CS6533: Interactive Computer Graphics
Fur and Bump Mapping

11/12/2013

The goal of this assignment is to draw a furry bunny, and perform some simple dynamics simulation on the fur of the bunny. Fur rendering is achieved by rendering multiple translucent layers of the same geometry, which we call *shells*, but with different per-vertex positional offsets.

You will start from the last assignment and add additional material and scene graph nodes to the code. The skeleton code for the assignment contains updated files that you should use to replace the ones you already have. It also contains new texture file `shell.ppm` and shaders for rendering the fur. After you have copied over all the files, make sure to do a `make clean` (under Linux/MacOSX) or a Build | Clean Solution (under Visual Studio) to remove old compiled files.

Note that this assignment may take some time to complete, so you are advised to start a day earlier than normal.

Preparation

You should follow `asst8-snippets.cpp` to modify your main source file. Be sure to read the comments there and understand them. The modifications contained in the file will insert code for creating new material, geometry, and scene graph nodes for drawing the bunny and its furs. Also there is a new GLUT call back function for "special keys" (`specialKeyboard`), which you use to handle the keyboard's arrow keys. Finally global variables are declared that will help you with doing dynamics simulation later.

Task 1: Rendering Straight Fur (4 points)

A furry surface is drawn in two stages.

- First the surface itself is drawn (This is the first shape node added as a child to `g_bunnyNode`).
- Next, shells are drawn using the `bunny-shell-*` shaders. You can see from the snippets that we we add one child shape node for each shell to `g_bunnyNode`). The ith shape nodes uses the material `g_bunyShellMats[i]`, which has the right opengl blending modes set.

To define the shells, first imagine a single *straight hair* starting at a vertex positioned at $\mathbf{p}$, and made up of m straight segments going in the direction of the unit normal $\hat{\mathbf{N}}$ at that vertex, ending at the straight tip $\mathbf{s}$, as in Figure 1.

$$\mathbf{s} = \mathbf{p} + (\texttt{g\_furHeight} \cdot \hat{\mathbf{N}}) \tag{1}$$

(Note, that $\hat{\mathbf{N}}$ is the "smooth" normal computed at each vertex, not one of the faces "flat" normals.)

A tunable parameter `g_furHeight` defines the total length of the fur: $\|\mathbf{s} - \mathbf{p}\|$. Let $\mathbf{n} = (\mathbf{s} - \mathbf{p})/m$. Then this particular straight hair defines m *shell vertex positions*, $\mathbf{p} + i\mathbf{n}$ for $i = 1, \ldots, m$. The ith shell is then formed by taking the original geometry of the surface, but moving each vertex to its ith shell vertex position.

The code already loads and renders the bunny, but the shells are not shown since their corresponding geometries have not yet been specified. You need to first finish the `updateShellGeometry()` function to specify the shell geometries (`g_bunnyShellGeometries[]`), and then call it at appropriate places to specify the shell geometries.

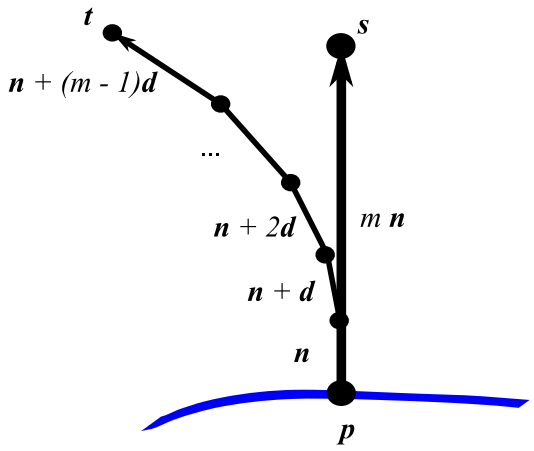

Figure 1: Combing hair: hair-tip at rest is **s**, but we will bend the hair from the root (**p**) to get to the current hair tip (**t**). Note that **n is not the unit normal**, just proportional to it ($\mathbf{n} = (\texttt{g\_furHeight}/m) \cdot \hat{\mathbf{N}}$, where $\hat{\mathbf{N}}$ is the vertex normal at **p**). Note also that the hair length at rest (`g_furHeight`) is $\|m \cdot \mathbf{n}\| = m\|\mathbf{n}\|$, which is a free (adjustable) parameter of your model. You choose m, the number of "segments" that the hair is split into (a value of 15-32 is usually sufficient). Given a unit normal at **p**, and a value for m, you can compute **n** such that $m\|\mathbf{n}\|$ is the desired length of the hair at rest. Then given **p**, **s**, **t**, m, and **n**, there is a single **d** that makes the hair bend to get to the desired hair tip **t**.

