Project 1 – Open GL Cube Using QT

I – Introduction:

The main goal of this project is to build an graphical application using the capabilities provided by QT Creator to emulate a 3D cube in OpenGL. For this purpose I have use a Linux Ubuntu OS that provides a simplified environment to develop applications in QT. Also important to notice is the fact that QT has a great support for OpenGl applications that can be recompiled natively in any OS.

II – Design of the Main Window:

The components for developing in QT are known as widgets. These components are native objects in QT that have different properties and functions that can be widely adapted for the purpose of the application being developed. The main components of this applications are:

a) myqglwidget.h and myqglwidget.cpp

b) qgl_mainwidget.h and qgl_mainwidget.cpp

c) main.cpp

For this project we have created a GL cube and also an animation to decide which cubes are going to be “on” at any given time. Another feature of this application is the ability to open an .xyz file containing the coordinates of an object that can be drawn on the GLWidget. In order to control the view of the GLWidget I needed to create a main layout of sliders, buttons, spinboxes and labels. The parameters we can control to modify the 3d view of the GL object are:

a) x, y and z rotation

b) Zoom (in and out)

c) Led Size

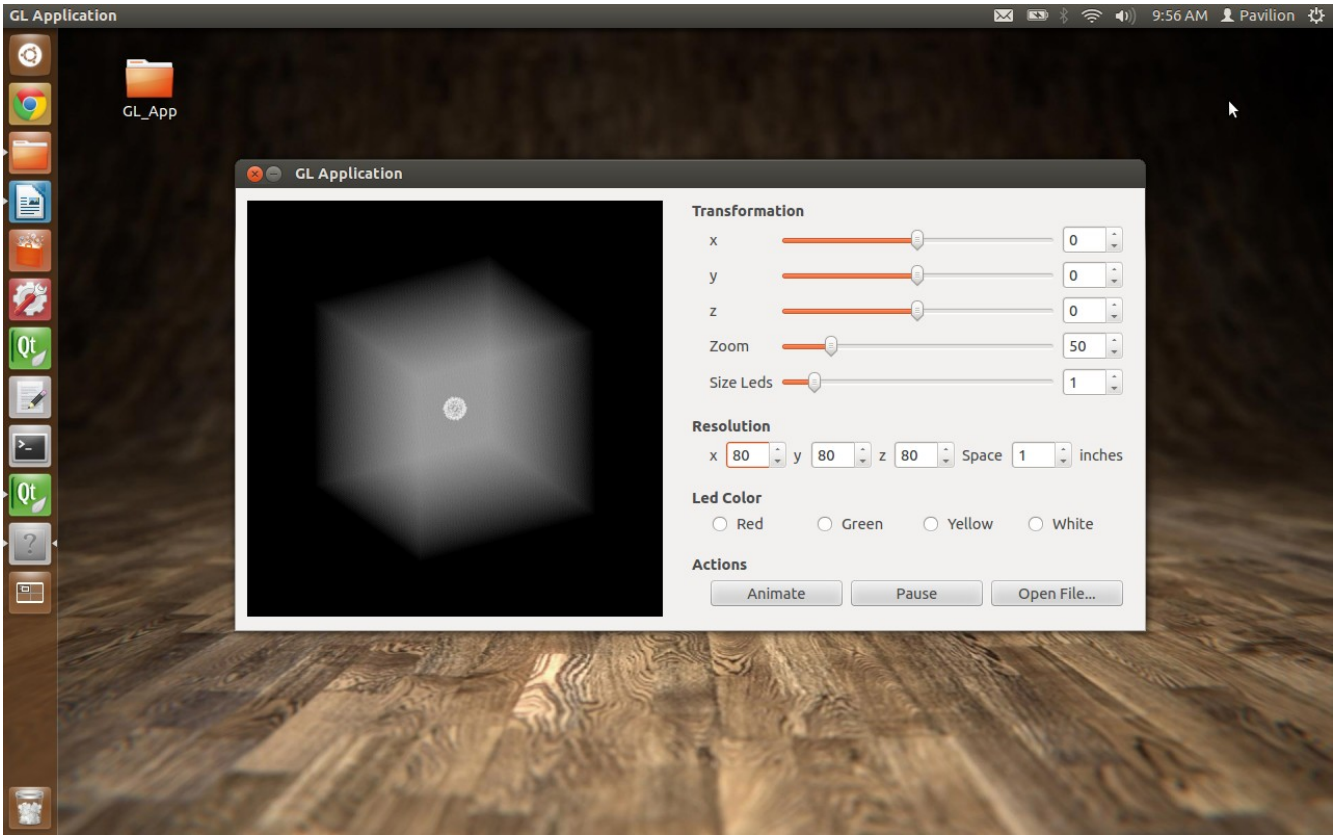
d) Resolution, amount of leds on the x, y and z direction

e) Spacing

f) Led Color (4 different options)

g) Actions: Animate, Pause and Open a File.

