Visualization of SDF scenes

Petr Fusek <xfusek08@stud.fit.vutbr.cz>

January 2, 2022

1 Introduction

This semester project is common for two classes: "GMU - Graphic and Multimedia Processors" (GMU) a "PGPa - Advanced Computer Graphics (in English)" (PGPa).

As a disclaimer I want to say, that the project is still work in progress and its final goal is out of scope of one semester work and it will be improved and build upon in my upcoming masters thesis. Thus so far reached results are more of "proof concept" than finished self-contained work.

1.1 Motivation

The final goal of upcomming thesis is develop a simple user-friendly 3D sculpting application using non-traditional technique for storing and rendering 3D models developed by *Media Molecule* [1] for its game *Dreams*™ which is basically a 3D game engine accessible to unexperienced non-technical users. Technology in the game is interesting because it allows to create highly detailed 3D models and scenes without worrying about polygon counts, texturing or levels-of-details.

This project is based on my research of basic principles into their technology and possible implementation of some simplified version for the PC users.

1.2 Goals

To stay in scope of the two classes, aim of this work will be a GPU-based evaluator (for GMU) and 3D renderer of resulting representation (for PGPa). The basic idea is having geometry of 3D model represented by *Constructive Solid Geometry* (CSG) of a small set of basic primitive shapes, which will be evaluated into a *Signed Distance Field*. This field will be rendered onto screen using optimized *rasterization pipeline* on the GPU.

2 Theory

Main source of technology is talk given by Alex Evans [Ale15] about technology in *Dreams*™.

2.1 Scene representation

The rendered scene is composed of *geometries* and *models*. Models in their core are an instance of a geometry positioned in world space with some potential shading data. Geometries can be shared by objects and are composed of list of *edits*. Edits are added into a list and each represents some incremental change to overall geometry. Changes are mostly additions and subtractions of some primitive volumetric shapes creating a simplified (entirely lef-leaning) CSG tree. Edit lists needs to be evaluated into a renderable data structure which is job of the *evaluator*.

This whole structure has several reasons. 3D sculpting by incrementally adding and subtracting volumes is more intuitive for the user. Resulting edit list is convenient compact representation of geometry (In theory just tents of KB for big detailed models.). And it is directly easily rendderable as composed *Signed*

[1] https://www.mediamolecule.com/

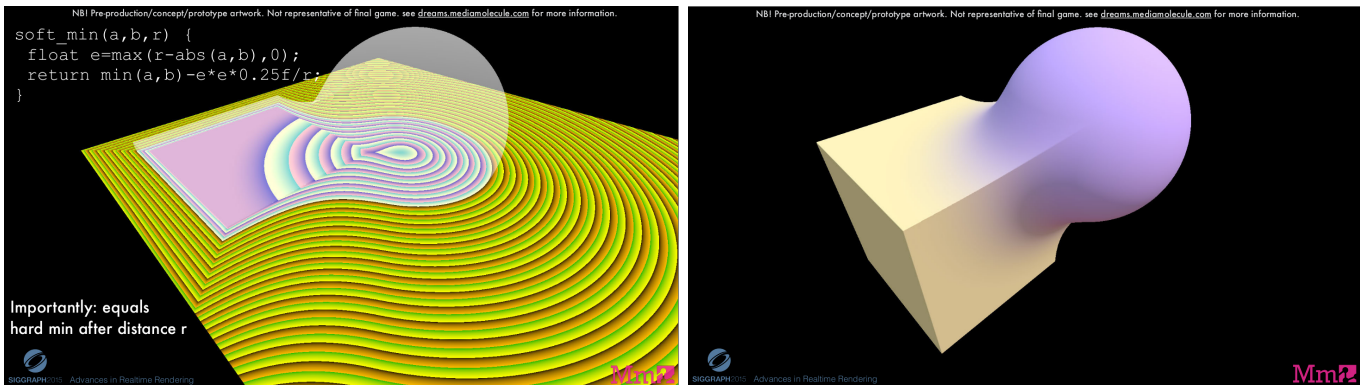

Figure 1: Left - Visualization of 3D SDF of two edits. Right - Smooth surfaces obtained by sampling the SDF. Images are taken from [Ale15]

Distance Function (SDF) by *ray-marching/sphere-tracing*, which is a popular technique used by pixel-shader programmers of *Shadertoy*[2] community. This has another advantage of implicitly representing smooth surfaces (see figure 1).

