

Computer Graphics – COMP3811

Coursework 2

Interactive Animated Scenes With OpenGL

1. Visual Scene and Background

The scene's context involves a pair of aliens that have come down to earth, specifically to a nightclub named "Planet Earth Disco" to party and live out their best lives. Despite their endearing appearance, they managed to capture the DJ, and will fly their spaceship back once they're done spinning and dancing to their favourite disco beats, How unfortunate for the DJ! Figure 1 shows the visual scene created with OpenGL.



Figure 1. Aliens are here! The visual scene created

In this section of the report we will talk about the background construction for this scene. The background consists of a room inside a nightclub, with complementary structures (walls, floor) that help visualise this indoor scene. The floor and wall are convex polygon structures built using `GL_Polygons`. Appropriate normals were applied to each of these polygon faces, for lightening purposes. Applying the wrong normals darkens the surface of the polygon structure, masking it from the light. Figure 2 shows a comparison between the shade of the wall when correct and incorrect normals (print pattern is darkened beyond recognition) are applied. Separate functions were created for these polygons, with an example of the floor function shown in figure 3. The functions created for this project were saved as private members inside the header file as well.

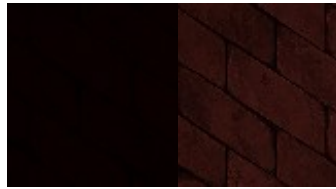


Figure 2. Stark contrast between incorrect (left) and correct (right) normal applied to polygon wall

Normals declared to be used for the polygon surfaces, with appropriate normal chosen for the polygon

The polygon is wrapped by a texture to create the pattern, and the right normals are applied to make the texture stand out. We will talk more about textures in section 4 of this report.

glBegin draws the primitive object GL_POLYGON; a convex polygon. glVertex3f provides the x,y,z coordinates for the polygon to be drawn. v

```
void AlienWidget::floor(){
    // Here are the normals, correctly calculated for the floor and wall faces
    GLfloat normals[][3] = { {1., 0., 0.}, {-1., 0., 0.}, {0., 0., 1.}, {0., 0., -1.}, {0., 1., 0.}, {0., -1., 0.} };
    glEnable(GL_TEXTURE_2D);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, _image.Width(), _image.Height(), 0,
        GL_RGB, GL_UNSIGNED_BYTE, _image.data);

    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    glNormal3fv(normals[4]);
    glBegin(GL_POLYGON);
        glTexCoord2f(0.0, 0.0);
        glVertex3f( 1.0, -1.0, 1.0);
        glTexCoord2f(1.0, 0.0);
        glVertex3f( 1.0, -1.0, -1.0);
        glTexCoord2f(1.0, 1.0);
        glVertex3f( -1.0, -1.0, -1.0);
        glTexCoord2f(0.0, 1.0);
        glVertex3f( -1.0, -1.0, 1.0);
    // glDisable(GL_TEXTURE_2D);
    glEnd();
    glDisable(GL_TEXTURE_2D);
}
```



Figure 3. Example of using Polygons to create the floor function

Similar function to figure 3 were also created for the wall functions. The design considered for these instances were that of a colourful floor pattern and a brick wall pattern to bring the visual scene to life and make it appropriate to the story we want to tell. Furthermore the background behind the room was set to black using `glClearColor` (value: 0.0, 0.0, 0.0, 0.1), to make the room stand out and give the illusion of viewing the room from the cameras perspective.

