Yet another OpenGL course

Sommaire

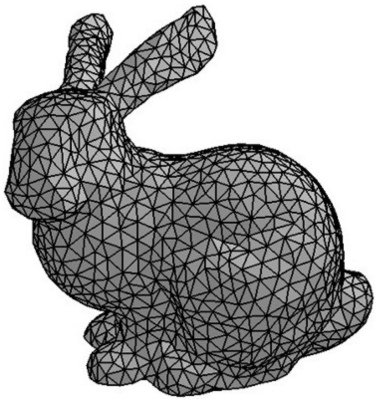
1 Introduction

Welcome to the wonderful land of computer graphics. This course will introduce basic knowledge related to image synthesis and common tools used in the open source community such as OpenGL and the GLUT. The goal here is not to give a detailed description of OpenGL or any particular library, for this matter there is online documentation.

The course outline will be kind of backward compared to traditional courses, I will directly present some examples of codes and their results and only then I will give a more general presentation of the underlying concepts behind the examples.

Image synthesis can be divided into to specialties, modeling and rendering. Modeling is the art of representing and manipulate 3D objects whereas rendering focus on generating a synthetic image from these objects. The first part of the course is going to be more related to rendering, we will learn how to use OpenGL in order to display some basic shapes like triangles or cubes, shade them, setup the camera etc.

But what is OpenGL actually ? OpenGL stands for open graphic library, therefore it is a library used to display 2D or 3D objects. Unlike the Windows counterpart (DirectX) OpenGL is multi-platform (Linux, Mac, Windows and more.) and multi-language (we will use the C implementation but we could use java, python, C# etc.). OpenGL will enable us to command the graphic processor in order to display our objects. Nowadays most personal computers has their dedicated graphic processor unit (GPU). These processor's architecture differs largely from the traditional CPU. There are specialized for one single task: rendering triangles. The task of an API such as OpenGL is to command the GPU by sending it programs to execute from the CPU. Therefore OpenGL is an extremely low level library just above the GPU driver. Common tasks to do with OpenGL is to render triangle mesh, compute lighting on triangles, doing some mirror effects, use textures on meshes etc. What OpenGL is not about is: window management, modeling complex meshes, handling keyboard, mouse, sound.



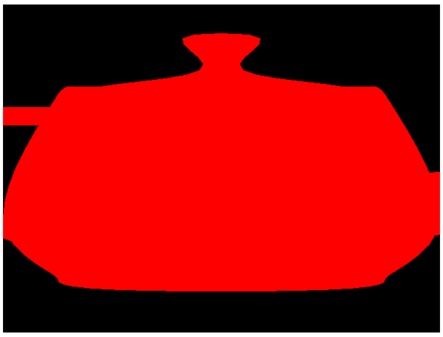
The stanford bunny is a famous triangle mesh used for testing purpose

2 An introduction to GLUT

2.1 Creating a window

Before programming your awesome Crisis© like 3D engine you need two create a window in which OpenGL can draw into. To do so, we are going to use the GLUT library. GLUT stands for OpenGL utility toolkit and enables the creation of a window. It also handles mouse and keystrokes events. Some basic geometries are also available such as cubes, cylinders or even a teapot. GLUT is the easiest way to create minimal example of OpenGL application so lets look at some code:

```
#include <stdlib.h>
#include <stdio.h>
#include <GL/glut.h>

/* Rendering the window */
void display(void){
  /* set the clear color */
  glClearColor(0.f, 0.f, 0.f, 1.f);
  /* clear the color buffer (i.e. clear the pixels)*/
  glClear(GL_COLOR_BUFFER_BIT);
  /* Define the color used to draw the next object */
  glColor3f(1., 0., 0.);
  /* Draw the teapot */
  glutSolidTeapot( 1. );
  /* Force the driver to execute the openGL commands that remain in the command buffer*/
  glFlush();
}

int main (int argc, char** argv){
  /* Initialize glut */
  glutInit(&argc, argv);
  /* set single buffering mode and RGB colors */
  glutInitDisplayMode(GLUT_SINGLE | GLUT_RGB);

  /* Create the window */
  glutInitWindowSize(640, 480);
  glutInitWindowPosition(240, 212);
  glutCreateWindow("My fancy window");

  /* Set the callback for the display function */
  glutDisplayFunc(display);
  /* Lauch glut's main loop */
  glutMainLoop();

  return 0;
}
```



Result

This code simply displays a red teapot on a black screen. Now lets describe the code step by step starting from the main: 'glutInit()' initialize the GLUT library. We then need to specify which type of buffer will be used to display the OpenGL results with 'glutInitDisplayMode()', GLUT_SINGLE means we are using a single buffer as opposed to double buffering (I will explain that later). 'glutInitWindowSize()' set respectively the width and the height of the window and 'glutWindowPosition()' its position according to the upper left corner of the screen. The window is finally created with 'glutCeateWindow()' but not displayed, the name of the window is also passed at the same time.

Now we need more attention on 'glutDisplayFunc()'. This takes in parameter a function, or more precisely a pointer to a function. Here we pass the pointer to our function 'display()' just by writing its name as a parameter. GLUT will then be able to remember this function and call it as much as it wants. Actually GLUT will call the function 'display()' each time the window needs to be refreshed (Some objects like other windows may hide the GLUT window and the window manager will tell GLUT to redraw its window).

The technique used here is is called a 'callback', we pass a function to the GLUT which will be later called back by the library. To use 'glutDisplayFunc()' you just have to write your own function which must be of type 'void someName(void)' and pass 'someName' as a parameter as you would pass an int or a float. If you want to learn more about pointer function just read the annex on the topic.

It is worth noticing that any call to an OpenGL function before the initialization of the GLUT will result in a undefined behavior. These are very hard bugs to detect so do yourself a favor and never forget that you can't call any OpenGL primitives (that is function beginning with a 'gl' prefix) before initializing the GLUT. Your application might work but you will end up with a moody software with OpenGL functions not doing what there are supposed to.

At the end of the GLUT setup we call the 'glutMainLoop()' which will effectively display the window. Once this function is called GLUT will enter in an infinite loop and call the function we've just passed as a parameter to fill the window buffer.

So now the interesting part with the 'display()' function. All the calls beginning by 'gl' are actual OpenGL functions. First we set the color used to clear the image buffer with 'glCLearColor()' the parameter are respectively the RGB (red, green, blue) colors and the transparency, all are floats between [0 1]. Here we setup a black color (0., 0., 0.) with no transparency (1.). The call to 'glCLear()' will write every pixels of the image buffer with the color we've just defined with 'glClearColor()'. There is several buffer used to draw and when calling 'glClear()' you must specify which buffer you want to clear, here its our color buffer so simply put 'GL_COLOR_BUFFER_BIT'.

'glColor3f()' setup the color which will be used to draw the next object here we choose red. 'glutSolidTeapot()' is not an OpenGL primitive but a GLUT call, I'm sure you have guessed every call to the GLUT its prefixed 'glut'. It will draw a teapot of size one by calling OpenGL functions. This is useful to test an application without having to load complex objects from a file.

Finally we call 'glFlush()' to tell OpenGL to draw our commands ('gl' functions) to the screen.

Hey! That's it? Why is it the image looking so crappy... Shouldn't we have a more 3Dish image ? The thing is, we need to setup a lot of things before getting something averagely good. We need to setup the camera position and characteristics like its aperture. We also need light to shade the teapot. By default there is no light (that's why the teapot looks uniformly red) and the camera is set to an orthographic mode (here the teapot should look a little less larger with a perspective projection). Be patient I'll explain it all, for now I want to focus on the GLUT.

2.2 Handling events

Ok first example had a big drawback we can't quit the application! We need to bound a key to the function 'exit(0)'. Once again we are going to use callbacks, because GLUT will call at each event (mouse keyboard) the function we specify with 'glutKeyboardFunc()'. Just add the line 'glutKeyboardFunc(key)' after the 'glutInit...()' and add this function to the file :

```
/* Handling of key events */
void key (unsigned char c, int mouseX, int mouseY) {
  switch (c) {
  /* 'q' : quit  the application */
  case 'q' :
    exit (0); break;
  }
}
```

'glutKeyboardFunc()' only takes functions of type 'void some˙name(unsigned char, in, int)' the function will be called each time a key is pressed. GLUT will call the function 'key()' and pass the character pressed to the first parameter and the mouse position to the other two.

