Universität Hamburg
DER FORSCHUNG | DER LEHRE | DER BILDUNG

BACHELOR THESIS

Multi-Material Support in OpenSCAD

vorgelegt von

Nicolaus Johannes Leopold gen. Eggert

MIN-Fakultät

Fachbereich Informatik

Technische Aspekte Multimodaler Systeme

Studiengang: Informatik

Matrikelnummer: 6423705

Erstgutachterin: Dr. Norman Hendrich

Zweitgutachter: Dr. Florens Wasserfall

Abstract

Abstract

OpenSCAD is an open source computer aided design package which is commonly used to model 3D-printed objects. It currently offers no native support for designing objects composed of more than one material, despite the fact that hardware capable of producing such objects has existed for many years now. The objective of this thesis is to add multi-material capabilities to the OpenSCAD software, allowing design and export of such objects without having to resort to workarounds. For this, new operations are introduced into the OpenSCAD script language to assign attributes to parts of the modeled object.

Zusammenfassung

OpenSCAD ist ein quelloffenes Programm für rechnerunterstütztes Konstruieren, welches häufigerweise zum Modellieren von 3D-gedruckten Objekten benutzt wird. Es bietet derzeitig keine Möglichkeiten zum Entwerfen von Objekten, welche aus mehr als einem Material bestehen, obwohl Hardware zum Produzieren solcher Objekte bereits seit vielen Jahren existiert. Das Ziel dieser Arbeit ist Unterstützung für Entwürfe bestehend aus mehreren Materialien innerhalb OpenSCADs zu bieten. Um dieses Ziel zu erreichen werden neue Operationen in die OpenSCAD Scriptsprache eingeführt, welche es erlauben die Attribute von Teilen des modellierten Objektes zu definieren.

Contents

Acronyms

3MF 3D manufacturing format

AMF Additive manufacturing file format

AST Abstract syntax tree

CAD Computer-aided design

CGAL Computational Geometry Algorithms Library

CSG Constructive solid geometry

FDM Fused deposition modeling

SVG Scalable Vector Graphics

XML Extensible Markup Language

1 Introduction

In recent years, 3D-printing (also sometimes called additive manufacturing) has become increasingly accessible to the average person, with an explosion of affordable hardware options being released for the hobby market, and both software and hardware offerings maturing considerably. A healthy open source ecosystem has sprung up around the technology not just in terms of software, but also in the form of many open source hardware options, kicked off in large part by the RepRap Project, which aimed to create and distribute a rapid prototyping machine, "designed to be able to print out a significant fraction of its own parts automatically." [1]

3D Printers can be based on one of a variety of technologies, which all work based on the principle of building up an object from a material, rather than by removing material [2]. The contents of this thesis are going to assume the process for creating these objects to be fused deposition modeling (FDM), though the specific technology used is of almost no relevance, since the focus is on modeling software, not the physical process of creating objects. FDM is a process by which a material (usually a thermoplastic material, in filament form) is melted and directly extruded into the desired shape, and built up into an object. This usually happens layer by layer, though not necessarily [3].

In the case of FDM, making objects composed of multiple materials or colors is, from a hardware perspective, as simple as using multiple extruders to build up the desired object from different filaments. This allows for the combination of elastic and non-elastic materials to have rubberized parts fused to otherwise solid components, water soluble support materials, or simply multi-colored objects.

Two main types of software are commonly used in conjunction with 3D printing: A computer-aided design (CAD) package for modeling the desired object, and a slicer which will translate the object geometry into G-Code, which is essentially a series of instructions for the 3D-printing hardware, which will result in the desired object being produced [4].

1.1 Motivation

While the ecosystem for 3D Printing has matured considerably over the last two decades with both hardware and software becoming increasingly reliable and easy to use, multi-material printing is comparatively still relatively immature. Despite hardware options for multi-material prints being available for several years now, commonly used CAD software and slicers still offer limited but growing support for multi-material printing.

OpenSCAD in particular is a popular CAD software package for the design of objects intended to be manufactured additively. Though the software supports defining multiple colors for different geometric shapes during modeling for organizational purposes, such information is

not preserved upon actual export of the models into slicer compatible file formats. The user-defined colors are also currently only rendered during preview mode, and their semantics are ill-fitted to modeling the properties of a geometry. There is also no support for differentiating between parts or materials of exported objects by other means.

The final geometry generated by OpenSCAD treats the whole modeled object as one homogeneous mass, despite supporting formats such as the additive manufacturing file format (AMF) and 3D manufacturing format (3MF), which are capable of describing non-homogeneous objects. Therefore, users of OpenSCAD are unnecessarily restricted in their ability to model multi-material prints using the software. While it is still technically possible to model such objects in OpenSCAD by exporting the individual components of the desired object separately, it unnecessarily complicates the workflow for creating such objects.

OpenSCAD works based on the principle of constructive solid geometry (CSG), which is the process of describing basic geometric shapes and performing boolean operations (as well as some other algorithmic transformations) on those shapes to combine them into complex three-dimensional geometries. This process happens via a domain specific language in OpenSCAD. As previously mentioned, while this language already supports defining colors for components, this information is not retained when actually exporting the objects in question, which limits the possibilities for which this tool can be used.

1.2 Objective

The goal of this thesis is to create native support for modeling and exporting multi-material objects in OpenSCAD to address this deficiency.

There is a longstanding feature request on the OpenSCAD issue tracker discussing the possibility of enabling support for multi-color and multi-material prints by adding new operations to allow the definition of different parts, materials and colors within an OpenSCAD model. The proposed operators match the different properties supported by the AMF and 3MF file formats. [5] The aforementioned feature request proposes an expansion of the color operation in OpenSCAD (and addition of new operations for assigning different materials and parts within a model), to be used to describe attributes of different geometric parts the final exported object.

New semantics are also needed for dealing with CSG operations on geometries of different attributes. Since OpenSCAD works by building up objects from separate geometries, rules have to be defined for combining such geometries, if their attributes do not match. Because the color operation already exists, but currently has to effect on the final generated geometry, attention has to be paid to backwards compatibility with existing scripts when defining its new semantics.

In order to achieve the stated goal of allowing multi-material support in OpenSCAD, it would technically be enough to limit the scope of this thesis to either the modification of the color operation, or introduction of the material operation by itself (the latter avoiding issues of backwards compatibility entirely). However, all three operations will necessitate touching roughly the same code paths. This is also the stated reason for them being grouped together in the feature request to begin with, so it should not pose significant additional difficulty to

implement all of them.

Currently OpenSCAD is written under the assumption that its aim is to always generate exactly one homogeneous geometry, and thus that any CSG operation will also only ever result in one single geometry. Additionally, it is assumed that OpenSCAD scripts and modules within these scripts which have multiple geometries at their root can safely be merged into one with a union operation. Since the goal is to be able to create multiple distinct geometries with different attributes like material alongside one another, we need to be able to represent results of operations which may contain more than one geometry.

Within the OpenSCAD codebase, resulting geometries are represented as a geometry object, which comes in a few different varieties. Luckily, one of the possible options is a list of geometries, which contains one or multiple subgeometries. However, these objects are currently only supported in very narrow parts of the codebase, as they are expected to only be generated in the final step of evaluating a geometry, and only if a currently experimental feature called lazy-union is enabled. Experimental OpenSCAD features are not yet fully committed to the code-base and disabled by default [8]. All geometry-modifying operations assume that they will never interact with such a geometry list, and do not support them. Thus, the ability to handle geometry lists within the geometry generation code needs to be implemented.

Additionally, geometries need to store attributes (of part, color and material), and need to retain them as they are manipulated and combined with other geometries into new ones. Semantics for interactions of geometries with different incompatible attributes also need to be defined and implemented.

Finally, these attributes then need to be respected when exporting OpenSCAD models into file formats that support them.

2 Fundamentals

2.1 Multi-Material 3D Printing

Multi-material 3D printing is the process of producing an object composed of different materials, using additive manufacturing technologies. Though the chosen frame of reference for this thesis is FDM, there are many other additive technologies which are also capable of producing multi-material objects, like e.g. selective laser sintering and material jetting. Multi-material prints are useful for a wide range of applications, such as multicolored prints, objects composed of dissimilar materials with different mechanical properties, embedding conductive filament within objects to create circuits, and creating easily removable support structures to aid additive manufacturing [6].

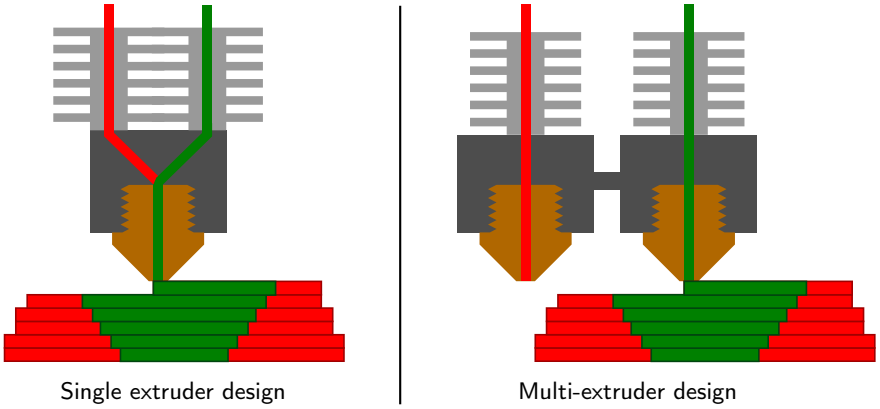


Figure 2.1: Two different designs for multi-material FDM printers

There are multiple ways FDM systems with multi-material support can be created. One possibility is to simply have multiple separate nozzles which each extrude their own filament, like in the system developed by Abilgaziyev et al. [7] An illustration of such a system can also be seen in fig. 2.1 on the right. Another option is to simply feed multiple filaments through one nozzle (seen on the left), this approach requires purging the nozzle whenever a material is changed.

Currently, if one wishes to design a multi-material object in OpenSCAD, one possibility is to design each material as a separate files, and to import all of them into the slicer and combine them there. This is not practical unless the connection surfaces of the different materials are extremely simple in shape. It also means that adjustments to the mating area have to be made once for each file representing the different materials.

A more practical approach is to design the object as one file with all the disparate materials separated in the code, making it easy to manually disable all but one material by modifying the script, and then exporting each material individually to separate files. Just as with the other method, these files are then recombined inside the slicer software.

If direct support for multi material prints were to exist within OpenSCAD, this extra step which also represents a potential source of user errors could be eliminated.

2.2 Constructive Solid Geometry

Constructive solid geometry is the underlying principle which OpenSCAD is based on. It is the process of building up complex shapes or surfaces by combining more primitive ones using boolean operations. This process can also be represented as a tree of operations and geometric primitives. Each leaf node in this tree is a geometric primitive, and all inner nodes are operations on one or more of those geometries. [9] In OpenSCAD, this tree of geometric primitives and operations is entered by the user in a bespoke script language.

Some examples of possible CSG operations are:

- **Intersection**, which results in a geometry consisting of only the space that all geometries which the operation is performed on share in common, i.e. the space where they overlap.
- **Union**, which simply merges its geometries into one.
- **Difference**, in which one or multiple geometries are subtracted from the first, leaving only the parts of the first geometry where it does not overlap with any of the others.

Many other operations are also supported in the OpenSCAD script language, including more complex ones like the Minkowski sum. The geometric primitives are simple shapes such as spheres, cylinders, cuboids, etc. More complex shapes can also be used as primitives by importing them from external files.

