Department of Software Engineering
School of Computer and Telecommunication Engineering

Practices of Computer Graphics

Author:
Domingo Martín

Translation: Domingo Martín

Curso 2019/20

Computer graphics

The great advantage of computer graphics, the possibility of creating virtual worlds without any limit, except those of human capabilities, implies a lot of work, since it is necessary to create a whole series of models or representations of all the things that are intended to be treatable by the computer.

Thus, it is necessary to create models of objects, of the camera, of the interaction of (virtual) light with objects, of movement, etc. Despite the difficulty and complexity, the results obtained usually compensate for the effort.

That is the objective of these practices: to convert the generation of graphics by computer into a satisfactory task, in the sense that it is something that is done "with desire".

For this, we have tried to make the difficulty to appear in a gradual and natural way. Following an incremental structure, in which each practice will be based on the previous one, we propose to start from the first practice, which will serve to take an initial contact, and end up with interaction.

We hope that the proposed practices achieve the objectives and serve to teach the basic concepts of Computer Graphics, and if it can be entertaining, the better.

Domingo Martín Perandrés

Contents

Practice 1

Modelling and rendering of simple 3D objects

1.1 Goals

With this practice we want the student to learn to:

- Create and use data structures that allow to represent simple 3D objects.
- Use OpenGL drawing primitives to draw objects.
- Distinguish between what it is to create a model and what it is to visualize it.

1.2 Development

For the development of this practice, the skeleton of an event-based graphic application is delivered, using Qt 5, with the graphic part made by OpenGL, and being C++ the programming language. The application not only contains the OpenGL initialization code and the capture of key events, but a class structure that allows to represent 3D objects, including axes and tetrahedron objects. It is also implemented a camera that moves on the surface of a sphere by pressing the cursor keys, and page forward and page back keys to simulate a zoom in and zoom out.

The student must study and understand the code that is delivered. Once this is done, you must add the functions that allow you to draw in fill mode and chess mode. In addition, following the hierarchy already defined, you must create the cube class and use an instance that allows you to visualize it when the key 2 is pressed.

Therefore, the following drawing modes will be available at the end:

- Points
- Lines
- Fill

- Chess

To be able to visualize in fill mode, you only have to use triangles as a drawing primitive, `GL_TRIANGLES`, and change the way it is displayed using the `glPolygonMode` instruction, so that the inside is drawn. For chess mode, it is enough to draw in solid mode but alternatively changing the fill color.

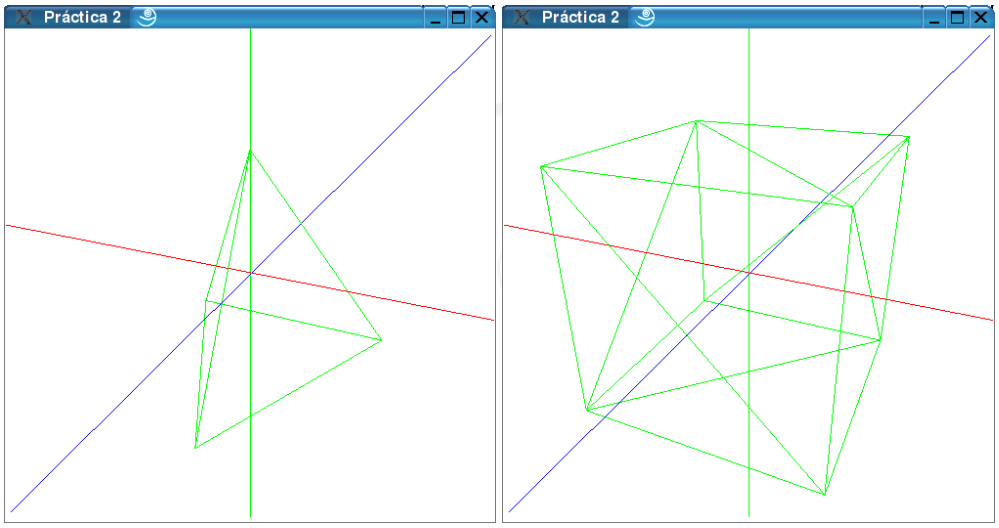
Define the following keys to activate the different modes and objects:

- Key p: to render in point mode
- Key l: to render in line mode
- Key f: to render in fill mode
- Key c: to render in chess mode
- Key 1: to activate the tetrahedron
- Key 2: to activate the cube

1.3 Evaluation

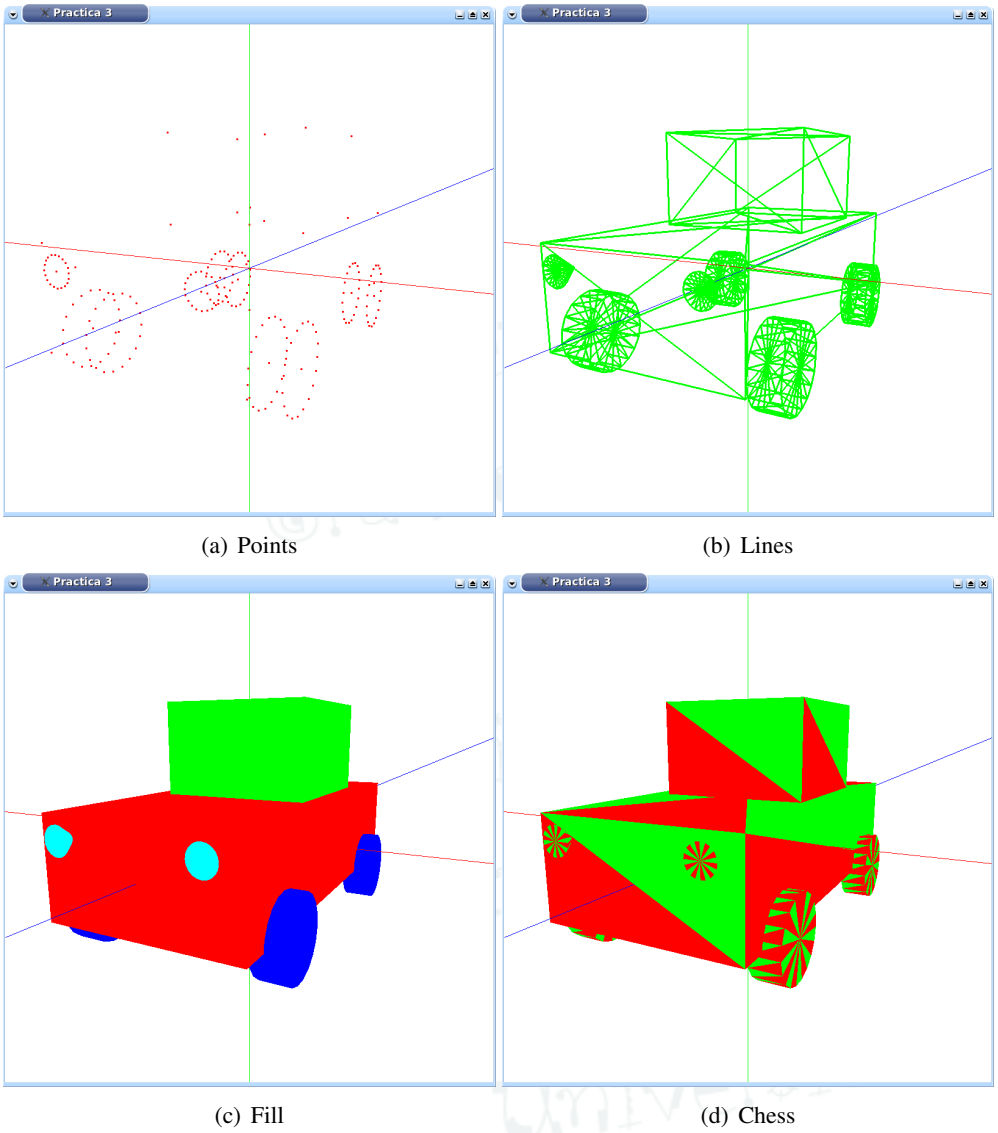
The evaluation of the practice, up to 10 points, will be done as follows:

- Creation of the cube class and the visualization of one instance (6 points)



(a) Tetrahedron (b) Cube

Figure 1.1: Tetrahedron and cube rendered in wire mode.DMP©



(a) Points (b) Lines

(c) Fill (d) Chess

Figure 1.2: Model of a car showing the different rendering modes.DMP©

- Creation of the code that allows to display in fill mode (2 points)
- Creation of the code that allows to display in chess mode (2 points)

1.4 Duration

The practice should be done in 1 session

1.5 Bibliography

- Mark Segal y Kurt Akeley;*The OpenGL Graphics System: A Specification (version 4.1)*; http://www.opengl.org/
- Edward Angel;*Interactive Computer Graphics. A top-down approach with OpenGl*; Addison-Wesley, 2000
- J. Foley, A. van Dam, S. Feiner y J. F. Hughes; *Computer Graphics: Principles And Practice, 2 Edition*; Addison-Wesley,1992
- M. E. Mortenson; *Geometric Modeling*; John Wiley & Sons,1985

Practice 2

PLY and polygonal models

2.1 Goals

With this practice we want the student to learn:

- To load models saved in external files in PLY format (Polygon File Format) and how to display them
- To model solid polygonal objects using simple techniques. In this case, the technique of modeling is based on rotating a profile around an axis, called circular sweeping

2.2 Development

2.2.1 PLY

PLY is a format for storing graphic models saving a list of vertice, a list of polygonal faces and various properties (colors, normals, etc.), that was developed by Greg Turk at Stanford University during the 1990s. For more information consult:

`http://www.dcs.ed.ac.uk/teaching/cs4/www/graphics/Web/ply.html`

This format is very easy to understand, with a header with information about the format of data, followed by the vertices data and the polygons data. This is an example:

```
ply
format ascii 1.0
element vertex 11
property float32 x
property float32 y
property float32 z
element face 1
property list uchar uint vertex_indices
end_header
```

