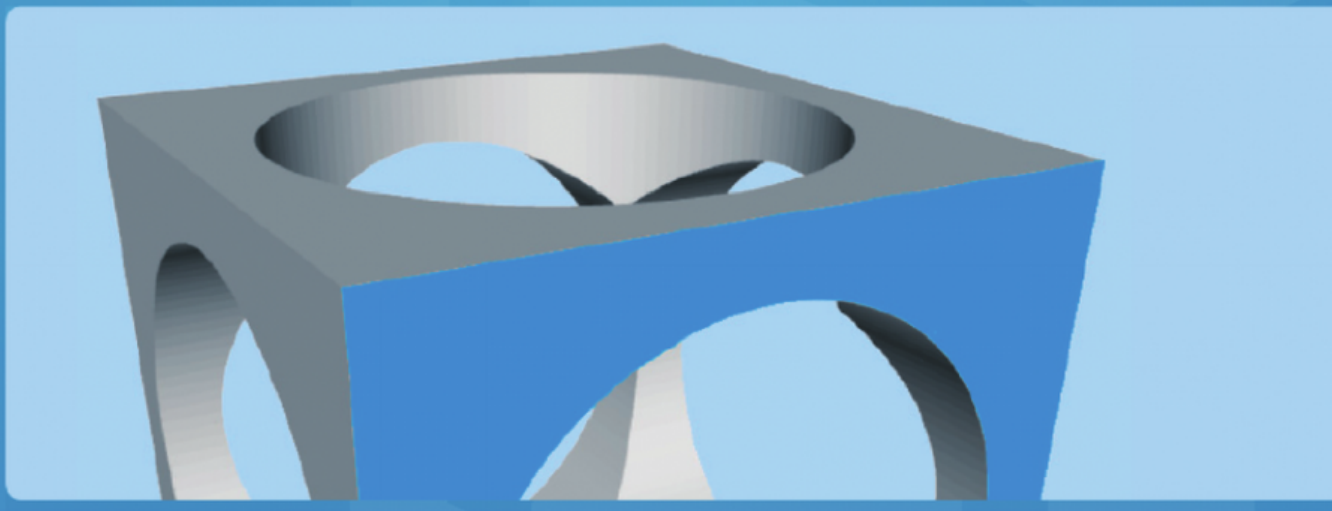
nclab

Learning for Industry 4.0

Introduction to 3D Modeling

A Project-Based Approach



Introduction to 3D Modeling

Introduction to 3D Modeling

Pavel Solin, Alberto Paoluzzi

Revision October 12, 2018

About the Authors

Dr. Pavel Solin is Professor of Applied and Computational Mathematics at the University of Nevada, Reno. He is an expert in scientific computing and the author of six monographs and many research articles in international journals.

Dr. Alberto Paoluzzi is Professor of Computer Graphics and CAD Design at the University of Rome in Italy, leader of the PLaSM project, and author of the famous monograph *Geometric Programming for Computer Aided Design, Wiley, 2003*.

Acknowledgment

We would like to thank many teachers for class-testing the course, and for providing useful feedback that helps us improve the textbook, the self-paced course, and the PLaSM language itself.

Preface

This course introduces the reader to 3D visualization, RGB colors, 2D and 3D shapes, geometrical transformations, and Boolean operations with geometrical objects. Its second and more advanced part introduces reference domains, reference maps, and parametric curves and surfaces.

The course is based on PLaSM (Programming Language of Solid Modeling) – a simple and elegant language based on Python where all objects, transformations, and operations are expressed simple commands. While progressing through the course, the reader also learns how to utilize more advanced elements of computer programming to simplify and automate the creation of 3D designs. The combination of geometry and programming is extremely powerful and rewarding. The PLaSM language is very intuitive and there is no need for prior knowledge of computer programming.

Good luck!

Pavel and Alberto

Table of Contents

1 Getting Started

Objectives:

- Learn basic facts about Solid Modeling and PLaSM.
- Learn to work with the PLaSM module.
- Learn about RGB colors.

1.1 Solid Modeling and PLaSM

The word "solid" in this context means "an object", as in "square is a two-dimensional solid", "cube is a three-dimensional solid". Solid Modeling, sometimes also called 3D Modeling or 3D Design, is a collection of rules and techniques for mathematical and computer modeling of solids. It is distinguished from related areas such as computer graphics by its emphasis on *physical fidelity*. In other words, accuracy of models that are used in 3D computer games is very different from the accuracy that is required in architecture, automotive industry, and other engineering areas. Solid Modeling is the basis of computer-aided design (CAD), engineering simulations, and other disciplines.

This 3D modeling course is based on PLaSM (Programming Language of Solid Modeling), a simple and elegant scripting language with Python syntax. In fact, PLaSM is a Python library – colors, shapes, geometrical transformations and everything else is defined via simple *commands*. Entire designs are simple *Python programs*. The PLaSM module is connected to a powerful computational geometry engine on a remote cloud server where these programs are evaluated, and resulting 3D geometries are sent back to your computer or tablet, and displayed in your web browser.

1.2 Launching PLaSM and running the demo script

The PLaSM module can be launched via the Creative Suite icon on Desktop:

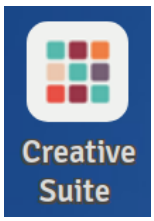


Fig. 1: Creative Suite icon on NCLab Desktop.

Alternatively, one can use the Start menu located in the bottom-left corner of the browser:

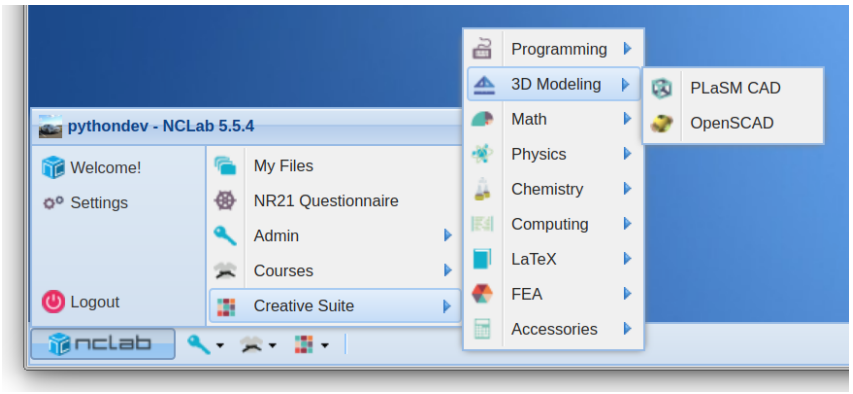


Fig. 2: Start menu.

The module opens with a demo script:

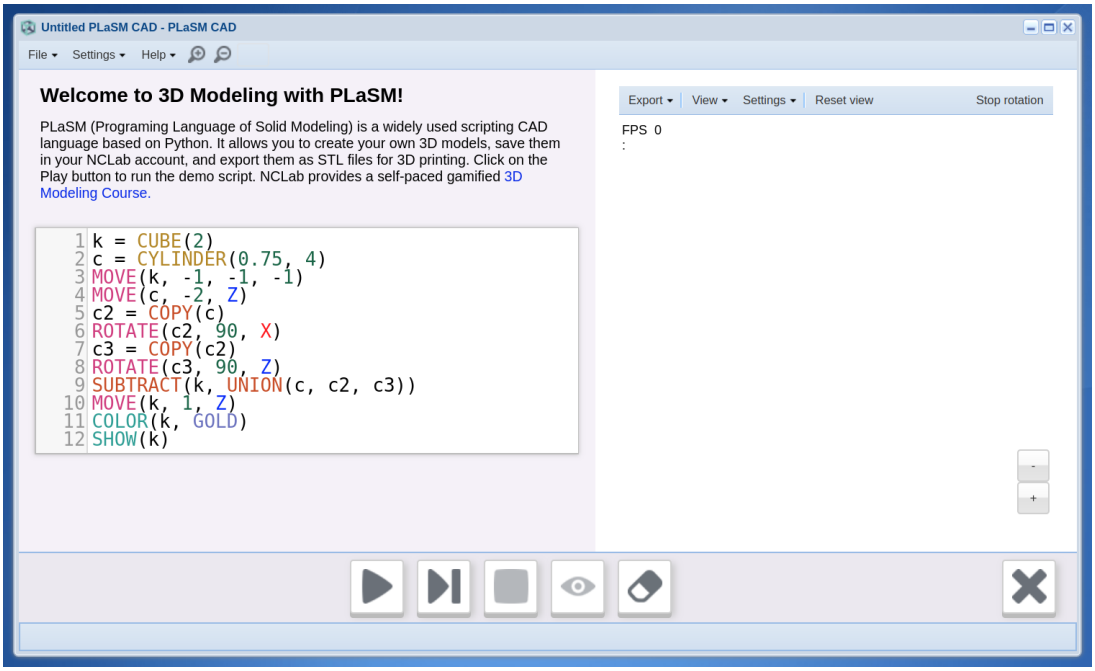


Fig. 3: PLaSM module with a demo script.

The purpose of the demo script is to show quickly how PLaSM works. In fact, it presents a lot of functionality:

- How to create a cube.
- How to create a cylinder.
- How to move objects.
- How to rotate objects.
- How to subtract objects from each other.
- How to display objects.

Run the demo script by pressing the Play button, and then allow few seconds for processing on the cloud and data transfer back to your computer or tablet. The resulting geometry is shown below:

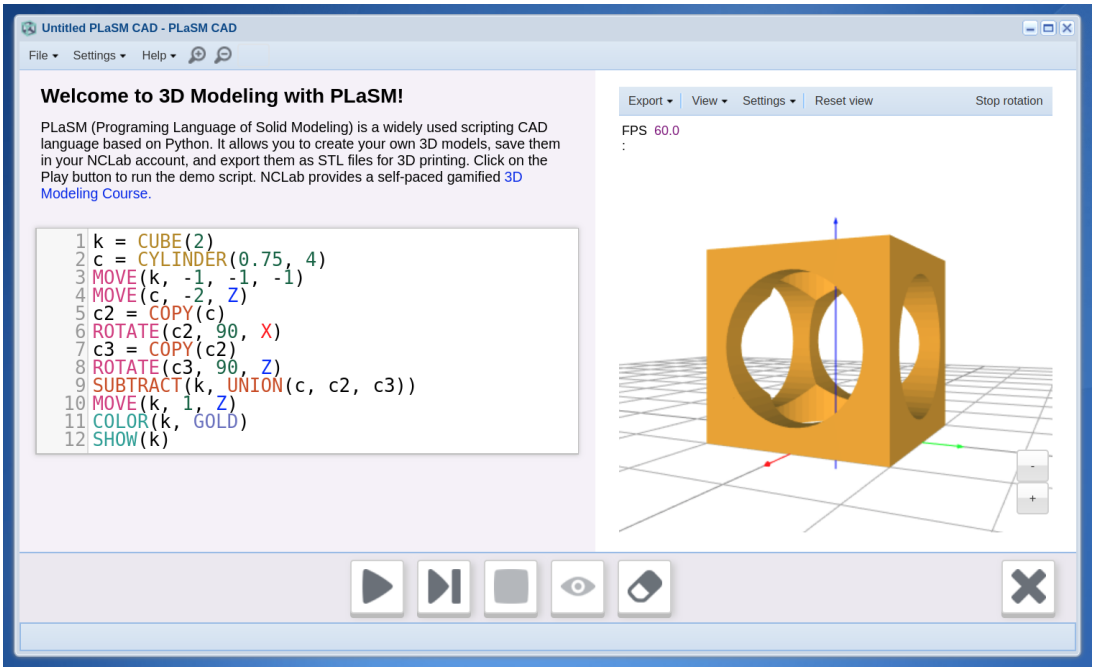


Fig. 4: Geometry created by the demo script.

Users who are familiar with Solid Modeling or have worked with another CAD system before, will probably have PLaSM figured out by now. For all others – please keep in mind that the demo script is a sneak-peek only. It's not meant to teach you 3D modeling – that's the purpose of this textbook.

1.3 3D printing

3D printing is the feature of our time, and NCLab fully supports it! NCLab users can send their designs to `support@nclab.com`. We will measure the volume and provide a quotation. The NCLab 3D printing service is not-for-profit and therefore less expensive than commercial 3D printing services.

For illustration, Fig. 5 shows a 3D print of the drilled cube created by the demo script.

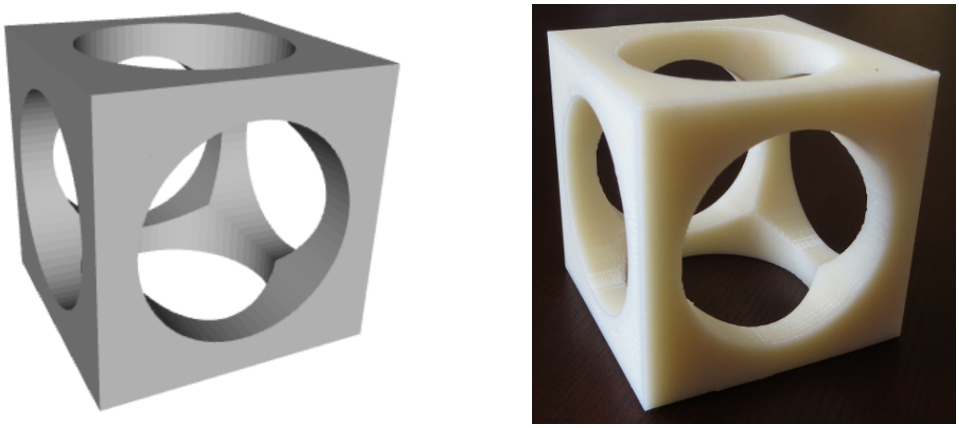


Fig. 5: 3D print of the drilled cube.

Fig. 6 shows a 3D print of a Roman temple model that we will build in Subsection 6.2.

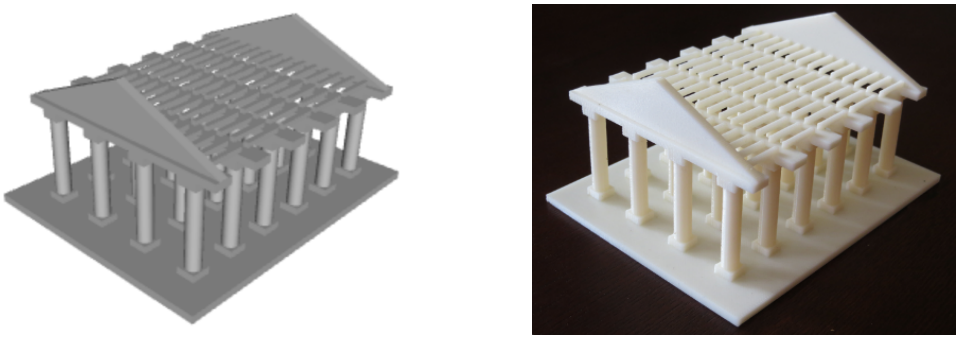


Fig. 6: 3D print of a Roman temple.

Fig. 7 shows a 3D print of a gear model that we will create in Subsection 6.4.

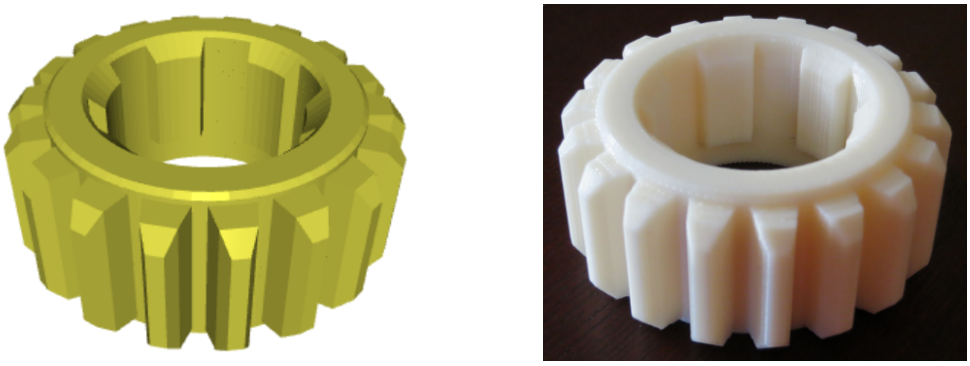


Fig. 7: 3D print of a gear model.

1.4 3D visualization

To better understand how 3D visualization works, imagine that your computer screen is a 2D plane that has a coordinate system (grid) attached to it: Horizontal axis X that lies in the plane of the screen and goes from left to right, vertical axis Y that also lies in the plane of the screen but goes from bottom to top, and a Z-axis that is perpendicular to the screen, as shown in Fig. 8.

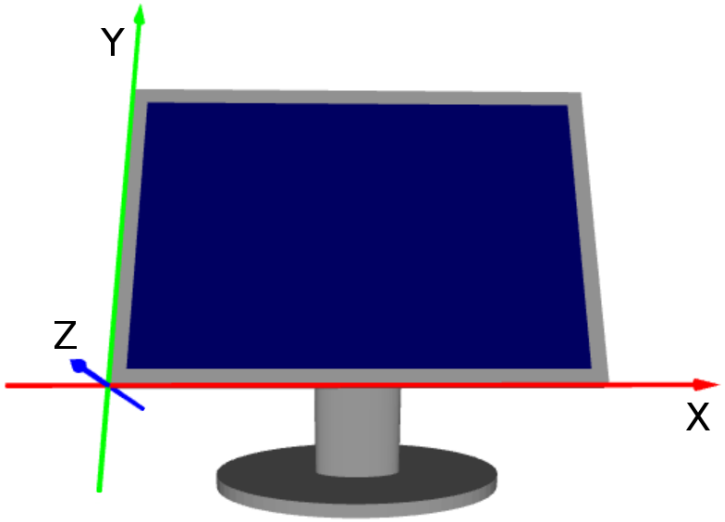


Fig. 8: Virtual grid associated with your computer screen.

The mouse can be used to rotate the object about all three axes X, Y and Z, and also to move it in all these directions. So, although the computer screen is flat, the visual experience is 3D. Before we explain how it works, run the PLaSM demo script and stop the rotation using the button in the upper right corner of the output window.

1. Turning

Hover the mouse pointer over the 3D object and press the left button. While holding it down, move the mouse to the left and to the right on the desk. The object on the screen will rotate about the Y axis. Moving the mouse up and down on the desk while holding down the left button will result in rotation about the X axis.

2. Panning

The missing rotation about the Z axis cannot be made up by combining the X and Y rotations. Therefore we use *panning*: Hover the mouse pointer over the object and press the left button again. While holding it down, start describing with the mouse small circles on the desk in the counter clockwise direction. The object on the screen will start tilting to the right (Z rotation). Similarly, circular motion in the clockwise direction will tilt the object to the left. Combined with the X and Y rotations described in point 1, now we have a complete set of all 3D rotations.

3. Moving

Holding down the middle button (or the mouse wheel) instead of the left one, and moving the mouse left to right will move the object on the screen in the X direction. Moving the mouse up and down on the desk will move the object in the Y direction. Missing the Z direction? That's why we have zooming!

4. Zooming

Zooming is done via the mouse wheel or by holding down the right mouse button and moving the mouse up and down on the desk. The object on the screen moves in the Z direction. The latter option is smoother. Hence, with zooming we have a complete set of 3D motions on the computer screen.

The mouse controls only adjust the view of the model in the X, Y and Z coordinates associated with your screen. They do not make any changes to the model itself.

1.5 RGB colors

According to the *additive color model* which is based on the human perception of colors, every color can be obtained by mixing the shades of Red, Green and Blue (R, G and B). These are called *additive primary colors* or just *primary colors*. The resulting color depends on the proportions of the primary colors in the mix. It is customary to define these proportions by an integer between 0 and 255 for each primary color. So, the resulting color is a triplet of real numbers between 0 and 255:

Color	R	G	B
Red	255	0	0
Green	0	255	0
Blue	0	0	255

These colors are shown in the following Fig. 9.

Fig. 9: Pure red [255, 0, 0], pure green [0, 255, 0], pure blue [0, 0, 255].

When the proportions of all three primary colors are the same, the result is a shade of grey. With [0, 0, 0] one obtains black, with [255, 255, 255] white:

Fig. 10: Black [0, 0, 0], dark grey [128, 128, 128], light grey [220, 220, 220].

Guessing RGB codes is not easy. If you need to find the numbers representing your favorite color, the best way is to google for "rgb color palette". You will find many pages that translate colors into RGB codes. Fig. 11 shows three "easy" colors cyan, pink and yellow along with their RGB codes.

Fig. 11: Cyan [0, 255, 255], pink [255, 0, 255], yellow color [255, 255, 0].

In Subsection 2.2 we will learn how colors are assigned to objects in PLaSM. In Subsection 3.1 we will see how various objects in one scene can be colored differently.

2 Library of Shapes

Objectives:

- Create a variety of 2D and 3D shapes.
- Assign colors to objects.
- Create convex hulls and extrude 2D objects to 3D.

This section serves mainly as a *database of shapes* and it is intended for reference rather than for systematic study. You may want to browse through it quickly to get an overview what shapes are available, but our goal is to move to design projects quickly. Note in Subsection 2.2 how colors are assigned to objects. Your first project is awaiting you in Section 3 and it will only require cubes and boxes.

2.1 Cube

The command `CUBE(a)` creates a cube of dimensions $a \times a \times a$. Every newly created object comes in a default steel color. To visualize objects, we use the `SHOW` command, that can be abbreviated by `V`. Type the two lines below into the PLaSM input cell and press the green arrow button:

```
c = CUBE(1)
SHOW(c)
```

After a moment, the following image should appear in your web browser:

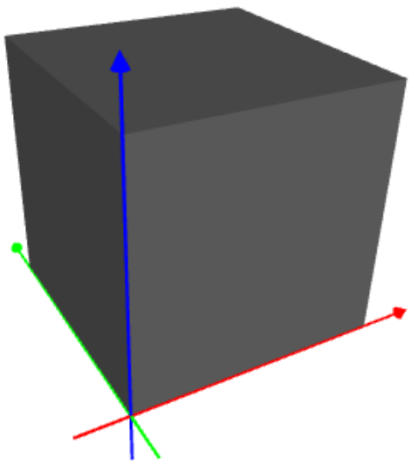


Fig. 12: Steel cube of dimensions $a \times a \times a$ with $a = 1$.

Note that the cube is located in the first quadrant, with its edges aligned with the coordinate axes x (red), y (green) and z (blue). One of its vertices lies at the origin $(0, 0, 0)$. Every cube created via the `CUBE` command will be positioned like this.

2.2 Coloring objects

The command `COLOR`, possibly abbreviated as `C`, assigns a given RGB color to a 2D or 3D object. Its use is best illustrated on the following example, where we change the color of the cube from steel to brass:

```
c = CUBE(1)
COLOR(c, BRASS)
SHOW(c)
```

Note that PLaSM keywords are written in *capital letters*. This is to avoid conflicts with user-defined variables (names of objects, custom colors, etc.). For newcomers to programming – in the above program, `c` is a user-defined variable (intentionally chosen to be lowercase), and `CUBE`, `COLOR`, `BRASS` and `SHOW` are PLaSM keywords. After executing the script, you will see the following:

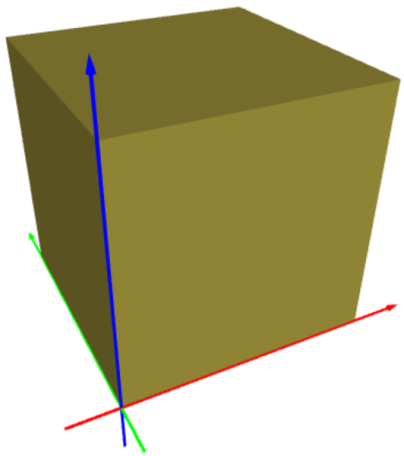


Fig. 13: Same cube as before, but now in brass color.

Note that the axes in the grid are color-coded using the word "RGB" for convenience: x = R, y = G, z = B.

The keyword `BRASS` is just a predefined triplet of numbers `[255, 250, 83]`. PLaSM offers the following predefined colors:

```
GREY = [128, 128, 128]
GREEN = [0, 255, 0]
BLACK = [0, 0, 0]
BLUE = [0, 0, 255]
BROWN = [139, 69, 19]
CYAN = [0, 255, 255]
MAGENTA = [255, 0, 255]
PINK = [255, 0, 255]
ORANGE = [255, 153, 0]
PURPLE = [128, 0, 128]
WHITE = [255, 255, 255]
RED = [255, 0, 0]
YELLOW = [255, 255, 0]
```

and metallic colors:

```
STEEL = [255, 255, 255]
BRASS = [181, 166, 66]
COPPER = [184, 115, 51]
BRONZE = [140, 120, 83]
SILVER = [230, 232, 250]
GOLD = [226, 178, 39]
```

2.3 Planar square

PLaSM makes it possible to work in a two-dimensional setting of the axes x and y (in addition to 3D). Here, the z axis is not present. The first 2D command that we will explore is `SQUARE(a)` which renders a square of edge length a. Its usage is illustrated by the following script:

```
s = SQUARE(3)
COLOR(s, BRASS)
SHOW(s)
```

The square is shown in Fig. 14. Again note where it is positioned - all newly created squares are positioned like this.