An overview of the application is as follow:



In the header of the class qgl_mainwidget we have declared the main frame that was used for this application, called QTFrame, which is widget that will later contain all the different sub-widgets such as buttons, spinboxes, group boxes in their specific layouts, and the GLWidget that was used to generate the Led Cube. We have instantiated QFrame as the main widget.

Details on the qgl_mainwidget header file:

```
#ifndef QGL_MAINWIDGET_H
#define QGL_MAINWIDGET_H
#include <QFrame>

////////////////widgets that will be used in the code:///////
class QLabel;
class QCheckbox;
class QGroupBox;
class QSpinBox;
class QSlider;
class QRadioButton;
class QLineEdit;
class QFrame;
class QGLWidget; //this is the widget that provides OpenGL natively on QT
class MyGLWidget;
class QPushButton;

class QGL_MainWidget : public QFrame
{
    Q_OBJECT

public:
    QGL_MainWidget(QWidget *parent = 0);

private:
    MyGLWidget *GlFrame;
    QGroupBox *Resolution;
    QLabel *x_label_resolution;
    QSpinBox *x_spin_resolution;
    QLabel *y_label_resolution;
    QSpinBox *y_spin_resolution; //etc, remainder variables were declared after.
```

After defining what would be the main frame of the application and the widgets to be used, we have then generated the implementation file qgl_mainwidget.cpp, here we instantiated every widget and give the specific parameters to each one. Then in the main.cpp file we just do a simple call to the QFrame widget:

```
#include <Qapplication>
#include "myqlwidget.h"
#include "qgl_mainwidget.h"

int main(int argc, char *argv[])
{
QApplication app(argc, argv);
QGL_MainWidget mainWin;
mainWin.setFixedSize(875, 425);
mainWin.show();
return app.exec();
}
```

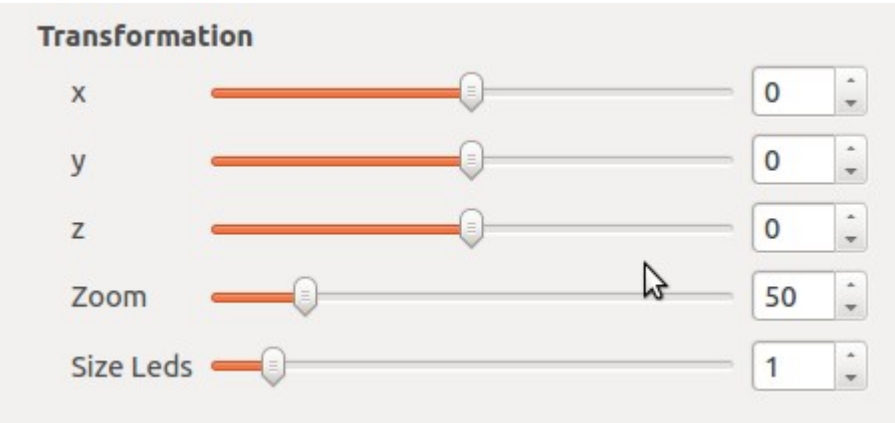
One of the problems I have found was related to the fact that when maximizing the window the general layout of the application gets modified and is not visualy appealing anymore so I was able to fix this problem by setting the window to be of a fixed size, providing an overall consistent experience on any screen configuration.

Details on the qgl_mainwidget implementation file:

On this file we have set all the different parameters for each of the widgets utilized on this project, following this convention:

small widgets ↔ connected with other small widgets → local layout → groupbox → Main_layoutright(left one is used for QGL Widget) → main layout → QFrame.

