

COMP3811: Computer Graphics
Coursework 2
Interactive Animated Scenes with OpenGL

Juan Camacho
sid: 201356237
sc19jcm

October 13, 2022

Contents

1	Introduction	2
1.1	Aims	2
1.2	Context	2
2	Band 4 (40%-50%)	3
2.1	Basic objects	3
2.2	Complex scene	7
2.2.1	Planets	8
2.2.2	Spaceship	8
2.2.3	Gear	9
2.2.4	Background	9
2.3	Materials and lighting	13
3	Band 3 (50%-60%)	16
3.1	User interaction	16
4	Band 2 (60%-70%)	22
4.1	Animation	22
4.1.1	Orbit	22
4.1.2	Oscillation	24
4.2	Convex objects	25
4.3	Texture mapping	25
5	Band 1 (70%-100%)	28
5.1	Hierarchical modelling	28
5.2	User interaction	29
6	Conclusion	30
6.1	Source code instructions	30
6.2	Video demonstrations	30
7	References	31

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.

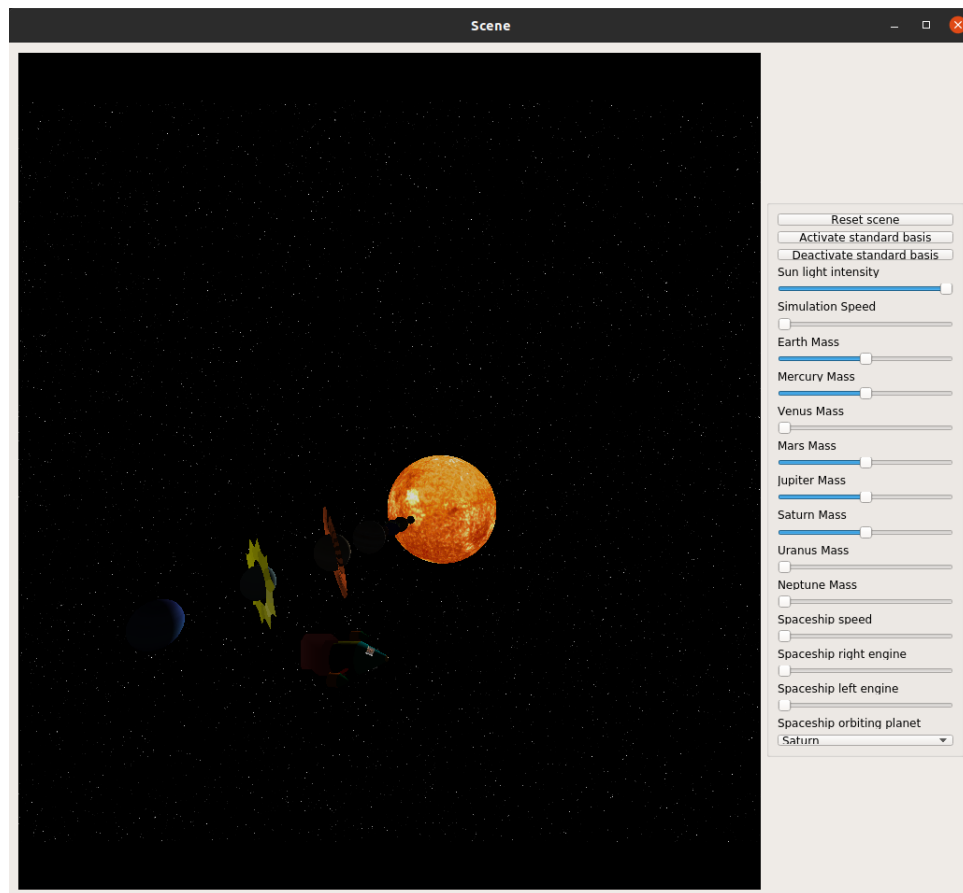


Figure 1: Screenshot of the final product

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:

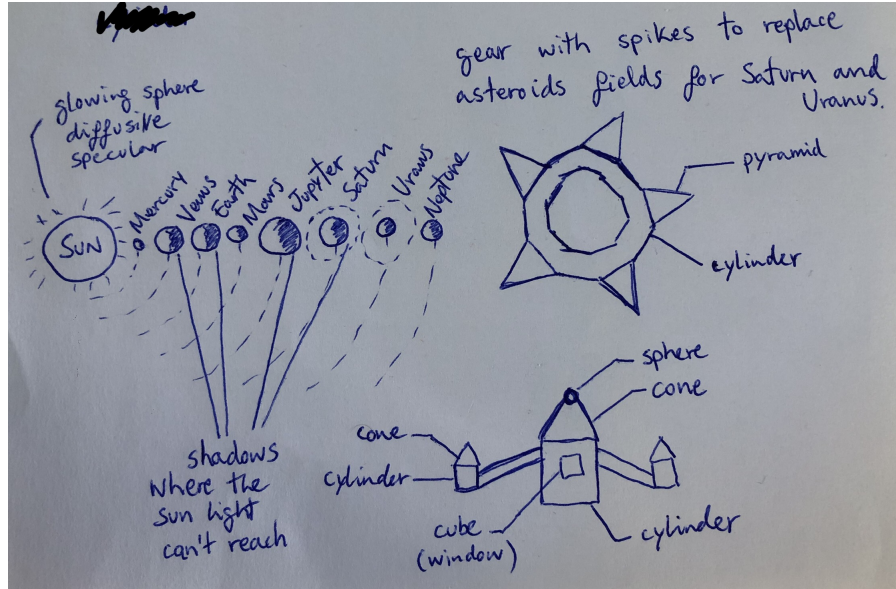


Figure 2: Draft of the complex scene

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.

```

1  #ifndef A25831C8_51A5_403F_B582_7BD4DC518B2A
2  #define A25831C8_51A5_403F_B582_7BD4DC518B2A
3
4  #include <GL/gl.h>
5
6  static const float PI = 3.1415926535;
7
8  /*
9   ..materialStruct
10
11   ..Lighting values of the materials
12
13  */
14  typedef struct materialStruct {
15      GLfloat ambient[4];
16      GLfloat diffuse[4];
17      GLfloat specular[4];
18      GLfloat shininess;
19  } materialStruct;
20
21  /*
22   ..cube
23
24   ..PARAMS
25   ..p_front.....the material properties of the object
26   ..flag.....determines whether to apply markus.ppm or Marc_Dekamps.ppm texture
27   ..texID.....texture ID to bind
28
29  */
30  void cube(const materialStruct* p_front, int flag, GLuint* texID);
31
32  /*
33   ..cylinder
34
35   ..PARAMS
36   ..p_front.....the material properties of the object
37
38  */
39  void cylinder(const materialStruct* p_front);
40
41  /*
42   ..openedCone
43
44   ..PARAMS
45   ..p_front.....the material properties of the object
46   ..r.....radius of one of the sides of the cylinder.
47   ..If radius is large, then it will be a deformed cylinder,
48   ..else it becomes a cone
49
50  */
51  void openedCone(const materialStruct* p_front, float r);
52
53  /*
54   ..gear
55   ..complex shape that uses the cylinder implementation together with a
56   ..pyramid polygon to create a gear-like object.
57
58   ..PARAMS
59   ..p_front.....the material properties of the object
60   ..n_spikes.....determines the number of pyramids (spikes) that the gear will have.
61
62  */
63  void gear(const materialStruct* p_front, int n_spikes);
64
65  #endif /* A25831C8_51A5_403F_B582_7BD4DC518B2A */

```

Figure 3: Definitions in utils/Shapes.h

```

82 void openedCone(const materialStruct* p_front, float r){
83
84     int N = 100; // number of faces
85     int n_div = 1; // number of height divisions
86
87     glMaterialfv(GL_FRONT, GL_AMBIENT, p_front->ambient);
88     glMaterialfv(GL_FRONT, GL_DIFFUSE, p_front->diffuse);
89     glMaterialfv(GL_FRONT, GL_SPECULAR, p_front->specular);
90     glMaterialf(GL_FRONT, GL_SHININESS, p_front->shininess);
91
92     float x0, x1, y0, y1;
93
94     float z_min = -1;
95     float z_max = 1;
96
97     float delta_z = (z_max - z_min)/n_div;
98
99     for (int i = 0; i < N; i++){
100         for(int i_z = 0; i_z < n_div; i_z++){
101             x0 = cos(2*i*PI/N);
102             x1 = cos(2*(i+1)*PI/N);
103             y0 = sin(2*i*PI/N);
104             y1 = sin(2*(i+1)*PI/N);
105
106             float z = 0;
107             glBegin(GL_POLYGON);
108             glVertex3f(x0,y0,z);
109             glNormal3f(x0,y0,0);
110             glVertex3f(x1,y1,z);
111             glNormal3f(x1,y1,0);
112             glVertex3f(r*x1,r*y1,z+delta_z); // apply radius to opposite side of cylinder
113             glNormal3f(x1,y1,0);
114             glVertex3f(r*x0,r*y0,z+delta_z); // apply radius to opposite side of cylinder
115             glNormal3f(x0,y0,0);
116             glEnd();
117         }
118     }
119 }

```

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

```

133 float z_min = 0.;
134 float z_max = 1.; // depth 1.0 max
135 float r = 1.5; // radius
136 float delta_z = (z_max - z_min)/n_div;
137
138 for (int i = 0; i < N; i++){
139     glBegin(GL_POLYGON);
140     for(int i_z = 0; i_z < n_div; i_z++){
141         x0 = cos(2*i*PI/N);
142         x1 = cos(2*(i+1)*PI/N);
143         x2 = r*cos(2*i*PI/N);
144         x3 = r*cos(2*(i+1)*PI/N);
145         y0 = sin(2*i*PI/N);
146         y1 = sin(2*(i+1)*PI/N);
147         y2 = r*sin(2*i*PI/N);
148         y3 = r*sin(2*(i+1)*PI/N);
149
150         // cylinder starts
151         glVertex3f(x0,y0,0); glNormal3f(x0,y0,0);
152         glVertex3f(x2,y2,0); glNormal3f(x2,y2,0);
153         glVertex3f(x2,y2,0.5); glNormal3f(x2,y2,0.5);
154         glVertex3f(x0,y0,0.5); glNormal3f(x0,y0,0.5);
155
156         glVertex3f(x2,y2,0); glNormal3f(x2,y2,0);
157         glVertex3f(x3,y3,0); glNormal3f(x3,y3,0);
158         glVertex3f(x3,y3,0.5); glNormal3f(x3,y3,0.5);
159         glVertex3f(x2,y2,0.5); glNormal3f(x2,y2,0.5);
160
161         glVertex3f(x3,y3,0); glNormal3f(x3,y3,0);
162         glVertex3f(x1,y1,0); glNormal3f(x1,y1,0);
163         glVertex3f(x1,y1,0.5); glNormal3f(x1,y1,0.5);
164         glVertex3f(x3,y3,0.5); glNormal3f(x3,y3,0.5);
165
166         glVertex3f(x1,y1,0); glNormal3f(x1,y1,0);
167         glVertex3f(x0,y0,0); glNormal3f(x0,y0,0);
168         glVertex3f(x1,y1,0.5); glNormal3f(x1,y1,0.5);
169         glVertex3f(x0,y0,0.5); glNormal3f(x0,y0,0.5);
170         // cylinder ends
171
172         // keep building on top of cylinder
173         if (i%2==0)
174         { // pyramid starts
175             x0 = x2; // old x2
176             x1 = x3; // old x3
177             x2 = 2.*cos(2*i*PI/N);
178             x3 = 2.*cos(2*(i+0.5)*PI/N);
179             y0 = y2;
180             y1 = y3;
181             y2 = 2.*sin(2*i*PI/N);
182             y3 = 2.*sin(2*(i+0.5)*PI/N);
183
184             // join the two points in the middle to make a triangle
185             x2 = x3;
186             y2 = y3;
187             float z = 0.25; // where the pyramid's top common vertex meets
188
189             glVertex3f(x0,y0,0); glNormal3f(x0,y0,0);
190             glVertex3f(x2,y2,z); glNormal3f(x2,y2,0);
191             glVertex3f(x2,y2,z); glNormal3f(x2,y2,0.5);
192             glVertex3f(x0,y0,0.5); glNormal3f(x0,y0,0.5);
193
194             glVertex3f(x2,y2,z); glNormal3f(x2,y2,0);
195             glVertex3f(x3,y3,z); glNormal3f(x3,y3,0);
196             glVertex3f(x3,y3,z); glNormal3f(x3,y3,0.5);
197             glVertex3f(x2,y2,z); glNormal3f(x2,y2,0.5);
198
199             glVertex3f(x3,y3,z); glNormal3f(x3,y3,0);
200             glVertex3f(x1,y1,0); glNormal3f(x1,y1,0);
201             glVertex3f(x1,y1,0.5); glNormal3f(x1,y1,0.5);
202             glVertex3f(x3,y3,z); glNormal3f(x3,y3,0.5);
203
204             glVertex3f(x1,y1,0); glNormal3f(x1,y1,0);
205             glVertex3f(x0,y0,0); glNormal3f(x0,y0,0);
206             glVertex3f(x1,y1,0.5); glNormal3f(x1,y1,0.5);
207             glVertex3f(x0,y0,0.5); glNormal3f(x0,y0,0.5);
208         } // pyramid ends
209     }
210     glEnd();
211 }

