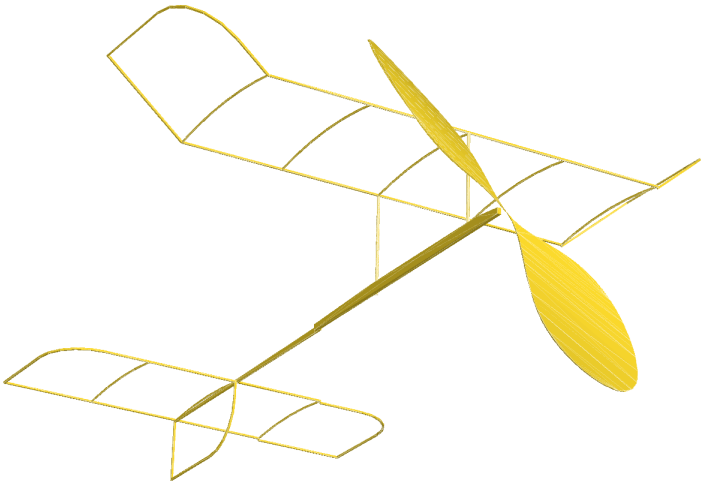
Designing an Indoor Model using OpenSCAD

Roie R. Black

January 15, 2021



Designing a new model airplane usually involves generating a plan of some sort, then constructing a prototype model from that plan. Of course you can use a pencil and paper to generate your plans, but if you think you might want to publish the plan, you will need to use some form of *Computer Aided Design* tool to produce your final plan. Unfortunately many popular CAD tools are complex, and often too expensive for the average modeler.

Having recently retired from teaching Computer Science, and finally getting back into model building, I decided to design a new indoor model for the *Limited Pennyplane* class. As part of the design process, I wanted to see that airplane in 3D even before I built the first prototype. I decided to use a different form of CAD tool: OpenSCAD [5], a tool designed for computer programmers!

While that description may discourage some folks from reading further, rest assured that this particular tool is simple enough that non-programmers can certainly master it. In fact, some teachers have successfully managed to get elementary school kids to use OpenSCAD to design simple 3D models.

OpenSCAD is an open-source (meaning free) 3D modeling program, available on all major platforms. It is commonly used by folks designing parts to be printed on 3D printers. What makes OpenScad different is how you generate the design. Instead of using your mouse to drag things around on the screen, you describe your model in a simple programming language. Formally, OpenSCAD uses something called *Constructive Solid Geometry* to construct your model, then gives you a visual interface you can use to examine your 3D model in detail.

I will only show example code from the project so you can get a feel for the design process. You are encouraged to explore the project website for much better documentation and complete source code. [4].

OpenSCAD

Installing OpenSCAD is oretty simple using instructions found on the projects website. Once it is installed, open it up and take a look at the basic interface:

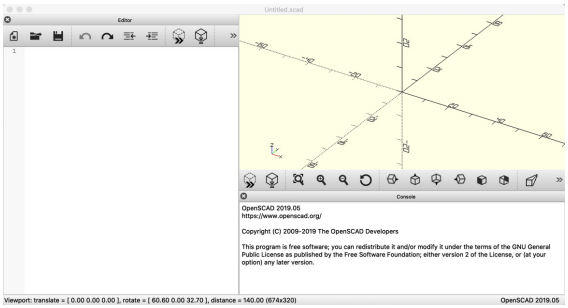


Figure 1: OpenSCAd interface

There are three areas we will be using in this view:

- Editor - the left panel where you type in your code.
- Preview - the top right panel will be where your model is displayed.
- Messages - the bottom right panel is where error messages will be shown when processing your code.

I will not try to show everything you need to know about the language *OpenSCAD* uses for describing a model. Instead, I will show fragments of code to give you a feel for what you need to write to design your model. The project website [4] has more details, as does the *OpenSCAD User Manual* [**?**].

A Handy "cheat-sheet" is available here: [2].

Organizing Your Code

OpenSCAD lets you write your code in one file or in multiple files. I like to split up a design into multiple files in order to keep them short and focused on one just part of the design. If you split things up you will need to use either an *include* or a *use* line specifying the file you want to access with the code in the present file. If you choose *include*, all that code in the second file will be processed as though it had been typed in the current file. On the other hand, using the *use* line only makes the names from the second file available in this one. The code in the second file will only be processed when those names are encountered in the current file.

A typical setup is to create a single file with variables you want every piece of code to be able to use. You *include* that file like so:

```
include <math-magik-data.scad>
```

Another example might be in a file defining the wing for this design. That file needs to use ribs, which I define in another file. In my wing file I would add this line

```
use <rib.scad>
```

You will see a lot of this notation in the full project code files on the project website.

The Designng Process

Building a 3D model is a trial and error process. You type in or modify your code, then click on a command to process that code. You look at the preview window to see your model, and search the message area for hints about what went wrong. Non-programmers will find this a bit frustrating, but this takes practice to master, so do not get discouraged. My advice is to always take small steps. That limits the number of problems you face in getting things to work.

The best way to learn anything new is to experiment. Beginning programmers are always searching the Internet for solutions they can copy into their problems, but the only thing you actually learn when copying and pasting stuff is how to copy and paste. You will learn far more by typing in code yourself - at least until you get more proficient at this. Reading the code

gives you a chance to really think about what is going on. There is nothing wrong with looking at code written by others. Many times studying that code will teach you how to better write your own code. I will show you enough code in this design to give you a feel for how you do things using *OpenSCAD*. The actual code I generated for this design is on the project Github account [4].

