A framework for textured Constructive Solid Geometry modelling

Botond János Kovács*
Supervised by: Dr. László Szécsi†

Department of Control Engineering and Information Technology
Budapest University of Technology and Economics
Budapest / Hungary

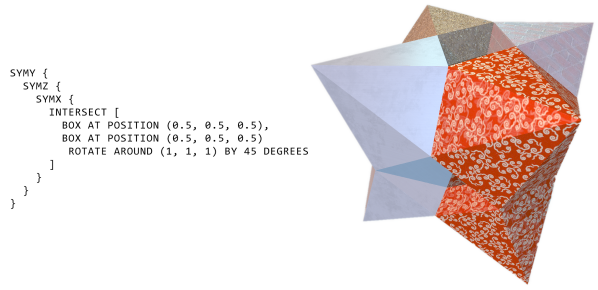


Figure 1: The framework in action. The code on the left produces the geometry shown on the right. This is not the complete code of the scene, as lighting and material parameters have been stripped for clarity. The image has been rendered using the application's production rendering feature, using sphere tracing. Texturing is achieved using triplanar mapping.

Abstract

We present a Java application and library that can be used to create triangle meshes from textual constructive solid geometry descriptions. The application can display scenes created with constructive solid geometry methods, using both triangle mesh approximation, and sphere marching. We show how GLSL code generators enable the portability of the signed distance functions, material property evaluation functions, and lighting contribution calculations between the different stages of the rendering pipeline, even between pipelines using different rendering techniques.

Keywords: Code generation, Optimization, Computer Graphics

1 Introduction

Three-dimensional scenes are described using many different types of objects, including geometric shapes, materials, or light sources. In computer graphics applications, we combine these objects to present an image of the scene they describe. Three-dimensional modelling is the act of creating the geometry, and there are multiple ways to create models. In production films or video games, triangle meshes are often used, and software dedicated to creating triangle mesh models operates on the polygons of these meshes. On the other hand, CAD software, often used in mechanical engineering or architectural engineering use constructive solid geometry as opposed to polygonal meshes. The CSG representation of these models also proves to be useful for performing simulations, or collision detection.

Constructive Solid Geometry[10] models may be presented using direct techniques, such as ray tracing, or indirect techniques, such as triangle mesh approximation. When using indirect techniques, a triangle mesh is usually required. Using interface-extraction techniques a triangle mesh can be produced for constructive solid geometry models in a discretised space. For large enough data adaptive grids can be used to alleviate the memory and computational time requirements of the interface-extraction, while introducing some preprocessing overhead. Since these algorithms have VRAM requirements in the gigabyte range, for optimal runtimes GPU pipelining must be utilised between the computational stages to avoid unnecessary copying of the data between RAM and VRAM.

In this paper, we show and compare methods for visualising textured CSG models. After the brief introduction of direct and indirect visualisation techniques, we describe our own formulation of the problem, and outline the details of our solution. Our application is modular in the sense that there exist multiple exchangeable implementations of the different stages of the pipelines. This allows us to compare different types of grids, interface-extraction algorithms or rendering techniques. We created a simple language to store the CSG operations, and the material and lighting properties of a scene. These scenes may be presented using one of the renderers, or exported as triangle meshes to Wavefront OBJ or glTF files. Most stages of these tasks are GPU-accelerated, and allow for efficient hardware utilisation.

*botondjanoskovacs@gmail.com
†szecsi@iit.bme.hu

2 Previous work

2.1 Direct visualisation

There are multiple ways of turning a constructive solid geometry description into a renderable format. We can achieve the best quality using direct visualisation techniques, such as *ray tracing*[11].

With ray tracing, we shoot imaginary rays through our image plane, and check for intersections with the geometry. To do so, we must find the mathematical equations of our objects, and solve the equation system formulated from the equations of these objects, and the ray's equation.

Another approach is *ray marching*, finding this intersection point by starting from the ray's origin, and taking small steps along the ray. At each step, we check if the point is inside any of the objects in the scene. If at any point we detect to have entered a shape, we can trace the actual intersection point by taking smaller steps backwards, eventually leaving the object again.

This process can be optimized if the distance function of our objects is known[1]. The distance function maps the shortest distance to the surface of the object to the input point. If at each step we evaluate this function, and take the minimum distance to all of our objects, a step with this distance can be taken in any direction from the sampling point, including our ray's direction. This improvement can dramatically decrease the number of steps required to find the surface intersection point.

