Announcments

- Sign up for crits!

Reading for Next Week

- FvD 16.1-16.3 – local lighting models
- GL 5 – lighting
- GL 9 (skim) – texture mapping

Modern Game Techniques

CS248 Lecture Nov 13

Andrew Adams

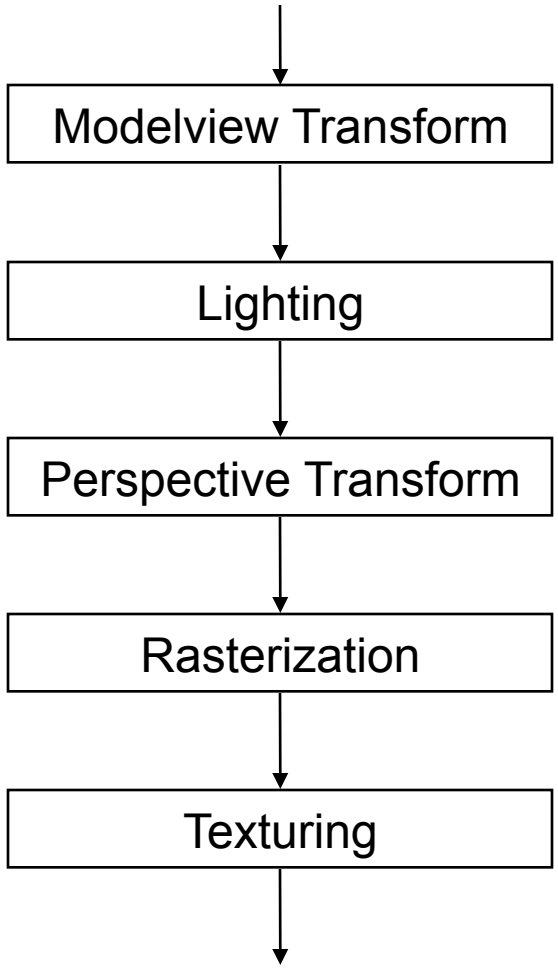
Overview

- The OpenGL Pipeline
- Vertex and Fragment Shaders
- Multipass Techniques
- Culling and Collision Detection

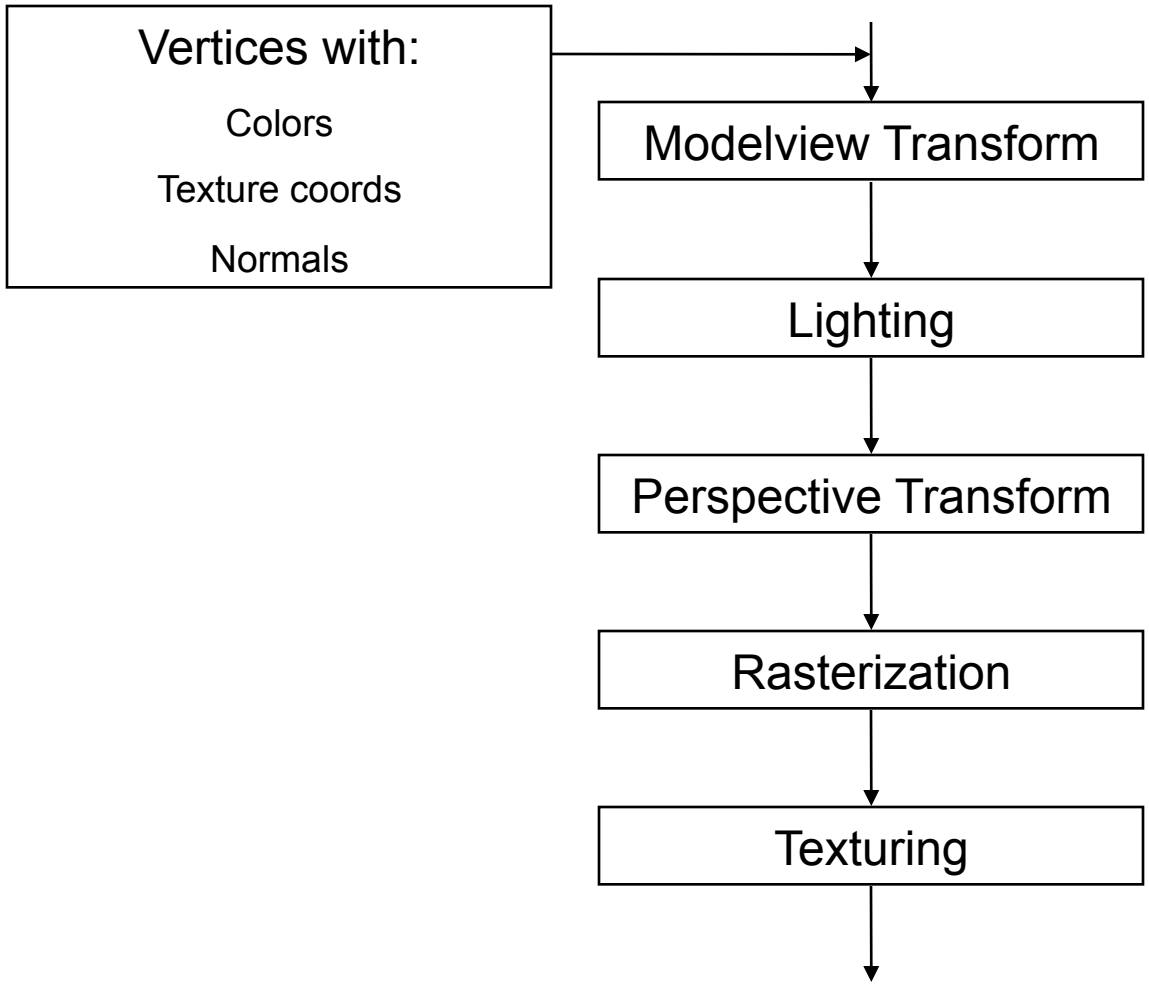
The OpenGL Pipeline

```
glBegin(GL_TRIANGLES);
glTexCoord2i(0, 0);
glNormal3f(0, 0, 1);
glColor3f(1, 0, 0);
glVertex3f(0, 0, 0);

glTexCoord2i(0, 1);
glColor3f(0, 1, 0);
glVertex3f(0, 1, 0);

glTexCoord2i(1, 0);
glColor3f(0, 0, 1);
glVertex3f(1, 0, 0);
glEnd();
```

