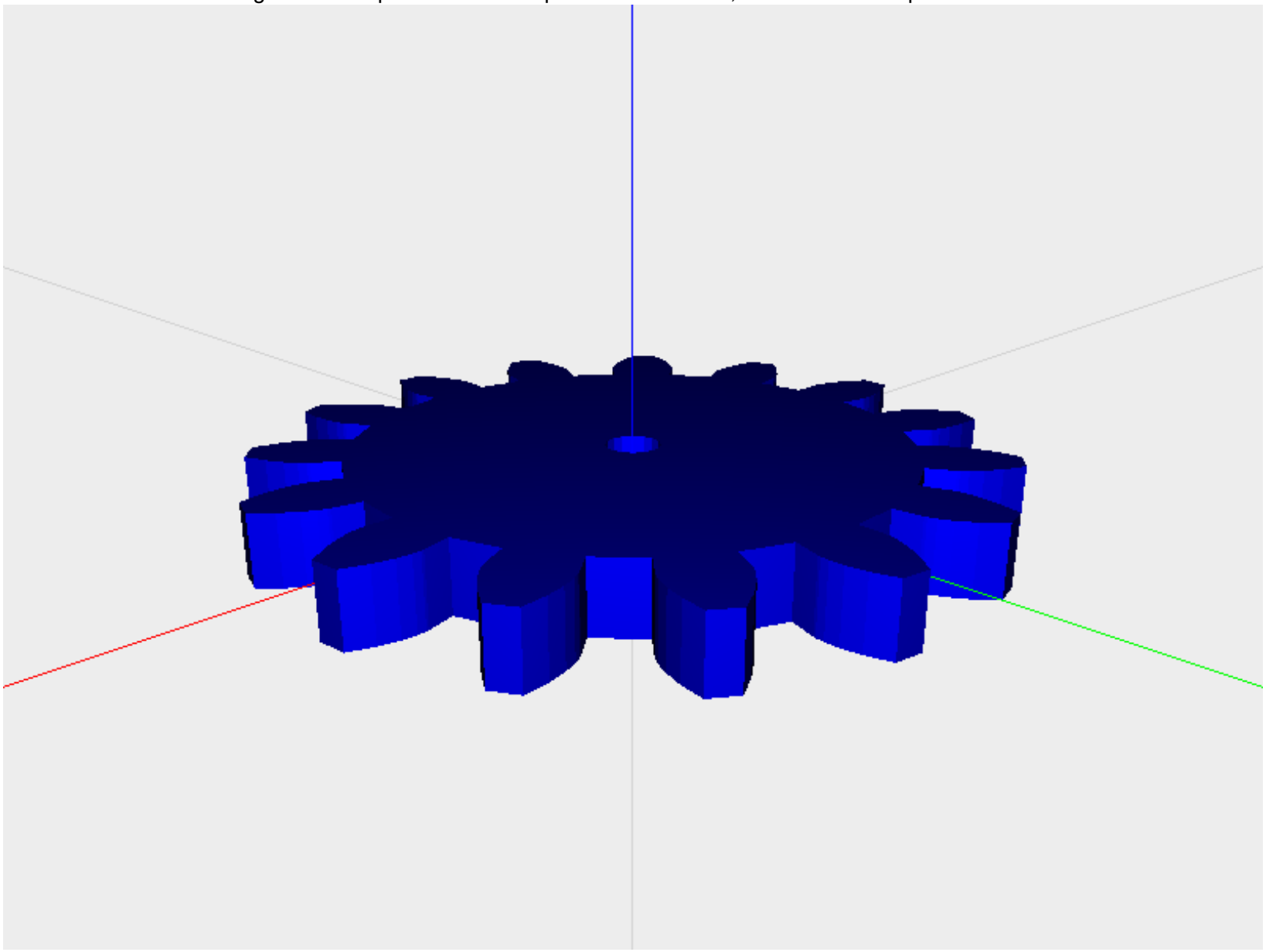
OpenJsCad

Web based solid modeling in Javascript: user editable parametric models, STL and DXF export.



Ready. Rendered in 2033ms

STL ▼ Generate STL

surfaces lines

Parameters:

Number of teeth:	15
Circular pitch:	10
Pressure angle:	20
Clearance:	0
Thickness:	5
Radius of center hole (0 for no hole):	2

Update

Playground

Try it by entering some code below. Anything you enter will be lost as soon as this page is reloaded; to build your own models you should instead store them in a .jscad file on your computer and use the **OpenJsCad parser**.

```
function main(params)
{
  // Main entry point; here we construct our solid:
  var gear = involuteGear(
    params.numTeeth,
    params.circularPitch,
    params.pressureAngle,
    params.clearance,
    params.thickness
  );
  if(params.centerholeradius > 0)
```

Update

Examples

Choose another example:

Involute Gears ▼ Update

About

OpenJsCad is a 2D and 3D modeling tool similar to OpenSCAD, but web based and using Javascript language. For example:

```
function main() {
  var cube = CSG.cube();
  return cube;
}
```

creates a cube with a radius of 1 and centered at the origin. The code should always contain a main() function. The main() function should return a CSG object (for a 3D solid) or a CAG object (for a 2D area). It's also possible to return multiple objects (a dropdown box will be shown to select the part to be displayed), see rendering multiple objects.

