UTT — LO13 — Practical sessions

F. Blachère, florian.blachere@utt.fr
H. Borouchaki, houman.borouchaki@utt.fr
UTT, GAMMA3

P23 — March 14, 2023

Contents

0 Short introduction to C language

This section is devoted to give you the basis of the C language. For documentation about the C language, you can look at the following web sites and at the UTT, courses NF04 and NF06 can help you.

- https://www.w3schools.com/c/index.php
- https://www.tutorialspoint.com/cprogramming/index.htm
- https://www.tutorialspoint.com/c_standard_library/
- https://en.cppreference.com/w/c

0.1 Main

The `main` is the core of you code and will be executed when you run your program. A minimal `main` is presented in Listing 1. This minimal program just print a message in the console.

```c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv){
    printf("Minimal main for introduction\n");
    return EXIT_SUCCESS;
}
```

Listing 1: Main function

All lines of instruction must end with a ';' and the blocks are delimited by '{' and '}'.

0.2 `Code::Blocks`

In order to write a C code you need a text editor and a compiler which may me included in an IDE (Integrated Development Environment). At the UTT, `Code::Blocks` is installed. In order to get it from your personal machine, you need to go on the website of `Code::Blocks`, then in the section "Downloads" get a version with `mingw` in the section "binary release".

To create a new project, you need to go in the menu "File>New>Project...", then select "Console Application", indicate that you will code in C (not C++!), and finally give a name to your project and the path where the files are saved. At the end, in the left menu, the new project is created and you will find a `main.c` file to edit.

To compile the code, you need to click on the yellow cog-wheel and to run it click on the green arrow. In the project menu on the left, the one in bold is the current project that will be compiled. In order to to change the status of a project you need to double click on it.

At the first run of `Code::Blocks`, if there is an error in blue about the compiler, you need to go in the menus: " Settings -> Compiler... -> Global compiler settings -> GNU GCC Compiler -> Toolchain executables " (fourth tab) then click on "Auto-detect" in order that `Code::Blocks` finds the path of the executables.

0.3 Variables and operations

In C, the definition of variables can be done everywhere but to have a more readable code, it is better to make them at the beginning of your code. The main types are:

integer	`int n;`
real	`float x;`

Table 1: Basics types in C

The mathematical operators (`+`, `-`, `*`, `/`) can be directly used as in Listing 2. Be careful, the `/` between two integers is the euclidean division. For instance, in C `1/3` gives 0 and `1.0/3.0` gives 0.33333....

```
int n, m;
n = 4;
n = n + 12;
m = 7;
```

Listing 2: Basic operators

It is possible to use short commands for things that are often used. For example, in the Listing 3 three commands increase `n` by 1.

```
int n;
n = n + 1;
n += 1;
n++;
```

Listing 3: Operators for incrementation

0.4 Inputs and outputs

To print things in the console, the function to use is `printf`. Some examples are presented in the Listing 4. First, the format is given, then the variables to print. `%d` is of an integer, `%f` for a real and `\n` is to break a

```
int n = 2;
float x = 5.78;
printf("The terme of rank %d is %f\n", n, x);
printf("The number after %d is %d\n", n, n+1);
```

Listing 4: Printing to the console

line.

0.5 Loops and conditions

To run a determined loop, the syntax is given in Listing 5. The general syntax, is `for(initialisation; ending; step){}`. The undetermined loops are written in the following manner: `while(test){instructions}` where the instructions are executed while the condition "test" is true.

```c
int i;
for (i=1; i<7; i+=2){
  printf("%d\n", i);
}
```

Listing 5: For loop

The conditional blocks are written as in Listing 6 using the logical operators "AND" (`&&`) and "OR" (`||`).

```c
double x;
if (x >= 1e6) {
  printf("x is tall\n");
} else if (x < 1e6 && x>=1) {
  printf("x is medium\n");
} else if (x<1 && x>=0) {
  printf("x is small\n");
} else {
  printf("None of the previous case\n");
}
```

Listing 6: Conditional blocks

0.6 Arrays

0.6.1 Static arrays

To use static arrays (where the size is known when compiling), the variable is defined directly. The access to the value is made with `[ ]`. Be careful, in C the indexes always start at 0. You can declare one dimensional arrays as in Listing 7 and also matrices (two dimensional arrays) as in Listing 8. For matrices, one may also work with a one dimensional of size $2 \times 3 = 6$ to achieve the same result. In this case `mat[i][j] = tab[j+3*i]` as C use row-major order[1].

```c
float array[5];
int i;
for (i=0; i<5; i++){
  array[i] = 2.1*i;
}
for (i=0; i<5; i++){
  printf("array[%d] = %f\n", i, tab[i]);
}
```

Listing 7: 1D static arrays

0.6.2 Dynamic arrays

When the size of the array is only known during the run of the code, you need to allocate it using pointers. The two previous example can be done using dynamic arrays using the code in Listing 9. As the memory is dynamically managed you need to free it using the `free` command at the end of your program.

[1] https://en.wikipedia.org/wiki/Row-_and_column-major_order

```
float mat[2][3];
int i, j;
for (i=0; i<2; i++){
    for (j=0; j<3; j++){
      mat[i][j] = i+54*j;
  }
}
for (i=0; i<2; i++){
    for (j=0; j<3; j++){
      printf("%f ", mat[i][j]);
    }
    printf("\n")
}
```

Listing 8: 2D static arrays

```
float *tab;
float **math;
tab = allocate(5*sizeof(float));
mat = allocate(2*sizeof(float*));
for (i=0; i<2; i++){
    mat[i] = allocate(2*sizeof(float));
}
// do something with the arrays
free(tab);
for (i=0; i<2; i++){
    free(mat[i]);
}
free(mat);
```

Listing 9: Dynamic arrays

0.7 Functions and structures

The aim of functions is to factorize code in order to reuse and simplify the reading. For instance, the function to compute the power of a number can be written as in Listing 10. This power function, is already written as

```
float power(float x, int n){
  float y;
  y = 1;
  for (i=1; i<=n; i++) {
      y = y * x;
  }
  return y;
}
```

Listing 10: Power function

`pow(x, n)` in the math library that can be use by adding `#include <math.h>` at the beginning of the code. For instance, the `main` (see Listing 1) of a program is a function.