Ok now its time to give the list of GLUT's function used to defined the callbacks (for keys mouse events etc.)

```
void glutDisplayFunc( void (*f)(void) );
void glutReshapeFunc( void (*f)(int width, int height) );
void glutKeyboardFunc( void (*f)(unsigned int key, int x, int y) );
void glutMouseFunc( void (*f)(int button, int state, int x, int y) );
void glutMotionFunc( void (*f)(int x, int y ) );
void glutSpecialFunc( void (*f)(int key, int x, int y) );
void glutIdleFunc( void (*f) ( void ) );
```

If your not used to pointer function the first glance might be a bit scary. I recommend the annex on pointer function if the list is unreadable to you.

So what's new? 'glutReshapeFunc()' takes into parameter the function called each time you re-size the window. The function will be called with the length and height of the new window. 'glutMouseFunc()' defines the callback for the mouse buttons pressed events, 'glutMotionFunc()' callback on the mouse movements events, 'glutSpecialFunc()' defines the callback for the special keys events (F1, F2, arrows etc.). Finally 'glutIdleFunc()' defines the callback when there is no events.

Here is an example of function you could pass to 'glutSpecialFunc()':

```
void specialKeys(int key, int x /* mouse abscissa*/, int y /* mouse ordinates */)
{
    switch(key) {
        case GLUT_KEY_F1 :  break;
        case GLUT_KEY_F2 : /* Do something */ break;
        case GLUT_KEY_F3 : /* Do something */ break;
    }
    glutPostRedisplay();
}
```

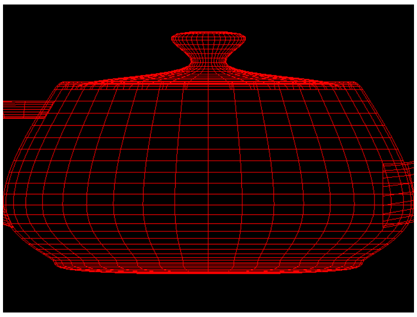
The 'glutPostRedisplay()' function tells GLUT to redraw the window by calling the display callback. There is plenty of documentation and good tutorials about GLUT and events, to get more details about GLUT and macros such as GLUT_KEY_F1, mouse handling... Google is your friend.

2.3 GLUT objects

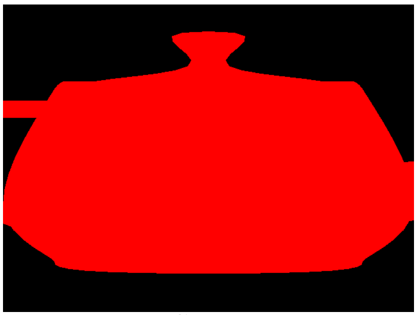
The GLUT offers predefined geometric primitives useful for tests :

```
void glut{Wire Solid}Cube( GLdouble size);
void glut{Wire Solid}Sphere( GLdouble radius, GLint slices, GLint stacks);
void glut{Wire Solid}Cone( GLdouble base, GLdouble height, GLint slices, GLint stacks);
void glut{Wire Solid}Torus( GLdouble innerRadius, GLdouble outerRadius, GLint nsides,
GLint rings);
void glut{Wire Solid}Teapot( GLdouble size);
void glut{Wire Solid}Tetrahedron(void);
void glut{Wire Solid}Decahedron(void);
```

You can choose either Wire or Solid to draw the object:



Wire



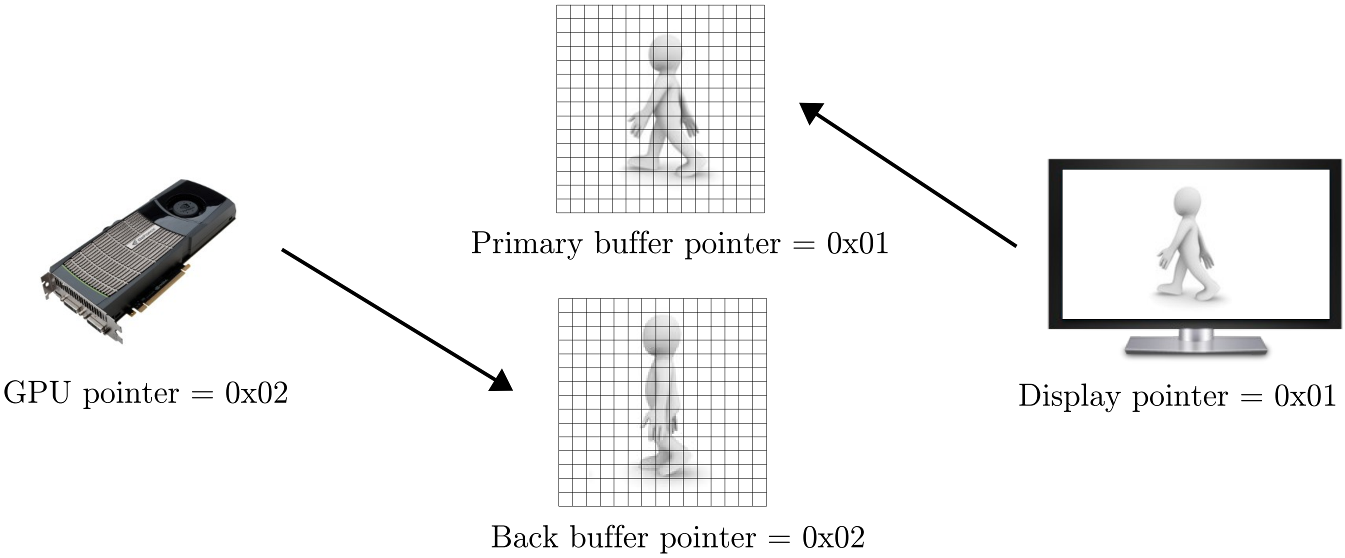
Solid

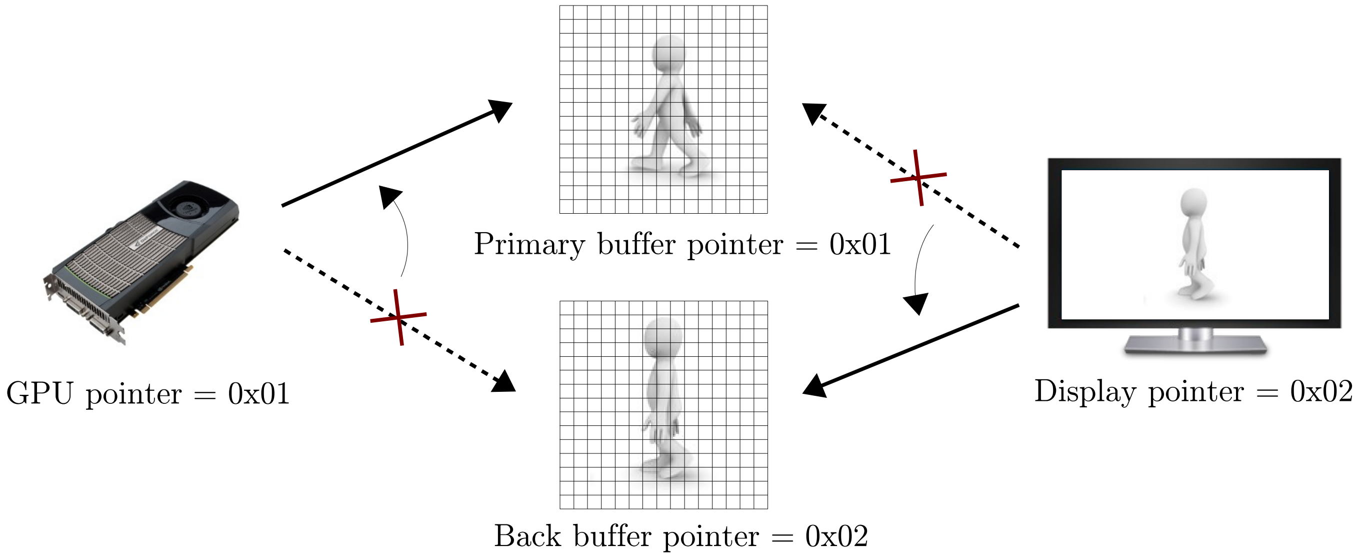
2.4 Double buffering

You may have come across the term double buffering playing video games. Now its time to finally understand what it is!