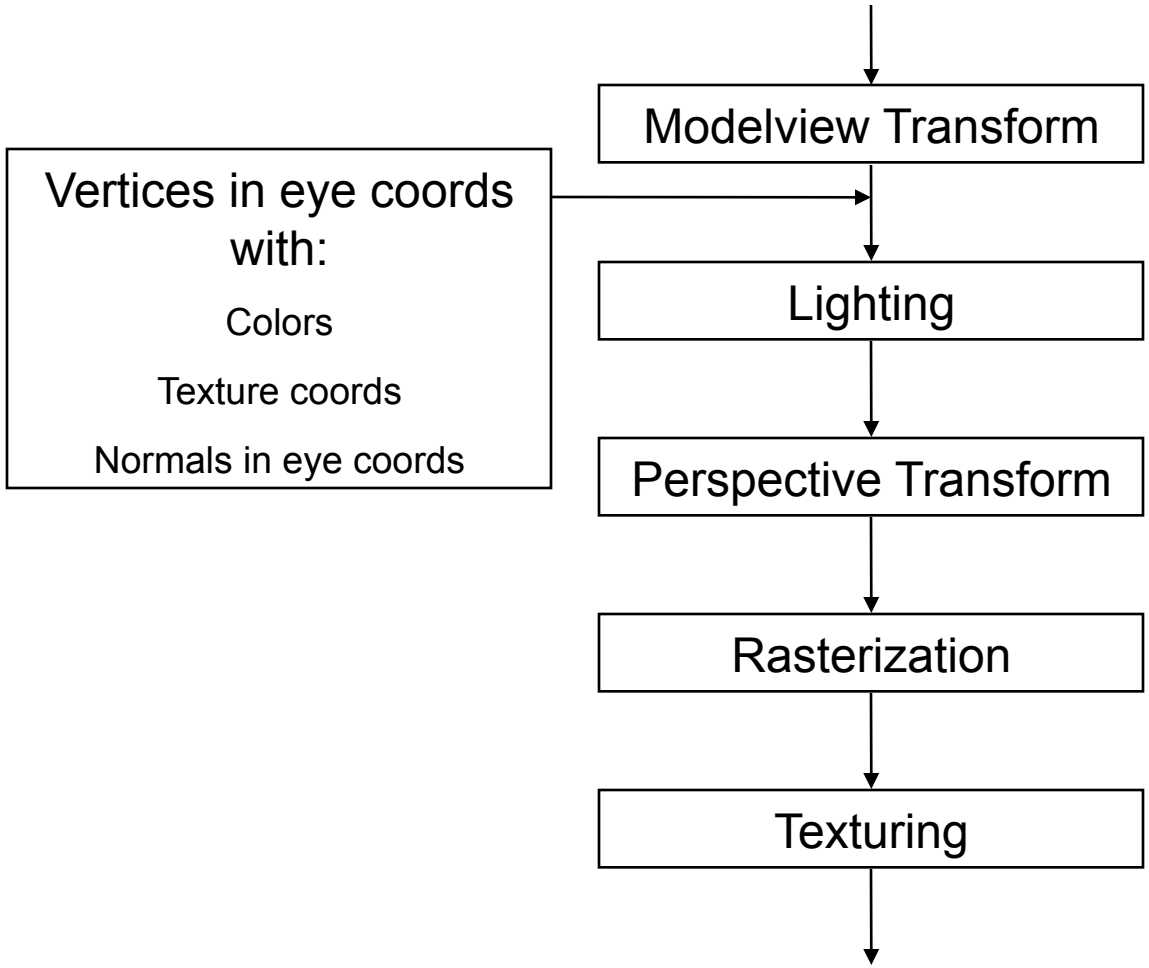
The OpenGL Pipeline



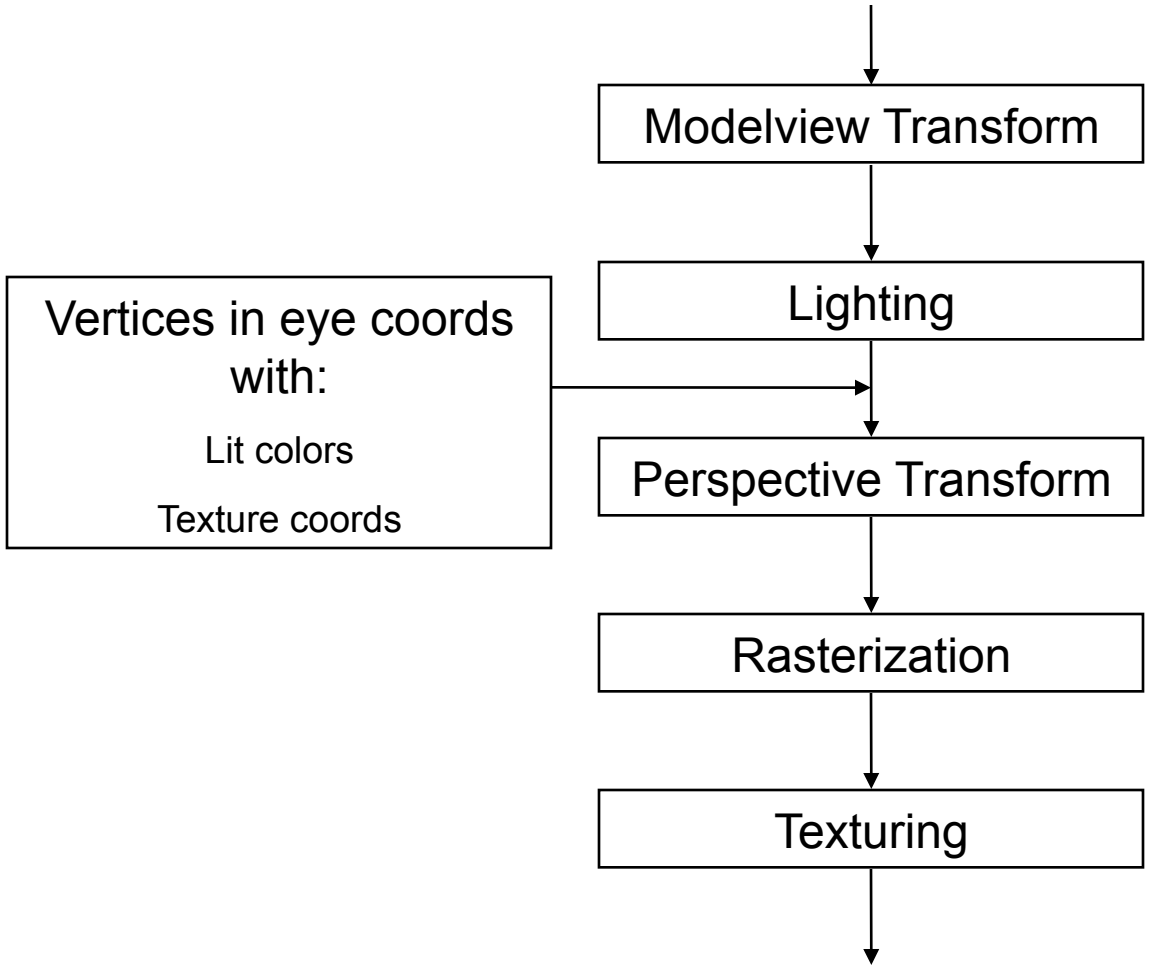
The OpenGL Pipeline



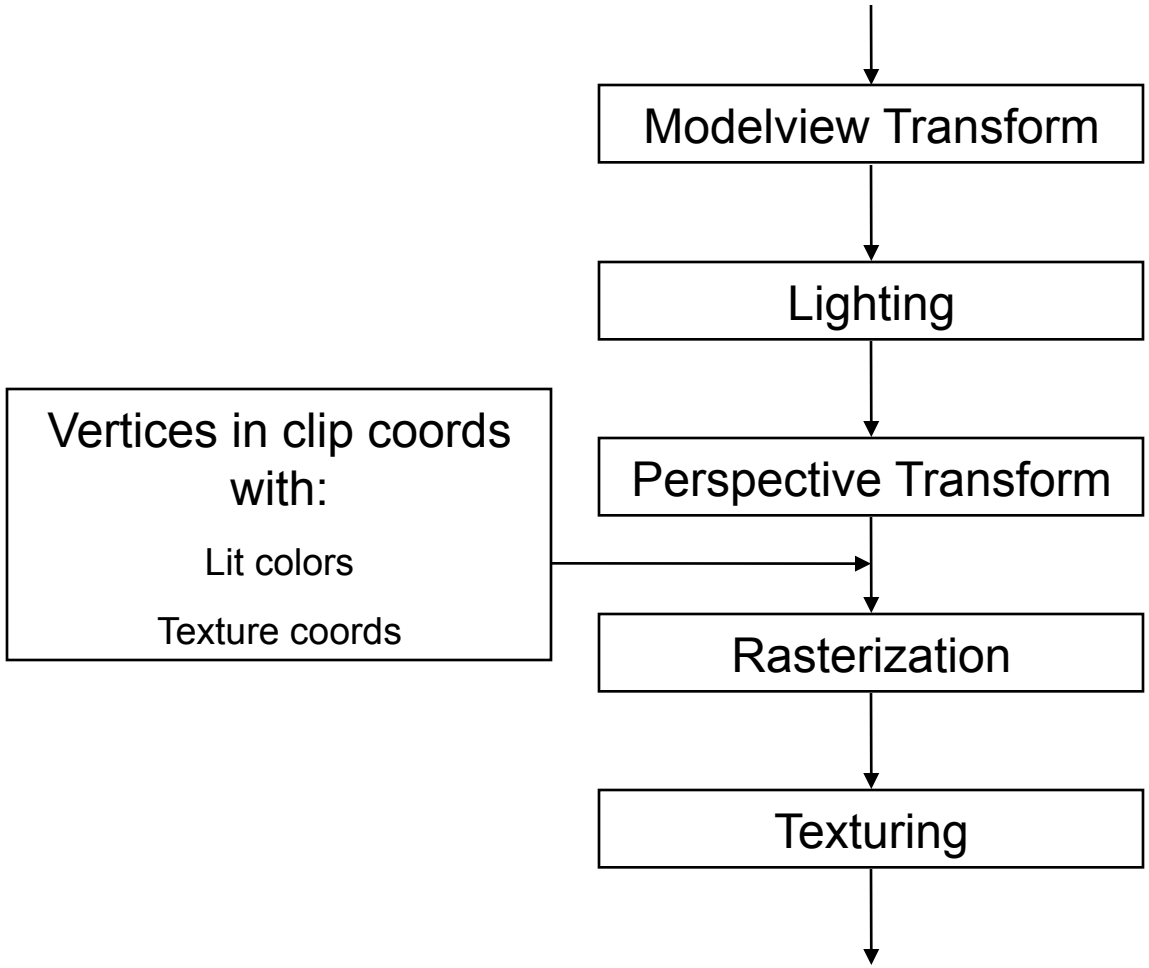
The OpenGL Pipeline



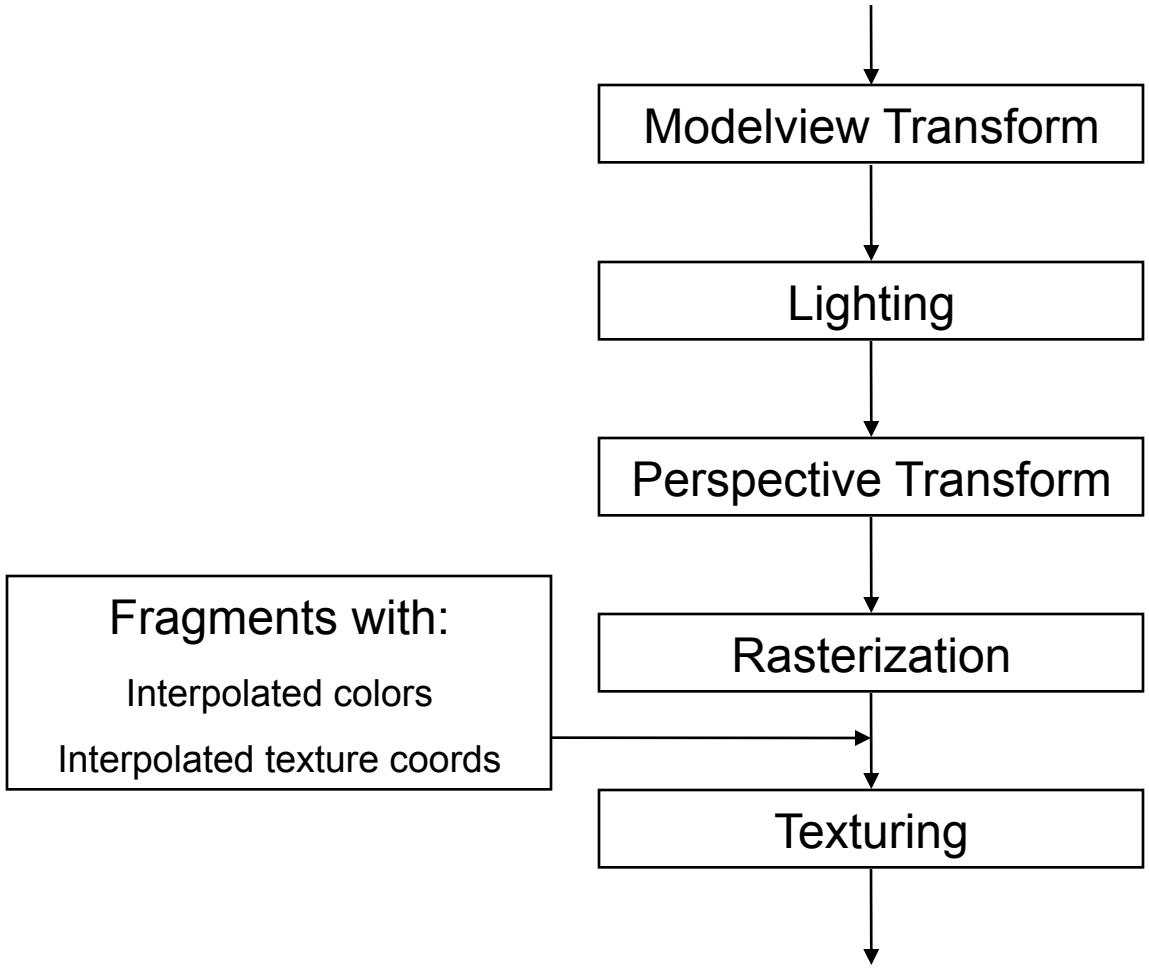
The OpenGL Pipeline



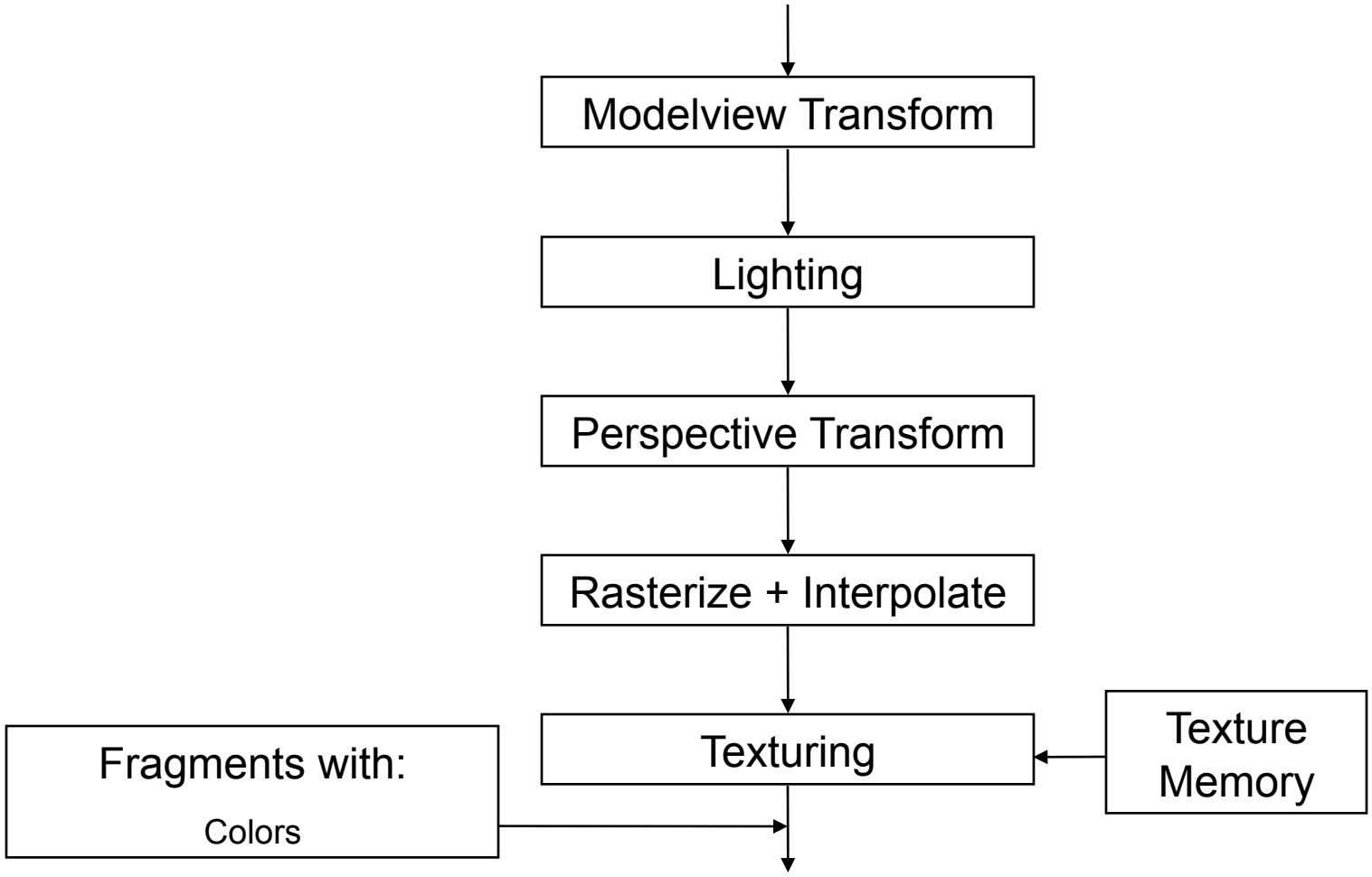
The OpenGL Pipeline



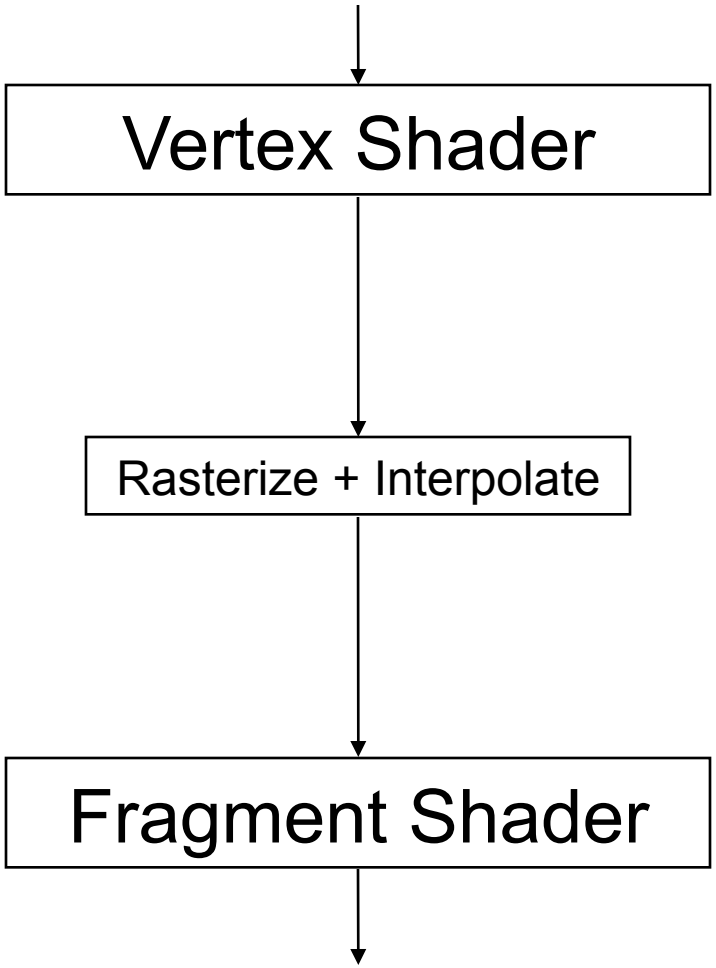
The OpenGL Pipeline



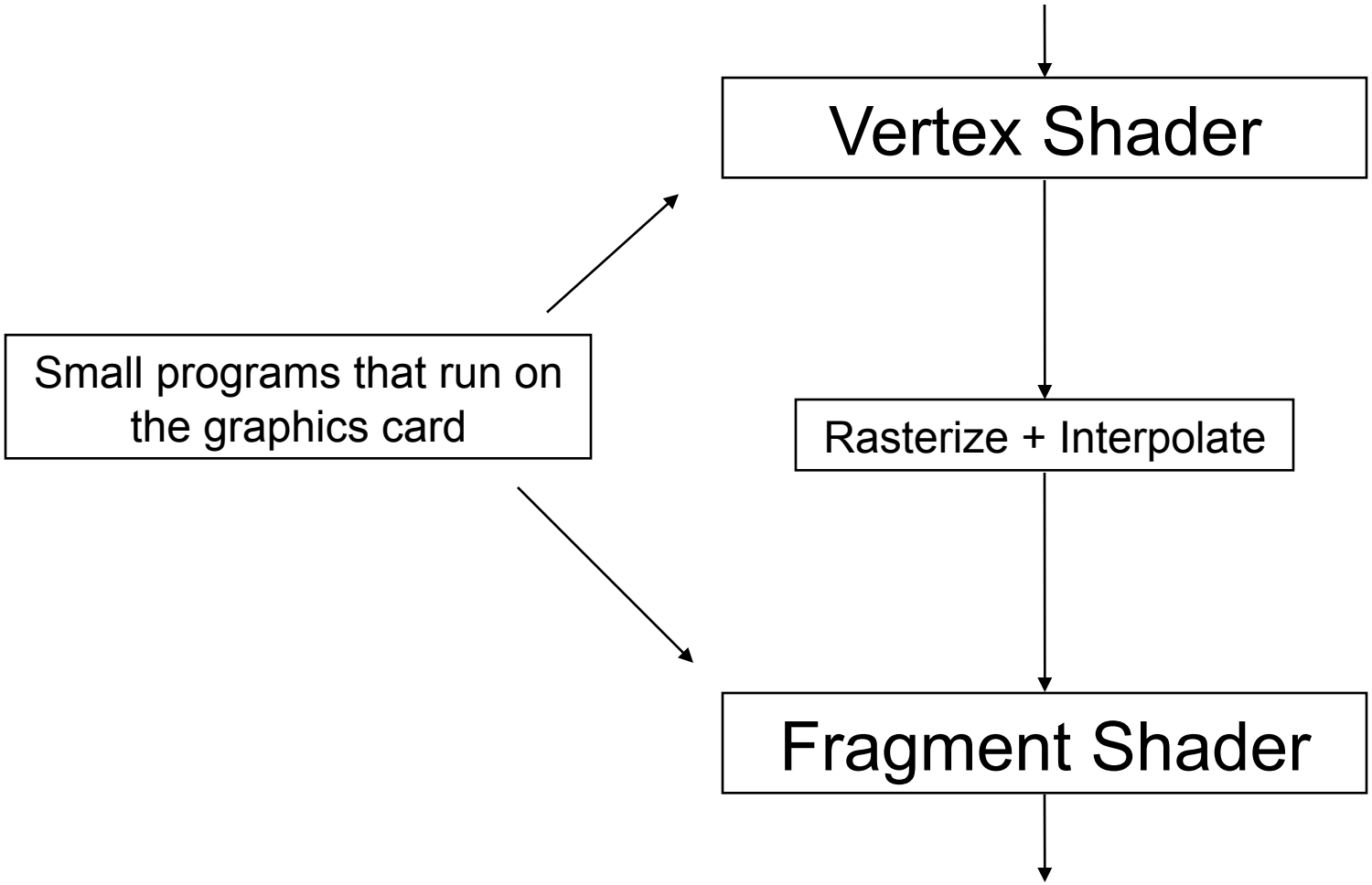
The OpenGL Pipeline



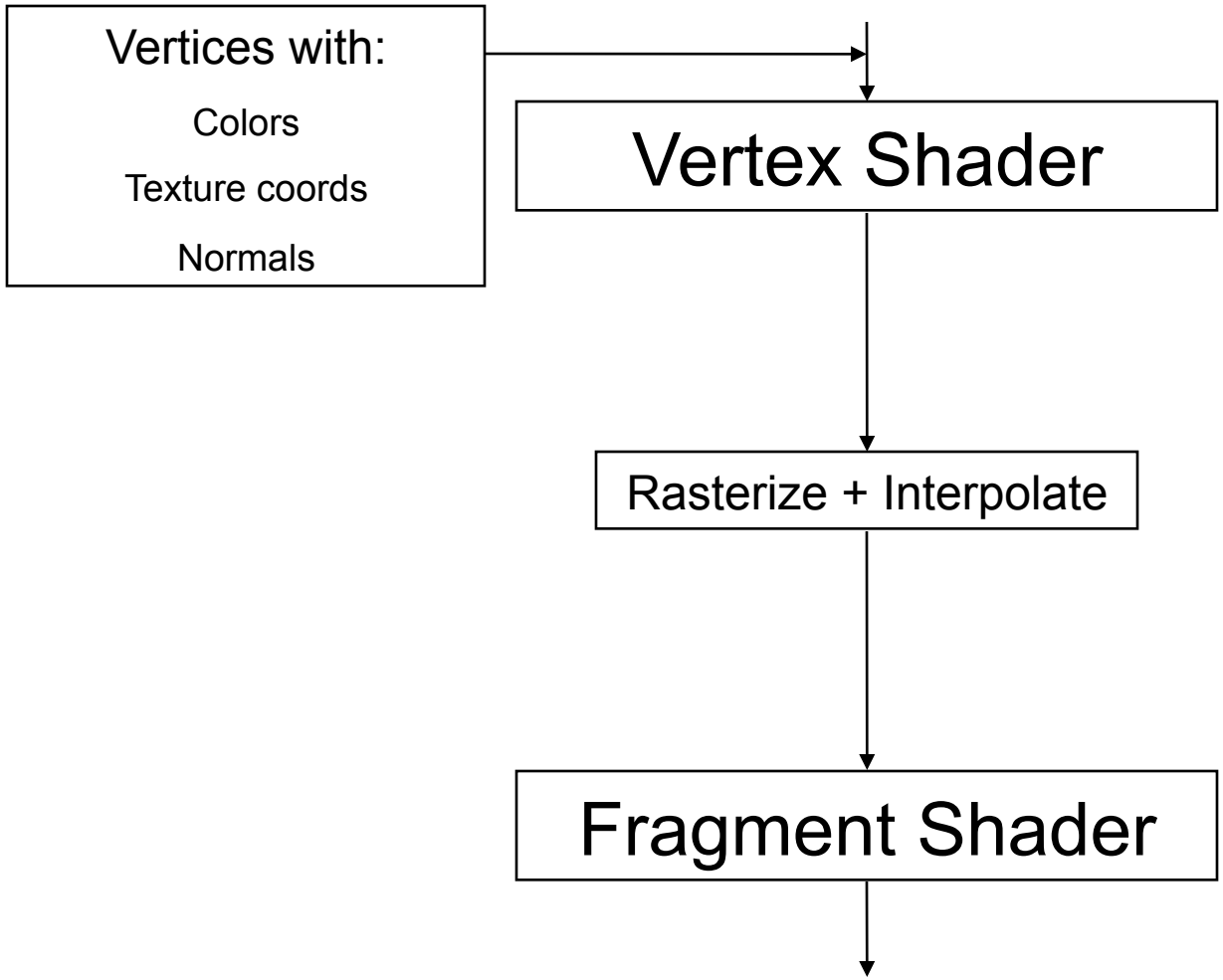
The OpenGL Pipeline with Shaders



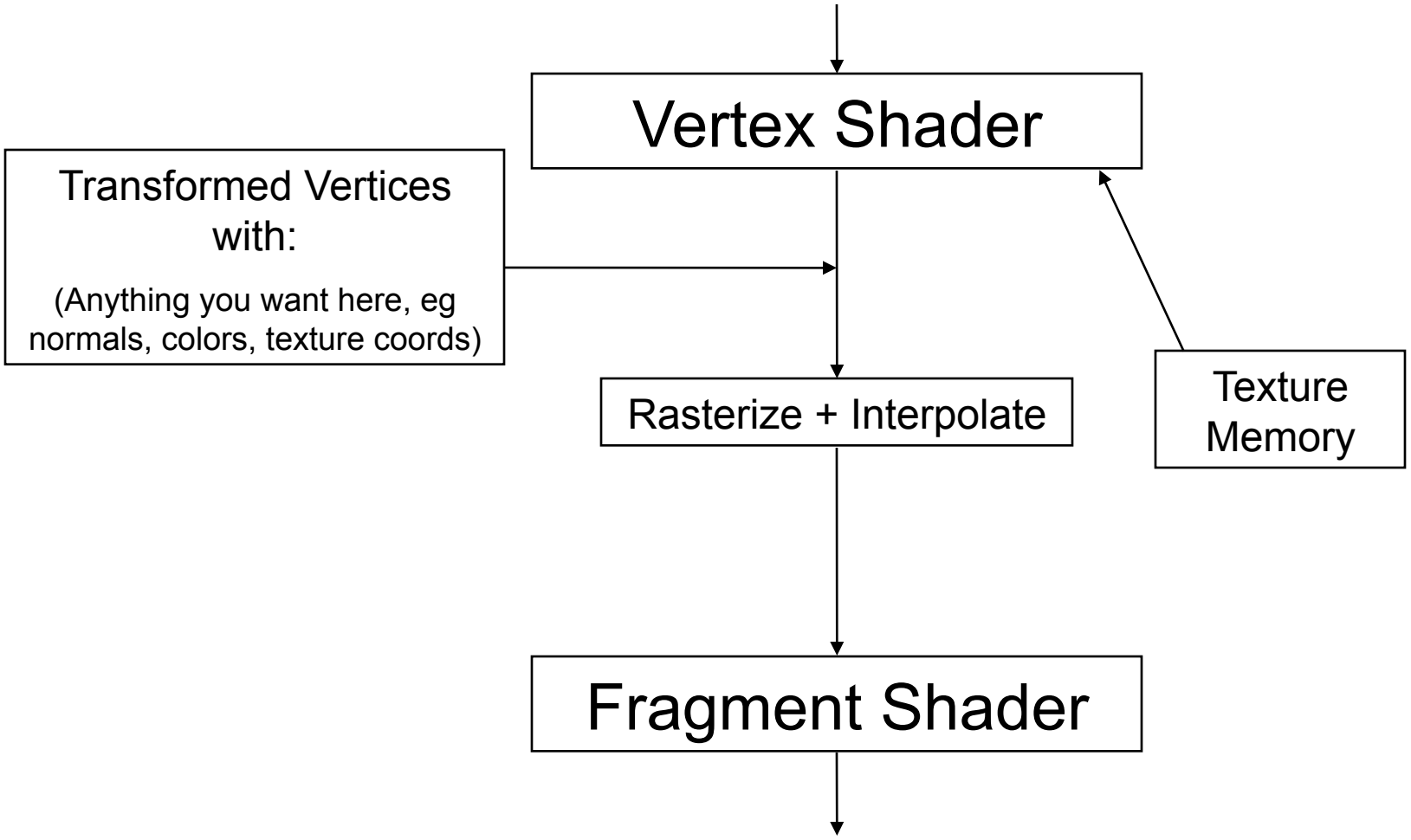
The OpenGL Pipeline with Shaders



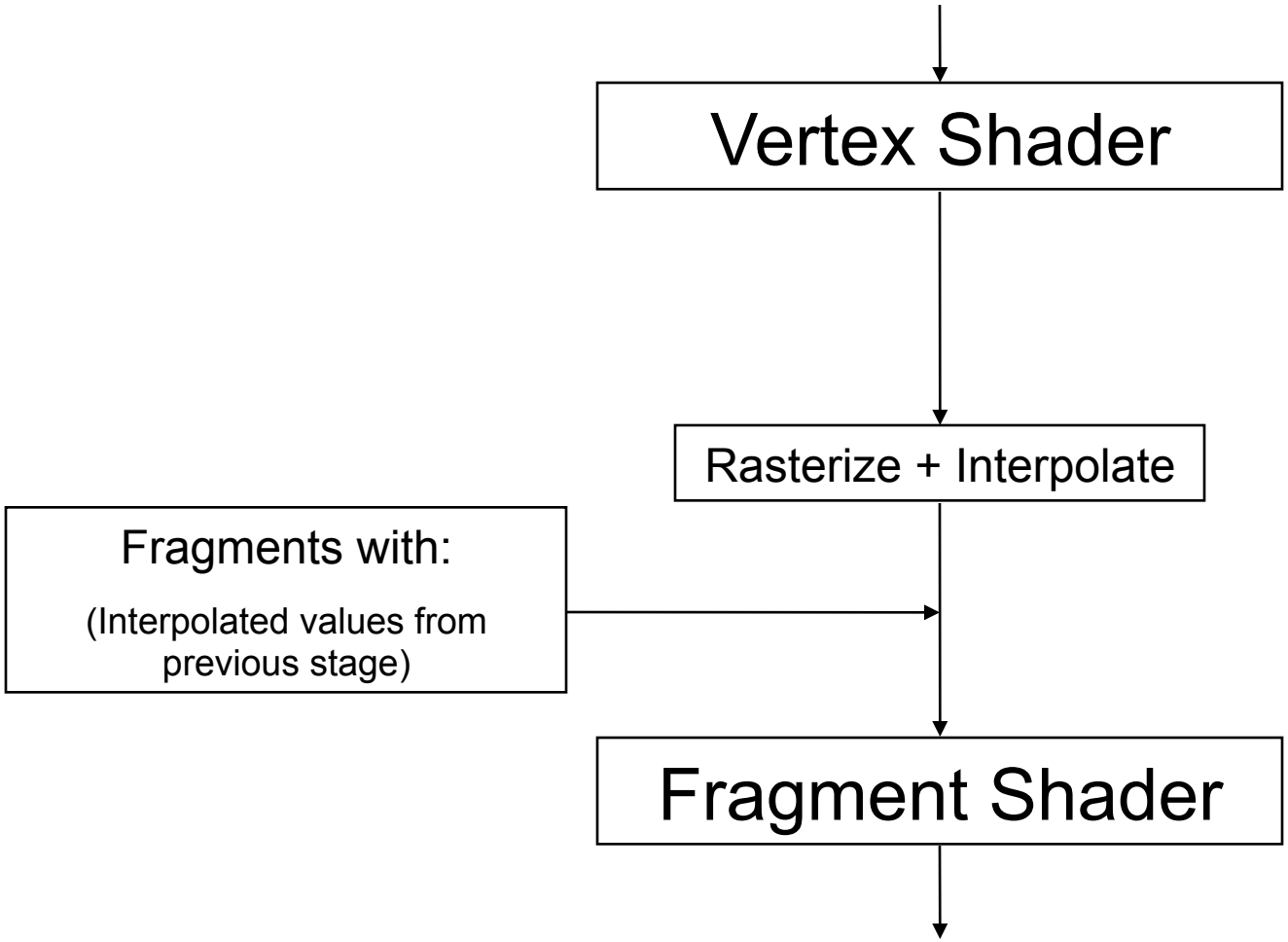
The OpenGL Pipeline with Shaders



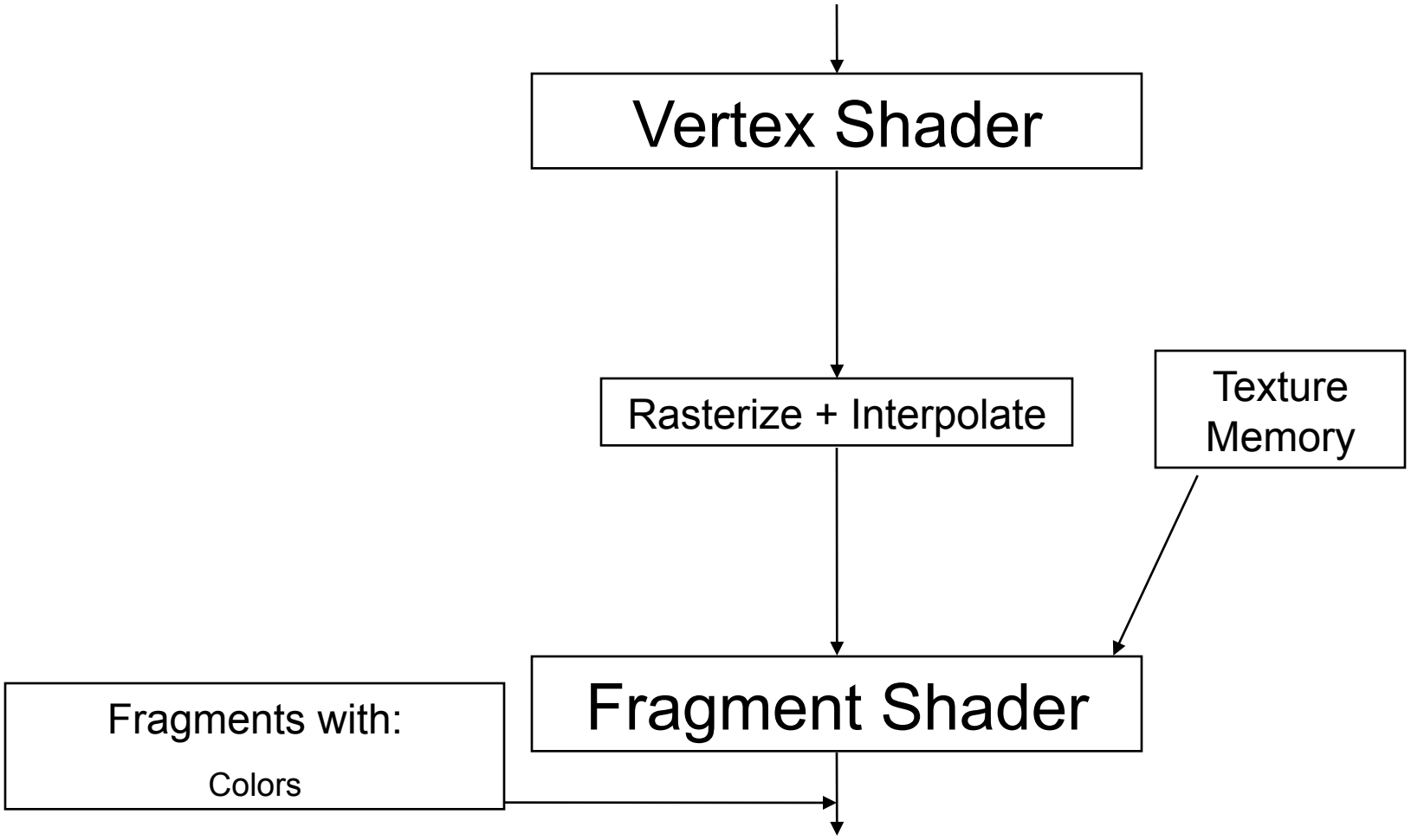
The OpenGL Pipeline with Shaders



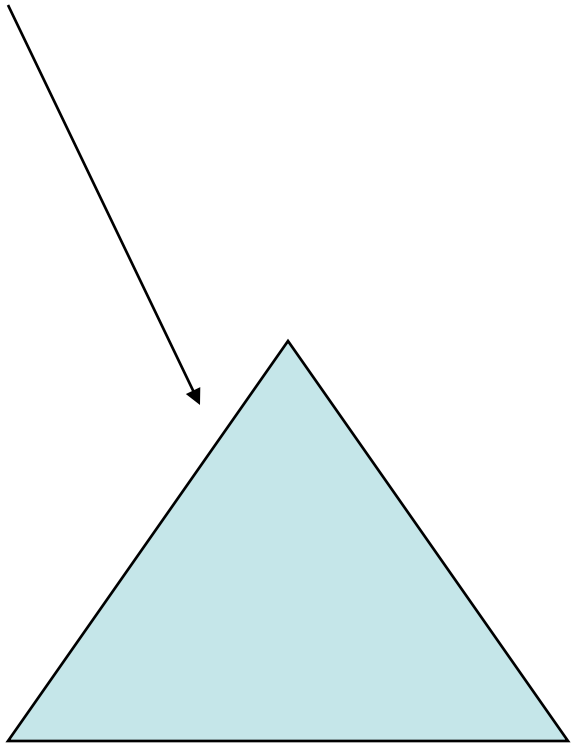
The OpenGL Pipeline with Shaders



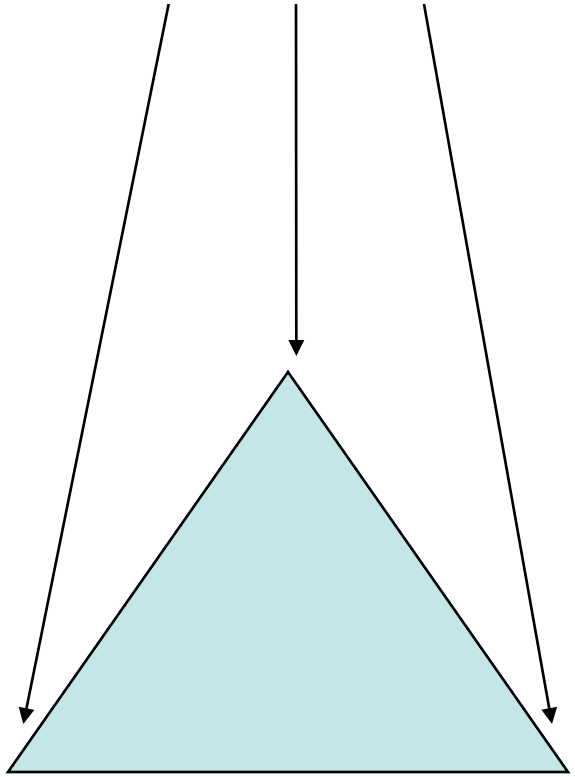
The OpenGL Pipeline with Shaders



How many times are the fragment and vertex shaders executed on this scene

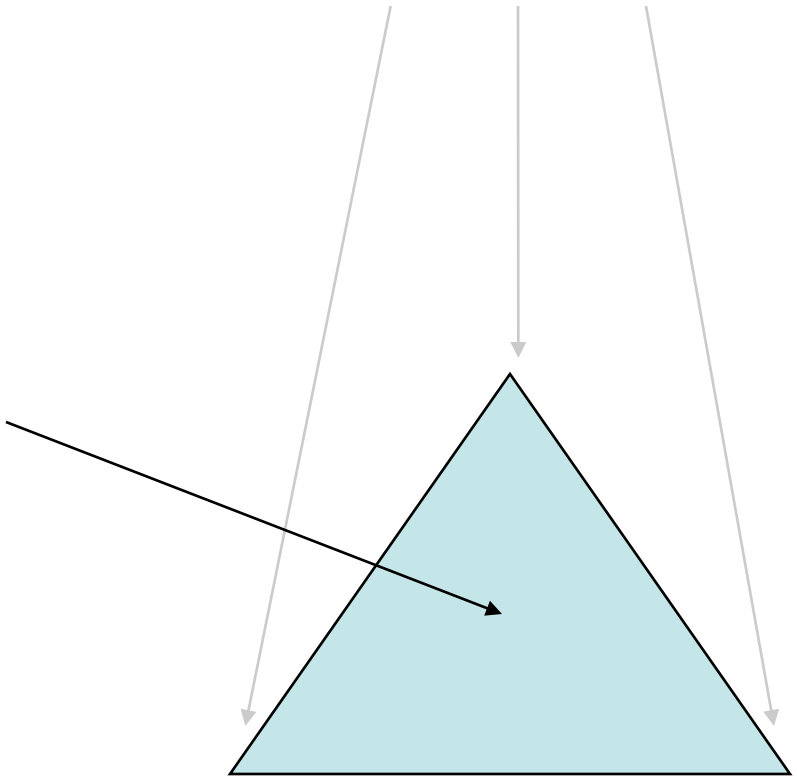


Vertex shader runs once per vertex



Vertex shader runs once per vertex

Fragment shader
runs once per pixel



Vertex Shader

```
void main() {
    gl_Position = gl_Vertex;
}
```

Fragment Shader

```
void main() {
    gl_FragColor = vec4(1, 0, 0, 1);
}
```

Vertex Shader

```
void main() {
    gl_Position = gl_Vertex;
}
```

Note:

Ignores modelview and projection matrices

Fragment Shader

```
void main() {
    gl_FragColor = vec4(1, 0, 0, 1);
}
```

GLSL Types

- Float types:
  - float, vec2, vec3, vec4, mat2, mat3, mat4
- Bool types:
  - bool, bvec2, bvec3, bvec4
- Int types:
  - int, ivec2, ivec3, ivec4
- Texture types:
  - sampler1D, sampler2D, sampler3D

GLSL Types

- A variable can optionally be:
  - Const
  - Uniform
    - Can be set from c, outside glBegin/glEnd
  - Attribute
