CS3388B: Lecture 11

February 28, 2023

11 Meshes, Buffers, and Retained Mode

In Lecture 10 we finally put all the pieces together on how to use the OpenGL rendering pipeline. Now, it's time to start doing some non-trivial rendering. As the number of vertices grows larger, we will need to be more cognizant of our rendering performance. That means moving away from immediate mode and toward retained mode!

But first, let's talk about the data that we will retain.

11.1 Triangle Meshes

A triangle mesh is a collection of triangles which are connected together into a **mesh**. They are connected together (sometimes only implicitly connected) by their common edges or vertices. We've already seen how a quad is subdivided into two triangles which share their hypotenuse. This is the simplest triangle mesh. But, in general, 3D artists and animators give us large meshes with thousands and thousands of triangles.

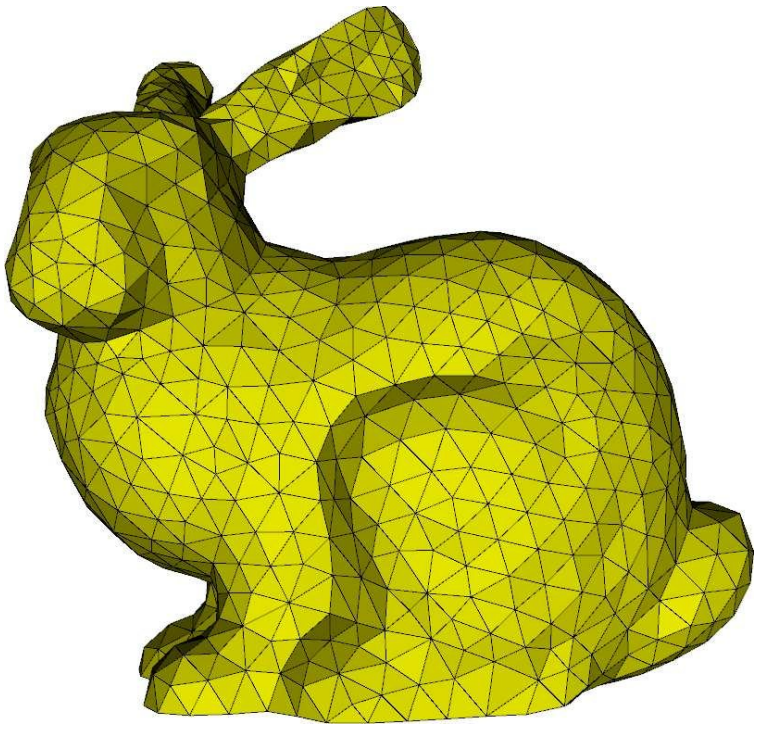

Figure 1: The Stanford Bunny as a mesh

The most common format of a triangular mesh is a **face-vertex mesh**. This type of mesh is spec-

ified in two parts. First, a list of individual vertices is specified. Then, a collection of individual triangles (faces) is specified using **vertex indices**.

Returning to a simple quad, its four vertices could be specified as:

```
(-1,-1),
(1, -1),
(1, 1),
(-1, 1)
```

Then, the two faces of its mesh can be specified using 6 indices: 0, 1, 2, 0, 2, 3. Notice this ordering specifies two triangles whose hypotenuses are (-1, -1) to (1,1). But with those same four vertices, we could specify a number of different triangles which still give the same mesh. It depends on what winding order we want and where we want the mesh to be divided. But, we usually leave that kind of details to the 3D artist. For us, we just get a list of vertices and a list of faces and we render it. To render large meshes we better use *retained mode*...

11.2 Vertex Buffer Objects

A **vertex buffer object** or VBO is an OpenGL object that is allocated on the server side and holds vertex attribute data. Much like texture objects (and all OpenGL objects), we follow the paradigm of generate, bind, and load.

Buffer objects are generic server-side stores of data;. When those buffers hold vertex data, they are called vertex buffer objects.

1. **Generate the buffer**: `glGenBuffers(int num, GLuint* IDs)` generates a `num` number of buffers objects and returns their **unique IDs** in the `IDs` array.

2. **Bind the buffer**: when we want to use the buffer for for vertex attributes, we bind the buffer to the `GL_ARRAY_BUFFER` target using `glBindBuffer(GL_ARRAY_BUFFER, bufferID)`.

3. **Fill the buffer**: The function `glBufferData` transfers data from the client side to a bound buffer target. It's similar in semantics to `memcpy`. `glBufferData(GL_ARRAY_BUFFER, numBytes, clientArray, usage)`

When we fill a buffer, we specify the expected **usage** of the buffer. The usage is a pair of parameters:

- `STREAM`, `STATIC`, `DYNAMIC`. In `STREAM`, buffer contents are modified once and used a few times. In `STATIC`, buffer contents are modified once and used many times. In `DYNAMIC`, buffer contents are modified and used many times.

