TP1-2 - Hello OpenGL!

A gentle introduction to OpenGL.

Computer Graphics,
2nd year, Multimedia track

Sessions 1-2

Version: v2025.1-rc2
(master @ 5458204 2025-03-11)

Contents

1 Objectives

The main objective of this TP is to get you familiar with the OpenGL libraries and learn the rendering pipeline of the library. We will start with a simple program that shows the basic structure of an OpenGL program, and we will analyse the main functions. In the second part of the TP we will see through a simple program how an OpenGL scene is modelled and the various transformations that can move the 3D objects and modify the rendering of the scene.

At the end of the TP you will be able to:

- set up the OpenGL environment
- use some basic 3D primitives to draw object in the scene
- place the camera and the 3D objects in the scene and move them around
- create an interactive program (user interface and user input)

1.1 Assignments

This tutorial comes with the code samples that will be discussed in this document. At the end of each section or subsection of the document you will be asked to answer some questions. Some of them will require you to slightly modify the code, others are more theoretical questions that will help you to verify what you have learnt. Finally, at the end of the tutorial you will be asked to write a program on your own.

Concerning the evaluation, **in group of two** write a short report with your answers to the given questions. The report can be very simple and short, what it is really important that your answers show that you have understood and thought about it (motivate the answers). Check the instructions on Moodle for the submission.

2 OpenGL

OpenGL is a graphic library (Open Graphic Library) and it is designed as a software interface to the graphics hardware on your computer: it actually defines an abstract, platform-independent API (Application Program Interface) rather than a real software library. The API is instead implemented in hardware and software by the GPU vendors (*e.g.* NVIDIA, ATI, Intel etc.) in order to better exploit their various machine architectures. This offers the advantage that OpenGL programs are easily portable to a variety of computers with different architectures and different operating systems. For this reason, OpenGL provides only basic functions to support and control the rendering, as well as primitives to draw geometric primitives such as points, lines and polygons. OpenGL doesn't provide functionalities to support windows or user interaction (like keyboard presses or mouse actions): these features, which are platform-dependent, are instead provided by a separate library called GLUT (the OpenGL Utility Toolkit). GLUT also provides some higher-level features, such as more complex geometric objects, as we will see in this TP. Finally, The OpenGL Utility Library (GLU) contains several routines that use lower-level OpenGL functions to perform tasks such as setting up matrices for specific viewing orientations and projections, performing polygon tessellation, and rendering surfaces. This library is provided as part of your OpenGL implementation.

OpenGL is a "state machine", which means that you specify various states or modes which remain in effect until changed. For example, we will see that the current color used for drawing the primitives is a state variable. You can set the current color to white, red, or any other

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

color, and thereafter every object is drawn with that color until you set the current color to something else. The current color is only one of many state variables that OpenGL preserves. Other state variables control, for example, the current viewing and projection transformations, as we will see later in this TP. We will see these elements through this and the next TPs, but it is important to keep in mind the idea of a "state machine" when you work with OpenGL in order to understand the effects of a given function.

2.1 OpenGL references

Some useful resources about OpenGL:

- Official web site of OpenGL, here you can find all the information you need about OpenGL http://www.opengl.org/documentation
- *OpenGL Programming Guide*, aka "The Red Book", is a classic reference book for learning OpenGL, it is available on-line (old editions) http://www.glprogramming.com/red/
- *OpenGL et GLUT: Une Introduction*, by Edmond Boyer (INRIA Grenoble-Rhône–Alpes), in French. http://www.ann.jussieu.fr/hecht/ftp/DEA/OpenGL.pdf
- OpenGL function documentation http://www.opengl.org/sdk/docs/man4/
- GLUT function documentation http://www.opengl.org/resources/libraries/glut/spec3/spec3.html
- Google it!... The easiest way to found the documentation of a function (or help) is to google the name of the function.

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

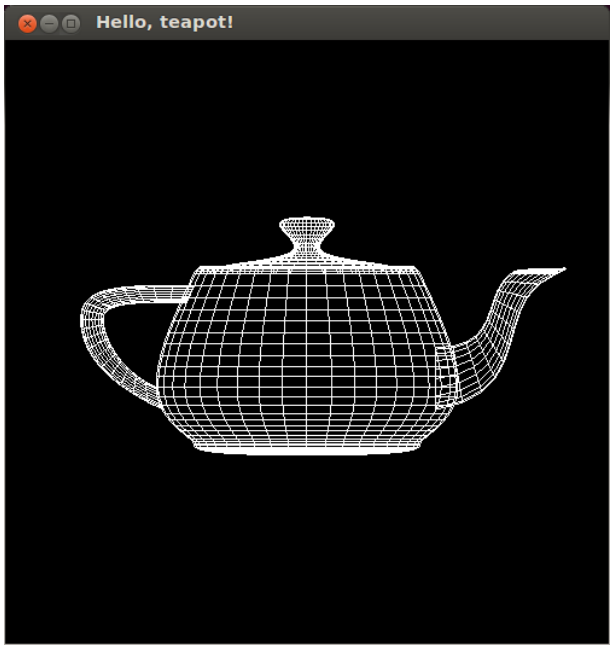


Figure 1: The teapot rendered by `helloteapot.cpp`

3 Hello teapot

We first start with a simple program that you can find inside the file `helloteapot.cpp`. This is the simplest program you can write in OpenGL and yet it will show most of the structure of an OpenGL application.

3.1 Compilation

Follow the instructions detailed in the `README.md` to build the code with CMake according to the operating system you are using (Linux, macOS or Windows).