I highly recommend building small files that generate one part of your overall design. Test that component until you are sure it looks like what you want. Then use that part in building other components. I like to work from small parts up to bigger assemblies, and that is how we will work through this design. Don't be afraid to fire up *OpenSCAD* and try things are you read this article. Of course, you should look at the project website to see all the code in greater detail.

Constructive Solid Geometry

OpenSCAD builds 3D models using a small set of primitive shapes, and a set of movement and combining operations to create more complex models.

Primitive Shapes

Openscad supports both 2D and 3D shapes. We will be using some simple 2D shapes, like circles and rectangles, and more complex 2D shapes like a polygon. The 3D shapes we will use include spheres, cylinders, and cubes. All of these shapes can be scaled and moved around using simple movement operations.

3D Primitives

For our first look at how you do things in OpenSCAD, here is a piece of code that will show the three basic 3D shapes:

Listing 1: *demo/demo*1.*scad*

```
cube();
translate([3,0,0])
  sphere();
translate([6,0,0])
  cylinder();
```

Figure 2 shows the result.

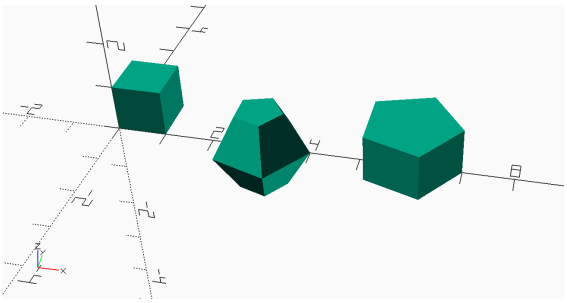


Figure 2: Demo 1

Each primitive shape is created at the origin. Cubes are created in the region where all three coordinates are positive. Spheres are created with the center of the sphere at the origin. The cylinder is centered along the **Z** axis. If you look closely, you will see a small representation of the coordinate directions at the lower left of this image.

We used a *translate* operation to move shapes aside so they do not overlap. The numbers inside square brackets control the distance we want to *translate* the following shape in the **[x,y,z]** directions. This bracketed group of numbers is is called a *vector* which we will use a lot in our work.

Notice how I indent code to show how things happen. In this example, we *translate* the following *cube* shape. The semicolon ends this command. Failing to put semicolons where they are needed is a common mistake when writing *OpenSCAD* code.

These shapes do not look quite right. The problem is that *OpenSCAD* generates approximations to the rounded shapes, using a set of small polygons to build up the model. If we make these polygons smaller, things look better. All we need to do to fix this is change the code so it looks like this:

Listing 2: *demo/demo*2.*scad*

```
cube();
translate([3,0,0])
  sphere($fn=100);
translate([6,0,0])
```

```
  cylinder($fn=100);
```

Figure 3 shows a much better result. The special variable *$fn* controls the resolution of rounded objects. Bigger numbers make things look smoother but cost of longer times to generate images on the screen.

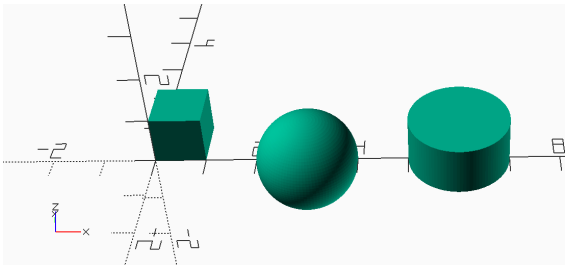


Figure 3: Demo 2

Some shapes are smart and can form different versions of themselves:

Listing 3: *demo/demo3.scad*

```
cube([1,3,1]);
translate([4,0,0])
  sphere(r=2, $fn=100);
translate([8,0,0])
  cylinder(
    r1=1, r2=0.25, $fn=100
  );
```

Figure 4 shows a warped cube and cylinder. Spheres are not so smart, they stay spheres unless we warp them with external commands.

2D primatives

We will use a few 2D shapes in this design, including the circle and square, which act much like their 3D counterparts. A more interesting 2D shape we will use is the *polygon*.

Listing 4: *demo/polygon – demo1.scad*

```
triangle_points =[
  [0,0],[100,0],[0,100],[
  10,10],[80,10],[10,80]
];
triangle_paths =[
  [0,1,2],[3,4,5]
];
polygon(
  triangle_points,
  triangle_paths,
  10
);
```

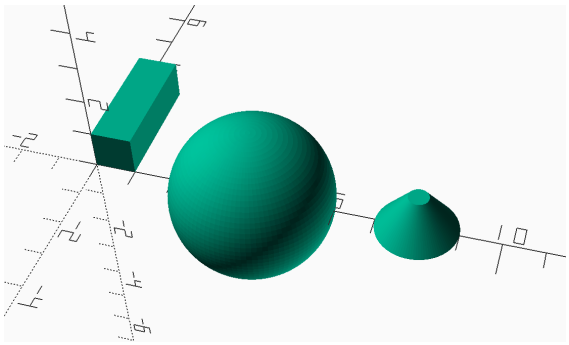


Figure 4: Demo 3

Here, we create two *variables* and set them equal to a list of vectors. 2D vectors have only 2 numbers, for the **X** and **Y** coordinate values. The first list defines a set of six points: three for the outer triangle, and three more for the inner triangle. The second list identifies *paths* meaning a continuous line that makes up a closed circuit, one for the outer triangle, and one for the inner triangle. The numbers refer to the position of vectors in the first list (programmers count starting at zero!) That final **10** parameter is not important here, it helps the operation work properly.