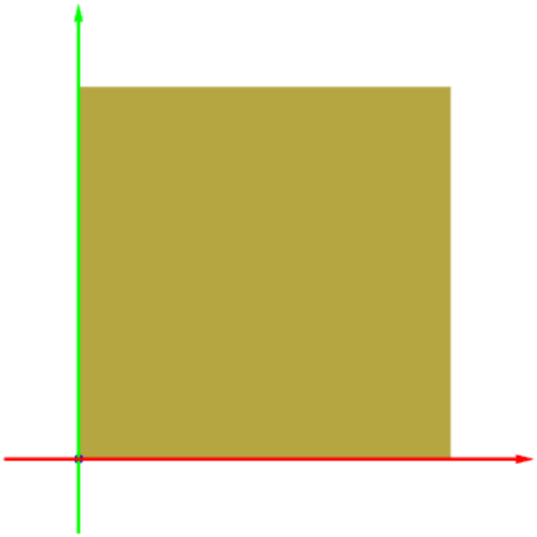


Fig. 14: Brass square of dimensions $a \times a$ with $a = 3$.

2.4 Square as thin 3D solid

The planar square introduced in the previous paragraph is a purely 2D object that cannot be combined with 3D objects. However, sometimes we need to do this – such as when constructing conic or cylindric sections. For this purpose, PLaSM has a command `SQUARE3D(a)` that creates a square of edge length a in the 3D space. Otherwise the square is located in the same way as the one created via the `SQUARE(a)` command. Sample script creating a square in the 3D space is

```
s = SQUARE3D(3)
COLOR(s, brass)
SHOW(s)
```

In reality, this object is a thin 3D solid (cube whose third dimension was scaled down to 0.001). Later we will learn how to rotate, move, and scale such objects, and how to intersect them with other 3D objects. The output of the above acript is shown in Fig. 15.

Fig. 15: Brass square of dimensions $a \times a$ with $a = 3$, as a thin 3D solid.

2.5 Box

The command `BOX(a, b, c)` creates a box of dimensions `a`, `b` and `c`. Let us change the code in the input cell to

```
b = BOX(3.0, 2.0, 1.0)
COLOR(b, COPPER)
SHOW(b)
```

and press the green arrow button. The result is displayed in Fig. 16.

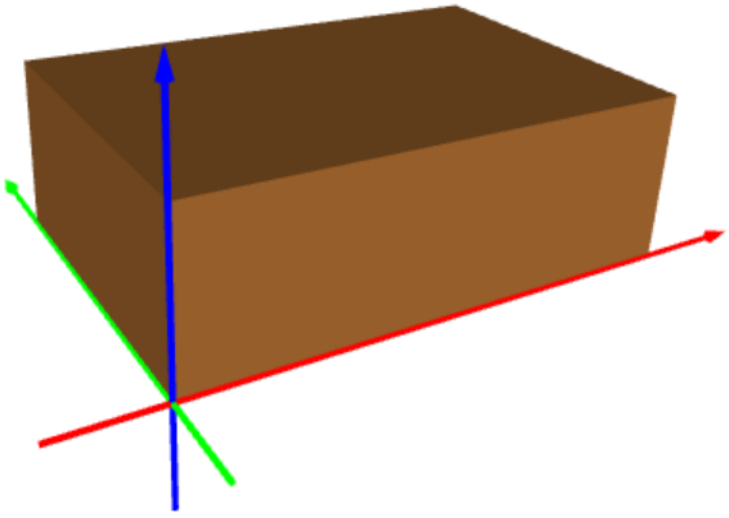


Fig. 16: Copper box of dimensions 3, 2 and 1.

2.6 Planar rectangle

Planar rectangle of dimensions a, b can be created with the command `RECTANGLE(a, b)`. For illustration, the output of a sample script

```
r = RECTANGLE(3, 1)
COLOR(r, COPPER)
SHOW(r)
```

is shown in Fig. 17. Note that the rectangle is located in the first quadrant with its edges aligned with coordinate axes, and that its bottom-left vertex lies at the origin $(0, 0)$.

Fig. 17: Copper rectangle of dimensions 3 and 1.

2.7 Rectangle as thin 3D solid

To operate with rectangles in the 3D space, we have the command `RECTANGLE3D(a, b)` which works analogously to the command `SQUARE3D(a)`. Its usage can be illustrated using the script

```
r = RECTANGLE3D(3, 1)
COLOR(r, COPPER)
SHOW(r)
```

that creates the same rectangle as before but renders it as thin 3D solid. The rectangle is again located in the first quadrant, with one vertex at the origin $(0, 0, 0)$, and its edges are aligned with the coordinate axes x and y. In reality, this is a box whose third dimension is 0.001. The output is shown in Fig. 18.

Fig. 18: Copper rectangle of dimensions $a \times b$ with $a = 3$, $b = 1$, as a thin 3D solid.

2.8 Tetrahedron

Tetrahedron in PLaSM is defined using four 3D points that do not lie in the same plane. Every 3D point in PLaSM is a triplet of real numbers enclosed in square brackets, such as $[0, 0, 0]$ (the origin). To keep our script readable, it is a good idea to introduce new variables for points. Such as, the points $[-2, 0, 0]$, $[1, 0, 0]$, $[0, 4, 1]$, $[0, 1, 2]$ can be called a, b, c and d. Then the command `TETRAHEDRON(a, b, c, d)` creates a tetrahedron with the vertices a, b, c and d. The output of the code

```
a = [-2, 0, 0]
b = [1, 0, 0]
c = [0, 4, 1]
d = [0, 1, 2]
tet = TETRAHEDRON(a, b, c, d)
COLOR(tet, BRONZE)
SHOW(tet)
```

is shown in Fig. 19.

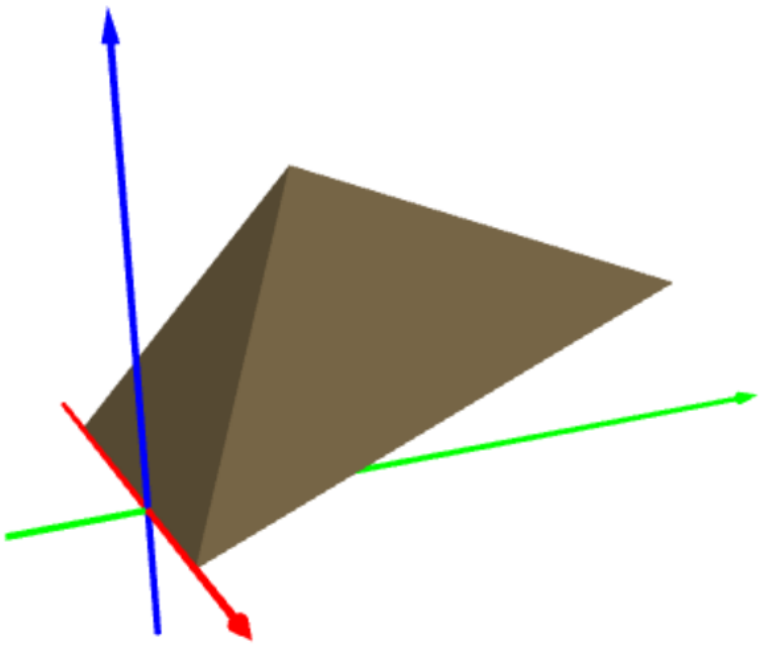


Fig. 19: Bronze tetrahedron with the vertices $[-2, 0, 0]$, $[1, 0, 0]$, $[0, 4, 1]$ and $[0, 1, 2]$.

2.9 Planar triangle

Planar triangles are created using the command `TRIANGLE(a, b, c)` where a, b, c are 2D points (pairs of real numbers enclosed in square brackets). For example, the triangle with vertices $[-1, 0]$, $[1, 0]$, $[0, 2]$ is created using the following code:

```
a = [-1, 0]
b = [1, 0]
c = [0, 2]
tria = TRIANGLE(a, b, c)
COLOR(tria, BRONZE)
SHOW(tria)
```

The output is shown in Fig. 20.

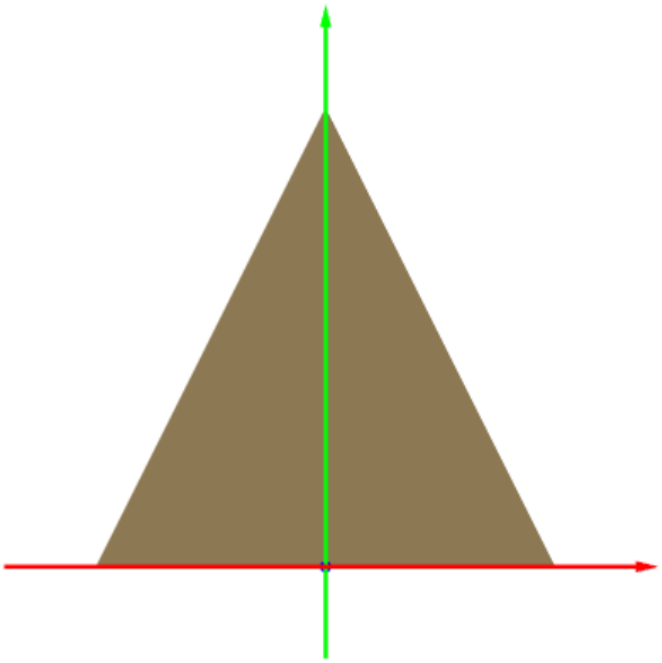


Fig. 20: Bronze triangle with the vertices $[-1, 0]$, $[1, 0]$ and $[0, 2]$.

2.10 Triangle as thin 3D solid

Knowing how squares and rectangles work in the 3D setting, you will not be surprized by learning that the command `TRIANGLE3D(a, b, c)` creates a 3D triangle. The points a, b, c are 2D points though, and the triangle will lie in the xy-plane. Let us create the same triangle as in the previous paragraph:

```
a = [-1, 0]
b = [1, 0]
c = [0, 2]
tria = TRIANGLE3D(a, b, c)
COLOR(tria, BRONZE)
SHOW(tria)
```

The output is shown in Fig. 21.

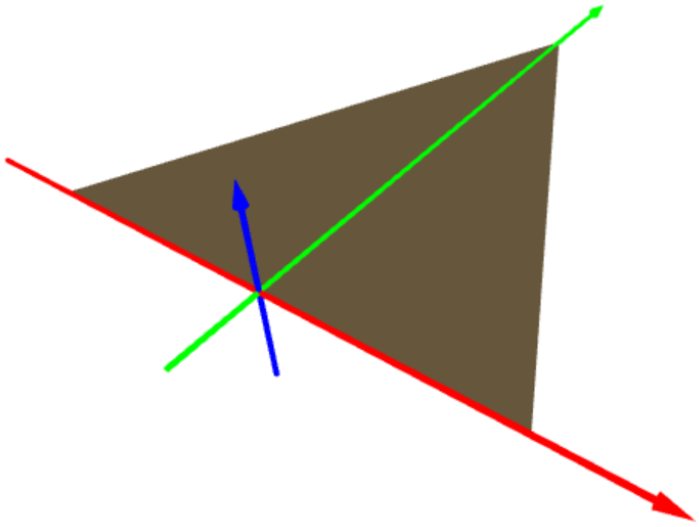


Fig. 21: Bronze triangle with the vertices $[-1, 0]$, $[1, 0]$ and $[0, 2]$, as a thin 3D solid.

2.11 Sphere

Sphere with radius r and center at $(0, 0, 0)$ is created using the command `SPHERE(r)`:

```
s = SPHERE(3.0)
COLOR(s, GOLD)
SHOW(s)
```

The output is shown in Fig. 22.

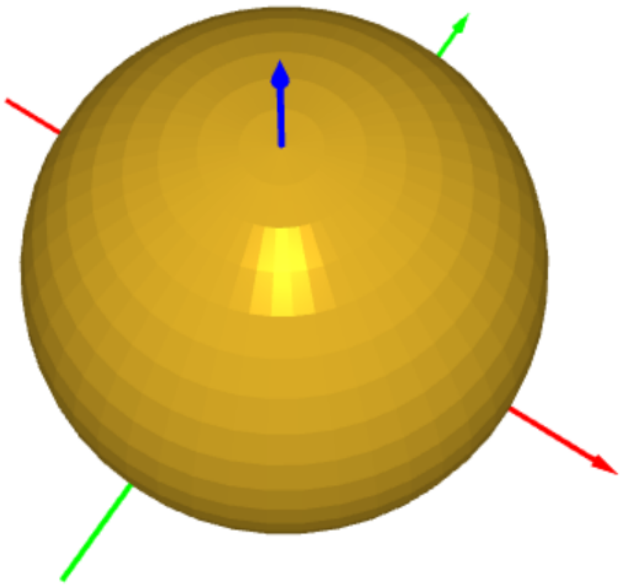


Fig. 22: Gold sphere with radius $r = 3$ and center at $(0, 0, 0)$.

Notice that the surface is approximated using small flat sections. We will discuss this in more detail in Subsection 2.14 where we will also show how the spherical surface can be made smoother or coarser.

2.12 Planar circle

Planar circle with radius r and center at $(0, 0)$ can be defined using the command `CIRCLE(r)`:

```
c = CIRCLE(3.0)
COLOR(c, GOLD)
SHOW(c)
```

The output is shown in Fig. 23.

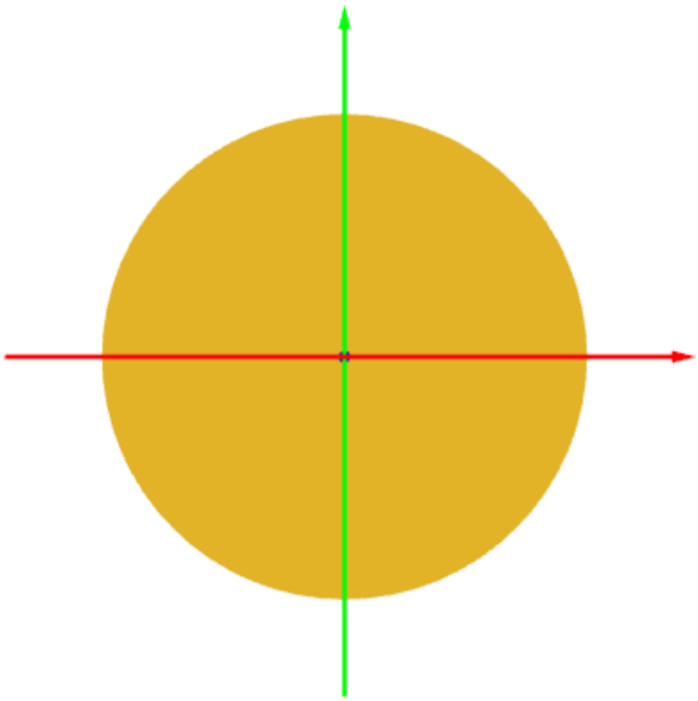


Fig. 23: Gold planar circle with radius $r = 3$ and center at $(0, 0)$.

In fact this is a polygon with many edges. In Subsection 2.14 we will show how to reduce the number of edges in order to obtain various equilateral polygons.

2.13 Circle as thin 3D solid

Finally, the command `CIRCLE3D(r)` creates a 3D circle with center at the origin $(0, 0)$ that lies in the xy-plane. This is the last in the series of 2D objects created as thin 3D solid, and it is constructed via the following script:

```
c = CIRCLE3D(3.0)
COLOR(c, GOLD)
SHOW(c)
```

The output is shown in Fig. 24.

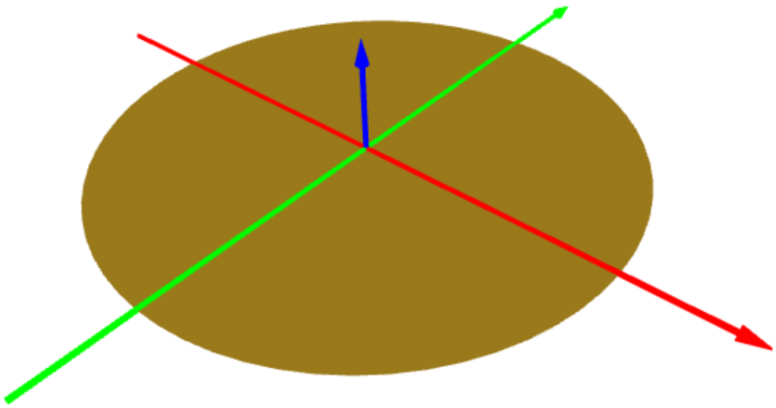


Fig. 24: Gold circle with radius $r = 3$ and center at $(0, 0)$, as a thin 3D solid.

As all the square, rectangle, and triangle in the 3D space also the circle is in reality a cylinder of height 0.001. The knowledge of this technical detail will be useful for performing operations such as intersection, union, difference and xor of these 2D objects.

2.14 Approximation of curved surfaces

In 3D computer graphics, curved surfaces are always approximated using small linear triangular segments. This is how computer hardware is built. Also PLaSM works in this way. Each curved object comes with a default division that the user can adjust if needed. For the `SPHERE` command, the default divisions are 64 in the angular direction and 32 in the z-direction. This yields a uniformly spaced grid. The division can be used as an optional argument. If there are two divisions, they are always enclosed in square brackets. Hence, the command

```
s = SPHERE(3.0)
```

is equivalent to

```
s = SPHERE(3.0, [32, 64])
```

It is easy to calculate that the default sphere is approximated using $2 \cdot 32 \cdot 64 = 4,096$ linear triangles. Therefore its rendering will take longer than a cone which has only around 100, and much longer than cube which only has 12. If you decide to make it look better by using twice finer division in both directions,

```
s = SPHERE(3.0, [64, 128])
```

then the number of linear triangles jumps to $2 \cdot 64 \cdot 128 = 16,384$. This is a lot, and you are likely to wait.

Making the number of linear triangles large means nicer image
but also more computing, more data transfer, and a longer wait.

The knowledge of the subdivisions can be used to create various interesting objects such as a diamond:

```
s = SPHERE(3.0, [2, 8])
COLOR(s, GOLD)
SHOW(s)
```

The output is shown in Fig. 25.

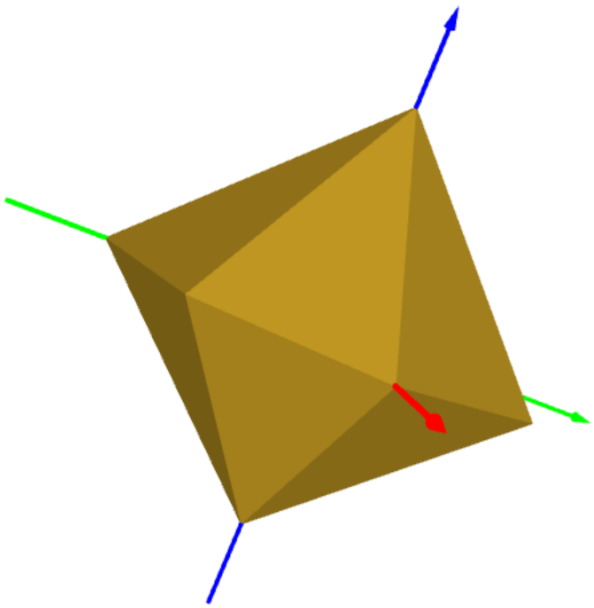


Fig. 25: Using `SPHERE` with subdivisions `[2, 8]` yields a diamond.

Another object that can be created using the `SPHERE` command with subdivisions `[64, 8]` is a baloon shown in Fig. 26.

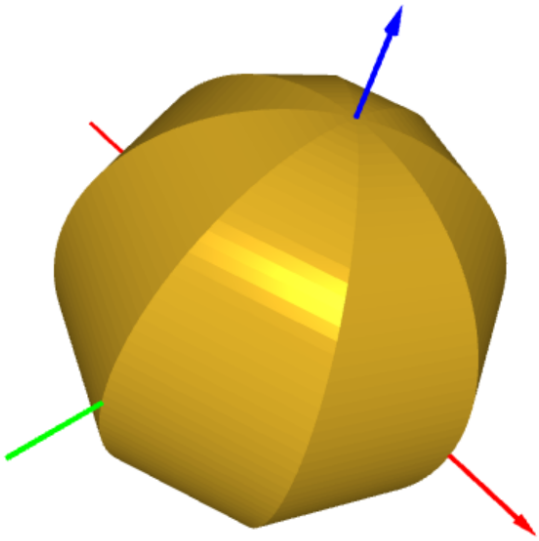


Fig. 26: Baloon (`SPHERE` with subdivisions `[64, 8]`).

in 2D, the `CIRCLE(r)` command is equivalent to `CIRCLE(r, 64)` and it approximates the circular boundary using an equilateral polygon with 64 edges. Various equilateral polygons can be created by using a number less than 64:

Fig. 27: `CIRCLE` with subdivisions `3`, `4`, `5` and `6`.

2.15 Prism

Given any 2D polygon `b` and a positive number `h`, the command `PRISM(b, h)` creates a prism with the base `b` and height `h`. For illustration, prisms of height `h = 3` corresponding to the polygons shown in Fig. 27 are depicted below:

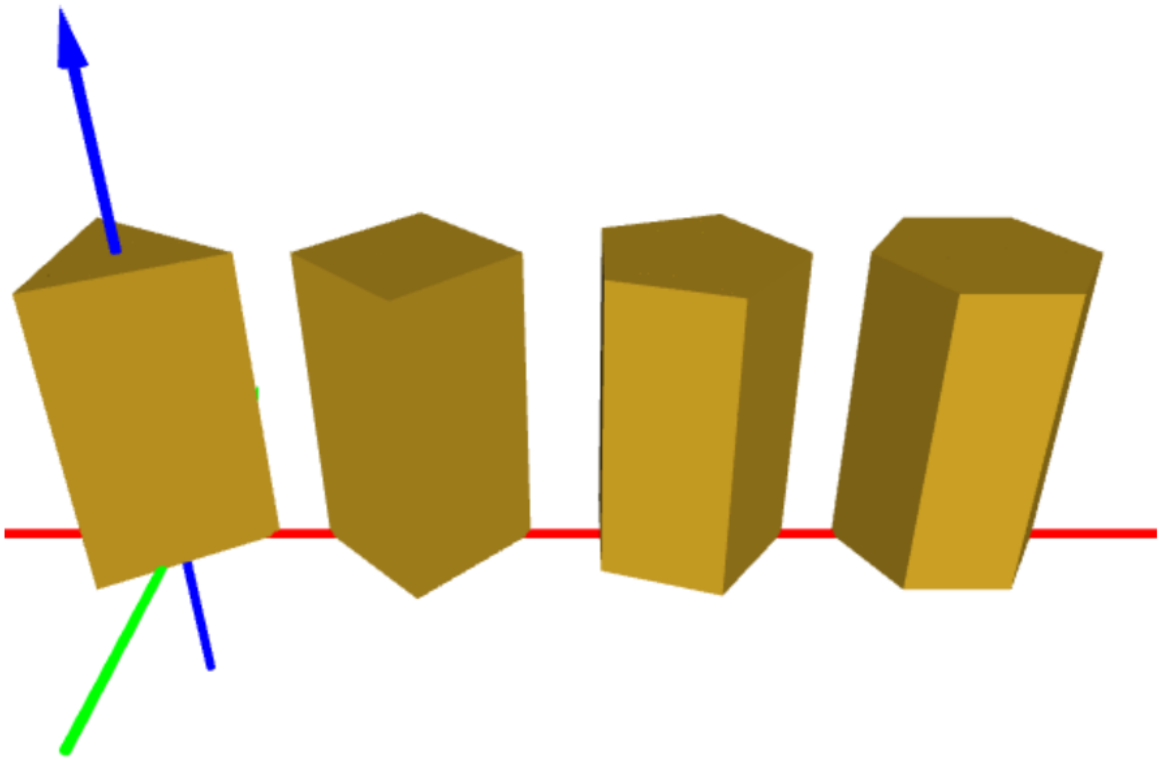


Fig. 28: Prisms with `h = 3` corresponding to polygons from Fig. 27.

The basis also can be a `SQUARE`, `RECTANGLE`, `TRIANGLE` and even a `CIRCLE`. In the last case, we obtain a cylinder. Since cylinders are used very often, PLaSM provides a separate command for them:.

