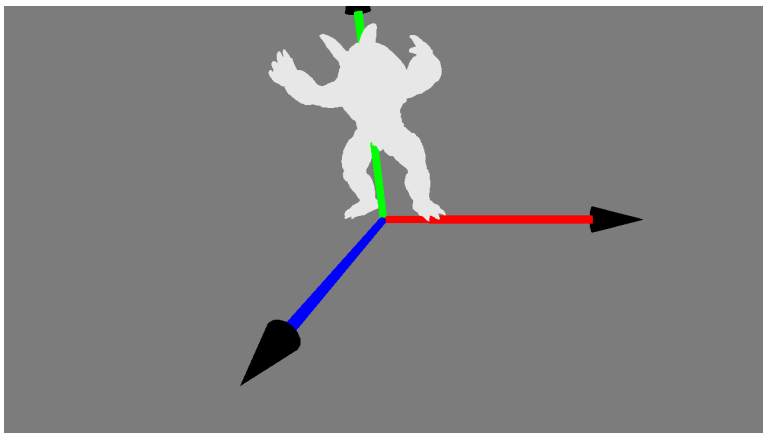
COMP3811 – Exercise G.4 Meshes and Models

Contents



Exercise G.3 covers mesh creation and loading. You will start by creating a few simple geometric meshes procedurally. We will combine these shapes to draw the coordinate axes shown in the teaser image. Then you will load a mesh from a Wavefront OBJ file from disk, using rapidobj.

It is recommended that you have completed Exercises G.2 and G.3 before this one.

Note: You are allowed to reuse code from your solutions to exercises in the courseworks, *if and only if* you are the sole author of the code in question. Keep this in mind when working on the exercises. While you are encouraged to discuss problems with your colleagues, you should not directly share any code. If you find solutions to your problems online, study the solution and then implement it yourself. Never copy-paste code, and always make sure you understand the code that you write!

If you run into problems, you are expected to attempt resolving your issues yourself first, e.g., by debugging your program. Do not just immediately ask for help. Assistants may ask you to show/explain what you've attempted so far. Help will only be provided if you have made a fair attempt first.

Recommendation: It might be tempting to just copy-paste the code from the PDF document into your solution while working through this exercise. I would recommend against this. You will gain far more by spending the small amount of extra time it takes to write the code yourself.

1 Geometric shapes

Make sure that you have your Exercise G.3 solution at hand. Transfer over select parts of the code:

- OpenGL Setup: perform the same setup as in G.3: enable the `FRAMEBUFFER_SRGB` state, enable depth testing and face culling, and set the clear color to e.g. a dark gray.
- Transformations: make sure you have valid world-to-camera space and projection matrices. You may want to reuse the arc-ball control setup.
- Shaders: You can reuse the shaders from G.3.
- Drawing: You may want to start by drawing the center cube (remove the rotation).

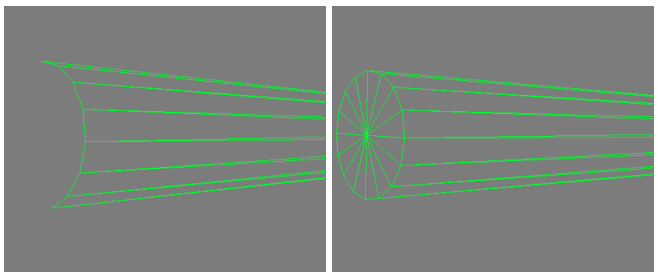


Figure 1: *Left: uncapped cylinder (wireframe). Face culling removes the back-facing triangle. Consequently, the "inside" surface of the cylinder is not visible. Right: capped cylinder. The caps closes the cylinder mesh at both ends.*

- Implementations of the matrix functions and operators, specifically for `Mat44f`.

Make sure the code runs, draws the center cube and lets you look around it. This will make debugging quite a bit easier down the road. In the following, we will essentially replace the cube with a more complex mesh built from cylinders and cones.

