Lab 2: An OpenGL Solar System

Goals

Testable goals (may be assessed through the exam):

- Understand the basics of affine transformations (translation, rotation and scaling)
- Understand fundamental OpenGL functionality, including
  - fundamental window buffer manipulation (clearing/swapping)
  - basic graphics primitives
  - hidden surface elimination using the depth test
  - perspective and projection matrices
  - color
  - lighting and materials
- Understand fundamental GLUT functionality, including
  - creating a window
  - specifying callbacks for redisplay, reshape, idle, and keyboard events

Non-testable goals (positive side-effects from completing the assignment):

- Gain experience in understanding other persons’ code
- Gain experience in using C++
- Gain experience in estimation of task completion time
- Gain experience in using a version control system
- Gain experience in using an integrated development environment
- Gain experience in using a debugger

A □ means that this is a step that you must do in order to complete the assignment.

Literature

A brief introduction to OpenGL

OpenGL is a portable, standardized application programming interface (API) that contains functionality for communicating with graphics hardware. Its main purpose is to provide a device-independent communication layer so that applications can create graphics regardless of the type of hardware that is installed. Obviously, the capabilities of graphics devices have evolved substantially over the past three decades (and continue to do so), so the API is being updated regularly to add new function calls that expose such changes. The core parts, however, have remained practically unchanged since the first version of the specification was published. OpenGL is mainly geared towards 3D graphics, but it is possible to use it to create 2D graphics as well.

The specification of OpenGL is maintained and developed by the Khronos Group, a membership-based commercial consortium that develops specifications for, among other things, computer graphics, parallel computing and dynamic media. The Khronos Group has a large number of member companies, including AMD, NVidia, Intel, ARM, Ericsson, Nokia, Apple, Sun Microsystems, Sony, Activision, and Epic Games.

A hardware manufacturer that wishes to support OpenGL must provide a graphics driver: a dynamically linked library (DLL) that implements the specification. Such drivers are normally copied to the system folder when you install the software for your graphics card. In the Windows system folder, there is also a DLL called opengl32.dll. This library is part of Windows and is provided by Microsoft. When opengl32.dll is loaded by an application, it queries the Windows registry to see if a hardware-specific driver is available. If a hardware-specific driver is found, opengl32.dll will load it and forward all OpenGL function calls to it. As a result, OpenGL applications don't have to know the name and manufacturer of the hardware driver – this increases the portability of the application substantially.

Note that hardware manufacturers have to implement the entire specification. In other words, if a driver supports OpenGL 1.3, then all functionality in the OpenGL specification up to and including version 1.3 must be fully supported. Unfortunately, opengl32.dll only “knows” about OpenGL 1.1 and it is not updated by Microsoft as new OpenGL versions come out. So to access functions from version 1.2 and beyond, you either have to obtain pointers to them using a function called wglGetProcAddress (which is implemented in opengl32.dll) or use a tool like OpenGL Extension Wrangler, which can be downloaded from http://glew.sourceforge.net/. In this assignment we will only need functionality from version 1.1, however.

OpenGL is designed to be portable: a given sequence of OpenGL commands produces the same result regardless of which operating system or hardware device they are running on. The drawback of portability is that OpenGL cannot know about such things as windows, menus or scrollbars, since the workings of such entities are operating system dependent. So in order to obtain access to a window canvas in which OpenGL can draw there must be some sort of functionality that acts as an intermediary, and this functionality must be implemented specifically for each operating system. In Windows, the intermediary is a library called WGL. OpenGL uses a data structure called rendering context to exchange information with the operating system. The rendering context is also where the values of OpenGL’s internal state variables are stored.

Although using WGL is quite straightforward, it does require some knowledge of Windows programming which is beyond the scope of this assignment (and it also excludes students who wish to do the
DH2320 Introduction to Visualization and Computer Graphics
Lab 2: An OpenGL Solar System

assignment on another platform). So we will use the Open GL Utility Library (GLUT) instead. GLUT is available for all common operating systems and contains portable functions for, among other things, creating OpenGL contexts and windows, dealing with keyboard and mouse input, and creating simple menus. It also contains a number of functions that draw simple geometrical primitives, such as sphere, cube, cone, and torus. GLUT and its corresponding documentation are available at http://www.opengl.org/resources/libraries/glut/.

An OpenGL solar system

Your task is to create an animated solar system using OpenGL and GLUT. The figures below exemplify how your application could look. The corresponding video can be found on the course homepage.

The application shows the Sun and the eight planets, together with Earth’s moon and four of Jupiter’s moons. In the background there are roughly 9000 stars; their positions and intensity are from an astronomical database. Some of the constellations are highlighted. The camera moves smoothly and shows the most important features of the application (sun/planets/moons, animated orbits, stars, constellations).

