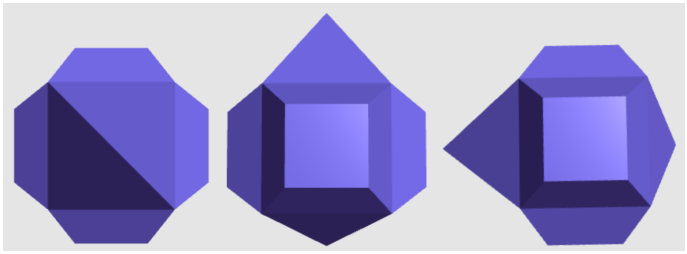
Interactive Graphics - Homework 1

Michela Proietti - 1739846

May 2, 2021

Exercise 1



The solid that has been created is made up by 26 vertices. It consists in a cube that has four truncated square pyramids on the top, bottom, right and left faces, and two pyramids on the front and back faces that have different heights, and therefore they introduce an asymmetry in the geometry, as we can see in the image on the side. There are three kinds of surfaces, namely rectangles, triangles and trapezoids. They have all been treated as quads, and in particular triangles have been treated as degenerated quads, in which the last two vertices coincide, as we can see in the *buildSolid()* function in the JS file. However, in the *quad* function, that takes as input the four vertices of the quad, we handle the various shapes differently. In particular, for triangles we only insert three vertices and the associated normals and texture coordinates in the corresponding arrays. Moreover, we also handle trapezoids differently with respect to rectangles/squares, in order to use the right texture coordinates, and to not have the texture stretched over the different surfaces. In fact, to each vertex we have associated two texture coordinates. For quads, we have simply used the texture coordinates (0,0), (0,1), (1,0) and (1,1) corresponding to the four angles. Then, we have defined the coordinates (0.5, 0.5) for the vertices of the two pyramids, and finally we have defined two more pairs of coordinates for the vertices of the smaller base of the trapezoids. Concerning the normals, they have been computed as the cross product between two consecutive sides of the quad, and they have been normalized and assigned to each vertex in the *quad* function. In image 1, we show the result we get by applying the checkerboard texture. As we can see, the squares in the texture are not stretched to adapt to the surfaces on which they are applied, and there is not the undesired effect of bilinear interpolation in the case of the trapezoids. Also an alternative texture has been created, that is the one left uncommented in the code, and it is shown in image 2.

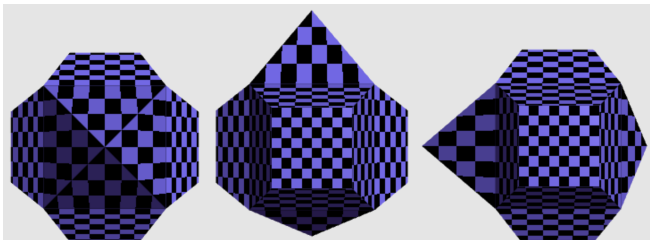


Figure 1: Checkerboard texture

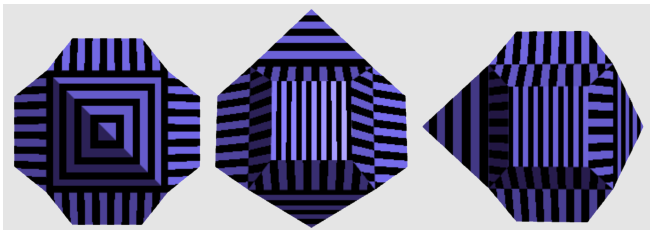


Figure 2: Stripes texture

Exercise 2

The barycenter of the solid is located along the z axis, and it is slightly shifted towards the higher pyramid. It has been computed by calculating the barycenters of all the simple solids that compose the whole geometry and then weighting them by the volume of the corresponding solid and dividing by the volume of the whole structure. Doing this, we have obtained that the barycenter is (0, 0, 0.1). In order to make the solid rotate around it, we performed a translation to bring the origin in the barycenter, we performed the rotation, and then the inverse translation. All this has been done in the *render* function, in which we first define the modelViewMatrix using the *lookAt* function, and then we perform the translation, the rotation, and the inverse translation.

