LMU LUDWIG-MAXIMILIANS-UNIVERSITÄT MÜNCHEN

INSTITUT FÜR INFORMATIK
LEHR- UND FORSCHUNGSEINHEIT FÜR
PROGRAMMIER- UND MODELLIERUNGSSPRACHEN

Institut für Informatik

# Methods for Polygonalization of a Constructive Solid Geometry Description in Web-based Rendering Environments

**Sebastian Steuer**

Diplomarbeit

| | |
|---|---|
| Beginn der Arbeit: | 2. Juli 2012 |
| Abgabe der Arbeit: | 21. Dezember 2012 |
| Aufgabensteller: | Prof. Dr. François Bry |
| Betreuer: | Stephan Drab |

## Erklärung

Hiermit versichere ich, dass ich diese Diplomarbeit selbständig verfasst habe. Ich habe dazu keine anderen als die angegebenen Quellen und Hilfsmittel verwendet.

München, den 21. Dezember 2012 .....

Sebastian Steuer

## Zusammenfassung

Physikalische Körper können auf verschiedene Arten repräsentiert werden. Konstruktive Festkörpergeometrie (CSG) setzt komplexe Körper aus primitiven Körpern mittels Mengenoperation zusammen. Grenzflächenrepräsentationen (BREPs) repräsentieren einen Körper als Zerlegung seiner Oberfläche. Um die jeweiligen Vorteile dieser Ansätze auszunutzen, ist es notwendig zwischen ihnen zu übersetzen.

Die Übersetzung einer CSG-Beschreibung in ein Dreiecksgitter der Oberfläche des beschriebenen Körpers ist eine Polygonalisierung. Primitive mit gekrümmten Oberflächen können früh polygonalisiert werden, indem sie durch Ebenen approximiert werden, deren Schnitte dann ausgewertet werden. Alternativ können sie als Sammlung ihrer definierenden Parameter repräsentiert, ihre Schnitte mittels analytischer Geometrie bestimmt und die Polygonalisierung spät ausgeführt werden.

Diese Arbeit stellt eine Realisierung des zweiten Ansatzes namens *Oberflächengraphrepräsentation* (SGR) vor. Sie ist begrenzt auf kugelförmige Primitive, kann aber hoffentlich auf allgemeinere gekrümmte Oberflächen erweitert werden.

Besondere Aufmerksamkeit gilt der CSG-Modellierung im Web, da die 3D-Grafikfähigkeiten von Browsern derzeit stark zunehmen. Ein Vergleich zwischen SGR und einer CSG-Implementierung in JavaScript sowie einem eigenständigen CSG-Modellierprogramm demonstriert einen Geschwindigkeitsvorteil der späten Polygonalisierung.

## Abstract

Physical objects can be represented in different ways. Constructive Solid Geometry (CSG) is a scheme where a complex object is constructed from primitive objects with operations on sets. Boundary representations (BREPs) represent an object by a decomposition of its surface. Conversion between such different schemes is necessary to utilise their respective advantages.

The conversion of a CSG description to a triangle mesh of the surface of the described solid is a polygonalization. Primitives with curved surfaces can be polygonalized early by approximation with planes whose intersections are then evaluated. Alternatively, they can be represented by a record of their defining parameters, the intersections are evaluated by analytic geometry, and the polygonalization is performed late.

This work presents a realization of the second approach named *Surface Graph Representation* (SGR). It is limited to spherical primitives but hopefully can be extended to more general curved surfaces.

Particular attention is given to CSG modeling on the web, as the capabilities for 3D graphics in browsers are increasing significantly recently. Comparing SGR to a CSG implementation in JavaScript and a stand-alone CSG modeling tool demonstrates superior performance for late polygonalization.

## Acknowledgements

First I would like to thank my advisor Stephan Drab who gave me invaluable help and insight during the whole process of creating this thesis. He and the whole team at Jambit provided a fantastic working environment, including copious amounts of coffee.

Also, I want to particularly thank Prof. Dr. François Bry who took great interest in the proposed topic from the beginning and followed my progress with enthusiasm and lots of helpful advice.

My family accompanied and supported me not only through this thesis, but through the whole course of my studies and indeed my whole life. I am deeply grateful to them.

Many thanks to my wife Pia for everything, she is the best.

Sebastian Steuer

# Contents

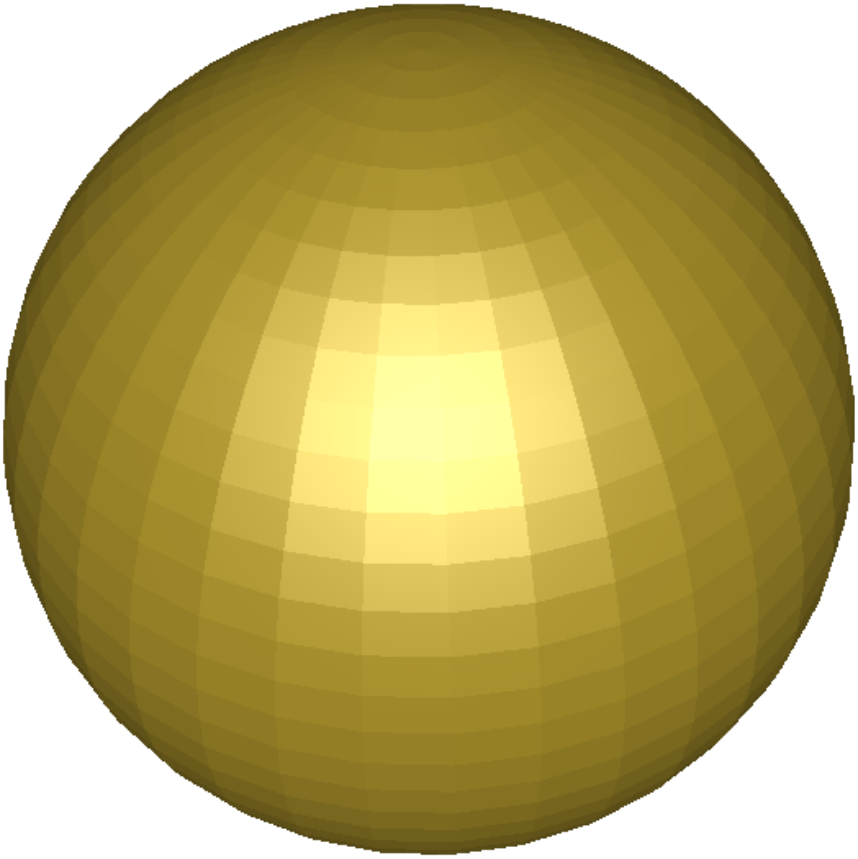


Figure 1: Polygonalization of `Sphere([0,0,0],1)`

# 1 Motivation

Working with representations of physical objects is an essential task in many areas of science and industry. It requires methods to create and modify such representations, to export them for further processing or visualization and to answer geometrical and topological queries. A system of algorithms and datastructures that provides such methods constitutes a geometric modeling system (GMS).

Obviously, the algorithmic and numerical nature of these problems makes it a well suited application for computers. Geometric modeling systens are a foundation for a broad range of fields like Computer Aided Design (CAD) and Computer Aided Manufacturing (CAM) in engineering, Computer Generated Imagery (CGI) in movies and 3D graphics in games.

The need to perform operations on these digital models in consistence with the real world dictates several mathematical properties, the choice of which depends on how strictly consistency with a real-world physical objects is to be conveyed. For example, an object that is bounded by a single plane and thus has infinite volume does not have a physical counter-

part, and could never be manufactured in any way. But its surface may be visualized as a projection on a virtual camera (e.g. the ground in a video game).

Many very different schemes for the modeling of physical objects can be derived from such constraints, each with specific advantages and disadvantages. Depending on the requirements of the intended application, the most suitable representation should be used.

The foremost task of the Graphics Processing Unit (GPU) is visualization; specifically the projection of the three dimensional model into the two dimensional image space of a virtual camera. Therefore, GPUs expect the description of the geometry of the scene to display as a list of triangles on the surfaces of the objects, a triangular mesh. This representation allows fast visibility checks and rendering of the surface of objects, but is rather inept for evaluating the volume of an object or performing complex manipulations while guaranteeing that the result remains a physically possible object. This format is very similar to the structure of the StereoLithography (.stl) file format specified in [21] that is commonly used as exchange format between CAD/CAM systems. Such a representation of an object by a decomposition of its surface is called a boundary representation (BREP).

For high-level construction and modification of objects, Constructive Solid Geometry (CSG) is a more favorable method. Here, complex objects are recursively constructed from transformations and Boolean operations on primitives. So a complex scene consists of unions, intersections and differences of e.g. cubes, spheres and cylinders. This is an intuitively accessible representation and allows elegant parametric construction of complex geometries. It guarantees the validity of objects and allows easy solution to the point problem (to decide whether a given point is inside or outside of the defined object), which is desireable for collision detection. On the other hand, it is not trivial to evaluate the actual surface bounding a solid described this way.

While CSG is a direct representation of a physical object, as the object is described by combinations of volumes composing it, a BREP is an indirect representation, as the volume is only given implicitely by the composition of the surface. While BREP and CSG are arguably the most common, several other schemes for the representation of solid objects are possible. An introduction to and classification of the most common methods is given in chapter 2 after a discussion of the underlying mathematical properties.

The existence of different schemes directly results in the need to perform conversions between them; at least to transform a representation like CSG in a format fit for visualization on the GPU. This work focuses on just this conversion from CSG descriptions to triangular meshes. This task, the polygonalization of a CSG description, is comprised of two distinct subtasks: A boundary evaluation, where the resulting surfaces of the operations on the primitives have to be determined and the generation of a list of triangles that lie on this surface.

The study of the conversion between CSG and BREPS has a long history. It is used in broad range of software, from CAD programs to level editors of video games. Several existing methods for CSG to triangular mesh conversion will be explored in chapter 3. A common feature of these conversions is the use of an intermediate data structure that is less explicit than the final mesh representation, but more so than the original abstract CSG

description, like a binary space partition (BSP).

The recent appearance of techologies like HTML5 and WebGL extend the world wide web as a platform for 3D content by enabling native processing of 3D graphics in browsers. At the same time, miniaturization and price reduction brougt advanced manufacturing tools like CNC mills, 3D-printers and lasercutters that were previously only affordable for specialized industry into the reach of a much broader userbase.[1] This already resulted in the creation of web platforms and applications for exchanging models of physical things.[2] [3] Solid modeling in the scope of a web application is on the rise.[4] CSG modeling on the web is also already approached.[5] [6]

This trend suggests to reevaluete existing methods for CSG polygonalization and to adapt them to this new environment and improving the tools to create and modify these models right inside a web browser.

A method for conversion based on intermediate datastructure denoted *Surface Graph Representation* (SGR) is presented in chapter 4. The implementation is restricted to CSG operations on spherical primitives but proves the feasability of the approach and demonstrates its improved performance on nonplanar primitives.

A comparison of the performance of the new conversion method and existing CSG renderers and evaluation of their algorithmic complexity is provided in chapter 5.

Chapter 6 outlines open questions and possible improvements on the proposed CSG to mesh conversion method, especially extending the approach from spherical objects to more general shapes, namely quadric surfaces.

The potential gain is further convergence between CAD/CAM and web technologies, so that models for physical objects can be displayed, edited and exchanged in a web browser, like it is already possible for text, images, audio and video, with the same benefits by collaboration and decentralization.

[1] `http://www.reprap.org/`
[2] `http://www.thingiverse.com/`
[3] `http://mediagoblin.org/news/3d-support.html`
[4] `https://tinkercad.com/`
[5] `http://evanw.github.com/csg.js/`
[6] `http://joostn.github.com/OpenJsCad/`

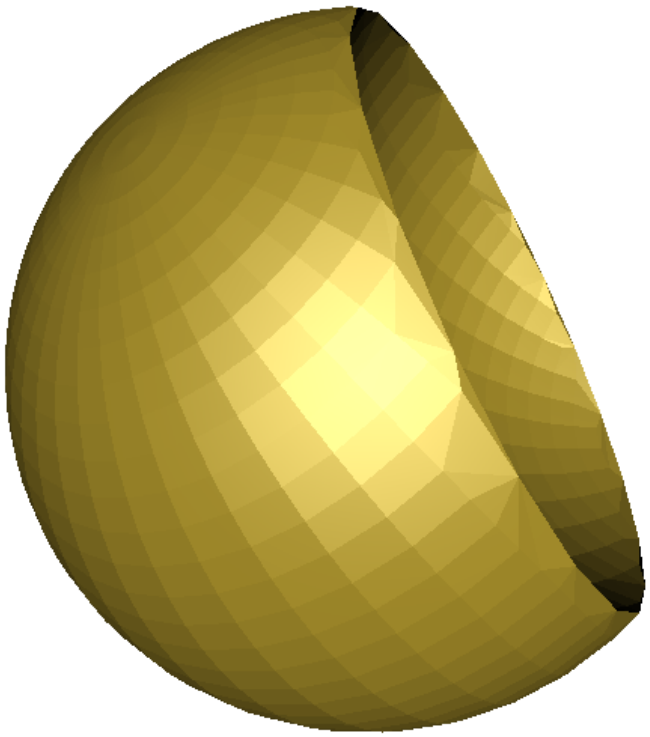


Figure 2: Polygonalization of `difference(Sphere([0,0,0],1), Sphere([1,0,0],1))`

## 2 Fundamental Concepts

In order to create a representation of a physical object and to do any meaningful operations on it, we need a framework that defines concepts like space, points, surfaces, volumes and their relations in consistence with the reality the objects originated in.

Requicha [17] gives a list of useful properties for an "abstract solid":

1. Rigidity: An abstract solid must have an invariant configuration or shape which is independent of the solid's location and orientation.
2. Homogeneous three dimensionality: A solid must have an interior and a solid's boundary cannot have isolated or "dangling" portions.
3. Finiteness: A solid must occupy a finite portion of space.
4. Closure under rigid motions and certain Boolean operations: Rigid motions (translations and/or rotations) or operations that add or remove material (welding, machining) must, when applied to solids, produce other solids.
5. Finite describability: There must be some finite aspect of the models of solids (e.g., a finite number of "faces") to ensure that they are representable in computers.

6. Boundary determinism: The boundary of a solid must determine unambiguously what is "inside" and hence comprises the solid.

A thorough introduction to a class of sets and operations that fulfill these properties is given in [19]. This chapter begins with a review of the most important principles.

## 2.1 Sets

If $A$ and $B$ are subsets of a reference set $U$ $(A \subseteq U, B \subseteq U))$, the following operations are defined. The empty set is denoted by $\varnothing$.

