Drawing the grid

For drawing an $N \times N \times N$ grid, we need to draw $O(N^3)$ lines. Assuming 2 points per line, 3 floating point coordinates per point, this approach will need to store $\sim 100^3 \times 2 \times 3 \times 4$ bytes just to draw grid lines ! Our implementation makes use of the inherent symmetries when drawing these grid lines to significantly reduce the memory required. We observe that in a grid line, many lines are parallel to each other and the endpoints differ only in one coordinate. Therefore, we only store points required to draw lines on X-Y plane and reuse these coordinates in various different ways to fill the complete grid. Because of this, we only require $O(N)$ memory for drawing grid as opposed to $O(N^3)$ in the naive approach.

$$\text{offset}_i = D_{\text{min}} + \frac{i-1}{N} D_{\text{max}}$$

- Usually for drawing a NxNxN grid one may need to define `2*(N^N^N)` vertices in the grid_vbo (for storing both x&y coords.), but here in this implementation we have tried to reduce the memory overhead from the graphics memory by using the `glDrawArraysInstanced()` function to draw the grid lines instead of using `glDrawArrays()`. This approach helped us to hugely reduce the memory overhead from `O(N^N^N)` to `O(N)`.
- This function helped us to reuse the limited set of vertices (`8*N` to be exact, 2 times for storing both ends of the vertices in both x and y axis) while drawing the grid-lines. We are storing only the end points of a grid-line in each axis. To ensure the robustness of the application, i.e. to handle varying grid sizes (N) and ensuring the cell size of `5x5x5, we have definedN_UNITSas 5 whereasDRAW_MINandDRAW_MAXwhich defines the bounds of the grid depends onN_CELLS` (N)and`N_UNITSsuch that it can accomodate a varying number of cells of size5x5x5‘` each.

NOTE: User can vary the value of `N_CELLS` in `main.hpp` to change the grid size, be default it is set to `100` which means a grid of `100x100x100` cells.

- We have created a separate array called **`grid_offsets`** which holds the offset distances between successive grid-lines which is then used by the grid vertex shader while drawing the grid-lines.
- Grid offsets are stored in a way such that the ith offset defines the distance between the i-1th and ith grid line. i.e:

```
grid_offsets[i] = DRAW_MIN + (i-1) * (DRAW_MAX - DRAW_MIN)/ N_CELLS
```

- While drawing the grid-lines, we first started with the horizontal lines in `x-z` plane, later we are rotating the lines by `90 degrees` to draw vertical lines in the `x-z` plane. This procedure draws a 2D grid in `x-z` plane, which is finally repeated in other y axis values to fill the whole 3D volume.
- Plots representing drawing of the grid lines:

Drawing cursor cube

Idea behind drawing the cursor cube was very similar to `Tutorial_02`. However, the main difference is that cube will be moving around the grid. We maintain a global position of cursor using 3 floating point numbers `cursor_x`, `cursor_y`, `cursor_z`. These point to the left-bottom-back coordinate of the cube that cursor is currently on. First, we create a static cube such that `cursor_x = cursor_y = cursor_z = 0` and push it to VBO. Then we use callbacks to listen to relevant keypresses and update these values accordingly. For every subsequent render of the cursor cube, we simply add the `[cursor_x, cursor_y, cursor_z]` vector to each vertex of the cube to get the current (and correctly translated) version of cube. Since this translation vector is common to all vertices, we pass it as a uniform to the vertex shader.

When the cursor cube is on a filled cell, we need to make it bigger for better visualization. To achieve this, we created one more cube which is larger than the standard cube. Our method of translating the cube using uniforms is still compatible with this larger version of cube since the larger cube's center is same as standard cube's center. Now we have 2 cubes – 1 standard and 1 larger. To switch between these, we simply maintain a pointer to their base addresses. If we detect that cube's state has changed (non-filled to filled or vice versa) then we update the pointer accordingly and refresh VBO to load the correct version

of cube. We also maintain a similar pointer for color attributes which by default points to default bright green color or points to model's color when the cube is on a filled cell.

Drawing model

Representation:

A model is simply a list of cube coordinates and their corresponding colors. We characterize each cube by its left-bottom-back coordinate and only store that in the model. C++-wise, this is simply a `std::map` which maps a `Point` (custom datatype similar to `glm::vec3`) representing left-bottom-back coordinate of a cube to a `Point` which represents color of that particular cube. The `Point` in key stores the $(x, y, z); x, y, z \in [D_{\text{min}}, D_{\text{max}}]$ coordinates while `Point` in value stores $(r, g, b); r, g, b \in [0, 1]$ values.

Drawing:

To draw multiple cubes, we can simply generate 12 triangles for each cube, push them to VBO along with their colors and simply do `glDrawArrays(GL_TRIANGLES, ...)`. However, this naive approach requires a lot of memory. In the worst case, when all 1M blocks are filled, we will be storing 12M triangles ! While modern GPUs can easily push billions of triangles, we will certainly be doing a lot of unnecessary work and occupy a lot of memory. Each triangle has 3 vertices, each vertex has 3 floating point coordinates which means this approach requires $12 \times 100^3 \times 3 \times 3 \times 4 \approx 400$MB of RAM just to store triangle coordinates. The exact same amount of RAM will also be required to store triangle colors (each vertex has 3 colors which are floating point as well) which means in total the program will require 800MB RAM to draw the $100 \times 100 \times 100$ cube.

We reduce these requirements significantly by observing that when 2 cubes are adjacent to each other, the faces where they meet each other are never seen under any viewing conditions. This means that we do not need to store triangles corresponding to these faces in the VBO at all ! We maintain a list of maybe-visible triangles and only pass that to the rasterizer to rasterize. A similar list is maintained which stores the colors of the corresponding triangles.

Insertion/Deletion:

Insertion/deletion of a cube uses a neat geometric observation. Let's say there's a cube at origin $(0, 0, 0)$ and we want to add a cube at $(0, 0, n)$. Here, the right face of cube at $(0, 0, 0)$ and the left face of the cube we want to add $(0, 0, n)$ coincide. As explained above, we will not store triangles for these coinciding faces in the VBO and therefore we must remove triangles for left face of $(0, 0, 0)$ from the existing list of triangles. To complete the addition of $(0, 0, n)$, we will add triangles corresponding to the remaining 5 faces of the cube at $(0, 0, n)$ since they do not coicide with any other cube. We can represent this operation as :

$$\text{insertAt}(x, y, z): \quad \text{addSet}, \text{removeSet} = \text{trianglesAt}(x, y, z),\ \text{trianglesList} \leftarrow \text{trianglesList} - \text{removeSet} + \text{addSet}$$

Since, the world is completely voxelized and that $\text{addSet} \cup \text{removeSet} = \text{allTriangles}$, $\text{addSet} \cap \text{removeSet} = \phi$ while removing the cube from (x, y, z) we can just swap removeSet and addSet to get the desired result. Note that here allTriangles represents the full list of 12 triangles corresponding to 6 faces of the cube.

$$\text{removeAt}(x, y, z): \quad \text{addSet}, \text{removeSet} = \text{trianglesAt}(x, y, z),\ \text{trianglesList} \leftarrow \text{trianglesList} - \text{addSet} + \text{removeSet}$$