You only need to run the configuration with CMake once, then you can modify, build and execute the code as explained in the `README.md`

3.2 Anatomy of an OpenGL program

```
1 #include <GL/gl.h>
2 #include <GL/freeglut.h>
```

The two `#include` lines include the OpenGL and GLUT header files. You will always have to include these headers, otherwise your program won't compile. Note that here we are using freeglut, an open source alternative to the original GLUT library, which had some license restrictions. Freeglut provide the same original functions of GLUT with few more new features.

3.2.1 Main

Now consider the `main` routine.

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

```c
// Main routine
int main ( int argc, char * argv[] )
{
    // initialize GLUT, using any commandline parameters passed to the program
    glutInit( &argc, argv );

    // setup the size, position, and display mode for new windows
    glutInitWindowSize( 500, 500 );
    glutInitWindowPosition( 0, 0 );
    glutInitDisplayMode( GLUT_RGB );

    // create and set up a window
    glutCreateWindow("Hello, teapot!");

    // Set up the callback functions for display
    glutDisplayFunc( display );
    // for the keyboard
    glutKeyboardFunc( keyboard );
    // for reshaping
    glutReshapeFunc( reshape );

    // tell GLUT to wait for events
    glutMainLoop();
}
```

`main` typically sets up the windows, event handling (keyboard presses, mouse clicks, window resizes, etc), and does any necessary initialization - everything except actually drawing the scene. Even if it is a simple program, the steps we will see in this `main` are pretty much the steps you'll see in every program:

1. `glutInit` initializes the GLUT library responsible for handling windows and user interaction. As written, you can pass parameters to the initialization routine at runtime by specifying them on the command-line but you don't need to worry about this, as it is meant only for very particular, advanced features.

2. The next batch of commands set up the environment for the creation of windows. They affect the current state, and the settings remain in effect until changed:

   `glutInitWindowSize` and `glutInitWindowPosition` define the size and position (upper left corner) of the window on the screen. If omitted, the default window size is 300×300 and its placement is left up to the window manager.

   `glutInitDisplayMode` specifies the display mode to be used for new windows. `GLUT_RGB` means that the RGB color model will be used. We will see later other options that can be passed.

3. Next, the window is created. `glutCreateWindow` creates a window with the previously-specified size, position, and display mode and gives it the specified title. The window won't actually be displayed, though, until `glutMainLoop` is called.

4. The next lines set up some callback functions.

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

`glutDisplayFunc` sets the display callback for the current window. The display callback is the function that is called each time the window needs to be displayed (such as when it first appears, if it has been resized, if it has been hidden and needs to be refreshed, *etc.*). This doesn't actually call the display callback function - it just registers it so that future display needs will invoke the callback.

`glutKeyboardFunc` sets the keyboard callback for the current window. Every time the user types into the window the specified function (in this case `keyboard`) is called so that the proper action can be executed. We'll see later in Section 3.2.4 how to configure the callback.

`glutReshapeFunc` sets the function that has to be called whenever the window is resized.

5. `glutMainLoop` tells the program to enter the GLUT event processing loop. This just means that the program sits and waits for things to happen, such as window refreshes, window resizes, mouse clicks, key presses, etc. `glutMainLoop` never exits, so it should always be the last line of the main program.

Now let's look at the other functions of the program.

3.2.2 Display

Let's start with the display function.

```
1  // function called everytime the windows is refreshed
2  void display ()
3  {
4      // clear window
5      glClear(GL_COLOR_BUFFER_BIT);
6      // draw a teapot as a wireframe
7      glutWireTeapot(.5);
8      // flush drawing routines to the window
9      glFlush();
10 }
```

This is the function that was registered as the display callback for the window. It can be named anything (`display` is just a convenient choice, not a requirement) but must have a void return value and no parameters. It contains everything that should happen each time the window is drawn:

1. `glClear` clears the current window (i.e. the one for which the callback was triggered). `GL_COLOR_BUFFER_BIT` is used to clear the information about pixel colours. (There is other information associated with what appears in a window, and more options for `glClear` will be seen later.)

2. Next, the geometry is drawn. In this case, the scene contains only a teapot. `glutWireTeapot` render the wire frame of a teapot; the parameter specifies the size. Remember: in OpenGL there are no units, so the value you set can be anything you want (centimetres, meters, kilometres...)

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

3. Finally, use `glFlush` to flush all the drawing commands. OpenGL caches drawing commands for efficiency, so nothing will actually appear on the screen until you call `glFlush`. This should generally be the last thing done by a display callback.

There are other 3D objects and primitives that can be drawn using the GLUT library, like a sphere, a cube, a cone and so on (see Table 1). For each of this element, GLUT provide two versions of the function, the `Wire` and the `Solid` version: as we saw in the example `Wire` generates a wireframe model, in the `Solid` version the faces of the object are not transparent. Each object is created and oriented with respect to a local reference system. For example, the centre of the sphere is placed at the origin of the reference system. We will see later in Section 4 how the objects are placed in the scene and how we can modify their position, size and orientation.

```
void glut[Solid|Wire]Cube( GLdouble size );
void glut[Solid|Wire]Sphere( GLdouble radius, GLint slices, GLint stacks );
void glut[Solid|Wire]Cone( GLdouble base, GLdouble height, GLint meridians, GLint stacks );
void glut[Solid|Wire]Teapot( GLdouble size );
void glut[Solid|Wire]Torus( GLdouble innerRad, GLdouble outerRad, GLint nsides, GLint rings );
```

Table 1: Some 3D drawing primitives provided by GLUT (here the doc).

Exercises

1. Replace `glutWireTeapot` with `glutSolidTeapot`. What do you see? What do you think it is still missing in order to see a more realistic rendering of the teapot?
2. Replace the teapot with a sphere using `glutWireSphere`. The parameter `slices` is the number of 'meridians' around the z-axis to draw, and `stacks` the 'parallels' along the z-axis. Try with different values for each of them and compare the results.

