Vivatsathorn Thitasirivit
643 21584 21
Chulalongkorn University

February 1, 2024

ASSIGNMENT 1 — Voxelization

1 Images of Objects

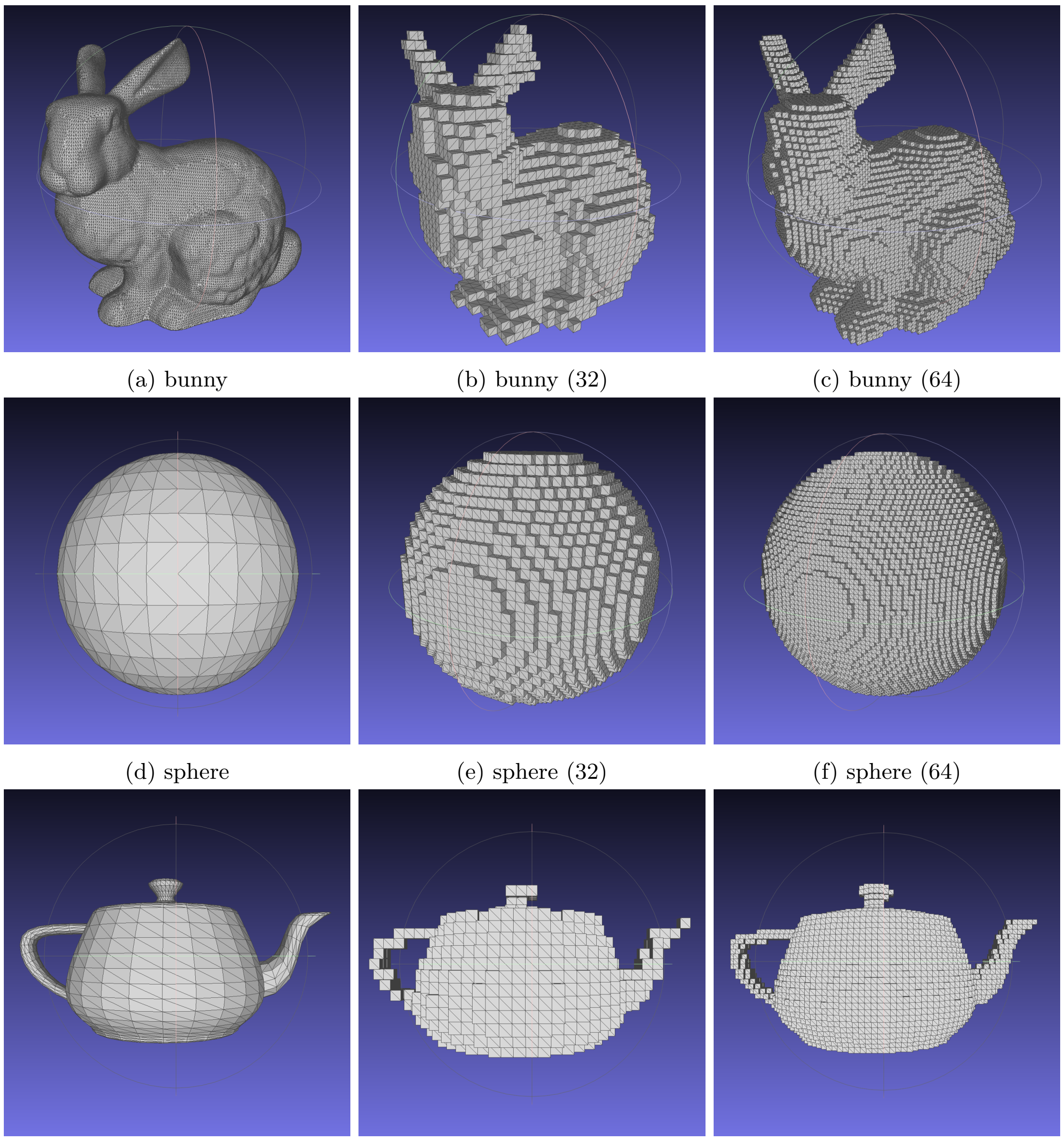

(a) bunny (b) bunny (32) (c) bunny (64)
(d) sphere (e) sphere (32) (f) sphere (64)
(g) teapot (h) teapot (32) (i) teapot (64)

Figure 1: Voxelization

2 References

I was reading on "How to check if ray intersects a triangle" and I found one article from 1997: `Fast, Minimum Storage Ray/Triangle Intersection` by Moller, T. and Trumbore, B., which provides C implementation to check ray-triangle intersection with face culling and non-culling method. I modified and added some conditions to make it applicable to the C++ voxelizer.