For Instance, for a spinbox widget, we will first connect it with another small widget that is related to it using signal and slots, then we will put these widgets on a local layout, after these steps, we will include the local layout in a groupbox, and finally groupboxes will be included in a right main layout, then included in a general main layout. Here is a typical GUI look and code for a group box with widgets:




```
Transforms = new QGroupBox("Transformation", this);

x_label_rotation = new QLabel("x");
x_slider_rotation = new QSlider(Qt::Horizontal);
x_spin_rotation = new QspinBox();
```

```
x_spin_rotation->setRange(-180, 180);
x_slider_rotation->setRange(-180, 180);
x_spin_rotation->setValue(0);
```

Here we can see that we have created the groupbox to contain the local layout, then after this we have instantiated a Qlabel with a text “x”, then a QSlider and QSpinBox. These last two elements has some specific properties that can be set up such range and initial values, so we have specified these parameters.

After initializing these widgets is logic to connect the spinboxes with the sliders, we do this by using QT signals and slots. It was a fairly new concept for me, but after working with it for a little while it became very interesting and convenient. Here is a small snippet showing the connection between a slider and a spinbox:

```
QObject::connect(z_spin_rotation, SIGNAL(valueChanged(int)), z_slider_rotation,
SLOT(setValue(int)));
QObject::connect(z_slider_rotation, SIGNAL(valueChanged(int)), z_spin_rotation,
SLOT(setValue(int)));
```

Here we can see how one widget emits a specific signal (i.e. value changed) that is picked up by another widget to change its own parameters. It was confusing to notice at the beginning that how this doesn't end in a infinite recursion of signal-response when connecting the second widget back to the first one, but this problem was solved on QT meta compiler (internal code).

After this we proceed to add all the widgets to one of three different layouts: horizontal, vertical or grid. In the case of the horizontal or vertical layouts, the elements are placed in order (so order here is important), but in the case of grid layout, we define the specifics coordinates of each widget as parts of a whole.

Snippet example of adding widgets to a grid local layout: We first create a local grid layout for the widgets, and then add the widgets to it:

```
QGridLayout *Transforms_layout = new QGridLayout();
Transforms_layout->addWidget(x_label_rotation, 1, 1);
```

```
Transforms_layout->addWidget(x_slider_rotation, 1, 2);
Transforms_layout->addWidget(x_spin_rotation, 1, 3);
```

Then after we finish adding the widgets to the layout, we add the layout to the groupbox:

```
Transforms->setLayout(Transforms_layout);
```

The next step consists of adding these groupboxes to the right main layout, which in turn will be added to the main layout together with the GLWidget

```
Main_layoutRight->addWidget(Transforms);
Main_layout->addWidget(GlFrame);
Main_layout->addStretch();
Main_layout->addLayout(Main_layoutRight);
```

The reason why we need to add this right layout refers to the necessity to combine the GLWidget with all the widgets (previously described) in the same mainLayout.

III – Design of the GLWidget:

The GLWidget is a special widget native on QT that has the capabilities to integrate OpenGL code into it. Since it consists of a big part of this application we have dedicated a special section to it.

Details on the myqlwidget header file:

In this file we have declare all the public and private variables and functions that are going to modify the display of the GL object on the QGLWidget.

GLWidget Variables:

```
int x_resolution;
int y_resolution;
int z_resolution;       //these three variables represent the x,y,z amount of
                        led lights for the Cube.

double spacingvalue;    //space between leds.

GLdouble redcolor;
GLdouble greencolor;
GLdouble bluecolor;     //these are the different components of the color
                        blends. We mix them to create yellow, and primary
                        colors.
```

```
double ledSize;         //this is the size of each led.

int ix,iy,iz;           //counters for coordinates.
int xCoordinate, yCoordinate, zCoordinate.
int radiousball, animateZrot; //variables for animation.

std::string filename; //filename for opening a file.

bool direction; //animation expanding or contracting
QTimer *timer; //timer for the animation.
```

The following functions are the most important for the modification of the Led Cube.