2.2 Rendering pipeline

Directly ray-marching whole complex scenes composed of large number of primitives in each pixel of the screen is to slow for real-time applications. So the whole task for rendering pipeline is to render large number of complex geometries (SDFs) onto screen in real time. Overview of the pipeline idea is on the figure 3. From the diagram we can see that evaluator will transform list of edit into a *Sparse Voxel Octree* (SVO) which will be stored in the GPUs memory. All models are rendered each frame using SVO representation and properties from currently rendered model.

2.3 Evaluator

Evaluator uses GPUs compute shader to create an SVO with brick pool into a GPU memory. Implementation of SVOs structure is mostly identical to the one used in Crassins Gigavoxels [Cra11]. Main two parts of the evaluated geometry are *Node Pool* and *Brick Pool*. Node Pool is and encoded octree representation stored in linear memory of gpu, where one node is represented by two integers. First one holds pointer to *child node tile* and second points to brick in brick pool. See figure 2.

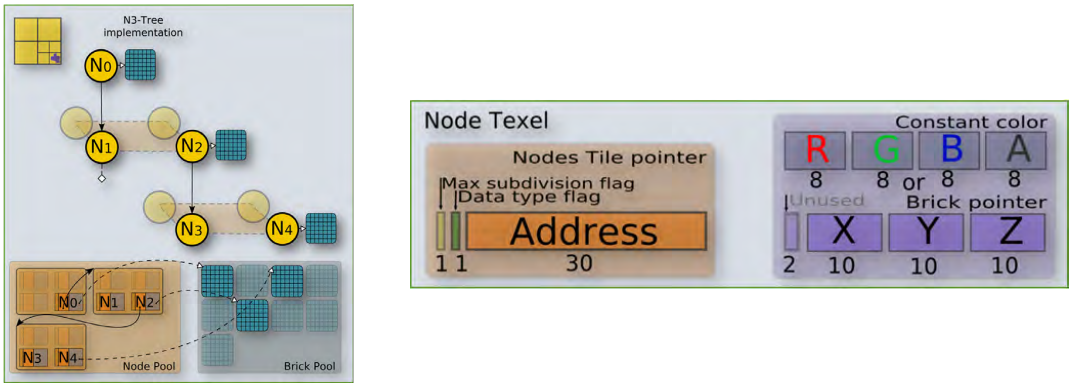

Figure 2: Sparse voxel octree structure in GPU memory. Nodes siblings are stored in tiles to which parent holds and offset pointer. Nodes may point to a brick stored in brick pool implemented using 3D texture atlas. Images are taken from [Cra11]

