# 02561 Computer Graphics

## Projection shadows, blending, and depth sorting

Jeppe Revall Frisvad

November 2024

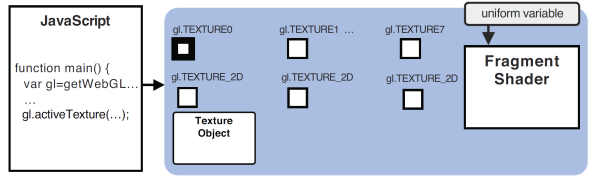
# Multitexturing

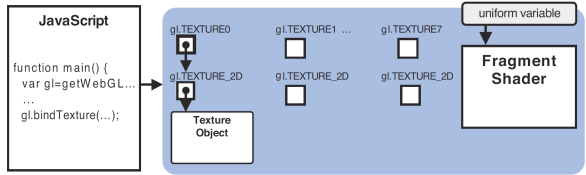
- Simplistic texture mapping fragment shader:

```
precision mediump float;
varying vec2 fTexCoord;
uniform sampler2D texMap;
void main()
{
  gl_FragColor = texture2D(texMap, fTexCoord);
}
```

- JavaScript binding a texture to an active texture:

```
gl.activeTexture(gl.TEXTURE0);
gl.bindTexture(gl.TEXTURE_2D, texture0);
```




- Setting the texture used in a shader to one of the active textures:

```
gl.uniform1i(gl.getUniformLocation(program, "texMap"), 0);
```

- The `0` corresponds to `gl.TEXTURE0`, use `1` for `gl.TEXTURE1`, etc.

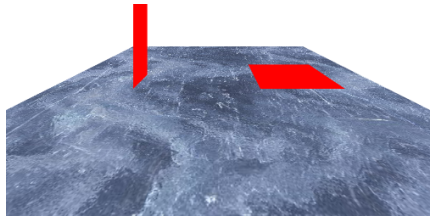
Enables multiple textures in one shader.
Enables one shader with changing textures.

# Exercise: one shader with changing textures (W08P1)

- Load a texture from a file (`gl.TEXTURE0`).
- Create a one-color-texture (`gl.TEXTURE1`):

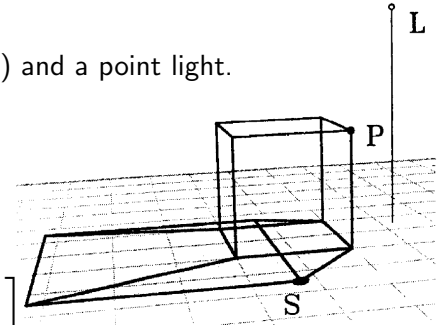
```
gl.texImage2D(gl.TEXTURE_2D, 0, gl.RGB, 1, 1, 0, gl.RGB, gl.UNSIGNED_BYTE, new Uint8Array([255, 0, 0]));
```

- Draw a scene with three quads using one shader with one texture variable but assigning a different texture for different objects.



# Projection shadows

- Suppose we have a flat ground plane ($y = y_g$) and a point light.
- We can then use perspective projection to project geometry to the ground plane.
- Drawing these projected shadow polygons in black, we have (black) shadows.
- How to do the projection?




Perspective projection: $\boldsymbol{M}_p = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & \frac{1}{d} & 0 & 0 \end{bmatrix}$ (projection to $y = d$ due to $w$-divide)

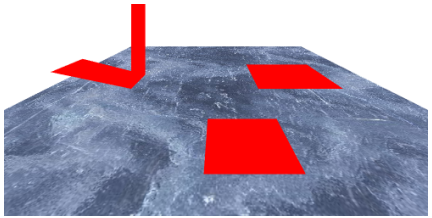
- If $\boldsymbol{p}_\ell = (x_\ell, y_\ell, z_\ell)$ is the light source position, then $d = -(y_\ell - y_g)$ and the model matrix of the shadow polygons is $\boldsymbol{M}_s = \boldsymbol{T}_{\boldsymbol{p}_\ell} \boldsymbol{M}_p \boldsymbol{T}_{-\boldsymbol{p}_\ell} \boldsymbol{M}$,

  where the $\boldsymbol{T}$-matrices are translations to and from a local space where the light source is in the origin, and $\boldsymbol{M}$ is the original model matrix of the object.

- J. F. Blinn. Me and My (Fake) Shadow. *IEEE Computer Graphics & Applications* 8(1), pp. 82–86. January 1988.

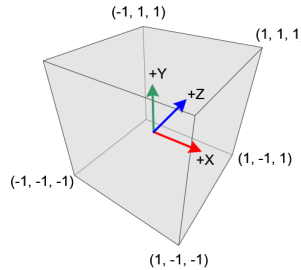
# Exercise: shadow projection matrix (W08P2)

- Set up a circulating point light in the plane $y = 2$: centre $(0, 2, -2)$, radius 2.
- It is good practice to include a button for toggling circulation on/off.
- Build a matrix $\boldsymbol{M}_s$ that projects shadow polygons to the plane $y = y_g = -1$.
- Draw shadow polygons using $\boldsymbol{M}_s$ as their model matrix.
- Draw the shadow polygons after the ground quad but before the red quads.



