COMP27112 Laboratory Exercise 2

Modelling with hierarchical transformations

Duration: 2 lab sessions; Marks: 20

This exercise is about modelling. Specifically, it's about modelling using 3D transformations, arranged hierarchically. Your task is to create an orrery – an animated 3D model of the orbital motions of the inner planets and their moons. Important: before you start this exercise, make sure you've read Chapters 8, 9 and 11 of the OpenGL manual, and have studied the lecture notes on viewing, transformations and modelling.

Learning outcomes

1. To understand the use of hierarchical coordinate transformations
2. To understand some concepts of real-time simulation
3. To understand the use of the OpenGL "camera"
4. To implement a demonstrator of the above concepts using OpenGL

1 The exercise

Historically, an orrery is a mechanical apparatus which represents the relative positions and motions of bodies in the solar system by metal balls moved by clockwork. First devised by two clockmakers around 1710, the idea was later copied and made famous by John Rowley for his patron, Charles Boyle, the fourth Earl of Cork and Orrery. Here's a typical modern mechanical orrery, designed and built by Brian Greig (www.orrerymaker.com). Turning the handle causes the planets to orbit around the sun, and the moons to orbit around planets. Some orreries use an electric motor to keep the planets and moons continuously moving.

Your task is to implement a computer graphics orrery simulation using OpenGL. Your orrery will take its planetary data from a data file, which we supply, and will have the following features:

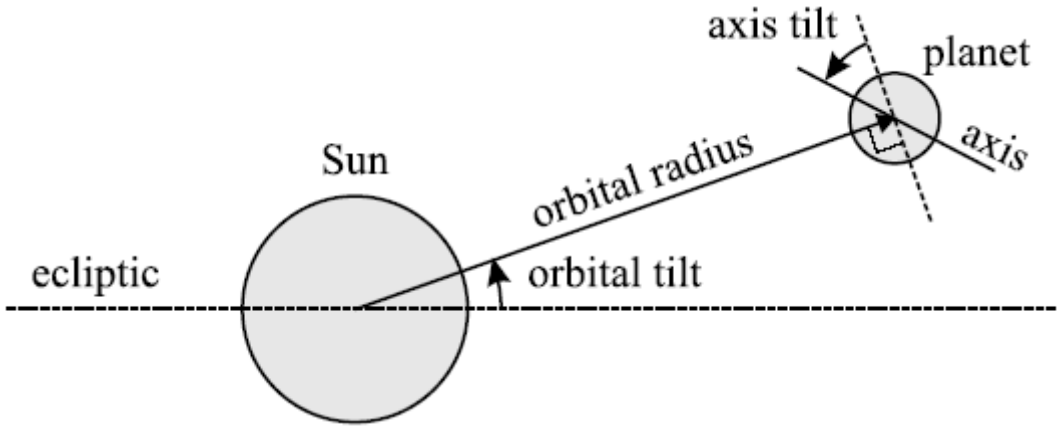
- the Sun is at the centre of the display, with the planets Mercury, Venus, Earth and Mars (and their moons) orbiting around it.
- each planet and moon rotates about its axis, which is also drawn (toggles on/off);
- the paths of the orbit of each planet and moon are drawn (toggles on/off);
- the simulation can be viewed from four different viewpoints: top, ecliptic, spaceship and earth;

There's obviously quite a lot to design and implement here, so we'll break it down into manageable stages. We'll start with an introduction to how planets move.

2 An introduction to planetary motion

Orbits. The planets of the solar system orbit the Sun in an anti-clockwise direction, as seen from a viewpoint above the Sun's (and Earth's) north pole. In the real solar system the orbits are slightly oval-shaped, but here we'll assume all orbits to be circular with a fixed orbital radius. The orbital period – the time taken to complete an orbit – varies for each planet. The Earth takes about 365 days; planets closer to the Sun (Mercury and Venus) take less time, whereas those further out (like Mars) take longer.