An example of a Geometry being built up using constructive solid geometry is pictured in fig. 2.2. In the OpenSCAD script language, the series of operations needed to produce this shape would look like the code in listing 2.1. Additional rotation operations are needed to produce cylinders facing in each cardinal direction.

```
$fn = 100; // Set the level of detail for the model
difference() {
  intersection() {
    cube([1, 1, 1], center = true);
    sphere(r = 0.7);
  }
  union() {
    rotate([0, 90, 0]) {
      cylinder(h = 2, r = 0.4, center = true);
    }
    union() {
      cylinder(h = 2, r = 0.4, center = true);
```

```
13        rotate([90, 0, 0]) {
14          cylinder(h = 2, r = 0.4, center = true);
15        }
16      }
17    }
18  }
```

Listing 2.1: An OpenSCAD script which produces a geometry close to that in fig. 2.2

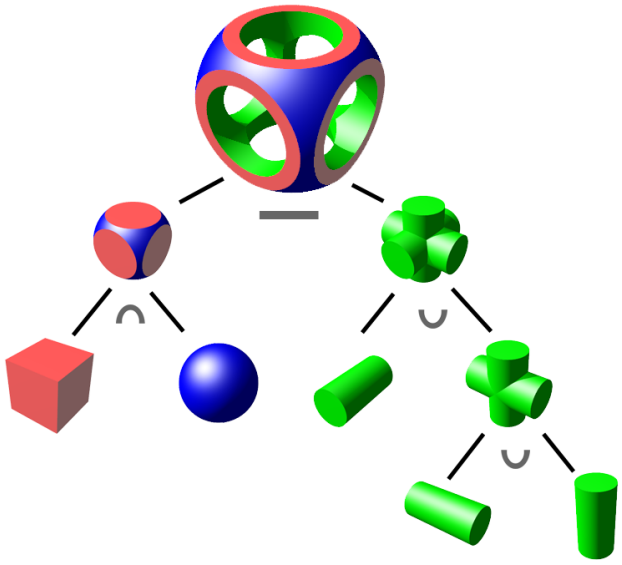


Figure 2.2: Illustration of a CSG tree, using union (∪), intersection (∩) and difference (−) operations, created by Wikimedia user Zottie [10]

Constructive Solid Geometry can apply to both two-dimensional shapes and three-dimensional objects. In its two-dimensional form, it's also commonly used in vector image creation software like Inkscape, but OpenSCAD also supports two-dimensional operations (though not the combination of two-dimensional and three-dimensional geometries, unless one is first converted to the other with operations like `projection()` or `linear_extrude()`).

It's worth pointing out that the example in fig. 2.2 assumes all geometries to be centered in the root of the Cartesian coordinate system. This is not necessarily always the case, and if two geometries that do not overlap are combined into one geometry with e.g. a `union()` operation, they will still consist of two disjunct geometrical shapes, just within the same "geometric object".

It should also be noted that while an OpenSCAD script can contain more than one CSG tree in the sense that there can be multiple commands at the root level of an OpenSCAD script. Internally all of these trees will be grouped into a single tree with an implicit root node, and in current release versions, this root node will also combine all geometries into a single one, acting as a `union()` operation at the top level of the OpenSCAD script.

2.3 OpenSCAD

OpenSCAD is a free and open source CAD software for creating models of solid three-dimensional objects (although support for two-dimensional models also exists). Instead of interactive modeling like in other CAD software, object geometries are described via a domain specific script language. This also allows creating parametric models, whose properties can be controlled by configurable parameters. OpenSCAD is a popular open source choice for modeling functional objects intended to be 3D-printed [11].

As previously mentioned, the underlying modeling technique is constructive solid geometry, though extrusion of two-dimensional outlines into three-dimensional shapes is also supported [12]. Internally, OpenSCAD represents modeled objects as a CSG tree, which also includes color information, if the `color()` operation is used. During preview of an OpenSCAD script (in the menu via "Design" → "Preview" or by pressing F5), this CSG tree will be used to display the geometry via the OpenCSG library, while trying to skip geometry calculations where possible, and instead performing operations like `difference()`, `union()` and `intersection()` directly during rendering. In preview mode, any geometries which have been given colors via the `color()` operation are also rendered in the specified color.

For the generation of the final geometry in render mode (in the menu via "Design" → "Render" or by pressing F6), the Computational Geometry Algorithms Library (CGAL) is used instead, to first calculate a single combined geometry from the CSG tree, and then to display it, and potentially export it into a file. This is done by traversing the CSG tree and generating a geometry object using the necessary CGAL methods to do the various calculations required to combine geometries. For operations on two-dimensional geometries, the Clipper library is used for this purpose instead. This step does not retain color information at all, which only has an effect on preview mode.

Currently, this process will also implicitly union the entire object into a single geometry. This geometry object is used for both rendering, and as a basis for file export (though there are also some file formats that do not use this object, but they are not relevant to this thesis).

2.4 The OpenSCAD Codebase

What follows is a high-level description of how the OpenSCAD codebase is structured, limited to the parts relevant to this thesis.

Let's imagine an OpenSCAD script was entered in the text editor integrated into OpenSCAD. The geometry this script produces is now supposed to be rendered before exporting it, and the option "Design" → "Render" is selected (or the shortcut F6 is pressed). Preview mode will also be discussed afterwards, but render mode is conceptually more straight-forward.

Behind the scenes, OpenSCAD uses GNU Bison, which is an open source parser generator for context-free grammars [13], to check OpenSCAD scripts for basic syntactical correctness and to parse the OpenSCAD script language as a context free grammar. While parsing, the OpenSCAD script is turned into an abstract syntax tree (AST), in the form of a tree of C++ Objects.

If the source code was parsed successfully, a `SourceFile` object is returned, which represents

the root node of the AST. Next, `instantiate` will be called on this object in order to generate a tree of CSG operations from the AST. All nodes of this tree inherit from the `Node` base class, with their types denoting the different operations and geometric primitives.

The CSG Tree is not quite a 1:1 Mapping of OpenSCAD operations to CSG node types. For example, `scale()`, `rotate()`, `translate()` and `mirror()` all get turned into three dimensional matrix transformations to be applied to the geometry, and thus are turned into the same object type: `TransformNode`. Additionally, calls to user-defined modules are replaced with a `GroupNode`, each containing a copy of the entire subtree within the module. This repetition does not have a significant performance penalty due to the use of caching preventing identical subtrees from having to be recalculated.

Nevertheless, the CSG tree can essentially be seen as the tree of operations built up in the script file, with each operation type for the most part also mapping to its own node type. Note however, that the CSG tree does not directly store any low-level geometry data yet, merely the CSG operations and their parameters, i.e. the data necessary to construct such a geometry. Incidentally, a representation of this CSG tree can also be viewed within the OpenSCAD interface by selecting "Design" $\rightarrow$ "Display CSG Tree...".

As already mentioned, each CSG operation and primitive is represented by a class inheriting from the `Node` base class. In addition to this class, each node type has a free-standing function which, given the relevant AST context, constructs its corresponding polymorphic node object and sets its attributes, essentially acting as a stand-in for the constructor. A free-standing function is used because it can be called from a function pointer. Besides constructing the given node, this function also makes a call to recursively instantiate all its children before returning. Each node type registers itself inside `Builtins::initialize()`, providing syntax hints for the code editor, and the aforementioned function pointer to create a new node of its type based on the relevant data from the AST.

The steps up until this point are necessary for both Preview and Rendering. The next step is where the different modes diverge, as Preview mode does not actually compute a complete Geometry from the CSG Operations, which is what will happen next for the Render mode.

Rendering involves once again traversing a tree which was just built, this time the CSG tree. The final result of this traversal will be a polymorphic `Geometry` object (of which there are a few different types). This all happens inside the `GeometryEvaluator` class. More specifically, its `EvaluateGeometry()` method, when given the root CSG node, recursively traverses the CSG tree using the visitor pattern, going through each node and generating geometries based on their respective operations from the bottom up, culminating in one final geometry once the root node is reached again. This geometry can be e.g. a polyhedron or a set of polygons (depending on the operations and geometries used to produce it). The `GeometryEvaluator` class is the heart of OpenSCAD, it is here where all CSG operations are actually processed to form new geometries (though the actual calculations to do so mostly happen in functions in the `CGALUtils` and `ClipperUtils` namespaces, which in turn delegate to the respective libraries they are named after).

Depending on the type of CSG node, different steps are taken to produce a geometry. A `LeafNode` is a node representing a geometric primitive, and thus has an inherent geometry, rather than a geometry derived from fusing existing geometries in some way. For example,

the OpenSCAD operations `cube([1,1,1])` or `sphere(r = 5)` result in a `CubeNode` and a `SphereNode` respectively, which both inherit from `LeafNode`. `LeafNode` objects provide their own geometry, so when visiting such a node, the `GeometryEvaluator` will simply call the `createGeometry()` method on this node. These nodes also do not support having child nodes.

For CSG operations which work on existing geometries, like `union()` or `difference()`, all child geometries are first collected, and the CGAL library is then used to calculate the resulting geometry (for two-dimensional operations, the Clipper library is used instead). Because of the recursive nature of the evaluation, the geometries corresponding to child nodes will always have been resolved already at this point.

Once the tree has been traversed, we are left with a single `Geometry` object representing the final calculated geometry. If the lazy-union experimental feature is enabled, this can also be a `GeometryList`, which is a subtype of `Geometry` which contains a list of multiple geometries. But while `GeometryEvaluator::evaluateGeometry()` may return such a `GeometryList`, it is not currently supported as an input by any of the node-processing algorithms inside the `GeometryEvaluator` class. It exists solely to skip the step of performing a union on all geometries within the OpenSCAD script root (and a few other situations), to save calculation time if the experimental lazy-union feature is enabled [14].

This `Geometry` object can then be rendered or exported. Each of the OpenSCAD supported file formats also have algorithms to take the various resulting Geometry types and translate them into the supported file formats. Since the `GeometryList` class exists to skip the implicit union at the root of the CSG tree, export algorithms do support exporting of `GeometryList` objects already.

The following geometry types exist:

- **Polygon2d**: For two-dimensional geometries. Contains one or multiple closed polygons.
- **PolySet**: Stores polygon meshes, usually three-dimensional, but also supports two-dimensional geometries. This is also used for rendering two-dimensional geometries in preview mode.
- **CGAL_Nef_polyhedron**: Container for a three-dimensional CGAL Nef polyhedron object.
- **CGALHybridPolyhedron**: Container for a three-dimensional CGAL hybrid mesh or hybrid Nef polyhedron object (used for the experimental fast-csg feature, for faster calculation of composite geometries).
- **GeometryList**: A list of other objects, potentially including more GeometryList objects.

In preview mode, the OpenCSG Library is used instead, and geometries are only partially constructed from the CSG tree, as some operations like intersections and differences are directly applied during the rendering step, rather than first generating a complete geometry (as a polyhedron) and then simply rendering it. However, operations which cannot be processed this way (like e.g. `minkowski()`) result in geometry generation for the entire subtree within

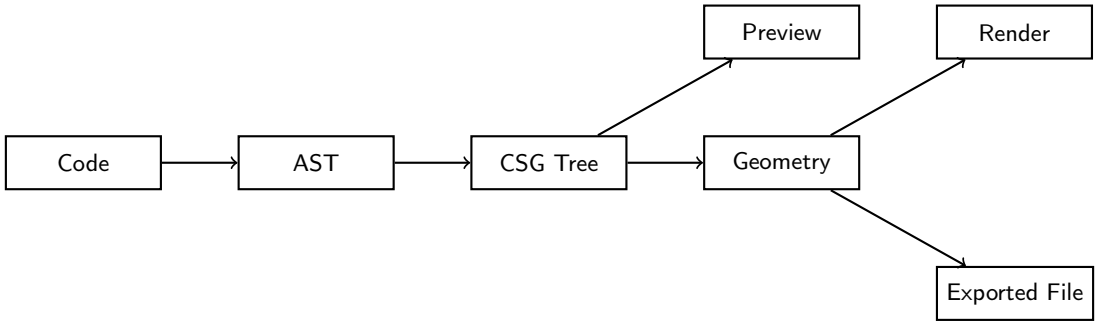


Figure 2.3: A simplified representation of the OpenSCAD pipeline

that operation. This is done by calling `GeometryEvaluator::evaluateGeometry()` on that subtree.

