WIKIBOOKS
Open books for an open world

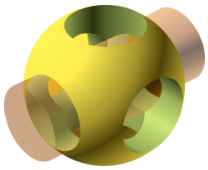
OpenSCAD Tutorial/Printable version

OpenSCAD Tutorial

The current, editable version of this book is available in Wikibooks, the open-content textbooks collection, at
https://en.wikibooks.org/wiki/OpenSCAD_Tutorial

Introduction

About OpenSCAD



OpenSCAD is a solid 3D CAD modelling software that enables the creation of CAD models through a scripting file. The domain specific language designed for this purpose allows the creation of fully parametric models by combining and transforming available primitives as well as custom objects.

About this tutorial

This tutorial assumes zero programming or CAD knowledge and is designed to guide you step by step through examples and exercises that will quickly build your understanding and provide you with the right tools to create your own models. Emphasis is placed on parametric design principles that will allow you to rapidly modify your creations and build your own library of reusable and combinable models.

The majority of presented examples and solutions to exercises are available as separate OpenSCAD scripts here (https://github.com/openscad/documentation/tree/master/OpenSCAD_Tutorial/Tutorial_Files).

As of 29-11-2019 this tutorial as well as all accompanying material were completely developed as a Google Season of Docs project.

Chapter 1

A few words about OpenSCAD

OpenSCAD is for crafting 3D models through the art of Constructive Solid Geometry. It unlocks a world of creativity, where elementary operations are like our building blocks.

Let's shape up some fun.

Getting started with the Tutorial

This tutorial will be your trusted guide. We'll be exploring examples and unveil the secrets of OpenSCAD. By the end of this tutorial, you will have the tools to forge your own unique 3D models, line by line.

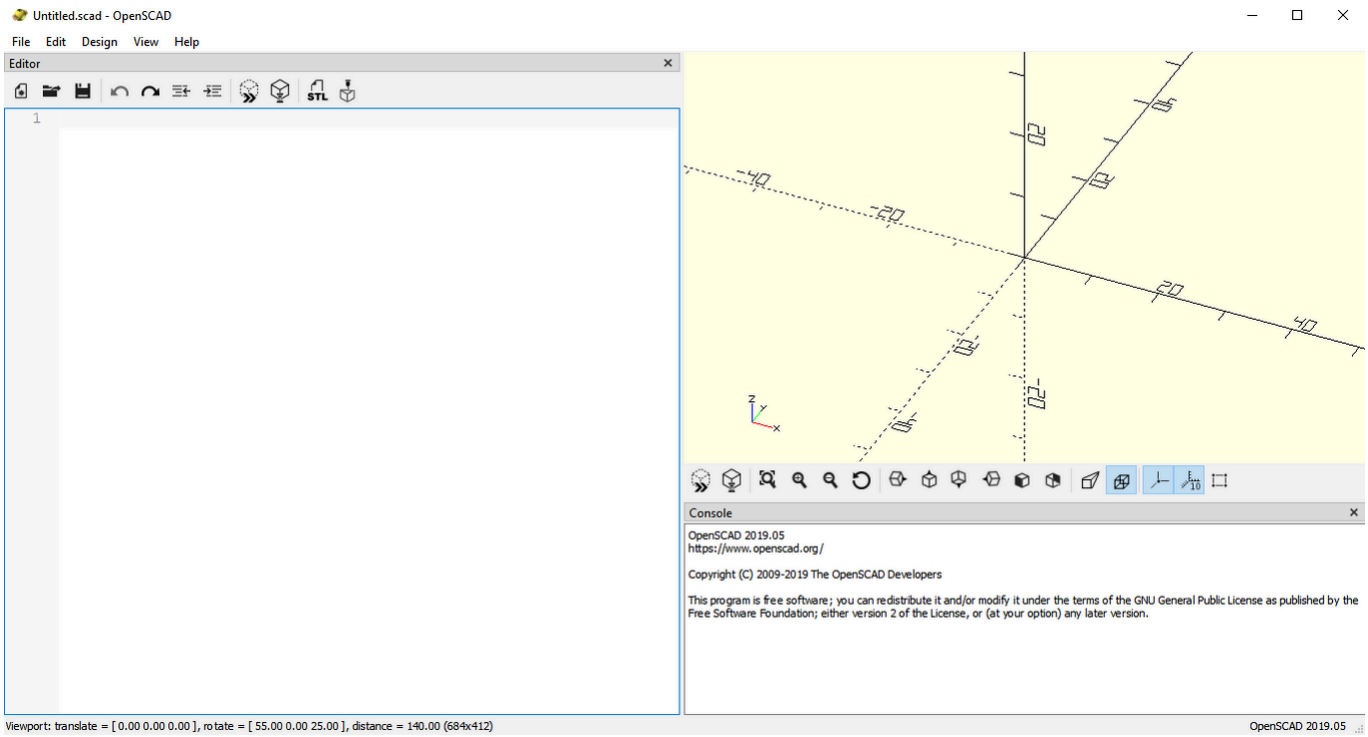
With each step, you'll gain confidence and expertise, honing your skills as an image creator. You will breathe code into your designs, crafting intricate structures and bringing your design ideas to fruition.

Throughout this tutorial, we'll be your companion, offering guidance to unlock the full potential of OpenSCAD.

You'll explore, learn, and create.

User Interface

After starting OpenSCAD the window should look similar to the image below.



The Window is divided in three columns.

1. In the left column, is seen the builtin text editor, where the true magic unfolds. As you enter keyboard commands you can view the transformation of code into art.
2. In the middle column, the displayed 3D View is where your design creations come to life. At the bottom lies the operations sequence console, always ready to lend a helping hand. It unravels the mysteries of mistakes and guides you towards mastery. It is your trusted guidance companion.
3. And note the right column, the GUI Customizer. It offers the user a gift of ease toolbar, a graphical interface to tweak and twist your model's parameters.

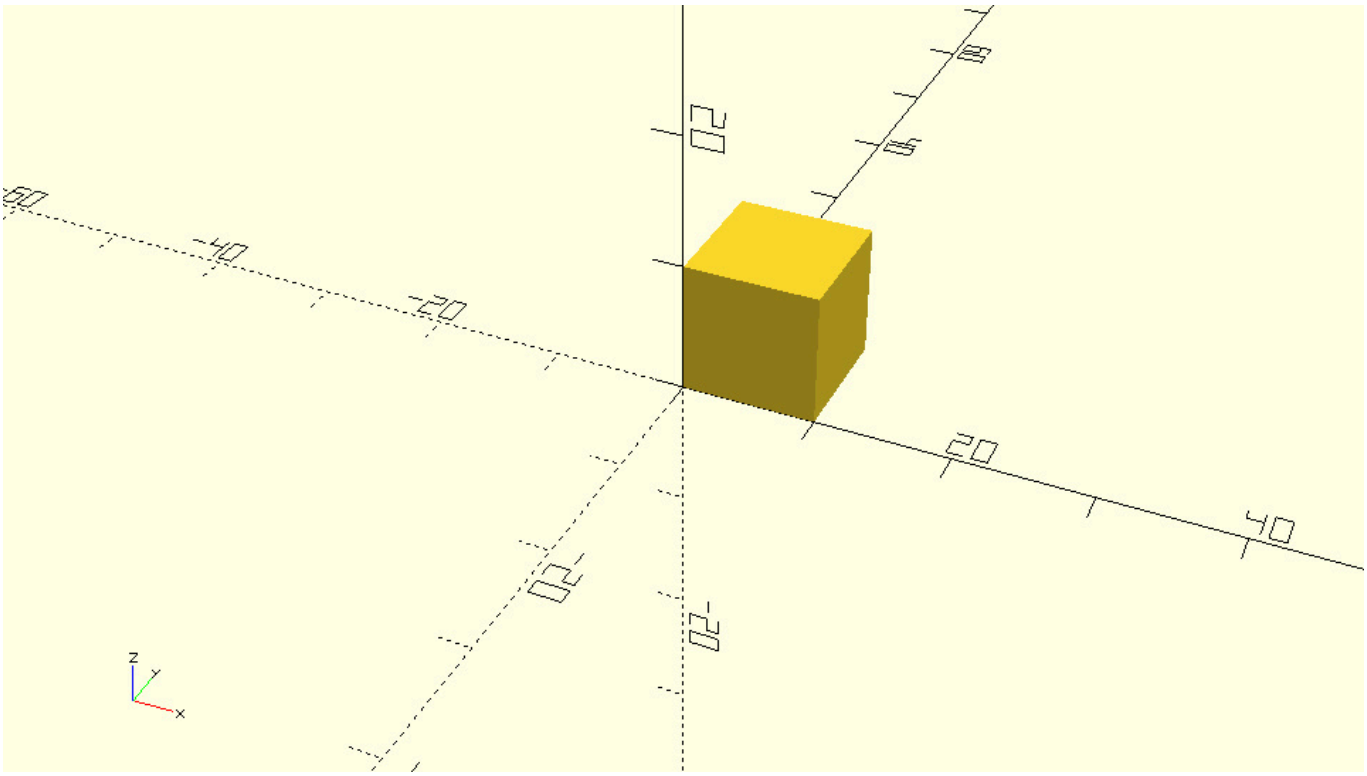
Creating your first object

Your first object is going to be a perfect cube with side length of 10. In order to create it you need to type the following code in the text editor and hit the preview (first) icon on the action bar below the reference axes.

Code

a_small_cube.scad

```
cube(10);
```



There are a few fundamental concepts that you should learn from the start regarding the OpenSCAD scripting language, especially if you don't have a programming background. The word 'cube' is part of OpenSCAD scripting language and is used to command OpenSCAD to create a cube. The 'cube' command is followed by a pair of parentheses, inside of which the parameter size is defined to be 10. Any definition of parameters that a command may require is always done inside a pair of matching parentheses that follow the command word. The semicolon after the last

parenthesis indicates the end of that statement and helps OpenSCAD to parse the script that you have typed in the text editor. Because a semicolon is used to indicate the end of each statement you have the freedom to format your code in any way you like by inserting whitespace.

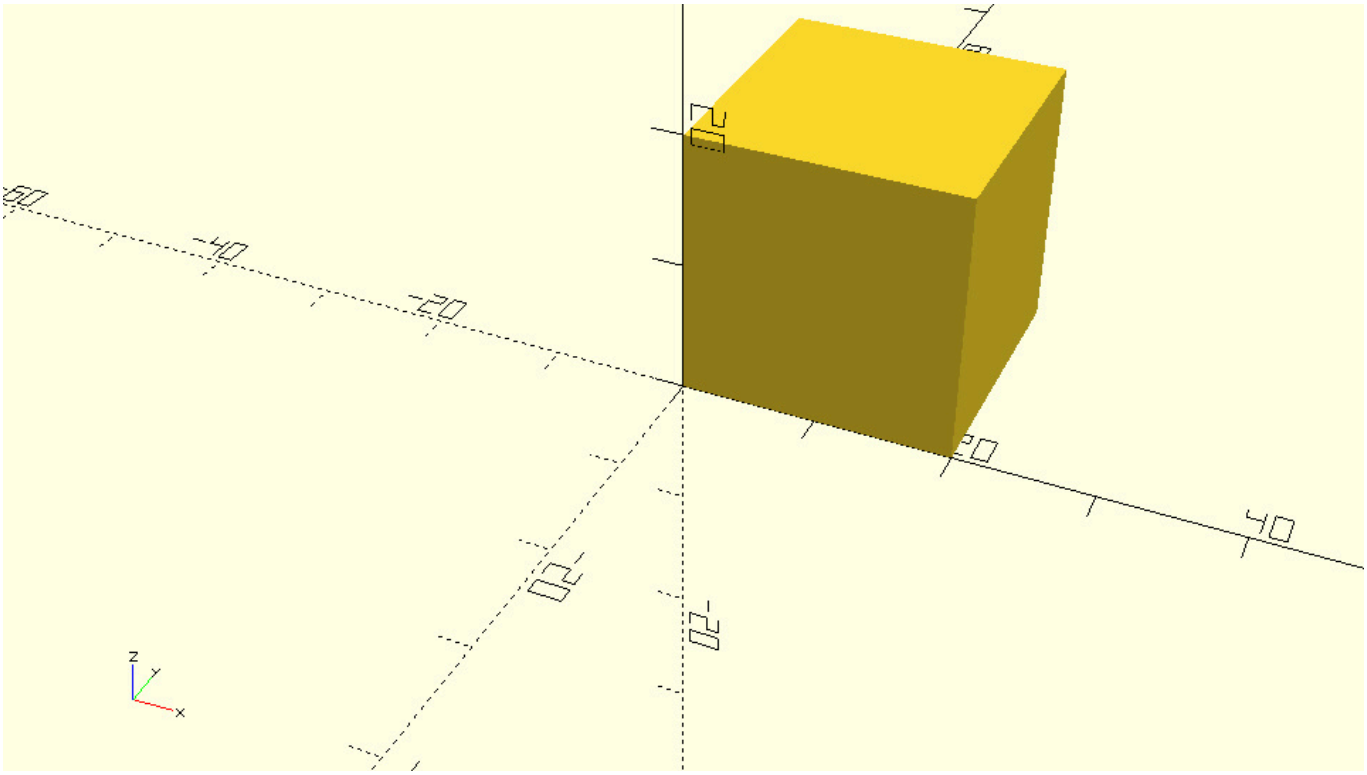
Exercise

Try adding some whitespace between the word 'cube' and the first parenthesis and then hit (select) the "preview" option. Is your cube created? Do you get any error message? Try adding some additional whitespace in different places and hit "preview" again to see what you can get away with before getting an error message in the console. What happens if you add whitespace between the syllables 'cu' and 'be' of the word 'cube' and hit "preview"? What happens if you delete the semicolon?

You just read "hit preview" three times in the last paragraph. When you hit "preview" OpenSCAD parses your script and creates the appropriate model. Every time you make a change to your script (ex. adding whitespace) or later when adding additional statements, you need to hit "preview" to see the effect of these changes.

Exercise

Try changing the size of the cube to 20 and see what happens. Did you remember to hit preview in order to see your changes take place?



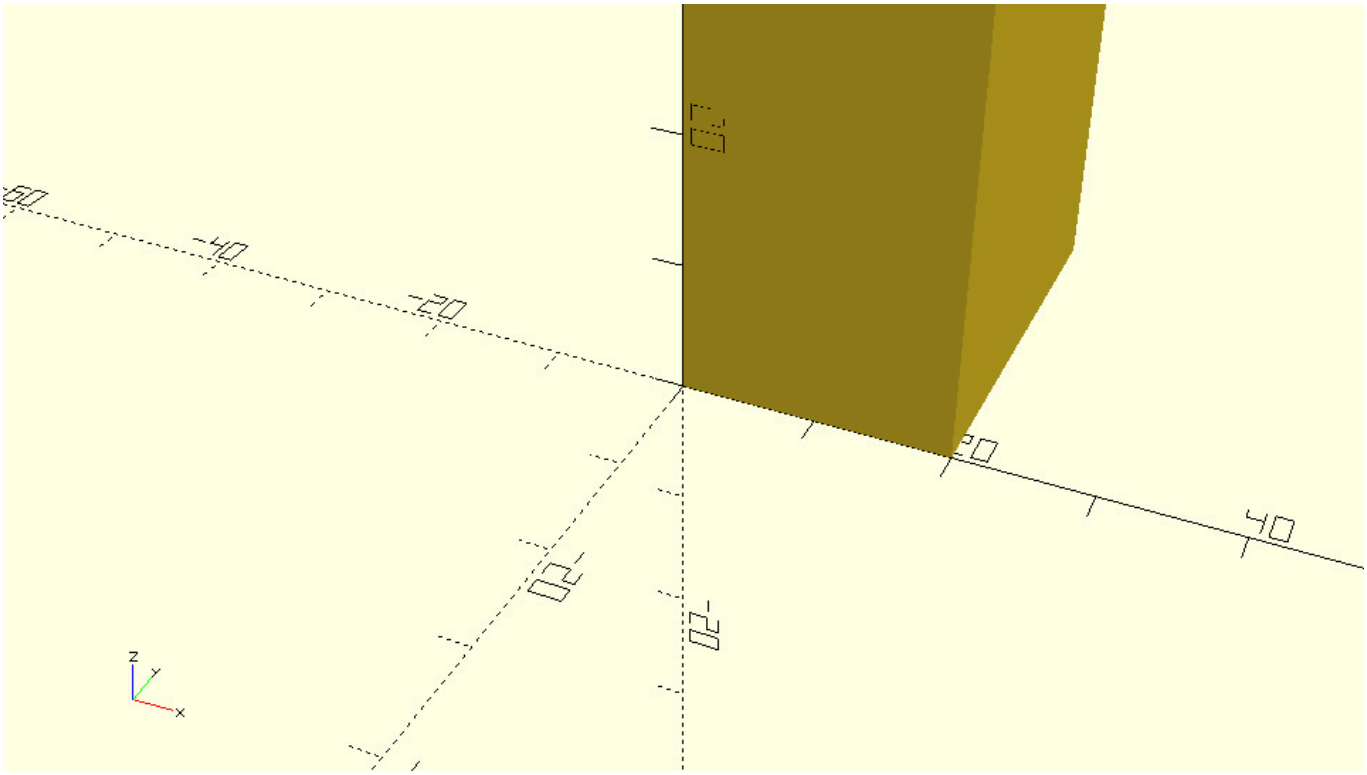
Creating a slightly different cube

A cube doesn't have to be perfect (equal distance). A cube can (be cubical) have different side lengths. Use the following statement to create a cube (cubical) with side lengths of 25, 35 and 55.

Code

a_different_cube.scad

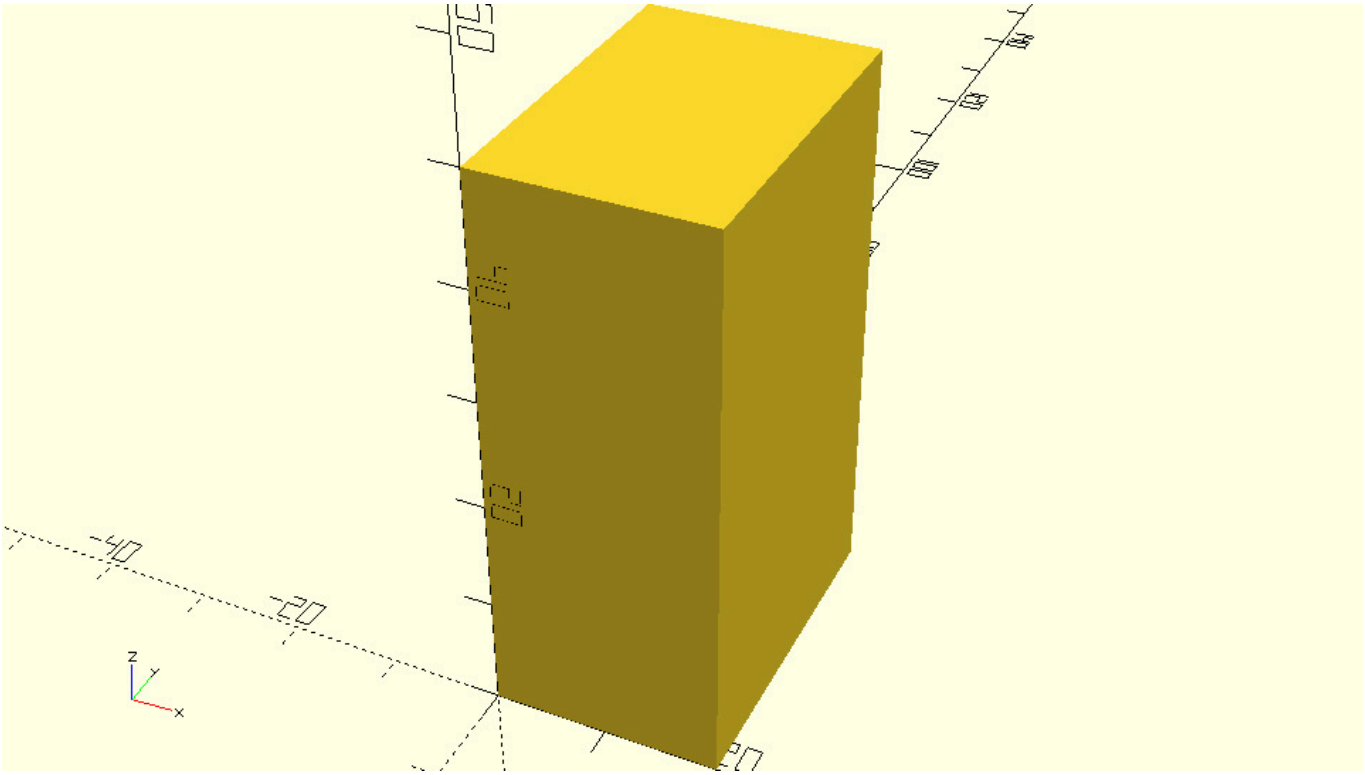
```
cube([25,35,55]);
```



The first thing that you should notice is that this cube is quite large compared to the previous one. In fact, it is large enough that it doesn't fit in the viewport. In order to fix this, you can move your mouse over the viewport and scroll out until you can see the whole cube. You can always zoom in and out by moving your mouse over the viewport and using the scroll wheel. Alternatively, you can use the zoom in (fourth) and out (fifth) icons on the action bar below the viewport. You can let OpenSCAD automatically choose a convenient zoom level by using the view all (third) icon in the same action bar.

Exercise

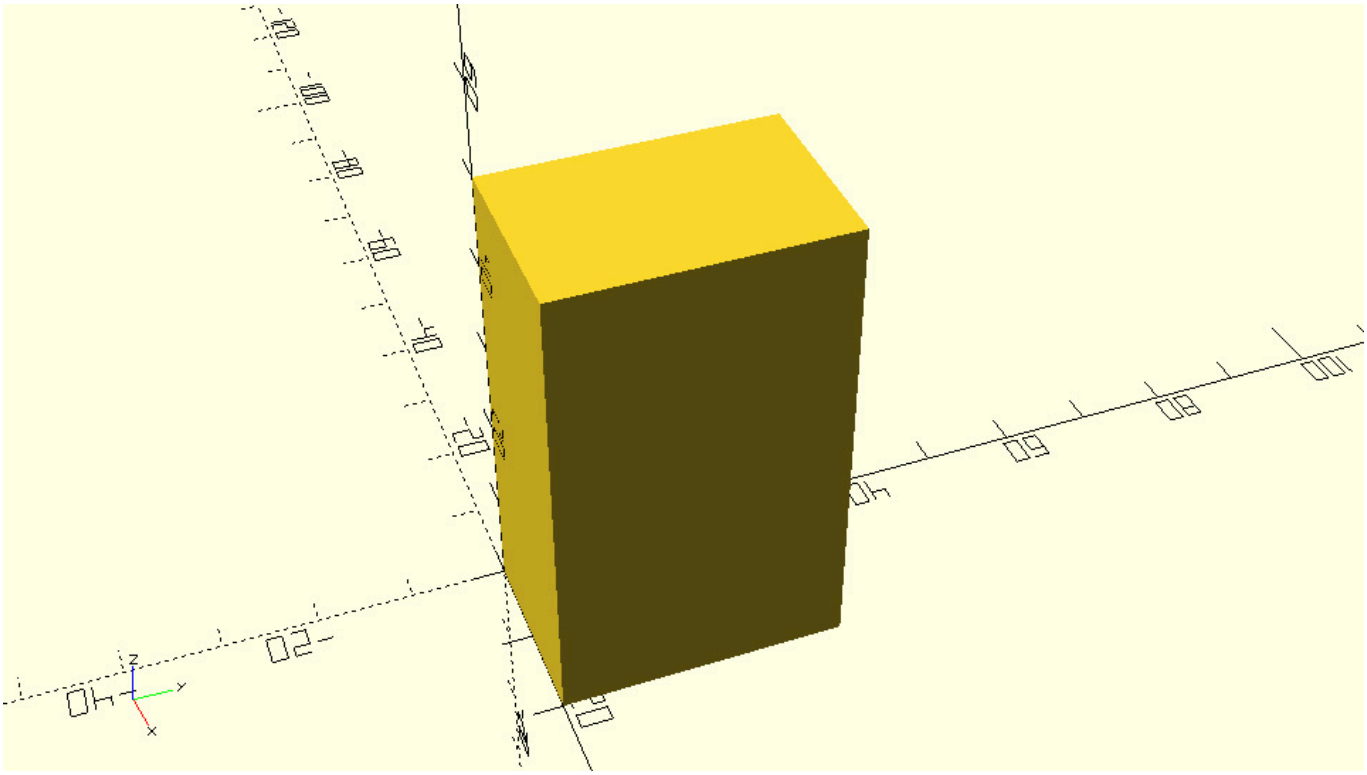
Try moving your mouse over the viewport and using the scroll wheel to zoom in and out. Try zooming in and out using the corresponding icons. Let OpenSCAD choose a zoom level for you.



Apart from zooming in and out you can also move and rotate the view of your model. To do so you need to move your mouse over the viewport and drag while holding right click to move or drag while holding left click to rotate. You can reset the view by using the reset view (sixth) icon on the action bar below the viewport.

Exercise

Try dragging your mouse over the viewport while holding right or left click to move or rotate the view of your model. See how long you can mess around before you need to reset the view.



The second thing that you should notice is that in order to create a cube with different side lengths you need to define a pair of brackets with three values inside the parentheses. The pair of brackets is used to denote a vector of values. The values of the vector need to be comma-separated and

correspond to the cube side lengths along X, Y and Z axis. When the cube command is used with a vector of three values as its input, OpenSCAD creates a cube with different side lengths that correspond to the values of the vector. Remember that you previously used the cube command to create a perfect cube by defining the value of the parameter size. Most OpenSCAD commands can be used with different parameters, even with more, less or no parameters to achieve different results.

Exercise

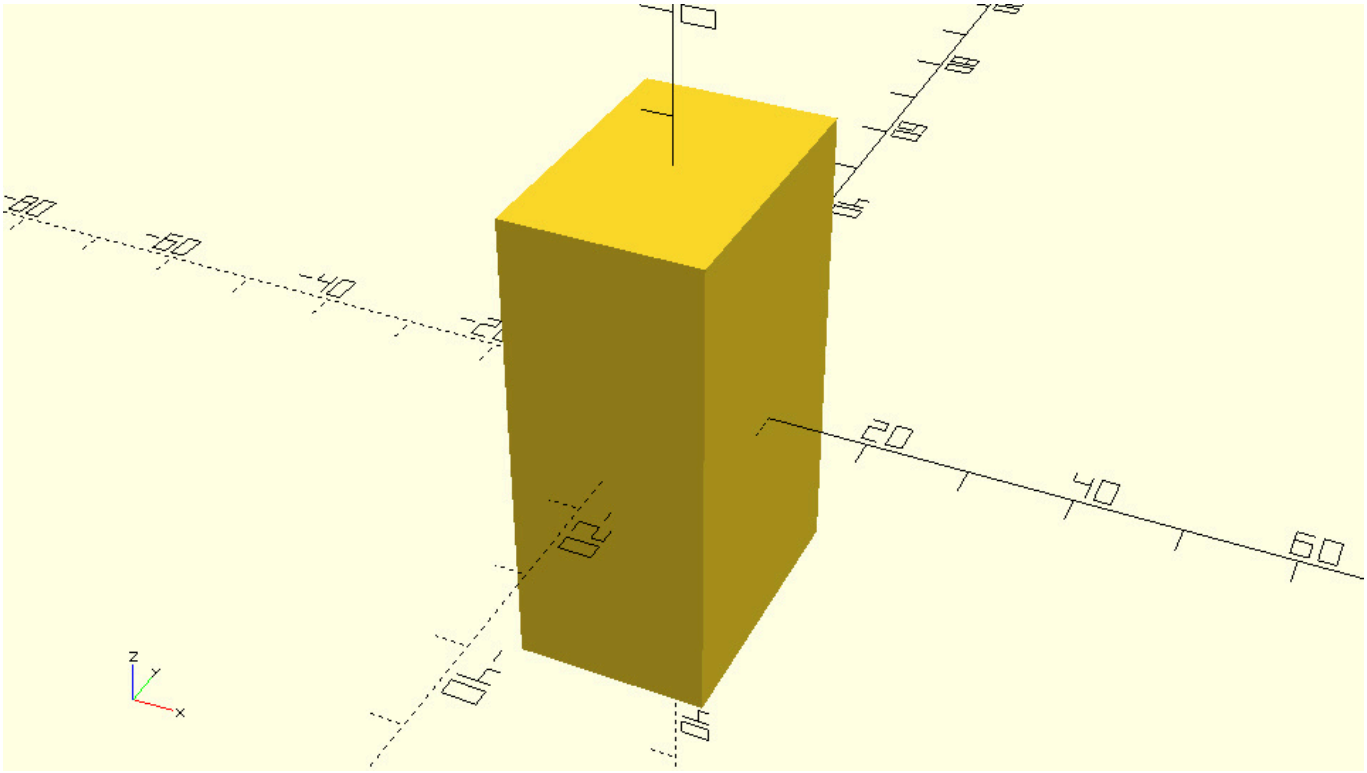
Try using the cube command with no parameters. What happens? Use the cube command to create a cube with side lengths of 50, 5 and 10. Use the cube command to create a perfect cube with side length of 17.25.

You should notice that every cube is created on the first octant. You can define an additional parameter named center and set it equal to true in order to make the cube centered on the origin. The complete statement is the following.

Code

a_centered_cube_with_different_side_lengths.scad

```
cube([20,30,50],center=true);
```



Notice that when more than one parameter is defined inside the parentheses, they need to be separated with a comma.

Exercise

Try creating a perfect cube or a cube with different side lengths. Use an appropriate additional input parameter to make this cube centered on the origin. If you like add some whitespace before and after the comma that separates the two parameters.

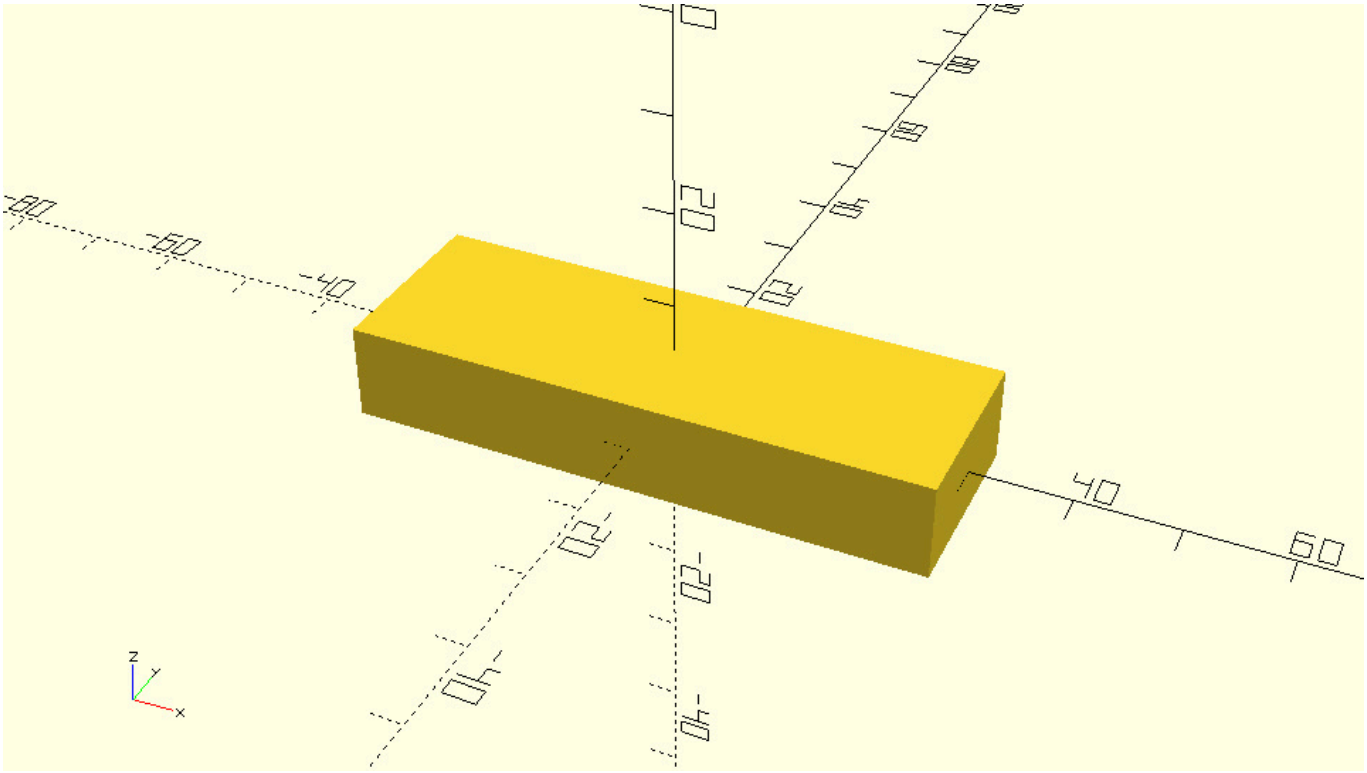
Adding more objects and translating objects

The constructive solid modelling approach uses a number of fundamental objects along with a number of ways to transform and combine these objects to create more complex models. The cube that you have been using in the previous examples is one such fundamental object. The fundamental objects are also called primitives and are directly available in OpenSCAD scripting language. A car for example is not an OpenSCAD primitive, as there is no corresponding keyword in the scripting language. This makes absolute sense because OpenSCAD is a set of modelling tools rather than a library of predefined models. Using the available tools, you can combine the available primitives to create your own car. To do this you need to know how to add more than one object to your model.

First create a cube with side lengths of 60, 20 and 10 that is centered on the origin.

Code

```
cube([60,20,10],center=true);
```

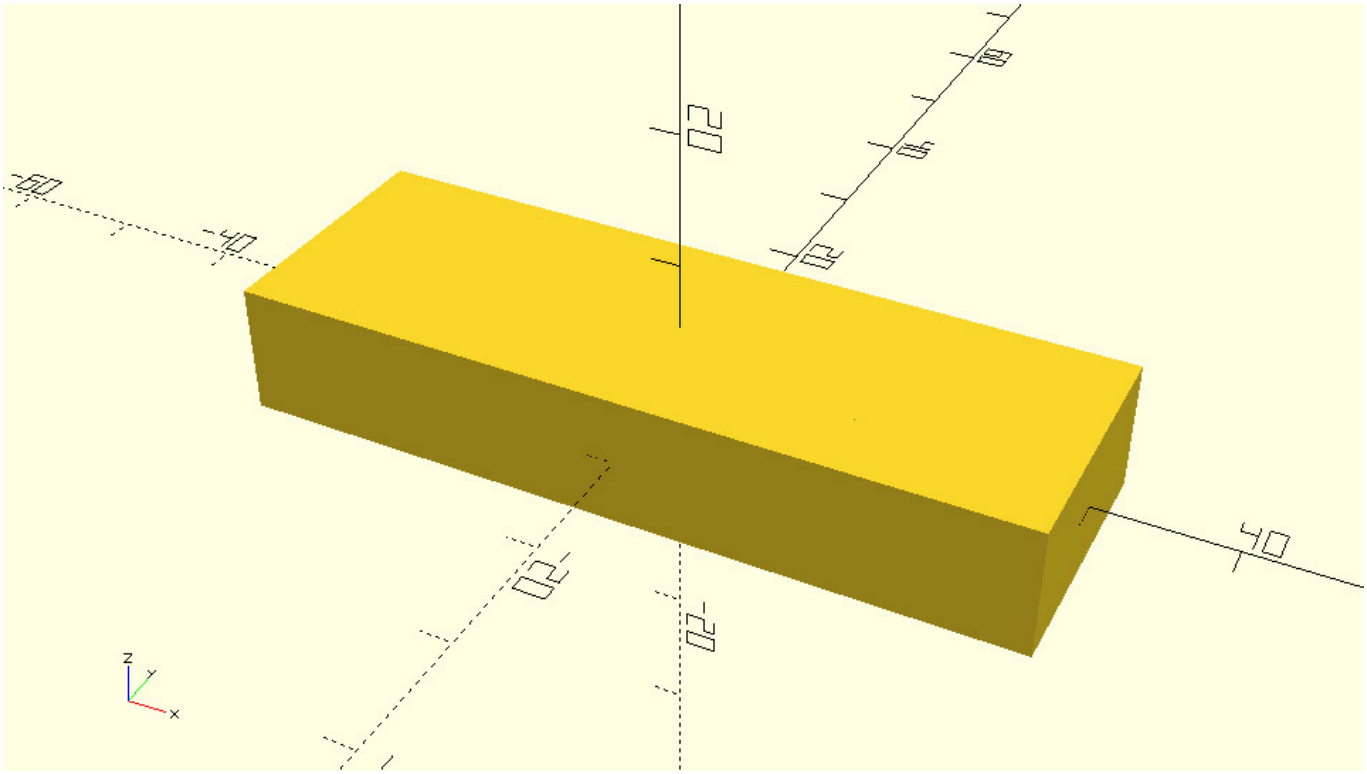


In order to add a second cube to your model type an identical statement in the next line of the text editor, but change the side lengths to 30, 20 and 10.

Code

a_smaller_cube_covered_by_a_bigger_cube.scad

```
cube([60,20,10],center=true);
cube([30,20,10],center=true);
```

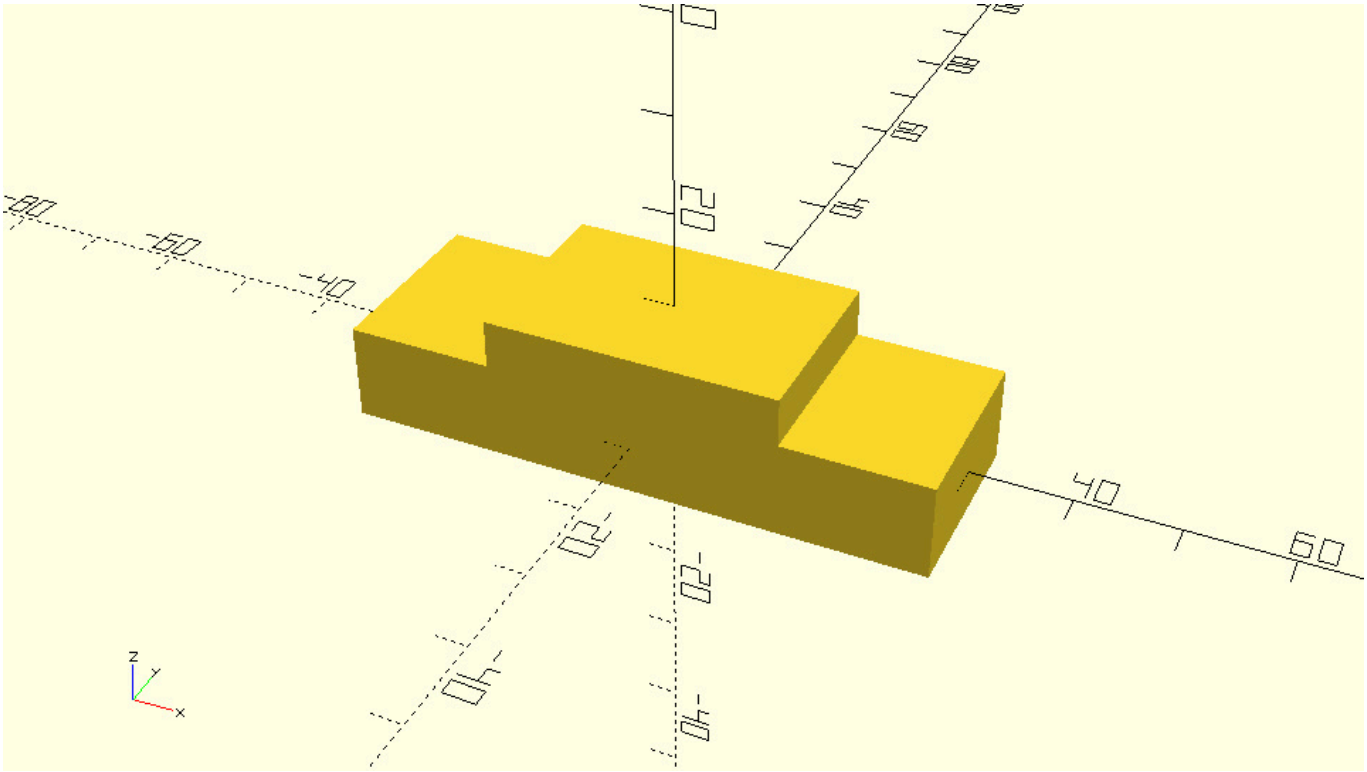


You should not be able to see any change in your model because the second cube is not larger than the first cube in any direction and is currently completely covered by the first cube. By modifying the second statement in the following way, you can translate the second cube to locate it partially above the top of the first cube.

Code

two_cubes.scad

```
cube([60,20,10],center=true);
translate([0,0,5])
    cube([30,20,10],center=true);
```



You achieved this by using the "translate" command which is one of the available transformations. The translate command as well as the rest of the transformations don't create any object on their own. They are rather applied on existing objects to modify them in a certain way. The translate command can be used to move an object to any point in space. The input parameter for the translate command is a vector of three values. Each value indicates the amount of units that the object will be moved along the X, Y and Z axis. You should notice that there is no semicolon after the translate command. What follows the translate command is the definition of the object that you want to translate. The semicolon is added at the end to indicate the completion of the statement.

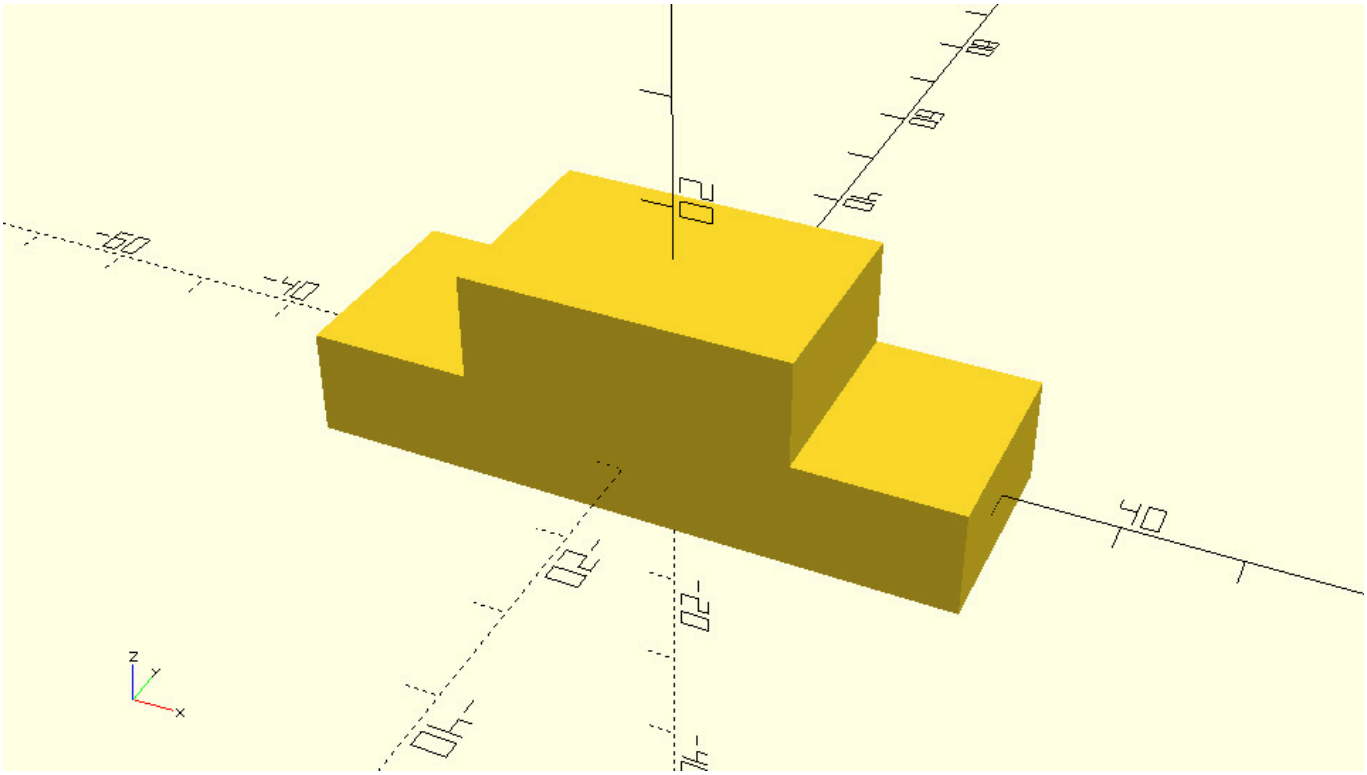
Exercise

Try changing the input parameter of the translate command so that the cube is translated 5 units along the X axis and 10 units along the Z axis. Try adding some whitespace if you would like to format this statement in a different way. Try adding a semicolon after the translate command.

Code

two_cubes_barely_touching.scad

```
cube([60,20,10],center=true);
translate([0,0,10])
    cube([30,20,10],center=true);
```

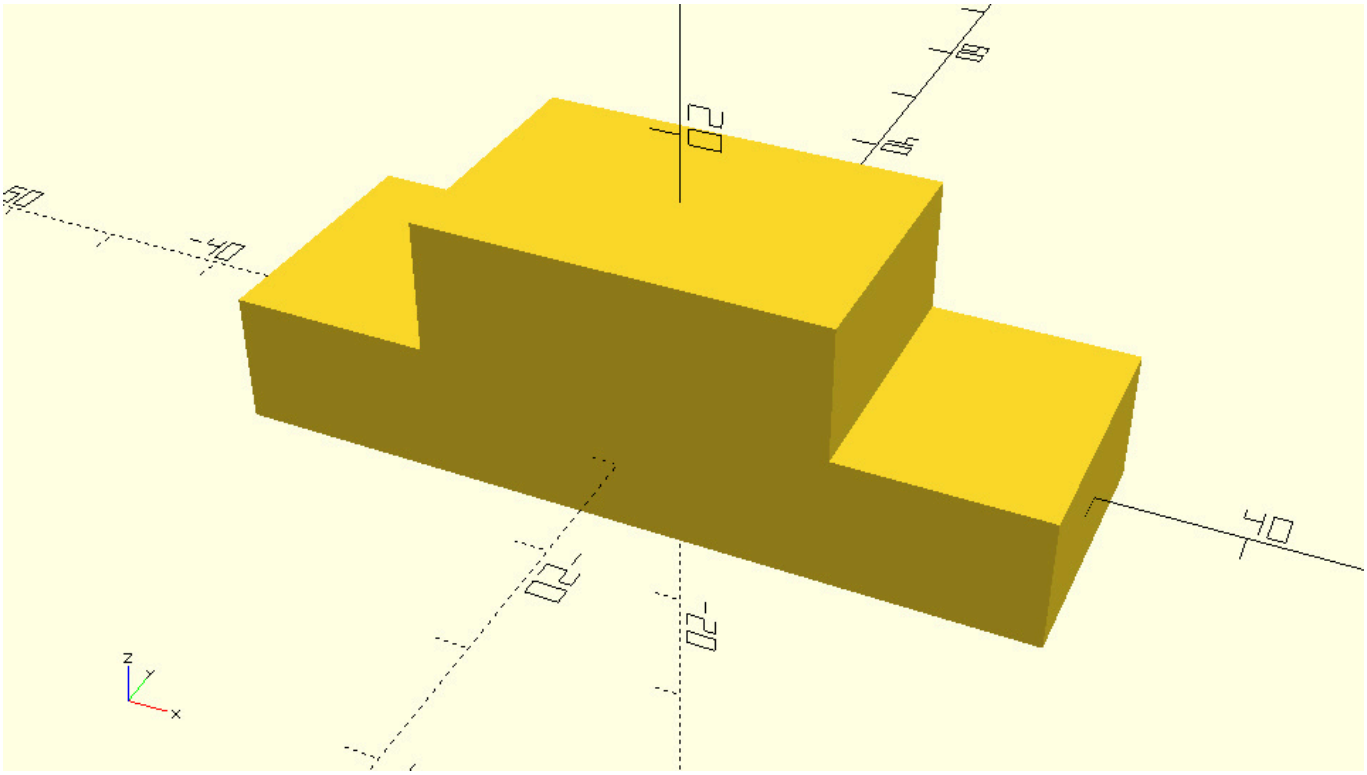


On the example above, the second cube sits exactly on top of the first cube. This is something that should be avoided as it's not clear to OpenSCAD whether the two cubes form one object together. This issue can be easily solved by always maintaining a small overlap of about 0.001 - 0.002 between the corresponding objects. One way to do so is by decreasing the amount of translation along the Z axis from 10 unit to 9.999 units.

Code

two_cubes_with_small_overlap.scad

```
cube([60,20,10],center=true);
translate([0,0,9.999])
    cube([30,20,10],center=true);
```

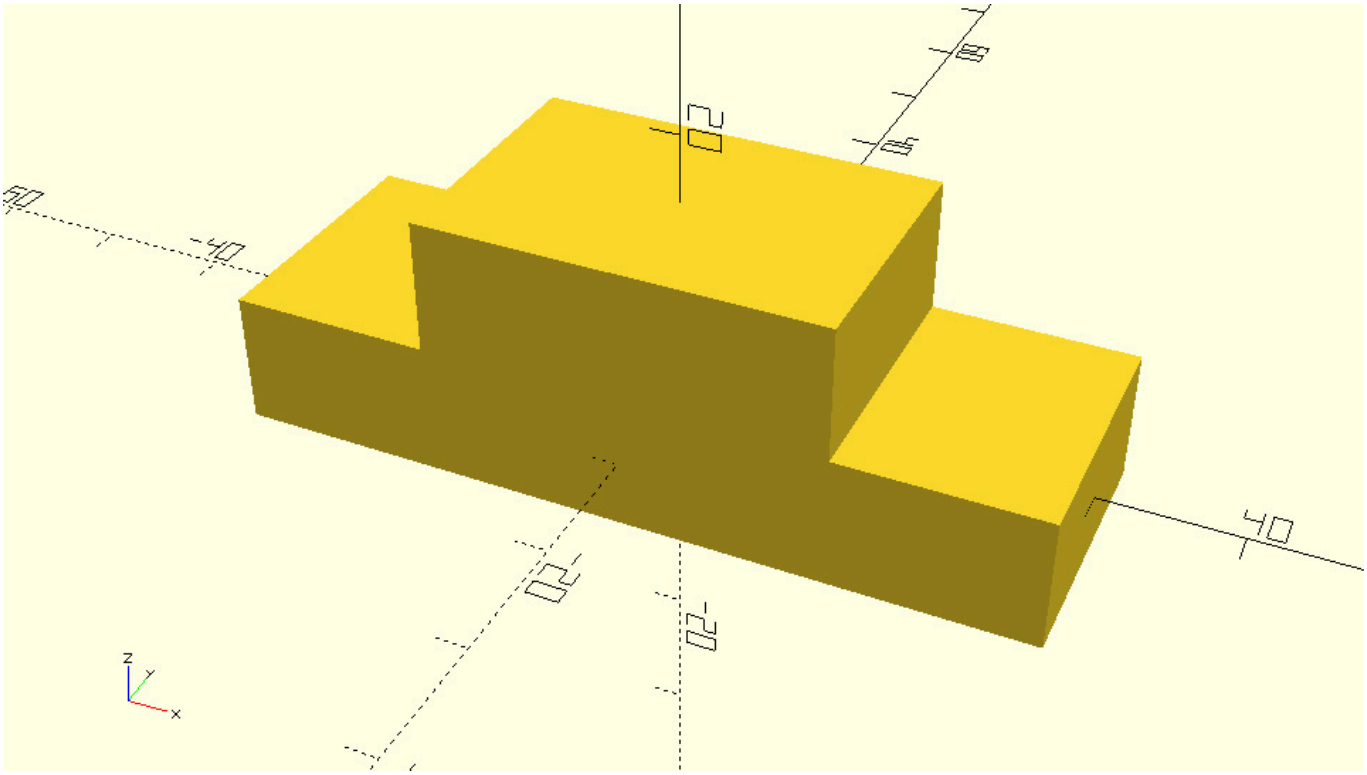


Another way to do so more explicitly is by subtracting 0.001 units from the corresponding value on the script.

Code

two_cubes_with_explicit_small_overlap.scad

```
cube([60,20,10],center=true);
translate([0,0,10 - 0.001])
    cube([30,20,10],center=true);
```



There is a third way. For not losing 0.001 from the top we could add a third cube with dimensions like the smaller cube, and height of 0.002 ([30, 20, 0.002]). The third cube will close the gap.

Code

third_cube_close_small_gap.scad

```
cube([60,20,10],center=true);
translate([0,0,10])
    cube([30,20,10],center=true);
translate([0,0,5 - 0.001])
    cube([30,20,0.002],center=true);
```

This is something you are going to encounter throughout the tutorial. When two objects are exactly touching each other, you should always guarantee a small overlap by subtracting or adding a tolerance of 0.001 units.

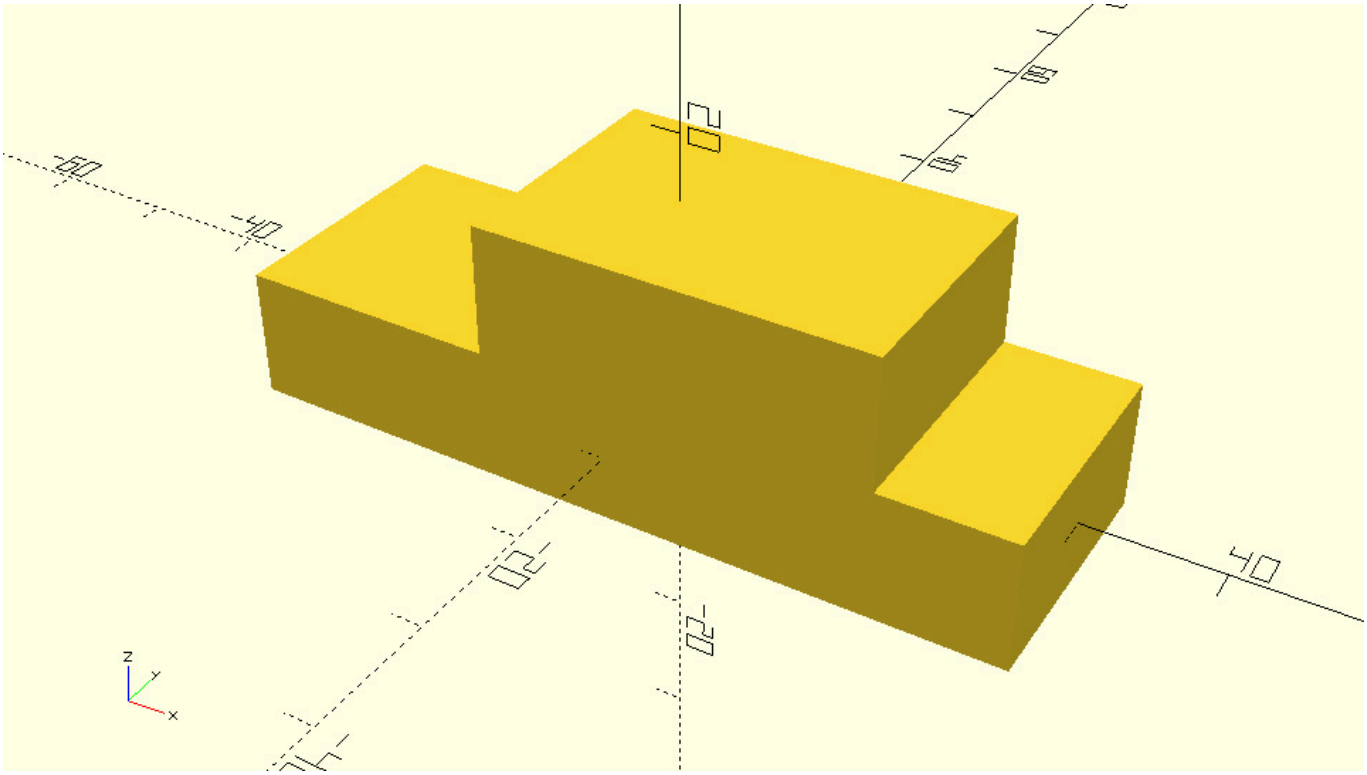
The cylinder primitive and rotating objects

The model that you just created looks like the body of a car that has bad aerodynamics. That's ok. You will be making the car look a lot more interesting and aerodynamic in the following chapters. For now, you are going to use the cylinder primitive and the rotate transformation to add wheels and axles to your car. You can create a wheel by adding a third statement that consists of the cylinder command. You will need to define two input parameters, h and r. The first one will be the height of the cylinder while the second one will be its radius.

Code

a_cylinder_covered_by_cubes.scad

```
cube([60,20,10],center=true);
translate([5,0,10 - 0.001])
    cube([30,20,10],center=true);
cylinder(h=3,r=8);
```

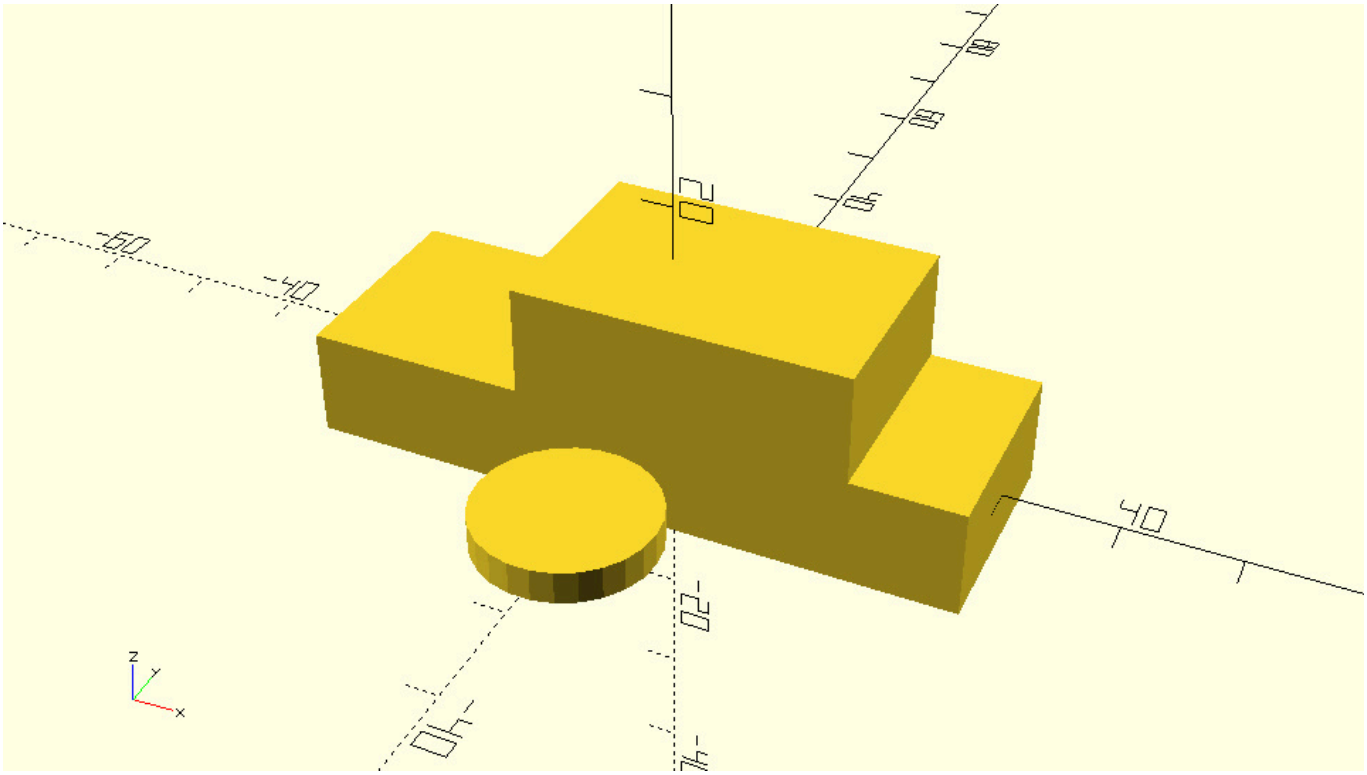


You should notice that the cylinder is hidden by the rest of the objects. You can use the translate command to make the cylinder visible by translating it 20 units along the negative direction of the Y axis.

Code

two_cubes_and_a_cylinder.scad

```
cube([60,20,10],center=true);
translate([5,0,10 - 0.001])
    cube([30,20,10],center=true);
translate([0,-20,0])
    cylinder(h=3,r=8);
```

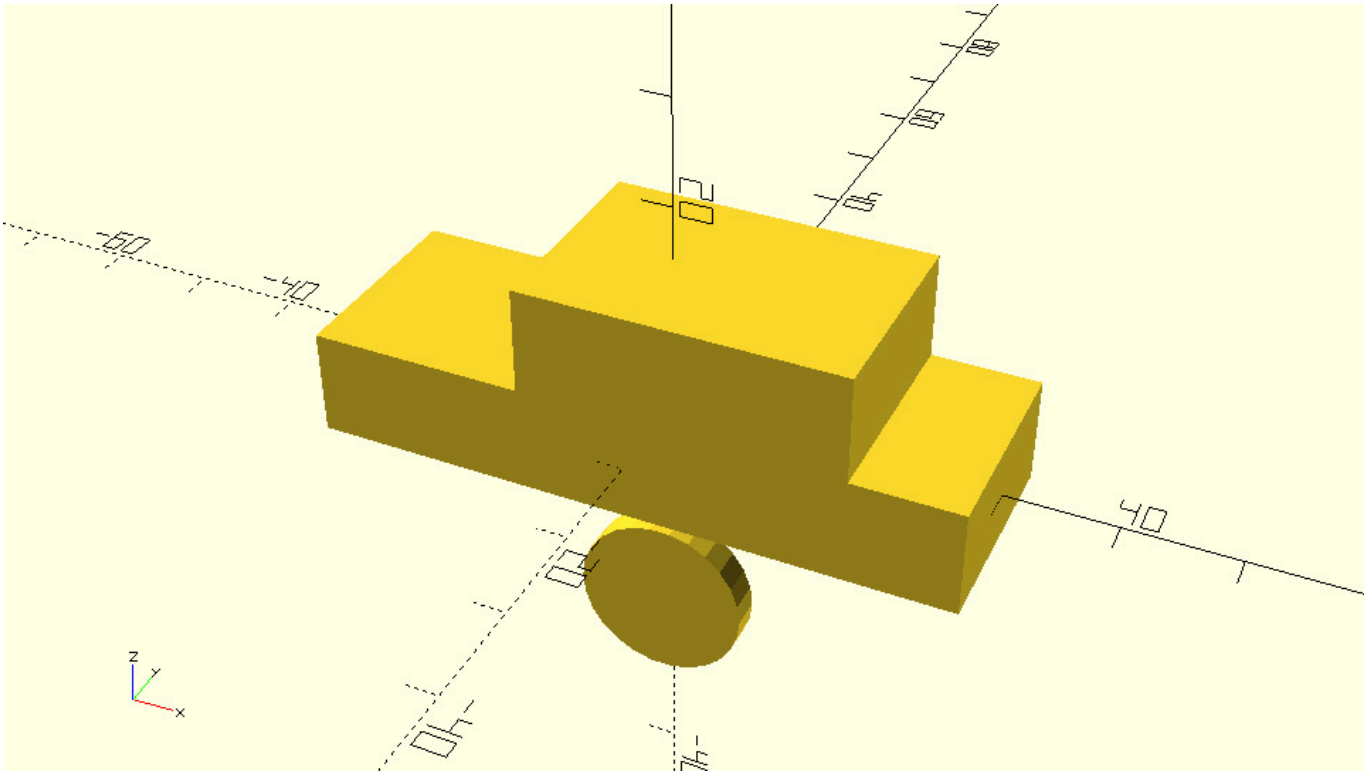


The wheel is now visible, but your car won't go anywhere if its not properly placed. You can use the rotate command to make the wheel stand straight. To do so you need to rotate it 90 degrees around the X axis.

Code

two_cubes_and_a_rotated_cylinder.scad

```
cube([60,20,10],center=true);
translate([5,0,10 - 0.001])
    cube([30,20,10],center=true);
rotate([90,0,0])
    translate([0,-20,0])
    cylinder(h=3,r=8);
```



The first thing you should notice is the absence of semicolon between the rotate and translate command. You should be already getting familiar with this concept. The semicolon is only added at the end of a statement. You can keep adding as many transformation commands as you like in a similar fashion, but you should not include a semicolon between them.

The second thing you should notice is that the rotate command has one input parameter, which is a vector of three values. In complete analogy to the translate command each value indicates how many degrees will an object be rotated around the X, Y and Z axis.

The third thing you should notice is that the wheel is standing straight but as a result of its rotation around the x axis it has moved below the car. This happened because the object was already moved away from the origin before it was rotated. A good practice for placing objects inside your model is first to rotate them and then to translate them to the desired position. Note that OpenSCAD generates the object, and then applies the object's transformations starting with the one immediately before the object definition and working backwards. So in order to rotate your object and then move it with a translation, specify the translation first in the code, followed by the rotation, followed by the object definition.

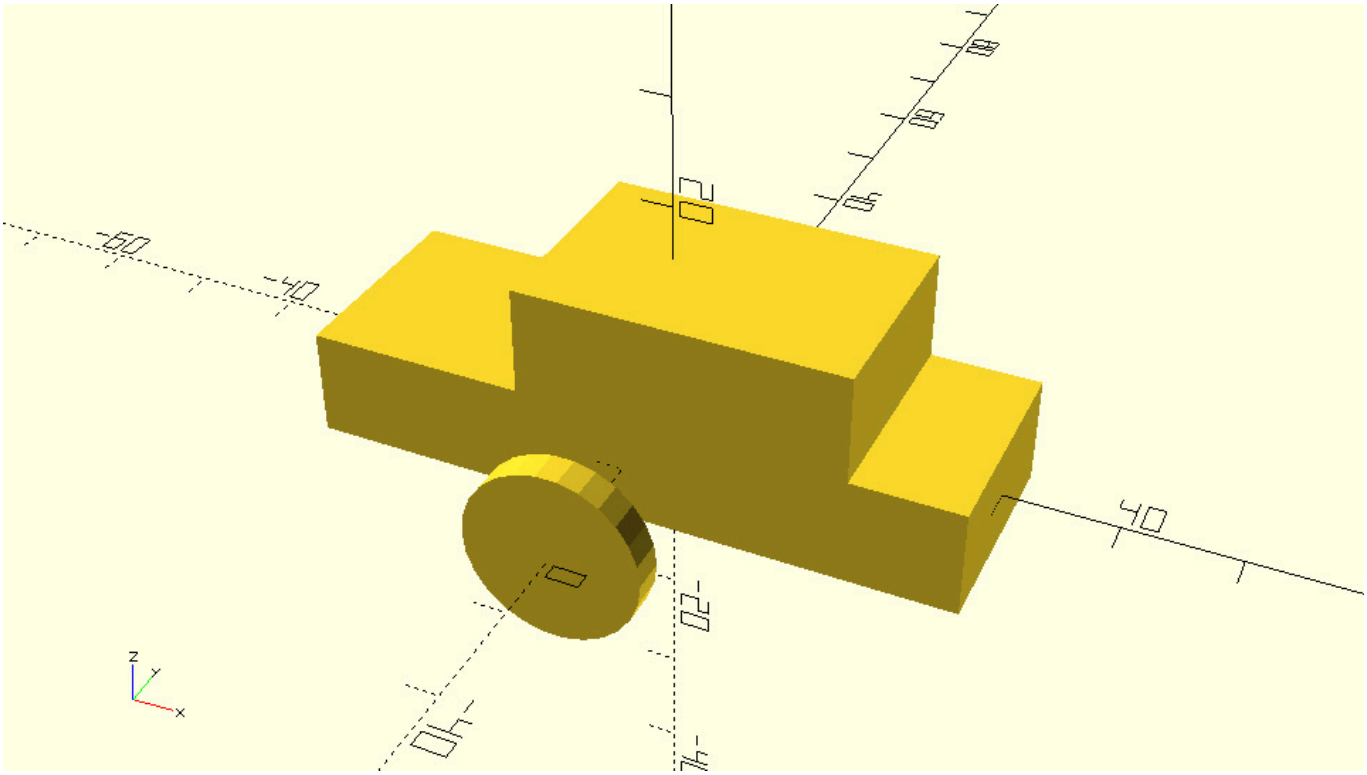
Exercise

Try first rotating the wheel and then translating it, by changing the order of the rotate and translate commands.

Code

two_cubes_and_a_rotated_and_translated_cylinder.scad

```
cube([60,20,10],center=true);
translate([5,0,10 - 0.001])
    cube([30,20,10],center=true);
translate([0,-20,0])
    rotate([90,0,0])
    cylinder(h=3,r=8);
```



It already looks a lot better than the previous position of the wheel.

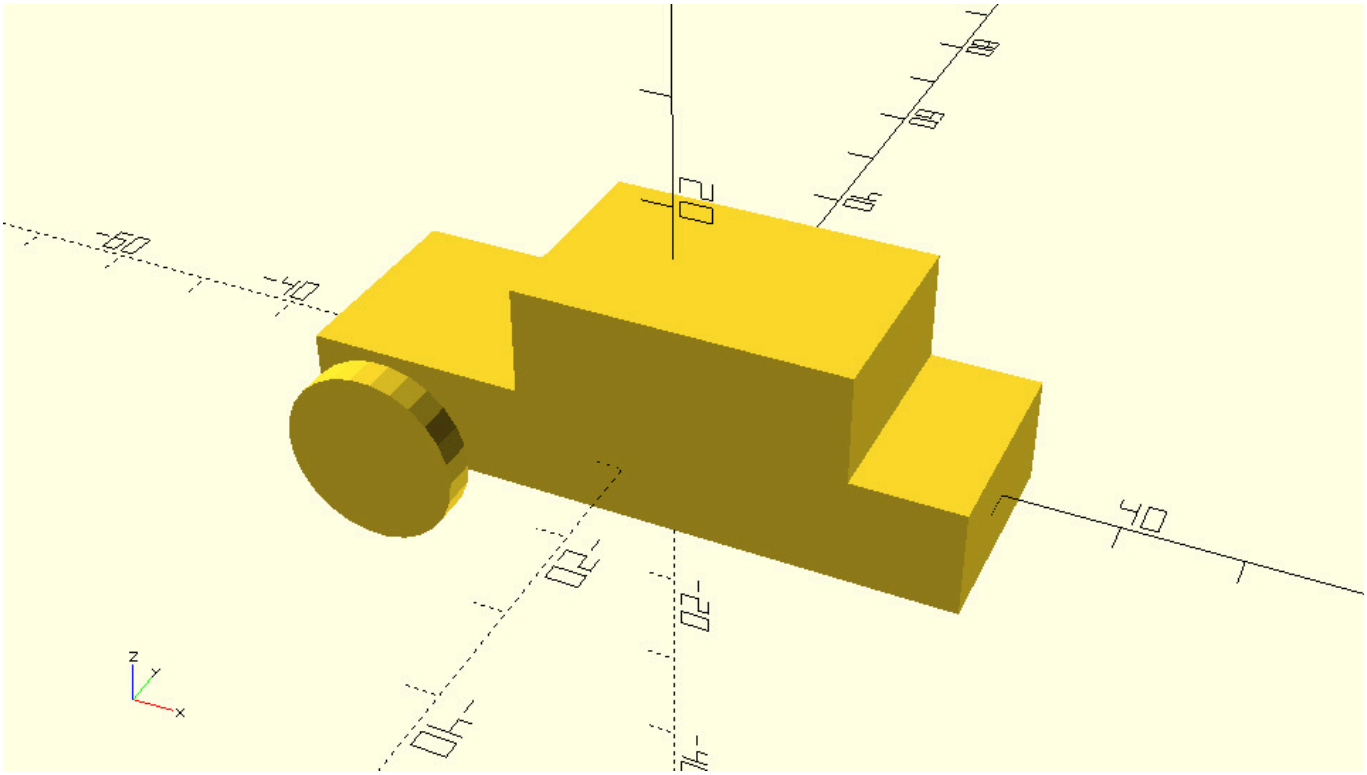
Exercise

Try modifying the input parameter of the translate command to make this wheel the front left wheel of your car.

Code

car_body_and_front_left_wheel.scad

```
cube([60,20,10],center=true);
translate([5,0,10 - 0.001])
    cube([30,20,10],center=true);
translate([-20,-15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8);
```



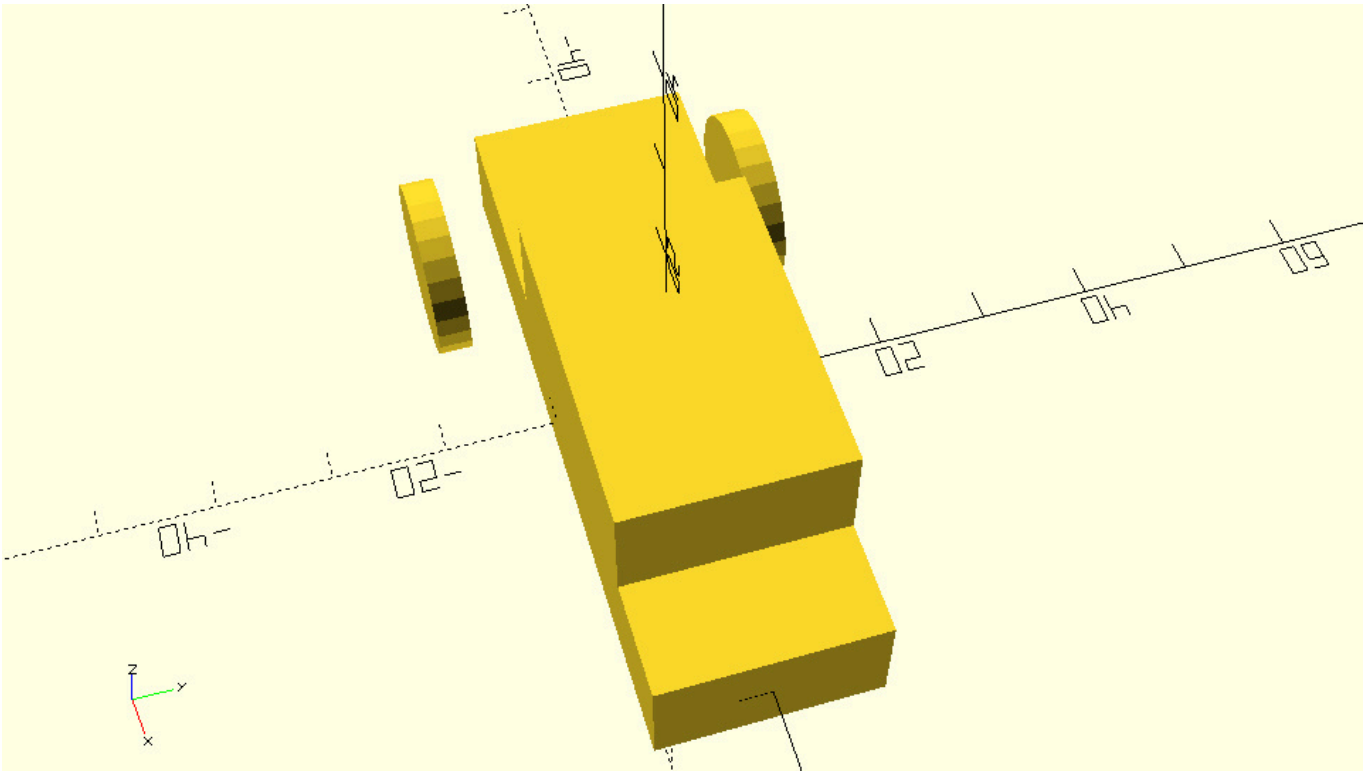
Exercise

Try adding the front right wheel of the car by duplicating the last statement and changing only the sign of one value.

Code

car_body_and_misaligned_front_wheels.scad

```
cube([60,20,10],center=true);
translate([5,0,10 - 0.001])
    cube([30,20,10],center=true);
translate([-20,-15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8);
translate([-20,15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8);
```



You should notice that the position of the wheels is not symmetric. This happened because the cylinder was not created centered on the origin.

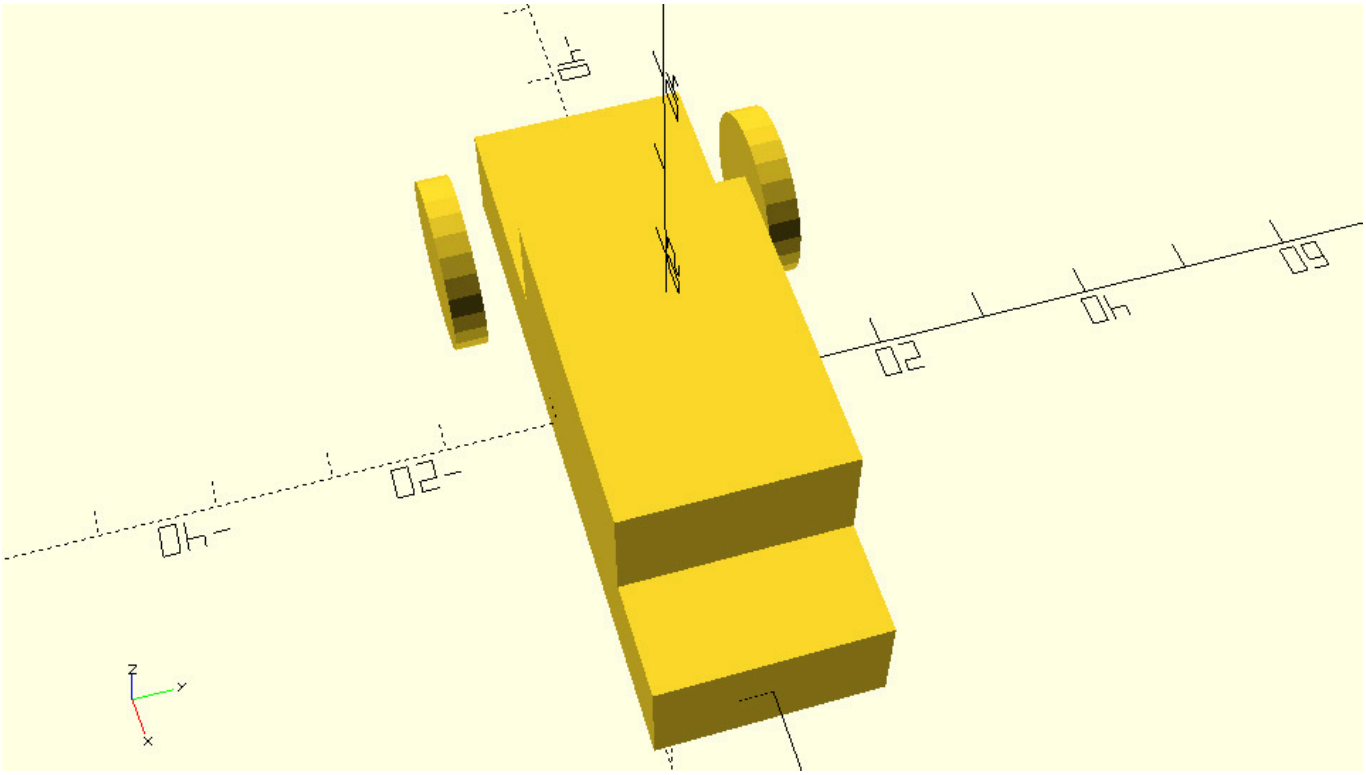
Exercise

Try adding an additional input parameter to the cylinder commands to denote to OpenSCAD that both wheels should be centered on the origin when first created. Is the position of your wheels symmetric now?

Code

car_body_and_aligned_front_wheels.scad

```
cube([60,20,10],center=true);
translate([5,0,10 - 0.001])
    cube([30,20,10],center=true);
translate([-20,-15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8,center=true);
translate([-20,15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8,center=true);
```



Completing your first model

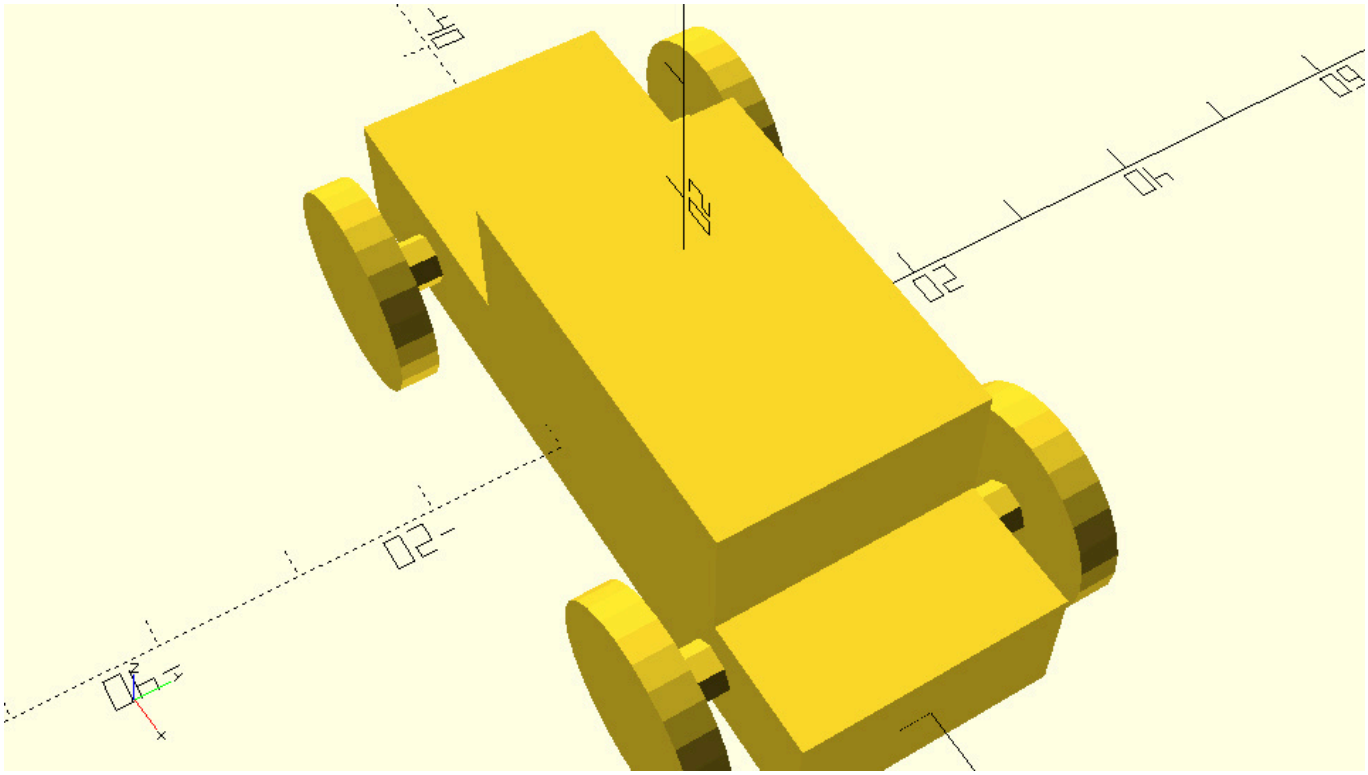
Exercise

Try using what you have learned to add the rear missing wheels to the car. Try adding a connecting axle to the front and rear wheels.

Code

completed_car.scad

```
cube([60,20,10],center=true);
translate([5,0,10 - 0.001])
    cube([30,20,10],center=true);
translate([-20,-15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8,center=true);
translate([-20,15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8,center=true);
translate([20,-15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8,center=true);
translate([20,15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8,center=true);
translate([-20,0,0])
    rotate([90,0,0])
    cylinder(h=30,r=2,center=true);
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=30,r=2,center=true);
```



You should notice that on the model above there is an overlap between the axles and the wheels that is equal to half the thickness of the wheels. If the model was created in a way that the wheels and the axles were just touching each other then there would be a need to ensure a small overlap between them as was done with the two cubes of the car's body.

One thing that may have bothered you so far is the resolution of the wheels. Until now you have been using OpenSCAD's default resolution settings. There are special commands in OpenSCAD language which allow you to have full control over the resolution of your models. Increasing the resolution of your model will also increase the required rendering time every time you update your design. For this reason, it is advised that you keep the default resolution settings when you are building your model and increase the resolution only after completing your design. Since the car example is completed, you can increase the resolution by adding the following two statements in your script.

Code

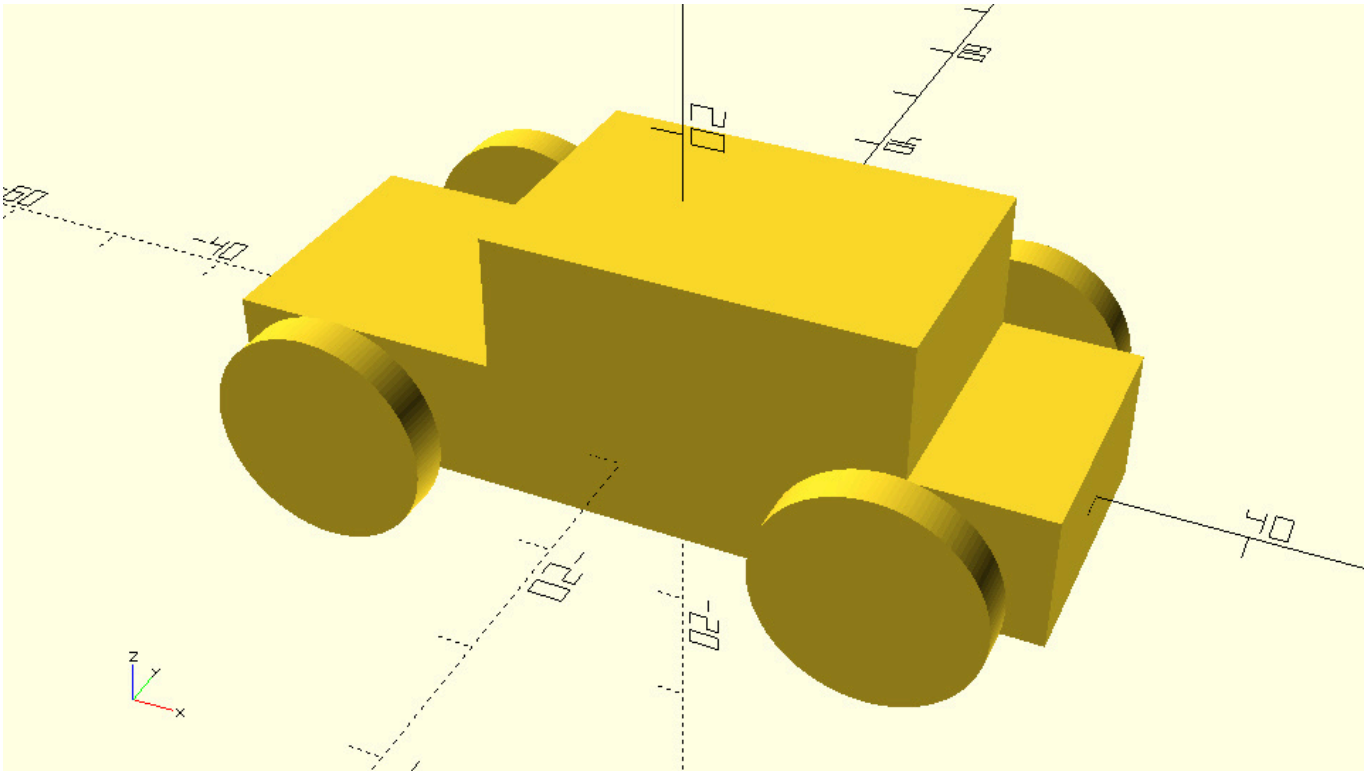
```
$fa = 1;
$fs = 0.4;
```

Try adding the above two statements at the beginning of the car's script. Do you notice any changes in the resolution of the wheels?

Code

completed_car_higher_resolution.scad

```
$fa = 1;
$fs = 0.4;
cube([60,20,10],center=true);
translate([5,0,10 - 0.001])
    cube([30,20,10],center=true);
translate([-20,-15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8,center=true);
translate([-20,15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8,center=true);
translate([20,-15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8,center=true);
translate([20,15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8,center=true);
translate([-20,0,0])
    rotate([90,0,0])
    cylinder(h=30,r=2,center=true);
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=30,r=2,center=true);
```



In reality $fa and $fs are special variables that determine the resolution of the model according to the values that have been assigned to them. Their exact function will be explained later and is something that you should not worry about yet. The only thing you need to keep in mind is that you can add these two statements in any script to achieve a resolution that is universally good for 3D printing. These two statements will be used in all examples throughout the tutorial in order to have visually appealing renderings.

Before sharing your script with your friends, it would be nice to include some comments to help them understand your script. You can use a double slash at the start of a line to write anything you like without affecting your model. By using a double slash OpenSCAD knows that what follows is

not part of the scripting language and simply ignores it.

Exercise

Try adding a comment above each statement to let your friends know what part of your model is created with each statement.

Solution
Code
[Expand]

It's time to save your model. Hit the save (third) icon on the action bar above the editor to save your script as a *.scad file. When you are creating a new model remember to save early and then save often to avoid accidentally losing your work.

If you would like to 3D print your car you can export it as an STL file. You should first hit the render (second) icon on the action bar below the viewport to generate the STL data of your model. You should then hit the export as STL icon on the action bar above the editor to save a STL file of your model.

Creating a second model

Exercise

Try using everything you learned to create a new simple model. It can be a house, an airplane or anything you like. Don't worry about making your model look perfect, just experiment with your new skills! You will keep learning more techniques to create awesome models in the following chapters.

Chapter 2

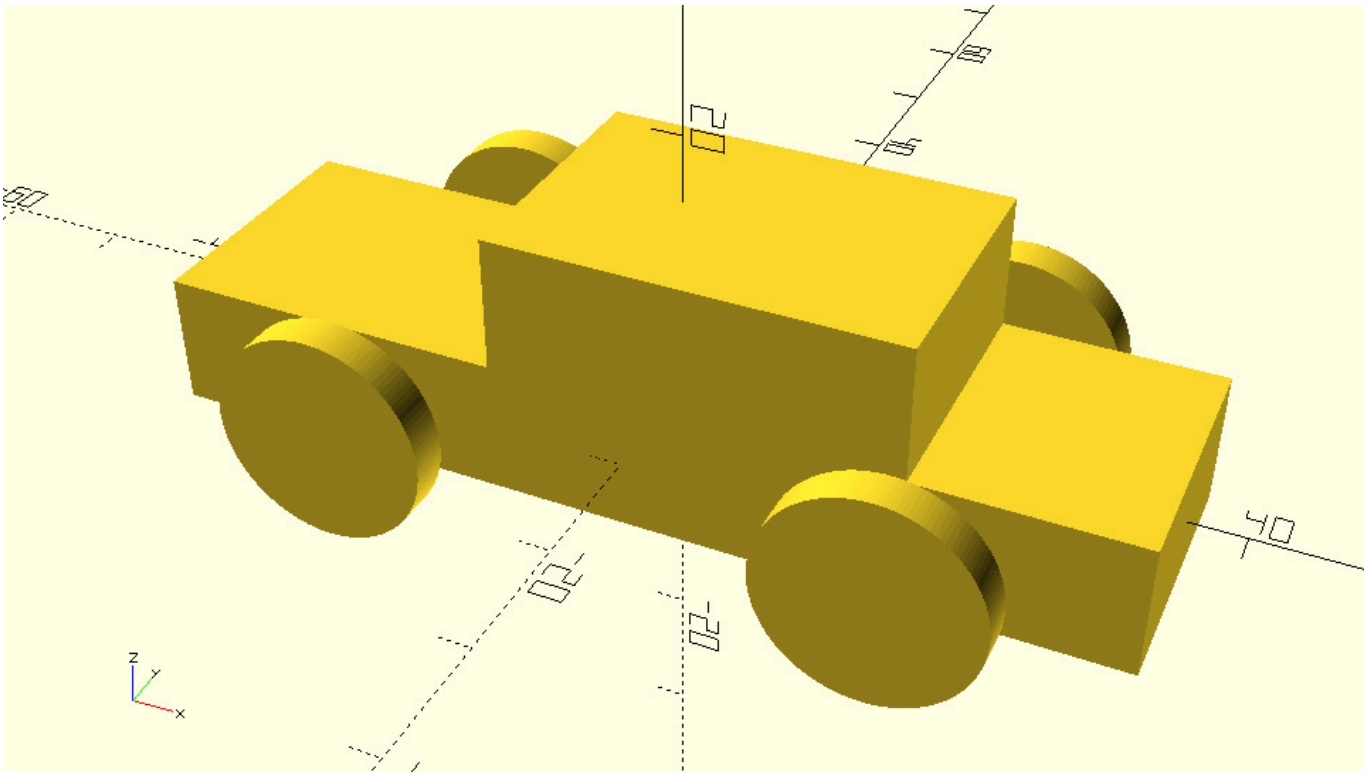
Scaling parts or the whole model

The model you created in the previous chapter was a great starting point for working with OpenSCAD, but perhaps after seeing it you recognized some aspects which should be changed. Here we will discuss strategies for modifying components of designs. One way to do so is by using the scale command, which is another one of the transformation commands. Modify the statement that creates the base of the car's body in the following way in order to increase the length of the body by a ratio of 1.2.

Code

car_with_lengthened_body_base.scad

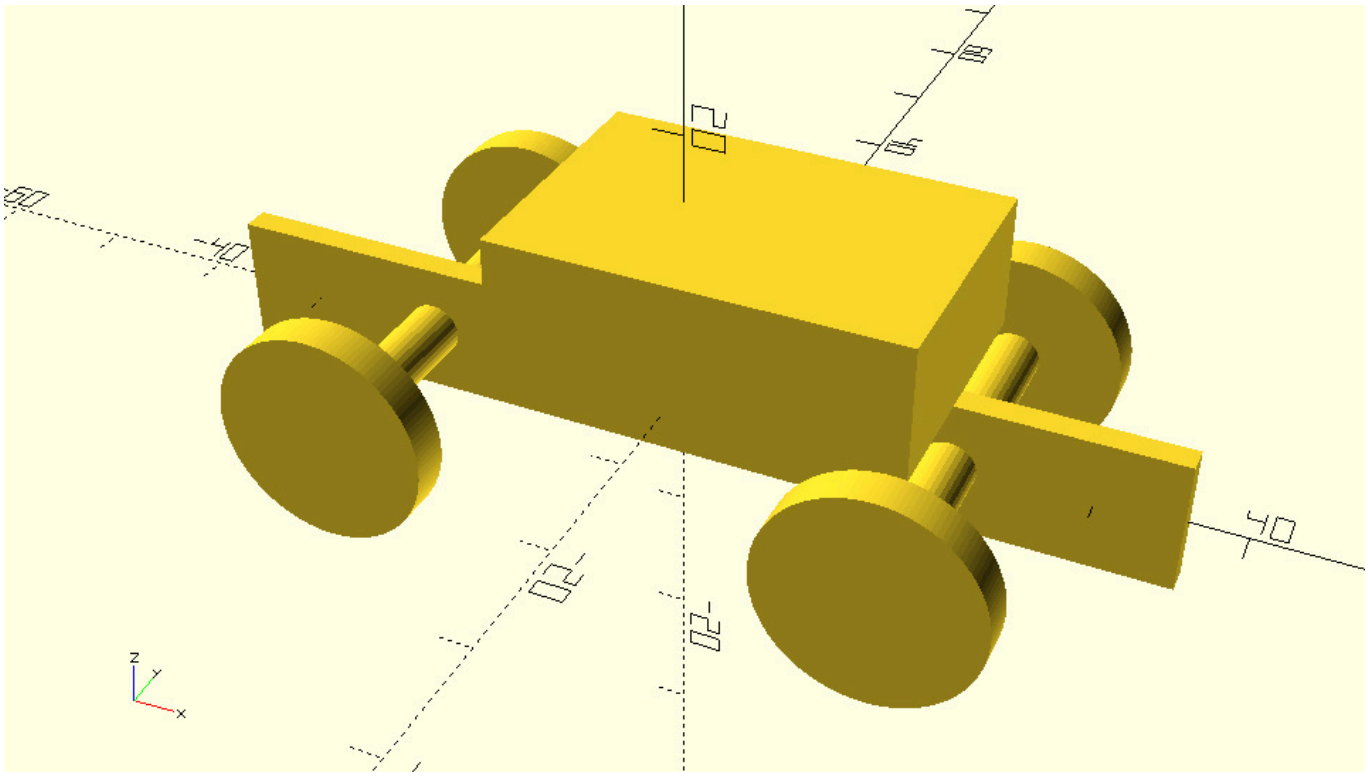
```
…
// Car body base
scale([1.2,1,1])
    cube([60,20,10],center=true);
…
```

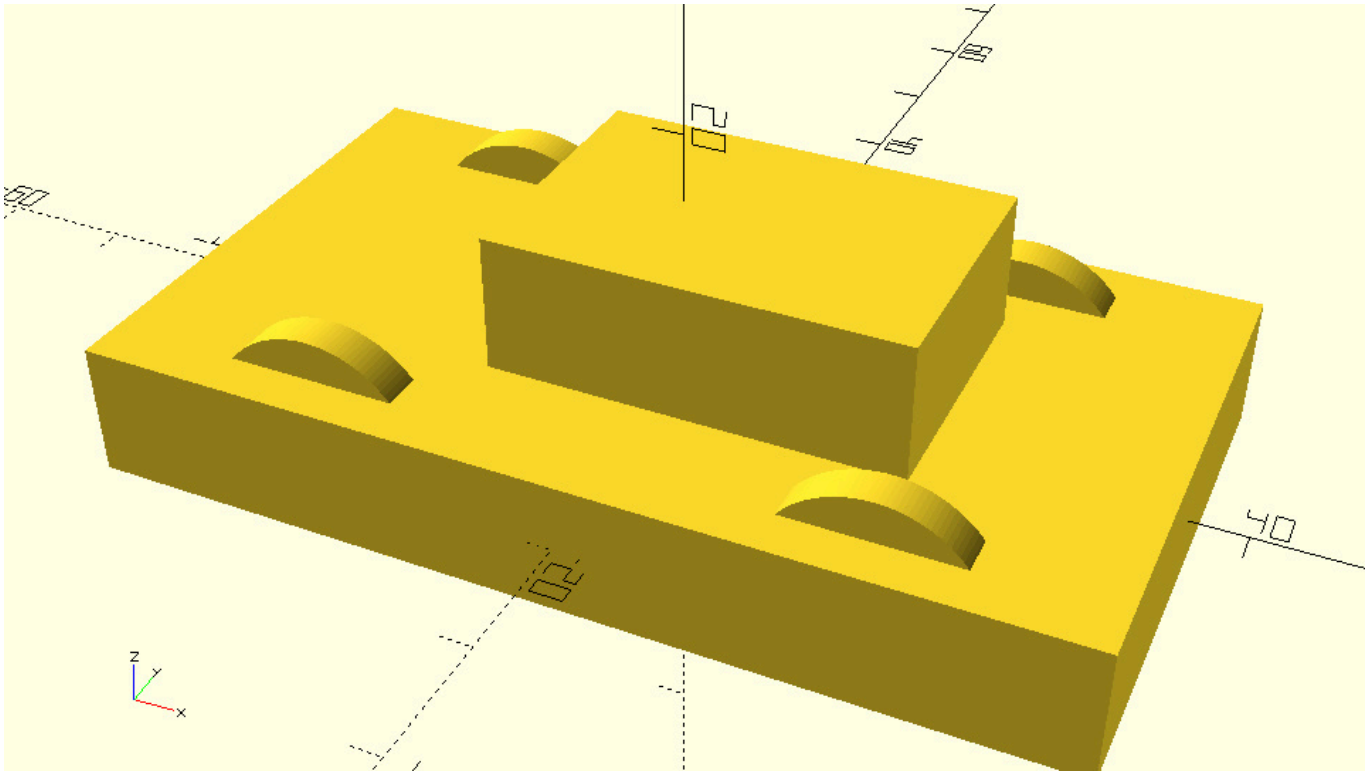


You should notice that the scale command is used like the transform and rotate commands. It is added to the left of an existing statement without including a semicolon in between and it has a vector of three values as an input parameter. In analogy to the translate and rotate commands each value corresponds to the scaling ratio along the X, Y and Z axis.

