CMSC 23700 **Introduction to Computer Graphics** **Project 3**
Winter 2010 **February 5**

Animation and shadows

Due: Friday, February 19, 10pm

1 Summary

In this project you will implement skeletal-based character animation and shadow mapping. The project can logically be divided into three steps: skeletal-animation, skinning the skeleton with a mesh, and rendering shadows. Each of these steps is non-trivial, so you should start immediately.

2 Skeletal animation

Skeletal animation is a technique for animating meshes, such as those that represent creatures in a 3D game. The basic idea is that the model is defined by a *hierarchical skeleton*, which consists of a tree of joints, and one or more triangle meshes, or skins, which are attached to the skeleton. Each joint (except the root) has a parent joint, a position in its parent's coordinate space, and an orientation. Rather than directly animate the mesh, the animator animates the skeleton and the mesh follows the skeleton's motion.

An animation is specified as a sequence of *poses*, which represent the skeleton's position at various points in time. Each pose is a complete skeleton; rendering occurs by interpolating between poses, then computing the vertex positions of the skin(s) for the interpolated skeleton, and then drawing the mesh.

3 Animating the skeleton

The file `guard.c` contains the definition of an animated character.[1] In order to render the skeleton, you will first have to compute an interpolated set of joints. This is done by linearly interpolating the positions and spherically interpolating the orientations between the previous keyframe and the next keyframe. The sample code includes a function `DrawSkeleton` that will draw a skeleton as a stick figure (you will have to provide the shader for it).

[1] Normally, such definitions are loaded from files.

4 Skinning the model

Instead of specifying the position of the vertices, we compute them from the positions of the joints (after interpolation). For each vertex V, there are n_V weights $W_1, \ldots, W_{n_V}$. The position of V is then defined by

$$\mathrm{Pos}(V) = \sum_{i=1}^{n_V} \mathrm{Pos}(W_i)$$

where the position of a weight W with associated joint J is defined by

$$\mathrm{Pos}(W) = b_W(\mathrm{Pos}(J) + \mathbf{R}_J \mathbf{v}_W)$$

where $\mathrm{Pos}(J)$ is the interpolated position of J, $\mathbf{R}_J$ is the interpolated orientation of J, b_W is W's bias, and $\mathbf{v}_W$ is W'w position in J's coordinate space. In the sample code, $\mathbf{R}_J$ is represented as a unit quaternion; the file `quat.h` provides operations on quaternions, such as spherical interpolation and transforming a vector by a quaternion.

5 Shadows

Shadows are one of the most important visual cues for understanding the relationship of objects in a 3D scene. As discussed in class, there are a number of techniques that can be used to render shadows using OpenGL. For this project, we will use shadow mapping.

The idea is to compute a map (*i.e.*, texture) that identifies which screen locations are shadowed with respect to a given light. We do this by rendering the scene from the light's point of view into the depth buffer. Then we copy the buffer into a *depth texture* that is used when rendering the scene. When rendering the scene, we compute a $\langle s, t, r \rangle$ texture coordinate for a point $\mathbf{p}$, which corresponds to the coordinates of that point in the light's clipping space. The r value represents the depth of $\mathbf{p}$ in the light's view, which is compared against the value stored at $\langle s, t \rangle$ in the depth texture. If r is greater than the texture value, then $\mathbf{p}$ is shadowed. To implement this technique, we must construct a transformation that maps eye-space coordinates to the light's clip-space (see Figure 1). Let

- $\mathbf{M}_{\mathit{model}}$ be the model matrix
- $\mathbf{M}_{\mathit{view}}$ be the camera's view matrix
- $\mathbf{M}_{\mathit{light}}$ be the light's view matrix, and
- $\mathbf{P}_{\mathit{light}}$ be the light's projection matrix.

A vertex $\mathbf{p}$ that has been transformed by OpenGL's model-view matrix will be $\mathbf{M}_{\mathit{view}}\mathbf{M}_{\mathit{model}}\mathbf{p}$. To transform this point to the light's homogeneous clip space, we first apply the inverse view matrix ($\mathbf{M}_{\mathit{view}}^{-1}$), then the light's view matrix ($\mathbf{M}_{\mathit{light}}$), and finally the light's projection matrix ($\mathbf{P}_{\mathit{light}}$). After the perspective division, the coordinate will be in the range $[-1 \ldots 1]$, but we need values in the range $[0 \ldots 1]$ to index the depth texture, so we add a scaling transformation ($\mathrm{S}(0.5)$) and a translation ($\mathrm{T}(0.5, 0.5, 0.5)$). The final matrix is

$$\mathbf{M}_{\mathit{tex}} = \mathrm{T}(0.5, 0.5, 0.5)\mathrm{S}(0.5)\mathbf{P}_{\mathit{light}}\mathbf{M}_{\mathit{light}}\mathbf{M}_{\mathit{view}}^{-1}$$