GLWidget Functions:

```
void initializeGL();
void paintGL();
void drawCube();
void drawFromFile();
void resizeGL(int width, int height);
QSize minimumSizeHint() const;
QSize sizeHint() const;
void mousePressEvent(QMouseEvent *event);
void mouseMoveEvent(QMouseEvent *event);
void setXRotation(int angle);
void setYRotation(int angle);
void setZRotation(int angle);
void setZoom(int zoom);
void setResolution_x(int x);
void setResolution_y(int y);
void setResolution_z(int z);
void setResolution_spacing(int spacing);
void changeColorRed(bool colorchoice);
void changeColorGreen(bool colorchoice);
void changeColorYellow(bool colorchoice);
void changeColorWhite(bool colorchoice);
void changeSize(int size);
void animate(bool yesno);
void changeAmplitude();
void openFile(bool);
void stopAnimation(bool);
void drawPoint(int x, int y, int z);
```

Details on the myqlwidget implementation file:

Here we have defined some starting values for the GLWidget variables, and implement each function:

Variables:

The values of these variables have been determined to be ideal for the default of the application.

```
    xRot = 64*16;
    yRot = 6;
    zRot = 31*16;       //these values provide the ideal cube rotation to start the
                        viewport
    x_resolution = 80;
    y_resolution = 80;
    z_resolution = 80;  //these are the starting resolution values

    spacingvalue = 1;   //spacing value by default

    redcolor = 1.0;
    greencolor = 1.0;
    bluecolor = 1.0;    //default color values

    ix=0;
    iy=0;
    iz=0;               //default coordinates to draw the cube

    radiousball = 4;    //radious of the sphere for animation
    filename=" ";
    direction = true; //this variable sets the direction of the animation.
    timer = new Qtimer(this); //timer for the animation
```

Functions:

```
void MyGLWidget::initializeGL()

{
    qglClearColor(Qt::black);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
    glShadeModel(GL_SMOOTH);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
}
```

The main purpose of this function is to initialize the GL object. I have learned that GL is a state machine so the changes made at the beginning will be present on the object unless it is changed later on. Here we define the background color to be black, we enable ShadeModel to be *smooth* and also enable blending so we can assign to each led a blend of colors (that's how we got yellow, for example).

```
void MyGLWidget::paintGL()

{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -10.0);
    glRotatef(xRot / 16.0, 1.0, 0.0, 0.0);
    glRotatef(yRot / 16.0, 0.0, 1.0, 0.0);
    glRotatef(zRot / 16.0, 0.0, 0.0, 1.0);
    if (filename == " ") drawCube();
    if (filename != " ") drawFromFile();
}
```

This function takes care of painting the GL object and is called by default. It first flush the GL object assigning the background color. Then it loads the identity matrix and perform a translation to adjust the viewport 10 units back in the z axis direction. Then it performs a rotation to show that the object is in 3d. If we leave the object without rotation it will show it as a square without a sense of depth. The one of the most important calls is made: draw the gl object that we define. If the filename is empty (as by default) then draw a sphere, otherwise draw the data from the file selected.

```
void MyGLWidget::drawCube()
{
    for (ix = -x_resolution/2; ix <= (x_resolution-1)/2 ; ix++){
          for ( iy = -y_resolution/2; iy <= (y_resolution-1)/2 ; iy++){
                for ( iz = -z_resolution/2; iz <= (z_resolution-1)/2 ; iz++){
                    if (radiousball  == round(sqrt (((ix*ix) + (iy*iy) +
                                    (iz*iz))))) //function to draw a sphere...
                    {
                        drawPoint(ix*spacingvalue, iy*spacingvalue,
                                    iz*spacingvalue);
                    }
                    else //this will draw off points
                    {
                        glPointSize(0.01f);
                        glBegin(GL_POINTS);
                        glColor4f(1, 1, 1, 0.03 );
                        glVertex3f(ix, iy, iz);
                        glEnd();
                    }
    }
        }
            }
}
```