Double buffering is a technique which consist to store the image in two buffers (or arrays if you fill uncomfortable with the term buffer). We use this method to avoid flickering when the image is displayed. What happens in simple buffering is that the graphic processor is computing at a varying frame-rate the synthetic image. It is very likely that your computer screen is not synchronized with the frame-rate of the graphic processor and when reading the single buffer half of the new image only have been written out. The result is the feeling of a flickering image.

With two buffers we can easily write to the first buffer and the screen can read from the second. When the image is entirely drawn we can swap the buffer. Swapping is done just by exchanging pointer address of the buffer to be read and and the buffer to be written, therefore the swapping is almost instantaneous and the flickering artifact avoided.





GLUT can handle double buffering easily, here are the changes:

```
void display(void){
  glClearColor(0.f, 0.f, 0.f, 1.f);
  glClear(GL_COLOR_BUFFER_BIT);
  glColor3f(1., 0., 0.);
  glutWireTeapot( 1. );
  /* Instead of glFlush() we use: */
  glutSwapBuffers();
  /* It will swap the buffers and then call glFlush() */
}

int main (int argc, char** argv){
  glutInit(&argc, argv);
  /* Replace GLUT_SINGLE by GLUT_DOUBLE */
  glutInitDisplayMode(GLUT_DOUBLE| GLUT_RGB);
  glutInitWindowSize(640, 480);
  glutInitWindowPosition(240, 212);
  glutCreateWindow("My fancy window");
  glutDisplayFunc(display);
  glutMainLoop();
  return 0;
}
```

3 OpenGL first taste

3.1 nomenclature

In order to not feel overwhelmed by the OpenGL functions it is important to know what rules the names of the functions follows. An OpenGL primitive will always be written with this form:

glSomeFunction–2,3,4˝–s,i,f,d˝[v] (TYPE coords);

The 'gl' prefix is append to the function name, then the numbers –2,3,4˝ indicates how many arguments the function takes (for instance a 2D point will be '2' for the x and y coordinates) –s,i,f,d˝ is here to tell the arguments types (s: short, I: int, f: float, d: double). Sometimes 'v' can be added which means parameters are passed as an array. There is often multiple version of the same function :

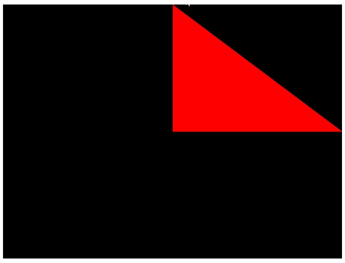
```
float pos[3] = {1.f, 1.f, 1.f};
glVertex3fv(pos);
glVertex3f(1.f, 1.f, 1.f);
glVertex2f(1.f, 1.f);
/* And so on. */
```

Macros and enum field will also begin by 'GL_...' as in the 'glClear(GL_COLOR_BUFFER˙BIT);' we have seen earlier.

3.2 Direct mode drawing

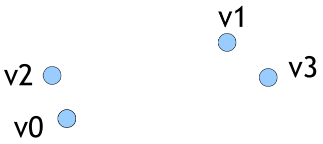
With OpenGL you can draw 3D objects using what is called the 'direct mode'. Before explaining where the name direct mode come from lets draw a triangle with this method:

```
void display(void){
  glClearColor(0.f, 0.f, 0.f, 1.f);
  glClear(GL_COLOR_BUFFER_BIT);

  glColor3f(1., 0., 0.);
  /* Draw a triangle */
  glBegin(GL_TRIANGLES);
  /* Defining a 3D (x,y,z) point of coordinates (0, 0, 0) */
  glVertex3f(0.f, 0.f, 0.f);
  /* Second point of coordinates (1, 0, 0) */
  glVertex3f(1.f, 0.f, 0.f);
  /* Last point of coordinates (0, 1, 0) */
  glVertex3f(0.f, 1.f, 0.f);
  glEnd();

  glFlush();
}
```

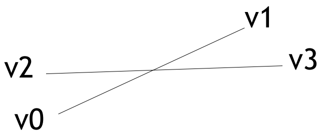


Result

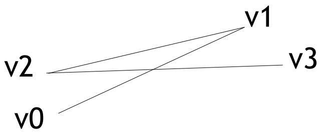
Note that 'glVertex...()' must always be called between a 'glBegin()' and a 'glEnd()'. A triangle will be drawn every three calls of glVertex...(). You can try other primitives such as lines points quads etc. I illustrate below the use of the different primitives available with OpenGL :



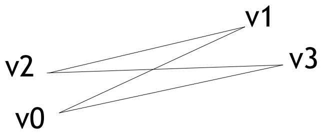
```
glBegin(GL_POINTS);
glVertex3f(v0x, v0y, v0z);
glVertex3f(v1x, v1y, v1z);
glVertex3f(v2x, v2y, v2z);
glVertex3f(v3x, v3y, v3z);
glEnd();
```



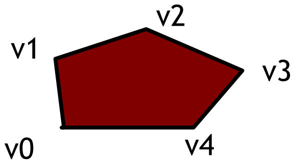
```
glBegin(GL_LINES);
glVertex3f(v0x, v0y, v0z);
glVertex3f(v1x, v1y, v1z);
glVertex3f(v2x, v2y, v2z);
glVertex3f(v3x, v3y, v3z);
glEnd();
```



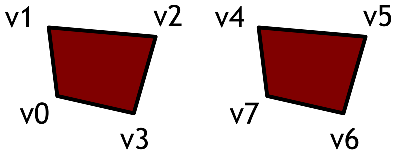
```
glBegin(GL_LINE_STRIP);
glVertex3f(v0x, v0y, v0z);
glVertex3f(v1x, v1y, v1z);
glVertex3f(v2x, v2y, v2z);
glVertex3f(v3x, v3y, v3z);
glEnd();
```



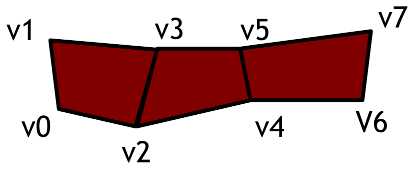
```
glBegin(GL_LINE_LOOP);
glVertex3f(v0x, v0y, v0z);
glVertex3f(v1x, v1y, v1z);
glVertex3f(v2x, v2y, v2z);
glVertex3f(v3x, v3y, v3z);
glEnd();
```



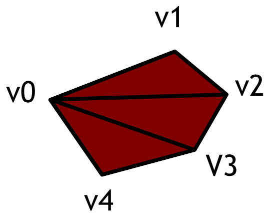
```
glBegin(GL_POLYGON);
glVertex3f(v0x, v0y, v0z);
glVertex3f(v1x, v1y, v1z);
glVertex3f(v2x, v2y, v2z);
glVertex3f(v3x, v3y, v3z);
glVertex3f(v4x, v4y, v4z);
glEnd();
```



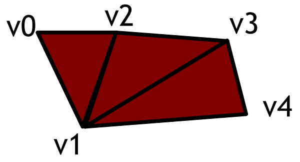
```
glBegin(GL_QUADS);
glVertex3f(v0x, v0y, v0z);
glVertex3f(v1x, v1y, v1z);
glVertex3f(v2x, v2y, v2z);
glVertex3f(v3x, v3y, v3z);

glVertex3f(v4x, v4y, v4z);
glVertex3f(v5x, v5y, v5z);
glVertex3f(v6x, v6y, v6z);
glVertex3f(v7x, v7y, v7z);
glEnd();
```



```
glBegin(GL_QUAD_STRIP);
glVertex3f(v0x, v0y, v0z);
glVertex3f(v1x, v1y, v1z);
glVertex3f(v2x, v2y, v2z);
glVertex3f(v3x, v3y, v3z);
glVertex3f(v4x, v4y, v4z);
glVertex3f(v5x, v5y, v5z);
glVertex3f(v6x, v6y, v6z);
glVertex3f(v7x, v7y, v7z);
glEnd();
```



```
glBegin(GL_TRIANGLE_FAN);
glVertex3f(v0x, v0y, v0z);
glVertex3f(v1x, v1y, v1z);
glVertex3f(v2x, v2y, v2z);
glVertex3f(v3x, v3y, v3z);
glVertex3f(v4x, v4y, v4z);
glEnd();
```



```
glBegin(GL_TRIANGLE_STRIP);
glVertex3f(v0x, v0y, v0z);
glVertex3f(v1x, v1y, v1z);
glVertex3f(v2x, v2y, v2z);
glVertex3f(v3x, v3y, v3z);
glVertex3f(v4x, v4y, v4z);
glEnd();
```

This way of drawing objects is 'unfortunately' deprecated since OpenGL 3.1 (released on March 24, 2009), however it is still available on most graphic drivers even with OpenGL 4.0. Direct mode is also a convenient way to test programs and is easy to understand when learning the basics of OpenGL. That's why I have chosen to present this anyway.