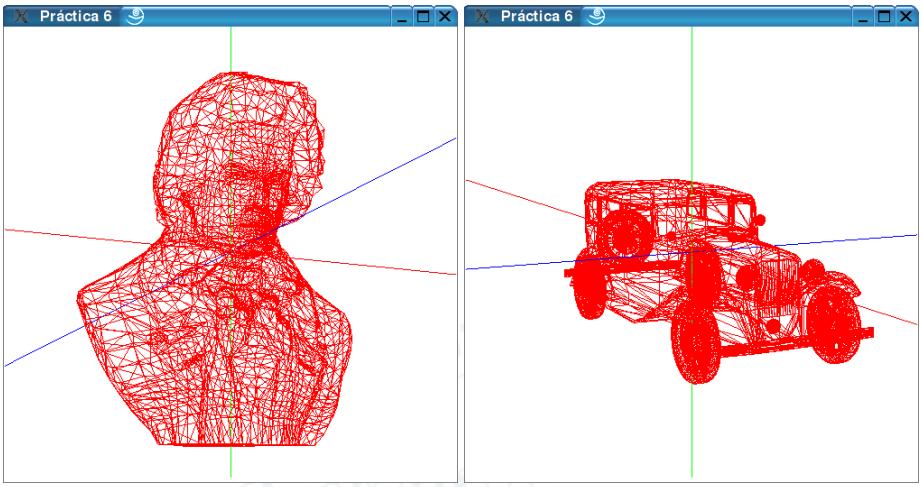


Figure 2.1: Example of PLY objects.DMP©

```
1.0 -1.4 0.0
1.0 -1.1 0.0
0.5 -0.7 0.0
0.4 -0.4 0.0
0.4 0.5 0.0
0.5 0.6 0.0
0.3 0.6 0.0
0.5 0.8 0.0
0.55 1.0 0.0
0.5 1.2 0.0
0.3 1.4 0.0
3 0 1 2
```

The first task in the practice is to load and show objects that are defined in PLY files. To allow this, the code of a basic PLY reader is given, only for objects made with triangles. From the returned raw data, the PLY object will be created using the data structures already defined.

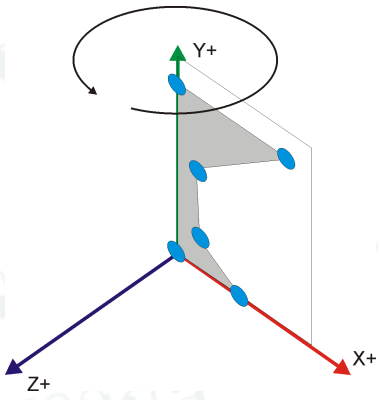


Figure 2.2: Circular sweeping.

2.2.2 Circular sweeping

In the case of the axes, the tetrahedron and the cube, we have been able to create them directly by entering the positions of the vertices and the indices of the triangles, since their number was very small. In general we have to search for other procedures that allow the creation of objects automatically. One of these procedures is the rotation of a profile around an axis, circular sweeping. The idea is very simple: given a curve and a number of division n, the curve is rotated an $\alpha = \frac{2\pi}{n}$ angle about an axis n times. In the figure 2.2 you can see a

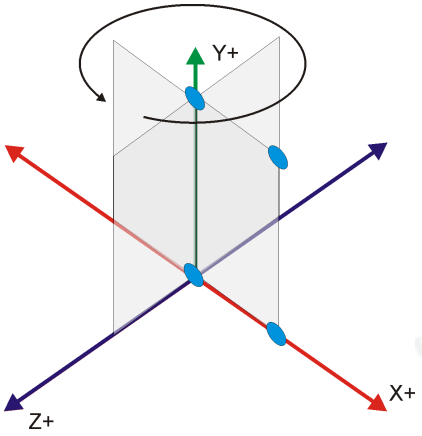


Figure 2.3: Profile for a cylinder

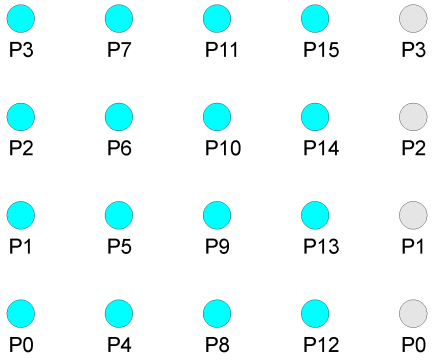


Figure 2.4: Unwrapping of points in a plane

generatrix curve, a polyline, to create a kind of cup.

In our explanation, the points of the generatrix curve or profile are rotated around the y axis, obtaining a series of profiles from which the faces can be created. To produce the rotation of each point of the profile, we use the sine and cosine functions, since it can be observed that what we are really doing is calculating the positions in a circle. For example, given a point with the following coordinates $P(x,y,0)$ (note that the coordinate z is equal to 0, because our generatrix is in that plane), what we want is to rotate it around the y axis. This implies that the y coordinate will not change. Therefore, we are only interested in the value of the x coordinate, which we can take as the radius R of the circumference. To calculate the points of a circle given the radius we use a well known formula:

$$x = R \cdot \cos(\alpha)$$
$$z = -R \cdot \sin(\alpha)$$
$$0 \leq \alpha \leq 2\pi$$

For example, if we want to obtain 3 profiles we only have to divide the full range, 360 degrees, by 3, obtaining the angles 0°, 120° and 240° (calculations are made in radians).

Let's look at a simple example to understand how the object construction is done. We are going to use a profile that allows us to build a cylinder. To do this we will use the 4-point profile shown in the figure 2.3. In addition, we only use 4 divisions to simplify the process.

The process of rotating the original profile allows you to create the other three. All new points will form the geometry of the object. An important detail to note is that points that are in the y axis, when rotated, will produce new points that have the same coordinates. That is, they are different points that occupy the same position. The storing of the points is done sequentially: first the points of the first profile, then the second one, etc. In addition, for each profile we maintain the order of the points, so that we will first store P_0, then P_1, up to P_3, and then will come $P'_0 = P_4$, $P'_1 = P5$, etc. Now we can render the model in point mode.

Once we have saved the geometry we want to create the triangles that represent the

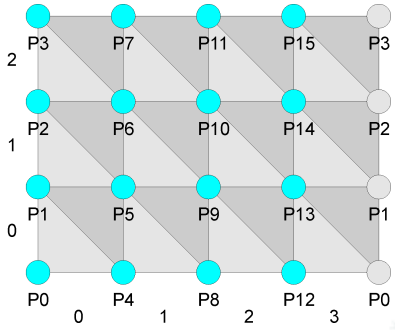


Figure 2.5: Sequential positions of points

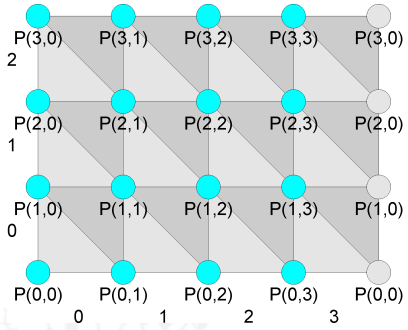


Figure 2.6: 2D positioning of dots

surface. The simplest way is to "forget" the shape of the object in 3 dimensions and create a way in which the important thing is the position of each point with respect to the others. For this we aim to flatten all the profiles on a plane, so that the location of the vertices maintains the relative position. We can see this scheme in the figure 2.4. In order to close the object it is necessary to create the triangles that join the points of the last profile with the points of the first profile. This is indicated by adding a column of gray dots, representing to the same ones of the first profile.

The triangles we want to build are shown in the figure 2.5. It is important to see that we have converted our set of points into a matrix, and therefore, we can treat this configuration in a simpler way using rows and columns. In our case we have 3 rows, from 0 to 2, and 4 columns, from 0 to 3. If we take a look, for each row and column configuration we have to create 2 triangles, which we will call even and odd ones (they coincide with their positional parity as we will see now).

What we are trying is to automate the process of building the triangles, by means of a simple code that is repeated. Therefore, we need to obtain the construction pattern. We will try to deduce it by carrying out, as an example, the generation of the triangles of a couple of positions in the matrix. As we know, a matrix can be traversed by rows and for each row by columns, or vice versa, by columns and for each column by rows. In the implementation, it doesn't care, you just have to be consistent.

For example, our code may be as follows:

```
for (int Row=0;Row<Num_rows;Row++){
  for (int Col=0;Col<Num_cols;Col++){
    // create even face
    // create odd face
  }
}
```

We will create the 2 triangles when Row=0 and Column=0. The even triangle, $Triangle_0$, will be made up of the points P_1, P_0 and P_4, while the odd triangle, $Triangle_1$, will be composed of the points P_4, P_5 and P_1 . An important detail is that we always have to indicate the points following the same direction, in this case, counter clockwise. We could also define

all triangles following the clockwise direction. What we cannot do is to mix both directions, because as we will see, from the vertices we will calculate the visibility of the triangles.

With only 2 triangles we cannot see any pattern. We will therefore create the 2 triangles when Row=0 and Column=1. In this case, the even triangle, $Triangle_2$, will be composed of the points P_5, P_4 and P_8, while the odd one, $Triangle_3$, will be composed of the points P_8, P_9 and P_5. Now you can start to see the patterns.

Let's look at the composition of the two even triangles (only the indexes of the points are indicated): $Triangle_0(1,0,4)$ and $Triangle_2(5,4,8)$. We can appreciate that the indices of the $Triangle_2$ can be obtained from the $Triangle_1$ simply by adding 4, which is the number of points that the profile has. It is easy to verify that the same thing happens when Row = 0 and Column = 2, but what happens when Column is 3? Let's check it out.

In principle the points of the triangles would be the following: $Triangle_6(13,12,16)$ and $Triangle_7(16,17,13)$. But we do not really have points 16 and 17 but we must use points 0 and 1. To do this, we must use the module operator (%), so that the value 16 returns us 0, and the value 17 returns us 1. In this case, it is enough for us to use as a divisor the total number of points generated, which is 16, 4 points times 4 profiles.

We are going to consider another way of dealing with the addressing of the points within the matrix that will make our code easier to read. To do this, we only have to address the points in a matrix way rather than a linear way, as shown in the figure 2.6. They are the same points, stored in the same way, but for convenience we will treat them as if they occupy positions in a matrix. Thus, we see that the point $P(0,0)$ corresponds to the point P_0, the $P(0,1)$ corresponds to the point P_4, or the $P(1,0)$ le corresponds the point P_1. The conversion is very simple: $P(Row,Columa) = P(Column \times Num\_points + Row)$.