2.16 Cylinder

Cylinder with radius `r` and height `h` can be defined using the command `CYLINDER(r, h)`, possibly abbreviated as `CYL(r, h)`. The cylinder's axis is in the z-direction, and the midpoint of its base circle lies at the origin $(0, 0, 0)$. The output of the sample code

```
r = 0.25
h = 1.0
c = CYL(r, h)
COLOR(c, GOLD)
SHOW(c)
```

is shown in Fig. 29.

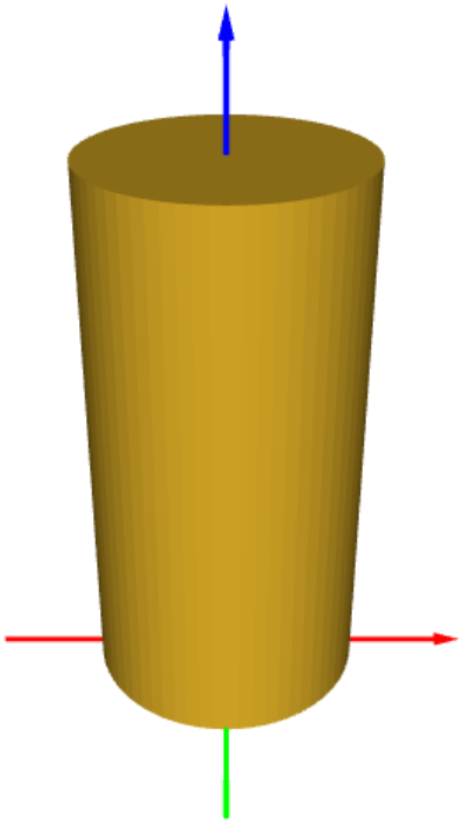


Fig. 29: Cylinder with radius `r = 0.25` and height `h = 1`.

By default, the curved surface is split into 64 vertical linear segments. To change the division to another integer number `m`, use

```
c = CYL(r, h, m)
```

This can be used to create prisms. By using `m = 3, 4, 5, 6` we obtain the prisms from Fig. 28.

2.17 Tube

Tube of inner radius `r1`, outer radius `r2` and height `h` is rendered using the command `TUBE(r1, r2, h)`. The tube is positioned in the global coordinate system same as

the cylinder – its axis coincides with the z-axis and the center of its base is at $(0, 0, 0)$. The output of the sample code

```
r1 = 0.2
r2 = 0.25
h = 1.0
t = TUBE(r1, r2, h)
COLOR(t, GOLD)
SHOW(t)
```

is shown in Fig. 30.

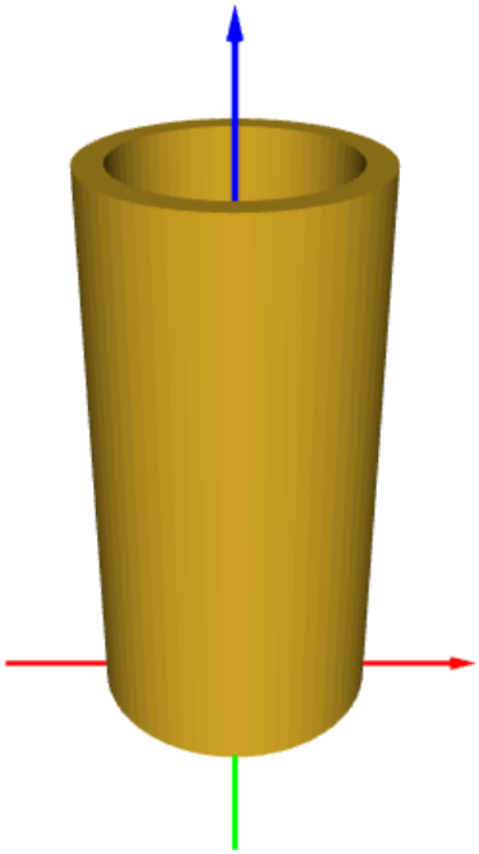


Fig. 30: Tube of inner radius `r1 = 0.2`, outer radius `r2 = 0.25` and height `h = 1`.

By default, the inner and outer curved surfaces are split each into 64 vertical linear segments. To change the division to another integer number `m`, use

```
t = TUBE(r1, r2, h, m)
```

This can be used to create tubes with various polygonal cross-sections.

2.18 Cone

Cone with radius `r` and height `h` is created using the command `CONE(r, h)`. The cone's axis coincides with the z-axis and the center of the base circle lies at the origin $(0, 0, 0)$. The output of the sample code

```
r = 5
h = 10
c = CONE(r, h)
COLOR(c, GOLD)
SHOW(c)
```

is shown in Fig. 31.

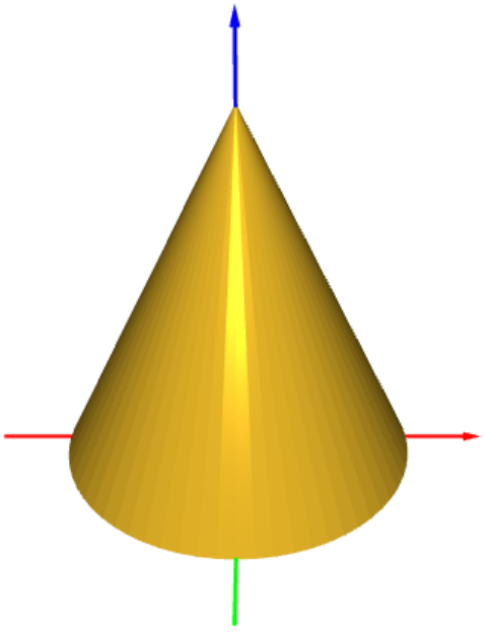


Fig. 31: Cone with radius `r = 5` and height `h = 10`.

By default, the curved surface is split into 64 linear triangles. To change the division to another integer number `m`, use

```
c = CONE(r, h, m)
```

The linear surface representation can be used to render pyramids, as shown in Fig. 32.

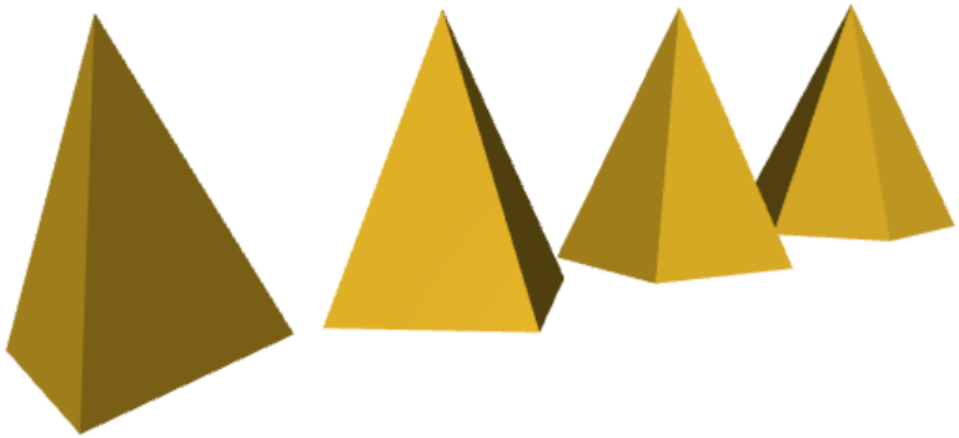


Fig. 32: Pyramids created using the `CONE` command with `r = 5, h = 10` and `m = 3, 4, 5, 6`.

2.19 Truncated cone

Truncated cone with bottom radius `r1`, top radius `r2` and height `h` is created using the command `TCONE(r1, r2, h)`. The cone's axis coincides with the z-axis and the center of the base circle lies at the origin $(0, 0, 0)$. The output of the sample code

```
r1 = 5
r2 = 2
h = 5
t = TCONE(r1, r2, h)
COLOR(t, GOLD)
SHOW(t)
```

The output is shown in Fig. 33.

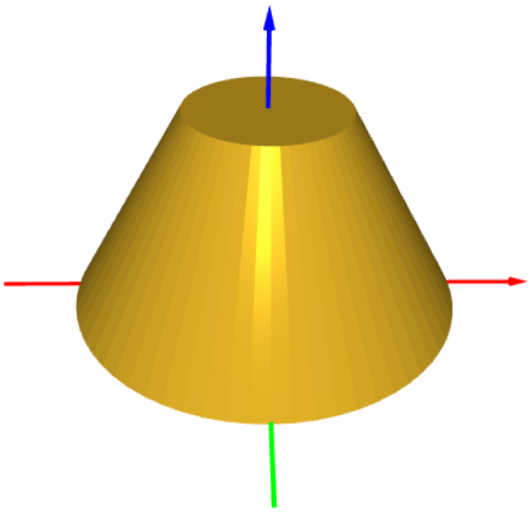


Fig. 33: Truncated cone with bottom radius `r1 = 5`, top radius `r2 = 2` and height `h = 5`.

By default, the curved surface is split into 64 trapezoidal linear segments. To change the division to another integer number `m`, use

```
t = TCONE(r1, r2, h, m)
```

The output is shown in Fig. 34.

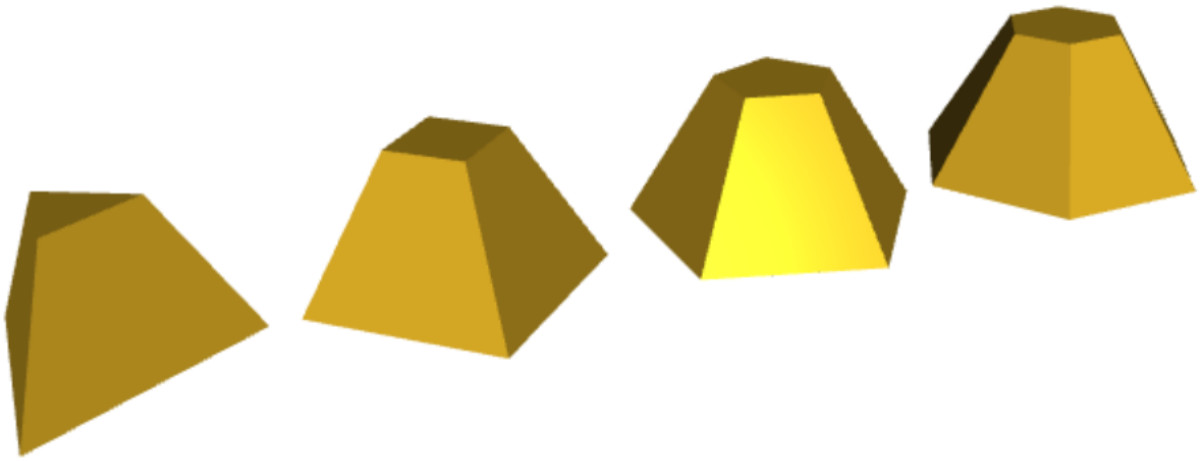


Fig. 34: Truncated pyramids with `r1 = 5, r2 = 2, h = 5` and `m = 3, 4, 5` and `6`.

2.20 Torus

Command `TORUS(r1, r2)` creates a torus (donut) with inner radius `r1` and outer radius `r2`. It is used as follows:

```
r1 = 3
r2 = 5
t = TORUS(r1, r2)
COLOR(t, GOLD)
SHOW(t)
```

The center of the torus is at the origin $(0, 0, 0)$ and its axis is the z-axis. This is illustrated in Fig. 35.

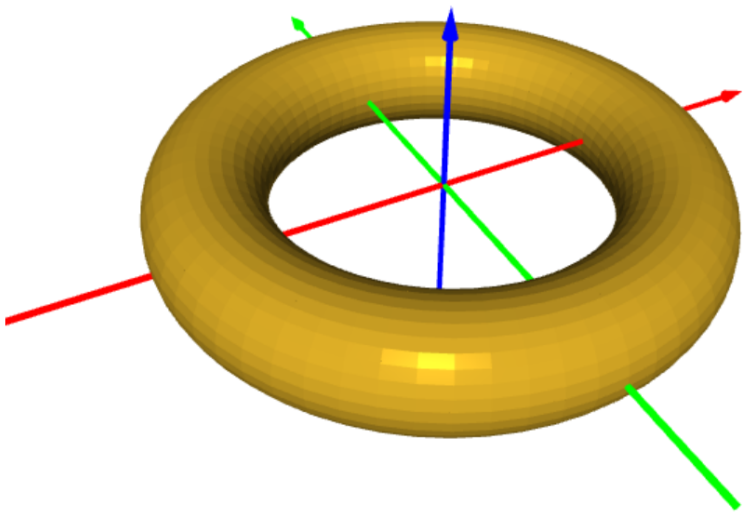


Fig. 35: Torus with inner radius `r1 = 3`, outer radius `r2 = 5` and center at the origin $(0, 0, 0)$.

By default, the major (large) circle is split into 64 linear segments and the minor (small) one into 32. Thus the default command `TORUS(r1, r2)` is equivalent to `TORUS(r1, r2, [64, 32])`. To change the divisions to other integer numbers `[m, n]`, use

```
t = TORUS(r1, r2, [m, n])
```

Also here, the piecewise-linear surface representation can be used to create interesting objects. Fig. 36 shows the output of the command `TORUS(3, 5, [64, 4])`.

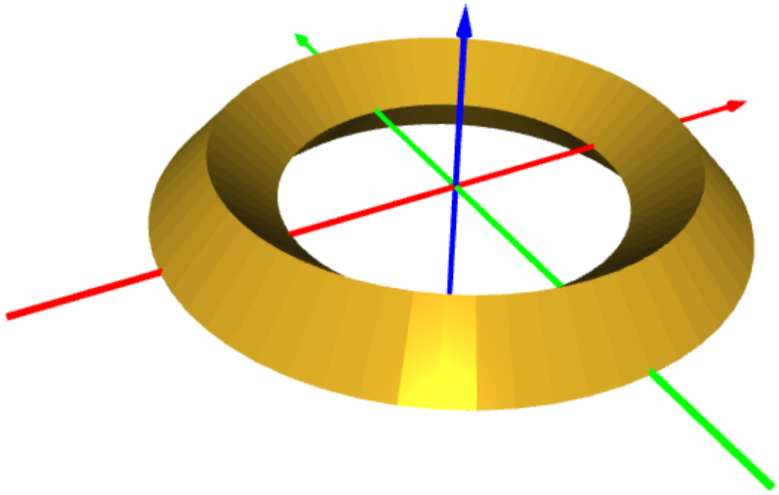


Fig. 36: Hollow disc.

Fig. 37 shows the output of the command `TORUS(3, 5, [4, 64])`.

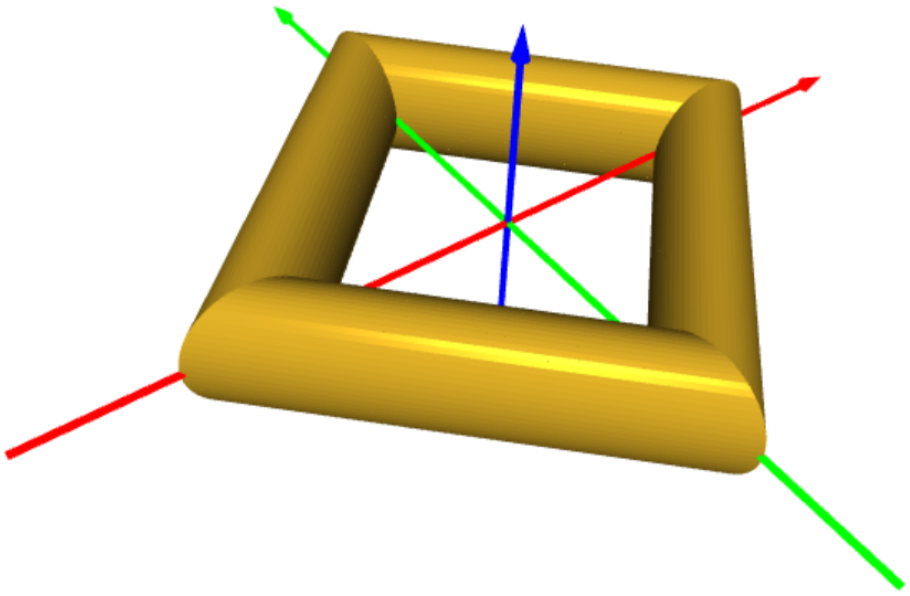


Fig. 37: Round pipe frame.

2.21 Convex hull

Constructing convex hulls of finite point sets in 2D and 3D is one of the most important algorithms in computational geometry. A 2D or 3D object is *convex* if for any two points located inside, the straight line connecting the points lies entirely inside. For illustration, Fig. 38 shows a convex object on the left and a non-convex one on the right.

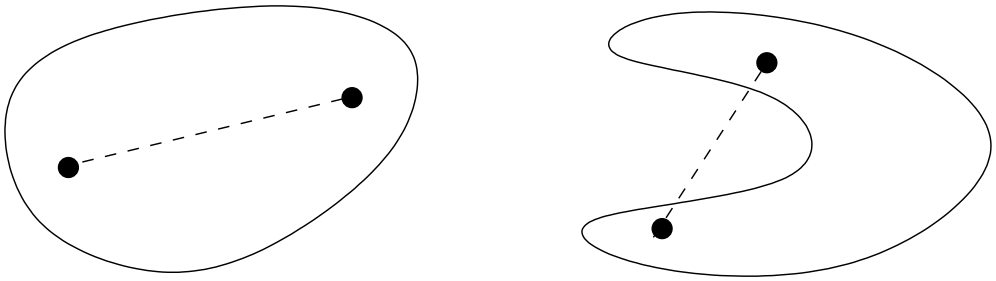


Fig. 38: Convex 2D object (left) and a non-convex one (right), along with sample straight lines connecting interior points.

By *convex hull* of a set of 2D or 3D points we mean the smallest convext set that contains all the points. To construct convex hull of a large number of points efficiently is not a trivial task at all. It can be proven mathematically that the complexity of finding a convex hull of n points is always at least $(n \log n)$ where n is the number of points

The complexity of more recent convex hull algorithms can be characterized in terms of both input size n and the output size h (the number of vertices of the hull). Such algorithms are called *output-sensitive algorithms*. They are often asymptotically more efficient than $(n \log n)$ algorithms in cases when h is much lower than n.

The earliest output-sensitive algorithm was introduced by Kirkpatrick and Seidel in 1986 (who called it "the ultimate convex hull algorithm"). A much simpler algorithm was developed by Chan in 1996, and is called Chan's algorithm. That's the one used in PLaSM.

The `CONVEXHULL` command (possibly abbreviated as `CHULL` or `CH`) takes a set of points and creates their convex hull:

```
o = CHULL([-1, 0], [1, 0], [0.5, 1], [-0.5, 1])
COLOR(o, GOLD)
SHOW(o)
```

This code renders a 2D trapezoid spanning the points $(-1, 0)$, $(1, 0)$, $(0.5, 1)$ and $(-0.5, 1)$ as shown in Fig. 39.

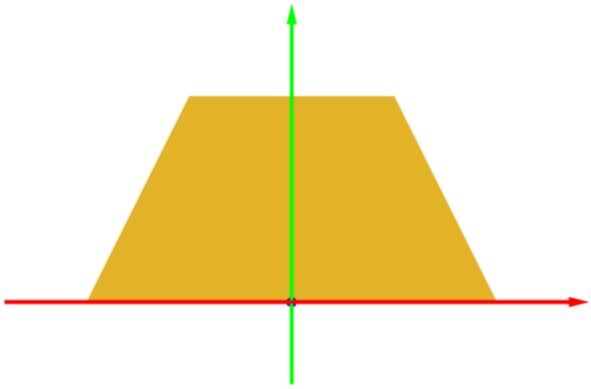


Fig. 39: Convex hull of four 2D points.

The code

```
o = CHULL([-1, -1, 0], [1, -1, 0], [1, 1, 0],
[-1, 1, 0], [0, 0, 1])
COLOR(o, GOLD)
SHOW(o)
```

creates a 3D object shown in Fig. 40.

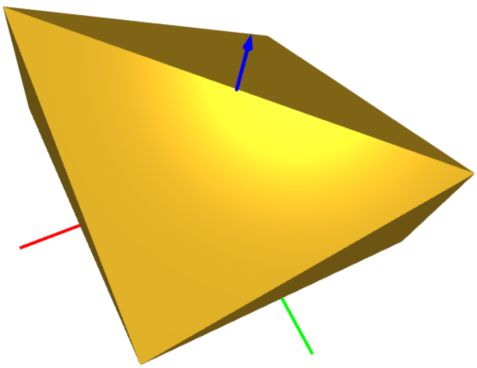


Fig. 40: 3D object constructed via convex hull of 6 points.

If you already have a list of points, defined for example as

```
L = [[-1, -1, 0], [1, -1, 0], [1, 1, 0], [-1, 1, 0], [0, 0, 1]]
```

then just insert it into the `CHULL` command:

```
out = CHULL(L)
```

2.22 Dodecahedron

A (regular) dodecahedron is an object with 12 identical pentagonal faces. It is one of the *Platonic solids* – convex symmetric 3D objects whose faces are regular polygons. There are only five of them – the (regular) tetrahedron, cube, octahedron, dodecahedron and icosahedron. For their extraordinary symmetry, these objects have been studied by geometers for thousands of years. In PLaSM, a dodecahedron is created simply as:

```
o = DODECAHEDRON
```

Output:

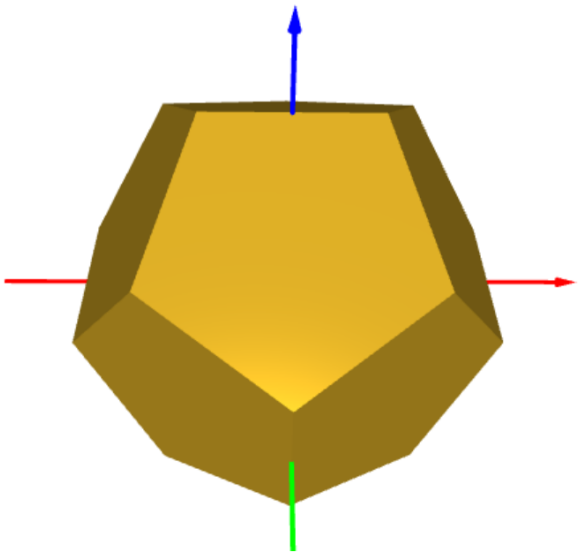


Fig. 41: Dodecahedron.

2.23 Icosahedron

A (regular) icosahedron is an object with 20 triangular faces that also belongs to the five Platonic solids. In PLaSM it is created via

```
o = ICOSAHEDRON
```

The object is shown in Fig. 42.

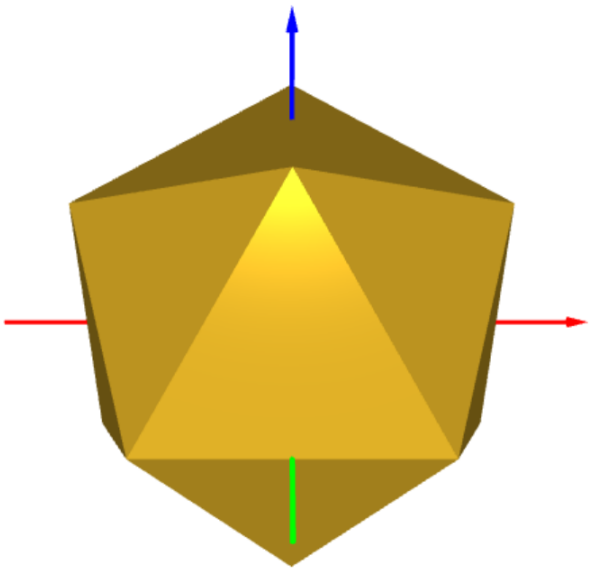


Fig. 42: Icosahedron.

2.24 Extrusion of 2D objects to 3D

An arbitrary 2D object that lies in the xy-plane can be extruded to 3D using the `PRISM` command that we already know from Subsection 2.15. For illustration, let us extrude a sample 2D polygon obtained as convex hull:

```
b = CHULL([0, 0], [2, 0], [1, 2], [-1, 2])
h = 0.5
o = PRISM(b, h)
```

The output is shown in Fig. 43.

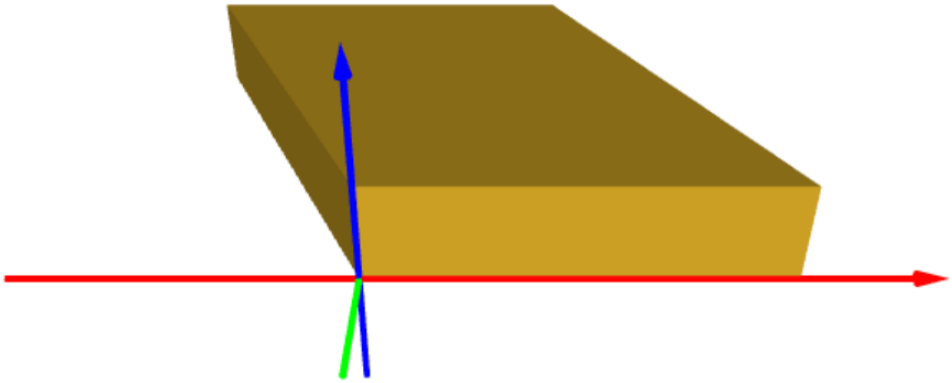


Fig. 43: Extrusion of a quadrilateral to 3D.