The instance of the object function (i.e. floor) is then called in the `paintGL` method. The `paintGL` method is provoked each time the widget needs painting and draws the object/functions declared inside it. Figure 4 shows the method `paintGL`. Since `OpenGL` is a state machine, we clear the colour and depth buffer first to start each frame with a clear state, as shown in figure 4. After setting the matrix mode to `modelview`, we can go ahead and call our instances. `Modelview` positions the geometry of the objects in the space to be viewed. The instance call referred by “this→floor();” (see figure 4) gets called, however we can manipulate the object to be rendered first by using functions such as `glScalef` (to stretch the object accordingly), `glTranslatef` (to place the object) and occasionally when required `glRotatef` (to rotate the object) around the x, y and z axis. By applying these functions we can make sure that the desired affect can be achieved, providing the illusion of these objects in a creative manner, before placing the call to the object. We place this between a push and pop matrix, so that these changes don’t apply to other objects in the `modelview` matrix.

```

void AlienWidget::paintGL()
{ // paintGL()
  // clear the widget
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  glEnable(GL_TEXTURE_2D);

  glShadeModel(GL_FLAT);
  // You must set the matrix mode to model view directly before
  glMatrixMode(GL_MODELVIEW);

  glMaterialfv(GL_FRONT, GL_AMBIENT,   white);
  glMaterialfv(GL_FRONT, GL_DIFFUSE,   white);
  glMaterialfv(GL_FRONT, GL_SPECULAR,   white);
  glMaterialf(GL_FRONT, GL_SHININESS,   1);

  glPushMatrix();
  glTranslatef(2.0,2.0,2.0);
  glScalef(2.0,2.0,2.0);
  this->floor();
  glPopMatrix();
}

```

Figure 4. The paintGL method explained and how instances of object functions are called in paintGL

2. Lightening and Colour Choice

In this section of the report we will talk about the lightening visuals and materials chosen to bring our scene to life. Figure 5 displays some of the colours and parameter attributes chosen for this project. The ambient, diffuse, specular are the RGBA (red, green, blue, alpha) reflectance of the material, and defined using a struct, which allow it to be used anywhere in the code. However we set the properties only in paintGL, due to our objects being drawn inside this particular method only. Figure 6 shows how materials and colours are set, either by using the pointer set to the global material, or in conjunction with the colours declared, or by using colours only (shown in figure 6 from top to bottom). The materials are declared just before calling an instance of the object (see figure 4 for example where a white colour is set before calling the floor function).

```

GLfloat cyan[] = {0.0, 1.0, 1.0, 0.0};
GLfloat gyellow[] = {0.5, 1.0, 0.0, 0.0};
GLfloat green[] = {0.0, 1.0, 0.0, 0.0};
GLfloat orange[] = {1.0, 0.5, 0.0, 0.0};
// Setting up material properties
typedef struct materialStruct {
  GLfloat ambient[4];
  GLfloat diffuse[4];
  GLfloat specular[4];
  GLfloat shininess;
} materialStruct;

static materialStruct turqMaterials = {
  { 0.1, 0.18725, 0.1745, 0.8 },
  { 0.396, 0.74151, 0.69102, 0.8 },
  { 0.297254, 0.30829, 0.306678, 0.8 },
  12.8
};

```

Figure 5. How colours and parameter values are declared as global.

```

p_front = &whiteShinyMaterials;

glMaterialfv(GL_FRONT, GL_AMBIENT,   p_front->ambient);
glMaterialfv(GL_FRONT, GL_DIFFUSE,   p_front->diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR,   p_front->specular);
glMaterialf(GL_FRONT, GL_SHININESS,   p_front->shininess);

p_front = &jadeMaterials;

glMaterialfv(GL_FRONT, GL_AMBIENT,   red);
glMaterialfv(GL_FRONT, GL_DIFFUSE,   p_front->diffuse);
glMaterialfv(GL_FRONT, GL_SPECULAR,   p_front->specular);
glMaterialf(GL_FRONT, GL_SHININESS,   1);

glMaterialfv(GL_FRONT, GL_AMBIENT,   red);
glMaterialfv(GL_FRONT, GL_DIFFUSE,   red);
glMaterialfv(GL_FRONT, GL_SPECULAR,   red);
glMaterialf(GL_FRONT, GL_SHININESS,   128);

```

Figure 6. How lightening materials are set up in the paintGL method before calling the object so that the right colour/materials are applied to it

Shininess of the material was also chosen between the values of 0-128.