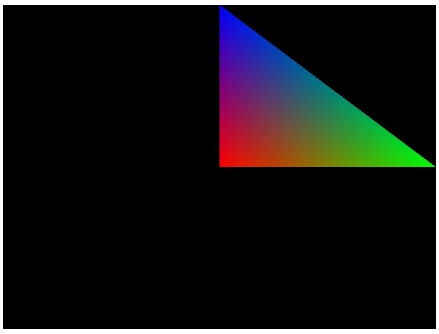
Why it is deprecated ? Well, it is extremely slow. Assume you want to render millions of triangles and each triangle is drawn with a call to three 'glVertex()' this mean you will have three millions of calls to the 'glVertex()' function. The sum of these calls are very costly. There is another problem, the OpenGL program is running on your processor and is using the main memory. The graphic processor (GPU) has no access to the main memory and OpenGL has to upload all the vertex coordinates to the

GPU memory. This transfer has also a cost even thought bandwidth between GPU and CPU is large. Drawing with direct mode forces OpenGL to upload the data to the GPU each time the 'glBegin()' and 'glEnd()' are called even if the geometry of the object is the same. People in charge of OpenGL choose to deprecate these functions in order to foster good practice. Bad practice leads to inefficient OpenGL programms and imply bad publicity for the OpenGL API.

Later on this course we will see how we can upload 3D objects to the GPU and draw them 'remotely' from the GPU and not directly on the CPU as we are doing here.

A last example with the triangle to demonstrate that we can attribute to each vertex different colors:

```
void display(void){
    glClearColor(0.f, 0.f, 0.f, 1.f);
    glClear(GL_COLOR_BUFFER_BIT);

    /* Draw a triangle */
    glBegin(GL_TRIANGLES);
    glColor3f(1., 0., 0.);
    glVertex3f(0.f, 0.f, 0.f);
    glColor3f(0., 1., 0.);
    glVertex3f(1.f, 0.f, 0.f);
    glColor3f(0., 0., 1.);
    glVertex3f(0.f, 1.f, 0.f);
    glEnd();

    glFlush();
}
```



Result

We see that OpenGL 'interpolates' color between the vertices. The color transition is smooth between every vertices. You will learn more about this interpolation per vertex in the OpenGL pipeline chapter.

3.3 Setup the camera

Until now we didn't really cared about the camera, that is to say from which point of view the scene is generated and which are the camera characteristics such as projection, aperture etc. Setting up the camera can be kind of a hassle with openGL primitives, fortunately, convenient functions are available in the GLU library. GLU is on top of OpenGL and provide higher level functions such as 'gluPersepective()' and 'gluLookAt()' to set the camera settings. GLU basicly calls OpenGL primitive and can save you some time. So, 'gluPersepective()' will be used to setup the perspective projection parameters, 'gluLookAt()' will help placing the camera position in the scene. As usual I'll give you an example and then the detailed explanation:

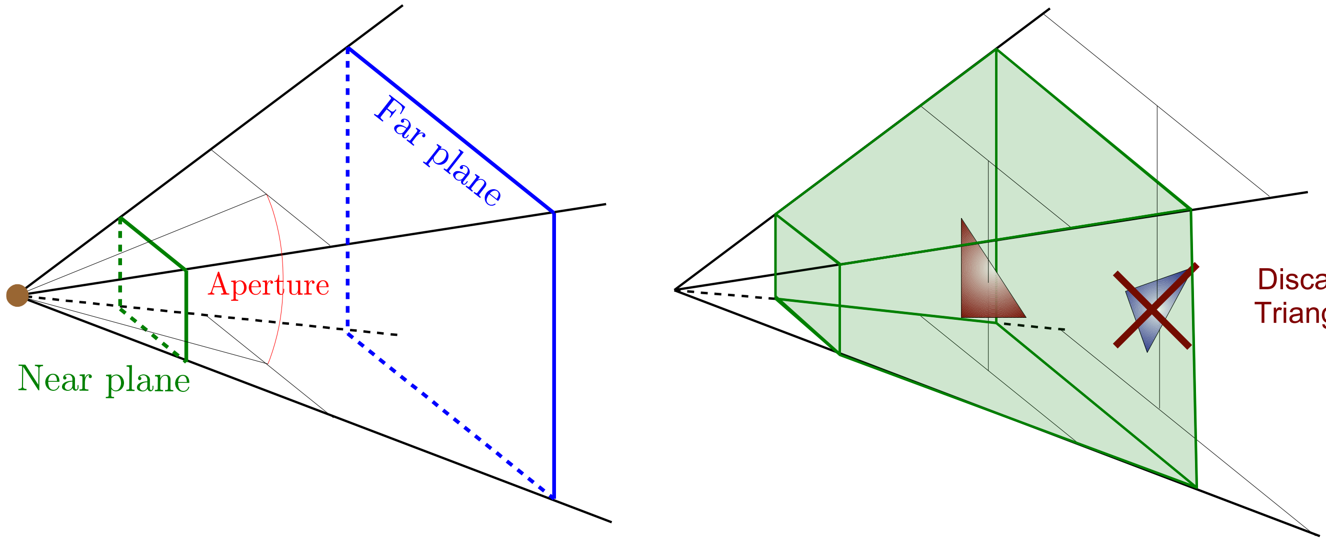
```
void display(void){
    glClearColor(0.f, 0.f, 0.f, 1.f);
    glClear(GL_COLOR_BUFFER_BIT);

    /* Switching to the projection matrix mode */
    glMatrixMode(GL_PROJECTION);
    /* Setting projection matrix to indentity */
    glLoadIdentity();
    /* Defining camera characteristics by setting proj matrix with gluPersepective */
    gluPerspective(60.f,          /* Aperture in degrees                */
                   640.f/480.f,   /* Window aspect ratio (width/height)*/
                   1.f,           /* near plane distance               */
                   100.f);        /* far plane distance                */
    /* Switching to the modelview matrix mode */
    glMatrixMode(GL_MODELVIEW);
    /* Setting modelview matrix to indentity */
    glLoadIdentity();
    /* placing the camera by setting the projection matrix with gluLookAt */
    gluLookAt(0.f, 0.f,  5.f,  /* camera origin */
              0.f, 0.f,  0.f,  /* aimed point of the camera */
              0.f, 1.f,  0.f); /* up vector (the y axis of the camera frame) */

    glColor3f(1., 0., 0.);
    glutSolidTeapot( 1. );
    glFlush();
}
```

I need to introduce new concepts about OpenGL architecture before skimming over this piece of code. OpenGL handles transformations (rotation, translations, scaling) with matrices. The camera characteristics is also handled with matrices. We are not explicitly using them, but it helps to understand the name of the function 'glMatrixMode()' and 'glLoadIdentity()'. OpenGL use two types of matrices the projection matrix which defines how the projection is done, and the modelview matrix which tells what transformations are applied to the objects (thus the 'model') and what transformations is applied to the camera (therefore the 'view').

When you call the 'glMatrixMode(GL˙PROJECTION)' primitive it tells OpenGL to change its state and every call to functions modifying OpenGL matrices will change only the projection matrix and not the modelview matrix. If you want to modify the modelview matrix you can do it similarly with 'glMatrixMode(GL˙MODELVIEW)'.