There is also a third renderer, the "ThrownTogether" Renderer. This works similar to preview mode, but subtracted geometries are rendered as regular geometries instead.

Lastly, it's worth noting that both CSG subtrees and geometries have a cache, so that if part of the CSG tree is changed, other parts do not have to be built from scratch, nor do their geometries. Additionally, identical subtrees do not have to be processed multiple times (which will also happen e.g. with repeated use of user-defined modules).

3 Approach

In this chapter, the approach taken to implement multi-material support in OpenSCAD is described. The semantics chosen for the newly introduced part and material operations, as well as the changed semantics of the existing color operation are discussed as well.

3.1 The color(), part() and material() Operations

As it stands, OpenSCAD will only generate geometries as one homogeneous mass, and the process for generating a final geometry pays no regard to attributes of geometries or segmentation of openscad scripts into different geometries, which are intended to either be printed separately or from different materials. In order to give OpenSCAD the ability to be able to handle these use cases, we need to first extend the OpenSCAD script language to be able to express our intentions to give attributes to geometries, which should also be preserved as these geometries are combined with others using CSG operations. We want to be able to segregate generated geometries by several attributes mirroring those of the desired printed objects: Different colors, materials and belonging to different parts.

As previously mentioned, a `color()` operation already exists in OpenSCAD, but bears no effect on the generated geometry, and only affects rendering in preview mode (and the "Thrown Together" renderer). Additionally, in preview mode, combined geometries of different colors are not truly merged, but their individual parts are rendered separately (in their respective colors, with z-fighting in case of overlap). The semantics of this operation now have to be changed to serve use cases concerning physical objects, and the operation should now define the color attribute of the geometry it is applied to. This attribute should also carry over to the final render and export, not just preview mode. If two objects are combined, the color of the first object within the combining operation should be used to determine the color.

Additionally, a `part()` and `material()` operation are also introduced, to be able to assign materials to geometries (in the form of an arbitrary string supplied by the author of the OpenSCAD script), and denote a part which a geometry belongs to, to allow modeling of multiple parts in a single OpenSCAD script. These operations mirror the geometry metadata supported by the 3MF and AMF file format. The semantics for combining geometries which differ in either of these two attributes are chosen to be different from those of the color operation.

3.2 Reconcilable and Irreconcilable Attributes

Alongside the concept of geometry attributes, the idea of reconcilable and irreconcilable attributes is now introduced. The part and material of a geometry are defined as irreconcilable

attributes, while the color attribute is defined as reconcilable. If any irreconcilable attribute differs among two geometries, they cannot be merged together. If, for example, a `union()` operation is performed on several geometries, but not all have the same irreconcilable attributes, they are first grouped by their irreconcilable attributes (i.e. their part and material values), and the union is then performed separately for each group. Thus, the result of this operation is not necessarily just one geometry anymore, but potentially multiple geometries of different attributes.

If two irreconcilable geometries need to be combined, their attributes have to first be overwritten to match by using the `material()` and/or `part()` operation on them.

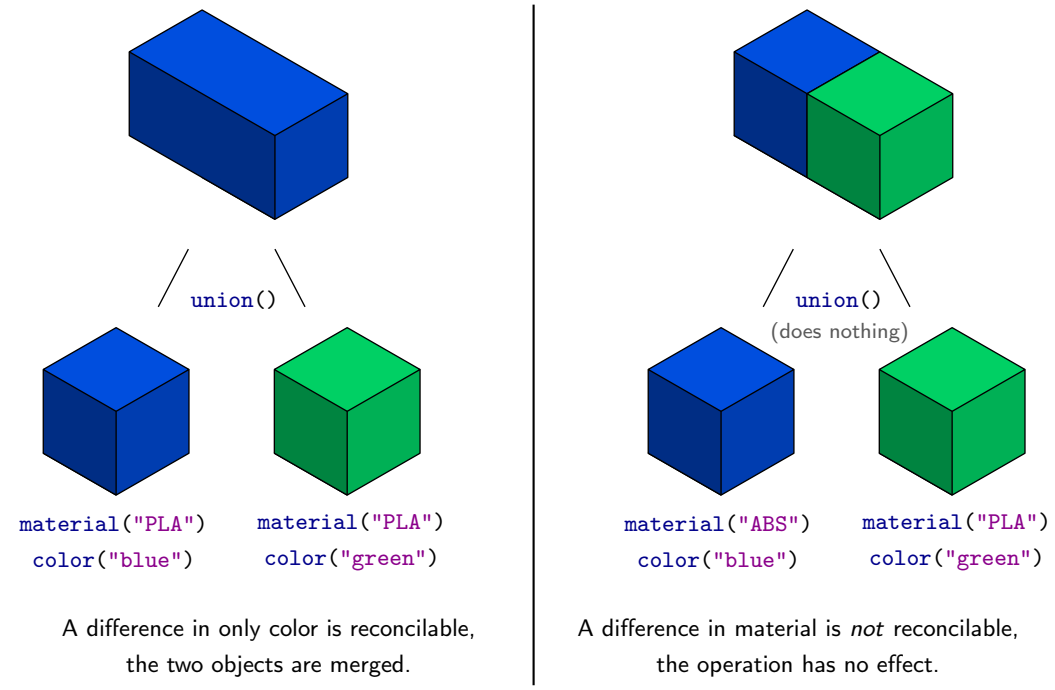


Figure 3.1: A reconcilable and an irreconcilable `union()` operation

Since the color attribute is defined as reconcilable, if an operation is performed on geometries of different colors (but matching irreconcilable attributes of part and material), they can be fused together into a new geometry. The resulting geometry will have the color attribute of the first listed geometry within the operation.

These semantics do not apply just to `union()` but any kind of operation, like e.g. `hull()` or `minkowski()`. For any operation, its child geometries are first grouped by their irreconcilable attributes and the operation is then performed once for each group, resulting in up to as many separate geometries as there were groups of unique irreconcilable attributes.

Unlike colors, part and material attributes have no visual effect on the rendered geometry. This is why it makes sense to be stricter about mixing and matching these attributes in operations which fuse geometries. This makes user errors easier to spot, rather than leaving

the end user wondering why material or part attributes are not as expected in the final export, when they were simply overwritten due to an accidental operation on non-matching geometries. Preventing these kind of geometries from merging in the first place makes this type of error apparent the moment it occurs, making it easier to resolve.

An additional advantage is that irreconcilability semantics allow us to apply operations selectively to geometries which were imported from an external file with the `import()` operation. If we have a 3MF file, and wish to modify a geometry within it, we can do this without affecting other geometries also contained in the file, as long as they are separated by part or material attributes. Imagine a hypothetical 3MF file containing two objects, one called "part1" and one called "part2". We either we do not have the OpenSCAD script file used to generate this 3MF file, or it was generated using another CAD tool. If we wish to merge an additional geometry onto this object, we can do so using irreconcilability to our advantage:

```
union() {
  import("hypothetical_file.3mf");
  part("part2") cube([5,5,5]);
}
```

Listing 3.1: Selectively modifying imported geometries

Another important consideration is that invocation of a module performs an implicit `union()` operation on its contents. Simply removing this implicit union would break backwards compatibility with existing OpenSCAD scripts. Take for example this code:

```
module two_objects() {
  cube([1,1,2]);
  cube([2,1,1]);
}
intersection() {
  two_objects();
  cube([1,1,1.5]);
}
```

Listing 3.2: An example why the implicit union performed by module definitions is useful

If we were to remove the implicit union from module invocation and have it return both cubes as separate geometries, this code would perform an `intersection()` operation on all three cubes. Therefore this `intersection()` operation would return only the geometry which all three cubes share in common, rather than that which the third cube has in common with either `cube([1,1,2]);` or `cube([2,1,1]);`, thus producing a different result.

This illustrates how removing the implicit `union()` operation would break backwards compatibility with scripts written under the assumption that modules do perform such an implicit union. Additionally, this code also serves as an example of how making this change would lead to counterintuitive behavior, since the module is listed within the `intersection()` operation as if it was one object and is expected to behave as such.

The irreconcilability semantics of material and part attributes allows us to keep the current behavior of modules while allowing them to contain multiple parts and geometries of different

materials. We also define special semantics for this implicit union operation, to allow preservation of different colors on otherwise reconcilable geometries. Color attributes are defined to be treated as irreconcilable specifically for the purpose of this implicit `union()` operation, to allow modules to contain geometries of different colors. However, in subsequent operations performed on a module invocation, the color attribute is treated as reconcilable again. This is discussed more in-depth in section 3.3.

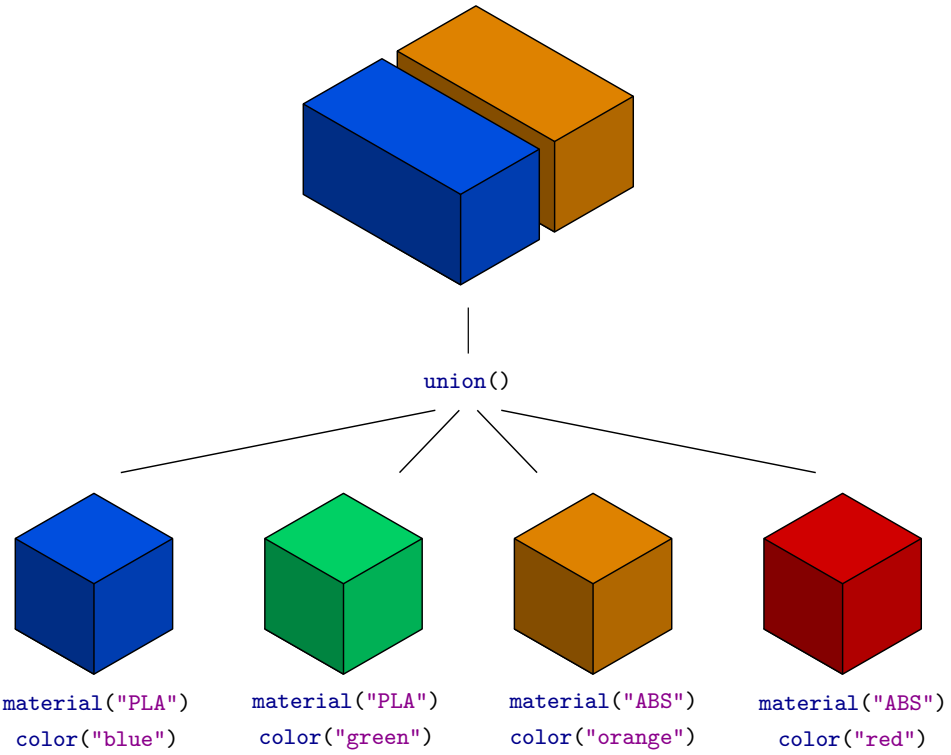


Figure 3.2: A `union()` operation encompassing multiple groups of reconcilable objects

Additionally, it should be noted that allowing geometries of different colors to merge ensures that backwards compatibility to old OpenSCAD scripts is retained, where the `color()` operation might have been used for organizational purposes. Making `color()` operations irreconcilable after the fact could otherwise alter the generated geometries of existing scripts in cases where geometries of different colors are combined.

