Debuggin the 3D Cube in WebGL

We started with the early result for the interactive cube.

We fixed the controls and checked the matrices.

We now add lighting.

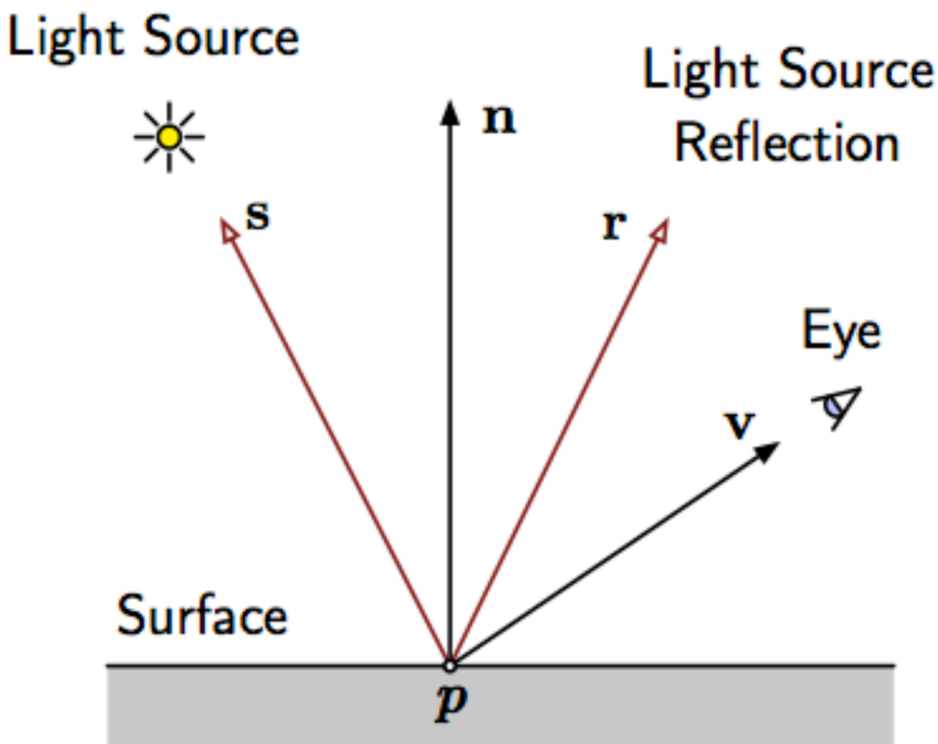
Explanation

👀 means that the code is already in the repository and you just need to look at it.
📋 means you can copy-paste the code and it should work.
📝 means that you need to create a new file
⭕ indicates that you need to do more than just copy-paste the code.
❌ indicates that you need to replace the old code with something new.

In any case you need to understand what you are doing.

Geometry

We now need surface normals to compute the shading according to the Phong model.



Face normals on the cube

👀 The new cube class from `utils.zip` contains a method that generates the normals.

```
generateNormals() {
    const normals = [];
    let front = [0, 0, 1]; // Front face normal
    ... // similar for all faces
    let numVerticesPerFace = 4; // Each face has 4 vertices
    for (let i = 0; i < numVerticesPerFace; i++) {
        normals.push(...front);
    }
    ... // a loop for each face
    return normals;
}
```

Get normals to the vertex shader

📋 We update the vertex shader code and add the normals as attribute.

```
in vec3 aNormal;
```

