Designing with OpenSCAD

A Young Lady's First Enchiridion

Wil Chung

Designing with OpenSCAD

A Young Lady's First Enchiridion

Wil Chung

This book is for sale at http://leanpub.com/designing_with_openscad

This version was published on 2015-03-30

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Chapter 5: Modeling the Engine

We're going to make parts to fit onto the model rocket engine. So that means we're going to have to make a 3D model of it. It doesn't need to be very detailed, since we're not going to be printing it, but it has to be the right size.

Measuring the engine

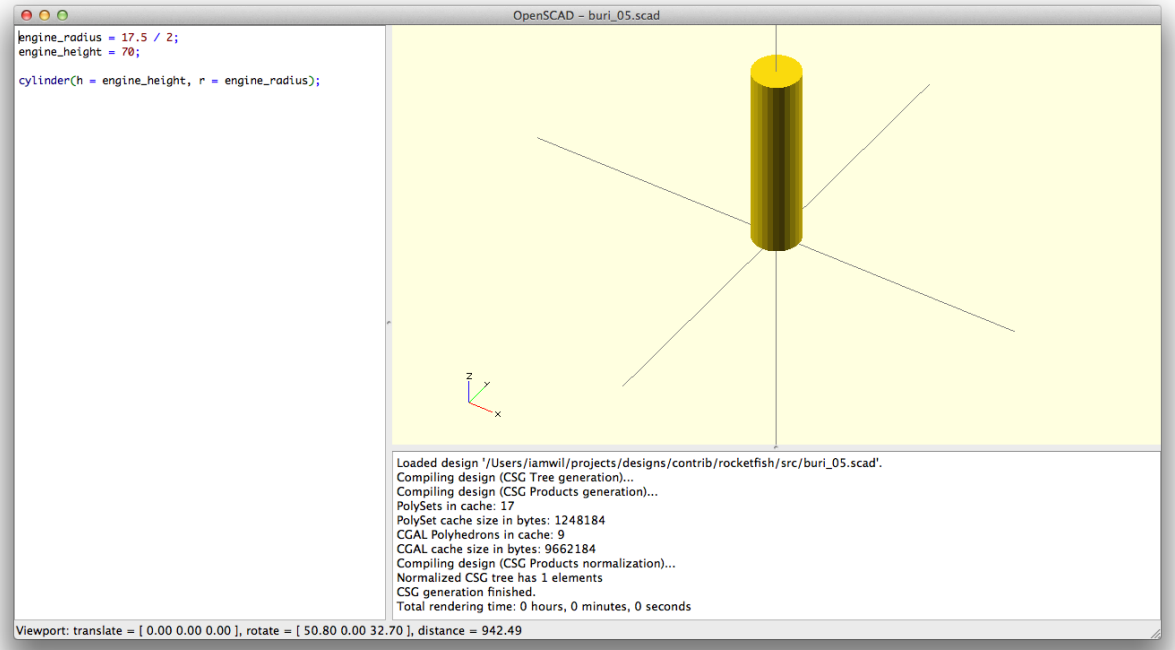
We can use calipers to measure the engine. We'll be using an A8-3 engine. The diameter is 17.5mm and it's 70mm long. We can model it as a cylinder.

Modeling the engine

Let's start by opening OpenSCAD and starting in the editor. Write the following code in the editor and save the file under `/src/buri.scad`

Once you've saved the file, you can see it render in the model side. You should see a cylinder.

```
1  // buri.scad
2
3  engine_radius = 17.5 / 2;
4  engine_height = 70;
5
6  // the engine
7  cylinder(engine_height, engine_radius, engine_radius);
```



engine

We model the engine as a cylinder. The first parameter is the height of the cylinder. The next parameter is the radius of the cylinder at the bottom. The last is the parameter of the radius at the top of the cylinder.

And that's it! It's pretty easy to make the model of the engine.