The semantics chosen do not mirror those previously discussed for this feature by the OpenSCAD developer community on the issue tracker. The proposed changes on the issue tracker suggested that single geometries could have regions of different colors (similar to the behavior of preview mode), so that e.g. applying a `union()` operation on a red and a blue rectangle would create a single geometry with two segments of different colors [5]. However, with such semantics, it becomes much harder to imagine sensible semantics for

almost any other operation. For example, using the `hull()` operation on two geometries will fill empty space in between those geometries, and it's unclear what color that space should inhabit. It seems that in all but the most simple cases, this behavior would greatly increase the complexity of implementation while its results are of limited usefulness. Even in the case of a union operation, it's ambiguous what color a part of the geometry where multiple differently colored geometries intersect. The fallback here would probably end up being that in cases of ambiguity, the color whichever geometry comes first is chosen. So it makes sense to just drop the concept of multi-color geometries entirely and enforce this rule across the board, making the behavior more predictable and explicit.

3.3 Implicit "Soft Union" for Modules and the Script Root

As already mentioned, definition of a module performs an implicit union on the objects within that module. While this doesn't result in any problems regarding part and material definitions within modules due to irreconcilability semantics, it does mean that modules containing parts of different colors would get merged into a single geometry of the same color (as long as part and material attributes are also the same). Take for example the following code:

```
module different_colors() {
  color("red") translate([0,0,10]) cube([10,10,10]);
  color("green") cube([20,10,10]);
  color("green") cube([10,20,10]);
}
different_colors();
```

Listing 3.3: A module containing different colored geometries

Without any change to module semantics, it would result in the geometry visible in fig. 3.3. The only way to prevent the different colors within the module from being merged together would be to give them different part or material attributes. However, doing so would have an impact on subsequent operations performed on the module invocation, unless the attributes are first overwritten again (which after invocation, can also only be done for the entire module, which might be undesirable).

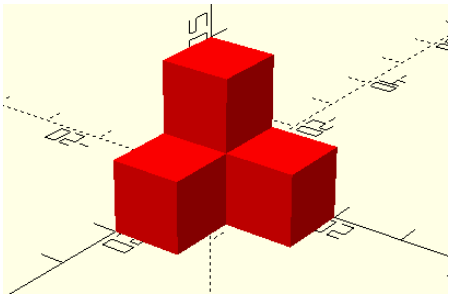


Figure 3.3: The resulting geometry, before introducing module "soft union" semantics

As previously mentioned, completely dropping the implicit union performed by modules similarly results in backwards compatibility breakage, and will cause unintuitive behaviour with things like e.g. the `intersection()` or `minkowski()` operation.

Instead, as a compromise solution, we perform a "soft union": We treat all attributes, including color, as irreconcilable, and union only geometries which have exact matching attributes. Thus, the result of listing 3.3 is now two geometries, with the two separate green parts merged into one, but not combined with the red cube, as can be seen in fig. 3.4.

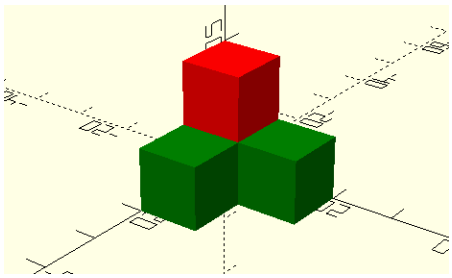


Figure 3.4: The resulting geometry, after introducing module "soft union" semantics

Note that this unfortunately still represents a break in backwards compatibility. Since the `color()` operation already existed before the work of this thesis, modules may have been defined containing different colors. If such modules are used within the previously mentioned operations, the results would differ from those of an unmodified OpenSCAD version. However, it's a smaller break in backwards compatibility, which should affect fewer files, and unlike abandoning the implicit union completely, can be solved by simply deleting the `color()` operations from the preexisting OpenSCAD scripts.

```
module two_reconcilable_geometries() {
  color("red") cube([10,10,20]);
  color("blue") cube([20,10,10]);
}

intersection() {
  two_reconcilable_geometries();
  cube([15,10,15]);
}
```

Listing 3.4: An example of an OpenSCAD script which would result in a different geometry because of this change

It seems like this is a better middle ground than having modules merge different colors together, which would be rather cumbersome to deal with.

The script root also performs an implicit union operation, unless the experimental lazy union feature is enabled. This too is changed to be a "soft union" instead. It could also have been removed entirely (which is what the lazy-union feature does), but this has an effect on exported files, resulting in separate geometries at the root being exported as separate objects. This might be useful in some cases, but since the introduction of the `part()` operation allows

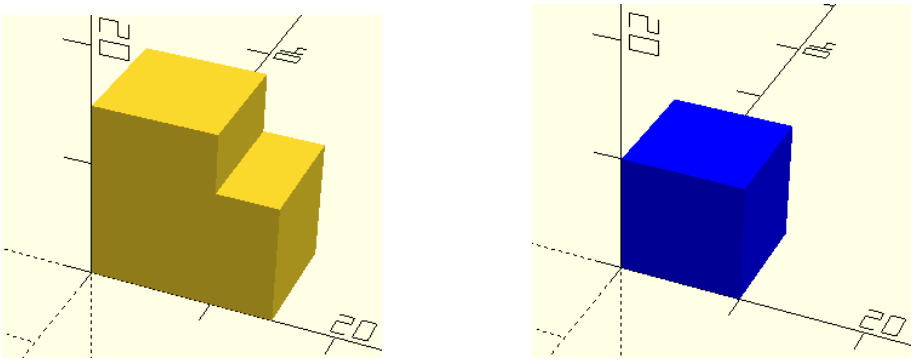


Figure 3.5: The resulting geometry of listing 3.4. The differently colored cubes of the module get treated as separate inputs to the `intersection()` operation now.

us to explicitly specify separate geometries, not combining geometries of matching attributes is arguably more of a hindrance than helpful.

It's worth noting that within the code, the implicit union operation of modules is owed to the fact that they get translated into a `group() {...}` node containing the module contents (this can also be seen when looking at the CSG tree inside OpenSCAD). The change of module semantics is thus implemented by changing the semantics for the `group()` operation. This means that the `group()` operation (which previously would behave identically to the `union()` operation) could now also be used inside OpenSCAD scripts directly to perform a union without merging different colors together.

3.4 Special difference() Semantics

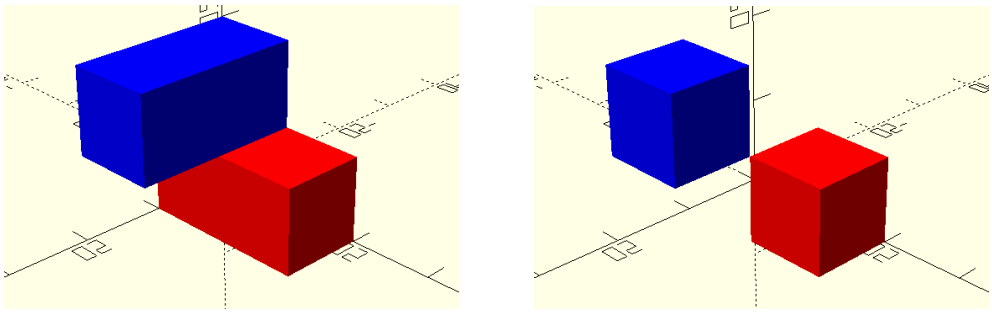
For the `difference()` operation, special semantics are also defined. Since a `difference()` operation always involves removing mass from a positive geometry, its resulting attributes are always unambiguous. Thus, it does not make sense to pay heed to part and material attributes for the geometries which are subtracted from the first listed geometry in a `difference()` operation. A subtracted geometry represents a void and thus cannot meaningfully have attributes to begin with. Additionally, if two separate parts are being modeled, it might make sense to subtract one from the other, to create a matching inverse shape (e.g. a screw and a thread). These special semantics allow you to do this without having to first overwrite the geometry attributes of the subtracted object to match.

Additionally, if the first geometry within a `difference()`-operation is a `GeometryList`, rather than unpacking that list and treating each geometry contained within it as if it was a separate child of the operation (which is how `GeometryList` objects are handled for other operations), the `difference()`-operation is performed once for each item in the list, with that item representing the positive shape, from which all other geometries are subtracted.

```
module two_parts() {
  color("red") cube([10,20,10]);
  part("other_part") color("blue") translate([0,0,10]) cube([20,10,10]);
}

difference() {
  two_parts();
  cube([10,10,20]);
}
```

Listing 3.5: An example of a difference operation with multiple components. The cube gets subtracted from each individual geometry.

This example would yield the model which can be seen in fig. 3.6. The cube inside the `difference()` operation is subtracted once from each geometry contained within `two_parts()`, regardless of geometry attributes.



Geometry of just `two_parts();`

Geometry of listing 3.5,
after the difference operation

Figure 3.6: The result of a difference operation with multiple components.

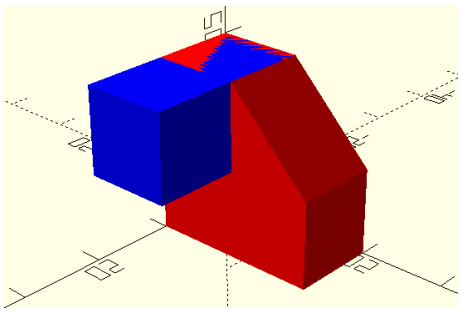
It's also worth noting that if a `GeometryList` contains multiple reconcilable geometries because of the special module semantics talked about in the previous section, a `difference()` operation will not combine them. So in this particular case, if we adjust the code to remove the `part("other_part")` operation and make the two geometries within the module reconcilable, it would still have the same visual result we can see in fig. 3.6.

```
module two_parts_reconcilable() {
  color("red") cube([10,20,10]);
  color("blue") translate([0,0,10]) cube([20,10,10]);
}

difference() {
  two_parts_reconcilable();
  cube([10,10,20]);
```

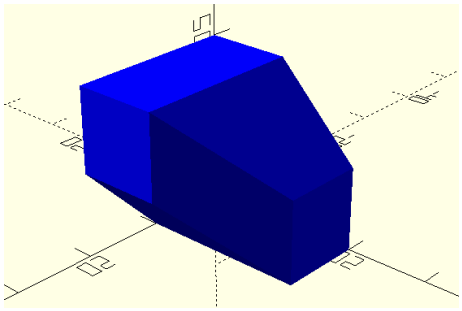
```
}
```

Listing 3.6: The code from listing 3.5 with the part attribute removed. The `difference()` operation does not combine reconcilable geometries.

This is owed to the fact that the `difference()` operation does not need to resolve irreconcilable attributes, and is not true of other operations. If we replace the `difference()` operation with a `hull()` operation, we get the results seen in fig. 3.7. We can also see z-fighting on the results of listing 3.5 due to overlapping geometries.



listing 3.5 with a `hull()` operation instead of difference



listing 3.6 with a `hull()` operation instead of difference

Figure 3.7: Performing operations besides `difference()` on a module with multiple reconcilable geometries will merge them with each other.

3.5 Other Possible Approaches not Taken

In this section, some alternate designs for implementing multi-material support in OpenSCAD are discussed.

3.5.1 Only Allow Attribute Declaration at the Top-Level of the Script

The initial idea for this thesis was to only allow annotations of different segments of geometry at the topmost level like a namespace, looking something like this:

```
geometry("PLA", "red"):
  cube([1,1,1]);

geometry("ABS", "green"):
  translate([1,0,0]) {
    union() {
      cube([1,1,1]);
      sphere(r = 0.7);
    }
```

```
10  }
```

Listing 3.7: A hypothetical syntax for only allowing attribute declaration at the top level of the script

Here, each segment represents one separate part of the multi-material object. In terms of expansion of the OpenSCAD script language, this would have been slightly more complex (requiring modifying the context-free grammar to only allow geometry operations at the top level). The actual implementation of the geometry-related logic would have been much simpler however. Due to the separation of geometry calculation and attribute assignment, it would avoid various previously discussed and upcoming edge cases entirely. This means that there is also never any ambiguity to what an operation should do to begin with, making this approach very intuitive. However, that is owed to the fact that the functionality here is a subset of that which was implemented in the end, and thus allows for much less flexibility.

