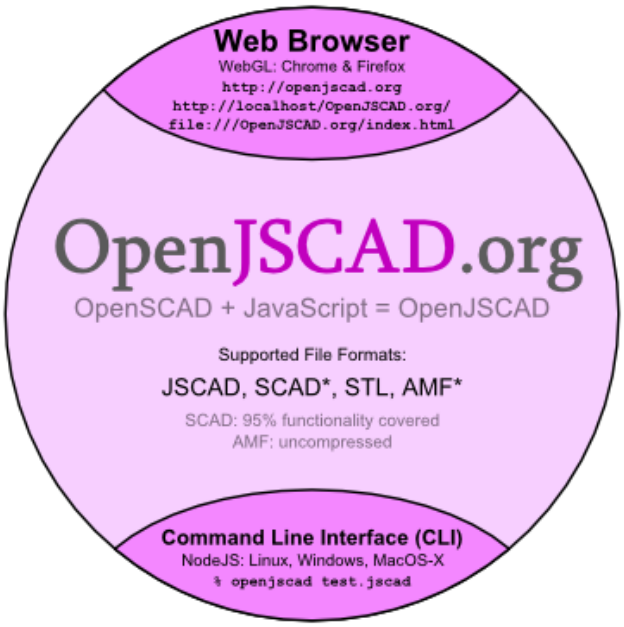
OpenJSCAD User Guide

Welcome to **OpenJSCAD.org** User and Programming Guide.

Just for sake of providing context, **OpenJSCAD.org** is built on OpenJsCad (Github) (https://joostn.github.com/OpenJsCad/), which itself was inspired by OpenSCAD.org (http://openscad.org), and essentially provides a programmers approach to develop 3D models. In particular, this enhancement is tuned towards creating precise models for 3D printing.



OpenJSCAD programmers using OpenSCAD may welcome

- ability to use JavaScript programming concepts and libraries
- ability to create and manipulate 3D shapes, as well as 2D shapes
- support for OpenSCAD source-code (approximately 80% of all functions)
- additional functions to ease the transition to OpenJSCAD

Read on to find about more about **OpenJSCAD**.

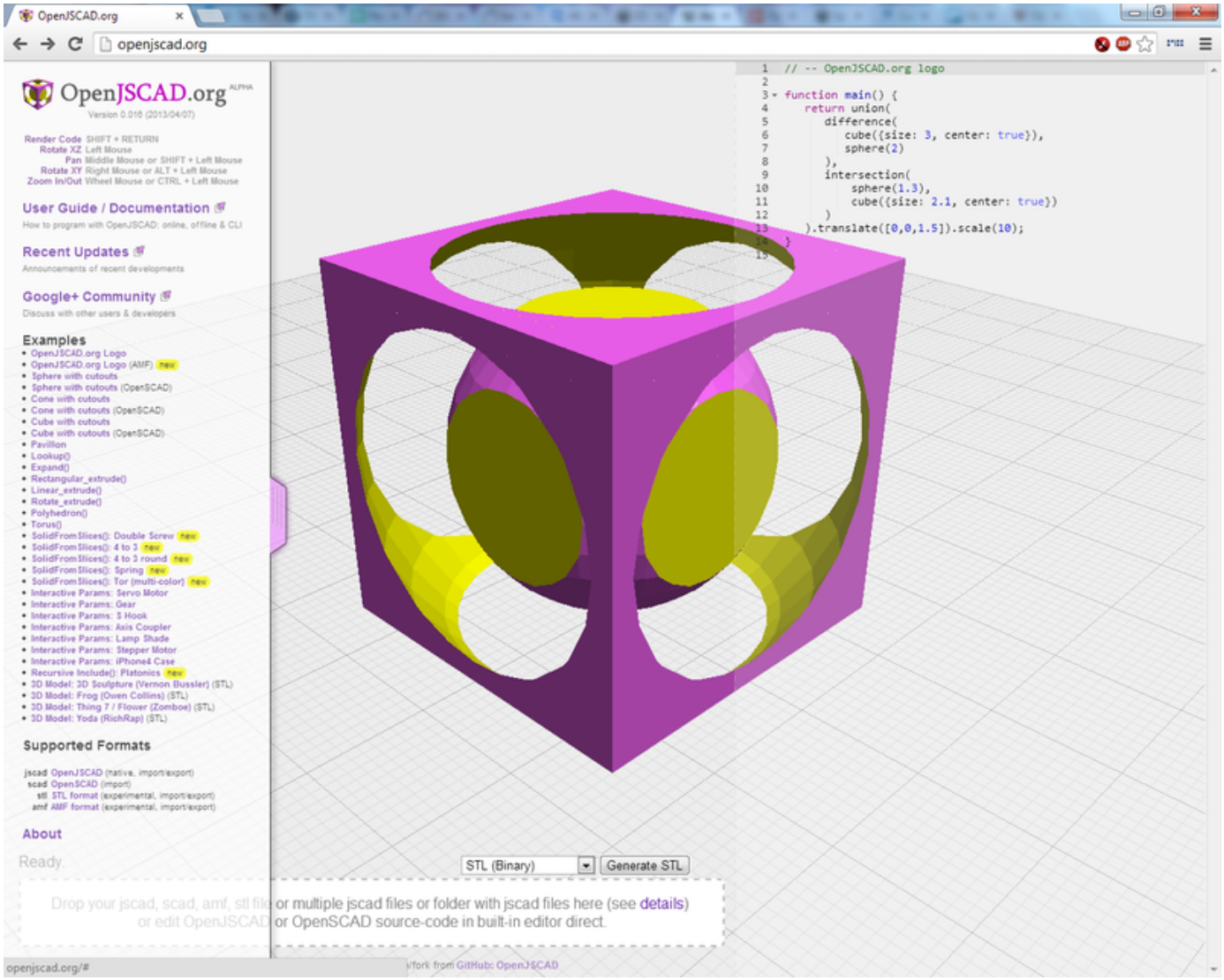
There is also a Quick Reference.

OpenJSCAD User Guide

Using OpenJSCAD via Web Browsers

OpenJSCAD can be used immediately by visiting the project website at www.openjscad.org (http://www.openjscad.org)

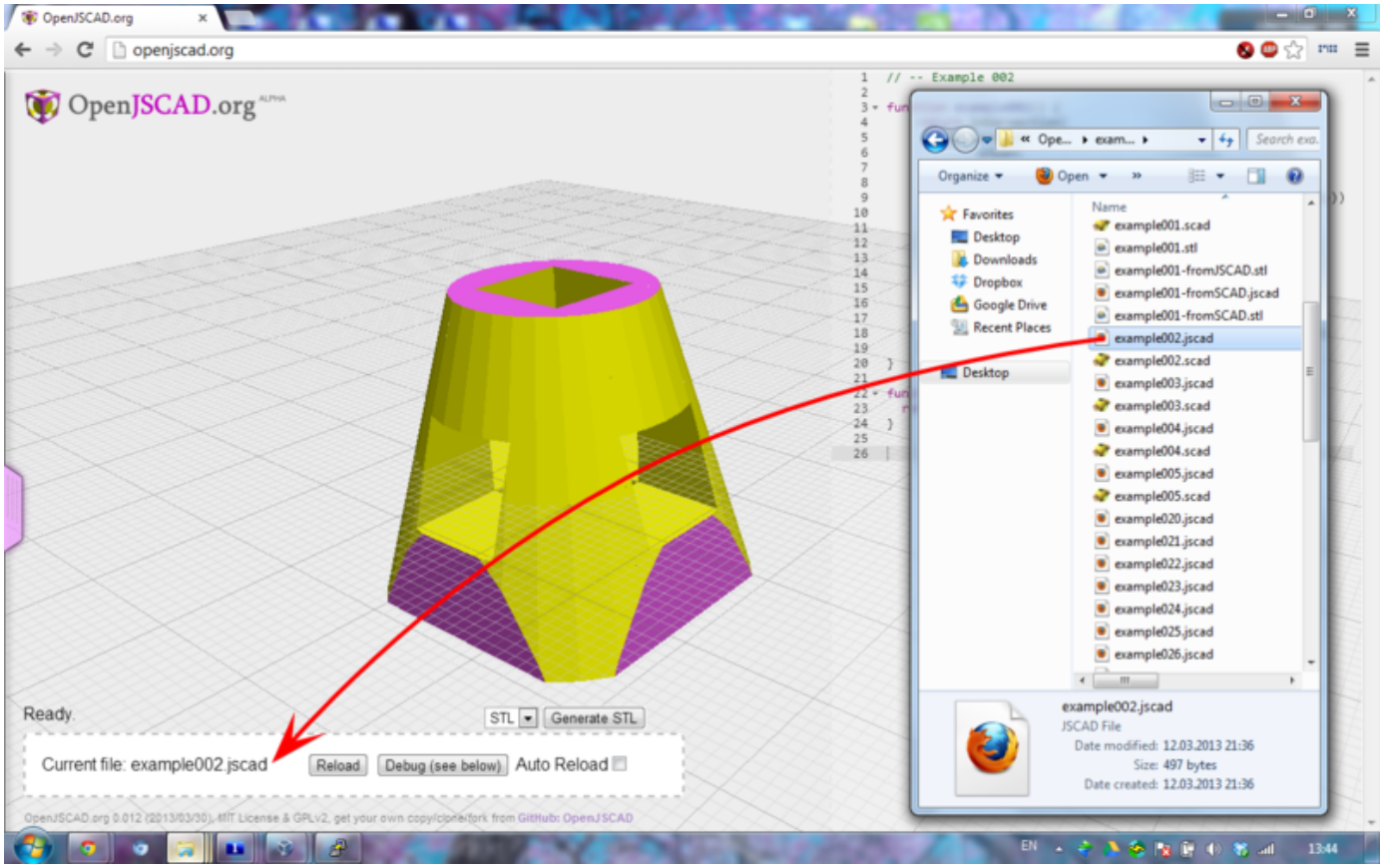
OpenJSCAD.org presents a page which shows a 3D viewer, as well as an editor:



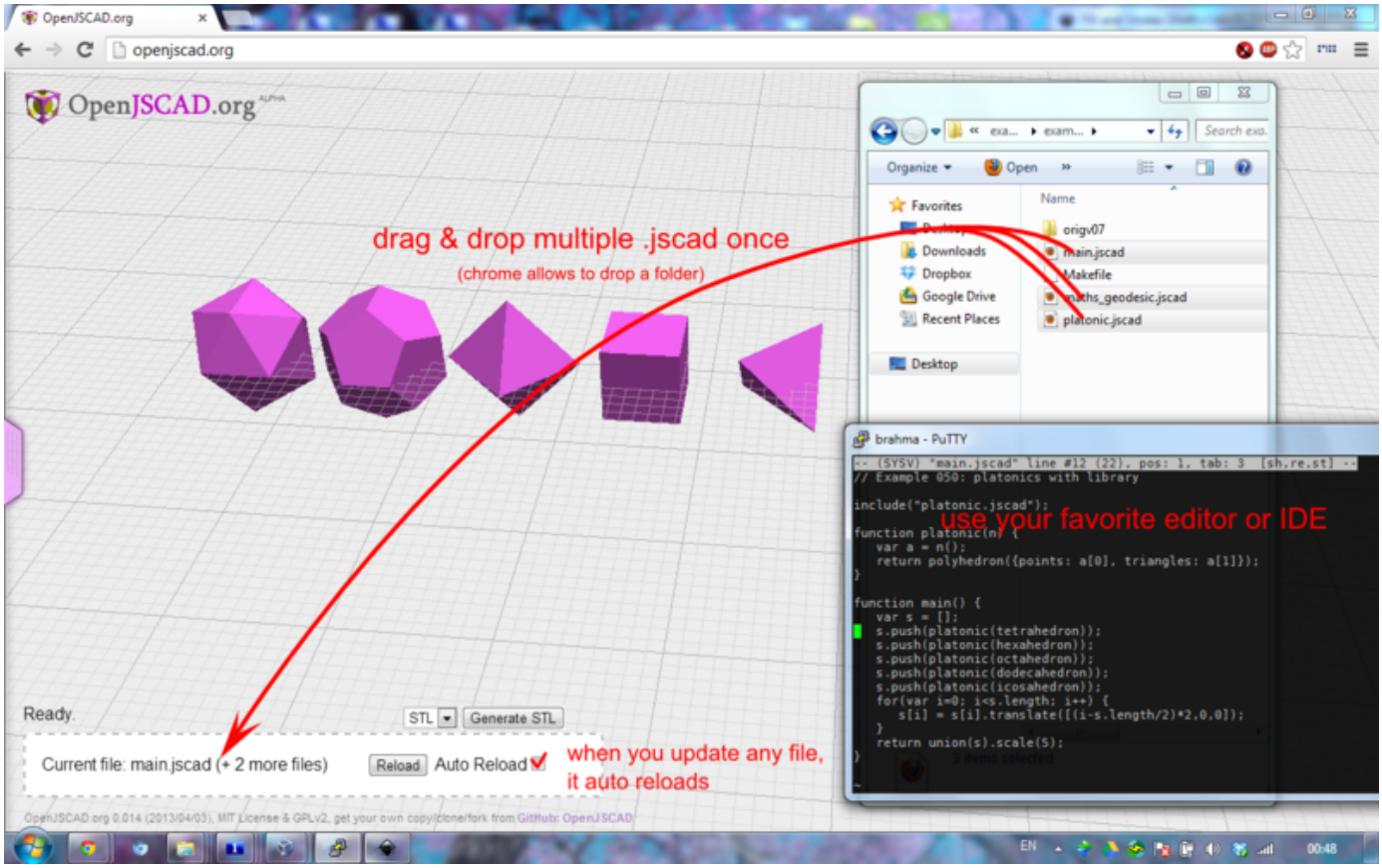
From here, you can

- edit online using the built-in editor, or
- edit off-line via your favorite editor

You can start editing by dragging a file or a folder to the area indicated.



