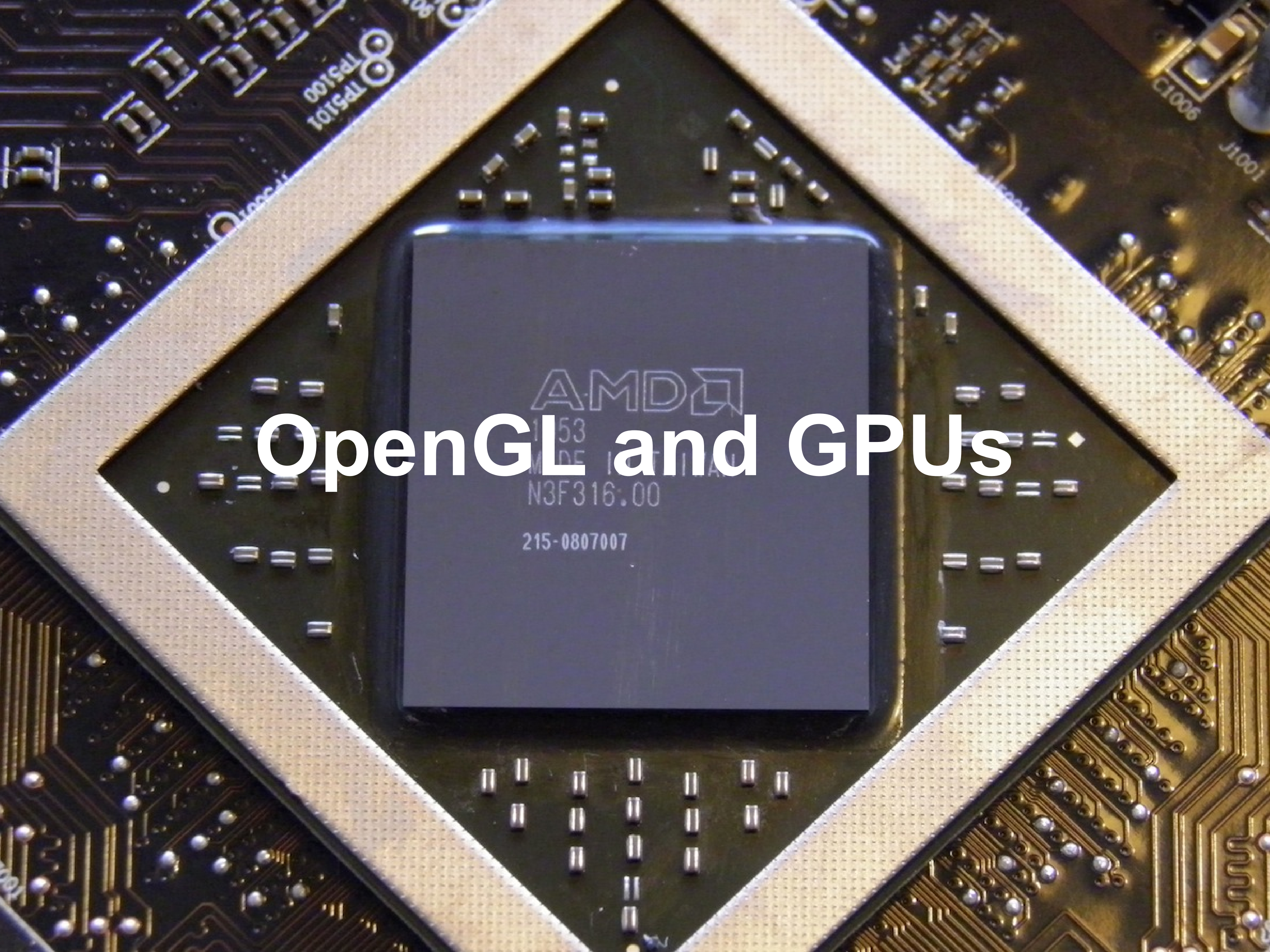
OpenGL and GPUs



What is OpenGL?

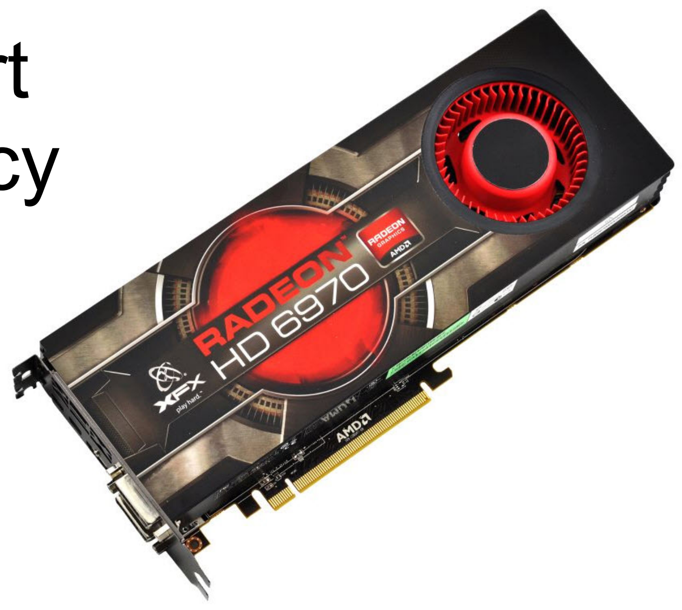
- OpenGL is a standard, like HTML and JPEG
    - OpenGL ES for mobile
    - WebGL for browsers
- GLU (OpenGL Utility Library)
    - Part of the OpenGL standard
    - Higher level: mesh primitives, tessellation
- GLUT (OpenGL Utility Toolkit)
    - Platform-independent windowing API for OpenGL
    - Not officially part of OpenGL
- Platform-dependent APIs for initialization
    - GLX (Linux), WGL (Windows), CGL (OS X)

History of OpenGL

- Started as standardization of IRIS GL, by Silicon Graphics
- Managed by non-profit Khronos Group
  - ATI, NVIDIA, Intel, Apple, Google, ...
  - Slow standardization process
  - OpenGL extensions for new features
- Originally a fixed-function pipeline
  - Now fully programmable

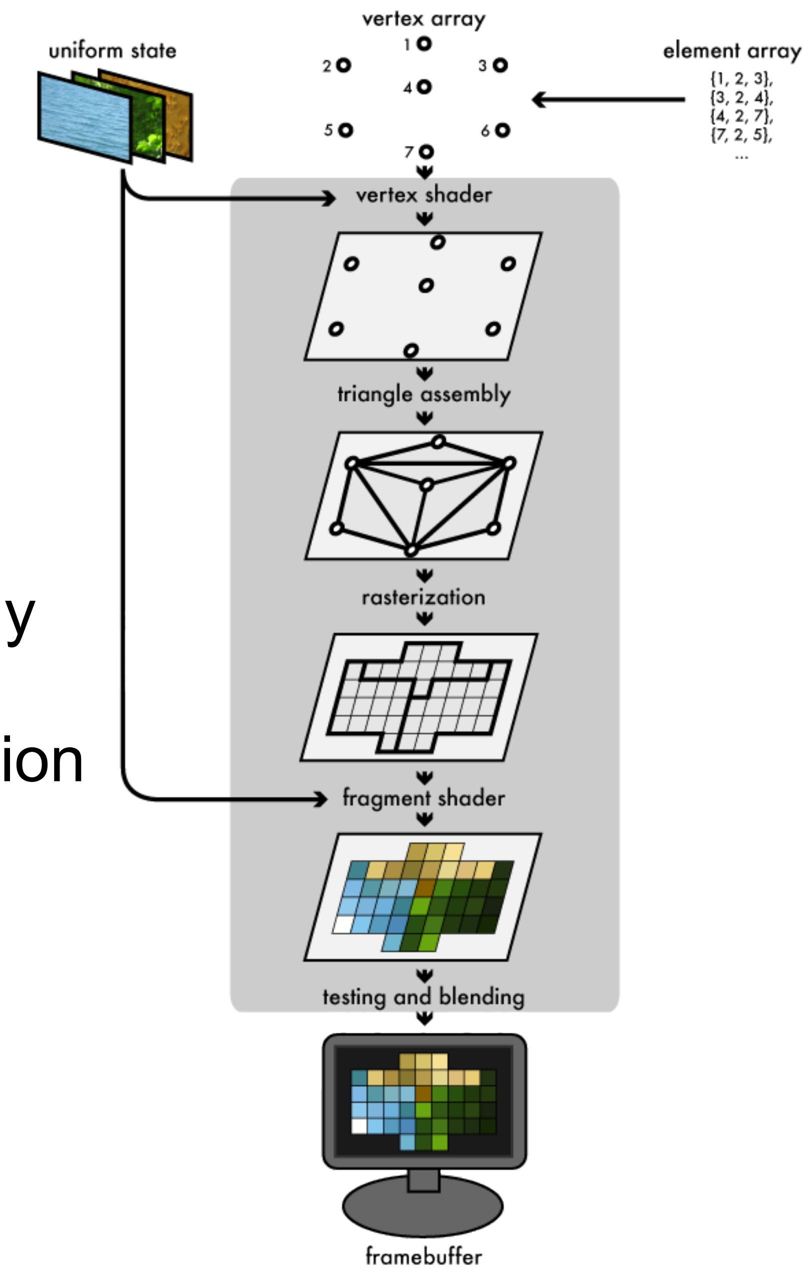
What is a GPU?

- Specialized hardware for 2D and 3D graphics
  - Special memory for textures and z-buffers
- Massively parallel
  - Hundreds of "cores"
- Hardware threading support
- High bandwidth, high latency
  - Hide latency with parallelism



How do GPUs Work?

- Stream processing
  - Restriction of parallel programming
  - Many kernels run independently on a read-only data set
  - OpenCL relaxes this restriction
- Pipeline in stages
  - Some stages are programmable
  - Modern pipelines are more complex (tessellation and geometry stages)



