CSUS COLLEGE OF ENGINEERING AND COMPUTER SCIENCE
Department of Computer Science

CSc 155 – Advanced Computer Graphics
Fall 2018
Dr. Muyan

Assignment #2 – 3D Model Viewing

Due Date: Monday, October 15th (11:59 PM)

The objective of this assignment is to gain experience: (1) using buffers to send vertex models and texture coordinates through vertex attributes, (2) building matrix transformations and sending them as uniform variables, (3) using texture images, (4) building viewing matrices for controlling a camera, and (5) building simple models and textures. This time the world is a "fantasy solar system". Your program will maintain a *camera* object manipulated by keyboard input to enable viewing the 3D world from different vantage points.

World Objects

The 3D world must contain at least the following objects:

1) one "sun" positioned at the origin
2) two or more "planets" orbiting the sun
3) orbiting "moons" for both of the "planets" (or for at least two planets, if you have more than two planets). each planet with moons should have at least one moon.
4) a single set of colored *axes* showing the world XYZ axis locations.

Each of the solar system objects should be instantiated from a class defining the object in its own local (also called model or object) coordinate system and manipulated using transformations to position and orient it in the world. The sun is located at the world origin and rotates on its own axis (like our real sun). The planets orbit around the sun while rotating on their own respective axes. The moons orbit around the planets while rotating on their own respective axes. Each object must have its own independent rate of rotation on its axis, and the planets/moons must have different orbital speeds.

At least one of the models used to build your solar system must be designed by you, by hand, including the vertices (that specify the vertex positions and texture coordinates) making up the model. It can be a simple shape, but don't just copy it from the book. At least one of the remaining objects must be instantiated from the graphicslib3D **`Sphere`** class. The remaining solar system objects can be all spheres or some of them can be graphicslib3D **`Torus`** objects. Please see "Using the Sphere class" section of Chapter 6 of the book for tips on getting the vertex positions and texture coordinates from the **`Sphere`** object. The same mechanism can also be used for the **`Torus`** object.

You must use the graphicslib3D **`MatrixStack`** class to manage the MV matrix transforms used for all of the bodies in your solar system. This system should use only one **`MatrixStack`** instance.

You may keep all the planets in the same plane (the way our real solar system works; this plane is called the *ecliptic*), but that is not a requirement – you may instead have different planets use totally independent orbital paths if you want. Also, it is **not** a requirement to deal with inter-planetary collisions; you may allow the planets to "pass through" each other.

Textures

All of the objects in your solar system must be textured using image texture maps. Your system must utilize a total of at least two different texture map images, (you are encouraged to utilize more). At least one of the textures must be created by you (such as with a paint tool), and at least one must come from an already available source (such as from the book's ancilliary files, or the web).

As indicated in the course syllabus, your submitted readme.txt file must clearly attribute the source(s) of every one the textures used in your project, and provide evidence that you have permission to use the texture. Depending on where you got an image, do this as follows:

- Permission has already been granted for textures from the book. If you use any of those, indicate they are from the textbook, and also copy the source information provided with the ancilliary files (look at SourcesReferences.txt under the ModelsTextures directory).
- If you created the texture yourself, just state that in your readme.txt.
- If you use a texture from the web, either provide evidence that you received permission from the copyright holder (e.g., an email), or provide a link to where it states that the images are public domain, or a link to their terms of use plus a brief note how you are complying with those terms.

Camera Control and World Axes

Your program must define a *camera* object whose orientation is managed using its own "UVN" axes. The camera object should have variables for *location* and for three unit vectors corresponding to its axes. **`Camera`** class should also have a method called **`computeView()`** that computes the View matrix based on camera location and orientation. Camera location and orientation values are then manipulated by single-character keyboard input, as follows:

w – move the camera forward a small amount (i.e. in the positive-N direction).
s – move the camera backward a small amount (i.e. in the negative-N direction).
a – move the camera a small amount in the negative-U direction (also called "strafe left").
d – move the camera a small amount in the positive-U direction (also called "strafe right").
e – move the camera a small amount in the negative-V direction ("move down").
q – move the camera a small amount in the positive-V direction ("move up").
← and → (left and right arrow) – rotate the camera by a small amount left/right around its V axis ("pan").
↑ and ↓ (up and down arrow) – rotate the camera by a small amount up/down around its U axis ("pitch").

Consider having member functions in **`Camera`** class to handle all the abovementioned operations.

Also, the SPACE bar should toggle the visibility of the world axes. You can use **`glDrawArrays(GL_LINES, …)`** to draw these axes (i.e., you can define XYZ axes as another object in your scene).