In order to use your favorite editor, make sure Auto Reload is selected. Any changes to the files will be reloaded automatically.



Note: Each browser (Firefox, Chrome, IE, etc) supports slightly different functionality. If you are having problems then please try another browser.

Using OpenJSCAD Offline

OpenJSCAD can be installed locally via GitHub or NPM. You can then use your browser like a locally installed application by opening 'index.html'. To find the location of your index.html you might want to find out where your OpenJSCAD resides. E.g. if OpenJSCAD was installed in /usr/local/lib/node_modules/@jscad/openjscad then

```
cd /usr/local/lib/node_modules/@jscad/openjscad
open index.html
```

will start your webbrowser (Example script works on MacOs)

Local Installation via NPM

OpenJSCAD can be easily installed using Node Version Manager (https://github.com/creationix/nvm) (NVM)

1. Down load and install NVM
2. After installing, type 'nvm install v6
3. Then type 'nvm use v6'

Note: A LTS version of Node.js > 6.0.0 is required. We test OpenJSCAD using both Node.js version 6.x.x & 7.x.x.

Now OpenJSCAD can be downloaded and installed by:

```
npm install -g @jscad/openjscad
```

Local Installation via GitHub

OpenJSCAD can be easily installed from GitHub as well. This is the preferred installation if creating a website for online use.

```
cd base-directory-of-website
git clone https://github.com/jscad/OpenJSCAD.org
cd OpenJSCAD.org
```

NOTE: You might need configuration changes to allow access to the some of the contents (examples etc).

Via Web Browser

Once installing locally, OpenJSCAD can be accessed by opening `index.html` with your browser. You can drag and drop a file or a folder to the browser.

Note: Chrome and other browsers do not support drag and drop when offline.

You can drag and drop any of the examples, as well as other designs or files.

Via Command-Line

```
% cd examples
% openjscad logo.jscad
% openjscad logo.jscad -o test.stl
% openjscad logo.jscad -of stl
```

which creates ``logo.stl`` or ``test.stl``.

Additional you can import OpenSCAD (.scad), STL ASCII or Binary (.stl) and AMF (.amf) files, and create .jscad, .stl (ASCII or Binary), dxf or .amf:

```
% openjscad example001.scad -o example001.jscad
% openjscad example001.scad -o example001.stl
% openjscad frog.stl -o frog.jscad
```

```
% openjscad frog.stl -o frog2.stl             # does actually stl -> jscad -> stl (ascii)
% openjscad example001.jscad -o example001.amf
% openjscad example001.jscad -of amf          # creates example001.amf
% openjscad example001.jscad -of stlb         # creates example001.stl (binary)
% openjscad example001.jscad -of stlb -o test.stl
```

- **-o** stands for output
- **-of** stands for output format (jscad, stl (default), stla, stlb, amf, dxf)

See also how to pass variables from the CLI to main().

Supported Language / File Formats

Currently following languages and file-formats are supported:

Format	Extension	Website	Notes
JSCAD	.jscad	OpenJSCAD.org (http://www.OpenJSCAD.org)	OpenJSCAD is a software for creating solid 3D CAD objects using your browser.
SCAD	.scad	OpenSCAD.org (http://www.OpenSCAD.org)	OpenSCAD is a software for creating solid 3D CAD objects. (Import only)
STL	.stl	Wikipedia: STL (file format)	STL files describe only the surface geometry of three-dimensional objects
AMF	.amf	Wikipedia: AMF (file format)	Additive Manufacturing File Format (very experimental)
DXF	.dxf	DXF (file format) (http://www.autodesk.com/techpubs/autocad/acadr14/dxf)	Drawing Interchange File Format (export only)
X3D	.x3d	Web3D.org (http://www.web3d.org/x3d/what-x3d)	Open standards file format to represent 3D objects using XML.
SVG	.svg	W3C SVG Standard (https://www.w3.org/TR/SVG/)	Scalable Vector Graphics Format

When you drag & drop files, the language or format is set according the file extension (.jscad, etc). When you start to edit directly using your browser, the default language is **JSCAD**.

Sharing Designs via Direct Link for Local or Remote JSCAD, SCAD, STL and AMF

You can share designs with other people by providing creating special URL which combines OpenJSCAD and Design.

Sharing Online Designs

1. http://openjscad.org/#http://openjscad.org/examples/slices/tor.jscad
2. http://openjscad.org/#http://www.thingiverse.com/download:164128 (STL)
3. http://openjscad.org/#http://pastebin.com/raw.php?i=5RbzVguT (SCAD)
4. http://openjscad.org/#http://amf.wikispaces.com/file/view/Rook.amf/268219202/Rook.amf (AMF)

Sharing Local Designs

1. http://openjscad.org/#examples/slices/tor.jscad
2. http://localhost/OpenJSCAD.org/#examples/slices/tor.jscad

OpenJSCAD Programming Guide

In general, designs are written using the JavaScript language. Training and help about JavaScript can be found online.

Creating a new design starts by writing simple scripts which call CSG functions, and other special functions, as provided by **OpenJSCAD**. OpenJSCAD executes the script, and renders the 3D design for viewing.

OpenJSCAD adheres to specific standards for passing parameters to functions. But most parameters are optional as default values are provided.

When 3D vectors are required, parameters can be passed as an array. If a scalar (single value) is passed for a parameter which expects a 3D vector, the scalar is used for the x, y and z values. In other words: radius: 1 will give radius: [1,1,1].

NEED EXAMPLE

Anatomy of a OpenJSCAD Script

An **OpenJSCAD** script must have at least one function defined, the *main()* function, which has to return a **CSG** object, or an array of non-intersecting CSG objects.

```
function main() {
   return union(sphere(), ...);    // an union of objects or
   return [sphere(), ...];        // an array of non-intersecting objects
}
```

or like this:

```
var w = new Array();
function a() {
   w.push( sphere() );
   w.push( cube().translate([2,0,0]) );
}
function main() {
   a();
   return w;
}
```

But this does not work.

```
var w = new Array();
w.push( sphere() );                    // Note: it's not within a function (!!)
w.push( cube().translate([2,0,0]) );

function main() {
    return w;
}
```

Because all CSG creations, like 3D primitives, have to occur within functions which are called by the *main()* function.

3D Primitives

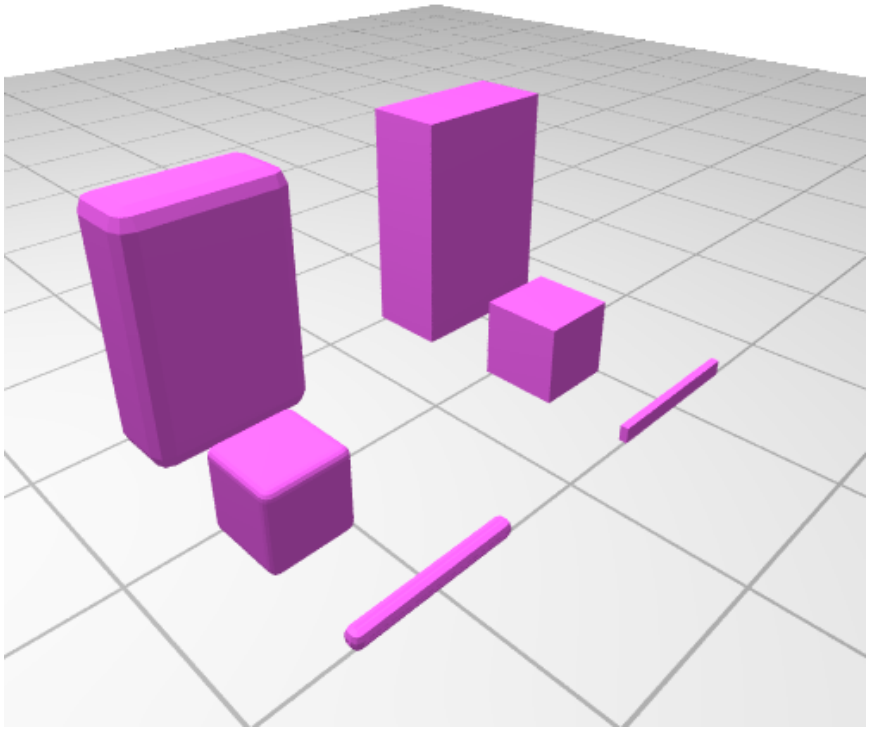
All rounded solids have a *resolution* parameter which controls tesselation. If *resolution* is set to 8, then 8 polygons per 360 degree of revolution are used. Beware that rendering time will increase dramatically when increasing the *resolution*. For a sphere, the number of polygons increases quadratically with the *resolution* used. If the *resolution* parameter is omitted, the following two global defaults are used

- CSG.defaultResolution2D
- CSG.defaultResolution3D

The former is used for 2D curves (circle, cylinder), the latter for 3D curves (sphere, 3D expand).

OpenSCAD like functions support the *fn* parameter, which is the numer of segments to approximate a sphere (default 32, total polygons per sphere fn*fn).

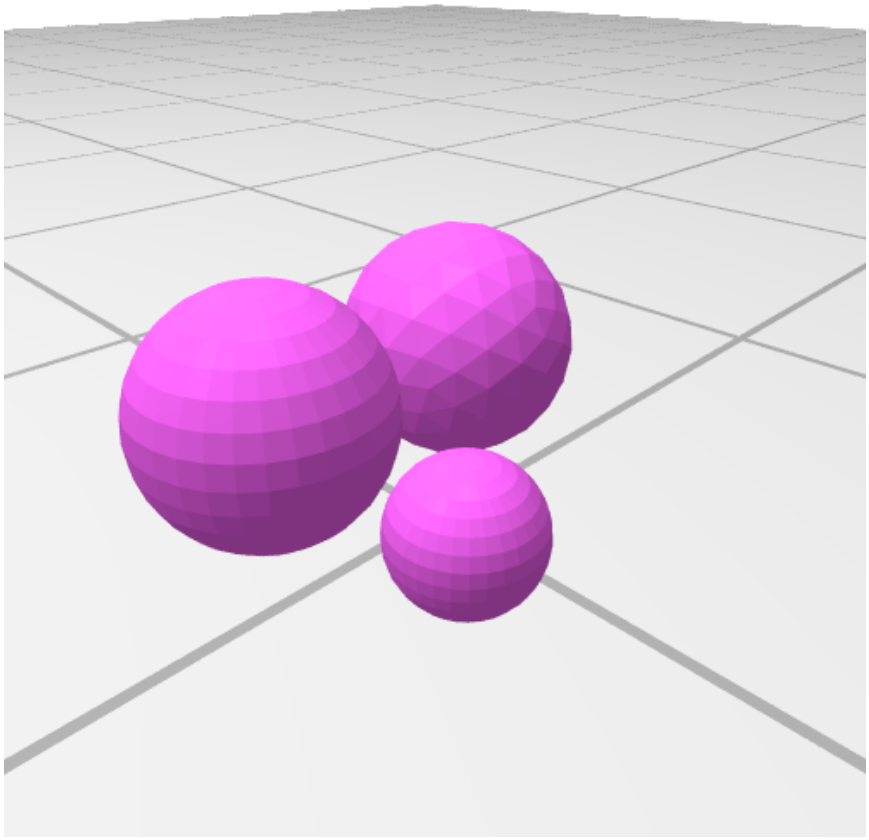
Cube



Cube or rather boxes can be created like this:

```
cube(); // openscad like
cube(1);
cube({size: 1});
cube({size: [1,2,3]});
cube({size: 1, center: true}); // default center:false
cube({size: 1, center: [false,false,false]}); // individual axis center true or false
cube({size: [1,2,3], round: true});

CSG.cube(); // using the CSG primitives
CSG.cube({
    center: [0, 0, 0],
    radius: [1, 1, 1]
});

CSG.cube({ // define two opposite corners
    corner1: [4, 4, 4],
    corner2: [5, 4, 2]
});

CSG.roundedCube({ // rounded cube
    center: [0, 0, 0],
    radius: 1,
    roundradius: 0.9,
    resolution: 8,
});
```

Sphere



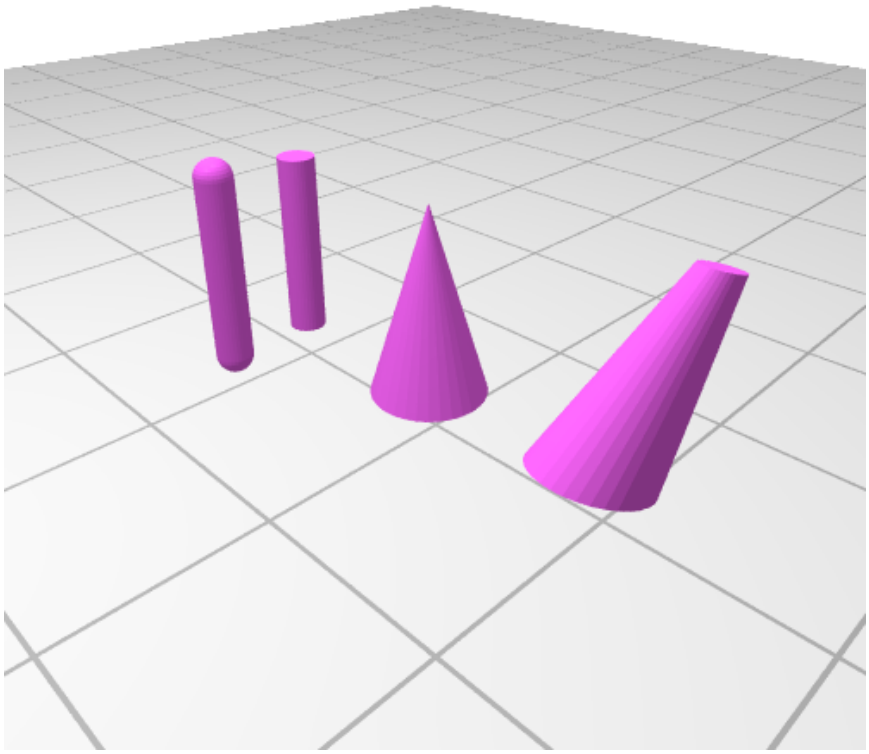
Spheres can be created like this:

```
sphere();                          // openscad like
sphere(1);
sphere({r: 2});                    // Note: center:true is default (unlike other primitives, as OpenSCAD)
sphere({r: 2, center: true});      // Note: OpenSCAD doesn't support center for sphere but we do
sphere({r: 2, center: [false, false, true]}); // individual axis center
sphere({r: 10, fn: 100 });
sphere({r: 10, fn: 100, type: 'geodesic'});  // geodesic approach (icosahedron further triangulated)

CSG.sphere();                      // using the CSG primitives
CSG.sphere({
  center: [0, 0, 0],
  radius: 2,                       // must be scalar
  resolution: 128
});
```

In case of ``type: 'geodesic'`` the fn tries to match the non-geodesic fn, yet, actually changes in steps of 6 (e.g. fn=6..11 is the same), fn = 1 reveals the base form: the icosahedron.

