Report 3

Created	@October 21, 2022 10:57 AM
Class	Graphics
Semester	

Preface

The code in the report, and the code on the actual project differ slightly. Reasons being is that I omitted some code that would of made the blocks here huge, but also because I redid the project to get a better understanding of the steps and while also cleaning up the actual code presented. Lastly, I might edit code before turn in and that may or may not be reflected in the report. Just know these concepts were applied one way or another.

Problem

The assignment given was to recreate this image by implementing the Phong shader model.
We were given much source code but we needed to code in the missing parts that were in `phong.vs` , `phong.frag` , `main.cpp` and `camera.h`.

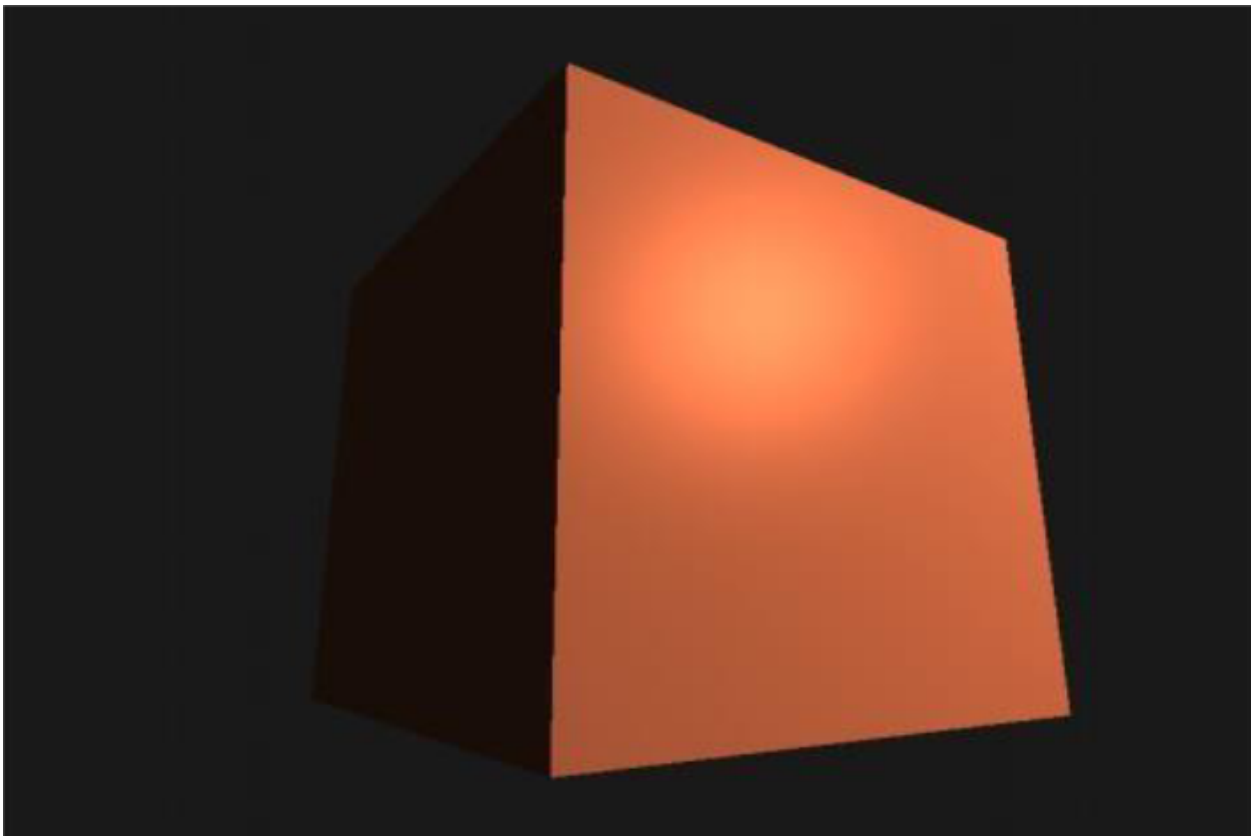


Image to be reproduced

Method

The two shaders are `phong.vs` and `phong.frag` which stand for vertex shader and fragment shader respectively. The vertex shader is where the cube's vertex data shoved into and then is transformed into normalized coordinates. From here, other shaders (that we are not interact with) are transforming those points into the geometry we specified. After a geometry and rasterization shader is used to draw said shapes, the fragment shader is then used in order to apply the Phong shading model. That fact is important for if we applied the shading in the vertex shader, we would not get Phong shading, but instead Gouraud shading. This is due to the fact that the vertex shader has fewer vertices to work with bringing down the cost of the computation, however a poorer result is reached. Therefore, the shading is done in the fragment shader where the geometry has already been rasterized therefore producing more vertices for computation. This is where ambient, diffuse and specular lighting are applied to that geometry we defined, again at a more expensive cost, but a nicer looking result . From a fixed point in space, light is shone onto the cube and we apply those different lighting elements.