To create the KeyStroke objects needed for rotating the camera and toggling the visibility of world axis you can use:

```
KeyStroke.getKeyStroke("LEFT"), KeyStroke.getKeyStroke("RIGHT"),
KeyStroke.getKeyStroke("UP"), KeyStroke.getKeyStroke("DOWN"),
KeyStroke.getKeyStroke("SPACE")
```

After moving the camera, to compute its new location, you can multiply the related unit vector of camera (U, V, or N) with a certain constant and add the resulting vector to the old camera location. You can use **`normalize()`** in graphicslib3D **`Vector3D`** and **`mult(double theMultiplier)`** and **`add(Point3D p2)`** in graphicslib3D **`Point3D`** to perform these operations. See appendix for more tips.

After the pan or pitch, to compute the new unit vectors of the camera, you can create a matrix which represents a rotation with specified degrees around the related camera axis (i.e., V-axis for pan and U-axis for pitch) and then multiply this matrix with the camera axes that need to be rotated (i.e., U and N axes for pan, and V and N axes for pitch). You can use **`rotate(double degrees, Vector3D axis)`** in graphicslib3D **`Matrix3D`** and **`mult(Matrix3D mat)`** in graphicslib3D **`Vector3D`** to perform these operations. See appendix for more tips.

Program Structure

You should use the JOGL **`FPSanimator`**. The timing must be based on elapsed time, so that the animation appears the same regardless of what type of machine it runs on. And you will need key bindings to handle the various keyboard commands. These key bindings then do whatever computations are appropriate (update the camera's location, update a rotation angle, etc. – but not invoke any OpenGL methods). Your **`display()`** method will then need to get the camera state, and build the appropriate *model*, *viewing* (by calling **`computeView()`**method in **`Camera`** class), and *perspective* transformations.

Additional Notes

- You may find the graphicslib3D **`Shape3D`** class (from which the **`Sphere`** and **`Torus`** extends from) useful as the base class from which to derive your own model class(es); however, this is not a requirement.
- It should be possible to see most of the objects on screen at the same time. Don't make the planets or moon so small (or so distant) that you can't see them when the sun is in view. Also, the rotation of solar system objects must not be so fast as to make it difficult to observe their textures.
- The lines showing the positive X, Y, and Z world axes should (when enabled) be colored as red, green, and blue, respectively. You can assign colors to XYZ axes using texture mapping.
- For the projection matrix, use a perspective matrix transformation like the one we discussed in class.
- You may add additional camera controls if you wish. For example, you might want to add a control to "look at" the sun which can become handy if you ever loose sight of the solar system when you are controlling the camera (i.e., it would allow you to automatically orient the camera, so that you look at the middle of the solar system where the sun is located). You may also add a control to "roll" your camera (i.e., rotate it around its N axis). If you add these additional controls, include an explanation in your readme file that specifies which keys are used for these controls.
- Your code must be contained in a Java package whose name is exactly "**`a2`**" (lower case). As before, the "main" class in your program be named exactly "**`Starter`**". It should be possible to run the program from a command prompt in the parent directory by typing the command:

```
java –Dsun.java2d.d3d=false a2.Starter
```

Deliverables

Submit to Canvas **TWO files (zip file and txt file) SEPERATELY** (i.e., do NOT place the txt file inside the zip file). The *ZIP file* should be named as YourLastName-YourFirstName-a#.zip (e.g., Doe-Jane-a2.zip) and should contain:

(1) your Java source files, compiled (.class) files, and GLSL shader files
(2) your texture image files
(3) a screen capture (.jpg) of your program running

The submitted files must be organized in the proper hierarchy in the ZIP file; that is, the .java and .class files must be contained within a subdirectory named “a2”.

The TEXT file (i.e., not a pdf, doc etc.) should be named as readme.txt and should list:

(1) a description of the object and texture that you created yourself
(2) source information for other textures you used
(3) additional information you want to share with the grader
(4) the lab and the lab machine you have used to test your program

You will receive the grader comments on your text file when grades are posted.

Appendix – Tips on Implementing Camera Operations

Changing camera location

Below you see a possible implementation for moving the camera in its positive-V axis (“move up”). You can implement other movement operations in a similar way…

```
//initialize Camera location and V axis (you can use different values than below)
Point3D Cam_loc = new Point3D(0,0,0);
Vector3D V = new Vector3D(0,1,0);
...
//move camera up
Point3D V_mov = new Point3D(V.normalize());
V_mov = V_mov.mult(amount); //amount is a small value such as 1.5
Cam_loc = Cam_loc.add(V_mov);
```

Changing camera orientation

Below you see a possible implementation for rotating the camera about its V axis (“pan”). You can implement other rotation operations in a similar way…

```
//initialize UVN axis (you can use different values than below)
Vector3D U = new Vector3D(1,0,0);
Vector3D V = new Vector3D(0,1,0);
Vector3D N = new Vector3D(0,0,1);
...
//pan the camera
Matrix3D V_rot = new Matrix3D();
V_rot.rotate(amount,V); //amount is a small value such as 3 degrees
U = U.mult(V_rot);
N = N.mult(V_rot);
```