Lighting the 3D Cube in WebGL

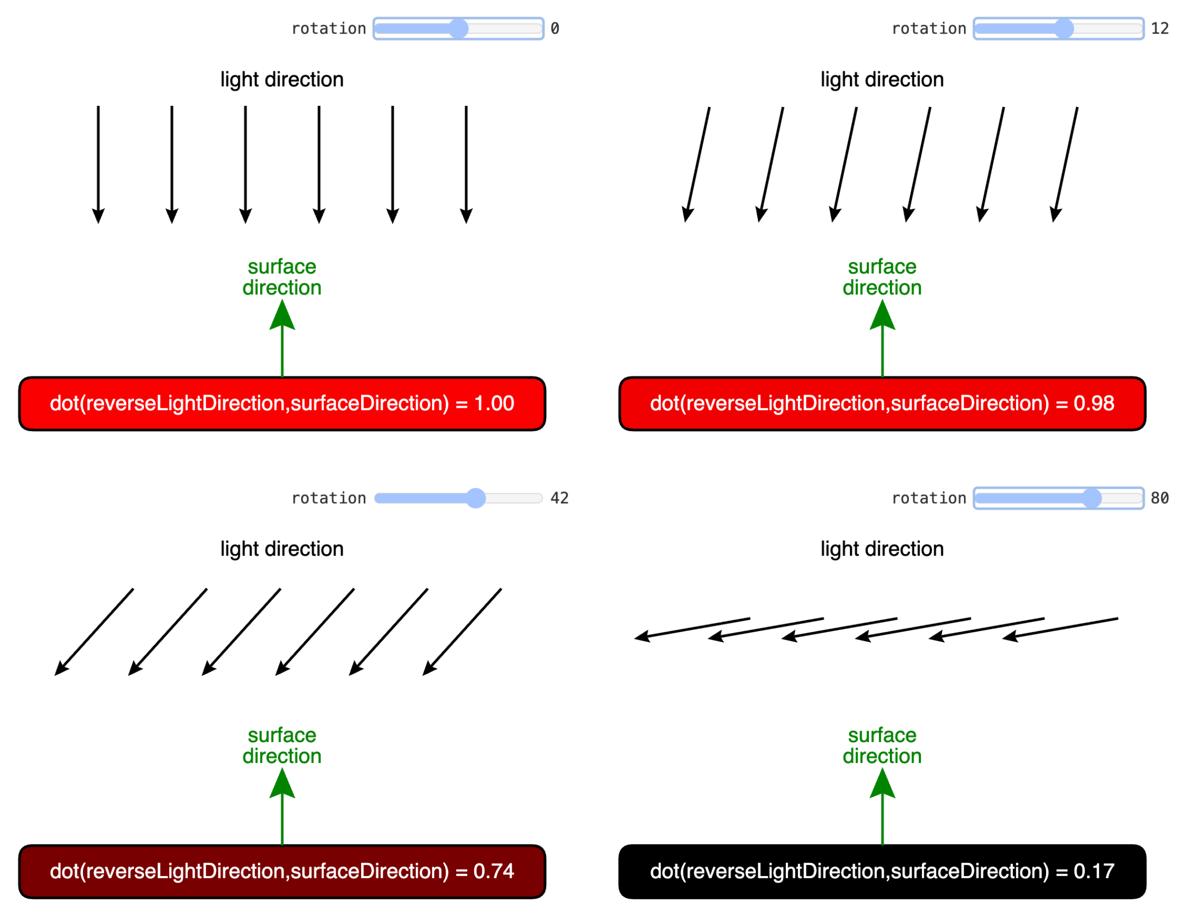
- We continue with the results from last time. We should all have an interactive, textured cube.
- We now add shading and lighting.

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.

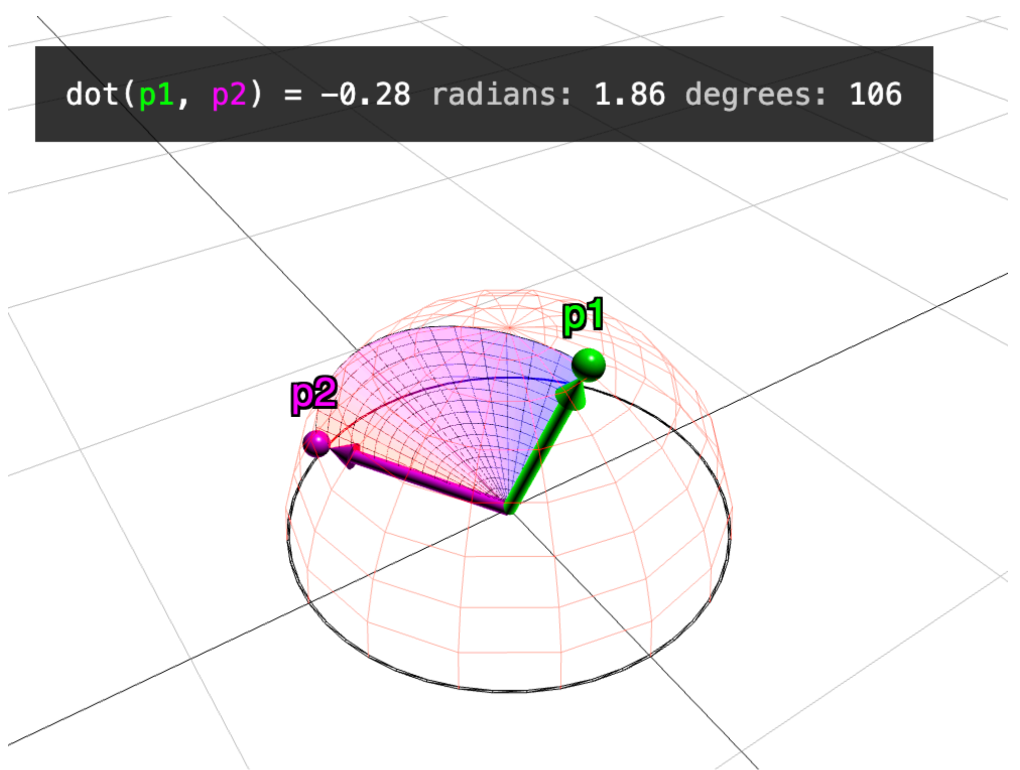
In any case you need to understand what you are doing.

