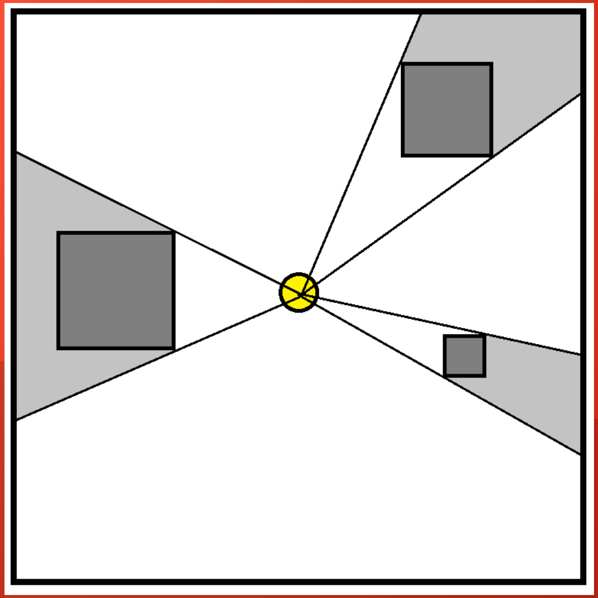
Omnidirectional Shadows, Cube Maps and the Geometry Shader

Omnidirectional Shadow Maps

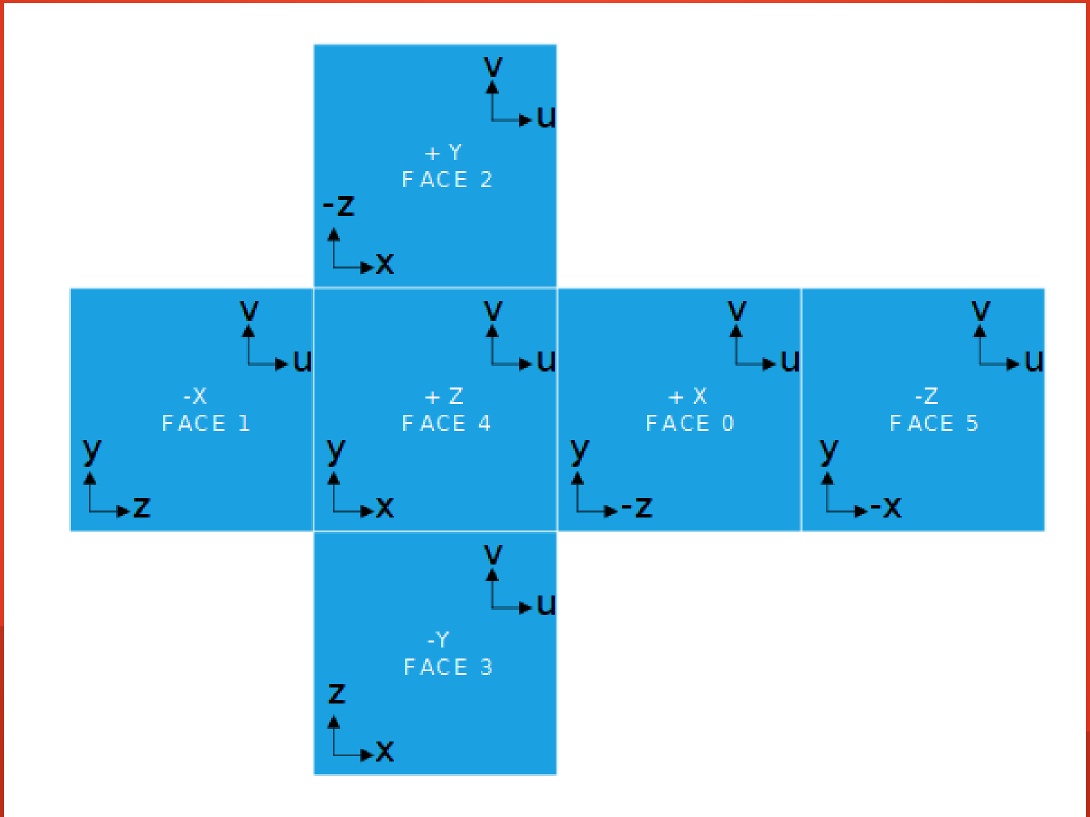
- Used for Point Lights and Spot Lights.
- Basic theory same as regular Shadow Maps…
- But need to handle shadows in EVERY direction!
- Can’t just use a single texture.
- Need MULTIPLE textures to handle every direction.
- Solution: A Cubemap.



Cubemaps

- Type of texture in OpenGL.
- Technically exists as 6 textures (one for each face) but can be referenced in GLSL as if only a single texture.
- glBindTexture(GL_TEXTURE_CUBE_MAP, depthCubemap);
- glTexImage2D(GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, ...)
- GL_TEXTURE_CUBE_MAP_POSITIVE_X, GL_TEXTURE_CUBE_MAP_NEGATIVE_X, GL_TEXTURE_CUBE_MAP_POSITIVE_Y, GL_TEXTURE_CUBE_MAP_NEGATIVE_Y, GL_TEXTURE_CUBE_MAP_POSITIVE_Z, GL_TEXTURE_CUBE_MAP_NEGATIVE_Z
- Each enum is one increment of the previous, so can loop through with incrementing value.