For example, in this implementation, geometries within user defined modules could not be given material, part or color annotations directly, and this would have to be specified where the module is called instead. This also means that each module can only contain one type of geometry, which prevents some use cases like e.g. defining modules to codify a physical offset between two different parts.

Another problem with this syntax is that it's not possible to re-import previously exported geometries with the `import()` operation and to then further modify them in a way that retains their attributes. Since attributes can only be defined at the top level, all objects imported from an AMF or 3mf file would be within the same assignment, and be turned into one object.

Additionally, since the color operation already exists, having two ways to define colors, only one of which affects the final geometry would be an added source of confusion.

The main advantage of this solution is its simplicity, both in implementation and lack of edge cases. However, for the end user, edge cases can be avoided by simply laying out OpenSCAD scripts in a similar way, while still offering the added flexibility of defining attributes within modules or other places if need be.

3.5.2 Make All Geometry Attributes Irreconcilable

The most obvious alternate approach would be to either make all geometry attributes irreconcilable, or to make them all reconcilable, rather than defining different semantics for different attributes. The reasoning for why this was not done has already been briefly stated already, but let's take a closer look at the implications of this design decision.

The case for not making all attributes irreconcilable is rather obvious, it would break backward compatibility with existing OpenSCAD scripts. Worse more, it would break backward compatibility completely silently without resulting in any error messages, but instead by producing unintended alternate geometries if existing scripts which use colors are loaded. Depending on where the color operation was used, this might happen in places where it might not be immediately obvious.

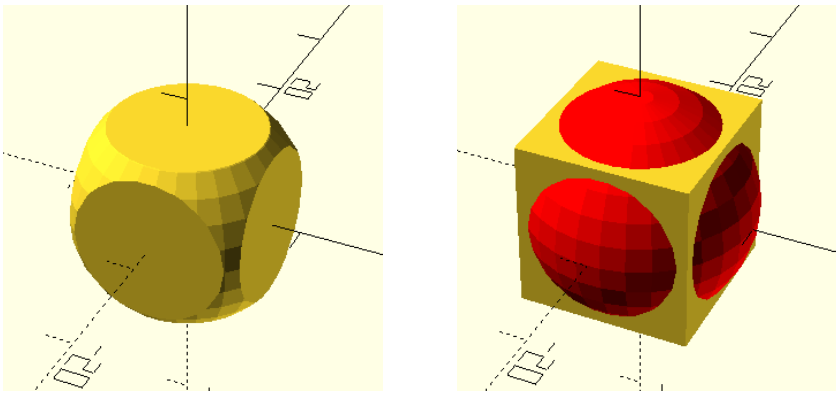
To drive this point home, let's look at a specific example of what this might look like:

```
1 intersection() {
```

```
2   color("red") sphere(10);
3   cube(15, center = true);
4 }
```

Listing 3.8: A script that would result in a different geometry if the color attribute were irreconcilable

The results of this script in current versions of OpenSCAD and in a hypothetical version with colors as irreconcilable attributes can be seen in fig. 3.8. Stray `color()` operations within the script fundamentally change the final geometry produced, when they were only intended to be a visual aid during the creation of the file under old semantics.



Current versions of OpenSCAD

With colors as irreconcilable attributes

Figure 3.8: A comparison of the geometry produced by listing 3.8 without and and with colors as irreconcilable attributes

Of course, it could be argued that the current implementation is also backwards incompatible, in that geometries which previously did not have their color attributes preserved will now do so, resulting in OpenSCAD scripts which previously created a single colorless geometry now resulting in a geometry with color attributes, or potentially even multiple geometries with different color attributes. And indeed, in the version developed as part of this thesis, the final geometry is also not identical to the one in current versions of OpenSCAD, it is red.

However, the geometric shape of the resulting object will be exactly as it was in old versions, and while the resulting exported files may have metadata that segments the geometry into different colors, this can simply be configured to be ignored and be printed as one homogeneous mass in the slicer software. Additionally, if a file format is used that does not support geometry attributes (like .stl, for example), this becomes a complete non-issue, as the geometry is exported identically to how it was previously.

It has to be pointed out that the special "soft union" semantics for modules which were defined in section 3.3 do already break backwards compatibility in ways which might produce wrong geometries in the exact same ways making colors irreconcilable might. Under the semantics defined in this thesis, there unfortunately seems to be no way to have modules be able

to produce geometries with the same part and material attribute but different colors without also breaking backwards compatibility. Giving modules the ability to preserve geometries differing only in color was chosen as the lesser of two evils, but it's not clear whether this is truly the better option.

Outside of module invocations, it's already possible to define geometries which differ only in color, since the implicit union operation at the root of the script file was also changed to be a "soft union" instead. Since breaking backwards compatibility in this instance would offer no additional benefit, the choice to make colors reconcilable was made instead.

3.5.3 Make All Geometry Attributes Reconcilable

The obvious rebuttal to the previous section that comes to mind is "if we can't make all geometry attributes irreconcilable, why not just make them all reconcilable?" Simply allow any operation of geometries with dissimilar attributes, and have the resulting geometry take on the attributes of the first listed geometry in that operation. And indeed this would work, and be fully backwards compatible with existing scripts. It would also greatly simplify implementation, as the assumption that any operation only ever results in one single geometry would hold true again.

However, it seems ill-advised to make it easy to accidentally overwrite the material (or part) of a geometry in a way that is visually completely opaque to the end user (since unlike colors, there will be no visual change between materials), and thus might only be realized once the final model is being sliced, or worse, printed. Furthermore, since this kind of code-error would be completely opaque within OpenSCAD, it would be hard to find the offending line of code causing an accidental overwriting of material, especially since there might be a lot of time between causing the error and spotting the error once the file is imported into a slicer program.

By making the non-visible geometry attributes irreconcilable, the material properties will either be preserved as expected, or the user will be able to spot the error right away when a combining operation does not actually combine the given materials in the expected way. That being said, there are cases where irreconcilability semantics may also silently do something other than what the user intended, e.g. when calling `union()` on two irreconcilable geometries, it's not obvious whether the action was performed or not if the geometries do not also have dissimilar colors, since a `union()` operation otherwise has no effect on visible appearance by itself.

Furthermore, the issue of modules also still needs to be dealt with regardless, if all attributes are made reconcilable. In order to truly not break backwards compatibility, the "soft union" performed by modules would also need to be disabled, as otherwise the issues already covered in the previous sections will once again crop up. If modules get no special treatment, this also means that we can no longer define modules containing geometries of different attributes.

Imported files containing geometries with different attributes would also once again be difficult to work with without ending up merging them all into one geometry.

The solution of making all attributes reconcilable is functionally very similar to only allowing top-level attribute declarations. It is arguably the worst of both worlds, allowing accidental overwriting of attributes without offering any real additional benefits.

3.6 Initial Implementation

In this section, the specific implementation of the changes described in the previous sections is discussed.

First, new CSG node classes are created for the part and material operation. These classes are quite simple, as they only have two purposes: First, to validate the arguments for their operation (which, as previously mentioned, technically happens in a standalone function defined alongside the class, but for all intents and purposes this acts as a constructor which can also be called via a function pointer), and secondly to act as a data structure to store those arguments in a standardized way. For example, inside the `color()` operation, it is possible to specify colors as either a triplet of numbers representing RGB values, or strings containing web color names like "aquamarine". Both variations will be translated into a `Color4f` member object of `ColorNode` containing 4 float values, the fourth being alpha transparency. Both `PartNode` and `MaterialNode` simply need to store an identifier string.

In order to be able to initialize leaf geometries with the correct attributes when they are created, the pseudo constructor function for `ColorNode`, `PartNode` and `MaterialNode` is extended to also recursively apply their attributes to all child nodes.

First, two structs are defined to store these attributes. Since attributes apply to geometries, they are defined inside the `Geometry` class, as follows:

```
1  struct Attributes {
2    std::string materialName = "";
3    std::string partName = "";
4    Color4f color = {-1.0f, -1.0f, -1.0f, 1.0f};
5
6    bool operator==(const Geometry::Attributes &o) const {
7      return materialName == o.materialName && partName == o.partName && color ==
       o.color;
8    }
9
10   bool operator!=(const Geometry::Attributes &o) const {
11     return materialName != o.materialName || partName != o.partName || color !=
       o.color;
12   }
13
14   bool operator<(const Geometry::Attributes &o)  const {
15     return materialName < o.materialName
16     || (materialName == o.materialName && partName < o.partName)
17     || (materialName == o.materialName && partName == o.partName && color < o.color);
18   }
19 };
20
21 struct IrreconcilableAttributes {
22   std::string materialName = "";
23   std::string partName = "";
24
25   //operators omitted for brevity, as they function analogous to Geometry::Attributes
```

```
};
```

Listing 3.9: Geometry attribute structs, with comparison operators to allow usage as a key in a `std::map`

Next, the node base class is extended with a new member variable called `derivedAttributes`, to store these attributes for every node. These attributes are recursively applied to all nodes in the CSG tree, not just nodes that are an attribute setting operation.

```
Geometry::Attributes derivedAttributes = {};

Geometry::Attributes getGeometryAttributes() const {
  return derivedAttributes;
}

Geometry::IrreconcilableAttributes getIrreconcilableGeometryAttributes() const {
  return {
    .materialName = derivedAttributes.materialName,
    .partName = derivedAttributes.partName
  };
}
```

Listing 3.10: Additions to the Node class

In order to populate this new member variable of the `Node` class, whenever a `ColorNode`, `PartNode` or `MaterialNode` is created via the pseudo-constructor function `builtin_color()`, `builtin_material()` or `builtin_part()`, the appropriate attribute is applied to that node, and also to all its children, recursively. Luckily, these pseudo-constructor functions are already in charge of instantiating all their child nodes, so every child node simply needs to be recursively marked after doing this.

Since each attribute node instantiates its children (like all nodes), and only then marks them with their attribute, this leaves us with correct attributes even in cases where attributes are overwritten, like this:

```
material("PLA") {
  material("ABS") {
    cube([1, 1, 1]);
  }
}
```

Listing 3.11: An example of nested attribute declarations

In this situation, the first node to be instantiated would be `material("PLA")`, which would then instantiate its child node, `material("ABS")`, before assigning any attributes. During its instantiation, `material("ABS")` also instantiates its children, and then recursively marks them with the material attribute "ABS". In this case the only child is one `CubeNode`, which is now marked with a derived material of "ABS". After this, all children of `material("PLA")` have been instantiated, so it now starts marking its children with its own material attribute value, overwriting the previous derived attribute of "ABS" with "PLA" `CubeNode` and the child `MaterialNode`.

All nodes now have their correct respective `derivedAttributes`. Thus, whenever a LeafNode creates a geometry, it can simply pass its attributes to that geometry. The constructors of the various polymorphic `Geometry` classes are extended by another argument, the geometry attributes.

Inside `GeometryEvaluator`, the CSG tree is traversed, and its geometries are generated, either from `LeafNode` objects or through combination operations on existing geometries. Each of these operations now needs to be made to support a `GeometryList` as an input, which they currently do not. The code currently assumes it would never encounter a `GeometryList` at this point. Additionally, each operation needs to be performed once for every group of unique irreconcilable attributes of the geometries within.

Save for a few exceptions, most operations currently start by gathering the geometries generated by their child nodes, using the `GeometryEvaluator::collectChildren3D()` or `GeometryEvaluator::collectChildren2D()` methods, which each return a list of nodes and respective geometries, contained within a `std::pair`. A new method which groups geometries by irreconcilable attributes called `collectReconcilableChildGroups()` is created as a replacement. It returns a map of `Geometry::IrreconcilableAttributes`, and one of these previously mentioned list of pairs for each unique attribute combination. This method also unpacks any `GeometryList` which it comes across, and sorts its children into the same map of lists, thereby automatically also allowing operations which use this method to support `GeometryList` objects as inputs.