Start by studying the `SimpleMesh` structure defined in `simple_mesh.hpp`. It is simple indeed: it just contains a list of positions and colors (for now, we will stick to vertices that consist of a position and color). The mesh is a "triangle soup", meaning that is it simply defined by a list of arbitrary, unrelated triangles.

1.1 Cylinders

We will start by writing code to create a cylinder. Study the `cylinder.hpp` and `cylinder.cpp` sources. The former declares the function `make_cylinder`:

```
SimpleMeshData make_cylinder(
   bool aCapped = true,
   std::size_t aSubdivs = 16,
   Vec3f aColor = { 1.f, 1.f, 1.f },
   Mat44f aPreTransform = kIdentity44f
);
```

It accepts a few arguments:

- `aCapped`: determines if the cylinder should be capped (closed at the ends). See Figure 1.
- `aSubdivs`: number of subdivisions along the cylinder's periphery.
- `aColor`: color of the cylinder.
- `aPreTransform`: transform applied to the cylinder's vertex positions at creation time.

The function returns a `SimpleMeshData` structure, which contains the cylinders vertex positions and vertex colors.

We always start by creating a cylinder that extends along the x-axis and is centered around the axis. It will start at $x = 0$, end at $x = 1$ and have radius $r = 1$. Later, we will use the `aPreTransform` to transform the "base" cylinder into different places and forms. The base cylinder consists of a shell and (optionally) two caps.

To create the shell, we walk around a circle in the YZ-plane in a fixed number of steps (∼`aSubdivs`). With each step, we create two triangles (=a quad) that form one segment in the cylinder's outer shell. The triangles are formed between the near end $x = 0$ and far end $x = 1$ and between the current and previous steps along the circle. We need to be careful to emit the vertices in the right order, such that the triangles they form are front-facing towards the outside.

Task: Implement the cylinder shell's generation in `make_cylinder` (`cylinder.cpp`):

```
   std::vector<Vec3f> pos;

   float prevY = std::cos( 0.f );
   float prevZ = std::sin( 0.f );

   for( std::size_t i = 0; i < aSubdivs; ++i )
   {
      float const angle = (i+1) / float(aSubdivs) * 2.f * 3.1415926f;

```

```
        float y = std::cos( angle );
        float z = std::sin( angle );

        // Two triangles (= 3*2 positions) create one segment of the cylinder's shell.
        pos.emplace_back( Vec3f{ 0.f, prevY, prevZ } );
        pos.emplace_back( Vec3f{ 0.f, y, z } );
        pos.emplace_back( Vec3f{ 1.f, prevY, prevZ } );

        pos.emplace_back( Vec3f{ 0.f, y, z } );
        pos.emplace_back( Vec3f{ 1.f, y, z } );
        pos.emplace_back( Vec3f{ 1.f, prevY, prevZ } );

        prevY = y;
        prevZ = z;
    }
```

Before we continue on to define the caps, it is a good time to ensure the shell is generated correctly.

Task: Return a `SimpleMeshData` structure from the `make_cylinder` function. Initialize the `colors` of the `SimpleMeshData` to `aColor` (note: one color is required for each vertex):

```
    std::vector col( pos.size(), aColor );

    return SimpleMeshData{ std::move(pos), std::move(col) };
```

Task: Implement the `create_vao` function. It should upload the data from a `SimpleMeshData` instance to two VBOs and create the corresponding VAO from those two buffers. It is declared in the `simple_mesh.hpp` header and its definition is in `simple_mesh.cpp`. You can mostly reuse VBO and VAO creation code from previous exercises. However, pay attention to the arguments to `glBufferData`:

```
    glBufferData( GL_ARRAY_BUFFER, aMeshData.positions.size() * sizeof(Vec3f), aMeshData
      ▷ .positions.data(), GL_STATIC_DRAW );
```

The `data()` method of `std::vector` returns a pointer to the data contained in the vector. The specification of `std::vector` guarantees that all elements in the vector reside in a linear memory buffer, making this operation possible. The `size()` method returns the *number of elements* in the vector. We therefore need to multiply it with the size of an element (here `sizeof(Vec3f)`) to compute the total size of the buffer in bytes.