An important take away from the `phong.vs` and `phong.frag` shaders are that they communicate with each other. Through a use of keywords like `in` and `out` the shaders can indicate variables that will talk amongst each other. `in` for any variables that are from external sources. `out` for any variables that will be shipped out of that shader into another.

Implementation

To implement the Phong shader model, a series of steps had to be taken.

Step 1

Our first hurdle was get a square into the screen. This square would be one side cube we could see and would be made possible once we set `gl_Position` correctly. Initially `phong.vs` would look like this:

```
#version 330 core
layout (location = 0) in vec3 position;
layout (location = 1) in vec3 normal;

out vec3 Normal;
out vec3 FragPos;

// Ant testing
out vec4 vColor;

uniform mat4 model;
uniform mat4 view;
uniform mat4 projection;

void main()
{
    // TODO: Your code here
    // Remember to set gl_Position (correctly) or you will get a black screen...
    gl_Position = vec4(position, 1.0);
    vColor = vec4(0.0, 0.0, 0.5, 1.0);

}
```

By setting `gl_Position` to a vec4, and handing it position, a vec3, and an extra float 1.0, we were able to set the position. In my case, I played with the color and had it set in the

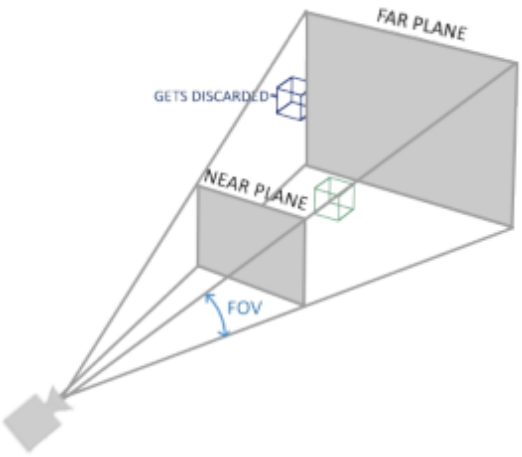
vertex shader and passed it along to the fragment shader testing `in`s and `out`s and their functionality.

```
#version 330 core
out vec4 color;

in vec3 FragPos;
in vec3 Normal;

uniform vec3 lightPos;
uniform vec3 viewPos;
uniform vec3 lightColor;
uniform vec3 objectColor;

void main() {
  // TODO: Replace with your code...
  // If gl_Position was set correctly, this gives a totally red cube
  //color = vec4(0.2, 0.5, 0.2, 1.0f);
  color  = vColor;
}
```

This resulted in the square shown in Figure 1

Step 2

This step would be adding the projection matrix. The view and model matrix are already given to us by default. The projection matrix is not and we have to tell is what to actual project for the viewport matrix to work properly. This is done in `main.cpp` and is done around line 200. Because we are suppose to recreate a perspective image, the projection must be of type `glm::perspective(fov,aspect,near,far)`. Fov will be the angle of the frustrum, how wide the camera can see. The aspect is the aspect ratio that is set from the viewport width divided by the viewport height. Lastly, near and far are two plane that are calculated by the by the geometry of the fov and aspect. These values - with near being close to zero and far being towards an arbitrary length away we'd want to see - are set to allow a view box and to constrain said view.



From Learn openGL - Graphics Programming

The actual code for this looks like:

```
...
glm::mat4 projection;
// TODO: set up the project matrix
projection = glm::perspective(glm::radians(80.0f), (float)WIDTH/(float)HEIGHT, 0.1f, 100.0f);
...
```

This is the last matrix we needed to complete in the transformation pipeline.

Step 3

Up next will be setting up the camera.This is done in `camera.h` and accomplished by setting up a lookAt matrix. Specifically:

```
glm::mat4 GetViewMatrix() {
    return glm::lookAt(Position, Position + Front, Up);
  }
```

The parameters for lookAt take the following form: `glm::lookAt(posVec3, targetVec3,upVec3)`

Each argument will be of type `vec3` and as follows, `posVec3` is the position of the camera, `targetVec3` will be be the referece being looked towards and `upVec3` is just the vector that signifies the up direction to the camera. Sort of irrelevant here, but super useful for a scene where ‘up’ is well defined. This however still returns the red square.

Step 4

This is the step that allows us to actually see the effort put in and see how the whole thing works together. In the vertex shader, we will now redefine the `gl_position` as so:

```
gl_Position = projection * view * model * vec4(position, 1.0);
```

Here, we multiply the 4 matrices, reading right to left, projection * view * model * local gives the `gl_position` matrix. This will complete the graphics pipeline and allow us to move around move the camera to view our shape. See figure 2.

As of currently, the shape has no defined edges, and is only a solid color. This will change in the up coming steps.

Step 5

Here, we just apply the color of the reference image to the color of the cube. This is done in the frag shader by making `color` set to:

```
color = vec4(lightColor * objectColor, 1.0f);
```

To view the coral color, see figure 3. As seen, we can sort of make a cube out as we are viewing it from the side and above.

