CSCI 441 - Lab 03
Friday, September 18, 2020
LAB IS DUE BY **FRIDAY SEPTEMBER 25 11:59 PM**!!

Today, we'll display more complex MD5 models which implement skeleton animation and skinning. We'll use VBOs and VAOs 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 and VAOs.

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

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

Step 0 – Setup CMakeLists.txt & CLion

This project has two `CMakeLists.txt` files. The first is located at the top level and the second is located inside of the `src/` folder. Be sure to update the `include_directories()` and `link_directories()` appropriately in each file.

Additionally, in CLion be sure to set the Run > Edit Configurations > Working Directory to the parent .. for the lab03 target.

There are technically two targets in this lab (`lab03` and `md5model`). Upon completion of this lab, you will have a stand alone MD5 Library that you can compile future programs against. For the lab, be sure to always be building the `lab03` target, as it will internally build and link against the `md5model` library.

Step 1 – Creating the Static Platform using VAO & VBO

Start off by building the existing code and run it. Woah! There's nothing there. Well if you want a little easter egg of what is coming, press the 's' key for a teaser. We'll need to complete a number of steps before we compile and test our code so get ready.

We ultimately will place a Hellknight into this world. Before we get there, we need to create a platform for the Hellknight to stand on. We must use VBOs to store our data. We'll need to store all of this data in an array and then send it to the GPU. We will be walking through the steps presented in the VAO Checklist.

Let's create a structure to hold the attributes for these vertices that make up our platform. Find `TODO #01` in `main.cpp`. We're going to put all of our buffer object code together (with one exception). Create a `struct`, we'll call it `VertexColored`, and it should contain a total of 6 floats to represent the x, y, z position and r, g, b color components. This is setting us up to make an interleaved array as we'll be keeping all of the vertex attributes together in memory.

Now that the struct is set up, at `TODO #02` go ahead and create an array of type `VertexColored`. It should have four elements and contain the following vertex attributes (be sure to enter them in the correct order).

Position	Color
(-10, 0, -10)	(1, 0, 0)
(10, 0, -10)	(0, 1, 0)
(-10, 0, 10)	(0, 0, 1)
(10, 0, 10)	(1, 1, 1)

We'll also need to tell the GPU how to connect the vertices to form triangles. Decide if you will be using triangles or a triangle strip to draw the quad later on. At `TODO #03` you'll need to create an array, of type `unsigned short`, of either 6 or 4 elements and the proper indices to form a quad.

All our data is set up, now it's time to send it to the GPU. We'll need to create a global variable to store our Vertex Array Object descriptor. Put this at `TODO #04A`. Now jump back to `TODO #04B` and generate a vertex array descriptor using `glGenVertexArrays()`. You can verify this step worked if your vaod is non-zero. Then bind it to make it active using `glBindVertexArray()`.

Recall that a VAO is used to encapsulate VBOs. We just made the VAO, let's make the VBOs now. We'll need to create a Vertex Buffer Object descriptor using `glGenBuffers()` and then bind it to the `GL_ARRAY_BUFFER` using `glBindBuffer()`. This goes at `TODO #05A`. Likewise, you'll know this step worked if your vbod is non-zero.

The VBO is set up, now to transfer the vertex data from the CPU over to the GPU. At `TODO #05B`, we'll make a call to `glBufferData()`. This method takes four arguments:

1. The target buffer. The vertex attributes go to `GL_ARRAY_BUFFER`, the same target we bound our VBO to.
2. The array size. Using `sizeof()`, pass the size of your vertex array.
3. The actual data. Pass your vertex array.
4. Hint to usage. Our platform will be continually drawn and always drawn in the same place. The data will never change so it will be static. The appropriate usage hint is `GL_STATIC_DRAW`.

At this point, the vertex data is now on the GPU. We need to tell the GPU where each attribute is located within the array. We'll do the vertex position first at `TODO #06A`. First, we need to enable the vertex position connection using the following command

`glEnableVertexAttribArray( vertexPositionLocation )`

We'll need to replace *`vertexPositionLocation`* with the value `Lab03BlackMagic::SHADER_ATTRIBUTES.vertexPosition` – this is an existing variable that is populated using Lab03BlackMagic and we'll discuss it next week – this is our hand wavy part.