Stage 1: The Vertex Array

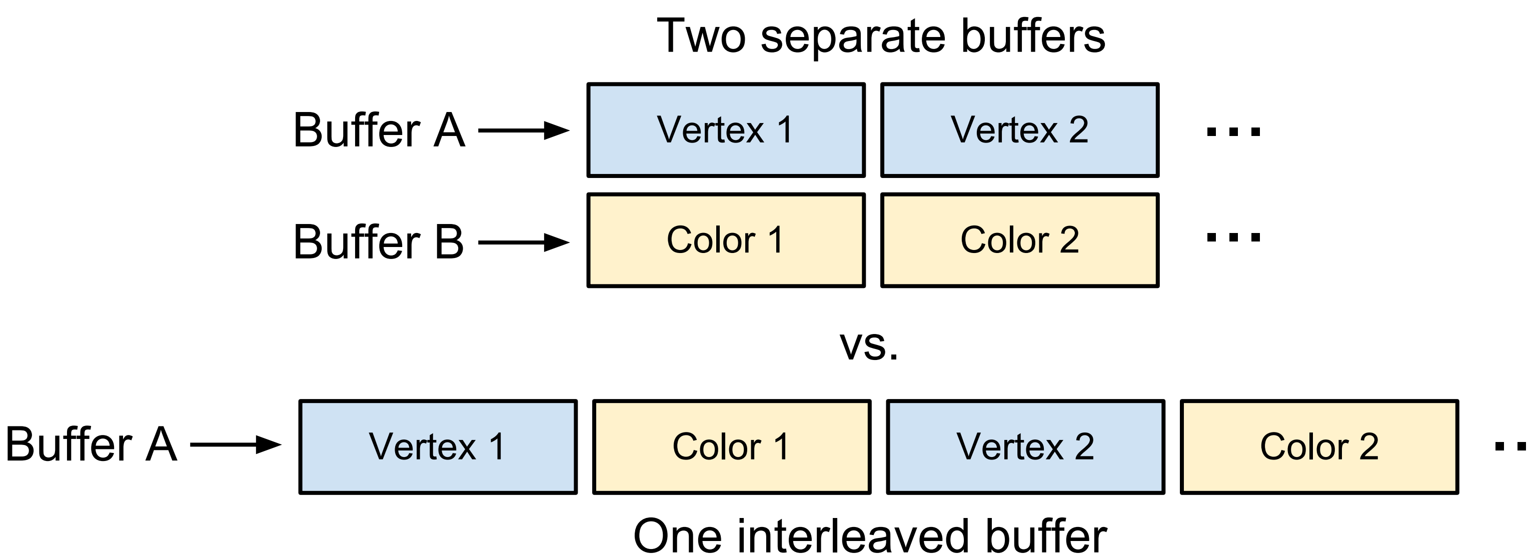
- Vertices specified by application
  - Also per-vertex data (colors, texture coordinates)
  - Ideally stored in optimized GPU memory
- OpenGL has several ways you shouldn't use:
  - Don't use glBegin() / glEnd()
    - Slowest method, start from scratch every frame
  - Don't use display lists
    - Pre-compiled OpenGL commands
    - Not any faster on some drivers
  - Don't use OpenGL "vertex arrays"
    - Data isn't stored on the GPU
    - Marginally better than calling glBegin() / glEnd()
  - Instead, use Vertex Buffer Objects (VBOs)

OpenGL Vertex Buffer Objects

- Handle to GPU memory for vertices
  - More efficient for geometry to stay on GPU
  - Contents may be modified after creation
- Can store any vertex attributes
  - Vertex position, normal, texture coordinates, ...

OpenGL Vertex Buffer Objects

- Array-of-structs vs struct-of-arrays
  - Improve static object speed by interleaving data



- Tradeoffs
  - Cache locality
  - Repeatedly modifying just one attribute

OpenGL Vertex Buffer Objects

```
// Initialization (interleaved data format)
float data[] = {
// Position   Color
   1,0,0,     1,0,0,   // Vertex 1
   0,1,0,     1,0,0,   // Vertex 2
   0,0,1,     1,1,0    // Vertex 3
};
unsigned int id;
glGenBuffers(1, &id); // Generate a new id
glBindBuffer(GL_ARRAY_BUFFER, id); // Bind the buffer

// Upload the data. GL_STATIC_DRAW is a hint that the buffer
// contents will be modified once but used many times.
glBufferData(GL_ARRAY_BUFFER, sizeof(data), data, GL_STATIC_DRAW);

glBindBuffer(GL_ARRAY_BUFFER, 0); // Unbind the buffer
```

OpenGL Vertex Buffer Objects

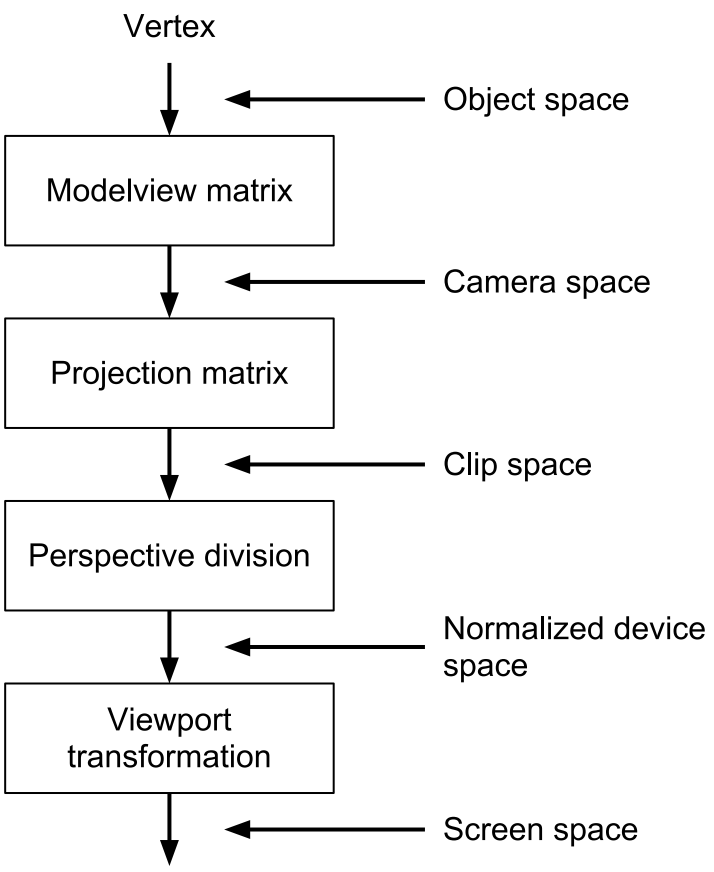
```
// Rendering (interleaved data format)
glBindBuffer(GL_ARRAY_BUFFER, id);
glEnableClientState(GL_VERTEX_ARRAY);
glEnableClientState(GL_COLOR_ARRAY);

unsigned int stride = sizeof(float) * (3 + 3); // Spacing between vertices
glVertexPointer(3, GL_FLOAT, stride, (char *) 0);
glColorPointer(3, GL_FLOAT, stride, (char *) (3 * sizeof(float)));
glDrawArrays(GL_TRIANGLES, 0, 3); // Draw 3 vertices (1 triangle)

glDisableClientState(GL_COLOR_ARRAY);
glDisableClientState(GL_VERTEX_ARRAY);
glBindBuffer(GL_ARRAY_BUFFER, 0);
```

OpenGL Vertex Buffer Objects

- Never initialize VBOs in the draw loop!
- More information
  - http://www.opengl.org/wiki/Vertex_Buffer_Object
  - http://www.opengl.org/wiki/VBO_-_more

Stage 2: The Vertex Shader

- Transform vertices
  - Object space to clip space
  - Perspective division done automatically in hardware
- Other computations
  - Vertex-based fog
  - Per-vertex lighting
- Pass data to fragment shader



Stage 2: The Vertex Shader

- Inputs
  - Uniform values (current matrices, lights)
  - Vertex attributes (colors, texture coordinates)
- Outputs
  - Vertex position: gl_Position (in clip space)
  - Varying values (colors, texture coordinates)
    - Will be inputs to fragment shader
    - Interpolated across triangles using perspective-correct linear interpolation

A Vertex Shader (GLSL)

```
// Input from C++, global per shader
uniform float scale;

// Output for the fragment shader, input for
// fragment shader (interpolated across the triangle)
varying vec4 vertex;

void main() {
    vertex = gl_Vertex; // Input from C++, per vertex
    vertex.xyz *= scale;
    gl_Position = gl_ModelViewProjectionMatrix * vertex;
}
```

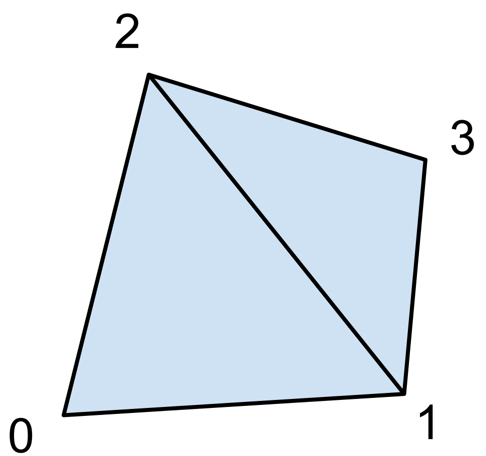
Vertex Processing on the GPU

- Vertices are processed in batches (of ~32)
  - Previous batch is cached
  - GPU runs through index buffer, reusing cached vertices
  - More performant to draw nearby triangles in sequence
  - Vertex shader likely run more than once per vertex

Automatically generated triangle strips are used to improve cache usage

Stage 3: Primitive Assembly

- Create primitives using index buffer
  - Index buffer contains offsets into vertex buffer
  - Used to share vertices between triangles
  - Also specifies rendering order



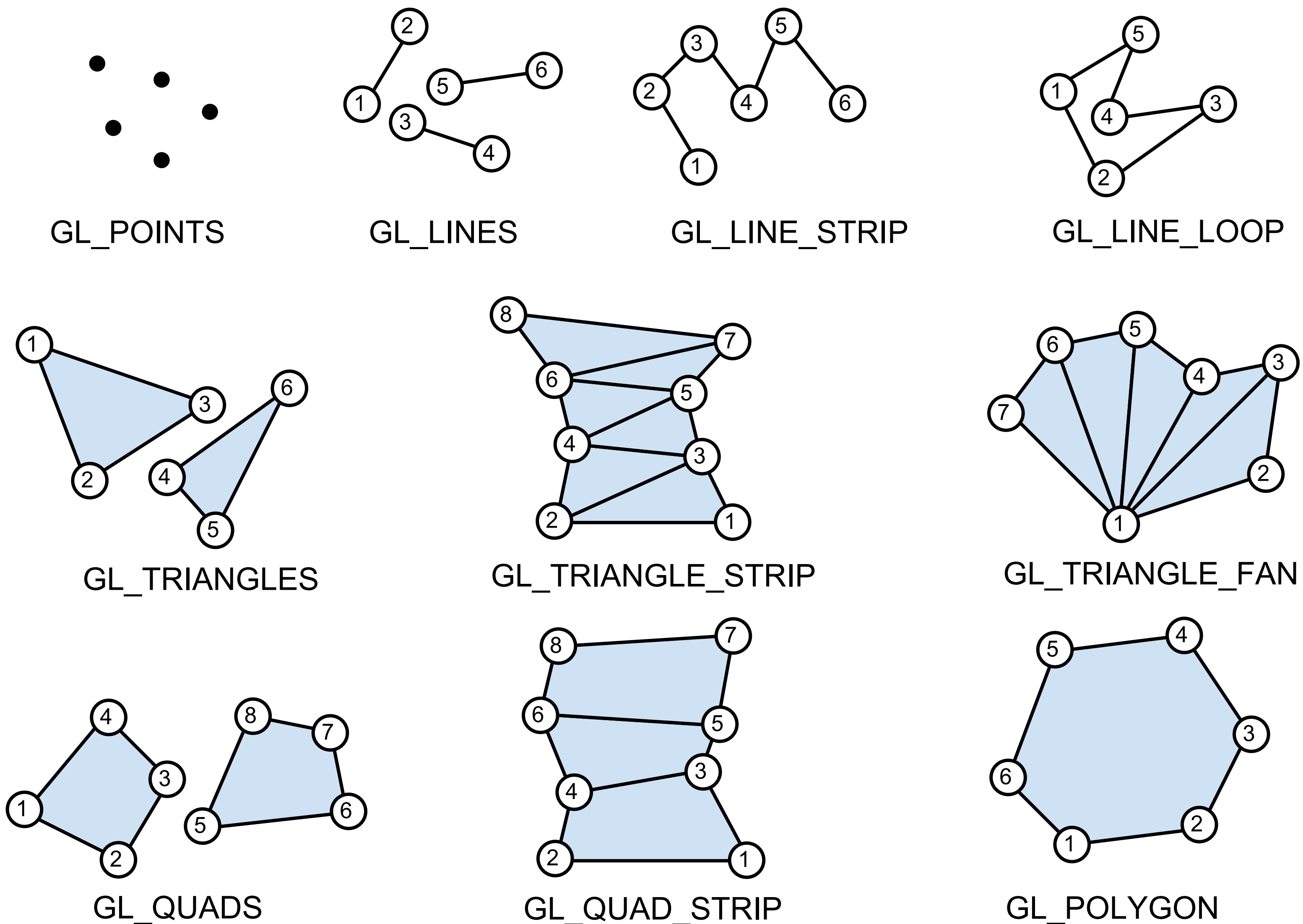
Vertex Buffer

0: (x, y, z)
1: (x, y, z)
2: (x, y, z)
3: (x, y, z)