> The `sizeof` operator returns the size of the structure/type passed to it. This size is determined at compile time. For a structure, it is often just the memory required to hold all the member variables. A common error is to try to use `sizeof(std::vector<Vec3f>)`. However, this returns the size of the `std::vector<Vec3f>` class, and *not* of the data held by the vector.
> The size of the `std::vector<Vec3f>` (i.e., the result of `sizeof(std::vector<Vec3f>)`) is somewhat implementation dependent. It is possible to implement a `std::vector`-like container with just three pointers, which would result in 12 bytes on a 64-bit system. This is indeed what one sees with the standard library of Visual Studio. However, both GCC (libstdc++) and Clang (libc++) use different implementations that use a whole 24 bytes for each `std::vector` instance!

Task: Verify that you can draw the shell of the cylinder. Compare to Figure 2. During loading/setup call `make_cylinder` to return a `SimpleMeshData` with the shell. Pass this to `create_vao` to create the corresponding VAO that can be used to draw the shell:

```
    auto testCylinder = make_cylinder( false, 16, {1.f, 0.f, 0.f} );
    GLuint vao = create_vao( testCylinder );
    std::size_t vertexCount = testCylinder.positions.size();
```

In addition to the VAO object ID (`vao`), we also need to remember the total vertex count (`vertexCount`) such that we can pass it to the `glDrawArrays()` function later. (Don't forget to update the `glDrawArrays()` call in the main loop!).

Rendering in wireframe mode is very helpful at this stage. You can use the `glPolygonMode` with the `GL_LINE` mode to draw triangles as just a set of lines:

```
    glPolygonMode( GL_FRONT_AND_BACK, GL_LINE );
```

You can place the call anywhere before `glDrawArrays`. In my reference solution, I ended up calling it each frame, just before the corresponding `glDrawArrays` call.

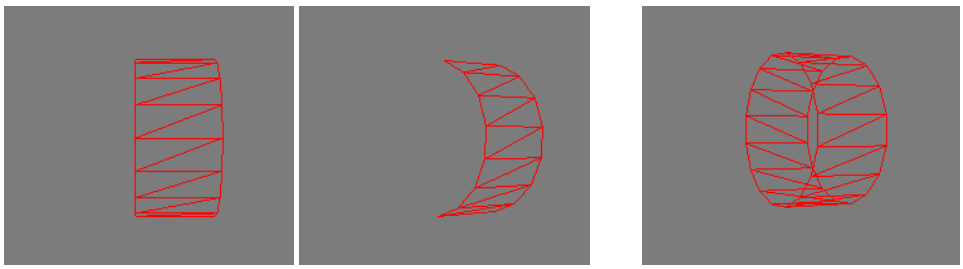


Figure 2: *Cylinder shell. (left) initial view. (Mid.) Slightly rotated. (Right) Without face culling.*

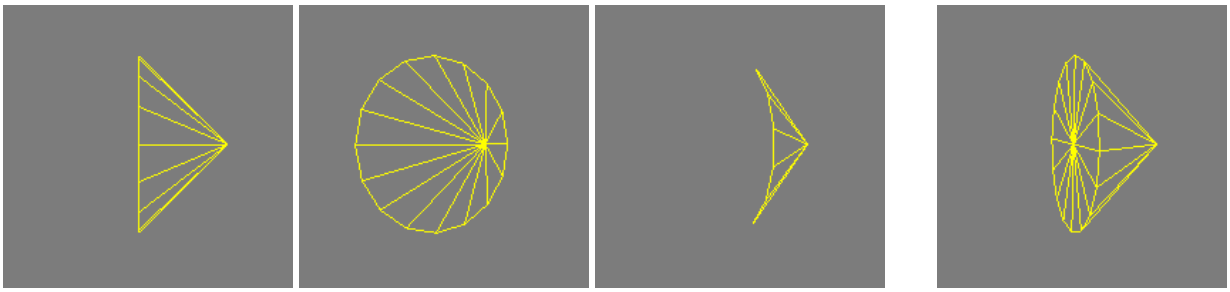


Figure 3: *Base cone from a few different angles. The rightmost image includes a cap, the others do not.*

Rotate the camera around the shell to verify that it is generated correctly with no gaps. (Alternatively, temporarily disable face culling.)

