Starting design with OpenSCAD

Adrian Johnstone, January 23, 2020

OpenSCAD is a free, text-based, 3D modelling tool which runs on Windows, OS X and Linux. It is easy to install and use, but complicated things sometimes need a lot of thought.

The learning curve – visual *vs* text-based tools

Most 3D design tools are visual: objects are drawn directly by the engineer and manipulated using a point-and-click interface. Skilled users can achieve excellent results but learning such a tool can take a long time.

There is an alternative: write a script describing how to build the 3D model out of a few geometrical primitives such as cylinders, spheres and cuboids, and then have the computer draw it for you. The OpenSCAD tool uses this approach and runs on all common current personal computers. There is a related tool called OpenJSCAD which runs purely in the browser and thus can even be used on tablets and phones.

These tools are easy to learn because they only provide a few very simple facilities. On the other hand, operations such as adding a fillet to a joint can probably be done with a single command in most visual packages, whereas in OpenSCAD some ingenuity is sometimes required.

Advantages and disadvantages

The main advantages of these script-based systems are

Interoperability: scripts are simple text files which can be edited with normal text editors, emailed and displayed without needing to install special software.

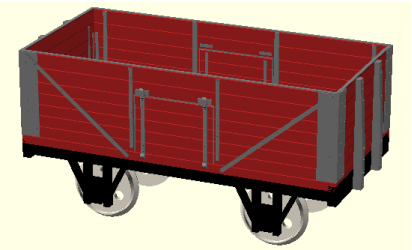
Modularity: the language encourages packaging of designs into small named modules that can be shared and reused

Paramaterisability: instead of using explicit numbers everywhere, we can used named values that are specified once and used many times: modifying the single named value allows global changes to the properties of a 3D model.

Precision: object locations and dimensions are specified numerically and exactly.

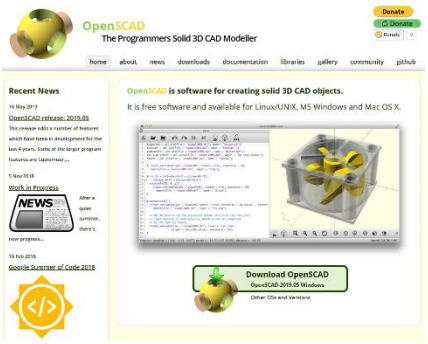
Free, mostly bug free and lots of free examples: these tools are simple compared to visual editors and have few untested dark corners. OpenSCAD has a vast number of users, so there is a lot of online help available.

The main disadvantage is that rather than just picking up the corner of an object on the screen and stretching it so that it looks right, the engineer has to work out numerically where it should go, and that often requires some trig and geometry. Specifications can also become quite large and that requires discipline when organising modules. For instance the script for this RCH wagon, with its multitude of plates, nuts and bolts, is over a thousand lines long.

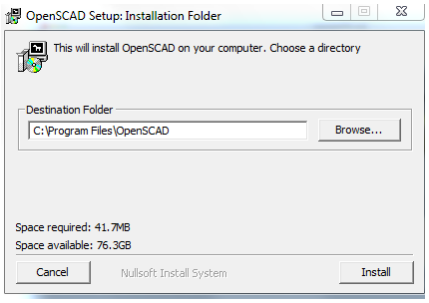


Installing OpenSCAD on Windows and generating 3D prints

1. Use a web browser to visit https://www.openscad.org/ which in January 2020 looked like this:

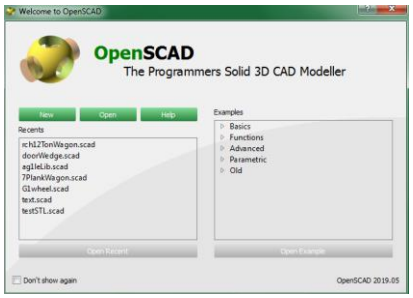


2. Click the **Download OpenSCAD** button and follow your browser's procedure to save and run the file OpenSCAD-2019.05-x86-64-Installer.exe which will take you to this dialog box:

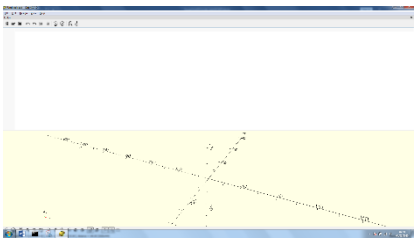


3. The default installation directory is fine, so just click the **Install** button. OpenSCAD is small, and so installation should take little time.