As functions, structures help to factorize the code. For instance, the code in Listing 11 can be reduced and improved to the one in Listing 12 using the declaration of a structure from Listing 13 at the beginning of your

code. It is also possible to define arrays (static or dynamic) of structures as in Listing 14.

```
void compare_vector(int n, float *vec1, int m, float *vec2){
    if (m == n){
        int i = 0;
        while(vec1[i] == vec2[i]){
            i++;
        }
    } else {
        print("Not the same size\n")
    }
}
```

Listing 11: Vector without structure

```
void compare_vector(struct vector vec1, struct vector vec2){
    if (vec1.size == vec2.size){
        int i = 0;
        while(vec1.data[i] == vec2.data[i]){
            i++;
        }
    } else {
        print("Not the same size\n")
    }
}
```

Listing 12: Vector with structure

```
struct vector{
    int size;
    float *data;
};
```

Listing 13: Structures

0.8 OpenGL

For all the next sections, you must start a new project by copying the skeleton from Moodle and then open the `LO13.cbp` file in Code::Blocks (https://www.codeblocks.org/). This file permits to link correctly with the various graphic libraries.

For a full documentation of the OpenGL commands, you can look at:

- https://www.khronos.org/registry/OpenGL-Refpages/gl2.1/
- https://www.opengl.org/resources/libraries/glut/spec3/spec3.html
- https://docs.gl/

```
struct vect static_vectors[2];
static_vectors[0].size = 2;
static_vectors[0].data = malloc(static_vectors[0].size*sizeof(float));
static_vectors[0].data[0] = 1;
static_vectors[0].data[1] = 2;

static_vectors[1].size = 3
static_vectors[1].data = malloc(static_vectors[1].size*sizeof(float))
static_vectors[1].data[0] = 1;
static_vectors[1].data[1] = 2;
static_vectors[1].data[2] = 3;

struct vect *dynamic_vectors;
dynamic_vectors = malloc(4*sizeof(struct vector))
dynamic_vectors[0].size = 2
dynamic_vectors[0].data = malloc(dynamic_vectors[0].size*sizeof(float));
dynamic_vectors[0].data[0] = 1;
dynamic_vectors[0].data[1] = 2;
```

Listing 14: Array of structures

1 Square

The first code to write must draw a square in 2D as in Figure 1.

1. First the GLUT library need to be initialized with `glutInit(&argc,argv)` function, where `argc` and `argv` are coming from the parameters of the `main` function.
2. Then you need to choose the display mode with `glutInitDisplayMode` function. This function needs some bit masks[a]. At the beginning only `GLUT_SINGLE` and `GLUT_RGB` are needed.
3. Then the size, position and title of the window need to be set with:
   - `glutInitWindowSize(lx,ly)`
   - `glutInitWindowPosition(x,y)`
   - `glutCreateWindow(name)`

   You need to choose the value of all the parameters.
4. The display function is then selected with `glutDisplayFunc(display)` where `display` is a function which draw. For that, you need to write a function `display` where:
   - the screen need to be cleared with `glClear(GL_COLOR_BUFFER_BIT)`
   - then draw the polygon by defining the fours vertices with four `glVertex2f(x,y)` between a `glBegin(GL_POLYGON)` and a `glEnd()`.
   - and flush everything to the screen with `glFlush()`.
5. Some initialisation are needed before the end:
   (a) the color of the background (used to clear the screen) with `glClearColor(r,g,b,a)`, where the four parameters are the levels of red, green, blue and alpha transparency that lie into $[0; 1]$.
   (b) the colors for drawing with `glColor3f(r,g,b)`