I know this is a bit confusing, but we will not need much of this kind of code in our design work. Remember to try things and see what happens.

Figure 5 shows a 2D shape with no thickness, although *OpenSCAD* gives it enough of a thickness to show up on the screen.

We can use this 2D shape to create a 3D object by *extruding it* in the **Z** direction:

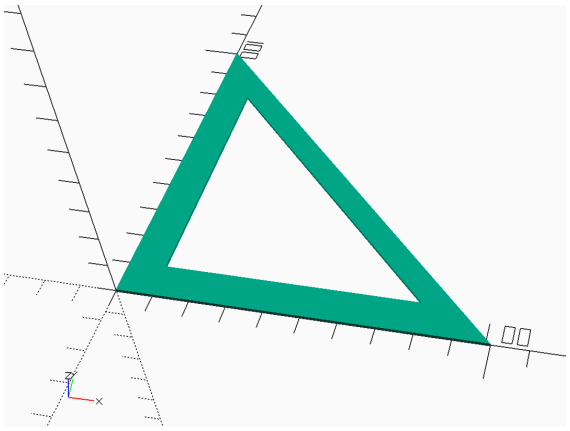


Figure 5: Polygon Demo 1

Listing 5: *demo/polygon – demo2.scad*

```
triangle_points =[
  [0,0],[100,0],[0,100],
  [10,10],[80,10],[10,80]
];
triangle_paths =[
  [0,1,2],[3,4,5]
];
linear_extrude(h=1)
    polygon(
      points = triangle_points,
      paths = triangle_paths,
      convexity=10
  );
```

Figure 6 definitely shows an interesting shape. We will use *extrusion* to make some parts that would be difficult to construct with just the basic primitive shapes.

Obviously, we can form some interesting things with *OpenSCAD*. But things get even more interesting when we start combining multiple shapes to form more complex objects.

Movement Operations

We saw the *translate* operation earlier. We can also *rotate* a shape. In this command we provide a vector of angles (in degrees) that we want to use to rotate the shape. Each number in the vector will be used to rotate the shape around the coordinate axis associated with that number. For instance **rotate([90,0,0])** will rotate the shape around the **X** axis ninety degrees. This operation uses the *right-hand* rule. If you want to rotate around the bf X axis, take your right hand and point the thumb in the direction of increasing **X** in your coordinate system. Your fingers "curl" around that axis in a positive direction.

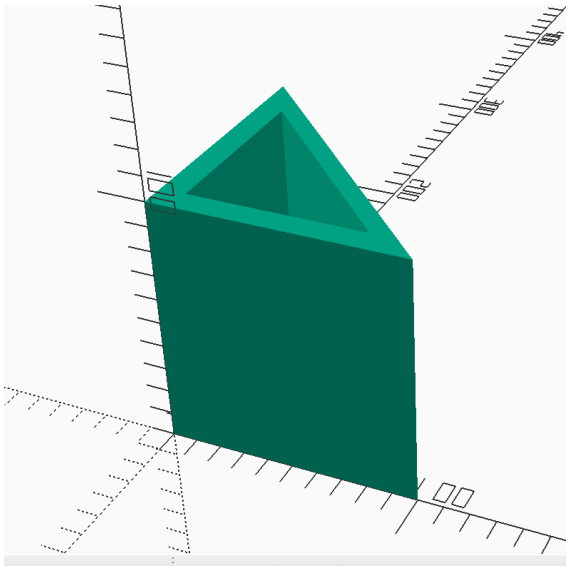


Figure 6: Polygon Demo 2

Combining translations and rotations is done by writing both commands like this:

```
translate)[10,15,0])
  rotate[90,0,0])
    cube(1,1,5);
```

It helps to read this bottom up. We are creating a *cube* at the origin. We rotate it so it is aligned the way we want, then we translate that result to the position we have chosen. The semicolon at the end of this list ends the command. Notice that I indent so show what I want my code to do.

Be warned that you can swap the *translate* and *rotate* commands, but you might not get the result you expect. Rotations are applied to the shape as it is positioned when the command is processed. If you rotate after translating, The shape will swing a long way!

0.1 Combining Operations

We form more complex objects by moving things around and combining them to form new objects. An example found in the *Wikipedia* article on CSG [1] demonstrates these operations.

Suppose you wanted to build something that looks like Figure 7.

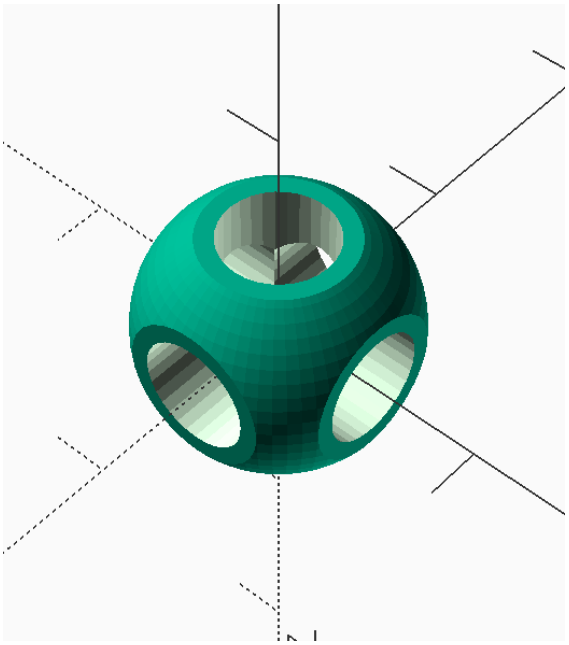


Figure 7: CSG Example Shape

We can form this shape using three cylinders, a sphere, and a cube. We use all three basic combining operations to construct the final shape.