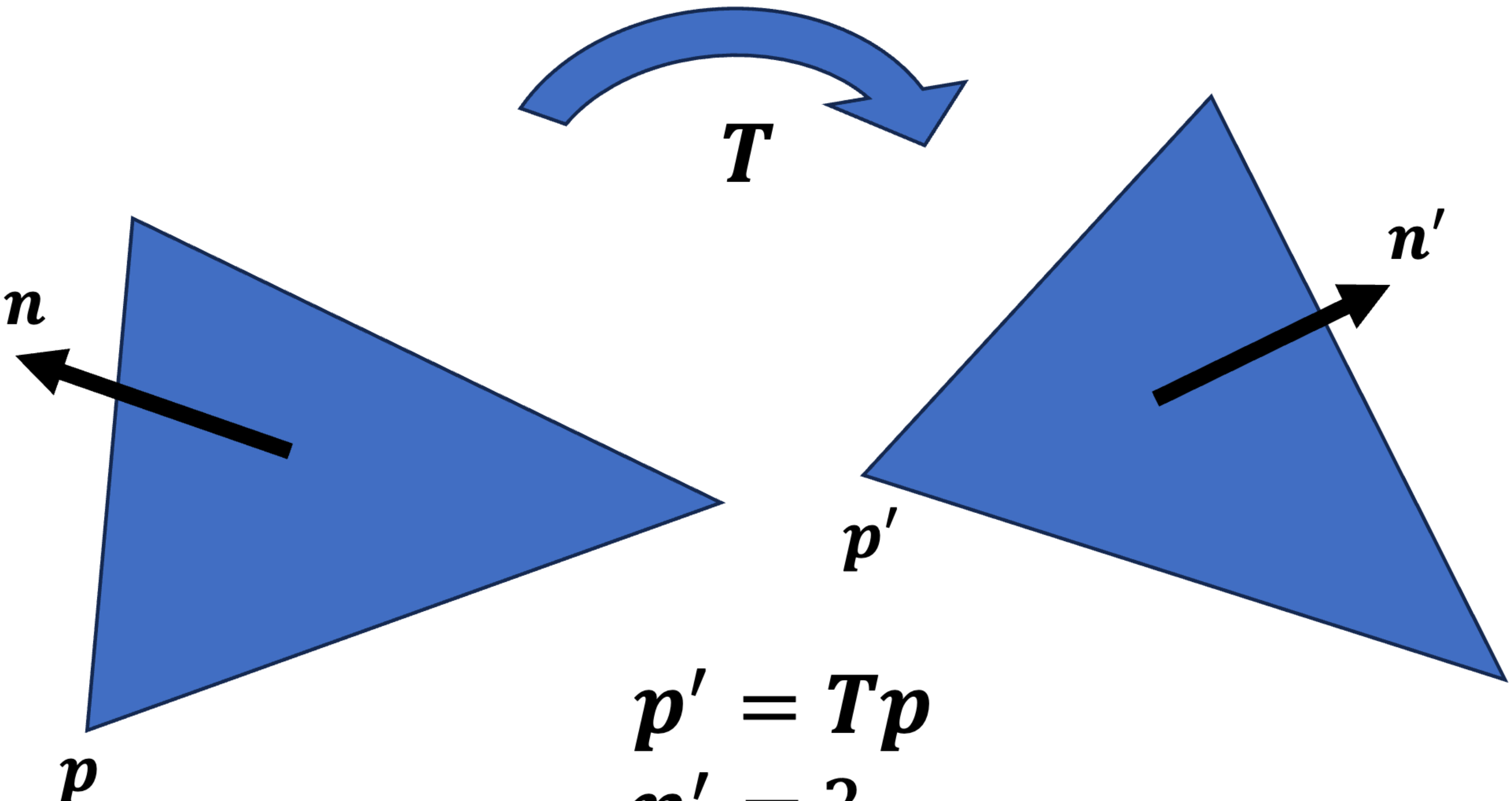
📋 In the script we read the normals into a buffer:

```
const normalBuffer = gl.createBuffer();
gl.bindBuffer(gl.ARRAY_BUFFER, normalBuffer);
gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(cube.normals), gl.STATIC_DRAW);
```

⭕ Use `connectShaderAttributes` to connect the buffer to the vertex shader.

How to transform the normals?

