Program #5: Modeling 3
Updated: March 6, 2024

CS 5465: Computer Graphics
R. M. Parry

This assignment is to be completed individually and the work you submit must be your own. You may edit any of the classes in the framework code and you may create new classes to complete the requirements. However, you do not need and should not use any other libraries.

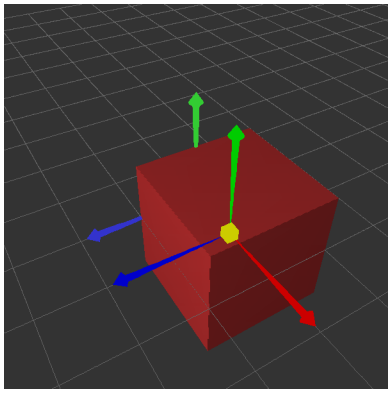
This is part 3 of the modeling assignment and includes adding "click-and-drag" manipulators to the graphical user interface.

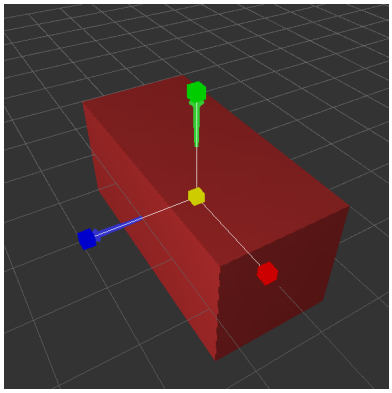
1 Assignment

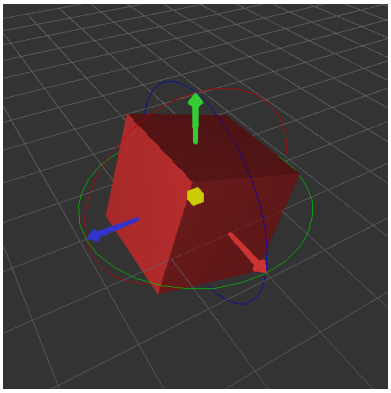
The same framework code from the previous modeler assignments provides a means for editing the transformations by typing in numbers in the attribute panel. This can be useful if you want to apply a particular transformation exactly, but to get things where you want them it is much easier to position them interactively and visually.

`https://cs.appstate.edu/rmp/cs5465/model3.zip`

A widely used approach to transforming objects in 3D is by the use of manipulators. Manipulators are user interface elements that appear in the 3D scene to give the user direct visual cues about the transformations being applied. Transformations are edited by clicking on parts of the manipulator and dragging. Here is a look at a three manipulators:


translate-manip-x


scale-manip-x


rotate-manip-x

It's important to remember that a manipulator operates on a transformation node, not directly on an object itself. As usual, this transformation applies to all objects below it in the tree, and there may be other transformations above and below it.

Transformations are represented as a nonuniform scale, followed by x, y, and z rotations, and then a translation. Each manipulator (rotation, scale, translate) affects one of these three pieces of the transformation. When the user adjusts *one axis* of a manipulator, the result is a change in exactly one number in the transformation (the x, y, or z component).

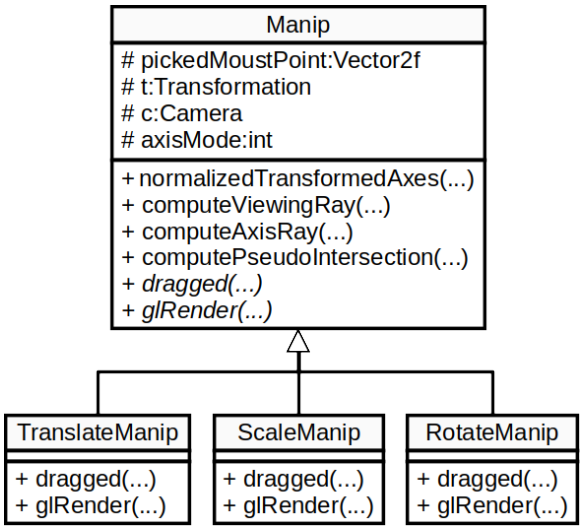
When manipulators are drawn on the screen they should appear in 3D at the position of the origin in the coordinates below the transformation being edited. The arrows of the scale and translation manipulators should point exactly along the axes that correspond to x, y, and z. We call these the *manipulation axes* of the manipulator. The circles of the rotation manipulator should be perpendicular to the three axes of rotation, unless the transformation being manipulated is below a

nonuniform scale in the tree.

The framework provides the infrastructure for drawing the manipulators and detecting when they have been clicked. It is your job to position the manipulators and axes in world space and to map the user's mouse motions into changes to the selected transformation. You should implement manipulators for translation, rotation, and scaling based on the requirements below. The general principle is that when the user drags the mouse in a direction that the manipulator handle can move, the handle should move with the mouse, staying glued to the mouse pointer. (There are exceptions where it is difficult or nonsensical to have the transformation follow the mouse.)