Exercise

Try modifying the input of the scale command in order to scale the base of the body by a factor of 1.2 along the X axis and a factor of 0.1 or 2 along the Y axis. Did you get anything that could be a Mars rover or a tank? Are you surprised with how different the models look compared to the original car?



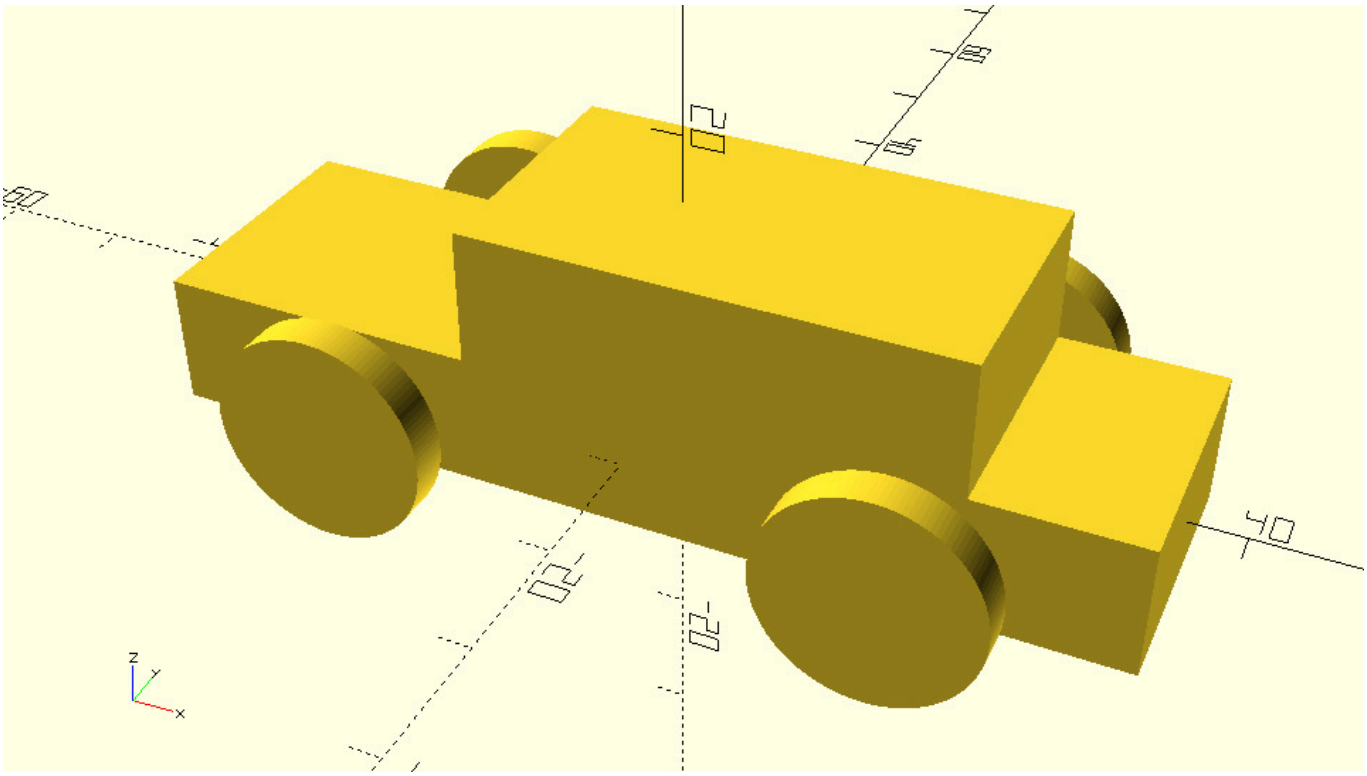


It is also possible to apply the same scale command or any other transformation command to more than one objects. Use the following code to apply the scale command to both the base and the top of the car's body.

Code

car_with_lengthened_body.scad

```
scale([1.2,1,1]) {
    // Car body base
    cube([60,20,10],center=true);
    // Car body top
    translate([5,0,10 - 0.001])
        cube([30,20,10],center=true);
}
```



The first thing you should notice is that in order to apply the scale command to more than one object, a set of curly brackets is used. The statements that define the corresponding objects along with their semicolons are placed inside the curly brackets. The curly brackets don't require a semicolon at the end.

The second thing you should notice is how the use of white space and comments increase the readability of your script. The following script is exactly equivalent, you can decide for yourself which one you'd rather have to read.

Code

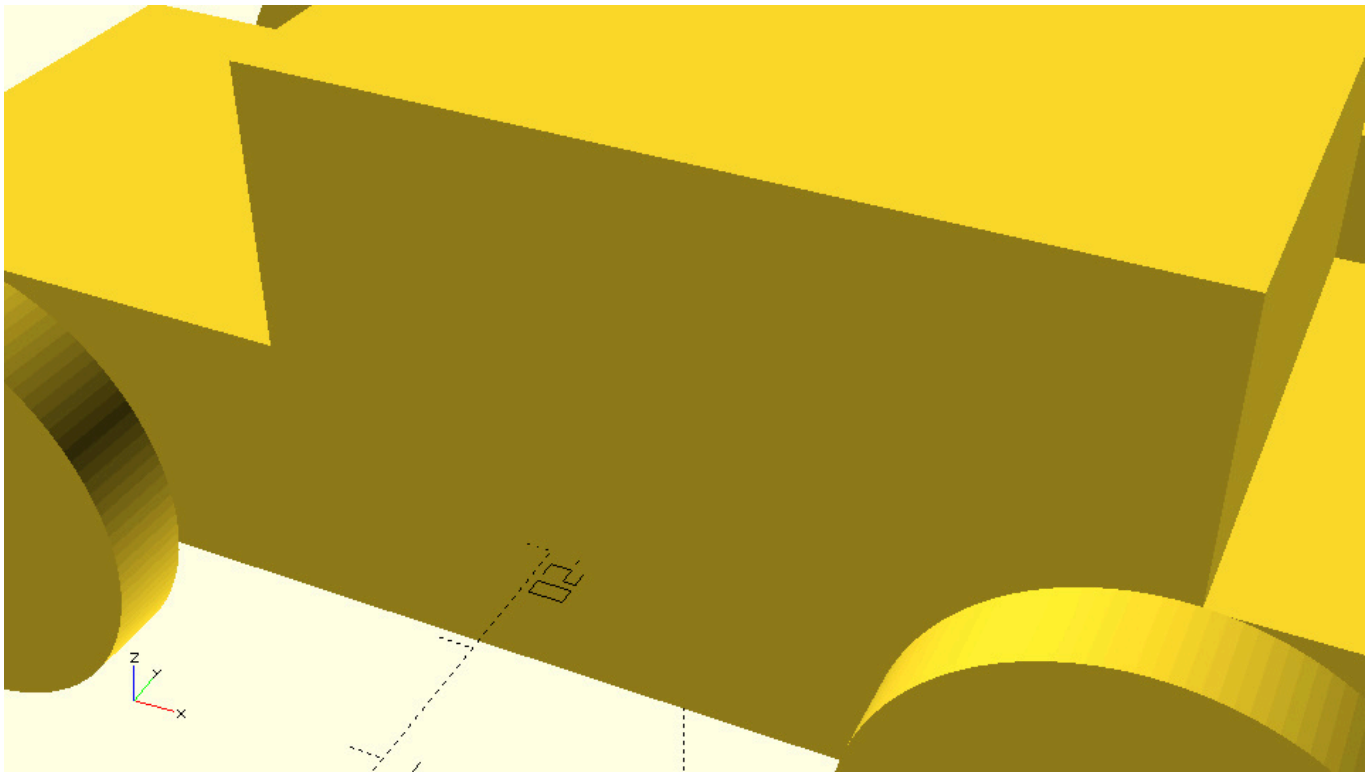
```
scale([1.2,1,1]) {
    cube([60,20,10],center=true);
    translate([5,0,10 - 0.001])
        cube([30,20,10],center=true);
}
```

Exercise

Try applying the scale command to your whole model. Did you remember to include all statements inside the curly brackets? What should be the relation between the scaling factors along the X and Z axis so that the wheels don't deform? What should the scaling factors be to get a car that has the same proportions but double the size?

- For the wheels not to deform, the scaling factors along the X and Z axis should be equal.

Code
[Expand]



Quick quiz

The following script is the model you created in the first chapter.

Code

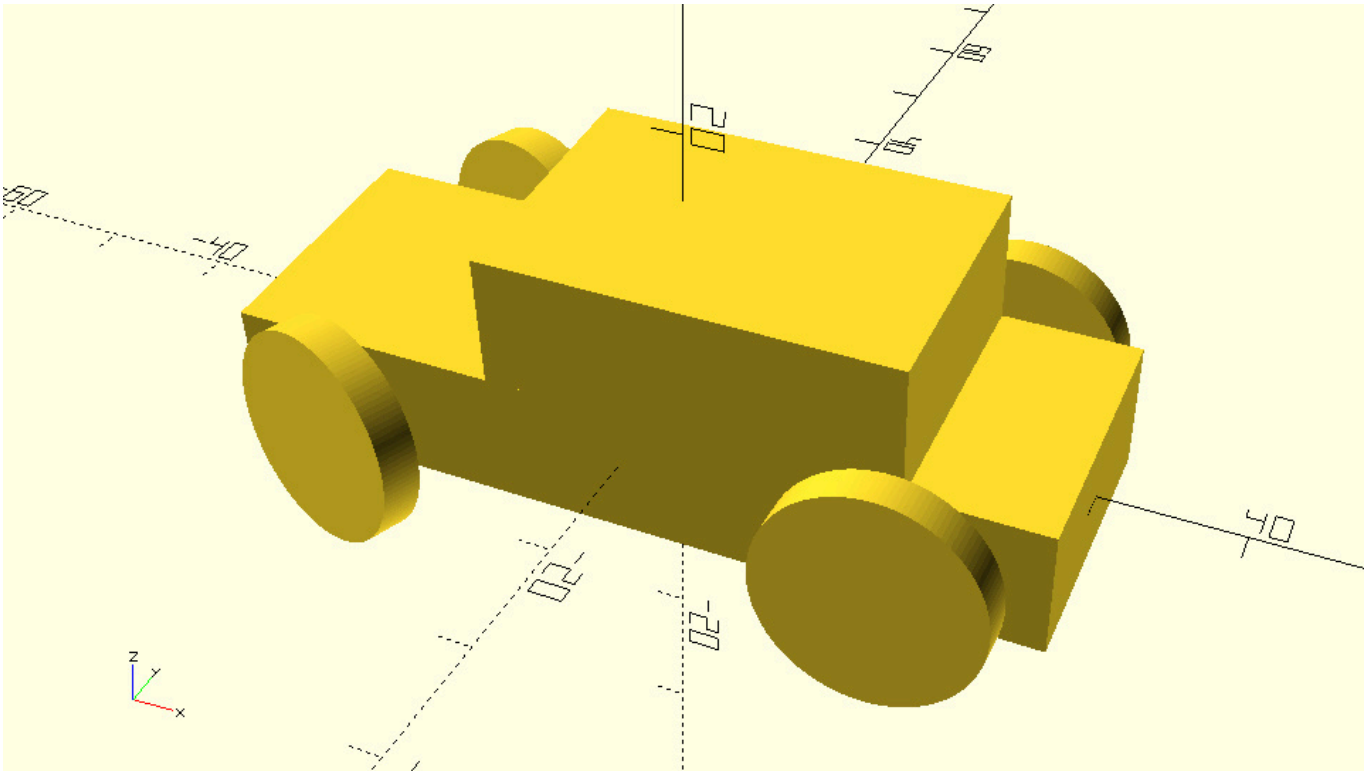
```
$fa = 1;
$fs = 0.4;
// Car body base
cube([60,20,10],center=true);
// Car body top
translate([5,0,10 - 0.001])
    cube([30,20,10],center=true);
// Front left wheel
translate([-20,-15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8,center=true);
// Front right wheel
translate([-20,15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8,center=true);
// Rear left wheel
translate([20,-15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8,center=true);
// Rear right wheel
translate([20,15,0])
    rotate([90,0,0])
    cylinder(h=3,r=8,center=true);
// Front axle
translate([-20,0,0])
    rotate([90,0,0])
    cylinder(h=30,r=2,center=true);
// Rear axle
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=30,r=2,center=true);
```

Exercise

Try rotating the front wheels by 20 degrees around the Z axis, as if the car was making a right turn. In order to make your model more convincing, try rotating the body of the car (base and top) by 5 degrees around the X axis in the opposite direction of the turn. To turn the wheels, modify the input parameters of existing rotate commands, to turn the body add a new rotate command.

Code

[Expand]



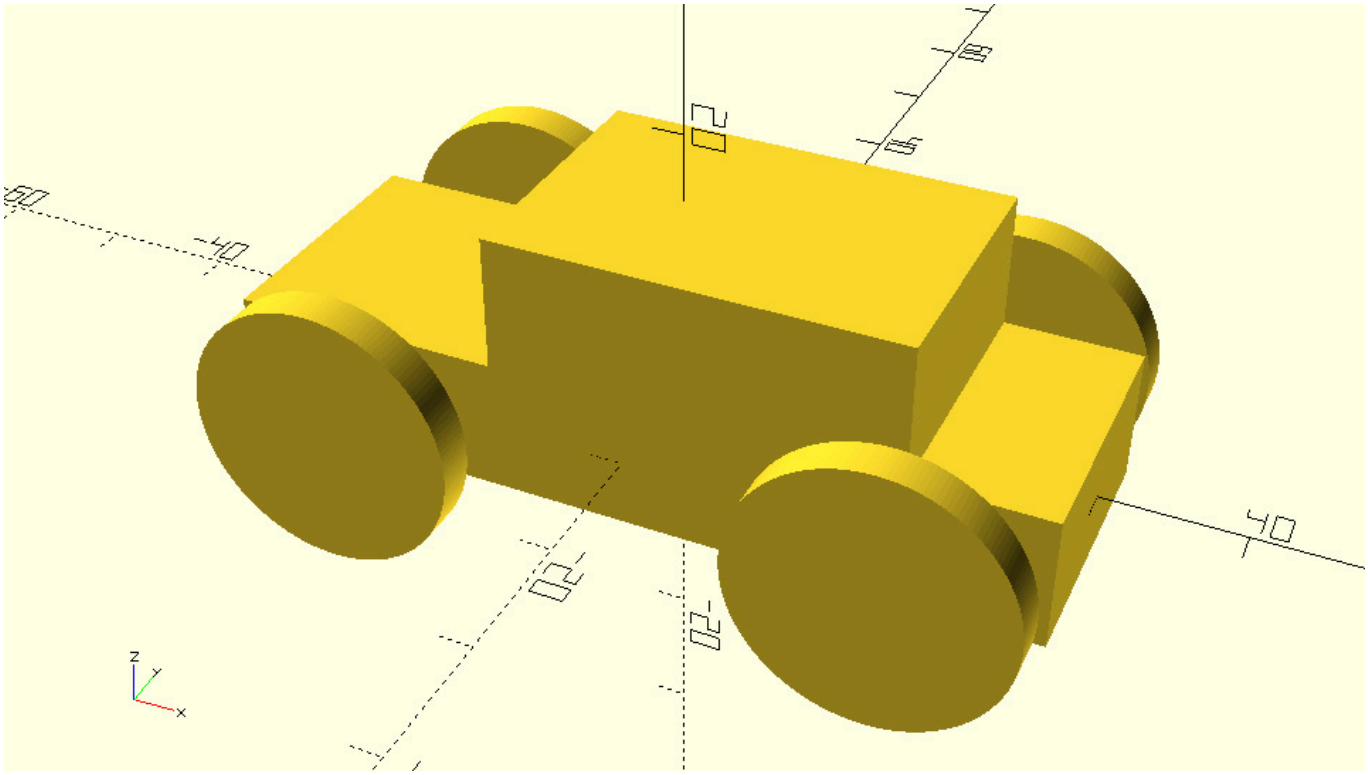
Parameterizing parts of your model

You should have gotten the point that a model is most of the times not intended to exist in one version. One of the powers of OpenSCAD scripting language lies in making easy the ability to reuse models over and over again or simply to play around with them until you are satisfied to commit to a final version. It's time to make some modifications to your car!

Exercise

Try changing the radius of the wheels to 10 units. How easily did you find which values to modify? Did you have to do the same thing four times?

Code
[Expand]



Although it wasn't that hard to change the size of wheels it could have been much simpler. First, it could have been easier to find which values to change. Second, you could have only one value to change since all wheels have the same radius. All this can be achieved with the use of variables. In the following script a variable for the radius of the wheels is introduced.

Code

```
wheel_radius = 8;
// Front left wheel
translate([-20,-15,0])
    rotate([90,0,0])
    cylinder(h=3,r=wheel_radius,center=true);
// Front right wheel
translate([-20,15,0])
    rotate([90,0,0])
    cylinder(h=3,r=wheel_radius,center=true);
// Rear left wheel
translate([20,-15,0])
    rotate([90,0,0])
    cylinder(h=3,r=wheel_radius,center=true);
// Rear right wheel
translate([20,15,0])
    rotate([90,0,0])
    cylinder(h=3,r=wheel_radius,center=true);
```

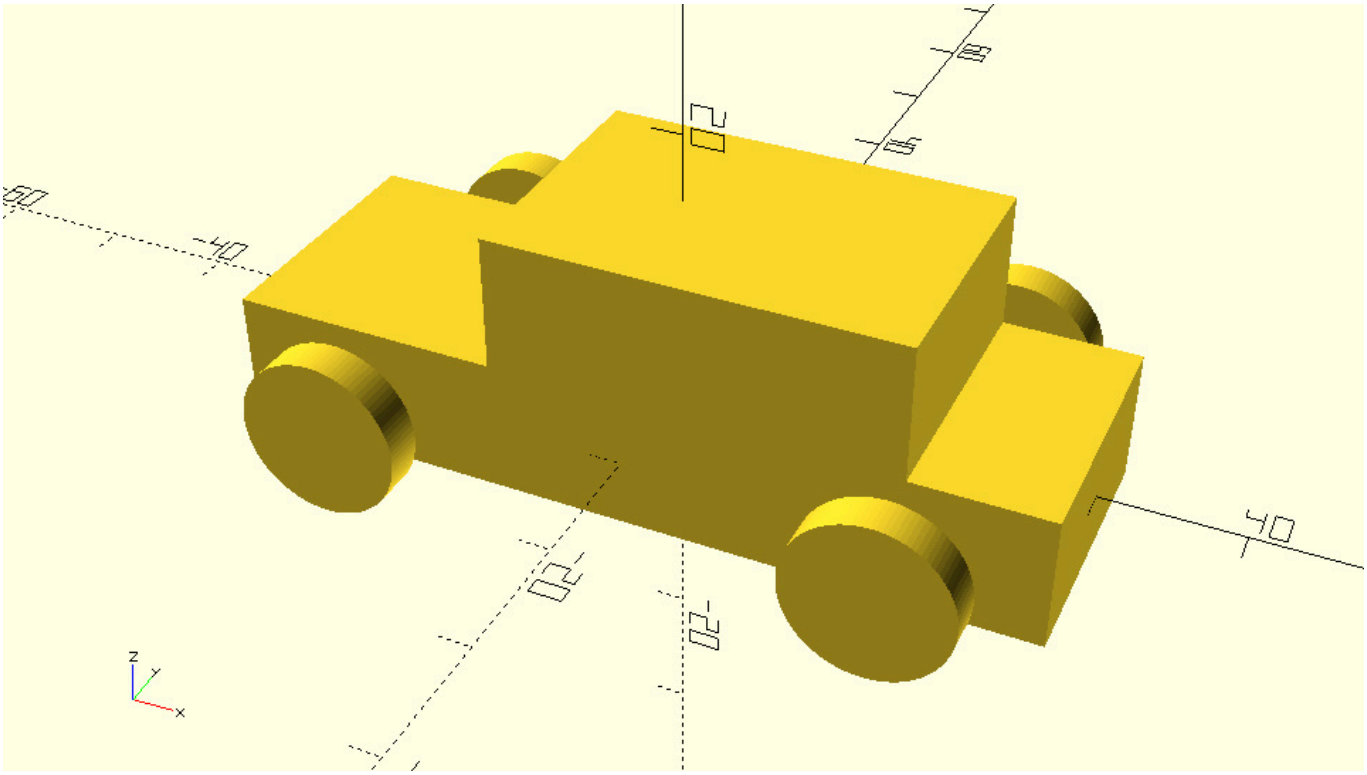
Every variable has two parts: a name and a value. In this example, the variable name is "wheel_radius". A valid variable name uses only alphanumeric characters and underscores (A-Z, a-z, 0-9, and _). After the variable name, an equals sign separates the name from the value, and is followed by the value itself. Finally, a semicolon is required at the end to denote the completion of that statement. It's a good practice to keep your variables organized by defining them all at the top of the document.

Once a variable is defined, it can be used in the code to represent its value. In this example, the cylinder commands have been modified to use the wheel_radius variable for the input parameter r. When OpenSCAD evaluates this script, it will set the input parameter r equal to the value of the wheel_radius variable.

Exercise

Try using a variable named wheel_radius to define the size of your car's wheels. Try changing the size of the wheels a few times by modifying the value of the wheel_radius variable. How much easier did you find changing the size of the wheels using the wheel_radius variable?

Code
[Expand]

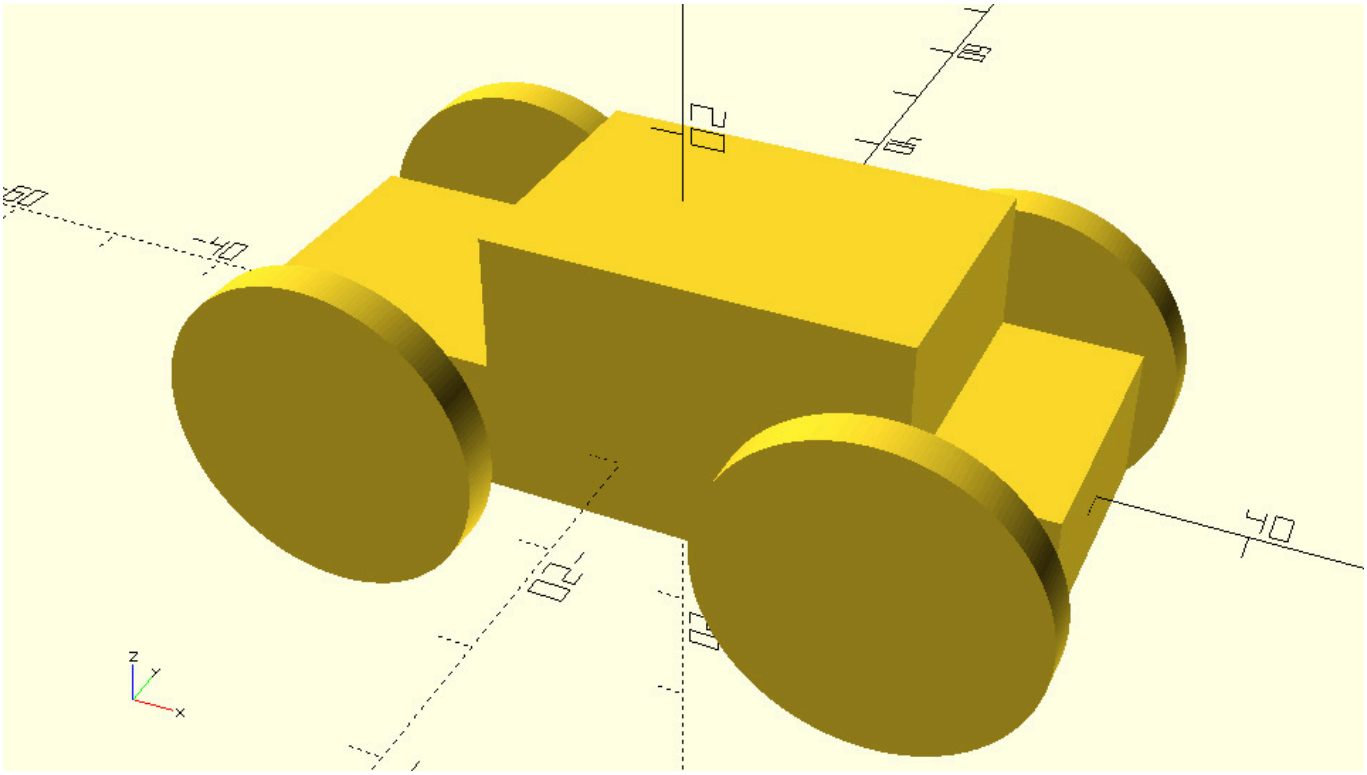


There is one important thing you should keep in mind about the behavior of variables in OpenSCAD. The variables in OpenSCAD behave like constants. They can hold only one value which they keep throughout the creation of your model. So, what happens if you assign a value to wheel_radius at the start of your script and then assign a new value to it after the definition of the two front wheels? Will the rear wheels have different size compared to the front wheels?

Exercise

Try assigning a different value to the wheel_radius variable right after the definition of the front wheels. Does your car have different front and rear wheel size?

Code
[Expand]



You should notice that all wheels have the same size. If multiple assignments to a variable exist, OpenSCAD uses the value of the last assignment. Even statements that make use of this variable and are defined before the last value assignment, will use the value of the last assignment.

OpenSCAD will also give a warning in this case: *WARNING: wheel_radius was assigned on line 3 but was overwritten on line 17*

Note

A variable assignment within { curly braces } only applies within those braces. Duplicate assignments at different levels of brace enclosure are not considered to conflict.

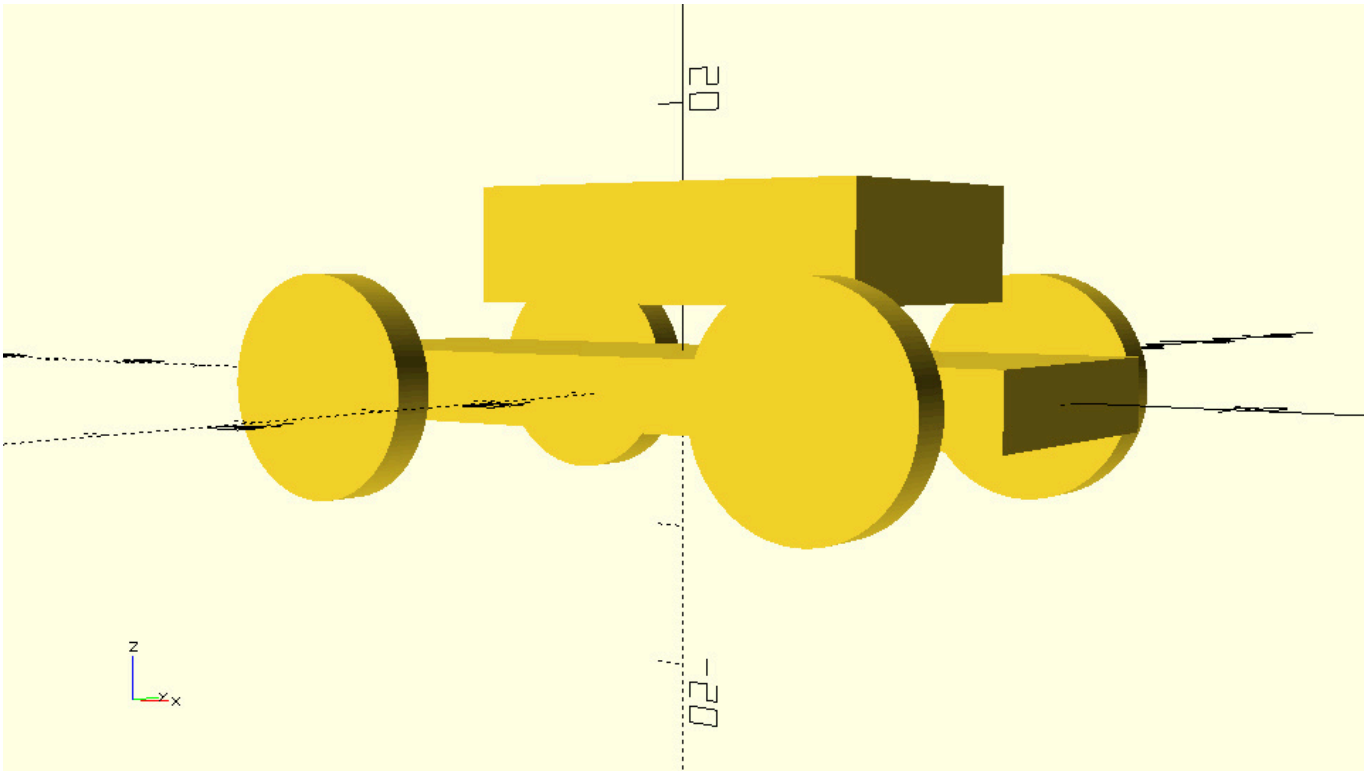
Parameterizing more parts of your model

You can now easily play around with the size of the wheels. It would be nice if you were able to customize more aspects of your model with such ease. You should notice for a moment that modifying the size of the wheels doesn't affect any other aspect of your model, it doesn't break your model in any way. This is not always the case.

Exercise

Try modifying the height of the car's body base and top by defining a base_height and a top_height variable and making the appropriate changes to the corresponding statements that define the base and the top. Assign the value 5 to the base_height variable and the value 8 to the top_height variable. What do you notice?

Code
[Expand]

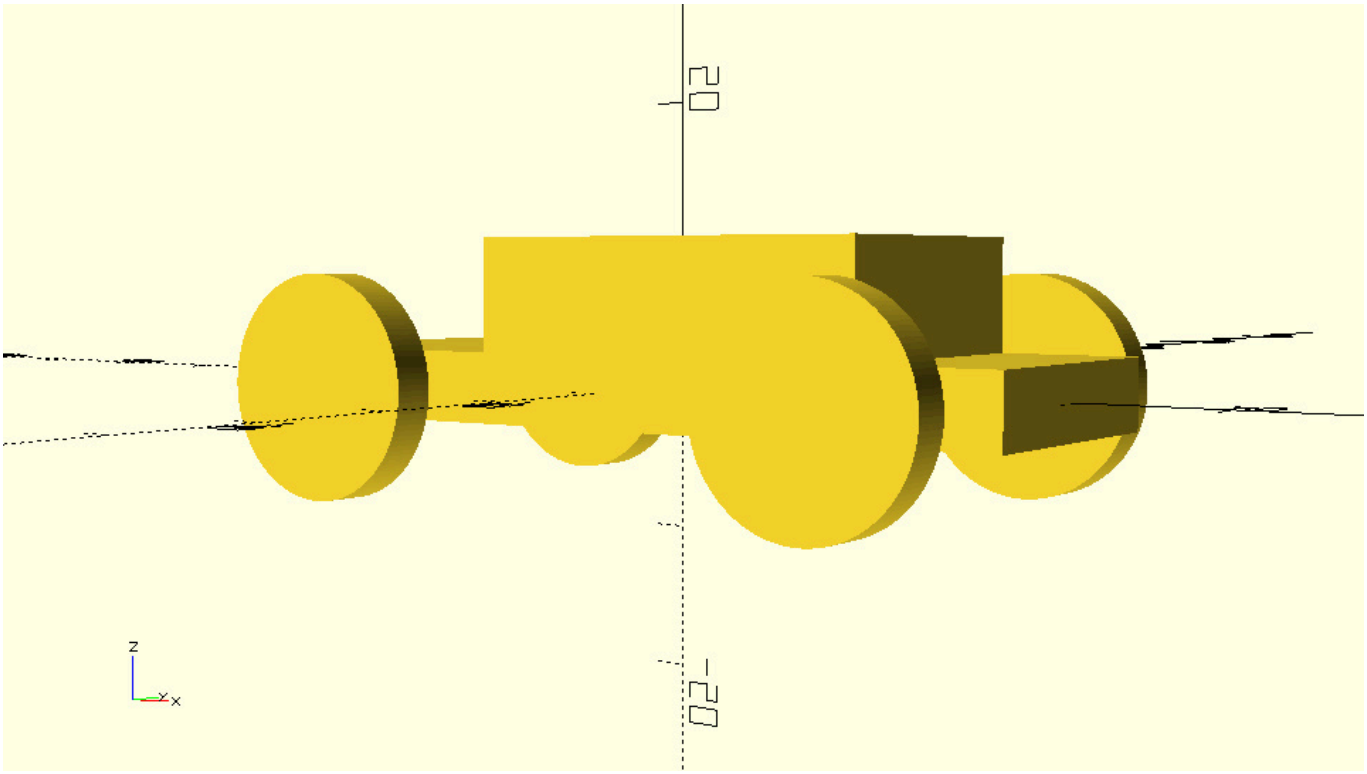


It is obvious that the body of the car stops being one as the base and the top separate. This happened because the correct position of the body's top is dependent on the height of the body's base and the height of body's top. Remember that in order to make the top sit on top of the base you had to translate the top along the Z axis by an amount equal to half the height of the base plus half the height of the top. If you want to parameterize the height of the base and the top you should also parameterize the translation of the top along the Z axis.

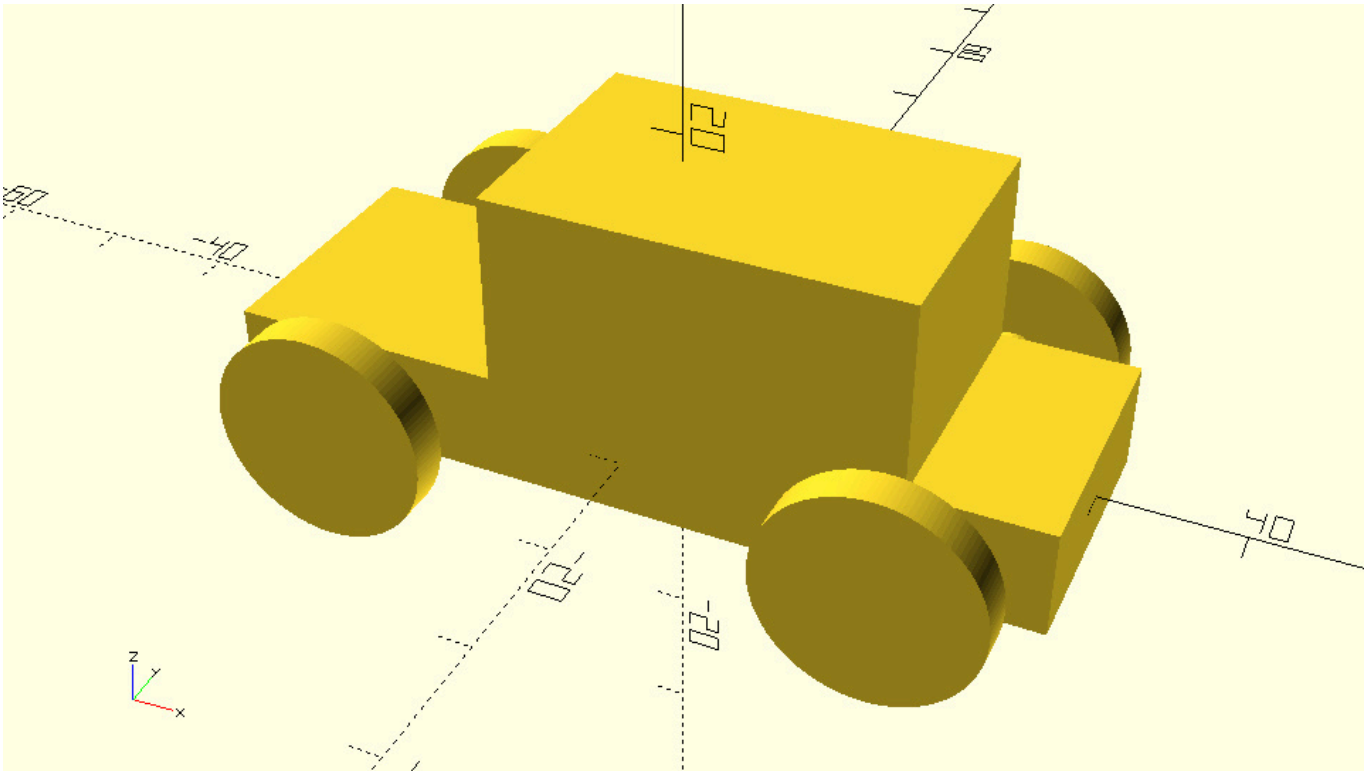
Exercise

Try parameterizing the translation of the body's top along the Z axis using the base_height and top_height variables to make it sit on top of the body's base. Try assigning different values to the base_height and top_height variables. Does the position of the body's top remain correct?

Code
[Expand]



Code
[Expand]

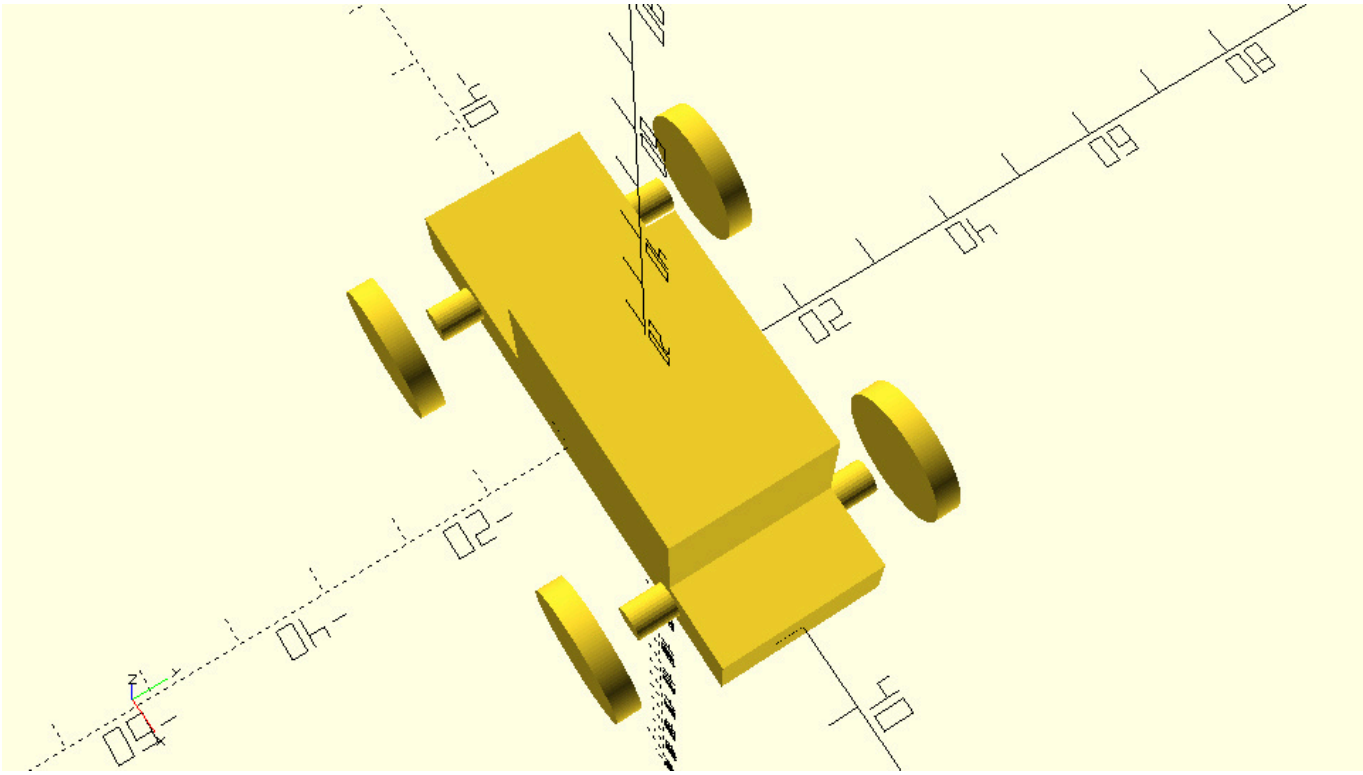


You should remember that every time you parameterize some aspect of your model you should also parameterize additional dependent aspects to prevent your model from breaking apart.

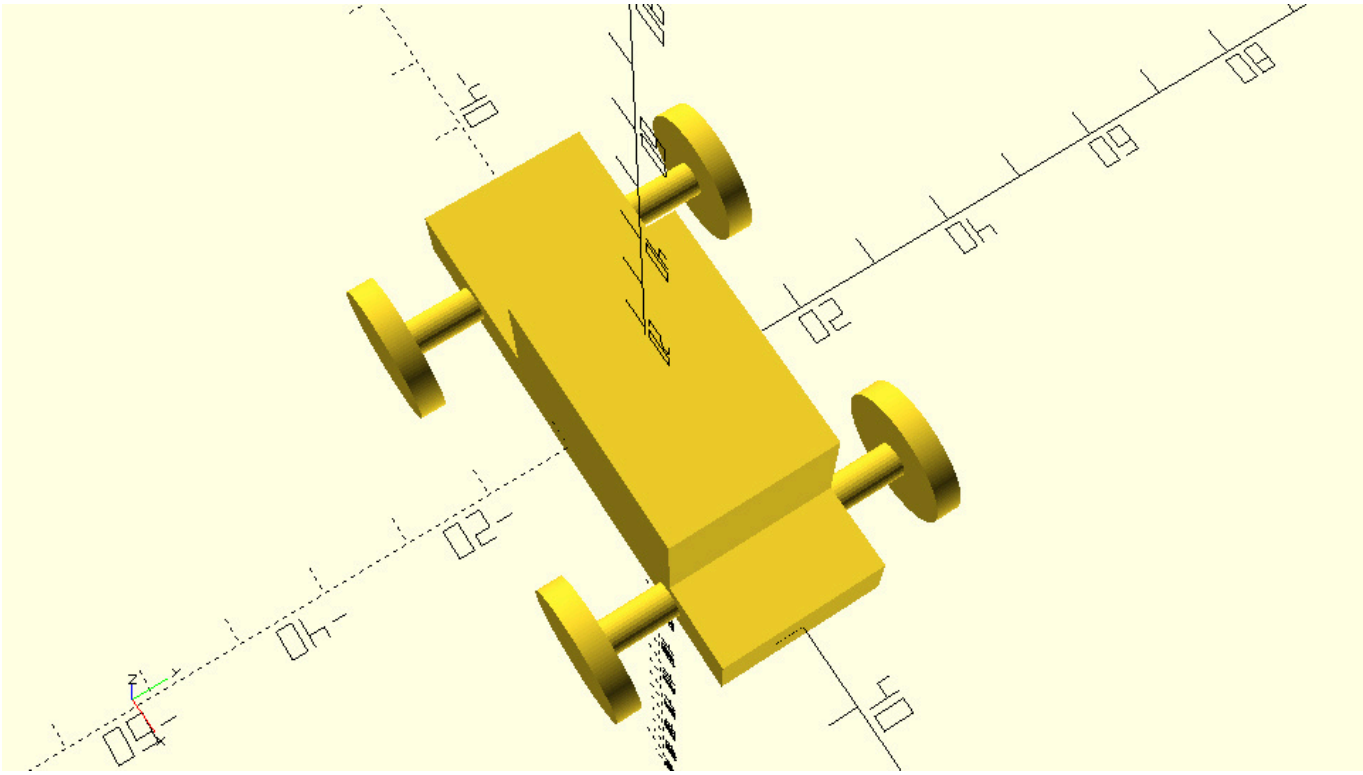
Exercise

Try parameterizing the track (separation between left and right wheels) using a new variable named track. Try assigning different values to the track variable. What do you notice? Does any other aspect of your model depend on the value of the track variable? If yes, use the track variable to parameterize it so your model doesn't break apart.

Code
[Expand]



Code
[Expand]



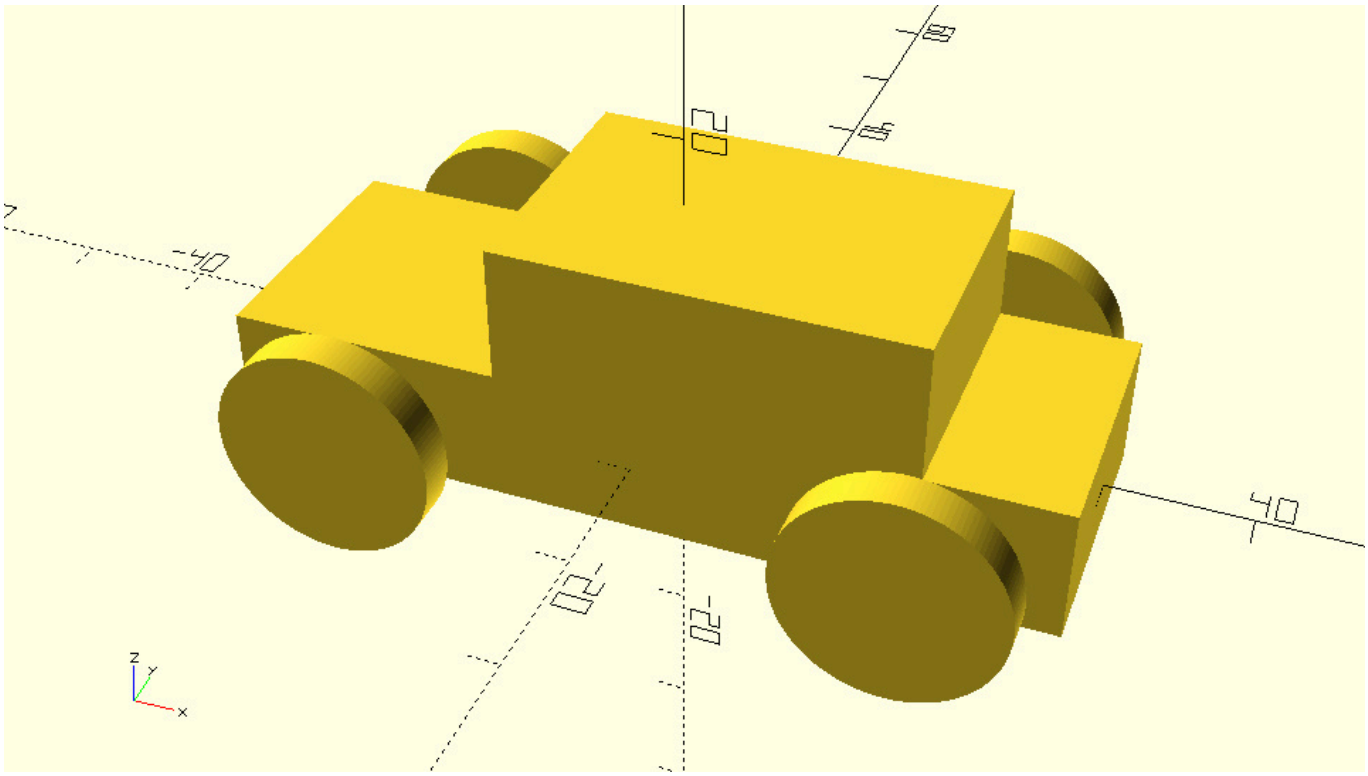
Challenge

The following script corresponds to the car model with parameterized wheel radius, base height, top height and track.

Code

car_from_parameterized_script.scad

```
$fa = 1;
$fs = 0.4;
wheel_radius = 8;
base_height = 10;
top_height = 10;
track = 30;
// Car body base
cube([60,20,base_height],center=true);
// Car body top
translate([5,0,base_height/2+top_height/2 - 0.001])
    cube([30,20,top_height],center=true);
// Front left wheel
translate([-20,-track/2,0])
    rotate([90,0,0])
    cylinder(h=3,r=wheel_radius,center=true);
// Front right wheel
translate([-20,track/2,0])
    rotate([90,0,0])
    cylinder(h=3,r=wheel_radius,center=true);
// Rear left wheel
translate([20,-track/2,0])
    rotate([90,0,0])
    cylinder(h=3,r=wheel_radius,center=true);
// Rear right wheel
translate([20,track/2,0])
    rotate([90,0,0])
    cylinder(h=3,r=wheel_radius,center=true);
// Front axle
translate([-20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
// Rear axle
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
```



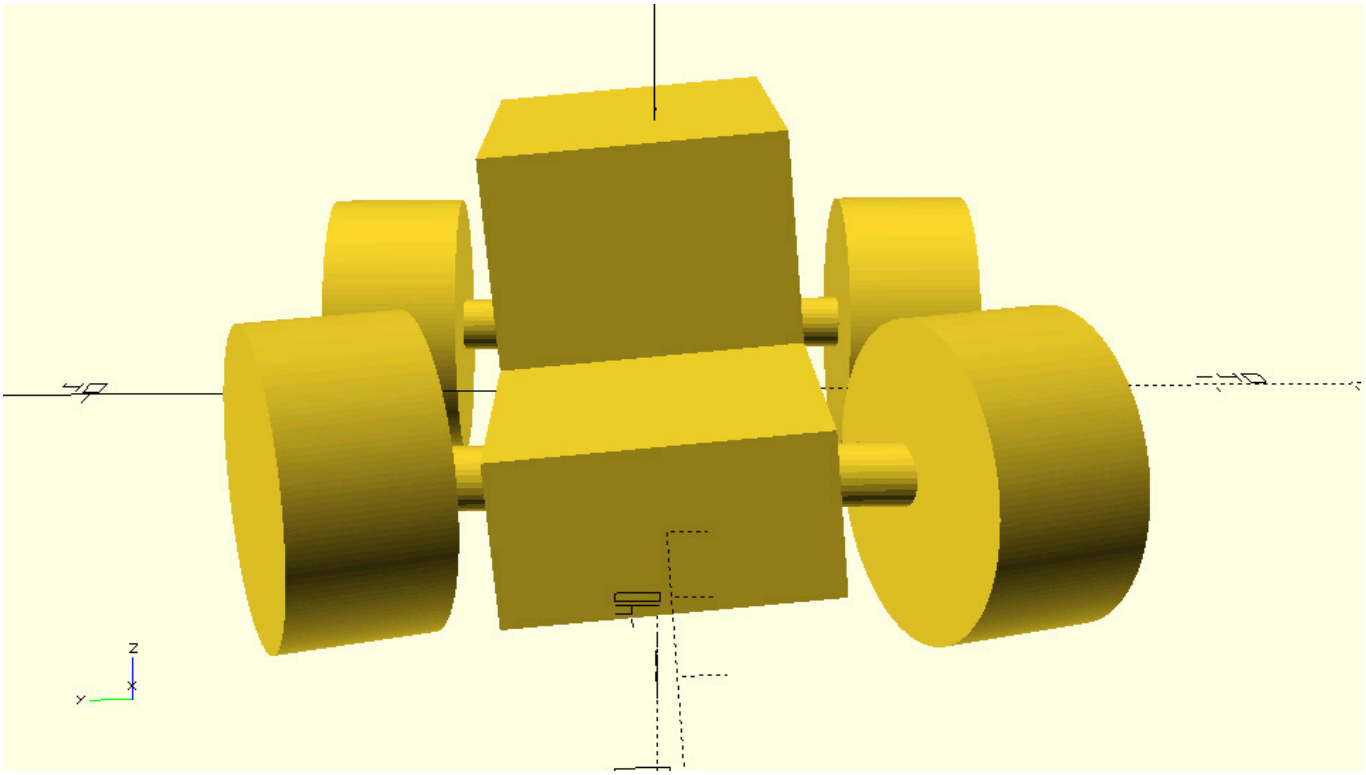
Exercise

Try using a wheel_width variable to parameterize the width of the wheels, a wheels_turn variable to parameterize the rotation of the front wheels around the Z axis and a body_roll variable to parameterize the rotation of the body around the X axis. Experiment with assigning different values to wheel_radius, base_height, top_height, track, wheel_width, wheels_turn and body_roll to create a version of the car that you like.

Code[Collapse]

turning_car_from_parameterized_script.scad

```
$fa = 1;
$fs = 0.4;
wheel_radius = 10;
base_height = 10;
top_height = 14;
track = 40;
wheel_width = 10;
body_roll = -5;
wheels_turn = 20;
rotate([body_roll,0,0]) {
    // Car body base
    cube([60,20,base_height],center=true);
    // Car body top
    translate([5,0,base_height/2+top_height/2 - 0.001])
        cube([30,20,top_height],center=true);
}
// Front left wheel
translate([-20,-track/2,0])
    rotate([90,0,wheels_turn])
    cylinder(h=wheel_width,r=wheel_radius,center=true);
// Front right wheel
translate([-20,track/2,0])
    rotate([90,0,wheels_turn])
    cylinder(h=wheel_width,r=wheel_radius,center=true);
// Rear left wheel
translate([20,-track/2,0])
    rotate([90,0,0])
    cylinder(h=wheel_width,r=wheel_radius,center=true);
// Rear right wheel
translate([20,track/2,0])
    rotate([90,0,0])
    cylinder(h=wheel_width,r=wheel_radius,center=true);
// Front axle
translate([-20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
// Rear axle
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
```



By now it should be clear to you that parameterizing your models unlocks the power of reusing, customizing and iterating your designs as well as that of effortlessly exploring different possibilities.

Parameterizing your own models

Have you put your new skills into use? Have you created any other models yourself?

Exercise

Try parameterizing a few aspects or more of the models that you have created. See how far you can go! Experiment with assigning various combinations of values to the variables that you have defined. See how different the versions of your designs can be.

Chapter 3

The sphere primitive and resizing objects

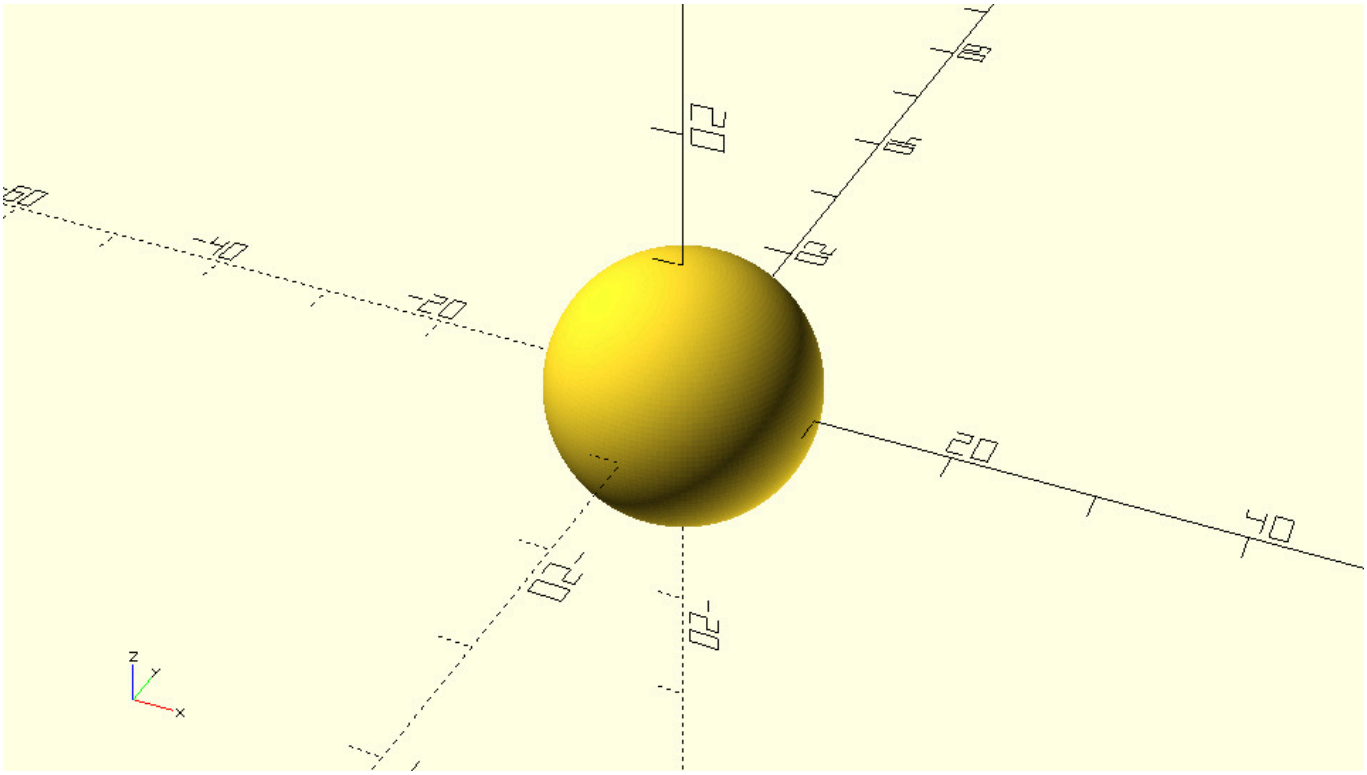
You showed the car to your friends and they were quite impressed with your new skills. One of them even challenged you to come up with different futuristic wheel designs. It's time to put your creativity to work and learn more OpenSCAD features!

So far you have been using the cube and cylinder primitives. Another 3D primitive that is available in OpenSCAD is the sphere. You can create a sphere using the following command.

Code

sphere.scad

```
sphere(r=10);
```



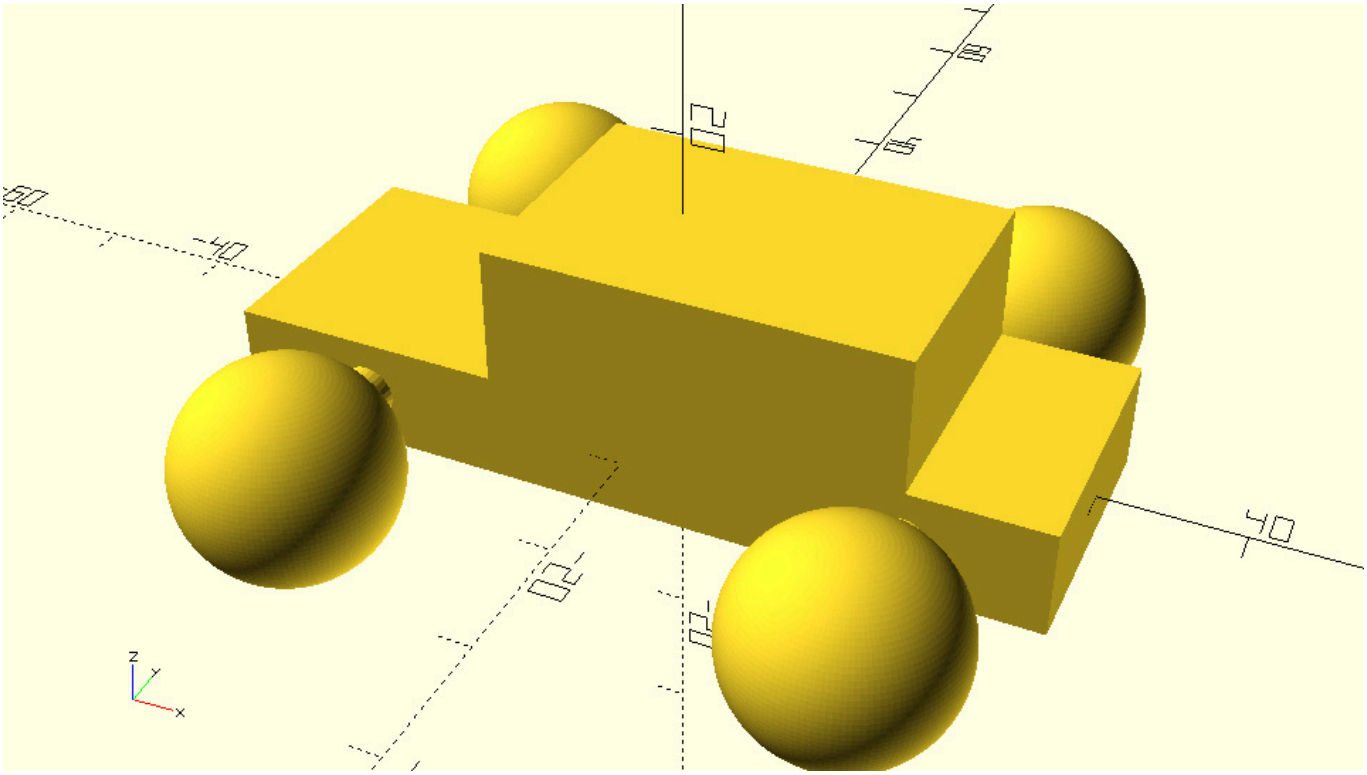
You should notice that the sphere is created centered on the origin. The input parameter r corresponds to the radius of the sphere.

One idea that came to your head was to replace the cylindrical wheels with spherical ones.

Exercise

Try making the wheels of your car spherical. To do so replace the appropriate cylinder commands with sphere commands. Is there still a need to rotate the wheels around the X axis? Is the wheel_width variable still required? Is there any visible change to your model when you modify the value of wheels_turn variable?

Code
[Expand]



The idea to use a sphere to create the wheels was nice. You can now squish the spheres to give them a more wheel like shape. One way to do so is using the scale command.

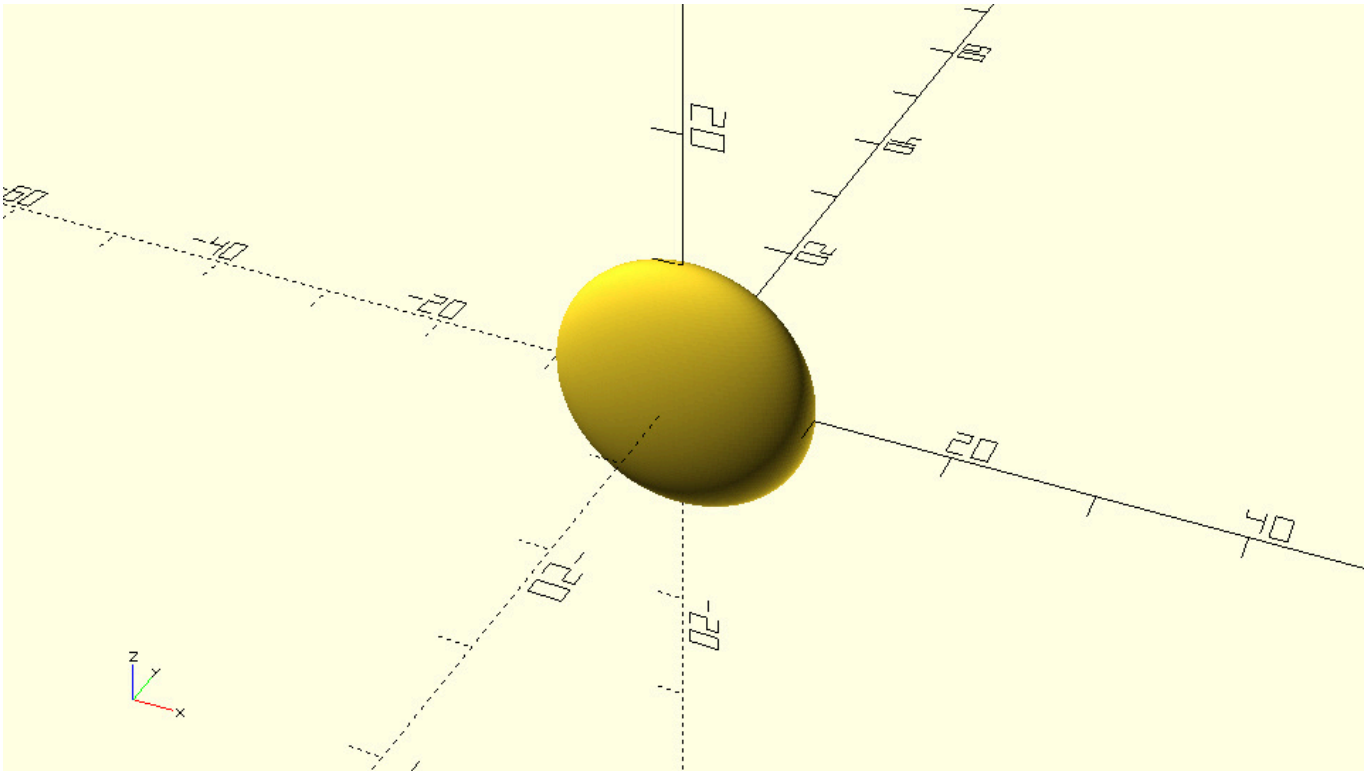
Exercise

Try creating a sphere with a radius of 10 units on a blank model. Use the scale command to scale the sphere by a factor of 0.4 only along the Y axis.

Code[Collapse]

narrowed_spherical_wheel_using_scale.scad

```
scale([1,0.4,1])
    sphere(r=10);
```

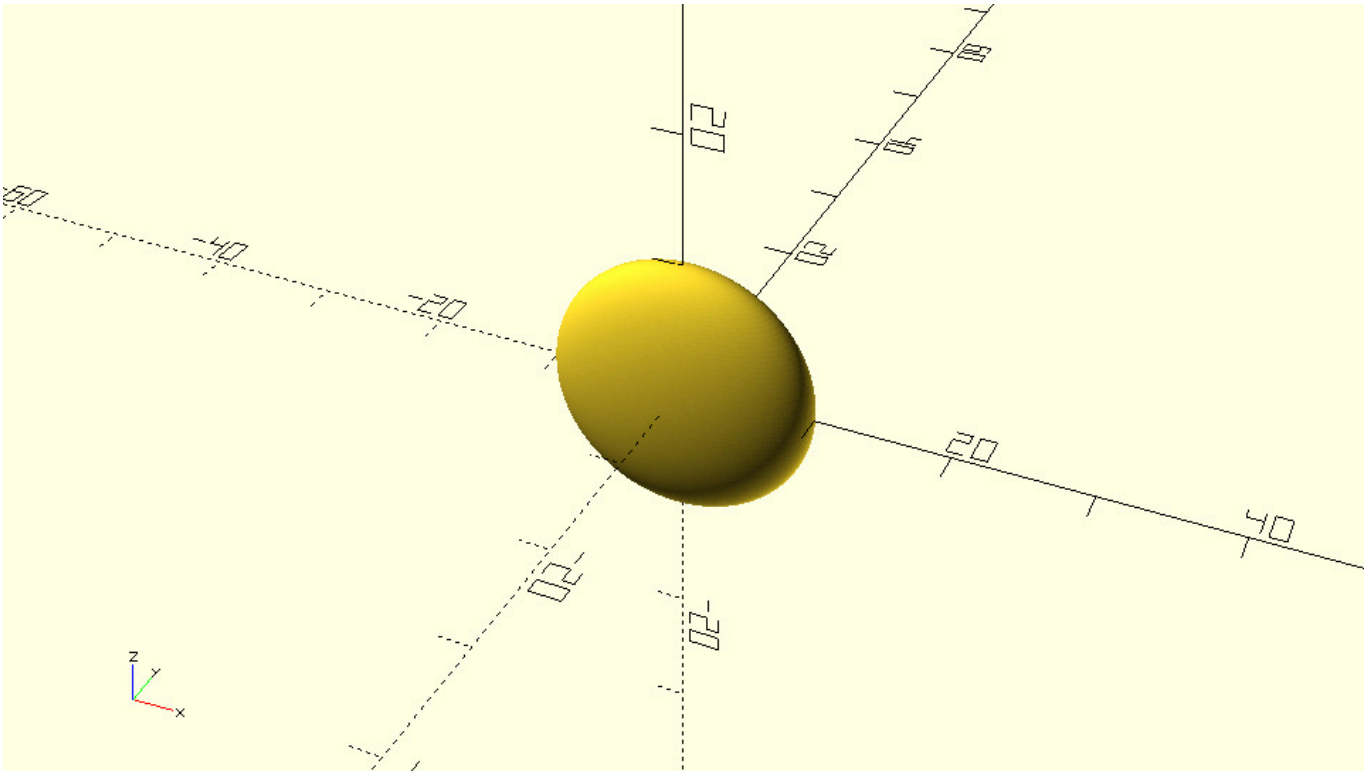


Another way to scale objects is by using the resize transformation. The difference between scale and resize is that when using the scale command, you have to specify the desired scaling factor along each axis but when using the resize command you have to specify the desired resulting dimensions of the object along each axis. In the previous example you started with a sphere that has a radius of 10 units (total dimension of 20 units along each axis) and scaled it by a factor of 0.4 along the Y axis. Thus, the resulting dimension of the scaled sphere along the Y axis is 8 units. The dimensions along the X and Z axis remain the same (20 units) since the scaling factors along these axes are equal to 1. You could achieve the same result using the following resize command.

Code

narrowed_spherical_wheel_using_resize.scad

```
resize([20,8,20])
    sphere(r=10);
```



When you are scaling/resizing an object and you are concerned about its resulting dimensions it is more convenient to use the resize command. In contrast when you are concerned more about the ratio of the resulting dimensions compared to the starting dimensions it is more convenient to use the scale command.

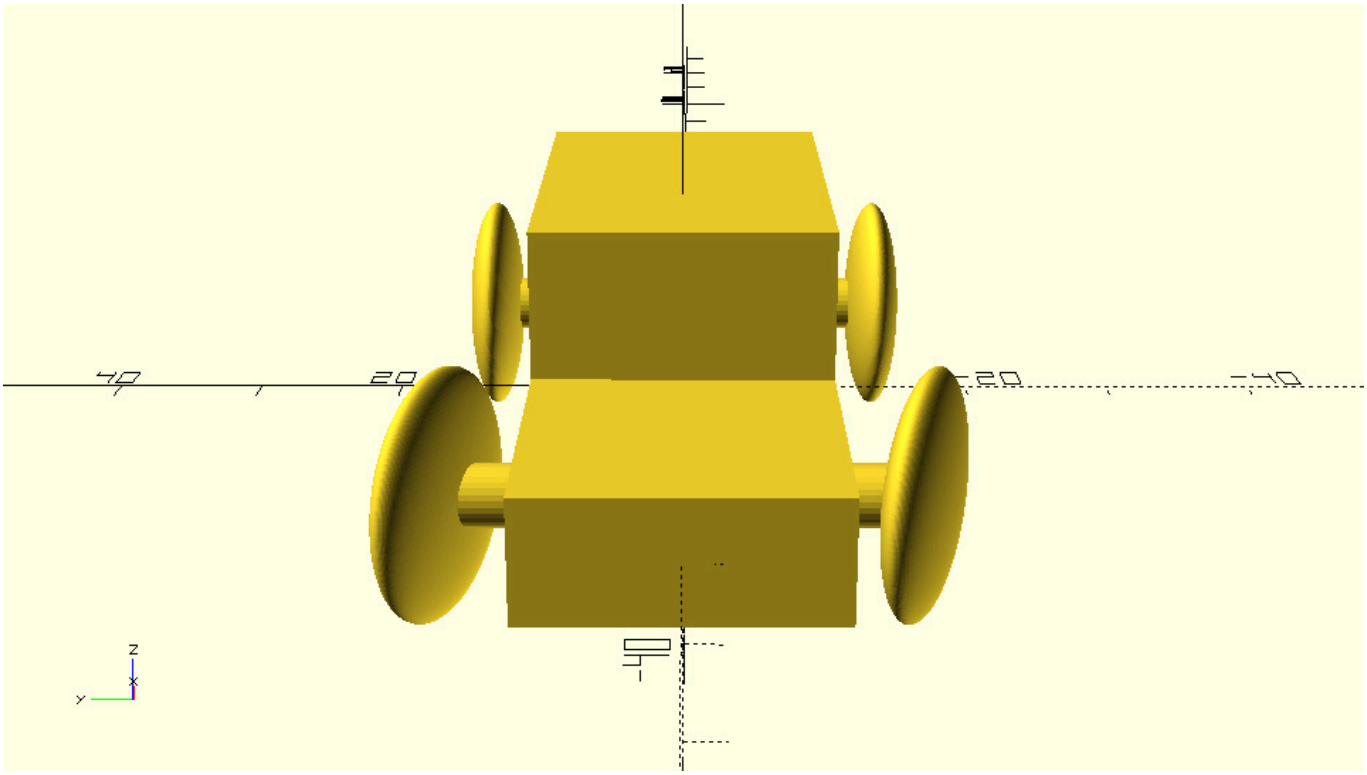
Exercise

Try squishing the spherical wheels of your car along the Y axis. Use the resize command and the wheel_width variable to have control over the resulting width of the wheels. Resize the wheels only along the Y axis.

Code[Collapse]

car_with_narrowed_spherical_wheels.scad

```
$fa = 1;
$fs = 0.4;
wheel_radius = 8;
base_height = 8;
top_height = 10;
track = 30;
wheel_width = 4;
body_roll = 0;
wheels_turn = -20;
rotate([body_roll,0,0]) {
    // Car body base
    cube([60,20,base_height],center=true);
    // Car body top
    translate([5,0,base_height/2+top_height/2 - 0.001])
        cube([30,20,top_height],center=true);
}
// Front left wheel
translate([-20,-track/2,0])
    rotate([0,0,wheels_turn])
    resize([2*wheel_radius,wheel_width,2*wheel_radius])
    sphere(r=wheel_radius);
// Front right wheel
translate([-20,track/2,0])
    rotate([0,0,wheels_turn])
    resize([2*wheel_radius,wheel_width,2*wheel_radius])
    sphere(r=wheel_radius);
// Rear left wheel
translate([20,-track/2,0])
    rotate([0,0,0])
    resize([2*wheel_radius,wheel_width,2*wheel_radius])
    sphere(r=wheel_radius);
// Rear right wheel
translate([20,track/2,0])
    rotate([0,0,0])
    resize([2*wheel_radius,wheel_width,2*wheel_radius])
    sphere(r=wheel_radius);
// Front axle
translate([-20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
// Rear axle
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
```



The new wheel design looks cool. You can now create a body that better suits this new style.

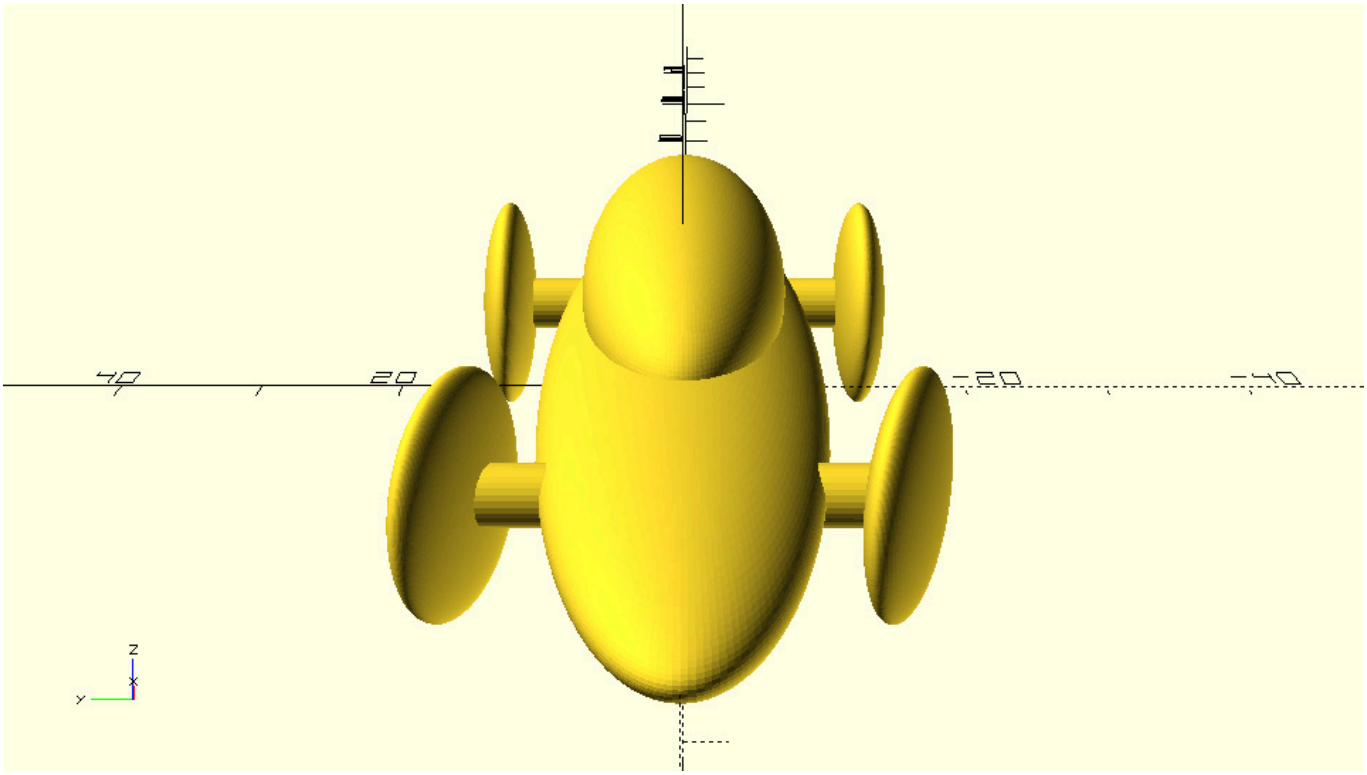
Exercise

Try using the sphere and resize/scale commands in place of the cube commands to create a body that matches the style of the wheels.

Code[Collapse]

car_with_narrowed_spherical_wheels_and_body.scad

```
$fa = 1;
$fs = 0.4;
wheel_radius = 8;
base_height = 8;
top_height = 10;
track = 28;
wheel_width = 4;
body_roll = 0;
wheels_turn = -20;
rotate([body_roll,0,0]) {
    // Car body base
    resize([90,20,12])
        sphere(r=10);
    // Car body top
    translate([10,0,5])
        resize([50,15,15])
        sphere(r=10);
}
// Front left wheel
translate([-20,-track/2,0])
    rotate([0,0,wheels_turn])
    resize([2*wheel_radius,wheel_width,2*wheel_radius])
    sphere(r=wheel_radius);
// Front right wheel
translate([-20,track/2,0])
    rotate([0,0,wheels_turn])
    resize([2*wheel_radius,wheel_width,2*wheel_radius])
    sphere(r=wheel_radius);
// Rear left wheel
translate([20,-track/2,0])
    rotate([0,0,0])
    resize([2*wheel_radius,wheel_width,2*wheel_radius])
    sphere(r=wheel_radius);
// Rear right wheel
translate([20,track/2,0])
    rotate([0,0,0])
    resize([2*wheel_radius,wheel_width,2*wheel_radius])
    sphere(r=wheel_radius);
// Front axle
translate([-20,0,0])rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
// Rear axle
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
```



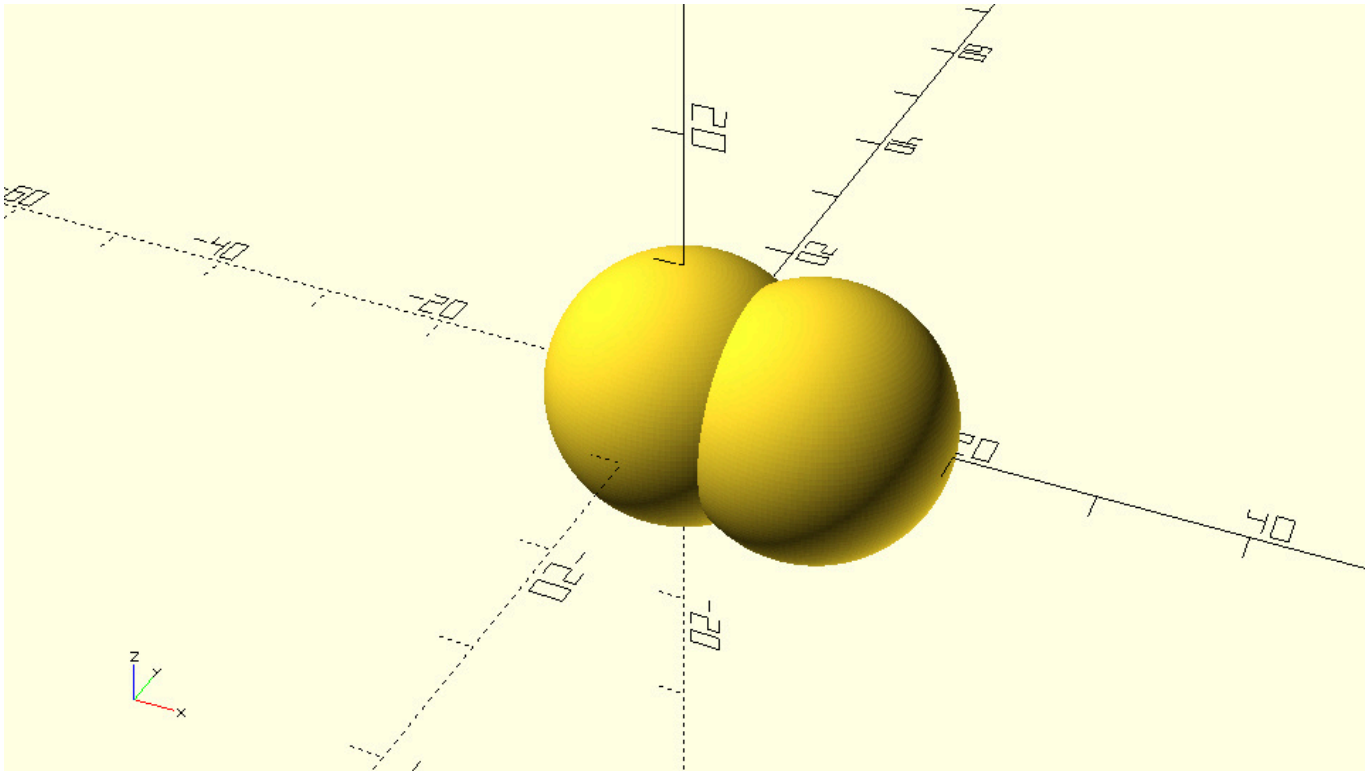
Combining objects in other ways

So far when you wanted to create an additional object in your model, you just added another statement in your script. The final car model is the union of all objects that have been defined in your script. You have been implicitly using the union command which is one of the available boolean operations. When using the union boolean operation, OpenSCAD takes the union of all objects as the resulting model. In the following script the union is used implicitly.

Code

union_of_two_spheres_implicit.scad

```
sphere(r=10);
translate([10,0,0])
    sphere(r=10);
```

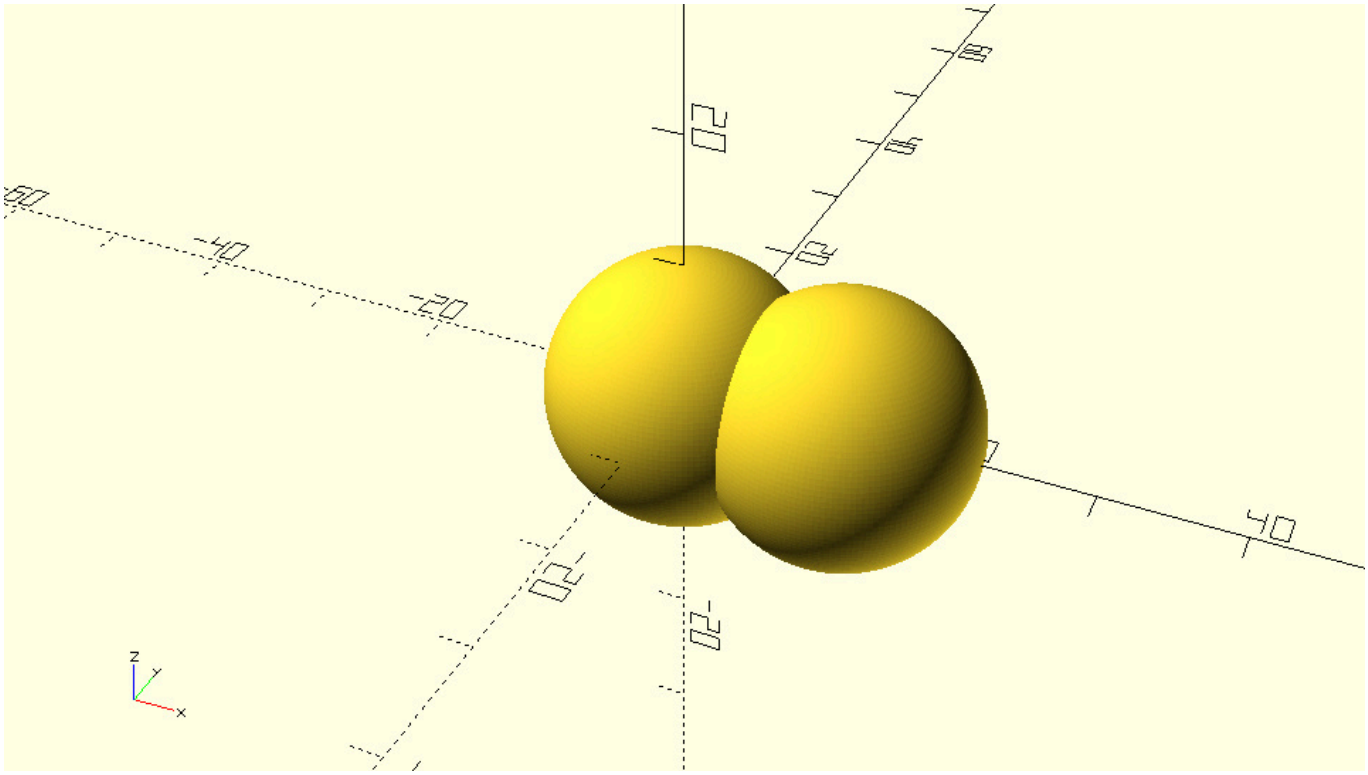


You can make the use of union explicit by including the union command in your script.

Code

union_of_two_spheres_explicit.scad

```
union() {
    sphere(r=10);
    translate([12,0,0])
        sphere(r=10);
}
```



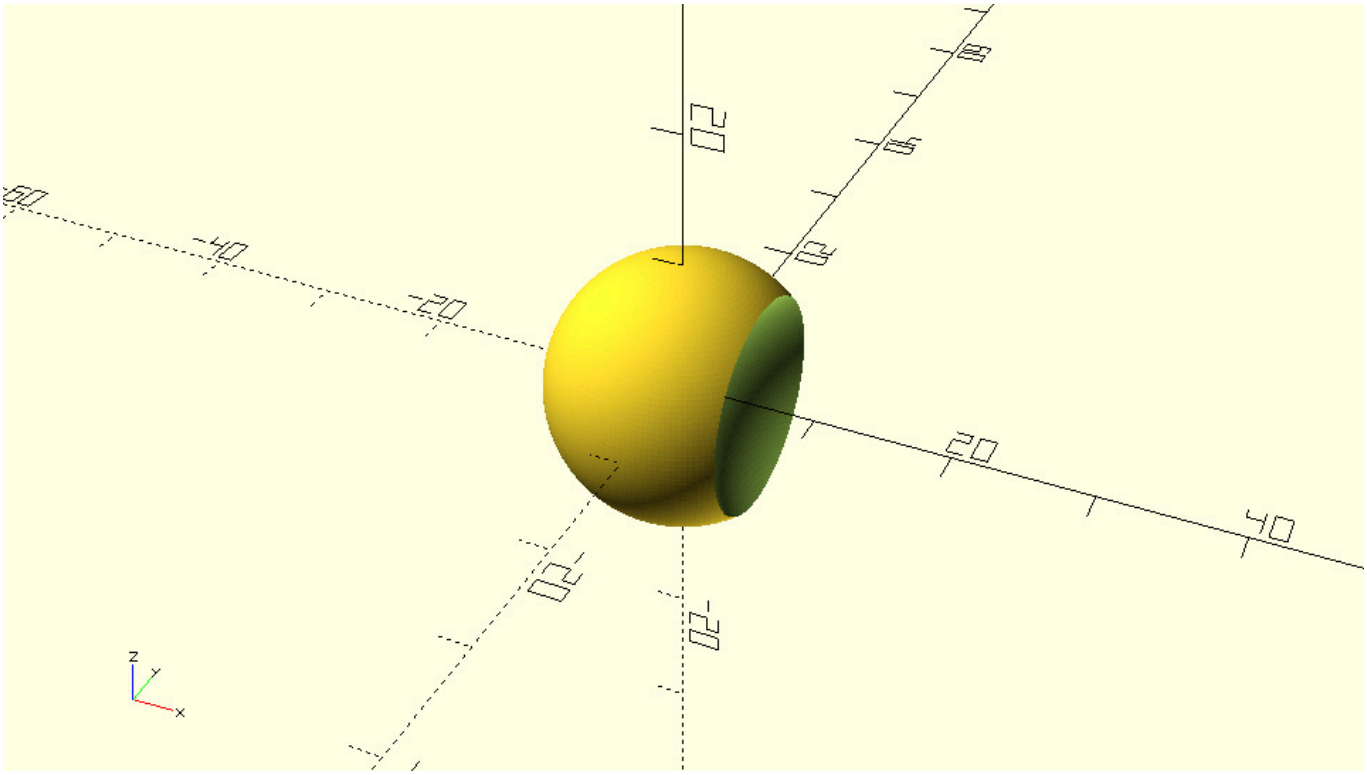
You should notice that the union command doesn't have any input parameters. This is true for all boolean operations. The union is applied to all objects inside the curly brackets. You should also notice that the statements inside the curly brackets have a semicolon at the end. In contrast there is no semicolon after the closing curly bracket. This syntax is similar to the use of transformations when applied to multiple objects.

In total there are three boolean operations. The second one is the difference. The difference command subtracts the second and all further objects that have been defined inside the curly brackets from the first one. The previous example results in the following model when using the difference operation instead of the union.

Code

difference_of_two_spheres.scad

```
difference() {
    sphere(r=10);
    translate([12,0,0])
        sphere(r=10);
}
```

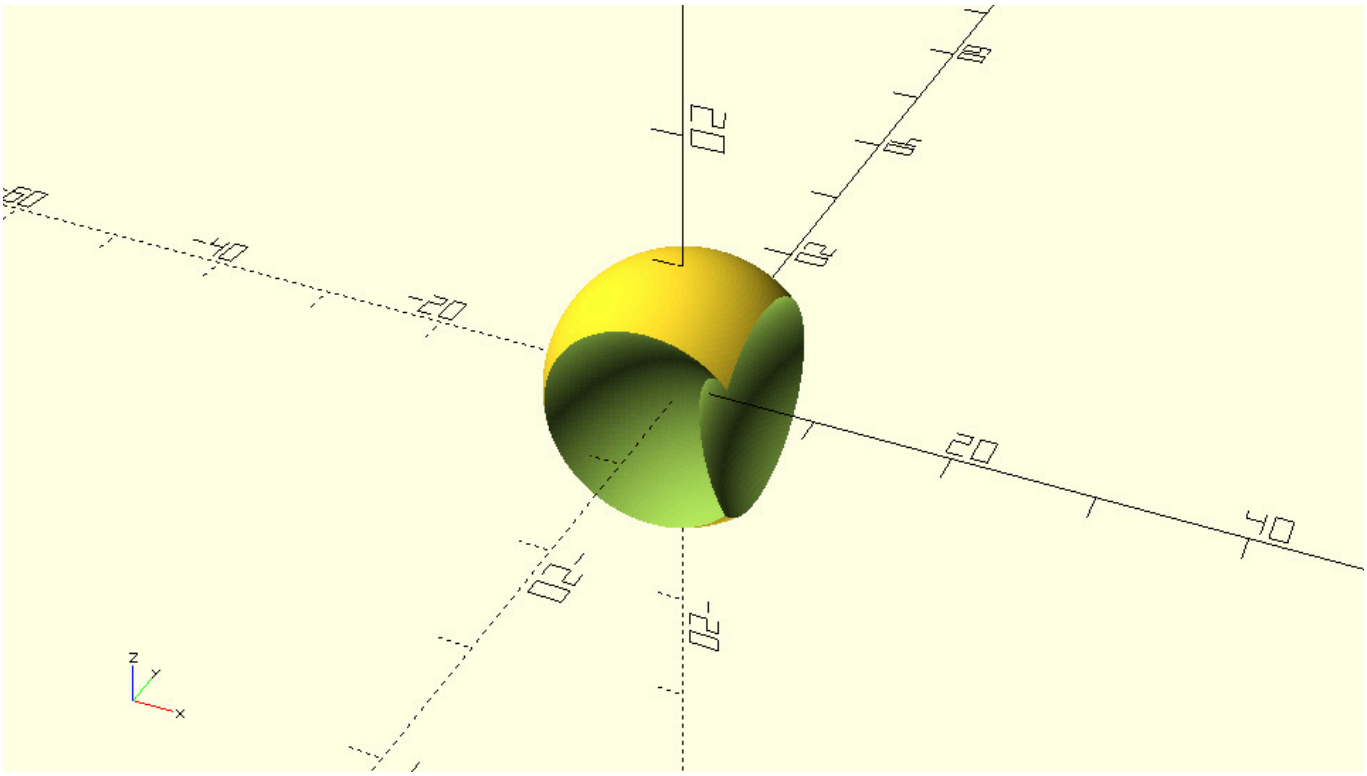


Further defined objects (third, fourth etc.) are also subtracted. The following example has three objects.

Code

difference_of_three_spheres.scad

```
difference() {
    sphere(r=10);
    translate([12,0,0])
        sphere(r=10);
    translate([0,-12,0])
        sphere(r=10);
}
```

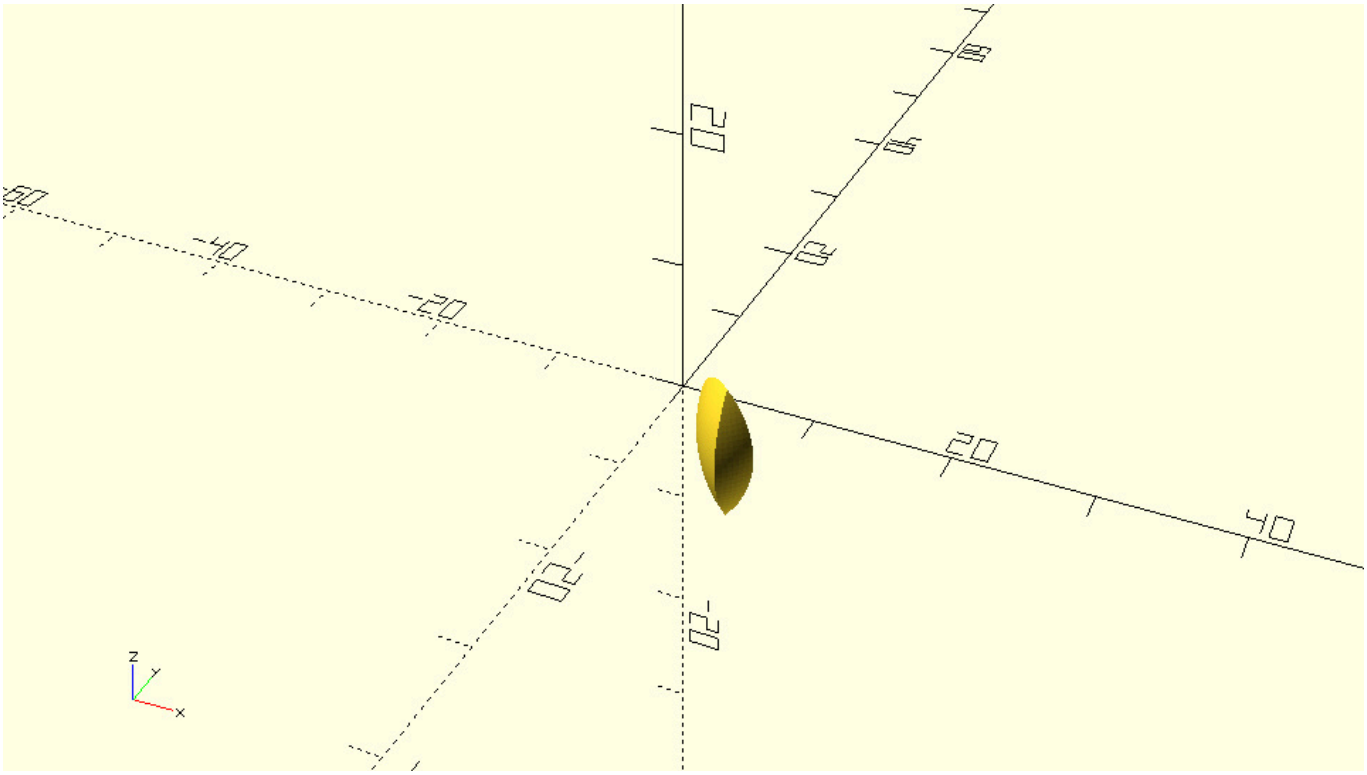


The third boolean operation is the intersection. The intersection operation keeps only the overlapping portion of all objects. The previous example results in the following model when the intersection operation is used.

