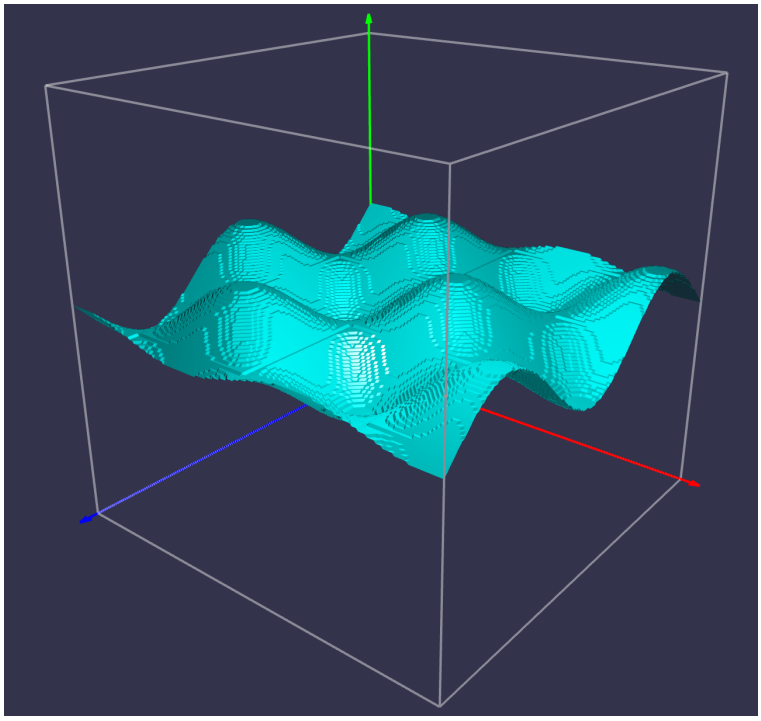
University of Western Ontario, Computer Science Department
CS3388B, Computer Graphics

Assignment 5 **Due: Sunday March 26, 2023**

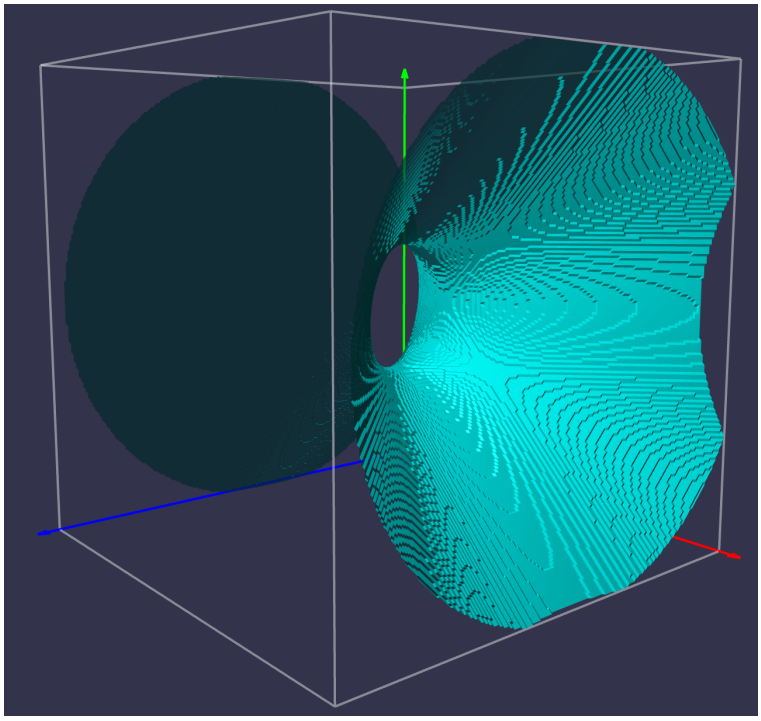
General Instructions: This assignment consists of 6 pages, 1 exercise, and is marked out of 100. For any question involving calculations you must provide your workings. *Correct final answers without workings will be marked as incorrect.* Each assignment submission and the answers within are to be solely your individual work and completed independently. Any plagiarism found will be taken seriously and may result in a mark of 0 on this assignment, removal from the course, or more serious consequences.

Submission Instructions: The answers to this assignment (code, workings, written answers) are to be submitted electronically to OWL. Ideally, any workings or written answers are to be typed. At the very least, clearly *scanned* copies (no photographs) of hand-written work. If the person correcting your assignment is unable to easily read or interpret your written answers then it may be marked as incorrect without the possibility of remarking. Include a **README** with any code.

Install Tips: I highly recommend downloading GLEW: `https://glew.sourceforge.net/` and (if working in C++) GLM: `https://www.opengl.org/sdk/libs/GLM/`. My example code (**which you should pillage** to help do this assignment) use GLEW and GLM.



The surface generated from $y - \sin(x)\cos(z)$ with an isovalue of 0.



The surface generated from $x^2 - y^2 - z^2 - z$ with an isovalue of -1.5.

Exercise 1. The goal of this assignment is generate triangle meshes and get started with non-trivial shaders in OpenGL.

At a high-level, your program will:

- Implement and execute the marching cubes algorithm on an arbitrary scalar field.
- Implement a shader *with lighting*.
- Manipulate the view matrix to have the appearance that the object is rotating around the origin.
- Write triangle mesh data to a file.

The end product of this assignment is to have a program which executes marching cubes, renders the result, and saves the resulting mesh to a PLY file.