Index Buffer

0, 1, 2
1, 3, 2

OpenGL Geometry Modes

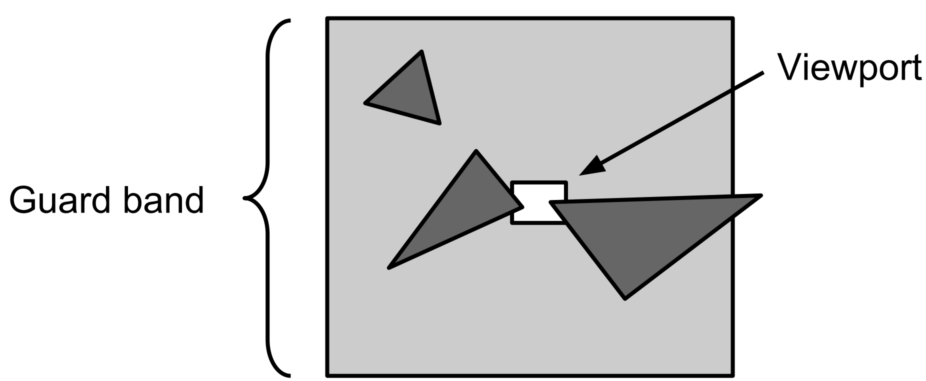


Stage 4: Clipping and Rasterization

- Clipping: Determine which triangles are visible
  - Clip triangles to view volume
  - Done in clip space where plane equations are simple (e.g. $-1 \leq x \leq 1$)
  - Cull back-facing polygons
- Does *not* include occlusion culling
  - Culling objects hidden behind other objects
  - Occlusion culling must be done by the application

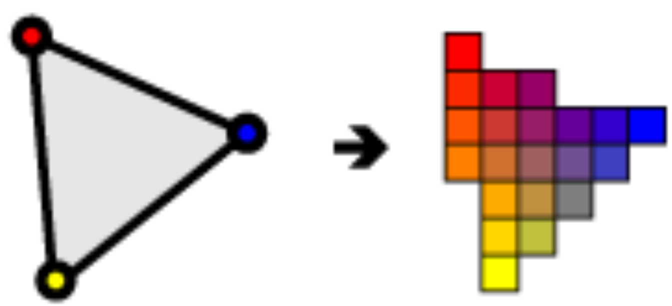
Clipping on the GPU

- Rasterization takes care of most clipping
  - We won't rasterize pixels off screen anyway
  - Can test for $z \leq 1$ to clip far plane
- Guard-band "clipping"
  - Really a method of avoiding clipping
  - Only need to clip when triangle lies out of integer range of rasterizer (e.g. $-32768 \leq x \leq 32767$), since interpolation breaks down



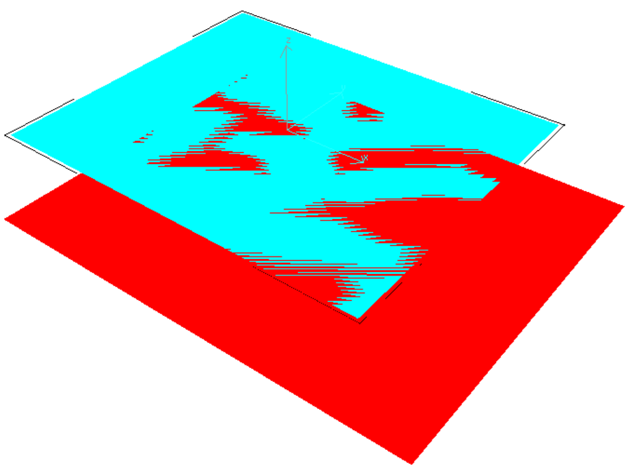
Rasterization on the GPU

- Turn triangles into pixels
  - Not done using scanlines on GPUs
- Usually done in tiles (maybe 8x8 pixels)
  - Tiles have better cache performance and shaders need to be run in 2x2 blocks, we'll see why later
  - Use coarse rasterizer first (maybe 32x32 pixels)
  - Very thin triangles are a worst case

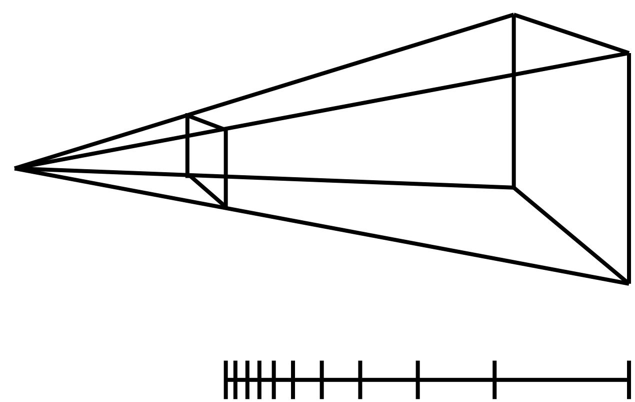


Z-Fighting

- Two roughly coplanar polygons map to the same z value
  - Due to limited z-buffer precision
  - Non-uniform precision due to perspective divide by w



Z-fighting artifacts



Non-uniform z-buffer precision

Fixing Z-Fighting in OpenGL

- Change the near clipping plane
  - Z-fighting error roughly proportional to 1 / near plane distance
  - Make near plane as far away as possible
  - Far plane also effects z-precision, but much less so
- Make vertices for both polygons the same
  - Generated fragments will have exact same depth value
  - Later one will draw over earlier one with the glDepthFunc(GL_LEQUAL) mode
  - Useful for applying decals with blending

Fixing Z-Fighting in OpenGL

- Polygon offsets
  - Adjust polygon depth values before depth testing
  - Also useful for drawing a wireframe on top of a mesh

```
// Draw polygon in front here (or wireframe)

// Move every polygon rendered backward a bit before depth testing
glPolygonOffset(1, 1);
glEnable(GL_POLYGON_OFFSET_FILL);

// Draw polygon in back here

glDisable(GL_POLYGON_OFFSET_FILL);
```

Stage 5: Fragment Shader

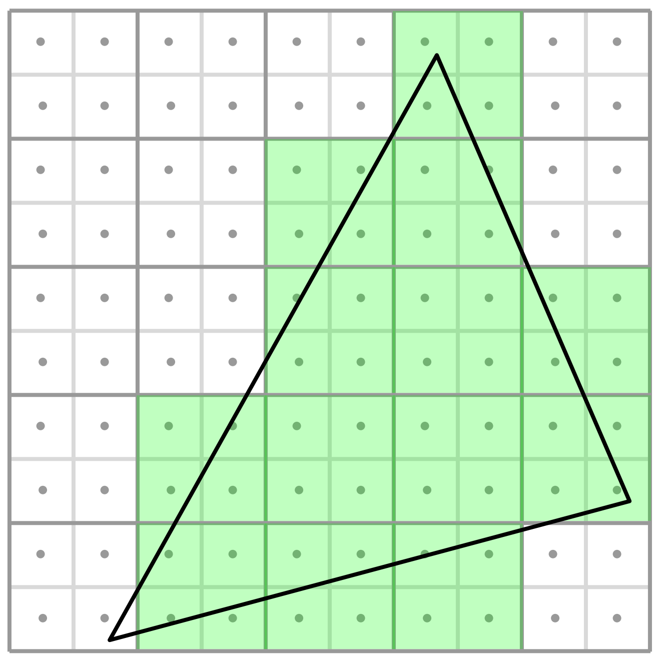
- Set the color for each fragment (each pixel)
- Inputs
  - Uniform values (lights, textures)
  - Varying values (normals, texture coordinates)
- Outputs
  - Fragment color: gl_FragColor (and optionally depth)
- Fragment shaders must run in parallel in 2x2 blocks
  - Need screen-space partial derivatives of texture coordinates for hardware texturing (mipmapping)
  - Compute finite differences with neighboring shaders
  - Can compute partial derivatives of any varying value this way

A Fragment Shader (GLSL)

```
// Input from the vertex shader,
// interpolated across the triangle
varying vec4 vertex;

void main() {
    // Compute the normal using the cross product of the x
    // and y screen-space derivatives of the vertex position
    vec3 pos = vertex.xyz;
    vec3 normal = normalize(cross(dFdx(pos), dFdy(pos)));

    // Visualize the normal by converting to RGB color
    gl_FragColor = vec4(normal * 0.5 + 0.5, 1.0);
}
```

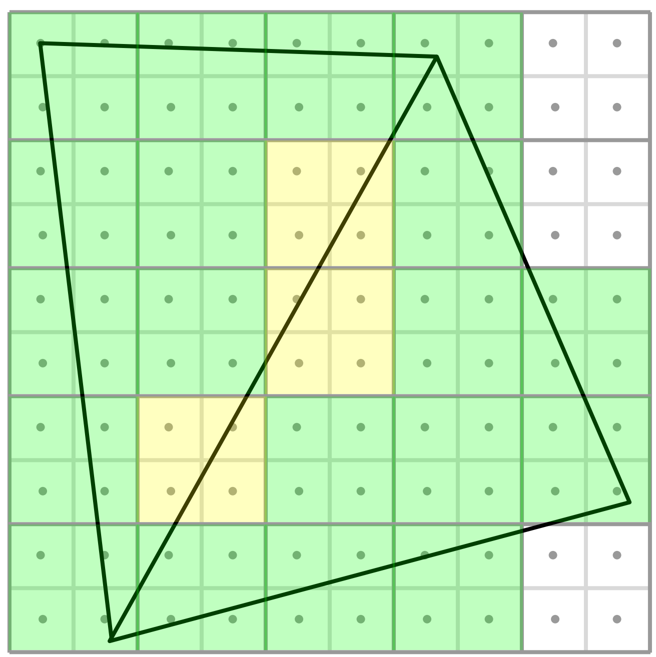
Overshading and Micropolygons

- Drawback to rasterization
  - Fragment shaders run in 2x2 blocks
  - Meshes are shaded more than once per pixel
  - Micropolygons are a worst case



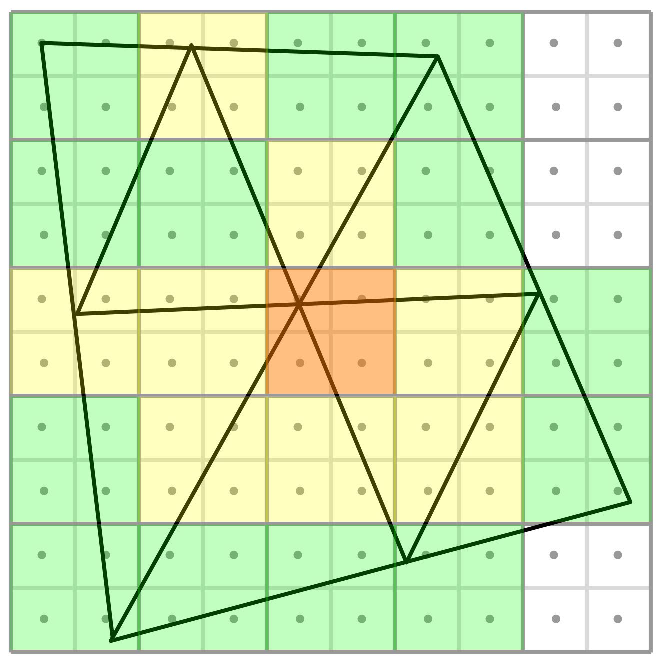
Overshading and Micropolygons

- Drawback to rasterization
  - Fragment shaders run in 2x2 blocks
  - Meshes are shaded more than once per pixel
  - Micropolygons are a worst case