Surface brightness with directional light



Interactive at webgl2fundamentals.org

Reminder: Dot Product



Interactive at webgl2fundamentals.org

Requirements for directional light

We need the following data to compute the shading:

- The **normal vector** of the surface
- The vector of the **light direction**

Face normals on the cube

📋 ⭕ We first need to implement 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.

Pass the normals to the fragment shader

📋 We need to pass the normals to the fragment shader. We can do this by defining a varying variable in the **vertex shader** and assigning the normal to it.

```
// a varying to pass the normal to the fragment shader
out vec3 vNormal;
...
void main() {
    ...
    // Pass the normal to the fragment shader.
    vNormal = aNormal;
}
```

Prepare the uniform variable for the light direction

- We need to prepare the uniform variable for the light direction in the script.

```
import * as vec3 from "./utils/vec3.js";
...
// set the light direction vector
const reverseLightDirectionLocation = gl.getUniformLocation(
        program,
        'uReverseLightDirection');
const reverseLightDirection = vec3.fromValues(0.5, 0.1, 1);
vec3.normalize(reverseLightDirection,
        reverseLightDirection);
...
// Set the light direction uniform
gl.uniform3fv(reverseLightDirectionLocation,
        reverseLightDirection);
```

Prepare the fragment shader

⭕ We need to prepare the **fragment shader** to receive the normal and the light direction.

```
in vec3 vNormal;
...
uniform vec3 uReverseLightDirection;
```

Update the fragement shader (directional light)

- Now the **fragment shader** can compute the shading for a directional light.