**Definition 2.1.0.1.**

$$\text{union} : A \cup B = \{p \mid p \in A \vee p \in B\}$$

$$\text{intersection} : A \cap B = \{p \mid p \in A \wedge p \in B\}$$

$$\text{difference} : A \setminus B = \{p \mid p \in A \wedge p \notin B\}$$

$$\text{complement} : \overline{A} = \{p \mid p \in U \wedge p \notin A\}$$

These operations have several properties:

**Property 2.1.0.1.** Union and intersection are commutative:

$$A \cup B = B \cup A$$

$$A \cap B = B \cap A$$

**Property 2.1.0.2.** Union and intersection are distributive over each other:

$$A \cup (B \cap C) = (A \cup B) \cap (A \cup C)$$

$$A \cap (B \cup C) = (A \cap B) \cup (A \cap C)$$

**Property 2.1.0.3.** The empty set $\varnothing$ and the reference set $U$ are identity elements for union and intersection:

$$A \cup \varnothing = A$$

$$A \cap U = A$$

**Property 2.1.0.4.** The complement satisfies:

$$A \cup \overline{A} = U$$

$$A \cap \overline{A} = \varnothing$$

These properties are characteristic for an important class of algebraic systems:

**Definition 2.1.0.2.** A set of elements $A, B, ...$ and the operations $A \cup B$, $A \cap B$ and $\overline{A}$, which satisfy properties 2.1.0.1 to 2.1.0.4 is called a *Boolean algebra*.

Among other properties for sets and Boolean algebras, one is of special interest as it shows how to substitute operations by each other:

**Property 2.1.0.5.** De Morgan's laws

$$\overline{A \cup B} = \overline{A} \cap \overline{B}$$

$$\overline{A \cap B} = \overline{A} \cup \overline{B}$$

$$\overline{A \setminus B} = \overline{A} \cup B$$

Using De Morgan's laws, any two of the operations $\cup, \cap, \setminus$ can be reduced to the remaining one and the complement.

**Property 2.1.0.6.** Any class of subsets of $U$ which is closed under $\cap$, $\cup$ and $\overline{X}$ is a boolean algebra.

With the Boolean algebra of sets we can operate on objects as subsets of a reference set. But not all subsets are a representation of possible physical object by the properties given at the beginning of this chapter. In order to satisfy the condition of homogeneous three dimensionality we have to introduce the concept of regular sets, based on topology.

### 2.1.1 Point-Set Topology

**Definition 2.1.1.1.** A solid $S$ partitions the reference set $U$ in three components: interior $int(S)$, exterior $ext(S)$ and boundary $bd(S)$.

$$int(S) \cup ext(S) \cup bd(S) = U$$

$$int(S) \cap bd(S) = bd(S) \cap ext(S) = int(s) \cap ext(S) = \varnothing$$

**Definition 2.1.1.2.** A continuous function $f(p)$ for every $p \in U$ is called the *defining function* for $S$ if

$$p \in ext(S) : f(p) < 0$$
$$p \in bd(S) : f(p) = 0$$
$$p \in int(S) : f(p) > 0$$

If such a function exists, $S$ is called a *halfspace*.

[20] [16]

### 2.1.2 Regular Sets

**Definition 2.1.2.1.** The closure of a set $S$ is the set with its boundary.

$$cl(S) = S \cup bd(S)$$

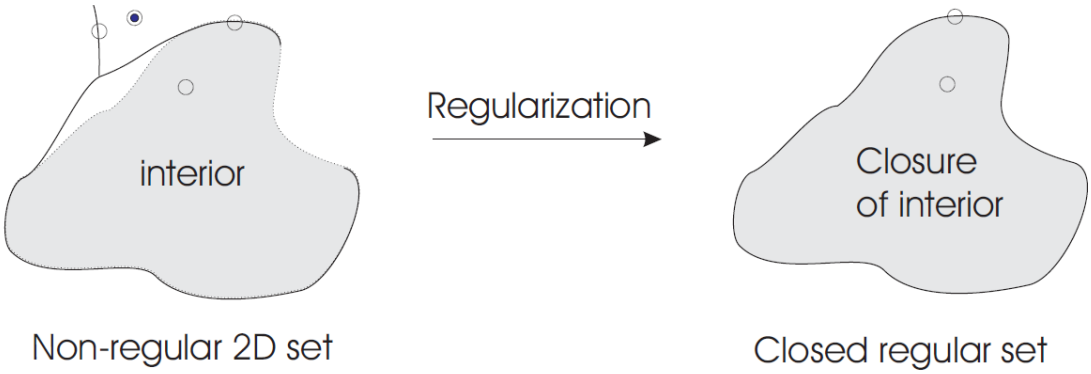



Figure 3: Regularization of sets. Illustration from [5].

**Definition 2.1.2.2.** A set $S$ is regular if it is the closure of its interior. The closure of the interior of a set is called the regularization of the set.

$$S = cl(int(S)) \iff S \text{ is regular}$$

As can be seen from the example in Figure 3 regularization removes "dangling" points and edges.

Sets resulting from Boolean operations -even on regular sets- are not necessarily regular, see figure 4 for an example. We define regularized variants of the Boolean operations:

**Definition 2.1.2.3.** Regularized Boolean set operations are

$$A \cup^* B = cl(int(A \cup B))$$

$$A \cap^* B = cl(int(A \cap B))$$

$$A \setminus^* B = cl(int(A \setminus B))$$

$$\overline{A}^* = cl(int(\overline{A}))$$

[19]

Usually, the interior of an object and its closure can not be directly evaluated in a given representation so other methods have to be used to preserve regularity.

Now we can operate on homogenous solids and can evaluate whether points are inside, outside or on the border of a solid. However, we can not yet evaluate any other relations, not even the distance between two points. For that, we have to define a *metric space*.

## 2.2 Metric Spaces

**Definition 2.2.0.4.** A *metric space* is a pair $(M, d)$ where $M$ is a vector space and $d$ is a function (called a metric) $d : M \times M \to \mathbb{R}$, so that for any $x, y, z \in M$

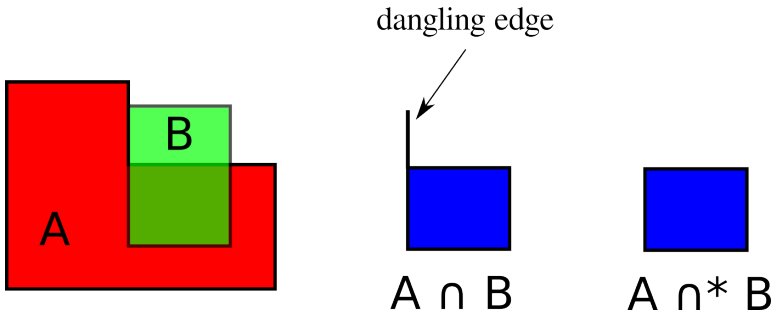



Figure 4: The result of Boolean operations on regular sets is not always regular. The intersection $A \cap B$ of the 2-dimensional regular shapes $A$ and $B$ leaves a dangling edge where $A$ and $B$ coincide in a line, making it not regular. Taking the closure of the interior of the intersection removes the dangling edge, yielding a regular result.

1. $d(x, y) \geq 0$
2. $d(x, y) = 0 \iff x = y$
3. $d(x, y) = d(y, x)$
4. $d(x, z) \leq d(x, y) + d(x, z)$

**Definition 2.2.0.5.** The Euclidian distance $d(p, q)$ for $p, q \in \mathbb{R}^n$ is defined as

$$d(p, q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2}$$

Our physical universe has three spatial dimensions and every point can be expressed as a vector of real-valued Cartesian coordinates. The Euclidean distance is the demonstrative distance measure of the physical world where the Pythagorean theorem holds true. Therefore to model representations of physical objects it is obvious to operate on $\mathbb{R}^3$ with the Euclidean distance as metric.

While other distances and spaces are regularely used for other applications, Euclidean space reflects our intuition about "space" and any spatial universe with a different metric would be very confusing. See [14].

Arbitrary elements of $\mathbb{R}$ are not representable in a computer, only a finite subset of elements that can be expressed as floating-point numbers. Hence floating point operations can not precisely represent true arithmetic operations, see listing 1 for examples. This trait has serious implications. For a further discussion see [11]. Some problems can be mitigated by appropriate rounding and relaxing the equality by comparison with an $\epsilon$-environment. However rounding errors may accumulate and we need to be aware that we are working in an environment with finite precision.

```
>>> 0.1 + 0.2
0.30000000000000004
>>> 0.1 + 0.2 == 3.0/10
False
>>> round(0.1 + 0.2) == 3.0/10
True

>>> sin(0) == 0
True
>>> sin(pi) == 0
False
>>> round(sin(pi)) == 0
True
```

Listing 1: Demonstration of arithmetic properties of floating-point values in python

### 2.2.1 Binary Space Partitions

A tree datastructure with defining functions as interior nodes (the *separators*) with 2 child nodes for the positive and negative sides and leaf nodes that are either "IN" or "OUT" is called a binary space partition. It recursively partitions a space with hypersurfaces given by the locus of zeros of the defining functions, with the regions at the leaves of the tree identified as inside or outside. [24]

Figure 5 shows a shape in $\mathbb{R}^2$ and its BSP tree. The separating hypersurfaces are lines.

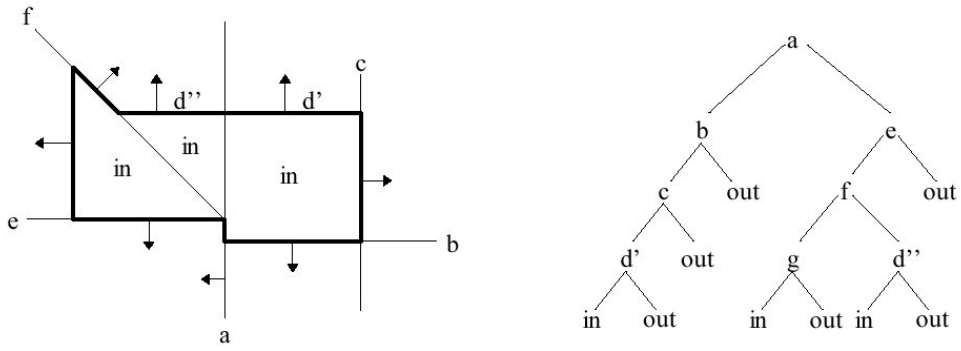



Figure 5: BSP representation (right) of two-dimensional shape (left). Illustration from [1].

## 2.3 Geometry

With a robust notation of space, points and their distances as well as solids and operations on them, we can start to explore geometry to construct objects.

### 2.3.1 Point

A point or vertex is a location vector of Cartesian coordinates in space.

**Definition 2.3.1.1.**

$$\vec{p} \in \mathbb{R}^3 : \vec{p} = \begin{pmatrix} p_x \\ p_y \\ p_z \end{pmatrix}$$

with $p_x, p_y, p_z \in \mathbb{R}$

**Definition 2.3.1.2.** For $\vec{p}$, $\vec{q} \in \mathbb{R}^3$ and a scalar $s \in \mathbb{R}$, the following operations are defined:

$$\vec{p} + \vec{q} = \begin{pmatrix} p_x + q_x \\ p_y + q_y \\ p_z + q_z \end{pmatrix}$$

$$\vec{p} - \vec{q} = \begin{pmatrix} p_x - q_x \\ p_y - q_y \\ p_z - q_z \end{pmatrix}$$

$$s\vec{p} = \begin{pmatrix} sp_x \\ sp_y \\ sp_z \end{pmatrix}$$

$$\vec{p}/s = \vec{p}(1/s) \text{ for } s \neq 0$$

$$\vec{p} \cdot \vec{q} = p_x q_x + p_y q_y + p_z q_z$$

$$\vec{p} \times \vec{q} = \begin{pmatrix} p_y q_z - p_z q_y \\ p_z q_x - p_x q_z \\ p_x q_y - p_y q_x \end{pmatrix}$$

$$|\vec{p}| = \sqrt{\vec{p} \cdot \vec{p}}$$

#### Transformations

**Definition 2.3.1.3.** If $T$ is a linear transformation mapping $\mathbb{R}^n$ to $\mathbb{R}^m$ and $\vec{x}$ is a column vector with $n$ entries, then

$$T(\vec{x}) = A\vec{x}$$

for a $n \times m$ matrix $A$.

In a geometric context, this reduces all linear transformations to matrix multiplication of coordinate vectors. Some transformations that are not linear in $\mathbb{R}^n$ (like translation and projection) can be represented as linear transformations in $\mathbb{R}^{n+1}$.

### 2.3.2 Plane

**Definition 2.3.2.1.** A plane $P$ is defined by any point on the plane $\vec{p}$ and a normal $\vec{n}$ orthogonal to the plane (with $|\vec{n}| = 1$)

$$P = (\vec{p}, \vec{n})$$

It is an halfspace with the defining function

$$f_P(\vec{x}) = \vec{n} \cdot (\vec{x} - \vec{p})$$

A plane has an infinite surface and bounds an infinite interior if understood as a halfspace. Thus, it is not an abstract solid by the properties at the beginning of the chapter, yet valid abstract solids can be constructed by boolean operations on multiple planar halfspaces. For example a cube from the intersection of six planes.

This principle can also be used to approximate objects with curved surfaces.

### 2.3.3 Circle

**Definition 2.3.3.1.** A circle $C$ is given by a midpoint $\vec{m}$, a normal $\vec{n}$ (with $|\vec{n}| = 1$) and a radius $r$.

$$C = (\vec{m}, \vec{n}, r)$$

**Property 2.3.3.1.** All points $\vec{x}$ in the plane of the circle satisfy the condition

$$(\vec{m} - \vec{x}) \cdot \vec{n} = 0$$

Those points are either

- inside the circle if $|\vec{m} - \vec{x}| - r < 0$
- on the circle if $|\vec{m} - \vec{x}| - r = 0$
- outside the circle if $|\vec{m} - \vec{x}| - r > 0$

A circle is no abstract solid, as it bounds no volume.

### 2.3.4 Sphere

**Definition 2.3.4.1.** A sphere $S$ is defined by a midpoint $\vec{m}$, a radius $r$ and an inversion flag $i \in \{-1, 1\}$

$$S = (\vec{m}, r, i)$$

With the defining function

$$f_S(\vec{x}) = (|\vec{p} - \vec{x}| - r)i$$

A non-inverted sphere (i = 1) where everything closer than $r$ to $\vec{m}$ is considered inside is an abstract solid, while an inverted one where everything farther away than $r$ from $\vec{m}$ is considered inside is not, as its volume is infinite. Nevertheless, valid abstract solids can be constructed from inverted spheres. For example a hollow sphere from an intersection of two concentric spheres with different radii, the one with the smaller radius inverted and the bigger one not.