Code

intersection_of_three_spheres.scad

```
intersection() {
    sphere(r=10);
    translate([12,0,0])
        sphere(r=10);
    translate([0,-12,0])
        sphere(r=10);
}
```



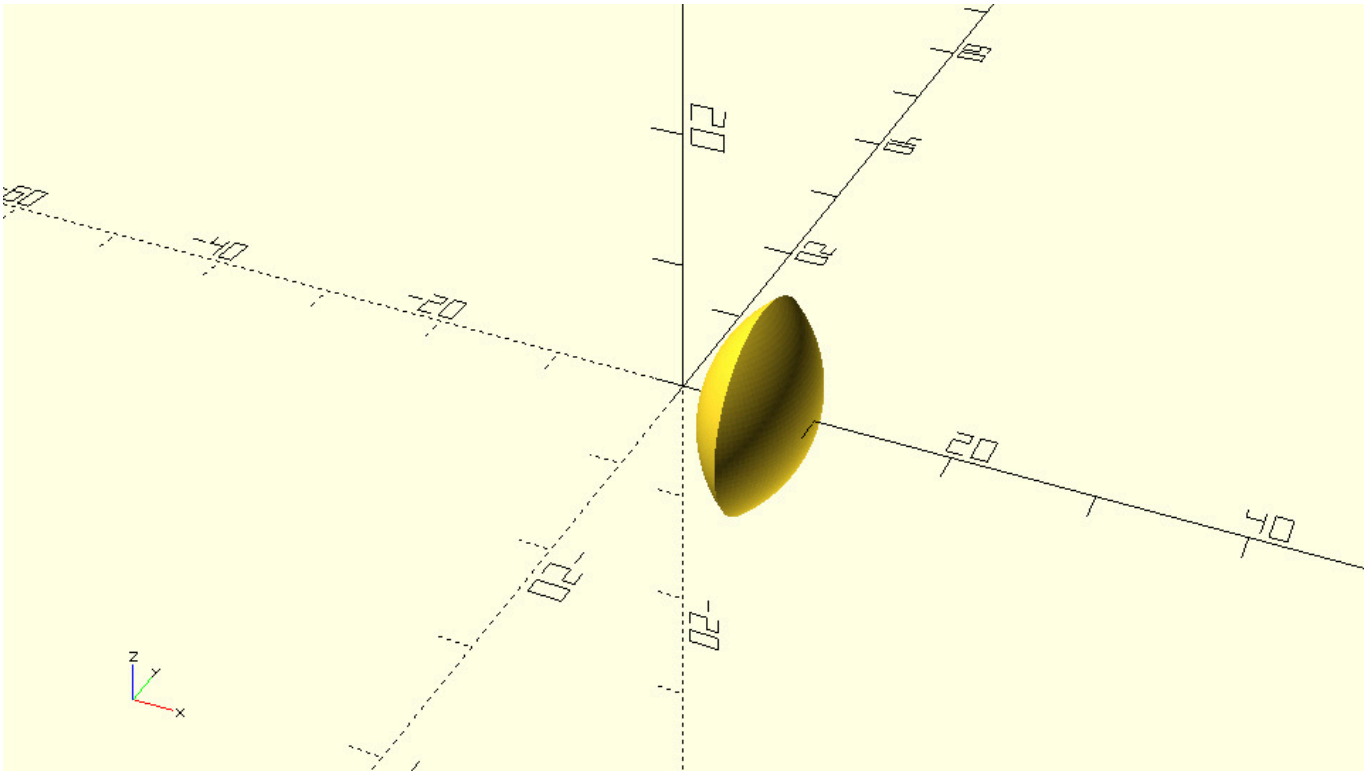
The resulting model is the common area of all three objects.

When only the first two spheres are defined inside the curly brackets, the intersection is the following.

Code

intersection_of_two_spheres.scad

```
intersection() {
    sphere(r=10);
    translate([12,0,0])
        sphere(r=10);
}
```



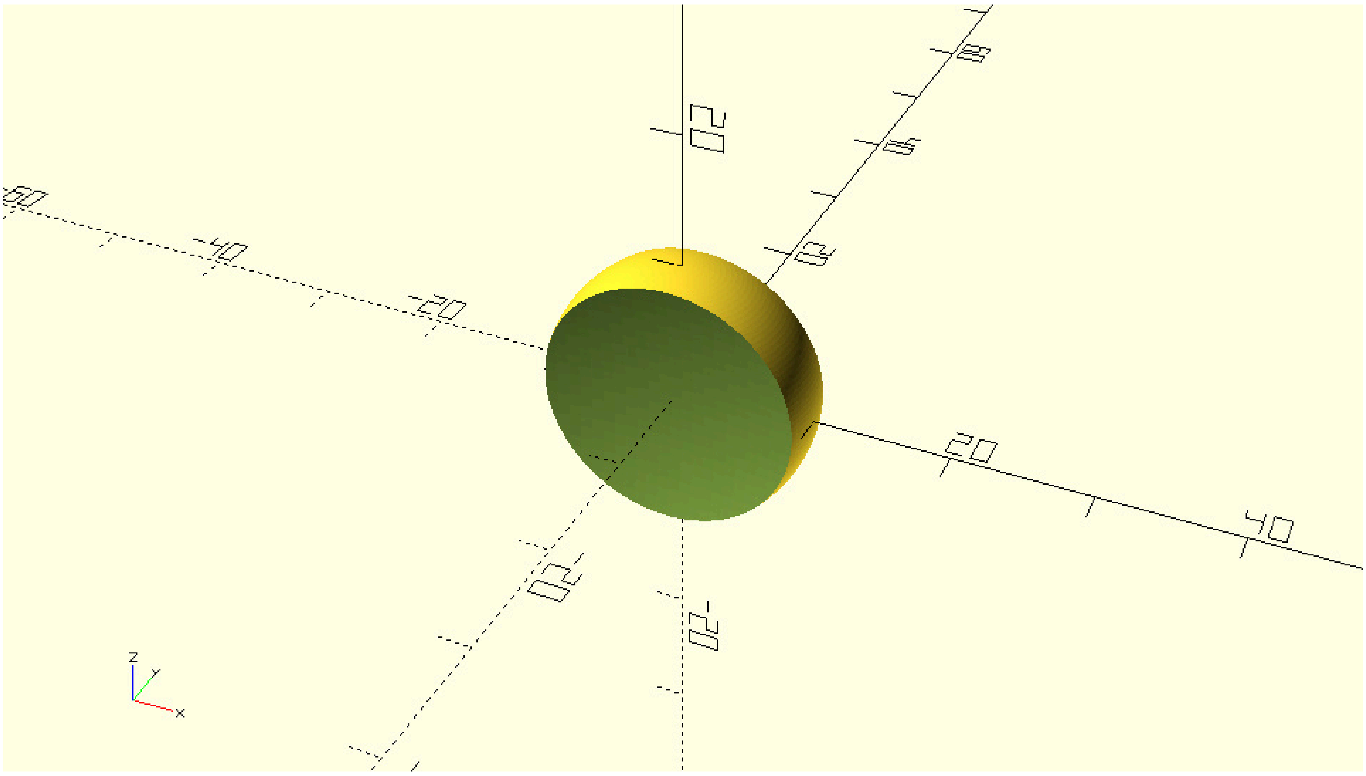
Exercise

Try using the difference operation to create a new wheel design. To do so first create a sphere and then subtract a portion of a sphere from both sides. The radius of the first sphere should be equal to the desired wheel radius (wheel_radius variable). The radius of the other two spheres should be equal to a side_spheres_radius variable. Given a hub_thickness variable what is the amount of units that the side spheres should be translated along the positive and negative direction of Y axis so that the thickness of the remaining material at the center of the first sphere is equal to the value of hub_thickness?

Code[Collapse]

wheel_with_spherical_sides.scad

```
$fa = 1;
$fs = 0.4;
wheel_radius=10;
side_spheres_radius=50;
hub_thickness=4;
difference() {
    sphere(r=wheel_radius);
    translate([0,side_spheres_radius + hub_thickness/2,0])
        sphere(r=side_spheres_radius);
    translate([0,- (side_spheres_radius + hub_thickness/2),0])
        sphere(r=side_spheres_radius);
}
```



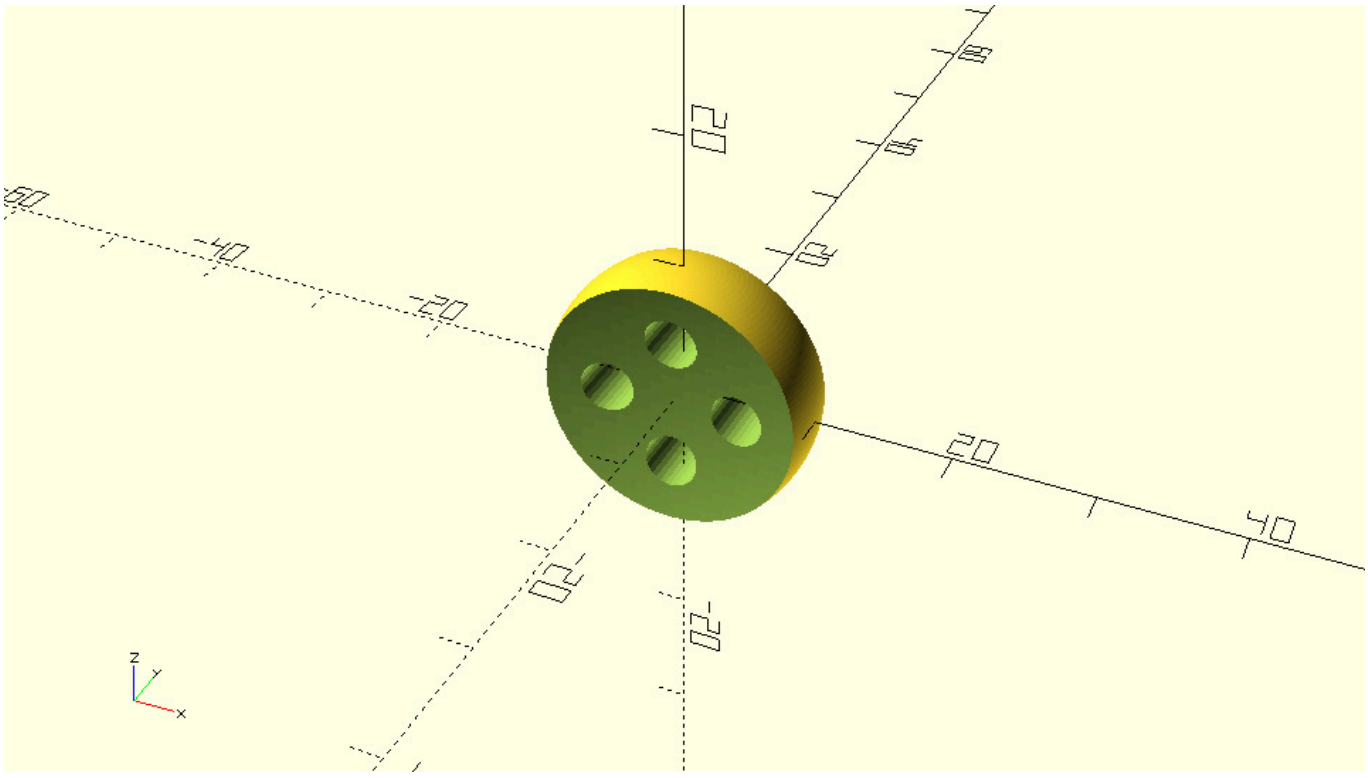
Exercise

Try removing some material from the wheels by subtracting four cylinders that are perpendicular to the wheel. The cylinders should be placed at half the wheel radius and be equally spaced. Introduce a cylinder_radius and a cylinder_height variable. The value of cylinder_height should be appropriate so that the cylinders are always longer than the thickness of the material they are removed from.

Code[Collapse]

wheel_with_spherical_sides_and_holes.scad

```
$fa = 1;
$fs = 0.4;
wheel_radius=10;
side_spheres_radius=50;
hub_thickness=4;
cylinder_radius=2;
cylinder_height=2*wheel_radius;
difference() {
    // Wheel sphere
    sphere(r=wheel_radius);
    // Side sphere 1
    translate([0,side_spheres_radius + hub_thickness/2,0])
        sphere(r=side_spheres_radius);
    // Side sphere 2
    translate([0,- (side_spheres_radius + hub_thickness/2),0])
        sphere(r=side_spheres_radius);
    // Cylinder 1
    translate([wheel_radius/2,0,0])
        rotate([90,0,0])
        cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    // Cylinder 2
    translate([0,0,wheel_radius/2])
        rotate([90,0,0])
        cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    // Cylinder 3
    translate([-wheel_radius/2,0,0])
        rotate([90,0,0])
        cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    // Cylinder 4
    translate([0,0,-wheel_radius/2])
        rotate([90,0,0])
        cylinder(h=cylinder_height,r=cylinder_radius,center=true);
}
```



Exercise

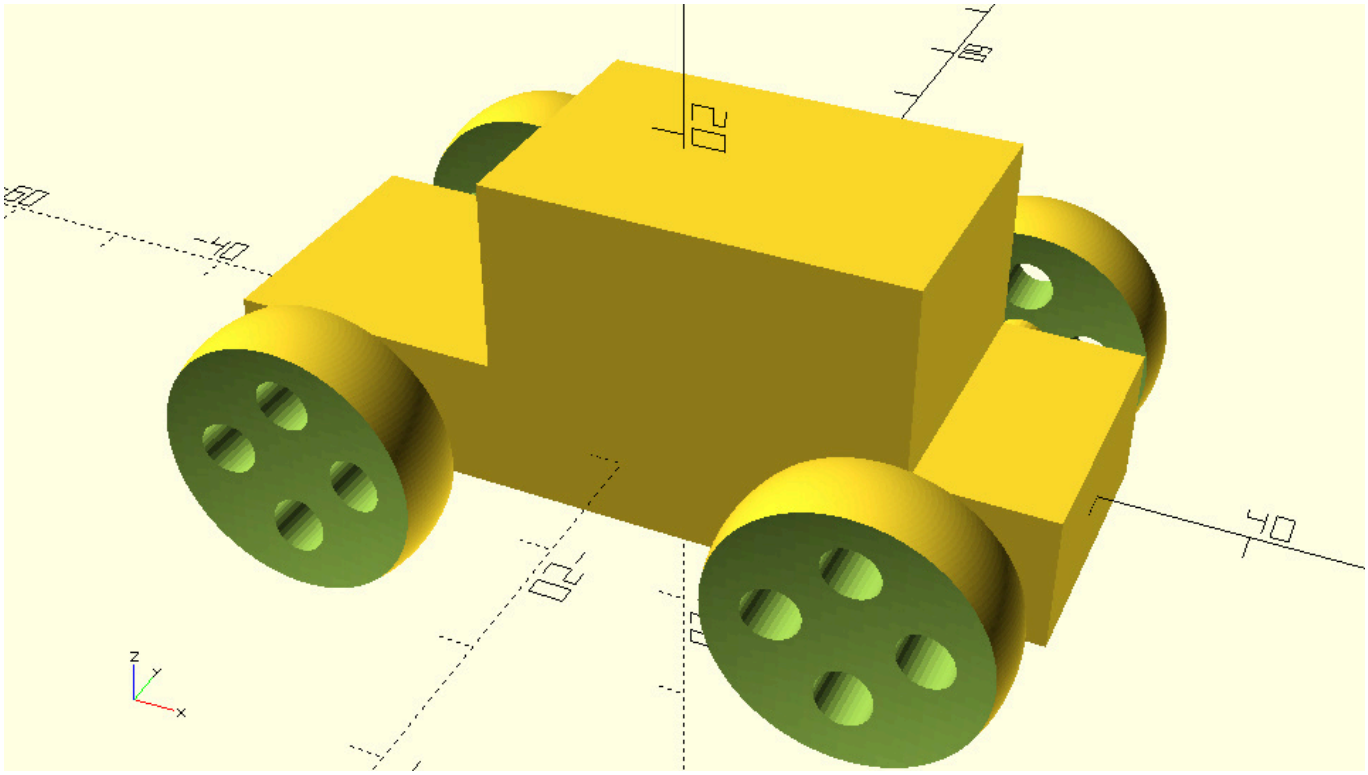
Try using the above wheels in one version of the car.

Code[Collapse]

car_with_wheels_with_spherical_sides_and_holes.scad

```
$fa = 1;
$fs = 0.4;
wheel_radius = 10;
base_height = 10;
top_height = 14;
track = 35;
wheel_width = 10;
body_roll = 0;
wheels_turn = 0;
side_spheres_radius=50;
hub_thickness=4;
cylinder_radius=2;
cylinder_height=2*wheel_radius;
rotate([body_roll,0,0]) {
    // Car body base
    cube([60,20,base_height],center=true);
    // Car body top
    translate([5,0,base_height/2+top_height/2 - 0.001])
        cube([30,20,top_height],center=true);
}
// Front left wheel
translate([-20,-track/2,0])
    rotate([0,0,wheels_turn])
    difference() {
        // Wheel sphere
        sphere(r=wheel_radius);
        // Side sphere 1
        translate([0,side_spheres_radius + hub_thickness/2,0])
            sphere(r=side_spheres_radius);
        // Side sphere 2
        translate([0,- (side_spheres_radius + hub_thickness/2),0])
            sphere(r=side_spheres_radius);
        // Cylinder 1
        translate([wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 2
        translate([0,0,wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 3
        translate([-wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 4
        translate([0,0,-wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
}
// Front right wheel
translate([-20,track/2,0])
    rotate([0,0,wheels_turn])
    difference() {
        // Wheel sphere
        sphere(r=wheel_radius);
        // Side sphere 1
        translate([0,side_spheres_radius + hub_thickness/2,0])
            sphere(r=side_spheres_radius);
        // Side sphere 2
        translate([0,- (side_spheres_radius + hub_thickness/2),0])
            sphere(r=side_spheres_radius);
        // Cylinder 1
        translate([wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 2
        translate([0,0,wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 3
        translate([-wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 4
```

```
        translate([0,0,-wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
}
// Rear left wheel
translate([20,-track/2,0])
    rotate([0,0,0])
    difference() {
        // Wheel sphere
        sphere(r=wheel_radius);
        // Side sphere 1
        translate([0,side_spheres_radius + hub_thickness/2,0])
            sphere(r=side_spheres_radius);
        // Side sphere 2
        translate([0,- (side_spheres_radius + hub_thickness/2),0])
            sphere(r=side_spheres_radius);
        // Cylinder 1
        translate([wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 2
        translate([0,0,wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 3
        translate([-wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 4
        translate([0,0,-wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
}
// Rear right wheel
translate([20,track/2,0])
    rotate([0,0,0])
    difference() {
        // Wheel sphere
        sphere(r=wheel_radius);
        // Side sphere 1
        translate([0,side_spheres_radius + hub_thickness/2,0])
            sphere(r=side_spheres_radius);
        // Side sphere 2
        translate([0,- (side_spheres_radius + hub_thickness/2),0])
            sphere(r=side_spheres_radius);
        // Cylinder 1
        translate([wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 2
        translate([0,0,wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 3
        translate([-wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 4
        translate([0,0,-wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
}
// Front axle
translate([-20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
// Rear axle
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
```


20
40
20
40
z
y
x

Chapter 4

Defining and using modules

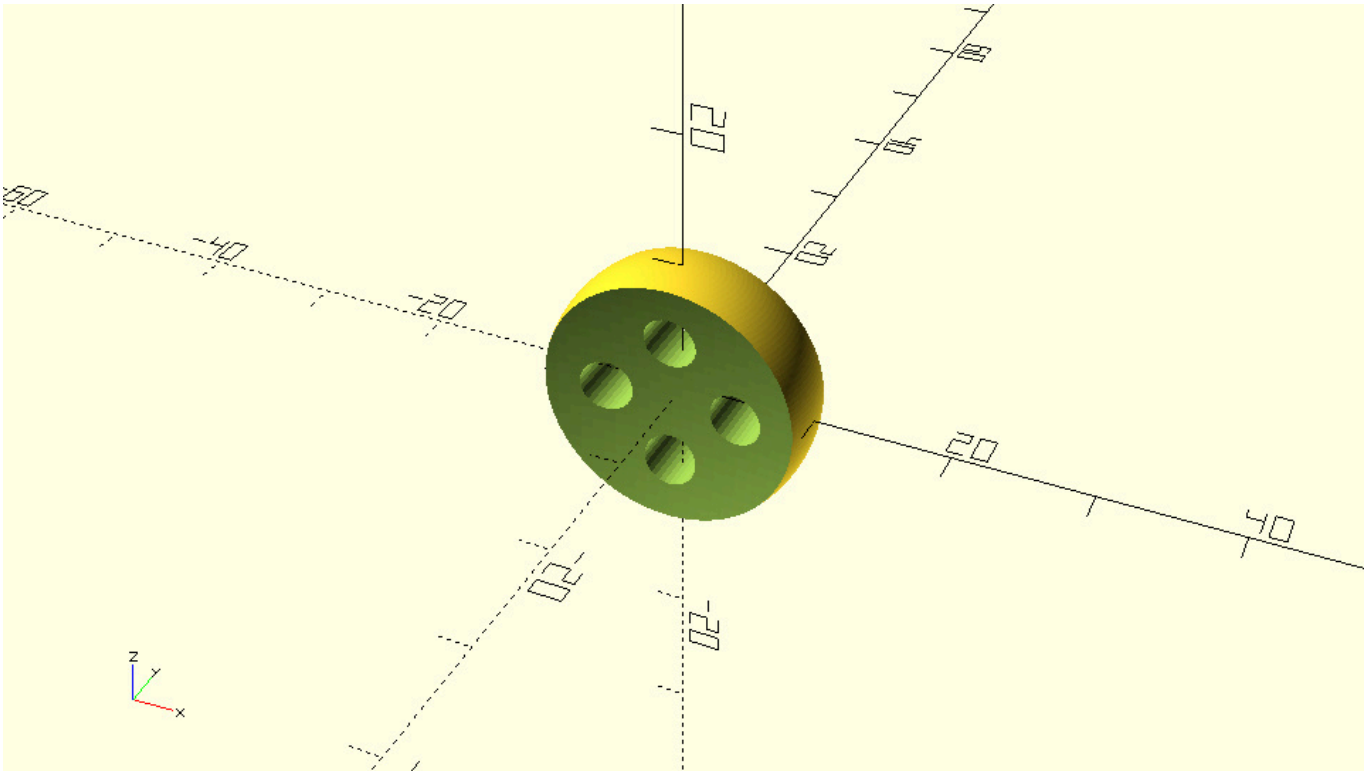
The script of the last example of the previous chapter got quite long. This was the result of replacing the simple cylindrical wheels (which require one statement to be created) with a more complex wheel design (which requires many statements to be created). To change the wheels from the simple to the complex design you have to identify all cylinder commands that define the simple wheels and replace them with commands that define the complex wheels. This process sounds similar to the steps you had to go through to change the diameter of the wheels. When no use of variables was made you had to identify the corresponding values in your script and replace them one by one with the new value. This repetitive and time-consuming process was improved with the use of a wheel_radius variable which enabled you to quickly and easily change the diameter of the wheel. Can you do anything though to improve the corresponding error-prone process when you want to change completely the design of the wheels? The answer is yes! You can use modules which is the analogue of variables applied to whole parts/models. You can define a part of your design or even your whole model as a module.

First remember for a moment the design of the complex wheel.

Code

wheel_with_spherical_sides_and_holes.scad

```
$fa = 1;
$fs = 0.4;
wheel_radius=10;
side_spheres_radius=50;
hub_thickness=4;
cylinder_radius=2;
cylinder_height=2*wheel_radius;
difference() {
    // Wheel sphere
    sphere(r=wheel_radius);
    // Side sphere 1
    translate([0,side_spheres_radius + hub_thickness/2,0])
        sphere(r=side_spheres_radius);
    // Side sphere 2
    translate([0,- (side_spheres_radius + hub_thickness/2),0])
        sphere(r=side_spheres_radius);
    // Cylinder 1
    translate([wheel_radius/2,0,0])
        rotate([90,0,0])
        cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    // Cylinder 2
    translate([0,0,wheel_radius/2])
        rotate([90,0,0])
        cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    // Cylinder 3
    translate([-wheel_radius/2,0,0])
        rotate([90,0,0])
        cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    // Cylinder 4
    translate([0,0,-wheel_radius/2])
        rotate([90,0,0])
        cylinder(h=cylinder_height,r=cylinder_radius,center=true);
}
```

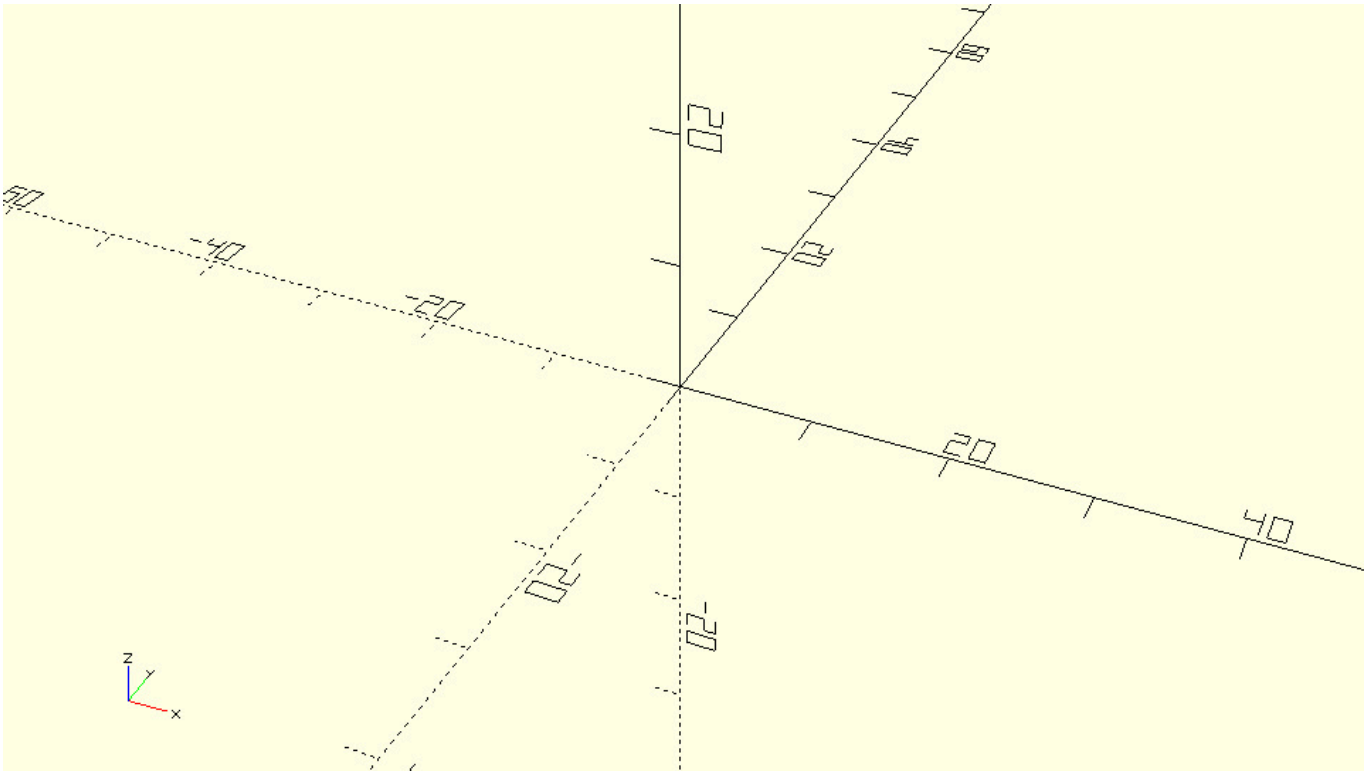


You can define the above wheel as a module in the following way.

Code

blank_model.scad

```
$fa = 1;
$fs = 0.4;
module wheel() {
    wheel_radius=10;
    side_spheres_radius=50;
    hub_thickness=4;
    cylinder_radius=2;
    cylinder_height=2*wheel_radius;
    difference() {
        // Wheel sphere
        sphere(r=wheel_radius);
        // Side sphere 1
        translate([0,side_spheres_radius + hub_thickness/2,0])
            sphere(r=side_spheres_radius);
        // Side sphere 2
        translate([0,- (side_spheres_radius + hub_thickness/2),0])
            sphere(r=side_spheres_radius);
        // Cylinder 1
        translate([wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 2
        translate([0,0,wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 3
        translate([-wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 4
        translate([0,0,-wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    }
}
```



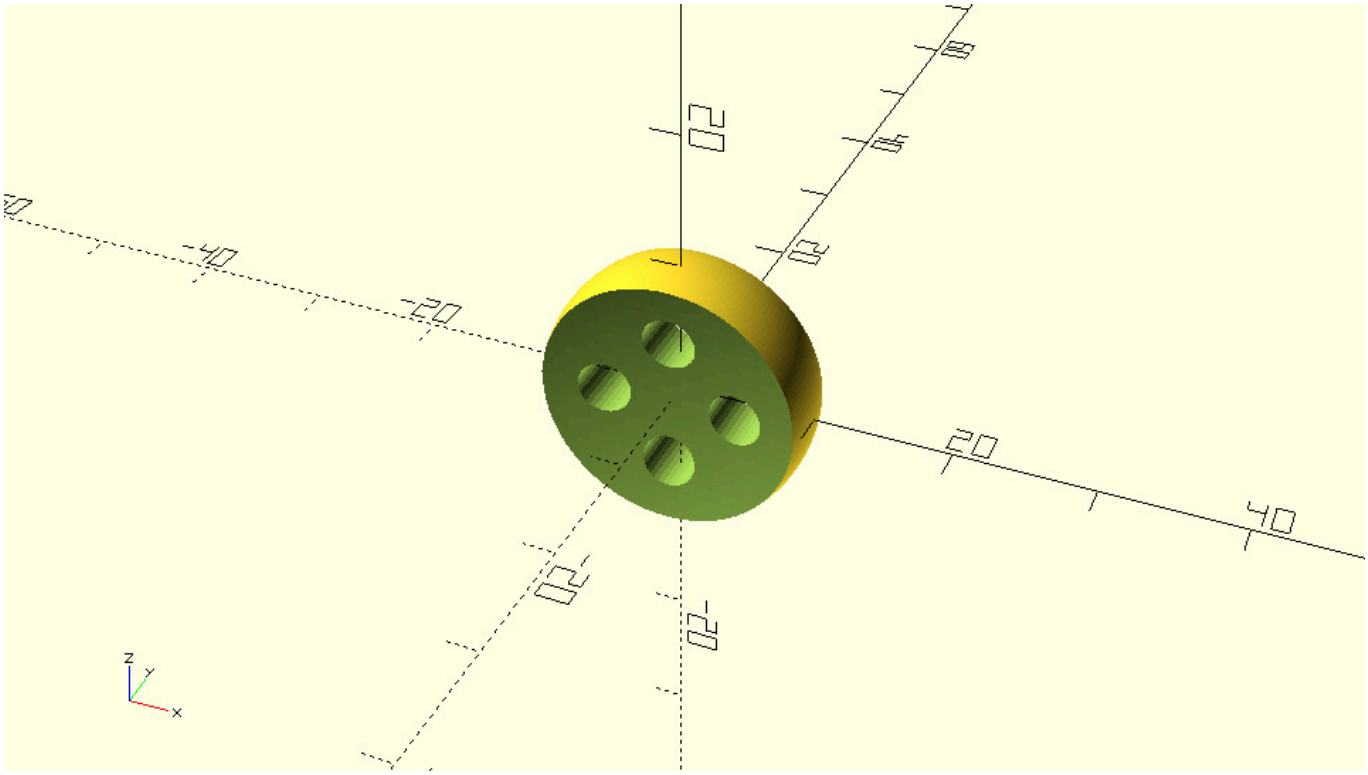
There are a few things that you need to get right. The first thing you should notice is that in order to define a module you have to type the word module followed by a name which you want to give to this module. In this case the module is named wheel. After the name of the module follows a pair of parentheses. Currently there is nothing inside the parentheses because no parameters have been defined for this module. Finally, after the pair of parentheses follows a pair of curly brackets. All commands that defined the corresponding object are placed inside the curly brackets. A semicolon is not required at the end.

The second thing you should notice is that OpenSCAD has not created any wheel. This is because you have just defined the wheel module but have not used it yet. In order to create a wheel you need to add a statement that creates a wheel, similar to how you would add a statement to create any primitive object (cube, sphere etc.).

Code

wheel_created_by_module.scad

```
$fa = 1;
$fs = 0.4;
module wheel() {
    wheel_radius=10;
    side_spheres_radius=50;
    hub_thickness=4;
    cylinder_radius=2;
    cylinder_height=2*wheel_radius;
    difference() {
        // Wheel sphere
        sphere(r=wheel_radius);
        // Side sphere 1
        translate([0,side_spheres_radius + hub_thickness/2,0])
            sphere(r=side_spheres_radius);
        // Side sphere 2
        translate([0,- (side_spheres_radius + hub_thickness/2),0])
            sphere(r=side_spheres_radius);
        // Cylinder 1
        translate([wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 2
        translate([0,0,wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 3
        translate([-wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 4
        translate([0,0,-wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    }
}
wheel();
```



You can think of defining modules as extending the OpenSCAD scripting language. When you have defined a wheel module it's like having an additional available primitive object. In this case the new object is the wheel that you have defined. You can then use this module similar to how you would use any other available primitive.

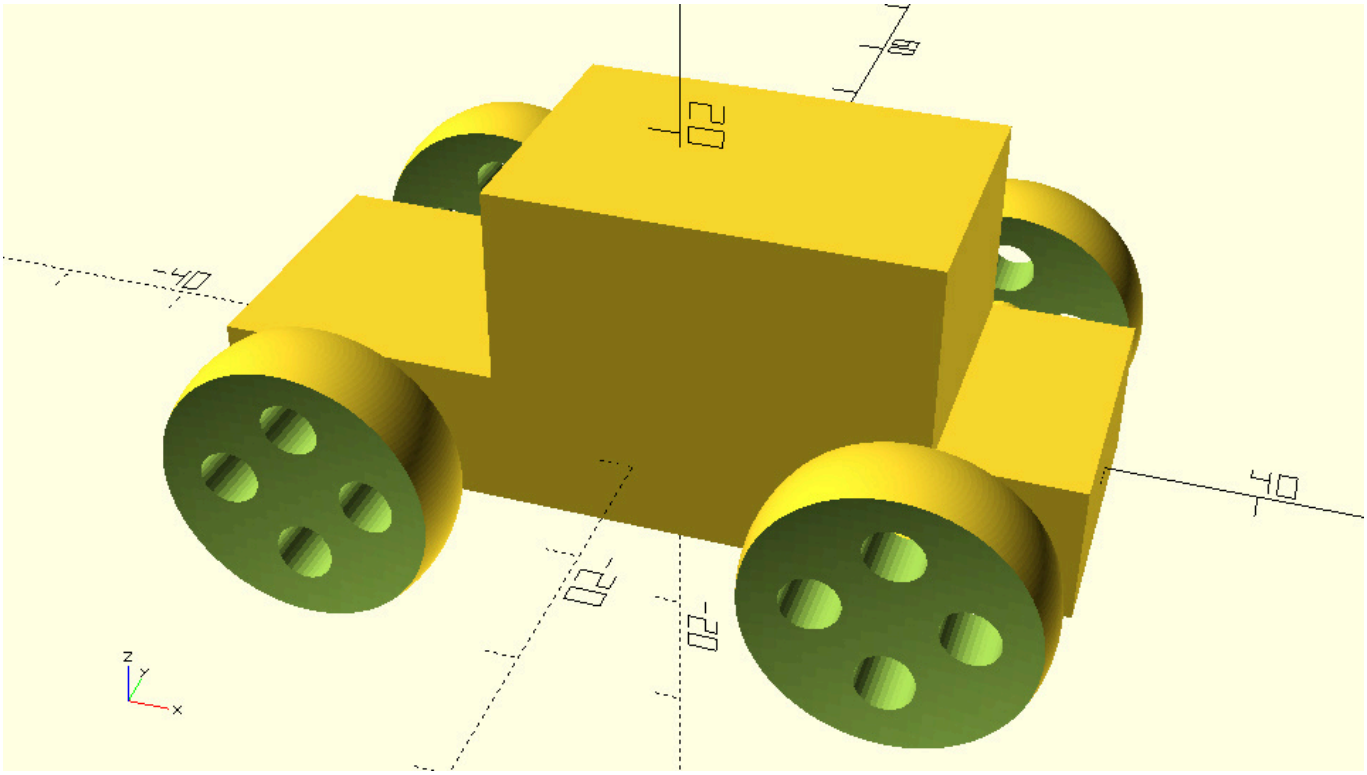
Exercise

Try defining the above wheel module in the car's script. Try creating the wheels of the car using the defined wheel module.

Code[Collapse]

car_with_wheels_created_by_module.scad

```
module wheel() {
    wheel_radius=10;
    side_spheres_radius=50;
    hub_thickness=4;
    cylinder_radius=2;
    cylinder_height=2*wheel_radius;
    difference() {
        // Wheel sphere
        sphere(r=wheel_radius);
        // Side sphere 1
        translate([0,side_spheres_radius + hub_thickness/2,0])
            sphere(r=side_spheres_radius);
        // Side sphere 2
        translate([0,- (side_spheres_radius + hub_thickness/2),0])
            sphere(r=side_spheres_radius);
        // Cylinder 1
        translate([wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 2
        translate([0,0,wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 3
        translate([-wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 4
        translate([0,0,-wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    }
}
$fa = 1;
$fs = 0.4;
base_height = 10;
top_height = 14;
track = 35;
body_roll = 0;
wheels_turn = 0;
rotate([body_roll,0,0]) {
    // Car body base
    cube([60,20,base_height],center=true);
    // Car body top
    translate([5,0,base_height/2+top_height/2 - 0.001])
        cube([30,20,top_height],center=true);
}
// Front left wheel
translate([-20,-track/2,0])
    rotate([0,0,wheels_turn])
    wheel();
 // Front right wheel
translate([-20,track/2,0])
    rotate([0,0,wheels_turn])
    wheel();
// Rear left wheel
translate([20,-track/2,0])
    rotate([0,0,0])
    wheel();
// Rear right wheel
translate([20,track/2,0])
    rotate([0,0,0])
    wheel();
// Front axle
translate([-20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
// Rear axle
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
```



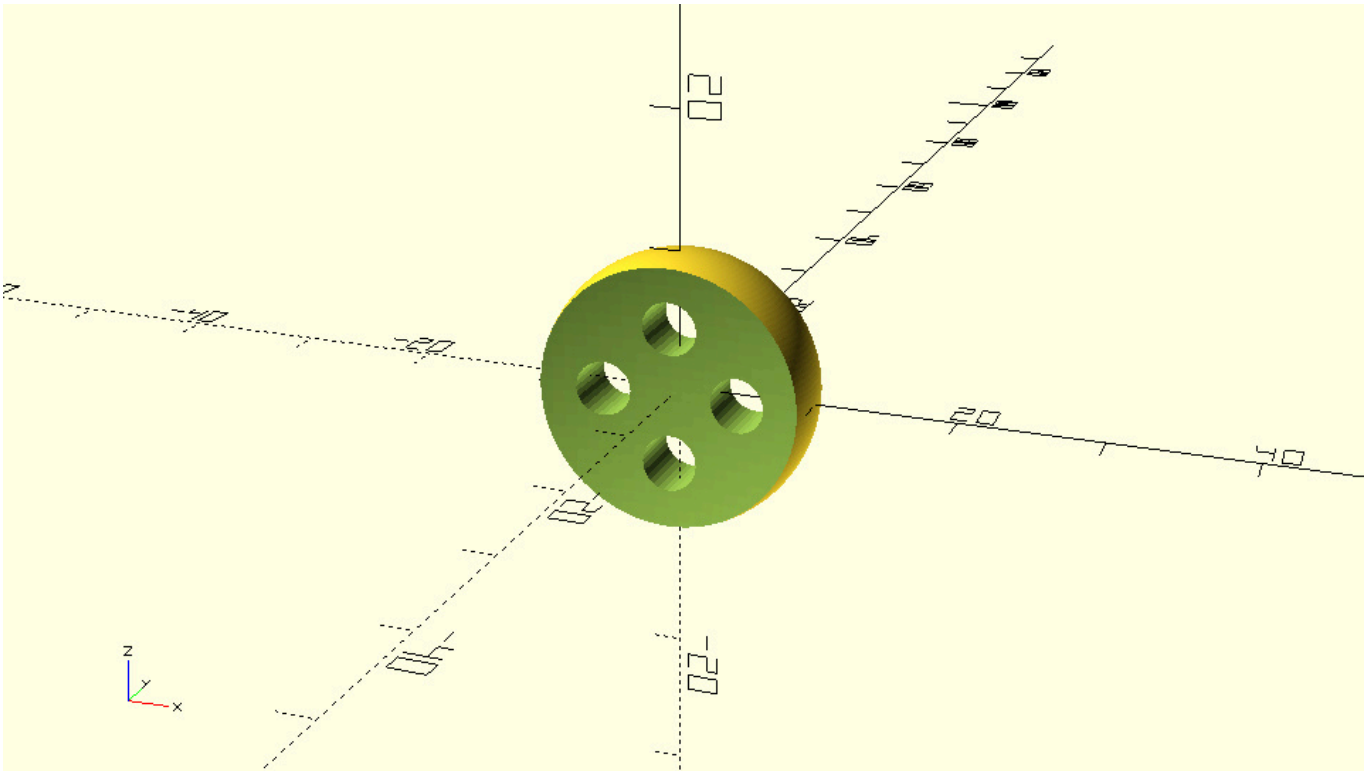
Parameterizing modules

The wheel design that was specified in the wheel module has a number of variables that can be used to customize it. These variables are defined inside the curly brackets of the wheel module's definition. As a result, while the output of the wheel module can be customized, the wheel module itself can create only one version of the wheel which corresponds to the values of the defined variables. This means the wheel module can't be used to create different wheels for the front and back axles. If you have been getting a feeling of the good practices of parametric design, you should realize that such a thing is not desired. It would be way better if the wheel module could be used to create different versions of the wheel. For this to happen the variables that are defined and used inside the wheel module, need to be defined as parameters of the wheel module instead. This can be done in the following way.

Code

wheel_created_by_parameterized_module.scad

```
$fa = 1;
$fs = 0.4;
module wheel(wheel_radius, side_spheres_radius, hub_thickness, cylinder_radius) {
    cylinder_height=2*wheel_radius;
    difference() {
        // Wheel sphere
        sphere(r=wheel_radius);
        // Side sphere 1
        translate([0,side_spheres_radius + hub_thickness/2,0])
            sphere(r=side_spheres_radius);
        // Side sphere 2
        translate([0,- (side_spheres_radius + hub_thickness/2),0])
            sphere(r=side_spheres_radius);
        // Cylinder 1
        translate([wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 2
        translate([0,0,wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 3
        translate([-wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 4
        translate([0,0,-wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    }
}
wheel(wheel_radius=10, side_spheres_radius=50, hub_thickness=4, cylinder_radius=2);
```



You should notice the definition of the module's parameters inside the parentheses. You should also notice that the value of each parameter is no longer assigned inside the curly brackets of the module's definitions. Instead, the value of the parameters is defined every time the modules is called. As a result, the module can now be used to create different versions of the wheel.

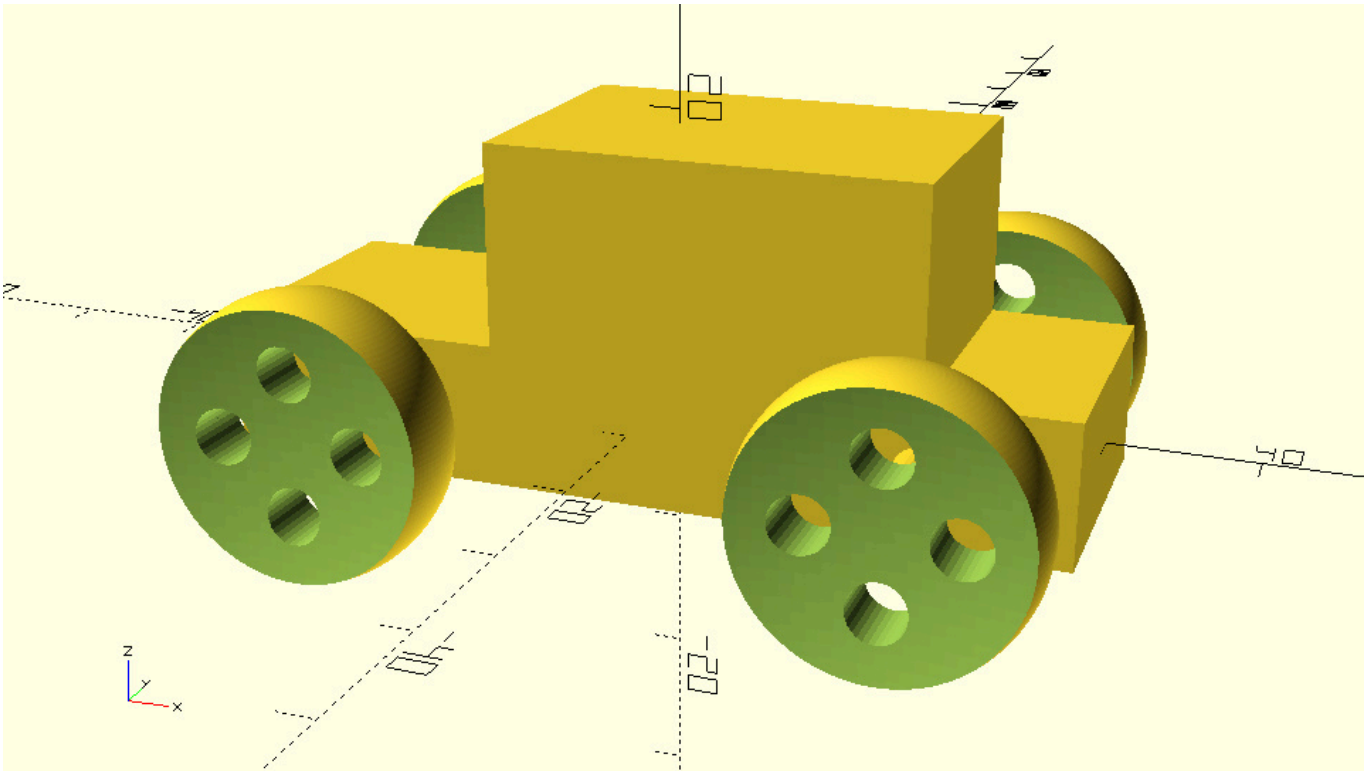
Exercise

Try defining the above wheel module in the car's script. Try creating the car's wheels by using the wheel module. When calling the wheel module pass the values of 10, 50, 4 and 2 to the corresponding wheel_radius, side_spheres_radius, hub_thickness and cylinder_radius parameters.

Code[Collapse]

car_with_wheels_created_by_parameterized_module.scad

```
module wheel(wheel_radius, side_spheres_radius, hub_thickness, cylinder_radius) {
    cylinder_height=2*wheel_radius;
    difference() {
        // Wheel sphere
        sphere(r=wheel_radius);
        // Side sphere 1
        translate([0,side_spheres_radius + hub_thickness/2,0])
            sphere(r=side_spheres_radius);
        // Side sphere 2
        translate([0,- (side_spheres_radius + hub_thickness/2),0])
            sphere(r=side_spheres_radius);
        // Cylinder 1
        translate([wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 2
        translate([0,0,wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 3
        translate([-wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 4
        translate([0,0,-wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    }
}
$fa = 1;
$fs = 0.4;
base_height = 10;
top_height = 14;
track = 35;
body_roll = 0;
wheels_turn = 0;
rotate([body_roll,0,0]) {
    // Car body base
    cube([60,20,base_height],center=true);
    // Car body top
    translate([5,0,base_height/2+top_height/2 - 0.001])
        cube([30,20,top_height],center=true);
}
// Front left wheel
translate([-20,-track/2,0])
    rotate([0,0,wheels_turn])
    wheel(wheel_radius=10, side_spheres_radius=50, hub_thickness=4, cylinder_radius=2);
 // Front right wheel
translate([-20,track/2,0])
    rotate([0,0,wheels_turn])
    wheel(wheel_radius=10, side_spheres_radius=50, hub_thickness=4, cylinder_radius=2);
// Rear left wheel
translate([20,-track/2,0])
    rotate([0,0,0])
    wheel(wheel_radius=10, side_spheres_radius=50, hub_thickness=4, cylinder_radius=2);
// Rear right wheel
translate([20,track/2,0])
    rotate([0,0,0])
    wheel(wheel_radius=10, side_spheres_radius=50, hub_thickness=4, cylinder_radius=2);
// Front axle
translate([-20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
// Rear axle
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
```



Exercise

Try defining a wheel_radius, side_spheres_radius, hub_thickness and cylinder_radius variable in the car's script and assign the values of 10, 50, 4 and 2 accordingly. Try using these variables to define the values of the wheel_radius, side_spheres_radius, hub_thickness and cylinder_radius parameters when calling the wheel module.

Code[Collapse]

```
wheel_radius=10;
side_spheres_radius=50;
hub_thickness=4;
cylinder_radius=2;
wheel(wheel_radius=wheel_radius, side_spheres_radius=side_spheres_radius,
hub_thickness=hub_thickness, cylinder_radius=cylinder_radius);
```

Exercise

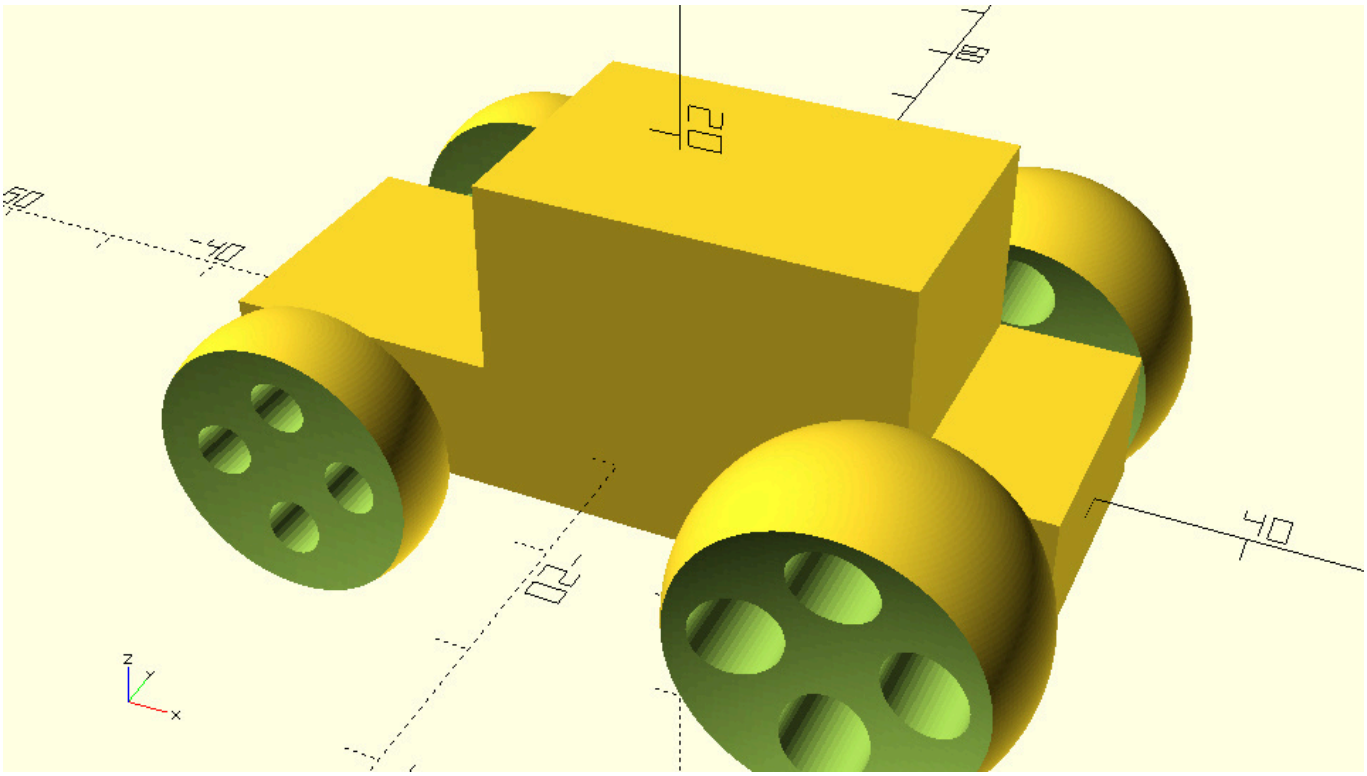
Try defining different wheel_radius, side_spheres_radius, hub_thickness and cylinder_radius variables for the front and rear axles. Try assigning a combination of values that you like to these variables. Remember to also edit the name of the variables in the respective calls of the wheel module.

Code[Collapse]

car_with_different_wheels.scad

```
module wheel(wheel_radius, side_spheres_radius, hub_thickness, cylinder_radius) {
    cylinder_height=2*wheel_radius;
    difference() {
        // Wheel sphere
        sphere(r=wheel_radius);
        // Side sphere 1
        translate([0,side_spheres_radius + hub_thickness/2,0])
            sphere(r=side_spheres_radius);
        // Side sphere 2
        translate([0,- (side_spheres_radius + hub_thickness/2),0])
            sphere(r=side_spheres_radius);
        // Cylinder 1
        translate([wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 2
        translate([0,0,wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 3
        translate([-wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 4
        translate([0,0,-wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    }
}
$fa = 1;
$fs = 0.4;
base_height = 10;
top_height = 14;
track = 35;
body_roll = 0;
wheels_turn = 0;

wheel_radius_front=10;
side_spheres_radius_front=50;
hub_thickness_front=4;
cylinder_radius_front=2;

wheel_radius_rear=12;
side_spheres_radius_rear=30;
hub_thickness_rear=8;
cylinder_radius_rear=3;

rotate([body_roll,0,0]) {
    // Car body base
    cube([60,20,base_height],center=true);
    // Car body top
    translate([5,0,base_height/2+top_height/2 - 0.001])
        cube([30,20,top_height],center=true);
}
// Front left wheel
translate([-20,-track/2,0])
    rotate([0,0,wheels_turn])
    wheel(wheel_radius=wheel_radius_front, side_spheres_radius=side_spheres_radius_front,
    hub_thickness=hub_thickness_front, cylinder_radius=cylinder_radius_front);
 // Front right wheel
translate([-20,track/2,0])
    rotate([0,0,wheels_turn])
    wheel(wheel_radius=wheel_radius_front, side_spheres_radius=side_spheres_radius_front,
    hub_thickness=hub_thickness_front, cylinder_radius=cylinder_radius_front);
// Rear left wheel
translate([20,-track/2,0])
    rotate([0,0,0])
    wheel(wheel_radius=wheel_radius_rear, side_spheres_radius=side_spheres_radius_rear,
    hub_thickness=hub_thickness_rear, cylinder_radius=cylinder_radius_rear);
// Rear right wheel
translate([20,track/2,0])
    rotate([0,0,0])
    wheel(wheel_radius=wheel_radius_rear, side_spheres_radius=side_spheres_radius_rear,
    hub_thickness=hub_thickness_rear, cylinder_radius=cylinder_radius_rear);
```

```
// Front axle
translate([-20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
// Rear axle
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
```



Defining default values of module's parameters

You can set a specific combination of values for the wheel module's parameters as default. This can be achieved in the following way.

Code

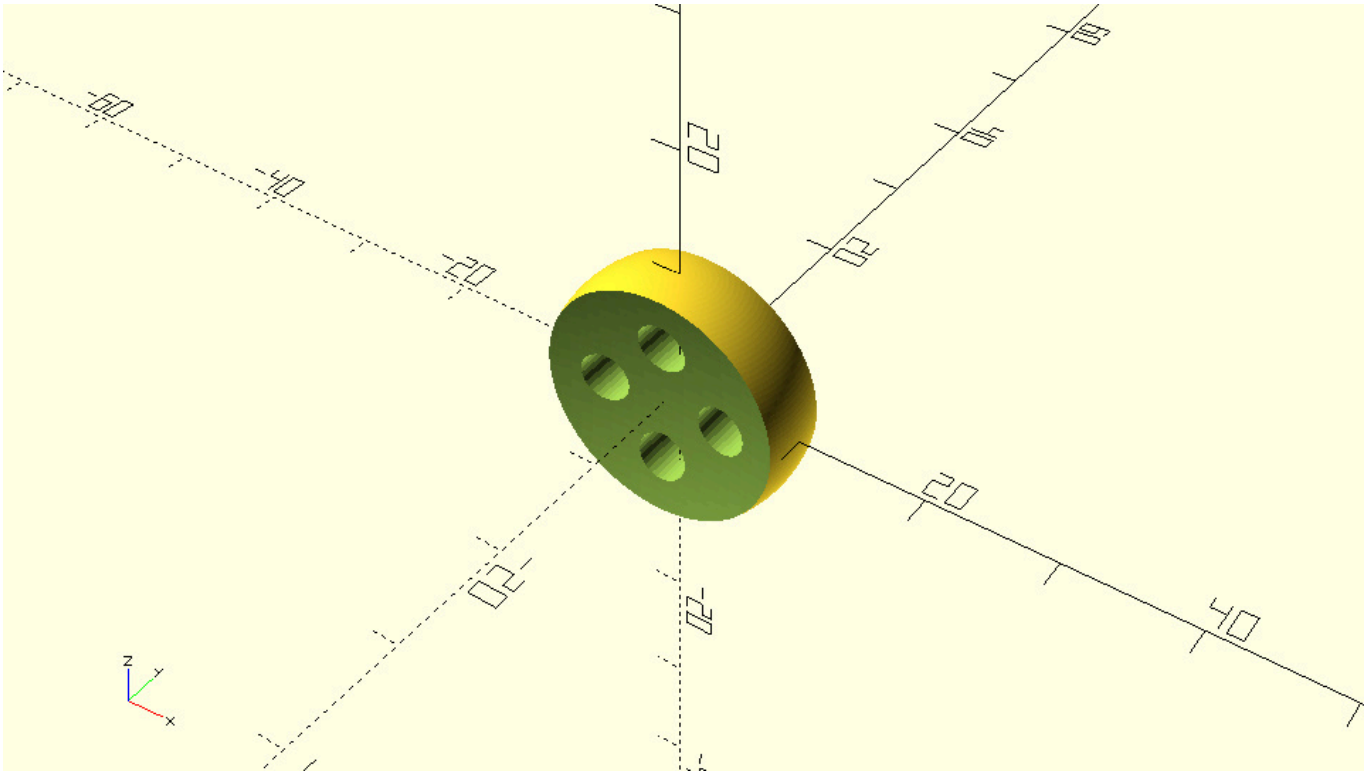
```
$fa = 1;
$fs = 0.4;
module wheel(wheel_radius=10, side_spheres_radius=50, hub_thickness=4, cylinder_radius=2) {
    cylinder_height=2*wheel_radius;
    difference() {
        // Wheel sphere
        sphere(r=wheel_radius);
        // Side sphere 1
        translate([0,side_spheres_radius + hub_thickness/2,0])
            sphere(r=side_spheres_radius);
        // Side sphere 2
        translate([0,- (side_spheres_radius + hub_thickness/2),0])
            sphere(r=side_spheres_radius);
        // Cylinder 1
        translate([wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 2
        translate([0,0,wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 3
        translate([-wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 4
        translate([0,0,-wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    }
}
```

You should notice that the default values are assigned inside the parentheses at the definition of the module. By defining default values for the module's parameters, you have more flexibility in the way the wheel module is used. For example, the simplest way to use the module is without specifying any parameters when calling it.

Code

wheel_created_by_default_parameters.scad

```
…
wheel();
…
```

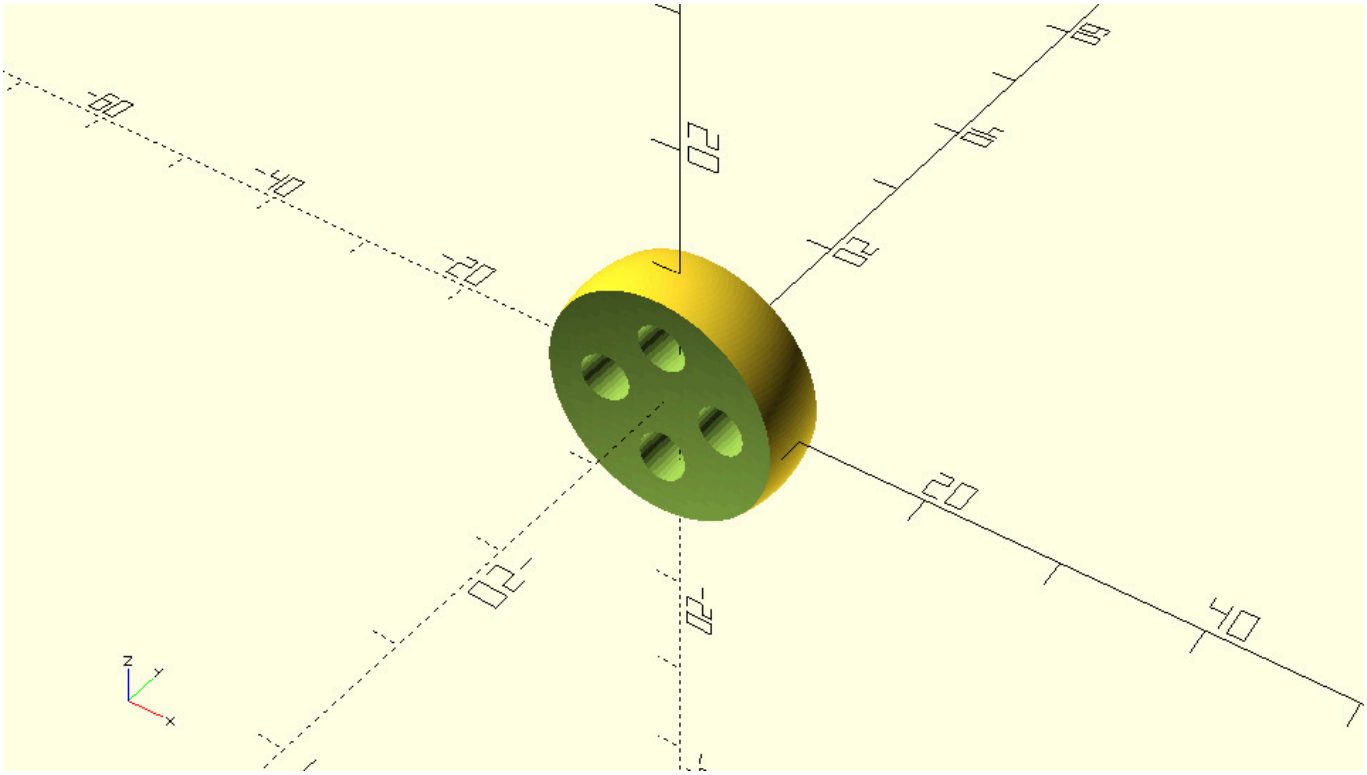


If a parameter's value is not specified when calling the wheel module, the default value for this parameter is used. The default values can be set equal to the most used version of the wheel. The default values can be overridden by assigning a new value to the corresponding parameters when calling the wheel module. None or any number of default values may be overridden. Thus, by specifying default values the wheel module can be used in any of the following ways as well as in many more.

Code

wheel_created_by_default_parameters.scad

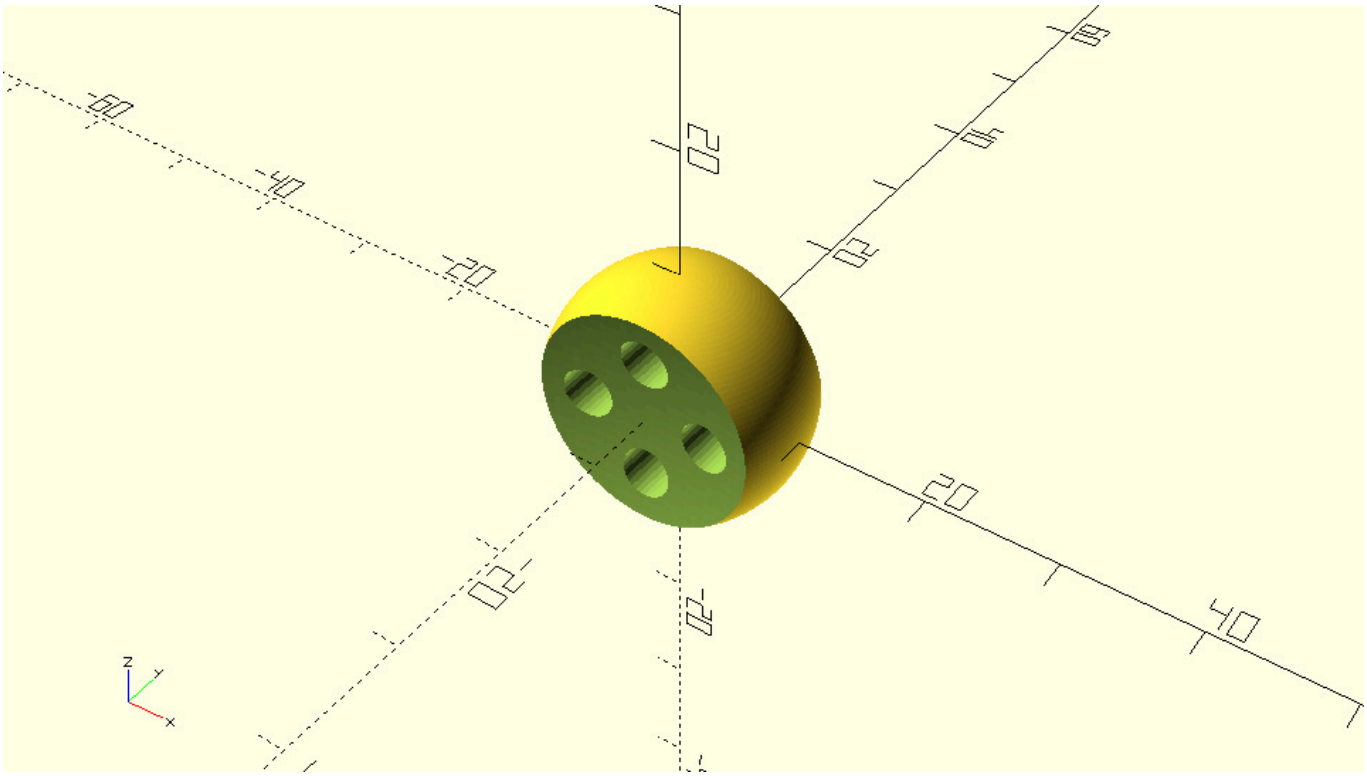
```
…
wheel();
…
```



Code

wheel_with_thicker_hub.scad

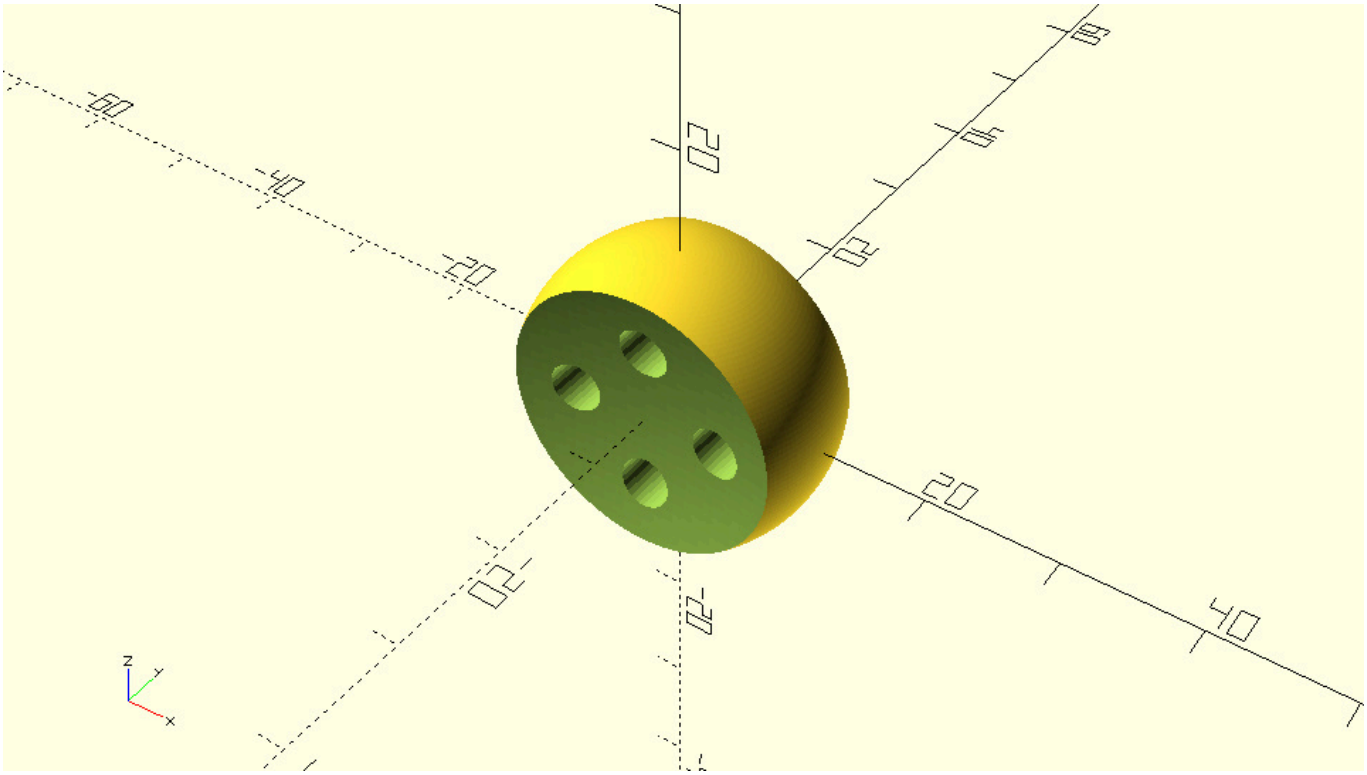
```
…
wheel(hub_thickness=8);
…
```



Code

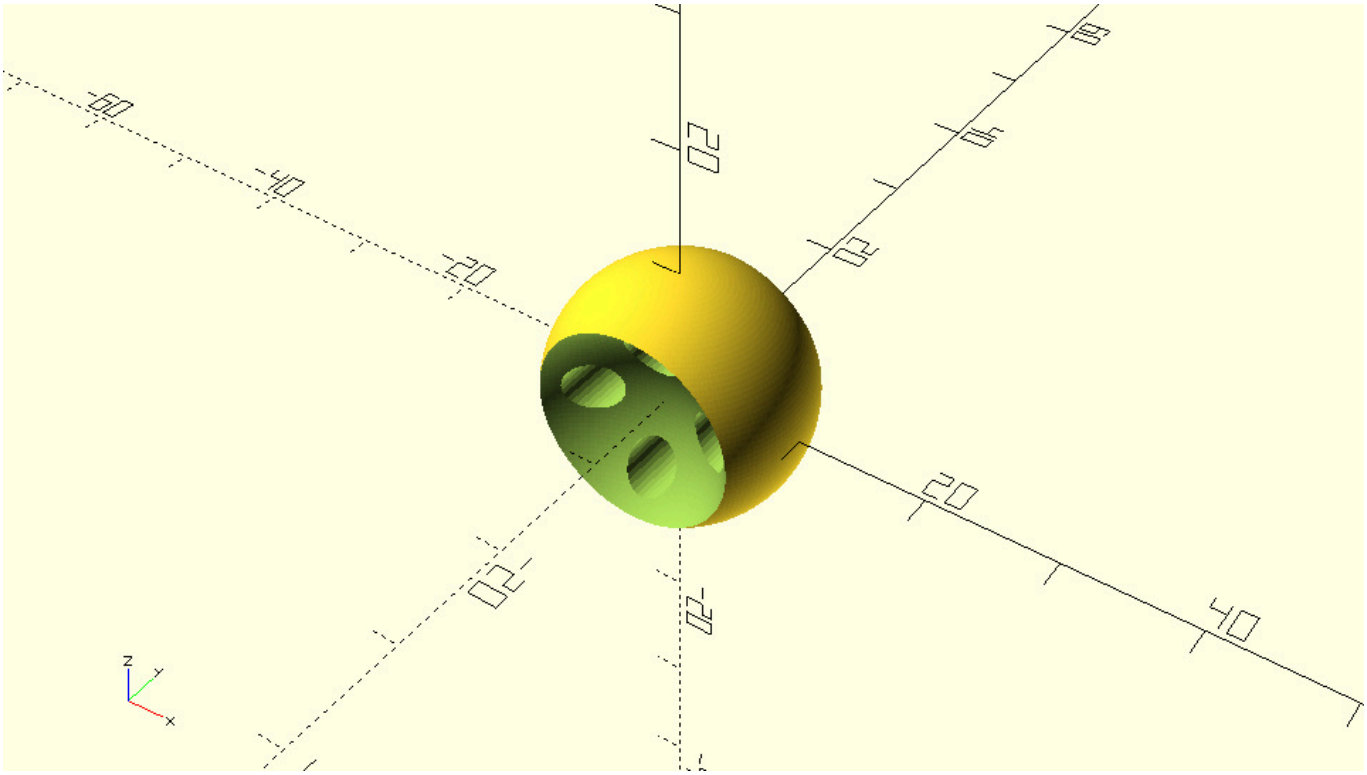
wheel_with_thicker_hub_and_larger_radius.scad

```
…
wheel(hub_thickness=8, wheel_radius=12);
…
```



Exercise

Include default values in the definition of the wheel module. Try creating a few wheels by overriding any number of default values. Can you make a wheel that looks like the following?



Code
[Expand]

Separating the whole model into modules

The use of modules is a very powerful feature of OpenSCAD. You should start thinking of your models as a combination of modules. For example, the car model can be thought of as a combination of a body, wheel and axle module. This opens the possibilities of further reusing and recombining your modules to create different models.

Exercise

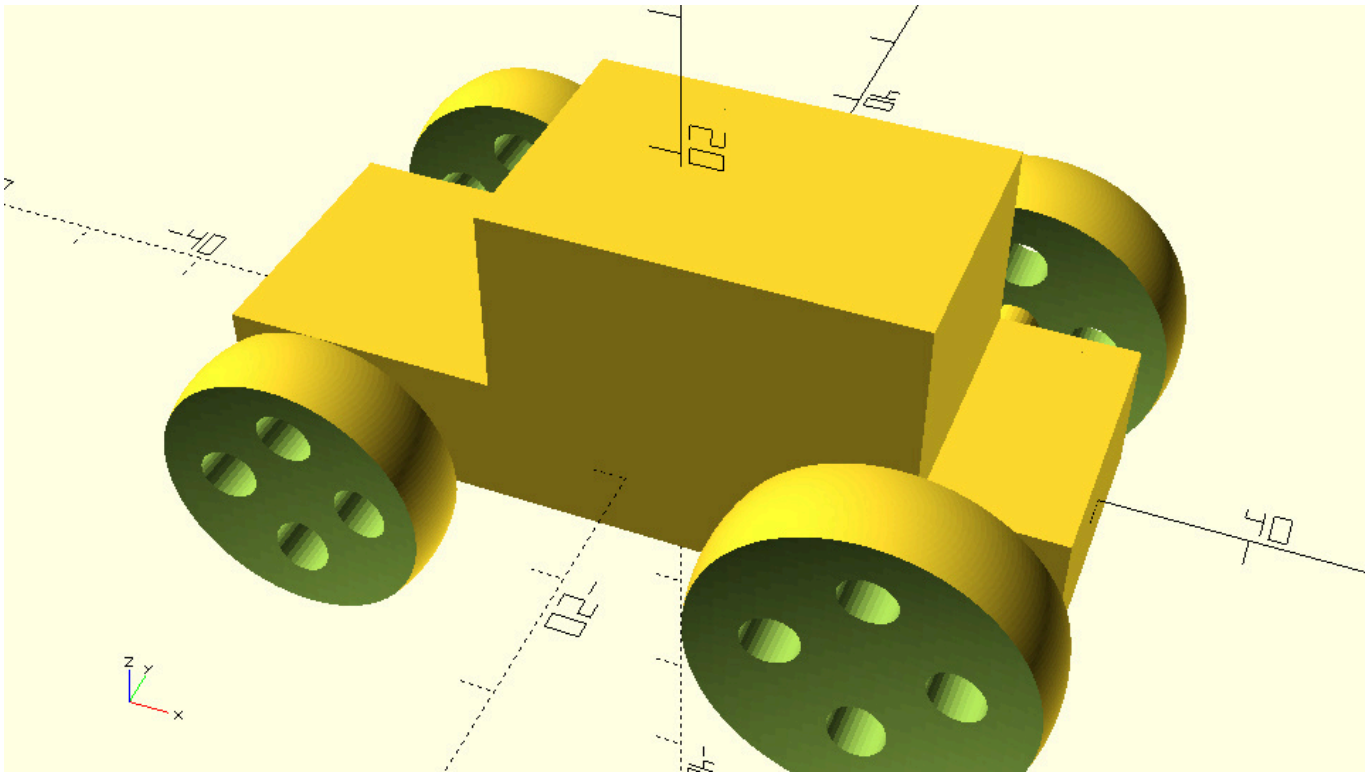
Try defining a body and an axle module. What parameters should the body and axle modules have? Try recreating the car using the body, wheel and axle modules. Give the parameters of the wheel module a default set of values that corresponds to the front wheels. Pass different values to the wheel module when creating the rear wheels by defining appropriate variables in your script. Set default values for the parameters of the body and axle modules too.

Code[Collapse]

car_with_different_wheels_and_default_body_and_axle.scad

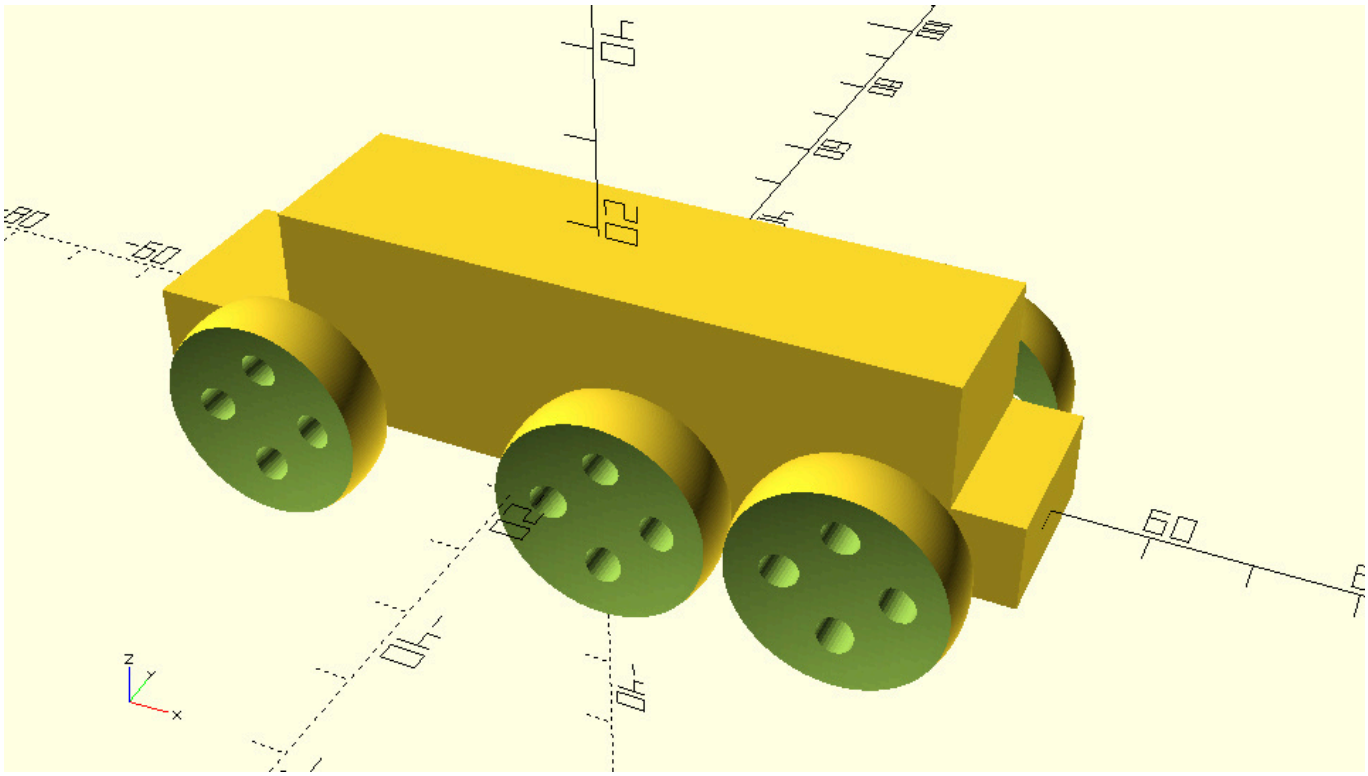
```
module wheel(wheel_radius=10, side_spheres_radius=50, hub_thickness=4, cylinder_radius=2) {
    cylinder_height=2*wheel_radius;
    difference() {
        // Wheel sphere
        sphere(r=wheel_radius);
        // Side sphere 1
        translate([0,side_spheres_radius + hub_thickness/2,0])
            sphere(r=side_spheres_radius);
        // Side sphere 2
        translate([0,- (side_spheres_radius + hub_thickness/2),0])
            sphere(r=side_spheres_radius);
        // Cylinder 1
        translate([wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 2
        translate([0,0,wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 3
        translate([-wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 4
        translate([0,0,-wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    }
}

module body(base_height=10, top_height=14, base_length=60, top_length=30, width=20, top_offset=5) {
    // Car body base
    cube([base_length,width,base_height],center=true);
    // Car body top
    translate([top_offset,0,base_height/2+top_height/2 - 0.001])
        cube([top_length,width,top_height],center=true);
}

module axle(track=35, radius=2) {
    rotate([90,0,0])
        cylinder(h=track,r=radius,center=true);
}

$fa = 1;
$fs = 0.4;
wheelbase = 40;
track = 35;
body_roll = 0;
wheels_turn = 0;
wheel_radius_rear=12;

// Body
rotate([body_roll,0,0]) {
    body();
}
// Front left wheel
translate([-wheelbase/2,-track/2,0])
    rotate([0,0,wheels_turn])
    wheel();
 // Front right wheel
translate([-wheelbase/2,track/2,0])
    rotate([0,0,wheels_turn])
    wheel();
// Rear left wheel
translate([wheelbase/2,-track/2,0])
    rotate([0,0,0])
    wheel(wheel_radius=wheel_radius_rear);
// Rear right wheel
translate([wheelbase/2,track/2,0])
    rotate([0,0,0])
    wheel(wheel_radius=wheel_radius_rear);
// Front axle
translate([-wheelbase/2,0,0])
    axle();
```

```
// Rear axle
translate([wheelbase/2,0,0])
    axle();
```



Exercise

Try reusing the body, wheel and axle modules to create vehicle that looks similar to the following.



Code[Collapse]

car_with_six_wheels.scad

```
module wheel(wheel_radius=10, side_spheres_radius=50, hub_thickness=4, cylinder_radius=2) {
    cylinder_height=2*wheel_radius;
    difference() {
        // Wheel sphere
        sphere(r=wheel_radius);
        // Side sphere 1
        translate([0,side_spheres_radius + hub_thickness/2,0])
            sphere(r=side_spheres_radius);
        // Side sphere 2
        translate([0,- (side_spheres_radius + hub_thickness/2),0])
            sphere(r=side_spheres_radius);
        // Cylinder 1
        translate([wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 2
        translate([0,0,wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 3
        translate([-wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 4
        translate([0,0,-wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    }
}

module body(base_height=10, top_height=14, base_length=60, top_length=30, width=20, top_offset=5) {
    // Car body base
    cube([base_length,width,base_height],center=true);
    // Car body top
    translate([top_offset,0,base_height/2+top_height/2 - 0.001])
        cube([top_length,width,top_height],center=true);
}

module axle(track=35, radius=2) {
    rotate([90,0,0])
        cylinder(h=track,r=radius,center=true);
}

$fa = 1;
$fs = 0.4;
track = 35;
body_roll = 0;
wheels_turn = 0;
base_length = 100;
top_length = 75;
top_offset = 5;
front_axle_offset = 30;
rear_axle_1_offset = 10;
rear_axle_2_offset = 35;
wheel_radius = 12;

// Body
rotate([body_roll,0,0]) {
    body(base_length=base_length, top_length=top_length, top_offset=top_offset);
}
// Front left wheel
translate([-front_axle_offset,-track/2,0])
    rotate([0,0,wheels_turn])
    wheel(wheel_radius=wheel_radius);
 // Front right wheel
translate([-front_axle_offset,track/2,0])
    rotate([0,0,wheels_turn])
    wheel(wheel_radius=wheel_radius);
// Rear left wheel 1
translate([rear_axle_1_offset,-track/2,0])
    rotate([0,0,0])
    wheel(wheel_radius=wheel_radius);
// Rear right wheel 1
translate([rear_axle_1_offset,track/2,0])
```

```
    rotate([0,0,0])
    wheel(wheel_radius=wheel_radius);
// Rear left wheel 2
translate([rear_axle_2_offset,-track/2,0])
    rotate([0,0,0])
    wheel(wheel_radius=wheel_radius);
// Rear right wheel 2
translate([rear_axle_2_offset,track/2,0])
    rotate([0,0,0])
    wheel(wheel_radius=wheel_radius);
// Front axle
translate([-front_axle_offset,0,0])
    axle();
// Rear axle 1
translate([rear_axle_1_offset,0,0])
    axle();
// Rear axle 2
translate([rear_axle_2_offset,0,0])
    axle();
```

Chapter 5

Creating and utilizing modules as separate scripts

In the previous chapter you learned one of the most powerful features of OpenSCAD, the module, and how it can be used for parametric design. You also had the chance to separate the car into different modules and then recombine them to create a different type of vehicle. Using modules can be also seen as a way to organize your creations and to build your own library of objects. The wheel module could potentially be used in a plethora of designs, so it would be great to have it easily available whenever desired without having to redefine it inside the script of your current design. To do so you need to define and save the wheel module as a separate script.

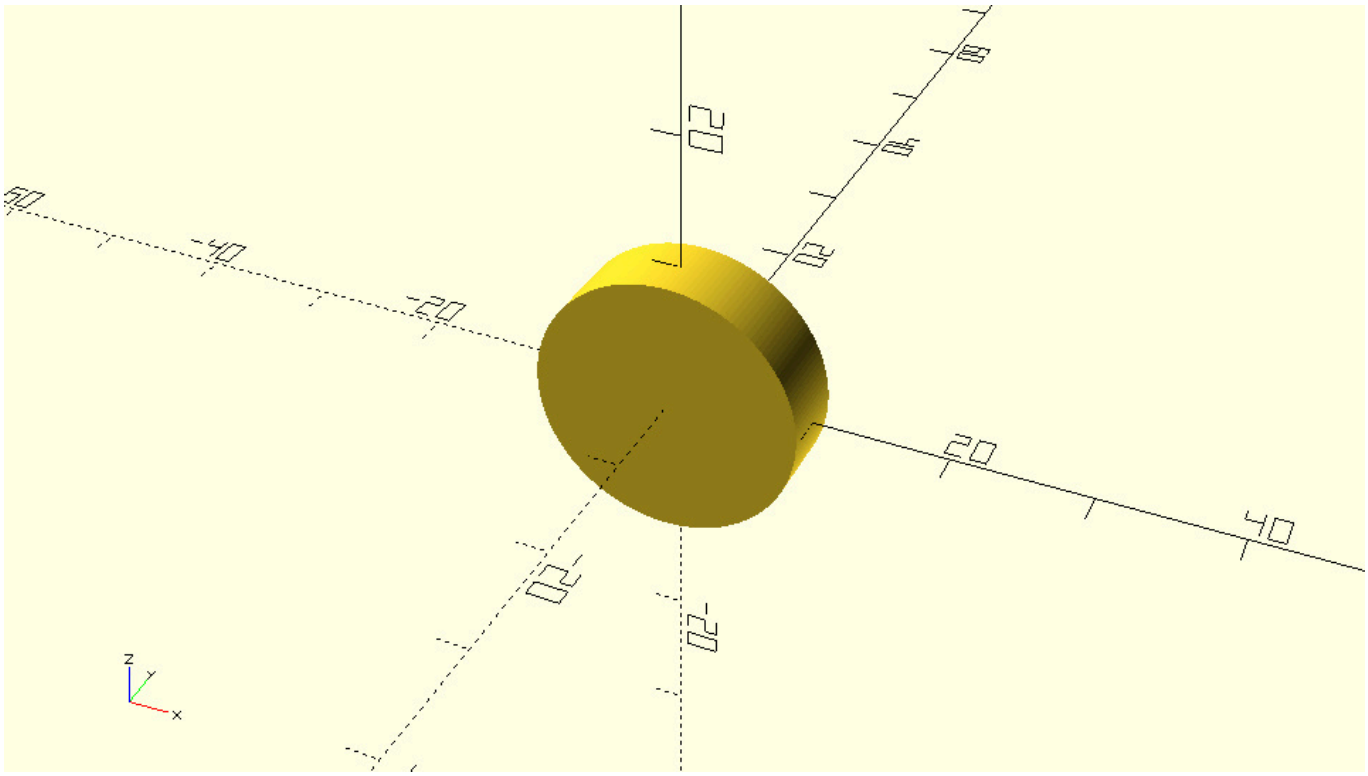
Exercise

Define the following simple_wheel module in a separate script file. In the same script make a call to the simple_wheel module so that you visually see what object this module creates. Save the script file as a scad file named simple_wheel.scad.

Code

simple_wheel.scad

```
$fa = 1;
$fs = 0.4;
module simple_wheel(wheel_radius=10, wheel_width=6) {
    rotate([90,0,0])
        cylinder(h=wheel_width,r=wheel_radius,center=true);
}
simple_wheel();
```



Now it's time to utilize this saved module in another design. First you need to create a new design.

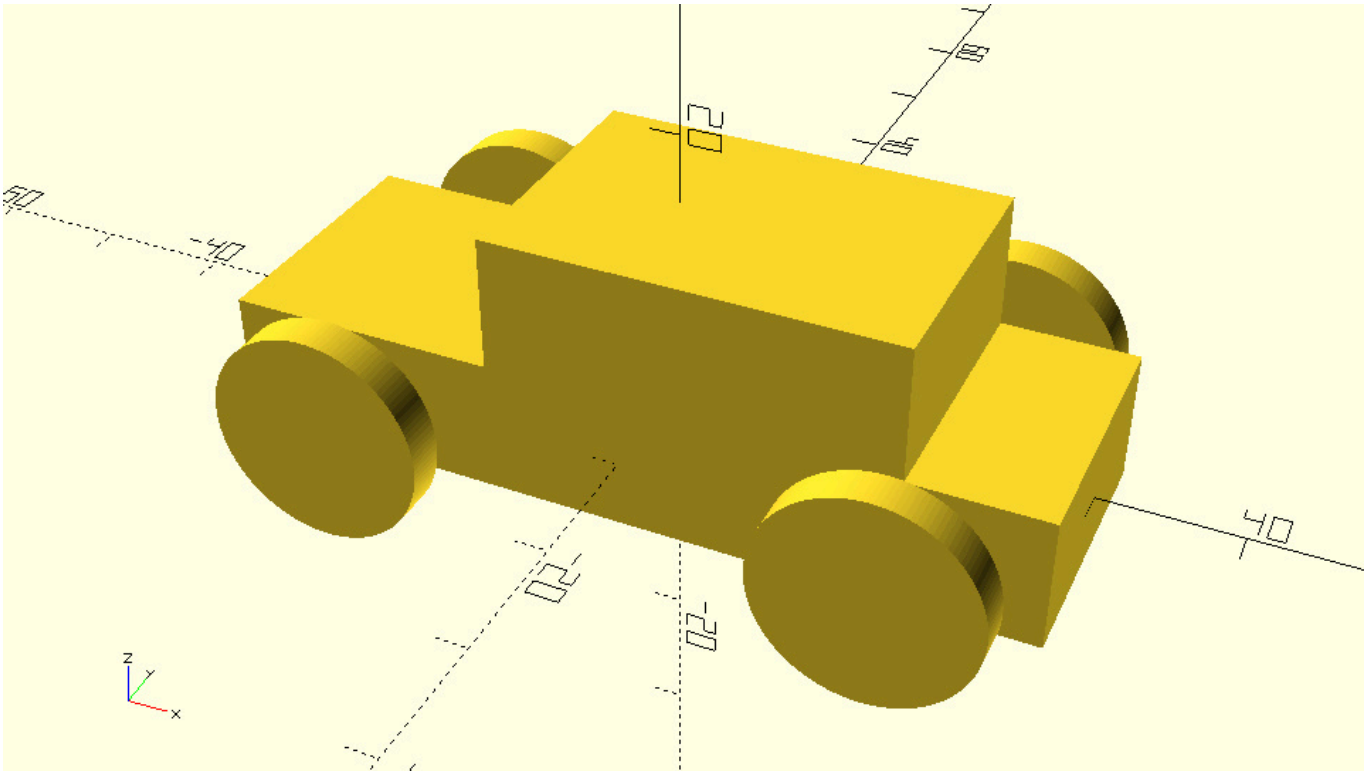
Exercise

Create a new script with the following car design. Give the script any name you like but save the script in the same working directory as the simple_wheel module.

Code[Collapse]

car_with_simple_wheels.scad

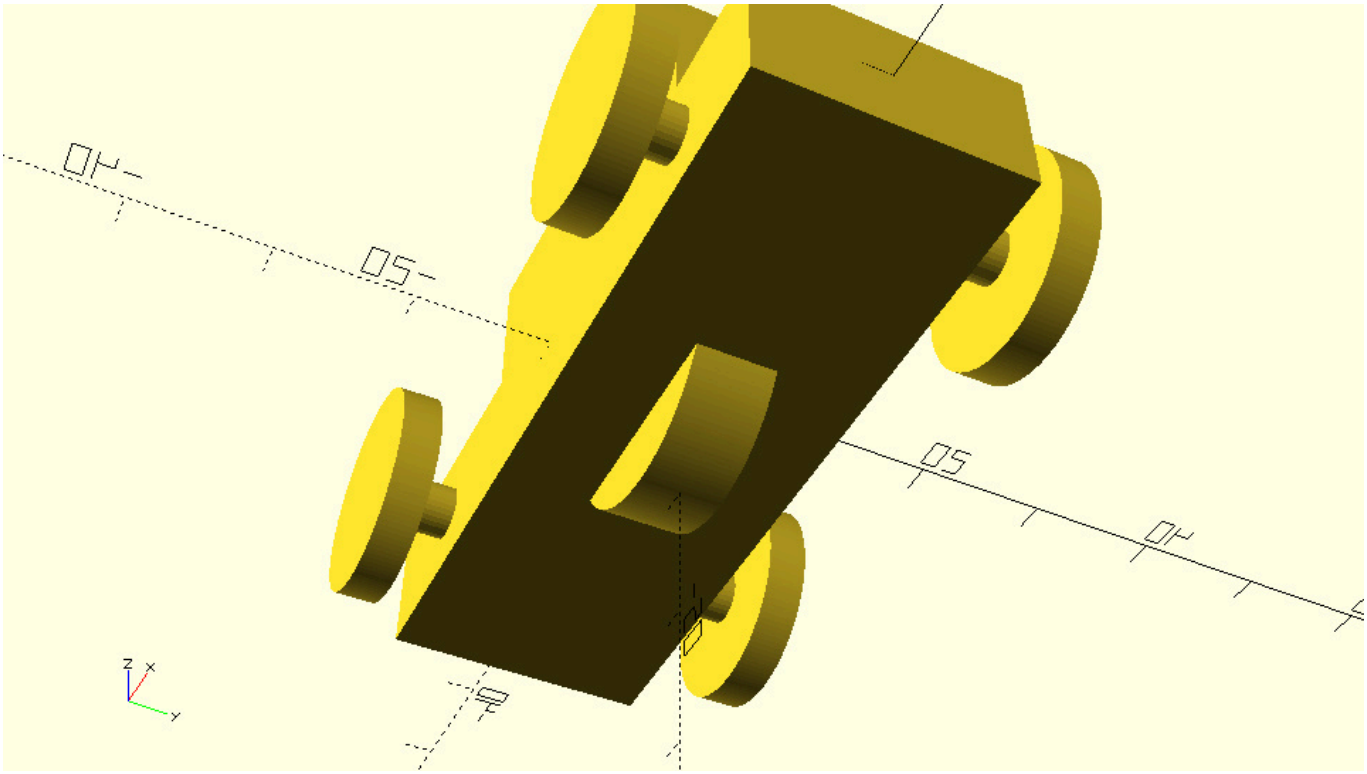
```
$fa = 1;
$fs = 0.4;
wheel_radius = 8;
base_height = 10;
top_height = 10;
track = 30;
// Car body base
cube([60,20,base_height],center=true);
// Car body top
translate([5,0,base_height/2+top_height/2 - 0.001])
    cube([30,20,top_height],center=true);
// Front left wheel
translate([-20,-track/2,0])
    rotate([90,0,0])
    cylinder(h=3,r=wheel_radius,center=true);
// Front right wheel
translate([-20,track/2,0])
    rotate([90,0,0])
    cylinder(h=3,r=wheel_radius,center=true);
// Rear left wheel
translate([20,-track/2,0])
    rotate([90,0,0])
    cylinder(h=3,r=wheel_radius,center=true);
// Rear right wheel
translate([20,track/2,0])
    rotate([90,0,0])
    cylinder(h=3,r=wheel_radius,center=true);
// Front axle
translate([-20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
// Rear axle
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
```



There are two ways in which the simple_wheel.scad script can be utilized in your car design. It can be either included or used. To include the script, you have to add the following statement at the top of car's script.