Next make a call to `glVertexAttribPointer()` and provide the following six arguments:

1. The location connection. This will be the same value you just used for the enable step.
2. The number of elements that make up the attribute. The vertex position consists of an x, y, and z component so this argument should be 3.
3. The data type of the attribute. The position is made up of floats so this should be `GL_FLOAT`.
4. If the data should be normalized or not. It shouldn't. Pass in `GL_FALSE`.

5. The stride between elements. What is the amount of memory that should be skipped over in between elements? We specified our data was interleaved, so we need to step over an entire vertex worth of data to get to the next one. Each vertex is of type `VertexColored` – the struct we created – so this argument should be `sizeof( VertexColored )`.
6. The offset pointer to where the first attribute is. The first position is at the beginning of the array so the pointer to the beginning is `(void*)0`.

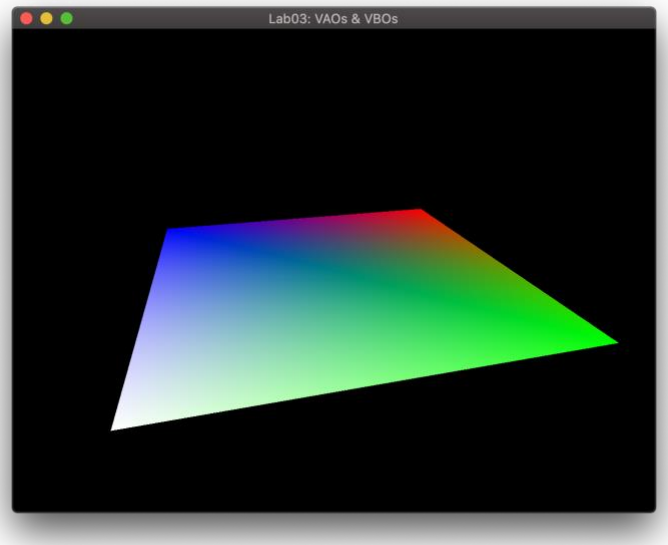
We need to repeat the same process for the color values as well. At `TODO #06B`, repeat the steps from `TODO #06A` for `Lab03BlackMagic::SHADER_ATTRIBUTES.vertexColor` in place of `vertexPosition`. The color information is made up of 3 float elements and the first color is stored in the 4th position of the array. The offset is then `(void*)(sizeof(float) * 3)`.

One last step for setup! At `TODO #07`, repeat steps `TODO #05A` & `05B` to send the index array to the GPU. This time the VBO target will be `GL_ELEMENT_ARRAY_BUFFER`. Again, the usage will be `GL_STATIC_DRAW`. Huzzah! Set up is done.

And now the moment of truth. Let's draw the platform. You'll find `TODO #08` in the `renderScene()` function. Bind our VAO that corresponds to the platform we created. We need to bind it because elsewhere in our program we're drawing other objects that have bound other VAOs. Finally, the actual draw call. We now need to use `glDrawElements()` and provide four arguments:

1. The primitive type. This should be either `GL_TRIANGLES` or `GL_TRIANGLE_STRIP` depending on how you previously created the index array.
2. The number of indices to process. This should be 6 or 4 depending on the value in the previous item.
3. The data type of index array. We previously set these up to be unsigned shorts so we should pass in `GL_UNSIGNED_SHORT`.
4. The offset to the first index. The indices start at the beginning, so this should be `(void*)0`.

We're ready to rock and roll. Run the program and drum roll……..if you see a floating platform like below



