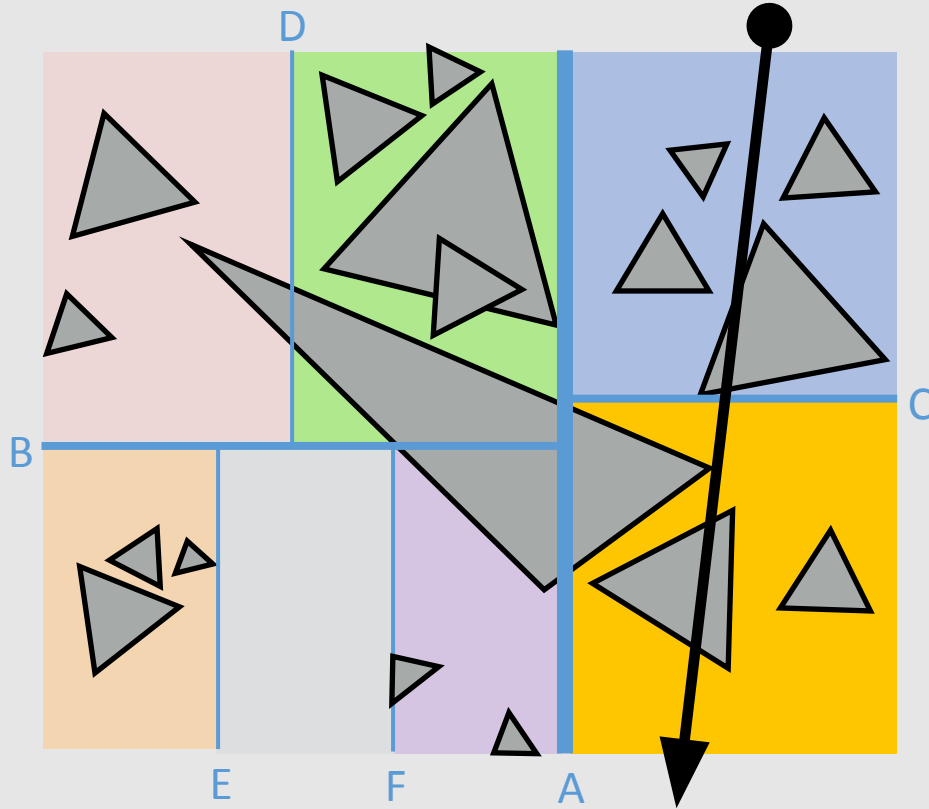
- Surface area ratio $\frac{S_A}{S_C}$ decreases with better-fitting bboxes
- Bounding box intersection cost C_{trav} increases with more compute required to check unaligned bbox

- **How to check for intersection with non-axis-aligned bbox?**

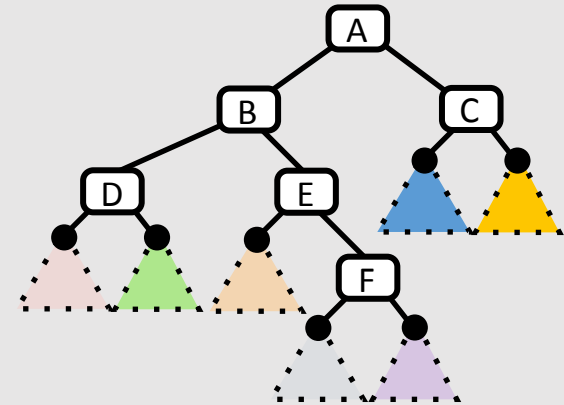
- Bbox now has an extra transform matrix T taking it from its local space to its parent space
 - Apply the **inverse** transform to the **ray** and compute axis-aligned intersections
- Larger memory overhead, now need to store the transform with each node



