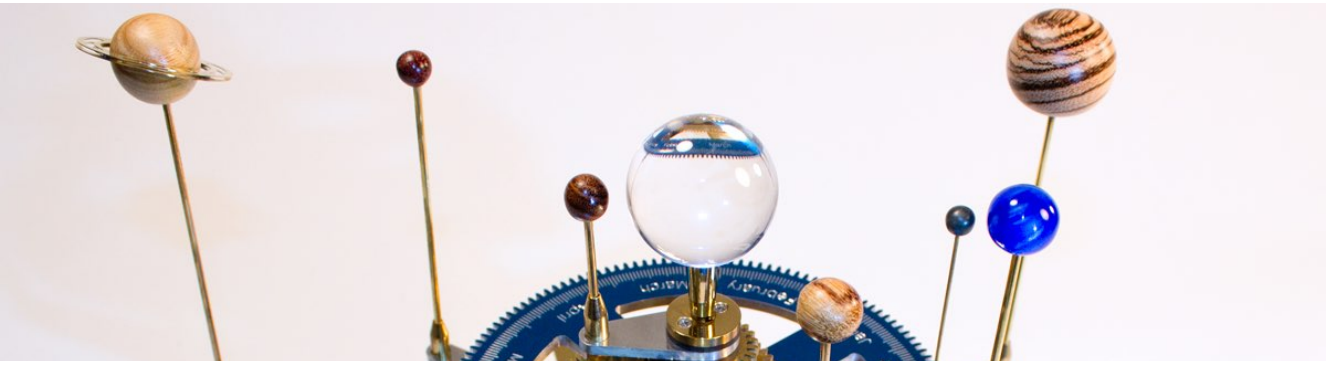
Real-Time Rendering

Assignment #5

CPSC 453 • Fall 2018 • University of Calgary



Overview & Objectives

An orrery is a model of the solar system that mechanically replicates the orbital motions of the planets around the sun. Detailed orreries also replicate the orbits of planets' satellites around their primaries. As with many other wonderful mechanical trinkets, orreries have mostly lost their significance with the advent of computer simulations and graphics.

Your job in this fifth and last assignment of CPSC 453 is to develop a virtual orrery that animates and renders the relative motions of the Sun, Earth, Moon, and Mars against a starry backdrop. This task allows you to assemble essentially everything you've learned in the course into a final showpiece. You will need to write C++ and OpenGL shader code to implement many of the concepts learned in class, including geometry specification, coordinate systems, reference frames, interactive input, hierarchical scene organization, model and view transforms, texture mapping, shading, and real-time animation.

This assignment consists of a written component and a programming component. The programming part is described first, but **note that written answers to the assignment questions are due ahead of the program submission.** You may work on both components with a single partner of your choosing, but remember that you may not work with the same partner you had from Assignments #1 through #4, or for the last assignment in this course.

Due Dates

Written component	**Monday, November 26 at 11:59 PM**
Programming component	**Friday, December 7 at 11:59 PM**

Programming Component

There are a total of five parts to this programming assignment. This component is worth a total of 20 points, with the point distribution as indicated in each part, and an additional possibility of earning a "bonus" designation. Since the bonus is a binary state, it will only be awarded to submissions that do an exemplary job of meeting the requirements.

The parts provide a suggestion for the progression of your implementation and an allocation of points. Specific requirements of your final program are described in each part, but you do not need to save or submit a separate scene for each part of the assignment.

Part I: A Sphere (4 points)

Starting from the initially provided boilerplate code, or any template that you may have created during this course, write a program that will generate triangle geometry to approximate a three-dimensional unit sphere. There may be several good ways to do this. One possibility is to write a C++ function that generates triangles from a parametric or other representation of a sphere and fills a vertex array with their vertex positions. Another is to program the tessellation shaders to generate a sphere "on the fly".