2.2 Indirect visualisation

There are also multiple indirect methods for visualising these kinds of surfaces.

Point-based methods[13] may use a large number of particles that are attracted towards the surface, and pushed by other particles. These particles may than be drawn as billboards to visualise the surface.

Another approach is to create a triangle mesh from the surface: by discretising the space into grid cells, the *marching cubes*[5] or the *dual-contouring*[2] algorithms can produce an approximation of the surface constructed of triangles. Both algorithms look for intersection points between the surface and the edges of the voxels. The marching cubes algorithm places points onto these edges, by interpolating them according to the distance values measured at the endpoints of the edge, and connects these points placed onto the voxel edges to form triangles. The dual-contouring algorithm uses a single point per voxel. If an edge of the voxel intersects the surface, these single points placed in the neighbouring voxels of the intersecting edge are connected to form a quad. The original dual-contouring algorithm finds the optimal points inside the voxels, by assigning a cost function to each edge intersection point, and finding the parameters that minimise the system of equations formulated from these cost functions. Our approach to finding the optimal point is to use the average of the edge intersection points.

2.3 Scene model

There exist many file formats for storing both triangle meshes and CSG models. For polygonal models, some of the most commonly used formats include the *Stanford PLY*, *Wavefront OBJ*, or the *glTF* file formats. For CSG models, OpenSCAD[3] introduced a functional, human readable language, that supports using variables, performing transformations, and operations on primitive objects. It also provides support for coloring the defined primitives, or CSG tree nodes. Our textual representation format is inspired by the OpenSCAD language, but uses a declarative syntax as opposed to functional programming.

2.4 Signed Distance Functions

One of the most important features of our CSG evaluator, is that it uses the *signed distance functions* of the primitive shapes, operators and transformations it supports. This simplifies the implementation of the *union*, *difference* and *intersection* operators, and allows us to apply transformations by applying the inverse of the transformation to the input point[7]. Methods for approximating distance to any implicit surface[6], or fractals[8] exist, and although they do not produce the exact Euclidean distance, the approximation is good enough to allow for sphere tracing, or triangle-mesh approximation.

3 Our work

We created a framework for describing three-dimensional scenes that contain *models*, *materials*, and *light sources*. The framework can be extended with new CSG operations, material types, or light source types. We use *signed distance functions* to model the surfaces of the objects. We use *code generators* to evaluate the CSG operation tree, and generate GLSL code that implements the total signed distance function of the CSG tree. The framework also requires CSG operators (and all other evaluable components) to provide a CPU implementation of the algorithms they realize. We designed the interface of the framework to allow for integration on two abstraction levels:

- Integrating on the **application level** means writing SurfaceLang code, or another application that *generates SurfaceLang* code, and using our application or high-level framework methods to *create a triangle mesh* of the scene, or to create a rendered image of the scene.
- Integrating on the **framework level** means using our framework's *code generator* (or evaluator) in an application in order to realize any features relying on the signed distance function, material function, or illumination function of the scene.

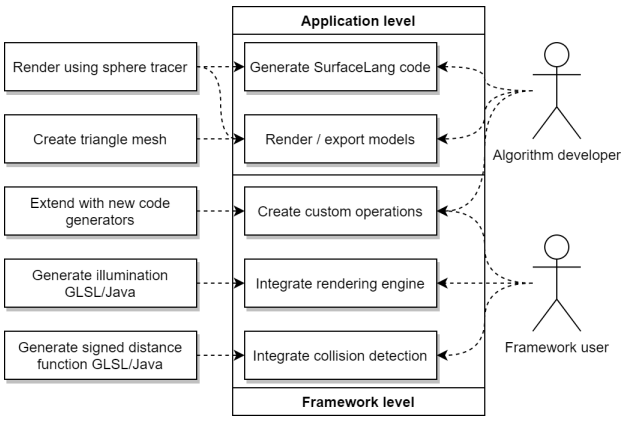


Figure 2: Overview of the framework use-cases

These abstraction levels are illustrated in 2.

3.1 Scene representation

The input of the application is a textual representation of the scene, written in the application's own language (see Figure 3). This text is parsed, and a graph of the different scene objects is created. There may be shapes, materials or light sources in the scene.