So for now, we'll comment out the engine, so it doesn't show when we create the nose cone. To comment something out, we put `//` in front of a line. That way, it doesn't get executed, and is treated as a note or annotation in the code.

```
1  // the engine
2  //cylinder(engine_height, engine_radius, engine_radius);
```

Summary

- We learned how to make a cylinder with variables
- We learned how to comment out a cylinder

Full Source

```
// buri.scad

engine_radius = 17.5 / 2;
engine_height = 70;

// the engine
//cylinder(engine_height, engine_radius, engine_radius);</b>
```

Chapter 8: Modules - Refactoring the engine as a module

Our code for the nose cone is a little unwieldy now, with lots of details. Having lots of details exposed makes it harder for our brain to handle it.

To help mitigate the effects of our tiny brain, we'll learn how to organize and modularize our code. We do this because:

1. We want to make the code easier to use.
2. We want to make the code easier to understand.
3. We want to make the code easier to construct.

All of this helps keep the code maintainable, because chances are, in the future, YOU'RE going to be the one that has to change it. Do your future self a favor! Make you code clean and readable.

Engine as a basic module

A module is a piece of code that we can write once, and use over and over again. Let's make the engine into a module. This means we define what an `a8_3_engine()` means, and then we call it when we want to use it.

Let's make the engine a module. We can define a module like so:

```
module a8_3_engine() {
  cylinder(engine_height, engine_radius + tol, engine_radius + tol);
}
```

The name of the module is `a8_3_engine()`, and the code inbetween the curly braces `{` is the definition of the module.

And we would be able to call the module by writing:

```
a8_3_engine();
```

We can put the definition of the module at the top of the file, and call the module where we had called the cylinder before. That means the code now looks like:

```
...

nose_assembly_thickness = 1.2;
nose_assembly_slot_height = 5;

module a8_3_engine() {
  cylinder(engine_height, engine_radius + tol, engine_radius + tol);
}

difference() {
  // the nose cone
  ...

  translate([0, 0, nose_assembly_slot_height]) {
    mirror([0, 0, 1]) {
      // the engine
      // remove the '#' that was in front of the cylinder()
      a8_3_engine()

      // the inner dome
      sphere(engine_radius + tol);
    }
  }
}
```

Why use modules?

This change might seems too simple to go to the trouble of doing, but what we're trying to learn here are the building blocks of how to make our code easier for humans to read and maintain.

Despite what you might think, code is meant to be read by humans, and incidentally executed by computers. Code is also read far more than it is written. Lastly, our brains are limited by how much they can hold.

Just as an industrial designer crafts tools to fit the curve of a human hand for greater comfort and effectiveness, when we write code, we should write it to enhance the strength of the human mind and shore up its weaknesses.

By using modules, we get some advantages that help fit the code to the curve of our minds.What are these advantages?

Our code is easier to use

Which is easier? Driving a car by pushing down the gas pedal with your foot, or timing the firings of the pistons in the engine? It's the former, because it's easier to do. We're much better at moving our foot than at pushing buttons at sub-millisecond intervals regularly.

In the same way, now that we have a reusable module, we can call with `a8_3_engine();` over and over again any time we want an engine! No longer will we have to remember to pass in the dimensions of the engine.

This is called abstraction, where we hide the *appropriate* details of an implementation away from the user of the module. That way, they can concentrate on high-level issues, rather than low-level details.

Our code is easier to understand

When we call modules in our code, we're effectively communicating our intent to any future readers of the code. Using clear module names, we can semantically understand what's being built. It's much easier to see that:

```
1  a8_3_engine();
```

means that we're constructing an engine here. It's not as obvious when looking at this:

```
1  cylinder(engine_height, engine_radius + tol, engine_radius + tol);
```

That's especially true when you come back to the code after not looking at it for months. And it becomes even more true when the module is much more complicated, with lots of translations, rotations, sphere, and cubes.

When we make it easier for us to read what's being written, it makes our code more maintainable.

Our code is easier to construct and change

When we use a module to represent an engine, we also make the code easier to maintain.

If in the future, there are changes to what an a8_3_engine looks like, we have exactly one place where we can change what `a8_3_engine();` means–in the definition of the a8_3_engine module.