```

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

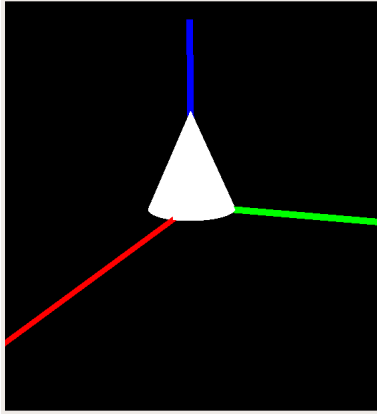


Figure 6: Instance of a cone object
(early implementation)

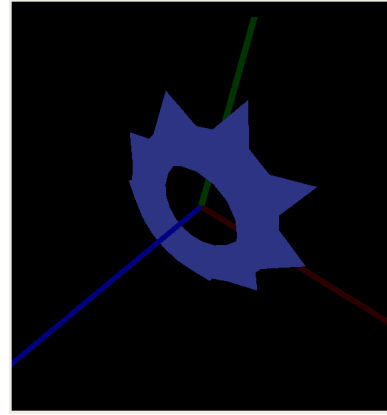


Figure 7: Instance of a gear object
(early implementation)

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.

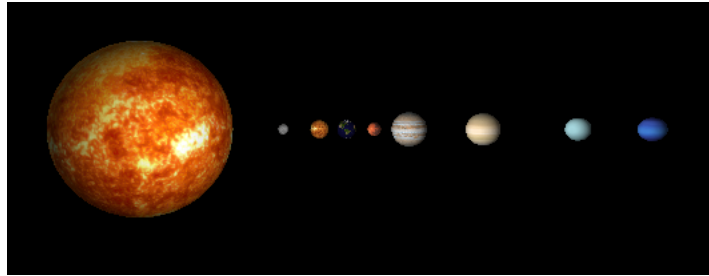


Figure 8: Solar system early stage

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.

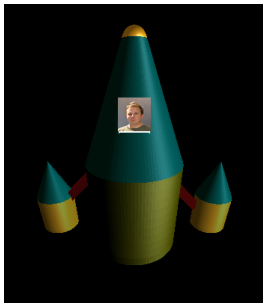


Figure 9: Spaceship front



Figure 10: Spaceship back

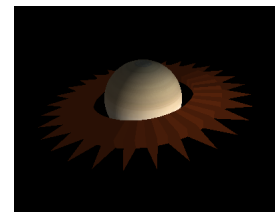


Figure 11: Saturn's ring



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]

```
1  #include <GL/glu.h>
2  #include "PlanetWidget.h"
3
4  // constructor
5  PlanetWidget::PlanetWidget(QWidget *parent, GLuint texID, double mass, double radius,
6  double* pose, double* momentum, int i)
7  : QGLWidget(parent),
8  _texID(texID),
9  _radius(radius),
10 _mass(mass),
11 _index(i)
12 { // constructor
13
14 _pose[0] = pose[0];
15 _pose[1] = pose[1];
16 _pose[2] = pose[2];
17
18 _momentum[0] = momentum[0];
19 _momentum[1] = momentum[1];
20 _momentum[2] = momentum[2];
21
22 } // constructor
```

Figure 13: PlanetWidget class defined in PlanetWidget.cpp

```

1  #ifndef A9E36041_4130_4D53_8670_2278B8F39A6E
2  #define A9E36041_4130_4D53_8670_2278B8F39A6E
3
4  #include "utils/Materials.h"
5  #include "utils/Shapes.h"
6  #include "GL/gl.h"
7  #include "GL/glu.h"
8
9  /*
10  ..drawSpaceship
11
12  ..PARAMS
13  .... GLuint* texID ..... array of texture IDs
14  .... float _time_sinewave ..... input value to the sine wave function determined by _time increments
15  .... float left ..... angle of rotation for the left engine
16  .... float right ..... angle of rotation for the right engine
17  */
18  void drawSpaceship(GLuint* texID, float _time_sinewave, float left, float right);
19
20  /*
21  ..left_engine
22
23  ..PARAMS
24  .... float ..... angle of rotation for the left engine
25  */
26  void left_engine(float);
27
28  /*
29  ..right_engine
30
31  ..PARAMS
32  .... float ..... angle of rotation for the right engine
33  */
34  void right_engine(float);
35
36  #endif /* A9E36041_4130_4D53_8670_2278B8F39A6E */