Exercise 3

In perspective projection, we have the center of projection, which is where the viewer is, and the projection plane, that is where the objects are projected. In the JS file, we defined the viewing parameters, namely *near* and *far*, that are respectively the minimum and maximum distance from the viewer, *radius*, which is the initial distance of the viewer from the origin, *theta_view* and *phi*, that are used to change the orientation, *fovy*, which is the angle of opening along the y-axis, and the *aspect* ratio, defined as the ratio between the canvas width and height. Moreover, we have defined the variable *at*, with which we have specified that the viewer is always looking at the origin, and *up*, with which we have specified that the up direction is always in the direction of the y-axis. Then, in the *render* function we have defined the modelViewMatrix using the function *lookAt(viewerPos, at, up)*, and the projectionMatrix using *perspective(fovy, aspect, near, far)*. In the HTML file, instead, we have computed the position as:

$$gl\_Position = uProjectionMatrix \times uModelViewMatrix \times aPosition;$$

because the projectionMatrix makes the transformation from world to camera coordinates, while the modelViewMatrix makes the transformation from camera to clip coordinates. Through some sliders it is possible to change the viewing parameters, and a button let us go back to their original values, which make it possible to entirely see the irregular geometry and the cylinder.

Exercise 4

In order to add the cylinder, we have made a function that builds the arrays with the vertices, the normals, the vertex colors and the texture coordinates. We also defined three functions that let us scale, translate and rotate the object in order to position it in the scene as we like. In our case, it is placed horizontally in front of the object, but in a higher position. The point light source we have initially is placed further along the z axis with respect to the cylinder, but it is in a lower position and slightly shifted to the right. This initial light is always there, and even when the neon lights get turned off it remains there in order to leave the other effects, such as the difference between per vertex and per fragment shading, still visible. A specific function has been implemented in order to define lights, and it treats differently the initial generic light and the three lights in the cylinder. In fact, in order to not make the object too lit, the three neon lights are weaker than the initial one. One of them has been placed at the center of the cylinder, while the other two are shifted towards the two caps. To not make the cylinder rotate with the solid, we have defined new vertex and fragment shaders, to which we sent the original modelViewMartix computed with the *lookAt* function, and in the case of the cylinder we have also an emissive term that gets summed up to the other terms of the illumination equation. Moreover, in the shaders we have set the α component of the color to 0.5, in order to make the cylinder translucent, to let light pass through it. Another difference with the shaders of the irregular geometry is that in this case we do not apply any texture. In the images on the side, we can see how the object is lighter as it is more lit when the neon lights are on, and in particular the upper part is almost white, because it reflects all the light from the four light sources. Furthermore, in the right image, we see that the the right part of the solid is slightly more lit with respect to the left part, because it receives also the light from the initial point light source. Finally, we can see how the cylinder, that initially just lets the light of the point light source pass through, becomes completely white and lit when the three lights inside it are on.

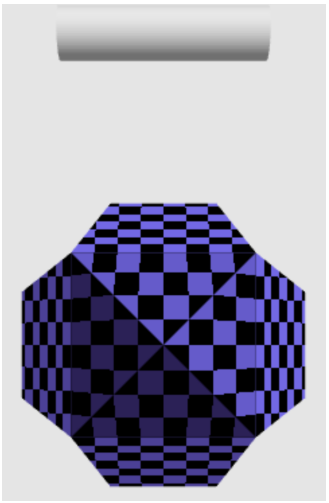


Figure 3: Neon light off

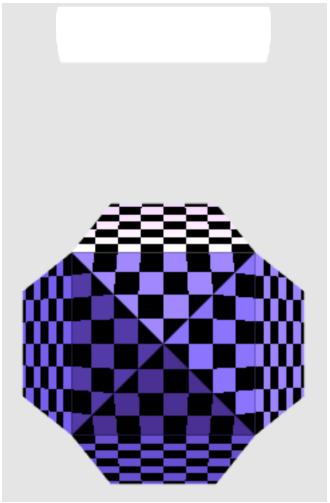


Figure 4: Neon light on

Exercise 5

In order to specify the material properties, we must specify the diffuse, specular and ambient components, so 9 absorption components, plus the shininess coefficient. In particular, we have defined the properties of the material of

the solid in the function *opaqueMaterial* and the shininess has been set to 100 in order to better see the differences between per vertex and per fragment shading models. All these parameters are multiplied by those of the lights, and these products are sent to the shaders and are used to compute the terms in the illumination equation.

