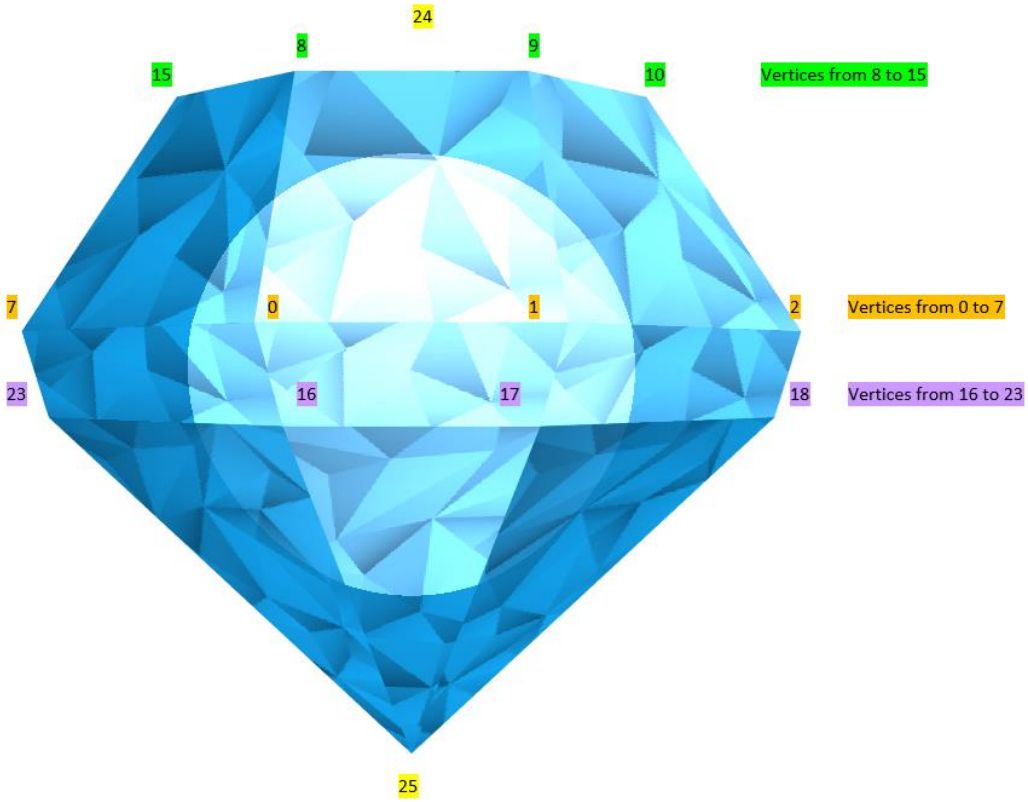
Interactive Graphics

Homework1

Valentina Sisti

1. Choice of the geometry, computation of the vertex normals

The first request that was asked to implement in the Homework was to replace the cube with a more complex shape of 20-30 vertices, so I decided to model the shape of a Diamond, composed by a total of 26 vertices. The resulting shape is the following:



To model the shape, I used the quad function (already in the code template that we were given), to compute the quadrilateral faces of the object, plus a second function, called octagon, that takes in input a starter vertex to start drawing the octagon and the center of the octagon and takes care of drawing the upper part of the object, and the "pyramid" at the bottom. In this way, starting from the starter vertex, the function draws the resulting "octagon-like" shape by drawing the eight triangles of which the octagon is composed (the eight triangles that every couple of adjacent vertices forms with the center they refer to).

The name "octagon", for this function, has come to life when I first started to define a function to draw the upper surface of the diamond, that at the beginning was a flat octagon in fact, but then I decided to raise a little bit the height of the center to give to the result a less-flat effect.

Then I realized that the "pyramid" at the bottom of the object could be constructed using the exact same function by simply changing the starter vertex and the "center vertex", even though it was not actually placed at the same height of the other eight vertices.

Depending on the height of the center, lower or higher than the height of the eight other vertices included, the "octagon" will result to be concave or convex. This detail must be taken into account during the computation of the vertex normals, because to compute the outward normal, the order between the two vectors in the cross product will be opposite in the two cases. For this reason, I introduced a third parameter in the octagon function, that I called normalsOrientationCheck, that is to tell to the function whether the normals had to be computed for a concave or a convex "octagon" (and I have associated a positive integer number for a convex portion, and a negative

integer number for a concave portion). In this way, the normals will be computed correctly in each of the two cases, even if they are managed by the same function.

```
t1 = subtract(vertices[i + 1], vertices[center]);
t2 = subtract(vertices[i], vertices[center]);
if (normalsOrientationCheck >= 0) {
    normal = vec3(cross(t2, t1));
} else {
    normal = vec3(cross(t1, t2));
}
```

2. ModelView and Perspective Projection matrices

To define the ModelView matrix I used the function lookAt, passing it the proper parameters, that are the eye vector (that is the vector with the position of the camera), the up vector, and the at vector (where the camera is looking towards).

To define the Perspective Projection matrix I used, instead, the perspective function, passing it the fovy, the aspect and the near and far distances.

All the parameters of both the ModelView matrix and the Perspective Projection matrix, except for the at and up vectors of the ModelView matrix, that I kept fixed, are controllable in the html file through associated sliders, so that the effect of their changes can be visualized when the script is executed.

