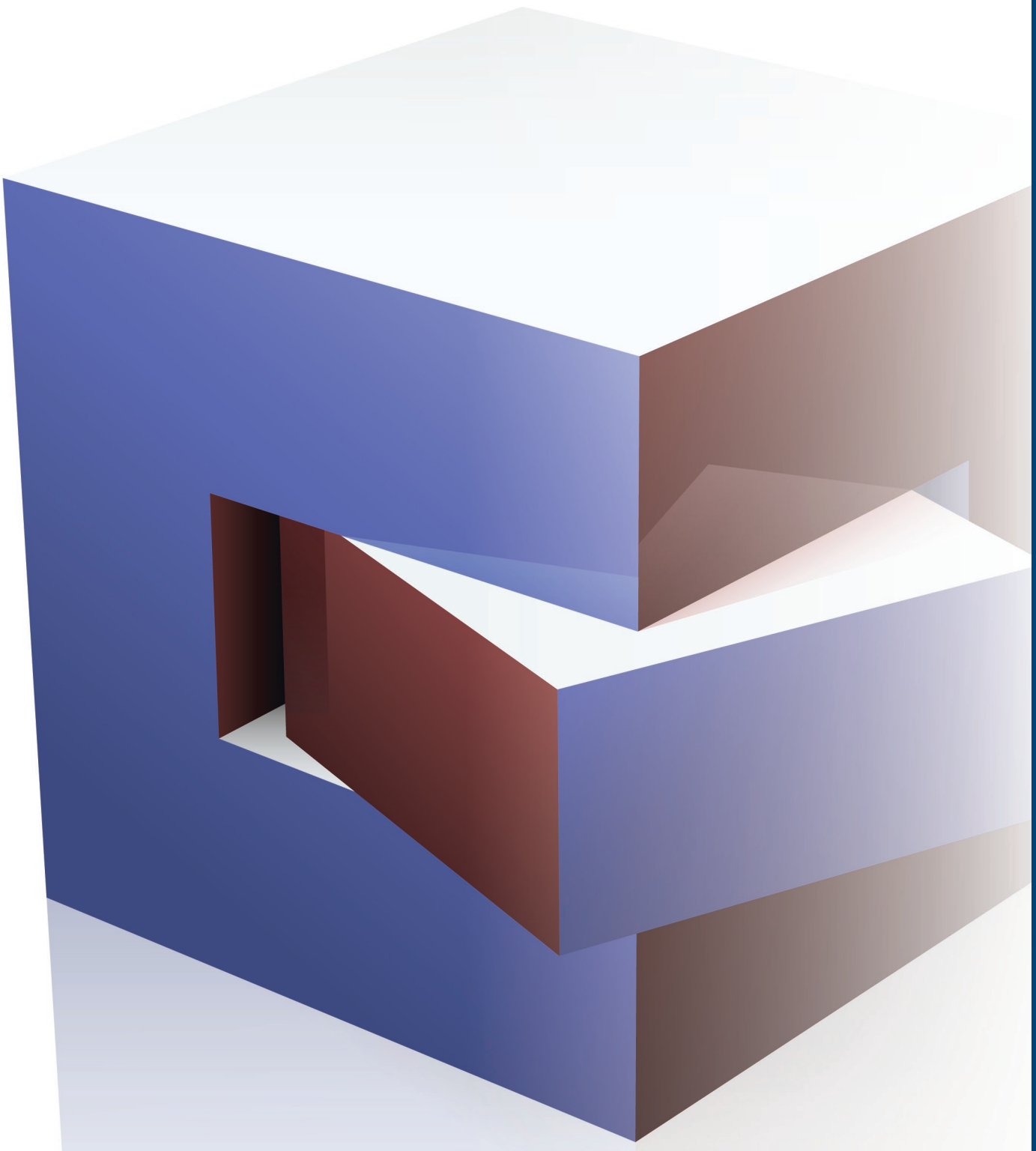
# CARVE DOCUMENTATION



## Software Documenation

September 2008

# Contents

# List of Figures

# List of Tables

# Listings

# 1. | Representation of polyhedra

Carve polyhedra are defined by collections of vertices (instances of `carve::poly::Vertex`) that define points in 3-dimensional space, and collections of faces (instances of `carve::poly::Face`) that define the connectivity of vertices. Because faces refer to vertices by pointer, vertex identity is determined by address rather than by location in 3-dimensional space.