[2] www.shadertoy.com

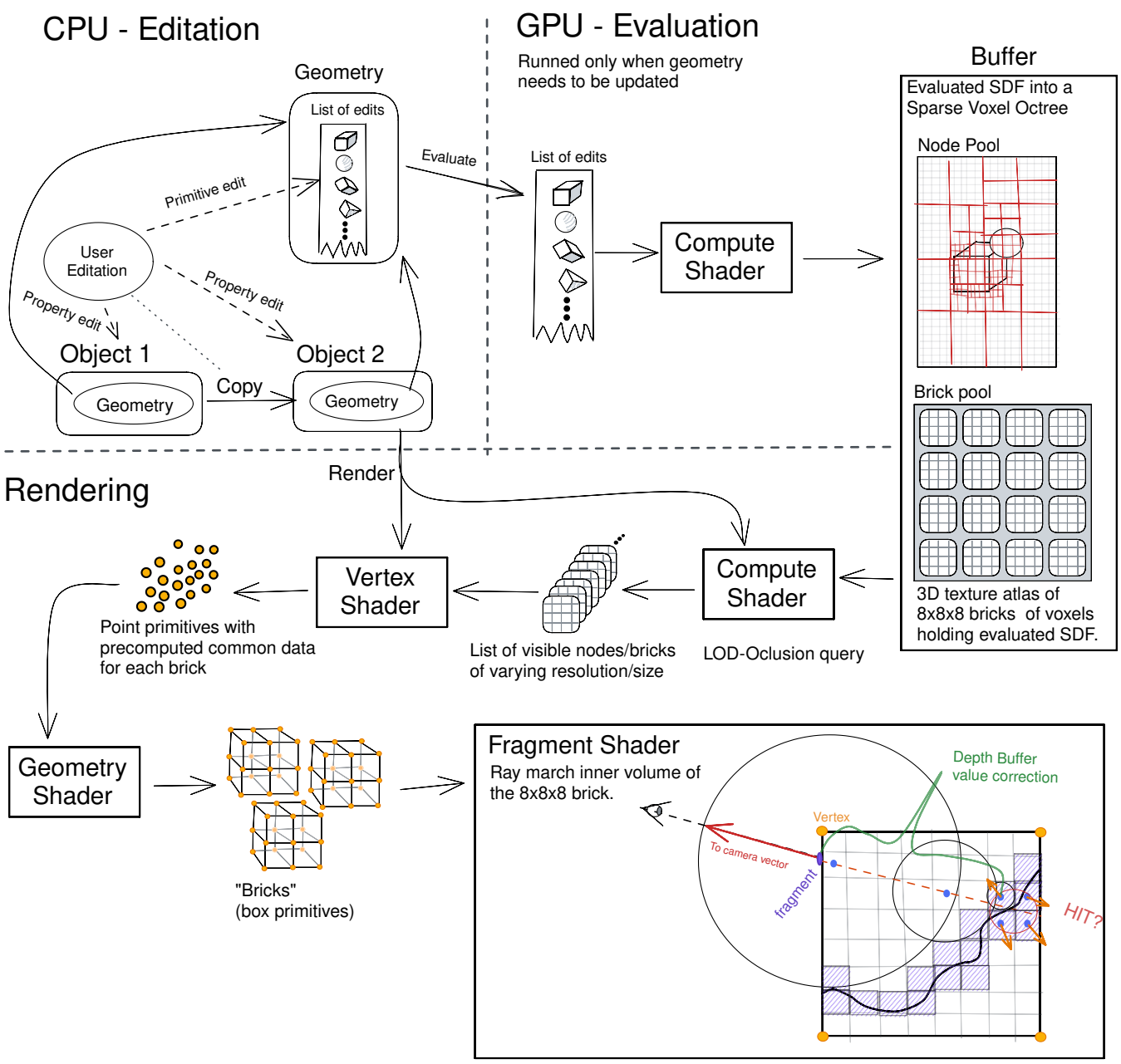

Figure 3: Idea of rendering pipeline.

Algorithm for constructing the SVO on octree was inspired by [Cra12] but it was redesigned to suite an SDF evaluation. Construction is done in Top-down approach evaluating one level of octree in one dispatch. Main rule is that currently evaluated node will be divided if it intersects with surface of the geometry, meaning that at least one voxel in potential node brick has a distance value smaller that size of that voxel. This means that when descending the levels all 512 voxels per node-brick needs to be sampled and if there is an intersection the node is marked to be divided and brick is stored into a brick pool. Therefore algorithm dispatches group of 512 threads per node in currently evaluated level and each group may spawn a new tile of 8 nodes for the next level.

2.4 Rendering

As can be seen on the diagram 3, rendering uses evaluated SVO with brick pool to render bricks of SDF data onto screen. The main strength of this approach is filtered LOD and good culling possibilities where all algorithms will work with the octree directly or with cloud of cubes in world space.

When rendering a model, its geometry is transformed to its position and nodes which holds a reference to a brick and are visible on screen are selected. The result is list of integer indices to a node pool associated with a model which could be submitted for rendering as a point cloud. On GPU side for each index are precomputed shading and transformation properties in vertex shader. Geometry shader will spawn cube to be rasterized onto screen on correct position. Fragment shader will ray-march from rasterized surface fragment until it hits a geometry surface or other side of the cube. When hit is detected, normal will is computed by sampling a gradient of the bricks SDF and for a simple lighting.