6. Finally you need to define how the projection is done with Listing 15. It first loads the projection matrix. Then, fill it with the identity matrix and the multiply it by the 2D orthographic projection matrix. Where the four parameters define the coordinates for the left and right vertical clipping planes then, for the bottom and top horizontal clipping planes.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
gluOrtho2D(left,right,bottom,top);
```

Listing 15: 2D projection matrix

7. At the end the (infinite) main loop is run with: `glutMainLoop()`.

[a] https://www.opengl.org/resources/libraries/glut/spec3/node12.html

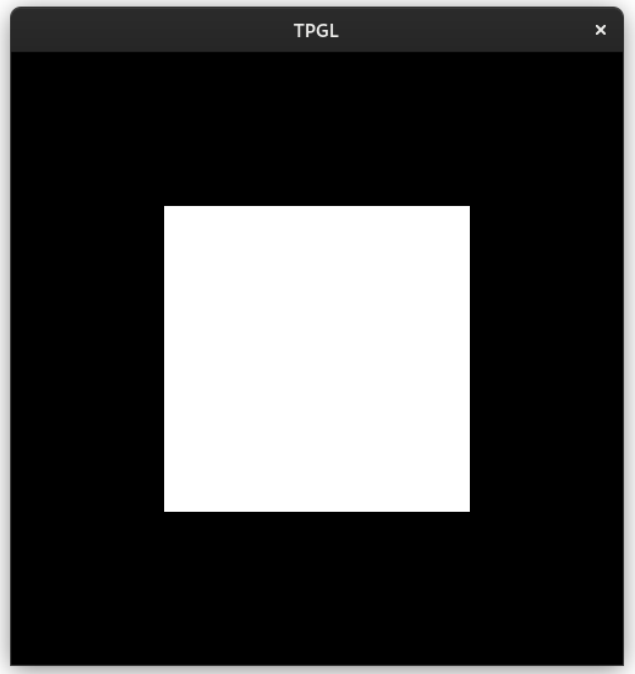


Figure 1: Square

2 Circle with a polygon

Then the aim is to draw a circle using a polygon.For instance with $N = 10$ points, the drawing is in Figure 2.

1. Copy the previous exercise and clear the four vertices of the square.
2. Write a for loop (`for(i=0; i<N; i++)`) to draw a circle like a polygon with N sides. For that, you need to use the polar coordinates of the points. In C, you must add the math library with `#include <math.h>` to have the constant π in `M_PI` and the trigonometric functions.

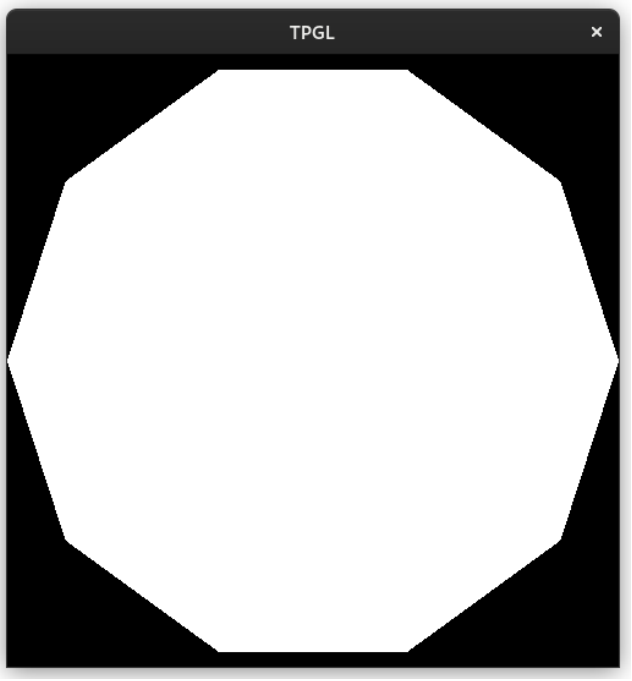


Figure 2: Circle with a polygon

3 Circle with triangles

As in the previous exercise, the aim is to draw a circle but this time using triangles.

1. Copy the previous exercise.
2. The command to draw triangles is `glBegin(GL_TRIANGLE)` instead of `glBegin(GL_POLYGON)`. Then all the groups of three points declared with `glVertex2f(x,y)` will create a triangle. Inside the loop define the coordinates of the three vertices of each triangles.
3. To check your results, you can add the command `glPolygonMode(GL_FRONT_AND_BACK,GL_LINE)` to draw the lines instead of filling the polygons. The results should look like Figure 3.

4 Colormap

1. With the circle made of triangles, change the colors for each triangle using `glColor3f(r, g, b)`.
2. Write the code to have a linear progression of colors from the angle 0 to 2π like in Figure 4

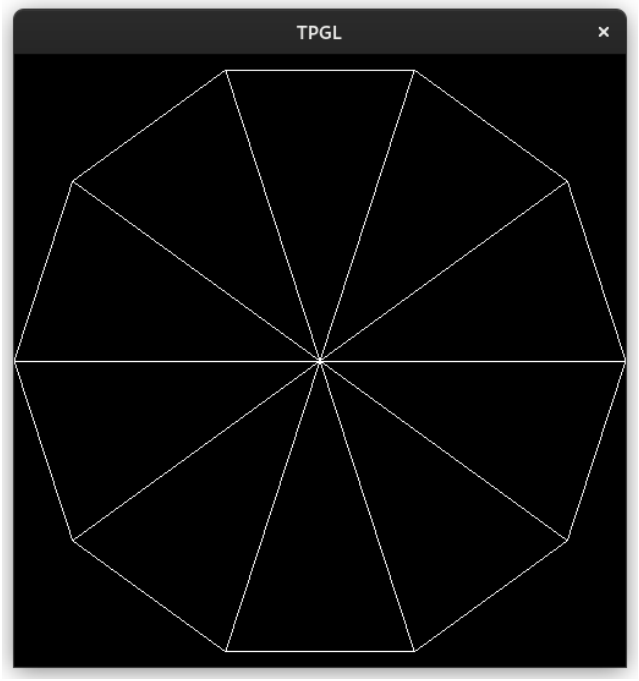


Figure 3: Circle with 10 triangles

Figure 4: Circle with colors

5 Cube

5.1 Discrete model

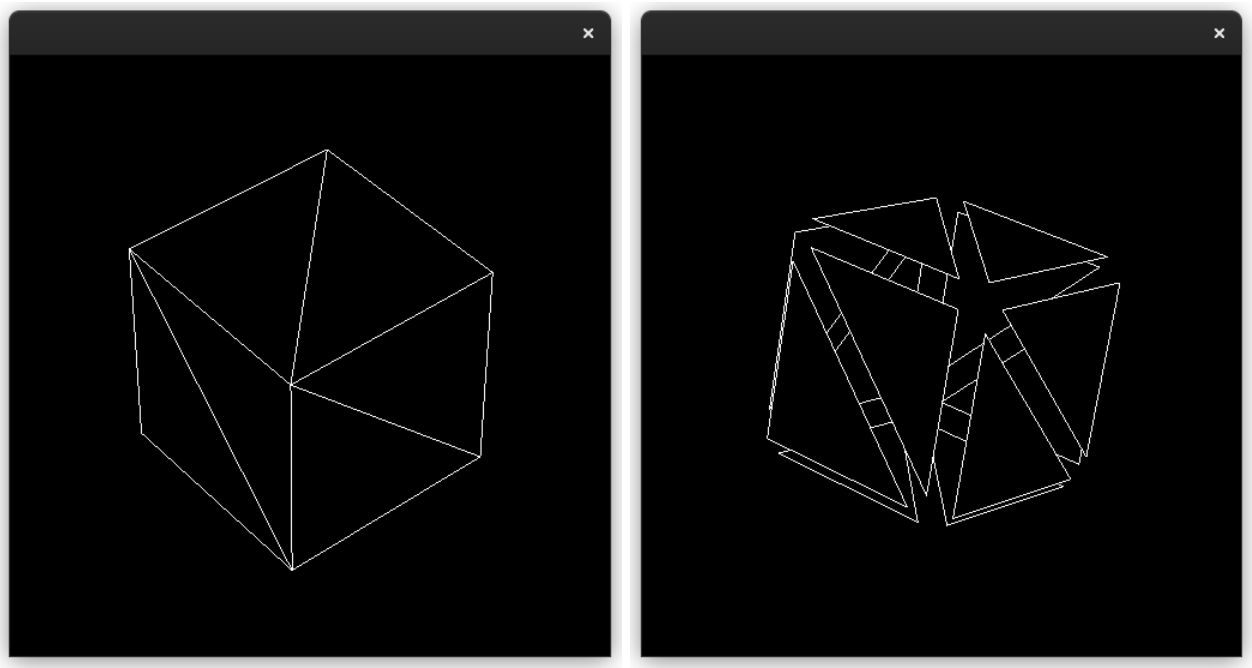
Before drawing in 3D, we need an object to plot. The first is a cube where each square face is composed of two triangles as in Figure 5. In order to save the coordinates of all points of the cube and the connectivity table of the triangles, on may use the structure defined in Listing 16.

1. Define the coordinates of the eight vertices of the unit cube (in $[0;1]^3$).
2. Define the triangles using a connectivity table.
3. Define a global variable of type `struct s_mesh` from Listing 16.

