[whistle]

Change the default light source to illuminate your scene more dramatically.

Allow for adjustable levels of detail for your character. You will need to add a UI control for this.

Come up with another whistle and implement it.  A whistle is something that extends the use of one of the things you are already doing.  It is part of the basic model construction, but extended or cloned and modified in an interesting way.

Implement the "Hitchcock Effect" described in class, where the camera zooms in on an object, whilst at the same time pulling away from it (the effect can also be reversed--zoom out and pull in). The transformation should fix one plane in the scene--show this plane. Make sure that the effect is dramatic--adding an interesting background will help, otherwise it can be really difficult to tell if it's being done correctly.

[bell]

Use a texture map on all or part of your character. (The safest way to do this is to implement your own primitives inside your model file that do texture mapping.)

Build a complex shape as a set of polygonal faces, using the "triangle" primitive to render them.  Examples of things that don't count as complex: a pentagon, a square, a circle. Examples of what does count:[dodecahedron](http://mathworld.wolfram.com/Dodecahedron.html), 2D function plot (z = sin(x2 + y)), etc.

Make an additional "animated" sequence your character can perform.  Although you can try to use a timed callback "add\_idle" (see [here](http://www.fltk.org/documentation.php) for more information), an easier solution is just to increment values each time your model's draw() function is called. If you use the menu option to turn on animation, your draw() function will be executed at around 30 times per second.

Add some widgets that control adjustable parameters to your model so that you can create individual-looking instances of your character.  Try to make these actually different individuals, not just "the red guy" and "the blue guy."

Add a "Frame All" feature that would change the position of the camera such that your model is entirely visible in the model window, regardless of the pose and location of your model.

[bell+whistle]The camera code has a constrained up vector -- modify the code to (1) handle camera twists (the framework is already there) and (2) to come up with an intuitive way for the user to twist the camera via mouse control.

A **display list** is a "recording" of OpenGL calls that gets stored on the graphics card. Thus, display lists allow you to render complicated polygons much more quickly because you only have to tell the graphics card to replay the list of commands instead of sending them across the (slow) computer bus. A display list tutorial can be found [here](http://www.lighthouse3d.com/opengl/displaylists/).

Implement a smooth curve functionality. Examples of smooth curves are [here](http://www.cs.washington.edu/education/courses/cse457/02sp/lectures/param-curves.pdf). These curves are a great way to lead into swept surfaces (see below). Functional curves will need to be demonstrated in some way. One great example would be to draw some polynomial across a curve that you define. Students who implement swept surfaces will ***not*** be given a bell for smooth curves. That bell will be included in the swept surfaces bell. Smooth curves will be an important part of the animator project, so this will give you a leg up on that.

Implement one or more non-linear transformations applied to a triangle mesh. This entails creating at least one function that is applied across a mesh with specified parameters. For example, you could generate a triangulated sphere and apply a function to a sphere at a specified point that modifies the mesh based on the distance of each point from a given axis or origin. Credit varies depending on the complexity of the transformation(s) and/or whether you provide user controls (e.g., sliders) to modify parameters.

[Heightfields](https://www.cs.washington.edu/education/courses/cse457/11au/projects/modeler/heightfield.html) are great ways to build complicated looking maps and terrains pretty easily.

[bell][bell]Add a function in your model file for drawing a new type of primitive. The following examples will definitely garner two bells ; if you come up with your own primitive, you will be awarded one or two bells based on its coolness.

1) Torus.

2). Surfaces of rotation - given a curve and an axis, draw the surface that results from sweeping the curve around the axis. This is really nice for making pottery :).

3). Rail surfaces - see Watt, p. 41.

4). Swept surfaces (this is worth **3** bells) -- given two curves, sweep one profile curve along the path defined by the other. These are also known as "generalized cylinders" when the profile curve is closed. This isn't quite as simple as it may first sound, as it requires the profile curve to change its orientation as it sweeps over the path curve. See [this page](http://web.archive.org/web/20060601133523/www.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15462/web.all/assignments/assignment2/gcyl.html) for some uses of generalized cylinders. [This document](http://web.archive.org/web/20031222121818/http:/www-2.cs.cmu.edu/afs/cs.cmu.edu/academic/class/15462/web.00s/assts/asst2/asst2.html) may be helpful as well, or see the parametric surfaces lecture from a previous offering of this class.

(Variable) Use some sort of procedural modeling (such as an [L-system](http://vlab.infotech.monash.edu.au/tutorials/l-systems/)) to generate all or part of your character. Have parameters of the procedural modeler controllable by the user via control widgets.

In addition to mood cycling, have your character react differently to UI controls depending on what mood they are in.  Again, there is some weight in this item because the character reactions are supposed to make sense in a story telling way.  Think about the mood that the character is in, think about the things that you might want the character to do, and then provide a means for expressing and controlling those actions.

[bell][bell][bell][bell]One difficulty with hierarchical modeling using primitives is the difficulty of building "organic" shapes. It's difficult, for instance, to make a convincing looking human arm because you can't really show the bending of the skin and bulging of the muscle using cylinders and spheres. There has, however, been success in building organic shapes using [metaballs](http://en.wikipedia.org/wiki/Metaballs). Implement your hierarchical model and "skin" it with metaballs. Hint: look up "marching cubes" and "marching tetrahedra" --these are two commonly used algorithms for volume rendering. Here is a [demo application](https://course.cse.ust.hk/comp4411/Password_Only/projects/modeler/metaballs_demo.zip) by Joe Hall and Eugene Hsu that shows metaballs in action.