3 Implementation

Project was implemented using a `C++20` with CMake[3] using OpenGL 4.6[4], glm[5], glfw[6] libraries. Used HW was: AMD Ryzen™ 7 5800H[7] with AMD Radeon™ Graphics.

3.1 Building

Project was developed and tested on linux under PoP_OS![8] distribution. All dependencies should be included in its repository directly or as git submodules. Projects source code is available on GitHub[9].

3.2 Implementation of evaluator on GPU

Evaluator was implemented using OpenGLs compute shaders API. Final algorithm is more or less same as was proposed in theory section. The final implementation of evaluated geometry is composed of three OpenGL buffers to store octree data and one 3D texture as a brick atlas as it shown in following pseudo code:

```
struct Octree {
    Buffer<uint> nodes; // pointer to child tile + has-brick/has-child flags
    Buffer<uint> nodeData; // nodes brick coordinates in brick pool
                           // or empty/full flag
    Buffer<vec4> verticies; // (x,y,z) - position of node center in octree space
                            // w - length of brick/nodes edge
    Texture3D<float> brickPool; // of some predefined resoluton N x N x N voxels
}
```

[3] www.cmake.org
[4] www.opengl.org
[5] www.glm.g-truc.net/0.9.9/index.html
[6] www.glfw.org
[7] www.amd.com/en/products/apu/amd-ryzen-7-5800h
[8] pop.system76.com
[9] github.com/xfusek08/SDFEdit

By octree space is meant coordinate system where octree of geometry is at the center.

3.2.1 Algorithm of SVO construction

On CPU there is implemented a maximum number of subdivision for the SVO. There are two shader programs performing two task: *Level evaluator* evaluates SDFs into bricks and alocates space for nodes in next level. *Level initiator* zeros allocated memory for new nodes and computes verticies for nodes in next level. Following pseudocode describes cpu side of the algorithm.

```
Program level_evaluator;
Program level_initiator;
AtomicCounter node_count = 8; // 1st tile - 8 uninitialized allocated nodes
int level_start_node = 0;
int level_node_count = node_count;

level_initiator.setLevelStartUniform(level_start_node);
level_initiator.dispatch(1, 8); // 1 group, 8 threads per group
for (i in 0 .. maxSubdivision) {
    level_evaluator.setLevelStartUniform(level_start_node);
    level_evaluator.dispatch(level_node_count, {8, 8, 8}); // 512 threads
    int next_level_start = level_start_node + level_node_count;
    int next_level_node_count = node_count - next_level_start;
    if (next_level_node_count == 0) {
        break;
    }
    level_initiator.setLevelStartUniform(level_start_node);
    level_initiator.dispatch(level_node_count, 8);
    level_start_node = next_level_start;
    level_node_count = next_level_node_count;
}
```

`level_initiator` is not very interesting. It is run for each node with one thread for each its child and when the node has child tile, then the child nodes are zeroed and vertex is computed based on local invocation index and vertex of the parent.

The evaluation is run with 8x8x8 threads in group for each node in current level and each thread is responsible for taking a sample of the SDF. Implemented algorithm on GPU is in simplified form in following pseudo code

```
shared uint divide;
shared uint brick_index;
void main() {
    if (localIndex == 0) divide = 0; barrier();
    uint node_index   = level_start_node + workgroupID; // common to all voxels
    vec4 node_vertex  = verticies[node_index];
    vec3 voxel_center = computeVoxelCenter(node_vertex, localID);
    float voxelSize   = computeVoxelSize(node_vertex);
    float sdfValue    = sampleSDF(voxel_center);

    if (abs(sdfValue) < voxelSize) {
        atomicAdd(divide, 1);
    }
    barrier();

    if (divide == 0) {
```

```
        if (localIndex == 0) {
            nodes[node_index] = 0;
            nodeData[node_index] = 0;
        }
    } else {
        if (localIndex == 0) {
            brickIndex = atomicCounterIncrement(brickCount);
        }
        barrier();

        uvec3 brick_coords = brickCoordsFromIndex(brickIndex);
        uvec3 global_voxel_coord = CalcvoxelCoords(brick_coords, localID);
        writeToImageTexture(global_voxel_coord, sdfValue);

        if (localIndex == 0) {
            uint child_tile_index = atomicCounterAdd(node_count, 8);
            nodeData[node_index] = encodeBrickCoords(brick_coords);
            nodes[node_index] = child_tile_index;
        }
    }
}
```