3. Directional and Spotlith light sources

A Directional light source is a light source that is modeled to be infinitely far away from the object, so it has the property that its rays shine in a single, uniform direction, that is to say that all its rays are parallel to each other. For this reason, the position of the directional light source it's used to calculate, in the vertex shader, the vector of the direction of the rays of the light, that will be the same for each vertex, and this check is done by setting the w=0.0 in the definition of the position of the directional light source, in the JavaScript file.

The Spotlight, instead, is similar to a point light, with the difference that the light rays are limited to a cone shape, and they do not radiate in all directions. So that the needed result was that only the fragments within the cone of the spotlight are illuminated. To do this computation in practice, it was needed to define the position of the spotlight, the direction of the cone of light and a cutoff angle, that represents the radius of the cone and that is the maximum angle within which the spotlight will illuminate, while the spotlight will not have any effect on the fragments beyond the cutoff value. To do this check, for each fragment I computed the cosine of the angle between spot_D (that is the vector of the direction of the spotlight) and - spot_L (that is, with the negative sign, the vector that points from the light source to the fragment that we are considering), and if this cosine is greater than the cosine of the cutoff angle, then we can say that that fragment is actually hit by the spotlight.

To make the final effect more realistic, I also multiplied by a spotFactor, that is computed in the following way:

$$spotFactor = (\cos(\alpha))^{spotExponent}$$

The final effect, when multiplying by this factor, is that the intensity of the light along a direction at an angle α (and it has to be considered only the α that are less than the cutoff angle) from the center of the spotlight is reduced by this spotFactor. The spotExponent controls how concentrated the light will be in the center of the spotlight.

The key point to obtain a perfect circular shape of the spotlight was to normalize again the two vectors spot_D and - spot_L once they have been passed from the vertex shader. In fact, their magnitude could not have been 1 anymore because of the interpolation between the vertex shader and the fragment shader, and the result, without normalizing again, would have been a spotlight that was not perfectly circular, so that would have been an incorrect result.

```
float cos = max(dot(normalize(spot_D), normalize(-spot_L)), 0.0);
```

Since the amount of illumination from a light source should decrease with increasing distance from the light (and for this reason, since the directional light is effectively at infinite distance, I computed this value only for the spotlight light source), I also taken into account an attenuation factor, that I calculated in the following way:

$$F_{att} = \frac{1}{k_c + k_l d + k_q d^2}$$

where d is the distance to the spotlight, and k_c, k_l, and k_q are, respectively, the constant, linear and quadratic attenuation of the light source.

For the spotlight I added in the html file the sliders to control the cutoff angle, the spotlight exponent and the constant, linear and quadratic attenuation. Then, since the spotlight was requested to be in a fixed position (that I placed in front of the object), the last parameters that I decided to control with sliders are the X, Y, and Z direction of the spotlight.

The material that I defined in the code has the ambient and diffuse components that are typical of a pearl material.

4. Cartoon Shading

Since there are two light sources, the illuminated diffuse color Ci and the shadowed diffuse color Cs needed to be calculated for both, so I defined these parameters as dir_Ci, dir_Cs, spot_Ci and spot_Cs, each computed as specified in the Algorithm for the Simple Cartoon Shade that we were given. For the diffuse components of both the directional and the spotlight, I also calculated the Kd and multiplied it together with the diffuse components of the light and the material.

Then, for both of the directional light and the spotlight, the value of $\mathrm{Max}\{\bar{L} \cdot \bar{n}, 0\}$ is computed and if it's ≥ 0.5, then the corresponding Ci is added to the total cartoon vector, multiplied together with the spotFactor and the spotAttenuation in the case of the spotlight, while in the opposite case, to the total cartoon vector is added the corresponding Cs.

```
//Directional Light, cartoon shading check
if(max(dot(dir_L, N), 0.0) >= 0.5){
  total_cartoon += dir_Ci;
}else{
  total_cartoon += dir_Cs;
}
```

```
//Spotlight cartoon shading check
if(max(dot(spot_L, N), 0.0) >= 0.5){
  float spot_factor = pow(cos, spot_exponent);
  total_cartoon += (spot_Ci)*spot_factor*spotAttenuation;
}else{
  total_cartoon += (spot_Cs)*spotAttenuation;
}
```

For simplicity, I declared a vec4 variable, called total_cartoon, to store the total cartoon shading calculation for both the directional and the spotlight, instead of having them in two separate variables that would have been summed anyway at the end in the final computation of the fragment color.

5. Texture Coordinates

Since the faces of the diamond that I implemented in my code have different shapes and dimensions, the simple choice of (0,0), (0,1), (1,0), (1,1) as the texture coordinates would not have been appropriate, because it would have resulted in a model that would have had the texture “stretched” in the bigger faces and “compressed” in the smaller ones.

So, in order to achieve a satisfying result, I proceeded in the following way:

For the upper octagon, I defined the texture coordinates in a way that they map an octagon in the texture space. Then I defined the other coordinates, for the quadrilateral faces and the triangle faces by mapping quadrilaterals and triangles in the texture space, trying to maintain (but without being too meticulous and accurate) the proportions between them, in order to achieve a result that would not have had the texture compressed in some faces and stretched in others, a thing that would have happened by not paying too much attention in defining the texture coordinates properly and defining instead only the same four coordinates for every face of the object.