4. Start OpenSCAD via the Windows start menu. The welcome dialog looks like this (but your list of recent files would initially be empty).



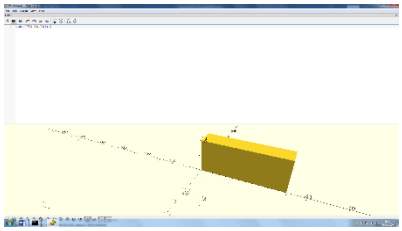
5. Clock on the **New** button. A window will open showing an empty text editor and an empty 3D preview.



6. Click on the white editor window, and type into it this line

```
cube([50,10,20]);
```

7. Press function key 5 (F5) and the script will be rendered



8. Click on the 3D preview window and try rotating and panning by click-and-dragging the left and right buttons. You can zoom in and out using your mouse wheel, or by using the  buttons at the bottom of the window.

9. The F5 preview function is fast for viewing but does not produce the detail needed for a 3D print. To prepare for printing, press F6 to do a full render. For simple objects, the appearance does not change. For complex objects, especially those with curves, there may be an appreciable delay.

10. Then press F7 to get the save STL dialog, save the STL file to your preferred location and try a test print.

-

A note on the examples

In the following examples, text that is to be typed directly into OpenSCAD is shown in

```
this font
```

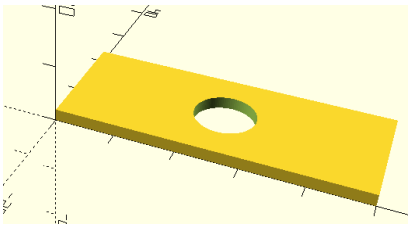
Each example begins with the complete text of the specification which can be cut and pasted (or typed) into a fresh, empty editor window.

Each example is also built up in stages. You can follow the development by starting with an empty editor window, and then at each stage insert the material highlighted `in this colour`.

Example 1 – boring holes in plates

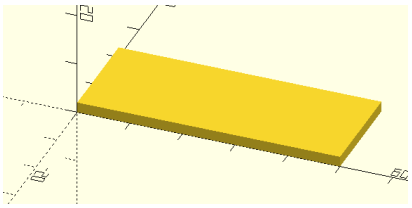
OpenSCAD uses *computational solid geometry* to build up complex objects from simple geometrical shapes and operations such as union, intersection and difference.

Here is a 50mm x 20mm plate, 2mm thick with a 10mm hole in its centre. In OpenSCAD, we specify this by taking a cuboid and removing a cylinder from it. This is the full specification:



```
difference() {
  cube([50,20,2]);
  translate([25,10,0]) cylinder(10,5,5);
}
```

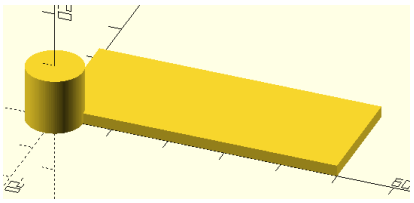
To build the example up in stages, begin with a new empty window, and type into it



```
cube([50,20,2]);
```

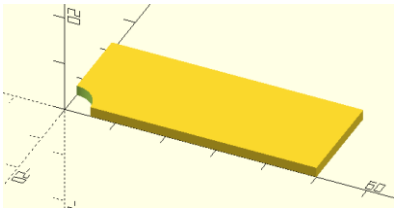
Press F5, and it will be rendered in the graphics window

Now add a 10mm high cylinder of radius 5mm by writing:



```
cube([50,20,2]); cylinder(10,5,5);
```

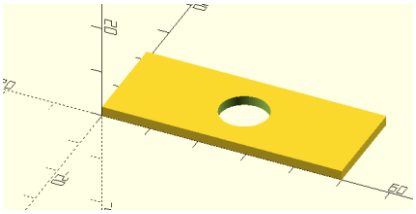
Objects can be 'subtracted' from one another using the **`difference()`** operation. If we write



```
difference() { cube([50,20,2]); cylinder(10,5,5); }
```

then the second object (the cylinder) is taken away from the first (the plate)

Objects can be moved about in space using the **`translate()`** operation. In OpenSCAD, points in space are specified using a *vector* which is written as a coordinate triple in square brackets: [x, y, z].