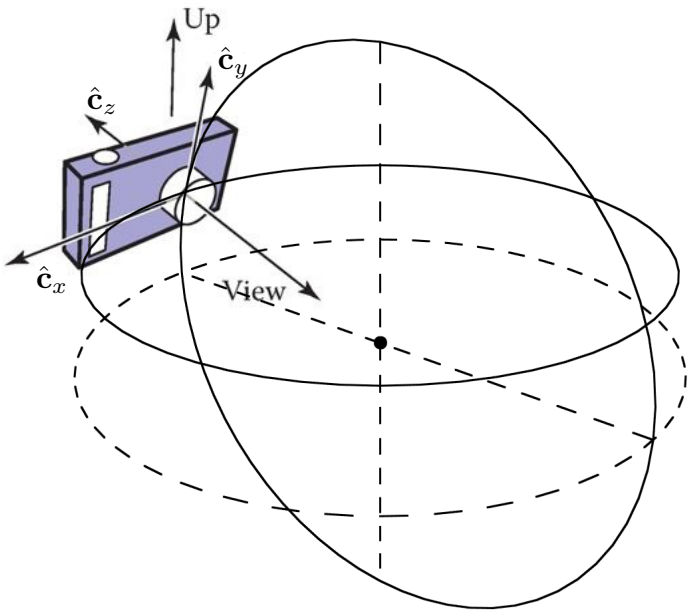
Use OpenGL to draw your sphere to your window. Don't worry about lighting, shading, or projection at this point — those will come later.

Notes & Hints:

- Without shading, your 3D sphere will look like a coloured circle when rendered on the screen. It will be difficult to tell how well your sphere is tessellated, or even if it is correct. You may find the helpful to render your geometry as wireframes, using `glPolygonMode(GL_FRONT_AND_BACK, GL_LINE)`, until you have completed the texturing and shading part of this assignment.

Part II: A Spherical Camera (4 points)

A spherical camera is a virtual camera that is positioned on the surface of a sphere and oriented so that it looks toward a fixed point in space. The camera's configuration can be fully described using spherical coordinates relative to some reference frame, usually the "world" coordinate frame. Given a spherical coordinate triplet, (θ, ϕ, r), respectively denoting altitude, azimuth, and radius, the cartesian (x, y, z) coordinates of the camera frame's origin can be calculated. Furthermore, the camera's "right" axis $\mathbf{c}_x$ is tangential to the equal-altitude line on the virtual sphere, the "up" $\mathbf{c}_y$ axis tangential to the equal-azimuth line, and the "optical" $\mathbf{c}_z$ axis points away from the sphere centre.



Create a spherical camera to view your scene in perspective. Set the camera to look at the centre of your virtual sphere, or world origin. Use mouse input to provide an intuitive means of continuously adjusting the azimuth and altitude, or θ and ϕ parameters, of your camera. For example, clicking and dragging the primary mouse button may move the camera to "orbit" around the centre point. Also provide a means to interactively adjust the camera's distance (radius parameter), perhaps by using the mouse scroll, secondary button, or keyboard input. Enforce reasonable limits for the altitude and distance parameters so that you don't end up extremely close, infinitely far away, or in gimbal lock.

Notes & Hints:

- It would be incredibly educational for you to construct and use your own projection matrix from the theory learned in class. However, you are welcome to use the `perspective()` function provided in GLM if you find it helpful.
- It would likewise be incredibly educational for you to construct and use your own view matrix, but you are welcome to use the `lookAt()` function provided in GLM if desired.
- Don't forget to call `glEnable(GL_DEPTH_TEST)` in your OpenGL initialization code, and to clear the depth buffer before rendering each frame, if you want to use OpenGL's Z-buffer algorithm for hidden surface removal.

Part III: A Scene Graph (4 points)

Set up a scene that contains at least four celestial bodies: the Sun, the Earth, the Moon, and Mars. You may find it convenient to define your "world" reference frame in terms of the Solar System, with an origin located at the centre of the Sun. Employ a data structure to organize your objects explicitly or implicitly so that the Earth's position can be described relative to the Sun, and the Moon's relative to the Earth. A scene graph would be the most natural way to

achieve this hierarchical organization, but any way you choose to store and maintain the transformations between your reference frames is acceptable, as long as you can correctly encode the relative displacements and orientations of the bodies for animation in Part V.