# Modifying the depth test

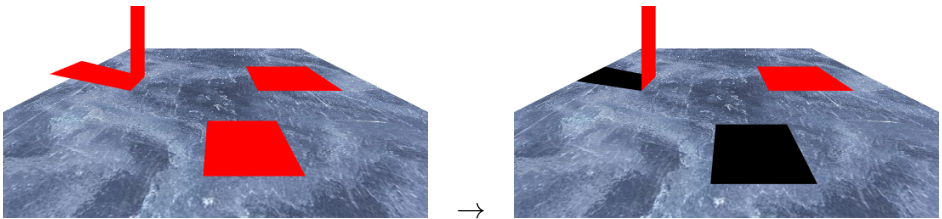
NDC space




- We usually want to draw the closest surface.
- This has smallest $z$-coordinate in NDC space.
- Default depth test `gl.depthFunc(gl.LESS);`
- This draws fragments when $z_{\text{ndc}}$ is less than the current value in the depth buffer.
- The depth buffer is cleared to $\max z_{\text{ndc}} = 1$.
- When using `gl.depthFunc(gl.GREATER);` nothing is drawn unless something closer to the camera was drawn beforehand.
- Draw the ground quad first, then draw the shadow polygons with a slightly offset plane ($y_g + \epsilon$ in $\boldsymbol{M}_s$), but only if $z_{\text{ndc}}$ is greater than the value in the depth buffer.
- Switch the depth buffer back to default `gl.depthFunc(gl.LESS);` and draw the scene objects normally as the final step.

# Exercise: shadow clipping with the depth test (W08P3)

- Enable depth testing and clear the depth buffer together with the color buffer.
- Introduce a small $\epsilon$-offset into the shadow projection matrix $\boldsymbol{M}_s$.
- Modify the depth test function to accept only greater $z_{\text{ndc}}$ values when (and only when) drawing shadow polygons.
- Introduce a uniform float variable (visibility) in your fragment shader that is 1 when drawing normally and 0 when drawing shadow polygons.
- Multiply the fragment color by this visibility variable to draw shadows black.



# Alpha blending

- Let us use alpha blending to make the shadows dark instead of black.
- Color vectors often have RGBA format, where A is alpha.
- Think of the current value in the color buffer as the destination color:

$$\boldsymbol{d} = (d_r, d_g, d_b, d_a)\,.$$

- The fragment color computed in the fragment shader is then the source color:

$$\boldsymbol{s} = (s_r, s_g, s_b, s_a)\,.$$

- The blending operation replaces $\boldsymbol{d}$ with $\boldsymbol{d}' = \boldsymbol{b} * \boldsymbol{s} + \boldsymbol{c} * \boldsymbol{d}$,
  where $*$ is element-wise multiplication and $\boldsymbol{b} = (b_r, b_g, b_b, b_a)$ and
  $\boldsymbol{c} = (c_r, c_g, c_b, c_a)$ are source and destination blending factors.
- Alpha blending is using the alpha-values to set the blending factors $\boldsymbol{b}$ and $\boldsymbol{c}$.

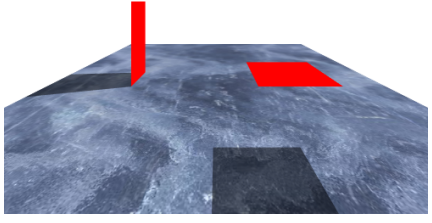
# Exercise: semi-transparent shadow polygons (W08P4)

- As per default, WebGL assumes that we would like to do alpha blending of what we draw in the canvas with whatever is beneath it in the browser window.
- To make alpha blending independent of the colors in the browser window, use:

```
var gl = WebGLUtils.setupWebGL(canvas, { alpha: false });
```

- To enable alpha blending, use: `gl.enable(gl.BLEND);`
- Setting the blend function is setting the blending factors $\boldsymbol{b}$ and $\boldsymbol{c}$.
- The most common blend function is $\boldsymbol{d}' = s_a\boldsymbol{s} + (1 - s_a)\boldsymbol{d}$, which is set by

```
gl.blendFunc(gl.SRC_ALPHA, gl.ONE_MINUS_SRC_ALPHA);
```



# Alpha blending when using WebGPU

- Blending is defined for the fragment target buffer.
- Set the blending function when creating the render pipeline

```
const pipeline = device.createRenderPipeline({
  layout: "auto",
  vertex: {
    module: wgsl,
    entryPoint: "main_vs",
    buffers: [vPositionBufferLayout, vColorBufferLayout]
  },
  fragment: {
    module: wgsl,
    entryPoint: "main_fs",
    targets: [{
      format: canvasFormat,
      blend: {
        color: { operation: "add", srcFactor: "src-alpha", dstFactor: "one-minus-src-alpha" },
        alpha: { operation: "add", srcFactor: "one", dstFactor: "zero" }
      }
    }]
  },
  primitive: {
    topology: "triangle-list",
  },
});
```

- The two `operation` fields correspond to `gl.blendEquation` or `gl.blendEquationSeparate`.
- The `src` and `dst` factors correspond to `gl.blendFunc` or `gl.blendFuncSeparate`.