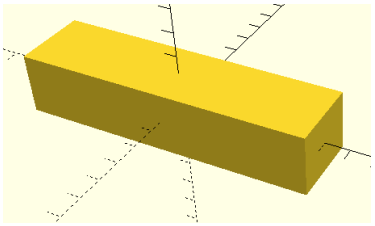
If the cylinder's position is moved to the centre of the plate, then we get the effect of a drilled hole:

```
difference() {
  cube([50,20,2]);
  translate([25,10,0]) cylinder(10,5,5);
}
```

Example 2 – a door wedge

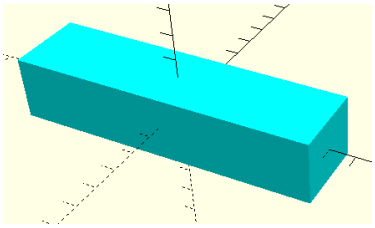
Here is a 3D printed door wedge which we can describe in OpenSCAD as the difference between two cuboids. First, we show the full specification, then after the line we shall build it up in stages.

```
difference() {
  color ("cyan") cube([120, 30, 25], center=true);
  translate([0,0,10])
  rotate([0,-7,0]) cube([125, 35, 25], center=true);
}
```



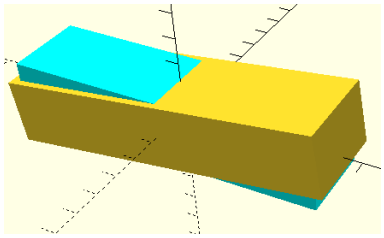
The wedge is 12cm by 3cm by 2.5cm at its highest point. OpenSCAD uses millimetres as the basic unit, so we make a cuboid of the correct size, centred at the origin by writing:

```
cube([120, 30, 25], center=true);
```



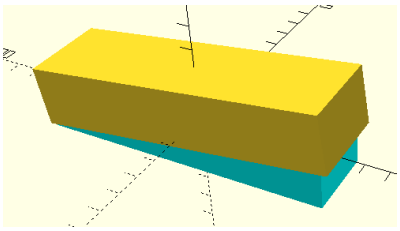
By default, everything in OpenSCAD renders as a sort-of golden yellow. When building up a design, it is useful to add colours to show the parts that are being currently manipulated: an object can be prefixed by a colour specification (written the American way):

```
color ("cyan") cube([120, 30, 25], center=true);
```



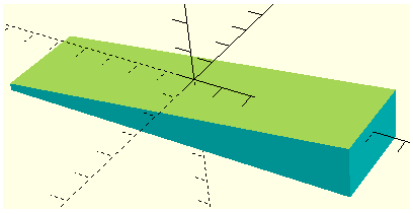
We now add a second cuboid, rotated by seven degrees around the Y axis to form the top surface of the wedge:

```
color ("cyan") cube([120, 30, 25], center=true);
rotate([0,-7,0]) cube([125, 35, 25], center=true);
```



The slanted cuboid needs to be raised by 10mm: a translation of 10 units along the Z axis:

```
color ("cyan") cube([120, 30, 25], center=true);
translate([0,0,10])
rotate([0,-7,0]) cube([125, 35, 25], center=true);
```



Finally, we ‘subtract’ the slanted cuboid from the original one to form the wedge:

```
difference() {
  color ("cyan") cube([120, 30, 25], center=true);
  translate([0,0,10])
  rotate([0,-7,0]) cube([125, 35, 25], center=true);
}
```