Shapes expose a signed distance function, which is used both in the sphere marching renderer, and the voxelization. Shapes are CSG operation nodes, that fall into one of three categories: operations, transformations, and primitive shapes. Signed distance functions are functions that take a point in the three-dimensional space as input, and output a single signed value, which is the distance to the closest point on the shape's surface. SDFs make a distinction between inner and outer points, with negative distance values being considered as being inside the shape, and positive values outside the shape.

The application uses 3D texturing, where **materials** define a boundary (a shape with a signed distance function) and certain material properties. When evaluating diffuse or specular components of the material at the surface point, the properties defined by the material are applied to the inner points of the boundary. Constant materials assign the same constant properties to all points inside the boundary values. Another material type in our solution is the triplanar material, which uses triplanar mapping[4] to calculate the diffuse and specular components of the material from the corresponding input image textures.

Light sources expose their contribution function, which can be evaluated at the surface points with the material properties provided as input. Light source contribution is calculated using the same set of calculations both in the triangle mesh renderer, and the sphere marching renderer. The light source contribution function implementation is expected to include shadowing, although this is currently only implemented correctly in the sphere marching renderer.

```
light ambient {
    energy: (0.2, 0.2, 0.2)
}

light directional {
    energy: (0.6, 0.4, 0.15),
    direction: (0.7071, 0.5, -0.5)
}

light directional {
    energy: (0.2, 0.35, 0.4),
    direction: (0.6325, 0.4472, 0.6325)
}

material constant {
    boundary: EVERYWHERE ,
    diffuse: (0.5, 0.5, 0.5),
    shininess: 40
}

SUBTRACT {
    A: SPHERE  AT  POSITION (-0.5, 0, 0),
    B: SPHERE  AT  POSITION (0.5, 0, 0)
}
```

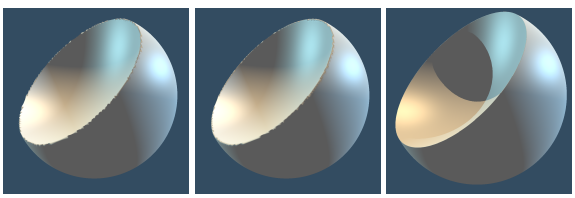
Figure 3: A simple scene represented in the application's own language

3.2 Scene portability

Since scenes carry information that is required in multiple stages of the visualisation pipeline, the framework requires components, that realize a certain algorithm, and can be executed both on the CPU and the GPU. We called these objects **evaluators**, and they have the single responsibility of providing both the CPU and the GPU implementation of a given function. Surfaces, or shapes are a type of evaluators, as they implement the signed distance function of the encapsulated body. The CPU implementation of evaluators is available as a simple callable method, while the GPU implementation is always a **code generator**, which is a function that is expected to return the GLSL code required to implement the algorithm.

3.3 Visualisation

Our framework supports the *direct* and *indirect* visualisation of scenes. The different supported techniques are demonstrated in Figure 4, and the next sections describe how our visualisation pipelines work.



(a) (b) (c)

Figure 4: Visualisation of the example scene described by Figure 3. Figure 4a) shows the marching cubes contour, Figure 4b) shows the dual-contouring result, and Figure 4c shows the sphere traced image.

3.4 Creating triangle meshes

When using the framework, triangle meshes can be created by executing the following process (illustrated in Figure 5):

- **1. Voxelization**: The signed distance field must be *discretized*, meaning it must be *evaluated* at certain points in space, and the results are saved in the *voxel storage*.
- **2. Interface-extraction**: Using the *voxelized data*, the *triangle mesh* of the data is created, which contains *vertices* and *faces*.

The resulting triangle mesh can be used in a forward or deferred rendering pipeline to create and image of the object.

Voxel storage objects define an interface for iterating voxels, and writing voxels: this enables us to create exchangeable voxel storage implementations, such as a *uniform grid*, or an *octree*. In the *interface-extraction* stage the triangle mesh data is generated from the voxels. The application supports interface-extraction using the *marching cubes* or the *surface nets* algorithm. The presentation stage is a simple forward renderer. By implementing constructive solid geometry operations in the form of signed distance functions, this process creates a mathematically more correct version of the models than methods that rely on creating a triangle mesh of the primitives beforehand.