**Property 2.3.4.1.** All points $\vec{x}$ on the surface of the sphere satisfy the condition

$$|\vec{x} - \vec{m}| = r$$

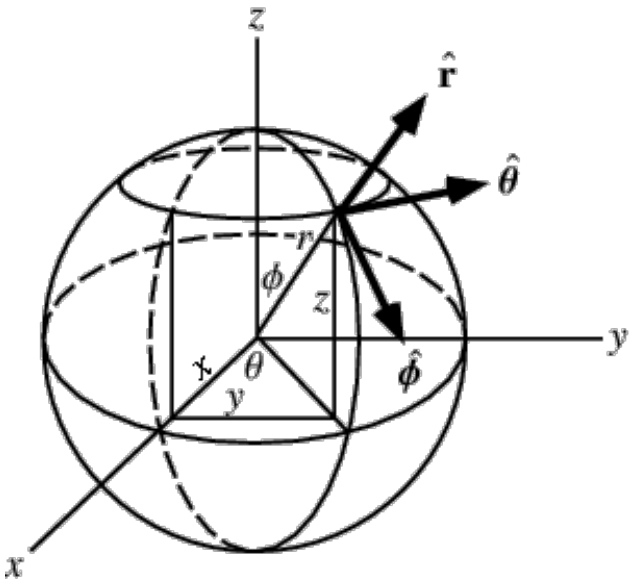



Figure 6: Spherical coordinates. Illustration from [26].

Positions on the surface of a sphere can be expressed as spherical coordinates, see figure 6.

**Property 2.3.4.2.** For an azimuthal angle $\theta$ in the x-y-plane from the x-axis with $0 \leq \theta \leq 2\pi$ and a polar angle $\phi$ from the positive z-axis with $0 \leq \phi \leq \pi$ and radius $r$ of the sphere, a point $\vec{p}$ on the surface of the sphere in Cartesian coordinates is:

$$\vec{p} = \begin{pmatrix} r\cos(\theta)\sin(\phi) \\ r\sin(\theta)\sin(\phi) \\ r\cos(\theta) \end{pmatrix}$$

[26]

**Property 2.3.4.3.** The relation between two arbitrary spheres $S_1$ and $S_2$ with distance $d_S = |\vec{m}_{S_1} - \vec{m}_{S_2}| \neq 0$ is:

- $d_S > r_{S_1} + r_{S_2}$ : No intersection, no containment
- $d_S + r_{S_1} > r_{S_2}$ : No intersection, $S_1$ is inside $S_2$

- $d_S + r_{S_2} > r_{S_1}$ : No intersection, $S_2$ is inside $S_1$
- $d_S < r_{S_2} + r_{S_1}$ : Intersection in a circle $C$

In the case of intersection, the distance $d_C$ of the center $m_C$ of the circle $C$ to $m_{S_1}$ is $d_C = \frac{r_{S_1}^2 + d_S^2 - r_{S_2}^2}{d_S}$. Using this distance we can describe $C$:

- $\vec{m}_C = \vec{m}_{S_1} + (\vec{m}_{S_2} - \vec{m}_{S_1})\frac{d_C}{d_S}$
- $r_C = \sqrt{r_{S_1}^2 - d_C^2}$
- $\vec{n}_C = \frac{(\vec{m}_{S_2} - \vec{m}_{S_1})}{|\vec{m}_{S_2} - \vec{m}_{S_1}|}$

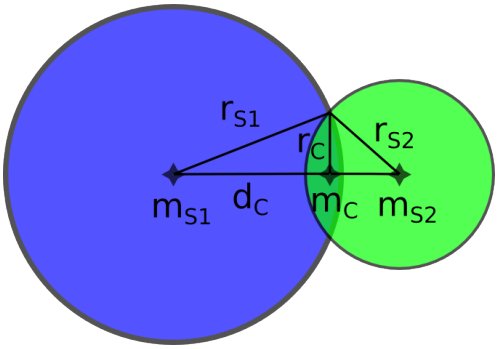



Figure 7: Circle of intersection of two spheres.

### 2.3.5 Quadric Surface

**Definition 2.3.5.1.** Quadric surfaces in Euclidean space are surfaces given by the locus of zeros of a quadratic polynomial.

$$\begin{pmatrix} x & y & z & 1 \end{pmatrix} Q \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = 0$$

with $x, y, z \in \mathbb{R}$, $Q \in \mathbb{R}^{4x4}$

**Property 2.3.5.1.** Any quadric can be normalized by chosing the coordinate directions as principal axes of the quadric. Among those normalized forms are

- Ellipsoid (generalization of sphere) with $\frac{x^2}{a^2} + \frac{y^2}{b^2} + \frac{z^2}{c^2} = 1$
- Elliptic cylinder (generalization of cylinder) with $\frac{x^2}{a^2} + \frac{y^2}{b^2} = 1$
- Planes with $x^2 = 0$

Quadrics allow a generalized handling of various surfaces. Ray casting can be performed analytically and transformations can be directly applied as matrix multiplications.

Efficient, general and robust algorithmic evaluation of intersections of quadrics is an active field of research.[8]

## 2.4 Solid Modeling

After reviewing the neccessary principles of set theory, topology and analytical geometry, we can now examine several common methods for modeling solids starting with Constructive Solid geometry.

### 2.4.1 Constructive Solid Geometry

Constructive Solid Geometry defines a solid by a sequence of operations on primitives or previous constructions. Primitives are parametrized solids, e.g. cubes, spheres and cylinders. Primitives and their combinations can be moved by rigid translations or scaled.

```
a = Cube(center = [0,0,0], r = [1,1,1]);
b = Sphere(center = [0,0,0], r = 1.35);
c = Cylinder(r = 0.7, start = [-1,0,0], end = [1,0,0]);
d = Cylinder(r = 0.7, start = [0,-1,0], end = [0,1,0]);
e = Cylinder(r = 0.7, start = [0,0,-1], end = [0,0,1]);
a.intersect(b).subtract(c.union(d).union(e))
```

Listing 2: CSG description

Listing 2 demonstretes a syntax describing a solid in CSG. Parsing such a synax yields a tree datastructure with primitives as leaves and Boolean operations and transformations as inner nodes. Figure 8 shows the CSG tree evaluated from listing 2 and figure 9 the polygonalization of the described object.

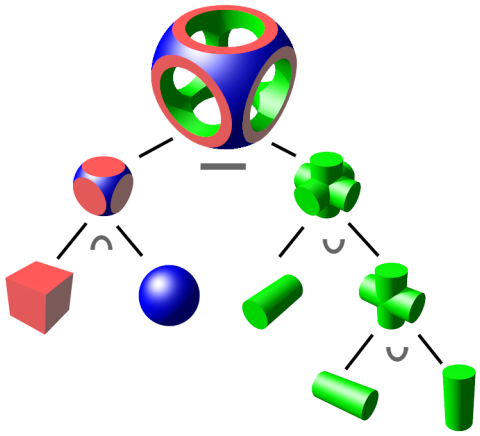


Figure 8: CSG tree. Illustration from [3].

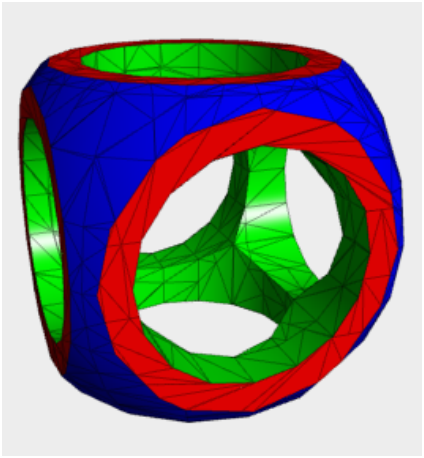


Figure 9: Polygonalization of the CSG description from listing 2. Illustration from [2].

A CSG description is concise and easily parametrized and edited, which makes it suitable format for procedural and high level modeling. But it does not carry any explicit information on the connectivity or even existance of a corresponding solid. A non-empty CSG description can represent empty space, for example the (Boolean) intersection of two disjunct primitives. To address these questions the boundary is evaluated and a complete or partial BREP is derived from the CSG description.

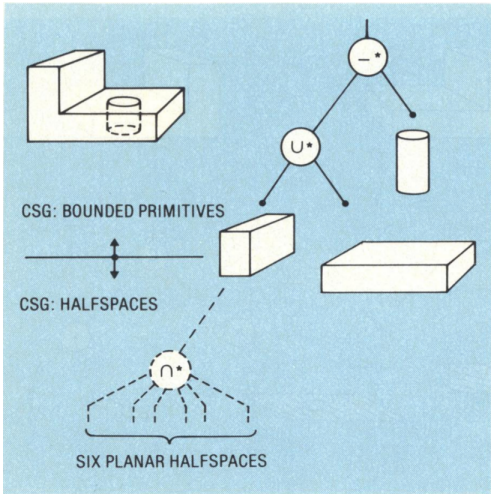



Figure 10: Construction of primitives from halfspaces in CSG. Illustration from [18].

**Primitives and halfspaces** If the combination operations are the regular Boolean operations from 2.1.2.3 and the primitives are valid abstract solids, then any resulting CSG representation is also valid. [22]

Commonly used primitives are cubes, spheres and cylinders and their generalizations but generally any bounded solid can be used.

It is possible to use halfspaces as simplest primitive. While a planar halfspace is not a bounded volume, a cube constructed from the intersection of 6 planes is, and a CSG implementation that exposes these cubes or other bounded volumes constructed from halfspaces as primitives also guarantees valid solids as output.[18] Figure 10 illustrates this relation of halfspaces to bounded primitives.

Arbitrary objects can be approximated with this method by using just planar halfspaces, especially commonly used non-planar primitives like spheres and cylinders. The cost for the ease of implementation of this approach is the large number of planes appearing, depending on the resolution of the approximation. The pairwise intersections between these planes has to be evaluated, making it computationally expensive.

**Operations** De Morgan's laws show that any two of the operations $\cup, \cap, \setminus$ can be reduced to the third one and the complement. So any CSG tree can be normalized in a way that only one binary operartion and the complement appear. This enables to write algorithms evaluating general CSG descriptions by just handling one selected operation and the complement.

Alternatively, the regular Boolean operations can be reduced to other operations, which will be shown in chapter 3.

### 2.4.2 Boundary Representations

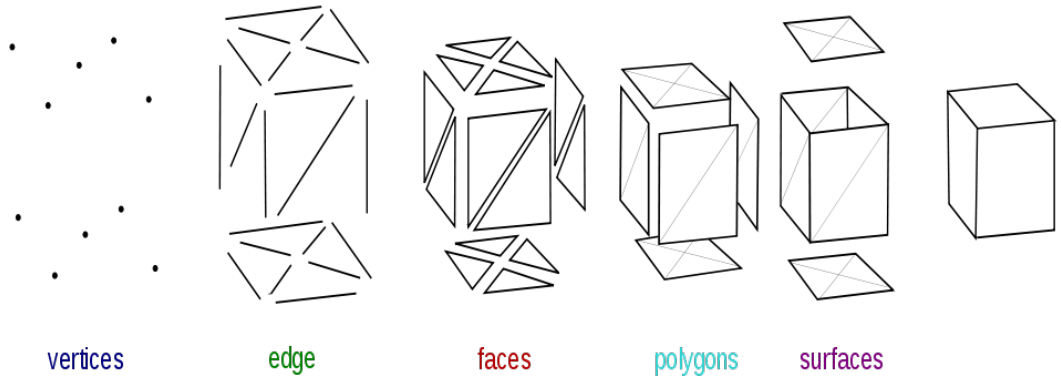



Figure 11: Composition of solids in BREPs. Illustration from [4].

Boundary representations or BREPs are schemes that represent a solid in term of its boundary, usually as hierarchic composition of parts with decreasing dimensionality (two-dimensional faces described by one-dimensional edges described by dimensionless points). Figure 11 shows the assembly of a solid from a BREP persepective.

**Constraints** In order to be a valid abstract solid, a BREP has to be a *closed, oriented manifold surface embedded in* $\mathbb{R}^3$. [13]

- On a *manifold surface* each point is homeomorphic to a disc. Self-intersecting surfaces are not manifolds.
- A manifold surface is *oriented* if any path on the manifold maintains the orientation of the normal. A Moebius strip is an example of a not oriented surface.
- An orieneted manifold surface is *closed* if it partitions $\mathbb{R}^3$ into points that are inside of, outside of and on the surface (definition 2.1.1.1). Spheres and (infinite) planes are closed, while a bounded plane, e.g. a single triangle is not closed.
- A closed, oriented manifold surface is *embedded in* $\mathbb{R}^3$ if geometric and not just topological information is known. "A sphere" is not embedded in $\mathbb{R}^3$, "the sphere at the origin with radius 1" is.

**Triangle mesh** A BREP with linear edges and planar faces is called a *polygon mesh*. The edges defining a planar face have to be coplanar. A triangle is the simplest polygon and always coplanar. Every other polygon can be expressed as a set of triangles. Therefore, it is an obvious candidate as fundamental component in BREPs. BREPs that are composed of triangular faces are called *triangle meshes* and are broadly used, foremost in GPUs, which are highly optimized for fast operations on those datastructures.

A naive implementation of a mesh might store the coordinates of a vertex for every face it appears in. As vertices usually appear as a connection between several faces in a solid, this is redundant and manipulations of vertices have to change data in several places.

Consequently, datastracures are used which reduce this redundance, most commonly index arrays, where all vertices are stored in an array and referred to by their indices from a second array of faces.

The manifold condition implies several conditions on the mesh:

- All pairs of vertices are disjoint.
- All pairs of edges are disjoint or intersect at one vertex.
- All pairs of faces are disjoint or intersect at a common edge.

**Right-hand rule** If a triangle (and with it a plane) is given by three points $\vec{p}_1, \vec{p}_2, \vec{p}_3$, the normal $\vec{n}$ of the plane can be evaluated as $\vec{n} = \vec{u} \times \vec{v}$ with $\vec{u} = \vec{p}_1 - \vec{p}_2, \vec{v} = \vec{p}_1 - \vec{p}_3$. The usual consensus in 3D modeling is that in a right-handed coordinate system, the direction of the normal is considered the outside of the face. In consequence, the orientation of a triangle can be inverted by swaping the order of any two defining points.

**STL** (STereoLithography) is a file format originally developed and specified ([21])by the company 3D systems for their CAD software It is today supported by many other programs and widely used for rapid prototyping and CAM.

STL only describes the surface geometry of an object without representation of color, texture or other properties. Both a binaray and ASCII representation are specified.

The surface is described by vertices and normals of faces, although some implementation ignore the normal information and derive the face normal by the right-hand rule.

```
solid name
  facet normal n_i n_j n_k
    outer loop
      vertex v1x v1y v1z
      vertex v2x v2y v2z
      vertex v3x v3y v3z
    endloop
  endfacet
endsolid name
```

Listing 3: STL ASCII file format

Listing 3 shows the structure of an ASCII STL file. The 'facet' block is repeated for every triangle. While the structure suggests that arbitrary polygons can be used or that a facet can consist of several polygons, virtually all applications expect just one triangle per facet.

**WebGL** WebGL[6] is a JavaScript API for rendering 3D (and 2D) graphics in a compatible web browser without plugins. It allows direct access to the GPU from the browser and so enables fast, hardware-accelerated 3D graphics embedded in web pages. While it is not yet implemented in all modern browsers, support is growing.

Besides the geometry of objects, which was already extensively covered in this chapter, rendering of a 3D scene takes a few more components:

- A (virtual) camera on which the 2D projection to be shown is rendered.
- Materials that define the visual properties like reflection, transparency and color of the objects. Materials are realized as *shaders*, small programs that are executed by the GPU during the rendering.