Note that your task is not to create an accurate planetarium tool! In the example above, all planetary orbits are circular (which is incorrect) and they are all in the same plane (which is also incorrect). The size of the planets is not at the same scale as their orbit radii, and the radii of the four Jupiter moons have been exaggerated to make them visible. The Sun is not shown to scale at all. However, the relationship between the orbit radii of the planets are correct, as is the relationship between the equatorial radii of the planets. The orbit period is also correct for the planets, although the starting points of the planets are incorrect. The positions of the stars are correct relative to one another but the latitude/longitude of the (imagined) viewer has been completely ignored, and there is no date/time compensation of the right ascension.

Here are the features that your application must include:

DH2320 Introduction to Visualization and Computer Graphics
Lab 2: An OpenGL Solar System

☐ Draw the Sun, the eight solar system planets, the Moon, Io, Europa, Ganymede, and Callisto so that their orbital radii are correct relative to one another. You may approximate the orbits with circles in a single plane.

☐ Draw the orbits of the planets and the moons.

☐ Draw the planets and the moons so that their equatorial radii are correct relative to one another. You may display the Sun at a different scale if you want to.

☐ The planets and moons must be lit by the Sun (but they do not have to cast shadows). They must have individual colors so they can be distinguished from one another.

☐ The planets and moons must be move according to their orbital period so that their movement speed is correct relative to one another (you obviously have to determine an appropriate speedup amount so that the movement is visible). You do not have to compute the correct initial position of the planets and moons.

☐ Draw the stars of the Yale Bright Star Catalogue. The relative positions of the stars have to be correct, but you don’t have to take viewer position and date/time into consideration, nor do you have to worry about the orientation of the ecliptic. Use the formula in the source code to compute an approximation of the relative brightness of the stars, and use that to compute their colors.

☐ Draw/highlight at least three stellar constellations of your choice.

☐ Animate the camera so that all the above features can be seen. The camera must move smoothly.

It is up to you to decide how to deal with the scaling issues when it comes to orbits and equatorial radii.

Instructions

☐ Download the skeleton source code and the GLUT library files from the course webpage.

☐ Create a Subversion repository for your code and add the C++ and header files, and estimate.txt.

☐ Create a Visual Studio project for the source code and compile it.

☐ Make sure that the program runs – it should create a black, empty window.

☐ Examine the skeleton source code carefully (use the debugger!) until you understand what it does. Do not proceed until you understand what the skeleton source code does! Use the OpenGL and GLUT documentation (see the literature list above).

☐ Make sure you understand what the assignment involves (read the rest of the instructions!), and think through the steps you have to carry out. Estimate the time (in hours) you will need to write the source code and enter this number in the file estimate.txt.

☐ In main.cpp, set an appropriate projection matrix in the reshape function. It is recommended that you use gluPerspective for this. (A field-of-view of around 45 degrees is useful in most applications, but feel free to experiment!)

☐ In main.cpp, use gluLookAt and glutSolidCube to position the camera and draw a cube (in the display function). Experiment with different camera positions so you get a feel for how this works.

☐ Commit your code to your Subversion repository.

☐ In planet.cpp, add code to draw planets and the Sun. It is recommended that you use glPushMatrix, glPopMatrix, glTranslate, glRotate, and glScale to position, rotate, and scale the planets and that you use glutSolidSphere to draw them. Use glLight and glMaterial to create/position a light source and set the color of the planets. Don’t forget to enable the depth test! Some basic figures about radii and orbits are available below.

☐ Commit your code.

☐ In planet.cpp, add code to animate the planets. Call the Update method from the idle function in main.cpp.

☐ Commit your code.
In `planet.cpp`, add code to draw the planetary orbits. It is recommended that you use the `GL_LINE_LOOP` primitive for this. Don't forget to disable lighting before you draw the lines! Set the line color with `glColor`.

Commit your code.

In `planet.cpp`, add code to draw the stars. It is recommended that you use the `GL_POINTS` primitive for this. Don’t forget to disable lighting before you draw the points! Set the star color with `glColor`. When drawing the stars in the `display` function in `main.cpp`, think carefully about how to deal with the fact that the stars should not move when the camera is translated. (Hint: Draw the stars first, with the depth test disabled. Use `glPushMatrix`, `glTranslate`, and `glPopMatrix` to center the stars on the current camera position.)

Commit your code.

In `planet.cpp`, add code to draw the constellations. Information about the star catalogue is available below.

Commit your code.