3D solids can be exported as STL files, 2D areas can be exported in a DXF file.

To build your own models, create a .jscad file with your javascript code and parse the file using the OpenJsCad parser. When finished, click on Generate STL and save the result in an .stl file, ready to be printed on your 3d printer.

Why use OpenJsCad?

Some of the benefits:

- Runs in your browser, no need to install any software.
- You can create parametric models with user editable parameters: parameters can be changed in the browser window, without the need to edit the source script. See the Gears demo at the top of this page for example.
- JavaScript is an extremely flexible language, supporting dynamic arrays, object oriented programming, closures, anonymous functions and more
- Solids are stored in variables. This allows for example conditional cloning of objects, something which is nearly impossible in OpenSCAD.
- Properties and Connectors (see below) make it very easy to attach objects to each other at predetermined points, even if you don't know the actual orientation or size.

- Extensive built in support for 2D and 3D math (classes for Vector2D, Vector3D, Plane, Line3D, Line2D)
- Debugging support: step through your code, set breakpoints, inspect variables, etc. See the OpenJsCad parser for details.

Viewer navigation

Click and drag to rotate the model around the origin. Hold down Alt to change the axis of rotation. Shift+Drag moves the model around. Zoom using the mouse wheel or using the slider below the model.

Tools, Libraries

- OpenSCAD.jscad: Wrapper functions to ease the translation from OpenSCAD to OpenJsCad, and command line rendering (with node.js) (by René K. Müller)

License

OpenJsCad is developed by Joost Nieuwenhuijse. Based on initial CSG.js copyright (c) 2011 Evan Wallace. Uses lightgl.js by Evan Wallace and Tomi Aarnio for WebGL rendering. Contributions by Alexandre Girard, Tom Robinson, jboecker, risacher, tedbeer, bebbi, Sahil Shekhawat. All code released under MIT license.

Contributions are welcome! It's all written in Javascript, so if you know how to use it you know how to modify it as well. To contribute go to OpenJsCad at GitHub (gh-pages tree), create your own fork and send me a pull request. Note that the code is maintained in the gh-pages branch, not the master branch. This is so that it can be accessed directly via Github pages.

Primitive solids

Currently the following solids are supported. The parameters are passed in an object; most parameters are optional. 3D vectors can be passed in an array. If a scalar is passed for a parameter which expects a 3D vector, it is used for the x, y and z value. In other words: `radius: 1` will give `radius: [1,1,1]`.

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 and CSG.defaultResolution3D. The former is used for 2D curves (circle, cylinder), the latter for 3D curves (sphere, 3D expand).

```
// a cube:
var cube = CSG.cube({
  center: [0, 0, 0],
  radius: [1, 1, 1]
});

// or alternatively by specifying two diagonally opposite corners:
var cube = CSG.cube({
  corner1: [-10, 5, 10],
  corner2: [10, -5, 1]
});

// a sphere:
var sphere = CSG.sphere({
  center: [0, 0, 0],
  radius: 2,            // must be scalar
  resolution: 32        // optional
});

// a cylinder:
var cylinder = CSG.cylinder({
  start: [0, -1, 0],
  end: [0, 1, 0],
  radius: 1,
  resolution: 16        // optional
});

// a cone:
var cone = CSG.cylinder({
  start: [0, -1, 0],
  end: [0, 1, 0],
  radiusStart: 1,
  radiusEnd: 2,
  resolution: 16        // optional
```

```
});

// a cone or cylinder sector:
var coneSector = CSG.cylinder({
  start: [0, 0, -1],
  end: [0, 0, 1],
  radiusStart: 2,
  radiusEnd: 1,
  sectorAngle: 90,
  resolution: 16        // optional
});

// like a cylinder, but with spherical endpoints:
var roundedCylinder = CSG.roundedCylinder({
  start: [0, -1, 0],
  end: [0, 1, 0],
  radius: 1,
  resolution: 16        // optional
});

// a rounded cube:
var cube = CSG.roundedCube({
  corner1: [-10, 5, 10],
  corner2: [10, -5, 1]
  roundradius: 3,
  resolution: 8,        // optional
});

// roundedCube works with a vector radius too:
var cube = CSG.roundedCube({
  radius: [10, 5, 8],
  roundradius: [3, 5, 0.01],
  resolution: 24
});
```

```
// a polyhedron (point ordering for faces: when looking at the face from the outside inwards, the points must be clockwise):
var polyhedron = CSG.polyhedron({
   points:[ [10,10,0],[10,-10,0],[-10,-10,0],[-10,10,0], // the four points at base
            [0,0,10]  ],                                  // the apex point
   faces:[ [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
});
```