The colour and material choices were carefully chosen to create a suitable scenario for our visualisation. Almost neon like colours were chosen for the spaceship, and bright colours alike were picked for the alien structures. The objects where textures were applied were drawn using a white colour, to make the texture project its true image onto the opengl object. Furthermore materials such as Ruby, Emerald, Turquoise were used in some instances or experimented with to get the desired outcome for the scene.

3. Hierarchical Structures

Hierarchical structures were created for the purpose of this assignment. The structures were built from glu and glut objects where fitting. The glu objects were created through a quadric object, where a pointer was linked to the quadric (see figure 8). These objects were also destroyed to free up memory and enhance performance. The declaration of these functions can be seen in figure 7.

```
pspaceship = gluNewQuadric();
ptop = gluNewQuadric();

gluDeleteQuadric(pspaceship);
gluDeleteQuadric(ptop);
```

Figure 7. How glu objects were created in the main widget file, using the spaceship structure as example.

```
GLUQuadricObj* pspaceship;
GLUQuadricObj* ptop;
```

Figure 8. How the quadric object to be used is declared in the header file for the widget.

Below we will explore the design chosen and implementation of the hierarchical structures in the graphical scene.

Spaceship



```
void AlienWidget::spaceship() {
    gluQuadricNormals(pspaceship, GL_SMOOTH);
    gluCylinder(pspaceship, 0.5, 0.6, 0.9, 60.0, 60.0);
}

void AlienWidget::top() {
    gluQuadricNormals(ptop, GL_SMOOTH);
    gluCylinder(ptop, 0.001, 0.6, 0.9, 60.0, 60.0);
}

glPushMatrix();
glScalef(0.3, 1.0, 0.4);
glTranslatef(5.8, 0.4, 1.5);
glutSolidCube(0.5);
glPopMatrix();
```

Figure 9 and 10 . Spaceship built on the left (9) and its implementation on the right (10)

The spaceship structure comprises of gluCylinder objects forming the spaceship body and the roof. Two separate functions were created, as can be seen in figure 10 (upper part), and the parameters for the gluCylinder were manipulated to produce the results we want. For instance, in figure 10 (upper part), we can see a clear difference between the base value which is set at 0.5 for the spaceship and 0.001 for the top part of the ship, to get the desired effect. The Cylinders were also linked to the pointer quadric object. Then in paintGL we were able to transform the cylinders by placing the top on the body and putting them in the correct location using glTranslatef. The spaceship also consists of four glutSolidCube objects, placed sideways acting as the arms. These solidcubes were called in paintGL as shown in figure 10 (lower part). The colours chosen were placed before each push and pop matrix to call the objects.

Disco Ball

```
void AlienWidget::discoball(){
    glEnable(GL_TEXTURE_2D);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, _image2.Width(),
    gluQuadricTexture(pdiscoball, GL_TRUE);
    gluQuadricNormals(pdiscoball, GL_SMOOTH);
    gluSphere(pdiscoball,0.5,8,8);

    glDisable(GL_TEXTURE_2D);
}

void AlienWidget::rod() {
    gluCylinder(prod, 0.05, 0.05, 4.0, 16.0, 1.0);
}
```

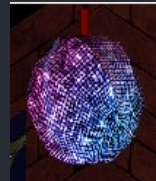


Figure 11. The disco ball on the right and its implementation on the left

The disco ball shown in figure 11 has been created using gluSphere object. The parameter values for gluSphere are the radius, slices and stacks. These were set accordingly, to create the right size for the disco ball and some elements of roughness of the sphere, by keeping the slices and stacks value at 8 (figure 11 code on the left, see gluSphere). We wanted to produce a glittery effect and therefore used a quadric normal set to smooth, as well as a texture. The colour and lightening materials were set to “White shiny material”, to bring out the effect of the disco ball. The disco rod above the ball was designed using gluCylinder, by manipulating the cylinder parameters. A function call was placed in paintGL between push and pop matrix stack for both the disco ball and the rod.