3.2.3 Viewport

The `reshape` function is called every time the window is resized by the user.

```
1 // Function called every time the main window is resized
2 void reshape ( int width, int height )
3 {
4     // define the viewport transformation;
5     glViewport(0, 0, width, height);
6 }
```

The `reshape` function defines what to do when the window is resized. It must have a `void` return type, and takes two `int` parameters, the new `width` and `height` of the window.

The OpenGL function `glViewport` sets the viewport, *i.e.* the area of the window in which the scene will be rendered. The area can be defined passing the `x` and `y` pixel coordinates of the lower left corner and dimensions of the window where the scene will be drawn (see Figure 2):

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

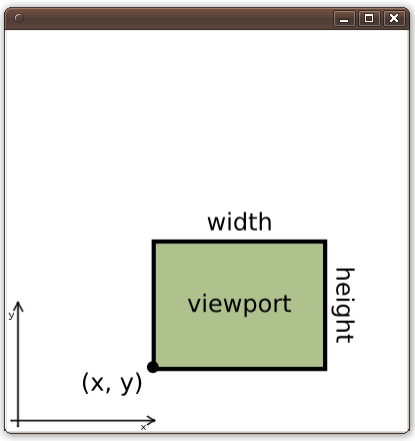


Figure 2: `glViewport` is used to place the rendered scene inside the current window.

```
void glViewport ( GLint x, GLint y, GLsizei width, GLsizei height );
```

Note that `GLint` and `GLsizei` are special OpenGL types that just correspond to the basic C/C++ `int` type.

Exercise

1. Try to manually resize the window. What happens? How can we preserve the aspect ratio of our scene? Change the call to `glViewport` so that every time the windows is resized, the viewport is always a square centred both vertically and horizontally inside the window, and its size is the maximum that can fit the window (*i.e.* its size is the minimum between the width and the height of the window).

3.2.4 Keyboard

Finally let's have a look at the keyboard callback:

```
// Function called everytime a key is pressed
void keyboard( unsigned char key, int x, int y )
{
    switch ( key )
    {
        case 27:
        case 'q':
            exit( EXIT_SUCCESS );
            break;
        default:
            break;
    }
    // marks the current window as needing to be redisplayed.
```


[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

```
14      glutPostRedisplay( );
15  }
```

This is the function that is called every time the user types inside the window. Again, the name is a free choice, but the function must have a void return value and take 3 parameters:

- the char `key` which contains the ASCII code of the pressed key,
- two integers `x` and `y` which contain the position of the mouse pointer inside the window.

Inside the function we can then manage the input provided by the user: in this case, if the user presses the Esc button (whose ASCII code is 27) or Q the program terminates.

Exercise

1. Add a new case for the letter 'w', so that every time it is pressed we can switch from a wireframe teapot to a solid teapot and viceversa.(Hint: you can define and use a global `bool` variable...)

3.3 Summing up

We have seen that the main sequences of an OpenGL program are:

1. Initialization of GLUT, creation and set up of the main window
2. Register the callback function, in particular:
   - the display function which allows us to draw the scene (`glutDisplayFunc`).
   - the reshape function which allows us to set up the viewport (`glutReshapeFunc`).
   - the keyboard function which allows us to manage the user input (`glutKeyboardFunc`).
3. Call `glutMainLoop()` to start the rendering and wait for events to process (*e.g.* a pressed key).

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

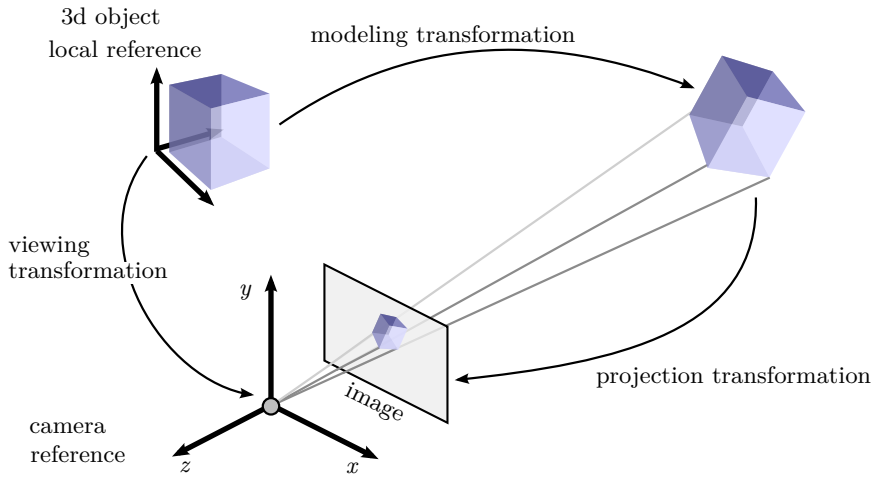


Figure 3: The OpenGL Viewing Pipeline.

4 Modelling an OpenGL scene

The first program `helloteapot` was a toy example that allowed us to understand the structure of an OpenGL program. The output, though, is admittedly disappointing, as we can only guess the shape of the teapot. In this section we will see how to position and orient 3D objects in space and how to establish the viewpoint of the scene so that we can have a better feeling of the three-dimensionality of the scene.

4.1 The OpenGL Viewing Pipeline

The OpenGL Viewing Pipeline is the sequence of transformations that a 3D object (more precisely, the geometric primitives that form the object) undergoes in order to be rendered inside the image, *i.e.* the set of transformations applied to a 3D point in space in order to get its 2D location on the image. In particular in this TP we will focus on 3 main transformations (see Figure 3):

- *the modeling transformation*; every 3D object (*e.g.* the teapot of the previous section, or the cube in Figure 3) is defined in a local reference system, *e.g.* such that one of the vertices of the cube is in the origin of the local reference system. The object can be then modified by applying some transformations such as rotations, translations, and scaling. The ensemble of these transformations is the modeling transformation.
- *the viewing transformation*; the viewing transformation brings the 3D objects in the camera reference system. If we fix the camera position and orientation in space, the viewing transformation is the transformation that allows us to express the 3D object in the camera reference system.
- *the projection transformation*; this is the transformation that projects the 3D points (expressed in the camera reference system) in order to generate the image. This transformation depends on the type of camera we are using, *i.e.* if we are using a perspective camera it will be a perspective transformation, an orthographic transformation in the case of an orthographic camera.