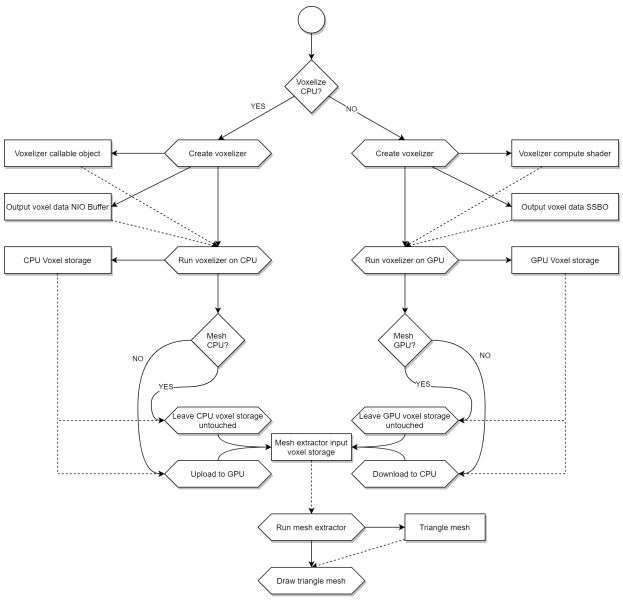


Figure 5: Overview of the triangle mesh pipeline

This Voxel interface makes no assumptions about the dimensions of the grid cell it corresponds to. This means, that no modifications are required to the interface-extraction algorithm in order to make it work in an adaptive grid. This comes with some runtime overhead, as the length of edges must be calculated for each voxel.

3.5 Sphere tracing

When rendering using the sphere tracing renderer, the signed distance function of the scene, as well as the lighting calculations and material property calculations are placed into a single shader program, that is executed for every pixel of the view. The renderer shoots a ray through the image plane for every filled pixel, and calculates hit points using the sphere marching algorithm. Normal vectors at the hit points are approximated using the central differences method. The renderer is capable of approximating soft shadows by using the distance values returned by the scene's SDF[9].

3.6 Hardware acceleration

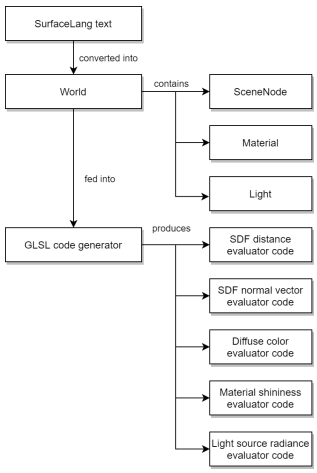
We use **code generators** to generate *GLSL code*, that implements a given *evaluable*. Figure 6 illustrates which parts of the algorithms are replaced with evaluables. These components are also implemented on the CPU, meaning that voxelization, interface-extraction or even rendering can also be implemented in CPU-only solutions.

We provided **GPU implementation** of the voxelization, and the marching cubes interface-extraction algorithm in the form of OpenGL compute shaders. With this combination, we achieve an optimal graphics pipeline, as no transfer is required between the RAM and VRAM during the process.

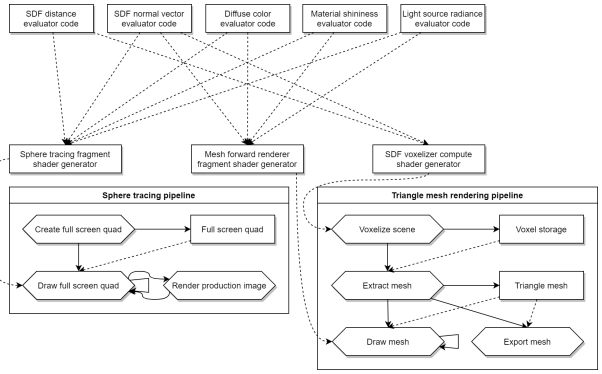
When using the surface nets algorithm, GPU voxelization can still be utilised to improve performance, but the texture contents must be copied to RAM before running the CPU-side interface extraction. Our interpretation of the voxels allows us to efficiently copy voxel data between RAM and VRAM, as no transformation of the data is needed. Our Voxel interface exposes methods that allow reading the position, distance and normal data of its 8 corners directly from the buffer copied from VRAM, and also allow for modifying data in these buffers that can be directly copied to VRAM. Data is stored in large Java NIO Buffers, continuously, and without duplicate voxel corners. Voxels are merely a view for this data, that cache the index of its 8 corners, and every time a corner read/write is requested, the operation is carried out in the NIO buffer directly at the cached corner index.