Code

```
include <simple_wheel.scad>
```



You should notice that something unexpected happened. A wheel has been created at the origin. This is the object of the simple_wheel script. When you use include, OpenSCAD treats the whole external script that you are including as if it was a part of your current script. In the simple_wheel.scad script, aside from the simple_wheel module definition, there is also a call to the simple_wheel module which creates a wheel object. As a result of using the include command this object is also created in the car's model. This is something that you are going to change by using the use instead of the include command, but don't bother about it for a moment.

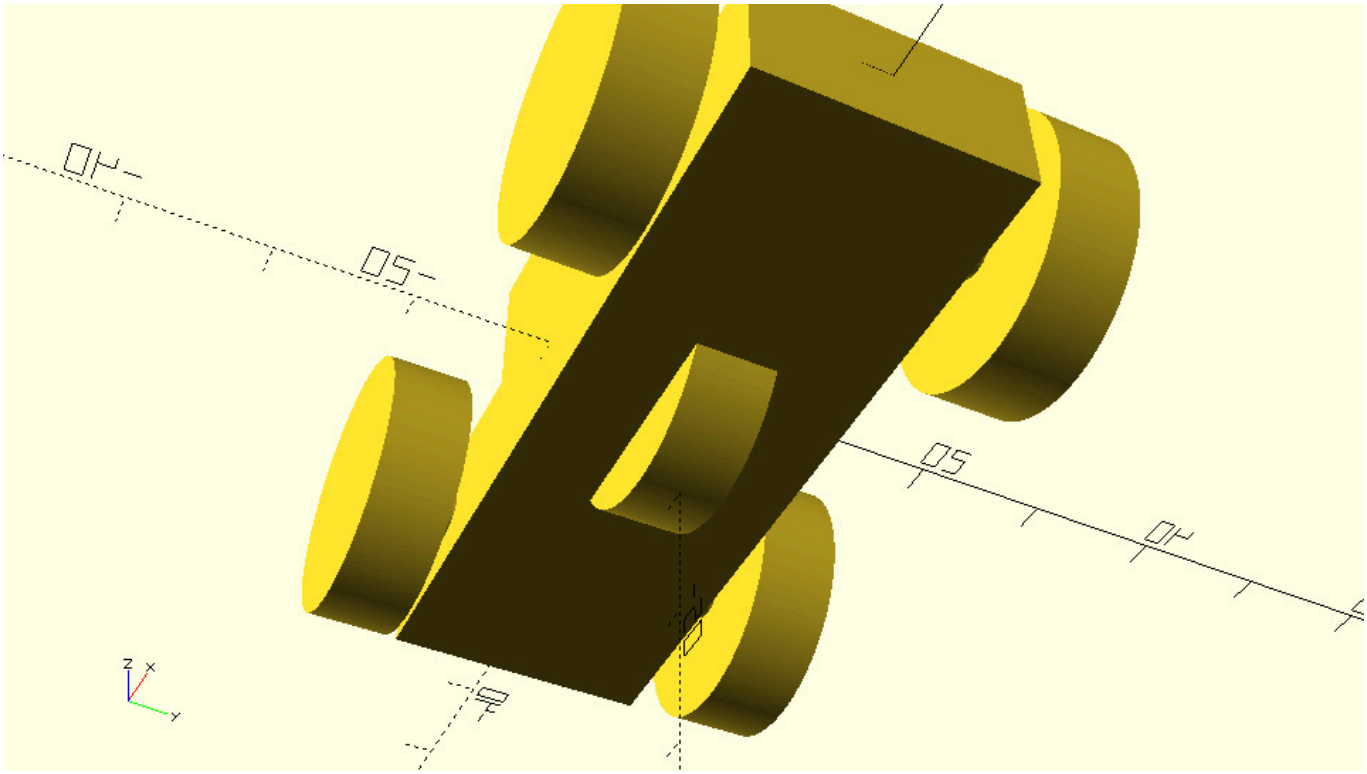
Exercise

The car's wheels are currently created with the cylinder command. Since the simple_wheel.scad script has been included in the car's script, the simple_wheel module should be available. Replace the cylinder commands with calls to the simple_wheel module. Do any rotate commands become unnecessary? The calls to the simple_wheel module shall not contain any definition of parameters.

Code[Collapse]

car_with_wheels_created_by_included_module.scad

```
include <simple_wheel.scad>
$fa = 1;
$fs = 0.4;
wheel_radius = 8;
base_height = 10;
top_height = 10;
track = 30;
// Car body base
cube([60,20,base_height],center=true);
// Car body top
translate([5,0,base_height/2+top_height/2 - 0.001])
    cube([30,20,top_height],center=true);
// Front left wheel
translate([-20,-track/2,0])
    simple_wheel();
// Front right wheel
translate([-20,track/2,0])
    simple_wheel();
// Rear left wheel
translate([20,-track/2,0])
    simple_wheel();
// Rear right wheel
translate([20,track/2,0])
    simple_wheel();
// Front axle
translate([-20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
// Rear axle
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
```



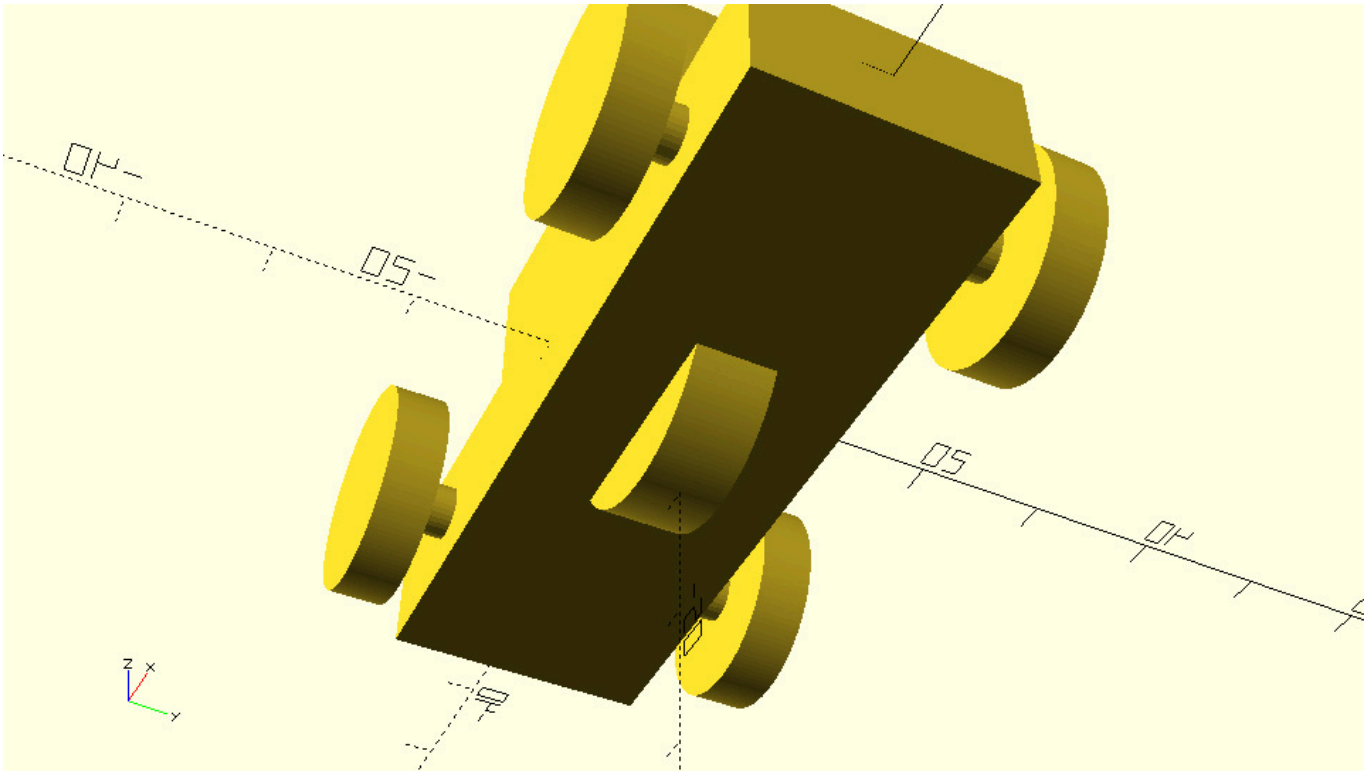
Exercise

Define the wheel_radius and wheel_width parameters in the calls to the simple_wheel module. To do so use the existing wheel_radius variable as well as a wheel_width variable that you are going to define. Set the variables equal to values that you like.

Code[Collapse]

car_with_narrower_wheels_created_by_included_module.scad

```
include <simple_wheel.scad>
$fa = 1;
$fs = 0.4;
wheel_radius = 8;
wheel_width = 4;
base_height = 10;
top_height = 10;
track = 30;
// Car body base
cube([60,20,base_height],center=true);
// Car body top
translate([5,0,base_height/2+top_height/2 - 0.001])
    cube([30,20,top_height],center=true);
// Front left wheel
translate([-20,-track/2,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Front right wheel
translate([-20,track/2,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Rear left wheel
translate([20,-track/2,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Rear right wheel
translate([20,track/2,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Front axle
translate([-20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
// Rear axle
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
```



From the above examples you should keep in mind that when you include an external script in your current script, the modules of the external script become available in your current script, but additionally any objects that were created in the external script are also created in the current one. Since the wheel at the origin is not desired in this case, it's time to use the use command instead of the include.

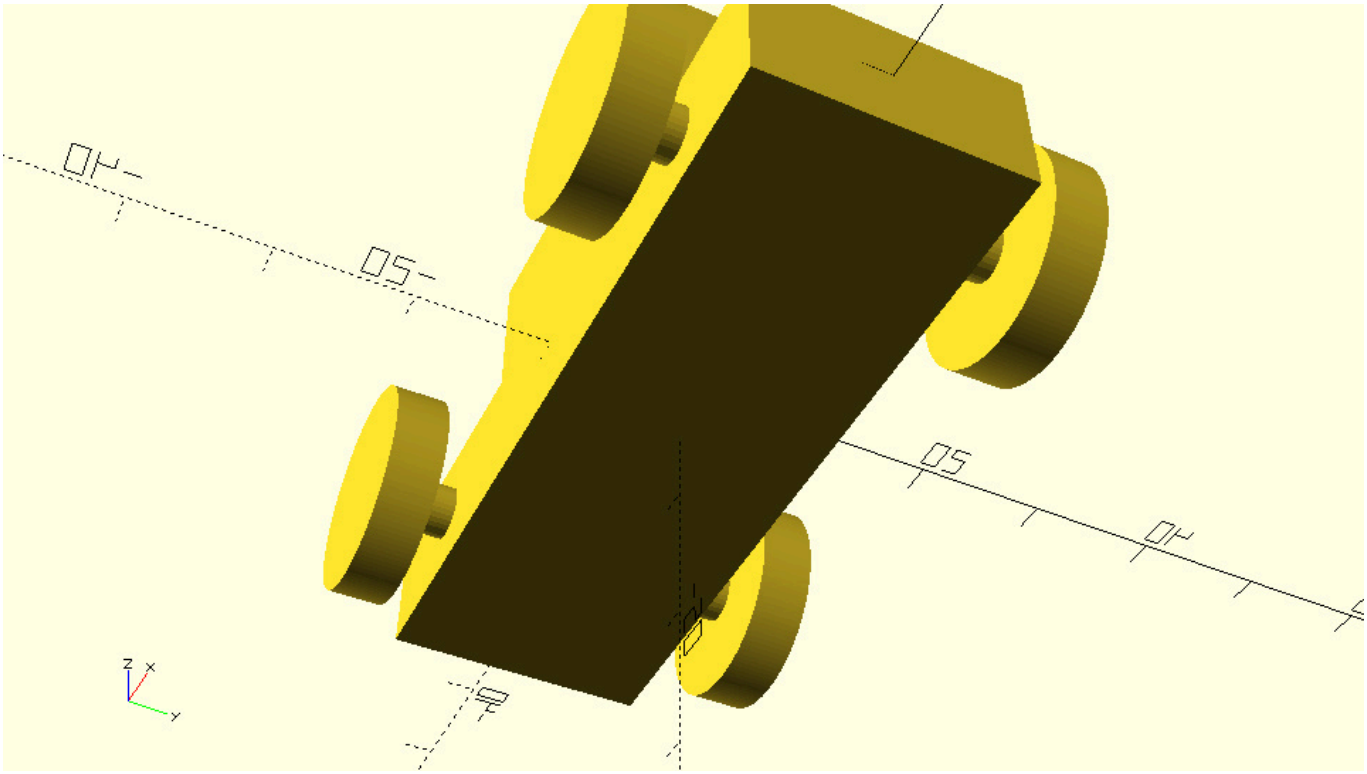
Exercise

Replace the include command of the last example with a use command.

Code[Collapse]

car_with_wheels_created_by_used_module.scad

```
…
use <simple_wheel.scad>
…
```



You should notice that a wheel is no longer created at the origin. You should keep in mind that the use command works like the include command with the only difference being that the use command doesn't create any objects, but rather just makes the modules of the external script available in the current script.

Using a script with multiple modules

In the previous example, the simple_wheel.scad script had only one module. The simple_wheel module. This doesn't have to always be the case.

Exercise

Add the following module in the simple_wheel.scad script. Rename the simple_wheel.scad script to wheels.scad.

Code

```
module complex_wheel(wheel_radius=10, side_spheres_radius=50, hub_thickness=4, cylinder_radius=2) {
    cylinder_height=2*wheel_radius;
    difference() {
        // Wheel sphere
        sphere(r=wheel_radius);
        // Side sphere 1
        translate([0,side_spheres_radius + hub_thickness/2,0])
            sphere(r=side_spheres_radius);
        // Side sphere 2
        translate([0,- (side_spheres_radius + hub_thickness/2),0])
            sphere(r=side_spheres_radius);
        // Cylinder 1
        translate([wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 2
        translate([0,0,wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 3
        translate([-wheel_radius/2,0,0])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
        // Cylinder 4
        translate([0,0,-wheel_radius/2])
            rotate([90,0,0])
            cylinder(h=cylinder_height,r=cylinder_radius,center=true);
    }
}
```

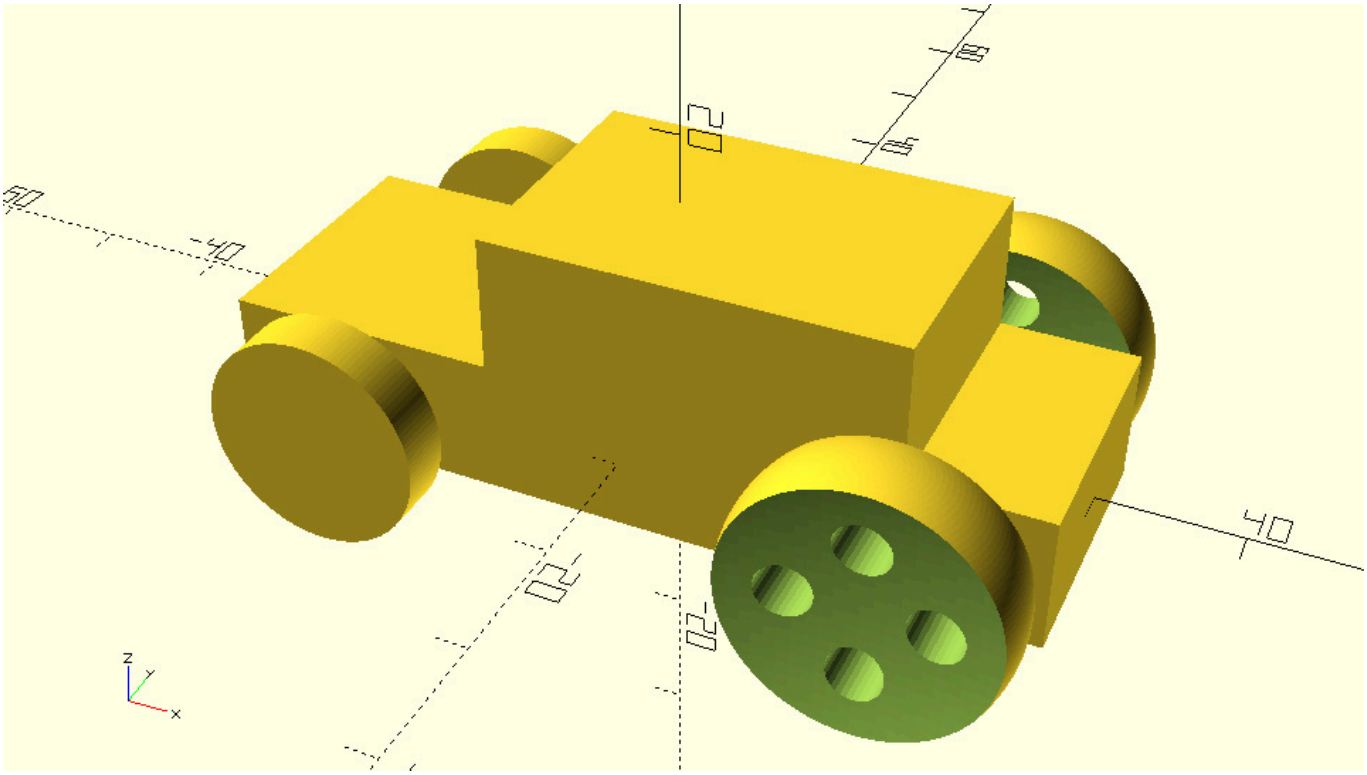
Exercise

Use the wheels.scad script in your car script. Use the simple_wheel module to create the front wheels and the complex_wheel module to create the rear wheels.

Code[Collapse]

car_with_different_wheels_from_used_modules.scad

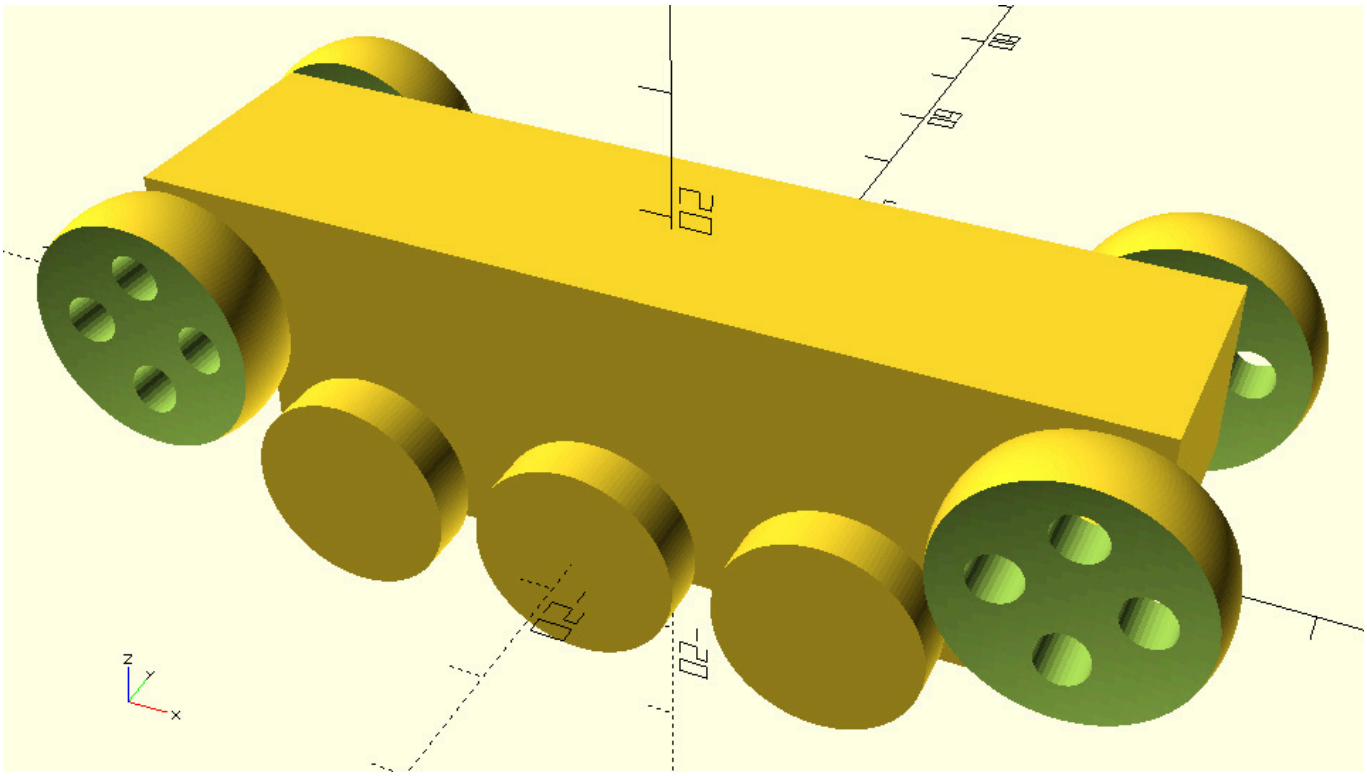
```
use <wheels.scad>
wheel_radius = 8;
wheel_width = 4;
base_height = 10;
top_height = 10;
track = 30;
// Car body base
cube([60,20,base_height],center=true);
// Car body top
translate([5,0,base_height/2+top_height/2 - 0.001])
    cube([30,20,top_height],center=true);
// Front left wheel
translate([-20,-track/2,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Front right wheel
translate([-20,track/2,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Rear left wheel
translate([20,-track/2,0])
    complex_wheel();
// Rear right wheel
translate([20,track/2,0])
    complex_wheel();
// Front axle
translate([-20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
// Rear axle
translate([20,0,0])
    rotate([90,0,0])
    cylinder(h=track,r=2,center=true);
```



With this example it should be clear that the name of the script doesn't have to be the same as the name of the module as well as that a script can contain multiple modules. There is no right or wrong way on how you should go about organizing your library of modules. In the last example a wheels.scad script which defines the different wheel modules was used. Alternatively you could have saved each module as a separate *.scad script.

Exercise

Create a vehicle_parts.scad script. Inside this script define a simple_wheel, complex_wheel, body and axle module. Use this script in another script named vehicle_concept to make the corresponding modules available. Use the modules to create a vehicle concept that looks similar to the following.



Code[Collapse]

car_with_ten_wheels.scad

```
use <vehicle_parts.scad>
$fa = 1;
$fs = 0.4;
wheel_radius = 8;
wheel_width = 4;
base_length = 60;
top_length = 80;
track = 30;
wheelbase_1 = 38;
wheelbase_2 = 72;
z_offset = 10;
body(base_length=base_length, top_length=top_length, top_offset=0);
// Front left wheel
translate([-wheelbase_2/2,-track/2,z_offset])
    complex_wheel();
// Front right wheel
translate([-wheelbase_2/2,track/2,z_offset])
    complex_wheel();
// Rear left wheel
translate([wheelbase_2/2,-track/2,z_offset])
    complex_wheel();
// Rear right wheel
translate([wheelbase_2/2,track/2,z_offset])
    complex_wheel();
// Front axle
translate([-wheelbase_2/2,0,z_offset])
    axle(track=track);
// Rear axle
translate([wheelbase_2/2,0,z_offset])
    axle(track=track);

// Middle front left wheel
translate([-wheelbase_1/2,-track/2,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Middle front right wheel
translate([-wheelbase_1/2,track/2,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Middle left wheel
translate([0,-track/2,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Middle right wheel
translate([0,track/2,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Middle rear left wheel
translate([wheelbase_1/2,-track/2,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Middle rear right wheel
translate([wheelbase_1/2,track/2,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Middle front axle
translate([-wheelbase_1/2,0,0])
    axle(track=track);
// Middle axle
translate([0,0,0])
    axle(track=track);
// Middle rear axle
translate([wheelbase_1/2,0,0])
    axle(track=track);
```

Using the MCAD library

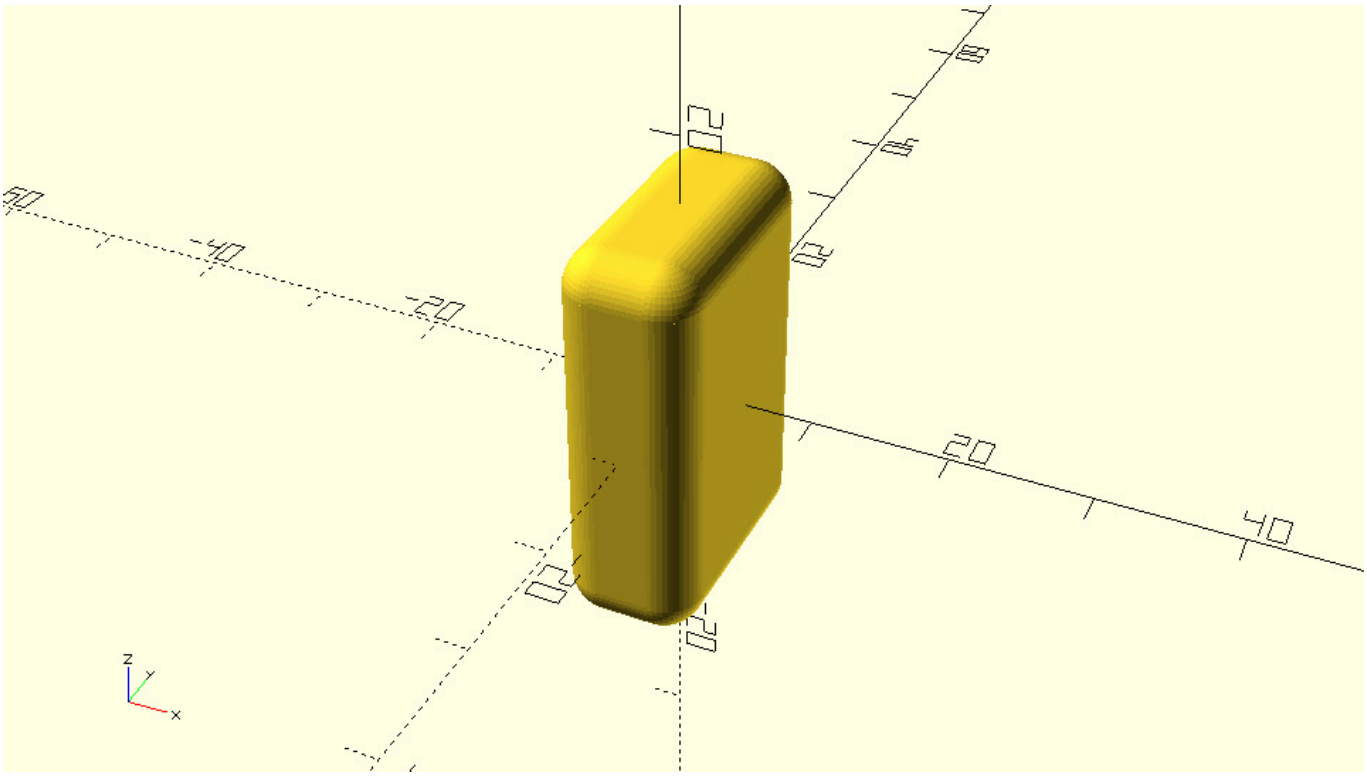
The MCAD library (https://github.com/openscad/MCAD) is a library of components commonly used in mechanical designs that comes with OpenSCAD. You can utilize objects of the MCAD library by using the corresponding OpenSCAD script and calling the desired modules. For example, there is a boxes.scad script which contains the model of a rounded box. The boxes.scad

script contains one module, which can be used to create the corresponding box. You can open this script to check what the parameters of this module are and then use it to add rounded boxes in your design. You can create a fully rounded box with side lengths of 10, 20 and 30 units as well as fillet radius of 3 units using the following script.

Code

completely_rounded_box.scad

```
use <MCAD/boxes.scad>
$fa=1;
$fs=0.4;
roundedBox(size=[10,20,30],radius=3,sidesonly=false);
```

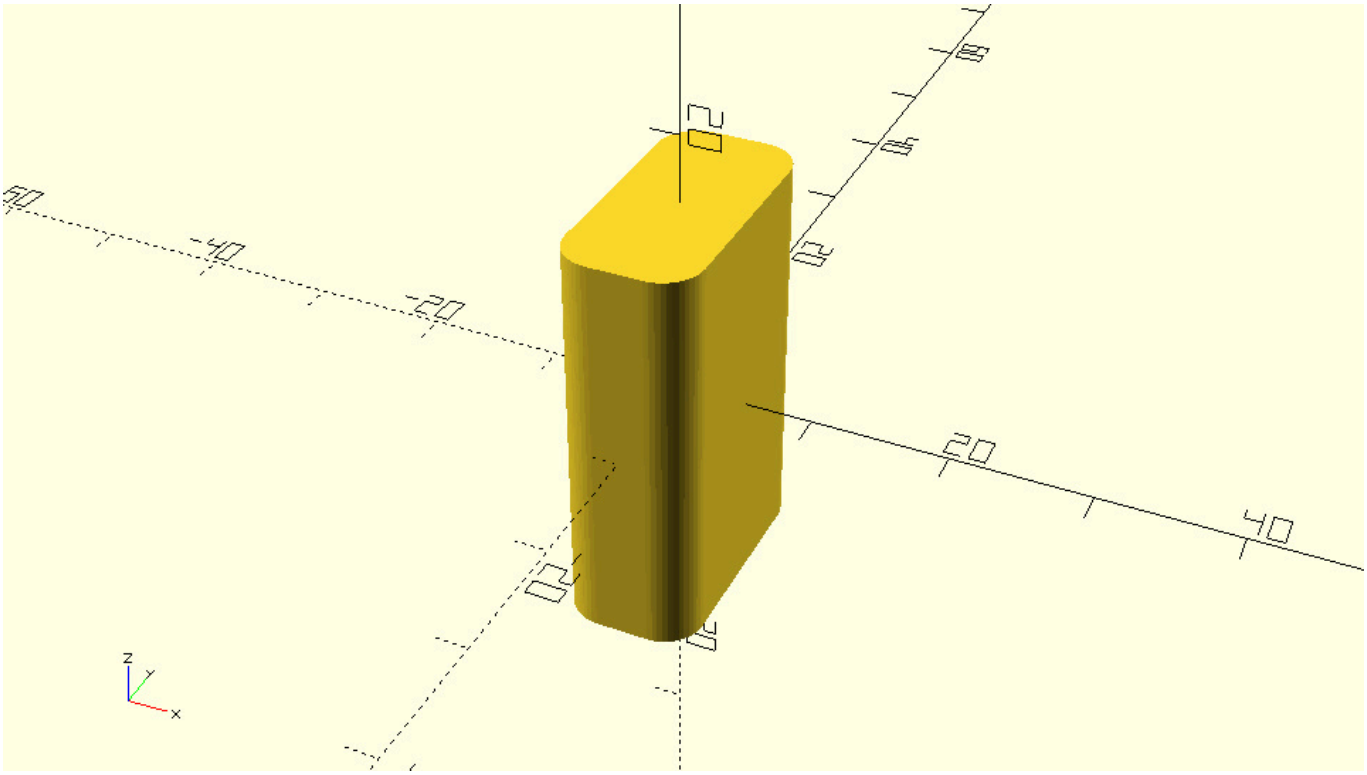


By setting the sidesonly parameter equal to true you can create a box of similar dimension that has only 4 rounded sides.

Code

sides_only_rounded_box.scad

```
use <MCAD/boxes.scad>
$fa=1;
$fs=0.4;
roundedBox(size=[10,20,30],radius=3,sidesonly=true);
```



The boxes.scad script is located in the MCAD directory which is under the libraries directory. The latter can be found in OpenSCAD's installation folder. Should you wish to have any of your own libraries accessible from any directory, you can add it in the libraries directory. You can also browse other available OpenSCAD libraries at https://www.openscad.org/libraries.html. Though, you should be aware that there are a very wide number of libraries available on GitHub and Thingiverse that far exceed those linked at OpenSCAD's libraries page.

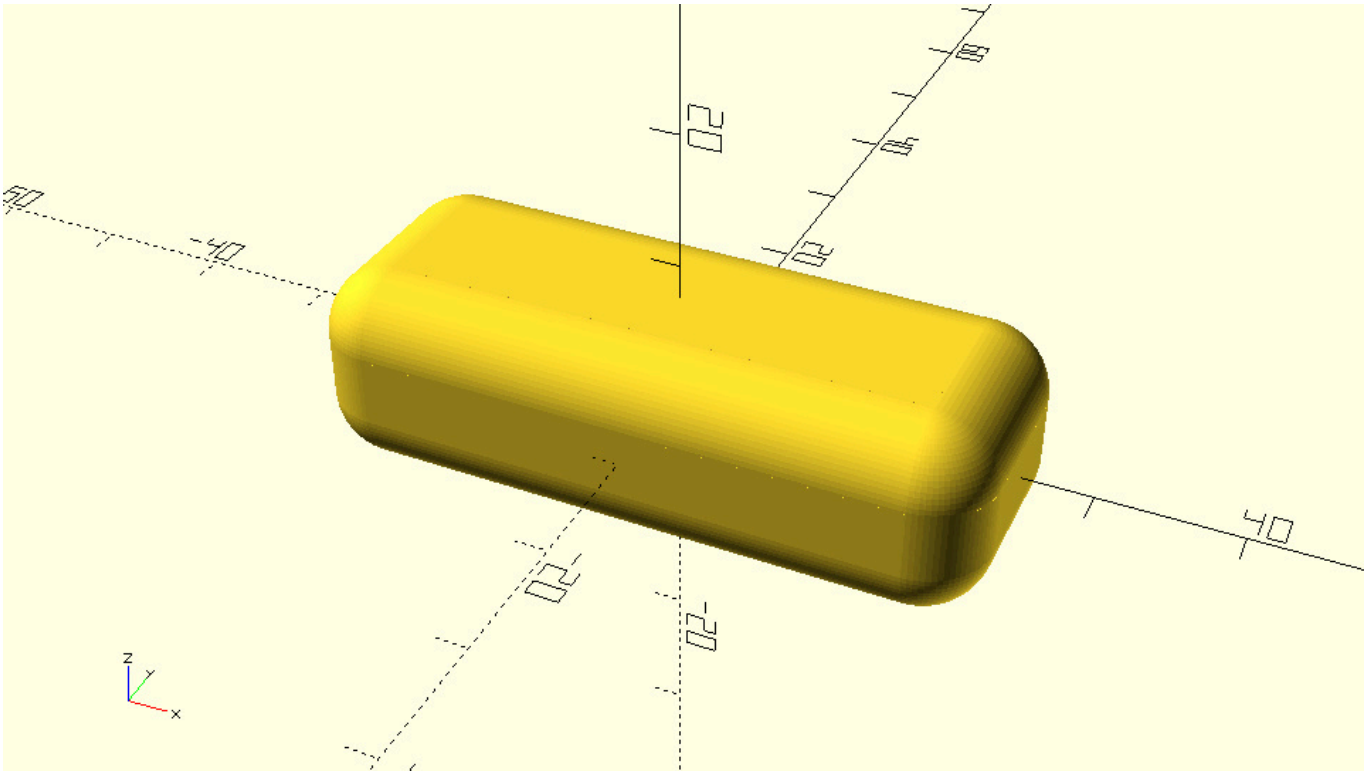
Exercise

Use the boxes.scad script of the MCAD library to create a fully rounded box with side lengths of 50, 20 and 15 units as well as fillet radius of 5 units.

Code[Collapse]

horizontal_completely_rounded_box.scad

```
use <MCAD/boxes.scad>
$fa=1;
$fs=0.4;
roundedBox(size=[50,20,15],radius=5,sidesonly=false);
```



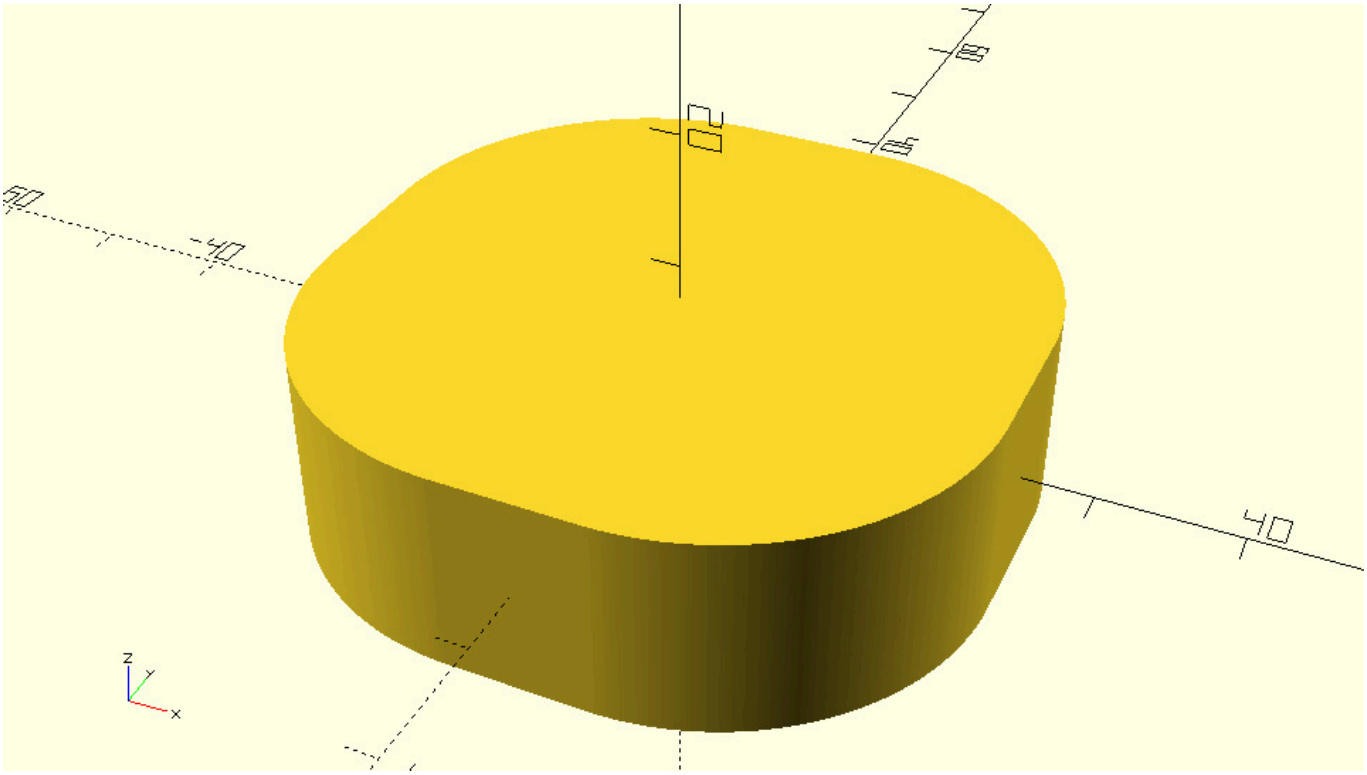
Exercise

Use the boxes.scad script of the MCAD library to create a rounded box with only 4 rounded sides with side lengths of 50, 50 and 15 units as well as fillet radius of 20 units.

Code[Collapse]

short_sides_only_rounded_box.scad

```
use <MCAD/boxes.scad>
$fa=1;
$fs=0.4;
roundedBox(size=[50,50,15],radius=20,sidesonly=true);
```



Creating even more parameterizable modules

So far, the only input to the modules that have been created was through the module's input parameters that were defined for each case. The complex_wheel module for example was able to create a plethora of parameterized wheels according to the chosen input parameters such as wheel_radius, hub_thickness etc.

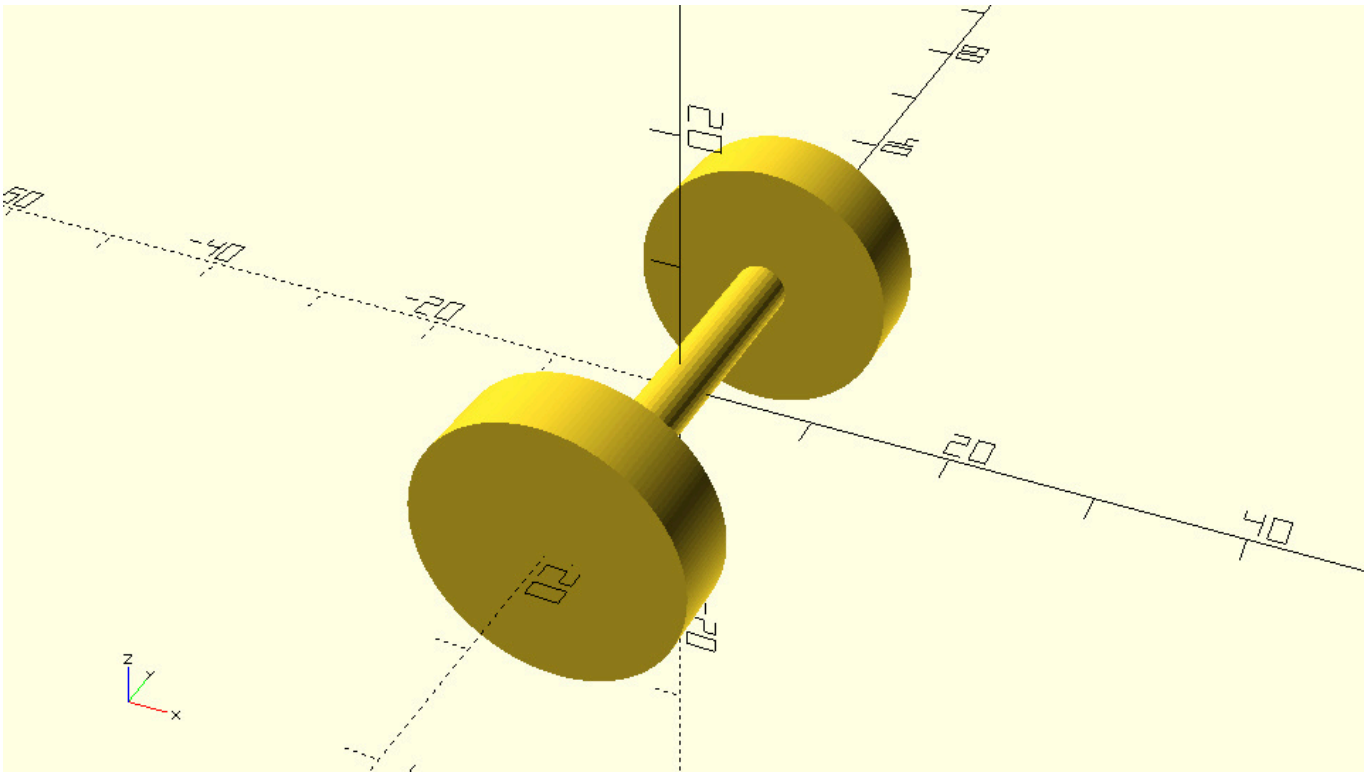
In your vehicle designs you have been using body, wheel and axle modules which when combined can produce various types of vehicles. In all the vehicle designs, two wheels along with an axle have been used together to form a set of wheels. You may have considered the need for an axle_wheelset module to simultaneously define all three objects with a single statement. And you would have been right to consider it! But there is a reason this module hasn't been created yet, and you are now going to find out why.

Throughout the previous chapters you have created two different wheel designs (simple_wheel and complex_wheel) and a single axle design. You can use your existing knowledge to combine the simple_wheel and axle modules in the following way.

Code

axle_with_simple_wheelset_from_module.scad

```
use <vehicle_parts.scad>
$fa = 1;
$fs = 0.4;
module axle_wheelset(wheel_radius=10, wheel_width=6, track=35, radius=2) {
    translate([0,track/2,0])
        simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
    axle(track=track, radius=radius);
    translate([0,-track/2,0])
        simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
}
axle_wheelset();
```

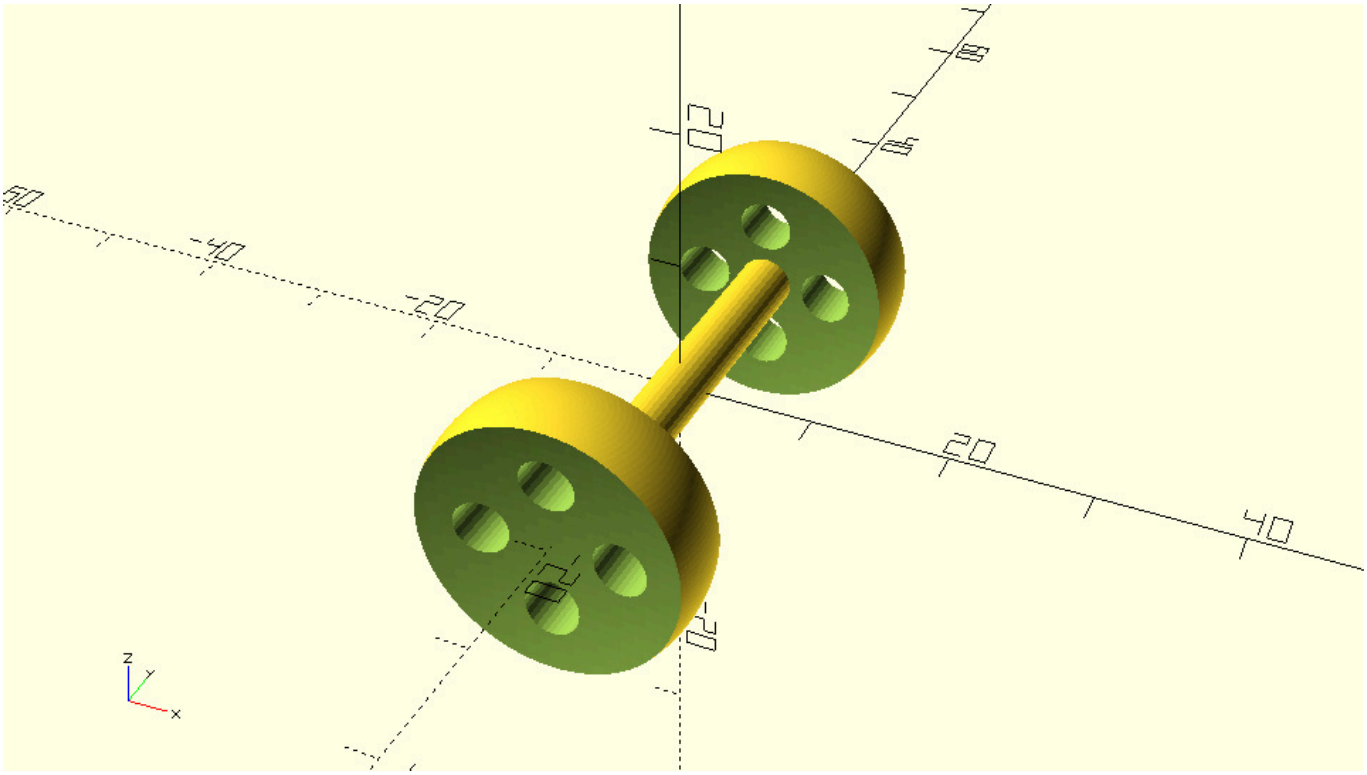


If you simply wanted to only use the above set of simple_wheels, that approach would be just fine. The problem though is that this axle_wheelset module is not really flexible and parameterizable to the desired degree. On one hand you can customize all input parameters, but on the other hand, could you swap the simple_wheel design with the complex one? The fact is that with the above approach in order to do so you would have to define a completely new module.

Code

axle_with_complex_wheelset_from_module.scad

```
use <vehicle_parts.scad>
$fa = 1;
$fs = 0.4;
module axle_wheelset_complex(wheel_radius=10, side_spheres_radius=50, hub_thickness=4,
cylinder_radius=2, track=35, radius=2) {
    translate([0,track/2,0])
        complex_wheel(wheel_radius=10, side_spheres_radius=50, hub_thickness=4, cylinder_radius=2);
    axle(track=track, radius=radius);
    translate([0,-track/2,0])
        complex_wheel(wheel_radius=10, side_spheres_radius=50, hub_thickness=4, cylinder_radius=2);
}
axle_wheelset_complex();
```



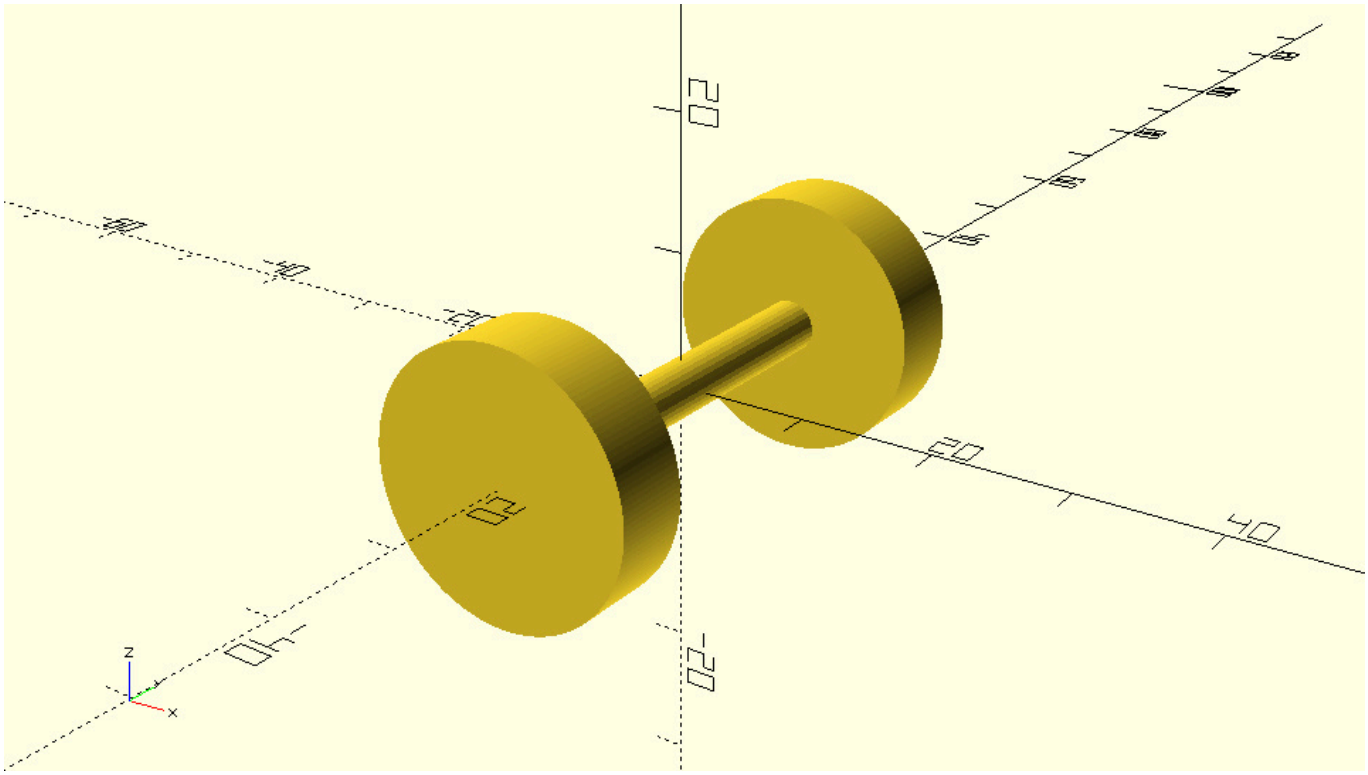
If you can't see yet how this is a problem, imagine the case where you had six different wheel designs and two different axle designs in your library. If you wanted to implement the axle_wheelset module you would need to define 12 different modules to cover all combinations of wheel and axle designs. Furthermore, if you were to add a new wheel or axle design in your collection, you would need to define a number of additional axle_wheelset modules which would make maintaining your library very hard.

The good thing is that the two modules above look very similar. If you could keep the structure of the module the same but have the specific choice of wheel design parameterized, then the problem could be solved. Fortunately, OpenSCAD supports this functionality and parameterizing the specific choice of wheel design can be achieved in the following way.

Code

axle_with_simple_wheelset_from_parameterized_module.scad

```
use <vehicle_parts.scad>
$fa = 1;
$fs = 0.4;
module axle_wheelset(track=35, radius=2) {
    translate([0,track/2,0])
        children(0);
    axle(track=track, radius=radius);
    translate([0,-track/2,0])
        children(0);
}
axle_wheelset() {
    simple_wheel();
}
```

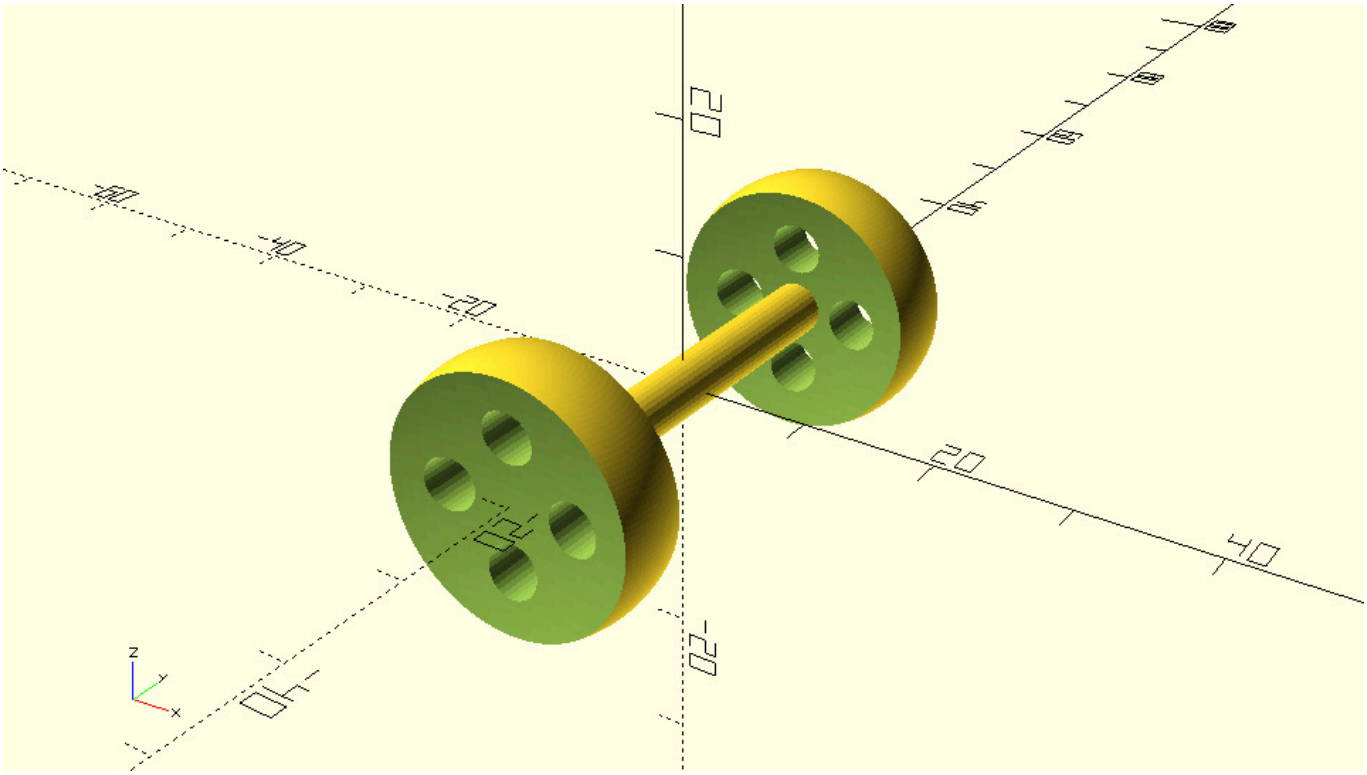


The wheel design can now be effortlessly changed, making this module a truly parametric one.

Code

axle_with_complex_wheelset_from_parameterized_module.scad

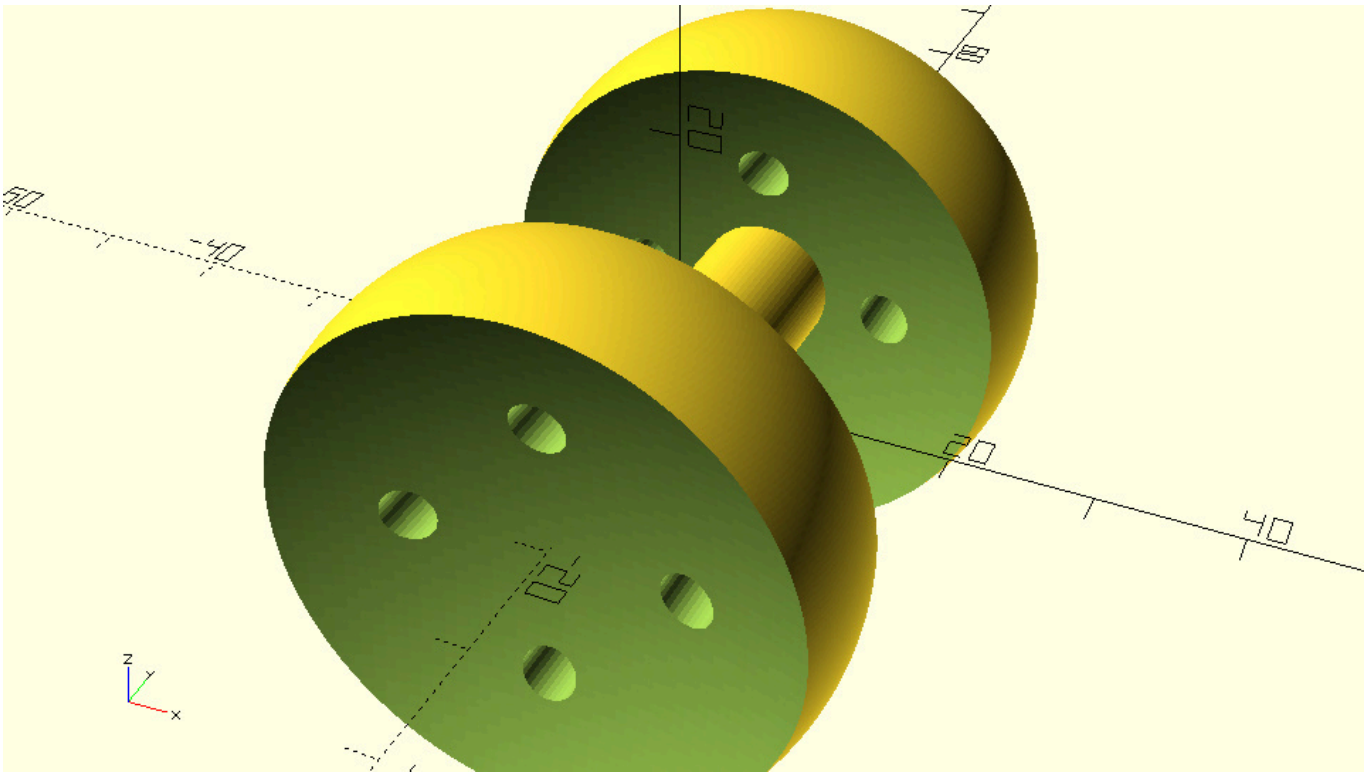
```
axle_wheelset() {
    complex_wheel();
}
```



Code

axle_with_large_complex_wheelset_from_parameterized_module.scad

```
axle_wheelset(radius=5) {
    complex_wheel(wheel_radius=20);
}
```



There is a very important concept that you should grasp here. The first thing you should notice is the definition of this new module. This new module is similar to the previous ones with the difference that the command children(0) is used in place of a call to a specific wheel module. The

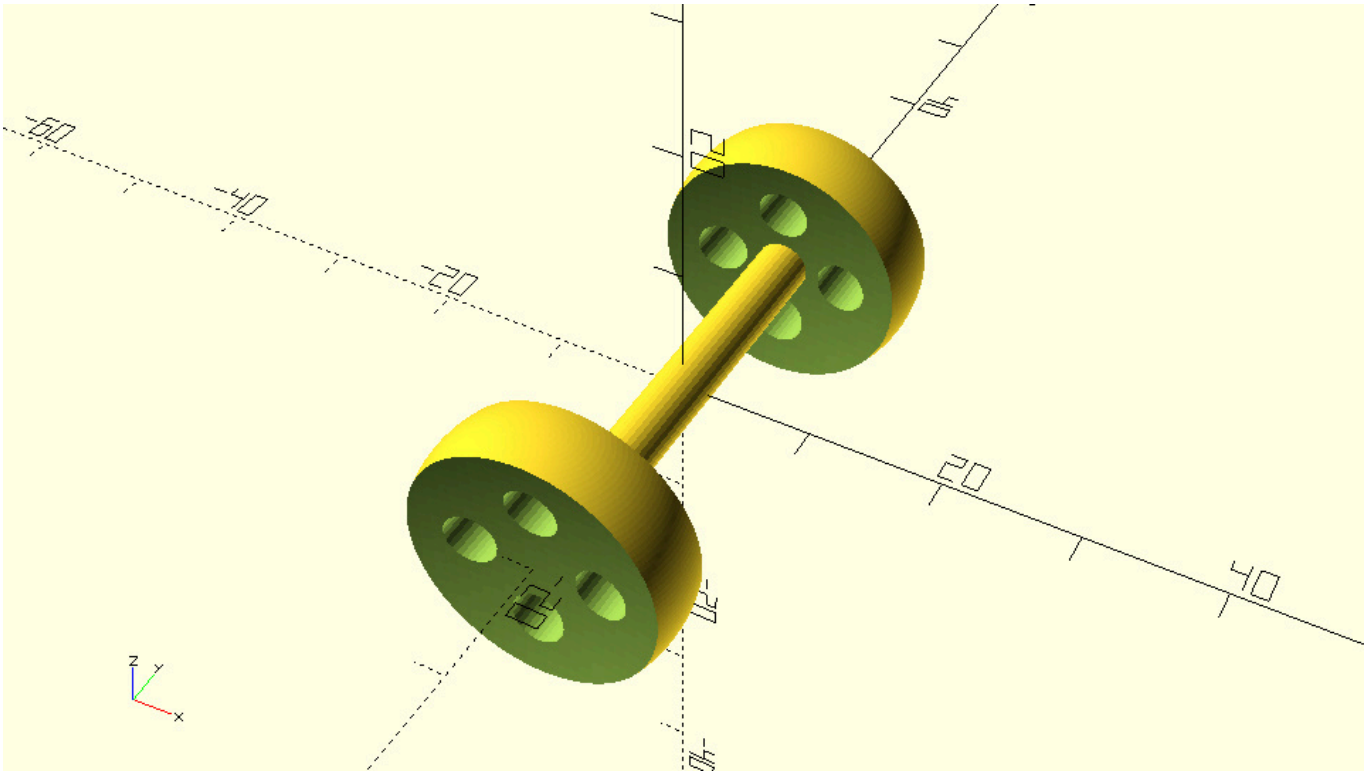
second thing you should notice is the call to the axle_wheelset module. The call to the axle_wheelset module contains a pair of curly brackets inside of which the specific wheel design to be used by module is defined each time. OpenSCAD keeps an ordered list of the objects that have been defined inside the curly brackets and numbers them starting from zero. These objects can then be referenced by the children command. The number that is passed inside the children command corresponds to the first, second, third etc. object that was defined inside the curly brackets, counting from zero. In the above example, only one object is defined inside the curly brackets. That is either a simple_wheel or a complex wheel object. This object is created every time the children(0) command is used. The children command is in essence a way to pass objects as input to a module.

The next examples can help make this concept more concrete. In the previous example there is no way to use the axle_wheelset module and end up creating an axle that has different wheels on each side. This would not happen even if you passed/defined two different objects inside the curly brackets when calling the axle_wheelset module, because only the first one, children(0), is referenced for both sides of the axle.

Code

axle_with_same_wheels_from_module.scad

```
axle_wheelset() {
    complex_wheel();
    simple_wheel();
}
```



In order to create an axle with different wheels on each side, the definition of the of the axle_wheelset module would need to be modified. Instead of referencing the first object, children(0), for both sides, the axle_wheelset module would need to reference the first object, children(0), for one side and the second object, children(1), for the second side.

Code

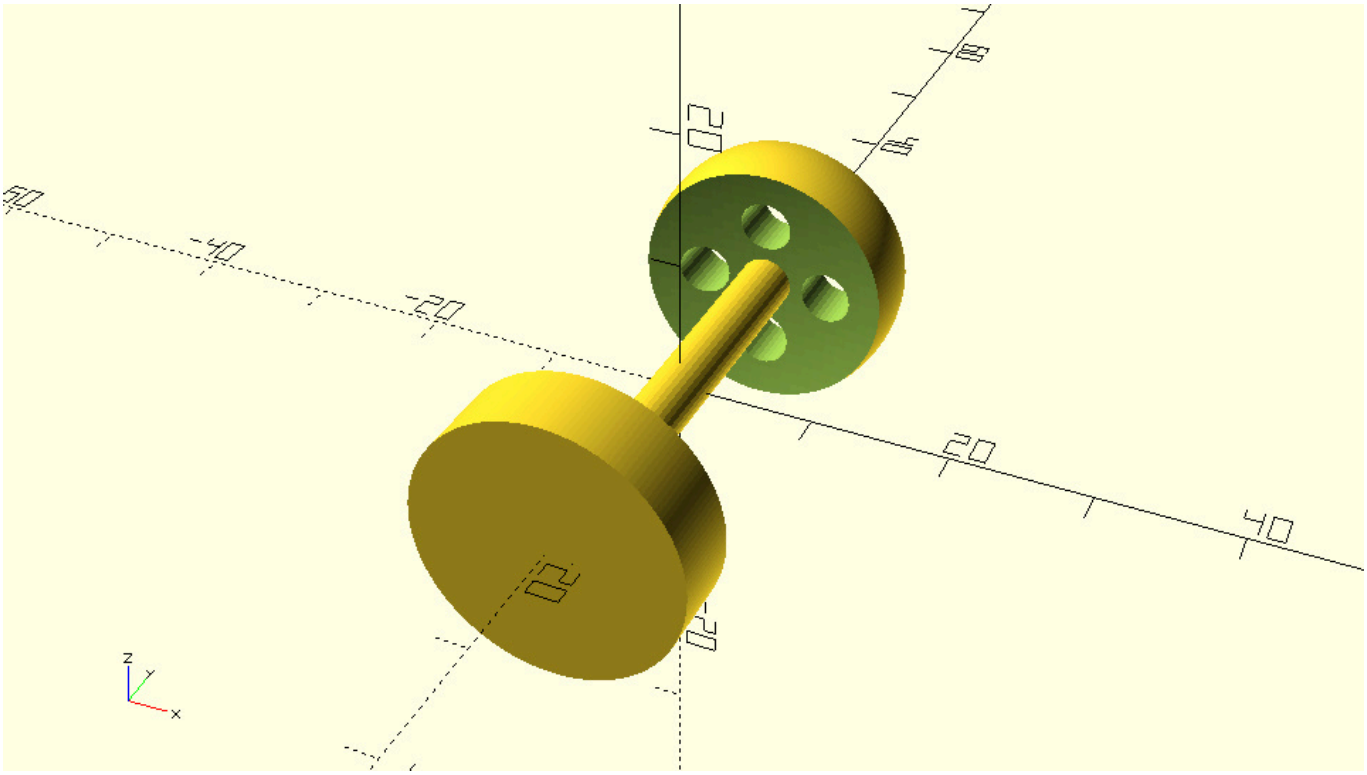
```
module axle_wheelset(track=35, radius=2) {
    translate([0,track/2,0])
        children(0);
    axle(track=track, radius=radius);
    translate([0,-track/2,0])
        children(1);
}
```

By defining two different wheel objects inside the curly brackets, the following model would be created.

Code

axle_with_different_wheels_from_parameterized_module.scad

```
axle_wheelset() {
    complex_wheel();
    simple_wheel();
}
```



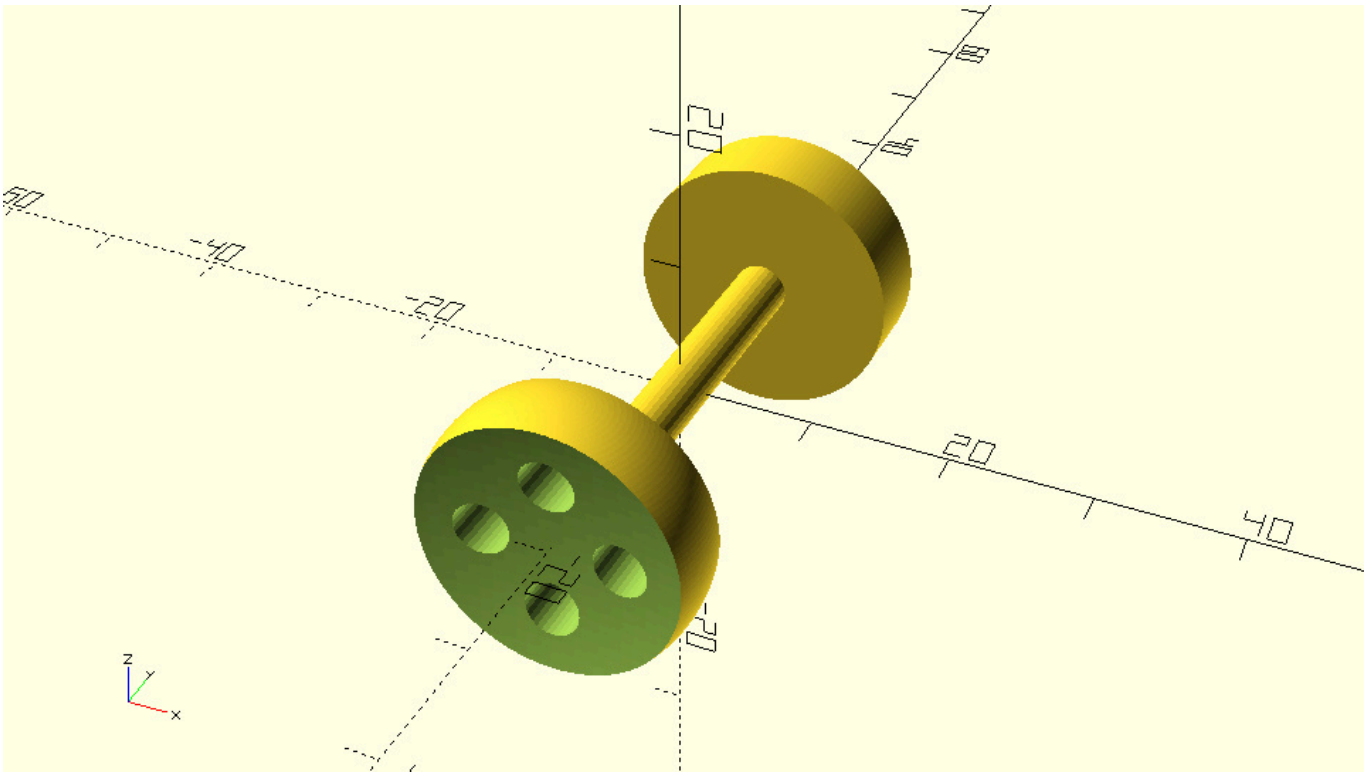
Exercise

Try swapping the order in which the wheels are defined inside the curly brackets when calling the axle_wheelset module. What happens?

Code[Collapse]

axle_with_flipped_different_wheels_from_parameterized_module.scad

```
axle_wheelset() {
    simple_wheel();
    complex_wheel();
}
```



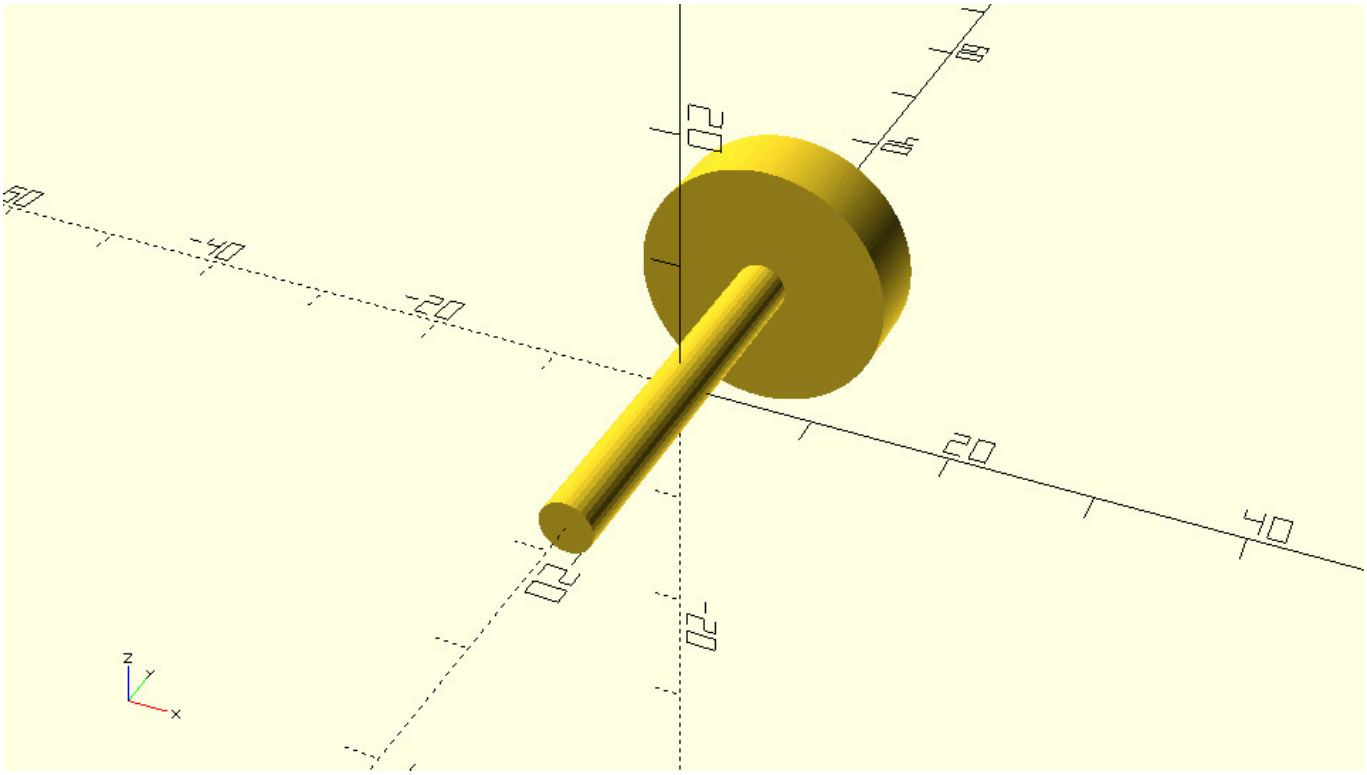
Exercise

Try defining only one wheel inside the curly brackets? Do you get an error message?

Code[Collapse]

axle_with_missing_wheel_from_parameterized_module.scad

```
axle_wheelset() {
    simple_wheel();
}
```



Exercise

Add an axle_wheel module on the vehicle_parts.scad script. Make use of the children command to parameterize the specific choice of wheel design. Use the vehicle_parts.scad script on another script to create any vehicle design that you like.

Challenge

The material you have been learning in the last two chapters gives you a powerful set of tools to start creating your own library of objects that can be flexibly combined and customized to create new designs.

Exercise

Think of any model that you would like to create. Break it down into different parts. Come up with alternative designs for each part and define modules that create them. What should the input parameters of each module be? Define one or more modules using the children functionality in order to flexibly combine the various parts that you have created.

Chapter 6

OpenSCAD variables

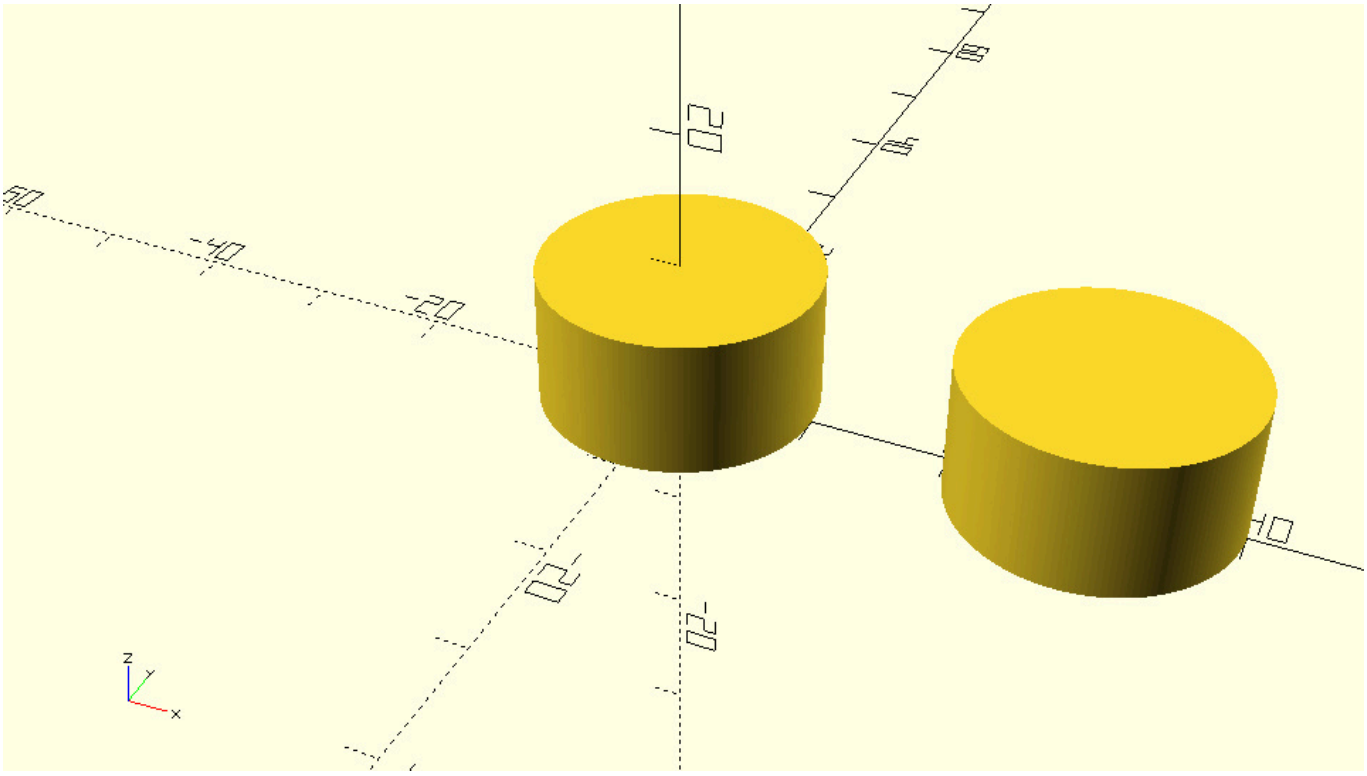
In the previous chapters you have made use of variables to parameterize your designs and make them easily customizable. Specifically, you have been assigning them numerical values at some part of your script and then using their stored value in some other part. For example, you can set a wheel_radius variable equal to the desired wheel radius and use that variable in the corresponding statements that create the wheels of your car. This way you can easily customize the radius of your car's wheels without having to search for and change multiple values, but only by directly changing the value of the wheel_radius variable.

You also learned about an important property of OpenSCAD variables. This is that a variable can only have one specific value. If you assign one value to a variable and then assign it a different value at a later part of the script, your variable will have only the final value throughout the execution of your design. This is demonstrated on the following example.

Code

two_cylinder_with_same_radius.scad

```
$fa=1;
$fs=0.4;
height=10;
radius=5;
cylinder(h=height,r=radius);
radius=10;
translate([30,0,0])
    cylinder(h=height,r=radius);
```



Both cylinders have a radius of 10 units, which is the last value that is assigned to the radius variable.

When variables store numerical values, they can be used to specify dimensions of different objects or define transformation commands. Numerical values aren't the only kind of values that can be assigned to a variable. Variables can also hold boolean values (true or false) as well as characters (a, b, c, d, ...). As you are going to see on the following topics, by using boolean or character variables you can further parameterize your models and modules.

Conditional variable assignment

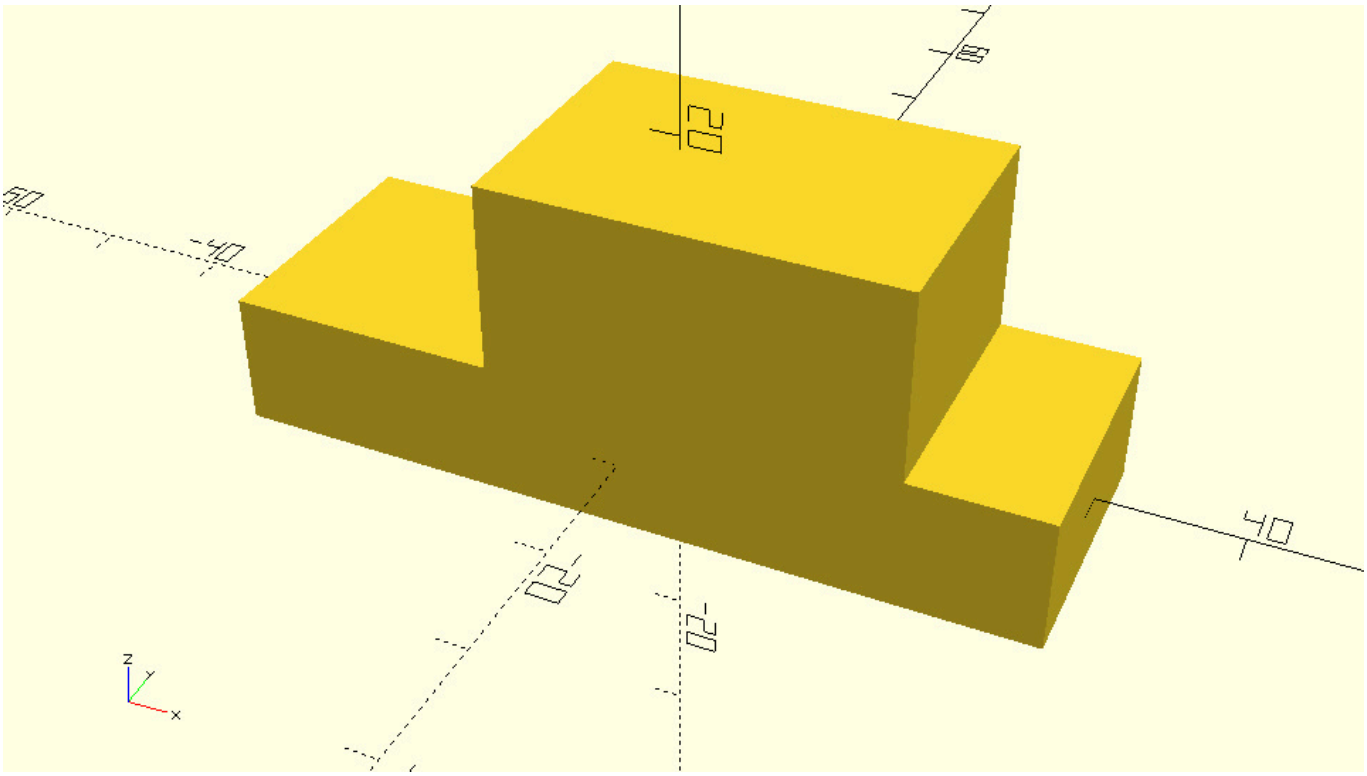
So far you have been assigning specific values to variables using appropriate assignment commands. There are cases, though, where you would prefer the assignment itself to be parametric and dependent on some aspect of your design.

The creation of a car's body requires the definition of various parameters. These parameters can be defined when calling the body module by using corresponding variables that have been defined in your script. One example of this is the following.

Code

parameterized_car_body.scad

```
use <vehicle_parts.scad>
$fa=1;
$fs=0.4;
base_length = 60;
top_length = 30;
top_offset = 5;
body(base_length=base_length, top_length=top_length, top_offset=top_offset);
```

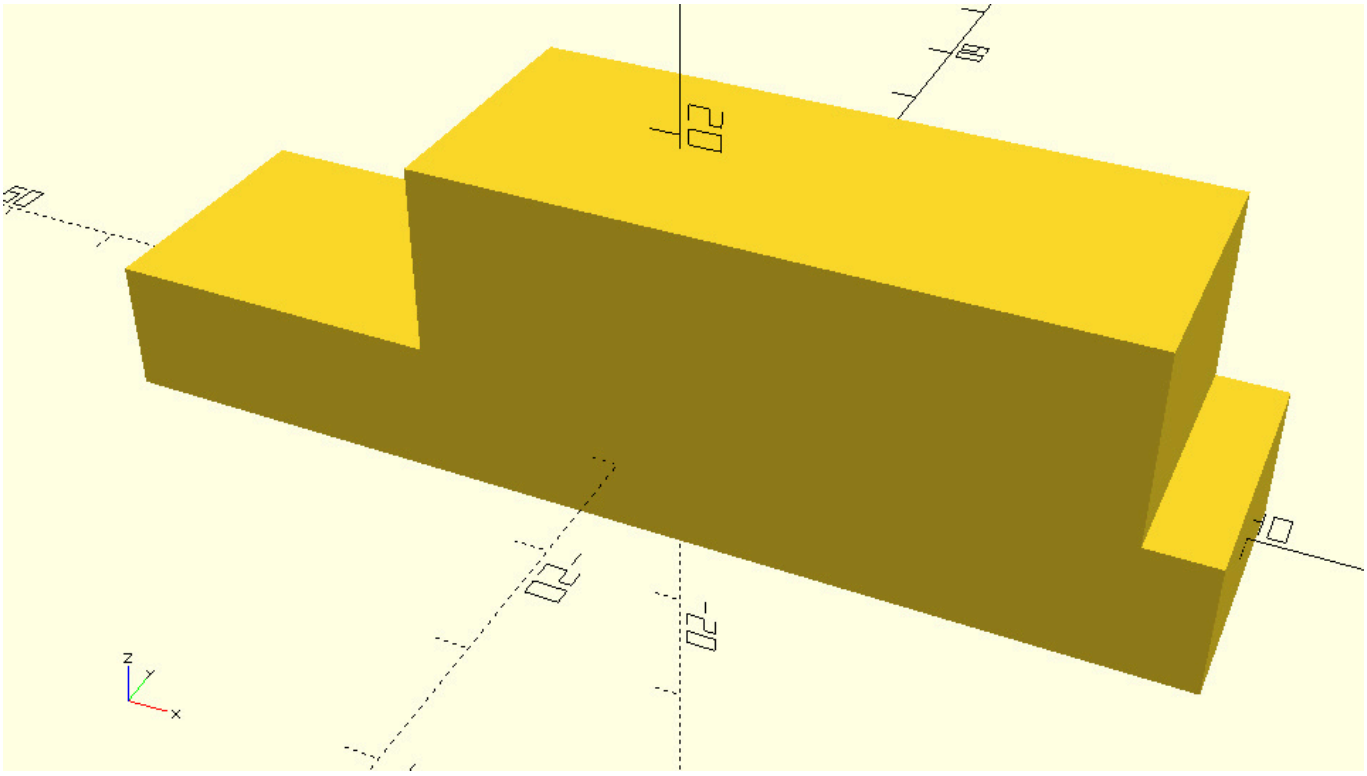


The above version of the car's body will be called the short version. By choosing different values for the variables a long version can also be created.

Code

long_car_body.scad

```
use <vehicle_parts.scad>
$fa=1;
$fs=0.4;
base_length = 80;
top_length = 50;
top_offset = 10;
body(base_length=base_length, top_length=top_length, top_offset=top_offset);
```



What if these two versions of the car's body are the only versions that you currently interested in? Is there a way to quickly switch between these two versions without having to modify each variable separately?

You may think modifying three variables isn't much work, but the number of required variables on more complex models can easily get unmanageable. Luckily there is a solution to this problem, which is the conditional assignment of variables. The conditional assignment of variables is a way to instruct OpenSCAD to assign different values to variables depending on whether some condition is true or false. In this case the condition is whether the car's body should be long or not. You can represent this condition by defining a long_body variable and setting it equal to true if you want the body to be long or equal to false if you don't want the body to be long.

The choice of a long body is represented by the following statement.

Code

```
long_body = true;
```

Respectively the choice of a short body is represented by the following statement.

Code

```
long_body = false;
```

The long_body variable is called a boolean variable because boolean values (true or false) are assigned to it. The next step is the definition of the conditional assignments which will assign the appropriate values to base_length, top_length and top_offset variables depending on the value of the long_body variable. These conditional assignments can be defined in the following manner.

Code

```
base_length = (long_body) ? 80:60;
top_length = (long_body) ? 50:30;
top_offset = (long_body) ? 10:5;
```

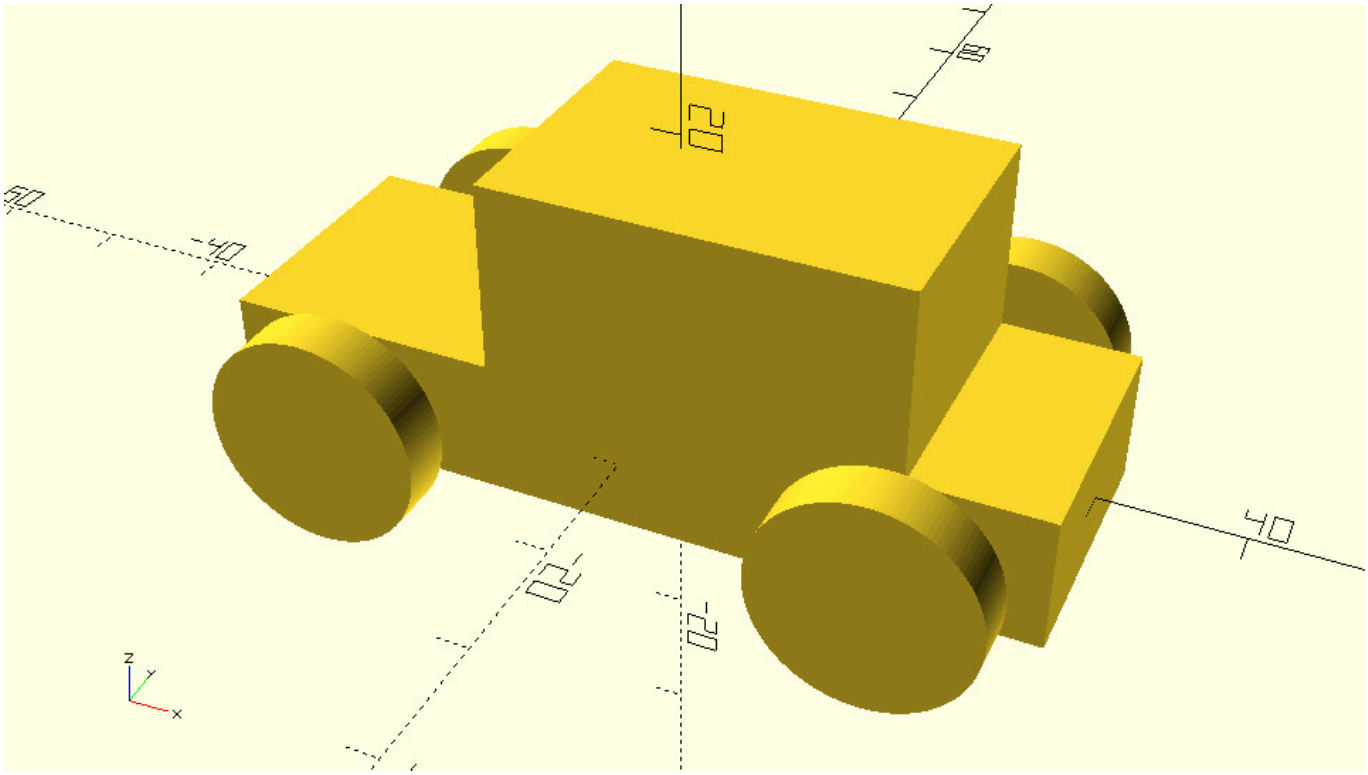
You should notice the following points about the definition of a conditional assignment. First the name of variable is typed out followed by the equal sign. Then follows a pair of parentheses that contains the condition which will be used in the conditional assignment. The condition in this case is a boolean variable. In general, the condition can also be a combination of logical and comparison operations between multiple variables. After the closing parenthesis follow a question mark and the two corresponding variable values that are separated by a colon. If the supplied condition is true, the first value will be the one assigned to the variable. If the supplied condition is false, the second values will be the one assigned to the variable.

By incorporating the above conditional assignments in your script, you can switch between a short and a long car body just by changing the long_body variable from false to true and vice versa.