The command `PRISM(b, h)` uses a single interval in the z-direction. For future reference, let us also introduce the command `EXTRUDE(b, h, angle, n)` that with `angle = 0` yields the same extruded geometry as the `PRISM` command, but it subdivides the extrusion height `h` into `n` equally long subintervals. This will be practical for working with curved shapes.

2.25 Grid and Cartesian product

PLaSM also provides useful commands `GRID` and `PRODUCT` that make it easy to work with grids and Cartesian product geometries. The command `GRID` takes a list of positive and negative numbers where the positive ones stand for intervals (more precisely for their lengths) and negative for the lengths of spaces between the intervals. The intervals and spaces are placed on the right of the origin in the real axis. For example,

```
g1 = GRID(5)
```

creates a single interval $(0, 5)$. The command

```
g2 = GRID(1, -0.5, 1, -0.5, 1)
```

creates three intervals $(0, 1)$, $(1.5, 2.5)$ and $(3, 4)$, leaving empty spaces of length 0.5 between them. There is no limit on the number of intervals. Next, the command

```
G = PRODUCT(g1, g2)
```

creates a 2D object `G` in the xy-plane which is the *Cartesian product* of the two grids `g1` and `g2`. This object is shown in Fig. 44.

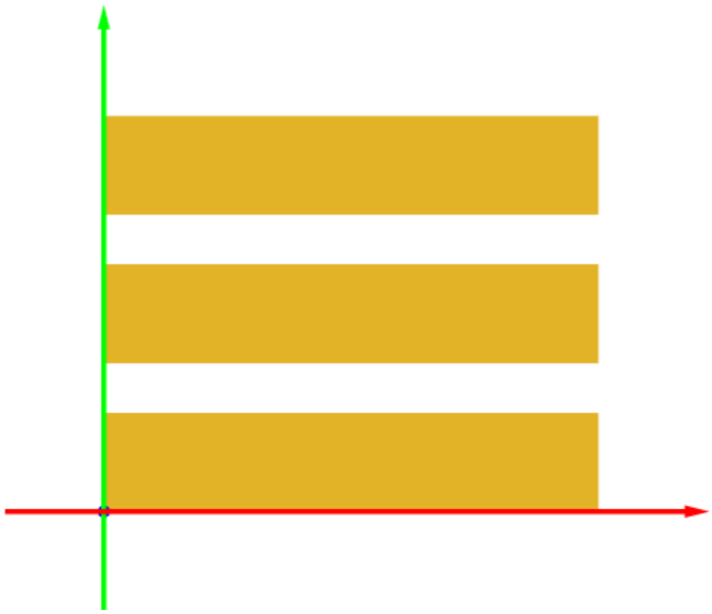


Fig. 44: Cartesian product of the grids `g1` and `g2`.

This 2D object can be multiplied with a third grid in the z-direction. Let's say that the third grid is

```
g3 = GRID(0.5, -1, 0.5, -1, 0.5)
```

Then the result of the Cartesian product

```
H = PRODUCT(G, g3)
```

is a 3D geometry shown in Fig. 45.

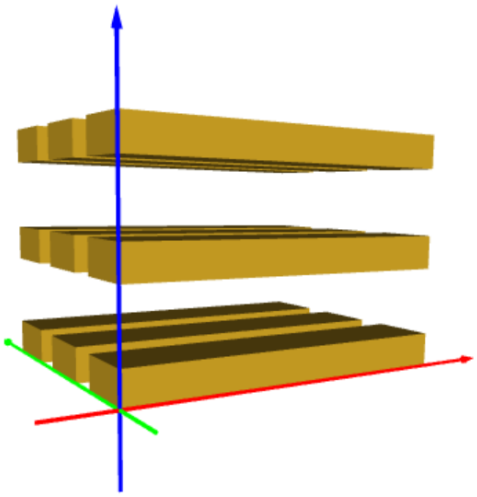


Fig. 45: Cartesian product of the grids `G` and `g3`.

Just to show one more application of `GRID` and `PRODUCT`, let's do:

```
g1 = GRID(3, -1, 3, -1, 3, -1, 3)
g2 = GRID(2, -1, 2, -1, 2)
g3 = GRID(1, -1, 1)
planar_grid = PRODUCT(g1, g2)
o = PRODUCT(planar_grid, g3)
```

The object is shown in Fig. 46.

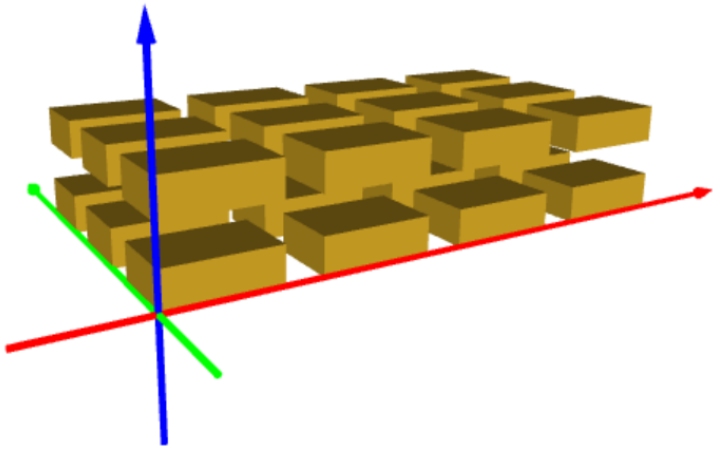


Fig. 46: Cartesian product of the grids `g1`, `g2` and `g3`.

3 First Projects

3.1 Aquarium stand

Objectives:

- Move objects using 3D vectors.
- Display differently colored objects in one scene.
- Merge multiple objects into a new solid object.

You just bought a new aquarium and need to build a stand for it. The stand that you have in mind is shown in Fig. 47.

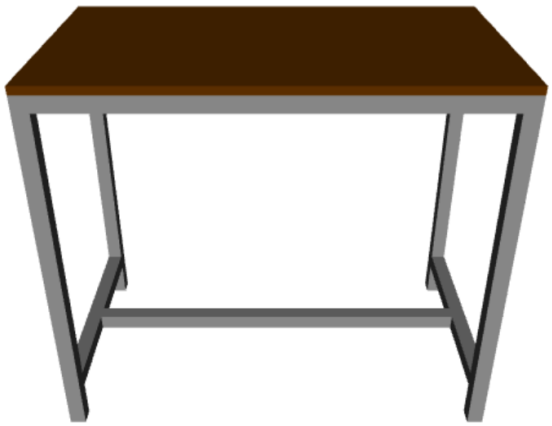


Fig. 47: Aquarium stand.

The dimensions of the aquarium are 70 cm, 40 cm, and 50 cm. Therefore, the outer horizontal dimensions of the stand will be 70 cm and 40 cm. The stand will be made of 1×1 inch bars (1 inch = 2.54 cm), and including a half-inch thick wooden plate on top its height should be 60 cm. For stability reasons, the stand should be reinforced with horizontal bars located 10 cm above ground. Ready?

Design: To begin with, create variables for all measures and dimensions. Doing this at the beginning of every new project is a good habit – it allows you to easily tweak the design later.

```
e = 2.54          # Edge of the cross-section of the bar.
height = 60       # Height of stands including wood.
height_r = 10     # Height of reinforcements.
xdim = 70         # Width of the aquarium.
ydim = 40         # Depth of the aquarium.
zdim = 50         # Height of the aquarium.
```

Notice that the hash symbol '`#`' introduces a *comment*. This is another Python thing – all text after this symbol till the end of line is ignored by the interpreter.

The first leg of the stand is defined using the `BOX` command:

```
l1 = BOX(e, e, height - 0.5 * e)
```

Next we need to create the three other legs. For this we need to introduce the `MOVE` command:

The command `MOVE(obj, tx, ty, tz)` moves the object `obj` by `tx`, `ty`, `tz` in the x, y and z axial directions, respectively. In other words, the object `obj` is moved by the 3D vector `(tx, ty, tz)`.

Minding the thickness of the bars, the remaining three legs of the satnd are created via

```
l2 = COPY(l1)
MOVE(l2, xdim - e, 0, 0)
l3 = COPY(l1)
MOVE(l3, 0, ydim - e, 0)
l4 = COPY(l1)
MOVE(l4, xdim - e, ydim - e, 0, 0)
```

The partial design is displayed in steel color as follows:

```
SHOW(l1, l2, l3, l4)
```

Fig. 48 shows how our design looks at the moment.

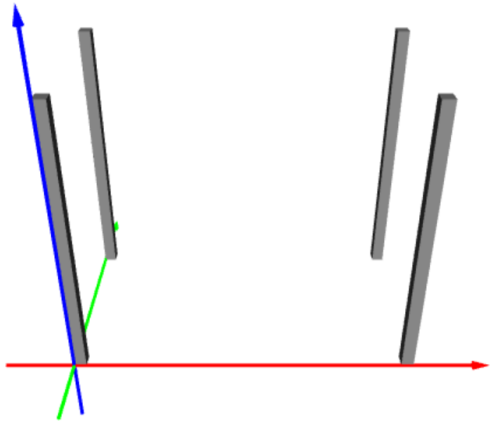


Fig. 48: Four legs of the stand.

Often there are multiple ways that lead to the same design. Here, the four legs together form a Cartesian product geometry: Two intervals in the x-direction times two intervals in the y-direction times one interval in the z-direction. So, the same could be done using the `GRID` and `PRODUCT` commands that were introduced in Subsection 2.25:

```
g1 = GRID(e, -width - 2*e, e)
g2 = GRID(e, -depth - 2*e, e)
g12 = PRODUCT(g1, g2)
g3 = GRID(height - 0.5 * e)
legs = PRODUCT(g12, g3)
```

Next let's take care of the horizontal bars. First the top ring:

```
# Create top bars:
t1 = BOX(xdim, e, e)
MOVE(t1, 0, 0, height - 0.5*e - e)
t2 = COPY(t1)
MOVE(t2, 0, ydim - e, 0)
t3 = BOX(e, ydim, e)
MOVE(t3, 0, 0, height - 0.5*e - e)
t4 = COPY(t3)
MOVE(t4, xdim - e, 0, 0)

# Display the frame:
SHOW(l1, l2, l3, l4, t1, t2, t3, t4)
```

The design after this step is shown in Fig. 49.

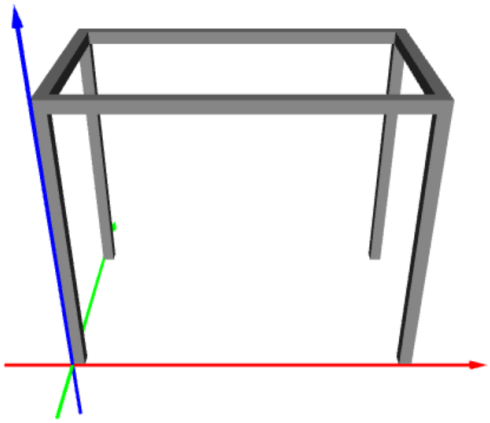


Fig. 49: Four legs and the top ring.

The reinforcement ring is obtained by lowering the top ring by `height - height_r`:

```
# Create bottom reinforcement bars:
b1 = COPY(t3)
MOVE(b1, 0, 0, -(height - height_r))
b2 = COPY(t4)
MOVE(b2, 0, 0, -(height - height_r))
b3 = COPY(t1)
MOVE(b3, 0, 0.5*(ydim - e), -(height - height_r))
```

Finally, we display the frame:

```
# Display the frame:
SHOW(l1, l2, l3, l4, t1, t2, t3, t4, b1, b2, b3)
```

The design after this step is shown in Fig. 50.

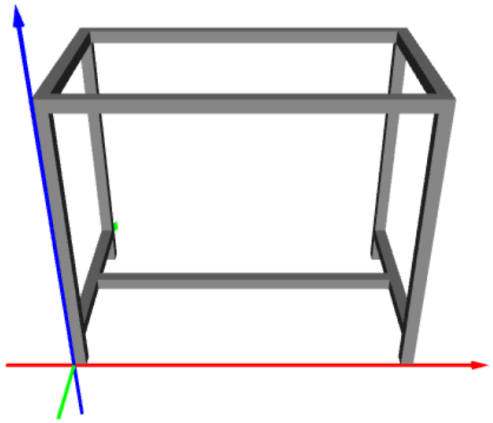


Fig. 50: Complete frame.

After checking visually that all parts of the frame are correct, let us create a new solid object for the frame by merging all 12 bars together. For this, we use the command `UNION`.

The command `UNION(obj1, obj2, obj3, ...)`, that can be abbreviated by `U`, creates a new solid object by merging the objects `obj1`, `obj2`, `obj3`, `...` together. Mathematically, the resulting object is the *set of all points that belong to at least one of these objects*.

In our case we create the new object for the frame as follows:

```
# Create new solid object by merging all parts:
frame = U(l1, l2, l3, l4, t1, t2, t3, t4, b1, b2, b3)
```

As every newly created object, `frame` has the default steel color. Last, the top plate is a `BOX(xdim, ydim, 0.5*e)` that is lifted up by `height - 0.5*e`:

```
# Create the top plate:
top = BOX(xdim, ydim, 0.5*e)
MOVE(top, 0, 0, height - 0.5*e)

# Display frame and top plate together:
wood = [102, 51, 0]
COLOR(top, wood)
SHOW(frame, top)
```

The final design is shown in Fig. 51.

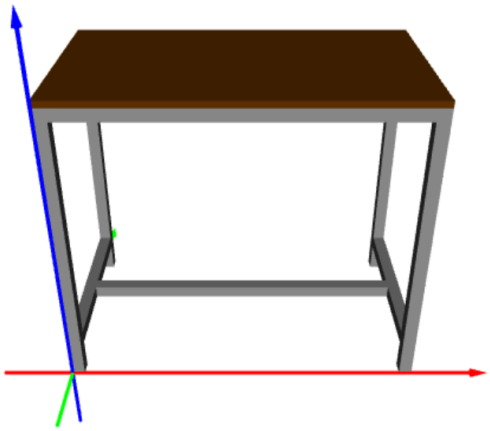


Fig. 51: Final design including the frame and the top plate.

3.2 Water molecule

Objectives: Rotate 3D objects.

Your task is to create a ball-and-stick model of the molecule of water (H_2O) shown in Fig. 52.

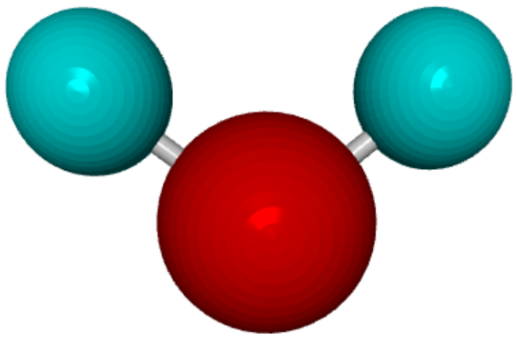


Fig. 52: Water molecule model: oxygen (red), hydrogens (cyan), and bonds (gray).

The only physical parameter that you have to preserve is the angle between the two hydrogen atoms which is 104.45 degrees. All other dimensions of your model should be variable.

Design: To begin with, define colors for the molecules and the bonds:

```
Color_O = [200, 0, 0]
Color_H = [0, 200, 200]
Color_bond = [200, 200, 200]
```

Next create variables for all parameters of the model:

```
R_O = 1.0          # Radius of the O atom.
R_H = 0.7          # Radius of the H atom.
R_bond = 0.1       # Radius of the sticks.
L_bond = 0.2       # Distance of O and H atoms.
Angle = 104.45     # Angle between H atoms (in degrees).
```

For practical reasons, let us redefine the length of the bond to span the distance between the centers of the O and H atoms:

```
L_bond = L_bond + R_O + R_H
```

Now create the atom of oxygen and one atom of hydrogen:

```
Sphere_O = SPHERE(R_O)
Sphere_H = SPHERE(R_H)
```

If we displayed them together right now, then the H atom would not be visible since it would be inside the O atom. So, let us move the H atom by the length of the bond in the direction of the x-axis:

```
MOVE(Sphere_H, L_bond, 0, 0)
```

The current state of our model is shown in Fig. 53.

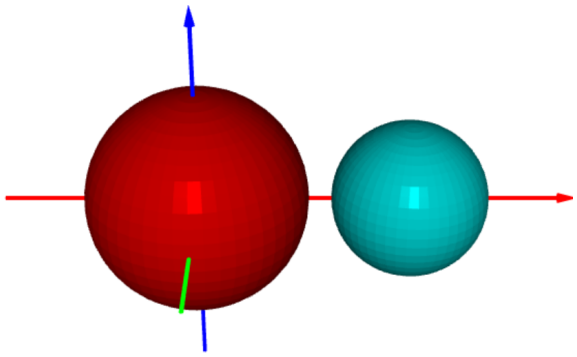


Fig. 53: O and H atoms.

Next create a cylinder (stick) representing the bond:

```
Bond = CYL(R_bond, L_bond)
```

After this command, the situation is depicted in Fig. 54.
Of course this is not what we want – the bond need to be connecting the O and H atoms. Hence we will need to rotate the cylinder by 90 degrees about the y-axis. This is done with the `ROTATE` command.

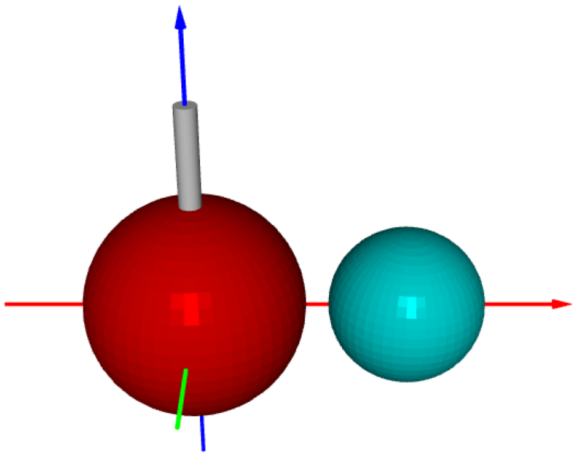


Fig. 54: Creating a cylinder to model the bond.

The command `ROTATE(obj, angle_in_degrees, axis)` that can be abbreviated as `R(obj, angle_in_degrees, axis)`, returns a new object that is obtained by rotating the object `obj` counter-clockwise (CCW) about the axis `axis` by angle `angle`. Here `axis = 1` for the x-axis, `axis = 2` for the y-axis and `axis = 3` for the z-axis. By default, `angle` is in degrees. Commands `ROTATERAD` and `RRAD` can be used with angles in radians.

Hence the following command will rotate the bond to the correct position:

```
ROTATE(Bond, -90, 2)
```

The updated geometry is shown in Fig. 55.

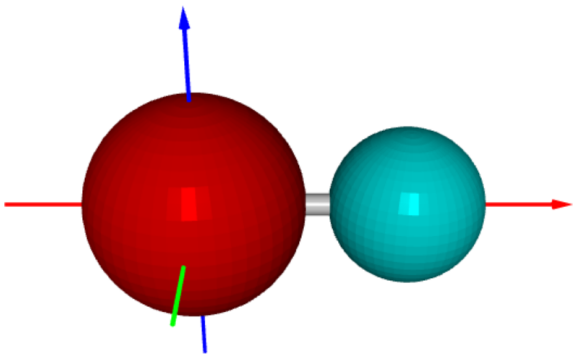


Fig. 55: Rotating the cylinder by 90 degrees.

Last, rotate both the cylinder and the hydrogen atom, assign colors, and diplay:

```
Bond_2 = COPY(Bond)
ROTATE(Bond_2, Angle, 3)
Sphere_H_2 = COPY(Sphere_H)
ROTATE(Sphere_H_2, Angle, 3)

COLOR(Sphere_O, Color_O)
COLOR(Sphere_H, Color_H)
COLOR(Sphere_H_2, Color_H)
COLOR(Bond, Color_bond)
COLOR(Bond_2, Color_bond)

SHOW(Sphere_O, Sphere_H, Bond, Sphere_H_2, Bond_2)
```

The final geometry of the molecule is displayed in Fig. 56.

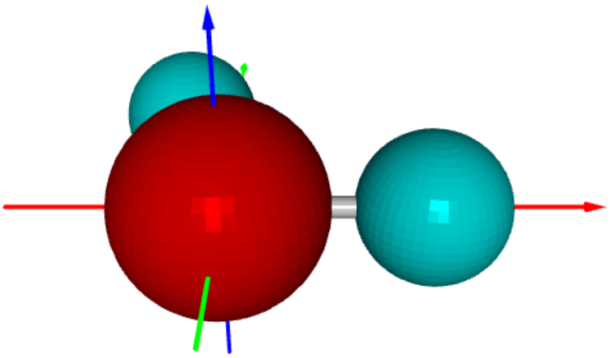


Fig. 56: Final design.

3.3 Coffee table

Objectives:

- Scale objects.
- Subtract objects from each other.

Your third task is to create a model of an oval coffee table that is shown in Fig. 57.

Fig. 57: Oval coffee table.

This is a real coffee table that is sold by the Fong Brothers Company in Los Angeles. The width, depth and height of the table are 48, 24 and 18 inches, respectively.

Design: As usual we first introduce variables for all design parameters:

```
w = 48.0            # Width.
d = 24.0            # Depth.
h = 18.0            # Height.
top_outer = 4.5     # Thickness of top (outer view).
top_inner = 1.0     # Thickness of top (inner view).
leg_width = 7.0     # Width of legs.
leg_depth = 2.0     # Depth of legs.
```

The departure point for the design will be a cylinder of radius `d/2` and height `h`:

```
body = CYL(d/2.0, h, 128)
```

which is shown in Fig. 58.

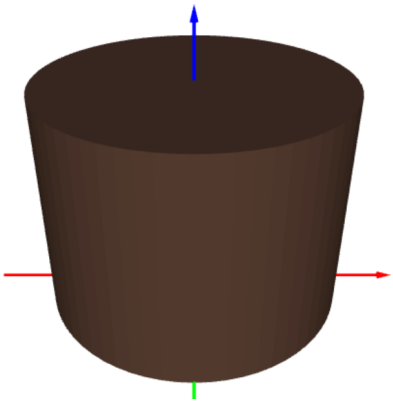


Fig. 58: The design starts with a cylinder.

In the next step, the cylinder will be transformed to an oval cylinder via the `SCALE` command.

The command `SCALE(obj, sx, sy, sz)` that can be abbreviated simply as `S(obj, sx, sy, sz)` stretches or shrinks the object `obj` by `sx`, `sy`, `sz` in the x, y and z axial directions, respectively.

We will be scaling an object `body` by `w / d` in the x-direction:

```
SCALE(body, w / d, 1.0, 1.0)
```

The result after scaling is shown in Fig. 59.

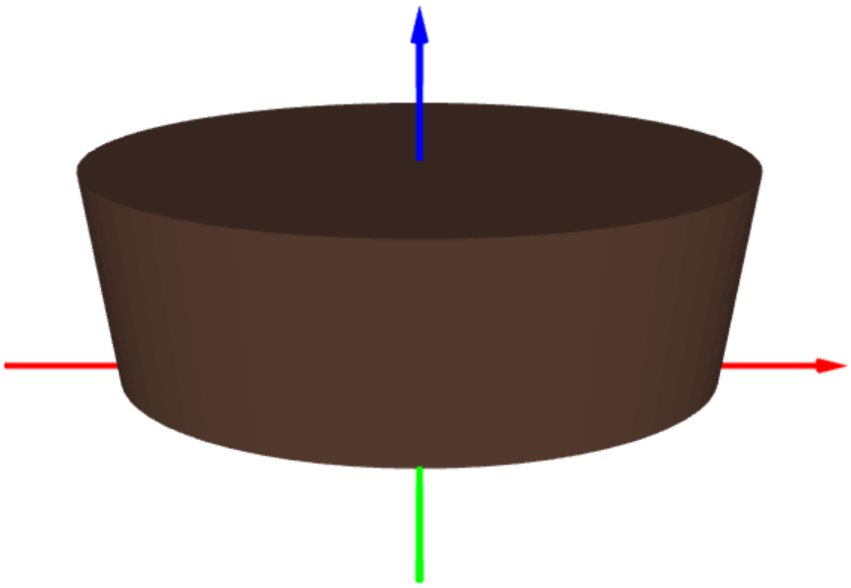


Fig. 59: Scaling by 2.0 in the x-direction.

