CMPUT 411/511—Computer Graphics

Assignment 3

Fall 2012
Department of Computing Science
University of Alberta

Due: 23:59:59 *Tuesday, November 13*
Worth: 20% of final grade
Instructor: Dale Schuurmans, Ath409, x2-4806, dale@cs.ualberta.ca

---

In this assignment you will implement a combined mesh-motion viewer (in C++/OpenGL). Your program will first read in a description of a 3D mesh model of a person and a 3D skeleton (which has been properly located inside the mesh). Next it will compute the skin attachment weights that connect each mesh vertex to bones in the skeleton. Your program will then read a motion capture sequence and animate the mesh model using smooth deformation. Finally, you will add simple ambient lighting and smooth shading to the surface triangles to make the animated model appear solid.

Note When submitting your program, you need to include a `Makefile` that allows the TA to simply run "`make`" and have your program compile properly on the lab machines. The TA will then run the executable "`./personviewer` ⟨meshfile.*obj*⟩ ⟨motionfile.*bvh*⟩" to test your program on the mesh stored in ⟨meshfile.obj⟩ and skeleton and motion sequence stored in ⟨motionfile.bvh⟩

When finished, please submit a *single* .zip archive containing your files on **eclass**.

Note A set of .obj files containing person mesh models and a set of .bvh files containing examples of human motion capture sequences are posted on the assignment web page in a3files.zip.

Note You can re-use your code from the previous assignments, or even code from previous solutions.

Implementing a viewer for motion capture sequences

You will implement a mesh-motion capture viewer, with executable called `personviewer`, that reads in a .obj file describing a person mesh model, and a .bvh file describing a simplified skeleton that has been embedded inside the mesh. The .bvh file will also provide an animation sequence for the skeleton. Your goal will be to first connect the mesh to the skeleton, then animate the mesh, and finally make the person appear solid.

Marks are given for the functionalities achieved by your viewer.

1. (1%) **Read in and print out valid .obj and .bvh files**

   Your program needs to accept two arguments that name a .obj and .bvh file to read. In particular, your program will be invoked with a command "`./personviewer` ⟨meshfile.*obj*⟩ ⟨motionfile.*bvh*⟩" where ⟨meshfile.obj⟩ is the name of the mesh file and ⟨motionfile.bvh⟩ is the name of the motion file to be read.

   It is very useful to have a simple output function for debugging and marking purposes. Therefore, when the user types the character 'w', your program should write to the output files `meshout.obj` and `motionout.bvh`. In particular, your program should write out a valid .obj file that represents the mesh model, and a valid .bvh file that represents the skeletal model and motion sequence that were read.

   Note The .obj file will contain 'vn' lines, which indicate vertex normals. Since you will be using vertex normals later in this assignment, you should also include these in the output.obj file.

2. **Attach the mesh to the skeleton**

   (a) (3%) ***Compute and print out a simplified skin-bone connection matrix***
   Compute a sparse matrix S such that

$$S_{ij} = \begin{cases} 1 & \text{if } j \text{ is closest bone to vertex } i \\ 0 & \text{otherwise} \end{cases}$$

   If there is tie between k bones for the closest to vertex i, then the row $S_{i:}$ should have $1/k$ in each of the closest bone columns.

   When the user types 'w', your program should write the S matrix to the file `S.out`. The output format should be: each line corresponds to one vertex (with vertices considered in order), where each line begins with the vertex number followed by a (space separated) list of the bone numbers the vertex is connected to.

   Note Throughout this assignment, the vertices should be ordered the same as the order they appear in the .obj file. The bones are ordered by the depth first traversal through the skeleton in the .bvh file.

   Note You should use Eigen package for sparse matrix operations and solving the linear system in Part (e) below. The Eigen package will be installed in the lab, but it is open source and you can get started with it yourself. Please refer to `http://eigen.tuxfamily.org/index.php?title=Main_Page` for more details.

(b) (3%) ***Compute and print out a skin to* visible-*bone connection matrix***
Compute a sparse matrix C such that

$$C_{ij} = \begin{cases} 1 & \text{if } j \text{ is closest } \textit{visible} \text{ bone to vertex } i \\ 0 & \text{otherwise} \end{cases}$$

If there is tie between k bones for the closest to vertex i, then the row $C_{i:}$ should have $1/k$ in each of the closest bone columns.

Note The exact condition for a nontrivial intersection between a line segment and a triangle is given by: $\beta \geq 0$, $\gamma \geq 0$, $\beta + \gamma \leq 1$ and $0 < \lambda < 1$. (That is, treat the triangle as a closed set but the line segment as open.)