3 Source Code Sample

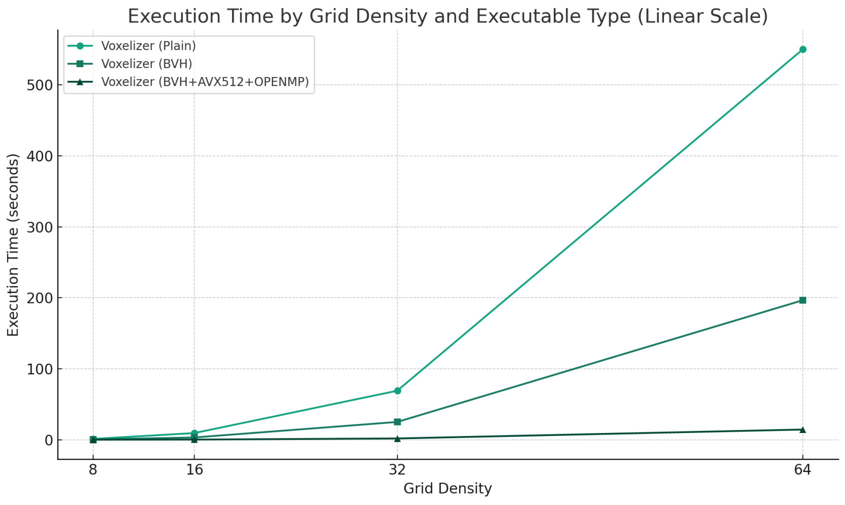
```cpp
void intersect_tri_2(CompFab::Ray &ray, CompFab::Triangle &triangle,
                     int &out, double &t, double &u, double &v) {
    // Called by int rayTriangleIntersection(CompFab::Ray &ray,
    //                                        CompFab::Triangle &triangle);
    //
    // From https://www.graphics.cornell.edu/pubs/1997/MT97.pdf
    // Two-sided face (with two direction option)
    CompFab::Vec3 E1, E2, T, P, Q;
    double det, inv_det;

    out = 0;

    E1 = triangle.m_v2 - triangle.m_v1;
    E2 = triangle.m_v3 - triangle.m_v1;

    P = ray.m_direction % E2;

    det = E1 * P;

    if (det > -EPSILON && det < EPSILON)
        return;

    inv_det = 1.0 / det;

    T = ray.m_origin - triangle.m_v1;

    u = (T * P) * inv_det;
    if (u < 0.0 || u > 1.0)
        return;

    Q = T % E1;

    v = (ray.m_direction * Q) * inv_det;
    if (v < 0.0 || u + v > 1.0)
        return;

    t = (E2 * Q) * inv_det;

    // Comment out for bidirectional ray
    if (t < 0)
        return;
    // End comment

    out = 1;
}
```