You also need to initialize the hair tip positions (`g_tipPos[]`) to their rest positions in `initSimulation()`.

Note that each item in the array `g_bunnyShellGeometries` is of type `shared_ptr<SimpleGeometryPNX>`. Compared with the `SimpleGeometryPNTBX` that we used in previous assignments, each vertex (of type `VertexPNX`) in the new geometry carries only position, normal, and a set of texture coordinates.

- When specifying a `VertexPNX`, since we are first rendering straight furs, you can pass in **n** defined above as the normal.
- The texture coordinates of such a `VertexPNX` are used to lookup the shell texture. Each triangle in the current shell will access the same portion of the texture, so they will use the same texture coordinates. You may start mapping each mesh triangle to the unit isosceles triangle in texture coordinates.

Since we are drawing the shells with alpha-blending, it is important that triangles are drawn in a back-to-front order, (otherwise the alpha blending formula does not make much sense). Sorting all the triangles is a very expensive operation, but since the bunny is a mostly closed surface, we can take the following shortcut: Always draw one entire layer of the shell at a time, and do so for each layer, going from the inner-most layer to the outer-most one. This ensures that things are drawn in roughly the right order.

In our scene graph, children of a transform nodes will always be drawn in the order they are inserted, so you want to make sure that when you specify the geometries, `g_bunnyShellGeometries[0]` corresponds to the inner most shell.

You will use the `GLUT_UP_KEY` and `GLUT_DOWN_KEY` to change the density of the fur (controlled by a variable called `g_hairyness`). You will implement this by changing the texture coordinates for the triangle. Simply makes each triangle's texture coordinates be (0, 0), (`g_hairyness`, 0), and (0, `g_hairyness`). Because we set the textures to "wrap", you can use coordinates outside of the 0..1 range. You will use the `GLUT_LEFT_KEY` and `GLUT_RIGHT_KEY` to change the length of the fur.

Task 2: Animating Straight Fur (4 points)

We next want to animate the fur by simulating dynamics. We need to change our definition of the *shell vertex positions* a little. Again consider a single hair starting from a vertex positioned at $\mathbf{p}$. We associate an additional "hair tip" variable $\mathbf{t}$ with each hair (and hence each mesh vertex). The hair is now defined as the straight line going from $\mathbf{p}$ to $\mathbf{t}$. Similarly the shell vertex positions are now points on this straight line, i.e., you should replace $\mathbf{s}$ with $\mathbf{t}$ in the previous definition of the shell vertex positions.

At the beginning of the program, $\mathbf{t}$ is initialized to $\mathbf{s}$ (defined in Task 1), which can be interpreted as the position of the hair tip at rest. Subsequently, $\mathbf{t}$ is pushed around by the combination of two forces: a gravity force pulling $\mathbf{t}$ down, and a spring force pulling $\mathbf{t}$ back toward the at-rest position $\mathbf{s}$. Since the hair has a fixed length, we also constrain $\|\mathbf{t} - \mathbf{p}\| = \texttt{g\_furHeight}$.

More concretely, $\mathbf{t}$ is updated via simple dynamics simulation based on the *Euler method*. You need to store a velocity vector $\mathbf{v}$ with each tip, initialized to $\mathbf{0}$. Let T be a "time step" constant. Then each *step* of the dynamics simulation is performed as:

1. Compute the total force $\mathbf{f}$ acting on the tip. The total force $\mathbf{f}$ is the sum of a gravity force $(0, -g, 0)$, where g is some constant, and a spring force. The spring force will be in the direction of the normalized $(\mathbf{s}-\mathbf{t})$ vector, and has the magnitude $(\|\mathbf{s}-\mathbf{t}\|\cdot\text{stiffness})$. Thus the spring force is simply $((\mathbf{s}-\mathbf{t})\cdot\text{stiffness})$
2. Update the tip position: $\mathbf{t} \leftarrow \mathbf{t} + T\mathbf{v}$.
3. Constrain $\mathbf{t}$ so that the length $\|\mathbf{t}-\mathbf{p}\| = \texttt{g\_furHeight}$. This is achieved by setting $\mathbf{t} \leftarrow \mathbf{p}+\texttt{g\_furHeight}\cdot(\mathbf{t}-\mathbf{p})/\|\mathbf{t}-\mathbf{p}\|$.
4. Update the velocity: $\mathbf{v} \leftarrow (\mathbf{v} + T\mathbf{f}) \cdot \text{damping}$. The damping parameter will be some number slightly less than 1.0.