Note: Creating spheres with high resolution and then operating with them, e.g. union(), intersection(), etc, slows down rendering / construction procedure due the large amount of polygons.

Cylinder

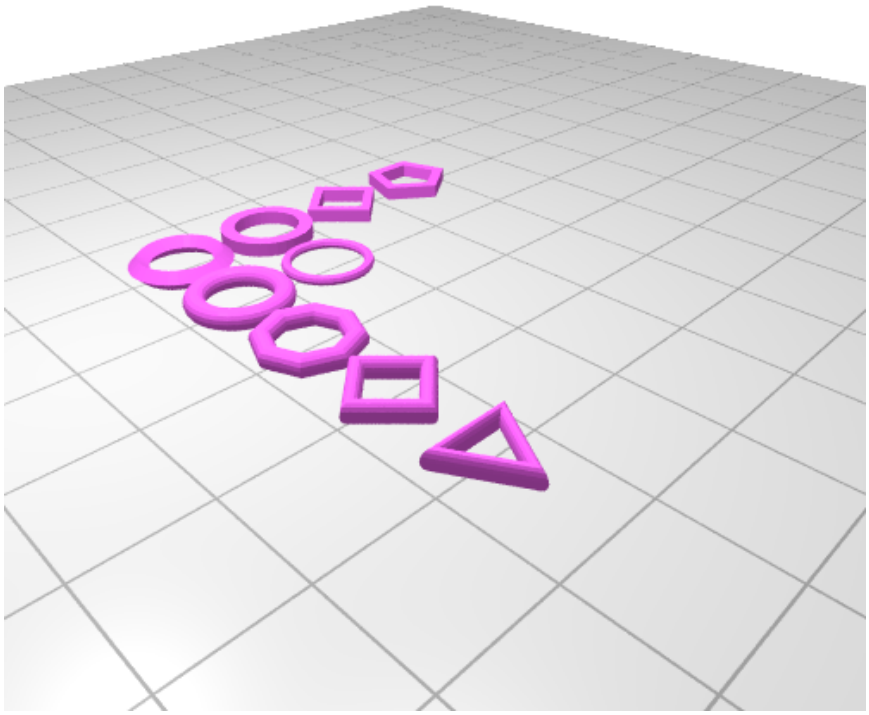


Cylinders and cones can be created like this:

```
cylinder({r: 1, h: 10});                 // openscad like
cylinder({d: 1, h: 10});
cylinder({r: 1, h: 10, center: true});   // default: center:false
cylinder({r: 1, h: 10, center: [true, true, false]});  // individual x,y,z center flags
cylinder({r: 1, h: 10, round: true});
cylinder({r1: 3, r2: 0, h: 10});
cylinder({d1: 1, d2: 0.5, h: 10});
cylinder({start: [0,0,0], end: [0,0,10], r1: 1, r2: 2, fn: 50});

CSG.cylinder({                           //using the CSG primitives
  start: [0, -1, 0],
  end: [0, 1, 0],
  radius: 1,                             // true cylinder
  resolution: 16
});
CSG.cylinder({
  start: [0, -1, 0],
  end: [0, 1, 0],
  radiusStart: 1,                        // start- and end radius defined, partial cones
  radiusEnd: 2,
  resolution: 16
});
CSG.roundedCylinder({                    // and its rounded version
  start: [0, -1, 0],
  end: [0, 1, 0],
  radius: 1,
  resolution: 16
});
```

whereas *fn* is the amount of segments to approximate the circular profile of the cylinder (default 32).

Torus



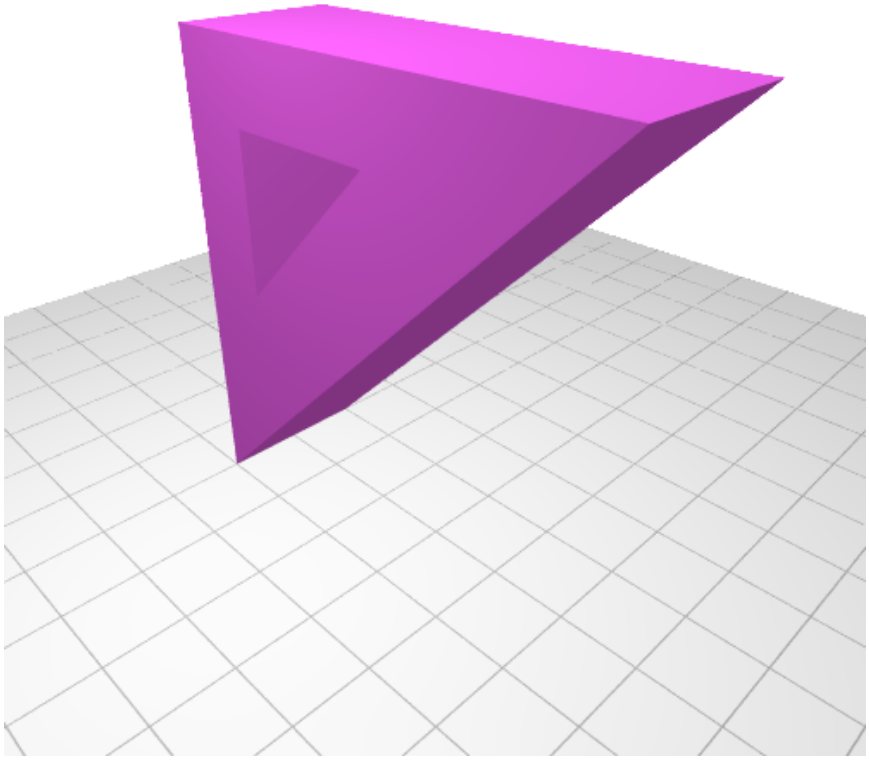
A torus is defined as such:

- ri = inner radius (default: 1),
- ro = outer radius (default: 4),
- fni = inner resolution (default: 16),
- fno = outer resolution (default: 32),
- roti = inner rotation (default: 0)

```
torus();                    // ri = 1, ro = 4;
torus({ ri: 1.5, ro: 3 });
torus({ ri: 0.2 });

torus({ fni:4 });           // make inner circle fn = 4 => square
torus({ fni:4,roti:45 });   // rotate inner circle, so flat is top/bottom
torus({ fni:4,fno:4,roti:45 });
torus({ fni:4,fno:5,roti:45 });
```

Polyhedron

Create a polyhedron with a list of points and a list of triangles or polygons. The point list is all the vertexes of the shape, the triangle list is how the points relates to the surfaces of the polyhedron:



```
polyhedron({      // openscad-like (e.g. pyramid)
  points: [ [10,10,0],[10,-10,0],[-10,-10,0],[-10,10,0], // the four points at base
            [0,0,10] ],                                  // the apex point
  triangles: [ [0,1,4],[1,2,4],[2,3,4],[3,0,4],          // each triangle side
               [1,0,3],[2,1,3] ]                         // two triangles for square base
});
```

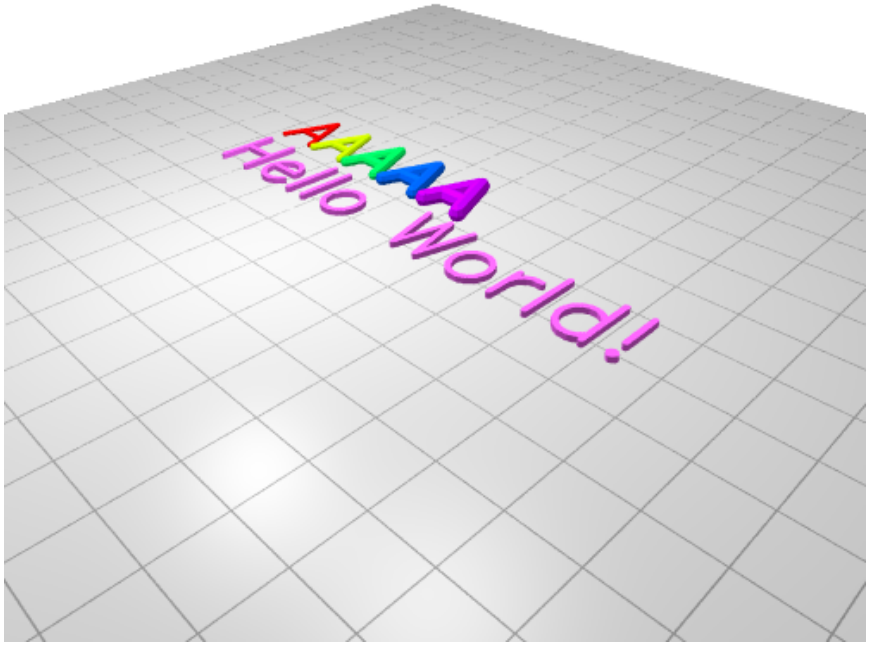
Additionally you can also define `polygons: [ [0,1,4,5], [..] ]` too, not just `triangles:`.

You can also create a polyhedron at a more low-level:

```
var polygons = [];
polygons.push(new CSG.Polygon([
      new CSG.Vertex(new CSG.Vector3D(-5,-5,0)),
      new CSG.Vertex(new CSG.Vector3D(2,2,5)),
      new CSG.Vertex(new CSG.Vector3D(3,3,15))
   ])
);
// add more polygons and finally:
solid = CSG.fromPolygons(polygons);
```

Text

``vector_text(x,y,string)`` and ``vector_char(x,y,char)`` give you line segments of a text or character rendered in vector:



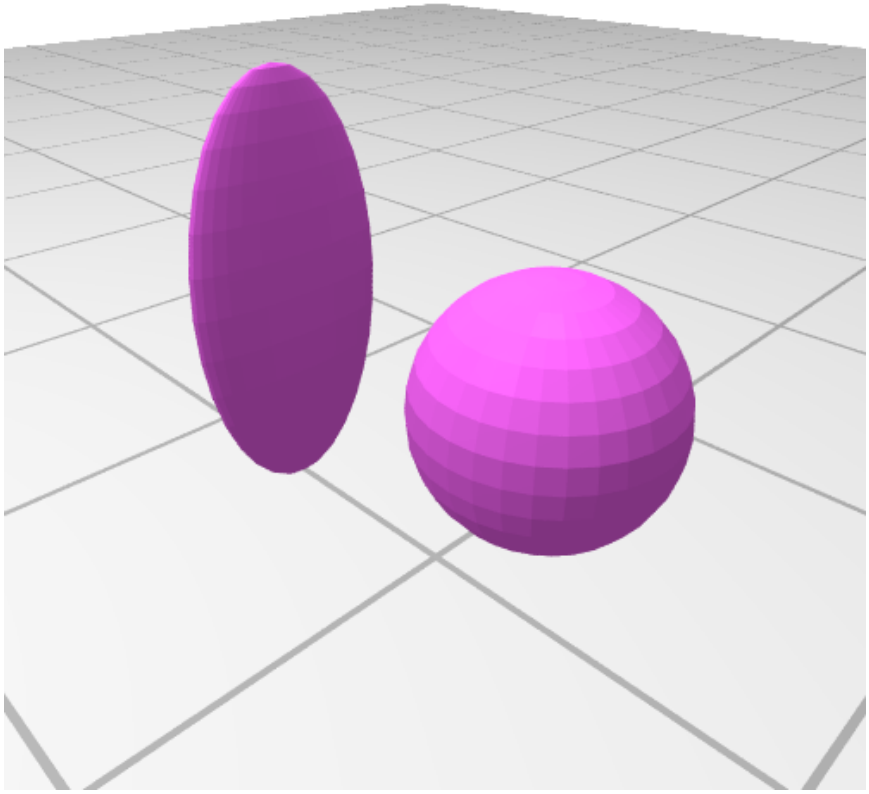

```
var l = vector_text(0,0,"Hello World!");   // l contains a list of polylines to be drawn
var o = [];
l.forEach(function(pl) {                   // pl = polyline (not closed)
   o.push(rectangular_extrude(pl, {w: 2, h: 2}));   // extrude it to 3D
});
return union(o);
```

Also multiple-line with "\n" is supported, for now just left-align is supported. If you want to dive more into the details, you can request a single character:

```
var c = vector_char(x,y,"A");
c.width;    // width of the vector font rendered character
c.segments; // array of segments / polylines
```