Faces are oriented anticlockwise in a right handed coordinate system. Although a face may consist of more than three vertices, all vertices of any given face must lie on a single plane.

A polyhedron defined by a set of faces and vertices consists of one or more connected surfaces. The decomposition of a set of faces into surfaces is computed automatically, and shared vertices (Figure 1.1) and edges (Figure 1.2) are handled correctly. A polyhedron may not, however, be self intersecting.

Each surface is either "closed" or "open". A closed surface obeys the property that for every edge (determined by a pair of consecutive vertices forming part of a face) there exists an edge of the opposite orientation that is part of some other face.

A closed surface bounds a non-zero (possibly infinite) volume of space. The space defined by a surface depends upon the orientation of its defining faces. By inverting the vertex order of all faces of a closed surface, the complementary volume is created. For example, a cube with faces ordered clockwise in a right-handed coordinate system describes the infinite volume consisting of all space except that delimited by the cube.

Used carefully, more than one closed surface may be combined to create a shell. A surface representing an infinite volume enclosed within a surface representing a finite volume represents a hollow solid, and such solids are handled correctly during CSG operations.

## 1.1 Construction of Polyhedra

A polyhedron may be constructed in a number of ways.

In the first case (Listing 1.1) a vector of faces and a vector of vertices is provided.

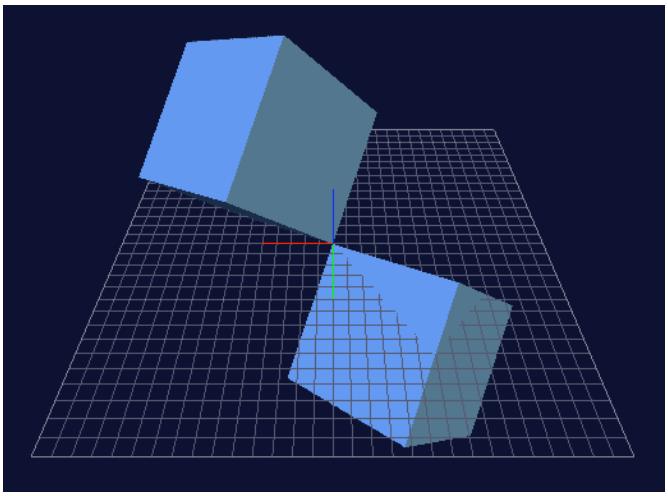


Figure 1.1: A polyhedron consisting of two surfaces sharing a vertex

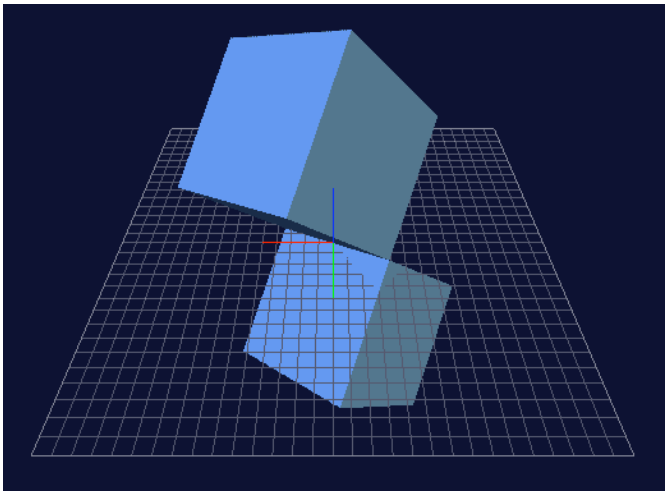


Figure 1.2: A polyhedron consisting of two surfaces sharing an edge

```
carve::poly::Polyhedron(
    std::vector<carve::poly::Face *> &_faces,
    std::vector<carve::poly::Vertex> &_vertices,
    bool _recalc = false);
```

Listing 1.1: `carve::poly::Polyhedron` constructor 1

```
carve::poly::Polyhedron(
    std::vector<carve::poly::Face *> &_faces,
    bool _recalc = false);
```