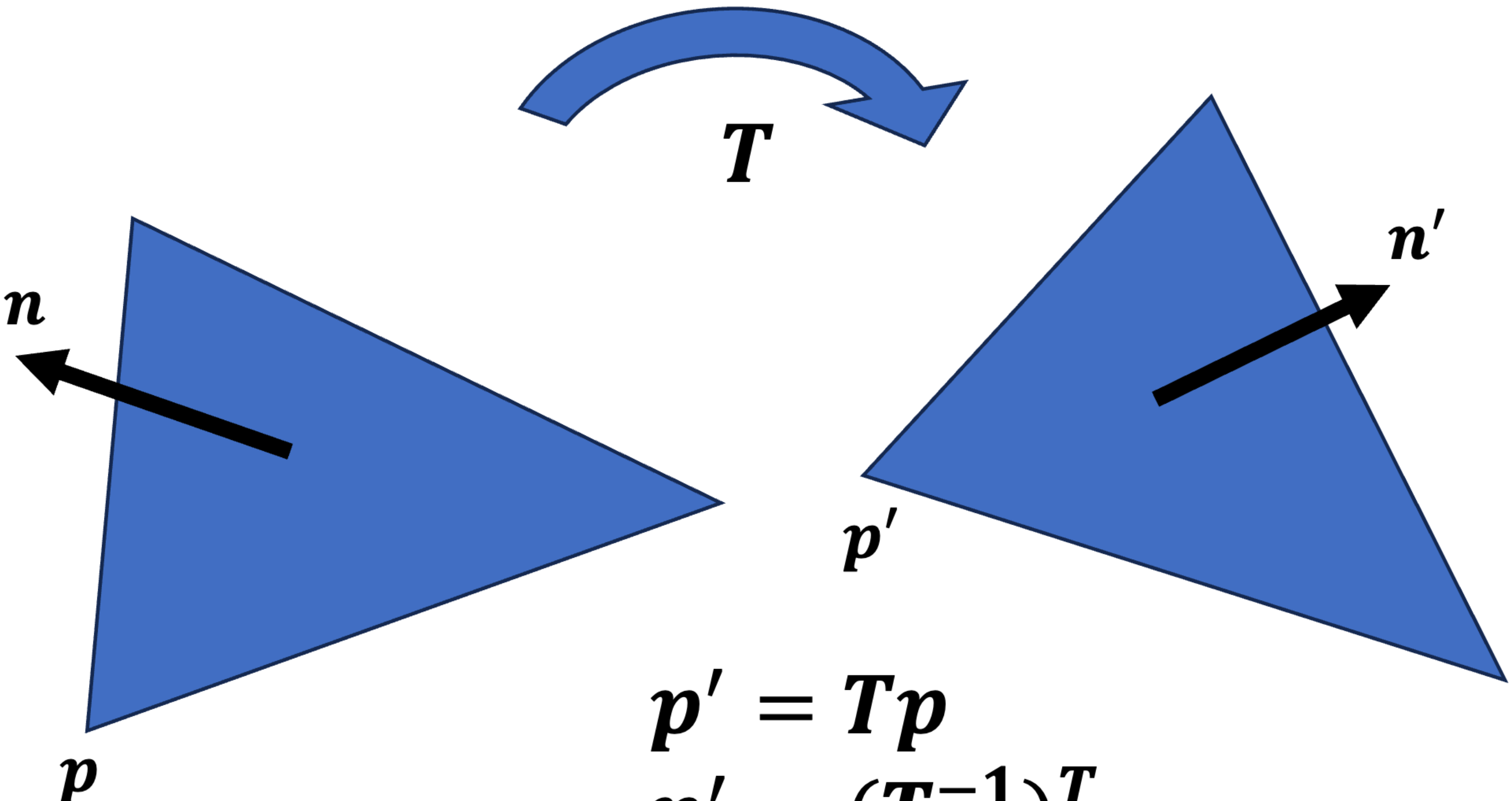
What happens to the normals if a triangle is transformed?



$$p' = Tp$$
$$n' = ?$$

Transforming the normals

We need to transform it with the transposed inverse transform.



$$\boldsymbol{p}' = \boldsymbol{T}\boldsymbol{p}$$
$$\boldsymbol{n}' = (\boldsymbol{T}^{-1})^{\boldsymbol{T}}$$

Transforming the normals (code)