This process allows us to create our model by revolution. But we only have the simplest version, which is displayed correctly but does not represent a correct model. The reasons are two: the duplication of the extreme points of the profile and the creation of triangles that have no area (degenerate triangles). These triangles are shown in red color in the figure 2.7.

The more optimized version removes repeated points. To do this, the construction of the object must distinguish between 3 zones: the lower cover, the central zone and the upper cover. In our algorithm we have to reflect this distinction. Thus, to create the points of the central zone we must create a new profile that includes all the original points except the extreme ones, in case that they exist. The points of the covers, 0, 1 or 2, will be saved at the end in the vertices vector. In this way, we know their fixed positions.

Then we will create the faces of the central zone using the same procedure explained previously. Finally we will create the triangles of the covers, if it is necessary. In the case of the covers, we could create triangles, as we have done so far, but we could also use the primitive triangle_fan.

These keys will be added:

- Key 3: to activate the cone
- key 4: to activate the cylinder
- Key 5: to activate the sphere

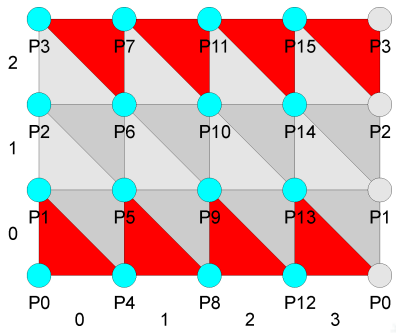


Figure 2.7: Degenerated triangles

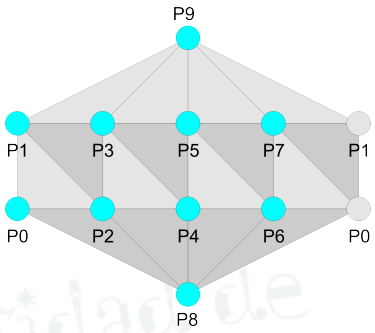


Figure 2.8: Optimized version

- Key 6: to activate the PLY object

2.3 Evaluation

The evaluation of the practice, up to 10 points, will be done as follows:

- Creating a class for PLY objects. Given the data set of the file, the class allows you to enter them according to the data structures already defined (2 points).
- Creating a class to obtain objects by revolution around an axis without containing degenerate triangles and repeated points. (3 points)
- Allow that points in the profile have any direction (1 point)
- Creation of the class to draw a cone. (1 point)
- Creation of the class to draw a cylinder. (1 point)
- Creation of the class to draw a sphere. (1 point)
- Creation of the class that allows to generate a revolution object from a profile defined in a PLY file. (1 point)

2.4 Extras

- To allow that the generatrix curve to be defined on any of the main axis. (2 points)

2.5 Duration

The practice should be done in 3 sessions

2.6 Bibliography

- Mark Segal y Kurt Akeley; *The OpenGL Graphics System: A Specification (version 4.1)*; http://www.opengl.org/
- P. Shirley y S. Marschner; *Fundamentals of Computer Graphics, 3rd Edition*; A K Peters Ltd. 2009.
- J. Vince; *Mathematics for Computer Graphics*; Springer 2006.

Practice 3

Hierarchical models

3.1 Goals

With this practice we want the student to learn to:

- Design hierarchical models of articulated objects.
- Control the animation parameters of the degrees of freedom of hierarchical models using OpenGL.
- Manage and use the OpenGL transformation stack.
- Modify the values of the transformations automatically to produce an animation

3.2 Development

3.2.1 Geometric transformations

We have already seen how to create simple 3D objects. Now we are going to see how more complex objects can be created by using simpler objects and applying geometric transformations to them, which is called hierarchical modeling. Therefore, the first thing we will have to understand are the geometric transformations, the mathematical procedure to make the model change in size, orientation and position, among other possibilities. In order to observe the effect of the transformations we will use the cube.

Given the cube, which has been created in such a way that it has a unit edge size and is centered with respect to the origin, what we are considering is how to make that the cube to be placed in another position, change its size (and in doing so even the shape), and change the orientation. Let's look at three types of transformations in more detail.

The transformation that allows you to move an object from one position to another is called **translation** (see figure 3.1). Since we want our object, the cube, to behave as if it were rigid, the movement that is applied to a vertex must be applied to all. Therefore, we only have to see how to move a point from one position to another and apply the same operation to all points of the model.

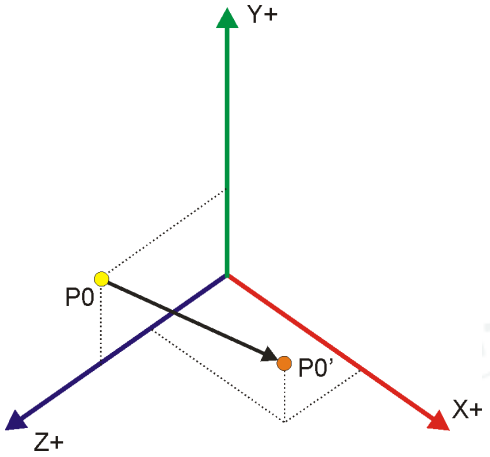


Figure 3.1: Translation

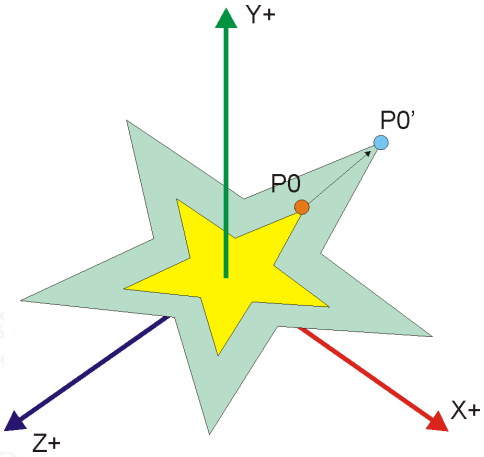


Figure 3.2: Scaling

If we have the point P_0 and the value of the translations $T = (T_x, T_y, T_z)$, the translated position is obtained by adding the offsets to the coordinates of the point: $P_0' = P_0 + T$:

$$\begin{aligned} x' &= x + T_x \\ y' &= y + T_y \\ z' &= z + T_z \end{aligned}$$

The transformation that allows you to change the size is called **scaling** (see figure 3.2). The operation that scales is multiplication. If we have the point P_0 and we have the scaling factors $S = (S_x, S_y, S_z)$, the scaled position is obtained by multiplying the scale factors by the coordinates of the point: $P_0' = P_0 \cdot S$:

$$\begin{aligned} x' &= x \cdot S_x \\ y' &= y \cdot S_y \\ z' &= z \cdot S_z \end{aligned}$$

It is important to note the following:

- If the scale factors are greater than 1 the object is enlarged.
- If the scale factors are less than 1 and greater than 0, the object is reduced.
- If the scale factors are 0, the object collapses to the point $(0,0,0)$.
- If the factors are negative, a reflection is produced, and the object is reversed.
- If the factors are all the same, homogeneous scaling occurs.

- If the factors are different, a heterogeneous escalation occurs.
- The scaling is done in relation to the origin of coordinates. That is, when scaling the point, the result will be on the line that joins the point with the origin. If the position with respect to which you want to perform the scaling is not the origin, you must first convert that position to the origin. We will see the process later.

The transformation that allows changing the orientation is called **rotation**. Since we are in 3 dimensions, we have the possibility to rotate the object with respect to infinite axis, but we are interested in the three main axes, having a different formulation for rotation with respect to the x axis (see figure 3.3), the y axis (see figure 3.4), and to the axis z (see figure 3.5). In this case, the parameter that defines a rotation is the angle we want to rotate. The formulas to perform the rotations are the following:

$$\begin{array}{lll} x' = x & x' = x \cdot \cos(\alpha) + z \cdot \sin(\alpha) & x' = x \cdot \cos(\alpha) - y \cdot \sin(\alpha) \\ y' = y \cdot \cos(\alpha) - z \cdot \sin(\alpha) & y' = y & y' = x \cdot \sin(\alpha) + y \cdot \cos(\alpha) \\ z' = y \cdot \sin(\alpha) + z \cdot \cos(\alpha) & z' = -x \cdot \sin(\alpha) + z \cdot \cos(\alpha) & z' = z \end{array}$$

It is important to note that the coordinate that is equal to the axis on which it is rotating does not change. Also, that like the scaling, the rotations are made having as a pivot the origin. If it is desired to rotate with respect to another pivot, it must first be converted into the origin.

Therefore, we already have the formulas that allow us to make the different transformations. Now we are going to see that, in general, objects undergo several transformations to achieve the final result. For example, it is very normal to apply a scaling plus a rotation plus a translation (instantiation transformation).

Other cases are scaling and rotations with respect to points that are not the origin: the procedure consists in converting the point with respect to which it is scaled or rotated to the origin with a translation, then the scaling or rotation is applied, and finally, the translation would be undone (see figure 3.6).

From these examples that we have commented, the important thing is to see that we have to produce a sequence of transformations to obtain the result. For example, in the instantiation transformation we would have to apply the following steps in the following order:

1. $P' = S(P)$
2. $P'' = R(P')$
3. $P''' = T(P'')$

The problem is that every time we apply a transformation, it must be applied to all points of the model: if our model consists of 1 million vertices, and there are n transformations, then we will have to apply n millions of transformations

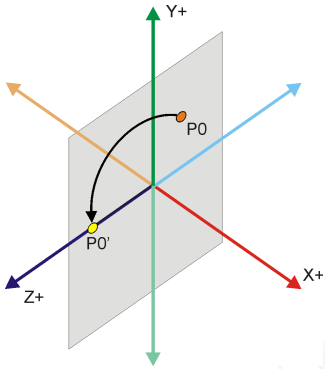


Figure 3.3: Rotation around x axis

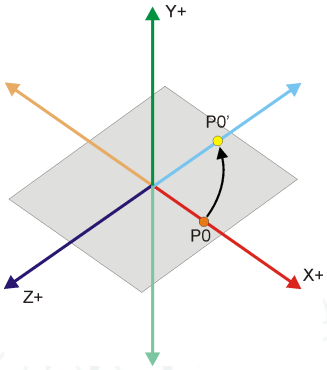


Figure 3.4: Rotation around y axis

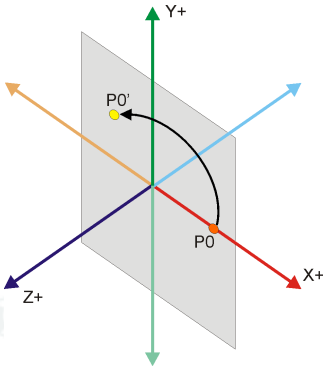


Figure 3.5: Rotation around z axis

The solution is to try to build a transformation that is capable of doing the same but with only one application. That is, we are looking for a T transformation such that $P''' = T(P)$. Is this possible? Yes, we just have to see the formulas with which we have defined the transformations and make a substitution. We are going to present a simpler case to see it: a scalation followed by a translation. For scaling we have $P' = T_s(P)$ and for translation we have $P'' = T_t(P')$. Let's use the formulas: $P' = P \cdot T_s$ and $P'' = P' + T_t$. Therefore, substituting P' in the second equation we will have $P'' = P \cdot S + T$. If instead of first doing the scaling and then the translation we reversed the order we would have $P'' = (P + T) \cdot S$.