Listing 1.2: `carve::poly::Polyhedron` constructor 2

```
carve::poly::Polyhedron(
    std::list<carve::poly::Face *> &_faces,
    bool _recalc = false);
```

Listing 1.3: `carve::poly::Polyhedron` constructor 3

```
carve::poly::Polyhedron(
    const std::vector<carve::geom3d::Vector> &vertices,
    int n_faces,
    const std::vector<int> &face_indices);
```

Listing 1.4: `carve::poly::Polyhedron` constructor 4

```
#include <carve/polyhedron.hpp>

carve::poly::Polyhedron *makeCube(
    const carve::math::Matrix &t = carve::math::Matrix()) {

  std::vector<carve::poly::Vertex> verts;
  std::vector<carve::poly::Face *> faces;

  verts.reserve(8);
  faces.reserve(6);

  verts.push_back(
    carve::poly::Vertex(t * carve::geom::VECTOR(+1.0, +1.0, +1.0)));
  verts.push_back(
    carve::poly::Vertex(t * carve::geom::VECTOR(-1.0, +1.0, +1.0)));
  verts.push_back(
    carve::poly::Vertex(t * carve::geom::VECTOR(-1.0, -1.0, +1.0)));
  verts.push_back(
    carve::poly::Vertex(t * carve::geom::VECTOR(+1.0, -1.0, +1.0)));
  verts.push_back(
    carve::poly::Vertex(t * carve::geom::VECTOR(+1.0, +1.0, -1.0)));
  verts.push_back(
    carve::poly::Vertex(t * carve::geom::VECTOR(-1.0, +1.0, -1.0)));
  verts.push_back(
    carve::poly::Vertex(t * carve::geom::VECTOR(-1.0, -1.0, -1.0)));
  verts.push_back(
    carve::poly::Vertex(t * carve::geom::VECTOR(+1.0, -1.0, -1.0)));

  faces.push_back(
    new Face(&verts[0], &verts[1], &verts[2], &verts[3]));
  faces.push_back(
    new Face(&verts[7], &verts[6], &verts[5], &verts[4]));
  faces.push_back(
    new Face(&verts[0], &verts[4], &verts[5], &verts[1]));
  faces.push_back(
    new Face(&verts[1], &verts[5], &verts[6], &verts[2]));
  faces.push_back(
    new Face(&verts[2], &verts[6], &verts[7], &verts[3]));
  faces.push_back(
    new Face(&verts[3], &verts[7], &verts[4], &verts[0]));

  // note that carve::poly::Polyhedron takes ownership of face
  // pointers and the contents of the vertex array.
  return new carve::poly::Polyhedron(faces, vertices);
}
```

Listing 1.5: Constructing a cube directly

```
#include <carve/input.hpp>
#include <carve/polyhedron.hpp>

carve::poly::Polyhedron *makeCube(
    const carve::math::Matrix &t = carve::math::Matrix()) {

  carve::input::PolyhedronData data;

  data.addVertex(t * carve::geom::VECTOR(+1.0, +1.0, +1.0));
  data.addVertex(t * carve::geom::VECTOR(-1.0, +1.0, +1.0));
  data.addVertex(t * carve::geom::VECTOR(-1.0, -1.0, +1.0));
  data.addVertex(t * carve::geom::VECTOR(+1.0, -1.0, +1.0));
  data.addVertex(t * carve::geom::VECTOR(+1.0, +1.0, -1.0));
  data.addVertex(t * carve::geom::VECTOR(-1.0, +1.0, -1.0));
  data.addVertex(t * carve::geom::VECTOR(-1.0, -1.0, -1.0));
  data.addVertex(t * carve::geom::VECTOR(+1.0, -1.0, -1.0));

  data.addFace(0, 1, 2, 3);
  data.addFace(7, 6, 5, 4);
  data.addFace(0, 4, 5, 1);
  data.addFace(1, 5, 6, 2);
  data.addFace(2, 6, 7, 3);
  data.addFace(3, 7, 4, 0);

  return data.create();
}
```

Listing 1.6: Constructing a cube using `carve::input::PolyhedronData`

# 2. | CSG Operations

The `carve::csg::CSG` class is responsible for managing CSG calculations. It provides methods for CSG binary computations between both closed and open polyhedra, as well as divison of polyhedra by their common line of intersection. The operation computed by methods of the `carve::csg::CSG` class may be chosen from the standard primitive binary operations or may be defined by the caller.