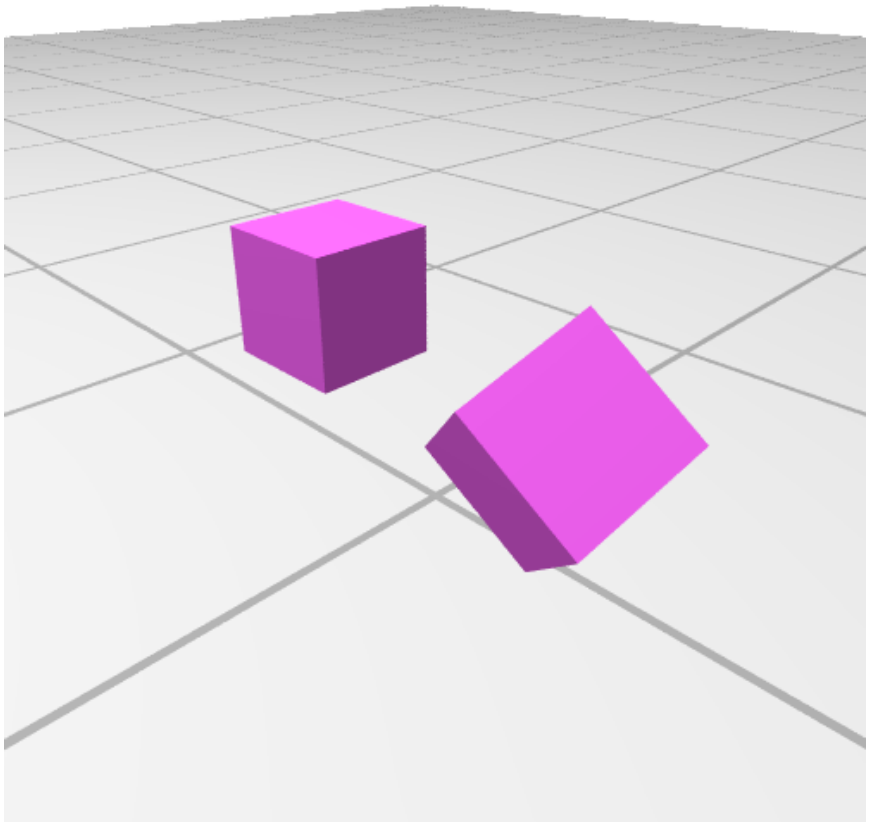
3D Transformations

Scale



```
var obj = sphere(5);
scale(2,obj);          // openscad like
scale([1,2,3],obj);    //       ''

obj.scale([1,2,3]);    // using CSG objects' built in methods
```

Rotate



```
var obj = cube([5,20,5]);
rotate([90,15,30],obj);       // openscad like
rotate(90,[1,0.25,0.5],obj);  //    ''

obj.rotateX(90);              // using CSG objects' built in methods
obj.rotateY(45);
obj.rotateZ(30);

obj.rotate(rotationCenter, rotationAxis, degrees)
obj.rotateEulerAngles(alpha, beta, gamma, position)
```

Translate

```
translate([0,0,10],obj);  // openscad like

obj.translate([0,0,10]);  // using CSG objects' built in methods
```

Center

Centering an object altogether or axis-wise:

```
center(true,cube());               // openscad-like all axis
center([true,true,false],cube());  // openscad-like axis-wise [x,y]

// false = do nothing, true = center axis

cube().center();                // using CSG objects' built in methods
cube().center('x','y');         // using CSG objects' built in method to center axis-wise [x,y]
```

It center() and .center() helps you to compose a symmetric whose complete size you don't know when composing, e.g. from parametric design.

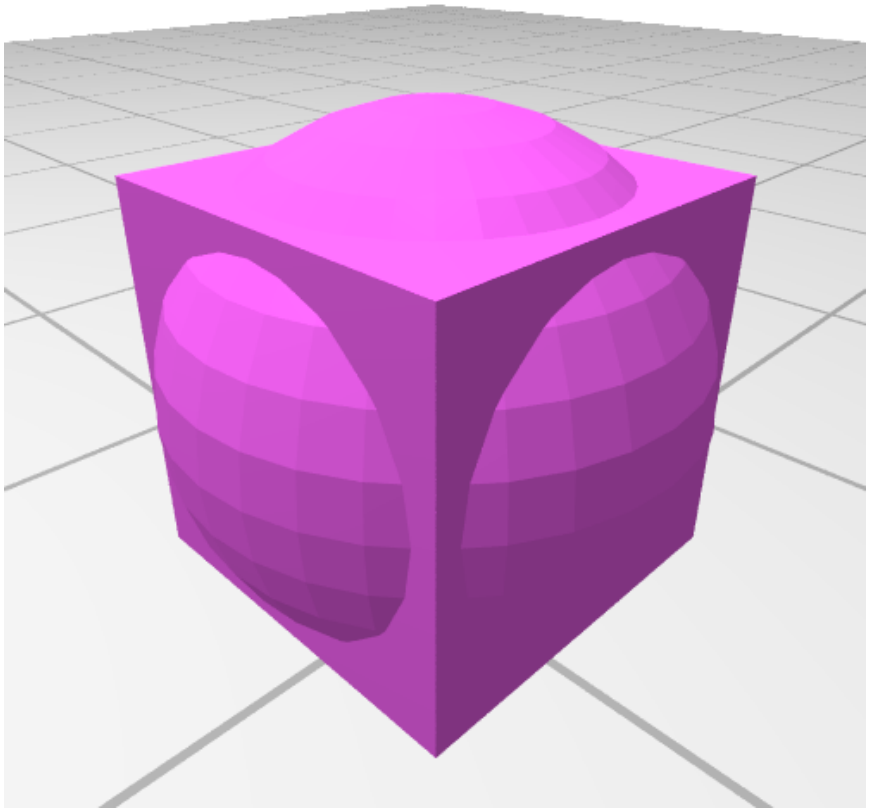
Matrix Operations

```
var m = new CSG.Matrix4x4();
m = m.multiply(CSG.Matrix4x4.rotationX(40));
m = m.multiply(CSG.Matrix4x4.rotationZ(40));
m = m.multiply(CSG.Matrix4x4.translation([-.5, 0, 0]));
m = m.multiply(CSG.Matrix4x4.scaling([1.1, 1.2, 1.3]));

// and apply the transform:
var cube3 = cube().transform(m);
```

Mirror

```
mirror([10,20,90], cube(1)); // openscad like

var cube = CSG.cube().translate([1,0,0]);   // built in method chaining

var cube2 = cube.mirroredX(); // mirrored in the x=0 plane
var cube3 = cube.mirroredY(); // mirrored in the y=0 plane
var cube4 = cube.mirroredZ(); // mirrored in the z=0 plane

// create a plane by specifying 3 points:
var plane = CSG.Plane.fromPoints([5,0,0], [5, 1, 0], [3, 1, 7]);

// and mirror in that plane:
var cube5 = cube.mirrored(plane);
```

Union

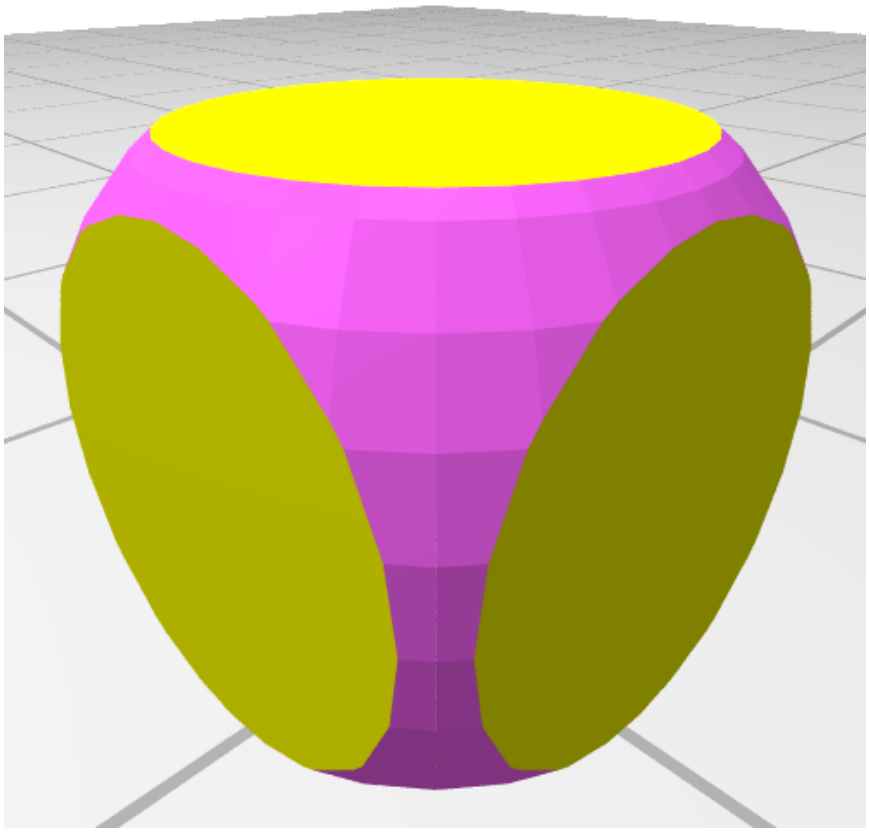


```
union(sphere({r: 1, center:true}),cube({size: 1.5, center:true}));  // openscad like
```

multiple objects can be added, also arrays.

```
sphere({r: 1, center:true}).union(cube({size: 1.5, center:true}));  // using CSG objects' built in methods
```

Intersection



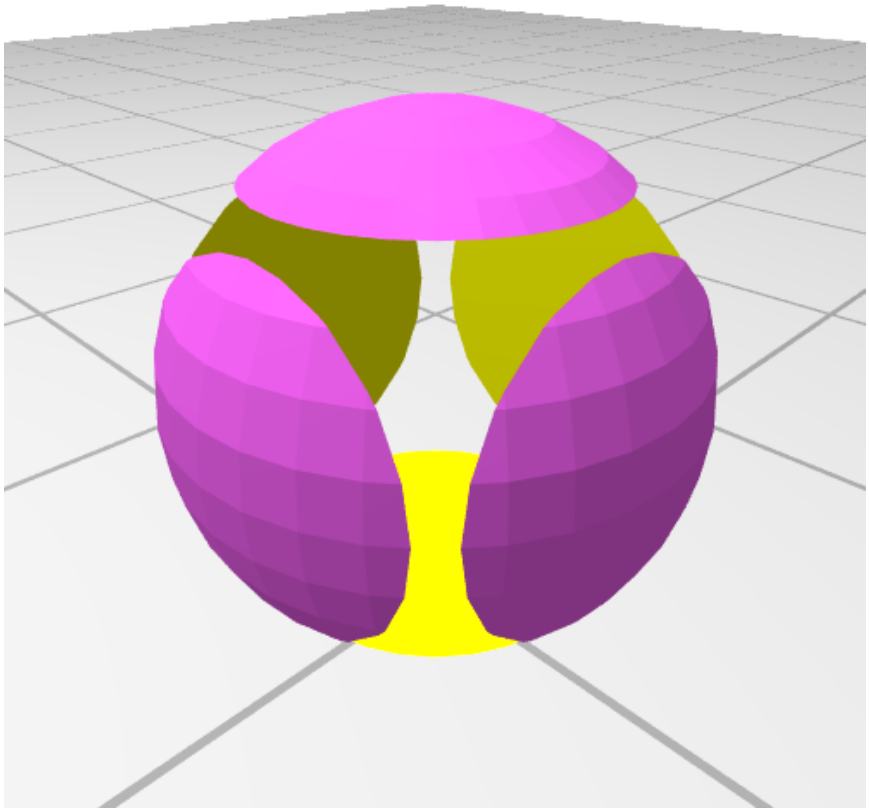
```
intersection(sphere({r: 1, center:true}),cube({size: 1.5, center:true})); // openscad like
```

multiple objects can be intersected, also arrays.

```
sphere({r: 1, center:true}).intersect(cube({size: 1.5, center:true}));   // using CSG objects' built in
methods
```

Note intersection() (openscad like) vs intersect() (function vs CSG objects' built in methods)

Difference (Subtraction)



```
difference(sphere({r: 1, center:true}),cube({size: 1.5, center:true}));    // openscad like
```

multiple objects can be differentiated (subtracted) from the first element, also arrays.

```
sphere({r: 1, center:true}).subtract(cube({size: 1.5, center:true}));      // using CSG objects' built in
methods
```

Note: difference() (openscad like) vs subtract() (method, object-oriented)

2D Primitives

Circle

```
circle();                         // openscad like
circle(1);
circle({r: 2, fn:5});             // fn = number of segments to approximate the circle
circle({r: 3, center: true});     // center: false (default)

CAG.circle({center: [0,0], radius: 3, resolution: 32});   // using CSG objects' built in methods
```

Square / Rectangle

```
square();                                  // openscad like
square(1);                                 // 1x1
square([2,3]);                             // 2x3
square({size: [2,4], center: true});       // 2x4, center: false (default)

CAG.rectangle({center: [0,0], radius: [w/2, h/2]});   // CAG built ins, where w or h = side-length of square
CAG.roundedRectangle({center: [0,0], radius: [w/2, h/2], roundradius: 1, resolution: 4});
```

Polygon

```
polygon([ [0,0],[3,0],[3,3] ]);                // openscad like
polygon({ points: [ [0,0],[3,0],[3,3] ] });
polygon({ points: [ [0,0],[3,0],[3,3],[0,6] ], paths: [ [0,1,2],[1,2,3] ] }); // multiple paths not yet
implemented

var shape1 = CAG.fromPoints([ [0,0],[5,0],[3,5],[0,5] ]);    // CAG built ins
```

2D Transformations

```
translate([2,2], circle(1));      // openscad like
rotate([0,0,90], square());       //     ''
shape = center(true, shape());    // center both axis
shape = center([true,false], shape()); // center axis-wise [x]

shape = shape.translate([-2, -2]);   // object methods
shape = shape.rotateZ(20);
shape = shape.scale([0.7, 0.9]);
shape = shape.center();          // center both axis
scape = shape.center('x');       // center axis-wise [x]
```

2D Paths

A path is simply a series of points, connected by lines. A path can be open or closed (an additional line is drawn between the first and last point). 2D paths are supported through the CSG.Path2D class. The difference between a 2D Path and a 2D CAG is that a path is a 'thin' line, whereas a CAG is an enclosed area.