This function is responsible of drawing the cube (with the leds off) and to turn on the leds that are reported to be part of a sphere function. As a result we can see inside the cube a sphere. Since the radius of the sphere can be modify as a function of time, we can expand and contract the sphere using a timer as an animation. When drawing points on the GL state machine, We begin by declaring the size of it, and the type (here points) and then the blend of colors, the first 3 refers to red, green and blue, and the last number (a float) represents the alpha channel or transparency. Then we finish by calling the method glEnd(). We do this inside 3 loops providing the positions for x, y and z positions.

```
void MyGLWidget::drawFromFile()
{
        string STRING;
        ifstream infile;
        infile.open (filename.c_str());
        getline(infile,STRING);
        int x,y,z;
        while(infile.peek() != -1){ // To get all the lines.
            int spacekey1 = 0;
            int spacekey2 = 0;
            string xc = "";
            string yc = "";
            string zc = "";
                getline(infile,STRING);
                for (int i = 0; i < STRING.length(); i++){
                    if (STRING[i] != ' '){
                    if (spacekey1 == 0){
                        xc = xc + STRING[i];
                    }
                    if (spacekey1 > spacekey2){
                        yc = yc + STRING[i];
                    }
                    if (spacekey1 < spacekey2){
                        zc = zc + STRING[i];
                    }
                    }
                    else {
                        if (spacekey1 == 0) spacekey1 = i;
                        else spacekey2 = i;}
                    }
                x = round(atoi(xc.c_str()));
                y = round(atoi(yc.c_str()));
                z = round(atoi(zc.c_str()));
                drawPoint(x*spacingvalue, y*spacingvalue, z*spacingvalue);
                if (timer->isActive()) setZRotation(animateZrot);
            }
        infile.close();
}
```

The function drawFromFile() can read any .xyz file and draw the coordinates given. It works by opening a .xyz file with a file reader and outputting every line into a variable that can be then separated into the corresponding x y and z coordinates. After getting each value we then convert each string into an integer and draw it. It uses the function drawPoint() that basically take the three coordinates given and draw the point in the space.

```
void MyGLWidget::resizeGL(int width, int height)
{
    int side = qMin(width, height);
    glViewport((width - side) / 2, (height - side) / 2, side, side);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glScalef(0.5, 0.5, 0.5);
    glOrtho(  (-(x_resolution+spacingvalue)/2), ( (x_resolution+spacingvalue)/2),
(-(y_resolution+spacingvalue)/2), ((y_resolution+spacingvalue)/2), 0.0,
15.0*z_resolution*2);
    glMatrixMode(GL_MODELVIEW);
}
```

The main purpose of this function is to modify the viewport of the GL object by modifying the view matrix. This is done by first calling GL_PROJECTION matrix mode, and loading the identity matrix. Then we scale it and later call glOrtho to display a paralell projection. After that we change GL to modelview again in order to modify the model matrix when we need to. Resize GL is receives two integers width and height that provide the parameters for the new viewing values. Basically this function resize the GL Object to fit into the GL Widget.

```
void MyGLWidget::mousePressEvent(QMouseEvent *event)
{
    lastPos = event->pos();
}
void MyGLWidget::mouseMoveEvent(QMouseEvent *event)
{
    int dx = event->x() - lastPos.x();
    int dy = event->y() - lastPos.y();
    if (event->buttons() & Qt::LeftButton) {
        setXRotation((xRot + 8 * dy)/16);
```

```
        setYRotation((yRot + 8 * dx)/16);
    } else if (event->buttons() & Qt::RightButton) {
        setZRotation((zRot + 8 * dx)/16);
    }
    lastPos = event->pos();
}
```

The purpose of these two functions is to receive the events from the mouse such press and move. When receiving the mouse press signal, it records the previous location and then rotate the cube acordingly.

```
void MyGLWidget::setYRotation(int angle)
{
    angle = angle * 16;
    if (angle != yRot) {
        yRot = angle;
        emit yRotationChanged(angle);
        updateGL();
    }
}
```

This function receives the angle as a parameter and normalize it by multiplying it by 16. I have found this to be necessary since the integers passed by the mouse event are different in range that the one passed by the spinbox on the QTWidgets. If I don't do this, then when changing the angle on the spinbox it would perform several complete rotations (which was not intended). After this, it passes that integer to paintGL() by updating the view matrix with the call updateGL();

```
void MyGLWidget::setZoom(int zoom)
{
    float myzoom = zoom / 100.00;
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glScalef(myzoom, myzoom, myzoom);
    glOrtho(  (-(x_resolution+spacingvalue)/2), ( (x_resolution+spacingvalue)/2),
(-(y_resolution+spacingvalue)/2), ((y_resolution+spacingvalue)/2), -10.0,
z_resolution/2);
    glMatrixMode(GL_MODELVIEW);
    emit zoomChanged(zoom);
    updateGL();
}
```

The setZoom function is responsible to change the view matrix on GL and scale it to any factor given. We divide the parameter given by 100 in order scale it by the corresponding factors. Do 1.1 scale means that it will be 10% bigger. We keep the current GL matrix to be in modelview mode in order to directly modify this matrix, so after scaling (done in the view matrix) we go back to the model matrix.

```
void MyGLWidget::setResolution_y(int y)
{
    y_resolution = y;
    updateGL();
    emit resolutionChanged_y(y);
}
```

This function will receive the integer in order to determine how many leds will be drawn on each direction. Then it will update the GL Object.