Position and scale your celestial bodies so that the *logarithm* of their real world sizes and distances match the relative scale in your virtual scene. Otherwise, the Earth would be more than 100 times smaller than the Sun, and always very far away, thus making it nearly impossible to see. Choose a base for your logarithm that makes your objects look reasonably sized and spaced apart in your scene (*i.e.* not too empty nor too full). You may need one scale/base for the radii of the bodies and another, different one for their orbital distances to get your final scene to look nice. You may also want to encode the orbital inclination and axial tilt of the bodies as rotations in your scene graph, later needed for Part V.

Notes & Hints:

- You may find it convenient to introduce reference frames in addition to the four fixed to each of the Sun, Earth, Moon, and Mars, so that each frame is only displaced or rotated about one axis from its parent. One example is a reference frame centred at the Earth, but oriented to match the world reference frame. Another is a reference frame for the Moon, but centred at the Earth's position and tilted by the Moon's orbital inclination. If you don't see the immediate utility of these frames, feel free to leave them out for now. Just remember you read this hint here if your animations in Part V are giving you grief.
- Feel free to obtain sizes, distances, and angles of the celestial bodies from any credible source you'd like. Wikipedia often serves as a good, though not always reliable, source: https://en.wikipedia.org/wiki/List_of_gravitationally_rounded_objects_of_the_Solar_System

Part IV: Texturing & Shading (4 points)

First, apply a texture map to each of your celestial bodies by programming the fragment shader. You will need to ensure that fragment shader has access to suitable texture coordinates. The most straightforward way to accomplish this is to modify your code form Part I to generate a (u, v) texture coordinate for each sphere/triangle vertex, then store them in a vertex attribute buffer. Alternatively, you can program the vertex shader to generate texture coordinates on the fly as a function of vertex position. Choose suitable texture images for each body, then load and bind them as OpenGL textures as you did in Assignment #2.

Next, introduce a point light source into your scene, positioned at the centre of the sun. In OpenGL, this usually just takes the form of a uniform vector variable in your shader. This will allow you to calculate a light direction vector for your lighting equation. You will also need surface normals which, like texture coordinates, can be stored and retrieved as vertex attributes or computed in the vertex shader. Program the fragment shader to apply a shading model (*e.g.* Equation 4.3 in Marschner & Shirley) to your objects. The Sun itself naturally

needs no shading: it emits the light that illuminates your other objects. Apply a diffuse reflection model for the Earth, the Moon, and Mars so that the side facing away from the Sun is dark. Add specular and/or ambient components as you desire to make your scene look as nice as you can. Do not shade the Sun with diffuse or specular reflection!

Finally, add an environment texture as backdrop to your scene so that your celestial bodies are not just sitting against a black void. Choose a suitable environment map of a star field. The easiest way to accomplish this is to add a very large sphere to your scene, centred at the camera position, but oriented to align with the world reference frame. Apply the star field as a texture map, but like for the Sun, do not apply diffuse or specular reflection.

Notes & Hints:

- You will likely want to use the `GL_TEXTURE_2D` texture type and `sampler2D` uniform type in your shader, rather than `GL_TEXTURE_RECTANGLE` and `sampler2DRect`. These allow you to index the texture image using (u, v) coordinates in the range [0, 1].
- Many sources of images exist that would make for great textures for your celestial bodies. Feel free to use any that you like as long as they have a fairly liberal licence for use, or you've purchased the rights. Please cite sources and share particularly good ones with your peers. Here are some possible starting points to look:
  - http://www.solarsystemscope.com/nexus/textures/
  - http://planetpixelemporium.com/index.php
  - http://www.shadedrelief.com/natural3/pages/textures.html

Part V: Animation (4 points)

Now that you have all your celestial bodies rendered beautifully with textures and shading, it is time to make them move. Animate the axial rotation of each body and the orbital rotation of the Earth, the Moon, and Mars about their respective primaries. Make both the axial and orbital rotation periods of each body accurate relative to each other. You will probably want to make the animation faster than "real life" speed, or else you'd need to wait a year for the Earth to make its way around the Sun! A rate of one day per second of animation time, or even faster, would be reasonable. Provide at minimum a means for pausing and restarting your animation, and if you'd like, add the ability to adjust the animation speed.