Cubemaps



Cubemaps

- Don’t actually need to use (u, v) values.
- Can access point on cube map with direction vector pointing to texel on cube map, from center of cube.
- This means we don’t need a light transform matrix for each point light!
- However we need 6 versions of the “projection x view” matrix, one for each of the 6 directions on the cube, for shadow map pass.

Omnidirectional Shadow Maps

- Using Perspective Projection.
- glm::perspective(glm::radians(90.0f), aspect, near, far);
- 90 degree perspective ensures all 360 degrees around one axis can be covered.
- Aspect is the width of one side of the cube divided by its height. This SHOULD be “1” for it to work properly! All cube dimensions are EQUAL!
- Near and far decide size of the cube (how far light reaches).

Omnidirectional Shadow Maps

- Create 6 light transforms with projection matrix, one for each direction.
- projection * upViewMatrix,
  projection * downViewMatrix,
  etc…
- Create view matrices using light position and direction.
- E.g.
  glm::lookAt(lightPos, lightPos + glm::vec3(1.0, 0.0, 0.0), glm::vec3(0.0,-1.0, 0.0));
- Direction is “lightPos + glm::vec3(1.0, 0.0, 0.0)” because it points to the ‘right’, in other words: In positive x direction.
- IMPORTANT: These matrices must line up with the Cubemap texture order (POSITIVE_X, NEGATIVE_X, POSITIVE_Y, etc).

Geometry Shader

- Vertex Shader only needs to do World Space transformation (i.e. multiply vertex by model matrix).
- Proection and View will be applied in the Geometry Shader.
- Geometry Shader is another shader type that happens between Vertex and Fragment shaders.
- Geometry Shader handles primitives (points, lines, triangles, etc).

Geometry Shader

- Vertex Shader handles individual vertices…
- Geometry Shader handles groups of vertices and can manipulate entire primtives.
- Can also create entirely NEW primitives.
- Don’t explicitly specify output variable.
- Instead, use “EmitVertex()” and “EndPrimitive()”.
- EmitVertex(): Creates vertex at location stored in gl_Position.
- EndPrimitive(): Stores primitive created by last EmitVertex() calls, and starts a new primitive.

Geometry Shader

```
#version 330

layout(triangles) in;
layout (triangle_strip, max_vertices=3) out;

void main()
{
    for(int i = 0; i < gl_in.length(); i++)
    {
        gl_Position = gl_in[i].gl_Position;

        EmitVertex();
    }
    EndPrimitive();
}
```

- “layout (triangles) in” specifies incoming primitive type is a triangle.
- Output is essentially the same but also specifies the number of vertices expected with it. If you output more than the max, they won’t be rendered.
- gl_in stores data for each vertex passed from Vertex Shader.

Geometry Shader

- One other value: gl_Layer
- Since we attach a cubemap to the Framebuffer, the Framebuffer has multiple layers: One for each output texture in the cubemap.
- Set the value of gl_Layer to determine which one to write to when calling EmitVertex.

Geometry Shader

- Why use Geometry Shader?
- Using the 6 transformation matrices and reassigning gl_Layer for each face…
- We can render each object 6 times for each of the directions of the light source…
- All in one render pass!
- Alternative: Do 6 shadow render passes and switch out the light transform matrix each time.

Using the Shadow Cubemap

- GLSL has type “samplerCube”.
- Bind Cubemap to this…
- When using texture, instead of uv co-ordinates, supply a direction vector.
- Use direction of light source to fragment being checked, no need for a light transform matrix!
- Using far plane, convert depth value to actual value:

```
float closest = texture(depthMap, fragToLight).r;
closest *= far_plane;
```

- Then compare this value to the length of fragToLight (distance from fragment to light source), and use that to determine if in shadow!

Multiple Point Lights – Common Mistake

- In theory it’s easy: One samplerCube for each Point Light.
- Shadow Map pass is done as previously stated.
- Render pass is also done as previously stated, however…
- By default, samplers are mapped to Texture Unit 0.
- If you already have a sampler2D mapped to Unit 0…
- And you have an array of unused Point Lights…
- Their samplerCubes will remain as default, i.e. Texture Unit 0!
- OpenGL forbids different types of sampler to be bound to the same Texture Unit.
- Solution: Find a way to ensure all sampler types have unique Texture Units.

Omnidirectional Shadow Maps - PCF

- Essentially the same concept, but with a third dimension (vector has 3 values).
- Could just do as before, but with third dimension it becomes intensive…
- And a lot of the samples will be very close to the original!
- One solution: Pre-defined offset directions that are likely to be well spaced.
- In coding lesson, we create 20 offset directions and use those.

Omnidirectional Shadow Maps - PCF

- Another optimisation: The pre-defined offsets are DIRECTIONS, not relative positions.
- We can scale how far we sample in a direction.
- So you can scale how far sample is, based on viewer distance!
- If user is close: Sample more close to original vector.
- If user is distant: Sample more distant from original vector.
- Essentially creating our own filter.

Summary

- Omnidirectional Shadows use Cubemaps to map shadows in all directions.
- Cubemaps are texture consisting of 6 sub-textures.
- Cubemap texels are referenced by a direction vector.
- Geometry Shader handles primitives.
- Geometry Shader can modify primitives and create entirely new ones… we use it to map to the cubemap from 6 views, allowing only a single shadow pass per light.
- Due to nature of cubemaps, no need for light transform matrix in render pass.
- PCF can use predefined offset directions.
- PCF can scale offsets based on viewer distance.
- Need to ensure samplerCubes aren’t bound to same texture unit as sampler2D!

See you next video!