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

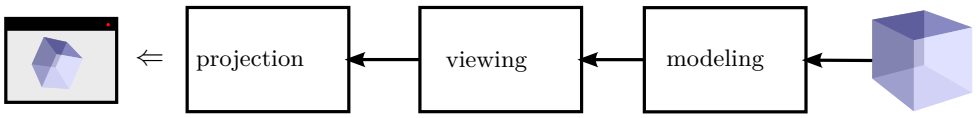


Figure 4: How the transformations and the relevant matrices are applied to a 3D object.

All these transformations are represented in OpenGL in the form of matrices. OpenGL uses the homogeneous coordinates to define these 3D transformations, hence the corresponding matrices are 4×4 matrices. Combining together the transformations corresponds to multiplying the matrices. In particular, in the OpenGL pipeline the corresponding matrices are applied to the 3D model in the order depicted in Figure 4. OpenGL uses 2 different matrices to store the 3 transformations above:

- `GL_PROJECTION` is the matrix that contains the projection transformation.
- `GL_MODELVIEW` is the matrix that is used to contain both the viewing and the modeling transformations. We will see later why the two transformations are stored in a single matrix.

Each of the above matrices (`GL_MODELVIEW` and `GL_PROJECTION`) is maintained by OpenGL as a graphics state of its state machine (as we said in Section 2, OpenGL is a state machine). This simply means that for each of the two matrices there exists a corresponding matrix that contains the current transformation matrix to apply to the objects during the rendering. Whenever we want to change the position of an object (modeling transformation) we need to change the current `GL_MODELVIEW` matrix; if we want to change the position of the camera in the scene (viewing transformation), again, we need to change the current `GL_MODELVIEW`. If we want to change the type of camera we are using or its internal parameters (hence, the projection transformation), we need to modify the `GL_PROJECTION` matrix.

We will see in the next two examples how we can modify these matrices. In particular, in the next example (Section 4.2) we will see how to set up the camera projection and how to place the camera in the scene. Then, in the second example we will see how to move and place the object(s) in the scene (Section 4.3).

4.2 Hello teapot: a new perspective

Let's see how we can set the type of camera to use and how we can place it in the scene in order to see the teapot of `helloteapot.cpp` under a different point of view. Create a copy of `helloteapot.cpp` and name it `helloteapot2.cpp`. See the last section of the README.md for the changes needed to be made in the `CMakeLists.txt`. Then copy the following piece of code just before the `glutMainLoop()` call in `main()`.

```
    // define the projection transformation
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60,1,1,10);

    // define the viewing transformation
    glMatrixMode(GL_MODELVIEW);
```

```
    glLoadIdentity();
    gluLookAt(1.0,2.2,1.0,0.0,0.0,0.0,0.0,1.0,0.0);
```

Now, if you compile and run the code you should see something like Figure 5. As you can see, the view point of the camera has change and now we can see the teapot from another angle. Let's see more in detail the code.

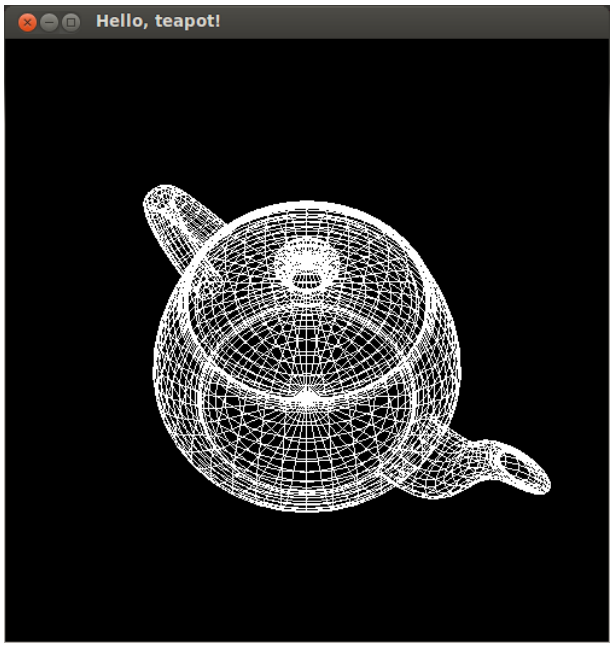


Figure 5: The teapot rendered by `helloteapot2.cpp`

4.2.1 The perspective transformation

The first part of the new code we added specifies the type of projection (*i.e.* the type of camera to use). In this example we use a classic perspective camera.

```
1 glMatrixMode(GL_PROJECTION);
2 glLoadIdentity();
3 gluPerspective(60,1,1,10);
```

The type of projection must be set before any drawing occurs. For this reason we put it in `main` before the GLUT main loop. We could have also placed it after `glClear` in `diplay`, but the projection never changes so there's no need to repeat the steps every time the window is drawn.

- The `glMatrixMode` function specifies which of the viewing pipeline matrices is to be modified: in our case since we are setting the projection transformation we need to modify the `GL_PROJECTION` matrix. `glMatrixMode(GL_PROJECTION)` means that any subsequent command will affect the projection transformation matrix. This is another example of how OpenGL works as a state machine.