3.7 Texturing

When creating triangle meshes of parametric surfaces by wrapping discretized planes around them, the so called UV map of the resulting mesh is also implicitly generated, allowing for easy texturing of the generated surface. When the interface-extraction algorithm is running in a volume however, there exists no trivial point-to-surface mapping, which makes texturing the generated triangles a hard problem. We propose 3D texturing to solve the problem. With 3D texturing, materials define a boundary volume, and a set of material properties. These material properties are



(a) Relation between SurfaceLang, the *World* interface, and the *GLSL code generators*



(b) Relation between the types of *generated GLSL code*, and the *pipelines* that depend on them.

Figure 6: Visualisations of what becomes generated code, and how is it used.

only valid inside the boundary volume of the material. When a given point in space can be located in more than one material's boundary volume, the first evaluated material's properties are considered to be applicable to the point.

We implemented two types of materials in the system: constant and triplanar materials. Constant materials apply the same predefined set of material properties inside their bounding volume, whereas triplanar materials use triplanar sampling of a 2D image texture to calculate material properties. Using triplanar sampling, we can project texture images onto our surfaces without the UV information, using only the surface world position and normal vectors. In triplanar mapping we sample our image texture three times, by using the projection of the world position vector onto the XY, XZ and YZ planes as UV coordinates. The normal vector of the surface essentially tells us which plane's image should contribute more to the given pixel, thus the normal vector's components are used as weights for blending the three colors.

4 Implementation

Our application is written using the Java 11 language, and uses Gradle for building and dependency management. We use the LWJGL library for interfacing with OpenGL and GLFW. The program can be run on Windows, Linux or OS X computers that support OpenGL 4.6. The application also comes with an editor for the scene files, that allows for editing scene objects using a GUI, or editing the code of the scene in a syntax-highlighting code editor.

4.1 SurfaceLang

SurfaceLang is the language created to represent CSG scenes. We use the ANTLR4 tool to generate the lexer and parser source code from the grammar definition files, and this process is automated by using the `antlr` Gradle plugin. SurfaceLang supports the following language features:

- **Scene nodes** are the geometric shapes in the scene. Scene nodes reference a *node template* that defines the input properties and the SDF of a CSG primitive shape, operation, or transformation. Scene nodes may have *children*, or *named children*.
- **Materializers** are the objects that apply certain material properties inside a bounding volume. The *boundary* is a scene node, and the *type* of the material may be *constant* or *triplanar*. Different types of materials come with different sets of properties.
- **Light sources** can be defined. Light sources have a *type* (currently *ambient* and *directional* are supported), and some *radiant power density* (3-component float vector) associated with them.
- **Prefabs** are node templates defined by the user. They take a single scene node inside their body, and can be used as node templates in the scene.
- **Resource textures** are the texture files that can be used by triplanar materials.

4.2 GLSL Poet

GLSL Poet is a module in our application that provides the GLSLStatement class, as well as its implementations. It also provides syntactic sugar in the form of public static helper methods, that can be used in the classes of signed distance functions, materials or light sources like a domain specific language to aid in the GLSL code generation. The syntax tree of GLSL programs is modelled as a set of classes, that each contain a single function called render. This function is responsible for generating the GLSL code that is represented by the instance of the class. *Code generators* are expected to return a list of *GLGL statement objects*, and this list of statements is considered to be the

implementation of the underlying algorithm. Figure 6b illustrates how this allows us to use the same implementation of an algorithm in different pipelines, for example how the *direct illumination* of a light source can be calculated in the sphere tracing renderer, or the triangle mesh forward renderer.

5 Results

We created many test scenes to demonstrate the capabilities of the CSG evaluator and the sphere tracing renderer. The textual representation allows for high level mesh generator programs, written in popular programming languages. We chose JavaScript to implement two simple mesh generator use-cases.