All these functions emit signals that are picked up in turn on the QT widgets using this snippet:

```
QObject::connect(zoom_spin_rotation, SIGNAL(valueChanged(int)), GlFrame,
SLOT(setZoom(int)));
QObject::connect(GlFrame, SIGNAL(zoomChanged(int)), zoom_slider_rotation,
SLOT(setValue(int)));
```

The Change color function also work in a similar way.

```
void MyGLWidget::animate(bool yesno)
{
    connect(timer, SIGNAL(timeout()), this, SLOT(changeAmplitude()));
    timer->start(10);
}
void MyGLWidget::stopAnimation(bool)
{
    timer->stop(); //stop the timer...
}
void MyGLWidget::changeAmplitude()
{
    if (direction) radiousball++;
    else radiousball--;
    if (radiousball == 50) direction = !direction;
    if (radiousball == 0) direction = !direction;
```

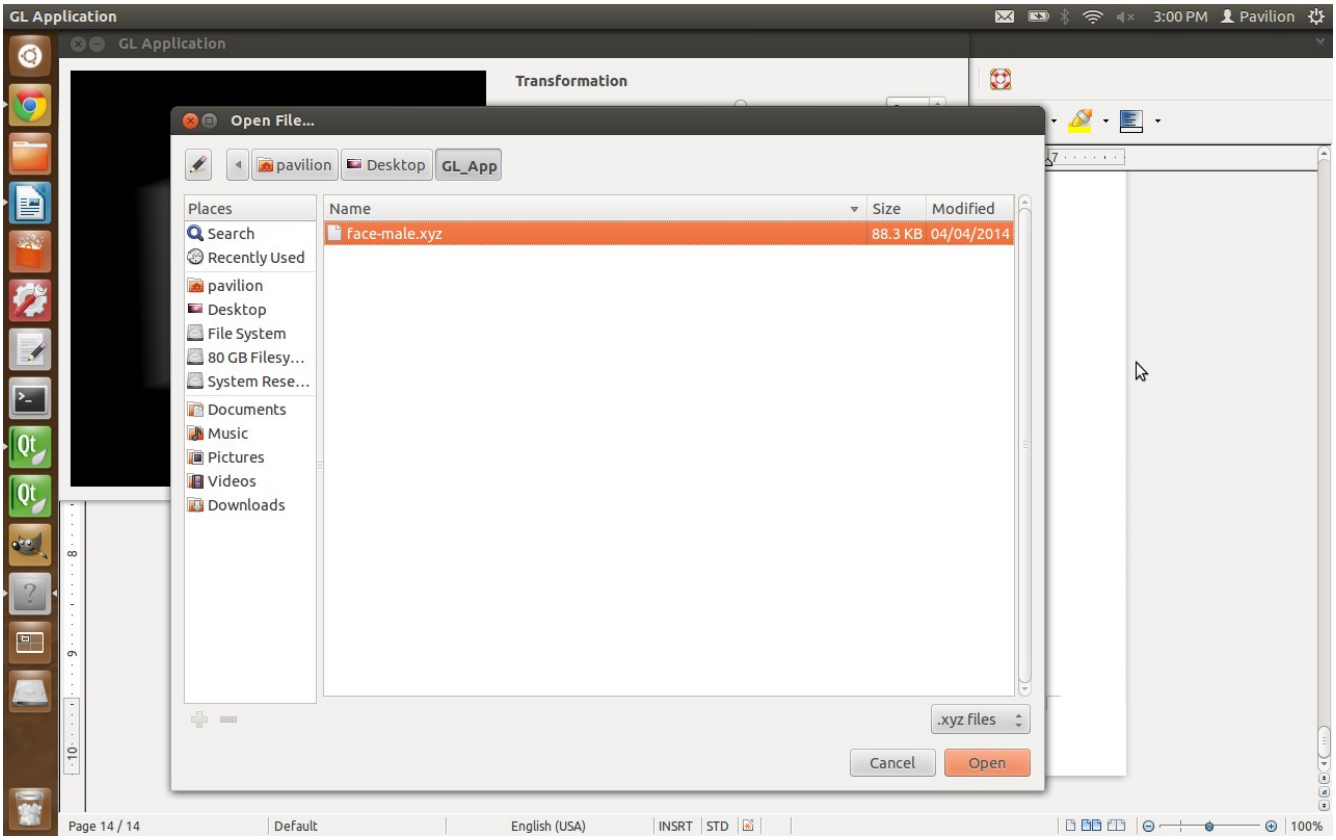
```
    if (filename != " "){
         animateZrot+=5; ///animation for the face...
    }
    updateGL();
}
```