For sampling SDFs is used function form Inigo Quilez web article[10]

Rendering is dependent on linear interpolation of SDF values in the 3D texture atlas. This creates problems on borders between two bricks, where values are inaccurate in all bordering voxels. This problem was solved by adding 1 voxel thick border for each brick creating ultimately 10x10x10 brick. Values for the new border voxels has to be sampled with actual value outside of the inner brick because otherwise ray marching produces unwanted artifacts along the edges of the rasterized brick, see image 4.

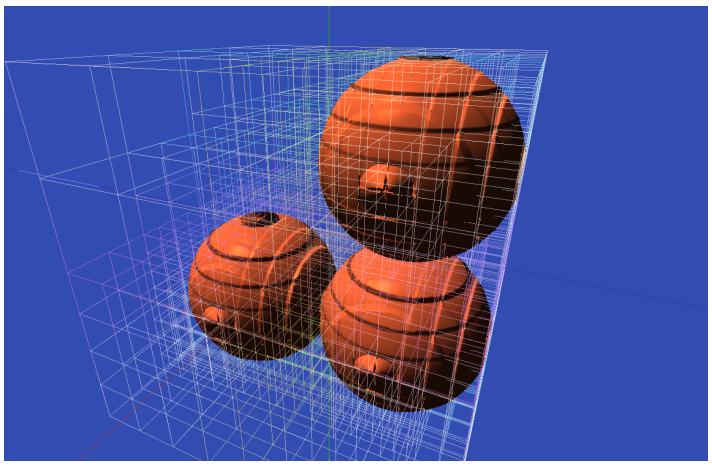

Figure 4: Ray marching artifacts when borders are simple copy of neighboring sdf values.

3.3 Rendering

Rasterization of selected bricks using vertex shader and geometry shader is straight forward. The read problem is selecting bricks for the rendering in "LOD-Oclusion query" pass. Unfortunately for this was not enough time and for each model all bricks from particular SVO level are submitted for rendering. Very crude and simple LOD method was implemented where SVO level for particular model is selected based on

[10] www.iquilezles.org/www/articles/distfunctions/distfunctions.htm

distance of the model from the camera. This allowed for testing of continuity of visual quality of model image when transitioning between different SVO levels.

Interesting part was actual ray-marching inside volume of the rasterized cube. The core algorithm is visualized on diagram 3 but it uses a linear interpolator of the texture sampler unit so sampled points are not in the center of the voxels on at exact locations where current ray step ended.

The main challenge was to correctly transform and orient marched cube and ray with camera, because the rasterized cube could be anywhere with any orientation depending on transform matrix of its model. At the end the needed transformation of brick and camera position are pre computed in previous stages so that in actual per-fragment shader only fragment position needs to be multiplied with pre-computed matrix.

Normal on hitted surface is computed as a gradient on the SDF function using a code described in [Qui].

Fragments with missed rays are discarded to make cube invisible. Last thing needed to implement for renderer to be temporary coherent was correcting a depth buffer value of the fragment which was done by projecting hit point into clip space and sampling its depth from homogenous coordinates.

4 Results

The final application is able to load scene from json file consisting of geometries and models. Then evaluate the geometries into SVO representation described above and render the models with minimal visual artifacts.

It can evaluate/resterize edits of 7 different primitives with additional “rounding” and “blending” properties. Each edit can be placed and rotated in space as well as each models. On figure 5 is screenshot of scene presenting all possible primitives rendered by the program.

