Introduction to Computer Graphics
Assignment 5 – Transformations and Viewing

Handout date: 26.03.2019

Submission deadline: 04.04.2019, 13:00 h

Late submissions are not accepted

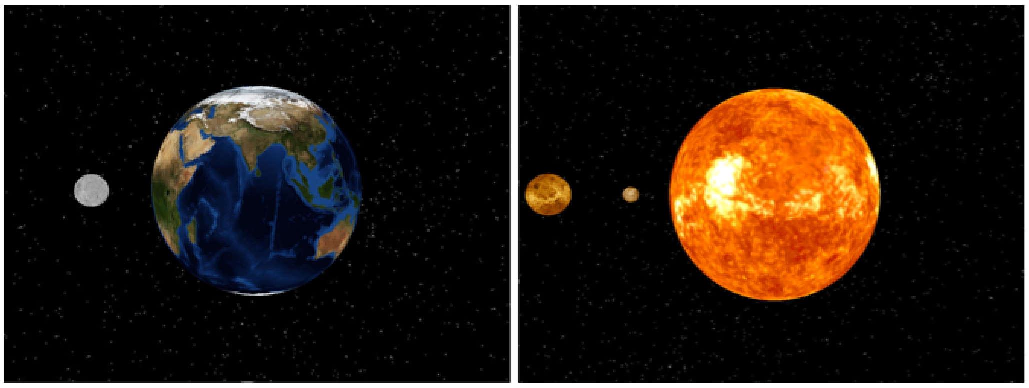


Figure 1: What you should see after launching your finished implementation (left) and the view of the Sun you should see after pressing key 1 (right).

In this assignment, you will build the scene containing the Solar System with the Sun, a couple of planets, a moon and a spaceship. The planets follow circular orbits to revolve around the Sun, and the Earth's moon will revolve around the Earth. Two example views are shown in Figure 1. Furthermore, you will implement the basic rendering pipeline so that each celestial object is displayed with its corresponding texture. Finally, you will place the eye (camera) in the scene, allowing its position and orientation to be adjusted by key presses as shown in Figure 3.

The framework code for this assignment extends the one from last week. "`todo`" comments have been inserted in `solar_viewer.cpp` to indicate where you need to add your implementations. If you already set up a GitHub repository to collaborate with your fellow group members, you can just copy the new `solar_viewer.cpp` over to your repository. However, we did update the code and `cmake` build system this week to improve compatibility with the INF03 lab computers, help catch bugs in your matrix/vector expressions, and speed up compilation (particularly on macOS); you're probably better off replacing all of the assignment 4 files with the new ones in `assignment_5.zip`. The reduction in compilation time is achieved by using the system's copy of the GLFW and GLEW libraries (if they exist) instead of building the included versions; if you want this speed-up, you'll need to install GLEW and GLFW, e.g. using `apt` on Ubuntu or `Macports/Homebrew` on macOS.

Overview

This exercise consists of 4 parts:

1. placing the objects in the scene
2. placing the eye in the scene (setting up the view matrix)
3. rendering the textured objects (building model, modelview, modelview-projection matrices)
4. responding to keyboard events controlling the eye

Unless you finish the first 3 of them, you won't be able to see anything but the Sun as in the last exercise. Therefore, we suggest you start by selecting one of the objects (say the Earth) and implementing the first three parts specifically for this object so that you get visual feedback that your code works as intended. Then you can extend the code to cover all the remaining objects. The fourth part is independent, but we suggest implementing it as soon as possible, since it will allow you to better inspect the scene by adjusting the eye's orientation as shown in Figure 3.