Alien

```
void AlienWidget::body(){
    gluQuadricNormals(pbody, GL_SMOOTH);
    gluSphere(pbody,0.3,60.0,60.0);
}

void AlienWidget::antenna() {
    gluQuadricNormals(pantenna, GL_SMOOTH);
    gluCylinder(pantenna, 0.05, 0.05, 0.1, 16.0, 1.0);
}

void AlienWidget::eye() {
    gluQuadricNormals(peye, GL_SMOOTH);
    gluSphere(peye,0.08,60.0,60.0);
}
```

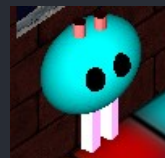


Figure 12. The Alien structure on the right with the design functions on the left

The alien structure design was first sketched out in rough (pencil and paper) and then implemented using the functions listed in figure 12. `gluSphere` was used to create the body, while in `paintGL` we used `glScalef` to stretch the sphere to achieve the body shape required. Another instance of the Sphere was used to get create the eyes. The radius was set small and in `paintGL` the eyes were pushed into the body structure using `glTranslatef`. The antennas on top of the body were created using `gluCylinder`, and morphed onto the bodyshape by applying the correct transformation function to it. `GIRotation` was also applied where needed. The legs were created using `glutSolidcube` objects.

4. Texture

Textures were a key part to produce the results required for this assignment. A total of five textures were used. Two of these came from the Texture directory provided, and to be used for this coursework. The theme therefore centred around these two images, portraying a story.



Figure 13. The textures provided for this coursework

To use a number of different images, a function was created, which allowed binary integers of the image file to be loaded and to be read (using `GLubyte`), through specifying the width and height of the image pixels (figure 14, left). A function called `imagefield` as shown on the right in figure 14, returns the image.

```

unsigned int nm = _width*_height;
for (unsigned int i = 0; i < nm; i++){
    std::div_t part = std::div((int)i, (int)_width);
    QRgb colval = p_qimage->pixel(_width-part.rem-1, part.q);
    _image[3*nm-3*i-3] = qRed(colval);
    _image[3*nm-3*i-2] = qGreen(colval);
    _image[3*nm-3*i-1] = qBlue(colval);
}

const GLubyte* Image::imageField() const
{
    return _image;
    return _image3;
    return _image4;
    return _image5;
}

Image::~Image()
{
    delete _image;
    delete _image3;
    delete _image4;
    delete _image5;
    delete p_qimage;
}

```

Figure 14. Reading in the pixels using width and height (left) and then returning image inside function `Imagefield` (right) (done in the file `Image.cpp`)

The image returned can then be declared in the widget header file and subsequently used in the main widget file as shown in figure 15, QImage class was also employed in our application to render the image and load the texture.

```
Image _image4;
Image _image5;

QImage* p_qimage;

_image4("./earth.ppm"),
_image5("./wall.jpg")
```

Figure 15. Image declared using QImage in the widget header file (left) and how the specific image to be used is specified in main widget (right)

```
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_NEAREST);
```

Figure 16. How texture mapping is done. This was declared inside the resizeGL method (which is called each time the widget is resized) of the main widget, and does not need to be declared again

```
void AlienWidget::walltext1(){
    glEnable(GL_TEXTURE_2D);
    glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB, _image4.Width(), _image4.Height(), 0, GL_RGB, GL_UNSIGNED_BYTE, _image4.imageField());

    glPolygonMode(GL_FRONT_AND_BACK, GL_FILL);
    glNormal3f(0.0,0.0,1.0);
    glBegin(GL_POLYGON);
    glTexCoord2f(0.0, 0.0);
    glVertex3f( -1.0, -1.0, 1.0);
    glTexCoord2f(1.0, 0.0);
    glVertex3f( -1.0, -1.0, -1.0);
    glTexCoord2f(1.0, 1.0);
    glVertex3f( -1.0, 1.0, -1.0);
    glTexCoord2f(0.0, 1.0);
    glVertex3f( -1.0, 1.0, 1.0);
    glEnd();
    glDisable(GL_TEXTURE_2D);
}
```

Setting the current texture coordinates for the vertices to create the polygon in this case

GLTexImage creates the texture object and we can see Imagefield being used to return the specified image inside its parameters

Figure 17. An example on how a texture is applied to the object function

Important to use glEnable and glDisable for the texture specified so that it doesnt affect the rest of the functions.