Now we need to cut off four corners in order to define the width of the legs. This is done via the command `DIFFERENCE`.

The command `DIFFERENCE(obj, cut1, cut2, ...)`, possibly abbreviated as `DIFF` or `D`, creates a new solid object by subtracting the objects `cut1`, `cut2`, `...` from the object `obj`. Mathematically, the resulting object is the *set of all points that belong to object* `obj` *but do not lie in* `cut1`, `cut2`, `....`

The object to be subtracted from the oval cylinder will be constructed by subtracting two boxes from a larger box:

```
b = BOX(w, d, h - top_outer)
MOVE(b, -w/2.0, -d/2.0, 0)
b_cut1 = COPY(b)
SCALE(b_cut1, leg_width / w, 1.0, 1.0)
b_cut2 = COPY(b)
SCALE(b_cut2, 1.0, leg_width / d, 1.0)
b = DIFF(b, b_cut1, b_cut2)
```

The object `b` is displayed in Fig. 60.

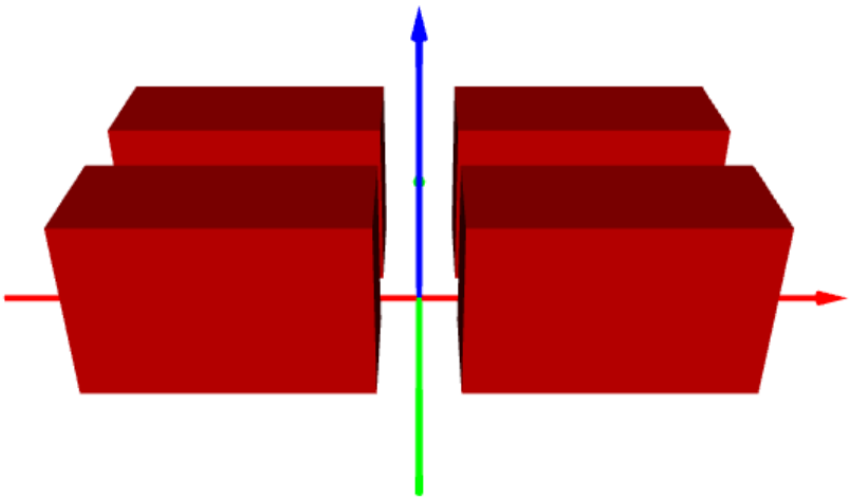


Fig. 60: Object to be subtracted from the oval cylinder.

Next we subtract `b` from `body`,

```
table = DIFF(body, b)
```

and display the result in Fig. 61.

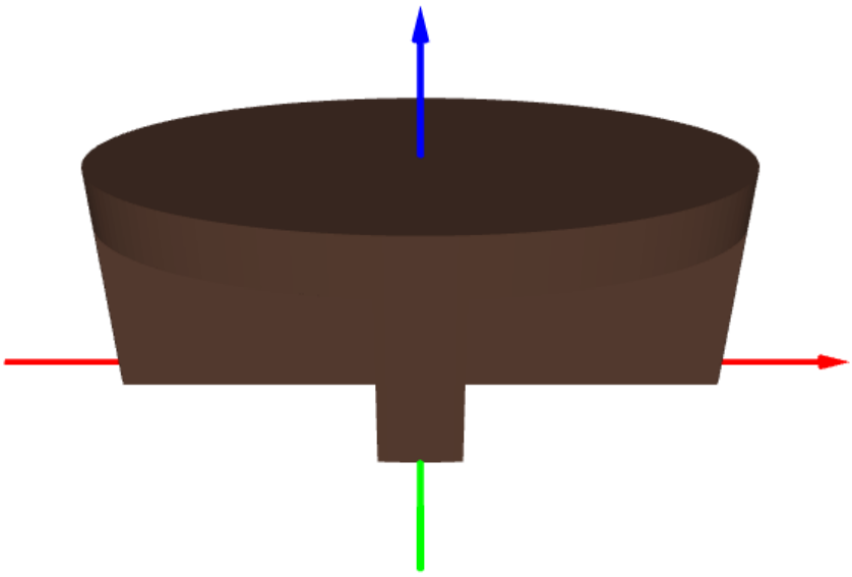


Fig. 61: Oval cylinder after subtracting the object `b`.

Note that we preserved the original oval cylinder in the variable `body`. Now we will scale it down as needed in order to remove the interior part of the table:

```
# Scale down the object "body":
SCALE(body, (w - 2*leg_depth) / w, \
      (d - 2*leg_depth) / d, (h - top_inner) / h)

# Display it along with the object "table":
SHOW(table, body)
```

The situation is depicted in Fig. 62.

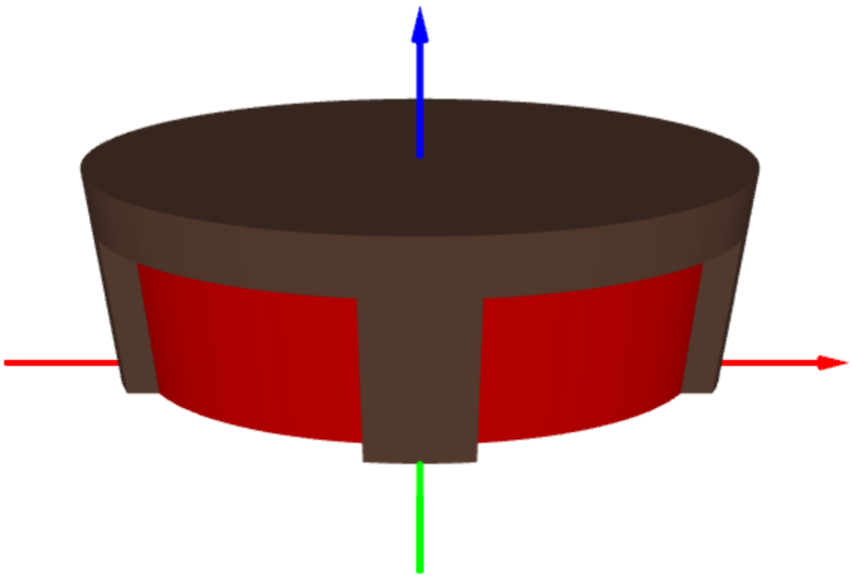


Fig. 62: Table before removing the inner part.

As the last step, we remove the object `body` from the object `table`:

```
table = DIFF(table, body)
```

The final design in shown in Fig. 63.

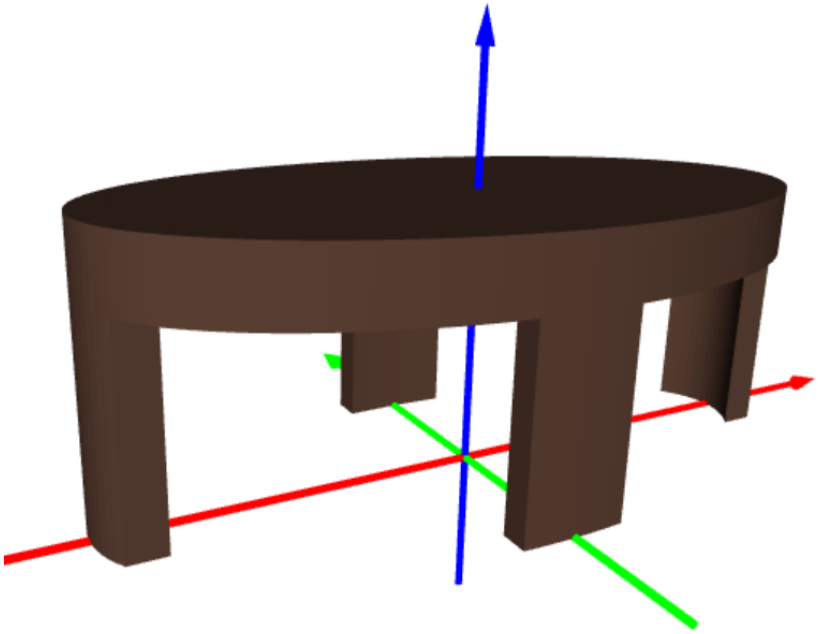


Fig. 63: Final design.

3.4 Geometry labs

Objectives:

- Intersect 3D objects with planes represented by thin 3D solids.
- Experiment with conic sections.

PLaSM makes it possible to intersect objects via the `INTERSECTION` command.

The command `INTERSECTION(obj1, obj2, obj3, ...)` that can be abbreviated as `I(obj1, obj2, obj3, ...)` creates a new solid object by intersecting the objects `obj1`, `obj2`, `obj3`, ... Mathematically, the resulting object is the *set of all points that belong to all the objects*. In other words, if some point does not lie in one of the objects, it does not belong to the intersection.

In this subsection we will be using cones and planes to construct conic sections. With what you learn here, you will be able to cut any other 3D objects.

Let us begin with creating a cone of radius `r = 1` and height `h = 2`:

```
r = 1.0
h = 2.0
cone = CONE(r, h, 128)
COLOR(cone, GOLD)
SHOW(cone)
```

The parameters `r` and `h` are variables that you can change them easily if needed. The optional third parameter `128` in the cone's definition is used to make the surface smoother-looking (recall from Subsection 2.18 that the default value is `64`). The cone is shown in Fig. 64.

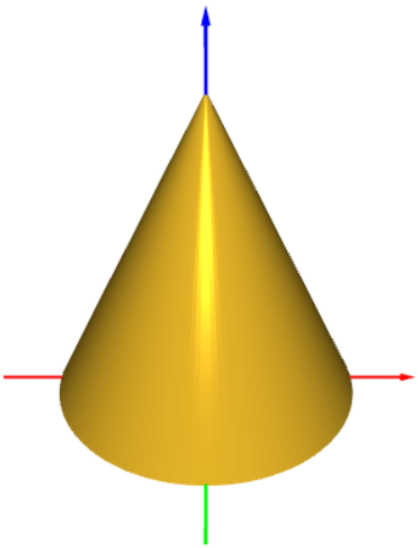


Fig. 64: Cone with radius `r = 1` and height `h = 2`.

Constructing a circle

A circle is obtained by cutting the cone with a plane that is normal (perpendicular) to its axis. Hence, let's add a plane, represented by a square of size `s = 3`. The square will be created using the `SQUARE3D` command and displayed along with the cone in steel color:

```
s = 3.0
plane = SQUARE3D(s)
COLOR(cone, GOLD)
SHOW(cone, plane)
```

The two objects are shown together in Fig. 65.

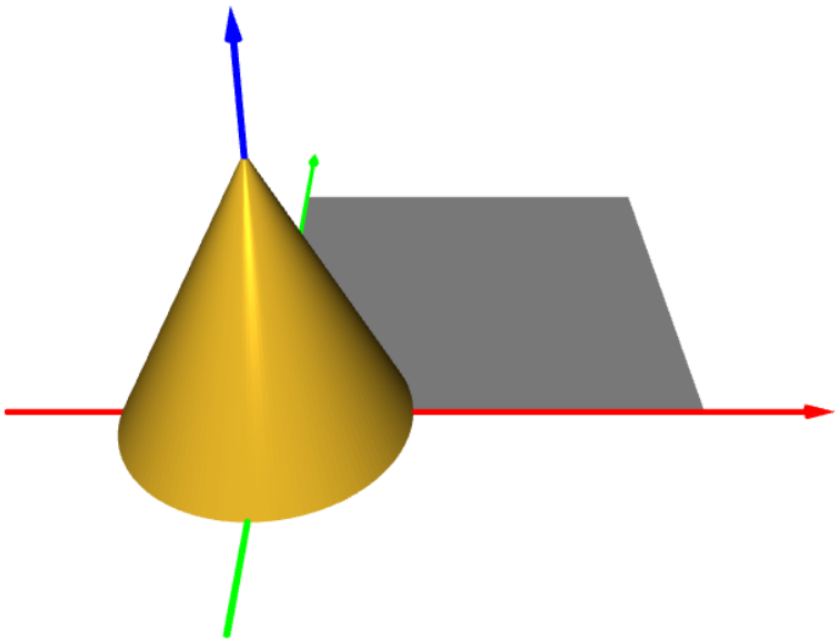


Fig. 65: Square is created at its default position in the first quadrant.

We need to move the square in the xy-plane so that the z-axis passes through its center, and at the same time to lift it by `e` in the z-direction. Hence the 3D translation vector is `(-s/2.0, -s/2.0, e)`:

```
e = 1.0
MOVE(plane, -s/2.0, -s/2.0, e)
COLOR(cone, GOLD)
SHOW(cone, plane)
```

The situation is depicted in Fig. 66.

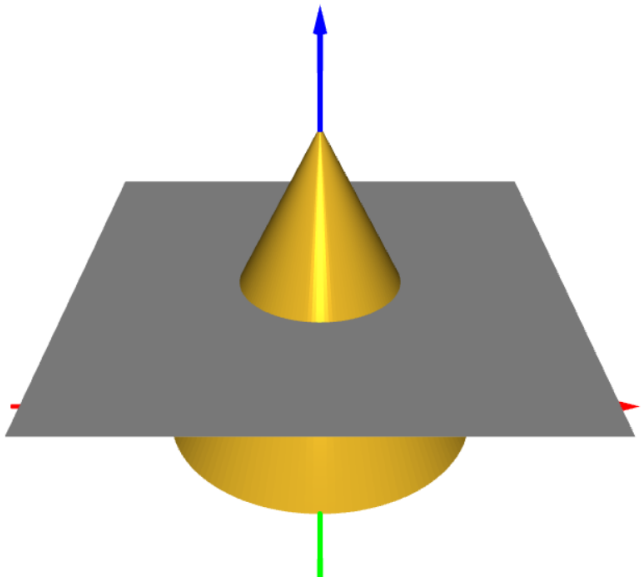


Fig. 66: Square is centered and lifted to elevation `e`.

Now we can create the conic section:

```
circle = INTERSECTION(cone, plane)
COLOR(circle, GOLD)
SHOW(circle)
```

The intersection is shown in Fig. 67.

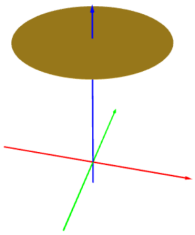


Fig. 67: The circular conic section.

We can also visualize the intersection with the rest of the plane. For this, we subtract the cone from the plane, and view it together with the circle:

```
pierced_plane = DIFF(plane, cone)
COLOR(circle, GOLD)
SHOW(circle, pierced_plane)
```

The ensemble is shown in Fig. 68.

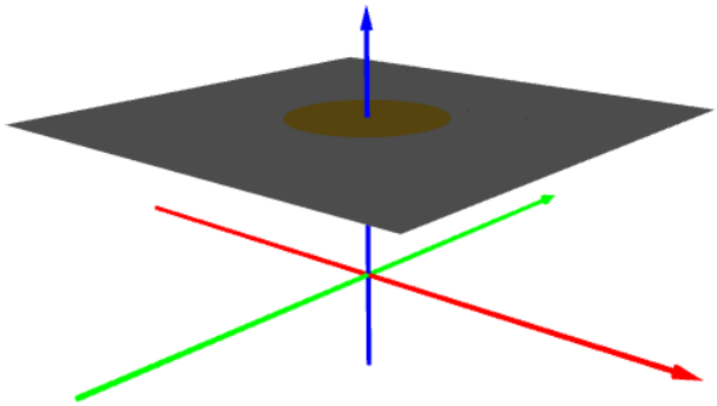


Fig. 68: The conic section displayed along with the rest of the plane.

Let's play some more! We can cut off the upper part of the cone, to obtain a truncated cone. This can be done using the same sequence of operations as above, only the plane will be replaced by a half-space, represented by a cube. Let us display the two objects together first:

```
s = 3.0
cube = CUBE(s)
e = 1.0
MOVE(cube, -s/2.0, -s/2.0, e)
COLOR(cone, GOLD)
SHOW(cone, cube)
```

The two objects are displayed in Fig. 69.

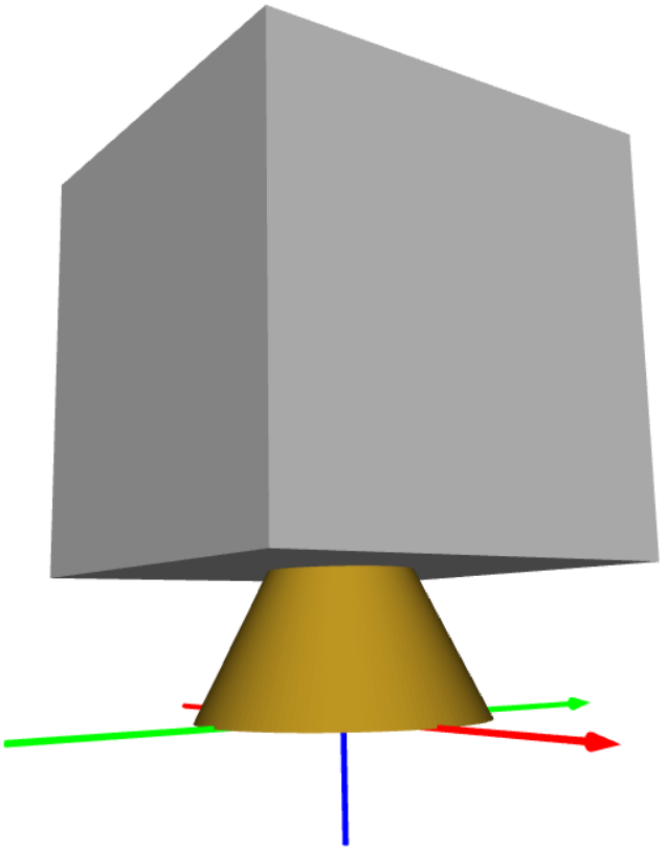


Fig. 69: The cone and the half-space, represented by a large cube.

Next let us remove the half-space from the cone:

```
tcone = DIFF(cone, cube)
COLOR(tcone, GOLD)
SHOW(tcone)
```

The result is shown in Fig. 70.

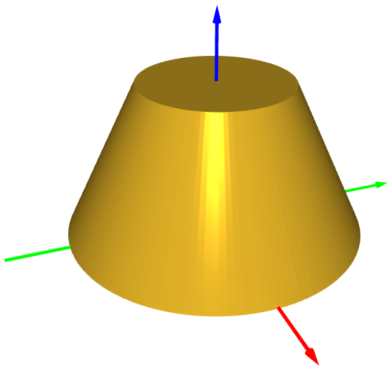


Fig. 70: Removing the half-space from the cone yields a truncated cone.

The upper part of the cone is an intersection of the cone with the cube:

```
tip = INTERSECTION(cone, cube)
COLOR(tcone, GOLD)
SHOW(tcone, tip)
```

The truncated cone in gold color, along with the upper part in steel color, are shown in Fig. 71.

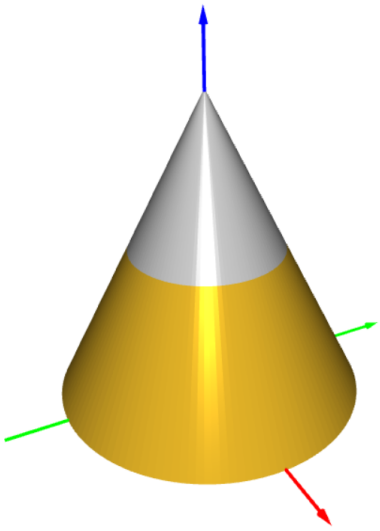


Fig. 71: The two parts of the cone shown in different colors.

The math behind the scenes

The equation for the cone of radius r and height h shown in Fig. 64 is

$$z = h\left(1 - \frac{\sqrt{x^2 + y^2}}{r}\right).$$

In other words, all 3D points that satisfy the above equation belong to the surface of the cone. The plane shown in Fig. 66 is described via the equation

$$z = e.$$

The intersection of the plane and the surface of the cone is formed by all points that satisfy the above two equations at the same time. Hence, we obtain

$$e = h\left(1 - \frac{\sqrt{x^2 + y^2}}{r}\right).$$

After a simple manipulation that involves dividing the equation by h, subtracting 1 from both sides, and multiplying the equation with r, we obtain

$$r\left(1 - \frac{e}{h}\right) = \sqrt{x^2 + y^2}.$$

Here, $\sqrt{x^2 + y^2}$ is the distance of the point (x, y, z) from the z-axis. Hence, the last equation represents a set of all 3D points whose distance from the z-axis is $r(1 - e/h)$. This is a cylindrical surface. But keeping in mind that we also have the equation $z = e$, the resulting set of points is a circle whose center lies on the z-axis and whose elevation above the xy-plane is e.

Constructing an ellipse

An ellipse is obtained by intersecting the cone with a plane that is not normal to the cone axis, but that is also not steeper that the slope of the cone. For this example we will use the cone from Fig. 64, but we will make it three times bigger (both r and h) and move it by -4 units on the z-axis so that its apex is at $z = 2$ as before:

```
r = 3.0
h = 6.0
cone = CONE(r, h, 128)
MOVE(cone, 0, 0, -4)
```

Next we add a plane represented by a square of size `s = 7`, move it so that its center lies on the z-axis, and rotate by 45 degrees about the y-axis:

```
s = 7.0
plane = SQUARE3D(s)
MOVE(plane, -s/2.0, -s/2.0, 0)
ROTATE(plane, 45, 2)
COLOR(cone, GOLD)
SHOW(cone, plane)
```

The situation is shown in Fig. 72.

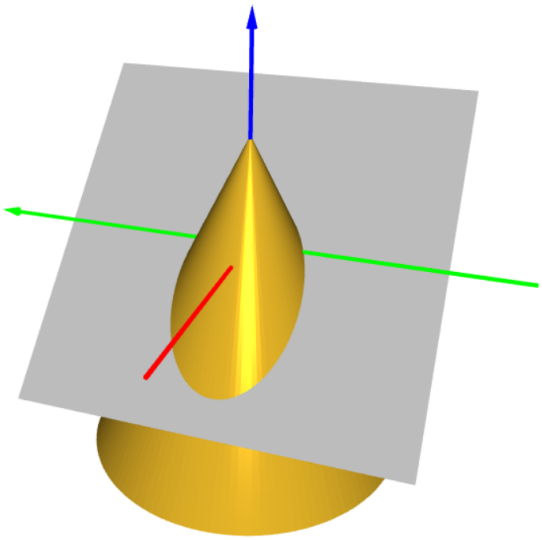


Fig. 72: The cone and the plane.

With both objects in place, we can create their elliptic intersection. It will be displayed in gold color along with the rest of the plane in steel color:

```
ellipse = INTERSECTION(cone, plane)
pierced_place = DIFF(plane, cone)
COLOR(ellipse, GOLD)
SHOW(ellipse, pierced_plane)
```

The result is shown in Fig. 73.

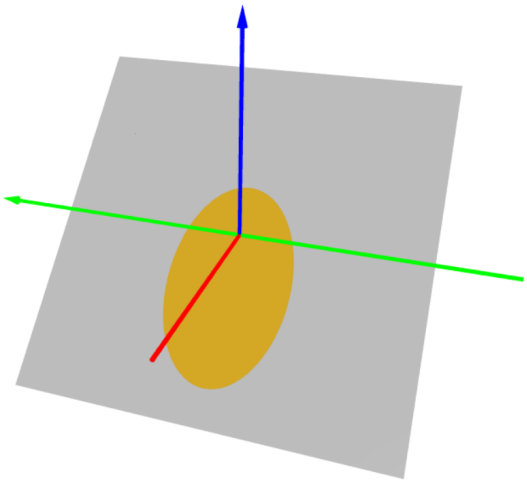


Fig. 73: The elliptic intersection.

Finally, if we like to, we can remove the upper part of the cone in the same way we did before – replacing the plane with a cube, translating and rotating the cube in the same way as the plane, and subtracting it from the cone. The result is shown in Fig. 74.

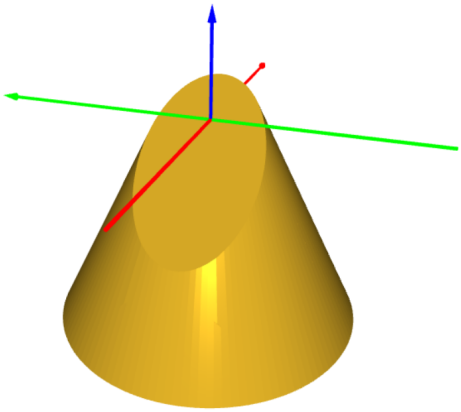


Fig. 74: Exposing the elliptic cut.

Constructing a parabola

Parabola is obtained when a cone is intersected with a plane that has the same slope as the surface of the cone, and that does not pass through its apex. We will stay with the

same cone as before, just the plane will be enlarged:

```
s = 10.0
plane = SQUARE3D(s)
MOVE(plane, -s/2.0, -s/2.0, 0)
```

To calculate the slope of the plane, we import the `arctan` function from Numpy, Python's numerical computation library:

```
from numpy import arctan
alpha = arctan(h/r) * 180 / PI
```

Next we rotate the plane by angle `alpha` about the y-axis:

```
ROTATE(plane, alpha, 2)
COLOR(cone, GOLD)
SHOW(cone, plane)
```

Note that this rotation is in radians, not in degrees – this is because the function `arctan` returns angles in radians. The two objects are shown in Fig. 75.