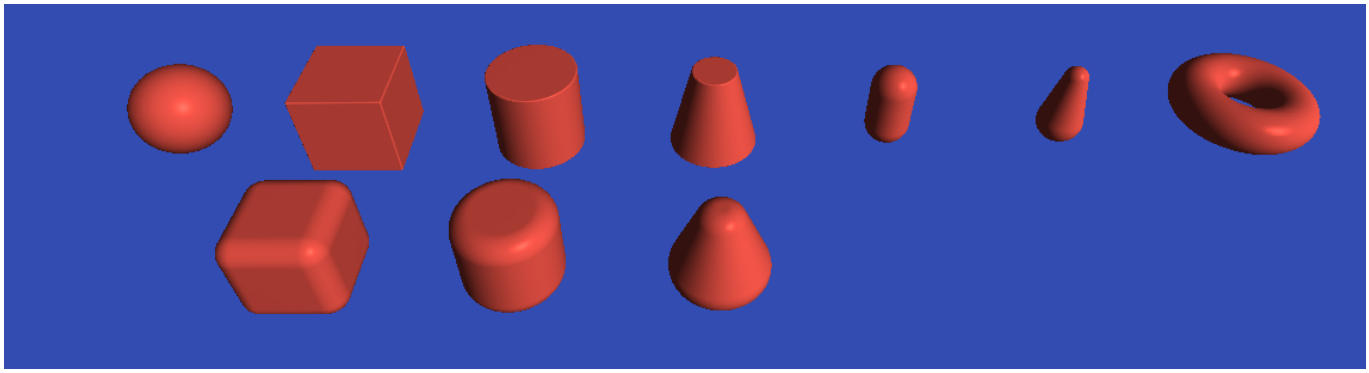

Figure 5: Sample of all possible shapes.

As a better usability demo, scene of chess board was modeled using json representation as can be seen on figure 6 and 7.

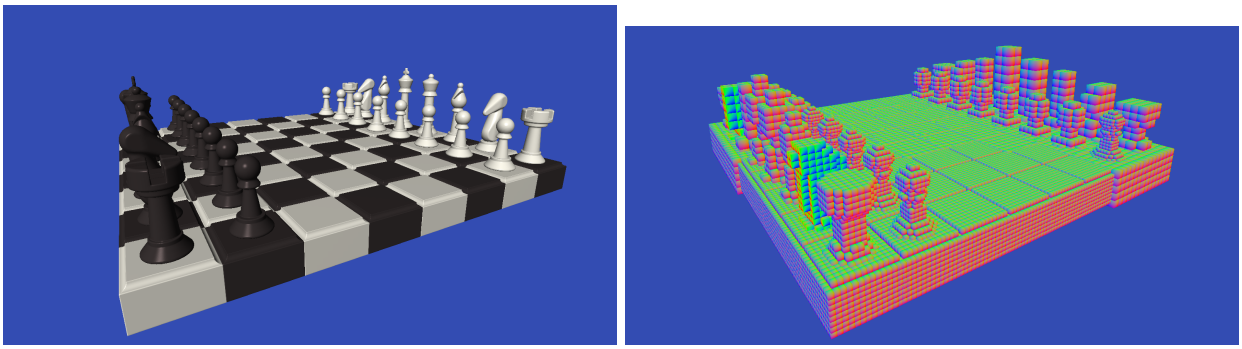

Figure 6: Chess demo scene with custom SDF models. Right - boxes rasterized without ray marching, there can be seen a different level of the SVO in further models.

There was also implemented a test for evaluator by randomly animating individual edits in geometry and re-evaluating it each frame (figure 8).