You will perform the above update for each hair inside the `hairsSimulationCallback` function.

Note that if the time step T (called `g_timeStep` in the code) is too big, the simulation might become unstable, i.e., the tip position $\mathbf{t}$ will move around without ever coming to a rest. On the other hand, if a small T is used, and only one step of the update is performed inside the `hairsSimulationCallback` function, and then rendered, your fur might move too slowly. A simple solution is to use a small T, but perform multiple steps (controlled by `g_numStepsPerFrame`) of update within `hairsSimulationCallback` before rendering the result.

It is probably easier to do all the simulation in *world space*. In particular, the gravity force always points down with respect to the world frame; Similarly, the coordinate vectors representing $\mathbf{t}$ and $\mathbf{v}$ should be defined with respect to the world frame. When you rotate or translate the bunny's frame do not directly update $\mathbf{t}$. Instead, only update the "at-rest" hair tip $\mathbf{s}$ and the hair origin $\mathbf{p}$, (since we will consider these points are fixed with respect to the bunny's frame). Then, the spring force and the hair length constraint together will force the actual hair tip $\mathbf{t}$ to move toward $\mathbf{s}$ during the dynamics simulation. In the code, the world space coordinates of $\mathbf{t}$ should be stored in `g_tipPos[]`. The world space coordinates of the hair tip velocities $\mathbf{v}$ should be stored in `g_tipVelocity[]`.

On the other hand, when you specify the shells' geometry, you need to provide the positions and normals in object coordinates. Thus you may need to do some transformations between world and object coordinates as needed (Recall our friend `getAccumPathRbt(...)`, but DON'T call it for every vertex! Just call it once for the object and store the result for later use).

Task 3: Combing Hair (or Curvy Fur) (2 points)

Right now our fur looks more like little sticks coming out of the bunny. We next want to make the hairs curved so they look more natural. This requires us to make some further modification to the definition of shell vertex positions.

We will now think of the hair as a polyline with m segments that starts at $\mathbf{p}$, and ends at $\mathbf{t}$, as opposed to a straight line going from $\mathbf{p}$ to $\mathbf{t}$. Denote the shell vertex positions as $\mathbf{p}_1, \ldots, \mathbf{p}_m$. For convenience, also denote $\mathbf{p}_0 = \mathbf{p}$. We want the last shell vertex to coincide with the hair tip, so $\mathbf{p}_m = \mathbf{t}$ and we want the

displacement between subsequent $\mathbf{p}_i$'s to satisfy $\mathbf{p}_{i+1} - \mathbf{p}_i = \mathbf{n} + i\mathbf{d}$ for some constant vector $\mathbf{d}$. You need to derive this vector $\mathbf{d}$ from the above relationships, which then defines the shell vertex points. Figure 1 illustrates these different quantities.

Note that when you render the ith shell ($i = 1, \ldots, m$), for each vertex you should now use $(\mathbf{p}_i - \mathbf{p}_{i-1})$ as the normal for the purpose of shading.

Task 4: Bump Mapping (3 points)

To make your pictures look even snazzier, let's add a bump mapping implementation.

Write Some GLSL

First of all, you have a texture `FieldstoneNormal.ppm` from assignment 6. Note that the 3 values making up its pixel will not be interpreted as RGB values, but rather as the 3 coordinates of a normal. If you refer to `initMaterials()`, you will see that this texture is bound to the `uTexNormal` uniformvariable of the `g_bumpFloorMat` material, which uses the shaders `normal-gl{2|3}.{f|v}shader`.

You need to change the fragment shader `normal-gl{2|3}.fshader` (depending on your `g_Gl2Compatible` setting) to read the appropriate pixel from the texture, transform it to eye space, and use it for shading calculation. This will give your geometry a high resolution look.