Placing the objects in the scene

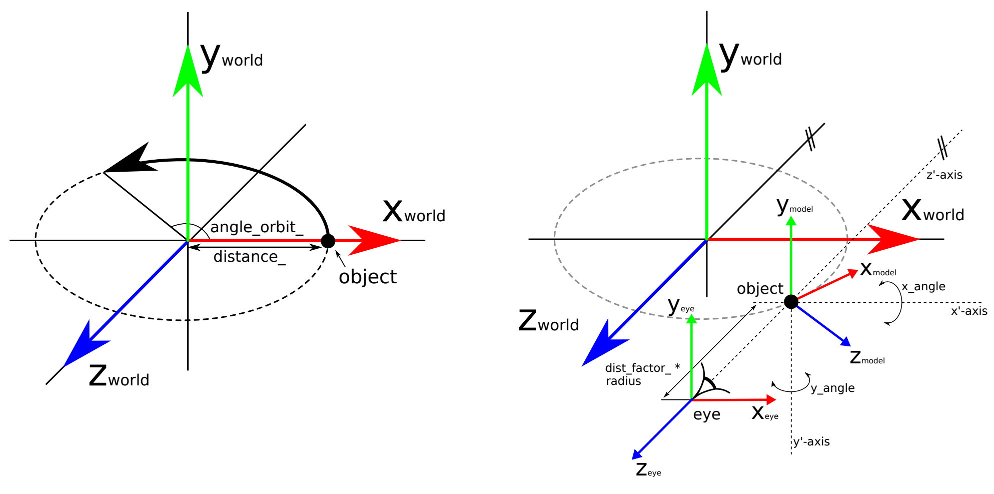
Fill the missing code in function `update_body_positions()`. The goal is to update the position (`pos_`) of each planet to place it on its circular orbit around the origin $(0,0,0)$ of the *world coordinate frame* (where the Sun is placed). The planets' positions along this orbit are given by the current value of angle `angle_orbit_`, which is gradually increased as the simulation time passes. See the Figure 2 (a) depicting the *world coordinate frame* and how `angle_orbit_` relates to it. Then, place the Earth's moon, which should orbit around the Earth's position (not the origin).

Placing the eye in the scene

Fill in the missing code in the function `paint()`. In this part, you will build the view matrix, which effectively positions and orients the eye in the scene. You should make the eye point at the selected object (a planet, moon, or the spaceship). By default, the eye points in the negative z-axis (see Figure 2 (b)). Given the object to be observed (selected by keys 1-7), first place the eye at (positive) distance `dist_factor_ * radius` on the z'-axis, which passes through the object's center and runs parallel to the z-axis of the *world coordinate frame*. (Note that `dist_factor_` can be increased/decreased by keys 8/9.)

Next, allow the view to orbit around the object by rotating the eye position around the x'-axis and y'-axis, which both originate at the object's center and which are parallel to the x-axis and y-axis of the *world coordinate frame* (see Figure 2). The rotations are given by variables `y_angle` and `x_angle`, which are adjusted by the arrow keys. You will first apply the rotation around the x'-axis, then around the y'-axis.

Please make use of the convenience function `mat4::look_at` which, given the eye's position, the center point to look at and the vector pointing upwards with respect to the



(a) (b)

Figure 2: (a) A celestial object placed within the *world coordinate system*. Its circular orbit is given by `distance_` and its current position on the orbit is given by `angle_orbit_`. (b) The eye position with respect to the *model coordinate frame*. It is placed on the positive z-axis at distance given by `dist_factor_` and `radius`. It is then rotated around x-axis and/or y-axis by the angles `x_angle` and `y_angle`.

eye's optical axis (y_{eye} in Figure 2), computes the appropriate view matrix. See Figure 3 depicting the effect of the arrow keys and keys 8/9 on the orientation of the eye.

If you are observing the ship (boolean `in_ship_`), the rotation of the eye around the x-axis should be fixed (i.e., ignore the current value of `x_angle`) such that the view resembles Figure 4, i.e. the eye overlooks the ship slightly from above. Note that it still must be possible to rotate the eye around y-axis using left/right arrow keys. Also, make use of the variable `Ship::angle_` to ensure the view remains at a fixed orientation with respect to the ship as the ship turns.

Rendering textured objects

Fill in the missing code in the function `draw_scene()`. Here you will implement the actual rendering of the objects. At this point, we do not use any lighting, and each object is shaded simply with its applied texture. Texturing will be covered in more depth later on; for this assignment, you only need to issue a call binding the appropriate texture prior to drawing the object, and everything else is handled for you. To render each object, you will have to construct the proper model, modelview and modelview-projection matrices, select a shader, and set certain uniform variables. See the `todo` comments in the `draw_scene()` function explaining how to proceed. Do not forget to also draw the background `stars_` as well—this is yet another (huge) sphere surrounding the Solar System bodies to give the impression of distant starry background.

