CSCI 441 - Lab 14
Friday, December 02, 2016
LAB IS DUE BY **SUNDAY DECEMBER 11 11:59 PM**!!

Today, we'll display more complex MD5 models which implement skeleton animation and skinning. We'll use VBOs to display the model. This lab was based off of the code provided at: http://tfc.duke.free.fr/coding/md5-specs-en.html.

That website also has an explanation of how MD5Mesh and MD5Anim files work. Each bone is represented as a quaternion, which stores the axis of rotation and angle in a four-component vector. The focus of this lab is not to understand how quaternions or skeletal animation works. Instead we'll make use of VBOs.

You will need to modify two files: `main.cpp` and `src/md5mesh.cpp`.

Please answer the questions as you go inside your `README.txt` file.

Step 1 – Drawing our MD5 Model

Skeleton animation works by traversing our skeleton tree. Each bone is connected to a parent. Each bone also has a rotation and orientation involved. Therefore, the orientation of a parent bone effects all child bones. The first step in drawing our model is to compute the orientation of all bones. This is an exercise in hierarchical modeling and transformations (and is done for you in `md5mesh.cpp:ReadMD5Model()` when the file is first read in since it is a static pose. If an animation is applied, then `md5anim.cpp:Animate()` and `InterpolateSkeletons()` computes the newly positioned skeleton.) .

We can verify the output at this point. Compile and run the program.

The first time, run the program with the following arguments:

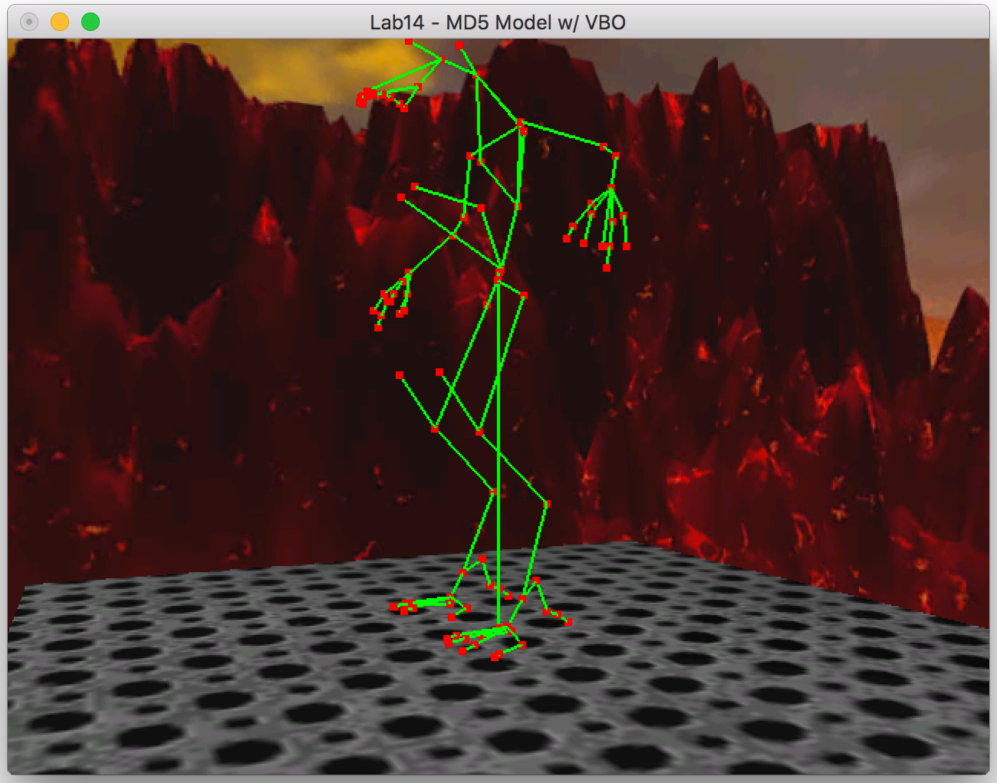
```
./lab14 models/monsters/hellknight/mesh/hellknight.md5mesh
```

This specifies which mesh to use.

When the program is finished loading, press 'b' to display the skeleton of the model. This is the base pose of the skeleton. Now run it a second time with the arguments:

```
./lab14   models/monsters/hellknight/mesh/hellknight.md5mesh
models/monsters/hellknight/animations/idle2.md5anim
```

This now applies an animation to the skeleton. Pressing 'b' again, you will see the skeleton move. You should see the following image.



Once the bones are placed, the skin must be placed on top of the bones. Each vertex of the skin is comprised of a number of weights dependent upon various bones. The skinning is done in the function `PrepareMesh()` in `md5mesh.cpp`. For each vertex, the weighted sum of every bone is added together to compute the final vertex position. Therefore, as bones are rotated and moved the skin will move with the bones based off of the weights and joint positions.

This code will go into `md5mesh.cpp`.