Step 6

Set up is now over, and we can actually start on shading. This step will cover ambient lighting and normals of the plane in order to advance in step 7. Ambient lighting is pretty simple as it is just the intensity of the light color toned down by a percentage and multiplied across the object. In the fragment shader, it looks like this:

```
void main()
{
    // TODO: Replace with your code...
    // If gl_Position was set correctly, this gives a totally red cube
    //color = vec4(vec3(1.f,0.f,0.f), 1.0f);

  // ambient lighting
  float ambientStr = 0.1f;
  vec3 ambient = ambientStr * lightColor;
```

```
  // results
  vec3 result  = ambient * objectColor;
  color = vec4(result,1.0f);
  //color = vec4(lightColor * objectColor, 1.0f);
}
```

This produced a a dark cube. See figure 4.

For the next sub step, we have to set up the diffuse and specular lighting. To do this, we must edit the vertext shader. Specifically:

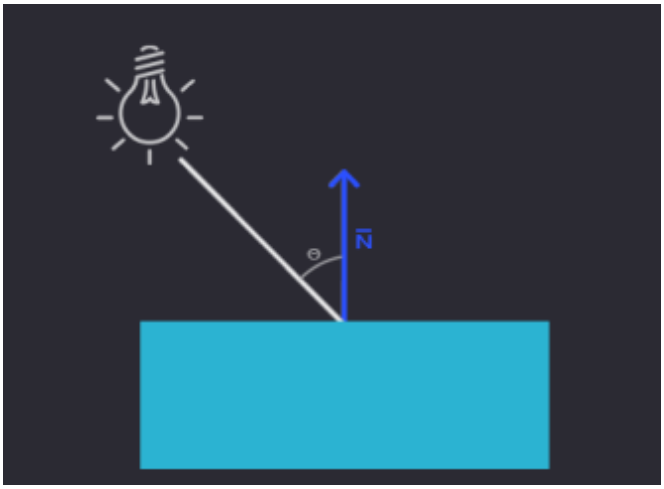
```
layout (location = 1) in vec3 normal;
out vec3 Normal;
out vec3 FragPos;

void main(){
gl_Position = projection * view * model * vec4(position, 1.0);
FragPos = vec3(model * vec4(position, 1.0));
Normal = nomral;
}
```

This does not change the image visually, but what is going on under the hood will be explained starting from the bottom. `Normal = normal;` just allows a `vec3` called `Normal` to be the the normal vector for any given vertex. This is given by `normal` of type `vec3` defined by the metadata location of `location = 1`. More specifically, the normals are already defined for us in the `main.cpp` under `GLfoat vertices[]` however they are just positioned in such a way to allow location 1 to be the normals we need.

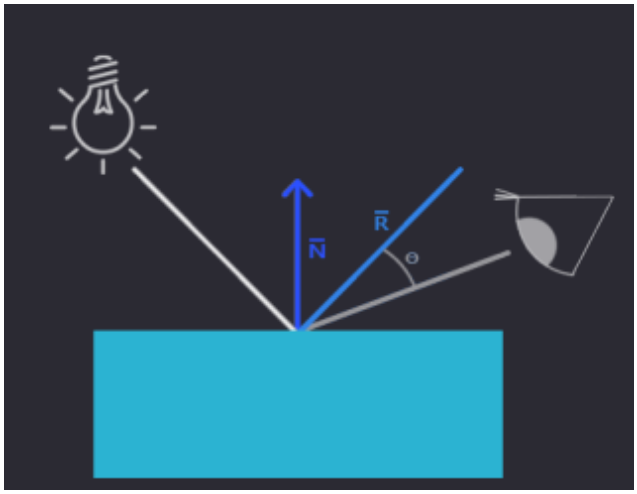
Next is the `FragPos`. This is set equal to a `vec3` that is is the product of the `model` matrix and the vertex positions I.E `vec4(position, 1.0)`. The reason for this is to get the lighting calculation each time the camera changes. With this done, we can move to the final step.

Step 7

Here we will add diffuse lighting and specular lighting. Diffuse lighting is the lighting that allows a matte color on the object and allows some shading be occluded parts of the cube. The light ray itself is not important, just the angle at watch the normal sit respective to the angle the light is coming in at. A larger angle means a darker surface.



Specular lighting is the lighting that allows the reflection of the light to hit the view or camera. This is achieved through having a light ray reflect back at an equal angle of attack and calculating the angle between the camera and the reflection.



The code for this is here in the fragment shader:

```
//diffuse
vec3 norm = normalize(Normal);
vec3 lightDir = normalize(lightPos - FragPos);
float diff = max(dot(norm, lightDir),0.0);
vec3 diffuse = diff * lightColor;

//specular
float specStr = 0.7;
vec3 viewDir = normalize(viewPos - FragPos);
vec3 reflectDir = reflect(-lightDir, norm);
float spec = pow(max(dot(viewDir, reflectDir), 0.0), 42);
```

```
vec3 specular = specStr * spec * lightColor;

vec3 result = (ambient+diffuse+specular) * objectColor;
color = vec4(result, 1.0);
```

Starting from the top, we have `vec3 norm` which just normalized the `Normal` as unit vectors which just gives the direction without magnitude or position. `vec3 lightDir` once again gives the direction of the difference in `lightPos - FragPos` vectors.