5.1 Trees

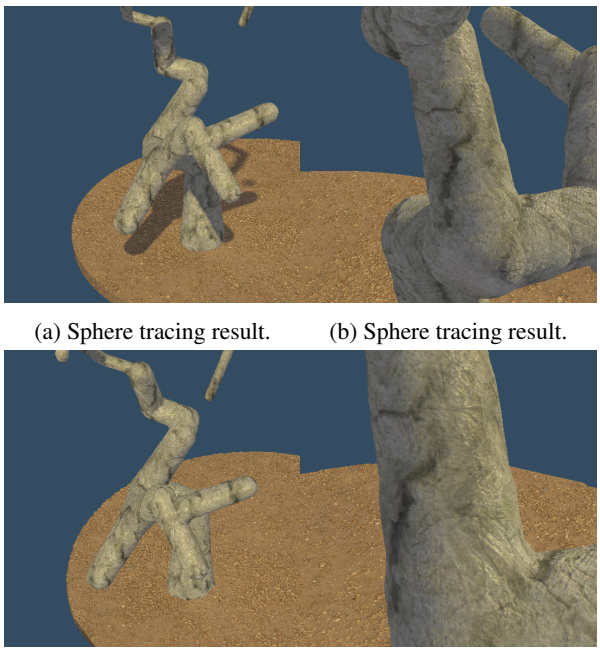
We employed the *Space Colonization Algorithm*[12] to generate branching tree-like structures. This program models the generated branch segments as `CAPSULE_LINE` nodes in the CSG tree, and its output is a SurfaceLang file. Figure 7 shows how a generated model looks like in our application. In the example, our program created 112 `capsule_line` objects, which represent the structure of the tree. We measured *frame rendering time* for both the triangle mesh renderer, and the sphere tracing renderer, as well as the *voxelization* and *triangle mesh extraction* time. This demo also demonstrates how scenes can be optimized: by placing the entire tree structure into a `gate` operator, we can skip the evaluation of the entire branching structure if our point is far enough. In this case, the sphere tracing renderer needs to evaluate the 112 capsules only in a small set of pixels, that actually contain the object (or are close enough). That is why we measured the rendering time of the sphere tracer in both the *short distance* and *long distance* cases. Our measurements can be found in Table 1 and Table 2.

Test Case	**Average frame time**
Sphere tracing close-up	99 ms
Sphere tracing far	42 ms
128x128x128 Marching cubes	0.7 ms
256x256x256 Marching cubes	0.8 ms

Table 1: Frame rendering times while running the tree demo.

Resolution	$T_{Voxelize}(ms)$	$T_{Extract}(ms)$	**Triangles**
128^3	480 ms	85 ms	70340
256^3	3161 ms	327 ms	284564

Table 2: Interface-extraction and voxelization performance while running the tree demo.



(a) Sphere tracing result. (b) Sphere tracing result.

(c) Marching cubes result. (d) Marching cubes result.

Figure 7: The tree demo with different renderers.

5.2 Rooms

Using the prefab feature and JavaScript, we created a simple *room generator*, that creates a 2D tile-map of predefined room structures, and outputs the prefabs corresponding to the different predefined room types into a SurfaceLang file. Figure 8a shows how the demo scene is rendered using the application. Tables 3, 4 and 5 summarizes our measurements of the room demo, including frame render times, and runtimes of the voxelization, and interface extraction.

Test Case	**Average frame time**
Sphere tracing	75.2 ms
127x127x127 Surface nets	3.7 ms
255x255x255 Marching cubes	3.9 ms

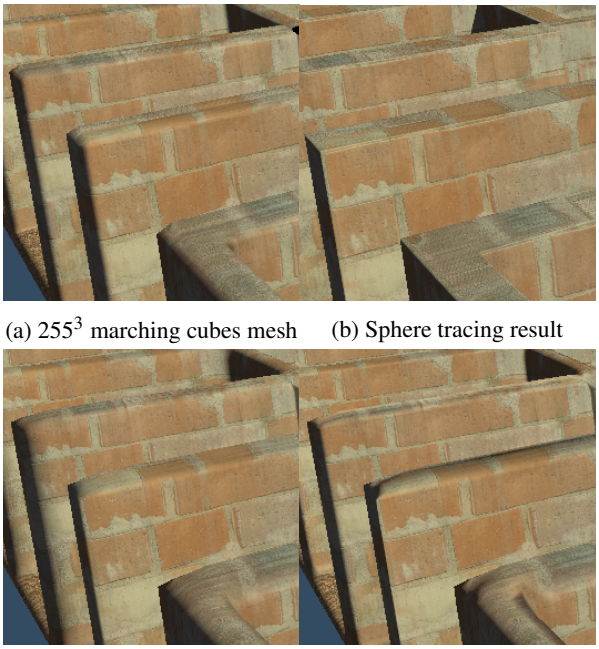
Table 3: Frame rendering times for the room demo

5.3 Smooth shadows

Another aspect we demonstrate is the soft shadowing technique used by the sphere tracing renderer. When using the soft shadowing option, a constant parameter k can be changed to make the shadows softer or sharper. Figure 9 shows how the renderer handles soft shadows.

