CPSC 426
Assignment 2 Coding: Free-Form Deformations (FFD)

Due in class, Feb 28, 2014

In this assignment, you will animate deformations of triangle meshes using *free-form deformations*, as presented in class. The assignment implements a $3 \times 3 \times 3$ lattice of control points, which implements second order Bezier deformations.

Question 0

Experiment with the solution binary, `gosoln`. Run from the command line without any arguments, this will read in the file `scene.txt`, which is also the same as `bunny.txt`. Experiment with the following keystroke commands:
`m`: toggles through three mesh display modes: wire, solid, shaded.
`l`: toggles through three FFD lattice display modes
`p`: print the current location of the control vertices
`PgUp, PgDown`: zooms in and out (use `fnUp` and `fnDown` on Mac)
`arrows`: rotate object
`0,1,3,4`: rotate through different animation **modes**
  `0`: show binding frame
  `1`: linear interpolation of control points between first and second keyframes
  `3`: simple physics-simulation (coordinated movement of control pts)
  `4`: simple physics-simulation (independent movement of control pts)

Experiment with loading other scenes:
`gosoln car.txt`: car model; uses keyframes that are different than the bindframe (mode 1)
`gosoln ear.txt`: lattice that bends the bunny ear (mode 1)
`gosoln bulge.txt`: animated bulging of bunny using a moving lattice with rebinding (mode 2) `gosoln plane.txt`: plane model
`gosoln cube.txt`: simple cube model, useful for debugging

Download the template code, then read it and understand it.

- GraphicsMath.h and GraphicsMath.cpp define the two classes `Vector3` (represents a 3D vector), and `Affine3` (represents an affine 3D tranformation –or equivalently a 3D coordinate frame– via a 4x4 matrix using homogeneous coordinates). They readily support most common linear algebra operations. You are not expected to modify these files, but feel free to do it if it is useful for you. Below is some example code:

```
Vector3 v(1.0, 2.5, -1.0); // 3D vector
v1[0] = v2[2]; // Assign the Z-coord of v2 to the X-coord of v1
v3 = v1 + 2*v2; // Vector addition and scalar multiplication
Affine3 frame(S, T, U, P0); // Coordinate frame defined by basis
                            // vectors S, T, U and origin P0
                            // Equivalently, an affine transform.
v2 = transform * v1; // Apply an affine transformation to a vector
TransInv = T.inverse(); // Compute the inverse transformation
```

- TriangleMesh.h and TriangleMesh.cpp define the class `TriangleMesh`. A mesh is made of vertices (`mesh.vertices`), and triangles (`mesh.triangles`), where each triangle stores the indices of the three vertices that defines it. You are not expected to modify these files either, but feel free to do so. The vertices and triangles can be loaded from an OBJ file, or/and programatically created/modified. For example:

```
TriangleMesh mesh("meshes/bunny.obj"); // Creates a mesh from OBJ file
mesh.vertex[157] = Vector3(12.0,2.5,-2.3); // Modify the 158th vertex
mesh.vertex[157][1] = 2.5; // Modify the Y-coord of the 158th vertex
```

- FreeFormDeformation.h and FreeFormDeformation.cpp define the class `LatticePoints` and FFD-related methods that are either provided or that you have to implement.

- main.cpp uses the above classes to display and deform meshes. Hitting the numeric keys 1 through 4 sets the variable `mode` to be the corresponding integer, and calls `initScene()` that initializes the global variables based on the chosen scenario. The variable `time`, in seconds, automatically cycles from 0 to `maxTime` (that you can initialize in `initScene()`). At each time step, the method `updateScene()` is automatically called, followed by `drawScene()`.

Question 1

[3 marks] Using the solution binary, modify `plane.txt` in order to:
(a) Make the model twice as tall, i.e., scale vertically by a factyor of two. This can be accomplished using the affine transformations associated with the keyframes.
(b) Further modify the affine transformations to have the plane fly forwards and backwards instead of up and down (mode 1)
(c) Now modify tyhe FFD control point offsets to have it flap its wings up and down (mode 1).

Question 2

[6 marks] Download and compile the template code. Create a data structure for storing the s,t,u FFD coordinates for each vertex. Note that `mesh.vertices.size()` gives the number of vertices. Working with `cube.txt` will simplify testing. In *main.cpp*, implement the method `computeFFDCoords(mesh,bindframe)` and use it to compute and save the the FFD coordinates for all mesh vertices. Note that this method will be called each time a mode key is pressed.

Next, implement `computeDeformedMesh()` in `main.cpp`. This should update the value of all the mesh vertices. Use second order Bezier basis functions. Test with `cube.txt` and then the more complex scenarios (mode 1).

Question 3

[3 marks] Implement a moving FFD lattice with dynamic rebinding to the original mesh. This is used to implement bulge seen in `bulge.txt` (mode 2). Make a copy of original mesh, using a global variable, in `initScene()`. Change `updateScene()` to recompute the FFD coordinates based on the original mesh and the `currentBindFrame`.

Question 4 (optional)

[up to 5 marks] Implement any of: (a) Implement a simple scene of your own using only two keyframes; (b) Implement a simple physical simulation of the control points, similar to mode 3 or 4; (c) Extend the system to handle multiple keyframes, using either piecewise linear or Catmull-Rom interpolation of control points; (d) Extend the system to allow multiple static models to be loaded in addition to the animated model. (e) Extend the system to allow multiple animated models.

Note that you will also have an opportunity in the last assignment to develop your own ideas by building on the code used for the assignments.

Hand-in Instructions

- Create a folder called `a2` under your cs426 directory and put all the source files and the `README.txt` file there.
- In the `README.txt` file, please provide your name and describe what functionalities you have implemented, as well as any kind of information you would like to give us for getting credit for partial implementation. If you dont complete all the requirements, please state clearly what you have tried, what problems you are having and what you think might be promising solutions. If you are using external sources, provide clear attribution.
- The assignment should be handed in with the exact command:

  ```
  handin cs426 a2
  ```

  This will handin your entire a2 directory tree by making a copy of your a2 directory, and deleting all subdirectories! (If you want to know more about this handin command, use: `man handin`)

Assignment grading

The assignment will be graded with face-to-face demos: you will demo your program for the TA. If your assignment is incomplete, you can concisely summarize your explanations and what you were trying to do in your README. You can also explain any extra credit features you implemented.

- The demo sessions times will be determined closer to submission date.
- You must ensure that your program compiles and runs on whatever medium you intend to demo on (laptop, lab computer, etc.). The face to face grading time slots are short, you will not have time to do any quick fixes ! If your code as handed in does not run during the grading session, your grade will directly reflect this.
- The code that you demo must match exactly what you submitted electronically: the TA will ask to see a long listing of the files that youre using in order to quickly verify that the file timestamps are before the submission deadline.
- Arrive at ICICS/CS 005 at least 10 minutes before your scheduled session. Double-check that your code compiles and runs properly.
- When the TA comes to your computer, run your assignment and go through the different provided scenarios. If you are shooting for the bonus points, show the grader the extra features and/or scenarios you created.