5. Interactive Element

The interactive element used in our implementation is that of a slider, which allows the user to move the spaceship in a linear upwards direction, as shown in figure 18.

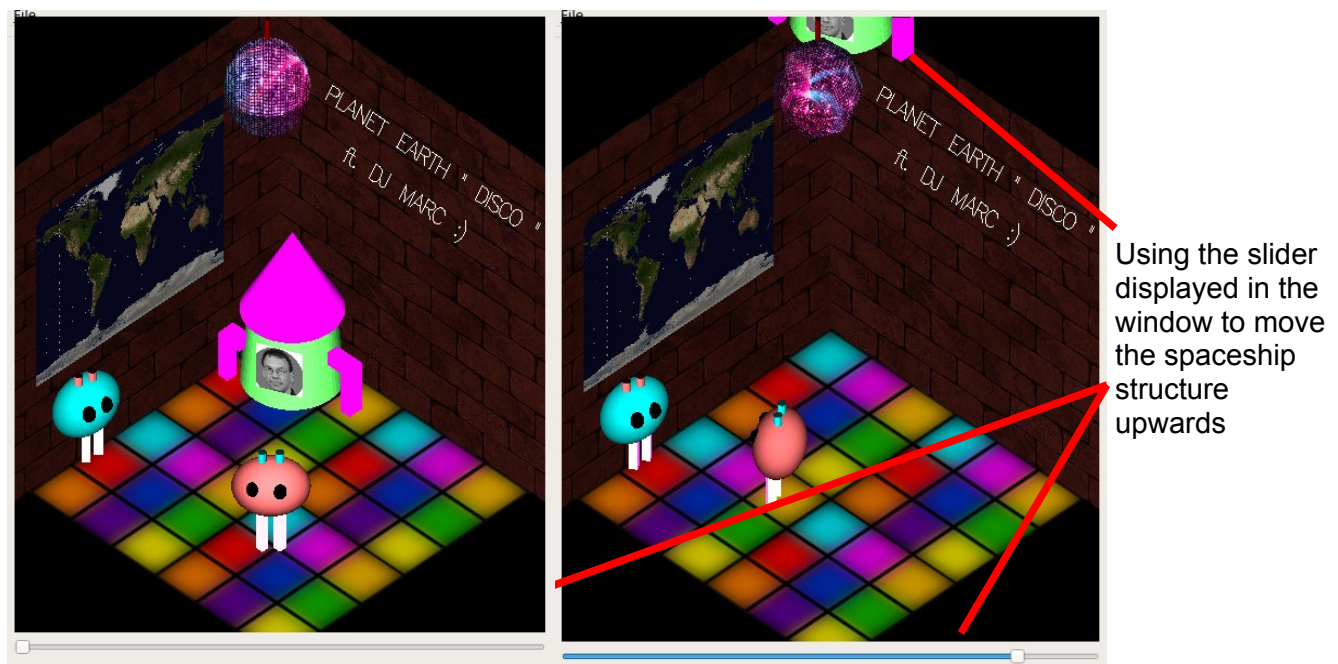


Figure 18. The use of slider as an interactive element in our graphics application

QSlider was utilised for this operation, where a slider is created in the main window of the application using signal and slot, as seen in figure 19. The slot refers to the function created in main widget called “updateFlying”, seen in figure 20.

```
// create slider
nVerticesSlider = new QSlider(Qt::Horizontal);
connect(nVerticesSlider, SIGNAL(valueChanged(int)), cubWidget, SLOT(updateFlying(int)));
windowLayout->addWidget(nVerticesSlider);
```

Figure 19. Slider created and connected using SIGNAL and SLOT operation

```
void AlienWidget::updateFlying(int i){
    flying = i;
    this->repaint();
}
```