```

Figure 14: Definitions of spaceship in Spaceship.h

```

500 void SolarSystemWidget::renderCubemap()
501 {
502     glPushMatrix();
503     glScalef(9.,9.,9.); // scale it to make it the background of the scene
504     // Render the right quad
505     glBindTexture(GL_TEXTURE_CUBE_MAP, texIDcube);
506     glBegin(GL_QUADS);
507     glTexCoord3f(-0.5f, -0.5f, -0.5f); glVertex3f(-0.5f, -0.5f, -0.5f);
508     glTexCoord3f(-0.5f, -0.5f, -0.5f); glVertex3f(-0.5f, -0.5f, -0.5f);
509     glTexCoord3f(-0.5f, -0.5f, -0.5f); glVertex3f(-0.5f, -0.5f, -0.5f);
510     glTexCoord3f(-0.5f, -0.5f, -0.5f); glVertex3f(-0.5f, -0.5f, -0.5f);
511     glEnd();
512     // Render the left quad
513     glBegin(GL_QUADS);
514     glTexCoord3f(-0.5f, -0.5f, 0.5f); glVertex3f(-0.5f, -0.5f, 0.5f);
515     glTexCoord3f(-0.5f, -0.5f, 0.5f); glVertex3f(-0.5f, -0.5f, 0.5f);
516     glTexCoord3f(-0.5f, -0.5f, 0.5f); glVertex3f(-0.5f, -0.5f, 0.5f);
517     glTexCoord3f(-0.5f, -0.5f, 0.5f); glVertex3f(-0.5f, -0.5f, 0.5f);
518     glEnd();
519     // Render the top quad
520     glBegin(GL_QUADS);
521     glTexCoord3f(-0.5f, 0.5f, -0.5f); glVertex3f(-0.5f, 0.5f, -0.5f);
522     glTexCoord3f(-0.5f, 0.5f, -0.5f); glVertex3f(-0.5f, 0.5f, -0.5f);
523     glTexCoord3f(-0.5f, 0.5f, -0.5f); glVertex3f(-0.5f, 0.5f, -0.5f);
524     glTexCoord3f(-0.5f, 0.5f, -0.5f); glVertex3f(-0.5f, 0.5f, -0.5f);
525     glEnd();
526     // Render the bottom quad
527     glBegin(GL_QUADS);
528     glTexCoord3f(-0.5f, -0.5f, 0.5f); glVertex3f(-0.5f, -0.5f, 0.5f);
529     glTexCoord3f(-0.5f, -0.5f, 0.5f); glVertex3f(-0.5f, -0.5f, 0.5f);
530     glTexCoord3f(-0.5f, -0.5f, 0.5f); glVertex3f(-0.5f, -0.5f, 0.5f);
531     glTexCoord3f(-0.5f, -0.5f, 0.5f); glVertex3f(-0.5f, -0.5f, 0.5f);
532     glEnd();
533     // Render the back quad
534     glBegin(GL_QUADS);
535     glTexCoord3f(-0.5f, -0.5f, -0.5f); glVertex3f(-0.5f, -0.5f, -0.5f);
536     glTexCoord3f(-0.5f, -0.5f, -0.5f); glVertex3f(-0.5f, -0.5f, -0.5f);
537     glTexCoord3f(-0.5f, -0.5f, -0.5f); glVertex3f(-0.5f, -0.5f, -0.5f);
538     glTexCoord3f(-0.5f, -0.5f, -0.5f); glVertex3f(-0.5f, -0.5f, -0.5f);
539     glEnd();
540     // Render the front quad
541     glBegin(GL_QUADS);
542     glTexCoord3f(-0.5f, -0.5f, 0.5f); glVertex3f(-0.5f, -0.5f, 0.5f);
543     glTexCoord3f(-0.5f, -0.5f, 0.5f); glVertex3f(-0.5f, -0.5f, 0.5f);
544     glTexCoord3f(-0.5f, -0.5f, 0.5f); glVertex3f(-0.5f, -0.5f, 0.5f);
545     glTexCoord3f(-0.5f, -0.5f, 0.5f); glVertex3f(-0.5f, -0.5f, 0.5f);
546     glEnd();
547     glPopMatrix();
548 }

```

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

```

430 void SolarSystemWidget::drawPlanet(PlanetWidget* planet)
431 {
432     // sun's glowing effect
433     GLfloat mat_emission[] = {0.8, 0.8, 0.8, 0.0};
434     glPushAttrib(GL_LIGHTING_BIT);
435     if (planet == sun)
436     {
437         glMaterialfv(GL_FRONT, GL_EMISSION, mat_emission);
438     }
439
440     PlanetWidget* planet_to_orbit; // spaceship's planet to orbit
441     if (planetIndex == 0)
442         planet_to_orbit = mercury;
443     else if (planetIndex == 1)
444         planet_to_orbit = venus;
445     else if (planetIndex == 2)
446         planet_to_orbit = earth;
447     else if (planetIndex == 3)
448         planet_to_orbit = mars;
449     else if (planetIndex == 4)
450         planet_to_orbit = jupyter;
451     else if (planetIndex == 5)
452         planet_to_orbit = saturn;
453     else if (planetIndex == 6)
454         planet_to_orbit = uranus;
455     else if (planetIndex == 7)
456         planet_to_orbit = neptune;
457
458     if (planet == earth) // render spaceship once per frame, that's why we make this check
459     {
460         glDisable(GL_TEXTURE_2D);
461         glPushMatrix();
462         glTranslatef(planet_to_orbit->pose[0], planet_to_orbit->pose[1], planet_to_orbit->pose[2]);
463         glPushMatrix();
464         glRotatef(spaceshipSpeed*_time, 1., 0., 0.);
465         glTranslatef(0., 1.5, 0.);
466         glRotatef(1*_time, 0., 0., 1.);
467         glScalef(0.2, 0.2, 0.2);
468         drawSpaceship(texID, _time_sinewave, left_engine_deg, right_engine_deg);
469         glPopMatrix();
470         glPopMatrix();
471         glEnable(GL_TEXTURE_2D);
472     }
473
474     // update planet's orbit according to their gravitational force to the rest of the bodies
475     updatePlanetOrbit(planet);
476     glTranslatef(planet->pose[0], planet->pose[1], planet->pose[2]);
477
478     // rotation as a function of time
479     glRotatef(0.5*_time, 0.f, 1.f, 0.f);
480
481     // create sphere and bind texture to it
482     GLUquadric *qobj = gluNewQuadric();
483     gluQuadricTexture(qobj, GL_TRUE);
484     gluQuadricNormals(qobj, GLU_SMOOTH); // One normal is generated for every vertex of a quadric
485     gluQuadricDrawStyle(qobj, GLU_FILL);
486     glRotatef(-90, 1.0, 0.0, 0.0); // correct orientation for planet's textures
487     glBindTexture(GL_TEXTURE_2D, texID[planet->_texID]);
488     gluSphere(qobj, planet->_radius, 200, 200);
489     gluDeleteQuadric(qobj);
490     glPopAttrib();
491 }