    - Can be set from c, per vertex
  - Varying
    - Set in the vertex shader, interpolated and used in the fragment shader

Vertex Shader

```
attribute float vertexGrayness;
varying float fragmentGrayness;
void main() {
    gl_Position = gl_Vertex;
    fragmentGrayness = vertexGrayness
}
```

Fragment Shader

```
uniform float brightness;
varying float fragmentGrayness;
void main() {
    const vec4 red = vec4(1, 0, 0, 1);
    const vec4 gray = vec4(0.5, 0.5, 0.5, 1);
    gl_FragColor = brightness * (red * (1 - fragmentGrayness) +
                                   gray * fragmentGrayness);
}
```

Vertex Shader

```
attribute float vertexGrayness;
varying float fragmentGrayness;
void main() {
    gl_Position = gl_Vertex;
    fragmentGrayness = vertexGrayness
}
```

Set once per vertex, like color, normal, or texcoord.

Fragment Shader

```
uniform float brightness;
varying float fragmentGrayness;
void main() {
    const vec4 red = vec4(1, 0, 0, 1);
    const vec4 gray = vec4(0.5, 0.5, 0.5, 1);
    gl_FragColor = brightness * (red * (1 - fragmentGrayness) +
                                 gray * fragmentGrayness);
}
```

Vertex Shader

```
attribute float vertexGrayness;
varying float fragmentGrayness;
void main() {
    gl_Position = gl_Vertex;
    fragmentGrayness = vertexGrayness
}
```

Set once per polygon, outside glBegin/glEnd

Fragment Shader

```
uniform float brightness;
varying float fragmentGrayness;
void main() {
    const vec4 red = vec4(1, 0, 0, 1);
    const vec4 gray = vec4(0.5, 0.5, 0.5, 1);
    gl_FragColor = brightness * (red * (1 - fragmentGrayness) +
                                 gray * fragmentGrayness);
}
```

Vertex Shader

```
attribute float vertexGrayness;
varying float fragmentGrayness;
void main() {
    gl_Position = gl_Vertex;
    fragmentGrayness = vertexGrayness
}
```

Matching Declarations