Code

car_with_normal_conditional_body.scad

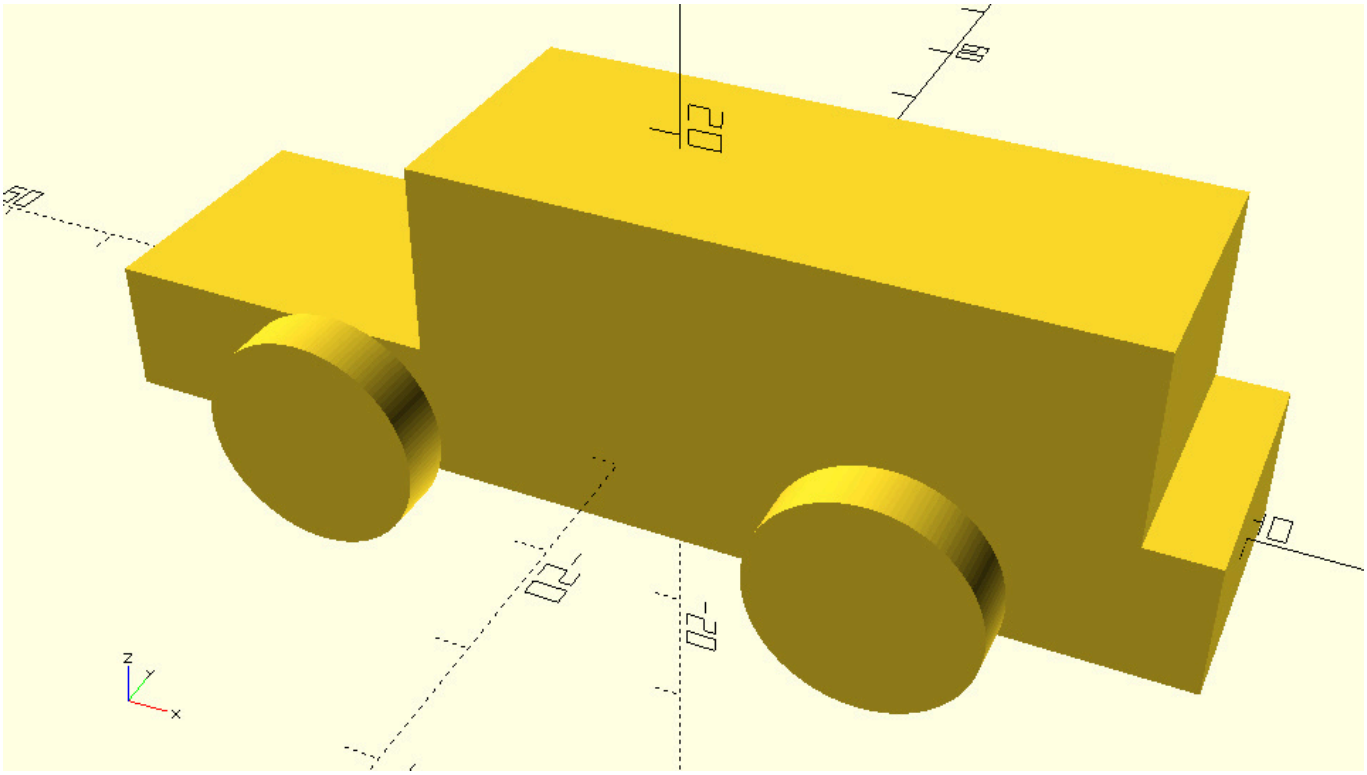
```
use <vehicle_parts.scad>
$fa=1;
$fs=0.4;

// Conditional assignment of body variables
long_body = false;
base_length = (long_body) ? 80:60;
top_length = (long_body) ? 50:30;
top_offset = (long_body) ? 10:5;
// Creation of body
body(base_length=base_length, top_length=top_length, top_offset=top_offset);

// Creation of wheels and axles
track = 30;
wheelbase = 40;
wheel_radius = 8;
wheel_width = 4;
// Front left wheel
translate([-wheelbase/2,-track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
 // Front right wheel
translate([-wheelbase/2,track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Rear left wheel
translate([wheelbase/2,-track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Rear right wheel
translate([wheelbase/2,track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Front axle
translate([-wheelbase/2,0,0])
    axle(track=track);
// Rear axle
translate([wheelbase/2,0,0])
    axle(track=track);
```



Code

car_with_long_conditional_body.scad

```
use <vehicle_parts.scad>
$fa=1;
$fs=0.4;

// Conditional assignment of body variables
long_body = true;
base_length = (long_body) ? 80:60;
top_length = (long_body) ? 50:30;
top_offset = (long_body) ? 10:5;
// Creation of body
body(base_length=base_length, top_length=top_length, top_offset=top_offset);

// Creation of wheels and axles
track = 30;
wheelbase = 40;
wheel_radius = 8;
wheel_width = 4;
// Front left wheel
translate([-wheelbase/2,-track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
 // Front right wheel
translate([-wheelbase/2,track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Rear left wheel
translate([wheelbase/2,-track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Rear right wheel
translate([wheelbase/2,track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Front axle
translate([-wheelbase/2,0,0])
    axle(track=track);
// Rear axle
translate([wheelbase/2,0,0])
    axle(track=track);
```



Exercise

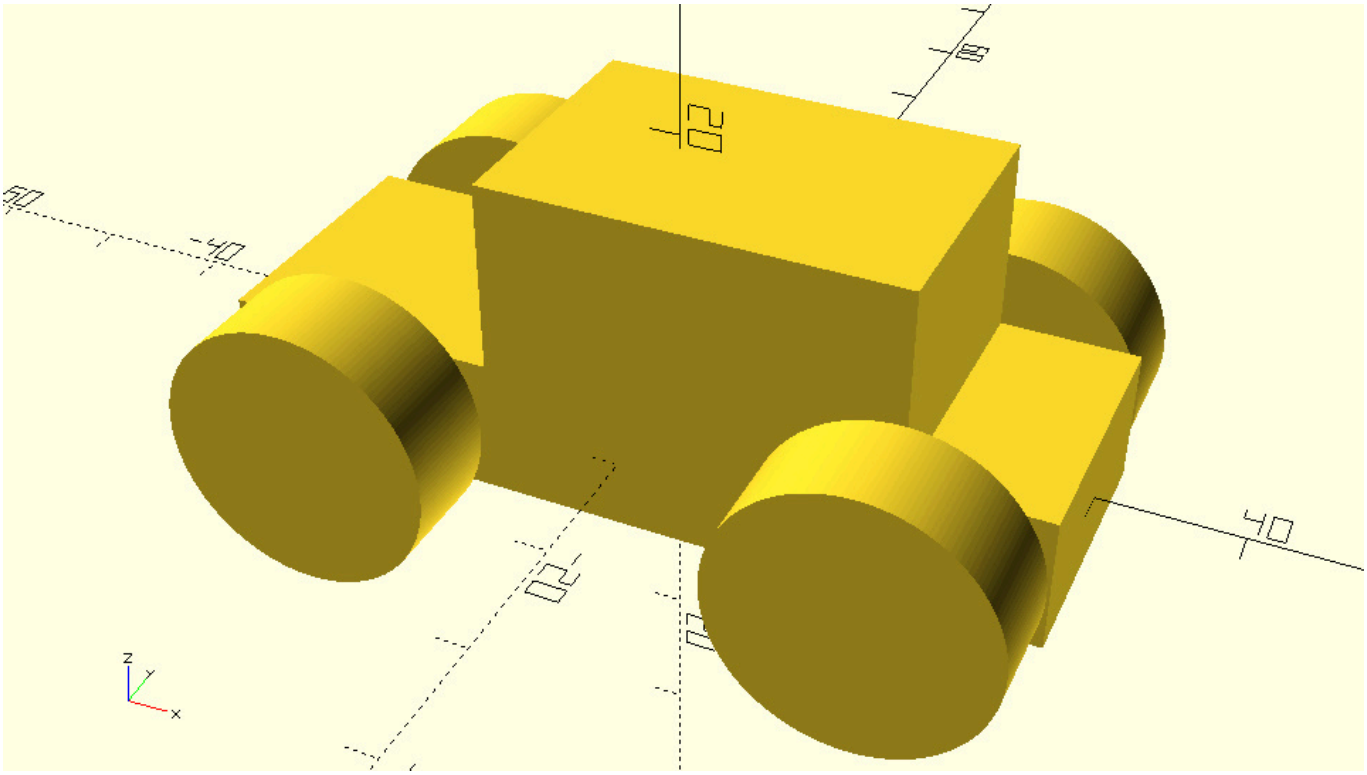
Add a large_wheels variable to the previous example. The variable should only take boolean values. Add two conditional assignments that assign different values to the wheel_radius and wheel_width variables. The large_wheels variable should be used as the condition for both assignments. If the large_wheels variable is false, the wheel_radius and wheel_width variables should be set equal to 8 and 4 units respectively. If the large_wheels variable is true, the wheel_radius and wheel_width variables should be set equal to 10 and 8 units respectively. Set appropriate values to the long_body and large_wheels variables to create the following versions of the car: short body - large wheels, short body - small wheels, long body - large wheels, long body - small wheels.

- short body - large wheels

Code[Collapse]

car_with_short_body_and_large_wheels.scad

```
use <vehicle_parts.scad>
$fa=1;
$fs=0.4;

// Conditional assignment of body variables
long_body = false;
base_length = (long_body) ? 80:60;
top_length = (long_body) ? 50:30;
top_offset = (long_body) ? 10:5;
// Creation of body
body(base_length=base_length, top_length=top_length, top_offset=top_offset);

// Creation of wheels and axles
large_wheels = true;
wheel_radius = (large_wheels) ? 10:6;
wheel_width = (large_wheels) ? 8:4;
track = 30;
wheelbase = 40;
// Front left wheel
translate([-wheelbase/2,-track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
 // Front right wheel
translate([-wheelbase/2,track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Rear left wheel
translate([wheelbase/2,-track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Rear right wheel
translate([wheelbase/2,track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Front axle
translate([-wheelbase/2,0,0])
    axle(track=track);
// Rear axle
translate([wheelbase/2,0,0])
    axle(track=track);
```

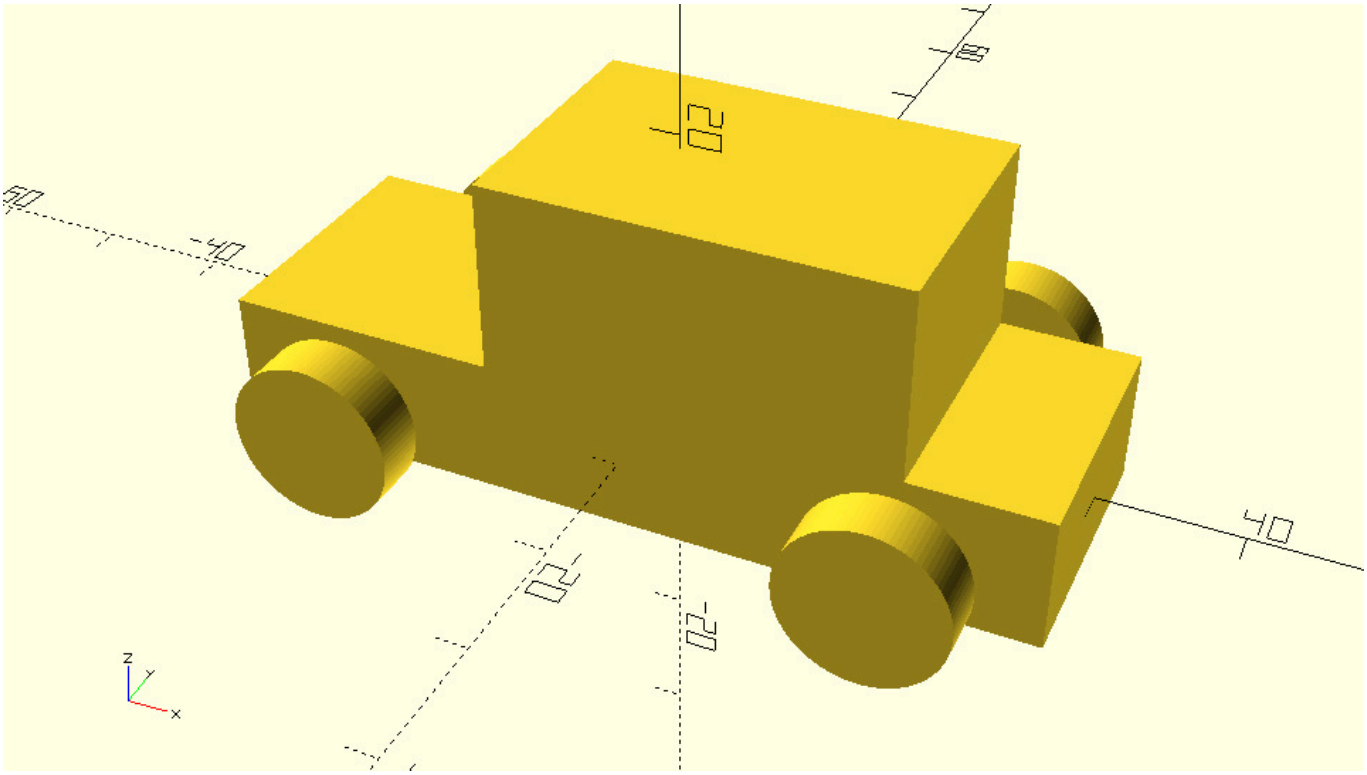


- short body - small wheels

Code[Collapse]

car_with_short_body_and_small_wheels.scad

```
…
long_body = false;
large_wheels = false;
…
```

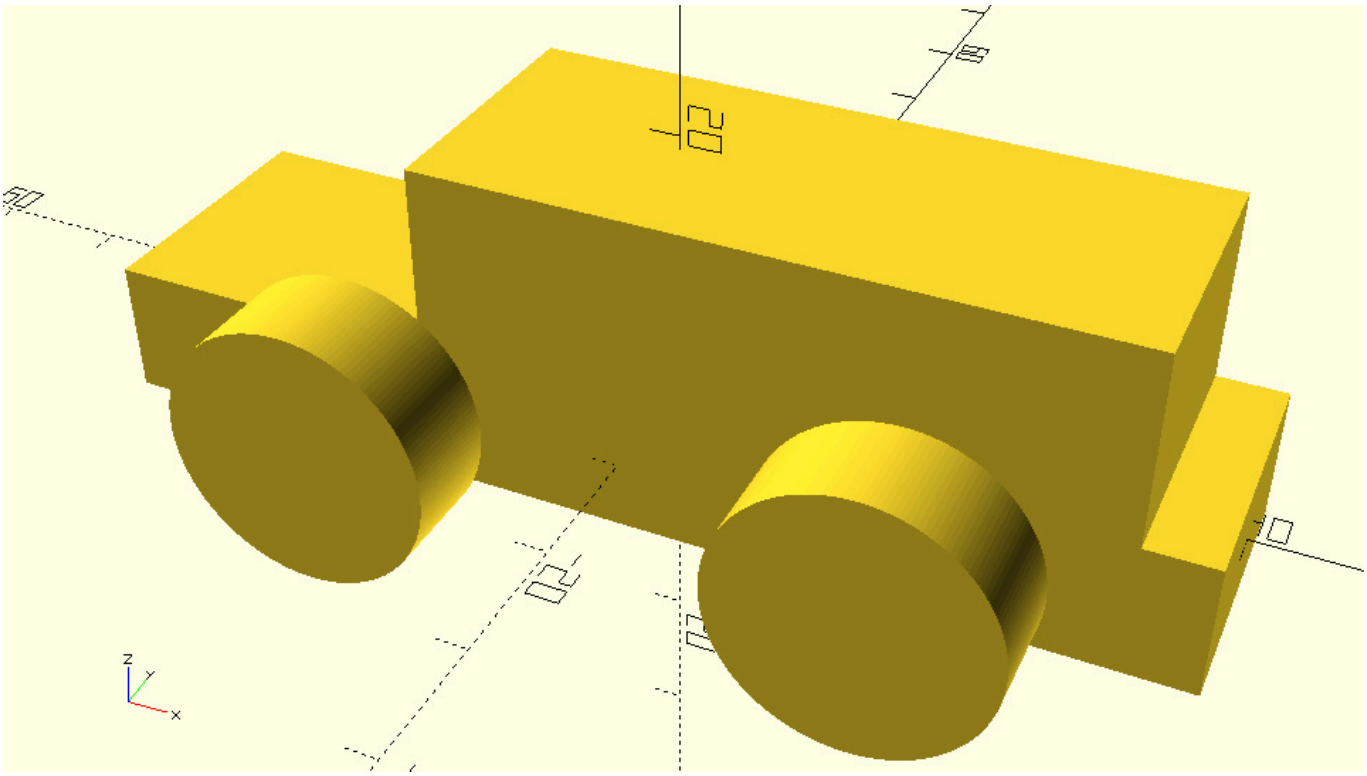


- long body - large wheels

Code[Collapse]

car_with_long_body_and_large_wheels.scad

```
…
long_body = true;
large_wheels = true;
…
```

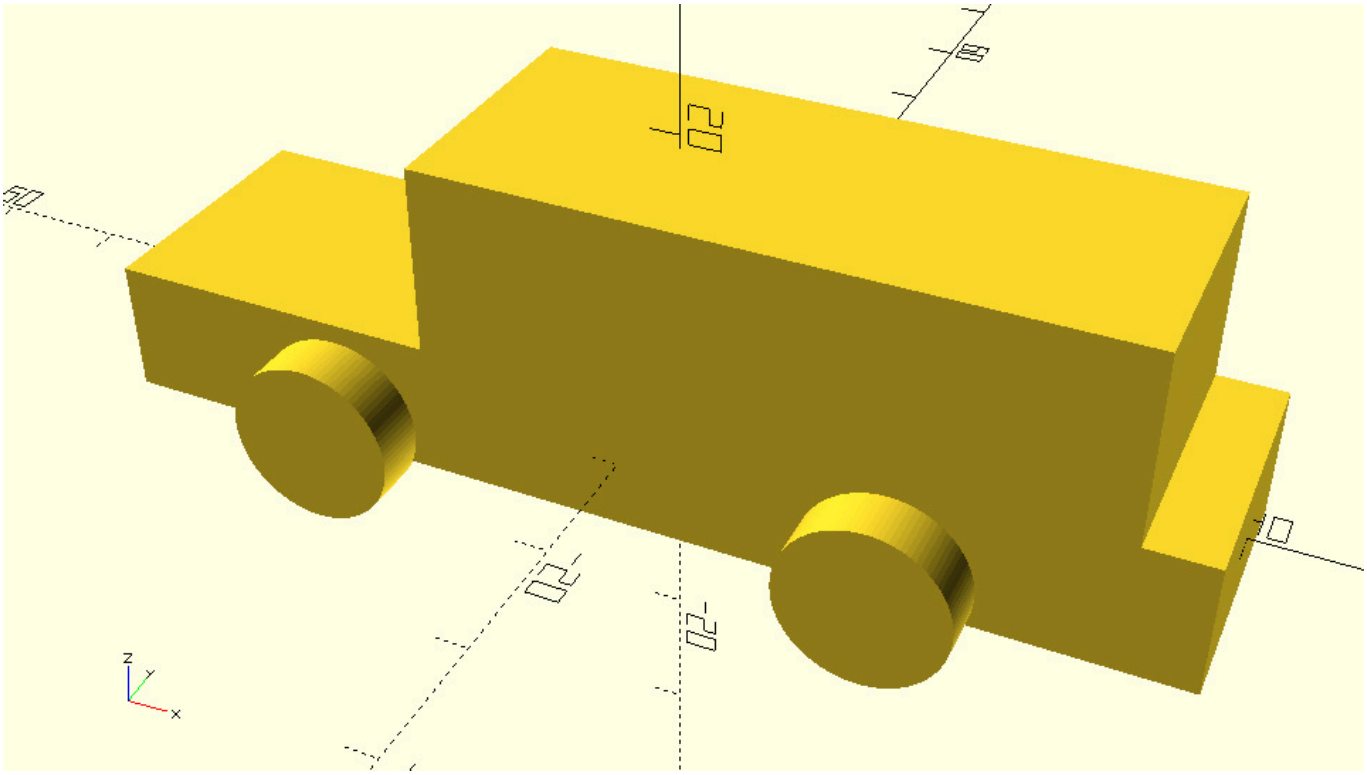


- long body - small wheels

Code[Collapse]

car_with_long_body_and_small_wheels.scad

```
…
long_body = true;
large_wheels = false;
…
```



More conditional variable assignments

The conditional assignment of variables can also be used with a properly adjusted syntax when there are more than two cases between which you would like to choose. In the previous example there were only two options for the body (short or long) and a boolean variable (long_body) was used to choose between those two options.

What if you want to be able to choose between four versions of the body (short, long, rectangular and normal)? A boolean variable can't be used to represent your choice of body version since it can only have two values (true or false). For this reason, you are going to use a character to represent your choice of body.

The choice of a short body will be represented by the character s.

Code

```
body_version = "s";
```

The choice of a long body will be represented by the character l.

Code

```
body_version = "l";
```

The choice of a rectangular body will be represented by the character r.

Code

```
body_version = "r";
```

The choice of a normal body will be represented by the character n.

Code

```
body_version = "n";
```

The conditional assignments when there are more than two options should take the following form.

Code

```
// base_length
base_length =
(body_version == "l") ? 80:
(body_version == "s") ? 60:
(body_version == "r") ? 65:70;

// top_length
top_length =
(body_version == "l") ? 50:
(body_version == "s") ? 30:
(body_version == "r") ? 65:40;

// top_offset
top_offset =
(body_version == "l") ? 10:
(body_version == "s") ? 5:
(body_version == "r") ? 0:7.5;
```

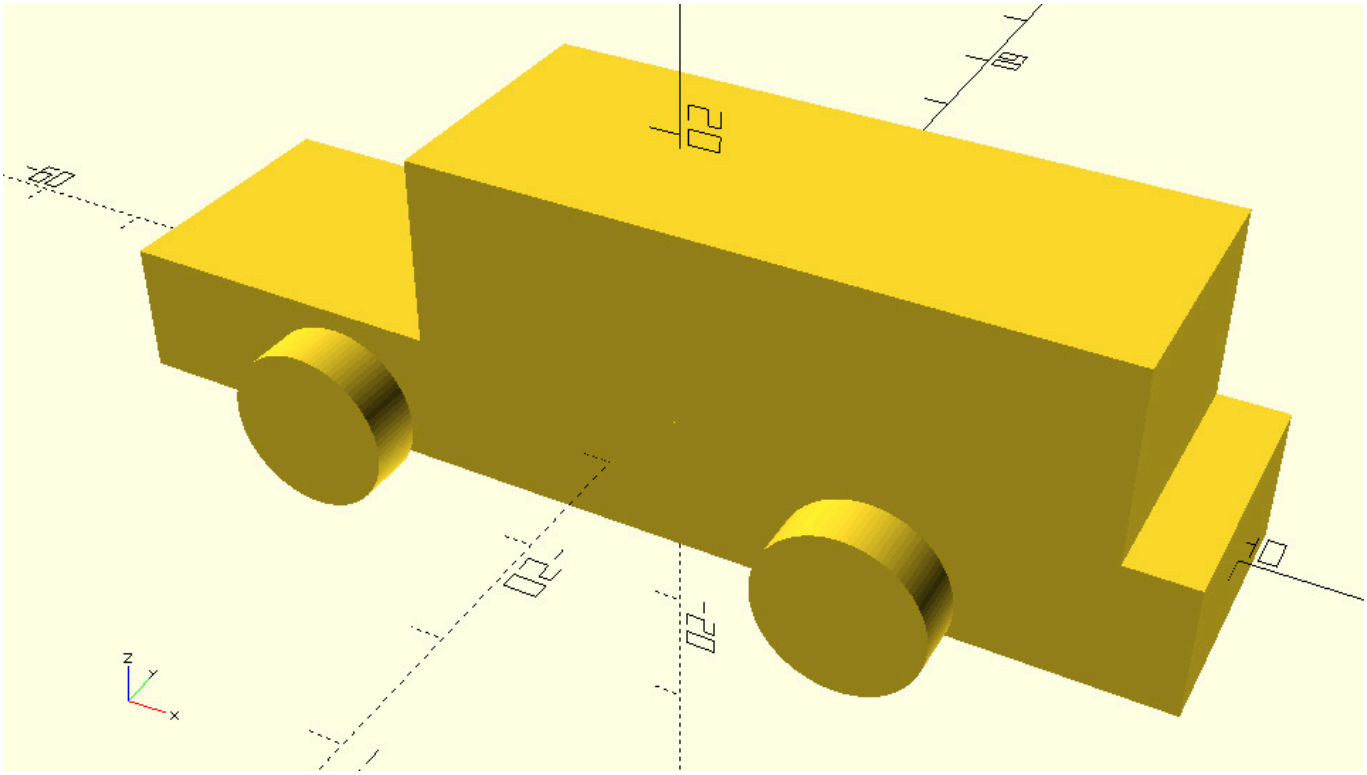
You should notice the following points about the definition of a conditional assignment when there are more than two options. First the name of the variable is typed out followed by the equal sign. Then follows a pair of parentheses that contains a condition, then a question mark, then the value to be assigned if the condition is true and then a colon. The previous sequence is repeated as required depending on the number of different available body versions. The last sequence is slightly different as it has an additional value which will be used as the default value when none of the conditions are true. In this case the default value corresponds to the normal version of the body. This is the reason why the character n that corresponds to the normal version of the body doesn't participate in any condition. Another thing you should notice is that the conditions are now comparison operations, specifically equality comparisons. If the value of the body_version variable is equal to the character that follows the double equal sign, then the condition is true and the corresponding value that follows the condition will be assigned to the variable.

By incorporating the above conditional assignments in your script, you can switch between a short, a long, a rectangular and a normal car body just by setting the body_version equal to the character s, l, r or n respectively.

Code

car_with_long_body_version.scad

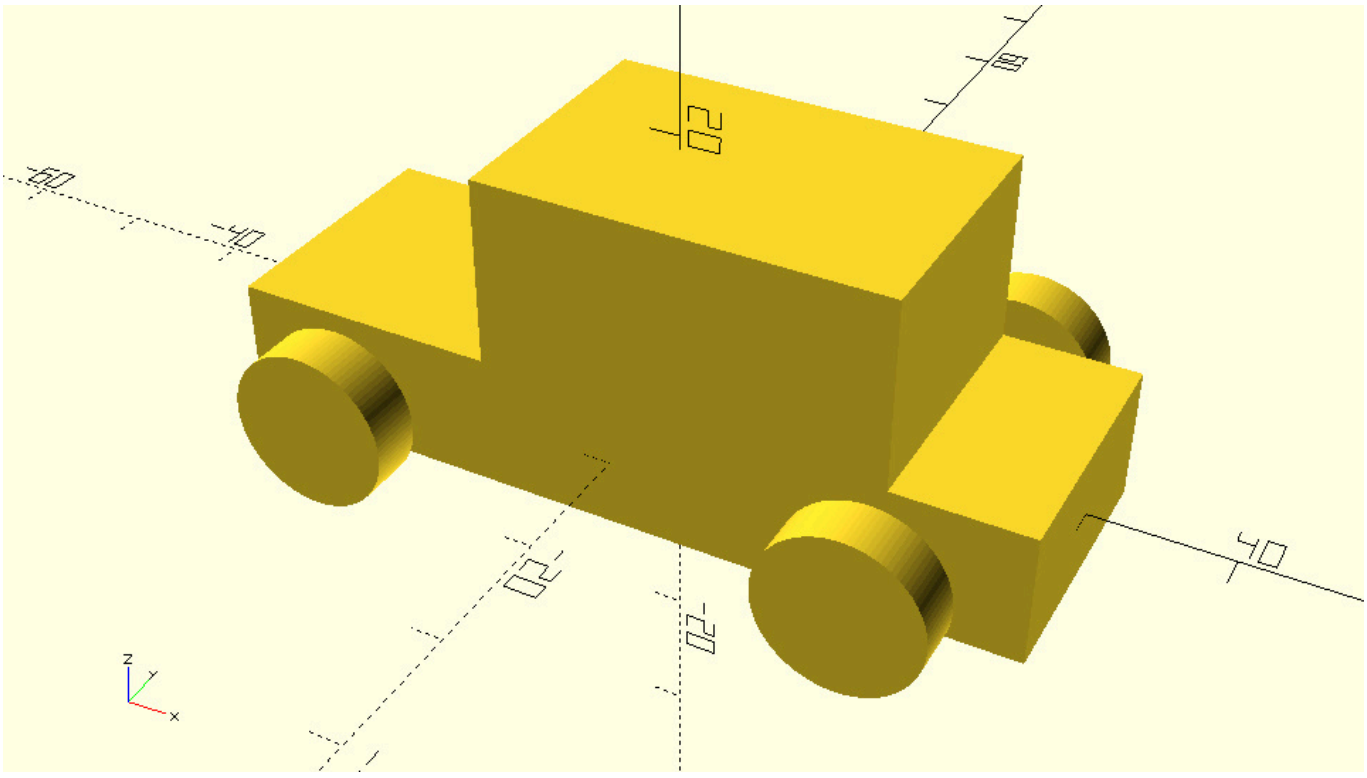
```
use <vehicle_parts.scad>
$fa=1;
$fs=0.4;

// Conditional assignment of body variables
body_version = "l";

// base_length
base_length =
(body_version == "l") ? 80:
(body_version == "s") ? 60:
(body_version == "r") ? 65:70;

// top_length
top_length =
(body_version == "l") ? 50:
(body_version == "s") ? 30:
(body_version == "r") ? 65:40;

// top_offset
top_offset =
(body_version == "l") ? 10:
(body_version == "s") ? 5:
(body_version == "r") ? 0:7.5;

// Creation of body
body(base_length=base_length, top_length=top_length, top_offset=top_offset);

// Creation of wheels and axles
large_wheels = false;
wheel_radius = (large_wheels) ? 10:6;
wheel_width = (large_wheels) ? 8:4;
track = 30;
wheelbase = 40;
// Front left wheel
translate([-wheelbase/2,-track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
 // Front right wheel
translate([-wheelbase/2,track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Rear left wheel
translate([wheelbase/2,-track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Rear right wheel
translate([wheelbase/2,track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Front axle
translate([-wheelbase/2,0,0])
    axle(track=track);
// Rear axle
translate([wheelbase/2,0,0])
    axle(track=track);
```



Code

car_with_short_body_version.scad

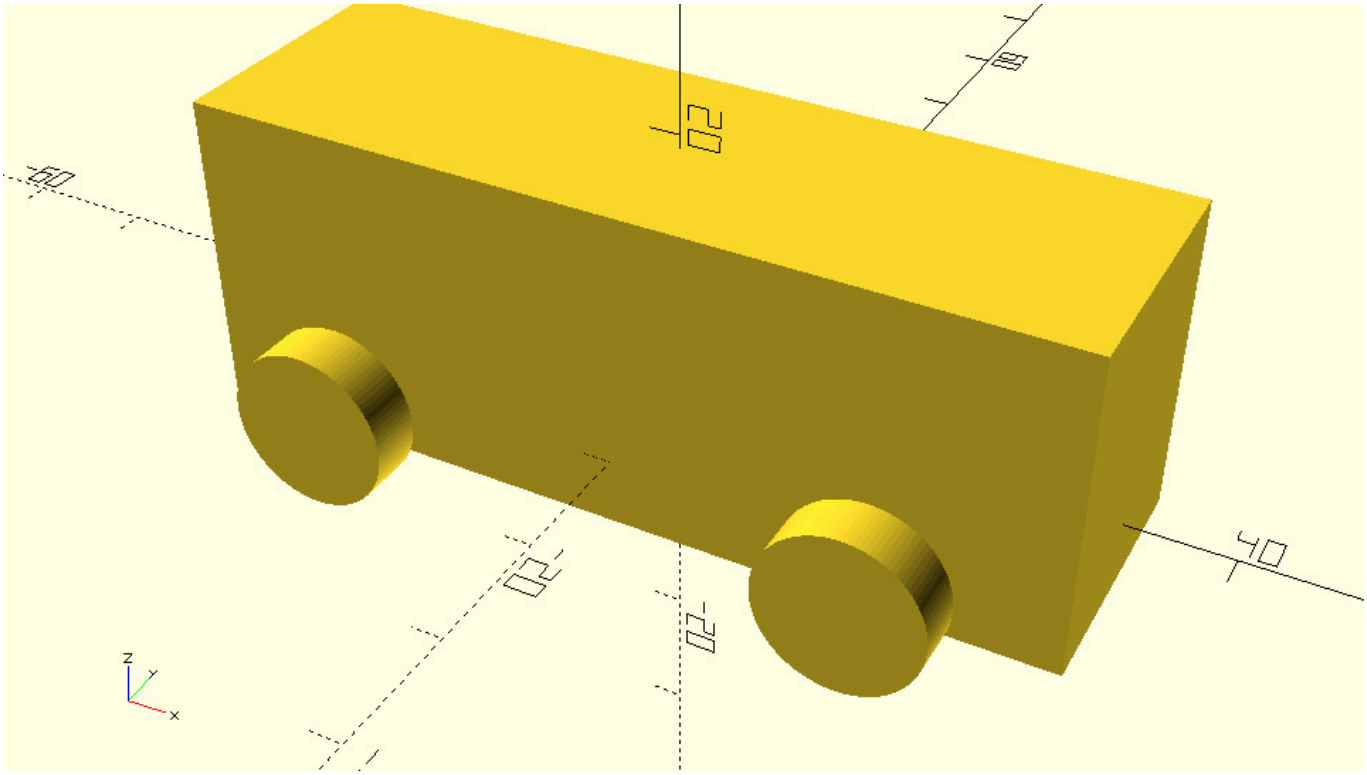
```
…
body_version = "s";
…
```



Code

car_with_rectangular_body_version.scad

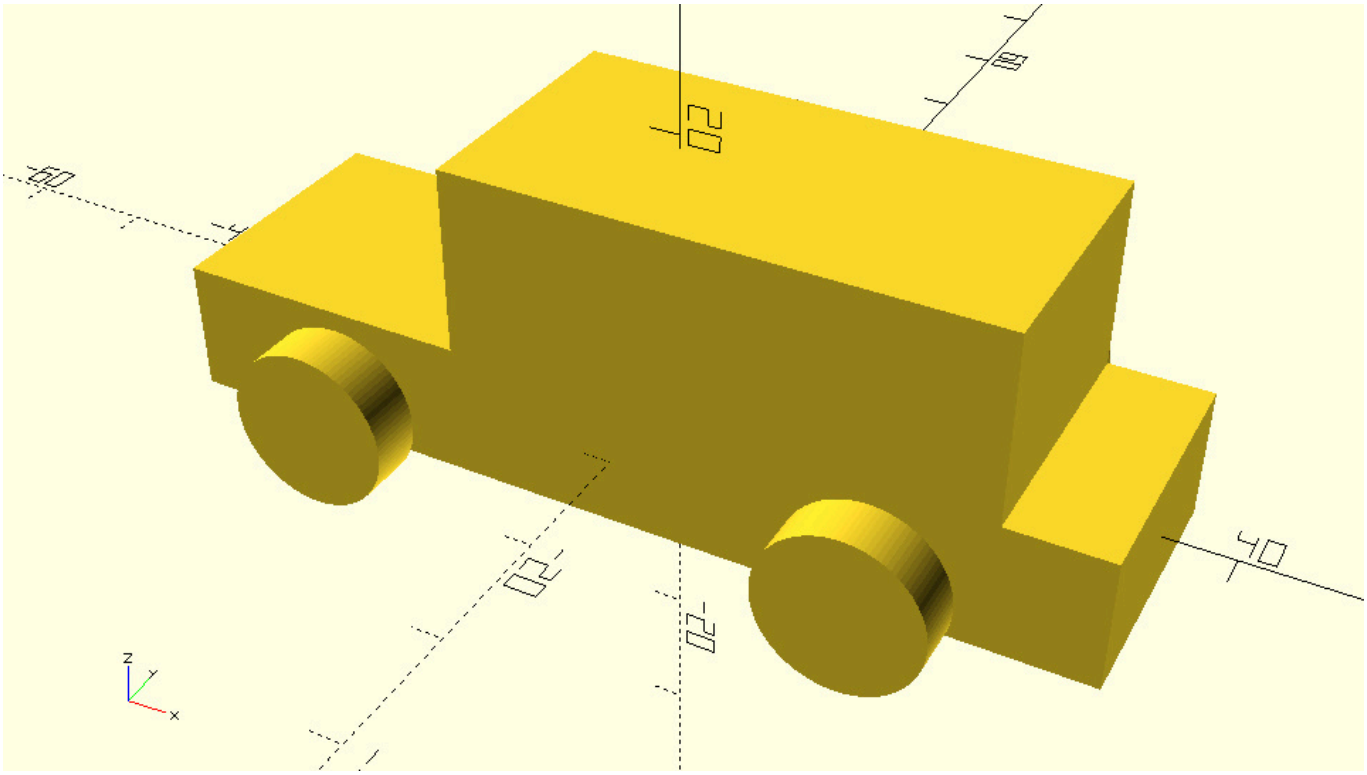
```
…
body_version = "r";
…
```



Code

car_with_normal_body_version.scad

```
…
body_version = "n";
…
```



Exercise

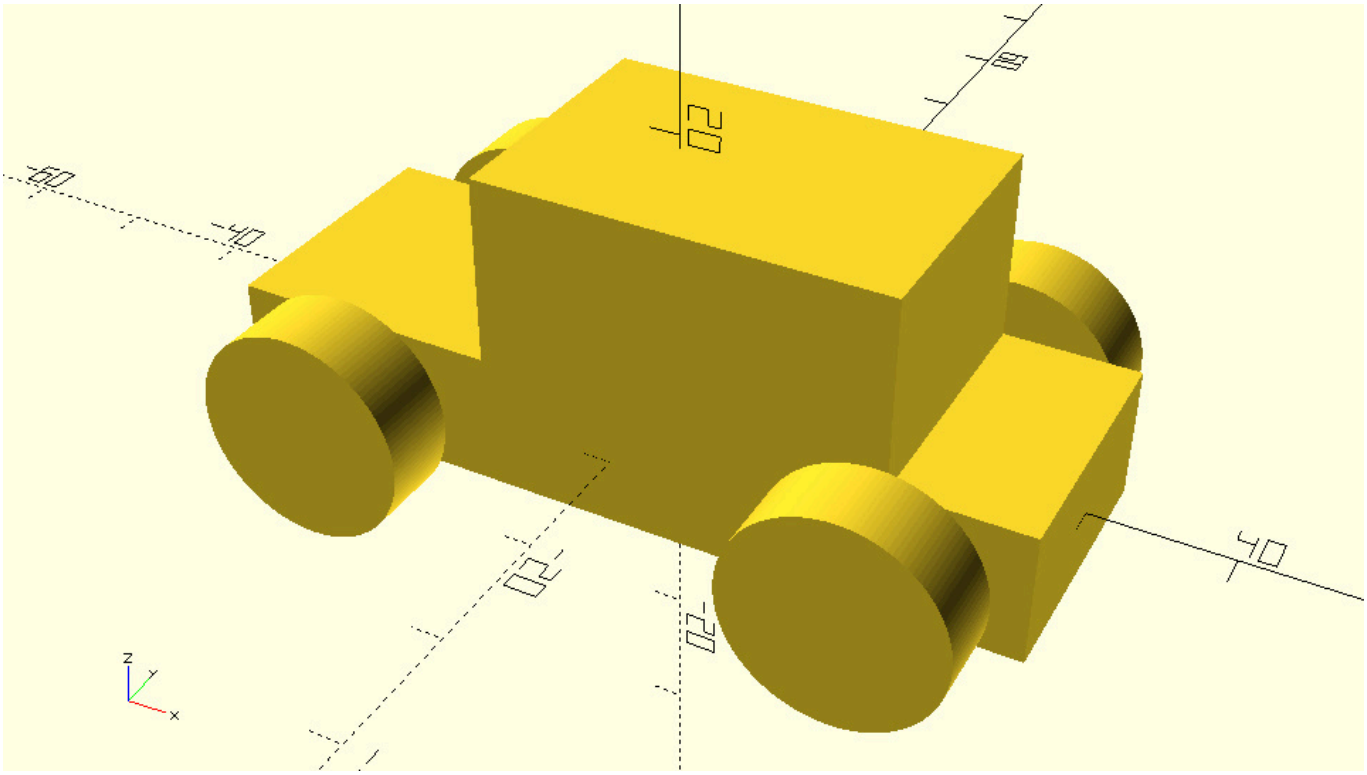
Add a wheels_version variable to the previous example. The variable should only take character values. Add appropriate conditional assignments that assign different values to the wheel_radius and wheel_width variables. The wheels_version variable should be used as the condition for both assignments. If the value of the wheels_version variable is the character s (small), the wheel_radius and wheel_width variables should be set equal to 8 and 4 units respectively. If the value of the wheels_version variable is the character m (medium), the wheel_radius and wheel_width variables should be set equal to 9 and 6 units respectively. If the value of the wheels_version variable is the character l (large), the wheel_radius and wheel_width variables should be set equal to 10 and 8 units respectively. The case of the small version of the wheels should be used as the default case of the conditional assignments. Set appropriate values to the body_version and wheels_version variables to create the following versions of the car: short body - medium wheels, rectangular body - large wheels, normal body - small wheels.

- short body - medium wheels

Code[Collapse]

car_with_short_body_and_medium_wheels.scad

```
…
body_version = "s"; //s-short, n-normal, l-large, r-rectangular
…
wheels_version = "m"; //s-small, m-medium, l-large
wheel_radius =
(wheels_version == "l") ? 10:
(wheels_version == "m") ? 8:6;
wheel_width =
(wheels_version == "l") ? 8:
(wheels_version == "m") ? 6:4;
…
```

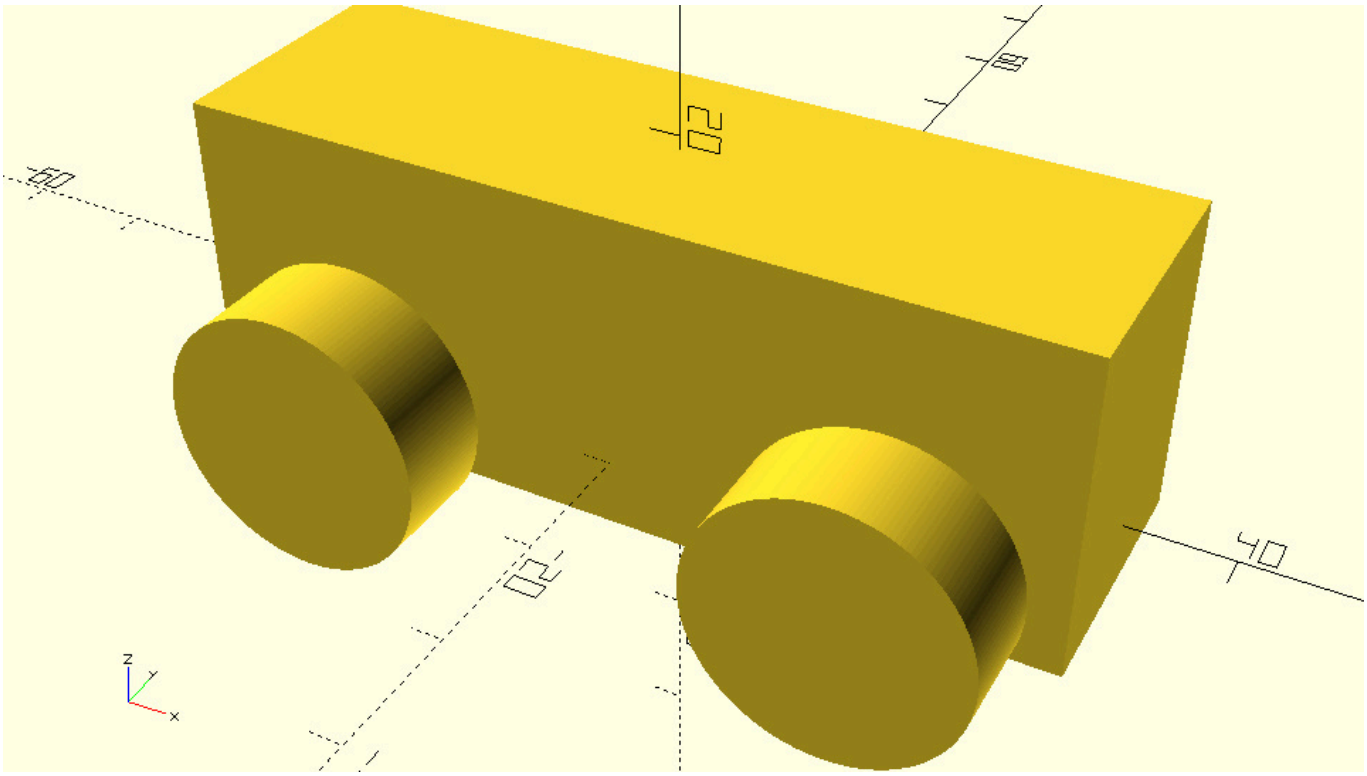


- rectangular body - large wheels

Code[Collapse]

car_with_rectangular_body_and_large_wheels.scad

```
…
body_version = "r"; //s-short, n-normal, l-large, r-rectangular
…
wheels_version = "l"; //s-small, m-medium, l-large
…
```

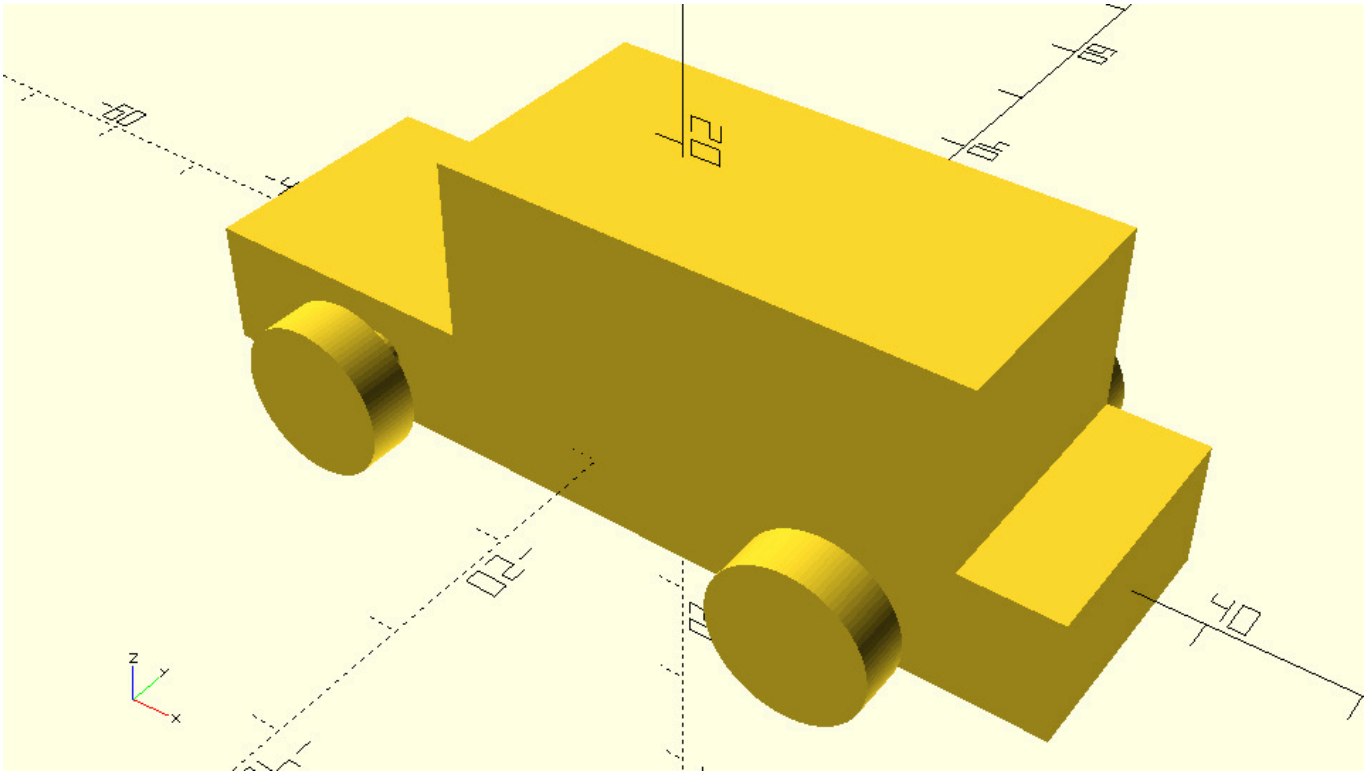


- normal body - small wheels

Code[Collapse]

car_with_normal_body_and_small_wheels.scad

```
…
body_version = "n"; //s-short, n-normal, l-large, r-rectangular
…
wheels_version = "s"; //s-small, m-medium, l-large
…
```



Conditional creation of objects - If statement

Conditional assignment of variables is a great tool to easily navigate between different but specific versions of your model. Using conditional assignments, you were able to define different body and wheel sizes for your car and effortlessly choose between them without having to manually provide the values for all involved variables every single time.

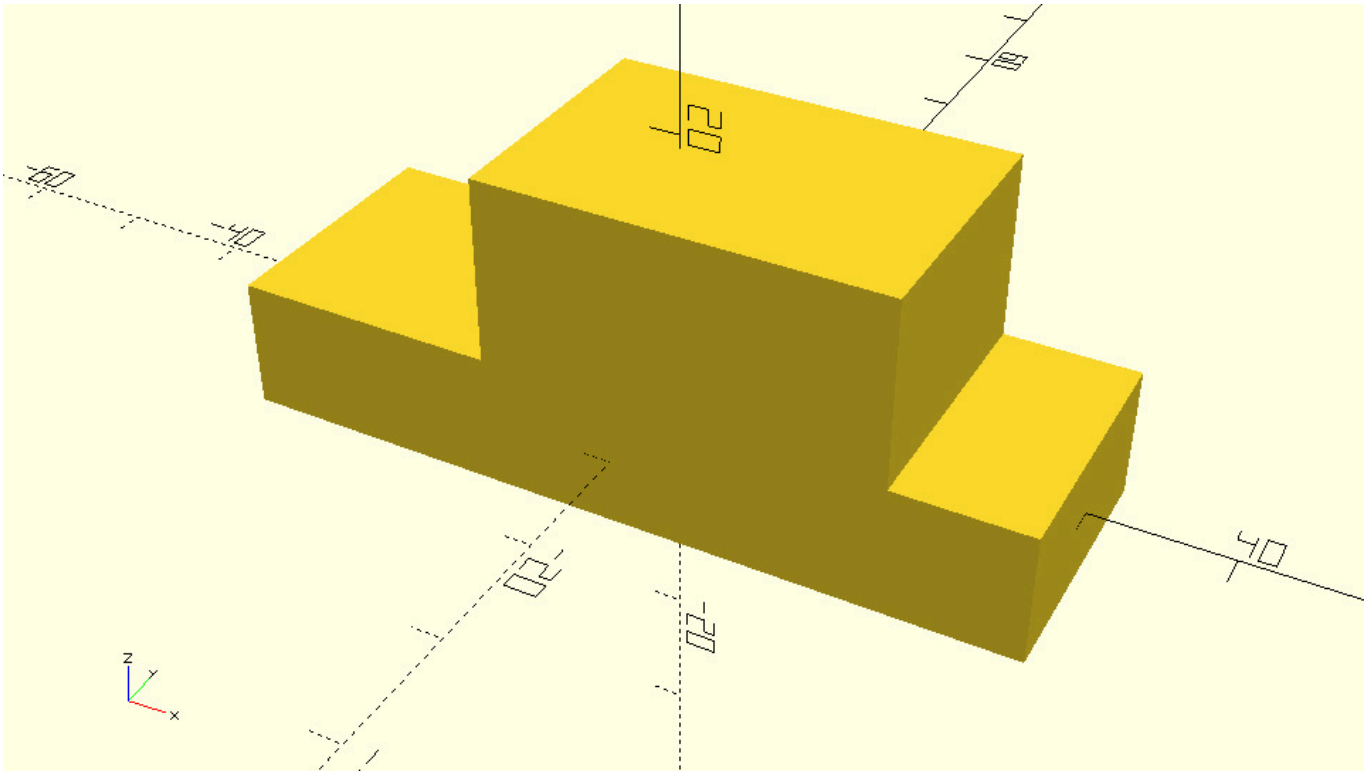
What if you wanted to have the same control over the type of wheel (ex. simple, round, complex) or body (ex. square, round)? What would this require? In order to achieve this, you would need to have conditional creation of objects, which can be achieved with the use of the if statement.

Before you go into customizing the type of wheel and body, you can get familiar with if statements with some shorter examples. Recall the car body module that you created in a previous chapter. The module has some input parameters which are used to create two cubes, one cube for the body's base and one for the body's top.

Code

car_body_from_module.scad

```
module body(base_height=10, top_height=14, base_length=60, top_length=30, width=20, top_offset=5) {
    // Car body base
    cube([base_length,width,base_height],center=true);
    // Car body top
    translate([top_offset,0,base_height/2+top_height/2])
        cube([top_length,width,top_height],center=true);
}
$fa = 1;
$fs = 0.4;
body();
```



Using an if statement you are going to see how the creation of the body's top can be parameterized. First you need to define an additional input parameter for the module. This parameter will be named top and will hold boolean values. If this parameter is false, the module should create only the base of the body. If it's true, it should also create the top of the body. This can be achieved by using an if statement in the following way.

Code

```
module body(base_height=10, top_height=14, base_length=60, top_length=30, width=20, top_offset=5,
top) {
    // Car body base
    cube([base_length,width,base_height],center=true);
    // Car body top
    if (top) {
        translate([top_offset,0,base_height/2+top_height/2])
        cube([top_length,width,top_height],center=true);
    }
}
$fa = 1;
$fs = 0.4;
```

You should notice the following points regarding the definition of the if statement. First the if keyword is typed out and then follows a pair of parentheses. Inside of the parentheses the condition that will dictate whether the if statement will be executed is defined. Lastly, there is a pair of curly brackets inside of which exist all statements that will be executed if the supplied condition is true. In this case the supplied condition is the boolean variable top, which represents your choice to create a car body that does or doesn't have a top part. The statement that is placed inside the curly brackets is the statement that creates the top part of the car's body.

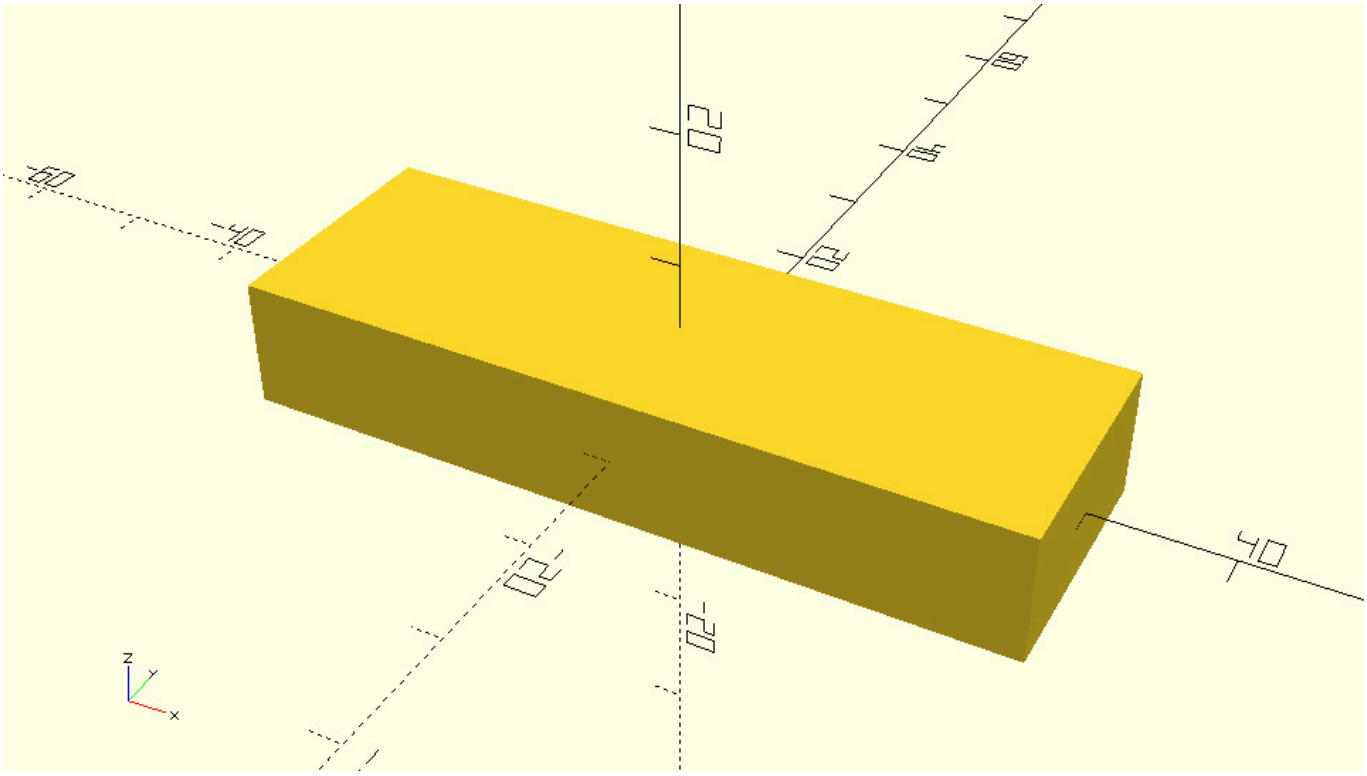
This particular form of if statement is known as a simple if statement. This means that if the condition is true then the corresponding commands are executed, otherwise nothing happens. There are two other forms of the if statement that will be covered later, but first take a moment to investigate how the new body module works in practice.

When the input parameter top is set to false, only the base of the body is created.

Code

car_body_without_top.scad

```
…
body(top=false);
…
```

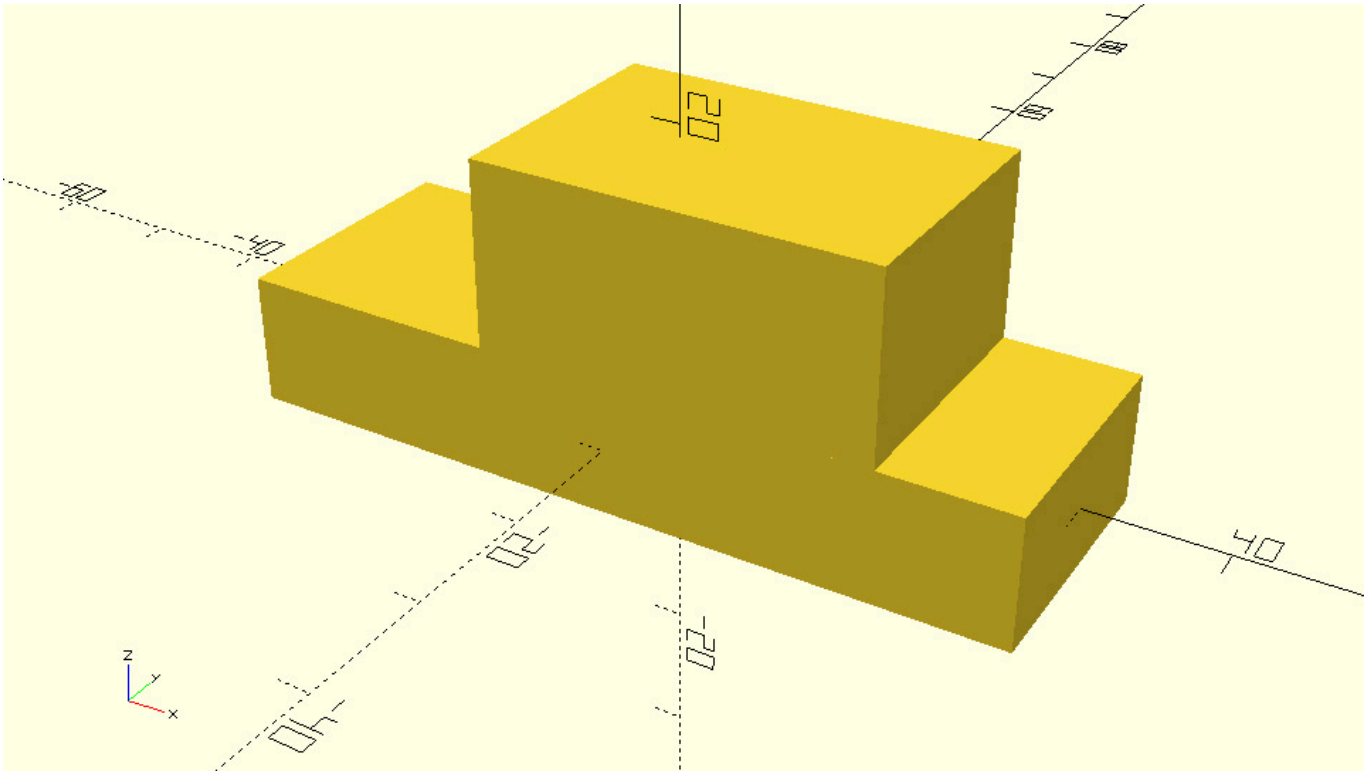


When it's set to true, both the base and the top of the body are created.

Code

car_body_with_top.scad

```
…
body(top=true);
…
```



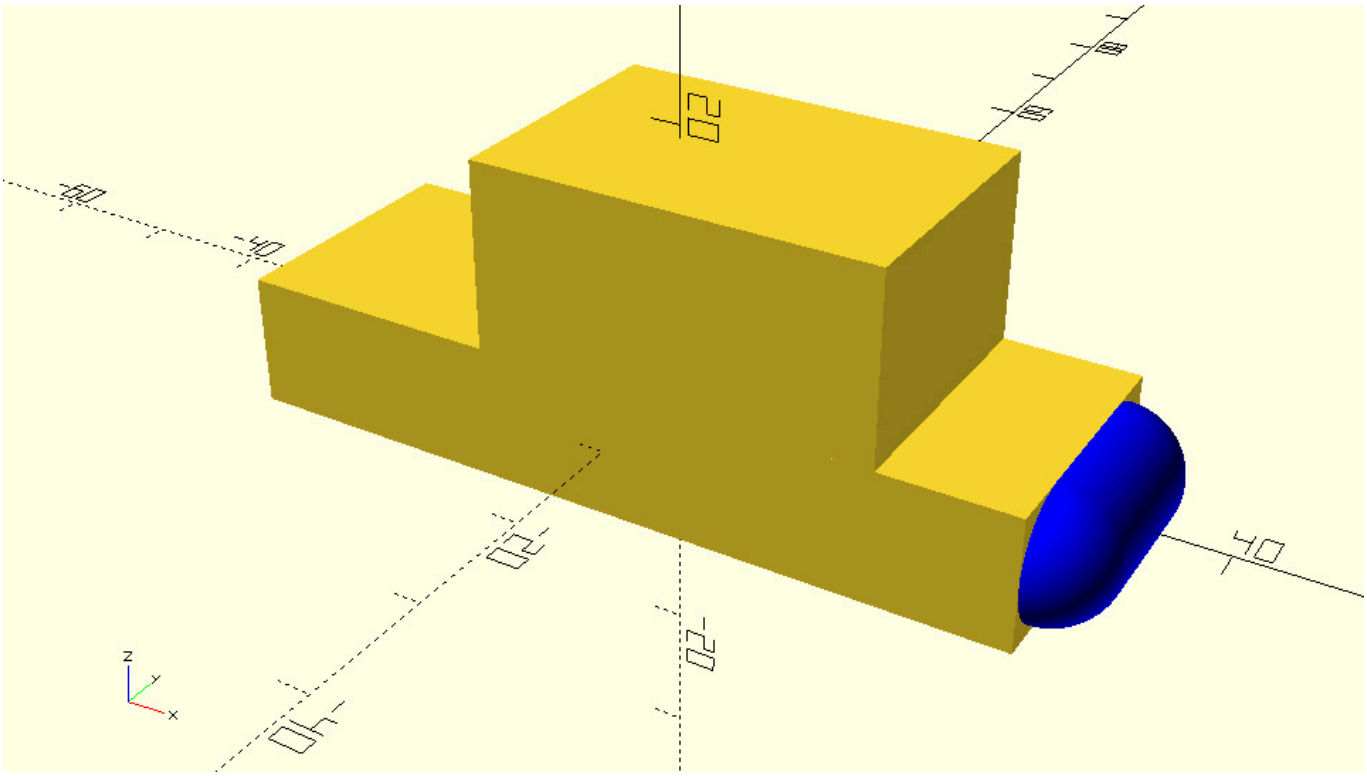
Exercise

Take a look at the following car body module. You will notice that the module has been modified to include the creation of a rear bumper. The color command that has been applied to the bumper simply adds a visual effect during preview. In this case, the color command is used to draw your attention on the newly added part; it shouldn't bother you any further than that.

Code

car_body_with_rear_bumper.scad

```
module body(base_height=10, top_height=14, base_length=60, top_length=30, width=20, top_offset=5,
top) {
    // Car body base
    cube([base_length,width,base_height],center=true);
    // Car body top
    if (top) {
        translate([top_offset,0,base_height/2+top_height/2])
            cube([top_length,width,top_height],center=true);
    }
    // Rear bumper
    color("blue") {
        translate([base_length/2,0,0])rotate([90,0,0]) {
            cylinder(h=width - base_height,r=base_height/2,center=true);
            translate([0,0,(width - base_height)/2])
                sphere(r=base_height/2);
            translate([0,0,-(width - base_height)/2])
                sphere(r=base_height/2);
        }
    }
}
$fa = 1;
$fs = 0.4;
body(top=true);
```



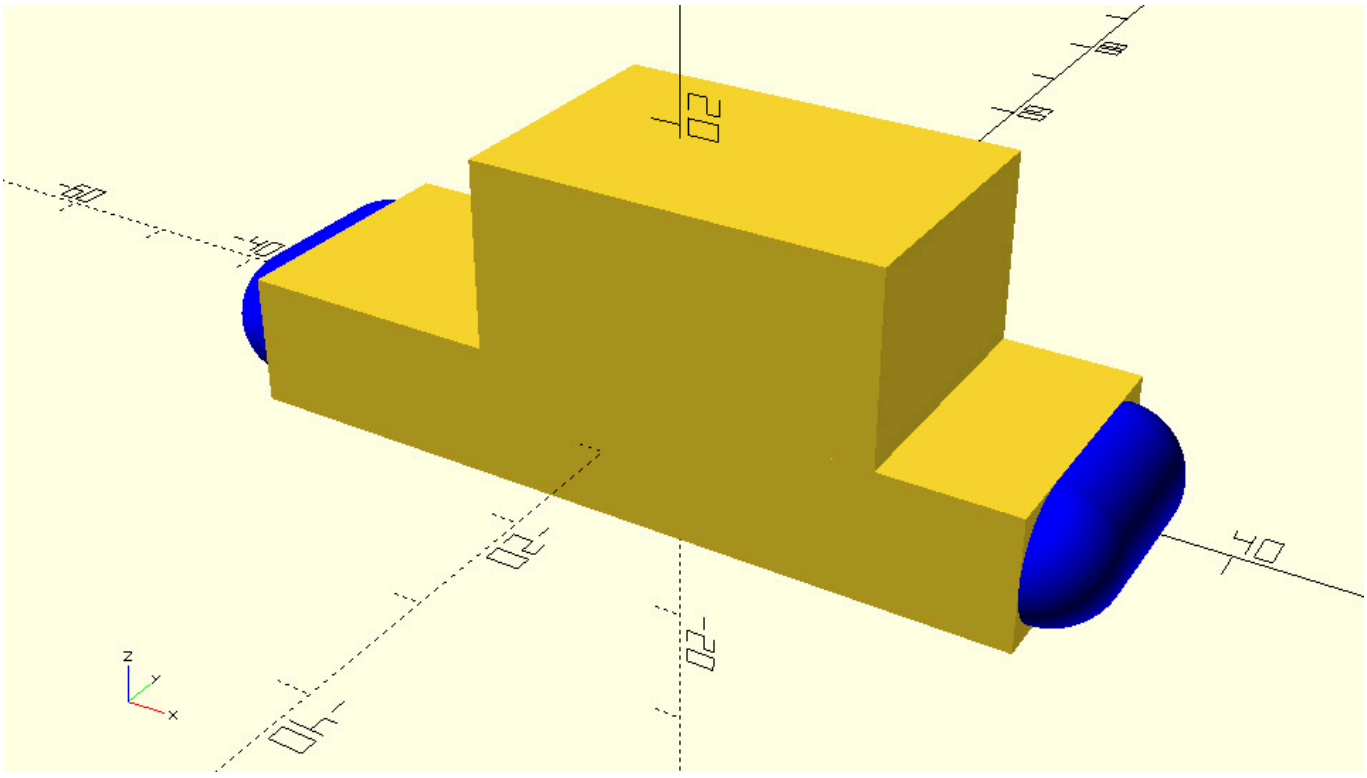
Exercise

To create the front bumper, copy and paste the statements that create the rear bumper and modify the translation statement accordingly.

Code[Collapse]

car_body_with_front_and_rear_bumper.scad

```
module body(base_height=10, top_height=14, base_length=60, top_length=30, width=20, top_offset=5,
top) {
    // Car body base
    cube([base_length,width,base_height],center=true);
    // Car body top
    if (top) {
        translate([top_offset,0,base_height/2+top_height/2])
            cube([top_length,width,top_height],center=true);
    }
    // Front bumper
    color("blue") {
        translate([-base_length/2,0,0])rotate([90,0,0]) {
            cylinder(h=width - base_height,r=base_height/2,center=true);
            translate([0,0,(width - base_height)/2])
                sphere(r=base_height/2);
            translate([0,0,-(width - base_height)/2])
                sphere(r=base_height/2);
        }
    }
    // Rear bumper
    color("blue") {
        translate([base_length/2,0,0])rotate([90,0,0]) {
            cylinder(h=width - base_height,r=base_height/2,center=true);
            translate([0,0,(width - base_height)/2])
                sphere(r=base_height/2);
            translate([0,0,-(width - base_height)/2])
                sphere(r=base_height/2);
        }
    }
}
$fa = 1;
$fs = 0.4;
body(top=true);
```



Exercise

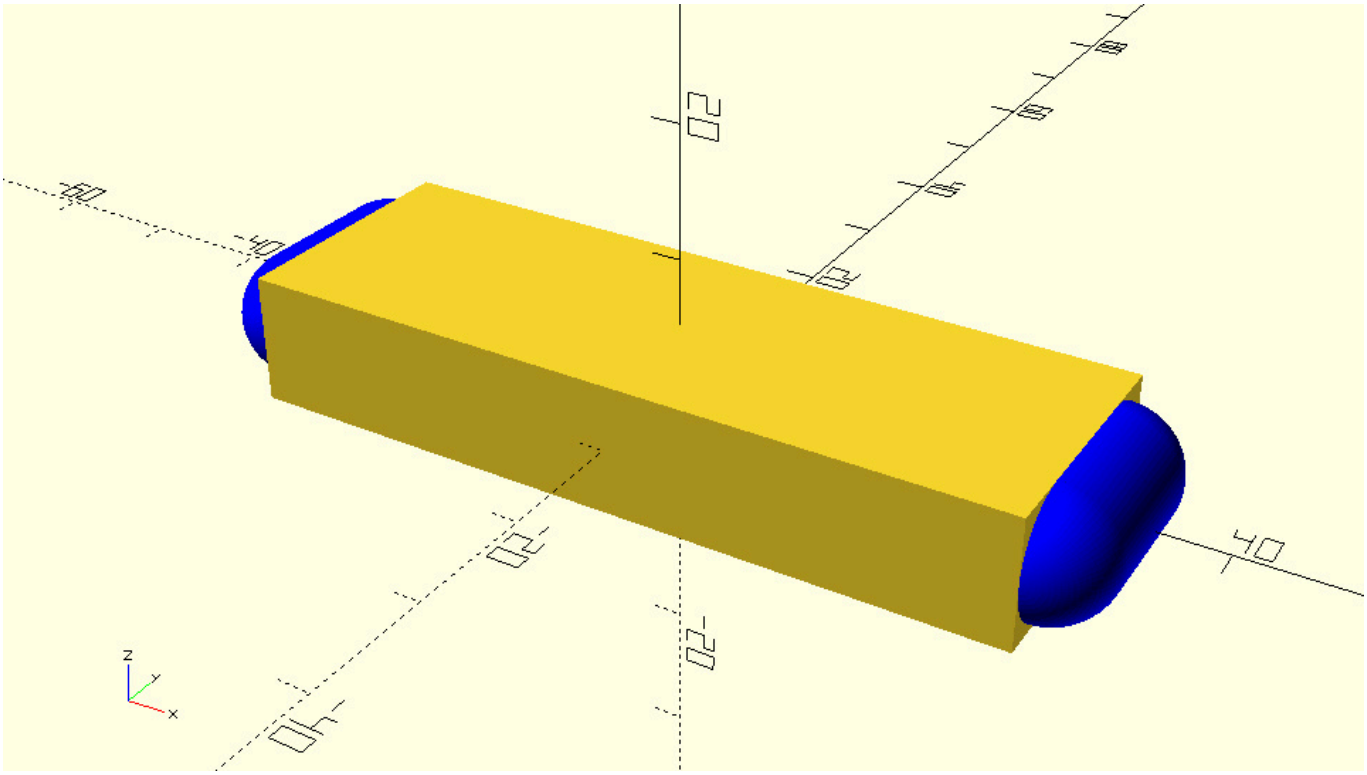
Define two additional input parameters for the body module. One named front_bumper and one named rear_bumper. Keeping in mind that these parameters should take boolean values define two if statements that conditionally create the front and rear bumpers. The front bumper should be created only if the front_bumper input parameter is true, the second bumper accordingly. Use the body module to create the following car bodies: base only - front and rear bumper, base and top - front bumper, base and top - front and rear bumper.

- base only - front and rear bumper

Code[Collapse]

body_without_top_with_front_and_rear_bumper.scad

```
module body(base_height=10, top_height=14, base_length=60, top_length=30, width=20, top_offset=5,
top, front_bumper, rear_bumper) {
    // Car body base
    cube([base_length,width,base_height],center=true);
    // Car body top
    if (top) {
        translate([top_offset,0,base_height/2+top_height/2])
        cube([top_length,width,top_height],center=true);
    }
    // Front bumper
    if (front_bumper) {
        color("blue") {
            translate([-base_length/2,0,0])rotate([90,0,0]) {
                cylinder(h=width - base_height,r=base_height/2,center=true);
                translate([0,0,(width - base_height)/2])
                    sphere(r=base_height/2);
                translate([0,0,-(width - base_height)/2])
                    sphere(r=base_height/2);
            }
        }
    }
    // Rear bumper
    if (rear_bumper) {
        color("blue") {
            translate([base_length/2,0,0])rotate([90,0,0]) {
                cylinder(h=width - base_height,r=base_height/2,center=true);
                translate([0,0,(width - base_height)/2])
                    sphere(r=base_height/2);
                translate([0,0,-(width - base_height)/2])
                    sphere(r=base_height/2);
            }
        }
    }
}
$fa = 1;
$fs = 0.4;
body(top=false,front_bumper=true,rear_bumper=true);
```

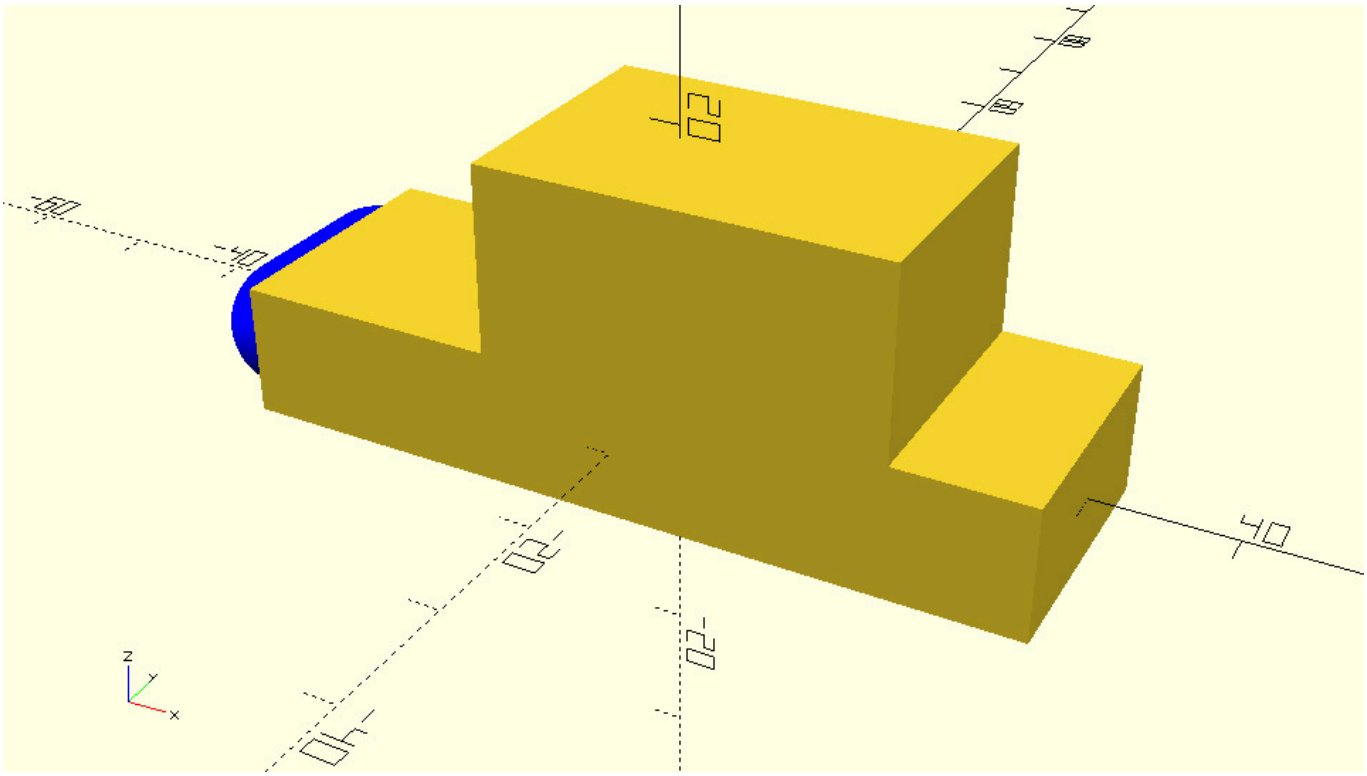


- base and top - front bumper

Code[Collapse]

body_with_top_with_front_bumper.scad

```
…
body(top=true,front_bumper=true,rear_bumper=false);
…
```

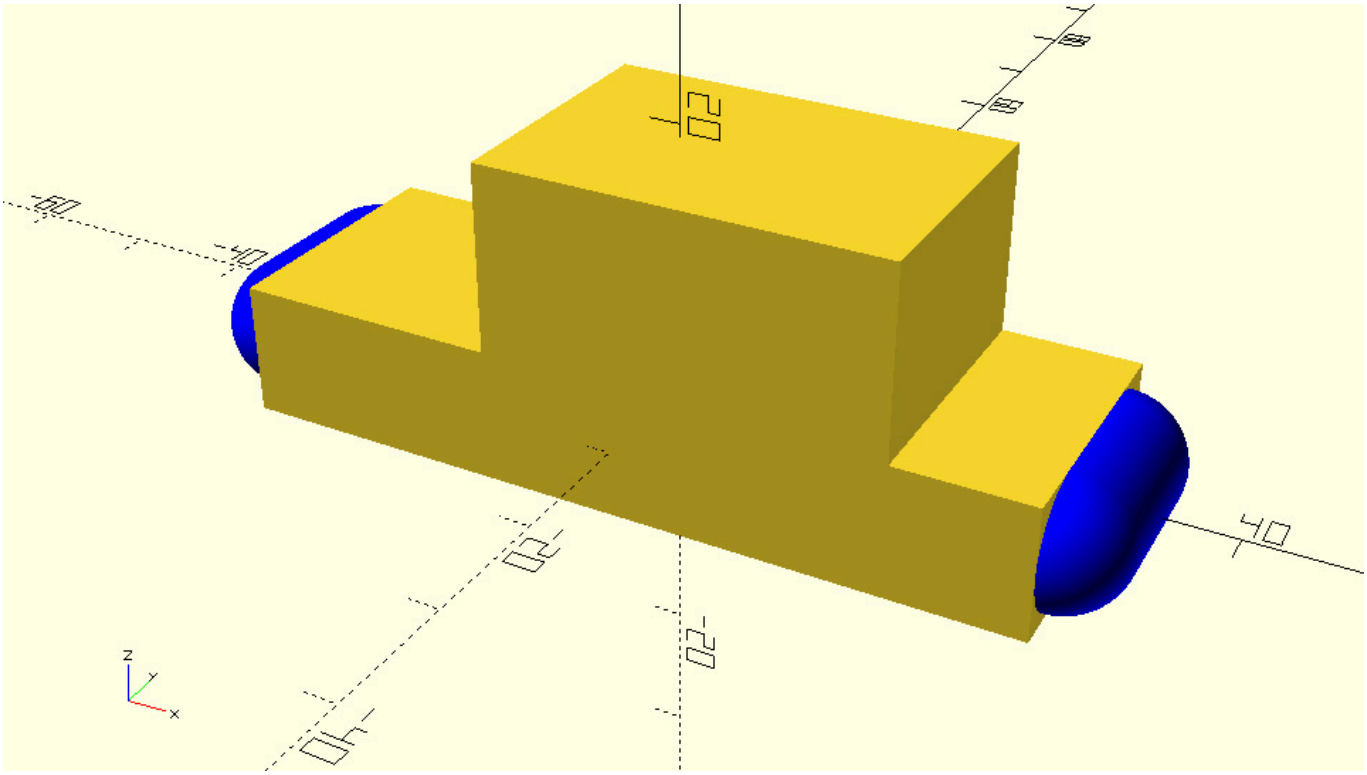


- base and top - front and rear bumper

Code[Collapse]

body_with_top_with_front_and_rear_bumper.scad

```
…
body(top=true,front_bumper=true,rear_bumper=true);
…
```



Challenge

In this chapter you learned about conditional assignment of variables and simple if statements. Specifically, you learned how to conditionally modify dimensions and transformations of parts of your designs as well as how to conditionally include or exclude parts from them. It's time to put these two together in a single car model.

Exercise

If you have been following along this tutorial you should have a vehicle_parts.scad script on your machine from a previous chapter. Open this script and update the body module according to the last example so that it has the ability to conditionally create the top of the body as well as a front and a rear bumper. Set default values for the newly added input parameters. Specifically set true, false and false as the default value of the top, front_bumper and rear_bumper variable accordingly. Save the changes and close the script.

Code[Collapse]

```
…
module body(base_height=10, top_height=14, base_length=60, top_length=30, width=20, top_offset=5,
top=true, front_bumper=false, rear_bumper=false) {
    // Car body base
    cube([base_length,width,base_height],center=true);
    // Car body top
    if (top) {
        translate([top_offset,0,base_height/2+top_height/2])
            cube([top_length,width,top_height],center=true);
    }
    // Front bumper
    if (front_bumper) {
        color("blue") {
            translate([-base_length/2,0,0])rotate([90,0,0]) {
                cylinder(h=width - base_height,r=base_height/2,center=true);
                translate([0,0,(width - base_height)/2])
                    sphere(r=base_height/2);
                translate([0,0,-(width - base_height)/2])
                    sphere(r=base_height/2);
            }
        }
    }
    // Rear bumper
    if (rear_bumper) {
        color("blue") {
            translate([base_length/2,0,0])rotate([90,0,0]) {
                cylinder(h=width - base_height,r=base_height/2,center=true);
                translate([0,0,(width - base_height)/2])
                    sphere(r=base_height/2);
                translate([0,0,-(width - base_height)/2])
                    sphere(r=base_height/2);
            }
        }
    }
}
…
```

Exercise

Given the following script that creates a car, make appropriate additions and modifications to the script in order to parameterize conditionally the design of the car. Specifically, you will need to define a body_version, a wheels_version, a top, a front_bumper and a rear_bumper variable that will be used for making design choices regarding the car's design. If necessary, review the previous examples and exercises of this chapter to remember what effect these variables should have on the design of the car and how to implement them. Use the resulting script to create a version of the car that you like.

- Given script

Code

basic_car_script.scad

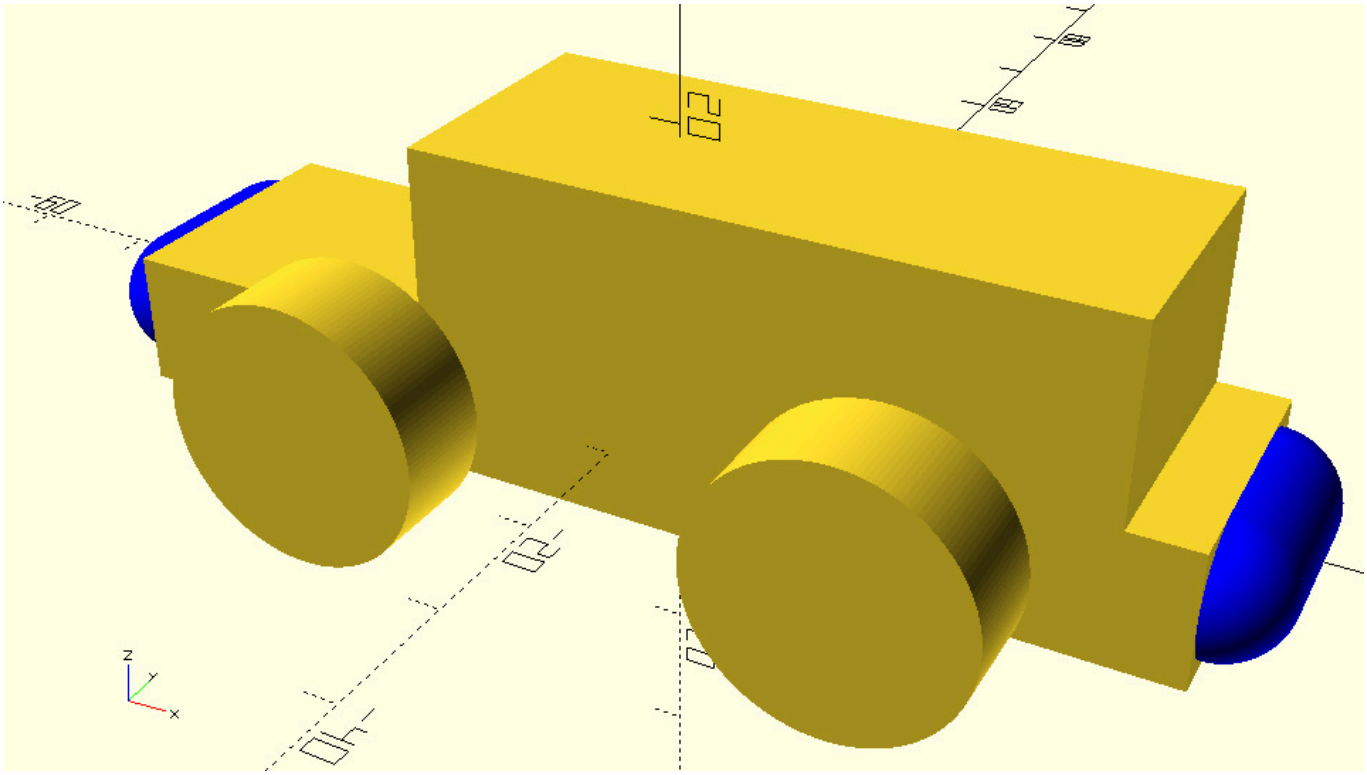
```
use <vehicle_parts.scad>
$fa=1;
$fs=0.4;

// Variables
track = 30;
wheelbase=40;

// Body
body();
// Front left wheel
translate([-wheelbase/2,-track/2,0])
    rotate([0,0,0])
    simple_wheel();
 // Front right wheel
translate([-wheelbase/2,track/2,0])
    rotate([0,0,0])
    simple_wheel();
// Rear left wheel
translate([wheelbase/2,-track/2,0])
    rotate([0,0,0])
    simple_wheel();
// Rear right wheel
translate([wheelbase/2,track/2,0])
    rotate([0,0,0])
    simple_wheel();
// Front axle
translate([-wheelbase/2,0,0])
    axle();
// Rear axle
translate([wheelbase/2,0,0])
    axle();
```

- Modified script

Code[Collapse]

car_from_highly_parameterized_script.scad

```
use <vehicle_parts.scad>
$fa=1;
$fs=0.4;

// Variables
body_version = "l"; //s-short, n-normal, l-large, r-rectangular
wheels_version = "l"; //s-small, m-medium, l-large
top = true;
front_bumper = true;
rear_bumper = true;
track = 30;
wheelbase=40;

// Conditional assignments
// Body: base_length
base_length =
(body_version == "l") ? 80:
(body_version == "s") ? 60:
(body_version == "r") ? 65:70;

// Body: top_length
top_length =
(body_version == "l") ? 50:
(body_version == "s") ? 30:
(body_version == "r") ? 65:40;

// Body: top_offset
top_offset =
(body_version == "l") ? 10:
(body_version == "s") ? 5:
(body_version == "r") ? 0:7.5;

// Wheels: radius
wheel_radius =
(wheels_version == "l") ? 10:
(wheels_version == "m") ? 8:6;

// Wheels: width
wheel_width =
(wheels_version == "l") ? 8:
(wheels_version == "m") ? 6:4;

// Body
body(base_length=base_length, top_length=top_length, top_offset=top_offset, top=top,
front_bumper=front_bumper, rear_bumper=rear_bumper);
// Front left wheel
translate([-wheelbase/2,-track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
 // Front right wheel
translate([-wheelbase/2,track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Rear left wheel
translate([wheelbase/2,-track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Rear right wheel
translate([wheelbase/2,track/2,0])
    rotate([0,0,0])
    simple_wheel(wheel_radius=wheel_radius, wheel_width=wheel_width);
// Front axle
translate([-wheelbase/2,0,0])
    axle(track=track);
// Rear axle
translate([wheelbase/2,0,0])
    axle(track=track);
```


Z
Y
X

Chapter 7

Creating repeating patterns of parts/models - For loops

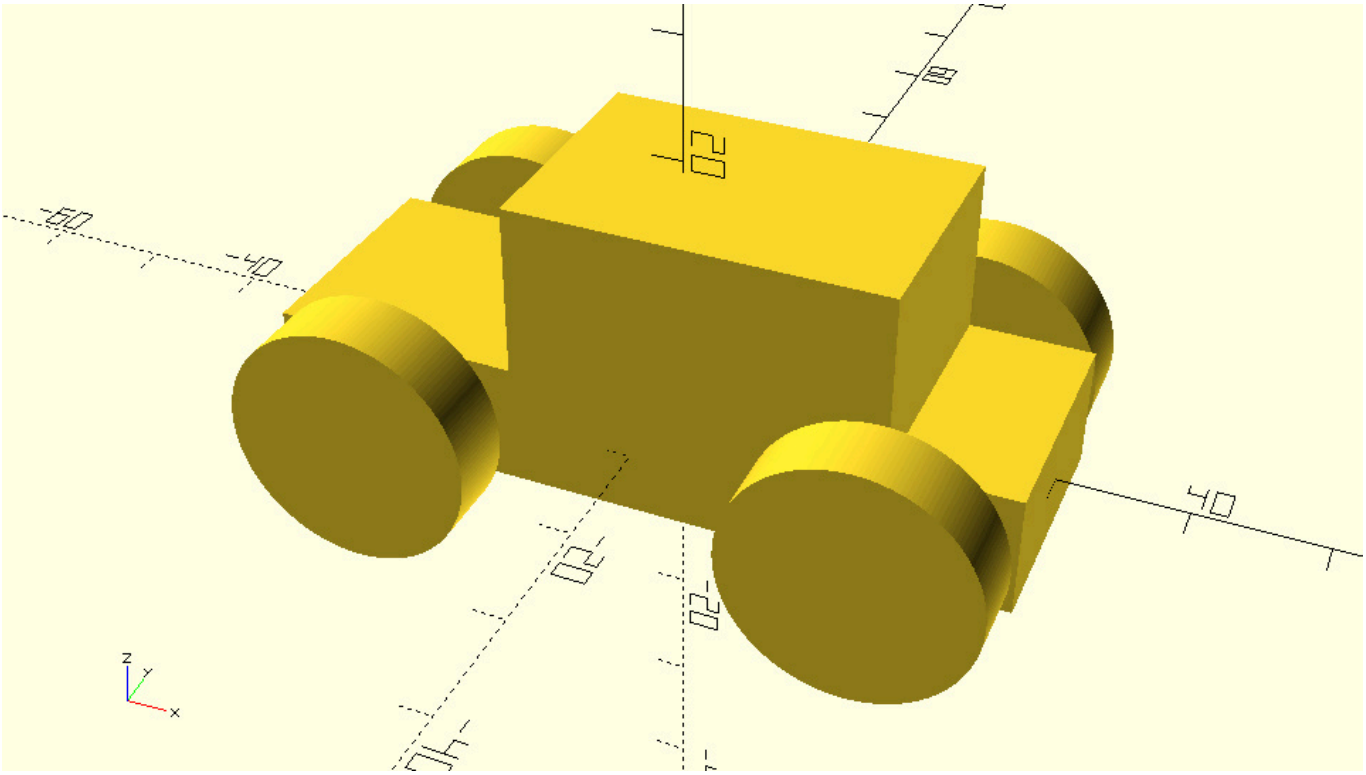
In the previous chapter you used if statements to control whether some part of your design should be created or not. In this chapter you are going to find out how you can create multiple parts or objects when they form a specific pattern.

Given the following car model as an example you are going to learn how to create such patterns.

Code

single_car.scad

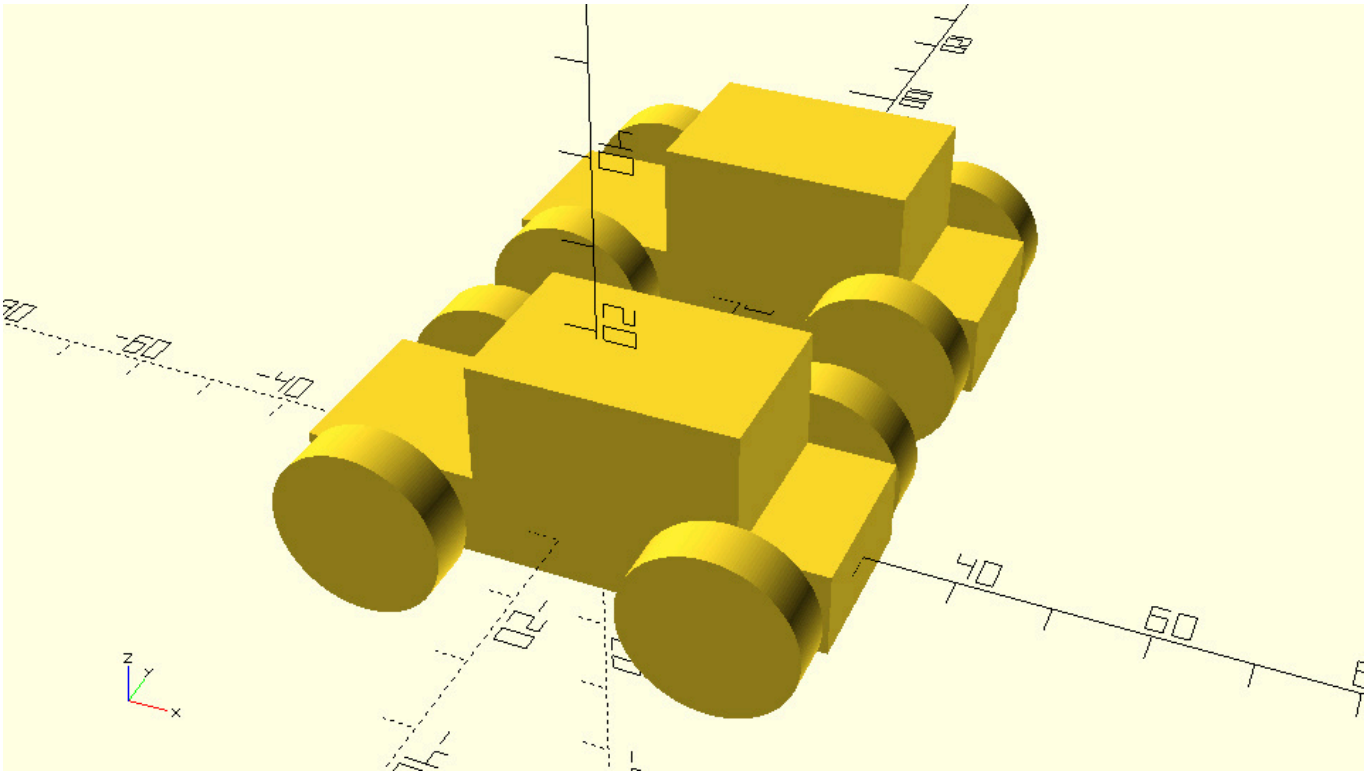
```
use <vehicle_parts.scad>
$fa=1;
$fs=0.4;

// Variables
track = 30;
wheelbase=40;
// Body
body();
// Front left wheel
translate([-wheelbase/2,-track/2,0])
    rotate([0,0,0])
    simple_wheel();
 // Front right wheel
translate([-wheelbase/2,track/2,0])
    rotate([0,0,0])
    simple_wheel();
// Rear left wheel
translate([wheelbase/2,-track/2,0])
    rotate([0,0,0])
    simple_wheel();
// Rear right wheel
translate([wheelbase/2,track/2,0])
    rotate([0,0,0])
    simple_wheel();
// Front axle
translate([-wheelbase/2,0,0])
    axle(track=track);
// Rear axle
translate([wheelbase/2,0,0])
    axle(track=track);
```



Exercise

Take the above car example and modify it in order to create another car. To avoid duplicating the code that creates the car you should create a car module. The module should have two input parameters, the track and the wheelbase of the car. The default value of the track and the wheelbase should be 30 and 40 units respectively. The first car should be positioned at the origin as the example above, the second car should be translated along the positive direction of the Y axis by 50 units.

Code
[Expand]



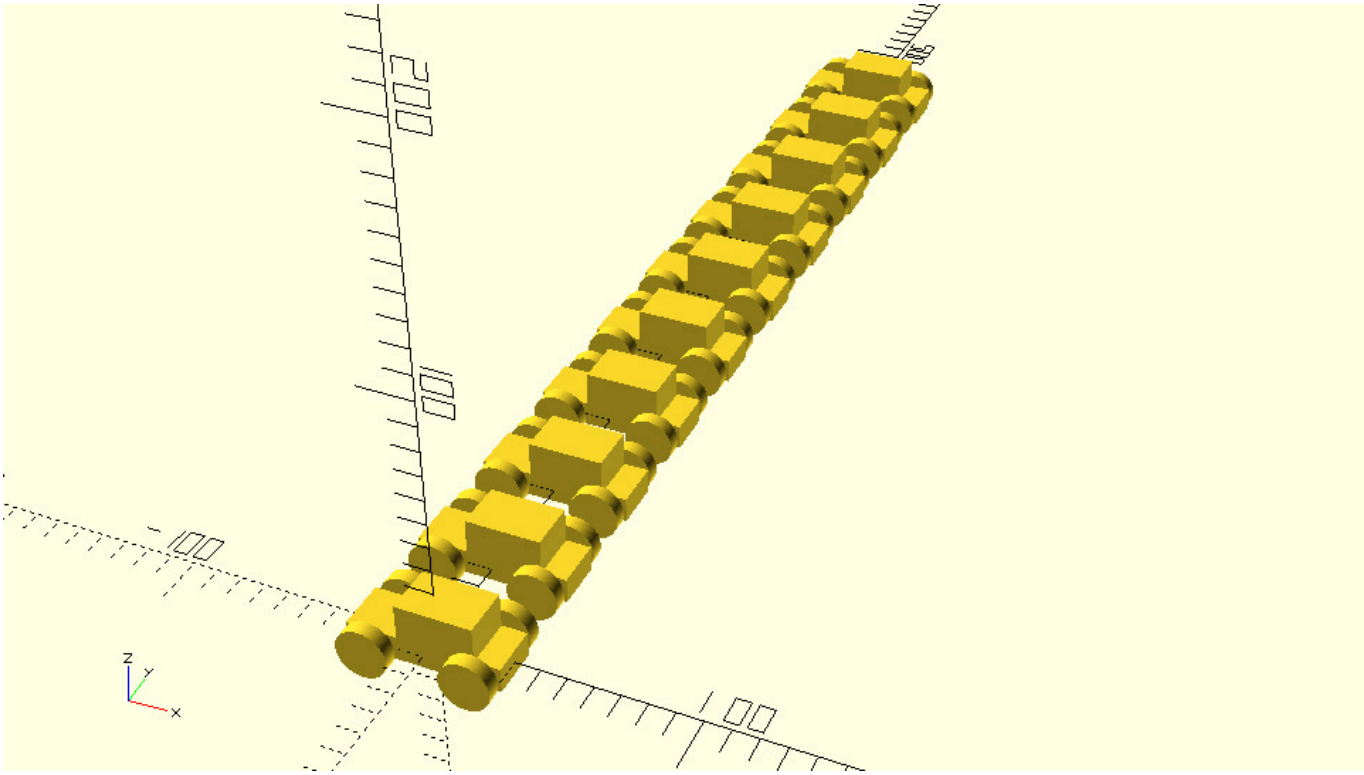
Exercise

Create eight additional cars along the positive direction of the Y axis so that you have ten cars in total. Every next car should be translated 50 units along the positive direction of the Y axis in comparison to the previous car.