Libraries like three.js[7] or lightgl[8] allow the definition of a scene on a high level of abstraction. See listinge 4 for an example how a minimal scene can be set up in a browser in JavaScript using three.js.

[7] `http://threejs.org/`
[8] `https://github.com/evanw/lightgl.js/`

```
var camera, scene, renderer;
var geometry, material, mesh;
var ratio = window.innerWidth / window.innerHeight;

init();
animate();

function init() {
  camera = new THREE.PerspectiveCamera(75, ratio, 1, 10000);
  camera.position.z = 1000;
  scene = new THREE.Scene();
  geometry = new THREE.CubeGeometry(200, 200, 200);
  material = new THREE.MeshBasicMaterial({color: 0xff0000});
  mesh = new THREE.Mesh(geometry, material);
  scene.add(mesh);
  renderer = new THREE.WebGLRenderer();
  renderer.setSize(window.innerWidth, window.innerHeight);
  document.body.appendChild(renderer.domElement);
}

function animate() {
  requestAnimationFrame(animate);
  mesh.rotation.x += 0.01;
  mesh.rotation.y += 0.02;
  renderer.render(scene, camera);
}
```

Listing 4: setting up a scene showing a rotating cube in a browser environment using three.js

### 2.4.3 Others

While BREPs and CSG are the most commonly used schemes for representing solids and the key aspects of this work, several other schemes exist and are briefly introduced in this section for an overview based on [17] and [18].

**Parametrized primitive instancing** This scheme is based on the notion of families of objects where each member of a family is distinguishable from the others by one or more parameters. Each family is called a *generic primitive* and individual objects are called *primitive instances*. For example, "bolts" are a generic primitive and a single bolt with a certain length and diameter is a primitve instance. Pure parametrized primitive instancing schemes provide no operations for combining instances to create more complex objects. Also, no general algorithms can be implemented to compute properties of represented solids. Family-specific properties make each generic primitive a special case.

**Spatial occupancy enumeration** This scheme is a list of spatial *cells* occupied by the solid. Space is partitioned into regions of fixed size, arranged in fixed grid, most commonly

cubes. Each cell or *voxel* may be represented by a single point such as its centroid. The representation of the solid is given by a list of cells with their representant in the interior of the solid.

**Cell decomposition** is a generalization of spatial occupancy enumeration. In cell decomposition, solids are represented by a decomposition in spatial cells which don't have to be uniform and arranged in a fixed grid. Cell decomposition can be seen as a special case of CSG with the cells as primitives and only a union operation on disjunct objects.

**Sweeping** A set moving through space can represent a volume given by the set and it's trajectory. Such a representation is especially useful for motion planing, e.g. of the tool head in a CNC machine or navigation of a robot.

Figure 12: Polygonalization of the CSG description

```
difference(
   Sphere([0,0,0],1),
   union(Sphere([1,0,0],1.3), Sphere([-1,0,0],1.3)))
```

# 3 Related Work

## 3.1 Brief History of CSG

The work on ray-tracing of objects constructed from Boolean operations by Goldstein and Nagel [12] in 1971 can be considered as the origin of CSG. CSG as a solid mdeling scheme was formalized by Ricci [20] two years later.

During the 1970s and 80s, CSG and other schemes for solid modeling were improved, formalized and extensively studied, and a broad range of 3D modeling applications were implemented. [17] [18] [19].

Thibault and Naylor [24] described how polyhedra can be represented using a BSP tree and how to perform Boolean operations on them, in essence a CSG to BSP conversion. Buchele and Roles [7] extended this approach from polyhedra to non-planar separators and also gave an opposite conversion algorithm from BSP to CSG, showing that the domains of CSG and BSP representations are equal.

Wyvill and van Overveld [27] presented a method for CSG to BREP conversion for implicit

surfaces. It uses an intermediate datastructure similar to the one presented in the next chapter which was partially inspired by it.

Since the turn of the millenium, techniques like image-space CSG rendering [23] [15] and new conversion approaches [25] were developed.

## 3.2 OpenSCAD

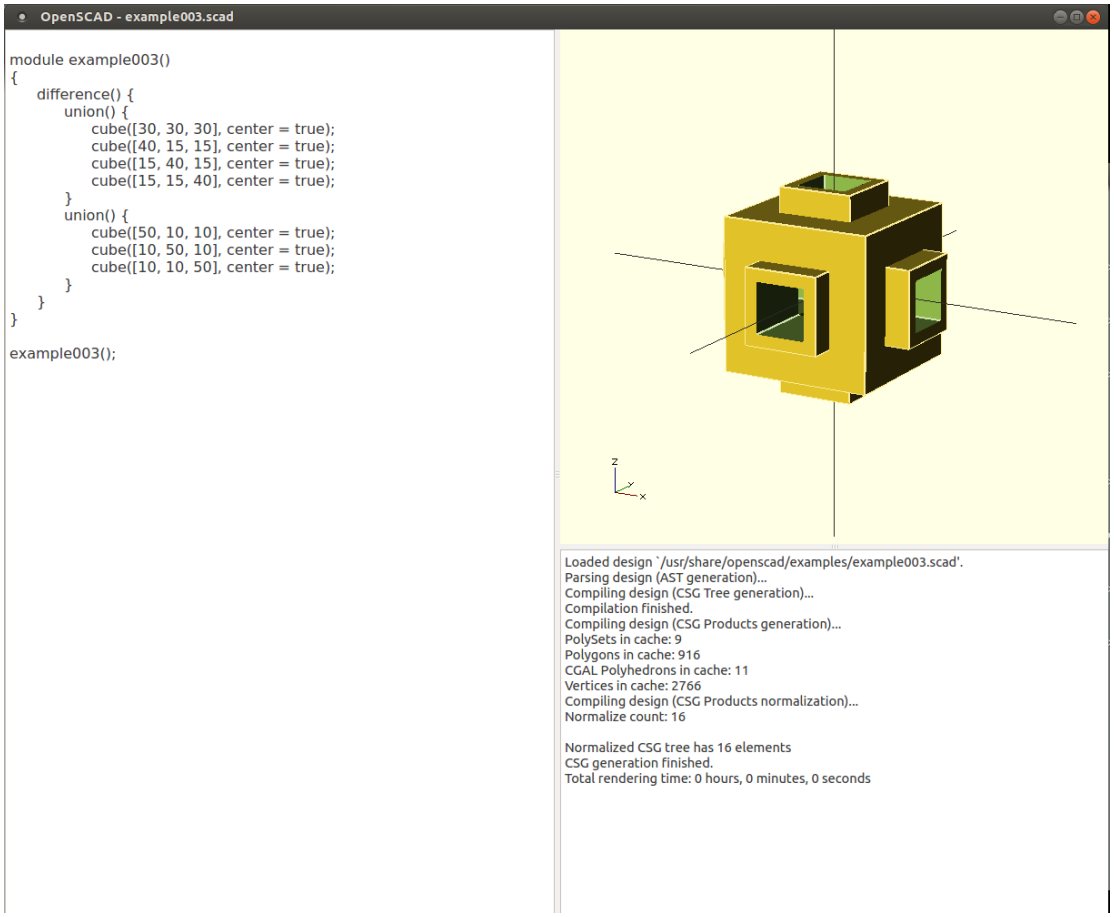



Figure 13: Screenshot of OpenSCAD

OpenSCAD [9] is a parametric CSG-based solid modeling software. In contrast to other tools, objects are not manipulated graphically (e.g. by dragging points with the mouse) but by editing the textual CSG description from which the object is rendered.

It employs two different mechanisms for processing the CSG description. For visualization of the object during editing it uses the OpenCSG library, which performs a rendering to image space. For export, the CGAL library is used to create a meshed BREP output in STL format.

[9] `http://www.openscad.org/`

Figure 13 shows a screenshot of the software, with the CSG description on the left and the visualization on the right. The grammar for the CSG description includes loops, conditions, variables, modules and other concepts that enable concise procedural parametric modeling.

### 3.2.1 openCSG

OpenCSG is an image-space CSG rendering library. Instead of evaluating the boundaries of an object in $\mathbb{R}^3$, the two-dimensional image of an CSG object from a given perspective is directly evaluated.

This is achieved by buffer structures in the GPU, which are intended to evaluate the visibility of surfaces. The Goldfeather algorithm and the SCS algorithm use these buffers to evaluate the composition of the CSG shapes.

The key benefit of this approach is that it is very fast, even complex shapes can be rendered in short time. This makes it attractive for preview rendering in interactive modeling. On the other hand, no evaluation of the boundary in $\mathbb{R}^3$ is performed, so it has to be accompanied with a different scheme to export the modeled object as a BREP.

A comprehensive presentation of the openCSG library and image-space CSG rendering is given by [15].

### 3.2.2 CGAL

To create a BREP from the CSG description, openSCAD utilizes the Computational Geometry Algorithms Library (CGAL). This library covers a very broad range of geometry applications, where CSG is one among many others.

To evaluate a CSG tree, CGAL uses two datastructures. One that stores the local neighborhood of each vertex, and one that connects these vertices in a hierarchy to edges, facets and volumes.

Refer to [9] for an introduction to the featureset of CGAL.

## 3.3 Web-based CSG

While CSG modeling in general has a long history and a broad range of implementations with different underlying schemes were realized over time, the options for CSG modeling on the web is more limited. To the authors knowledge, there is currently just one library that provides this functionality client side, csg.js. Based upon that is openJsCad, which aims to realize the functionality of OpenSCAD as a procedural, parametric CSG modeler as a web application.

### 3.3.1 csg.js

csg.js[10] is an implementation of a CSG to triangular mesh conversion based on a intermediate BSP-tree datasstructure. The separtors of the BSP tree are planes. For every separator, the polygons on that plane (its intersections with other planes) are also stored in the node. They are evaluated at the construction of the BSP tree from the CSG description and after that retrieved for rendering.

The polygons already carry information about their orientation, which leads to a "leaveless" BSP tree as it does not need explicit "IN" and "OUT" nodes.

The regularized Boolean operations `union`, `intersection` and `difference` are implemented as applications of the operations `clip` and `complement` and `merge`. See algorithms 1-3 on how they are realized.

**Algorithm 1** `union`

Input: Regular solids $A, B$
Output: Regular solid

$A', B' :=$ `clip`$(A, B)$, `clip`$(B, A)$
$B'' :=$ `complement`(`clip`(`complement`$(B')$, $A'$)
**return** `merge`$(A', B'')$

**Algorithm 2** `intersection`

Input: Regular solids $A, B$
Output: Regular solid

$A' :=$ `complement`$(A)$
$B' :=$ `complement`(`clip`$(B, A')$)
$A'', B'' :=$ `clip`$(A', B')$, `clip`$(B', A')$
**return** `complement`(`merge`$(A'', B'')$)

**Algorithm 3** `difference`

Input: Regular solids $A, B$
Output: Regular solid

$A' :=$ `complement`$(A)$
$A'', B' :=$ `clip`$(A', B)$, `clip`$(B, A')$
$B'' :=$ `complement`(`clip`(`complement`$(B')$, $A''$))
**return** `complement`(`merge`$(A'', B'')$)

`clip(`$A$`,` $B$`)` removes everything in $A$ that is in $B$. `complement(`$A$`)` inverts the orientation of all surfaces in $A$. `merge(`$A$`,`$B$`)` combines all polygons in $A$ and $B$.

An application of this procedure on the example in figure 4 shows that dangling edges are

[10] `http://evanw.github.com/csg.js/`

removed due to the invariance of the boundary to inversion and thus eventually clipped away.

In csg.js, these operations are implemented on BSP trees as arguments and result. Primitives are instantiated by setting up a BSP with the appropriate (planar) halfspaces. Cubes and other polygonal solids are exactly represented that way, though primitives with curved surfaces like spheres and cylinders have to be approximated by a polygon with a given resolution. This approach of *early polygonalization* simplifies the implementation by only operating on planar halfspaces and their linear intersections, at the cost of introducing a large number of halfspaces into the model for every curved primitive.v

### 3.3.2 OpenJsCad

OpenJsCad[11] is an implementation of the OpenSCAD approach in JavaScript for web-based CSG modelling, using WebGL for visualization.

Unlike OpenSCAD, there is no shortcut algorithm used for preview. A polygonal boundary is evaluated in $\mathbb{R}^3$ for visualization via WebGL as well as for exporting the model in STL format.

OpenJsCAD extends upon the parametric construction approach of OpenSCAD by exposing the CSG primitives and Boolean operations as objects and methods in JavaScript and so the CSG description becomes in essence a domain-specific language inside JavaScript.

[11] `http://joostn.github.com/OpenJsCad/`

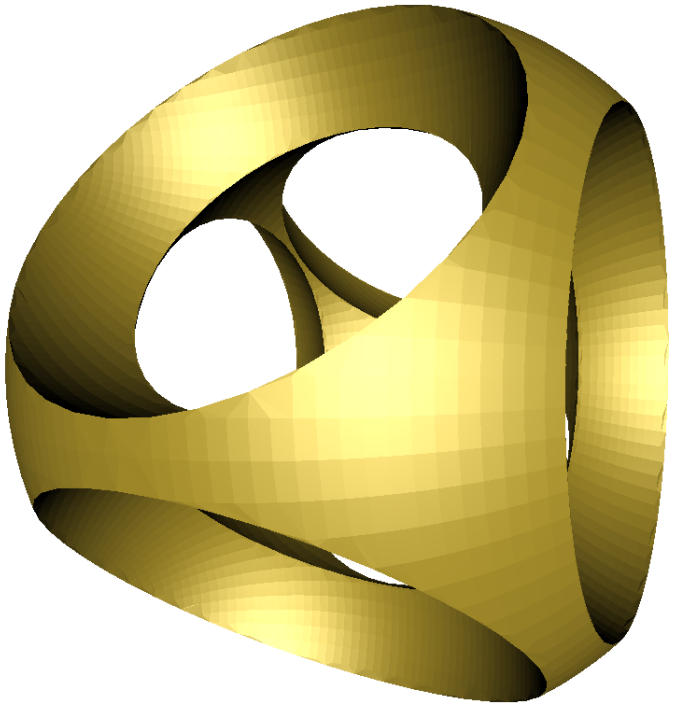


Figure 14: Polygonalization of the CSG description
`difference(`
`  Sphere([0,0,0],1),`
`  union(Sphere([0,0,0],0.5),` $\{$`Sphere`$([x, y, z], 0.7) \mid x, y, z \in \{-1, 1\}\}))$

# 4 Surface Graph Representation

The primary intent of this work is to explore how CSG on the web can be improved. With the implementation of csg.js as a starting point and its restriction to primitives composed from planar halfspaces, improving the capabilities for curved primitives suggested itself as a promising task.

In order to keep the implementation as simple as possible, the presented approch is limited to spheres although a generalization to quadrics or other algebraic surfaces is hopefully possible.

The representation utilized in csg.js, a BSP tree with planar separators annotated with the polygons on the plane of each node is unsuitable because we want do defer the polygonalization of the curved surfaces as long as possible.