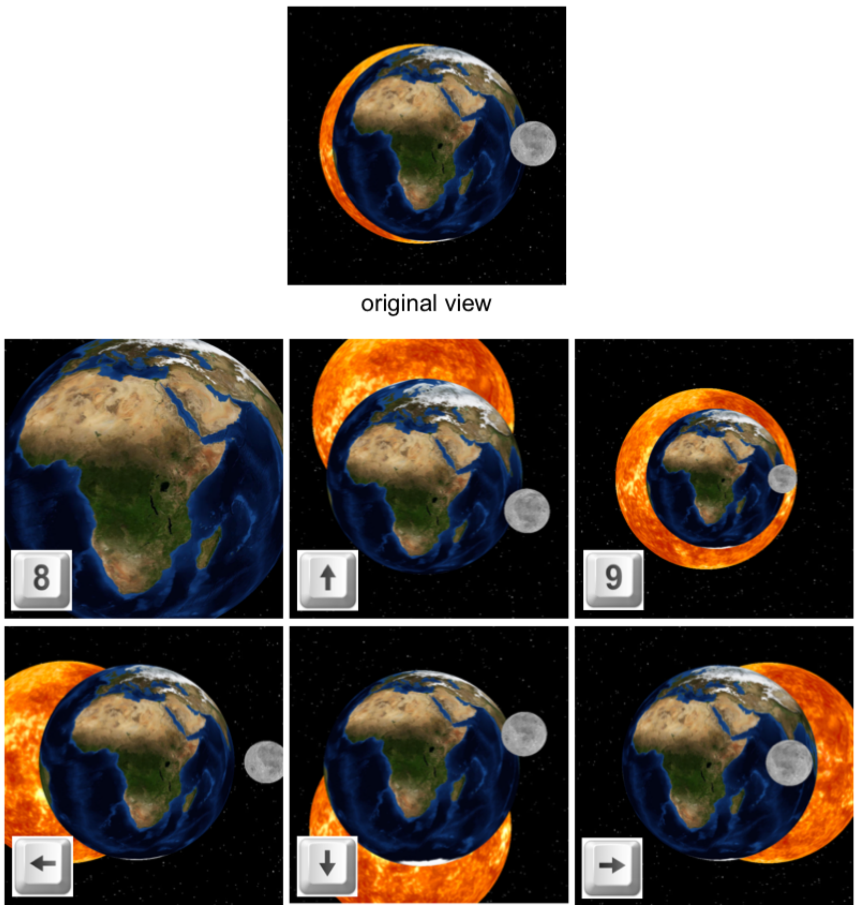


Figure 3: The effect of navigating the eye using keyboard arrows and numbers. The left/right arrows rotate the eye around y-axis, the up/down arrows rotate the eye around x-axis and the numbers 8/9 translate the eye along the z-axis. Note that in this particular figure we already oriented the eye such that it would observe both the Earth and the Sun (i.e. this is not what you will first see once you run your finished implementation.)

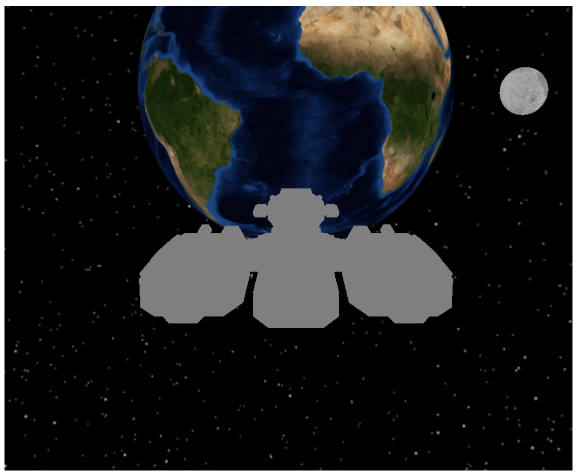


Figure 4: A possible eye position overlooking the spaceship.

Responding to keyboard events

Fill in the missing code in the `keyboard()` function. The objective is to make the program respond to the keys 8 and 9, which should decrease and increase the `dist_factor_` variable controlling the distance between the eye and currently observed object.

Grading

Each part of this assignment is weighted as follows:

- Placing the objects in the scene: 30%
- Placing the eye in the scene: 30%
- Rendering textured objects: 30%
- Responding to keyboard events: 10%

What to hand in

A .zip compressed file renamed to `Exercise`*N*`-Group`*I*`.zip` where *N* is the number of the current exercise sheet and *I* is the number of your group. It should contain **only**:

- The files you changed (in this case, `solar_viewer.cpp`).
- A couple of screenshots clearly showing that you can display the planets, the moon, the spaceship and that you can navigate the eye using arrow keys and keys 8/9.
- A `readme.txt` file containing a description of how you solved each part of the exercise (use the same numbers and titles) and whatever problems you encountered. Indicate what fraction of the total workload each project member contributed.

Submit solutions to Moodle before the deadline. Late submissions receive 0 points!