COMP3811: Computer Graphics
Coursework 2
Interactive Animated Scenes with OpenGL

Juan Camacho
sid: 201356237
sc19jcm

January 8, 2022

Contents

1 Introduction

Space exploration allows humans to prove or disprove theories that scientists rise, and the technology takes credit for it. Space simulations are used a many research areas to make future predictions from data. From the given textures for this project, the Earth map texture immediately gave me the idea to create a simulation of our solar system.

1.1 Aims

- Meet the requirements of each band of the project specification.
- Create a scene using real physics formulas to generate the movement of the space objects.

1.2 Context

For a better understanding of the context of this project, the final result is given in Figure 1. This will hopefully justify the design choices on each phase of the development.

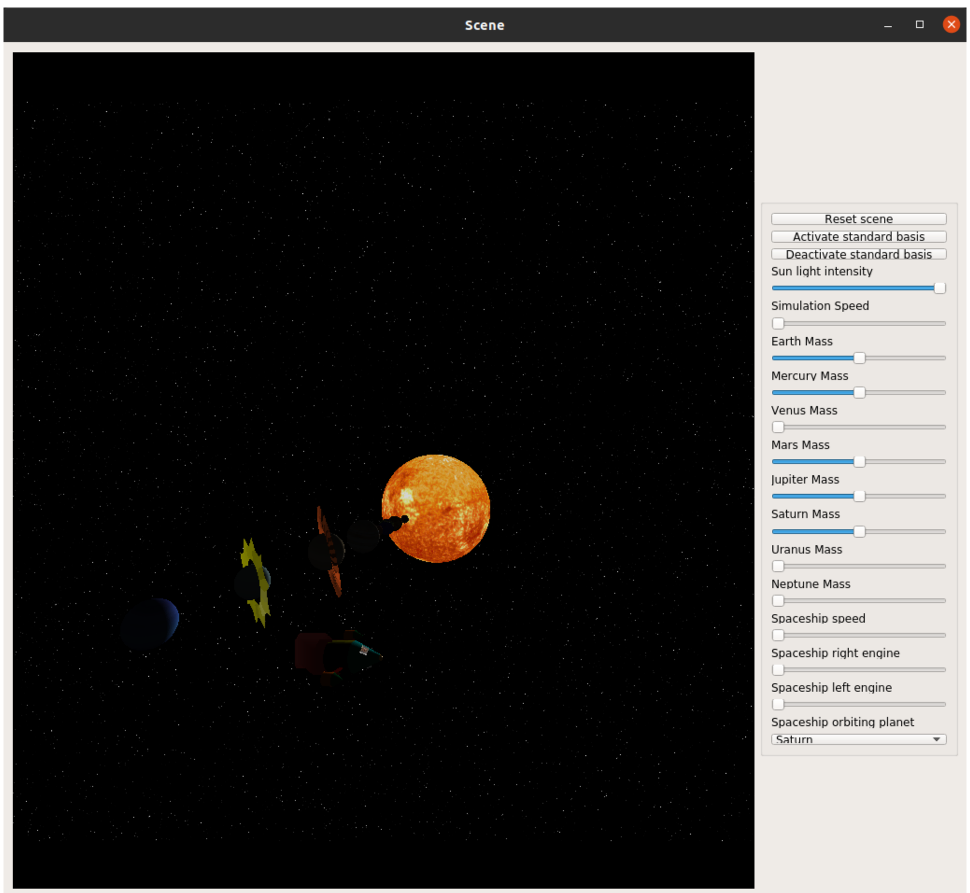

Figure 1: Screenshot of the final product

2 Band 4 (40%-50%)

The requirements of band 4 were to create a reasonable complex scene using instancing of different objects, and adding light and material properties to the scene that allow for diffuse and specular lighting.
A rough layout of the scene and the objects in it was drawn:

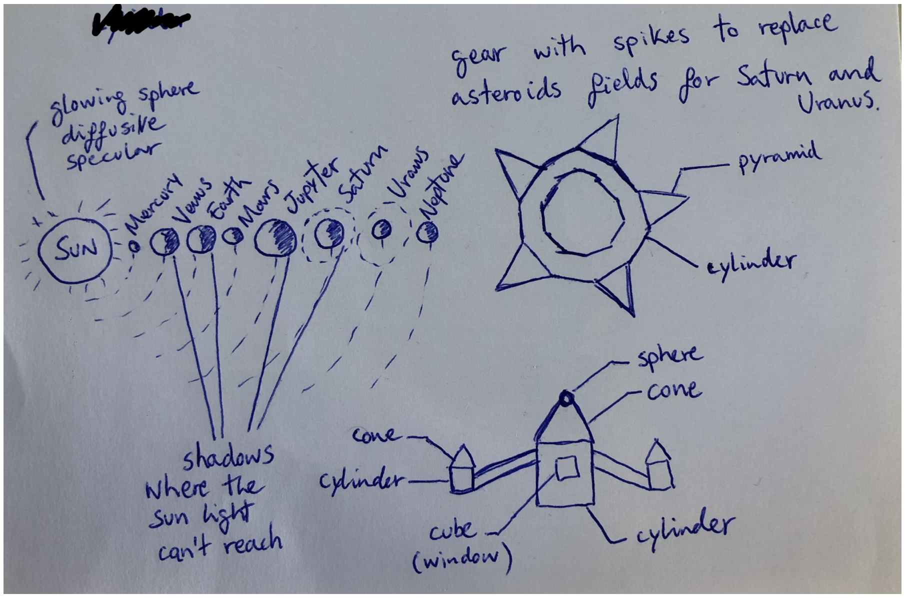

Figure 2: Draft of the complex scene

2.1 Basic objects

To make the instantiation of objects easier and in order to build more complex ones, primitive shapes were defined and implemented as described in Table 1.

Shape	Implementation	Orientation	Description
cube	GL_POLYGON	outside normals	taken from COMP3811 tutorials.
cylinder	GL_POLYGON	outside normals	taken from COMP3811 tutorials.
sphere	gluSphere object	outside normals	quadrics object
disk	gluDisk object	outside normals	quadrics object
cone	GL_POLYGON	outside normals	derived from COMP3811 tutorials cylinder.
pyramid	GL_POLYGON	outside normals	square based pyramid, built within implementation of the complex gear object.

Table 1: Basic shapes

The choice for these shapes was decided from the initial design of the scene 2. The Sun is considered to be the light source of the scene, thus the normal vectors of the polygons point outwards. The declarations of the shapes can be found in Figure 3, and the implementation of the new polygons such as the gear (containing pyramids) and the cone is in Figures 4 and 5. Initial instances of the development of the cone and the gear can be seen in Figures 6 and 7.

```
#ifndef A25831C8_51A5_403F_B582_7BD4DC518B2A
#define A25831C8_51A5_403F_B582_7BD4DC518B2A

#include <GL/gl.h>

static const float PI = 3.1415926535;

/*
  materialStruct

  Lighting values of the materials

*/
You, 18 hours ago | 1 author (You)
typedef struct materialStruct {
  GLfloat ambient[4];
  GLfloat diffuse[4];
  GLfloat specular[4];
  GLfloat shininess;
} materialStruct;

/*
  cube

  PARAMS
    p_front     the material properties of the object
    flag        determines whether to apply markus.ppm or Marc_Dekamps.ppm texture
    texID       texture ID to bind

*/
void cube(const materialStruct* p_front, int flag, GLuint* texID);

/*
  cylinder
  
  PARAMS
    p_front     the material properties of the object

*/
void cylinder(const materialStruct* p_front);

/*
  openedCone
  
  PARAMS
    p_front     the material properties of the object
    r           radius of one of the sides of the cylinder. 
                If radius is large, then it will be a deformed cylinder, 
                else it becomes a cone

*/
void openedCone(const materialStruct* p_front, float r);

/*
  gear
  complex shape that uses the cylinder implentation together with a 
  pyramid polygon to create a gear-like object.

  PARAMS
    p_front     the material properties of the object
    n_spikes    determines the number of pyramids (spikes) that the gear will have.

*/
void gear(const materialStruct* p_front, int n_spikes);

#endif /* A25831C8_51A5_403F_B582_7BD4DC518B2A */
```

Figure 3: Definitions in `utils/Shapes.h`

```cpp
void openedCone(const materialStruct* p_front, float r){

        int N           = 100;      // number of faces
        int n_div    =  1;         // number of height divisions

      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);

    float x0, x1, y0, y1;

    float z_min = -1;
    float z_max = 1;

    float delta_z = (z_max - z_min)/n_div;

    for (int i = 0; i < N; i++){
        for(int i_z = 0; i_z < n_div; i_z++){
            x0 = cos(2*i*PI/N);
            x1 = cos(2*(i+1)*PI/N);
            y0 = sin(2*i*PI/N);
            y1 = sin(2*(i+1)*PI/N);

            float z = 0;
            glBegin(GL_POLYGON);
            glVertex3f(x0,y0,z);
                glNormal3f(x0,y0,0);
            glVertex3f(x1,y1,z);
                glNormal3f(x1,y1,0);
            glVertex3f(r*x1,r*y1,z+delta_z);  // apply radius to opposite side of cylinder
                glNormal3f(x1,y1,0);
            glVertex3f(r*x0,r*y0,z+delta_z);  // apply radius to opposite side of cylinder
                glNormal3f(x0,y0,0);
            glEnd();
        }
    }
}
```