4 Extra Credits 1

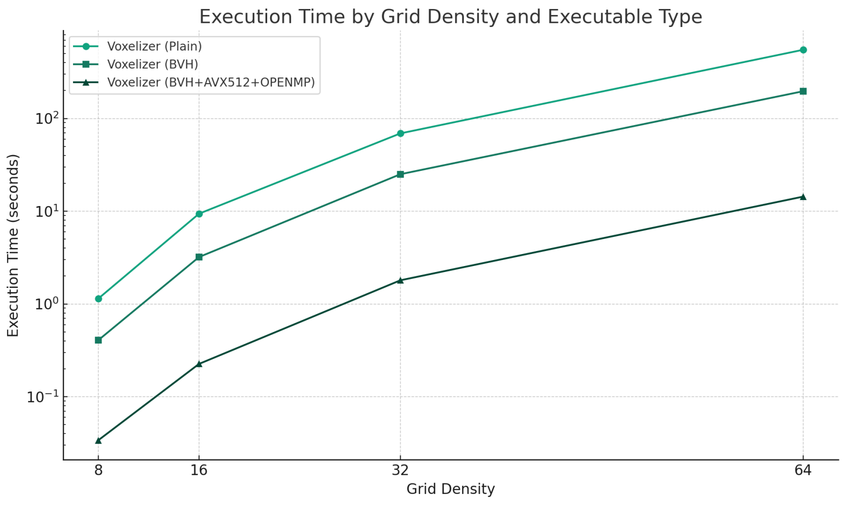
I did extra credits #1 by heavily optimizing the mesh query from $O(N)$ to $O(\log^k N)$ by implementing Boundary Volume Hierarchy (BVH) with binning construction to improve traversal and tree construction efficiency. `This article` helped me go through most of the implementation. I did write the whole BVH from scratch using basic C++ template and template metaprogramming to further optimize the runtime speed by unfolding and doing computation at compile-time. The alignment of memory allocation was also made to aid compiler's SIMD optimization (In this case, AVX-512 on AMD: `xmm`, `ymm`, and `zmm` registers were all used as checked in the output assembly). I also used OpenMP to parallelize the N^3 voxel loops (collapsed) to further improve performance on CPU.

The BVH structure can also be ported to CUDA by just changing the memory allocation and implementing device function kernel calling in theory, but I think I will leave that here.

I have tested and compared 3 versions: plain-old bruteforce, BVH, and BVH+AVX512+OPENMP and the optimization results were greatly improved. `See my source code here`.



(a) Linear graph



(b) Log-linear graph

Figure 2: Execution time - Voxel grid density (Bunny, 70k mesh)

4.1 Raw Data

```
Voxelizer (Plain) uses 1.143497 seconds for 8 grid.
Voxelizer (BVH) uses 0.407441 seconds for 8 grid.
Voxelizer (BVH+AVX512+OPENMP) uses 0.033966 seconds for 8 grid.

Voxelizer (Plain) uses 9.408084 seconds for 16 grid.
Voxelizer (BVH) uses 3.206270 seconds for 16 grid.
Voxelizer (BVH+AVX512+OPENMP) uses 0.225881 seconds for 16 grid.

Voxelizer (Plain) uses 69.052401 seconds for 32 grid.
Voxelizer (BVH) uses 25.062985 seconds for 32 grid.
Voxelizer (BVH+AVX512+OPENMP) uses 1.800340 seconds for 32 grid.

Voxelizer (Plain) uses 550.160452 seconds for 64 grid.
Voxelizer (BVH) uses 196.585359 seconds for 64 grid.
Voxelizer (BVH+AVX512+OPENMP) uses 14.368764 seconds for 64 grid.
```

5 Extra Credits 2

I did extra credits #2 (multi-ray casting) as do-able in this timeframe. I used cumulative bitwise OR to decide whether the voxel is inside or outside the boundary, which I think is naive and prone to errors as I have experimented so far. Here's some of the snippets from the source code involving multi-ray casting option.

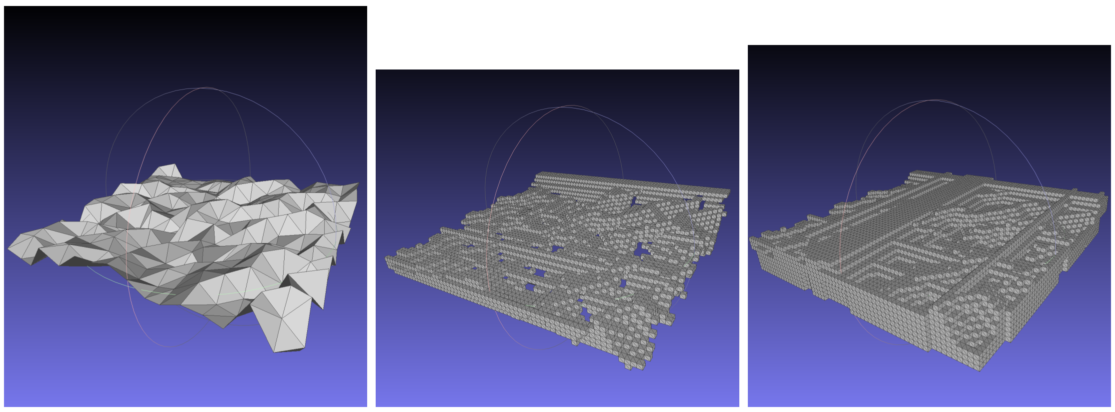
```
int main(...) {
    ...
    CompFab::Vec3 directions[3] = {
            {1., 0., 0.},
            {0., 1., 0.},
            {0., 0., 1.},
    };
    ...
    for (CompFab::Vec3 &direction: directions) {
        ...  // Iterating over each voxel
        {{{
            ...
            num_hits = numSurfaceIntersections(voxelPos, direction);
            (g_voxelGrid->m_insideArray)[k * (nx * ny) + j * ny + i] |= (num_hits % 2);
        }}}
    }
}
```