Exercise 6

In both models, in the vertex shader we compute the vertex position in eye coordinates, then we compute the vector L, that goes from the vertex position to the light source. Afterwards, since the position of the vertex is expressed in view coordinates and since the position of the viewer in view coordinates is the origin, then we are left with $E = normalize(-pos)$. Finally, in order to transform the vertex normal into eye coordinates, we compute $N = normalize(uNormalMatrix * aNormal.xyz)$ using the normalMatrix, that is similar to the modelViewMatrix, but it is such that we cannot apply to normals transformations that change their shape or orientation. Then, we need to do the lighting computation, that is done in the vertex shader in the per vertex shading model and in the fragment shader in the per fragment shading model.
However, in both cases, what we do is computing the halfway vector, and then the ambient, diffuse and specular terms in the illumination equation, that get summed up and give the resulting color. When we need to apply a texture, the final color of each fragment gets multiplied by *texture(uTextureMap, vTexCoord)*. In orther to switch between the two models, we change the value of a boolean variable through a button, and its value is sent to the shaders as a uniform variable, that is used to understand whether to perform the lighting computation in the vertex or in the fragment shader, without the need of creating two pairs of shaders. As we can see from the figure on the side, the results we get are not so different, because the surfaces are smooth, but we can see that the light is more spread in the per vertex shading than in the per fragment case.

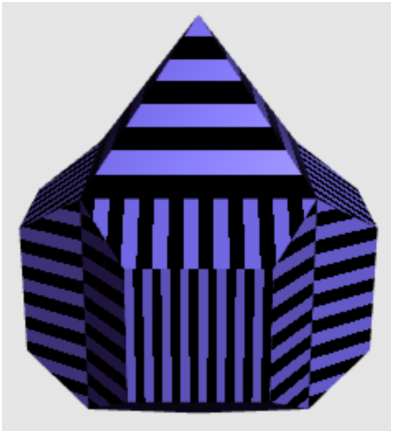


Figure 5: Per vertex shading

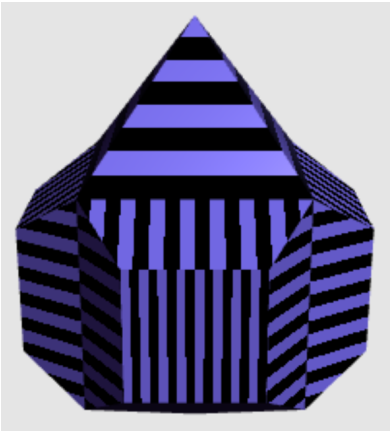


Figure 6: Per fragment shading

Exercise 7

In order to define the data for the bump texture, we have alternated areas in which we have all 1s and areas in which we have all 0s, and we use a function *normalst* which lets us understand if we are at the border between different regions, and therefore we need to apply the displacement along the normals. Then, we convert the data into ubytes for obtaining the texture. Moreover, we have created an array with a tangent for each vertex, in which we have used the edges of each surface, which we also use in order to compute the normals. In order to apply the displacement in the shaders, we have used a uniform variable so that we do not need to define a different pair of shaders. Furthermore, when the bump texture is applied, the neon lights get turned off and we cannot switch anymore between per vertex and per fragment shading. In images 7 and 8 we see two examples of bump textures that have been used (the first one has been commented out in the code as alternative bump texture). In the first one we have created a sort of grater, with raised points, while in the second one we have an alternation of raised and low square areas.

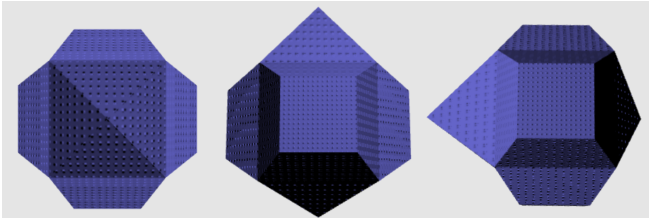


Figure 7: Second bump texture

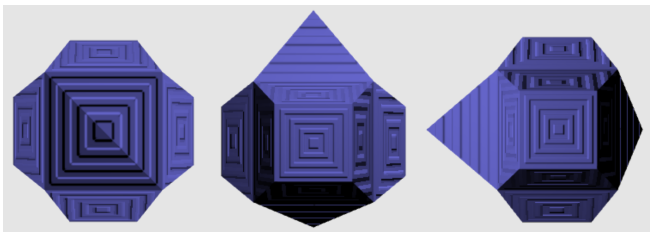


Figure 8: First bump texture