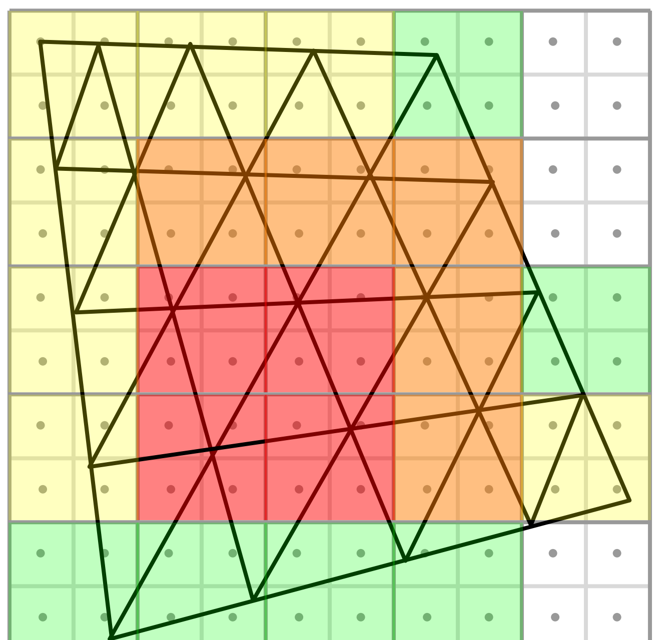
Overshading and Micropolygons

- Drawback to rasterization
  - Fragment shaders run in 2x2 blocks
  - Meshes are shaded more than once per pixel
  - Micropolygons are a worst case



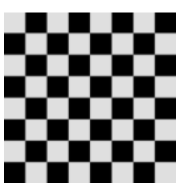
Overshading and Micropolygons

- Drawback to rasterization
  - Fragment shaders run in 2x2 blocks
  - Meshes are shaded more than once per pixel
  - Micropolygons are a worst case

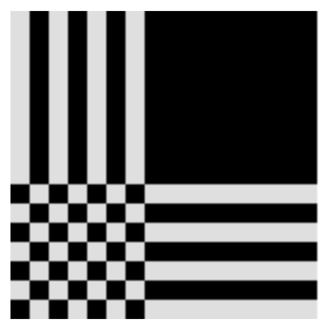


OpenGL Texture Mapping

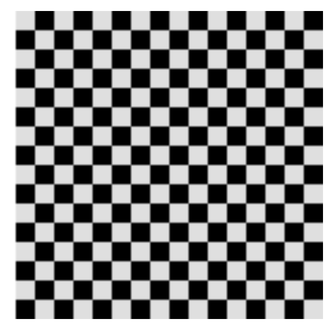
- Big arrays of data with hardware filtering support
  - Dimensions: 1D, 2D, 3D
  - Texture formats: GL_RGB, GL_RGBA, ...
  - Data types: GL_UNSIGNED_BYTE, GL_FLOAT, ...
- Texture wrap modes



Texture



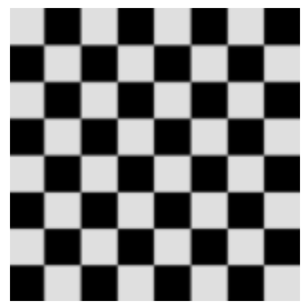
GL_CLAMP_TO_EDGE



GL_REPEAT

OpenGL Texture Filtering

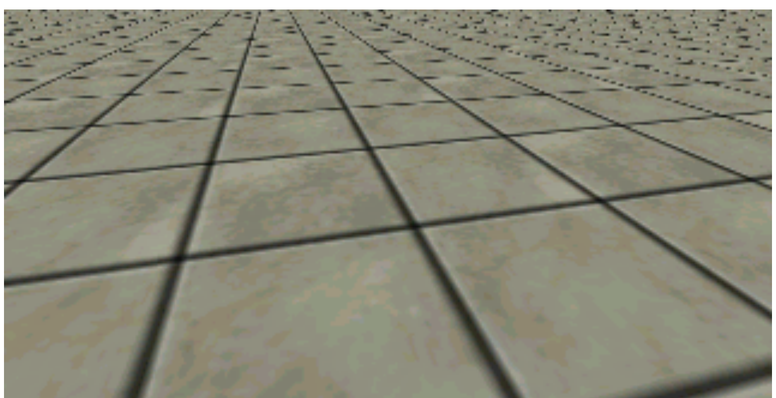
- Each texture has two filters
  - Magnification vs minification
- Nearest-neighbor: GL_NEAREST



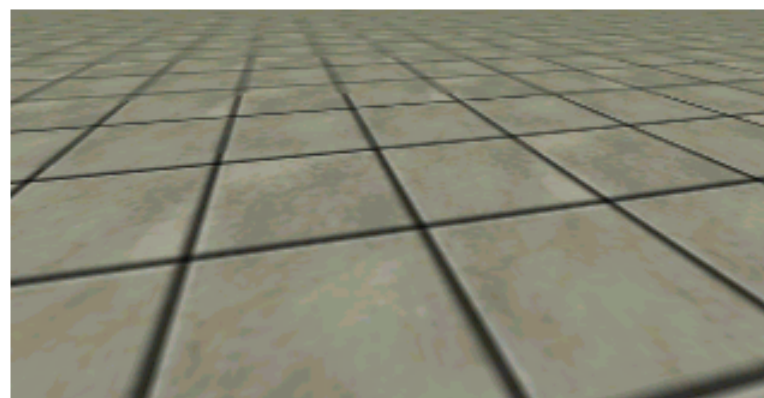
- Bilinear interpolation: GL_LINEAR

OpenGL Mipmapping

- Pre-filtered textures used to avoid aliasing
  - Uses smaller versions of textures on distant polygons
  - Screen-space partial derivative of texture coordinate used to compute mipmap level
  - Can also filter between two closest mipmap levels (trilinear filtering)



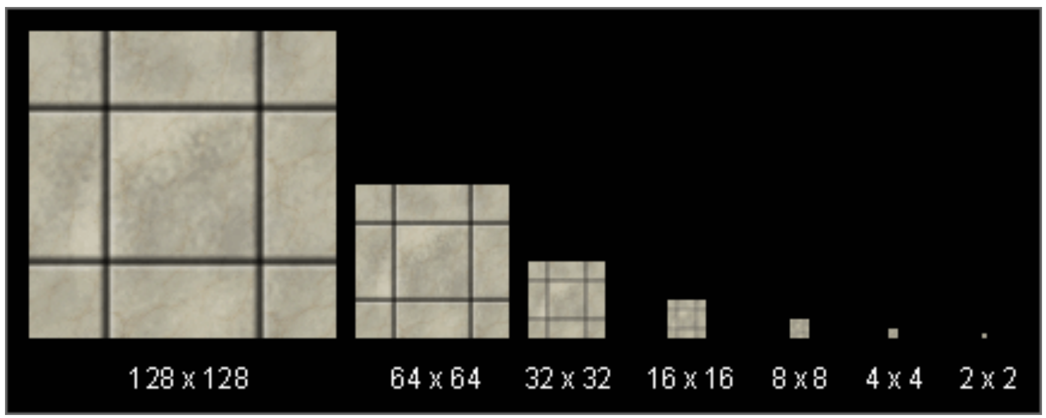
Without mipmapping



With mipmapping

OpenGL Mipmapping

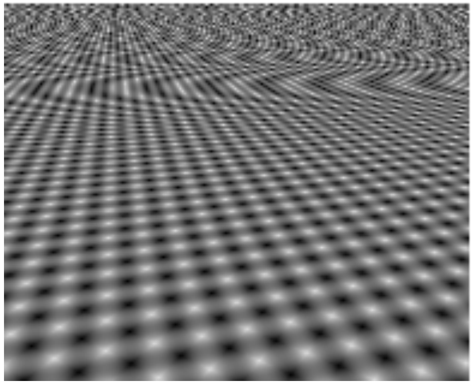
- Repeatedly halve image size until 1x1, store all levels



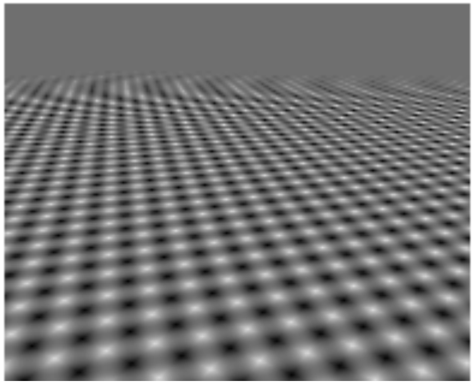
- Done automatically with gluBuild2DMipmaps()
    - Make sure to set the correct texture minification filter (i.e. GL_LINEAR_MIPMAP_LINEAR)

Anisotropic Filtering

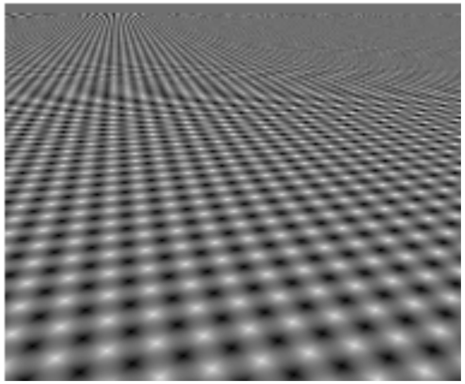
- Used in addition to mipmapping
- Improves distant textured polygons viewed from an angle



Bilinear filter



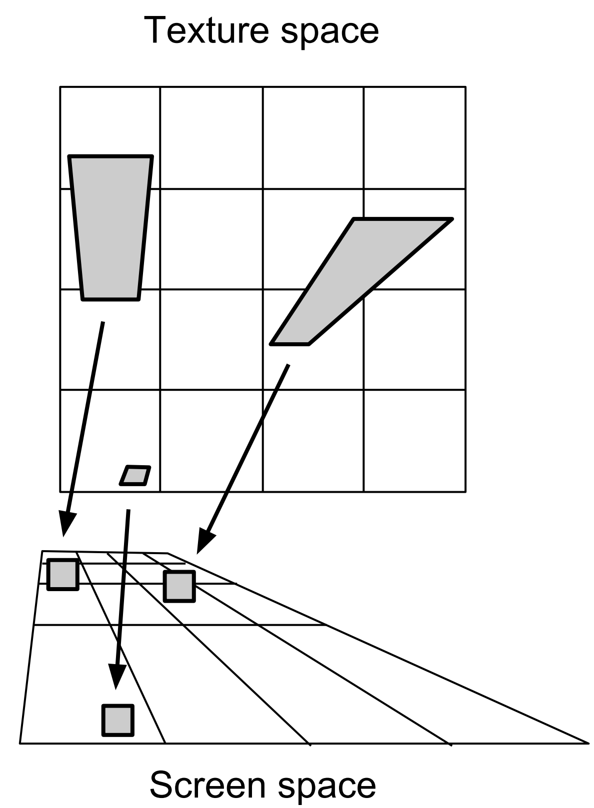
Mipmapping



Anisotropic filtering

Anisotropic Filtering

- Texture sampled multiple times in projected trapezoid
  - Sampling pattern is implementation dependent
- Each sample uses trilinear filtering
  - 16 anisotropic samples require 128 texture lookups!
- Supported in OpenGL through an extension:
  - glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_MAX_ANISTROPY_EXT, <float greater than 1>);



