02561 Computer Graphics

Shadow mapping and off-screen buffers

Jeppe Revall Frisvad

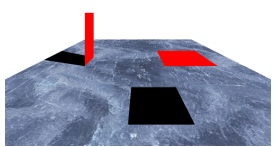
November 2025

Drawing objects with different shaders

- Note that the vertex and fragment shaders have an `entryPoint` in the pipeline. These are the names of the WGSL shader functions to be used.
- Make different pipelines and different bind groups for each set of shaders functions that you need to use for drawing.
- Make different uniform buffers for the different bind groups if the uniform variables need to change.
- In the render pass, set the pipeline and the bind group (and the buffers if they need to change) before drawing.
- If fragment shaders have different output targets, they need to be implemented using different render passes with different color and depth attachments.
- A 32-bit floating point RGBA render target of resolution 1024×1024:

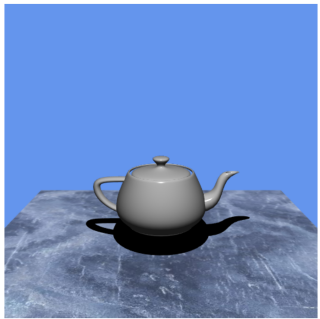
```
const renderTexture = device.createTexture({
    size: [1024, 1024, 1],
    format: 'rgba32float',
    usage: GPUTextureUsage.RENDER_ATTACHMENT | GPUTextureUsage.TEXTURE_BINDING,
});
```

Exercise: Jumping teapot on a marble table top (W09P1-2)



+

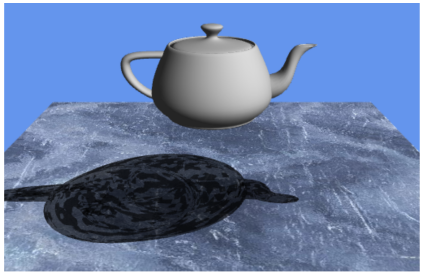
→



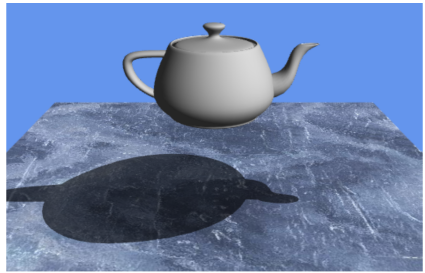
- Starting from the result of Worksheet 8, replace the two shadow casting quads by the Newell teapot (loaded from an obj file as in Worksheet 5).
- The ground plane and the teapot need different shaders.
- Since the two objects have different vertex attributes, it makes sense to keep them in different vertex buffers.
- Create buttons for switching light circulation and vertical teapot movement on/off.

Projection shadows and alpha blending (improving with stencil)

- Alpha blending multiple shadow polygons with each other leads to an undesired non-uniform shadow effect.
- We can add an 8-bit stencil buffer to the depth buffer by replacing `"depth24plus"` with `"depth24plus-stencil8"`.
- The stencil buffer is used as a mask to only draw shadow polygons once in a pixel (only draw if a test passes, change the value in the buffer when drawing, after this the test fails and no more shadow polygons are drawn in this pixel).



$\rightarrow$



Using the stencil buffer

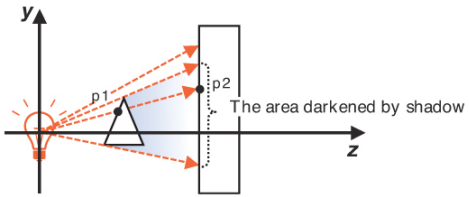
- In the render pass, we can set a reference value: `pass.setStencilReference(1);`
- In the pipeline, we can decide how to use the reference value:

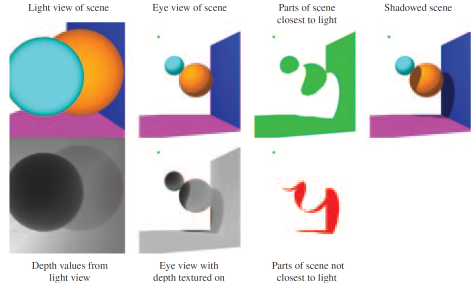
```
depthStencil: {
    depthWriteEnabled: true,
    depthCompare: 'less',
    format: 'depth24plus-stencil8',
    stencilFront: { compare: 'always', passOp: 'replace', },
    stencilBack:  { compare: 'always', passOp: 'replace', }
},
```

- Compare functions: `never`, `less`, `equal`, `less-equal`, `greater`, `not-equal`, `greater-equal`, `always`.
- Stencil operations (a `failOp` can be set as well): `keep`, `zero`, `replace`, `invert`, `increment-clamp`, `decrement-clamp`, `increment-wrap`, `decrement-wrap`.
- The stencil buffer is cleared together with the depth buffer:

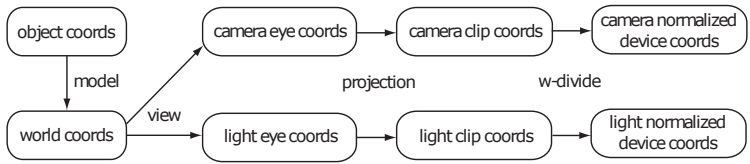
```
depthStencilAttachment: {
    view: depthTexture.createView(),
    depthLoadOp: "clear", depthClearValue: 1.0, depthStoreOp: "store",
    stencilLoadOp: "clear", stencilClearValue: 0, stencilStoreOp: "store",
    stencilReadOnly: false,
}
```

Shadow mapping



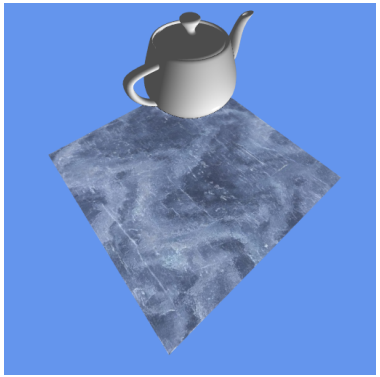


- Render the scene from the point of view of the light source.
- Store the depth seen from the light source ($\boldsymbol{p}_1$) in a shadow map (a texture).
- For each observed fragment ($\boldsymbol{p}_2$), check if the depth of $\boldsymbol{p}_2$ as seen from the light source is significantly different from the depth of $\boldsymbol{p}_1$ available in the shadow map.