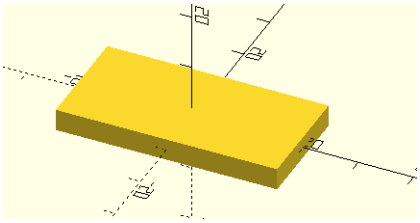
Example 3 – a labelled key fob

This key fob is the intersection between a cuboid and a large cylinder (to form the curved ends) with a small cylinder removed (to form the hole) and some text extruded onto the top. Here is the full specification:

```
$fn=180;
length=20;

difference(){
  intersection() {
    cube([length*2,length,4], center=true);
    cylinder(d=2*length, h=8, center=true);
  }
  translate([-0.75*length,0,0]) cylinder(d=5, h=8,center=true);
}

translate([-0.4*length,1,1]) linear_extrude(2)text("Shed", 6 );
translate([-0.4*length,-6,1]) linear_extrude(2)text("Door", 6);
```

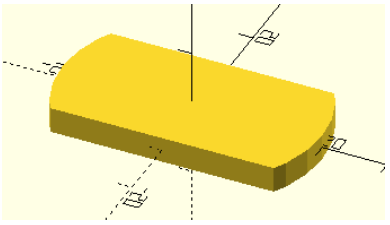
There is quite a lot going on here, so we'll now build it up piece by piece, starting with a simple plate. For the first time, we shall use a *variable* to create a named value that we can use in several places. The advantage of using variables is that we can change the value of the variable and then that new value will be used everywhere. This allows us, for instance, to scale models up and down with a single change.

```
length=20;

    cube([length*2,length,4], center=true);
```



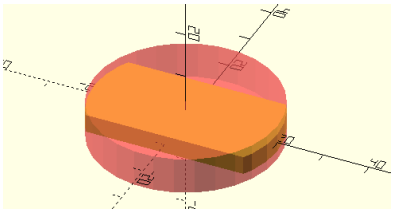
We make the curved ends by taking the *intersection* between a cylinder and the plate:

```
length=20;

  intersection() {
    cube([length*2,length,4], center=true);
    cylinder(d=2*length, h=8, center=true);
  }
```



When working with the `difference()` and `intersection()` operations it can be very helpful to see the 'other' object. Adding a # character in front of a an object causes it to be drawn as a ghostly image:

```
length=20;

  intersection() {
    cube([length*2,length,4], center=true);
    # cylinder(d=2*length, h=8, center=true);
  }
```

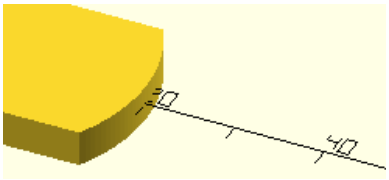


If you remove the # and re-render, the ghost image will go away.

If you look closely, you'll notice that the curve on the end of the fob is faceted. The objects that OpenSCAD makes are really just assemblies of triangles: two triangles make a rectangle, and a series of rectangles angled together makes the wall of a cylinder, and so on.

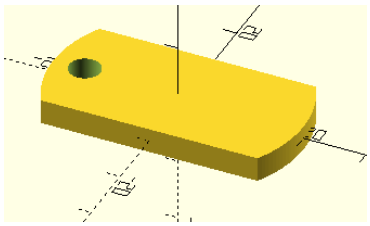
The amount of time required to render a design depends on the total number of triangles in it, so OpenSCAD by default uses rather large triangles. Designs then render quickly, but can look a bit chunky. When you have a design that you are happy with (and before printing!) you can increase the number of facets in a complete circle using a special variable called **`$fn`** Setting its value to, say, 180, gives a much smoother appearance. If you have a good graphics card, you may find that you can work with high values all of the time.

```
$fn=180;
length=20;

  intersection() {
    cube([length*2,length,4], center=true);
    cylinder(d=2*length, h=8, center=true);
  }
```



Next we put in the hole for the keyring, using the same approach as in the first example: take away a cylinder of the preferred diameter, moved (translated) to the correct coordinates:

```
$fn=180;
length=20;

difference(){
  intersection() {
    cube([length*2,length,4], center=true);
    cylinder(d=2*length,h=8,center=true);
  }
  translate([-0.75*length,0,0]) cylinder(d=5, h=8,center=true);
}
```