Code[Collapse]

row_of_ten_cars_along_y_axis.scad

```
…
car();
translate([0,50,0])
    car();
translate([0,100,0])
    car();
translate([0,150,0])
    car();
translate([0,200,0])
    car();
translate([0,250,0])
    car();
translate([0,300,0])
    car();
translate([0,350,0])
    car();
translate([0,400,0])
    car();
translate([0,450,0])
    car();
…
```

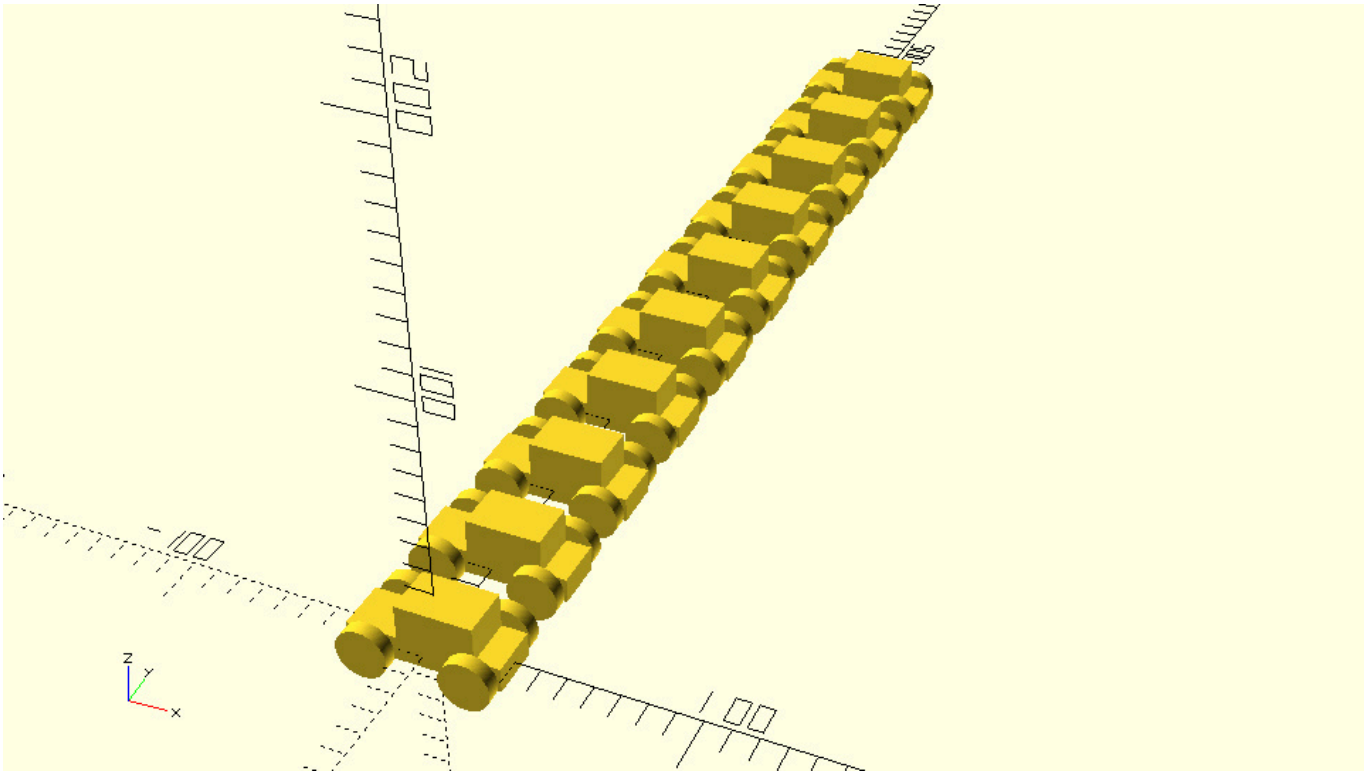


After the previous exercise, you have probably realized that creating a pattern of cars in this manner is not very efficient; a new statement has to be written for every car, and this results in a lot of code duplication in your script. Typically, you can use a for loop to achieve the same results a lot easier. The for loop provides a way to repeat the same set of statements a certain number of times, with small, predictable changes applied each time. Take a look at the following example.

Code

row_of_ten_cars_along_y_axis_with_for_loop.scad

```
…
for (dy=[0:50:450]) {
    translate([0,dy,0])
        car();
}
…
```



There are a few things you should notice about the syntax of the for loop. First the keyword for is typed out followed by a pair of parentheses. Inside the parentheses the variable of the for loop is defined. It's advisable to give a descriptive name to the for loop variables when applicable. In this case the variable is named dy because it represents the number of units that each car needs to be translated along the Y axis. The definition of the variable indicates that its first value will be 0 units and all following values will be incremented by 50 units each until the value 450 is reached. This means that the variable dy will take ten different values in total throughout the execution of the for loop repetitions. These values are 0, 50, 100, 150, 200, 250, 300, 350, 400 and 450. These values form a vector, which in contrast to a single value is a sequence of values. In the first repetition the variable will take the first value of the vector, which is 0. In the second repetition the second value, which is 50. And so forth. The different consecutive values that the for loop variable takes throughout the repetitions of the for loop is the key concept which makes the for loop suitable for creating patterns of multiple parts or models. Finally, after the closing parenthesis follows a pair of curly brackets. Inside the curly brackets exist the statements that will be executed repeatedly as many times as the number of values of the for loop variable. In this case the single statement inside the curly brackets will be executed 10 times which is the number of values that the dy variable will take. To avoid creating 10 cars that are completely overlapping the amount of translation along the Y axis on each repetition of the for loop is parameterized using the dy variable. The dy variable has a different value on each repetition of the for loop thus creating the desired pattern.

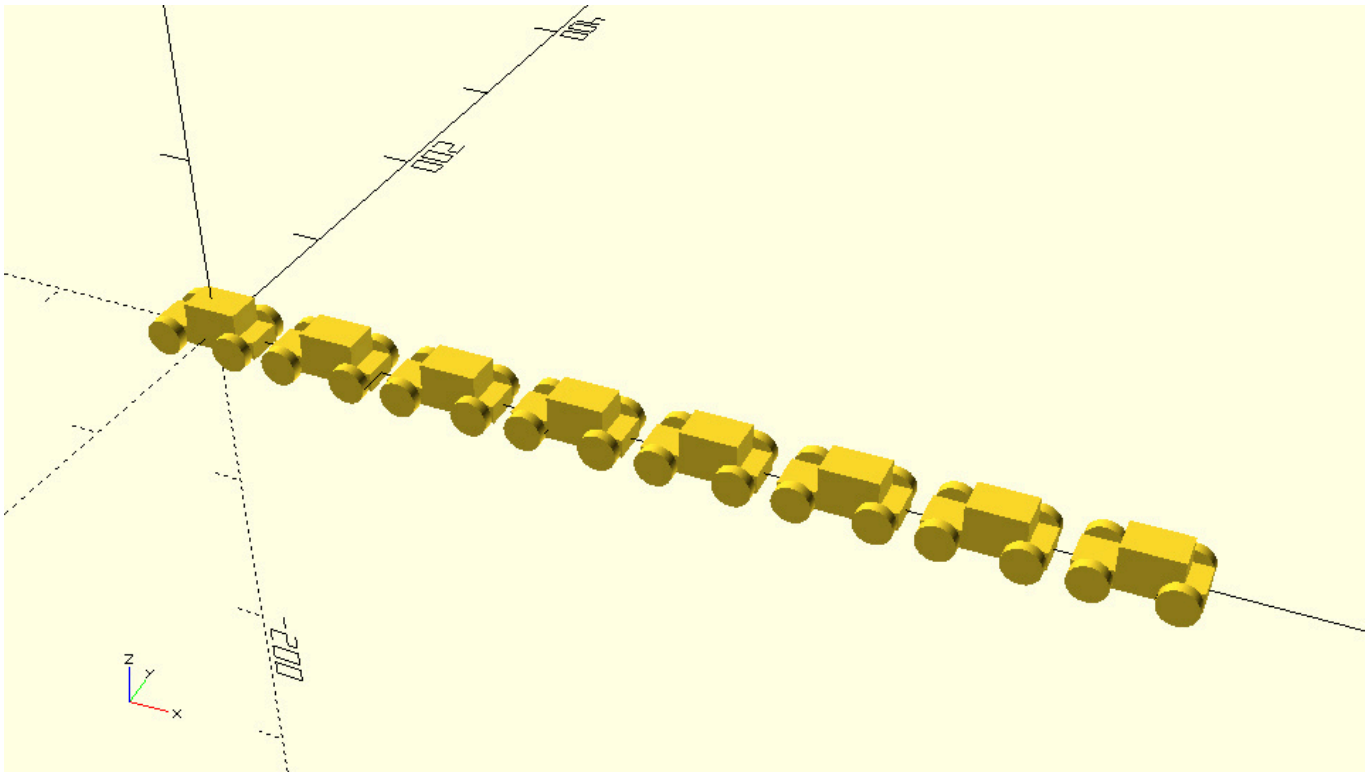
Exercise

Use a for loop to create a pattern of cars. The first car should be centered at the origin. Every next car should be placed behind the previous car. Specifically, every next car should be translated 70 units along the positive direction of the X axis in comparison to the previous car. The pattern should be consisted out of 8 cars in total. The for loop variable should be named dx.

Code[Collapse]

row_of_eight_cars_along_x_axis.scad

```
…
for (dx=[0:70:490]) {
    translate([dx,0,0])
        car();
}
…
```



Creating more complex patterns

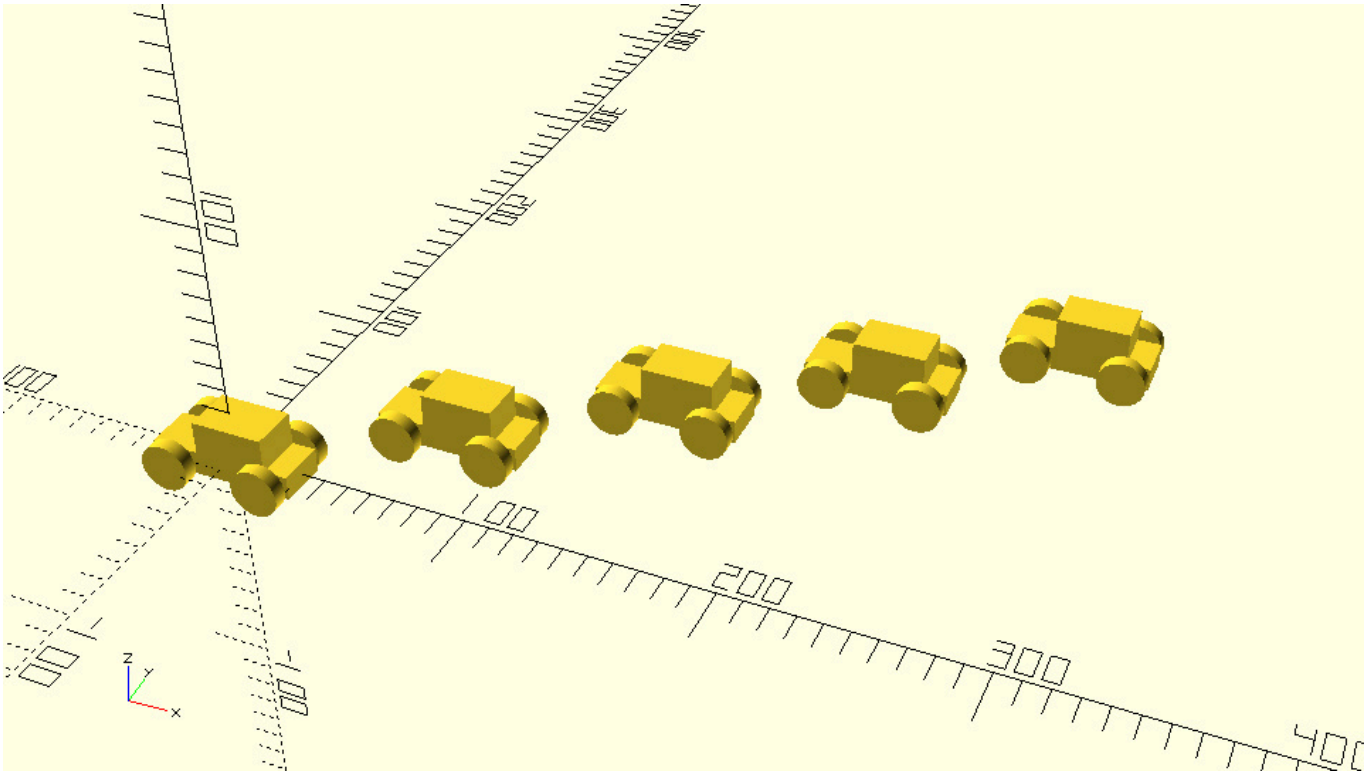
In the previous examples the for loop variable dy was used directly to modify some aspect of each individual model that composes the pattern. The only aspect that was modified was the translation of each model along the Y or X axis. On each repetition the value of the dy variable was equal to the desired translation on each model.

When more than one aspect of the model needs to be modified it's a better practice for the for loop variable to take integer values 0, 1, 2, 3 etc. The required values for modifying different aspects of the model (ex. translation along some axis, scaling of some part) are then produced from those integer values that the for loop variable takes. In the following example this concept is used to simultaneously translate each car 50 and 70 units along the positive direction of the Y and X axis.

Code

diagonal_row_of_five_cars.scad

```
…
for (i=[0:1:4]) {
    translate([i*70,i*50,0])
        car();
}
…
```



There are a few things you should notice. The for loop variable is now named i. When the for loop variable is used in this way it's usually called index and given the name i. Since the for loop variable takes integer values you need to multiply it by a proper number to produce the desired amount of translation along each axis. Specifically, the desired amount of translation along the Y axis is produced by multiplying the for loop variable by 50. Similarly the desired amount of translation along the X axis is produced by multiplying the for loop variable by 70.

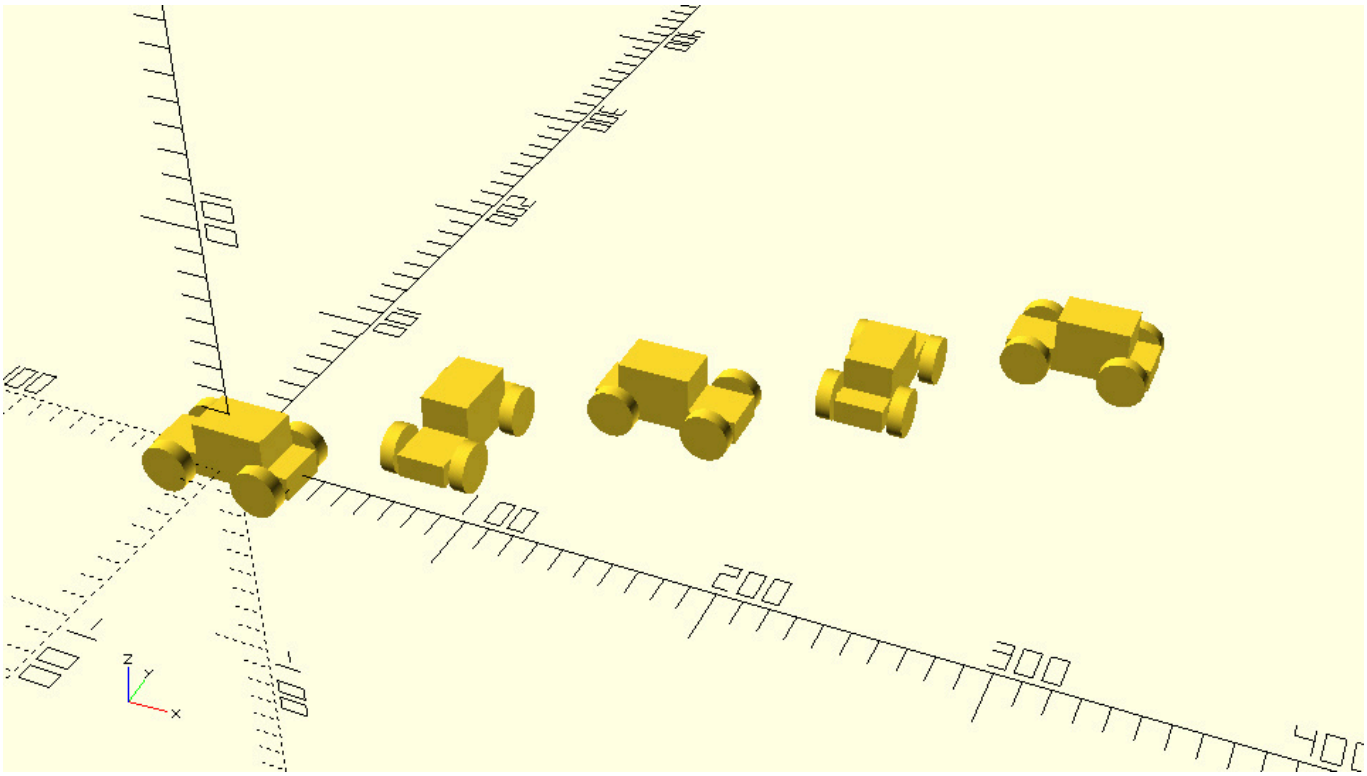
Exercise

Add a rotation transformation to the previous example in order to turn the car around the Z axis. The first car shouldn't be turned at all. Each next car should be turned by 90 degrees around the positive direction of rotation of the Z axis in comparison to the previous car. The positive direction of rotation of the Z axis is the one that would rotate the X axis towards the Y axis. Does the rotation transformation need to be applied before or after the translation transformation in order to keep the cars in the same position?

Code[Collapse]

diagonal_row_of_five_twisted_cars.scad

```
…
for (i=[0:1:4]) {
    translate([i*70,i*50,0])
        rotate([0,0,i*90])
        car();
}
…
```

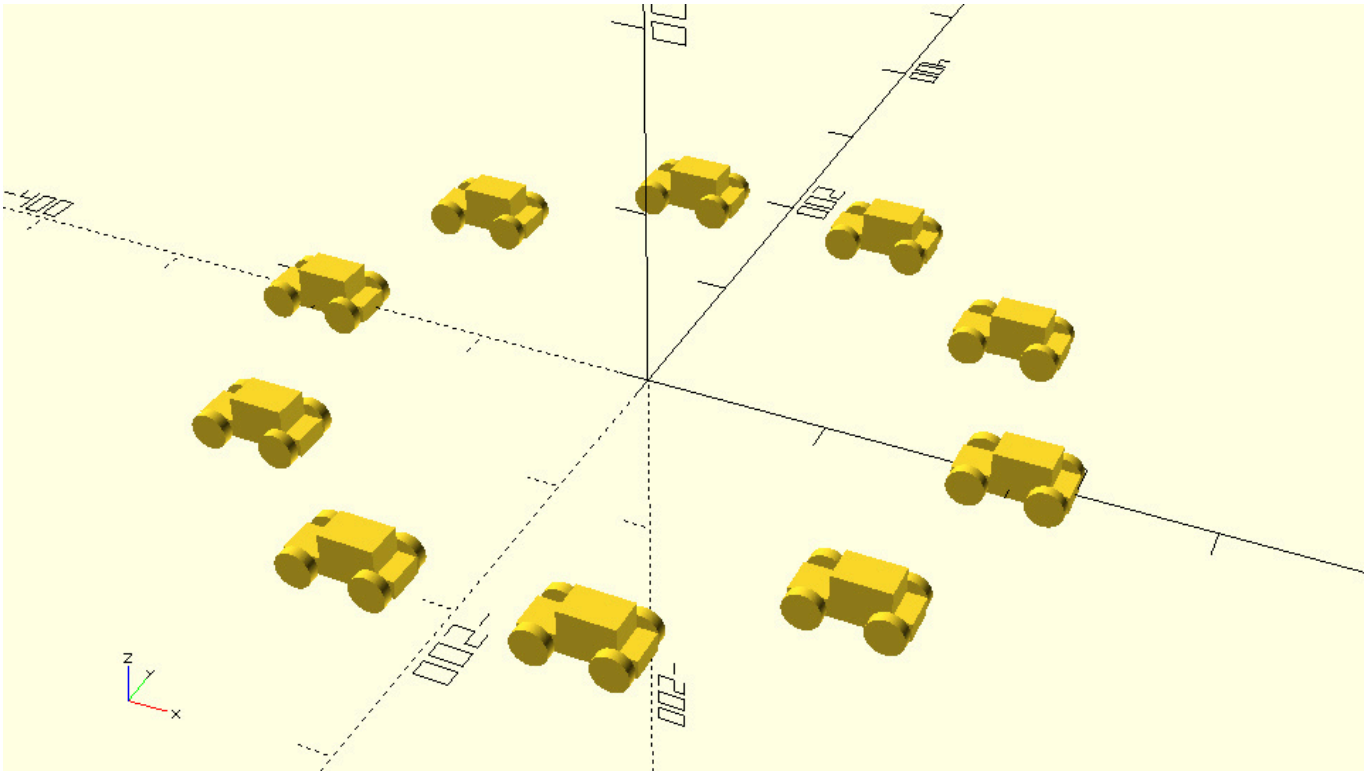


The patterns you are creating are already becoming cooler, here is an interesting one.

Code

circular_pattern_of_ten_cars.scad

```
…
r = 200;
for (i=[0:36:359]) {
    translate([r*cos(i),r*sin(i),0])
        car();
}
…
```



In the above pattern the cars have been placed at equally spaced points at the circumference of a perfect circle that has a radius of 200 units. There are a few important points you should pay attention to if you wish to create such patterns.

The first is that in order to create a circular pattern you need to use polar coordinates. Depending on your background you may have noticed the use of polar coordinates just by glancing at the code or you may have no idea what it is. In the latter case, the only thing that you need to know is that polar coordinates are a way to produce the X and Y coordinates of a given point of a circle when you know the radius of the circle and the angle that corresponds to that point. The angle 0 corresponds to the point of the circle that belongs to the positive direction X axis. The positive counting direction of the angle is from the X to the Y axis. This means the positive Y axis corresponds to 90 degrees, the negative X axis to 180 degrees, the negative Y axis to 270 degrees and if you complete the circle the positive X axis to 360 degrees. According to the polar coordinates the X coordinate can be calculated by multiplying the radius of the circle by the cosine of the angle, while the Y coordinate can be calculated by multiplying the radius of the circle by the sine of the angle. This is how the desired amount of translation along the X and Y axis is produced.

The second thing you should notice is what values the for loop variable i takes. The variable i starts from 0 and is incremented by 36 on each repetition in order to position 10 equally spaced cars on the circle (360/10 = 36). The first car that is created at 0 angle and the car that correspond to 360 degrees would be exactly overlapping. In order to avoid this, you need to instruct the for loop variable to stop incrementing before it reaches 360. If you are lazy calculating 360 - 36 = 324, you can just put the limit at 359. This will work fine because the for loop variable will only take the values 0, 36, 72, 108 144, 180, 216, 252, 288 and 324, since incrementing by another 36 units would result in 360 which exceeds 359.

By using additional variables and naming them properly you can make your scripts more descriptive and usable so that it's easier for anyone (or even you at a later point in time) to understand what they are doing and how to use them. For example, the previous script can take the following form.

Code

```
…
r = 200; // pattern radius
n = 10; // number of cars
step = 360/n;
for (i=[0:step:359]) {
    angle = i;
    dx = r*cos(angle);
    dy = r*sin(angle);
    translate([dx,dy,0])
        car();
}
…
```

On the above script it is self-explanatory that the for loop variable i corresponds to the angle. It is also clearer what the amount of translation along each axis is. In addition, it is easy to customize this pattern by changing the radius and/or the number of cars.

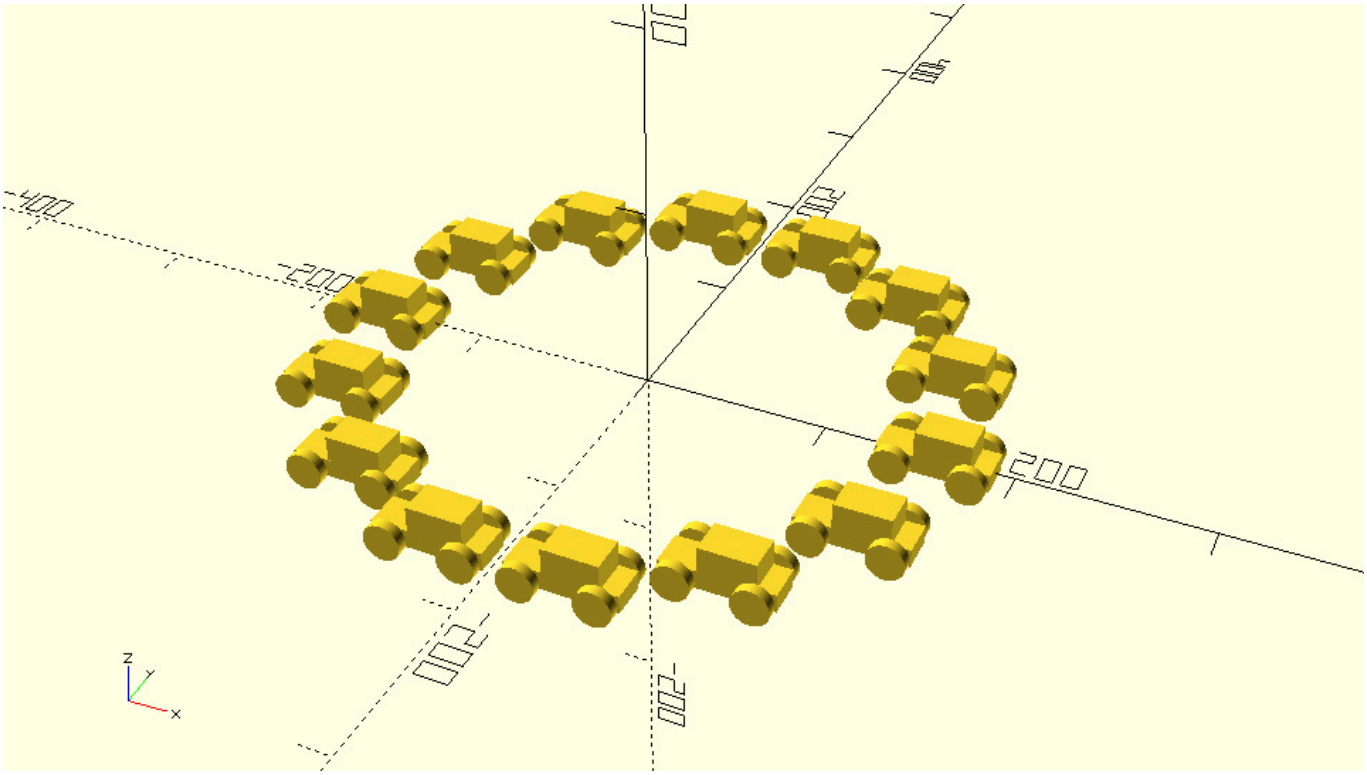
Exercise

Modify the appropriate values on the above script in order to create a pattern of 14 equally spaced cars on the circumference of circle with radius of 160 units.

Code[Collapse]

parametric_circular_pattern_of_fourteen_cars.scad

```
…
r = 160; // pattern radius
n = 14; // number of cars
step = 360/n;
for (i=[0:step:359]) {
    angle = i;
    dx = r*cos(angle);
    dy = r*sin(angle);
    translate([dx,dy,0])
        car();
}
…
```



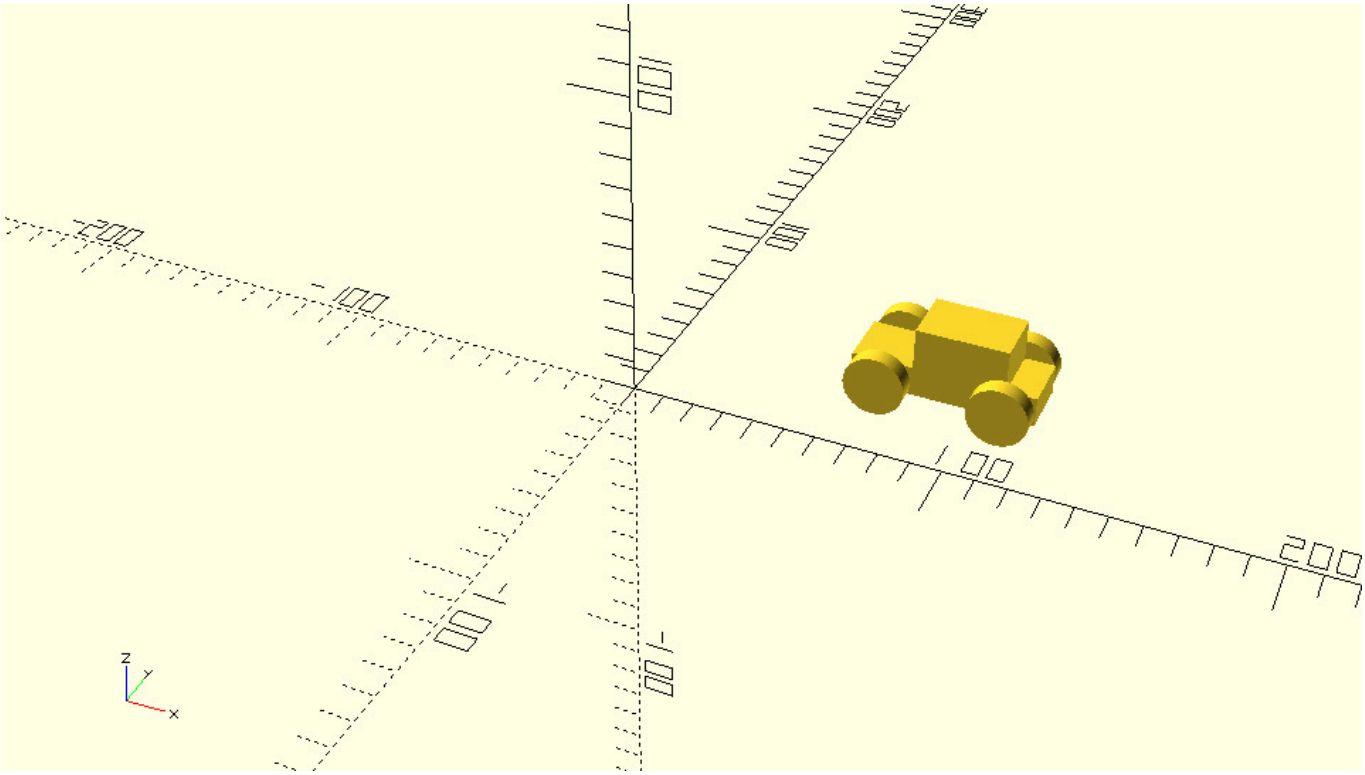
Exercise

If you are not familiar with polar coordinates play around with the following script. Try assigning different values to the radius and the angle variables and see what the resulting position of the car is.

Code[Collapse]

car_positioned_with_polar_coordinates.scad

```
…
radius = 100;
angle = 30; // degrees
dx = radius*cos(angle);
dy = radius*sin(angle);
translate([dx,dy,0])
    car();
…
```



Challenge

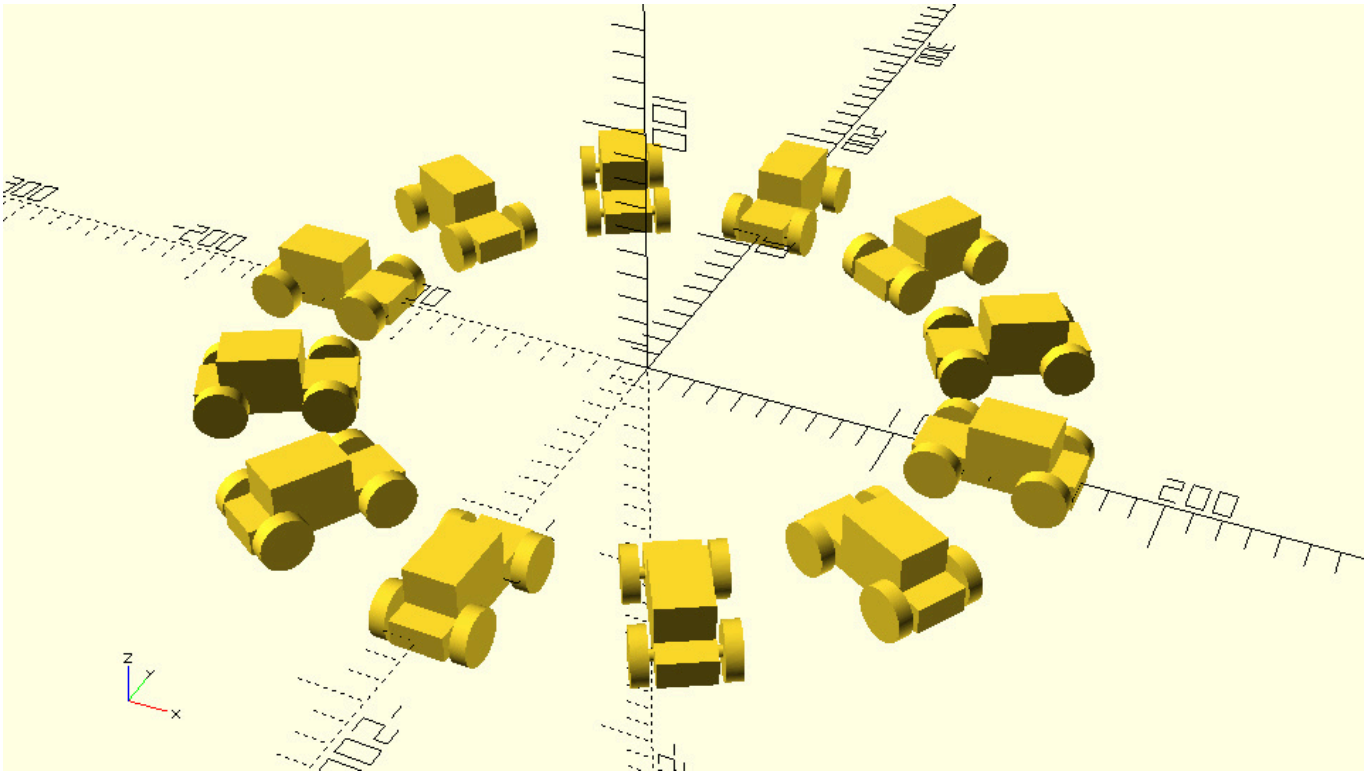
Code

parametric_circular_pattern_of_fourteen_cars.scad

```
…
r = 160; // pattern radius
n = 14; // number of cars
step = 360/n;
for (i=[0:step:359]) {
    angle = i;
    dx = r*cos(angle);
    dy = r*sin(angle);
    translate([dx,dy,0])
        car();
}
…
```

Exercise

The above script was used to create a circular pattern of cars. Modify the above script by adding a rotation transformation in order to make all car face the origin. Use your modified script to create a pattern that has 12 cars and a radius of 140 units.



Code[Collapse]

cars_facing_at_the_origin.scad

```
…
r = 140; // pattern radius
n = 12; // number of cars
step = 360/n;
for (i=[0:step:359]) {
    angle = i;
    dx = r*cos(angle);
    dy = r*sin(angle);
    translate([dx,dy,0])
        rotate([0,0,angle])
        car();
}
…
```

Exercise

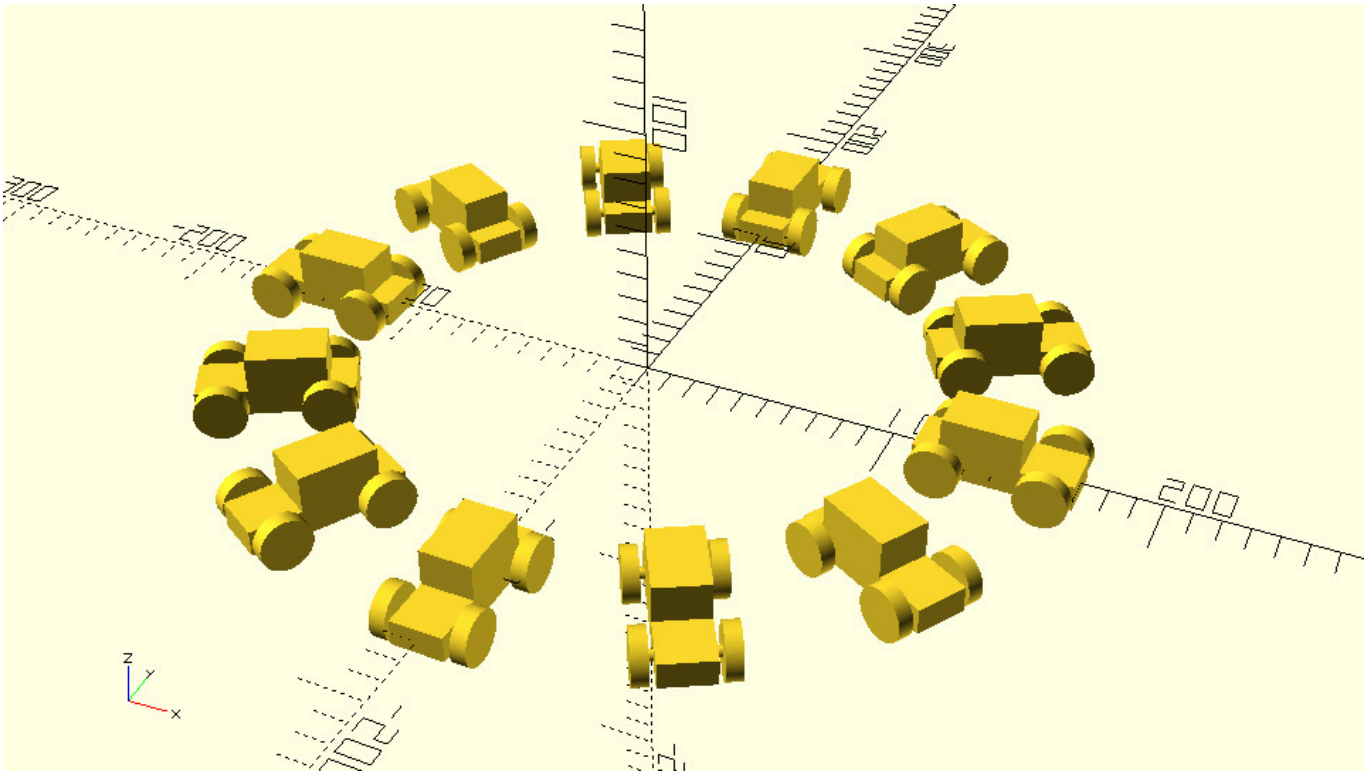
Make appropriate changes to the above script to create: i) one pattern where all cars are facing away from the origin ii) one pattern where all cars are oriented tangentially as if driving counter clockwise around a circle iii) and one as if driving clockwise around a circle.

- facing away from the origin

Code[Collapse]

cars_facing_away_from_the_origin.scad

```
…
translate([dx,dy,0])
    rotate([0,0,angle - 180])
    car();
…
```

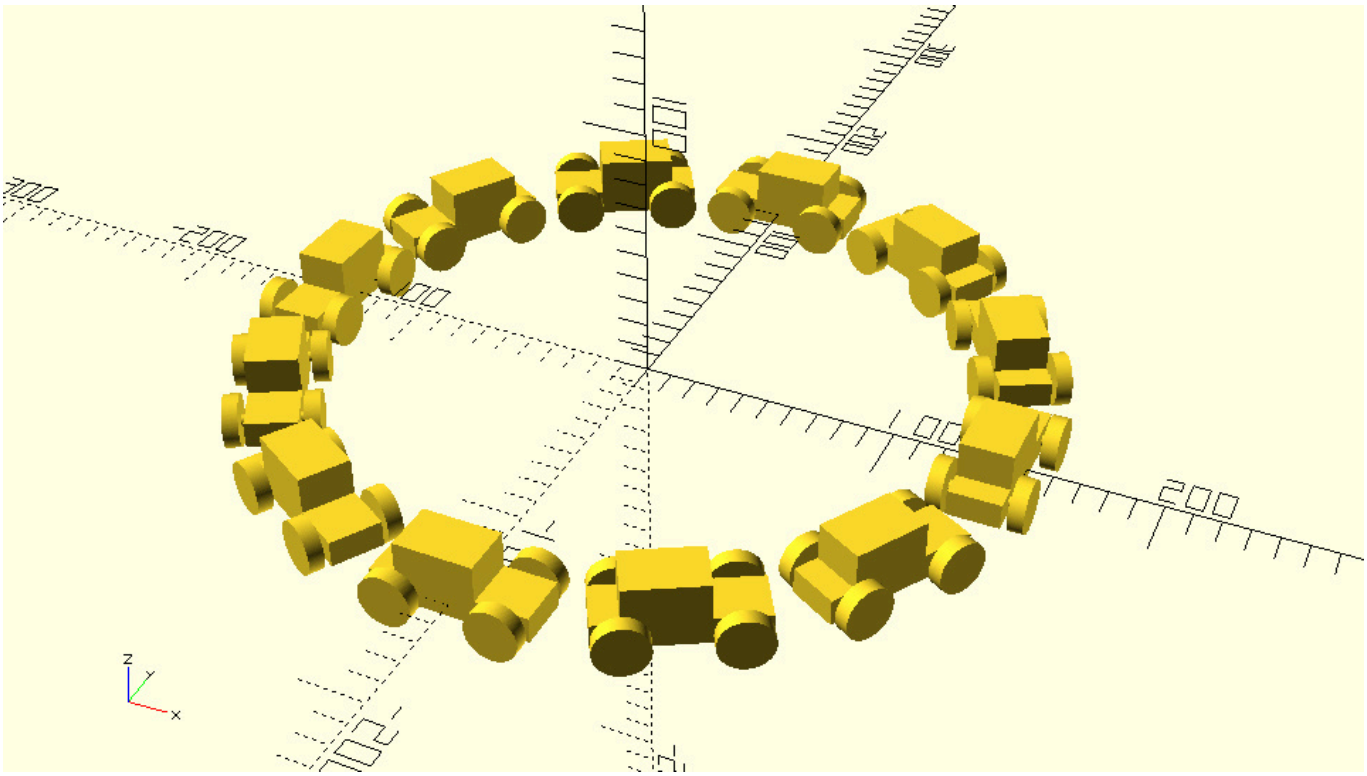


- driving counterclockwise

Code[Collapse]

cars_driving_counter_clockwise.scad

```
…
translate([dx,dy,0])
    rotate([0,0,angle - 90])
    car();
…
```

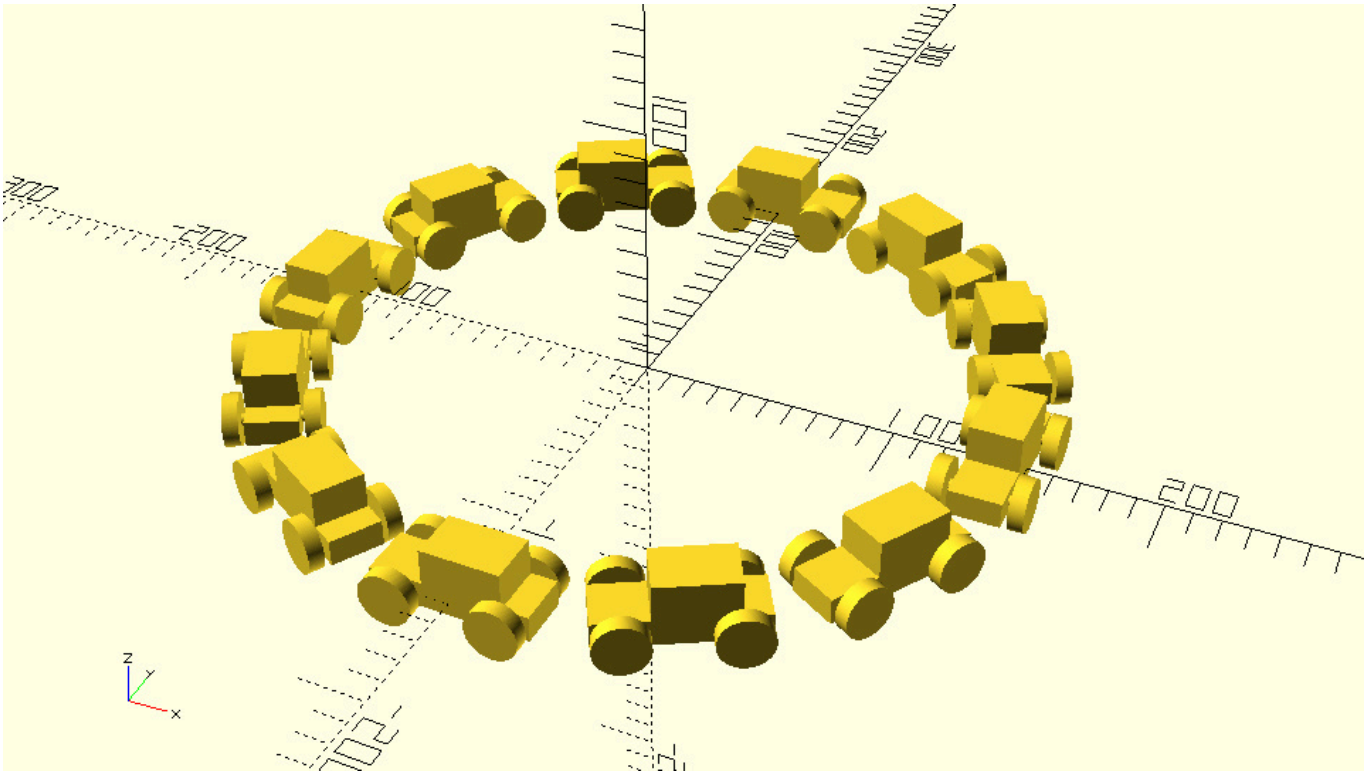


- driving clockwise

Code[Collapse]

cars_driving_clockwise.scad

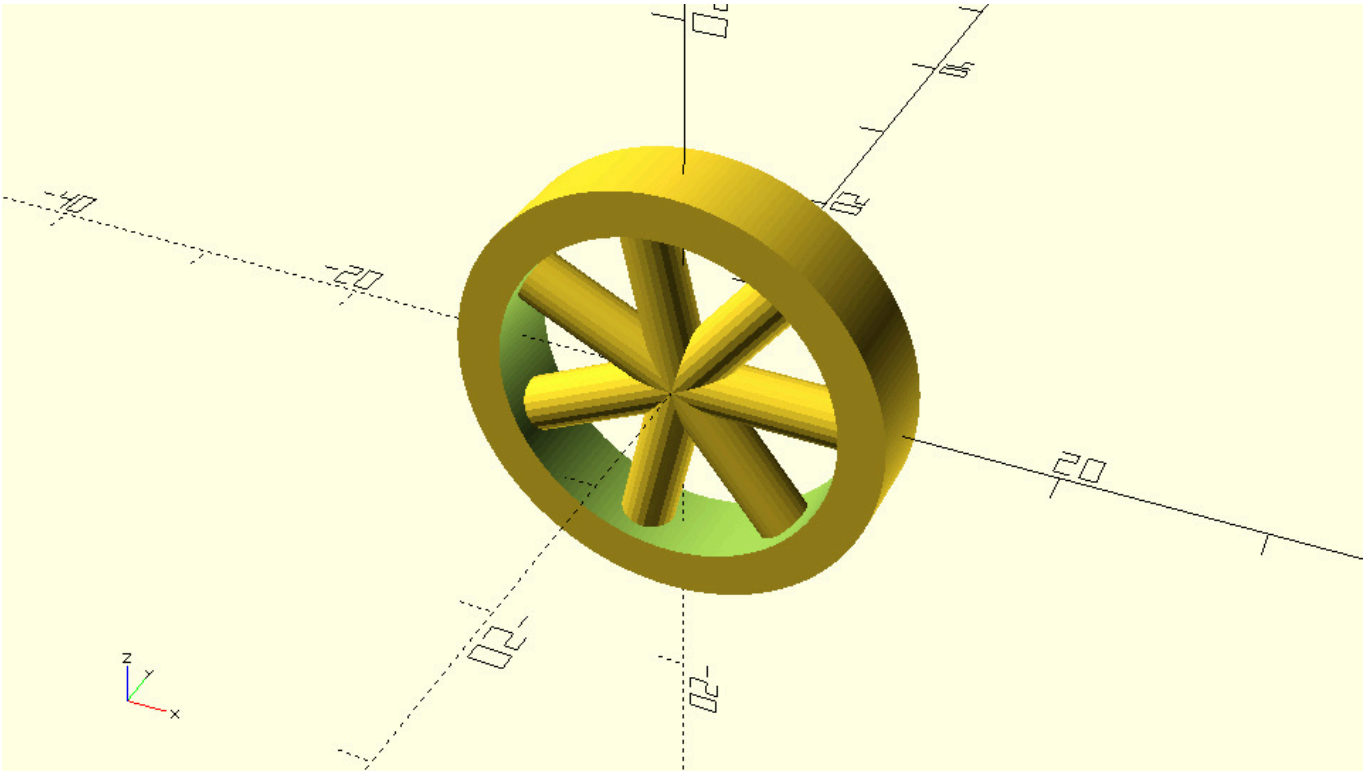
```
…
translate([dx,dy,0])
    rotate([0,0,angle - 270])
    car();
…
```



Now that you are getting a hold of using for loops to create patterns it's time to put you new skills in the development of a more sophisticated wheel design!

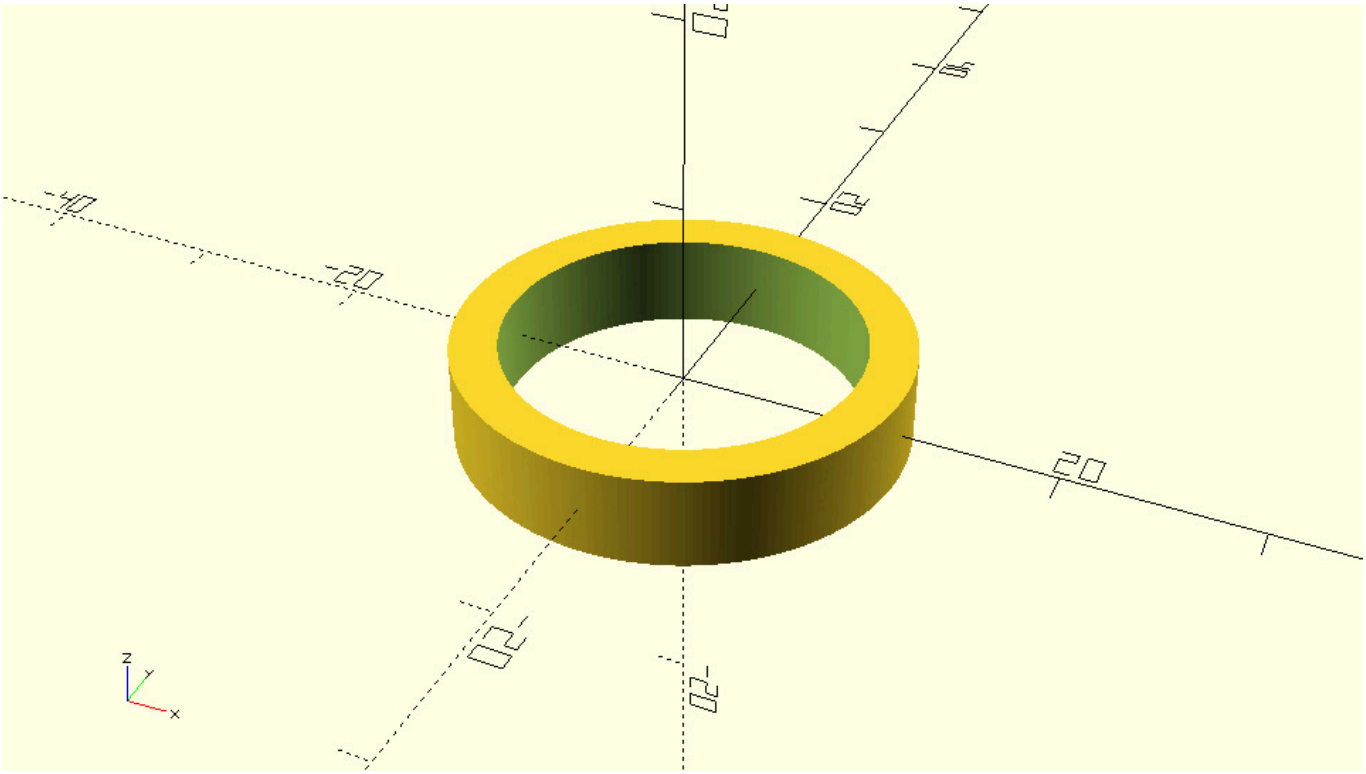
Exercise

If you feel confident with your OpenSCAD skills or if you would like to experiment a bit more, try to build a new module named spoked_wheel that creates the following wheel design. If you would like some more guidance with creating this module go through the following exercises instead.



Exercise

If you feel more comfortable with some additional guidance that's fine. Create a new module named spoked_wheel that has 5 input parameters. The input parameters should be named radius, width, thickness, number_of_spokes and spoke_radius. Give these variables the default values of 12, 5, 5, 7 and 1.5 respectively. Use the cylinder and difference commands to create the ring of the wheel by subtracting a small cylinder from a bigger one. The model that you need to create can be seen on the following image. For this step you are only going to use the radius, width and thickness variables. Remember that when you are subtracting one object from another it needs to clear the face of the other object to avoid any errors. Keep this in mind when defining the height of the small cylinder. You will also need to calculate the radius of the small cylinder from the radius and thickness variables. You can use a variable named inner_radius to store the result of the appropriate calculation and then use it to define the radius of the smaller cylinder.



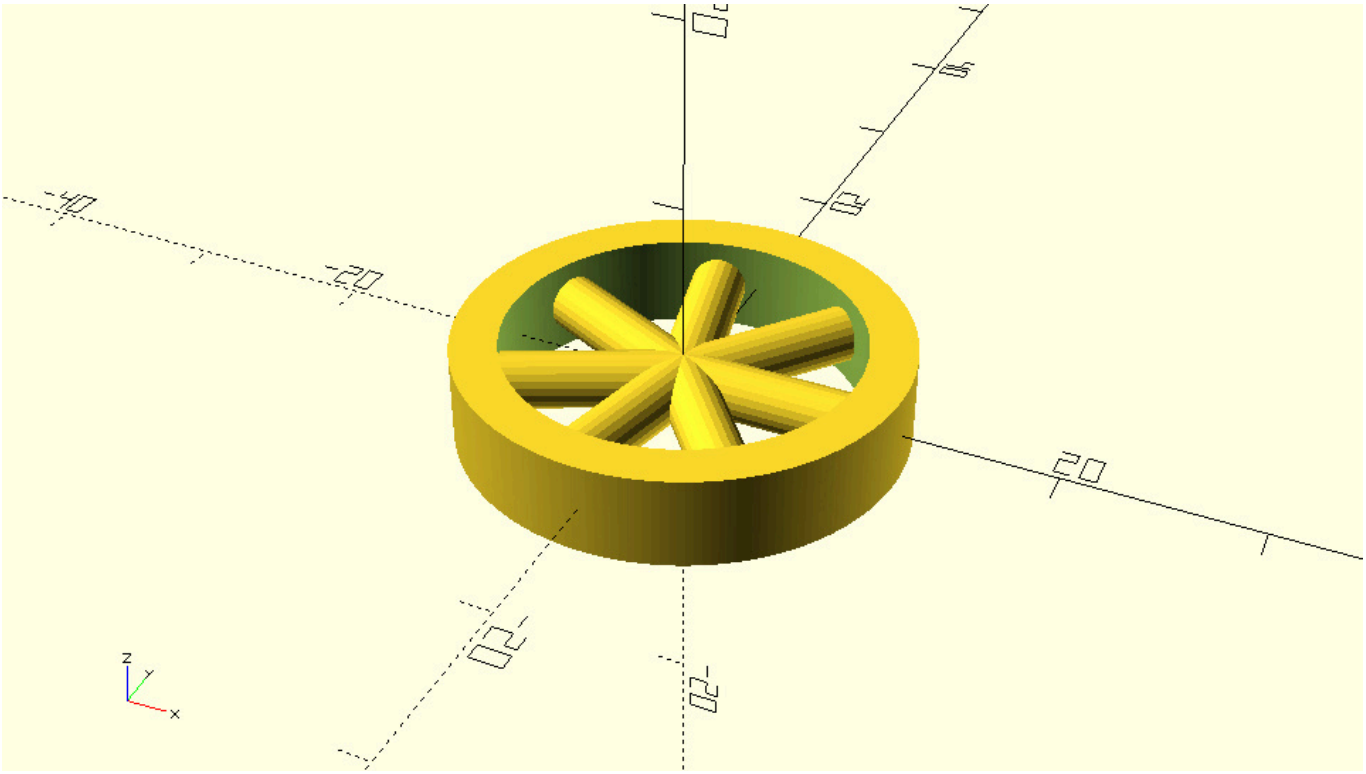
Code[Collapse]

ring_of_spoked_wheel.scad

```
…
module spoked_wheel(radius=12, width=5, thickness=5, number_of_spokes=7, spoke_radius=1.5) {

    // Ring
    inner_radius = radius - thickness/2;
    difference() {
        cylinder(h=width,r=radius,center=true);
        cylinder(h=width + 1,r=inner_radius,center=true);
    }
}
spoked_wheel();
…
```

Exercise

Extend the previous module to additionally create the spokes of the wheel as seen on the following image. The spokes of the wheel need to be cylindrical. The length of the spokes needs to be appropriate so that each spoke spans from the center of the ring to its half thickness. You will have to use a for loop to create the spokes as a pattern. Feel free to review previous for loop example that can help you with this.



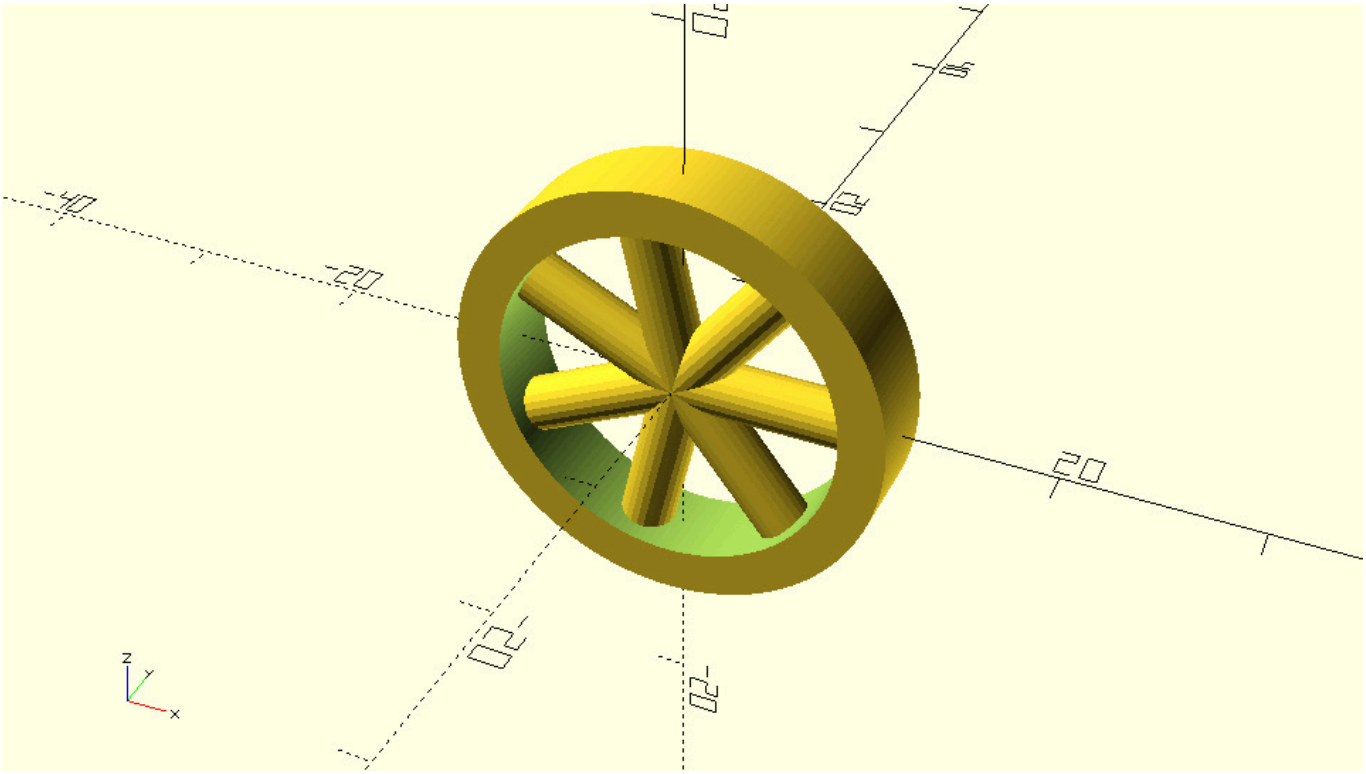
Code[Collapse]

spoked_wheel_horizontal.scad

```
…
module spoked_wheel(radius=12, width=5, thickness=5, number_of_spokes=7, spoke_radius=1.5) {

    // Ring
    inner_radius = radius - thickness/2;
    difference() {
        cylinder(h=width,r=radius,center=true);
        cylinder(h=width + 1,r=inner_radius,center=true);
    }

    // Spokes
    spoke_length = radius - thickness/4;
    step = 360/number_of_spokes;
    for (i=[0:step:359]) {
        angle = i;
        rotate([0,90,angle])
            cylinder(h=spoke_length,r=spoke_radius);
    }
}
spoked_wheel();
…
```

Exercise

For the new wheel design to be compatible with the existing wheel designs and modules that you have created throughout the tutorial, it needs to be rotated to stand straight as in the following image. Add an appropriate rotation transformation to do so.



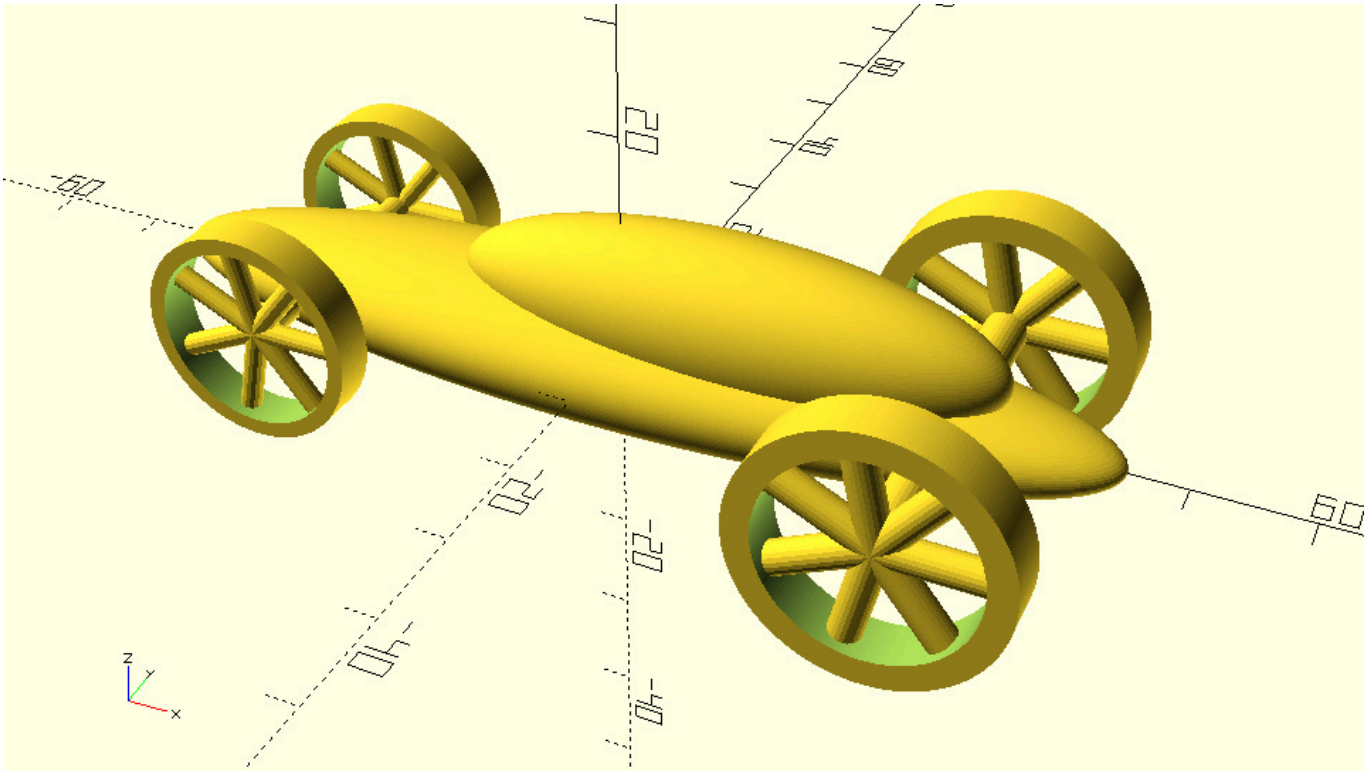
Code[Collapse]

spoked_wheel.scad

```
…
module spoked_wheel(radius=12, width=5, thickness=5, number_of_spokes=7, spoke_radius=1.5) {

    rotate([90,0,0]) {
        // Ring
        inner_radius = radius - thickness/2;
        difference() {
            cylinder(h=width,r=radius,center=true);
            cylinder(h=width + 1,r=inner_radius,center=true);
        }

        // Spokes
        spoke_length = radius - thickness/4;
        step = 360/number_of_spokes;
        for (i=[0:step:359]) {
            angle = i;
            rotate([0,90,angle])
                cylinder(h=spoke_length,r=spoke_radius);
        }
    }
}
…
spoked_wheel();
…
```

Exercise

Add the spoked_wheel module on the vehicle_parts.scad script. Use the new wheel design in one of your car models. If you don't have any ideas you can try replicating the following model.



Code[Collapse]

car_with_spoked_wheels.scad

```
use <vehicle_parts.scad>

$fa = 1;
$fs = 0.4;

front_track = 24;
rear_track = 34;
wheelbase = 60;

front_wheels_radius = 10;
front_wheels_width = 4;
front_wheels_thickness = 3;
front_spoke_radius = 1;

front_axle_radius = 1.5;

// Round car body
resize([90,20,12])
    sphere(r=10);
translate([10,0,5])
    resize([50,15,15])sphere(r=10);

// Wheels
translate([-wheelbase/2,-front_track/2,0])
    spoked_wheel(radius=front_wheels_radius, width=front_wheels_width,
thickness=front_wheels_thickness, spoke_radius=front_spoke_radius);
translate([-wheelbase/2,front_track/2,0])
    spoked_wheel(radius=front_wheels_radius, width=front_wheels_width,
thickness=front_wheels_thickness, spoke_radius=front_spoke_radius);
translate([wheelbase/2,-rear_track/2,0])
    spoked_wheel();
translate([wheelbase/2,rear_track/2,0])
    spoked_wheel();

// Axles
translate([-wheelbase/2,0,0])
    axle(track=front_track, radius=front_axle_radius);
translate([wheelbase/2,0,0])
    axle(track=rear_track);
```

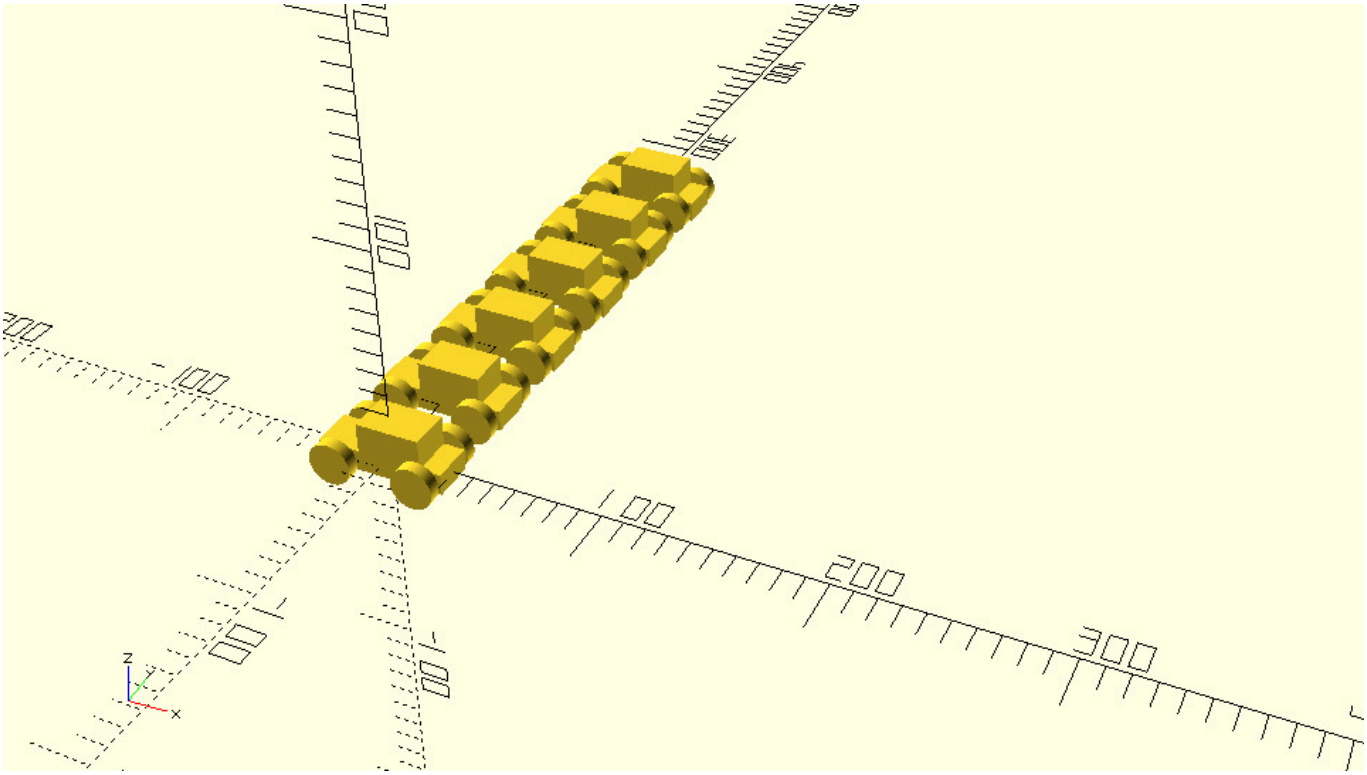
Creating patterns of patterns - Nested for loops

The following script creates a row of cars along the Y axis.

Code

row_of_six_cars_along_y_axis.scad

```
…
n = 6; // number of cars
y_spacing = 50;
for (dy=[0:y_spacing:n*y_spacing-1]) {
    translate([0,dy,0])
        car();
}
…
```



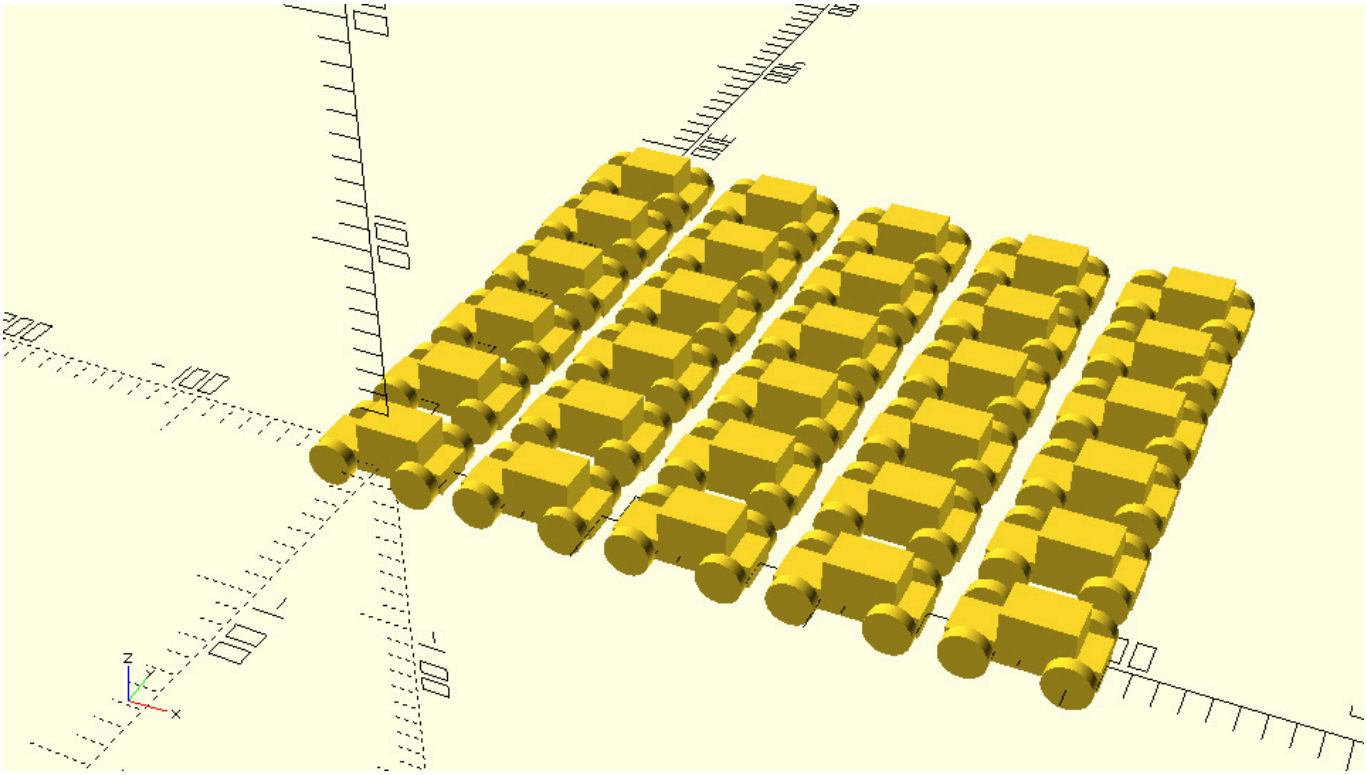
Exercise

Modify the above script to create 4 additional rows of cars. Each row should be translated by 70 units along the positive direction of the X axis in comparison to the previous row.

Code[Collapse]

five_rows_of_six_cars.scad

```
…
n = 6; // number of cars
y_spacing = 50;
for (dy=[0:y_spacing:n*y_spacing-1]) {
    translate([0,dy,0])
        car();
}
for (dy=[0:y_spacing:n*y_spacing-1]) {
    translate([70,dy,0])
        car();
}
for (dy=[0:y_spacing:n*y_spacing-1]) {
    translate([140,dy,0])
        car();
}
for (dy=[0:y_spacing:n*y_spacing-1]) {
    translate([210,dy,0])
        car();
}
for (dy=[0:y_spacing:n*y_spacing-1]) {
    translate([280,dy,0])
        car();
}
…
```

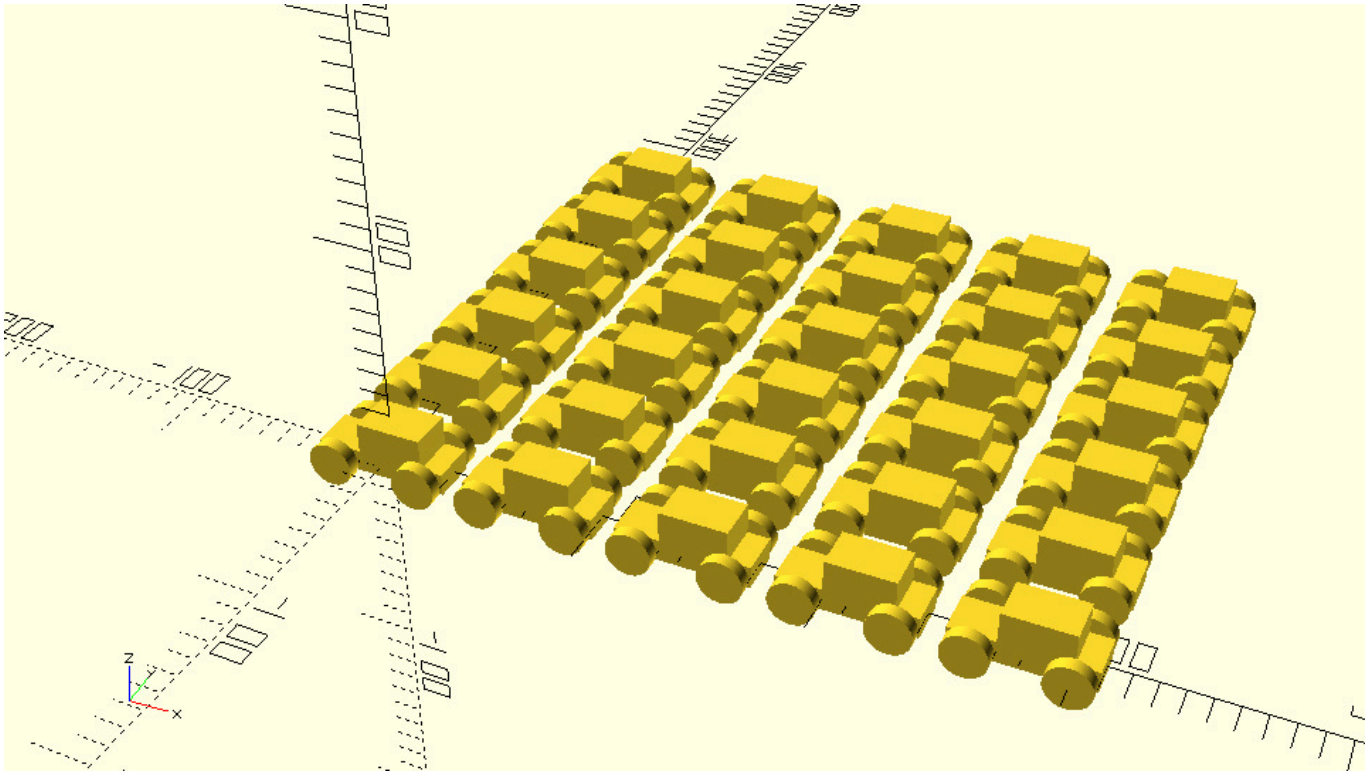


If you have been paying close attention to the tutorial so far, you may have noticed that the script above is not very efficient. It has a lot of code duplication and the number of rows can't be easily modified. You faced a similar situation in the beginning of this chapter when you wanted to create a row of cars. To solve that problem, you wrapped the statement that creates a piece of the pattern (a single car) inside a for loop. This generated the whole pattern (a row of cars) without having to type out a statement for each individual car. The same principle can be applied here. In this case, the repeating pattern will be the row of cars, which itself is a repeating pattern of individual cars. Following the same process as before, the statements that create a row of cars will be placed inside a for loop in order to create the pattern of rows of cars. The result is that a for loop is placed inside of another for loop. For loops that are used in this way are called nested for loops. The following example demonstrates this concept.

Code

five_rows_of_six_cars_with_nested_for_loops.scad

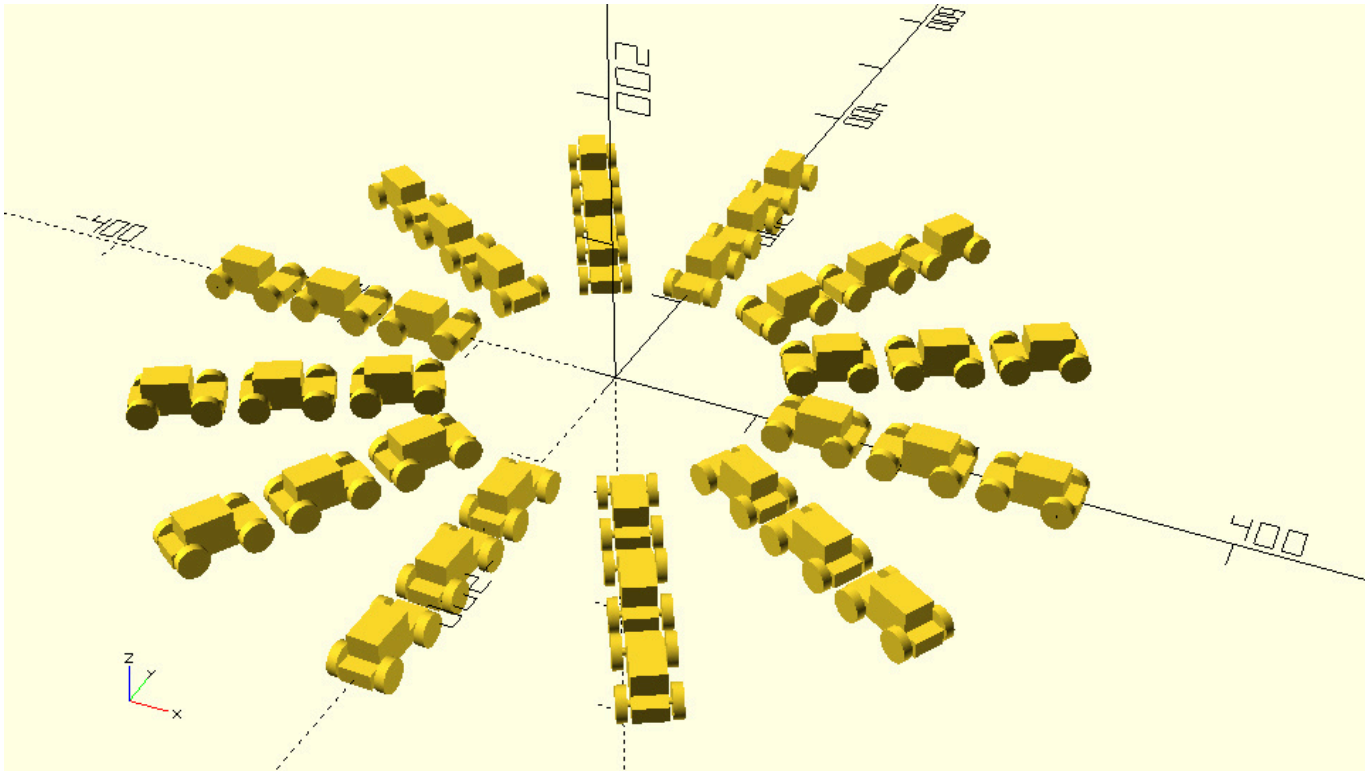
```
…
n_cars = 6;
y_spacing = 50;
n_rows = 5;
x_spacing = 70;
for (dx=[0:x_spacing:n_rows*x_spacing-1]) {
    for (dy=[0:y_spacing:n_cars*y_spacing-1]) {
        translate([dx,dy,0])
            car();
    }
}
…
```



You should notice the following concept. During the first repetition of the outer for loop, all iterations of the inner for loop are executed, thus creating the first row of cars. During the second repetition of the outer for loop all repetitions of the inner for loop are executed, thus creating the second row of cars. And so forth. Each row is positioned by the dx variable, which holds the parameterized translation along the X axis. During each iteration of the outer loop, a new value of dx is used. This value then holds steady while the inner loop executes and modifies the dy value. In this way, a row of cars is generated at each value of dx.

Exercise

Use nested for loops to create three circular patterns of cars similar to the image below. The for loop variable of the outer loop should be used to parameterize the radius of each pattern. The radius of the circular patterns should be 140, 210 and 280 units respectively. Each pattern should be consisted of 12 cars.



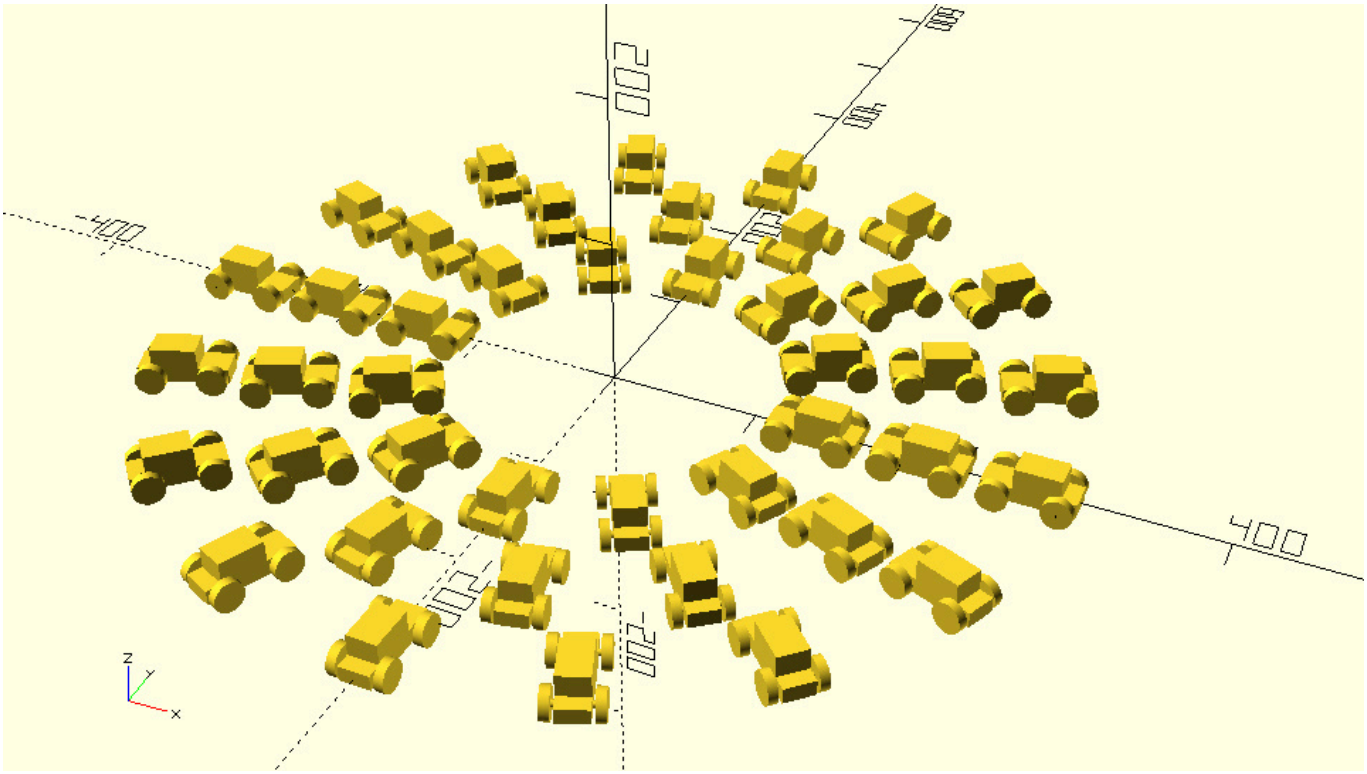
Code[Collapse]

three_circular_patterns.scad

```
…
n = 12; // number of cars
step = 360/n;
for (r=[140:70:280]) {
    for (angle=[0:step:359]) {
        dx = r*cos(angle);
        dy = r*sin(angle);
        translate([dx,dy,0])
            rotate(angle)
            car();
    }
}
…
```

Exercise

Modify the script of the previous exercise so that not only the radius but also the number of cars is different for each pattern. To do so use an index variable i as the variable of the outer loop instead of the variable r that corresponds to the radius. The variable r should be calculated at each repetition of the outer for loop according to the formula r = 70 + i*70. Additionally, on each repetition of the outer for loop the n variable should take different values according to the formula n = 12 + i*2. The step variable also needs to be updated on each repetition of the outer for loop. The i variable should take the values 1, 2 and 3.



Code[Collapse]

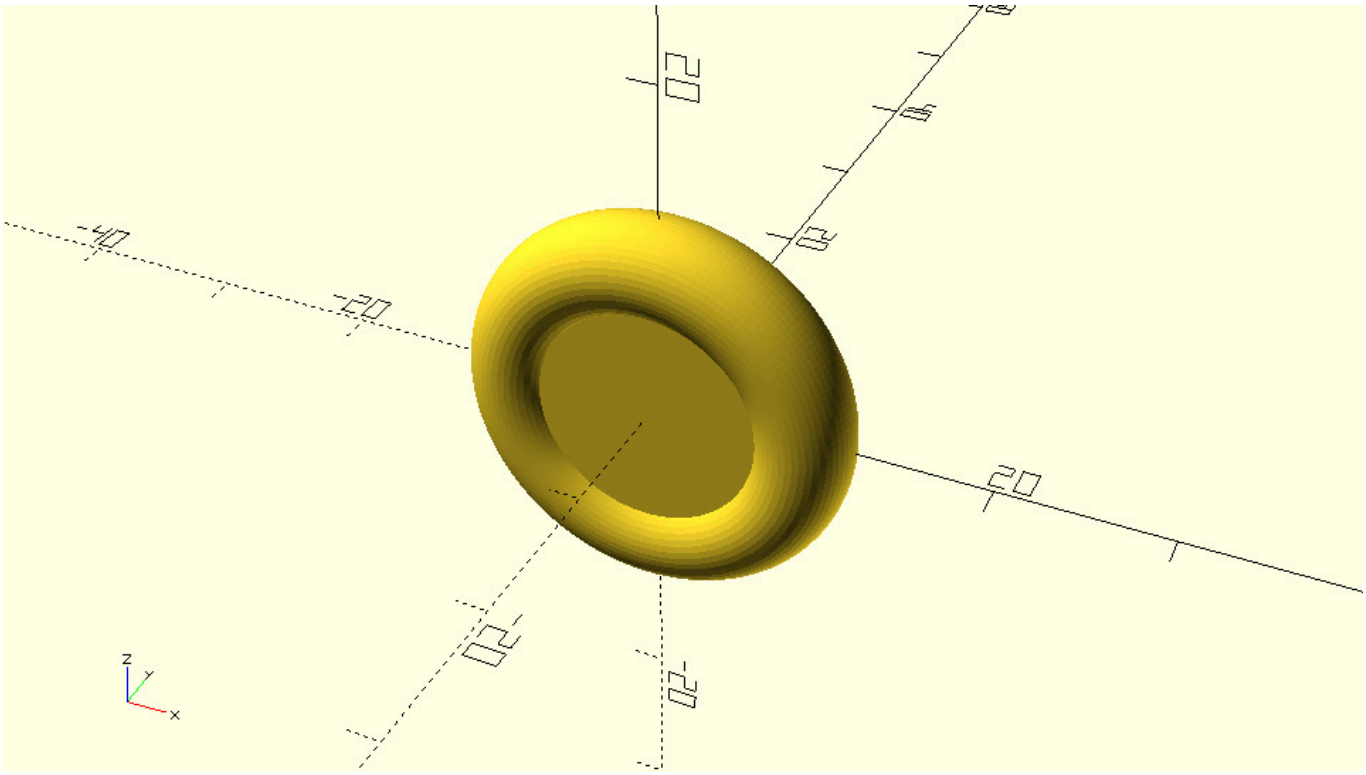
three_circular_patterns_with_increasing_number_of_cars.scad

```
…
for (i=[1:1:3]) {
    r = 70 + i*70;
    n = 12 + i*2;
    step = 360/n;
    for (angle=[0:step:359]) {
        dx = r*cos(angle);
        dy = r*sin(angle);
        translate([dx,dy,0])
            rotate(angle)
            car();
    }
}
…
```

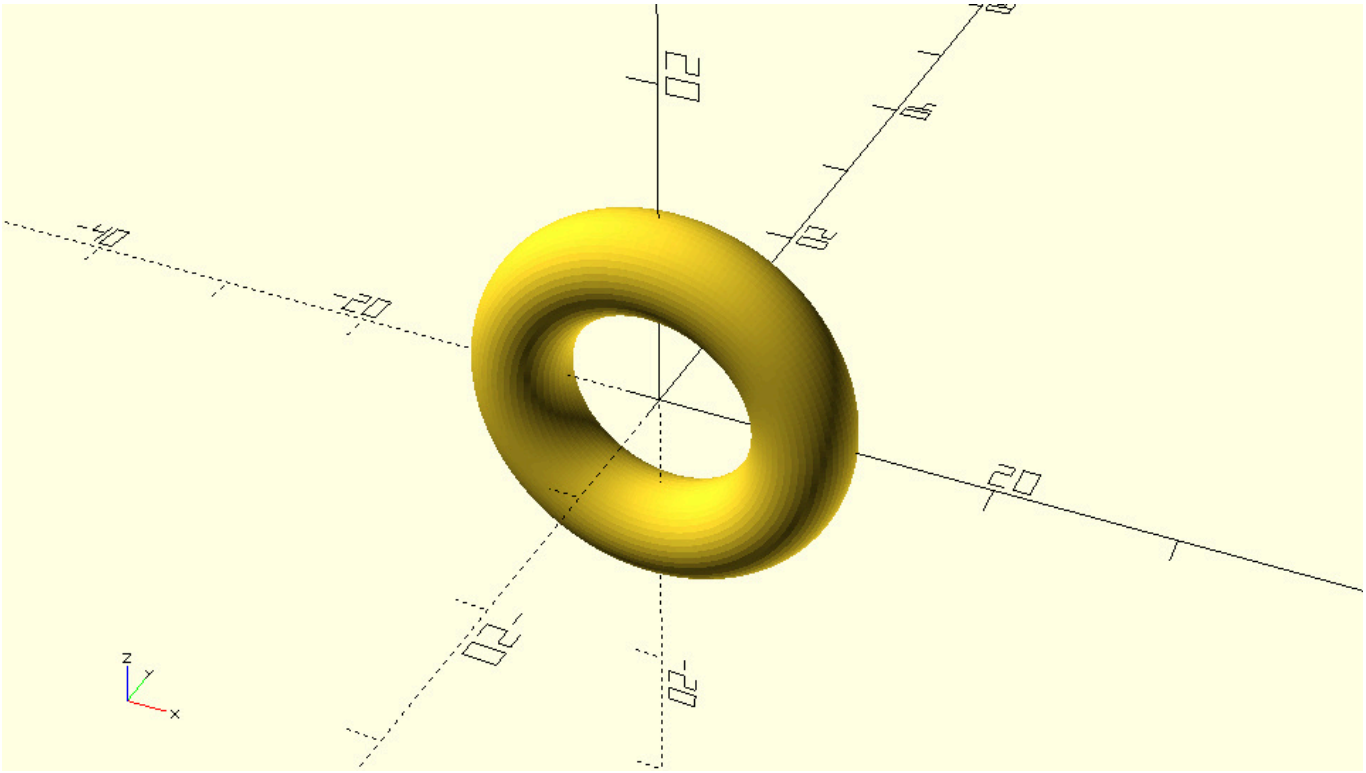
Chapter 8

Rotationally extruding 3D objects from 2D objects

So far you have been creating a lot of models and customizing your car designs while developing solid parametric modelling skills and exploring different features of OpenSCAD. It's quite impressive when you consider that every model you have created so far makes use of just three primitives: the sphere, the cube and the cylinder. By combining these primitives with the transformation commands you can create a plethora of models, but there are still models that can't be created by using these primitives alone. One such example is the following wheel design.



The above wheel design requires the creation of an object that looks like a donut.

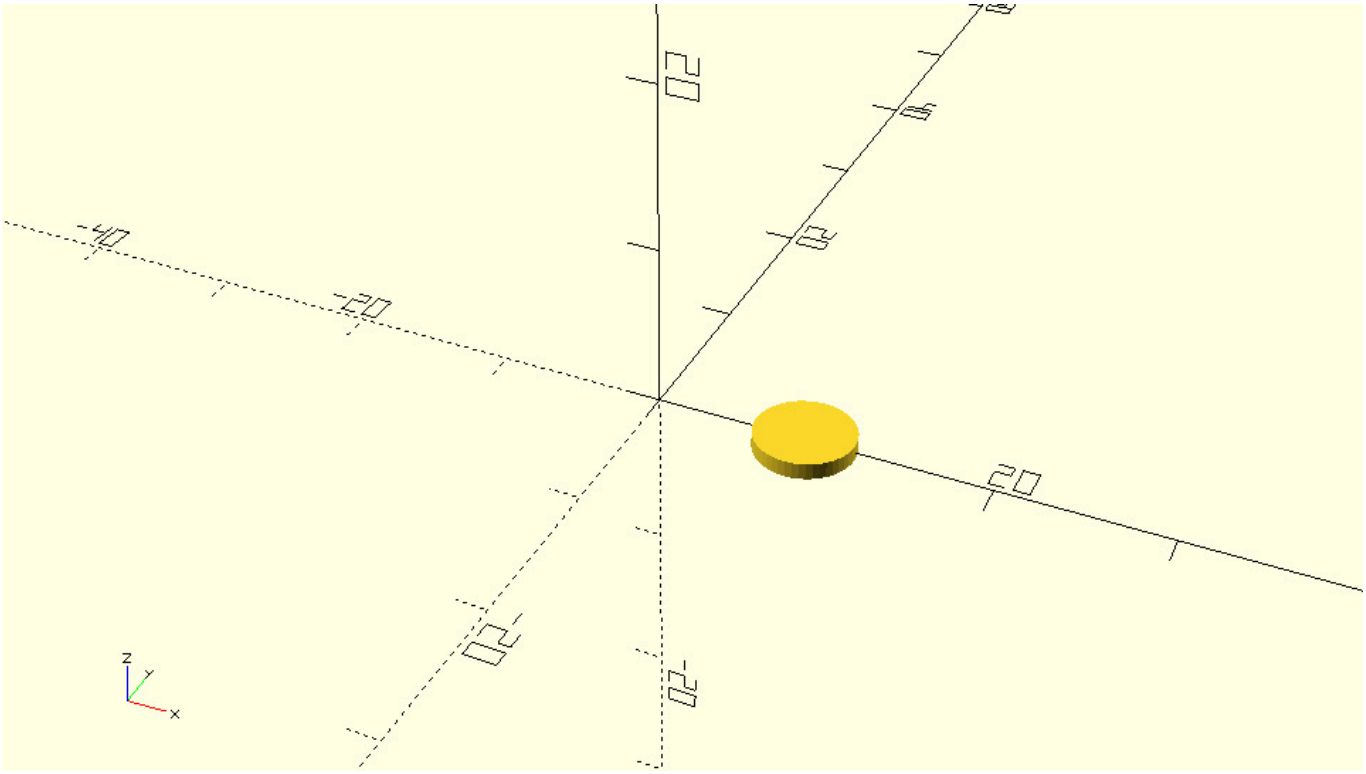


This donut shaped object can't be created with the use of the sphere, cube and cylinder primitives. Instead it requires the use of 2D primitives and a new command which can create 3D shapes from 2D profiles. Specifically, the donut can be created by first defining a circular 2D profile using the circle primitive and then rotationally extruding this profile using the rotate_extrude command.