The planets of the solar system each orbit the Sun in a slightly different plane. The plane in which the Earth orbits is known as the **ecliptic**. The orbital planes of the other planets are slightly inclined to the ecliptic, and have an associated orbital tilt angle of a few degrees, as shown here:



Rotation. The Sun, the planets and their moons each spin on their individual axes, and each body has its own rotation period. The axis of rotation of each body is tilted with respect to a line perpendicular to its orbital plane (in the Sun's case, with respect to the Earth's orbital plane – the ecliptic). The figure above also shows the axis tilt. Planetary rotations are anti-clockwise, as viewed from above the planet's north pole. Venus's axis of rotation happens to be tilted by almost 180 degrees, which makes it appear to spin in the opposite direction.

The absolute direction of the rotation axis remains constant throughout each orbit. For example, if the Earth's north pole leans towards the Sun at a given point on its orbit, then it will lean away from the Sun half an orbit later. This gives rise to the seasons. One way of maintaining a constant tilt direction is to translate (rather than rotate) the planets on their orbit around the Sun, as orbital rotation will have the unwanted side-effect of modifying the direction of the planet's spin axis.

Issues of scale. Planets have sizes which are much smaller than their orbits – typically one ten-thousandth as big, or even less. To be of any practical use, orreries therefore tend to greatly magnify planetary radii in order to allow the planets to be clearly seen

within an orbital overview. Increasing the radii of bodies by a factor of 1,000 gives reasonable results.

Note, however, that in the data file we provide (see below), the Sun's radius has been artificially reduced by a factor of 50 for diagrammatic purposes. Similarly, the orbital radii of the Moon, Phobos and Deimos have been increased compared with their true values.

Otherwise, for the orbital radii of the planets, we're working with the true values, which are quite large. If you imagine your world coordinates as a sphere centred on the origin, the radius would be about about 300,000,000 km.

Preparation

We're providing a skeleton program to get you started. So first take your own copy:

```
$ cd ~/COMP27112
$ mkdir ex2
$ cd ex2
$ cp /opt/info/courses/COMP27112/Lab/ex2/* .
```

Take a look at **`ex2.c`**, and check it compiles OK with **`cogl`**.

3 The data file

The orrery parameters are specified by a data file, **`sys`**, which contains information about all the bodies (by "body", we mean "planet" or "moon"). **`sys`** begins with an integer which gives the number of bodies described in the file. Then, there is a block of data for each body, with the following format. We'll use the block for Mercury as an example:

```
Mercury      name of body
0.7 0.2 0.7  colour (red, green, blue)
57910000.0   distance to parent body (km)
7.004        angle of orbit relative to earth's orbit (degrees)
87.97        time taken to orbit (days)
2439.0       size of body (km)
2.0          tilt of axis relative to body's orbital plane (degrees)
58.65        body's period of rotation (days)
0            index of parent body
```

For a particular body its "index of parent body" indicates which is its parent body – the body it orbits. The index number for a body is its position in the data file: the Sun is 0, Mercury 1, Venus 2, and so on. For a planet, its parent body will be the Sun (body 0); for a moon, it will be its parent planet. So, for example, Phobos (body 6) will have a parent body index of 4 (for Mars).

Take a look at the **`sys`** file and familiarise yourself with it.

4 The data structure

We use a **`struct`** to hold the details of each body, and we store these in an array. You'll see that the first set of fields in the struct matches exactly a data block from **`sys`**:

```
typedef struct {
/* The following fields are read from "sys" and do not change
   subsequently */
char name[20];                  /* name */
GLfloat r, g, b;                /* colour */
GLfloat orbital_radius;         /* distance to parent body (km) */
GLfloat orbital_tilt;           /* angle of orbit relative
                                   to earth's orbit (degrees) */
GLfloat orbital_period;         /* time taken to orbit (days) */
GLfloat radius;                 /* size of body (km) */
GLfloat axis_tilt;              /* tilt of axis relative to body's
                                   orbital plane (degrees) */
GLfloat rot_period;             /* body's period of rotation (days) */
GLint orbits_body;              /* index of parent body */

/* The following fields change as the simulation runs */
GLfloat spin;                   /* current spin value (degrees) */
GLfloat orbit;                  /* current orbit value (degrees) */
} body;
```

The final two fields in the struct will be used to store the body's current spin (axis rotation) angle and orbit angle, as the orrery runs.