5.4 Comparison

We compare the performance, quality, and use of our modelling framework to *OpenSCAD* and *Blender*. **OpenSCAD** - marketed as "*The programmers solid 3D CAD*



(a) 255^3 marching cubes mesh (b) Sphere tracing result

(c) 127^3 surface nets mesh, normal resampling *off* (d) 127^3 surface nets mesh, normal resampling *on*

Figure 8: The room demo scene.

Resolution	**Platf**	**Algo**	**R?**	**T**(*ms*)	**Triangles**
127^3	CPU	SN	*no*	2403	133740
127^3	CPU	SN	*yes*	78336	133740
255^3	GPU	MC	*no*	327	541974
160^3	GPU	MC	*yes*	2488	261006
255^3	CPU	MC	*no*	2552	541974
255^3	CPU	MC	*yes*	47531	541974

Table 4: Inteface-extraction performance while running the room demo. *R?* means if *normal resampling* was enabled during the benchmark. Normals may be resampled from the scene's SDF at the points where vertices were placed by an interface-extraction algorithm, this generates better normal vectors. It also has a high impact on voxelization runtime, as the SDF is evaluated 6 times during normal approximation.

Resolution	**Platf**	**Runtime**(*ms*)
127x127x127	CPU	97449 ms
160x160x160	GPU	2426 ms
255x255x255	GPU	9392 ms

Table 5: Voxelization performance while running the room demo

modeller" - is an application, that allows for modelling 3-dimensional objects in a functional programming language, visualising these objects, and saving these models as *STL* files. This file format is widely used for modelling 3D-printable objects. **Blender** is an open-source 3-dimensional modelling, animating and rendering software, designed to be used by 3D artists. The application is widely used in the game, and production film industries.

The usage of our application is comparable to OpenSCAD. We use a *declarative* language to store the scenes, and currently do not have support for variables, or loops. The function paradigm of OpenSCAD may be realised with *prefabs*. OpenSCAD has basic support for coloring the object, but SurfaceLang has support for not only describing the geometry, but also its material properties, and light sources in the scene. This makes the available set of tools comparable to Blender instead, but lacks support features such as animation.

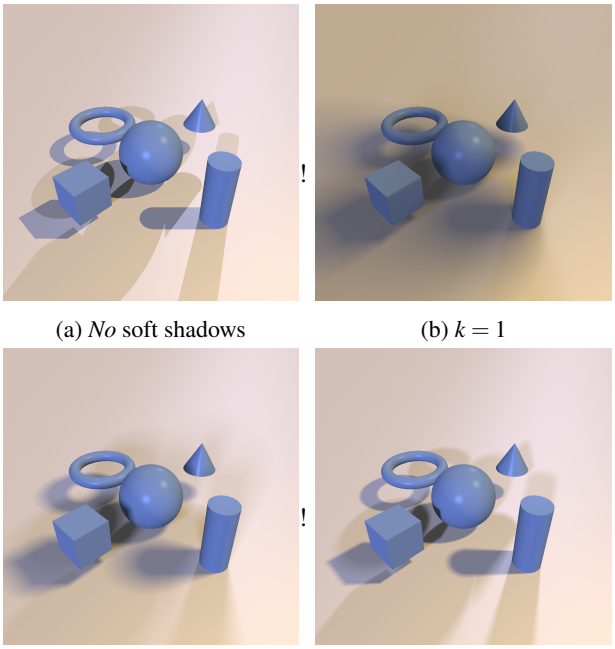
In Blender, CSG modelling can be achieved by using the *Boolean* operator, while OpenSCAD is built around the concept of CSG modelling. Both applications approximate primitives with triangle meshes, and carry out CSG operations on these meshes. Our framework on the other hand handles the whole CSG tree as a single signed distance function, which means there is no intermediate approximation of the results, and when using large resolution grids, or direct visualisation, the mathematically correct surface is presented.