Texture Coordinates: each vertex will need (x, y) texture coordinates. We will build these in to our ground, cube, and sphere geometry objects. These attributes are passed into the vertex shaders for the draw call. You can access them as `vTexCoord` in the fragment shader.

Data range: The texture stores its pixel data as triplets (rgb) of real numbers between 0 and 1, while normal coordinates are triplets (xyz) of numbers in the range -1 to 1. Thus you need to apply a scale and then shift to the data before using it. The scale and shift should be chosen so that $0 \mapsto -1$ and $1 \mapsto 1$.

Yet more matrix stuff: We want to store our normal map data in such a way that we can use one data patch and tile it around a curvy surface (like a sphere or a mesh). As such, we don't want to represent the normal data in the texture as coordinates with respect to the world or object or joint or even bone frame.

For the sake of notation, let $\vec{\mathbf{b}}^t = \vec{\mathbf{e}}^t M$ be the not necessarily orthonormal frame associated with a "bone" of your object (say the lower arm), about to be drawn, and M the associated model view matrix. (In our case, the "bone" is just the ground).

We will let each vertex of this bone have its own "tangent frame" represented as $\vec{\mathbf{t}}^t = \vec{\mathbf{b}}^t T$. The data for T will be passed as three `vec3` vertex attribute variables: `aTangent`, `aBinormal`, and `aNormal`. These will represent the three columns making up the upper left 3 by 3 submatrix of T. Since we will be dealing with the coordinates of vectors and not points, we will not need any translational data in T. Our convention is that the data in our bump mapping texture (after scaling and shifting to the correct data range), $\mathbf{n} = [n_r, n_g, n_b, 0]^t$, expresses the normal wrt to the $\vec{\mathbf{t}}^t$ frame. Thus the tangent frame coordinates of the normal's coordinate vector are $T\mathbf{n}$ and its eye coordinates are $M^{-t}T\mathbf{n}$.

Since we will not get hold of the normal data until we get to the fragment shader, we will do the following: at the vertex shader we pass $M^{-t}T$ as a varying variable called `vNTMat` (for normal matrix times tangent frame matrix) to the fragment shader. Then at the fragment shader, you will multiply $\mathbf{n}$, the normal data fetched, by this matrix.

In the fragment shader if have $\mathbf{v} := [x_e, y_e, z_e, 0]^t$ the eye coordinates of a vector that we want to dot with a normal, we simply compute

$$\texttt{dot}(\texttt{normalize}(\texttt{vNTMat} * \mathbf{n}), \texttt{normalize}(\mathbf{v}))$$

Parameters

As you have probably noticed, this assignment involves a lot of tunable parameters, such as the spring stiffness constant, the gravity, and so on. Suitable constants will produce visually pleasing results. For your convenience, we already provide default values of these constants as part of `asst8-snippets.cpp`. They're listed below again for convenience.

- `g_numShells`: Number of shell layers, $m = 32$
- `g_furHeight` $= 0.21$
- `g_stiffness` $= 4$
- `g_damping` $= 0.96$
- `g_gravity` $= (0, -0.5, 0)$, or $g = 0.5$.
- `g_timeStep`: Time step $T = 0.02$
- `g_numStepsPerFrame`: Number of simulation steps within `hairsSimulationCallback()`: 10

Coding Notes

You might find the following things helpful as you write the codes:

- If you are working on windows: you can use the "Release mode" build to speed up things. If you're on Mac or Linux, do a `make clean` then followed by `make OPT=1` to get an optimized build that runs much faster. You only need to do the `make clean` once.
- The function `updateShellGeometry()` is potentially expensive, so you will want to call it only before rendering, instead of every time in your `hairsSimulationCallback()`, as `hairsSimulationCallback()` might be called at a much higher frequency than your render rates. One way to do so is to have a global variable `bool g_shellNeedsUpdate`. You always set it to true in `hairsSimulationCallback()`. Then in `drawStuff`, you call `updateShellGeometry()` only when `g_shellNeedsUpdate` is true. Finally, you reset it to false within `g_updateShellGeometry()`.
- The `normalize` implementation of `cvec` will assert upon zero-length vector. You might want to change it so that if the vector is near zero, nothing is done;