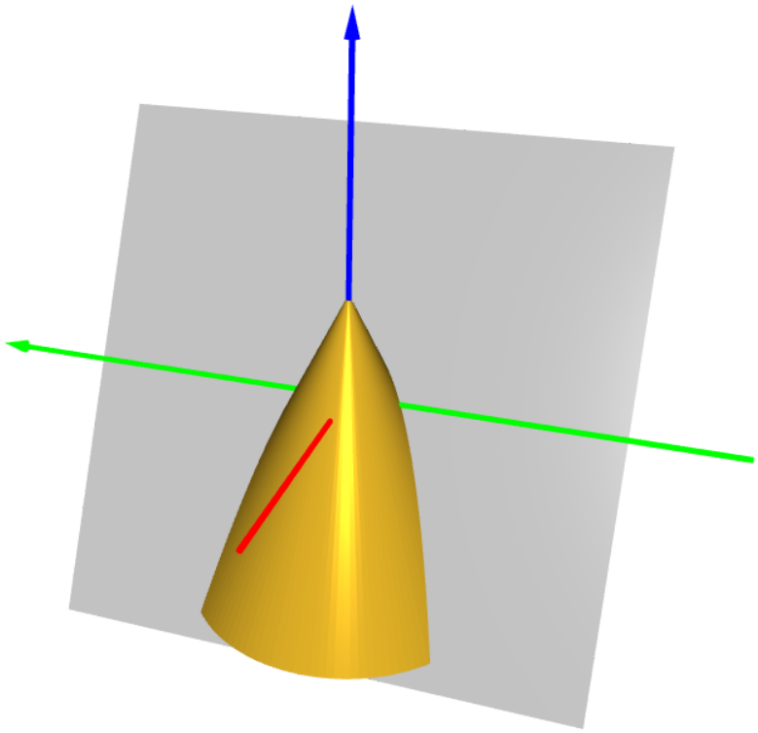


Fig. 75: The cone and the plane.

Now we can calculate the intersection of the plane with the cone, and display it along with the rest of the plane:

```
parabola = INTERSECTION(cone, plane)
pierced_plane = DIFF(plane, parabola)
COLOR(parabola, GOLD)
SHOW(parabola, pierced_plane)
```

The parabola is shown in Fig. 76.

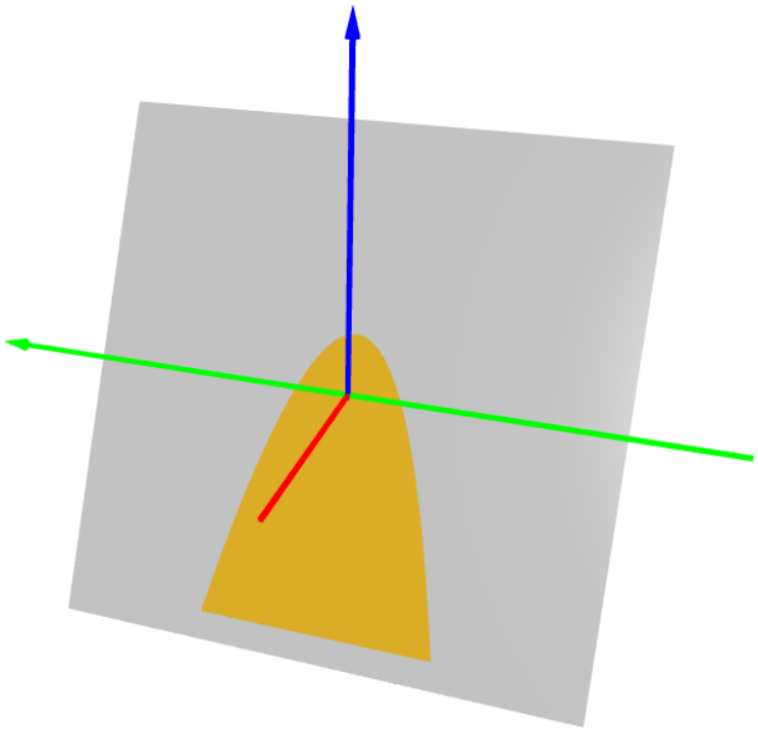


Fig. 76: Parabolic intersection (truncated).

Note that the parabola is truncated because both the cone and the plane are represented by objects of finite size. Ideally, if both of them were infinite, also the parabola would extend to infinity.

Last we can replace the plane with a cube, and remove the upper part of the cone:

```
s = 10.0
cube = CUBE(s)
MOVE(cube, -s/2.0, -s/2.0, 0)
ROTATE(cube, alpha, 2)
tcone = DIFF(cone, cube)
COLOR(tcone, GOLD)
SHOW(tcone)
```

The result is shown in Fig. 77.

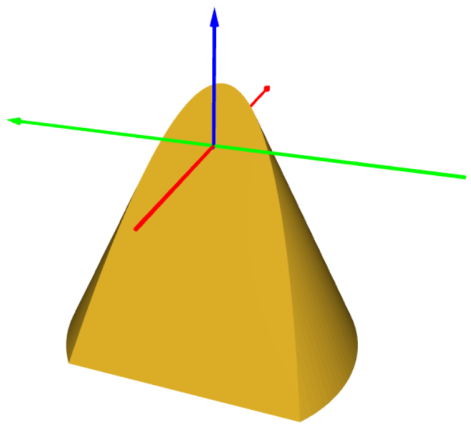


Fig. 77: Exposing the parabolic cut.

Constructing a hyperbola

Hyperbola is obtained by intersecting the cone with a plane that is either parallel to the axis of the cone, or its slope is steeper than the slope of the surface of the cone.

Hyperbolas have two arms, therefore we will use the full mathematical form of the cone which we did not need until now. It consists of two "usual" cones that share the same axis and touch at the apex. They are constructed as follows:

```
r = 3.0
h = 6.0
cone1 = CONE(r, h, 128)
MOVE(cone1, 0, 0, -6)
cone2 = COPY(cone1)
ROTATE(cone2, 180, 1)
cone = UNION(cone1, cone2)
```

The double cone is shown in Fig. 78.

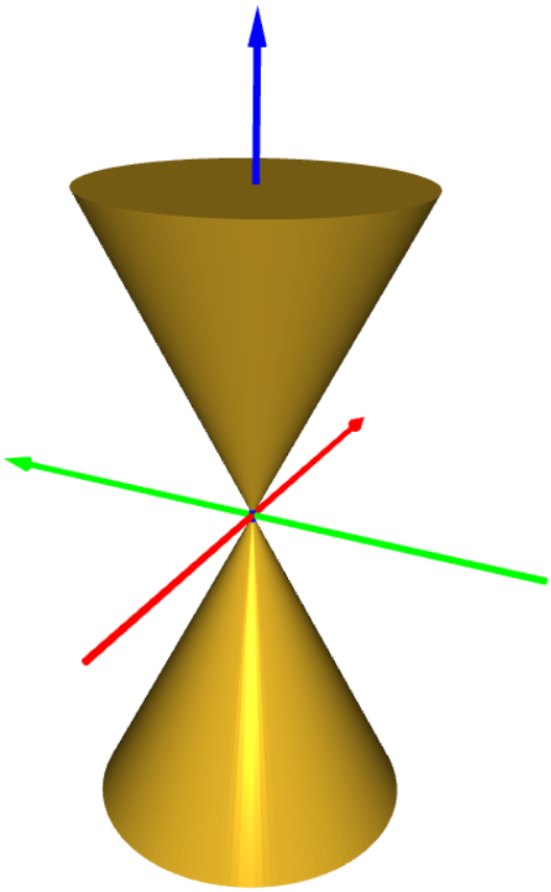


Fig. 78: The double cone.

Next let us create the plane. We will create a sufficiently large square using the `SQUARE3D` command, and move it so that its center lies on the z-axis:

```
s = 12.0
plane = SQUARE3D(s)
MOVE(plane, -s/2.0, -s/2.0, 0)
```

The plane is rotated by angle `alpha` about the y-axis. This angle does not have to be 90 degrees as in our example below. But the slope of the cone must not become less than or equal to the slope of the cone – otherwise we would end up with an ellipse or parabola:

```
alpha = 90
ROTATE(plane, alpha, 2)
```

The plane should not pass through the apex:

```
D = -0.5
MOVE(plane, D, 0, 0)
```

Let's display the two objects together:

```
COLOR(cone, GOLD)
SHOW(cone, plane)
```

The double cone and the plane are shown in Fig. 79.

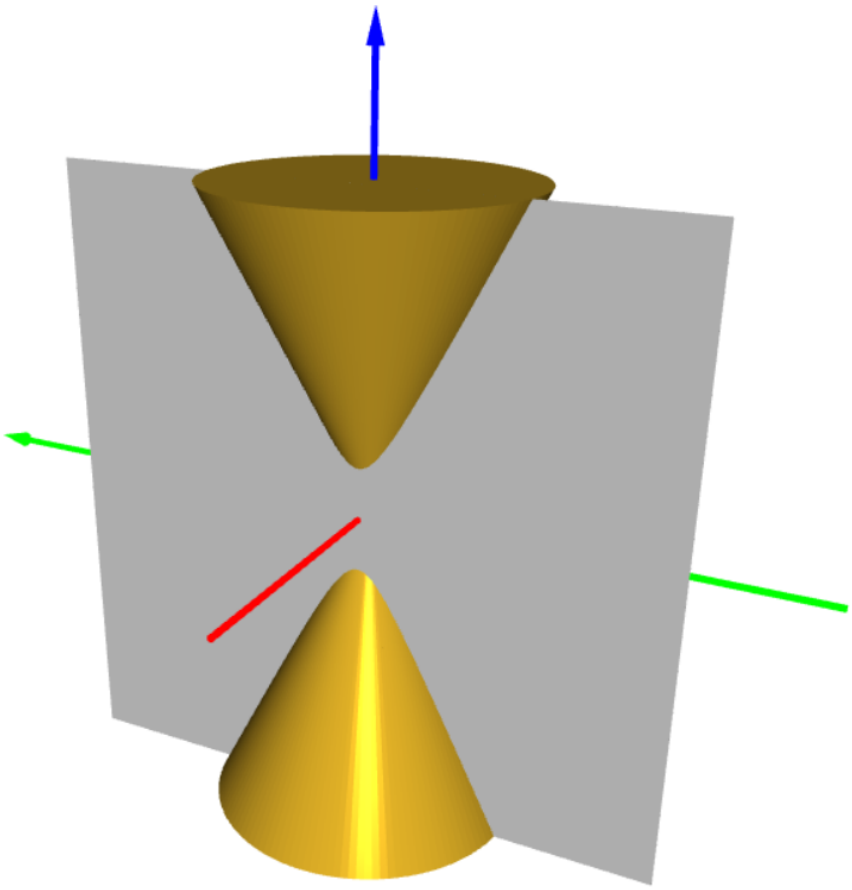


Fig. 79: The double cone and the plane.

Now we can already create the hyperbolic intersection:

```
hyperbola = INTERSECTION(cone, plane)
pierced_plane = DIFF(plane, hyperbola)
COLOR(hyperbola, GOLD)
SHOW(hyperbola, pierced_plane)
```

The output is shown in Fig. 80.

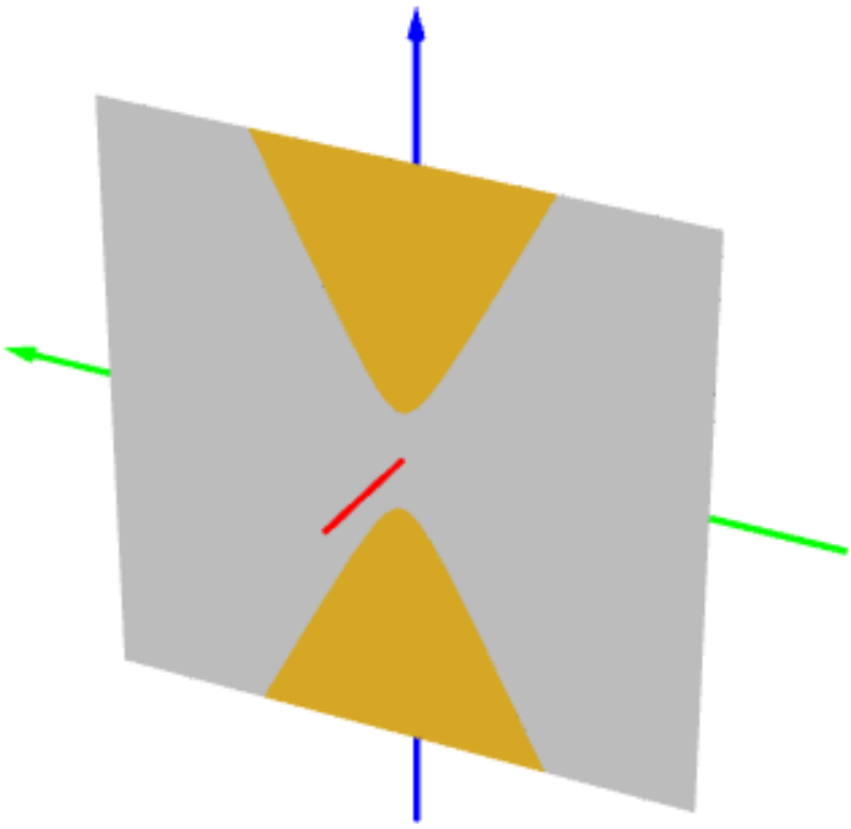


Fig. 80: Hyperbolic intersection.

Finally, as we did in the previous examples, let us replace the plane with a cube, and expose the hyperbolic cut by removing part of the double cone:

```
# Replace the plane with a cube, move
# and rotate it in the same way, and subtract
# from the cone:
s = 12.0
cube = CUBE(s)
MOVE(cube, -s/2.0, -s/2.0, 0)
# Rotate the cube by 90 degrees
# about the y-axis.
alpha = 90
ROTATE(cube, alpha, 2)
```

```
# Move the cube a bit from the origin:
D = -0.5
MOVE(cube, D, 0, 0)
# Subtract the cube from the cone:
tcone = DIFF(cone, cube)
COLOR(tcone, GOLD)
SHOW(tcone)
```

What remains of the double cone is shown in Fig. 81.

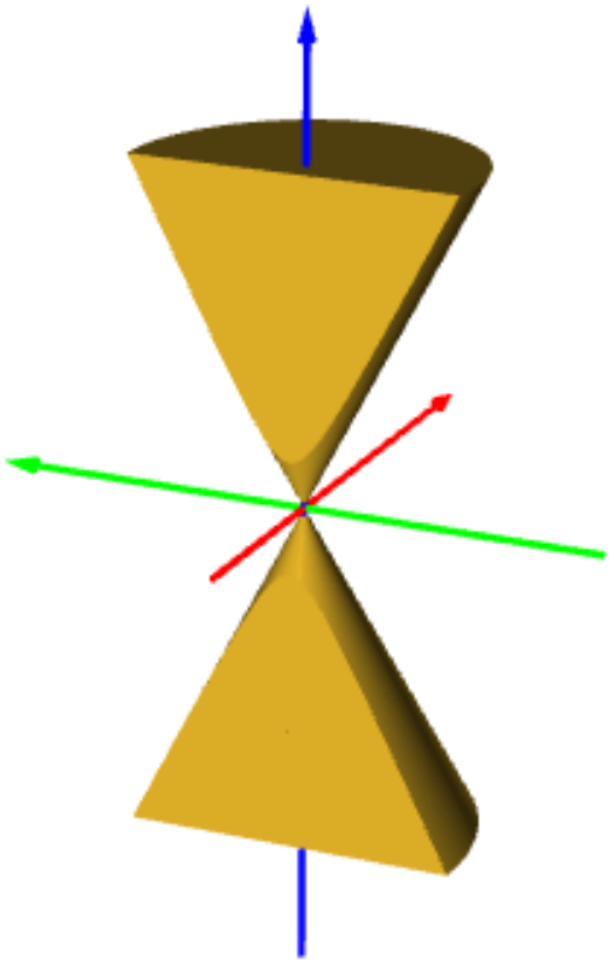


Fig. 81: Exposing the hyperbolic cut.

3.5 3D puzzle

Objectives: Intersect 3D objects.

Task

Construct a 3D object that can fit exactly into all three holes in the wooden piece shown in Fig. 82!

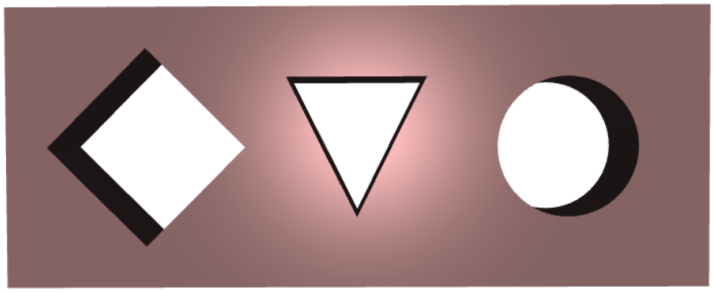


Fig. 82: 3D puzzle.

The three holes have the following parameters:

- 2×2 inch square,
- symmetric triangle of base 2 inches and height 2 inches,
- circle of radius 1.0 inches.

Impossible? Turn the page.

Solution

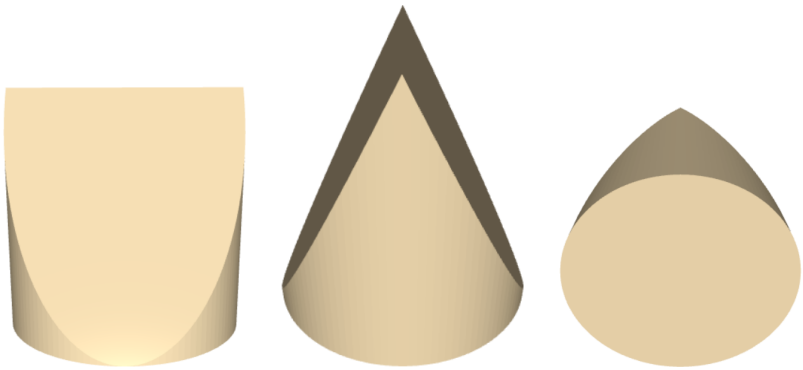


Fig. 83: Solution of the 3D puzzle.

Here is the script that creates the object:

```
# First create a cube of size 2:
c = CUBE(2.0)

# Move the cube so that its center lies at (0, 0, 0):
MOVE(c, -1, -1, -1)

# Next create a cylinder with radius 1 and height 2:
cyl = CYL(1, 2)

# Move also the cylinder so that its center lies at (0, 0, 0):
MOVE(cyl, 0, 0, -1)

# Create a prism via convex hull:
p = CHULL([1, -1, -1], [1, 0, 1], [1, 1, -1], \
          [-1, -1, -1], [-1, 0, 1], [-1, 1, -1])

# Calculate the intersection of the three objects:
out = INTERSECTION(c, cyl, p)
```

4 Advanced Topics

Objectives:

- Learn about the XOR operation with solid objects.
- Learn more details about scaling.
- Learn advanced alignment operations.
- Learn to measure dimensions of objects.
- Learn to use Boolean operations more efficiently.

4.1 XOR of objects

XOR ("exclusive or") is after the union, intersection, and difference of objects the last important Boolean operation of solid modeling.

The command `XOR(obj1, obj2)` creates a new solid object by calculating the union of the objects `obj1` and `obj2`, and subtracting their intersection from it. Mathematically, the resulting object is the *set of points that lie either in* `obj1` *or in* `obj2` *but not in both*.

For illustration let us overlap two differently rotated cubes, show their union and intersection, and calculate their XOR:

```
# Create a cube of size 2:
c = CUBE(2)

# Move c so that its center lies at origin (0, 0, 0):
MOVE(c, -1, -1, -1)

# Rotate cube c by 45 degrees about the z-axis:
d = COPY(c)
ROTATE(d, 45, 3)

# Assign colors:
union = UNION(c, d)
COLOR(union, GOLD)
intersection = INTERSECTION(c, d)
COLOR(intersection, RED)

# Display the result:
SHOW(union, intersection)
```

```
# Calculate XOR of c and d, and assign gold color to it:
e = XOR(c, d)
COLOR(e, GOLD)

# Display the result:
SHOW(e)
```

The two outputs are shown in the left and right parts of Fig. 84.

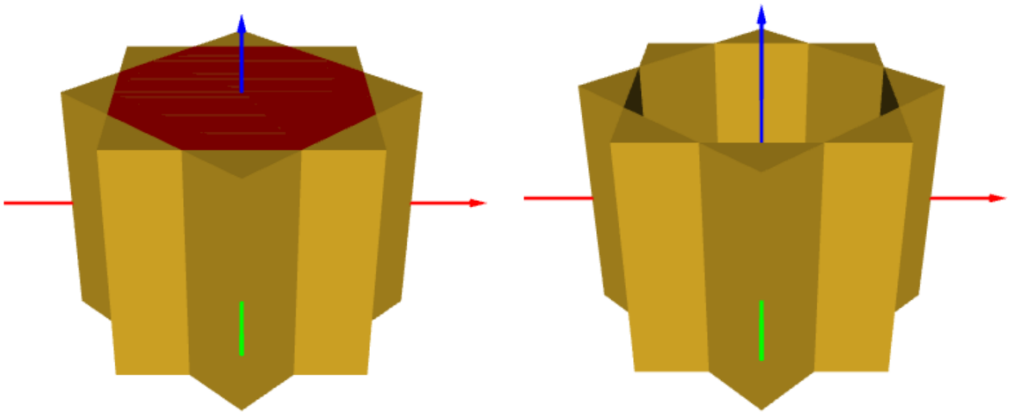


Fig. 84: Left: cubes `c` and `d` with their intersection in red. Right: the XOR of `c` and `d`.

4.2 More on scaling

When applying the command `SCALE(obj, sx, sy, sz)`, or its abbreviation `S(obj, sx, sy, sz)`, the x-coordinates of all points in the object `obj` are multiplied with `sx`, all y-coordinates with `sy`, and all z-coordinates with `sz`. Hence, if the object `obj` is not centered at the origin, it *moves in addition to being scaled*. This is best illustrated using a simple script where a cube is moved away from the origin and then scaled down in size several times:

```
# Create a cube of size 8 and
# move it away from the origin (0, 0, 0):
c = CUBE(8)
MOVE(c, 8, 8, 8)
```

```
# Then scale it down to 50% of size several times:
d = COPY(c)
SCALE(d, 0.5, 0.5, 0.5)
e = COPY(d)
SCALE(e, 0.5, 0.5, 0.5)
f = COPY(e)
SCALE(f, 0.5, 0.5, 0.5)
g = COPY(f)
SCALE(g, 0.5, 0.5, 0.5)
h = COPY(g)
SCALE(h, 0.5, 0.5, 0.5)
# Create compound object:
s = [c, d, e, f, g, h]
# Assign gold color to it and display:
COLOR(s, GOLD)
SHOW(s)
```

The output is shown in Fig. 85.

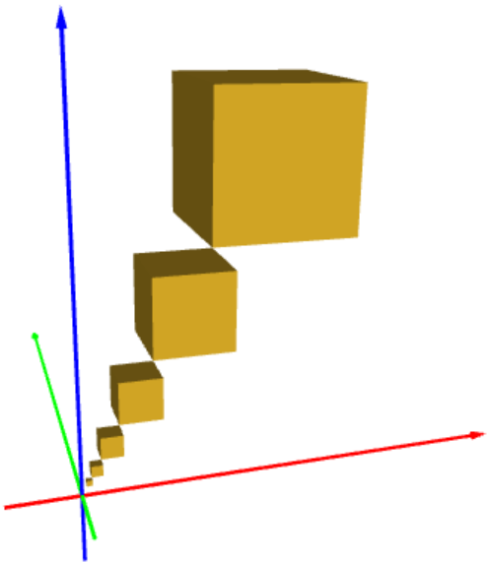


Fig. 85: This experiment illustrates that scaling objects also moves them in space if they are not centered at the origin $(0, 0, 0)$.

4.3 Commands `TOP` and `BOTTOM`

The command `TOP(obj1, obj2)` moves the object `obj2` so that it is positioned right above `obj1`. For illustration, let us create a cube and a sphere, and move the sphere on top of the cube:

```
c = CUBE(2.0)
s = SPHERE(1.0)
TOP(c, s)
```

The output is shown in Fig. 86.

