Computer Graphics DT 3025
Lab 3

Stevan Tomic and Martin Magnusson, November 2015
Based on material by Mathias Broxvall

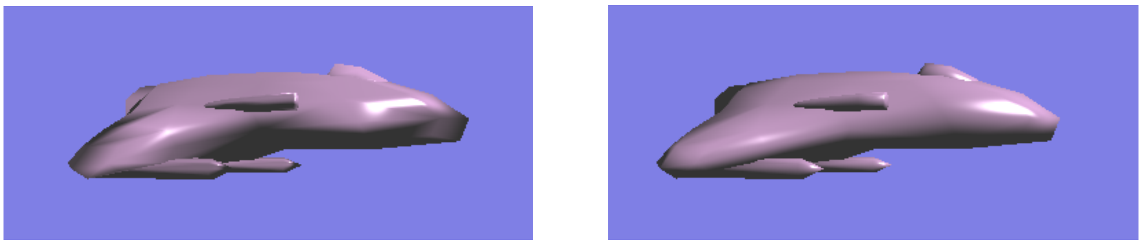

Figure 1: Fighter model file with flat normals (left) and with smooth normals (right).

Putting it all together

You will now continue by using what you have learned in the previous assignments to make some more realistic scenes than drawing a mere cube. For this purpose you will use a simple 3D fileformat (AC3D) and a custom reading function that should be enough for your purposes.

Exercise 3.1 - Drawing an object.
Use the new `glUtil.cc` file and use it to load the example model file `fighter.ac` in the initialization of your resources. You can do this by first instancing the `AC3DObjectFactory` class and using the `loadAC3D` function. The reason why we are using an instanced factory class for this will become apparent later on.

The factory will return to you a *tree* of AC3D objects corresponding to the hierachy that appears in the file. Note that in the given example file, *only the leaves* of this hierarchical tree of objects are interesting.

In your `doRedraw` code, traverse the tree (see code sketch in Algorithm 1) and extract the vertices and surface indices from the object in each leaf, and render your object using `glDrawElements` calls.

Note that you need to allocate a new array of suitable dimensions and put the indices of the primitive surfaces there. Since the object contains different types of primitives (triangles and quads), you need to extract separate sets of indices for both of them and to make two calls to the `glDrawElements` functions (see code sketch in Algorithm 2).

This part of the lab requires a fair amount of coding, just to get the vertices and indices populated correctly, before you get to the “actual graphics” part. However, such operations are ubiquitous in real-time graphics programming, so it makes sense to practice this kind of coding too.

Change your fragment shader code from above so that it can draw an object *something* when not given a normal.

Verify that you can see something on the screen with some form of silhouette.

Next, you need to create the normals for the object since these are not given as part of the AC3D file format.

Start by allocating space for *one normal per vertex* and initialize each normal to be zero.

Algorithm 1 Object tree traversal pseudocode

```
void World::modelRedraw(AC3DObject* node, Resources* r)
{
  if node has >0 children
  {
    // This is not a leaf node
    for (int i=0; i<numOfChildren; i++)
    {
      modelRedraw(node->children[i], r);
    }
  }
  else
  {
    // This is a leaf node
    modelRedrawLeaf(node);
  }
}
```

Next, loop over all surfaces and compute the normal over each surface.

$$\vec{N}_s = \text{normalize}(\vec{P}_{s,1} - \vec{P}_{s,0}) \times \text{normalize}(\vec{P}_{s,2} - \vec{P}_{s,1})$$

where $\vec{P}_{s,i}$ are the vertex references by corner i of the corresponding surface s. Note that the fileformat standard of AC3D guarantees that the corners are given in a consistent order – so this normal computation will correspond to the geometrically correct value.

Next, set the normal of all of the vertices used each surface to this value. Note that this is a hack! It is only useful for quickly seeing that you have reasonable normals, before setting them correctly in the next step. Setting the normals in this way will overwrite any old value, so it is random which of the 3 or 4 possible normals is actually associated with each vertex that has 3 or 4 surfaces connected to it! If you have done this correctly it should now look similar to the left image in Figure 1.