Note that each body has an axial tilt, meaning that its axis of rotation is not perpendicular to its orbital plane. The Moon has an orbital inclination with respect to the Earth's equator, meaning that its orbital plane is tilted from the Earth's axis of rotation. The Earth and Mars also have an orbital inclinations with respect to the Sun's equator, and you may choose whether you prefer to tilt the Sun's rotational axis or the Earth's orbital plane for your orrery. You must capture axial tilts and orbital inclinations in your animation to receive full credit for this part. You may ignore orbital eccentricity and have the bodies follow circular orbits.

Notes & Hints:

- As with Part III in Assignment #3, you will probably need to use `glfwPollEvents()` in your main loop, rather than the call to `glfwWaitEvents()` that was in the original template code, to run a smooth animation of your virtual orrery.
- You may also find the `glfwGetTime()` function helpful for timing your animations.

Bonus Part: A Realistic Earth or a Complete Orrery

There are two possibilities for earning a bonus in this assignment. Each is described in the paragraphs below. If you're so compelled, you may complete both to earn a "mega-bonus"!

The first possibility is to use some of the advanced texture mapping techniques you learned in class, combined with the programmability of the fragment shader, to create an Earth that looks as realistic as possible. Here are some suggestions for what you might do:

- Normal or bump mapping: shade the mountains and valleys to stand out.
- Specular mapping: the oceans may look better if they were shinier than land masses.
- Cloud texture(s): animate clouds that move over the Earth's atmosphere.
- Day/night textures: use a dark texture with city lights for the side away from the Sun.
- Shadows: simulate the effects of solar and lunar eclipses!

You need not implement all of the above, and you are welcome to try other ideas you might have. You will receive this bonus if you can demonstrate a visually compelling model that incorporates at least three advanced techniques. If you complete this bonus, add an option to set your spherical camera to orbit the Earth, rather than the Sun, so that everyone can appreciate the beauty of your artwork.

The second possibility of earning a bonus is to flesh out your virtual orrery to include all the planets of the solar system and as many of their satellites as you can find information about. The goal would be to build your program into an educational or exploration tool, and some valuable capabilities may include:

- An ability to slow down or speed up the animation to appreciate slow or fast objects.
- An ability to centre the camera on different planets to appreciate those perspectives.
- Incorporate orbital eccentricity.
- Displaying textual information such as planet names, vital statistics, etc.

Again, you need not implement all of the above, and you may add any features that you think would enhance the experience of exploring your virtual orrery. Submissions that demonstrate a complete solar system with at least two innovative features to enhance the exploration experience will receive this bonus.

Submission

We encourage you to learn the course material by discussing concepts with your peers or studying other sources of information. However, all work you submit for this assignment must be your own, or explicitly provided to you for this assignment. *Submitting source code you did not author yourself is plagiarism!* If you wish to use other template or support code for this assignment, please obtain permission from the instructors first. Cite any sources of code you used to a large extent for inspiration, but did not copy, in completing this assignment.

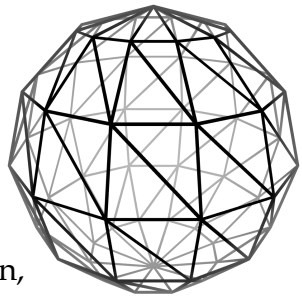
Please upload your source file(s) to the appropriate drop box on the course Desire2Learn site, and indicate any late days used here. Include a "readme" file that briefly explains the mouse and keyboard controls for operating your program, the platform and compiler (OS and version) you built your submission on, and specific instructions for compiling your program if needed. If your program does not compile and run on the graphics lab computers in MS 239, *the onus is on you to ensure that your submission runs on your TA's grading environment for your platform!* Broken submissions will be returned for repair at a minimum penalty of one late day, and may not be accepted if the problem is severe. Ensure that you upload any supporting files (*e.g.* makefiles, shaders, texture images) needed to compile and run your program, but please do not upload compiled binaries or object files.

Your program must also conform to the OpenGL 3.2+ Core Profile, meaning that you should not be using any functions deprecated in the OpenGL API, to receive credit for this part of the assignment. We highly recommend using the official OpenGL 4 reference pages as your definitive guide, located at: https://www.opengl.org/sdk/docs/man/.