Stage 6: Depth Testing and Blending

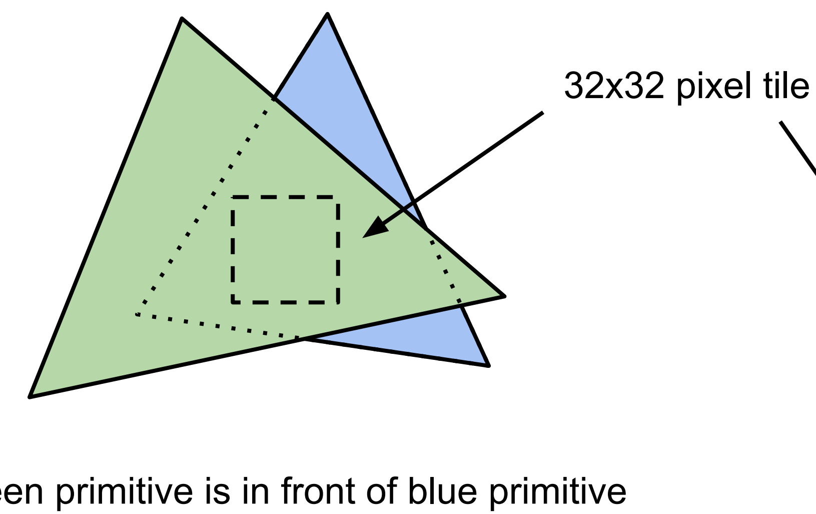
- Depth testing
  - Reject pixel if behind existing objects
  - Different modes: GL_LESS, GL_GREATER, ...
- Early-Z culling
  - Move depth testing before fragment shader
  - Avoids some fragment shader evaluations
  - Only used if fragment shader doesn't modify depth
  - Drawing front-to-back is much faster than drawing back-to-front in the presence of overdraw

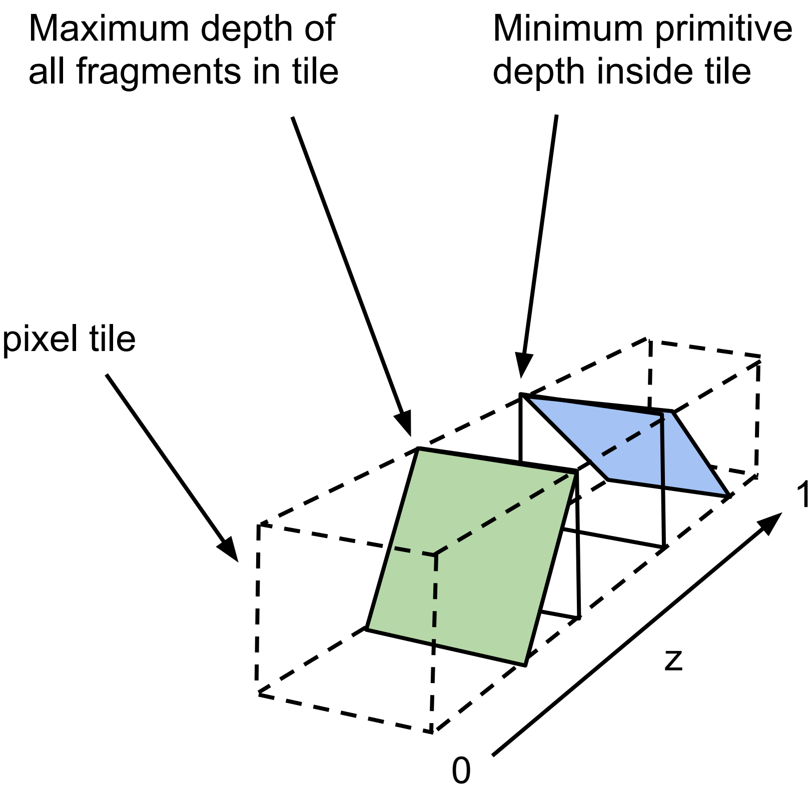
Hierarchical Z-Buffer Culling

- Hierarchical Z-buffer (maybe 32x32 tiles)
  - Early-z culling at tile resolution
  - Occurs before fine rasterization and early-z culling
  - Tiles store maximum depth of their fragments
  - Reject all fragments for a primitive within a tile if minimum fragment depth > maximum tile depth
- Changing depth testing mode mid-frame disables Hi-Z culling, so don't do that!

Hierarchical Z-Buffer Culling

- Blue primitive can be culled as completely occluded inside tile





Blending

- Blend source and destination RGBA colors
  - destination = current framebuffer color
  - source = color of fragment being blended
  - color = source • sfactor + destination • dfactor
  - Specify using glBlendFunc(sfactor, dfactor)
- Factors (each is a 4-vector):

GL_ZERO	GL_ONE
GL_SRC_ALPHA	GL_ONE_MINUS_SRC_ALPHA
GL_SRC_COLOR	GL_ONE_MINUS_SRC_COLOR
GL_DST_ALPHA	GL_ONE_MINUS_DST_ALPHA
GL_DST_COLOR	GL_ONE_MINUS_DST_COLOR

Blending

- Example: alpha blending (linear interpolation)
  - sfactor = GL_SRC_ALPHA
  - dfactor = GL_ONE_MINUS_SRC_ALPHA
- Alpha blending numerical example
  - source = (1, 1, 0, 0.2)
  - destination = (0, 1, 0.6, 0.5)
  - color = source • (0.2, 0.2, 0.2, 0.2) + destination • (0.8, 0.8, 0.8, 0.8)
  - color = (0.2, 0.2, 0, 0.04) + (0, 0.8, 0.48, 0.4)
  - color = (0.2, 1, 0.48, 0.44)
- Colors clamp to [0, 1] range

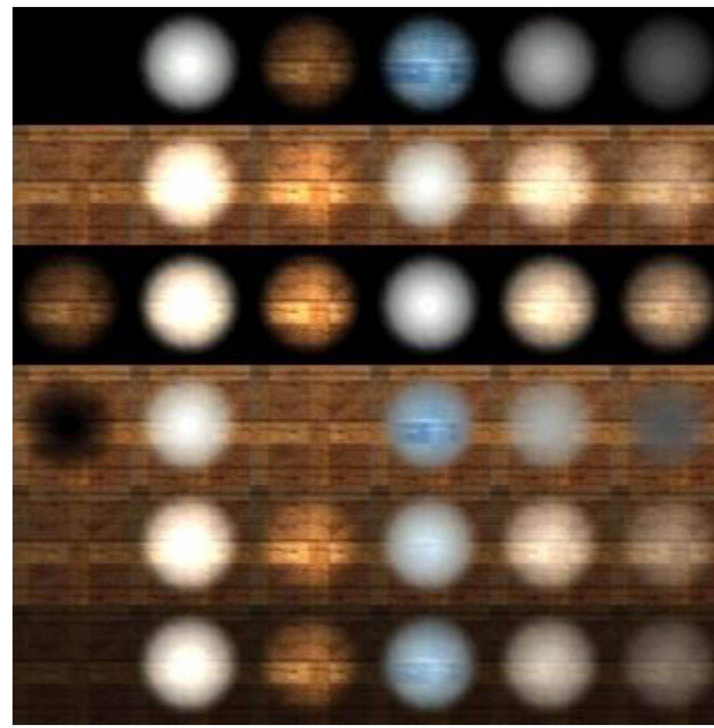
Blending

Destination image (current framebuffer contents)

Source image (fragments being blended in front)

GL_ZERO
GL_ONE
GL_DST_COLOR
GL_ONE_MINUS_DST_COLOR
GL_DST_ALPHA
GL_ONE_MINUS_DST_ALPHA

GL_ZERO
GL_ONE
GL_SRC_COLOR
GL_ONE_MINUS_SRC_COLOR
GL_SRC_ALPHA
GL_ONE_MINUS_SRC_ALPHA



Blending

- Non-programmable for a reason
  - Fragments need to be blended together in order
  - Serial task, cannot be parallelized
  - Custom code would be unacceptably slow
- Sensible combinations for games:
  - *glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);*

    Regular alpha blending, order dependent
  - *glBlendFunc(GL_ONE, GL_ONE);*

    Additive blending, order independent
  - *glBlendFunc(GL_ONE_MINUS_DST_COLOR, GL_ONE);*

    Additive blending with saturation, also order independent but looks much better than additive!

Stage 7: The Framebuffer

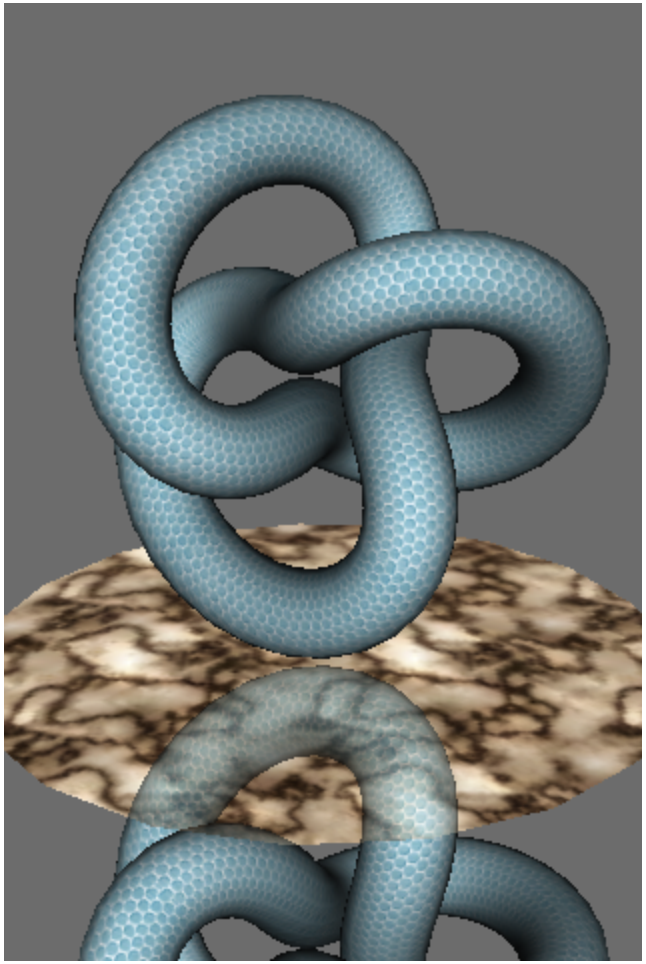
- A collection of 2D arrays used by OpenGL
- Color buffer
  - Stores RGBA pixel values (32 bits)
- Depth buffer
  - Stores a single integer depth value (commonly 24 bits)
- Stencil buffer
  - Linked with depth buffer
  - Stores a bitmask that can be used to limit visibility (commonly 8 bits)
- Accumulation buffer
  - Used to store intermediate results
  - Outdated and historically slow