Paths can be constructed either by giving the constructor an array of 2D coordinates, or through the various CSG.Path2D member functions, which include:

- arc(endpoint, options): return a circular or ellipsoid curved path (see example below for usage).
- appendPoint([x,y]): create & return a new Path2D containing the callee's points followed by the given point.
- appendPoints([[x,y],...]): create & return a new Path2D containing the callee's points followed by the given points. [Note: as of 2016/08/13, this method also modifies the callee; this is probably a bug and might be changed in the future; see issue:165 (https://github.com/Spiritdude/OpenJSCAD.org/issues/165)]
- appendBezier([[x,y],...], options): create & return a new Path2D containing the callee's points followed by a Bezier curve ending at the last point given; all but the last point given are the control points of the Bezier; a null initial control point means use the last two points of the callee as control points for the new Bezier curve. options can specify {resolution: <NN>}.

Paths can be concatenated with .concat(), the result is a new path.

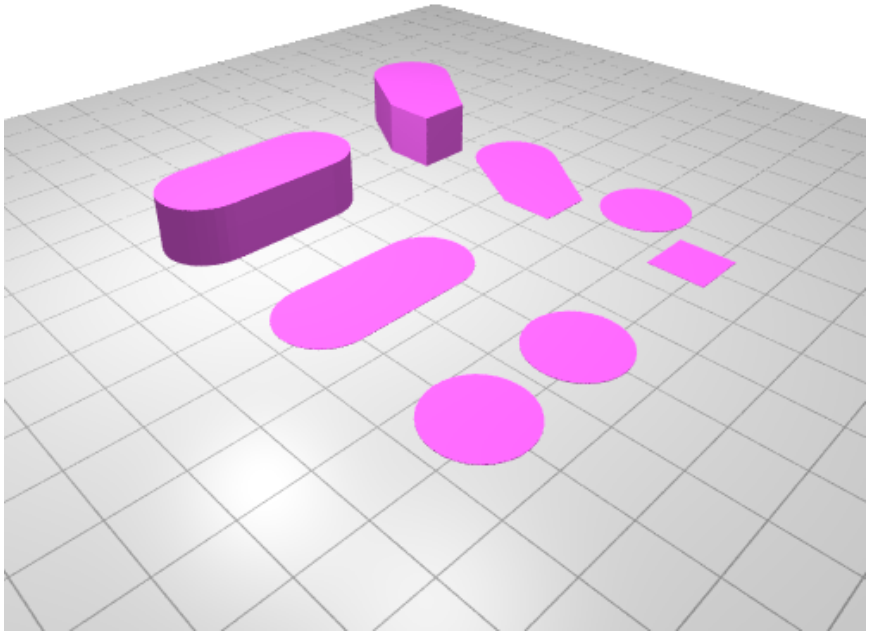
A path can be converted to a CAG in two ways:

- expandToCAG(pathradius, resolution) traces the path with a circle, in effect making the path's line segments thick.
- innerToCAG() creates a CAG bounded by the path. The path should be a closed path.

Creating a 3D solid is currently supported by the rectangularExtrude() function. This creates a 3D shape by following the path with a 2D rectangle (upright, perpendicular to the path direction):

```
var path = new CSG.Path2D([ [10,10], [-10,10] ], /* closed = */ false);
var anotherpath = new CSG.Path2D([ [-10,-10] ]);
path = path.concat(anotherpath);
path = path.appendPoint([10,-10]);
path = path.close(); // close the path

// of course we could simply have done:
// var path = new CSG.Path2D([ [10,10], [-10,10], [-10,-10], [10,-10] ], /* closed = */ true);

// We can make arcs and circles:
var curvedpath = CSG.Path2D.arc({
  center: [0,0,0],
  radius: 10,
  startangle: 0,
  endangle: 180,
  resolution: 16,
});
```

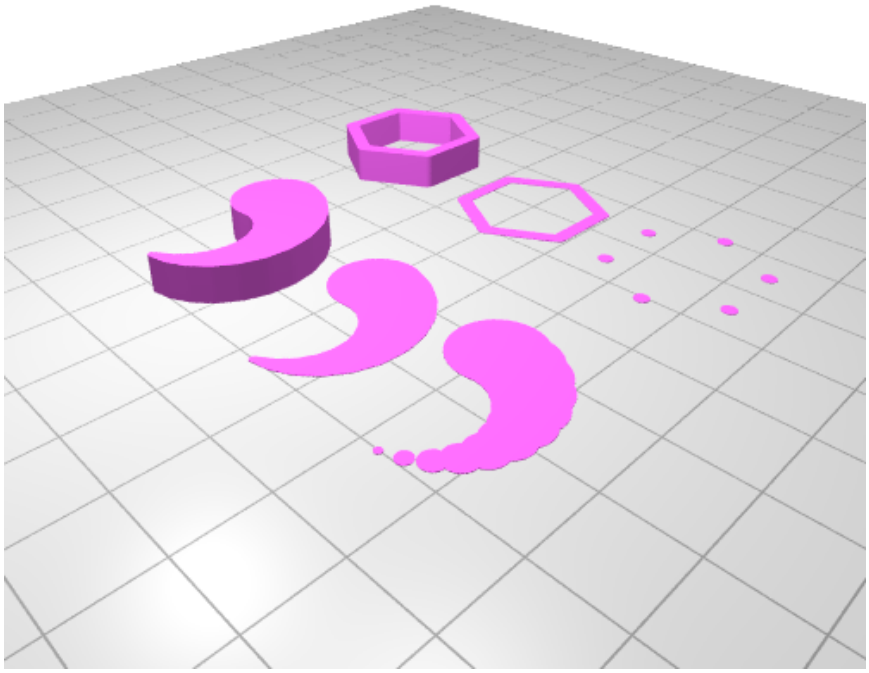
Hull



You can convex hull multiple 2D polygons (e.g. circle(), square(), polygon()) together.

```
var h = hull( square(10),circle(10).translate([10,10,0]) );

linear_extrude({ height: 10 }, h);
```

Chain Hull

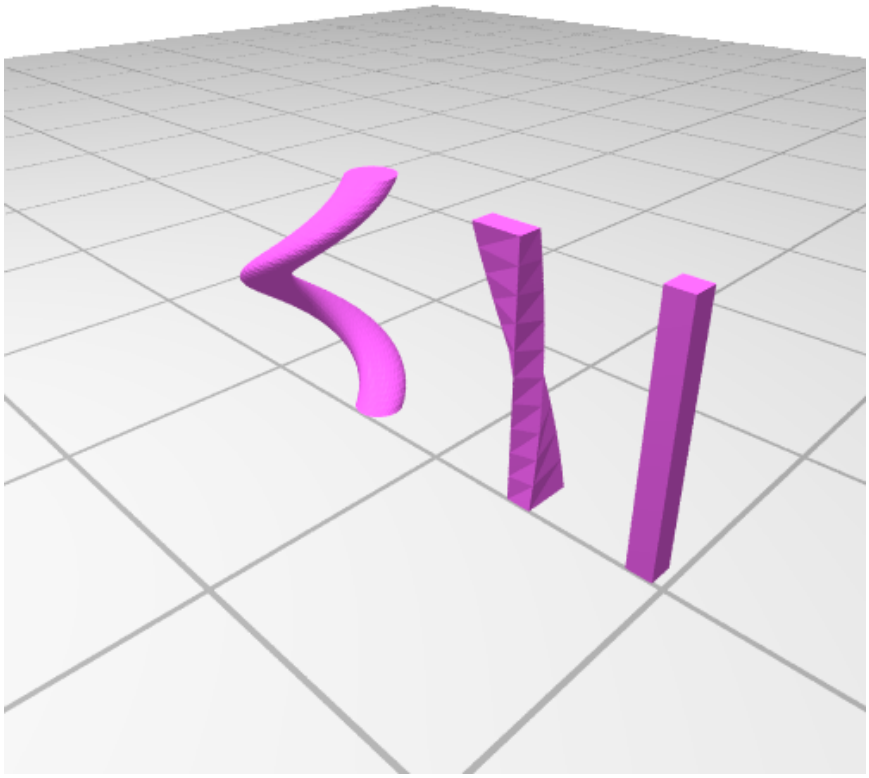


Chained hulling is a variant of hull on multiple 2D forms, essentially sequential hulling and then union those, based on an idea by Whosa whatsis (https://plus.google.com/u/0/105535247347788377245/posts/aZGXKFX1ACN):

```
chain_hull(
    circle(), circle().translate([2,0,0]), ... );   // list of CAG/2D forms

var a = [];
a.push(circle());
chain_hull( a );                        // array of CAG/2D forms

chain_hull({closed: true},              // default is false
   [circle(),circle().translate([2,0,0]),circle().translate([2,2,0])]);
   // notice that with parameter {closed:true}, hull_chain requires an array
```

Extruding / Extrusion

Linear Extrude



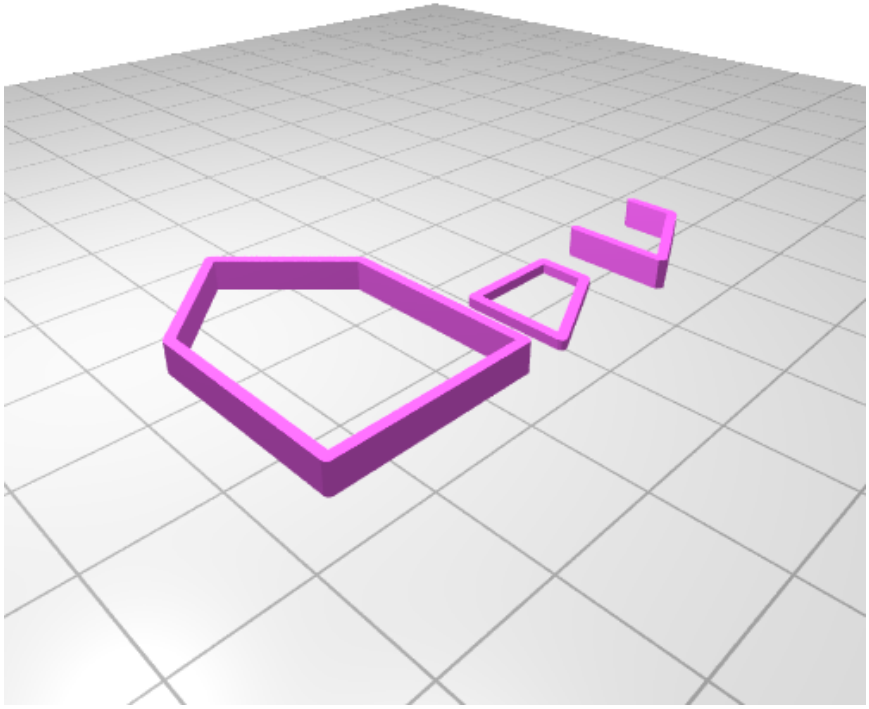
Extruding 2D shapes into 3D, given height, twist (degrees), and slices (if twist is made):

```
// openscad like
linear_extrude({ height: 10 }, square());
linear_extrude({ height: 10, twist: 90 }, square([1,2]));
linear_extrude({ height: 10, twist: 360, slices: 50}, circle().translate([1,0,0]) );

linear_extrude({ height: 10, center: true, twist: 360, slices: 50}, translate([2,0,0], square([1,2])) );
linear_extrude({ height: 10, center: true, twist: 360, slices: 50}, square([1,2]).translate([2,0,0]) );
```

Linear extrusion of 2D shape, with optional twist. The 2d shape is placed in in z=0 plane and extruded into direction <offset> (a CSG.Vector3D). The final face is rotated <twistangle> degrees. Rotation is done around the origin of the 2d shape (i.e. x=0, y=0) twiststeps determines the resolution of the twist (should be >= 1), returns a CSG object:

```
// CAG build in method
var c = CAG.circle({radius: 3});
extruded = c.extrude({offset: [0,0,10], twistangle: 360, twiststeps: 100});
```

Rectangular Extrude



Extrude the path by following it with a rectangle (upright, perpendicular to the path direction), returns a CSG solid.