We want to be able to see more than just the skeleton. At `TODO #1` in `md5mesh.cpp` we need to set up values in our vertex array to be used in our pseudo-VBO. (Disclaimer: this is not a true VBO implementation but is in the same vain. This is the implementation we discussed in class.) The array is already created, `vertexArray`. We now need to set each element in our array equal to the computed final vertex. The `for` loop goes through each vertex, so we want to assign the final vertex to the `i`th element in our array. The `vec3_t` type is a typedef for a float array, so we will need to do element by element assignment into our array.

Now that the mesh is prepared and our vertex array has the values set, let's draw! At `TODO #2`, the first thing we must do is enable the client state for our `GL_VERTEX_ARRAY`. Then at `TODO #3`, we'll pass in our vertex array with `glVertexPointer()`. This function takes four arguments:

1. The number of elements in the array that comprise a vertex (we are using x, y, & z)
2. The data type of the elements in the array (they are floats)
3. The stride between vertices in the array (they are consecutive, so 0)
4. And then finally the array itself

Now we have to actually pass the data to the GPU and tell it how to draw. We'll use the function `glDrawElements()` which takes a number of arguments:

1. The primitive to draw. We want to draw triangles
2. The number of vertices in the array. We can access this in a number of ways from the mesh, but we'll take the number of triangles times 3. We can get the number of triangles off the mesh by using `mesh->num_tris`
3. The data type of the indices we are passing in. They are unsigned integers.
4. The index array itself which is stored in `vertexIndices`

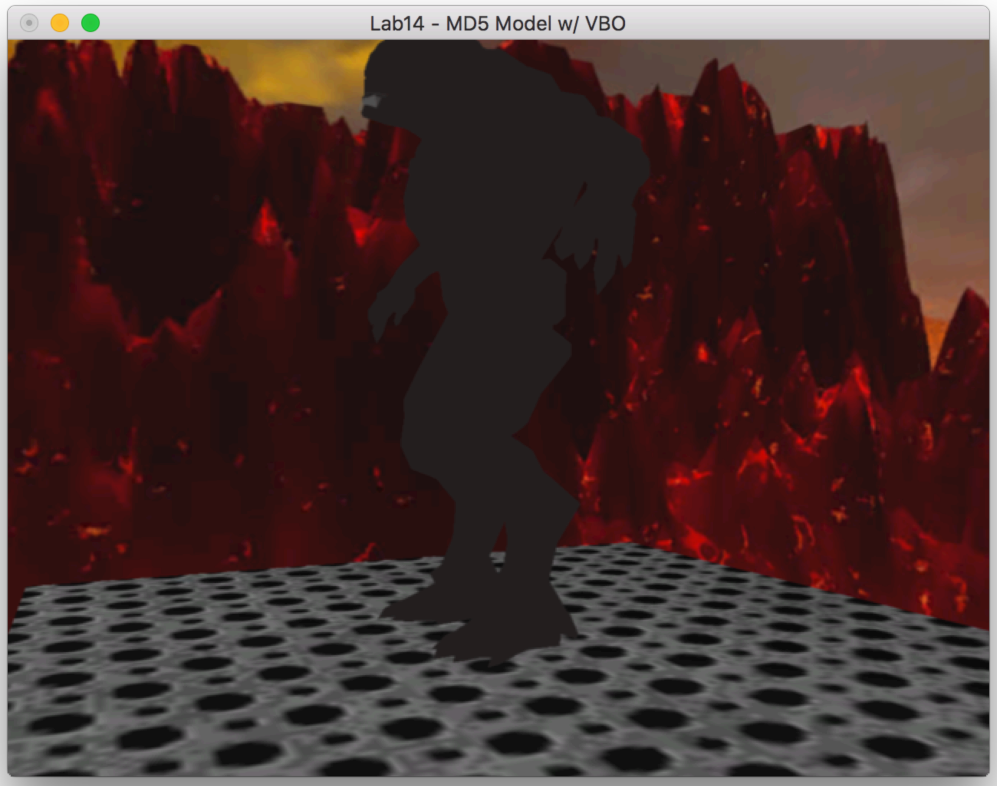
Recall from class, this index array states how to combine the vertices in the triangle. So `vertexIndices[0]`, `vertexIndices[1]`, `vertexIndices[2]` are three indices of our `vertexArray`. The GPU internally makes a triangle out of the following vertex array elements:

1. `vertexArray[ vertexIndices[0] ]`
2. `vertexArray[ vertexIndices[1] ]`
3. `vertexArray[ vertexIndices[2] ]`

and so forth.

Lastly at `TODO #4`, disable the client state for the vertex array.

Compile and run. You should now see the silhouette of our Hellknight. If you press the 'm' key, this will display just the mesh and you can see the full outline of every triangle.