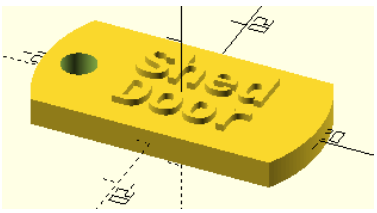
Now we can add the text. OpenSCAD can use any font that your system provides, but be aware that if you want your specification to work on all computers then using, say, Mac- or Windows-specific fonts may cause problems. The default font is a sans-serif form of lettering.

The **`text()`** directive lays text out as a two-dimensional object in the X-Y plane, that is the letters have no height in the z-plane. To give them height, we *extrude* them along the Z-axis. This simple specification asks for letters 6mm high, which are extruded 2mm up the z-axis.


```
linear_extrude(2)text("Shed", 6 );
```

So finally, we add the last two lines of the specification, extruding the text as above and moving the letters 40% of the way down the length of the key fob, written as **`-0.4*length`**

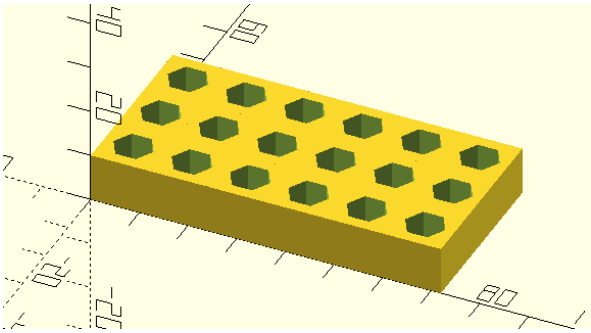
```
$fn=180;
length=20;

difference(){
  intersection() {
    cube([length*2,length,4], center=true);
    cylinder(d=2*length, h=8, center=true);
  }
  translate([-0.75*length,0,0]) cylinder(d=5, h=8, center=true);
}

translate([-0.4*length,1,1]) linear_extrude(2)text("Shed", 6 );
translate([-0.4*length,-6,1]) linear_extrude(2)text("Door", 6);
```



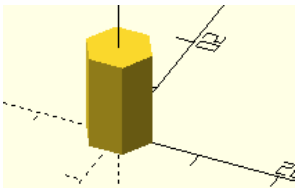
Example 4 – screwdriver bit tray

This last example illustrates reuse of sub-specifications and various forms of paramaterisation. The result is a *generator* for a tray to hold hex bits which can adjust itself to different bit spacings and capacities.

```
module bitBlock(size=12) {
  difference(){
    cube([size,size,10]);
    translate([size/2,size/2,2])
      cylinder(h=12, d=8.3, $fn=6);
  }
}

module tray(x=6, y=3, spacing=12) {
  for (y = [0:y-1])
    for (x = [0:x-1])
      translate([x*spacing, y*spacing, 0])
        bitBlock(spacing);
  }

tray();
```

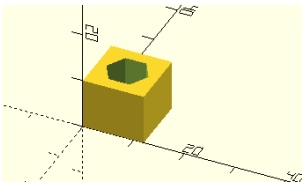


The starting point for this design in a 'cylinder' with six sides, which we construct by specifying `$fn=6` as part of the cylinder command. The 'diameter' of the hexagon (8.3mm) is slightly larger than the shank of a standard hex bit.



```
cylinder(h=12, d=8.3, $fn=6);
```

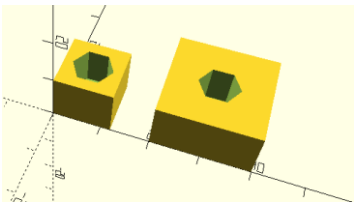
By subtracting this hex block from a cuboid, we get the basic building block of the bit tray:



```
size = 12;
  difference(){
    cube([size,size,10]);
    translate([size/2,size/2,2])
      cylinder(h=12, d=8.3, $fn=6);
  }
```

The key to managing complexity in any kind of system – mechanical, electronic, software – is *design reuse*. The trick is to package up a bit of the design into an independent assembly or sub-design which has a name, and might have some parameters, like a size, or a colour. Once we have packaged it up, we can ask for as many copies as we like, and we can use the parameters to customise the copies.