Using modules is better than cutting and pasting `cylinder(engine_height, ...)` eveywhere we want to create an engine, because if you want to model the engine differently, you now have multiple places in the code that need to be changed. That process can cause headaches and be error prone.

Summary

We learned:

- How to create a module
- How to call and use a module
- Why we would want to use modules

Full Source

```
//buri.scad

tol = 0.4;

engine_radius = 17.5 / 2;
engine_height = 70;

nose_assembly_thickness = 1.2;
nose_assembly_slot_height = 5;

module a8_3_engine() {
  cylinder(engine_height, engine_radius + tol, engine_radius + tol);
}

difference() {
  // the nose cone
  difference() {
    scale([1, 1, 3]) {
      sphere(engine_radius + nose_assembly_thickness);
    }
    translate([0, 0, -4 / 2 * engine_radius]) {
      cube(4 * engine_radius * [1, 1, 1], true);
    }
  }

  translate([0, 0, nose_assembly_slot_height]) {
    mirror([0, 0, 1]) {
      // the engine
      // remove the '#' that was in front of the cylinder()
      a8_3_engine()

```

```
32        // the inner dome
33        sphere(engine_radius + tol);
34      }
35    }
36  }
```

Chapter 10: Modules - Modules that change its children

Modules can be used in different ways. So far, we've only used it to represent objects and assemblies. However, we can use it to it modify objects, much like `translate()` and `rotate()` do.

That way, we can not only build object assembly abstractions, but also object transformation abstractions to help the maintability and clarity of our code.

Using resize for tolerance

We're going to refactor tolerance into a module that modifies objects. When we look inside of `a8_3_engine()`, we see that tolerance adjustments aren't inherent to the concept of a `a8_3_engine()` module. Hence, we want to separate those two concerns into two different modules.

First, we'll introduce `resize()`

```
module a8_3_engine() {
  resize([engine_radius * 2, engine_radius * 2, engine_height] + tol * [1, 1, 1]\
) {
    cylinder(engine_height, engine_radius, engine_radius);
  }
}
```

`resize` resizes the child objects (in this case, the cylinder) to dimensions specified. What gets passed into resize is an array for the dimensions in each of the x, y, and z directions. As a result, notice that the elements of the resize array is in a different order than the arguments for cylinder.

What we passed into resize is a bit of array math. It's simple to follow.

```
[engine_radius * 2 + tol, engine_radius * 2 + tol, engine_height + tol]
 = [engine_radius * 2, engine_radius * 2, engine_height] + [tol, tol, tol]
 = [engine_radius * 2, engine_radius * 2, engine_height] + tol * [1, 1, 1]
```

We originally had the top array, and we can reduce it step by step to the last one. So by passing it in, we're increasing every dimension of the cylinder by an extra `tol`

Making the with_tol() module

We're going to abstract the tolerance adjustment into a module. Because OpenSCAD doesn't let us query the size of an object, we'll need to pass it in ourselves.

Let's create a module called `with_tol()`.

```
module with_tol(size) {
  resize(size + tol * [1, 1, 1]) {
    children();
  }
}
```

Notice this is similar to what we had before, but now, size is passed into `with_tol()`, and we resize `children()`, which are the child modules under `with_tol()`

To use it, we now rewrite `a8_3_engine()` as:

```
module a8_3_engine() {
  with_tol([engine_radius * 2, engine_radius * 2, engine_height]) {
    cylinder(engine_height, engine_radius, engine_radius);
  }
}
```

Notice that the array we pass into `with_tol()` is what's referred to as `size` inside of `with_tol()`. The `cylinder()` (and everything inside the curly braces of `with_tol`) that's used as a child module under with_tol() is referred to as `children()` inside of `with_tol()`.