CSG operations

The 3 standard CSG operations are supported. All CSG operations return a new solid; the source solids are not modified:

```
var csg1 = cube.union(sphere);
var csg2 = cube.intersect(sphere);
var csg3 = cube.subtract(sphere);

// combine multiple solids in one go (faster):
var csg1 = cube.union([solid1, solid2, solid3]);
var csg2 = cube.intersect([solid1, solid2, solid3]);
var csg3 = cube.subtract([solid1, solid2, solid3]);
```

Transformations

Solids can be translated, scaled and rotated. Multiple transforms can be combined into a single matrix transform. For matrix and vector math see below.

```
var cube = CSG.cube();

// translation:
var cube2 = cube.translate([1, 2, 3]);

// scaling:
var largecube = cube.scale(2.0);
var stretchedcube = cube.scale([1.5, 1, 0.5]);
```

```
// rotation:
var rotated1 = cube.rotateX(-45); // rotate around the X axis
var rotated2 = cube.rotateY(90);  // rotate around the Y axis
var rotated3 = cube.rotateZ(20);  // rotate around the Z axis

// combine multiple transforms into a single matrix transform:
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);
```

Mirroring

Solids can be mirrored in any plane in 3D space. For plane math see below.

```
var cube = CSG.cube().translate([1,0,0]);

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);
```

Centering

Solids can be centered

```
var cube = CSG.cube().translate([1, 2, 3]);

var cubeC = cube.center();
// only center on selected axes
var cubeCx = cube.center('x');
var cubeCxz = cube.center('x', 'z');
```

Cutting by a plane

A solid can be cut by a plane; only the part on the back side is kept:

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

// create a plane by specifying 3 points:
var plane1 = CSG.Plane.fromPoints([5,0,0], [7, 1, 0], [3, 1, 7]);

// or by specifying a normal and a point on the plane:
var plane2 = CSG.Plane.fromNormalAndPoint([3, 1, 2], [5, 0, 0]);

// and cut by the plane:
var part1 = cube.cutByPlane(plane2);

// or if we need the other half of the cube:
var part2 = cube.cutByPlane(plane2.flipped());
```

Expansion and contraction

Expansion can be seen as the 3D convolution of an object with a sphere. Contraction is the reverse: the area outside the solid is expanded, and this is then subtracted from the solid.

Expansion and contraction are very powerful ways to get an object with nice smooth corners. For example a rounded cube can be created by expanding a normal cube.

Note that these are expensive operations: spheroids are created around every corner and edge in the original object, so the number of polygons quickly increases. Expansion and contraction therefore are only practical for simple non-curved objects.

expand() and contract() take two parameters: the first is the radius of expansion or contraction; the second parameter is optional and specifies the resolution (number of polygons on spherical surfaces, per 360 degree revolution).

```
var cube1 = CSG.cube({radius: 1.0});
var cube2 = CSG.cube({radius: 1.0}).translate([-0.3, -0.3, -0.3]);
var csg = cube1.subtract(cube2);
var rounded = csg.expand(0.2, 8);
```

StretchAtPlane

This cuts the object at the plane, and stretches the cross-section there by length amount.

```
var stretched_csg = csg.stretchAtPlane(normal, point, length);
In the gear example, function main, try replacing
return gear;
with
return gear.stretchAtPlane([0, 0.05, 1], [0, 0, 0], 30);
```

Using 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]);
```

For an example checkout the Servo motor example.

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);
```

For a more complete example see the Servo motor example.

Determining the bounds of an object, lay object onto surface

The getBounds() function can be used to retrieve the bounding box of an object. getBounds() returns an array with two CSG.Vector3Ds specifying the minimum x,y,z coordinate and the maximum x,y,z coordinate.

lieFlat() lays an object onto the z=0 surface, in such a way that the z-height is minimized and it is centered around the z axis. This can be useful for CNC milling: it will transform a part of an object 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 lieFlat() the function getTransformationToFlatLying() can be used, which returns a CSG.Matrix4x4 for the transformation. Or use getTransformationAndInverseTransformationToFlatLying() to get both the forward transformation and the reverse 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);

// or:
var obj = cube3.getTransformationAndInverseTransformationToFlatLying();
var forwardtransformation=obj[0];
var backtransformation=obj[1];
cube3 = cube3.transform(forwardtransformation);

return cube3;
```

2D shapes

Two dimensional shapes can be defined through the CAG class. CAG stands for 'Constructive Area Geometry' which is the 2D variant of CSG. Arbitrary shapes can be constructed through CAG.fromPoints(), or one of the primitives CAG.circle(), CAG.rectangle() or CAG.roundedRectangle() can be used. Most of the operations for 3D CSG solids are defined for 2D CAG

shapes as well:

- union, subtract, intersect
- translate, scale, rotate, mirror. For rotation rotateZ() should be used, which rotates about the origin.
- expand(), contract()

CAG shapes can be converted into a 3D solid through extrusion:

- The extrude() function places the 2D area onto the 3D z=0 plane, and extrudes in the specified direction. Extrusion can be done with an optional twist. This rotates the solid around the z axis (and not necessariy around the extrusion axis) during extrusion. The total degrees of rotation is specified in the twistangle parameter, and twiststeps determine the number of steps between the bottom and top surface.
- To extrude in an arbitrary plane in 3D space, use extrudeInOrthonormalBasis(orthonormalbasis, depth [, options]). An orthonormal basis is a combination of a plane and a right-hand vector (read more about orthonormal bases below). depth is the extrusion thickness. By default this extrudes asymmetrically, extending from the plane. Currently one option is supported: {symmetrical: true}, which causes the extrusion to be done symmetrically in two directions about the plane.
- To extrude in one of the standard cartesian planes, use extrudeInPlane(axis1, axis2, depth [, options]). axis1 and axis 2 are one of ["X","Y","Z","-X","-Y","-Z"]. The 2D X coordinate is mapped to the 3D axis specified by axis1, and the 2D Y coordinate is mapped to the 3D axis specified by axis2. For example, extrudeInPlane("-Z","X", 10) extrudes such that the 2D x coordinate maps to the 3D negated z coordinate, and the 2D y coordinate maps to the 3D x coordinate. For options see extrudeInOrthonormalBasis above.

A 3D solid can be converted back to a 2D CAG using sectionCut() and projectToOrthoNormalBasis(). sectionCut() cuts the solid by a plane and returns the 2D intersection as a CAG object. projectToOrthoNormalBasis() works similarly but instead of cutting by a thin plane, it can be seen as the projection of the solid onto a plane using a distant light source; the result is a CAG representing the shadow of the object. Both functions take a CSG.OrthoNormalBasis as the argument. An orthonormal basis is a combination of a plane and a right-hand vector (read more about orthonormal bases below).

Note that your main() function may return a CAG object: in this case OpenJsCad will display it as a 'thin' 3D shape. The result can be exported as a DXF file, ready for laser cutting or CNC routing.

```
// Create a shape; argument is an array of 2D coordinates
var shape1 = CAG.fromPoints([[0,0], [5,0], [3,5], [0,5]]);

// 2D primitives:
var shape2 = CAG.circle({center: [-2, -2], radius: 4, resolution: 20});
```

```
var shape3 = CAG.rectangle({center: [5, -2], radius: [2, 3]});
var shape4 = CAG.rectangle({corner1: [-10, 20], corner2: [15, -30]});
var shape5 = CAG.roundedRectangle({center: [5, 7], radius: [4, 4], roundradius: 1, resolution: 24});
var shape6 = CAG.roundedRectangle({corner1: [-2, 3], corner2: [4, -4], roundradius: 1, resolution: 24});

// Expand one of the shapes: first parameter is expand radius, second is the resolution:
shape1 = shape1.expand(1, 20);

var shape = shape1.union([shape2, shape3, shape4]);

// Do some transformations:
shape.center();
shape=shape.translate([-2, -2]);
shape=shape.rotateZ(20);
shape=shape.scale([0.7, 0.9]);

// And extrude. This creates a CSG solid:
var extruded = shape.extrude({
  offset: [0.5, 0, 10],   // direction for extrusion
  twistangle: 30,       // top surface is rotated 30 degrees
  twiststeps: 10        // create 10 slices
});

// extrude such that the 2D x coordinate maps to the 3D negated z coordinate,
// and the 2D y coordinate maps to the 3D x coordinate:
var extruded2 = shape.extrudeInPlane("-Z","X", 10);


// extrude by rotation around y-axis at the origin.
// Note: for this to work, shape shouldn't have point in negative x space
var shapeX = shape.translate([20, 0]);
var rotExtruded = shapeX.rotateExtrude({
  angle: 210,        // degrees of rotation; 360 deg if left out
  resolution: 200    // resolution, optional
});
```

```
var cube2d = CSG.cube({radius: 10}).rotateZ(45);
var z0basis = CSG.OrthoNormalBasis.Z0Plane();
var cag = cube2d.sectionCut(z0basis);
// or:
var cag = cube2d.projectToOrthoNormalBasis(z0basis);
```

Using a CNC router it's impossible to cut out a true sharp inside corner. The inside corner will be rounded due to the radius of the cutter. The CAG.overCutInsideCorners(radius) function compensates for this by creating an extra cutout at each inner corner so that the actual cut out shape will be at least as large as needed:

```
var r1=CAG.rectangle({center:[0,0], radius: 4});
var r2=CAG.rectangle({center:[2,0], radius: 2});
var r3=r1.subtract(r2);
var cutterradius=1;
var r4=r3.overCutInsideCorners(cutterradius);

return r4;
```

For an example of 2D shapes see the Parametric S hook example.