Here is the OpenSCAD code used to generate this shape:

Listing 6: *demo/csg – demo.scad*

```
module core() {
  union() {
    cylinder(
      r=0.25,
      h=2,
      center=true,
      $fn=32
    );
    rotate([90,0,0])
      cylinder(
        r=0.25,
        h=2,
        center=true,
        $fn=32
      );
    rotate([0,90,0])
      cylinder(
        r=0.25,
        h=2,
        center=true,
        $fn=32
      );
  }
}

module round_cube() {
  intersection() {
    cube(
      [1,1,1],
      center=true
    );
    sphere(
      r=0.6,
      $fn=64,
      center=true
    );
  }
}

module part() {
  difference() {
    round_cube();
    core();
  }
```

```
}

core();
```

There is a point to be made here. We can move two objects together so they touch, like a rib to a spar, but we do not really need to join them together in this design work. Visually, things will look right, but the two objects remain separate. Joining them together to make a combined part would be important if we were going to 3D print the object. Since we do not have he technology to print with balsa (yet), I will not worry about combining the components of our design to create a single airplane object.

Modules

OpenSCAD lets you package a number of operations in a *module* that you can activate later, one or more times. In fact those primitive shapes were all predefined *modules*. The module can have parameters, which makes this a powerful way to manage shapes that are similar, but differ depending on the parameters you specify. We saw that when we showed "warped" shapes earlier. We will create a basic rib module for this model, and use parameters to control the exact rib we want.

All modules have a unique name in your code. The name you choose should help you remember what the module is all about. In this example, we are interested in the final **part** shape, which is constructed using the difference operation. This final module uses two supporting modules to build the part. You can write your code almost any way you like, but it is common to use spaces,indentations, and newlines to organize your code to make reading it easier. Also, we surround a sequence of individual operations inside of curly braces when needed. I always indent any code inside of these braces.

When you add parameters to a module, you define names for each one between the parentheses. Commas separate parameters if you have more than one. You can optionally provide a default value for each parameter by adding an equal sign followed by the default value you want. When you activate the module, you must provide actual values you want the module to use. You can just provide a sequence of numbers in the right order with commas separating them, or you can add the parameter name from the definition, an equal sign, then the new value you want. In this case, the order is not important, and you can leave off any parameters where you are happy with the default value. The rules for all of this are detailed in the *OpenSCAD User Manual* [**?**], so I will not go further in this discussion here.

Building the Example Shape

To build this part, we first set up three cylinders, aligned along each coordinate axis. The bf center parameter, sets each cylinder up with the origin of the coordinate system at the exact center of the cylinder. Remember, The **$fn=32** parameter is really only needed to make the cylinders actually look round.

Notice that all three of these cylinders occupy the same space. In the real world, we could not do that, but in our 3d modeling world this is common. We form the **union** of these three overlapping cylinders to form one merged shape.

The outer shell of our part is made up of the **intersection** of a sphere and a cube. We size the cube shape so it trims off six sides of the sphere where holes will end up. Finally, we use the **difference** operator to carve out the inside of the part, using our three-cylinder shape.

Successfully building 3D models involves visualizing what you want, then arranging simple shapes as needed and performing these three basic combining operations to generate the gadget you want! It takes practice! The more you experiment the better you will get!

I encourage you to fire up *OpenSCAD* and type in this code. You will be better able to see how things work by doing this!

Design Constraints

The *Limited Pennyplane* class rules define a few constraints on dimensions for our model. Specifically we must honor these limits:

- **max_wing_span** - 18”
- **max_wing_chord** - 5”
- **max_stab_span** - 12”
- **max_spab_chord** - 4”
- **max_length** - Max Prop to Tail length = 20”
- **max_prop_diameter** - 12”

What this means is that the model must fit in a box that measures **max_wing_span** wide by **max_length** long. There is no limit on how tall this box can be.

Furthermore, the wing must fit in a smaller box measuring **max_wing_span** by **max_wing_chord**. The stabilizer must fit in a similar box measuring **max_stab_span** by **max_stab_chord**. There are no constraints on where these boxes fit inside the outer box. The propeller is only limited by diameter, blade shape it up to the designer. However, the **max_length** constraint is measured from the forward-most point, usually on the propeller, to the aft-most point on the model. We could build a pusher, but I have not considered that idea.

Note: The labels in this diagram are abbreviations for the names shown above. In my code I will use full names to improve readability of the code. inputwing

Stabilizer and Fin

The code we created to build the wing provides everything needed to build the stabilizer and fin.

stabilizer

The stab is identical to the wing, except in dimension ans number of ribs. The modules used for the wing give us everything needed to build the stabilizer.

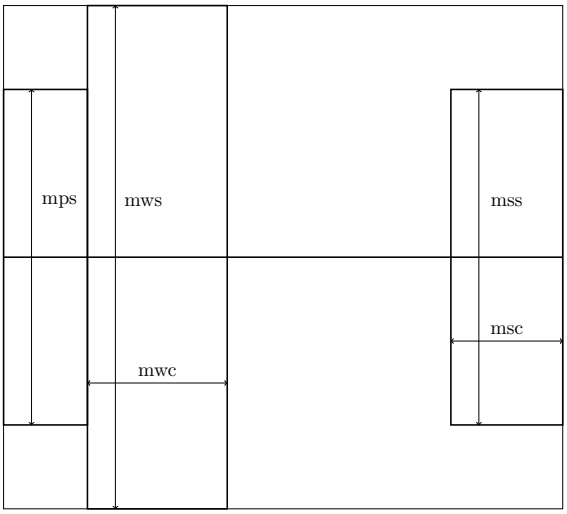


Figure 8: LPP Design Constraints

Figure 9 shows the result.