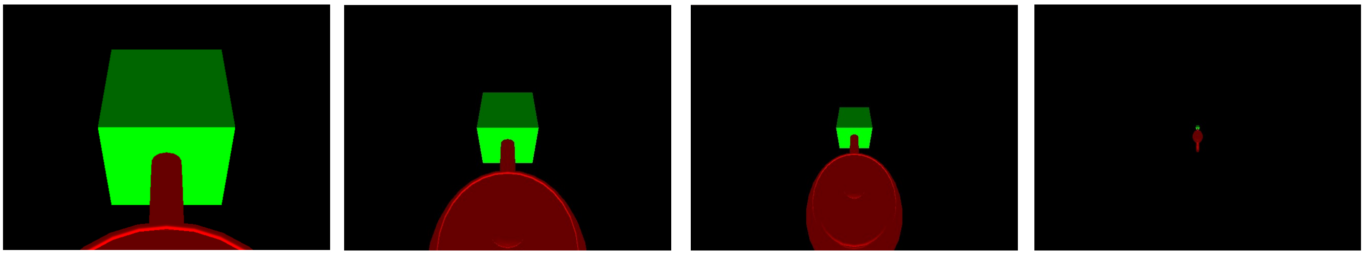
The call to 'glLoadIdentity()' set the current matrix (modelview or projection) to the identity matrix. We have to ensure the matrix is set to identity because OpenGL multiples its current matrix with matrices generated by functions like 'gluPersepective()' and then replace the current matrix by the result of this multiplication. As you may remember multiplying the identity I with a matrix M equals M ($I.M=M$). Anyway setting the current matrix to identity is a standard initialization like affecting zero to a pointer. Don't worry if these things are still unclear more explanations are coming up in the chapter about transformations.

The function ' gluPerspective()' is designed to setup the projection matrix so make sure OpenGL matrix mode is projection, and don't forget to initialize to the identity matrix. Aside from the aspect ratio parameter you can find all the other parameter in the figure below:



Computer cannot render an image at an infinite distance so we must defined the farthest renderable distance with the far plane. The near plane defines the plane were objects are projected onto, sometimes we will call it the image plane. The near plane and far plane distances are defined from the camera origin (brown point). Every objects outside the viewing volume are clipped. This volume is called the Frustum and its represented in green on the right figure. Exterior scenes in video games usually use fog, the reason is that an object will appear cut in half if it lies on the far plane.

You can define the opening angle of the camera in the 'y' direction which I call aperture. The opening angle in the x direction is deduced from the aspect ration of the window. The ratio is defined as the width over the length. The wider the aperture the more objects will be deformed by the perspective. For the same size objects farther from the camera will look smaller:

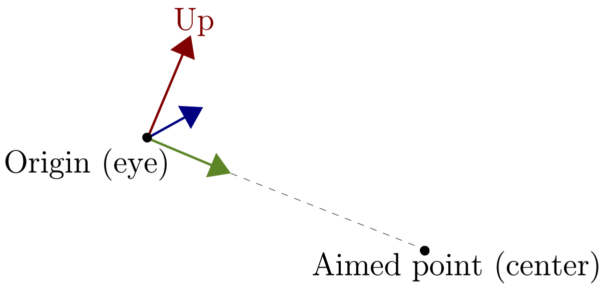


35° 75° 100° 170°

The figure above shows the same scene but with different values of apertures.

Setting the camera position requires to switch to modelview matrix mode. The call to 'gluLookAt()' will set the modelview matrix in order to translate and rotate the world so the camera will be at the right position. Again don't worry if you do not understand this matrix thing, just remember OpenGL use matrices for transformations and projection. Eventually I will explain more precisely.

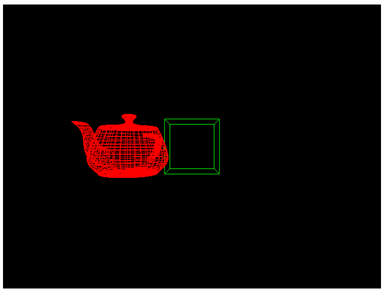
First three parameters of 'gluLookAt()' takes the origin $[x \quad y \quad z]$ of the camera the next three parameters the point aimed by the camera, and last three the up vector which is the 'y' axis of the camera's local frame:



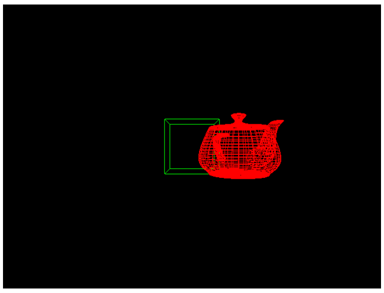
3.4 Moving the objects

It's time to animate some objects and move them around the scene. Moving the objects will be done with OpenGL's primitives such as 'glRotate()', 'glTranslate()' and glScale(). For the animation we will setup another callback function with the GLUT when the application is idle, that is to say when no events are detected. The callback will simply increment a global variable and tell GLUT to redisplay the scene (this is done with 'glutPostRedisplay()'). So what does it looks like:

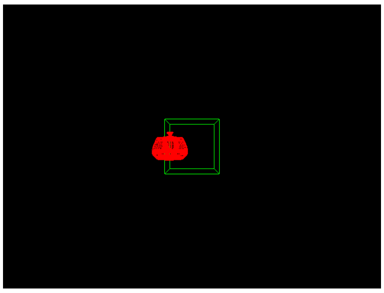
```
int angle = 0;

/* Callback when no events occurs */
void idle()
{
    /* Increment angle rotation between [0 360] */
    angle = (angle+1) % 360;
    /* Telling GLUT to call display() to rdraw the scene */
    glutPostRedisplay();
}

void display(void){
    glClearColor(0.f, 0.f, 0.f, 1.f);
    glClear(GL_COLOR_BUFFER_BIT);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.f, 640.f/480.f, 1.f, 100.f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.f, 0.f,  5.0f,
              0.f, 0.f,  0.f,
              0.f, 1.f,  0.f);

    /* Setting drawing color to green */
    glColor3f(0., 1., 0.);
    /* draw a wired cube at (0, 0, 0) */
    glutWireCube(1.f);

    /* rotate scene around the vector (0, 1, 0)(i.e. 'y' axis ) */
    /* rotation is about 'angle' degrees */
    glRotatef((float)angle, 0.f, 1.f, 0.f);
    /* translate the scene about the vector (2, 0, 0) */
    glTranslatef(2.f, 0.f, 0.f);

    glColor3f(1., 0., 0.);
    glutWireTeapot( 0.5f );
    glutSwapBuffers();
}

int main (int argc, char** argv){
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(240, 212);
    glutCreateWindow("My fancy window");

    glutDisplayFunc(display);

    glutIdleFunc(idle);

    glutMainLoop();
    return (0);
}
```



Step 1



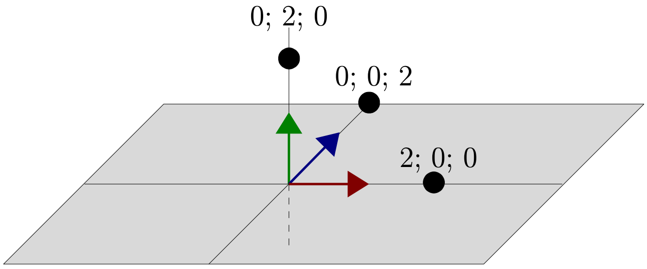
Step 2



Step 3

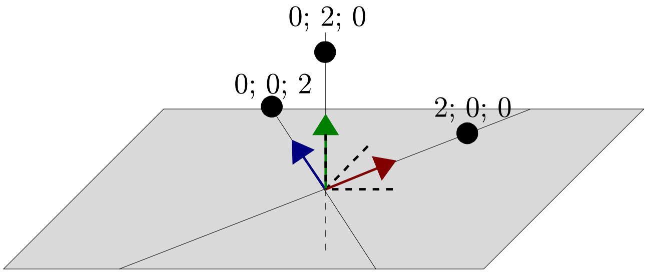
So the function 'idle()' will be called in the GLUT infinite loop when no events are detected. Since the new position of the teapot depends on how fast 'idle()' is called the animation speed is not constant. The rotation speed will depends on your computer speed. To fix that, you could use a timer and only increment the rotation angle if a specific amount of time has past.