Fragment Shader

```
uniform float brightness;
varying float fragmentGrayness;
void main() {
    const vec4 red = vec4(1, 0, 0, 1);
    const vec4 gray = vec4(0.5, 0.5, 0.5, 1);
    gl_FragColor = brightness * (red * (1 - fragmentGrayness) +
                                 gray * fragmentGrayness);
}
```

Vertex Shader

```
attribute float vertexGrayness;

varying float fragmentGrayness;

void main() {

    gl_Position = gl_Vertex;

    fragmentGrayness = vertexGrayness

}
```

Set at each vertex in vertex shader

Interpolated in hardware

Fragment Shader

```
uniform float brightness;

varying float fragmentGrayness;

void main() {

    const vec4 red = vec4(1, 0, 0, 1);

    const vec4 gray = vec4(0.5, 0.5, 0.5, 1);

    gl_FragColor = brightness * (red * (1 - fragmentGrayness) +

                                 gray * fragmentGrayness);

}
```

Available in fragment shader

Creating your own shaders

- Get at the extension functions
  - use glew http://glew.sourceforge.net/
- Load the shaders as strings
- Compile them
- Link them into a ‘program’
- Enable the program
- Draw something
- (Live Demo)

References

- A great tutorial and reference:
  - http://www.lighthouse3d.com/opengl/glsl/
- The spec for GLSL, includes all the available builtin functions and state:
  - http://www.opengl.org/documentation/glsl/
- The OpenGL ‘orange book’ on shaders is also good

Multipass Techniques

- Many cool effects are easier to achieve by rendering the scene multiple times:
  - Render the scene
  - Read the scene into a texture
  - Render that texture on the screen in some interesting way

Motion Blur

1) T = empty texture
2) Render a slightly dimmed T
3) Render the scene
4) Copy the framebuffer into T
5) Goto 2

(Live Demo)

Motion Blur: Two Questions

- Question 1: What's incorrect about this motion blur?