- Insert this code lines at the right positions.

```
const uModelViewInvTLocation = gl.getUniformLocation(program,
            'uModelViewInverseTransposedMatrix');
...
const modelViewInvTMatrix = mat4.create();
mat4.invert(modelViewInvTMatrix, modelViewMatrix);
mat4.transpose(modelViewInvTMatrix, modelViewInvTMatrix);
...
gl.uniformMatrix4fv(uModelViewInvTLocation, false, modelViewInvTMatrix);
```

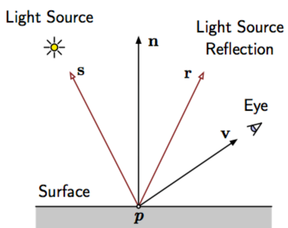
Transforming the normals (in the shader)

- Insert these code lines to the vertex shader.

```
// varying to pass the normal to the fragment shader
out vec3 vNormal;

...

// in the main function apply the transformation
vNormal = mat3(uModelViewInverseTransposedMatrix) * aNormal;
```

Preparing the needed vectors

- We need to provide the vector from the surface to the view/camera/eye position as well as to the light position. We define these vectors in the vertex shader as varying variables to pass them to the fragment shader.




```
uniform vec3 uLightWorldPosition; // light source
uniform vec3 uViewWorldPosition; // eye
...
out vec3 vSurfaceToLight; // s
out vec3 vSurfaceToView; // v
```

Calculating the needed vectors

- We need to calculate the vectors in the vertex shader. We will use the **modelview** matrix to compute the world position of the surface and then calculate the vectors from this position to the light and view positions.

```
// compute the world position of the surface
vec4 worldPosition = uViewMatrix * uModelMatrix * aPosition;
worldPosition = worldPosition / worldPosition.w;
vec3 surfaceWorldPosition = worldPosition.xyz;

// compute the vector of the surface to the light
// and pass it to the fragment shader
vSurfaceToLight = uLightWorldPosition - surfaceWorldPosition;

// compute the vector of the surface to the view/camera
// and pass it to the fragment shader
vSurfaceToView = uViewWorldPosition - surfaceWorldPosition;
```

Fragment Shader

- We need to add the variables also to the fragment shader code.

```
in vec3 vSurfaceToLight;
in vec3 vSurfaceToView;

uniform vec3 uReverseLightDirection;
```

Setting up the light position initially

- We need to set the light position in the world coordinates.

```
// set a point light position
let pointLightPos = vec3.fromValues(0.5, 0.0, 1.0);

// set the light direction.
const lightTarget = vec3.fromValues(0.0, 0.0, 0.0);
let revLightDir = vec3.subtract(vec3.create(), pointLightPos, lightTarget);
vec3.normalize(revLightDir, revLightDir);
```

Get the data to the GPU

- We will use a uniform variable to pass the light position to the shader.

```
const lightWorldPositionLocation = gl.getUniformLocation(program,
        "uLightWorldPosition");
const viewWorldPositionLocation = gl.getUniformLocation(program,
        "uViewWorldPosition");
const reverseLightDirectionLocation = gl.getUniformLocation(program,
        "uReverseLightDirection");
...
gl.uniform3fv(lightWorldPositionLocation, pointLightPos);
gl.uniform3fv(viewWorldPositionLocation, viewPos);
gl.uniform3fv(reverseLightDirectionLocation, revLightDir);
```

Drawing to the Screen

Update the normal transform matrix

⭕ We need to update the normal transform matrix in the render loop. This is necessary because the model matrix changes when we rotate the cube.

```
// Update the inverse transpose matrix
mat4.multiply(modelViewMatrix, viewMatrix, modelMatrix);
mat4.invert(modelViewInvTMatrix, modelViewMatrix);
gl.uniformMatrix4fv(uModelViewInvTLocation, true, modelViewInvTMatrix);
```

Note that this matrix is sometimes called the "normal matrix" in the literature.