Simplified (openscad like, even though OpenSCAD doesn't provide this) via rectangular_extrude(), where as

- w: width (default: 1),
- h: height (default: 1),
- fn: resolution (default: 8), and
- closed: whether path is closed or not (default: false)

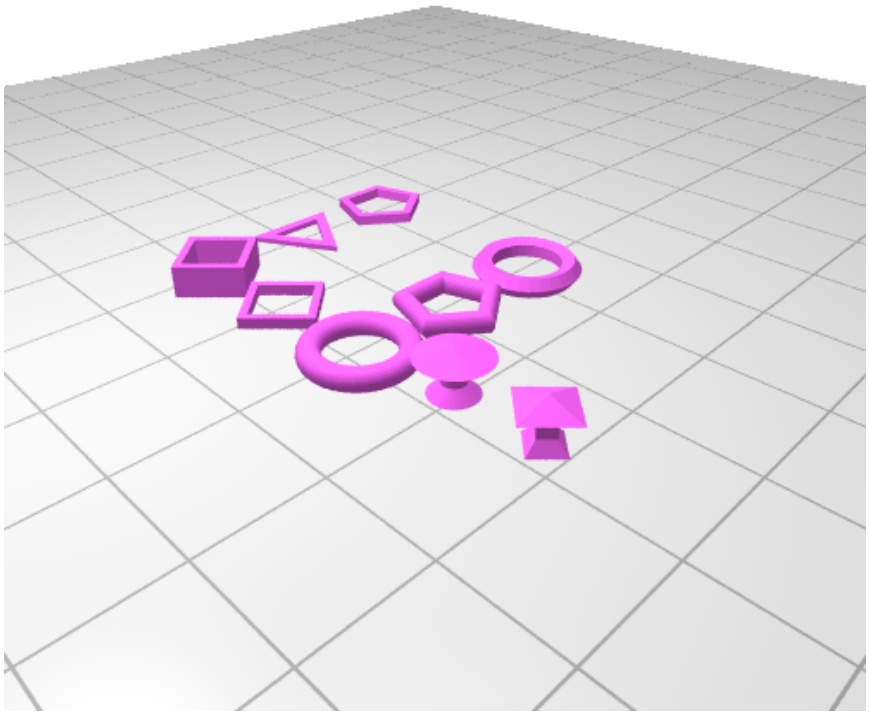
```
rectangular_extrude([ [10,10], [-10,10], [-20,0], [-10,-10], [10,-10] ],  // path is an array of 2d coords
    {w: 1, h: 3, closed: true});
```

or more low-level via rectangularExtrude(), with following unnamed variables:

1. width of the extrusion, in the z=0 plane
2. height of the extrusion in the z direction
3. resolution, number of segments per 360 degrees for the curve in a corner
4. roundEnds: if true, the ends of the polygon will be rounded, otherwise they will be flat

```
// first creating a 2D path, and then extrude it
var path = new CSG.Path2D([ [10,10], [-10,10], [-20,0], [-10,-10], [10,-10] ], /*closed=*/true);
var csg = path.rectangularExtrude(3, 4, 16, true);   // w, h, resolution, roundEnds
return csg;
```

Rotate Extrude



Additional also rotate_extrude() is available:

```
// openscad-like
rotate_extrude( translate([4,0,0], circle({r: 1, fn: 30, center: true}) ) );

// using CSG objects' built in methods to translate
rotate_extrude({fn:4}, square({size: [1,1], center: true}).translate([4,0,0]) );

rotate_extrude( polygon({points:[ [0,0],[2,1],[1,2],[1,3],[3,4],[0,5] ]}) );
rotate_extrude({fn:4}, polygon({points:[ [0,0],[2,1],[1,2],[1,3],[3,4],[0,5] ]}) );
```

You can essentially extrude any 2D polygon (circle, square or polygon).

Properties

The 'property' property of a solid can be used to store metadata for the object, for example the coordinate of a specific point of interest of the solid. Whenever the object is transformed (i.e. rotated, scaled or translated), the properties are transformed with it. So the property will keep pointing to the same point of interest even after several transformations have been applied to the solid.

Properties can have any type, but only the properties of classes supporting a 'transform' method will actually be transformed. This includes CSG.Vector3D, CSG.Plane and CSG.Connector. In particular CSG.Connector properties (see below) can be very useful: these can be used to attach a solid to another solid at a predetermined location regardless of the current orientation.

It's even possible to include a CSG solid as a property of another solid. This could be used for example to define the cutout cylinders to create matching screw holes for an object. Those 'solid properties' get the same transformations as the owning solid but they will not be visible in the result of CSG operations such as union().

Other kind of properties (for example, strings) will still be included in the properties of the transformed solid, but the properties will not get any transformation when the owning solid is transformed.

All primitive solids have some predefined properties, such as the center point of a sphere (TODO: document).

The solid resulting from CSG operations (union(), subtract(), intersect()) will get the merged properties of both source solids. If identically named properties exist, only one of them will be kept.

```
var cube = CSG.cube({radius: 1.0});
cube.properties.aCorner = new CSG.Vector3D([1, 1, 1]);
cube = cube.translate([5, 0, 0]);
cube = cube.scale(2);
// cube.properties.aCorner will now point to [12, 2, 2],
// which is still the same corner point

// Properties can be stored in arrays; all properties in the array
// will be transformed if the solid is transformed:
cube.properties.otherCorners = [
  new CSG.Vector3D([-1, 1, 1]),
  new CSG.Vector3D([-1, -1, 1])
];

// and we can create sub-property objects; these must be of the
// CSG.Properties class. All sub properties will be transformed with
// the solid:
cube.properties.myProperties = new CSG.Properties();
cube.properties.myProperties.someProperty = new CSG.Vector3D([-1, -1, -1]);
```

Connectors

The CSG.Connector class is intended to facilitate attaching two solids to each other at a predetermined location and orientation. For example suppose we have a CSG solid depicting a servo motor and a solid of a servo arm: by defining a Connector property for each of them, we can easily attach the servo arm to the servo motor at the correct position (i.e. the motor shaft) and orientation (i.e. arm perpendicular to the shaft) even if we don't know their current position and orientation in 3D space.

In other words Connector give us the freedom to rotate and translate objects at will without the need to keep track of their positions and boundaries. And if a third party library exposes connectors for its solids, the user of the library does not have to know the actual dimensions or shapes, only the names of the connector properties.

A CSG.Connector consist of 3 properties:

- point: a CSG.Vector3D defining the connection point in 3D space
- axis: a CSG.Vector3D defining the direction vector of the connection (in the case of the servo motor example it would point in the direction of the shaft)
- normal: a CSG.Vector3D direction vector somewhat perpendicular to axis; this defines the "12 o'clock" orientation of the connection.

When connecting two connectors, the solid is transformed such that the **point** properties will be identical, the **axis** properties will have the same direction (or opposite direction if mirror == true), and the **normal**s match as much as possible.

Connectors can be connected by means of two methods: A CSG solid's *connectTo()* function transforms a solid such that two connectors become connected. Alternatively we can use a connector's *getTransformationTo()* method to obtain a transformation matrix which would connect the connectors. This can be used if we need to apply the same transform to multiple solids.

```
var cube1 = CSG.cube({radius: 10});
var cube2 = CSG.cube({radius: 4});

// define a connector on the center of one face of cube1
// The connector's axis points outwards and its normal points
// towards the positive z axis:
cube1.properties.myConnector = new CSG.Connector([10, 0, 0], [1, 0, 0], [0, 0, 1]);

// define a similar connector for cube 2:
cube2.properties.myConnector = new CSG.Connector([0, -4, 0], [0, -1, 0], [0, 0, 1]);

// do some random transformations on cube 1:
cube1 = cube1.rotateX(30);
cube1 = cube1.translate([3.1, 2, 0]);

// Now attach cube2 to cube 1:
cube2 = cube2.connectTo(
  cube2.properties.myConnector,
  cube1.properties.myConnector,
  true,   // mirror
```

```
  0       // normalrotation
);

// Or alternatively:
var matrix = cube2.properties.myConnector.getTransformationTo(
  cube1.properties.myConnector,
  true,   // mirror
  0       // normalrotation
);
cube2 = cube2.transform(matrix);

var result = cube2.union(cube1);
```

Bounds & Surface Laying

The *getBounds()* function can be used to retrieve the bounding box of an object, returning an array with two CSG.Vector3Ds specifying the minimum X,Y,Z coordinates and the maximum X,Y,Z coordinates.

The *lieFlat()* function lays an object onto the Z surface, in such a way that the Z-height is minimized and the object is centered around the Z axis. This can be useful for CNC milling, allowing an object to be transform into the space of the stock material during milling. Or for 3D printing: it is laid in such a way that it can be printed with minimal number of layers. Instead of the *lieFlat()* function, the *getTransformationToFlatLying()* function can be used, which returns a CSG.Matrix4x4 for the transformation.

```
var cube1 = CSG.cube({radius: 10});
var cube2 = CSG.cube({radius: 5});

// get the right bound of cube1 and the left bound of cube2:
var deltax = cube1.getBounds()[1].x - cube2.getBounds()[0].x;

// align cube2 so it touches cube1:
cube2  = cube2.translate([deltax, 0, 0]);

var cube3 = CSG.cube({radius: [100,120,10]});
// do some random transformations:
cube3 = cube3.rotateZ(31).rotateX(50).translate([30,50,20]);
// now place onto the z=0 plane:
cube3  = cube3.lieFlat();

// or instead we could have used:
var transformation = cube3.getTransformationToFlatLying();
cube3 = cube3.transform(transformation);

return cube3;
```

Colors

OpenSCAD-like:

```
color([r,g,b], object, object2 ...); // for example, color([1,1,0],sphere());
color([r,g,b], array);
color([r,g,b,a], object, object2 ...);
color([r,g,b,a], array);
color(name, object, object2 ...); // for example, color('red',sphere());
color(name, a, object, object2 ...); // for example, color('red',0.5, sphere());
color(name, array);
color(name, a, array);
```

Whereas the named colors are case-insensitive, e.g. 'RED' is the same as 'red'.

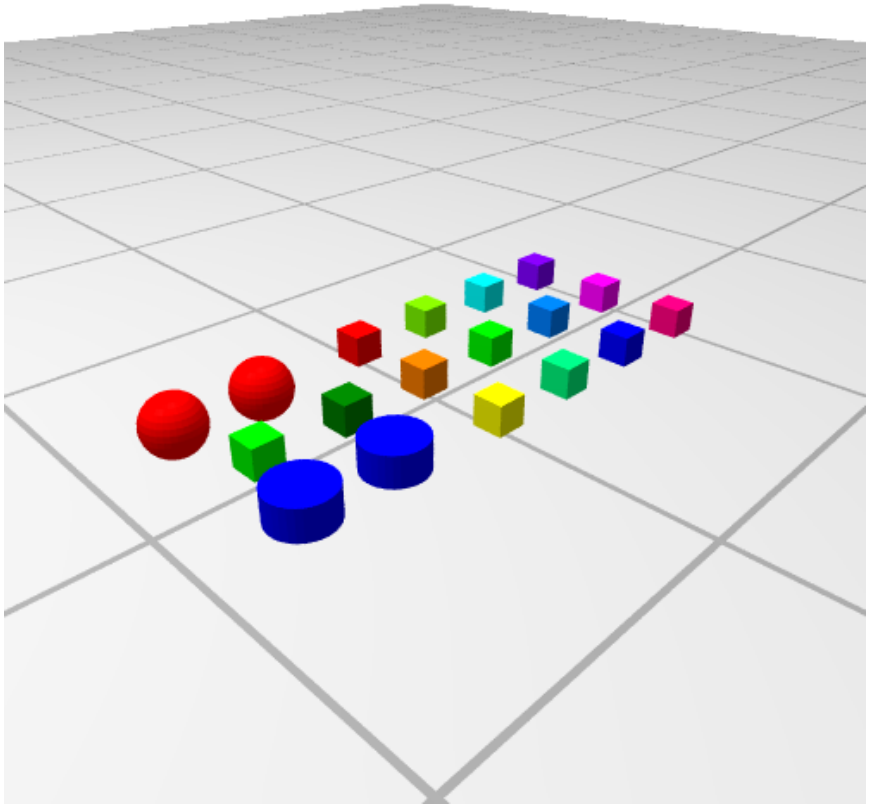
using CSG objects' built in methods (r, g, b must be between 0 and 1, not 0 and 255):

```
object.setColor([r,g,b]);
object.setColor([r,g,b,a]);
object.setColor(r,g,b);
object.setColor(r,g,b,a);
object.setColor(css2rgb('dodgerblue'));
```

Examples:

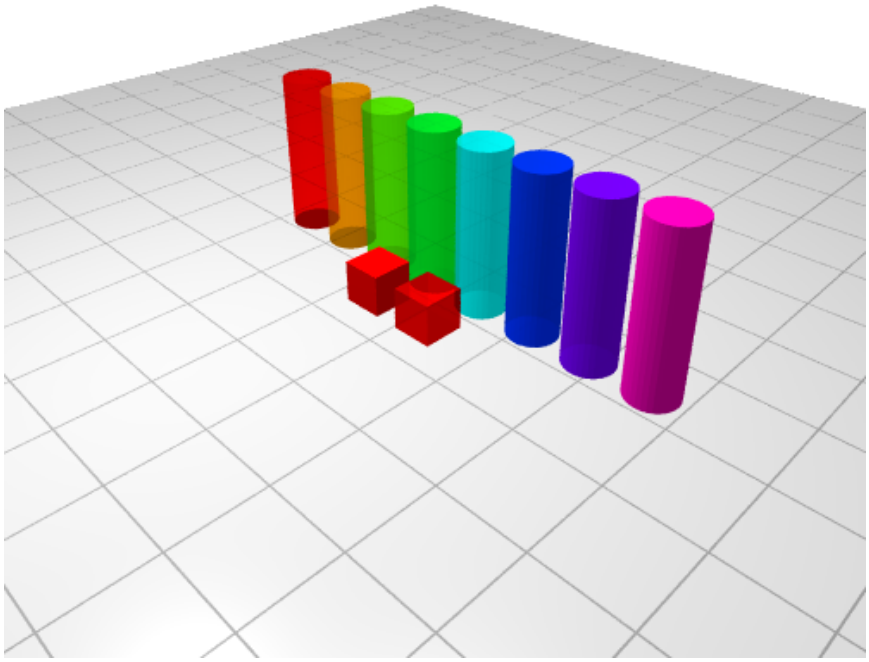
```
color([1,0.5,0.3],sphere(1));                       // openscad like
color([1,0.5,0.3],sphere(1),cube(2));
color("Red",sphere(),cube().translate([2,0,0]));    // named color (case-insensitive)

sphere().setColor(1,0.5,0.3);                       // built in methods
sphere().setColor([1,0.5,0.3]);
```

See the Extended Color Keywords (https://www.w3.org/TR/css3-color/#svg-color) for all available colors.