The Camera.

1. You should be able to "click and drag" the object in view so that it appears to spin in-place around the origin. This effect will actually be implemented using camera movement. See, for example, the plot generated by Google from this google search.
2. This can be achieved using *spherical coordinates* (`https://en.wikipedia.org/wiki/Spherical_coordinate_system`).
3. We are going to use a create a limited "first-person camera", where you (the character) are viewing the world as if you were in it. You will directly be controlling the camera. This will be reviewed in Lecture 14.
4. The key idea is that we will specify the camera's position relative to the world origin using three parameters: r, θ, φ. During a "click and drag", any horizontal motion will modify θ; any vertical motion will modify φ.
5. We modify r using the arrow keys. Pressing the up arrow key will decrease r, moving the camera closer to the world origin. (*but stopping before r reaches 0*). Pressing the down arrow key will increase r, moving the camera further from the world origin.
6. Begin with the camera being positioned at $(5, 5, 5)$ in world space and looking at $(0, 0, 0)$ in world space.

The Marching.

Implement a function for marching cubes. Its parameters are: a function encoding a *scalar field*, an *isovalue*, a *minimum value* for the grid, a *maximum value* for the grid, and a *step size*. It should return a list (or `std::vector`) of floats encoding the x, y, z positions of the generated vertices.

```
std::vector<float> marching_cubes(
    std::function<float(float, float, float)> f,
    float isovalue,
    float min,
    float max,
    float stepsize);
```

- `f` is a function which takes 3 floats (`x,y,z`) and returns a single float. Thus, `f` implements a scalar field.
- `isovalue` is a number which represents the value in the scalar field to use as the boundary. Let `val` be the value returned by the function call `f(x,y,z)`. If `val < isovalue` then the point (x, y, z) is considered to be *inside* the object. If `val > isovalue` then the point (x, y, z) is considered to be *outside* the object.
- `min` is the minimum value for each of x, y, and z to use during the marching.
- `max` is the maximum value for each of x, y, and z to use during the marching. Thus, the grid over which the algorithm marches is a always a cube.
- `stepsize` is the side length of each cube during the march.
- To help with your implementation, the marching cubes lookup table (with 256 entries) is provided on OWL.

The Processing.

1. Implement a function `compute_normals` which takes the list of floats generated by `marching_cubes` and returns another list of floats which encodes the normal vectors for each vertex in the input list.
    - All three vertices which make up a single triangle should have the same normal vector. But, since each vertex has its own normal vector, that vector should be repeated thrice in the list (so the input list and the output list of floats will be of the same size.)
    - Make sure each normal vector is *normalized*.
    - Assume a counter clockwise winding order when computing normals.
2. Write a function `writePLY` with 3 parameters: `vertices`, the list of floats generated by `marching_cubes`; `normals`, the list of floats generated by `compute_normals`, and `fileName`, a string for the name of the file.
    - This function will create the file named `fileName` and write to it a *valid* PLY file encoding the vertices and normals in `vertices` and `normals`.

- This function **does not and should not** share vertices between faces. Let every face be declared by three unique vertices. Thus, the faces in your PLY file should be written like:

```
3 0 1 2
3 3 4 5
3 6 7 8
3 9 10 11
3 12 13 14
...
```

The Rendering.

1. Draw a box around the marching volume. That is, draw the cube whose opposite corners are (`min`, `min`, `min`) and (`max`, `max`, `max`). Where `min` and `max` are as the parameters to `marching_cubes`. Only draw the edges of this box.

2. Draw three coordinate axes corresponding to the positive x direction, positive y direction, and positive z direction. The tail of each axis should start at (`min`, `min`, `min`) in world coordinates and have length `max - min`.

3. Store your vertex positions and normals in VBOs and use a VAO alongside `glDrawArrays` to draw the triangles generated by marching cubes.

4. Draw the marching cubes triangles using a *Phong-like shader*. That is, a shader which incorporates lighting, ambient, diffuse, and specular colors. (See Lecture 13).

   (a) Implement the computation of the required vectors in the vertex shader (so they can be interpolated before being input to the fragment shadeR).

   (b) Implement the color computation in the fragment shader.

   (c) The vertex shader should use three uniform variables: `MVP`, the model-view-projection matrix; `V`, the view matrix; and `LightDir`, a `vec3` describing the direction of the light source.

   (d) The fragment shader should use a uniform variable `modelColor` which is the base color to be used for the diffuse component.

   (e) Let the ambient color be $(0.2, 0.2, 0.2)$. Let the specular color be $(1, 1, 1)$. Let *shininess* be 64.

   (f) Your fragment shader must correctly compute the combination of ambient, diffuse, and specular colors. The latter two require computing the correct lighting direction.

The Bonus.

If you are up to a challenge here is a bonus task, worth 25% extra on the overall assignment (that's 3.75% of your overall mark in the course).

- Somehow implement your program so that the triangle mesh is rendered **during** the marching algorithm. Thus, your mesh should continuously "grow" as the algorithm progresses.

The Submission.

Submit to OWL:

1. Your source code. (One or more files depending on how you defined the classes).
2. A README explaining what you did and why you did it like that, any known bugs, instructions for compiling. It doesn't have to be long, but it should be informative to the person marking.
3. Two screenshots of your program rendering the two functions described on page 2. If you find any other cool scalar fields, submit a screenshot of that too!
4. The PLY file generated for each of the two functions described on page 2.