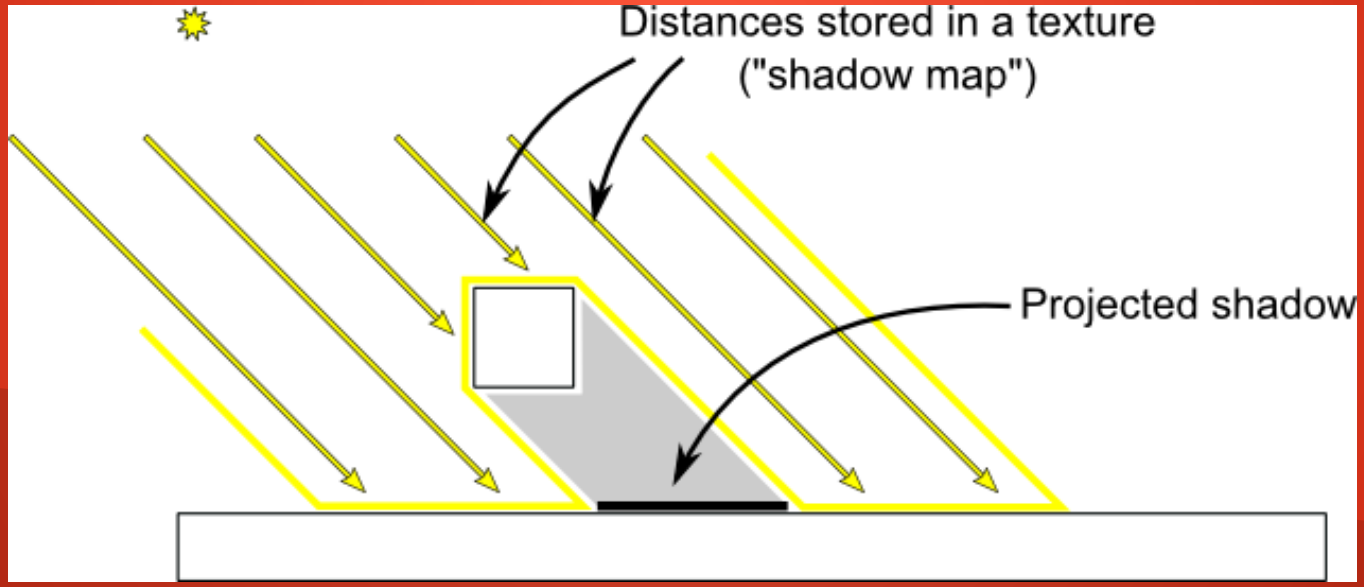
Shadow Mapping

Shadow Mapping

- Literally create a “map” of the shadows made by a light.
- Use this map to determine where not to apply light.
- The map is held as a 2D texture (sampler2D in shader).
- Map is created using a “Framebuffer”.
- Framebuffer then writes to texture.
- Therefore: At least two rendering passes needed!
- One for creating shadow map, second for drawing scene.

Shadow Mapping – Creating Map

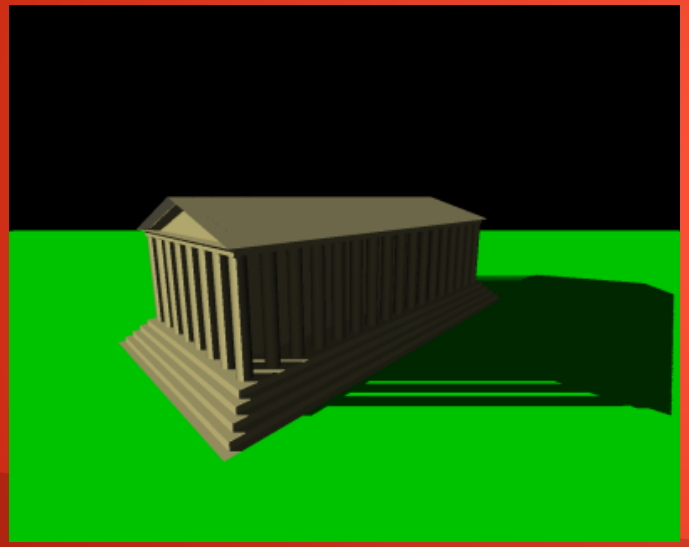
- For first pass: Render the scene from perspective of a light source.