Vertical Fin

The fin is a slightly different version of the tips. The only addition here is a square spar at the base of the fin. As we will see, we will mount the fin at an offset, and not glue it directly to the tail boom.

Figure 10 shows the fin.

Motor Stick

With the flying surfaces set up, we can now turn to the fuselage parts. The most important of these is the motor stick, which supports everything else in the model. The motor stick must also bear the forces imposed by the wound up rubber motor that will power this craft.

We can set up the motor stick several ways, but I will use a *polygon* to define the basic shape, then use *linear_extrude* to generate the actual object.

Here is the basic layout we will use:

This design provides support for the front bearing and the rear hook. The module only needs one pa-

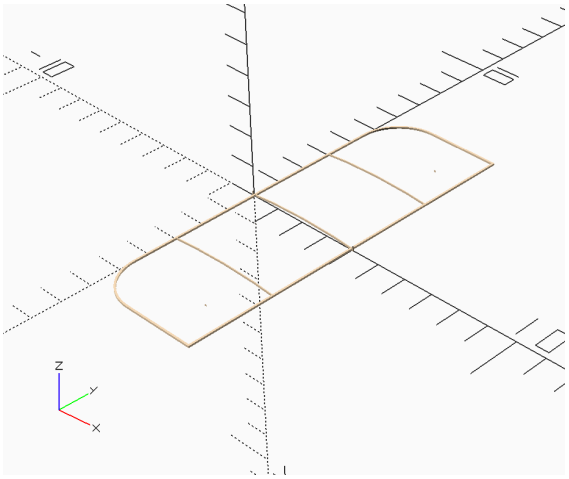


Figure 9: Stabilizer

rameter: the thickness of the stock you will be using for this part.

```
module motor_stick(thickness=1/8) {
  ...
}
```

The final shape is centered along the **X axis** with the bottom of the stick lying on that axis.

I added one new feature to this module. You can ask *OpenSCAD* to color shapes using the *color* command. For this motor stick, I just added this code to the module:

```
module motor_stick*thickness=1/8) {
  color(WOOD_Balsa)
    rotate[(90,0,0)])
      translate([0,0,thickness/2])
  ...
}
```

```
I also added a single line at the top of t

\begin{lstlisting}
include <colors.scad>
```

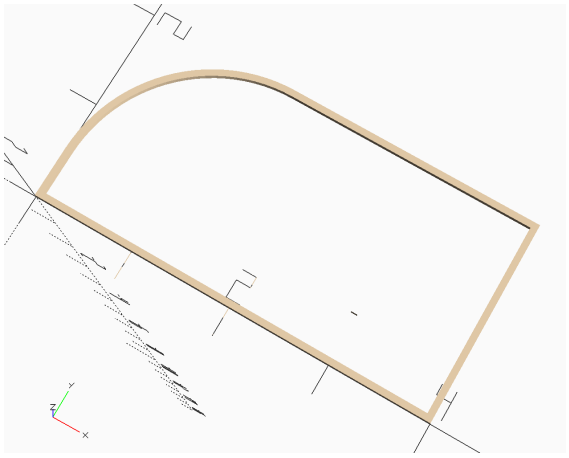


Figure 10: Vertical Fin

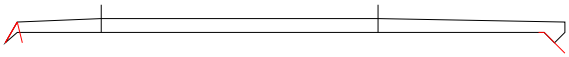


Figure 11: Basic Motor Stick

The **colors.scad** file is one I found online. I added my own color for balsa!

1 Tail Boom

After building the motor stick, the tail boom is something of a letdown. All we need here is a basic stick, but tapered from the front to the rear.

```
module tail_boom(
        thickness = 1/16,
        front_height = 3/16,
```

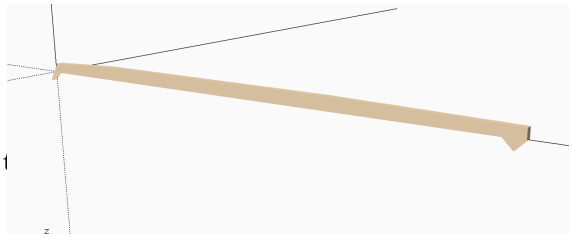


Figure 12: motor-stick.png

```
        rear_height = 1/16
        ) {
  ...
}
```

We will see this part when we assemble the airplane

2 Propeller

Now for an interesting component - the propeller. This is not just a flat part, it has an interesting shape, and figuring out how to model this thing took some time. In the end, I remembered how most of us build indoor propellers. We take a thin sheet of balsa, cut out the blade profile we want, then soak that blade for a while. Next, we tape it to the side of a round can at an angle and bake it. When it dries, it has a curvature that will work fairly well.

It struck me that I could generate the same shape by creating a polygon that represents the blade outline, then extrude that to form a very thick blade (bigger that the can!) I then build a hollow cylinder with a thickness that matches our desired blade thickness and slide the extruded blade into that cylinder. The *intersection* of these two shapes will leave us with a curved blade. Neat!

The code for this got fairly involved, so I will refer you to the project website for details.

```
module prop_blade(
        span =
  ) {
...
}
```

2.1 Blade Planform

The shape of the blade seems to be a matter of taste. Many builders design blades that will provide for a flair so that the prop will have a higher pitch when the plane is launched with full torque from the motor. Moving the prop spar toward the trailing edge of the blade provides this flair.