Here is some notions to understand how OpenGL transformations works. By default OpenGL will draw points according to a centered frame. So drawing points with 'glVertex3()' will draw points knowing its coordinates corresponding to this frame:



Frame 0

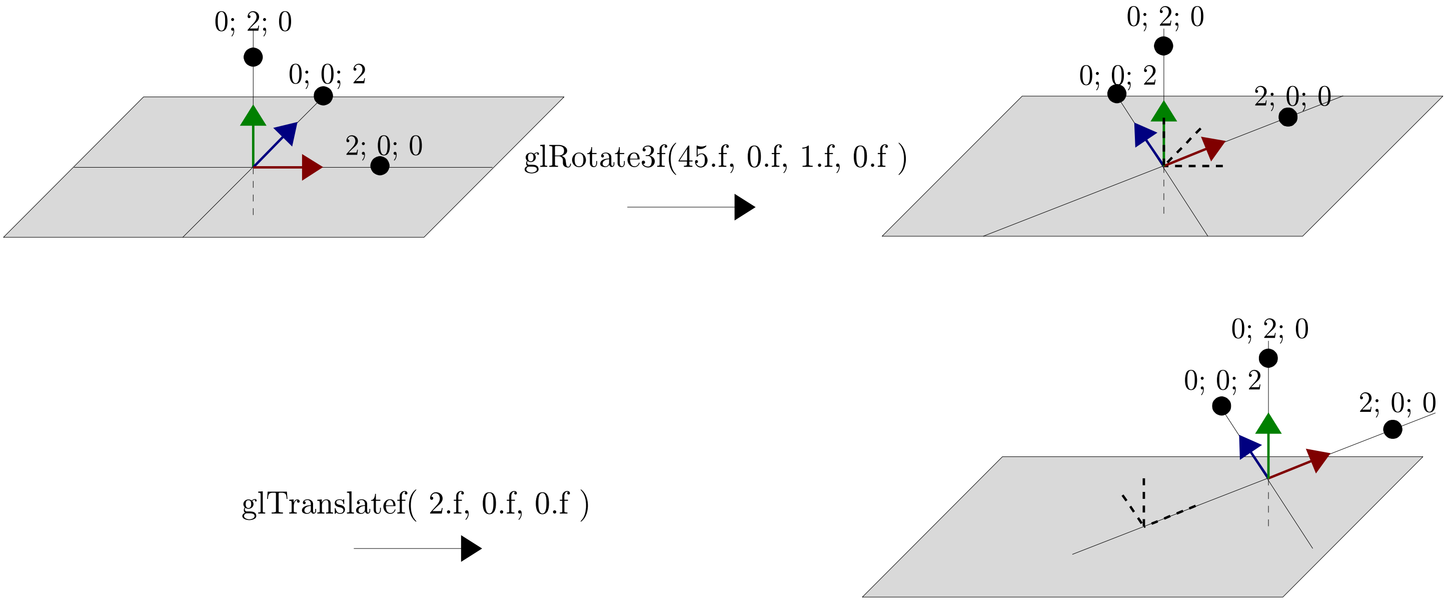
The figure show three points $p_0(2;0;0)$, $p_1(0;2;0)$, $p_2(0;0;2)$ drawn according to the centered frame 0. Now assume we do a 'glRotate(90, 0, 1, 0)', it will rotate the scene about ninety degrees around the y axis. This actually create a new frame (frame 1) which is the rotation of frame 0. Drawing the point with the same coordinates with 'glVertex3f()' will draw the points and rotate them according to frame 0 :



Frame 1
(dot lines represents frame 0)

Here is a little example of what is happening when doing a rotation and a translation in a row:

```
    glRotatef(45.f, 0.f, 1.f, 0.f);
    glTranslatef(2.f, 0.f, 0.f);
    glColor3f(0., 0., 0.);

    glBegin(GL_POINTS);
    glVertex3f(2.f, 0.f, 0.f);
    glVertex3f(0.f, 2.f, 0.f);
    glVertex3f(0.f, 0.f, 2.f);
    glEnd();
```

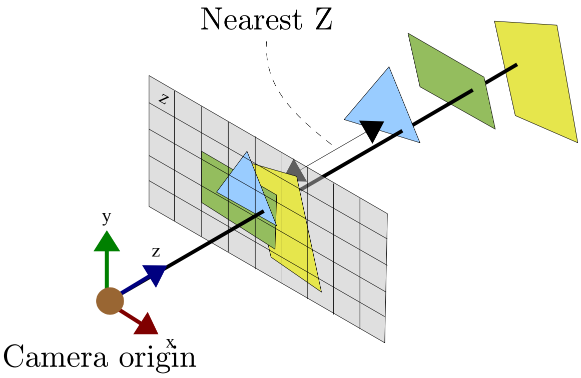


Each figure represent the frame that will be used to draw a point after a call to a transformation function.

3.5 Depth buffer

You may have noticed in the previous example that the teapot always appears in front of the cube, even though the teapot is supposed to be behind the cube. This is because we haven't activated the depth test. In OpenGL the depth test mechanism enable one to draw the objects without having to sort them by depth. With the previous example we were drawing the cube then the teapot, without the depth test OpenGL will simply draw objects after objects over the last drawn. That's why the teapot is always in front of the cube.

Since sorting the objects by depth is tedious but mostly because there are cases it is not enough, OpenGL uses the depth test (or also called the z-test). When activated OpenGL will use a buffer image called the depth buffer as large as the buffer image color. Instead of storing the color for each pixel the depth buffer will store the depth, that is to say the z coordinate of the nearest object corresponding to that pixel (in the camera local frame):

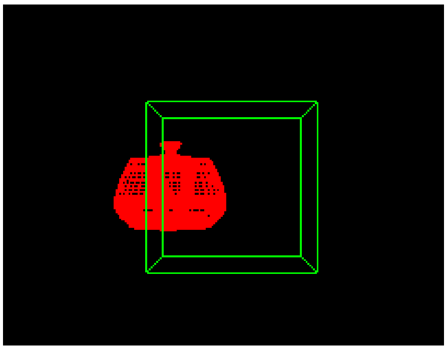


With the depth test activated each time an object is drawn OpenGL will check for every pixel if its the new pixel we want to color is the nearest. If it is then the previous depth is erased by the new depth of the object and the pixel color will be changed with the object's color.

Here is how you can enable the z-test:

```
int angle = 0;
```

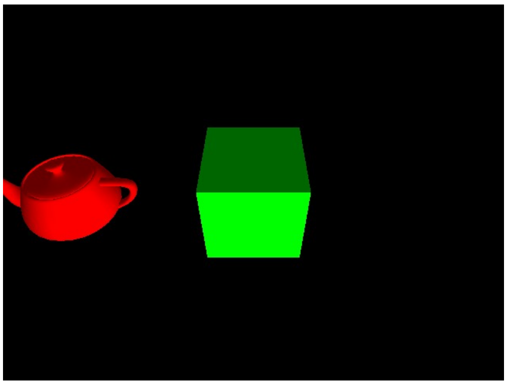
```
void idle()
{
    angle = (angle+1) % 360;
    glutPostRedisplay();
}

/* Rendering the window */
void display(void){
    /* Enabling the z-test with OpenGL */
    glEnable(GL_DEPTH_TEST);
    glClearColor(0.f, 0.f, 0.f, 1.f);
    glClearColor(0.f, 0.f, 0.f, 1.f);
    /* Don't forget to erase the depth buffer too now! */
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.f, 640.f/480.f, 1.f, 100.f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.f, 0.f,  5.0f,
              0.f, 0.f,  0.f,
              0.f, 1.f,  0.f);

    glColor3f(0., 1., 0.);
    glutWireCube(1.f);

    glRotatef((float)angle, 0.f, 1.f, 0.f);
    glTranslatef(2.f, 0.f, 0.f);

    glColor3f(1., 0., 0.);
    glutWireTeapot( 0.5f );
    glutSwapBuffers();
}

int main (int argc, char** argv){
    glutInit(&argc, argv);
    /* telling GLUT to add a depth buffer (GL_DEPTH) in addition to the color buffer */
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(240, 212);
    glutCreateWindow("My fancy window");

    glutDisplayFunc(display);
    glutIdleFunc(idle);

    glutMainLoop();
    return (0);
}
```