- `DRAW`, `READ`, `COPY`. In `DRAW`, data is sent from client-side to server-side and buffer contents are used for drawing. In `READ`, data is read from the server-side top fill the buffer and then returned to the application. In `COPY`, buffer contents are both filled used a server-side read and used for drawing.

The most typical is `GL_STATIC_DRAW`: you specify the vertex attribute data once and then use it to draw over and over again.

Say you have an array of floats `vertexData` holding the positions of a bunch of vertices. The following code generates a buffer object and copies all that vertex data into that buffer.

```
glGenBuffers(1, &vertexBufferID);
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferID);
glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT)*numVertices*3,
             vertexData, GL_STATIC_DRAW);
```

We repeat this pattern for every vertex attribute we want to use to draw. We can fill another buffer with, for example, color data. Or texture coordinates, or normals, or whatever values we want to send to our custom vertex shader.

Now the question is, how do we tell the shader or OpenGL where those vertex attributes are available? In fact, we already know how to do this: `glVertexAttribPointer`. This function has two options:

- What we saw in Lecture 10 is that we pass a client-side array (pointer) as the last parameter to be read during the draw calls. This is only the case if there is no buffer bound to `GL_ARRAY_BUFFER`
- If a buffer is bound to `GL_ARRAY_BUFFER`, then the last parameter is rather interpreted as an *offset* into the bound buffer (an index into the buffer from where to start reading).

```
glGenBuffers(1, &vertexBufferID);
glBindBuffer(GL_ARRAY_BUFFER, vertexBufferID);
glBufferData(GL_ARRAY_BUFFER, sizeof(GL_FLOAT)*numVertices*3,
             vertexData, GL_STATIC_DRAW);

glEnableVertexAttribArray(0);          //enable location 0
glVertexAttribPointer(
    0,                                 // attribute location in shader
    3,                                 // this attribute has 3 coordinates
    GL_FLOAT,                          // type
    GL_FALSE,                          // normalized?
    0,                                 // stride
    (void*) 0                          // vertex buffer offset
);
```

We can repeat this process for each VBO and each attribute we want to pass to the vertex shader and then call `glDrawArrays` as before to render the data.

11.3 Drawing vertices by index

However, `glDrawArrays` draws the data in the buffers **in order**, one after the other, without reusing any of the vertices. If want to render a triangle mesh, we need to specify a list of vertices and then a list of indices to render rather than just all the vertices one after the other. This is

where `glDrawElements` can be used. But first, we need to specify an **element buffer**.

Much like vertex attribute data, we generate a buffer and bind it to some target, and then fill that buffer. The only differences are that we bind the buffer to `GL_ELEMENT_ARRAY_BUFFER` and that the data we fill the buffer with are unsigned integers (since they are indices). These buffers are sometimes called **element buffer objects** or **EBOs**.

```
//enable and specify vertex attributes

glGenBuffers(1, &elementBufferID);
glBindBuffer(GL_ELEMENT_ARRAY_BUFFER, elementBufferID);
glBufferData(GL_ELEMENT_ARRAY_BUFFER, sizeof(GL_UNSIGNED_INT)*numVertices,
             indexArray, GL_STATIC_DRAW);

glDrawElements(GL_TRIANGLES, numVertices, GL_UNSIGNED_INT, (void*) 0);
```

In the previous `glDrawElements` call, we specify 0 as the last parameter to represent the offset into the currently bound `GL_ELEMENT_ARRAY_BUFFER`.

11.4 Vertex Array Objects

Specifying `glVertexAttribPointer` for each attribute for each draw call is kind of annoying. Here is now where we introduce **vertex array objects** or **VAOs**. These special kinds of OpenGL objects encapsulate all the vertex attributes specifications and pointers so that you only have to specify the attribute pointer once. Isn't that nice?

The key idea is that, when a VAO is bound, any call to `glVertexAttribPointer` is **captured** by the VAO so as to "remember" the currently bound `GL_ARRAY_BUFFER` and the parameters specified to `glVertexAttribPointer`. The VAO will also remember a when a buffer is bound to `GL_ELEMENT_ARRAY_BUFFER`.

Then, one only needs to bind the VAO before a draw call to "recover" all the state which was previously captured by the VAO.

Using a VAO is as follows:

- **Generate** a VAO: `glGenVertexArrays(int num, GLuint* IDs)`, to generate `num` number of VAOs and return the unique IDs of those VAOs in the `IDs` array.
- **Bind** the VAO: `glBindVertexArray(GLuint vertexArrayID)`. Unlike buffers, which have several choices of targets to be bound to, there is exactly one bind target for vertex arrays. Therefore, we only specify the VAO ID which we want bound. We must bind the VAO **before** binding buffers objects and specifying vertex attributes.
- **Specify vertex attributes**. With the VAO bound, go through the process of specifying vertex attributes, binding buffers to `GL_ARRAY_BUFFER`, calling `glVertexAttribPointer`. If using `glDrawElements`, then you can also find to `GL_ELEMENT_ARRAY_BUFFER`.