Here is one way of packaging up and then using the basic element we have just designed:



```
module bitBlock(size=12) {
  difference(){
    cube([size,size,10]);
    translate([size/2,size/2,2])
      cylinder(h=12, d=8.3, $fn=6);
  }
}
bitBlock();
translate([20,0,0]) bitBlock(20);
```

In OpenScad, the packaging device is called a module. It must have a name, and in parentheses a (possibly empty) list of parameters. Following that the *body* of the module is written within braces { }.

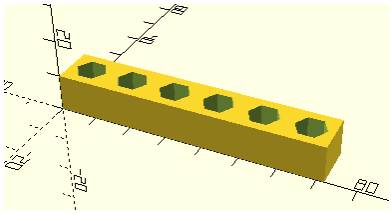
So, we can package up the basic element into an independent sub-assembly by enclosing it in braces, and prefacing it with the word **`module`** followed by a name (I have chosen **`bitBlock`**) and then the parameter list. In this case I have taken the size of the block which was already setup using the variable **`size`**, and simply moved that into the parameter list.

Names must start with an alphabetic character. They can then continue with a sequence of digits or alphabetic characters or the underscore character _ You may not have spaces or non-alphanumeric characters in a name apart from the underscore character. In addition, you must avoid using any names that are built into OpenSCAD such as, for instance, the word **`module`**.

A module definition does not actually create any objects in the rendered output. To do that you have to *call* the module definition. I have made two calls. The first one `bitBlock();` creates a 'standard' element at coordinates (0,0,0) (the origin). If I supply a numeric argument to the call then I can change its size. The second call `translate([20,0,0]) bitBlock(20);` makes an element that is 20mm on a side instead of the default 12mm. I have used a **`translate`** command to move it over by 20mm so that it doesn't get created on top of the first one.

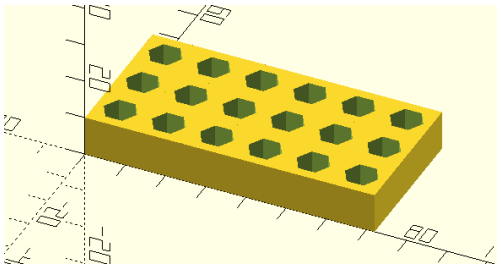
If a lot of copies are required, it would get very tedious to have to call them up individually. We can use a *for loop* to get multiple copies. So, for instance, this specification says to take a variable called xx, and set it to the sequence of values 0, 1, 2, 3, 4, 5. For each element of the sequence, call the **`bitBlock`** module and **`translate`** it so that its x coordinate is a multiple of the size of the block:

```
spacing = 12;
    for (xx = [0:5])
      translate([xx*spacing, 0, 0])
        bitBlock(spacing);
```

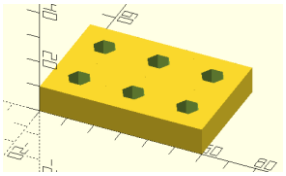


Now we can make another module which assembles together the **`bitBlock`** modules into a two-dimensional array:

```
module bitBlock(size=12) {
  difference(){
    cube([size,size,10]);
    translate([size/2,size/2,2]) cylinder(h=12, d=8.3, $fn=6);
  }
}

module tray(x=6, y=3, spacing = 12) {
  for (yy = [0:y-1])
    for (xx = [0:x-1])
      translate([xx*spacing, yy*spacing, 0])
        bitBlock(spacing);
  }


tray();
```



This new module has three parameters which set the size of the tray in terms of the number of bits it can accommodate in the X and Y directions, and the spacing between the holes. The defaults are for a 6x3 tray with 12mm between holes.



We could instead specify `tray(3,2,20);` to get thus lower density version. The key idea of this *paramterisation* is that with careful design we make modules which can adapt to different needs, perhaps by changing the scale of a model, or altering the number of planks in a wagon.