Assignment Questions

The written component consists of five questions that correspond to respective parts of the programming assignment, and is worth 10 points in total. When answering the questions, show enough of your work, or provide sufficient explanation, to convince the instructors that you arrived at your answer by means of your own.

Question 1: Sphere (2 points)

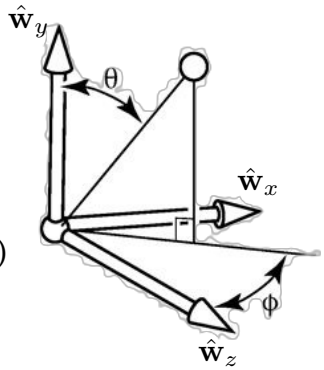


Consider a polygonal approximation to a sphere formed by evaluating each of the azimuth and altitude parameters at 30° intervals, then tessellating each set of four adjacent points with two triangles. The result may look something like the polyhedron shown on the right.

A. How many triangles are present in this polyhedral approximation, in total?

B. What is the ratio between the surface areas of the *smallest* and the *largest* triangle in this polyhedral approximation of a sphere?

Question 2: Spherical Camera (2 points)



A. Consider a camera positioned at spherical coordinates $(\theta, \phi, r) = (60°, 135°, 4)$. What is the camera's position in cartesian (x, y, z) coordinates? Use the convention for altitude (θ) and azimuth (ϕ) angles with respect to the world coordinate frame shown in the diagram on the right (similar to Figure 2.35 in your course text).

B. Assume the camera in 2A were oriented as described in Part II. Write its three camera frame basis vectors, $\mathbf{c}_x$, $\mathbf{c}_y$, and $\mathbf{c}_z$, expressed as measures in the world frame basis.

Question 3: Scene Graph (2 points)

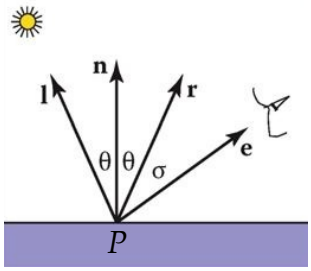
A. Write the 4×4 view transformation matrix for the camera described in Question 2. In other words, if we call the world reference frame W and the camera reference frame C, write the matrix ${}^{C}\mathbf{M}^{W}$ that converts a vector or position expressed in the world coordinates to one expressed in camera coordinates.

B. Consider a third reference frame, B, that is rotated 45° counter-clockwise about the **z**-axis from the world frame, and displaced 1 unit along the world **x**-axis. Write the 4×4 model transformation matrix, ${}^{W}\mathbf{M}^{B}$, for the B frame.

Question 4: Texturing & Shading (2 points)

Suppose you were rendering the scene in Question 3 through the camera in Question 2, and you've added a point light source located at the origin of the world coordinate frame. Imagine you were using the Phong equation to shade a point P on the surface of an object described in B-frame coordinates:

$$\left[\mathbf{r}^{P/B}\right]_B = \begin{bmatrix} 0 \\ \sqrt{2} \\ 0 \end{bmatrix}$$



A. What should you use as the light direction vector, **l**, expressed in camera frame coordinates?

B. What should you use as the view direction vector, **e**, expressed in camera frame coordinates?

Question 5: Animation (2 points)

A. The Moon's orbital period about the Earth is approximately 27.3 days. If you were running an animation at 60 frames per second, such that one second of animation time corresponded to one day of real time, how many degrees would you need to rotate the moon about the Earth at every frame?

B. The Earth's orbital period is approximately 365 days, which means it will make a full rotation around a fixed point at the centre of the Sun in that period of time. The Sun itself has a rotational period of approximately 25.4 days, meaning that it will make a full turn about its own rotational axis in that amount of time. If in your animation you rotated the Earth about a reference frame fixed to the Sun (*i.e.* a frame that rotates along with the Sun) at a rate of one revolution every 365 days (or seconds), what would be the perceived orbital period of the Earth in your program?

You may submit your answers in digital form (typed or scanned) in the appropriate drop box on the course Desire2Learn site, or as hard copy directly to your TA in the tutorial before the assignment is due. Remember that late days may not be used for the written component!