If we try to compose several transformations using the explicit equations, this becomes in a very complex algorithm because we must be able to compose any number of transformations in any order. So, we have to find another solution: to rewrite the transformations as matrices. By doing that, to transform a vertex is to convert it to a 1D vector and multiply it with the matrix that represents the transformations. This will also have the effect that to combine transformations is as easy as multiply matrices! Let's see how to do this rewriting.

Since we are going to make the transformations in the 3D space and that the points are defined in the 3D space, we are going to look for our matrices in the 3D space. Therefore, what we want is a 3×3 matrix that, when multiplying by the column vector of a point gives us the result:

$$\begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a & d & g \\ b & e & h \\ c & f & i \end{bmatrix} \begin{bmatrix} x \\ y \\ z \end{bmatrix} \quad \rightarrow \quad \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix} = \begin{bmatrix} a \cdot x + d \cdot y + g \cdot z \\ b \cdot x + e \cdot y + h \cdot z \\ c \cdot x + f \cdot y + i \cdot z \end{bmatrix}$$

Now we have to consider the transformations and match both equations to be able to find the values of the coefficients: Let's see how it is done with the scaling:

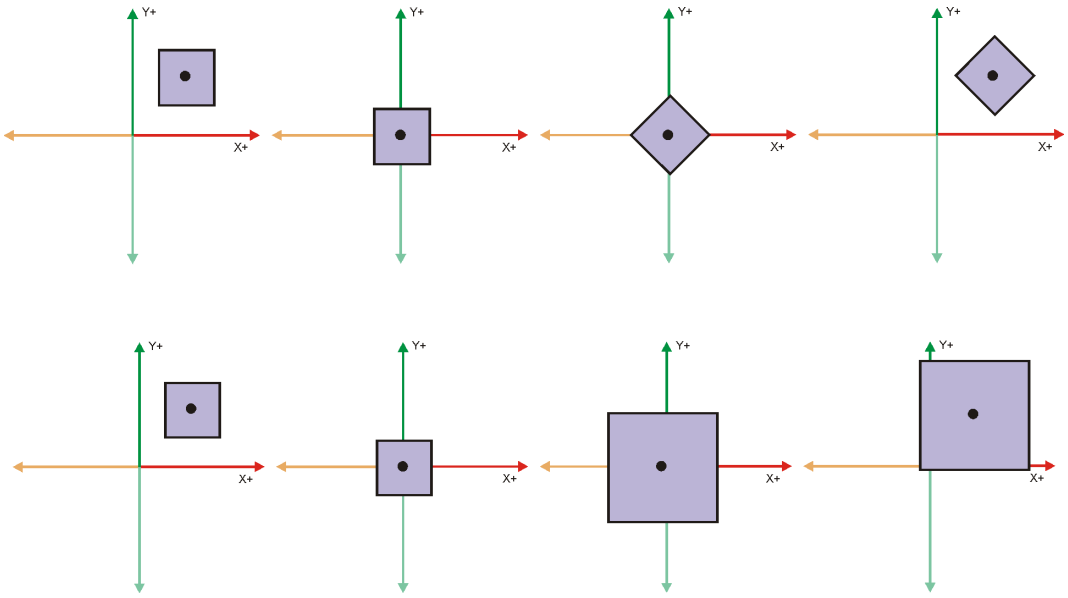


Figure 3.6: Rotation and scaling with respect to a pivot point that is not the origin

$$\begin{array}{lcll} x' = x \cdot S_x & & x' = a \cdot x + d \cdot y + g \cdot z & x \cdot S_x = a \cdot x + d \cdot y + g \cdot z \\ y' = y \cdot S_y & = & y' = b \cdot x + e \cdot y + h \cdot z & \rightarrow \quad y \cdot S_y = b \cdot x + e \cdot y + h \cdot z \\ z' = z \cdot S_z & & z' = c \cdot x + f \cdot y + i \cdot z & z \cdot S_z = c \cdot x + f \cdot y + i \cdot z \end{array}$$

It is easy to see that the coefficients that match both sides of the equations are the following:

$$\begin{bmatrix} S_x & 0 & 0 \\ 0 & S_y & 0 \\ 0 & 0 & S_z \end{bmatrix}$$

We can apply the same process for the rotations and obtain the appropriate coefficients, but with the case of the translations we find a problem since the matrix is as follows:

$$\begin{bmatrix} 1 + \frac{T_x}{x} & 0 & 0 \\ 0 & 1 + \frac{T_y}{y} & 0 \\ 0 & 0 & 1 + \frac{T_z}{z} \end{bmatrix}$$

The problem is that the coefficients of the matrix depend on the coordinates of the point that we want to transform with the matrix. This implies that if we have 1 million points we have to create 1 million of matrices which does not make sense in our case.

To solve this, we are going to expand the dimension of the space in which we are going to make the transformations, going from one of 3×3 to another of 4×4 or projective space. It is also said that we are going to perform the calculations in homogeneous coordinates.

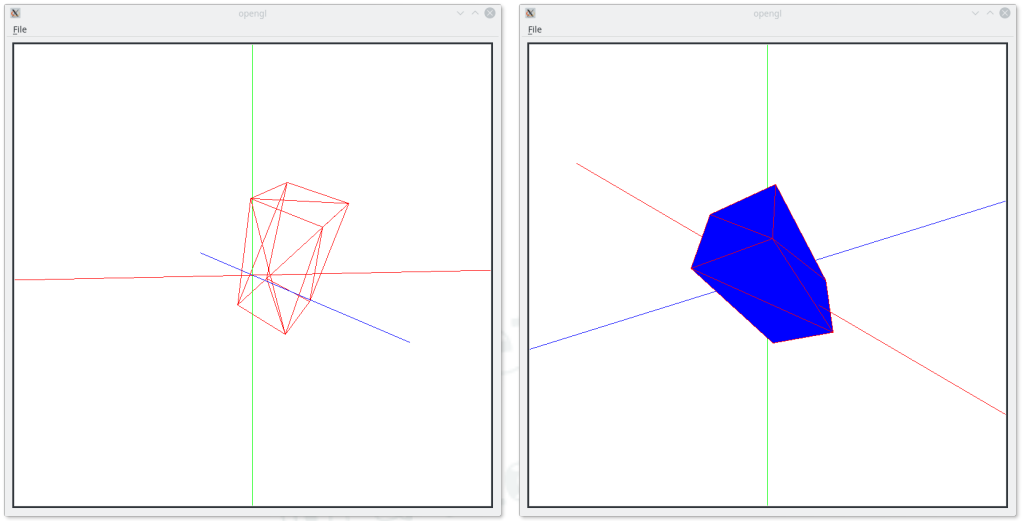


Figure 3.7: Example of applying transformations to a cube

How do we pass a point in 3D coordinates to another in 4D? We will use the following formulation: $x' = x \cdot w$, $y' = y \cdot w$ and $z' = z \cdot w$, for any $w \neq 0$. In this case, we have the following: $(x, y, z) \rightarrow (x \cdot w, y \cdot w, z \cdot w, w)$. The transition from 4D to 3D is done with the inverse operation: $(x, y, z, w) \rightarrow (\frac{x}{w}, \frac{y}{w}, \frac{z}{w}, 1)$.

Keep in mind that each point in 3D coordinates corresponds to infinite points (a line) in 4D, when the value of w varies. We can simplify the operations to be carried out if we choose $w = 1$. In this case, the transition from 3D to 4D is trivial: $(x, y, z) \rightarrow (x, y, z, 1)$.