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 contructed either by giving a series of 2D coordinates, or through the CSG.Path2D.arc() function, which creates a circular curved path. Paths can be concatenated, 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. Two methods are provided:
// arc() creates an circular segment with a specified center and radius:
var curvedpath = CSG.Path2D.arc({
  center: [0,0,0],
  radius: 10,
  startangle: 0,
  endangle: 180,
  resolution: 16,
});

// or appendArc() can be used to create a circular or elliptical segment between two endpoints.
// appendArc() closely follows the SVG arc specification:
// http://www.w3.org/TR/SVG/paths.html#PathDataEllipticalArcCommands
// The starting point for the arc is the current endpoint of the path
// path2d = path2d.appendArc(endpoint, options);
// endpoint is the destination coordinate
// options:
//      .resolution      // smoothness of the arc (number of segments per 360 degree of rotation)
// to create a circular arc:
//      .radius
// to create an elliptical arc:
//      .xradius
//      .yradius
```

```
//      .xaxisrotation   // the rotation (in degrees) of the x axis of the ellipse with respect
//                       // to the x axis of our coordinate system
// this still leaves 4 possible arcs between the two given points. The following two flags select which one we draw:
//      .clockwise       // = true | false (default is false). Two of the 4 solutions draw clockwise with respect to
//                       // the center point, the other 2 counterclockwise
//      .large           // = true | false (default is false). Two of the 4 solutions are an arc longer than 180
//                       // degrees, the other two are <= 180 degrees
var curvedpath2 = new CSG.Path2D([[0,0], [10,0]]);
// create an elliptical arc from [10,0] to [15,0]:
curvedpath2 = curvedpath2.appendArc([15,0], {
  xradius: 4,
  yradius: -6,
  xaxisrotation: 30,
  resolution: 48,
  clockwise: false,
  large: true,
});

// Extrude the path by following it with a rectangle (upright, perpendicular to the path direction)
// Returns a CSG solid
//   width: width of the extrusion, in the z=0 plane
//   height: height of the extrusion in the z direction
//   resolution: number of segments per 360 degrees for the curve in a corner
//   roundEnds: if true, the ends of the polygon will be rounded, otherwise they will be flat
var csg = path.rectangularExtrude(3, 4, 16, true);
return csg;
```

2D Bézier curves

Smooth 2D shapes can be constructed using CSG.Path2D's appendBezier() method. A Bézier curve is a smooth line between two endpoints, guided by control points. The curve in general only passes through the endpoints. The intermediate control points guide the curve in a certain direction but the curve in general does not go through the intermediate control points. In vector image editing applications (such as Inkscape) usually cubic Bézier are used: such a curve has two control points inbetween the two endpoints. But in fact appendBezier() supports any order Bézier curve.

```
path2d.appendBezier(controlpoints[, options]) - create a single Bézier segment

controlpoints: an array of control points. Each control point is a 2d coordinate (can be an [x,y] array
  or a CSG.Vector2D). The Bézier curve starts at the last point in the existing path2d (hence path2d must
  not be empty), and ends at the last given control point. Other control points are intermediate control points.
  The first control point may be null. In that case a smooth transition is ensured: the null control
  point is replaced by the second last control point of the previous Bézier curve mirrored into the
  starting point of this curve. In other words the incoming gradient matches the outgoing gradient at
  the current starting point.

options: an optional object. Currently only one option is supported:
  resolution: controls the smoothness of the curve: at least this number of segments will be used per
  360 degree rotation. If omitted, CSG.defaultResolution2D will be used.
```

An example:

```
var path=new CSG.Path2D([[0,0]]);  // beware of the double array: we must pass an array of 2d coordinates

// bezier curve from [0,0] to [10,10]
// with 3 intermediate control points so it is a 3rd order bezier
// Since we do not give a resolution option, CSG.defaultResolution2D is used to determine the number of vertices
path = path.appendBezier([[2,8], [5,0], [8,6], [10,10]]);

// cubic bezier curve from [10,10] to [10,0]
// first control point is null: this forces a smooth transition from the previous bezier curve
// In this case the null control point will be replaced by [10,10]+([10,10]-[8,6]) = [12,14]
// i.e. the last control point will be mirrored
// Here we give the optional options object, setting a resolution of 24 steps per 360 degree of revolution
// So the angle between each subsequent side is guaranteed to be less than 15 degrees
path = path.appendBezier([null, [10,0]], {resolution: 24});

// close the path and convert to a solid 2D shape:
path = path.close();
var cag = path.innerToCAG();
```

```
return cag;
```

ConnectorLists

A CSG can be created by following a CSG.ConnectorList with a CAG, or a function returning a CAG. Some helper functions are provided to ease creating a CSG.ConnectorList

```
// objective: place a cag (cagFunc or static cag) along a list of connectors.
// cover the resulting skeleton with polygons to create the resulting CSG

// The basics work like this:

// create dynamic cag as function(connector)
var cagFunc = function(point, axis, normal) {
    return CAG.circle({radius: Math.abs((point.x-8)/10+point.length()) + 0.01});
};