Now, each operation method which previously used one of the collectChildren methods is modified to use this new method, and perform its operation once for every unique set of irreconcilable attributes. The resulting geometries of each such operation are given the attributes of the first Geometry in the list, thereby also retaining their irreconcilable attributes. If there are multiple resulting geometries, they are returned as a `GeometryList`.

Some operations do not use either of the collectChildren methods, they must also be made to account for `GeometryList` objects as inputs. Additionally, the `GeometryList` class needs to support `transform()` and `resize()` operations which all other polymorphic Geometry objects support, as they are currently just stubs. This involves simply performing a `transform/resize` on each member of the `GeometryList`.

Further, two-dimensional operations in the codebase assume that a two-dimensional geometry is always a `Polygon2d`, which need to be made to account for a `GeometryList` containing two-dimensional geometries, instead of simply assuming a cast to `Polygon2d` will always succeed if the `getDimension()` method of the geometry returns two.

Having made these changes, geometries with attributes can now be generated, and the correct semantics will be applied. Next, the `CGALRenderer` class, which is in charge of rendering fully calculated geometries, has to be extended to correctly color geometries based on their attributes. This is different from preview mode, which has its own renderer class.

The `OpenCSGRenderer` class used by the preview mode needs to be changed to support rendering GeometryList objects (the CGALRenderer already supports them), and to apply color based on geometry attributes, so that combined geometries are colored correctly. These changes alone are insufficient for correctly rendering geometries while factoring in irreconcilability semantics. Unfortunately the work on preview mode was not finished in time, which is talked

about more in-depth in section 3.7.3.

Finally, exported files need to retain the material, color and part information. For both the AMF and 3MF format, a file can be split up into multiple objects, which can each be given a name. The 3MF format supports this via a dedicated attribute [15] and the AMF format via an optional `<metadata>` tag [16]. The identifier string passed to the `part()` operation is chosen as the name of an exported object. It should be noted however, that if there are multiple top-level geometries which have the same part attribute but differ in other attributes (including color because of the "soft union" performed at the root level), they are exported as multiple objects which share the same name. This allows them to be individually selected and configured when imported into slicer software. Material and color attributes are also written into exported AMF and 3MF files, but neither PrusaSlicer nor Cura seem to support these attributes at this point in time, and only respect the existence of separate objects within these files, but not their respective colors or materials. Therefore, materials have to be configured manually for each object, and objects are not displayed in their specified colors.

AMF is an XML-based format. Exporting AMF-files in OpenSCAD happens via an ad hoc implementation, which simply writes the required XML directly into a `std::ostream` as it parses each geometry. In order to assign material properties to a geometry, the material has to first be defined using a `<material>` tag. It's worth noting that materials within the AMF-format consist of a material name and a color, so each combination of material and color needs one material definition inside the file. This definition is then given an id, which is referenced when the geometry data is specified. We simply build a map of material-color-combinations as we come across them, assign each a unique material ID, and write their tags into the file. Objects volumes are then assigned the appropriate material IDs.

3MF files on the other hand are generated using lib3mf. The 3MF-format is also an XML-based format (albeit a compressed one, consisting of multiple files), which makes it easy to verify if changes were implemented correctly. OpenSCAD supports both lib3mf version one and two, of which version one seems to still be more commonly in use. For example, Ubuntu 22.10, Debian Sid and Arch Linux still use lib3mf version one as of this writing [17] [18] [19]. Fedora Linux, on the other hand, uses version two [20]. Unfortunately, lib3mf version one is relatively poorly documented. OpenSCAD also uses the C API rather than the C++ one for version one of the library, which uses void pointers which have been renamed with `typedef` for most types, making inheritance hierarchies opaque. For version two, the C++ API was used instead. Support has been implemented for both versions.

Export of colors and part names as `class` attribute values for Scalable Vector Graphics (SVG) files has also been added, because it was very straightforward to do so.

Import has been made attribute aware for only 3MF-files, utilizing either either major version of the lib3mf library.

3.7 Implementation Problems and Edge Cases

After this initial implementation was complete, several issues with the approach were uncovered during testing, in part because of OpenSCADs excellent test suite.

3.7.1 Null Geometries and Attribute Resolution

In most cases, when operations which combine multiple geometries (like `difference()`, `hull()`, etc.) are performed, all children of that operation are first collected, as a pair of both a node in the CSG tree and an object representing its calculated geometry. Any such geometry can be empty. For example, if you subtract a geometry from itself using the `difference()` operation, the result will be a `Geometry` object which inhabits no space, like e.g. a `PolySet` object with 0 facets. Any geometry can be checked for emptiness using the `isEmpty()` method.

Separately from an empty geometry object, there are also null geometries, which are represented by a null pointer instead of a pointer to a geometry object, i.e. an absence of a `Geometry` object in that aforementioned pair of node and geometry. For example, if a 2d geometry is used in a 3d operation, that geometry is replaced with a null pointer while collecting all the geometries that are affected by this operation (the pointer to its node is kept, however). Performing an operation on nothing (e.g. `hull() {}`) will also result in a null geometry, as will performing most operations on children which contain only empty geometries.

```
difference() {
  cube([1,1,1]);
  cube([1,1,1]);
}
```

Listing 3.12: This script will produce an empty geometry

```
hull() {
  difference() {
    cube([1,1,1]);
    cube([1,1,1]);
  }
}
```

Listing 3.13: This script will produce a null geometry

It is important to not just omit such geometries completely in the list of affected geometries, since the ranking in that list potentially matters to the operation. Most prominently, if a `difference()` operation is performed and the first geometry is a null geometry, if it was omitted entirely, the difference operation would instead subtract from the next geometry in the list, which itself was meant to also be subtracted instead.

However, such null geometries obviously do not have attributes since they are nothing but mere null pointers. Thus, we lack information needed for the resolution of irreconcilable attributes, which are usually stored inside the `Geometry` object.

Since we do have access to the CSG node which produced the null geometry, we can try to retrieve the attributes from the node instead, but this is also problematic. Take for example this code:

```
intersection() {
```

```
  part("support") material("PLA");
  part("support") material("PLA") cube([1,1,1]);
}
```

Listing 3.14: An intersection with a null geometry, which ostensibly has the same irreconcilable attributes as its sibling geometry

At first glance it appears that both nodes inside the intersection() block share the same irreconcilable attributes: a part of "support" and a material of "PLA". However, since the first item within this intersection block is a null geometry, we must look to the node for its attributes.

However, the first node immediately within the intersection operation is the `part("support")` operation, and while the `PartNode` is aware of its own part attribute, it does not inherit from its child node, `material("PLA")`, and thus has no derived material attribute defined, making it irreconcilable with the second geometry. This means that two separate operations are performed for each of the irreconcilable groups, causing the wrong result (the correct result here would be another null geometry, and instead the cube remains).

There appears to be no good way to resolve this problem. One could suggest traversing the node until a leaf node is reached, and then taking the attributes from that node. And indeed, this would yield the intended result in the example in listing 3.14, but since nodes can have multiple children, this is not an unambiguous operation.

```
intersection() {
  part("support") {
    material("PLA");
    material("ABS");
  }
  part("support") material("PLA") cube([1,1,1]);
}
```

Listing 3.15: It is not clear which material attribute is correct for the null geometry

In this example, the first geometry within the `intersection()` operation is still a null geometry, but its intended material is no longer clear.

The examples given here might seem contrived, and in their simplified form, they certainly are. However, this problem can also arise under more natural circumstances, such as performing operations on empty geometries caused by specific configurations of a parametric model.

It's also worth noting that a similar problem has cropped up before and has been discussed on the OpenSCAD issue tracker in issue #666, #3311 and #3312 without the context of reconcilable and irreconcilable geometries. Here, null geometries yielded the wrong results in preview mode, because null geometries were ignored completely [21] [22] [23].

Because there appears to be no consistent way to retrieve null geometry attributes from its corresponding CSG node, the decision was made to treat all null geometries as having no attributes instead, in order for them to at least behave consistently. Additionally, the approach of calculating node attributes while the CSG tree is being built turned out to also have problematic side effects, and was later replaced completely.

3.7.2 Resolving Attributes While Building the CSG Tree Causes Geometry Changes

As previously mentioned, the initial implementation of geometry attributes resolved all attributes while the CSG tree was being built, by applying attributes recursively to all child-nodes in the CSG tree. This meant that a geometry could already be assigned its attributes when it was being created, either directly from the node creating it (if creating a geometry from a `LeafNode`), or from its parent geometries (in the case of being the result of a CSG operation combining existing geometries), and the attributes of geometries would never have to be touched again after their creation.

While this approach does work, it creates unintuitive behaviors in cases where geometry attributes are set, and then later overwritten again further up the CSG tree. Take for example this code:

```
$fn = 100;
module rods() {4
  hull() {
    part("left") sphere(r = 1);
    part("left") translate([10,0,0]) sphere(r = 1);
    part("right") translate([10,10,0]) sphere(r = 1);
    part("right") translate([0,10,0]) sphere(r = 1);
  }
}

rods();
```

Listing 3.16: A module performing a `hull()` operation on geometries with different part attributes

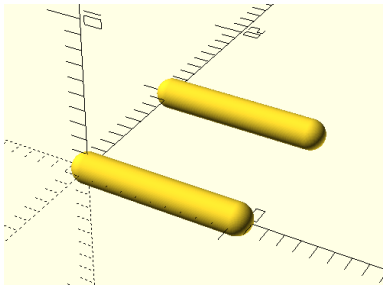


Figure 3.9: The combination of irreconcilable geometries within the hull-operation results in two separate hull-operations being performed

The code in listing 3.16 results in the geometry seen in fig. 3.9. Due to belonging to separate parts and thus being irreconcilable, two `hull()` operations are performed on the two sets of spheres, resulting in two separate geometries, one belonging to the part "left" and one belonging to the part "right".

If we modify this code, to overwrite the part attribute to be "both" after the hull operation

is performed, the resulting geometry is also changed, despite being derived from the exact same code (which furthermore is separated out into a module). The new resulting geometry can be seen in fig. 3.10.

```
$fn = 100;
module rods() {
  hull() {
    part("left") sphere(r = 1);
    part("left") translate([10,0,0]) sphere(r = 1);
    part("right") translate([10,10,0]) sphere(r = 1);
    part("right") translate([0,10,0]) sphere(r = 1);
  }
}

part("both") rods();
```

Listing 3.17: Applying a `part()` operation to the module from listing 3.16 results in a changed geometry

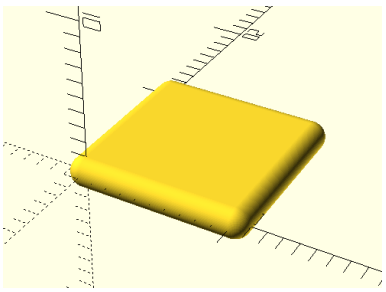


Figure 3.10: The resulting geometry of listing 3.17

An end user would presumably not expect that applying a part attribute to a CSG tree within the OpenSCAD script would also result in a changed geometry. If the code to produce the geometry is encapsulated within a module, it's even less clear to someone writing an OpenSCAD script why this might be happening.