Task: Implement the generation of the geometry of the caps. Generate caps only when the `aCapped` argument is set to true. Make sure that the caps are generated with the correct facing by looking at the cylinder from both sides (enable face culling again if you disabled it previously). Check that a color is generated for each of the cap vertices as well.

Task: Finally, implement the transformation of the positions with the `aPreTransform` matrix:

```
for( auto& p : pos )
{
   Vec4f p4{ p.x, p.y, p.z, 1.f };
   Vec4f t = aPreTransform * p4;
   t /= t.w;

   p = Vec3f{ t.x, t.y, t.z };
}
```

Generate different cylinders by passing different transformations to the `make_cylinder` function:

```
auto testCylinder = make_cylinder( true, 16, {0.f, 1.f, 0.f},
   make_rotation_z( 3.141592f / 2.f ) *
   make_scaling( 5.f, 0.1f, 0.1f ) // scale X by 5, Y and Z by 0.1
);
```

This may require you to implement the new `make_scaling` function in the `vmlib/mat4.hpp` header.

1.2 Cones

Task: Implement the `make_cone` function declared in the `cone.hpp` header and defined in `cone.cpp`.

Refer to Section 1.1 for inspiration. A cone is essentially a cylinder, where all vertices on one side are collapsed to a single point in the middle (you will also only need to generate one cap, if requested). See Figure 3 for the "base cone" from a few different view angles.

1.3 Arrows

We will now create an arrow mesh. An arrow is just an elongated cylinder with a cone at the end of it. Fortunately, we already know how to create both cylinders and cones.

A simple strategy is to create a cylinder and a cone independently, and with a separate VAO. The VAOs could then be drawn consecutively, with two different model-to-world transforms, to place them at the end of each other.

Figure 4: *Arrow created from a cylinder and a cone.*

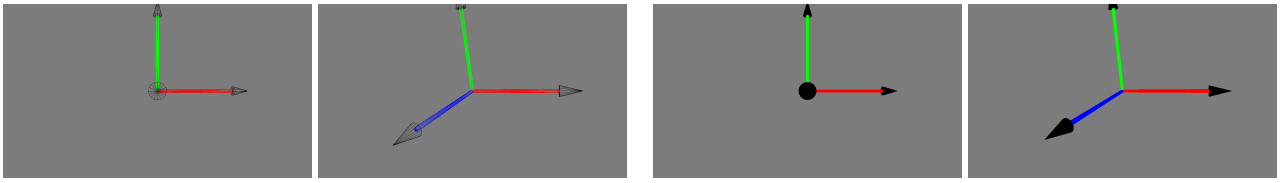


Figure 5: *Three arrows showing the coordinate axes. The left two pictures are drawn in wireframe mode, and the right two using the default* `GL_FILL` *polygon mode.*

Optional: Implement the above simple strategy.

However, in modern terms, both the cylinder and cone meshes are tiny. We can therefore just combine the two into a single larger mesh, and draw that. This is where our `aPreTransform` argument comes in very handy.

Task: Create a arrow pointing along the x-axis from a cylinder scaled by $x = 5$, $y = 0.1$ and $z = 0.1$, and a cone scaled by $x = 1$, $y = 0.3$ and $z = 0.3$ and translated to the end of the cylinder. Merge the two meshes using the `concatenate` function declared in `simple_mesh.hpp`:

```
auto xcyl = make_cylinder( true, 16, {1.f, 0.f, 0.f},
   make_scaling( 5.f, 0.1f, 0.1f )
);
auto xcone = make_cone( true, 16, {0.f, 0.f, 0.f},
   make_scaling( 1.f, 0.3f, 0.3f ) * make_translation( { 5.f, 0.f, 0.f } )
);

auto xarrow = concatenate( std::move(xcyl), xcone );
```

(The `concatenate` function is already defined, so you do not have to implement it yourself.)