The CSG computation may be influenced by the registration of hook objects that can be used to perform such tasks as triangulation of result faces and transfer and interpolation of attributes from source polyhedra to the result.

## 2.1 CSG Operations on Closed Manifolds

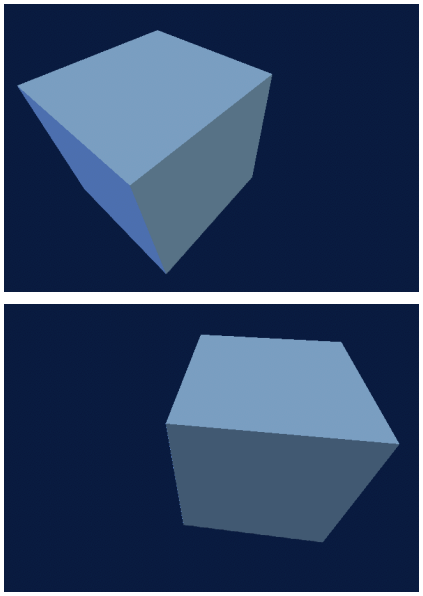
The `compute` method of `carve::csg::CSG` has two prototypes:

```
carve::poly::Polyhedron  *compute(
    const carve::poly::Polyhedron  *a,
    const carve::poly::Polyhedron  *b,
    carve::csg::CSG::OP op,
    carve::csg::V2Set *shared_edges = NULL,
    carve::csg::CSG::CLASSIFY_TYPE classify_type = CLASSIFY_NORMAL);
```

```
carve::poly::Polyhedron  *compute(
    const carve::poly::Polyhedron  *a,
    const carve::poly::Polyhedron  *b,
    carve::csg::CSG::Collector &collector,
    carve::csg::V2Set *shared_edges = NULL,
    carve::csg::CSG::CLASSIFY_TYPE classify_type = CLASSIFY_NORMAL);
```

These methods compute a boolean operation between polyhedra `a` and `b`. In the first case, the operation is determined by the enumeration `carve::csg::CSG::OP`, which can take the values:

- `UNION`
- `INTERSECTION`
- `A_MINUS_B`
- `B_MINUS_A`
- `SYMMETRIC_DIFFERENCE`



Manifold A

Manifold B

Table 2.1: Input polyhedra.

Results for these boolean operations are shown in Table 2.1 and Table 2.2.

In the second case, the result is determined by a custom collector. Custom collectors allow the caller to programatically define which regions of the intersected input polyhedra appear in the output.

The `shared_edges` parameter provides a way to access the computed set of edges that defines the point of intersection of the two polyhedra.

The algorithm used to classify connected components of the intersected polyhedra is determined by the parameter `classify_type`. The type of `classify_type` is an enumeration taking values from the set {`CLASSIFY_NORMAL`, `CLASSIFY_EDGE`}.

The classifier is responsible for classifying portions of each polyhedron bounded by the line of intersection as either:

- `carve::csg::FACE_OUT`
  The group is outside the space defined by the opposing polyhedron.
- `carve::csg::FACE_IN`
  The group is inside the space defined by the opposing polyhedron.
- `carve::csg::FACE_ON_ORIENT_IN`
  The group is lying on the surface of the opposing polyhedron, oriented towards its interior.
- `carve::csg::FACE_ON_ORIENT_OUT`
  The group is lying on the surface of the opposing polyhedron, oriented towards its exterior.

with respect to (each surface of) the opposing polyhedron.