Figure 4: `openedCone()` implementation, `utils/Shapes.cpp`

```
  float z_min = 0.;
  float z_max =  1.;  // depth 1.0 max
  float r = 1.5      // radius
  float delta_z = (z_max - z_min)/n_div;
  
  for (int i = 0; i < N; i++){
glBegin(GL_POLYGON);
    for(int i_z = 0; i_z < n_div; i_z++){
      x0 = cos(2*i*PI/N);
      x1 = cos(2*(i+1)*PI/N);
      x2 = r*cos(2*i*PI/N);
      x3 = r*cos(2*(i+1)*PI/N);
      y0 = sin(2*i*PI/N);
      y1 = sin(2*(i+1)*PI/N);
      y2 = r*sin(2*i*PI/N);
      y3 = r*sin(2*(i+1)*PI/N);

      // cylinder starts
      glVertex3f(x0,y0,0);    glNormal3f(x0,y0,0);
      glVertex3f(x2,y2,0);    glNormal3f(x2,y2,0);
      glVertex3f(x2,y2,0.5);  glNormal3f(x2,y2,0.5);
      glVertex3f(x0,y0,0.5);  glNormal3f(x0,y0,0.5);

      glVertex3f(x2,y2,0);    glNormal3f(x2,y2,0);
      glVertex3f(x3,y3,0);    glNormal3f(x3,y3,0);
      glVertex3f(x3,y3,0.5);  glNormal3f(x3,y3,0.5);
      glVertex3f(x2,y2,0.5);  glNormal3f(x2,y2,0.5);

      glVertex3f(x3,y3,0);    glNormal3f(x3,y3,0);
      glVertex3f(x1,y1,0);    glNormal3f(x1,y1,0);
      glVertex3f(x1,y1,0.5);  glNormal3f(x1,y1,0.5);
      glVertex3f(x3,y3,0.5);  glNormal3f(x3,y3,0.5);

      glVertex3f(x1,y1,0);    glNormal3f(x1,y1,0);
      glVertex3f(x0,y0,0);    glNormal3f(x0,y0,0);
      glVertex3f(x1,y1,0.5);  glNormal3f(x1,y1,0.5);
      glVertex3f(x0,y0,0.5);  glNormal3f(x0,y0,0.5);
      // cylinder ends

      // keep building on top of cylinder
      if (i%2==0)
      { // pyramid starts
        x0 = x2; // old x2
        x1 = x3; // old x3
        x2 = 2.*cos(2*i*PI/N);
        x3 = 2.*cos(2*(i+0.5)*PI/N);
        y0 = y2;
        y1 = y3;
        y2 = 2.*sin(2*i*PI/N);
        y3 = 2.*sin(2*(i+0.5)*PI/N);

        // join the two points int he middle to make a triangle
        x2 = x3;
        y2 = y3;
        float z = 0.25; // where the pyramid's top common vertex meets

        glVertex3f(x0,y0,0);    glNormal3f(x0,y0,0);
        glVertex3f(x2,y2,z);    glNormal3f(x2,y2,0);
        glVertex3f(x2,y2,z);  glNormal3f(x2,y2,0.5);
        glVertex3f(x0,y0,0.5);  glNormal3f(x0,y0,0.5);

        glVertex3f(x2,y2,z);    glNormal3f(x2,y2,0);
        glVertex3f(x3,y3,z);    glNormal3f(x3,y3,0);
        glVertex3f(x3,y3,z);  glNormal3f(x3,y3,0.5);
        glVertex3f(x2,y2,z);  glNormal3f(x2,y2,0.5);

        glVertex3f(x3,y3,z);    glNormal3f(x3,y3,0);
        glVertex3f(x1,y1,0);    glNormal3f(x1,y1,0);
        glVertex3f(x1,y1,0.5);  glNormal3f(x1,y1,0.5);
        glVertex3f(x3,y3,z);  glNormal3f(x3,y3,0.5);

        glVertex3f(x1,y1,0);    glNormal3f(x1,y1,0);
        glVertex3f(x0,y0,0);    glNormal3f(x0,y0,0);
        glVertex3f(x1,y1,0.5);  glNormal3f(x1,y1,0.5);
        glVertex3f(x0,y0,0.5);  glNormal3f(x0,y0,0.5);
      } // pyramid ends
    }
glEnd();
  }
```

Figure 5: `gear()` implementation in `utils/Shapes.cpp`. Top part of the function initialises variables and materials properties.

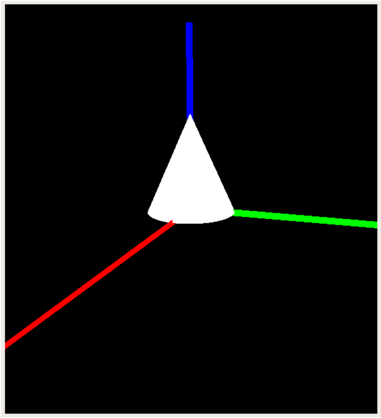

Figure 6: Instance of a cone object
(early implementation)

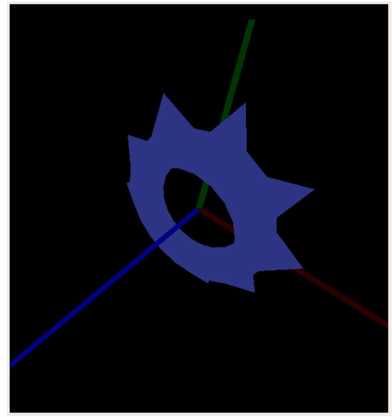

Figure 7: Instance of a gear object
(early implementation)

2.2 Complex scene

The complex scene was built with instances of the basic shapes defined in Table 1. the scene is comprised of a centered star, the Sun, and eight planets: Mercury, Venus, Earth, Mars, Jupiter, Saturn, Uranus and Neptune. An initial stage of the scene development is shown in Figure 8. Planets were obtained from the result of applying their corresponding textures to each `gluSphere()` object.

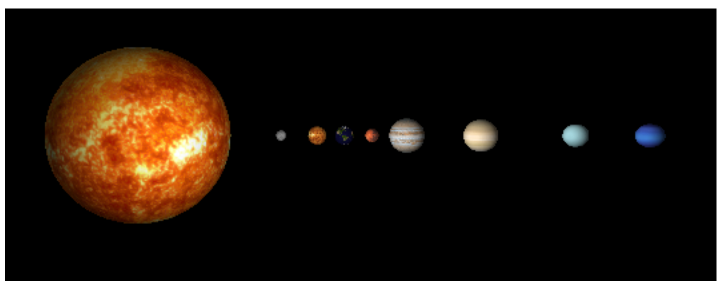

Figure 8: Solar system early stage

A spaceship, shown in Figures 9 and 10, was created from the basic shapes with the mission to carry Marc and Markus around the different planets. The ring with spikes (`gear()` in Figures 3 and 5) that simulates the fields of asteroids of Saturn and Neptune is shown in Figure 11. The solar system scene is contained within a cubemap of stars which defines an infinite space background, Figure 12 shows the rendered background.