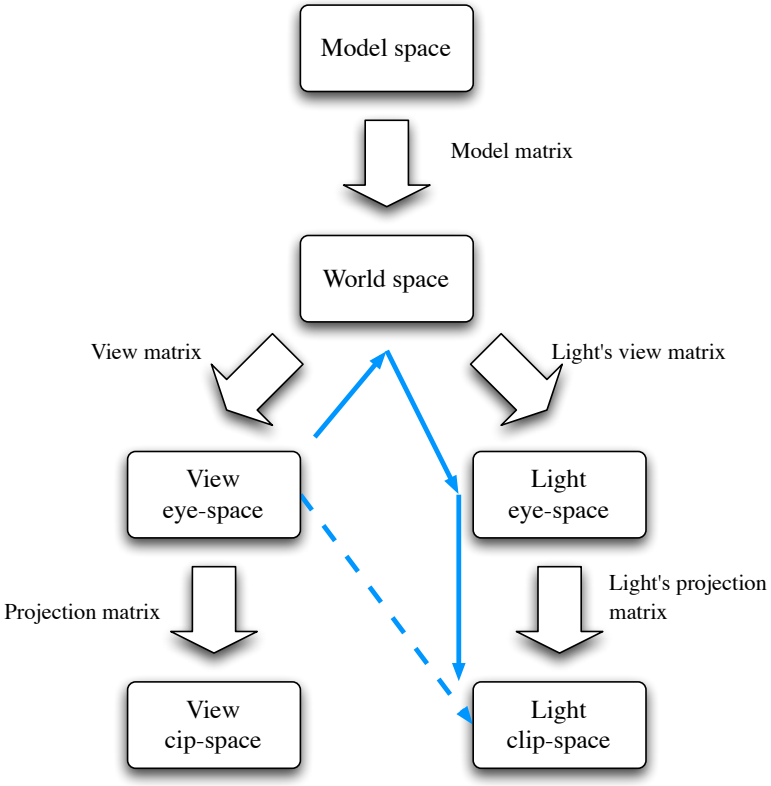

Figure 1: The coordinate-space transforms required for shadow mapping

Multiplying this matrix by the transformed point gives us

$$\begin{aligned}\mathbf{p}' &= \mathbf{M}_{tex}\mathbf{M}_{view}\mathbf{M}_{model}\mathbf{p} \\ &= \mathrm{T}(0.5, 0.5, 0.5)\mathrm{S}(0.5)\mathbf{P}_{light}\mathbf{M}_{light}\mathbf{p}\end{aligned}$$

The sample code defines a fixed directional light l. You will need to compute the ambient (A_l) and diffuse (D_l) intensity vectors using the techniques of Project 2. You should also compute a value S_l, which is 0.5 when the pixel is in shadow and 1.0 otherwise. Then, assuming that the input color is C_{in}, A is the global ambient light level, and $\mathcal{L}$ is the set of enabled lights, the resulting color should be

$$C_{\text{out}} = C_{\text{in}} \, \mathbf{max}\left(A, \, \mathbf{clamp}\left(\sum_{l \in \mathcal{L}} S_l(A_l + D_l)\right)\right)$$

5.1 User interface

The sample code supports an interface for manipulating the view using either the keyboard or mouse. In addition, you will add code to handle the following key command:

`m` toggle mesh mode between no mesh, wireframe, and mesh

6 Extra credit

Some (but not all) of the skins have specular and bump map textures. If the color texture for a skin is in the file `"foo.png"`, then the specular map is in `"foo_s.png"` and the bump map is in `"foo_h.png"`. For extra credit, you may implement specular highlights and bump mapping when rendering the model's mesh. Note that the specular maps are three-channel maps and thus do not specify an exponent.

7 Hints

Break the project into stages; get each stage working before starting on the next.

1. get the skeleton animated,
2. compute the mesh and render it as a wireframe,
3. add the textures for to the mesh,
4. render the shadow buffers, and
5. add shadowing to the rendering of the model.

Getting the shadowing to work is difficult, so you should try to get the other parts done by the end of the first week so you have plenty of time for the last two steps.

Because the light is directional, you will need to use an orthographic projection when rendering the shadow map. Also, since its position is fixed, you can compute its model-view and projection matrices as startup time. You can also precompute the texture matrix needed to map eye-space vertices to the light's clipping space.

You may find it useful to render the shadow map to the screen as a debugging aid. One way to do this is to create a second window that is the shadowmap size. You can map the depth values to a grey scale (i.e., 0 maps to black and 1 maps to white) using a simple shader. The GLUT library supports multiple windows (or you can dump the shadow map to a file).