```

Figure 16: drawPlanet() defined in SolarSystemWidget.cpp

2.3 Materials and lighting

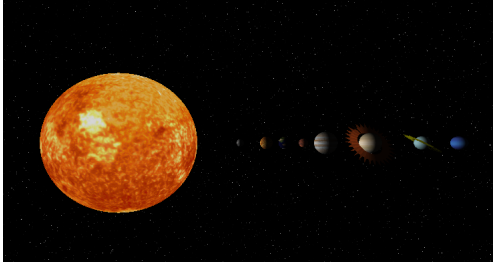


Figure 17: Perspective view of complex scene, diffuse light only



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

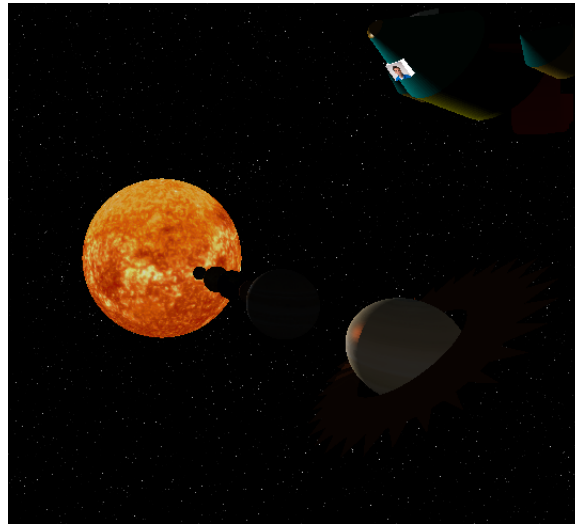


Figure 19: Perspective view of complex scene, diffuse and specular

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].

```

1  #ifndef B0B6A216_C18B_40CC_ABB7_B5BB23BDB9E6
2  #define B0B6A216_C18B_40CC_ABB7_B5BB23BDB9E6
3
4  #include "utils/Shapes.h"
5
6  static materialStruct brassMaterials = {
7      {0.33, 0.22, 0.03, 1.0},
8      {0.78, 0.57, 0.11, 1.0},
9      {0.99, 0.91, 0.81, 1.0},
10     27.8
11 };
12
13 static materialStruct whiteShinyMaterials = {
14     {1.0, 1.0, 1.0, 1.0},
15     {1.0, 1.0, 1.0, 1.0},
16     {1.0, 1.0, 1.0, 1.0},
17     100.0
18 };
19
20 static materialStruct yellowPlastic = {
21     {0.0f, 0.0f, 0.0f, 1.0f},
22     {0.5f, 0.5f, 0.0f, 1.0f},
23     {0.60f, 0.60f, 0.50f, 1.0f},
24     12.0f};
25
26 static materialStruct ruby = {
27     {0.1745f, 0.01175f, 0.01175f, 0.55f},
28     {0.61424f, 0.04136f, 0.04136f, 0.55f},
29     {0.727811f, 0.626959f, 0.626959f, 0.55f},
30     76.8f};
31
32 static materialStruct cyanPlastic = {
33     {0.0f, 0.1f, 0.06f, 1.0f},
34     {0.0f, 0.50980392f, 0.50980392f, 1.0f},
35     {0.50196078f, 0.50196078f, 0.50196078f, 1.0f},
36     32.0f};
37
38 static materialStruct polishedCopper = {
39     {0.2295f, 0.08825f, 0.0275f, 1.0f},
40     {0.5508f, 0.2118f, 0.066f, 1.0f},
41     {0.580594f, 0.223257f, 0.0695701f, 1.0f},
42     51.2f};
43
44 #endif /* B0B6A216_C18B_40CC_ABB7_B5BB23BDB9E6 */

```

Figure 20: Definitions in utils/Materials.h

```

311 void SolarSystemWidget::createSolarSystem()
312 {
313     glDisable(GL_TEXTURE_CUBE_MAP);
314     // rotate around the scene using the arcball camera rotation - drag mouse
315     if (rotate)
316     {
317         glRotatef(2 * qRadiansToDegrees(angle), objSpaceRotAxis[0], objSpaceRotAxis[1], objSpaceRotAxis[2]);
318     }
319     // Zoom in and out of the scene
320     glScalef(1 - scrollDelta/160, 1 - scrollDelta/160, 1 - scrollDelta/160);
321
322     // diffuse lighting set by the user
323     GLfloat dif[] = {sunlightIntensity, sunlightIntensity, sunlightIntensity, 1.};
324     GLfloat spec[] = {0.8, 0.8, 0.8, 1.};
325     // global ambient light values
326     float ambientLevel[] = { 0.15, 0.15, 0.15, 1. };
327     glLightModelfv(GL_LIGHT_MODEL_AMBIENT, ambientLevel);
328
329     glEnable(GL_LIGHTING);
330
331     glPushMatrix();
332     glScalef(0.1, 0.1, 0.1); // make solar system very small to fit inside cubemap
333
334     glPushMatrix();
335     GLfloat centerLight[] = {0., 0., 0.};
336     glLightfv(GL_LIGHT0, GL_POSITION, centerLight);
337     glLightfv(GL_LIGHT0, GL_DIFFUSE, dif);
338     glLightfv(GL_LIGHT0, GL_SPECULAR, spec);
339     glEnable(GL_LIGHT0); // set lighting conditions for the scene
340     drawPlanet(sun);
341     glPopMatrix();
342
343     glPushMatrix();
344     drawPlanet(mercury);
345     glPopMatrix();
346
347     glPushMatrix();
348     drawPlanet(venus);
349     glPopMatrix();
350
351     glPushMatrix();
352     drawPlanet(earth);
353     glPopMatrix();
354
355     glPushMatrix();
356     drawPlanet(mars);
357     glPopMatrix();
358
359     glPushMatrix();
360     drawPlanet(jupyter);
361     glPopMatrix();
362
363     glPushMatrix();
364     drawPlanet(saturn);
365     glPopMatrix();
366
367     glPushMatrix();
368     drawPlanet(uranus);
369     glPopMatrix();
370
371     glPushMatrix();
372     drawPlanet(neptune);
373     glPopMatrix();
374
375     glDisable(GL_TEXTURE_2D);
376     // saturn's asteroid ring
377     glPushAttrib(GL_LIGHTING_BIT);
378     glPushMatrix();
379     glTranslatef(saturn->pose[0], saturn->pose[1], saturn->pose[2]);
380     glRotatef(-90, 1.0, 0.0, 0.0);
381     glRotatef(0.5*_time, 0.5f, 1.f, 0.5f);
382     glScalef(0.35, 0.35, 0.1);
383     gear(&polishedCopper, 60);
384     glPopMatrix();
385     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)
Deactivate standard basis	QPushButton	None	Deactivate standard basis (used for debugging)
Sun light intensity	QSlider	range(-10,10)	Magnitude of diffuse from Sun (<code>GL_LIGHT0</code>)
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 left 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 <code>MODELVIEW</code> matrix to a different perspective.

Table 2: Qt user interface, tabulated

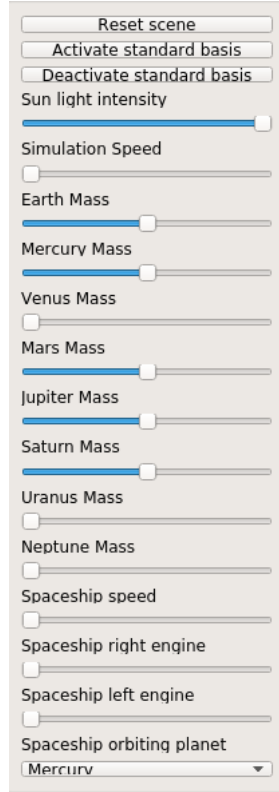


Figure 22: Qt user interface, actual

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 `SolarSystemWidget` `QGLWidget` object. The setup for a few signal instances and their connections is shown in Figures from 27 to 29.