```
struct s_mesh {
    int nVertices, nTriangles;
    float vertices[8][3];
    int triangles[12][3];
};
```

Listing 16: Structure for a cube

4. Write a function `initCube` that fills the variable previously defined.



(5.1) Cube (5.2) Cube with shrink (done in Section 11)

Figure 5: Cube with triangles

5.2 Projection and view matrices

To draw a 3D object, you need to define the projection and view matrices.

1. For the view matrix, you need to use the code from Listing 17.

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(ox, oy, oz, fx, fy, fz, vx, vy, vz);
```

Listing 17: View matrix

   Define the values used in `gluLookAt` and insert the code inside a function named `viewMatrix`.

2. The projection matrix is built using the code from Listing 18 which is closed to the one in Listing 15.

```
glMatrixMode(GL_PROJECTION);
glLoadIdentity();
glFrustum(umin, umax, vmin, vmax, dmin, dmax);
```

Listing 18: Projection matrix

   Define the values used in `glFrustum` and insert the code inside a function named `projectionMatrix`.

5.3 Display

1. Before drawing, you need to define the colors with `glClearColor(r,g,b,a)` and `glColor3f(r,g,b)` as previously. Write those two commands in a `initColors` function.
2. Next, you need to write the `display` function to draw the cube using loops and the `glVertex3f(x,y,z)` function to draw a vertex.
3. You also need to define all the OpenGL initialisation of OpenGL in an `initOpengGL` function (you can copy them from a previous exercise) as in Listing 19.

```
void initOpengGL(int argc, char **argv)
{
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);
    glutInitWindowSize(1000, 1000);
    glutInitWindowPosition(10, 10);
    glutCreateWindow("TPGL");
    glutDisplayFunc(display);
}
```

Listing 19: Initialisation of OpenGL

4. In your main function you call all the previous functions, to get something like Listing 20.

```c
int main(int argc, char **argv)
{
    initOpengGL(argc, argv);
    initCube();
    initColors();
    viewMatrix();
    projectionMatrix();
    glutMainLoop();
    return EXIT_SUCCESS;
}
```

Listing 20: Main function calling the others

5.4 OpenGL structure

Many parameters will be needed to control all the aspects of the drawing with OpenGL. In order to improve the code, one will use a structure for all theses parameters. For instance, it is possible to use the one defined in Listing 21.

```c
struct s_opengl {
    float observator[3], focalPoint[3], vertical[3];
    float u[2], v[2], dist[2];
};
```

Listing 21: Structure for OpenGL

1. Define a global variable of type `struct s_opengl` and initialize it in a `initParameters` function.
2. Rewrite the `projectionMatrix` and `viewMatrix` using the new structure. All the new parameters in the following sections must go to this new structures and be initialized in this function. For instance, instead of `umax`, you will write `opengl.u[1]`.

6 Events

6.1 Reshape

1. With the code of the cube, try to reshape the windows. Does the cube keep its shape?
2. In order to avoid this phenomenon you must register a `reshape` function like you registered the `display` function. You use the `glutReshapeFunc(reshape)` function. The `reshape` function is given by `void reshape(int width, int height)` where the inputs are the new size of the window.
3. Write the `reshape` function to compute and save (in the `struct s_opengl`) the anisotropy factor of the window.
4. At the end of the function, you need to set the viewport to the whole window using: `glViewport(0,0,width,height);`.
5. Use this anisotropy factor to recompute the projection matrix.

6.2 Mouse

The mouse is controlled using two functions: `mouse` and `motion`. The first is running when clicking and the second when the mouse is moving. At a click the function `void mouse(int button, int event, int x, int y)`, to register with `glutMouseFunc(mouse)`, gives you the button (`GLUT_LEFT_BUTTON`, `GLUT_MIDDLE_BUTTON` or `GLUT_RIGHT_BUTTON`), the effect on the button (`GLUT_DOWN` or `GLUT_UP`) and the coordinates of the mouse. When the mouse is moving, the function used is `void motion(int x, int y)` (registered by `glutMotion(motion)`) and it gives you the new coordinates of the mouse.

6.2.1 Geometric transformations

In order to draw, one need to to know all the geometric transformations previously done.

1. In the OpenGL structure (Listing 21) add a `geometricTransformations` variable to save the 4×4 matrix of all previous geometric transformations. For that use a vector of size 16 ($= 4 \times 4$).
2. In the same structure add a variable `flagTransformation` to save the current transformation that is done. For instance, 0 for no transformation and 1 for a rotation and 2 for a rotation.
3. In the `initOpengGL()` function, initialized this matrix to the identity of size 4 and the flag to 0 (no motion).
4. The geometric transformations matrix need to be updated at each transformation. One need to write a `updateCurrentTransformation()` function that looks like Listing 22.

```
void updateCurrentTransformation()
{
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    computeMatrixTransformations();
    glMultMatrixf(opengl.geometricTransformations);
    glGetFloatv(GL_MODELVIEW_MATRIX, opengl.geometricTransformations);
}
```

Listing 22: `updateCurrentTransformation()`

   In this listing, the view matrix is set then filled with the identity. Then, the current transformation (that will be detailed in the next sections) are loaded with the `computeMatrixTransformations()` function. After that, all the previous geometric transformations are loaded. Finally, the new matrix with the current and old transformations is saved.

5. Update the `viewMatrix()` to multiply the current view matrix with this new matrix (before the `gluLookAt` function).
6. Update the `display()` function to call the `updateCurrentTransformation()` before computing the view matrix.