1.1 Class Diagram

The `modeler.manip` package contains the classes used for mouse manipulation of the scene. The abstract `Manip` class keeps track of the picked mouse point, the axis mode (`PICK_X`, `PICK_Y`, `PICK_Z`, or `PICK_OTHER`), current transformation being affected, and the camera. The concrete base classes implement the manipulator for translation, scale, and rotation.



1.2 Overview

The manipulators are all subclasses of `Manip`, and the main action is in the `dragged` method. This method has arguments `mousePosition`, which is the point in the viewport where the mouse is, and `mouseDelta`, which is the offset from the previous point that was reported. The `Manip` class also stores the point where the user first clicked in the field `pickedMousePoint`.

One good way of setting up the follows-mouse constraint for manipulation along axes is as follows. If the user clicks on a point on a manipulation axis, then drags to another point on the axis, this means the eye rays corresponding to the initial and final mouse positions both intersect the axis. By computing these intersection points you can figure out what the transformation is that takes one to the other. If the user's clicks aren't exactly on the axis, this means the rays don't exactly intersect the axis, but we can get a reasonable result if we just use the point of closest approach (the "pseudo-intersection" point) instead of the exact intersection. These computations are pretty easy to do using parametric lines, as we did in ray tracing. It breaks down like this:

1. Write a method to compute the parametric line (the viewing ray) in world space corresponding to a point in the image. The mouse position is given as a 2D vector in the range $[-1, 1]$. The

view volume is defined in the perspective camera as a field-of-view in the y direction (`fovy`). That's the angle at the view point between the bottom and top of the view volume. The camera also provides the aspect ratio (`aspect`) which is the ratio between the width and height of the view volume. It also has the **u** and **v** vectors for the view plane stored in `right` and `up` fields, respectively.

2. Write a method to compute the parametric line in world space that describes the manipulation axis. You had to do this anyway to position the manipulator in space.

3. Write a method to compute the pseudo-intersection of two parametric lines, returning the two t values for the closest pair of points on the two lines.

4. Do the manipulation by using this machinery to compute the initial and final t values on the manipulation axis; then updating the transformation is a simple matter.

You may find that your manipulator behaves a little strangely if the vanishing point of the manipulation axis is in the image and the user drags past it. This is OK.

1.3 Compute Axis Ray

The `Manip.computeAxisRay` method edits the point `p` and vector `d` in place. The point `p` should be the origin of the current transformation, after any translations including the current one. The direction vector should point in the positive axis direction. The axis direction should combine all rotations and scales from its ancestors but not the current transformation.

1.4 Compute Viewing Ray

The `Manip.computeViewingRay` method takes in a 2D mouse location in screen coordinates and edits the point `p` and vector `d` in place. The ray should originate at the camera's `eye` position, pass through the 2D mouse point and arrive on the parallel plane that passes through the camera's `target` location. The camera's `computeMotion` method may be helpful.

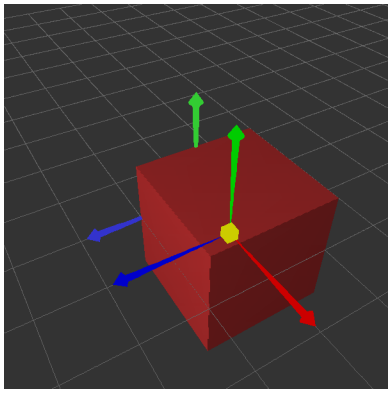
1.5 Compute Pseudo-intersection

We want to compute the nearest point on the selected axis to the viewing ray. To do this we compute the "pseudointersection" between the two rays. The pseudointersection is the pair of closest points on the two rays.

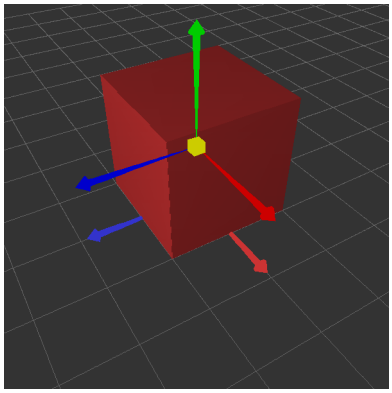
The `Manip.computePseudointersection` takes two rays as input, and solves for the t-value for each ray that produces the pair of 3D points that are closest to each other. Only, the t-value for the second ray is returned.