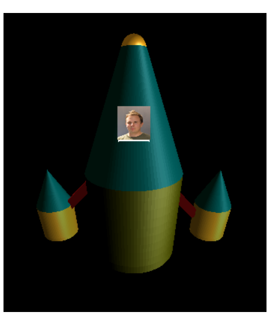

Figure 9: Spaceship front

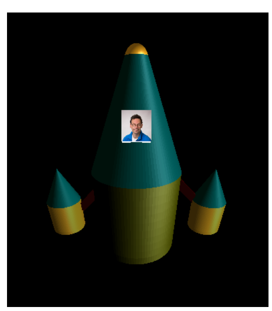

Figure 10: Spaceship back

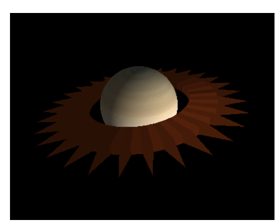

Figure 11: Saturn's ring

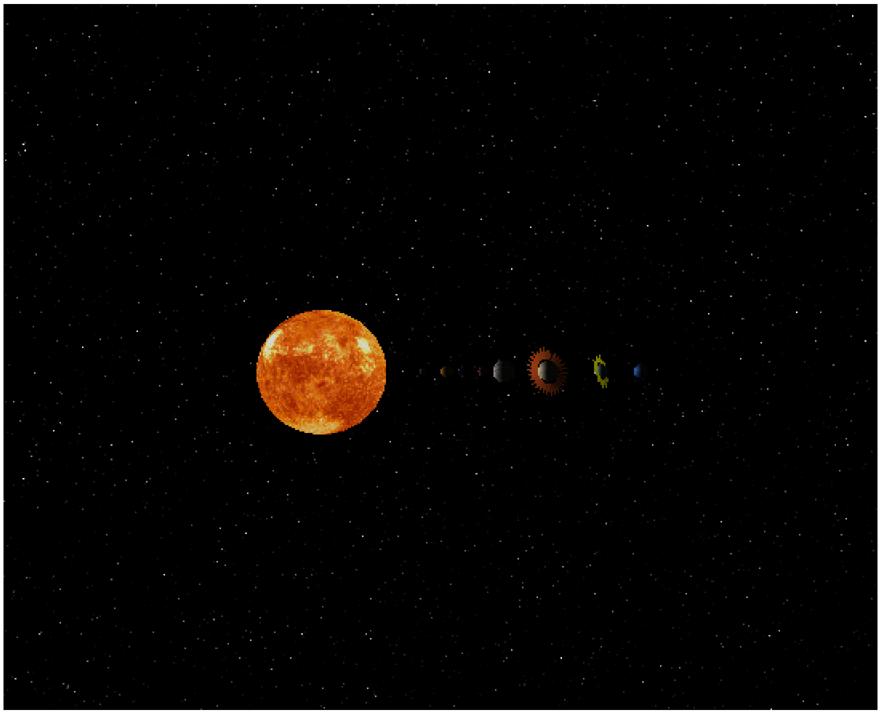

Figure 12: Solar system within a cubemap simulating other stars away

Materials are defined in 20 of type `materialStruct` (declared in 3) to be applied to the complex shapes to generate the ambient, diffuse, specular and shiny lighting properties of each of the objects.

The following subsections give a more detailed explanation of the transformations that the objects in the scene involve.

2.2.1 Planets

Each planet is an instance of `gluSphere()` with its correspondent planet texture, each planet with `GLU_SMOOTH` `gluQuadricNormals` normals. The sun is in the center of the standard basis, the rest of the planets are transformed in order from closest to furthest according to their distances from the Sun, on the positive x direction in the start of the simulation. Each planet, including the Sun, has its own rotation around the up vector, which is the y-axis. `drawPlanet()` (defined in Figure 16) instantiates a planet in the scene with the given parameters to calculate their orbit, mass, momentum, radius of sphere and their texture ID. Each planet is of type `PlanetWidget` which is declared in `PlanetWidget.h` and defined in Figure 13.

2.2.2 Spaceship

An instance of a `cylinder()` (Figure 3) is used with `yellowPlastic` material properties as the main body of the craft. It has a instance of an `openedCone()` (Figure 3) with `cyanPlastic` material properties on top of the cylinder. The `openedCone` instance is instantiated such that one of the polygon's opposite ends has a different radius to the other end. Therefore, in this case, it is an opened cone such that a `gluSphere()` object can fit at the end of the polygon to create the tip of the craft. The bottom of the `cylinder` is covered by a `gluDisk()` instance with `ruby` material properties. The spaceship is comprised of two engines placed on opposite sides, each of them is built by a `cylinder()` instance with a `openedCone` instance on top with `brassMaterials` and `cyanPlastic` material properties respectively. Both `openedCone` instances are created with a very small radius by their top end such that they look as if they were a cone. The bottom of each of the engines is covered by an instance of `gluDisk()` with material properties `polishedCopper`. Both engines are joined to the main body of the spaceship through scaled `cube()` instances that result in rectangles. The `drawSpaceship()` function declared in Figure 14 with an extense declaration in `Spaceship.cpp` instantiates the spaceship in the scene. Part of its declaration can be seen in Figure 38 that shows the transformations, rotations, and basic objects by which the right engine is comprised of.

2.2.3 Gear

Saturn and Neptune have asteroid fields which in this project were simulated by "gears with spikes", instanced by `gear()` in 5. Saturn's ring shown in Figure 11 with material properties `polishedCopper` is comprised of a cylinder with a top layer of pyramids (Figure 5, lines 173-207), these were implemented within `gear()` function. In the other hand, Neptune's gear has material properties `yellowPlastic`. The rings are translated to their planet's coordinates and have a radius larger than their the planet that they orbit. The chosen materials might not match the colour of the planet, but they respond to lighting conditions with vibrant colours that catch our eyes.

2.2.4 Background

A cubemap of 6 textures is loaded by `loadCubemap()` and rendered by `renderCubemap` (Figure 15), both declared in (declared in `SolarSystemWidget.cpp`). A single instance of `renderCubemap()` shown in Figure 12 renders a cubemap which is scaled to contain the solar system. The textures of the cubemap represent an infinite amount of bright stars, without any light properties attributed to it.

Textures for the planets and for the cubemap were obtained from [2]

```cpp
#include <GL/glu.h>
#include "PlanetWidget.h"

// constructor
PlanetWidget::PlanetWidget(QWidget *parent, GLuint texID, double mass, double radius,
                           double* pose, double* momentum, int i)
  : QGLWidget(parent),
    _texID(texID),
    _radius(radius),
    _mass(mass),
    index(i)
{ // constructor

  _pose[0] = pose[0];
  _pose[1] = pose[1];
  _pose[2] = pose[2];

  _momentum[0] = momentum[0];
  _momentum[1] = momentum[1];
  _momentum[2] = momentum[2];

} // constructor
```

Figure 13: `PlanetWidget` class defined in `PlanetWidget.cpp`

```c
#ifndef A9E36041_4130_4D53_8670_2278B8F39A6E
#define A9E36041_4130_4D53_8670_2278B8F39A6E

#include "utils/Materials.h"
#include "utils/Shapes.h"
#include "GL/gl.h"
#include "GL/glu.h"

/*
  drawSpaceship

  PARAMS
    GLuint* texID               array of texture IDs
    float _time_sinewave        input value to the sine wave function determined by _time increments
    float left                  angle of rotation for the left engine
    float right                 angle of rotation for the right engine
*/
void drawSpaceship(GLuint* texID, float _time_sinewave, float left, float right);

/*
  left_engine

  PARAMS
    float                       angle of rotation for the left engine
*/
void left_engine(float);

/*
  right_engine

  PARAMS
    float                       angle of rotation for the right engine
*/
void right_engine(float);

#endif /* A9E36041_4130_4D53_8670_2278B8F39A6E */
```

Figure 14: Definitions of `spaceship` in `Spaceship.h`

```
void SolarSystemWidget::renderCubemap()
{
  glPushMatrix();
  glScalef(9.,9.,9.);   // scale it to make it the background of the scene
  // Render the right quad
    glBindTexture(GL_TEXTURE_CUBE_MAP, texIDcube);
    glBegin(GL_QUADS);
    glTexCoord3f(  0.5f, -0.5f,  0.5f );  glVertex3f(  0.5f, -0.5f,  0.5f );
    glTexCoord3f(  0.5f, -0.5f, -0.5f );  glVertex3f(  0.5f, -0.5f, -0.5f );
    glTexCoord3f(  0.5f,  0.5f, -0.5f );  glVertex3f(  0.5f,  0.5f, -0.5f );
    glTexCoord3f(  0.5f,  0.5f,  0.5f );  glVertex3f(  0.5f,  0.5f,  0.5f );
    glEnd();
  // Render the left quad
  glBegin(GL_QUADS);
    glTexCoord3f(  -0.5f, -0.5f,  0.5f ); glVertex3f(  -0.5f, -0.5f,  0.5f );
    glTexCoord3f(  -0.5f, -0.5f, -0.5f ); glVertex3f(  -0.5f, -0.5f, -0.5f );
    glTexCoord3f(  -0.5f,  0.5f, -0.5f ); glVertex3f(  -0.5f,  0.5f, -0.5f );
    glTexCoord3f(  -0.5f,  0.5f,  0.5f ); glVertex3f(  -0.5f,  0.5f,  0.5f );
  glEnd();
  // Render the top quad
  glBegin(GL_QUADS);
    glTexCoord3f( -0.5f,  0.5f, -0.5f ); glVertex3f( -0.5f,  0.5f, -0.5f );
    glTexCoord3f( -0.5f,  0.5f,  0.5f ); glVertex3f( -0.5f,  0.5f,  0.5f );
    glTexCoord3f(  0.5f,  0.5f,  0.5f ); glVertex3f(  0.5f,  0.5f,  0.5f );
    glTexCoord3f(  0.5f,  0.5f, -0.5f ); glVertex3f(  0.5f,  0.5f, -0.5f );
  glEnd();
  // Render the bottom quad
  glBegin(GL_QUADS);
    glTexCoord3f( -0.5f, -0.5f, -0.5f ); glVertex3f( -0.5f, -0.5f, -0.5f );
    glTexCoord3f( -0.5f, -0.5f,  0.5f ); glVertex3f( -0.5f, -0.5f,  0.5f );
    glTexCoord3f(  0.5f, -0.5f,  0.5f ); glVertex3f(  0.5f, -0.5f,  0.5f );
    glTexCoord3f(  0.5f, -0.5f, -0.5f ); glVertex3f(  0.5f, -0.5f, -0.5f );
  glEnd();
  // Render the back quad
  glBegin(GL_QUADS);
    glTexCoord3f(  0.5f,  -0.5f,   -0.5f ); glVertex3f(  0.5f, -0.5f, -0.5f );
    glTexCoord3f( -0.5f,  -0.5f,   -0.5f ); glVertex3f( -0.5f, -0.5f, -0.5f );
    glTexCoord3f( -0.5f,   0.5f,   -0.5f ); glVertex3f( -0.5f,  0.5f, -0.5f );
    glTexCoord3f(  0.5f,   0.5f,   -0.5f ); glVertex3f(  0.5f,  0.5f, -0.5f );
  glEnd();
  // Render the front quad
    glBegin(GL_QUADS);
    glTexCoord3f(  0.5f,  -0.5f,   0.5f ); glVertex3f(  0.5f, -0.5f,  0.5f );
    glTexCoord3f( -0.5f,  -0.5f,   0.5f ); glVertex3f( -0.5f, -0.5f,  0.5f );
    glTexCoord3f( -0.5f,   0.5f,   0.5f ); glVertex3f( -0.5f,  0.5f,  0.5f );
    glTexCoord3f(  0.5f,   0.5f,   0.5f ); glVertex3f(  0.5f,  0.5f,  0.5f );
    glEnd();
  glPopMatrix();
}
```

