Lab 10: Modeler

Introduction

In this lab you will implement 3D interaction controls for a simple 3D modeling program. This lab relies on concepts from Camtrans, Sceneview and Intersect. The general idea is that we want to be able to transform objects in a scene just by clicking and dragging on them. The goal of this lab is to give you some practice interacting with objects in a scene, which is an optional feature for your final project. You will have to implement the following object interactions:

- Uniform scale
- Look vector translate
- Film plane translate
- Trackball rotation

Copy the code from /course/cs123/src/labs/lab10 and open up lab/modelerscene.cpp. Note that all of your work for this lab will be in the ModelerScene::mouseDragged() method.

Selecting the object

The ModelerScene class has an array (actually, a QList) of primitives called m_primitives, a pointer to the currently selected primitive called m_selection, and the world-space coordinate of the last click called m_hit (which will be on the surface of m_selection). We have written a selection method called setSelection() that takes in the mouse position and sets m_selection and m_hit. Each primitive contains the object’s modeling transformation which we can get using m_selection->getMatrix() and set using m_selection->setMatrix(Matrix4x4 m).

Step 1 (uniform scale)

We will start with the simplest of the interactions, uniform scaling. For this interaction, you should uniformly scale (same scale factor in all directions) the selected object based on the vertical distance between the current mouse position
and the previous mouse position. If you drag the mouse up your object should scale up, and if you drag it down it should scale down. One good equation to use is this:

\[ f = s^{\Delta y} \]

This gives the scale factor \( f \) in terms of a speed \( 0 \leq s \leq 1 \) and the change in vertical mouse position \( \Delta y \). You will find the delta variable helpful for the change in mouse position. To actually modify the current object, you will need to change the primitive’s modeling transformation. We’re modifying the original object so we want to apply the transformation before any of the other modeling transformations, which means we want to left-multiply it by our current modeling transformation. The support code will automatically re-render the scene for you.

**Step 2 (look vector translate)**

Look vector translation will use the same up-down mouse input, except dragging up this time will translate the object down the look vector (away from you) and dragging down will translate it towards you. Use the camera variable to get the look vector and find a translation speed that feels right. Unlike uniform scaling, we’re translating the object down the look vector after all other transformations have been applied, so we want to right-multiply it by our current modeling transformation to get the new transformation matrix.

**Step 3 (film plane translate)**

The next two transformations are a little trickier. For film plane translation, we want to translate our selected object in a direction parallel to the current film plane. We also want the translation to follow our mouse so that when we release our mouse it is pointing to the same position on the object. We will accomplish this by defining the translation plane as the plane passing through the intersection point (stored in \( m\text{-hit} \)) that is parallel to the film plane.

The diagram below represents this translation, where \( I_{old} \) is the intersection point on the object, \( I_{new} \) is where we want the intersection point to be, \( L \) is the look vector, and \( R \) is the mouse ray.
In intersect, you calculated \( R \) using the film-to-world matrix, but for this lab you can just use `getMouseRay(Vector2 mouse, Camera camera)`. Notice that \( I_{\text{new}} \) is the intersection of the mouse ray and the translation plane. Now the problem is as follows: we know \( L, R \), (both should be normalized) and \( I_{\text{old}} \), and we wish to compute \( I_{\text{new}} \) (because this gives us the translation vector \( I_{\text{new}} - I_{\text{old}} \)).

Let \( V_{\text{old}} = I_{\text{old}} - \text{eye} \) and \( V_{\text{new}} = I_{\text{new}} - \text{eye} \). Don’t normalize them!

Then, to solve this problem, we note that:

\[
V_{\text{old}} \cdot L = V_{\text{new}} \cdot L
\]

Ask a TA if that isn’t clear.

Next, we look at the equivalent form of the dot product (\( A \cdot B = |A||B|\cos(\theta) \) where \( \theta \) is the angle between \( A \) and \( B \)):

\[
|V_{\text{old}}||L|\cos(\theta_{\text{old}}) = |V_{\text{new}}||L|\cos(\theta_{\text{new}})
\]

Which means that the magnitude of \( V_{\text{new}} \) is simply:

\[
|V_{\text{new}}| = \frac{V_{\text{old}} \cdot L}{\cos(\theta_{\text{new}})}
\]

Finally, to get \( I_{\text{new}} \), we can cast a ray from the eye in the direction of \( R \) with magnitude \( |V_{\text{new}}| \).

From there you can find the translation vector from \( I_{\text{old}} \) to \( I_{\text{new}} \). Construct the translation matrix and multiply it with our modeling transformation. Which
side of our modeling transformation should it be applied to? Remember to update \( m_{hit} \) to your new hit point at the end of your film plane translation.

**Step 4 (trackball rotate)**

Trackball rotation is widely used in interactive graphics. The idea is that to circumscribe (or inscribe) an object with a sphere and rotate that sphere by dragging the mouse, similar to rotating the object with your finger. We have provided a method called \( \text{intersectSphere} \) (Matrix 4x4 \( \text{cur}_\text{model} \), Vector 4 \( \text{eye} \), Vector 4 \( \text{d} \), Vector 4& \( \text{hit} \)) that returns true if there is an intersection, and false otherwise. If there is an intersection, it modifies the last argument \( \text{hit} \) to hold the world-space intersection point of the circumscribing sphere and the ray starting from \( \text{eye} \) and traveling along \( \text{d} \).

We need two things in order to find an appropriate rotation matrix: an axis of rotation and an angle of rotation. To find these, we will need to find the vectors from the center of the sphere to the intersection points found using the old and current mouse positions. Note that you should only perform the rotation if \( \text{intersectSphere}() \) returns true for both the old mouse position and the current mouse position.

**Center of the sphere** To find the center of the sphere, we just need to take the modeling transformation for our object and apply it to a point representing the origin \((0,0,0,1)\) (or just read the right most column of the matrix since all we’re looking for is the translation component).

**Intersection points** We need two intersection points on the sphere, \( V_1 \) and \( V_2 \):

![Diagram of trackball rotation](image)

You should find \( V_1 \) and \( V_2 \) by calling \( \text{intersectSphere}() \) twice. From these three points we get vectors \( a \) and \( b \)—the vector from the center to \( V_1 \) and the vector from the center to \( V_2 \). All we have to do to find the axis of rotation is take
the cross product of $a$ and $b$ and normalize the result. Be careful of which order you take the cross product. (Or don’t and just switch it later if it’s wrong...)

To find the angle of rotation about the axis, we need to find the angle between vectors $a$ and $b$. This is accomplished with the cosine law:

$$\cos \theta = \frac{a \cdot b}{|a||b|}, \quad \theta = \arccos \left( \frac{a \cdot b}{|a||b|} \right)$$

Now that you have the axis and angle of rotation, construct the corresponding rotation matrix using the center of the sphere as your center of rotation. Remember getRotMat()?

If we’re applying the transformation about the current center of the sphere (the result of our current modeling transformation being applied to the object), on which side of our modeling transformation should we place this matrix?

**Finishing up**

Once you have all four interactions done, show a TA your work to get checked off for this lab.

Congratulations, you have completed the final CS123 lab! With this introduction to real time graphics and shaders, we hope that you will be able to come up with some even more exciting and creative final project ideas. We hope you enjoyed these labs as much as we enjoyed making them. Thanks again for being our guinea pigs!

- The CS123 TAs