3.6 Adding lighting

Until now our scenes where not very realists, we were missing the lighting which will give more depth to our 3D objects by shading them. In the example below I show how to use a predefined light with OpenGL:

```
void init_light(){
    glEnable(GL_LIGHTING);
    glEnable(GL_LIGHT0);
    // Create light components
    GLfloat ambientLight[]  = { 0.2f, 0.2f, 0.2f, 1.0f }; // RGB and alpha channels
    GLfloat diffuseLight[]  = { 0.8f, 0.8f, 0.8f, 1.0f }; // RGB and alpha channels

    // Assign created components to GL_LIGHT0
    glLightfv(GL_LIGHT0, GL_AMBIENT,  ambientLight);
    glLightfv(GL_LIGHT0, GL_DIFFUSE,  diffuseLight);

    glEnable(GL_COLOR_MATERIAL);
    glColorMaterial(GL_FRONT, GL_AMBIENT_AND_DIFFUSE);
}

/* Rendering the window */
void display(void){
    glEnable(GL_DEPTH_TEST);
    glClearColor(0.f, 0.f, 0.f, 1.f);
    glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    gluPerspective(60.f, 640.f/480.f, 1.f, 100.f);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    gluLookAt(0.f, 2.5f,  2.5f,
              0.f, 0.f,  0.f,
              0.f, 1.f,  -1.f);

    /* Setup the light position which is (0, 0, 3) */
    /* fourth value '1' means its a point light    */
    /* '0' would have meant directionnal light     */
    GLfloat position[] = { 0.f, 0.f, 3.f, 1.0f };
    glLightfv(GL_LIGHT0, GL_POSITION, position);

    glColor3f(0., 1., 0.);
    glutSolidCube(1.f);

    glRotatef((float)angle, 0.f, 1.f, 0.f);
    glTranslatef(2.f, 0.f, 0.f);

    glColor3f(1., 0., 0.);
    glutSolidTeapot( 0.5f );
    glutSwapBuffers();
}

int main (int argc, char** argv){
    glutInit(&argc, argv);
    glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
    glutInitWindowSize(640, 480);
    glutInitWindowPosition(240, 212);
    glutCreateWindow("My fancy window");

    glutDisplayFunc(display);

    glutIdleFunc(idle);

    init_light();

    glutMainLoop();
```



```
    return (0);
}
```

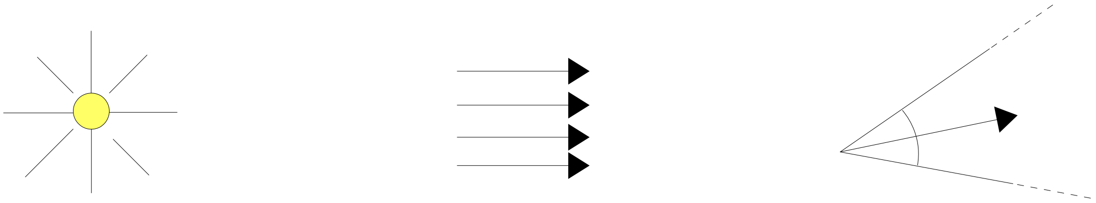
So what's in 'init_light()'? In this function first we activate OpenGL lighting with 'glEnable(GL_LIGHTING)' and then activate the light number zero with another 'glEnable(GL_LIGHT0)'. OpenGL can handle up to height lights usually, you can check this with function like 'glGetIntegerv(GL˙MAX˙LIGHTS, &nb˙light)'. Note also that you can disable lighting or a specific light at any moment with a 'glDisable()' and the corresponding macro you used with 'glEnable()'.

For now I'll skip detailled explanation on the 'glLightfv()' calls. There are setting up the light color. What might seems weird is the two calls to 'glLightfv()' with GL_AMBIANT and GL_DIFFUSE to setup one color. It's because the color is seperate in several components. Here I only care about two but there is actually three components you can set up: ambiant, diffuse and specular.

Because I'm using this decomposotion of colors with lights I need to use the same mechanism for objects. This is done with 'glEnable(GL_COLOR_MATERIAL)'. At last I need to specify which components of colors I'm using for the objects with 'glColorMaterial()' the GL_FRONT tells I'm only coloring triangles that are facing us. GL_AMBIENT_AND_DIFFUSE indicates that I'm using ambiant and diffuse components and specular is desactivated.

A last call is made to 'glLightfv()' in the display loop to define the light position which is centered about the point (0, 0, 3). But there is a fourth component, when set to one it means the light is a point light. This fourth component will be explained in the transformation chapter about homogenous coordinates.

In OpenGL there is three types of lights, point lights, directionnal lights and spot lights:



Point light

Directionnal light

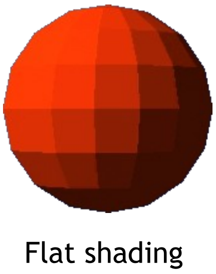
Spot light

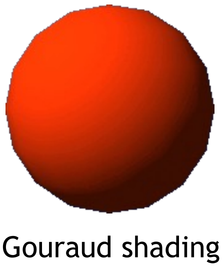
- ✔ Point lights are *omni*-directional
- ✔ Directionnal lights are placed at the infinity were just the direction is specified
- ✔ Spot lights only lit areas inside there ligth cone

Understand lighting thoroughly is behond the chapter, it will make sense after the chapter about lightings models.

3.6.1 Gouraud shading vs Flat shading

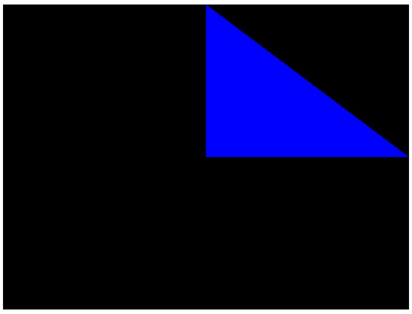
You can try to add to the previous example in the light initialization function a call to 'glShadeModel(GL_SMOOTH)' or 'glShadeModel(GL_FLAT)'. It is two different shading mode. GL_SMOOTH is commonly called the gouraud shading, GL_FLAT simply stands for flat shading. This is what you will saw:

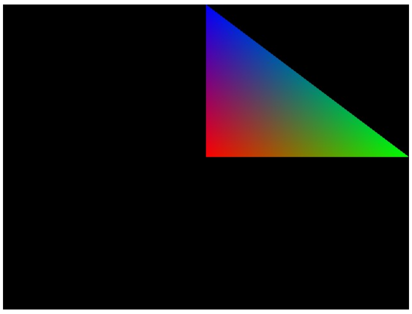




With flat shading every faces as the same color whereas with Gouraud shading color smoothly blends between faces. More precisely in flat shading OpenGL takes the color of a single vertex belonging to a face and color the face only with that color. With Gouraud shading OpenGL will compute at each vertex of a face a color and smoothly interpolate these colors as we have seen with the triangle example (c.f. section Direct mode drawing).

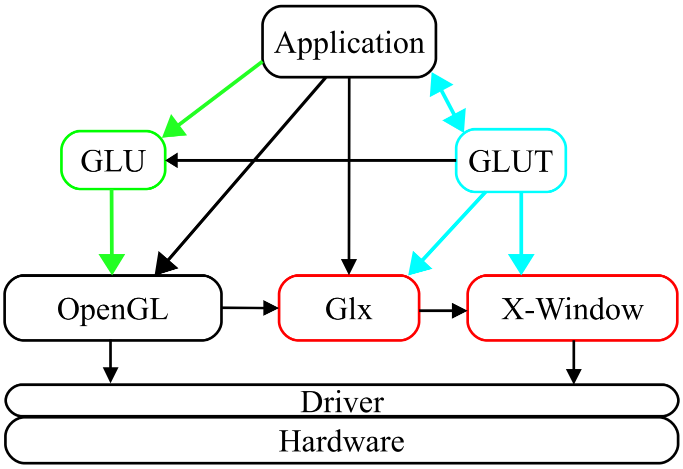
You can try flat and gouraud shading on the triangle example and you will see this:


Flat shading


Gouraud shading

3.7 OpenGL architecture and interactions

You have seen in the previous sections OpenGL, GLUT and GLU functions. The scheme below shows how your application and these libraries interact between each other:



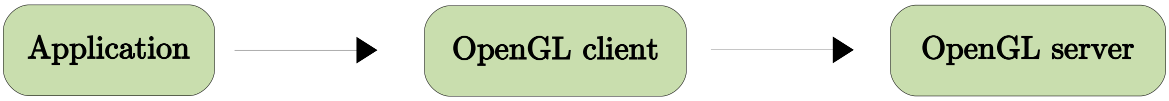
Here is a summary of the above figure:

- Application: obviously your revolutionary OpenGL based program
- OpenGL: a magnificent library used to display polygons and do awesome real-time rendering
- GLU: a higher level library on top of OpenGL which provide some useful functions which hides some complex OpenGL procedures.