The aim of the two next sections is to properly define the geometric transformations in the function `computeMatrixTransformations()`. For that two commands are necessary:

1. The rotation is made using the command `glRotatef(angle,x,y,z)` where `angle` is the angle of rotation and $(\mathtt{x}, \mathtt{y}, \mathtt{z})^T$ is the axis of rotation.
2. The translation uses the command `glTranslatef(x,y,z)` where $(\mathtt{x}, \mathtt{y}, \mathtt{z})^T$ is the vector of translation.

6.2.2 Rotation

A click on the left button of the mouse should enable the rotation and when moving with the button still down, the object should rotate.

The first step is to manage the click that enable the rotation.

1. In the `mouse()` function add a test to detect a click (`GLUT_DOWN`) on the left button (`GLUT_LEFT_BUTTON`). This click set the flag `flagTransformation` to 1.
2. Besides, one need to save the initial coordinates of the mouse into the OpenGL structure as it will be needed to compute the move.
3. When releasing the left button (`GLUT_UP`) set the flag to 0.

Then, one need to add some code in the `motion()` function to manage the displacement of the mouse and convert it to angles to make the proper rotation.

1. The first step is to compute the vector of `displacement` from the initial coordinates of the mouse (from the function `mouse()` that you saved in the OpenGL structure) and the current coordinates from the arguments of `motion()`.
2. Then, you need to save the new initial position of the mouse for the next call of the `motion()` function.
3. Finally, if the `flagTransformation` is set to 1 you need to convert the `displacement` to two angles (that you need to save in the OpenGL structure). One angle for the rotation around the u axis of the view frame and the second for the rotation around the v axis. This conversion in made in order to have a smooth rotation when moving the mouse: you need to choose a parameter to make the scaling from the pixel of displacement to the angle in radian.

Finally, one need to make the rotation in the `computeMatrixTransformations()`, if the flag `flagTransformation` is set to 1:

1. Translate the focal point to the center of the view frame (u, v, n). For that, it is necessary to save the coordinates of the focal point in the view frame:

$$F_{(u,v,n)} = \begin{pmatrix} 0 \\ 0 \\ -\|O_v F\| \end{pmatrix}$$

   where O_v is the observer and F the focal point.
2. Make the rotation with the two angles around the u and v axis.
3. Move back the focal point.

6.2.3 Translation

As for the rotation, the first step is to manage the click that enables the translation.

1. In the `mouse()` function add a test to detect a click (`GLUT_DOWN`) on the right button (`GLUT_RIGHT_BUTTON`). This click set the flag `flagTransformation` to 2.
2. Besides, you also need need to save the initial coordinates of the mouse and reset the flag to 0 when releasing the button.

Then, one need to add some code in the `motion()` function to manage the displacement of the mouse and convert it to a vector of translation.

1. You keep the lines from the rotation about the `displacement` and the backup of the new initial position of the mouse.
2. In order to make a smooth translation a good conversion from the `displacement` to the vector of translation (that you need to save in the OpenGL structure) is needed:
   (a) First, you need to convert the displacement in the view frame to a displacement in view plane. This view plane is at a distance $d_{\min}$ from the observer in the view frame. The Thales theorem permits to get the relation.
   (b) Then, you need to move from the view plane (in $[-1;1]$) to the real pixels in the screen using a cross-multiplication to get something proportional. You need to adjust this second relation with the value of the anisotropy and zoom parameters.

Finally, one need to make the translation in the `computeMatrixTransformations()`, if the flag `flagTransformation` is set to 2.

1. Translate the whole object using the vector of translation previously computed.
2. Update the coordinates of the focal point in the view frame by translating it.

6.3 Keyboard

6.3.1 Quit

1. One can interact with the drawing with the keyboard. The registration is made with the `glutKeyboardFunc(keyboard)`. The `keyboard` function is given by `keyboard(unsigned char key, int x, int y)` where the inputs are the key pressed on the keyboard and the coordinates of the mouse.
2. Write the `keyboard` function to quit the program when typing on `'q'`.

6.3.2 Zoom

1. In the `keyboard` function add some `case` statement (find on the web the documentation about `switch` in C).
2. When pressing on `'z'` or `'Z'` increase or decrease a `zoom` parameter from the `struct s_opengl` that you will reuse in the projection matrix to zoom in or out.
3. After recomputing the projection matrix, you may reload the drawing with `glutPostRedisplay()`.
4. In order to have a better control on the zoom, add parameters to limit the zoom and set the zoom factor in the `struct s_opengl`.

6.3.3 Reset to initial view

1. Add one `case` statement in the `keyboard`function to reset the object to the initial view (no zoom, no rotation, no translation, ...) when pressing on `'i'`.

6.4 Double buffering

1. When redisplaying quickly the window (for instance, with the reshape), the drawing may be slow. In order to improve the performance, one may switch from single buffering (`GLUT_SINGLE`) to double buffering (`GLUT_DOUBLE`) in the `glutInitDisplayMode` function.
2. After making this change, you also need to change the command `glFlush()` from the end of the display function, to the `glutSwapBuffers()` function.

7 Mesh

The aim is to replace the cube from Section 5.1 by the model of an object from a `.mesh` file.

1. Get from Moodle the archive `07_mesh.zip` containing the files needed to read a `.mesh` file.
2. At the beginning of your `main.c` file add the line `#include "mesh.h"` to get the structure and the function needed.
3. Then look at the head of the `mesh.h` file to see the structure used (recalled in Listing 23) and compare it to the structure for the cube from Listing 16.

```
typedef struct s_point {
    float x[DIM];
    float normal[DIM];
} t_point;

typedef struct s_triangle {
    int indexes[NVERTICESTRI];
    float normal[DIM];
    float angles[NVERTICESTRI];
} t_triangle;

typedef struct s_mesh {
    int number_of_vertices;
    t_point *vertices;
    int number_of_triangles;
    t_triangle *triangles;
} t_mesh;
```

Listing 23: Structure for a mesh

4. Modify the code of the function `initCube` and `display` to use this new structure and check if everything is working as previously.
5. Finally, get the `tiger.mesh` file from Moodle and replace the call to `initCube` by the call to the new `defineModel` function to read the file. You should get a result like in Figure 6.