Figure 20. The function created in main widget

This function is also declared as a public slot in the header widget file, where the int variable “flying” is also declared and set to 0 in main widget. The function “updateFlying” uses this variable to increment the value as seen in figure 20. This is then used in paintGL inside a push and pop

matrix, and manipulates the vertical y direction using `glTranslatef` as can be seen in figure 21. Inside the push and pop matrix we declare additional object calls to spaceship members, so that the whole of the structure can move upwards.

```
glPushMatrix();  
glTranslatef(0.0, 0.05*flying, 0.0);  
  
p_front = &emeraldMaterials;
```

Figure 21. How the “flying” int value is used inside `paintGL` to increment the y position with the slider for the structures to be defined underneath the push matrix

6. Animation

Animation in the scene is present between three of the structures created; the disco ball spinning, the pink coloured alien rotating around the scene in a circular motion to show off its dance moves, and the legs of both aliens rotating about the body.



This alien is rotating its body in a circular motion

The legs rotate about the body for both aliens

Figure 22. The rotating alien

The implementation for this is done through the use of a timer using the class `QTimer` to create the animation. As displayed in figure 23 below, the timer pointer is declared, which is done in the main window file.

```
ptimer = new QTimer(this);  
connect(ptimer, SIGNAL(timeout()), cubWidget, SLOT(updateAngle()));  
ptimer->start(0);
```

Figure 23. Using a timer with `SIGNAL` and `SLOT` connected to the function to be used in main widget

The function “`updateAngle`” is created in main widget (figure 24) and called by the timer in the main window. In the header file for the widget, the function is declared as a public slot and the angle “angle” in the function is also set as a float value inside the header file. The angle is initially declared as 0 in the constructor of the main widget, but incremented by a value of 2.5 in the function. This value of “angle” can then be used to manipulate the rotations of the objects. For instance, for the alien body, we place a `glRotatef` with the angle manipulating the y axis between a push and pop matrix. The remaining body parts of the alien can then be defined inside this push

and pop stack, to make sure rotation occurs for every part of the alien body structure. Furthermore to determine the radius of rotation, we place it between two glTranslatef functions. This is show is figure 25. Single elements such as the alien legs and disco ball are animated using the value of angle inside a glRotatef in their respective push and pop matrices. An example of how rotation of the legs are brought about is show in figure 26.

```
void AlienWidget::updateAngle(){
    angle += 2.5;
    this->repaint();
}
```

Figure 24. The function in the main widget

```
glPushMatrix();
glTranslatef(2.5,0.0,2.5);
glRotatef((double)angle,0.,1.,0.);
glTranslatef(-2.5,-0.0,-2.0);
```

Figure 25. How the angle value is used to cause rotation for the whole alien structure

```
glPushMatrix();
glScalef(0.2,0.9,0.2);
glTranslatef(15.9,0.6,10.0);
glRotatef((double)angle,0.,1.,0.);
glutSolidCube(0.5);
glPopMatrix();
```

Figure 26. How the angle value is used to cause rotation for a single part of the complete structure

7. Additional - Text on the wall and directory structure

The text on the wall inside the club was designed for further context to the scene, its function implementation can be seen in figure 27 and 28 below. Glu Stroke character was used in this context (figure 28) to render the text and set the font as well.

```
void AlienWidget::text(){
    char buff[80];
    strcpy(buff,"PLANET EARTH '' DISCO '');
    int len;
    len = strlen(buff);
```

Figure 27. The string created to be displayed

```
glTranslatef(175.0,260.0,1.0);
glScalef(0.08,0.08,0.08);
glRotatef(-30.,0.,0.,1.);

for ( int i = 0; i < len; ++i )
{
    glutStrokeCharacter(GLUT_STROKE_ROMAN, buff[i]);
}
```

Figure 28. Transformation applied to display the text the way we want

The directory contains the following:

- Main file - To create the window
- Window Header and Main – Layout of window widget
- Widget Header and Main – Code to create the scene
- Images – For texturing purposes
- README – Brief context and program execution instructions
- Makefile, .pro file and additional self generated files i.e. moc files
- Report in PDF