Update the fragement shader (directional light)

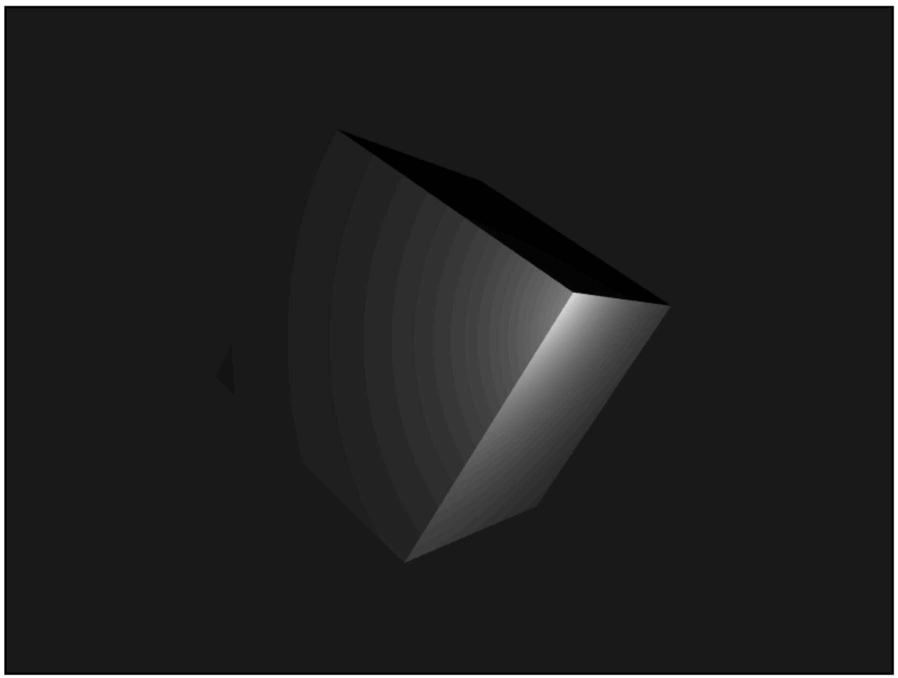
```
void main() {
    vec3 normal = normalize(vNormal);

    float dirLight = dot(normal, uReverseLightDirection);

    // Lets multiply just the color portion (not the alpha)
    outColor = vFragColor;
    outColor.rgb *= dirLight;
}
```

Improvement: Update the fragement shader (point light)

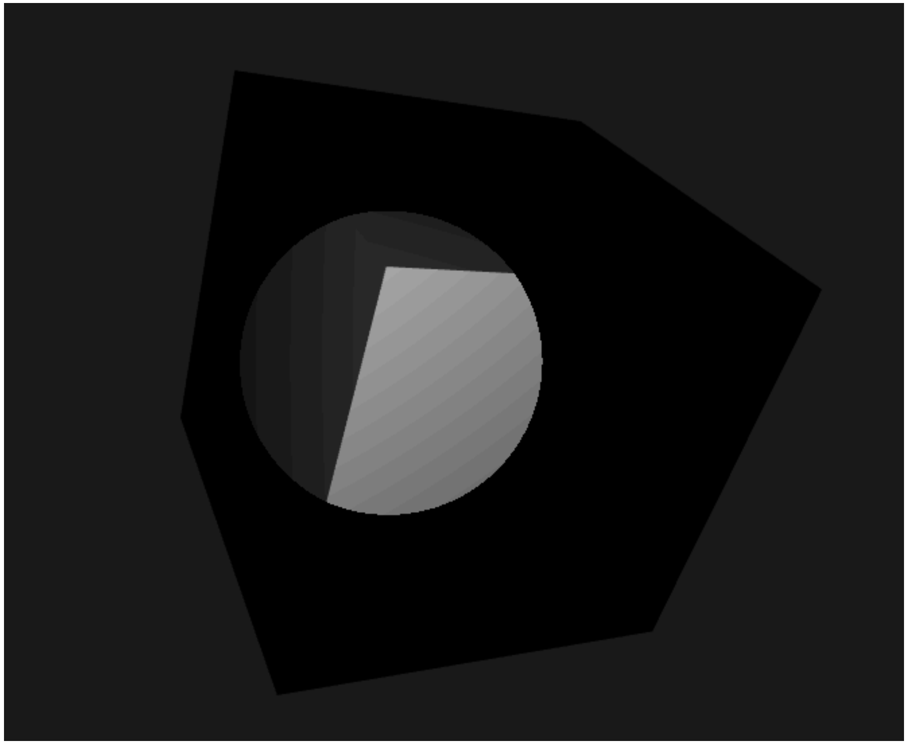
```
void main() {
   vec3 normal = normalize(vNormal);

   vec3 surfaceToLightDirection = normalize(vSurfaceToLight);
   vec3 surfaceToViewDirection = normalize(vSurfaceToView);

   float pointLight = dot(normal, surfaceToLightDirection);

   // Lets multiply just the color portion (not the alpha)
   outColor = vFragColor;
   outColor.rgb *= pointLight;
}
```