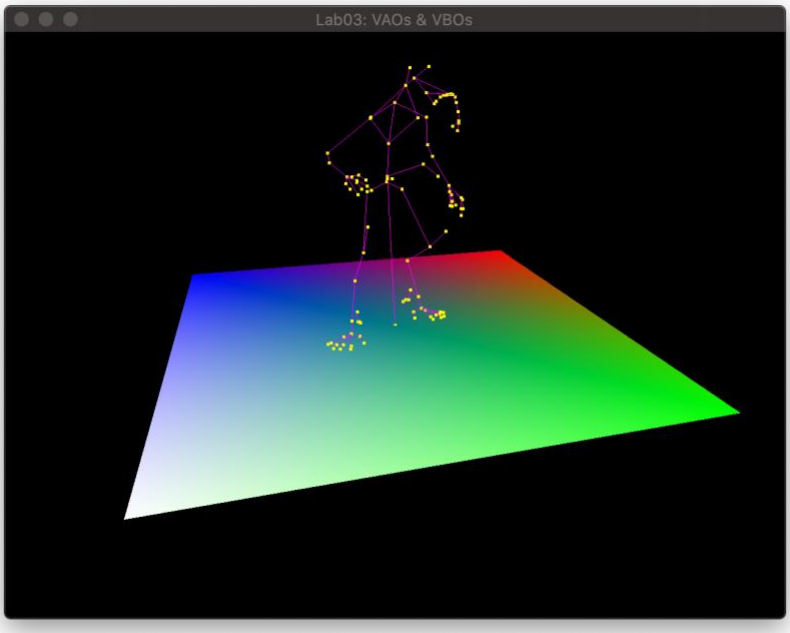
then your VAO & VBO are all properly set up! Time to perform the patented self high five. Raise your left hand over your head with your palm facing right. Then using your right hand, give your left hand a high five. Immediately after the ::clap:: sound, use your right hand to pat yourself on the back.

We've made a static VAO/VBO. Let's get our Hellknight onto the platform and create a dynamic VAO/VBO.

Step 2 – Drawing our dynamic 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::read_MD5_model()` when the file is first read in since it is a static pose. If an animation is applied, then `md5anim.cpp::animate()` and `interpolate_skeletons()` computes the newly positioned skeleton.) . The animation is time based, so you will see code checking the current time each frame and passing the delta time to the animate function.

We can verify what our skeleton looks like and how it dances. Rerun the program. When it is finished loading, press 's' to view the skeleton. 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 `prepare_mesh()` 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. The final vertex position of the skin mesh will be stored in the variable `vertexArray` and the corresponding texture coordinates will be stored in `texelArray`. Make note of these names as we'll be using them momentarily.

Find `TODO #09` in `md5mesh.cpp`. This next set of code will go in the `AllocVertexArrays()` function to set up the VAO & VBOs for the MD5 model. We'll need to go through the same steps we had done before to generate, bind, and populate our VAO and VBOs. At `TODO #09A`, create global variables to store the descriptor for our VAO and both VBOs that we'll need. For the platform, we only needed a

reference to our VAO, but since this object will be dynamic and data will be changing we need to keep a reference to our VBOs as well. A reminder of the steps to go through at `TODO #09`:

1. Generate vertex array
2. Bind vertex array
3. Generate vertex buffer
4. Bind `GL_ARRAY_BUFFER`
5. Buffer data to `GL_ARRAY_BUFFER`
    - This time we are going to set up the memory on the GPU slightly differently. We only want to allocate the maximum amount of memory we'll need but not actually send any data over yet. We'll be sending data every time we render so that will occur later. Therefore, when using `glBufferData()`, the arguments should be:
        i. The target. `GL_ARRAY_BUFFER`
        ii. The size. We have a variable available, `max_verts`, that stores the maximum number of vertices we may see in our model. Each vertex is made up of five floats (3 for position, 2 for texCood). The size is then `5 * sizeof(float) * max_verts`.
        iii. The data. We'll send this later. For now it should be `NULL`.
        iv. Usage hint. Since our data will be changing, we'll use `GL_DYNAMIC_DRAW` this time.
6. Connect vertex attribute pointers for `vPosAttribLoc` and `vTexCoordLAttribLoc`.
    - For this model, our data will be non-interleaved. Therefore, the stride we pass to `glVertexAttribPointer()` will be 0. This means that each attribute is located next to each other. We need to provide the proper offsets for each attribute.
        i. For `vPosAttribLoc`, the first position is located at the start of our array. The offset is then `(void*)0`.
        ii. For `vTexCoordLAttribLoc`, the first tex coord is located after all of the positions in our array. The offset needs to skip over `max_vert` number of positions and each position is made up of 3 floats. I'm not going to explicitly give you this one, but it looks very similar to 5.ii above with one minor change.
7. Generate vertex buffer for IBO
8. Bind `GL_ELEMENT_ARRAY_BUFFER`
9. Buffer data to `GL_ELEMENT_ARRAY_BUFFER`
    - Like the vertex attributes, we will not actually send the indices over just yet. Just allocate memory. We'll follow the same process as 5 above but use `GL_ELEMENT_ARRAY_BUFFER` for the target and a different size. The size of our indices will correspond to how many triangles we are drawing. This is stored in the variable `max_tris`. Each triangle is made up of three vertices and all our indices are stored as unsigned ints. Therefore the allocated size should be `sizeof(GLuint) * max_tris * 3`.