While the CSG to BSP conversion by Buchele and Roles [7] explicitely allows curved separators in the BSP tree, it is not obvious how to polygonalize the BSP representation into a mesh of the surface. The surface of the object only consists of the BSP separators, but not all parts of the BSP separators are part of the surface of the object. Intersecting separa-

tors separate relevant and irrelevant parts of the surface. A scheme were the evaluation of relevant parts of the surface for polygonalization more obvious would be desirable.

The approach presented in this chapter uses the method for regular Boolean operations of csg.js, albeit on a datastructure that is not a BSP tree but a high-level boundary representation. This was inspired the realization in CGAL, [27] and *scene graphs*, which are commonly used to describe a scene at a high level of abstraction in 3D-modeling.

Surface graphs are a high-level boundary representation, keeping descriptive information about the geometry and topology of objects composed by Boolean operations from curved primitives. As already stated, the implementation presented below is limited to spheres as primitives.

## 4.1 Surface Graph

A surface graph consists of sets of surfaces $ss$, borders between these surfaces $bs$ and intersection points between the borders $ps$. An inversion flag $i$ especially differentiates between "nothing" and "everything".

The constructor `SurfaceGraph`$(ss, bs, ps, i)$ returns a surface graph $SG$ with its components given by the arguments. The components can be accessed with $SG$.`surfaces`, $SG$.`borders`, $SG$.`points` and $SG$.`inverted`.

A deep copy of a surface graph $SG$ is created with `copy`$(SG)$.

The operation `join` combines two surface graphs:

$$\begin{aligned}\texttt{join}(SG_1, SG_2) = \texttt{SurfaceGraph}(&SG_1.\texttt{surfaces} \cup SG_2.\texttt{surfaces},\\ &SG_1.\texttt{borders} \cup SG_2.\texttt{borders},\\ &SG_1.\texttt{points} \cup SG_2.\texttt{points},\\ &SG_1.\texttt{inverted})\end{aligned}$$

### 4.1.1 Surfaces

The basic element in a surface graph are surfaces, in our case spheres.

A sphere primitive $S$ is constructed with `Sphere`$(\vec{m},r)$ for $\vec{m} \in \mathbb{R}^3, r \in \mathbb{R}$:

$$\texttt{Sphere}(\vec{m}, r) = \texttt{SurfaceGraph}(\{(\vec{m}, r, \text{False})\}, \varnothing, \varnothing, \text{False})$$

The single surface element is a collection of parameters for a sphere by definition 2.3.4.1. Radius $r$, center $\vec{m}$ and the inversion flag can later be accessed from a surface $s \in SP$.surfaces with $s$.`radius`, $s$.`center` and $s$.`inverted`.

This is the only available primitive. Inverted spheres are not allowed as primitives (as they are not an abstract solid), so the inversion flag of surface and the inversion flag for $SP$ are set as `False`.

Figure 15 shows such a minimal non-empty surface graph of a single primitive, figure 1 shows its polygonalization.

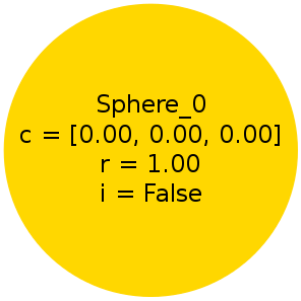



Figure 15: Surface graph of `Sphere([0,0,0],1)`

### 4.1.2 Borders

Two (two-dimensional) surfaces can intersect in a one-dimensional shape. In the case of spheres this shape is a circle. Defining parameters for these circles are stored in a surface graphs as *borders*. The borders in the SG are all circular intersections between spheres that are part of the represented object. Every border separates two spheres into two parts each. In addition to the paramaters for the circle, a border carries information which parts of each of the two bordering surfaces are part of the object or *relevant*.

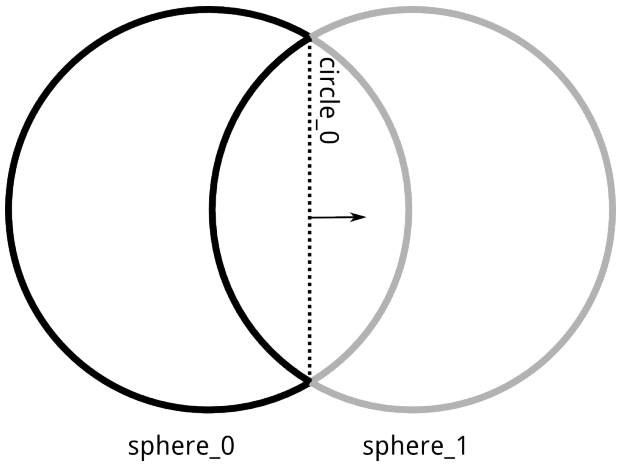



Figure 16: XY-plane of `difference(Sphere([0,0,0],1),Sphere([1,0,0],1))`. Borders are dottet and normals are given by the arrows. Not relevant parts of surfaces and borders are grey.

A border $b$ given by a circle $C$, the two intersecting surfaces $S_1$, $S_2$ and their direction flags $dir_{S_1}$, $dir_{S_2}$. It can be constructed with `Border`$(C, S_1, S_2, dir_{S_1}, dir_{S_2})$. The components can be accessed with $b$.`circle`, $b$.`side1`, $b$.`side2`, $b$.`dir1` and $b$.`dir2`.

$C$ are the parameters for a circle by definition 2.3.3.1, which can be accessed by $C$.`center`, $C$.`radius` and $C$.`normal`.

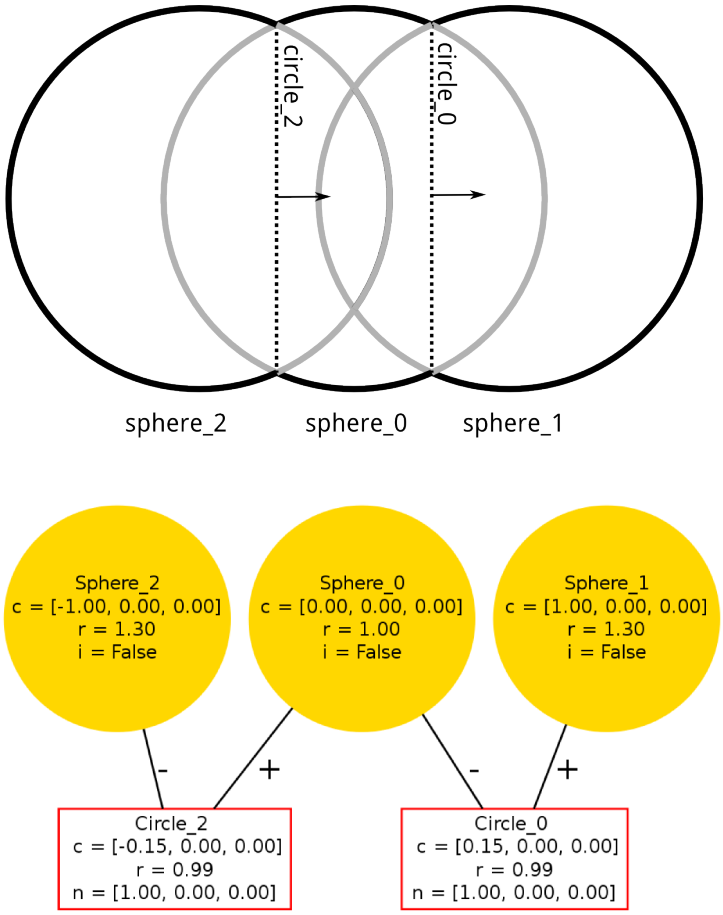



Figure 17: XY-plane (top) and surface graph (bottom) of the CSG description

```
union(
   Sphere[0,0,0],1),
   union(Sphere([1,0,0],1.3), Sphere([-1,0,0],1.3)))
```

The direction flags control which parts of the connected surfaces are relevant: If $dir_S$ is true, the part of $S$ that is in the direction of the normal of $C$ is relevant. If it is false, the part opposing the normal is. Figure 16 illustrates this principle for the difference of two spheres. A key observation is that in a valid abstract solid the two direction flags of a border are the same if the inversion flags of the two connected surfaces are different and vice versa.

Only intersections which appear in relevant parts of an object are present in a surface graph. Figure 17 demonstrates a union of three spheres. The outer two intersect, but the intersection does not appear in the surface graph, because it is an intersection of two irrelevant parts. In figure 18, the arrangement of the three spheres is the same, but the outer ones are substracted from the one in the middle. Here, all three circles of intersection appear between relevant parts of the surfaces and are thus included in the surface graph.

Figure 19 shows the result of the Boolean operations on two spheres as surface graphs and polygonalized surfaces. The circle of intersection is the same for all three operations, though

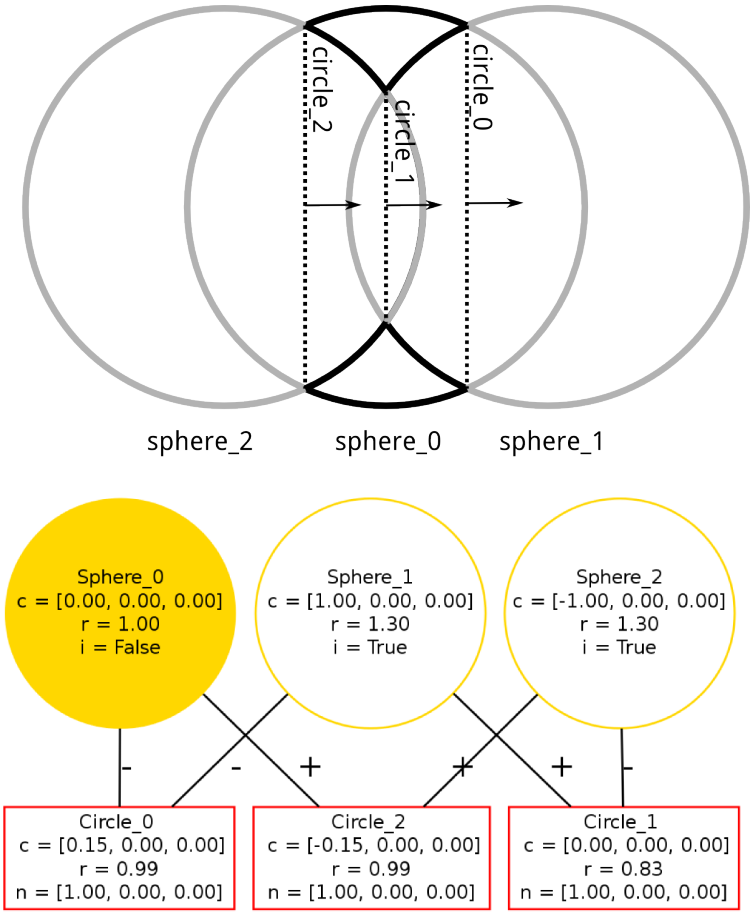



Figure 18: XY-plane (top) and surface graph (bottom) of the CSG description

```
difference(
   Sphere([0,0,0],1),
   union(Sphere([1,0,0],1.3), Sphere([-1,0,0],1.3)))
```

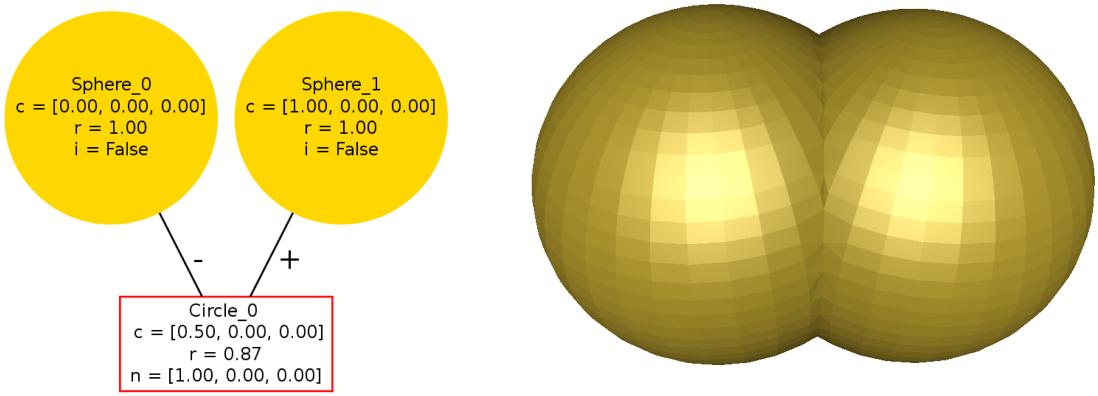
See figure 12 for its polygonalization

the directions are different, changing which parts of the surfaces are relevant and therefore polygonalized and rendered. Edges are labeled with the direction flags, "+" for the direction of the normal and "-" for the opposite.

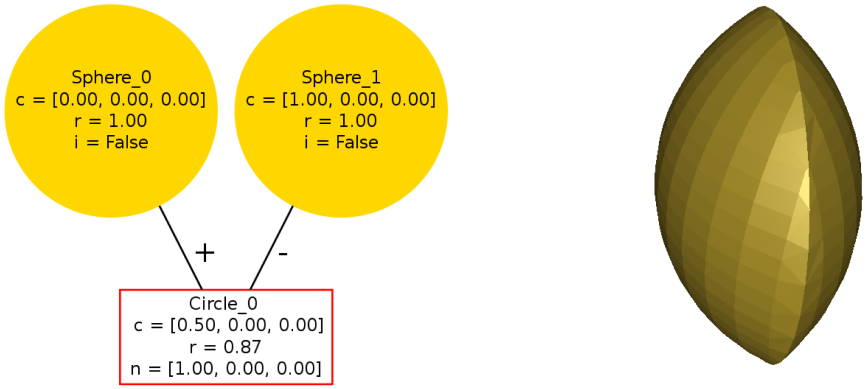
### 4.1.3 Border Points

Like spheres may intersect in circles, circles can intersect in points. These intersection points can appear, when at least three spheres intersect. Similar to borders, *border points* also have information which part of a border is relevant.

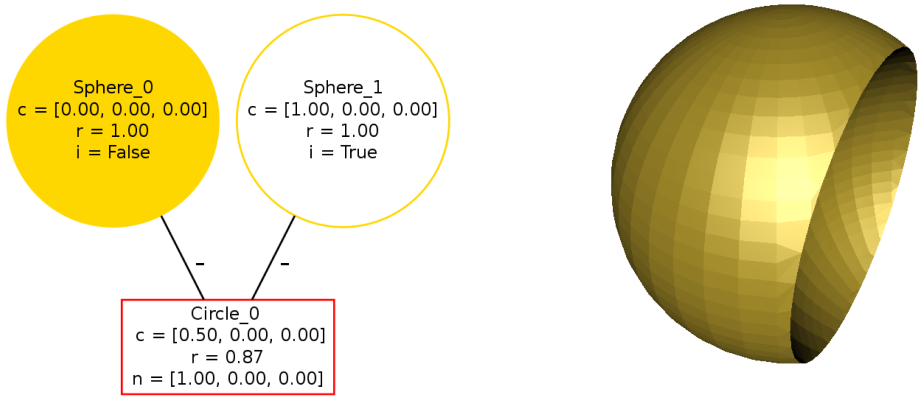
A border point $p$ is given by two points $P_1$, $P_2$, the two borders intersecting at these points $b_1$, $b_2$ and the direction flags $dir_{b_1}$, $dir_{b_2}$. It is constructed with `Border`$(P_1, P_2, b_1, b_2, dir_1, dir_2)$. The components are accessed with $p$.`point1`, $p$.`point2`, $p$.`side1`, $p$.`side1`, $p$.`dir1` and $p$.`dir2`.