Here, the body of the arrow (the cylinder) is red, and the pointy bit (the cone) is black. Draw it and verify that it matches what we expect (Figure 4).

Task: Create two more arrows for the y- and z-axes, respectively. The arrows should point along the indicated axis. Give each arrow's body a different color (e.g., green for the y-arrow and blue for the z-arrow). Merge all three arrows into a single mesh using repeated calls to `concatenate`.

Draw the results and compare to Figure 5.

Optional

Optional: The repeated calls to to `concatenate` result in a large number of (re-)allocations. While we don't really care about performance too much in this phase – this is only done once, when first loading resources – it is somewhat irksome. Create a function that takes an arbitrary number of `SimpleMeshData` objects, computes the total number of vertices present in all of them, allocates memory for the result once, and merges all meshes into the resulting buffer. Take care that you don't accidentally create copies of the `SimpleMeshData` instances when passing them to the function!

Optional: The above can be improved further. Instead of creating the merged `SimpleMeshData` instance with the data of all input meshes, and then uploading the merged instance to OpenGL with `create_vao`, we could just define a `create_vao` that takes multiple `SimpleMeshData` instances. It would then create VBOs large enough to hold the positions/colors from all passed-in meshes, and copy each mesh's data into the VBOs directly. Implement this. You will need either glBufferSubData or (advanced) glMapBuffer.

Optional: Create a new function similar to `make_cylinder`. It will also create a cylinder. However, the function should have an additional argument, which defines the number of subdivisions along the length of the cylinder.

Advanced: Extend the previous function to create a bent cylinder.

2 Mesh loading

The next goal is to draw a model loaded from an external file

2.1 Armadillos

Unlike arrows, armadillos are not just a combination of cylinders and cones. In fact, it would be quite difficult to create an armadillo from any combination of simple geometric primitives programmatically. We will therefore load the armadillo mesh from a Wavefront OBJ file (`assets/Armadillo.obj`). The original Armadillo model used here stems from the Stanford 3D scanning repository, and was (presumably) acquired through a 3D scan. You can find other models in the repository as well (but please read the notes on their usage first!). The models are provided in the PLY format, so you might need to use software such Blender to convert the models from PLY to OBJ. (The included Armadillo Wavefront OBJ mesh is additionally scaled down, rotated and translated compared to the original PLY mesh.)

We will use the rapidobj OBJ file loader to do the heavy lifting.

Task: Briefly inspect the Armadillo OBJ files. The files are text files. Open `assets/Armadillo.obj` and `assets/Armadillo.mtl` in a text editor.

The former file contains the geometric data; the latter defines the materials used by the model. In this case, the material file only defines a single material called `None`. It is a very simple material with no textures and a uniform light gray appearance.

The geometry is defined mainly by three distinct types of lines. Lines starting with the letter `v` define a vertex position. Lines starting with the letters `vn` define a vertex normal. Lines starting with the letter `f` define a face, which forms one or more triangle. Each face is constructed from three or more positions, texture coordinates and normals. The sample model does not contain texture coordinates, so the index for those is omitted in the face defintions. Texture coordinates would be defined by lines starting with the letters `vt`.

For now, we will load the `Armadillo.obj` model with rapidobj and return it as a `SimpleMeshData` instance. That way, we can use the existing infrastructure for drawing.