Great! The GPU has set aside memory to use when it receives the data later on. Let's send it the data and get to drawing. In `draw_mesh()` above at `TODO #10`, we'll send our data over. To make sure we are working with the right object, we first need to bind our VAO. And now this is where it is different for dynamic objects.

We had stated the beauty of the VAO is we could bind one object and all associated objects are bound and applied properly. That is true, for static objects. But when we have a dynamic object and we need to update memory on the GPU, we need to rebind the VBO that the data applies to. At this point, bind the VBO that corresponds to the MD5 model's array buffer. We're now ready to send the data of the skinned mesh to the GPU.

We'll use the `glBufferSubData()` function to send over our non-interleaved data in multiple chunks. This method takes four arguments:

1. The target. We're sending the vertex attribute data first, so this is again `GL_ARRAY_BUFFER`.
2. The offset to where this data should be placed within the array. We'll send our vertex positions first and this goes at the start. The offset is 0.
3. The size of data being sent. Our `mesh` contains `num_verts` and each vertex is made up of 3 floats. The total size is then `3 * sizeof( float ) * mesh->num_verts`.
4. The data itself. We mentioned before the vertex positions would be stored in `vertexArray`. Let's pass it through.

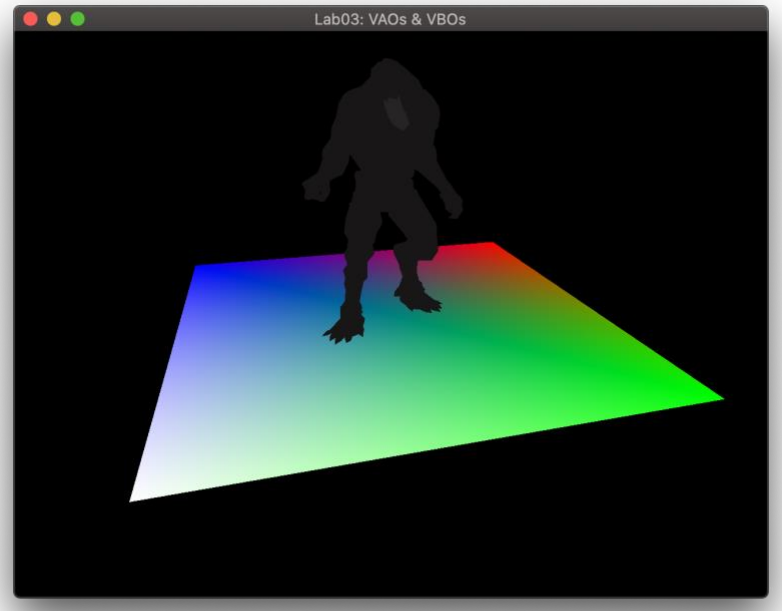
Next we need to bind the VBO corresponding to the index array. And we'll now buffer the data for our indices. This time for `glBufferSubData()` we'll pass:

1. Target (`GL_ELEMENT_ARRAY_BUFFER`)
2. Offset to where data should start. It starts at the beginning.
3. Size of data. Similar to when we allocated data, each triangle is made up of three vertices represented by unsigned ints. The number of triangles is stored on the mesh in num_tris. The size should look like `3 * sizeof( GLuint ) * mesh->num_tris`.
4. The data itself. The indices are stored in an array called `vertexIndices`.

We're ready to draw! Skip down to `TODO #11` and we'll create a call go `glDrawElements()`. Recall this method takes four arguments:

1. The primitive type. We want to use `GL_TRIANGLES`.
2. The number of indices to render. Our `mesh` contains `num_tris` and each triangle contains 3 indices which equals `mesh->num_tris * 3`.
3. The data type of our indices. They are unsigned ints so `GL_UNSIGNED_INT`.
4. The offset to where to start. The indices start at the beginning so `(void*)0`.

We're ready! Compile and run. Oooo spooky. We have a ghostly Hellknight standing on our platform.