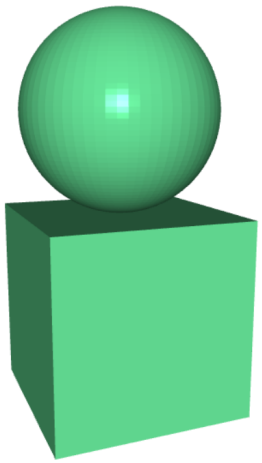


Fig. 86: Moving a sphere on top of a cube with the `TOP` command.

Analogously, the command `BOTTOM(obj1, obj2)` moves the object `obj2` under the object `obj1`. Fig. 87 shows the output of the script

```
c = CUBE(2.0)
s = SPHERE(1.0)
BOTTOM(c, s)
```

In both cases, the object `obj2` is aligned in such a way that its center is right above / below the center of `obj1`.

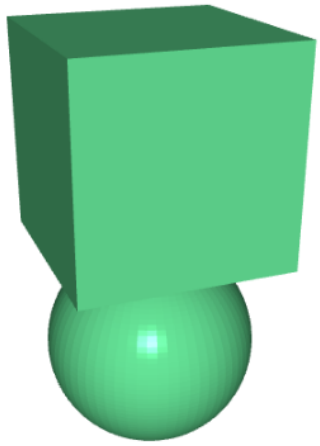


Fig. 87: Moving a sphere under a cube with the `BOTTOM` command.

4.4 Measuring dimensions and printing out information

Sometimes we need to measure the dimensions of an object. The command `SIZE( obj, 1)` returns the size of the object `obj` in the first spatial direction (x-axis). Similarly, commands `SIZE(obj, 2)` and `SIZE(obj, 3)` return the size of the object `obj` in the direction of the y-axis and in the direction of the z-axis, respectively. The command `print` can be used to print out the results. For example, the output of the script

```
a = BOX(3, 2, 1)
print "Object dimension in x-direction:", SIZE(a, 1)
```

will have the following output:

```
Object dimension in x-direction: 3.0
```

The values returned by the `SIZE` command can be stored in variables and more of them can be printed at the same time. For illustration, the script

```
a = CYL(1.0, 2.0)
b = COPY(a)
SCALE(b, 2.0, 1.0, 0.5)
dim1 = SIZE(b, 1)
dim2 = SIZE(b, 2)
dim3 = SIZE(b, 3)
print "Object dimensions:", dim1, dim2, dim3
```

will print:

```
Object dimensions: 4.0 2.0 1.0
```

4.5 Boolean operations – doing it the wrong way, doing it the right way

Boolean operations can lead to long computations if not done properly. To make them efficient, they should be as much *local* as possible, meaning that they should occur between as *geometrically simple objects* as possible.

We will demonstrate this on an example where we first create a vault and then play a gangster and drill a hole into it. Do not worry about the lenght of the script below because the vault has many small parts. But the fact that there are many parts is important here. We will show that while performing a Boolean operation between the drill and the entire vault is extremely time consuming, drilling into just the one relevant part of the geometry is done quickly. First, let us create the vault:

```
# Define main vault body:
body = BOX(1, 1, 1.2)
interior = BOX(0.86, 0.95, 1.06)
MOVE(interior, 0.07, 0, 0.07)
body = DIFF(body, interior)

# Top part:
top = CHULL([0, 0, 1.2], [1, 0, 1.2], [1, 1, 1.2], \
            [0, 1, 1.2], [0.01, 0.01, 1.21], [0.99, 0.01, 1.21], \
            [0.99, 0.99, 1.21], [0.01, 0.99, 1.12])

shelf = BOX(1, 0.85, 0.02)
MOVE(shelf, 0, 0.1, 0.4)
shelf2 = COPY(shelf1)
MOVE(shelf2, 0, 0, 0.4)

# Front door:
door = BOX(0.84, 0.05, 1.04)
MOVE(door, 0.08, 0, 0.08)
door_frame = BOX(0.86, 0.03, 1.06)
MOVE(door_frame, 0.07, 0.02, 0.07)

# Truncated cone under the handles:
tcone = TCONE(0.14, 0.13, 0.02)
ROTATE(tcone, 90, 1)
MOVE(tcone, 0.5, 0, 0.6)
```

```
# Cylinder that holds the handles:
cyl = CYL(0.07, 0.1)
ROTATE(cyl, 90, 1)
MOVE(cyl, 0.5, 0, 0.6)

# Small truncated cone at the handles:
stcone = TCONE(0.07, 0.06, 0.01)
ROTATE(stcone, 90, 1)
MOVE(stcone, 0.5, -0.1, 0.6)

# Handles:
h1 = CYL(0.022, 0.3)
ts = TCONE(0.022, 0.017, 0.01)
MOVE(ts, 0, 0, 0.3)
h1 = U(h1, ts)
ROTATE(h1, 180./24, 1)
ROTATE(h1, -7*180./24, 2)
h2 = COPY(h1)
ROTATE(h2, -2*180./3, 2)
h3 = COPY(h2)
ROTATE(h3, -2*180./3, 2)
MOVE(h1, 0.5, -0.06, 0.6)
MOVE(h2, 0.5, -0.06, 0.6)
MOVE(h3, 0.5, -0.06, 0.6)

# Bolts:
b1 = CHULL([0.81, 0, 0.19], [0.83, 0, 0.17], [0.97, 0, 0.17], \
[0.97, 0, 0.26], [0.83, 0, 0.26], [0.81, 0, 0.24], [0.815, -0.01, 0.195], \
[0.835, -0.01, 0.175], [0.965, -0.01, 0.175], [0.965, -0.01, 0.255], \
[0.835, -0.01, 0.255], [0.815, -0.01, 0.235])
b2 = COPY(b1)
MOVE(b2, 0, 0, 0.77)

# Bolt cylinder:
bc = CYL(0.015, 0.1, 16)
bc1 = COPY(bc)
MOVE(bc1, 0.925, -0.009, 0.165)
bc2 = COPY(bc1)
MOVE(bc2, 0, 0, 0.77)

# Small bolt sphere:
sbc = SPHERE(0.01, [8, 8])
ROTATE(sbc, 90, 1)
bsc1 = COPY(sbc)
MOVE(sbc1, 0.845, -0.01, 0.195)
sbc2 = COPY(sbc1)
MOVE(sbc2, 0, 0, 0.035)
sbc3 = COPY(sbc1)
MOVE(sbc3, 0, 0, 0.77)
sbc4 = COPY(sbc3)
MOVE(sbc4, 0, 0, 0.035)
```

```
# Shelves:
# Vertical handle:
vh = CYL(0.02, 0.5, 16)
MOVE(vh, 0.15, -0.04, 0.35)

# Vertical handle supports:
vh0 = CYL(0.015, 0.06, 16)
ROTATE(vh0, 90, 1)
MOVE(vh0, 0.15, 0.02, 0.4)
vh1 = COPY(vh0)
MOVE(vh1, 0, 0, 0.4)

# Put everything together:
rest = [top, shelf1, shelf2, door, door_frame, tcone, cyl, stcone,
        h1, h2, h3, b1, b2, bc1, bc2, sbc1, sbc2, sbc3, sbc4, vh, vh0, vh1]
vault = [body, rest]
SHOW(vault)
```

The output is shown in Fig. 88.

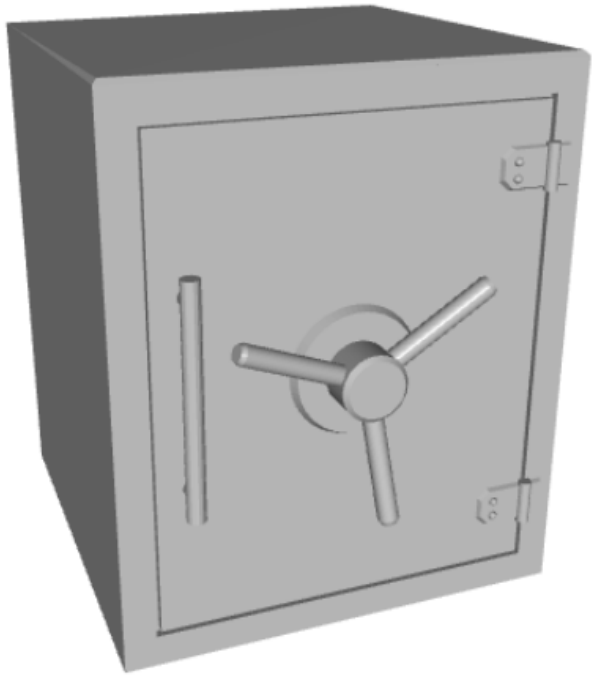


Fig. 88: Vault model.

Now let's define the drill:

```
drill = CYL(0.1, 0.5)
ROTATE(drill, 90, 2)
MOVE(drill, 0.1, 0.5, 0.55)
```

Next let's drill into the vault. First we will do it the wrong way – drilling into the `vault` object that consists of many parts.

```
drilled_vault = DIFF(vault, drill)
SHOW(drilled_vault)
```

You can run this script but it will take a very long time.

The correct way to do this is to perform the Boolean operation between the `drill` and the `body` because the `body` object is much simpler than the complete `vault`:

```
drilled_body = DIFF(body, drill)
SHOW(drilled_body, rest)
```

Now the computation takes just a few milliseconds! The output is shown in Fig.89.

Fig. 89: Vault after subtracting the drill from it.

5 Primer in Python Programming

In order to take full advantage of PLaSM, one should learn more about the Python programming language. This primer will give you a useful basic information. If you are interested in computer programming, NCLab provides a comprehensive course that includes a textbook, interactive exercises, review questions, and solution manual.

5.1 Defining and using variables

We learned before that it is a good habit to make our designs *parametric*. This allows us to adjust them easily any time. Designs that use parameters (variables) rather than raw numbers also tend to be more elegant and less prone to mistakes.

To illustrate this, let us create a 3D model of a pill that is formed by a cylinder with radius `3.0` mm and height `10.0` mm, that ends at both sides with a spherical cap. Moreover the pill has two colors – one half is red and one half blue. Of course this can be done as follows:

```
c1 = CYL(3.0, 5.0)
c2 = COPY(c1)
MOVE(c2, 0, 0, 5.0)
s1 = SPHERE(3.0)
s2 = COPY(s1)
MOVE(s1, 0, 0, 10.0)
half_1 = UNION(c1, s1)
half_2 = UNION(c2, s2)
COLOR(half_1, RED)
COLOR(half_2, BLUE)
SHOW(half_1, half_2)
```

This is a "brute force" approach which does not use any parameters. The result shown in Fig. 90 is correct.
But imagine that we need to change the length of the base cylinder from `10.0` mm to `12.0` mm. We have to make changes on lines 1, 2, and 4 – so there are at least three places where a mistake can easily be made.

The design can be made safer by using variables. Let us define variables `r` and `h` on the first two lines, and then use them instead of raw numbers:

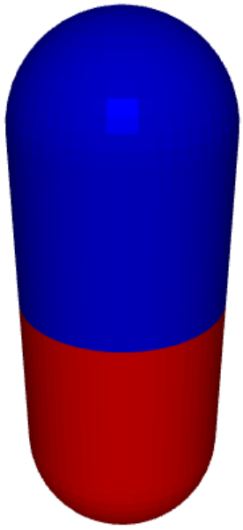


Fig. 90: 3D model of a pill.

```
r = 3.0
h = 10.0

c1 = CYL(r, h / 2.0)
c2 = COPY(c1)
MOVE(c2, 0, 0, h / 2.0)
s1 = SPHERE(r)
s2 = COPY(s1)
MOVE(s2, 0, 0, h)
half_1 = UNION(c1, s1)
half_2 = UNION(c2, s2)
COLOR(half_1, RED)
COLOR(half_2, BLUE)
SHOW(half_1, half_2)
```

Not counting the first two lines, this script has the same length as the previous one. But, any change of the pill's dimensions is now effortless. No need to study the program and make changes at several places - it is enough to change the values of `r` and `h` at the beginning of the program.

5.2 The Numpy library

Python comes with powerful computational libraries that we can use for our PLaSM designs. In particular, Numpy (numerical computations library) contains vast functionality related to mathematical functions and numerical computations. Whenever we want to use some constant such as π or e, or when we want to use some mathematical function such as $\log(x)$, $\sin(x)$ or $\sqrt{x}$, we import it from Numpy:

```
from numpy import pi, e, log, sin, sqrt
```

We can also import everything using the asterisk symbol '`*`':

```
from numpy import *
```

Applications of the Numpy library will be shown in the examples at the end of this section.

5.3 Python lists

Whenever we need to create a set of objects, such as a set of points, we can use a Python list for that. Empty Python list is created using a pair of square brackets:

```
L = []
```

Here, `L` is the name of the list, and any other name would be fine as well, as long as it does not clash with some Python or PLaSM command. We can also create a nonempty list. Say that we need a list that contains the points $(0, 0)$, $(1, 0)$, $(0, 1)$. This can be done as follows:

```
L = [[0, 0], [1, 0], [0, 1]]
```

You are right, each point is a list as well! Sometimes we need to fill a list using some procedure because not all its items are known at the beginning. For this, we can use the command `append()`. For example, the point $(2, 4)$ can be added to the existing list `L` via

```
L.append([2, 4])
```

Python provides a number of useful operations with lists including inserting an item at an arbitrary position, deleting an item from an arbitrary position, reversing and sorting

lists, merging lists, counting occurences of some item, etc. These are beyond our primer but you can find them in the NCLab Python textbook.

5.4 Printing

Control prints are a practical way to check that all values are as they should be. Printing is done via the `print` command. For example, the script

```
L = [[0, 0], [1, 0], [0, 1]]
print L
```

has the following output:

```
[[0, 0], [1, 0], [0, 1]]
```

Printing the value of a single variable is done in the same way, and we can make the output more informative by actually describing what is printed. The script

```
h = 5.0
print "h =", h
```

yields the following output:

```
h = 5.0
```

5.5 Loops

Let us mention one last scripting technique for now, which is to repeat some command or sequence of commands several times. To repeat something `N` times, we use the following construct:

```
for i in range(N):
    do_something
```

The indentation matters as it indicates the body of the loop (set of commands to be repeated). For example, the following script will print a sequence of integers between 0 and 5 (not including 5):

```
for i in range(5):
    print i
```

Output:

```
0
1
2
3
4
```

Once more – notice that the printed sequence ends with `4`, not with `5`. This is a common source of beginner's mistakes.

Let us show one more example of how repetition can be used: We will create a list of points $(0, 0)$, $(1, 0)$, $(2, 0)$, ..., $(N, 0)$ where `N` is some positive integer. Let us write the script initially for `N = 5`, and at the end we will print it:

```
N = 5
L = []
for i in range(N+1):
    L.append([i, 0])
print "L =", L
```

Output:

```
L = [[0, 0], [1, 0], [2, 0], [3, 0], [4, 0], [5, 0]]
```

If we needed to create another list for `N = 100`, the script would remain the same, just the line where `N` is defined would be changed to

```
N = 100
```

Scripting can save lots of work! Let us practice our scripting skills on some simple examples.

5.6 Example 1 - programming a polygon

First we will generate a polygon with N equally-long edges that is inscribed in a circle with radius R. Notice that the points $(R\cos(a), R\sin(a))$ where a is an angle between 0 and 2π, lie on the circle with center $(0, 0)$ and radius R. We begin with importing the sine and cosine fundtions, and the number π from Numpy:

```
from numpy import sin, cos, pi
```

Next create a variable `R` for the radius and `N` for the number of edges:

```
N = 15
R = 5
```

Now the scripting part comes: We create an empty list `points`, and in a loop append the `N` points that lie on the circle:

```
points = []
for i in range(N):
    a = i * 2. * pi / N
    points.append([R * cos(a), R * sin(a)])
poly = CHULL(points)
```

Note the asterisk '`*`' in front of the list `points` on the last line. Its purpose is to "take the square brackets off the list" – decompose the list `points` into a sequence of its entries as expected by the `CHULL` command.

Finally, let us view the polygon `poly` in copper color:

```
COLOR(poly, COPPER)
SHOW(poly)
```

The result is displayed in Fig. 91.

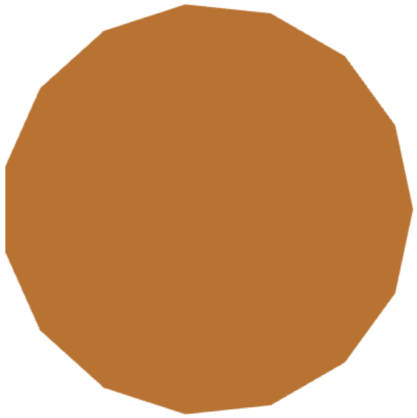


Fig. 91: Equilateral polygon with 15 edges.

5.7 Example 2 - programming a cone

Next let us tweak the above script to construct a cone with radius `R` and height `H`. All we need to do is to add a zero third coordinate to all points and add one more point for the apex. The `CHULL` command will take care of the rest. Since the script is so similar to the last one, it is not necessary to comment all steps in detail:

```
from numpy import sin, cos, pi
R = 5
H = 10
subdiv = 128
```

The only other thing worth mentioning is the parameter `subdiv` that plays the role of `N` from the previous script – in fact this is the optional third parameter of the `CONE` command!

The list of 3D points forming the cone is created as follows:

```
points = []
for i in range(subdiv):
    angle = i * 2. * pi / subdiv
    points.append([R * cos(angle), R * sin(angle), 0])
points.append([0, 0, H])
c = CHULL(points)
```

Finally, let us view the polygon `poly` in copper color:

```
COLOR(c, COPPER)
SHOW(c)
```

The result is displayed in Fig. 92.

5.8 Example 3 - programming arrays of objects

In this example, we will create a field of 400 cylindrical poles. Certainly this is something that we would like to do manually. Here is a script that does it for us:

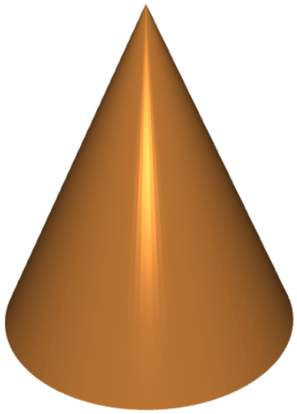


Fig. 92: Cone whose curved surface is approximated with 128 straight segments.

```
# Define master cylinder and N:
c = CYL(0.1, 1, 16)
N = 20
# Duplicate the cylinder N^2 times:
columns = []
for i in range(N):
  for j in range(N):
    d = COPY(c)
    MOVE(d, i, j, 0)
    columns.append(d)
# View the result:
SHOW(columns)
```

The result for $N = 20$, which means 400 columns, is shown in Fig. 93.

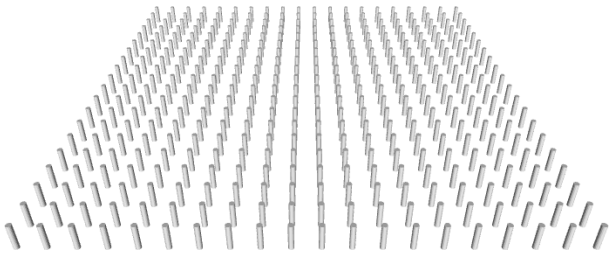


Fig. 93: 400 columns.

More columns? By changing N to 50 and running the script again, we instantly create 2,500 of them:

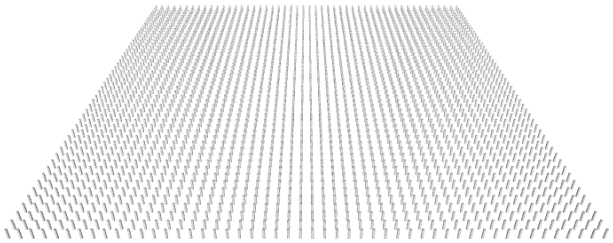


Fig. 94: 2500 columns.

6 Advanced Projects

Objectives:

- Practice your scripting skills.
- Create more complex designs.
- Learn to decompose complicated designs into smaller steps.

6.1 Carrousel

Let us start with something simple - a carrousel created by translating anf rotating a few thin toruses:

```
# Define parameters:
r1 = 0.95
r2 = 1.0
r_mid = 0.5 * (r1 + r2)
h = 0.5 * (r2 - r1)

# Create two master toruses:
t1 = TORUS(r1, r2)
t2 = TORUS(r_mid*cos(PI/6) - h, r_mid*cos(PI/6) + h)

# Create two smaller horizontal toruses:
elevation = r_mid * sin(PI/6)
MOVE(t2, 0, 0, elevation)
t3 = COPY(t2)
MOVE(t3, 0, 0, -2 * elevation)

# Create six vertical toruses:
t4 = COPY(t1)
ROTATE(t4, 90, 1)
t5 = COPY(t4)
ROTATE(t5, 30, 3)
t6 = COPY(t5)
ROTATE(t6, 30, 3)
t7 = COPY(t6)
ROTATE(t7, 30, 3)
t8 = COPY(t7)
ROTATE(t8, 30, 3)
t9 = COPY(t8)
ROTATE(t9, 30, 3)

# Assign colors:
COLOR(t1, RED)
COLOR(t2, GREEN)
COLOR(t3, BLUE)
COLOR(t4, YELLOW)
COLOR(t5, YELLOW)
COLOR(t6, YELLOW)
COLOR(t7, YELLOW)
COLOR(t8, YELLOW)
COLOR(t9, YELLOW)

# Display everything:
SHOW(t1, t2, t3, t4, t5, t6, t7, t8, t9)
```

The outpur is shown in Fig. 95.

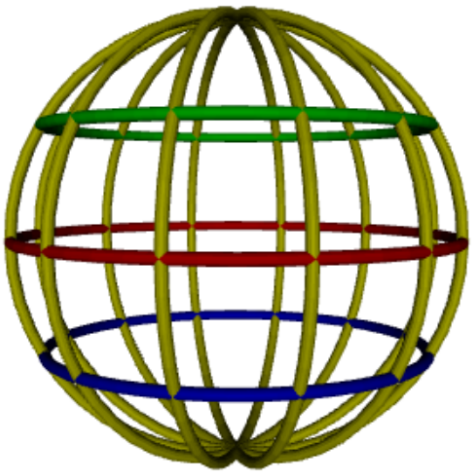


Fig. 95: Carrousel.

6.2 Temple

In this subsection we will build the temple that we have seen in Subsection 1.3:

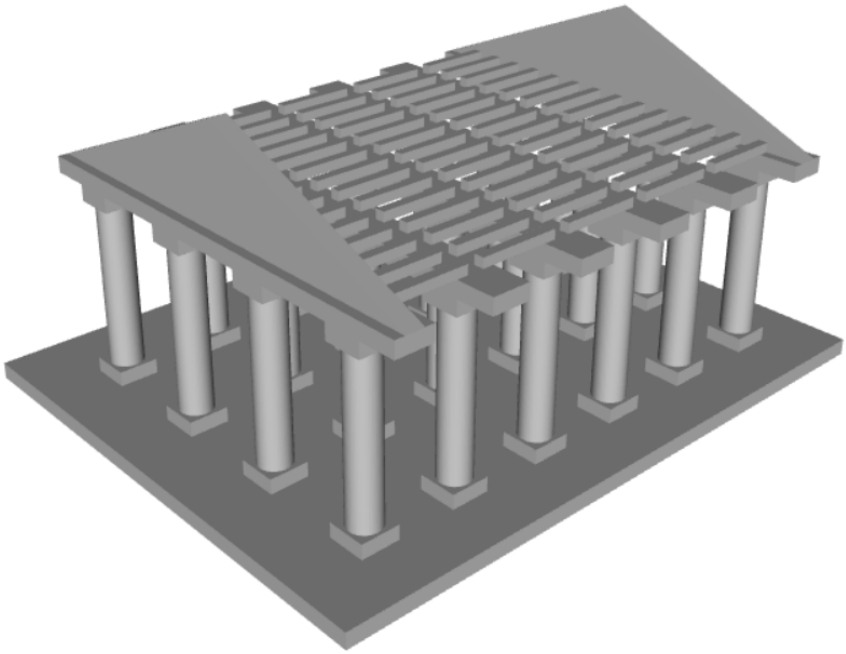


Fig. 96: 3D model of a temple.

Let us begin with defining variables for the radius and height of the columns:

```
# Column radius and height:
r = 1.0
h = 12.0
```

Next let's create one column:

```
# Create one column:
basis = BOX(2*r*1.2, 2*r*1.2, h/12.0)
trunk = CYLINDER(r, (10.0/12.0)*h)
capital = COPY(basis)
beam = COPY(capital)
SCALE(beam, 3, 1, 1)
column = TOP(TOP(TOP(basis, trunk), capital), beam)
column = WELD(column)
```

The column is displayed in Fig. 97.