Rendering from the point of view of the light source

- The light source needs its own projection and view matrices: $\boldsymbol{P}_\ell$ and $\boldsymbol{V}_\ell$.
  - Directional light: orthographic projection matrix.
  - Point light: perspective projection matrix.
- Use $\boldsymbol{P}_\ell$ and $\boldsymbol{V}_\ell$ to make the view volume a tight bound around the scene.

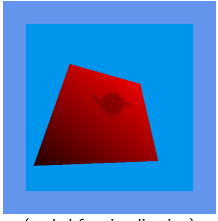


Rendering depth as seen from the light to a texture (W09P3)

- Rendering to a texture requires a pre-pass (a render pass before the regular pass).
- We make a pipeline for the pre-pass that uses a set of shaders simply drawing the fragment depth:

```
@vertex
fn main_vs_depth(@location(0) inPos: vec4f) -> @builtin(position) vec4f
{
    return uniforms.lightViewProj*uniforms.model*inPos;
}

@fragment
fn main_fs_depth(@builtin(position) fragcoord: vec4f) -> @location(0) vec4f
{
    return vec4f(vec3f(fragcoord.z), 1.0);
}
```



(scaled for visualization)

- The depth value z_ℓ (`fragcoord.z`) is the z-coordinate in NDC space:

$$z_\ell = \frac{p_{\text{clip},\ell,z}}{p_{\text{clip},\ell,w}} \quad , \qquad \boldsymbol{p}_{\text{clip},\ell} = \boldsymbol{P}_\ell \boldsymbol{V}_\ell \, \boldsymbol{p}_{\text{world}} \, .$$

- The same view-projection transformation ($\boldsymbol{P}_\ell \boldsymbol{V}_\ell$, `uniforms.lightViewProj`) is used by other shaders for lookup into the shadow map.

Shadow mapping in the fragment shader (W09P4)

- What are the texture coordinates to be used for the shadow map?
- We need the NDC fragment position from the perspective of the light source scaled to $[0, 1]$ (`shadowCoords`):

$$(u_\ell, v_\ell, z_\ell) = \frac{\boldsymbol{p}_{\text{clip},\ell,xyz}}{p_{\text{clip},\ell,w}} * (0.5, -0.5, 1.0) + (0.5, 0.5, 0.0)\,, \qquad \boldsymbol{p}_{\text{clip},\ell} = \boldsymbol{P}_\ell \boldsymbol{V}_\ell \boldsymbol{p}_{\text{world}}\,.$$

- The texture coordinates are then $(u_\ell, v_\ell) * (W, H)$, where $W \times H$ is the resolution of the shadow map and $*$ is elementwise multiplication.
- The depth of $\boldsymbol{p}_{\text{world}}$ from the light is z_ℓ and the depth seen from the light source is

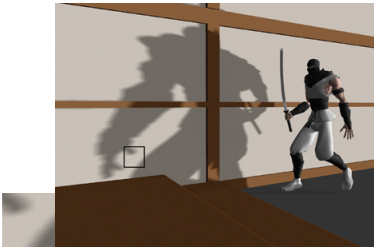
```
let depth = textureLoad(shadowMap, shadowCoords.xy*vec2f(W, H), 0).r;
```

- The fragment is in shadow if `depth` $< z_\ell - 0.001$
  (this means that some other surface is seen from the light).
- The surface should only reflect ambient light ($k_d L_a$) when in shadow.

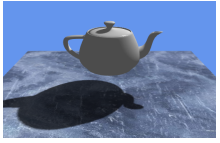
Percentage-closer filtering

- Aliasing (staircase/pixelation/jaggies) is a significant issue in shadow mapping.
- Sampling an area of 4-by-4 texels is a significant improvement.

0	0	0	0	0	0	0	0	1
0	0	0	0	0	0	1	1	1
0	0	0	0	0	1	1	1	1
0	0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1	1
0	0	0	0	1	1	1	1	1
1	1	1	1	1	1	1	1	1



- For a shadow map of resolution `uniforms.shadowMapRes`:



```
fn texture_pcf(shadowCoords: vec3f, epsilon: f32) -> f32
{
    var result = 0.0f;
    for(var y = -1.5; y <= 1.5; y += 1.0) {
        for(var x = -1.5; x <= 1.5; x += 1.0) {
            let texCoords = vec2u(shadowCoords.xy*uniforms.shadowMapRes + vec2f(x, y));
            let depth = textureLoad(shadowMap, texCoords, 0).r;
            result += select(1.0, 0.0, depth < shadowCoords.z - epsilon);
        }
    }
    return result/16.0;
}
```

Michael Bunnell and Fabio Pellacini. Shadow map antialiasing.
In *GPU Gems*. Chapter 11. Addison-Wesley, 2004.
https://developer.nvidia.com/gpugems/GPUGems/gpugems_ch11.html