Additionally, the approach of resolving attributes ahead of time also caused problems with caches used by OpenSCAD. OpenSCAD has a built in cache, which stores a partial CSG tree and its resulting geometry. This is based on the assumption that two identical CSG subtrees will always result in the same geometry, which is broken by preresolution of attributes.

```
color("blue") translate([-15,0,0]) {
   union() {
    cube([10,20,10]);
    cube([20,10,10]);
  }
}
color("red") translate([15,0,0]) {
  union() {
```

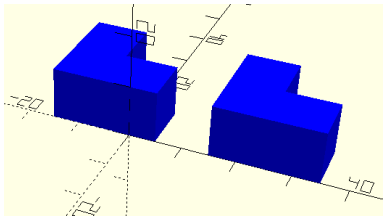


Figure 3.11: The unexpected result of listing 3.18

```
9       cube([10,20,10]);
10      cube([20,10,10]);
11    }
12  }
```

Listing 3.18: An OpenSCAD script which produced unexpected results due to caching

Because the CSG subtree starting at the `union()` operation on line 2 and line 7 are identical, the geometry produced by this CSG subtree is cached. However, because the attributes are resolved before any geometries are generated, the geometry that ends up being cached has a color attribute of "blue". When the subtree on line 7 is processed, this geometry is recalled, and thus the second geometry ends up having the wrong color, as can be seen in fig. 3.11.

In order to solve these problems, the implementation was changed to no longer recursively calculate derived attributes of CSG nodes. Instead, when a `LeafNode` creates a geometry, it is created without any attributes. Attributes are then applied to geometries only whenever a `ColorNode`, `MaterialNode` or `PartNode` is traversed inside the `GeometryEvaluator` class. All child geometries of these nodes are given attributes matching those of the node.

This way, the geometries created by both `union()` operations in listing 3.18 are void of attributes, and thus identical, solving the cache issues. Colors are only applied to them once they are passed through their respective `color()` operations, at which point the CSG subtrees are no longer identical. Further, The two groups of geometries in listing 3.17 will always be irreconcilable at the point when the geometry is calculated, which means that overwriting the part attribute of the `rods()` module will no longer produce a change in geometry either.

3.7.3 Attributes and the OpenCSG Renderer (Preview Mode)

For preview mode, OpenSCAD uses the OpenCSG library. In this mode, instead of generating a complete geometry, an approximation of the finished object is rendered from only partially generated geometries, with some CSG operations like `union()` and `difference()` being performed during rendering instead, if possible.

Unfortunately, this presents a number of roadblocks for the work of this thesis. For example, because `difference()` operations are resolved during rendering, if a geometry is subtracted from one with a different color, the surfaces produced by the negative of the subtracted geometry will be in the wrong color.

```
difference() {
  color("red") cube([10,10,10]);
  translate([5,5,5]) color("green") cube([10,10,10]);
}
```

Listing 3.19: Subtracting two colored geometries from one another.

Because of this, the preview for listing 3.19 will not match what you would expect when dealing with volumetric colors rather than colored surfaces, as can be seen in fig. 3.12.

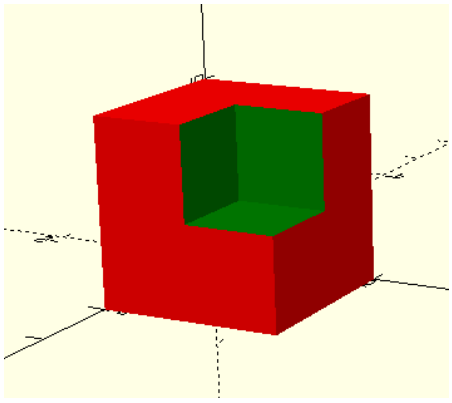


Figure 3.12: Preview of a subtraction of differently colored cubes in preview mode

Furthermore, since `union()` operations are also simulated during rendering if possible, this means that shapes which gain colors due to being inside a `union()` operation with a colored element as the first child, will not be shown in the correct color as can be seen in fig. 3.13.

```
union() {
  color("green") cube([5,5,5]);
  translate([10,0,0]) cube([5,5,5]);
}
```

Listing 3.20: Due to color being a reconcilable attribute, both cubes are supposed to be merged to one geometry which is green

Intersections are also handled in a special way to speed up rendering during preview mode. This means that they do not respect attributes in preview mode, and all child geometries of an `intersection()` operation are treated as if they are reconcilable when previewed.

```
intersection() {
  part("part1") cube([1,1,1]);
  part("part2") translate([2,1,1]) cube([1,1,1]);
}
```

Listing 3.21: Irreconcilable geometries are not treated as such during previews of intersections.

Thus, in preview mode, the code in listing 3.21 will produce no geometry at all, since the two cubes do not overlap. However, because their attributes are reconcilable, both cubes

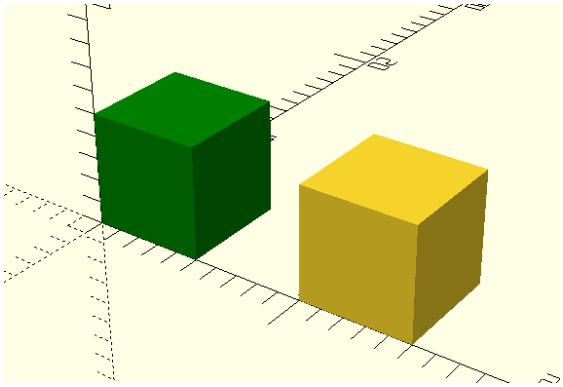


Figure 3.13: The second cube is not rendered green in preview mode, because `union()` operations are rendered as separate geometries in preview mode.

should be rendered unchanged, as is the case when rendering the final geometry.

The simple way to resolve this problem would be to simply drop the special handling of these operations in preview mode. However, the entire point of preview mode is to be able to skip potentially expensive geometry calculations for these operations in order to produce a faster render of the final geometry. If we cannot do this anyway, preview mode might as well be removed entirely. Unfortunately no good solution to these issues was found in the time which was available for producing this thesis.

3.7.4 3MF Materials Must Have a Color

Lib3MF unfortunately does not allow adding a material without also specifying a color for that material [24]. Inside Lib3MF, colors are represented as structs of 4 bytes representing the RGBA values of the material color. This means, when exporting 3MF-files, a default color has to be chosen, and no matter which value is picked, it will be in collision with a color that can also be specified using the `color()` operation.

Consequently, when you export an OpenSCAD file in the 3MF-format and then re-import the same file using the `import()` operation, objects without a color will now appear white (the default color which was chosen). The only way around this would be to choose a technically valid RGBA color as the default color and ignore that color on import. Since exporting files and then re-importing them is probably a less likely use case than simply exporting them for use in another program, this approach was not taken.

4 Discussion and Evaluation

This chapter takes a look at the initial stated goals, and evaluates to what extent they have been met. It will also examine what improvements could be made to build on the work in this thesis.

4.1 Basic Evaluation

The initial goal was to implement support for creation and export of multi-material models in OpenSCAD, by allowing the specification of color, material and belonging to a specific part of a geometry via the OpenSCAD script language. This goal has been achieved, as it is now possible to do all of these things in the modified version of OpenSCAD produced as part of this thesis.

However, the most glaring flaw of the work in this thesis is the lack of support for preview mode, owed to the fact that its rendering shortcuts do not play well with the new semantics which have been defined for attribute-dependent behavior of CSG operations. The initial goal of this thesis cannot truly be declared as fully accomplished if an essential part of the OpenSCAD modeling workflow lacks support for the new additions.

Nevertheless, it is now possible to use the software to model and export non-homogeneous objects, and rendering the fully calculated geometry via "Design" → "Render" can be substituted for preview mode, though it is slower to produce results.

4.2 Practical Use and Conclusion

Beyond the prima facie adherence to the stated objective, it's also worth taking a look at the results from a practical usage standpoint.

One important note about the final implementation is that it is now possible to create a file containing multiple geometries which occupy the same physical space as one another. This means that we are now able to export files containing geometries which are impossible to cleanly map to a physical object.

```
1 color("red") cube([10,10,20]);
2 color("green") cube([10,20,10]);
```

Listing 4.1: A script producing two geometries which intersect one another

While the objects can still be moved around separately inside common slicer software like PrusaSlicer, this represents a potential pitfall for the modeling of physical objects. For the modeling of non-physical objects, it could be argued to be an advantage, since preventing

geometries from occupying the same space limits the total possibilities of what can be created using OpenSCAD. Any non-overlapping geometry can also be achieved by simply subtracting the final geometries inside a script from one another. However, the focus of this thesis is the use case of 3D printing, so having to manually subtract objects from one another is more of a hindrance than feature.

Another point of consideration is that the semantics which were chosen are not intuitive. It might be hard for someone to understand why geometries of different part and material attributes are not possible to combine without first marking them as the same, especially in light of the fact that color attributes do not behave the same way. This represents a compromise between power of expressiveness and backwards compatibility, but it's hard to imagine the same semantics being chosen by anyone implementing similar functionality on a new software package, not beholden to backwards compatibility with existing scripts.

Most of the labor in design and bug-fixing has been dealing with edge cases arising from situations where the intent of the end-user writing an OpenSCAD script is ambiguous. This is also why whichever chosen semantics for combining geometries with different attributes end up being somewhat unintuitive (except perhaps forbidding to do so entirely) as different users may simply have different expectations on what the correct behavior should be.

It's worth noting that almost all these edge cases would never crop up to begin with when OpenSCAD scripts are written in such a way that geometry attributes are only ever defined at the very root of the CSG tree (or as a direct child of another attribute declaration). This will cause the entire geometry to first be generated as it would be in an unmodified version of OpenSCAD, and only then be given attributes at the very end.

It's probably a good coding practice to separate code into distinct units for each material and/or part anyway, especially since different geometries may overlap, and it might be necessary to subtract them from one another. Say for example you are modeling a handle for a tool, and wish to add a rubberized outer layer. It makes sense to separate the handle and rubber grip into their own modules, and then in the final step subtract one from the other to ensure that they do not overlap.

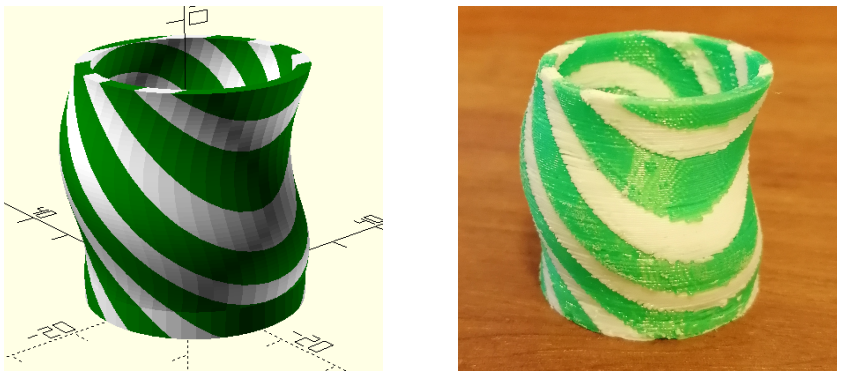
```
module handle() {
  ...
}

module rubberized_grip() {
  ...
}

material("PLA") difference() {
  handle();
  rubberized_grip();
}

material("TPA") rubberized_grip();
```

Listing 4.2: An example object composed of two materials, each given its own module

This also completely eliminates the need to worry about behaviors related to irreconcilable geometries. Of course, expecting people to structure their code in a very specific way is perhaps a little too optimistic, and could end up preventing creative use cases that were not accounted for, in any case. Additionally, this approach fails once dealing with imported files containing geometries with attributes, where encountering attribute semantics becomes unavoidable.

While the semantics which were chosen are not the simplest to understand, they are more powerful than a simpler solution, and allow better synergy with modules and file import, allowing for selective manipulation of different objects. Syntactically, they fit well into the existing model of OpenSCAD, and preserve a reasonable degree of backwards compatibility.

The source code to the version of OpenSCAD developed as part of this thesis can be found at https://github.com/nle1990/openscad-mm/. It's also worth noting that for 3MF import and export support, it's important to install the development version of lib3mf before compiling the software. On Debian and Ubuntu Linux this can be done by installing the lib3mf-dev package. Alternatively, it can also be compiled from source.