These three functions work in accordance. They are responsible for the animation. The sphere function has a radius that can be modify so the sphere volume can expand and contract. When clicking the button “animate” in the QT widgets the function animate(bool) emits a signal to the changeAmplitude() function which will modify the radius variable as a function of time. Then we have also implemented the stopAnimation(bool) function which only purpose is to stop the timer.

```
void MyGLWidget::openFile(bool){

        timer->stop(); //stop the timer from previous rendering
        QString fn = QFileDialog::getOpenFileName(this, tr("Open File..."),
QString(), tr(".xyz files (*.xyz);;All Files (*)"));
        filename = fn.toUtf8().constData(); //convert Qstring into std::string...
        animateZrot = 35;
        setXRotation(-102); //set the correct values to show the face nicely...
        setYRotation(10);
        setZRotation(35);
        setZoom(33);
}
```

The openFile(bool) function will start a QFileDialog widget and will allow to select a filename and save it as a Qstring variable called fn. This variable will be converted in a regular c++ string variable to be used in the drawFromFile() function that will draw the points contained in this specific file path. QFileDialog is a widget that was extremely useful in converting the application in more user friendly. This is a screen shot showing the QFileDialog window:




```
void MyGLWidget::drawPoint(int x, int y, int z)
{
    glPointSize(ledSize);
    glBegin(GL_POINTS);
    glColor4f(redcolor, greencolor, bluecolor, 0.75 );
    glVertex3f(x, y, z);
    glEnd();
}
```

The two main functions used to draw objects on Open GL use this function to draw the points in the space. Is just a trivial function that takes 3 integer as the coordinates for a point.

IV – Problems I have found and solutions:

Some of the problems I have found when designing this project are:

a) No Previous knowledge-experience in 3d: I have manage to learn what was necessary

to be able to develop a 3d application using OpenGL. I have learned how to use a modeling matrix and make transformations on it as well as how to use the view matrix and how it can be used to generate an effective viewport.

b) Centered Zooming: One of the biggest problems I have found along the road consisted in being able to zoom in and out maintaining the center. In previous test I have always showed an off-centered zooming that requires to perform some rotations in order to be able to see all the objects. I was able to find the correct calculations to show the element always centered in the viewport.

c) Hardware related issues: Another challenge I have found consisted in no being able to display functions using cubes in my hardware configuration. Originally I have created a code that will perform translations in the model matrix, and then redraw the cubes in the space, but later opted for a simpler way to draw the leds using just points since they offered similar properties than the cube: size, color blending and transparency.

d) Opening File Function: I wanted to created a function that will read any file that contain xyz coordinates but I have found that in QT environment some c++ language characteristics have changed. For example a simple string variable. QT offers a more complex string variable called QString, I have to do some reasearch and then find that I needed to convert this file into a native c++ string in order to use it for passing the name of the file. Then later I was able to use this string (which is the return from the QFileDialog) to perform the reading. I separated the variables into its x, y and z components and the drawing was done correctly.

e) Color Change: Another problem I have found was in regards of the colors of the light. When we want to be able to use blending we would expect to the able to create a function that can be connected to a QTWidget signal that can emit the three color parameters to be pass to the GL Object, but QT does not support this feature. The solution was to create one