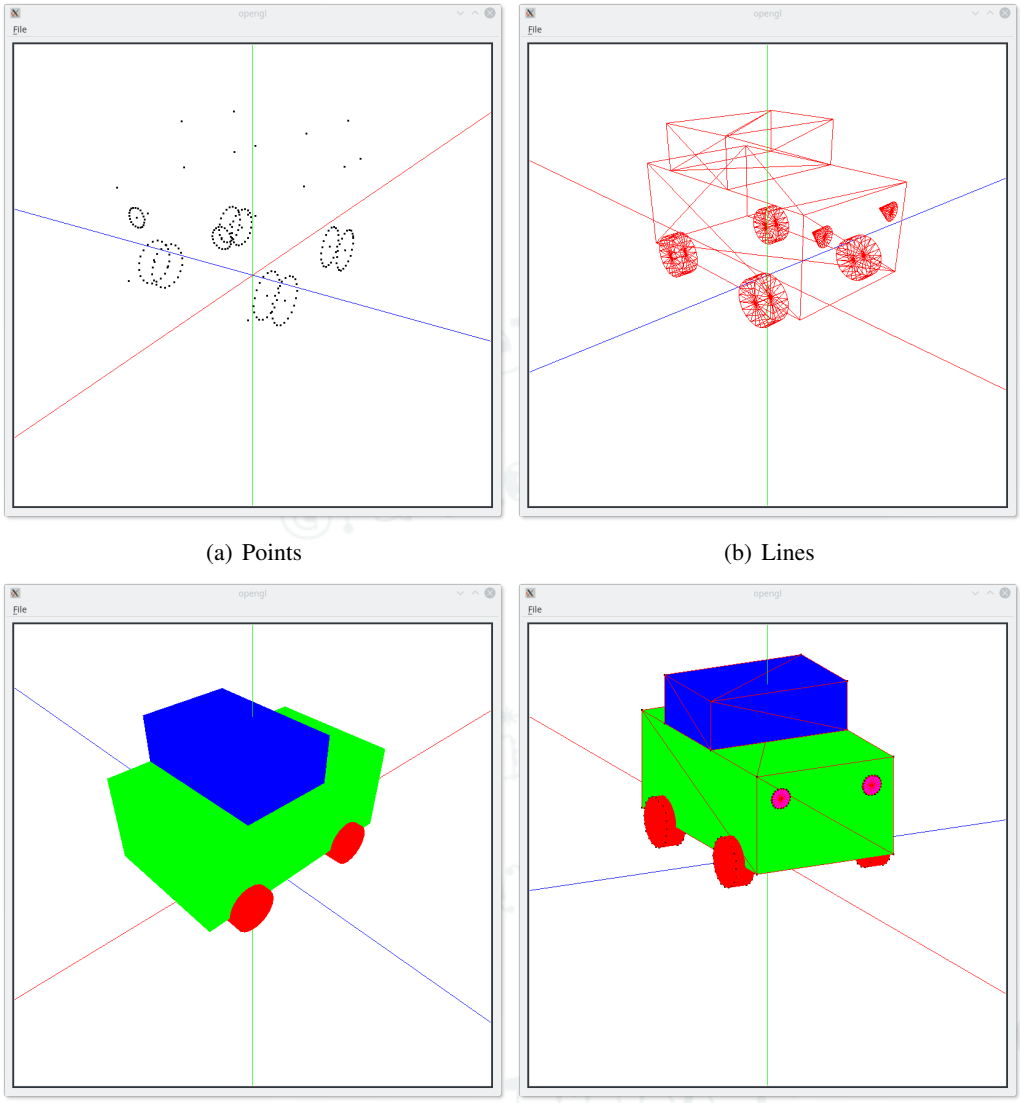
If we have passed the points to 4D, the matrices that represent the transformations must be changed accordingly. In this case, the translation can be easily integrated. Let's see how the different transformation matrices look for translation and scaling:

$$\begin{bmatrix} 1 & 0 & 0 & T_x \\ 0 & 1 & 0 & T_y \\ 0 & 0 & 1 & T_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \qquad \begin{bmatrix} S_x & 0 & 0 & 0 \\ 0 & S_y & 0 & 0 \\ 0 & 0 & S_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

And those of the rotations:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos(\alpha) & -\sin(\alpha) & 0 \\ 0 & \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} cos(\alpha) & 0 & \sin(\alpha) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\alpha) & 0 & \cos(\alpha) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Therefore, we have already seen how it is possible to create the matrices that represent the transformations. And most importantly, this way of writing transformations has the great advantage that the process of combining transformations becomes a simple matrix multiplication. If we have the following matrices that perform a scaling, M_s, a rotation, M_r,



(a) Points (b) Lines

(c) Fill (d) All

Figure 3.8: Car example

and a translation, M_t, the matrix that performs the three transformations in that order is the product of the three: $M' = M_t \cdot M_r \cdot M_s$. In addition, it is true that the inverse of the matrix is the inverse transformation, that is, the matrix that undoes the transformation, which can be difficult to calculate. Mathematically, $M'^{-1} = M_s^{-1} \cdot M_r^{-1} \cdot M_t^{-1}$.

The regularity and simplicity make the use of 4x4 matrices the norm in all the graphic libraries and GPUs, in spite of having to carry out more operations.

In figure 3.7) an example is shown in which the transformations are applied to the cube.

3.2.2 Hierarchical modeling

With the use of geometric transformations we have raised the pillars to be able to build objects through what is called hierarchical modeling. The idea is very simple: build more complex objects by using simpler objects. In our example we want to build a very simple looking car from a cube, a cone and a cylinder. From the cube we will create the chassis by means of two pieces, the four wheels will be created from the cylinder and the two lights from the cone. The result can be seen in the figure 3.8.

The problem we have to solve is to understand how from a single object defined in a certain way and position we can generate others that can have different size, orientation and position. The answer is in the use of transformations.

As we have seen in the previous example, from a cube unit centered on the origin we have managed to visualize another cube that had different size, orientation and position. What would happen if the modeling transformation is changed and the cube is redrawn without deleting the scene? Well, you would get a second cube. And if I add a third transformation and the cube is redrawn? A third cube would be added to the image. It is easy to see that you can draw all the transformed cubes you want. And more importantly, only one cube object has been needed to do so.

This is the idea of symbols and instances: a symbol is defined only once but you can draw, instantiate, all the times you want. The instantiation is carried out by means of the application of a transformation that, normally and in its most general form, is composed of an scalation, one or several rotations and a translation. Therefore we already know the basic mechanism for modeling: a transformation matrix is created and the object is drawn applying a transformation.

Now we have to understand how we can use simpler objects to build more complex objects. The key idea is that of a hierarchy, that is, to establish dependencies. Let's look at it with a real example: the movement of one of our arms. If we look at one arm, there is a relationship of dependence on the movement of the joints: I can move a finger independently, but if I move my hand it implies that the finger will move along with the hand. Likewise, if I move the radius and the ulna, my hand will move, and if I move the humerus, the radius and the ulna will move. There is a dependence on movement in such a way that the movement of a part is transmitted to those of which it is composed. This idea can be extended to modeling.

For example, from the cylinder you can create the wheel object, and from the wheel object you can create the four wheels of the car. With the cone you could create the light object and use it twice. With the cube you would create the chassis. With all these pieces you could create the car object, and from a car draw many cars to create a road full of them.

It should be noted that hierarchical modeling seeks the reuse of the elements making modeling easier.

The mechanism to implement the passing of transformations from one level to the next one is by using the stack. The property of the stack is that with the push operation the current values can be passed to the called object, and when it finish to recover them. The instructions that we can use are: *glPushMatrix* and *glPopMatrix*.

These are the operations that produce the *glPushMatrix*:

1. $Top = Top + 1$
2. Copy the content of position $Top - 1$ to new Top

These are the operation that produce the *glPopMatrix*:

1. $Top = Top - 1$

The idea is that every time one o more transformations are applied to an object, the context must be saved. Let's see with a generic example that creates a wheel from a cylinder, and then an wheels axis with two wheels, using the wheel object and the cylinder. For example, the object wheel is created in this way:

```
void wheel::draw_fill()
{
  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
  glRotatef(90,0,0,1);
  glScalef(4,0.4,4);
  Cylinder.draw_fill();
  glPopMatrix();
}
```

The object wheels_axis is created with a cylinder and 2 wheels:

```
void wheels_axis::draw_fill()
{
  glMatrixMode(GL_MODELVIEW);
  glPushMatrix();
  glRotatef(90,0,0,1);
  glScalef(4,1,4);
  Cylinder.draw_fill();
  glPopMatrix();

  glPushMatrix();
  glTranslatef(-2.5,2,0,0);
  Wheel.draw_fill();
  glPopMatrix();

  glPushMatrix();
  glTranslatef(2.5,2,0,0);
  Wheel.draw_fill();
  glPopMatrix();
}
```

3.2.3 Animation

Once we have created the model in a static way, we are going to explain how to apply movement to the objects and how to produce an animation. The way to produce the movement is very easy: to apply transformations that can change.

For example, if we want to rotate the wheel that we have previously created, we could do that in this way:

```
  ...
  glRotatef(α,1,0,0);

  glRotatef(90,0,0,1);
  glScalef(4,1,4);
  Cylinder.draw_fill();
  glPopMatrix();
  ...
```

The important instruction is glRotatef(α,1,0,0). The idea is to change the value of α and in this way to change the transformation that will be applied. This will be done with the use of keys (see below) for each of the movements.

For producing the animation, the automatic change of the values, we are going to use a particular event. The application is continously checking if there is any event to serve, but usually they are managed very fast and there is a lot of idle time. The idea is to use this idle time to produce the continuous changes that will produce the animation.

This can be implemented in Qt with a QTimer, a kind of countdown clock, with a time 0. We will connect the event of the QTimer with our own slot function. Our function must update the values of the parameters. For example:

```
void idle_event()
{
  ...
  α = α + Angle_step;
  ...

  update();
}
```

Remember that once the variables have been updated, you must indicate that the scene has to be redrawn with `update`.

Once we have explained all the components, the student must add the code that represents and draws a hierarchical model with at least a hierarchy of 5 levels and with 3 degrees of freedom, whose parameters can be modified by keyboard.

For this practice, the following keys must be incorporated:

- Key 7: Activate hierarchical object
- Key A: Activate / deactivate the animation
- Keys Q/W: modifies the first degree of freedom of the hierarchical model (increases/decreases)
- Keys S/D: modifies the second degree of freedom of the hierarchical model (increases/decreases)
- Keys Z/X: modifies the third degree of freedom of the hierarchical model (increases/decreases)
- Keys E/R: increase/decrease the rate of modification of the first degree of freedom of the hierarchical model

- Keys T/Y: increase/decrease the rate of modification of the second degree of freedom of the hierarchical model
- Keys U/I: increase/decrease the rate of modification of the third degree of freedom of the hierarchical model

3.3 Evaluation

The evaluation of the practice, up to 10 points, will be done as follows:

- Creation of the hierarchical model and the corresponding classes with at least 5 levels and 3 degrees of freedom, including rotations and translations. (4 points)
- Creation of the model graph (will be included as a PDF) (1 point)
- Inclusion of the transformations so that the model has 3 degrees of freedom and can be moved with the indicated keys. (3 points)
- Animation. (2 points)

3.4 Extras

- To control the speed of each degree of freedom. (2 points)

3.5 Duration

The practice should be done in 3 sessions

3.6 Bibliography

- Mark Segal y Kurt Akeley; *The OpenGL Graphics System: A Specification (version 4.1)*; http://www.opengl.org/
- Edward Angel;*Interactive Computer Graphics. A top-down approach with OpenGl*; Addison-Wesley, 2000
- J. Foley, A. van Dam, S. Feiner y J. F. Hughes; *Computer Graphics: Principles And Practice, 2 Edition*; Addison-Wesley,1992
- P. Shirley y S. Marschner; *Fundamentals of Computer Graphics, 3rd Edition*; A K Peters Ltd. 2009.