// connector list
var cs = new CSG.ConnectorList();
for (var i = 0; i < 10; i++) {
    cs.appendConnector(
        new CSG.Connector([i, -i, 0], [1, -i/5, i/6], [0, 1, 0])
    );
}
// create resulting CSG
return cs.followWith(cagFunc);

// To ease creating curved ConnectorLists etc., we can make one from a Path2D.
var curvedpath = CSG.Path2D.arc({
  center: [0,0,0],
  radius: 10,
  startangle: 0,
  endangle: 180,
```

```
  resolution: 50
});

// When creating a ConnectorList from this, it knows the points and will turn them
// into 3d points by adding a 0 z height.
// Then there are 2 ways to add corresponding axisvectors that define the CAG direction:
// This signature calculates pseudo-tangents for the axisvectors. Only the first and the
// last axisvector have to be manually given, as tangents for these cannot be found
var cs2 = CSG.ConnectorList.fromPath2D(curvedpath, [0, 1, 0], [0, -1, 0]);
var csg = cs2.followWith(CAG.rectangle());
return csg;

// Or if there are special requirements on the directions, these can be provided using
// a function
var cs3 = CSG.ConnectorList.fromPath2D(curvedpath, function(pt, i) {
    return pt.angleDegrees() + 90;
});
return cs3.followWith(CAG.circle());
```

Interactive parametric models

It is possible to make certain parameters editable in the browser. This allows users not familiar with JavaScript to create customized STL files.

To do so, add a function getParameterDefinitions() to your .jscad source. This function should return an array with parameter definitions. Currently 6 parameters types are supported: float, int, text, longtext, bool and choice. The user edited values of the parameters will be supplied as an object parameter to the main() function of your .jscad file.

A float, int, bool or text parameter is created by including the following object in the array returned by getParameterDefinitions():

```
{
  name: 'width',
  type: 'float',                    // or 'text', 'int', 'longtext', 'bool'
```

```
  initial: 1.23,                    // optional, sets the initial value
  // 'default': 1.23                // this is still supported for backward compatibility but deprecated ('default' is a Javascript keyw
  caption: 'Width of the thingy:',  // optional, displayed left of the input field
                                    // if omitted, the 'name' is displayed (i.e. 'width')
}
```

A 'choice' parameter is created using the following object:

```
{
  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
}
```

To use the values add an argument to your main() function. This argument will be supplied an object with the user edited parameter values:

```
function main(params)
{
  // custom error checking:
  if(params.width <= 0) throw new Error("Width should be positive!");

  if(params.shape == "TRI")
  {
    // do something
  }
}
```

A complete example. Copy/paste it into the Playground at the top of this page to see how it works:

```
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;
}
```

Or see the Involute Gears example for another example of interactive parameters.

Rendering multiple objects

In particular when laser cutting or CNC machining, an artwork often consists of multiple objects that will be machined separately. Usually the model is designed as a whole, and then the partial objects are created from the complete model. To facilitate this it's possible to return multiple objects from the main() function. This avoids the need to render the jscad file multiple times to get the STL or DXF file for each part. Returning multiple objects can be done simply by returning an array of CSG and/or CAG objects. Alternatively each object can be given a default file name and a caption by returning an object as follows:

```
function main()
{
  var mycube = CSG.cube({radius: 10});
  var mycircle = CAG.circle({radius: 10});

  // we could just do:
  // return [mycube, mycircle];

  // give each element a description and filename:
  return [
    {
      name: "cube",              // optional, will be used as a prefix for the downloaded stl file
      caption: "A small cube",   // will be shown in the dropdown box
      data: mycube,
    },
```

```
    {
      name: "circle",           // optional, will be used as a prefix for the downloaded dxf file
      caption: "Circle",
      data: mycircle,
    },
  ];
}
```

If more than 1 object is returned, a dropdown box is displayed for selecting the part to be displayed and/or downloaded. The jscad is not re-rendered so switching between parts happens fast.

Of course it's still possible to return just a single CSG or CAG:

```
function main()
{
  return CSG.cube({radius: 10});
}
```

Orthonormal basis

An orthonormal basis can be used to convert 3D points to 2D points by projecting them onto a 3D plane. An orthonormal basis is constructed from a given plane. Optionally a 'right hand' vector can be given, this will become the x axis of the two dimensional plane. If no right hand vector is given, a random one is chosen.

CSG.OrthoNormalBasis.Z0Plane() creates an orthonormal basis for the z=0 plane. This transforms (xx,yy,zz) 3D coordinates into the 2D (xx, yy) coordinates, or vice versa from (xx, yy) into (xx, yy, 0).

CSG.OrthoNormalBasis.GetCartesian(axis1, axis2) creates an orthonormal basis for the standard XYZ planes. axis1 and axis 2 are one of ["X","Y","Z","-X","-Y","-Z"]. The orthonormal basis is constructed such that the 2D x coordinate is mapped to the 3D axis specified by axis1, and the 2D y coordinate is mapped to the 3D axis specified by axis2.
For example: CSG.OrthoNormalBasis.GetCartesian("-Y","Z") will return an orthonormal basis where the 2D X axis maps to the 3D negated Y axis, and the 2D Y axis maps to the 3D Z axis.
CSG.OrthoNormalBasis.GetCartesian("X","Y") is the same as CSG.OrthoNormalBasis.Z0Plane()

Use to2D() and line3Dto2D() to convert from the 3D space to the 2D plane. Use to3D() and line2Dto3D() to convert the other way.

getProjectionMatrix() gives the projection matrix to transform into the orthonormal basis. getInverseProjectionMatrix() gives the matrix to transform back into the original basis.