6 Conclusions

Modelling with CSG, as opposed to polygon meshes is sometimes a more intuitive way of creating geometry for scenes. Using SDFs, the process of turning a CSG scene into a rendered image, or a triangle mesh is simple and precise. The pipeline allows for parallelization and efficient usage of the graphics pipeline, which makes polygonization and voxelization orders of magnitude faster than a naive CPU implementation, which allows for interactive modelling workflows. We have also shown, that sphere tracing is a viable technique for displaying CSG models in good quality, in real-time. Our framework is extendable with user implementations of CSG primitives, operators or transformations, but comes without the usual overhead of extensibility, as the resulting evaluator code is generated, and creates no indirections, or unnecessary function calls in the evaluation pipeline.

6.1 Limitations

There are a number of known bugs in the software. One of the most notable problems in its current state is the fact that during the dual contouring or marching cubes algorithms the SDF is unknown, and these algorithms can only use the input voxel data to generate triangles. This means, that surface normals are trilinearly interpolated between surface-edge intersection points, which in some cases leads to poor normals in the output mesh. Our software only uses uniform grids, and this substantially increases the runtime and memory requirements of our pro-



(a) *No* soft shadows (b) $k = 1$

(c) $k = 4$ (d) $k = 16$

Figure 9: The soft shadowing demo scene.

gram, compared to industry-standard CAD modelling solutions.

6.2 Future work

We aim to use the framework as a foundation for developing geometric effects, such as distortion. Another possible future use-case of the framework is the development of high-level mesh generator algorithms: these programs output SurfaceLang files, that can be processed by out framework. For this use-case however, a command line interface would need to be created, and headless operation should be supported to allow for integration into server-side applications.

Another future goal is to provide a *virtual machine* for CSG nodes, and instead of having to program both the `CPUEvaluator` and `GPUEvaluator` implementation of the shape, operator or transformation, we could implement these functions by creating a single evaluable expression. To accomplish this, a large number of operations realized in the GLSL language must be supported, such as *floating point arithmetics* or *matrix multiplication*. The virtual machine could be extended in a way to allow for *interpreted implementations*, which would mean that the SDF calculation can be embedded in the SurfaceLang file.

Optimisations, such as reordering of the CSG tree, or the introduction of *bounding volume hierarchies* can have great impact on signed distance function evaluation time, and all visualisation pipelines depend on the SDFs. Both sphere tracing and the voxelization performance could be improved by optimising the CSG tree before generating the GLSL code, or the CPU evaluator object. Recursion can also be avoided by using code generation to create the Java classes in runtime that evaluate the operation tree, which leads to reduced stack usage, and faster CPU evaluators.

References

[1] John C Hart. Sphere tracing: A geometric method for the antialiased ray tracing of implicit surfaces. *The Visual Computer*, 12(10):527–545, 1996.

[2] Tao Ju, Frank Losasso, Scott Schaefer, and Joe Warren. Dual contouring of hermite data. In *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, pages 339–346, 2002.

[3] Marius Kintel. *The OpenSCAD Language Reference*, 2020.

[4] Eric Stephen Lengyel and John D Owens. *Voxel-based terrain for real-time virtual simulations*. Citeseer, 2010.

[5] William E Lorensen and Harvey E Cline. Marching cubes: A high resolution 3d surface construction algorithm. *ACM siggraph computer graphics*, 21(4):163–169, 1987.

[6] Inigo Quilez. distance estimation. `https://iquilezles.org/www/articles/distance/distance.htm`.

[7] Inigo Quilez. distance functions. `https://iquilezles.org/www/articles/distfunctions/distfunctions.htm`.

[8] Inigo Quilez. distance to fractals. `https://iquilezles.org/www/articles/distancefractals/distancefractals.htm`.

[9] Inigo Quilez. soft shadows in raymarched sdfs. `https://iquilezles.org/www/articles/rmshadows/rmshadows.htm`.

[10] Aristides AG Requicha and Herbert B Voelcker. Constructive solid geometry. 1977.

[11] Scott D Roth. Ray casting for modeling solids. *Computer graphics and image processing*, 18(2):109–144, 1982.

[12] Adam Runions, Brendan Lane, and Przemyslaw Prusinkiewicz. Modeling trees with a space colonization algorithm. *NPH*, 7:63–70, 2007.

[13] Kees van Kooten, Gino van den Bergen, and Alex Telea. *Point-based visualization of metaballs on a GPU*. Addison-Wesley Longman, 2007.