If you have a sufficiently complex model, you'll soon realize what a pain it is to have to play with all the sliders to pose your character correctly. Implement a method of adjusting the joint angles, etc., directly though the viewport. For instance, clicking on the shoulder of a human model might select it and activate a sphere around the joint. Click-dragging the sphere then should rotate the shoulder joint intuitively. For the elbow joint, however, a sphere would be quite unintuitive, as the elbow can only rotate about one axis. For ideas, you may want to play with the Maya 3D modeling/animation package, which is installed on the workstations in 228. Credit depends on quality of implementation.

Another method to build organic shapes is [subdivision surfaces](http://mrl.nyu.edu/publications/subdiv-course2000/). Implement these for use in your model. You may want to visit [this](http://www.cs.washington.edu/education/courses/557/99sp/projects/) to get some starter code.

[bell][bell][bell][bell][bell][bell][bell][bell]

**Inverse kinematics**

You might notice after building your model that it's difficult to have very "goal-oriented" motion. Given a model of a human, for instance, if the goal is to move the hand to a certain coordinate, we might have to change the shoulder angle, elbow angle -- maybe even the angle of the knees if the feet are constrained to one position. Implement a method, given a set of position constraints like

left foot is at (1,0,2)  
right foot is at (3,0,4)  
left hand is at (7,8,2)

that computes the intermediate angles necessary such that all constrains are satisfied (or, if the constraints can not be satisfied, the square of the distance violations is minimized). For an additional [bell] [bell] [bell] [bell] 4 bells, make sure that all angle constraints are satisfied as well. In your model, for instance, you might have a slider that constrains the elbow angle between 30 and 180 degrees.  Look [here](https://course.cse.ust.hk/comp4411/Password_Only/projects/modeler/inverse-kinematics.pdf) for some related material.

The hierarchical model that you created is controlled by forward kinematics; that is, the positions of the parts vary as a function of joint angles. More mathematically stated, the *positions* of the joints are computed as a function of the *degrees of freedom* (these DOFs are most often rotations). The problem of inverse kinematics is to determine the DOFs of a model to satisfy a set of positional constraints, subject to the DOF constraints of the model (a knee on a human model, for instance, should not bend backwards).

This is a significantly harder problem than forward kinematics. Aside from the complicated math involved, many inverse kinematics problems do not have unique solutions. Imagine a human model, with the feet constrained to the ground. Now we wish to place the hand, say, about five feet off the ground. We need to figure out the value of every joint angle in the body to achieve the desired pose. Clearly, there are an infinite number of solutions. Which one is "best"?

Now imagine that we wish to place the hand 15 feet off the ground. It's fairly unlikely that a realistic human model can do this with its feet still planted on the ground. But inverse kinematics must provide a good solution anyway. How is a good solution defined?

Your solver should be fully general and not rely on your specific model (although you can assume that the degrees of freedom are all rotational). Additionally, you should modify your user interface to allow interactive control of your model though the inverse kinematics solver. The solver should run quickly enough to respond to mouse movement.

### View-dependent adaptive polygon meshes

The primitives that you are using in your model are all built from simple two dimensional polygons. That's how most everything is handled in the OpenGL graphics world. Everything ends up getting reduced to triangles.

Building a highly detailed polygonal model often requires millions of triangles. This can be a huge burden on the graphics hardware. One approach to alleviating this problem is to draw the model using varying levels of detail. In the modeler application, this can be done by specifying the quality (poor, low, medium, high). This unfortunately is a fairly hacky solution to a more general problem.

First, implement a method for controlling the level of detail of an arbitrary polygonal model. You will probably want to devise some way of representing the model in a file. Ideally, you should not need to load the entire file into memory if you're drawing a low-detail representation.

Now the question arises: how much detail do we need to make a visually nice image? This depends on a lot of factors. Farther objects can be drawn with fewer polygons, since they're smaller on screen. See Hugues Hoppe's work on [View-dependent refinement of progressive meshes](http://research.microsoft.com/~hoppe/#vdrpm) for some cool demos of this. Implement this or a similar method, making sure that your user interface supplies enough information to demonstrate the benefits of using your method. There are many other criteria to consider that you may want to use, such as lighting and shading (dark objects require less detail than light ones; objects with matte finishes require less detail than shiny objects).

### Hierarchical models from polygon meshes

Many 3D models come in the form of static polygon meshes. That is, all the geometry is there, but there is no inherent hierarchy. These models may come from various sources, for instance [3D scans](http://www.cyberware.com/wb-vrml/index.html). Implement a system to easily give the model some sort of hierarchical structure. This may be through the user interface, or perhaps by fitting an model with a known hierarchical structure to the polygon mesh (see [this](http://www.merl.com/papers/docs/TR2000-13.pdf) for one way you might do this). If you choose to have a manual user interface, it should be very intuitive.

Through your implementation, you should be able to specify how the deformations at the joints should be done. On a model of a human, for instance, a bending elbow should result in the appropriate deformation of the mesh around the elbow (and, if you're really ambitious, some bulging in the biceps).