Union




Intersection




Difference

Figure 19: Boolean operations on two spheres in SGR (left) and polygonalization of the resulting surface (right).

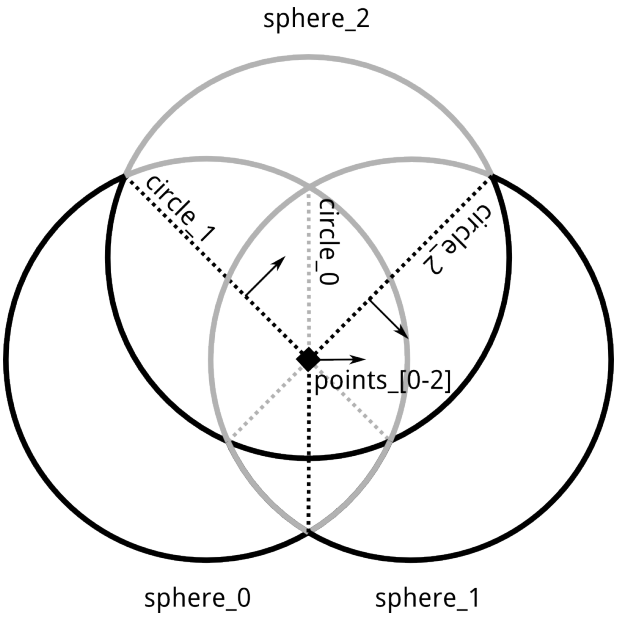



Figure 20: XY-plane of the CSG description

```
difference(
   union(Sphere([0,0,0],1),Sphere([1,0,0],1)),
   Sphere([0.5,0.5,0],1))
```

Just like the direction flags of borders control which parts of a surface are relevant, the direction flags of border points control which parts of a border are relevant. If $dir_{b_1}$ is true, the part of $b_1$ that is in the direction of the normal of $b_2$ is relevant. If it is false, the part opposing the normal of $b_2$ is relevant. As for the borders, edges connecting border points and borders are labeled with the direction flag in the surface graphs.

Unlike borders, which are unique in a surface graph for the parameters of the circle, multiple border points can appear for the same points $P_1, P_2$ between different borders. In figure 20 three pairs of identical points appear as the pairwise intersection of the three borders. Figure 21 gives the surface graph and the polygonalization for that CSG object.

In comparison with figure 22, the union of the same arrangement of spheres, it is noticable that the direction flags of the border points stay the same for both cases, only the inversion of one sphere and the direction of the border edges to it change. A consideration based on figure 20 convices that this is indeed correct, the relevant parts of the borders are the same in both cases.

As a concluding example that demonstrates how a more complex object is represented as a surface graph, figure 23 shows the surface graph for the object in figure 14 at the beginning of this chapter.

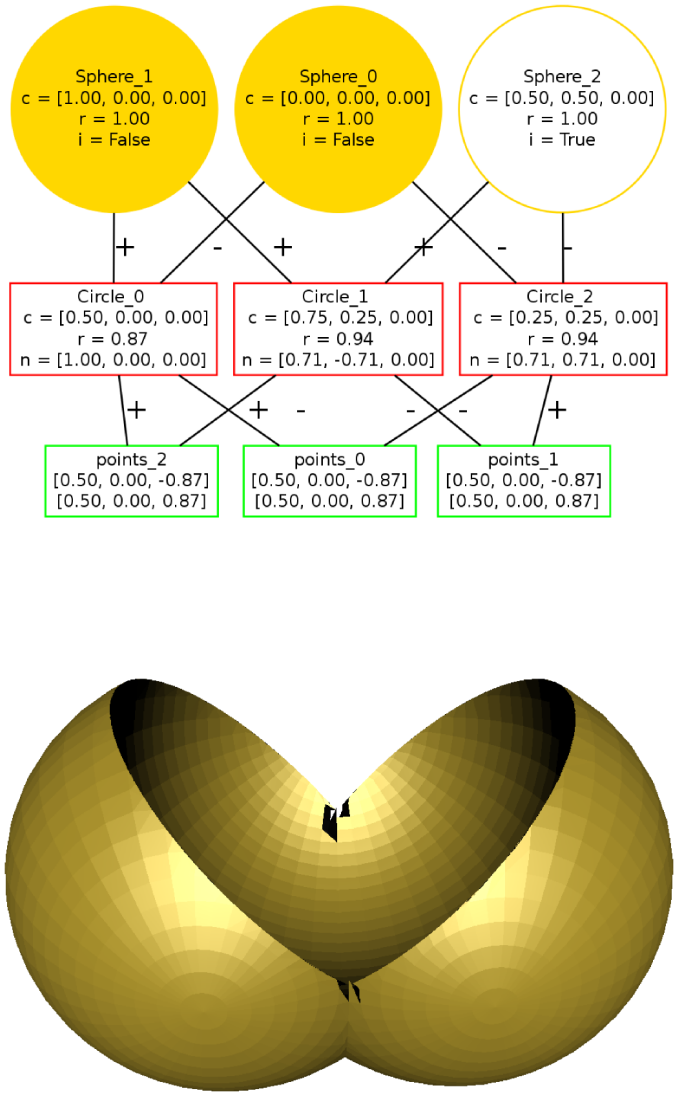



Figure 21: Surface graph (top) and polygonalization (bottom) of the CSG description

```
difference(
   union(Sphere([0,0,0],1),Sphere([1,0,0],1)),
   Sphere([0.5,0.5,0],1))
```

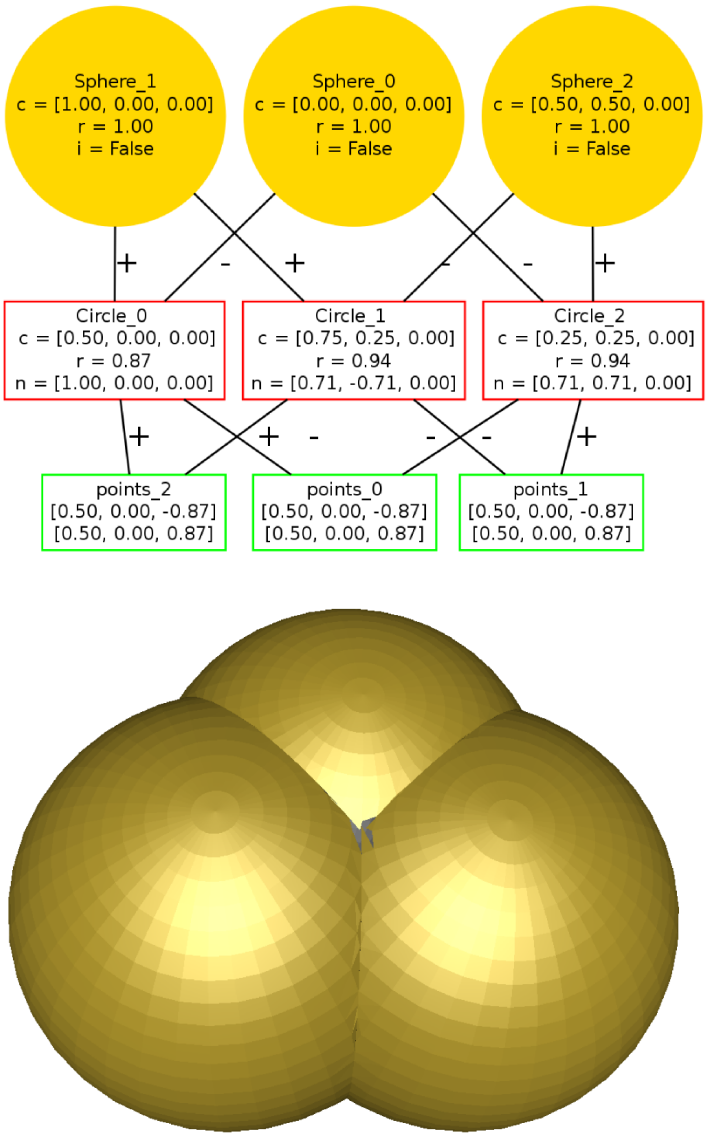



Figure 22: Surface graph (top) and polygonalization (bottom) of the CSG description

```
union(
   union(Sphere([0,0,0],1),Sphere([1,0,0],1)),
   Sphere([0.5,0.5,0],1))
```

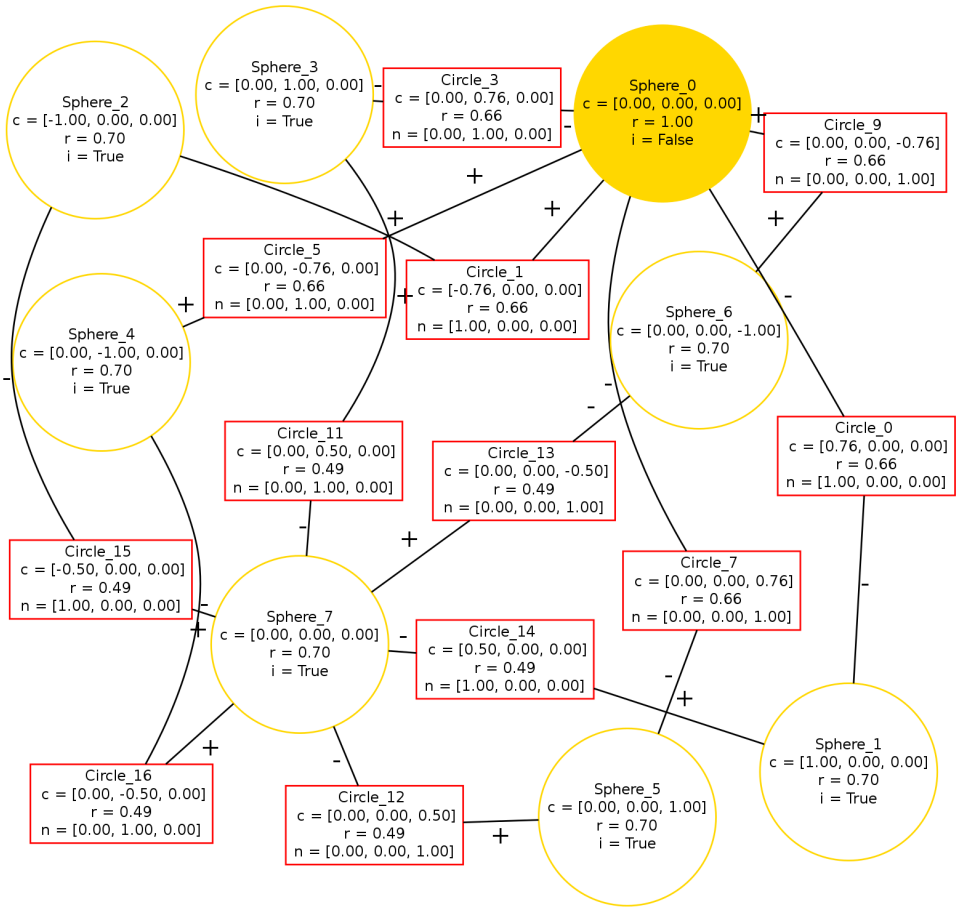



Figure 23: Surface graph of the CSG description
`difference(`
`Sphere([0,0,0],1),`
`union(Sphere([0,0,0],0.5),` $\{$`Sphere`$([x, y, z], 0.7) \mid x, y, z \in \{-1, 1\}\}))$
See figure 14 for its polygonalization

## 4.2 CSG to Surface Graph

The polygonalization of a CSG description is separated into two distinct steps. First the evaluation of the surface graph from the CSG description. Secondly the polygonalization of the surfaces in the surface graph.

The application of the Boolean operations are reduced to `complement`, `clip`, and `merge` operations, like in the csg.js implementation. But the different datastructure and approach to polygonalization requires these operations to be implemented differently.

In the following, the mechanism of these three operations is explained.

### 4.2.1 Complement

**Algorithm 4** `complement`

Input: Surface graph $A$
Output: Surface graph

$R := \texttt{copy}(A)$
**for all** $s \in R.\texttt{surfaces}$ **do**
  $s.\texttt{inverted} := \neg s.\texttt{inverted}$
**end for**
$R.\texttt{inverted} := \neg R.\texttt{inverted}$
**return** $R$

The realization of `complement`, given by 4 is very straightforward. For a given surface graph it returns a copy of the graph with the inversion flag of all surfaces and the inverion flag of the surface graph itself toggled. To prevent undesired side effects end enhance encapsulation, all presented algorithms do not modify their arguments, but return a new independet datastructure.

### 4.2.2 Clip

**Algorithm 5** `clip`

Input: Surface graphs $A, B$
Output: Surface graph

$R := \texttt{SurfaceGraph}(\varnothing, \varnothing, \varnothing, A.\texttt{inverted})$
**if** $B.\texttt{inverted} \wedge |B.\texttt{surfaces}| == 0$ **then**
  **return** $R$
**end if**
**for all** $s_A \in A.\texttt{surfaces}$ **do**
  $A_s := \texttt{just}(A, s_A)$
  $addA_s :=$ True
  **for all** $s_B \in B.\texttt{surfaces}$ **do**
    $R_{clip}, skip := \texttt{clip\_sphere}(A_s, \texttt{just}(B, s_B))$
    $R := \texttt{join}(R, R_{clip})$
    **if** $skip$ **then**
      $addA_s :=$ False
    **end if**
  **end for**
  **if** $addA_s$ **then**
    $R := \texttt{join}(R, A_s)$
  **end if**
**end for**
**return** $R$

The operation `clip(`$A$`,`$B$`)` for two surface graphs $A, B$ returns a surface graph that represents every part of $A$ that is not in $B$, similar to the (not regularized) difference ($\setminus$). If surfaces in $A$ intersect surfaces in $B$, only the new borders are created in the resulting

**Algorithm 6** `just`

Input: Surface graph $A$, surface $s$
Output: Surface graph

**if** $s \notin A.\texttt{surfaces}$ **then**
  **return** $\texttt{SurfaceGraph}(\varnothing, \varnothing, \varnothing, \text{False})$
**end if**
$bs := \{b \in A.\texttt{borders} \mid b.\texttt{side1} == s \vee b.\texttt{side2} == s\}$
$ps := \{p \in A.\texttt{points} \mid p.\texttt{side1} \in bs \wedge p.\texttt{side2} \in bs\}$
**return** $\texttt{SurfaceGraph}(\{s\}, bs, ps, A.\texttt{inverted})$

surface graph, but not the new surfaces from $B$. They are added and the borders joined in `merge`.