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

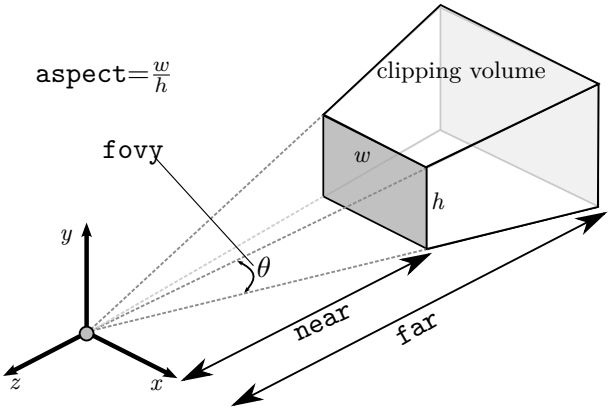


Figure 6: The clipping volume.

- `glLoadIdentity()` reset the current `GL_PROJECTION` matrix. Since it's the first time in the code that we access to that matrix, initially, it is set to some default matrix. After the execution of `glLoadIdentity()` it is now set to the identity.
- With the next call to the function `gluPerspective` we set up the perspective camera with its relevant parameters. The function is defined as (the doc here):

```
1 void gluPerspective( GLdouble fovy, GLdouble aspect, GLdouble zNear, GLdouble zFar );
```

where `fovy` is the angle (in degrees) of the field of view, in the y direction (see Figure 6). `aspect` specifies the aspect ratio of the width of viewer's angle of view to the height (*i.e.* the aspect ratio of the final image). Finally, `zNear` and `zFar` are the distance from the camera center and along the z-axis of the near and far clipping planes, respectively (see Figure 6). The clipping planes limits the portions of the scene that will be rendered in the final image: any object (or parts of object) that lies between the near and far clipping planes are rendered, the other will be discarded. Together with the field of view and the aspect ratio, they define the clipping volume, *i.e.* the volume of the scene that is actually rendered by the camera (the truncated pyramid in Figure 6). This allows to discard all the objects that are not inside the clipping volume during the rendering process.

Given the parameters, `gluPerspective` generates the associate perspective matrix (say, `persp`) and it update the `GL_PROJECTION` matrix by multiplying it by the new perspective matrix `persp`. Figure 7 shows how the content of the projection matrix `GL_PROJECTION` evolves after each function call

Exercises

1. Try to increase and decrease the value of the parameter `fovy` of `gluPerspective`. What happens and why?
2. Try to increase the value of the parameter `zNear` of `gluPerspective`. What happens for values, *e.g.*, greater than 2.1?

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

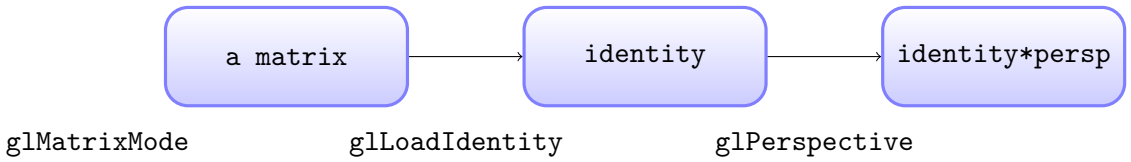


Figure 7: The evolution of the `GL_PROJECTION` matrix after each call.

4.2.2 The viewing transformation

Let's now look at the second part of the new code:

```
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();
gluLookAt(1.0,2.2,1.0,0.0,0.0,0.0,0.0,1.0,0.0);
```

This part of the code sets the camera position and orientation in the scene:

- just like in the previous case, `glMatrixMode` specifies which matrix is to be modified: as explained earlier, the viewing transformation is part of the `GL_MODELVIEW` matrix. Now, any subsequent command will affect the modelview matrix.
- Again, `glLoadIdentity()` reset the current `GL_MODELVIEW` matrix to the identity matrix.
- Finally, `gluLookAt` sets the position of the camera in the scene. The function is defined as (the doc here):

```
void gluLookAt( GLdouble eyeX, GLdouble eyeY, GLdouble eyeZ,          //the viewpoint
                GLdouble centerX, GLdouble centerY, GLdouble centerZ, // look_at
                GLdouble upX, GLdouble upY, GLdouble upZ );           // UP vector
```

This sets the camera position at point O(`eyeX`, `eyeY`, `eyeZ`); the camera will be looking at point A(`centerX`, `centerY`, `centerZ`). The direction OA will be the optical axis of the camera. At this point, in order to define the orientation of the camera, we still have a degree of freedom (*i.e.* we can rotate the camera around the optical axis). The "up-vector" U(`upX`, `upY`, `upZ`) defines the "up" direction of the camera and set the orientation of the camera. In this case the camera will be placed in $A(1.0, 2.2, 1.0)$, it will look at the origin $O(0, 0, 0)$ and the "up-vector" is $U(0, 1, 0)$[1]. The teapot is centered by default in $(0, 0, 0)$. By default the OpenGL camera is placed in $(0, 0, 0)$ and looks at $(0, 0, -1)$ (*i.e.* the negative part of the z-axis) and the up-vector is the y-axis.

Similarly to `gluPerspective`, `gluLookAt` generates the associate viewing transformation (for the sake of the explanation let's call it `viewing`) and it update the `GL_MODELVIEW` matrix by multiplying it by the new perspective matrix viewing. Therefore, the three steps to set the camera position are similar to those of Figure 8.

[1] **NB**: it is not necessary for the "up-vector" to be a normalized vector, or perpendicular to the optical axis. The only constraint is that it must not be parallel to the optical axis, otherwise nothing will be displayed.