`float diff` will give the dot product of the `norm` and `lightDir` and compare it to see which is the max of `dot(norm, lightDir)` and `0.0`. If any products are < 0, then they will be occluded with 0 light being shone on them producing a blacker face.

Finally, this `diff` value is multiplied by the `lightColor` and is the diffuse vector that will be summed with to the result at the end.

See figure 5 for a combined ambient and diffuse lighting.

For the last and final lighting lets go over the players.

`float specStr` is just the strength of the specular lighting aspect.

`vec3 viewDir` is set to the direction of the normalized difference of vectors `viewPos - FragPos`. This gives just the direction the fragment is viewed from.

`vec3 reflectDir` is the reflection of light given by the `reflect(-lightDir, norm)`. Specifically the equal but opposite angle of incident vector.

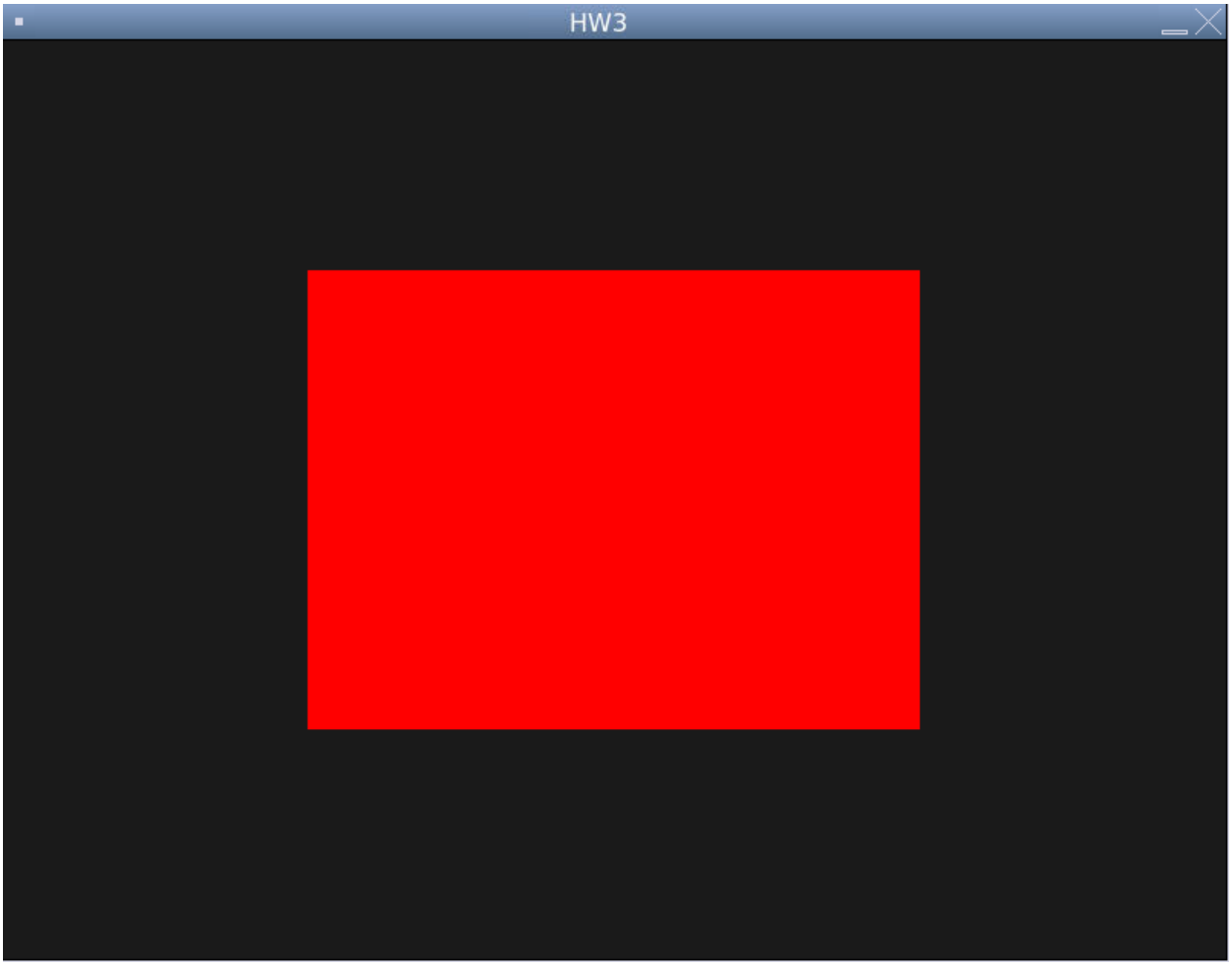
`float spec` is the a value that comes from the dot product of `viewDir` and `reflectDir` making sure that value is not negative, and then raising that power to 42 to get a semi shiny surface. Any negative dot products will be occluded and not give any specular shine.