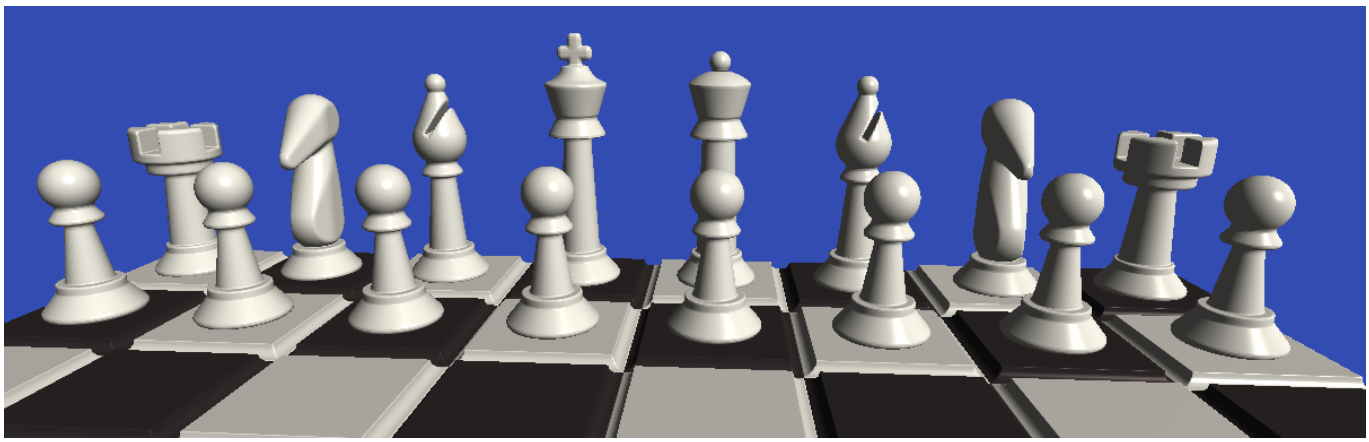

Figure 7: Detail of SDF models of chess figures.

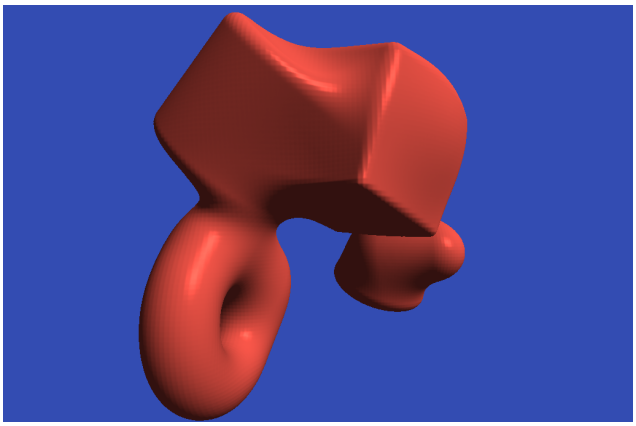

Figure 8: Evaluator real-time test.

5 Conclusion

This work managed to successfully represent scene composed of SDF models and render it using per-brick rasterization. Implemented algorithm are not optimal and it could be improved. Implemented stress test showed that evaluator cannot handle deeper levels of SVO or larger number of primitives in real time, but this is probably because of inefficient implementation using alternation of 2 programs, unoptimized implementation of SDF for primitive and no hierarchical acceleration data structure of the list of primitives. In the future there will be major refactoring of whole evaluation using single program and possibly persistent threads to construct whole SVO in one dispatch. For edit lists there can be implemented an acceleration spatial data structure. And actual SDF computation could be accelerated using efficient max-norms [Var].

Rendering works fine, more work is needed to be done in actual "LOD-Oclusion Query", where ideally lod will be per-brick and only unocluded and visible bricks will be submitted for rendering. To enhance visual quality there is a potential of colorization per-voxel, implementing shadows and global illumination models.

References

[Ale15] Evans Alex. Alex evans at umbra ignite 2015: Learning from failure, 2015. `https://www.mediamolecule.com/blog/article/siggraph_2015`.

[Cra11] Cyril Crassin. Gigavoxels: A voxel-based rendering pipeline for efficient exploration of large and detailed scenes, 2011. `https://maverick.inria.fr/Membres/Cyril.Crassin/thesis/CCrassinThesis_EN_Web.pdf`.

[Cra12] Cyril Crassin. Octree-based sparse voxelization using the gpu hardware rasterizer, 2012. https://www.seas.upenn.edu/~pcozzi/OpenGLInsights/OpenGLInsights-SparseVoxelization.pdf.

[Qui] Inigo Quilez. normals for an sdf - 2015. https://www.iquilezles.org/www/articles/normalsSDF/normalsSDF.htm.

[Var] Gokul Varadhan. Efficient max-norm distance computation and reliable voxelization. http://gamma.cs.unc.edu/RECONS/maxnorm.pdf.