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

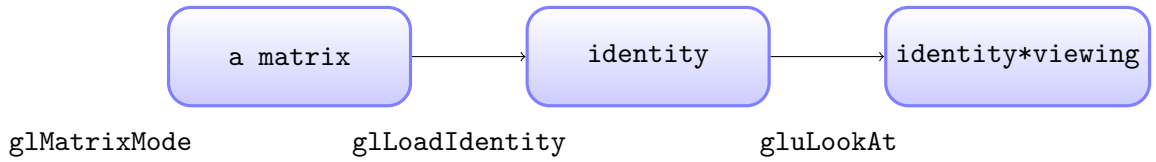
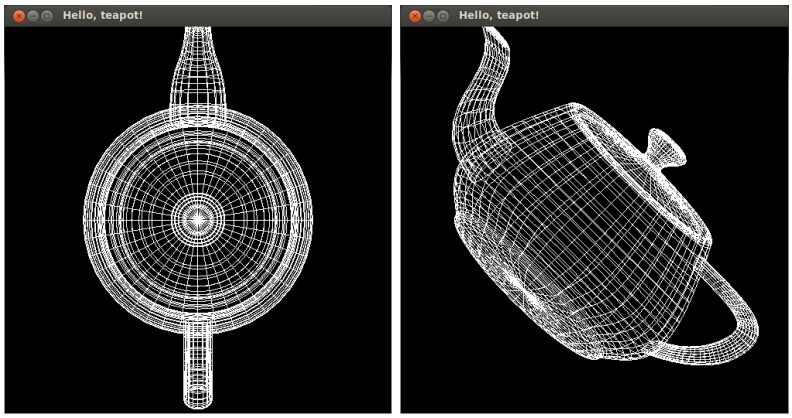


Figure 8: The evolution of the `GL_MODELVIEW` matrix after each call.



(a) (b)

Figure 9: Try to properly set the camera in order to get these views of the teapot.

Exercises

1. What happens if we invert the up vector, *i.e.* we set it to (0,-1,0)?
2. Set the camera so that we can see the teapot as in Figure 9 (a-b)? How do you change the "up-vector"?

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

4.3 Oh no! More teapots...

Now it's time to see how to place and move objects in the scene.

4.3.1 The modeling transformation

As already stated in Section 4.1, the modeling transformations are stored in the `GL_MODELVIEW` matrix. In order to place and move objects in the scene we have to modify this matrix by applying transformations such as rotations, translations and scaling. OpenGL provides the following functions in order to apply transformations to the current matrix:

- `void glTranslatef(GLfloat x, GLfloat y, GLfloat z)` (doc)
  Multiplies the current matrix by a matrix that moves (translates) an object by the given `x`, `y`, and `z` values (or moves the local coordinate system by the same amounts).
- `void glRotatef(GLfloat angle, GLfloat x, GLfloat y, GLfloat z)` (doc)
  Multiplies the current matrix by a matrix that rotates an object (or the local coordinate system) in a counter-clockwise direction about the axis connecting the origin and the point (`x`, `y`, `z`). The angle parameter specifies the angle of rotation in degrees.
- `void glScalef(GLfloat x, GLfloat y, GLfloat z)` (doc)
  Multiplies the current matrix by a matrix that stretches, shrinks, or reflects an object along the axes. Each `x`, `y`, and `z` coordinate of every point in the object is multiplied by the corresponding argument `x`, `y`, or `z`.

Each function acts in a similar way as `gluLookAt` did previously: it multiplies the current matrix by the corresponding transformation matrix (a translation, a rotation or a scaling matrix).

Finally, OpenGL rather than maintaining a single current matrix for `GL_MODELVIEW`, it maintains a stack of (eventually different) matrices. The matrix at the top of the stack is considered the current matrix and it is the one actually used as current matrix. Matrices can be added to top of the stack ("push") and removed from the top of the stack ("pop"). OpenGL provides two functions to this end:

- `void glPushMatrix()` (doc) Adds a copy of the current matrix at the top of the stack.
- `void glPopMatrix()` (doc) Remove the current matrix from the top of the stack, so that the new current matrix is the one below it on the stack.

Let's see in detail how it works and why it is useful with the next example.

4.3.2 Teapots!

Open the file `moreteapots.cpp`, compile it and run it. The window should display a scene like the one in Figure 10. As you can see we have now 3 teapots in 3 different colors, with a different (apparent) size and position in space.

If you look at the code you see that it is the same as the last version of `helloteapot.cpp`, the only difference is in `display` function[2]. Here is the new part of code which is responsible of

[2] Actually, the position and the orientation of the camera is also different, *c.f.* `gluLookAt` in `main()`.

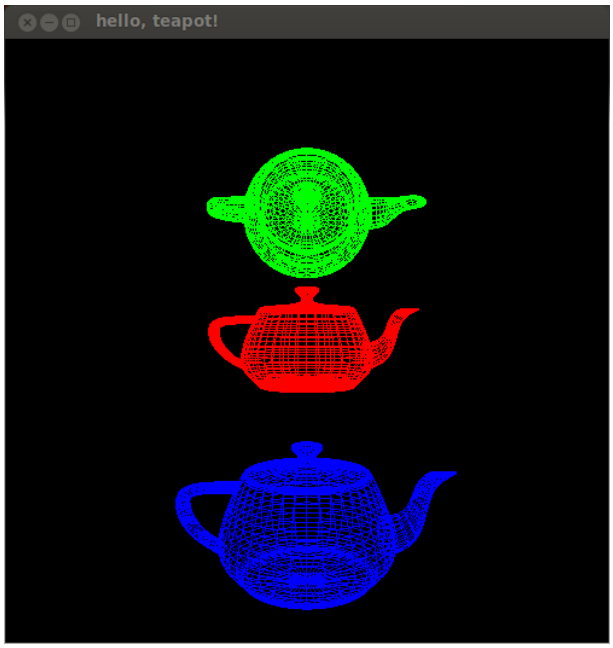


Figure 10: The scene rendered by `moreteapots.cpp`.