Lastly, `vec3 specular` is equal to the product of `spectStr*spec*lighColor` and this product is summed with `ambient + diffuse + specular` and then multiplied by the objects color. This result is fed into the `color` vec4 and we get our final result, figure 6.

Result

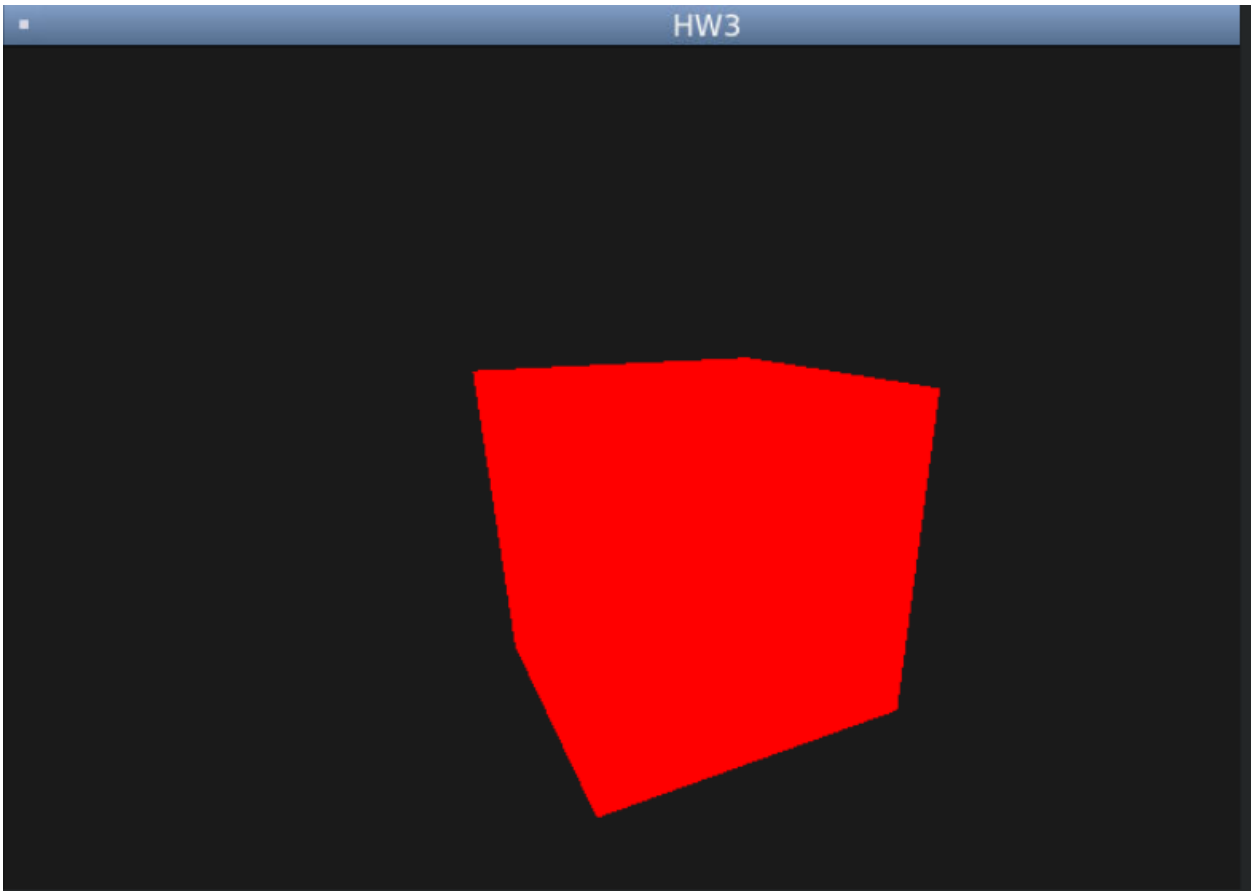
The Phong shading model was final implemented and the results speak for themselves. Moving around the world and being able to see the different reflections and highlights is very interesting and coding by hand (which much starter code) gave a huge understanding of how those concepts are applied. Along with that using GLSL, that is openGL's graphics language gave me a better understanding of the maths behind the screen.