Figure 15: Cubemap texture mapping in `renderCubemap()` function declaration in `SolarSystemWidget.cpp`

```cpp
void SolarSystemWidget::drawPlanet(PlanetWidget* planet)
{
    // sun's glowing effect
    GLfloat mat_emission[] = {0.8, 0.8, 0.8, 0.0};
    glPushAttrib(GL_LIGHTING_BIT);
    if (planet == sun)
    {
        glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
    }

    PlanetWidget* planet_to_orbit;     // spaceship'a planet to orbit
    if (planetIndex == 0)
        planet_to_orbit = mercury;
    else if (planetIndex == 1)
        planet_to_orbit = venus;
    else if (planetIndex == 2)
        planet_to_orbit = earth;
    else if (planetIndex == 3)
        planet_to_orbit = mars;
    else if (planetIndex == 4)
        planet_to_orbit = jupyter;
    else if (planetIndex == 5)
        planet_to_orbit = saturn;
    else if (planetIndex == 6)
        planet_to_orbit = uranus;
    else if (planetIndex == 7)
        planet_to_orbit = neptune;

    if (planet == earth)    // reder spaceship once per frame, that's why we make this check
    {
        glDisable(GL_TEXTURE_2D);
        glPushMatrix();
        glTranslatef(planet_to_orbit->_pose[0],planet_to_orbit->_pose[1],planet_to_orbit->_pose[2]);
        glPushMatrix();
        glRotatef(spaceshipSpeed*_time, 1.,0.,0.);
        glTranslatef(0., 1.5, 0.);
        glRotatef(1*_time, 0.,0.,1.);
        glScalef(0.2, 0.2, 0.2);
        drawSpaceship(texID, _time_sinewave, left_engine_deg, right_engine_deg);
        glPopMatrix();
        glPopMatrix();
        glEnable(GL_TEXTURE_2D);
    }

    // update planet's orbit according to their gravitational force to the rest of the bodies
    updatePlanetOrbit(planet);
    glTranslatef(planet->_pose[0],planet->_pose[1],planet->_pose[2]);

    // rotation as a function of time
    glRotatef(0.5*_time, 0.f, 1.f, 0.f);

        // create sphere and bind texture to it
    GLUquadric *qobj = gluNewQuadric();
    gluQuadricTexture(qobj, GL_TRUE);
    gluQuadricNormals(qobj, GLU_SMOOTH); // One normal is generated for every vertex of a quadric
    gluQuadricDrawStyle(qobj, GLU_FILL);
    glRotatef(-90, 1.0, 0.0, 0.0); // correct orientation for planet's textures
    glBindTexture( GL_TEXTURE_2D, texID[planet->_texID] );
    gluSphere(qobj, planet->_radius, 200, 200);
    gluDeleteQuadric(qobj);
    glPopAttrib();
}
```

Figure 16: `drawPlanet()` defined in `SolarSystemWidget.cpp`

2.3 Materials and lighting

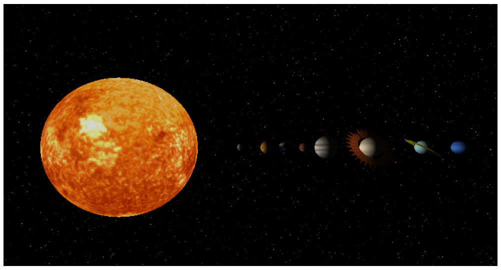

Figure 17: Perspective view of complex scene, diffuse light only

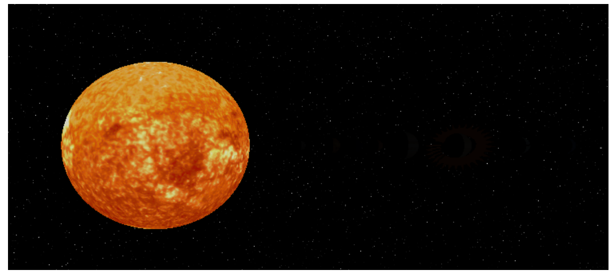

Figure 18: Perspective view of complex scene, ambient light only

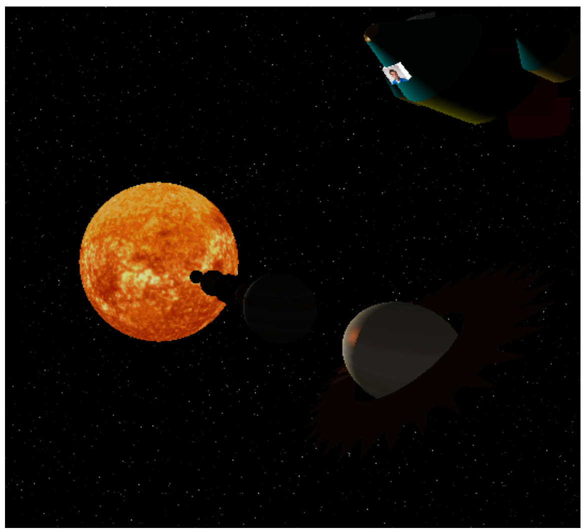

Figure 19: Perspective view of complex scene, diffuse and specular

Only six materials of type `materialStruct` (Figure 3) are defined in Figure 20 that represent the materials used in the complex objects described above.

In terms of lighting conditions,

- the background does not show any sign of lighting, only ambient light applies to it, Figure 21.
- the sun is a glowing sphere, this effect is acquired with the light property GL_EMISSION which is set in `drawPlanet()` (Figure 16).
- The scene contains only `GL_LIGHT0` which is placed in the center of the scene, i.e. within the Sun. `GL_LIGHT0` is assigned `GL_DIFFUSE` and `GL_SPECULAR` OpenGL light properties shown in Figure 21. The `GL_POSITION` of `GL_LIGHT0` creates the sense of day and night on the planet's surfaces. The back faces of the planets are not illuminated, creating a shadow which defines the night.
- Ambient light is set as a global lighting property (Figure 21) calling OpenGL's function `glLightModelfv()` with property `GL_LIGHT_MODEL_AMBIENT`.