Practice 4

Lighting and texturing

4.1 Goals

With this practice we want to obtain a better realism by using lighting and texturing. So, the student need to learn to:

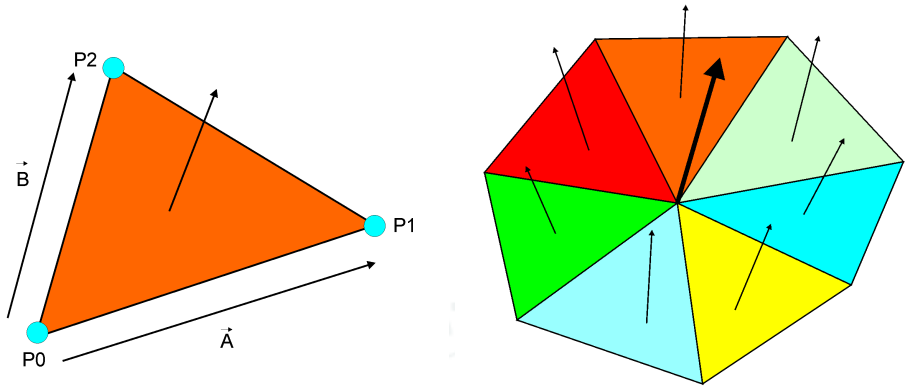
- Lighting
  - Generate the normal faces and vertices for a polygonal model made with triangles
  - Add and use light sources
  - Add and use materials
  - Light an object
- Texturing
  - Compute the texture coordinate for a simple object sencillo
  - Load a texture and show it without lighting

4.2 Development

The objective of this practice is to achieve a greater realism in the visualization of the scenes and for this we will integrate the lighting process and the use of textures.

In order to apply the lighting it is necessary to have at least one object, a light source, and a model that indicates how the light is reflected in that object. To calculate the reflection, not only the position of the light is important (other parameters such as color are less important) but we must also know the orientation of the surface. For this it is necessary to calculate the normal vectors.

Although with the illumination a substantial improvement in the realism is obtained, to improve it we draw on the textures, which consists, in a very general way, in attaching a photo to an object or part of it. To do this we will have to see how image and object are related by texture coordinates.



(a) Normal of a triangle (b) Normal of a vertex

Figure 4.1: Computing the normals. DMP©

4.2.1 Computing the normals

The first thing we are going to do is calculate the normales. On a surface of which we have its exact description, we can compute the normal vector at the interesting point on the surface. Our models based on vertices and triangles, in general [1], are approximations of curved and continuous models, so we also obtain, in general, approximations with the normals.

Since we have triangles and vertices, we can calculate the normals of these elements. We start by computing the normals of the triangles.

The calculation of the normal of a triangle is very simple. The normal plane that includes the triangle is obtained as follows. Given the points P_0, P_1 and P_2, we can calculate the vectors $\vec{A} = P_1 - P_0$ and $\vec{B} = P_2 - P_0$. If we apply the vector product $\vec{A} \cdot \vec{B}$ we get the normal vector, $\vec{N}$, whose direction is given by the rule of the right hand (in general, it is understood that the normal points towards the outside of the polygon) (see figure 4.1(a)).

The normal of the face is used not only for lighting calculations but also serves to determine the orientation of the face, if it is oriented inwards or outwards. It is important that it points outward so that the face can be visualized, if this option is selected by means of the glPolygonMode instruction, where only the faces facing forward are shown, using the value GL_FRONT. One solution is to visualize boths sides of the faces, back and front sides using the value GL_FRONT_AND_BACK, but unless you want to do it because it is planned to enter inside the object, this option will be avoided to not affect the performance of OpenGL, which can remove from the calculations all those faces that are not visible in relation to the observer. To determine whether a face is visible or not from the observer position, it is sufficient to make the scalar or dot product between the normal and the vector that is formed between the position of the observer and a position of the face.

For the calculation of the normal of a vertex in a triangles model, we use the normal of

[1] If the object is polygonal, such as a cube, the representation is accurate

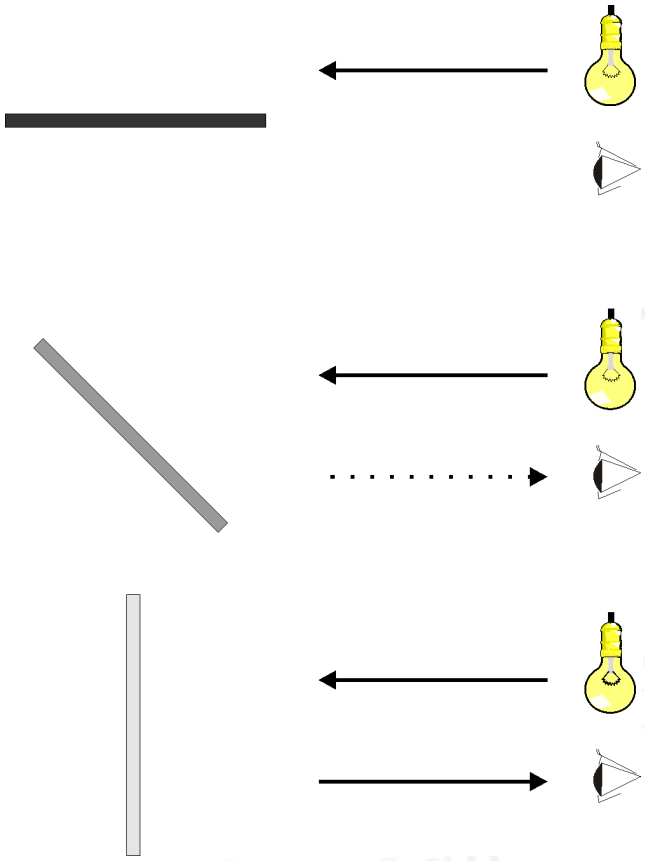


Figure 4.2: The effect of the direction of the diffuse reflection. DMP©

the triangles that converge in such vertex and calculate the average value of the normal (see figure 4.1(b)). This is:

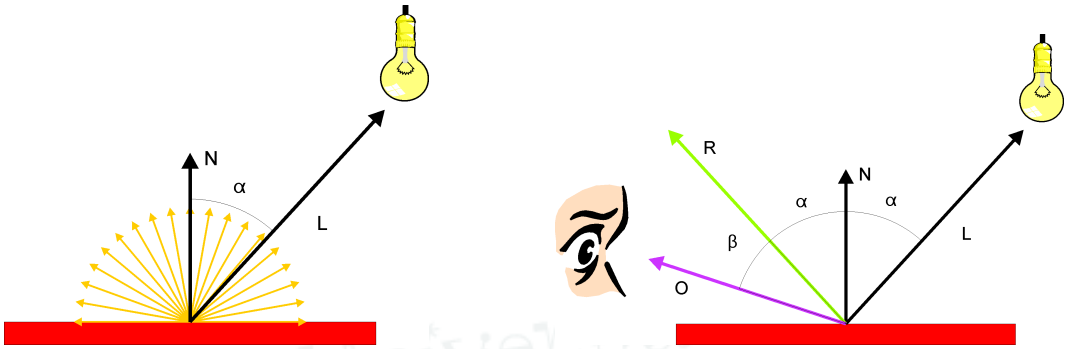
$$\vec{N} = \frac{\sum_{i=1}^{n} \vec{N}_i}{n}$$

It is very important that once we have calculated the normal vectors they must be normalized.

4.2.2 Lighting

Once we have the normals, we show how the different components are included to produce the lighting. To achieve greater realism through lighting, we are going to explain the two modes implemented in OpenGL: flat smoothing, FLAT_SHADING, and Gouraud smoothing, SMOOTH_SHADING.

To simulate lighting we need a reflection model: the interaction between light and objects. The one we are going to use is a simple model that has three components: environmental, diffuse and specular.



(a) Diffuse reflection (b) Specular reflection

Figure 4.3: Types of reflections. DMP©

Let's start with the easiest component to understand, and in many cases the most important, the diffuse component. The basic idea to understand their behavior is to comprehend that the orientation of the object with respect to the light source modifies the amount of light reflected: the more perpendicular it is to the rays of light the more it will reflect. Figure 4.2 shows the idea.

Since the surfaces we are working with are triangles, it implies that the more perpendicular the plane in which the triangle is inscribed, the greater the amount of light reflected. How is the plane defined? Through the use of the normal! If we use the normal to define the orientation of the triangle, the reflection will be 0 when it is perpendicular to the vector that indicates the direction of the light, and it will be maximum when the two vectors are parallel. That is, when the vectors are perpendicular, the value must be 0 and when they are parallel it must be maximum. We can see that the scalar product has that property. Or if we take into account the angle that is formed between both vectors, when the angle is 0 the value must be maximum and when the angle is 90 degrees, the value must be 0. The cosine function complies with this property, if we understand that the value it returns will serve as a modulator. Thus, $\cos(0) = 1$ and $\cos(90) = 0$. To calculate the cosine we will use the scalar product: $\vec{N} \cdot \vec{L} = |\vec{N}| \cdot |\vec{L}| \cdot \cos(\alpha)$. If we have the normalized vectors, then $|\vec{N}| = 1$ and $|\vec{L}| = 1$ and therefore, the equation looks like this: $\vec{N} \cdot \vec{L} = \cos(\alpha)$. Figure 4.3(a) shows the effect of diffuse reflection and the arrangement of vectors. It is important to note that for this component the position of the observer does not affect the result. That is, when a ray of light arrives the object reflects in all directions. Most of the objects we see usually have a mostly diffuse component.

It is also important to keep in mind that the scalar product can give negative values, in such case it implies that the front of the face is oriented backwards. This method can also be used to avoid calculations of faces that are not visible, it is the so-called *culling* that can be enabled in OpenGL.

There are objects that reflect light in another way, which we will call specular reflection. An example are mirrors, almost an ideal specular reflector. In these types of objects, an input ray is reflected as an output ray with the same entry angle. The idea is shown in the figure 4.3(b). In this case, the position of the observer is important: depending on the

observer, it is possible that you will or will not see the reflected ray.