Figure 1



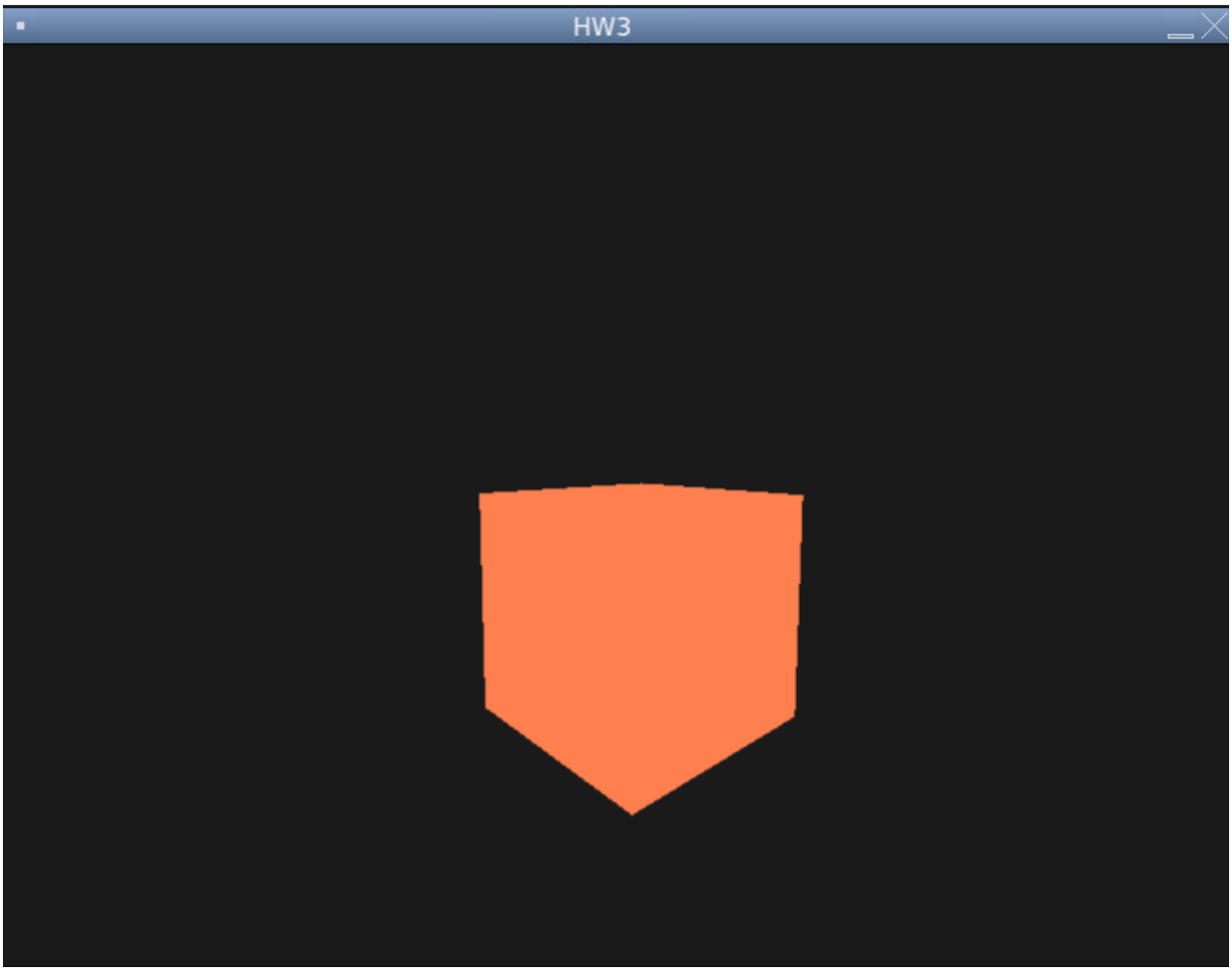
Initial red square

Figure 2



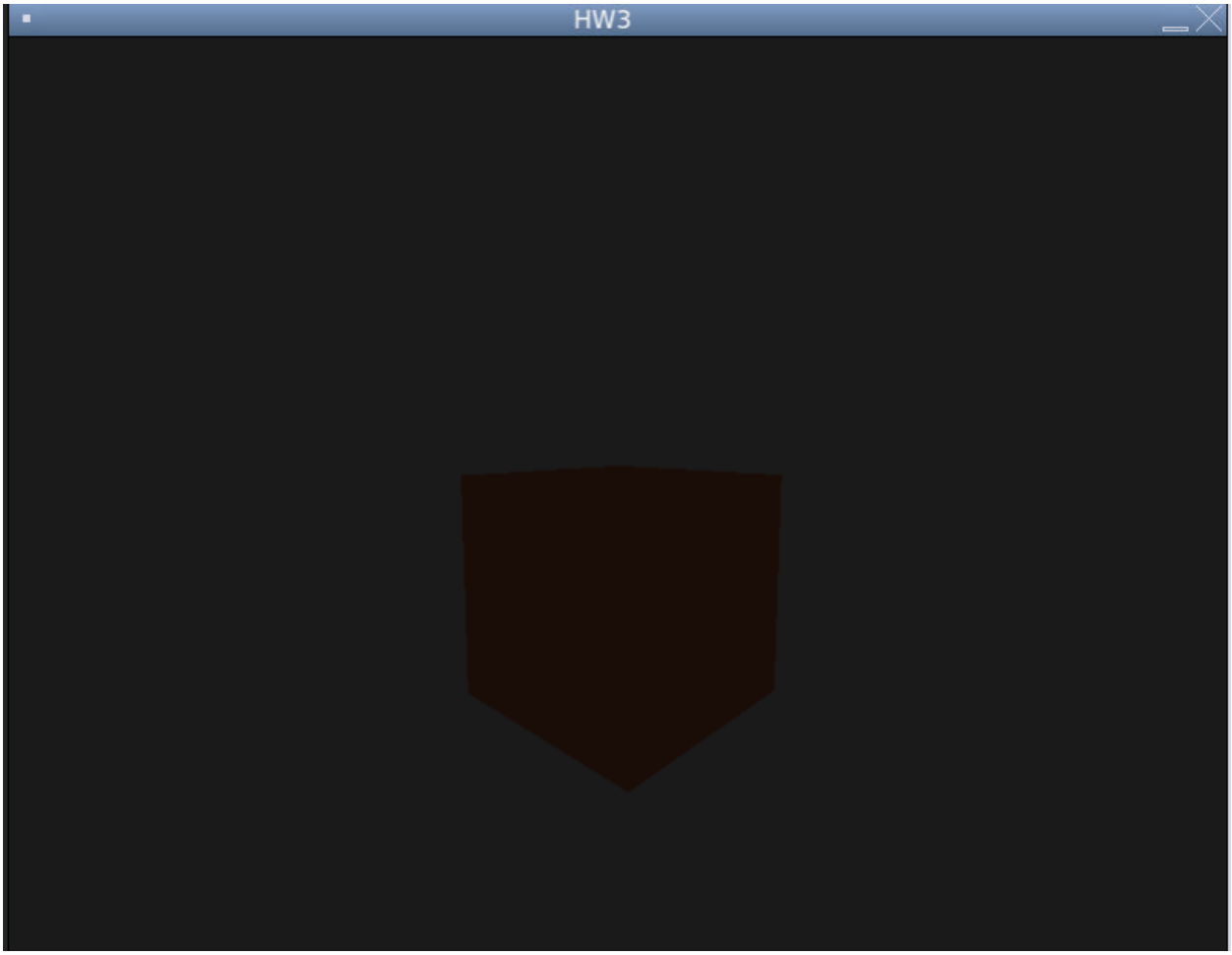
The red square is now a red cube in 3D space.