I decided to generate a simple blade layout, with parameters that can be adjusted to give the flair you want. Figure 13 shows the general layout.

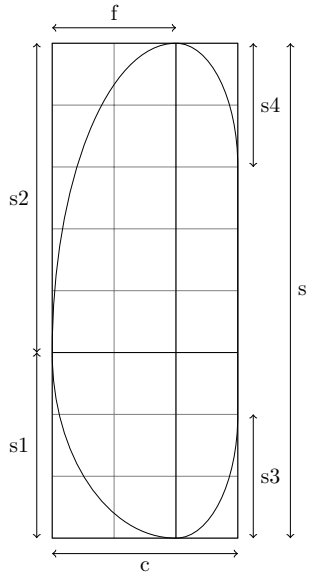


Figure 13: Prop Blade Layout

2.2 Prop Spar

The prop spar is a tapered cylinder, something easy to generate with the *cylinder* shape. Parameters are provided so the size of the spar can be adjusted.

Here is what I ended up producing, after rotating it back upright.

Final Assembly

Now that we have all of the major components defined, it is time to put things together and see the completed model.

Mounting Components

The wing will be mounted on top of two hard balsa posts using paper tubes. These wing posts are glued to the motor stock on the bottom and rounded at the

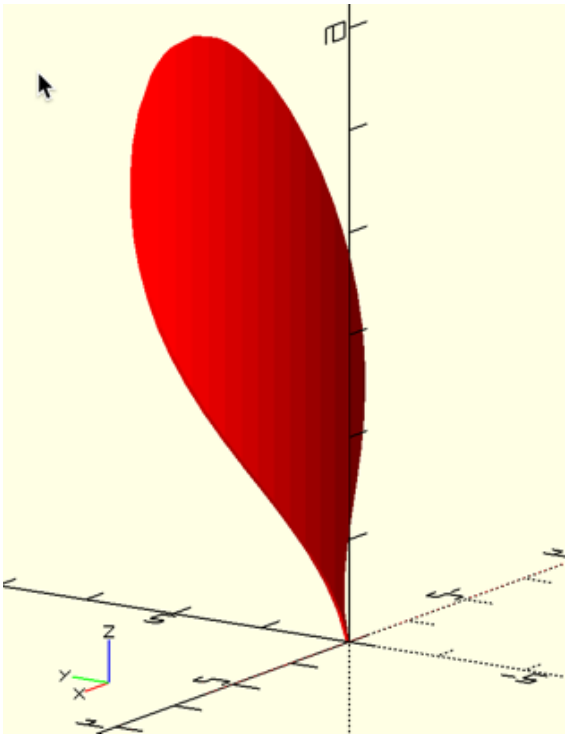


Figure 14: Prop Blade

top to slide into paper tubes that will be glued to the wing structure.

A similar arrangement is used to attach the stabilizer on top of the tail boom.

The rudder is glued to the bottom on the tail boom, but is offset to provide for a turn during flight. A small stick of balsa will be glued to the rear of the tail boom and the fin to provide needed support. Should this need adjusting, the rear attachment can be unglued and repositioned.

Paper Tubes

Paper tubes are formed on a mandrel using a strip of tissue soaked in thin glue and twisted around that mandrel. The resulting tube, after pulling off of the mandrel, will be stiff enough to provide the support needed. The posts must be carefully sanded to ensure a tight fit.

The code that creates a paper tube, it is just a very skinny hollow cylinder.

Figure **??** shows an image showing how the tubes are attached to both the wing and stabilizer:

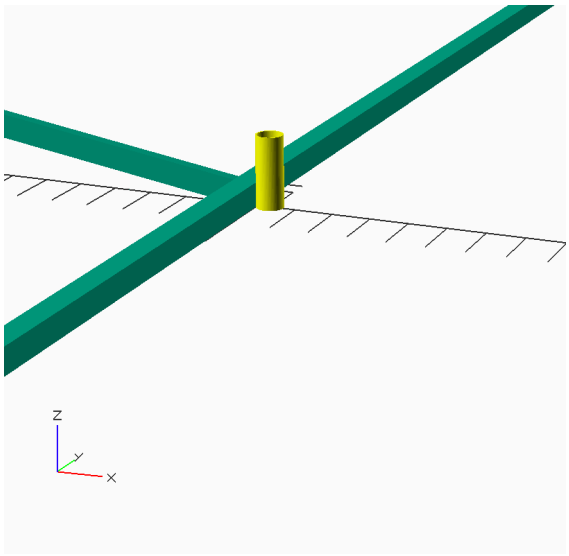


Figure 15: Wing Mount Tube

Mounting Posts

The posts used to attach the wing and stabilizer are simple sticks of hard balsa with rounded tops that are sized for a tight fit in the paper tubes.

Since we need several mounting posts, the code that generates the post is placed in another module:

To attach the wing, we need to attach two posts to the motor stick:

Mounting Tail Group

Both the stabilizer and fin mount on the tail boom.

Stabilizer

The stabilizer is mounted on top of the tail boom, using small posts and paper tubes to allow for minor

adjustments when flying.

To get started, we set up a few dimensions in the data file

2.2.1 Vertical Fin

The rudder is simple attached to the bottom of the tail boom, Since that side of the boom is canted by the trimming we performed earlier, we need to rotate the rudder slightly during positioning. however, we allow for left rudder attachment to provide a left turn flight path.