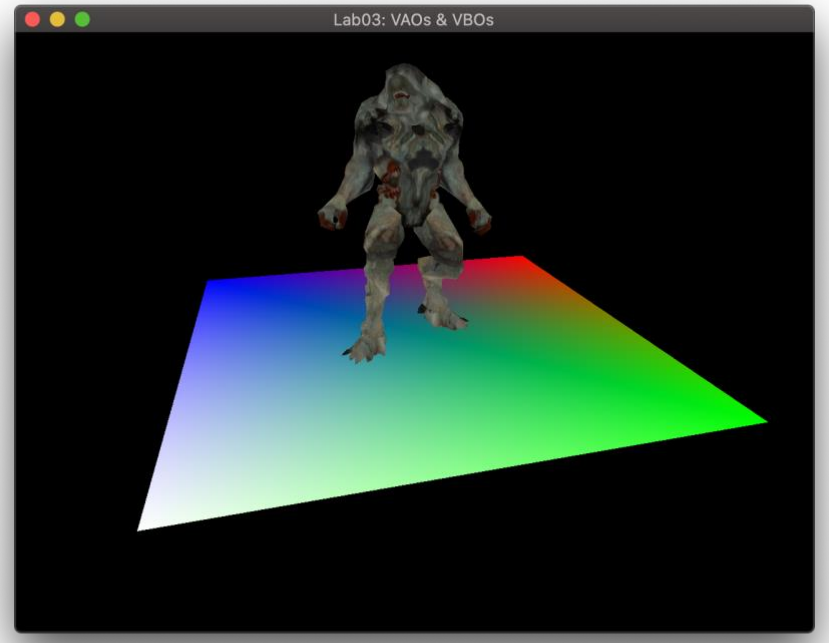


We only sent over the vertex positions, we did not send the texture coordinates. Let's give the Hellknight some new clothes. Back in the section for `TODO #10`, after we buffered the `vertexArray` data to

GL_ARRAY_BUFFER, we need to send one more piece of data. Add a second glBufferSubData() call for the target GL_ARRAY_BUFFER. This time the last 3 arguments will be slightly different:

1. The offset where to start. We stated this data was non-interleaved and the texture coordinates would be stored after all the vertex positions. We therefore need to skip over the maximum number of vertices we may have. This matches the offset we provided to the vertex attribute pointer for vTexCoordAttribLoc. We have max_verts of 3 floats each, so sizeof(float) * 3 * max_verts.
2. The size of data being sent. Our mesh is made up of num_verts and each vertex has 2 floats denoting the texture coordinate. Therefore the size is 2 * sizeof(float) * mesh->num_verts.
3. And the data. The texture coordinates are stored in the variable texelArray.

Let's do it. Compile. Run. And there he is! Our Hellknight is now in Hell.



Perfect! But one last thing. Let's be good little coders and clean up the memory we are using, both CPU side and GPU side. In FreeVertexArrays(), the CPU memory is already being returned. Let's return the GPU memory by calling glDeleteBuffers() and pass in all our VAO and VBOs for the MD5 model. Put these calls at TODO #12A. Now return to main.cpp and at TODO #12B, delete the VAO for our platform.

If you compile and run again, you won't see anything different. But you will sleep well knowing you do not have any GPU memory leaks.

Congrats on properly implementing Vertex Array Objects, Vertex Buffer Objects, and Index Buffer Objects. Next week we will remove the Black Magic curtain and create our own shader to render all of our objects.

Step 3 – Copy the md5model Library

If you would like to use the MD5 Model library in other labs/assignments/projects, then you now have a completed model library. Copy the contents of this lab's include/ folder to where your other OpenGL libraries are. (For instance Z:/CSCI441/include).

To copy the library file, you'll need to dig a little bit. Inside of `cmake-build-debug/src` you will find `libmd5model.a`. Copy this file to where your lib/ files are. (For instance Z:/CSCI441/lib).

Now, whenever you wish to use an MD5 Model you would `#include <MD5/md5model.h>` and then link against `md5model`.

You would need to follow the flow from this lab to actually use the model and you will need a shader, which we will talk about next week. But we have the model rendering in place.

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 all your source code and README.txt with questions. Name the zip file <HeroName>_L03.zip. Upload this on to Canvas under the L03 section.

LAB IS DUE BY **FRIDAY SEPTEMBER 25 11:59 PM**!!