COMP SCI 3GC3: Computer Graphics
Assignment 3

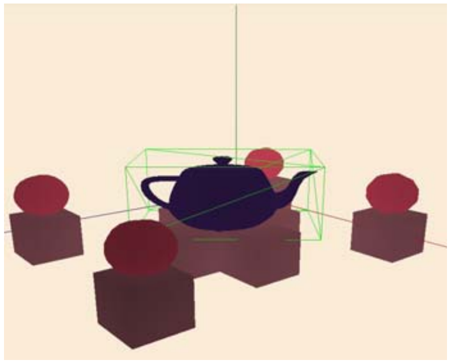
Due: Tuesday, Dec. 8, 2015 at 12:00pm (noon).

NO LATE ASSIGNMENTS ACCEPTED, please aim for soft deadling of Dec 3.

This project is worth 10% of your final grade. You may work in teams of two for this assignment.

Simple Modeler

Develop a simple 3D modeling system using OpenGL and C/C++. The system should support selection (via ray-casting) of objects added to the scene, and allow subsequent interactive manipulation (i.e., transformation) of the objects. The figure to the right shows an example of this. The teapot is currently selected, as indicated by the wireframe box around it; hence any transformation commands applied will affect the teapot currently.



The actual rendering of objects in the scene should utilize a scene-graph: simply loop over your object list, and render each object at its corresponding world coordinate location. The main challenge in this assignment is to use ray- casting to allow mouse-based interaction with the scene.

Scene Object Structure & Rendering

Start by creating a scene object structure/class. It will need to contain (at minimum) the object position (a 3D point), orientation (could be represented as a 3D point, with each value representing the rotation angle about the corresponding axis), scale factor (could also be represented as a 3D point), its drawing material, its object type (i.e., cube, sphere, etc. – see below), and bounding volume information for ray intersection tests: either six bounding planes, or the "min" and "max" corners of the bounding box (which can be used to compute the bounding planes on the fly). You may also need to add additional fields depending on your specific implementation details. Create a list of these scene objects (e.g., using the STL list) to keep track of the position of all objects in the scene.

The display loop should simply iterate over the list of objects, and render each as unit-sized version of the object (based on its object-type field) subject to any transformations applied. You may use the various glutSolid<SHAPE> functions for this. You should also draw a large (e.g., 100 x 100 units) ground plane on which to display the objects. Your scene should use lighting, see below. Note that by using the built-in shapes, you do not need to calculate the object normals except for the "floor" plane. If using any custom object, you will have to compute these, however.

Adding/Deleting Objects

Your scene (and object list) should be initially empty. Pressing keys should add new objects to the scene at the world origin – which keys are used for this is up to you. The newly created object should be initially selected (i.e., the target for manipulation). Right clicking on an object should delete that object from the scene. You will need to use mouse ray picking for this (see below).

Interacting with Objects

Once an object is selected, it should be indicated as such with some kind of clear visual feedback (e.g., in the figure above, the green wireframe bounding box on the teapot). The selected object can be manipulated in real-time via the keyboard. Your system should support translate, rotate, and scale transforms (both positive and negative transforms in all axes). The actual keys used to accomplish this are left to you, but must be documented. For example, you might bind the arrow keys to movement in the xz plane – so left and right would move the object in the x axis while up and down would move the object in and out along the z axis. To avoid running out of keys, consider using modifier keys (ALT, CTRL, SHIFT) with appropriate keys (e.g., pressing Z might rotate the object in the +z direction, while pressing SHIFT+Z rotates it in the –z direction, while CTRL+Z increases its z-scale factor, and CTRL+SHIFT+Z decreases it z scale factor). Additional optional features are outlined below to enhance this interaction (e.g., mouse-based instead of keyboard-based manipulation is possible).

Mouse-Ray Picking

Ray-based picking is the key to this assignment. First, you will have to compute a mouse ray upon clicking. A ray is a line in space, originating at a specific point with a (normalized) direction vector. The mouse ray can be found by "un-projecting" the clicked xy coordinate. The gluUnproject command takes the modelview and projection matrices and inverts them to invert the projection process. This will transform the provided xy point into a vector, specifically given by two 3D points: one on each of the near and far clipping planes. The mouse ray direction is then the normalized vector between these two points. The origin of the mouse ray is the camera's world-coordinate position (i.e., conceptually, mouse rays originate at the "eye", pass through the cursor, and into the scene).

Ray Intersection Tests

To determine the clicked 3D point and object, test the mouse ray for intersections all objects in the list, plus the ground plane. This will require a function to perform both a ray-plane intersection test and a ray-box intersection test. The ray-box test actually consists of 6 bounded ray-plane tests, with an added step to ensure that the intersection point is in the extents of the given face of the box. For example, consider a ray which strikes the +X plane of a unit cube centered at (0, 0, 0) at the point (1, 5, 2). Although the ray hit the plane of the +X face of the cube, it is outside of the extents of the cube (which would be between -0.5 and +0.5 in each of the y and z directions. Hence this ray would miss that face of the cube.

All intersection points should be kept in a list, which is then searched for the nearest intersection point, in order to allow selection of only the closest object. The reason your scene object data structure should include bounding box information is because it simplifies the ray-intersection tests – i.e., you only need to test the box surrounding that object, rather than testing the actual object's geometry (which is *far* more complex). Note that when the object is transformed, so too must be its bounding box. Alternatively, you can transform your ray by the inverse transformation applied to the object (to transform the ray into the object's coordinate space).

(Hint: A plane consists of four parameters, A, B, C which define the normal of the plane, and D, the distance from the origin to the plane. The following article explains how to find D given A, B and C: http://mathworld.wolfram.com/Plane.html). Note also that the A, B, C and D parameters of the plane are used for the intersection test with the ray.