Code excerpt:

```
o.push( color([1,0,0],sphere()) );
o.push( color([0,1,0],cube()) );
o.push( color([0,0,1],cylinder()) );

o.push( color("red",sphere()) );
o.push( color("green", cube()) );
o.push( color("blue", cylinder()) );

for(var i=0; i&lt;1; i+=1/12) {
   o.push( cube().setColor(hsl2rgb(i,1,0.5)) );
}
```



Code:

```
function main() {
   var o = [];
   for(var i=0; i&lt;8; i++) {
      o.push(cylinder({r:3,h:20}).
         setColor(
            hsl2rgb(i/8,1,0.5).  // hsl to rgb, creating rainbow [r,g,b]
            concat(1/8+i/8)      // and add to alpha to make it [r,g,b,a]
         ).translate([(i-3)*7.5,0,0])
      );
   }
   o.push(color("red",cube(5)).translate([-4,-10,0]));
   o.push(color("red",0.5,cube(5)).translate([4,-10,0]));
   return o;
}
```

Note: There are some OpenGL Transparency Limitations, e.g. and depending on the order of colors, you might not see through otherwise partially transparent objects.

Color Spacing Conversion

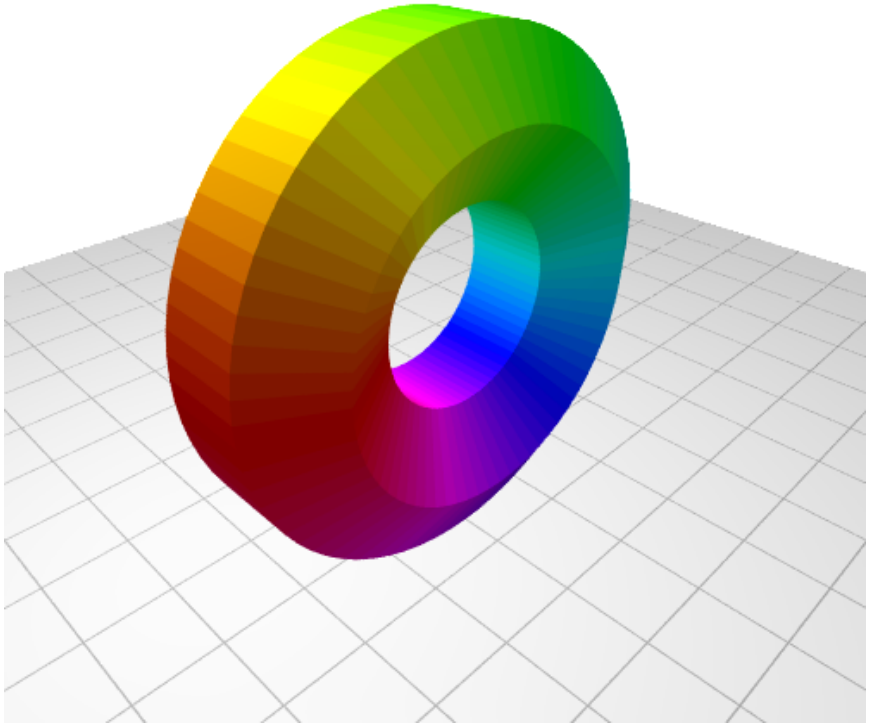
Following functions to convert between color spaces:

```
var hsl = rgb2hsl(r,g,b); // or rgb2hsl([r,g,b]);
var rgb = hsl2rgb(h,s,l); // or hsl2rgb([h,s,l]);
var hsv = rgb2hsv(r,g,b); // or rgb2hsv([r,g,b]);
var rgb = hsv2rgb(h,s,v); // or hsv2rgb([h,s,v]);
```

whereas

- r,g,b (red, green, blue)
- h,s,l (hue, saturation, lightness)
- h,s,v (hue, saturation, value)

E.g. to create a rainbow, t = 0..1 and .setColor(hsl2rgb(t,1,0.5))



See the Tor (multi-color) (http://openjscad.org/#examples/slices/tor.jscad) for an example.

Echo

```
a = 1, b = 2;
echo("a="+a,"b="+b);
```

prints out on the JavaScript console: *a=1, b=2*

Mathematical Functions

Javascript provides several functions through the Math library (https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Math). In addition, the following OpenSCAD compatible functions are available:

```
sin(a);                    // a = 0..360
cos(a);                    //     ''
asin(a);                   // a = 0..1, returns 0..360
acos(a);                   //       ''
tan(a);                    // a = 0..360
atan(a);                   // a = 0..1, returns 0..360
atan2(a,b);                // returns 0..360
ceil(a);
floor(a);
abs(a);
min(a,b);
max(a,b);
rands(min,max,vn,seed);    // returns random vectors of vn dimension, seed not yet implemented
log(a);
lookup(ix,v);              // ix = index, e.g. v = [ [0,100], [10,10], [20,200] ] whereas v[x][0] = index,
v[x][1] = value
                           //    return will be linear interpolated (e.g. lookup(5,[ [0,100], [10,10],
[20,200] ]) == 45

pow(a,b);
sign(a);                   // -1, 0 or 1
sqrt(a);
round(a);
```

Direct OpenSCAD Source Import

An OpenSCAD to OpenJSCAD translator is included, however the following features aren't working yet:

- DXF import and manipulation (e.g. import_dxf, dxf-cross, dxf_dim functions).
- rotate_extrude() (**Note:** OpenJSCAD supports rotate_extrude())
- minkowski() and hull() transformations (**Note:** OpenJSCAD supports hull())
- global variable such as $fa, $fs
- Modifier characters: #, !, %
- List Comprehension such as: list = [for (i = [2, 3, 5, 7, 11]) i * i];

You can edit OpenSCAD source in the built-in editor, just make sure the first line says:

```
//!OpenSCAD
```

then the source-code is shown with OpenSCAD syntax.

Further CAD languages support might arrive at a later time.

Converting OpenSCAD to OpenJSCAD

In order to translate your OpenSCAD into native OpenJSCAD code, consider this comparison.

OpenSCAD

```
union() {
      //cube(size=[30,30,0.1],center=true);
      translate([3,0,0]) cube();
      difference() {
         rotate([0,-45,0]) cube(size=[8,7,3],center=true);
         sphere(r=3,$fn=20,center=true);
      }
      translate([10,5,5]) scale([0.5,1,2]) sphere(r=5,$fn=50);
      translate([-15,0,0]) cylinder(r1=2,r2=0,h=10,$fn=20);

   for(i=[0:19]) {
      rotate([0,i/20*360,0])
      translate([i,0,0])
      rotate([0,i/20*90,i/20*90,0])
      cube(size=[1,1.2,.5],center=true);
   }
}
```

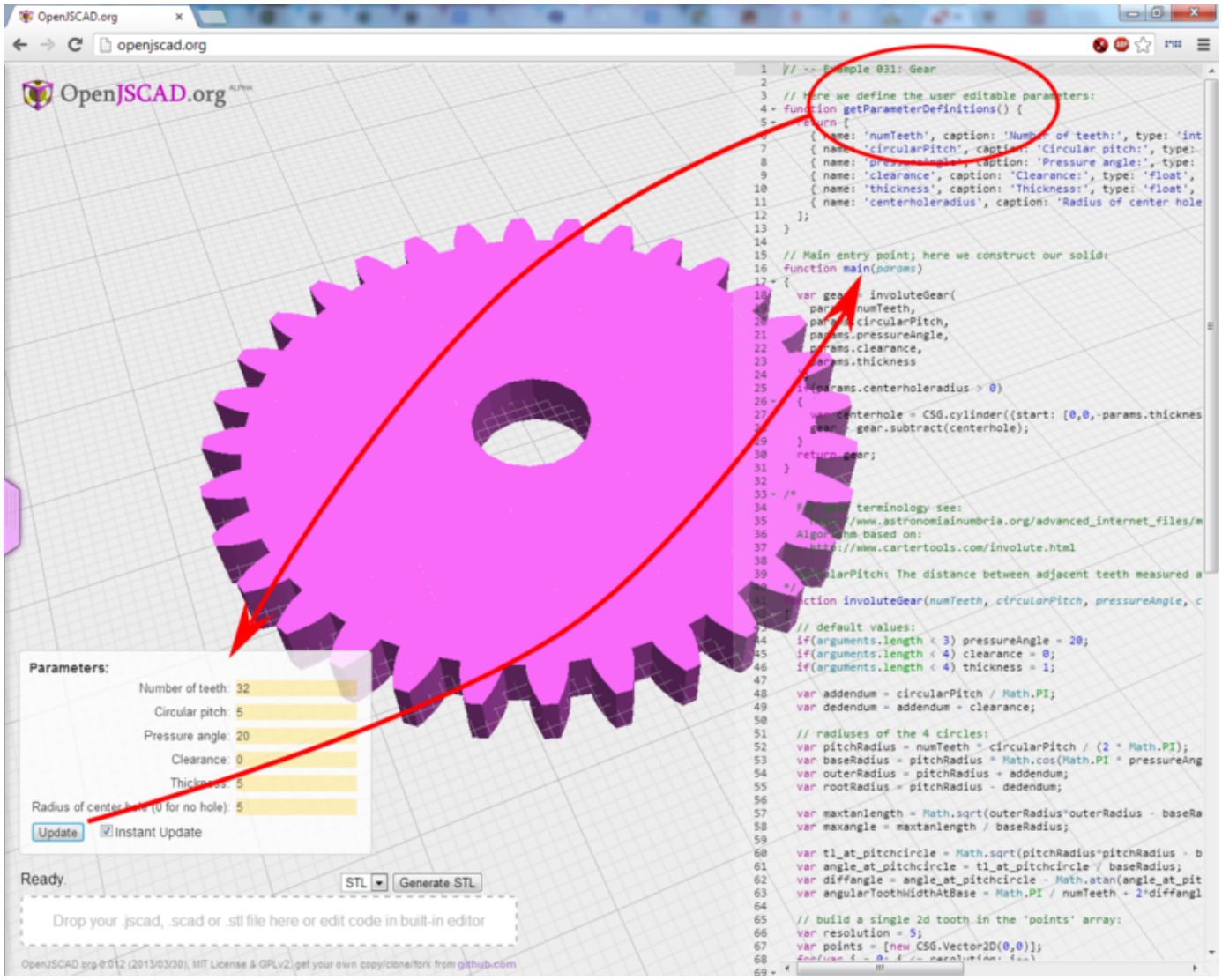
OpenJSCAD

```
function main() {
   var cubes = new Array();
   for(i=0; i&lt;20; i++) {
      cubes[i] = rotate([0,i/20*360,0],
         translate([i,0,0],
         rotate([0,i/20*90,i/20*90,0],
         cube({size:[1,1.2,.5],center:true}))));
   }
   return union(
      //cube({size:[30,30,0.1],center:true}),
      translate([3,0,0],cube()),
      difference(
         rotate([0,-45,0], cube({size:[8,7,3],center:true})),
         sphere({r:3,fn:20,center:true})
      ),
      translate([10,5,5], scale([0.5,1,2], sphere({r:5,fn:50}))),
      translate([-15,0,0], cylinder({r1:2,r2:0,h:10,fn:20})),
      cubes
   );
}
```

Essentially whenever named arguments in OpenSCAD appear *func(a=1)*, translate it into *func({a:1})*, for example:

```
// OpenSCAD
translate([0,0,2]) sphere(size=2,$fn=50)
// becomes OpenJSCAD
translate([0,0,2], sphere({size:2,fn:50}));
// or
sphere({size:2,fn:50}).translate([0,0,2]);
```

Interactive Parametric Models



Models can have interactive parameters, allowing users to change values via familiar forms, i.e. typing in numbers, sliding bars, pulling down menus, etc. This allows the user to change values and create any number of possible combinations, allowing the model to become dynamic in nature. Any number of custom designs can be created, which can then be down loaded in any supported format.

Interactive parameters are possible by adding a specific function called *getParameterDefinitions()*. This function can be added anywhere in to your JSCAD script, but must return an array of parameter definitions.

```
function getParameterDefinitions() {
  return [{ name: 'width', type: 'float', initial: 10, caption: "Width of the cube:" }];
}
```

The parameter definitions are used to create a set of fields which the user can change, i.e. options. The values of the fields are supplied to the *main()* function of your JSCAD script. For example, the value of the 'width' parameter is supplied as the 'params.width' attribute, and so on.

```
function main(params) {
  // custom error checking:
```

```
  if(params.width &lt;= 0) throw new Error("Width should be positive!");
  var mycube = CSG.cube({radius: params.width});
  return mycube();
}
```

All the common HTML5 field types are available as interactive parameters. This includes checkbox (boolean), color, date, email, float, int, number, password, slider, text and url. As well as two special parameter types for pull down choices, and grouping parameters.

A minimum parameter specification contains only 'name' and 'type'. However, a complete parameter specification should have a 'caption' and 'initial' value. In addition, there are 'min', 'max', 'step', 'checked', 'size, 'maxlength', and 'placeholder' which are relevant to specific parameter types. Just keep try different combinations to get a nice parameter specification. Here's an good example.

```
{
  name: 'width',
  type: 'float',
  initial: 1.5,
  caption: 'Width of the thingy:',
  min: 1.0,
  max: 5.0,
  step: 0.5
}
```

In addition, there is the 'choice' type which creates a drop down list of choices for the user. The choices are provided by specifying the 'values' and 'captions'. The chosen value is passed into the *main()* function of the model.

```
{
  name: 'shape',
  type: 'choice',
  values: ["TRI", "SQU", "CIR"],              // these are the values that will be supplied to your script
  captions: ["Triangle", "Square", "Circle"], // optional, these values are shown in the listbox
                                              // if omitted, the items in the 'values' array are used
  caption: 'Shape:',                          // optional, displayed left of the input field
  initial: "SQU",                             // optional, default selected value
                                              // if omitted, the first item is selected by default
                                              // NOTE: parameter "default" is deprecated
}
```

A complete example:

```
function getParameterDefinitions() {
  return [
    { name: 'width', type: 'float', initial: 10, caption: "Width of the cube:" },
    { name: 'height', type: 'float', initial: 14, caption: "Height of the cube:" },
    { name: 'depth', type: 'float', initial: 7, caption: "Depth of the cube:" },
    { name: 'rounded', type: 'choice', caption: 'Round the corners?', values: [0, 1], captions: ["No thanks",
"Yes please"], initial: 1 }
  ];
}

function main(params) {
  var result;
  if(params.rounded == 1) {
    result = CSG.roundedCube({radius: [params.width, params.height, params.depth], roundradius: 2,
resolution: 32});
  } else {
    result = CSG.cube({radius: [params.width, params.height, params.depth]});
  }
  return result;
}
```

There are plenty of examples of "Interactive" parameters available at OpenJSCAD.org (http://www.openjscad.org/). See the Quick Reference for more information about the available parameter types, and browser support.