Code

circular_profile.scad

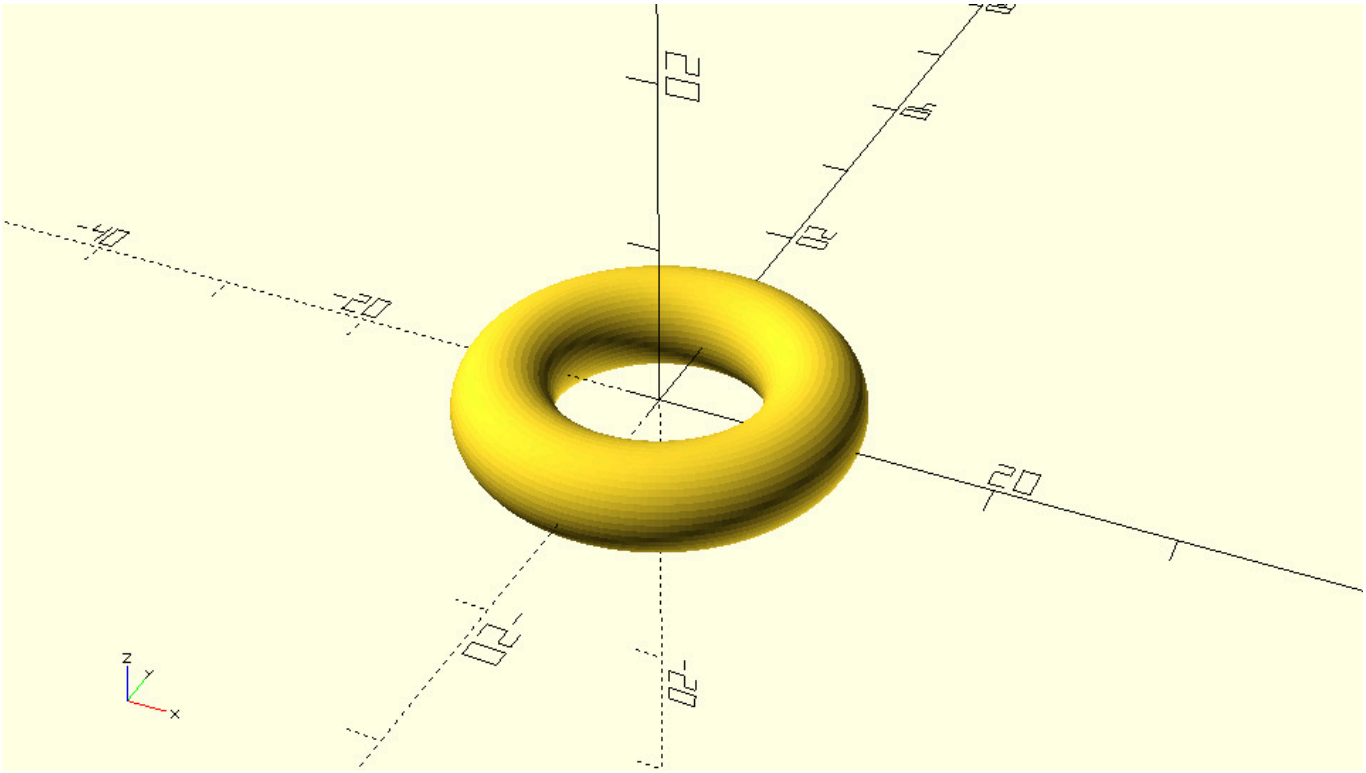
```
$fa = 1;
$fs = 0.4;
wheel_radius = 12;
tyre_diameter = 6;
translate([wheel_radius - tyre_diameter/2, 0])
    circle(d=tyre_diameter);
```



Code

extruded_donut.scad

```
$fa = 1;
$fs = 0.4;
wheel_radius = 12;
tyre_diameter = 6;
rotate_extrude(angle=360) {
    translate([wheel_radius - tyre_diameter/2, 0])
        circle(d=tyre_diameter);
}
```

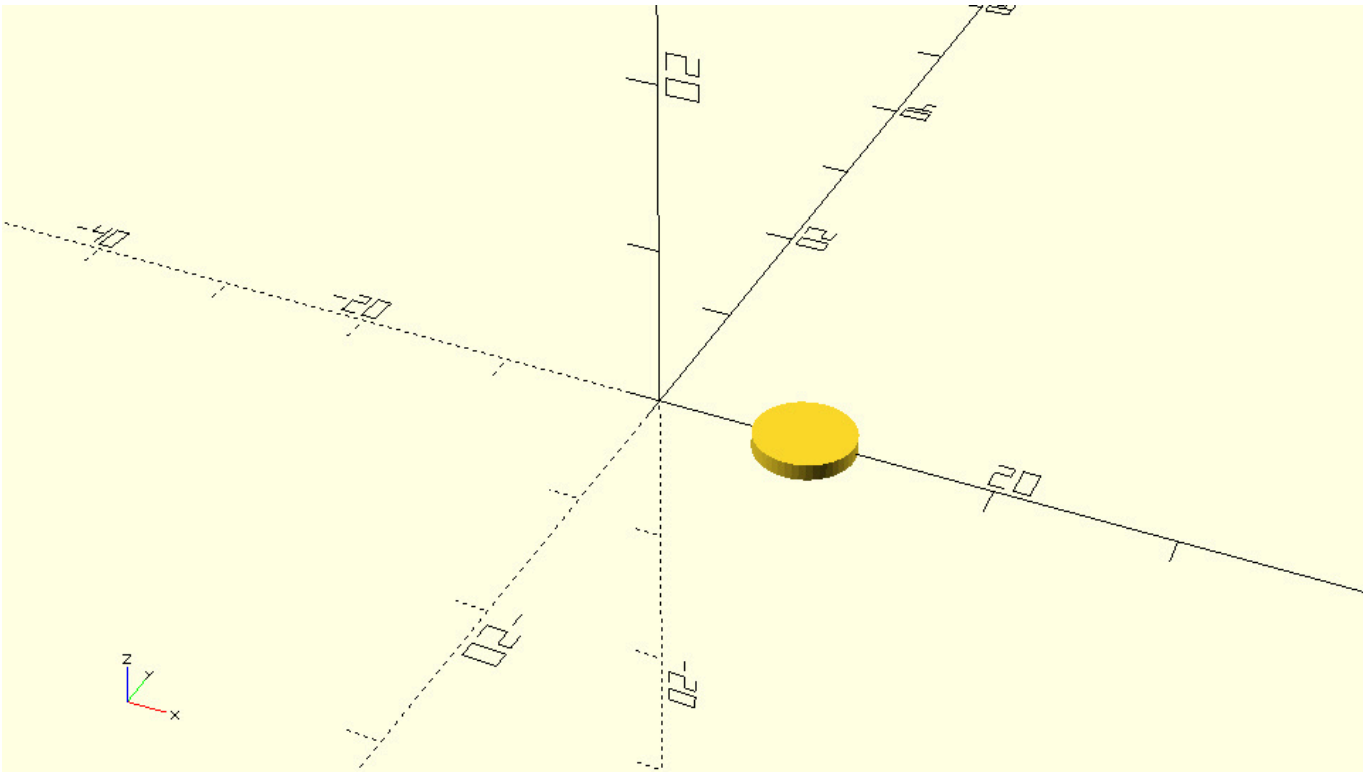


There are a few things you should notice about the 2D profile that you have created. In this case, the 2D profile is created using the circle command, and the diameter is set equal to the tyre_diameter variable. This was done because the donut-shaped object will correspond to the tyre of the wheel. Later on, you may discover other 2D primitives like the square command.

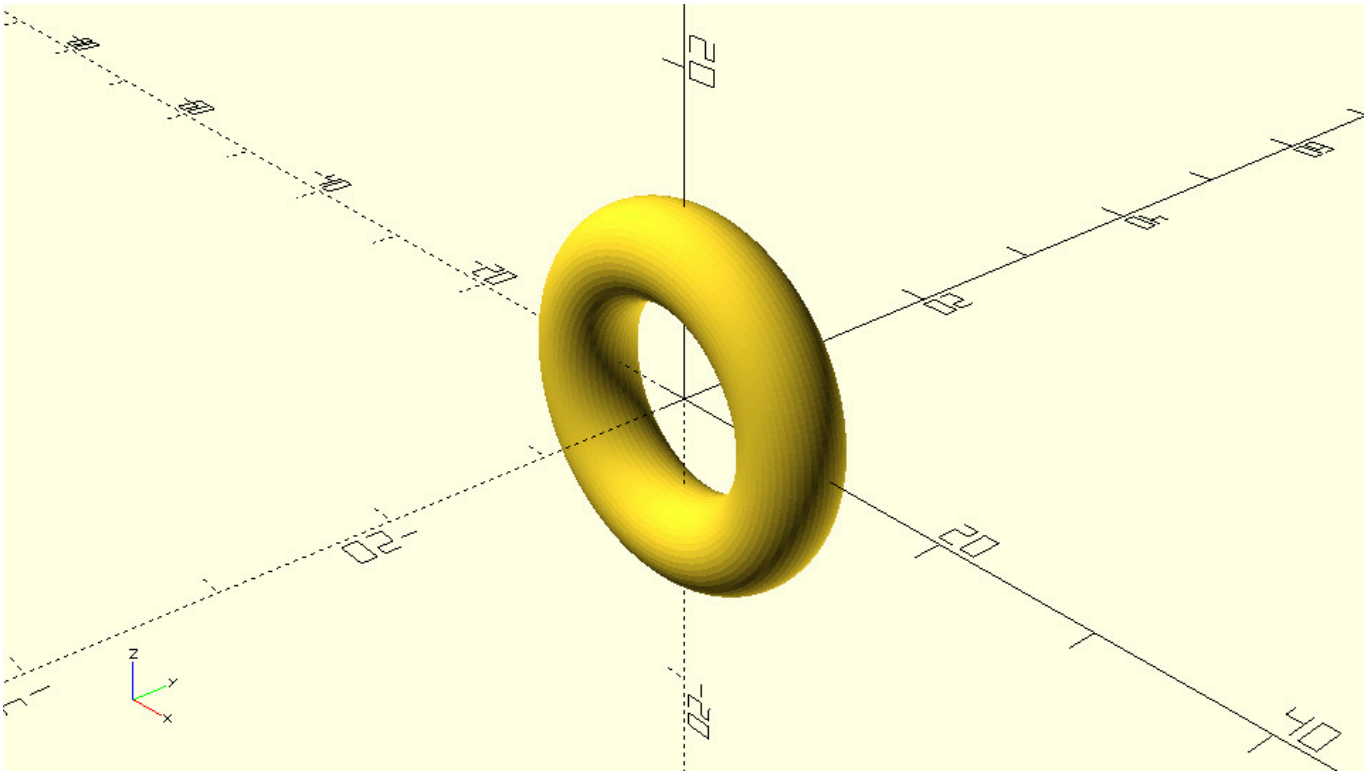
Any 2D profiles you plan to extrude should be created on the X-Y plane, and typically in the region where X is positive. Defined mathematically, 2D profiles usually reside where $X \geq 0$, and $Z = 0$. Also, 2D profiles always have zero thickness. This means that 2D profiles are never used directly as a part of the model, but are instead used in conjunction with the rotate_extrude and linear_extrude commands to define 3D objects.

You should notice a few things about the use of the rotate_extrude command as well. The rotate extrude command is used to create 3D objects, and always requires a 2D profile as an input. The commands that create the desired 2D profile need to be placed inside the pair of curly brackets that follows the rotate_extrude command. The 3D object that is created by the rotate_extrude command is the result of rotating the 2D profile around the Y axis. The resulting 3D object is then placed so that its axis of rotation lies along the Z-axis. This quirk can take some getting used to at first, so it may help to review the process one step at a time.

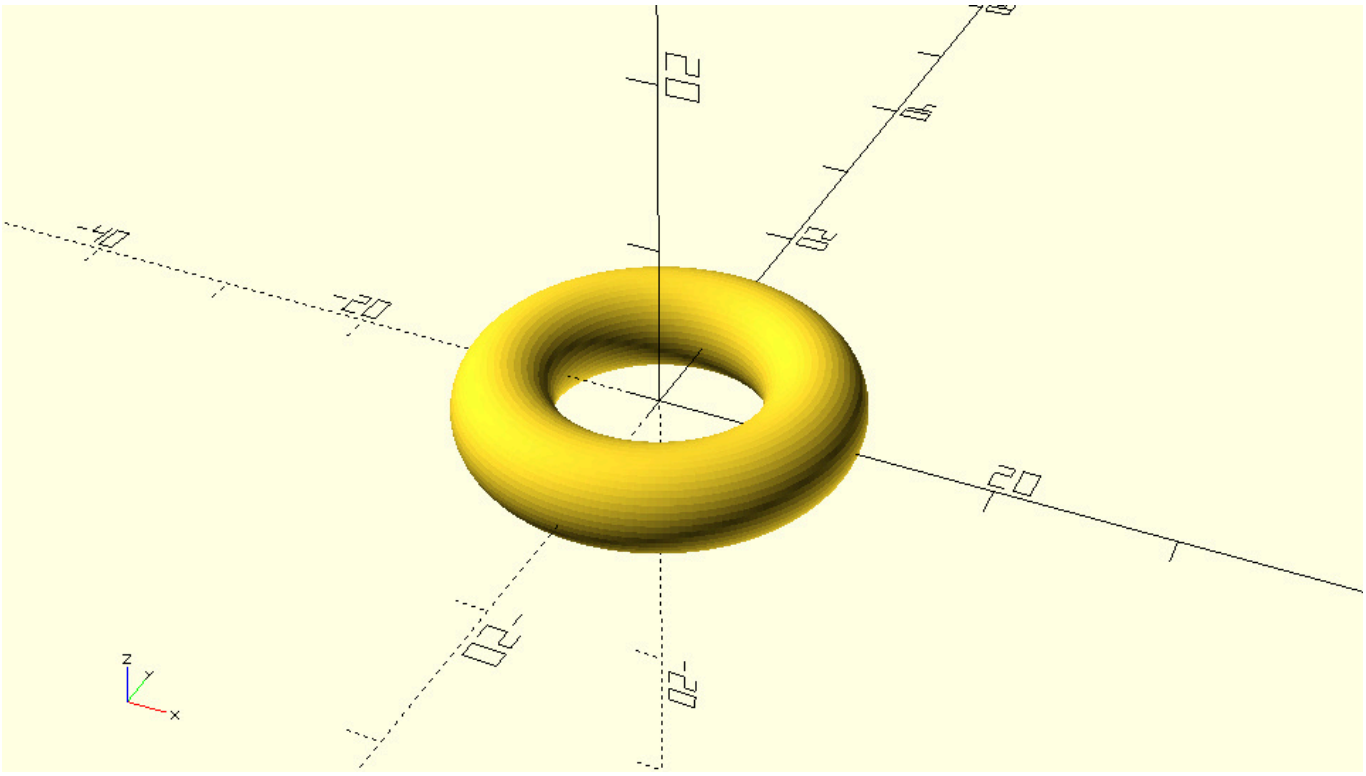
First, the rotate_extrude command takes a 2D profile as an input.



Then it creates a 3D object which is the result of rotating the supplied 2D profile around the Y axis.

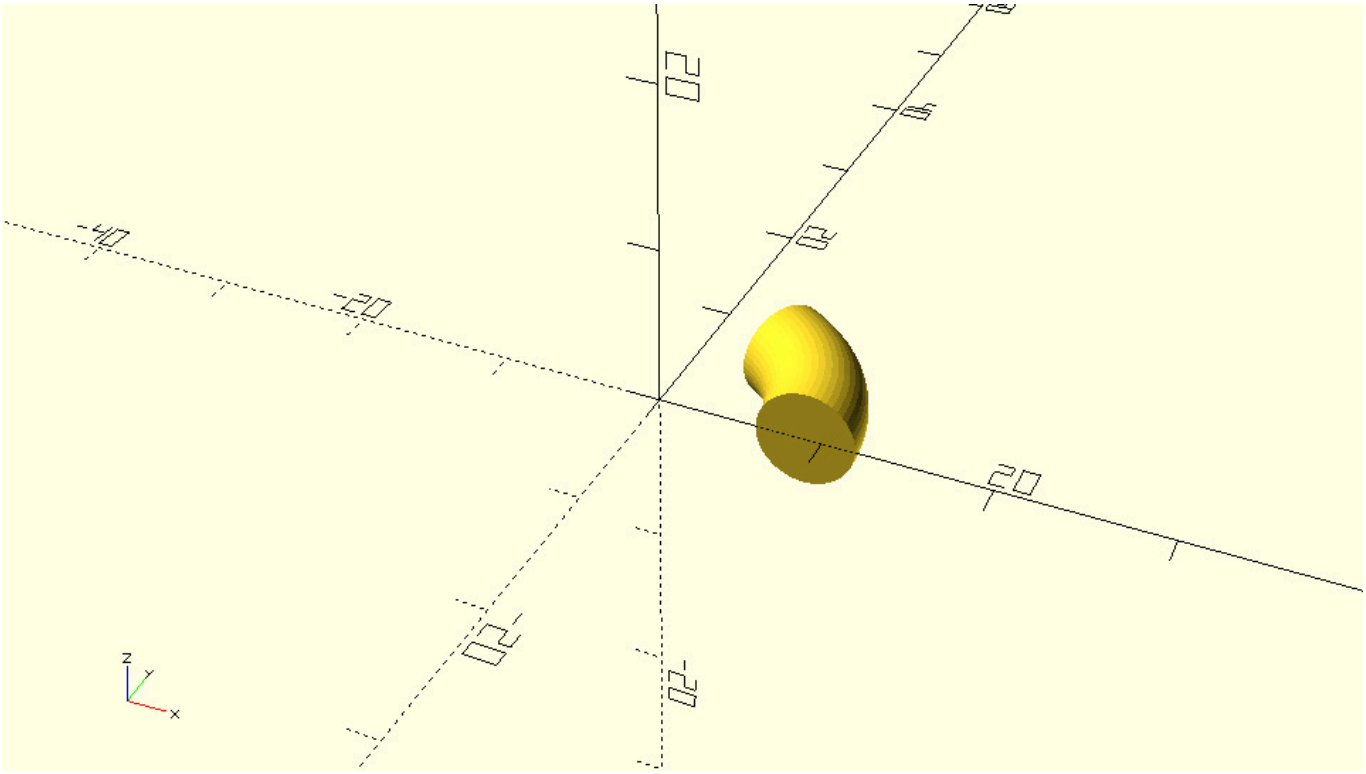


Finally, it places the 3D model as if it were rotated by 90 degrees around the X axis. The result is that the Y axis which the model was revolved around has been rotated up to align with the Z axis.

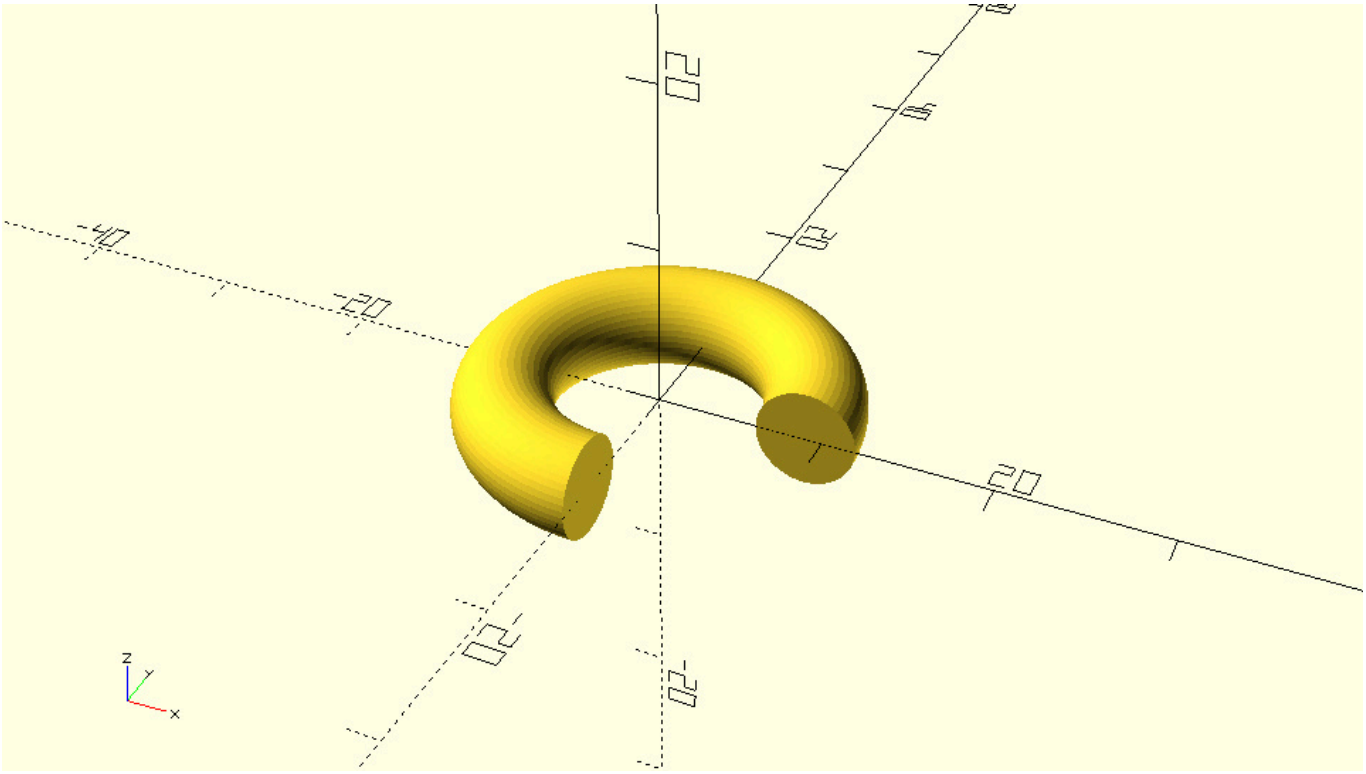


The rotate_extrude command has one input parameter named angle. The angle parameter is used to define how many degrees the 2D profile will be rotated around the Y axis. In this case the angle parameter is set equal to 360 degrees which corresponds to a full circle.

Setting the angle parameter equal to 60 degrees would create the following model.



While setting it equal to 270 degrees would create the following one. And so forth.



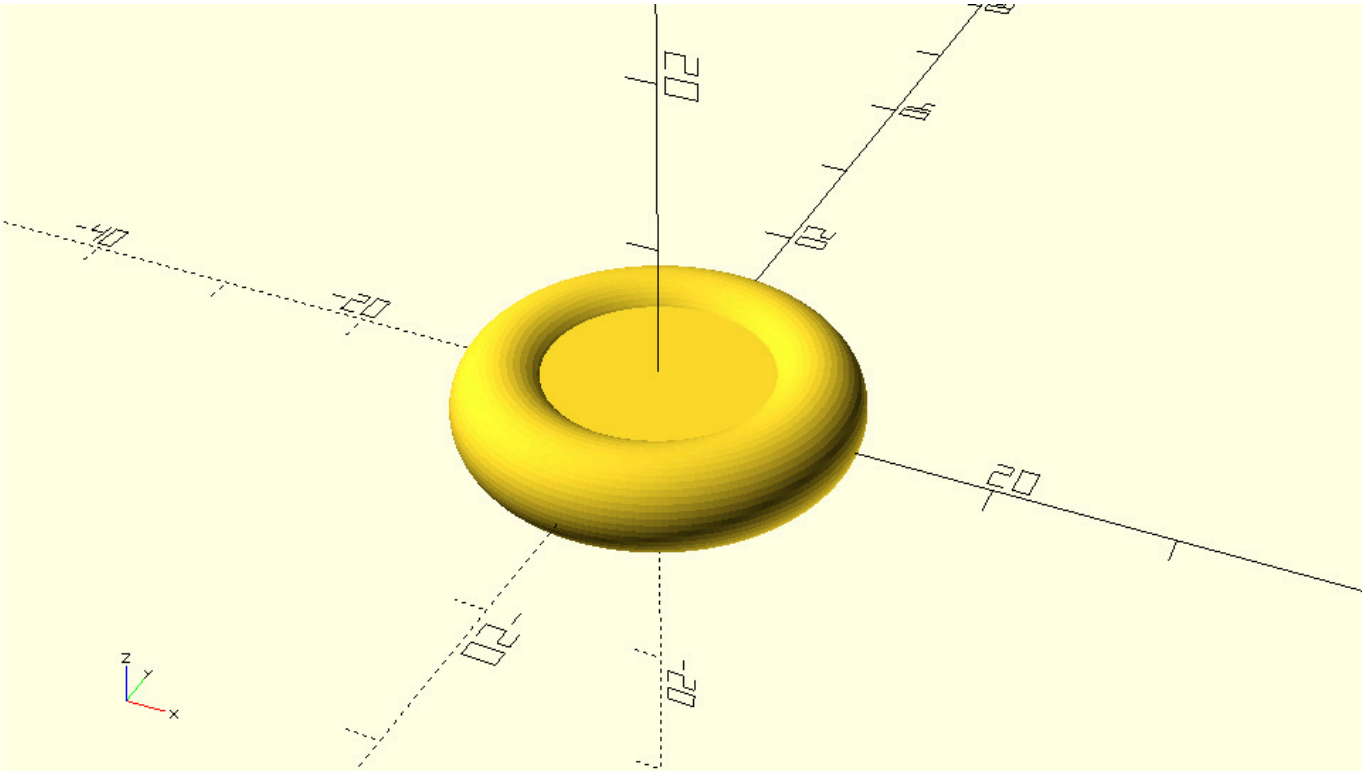
Exercise

Complete the new wheel design by defining the missing cylinder object. The height of the cylinder should be equal to the value of a wheel_width variable, while the radius of the cylinder should be equal to wheel_radius - tyre_diameter/2. The cylinder should be centred on the origin.

Code[Collapse]

rounded_wheel_horizontal.scad

```
$fa = 1;
$fs = 0.4;
wheel_radius = 12;
wheel_width = 4;
tyre_diameter = 6;
rotate_extrude(angle=360) {
    translate([wheel_radius-tyre_diameter/2,0])
        circle(d=tyre_diameter);
}
cylinder(h=wheel_width, r=wheel_radius - tyre_diameter/2, center=true);
```

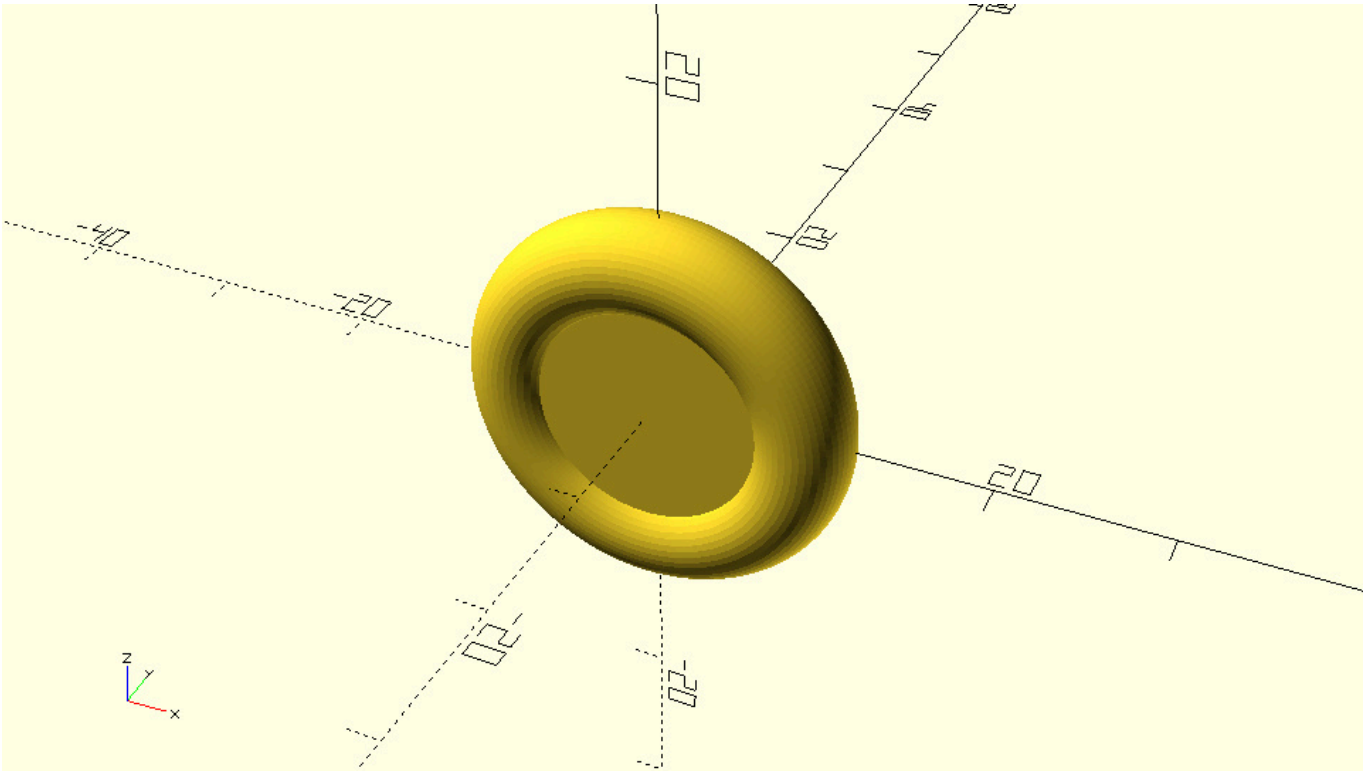


Exercise

To make this wheel compatible with the models from the previous chapters rotate it by 90 degrees around the X axis. Turn this wheel design into a module named rounded_simple_wheel and add it on your vehicle_parts.scad script for later use.

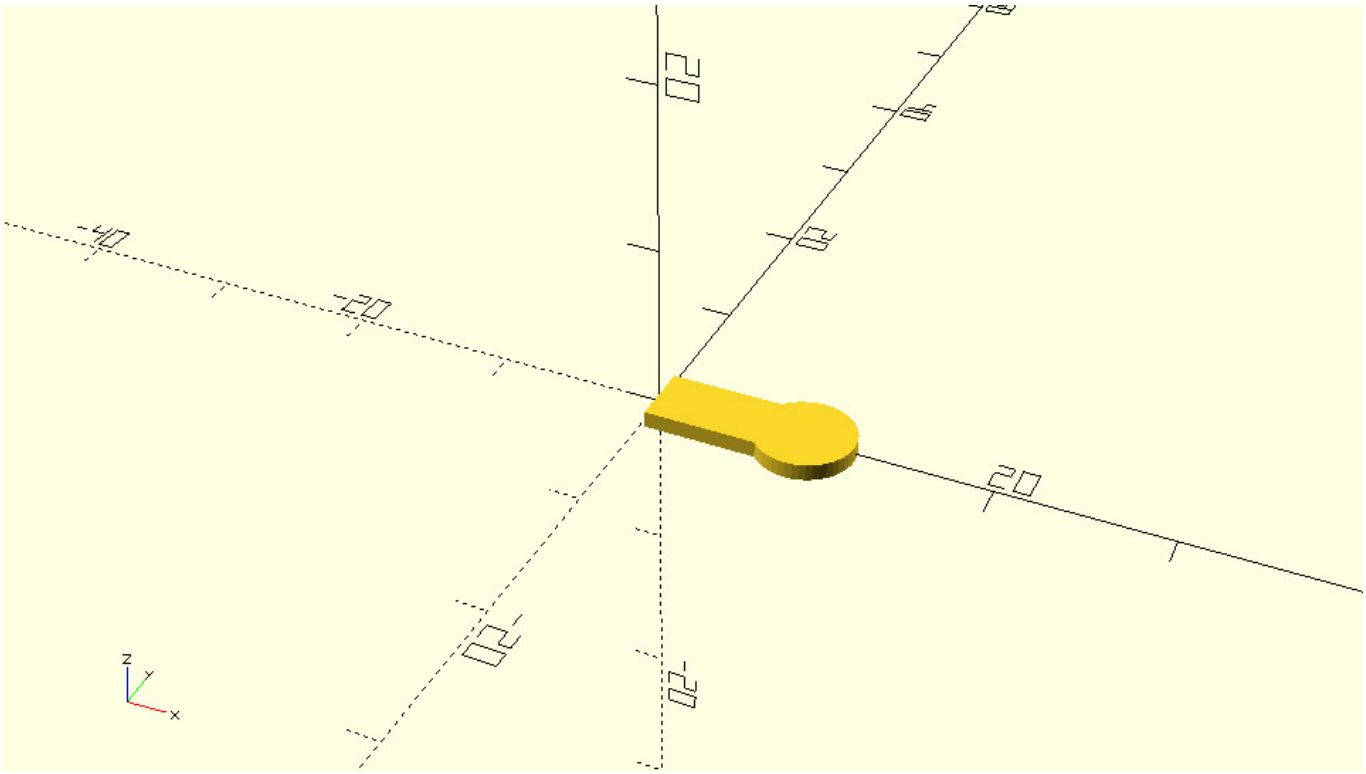
Code[Collapse]

```
…
module rounded_simple_wheel(wheel_radius=12, wheel_width=4, tyre_diameter=6) {
    rotate([90,0,0]) {
        rotate_extrude(angle=360) {
            translate([wheel_radius-tyre_diameter/2,0])
                circle(d=tyre_diameter);
        }
        cylinder(h=wheel_width, r=wheel_radius - tyre_diameter/2, center=true);
    }
}
…
```



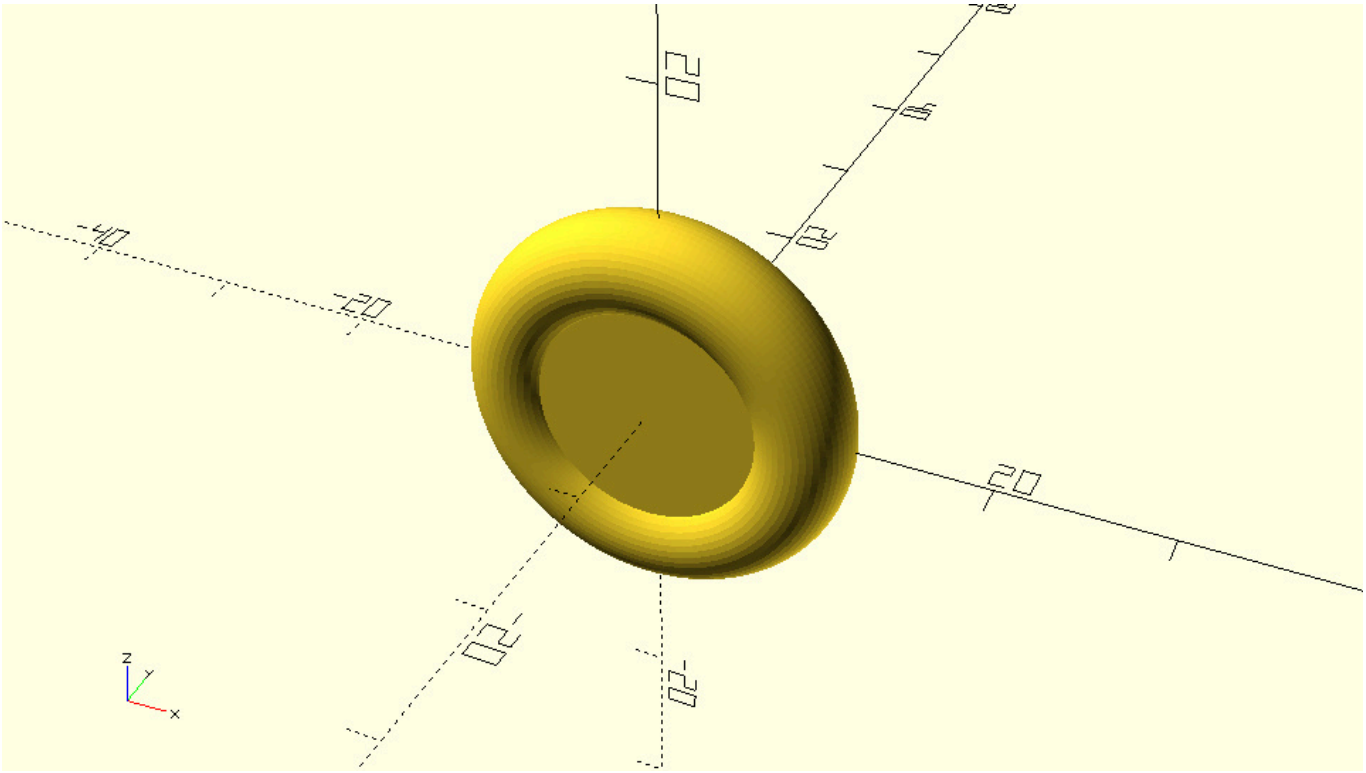
Exercise

The above wheel is an axisymmetric object, which means it exhibits symmetry around an axis. Specifically, the axis of symmetry is the axis around which the 2D profile was rotated to form the 3D object. When an object is axisymmetric it can be created with just one rotate_extrude command as long as the appropriate 2D profile is supplied. This is not the case with the above wheel design as the center part was added with a cylinder command separate from the rotational extrusion. Remove the cylinder command from the above module and make appropriate additions on the supplied 2D profile so that the whole wheel is created by the rotate_extrude command.



Code[Collapse]

```
…
translate([wheel_radius-tyre_diameter/2,0])
    circle(d=tyre_diameter);
translate([0,-wheel_width/2])
    square([wheel_radius-tyre_diameter/2,wheel_width]);
…
```



Code[Collapse]

```
…
module rounded_simple_wheel(wheel_radius=12, wheel_width=4, tyre_diameter=6) {
    rotate([90,0,0]) {
        rotate_extrude(angle=360) {
            translate([wheel_radius-tyre_diameter/2,0])
                circle(d=tyre_diameter);
            translate([0,-wheel_width/2])
                square([wheel_radius-tyre_diameter/2,wheel_width]);
        }
    }
}
…
```

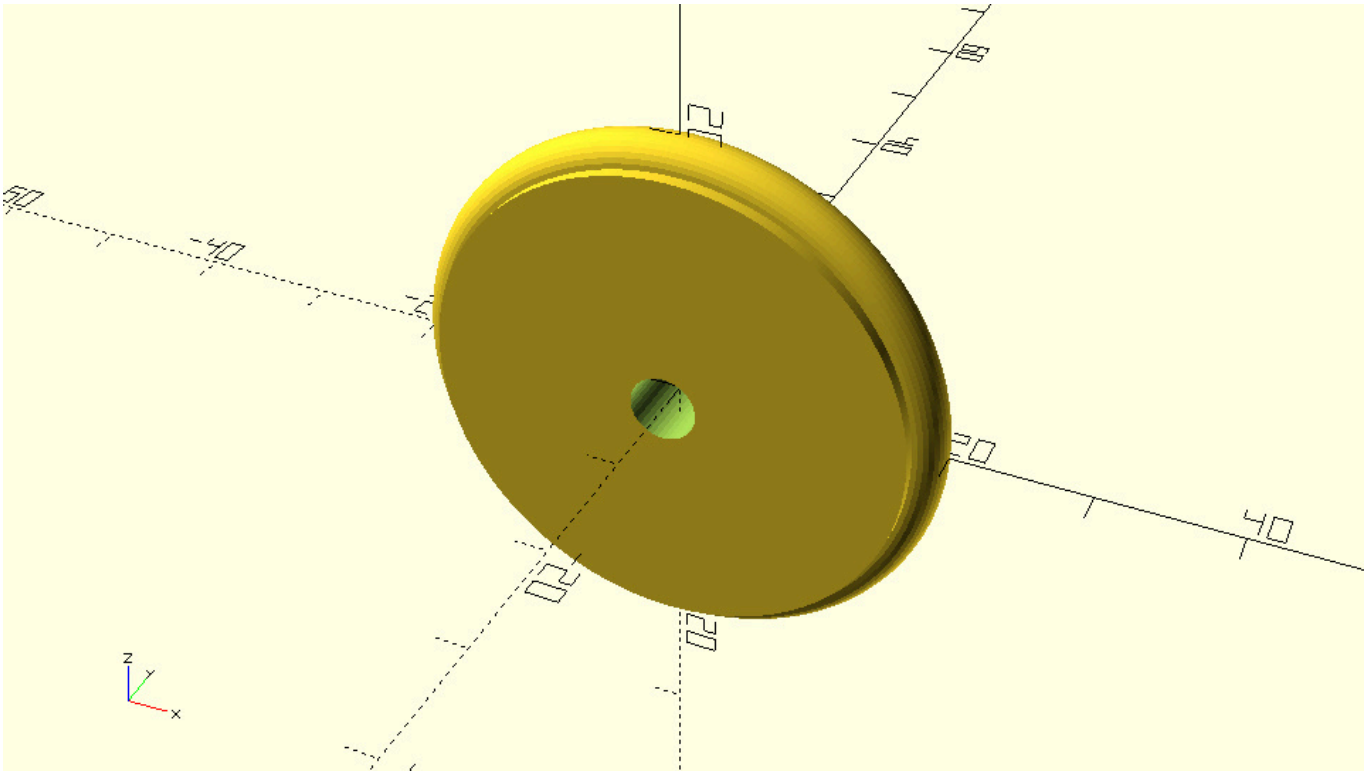
You should remember that axisymmetric objects can be completely created out of a rotate_extrude command. The previous wheel design is a concrete example of this. Whether you decide to create an axisymmetric object by supplying the 2D profile of the whole object and by using a single rotate_extrude or by using a rotate_extrude command only for its parts that can't be created in any other way, depends on each case and is up to you. If you wanted for example to further modularize your wheel designs and separate them into combinable tire and rim modules, you would inevitably need to create the donut-shaped tire using a rotate_extrude command. Since in this case the rim of the wheel would be a separate module without a rotate_extrude command already present in it, the simplest and most straight forward way to create it would be by using a cylinder command.

Challenge

It's time to put your new knowledge into practice to create a rim for a mini robot car project.

Exercise

Extend the rounded_simple_wheel module, so that the wheel design has a hole on its hub that can be used for mounting it on an axle. To do so you would need to subtract a cylinder from the existing model using a difference command. The diameter of the hole should be equal to a new module input parameter named axle_diameter. The default value of this parameter should be 3 units. By the way in which you define the height of the cylinder you should guarantee that the cylinder is always a bit longer than the width of the wheel to avoid any errors when using the difference command. After saving the modified module you should use it to create a version a wheel with wheel_radius, wheel_width, tire_diameter and axle_diameter of 20, 6, 4 and 5 units respectively.



Code[Collapse]

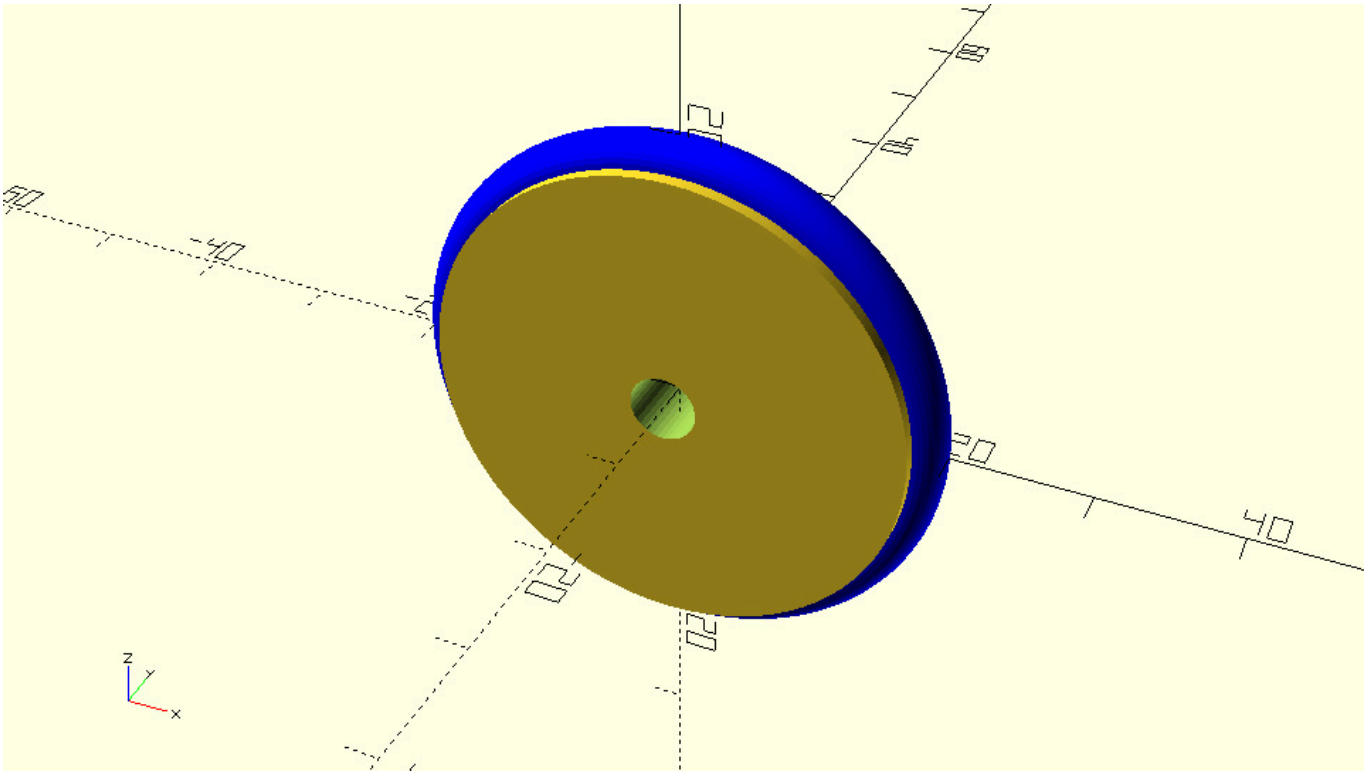
robot_wheel.scad

```
…
module rounded_simple_wheel(wheel_radius=12, wheel_width=4, tyre_diameter=6, axle_diameter=3) {
    difference() {

        // wheel
        rotate([90,0,0]) {
            rotate_extrude(angle=360) {
                translate([wheel_radius-tyre_diameter/2,0])
                    circle(d=tyre_diameter);
                translate([0,-wheel_width/2])
                    square([wheel_radius-tyre_diameter/2,wheel_width]);
            }
        }

        // axle hole
        rotate([90,0,0])
            cylinder(h=wheel_width+1,r=axle_diameter/2,center=true);
    }
}
…
```

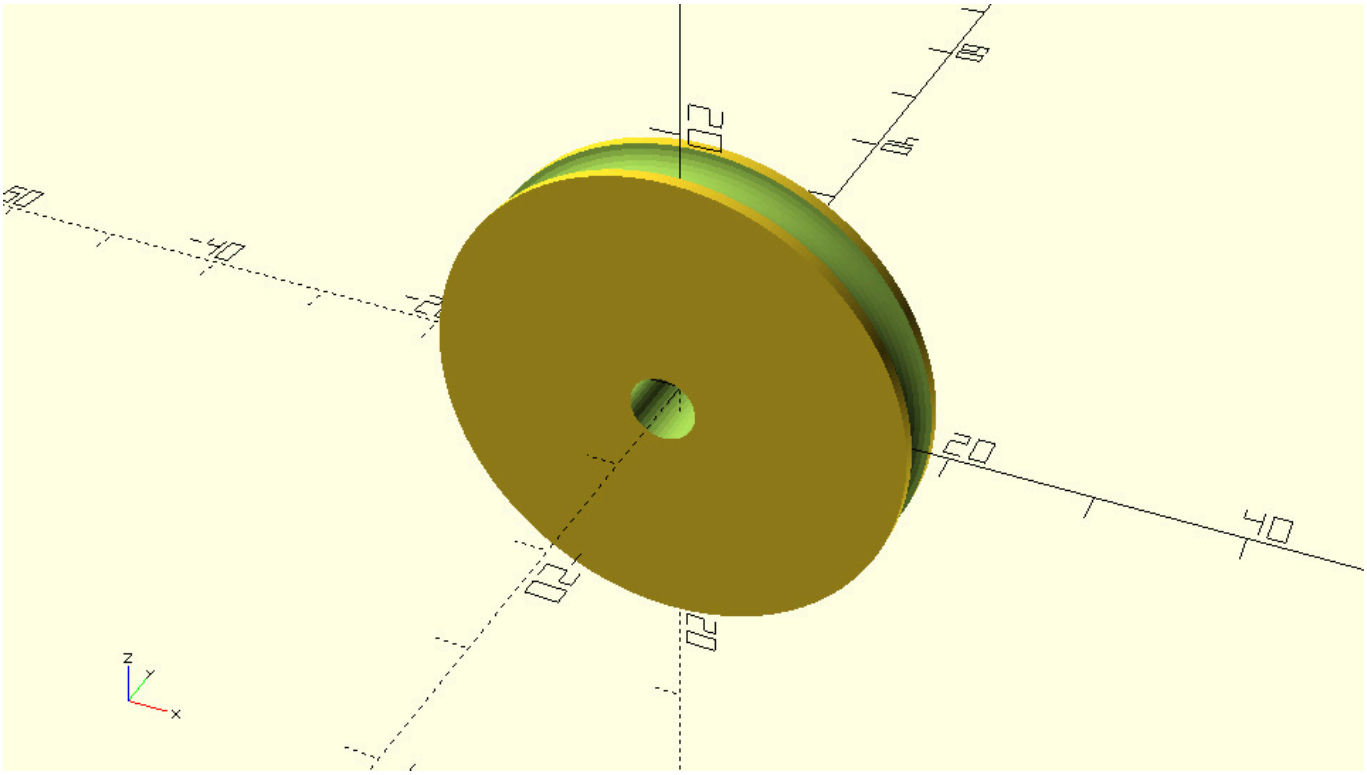
Code[Collapse]

```
…
rounded_simple_wheel(wheel_radius=20, wheel_width=6, tyre_diameter=4, axle_diameter=5);
…
```

This wheel design looks about right for a mini robot car application, but you could do something to give more traction to your robot. Instead of 3D printing the whole wheel, you could just 3D print the rim and then add an O-ring or rubber band as the tire for more traction. The resulting wheel would look like the following image, where the O-ring or rubber band is represented by the blue color.



The corresponding rim that you would have to 3D print in this case is the following.



Exercise

Using the rounded_simple_wheel module as a guide, create a new module named robot_rim. The robot_rim module should have the same input parameters as the rounded_simple_wheel module. Add all necessary commands to the robot_rim module so that it creates the above rim design. There are two ways in which you can do this.

- The first way is to subtract the circle that corresponds to the tire from the square that corresponds to the rim in the definition of the design's 2D profile inside the rotate_extrude command and then to additionally subtract the axle's cylinder from the resulting 3D object to get the final rim design.
- The second way is to subtract a donut-shaped object that corresponds to the tire and the cylinder that corresponds to the axle hole from the larger cylinder that corresponds to the rim.

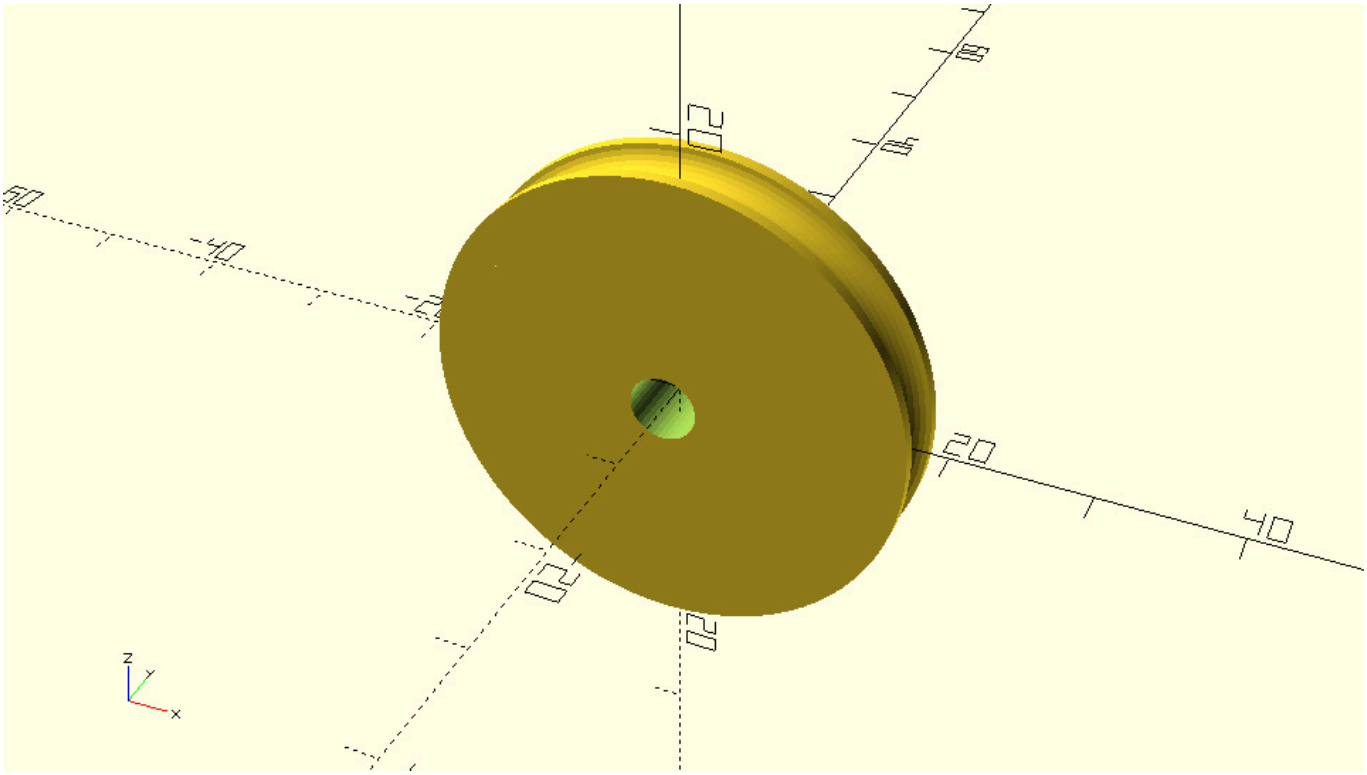
Remember, while there is some conventional wisdom surrounding good design practices, there is no objectively right or wrong choice. In practice you would most likely go with the option that first came to mind or that made the most sense to you. For the sake of this exercise, consider trying it both ways to see which approach you like the most.

- First approach

Code[Collapse]

robot_rim_from_profile_difference.scad

```
…
module rounded_simple_wheel(wheel_radius=12, wheel_width=4, tyre_diameter=6, axle_diameter=3) {
    rotate([90,0,0])difference() {

        // resulting rim
        rotate_extrude(angle=360) {
            difference() {

                // cylindrical rim profile
                translate([0,-wheel_width/2])
                    square([wheel_radius-tyre_diameter/2,wheel_width]);

                // tire profile
                translate([wheel_radius-tyre_diameter/2,0])
                    circle(d=tyre_diameter);
            }
        }

        // axle hole
        cylinder(h=wheel_width+1,r=axle_diameter/2,center=true);
    }
}
…
```

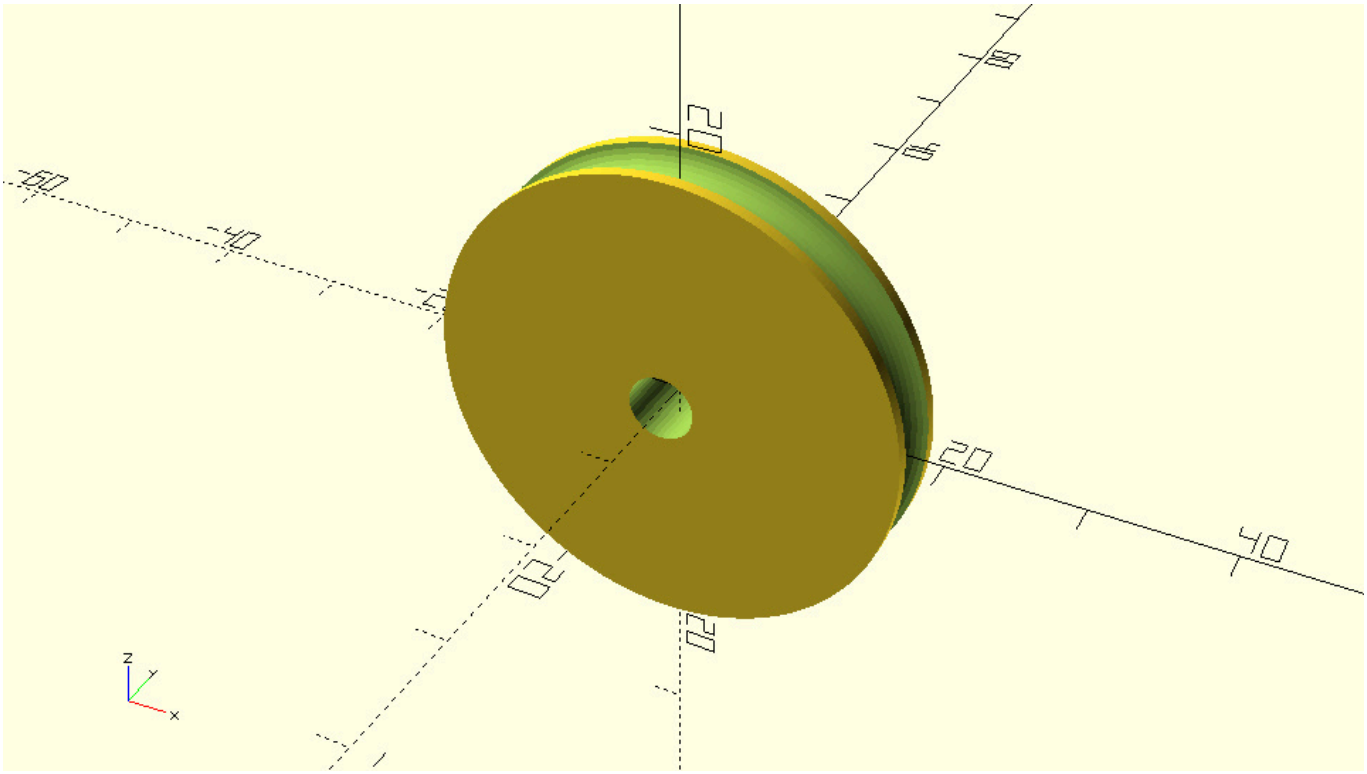


- Second approach

Code[Collapse]

robot_rim_from_3d_object_difference.scad

```
…
module rounded_simple_wheel(wheel_radius=12, wheel_width=4, tyre_diameter=6, axle_diameter=3) {
    rotate([90,0,0])
        difference() {

            // cylindrical rim
            cylinder(h=wheel_width,r=wheel_radius-tyre_diameter/2,center=true);

            // tire
            rotate_extrude(angle=360) {
                translate([wheel_radius-tyre_diameter/2,0])
                    circle(d=tyre_diameter);
            }

            // axle hole
            cylinder(h=wheel_width+1,r=axle_diameter/2,center=true);
        }
}
…
```

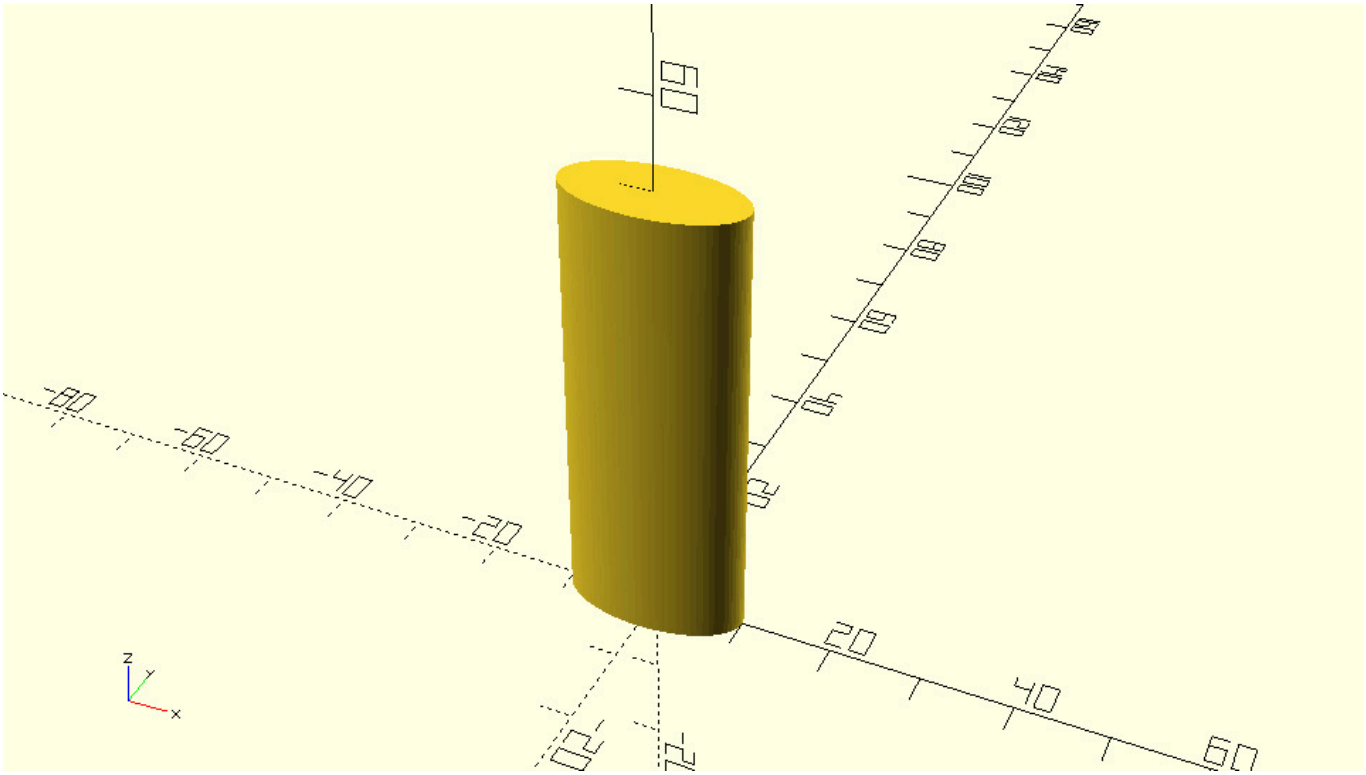


Often, it's helpful to consider the manufacturing process that will be used in an object's creation when designing a new part. Usually, this consideration promotes designs that accommodate the manufacturing method at hand, but it can guide your modeling process as well.

For example, consider a scenario where instead of using additive manufacturing like 3D printing to manufacture this robot wheel, you used subtractive manufacturing like lathe or a mill. In this case you might opt to use the second approach, since it would more closely replicate the manufacturing process at hand, and could give you a better estimate of how many steps the final manufacturing process might take.

Linearly extruding 3D objects from 2D objects

As briefly mentioned, there is another OpenSCAD command that can be used to create 3D objects from supplied 2D profiles. This is the linear_extrude command. In contrast to the rotate_extrude command, linear_extrude creates a 3D object by extending along the Z axis a 2D profile that lies on the XY plane. Similar to the rotate_extrude command, linear_extrude can be used when the 3D object you want to create can't be directly created by combining available 3D primitives. One such example is the following.

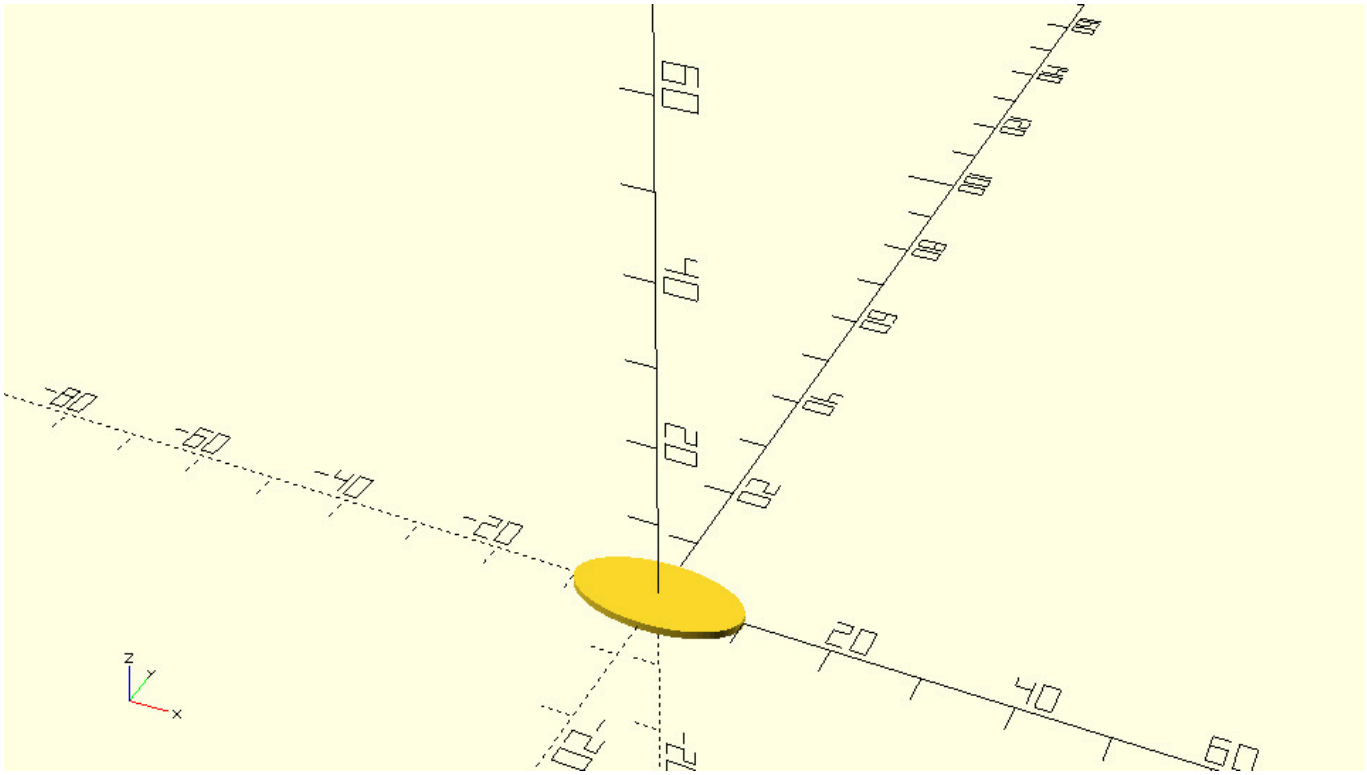


Code

extruded_ellipse.scad

```
$fa = 1;
$fs = 0.4;
linear_extrude(height=50)
    scale([2,1,1])
    circle(d=10);
```

The above object is a tube that has the following profile.



Code

ellipse_profile.scad

```
$fa = 1;
$fs = 0.4;
scale([2,1,1])
    circle(d=10);
```

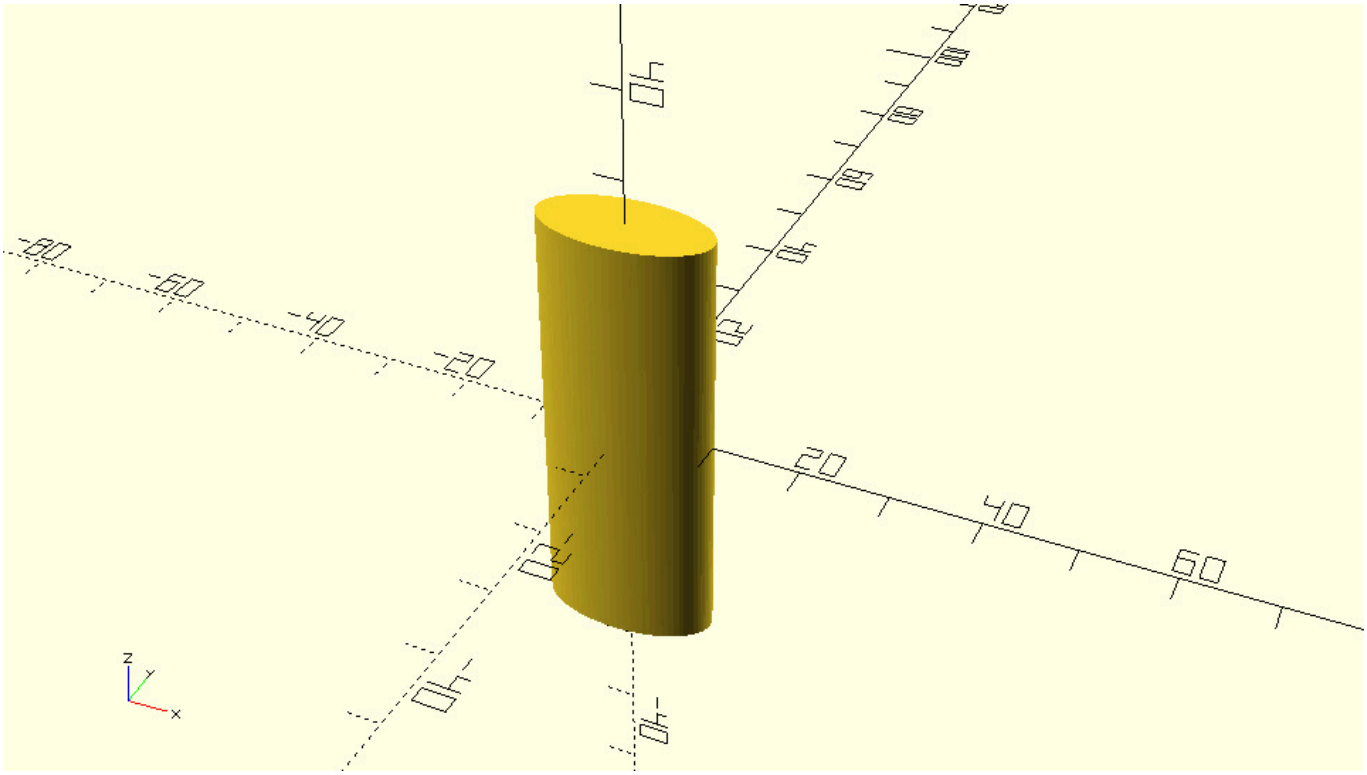
There are a few points you should notice regarding the use of linear_extrude. The syntax of linear_extrude is similar to the syntax of the rotate_extrude command. The commands that create the 2D profile that will be extruded along the Z axis need to be placed inside a pair of curly brackets that follows the linear_extrude command. The parameter height is used to define how many units along the Z axis the 2D profile is going to be extruded. By default, the 2D profile is extruded along the positive direction of the Z axis by an amount of units equal to the value assigned to the height parameter.

By passing an additional parameter named center and setting it equal to true, the 2D profile is extruded along both directions of the Z axis. The total length of the resulting object will still be equal to the height parameter.

Code

centered_extrusion.scad

```
…
linear_extrude(height=50,center=true)
    scale([2,1,1])
    circle(d=10);
…
```

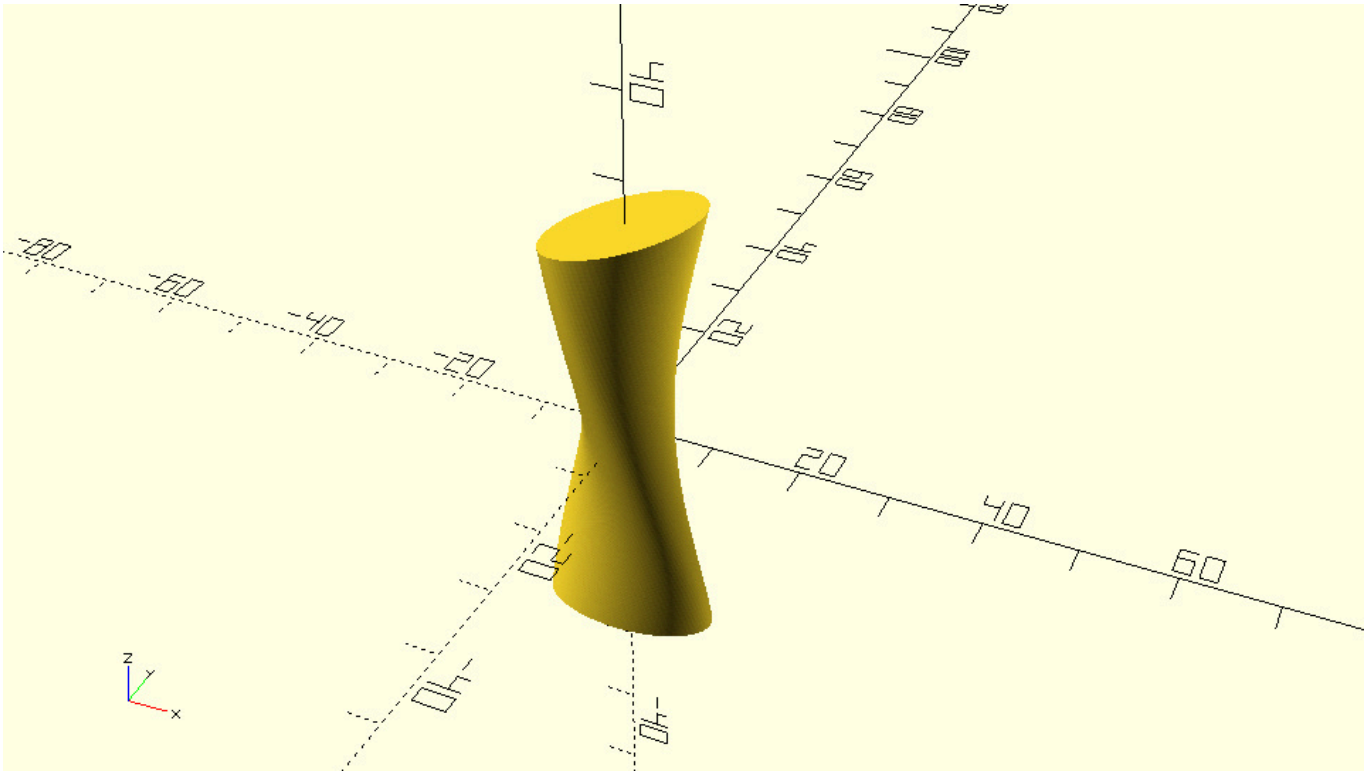


An additional parameter named twist can also be used to twist the resulting 3D object around the Z axis by the specified angle.

Code

extrusion_with_twist.scad

```
…
linear_extrude(height=50,center=true,twist=120)
    scale([2,1,1])
    circle(d=10);
…
```

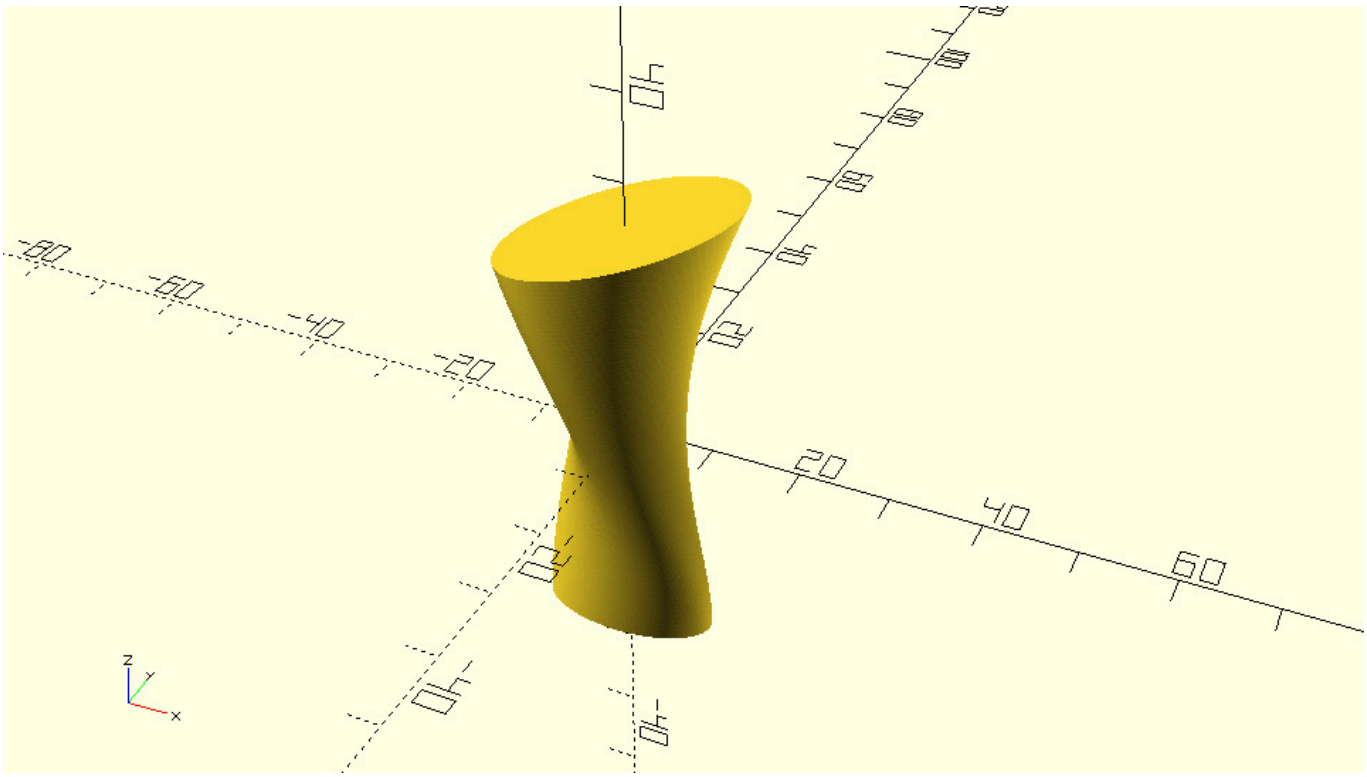


Finally, another parameter named scale can be used to scale one end of the resulting 3D by the specified scaling factor.

Code

extrusion_with_twist_and_scale.scad

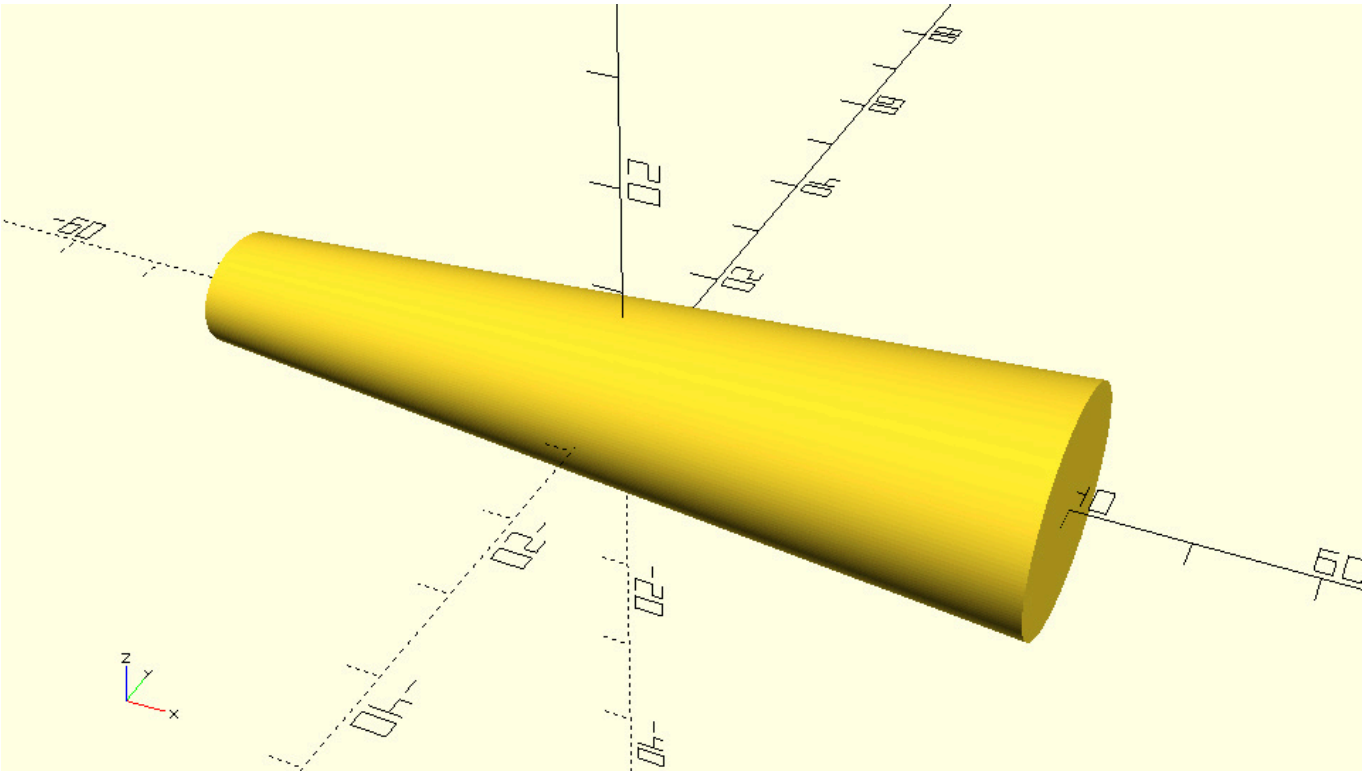
```
…
linear_extrude(height=50,center=true,twist=120,scale=1.5)
    scale([2,1,1])
    circle(d=10);
…
```



It should be pretty clear by now how the rotate_extrude and linear_extrude commands give you the ability to create objects that wouldn't be possible by directly combining the available 3D primitives. You can use these commands to create more abstract and artistic designs but let's see how you could use the linear_extrude command to create a new car body.

Exercise

Use the linear_extrude command similar to the above examples in order to create the following car body. You should create a new module named extruded_car_body. The module should have a length, rear_height, rear_width and scaling_factor input parameter. The default values of the parameters should be 80, 20, 25 and 0.5 units respectively. The length and scaling factor parameters of the module will be used in the call to linear_extrude command to set the values of its height and scale parameters. The supplied 2D profile should be a circle that has been resized according to the rear_height and rear_width parameters.



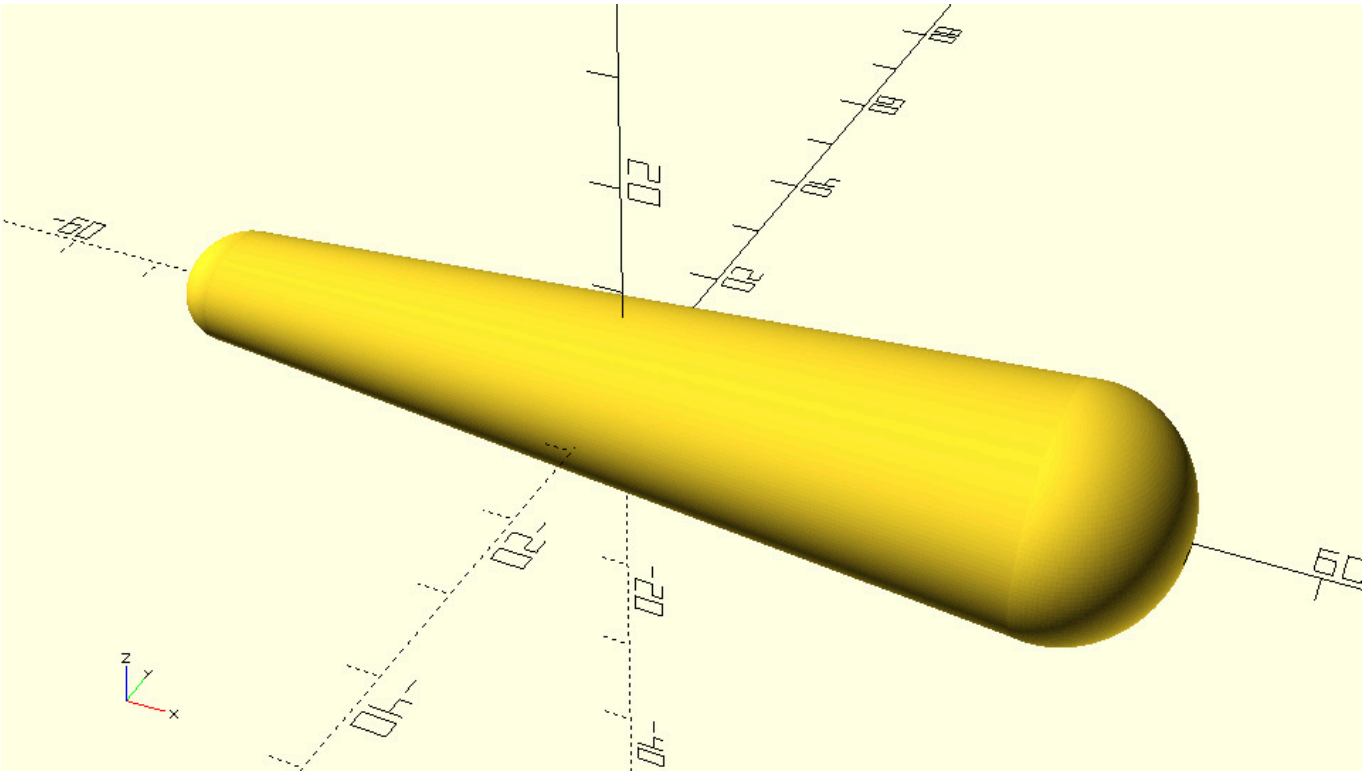
Code[Collapse]

extruded_car_body.scad

```
module rounded_car_body(length=80, rear_height=20, rear_width=25, scaling_factor=0.5) {
rotate([0,-90,0])
    linear_extrude(height=length,center=true,scale=scaling_factor)
    resize([rear_height,rear_width])
    circle(d=rear_height);
}
```

Exercise

Extend the previous module by adding a boolean input parameter named rounded. The default value of the parameter should be false. If the rounded parameter is set to true, then two additional objects should be created at the front and rear of the body in order to make in rounded as in the following image. These two objects are spheres that have been resized and scaled. Try figuring out an appropriate way to resize and scale the sphere to achieve a result similar to the image below.



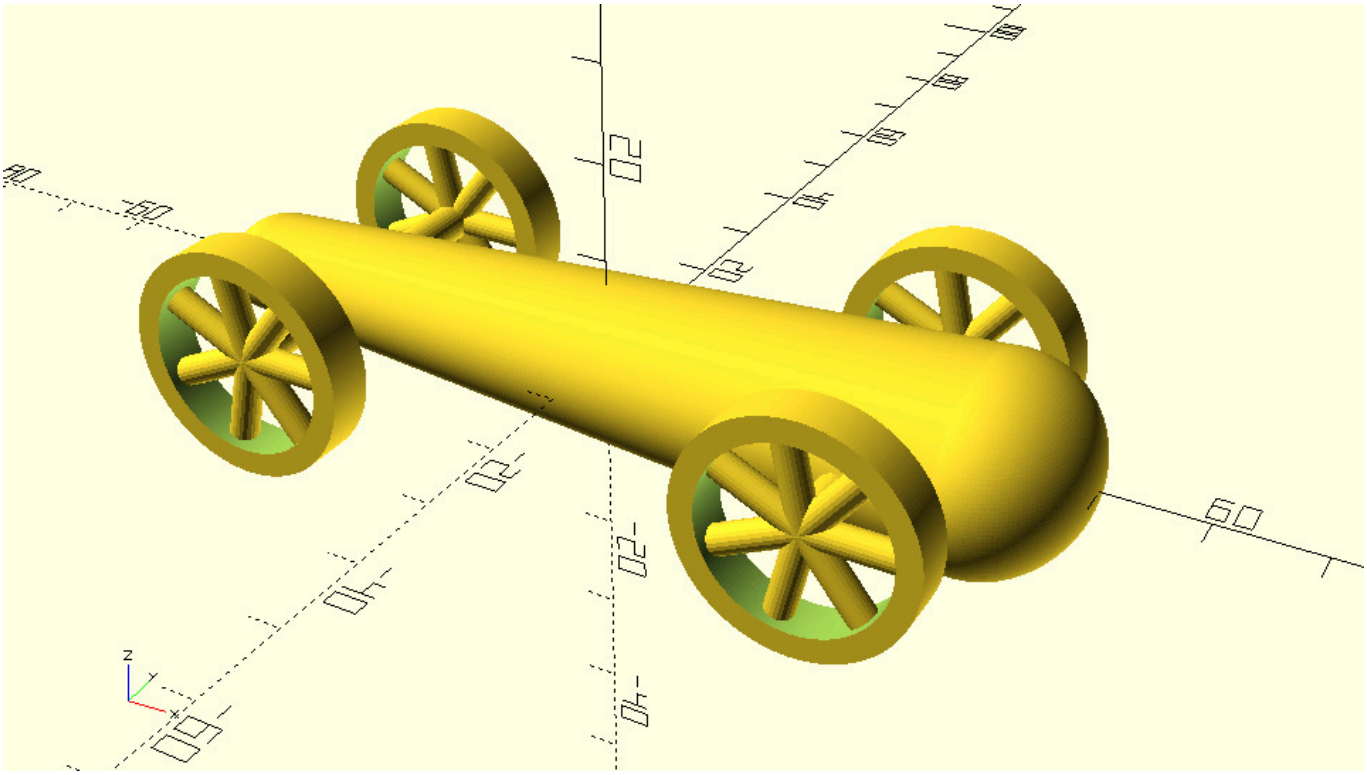
Code[Collapse]

rounded_extruded_car_body.scad

```
…
module rounded_car_body(length=80, rear_height=20, rear_width=25, scaling_factor=0.5, rounded=false)
{

    // center part
    rotate([0,-90,0])
        linear_extrude(height=length,center=true,scale=scaling_factor)
        resize([rear_height,rear_width])
        circle(d=rear_height);

    if (rounded) {
        // rear part
        translate([length/2,0,0])
            resize([rear_height,rear_width,rear_height])
            sphere(d=rear_height);

        // front part
        translate([-length/2,0,0])
            scale(scaling_factor)
            resize([rear_height,rear_width,rear_height])
            sphere(d=rear_height);
    }
}
…
```

Exercise

Use the new rounded body in any car design that you like.



As mentioned, the rotate_extrude and linear_extrude commands can also be used to create more abstract objects. When the supplied 2D profile is created using the available circle and square 2D primitives and when the twist and scale parameters of the linear_extrude command are not utilized, then the resulting 3D object could also be directly created using the available 3D primitives. What really makes the use of these commands much more powerful is the ability to create any 2D profile that is not a combination of circles and squares but rather an arbitrary shape. This ability is available through the use of the polygon 2D primitive which you are going to learn about in the next chapter.

Chapter 9

Doing math calculations in OpenSCAD

So far you have learned that OpenSCAD variables can hold only one value throughout the execution of a script, the last value that has been assigned to them. You have also learned that a common use of OpenSCAD variables is to provide parameterization of your models. In this case every parameterized model would have a few independent variables, whose values you can change to tune that model. These variables are usually directly assigned a value as in the following examples.

Code

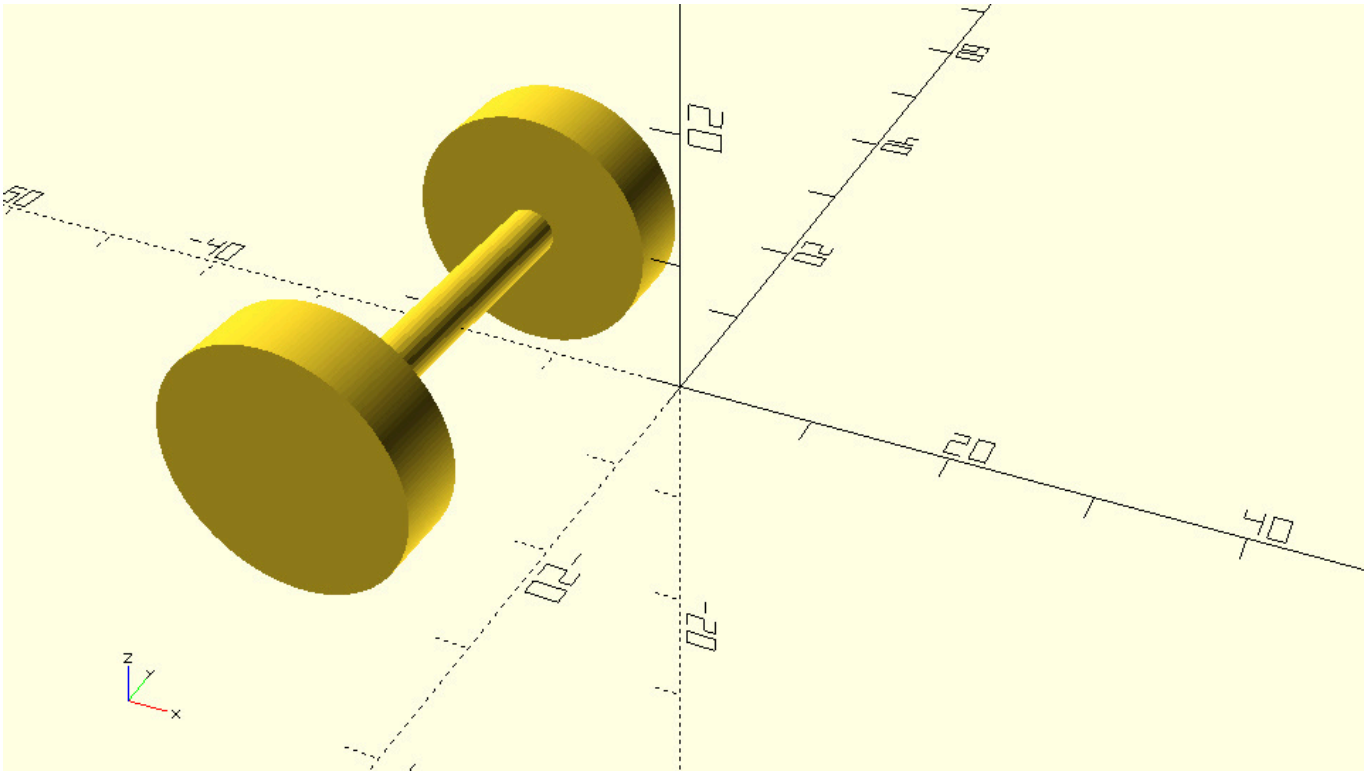
```
…
wheel_diameter = 12;
…
body_length = 70;
…
wheelbase = 40;
…
// etc.
…
```

Another thing that you have seen a few times already, but which has not been mentioned explicitly is the ability to perform mathematical operations using variables and hard-coded values in your script. One example of this is in implementing the car's wheelbase. Recall that the car's axles and wheels were translated along the X axis and away from the origin by half the value of the wheelbase. Since in this case the wheelbase is a variable that has already been defined in your script, you can calculate the amount of units by dividing the wheelbase variable by two. A similar thing was done with the track variable to place the left and right wheels of the car. Recall that the left and right wheels were translated along the Y axis and away from the origin by half the value of the track.