```

760 void SolarSystemWidget::wheelEvent(QWheelEvent *event){
761     scrollDelta += event->delta() / 120;
762 }

```

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

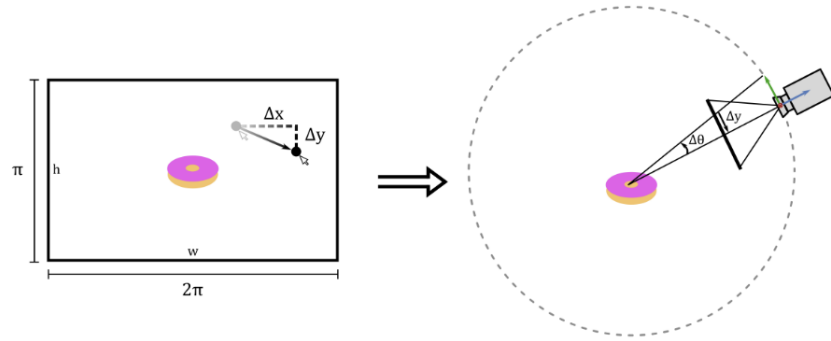


Figure 24: Visual representation of the implemented the "arcball" camera (Image from [6])

```

85 void SolarSystemWidget::mousePressEvent(QMouseEvent* event)
86 {
87     rotate = 0;
88     if(event->button() == Qt::LeftButton)
89     {
90         // set mouse starting and ending positions
91         xStart = event->x();
92         yStart = event->y();
93
94         xEnd = event->x();
95         yEnd = event->y();
96
97         // set true to use arcball camera
98         scribble = 1;
99         rotate = 1;
100     }
101 }
102
103 void SolarSystemWidget::mouseReleaseEvent(QMouseEvent* event)
104 {
105     // stop moving and do not use arcball camera
106     if(event->button() == Qt::LeftButton) {
107         scribble = 0;
108         rotate = 0;
109     }
110 }

```

Figure 25: QMouseEvent events dragging mouse movement simulation

```

109 void SolarSystemWidget::mouseMoveEvent(QMouseEvent *event){
110     if(event->buttons() && Qt::LeftButton)
111     {
112         if(rotate)
113         {
114             xEnd = event->x();
115             yEnd = event->y();
116             updateModelViewMatrix();
117         }
118         xStart = event->x();
119         yStart = event->y();
120     }
121 }
122
123
124 void SolarSystemWidget::updateModelViewMatrix()
125 {
126     QVector3D v = calculateArcBallVector(xStart, yStart); // from the mouse
127     QVector3D u = calculateArcBallVector(xEnd, yEnd);
128
129     // angle to rotate
130     angle = std::acos(std::min(1.0f, QVector3D::dotProduct(v, u))); // min to avoid acos to give us values > 1
131
132     rotAxis = QVector3D::crossProduct(v,u); // axis that we will rotate about
133
134     float current_matrix[16];
135     glGetFloatv(GL_MODELVIEW_MATRIX, current_matrix); // get current MODELVIEW matrix
136     mRotate = QMatrix4x4(current_matrix);
137     QMatrix4x4 eye2objSpaceMat = mRotate;
138
139     // QMatrix4x4 * QVector3D
140     // Returns the result of transforming point according to matrix, with the matrix applied pre-point
141     objSpaceRotAxis = eye2objSpaceMat * rotAxis;
142
143     xStart = xEnd;
144     yStart = yEnd;
145 }
146
147 QVector3D SolarSystemWidget::calculateArcBallVector(int x, int y)
148 {
149     // convert x and y screen coordinates to normalised device coordinates (NDC), ignoring z component
150     QVector3D pt_ndc = QVector3D(2.0 * x / mWidth - 1.0, 2.0 * y / mHeight - 1.0, 0);
151     pt_ndc.setY(pt_ndc.y() * -1);
152
153     /*
154     Computes z-coordinates for (x',y') NDC by trying to map them to points on the surface of a sphere
155     of radius 1, centered at the origin of our NDC system
156     compute z-coordinates by executing pythagoras theorem with two conditions:
157     z <= 1.0 or z > 1
158     */
159     float z = pt_ndc.x() * pt_ndc.x() + pt_ndc.y() * pt_ndc.y();
160
161     if(z <= 1.0)
162         pt_ndc.setZ(std::sqrt(1.0 - z));
163     else
164         pt_ndc.normalize();
165
166     return pt_ndc;
167 }

```

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

```

88 void SceneWindow::createHorizontalGroupBox()
89 {
90     horizontalGroupBox = new QGroupBox(tr(""),this);
91     horizontalGroupBox->setMinimumSize(112, 612);
92     horizontalGroupBox->setMaximumSize(212, 612);
93     QVBoxLayout *layout = new QVBoxLayout(this);
94
95     // buttons
96     buttonR = new QPushButton(tr("Reset scene"));
97     layout->addWidget(buttonR);
98     activateBasis = new QPushButton(tr("Activate standard basis"));
99     layout->addWidget(activateBasis);
100    deactivateBasis = new QPushButton(tr("Deactivate standard basis"));
101    layout->addWidget(deactivateBasis);
102
103    // sliders
104    Sunlight = new QSlider(Qt::Horizontal);
105    Sunlight->setMinimum(-10);
106    Sunlight->setMaximum(10);
107    Sunlight->setValue(10);
108
109    OrbitSpeed = new QSlider(Qt::Horizontal);
110    OrbitSpeed->setMinimum(0);
111    OrbitSpeed->setMaximum(100);
112    OrbitSpeed->setValue(0);
113
114    EarthSlider = new QSlider(Qt::Horizontal);
115    EarthSlider->setMinimum(10);
116    EarthSlider->setMaximum(30);
117    EarthSlider->setValue(20);

```

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