Flight Box

Since indoor model airplanes are so extremely light and fragile, it is important that you give thought to how you are gong to transport the model. Most builders do not show up at a flying site with just one model. Often, they bring several for each event they want to fly. Many serious flyers build a nice carrying case for their models. I decided to design one for Math-Magik!

When I was first getting into indoor competitions back in my college days, I met Bud Tenny, who was the editor of a now-classic series of newsletters called *Indoor News and Views*. These newsletters provided a wealth of information on building indoor airplanes, and provided tips on how to fly them. Often there were sketches of simple boxes made out of cardboard, or foam board that looked like possibilities. However, I also have a copy of Ron Williams *Building and Flying Indoor Model Airplanes* [6] , and ran into the ECIM case, originally designed by the *East Coast Indoor Modelers* group. The design presented here is a version of that case:

2.3 Basic Construction

As seen in figure 16, the box is made up of three sections. One holds a number of wings (four are shown here). One holds the stabilizers, and the middle section holds fuselages and propellers. The two end sections are hinged to the center section so it opens as

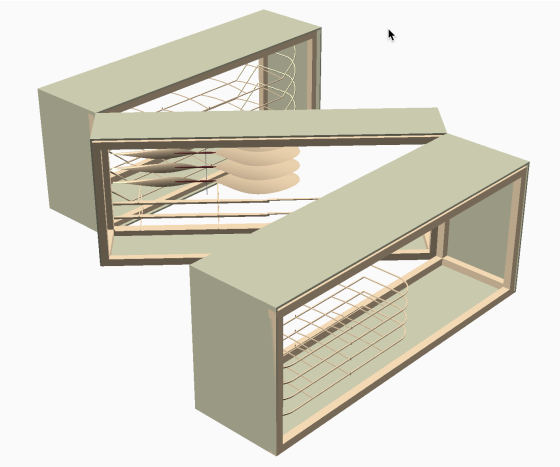


Figure 16: Flight Box

seen above. Closed, this design measures 22"w x 9"h x 16"d. The outer sections are 6 inches deep and the center section ins 4 inches deep.

The sides are constructed out of 1/8" plywood and all corners are braced with 1/2" pine strips. The hinges are piano hinges that run the full height of the edges.

2.4 OpenSCAD Design

Designing this box with OpenSCAD turns out to be a nice way to figure out how parts of the airplane will be transported and stored. The design of the airplane structure presented so far allows both the wing and stabilizer to be removed and stored separately. By attaching the wing posts to the motor stick and using paper tubes for the attachments to flying surfaces on top of those posts, the wings are not cluttered with structure and can be stacked neatly. The stabilizers also stack nicely.

Laying the body on its side lets those be stacked as well.

That leaves the propellers! They can be stacked separately in the space above the fuselage sections,.

When all of these items are placed in the box design, there is still room for another model. My current

thinking is to adds an A-6 model or two to this box's payload!

I hope you see how handy a tool like *OpenSCAD* can be when you want to create things to support your hobby!

Generating STL Files

Many folks who use OpenSCAD generate *STL* files from their design. These files are used as part of the 3D printing process. Although I wish we could 3D print flyable model airplanes, that is not what we will do with the STL files in this project.

STL stands for *Standard Tessellation Library* [3]. Translated, that means an STL file is a list of triangular objects that describe the surface of a 3D object. These triangles cover the surface making a "water tight" approximation to the real surface. Triangles are guaranteed to be flat in the 3D space, and if they are tiny enough, they can be rendered to create a 3D display of the object, or sent to a 3D printer after suitable processing of all the triangles so the printer knows where the surface is $thatisatopicforanotherarticle$!

My purpose in introducing these triangles is simple. I want to know how much my model will weigh, and I want to know where the center of gravity will be when the design is constructed. Figuring out these two details is impossible using conventional building techniques: you build the model, weight it, and figure out the center of gravity manually. Computer geeks never do anything manually if they can get their computer to do the work.

I found a nice Python library that is all set up to figure out the volume of a 3D shape defined in an STL file, and return it's center of gravity location. It is simple enough to add in the predicted wood density for the design and come up with a weight estimate for each part. A little post-processing of all this data will give us some estimate of the total designs weight. Unfortunately, figuring out the weight of the glue is not so simple. For that we need to know where each glue joint will be and the surface area of the glue joint. I am working on that problem now, but am not ready to show any results yet.

Knowing where each part will be placed in the final design, and the CG data for that part will let us predict the CG for the complete model as well!

Let's demonstrate this idea using a simple rib section.

2.5 Estimating Weights

I have worked to set up each major part you would need to create out of balsa, or other materials as a module we can activate separately. In this example, I will show how to estimate the weight of a single wing rib.

For this section, I will be using OpenSCAD as a *command-line* tool, something that may not be familiar to non-programmers. Details on this are on the project website, but basically, we will open up a window where we can type in commands to the operating system. This will involve running a **cmd.exe** on a Windows system, and opening up a *terminal* on Mac and Linux systems.

We will run OpenSCAD as a tool and will not open up the graphical user interface you normally see. In this mode, OpenSCAD will read a file, process it and generate an output file containing the results we want. In this example, we will be asking it to generate an STL file from a part definition *scad* file.

My **rib.scad** file is set up to display a single rib if you load it in OpenSCAD normally. The command I give to my Operating System to generate the *STL* file looks like this:

```
openscad rib.scad -o rib.stl
```

That is not so bad. If you check the directory where we ran this command (the one containing **rib.scad**, you will see a new file named **rib.stl**. You can examine this file, but all it contains is a bunch of definitions of the triangles needed to enclose a single rib.