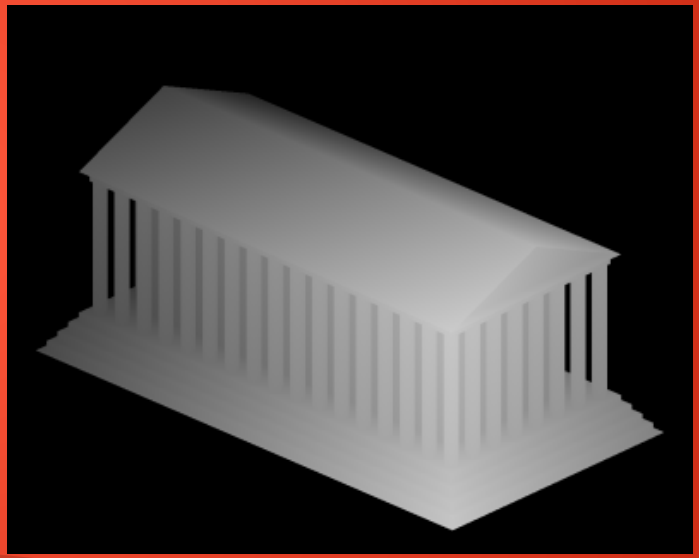


Shadow Mapping – Creating Map

- Shaders don’t just create colour output!
- Recall from Rendering Pipeline: Per-Sample Operations.
- Depth Tests using Depth Buffer values.
- Depth Buffer is another buffer along with Colour Buffer that holds a value between 0 and 1 that determines how deep in to frustum a fragment is.
- 0 is on the Near Plane (close to the camera).
- 1 is on the Far Plane (far from the camera).

Shadow Mapping – Creating Map





Shadow Mapping – Creating Map

- How to extract depth buffer data?
- Framebuffer Object!
- Normally, Framebuffer bound is ‘0’.
- This is the default buffer (the one drawn to the screen when buffer swap is called).
- We can find a separate Framebuffer and draw to that…
- Then use the data as we wish!

Shadow Mapping – Creating Map

- glGenFramebuffers(1, &FBO);
- Create a texture the usual way, but…
- glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, width, height, 0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
- GL_DEPTH_COMPONENT: Single float value, unlike RGB which had three.
- Data is NULL, so we have created an empty texture with dimensions width x height.

Shadow Mapping – Creating Map

- Set Framebuffer to write to texture with:

  glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_2D, textureID, 0);

- GL_DEPTH_ATTACHMENT: Tells Framebuffer to only write Depth Buffer data.
- glDrawBuffer(GL_NONE);

  glReadBuffer(GL_NONE);

- These override colour data draw/read operations. We don’t want to output colour with our shadow map!

Shadow Mapping – Creating Map

- Shader itself is simple:
  - 1. Apply Projection and View matrices as if light source is the camera.
  - 2. Apply model matrix of each object.
  - 3. Fragment Shader isn't even needed: Depth buffer is written automatically.
- Directional Light shadow map works differently to Point/Spot Light shadow maps!
- View Matrix position should consist of reverse of Directional Light's direction (simulating light in that direction).
- View Matrix direction is simply the direction of the light.
- Projection Matrix is different: Frustum of Perspective Projection fans out! Directional Light rays are all parallel, they must not fan out.
- Solution: Orthographic Projection Matrix.
- glm::ortho(-20.0f, 20.0f, -20.0f, 20.0f, 0.01f, 100.0f);

Shadow Mapping – Using Map

- After rendering the scene with the Shadow Map shader, the texture bound to it is occupied with Shadow Map data.
- Make sure to unbind the Framebuffer used for the shadow map!
- Now we need to bind the texture to our main shader and use it.

Shadow Mapping – Using Map

- Need access to the View Matrix used in the Shadow Map Shader (the one using the light's perspective).
- Use this to get the current fragment position in relation to the light source.
- Need to create a way to access points on the Shadow Map with the light source perspective's fragment co-ordinates…
- Therefore, need to convert light source perspective fragment's co-ordinates to "Normalized Device Co-ordinates" (values between -1 and 1, like when we started).

Shadow Mapping – Using Map

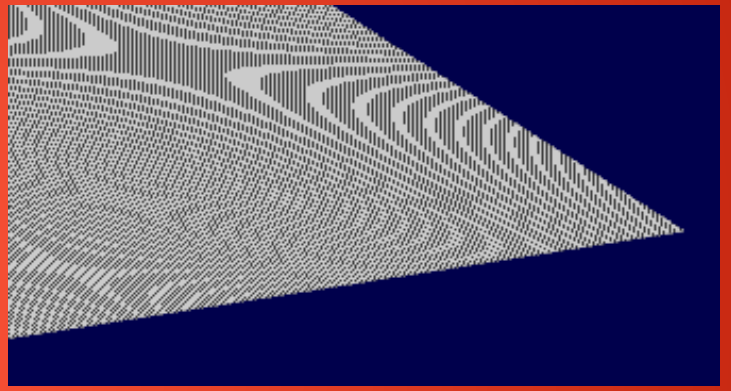
- Need to perform a “perspective divide”.
- Similar to how co-ordinates are created when moving to the Fragment Shader anyway…
- However this is only applied to gl_Position.
- We need to do it manually for the position relative to the light source.
- Easy calculation: Divide vector by it’s ‘w’ component. This is why we use a vec4!
- vec3 projCoords = LightSpacePos.xyz / LightSpacePos.w;
- Then we need to scale the projCoords to 0, 1, to match the 0, 1 values of a texture (recall textures use u- and v-axis between 0 and 1).
- projCoords = (projCoords * 0.5) + 0.5;