The implementation of `clip` (algorithm 5) starts by creating an empty surface graph $R$. It iterates through all combinations of surfaces between $A$ and $B$. `just` (algorithm 6) retrieves partial surface graphs $A_s, B_s$ with just a single surface and all its borders and border points for both of them, and compares those two in the operation `clip_sphere`. The result of this comparison is a tuple of a partial surface graph $R_{clip}$ and a flag *skip*. $R_{clip}$ are the parts that are certainly in the output and *skip* is true if the partial surface $A_s$ is certainly not part of the result. If that is not the case for all surfaces in $B$, $A_s$ is added to $R$. Finally, $R$ is returned.

In `clip_sphere` (algorithm 7), two spheres with their borders and border points are compared. All borders and border points in a SGR are created here.

The single surface in the partial surface graph $A$ is accessed with $A.\texttt{surfaces}[0]$, and likewise for $B$. The outmost if-clause differentiates between intersection and the two variants of containment (cf. property 2.3.4.3). For independence, the fallthrough where an empty surface graph and false are returned is appropriate.

In the case of intersection, the circle of intersection between the two spheres is evaluated with `get_border_circle` by the method in property 2.3.4.3. If the circle is relevent (not on irrelevant parts of $A$ or $B$), the new border $b_{new}$ is created, but only for the $A$ side, the $B$ side and direction are set to NULL. `union`, `intersection` and `difference` always call `clip` for both solids on each other and the borders will be joined in `merge`.

The borders and border points of $A$ are filtered and sorted to borders and points inside the new borders and intersecting borders. Borders and points outside are discarded. For every intersecting border, the border points are created. Finally the partial surface graph for $A$ with relevant old and new borders and border points is returned.

In the case of containment, depending on the inversion of the surfaces, $A$ might be be clipped away by $B$ and thus *skip* is set true. Additionally in some cases the complement of $B$ is a new surface in the result. [12]

[12] For example, a small inverted sphere $B$ lies within a bigger not inverted sphere $A$. Every point that lies in $A$ but not in $B$ is inside the complement of $B$.

### 4.2.3 Merge

The operation `merge` (algorithm 8) combines two surface graphs into one, similar to `join`. Two characterstics distinguish them: In `merge`, surfaces that just differ by their inversion flag are joined together, both of them are omitted in the result. Also, matching incomplete borders created in `clip_sphere` are joined together here.

## 4.3 Surface Graph to Mesh

The operation `mesh` (algorithm 9) that retrieves a list of triangles from a surface graph at a given resolution $res$ is very simple. The operation `mesh_sphere` is called for the partial surface graph for every surface and the results are joined together and returned. This points out how the actual generation of polygons in a surface graph is an operation local to each surface in the graph.

The more fundamental aspects of polygonalization happens in `mesh_sphere` (algorithm 10). A two-dimensional array $P$ of size $res \times res$ is created. Latitude and longitude for all points on the sphere depending on $res$ are evaluated, `sphere_points` gives their coordinates by property 2.3.4.2. `border_if_irrelevant` returns its first argument if that point is relevant, depending on the closest relevant border. Otherwise it returns the closest point on that border.[13] All points are stored in $P$.

After filling $P$, it is iterated through a second time. In this pass, every point and its lower-left, right, and lower neighbour are extracted. If they are *compatible* to each other, meaning they are pairwise relevant to each other, two triangles are added to the result, the order of their points depending on the inversion of the surface.

The check for compatability depends on a datastructure that tracks which point on the sphere was mapped to which border which was omitted for the sake of clarity of the algorithm.

[13] Border circles are also segmented into $res$ parts.

**Algorithm 7** `clip_sphere`

Input: Surface graphs $A, B$
Output: (Surface graph, [True | False])

$s_A, s_B := A.\texttt{surfaces}[0], B.\texttt{surfaces}[0]$
$d := s_A.\texttt{center} - s_B.\texttt{center}$
$skip :=$ False
**if** $d < s_A.\texttt{radius} + s_B.\texttt{radius}$ **then**
  $circle :=$ `get_border_circle`$(s_A, s_B)$
  **if** `is_relevant`$(circle, A, B)$ **then**
    $dir_b :=$ `get_border_dir`$(circle, s_A, s_B)$
    $b_{new} :=$ `Border`$(circle, A, \text{NULL}, dir_b, \text{NULL})$
    $(bs_{inside}, ps_{inside}, bs_{intersecting}) :=$ `sort_borders`$(b_{new}, A)$
    $ps_{new} := \{\}$
    **for all** $b_{old} \in bs_{intersecting}$ **do**
      $(p1, p2) :=$ `get_border_points`$(b_{new}, b_{old})$
      $(dir_1, dir_2) :=$ `get_border_point_dirs`$(b_{new}, b_{old})$
      $ps_{new} := ps_{new} \cup$ `BorderPoint`$(p1, p2, b_{new}, b_{old}, dir_1, dir_2))$
    **end for**
    $R :=$ `SurfaceGraph`$(\{s_A\}, bs_{inside} \cup \{b_{new}\}, ps_{inside} \cup ps_{new}, A.\texttt{inverted})$
    **return** $(R, \text{True})$
  **end if**
**else if** $d + s_A.\texttt{radius} > s_B.\texttt{radius}$ **then**
  **if** $s_B.\texttt{inverted}$ **then**
    $skip :=$ True
    **if** $\neg s_A.\texttt{inverted}$ **then**
      $R :=$ `complement`$(B)$
    **end if**
  **end if**
**else if** $d + s_B.\texttt{radius} > s_A.\texttt{radius}$ **then**
  **if** $\neg s_B.\texttt{inverted}$ **then**
    $skip :=$ True
    **if** $s_A.\texttt{inverted}$ **then**
      $R :=$ `complement`$(B)$
    **end if**
  **end if**
**end if**
**return** $(R, skip)$

**Algorithm 8** `merge`

Input: Surface graphs $A, B$
Output: Surface graph

```
ss := {}
for all s_A ∈ A.surfaces do
  add_s_A := True
  for all s_B ∈ B.surfaces do
    if s_A == inverse(s_B) then
      add_s_A := False
    else
      ss := ss ∪ {s_B}
    end if
  end for
  if add_s_A then
    ss := ss ∪ {s_A}
  end if
end for
bs := {}
for all b_A ∈ A.borders do
  for all b_B ∈ B.borders do
    if b_B.side2 ≠ NULL then
      bs := bs ∪ {b_B}
    else if b_A.circle == b_B.circle then
      bs := bs ∪ {Border(b_A.circle, b_A.side1, b_B.side1, b_A.dir1, b_B.dir1)}
    end if
  end for
  if b_A.side2 ≠ NULL then
    bs := bs ∪ {b_A}
  end if
end for
return SurfaceGraph(ss, bs, A.points ∪ B.points, A.inverted ∧ B.inverted)
```

**Algorithm 9** `mesh`

Input: Surface graph $A$, resolution $res$
Output: Triangle mesh

```
F := {}
for all s ∈ A.surfaces do
  F := F ∪ {mesh_sphere(just(A, s))}
end for
return F
```

**Algorithm 10** `mesh_sphere`

Input: Surface graph $A$, resolution $res$
Output: Triangle mesh

$F, s := \{\}, A.\texttt{surfaces}[0]$
$P[res][res]$
**for** $i := 0$ **to** $res - 1$ **do**
  $lat := i\pi/(res - 1)$
  **for** $j := 0$ **to** $res - 1$ **do**
    $long := j\pi/res$
    $p := \texttt{sphere\_point}(s, lat, long)$
    $p := \texttt{border\_if\_irrelevant}(p, A)$
    $P[i][j] := p$
  **end for**
**end for**
**for** $i := 0$ **to** $res - 2$ **do**
  **for** $j := 0$ **to** $res - 1$ **do**
    $p := P[i][j]$
    $left := P[i + 1][(j - i)\%res]$
    $right := P[i][(j + i)\%res]$
    $down := P[i + 1][j]$
    **if** $p, left, right, down$ are compatible **then**
      **if** $\neg s.\texttt{inverted}$ **then**
        $F := F \cup \{(p, left, down), (p, down, right)\}$
      **else**
        $F := F \cup \{(p, down, left), (p, right, down)\}$
      **end if**
    **end if**
  **end for**
**end for**
**return** $F$

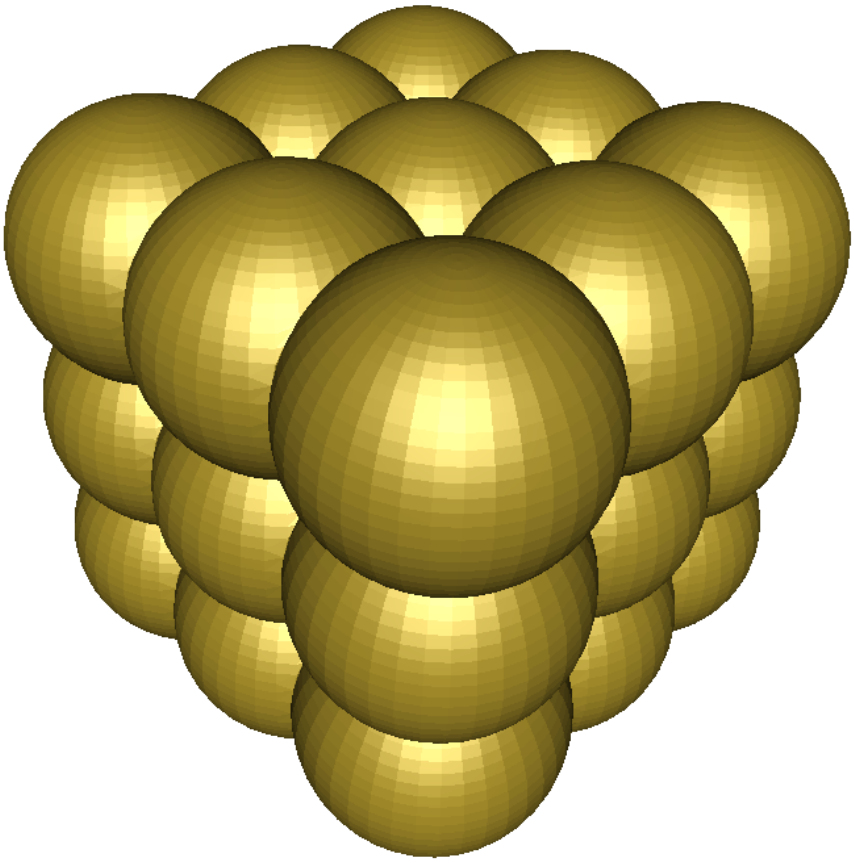


Figure 24: Polygonalization of the CSG description
`union(` $\{$`Sphere`$([x, y, z], 0.8) \mid x, y, z \in \{-1, 0, 1\}\}$ `)`

# 5 Evaluation

## 5.1 Algorithmic Complexity

Buchele and Roles [7] estimate that Thiebault and Naylor's approach [24] converts a CSG tree with $n$ nodes into a (plane-separated) BSP tree in $O(2^n)$. They state a worst-case runtime of $O(n^4)$ for their conversion to a BSP with curved separators and argue that in many practical cases it might come down to $O(n^2)$ or $O(n)$.

The same estimation seems to be true for the polygonalization by csg.js (for $n$ primitives composed of a constant number of halfspaces) as well as (for $n$ sphere primitives) for the previously presented SGR method. In the worst case for all $n^2$ parings of primitives, $n^2$ intersections have to be evaluated, giving a runtime of $O(n^4)$. However, this is only the case when all primitives intersect each other and lowers for less dense interconnected CSG constructs.

Spheres have to be approximated by planar halfspaces in csg.js. An approximation of a sphere with resolution $r$, where $r$ is the number of segments for a circle, consists of $r^2$ planar halfspaces. So it converts $n$ spheres with resolution $r$ in $O(n^4 * r^2)$ time.

In SGR, the evaluation of the surface graph is independent from the resolution. Only the polygonalization step, which operates on $n$ surfaces and has to evaluate $n^2$ borders and border points in the worst case depends on the resolution, and runs in $O(n^3 * r^2)$ time. So all together, it polygonalizes an object described from a CSG tree of $n$ spheres in $O(n^4 + n^3 * r^2)$ time at the worst.

## 5.2 Profiling

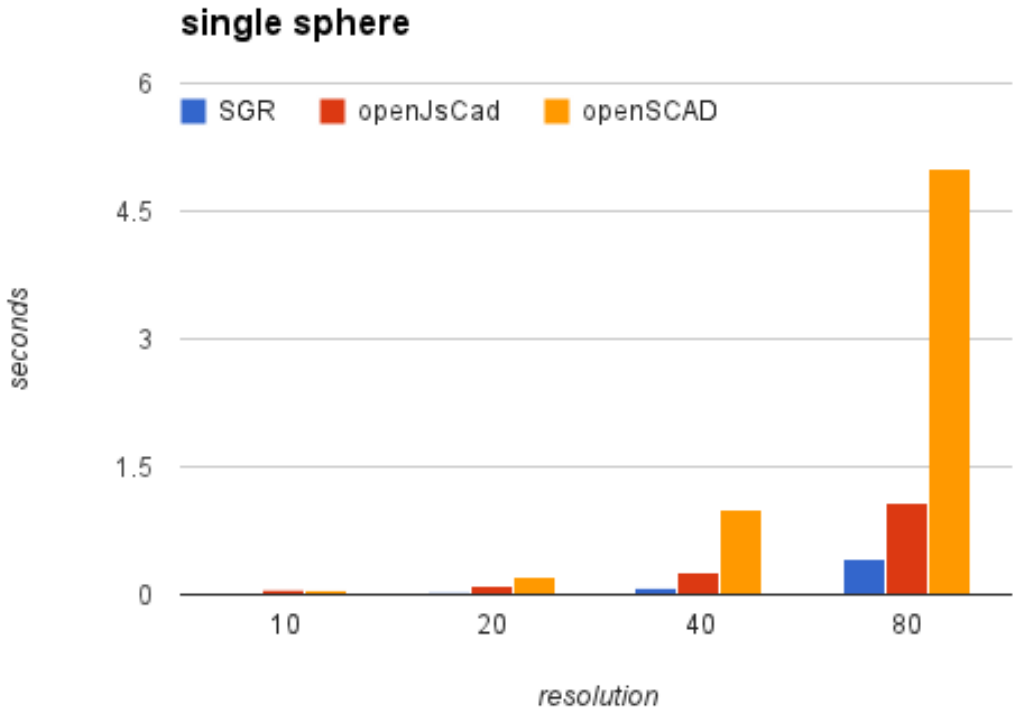



Figure 25: Runtime for polygonalization of a single sphere with variable resolution in SGR, openJsCad and openSCAD.

To estimate real-world performace differences, several CSG constructs were polygonalized with SGR, openJsCad and openSCAD and the runtime of each different approach was measured. In openSCAD, the CGAL evaluation was used, as it constructs the mesh in $\mathbb{R}^{\not\vdash}$. The image-space preview rendering was very fast for all test cases, in the range of several frames per second.