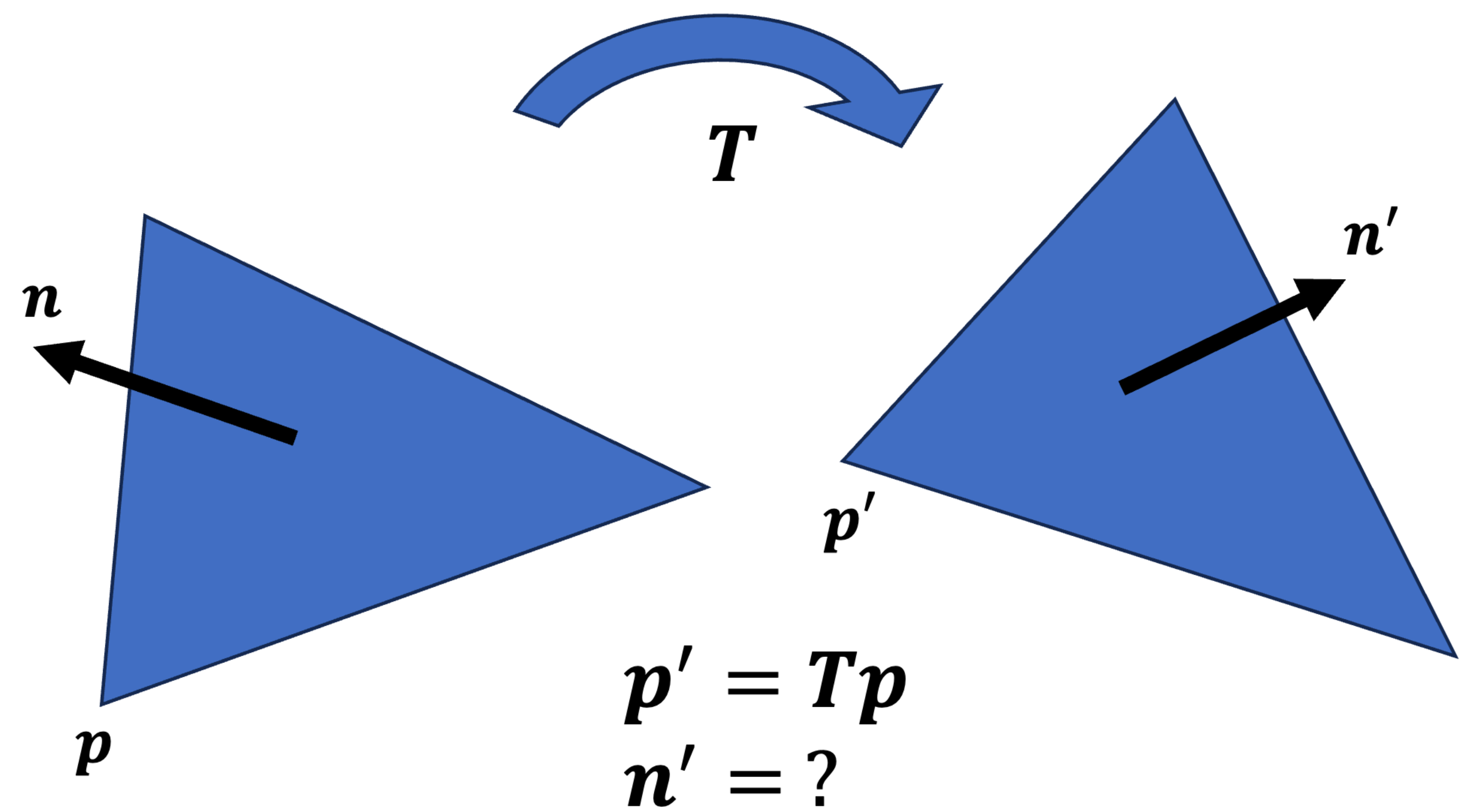
```
void main() {
    vec3 normal = normalize(vNormal);
    float dirLight = dot(normal, uReverseLightDirection);

    // here the outColor is set as before,
    // e.g., with the texture color or the vertex color

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

Result: Shaded Cube

Check if the lighting is correct. The cube should be shaded, but if you rotate it, the shading does not change. This is because we have not yet transformed the normals according to the rotation of the cube.

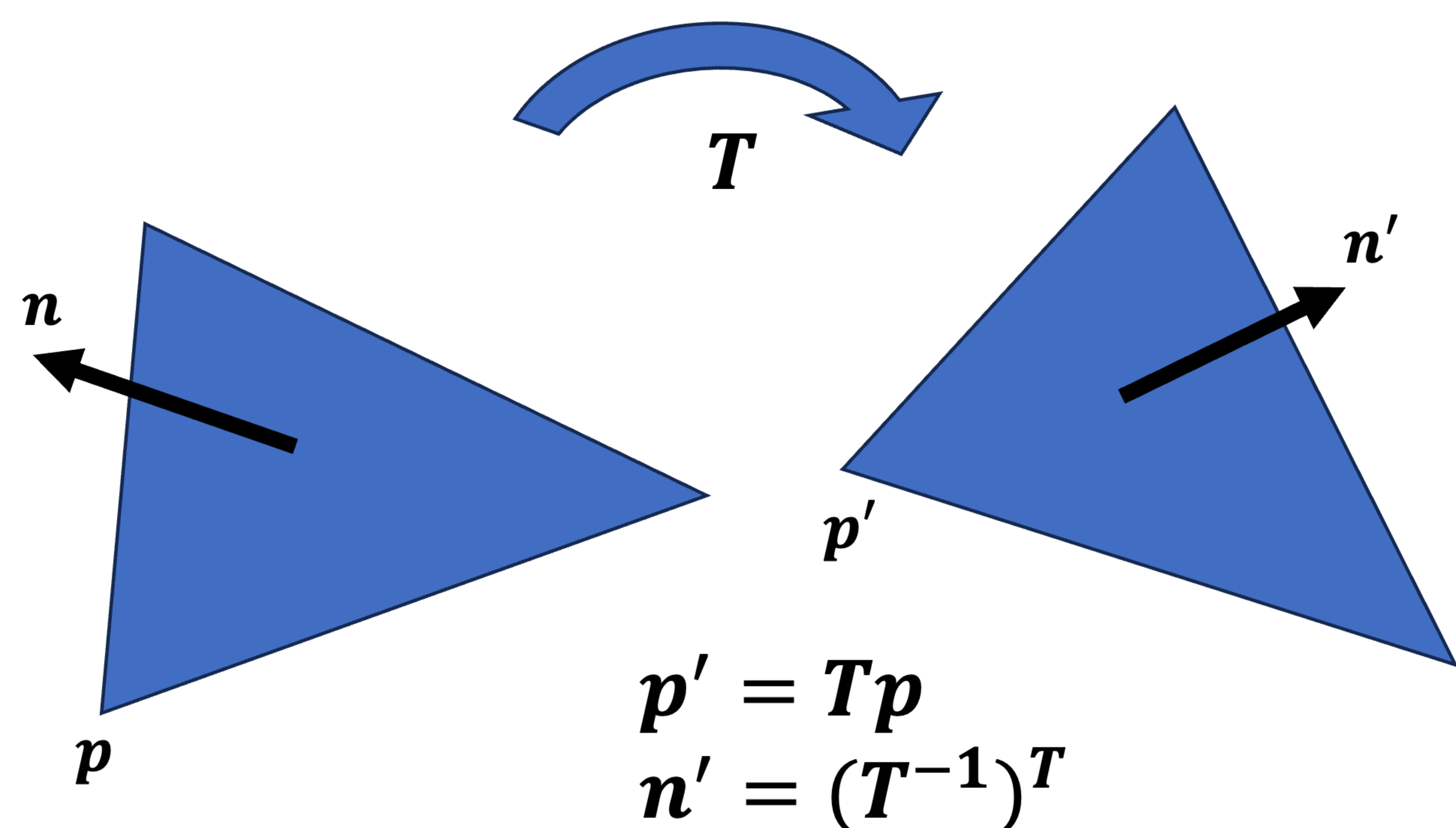
How to transform the normals?

What happens to the normals if a triangle is transformed?



Transforming the normals

We need to transform it with the transposed inverse transform.



Transforming the normals (code)

- Insert these code lines at the right positions.