The results from enabling `GL_LIGHT0` with the light properties described above can be seen in the Figures from 17 to 19. Different online sources were visited to better understand lighting in OpenGL [7, 8, 9].

```
#ifndef B0B6A216_C18B_40CC_ABB7_B5BB23BDB9E6
#define B0B6A216_C18B_40CC_ABB7_B5BB23BDB9E6

#include "utils/Shapes.h"

static materialStruct brassMaterials = {
    { 0.33, 0.22, 0.03, 1.0},
    { 0.78, 0.57, 0.11, 1.0},
    { 0.99, 0.91, 0.81, 1.0},
    27.8
};

static materialStruct whiteShinyMaterials = {
    { 1.0, 1.0, 1.0, 1.0},
    { 1.0, 1.0, 1.0, 1.0},
    { 1.0, 1.0, 1.0, 1.0},
    100.0
};

static materialStruct yellowPlastic = {
    {0.0f,0.0f,0.0f,1.0f },
    {0.5f,0.5f,0.0f,1.0f },
    {0.60f,0.60f,0.50f,1.0f },
    12.0f};

static materialStruct ruby = {
    { 0.1745f, 0.01175f, 0.01175f, 0.55f },
    {0.61424f, 0.04136f, 0.04136f, 0.55f },
    {0.727811f, 0.626959f, 0.626959f, 0.55f },
    76.8f};

static materialStruct cyanPlastic = {
    {0.0f,0.1f,0.06f ,1.0f},
    {0.0f,0.50980392f,0.50980392f,1.0f},
    {0.50196078f,0.50196078f,0.50196078f,1.0f},
    32.0f};

static materialStruct polishedCopper = {
    {0.2295f, 0.08825f, 0.0275f, 1.0f},
    {0.5508f, 0.2118f, 0.066f, 1.0f},
    {0.580594f, 0.223257f, 0.0695701f, 1.0f},
    51.2f};

#endif /* B0B6A216_C18B_40CC_ABB7_B5BB23BDB9E6 */
```

Figure 20: Definitions in `utils/Materials.h`

```cpp
void SolarSystemWidget::createSolarSystem()
{
  glDisable(GL_TEXTURE_CUBE_MAP);
  // rotate around the scene using the arcball camera rotation - drag mouse
  if (rotate)
  {
    glRotatef(2 * qRadiansToDegrees(angle), objSpaceRotAxis[0], objSpaceRotAxis[1], objSpaceRotAxis[2]);
  }
  // Zoom in and out of the scene
  glScalef(1 - scrollDelta/160, 1 - scrollDelta/160, 1 - scrollDelta/160);

  // diffuse lighting set by the user
  GLfloat dif[] = {sunlightIntensity, sunlightIntensity, sunlightIntensity, 1.};
  GLfloat spec[] = {0.8, 0.8, 0.8, 1.};
  // global ambient light values
  float ambientLevel[] = { 0.15, 0.15, 0.15, 1. };
  glLightModelfv( GL_LIGHT_MODEL_AMBIENT, ambientLevel );

  glEnable(GL_LIGHTING);

  glPushMatrix();
    glScalef(0.1, 0.1, 0.1); // make solar system very small to fit inside cubemap

    glPushMatrix();
      GLfloat centerLight[] = {0., 0., 0.};
      glLightfv(GL_LIGHT0, GL_POSITION, centerLight);
      glLightfv(GL_LIGHT0, GL_DIFFUSE, dif);
      glLightfv(GL_LIGHT0, GL_SPECULAR, spec);
      glEnable(GL_LIGHT0); // set lighting conditions for the scene
      drawPlanet(sun);
    glPopMatrix();

    glPushMatrix();
      drawPlanet(mercury);
    glPopMatrix();

    glPushMatrix();
      drawPlanet(venus);
    glPopMatrix();

    glPushMatrix();
      drawPlanet(earth);
    glPopMatrix();

    glPushMatrix();
      drawPlanet(mars);
    glPopMatrix();

    glPushMatrix();
      drawPlanet(jupyter);
    glPopMatrix();

    glPushMatrix();
      drawPlanet(saturn);
    glPopMatrix();

    glPushMatrix();
      drawPlanet(uranus);
    glPopMatrix();

    glPushMatrix();
      drawPlanet(neptune);
    glPopMatrix();

    glDisable(GL_TEXTURE_2D);
    // saturn's asteroid ring
    glPushAttrib(GL_LIGHTING_BIT);
      glPushMatrix();
        glTranslatef(saturn->_pose[0],saturn->_pose[1],saturn->_pose[2]);
        glRotatef(-90, 1.0, 0.0, 0.0);
        glRotatef(0.5*_time, 0.5f, 1.f, 0.5f);
        glScalef(0.35,0.35,0.1);
        gear(&polishedCopper, 60);
      glPopMatrix();
    glPopAttrib();
```

Figure 21: `createSolarSystem()` function declaration in `SolarSystemWidget.cpp`

3 Band 3 (50%-60%)

Band 3 requires to contain at least one element of user interaction.

3.1 User interaction

Name	Type	Values	Description
Reset Scene	QPushButton	None	Reset the scene to its initial state
Activate standard basis	QPushButton	None	Activates standard basis (used for debugging)
Activate standard basis	QPushButton	None	Deactivate standard basis (used for debugging)
Sun light intensity	QSlider	range(-10,10)	Magnitude of diffuse from Sun (`GL_LIGHT0`)
Simulation speed	QSlider	range(0,100)	Accelerates the orbit speed of each of the planet around the Sun
Earth Mass	QSlider	range(10,30)	Earth Mass slider that affects simulation orbits
Mercury Mass	QSlider	range(10,30)	Mercury Mass slider that affects simulation orbits
Venus Mass	QSlider	range(10,30)	Venus Mass slider that affects simulation orbits
Mars Mass	QSlider	range(10,30)	Mars Mass slider that affects simulation orbits
Jupiter Mass	QSlider	range(10,30)	Jupiter Mass slider that affects simulation orbits
Saturn Mass	QSlider	range(10,30)	Saturn Mass slider that affects simulation orbits
Neptune Mass	QSlider	range(10,30)	Neptune Mass slider that affects simulation orbits
Spaceship orbiting planet	QComboBox	Planets in the system	Select planet for the spaceship to orbit
Spaceship right engine	QSlider	range(10,3600)	Rotates the position of the right engine
Spaceship left engine	QSlider	range(10,3600)	Rotates the position of the right engine
Spaceship speed	QSlider	range(10,100)	Accelerates the spaceship's orbit around Saturn
Scroll wheel zoom	QWheelEvent	None	Zooms in and out of the scene
Arcball camera (mouse drag)	QMouseEvent	None	Mouse interacts with the scene by pressing and releasing to update `MODELVIEW` matrix to a different perspective.

Table 2: Qt user interface, tabulated

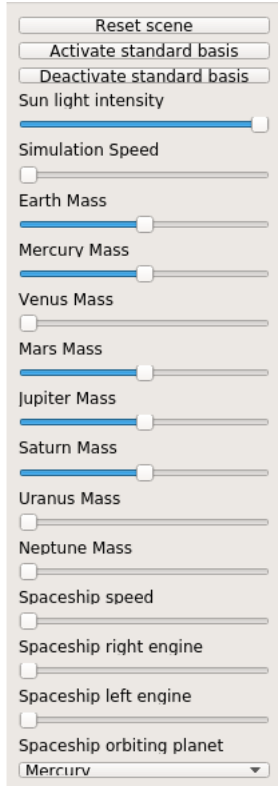

Figure 22: Qt user interface, actual

User interfaces provided through Qt (Table 2 and Figure 22) allow users interact with simulation through a different set of actions such as the light intensity of the Sun in the solar system, the mass of each of the planets conforming the solar system can be adjusted, the simulation speed for all the orbits of the planets, the spaceship speed orbiting around any planet, the planet that the spaceship orbits, right and left engines positions, and there also exists a "Reset scene" button to reset the values of the scene to default.

The second and third buttons are dedicated to activate and deactivate the standard coordinate axes. This was used during the development of the scene to debug position related issues.

Zoom in and out of the scene was implemented to allow the user to change perspective by re-scaling the scene, Figure 23. An "arcball camera" was implemented to translate and rotate around the center of the scene as shown in Figure 24, using `QMouseEvent` pressing and releasing events (Figure 25) to update the `MODELVIEW` matrix of the scene (`updateModelViewMatrix()` and `calculateArcBallVector()` in Figure 26). These two mouse interactive interfaces facilitate the user to have a better 3D experience in the simulation.

To better understand what the "arcball camera" is, there are helpful online sources [4, 5. 6] that I visited.

Signals from `SceneWindow.cpp` of each widget are connected to public slots functions in `SolarSystemWidget.cpp` to update the state of `SolarySystemWidget QGLWidget` object. The setup for a few signal instances and their connections is shown in Figures from 27 to 29.