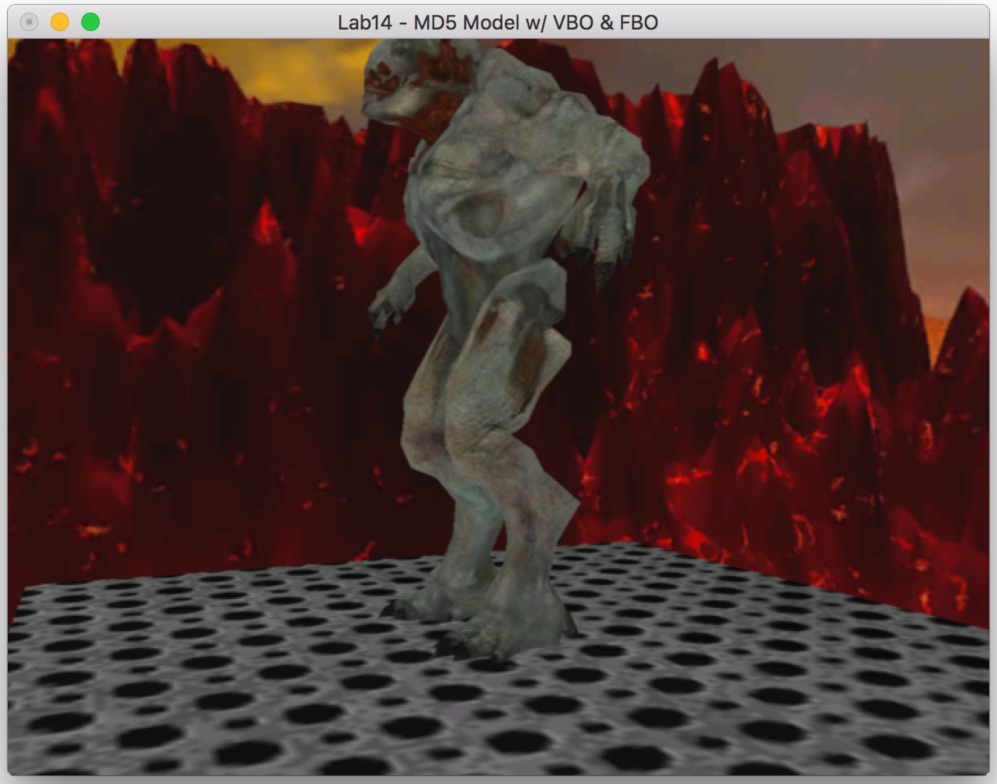
Well, now our shadowy figure is moving around but let's add some texture. At `TODO #5` pass the mesh vertex's texture coordinate into our texture array. The texture array is already created and called `texelArray`. You can access the texture coordinate for the `i`th vertex with the command `mesh->vertices[i].st`. And again, this is a two element array so needs to be assigned element by element.

Now we'll pass our texel array to the GPU. First, at `TODO #2` again, enable the client state for `GL_TEXTURE_COORD_ARRAY`. After our `glVertexPointer()` in `TODO #3`, pass in our texel array with `glTexCoordPointer()`. This takes four arguments:

1. The number of elements in the array that comprise a texture coordinate (we are using only s & t)
2. The data type of the elements in the array (they are floats)
3. The stride between texture coordinates in the array (they are consecutive, so 0)
4. And then finally the array itself

Lastly, disable the client state after the drawing is completed at `TODO #4`.

Compile and run. Your Hellknight should be properly placed in Hell now. The MD5 shader provided only does diffuse texture mapping. The model comes with a normal map, specular map, and height map. It is left as a further exercise to modify the MD5 model and shader to implement these extra features (no extra credit achievement for this, just for coolness factor). You will notice the textures are already bound in `DrawMesh()` and the uniforms would be bound in `setupShaders()`.



Step 2 – Adding in an FBO and BlurIt!

This code will go in `main.cpp`.

On the resources page there is a document about setting up and using a Framebuffer Object (https://blackboard.mines.edu/bbcswebdav/courses/Fall.2016.CSCI441A/docs/FBOs.pdf). Use that document to setup your FBO at `TODO #6`. Create global variables as necessary. Use a framebuffer size of `1024x1024`. You will go up to the part that checks the status of the FBO.

Now, at `TODO #7`, bind your FBO and render your scene to it. Don't forget to unbind it at `TODO #8`.

Continue on with the document to render the texture to a 2D quad filling the screen at `TODO #9`.

Compile and run. Your scene should like it had before and there should be no discernable difference.

Lastly, when you render the 2D quad of your texture, we'll apply the blur shader to this quad. I have provided the completed blur shader for you and you may have seen it gets loaded in already. What you need to do is use the blur program and pass two uniform variables to it.

The first is the size of our framebuffer (it is square so can use the width or height).

The second is our blur size. There is a global variable `BLUR_SIZE` that tracks this information.

Don't forget to turn the shader program off after we're done rendering our quad.

Compile and run. Now, pressing '+' should make the scene blurrier and pressing '-' should make it less blurry.

Congrats! Now you can display cool MD5 models. You may have trouble finding them online, but you can make your own in Blender! Oh, and more importantly we can use FBOs to do postprocessing effects.

Q1: Was this lab fun? 1-10 (1 least fun, 10 most fun)
Q2: How was the write-up for the lab? Too much hand holding? Too thorough? Too vague? Just right?
Q3: How long did this lab take you?
Q4: Any other comments?

To submit this lab, zip together your source code, `Makefile`, screenshot, and `README.txt` with questions. Name the zip file `<HeroName>_L14.zip`. Upload this on to Blackboard under the Lab14 section.

LAB IS DUE BY **SUNDAY DECEMBER 11 11:59 PM**!!