Other Features

1. Object Support
   Your modeler should support at minimum five of the following objects: cube, sphere, cone, cylinder (as a special case of a cone), torus, teapot, tetrahedron, octahedron, dodecahedron, icosahedron. All of these can be drawn using glutSolid<SHAPE> where <SHAPE> is the appropriate object type (except cylinder, which is drawn as a cone with the same diameter on each end). See the GLUT documentation for details of these functions.

2. Lighting
   You should provide at least two light sources in the scene. Depict light sources as a small sphere. Their xyz position should be user controllable by pressing keys. Note that if you use the GLUT cubes, you will not need to specify normals except for the floor (ground plane).

3. Materials
   You should support at least five different materials. See the tutorial on lighting for details of how to encapsulate materials as a structure. Pressing the 1 through 5 keys should change the current drawing material, i.e., the initial material for all subsequently drawn objects. Pressing the "m" key should apply the current drawing material to the object selected to by the mouse cursor (recall that the keyboard function also takes the x and y coordinates of the mouse as parameters – re-use your ray intersection test here!). This will allow you to change the material of objects after they are added to the scene.

4. Scene rotation
   Pressing certain keys (your choice) on the keyboard should rotate the scene about its centre in the y and z axes.

5. Reset – "R" key
   Pressing the "r" key should clear the object list, setting it back to the empty starting scene.

6. Load / Save – "L" and "S" keys
   Pressing the "S" key should save the current scene to a file (text file is fine). Upon pressing the key, the console should prompt the user to enter a file name, and save the object list to that file. Pressing the "L" key should allow the user to load previously saved scene files. These can be represented as simply writing out all relevant data for all objects in the scene in text.

7. Graphics Features
   You should use backface culling for efficiency.

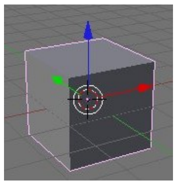
Implement any one (or optionally two for bonus marks) of the following extra features:

8. Texture Mapping
   Add three "materials" that use pre-loaded textures (check out the freeimage library for jpeg if you don't like ppm) instead of solid materials.

9. Camera Control
   Implement a simple camera model to allow the user to fly around the scene and view the scene from different points. You should allow for key-based movement, and mouse-based rotation of the camera.

10. Constraint-Based Object Movement
    Allow the user to click and drag objects to move them around, e.g., using the middle mouse button. Moving objects should maintain contact with another surface behind them (e.g., the ground plane or other objects) at all times. To implement this, perform continual ray-intersection tests between the mouse ray and the scene (excluding the moving object!!) while moving the object. Upon release, the object should "attach itself" to the model in the new position, i.e., avoiding intersection with other objects or the floor.

11. Handle-Based Object Transformation
    Allow the user to select objects (e.g., using the middle mouse button) to perform one of the three transformations (you can pick this option twice to count as two features with two different transformations). Upon clicking the object, a set of "handles" should appear around it, similar to those provided for translating objects in 3D Studio Max (see right). These generally appear similar to a small axis displayed around the object, and allow transformation in one axis at a time. For example, the handles to the right allow translation along each of the three world axes (x, y and z). You can visualize these handles in any convenient way. Clicking the handles will limit manipulation to only that axis at a time (i.e., use a ray intersection test to determine which handle was clicked). No collision detection is necessary, and objects may be left floating in space upon completion of a move.



12. Light Source ray picking
    Allow the user to click on a given light source (represented by a small sphere) to select it for subsequent movement operations, rather than selecting with a key. This will require adding a ray-sphere intersection test.

13. Additional Custom Shape
    Add an additional shape (some kind of mesh) requiring custom code to render. This likely requires specification of surface normal for your object. It might be rendered according to a function, or a pre-loaded mesh. It should *not* appear in the list of shapes that appears above.

14. Oct tree - efficiency
    Implement an oct tree – a data structure that recursively spatially subdivides the scene in to 8 octants. Store your objects at the leaves of the oct tree (rather than in a list) based on their positions. Use ray-box tests against the higher levels of tree to trivially reject entire groups of objects when performing the ray-intersection test, i.e., if a ray doesn't hit a certain octant, it cannot hit any of the objects contained in that octant. This should be much faster than searching a list when the number of objects gets large enough. Note: This is a challenging option! Not for the faint of heart!

15. Shadows
    Add shadows to the scene – objects should cast shadows away from light sources onto the ground plane and each other.

16. Other Feature
    Implement some cool feature of your own design! These should be of similar complexity to the other features listed above. If you aren't sure about a feature, feel free to contact me to ask about it.

<u>Clearly indicate in your documentation which extra features you implemented.</u>

When make is called, a program called "Modeller.x" should be compiled and generated.

Submission Notes

All programs must initially write to the DOS/Unix shell a list of commands (keyboard or menu items) and their uses in the program. The marker should not have to look at your source code to figure out how to run a program. Marks will be deducted otherwise!

You have the option of implementing your assignment on your platform of choice, BUT please make sure your programs can be fully compiled and executed on the departmental machines (e.g., ITB 237). Also be sure to include any additional libraries that you use. If the TA has to hunt for a specific library to compile your code, you will lose marks.

Submit your source code, makefile, readme, and any other resources to A3 folder of your SVN trunk. All source code and makefile should be located in A3 and not a subfolder. When 'make' is called, it should produce programs named: "Modeller.x".

Familiarize yourself with the department's policy on plagiarism and the university regulations on plagiarism and academic misconduct. Plagiarism will not be tolerated, and will be dealt with harshly.

If you choose to work in a group of two, please add a readme or print to the terminal both of your names and student numbers. Only one person should submit the code to the SVN.

Have fun, and be creative!