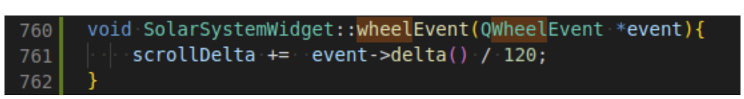

Figure 23: Declaration of `wheelScrollEvent()` in `SolarSystemWidget.cpp`

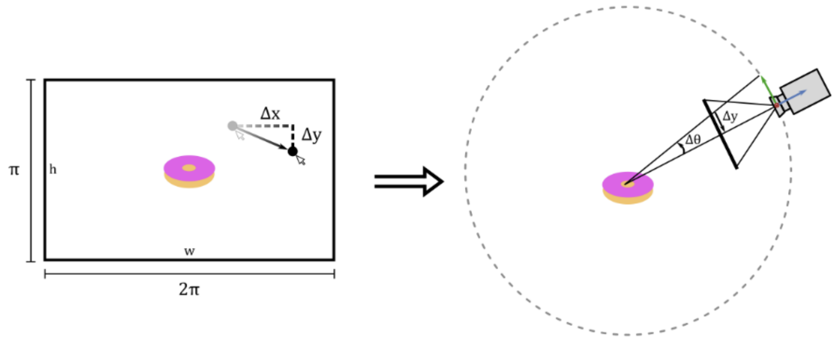

Figure 24: Visual representation of the implemented the "arcball" camera (Image from [6])

```cpp
void SolarSystemWidget::mousePressEvent(QMouseEvent* event)
{
  rotate = 0;
  if(event->button() == Qt::LeftButton)
  {
    // set mouse starting and ending positions
    xStart = event->x();
    yStart = event->y();

    xEnd = event->x();
    yEnd = event->y();

    // set true to use arcball camera
    scribble = 1;
    rotate = 1;
  }
}

void SolarSystemWidget::mouseReleaseEvent(QMouseEvent* event)
{
  // stop moving and do not use arcball camera
  if (event->button() == Qt::LeftButton) {
    scribble = 0;
    rotate = 0;
  }
}
```

Figure 25: `QMouseEvent` events dragging mouse movement simulation

```
void SolarSystemWidget::mouseMoveEvent(QMouseEvent *event){
    if(event->buttons() && Qt::LeftButton)
    {
        if(rotate)
        {
            xEnd = event->x();
            yEnd = event->y();
            updateModelViewMatrix();
        }
        xStart = event->x();
        yStart = event->y();
    }

}

void SolarSystemWidget::updateModelViewMatrix()
{
    QVector3D v = calculateArcBallVector(xStart, yStart); // from the mouse
    QVector3D u = calculateArcBallVector(xEnd, yEnd);

    // angle to rotate
    angle = std::acos(std::min(1.0f, QVector3D::dotProduct(v, u))); // min to avoid acos to give us values > 1

    rotAxis = QVector3D::crossProduct(v,u); // axis that we will rotate about

    float current_matrix[16];
    glGetFloatv(GL_MODELVIEW_MATRIX, current_matrix); // get current MODELVIEW matrix
    mRotate = QMatrix4x4(current_matrix);
    QMatrix4x4 eye2ObjSpaceMat = mRotate;

    // QMatrix4x4 * QVector3D
    // Returns the result of transforming point according to matrix, with the matrix applied pre-point
    objSpaceRotAxis = eye2ObjSpaceMat * rotAxis;

    xStart = xEnd;
    yStart = yEnd;
}

QVector3D SolarSystemWidget::calculateArcBallVector(int x, int y)
{
    // convert x and y screen coordinates to normalised device coordinates (NDC), ignoring z component
    QVector3D pt_ndc = QVector3D(2.0 * x / mWidth - 1.0, 2.0 * y / mHeight  - 1.0 , 0);
    pt_ndc.setY(pt_ndc.y() * -1);

    /*
    Computes z-coordinates for (x',y') NDC by trying to map them to points on the surface of a sphere
    of radius 1, centerted at the origin of our NDC system
    compute z-coordinates by executing pythagoras theorem with two conditions:
            z <= 1.0 or z > 1
    */
    float z = pt_ndc.x() * pt_ndc.x() + pt_ndc.y() * pt_ndc.y();

    if(z <= 1.0)
        pt_ndc.setZ(std::sqrt(1.0 - z));
    else
        pt_ndc.normalize();

    return pt_ndc;
}
```

Figure 26: "arcball" camera functions that update the modelview matrix with respect to mouse dragging interaction

```cpp
void SceneWindow::createHorizontalGroupBox()
{
    horizontalGroupBox = new QGroupBox(tr(""),this);
    horizontalGroupBox->setMinimumSize(112, 612);
    horizontalGroupBox->setMaximumSize(212, 612);
    QVBoxLayout *layout = new QVBoxLayout(this);

    // buttons
    buttonR = new QPushButton(tr("Reset scene"));
    layout->addWidget(buttonR);
    activateBasis = new QPushButton(tr("Activate standard basis"));
    layout->addWidget(activateBasis);
    deactivateBasis = new QPushButton(tr("Deactivate standard basis"));
    layout->addWidget(deactivateBasis);

    // sliders
    Sunlight = new QSlider(Qt::Horizontal);
    Sunlight->setMinimum(-10);
    Sunlight->setMaximum(10);
    Sunlight->setValue(10);

    OrbitSpeed = new QSlider(Qt::Horizontal);
    OrbitSpeed->setMinimum(0);
    OrbitSpeed->setMaximum(100);
    OrbitSpeed->setValue(0);

    EarthSlider = new QSlider(Qt::Horizontal);
    EarthSlider->setMinimum(10);
    EarthSlider->setMaximum(30);
    EarthSlider->setValue(20);
```

Figure 27: creation of a `QGroupBox` declaration in `SceneWindow.cpp` where some interactive widgets are instantiated

```cpp
    // Planet that the spaceship orbits around
    spaceship_panetOrbit = new QComboBox();
    spaceship_panetOrbit->addItem("Mercury");
    spaceship_panetOrbit->addItem("Venus");
    spaceship_panetOrbit->addItem("Earth");
    spaceship_panetOrbit->addItem("Mars");
    spaceship_panetOrbit->addItem("Jupyter");
    spaceship_panetOrbit->addItem("Saturn");
    spaceship_panetOrbit->addItem("Uranus");
    spaceship_panetOrbit->addItem("Neptune");

    layout->addWidget(new QLabel(tr("Sun light intensity")));
    layout->addWidget(Sunlight);

    layout->addWidget(new QLabel(tr("Simulation Speed")));
    layout->addWidget(OrbitSpeed);

    layout->addWidget(new QLabel(tr("Earth Mass")));
    layout->addWidget(EarthSlider);
```

Figure 28: `QComboBox` instantiated and other interactive widgets added to instance of `QGroupBox` layout shown in Figure 27

```cpp
	pTimer = new QTimer;
	    pTimer->start(10);

    connect(pTimer, SIGNAL(timeout()),  solarSystemWidget, SLOT(updateAngle()));

	// orbit speed or simulation speed
	connect(OrbitSpeed, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updateOrbitSpeed(int)));

	// sun light diffusive intensity
	connect(Sunlight, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updateSunLight(int)));

	// reset button
	connect(buttonR, SIGNAL(released()),  solarSystemWidget, SLOT(resetOrbitValues()));
	connect(buttonR, SIGNAL(released()),  this, SLOT(resetAllSliders()));

	// standard basis activation for debuging purposes
	connect(activateBasis, SIGNAL(released()),  solarSystemWidget, SLOT(activateStandardBasis()));
	connect(deactivateBasis, SIGNAL(released()),  solarSystemWidget, SLOT(deactivateStandardBasis()));

	// planets mass sliders
	connect(EarthSlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetEarthMass(int)));
	connect(mercurySlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetMercuryMass(int)));
	connect(venusSlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetVenusMass(int)));
	connect(marsSlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetMarsMass(int)));
	connect(jupyterSlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetJupyterMass(int)));
	connect(saturnSlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetSaturnMass(int)));
	connect(uranusSlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetUranusMass(int)));
	connect(neptuneSlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetNeptuneMass(int)));
	connect(spaceship, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updateSpaceshipSpeed(int)));

	// engine sliders
	connect(right_engine, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updateSpaceship_rightEngine(int)));
	connect(left_engine, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updateSpaceship_leftEngine(int)));

	// spaceship planet to orbit
	connect(spaceship_panetOrbit, SIGNAL(currentIndexChanged(int)), solarSystemWidget, SLOT(setPlanetIndex(int)));
```

Figure 29: Signal's connection to slots in `SceneWindow.cpp`

4 Band 2 (60%-70%)

Band 2 requires to have at least one element of animation, one convex object constructed from polygons and texture mapping.

4.1 Animation

Animation is created through constant updates to the `MODELVIEW` matrices in the scene over time. The widget `QTimer` from `SceneWindow QWidget` updates the scene targeting 100 frames per second. On every second, the orbit of each of the planets, rings around Saturn and Neptune, orbit of spaceship and propulsion of spaceship animation are updated to simulate movement.

In this planetary simulation, two different movement animations were implemented and they are described below.

4.1.1 Orbit

The orbit of each of the planets around the sun was achieved with the gravitational force function (2). The real gravitational constant,

$$G = 6.67408 \times 10^{-11} \tag{1}$$

was not used, and instead it was given a value of 1.0. The formula 2 considers the masses between two planetary bodies and their distance. For each planet in the solar system, including the sun, their gravitational forces are calculated to update their momentum and their position respectively in their rotation around the Sun. Implementation of equation 2 is shown in Figure 30.

Momentum defines the velocity per mass. If we increase the mass of a body and not the momentum, the heavier body will travel less distance because it has less velocity per mass. Momentum is defined for every planet to be

- momentum is static in the x direction,
- constant momentum in the y direction, and
- increasing momentum in the z direction.

The application of momentum to each planet is shown in `updatePlanetOrbit()` in Figure 31.

Notice that the sun has a mass of 1000.0 so that it remains in the center of the system, with constant momentum (0,0,0).