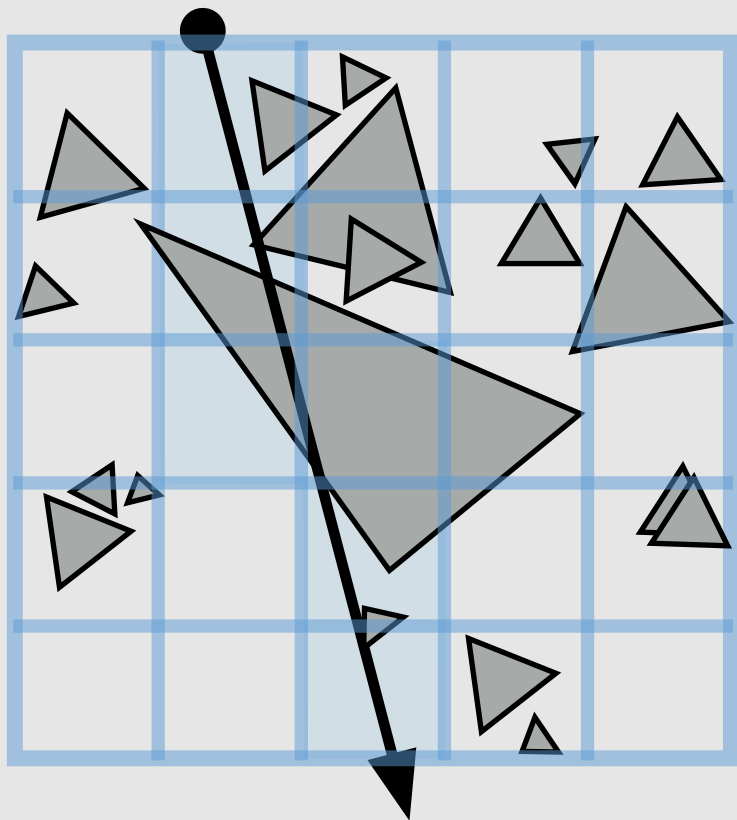
K-D Trees



- Recursively partition space via axis-aligned partitioning planes
 - Interior nodes correspond to spatial splits
 - Node traversal proceeds in front-to-back order
 - Unlike BVH, can terminate search after first hit is found
 - Still $O(\log(N))$ performance

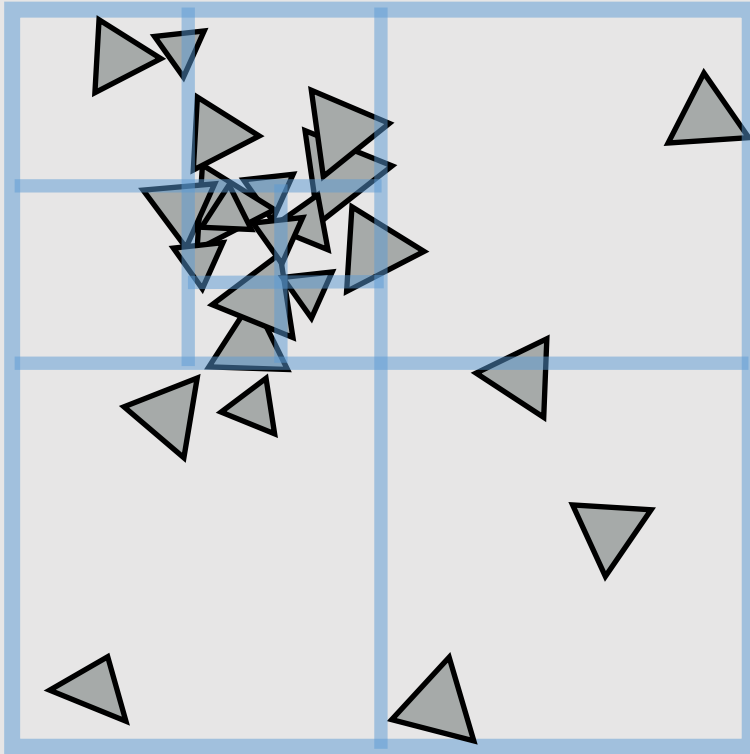


Uniform Grid



- Partition space into equal sized volumes (volume-elements or “voxels”)
- Each grid cell contains primitives that overlap voxel. (very cheap to construct acceleration structure)
- Walk ray through volume in order
 - Very efficient implementation possible (think: 3D line rasterization)
 - Only consider intersection with primitives in voxels the ray intersects
- What is a good number of voxels?
 - Should be proportional to total number of primitives N
 - Number of cells traversed is proportional to $O(\sqrt[3]{N})$
 - A line going through a cube is a cubed root
 - Still not as good as $O(\log(N))$

Quad-Tree/Octree



- Like uniform grid, easy to build
- Has greater ability to adapt to location of scene geometry than uniform grid
 - Still not as good adaptability as K-D tree
- **Quad-tree:** nodes have 4 children
 - Partitions 2D space
- **Octree:** nodes have 8 children
 - Partitions 3D space