Figure 3



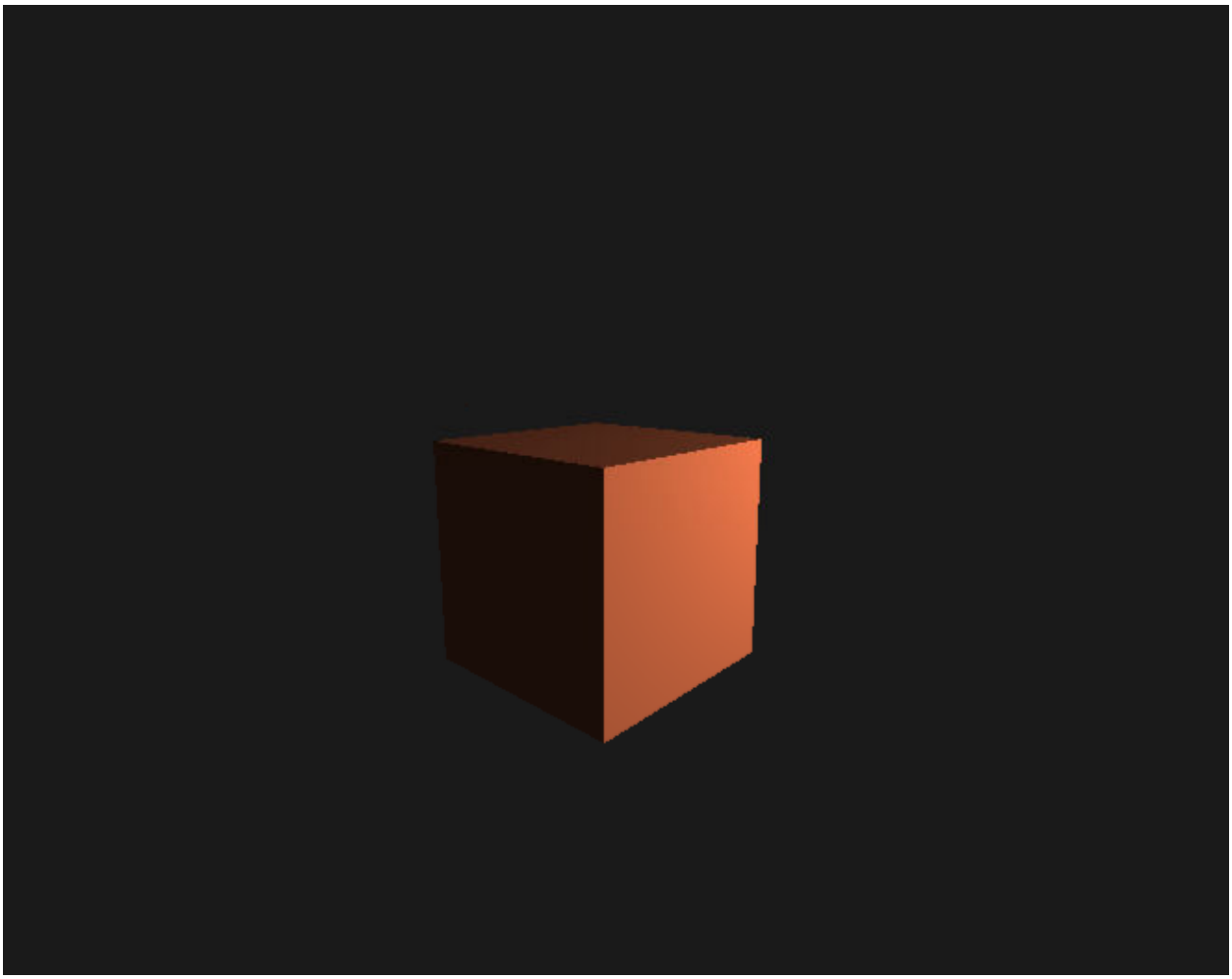
The cube is now a coral color without any shading