drawing the three teapots in different positions (and colors)[3]:

```
glMatrixMode(GL_MODELVIEW);          // set current matrix as GL_MODELVIEW

// draw scene
glPushMatrix();                   // add a copy of the curr. matrix to the stack
  glPushMatrix();                   // add a copy of the curr. matrix to the stack
    glTranslatef(0, 0, -3);             // translate -3 on the z
    glColor3f(1.0f, 0.0f, 0.0f);        // set drawing color to red
    glutWireTeapot( 1 );                // MIDDLE red teapot
    glTranslatef(0, 2, 0);              // translate 2 on the y
    glColor3f(0.0f, 1.0f, 0.0f);        // set drawing color to blue
    glRotatef(90, 1.0f, 0.0f, 0.0f);    // rotate 90 deg around x
    glutWireTeapot( 1 );                // TOP green teapot
  glPopMatrix();                      // pop the current matrix
  glTranslatef(0, -2, -1);            // translate -2 on y and -1 on z
  glColor3f(0.0f, 0.0f, 1.0f);        // set drawing color to blue
  glutWireTeapot( 1 );                // BOTTOM blue teapot
glPopMatrix();                      // pop the current matrix
```

`glMatrixMode` specifies that we are going to work on the `GL_MODELVIEW` matrix. The next part of the code defines the modeling transforms and the scene elements. The middle teapot is drawn at $(0, 0, -3)$, the top teapot is drawn rotated by 90 deg around the x-axis at $(0, 2, -3)$, and the

[3]The code has been re-indented just for the sake of readability, to better visualize the operations on the stack. It is not necessary to indent the code like that, but it could help when you have many objects and you are dealing with several matrices on the stack.

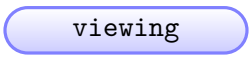
[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

bottom teapot at $(0, -2, -1)$. Why is it so? The transforms accumulate, so the `glRotatef(90, 1.0, 0.0, 0.0)` just before the top teapot is applied to a current matrix which already includes the `glTranslatef(0,2,0)` and `glTranslatef(0,0,-3)` from before the middle teapot. However, the first `glPopMatrix`, combined with the previous `glPushMatrix` means that the matrix involving the transforms for the middle and top teapots is removed before the bottom teapot's `glTranslatef(0,-2,-1)` so this applies to the same matrix that the middle teapot's `glTranslatef(0,0,-3)` does, thus putting the bottom teapot at $(0, -2, -1)$ instead of $(0, 0, -4)$.

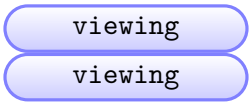
Finally note that we used the function `glColor3f` to set the drawing color for each teapot. The `glColor3f` function sets the current drawing color to the color specified by the 3 parameters, in this case expressed in the RGB components. Each parameter must be a `float` from 0 to 1 to indicate the value of the corresponding RGB component. Again, this is another example of the OpenGL's state machine, once the drawing color is set it remains set until it is further changed: if we comment out the second `glColor3f`, the second teapot will be drawn with the same color as the first one.

Let's see more in detail what happens to the stack of `GL_MODELVIEW` matrices.

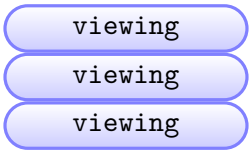
The display callback is not called until `glutMainLoop`, so the rest of main is guaranteed to be executed before anything in display. This means that the `GL_MODELVIEW` matrix stack (representing the combined modeling and viewing transformation steps of the viewing pipeline) contains only one matrix, which is set by `glLookAt` to the corresponding viewing transformation, *e.g.* `viewing` (*c.f.* also Figure 8). Hence the current modelview matrix on the stack is:



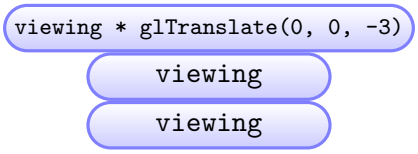
The viewing transform is the current modelview matrix since it is on top. Now, the window is displayed on the screen for the first time and `display` is called. The first `glPushMatrix` pushes a copy of the current matrix on top of the modelview matrix stack, making that copy the new current matrix:



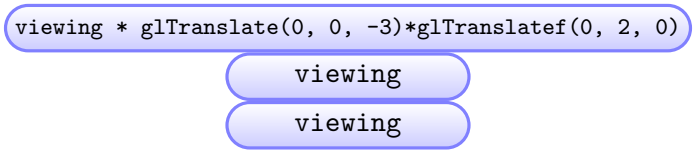
The second `glPushMatrix` does the same thing:



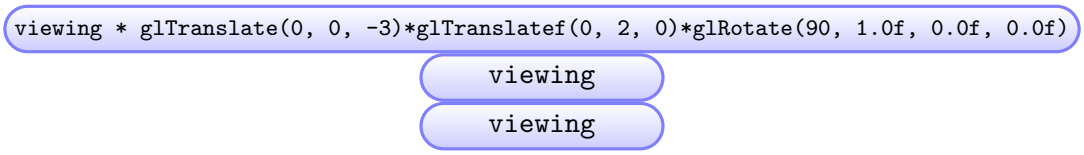
Now, the first translation is combined with the current matrix:



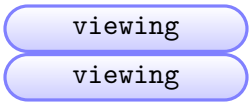
The middle teapot, the red one, is drawn using `viewing transform * glTranslatef(0,0,-3)` as modeling transformation for that teapot. The next translation is again combined with the current matrix:



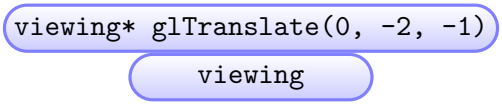
Idem for the next rotation:



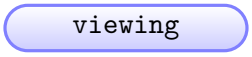
The top green teapot is drawn using this current matrix. Next, the first `glPopMatrix` pops the top matrix from the stack, making the new top matrix the new current matrix:



Thus, the next `glTranslatef(0,-2,-1)` is applied to just the viewing transform and the bottom blue teapot is drawn using the current matrix:



The final `glPopMatrix` restores the modelview stack to the same state as it was when display was first called:



This is very important because the next time `display` is called, everything will be positioned the same way it was the first time.