8 Various drawing modes

The aim of this section is to define various drawing mode that will be accessible using the key '0' to '9' as in Figure 7 and 9 and 10. In order to achieve such drawing, several step are detailed in the next subsections.

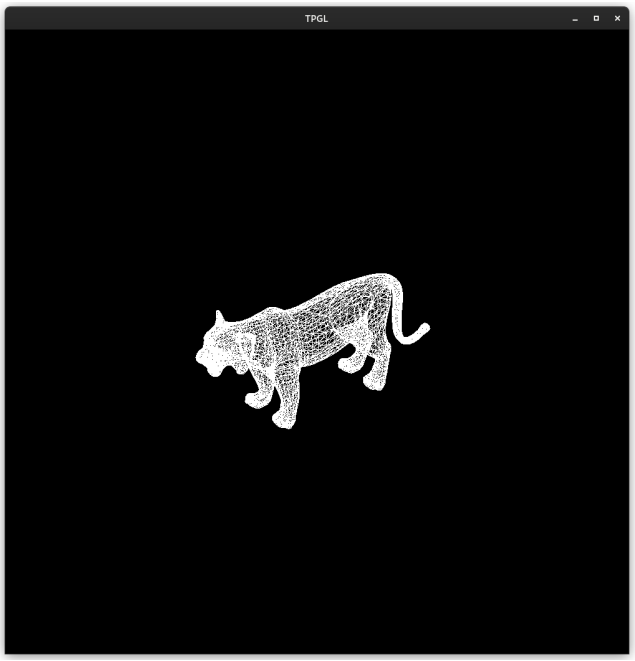


Figure 6: Tiger from `tiger.mesh`

1. First, you need to a variable `mode` to the opengl structure in order to save the current mode used. Then, you need to get this mode (from 0 to 9) from the keyboard by adding some `case` in the `keyboard` function.
2. Then, in the display function you need to add a `switch` on this new variable and write the code for each mode in a `case`.

8.1 Hidden parts

The mode 1 from Figure 7.1 is made by filing the triangles with a color per triangle.

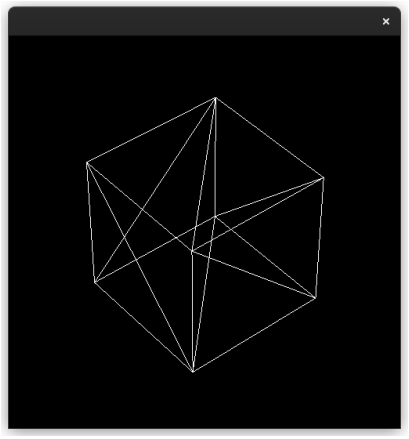
1. In order to fill the triangles, one need to set `glPolygonMode(GL_FRONT_AND_BACK,GL_FILL)` before drawing the triangles.
2. Then, in the loop of the triangles change the colors (using `glColor3f`) for each triangles.
3. Do you see something like in Figure 7.1?
4. To get a proper drawing, one need to enable the "depth test" in order to hide the drawing in the background.
5. For that, several modifications are needed:
   (a) First, add `GLUT_DEPTH` in `glutInitDisplayMode`
   (b) Then before drawing, you need to enable the depth test with `glEnable(GL_DEPTH_TEST)` and disable it at the end with `glDisable(GL_DEPTH_TEST)` like the `glBegin` and `glEnd` functions.
   (c) Finally, when clearing the screen you also need to clear the `GL_DEPTH_BUFFER_BIT`.

   Make the modifications and try again to draw.

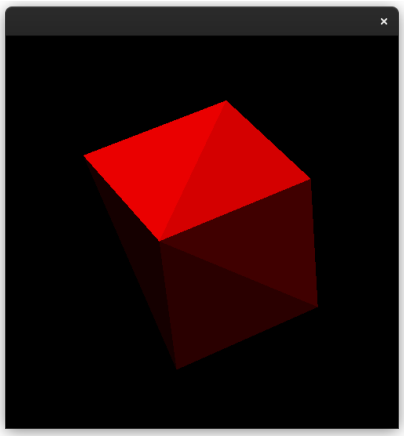
8.2 Polygon offset

In order to draw like in Figures 7.2 and 7.3 one need to draw the object twice: first the line (`GL_LINE`) in a color and then the faces (`GL_FILL`) in another color with the depth test activated (you can also reverse the order).

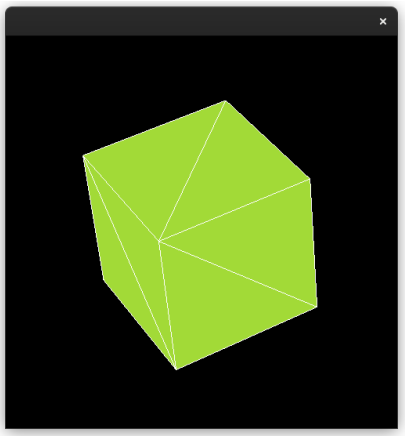
1. Add the modes 2 and 3 by duplicating your code to draw twice the object.
2. Try to draw first the faces then lines. After reverse the order and check the difference.
3. In order to fix the problem, you need to apply an offset to the drawing of the line or the faces. This offset is enabled by adding the command `glPolygonOffset(1.,1.)` followed by `glEnable(GL_POLYGON_OFFSET_LINE)` or `glEnable(GL_POLYGON_OFFSET_FILL)` depending on the order of drawing and of your machine. Finally, then offset is disabled by the corresponding `glDisable`.



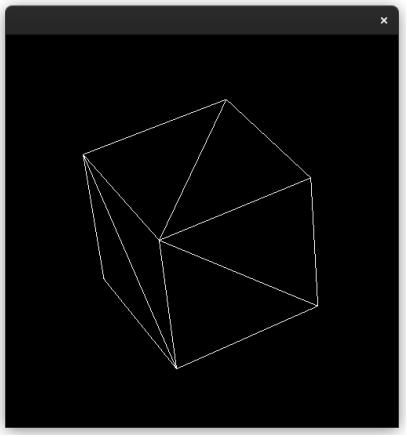
(7.0) Line



(7.1) Fill



(7.2) Fill and line



(7.3) Line hidden

Figure 7: Drawing modes

9 Lighting

The six next modes need the activation of the light and the parameters of the light depend on the material used for the object.