The spaceship can orbit any planet selected by the user, and its orbit is calculated by a call to `glRotatef()` in the x direction by `spaceshipSpeed*_time` (Figure 16, line 464) degrees.

$$\vec{F_g} = G\frac{m_1 m_2}{|\vec{r}|^2}\hat{r} \tag{2}$$

The use of these physics formulas was decided from online sources from where I obtained information regarding how they work and their use [12].

```cpp
double* SolarSystemWidget::gforce(PlanetWidget* p1, PlanetWidget* p2)
{
    // calculate distance between p1 (star) and p2 (planet)
    double r[3] = {p1->_pose[0] - p2->_pose[0],
                   p1->_pose[1] - p2->_pose[1],
                   p1->_pose[2] - p2->_pose[2]};

    double dist_r = sqrt(pow(r[0], 2)+pow(r[1], 2)+pow(r[2], 2));

    // unit vector in the direction from p1 to p2
    double r_hat[3] = {r[0]/dist_r,
                       r[1]/dist_r,
                       r[2]/dist_r};

    double G = 1.0; // gravitational constant. In real world is 6.67e-11

    // apply Newton's law of universal gravitation F=(G*m1*m2)/r^2
    double forceM = (G*p1->_mass*p2->_mass)/(pow(dist_r, 2));
    double* forceV = new double[3];

    // magnitud and direction into one vector
    forceV[0] = -forceM*r_hat[0];
    forceV[1] = -forceM*r_hat[1];
    forceV[2] = -forceM*r_hat[2];

    return forceV;
}
```

Figure 30: Declaration of `gforce()` function equivalent to equation 2 in `SolarSystemWidget.cpp`

```cpp
void SolarSystemWidget::updatePlanetOrbit(PlanetWidget* p1)
{

    float dt = speedInc;    // simulation speed controller
    std::vector<PlanetWidget*> planets{sun, mercury, venus, earth, mars, jupyter, saturn, uranus, neptune};

    // find planet p1 in the planets vector and remove it
    PlanetWidget* p2;
    for (uint i=0; i<planets.size(); i++){
        p2 = planets.at(i);
        if (p1 == p2){
            planets.erase(planets.begin()+i);
        }
    }

    // calculate gravitational force between planet p1 and the rest of the planet in the system
    double** _gforce = new double* [(int)planets.size()];
    _gforce[0] = gforce(p1, planets.at(0));
    _gforce[1] = gforce(p1, planets.at(1));
    _gforce[2] = gforce(p1, planets.at(2));
    _gforce[3] = gforce(p1, planets.at(3));
    _gforce[4] = gforce(p1, planets.at(4));
    _gforce[5] = gforce(p1, planets.at(5));
    _gforce[6] = gforce(p1, planets.at(6));
    _gforce[7] = gforce(p1, planets.at(7));

    // sum the resuls from the gforce calls
    double* gforce_sum = addVectors(_gforce, sizeof(_gforce));

    p1->_forceVector[0] = gforce_sum[0];
    p1->_forceVector[1] = gforce_sum[1];
    p1->_forceVector[2] = gforce_sum[2];

    // calculate momentum
    p1->_momentum[0] = p1->_momentum[0] + p1->_forceVector[0] * dt;
    p1->_momentum[1] = p1->_momentum[1] + p1->_forceVector[1] * dt;
    p1->_momentum[2] = p1->_momentum[2] + p1->_forceVector[2] * dt;
    // calculate posision
    p1->_pose[0] = p1->_pose[0] + p1->_momentum[0] / p1->_mass*dt;
    p1->_pose[1] = p1->_pose[1] + p1->_momentum[1] / p1->_mass*dt;
    p1->_pose[2] = p1->_pose[2] + p1->_momentum[2] / p1->_mass*dt;
}
```

Figure 31: Function that updates the orbit of each planet using gravitational force and momentum, declared in `SolarSystemWidget.cpp`

4.1.2 Oscillation

Sine waves are used to animate the propulsion medium of the spaceship. By using the sine wave equation 3 below, the sine wave used is defined with a wave length value of 2 ($\lambda = 2$) and an amplitude of 0.5. Initially, one wave length is drawn between intervals -1 and 1. The intervals where the wave length is calculated are incremented by 0.1 every frame by the argument `_time_sinewave` passed to `drawSpaceship()` call in function `drawPlanet(), line 468` in Figure 16. Therefore, applying an increasing translation (Figure 32, line 38) on every frame along the positive z axis, results in the animation of the sine wave oscillating. The implementation of the sine wave with the parameters explained above is shown in Figure 32, and it is drawn by OpenGL using `GL_POINTS` with `ruby` material properties.

$$y = A * sin(k * x), k = 2\pi/\lambda \tag{3}$$

where A is the amplitude and k is the wave length.

Visual representations of the orbit and the oscillation animations can be seen from Figures 33 to 35. Although, it is suggested to see it by running the code or watching the videos **??**.

```
30              // fire behind spaceship defined by sine wave function
31              glPushMatrix();
32                glTranslatef(0.,0.,-1.);
33                  float k, x, y;
34                  float wave_length = 2;
35                  float amplitude = 0.5;
36                  float inc = 0.05;
37                  glPointSize(26);
38                  glTranslatef(0.,0,_time_sinewave);
39                  glBegin(GL_POINTS);
40                      for(x=-1+_time_sinewave;x<=1+_time_sinewave;x+=inc)
41                      {
42                        k = 2 * M_PI / wave_length;
43                        y = amplitude * sin(k * x);
44                        glVertex3f(0, y, -x);
45                        glVertex3f(0.2, y, -x);
46                        glVertex3f(-0.2, y, -x);
47                        glVertex3f(0, y+0.2, -x);
48                        glVertex3f(0.2, y+0.2, -x);
49                        glVertex3f(-0.2, y+0.2, -x);
50                      }
51                  glEnd();
52              glPopMatrix();
53            glPopAttrib();
54          glPopMatrix();
```

Figure 32: Implementation of sine wave animation in the rear of the spaceship

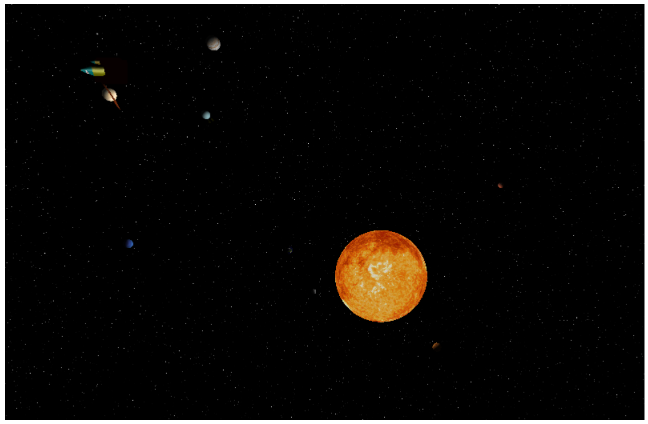

Figure 33: Planets orbiting the Sun

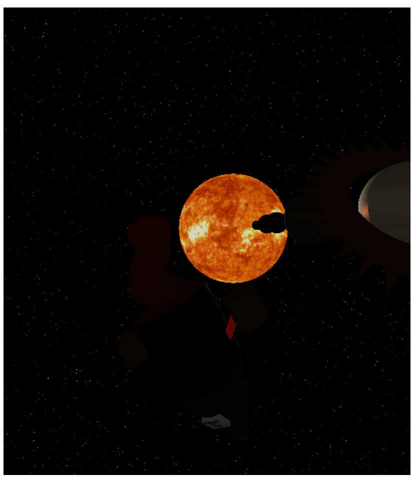

Figure 34: sine wave function simulating movement attached to rocket

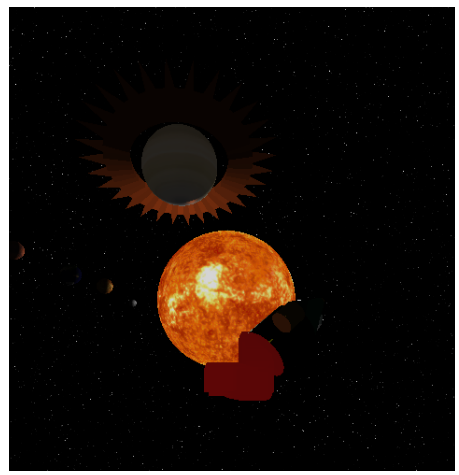

Figure 35: sine wave function simulating movement attached to rocket (different perspective)

4.2 Convex objects

The construction of the spaceship is made up of basic convex shapes, as well as the asteroid rings.

4.3 Texture mapping

As mentioned before, each planet in the scene has its own texture. Texture loading is done by `loadTextures()` function (Figure 37). The open source library used to read the textures is FreeImage [10]. The implementation of texture mapping is done by either calls to `glBindTexture()` or to `glTexCoord3f()` to store texture coordinates for the polygon being constructed. For example, instances of `cube()` (Figure 3) map the entire image (if texture is flagged) to the front face of the polygon with calls to `glTexCoord3f()` with texture values between 0 and 1.