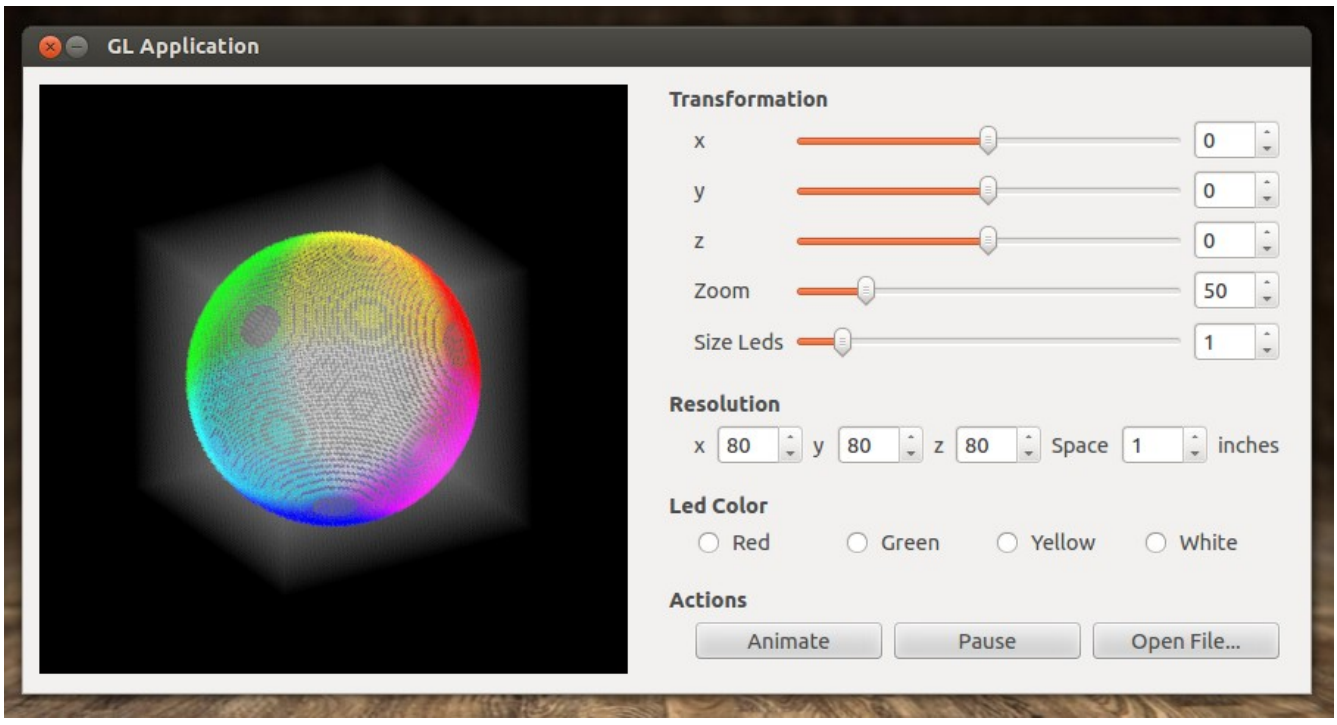
fucntion for each color, then each function will pass the blending to the GL object after glupdate().

V – Future Improvements:

a) Support for more animations: The real Led cube does provide several animations right out the box, in this application we have only provide one object with one animation, it would be great to add support for more.

b) More resolution for files that are being read: If we can create a histogram to map more data points into the Led Cube we can use the points given in the .xyz file as guide points, recreating the model with more resolution.

c) 3d Plotting support with color blending: It is possible to expand what we have at this moment to provide an input for function plotting, which will result of a great help to model mathematical 3d functions. We could also determine that if points are of a certain range we can color them differently. I have modify the code slightly to show a bit of more color blending.



d) Full Screen Open GL visualization: Another future improvement I will add would be the full screen feature. This will allow the user to be able to utilize the maximum resources of the system to accelerate and improve the GL object. We can also utilize more modern versions of GL (such as fragments rendering) in order to improve the overall full screen viewing experience (adding anti aliasing, for example).

VI – Outcome of the Learning experience:

a) Learning how to develop applications using QT, general frame of an application, widgets creation, signal-slots connections, specific QT Widgets: Buttons, spinboxes, groupboxes, labels, sliders, QdialogBox, QGLWidget, .

b) Creative thinking and general problem resolution.

c) Learning the basics of an Open GL application. General idea of an application in the space. GL state machine initialization, modification of model matrix and view matrix. GL scaling, mouse event integration, color blending, model drawing, model animation, orthographic projection, GL and QT integration with controls passed to the GL matrices (model and view).