Figure 4



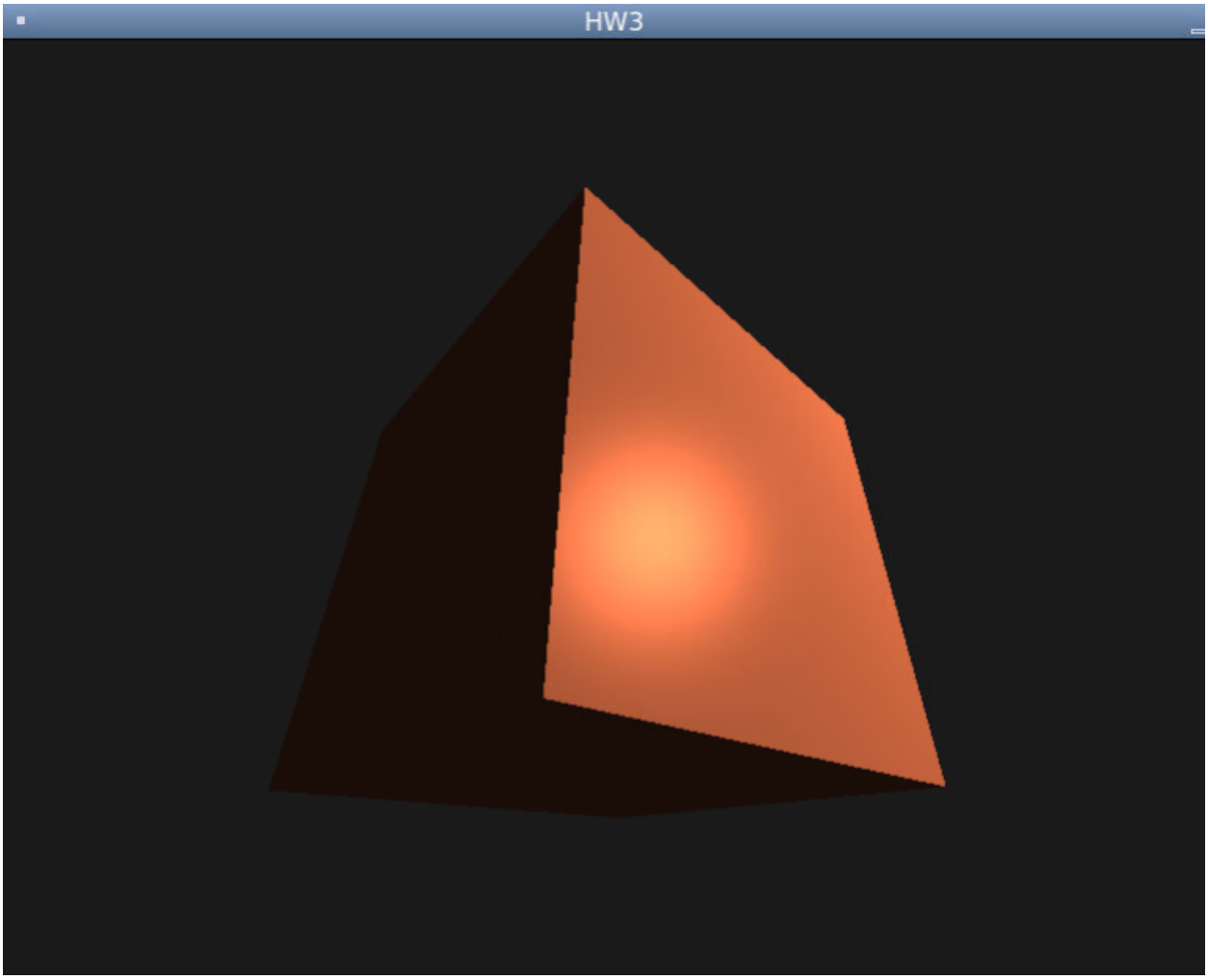
The cube now has ambient lighting applied at around a strength of 0.1 of normal lighting

Figure 5



Here, diffuse lighting is applied. We can see 3 faces and how their normals give off 3 different lightings.

Figure 6



The final, completed Phong shaded cube. With specular lighting added, that shine can move around and disappear depending on angle of view.