```

169 // Planet that the spaceship orbits around
170 spaceship_paneOrbit = new QComboBox();
171 spaceship_paneOrbit->addItem("Mercury");
172 spaceship_paneOrbit->addItem("Venus");
173 spaceship_paneOrbit->addItem("Earth");
174 spaceship_paneOrbit->addItem("Mars");
175 spaceship_paneOrbit->addItem("Jupyter");
176 spaceship_paneOrbit->addItem("Saturn");
177 spaceship_paneOrbit->addItem("Uranus");
178 spaceship_paneOrbit->addItem("Neptune");
179
180 layout->addWidget(new QLabel(tr("Sun light intensity")));
181 layout->addWidget(Sunlight);
182
183 layout->addWidget(new QLabel(tr("Simulation Speed")));
184 layout->addWidget(OrbitSpeed);
185
186 layout->addWidget(new QLabel(tr("Earth Mass")));
187 layout->addWidget(EarthSlider);

```

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

```

22 pTimer = new QTimer;
23 .....pTimer->start(10);
24
25 .....connect(pTimer, SIGNAL(timeout()), solarSystemWidget, SLOT(updateAngle()));
26
27 // orbit speed or simulation speed
28 connect(OrbitSpeed, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updateOrbitSpeed(int)));
29
30 // sun light diffusive intensity
31 connect(Sunlight, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updateSunLight(int)));
32
33 // reset button
34 connect(buttonR, SIGNAL(released()), solarSystemWidget, SLOT(resetOrbitValues()));
35 connect(buttonR, SIGNAL(released()), this, SLOT(resetAllSliders()));
36
37 // standard basis activation for debugging purposes
38 connect(activateBasis, SIGNAL(released()), solarSystemWidget, SLOT(activateStandardBasis()));
39 connect(deactivateBasis, SIGNAL(released()), solarSystemWidget, SLOT(deactivateStandardBasis()));
40
41 // planets mass sliders
42 connect(EarthSlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetEarthMass(int)));
43 connect(mercurySlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetMercuryMass(int)));
44 connect(venusSlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetVenusMass(int)));
45 connect(marsSlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetMarsMass(int)));
46 connect(jupyterSlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetJupyterMass(int)));
47 connect(saturnSlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetSaturnMass(int)));
48 connect(uranusSlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetUranusMass(int)));
49 connect(neptuneSlider, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updatePlanetNeptuneMass(int)));
50 connect(spaceship, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updateSpaceshipSpeed(int)));
51
52 // engine sliders
53 connect(right_engine, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updateSpaceship_rightEngine(int)));
54 connect(left_engine, SIGNAL(valueChanged(int)), solarSystemWidget, SLOT(updateSpaceship_leftEngine(int)));
55
56 // spaceship planet to orbit
57 connect(spaceship_pametOrbit, 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,

$$G = 6.67408 \times 10^{-11} \quad (1)$$

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.

$$\vec{F}_g = G \frac{m_1 m_2}{|\vec{r}|^2} \hat{r} \quad (2)$$

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

```

402 double* SolarSystemWidget::gforce(PlanetWidget* p1, PlanetWidget* p2)
403 {
404     // calculate distance between p1 (star) and p2 (planet)
405     double r[3] = {p1->pose[0] - p2->pose[0],
406                   p1->pose[1] - p2->pose[1],
407                   p1->pose[2] - p2->pose[2]};
408
409     double dist_r = sqrt(pow(r[0], 2)+pow(r[1], 2)+pow(r[2], 2));
410
411     // unit vector in the direction from p1 to p2
412     double r_hat[3] = {r[0]/dist_r,
413                       r[1]/dist_r,
414                       r[2]/dist_r};
415
416     double G = 1.0; // gravitational constant. In real world is 6.67e-11
417
418     // apply Newton's law of universal gravitation  $F=(G*m1*m2)/r^2$ 
419     double forceM = (G*p1->mass*p2->mass)/(pow(dist_r, 2));
420     double* forceV = new double[3];
421
422     // magnitud and direction into one vector
423     forceV[0] = -forceM*r_hat[0];
424     forceV[1] = -forceM*r_hat[1];
425     forceV[2] = -forceM*r_hat[2];
426
427     return forceV;
428 }

```

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

```

543 void SolarSystemWidget::updatePlanetOrbit(PlanetWidget* p1)
544 {
545     // float dt = speedInc; // simulation speed controller
546     std::vector<PlanetWidget*> planets{sun, mercury, venus, earth, mars, jupyter, saturn, uranus, neptune};
547
548     // find planet p1 in the planets vector and remove it
549     PlanetWidget* p2;
550     for (uint i=0; i<planets.size(); i++){
551         p2 = planets.at(i);
552         if (p1 == p2){
553             planets.erase(planets.begin()+i);
554         }
555     }
556
557     // calculate gravitational force between planet p1 and the rest of the planet in the system
558     double* _gforce = new double* [(int)planets.size()];
559     _gforce[0] = gforce(p1, planets.at(0));
560     _gforce[1] = gforce(p1, planets.at(1));
561     _gforce[2] = gforce(p1, planets.at(2));
562     _gforce[3] = gforce(p1, planets.at(3));
563     _gforce[4] = gforce(p1, planets.at(4));
564     _gforce[5] = gforce(p1, planets.at(5));
565     _gforce[6] = gforce(p1, planets.at(6));
566     _gforce[7] = gforce(p1, planets.at(7));
567
568     // sum the results from the gforce calls
569     double* gforce_sum = addVectors(_gforce, sizeof(_gforce));
570
571     p1->forceVector[0] = gforce_sum[0];
572     p1->forceVector[1] = gforce_sum[1];
573     p1->forceVector[2] = gforce_sum[2];
574
575     // calculate momentum
576     p1->momentum[0] = p1->momentum[0] + p1->forceVector[0] * dt;
577     p1->momentum[1] = p1->momentum[1] + p1->forceVector[1] * dt;
578     p1->momentum[2] = p1->momentum[2] + p1->forceVector[2] * dt;
579
580     // calculate position
581     p1->pose[0] = p1->pose[0] + p1->momentum[0] / p1->mass*dt;
582     p1->pose[1] = p1->pose[1] + p1->momentum[1] / p1->mass*dt;
583     p1->pose[2] = p1->pose[2] + p1->momentum[2] / p1->mass*dt;
584 }

```

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.