Code

axle_with_wheelset.scad

```
use <vehicle_parts.scad>
$fa = 1;
$fs = 0.4;

wheelbase = 40;
track = 35;

translate([-wheelbase/2, track/2])
    simple_wheel();
translate([-wheelbase/2, -track/2])
    simple_wheel();
translate([-wheelbase/2, 0, 0])
    axle(track=track);
```

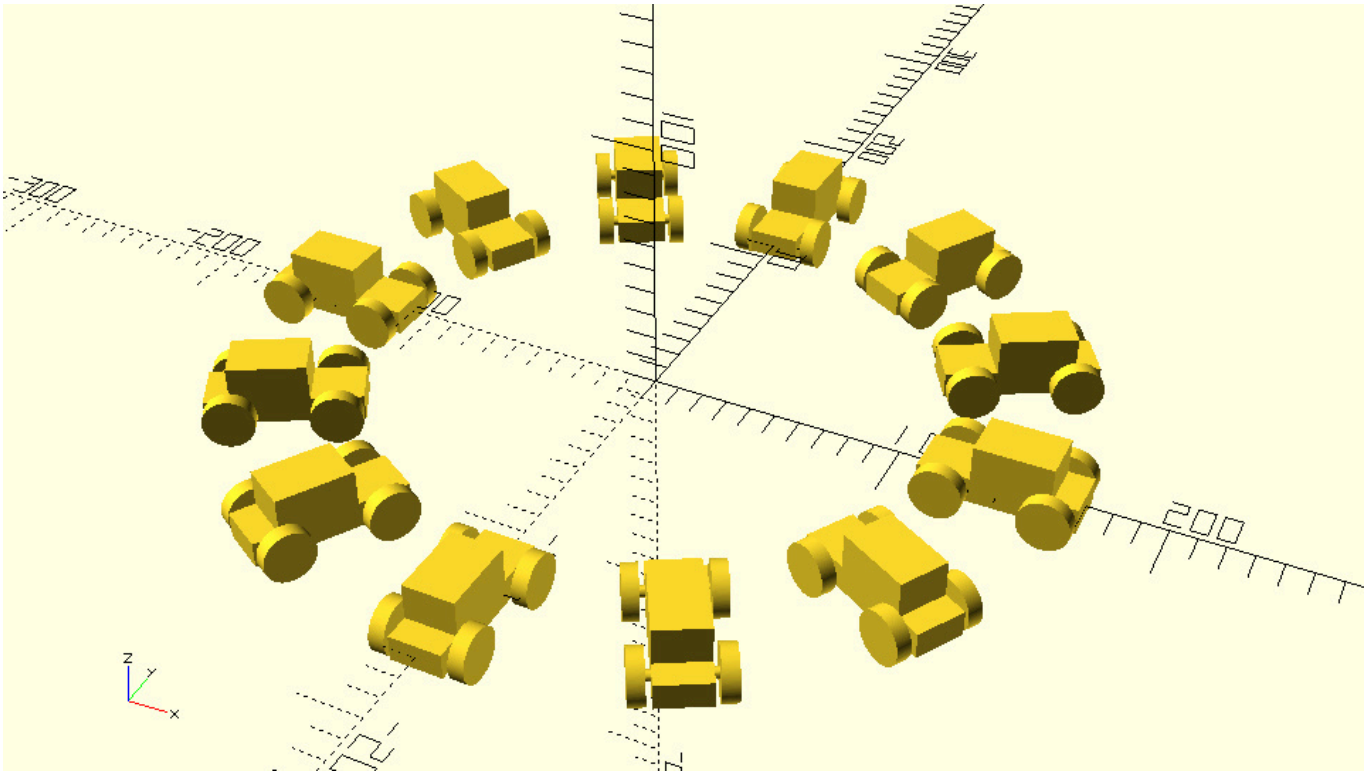


Addition, subtraction, multiplication and division are represented in OpenSCAD with the signs +, -, * and /. Apart from these fundamental operations, there are also a number of additional mathematical operations that can be useful when building more complex models. Two examples of this are the cosine and sine functions that you used to define a circular pattern of cars. Specifically, you used the cosine and sine functions to transform the polar coordinates of each car into Cartesian coordinates in order to translate it in its proper position. You can find all available math functions briefly listed in the cheat sheet.

Code

circular_pattern_of_cars.scad

```
…
r = 140; // pattern radius
n = 12; // number of cars
step = 360/n;
for (i=[0:step:359]) {
    angle = i;
    dx = r*cos(angle);
    dy = r*sin(angle);
    translate([dx,dy,0])
        rotate([0,0,angle])
        car();
}
…
```

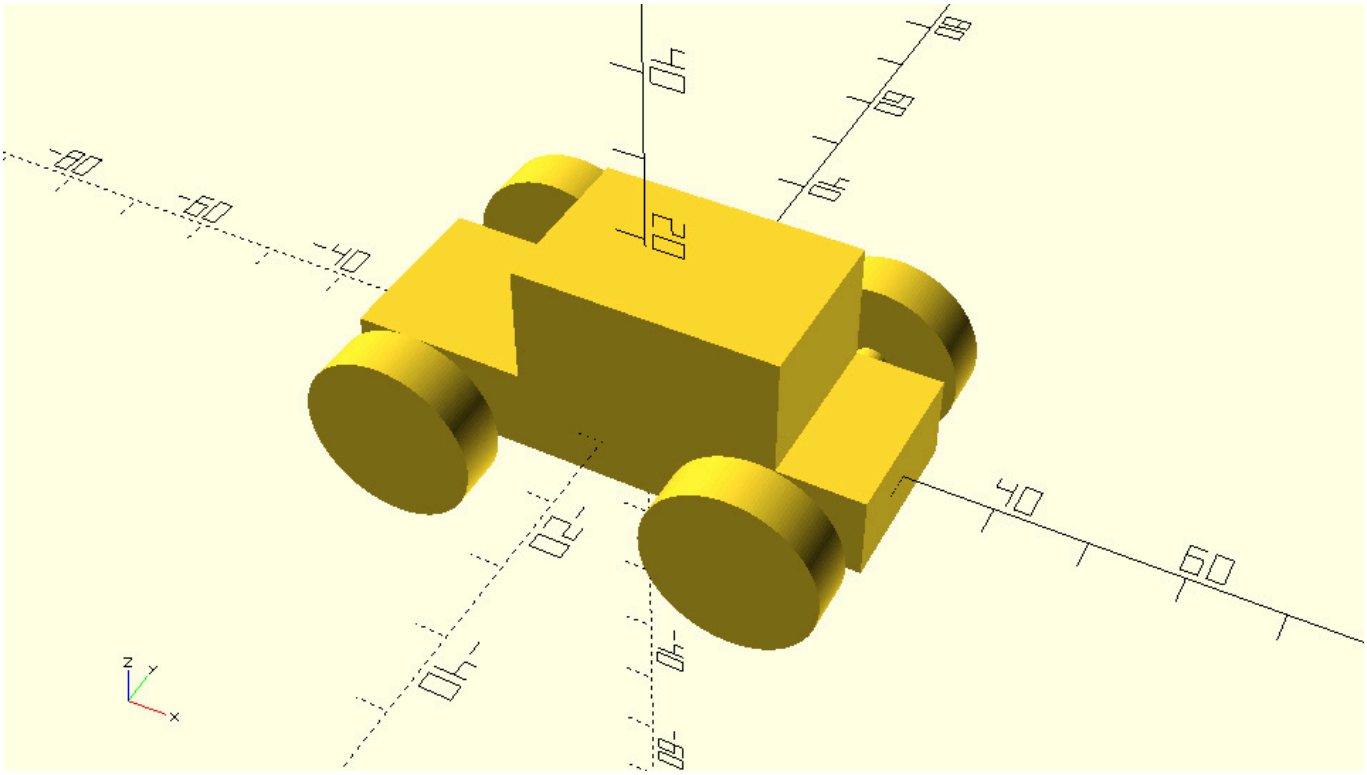


In the above case you are not only using available mathematical operations in your script, but you are also defining two additional variables dx and dy to store the result of your calculations in order to increase the readability of your script. This is something that could also be done in your car models. Take for example the following car model.

Code

car.scad

```
use <vehicle_parts.scad>
$fa = 1;
$fs = 0.4;

wheelbase = 40;
track = 35;

// Body
body();
// Front left wheel
translate([-wheelbase/2,-track/2,0])
    simple_wheel();
 // Front right wheel
translate([-wheelbase/2,track/2,0])
    simple_wheel();
// Rear left wheel
translate([wheelbase/2,-track/2,0])
    simple_wheel();
// Rear right wheel
translate([wheelbase/2,track/2,0])
    simple_wheel();
// Front axle
translate([-wheelbase/2,0,0])
    axle(track=track);
// Rear axle
translate([wheelbase/2,0,0])
    axle(track=track);
```



In the above model, mathematical operations are used to calculate the required amount of translation for each wheel and axle along the X and Y axis.

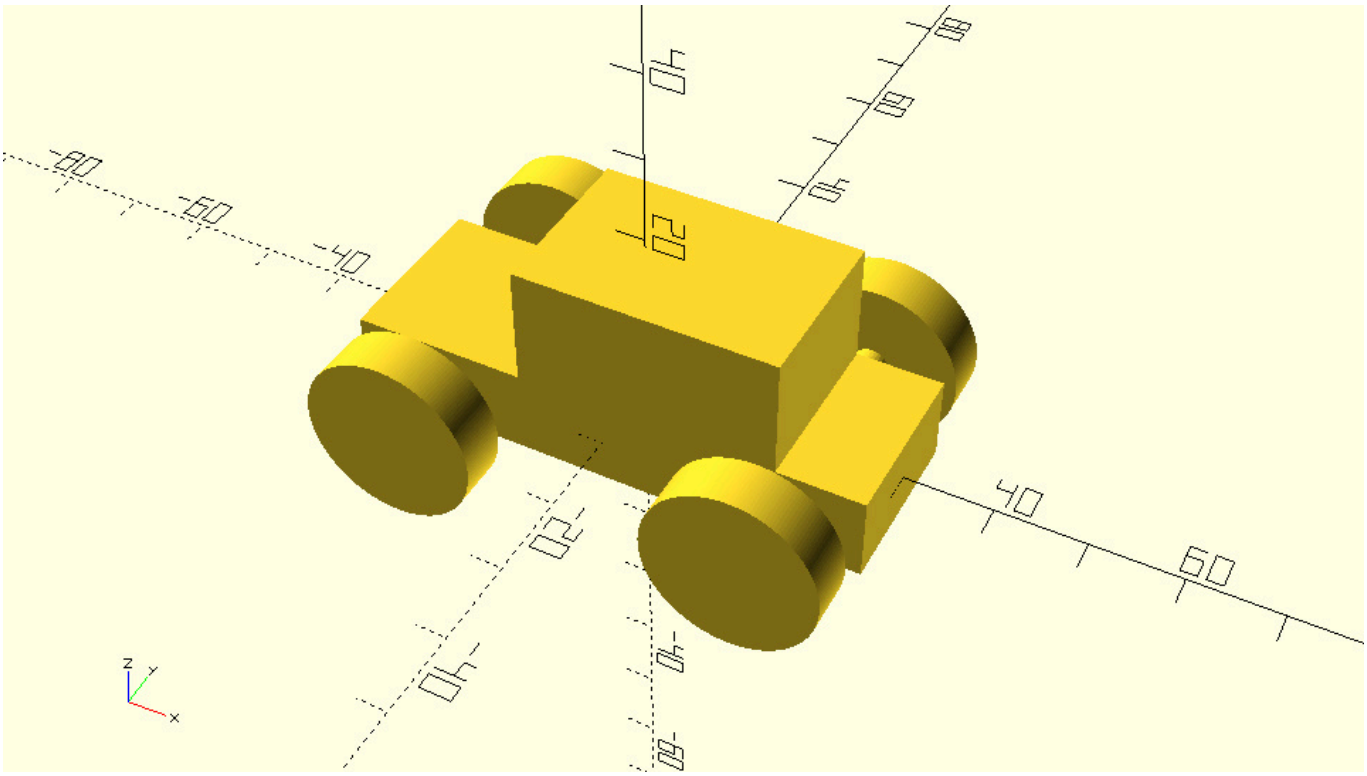
Exercise

Modify the above script in order to improve its readability and avoid repeating the same mathematical operations multiple times. To do so you should introduce two new variables named half_wheelbase and half_track. Use the corresponding mathematical calculation to set these variables equal to half the value of the wheelbase and the track variables accordingly. Replace the repeating mathematical operations in the translation commands with the use of these two variables.

Code[Collapse]

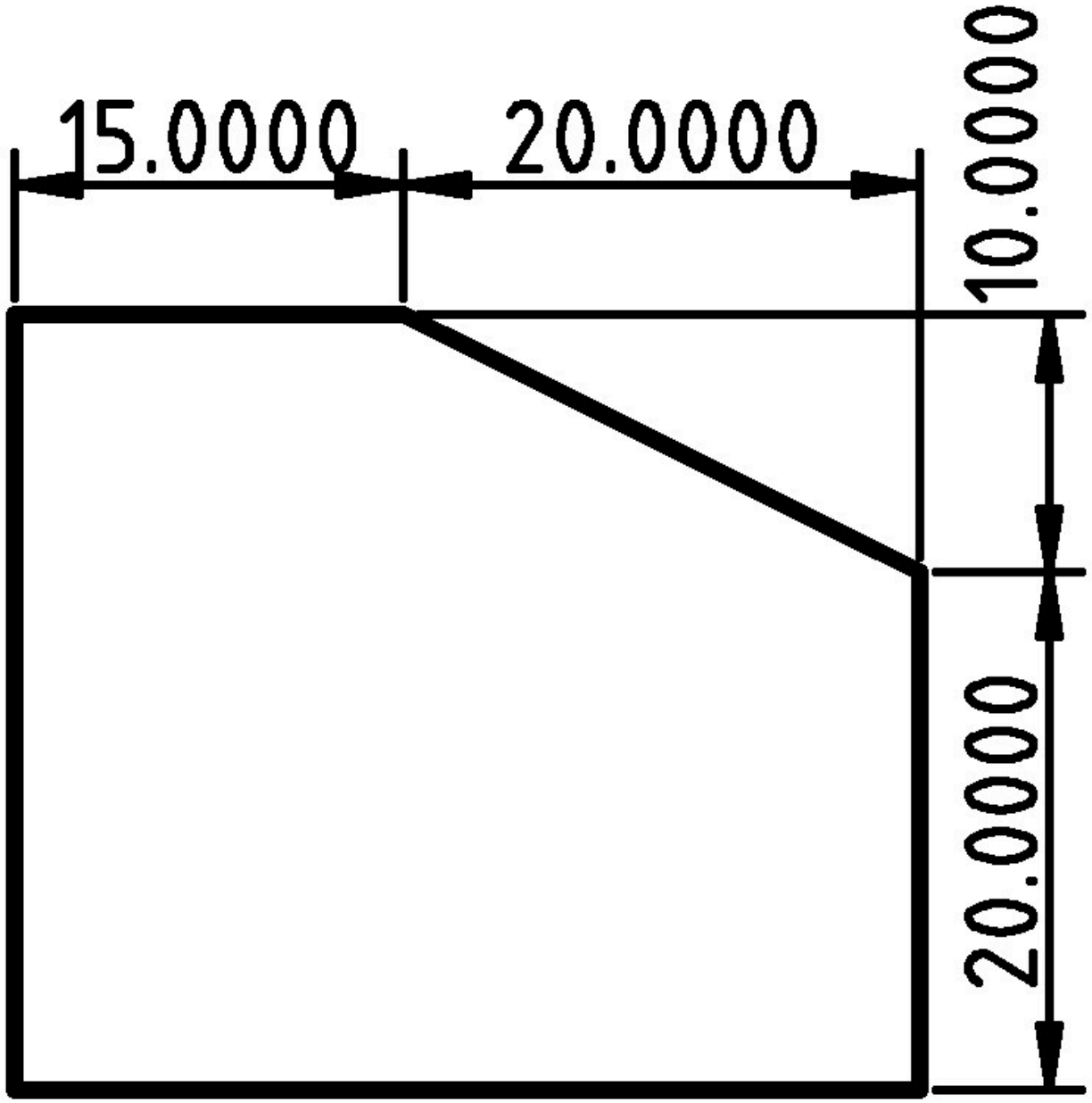
car_with_additional_variables.scad

```
use <vehicle_parts.scad>
$fa = 1;
$fs = 0.4;

wheelbase = 40;
track = 35;

half_wheelbase = wheelbase/2;
half_track = track/2;

// Body
body();
// Front left wheel
translate([-half_wheelbase,-half_track,0])
    simple_wheel();
 // Front right wheel
translate([-half_wheelbase,half_track,0])
    simple_wheel();
// Rear left wheel
translate([half_wheelbase,-half_track,0])
    simple_wheel();
// Rear right wheel
translate([half_wheelbase,half_track,0])
    simple_wheel();
// Front axle
translate([-half_wheelbase,0,0])
    axle(track=track);
// Rear axle
translate([half_wheelbase,0,0])
    axle(track=track);
```



Creating any 2D object with the polygon primitive

Aside from the circle and square 2D primitives, there is another primitive that lets you design practically any 2D object. This is the polygon primitive, which lets you define 2D objects by providing a list that contains the coordinates of their points. Let's say you want to design the following 2D part.

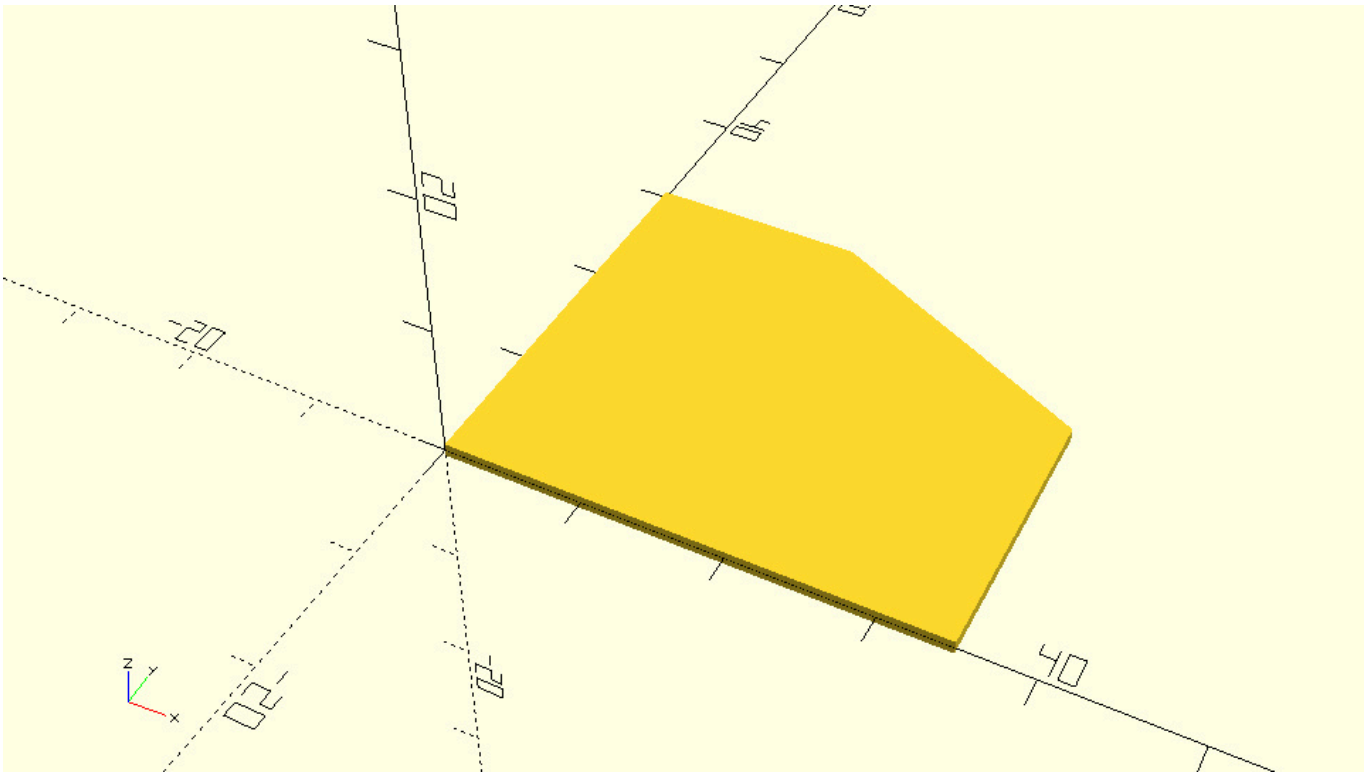


One way to go about designing this part without using the polygon primitive would be to start from a square that corresponds to the outer dimensions of this part and then subtract a properly rotated and translated square from its top right corner. Calculating the proper angle of rotation and amount of translation would be a time-consuming task. Additionally, following this strategy for a more complex object wouldn't be possible. Instead, you can create this object using the polygon primitive in the following way.

Code

profile_1_polygon.scad

```
p0 = [0, 0];
p1 = [0, 30];
p2 = [15, 30];
p3 = [35, 20];
p4 = [35, 0];
points = [p0, p1, p2, p3, p4];
polygon(points);
```



There are a few things you should notice regarding the use of the polygon primitive. The polygon primitive uses a list of points as inputs. The points, or vertices, are represented using pairs of X and Y coordinates, and are provided in order. When defining the list, you may start with any vertex you like and you can traverse them in either a clockwise or counterclockwise order. In the example above, the first vertex is at the origin (0,0), while the remaining vertices are listed in a clockwise direction. All vertices (pairs of X and Y coordinates) p0, p1, ..., p4 are placed inside a list named points. This list is then passed to the polygon command to create the corresponding object.

Whether a variable has only a single value or it's a list of values, you can print its content on the console using the echo command.

Code

```
…
echo(points);
…
```

The output in the console is: [[0, 0], [0, 30], [15, 30], [35, 20], [35, 0]]

Naming each point separately (p0, p1, ...) is not required but it's recommended to better keep track of your design. You could also directly define the list of points to be passed to the polygon command.

Code

```
…
points = [[0, 0], [0, 30], [15, 30], [35, 20], [35, 0]];
…
```

Moreover, you don't even have to define a variable to store the list of points. You can directly define the list of points when calling the polygon command.

Code

```
…
polygon([[0, 0], [0, 30], [15, 30], [35, 20], [35, 0]]);
…
```

The above practices are not recommended. Instead the use of additional variables as in the first example is encouraged in order to make your scripts more readable and extendable.

You could also parameterize the definition of the points' coordinates according to the given dimensions, which will give you the ability to rapidly modify the dimensions of your object. This can be achieved by introducing one variable for each given dimension and by defining the coordinates of the points using appropriate mathematical expressions.

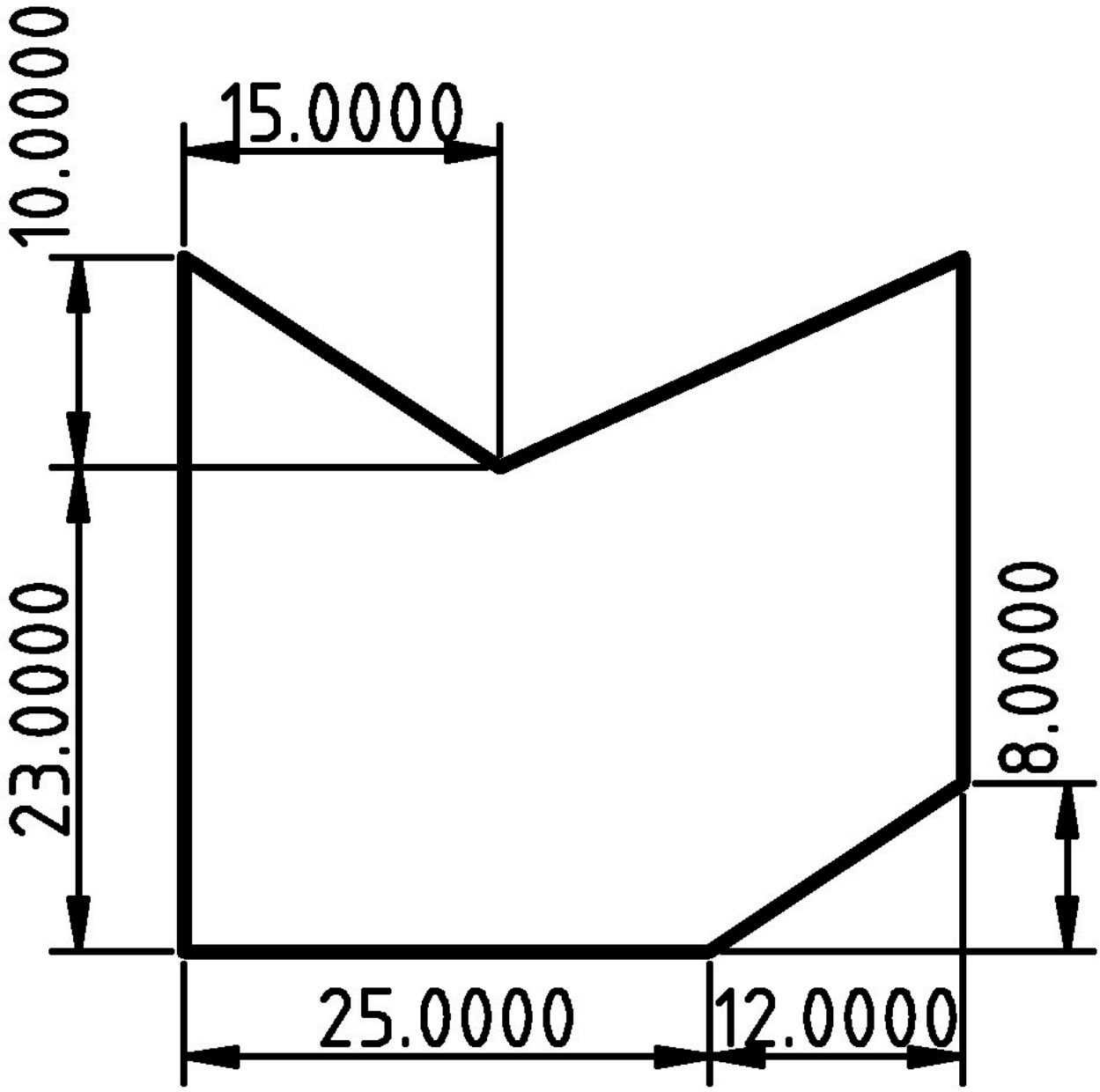
Code

profile_1_polygon_parametric.scad

```
// Given dimensions
d1 = 15;
d2 = 20;
h1 = 20;
h2 = 10;
// Points
p0 = [0, 0];
p1 = [0, h1 + h2];
p2 = [d1, h1 + h2];
p3 = [d1 + d2, h1];
p4 = [d1 + d2, 0];
points = [p0, p1, p2, p3, p4];
// Polygon
polygon(points);
```

Exercise

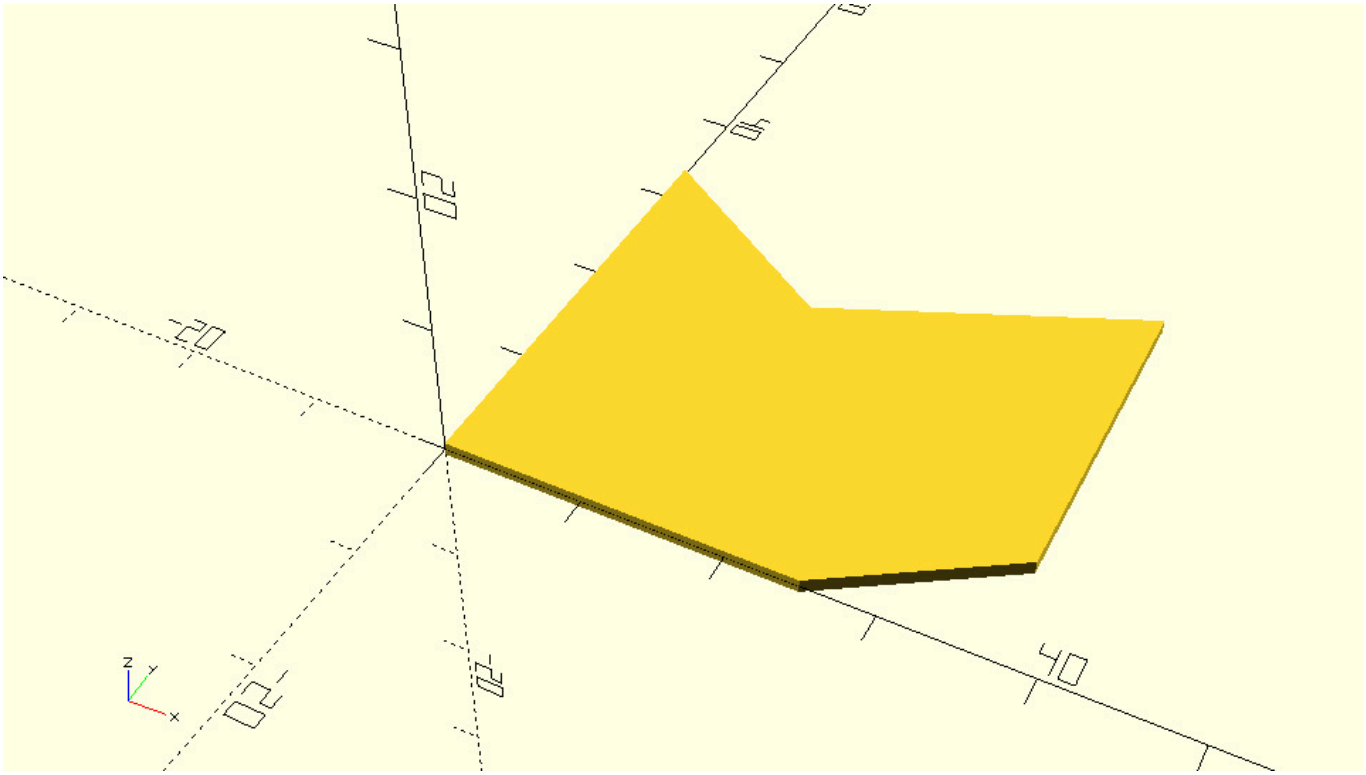
Create the following 2D object using the polygon primitive. To do this you will need to define a list that contains the pairs of X and Y coordinates of the object's points. Remember that the points should be defined in an appropriate order. You should first store the coordinates of each point on separate variables named p0, p1, p2, … and then define a list of all points and store it in a variable named points. This list will be passed on the polygon command. The definition of each point's coordinates should be parametric in relation to the given dimensions which can be achieved by using appropriate mathematical expressions. To do this you will also need to define a variable for each given dimension.



Code[Collapse]

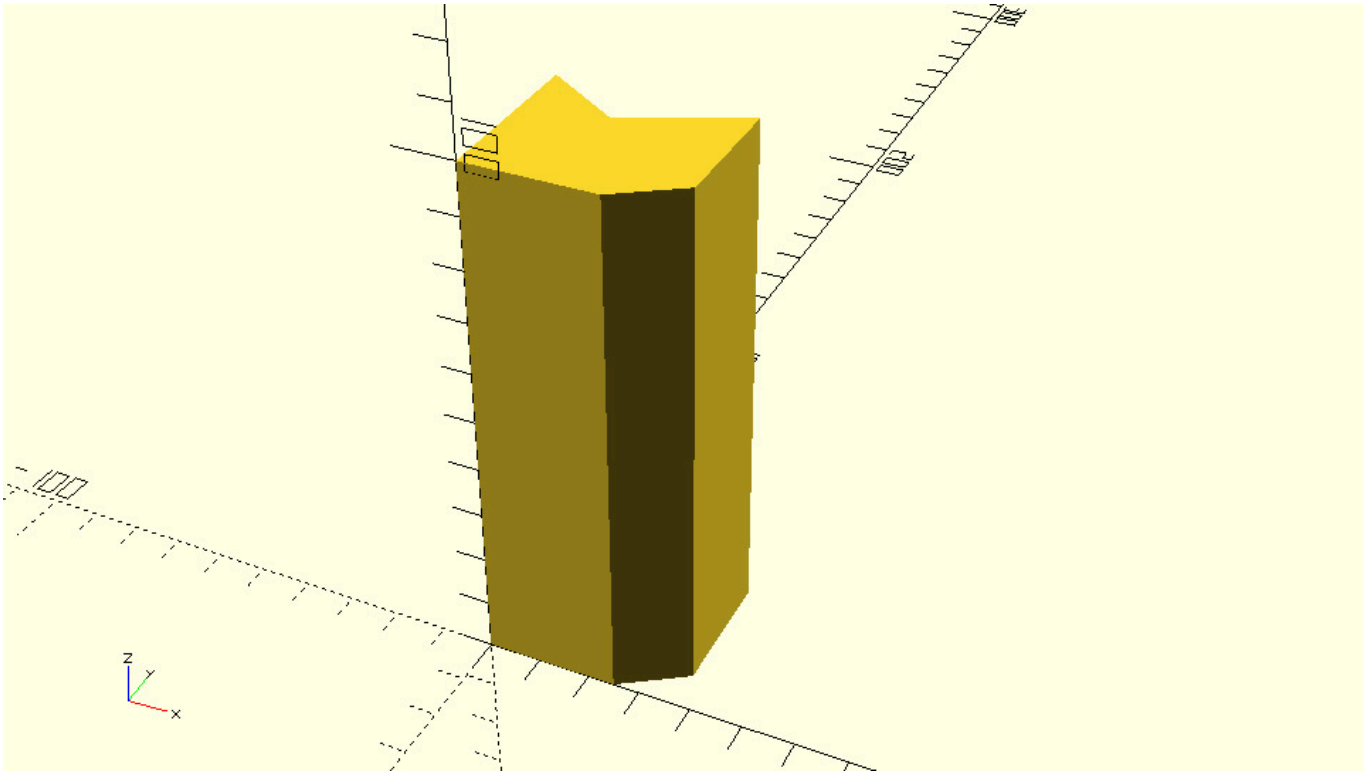
profile_2_polygon.scad

```
// Given dimensions
h1 = 23;
h2 = 10;
h3 = 8;
d1 = 25;
d2 = 12;
d3 = 15;
// Points
p0 = [0, 0];
p1 = [0, h1 + h2];
p2 = [d3, h1];
p3 = [d1 + d2, h1 + h2];
p4 = [d1 + d2, h3];
p5 = [d1, 0];
points = [p0, p1, p2, p3, p4, p5];
// Polygon
polygon(points);
```



Exercise

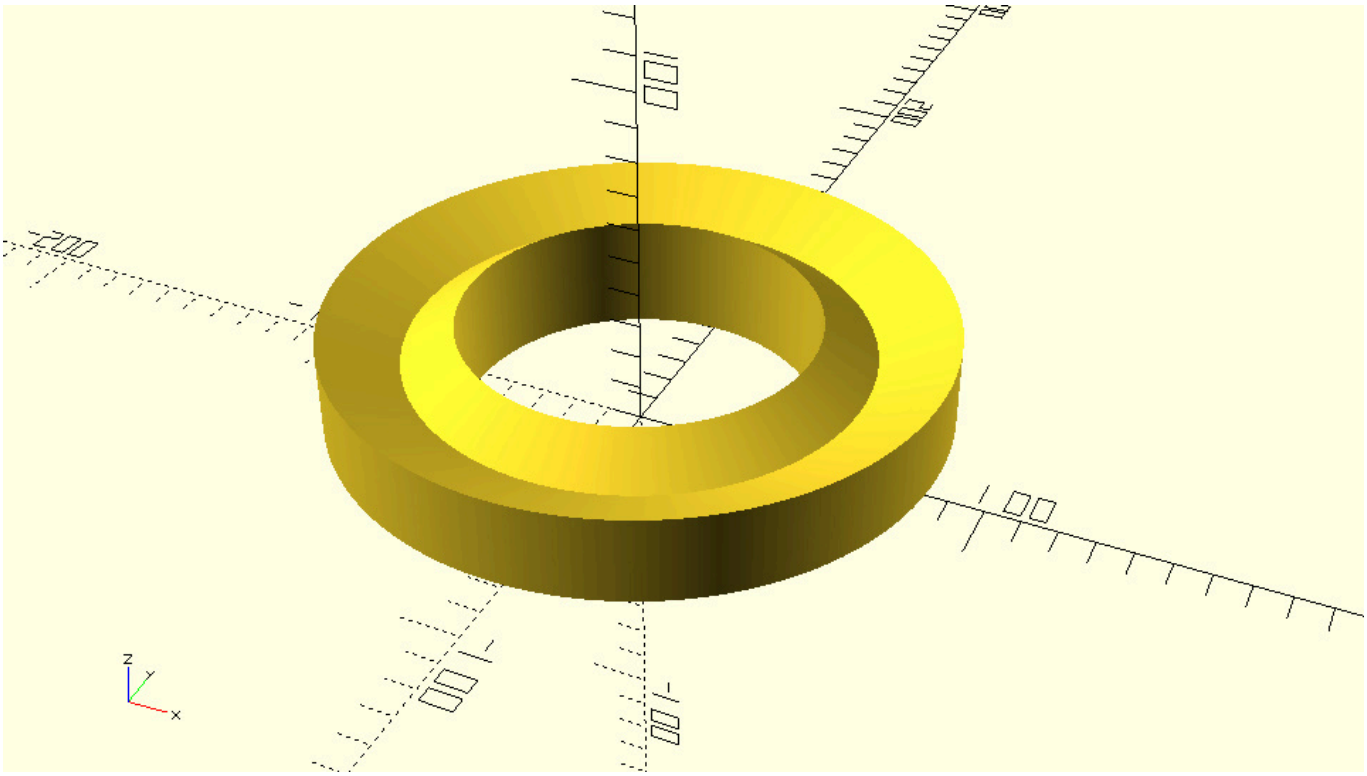
Using the linear_extrude and rotate_extrude commands create a tube and a ring respectively that have the above profile. The tube should have a height of 100 units. The ring should have an inner diameter of 100 units. How many units do you need to translate the 2d profile along the positive direction of the X axis to achieve this?



Code[Collapse]

linearly_extruded_polygon.scad

```
…
linear_extrude(height=100)
    polygon(points);
…
```

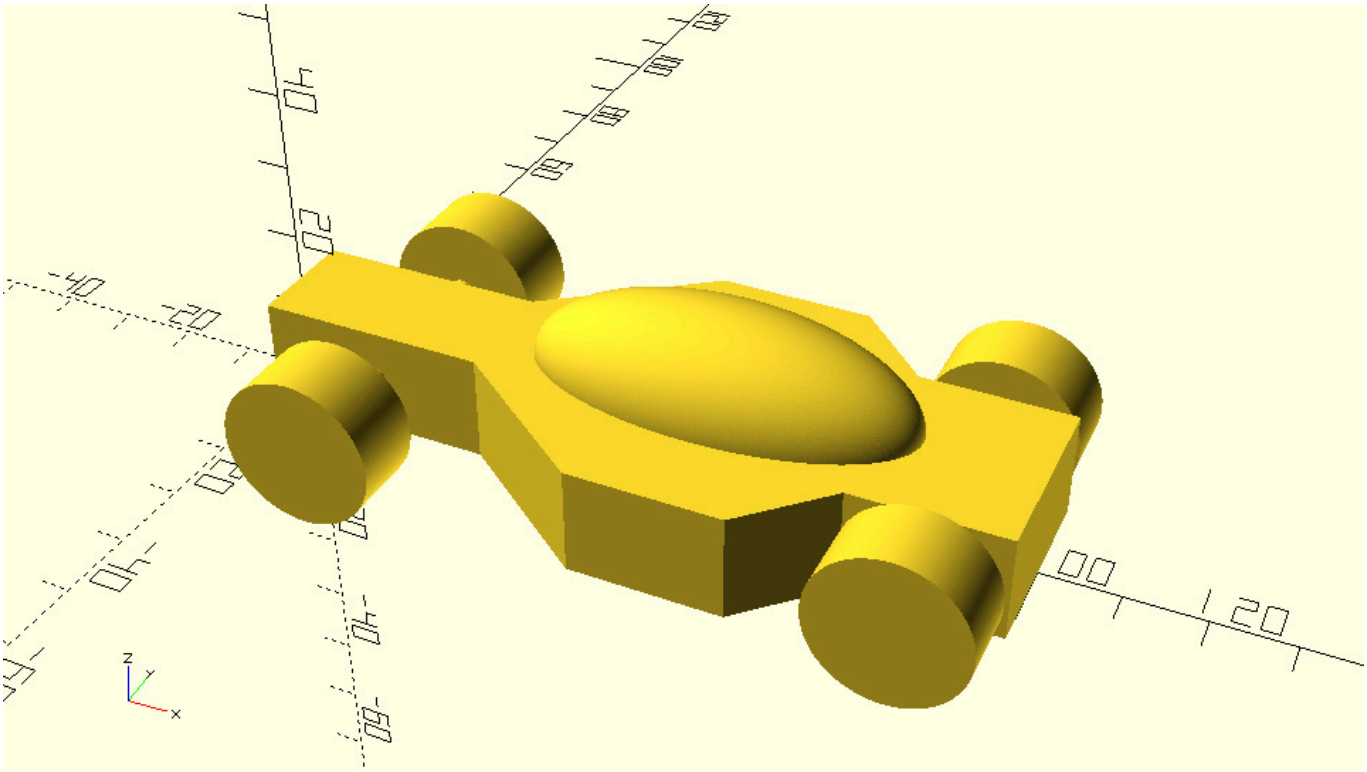


Code[Collapse]

rotationaly_extruded_polygon.scad

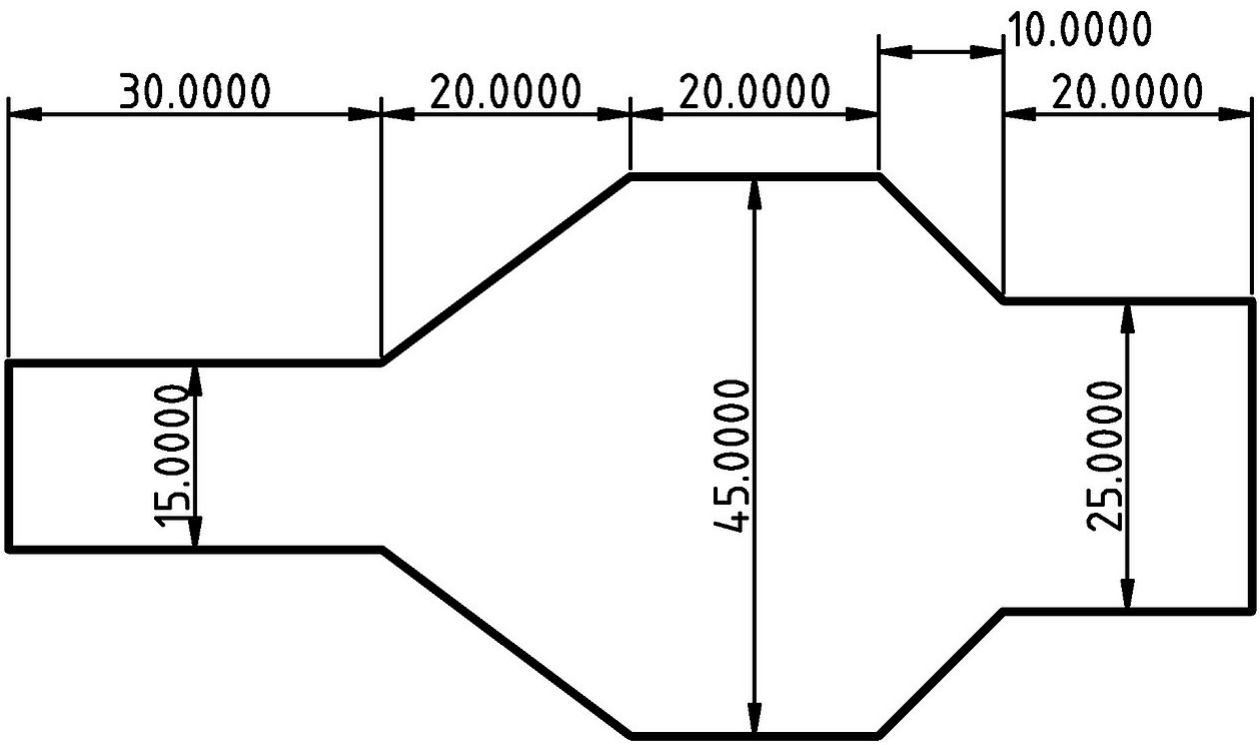
```
…
rotate_extrude(angle=360)
    translate([50,0,0])
    polygon(points);
…
```

It's time to use your new skills to create UwU racing car!



Exercise

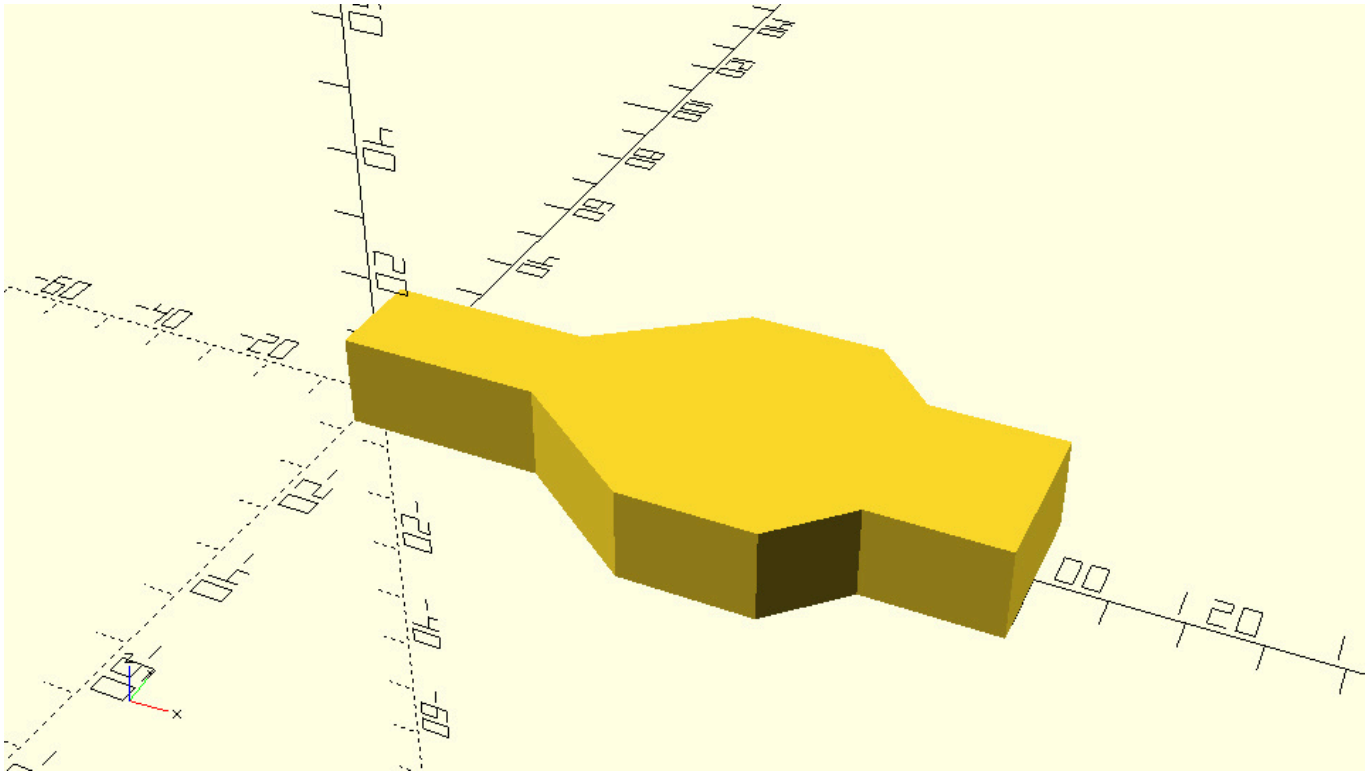
Create the above car body using the polygon command. To do so you will have to define each point of the design, add them all on a list and pass this list to the polygon command. The definition of each point's coordinates should be parametric in relation to the given dimensions. Remember that in order to do this you will have to define a variable for each given dimension and calculate each point's coordinates from these variables using appropriate mathematical expressions. You should extrude the created 2D profile to a height of 14 units.



Code[Collapse]

racing_car_body.scad

```
// model parameters
d1=30;
d2=20;
d3=20;
d4=10;
d5=20;

w1=15;
w2=45;
w3=25;

h=14;

// right side points
p0 = [0, w1/2];
p1 = [d1, w1/2];
p2 = [d1 + d2, w2/2];
p3 = [d1 + d2 + d3, w2/2];
p4 = [d1 + d2 + d3 + d4, w3/2];
p5 = [d1 + d2 + d3 + d4 + d5, w3/2];

// left side points
p6 = [d1 + d2 + d3 + d4 + d5, -w3/2];
p7 = [d1 + d2 + d3 + d4, -w3/2];
p8 = [d1 + d2 + d3, -w2/2];
p9 = [d1 + d2, -w2/2];
p10 = [d1, -w1/2];
p11 = [0, -w1/2];

// all points
points = [p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11];

// extruded body profile
linear_extrude(height=h)
    polygon(points);
```



Exercise

As mentioned previously, you can use additional variables to increase the readability of your script and to avoid repeating mathematical operations. Can you find a way to do so in the above script?

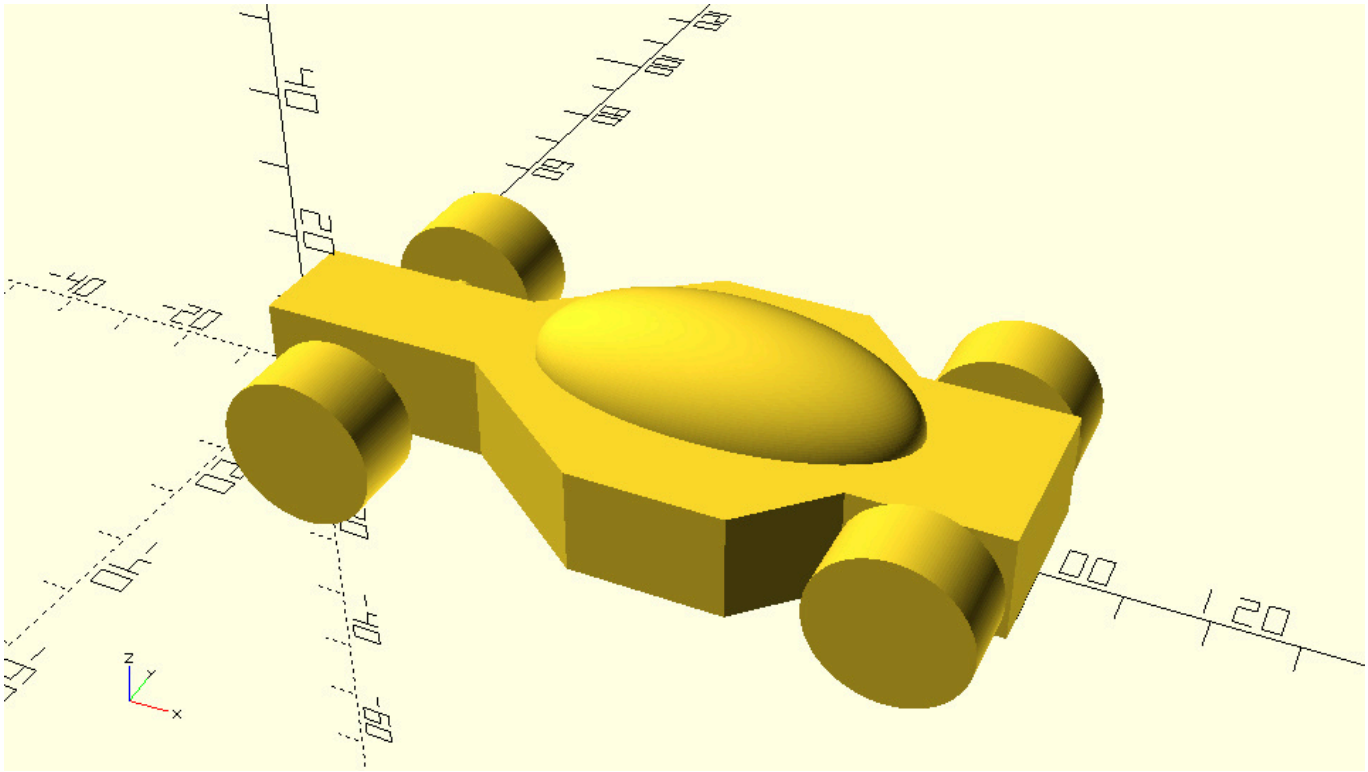
Code[Collapse]

racing_car_body_with_extra_variables.scad

```
// model parameters
d1=30;
d2=20;
d3=20;
d4=10;
d5=20;

w1=15;
w2=45;
w3=25;

h=14;

// distances to lengths
l1 = d1;
l2 = d1 + d2;
l3 = d1 + d2 + d3;
l4 = d1 + d2 + d3 + d4;
l5 = d1 + d2 + d3 + d4 + d5;

// right side points
p0 = [0, w1/2];
p1 = [l1, w1/2];
p2 = [l2, w2/2];
p3 = [l3, w2/2];
p4 = [l4, w3/2];
p5 = [l5, w3/2];

// left side points
p6 = [l5, -w3/2];
p7 = [l4, -w3/2];
p8 = [l3, -w2/2];
p9 = [l2, -w2/2];
p10 = [l1, -w1/2];
p11 = [0, -w1/2];

// all points
points = [p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11];

// extruded body profile
linear_extrude(height=h)
    polygon(points);
```

Exercise

Try completing the racing car design by adding the remaining objects to the above body.


z
y
x

Code[Collapse]

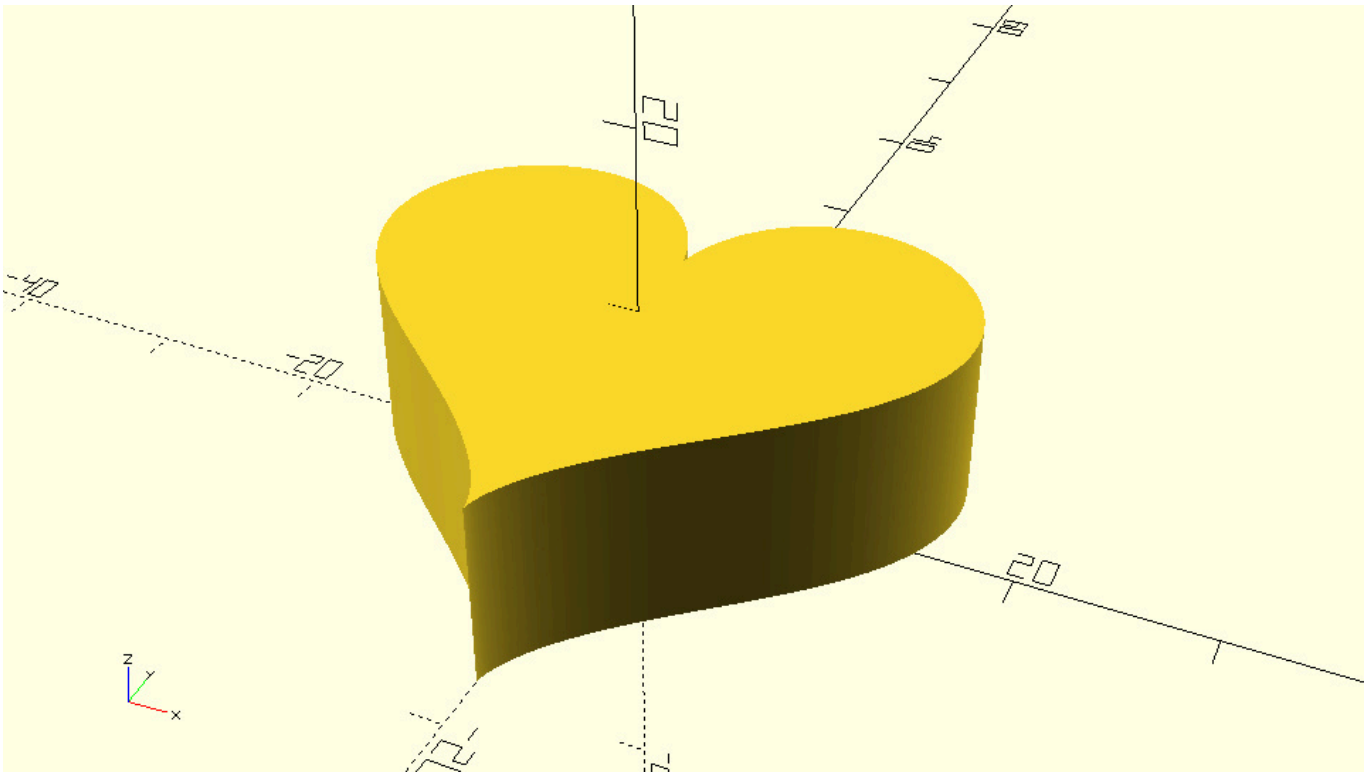
racing_car.scad

```
use <vehicle_parts.scad>

$fa = 1;
$fs = 0.4;

// model parameters
d1=30;
d2=20;
d3=20;
d4=10;
d5=20;

w1=15;
w2=45;
w3=25;

h=14;
track=40;

// distances to lengths
l1 = d1;
l2 = d1 + d2;
l3 = d1 + d2 + d3;
l4 = d1 + d2 + d3 + d4;
l5 = d1 + d2 + d3 + d4 + d5;

// right side points
p0 = [0, w1/2];
p1 = [l1, w1/2];
p2 = [l2, w2/2];
p3 = [l3, w2/2];
p4 = [l4, w3/2];
p5 = [l5, w3/2];

// left side points
p6 = [l5, -w3/2];
p7 = [l4, -w3/2];
p8 = [l3, -w2/2];
p9 = [l2, -w2/2];
p10 = [l1, -w1/2];
p11 = [0, -w1/2];

// all points
points = [p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11];

// extruded body profile
linear_extrude(height=h)
    polygon(points);

// canopy
translate([d1+d2+d3/2,0,h])
    resize([d2+d3+d4,w2/2,w2/2])
    sphere(d=w2/2);

// axles
l_front_axle = d1/2;
l_rear_axle = d1 + d2 + d3 + d4 + d5/2;
half_track = track/2;

translate([l_front_axle,0,h/2])
    axle(track=track);
translate([l_rear_axle,0,h/2])
    axle(track=track);

// wheels
translate([l_front_axle,half_track,h/2])
    simple_wheel(wheel_width=10);
translate([l_front_axle,-half_track,h/2])
    simple_wheel(wheel_width=10);

translate([l_rear_axle,half_track,h/2])
    simple_wheel(wheel_width=10);
```

```
translate([l_rear_axle,-half_track,h/2])
    simple_wheel(wheel_width=10);
```

Creating more complex object using the polygon primitive and math

From the above examples it should be quite obvious that the polygon primitive opens up possibilities to create objects that would hardly be possible by just using the fundamental 2D or 3D primitives. In these examples you created custom 2D profiles by defining the coordinates of their points according to a given design. To unlock the true power of the polygon command though, and to create even more complex and impressive designs, you have to programmatically define the points of your profiles using math. This is because defining each point separately is not scalable to the hundreds of points that are required to design smooth non-square profiles. One example of this is the following heart. Can you manually define the required points to create it? There is no way.



Instead of manually defining each point, which would be practically impossible, this model was programmatically defined using the following parametric equations.

$x = 16 \cdot \sin(t)^3$
$y = 13 \cdot \cos(t) - 5 \cdot \cos(2 \cdot t) - 2 \cdot \cos(3 \cdot t) - \cos(4 \cdot t)$

When the range of the t variable covers values from 0 to 360 degrees, the above equations give the X and Y coordinates for the outline of the heart, starting from the top middle point and moving in a clockwise direction. Using the above equations, a list that contains each point's coordinates can be generated in the following way.

Code

```
points = [ for (t=[0:step:359.999]) [16*pow(sin(t),3), 13*cos(t) - 5*cos(2*t) - 2*cos(3*t) -
cos(4*t)]];
```

There are a few things you should notice about the syntax of list generation. First the name of the variable where the list will be stored is typed out. Then follows the equal sign (as in every variable assignment) and a pair of square brackets. Inside the pair of square brackets, the first thing that is typed out is the keyword *for*. After the keyword *for* follows a pair of parentheses inside of which the consecutive values that the corresponding variable is going to take are defined. This variable is similar to the for loop variable encountered in for loops. The number of elements that the generated list is going to have is equal to the number of values that this variable is going to take. For every value that this variable takes, one element of the list is defined. What each element of the list is going to be, is specified after the closing parenthesis. In this case each element of the generated list is itself a list that has two elements, one for each coordinate of the corresponding point. The *t* variable goes from 0 to 360 which is the required range to produce the whole outline of the heart. Since the polygon shouldn't include duplicate points, 359.999 is used instead of 360. By choosing a smaller or bigger value for the step variable, the amount of points that will be created can be controlled. Specifically, in order to create n points the step variable needs to be defined in the following way.

Code

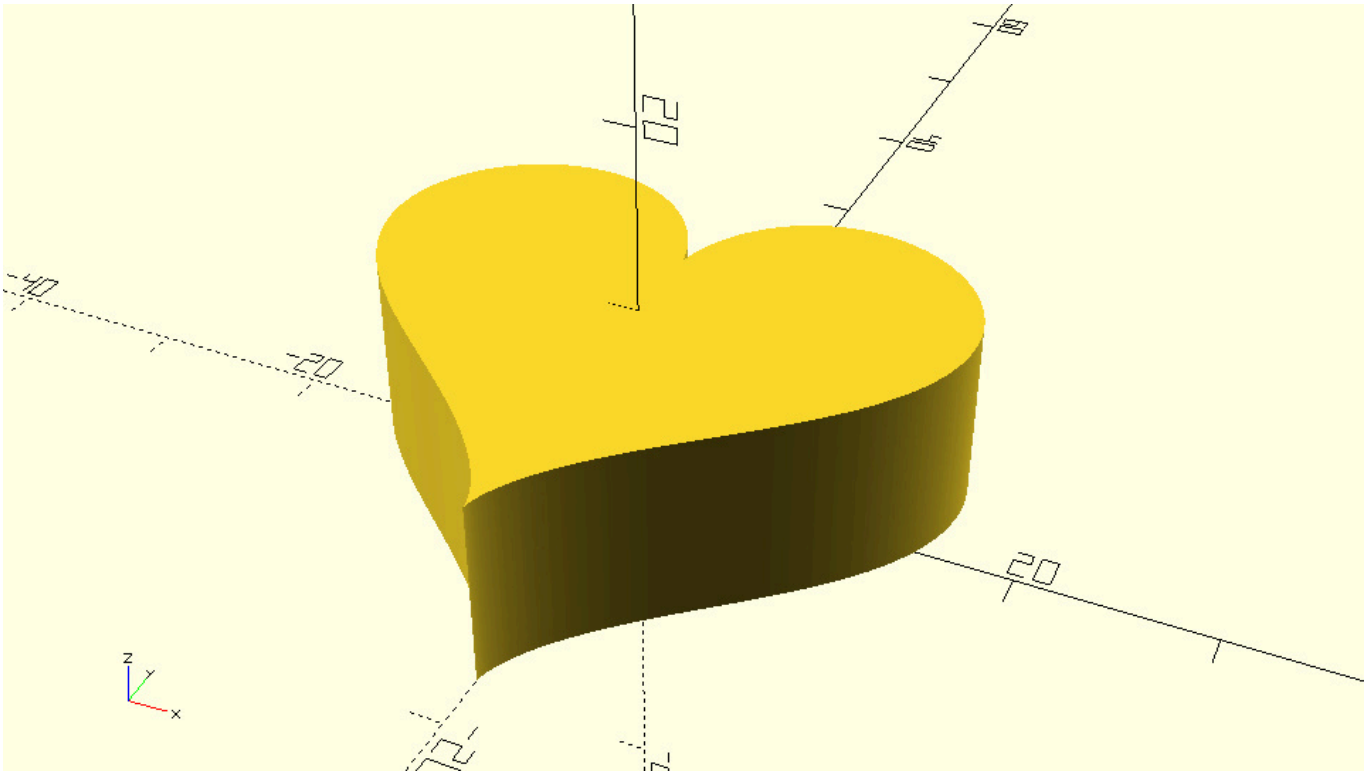
```
step = 360/n;
```

Putting all these together, the heart can be created with the following script.

Code

heart.scad

```
n = 500;
h = 10;
step = 360/n;
points = [ for (t=[0:step:359.999]) [16*pow(sin(t),3), 13*cos(t) - 5*cos(2*t) - 2*cos(3*t) -
cos(4*t)]];
linear_extrude(height=h)
    polygon(points);
```



You can see that by using 500 points the resolution of the outline is very good.

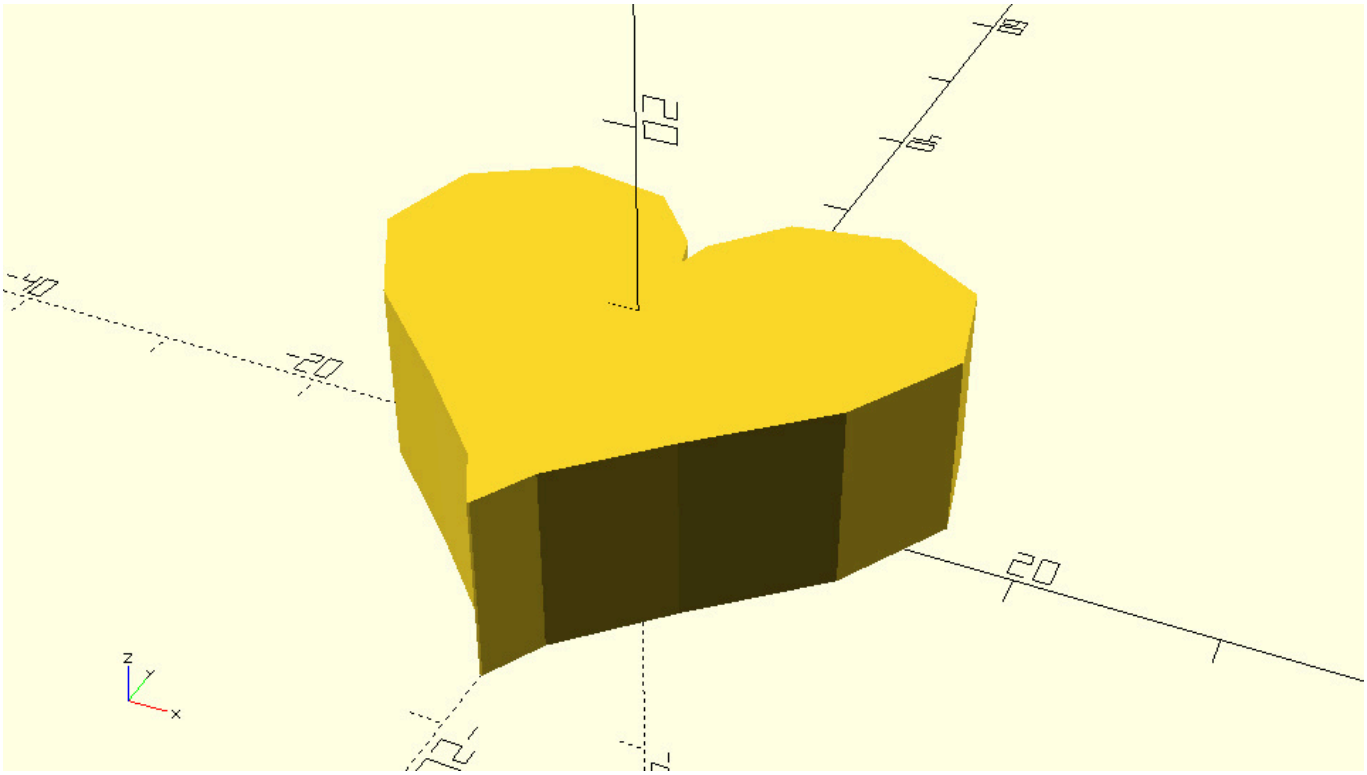
Exercise

Modify the above script so that the outline of the heart is consisted out of 20 points.

Code[Collapse]

heart_low_poly.scad

```
…
n = 20;
…
```



Using more or less points is up to you. If you want to create an object that closely resembles the underlying mathematical equations, you should increase the number of points. Instead, if you opt for a low poly style you should decrease the number of points.

Exercise

Time for some quick list generation practice before you move on. Use the newly introduced syntax (variable = [for (i=[start:step:end]) …];) to generate the following lists:

i. [1, 2, 3, 4, 5, 6]
ii. [10, 8, 6, 4, 2, 0, -2]
iii. [[3, 30], [4, 40], [5, 50], [6, 60]]

Solution for exercise i, [1, 2, 3, 4, 5, 6]:

Code[Collapse]

```
x = [ for (i=[1:6]) i];
```

Solution for exercise ii, [10, 8, 6, 4, 2, 0, -2]:

Code[Collapse]

```
x = [ for (i=[10:-2:-2]) i];
```

Solution for exercise iii, [[3, 30], [4, 40], [5, 50], [6, 60]]:

Code[Collapse]

```
x = [ for (i=[3:6]) [i, i*10]];
```

Note that when no step is provided, a default of 1 is used.

OpenSCAD also allows you to define your own mathematical functions, which can be useful when a mathematical expression is particularly long, or when you would like to use it multiple times. These functions work similarly to modules, except that instead of defining a design they define a mathematical process. After a function has been defined, you can use it by invoking its name and providing the necessary input parameters. For example, you can define a function that accepts the parameter t, and returns the X and Y coordinates of the corresponding point of the heart's outline.

Code

```
function heart_coordinates(t) = [16*pow(sin(t),3), 13*cos(t) - 5*cos(2*t) - 2*cos(3*t) - cos(4*t)];
```

In this case the script that creates the heart would take the following form.

Code

```
n = 500;
h = 10;
step = 360/n;
function heart_coordinates(t) = [16*pow(sin(t),3), 13*cos(t) - 5*cos(2*t) - 2*cos(3*t) - cos(4*t)];
points = [ for (t=[0:step:359.999]) heart_coordinates(t)];
linear_extrude(height=h)
    polygon(points);
```

There are a few things you should notice about the definition of a function. First the word function is typed out. Then follows the name that you want to give to the function. In this case the function is named heart_coordinates. After the name of the function follows a pair of parentheses that contains the input parameters of the function. Like module parameters, the input parameters in a function can have default values. In this case, the only input parameter is the polar angle of the current step t and it's not given a default value. After the closing parenthesis follows the equal sign and the command that defines the pair of X and Y coordinates of the heart's outline.

The generation of the list of points can also be turned in a function. This can be done in a similar fashion as follows.

Code

```
function heart_points(n=50) = [ for (t=[0:360/n:359.999]) heart_coordinates(t)];
```

In this case the script that creates the heart would take the following form.

Code

```
n=20;
h = 10;
function heart_coordinates(t) = [16*pow(sin(t),3), 13*cos(t) - 5*cos(2*t) - 2*cos(3*t) - cos(4*t)];
function heart_points(n=50) = [ for (t=[0:360/n:359.999]) heart_coordinates(t)];
points = heart_points(n=n);
linear_extrude(height=h)
    polygon(points);
```

In short you should remember that any command that returns a single value or a list of values can be turned into a function. Like modules, functions should be used to organize your designs and make them reusable.

Exercise

Since you have already defined the function for generating the list of points that is required to create a heart, it would be a good idea to also define a module that creates a heart. Create a module named heart. The module should have two input parameter h and n corresponding to the height of the heart and the number of points used. The module should call the heart_points function to create the required list of points and then pass that list to a polygon command to create the 2D profile of the heart. This profile should be extruded to the specified height.

Code[Collapse]

heart.scad

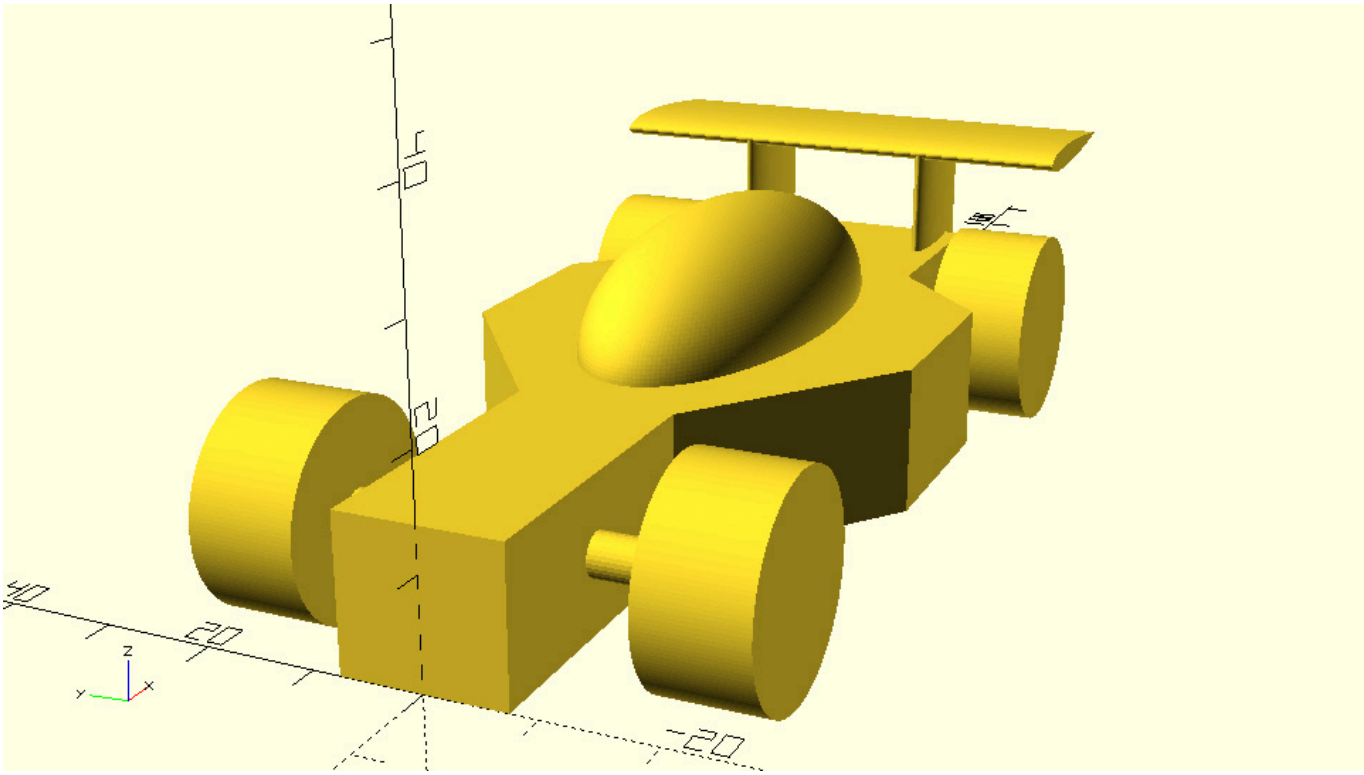
```
…
module heart(h=10, n=50) {
    points = heart_points(n=n);
    linear_extrude(height=h)
        polygon(points);
}
…
```

Exercise

You can save the heart_coordinates and heart_points functions along with the heart module on a script named heart.scad and add it on your libraries. Every time you want to include a heart on a design that you are working on, you can use the use command to make the functions and modules of this script available to you.

Another challenge

You are going to put your new skills into practice to create an aerodynamic spoiler for your racing car!



Exercise

You are going to use a symmetrical 4-digit NACA 00xx airfoil for your spoiler. The half thickness of such an airfoil on a given point x is given by the following formula:

$$y_t = 5t(0.2969\sqrt{x} - 0.1260x - 0.3516x^2 + 0.2843x^3 - 0.1015x^4)$$

In the above formula, x is the position along the chord with 0 corresponding to the leading edge and 1 to the trailing edge, while t is the maximum thickness of the airfoil expressed as a percentage of the chord. Multiplying t by 100 gives the last two digits in the NACA 4-digit denomination.

Create a function named naca_half_thickness. The function should have two input parameters: x and t. Given x and t, the function should return the half thickness of the corresponding NACA airfoil. The x and t input parameters shouldn't have any default value.

Code[Collapse]

naca_airfoil_module.scad

```
…
function naca_half_thickness(x,t) = 5*t*(0.2969*sqrt(x) - 0.1260*x - 0.3516*pow(x,2) +
0.2843*pow(x,3) - 0.1015*pow(x,4));
…
```

Exercise

Create a function named naca_top_coordinates. This function should return a list of the X and Y coordinates for the top half of the airfoil. The first point should correspond to the leading edge while the last point should correspond to the trailing edge.

The function should have two input parameters: t and n. The parameter t should correspond to the airfoil's maximum thickness, while the parameter n should correspond to the number of created points. The list of points should be generated using an appropriate list generation command. You will need to call the naca_half_thickness function.

Code[Collapse]

naca_airfoil_module.scad

```
…
function naca_top_coordinates(t,n) = [ for (x=[0:1/(n-1):1]) [x, naca_half_thickness(x,t)]];
…
```

Exercise

Create a similar function named naca_bottom_coordinates that returns a list of points for the bottom half of the airfoil. This time the points should be given in reverse order. The first point should correspond to the trailing edge while the last point should correspond to the leading edge. This is done so that when the lists that are generated from the naca_top_coordinates and naca_bottom_coordinates functions are joined, all point of the airfoil are defined in a clockwise direction starting from the leading edge, thus making the resulting list suitable for use with the polygon command.

Code[Collapse]

naca_airfoil_module.scad

```
…
function naca_bottom_coordinates(t,n) = [ for (x=[1:-1/(n-1):0]) [x, - naca_half_thickness(x,t)]];
…
```

Exercise

Create a function named naca_coordinates that joins the two lists of points. You can use OpenSCAD's built in function "concat" to join lists together. Pass both lists as inputs to concat to join them together.

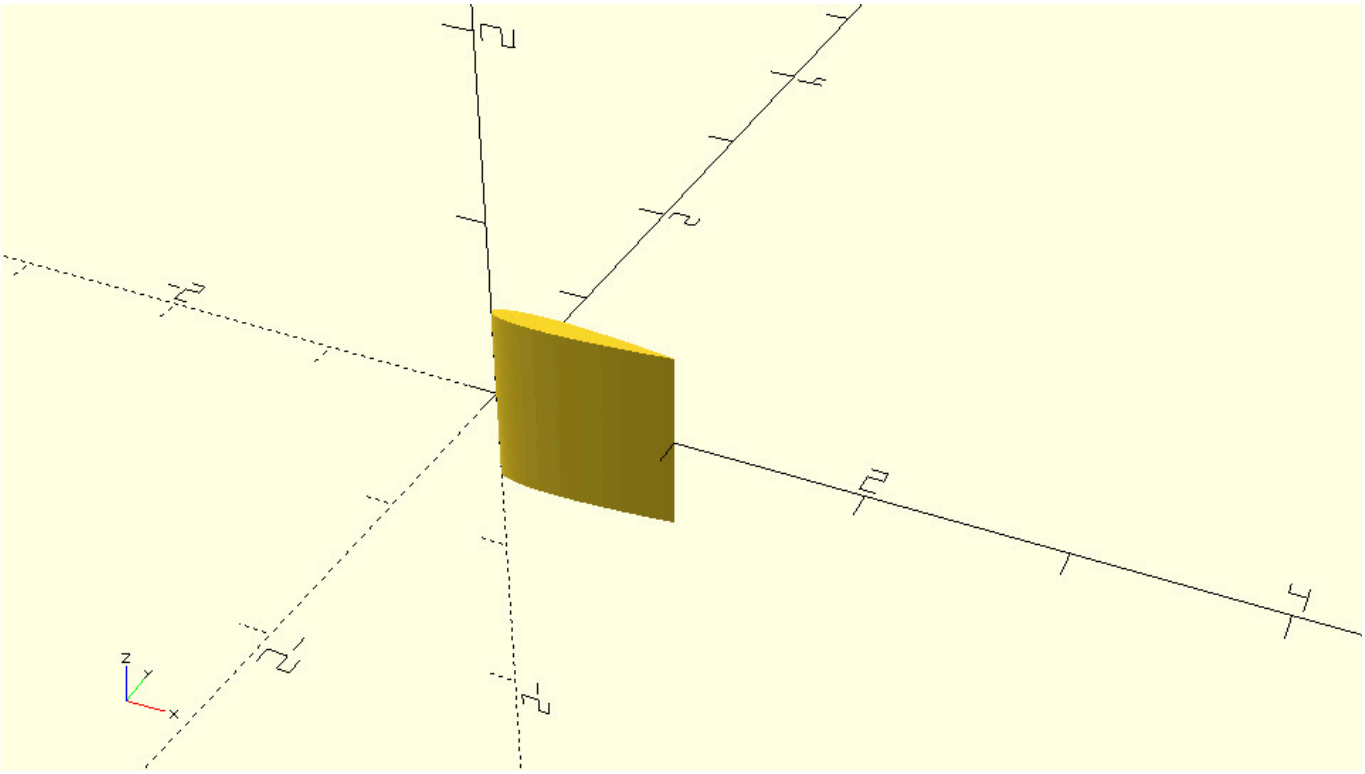
Code[Collapse]

naca_airfoil_module.scad

```
…
function naca_coordinates(t,n) = concat(naca_top_coordinates(t,n), naca_bottom_coordinates(t,n));
…
```

Exercise

Try using the naca_coordinates function to create a list that contains the points for an airfoil with a maximum thickness of 0.12 and with 300 points on each half. The list should be stored in a variable named points. The points variable should be passed to a polygon command to create the airfoils 2D profile.



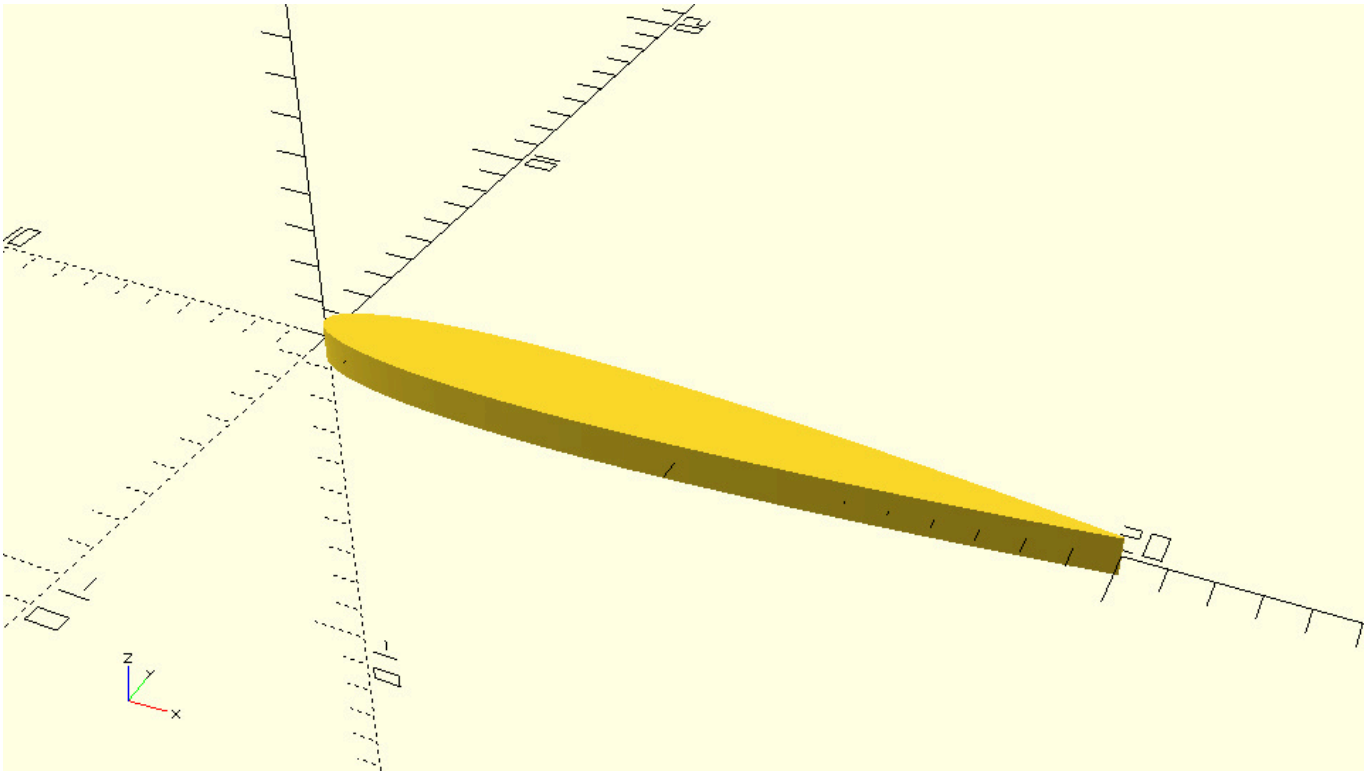
Code[Collapse]

small_airfoil_polygon.scad

```
…
points = naca_coordinates(t,n);
polygon(points);
…
```

Exercise

The chord of the above airfoil is 1 unit. Can you use an appropriate scale command to enlarge the airfoil? The desired chord should be defined on a variable named chord and the scale command should be defined in relation to this variable. Create an airfoil that has a chord of 20 units.



Code[Collapse]

scaled_airfoil_polygon.scad

```
…
chord = 20;
points = naca_coordinates(t=0.12,n=300);
scale([chord,chord,1])
    polygon(points);
…
```

Exercise

Turn the above script into a module named naca_airfoil. The module should have three input parameter, chord, t and n. There shouldn't be default values for any of the input parameters.

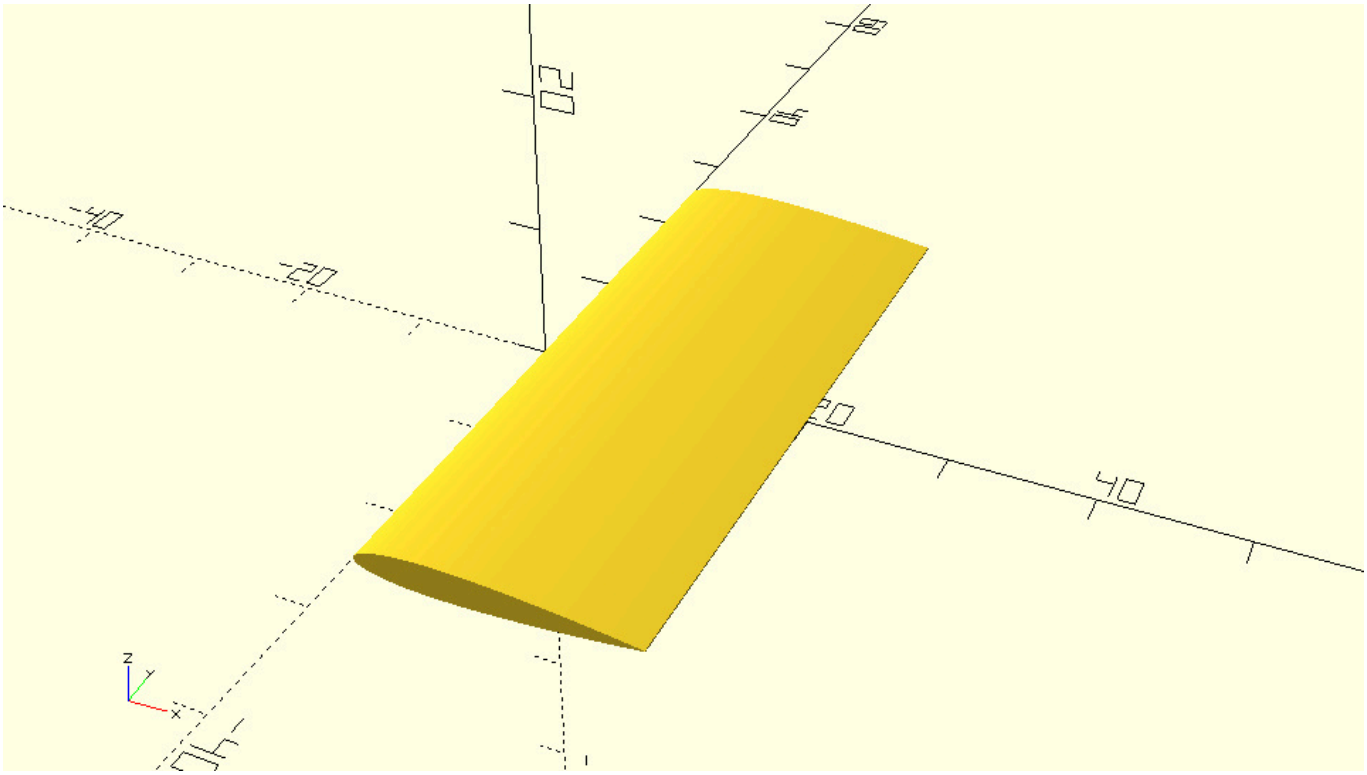
Code[Collapse]

naca_airfoil_module.scad

```
module naca_airfoil(chord,t,n) {
    points = naca_coordinates(t,n);
    scale([chord,chord,1])
        polygon(points);
}
```

Exercise

All you have to do now to create a wing out of the airfoil is to apply a linear_extrude command on the 2D airfoil profile. Create a module named naca_wing that does that. The naca_wing module should have two additional input parameters compared to the naca_airfoil module, span and center. The span parameter should correspond to the height of the extrusion while the center parameter should dictate whether the extrusion is executed along only the positive direction of the Z axis or along both directions. The span parameter shouldn't have a default value while the default value of the center parameter should be false. Can you use the naca_wing module to create the following wing? The following wing has a span of 50 units, while the airfoil of the wing has a chord of 20 units, a maximum thickness of 0.12 and 500 points on each half. You will have to additionally use a rotation transformation to place the wing as in the following image.



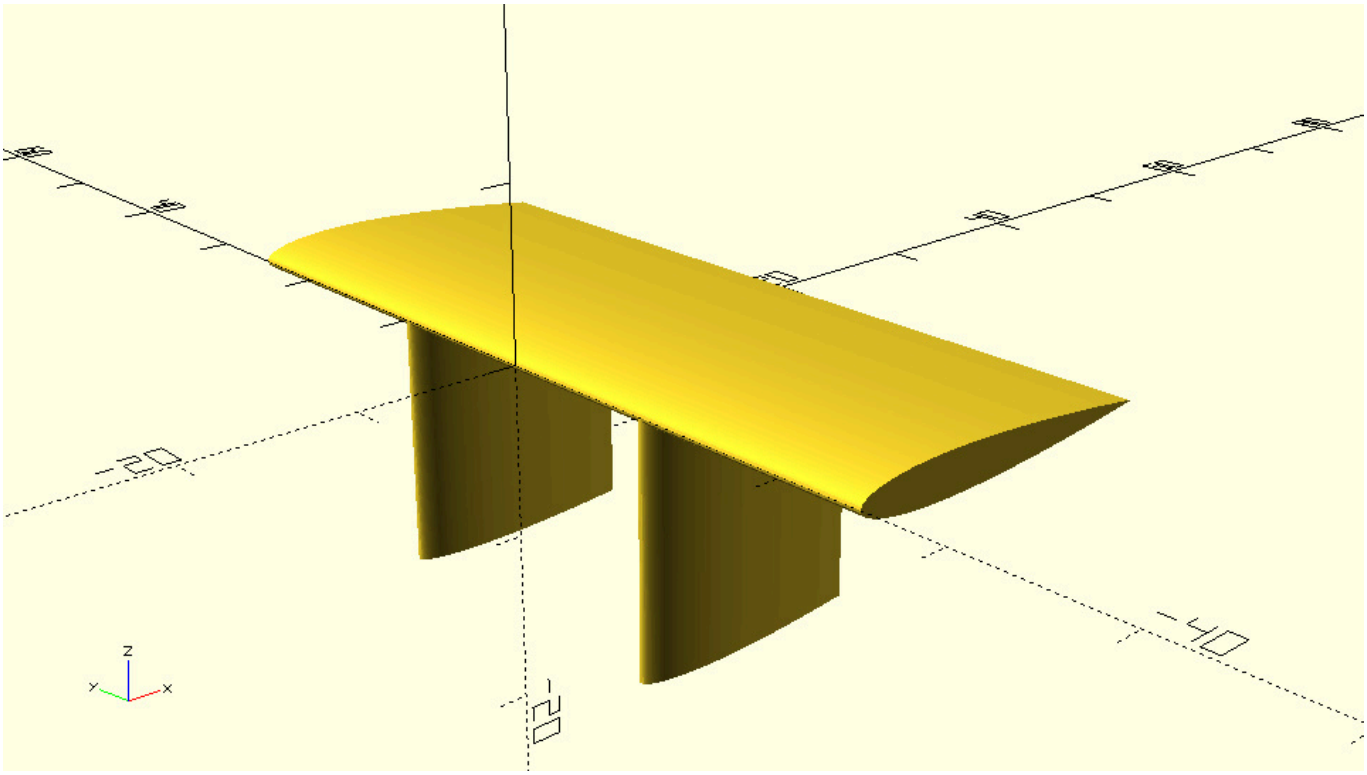
Code[Collapse]

spoiler_wing.scad

```
…
module naca_wing(span,chord,t,n,center=false) {
    linear_extrude(height=span,center=center) {
        naca_airfoil(chord,t,n);
    }
}
…
rotate([90,0,0])
    naca_wing(span=50,chord=20,t=0.12,n=500,center=true);
…
```

Exercise

Use the naca_wing module to add two smaller vertical wings on the previous example in order to create the spoiler of the car. The smaller wings should have a span of 15 units as well as a chord of 15 units.



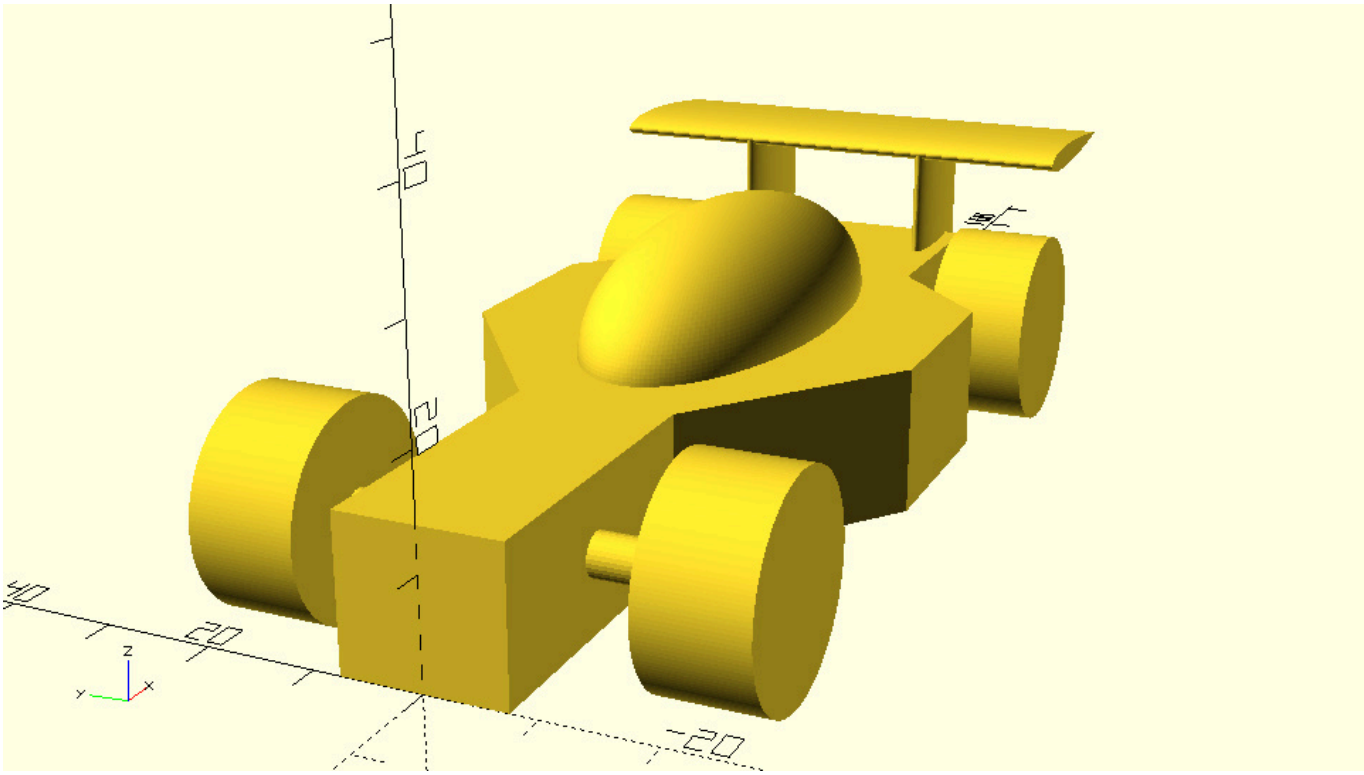
Code[Collapse]

spoiler.scad

```
…
rotate([90,0,0])
    naca_wing(span=50,chord=20,t=0.12,n=500,center=true);
translate([0,10,-15])
    naca_wing(span=15,chord=15,t=0.12,n=500);
translate([0,-10,-15])
    naca_wing(span=15,chord=15,t=0.12,n=500);
…
```

Exercise

Add the above spoiler on the racing car design to complete it.


40
20
0
-20
Z
Y
X

Code[Collapse]

racing_car_with_spoiler.scad

```
use <vehicle_parts.scad>

$fa = 1;
$fs = 0.4;

// model parameters
d1=30;
d2=20;
d3=20;
d4=10;
d5=20;

w1=15;
w2=45;
w3=25;

h=14;
track=40;

// distances to lengths
l1 = d1;
l2 = d1 + d2;
l3 = d1 + d2 + d3;
l4 = d1 + d2 + d3 + d4;
l5 = d1 + d2 + d3 + d4 + d5;

// right side points
p0 = [0, w1/2];
p1 = [l1, w1/2];
p2 = [l2, w2/2];
p3 = [l3, w2/2];
p4 = [l4, w3/2];
p5 = [l5, w3/2];

// left side points
p6 = [l5, -w3/2];
p7 = [l4, -w3/2];
p8 = [l3, -w2/2];
p9 = [l2, -w2/2];
p10 = [l1, -w1/2];
p11 = [0, -w1/2];

// all points
points = [p0, p1, p2, p3, p4, p5, p6, p7, p8, p9, p10, p11];

// extruded body profile
linear_extrude(height=h)
    polygon(points);

// canopy
translate([d1+d2+d3/2,0,h])
    resize([d2+d3+d4,w2/2,w2/2])
    sphere(d=w2/2);

// axles
l_front_axle = d1/2;
l_rear_axle = d1 + d2 + d3 + d4 + d5/2;
half_track = track/2;

translate([l_front_axle,0,h/2])
    axle(track=track);
translate([l_rear_axle,0,h/2])
    axle(track=track);

// wheels
translate([l_front_axle,half_track,h/2])
    simple_wheel(wheel_width=10);
translate([l_front_axle,-half_track,h/2])
    simple_wheel(wheel_width=10);

translate([l_rear_axle,half_track,h/2])
    simple_wheel(wheel_width=10);
translate([l_rear_axle,-half_track,h/2])
    simple_wheel(wheel_width=10);
```

```
// spoiler
use <naca.scad>

module car_spoiler() {
    rotate([90,0,0])
        naca_wing(span=50,chord=20,t=0.12,n=500,center=true);
    translate([0,10,-15])
        naca_wing(span=15,chord=15,t=0.12,n=500);
    translate([0,-10,-15])
        naca_wing(span=15,chord=15,t=0.12,n=500);
}

translate([l4+d5/2,0,25])
    car_spoiler();
```