```
// inverse transposed model matrix
const modelInverseTransposeLocation = gl.getUniformLocation(
        program,
        'uModelInverseTransposeMatrix');
let modelInverseTranspose = mat4.clone(modelViewMatrix);
mat4.invert(modelInverseTranspose, modelInverseTranspose);
mat4.transpose(modelInverseTranspose, modelInverseTranspose);
...
gl.uniformMatrix4fv(modelInverseTransposeLocation,
        false,
        modelInverseTranspose);
```

Do not forget to update the normal transform matrix in the interaction or render loop, because the model matrix changes when we rotate the cube.

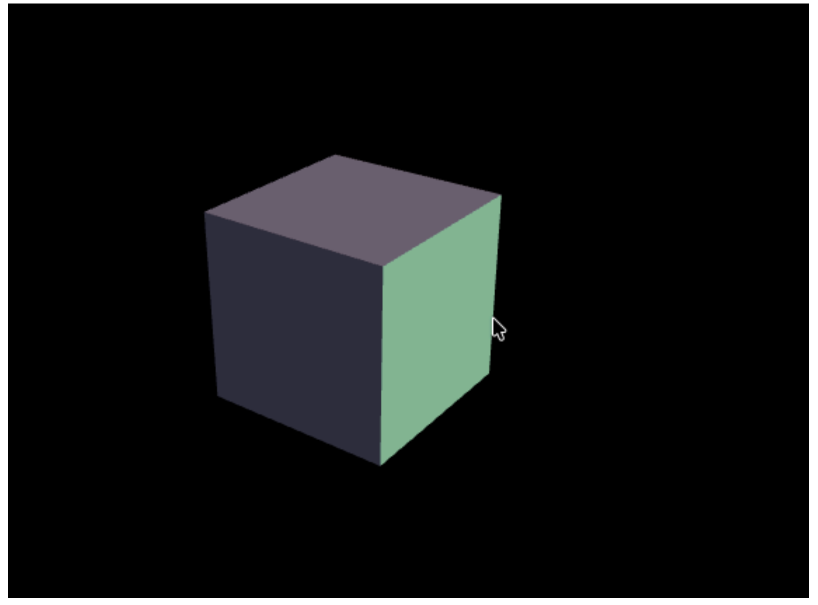
Transforming the normals (in the shader)

- Insert these code lines to the **vertex shader**.

```
uniform mat4 uModelInverseTransposeMatrix;

...

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

The normals should now be transformed correctly and the shading should change when you rotate the cube.

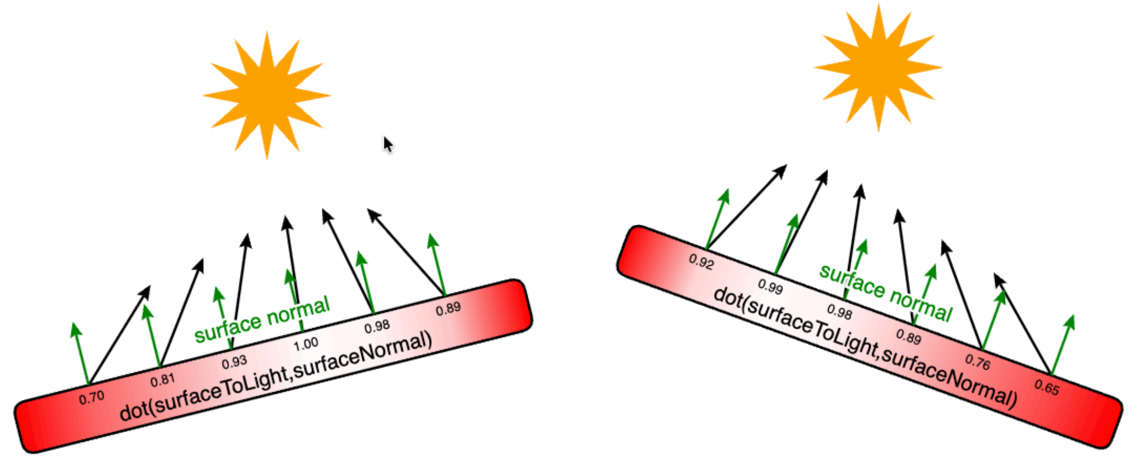
Better shaded cube



Now the shading should behave correctly on rotation. You can see that the faces that are facing the light source are brighter than the ones that are facing away from it.

Point light

Next step is to implement a point light. A point light is a light source that emits light in all directions from a single point in space. Hence, the faces of the cube should be shaded less uniformly depending on their position relative to the light source.



Interactive at webgl2fundamentals.org

Preparing the needed vectors

If we rotate the surface each point on the surface has a different surface to light vector. The dot product of the surface normal and each individual surface to light vector gives us a different value at each point on the surface.