(from a sampling point of view)

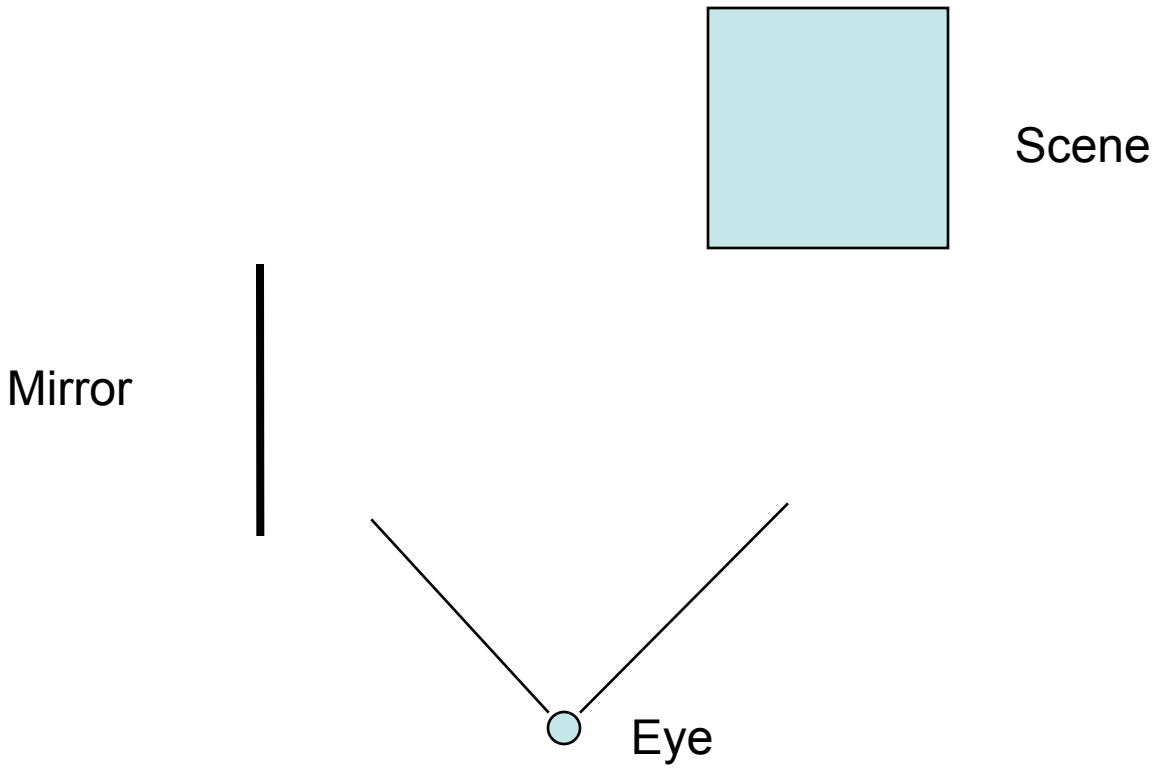
- Question 2: How could this be modified to make a zoom blur?

Sharpen Filter

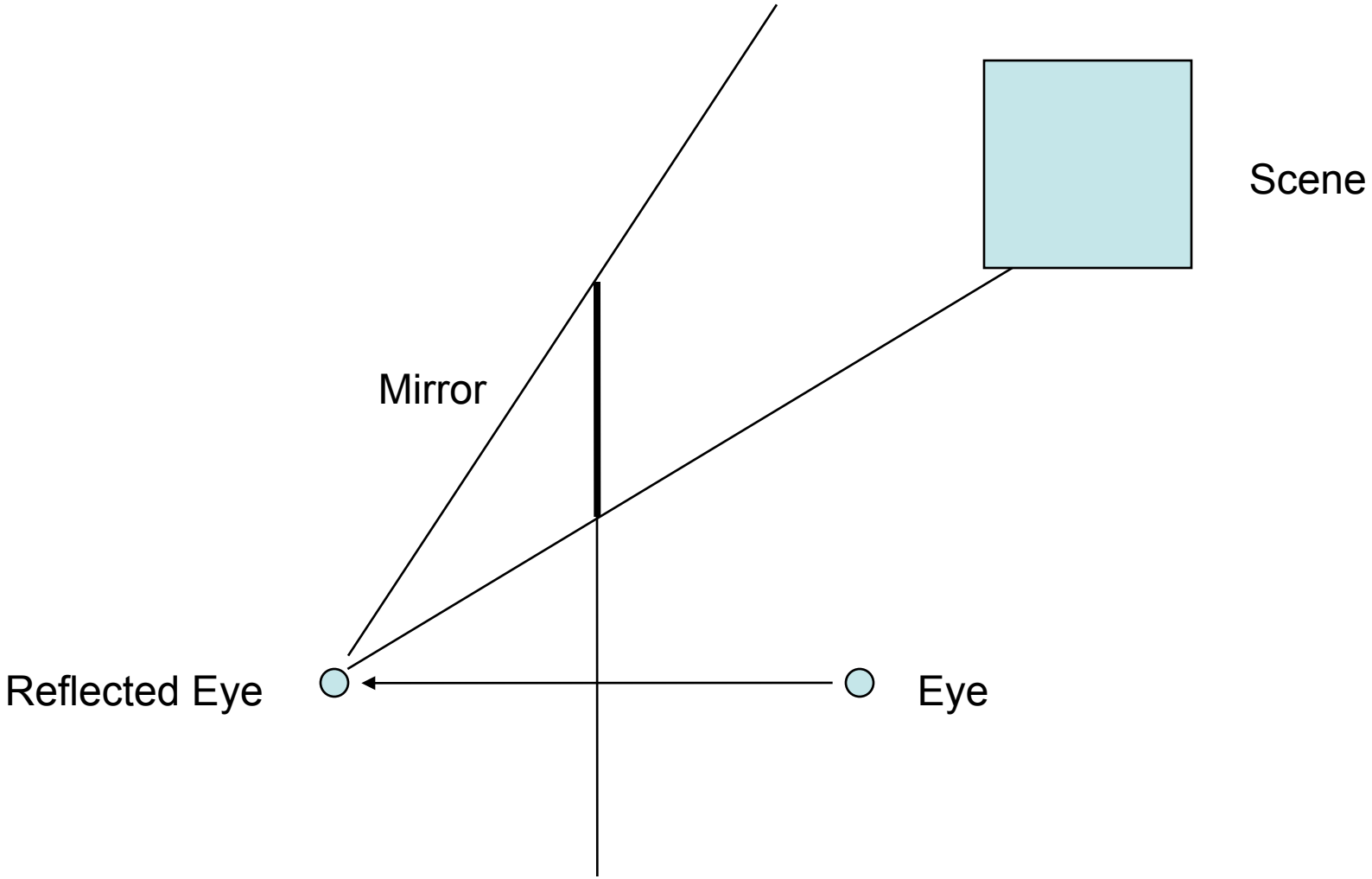
- Render the scene
- Save it as a texture
- Render the texture, using a shader to convolve the image

(Live Demo)

Reflections



Reflections



Reflections

- Reflect the eye about the plane of the mirror
- Render a view from that point, with frustum subtended by the mirror.
- Save the framebuffer into a texture
- Render the scene as normal, placing that texture over the mirror

Question

- How could you use multipass rendering, vertex, and fragment shaders to render a pond with ripples that reflects a cloudy sky?

Question

- How did I do this?
  - 4 quads, 3 are texture mapped
  - 1 shader
  - 2 framebuffer sized textures

(Live Demo)

Collision Detection and Culling

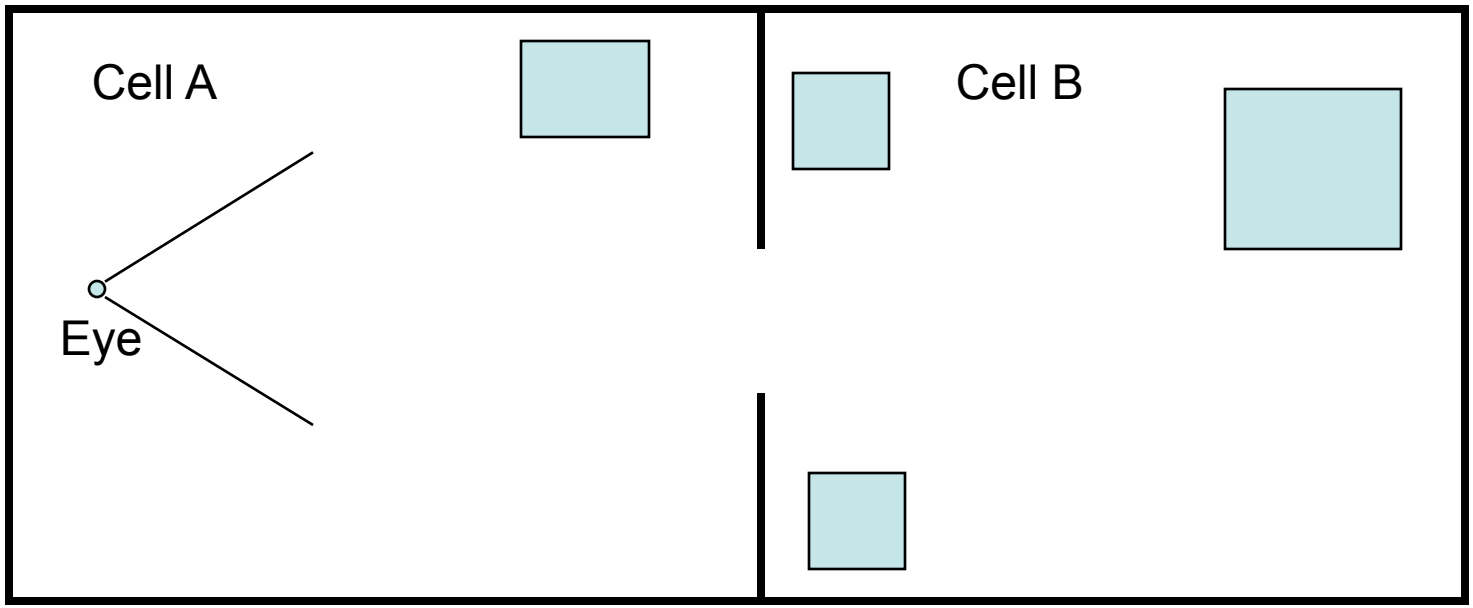
- The collision detection problem:
  - Given n objects with arbitrary shapes and positions, efficiently detect which objects intersect with which other objects.
  - $O(n^2)$ is bad
  - $O(n \ln(n))$ is good

Collision Detection and Culling

- The frustum culling problem:
  - Given n objects with arbitrary shapes and positions, efficiently detect which objects intersect with the viewing frustum.
  - O(n) is bad
  - O(ln(n)) is good
- Frustum culling is a special case of collision detection – collision with the frustum

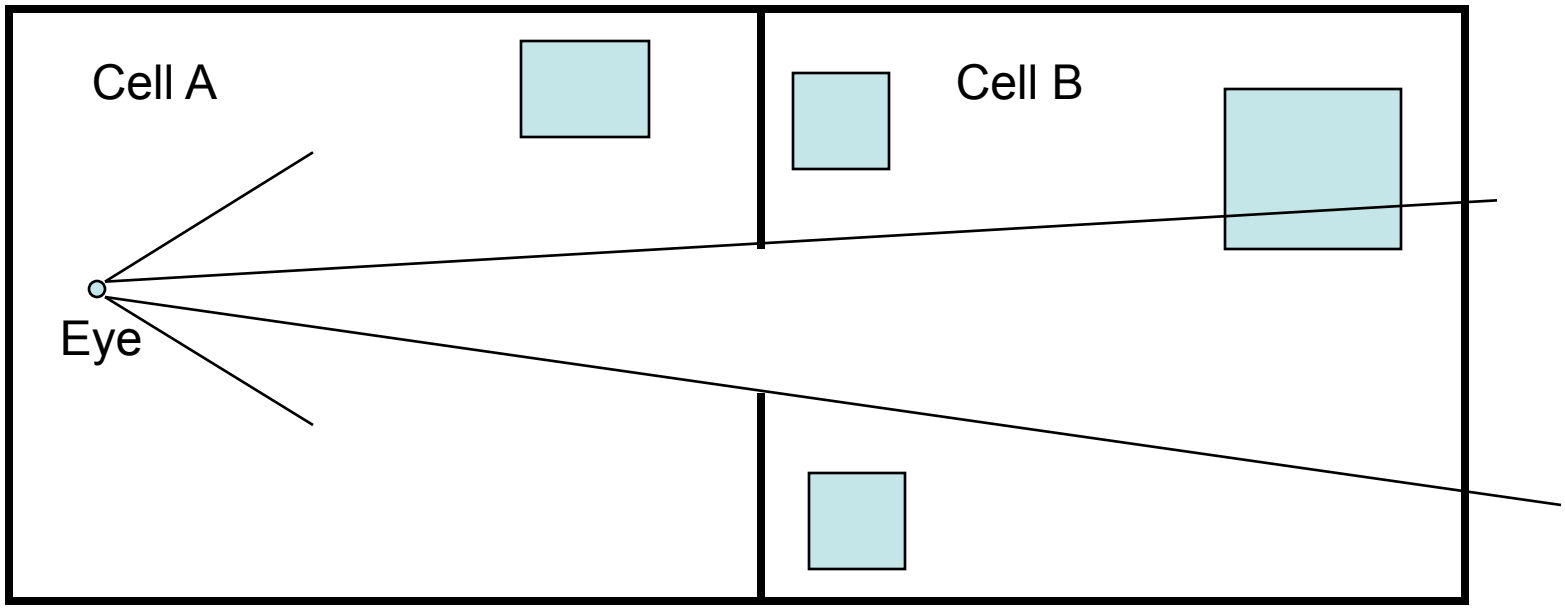
Portal Culling

- Your world is divided into cells, with rectangular doorways between the cells.