Making the chop() module

Now that we know how to make modules that operate on children, we're going make other abstractions. This time, we want a module that chops off an entire side of a plane of a model, which is what we did to make the parabolic cone.

```
module chop(r, plane) {
  difference() {
    children();

    theta = plane[0] == 0 ? 0 : atan(plane[1] / plane[0]);
    phi_abs = atan(sqrt(pow(plane[0], 2) + pow(plane[1], 2)) / plane[2]);
    phi_sign = plane[0] > 0 ? phi_abs : -phi_abs;
    phi = plane[2] >= 0 ? phi_sign : phi_sign + 180;

    // cut off below plane
    rotate([0, 0, theta])
      rotate([0, phi, 0])
        translate(-r * [0, 0, 1])
          cube((2 * r) * [1, 1, 1], true);
  }
}
```

Once again, since OpenSCAD doesn't allow you to query the size of the children, we'll need to pass it into `chop()`. That's what the 'r' parameter is, where 'r' is the largest dimension of the children modules. `plane` is the normal vector to the plane that you want to chop the module.

The calculations for the spherical coordinate angles, theta and phi, where theta is the angle between X and Y dimensions, and phi is the angle between the hypotenuse (of X and Y) and the Z dimension. Because arctangent is only valid between 90 and -90, we need to change phi depending on the signs of the X dimension of plane and the Z dimension of plane.

In order to do so, we need to use the tertiary form of an if statement. It takes the form:

```
[_condition_] ? [_if condition is true_] : [_if condition is false_]
```

Because OpenSCAD is a *declarative functional language*, you cannot assign a variable more than once. In fact, it's much better to think of variables in OpenSCAD as overridable constants. Hence, we'd need to use a teriary statement to decide what the value should be, and only assign it to a variable once.

```
// this is wrong
if (plane[0] == 0) {
  theta = 0;
} else {
  theta = atan(plane[1] / plane[0]);
}
```

```
// this is right
theta = plane[0] == 0 ? 0 : atan(plane[1] / plane[0]);
```

And now, we can use `chop()` like so:

```
module parabolic_cone(height, radius) {
  chop(10, [0, 0, 1]) {
    scale([1, 1, height / radius]) {
      sphere(radius);
    }
  }
}
```

That greatly simplifies `parabolic_cone()`, and we can reuse `chop()` in other places of the code.

Summary

We learned

- Creating modules that modify its children
- How to use resize()
- Tertiary operator for if statements
- How OpenSCAD cannot assign a variable more than once

Full Source

```
tol = 0.4;

engine_radius = 17.5 / 2;
engine_height = 70;

nose_assembly_height = 26.25;
nose_assembly_thickness = 1.2;
nose_assembly_slot_height = 5;
nose_assembly_radius = engine_radius + nose_assembly_thickness;

module with_tol(size) {
  resize(size + tol * [1, 1, 1]) {
    children();
  }
```

```
}

// chops off model on entire side of plane
module chop(r, plane) {
  difference() {
    children();

    theta = plane[0] == 0 ? 0 : atan(plane[1] / plane[0]);
    phi_abs = atan(sqrt(pow(plane[0], 2) + pow(plane[1], 2)) / plane[2]);
    phi_sign = plane[0] > 0 ? phi_abs : -phi_abs;
    phi = plane[2] >= 0 ? phi_sign : phi_sign + 180;

    // cut off below plane
    rotate([0, 0, theta])
      rotate([0, phi, 0])
        translate(-r * [0, 0, 1])
          //with_tol((2 * r) * [1, 1, 1])
            cube((2 * r) * [1, 1, 1], true);
  }
}

module a8_3_engine() {
  with_tol([engine_radius * 2, engine_radius * 2, engine_height]) {
    cylinder(engine_height, engine_radius, engine_radius);
  }
}

module parabolic_cone(height, radius) {
  chop(10, [0, 0, 1]) {
    scale([1, 1, height / radius]) {
      sphere(radius);
    }
  }
}

difference() {
  // the nose cone
  parabolic_cone(nose_assembly_height, nose_assembly_radius);

  translate([0, 0, nose_assembly_slot_height]) {
    mirror([0, 0, 1]) {
      // the engine
```

```
57        // remove the '#' that was in front of the cylinder()
58        a8_3_engine()
59
60        // the inner dome
61        sphere(engine_radius + tol);
62      }
63    }
64  }
```