Hence, we need not just a light direction, but a position of the light source in the world coordinate space.

Updating the vertex shader

⭕ We need to calculate the vector from the surface to the light source in the vertex shader. We will use the **modelview** matrix to compute the world position of the surface and then calculate the vector from this position to the light position.

```
uniform vec3 uLightWorldPosition;
...
out vec3 vSurfaceToLight;
...
// in main()
    // compute the world position of the surface
    vec3 surfaceWorldPosition = (uModelViewMatrix * aPosition).xyz;

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

Updating the fragment shader

- We need to add the variable also to the fragment shader code and compute the light intensity using the dot product.

```
in vec3 vSurfaceToLight;

// in the main()
    vec3 surfaceToLightDirection = normalize(vSurfaceToLight);
    float pointLight = dot(normal, surfaceToLightDirection);
    ...
    outColor.rgb *= pointLight;
```

Setting up the light position initially

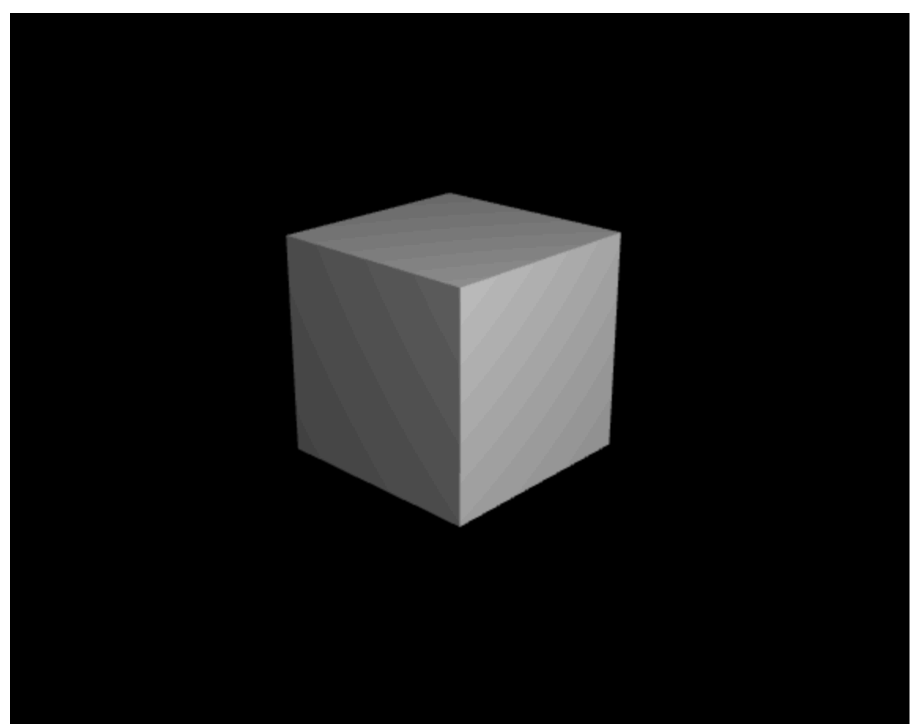
⭕ We need to set the light position in the world coordinates and bring it to the GPU.