1.6 Translation Manipulator

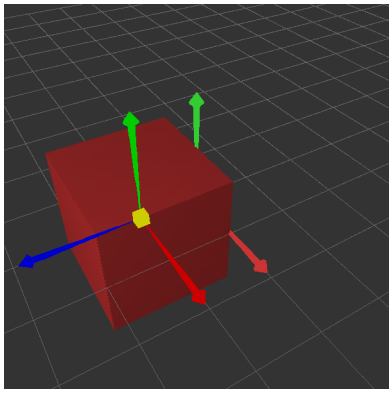
The translation manipulator displays the three coordinate axes for the transformation, and the user moves the selected object by clicking on one axis and dragging in the direction of desired motion. By clicking on the center (yellow box) of the manipulator one can drag the objects around freely in the viewport, resulting in a translation parallel to the view plane.



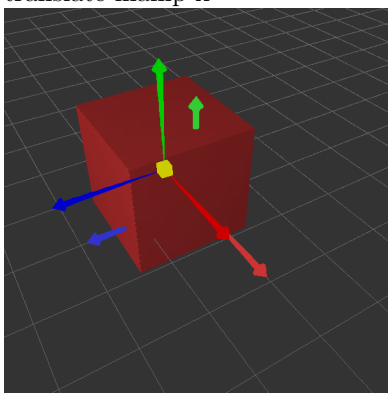
translate-manip-x



translate-manip-y



translate-manip-z



translate-manip-o

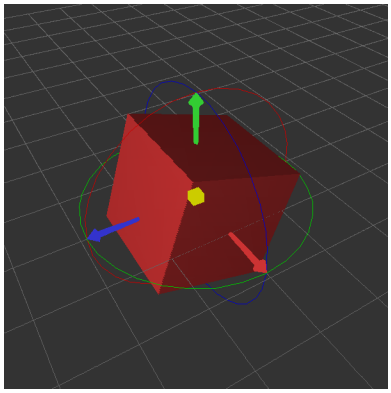
The translation manipulator displays three arrows that represent the directions of motion that will result from an x, y, and z translation in the coordinates of the selected transform. If the user clicks and drags exactly in the direction of the axis, the resulting translation should exactly follow the mouse. When the drag is not parallel to the selected axis, the translation should follow the mouse as much as possible while still operating along the selected axis.

If the user clicks the center of the manipulator (yellow box), you should apply a translation that is parallel to the view plane so that the origin of the manipulator moves the same distance as the mouse— that is, if you move the mouse to the right 5 pixels and up 2 pixels, the manipulator should move to the right 5 pixels and up 2 pixels, carrying the affected objects with it. This should be true no matter what transformations are above the selected one.

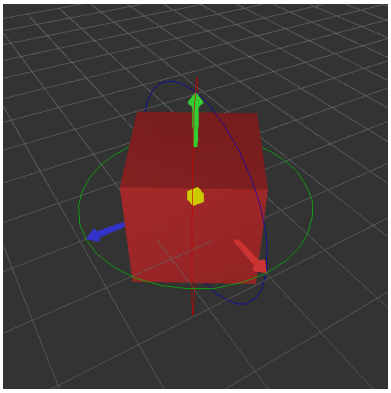
1.7 Rotation Manipulator

The rotation manipulator displays three circles on a sphere surrounding the origin of the transformation's coordinates. Each circle is perpendicular to one of the rotation axes. Clicking on that circle and dragging performs a rotation around that axis. Because getting the transformation to follow the mouse is complicated, for this manipulator, just map vertical mouse motion directly to the rotation angle such that dragging the entire height of the viewport results in two revolutions.

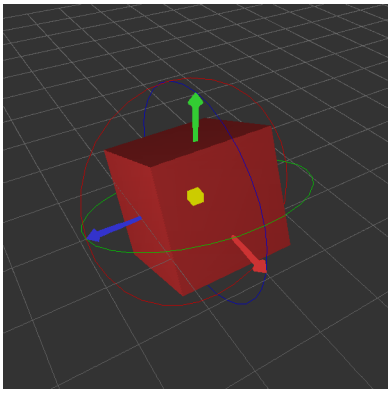
When it works the result of the tests should look like this:



rotate-manip-x



rotate-manip-y

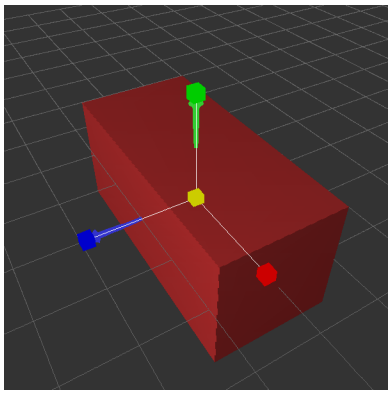


rotate-manip-z

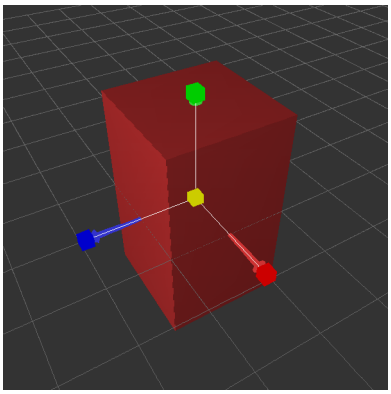
1.8 Scale Manipulator

The scaling manipulator shows the three scaling axes by drawing three lines with small boxes the ends. If the user clicks on one of these three axes and drags in the direction of the axis, a scale should be applied such that a point on an affected object that started under the mouse would stay under the mouse. For example, dragging the mouse from a position of 0.5 to 1.0 on the axis should double the scale; dragging from 1.5 to 2.0 should increase the scale by 133%. If the user clicks on the center cube and drags, a uniform scale should be applied. Just map vertical mouse motion such that dragging the entire height of the viewport results in scaling each axis by 4 times.

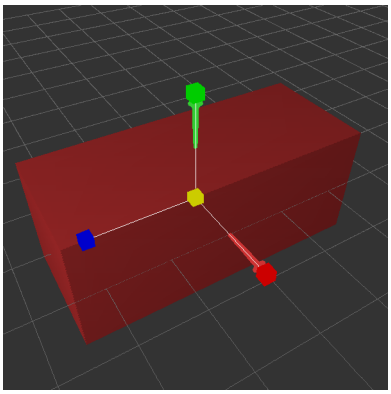
When it works the result of the tests should look like this:



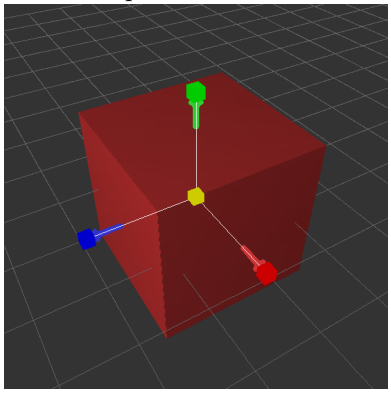
scale-manip-x



scale-manip-y

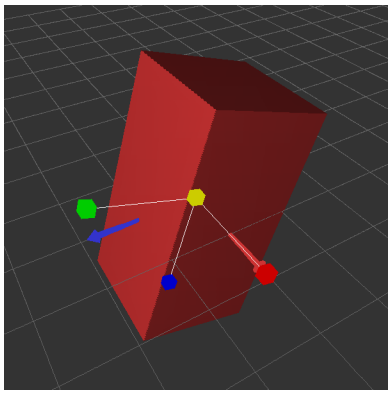


scale-manip-z

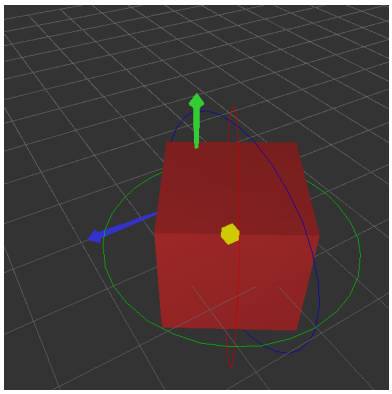


scale-manip-o

1.9 Some Combined Manipulations



rotate-scale-manip



translate-rotate-manip

2 Java Development

To work on this program you can use your favorite Java IDE such as Eclipse or VS Code. We will be using **Java 8** with **JUnit4**. Later versions of Java will not work for some of the assignments. The working directory for your project should be the `model` folder. The source location should be the `model/src` folder.

You can install the JDK for Java 8 here:
`https://adoptium.net/temurin/releases/?version=8`

On linux, you may install OpenJDK version:
`https://openjdk.org/install/`

You can get JUnit4 here:
`https://github.com/junit-team/junit4/wiki/Download-and-Install`

2.1 Running the Program

To run the program on a particular XML file, you can run `modeler.MainFrame` as a Java Program with zero or more XML files as arguments. If you provide no arguments, the GUI opens and you can interact with it.

Render an XML scene using the File→Open dialog.

Access the manipulators by selecting a transformation in the "Scene" panel, and Edit→"* Selected"

If you run the program with an XML file as command line argument, the GUI will open, create the scene, render the image and save it to a PNG file in the same directory as the XML, then close the window.

2.2 Running the Tests

Use the standard way to run JUnit4 tests in your IDE to run the tests in `tests.ModelTest3`.

3 Submissions

Your assignment will be graded by Web-CAT. Compress your `src` directory into a ZIP file and upload it here:
`http://webcatvm.cs.appstate.edu:8080/Web-CAT`