To save you some time, and so you can concentrate on doing interesting graphics and not boring C coding, we've provided the function **`readSystem()`** for you. It reads **`sys`** and stores all the body data in the array of structs.

5 A note about the structure of the simulation

Once the data structures have been initialised, the "engine" of the simulation is the GLUT **`idle`** function, which GLUT automatically calls at the end of each cycle of the GLUT main loop (it animated the mysterious object in Lab 1). We've registered the callback function **`animate()`** as the **`idle`** function, and as you'll see from the skeleton program, each time **`animate()`** is called, it first increments the current date by **`TIME_STEP`** days. It then runs through each of the bodies in the array, and updates its current spin angle, and its current orbit angle.

It then calls **`glutPostDisplay()`** to notify OpenGL that the **`display()`** function needs to be called.

It's then the job of the code in **`display()`** to consult the current values of spin angle and orbit angle for each body, as well as the other information about the body, and to draw it in the right place.

6 A note about using a full-screen display

Because the simulation is quite large, we're going to draw it full-screen. We do this using a GLUT function not mentioned in the OpenGL manual: **`glutFullScreen()`**. We call this function instead of the **`glutInitWindowSize()`** function in Lab Exercise 1.

7 The exercise task by task

We're suggesting that you approach the problem in a number of stages, as described below. But it's really up to you. If you'd prefer to tackle it differently, that's fine. The way the orrery actually looks is also up to you. Other than the colours of the bodies,

which are specified in **sys**, we encourage you to be creative with your use of attributes and visual style. What is important is that your orrery satisfies the functional requirements listed above.

7.1 Task 1: Setting the view

Your first task is to establish a view – the skeleton program doesn't do this. Because we want to have several different views available, the first thing the **display()** function must do is set the view accordingly. We've provided a function, **setView()** which does this, according to the value of the global variable **current_view**, which is set in response to a menu selection, in function **menu()**.

Use **gluLookAt()** to define a "top view", from above the solar system, looking down the y-axis towards the origin. For now, we suggest you draw a sphere at the origin to represent the Sun, just to check your view is OK. The radius of the Sun as given in **sys** is 13,920 km, and remember we're multiplying all body radii by 1,000 to help with the visual scale. So you are going to have to set your eyepoint quite a distance away from the origin. Write a function to draw the world coordinate axes – using different colours for each axis – centred on the origin. This will help you check the planetary motion and the views are correct. Arrange for this axis to be toggled on and off by pressing the 'a' key on the keyboard (see Example 2 in Section 4.6 of the OpenGL manual).

7.2 Task 2: Drawing the starfield

Now you have a view, draw the starfield, which is static – the stars don't move. You can do this by simply creating a large number of points in 3D space. You can draw points in OpenGL using the following:

```
glBegin (GL_POINTS);
glVertex3f (x, y, z);
   /* and all the other points */
glEnd ();
```

By default, a point is drawn as a single pixel, which, if the drawing colour is white, looks a bit like a star. Your task is to complete the function **drawStarfield()**. We've provided you with a function, **myRand()**, which returns a random float in the range [0,1]. Use this random value to plot 1000 points within a large cube which has its centre at the origin. The size of the cube should be at least as large as the inner solar system, say 600 million km along each edge.

7.3 Task 3: Modelling the Sun

Now, model the Sun properly. Although **readSystem()** reads all the bodies' data into the array, just arrange (temporarily) for your **drawBody()** function to draw body 0 – the Sun. Make sure the Sun rotates about its axis, and draw the axis as a line which extends a bit above and below the Sun, so it's clearly visible. Remember the axis needs to be tilted.