- **Unbind** the VAO: Since the VAO, and OpenGL, is a state machine, we want to avoid unintended state being captured by the VAO. After specifying all the vertex attributes for a particular mesh/object to draw, one should **unbind** by calling `glBindVertexArray(0)` or bind a different VAO.
- **Bind** the VAO again just before drawing to recover all the buffer bindings and vertex attribute specifications.
- **Draw**. Call `glDrawArrays` to draw a number of vertices from the bound buffers, OR, call `glDrawElements` to draw using an element buffer object and thus vertex indices.

Put it all together: see L11.cpp for an example of using VBOs and VAOs.

11.5 Mesh Files

Using VBOs and VAOs, we know how to store mesh data and draw it. But where does the mesh data come from? Some artist creates the data and gives it to you, typically as a file. There are many different mesh data file formats. Some give lots of possibilities to include different vertex attributes, primitive types, etc.

Computer-Aided Design (**CAD**) programs are very sophisticated and have lots of proprietary file formats. Some softwares are AutoCAD, Solidworks, Maya, 3ds. They use file types like `.dwg`, `.fbx`, `.3ds`. Of course, open-source alternatives exist as well. MeshLab, Blender, VTK are some examples.

FBX files are particularly important in computer graphics as they are fundamental to industries like animation and video games. More generic and interoperable file formats are `.ply` and `.obj`.

11.5.1 Raw Mesh Files

The simplest kind of mesh file is a `.raw` file. Each line of this file specifies three vertices which connect together to form a single triangle. Each vertex is specified by its X, Y, Z coordinates. Thus, each line of this file is 9 floats, separated by spaces. In this format, there are no explicit connections between triangles. Each triangle is independent and connections are implied by two vertices having the same position.

The following 12 lines specify 12 triangles and 36 vertices in the `.raw` format. They form a cube.

```
0.0 0.0 0.0 1.0 0.0 -1.0 1.0 0.0 0.0
0.0 0.0 0.0 0.0 0.0 -1.0 1.0 0.0 -1.0
0.0 0.0 -1.0 1.0 1.0 -1.0 1.0 0.0 -1.0
0.0 0.0 -1.0 0.0 1.0 -1.0 1.0 1.0 -1.0
1.0 0.0 0.0 1.0 0.0 -1.0 1.0 1.0 0.0
1.0 0.0 -1.0 1.0 1.0 -1.0 1.0 1.0 0.0
0.0 0.0 0.0 0.0 1.0 0.0 0.0 0.0 -1.0
0.0 1.0 0.0 0.0 1.0 -1.0 0.0 0.0 -1.0
0.0 1.0 0.0 1.0 1.0 0.0 1.0 1.0 -1.0
```

```
0.0 1.0 0.0 1.0 1.0 -1.0 0.0 1.0 -1.0
0.0 0.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0
0.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 0.0
```

11.5.2 PLY Files

The PLY file format was developed at Stanford University and is intended to store the information of a single 3D object as a triangle mesh. This format (almost always) specifies a mesh using the face-vertex format (specifying edges and triangle strips is also supported but rarely used). It also allows the storage of arbitrary vertex attributes (although not all implementation of PLY file readers support all attributes).

A PLY file is broken into three parts:

- A header, giving the metadata of the file
- Vertices, one per line, given as positions and attributes
- Faces, one per line, specified as vertex indices.

The first line a ply file is always a single line with the three characters `ply`. The second line of a ply file describes its encoding and version number. There are three options (currently):

```
format ascii 1.0
format binary_little_endian 1.0
format binary_big_endian 1.0
```

Encoding a mesh as a binary file is more space-efficient and allows faster reading/writing. However, ascii allows the file to be human readable. We will only use ascii in this course.

Then, the next lines a PLY file are its header. A header is comprised of the following:

- Comments: lines beginning with `comment` are comments are not part of the header or mesh data.
- Elements: lines beginning with `element` declare the start of the definition particular kind of geometric object, and specifies the number of those elements which are included in the file after the header.
- Properties: lines beginning with `property` specify a property's type (int, float, etc.) and its interpretation (x position, y position, color, etc.) A property applies to the most recently declared `element`.

Property types can be one of: `char`, `uchar`, `short`, `ushort`, `int`, `uint`, `float`, `double`, or lists of one of those types.

Typical vertex property (names) are `x`, `y`, `z`, `u`, `v`, `nx`, `ny`, `nz`, `red`, `green`, `blue`, `alpha`. They have obvious interpretations. A vertex's position is (`x`, `y`, `z`). A vertex may have a normal (`nx`, `ny`, `nz`), a color (`red`, `green`, `blue`, `alpha`), or texture coordinates (`u`, `v`).