```
// point lighting
const lightWorldPositionLocation = gl.getUniformLocation(program,
        "uLightWorldPosition");
...
gl.uniform3fv(lightWorldPositionLocation, [1, 1, 2]);
```

Here, we choose `(1,1,2)` as light position in order to place the light above right of the cube.

Result: Even better shaded Cube

You should be able to see a shaded cube.

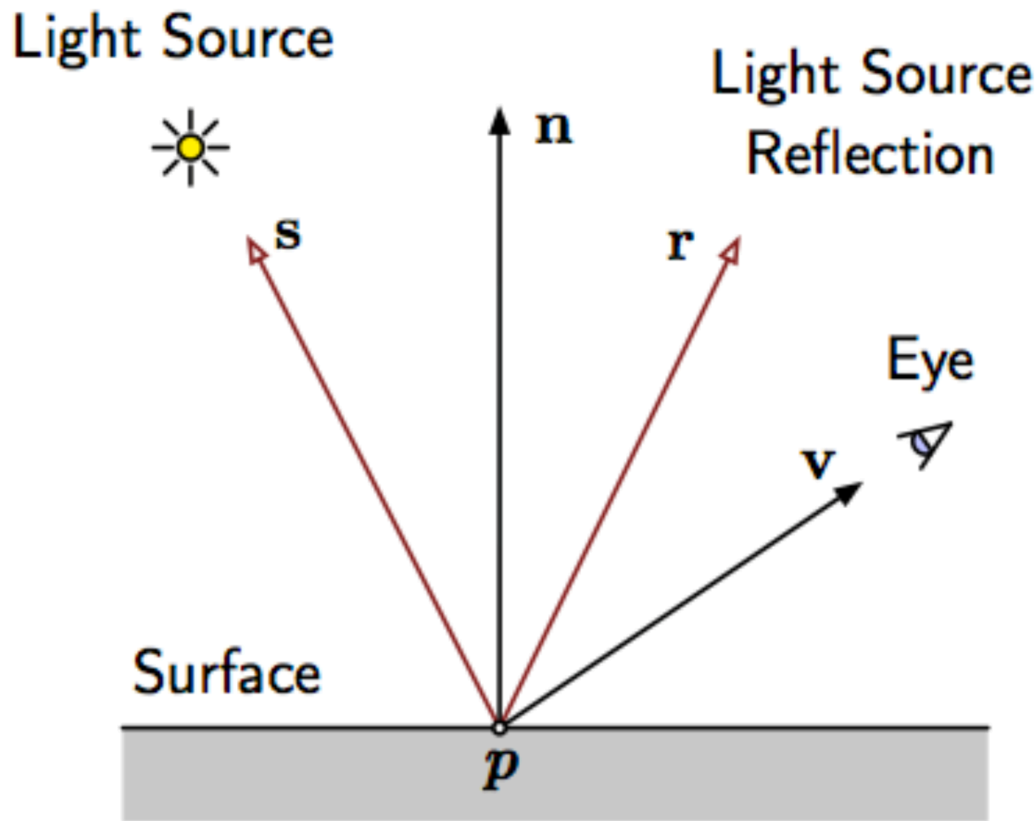


Specular highlights and color

Next, we add specular highlights and color in order to be able to simulate various materials.

Specularity in the real world

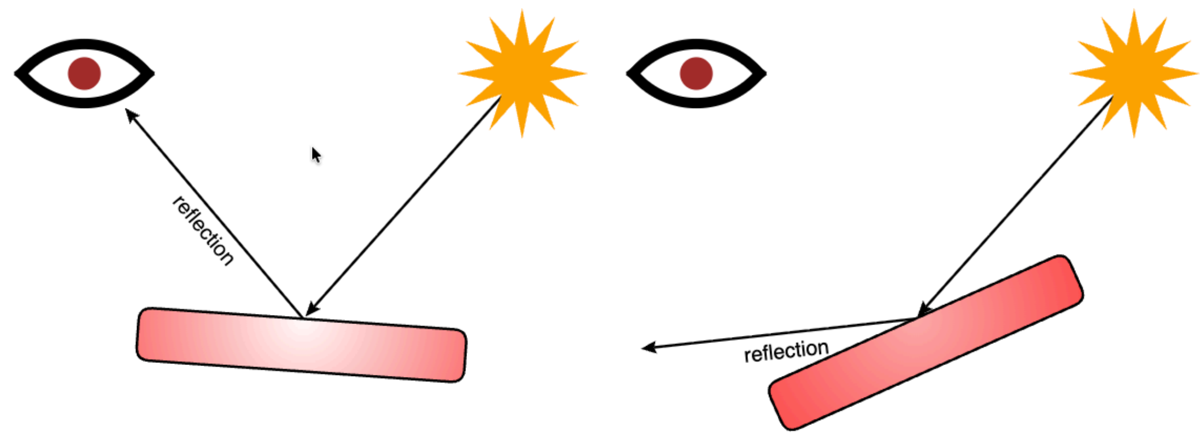
Phong model



The Phong model defines the direction of the reflection.

Reflection position

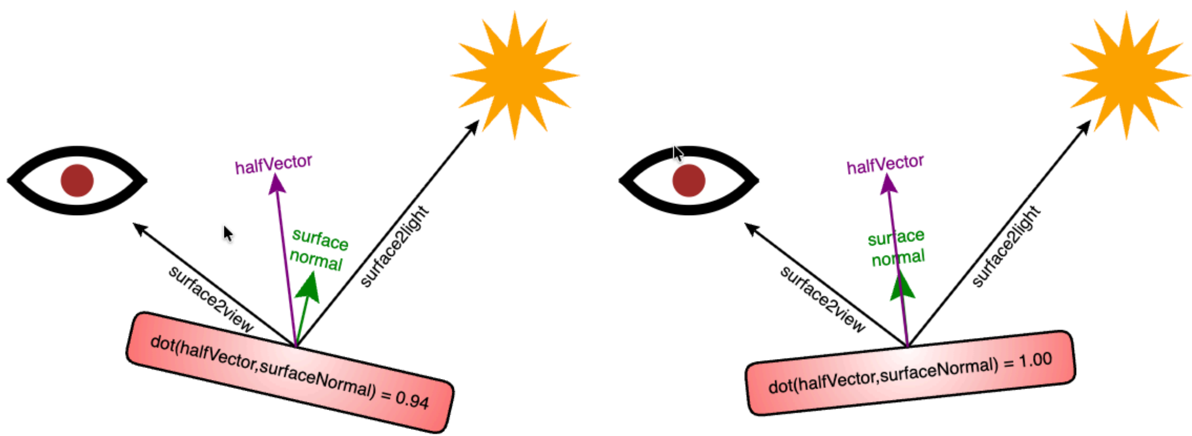
The reflection is only visible, if it points towards the eye resp. the camera.



Interactive at webgl2fundamentals.org

Computing the reflection intensity

We compute the `halfVector` that sits in the middle between the vectors from the surface to the camera/eye respectively to the light source. The reflection is strongest, if this vector is aligned with the surface normal. Hence the mathematical tool of choice is again the **dot product**.



Interactive at webgl2fundamentals.org

Updating the vertex shader

- Add those variables and the computation.