Similarly to how the the textures of the planets are loaded by `loadTextures()`, the textures defining the cubemap are loaded in a similar way by `loadCubemap()` function in `SolarSystemWidget`. Although, in this case binding to `GL_TEXTURE_CUBE_MAP`. To maintain the order in which the texture faces of the cube are loaded, OpenGL sets a texture target parameter to each texture of the cubemap at loading time. `GL_TEXTURE_CUBE_MAP_POSITIVE_X` is the beginning of an OpenGL enum which is composed of:

Texture target	Orientation
GL_TEXTURE_CUBE_MAP_POSITIVE_X	Right
GL_TEXTURE_CUBE_MAP_NEGATIVE_X	Left
GL_TEXTURE_CUBE_MAP_POSITIVE_Y	Top
GL_TEXTURE_CUBE_MAP_NEGATIVE_Y	Bottom
GL_TEXTURE_CUBE_MAP_POSITIVE_Z	Back
GL_TEXTURE_CUBE_MAP_NEGATIVE_Z	Front

Figure 36: Table describing 6 special texture targets for targeting a face of the cubemap. Table from [9]

Since we use an array to store the cubemap textures, we can loop through them linearly incrementing by 1 and calling `glTexImage2D()` with its correspondent texture target. Additionally, the textures names are defined in `cubemapTextures[6]` array (`SolarSystemWidget.h`) in the same order that OpenGL tags each face of the cube map.

The render function of the cubemap takes each loaded texture and maps it to the corresponding face of the cube, this is shown in Figure 15, where calls to `glTexCoord3f()` are performed to do the mapping.

The images `markus.ppm`, `Marc_Dekamps.ppm.ppm` were used as passengers on board of the spaceship seen through the windows. The texture `Mercatorprojection.ppm` was applied to the Earth `gluSphere()` instance.

Before learning how to read textures with FreeImage, I went through some tutorials [7] where they give examples on how to load and apply textures in OpenGL.

```cpp
void SolarSystemWidget::loadTextures()
{
    // Loading Pictures to get width, height and Color Channel Information
    glGenTextures( nTex, texID );  // Get the texture object IDs.
    for (int i=0; i<nTex; i++){
        iData[i] = 0;
        FREE_IMAGE_FORMAT format = FreeImage_GetFIFFromFilename(textures[i]);
        if (format == FIF_UNKNOWN) {
            printf("Unknown file type for texture image file %s\n", textures[i]);
            continue;
        }
        FIBITMAP* bitmap = FreeImage_Load(format, textures[i], 0);

        if (!bitmap) {
            // printf("Cannot load file image %s\nSTB Reason: %s\n", textures[i], stbi_failure_reason());
            printf("Cannot load file image %s. \n", textures[i]);
            continue;
        }
        else{
            printf("Textures %s successfully loaded. \n", textures[i]);
        }

        FIBITMAP* bitmap2 = FreeImage_ConvertTo24Bits(bitmap); // convert to RGB format
        if (i == 0 || i == 1)
        {
          bitmap2 = FreeImage_Rotate(bitmap2, 180);
        }
        FreeImage_Unload(bitmap);
        iData[i] = FreeImage_GetBits(bitmap2);
        width[i] = FreeImage_GetWidth(bitmap2);
        height[i] = FreeImage_GetHeight(bitmap2);
        if (iData[i]) {
            printf("Texture image loaded from file %s, size %dx%d\n",
                                        textures[i], width[i], height[i]);
            int format; // The format of the color data in memory, depends on platform.
            if ( FI_RGBA_RED == 0 )
                format = GL_RGB;
            else
                format = GL_BGR;
            glBindTexture( GL_TEXTURE_2D, texID[i] );  // Will load image data into texture object #i
            glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width[i], height[i], 0, format,
                                        GL_UNSIGNED_BYTE, iData[i]);
            glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); // Required since there are no mipmaps.
            // glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
        }
        else {
            printf("Failed to get texture data from %s\n", textures[i]);
        }
    }
}
```

Figure 37: Function that reads and loads the textures to bind them by OpenGL

5 Band 1 (70%-100%)

Band 1 requires the scene to contain an object that requires hierarchical modelling and displays motion in some of its parts. User interaction with this model is available through the user interface.

5.1 Hierarchical modelling

The space ship required hierarchical modelling since it is made up of the main body (cylinder and cone) and two side rocket engines (Figures 9 and 10). The body and the engines can be drawn as children of the craft instance with their pose dependent on the parent.

Animation of the spaceship shows that it can orbit around any planet at the desired speed by the user, while each of its engines can be set to point to a different position without affecting the orbit of the planet as a result of hierarchical modelling. Figure 38 shows the structure, in particular, of the right engine which calls `glRotatef()` with the argument `float deg` set by the user through the Qt user interface (spaceship right engine QSlider in Figure 2). The hierarchical model of the spaceship is defined in `Spaceship.cpp`, due to its length only code block that defines the right engine is shown below.

```
void right_engine(float deg)
{
  // gluDisk bottom of engine
  glPushMatrix();
  glRotatef(deg, 1.0, 0, 0);
  glPushMatrix();
    // materials of gludisk right engine bottom
    materialStruct* rightEngine = &polishedCopper;
    glMaterialfv(GL_FRONT, GL_AMBIENT,    rightEngine->ambient);
    glMaterialfv(GL_FRONT, GL_DIFFUSE,    rightEngine->diffuse);
    glMaterialfv(GL_FRONT, GL_SPECULAR,   rightEngine->specular);
    glMaterialf(GL_FRONT, GL_SHININESS,   rightEngine->shininess);
    glTranslatef(-1.8, 0., -0.6);
    GLUquadric *qobjR = gluNewQuadric();
    // One normal is generated for every vertex of a quadric. This is the initial value.
    gluQuadricNormals(qobjR, GLU_SMOOTH);
    gluQuadricDrawStyle(qobjR, GLU_FILL);
    gluDisk(qobjR, 0, 0.4, 200, 200);
    gluDeleteQuadric(qobjR);
  glPopMatrix();
  glPushMatrix();
    glPushAttrib(GL_LIGHTING_BIT);
      glTranslatef(-1.8, 0., -0.2);
      glScalef(0.4,0.4,0.4);
      // cylinder of right engine
      cylinder(&brassMaterials);
      glTranslatef(0.,0.,1.0);
      // draw cone at top of the cylinder that forms the engine
      openedCone(&cyanPlastic, 0.01);
    glPopAttrib();
  glPopMatrix();
  glPopMatrix();
}
```

Figure 38: Hierarchical implementation of the right engine of the spaceship in `Spaceship.cpp`

The implementations of `renderCubemap()` and `createSolarSystem()` might also make use or hierarchical modelling, where the former contains the solar system within a cube, and the latter creates a complex scene out of basic shapes.

5.2 User interaction

Users can interact with the spaceship hierarchical model by setting the spaceship right and left engine sliders to rotate in the x direction, and changing the planet of orbit in the drop down menu, outlined in 3. The animation of the propulsion engine in the rear on the craft is explained in 4.1.

6 Conclusion

All in all, I have been able to create my own solar system even though its parameters might not be well tuned for a good simulation. However, I believe that I have met the requirements of the project, which made me learn how to use the huge state machine that OpenGL is. It was interesting to simulate the targeted system using real world physics formulas, texturing polygons, creating new polygons for the scene, applying lighting properties and programming an immersive user interface.

6.1 Source code instructions

To run the application from source, enter the following commands from the project directory in a terminal:

```
cd src/include/
unzip FreeImage3180.zip
cd FreeImage
make -f Makefile.fip
cd ../../
qmake or qmake-qt5
make
./Scene
```

6.2 Video demonstrations

The testing of the scene through feng-linux/gpu result in low frame rate performance. Links to videos running the scene at higher frame rate here:

- https://www.youtube.com/watch?v=0rLoSbXruZY

7 References

1. This report was written using https://github.com/spookycouch/comp3811_cg/blob/master/doc/report.pdf as a template for the sections to cover.

2. https://www.solarsystemscope.com/textures/

3. http://titan.csit.rmit.edu.au/~e20068/teaching/i3dg&a/2016/tute-2.html

4. https://en.wikibooks.org/wiki/OpenGL_Programming/Modern_OpenGL_Tutorial_Arcball

5. http://courses.cms.caltech.edu/cs171/assignments/hw3/hw3-notes/notes-hw3.html

6. https://asliceofrendering.com/camera/2019/11/30/ArcballCamera/

7. https://math.hws.edu/graphicsbook/

8. http://www.glprogramming.com/red/

9. https://learnopengl.com/

10. http://graphics.stanford.edu/courses/cs148-10-summer/docs/FreeImage3131.pdf

11. http://www.it.hiof.no/~borres/j3d/explain/light/p-materials.html

12. https://www.youtube.com/watch?v=4ycpvtIio-o