The Stencil Buffer

- Per-pixel test, similar to depth buffer
  - Test each fragment against value from stencil buffer, reject fragment if stencil test fails
  - Distinct cases allow for different behavior when
    - Stencil test fails
    - Stencil test passes but depth test fails
    - Stencil and depth tests pass
- OpenGL stencil functions:
  - glEnable(GL_STENCIL_TEST)
  - glStencilFunc(function, reference, bitmask)
  - glStencilOp(stencil_fail, depth_fail, depth_pass)
  - glStencilMask(bitmask)

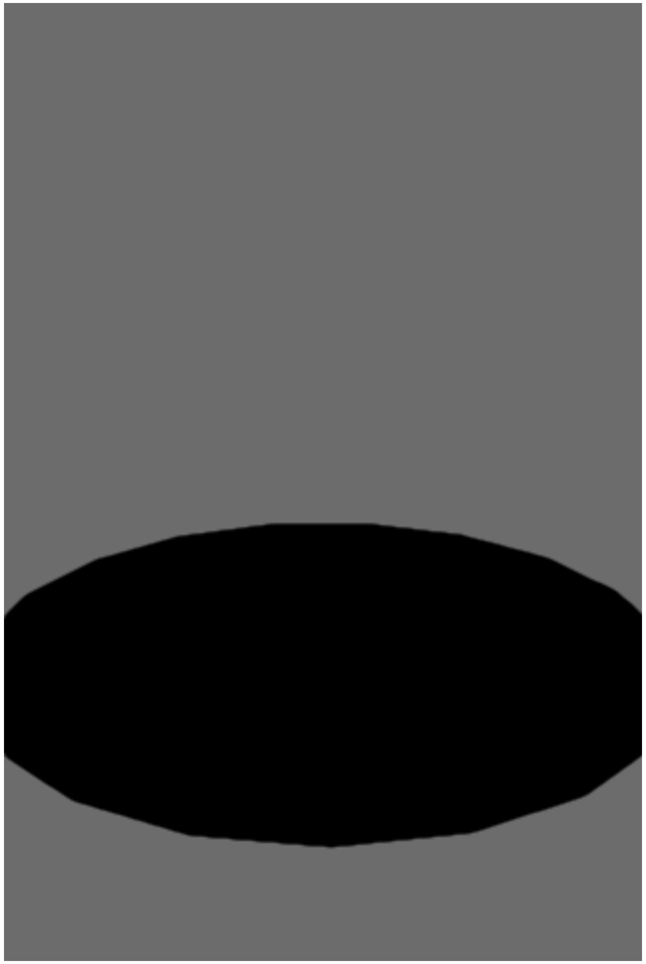
The Stencil Buffer

- Example: Reflections
  - glScalef(1, -1, 1) is close
  - Need to restrict rendering to inside reflective polygon



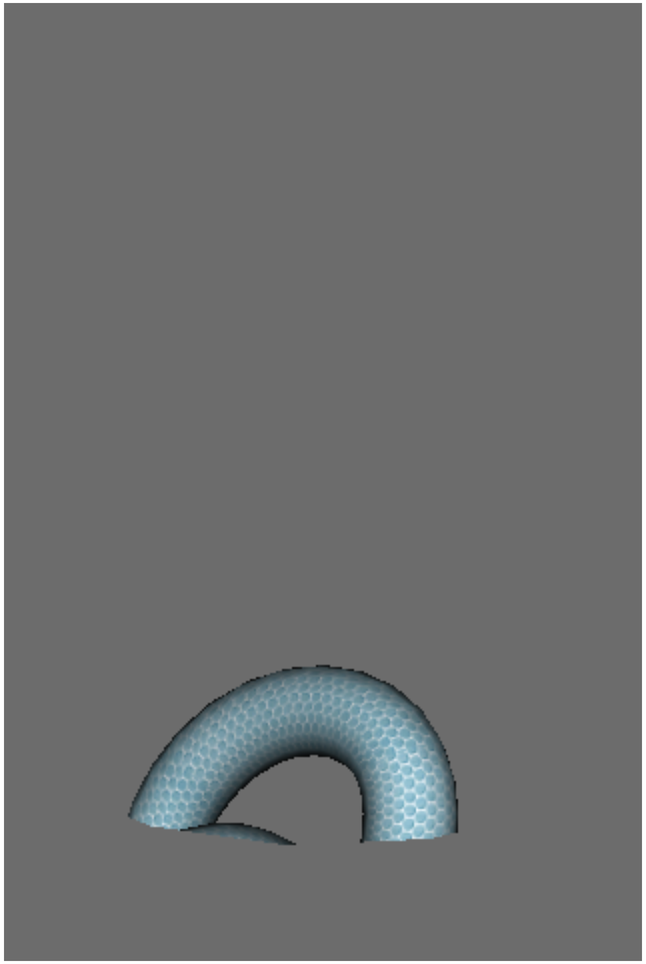
The Stencil Buffer

- Example: Reflections
  - First render reflective polygon
  - Set stencil buffer to 1 wherever polygon is
  - Don't render anything to the color or depth buffers



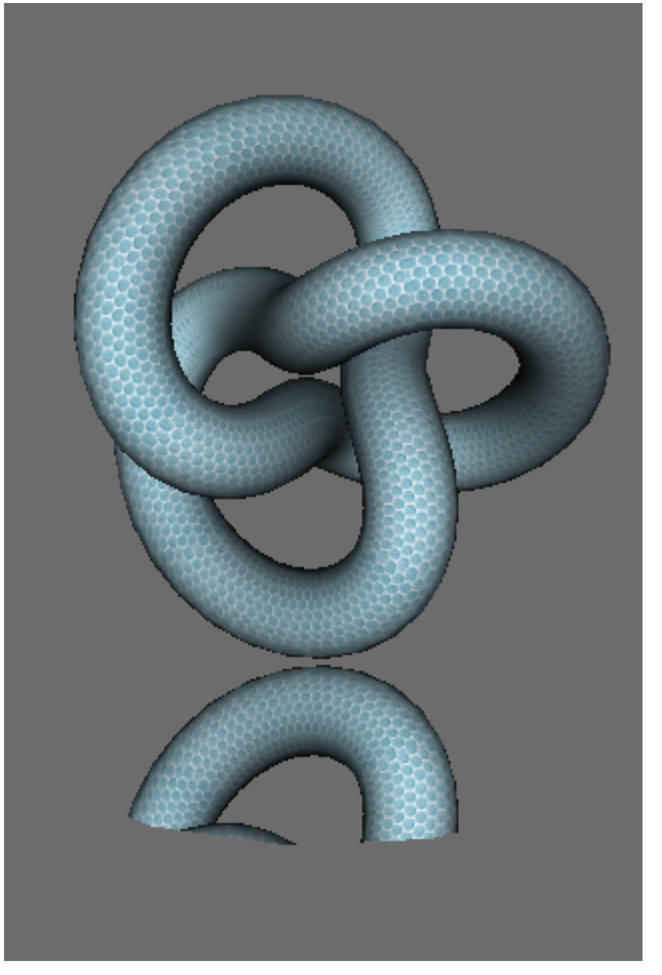
The Stencil Buffer

- Example: Reflections
  - Draw reflected object next
  - Restrict drawing to where stencil buffer is 1



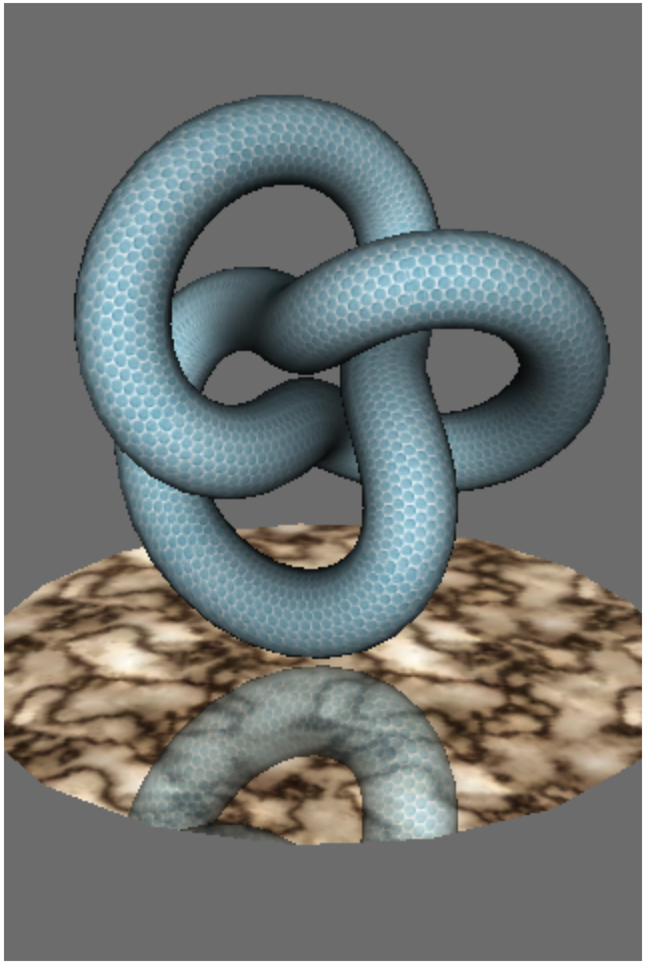
The Stencil Buffer

- Example: Reflections
  - Draw object normally
  - Ignore stencil buffer values



The Stencil Buffer

- Example: Reflections
    - Draw the reflected polygon again, this time with blending
    - Use alpha to control surface reflectivity

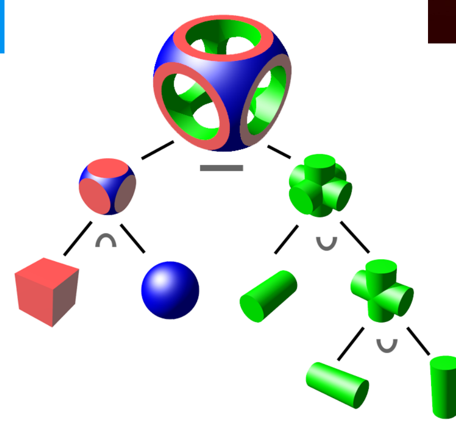


The Stencil Buffer

- Advanced rendering tricks when paired with depth buffer:

Stenciled shadow volumes

Mirrors and portals



CSG rendering

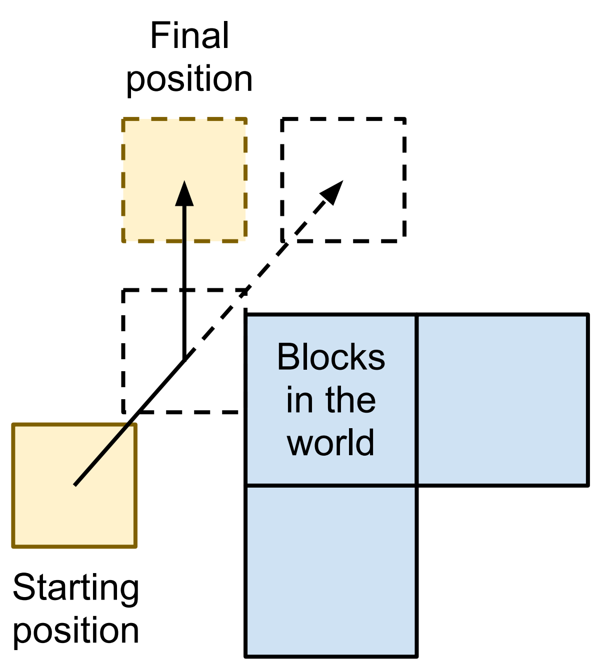
Minecraft Week 2

Minecraft: Week 2

- First-person movement
  - Similar to Warm-up
  - Collision detection and response
- Speed increases
  - VBOs for chunk rendering
  - Per-chunk frustum culling