When the user types 'w', your program should write the C matrix to the file `C.out`. The output format should be: each line corresponds to one vertex (with vertices considered in order), where each line begins with the vertex number followed by a (space separated) list of the bone numbers the vertex is connected to.

(c) (1%) ***Compute and print out the vector of mesh vertex importances***
Let d_{ij} denote the minimum distance from vertex i to bone j. Let $\textit{visible}(i)$ denote the set of bones that are visible from vertex i. Compute a vector $\mathbf{h}$ such that

$$h_i = \frac{1}{d_i^2} \qquad \text{where} \qquad d_i = \begin{cases} \min\limits_{j \in \textit{visible}(i)} d_{ij} & \text{if } \textit{visible}(i) \neq \emptyset \\ \infty & \text{otherwise} \end{cases}.$$

When the user types 'w', your program should write the h vector to the file `h.out`. The output format should be: each line corresponds to one vertex (with vertices considered in order), where each line begins with the vertex number followed by the corresponding h value.

(d) (1%) ***Compute and print out the Laplacian matrix for the mesh graph***
Compute a sparse adjacency matrix A, then compute the sparse Laplacian matrix L such that $L = \Delta(A\mathbf{1}) - A$.

When the user types 'w', your program should write L to the file `L.out`. The output format should be: each line corresponds to one vertex (with vertices considered in order), where each line begins with the vertex number i followed by a list of (column,value) pairs describing the nonzeros entries in the row $L_{i:}$.

(e) (2%) ***Compute and print out the skin-bone attachment matrix***
Set $\beta = 1$, then compute the skin-bone attachment weights W by solving the linear system for each bone j

$$(\beta L + \Delta(\mathbf{h}))W_{:j} = \Delta(\mathbf{h})C_{:j}.$$

When the user types 'w', your program should write the W matrix to the file `W.out`. The output format should be: each line corresponds to one vertex (with vertices considered in order), where each line begins with the vertex number i followed by a list of the bone attachment weights for each bone j (with bones considered in order).

3. **Display and animate the mesh and skeleton**

   (a) (1%) ***Display the initial pose and place the camera in a good viewing position and orientation***

   Use a perspective camera model. Define an initial view frustum that captures the entire figure in the camera's field of view throughout the entire motion sequence, keeping the figure large enough to remain recognizable. This will require you to do some simple pre-analysis of the mesh, skeleton and the animation sequence. Make sure that the camera-up direction coincides with the figure-up direction.

   Display the mesh and skeleton in the default initial pose, drawing the figure in white against a black background.

   (b) (1%) ***Add camera control***

   When the user types:
   ⟨leftarrow⟩ or ⟨rightarrow⟩: translate the camera -0.1 or 0.1 units in the camera x direction (dolly),
   ⟨downarrow⟩ or ⟨uparrow⟩: translate the camera -0.1 or 0.1 units in the camera y direction (crane),
   'i' or 'I': translate the camera -0.1 or 0.1 units in the camera z direction (zoom),
   't' or 'T': rotate the camera -10 or 10 degrees counterclockwise around the camera x axis (tilt),
   'a' or 'A': rotate the camera -10 or 10 degrees counterclockwise around the camera y axis (pan),
   'c' or 'C': rotate the camera -10 or 10 degrees counterclockwise around the camera z axis (roll).

   (c) (2%) ***Animate the mesh using simple linear deformation***

   Animate the mesh using simple linear deformation (using the weights in W) to relocate each mesh vertex. In particular, when the user types:
   'p': play the animation, running in a continuous loop that wraps around at the end of the sequence back to the beginning (use a default animation speed of 120 frames per second),
   'P': pause the animation,
   's': stop the animation, restore and redisplay the figure in its initial configuration,
   'q': exit the program.

   (d) (2%) ***Animate the mesh using combined linear and spherical (bend and twist) deformation***

   Use the same animation controls as above, but when the user types:
   'B': use the combined linear and spherical (bend and twist) deformation,
   'b': use simple linear deformation.

4. **Add light and shading to display and animate a solid figure**

   Finally, to move beyond displaying wire meshes, you will add simple ambient lighting and smooth shading to the surface triangles to make the animated model appear solid. To do so, you will need to define a light source, define material properties for the triangles, and enable the vertex-normals.

   (a) (2%) ***Add an ambient light source and use smooth shading to show a solid figure***

   When the user types:
   'L': display the figure as a smoothly shaded, solid object under ambient lighting,
   'l': display the figure as a wire frame mesh.

   (b) (1%) ***Animate the solid figure***

   Incorporate the 'L' and 'l' controls with the animation.