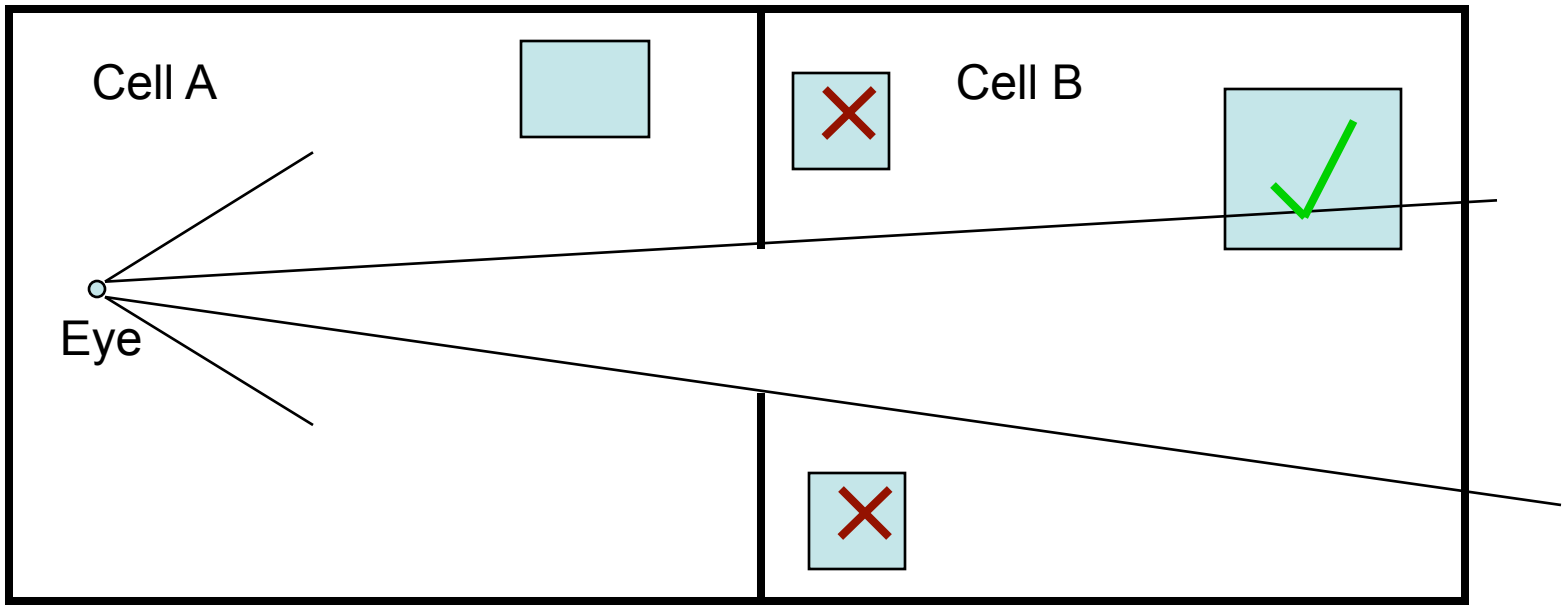
Portal Culling

- From cell A, what's visible in cell B?



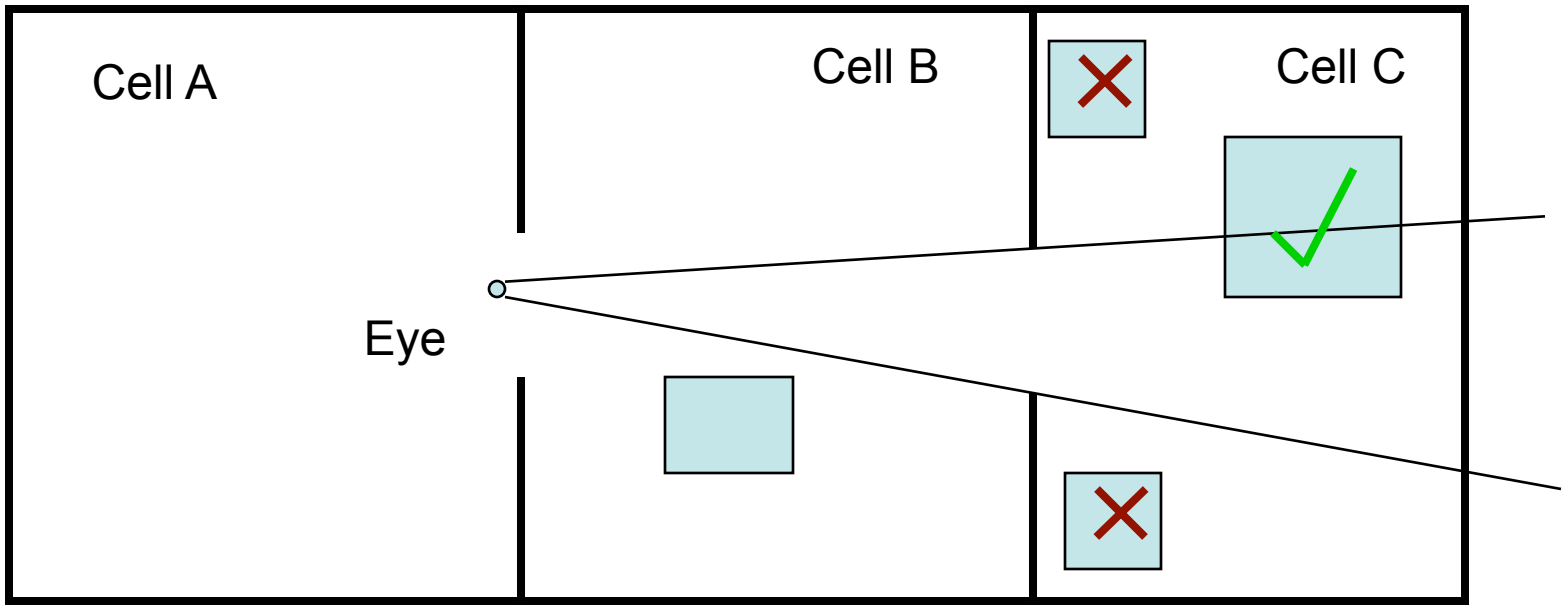
Portal Culling

- This is just frustum culling against the frustum subtended by the portal.



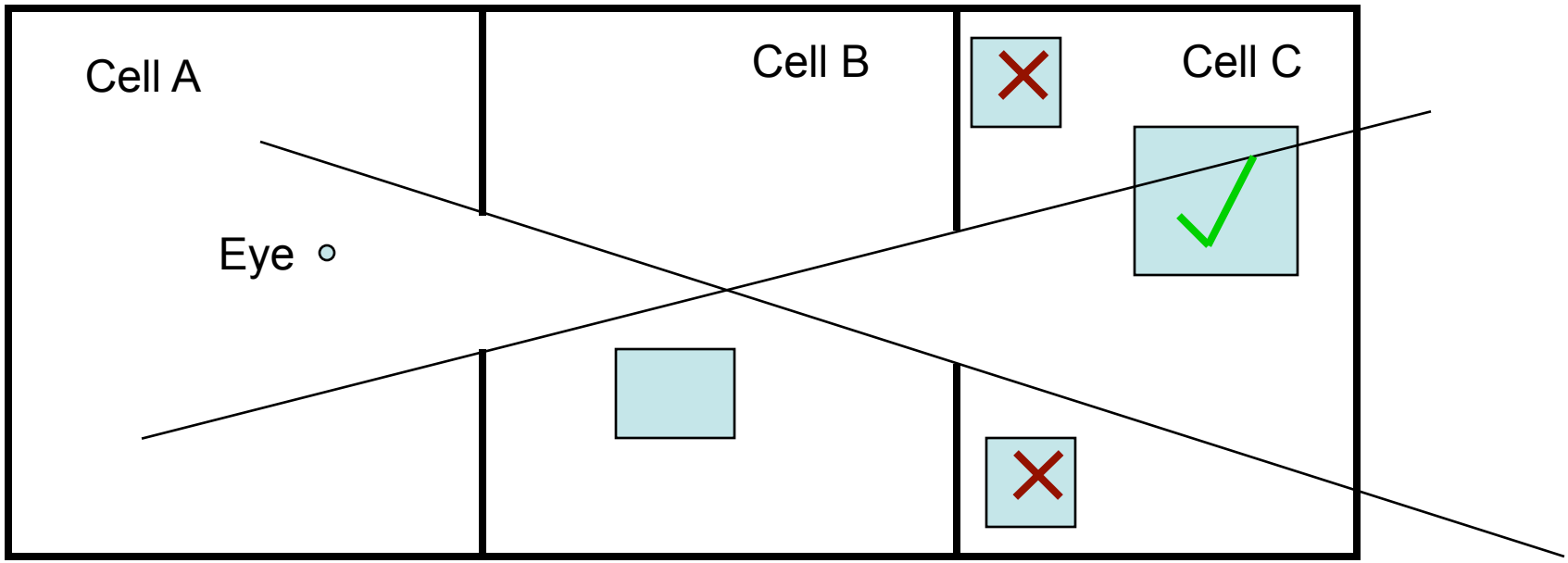
Portal Culling

- Some objects in C are NEVER visible from anywhere in A



Portal Culling

- For all pairs of cells x, y, precompute the set of objects in cell x that could be seen from cell y. Draw only them.

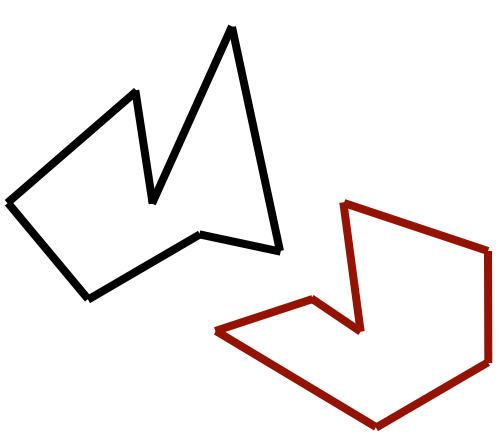


Collision Detection and Culling

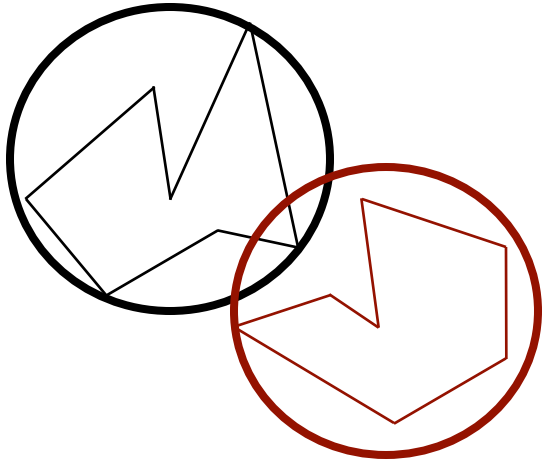
- Frustum Culling and portal culling are just collision detection against frustums.
- A good collision detection framework solves these culling problems for you.

Collision Detection Simplified

- Check (cheaply) against bounding volumes
  - spheres or boxes
- If they collide, do the expensive check



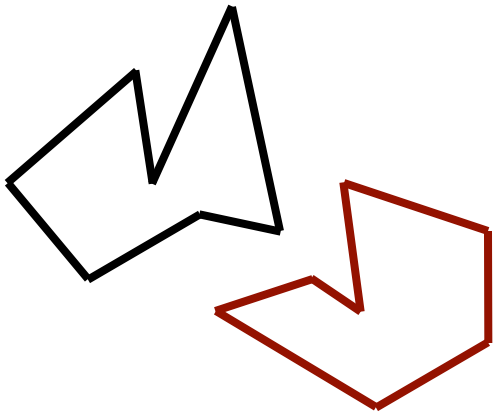
Hard



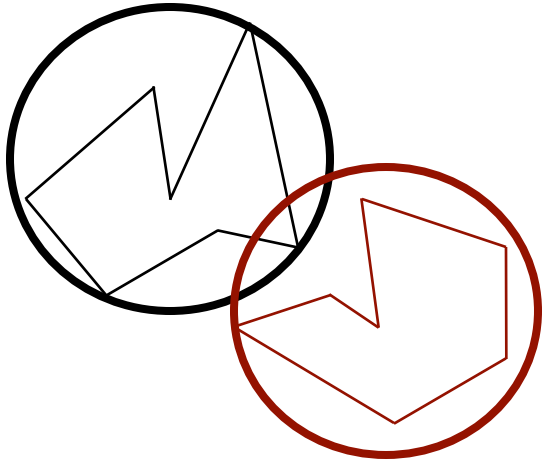
Easy

Collision Detection Simplified

- Why do you think spaceships in computer games have spherical shields?