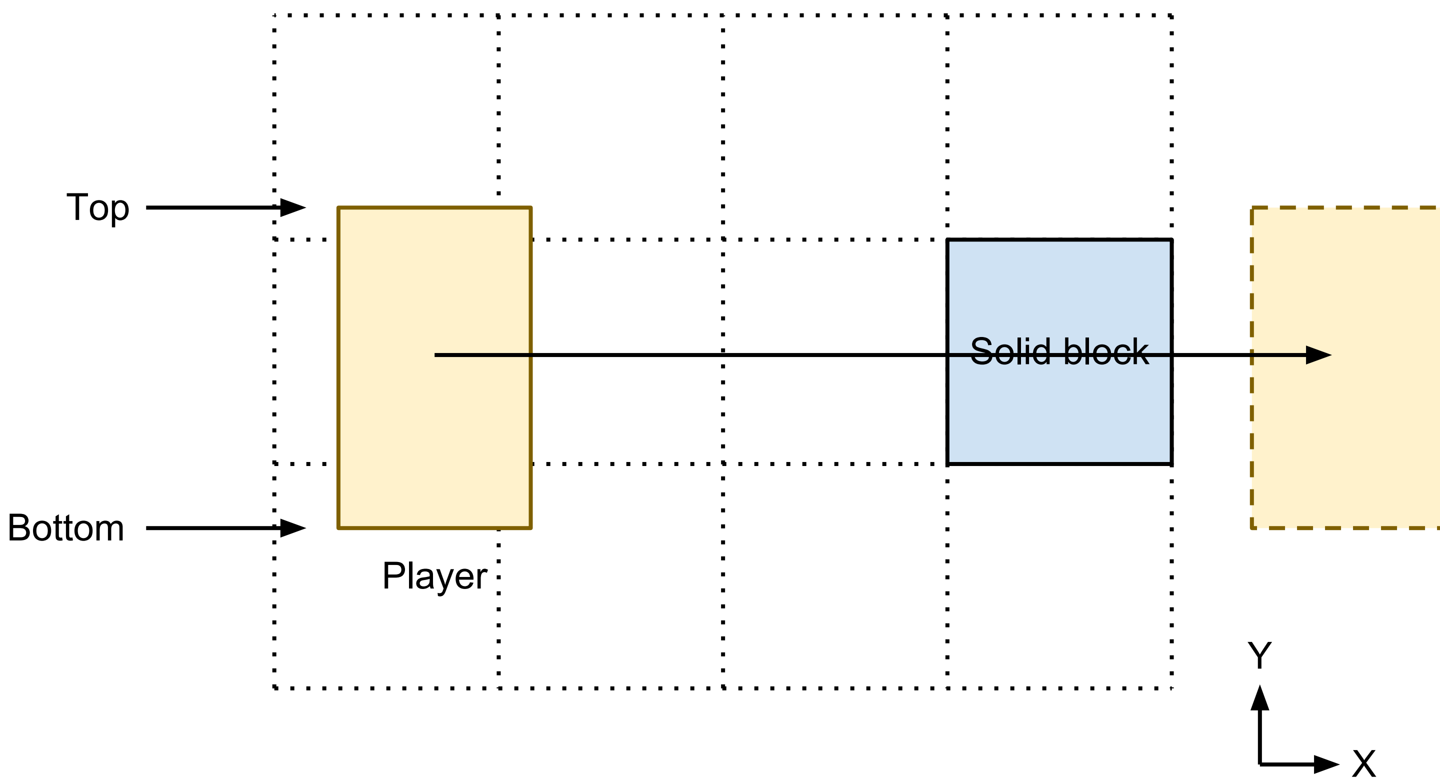
Collision Detection and Response

- Move along x, y, and z axes separately
- Stop before AABB around player moves into a solid block
- Player will automatically slide along surfaces



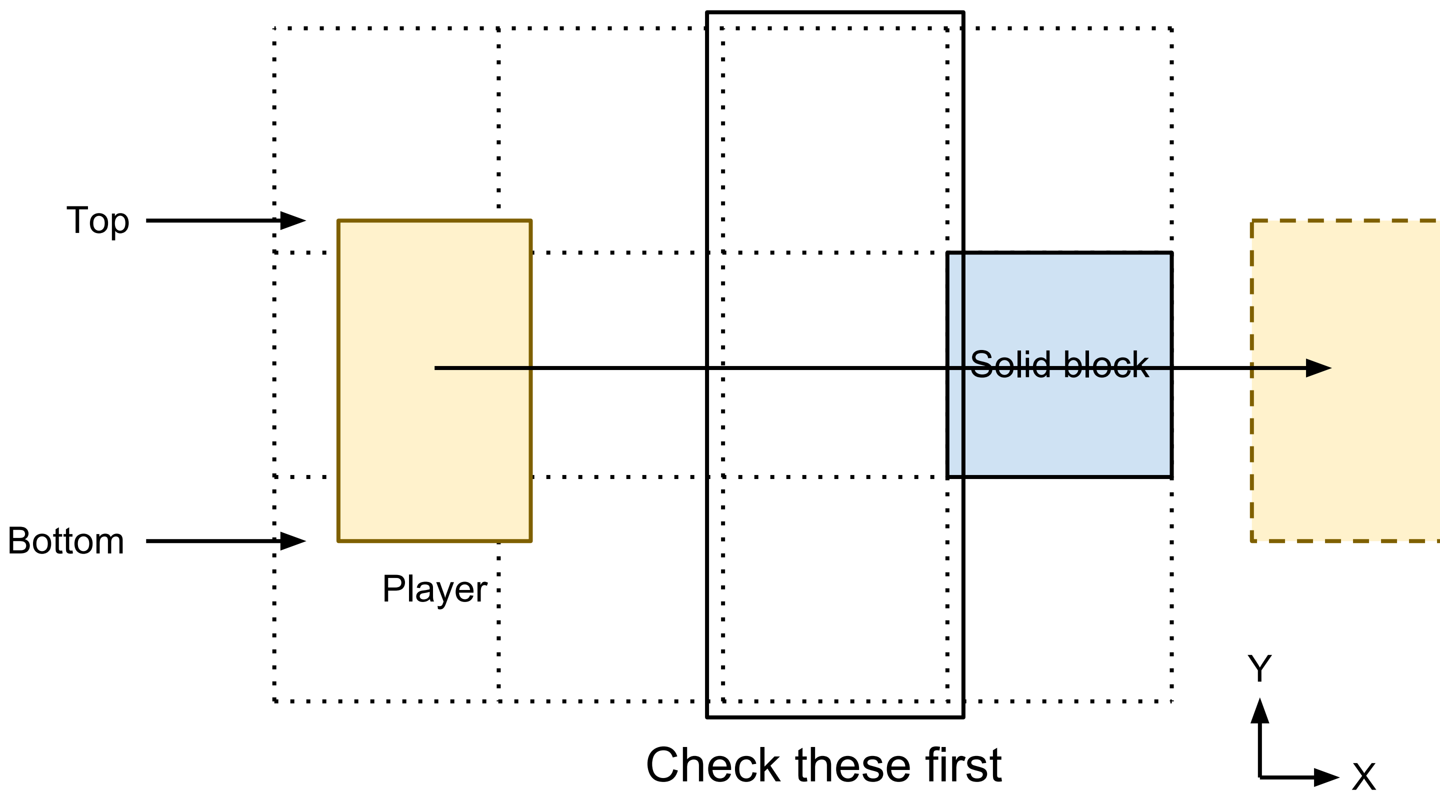
Example: Collision Sweep Along X-Axis

- Search cells from floor(player.bottom) to ceil(player.top)
- Player is moving in increasing x, check cells in that order



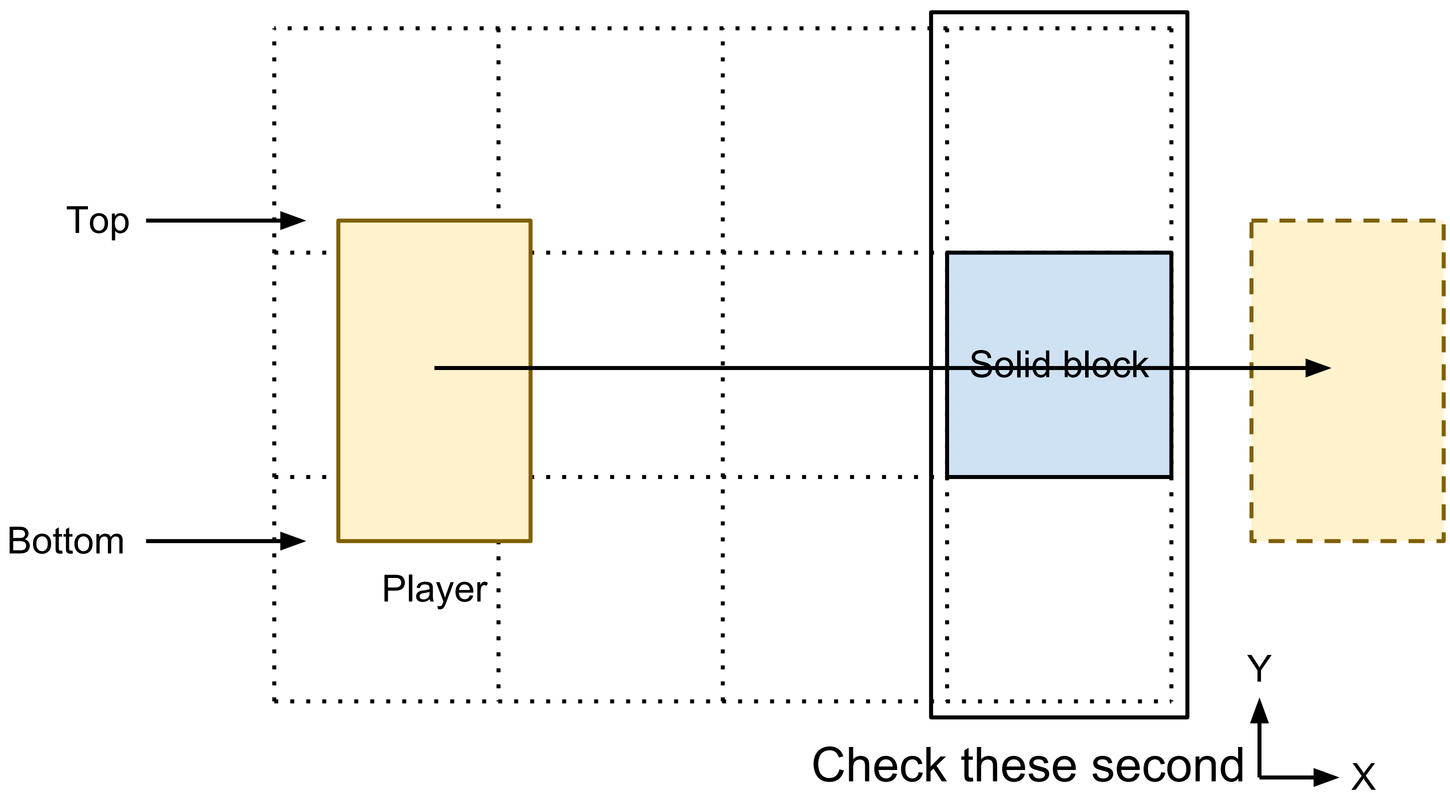
Example: Collision Sweep Along X-Axis

- Search cells from floor(player.bottom) to ceil(player.top)
- Player is moving in increasing x, check cells in that order