```
// construct a plane:
var plane = CSG.Plane.fromNormalAndPoint([1,1,0], [0,0,1]);
var orthobasis = new CSG.OrthoNormalBasis(plane);
// or if we would like a specific right hand vector:
// var orthobasis = new CSG.OrthoNormalBasis(plane, [0,0,1]);

var point3d = new CSG.Vector3D(1,5,7);
var point2d = orthobasis.to2D(point3d);
var projected = orthobasis.to3D(point2d);
```

Getting features (volume and area)

The volume and surface area of a solid can be calculated using getFeatures(). For performance reasons, the function is designed such that it can calculate multiple features in a single pass.

```
function main() {
  var cube = CSG.cube({radius: 1});

  // var volume = cube.getFeatures("volume");
  // var area = cube.getFeatures("area");

  var features = cube.getFeatures(["volume","area"]);
  var volume = features[0];
  var area = features[1];

  OpenJsCad.log("volume: "+volume+"; area: "+area);   // volume: 8; area: 24
  return cube;
}
```

Miscellaneous

Solids can be given a color using the setColor() function. setColor() expects an array with 3 or 4 elements [r, g, b, [alpha]]. Beware: this returns a new solid, the original solid is not modified! Faces of the solid will keep their original color when doing CSG operations (union() etc). Color values range between 0.0 and 1.0 (not 255).

```
var cube1 = CSG.cube({radius: 10});
cube1 = cube1.setColor([0.5, 0, 0]);

var cube2 = CSG.cube({radius: 10});
cube2 = cube2.setColor([0, 0.5, 0]);
cube2 = cube2.translate([5,1,4]);

var result = cube1.subtract(cube2);
// the resulting solid will have faces with 2 different colors
```

2D and 3D Math

There are utility classes for many 2D and 3D operations. Below is a quick summary, for details view the source of csg.js:

```
// --------- Vector3D --------------------
var vec1 = new CSG.Vector3D(1,2,3);       // 3 arguments
var vec2 = new CSG.Vector3D( [1,2,3] );   // 1 array argument
var vec3 = new CSG.Vector3D(vec2);        // cloning a vector
// get the values as: vec1.x, vec.y, vec1.z
// vector math. All operations return a new vector, the original is unmodified!
// vectors cannot be modified. Instead you should create a new vector.
vec.negated()
vec.abs()
vec.plus(othervector)
vec.minus(othervector)
vec.times(3.0)
vec.dividedBy(-5)
vec.dot(othervector)
```

```
vec.lerp(othervector, t)  // linear interpolation (0 <= t <= 1)
vec.length()
vec.lengthSquared()       // == vec.length()^2
vec.unit()
vec.cross(othervector)    // cross product: returns a vector perpendicular to both
vec.distanceTo(othervector)
vec.distanceToSquared(othervector)  // == vec.distanceTo(othervector)^2
vec.equals(othervector)
vec.multiply4x4(matrix4x4)   // right multiply by a 4x4 matrix
vec.min(othervector)        // returns a new vector with the minimum x,y and z values
vec.max(othervector)        // returns a new vector with the maximum x,y and z values

// --------- Vector2D ---------------------
var vec1 = new CSG.Vector2D(1,2);       // 2 arguments
var vec2 = new CSG.Vector2D( [1,2] );   // 1 array argument
var vec3 = new CSG.Vector2D(vec2);      // cloning a vector
// vector math. All operations return a new vector, the original is unmodified!
vec.negated()
vec.abs()
vec.plus(othervector)
vec.minus(othervector)
vec.times(3.0)
vec.dividedBy(-5)
vec.dot(othervector)
vec.lerp(othervector, t)  // linear interpolation (0 <= t <= 1)
vec.length()
vec.lengthSquared()       // == vec.length()^2
vec.unit()
vec.normal()              // returns a 90 degree clockwise rotated vector
vec.distanceTo(othervector)
vec.distanceToSquared(othervector)  // == vec.distanceTo(othervector)^2
vec.cross(othervector)    // 2D cross product: returns a scalar
vec.equals(othervector)
vec.min(othervector)        // returns a new vector with the minimum x and y values
vec.max(othervector)        // returns a new vector with the maximum x and y values
```