Parameters via the Command Line Interface

The command line interface, ``openjscad``, can pass parameters for the interactive designs. Either by

```
 key value
```

or

```
key=value
```

For example:

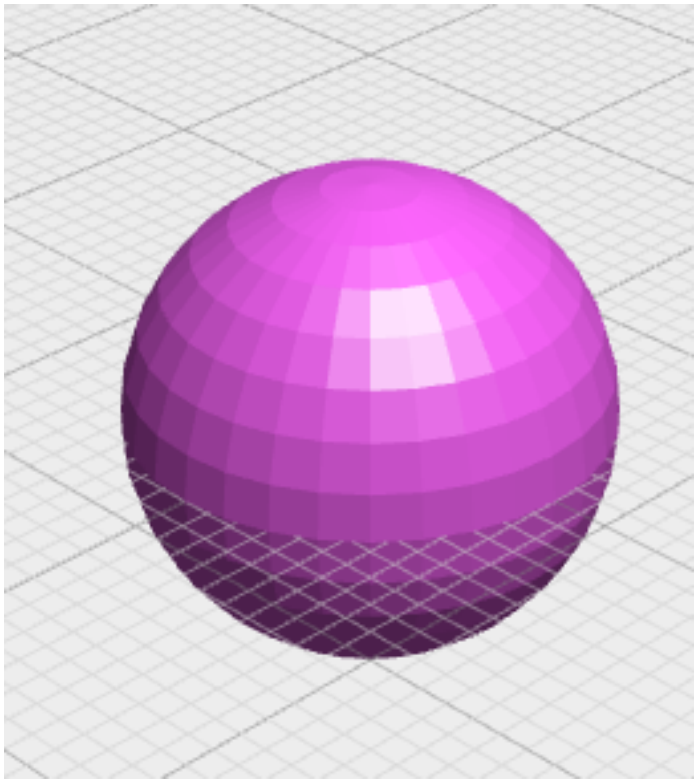
```
% openjscad name_plate.jscad --name "Just Me" --title "Geek" -o JustMe.amf
% openjscad name_plate.jscad "--name=Just Me" "--title=Geek" -o JustMe.amf
```

Maintaining Larger Projects

Including Files

After creating a few designs using OpenJSCAD, common functions and parts will become useful. The *include()* function allows one OpenJSCAD script to include another OpenJSCAD script.

Simple Example



see also

- https://github.com/BITPlan/docker-openjscad/tree/master/workspace/testinclude
- https://github.com/gilboonet/designs/tree/master/testInclude

```
// main.jscad
include("lib.jscad");

function main() {
  myLib();
  return myLib.b(2);
}
```

and

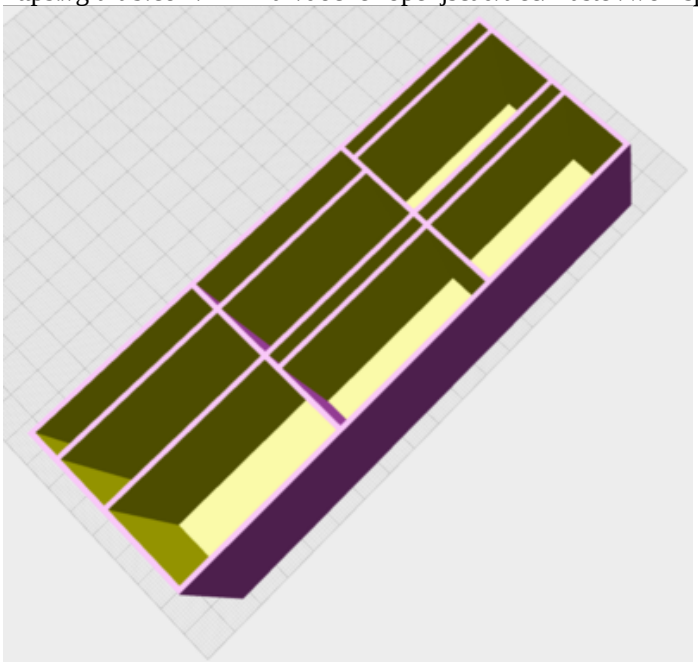
```
// lib.jscad
myLib = function() {
   var a = function(n) {  // internal
      return n*4;
```

```
    }
    myLib.b = function(n) {      // public
       return sphere(a(n));
    }
}
```

Note: the main must contain the call: myLib()

Object oriented example

A design for a Remote Control Holder that uses a class "Box" which is included from "Box.jscad". see also https://github.com/BITPlan/docker-openjscad/tree/master/workspace/RCHolder



main.jscad

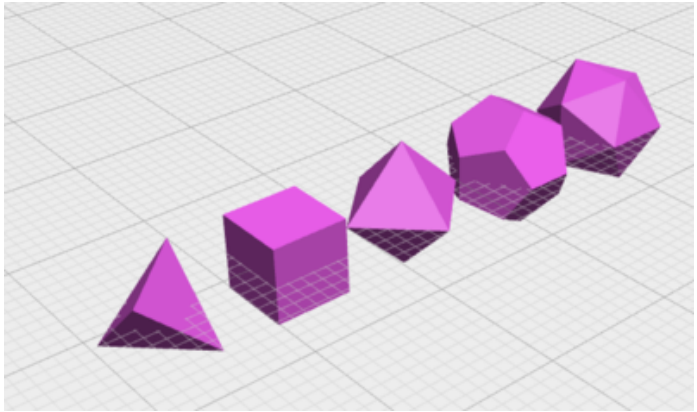
```
// title      : Remote Control Holder Test
// author     : Wolfgang Fahl
// license    : Apache License
// revision   : 0.0.1
// tags       : Cube
// file       : RCHolder/main.jscad
include ("Box.jscad");

//
function main() {
  BoxFactory();
  width = 55;
  height = 45;
  len = 30;
  wall = 1.5;
  var boxes = [];
  box=BoxFactory.create(width, len, height, wall, false);
  boxo= BoxFactory.create(width,len,height,wall,false);

  x0=-width*1.5;
  y0=-95
  box.at(x0, y0, 0);
  var ls = [30, 25, 25, 20, 20, 25];
  i=0;
  x=0;
  ls.forEach(function(length) {
    box.length=length;
    if (++i>3) {
      box.x=x0;
      box.y=y0+len-wall;
      i=0;
    }
    boxo.x=box.x;
    boxo.y=box.y;
    boxo.z=box.z;
    boxes.push(box.box());
    boxes.push(boxo.box());
```

```
    box.move(width-wall,0,0);
  });
  return union(boxes);
}
```

Box.jscad

```
// title      : Box Test
// author     : Wolfgang Fahl
// license    : Apache License
// revision   : 0.0.1
// tags       : Cube
// file       : Box.jscad
class Box {
  constructor(width, length, height, wall, center) {
    this.width = width;
    this.length = length;
    this.height = height;
    this.wall = wall;
    this.center = center;
    this.x = 0;
    this.y = 0;
    this.z = 0;
  }

  /**
   *  create a box
   */
  box() {
    return difference(
      cube({
        size: [this.width, this.length, this.height],
        center: this.center
      }),
      cube({
        size: [this.width - this.wall * 2, this.length - this.wall * 2, this.height - this.wall],
        center: this.center
      }).translate([this.wall, this.wall, this.wall])
    ).translate([this.x, this.y, this.z])
  }

  at(x, y, z) {
    this.x = x;
    this.y = y;
    this.z = z;
  }

  move(x, y, z) {
    this.at(this.x + x, this.y + y, this.z + z);
  }
}

BoxFactory=function () {
   BoxFactory.create=function(pWidth, pLength, pHeight, pWall, pCenter) {
     return new Box(pWidth,pLength,pHeight,pWall,pCenter);
   }
}
```

platonics example



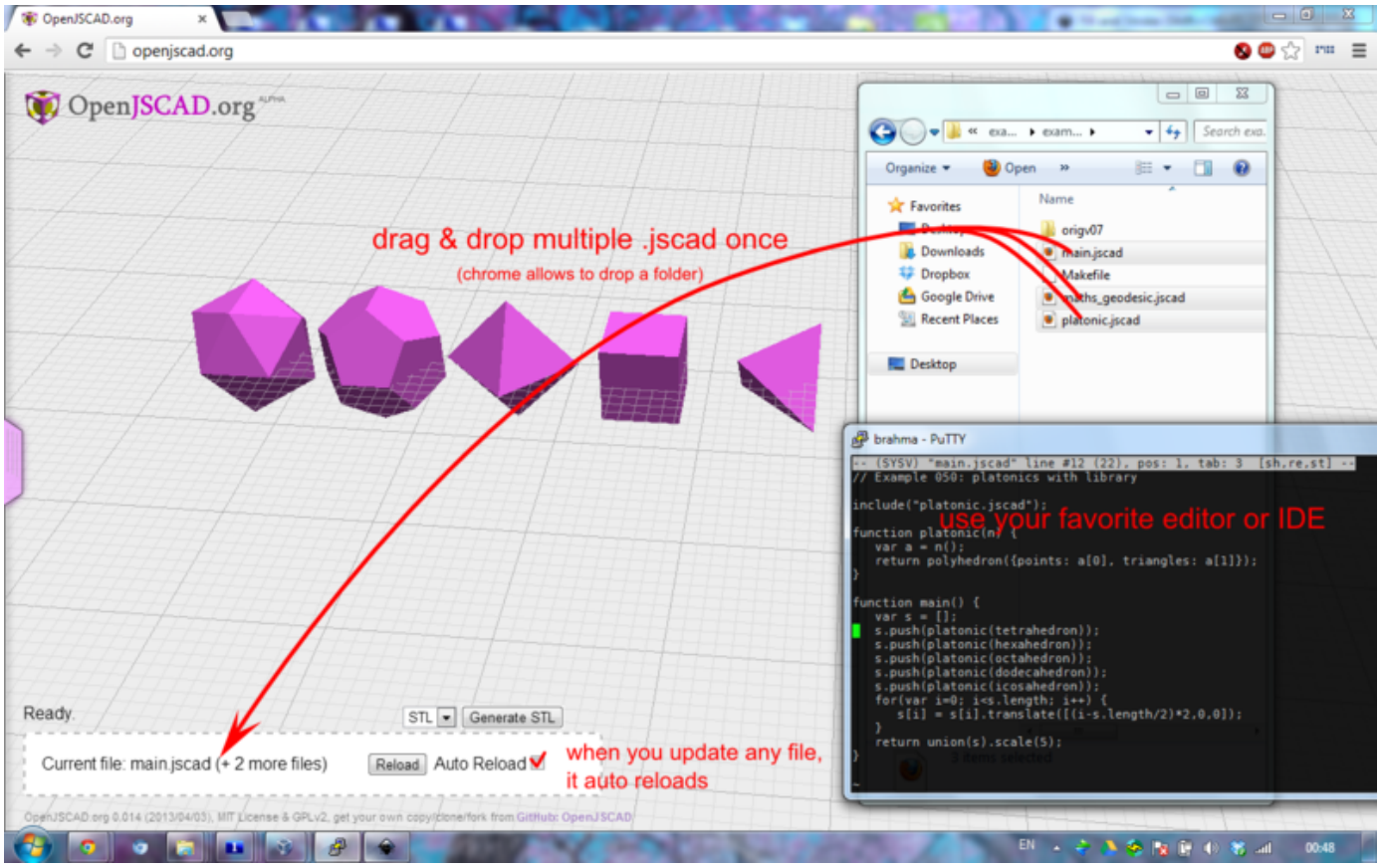
See Platonics Example (https://github.com/jscad/OpenJSCAD.org/tree/master/packages/examples/platonics) with a recursive use of include(). Yet, it's a rather bad example of not localize the function names. A clear writing style-guide will follow how an OpenJSCAD library should look like.

Support of include()

The include() function is supported

- web-online remote (e.g. http://openjscad.org/): given you drag & dropped the files, or they are available on the web-server (e.g. the examples)
- web-online local (e.g. http://localhost/OpenJSCAD/): given you drag & dropped the files, or they are available on the local web-server
- web-offline local (e.g. file://..../OpenJSCAD/index.html): given you drag & dropped the files
- command-line interface (CLI): given they are locally available in the filesystem

Example of a setup with drag & dropped files:



File Layout of JSCAD Projects

Assuming you want to create a larger OpenJSCAD project, you might use *include()* to split up the functionality:

```
ProjectName/
   main.jscad          # this one contains the "function main()", this file will be <b>executed</b>
   addon.jscad         # this file could be include("addon.jscad") in main.jscad
   optimizer.jscad     #              ''     include("optimizer.jscad") in main.jscad or also in addon.jscad
   Makefile            # possible Makefile to do the same on CLI
```

Note: The file named **main.jscad** must be present, and must contain the "function main()" declaration.

Developing with Multiple JSCAD Files

Depending on your browser and your local setup, following applies:

- Chrome (Version 26):

```
* Online (http://...): drag & drop entire folder, e.g. ProjectName/ to the drag & drop zone
* Offline (file://...): drag & drop all jscad files (but not folder) to the drag & drop zone
```

- Firefox (Version 19+): drag & drop all jscad files of the project to the drag & drop zone
- Opera: not yet working (WebGL support not yet available)
- IE10: not yet working (WebGL support not yet available)

Addendum

- Constructive Solid Geometry (CSG) Library (https://github.com/jscad/csg.js), is a modelling technique that uses Boolean operations like union and intersection to combine 3D solids.