CS6533: Interactive Computer Graphics
3D Coordinate Systems

09/17/2013

Assignment Objectives

Your program will use OpenGL to draw two cubes as well as some scenery, the cubes will become robots in the future. You will implement a number of different viewpoints, and allow the cubes to be moved around the scene. They will move with respect to various frames, including the two cubes' frames and external points of view. Skeleton codes are provided for this assignment.

Step 1: Data Structures and Initial Setup

We provide you with a matrix library which will allow you to compose affine transformations, apply them to vectors, and convert matrices to and from OpenGL's required format. You will use this matrix library to represent the coordinate transformations and frames for various objects. Recall, that for any object, we can express its frame, $\vec{\mathbf{o}}^t$, as $\vec{\mathbf{o}}^t = \vec{\mathbf{w}}^t O$ where $\vec{\mathbf{w}}^t$ is the world frame and O is some appropriate matrix.

Start with a single cube (or other simple object) in the center of the world frame. The cube should have a corresponding matrix that stores its frame's information.

Add in a second cube and set the matrices (describing rotation and translation for the cubes) such that the two cubes do not overlap. Also make these cubes different colors to easily distinguish them from each other.

Step 2: Complete `linFact` and `transFact` (1 point)

Implement the functions `linFact` and `transFact` in `matrix4.h`. The purpose of these functions will be described in class (also in Chapter 6 of the book).

Step 3: Views (1 point)

Your rendering is done with respect to some eye frame. There is currently some default eye that has been set up. We want to now make the choice of eye frame interactive. Program in the ability to cycle between 3 different choices for the eye when the 'v' key is pressed:

1. The sky camera frame
2. The frame of cube 1
3. The frame of cube 2

Each time the 'v' key is pressed, the program should output which view mode it is currently on. (Recall our convention that cameras look down the negative z axis).

Step 4: Object Manipulation (3 points)

Manipulation Modes (2 points)

Now that we want to allow for greater user interaction, we'll have to track two key pieces of information (you may store them as application globals): (1) the object we wish to manipulate (2) $\vec{\mathbf{a}}^t$: the frame with respect to which it is manipulated. Make it so that pressing the 'o' key cycles through the different objects that we can manipulate. These objects are:

1. The sky camera
2. Cube 1 (later to be robot 1)
3. Cube 2 (later to be robot 2)

Depending on the current eye and current object being manipulated, we have a couple possibilities for $\vec{\mathbf{a}}^t$.

If the current object being modified is a cube and the current eye is the sky camera, then the only available frame for $\vec{\mathbf{a}}^t$ should be the *cube-sky* frame. This is a hybrid frame having the center coinciding with the cube's center, but with axes that are parallel to the sky camera frame's axes.

If the current object being manipulated is cube i and the current eye is cube j's, then $\vec{\mathbf{a}}^t$ should be the *cube i-cube j* frame. This is a frame with center coinciding with cube i's center and axes parallel to cube j's axes.

If the current object being modified is the sky camera and the eye is the sky camera, pressing 'm' should switch between two frames for $\vec{\mathbf{a}}^t$:

1. *World-sky* frame: center at world's origin and axes aligned with sky camera.
2. *Sky-sky* frame: the sky camera's frame.

You should not allow modifying the sky camera when the current camera view is a cube view.

Mouse Movement (1 point)

Now implement the actual response to user interaction. Allow the user to use the mouse to rotate and translate the sky camera and cube objects. The controls should be:

1. Left button: moving the mouse right/left rotates around the y-direction.
2. Left button: moving the mouse up/down rotates around the x-direction.
3. Right button: moving the mouse right/left translates in the x-direction.
4. Right button: moving the mouse up/down translates in the y-direction.
5. Middle button (or both left and right buttons down): moving the mouse up/down translates in the z-direction.

The signs of the rotations/translations depend on the current choice of eye, the object that is being manipulated, and $\vec{\mathbf{a}}^t$ as follows: (When in doubt, compare to the solution executable.)

1. In the simplest case, when we manipulate one of the cubes and the eye is at the other cube, or the eye is at the sky camera, then:
   (a) Pressing the left button: a rightward mouse displacement corresponds to a positive y-rotation (w.r.t. a right-handed coordinate system).
   (b) Pressing the left button: a upward mouse displacement corresponds to a negative x-rotation (w.r.t. a right-handed coordinate system).

(c) Pressing the right button: a rightward mouse displacement corresponds to a positive x-translation.

(d) Pressing the right button: a upward mouse displacement corresponds to a positive y-translation.

(e) Pressing the middle button: a upward mouse displacement corresponds to a negative z-translation.

2. If we are manipulating the sky camera, while the eye is at the sky camera, and $\vec{\mathbf{a}}^t$ is the world-sky frame, then we invert the sign of both the rotations and the translations that we apply when moving the mouse.

3. else we invert the sign of only the rotations that we apply when moving the mouse.

Reset Scene (1 point)

Reseting the entire scene is a common functionality provided in most 3D graphics applications. Implement the `reset()` function in `asst2.cpp` (make sure it's called when pressing 'r') so that:

1. The transformations of the sky camera frame as well as the cube frames are reset to default.

2. The current view and the current object being manipulated are also reset to default.

3. In sky camera mode, $\vec{\mathbf{a}}^t$ is reset to the *World-Sky* frame.