In an ideal mirror reflection, the angle between the reflected ray and the direction of the observer would have to be 0. What normally happens is that the mirror-like reflectors are not ideal and a certain dispersion occurs around the reflected ray. To model this behavior again we drawn on to cosine function, but this time, for the angle β that is formed between $\vec{R}$ and $\vec{0}$: when β is 0 you get the maximum reflection and when it is 90^o the minimum is obtained. Since we can find more or less specular objects, it is necessary to add a modifier of the cosine function: raise the cosine to a power, with values between 0 and infinite. Finite values are used in real implementations. Therefore, the specular reflection would be as follows $\vec{R} \cdot \vec{O} = \cos^n(\beta)$, with the vectors $\vec{R}$ and $\vec{O}$ normalized and n being the exponent.

The last component is the environmental one. With this component it is intended to model a kind of constant light flux that comes from all directions. This flow is not imagined but has a physical foundation. Imagine a small room without any lighting except the light that enters through a small hole. An observer who is inside will be able to see the different parts of it. The reason is that the light begins to reflect on the walls again and again making visible what in principle was not directly illuminated. Therefore, it is a component that came from the numerous reflections of the light rays on the different objects. It is easy to see that as the rays are reflected they will acquire the hue of the material of the object that produces the reflection. In a global lighting model, all these reflections are taken into account. In the simple lighting model that we are explaining, a local model, this component is simplified by using a constant low intensity value. In addition, avoid the effect that faces that are not directly lit to have black color, which is unnatural.

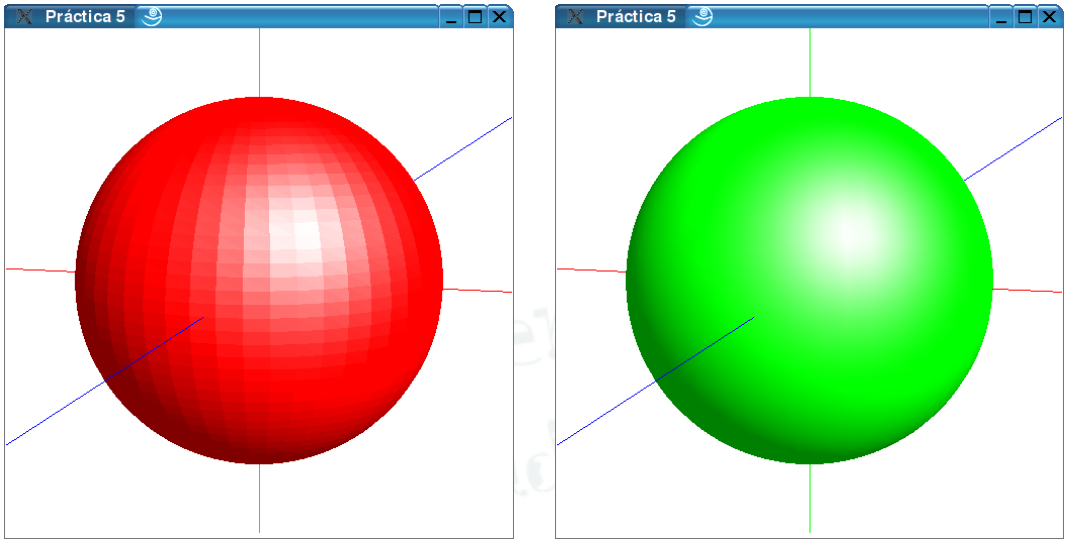
Let's recap. Our light reflection model is as follows: $I_r = A + D + E$. That is, the intensity of light reflected by an object is the sum of the environmental, diffuse and specular components. We have seen that the value of diffuse and specular reflections depends on the orientation of the object with respect to the light in the case of diffuse reflection, and the observer with respect to the ray reflected in the specular reflection. Now we have to include the intensity of the light source and the material: if there is no light we cannot illuminate anything, the material of the object modulates the reflection, for example a dark color reflects less light than a light color.

If the intensity of the light source is I_l, if the capacity of the material to reflect the diffuse component is K_d, the capacity of the material to reflect the specular component is K_e, and the environmental reflection is modeled with K_a, the new formula is:

$$I_r = I_l * K_a + I_l * K_d * cos(\alpha) + I_l * K_e * \cos^n(\beta)$$

Since both light and materials are defined as RGB colors, the above formula must be extended to all three components.

An important detail to consider in the equation that we have explained is the possibility of obtaining values of reflected intensity greater than 1, when in OpenGL, the maximum intensity is 1. Let's look at an example. If we assume that the value of the light is $I_l = 1$, white light, the environmental component is $K_a = 0.25$, the diffuse component is $K_d = 0.8$, and the specular component is $K_d = 0.5$, a medium gray, we can see that at least the intensity reflected will be $I_r = 1 * 0.25 \rightarrow 0.25$, but the maximum value will be given when α and β are 0. In this case the value will be: $I_r = 1 * 0.25 + 1 * 0.8 * 1 + 1 * 0.5 * 1 \rightarrow 1.55$. This



(a) Flat shading (b) Smooth shading

Figure 4.4: Types of shading. DMP©

value has to be trimmed to 1. What we will see is that many parts of the object are seen with maximum intensity. Therefore, it is important to correctly modulate material values, since it will be quite normal to use white light sources $I_l(1,1,1)$.

Finally, let's look at the two possibilities OpenGL offers to apply lighting: flat shading (GL_FLAT) and Gouraud shading (GL_SMOOTH). The difference is that in the flat shading only one normal is assigned to the three vertices of each triangle, the normal of the triangle and therefore a uniform color is obtained for the entire face (Figure 4.4(a)). In Gouraud shading each vertex is assigned its corresponding normal. Therefore we will have three colors in the vertices and for the intermediate positions OpenGL applies an interpolation producing the effect that it looks like the surface is curved (Figure 4.4(b)).

To be able to illuminate the scene we need to define at least one light source. For this we will use the OpenGL glLight function in its different versions. With this function we can control things like position, color, if it is a focus type, etc. In our case, we will use it to define the position and color. To do this, use glLight(LIGHT, PARAMETER, VALUE). LIGHT is the light we want to change. There are eight by default and it will be named with GL_LIGHT0 until GL_LIGHT7. As we are going to change the position we use as PARAMETER the value GL_POSITION. Finally in VALUE we would put the direction of a vector with the coordinates of the position of the light. It is important to note that 4 coordinates must be sent, taking into account that the w is interpreted as follows: if $w = 0$ then the light is at infinity; if $w \neq 0$ then the light is not at infinity.

A very important detail to consider when indicating the position of the light source is that the transformation defined in the GL_MODELVIEW is applied to it. This allows the lights to change position through transformations.

While for GL_LIGHT0 the color is defined as white, for the rest of the lights it is black.

Therefore, if we add a second light it will be necessary to indicate its position and also its color. For this, the same function is used but changing the PARAMETER to GL_DIFFUSE, GL_SPECULAR, etc.

Remember that you have to enable lighting and also the lights to be used so that it can be displayed correctly.

Materials are very easy to define using the glMaterial function. The format is as follows: glMaterial(SIDE, PARAMETER, VALUE). SIDE is the side of the face on which we want to make the modification. It can be the front side, the rear or both. We have already seen this with glPolygonMode. The parameters that will interest us are GL_DIFFUSE, GL_SPECULAR, GL_AMBIENT and GL_SHININESS, which have a direct correspondence with the lighting model explained above. Just keep in mind that the brightness is a single float while the other parameters need a color.

4.2.3 Textures

Now we are going to explain one of the functionalities that allows to obtain greater realism: the textures.

Imagine that we want to produce a realistic 2D image: a good painter will achieve a good result with a lot of effort, but nothing will be better than a photograph. This idea can be extended to the visualization of realistic models: only with vertices and colors it is very difficult that we achieve a high degree of realism, and if it is achieved it is only possible with a large number of vertices.

Let's give an example. Imagine we want to model a wooden plank. The shape of it is very easy to model with a parallelepiped, a stretched cube. With 8 vertices and 8 colors, little can be achieved to simulate wood. The solution would be to increase the number of vertices until we achieve the desired level of detail.

The solution proposed with the textures is as follows: we take a photo of each side of the wood and each photo is attached to one side of the model. The geometry in simple but the visualization is realistic. We will not go into how the image can be pasted on the model, but it helps to think that the photograph has been printed on a fabric that is deformable, so that we adjust it to the object.

The steps to apply a texture are shown in the figure 4.5. The first step is to pass a matrix image to a normalized coordinate system, with coordinates u and v, fulfilling that $0 \leq u \leq 1$ and $0 \leq v \leq 1$. This normalization allows to make the real size of the image of your application independent of the model.

The next step is to assign the corresponding texture coordinates to each point in the model. In the example in the image, to represent a rectangle we need 2 triangles. If we have 4 points, P_0, P_1, P_2 P_3, the T_0 triangle could be made up of the $points(P_2, P_0, P1)$ and the T_1 triangle for the $points(P_1, P_3, P2)$. Since we want to show all the texture, that implies the following texture coordinate assignment:

- $(0,0) \rightarrow P_0$
- $(1,0) \rightarrow P_1$

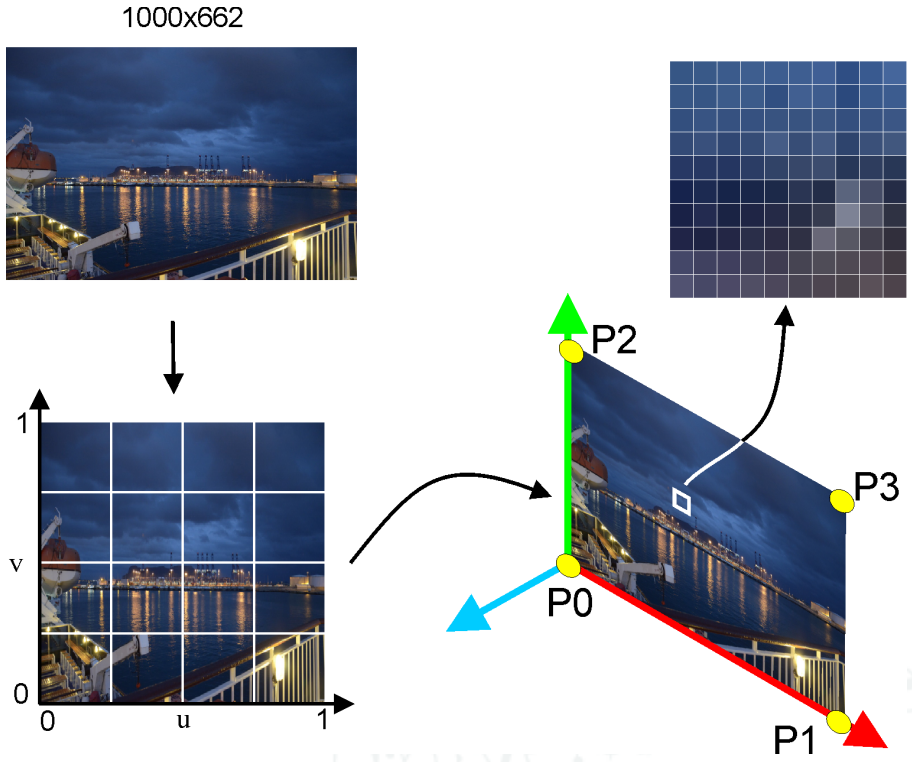


Figure 4.5: Steps to apply a texture. DMP©

- $(0,1) \rightarrow P_2$
- $(1,1) \rightarrow P_3$

If we only wanted to show the central part, the coordinates could be the following:

- $(0.25,0.25) \rightarrow P_0$
- $(0,75,0.25) \rightarrow P_1$
- $(0.25,0.75) \rightarrow P_2$
- $(0.75,0.75) \rightarrow P_3$

It is important to note with these two examples that the difference in what is shown has been achieved simply by changing the texture coordinate, not the coordinates of the points.

Once the correspondence has been made, OpenGL takes care of the hard work, making the correspondence between the pixels of the input image and the pixels of the output image, solving the different scale problems that exist, performing the interpolation, adjusting the perspective, if any, etc.

One detail to keep in mind is that image formats usually use a left coordinate system, with the origin in the upper left corner while OpenGL uses a right coordinate system with the origin in the lower left corner. Therefore we apply a horizontal reflection.

4.2.4 Implementation

The student will have to:

- Lighting
  - Create the code that allows you to calculate the normals of the triangles and vertices. The code will be added _object3D class. This will allow to apply the new functionality to all objects
  - Create the code that allows drawing with flat and Gouraud shading. The code will be added in the functions already defined in _object3D class. This will allow to apply the new functionality to all objects
  - Add a first white light at infinite
  - Add a second magenta light not at infinite
  - Move the second light around the object when the animation is activated
  - Add three materials
  - Be able to visualize the lighting with the sphere. The functions to calculate the normals of the triangles and vertices will be overloaded.
- Texturing
  - Create the class for the chess_board. It will be centered with respect to the origin and located in the plane $z = 0$. Texture coordinates will be calculated so that the image occupies the entire board
  - Create the code that allows you to draw the texture without lighting and with illumination with flat and Gouraud shading. The code will be added in the functions already defined in _object3D. This will allow to apply the new functionality to all objects

A small change is made in the control with the remaining keys as follows:

- Key p: Display in points mode
- Key l: Display in lines/edges mode
- Key f: Display in fill mode

- Key 1: Activate tetrahedron
- Key 2: Activate cube
- Key 3: Activate cone
- Key 4: Activate cylinder
- Key 5: Activate sphere

- Key 6: Activate loaded PLY object
- Key 7: Activate hierarchical object
- Key 8: Activate dashboard

- Key A: Activate/deactivate the animation

- Keys Q/W: modifies first degree of freedom of the hierarchical model (increases/decreases)
- Keys S/D: modifies second degree of freedom of the hierarchical model (increases/decreases)
- Keys Z/X: modifies third degree of freedom of the hierarchical model (increases/decreases)

- Keys E/R: increase/decrease the rate of modification of the first degree of freedom of the hierarchical model
- Keys T/Y: increase/decrease the rate of modification of the second degree of freedom of the hierarchical model
- Keys U/I key: increase/decrease the rate of modification of the third degree of freedom of the hierarchical model

- Key F1: Solid mode display
- Key F2: Display in chess mode
- Key F3: Display in flat shaded lighting mode
- Key F4: Gouraud shaded lighting mode display
- Key F5: Display in unlit texture mode
- Key F6: Display in texture mode with flat shaded lighting
- Key F7: Gouraud shaded lighting texture mode display

- Key J key: Activates/deactivates the first light
- Key K key: Activates/deactivates the second light

- Key M key: consecutive selection between the three materials

4.3 Evaluation

The evaluation of the practice, up to 10 points, will be done as follows:

- Creation and proper operation of normals for all objects except the sphere (2 points)
- Optimized creation and correct operation of the normals for the sphere (1 point)
- Creation and correct operation of the lights (1 points)
- Creation and correct operation of materials (1 point)
- Correct implementation of shading modes (1 point)
- Correct implementation of textured drawn mode (1 point)
- Creation of the board (1 point)
- Assignment of texture coordinates for the board (2 points)

4.4 Extras

- Assignment of texture coordinates for the cylinder. (1 point)
- Assignment of the texture coordinates for the sphere. (1 point)

4.5 Duration

The practice should be done in 3 sessions

4.6 Bibliography

- Mark Segal y Kurt Akeley; *The OpenGL Graphics System: A Specification (version 4.1)*; http://www.opengl.org/
- Edward Angel;*Interactive Computer Graphics. A top-down approach with OpenGl*; Addison-Wesley, 2000
- J. Foley, A. van Dam, S. Feiner y J. F. Hughes; *Computer Graphics: Principles And Practice, 2 Edition*; Addison-Wesley,1992
- P. Shirley y S. Marschner; *Fundamentals of Computer Graphics, 3rd Edition*; A K Peters Ltd. 2009.

Practice 5

Interaction and camera

5.1 Goals

With this practice we want the student to learn to:

- use mouse events to move the camera.
- create a camera with parallel projection and zoom with it
- perform the selection operation, textit pick, to an object.

5.2 Development

The student will have to add the code so that the two degrees of freedom in the movement of the mouse become the two rotations of the camera that is implemented by default. The movement of the wheel will be mapped with the zoom in and out of the camera.

Using the c and v keys you can switch between a camera with perspective projection and a parallel one. The corresponding values will be saved so that the change does not produce discontinuities.

A selection with the mouse pointer, *pick*, will be implemented so that you select one of the triangles of the object, which will be shown in yellow. For this, the following will have to be done:

- Add to each object a variable that indicates if a triangle is selected and if so, the number of the triangle
- Modify the right-click event so that when released, the selection is made and the selected triangle is displayed if it is in draw_fill mode
- Modify the draw_fill drawing method so that you can paint the selected triangle in yellow

- Define the draw_selection function which is responsible for drawing the triangles of the model but making the color of each depends on its position. This implies that the position must be converted to a color
- Create the *pick* function

For the initial development of the selection a single object will be used (cube, sphere, Beethoven, etc.), but for the evaluation an array of $n \times n$ objects arranged in n rows by n columns will be drawn.

The following keys will be added:

- Key c: Perspective projection
- Key v: Parallel projection

5.2.1 Picking by color

The idea of using color for the selection is very simple: each object we want to be able to select will have a unique identifier associated with it. When drawing each object, the identifier is converted to a color, which, therefore, will also be unique. The z-buffer algorithm ensures that only the closest visible pixels are drawn. The only thing left is, given the position of the selection, the x and y coordinates of the mouse pointer, read the color of the corresponding pixel and convert it back to an identifier.

Normally the position is used as the unique identifier. For example, if we are identifying the triangles of an object, each one will occupy a position in the vector of triangles.

5.2.2 Integer to color conversion

To be able to convert an integer to an RGB color you have to take into account how both are stored. An integer is stored in 4 bytes, XYZW, and we will assume that a color is also stored in 4 bytes, XRGB. The first thing we can understand is that since I have 3 bytes to define the color, only the first 24 bits of the integer can be converted. This implies that 16777216 objects can be selected. Actually, we must subtract one, being 16777215, since we need a color, white, to define that there is no object. This implies that the background color will be white.

Thus, we can establish the following relationship between positions of the integer value and the positions of the colors: Y $\rightarrow$ R, Z $\rightarrow$ G, W $\rightarrow$ B. This means that the first values will fill the blue component first, then the green component and finally the red component.

For the assignment we will use our knowledge about the use of bit masks and shift operations. Thus, to obtain the part corresponding to the red component we would have to first extract it with a mask and then apply a shift: Red = (Position & 0x00FF0000)>> 16. Remember that & is the bitwise AND operator.

With this operation we have the red component, but with an integer value, while OpenGL uses standardized floating values. To achieve this, we have to convert our component to floating and divide it by 255.0.

This process should be applied to the other two components but taking into account their position.

To pass from RGB to an integer value, the reverse process must be applied. You just have to have one detail: if the machine is *litte indian*, the least significant values are in the smallest positions and the most significant values in the highest. Remembering, if we have the following number 0x12345678, on a machine with architectural *litte indian* (Intel, AMD) the values will be saved in the following positions, from least to greatest: 0x78563412. Make the necessary adjustments to make the change correctly.

5.3 Evaluation

The evaluation of the practice, up to 10 points, will be done as follows:

- Move the camera with the mouse (2 points)
- Implement the orthogonal camera and zoom (2 points)
- Implement color selection on a single object showing the selected triangle (4 points)
- Implement the color selection of an object among several ones and display it (to create a matrix of 4×4 PLY objects) (2 points)

5.4 Extras

- Implement color selection among several objects in such a way that the selected object and the selected triangle are displayed. (2 pts.)
- Select an object and make the camera revolves around it (2 pts)

5.5 Duration

The practice should be done in 2 sessions

5.6 Bibliography

- Mark Segal y Kurt Akeley;*The OpenGL Graphics System: A Specification (version 4.1)*; http://www.opengl.org/
- Edward Angel;*Interactive Computer Graphics. A top-down approach with OpenGl*; Addison-Wesley, 2000
- J. Foley, A. van Dam, S. Feiner y J. F. Hughes; *Computer Graphics: Principles And Practice, 2 Edition*; Addison-Wesley,1992

- M. E. Mortenson; *Geometric Modeling*; John Wiley & Sons,1985
- http://www.lighthouse3d.com/opengl/picking/index.php