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.

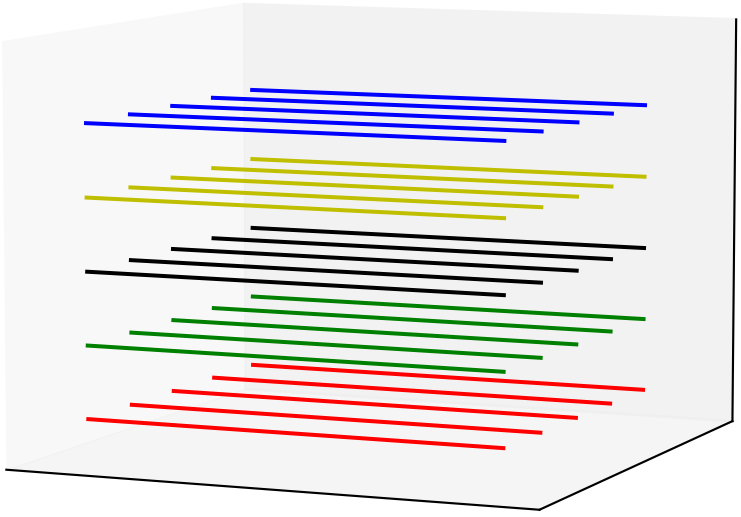

Figure 1: Illustration of instanced drawing for $N = 5$, instances = 5

Figure above shows our approach for $N = 5$. We only store coordinates to draw red colored lines in the VBO. To draw next set of lines (green, black, yellow and blue), we can use these exact same points in (x, y) but we must draw them at some z height. We pass this offset z information via a uniform type of shader variable where offset of i^{th} line is calculated as follows:

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

This completes all the lines that need to be drawn for this $5 \times 5 \times 5$ grid that are parallel to XY plane. Now we can use the same trick to draw lines parallel to XZ and YZ planes by just swapping around the coordinates. Note that for drawing lines parallel to other planes we do not need to push more points to VBO, we just need to change their ordering in the shader by accessing the `gl_InstanceID` variable. For first $N + 1$ instances, we use the uniform array value as the z while using the VBO information as x, y. For next $N + 1$ instances, we use the uniform array value as the x value while the VBO information as y, z and for the last $N + 1$ instances, we use the uniform array value as y value while interpreting the VBO information as x, z.

Overall, we use `glDrawArraysInstanced(GL_LINES, 0, 2*(N+1), 3*(N+1))` to draw the entire grid in one draw call while storing only $O(N)$ points in the VBO.

Drawing the 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 the 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 :

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

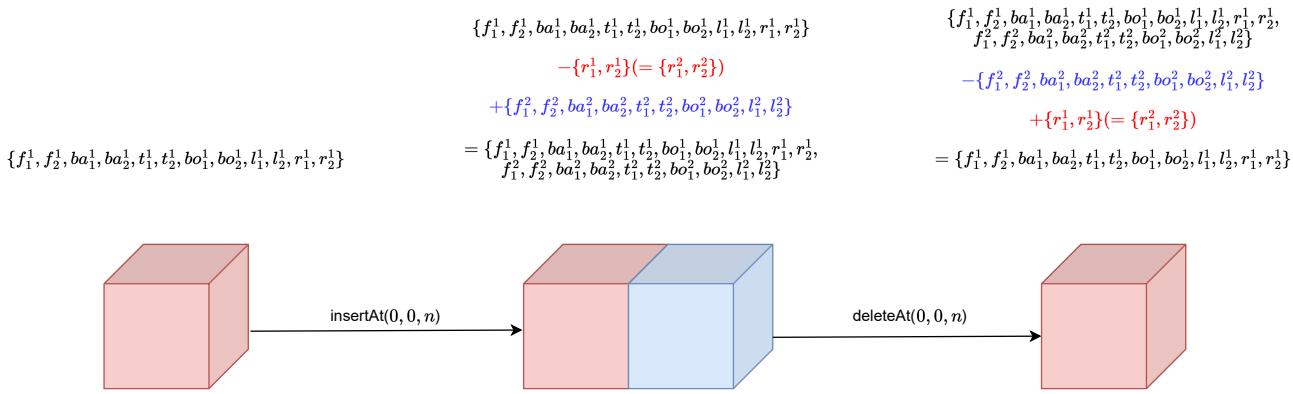

Figure 2: Maintaining the list of triangles with insert/delete

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

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

To continue the last example, let's say now we remove the cube inserted at $(0, 0, n)$ then not only do we have to remove the faces corresponding to 5 faces of the cube but we must make the left face of the cube at $(0, 0, 0)$ visible again.

With this kind of insertion or deletion, we are guaranteed that in case of model with 1M blocks, only $6 \times 2 \times 100 \times 100$ triangles will be visible at any given time. Despite this, the user may delete or insert the cube at the center of the model which is not visible through any of the sides. Therefore, we must overestimate the number of triangles in order to prevent out-of-memory errors.

For this, consider any face of the cube. Maximum of $N \times N$ quads are visible to us. But this face can be hollow from inside which can lead to a similar quad of size $(N-1) \times (N-1)$ visible from the inside. This can be further extended to get total number of "maybe-visible" (not from a visibility sense but from our algorithm which considers visibility through neighbors) will be $n^2 + (n-1)^2 + \cdots = \sum_{j=1}^{n} j^2 = n(n+1)(2n+1)/6$. Therefore, total number of triangles $= 2n(n+1)(2n+1)$ since there are 6 faces to a cube and 2 triangles/quad.

This gives maximum amount of memory we need for storing triangle data as $2 \times 100 \times 101 \times 201 \times 3 \times 3 \times 4 \approx 140$MB of RAM. The same amount will also be required to store colors. This means the maximum amount of RAM required in our approach is ≈ 280MB which is much better than ≈ 800MB requirement of naive approach.

Saving/Loading:

Saving requires us to store all the cubes in lexicographic order. This requirement is satisfied for free since C++ `std::map` stores data in lexicographic order of keys (which are `Point` instances in our case) by default ! Saving simply enumerates over all of the cubes and writes their information on every line. We also additionally store the value of N on the very first line. This value of N can be useful (although we did not implement it) to center small models into larger models and also detecting whether user is loading a larger model into smaller grid.

Loading procedure is similar to inserting a single cube except done for collection of all cubes after they have been read from the file. We also utilize the value of N read from the first line of the file to warn user about potentially loading a larger model in smaller grid size. If any of the model blocks are outside the grid boundaries, we simply ignore such blocks and notify the user.