Use the `CSpline` class to create the camera motion. You probably have to experiment a bit until you get the camera sweeps you want. Use the `SmoothStep` function to ease-in/ease-out the camera movement. (Hint: Send a parameter that varies linearly between 0.0 and 1.0 to `SmoothStep` and pass the result to `CSpline::Evaluate`.)

Commit your code.

Enter the time it took to write the code for this assignment in `estimate.txt`.

Commit your code and `estimate.txt`.

Submit your source code (`main.cpp`, `planet.cpp`, and any other file you may have modified) and `estimate.txt` to complete the assignment.

### The solar system

Here is some basic numerical information from Wikipedia about a few of the astronomical bodies that make up the Solar system. Note that the values are approximate.

<table>
<thead>
<tr>
<th>Name</th>
<th>Distance from Sun (Astronomical Units)</th>
<th>Equatorial radius (x Earth)</th>
<th>Orbit period (days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sun</td>
<td>0</td>
<td>109</td>
<td>-</td>
</tr>
<tr>
<td>Mercury</td>
<td>0.4</td>
<td>0.383</td>
<td>88</td>
</tr>
<tr>
<td>Venus</td>
<td>0.7</td>
<td>0.949</td>
<td>225</td>
</tr>
<tr>
<td>Earth</td>
<td>1</td>
<td>1</td>
<td>365</td>
</tr>
<tr>
<td>Mars</td>
<td>1.5</td>
<td>0.533</td>
<td>687</td>
</tr>
<tr>
<td>Jupiter</td>
<td>5.2</td>
<td>11.209</td>
<td>4331</td>
</tr>
<tr>
<td>Saturn</td>
<td>9.5</td>
<td>9.449</td>
<td>10759</td>
</tr>
<tr>
<td>Uranus</td>
<td>19.6</td>
<td>4.007</td>
<td>30799</td>
</tr>
<tr>
<td>Neptune</td>
<td>30</td>
<td>3.883</td>
<td>60190</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Distance from parent (Astronomical Units)</th>
<th>Equatorial radius (x Earth)</th>
<th>Orbit period (days)</th>
</tr>
</thead>
<tbody>
<tr>
<td>The Moon</td>
<td>0.00257</td>
<td>0.374</td>
<td>27.29</td>
</tr>
<tr>
<td>Io</td>
<td>0.00282</td>
<td>0.286</td>
<td>1.77</td>
</tr>
<tr>
<td>Europa</td>
<td>0.00449</td>
<td>0.245</td>
<td>3.55</td>
</tr>
<tr>
<td>Ganymede</td>
<td>0.00716</td>
<td>0.413</td>
<td>7.15</td>
</tr>
<tr>
<td>Callisto</td>
<td>0.0126</td>
<td>0.377</td>
<td>16.69</td>
</tr>
</tbody>
</table>
Stars and constellations

To draw the stars, you will use data from the *Yale Bright Star Catalogue* (YBSC). This catalogue contains the position and luminous magnitude of the 9110 stars that are visible to the naked eye from Earth. The file *star_catalogue.txt* contains some of the YBSC data in a format suitable for the *CStarCatalogue* class. The *ReadFromFile* method reads the star catalogue and converts from polar coordinates to cartesian coordinates.

The YBSC indices of the 88 celestial constellations can be found here: [http://www.alcyone.de/bsc_by_constellation.html](http://www.alcyone.de/bsc_by_constellation.html).

When you add a constellation, remember that *the YBSC starts counting at 1, whereas the CConstellation class begins counting at 0!* Also note that the order of the stars in the YBSC may be different from the order needed to produce the correct line drawing of the constellation. Wikipedia has good illustrations of all the constellations that you can use to figure out a good ordering.

---

5 If you want to look at the raw data, it can be downloaded from here: [http://www-kpno.kpno.noao.edu/Info/Caches/Catalogs/BSC5/catalog5.html](http://www-kpno.kpno.noao.edu/Info/Caches/Catalogs/BSC5/catalog5.html).

A good web search engine into the database can be found here: [http://www.alcyone.de/brightest_stars.html](http://www.alcyone.de/brightest_stars.html).

6 The YBSC uses right ascension and declination to specify star positions. To convert to polar coordinates (degrees), these formulas were used:

\[
\text{declination} \_\text{deg} = \text{degrees} + (\text{arcminutes} / 60) + (\text{arcseconds} / 3600) \\
\text{rightascension} \_\text{deg} = 360 * (\text{degrees} + (\text{arcminutes} / 60) + (\text{arcseconds} / 3600)) / 24
\]

If you want to learn more, these tutorials are good places to start:
[http://astro.unl.edu/classaction/animations/coordsmotion/radecdemo.html](http://astro.unl.edu/classaction/animations/coordsmotion/radecdemo.html)