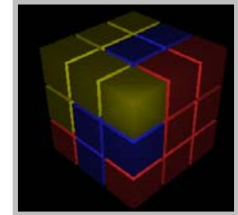
Dan Morris
CS 148

Handout #09
Summer 2005

Project 2
TOBOR and RUBIX



Due: Thursday, 7/14, 11:59pm

Project Goals

This assignment primarily deals with transformations in OpenGL. You will set up transformation hierarchies to draw and animate two complex objects using simple primitives.

Supplies

This project again uses OpenGL and GLUT. We've given you two .cpp files – tobor.cpp and rubix.cpp – to help get you started; your makefile will build both of these. The starter files set up OpenGL and GLUT for you and drawn some simple objects. The planetsup.cpp example that we discussed in class is also a valuable resource to get started, since it too uses a hierarchy of transformations.

You can get the stencil files and the demos from the syllabus page, or from:

/usr/class/cs148/asgn/pp2
/usr/class/cs148/demos

Part 1 – TOBOR:

You are Dr. Drago, the mad scientist intent on creating a 21st century Frankenstein (named TOBOR) from metal parts and silicon chips. Your goal is to infuse it with life and set it upon an unsuspecting Earth in order to achieve world domination. However, in order to solicit venture capital to support this project, you're preparing an OpenGL demo to show the power of your scheme.

In order to complete your diabolical task, you must create an object hierarchy that represents the body and limbs of TOBOR. Your robot should have a torso, two two-segment arms, two two-segment legs, and a head. Each upper arm will draw in the reference frame of the torso, each lower arm in the reference frame of the corresponding upper arm, etc.

Once you have modeled your robot, you need to bring him to life, all the while keeping him under your strict control. You should support the following keys:

- '**1**' & '**q**': right shoulder rotation
- '**2**' & '**w**': right elbow rotation
- '**3**' & '**e**': left should rotation
- '**4**' & '**r**': left elbow rotation

- '**5**' & '**t**': left leg rotation
- '**6**' & '**y**': left knee rotation
- '**7**' & '**u**': right leg control
- '**8**' & '**i**': right knee rotation
- **'a'** & **'s'**: rotate TOBOR

Once you have this working and you feel confident that TOBOR will not pummel you, another set of keys 'turns TOBOR on', initiating wild animatronic fun. The keys you need to support here are:

- 'z': animates right arm
- 'x': animates left arm
- 'c': animate legs

Animation can be just about any set of smooth movements that use both joints on a limb. See the demo for examples.

The starter code (tobor.cpp) sets up an OpenGL world with lights and a camera and draws a simple sphere, so you just need to fill in four functions:

- init_structures – called when your program starts up so you can set up your robot
- draw_robot – called every frame when it's time to draw
- keyboard – your GLUT keyboard callback
- idle – your GLUT idle callback (for animation)

Requirements:

- You must do all your drawing in display lists… this should save you time anyway
- You must use glPushMatrix and glPopMatrix to set up reference frames for each limb
- Limbs should be different colors than their parents (see the discussion of SetMaterial()) below

Support code

- One new piece of support code you'll work with is SetMaterial(). We want TOBOR to look pretty, so instead of just using glColor, we'll use OpenGL materials. We've set up a few for you; you can just call:

  SetMaterial(MAT_BLUE);

  …for example, to change the current material color to a nice shiny blue.

- Feel free to use glutSolidSphere and glutSolidCube to draw primitives; you can scale and rotate them to make them look more interesting. You don't need to use glVertex anywhere in this assignment, although you're welcome to create more complex primitives.

- How you define your reference frames is up to you… the arm may extend along its own y axis, its own z axis, etc. To help you find the current reference frame, we've given you a drawAxes() function that draws a set of coordinate axes at the current origin. This should be helpful in debugging your program.