To specify a property is a list of elements we use:

```
property list uchar TYPE NAME
```

This tells us that the property is a list of `TYPE` numbers whose name is `NAME`. Particular instances of this property (after the header) start with a single `uchar` which indicates the number of elements in that particular list.

When a header has specified all its elements and properties, a header ends with a single line `end_header`.

```
ply
format ascii 1.0
comment this file is a cube of 8 vertices and 6 faces
element vertex 8
property float x
property float y
property float z
element face 6
property list uchar int vertex_indices
end_header
```

Every line after the header is data specifying elements. Mesh data should be specified in the order in which elements are declared in the header, grouped by element type. Typically, one lists all vertices and then all faces.

In ply files, faces are actually listed as **triangle fans**. The previous header says there will be 6 faces, each encoded as a list of integers representing vertex indices. Since we know a cube has 6 square faces, each face must be encoded as two triangles. Since each face is a triangle fan, this means 4 vertices are needed.

```
0 0 0
0 0 1
0 1 1
0 1 0
1 0 0
1 0 1
1 1 1
1 1 0
4 0 1 2 3
4 7 6 5 4
4 0 4 5 1
4 1 5 6 2
4 2 6 7 3
4 3 7 4 0
```

Of course, we could get the same cube by listing each triangle individually and having 2*6 = 12 faces of triangles. In that case, we would need to change the number of faces in the header to match:

```
ply
format ascii 1.0
comment this file is a cube of 8 vertices and 6 faces
element vertex 8
property float x
property float y
property float z
element face 12
property list uchar int vertex_indices
end_header
0 0 0
0 0 1
0 1 1
0 1 0
1 0 0
1 0 1
1 1 1
1 1 0
3 0 1 2
3 0 2 3
3 7 6 5
3 7 5 4
3 0 4 5
3 0 5 1
3 1 5 6
3 1 6 2
3 2 6 7
3 2 7 3
3 3 7 4
3 3 4 0
```

If we wanted a colored cube, where each vertex has a color vertex attribute, we would need to specify each vertex element in the header as:

```
element vertex 8
property float x
property float y
property float z
property float red
property float green
property float blue
```

Then, each vertex is specified as 6 floats on one line.

11.5.3 OBJ files

PLY files, semantically, give us a way to describe a single object to render. Another very common format is OBJ files. These files are similar in that they give lists of vertices and then lists of faces

using vertex indices.

However, OBJ files have a different structure. Each line is self-declaring of what that line encodes.

- Lines starting with `v` declare a vertex position
- Lines starting with `vt` declare a texture coordinate
- Lines starting with `vn` declare a normal
- Lines starting with `f` declare a face using indices (**1-indexed**!) of previously declared vertex positions (and optionally texture coordinates and normals).

Typically, OBJ files group together lines of the same type. For example, all lines describing vertex position, then all lines describing texture coordinates, then all lines describing the faces.

```
#this is a comment

#start listing vertices
v 0 0 0
v 0 0 1
v 0 1 1
v 0 1 0
v 1 0 0
v 1 0 1
v 1 1 1
v 1 1 0

#start listing faces
f 1 2 3
f 1 3 4
f 8 7 6
f 8 6 5
f 1 5 6
f 1 6 2
f 2 6 7
f 2 7 3
f 3 7 8
f 3 8 4
f 4 8 5
f 4 5 1
```

If we want faces to use more than just vertex positions, they use separate indices into the list of vertex normals, vertex texture coordinates, etc. Whereas ply files explicitly link vertex attributes to each vertex, OBJ files allows normals and texture coordinates to repeated and referenced by indices as well. In OBJ files, faces generically have the format:

```
f v1/vt1/vn1 v2/vt2/vn2 v3/vt3/vn3
```

The `/vti` and `/vni` is optional. If we want to specify vertex with texture coordinates we just need `vi/vti/` If we want to specify only normals without texture coordinates, we use two forward slashes and leave the texture index blank: `vi//vni`.

```
#this is a comment

#start listing vertices
v 0 0 0
v 0 0 1
v 0 1 1
v 0 1 0
v 1 0 0
v 1 0 1
v 1 1 1
v 1 1 0

#start listing texture coordinates
vt 0 0
vt 1 0
vt 0 1
vt 1 1

#start listing faces
f 1/1 2/2 3/3
f 1/1 3/3 4/4
f 8/1 7/2 6/3
f 8/1 6/3 5/4
f 1/1 5/2 6/3
f 1/1 6/3 2/4
f 2/1 6/2 7/3
f 2/1 7/3 3/4
f 3/1 7/2 8/3
f 3/1 8/3 4/4
f 4/2 8/3 5/1
f 4/2 5/1 1/2
```