```
uniform vec3 uViewWorldPosition;

...

out vec3 vSurfaceToView;

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

Update the fragment shader

- Add those variables and the computation.

```
in vec3 vSurfaceToView;

...

// in the main()
    vec3 surfaceToViewDirection = normalize(vSurfaceToView);
    vec3 halfVector = normalize(surfaceToLightDirection +
            surfaceToViewDirection);
    float specular = dot(normal, halfVector);
    ...
    outColor.rgb += specular;
```

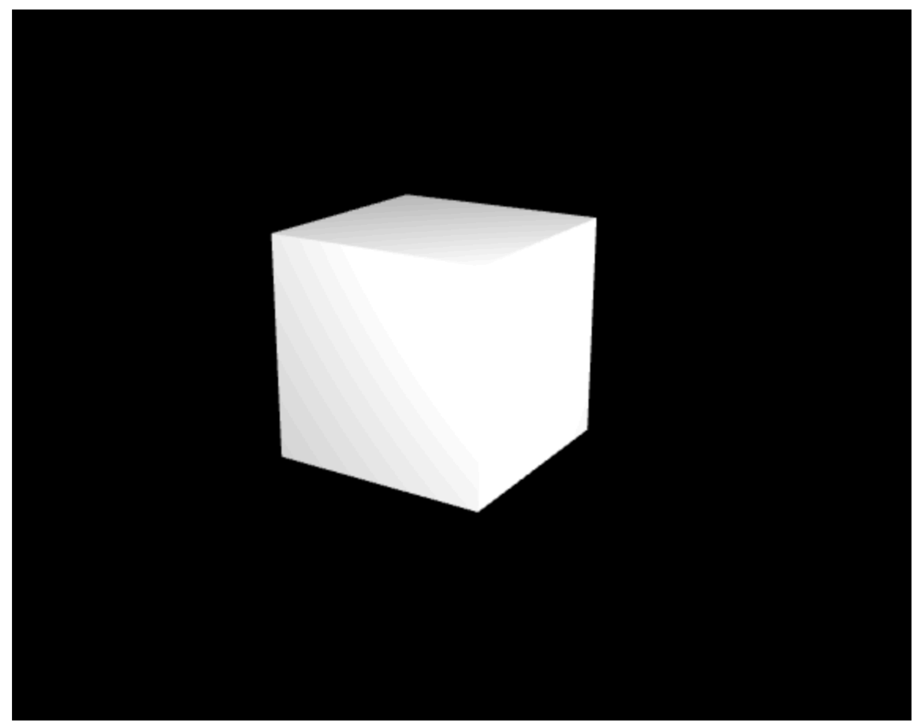
Update the application

- We further need to fill the implemented variables with the correct values.

```
const viewWorldPositionLocation = gl.getUniformLocation(program,
            "uViewWorldPosition");
...
gl.uniform3fv(viewWorldPositionLocation, camera);
```

- Define the necessary `camera` variable and use the values from the projection matrix.

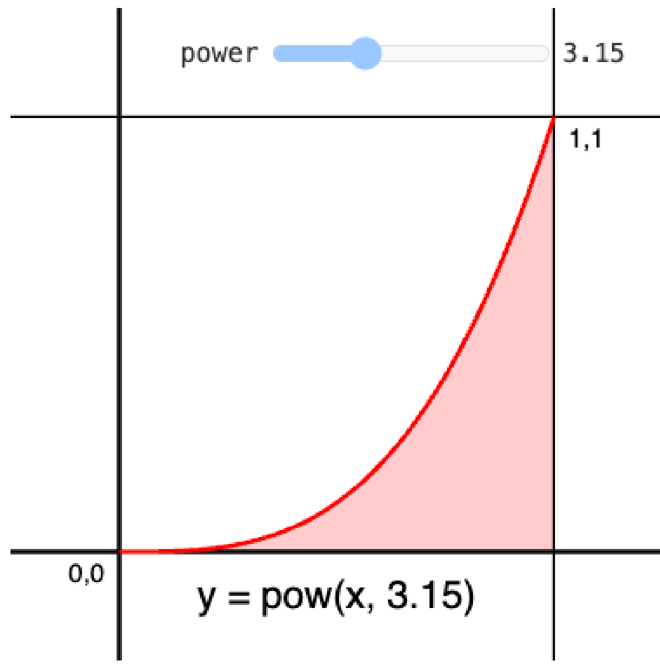
Result: a bright cube



Maybe a little bit too bright 🤷

Fixing the brightness

We can fix the brightness by raising the dot-product result to a power.