Hard



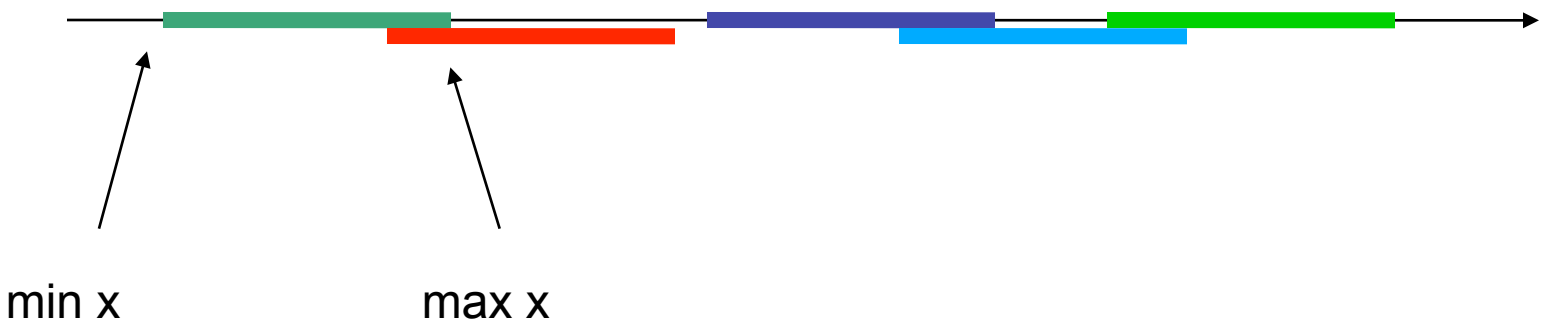
Easy

Collision Detection = Sorting

- What's a good collision detection algorithm for line segments of constant width in 1D?

Collision Detection = Sorting

- Sort by min x, then scan along, checking backwards until max x of the earlier polygon < min x of this polygon
- Complexity O(n ln(n))
- Still works for segments of varying width?



Collision Detection = Sorting

- Algorithm 1:
  - Sort all your bounding spheres along an arbitrary direction, eg z
  - Scan along the list, checking sphere/sphere distance between each sphere and the previous k
  - For spheres that collide, do more expensive polygon/ polygon checks (for culling we can be conservative and skip expensive checks).
- Only works if spheres have known max size, and are small. Otherwise k = O(n)

Collision Detection = Sorting

- Algorithm 2: Bucket sorting
  - Make a big array of cells for all space
  - Put a reference to each sphere in each cell it intersects (O(1) cells)
  - Traverse all the cells, looking for cells with more than one thing in them, do more accurate comparisons on those things.

Collision Detection = Sorting

- Algorithm 2: Bucket sorting
  - Have to be careful to avoid double counting collisions
  - Can rasterize large objects like frustums into the cells using a 3D rasterizer algorithm
  - Complexity = number of cells
  - Small cells make each object appear in many cells - inefficient
  - Large cells may have many objects per cell

Collision Detection = Sorting

- Algorithm 3: Hierachical bucket sorting
  - Apply algorithm 2 on **n** objects with **k** large cells
  - Each cell has O(n/k) objects in it that we need to check for collisions
  - Recurse with smaller cells!
  - If few objects in a cell, don’t bother recursing

Collision Detection = Sorting

- k = 8
  - ‘octtree collision detection’
  - Divide each cell in half in x, y, and z
- k = 2
  - ‘kd-tree collision detection’
  - Divide each cell in two in x OR y OR z
  - Doesn’t necessarily divide in half
- Both are good

Temporal Coherency

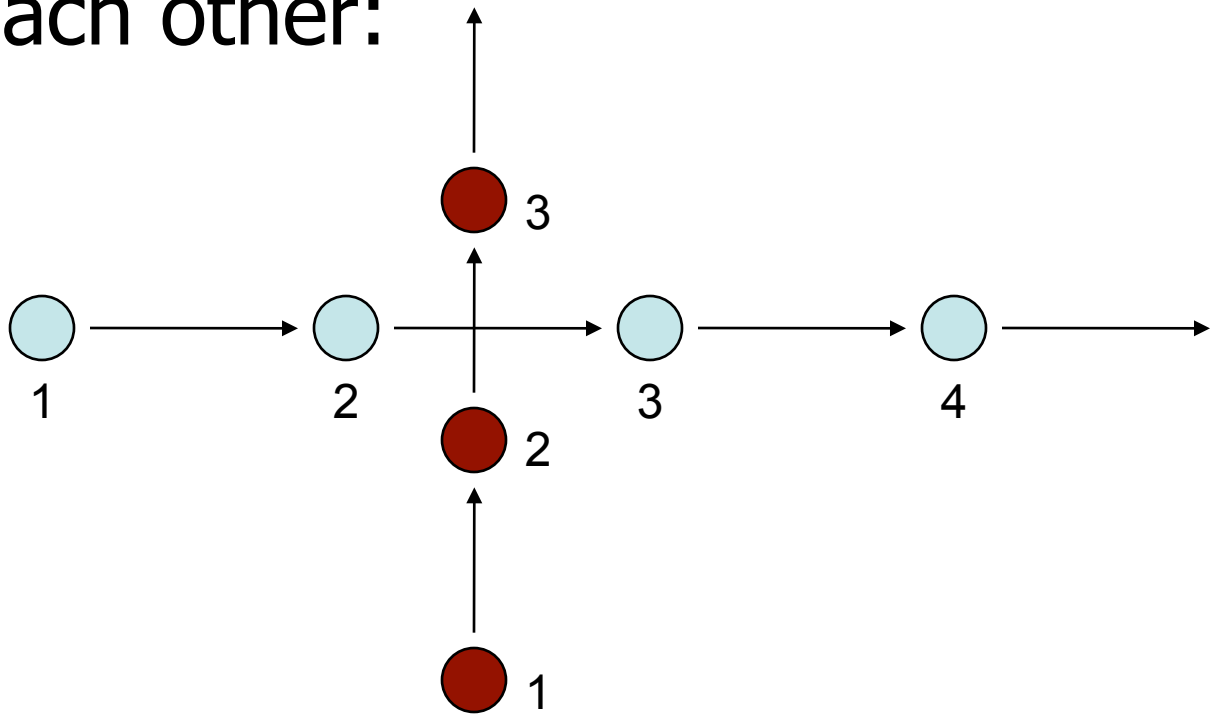
- We don’t want to recreate this whole recursive data structure every frame
- Static objects need not be reinserted every frame
- Moving objects don’t jump around at random, so you can be clever about moving them in the tree
  - O(1) insert cost instead of O(ln(n))

Low level collision detection

- You've detected two objects might collide based on bounding volumes... now what?
- A) Triangle vs triangle tests for each pair of triangles in the objects.
  - Tedious algorithm, look it up online
- B) Nearly collided is good enough
  - Culling
  - Arcade game collision detection

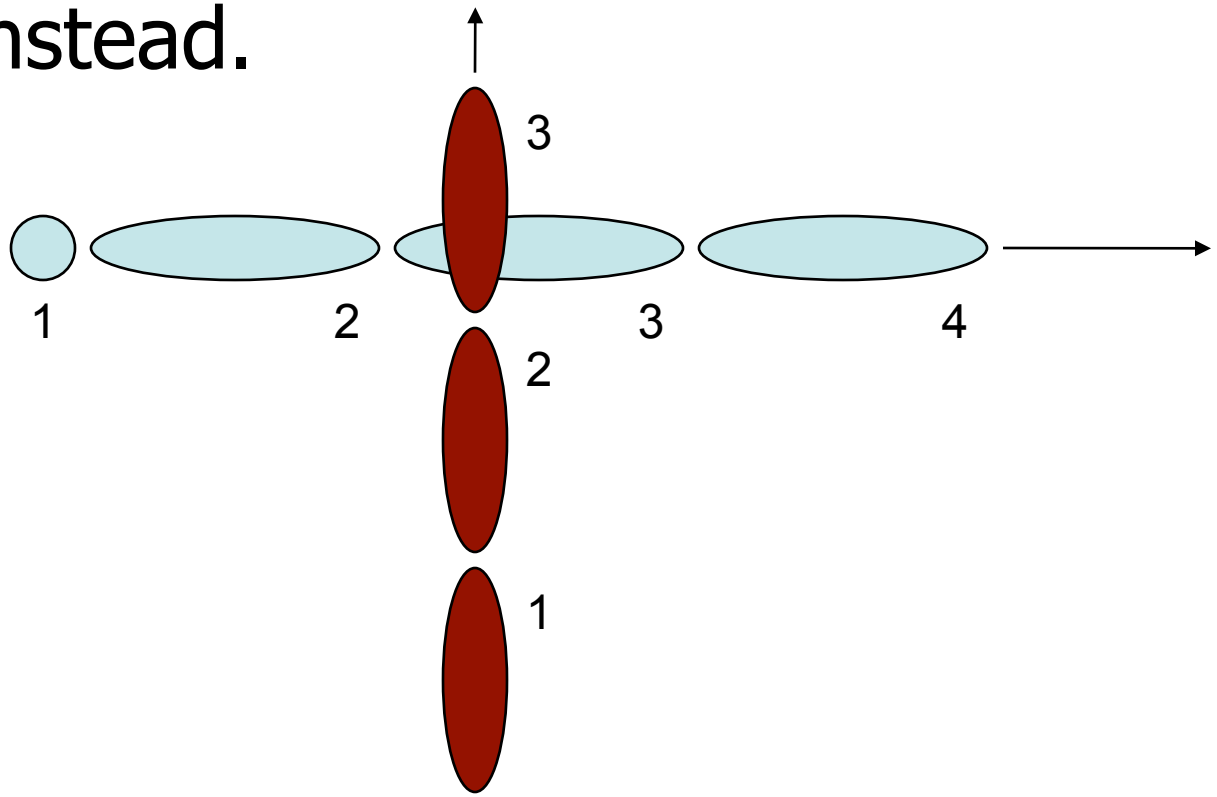
Fast Objects

- Quickly moving objects might jump over each other:



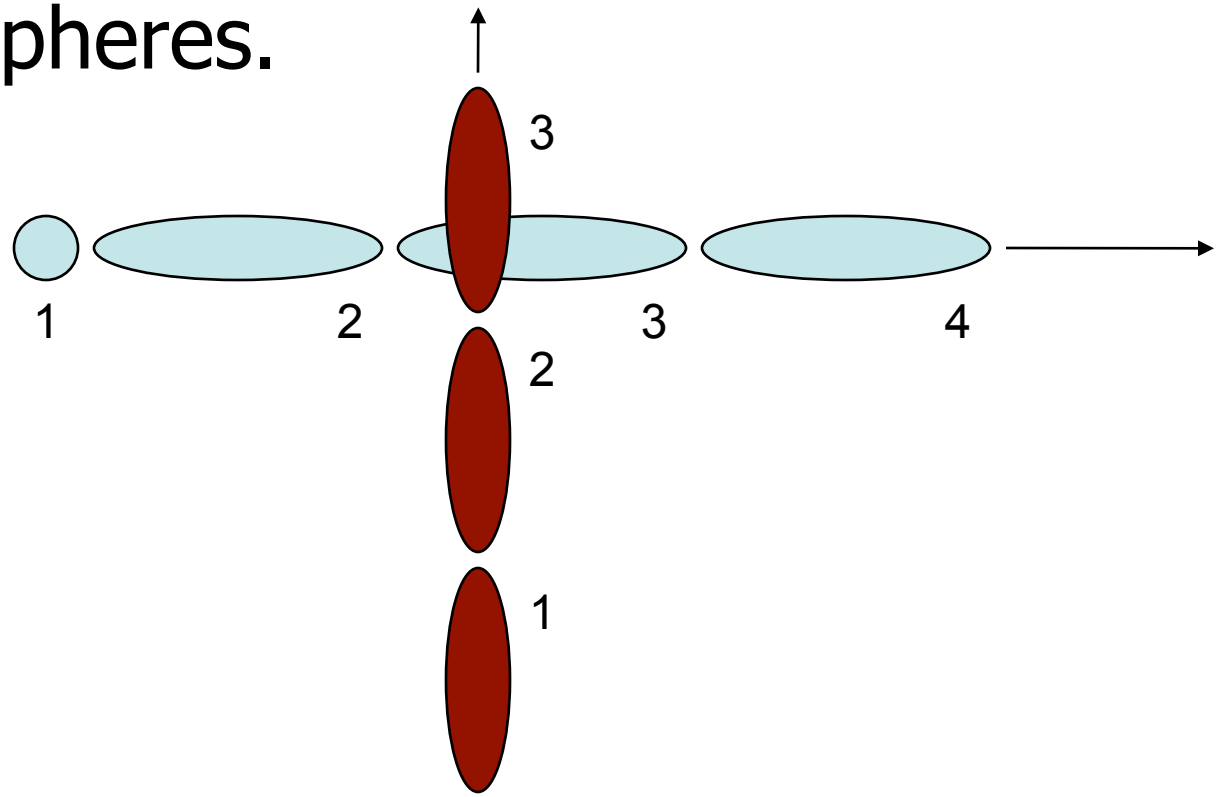
Fast Objects

- This is a sampling problem. Test collisions against motion blurred object shapes instead.