7.4 Task 4: Modelling a planet

Now alter your **`drawBody()`** function to draw bodies 0 and 1 – the Sun and Mercury. You will need to apply to following composite transformation T_{planet} to Mercury, where T_{vert} is applied first, then T_{spin}, and so on, as follows:

$$T_{planet} = T_{otilt} \times T_{orb} \times T_{atilt} \times T_{spin} \times T_{vert}$$

where:

- T_{otilt} is the orbital tilt
- T_{orb} is a translation along the orbit
- T_{atilt} is the axis tilt;
- T_{spin} is the spin around the body's axis
- T_{vert} is a rotation by 90^o about the X-axis. (This has nothing whatsoever to do with planetary modelling, however! It's because we're using **`glutWireSphere()`** to represent each body, and by default this function draws a sphere with its poles horizontal instead of vertical.)

7.5 Task 5: Draw the orbital path

Now extend **`drawBody()`** to draw an approximated circle to represent the orbit of each body. You'll need to complete the **`drawOrbit()`** function, to draw a regular polygon with **`ORBIT_POLY_SIDES`** to approximate the orbit circle. Hint: on the back of an envelope, sketch a regular polygon (say, a hexagon) centred on the origin. Draw lines from the origin to each vertex. Use simple trigonometry to work out the coordinates of each vertex.

7.6 Task 6: Modelling a planet with a moon

Now extend **`drawBody()`** so that it also draws the Moon, and Mars's moons. For a body which is a moon, it needs an extra transformation to position it relative to its parent.

7.7 Task 7: Three more views

Now complete the menu options for the ecliptic, spaceship, and Earth views. The ecliptic view is that seen by a camera positioned far out along on the positive Z-axis, looking towards the Sun. To take the spaceship view, choose an eyepoint somewhere out in space looking towards the Sun.

Hint: setting views can involve quite a bit of trial and error to get the effect you're looking for (just like using a real camera). So to avoid endless editing and re-**`cogl`**-ing of your program, you might like to temporarily get the program to prompt you to enter eyepoint coordinates from the command shell (use **`scanf()`**, and disable **`glutFullScreen()`** to read these), so you can experiment with various settings until you get a view you're happy with. Then, you can hard-code that into the program.

Add the Earth view. The camera sits in space just "over the shoulder" of Earth, looking towards the Sun (with a protective filter, of course). The camera moves with the Earth along its orbit, but doesn't spin along with the Earth – such a view would make us dizzy.

7.8 Task 8: Fly around the orrery

Utilise the ideas from Lab 1 to allow you to take a view from a spaceship which you fly through space using the keyboard.

8 Deliverables

You must **`submit`** your completed ex2.c. The marking scheme is:

Task	Marking criteria	marks
1 Setting the top view	Top view working as required	1
	Sun drawn as sphere	1
	3D axis correctly drawn + on/off toggle	2
2 The starfield	Stars drawn as pixels, random pattern	2
3 Modelling the sun	Sun rotating about its axis	2
	Sun's axis drawn at correct angle	1
4 Modelling a planet	Orbital tilt	1
	Path along orbit	1
	Axis tilt	1
5 Draw orbital path of planet	Approximated circular orbit	1
6 Moons around planets	Moon drawn correctly as sphere	1
	Moon correctly orbiting planet	1
7 Three more views	One mark for each of 3 correct views	3
8 Fly around the orrery	Flying around, controlled by keys	2

9 Other ideas (optional, educational, and fun, but not marked)

- Extend **`drawBody()`** to display the name of each body, near the body's north pole. The body's name will inherit the orbital transformations of the body, so it moves in space with the body. Chapter 11 of the OpenGL manual explains how to display text.
- Display the current date (number of days into the simulation) in a suitable position which is unaffected by the motion of the orrery. Hint: You need to temporarily modify the application's view and projection, in order to superimpose on-screen text which is independent of the application's usual view and projection. You might like to refer to the sample code for an on-screen frame-rate counter, from the COMP27112 webpage.
- Implement views for observers standing on each of the bodies.
- Draw a space probe orbiting the Moon.
- On a certain key-press, launch a rocket from earth, which travels to Mars and enters Mars orbit.
- Do some research to track down actual planetary positions for a given date, and use this information to initialise each body's orbit angle. Your orrery will then be a reasonably accurate tool for planet spotting.
- Render the bodies using the local illumination model.

[ends]