## 2.2 CSG Operations on Open Manifolds

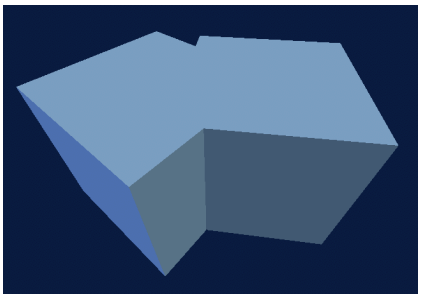
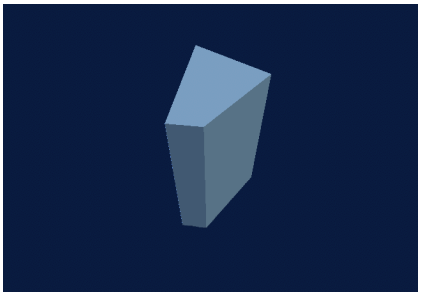
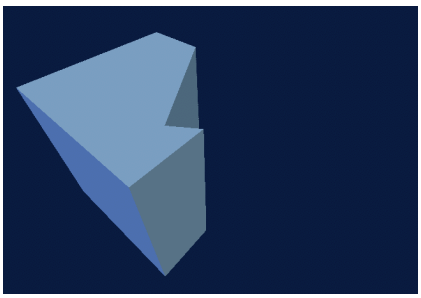
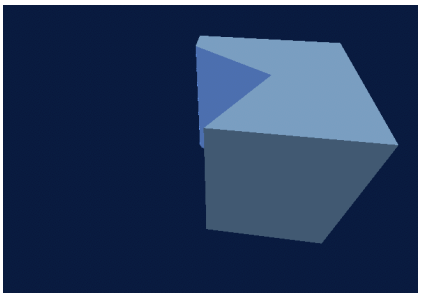
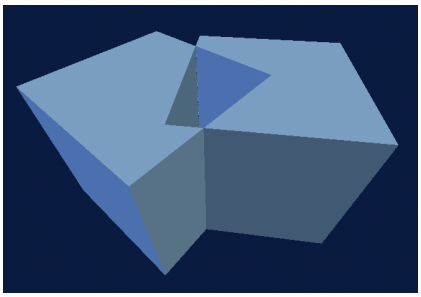
| | |
|---|---|
|  | Operation: Union (A \| B)<br>Enumeration: `carve::CSG::UNION` |
|  | Operation: Intersection (A & B)<br>Enumeration: `carve::CSG::INTERSECTION` |
|  | Operation: Difference (A - B)<br>Enumeration: `carve::CSG::A_MINUS_B` |
|  | Operation: Difference (B - A)<br>Enumeration: `carve::CSG::B_MINUS_A` |
|  | Operation: Symmetric Difference (A B)<br>Enumeration: `carve::CSG::SYMMETRIC_DIFFERENCE` |

Table 2.2: The result of predefined CSG operations.

# 3. | Attribute Interpolation

```
#include <carve/interpolator.hpp>

struct tex_t {
  float u, v;

  tex_t() : u(0.0f), v(0.0f) { }
  tex_t(float _u, float _v) : u(_u), v(_v) { }
};

// interpolated attributes must support scalar multiplication.
tex_t operator*(double s, const tex_t &t) {
    return tex_t(t.u * s, t.v * s);
}

// interpolated attributes must support operator+=.
tex_t &operator+=(tex_t &t1, const tex_t &t2) {
    t1.u += t2.u; t1.v += t2.v;
    return t1;
}

void associateTextureVertices(
    carve::poly::Polyhedron *cube,
    carve::interpolate::FaceVertexAttr<tex_t> &fv_tex) {

  fv_tex.setAttribute(cube->faces[0], 0, tex_t(1.0f, 1.0f));
  fv_tex.setAttribute(cube->faces[0], 1, tex_t(0.0f, 1.0f));
  fv_tex.setAttribute(cube->faces[0], 2, tex_t(0.0f, 0.0f));
  fv_tex.setAttribute(cube->faces[0], 3, tex_t(1.0f, 0.0f));

  // ... continue to record other texture coordinates by
  //     face pointer and vertex number.
}
```

Listing 3.1: Associating texture coordinates with a cube

```
#include <carve/csg.hpp>

carve::poly::Polyhedron *doCSG() {
  carve::poly::Polyhedron *result;

  carve::poly::Polyhedron *cube_1 = makeCube();
  carve::poly::Polyhedron *cube_2 = makeCube(
    carve::math::Matrix::ROT(.4, .2, .3, .4));

  carve::interpolate::FaceVertexAttr<tex_t> fv_tex;
  associateTextureVertices(cube_1, fv_tex);

  carve::csg::CSG csg;
  fv_tex.installHooks(csg);
  result = csg.compute(cube_1, cube_2, carve::csg::CSG::A_MINUS_B);
}
```

Listing 3.2: Interpolating texture coordinates during a CSG operation