Using the Python library to get the rib data is pretty easy. Here is the code I used:

Listing 7: *stl/getvolume.py*

```
import numpy
import math
import os
import stl
import sys
from stl import mesh

def find_mins_maxs(obj):
    minx = maxx = miny = maxy = minz = maxz = None
    for p in obj.points:
        # p contains (x, y, z)
        if minx is None:
            minx = p[stl.Dimension.X]
            maxx = p[stl.Dimension.X]
            miny = p[stl.Dimension.Y]
            maxy = p[stl.Dimension.Y]
            minz = p[stl.Dimension.Z]
            maxz = p[stl.Dimension.Z]
        else:
            maxx = max(p[stl.Dimension.X], maxx)
            minx = min(p[stl.Dimension.X], minx)
            maxy = max(p[stl.Dimension.Y], maxy)
            miny = min(p[stl.Dimension.Y], miny)
            maxz = max(p[stl.Dimension.Z], maxz)
            minz = min(p[stl.Dimension.Z], minz)
    print("Shape bounds are:")
    print("  X: {0} <-> {1}".format(minx, maxx))
    print("  Y: {0} <-> {1}".format(miny, maxy))
    print("  Z: {0} <-> {1}".format(minz, maxz))

def get_volume(obj):
    volume, cog, inertia = \
        obj.get_mass_properties()
    return volume, cog

    print("Shape volume: {0}".format(volume))
    print("Shape CG
: {0}".format(cog))

def main():
    try:
        fname = sys.argv[1]
        if fname.endswith('.scad'):
            fname = fname.split('.')[0]
        density = sys.argv[2]
        print("Processing {0} with density {1}".for
    except:
        print("No file to process")
        sys.exit(1)

    cmd = "openscad {0}.scad -o {0}.stl".format(fna
    os.system(cmd)

    fmesh = mesh.Mesh.from_file(fname+'.stl')
    find_mins_maxs(fmesh)
    volume, cog = get_volume(fmesh)
    print("Shape volume: {0} cubic inches".format(v
    print("Shape CG
: {0}".format(cog))
    weight = volume * float(density) * 453.592/173
    print("Estimated weight = {0} grams".format(we

main()
```

This program is run from the command line as follows:

```
python genvolume.py rib 4.5
```

The parameters are the name of the shape file, and the density in pounds per cubit foot.

This will process the rib.scad file in the current directory, and print out the volume and CG information. It also prints the bounds of the box occupied by the shape, and the expected weight for the part.

Here is the output I saw from my test:

```
\$ python getvolume.py rib 4.5
Processing rib with density 4.5
Shape bounds are:
  X: 0.0 <-> 5.0
  Y: -0.015625 <-> 0.015625
  Z: -2.543129937748745e-07 <-> 0.3625000119209289
Shape volume: 0.009852695666874448 cubic inches
Shape CG    : [2.50000603e+00 6.15394834e-11 2.2725
Estimated weight = 0.011571327789516752 grams
```

That is pretty cool, and the numbers look reasonable for this rib!

At present, I am working on setting up a system to calculate the total weight and center of gravity data for this design. The input will be a data file listing the parts needed to build th design, where those parts will be located. The output will be the total weight and center of gravity estimates, as well as the total bounds of the airplane, - so you can check that you satisfy the rule constraints.

Biography

In the Summer of 1955 I was delivering the evening newspaper in Falls Church, Virginia, when I rounded the corner of an apartment building and saw a man release the propeller on a rubber-powered model airplane. The plane circled in front of this man's home for several minutes, and magically landed where it had started. The airplane was a Henderson Gadfly, published in Model Airplane News that year. I was fascinated by that sight, and decided to figure out how the airplane managed to do that. I talked the man into giving me the plans he used to build the model, traced from the magazine. I still has those plans to this day!) Soon, a couple of my friends and I decided to start building model airplanes of our own. We all took a bus to downtown Washington, D.C (kids could do that back then), and joined the Academy of Model Aeronautics. We also joined the Fairfax Model Associates and began competing in a variety of events, mostly control line and gas free flight. At one meeting, Bill Bigge, an internationally known indoor model builder, was the guest speaker. I got my first look at a new form of model airplane. The indoor models Bill brought to the meeting were fascinating, and cheap enough even a kid with a limited allowance could build one. Bill became my mentor, and I managed to build an ornithopter and helicopter and set two national records! I spent 20 years as an officer in the USAF, then another 17 years teaching college-level Computer Science. I finally retired for good in 2018, and moved with my wife to Kansas City, where I joined the Heart of America Free Flight Association, and again began flying model airplanes, this time focusing on rubber and electric powered outdoor free flight, and indoor events.When not building model airplanes, I am active in Amateur Radio and am currently authoring a book on Computer Architecture.

References

[1] Constructive solid geometry - Wikipedia, 2021. URL `https://en.wikipedia.org/wiki/Constructive{_}solid{_}geometry`.

[2] OpenSCAD CheatSheet, 2021. URL `https://www.openscad.org/cheatsheet/`.

[3] STL (file format) - Wikipedia, 2021. URL `https://en.wikipedia.org/wiki/STL{_}(file{_}format)`.

[4] R. Black. Math Magik LPP, 2021. URL `https://rblack42.github.io/math-magik-lpp/`.

[5] M. Kintel. OpenSCAD - The Programmers Solid

3D CAD Modeller, 2021. URL `https://www.openscad.org/`.

[6] R. Williams. *Building & Flying Indoor Model Airplanes*. Ron Williams, 2008. ISBN 978-0-615-20203-7.