Order of the Transformations Let's consider the green teapot. When it is drawn, the current transformation matrix, as we have seen, is

`viewing * glTranslate(0, 0, -3)*glTranslatef(0, 2, 0)*glRotate(90, 1, 0, 0)`.

If we apply this transformation matrix, the teapot will first undergo the rotation of 90 degrees, then the translation of 2 on y, and finally the translation of -3 on z. Which is the reverse order of transformations we see in the code! A way to view this sequence of transformations is to imagine that a local reference system is tied to the object you're drawing. All operations occur relative to this coordinate system that moves and rotates in a global reference system.

Back to the green teapot, imagine that the teapot has its own local reference system, in which the origin is at the centre of the teapot. Then `glTranslate(0, 0, -3)` moves this local reference system of the teapot along the z-axis. The next `glTranslatef(0, 2, 0)` occurs along the (now-translated) y-axis. Finally, the rotation occurs about the (translated) origin, so the teapot rotates in place in its position on the axis. With this approach, the matrix multiplications now appear in the natural order in the code. These are two equivalent ways to think about transformations in OpenGL.

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

Moving the camera or the objects? Remember that, to achieve a certain scene composition in the final image, either you can move the camera, or you can move all the objects in the opposite direction. For example, a modeling transformation that rotates an object counter-clockwise is equivalent to a viewing transformation that rotates the camera clockwise.

Finally, keep in mind that the viewing transformation commands must be called before any modeling transformations are performed, so that the modeling transformations take effect on the objects first.

Exercises

1. What happens if the second `glPopMatrix` in display isn't there? Try resizing the window or partially covering it, to force a repaint. Why does this occur? (Consider what happens to the modelview stack.)
2. What happens if the `gluLookAt(0.0,0.0,5.0,0.0,0.0,0.0,0.0,1.0,0.0)` line in main is moved after the second `glPushMatrix` in display? Why does this occur? (Consider what happens to the modelview stack.)
3. What happens instead if the `glLoadIdentity` line in main is moved to just before the "draw scene" comment in display, and why? (Consider what happens to the modelview stack.)

4.4 Summing up

We have seen how to place and move objects and the camera in a OpenGL scene. In particular, remember that:

- OpenGL maintains two different matrices:
  - `GL_PROJECTION`, which models the projection transformation
  - `GL_MODELVIEW`, which models the viewing and the modeling transformations
- `glMatrixMode` allows to specify which of the two matrices we intend to work with.
- `gluPerspective` allows to set the parameters of the perspective projection of the camera (`GL_PROJECTION` mode)
- `gluLookAt` allows to set the camera position and orientation in the scene (`GL_MODELVIEW` mode).
- `glTranslatef`, `glRotatef`, `glScalef` allows to modify the current matrix by applying the relevant transformation matrix in order to move and place objects in the scene (`GL_MODELVIEW` mode).
- we can manage a stack of matrices using `glPushMatrix` and `glPopMatrix`.

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

Final Exercise – Navigator

We want to build a "navigator" that moves the camera in the scene. Create a new file named `navigator.cpp`. The program must show the same objects with the same mutual position of the `moreteapots.cpp`. The navigator then must behave as follows:

- we move the camera in the scene using the classic WASD keyboard pattern:
  - W and S will make the camera move forward and backward on the z-axis, respectively;
  - A and D will make the camera translate on the x-axis, on the left and on the right, respectively;
- we want to move the camera up and down along the y-axis using the key Q and Z, respectively;
- we want to orbit around the objects using the arrow keys of the keyboard:
  - ← and → will make the camera rotate around the y-axis, on the left and on the right, respectively;
  - ↑ and ↓ will make the camera rotate around the x-axis, upwards and downwards, respectively;

Some hints:

- if you are using a AZERTY keyboard you can use the ZQSD pattern and AW for the movement on the z-axis. For the QWERTY keyboard, note that Q was previously used to quit the program, now only Esc will terminate the program;
- define 3 global variables that store the location of the camera (for x,y,z respectively) and two others for the angular orientations (one for each axis). Their values has to be updated accordingly every time the corresponding key is pressed.
- to ease your work, first try to implement only the WASD and QZ movements of the camera. Whenever it works, add the second part about the rotation.
- define two constants that define the increment / decrement of the position and of the angular orientation, *e.g.*:

  ```
  #define SPEED 0.1 //OpenGL unit
  #define ANG_SPEED 0.5 //degrees
  ```

  This value will be added / subtracted accordingly every time the corresponding key is pressed. Experiment with different values to find an optimal value.
- in order to process the arrow keys, GLUT provides a special callback, `glutSpecialFunc`. Here is the documentation. It works exactly like `glutKeyboardFunc` but it is called only when some special key of the keyboard (*e.g.* the arrows) are pressed. Register the callback in `main` and implement a function that manage the arrow keys (`GLUT_KEY_LEFT` and so on).

[git] • Branch: master @ 5458204 • Release: v2025.1-rc2 (2025-03-11)
Head tags: v2025.1-rc2)

In order to add `navigator.cpp` to the building system, add the file to the tp directory, then edit the `CMakeLists.txt` file and uncomment the last three lines and ave it. Finally, in the `build` directory rerun `cmake ..` and you are good to compile the source file. On Windows, close VS before performing the above operation and then reload the solution file.