```
vec.multiply4x4(matrix4x4)   // right multiply by a 4x4 matrix
vec.toVector3D(z)         // convert to a vector3D by adding a z coordinate
vec.angleDegrees()        // returns the angle of the vector: [1,0] = 0 degrees, [0, 1] = 90 degrees, etc
vec.angleRadians()        // ditto in radians
var vec = CSG.Vector2D.fromAngleDegrees(degrees);  // returns a vector at the specified angle
var vec = CSG.Vector2D.fromAngleRadians(radians);  // returns a vector at the specified angle

// --------- Matrix4x4 --------------------
var m1 = new CSG.Matrix4x4();          // unity matrix
var m2 = new CSG.Matrix4x4( [1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1] );
  // elements are passed in row order
var result = m1.plus(m2);
var result = m1.minus(m2);
var result = m1.multiply(m2);
// matrix vector multiplication (vectors are padded with zeroes to get a 4x1 vector):
var vec3d = m1.rightMultiply1x3Vector(vec3d);  // matrix * vector
var vec3d = m1.leftMultiply1x3Vector(vec3d);   // vector * matrix
var vec2d = m1.rightMultiply1x2Vector(vec2d);  // matrix * vector
var vec2d = m1.leftMultiply1x2Vector(vec2d);   // vector * matrix
// common transformation matrices:
var m = CSG.Matrix4x4.rotationX(degrees);      // matrix for rotation about X axis
var m = CSG.Matrix4x4.rotationY(degrees);      // matrix for rotation about Y axis
var m = CSG.Matrix4x4.rotationZ(degrees);      // matrix for rotation about Z axis
var m = CSG.Matrix4x4.rotation(rotationCenter, rotationAxis, degrees); // rotation about arbitrary point and axis
var m = CSG.Matrix4x4.translation(vec3d);      // translation
var m = CSG.Matrix4x4.scaling(vec3d);          // scale
var m = CSG.Matrix4x4.mirroring(plane);        // mirroring in a plane; the argument must be a CSG.Plane
// matrix transformations can be concatenated:
var transform = CSG.Matrix4x4.rotationX(20).multiply(CSG.Matrix4x4.rotationY(30));
// Use a CSG solid's transform() method to apply the transformation to a CSG solid

// ------------ Plane -------------------------
// a 3D plane is represented by a normal vector (should have unit length) and a distance from the origin w
// the plane passes through normal.times(w)
var plane1 = new CSG.Plane(normal, w);
```

```
// Or we can construct a plane from 3 points:
var plane2 = CSG.Plane.fromPoints(p1, p2, p3);
// Or from a normal vector and 1 point:
var plane3 = CSG.Plane.fromNormalAndPoint(normal, point);
// Flip a plane (front side becomes back side):
var plane4 = plane3.flipped();
// Apply transformations (rotation, scaling, translation):
var transformed = plane3.transformed(matrix4x4);  // argument is a CSG.Matrix4x4
// Intersection of plane and 3d line:
var point = plane3.intersectWithLine(line);        // argument is CSG.Line3D, returns a CSG.Vector3D
// Intersection of 2 planes:
var line = plane3.intersectWithPlane(plane);       // argument is another CSG.Plane, returns a CSG.Line3D
// Distance to point:
var w = signedDistanceToPoint(point);             // argument is CSG.Vector3D, returns a float (positive
                                                  //    if in front of plane, negative if in back)

// ------------ Line3D --------------------------
// A line in 3d space is represented by a point and a direction vector.
// Direction should be a unit vector. Point can be any point on the line:
var line = new CSG.Line3D(point, direction);      // argumenst are CSG.Vector3D
// or by giving two points:
var line = CSG.Line3D.fromPoints(p1, p2);         // argumenst are CSG.Vector3D
var point = intersectWithPlane(plane);            // == plane.intersectWithLine(this)
var line2 = line.reverse();                       // same line but reverse direction
var line2 = line.transform(matrix4x4);            // for rotation, scaling, etc
var p = line.closestPointOnLine(point);           // project point onto the line
var d = line.distanceToPoint(point);

// ------------ Line2D --------------------------
// A line in 2d space is represented by a normal vector and a distance w to the
// origin along the normal vector
// A line in 2d space is represented by a point and a direction vector.
// Direction should be a unit vector. Point can be any point on the line:
var line = new CSG.Line2D(CSG.Line2D(normal,w));
// or by giving two points:
```

```
var line = CSG.Line2D.fromPoints(p1, p2);        // argumenst are CSG.Vector2D
var line2 = line.reverse();                      // same line but reverse direction
var line2 = line.transform(matrix4x4);           // for rotation, scaling, etc
var point = line.origin();                       // returns the point closest to the origin
var dir = line.direction();                      // direction vector (CSG.Vector2D)
var x = line.xAtY(y);                            // returns the x coordinate of the line at given y coordinate
var d = absDistanceToPoint(point);               // returns the absolute distance between a point and the line
var p = line.closestPoint(point);                // projection of point onto the line
var point = line.intersectWithLine(line2);       // intersection of two lines, returns CSG.Vector2D
```