The first and most obvious question is -what is a computer program? The answer varies from programmer to programmer -but this is mine. “A computer program is one of only the two commercial applications of Poetry”. Both deal with how words are used and how they impact the reader, (or computer). Poems can small like Haiku or long like the Head Ransom Ballard. Note that both the Haiku and the Head Ransom follow rigid rules of construction. This is the SYNTAX of the program. There are many ways to express a line of Poetry or English for instance:

“Her hair was the colour of corn in the autumn” or “It was brown”. Both mean the same thing…

Just like Poetry a program should break up into Verses and Choruses or MODULES. This is written in the style called GRAPES. Which is supposed to look like grapes hanging off a vine. This is my normal “go to” style of editing a program.

```
//canopy angle bracket
//
//print one bracket

cube([50,2.5,5]);
translate([2,2,0])
{
	rotate([0,0,135])
	{
		cube([50,2.5,5]);
	}
}
;
translate([50,2.5,0])
{
	rotate([0,0,157])
	{
		cube([90,2.5,5]);
	}
}
;
translate([5,11,0])
{
	difference()
	{
		cylinder(2.5,9.5,9.5);
		cylinder(2.5,7,7);
	}
}
;
//END
```

Each statement exists on it own line. Thus when a error is flagged it CAN only be that statement. Where a command is used then those things affected by it are surrounded by “braces” **{** and **}** and

running your finger down the screen from an open **{** to a closed **}** shows the lines affected by this command. Each “verse” of the program has a proper terminator - a semicolon **;** this shows where it is safe to yank this module and place it somewhere else. The program includes a proper **//END** statement to show you that you have ALL of the program!

There might be a time you wish to make multiple copies of a module. Here I have simply copied one module and rubber stamped it until I have enough vertical bars.

```
// domestic gate

//gate posts
cube([10,60,10]);
translate ([80,0,0])
{
        cube([10,60,10]);
}
;

//upper and lower bars
translate([0,10,0])
{
        cube([90,10,3]);
}
translate([0,40,0])
{
        cube([90,10,3]);
}
;

//pickets
translate([15,0,0])
{
        cube([5,55,5]);
}
;
translate([25,0,0])
{
        cube([5,55,5]);
}
;
translate([35,0,0])
{
        cube([5,55,5]);
}
;
translate([45,0,0])
{
        cube([5,55,5]);
```

```
}
;
translate([55,0,0])
{
	cube([5,55,5]);
}
;
translate([65,0,0])
{
	cube([5,55,5]);
}
;
translate([75,0,0])
{
	cube([5,55,5]);
}
;
//END
```

These can be repeated and copied between programs and repeated as required for number of dependants (there is a fixed number or you run out). This is called ITERATION. The standard method of doing a fixed number of repeats is called “The For – Next Loop”. This program produces an Edwardian brick wall. There are three For-Next loops in it called i j and k. The first puts the bricks into the wall the second raises the layer of the brick height.

```
// Flemish Bond brick wall. Edwardian brick size.

//Has weeping holes
//bricks 9mm x 7.5mm x 3mm with 0.5mm morter gap
//Print 6 rows of 10 stretchers and 10 headers
//Flemish Star by offset alternate rows by -7.
rotate([90,0,0])
{
	for (k=[0:7:70]) //number of rows of bricks x 2
	{
		for (j=[0:20:19])
		{
			for (i=[0:14.5:145])
			{
				translate([i,j,k])
				{
					cube([9,5,3]); // a stretcher brick
				}
				translate([(i+9),j,k])
				{
					cube([0.5,4,4.0]); //morter between stretcher brick
				}
```

```
                                translate([(i+9.5),j,k])
                                {
                                        cube([4.5,5,3]); //a header brick
                                }
                                translate([(i+14),j,k])
                                {
                                        cube([0.5,4,4.0]); //morter between header brick
                                }
                        }
;
        translate([0,0+j,3+k])
        {
                cube([160,4,0.5]); // morter the length of bricks
        }
;
//stagger next row by 7 to produce "star"
        for (i=[-7:14.5:138])
        {
                translate([i,0+j,3.5+k])
                {
                        cube([9,5,3]); // a stretcher brick
                }
                translate([(i+9),0+j,3.5+k])
                {
                        cube([0.5,4,4.0]); //morter between brick
                }
                translate([(i+9.5),0+j,3.5+k])
                {
                        cube([4.5,5,3]); //a header brick
                }
                translate([(i+14),0+j,3.5+k])
                {
                        cube([0.5,4,4.0]); //morter between brick
                }
        }
;
        translate([-7,0+j,6.5+k])
        {
                cube([160,4,0.5]); // morter the length of bricks
        }
}
}
}
;
//END
```

Having mastered an easily corrected and productive style the question you now have to ask is : what do I do with it and how do I manipulate it?

OpenScad is unique in that it is almost a copy of PICNIC which is an old CAD/CAM language for punched tape NC machines to use. It was familiar enough for programmers who normally wrote the company accounts and personnel systems to transfer to production machinery side. There are “families” of computer languages some of which have grown and died and some well -should have been murdered at birth… Some flourish in new forms. BASIC and PASCAL were both written to enable the teaching of computer languages. OpenScad is half way between the two and is purely text based. Thus there is no building of a shape on screen as with Inkscape, Bryce or Turbo CAD.

There are no “primitives” -except those you make yourself.

There is a book I would recommend you to look at. Although it is not OpenScad or even a book on a common computer language -but download a copy of Leo Brodies’ “Breaking Forth” and “Thinking Forth”. I have always found them good reading and a good laugh!

My “PC” uses Debian 10 Linux and yes, some parts of the OS I have written myself.

Regards

Ralph