9.1 Light source

1. First you need to add create a new structure `struct s_source`. This new structure contains four arrays of size four in order to define:
   (a) the position of the light. For instance $(1, 1, 1, 0)^T$ indicates a source in the direction $(1, 1, 1)^T$ located at the infinity.
   (b) the values of individual light source parameters: ambient, diffuse and specular. Each contains four values that specify the proper RGBA intensity of the light.
2. Then, you create a variable `source` of type `struct s_source` in the OpenGL structure.
3. This new variable in the OpenGL structure needs to be initialized. For instance, you can create a white light by setting the three arrays to $(1, 1, 1, 0)^T$.

Once the variable `source` is filled, one need to enable/disable the light in the code.

1. To disable the light, the commands are in Listing 24.

```
void disableLight()
{
    glDisable(GL_LIGHT0);
    glDisable(GL_LIGHTING);
}
```

Listing 24: Commands to disable the light

The first command disable the first light source (the source are counted from 0 to 9) and the second disable the lighting.

2. To enable the light, the two previous commands are reused with a `glEnable` as in Listing 25.

```
void enableLight()
{
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);

    glPushMatrix();
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glLightfv(GL_LIGHT0, GL_POSITION, opengl.source.position);
    glPopMatrix();

    glLightfv(GL_LIGHT0, GL_DIFFUSE, opengl.source.diffuse);
    glLightfv(GL_LIGHT0, GL_SPECULAR, opengl.source.specular);
    glLightfv(GL_LIGHT0, GL_AMBIENT, opengl.source.ambient);
    glLightModeli(GL_LIGHT_MODEL_TWO_SIDE, GL_TRUE);
}
```

Listing 25: Commands to enable the light

Then, the parameters (position, diffuse, specular, ambient) are given to OpenGL with the

`glLightfv` commands. For the position, the view matrix needs to be loaded to be properly defined. The `glPushMatrix` and `glPopMatrix` permits to keep the previous view matrix.

Finally, the `glLightModeli` commands permits to consider the two faces of the triangles (if the normals are not correctly oriented).

9.2 Materials

1. First you need to add create a new structure `struct s_material`. This new structure contains three arrays of size four (ambient, diffuse, specular) and a real in order to define:
   (a) the values of individual light material parameters: ambient, diffuse and specular. Each contains four values that specify the proper RGBA reflectance of the material.
   (b) the shininess of the material.
2. Then, you create fours variables of type `struct s_source` in the OpenGL structure as in Listing 26. The first three are for the characteristics of three materials and the last one is for the current material.

```
struct s_material bronze, copper, plastic, *material;
```

Listing 26: Variables for material

3. All these new variables in the OpenGL structure needs to be initialized. Look for the values of the parameters on the web. For the current material the syntax is presented in Listing 27.

```
opengl.material = &opengl.plastic;
```

Listing 27: Set the current material

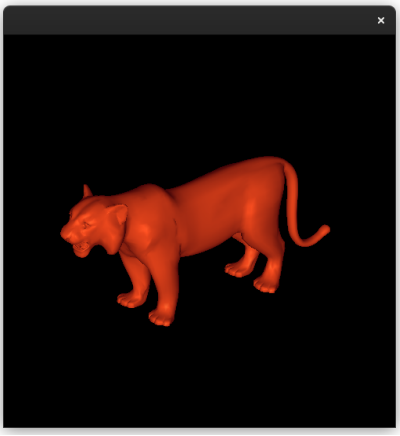
1. As for the light, once the material is defined, one need to give the parameters of this material to OpenGL with the commands presented in Listing 28.

```
void setMaterial()
{
    glMaterialfv(GL_FRONT_AND_BACK, GL_DIFFUSE, opengl.material->diffuse);
    glMaterialfv(GL_FRONT_AND_BACK, GL_SPECULAR, opengl.material->specular);
    glMaterialfv(GL_FRONT_AND_BACK, GL_AMBIENT, opengl.material->ambient);
    glMaterialf(GL_FRONT_AND_BACK, GL_SHININESS, opengl.material->shininess);
}
```

Listing 28: Set the material

2. Call this function in the `initOpengGL()` function.

(8.1) Red plastic



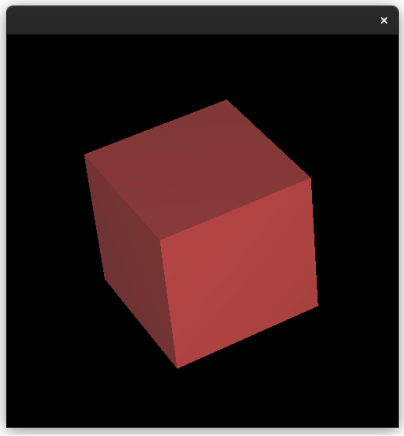
(8.2) Copper

(8.3) Bronze

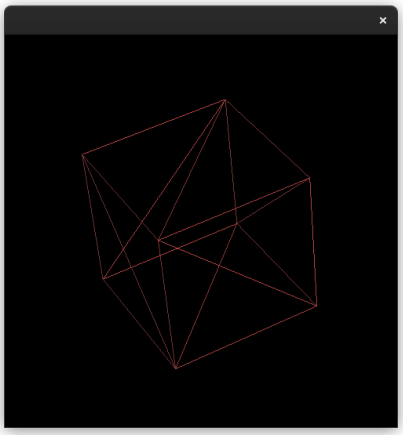
Figure 8: Materials

9.3 Flat shading

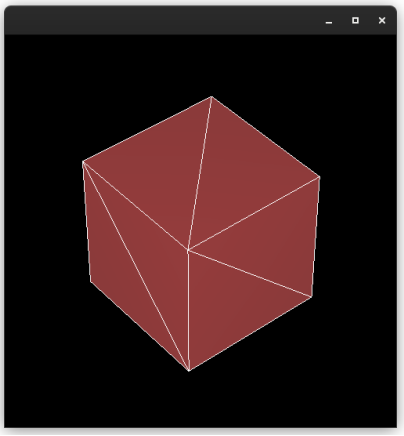
1. The flat shading is enabled using the command `glShadeModel(GL_FLAT)`.
2. The light source need to be enabled before drawing and disabled at the end.
3. Then, when looping on the triangles, you need to give the normal to OpenGL using the command `glNormal3fv`.
4. Write the code for the modes 4, 5 and 6 from Figures 9.4 and 9.5 and 9.4 using the depth test and the polygon offset from previous sections.