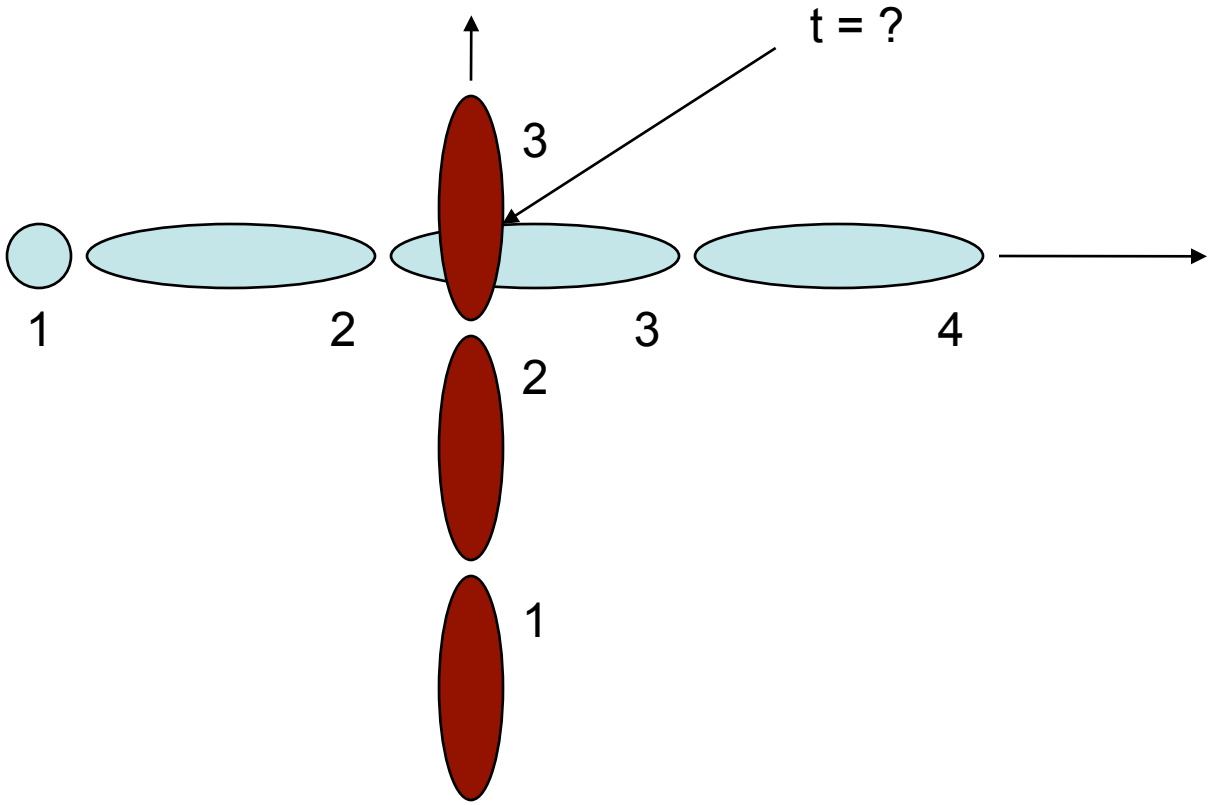
Fast Objects

- Use shortest distance between two line segments, or ellipsoids, or larger bounding spheres.



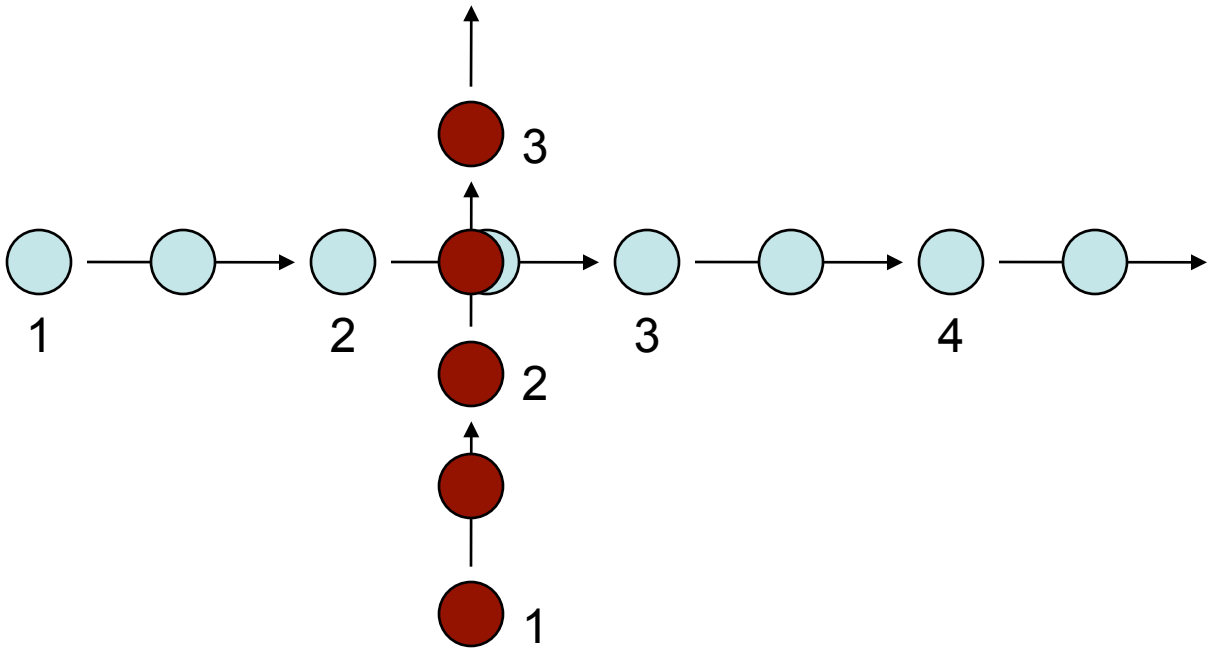
Fast Objects

- Solve for precise time to do good physics calculations



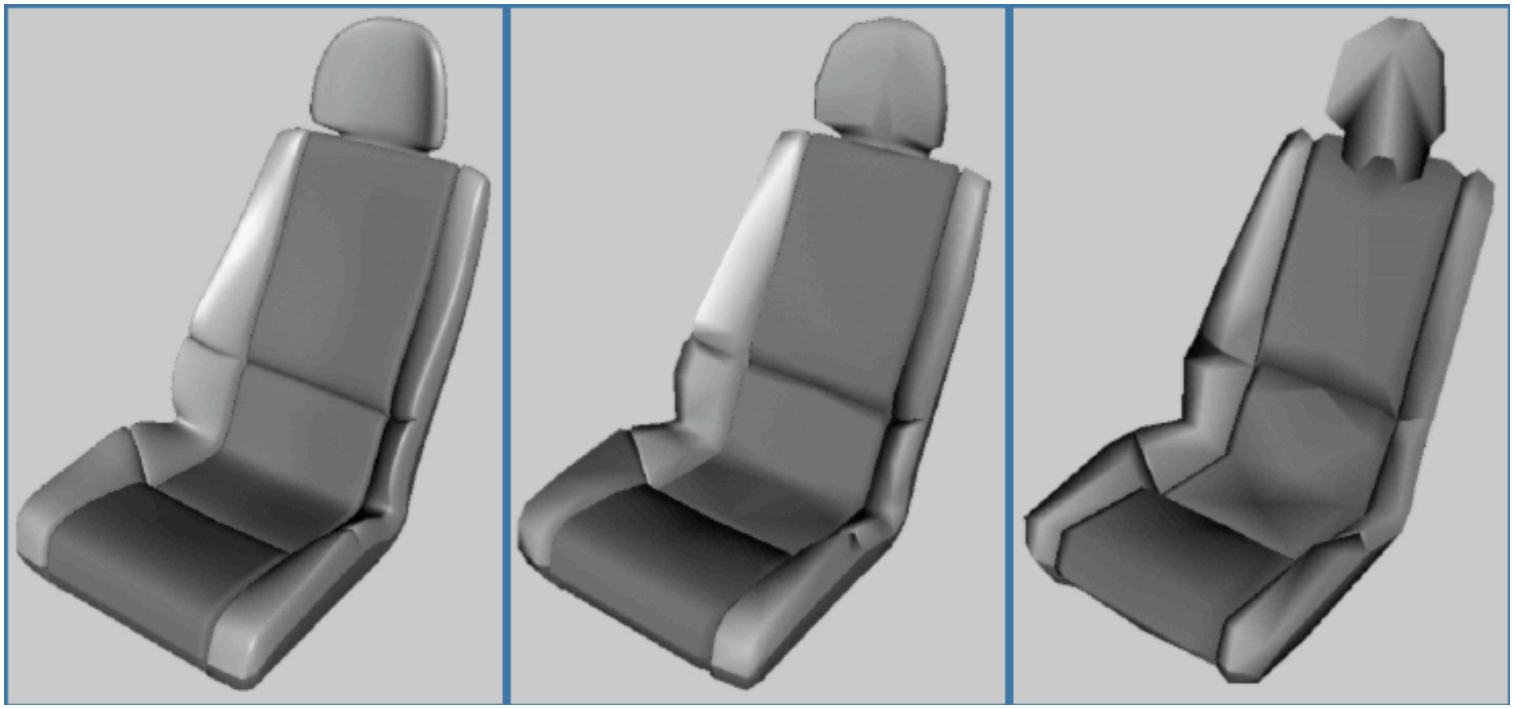
Fast Objects

- Alternative solution: Up the sampling rate
  - Do many physics time steps per drawn frame



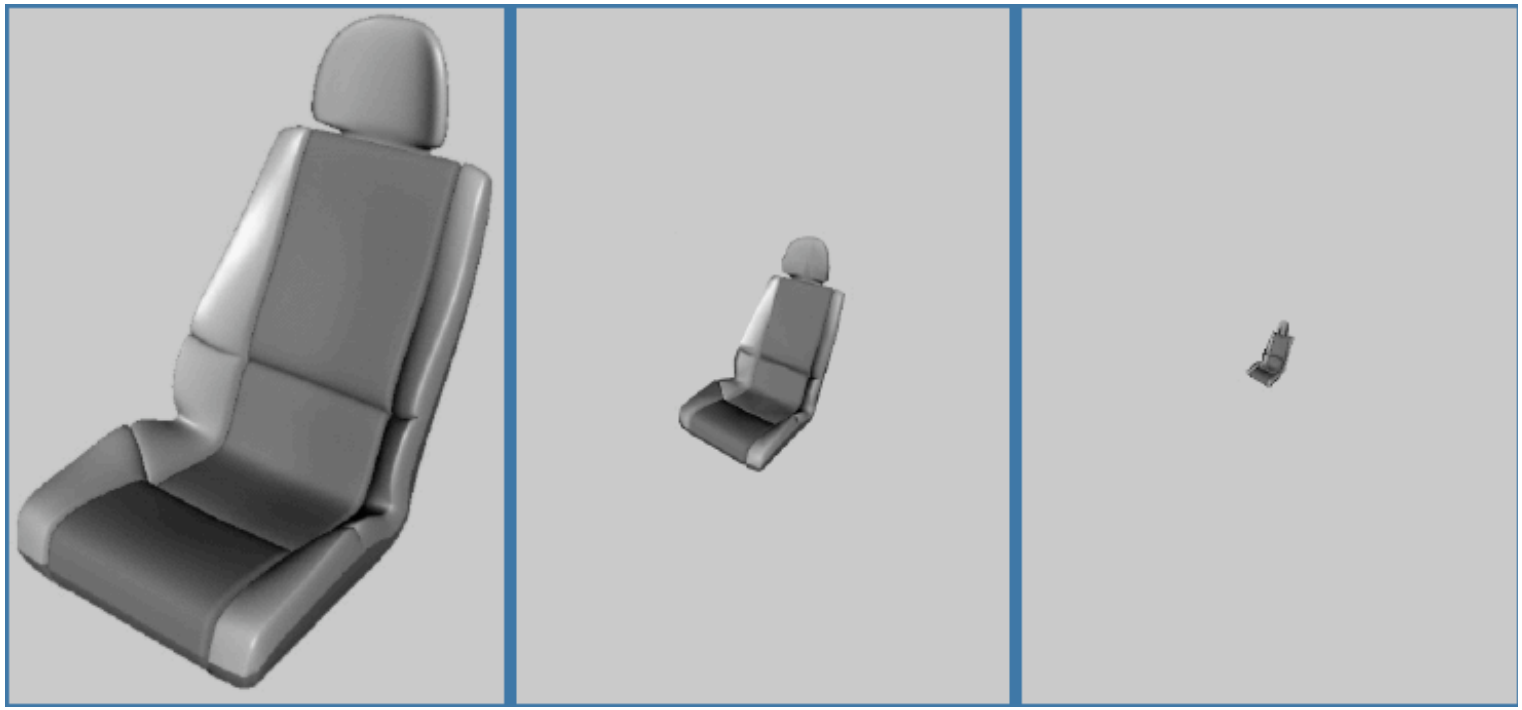
Detail Culling/LOD

- Things that are far away need not be drawn well
  - Use simplified meshes, simpler shaders, etc



Detail Culling/LOD

- Things that are far away need not be drawn well
  - Use simplified meshes, simpler shaders, etc



Detail Culling/LOD

- You need to transition between different levels of details
- As an object approaches you can
  - Pop from one model to the next
    - most games do this
  - Alpha blend from one model to the next
  - Subdivide polygons and morph into the new shape
    - Dynamic meshes are expensive!

3 Most Important Things?

Challenge

- How would you implement a rippling pond that accurately reflects the entire world, including near objects, far objects, and the sky?