OpenSCAD Render Printed Object

Figure 4.1: An example print of an object designed using the newly added multi-material operations

```
module vase() {
  translate([0,0,4]) {
    linear_extrude(height = 30, convexity = 10, twist = 360, $fn = 50) {
      translate([1.5, 0, 0]) difference() {
        circle(r = 16);
        circle(r = 14);
      }
    }
  }
  translate([1.5, 0, 0]) linear_extrude(height = 4, convexity = 10, $fn =
    50) {
    circle(r = 16);
  }
```

```
}

module vase_color1() {
  difference() {
    vase();
    for(i = [1 : 1 : 8]) translate([-40,-20,-40 + 10*i]) rotate([45,0,0])
    cube([80,80,3]);
  }
}

color("green") vase_color1();
color("white") difference() {
  vase();
  vase_color1();
}
```

Listing 4.3: The source code used to produce the object in fig. 4.1

4.3 Future Work

There are several ways in which the current implementation could be improved. Some ideas to further build on the work in this thesis are discussed in this section.

4.3.1 Visually Distinguish Non-Visible Attributes

One improvement which has been hinted at already in previous sections is marking the non-visible attributes (i.e. part and material) in some way during rendering, to make it visually apparent which segments are going to be exported as distinct objects. The easiest thing to do would be to give each unique attribute combination a color, but obviously this conflicts with the `color()` operation. A possible solution would be to only do this for elements that do not have a color assigned to them, but while this would make distinct part and material attributes more apparent, it has the drawback of making the color attributes of any given geometry less apparent, since it's no longer clear whether a geometry has the color its being displayed at, or if the color is merely a stand-in for other attributes.

Another possibility would be to give the geometries a texture, perhaps even just a repeating text which spells out their material and part attribute on its surfaces. This would be less trivial to do than just picking a color, but avoid ambiguity. Alternatively, each unique material/part combination could be given its own outline color. A perhaps less cluttered looking option would be to display a tooltip with attributes, when hovering the mouse pointer over a geometry inside the render window.

Regardless of the exact nature this would be implemented, it makes sense to mark objects by their irreconcilable attributes in some way as a visual aid for modeling and especially to simplify spotting errors without having to rely on reading the script code alone.

4.3.2 Support More File Formats

File export has only been made attribute aware for 3MF, AMF and SVG (for two-dimensional geometries). It makes sense to support exporting attributes for more formats that allow such options. Additionally, even for file formats which do not directly support attributes or multiple geometries, it might make sense to offer an option to export each individual geometry of an OpenSCAD script as its own file, which could then be separately imported into slicer software. The STL file format in particular continues to be the most popular format for 3D printing, but only supports pure geometry information, and cannot store attributes [25].

Support for attribute-aware file import is even more limited, currently only supporting 3MF files. Here too, it would make sense to extend this support to other formats, which offer definition of matching geometry attributes.

4.3.3 Optionally Modifying Irreconcilability Semantics of Operations

Another useful addition would be to allow each CSG operation to specify its semantics explicitly. This would be useful for situations where we do not have full control over the exact geometries fed into an operation, and cannot manually perform separate operations on them, which is the case with external modules and file imports. For example, we might want to perform an operation on two geometries of different colors without combining them. If we were able to instruct an operation to treat colors as irreconcilable as well, we would be able to do this.

```
union(semantics = "strict") {
  external_module();
  color("green") cube([10,10,10]);
}
```

Listing 4.4: A hypothetical syntax for requesting strict attribute matching

Similarly, in the case of the `difference()` operation, when working with external files or modules, its more lax attribute semantics might become a hindrance. The "semantics" parameter would allow us to only subtract from some geometries within a module, by matching attributes of subtracted geometries.

Another option would be to perform a more lax matching of attributes, and instruct an operation to simply treat all attributes as reconcilable, making it easy to merge everything together.

```
union(semantics = "permissive") {
  external_module();
}
```

Listing 4.5: A hypothetical syntax for requesting no attribute matching at all

This would be the less useful of the two new options, as the same can be achieved by simply overwriting both part and material attributes:

```
union() {
  part("") material("") external_module();
}
```

Listing 4.6: An alternate way of achieving the results of listing 4.5

Additionally, it might also make sense to set the default semantics for the whole script, e.g. by writing into a special variable

```
$semantics = "permissive";
union() {
  material("PLA") cube([5,5,20]);
  material("ABS") cube([5,20,5]);
}
```

Listing 4.7: A hypothetical syntax for changing the attribute semantics of the whole script

4.3.4 A Null Geometry Object

The problem of null geometries not retaining their attributes was already covered in section 3.7.1, and the chosen solution was to treat them as if they have the default attributes. Another potential solution is introducing a new polymorphic `Geometry` type representing a null geometry. This would be an object which stores no geometry data, but can have attributes applied to it, to replace the current use of null pointers as null geometries. This would be a rather invasive change, as anywhere where a pointer to a geometry is set to null, an object of this type would have to be created. Likewise, anywhere where a geometry pointer is checked for being null, a dynamic pointer cast would have to be performed instead, checking for this type of object.

But this alone would not be sufficient, because in cases where null pointers are normally returned, it's not enough to just return a null geometry attribute, it must also inherit the attributes from its parent geometry, if one exists. For example, if a two-dimensional geometry is used in a three-dimensional operation, it is replaced with a null pointer. The null geometry this two-dimensional geometry would be replaced with instead would then have to be assigned the attributes of that two-dimensional geometry.

If, for example, an operation is performed on only a null geometry, the same null geometry which was given as an input would have to be returned again, to retain its outputs. Further, if an operation is performed on multiple empty geometries or null geometries of different attributes, multiple null geometries would have to be returned, one for each set of attributes. These would also have to be carried forward through other operations. On some operations, null geometries are often just discarded in a for-loop iterating through all the geometries an operation is about to be performed on, they would now have to be carried forward and returned if no result with the same attributes is produced.

This also means that operations could now produce both geometries and null geometries, or even multiple different null geometries as a result.

```
union() {
  part("part1") {}
  part("part2") cube([1,1,1]);
}
```

Listing 4.8: An operation which would produce a geometry and a null geometry simultaneously

The drawback of this change would be that modules might end up returning null geometries, which affect operations performed on that module with geometries of the same attributes of that null geometry. For example, `intersection()` operations would always produce an empty geometry for geometries of the same attributes as such a returned null geometry. It might be hard for an end-user to understand why this is happening, especially in cases of null geometries returned by modules, since there is no way to detect the presence of a null geometry visually, despite it potentially having an effect on the results of operations.

While this is an example where null geometries might produce unexpected results, they make many other scenarios (which are arguably more common) much more intuitive.

```
intersection() {
  part("part1") {}
  part("part1") cube([1,1,1]);
}
```

Listing 4.9: This code would now produce an empty geometry, as expected

Bibliography

[1] Rhys Jones, Patrick Haufe, Edward Sells, Pejman Iravani, Vik Olliver, Chris Palmer, Adrian Bowyer. (2011). *RepRap - the replicating rapid prototyper.* In: Robotica 29(1), pp. 177-191

[2] Campbell, Thomas & Williams, Christopher & Ivanova, Olga & Garrett, Banning. (2011). *Could 3D Printing Change the World? Technologies, Potential, and Implications of Additive Manufacturing.*

[3] Sai, P.Chennakesava & Yeole, Shivraj Narayan. (2014). *Fused Deposition Modeling - Insights.*

[4] Colin Dow. (2022). *Simplifying 3D Printing with OpenSCAD: Design, build, and test OpenSCAD programs to bring your ideas to life using 3D printers.* pages 31 - 39.

[5] OpenSCAD Issue Tracker: Feature Request: color(), material() and part()
https://github.com/openscad/openscad/issues/1608

[6] Rafiee M, Farahani RD, Therriault D. *Multi-Material 3D and 4D Printing: A Survey*. Adv Sci (Weinh). 2020 Apr 30;7(12):1902307. doi: 10.1002/advs.201902307. PMID: 32596102; PMCID: PMC7312457.

[7] Abilgaziyev, A. & Kulzhan, T. & Raissov, N. & Ali, Md & Ko, Match & Mir- Nasiri, Nazim. (2015). Design and development of multi-nozzle extrusion system for 3D printer. 1-5. 10.1109/ICIEV.2015.7333982.

[8] OpenSCAD GitHub Wiki: Experimental Features
https://github.com/openscad/openscad/wiki/Experimental-Features

[9] James D. Foley, Andries van Dam, Steven K. Feiner, John F. Hughes. (1995). *Computer Graphics. Principles and Practice in C.* pages 557 - 558

[10] Illustration created by Wikimedia user Zottie under CC-BY-SA 3.0
https://commons.wikimedia.org/wiki/File:Csg_tree.png

[11] Joshua M. Pearce. *Open-Source Lab: How to Build Your Own Hardware and Reduce Research Costs*. (2014). pages 165 - 166

[12] Gohde, Justin & Kintel, Marius. (2021). *Programming with OpenSCAD: A Beginner's Guide to Coding 3D-Printable Objects.*

[13] GNU Bison documentation
https://www.gnu.org/software/bison/manual/bison.html#Introduction

[14] OpenSCAD Issue Tracker: Lazy union (aka. no implicit union)
https://github.com/openscad/openscad/issues/350

[15] 3MF Core Specification
https://github.com/3MFConsortium/spec_core/blob/master/3MF%20Core%20Specification.md

[16] ISO/ASTM 52915:2016. (2016). *Specification for Additive Manufacturing File Format (AMF) Version 1.2*

[17] lib3mf package : Ubuntu
https://launchpad.net/ubuntu/+source/lib3mf

[18] Debian – Details of package lib3mf1 in sid
https://packages.debian.org/sid/lib3mf1

[19] Arch Linux - lib3mf-1 1.8.1-6 (x86_64)
https://archlinux.org/packages/community/x86_64/lib3mf-1/

[20] lib3mf - Fedora Packages
https://packages.fedoraproject.org/pkgs/lib3mf/lib3mf/index.html

[21] OpenSCAD Issue Tracker: inconsistency between F5 and F6 with intersection_for
https://github.com/openscad/openscad/issues/666

[22] OpenSCAD Issue Tracker: Subtracting from empty union yields nonempty result
https://github.com/openscad/openscad/issues/3311

[23] OpenSCAD Issue Tracker: Preview renders cube(0) intersection incorrectly
https://github.com/openscad/openscad/issues/3312

[24] lib3mf Documentation - AddMaterial()
https://lib3mf.readthedocs.io/en/master/source/Cpp/lib3mf_BaseMaterialGroup.html#_CPPv4N6Lib3MF18CBaseMaterialGroup11AddMaterialERKNSt6stringERK6sColor

[25] Colin Dow. (2022). *Simplifying 3D Printing with OpenSCAD: Design, build, and test OpenSCAD programs to bring your ideas to life using 3D printers.* pages 56 - 58.

Eidesstattliche Erklärung

Hiermit versichere ich an Eides statt, dass ich die vorliegende Arbeit im Bachelorstudiengang Informatik selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel — insbesondere keine im Quellenverzeichnis nicht benannten Internet-Quellen — benutzt habe. Alle Stellen, die wörtlich oder sinngemäß aus Veröffentlichungen entnommen wurden, sind als solche kenntlich gemacht. Ich versichere weiterhin, dass ich die Arbeit vorher nicht in einem anderen Prüfungsverfahren eingereicht habe und die eingereichte schriftliche Fassung der elektronischen Abgabe entspricht.

Hamburg, den 04.05.2023 Nicolaus Johannes Leopold gen. Eggert

Veröffentlichung

Ich stimme der Einstellung der Arbeit in die Bibliothek des Fachbereichs Informatik zu.

Hamburg, den 04.05.2023 Nicolaus Johannes Leopold gen. Eggert