- To get you started, we're also going to give you a function that you might use to build the left upper-arm of your robot. This should get you thinking about what things you need to keep track of. Note that you could also have a generic constructUpperArm routine that gets run once for the left arm and once for the right; this approach requires a little more thinking but will end up requiring half as much code. Either approach will get full credit.

```
void construct_left_arm(void) {

  // Set up a display list
  TOBOR_LEFT_ARM_DL = glGenLists(1);
  glNewList(TOBOR_LEFT_ARM_DL,GL_COMPILE);

  // Set up the reference frame for the shoulder
  glTranslatef(1.1,0.25,0.0);
  glScalef(0.5,0.5,0.5);

  // Draw a gray sphere for the shoulder
  SetMaterial(MAT_GRAY);
  glutSolidSphere(0.5,20,20);

  // Walk down the y axis and draw a scaled cube
  // for the upper arm
  glTranslatef(0.0,-1.10,0.0);
  glScalef(0.5,1.5,0.5);
  SetMaterial(MAT_WHITE);
  glutSolidCube(1.0);
  glEndList();

}
```

Extra credit opportunities abound as usual; contact us with questions…

Part 2 – RUBIX:

Now that you understand how to apply transformations to objects, it's time to apply your knowledge to something more interesting. Rubik's Cube has been one of the world's most popular toys since it's creation in 1974. However, kids today only want to play computer games, so as your get-rich-quick scheme, you're going to write a program that allows you to play with Rubik's Cube in OpenGL!

However, we don't want anyone to get sued, so instead we will create "RubiX", a variation where instead of having six different colored sides, there are three different colored blocks, and the RubiX Cube is "solved" when all the colors are lined up in three rows (when you start up the demo, it is in a "solved" state).

The keys you need to implement are:

- '1','2','3' Select a segment to rotate
- 'q','Q' Rotate forward and backward around the x-axis
- 'w,'W' Rotate forward and backward around the y-axis
- 'e','E' Rotate forward and backward around the z-axis
- 'a','z','s','x' Rotate the whole cube so you can see different parts of it
- 'r' Reset the cube to its initial 'solved' state

You should animate the rotation like the demo does. You can ignore keystrokes that arrive while a segment is busy rotating.

The starter code lives in rubix.cpp; it's almost identical to the TOBOR starter code, you have the same support code, and you need to fill in the same functions. You do not need to use display lists for this part of the assignment.

Tips

- We've given you a data structure that you might use to store your blocks (it simply stores a color right now), and we've given you a 3x3x3 array of Blocks. When you initialize your world, you should initialize the array so the blocks are solve. I.e. all nine blocks in blocks[0][y][z] are the same color, all nine blocks in [1][y][z] are the same colors, and so forth...

- You will need to store the currently-selected segment (based on whether the user has pressed '1', '2', or '3') and – while you're

- This assignment will be *much* easier if – after a rotation is finished animating – your re-arrange the blocks so there are no rotations any more; i.e. I should just be able to look at your blocks[][][] array to figure out which colors are where. We recommend that you start by thinking about rotating a segment around the z axis, and writing down on paper which block ends up where after a 90 degree rotation. Also note that you only have to figure this logic out

for rotation in one direction, since you can apply it three times to get the same effect as rotating once in the opposite direction.

- We recommend drawing each cube as a glutSolidCube() with side length of 0.9; you shouldn't worry too much about rendering for this part of the assignment.

- A great extra credit option – worth 10 points – is to offer a key that sends your cube into a random state and animates the solution from that state. This is not as tricky as it seems; you don't need to know anything about how to solve a Rubick's Cube to do it…

Deliverables

All submissions are due by 11:59 p.m. on the date specified.

You must include a README. The README should document what your program does, any bugs the TA should be aware of, any extra credit you have implemented and any additional information you think would aid the TA in grading your project.

You should submit your program electronically using the submit script. First, while on a Linux machine (**remember, your project must run on Linux!),** you should go to the directory where your code resides. Be sure to remove any executables or .o files as we will compile it ourselves when we grade it. Then you should run:

```
/usr/class/cs148/bin/submit
```

while in the directory. This script will ask you a few questions to make sure all is well in the universe, then it will copy all of your code files, your makefile, and your README and put them in the submissions directory to be graded. The submit script should list all the files submitted. We will be checking time stamps and using this information to track late hours.

Please be aware that your projects in this course will be graded primarily on their output. We will grade your projects by running them through various tests. Because of the number of projects the TA must grade, it will not be possible for them to analyze your programs to determine why they failed on a particular test case. Your best bet is to thoroughly test your program, and compare your output to the sample solutions.