Task: Implement the `load_wavefront_obj` function declared in `loadobj.hpp` and defined in `loadobj.cpp` as follows:

```
SimpleMeshData load_wavefront_obj( char const* aPath )
{
    // Ask rapidobj to load the requested file
    auto result = rapidobj::ParseFile( aPath );
    if( result.error )
        throw Error( "Unable to load OBJ file '%s': %s", aPath, result.error.code.message
        ▷ ().c_str() );

    // OBJ files can define faces that are not triangles. However, OpenGL will only render triangles (and lines
    // and points), so we must triangulate any faces that are not already triangles. Fortunately, rapidobj can do
    // this for us.
    rapidobj::Triangulate( result );

    // Convert the OBJ data into a SimpleMeshData structure. For now, we simply turn the object into a triangle
    // soup, ignoring the indexing information that the OBJ file contains.
    SimpleMeshData ret;

    for( auto const& shape : result.shapes )
    {
        for( std::size_t i = 0; i < shape.mesh.indices.size(); ++i )
        {
            auto const& idx = shape.mesh.indices[i];

            ret.positions.emplace_back( Vec3f{
                result.attributes.positions[idx.position_index*3+0],
                result.attributes.positions[idx.position_index*3+1],
                result.attributes.positions[idx.position_index*3+2]
            } );

             // Always triangles, so we can find the face index by dividing the vertex index by three
            auto const& mat = result.materials[shape.mesh.material_ids[i/3]];

```

```
            // Just replicate the material ambient color for each vertex...
            ret.colors.emplace_back( Vec3f{
               mat.ambient[0],
               mat.ambient[1],
               mat.ambient[2]
            } );
         }
      }

      return ret;
}
```

There are essentially three phases to the loading. First, rapidobj parses the input file. Second, we ask rapidobj to triangulate any non-triangle faces (OBJ files can define quads and arbitrary polygons). Third, we convert the data returned by rapidobj into our own `SimpleMeshData` structure. At the moment, we are only interested in the positions, which simplifies this a bit. The vertex colors are based on the ambient color of the associated material.

Task: In `main()`, load the `asssets/Armadillo.obj` file with `load_wavefront_obj`. Create a separate VAO for the Armadillo mesh. Draw the mesh after drawing the coordinate axes with `glDrawArrays` (you will also need to store the number of vertices in the Armadillo mesh somewhere).

Compare your results to the teaser image (first page).

Optional

Optional: Draw a second copy of the armadillo next to the first one. Do so without duplicating the Armadillo vertex data (i.e., reuse the same VAO).

Optional: How many Armadillos can you draw before experiencing a significant dip in rendering speed (=frame rate)? See Figure 6 for an example. can you determine if you are bound by CPU or by GPU? (This will depend on your hardware!)

Advanced: There are more efficient ways to draw many copies of the same object. This is generally referred to as instancing, and specialized functions like `glDrawElementsInstanced` or `glMultiDrawElements` exist for this purpose. Can you make that work? (You might need to use an UBO for all the uniform data.) Does this improve rendering speed? (Why? Why not?)

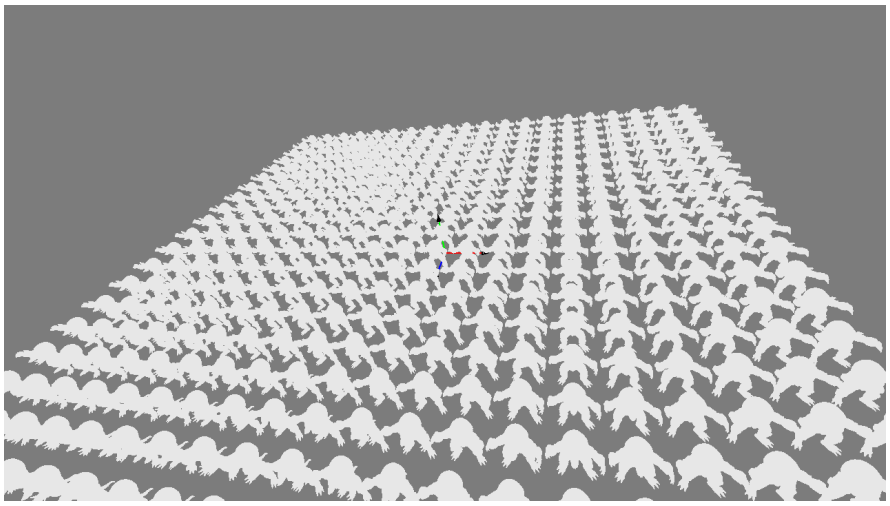


Figure 6: *Armadillos arranged in a 2D lattice. Drawing the Armadillo 400 times totals at about 130Mtris. This is noticeably slower on a high end computer.*