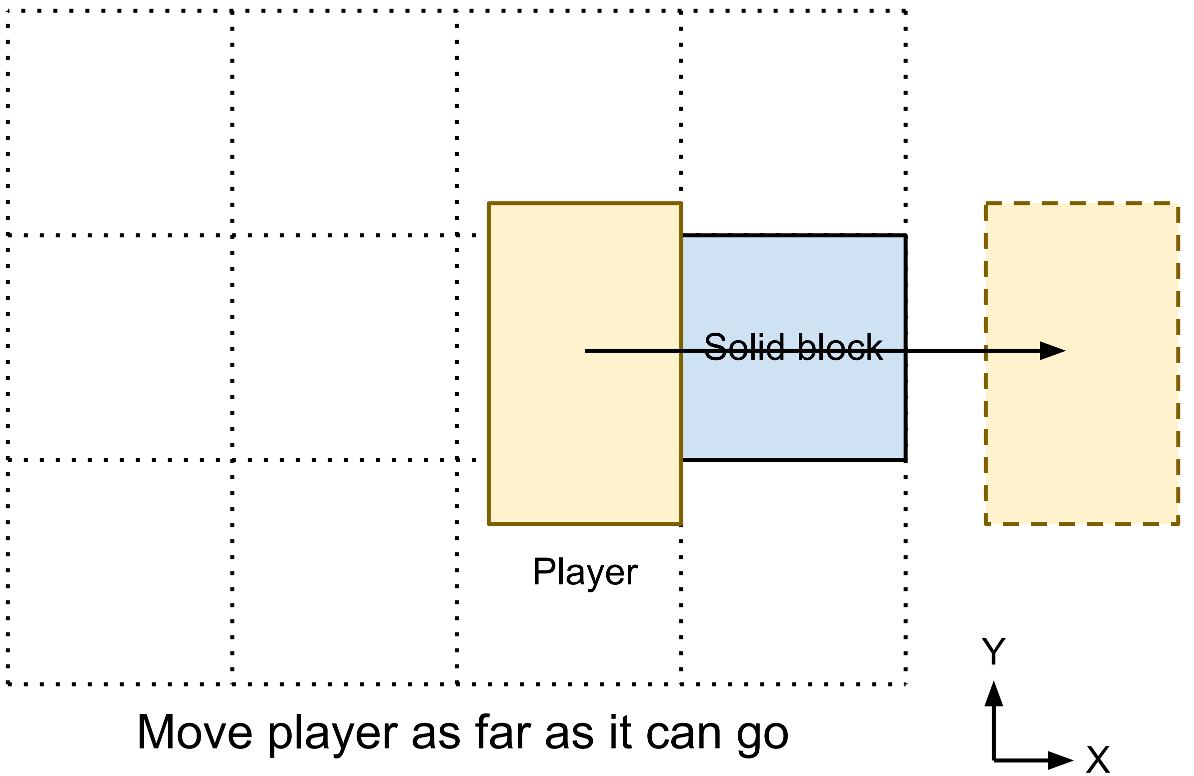
Example: Collision Sweep Along X-Axis

- Search cells from floor(player.bottom) to ceil(player.top)
- Player is moving in increasing x, check cells in that order



Example: Collision Sweep Along X-Axis

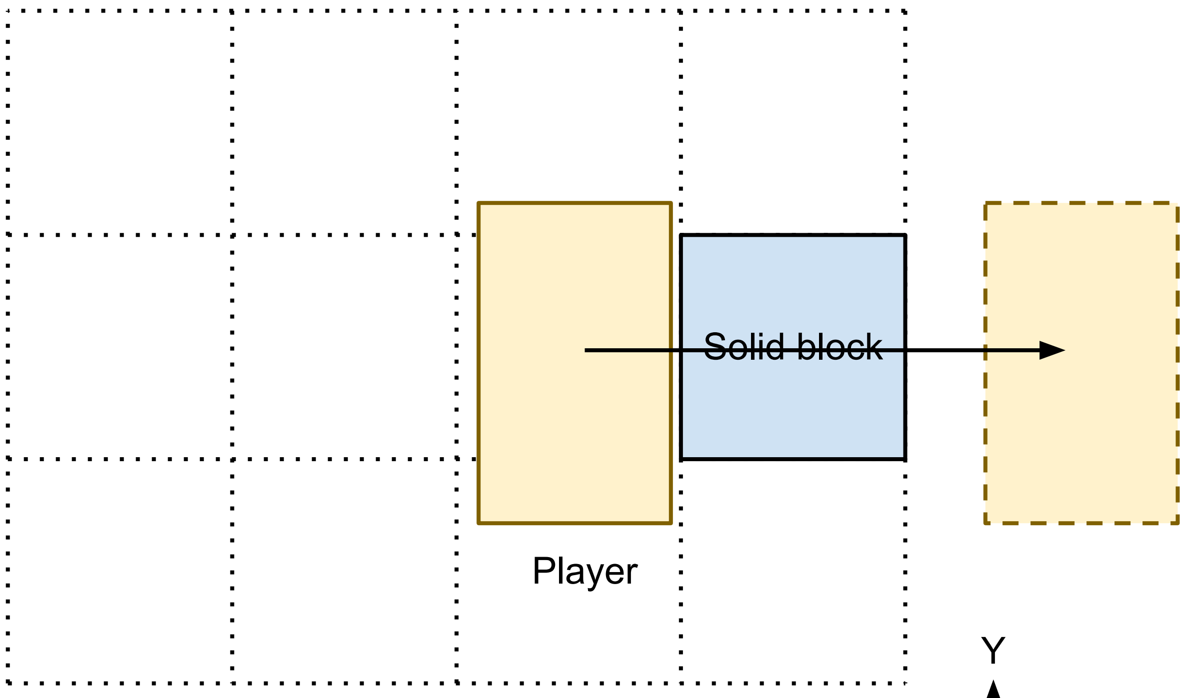
- Search cells from floor(player.bottom) to ceil(player.top)
- Player is moving in increasing x, check cells in that order



Move player as far as it can go

Example: Collision Sweep Along X-Axis

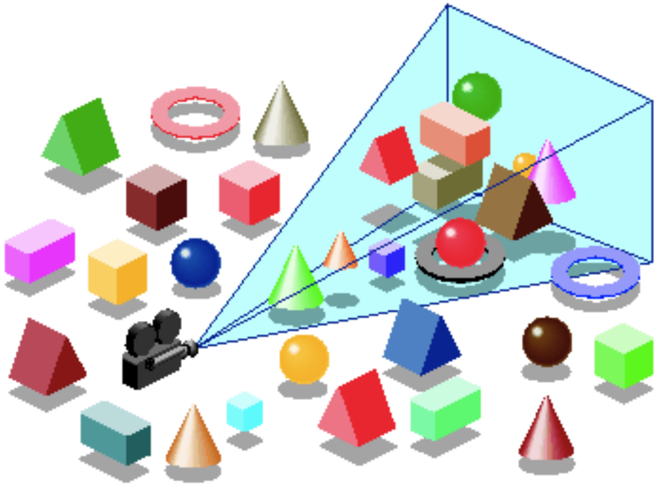
- Search cells from floor(player.bottom) to ceil(player.top)
- Player is moving in increasing x, check cells in that order



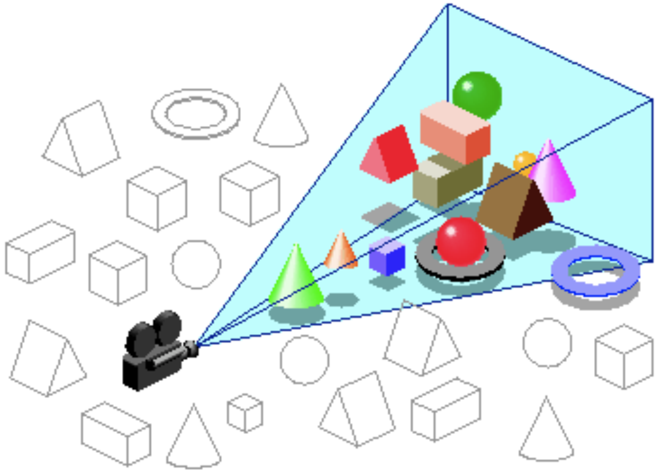
Move player back by epsilon (0.00001)

View Frustum Culling

- Optimization: only draw what the camera can see
  - GPU doesn't waste time drawing objects behind you
  - Camera will never see anything outside the view volume



vs



Extracting the View Frustum

- Frustum defined by 6 planes
- Matrix to the left is the product of projection and modelview matrices, numbered by OpenGL element order
- Extract matrices with glGetFloatv
- Plane equation is given by the four-vector (a, b, c, d) where ax + by + cz + d = 0

$$\begin{array}{cc} \begin{bmatrix} 0 & 4 & 8 & 12 \\ 1 & 5 & 9 & 13 \\ 2 & 6 & 10 & 14 \\ 3 & 7 & 11 & 15 \end{bmatrix} & \begin{array}{l} \leftarrow r0 \\ \leftarrow r1 \\ \leftarrow r2 \\ \leftarrow r3 \end{array} \end{array}$$

Projection matrix • modelview matrix

Clipping plane	−x	−y	−z	+x	+y	+z
Plane equation	r3 − r0	r3 − r1	r3 − r2	r3 + r0	r3 + r1	r3 + r2

Frustum Culling Tests

- Axis-Aligned Bounding Box (AABB) test
  - For each view frustum plane, reject if all 8 corners are behind that plane
    - For point (x, y, z), reject if $ax + by + cz + d < 0$
- Sphere test
  - Reject if it's behind any one plane
    - If $ax + by + cz + d < -\text{radius}$
  - Needs normalized planes
    - Divide (a, b, c, d) by $\text{sqrt}(a^2 + b^2 + c^2)$

C++ Tip of the Week

- Compile-time asserts
  - Useful for avoiding confusing run-time errors
  - Failed assertions mean the program doesn't compile
  - Plain C++ doesn't have built-in support
    - C++11 supports this through static_assert()
- Check that VBO formats are tightly packed

```
struct Vertex {
   float position[3];
   float color[3];
};

STATIC_ASSERT(sizeof(Vertex) == sizeof(float) * 6);
```

C++ Tip of the Week

- Must be added using hacks
  - Can sometimes lead to confusing error messages
  - Common methods for causing compiler errors:
    - Duplicate case statement (below)
    - Declare an array of negative size
    - Template specialization failure

```
// This has the advantage that it doesn't create any new
// local variables or typedefs, but must be placed inside
// a method body (and cannot be at global scope)
#define STATIC_ASSERT(condition) \
    switch(0){case 0:case condition:;}
```

Good Luck!

- Further reading
  - http://fgiesen.wordpress.com/2011/07/09/a-trip-through-the-graphics-pipeline-2011-index/ (highly recommended for advanced insights)
  - http://duriansoftware.com/joe/An-intro-to-modern-OpenGL.-Chapter-1:-The-Graphics-Pipeline.html (good walk-through introduction)

Weeklies!