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

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

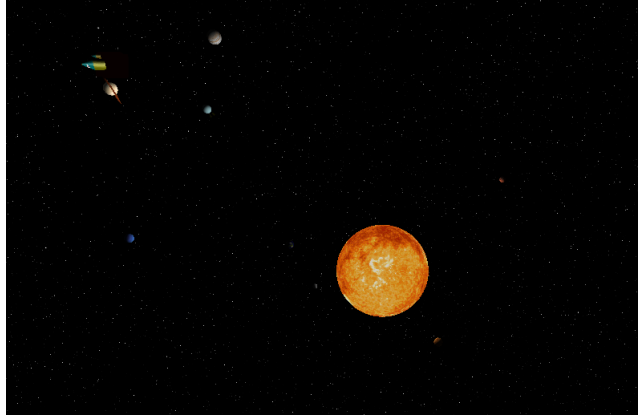


Figure 33: Planets orbiting the Sun

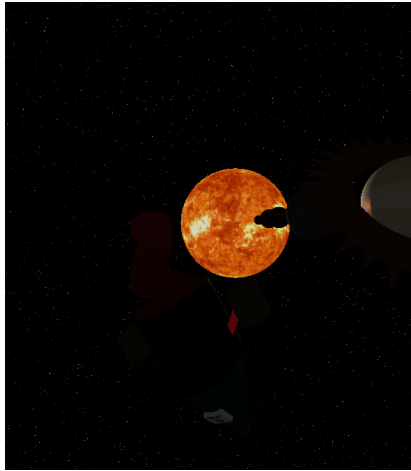


Figure 34: sine wave function simulating movement attached to rocket

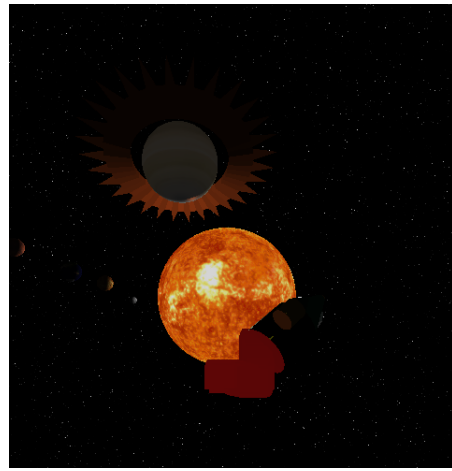


Figure 35: sine wave function simulating movement attached to rocket (different perspective)

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.

```

823 void SolarSystemWidget::loadTextures()
824 {
825     // Loading Pictures to get width, height and Color Channel Information
826     glGenTextures( nTex, texID ); // Get the texture object IDs.
827     for (int i=0; i<nTex; i++)
828     {
829         iData[i] = 0;
830         FREE_IMAGE_FORMAT format = FreeImage_GetFIFFromFilename(textures[i]);
831         if (format == FIF_UNKNOWN) {
832             printf("Unknown file type for texture image file %s\n", textures[i]);
833             continue;
834         }
835         FIBITMAP* bitmap = FreeImage_Load(format, textures[i], 0);
836         if (!bitmap) {
837             // printf("Cannot load file image %s\nSTB Reason: %s\n", textures[i], stbi_failure_reason());
838             printf("Cannot load file image %s. \n", textures[i]);
839             continue;
840         }
841         else {
842             printf("Textures %s successfully loaded. \n", textures[i]);
843         }
844     }
845     FIBITMAP* bitmap2 = FreeImage_ConvertTo24Bits(bitmap); // convert to RGB format
846     if (i == 0 || i == 1)
847     {
848         bitmap2 = FreeImage_Rotate(bitmap2, 180);
849     }
850     FreeImage_Unload(bitmap);
851     iData[i] = FreeImage_GetBits(bitmap2);
852     width[i] = FreeImage_GetWidth(bitmap2);
853     height[i] = FreeImage_GetHeight(bitmap2);
854     if (iData[i]) {
855         printf("Texture image loaded from file %s, size %dx%d\n",
856             textures[i], width[i], height[i]);
857         int format; // The format of the color data in memory, depends on platform.
858         if ( FI_RGBA_RED == 0 )
859             format = GL_RGB;
860         else
861             format = GL_BGR;
862         glBindTexture( GL_TEXTURE_2D, texID[i] ); // Will load image data into texture object #i
863         glTexImage2D(GL_TEXTURE_2D, 0, GL_RGBA, width[i], height[i], 0, format,
864             GL_UNSIGNED_BYTE, iData[i]);
865         glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR); // Required since there are no mipmaps.
866         // glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_MIRRORED_REPEAT);
867     }
868     else {
869         printf("Failed to get texture data from %s\n", textures[i]);
870     }
871 }
872 }

```

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.

```
193 void right_engine(float deg)
194 {
195     // gluDisk bottom of engine
196     glPushMatrix();
197     glRotatef(deg, 1.0, 0, 0);
198     glPushMatrix();
199     // materials of gludisk right engine bottom
200     materialStruct* rightEngine = &polishedCopper;
201     glMaterialfv(GL_FRONT, GL_AMBIENT, rightEngine->ambient);
202     glMaterialfv(GL_FRONT, GL_DIFFUSE, rightEngine->diffuse);
203     glMaterialfv(GL_FRONT, GL_SPECULAR, rightEngine->specular);
204     glMaterialf(GL_FRONT, GL_SHININESS, rightEngine->shininess);
205     glTranslatef(-1.8, 0., -0.6);
206     GLUquadric *qobjR = gluNewQuadric();
207     // One normal is generated for every vertex of a quadric. This is the initial value.
208     gluQuadricNormals(qobjR, GLU_SMOOTH);
209     gluQuadricDrawStyle(qobjR, GLU_FILL);
210     gluDisk(qobjR, 0, 0.4, 200, 200);
211     gluDeleteQuadric(qobjR);
212     glPopMatrix();
213     glPushMatrix();
214     glPushAttrib(GL_LIGHTING_BIT);
215     glTranslatef(-1.8, 0., -0.2);
216     glScalef(0.4, 0.4, 0.4);
217     // cylinder of right engine
218     cylinder(&brassMaterials);
219     glTranslatef(0., 0., 1.0);
220     // draw cone at top of the cylinder that forms the engine
221     openedCone(&cyanPlastic, 0.01);
222     glPopAttrib();
223     glPopMatrix();
224     glPopMatrix();
225 }
```

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

The implementations of `renderCubemap()` and `createSolarSystem()` might also make use of 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>