Next, try to set the value of the normal for any vertex instead to be the *average* of the value of all the surfaces connected to it. *Hint:* this can be done easily by adding the XYZ parts to the normals in the loop over all surfaces, and in a second phase just normalizing the normals. Alternatively you can use the W part of the normals to store how many values have been added into them and make a division by W in the second phase (and resetting W to be zero at the time). Other possibilities of the second alternative is to weight the average differently for the different surface affecting a vertex, adding more (or less) contribution depending on the size of the surface.

If you have done this correctly it should now look similar to the right image in Figure 1.

Exercise 3.2 - Using buffer objects.

If you were to add lots of instances of your fighter class implemented in the previous exercise your code would start running slowly due to the bottleneck between the CPU and GPU memory. This is because you are sending all the 3D data multiple times to the GPU for every frame – even

Algorithm 2 Draw a (sub-)object

```
void World::modelRedrawLeaf(AC3DObject* obj)
{
  glm::ivec4* indices = new glm::ivec4[nSurfaces];
  glm::vec4* normals = new glm::vec4[nVertices];
  glm::vec4* colors ...

  ...

  // Create indices
  for (int i=0; i<nSurfaces; i++)
  {
    AC3DSurface surface = obj->surfaces[i];
    for (int j=0; j < surface.nVertices; j++) // nVertices is 3 or 4
    {
      indices[i][j] = surface.vert[j].index;
    }
  }

  // Create normals:
  // First create surface vectors
  glm::vec4 surface_normals[nSurfaces];
  for (int i = 0; i < nSurfaces; i++)
  {
    // Find three vertices of this surface.
    // Estimate normal from them.
    // Assign normals to vertices, 'hacky' or sensibly, as in the text.
  }

  // Set up and enable glVertexAttribPointers

  // Draw:
  for (int i = 0; i < obj->nSurfaces; i++) {
    if triangle
    {
      glDrawElements(GL_TRIANGLES, ...
    }
    else
    {
      glDrawElements(GL_QUADS, ...
    }
  }
}
```

though you only ever create/modify the model once. In this exercise you will rectify this by using *buffer objects* that can be used to store vertex attributes as well as indices on the GPU.[1]

Start by allocating a *single* buffer *on the CPU* that can be used for storing your vertices and normals. You may either choose to store all the normals after all the vertices in this buffer, or to interleave them by storing a normal after every vertex position.

Draw your objects with the old-fashioned *CPU-based* drawElements that you have used before, but using this buffer. If you choose to interleave the data you need to think about both the right starting address for passing in vertices respectively normals and to think about the *stride* argument given to `glVertexAttribPointer`. Note that all offsets and stride arguments are given in *bytes* to OpenGL. Also note that the stride argument is the number of bytes *from start to start* of two sequential attributes, not from end to start. (The situation when the attributes are densely packed, rather than interleaved, is a special case when you can specify a stride of 0, even though the start-to-start distance is not 0.)

Next, continue by allocating a new buffer object (using `glGenBuffers`), bind it as a `GL_ARRAY_BUFFER` object (using `glBindBuffer`) and copy the data from your buffer above into it (using `glBufferData`). Change your calls to `glVertexAttribPointer` to be done *relative to the buffer* (so no more pointers to a CPU-side array). Note that you need to compute the offsets of the corresponding vertex attributes and cast them into `(GLvoid *)` type.

Verify that your code works and give the same images as before. You have now done half the work in reducing the amount of data that is transferred.

Continue by allocating one or two buffer object(s) to store the index data for quads and triangles. Bind this object(s) as a `GL_ELEMENT_ARRAY_BUFFER` and upload the corresponding indices to it using the `glBufferData` function.

In your draw routine, bind the correct elements array buffer. Again, use the *relative address* in this buffer as the pointer to the data.

Finally, you may (if you want) allocate a *vertex array* object and use this object to store the (a) bound vertex buffer, (b) vertexAttribPointer arguments and (c) enabled attributes. This reduces the number of function calls that are needed in the draw routine for an individual object to four: (a) binding the vertex array, (b) binding the index buffer(s) and (c) calling draw elements twice.

Examination

To pass the lab you must hand in a demonstrate and hand in a report covering exercise 3.1 and 3.2. Both the report and the demonstration must pass in order for you to pass the lab. The grade for this lab is only *pass* or *fail*.

Deadline December 1

[1]Note that there exists many different forms of buffer objects and you can store also other data such as uniform variables in them