(9.4) Flat shading



(9.5) Skeleton with flat shading

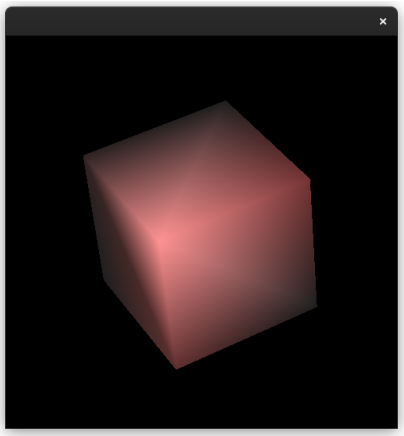


(9.6) Line and flat shading

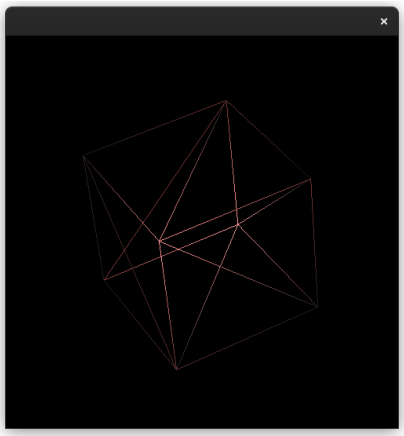
Figure 9: Drawing modes with flat shading

9.4 Phong (or smooth) shading

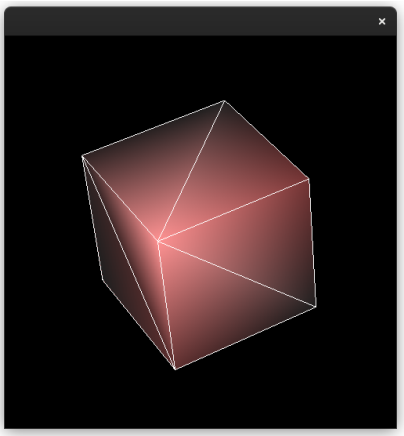
1. The Phong shading is enabled using the command `glShadeModel(GL_SMOOTH)` and instead of giving a normal per triangle, the normal is given by vertices.
2. Build the last three modes (Figures 10.7 and 10.8 and 10.9) using the previous sections and the smooth shading. The three components of the normal of vertices `i` are saved in the `mesh.vertices[i].normal` array.



(10.7) Smooth shading



(10.8) Skeleton with smooth shading



(10.9) Line and smooth shading

Figure 10: Drawing modes with smooth shading

10 Menu

To access more easily to all the previous options one can create a menu usable with the mouse as in Figure 11. The menu ans the submenus are declared as the other events: you need to register them and write the proper function.

1. In a new `menuInit()` function register a function `menu` that will manage the menu.
2. The `void menu(int value)` get the `value` from OpenGL and executes the proper action.
3. The entries of the menu are created with `glutAddMenuEntry("name", value)` where `"name"` is the name of the entry and `value` is a unique value given the `menu` function. For instance, `glutAddMenuEntry("quit", 0)` will create the `"quit"` entry associated to the value `0`. This value needs to be treated in the `menu` function.
4. Add the entries to quit and initialize as when using the keyboard with `'q'` and `'i'`.
5. The submenu for the materials needs to be registered before the main menu with the code from Listing 29 where `subMenuMaterials` is also a function that will treat the various `value` as `menu`.

```
int idx_subMenuMaterials = glutCreateMenu(subMenuMaterials);
```

Listing 29: Create the submenu for materials

(a) The you can add the entries for the various materials as previously (with `glutAddMenuEntry`) and treat them in a `void subMenuMaterials(int value)` function.

(b) The submenu is added to the main menu with the command from Listing 30.

```
glutAddSubMenu("materials", idx_subMenuMaterials);
```

Listing 30: Add the submenu for materials

6. Finally, the menu is attached to the middle click of the mouse with: `glutAttachMenu(GLUT_MIDDLE_BUTTON)`.

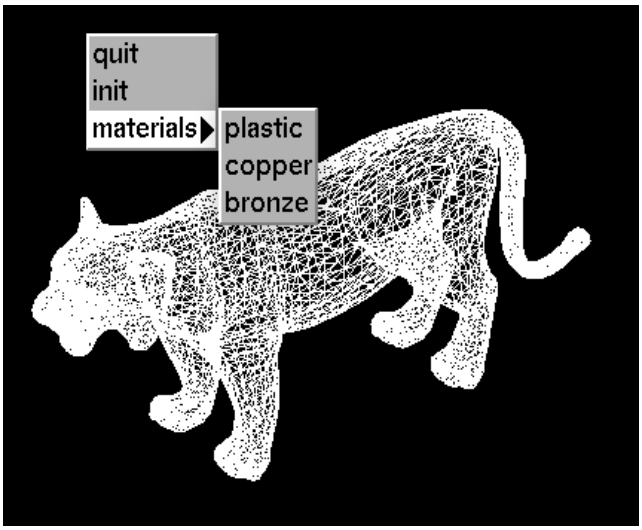


Figure 11: Menu

11 Shrink

To easily detects visually some defaults in a mesh, it can be useful to reduce the size of the elements as in Figure 5.2.

1. In order to do so, one need to save in the `struct s_mesh` the centroid of each triangles. Then, you need to write a function to compute the coordinates of all centroids and then call it in the `initModel()` function.
2. A real `shrink` parameter, needs to be added in the OpenGL structure and initialized in the proper function.
3. This parameter is increases/decreased when pressing on `'s'/'S'`. Add this functionality to the `keyboard` function.
4. Finally, when drawing, instead of giving the coordinates of the vertices `x` you give the coordinates reduced by the `shrink` factor.