In this particular example, I used orthonormal basis (in this case, standard basis): $\{\hat{e}_X, \hat{e}_Y, \hat{e}_Z\}$
I attempted to utilize GPU acceleration via NVIDIA CUDA. I also attempted to implement OpenVDB structure into the program, but there was a lot lot of work to be done.

6 Known Problems

I did not have problems for most parts, but the obvious problem was the program is very slow as number of meshes, number of rays, and grid resolution increase (Overall knr^3). As I was doing one of the extra credits (multi-direction ray test), I used a hollow cylinder model (generated in OpenSCAD) as an input model and I saw artifacts in the output voxelized object. There are also more artifacts present in multi-ray version than uni-ray version. I suspected there were flaws in the multi-ray casting checking logic (cumulative bitwise OR).

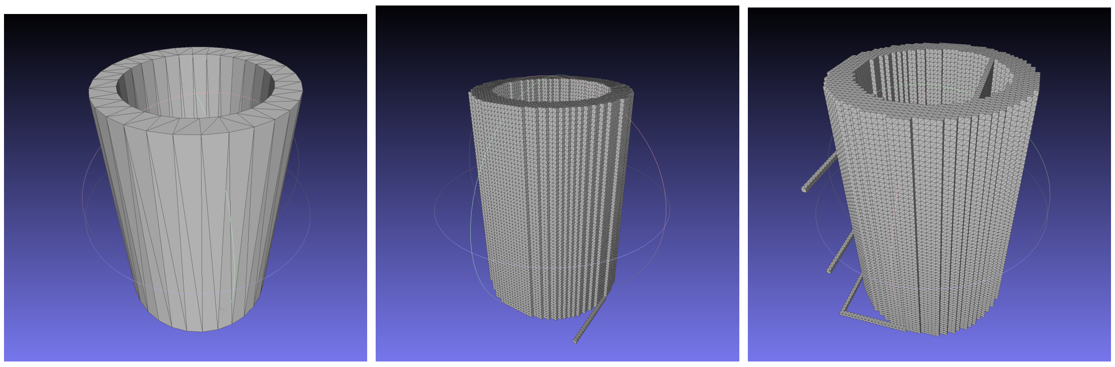
For `crumbled` model, which is a non-closed mesh object with no convex boundary to it, the multi-ray version makes it particularly solid, but that might not exactly be the goal.



(a) crumbled (b) crumbled (uni 64) (c) crumbled (multi 64)

Figure 3: Uni-ray vs Multi-ray: Crumbled

For `hollow` model, which is a closed mesh object with hole in the middle, there are some artifacts on it. At first, I thought it was the defect from the model file itself, but as I tested more, it might be from the ray casting checking algorithm. Nevertheless, overall results look promising.



(a) hollow (b) hollow (uni 64) (c) hollow (multi 64)

Figure 4: Uni-ray vs Multi-ray: Hollow

If I had more time, I would restudy the multi-ray casting logic and re-implement it in the more correct way.

7 Executable

I have modified parameters for the executable so that it can be tested easily with scripts without recompilation.

```
Usage: voxelizer <input_mesh_file> <output_mesh_file> <resolution> <multidirection?>
<input_mesh_file>       Input mesh file name
<output_mesh_file>      Output mesh file name
<resolution>            Voxelizer grid resolution, e.g., 16, 32, 64
<multidirection?>       0 for uni-directional ray casting
                        1 for tri-directional ray casting
```

Submitted by Vivatsathorn Thitasirivit on February 1, 2024.