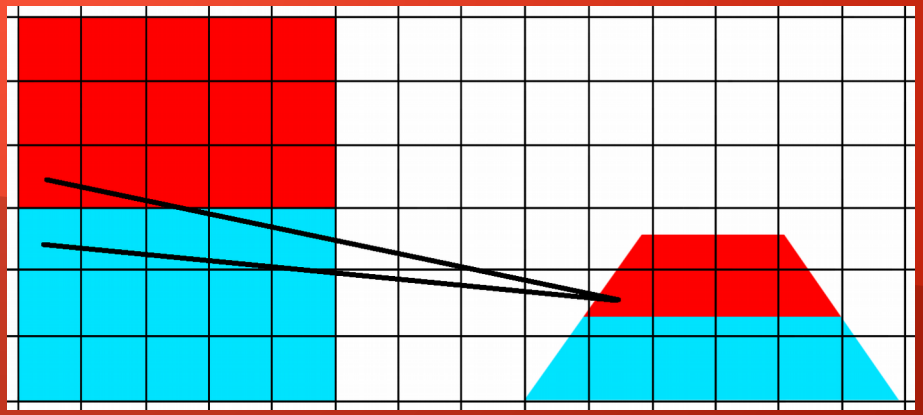
Shadow Mapping – Using Map

- Now use texture function to get closest depth measured during Shadow Map pass.
- float closest = texture(shadowMap, projCoords.xy).r;
- Grab z value from projCoords.
- z-axis on normalised co-ordinates is between 0 and 1, just like depth, and so can be treated as such.
- Compare current and closest depth…
- If current larger than closest: It is further away than the first point the light hits at that fragment! So it must be in shadow.
- Otherwise: It is the same point, so it must be getting lit by the light.
- To apply shadow, simply add or remove diffuse and specular (retain ambient, remember: Ambient Light is ALWAYS present).
- colour = fragColour * (ambient + (1.0 - shadow) * (diffuse + specular));

Shadow Mapping – Shadow Acne

- Shadow Acne occurs due to resolution issues.
- Imagine lighting a surface at an angle…
- When rendering from a less slanted angle, two pixels may converge to one texel on the shadow map!
- One point could be mistaken as being behind a point next to it.





Shadow Mapping – Shadow Acne

- Solution: Add a slight bias.
- Effectively moving everything slightly towards the camera to fake a closer depth.
- Try to keep the bias small, or…
- “Peter Panning” occurs.
- Bias offset causes areas close to shadow source to disappear because depth values are close.

Shadow Mapping – Oversampling

- What about areas outside of the Projection frustum used to create the shadow map?
- Values will be outside 0,1 range and therefore always create shadows!
- Solution:
  - Set texture type to use border with values all consisting of 0 (always lowest depth value so always lit).
  - For values beyond far plane and therefore greater than 1: Initialise to 0.
- See coding video for implementation.

Shadow Mapping – PCF

- Edges of shadows are limited to resolution of texture shadow map is written to.
- This causes unsightly pixelated edges.
- Solution: Sample surrounding texels and calculate average. Apply only partial shadows for shadowed areas.
- Also known as: Percentage-Closer Filtering (PCF).
- Can get dangerously intensive if not used correctly.

Shadow Mapping – PCF

- Get depth values of surrounding texels, such as the 8 immediate surrounding.
- Determine if in shadow.
- If yes: Increment shadow value.
- When done, divide shadow value by number of samples taken.
- Apply percentage of shadow using this value.
- E.g. Shadow value calculated as 3, and 9 samples are taken. 3/9 = 0.333… So apply 33% shadow to that pixel.
- More samples: Better fade effect, but…
- Keep in mind, this set of samples will be taken for EVERY fragment, so instead of being one calculation, it becomes 9x calculations just for using immediate surrounding texels!

Summary

- Shadows created by texture maps of depth data.
- Depth data created by rendering scene from point of view of light source.
- Do two passes: One to create shadow maps and one to render scene.
- Compare depth of fragment from light’s perspective to value on shadow map texture.
- Add bias to remove shadow acne.
- Set values from beyond sampling region to ‘0’ (no shadow).
- Use PCF algorithms to fade shadow edges.

See you next video!