Interactive at webgl2fundamentals.org

Adding this power factor to the shader

We add another variable as uniform to the **fragment** shader.

```
uniform float uShininess;
// in the main()
    float light = pointLight;
    float specular = 0.0;
    if (light > 0.0) {
        specular = pow(dot(normal, halfVector), uShininess);
    }
```

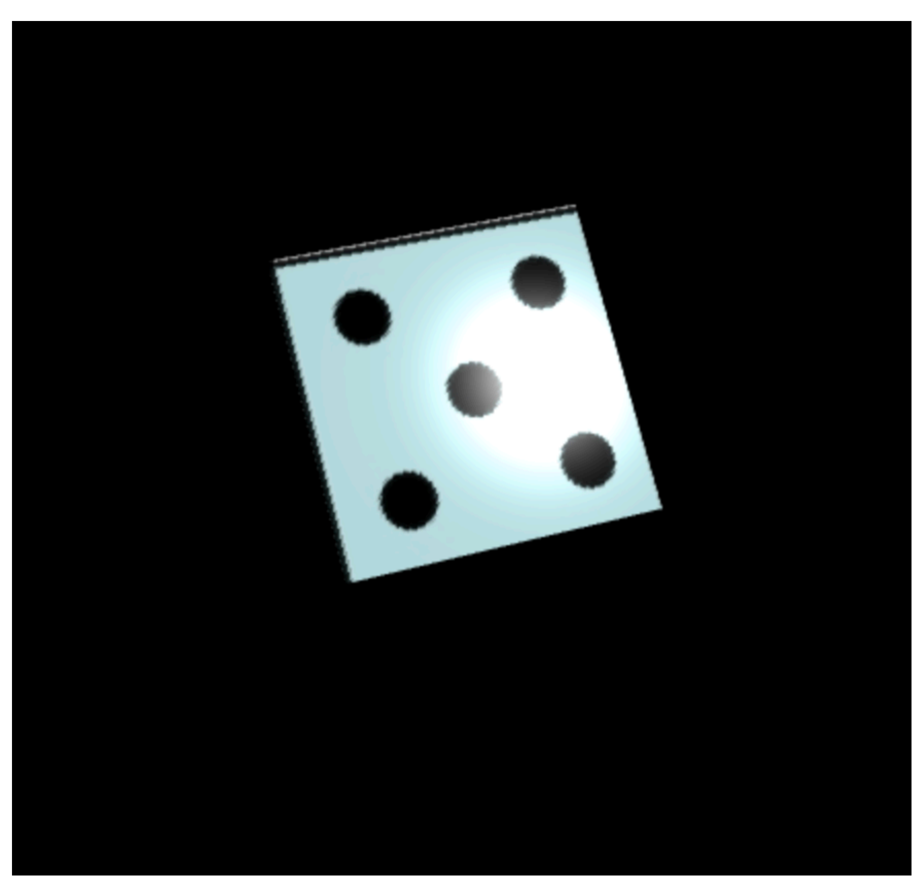
Shiniess parameter

📋 We set the value as usual in the script:

```
const shininessLocation = gl.getUniformLocation(program,
            "uShininess");
gl.uniform1f(shininessLocation, shininess);
```

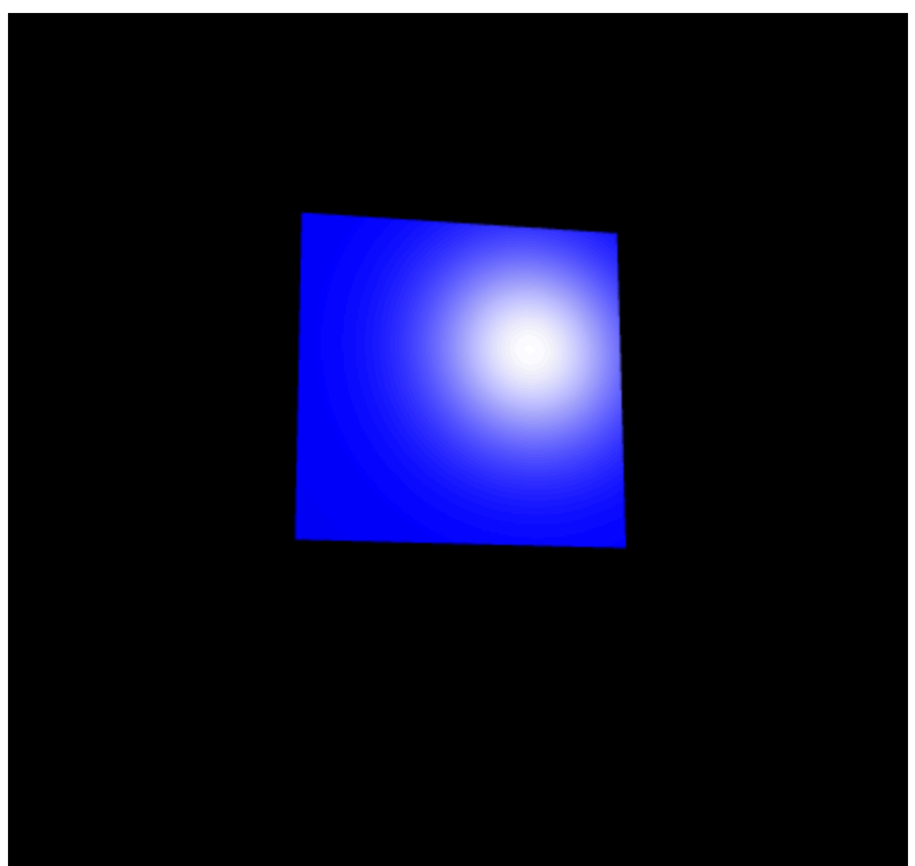
⭕ Add the variable `shininess` and another slider to adjust the value. Find out an appropriate value range.

Result: Shading with specular highlights



Color

In order to better see the highlights, we can add color to the lights - simply by multiplication.



Task

Add colors to the lights and specular highlights.