Result: Shaded Cube

You should be able to see a shaded cube.



Next step is to create a spot light

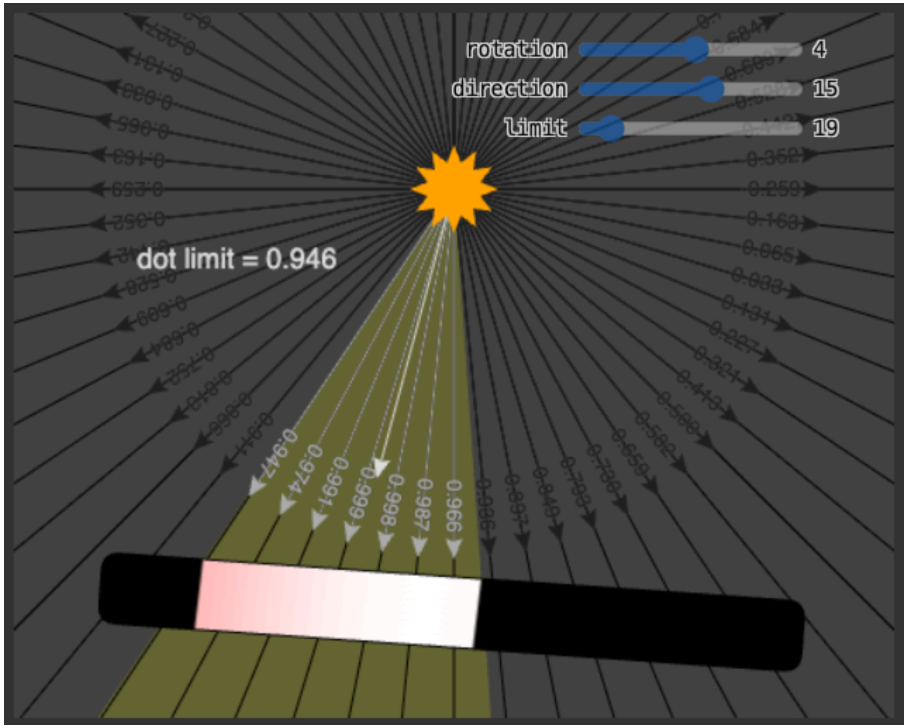


Preparation

We first set the light directly from the front.

```
// set the point light position
let pointLightPos = vec3.fromValues(0.0, 0.0, 1.0);
```

Computation in the fragment shader




```
float dotFromDirection = dot(surfaceToLightDirection, uReverseLightDirection);
```

Using the step function

```
 // inLight will be 1 if we're inside the spotlight and 0 if not
float inLight = step(uOuterLimit, dotFromDirection);
float spotLight = inLight * pointLight;
outColor = vFragColor;
outColor.rgb *= spotLight;
```

Code completion

⭕ We need to add some variables or uniforms to control the spotlight.

- `uOuterLimit` needs to be set
- Start with some value in the shader.
- Extend to a uniform, that can be changed in the script.
- Add a slider to control it interactively.