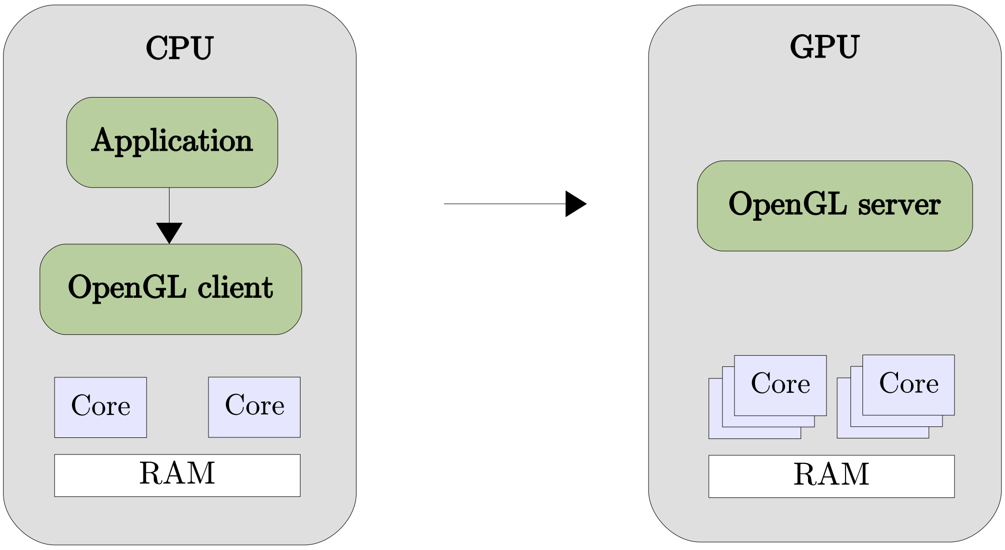
✔ Glx: Linux specific library used by OpenGL to communicate with the window manager
✔ X-Window: Linux window manager
✔ GLUT: A higher level library on top of Glx and X-window (or the Win32 API for windows) extremely simplified for ease of use to handle a single window and events (keyboard, mouse etc.)

3.7.1 OpenGL client-server model

OpenGL model is a client server model. When calling 'gl' functions your application is giving order to the client side of OpenGL and then the client transmit the order to the server:

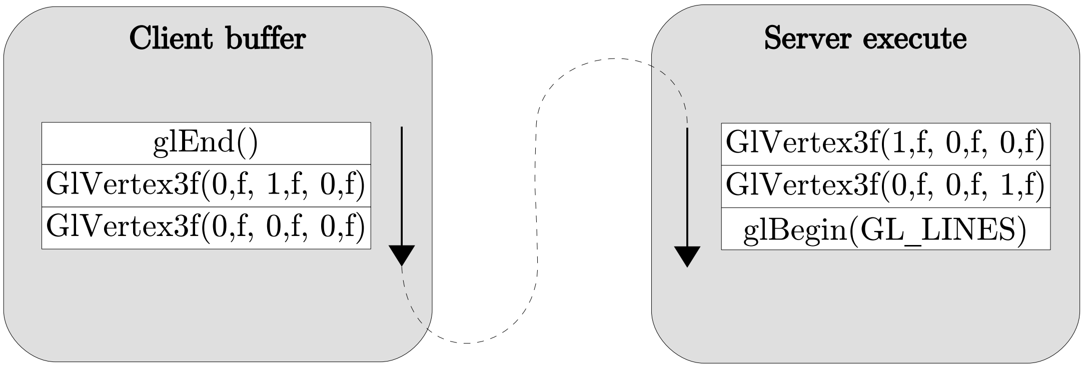


Most of the time client and server are running on the same machine. This architectrure enables OpenGL to seperate the load between CPU and GPU. The client being executed on CPU and server on GPU:



Thanks to this architecture OpenGL command can be run asynchronously. When calling 'gl' commands the client return the control to the application without waiting for the server to have completed its task. This avoid the client side to be idle for nothing. More operation can be computed on the CPU while the GPU is working on the last commands.

Note that the client can also bufferize the commands and send them later:


Client buffer
glEnd()
GlVertex3f(0,f, 1,f, 0,f)
GlVertex3f(0,f, 0,f, 0,f)
Server execute
GlVertex3f(1,f, 0,f, 0,f)
GlVertex3f(0,f, 0,f, 1,f)
glBegin(GL_LINES)

This explains the 'glFlush()' command we have seen earlier. It tells the OpenGL client to flush its command buffer. 'glFlush()' only returns when the buffer is empty. This does not mean the commands has all been executed on server side. Their exists another command 'glFinish()' which effectively forces OpenGL to wait until the commands has been processed on the server side. For performance consideration we usually do not use this function.

To provide a consistent behavior OpenGL garantees the following properties:

- ✔ OpenGL commands are executed in order as there are received from the application.
- ✔ OpenGL copies client data at call-time. When an application calls an OpenGL function, the OpenGL client copies any data provided in the parameters before returning control to the application. For example, if a parameter points at an array stored in application memory, OpenGL must copy that data before returning. Therefore, an application is free to change its owns memory regardless of calls it makes to OpenGL.

3.7.2 The state machine

OpenGl is a state machine, this means when changing OpenGL state it will persist until you change that state to another value. State machine is a nice expression to say I am using global variables without saying it. Each time you call a 'glEnable(GL_SOMETHING)' or 'glDisable(GL_SOMETHING)' it basicaly change a global variable of boolean type and it won't change until you call again either a 'glEnable(GL_SOMETHING)' or 'glDisable(GL_SOMETHING)'.

As I showed you there is plenty of OpenGL states that you can disable or enable with the 'gl–Dis—En″able()' but function like 'glMatrixMode()' are also altering the OpenGL state by changing which matrix should be used. Likewise 'glRotate()' & Co modifies the current matrix which is part of OpenGL sate. The current matrix is used to transformed objects, if you changed the matrix with a 'glTranslate()' the next object you'll draw will be at a different location. Obviously OpenGL matrix state has changed.

Handling OpenGL is an important aspect of OpenGL programming. One can access states with the function:

glGet–Type″v(GLenum state˙name, Type* array)

Where 'Type' can be replaced by –Integer—Float—Boolean—Double. Here is some states you can get:

```
    GLboolean state;
    glGetBooleanv(GL_DEPTH_TEST, &state); // state is true if depth test is activated
    glGetBooleanv(GL_LIGHT0, &state);     // state is true if the light 0 is activated

    GLint mode;
    glGetIntegerv(GL_MATRIX_MODE, &mode);
    if(mode == GL_PROJECTION_MATRIX){
        // Do something
    }else if(mode == GL_MODELVIEW_MATRIX){
        // Do something
    }
```

There is a long list available, keep in mind that for each state there is a way to get its value, when the time come you can even retreive OpenGL matrices values with a 'glGetFloatv()'.

3.7.3 The extension system

Wikipedia say it very well so here it is:
“

The OpenGL standard allows individual vendors to provide additional functionality through extensions as new technology is created. Extensions may introduce new functions and new constants, and may relax or remove restrictions on existing OpenGL functions. Each vendor has an alphabetic abbreviation that is used in naming their new functions and constants. For example, Nvidia's abbreviation (NV) is used in defining their proprietary function glCombinerParameterfvNV() and their constant GL_NORMAL_MAP_NV.

It may happen that more than one vendor agrees to implement the same extended functionality. In that case, the abbreviation EXT is used. It may further happen that the Architecture Review Board ”blesses” the extension. It then becomes known as a standard extension, and the abbreviation ARB is used. The first ARB extension was GL˙ARB˙multitexture, introduced in version 1.2.1. Following the official extension promotion path, multitexturing is no longer an optionally implemented ARB extension, but has been a part of the OpenGL core API since version 1.3.

Before using an extension a program must first determine its availability, and then obtain pointers to any new functions the extension defines. The mechanism for doing this is platform-specific and libraries such as GLEW and GLEE exist to simplify the process.

“

3.8 Conclusion

In a nutshell what have we learned?

✔ Basic usage of GLUT (creating a window, setting up callbacks, using predefined objects setup double buffering)
✔ Direct mode drawing with OpenGL
✔ Setup the camera
✔ Handling transformations
✔ Handling lights

But these are only tools, and we need to see the bigger picture. In order to generate synthetic images what kind of information do we need, and what type of computation are involved?

First we handle **geometry**. Objects surface can be represented with polygons which is what OpenGL knows to render directly, yet we could use equations, points, voxels etc. This geometry has to be placed by using affine transformations. Geometry **materials** has to be described, we have seen that we can associate at each vertex a color with OpenGL, and later we will use textures to further describe the geometry materials. Finally **lighting** is to be specified by placing the lights and camera positions and defining their characteristics (Perspective/Orthogonal projection, point/directional light etc.)

Once all the data has been set the image has to be computed, lighting must be computed over the mesh, this is done with a specific **lighting model** (OpenGL uses the blinn-phong model for instance). We will have shaded objects, shadows etc.