The first scene is a single sphere, which was polygonalized with different resolutions. The

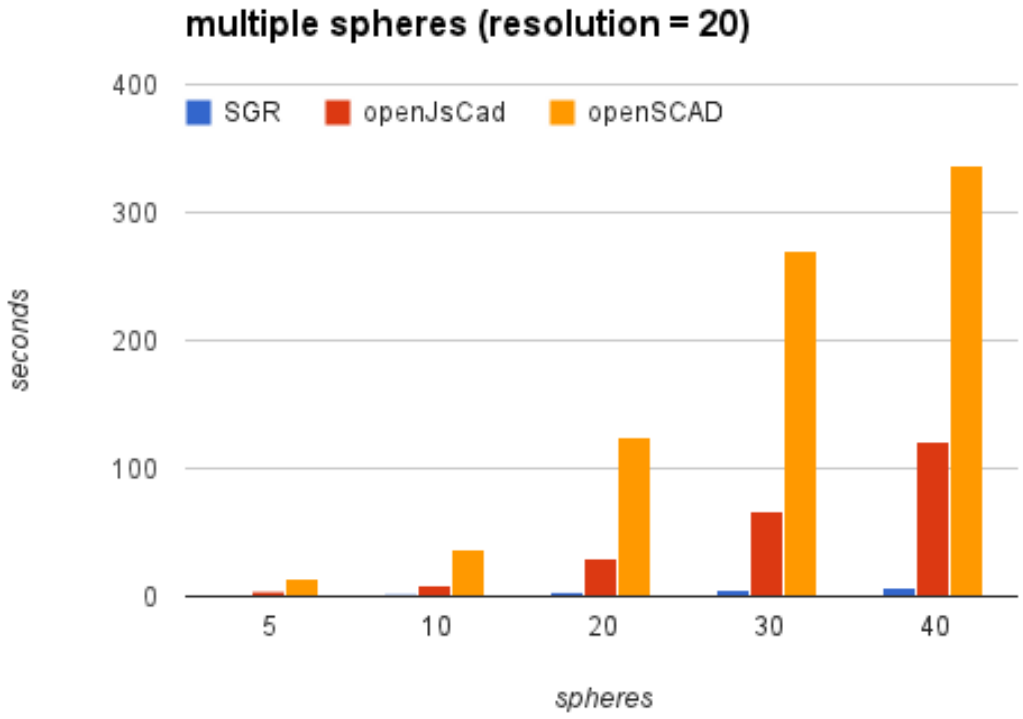



Figure 26: Runtime for polygonalization of a variable amount of spheres with resolution 20 in SGR, openJsCad and openSCAD.

results are presented in figure 25.

To evaluate the influence of the number of primitives, the second test case is the union of a varying number of spheres arranged along the X-axis each intersecting their neighbours. See figure 26 for the results.

An arrangement of 27 spheres in a cube with intersecting borders was chosen to stand for heavily interconnected objects. It was polygonalized again in different resolutions and results are given in 27.

Common to all the test cases is that the SGR approach is faster than both contestors in every case, the advance increasing for higher resolutions and higher numbers of primitives. This advantage may not be overrated; it comes at the price of high specialization. The limitation to spheres as primitives strongly limits any practical applications. It shows that handling of curved primitives in a abstract way can severly improve performance though. Even if an extension of SGR to quadric primitives complicates the evaluation of the surface graph, early approximation to planes appear prohibitivly slow for significant numbers of primitives and high resolutions.

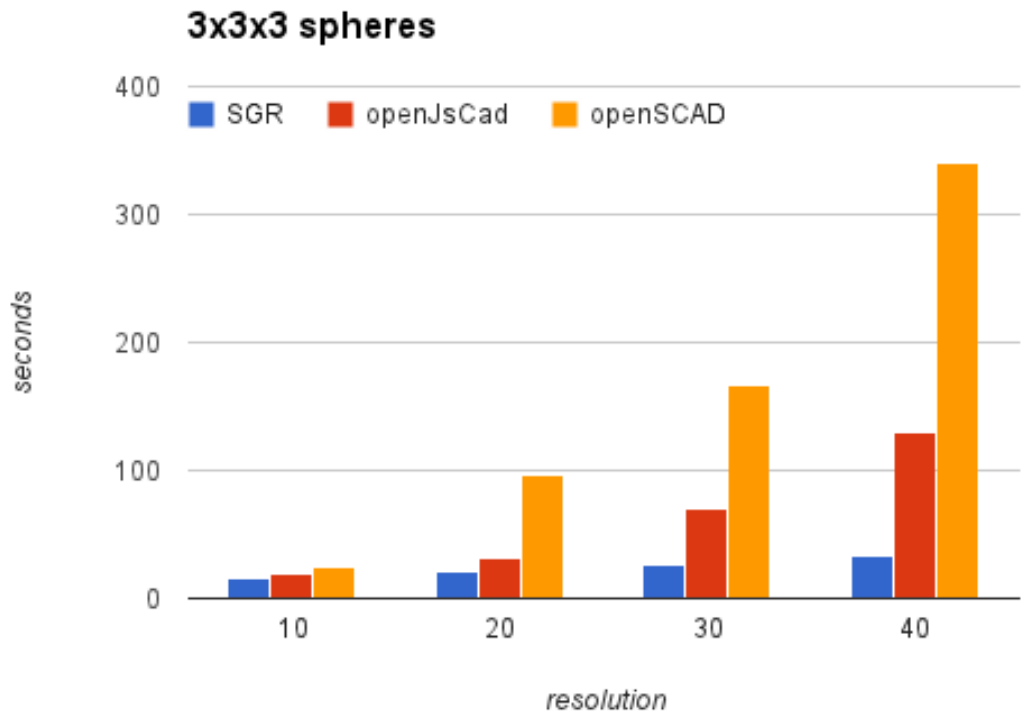



Figure 27: Runtime for polygonalization of 3x3x3 (=27) spheres arranged in a cube with variable resolution in SGR, openJsCad and openSCAD.

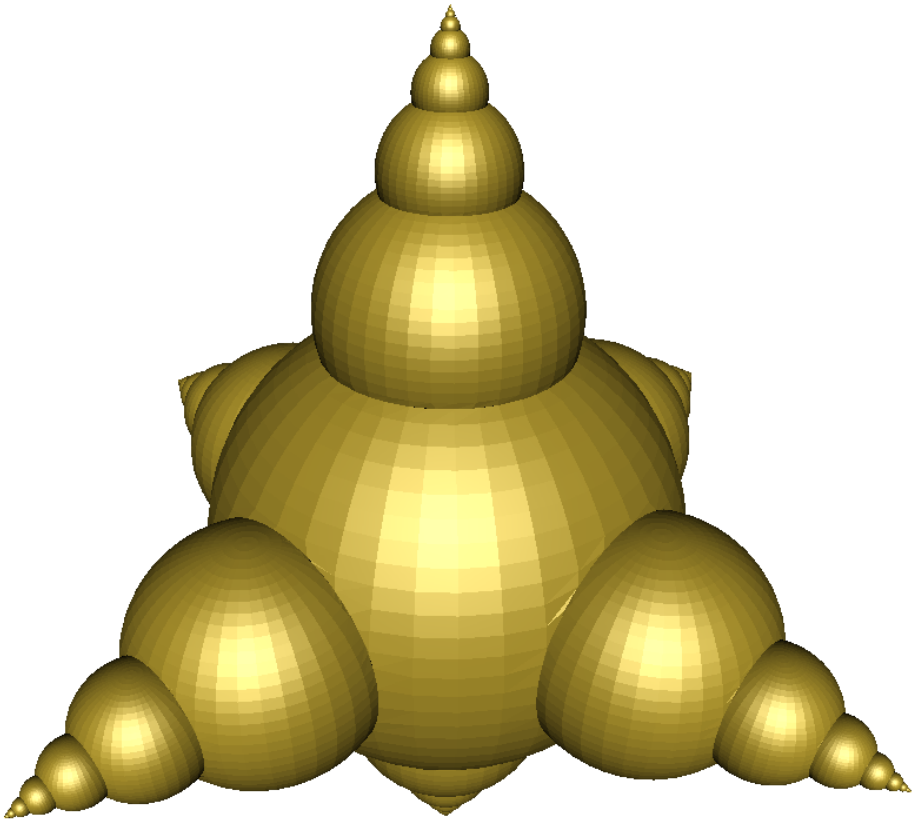


Figure 28: Polygonalization of a fractal CSG description.

# 6 Future Work

**Quadrics** The presented SGR approach to polygonalize CSG descriptions is limited to spheres, but demonstrates that a datastructure that allows parametric representation of curved primitives significantly improves performance compared with an approximation by planes. An generalization to quadric surfaces would allow an exact, parametric representation of the most common CSG primitives and an efficient application of the Boolean operations on them. This seems to be the most pressing task to turn SGR from a proof of concept to a tool actually useful in practice.

The most drastic changes would affect the operations `clip_sphere` and `mesh_sphere`, as they would have to be almost completely rewritten for a correct handling of quadrics. Borders would be generalized from circles to intersections of quadric surfaces and border points to intersections of those intersections. The method for evaluting relevant parts would have to be adapted to those more general objects, but the fundamental approach of SGR would stay the same.

**Correctness of SGR** While this work demonstrates the feasability of the SGR approach, the presented implementation performs rather intricate manipulations whose results have to to carefully coordinated to yield useful results. A few missing surfaces in figures 21 and 22 are evidence that at least some bugs escaped the author's scope. A more rigorous and reduced formalization would certainly be beneficial.

**Web applications** Another improvement, not just for SGR but for 3D modeling in general would be the creation or extension of *rich internet applications* (RIAs) [10] for solid modeling, be it CSG or other schemes. WebGL provides fast client-side visualization of the models and integration into the Internet allows simple collaboration and distribution of models.

**Variable resolution and caching** A surface graph can be evaluated independently from a resolution paramater. It would be possible to once evaluate the SGR and create several polygonalizations with increasing resolution. During interactive modeling, the user would get a quick rough estimation of the object currently edited which would become increasingly more precice.

Independently, polygonalizations of parts of a surface graph, namely surfaces with all their borders and border points, could be cached after evaluation, preferably in a format that is invariant to transformations like translation, rotation and scaling.

**Convergence with slicing algorithms** Often a modeled object is inteded for some kind of CAM process, e.g. to be printed on a 3D-printer or cut in a lasercutter. In that cases, it is necessary to slice the final object into layers and then further reduce it into paths on those layers for the toolhead of the machine to travel along.

An interesting application of SGR would be to omit the boundary evaluation in that case and directly evaluate those layers and paths from the surface graph. Exact paths on curved boundarys could be evaluated this way, while export as a polygon mesh and subsequent slicing lead inevitably to a loss of precision.

**Conclusion** The general idea of improving 3D modeling on the web developed into a journey through logic, set theory and geometry. But the results of the trip seem promising enough to carry on. A more robust, more general version of SGR might actually become a useful tool for CSG modeling, on the web and elsewhere.

# List of Figures

# List of Algorithms

## References

[1] http://www.vorlesungen.uni-osnabrueck.de/informatik/ifc2000-01/pvs/html/50.html. Retrived 20 December 2012.

[2] http://evanw.github.com/csg.js/. Retrived 20 December 2012.

[3] Constructive solid geometry. http://en.wikipedia.org/wiki/Constructive_solid_geometry. Retrived 20 December 2012.

[4] Polygon mesh. http://en.wikipedia.org/wiki/Polygon_mesh. Retrived 20 December 2012.

[5] Solid modeling. http://en.wikipedia.org/wiki/Solid_modeling. Retrieved 20 December 2012.

[6] WebGL specification. https://www.khronos.org/registry/webgl/specs/1.0/. Retrived 20 December 2012.

[7] S.F. Buchele and A.C. Roles. Binary space partitioning tree and constructive solid geometry representations for objects bounded by curved surfaces. In *CCCG*, pages 49–52, 2001.

[8] L. Dupont, S. Lazard, S. Petitjean, and D. Lazard. Towards the robust intersection of implicit quadrics. *Uncertainty in Geometric Computations*, pages 59–68, 2002.

[9] A. Fabri, G.J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel: A basis for geometric computation. *Applied Computational Geometry Towards Geometric Engineering*, pages 191–202, 1996.

[10] P. Fraternali, G. Rossi, and F. Sánchez-Figueroa. Rich internet applications. *Internet Computing, IEEE*, 14(3):9–12, 2010.

[11] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys (CSUR)*, 23(1):5–48, 1991.

[12] R.A. Goldstein and R. Nagel. 3-D visual simulation. *Simulation*, 16(1):25–31, 1971.

[13] C.M. Hoffmann. *Geometric and solid modeling: an introduction*. Morgan Kaufmann Publishers Inc., 1989.

[14] T. Hull. HP Lovecraft: a horror in higher dimensions. *Math Horizons*, 13(3):10–12, 2006.

[15] F. Kirsch and J. Döllner. OpenCSG: a library for image-based CSG rendering. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, pages 49–49. USENIX Association, 2005.

[16] A. Pasko, V. Adzhiev, A. Sourin, and V. Savchenko. Function representation in geometric modeling: concepts, implementation and applications. *The Visual Computer*, 11(8):429–446, 1995.

[17] A. Requicha. Representations of rigid solid objects. *Computer Aided Design Modelling, Systems Engineering, CAD-Systems*, pages 1–78, 1980.

[18] A. Requicha. Solid modeling: A historical summary and contemporary assessment. *IEEE COMP. GRAPHICS & APPLIC.*, 2(2):9–24, 1982.

[19] A. Requicha and R. Tilove. Mathematical foundations of constructive solid geometry: General topology of regular closed sets. *Technical Memo*, 27(3):1–29, 1978.

[20] A. Ricci. A constructive geometry for computer graphics. *The Computer Journal*, 16(2):157–160, 1973.

[21] L. Roscoe et al. Stereolithography interface specification. *America-3D Systems Inc*, 1988.

[22] J.R. Rossignac and A.A.G. Requicha. Solid modeling. 1999.

[23] N. Stewart, G. Leach, and S. John. Linear-time CSG rendering of intersected convex objects. *Journal of WSCG*, 10(2):437–444, 2002.

[24] W.C. Thibault and B.F. Naylor. Set operations on polyhedra using binary space partitioning trees. In *ACM SIGGRAPH Computer Graphics*, volume 21, pages 153–162. ACM, 1987.

[25] M.S.G. Tsuzuki, F.K. Takase, M.A.S. Garcia, and T.C. Martins. Converting CSG models into meshed B-Rep models using euler operators and propagation based marching cubes. *Journal of the Brazilian Society of Mechanical Sciences and Engineering*, 29(4):337–344, 2007.

[26] E.W. Weisstein. Spherical coordinates. From MathWorld–A Wolfram Web Resource. `http://mathworld.wolfram.com/SphericalCoordinates.html` Retrived 20 December 2012.

[27] B. Wyvill and K. van Overveld. Polygonization of implicit surfaces with constructive solid geometry. *International Journal of Shape Modeling*, 2(04):257–274, 1996.