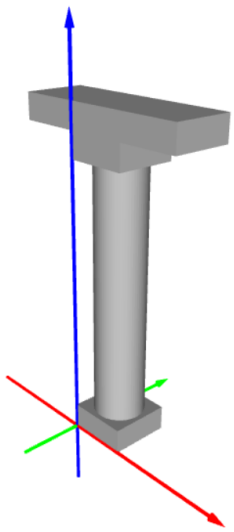


Fig. 97: One column.

In the next step we will form a column row by placing `n = 4` columns next to each other.

```
# Number of columns in one row:
n = 4

# Column row:
colrow = []
for i in range(n):
  newcol = COPY(column)
  MOVE(newcol, i * 6*r*1.2, 0, 0)
  colrow.append(newcol)
```

The column row is displayed in Fig. 98.

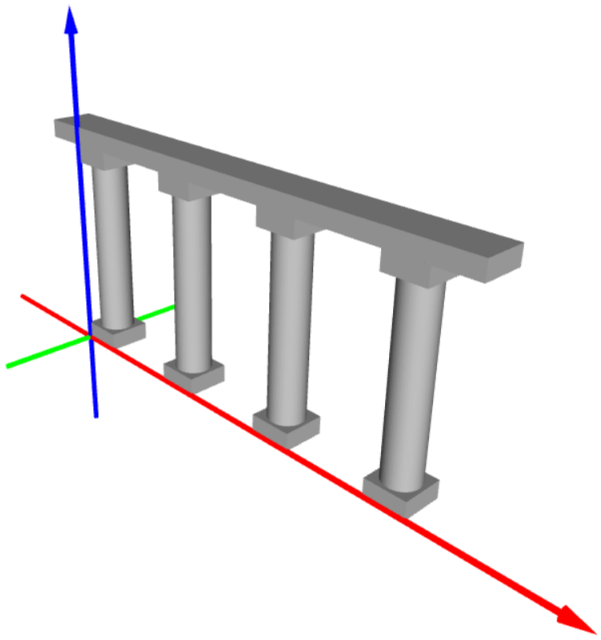


Fig. 98: Column row.

Now we can create the gable and position it on top of the column row:

```
# Create gable:
triangle = TRIANGLE([0, 0], [n*3*2*r*1.2, 0], [n*3*r*1.2, h/2])
prism = PRISM(triangle, r*1.2)
gable = COPY(prism)
ROTATE(gable, 90, 1)

# Portal:
MOVE(gable, -2*r*1.2, 1.5*r*1.2, h/12.0 + (10.0/12.0)*h
     + 2*h/12.0)
portal = STRUCT(colrow, gable)
```

The portal is shown in Fig. 99.

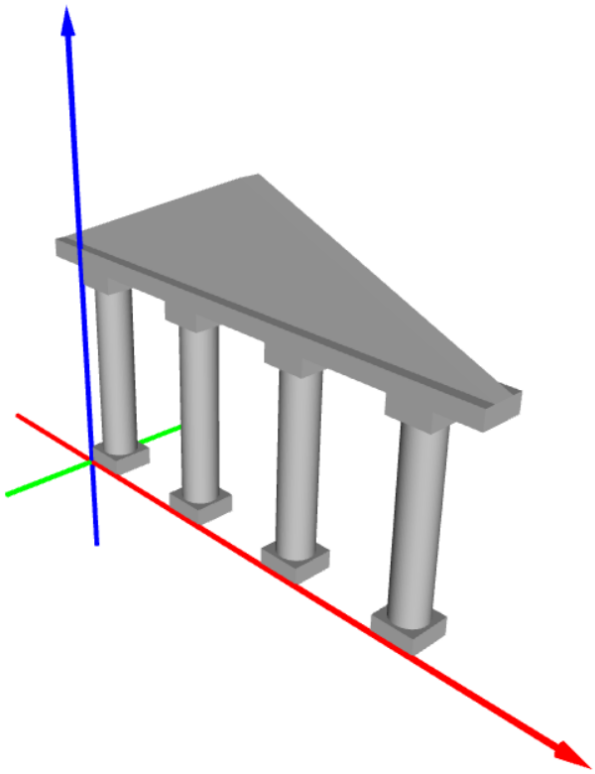


Fig. 99: The portal.

Append `m = 4` inner column rows, and then once more the portal at the end:

```
# Number of inner column rows:
m = 4

# Temple base as a list:
temple_base = [portal]
for i in range(1, m+1):
    newcolrow = COPY(colrow)
    MOVE(newcolrow, 0, 6*i, 0)
    temple_base.append(newcolrow)
newportal = COPY(portal)
MOVE(newportal, 0, 6*(m+1), 0)
temple_base.append(newportal)
```

The temple base is displayed in Fig. 100.

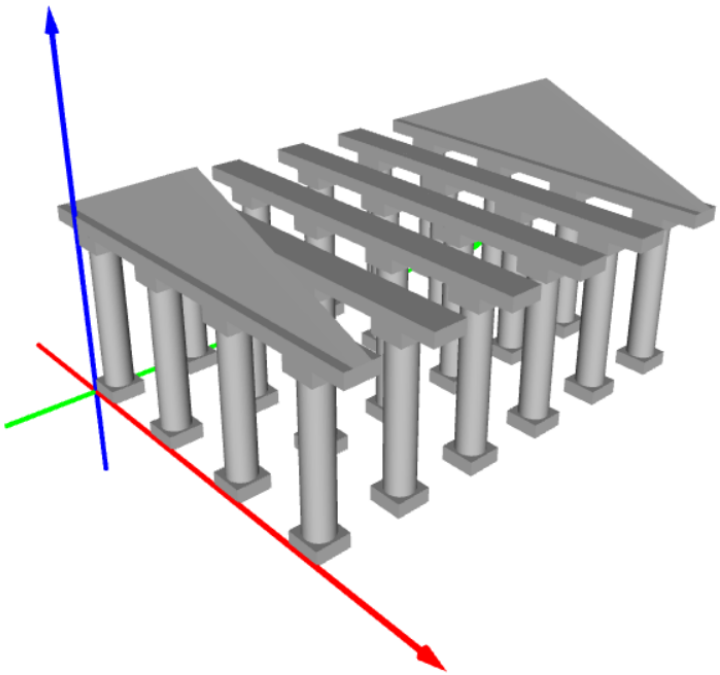


Fig. 100: Temple base.

In the next step we create the foundation plate which is just a box of dimensions 32, 40 and 1 m.

```
# Create a plate for the temple to stand on:
ground = BOX(32, 40, 1)
MOVE(ground, -4.6, -3.8, -0.9)
```

The last component we need are the secondary roof beams. They are created using the `GRID` and `PRODUCT` command that we learned in Subsection 2.25:

```
# Secondary roof beams:
x_intervals = GRID(14 * [0.6, -1.2])
y_intervals = GRID([-0.7] + 5 * [-1, 5])
z_interval = GRID([-13, 0.6])
secondary_beams = POWER(POWER(x_intervals, y_intervals), z_interval)
```

The secondary roof beams are displayed in Fig. 101.

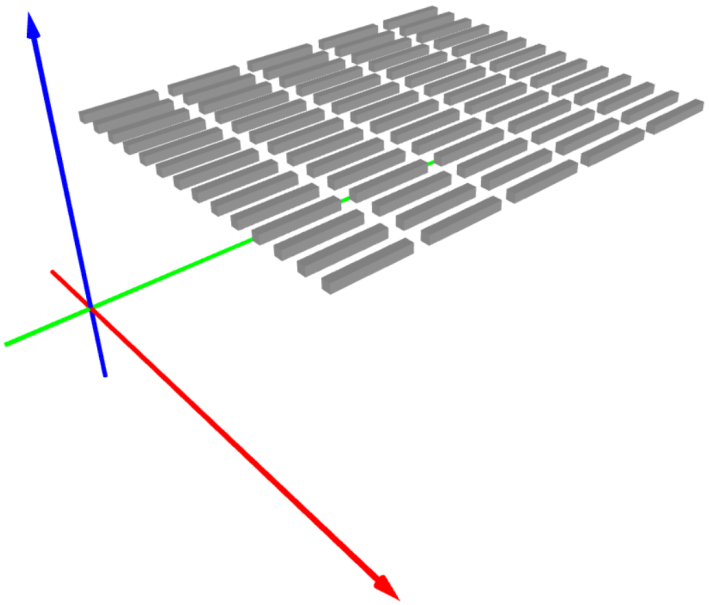


Fig. 101: Secondary roof beams.

After putting everything together,

```
# Put all parts together:
out = UNION(temple_base, secondary_beams, ground)
COLOR(out, WHITE)
```

we obtain the model from Fig. 96.

6.3 Sierpinski fractals

Sierpinski fractals are famous fractal sets that saw the light of the world around 1915 before the Mandelbrot's and Julia's fractals appeared. The *Sierpinski triangle* and *carpet* are shown in Fig. 102.

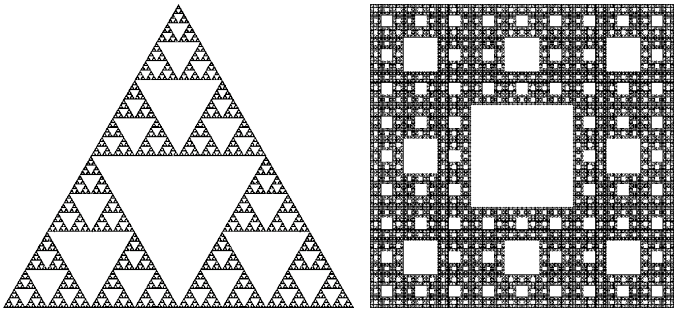


Fig. 102: Sierpinski triangle and carpet.

Both these fractals have their 3D versions, called the *Sierpinski tetrahedron* and *Menger sponge*, respectively. They are easy to find in the Internet.

We will only discuss the triangle here since the other shapes work in a similar way. The triangle fractal is constructed by recursively subtracting smaller equilateral triangles from an original equilateral triangle, as shown in Fig. 103.
This process is infinite in mathematical theory, while in reality we only can perform a finite number of steps. Moreover, the procedure that we just described is good from the mathematical point of view, while it is not the best from the point of view of computer implementation. Why is this?

We learned in Subsection 4.5 that it is not a good idea to perform Boolean operations between complex objects with many faces. This leads to a large number of operations and the computation takes a long time. Subtracting a triangle from a fractal that already has many holes is exactly what we should avoid.

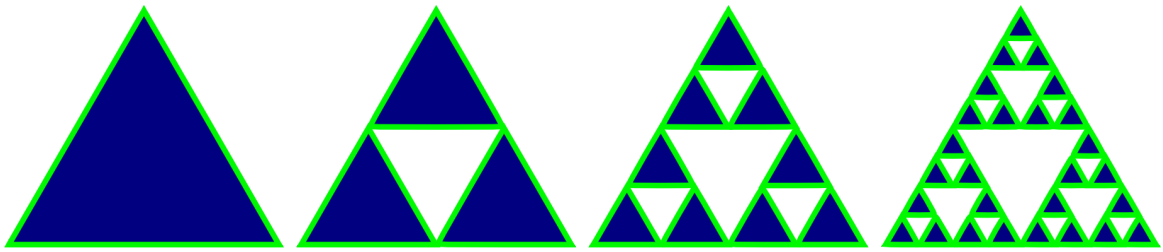


Fig. 103: Construction of the Sierpinski triangle fractal.

Fortunately, there is an elegant way around this which does not use any Boolean operations at all. We start by creating an object `st` (Sierpinski Triangle) that will be just a single equilateral triangle $(0,0)$, $(1,0)$, $(1/2, \sqrt{3/4})$.

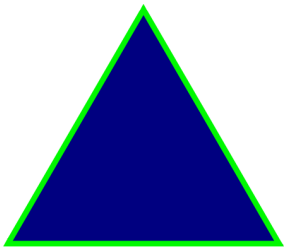


Fig. 104: Initial object `st`.

Then we add to the object `st` two identical objects that are obtained via translating `st` to "east" by the 3D vector $(1,0,0)$ and to "north-east" by the 3D vector $(1/2, \sqrt{3/4}, 0)$.

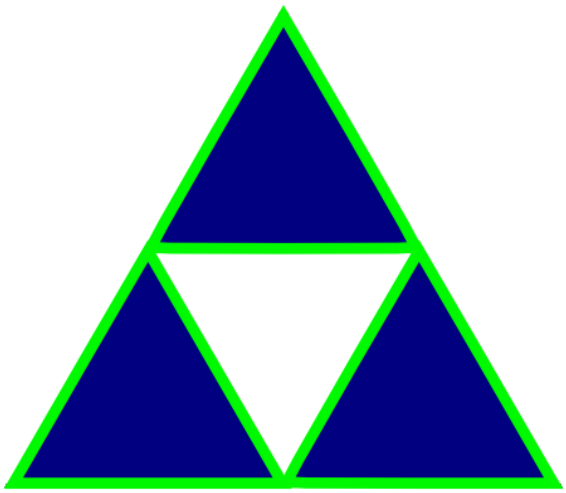


Fig. 105: Object `st` after adding two identical copies to it.

Of course we do not want our fractal to grow infinitely, so we rescale the new object `st` (that comprises three triangles now) by $1/2$ in the x and y directions.

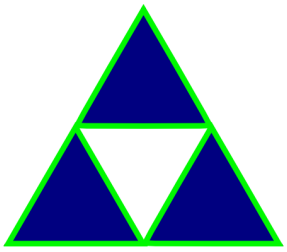


Fig. 106: Object `st` scaled back to original size.

Now it is sufficient to apply to the new object `st` the procedure that we described above – adding two identical copies and rescale – as many times as we want. This will create the Sierpinski Triangle fractal. The program looks as follows:

```
# Number of levels of the Sierpinski triangle:
# Do not go much higher than 6, number of
# operations grows exponentially.
level = 6

# Height of equilateral triangle with unit base:
from numpy import sqrt
h = sqrt(0.75)

# Initial object 'st' is a triangle.
st = TRIANGLE([0, 0], [1, 0], [0.5, h])

# Add two identical copies of 'st'
# and rescale back to (0, 1):
for i in range(level - 1):
    st1 = COPY(st)
    MOVE(st1, 0.5, h)
    st2 = COPY(st)
    MOVE(st2, 1.0, 0.0)
    st = UNION(st, st1, st2)
    SCALE(st, 0.5, 0.5)

# Display in blue color:
COLOR(st, BLUE)
SHOW(st)
```

The resulting geometry for `k = 6` levels is shown in Fig. 107. It is easy to calculate that with k levels, the geometry consists of 3^{k-1} triangles. This means exponential growth – with $k = 6$ there are 243 triangles, with $k = 7$ there are 729, with $k = 8$ there are 2187 etc. Therefore, do not use `k` much higher than 6.

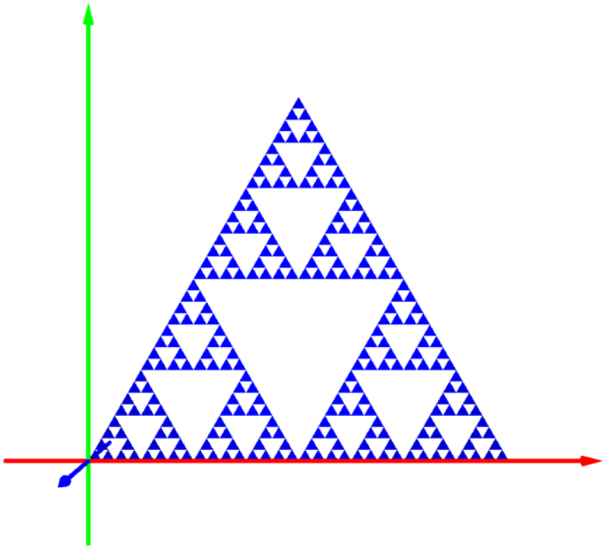


Fig. 107: Resulting geometry for `k = 6`.

6.4 3D gear

In this section we will build the 3D gear that we have seen in Subsection 1.3. In the first step let us define a color along with parameters and measures of one tooth, and create one tooth as a convex hull.

```
# Number of teeth:
N = 18

# Width of the tooth:
x0  = 8
dx1 = 1
dx2 = 2.5

# Height of the tooth:
y0  = 30
dy1 = 0.5
dy2 = 4

# Depth of the tooth:
z1 = 5
z2 = 8
```

```
# Base inner and outer radius, height:
# (Do not change these)
base_rin  = 22
base_rout = 30
base_h    = 32

# Vertices:
pts_base = [[0, 0, 0], [x0, 0, 0], [x0, y0, 0], [0, y0, 0]]
pts_mid  = [[dx1, dy1, z1], [x0 - dx1, dy1, z1], [x0 - dx1, y0 - dy1, z1],
[dx1, y0 - dy1, z1]]
pts_top  = [[dx2, dy2, z2], [x0 - dx2, dy2, z2], [x0 - dx2, y0 - dy2, z2],
[dx2, y0 - dy2, z2]]

# Merging the point lists:
pts = pts_base + pts_mid + pts_top

# Toothe is a convexhull of the points:
t = CHULL(pts)
```

The result is displayed in Fig. 108.

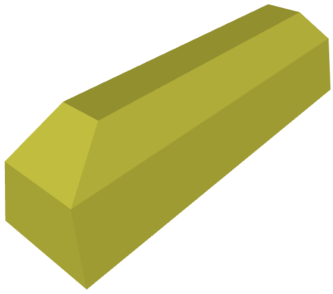


Fig. 108: One tooth of the gear.

Next we create all teeth by translating and replicating the first one:

```
# Rotate and translate the tooth:
ROTATE(t, 90, 1)
ROTATE(t, 90, 3)
MOVE(t, base_rout - 0.3, -x0/2., 0)

# Calculate the angle of rotation:
angle = 360./N

# Create all teeth:
gear = []
for i in range(N):
    ROTATE(t, i * angle, 3)
    gear.append(t)
```

The result is displayed in Fig. 109.

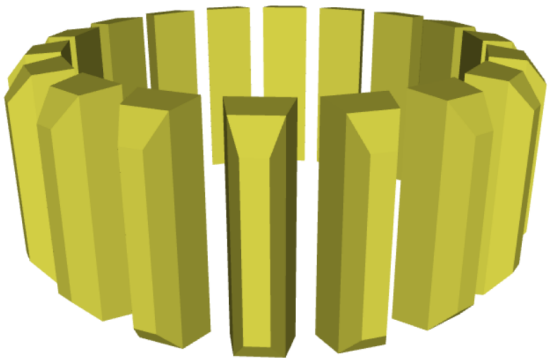


Fig. 109: Teeth are created by replicating the first one.

In the next step we create the base of the gear:

```
# Base:
base = TUBE(base_rin, base_rout, base_h, 128)
dz = (base_h - y0)/2.
MOVE(base, 0, 0, -dz)
```

The result is displayed in Fig. 110.

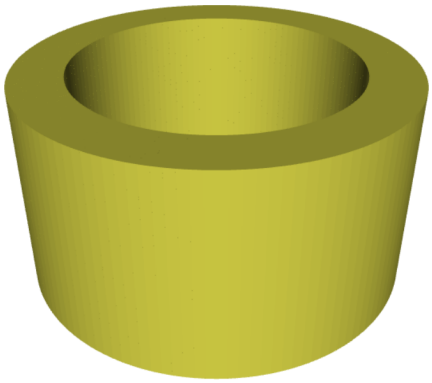


Fig. 110: Base of the gear.

In order to construct the top ring, we use a regular and a truncated cone.

```
# Truncated cone:
tc1 = TCONE(base_rout, base_rout - 1, 1)
MOVE(tc1, 0, 0, base_h)

# Regular cone:
cone = CONE(base_rout, base_rout)
ROTATE(cone, 180, 2)
MOVE(cone, 0, 0, 40)

# Assign colors:
COLOR(cone, RED)
COLOR(tc1, BRASS)

# Display the objects together:
SHOW(tc1, cone)
```

The result is displayed in Fig. 111.

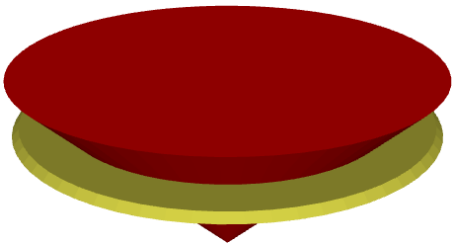


Fig. 111: Getting ready to drill a hole into a truncated cone.

The hole is drilled by subtracting the regular cone from the truncated one.

```
# Difference of the cones:
tc = DIFF(tc1, cone)
```

The result is displayed in Fig. 112.

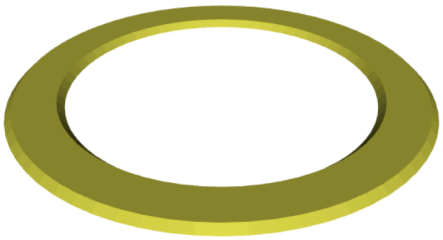


Fig. 112: Top ring.

Next the ring and the teeth are attached to the base.

```
# Put the ring above the base:
base = TOP(base, tc)

# Attach teeth:
gear.append(base)
```

The result is displayed in Fig. 113.

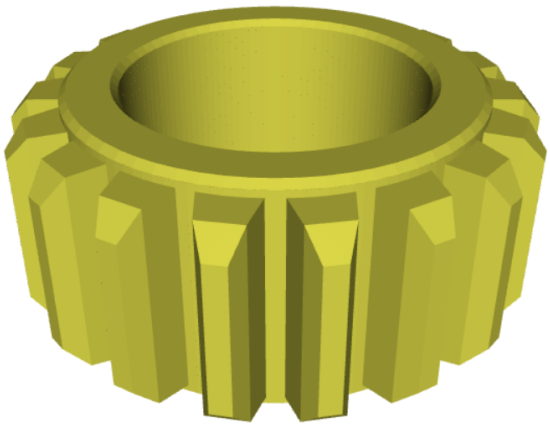


Fig. 113: Attach the ring and the teeth to the base.

Last we need to build the inner part of the gear. First, create a tube and three boxes:

```
# Inner layer thickness:
inner_dr  = 3

# Create inner tube:
inner = TUBE(base_rin - inner_dr, base_rin, base_h * 0.75)
MOVE(inner, 0, 0, base_h * 0.25)

# Define three boxes:
b1 = BOX(50, 10, 35)
MOVE(b1, -25, -5, 0)
b2 = COPY(b1)
ROTATE(b2, 60, 3)
b3 = COPY(b2)
ROTATE(b3, 60, 3)
str = [b1, b2, b3]

# Assign colors:
COLOR(inner, BRASS)
COLOR(str, RED)

# Display the objects together:
SHOW(inner, str)
```

The result is displayed in Fig. 114.

Fig. 114: Base for the inner part and boxes to be subtracted.

Next we subtract the three boxes from the tube.

```
# Subtract all three boxes from the tube:
inner = DIFF(inner, b1, b2, b3)
```

The result is displayed in Fig. 115.

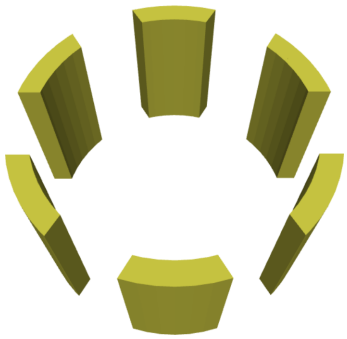


Fig. 115: Result after subtracting the boxes.

Then we subtract the original cone from the inner part.

```
# Subtract cone from inner part:
inner = DIFF(inner, cone)

# Translate the inner part:
MOVE(inner, 0, 0, -dz)

# Assign colors:
COLOR(inner, BRASS)
COLOR(cone, RED)

# Display the objects together:
SHOW(inner, cone)
```

The result is displayed in Figs. 116 and 117.

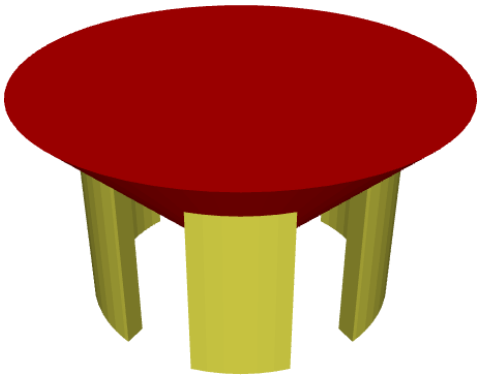


Fig. 116: Cone to be subtracted from the inner part.

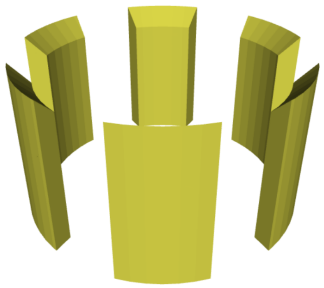


Fig. 117: Result after subtracting the cone.

Finally we attach the inner part to the gear:

```
# Append inner part to gear:
gear.append(inner)

# Display the gear:
COLOR(gear, BRASS)
SHOW(gear)
```

The final geometry is shown in Fig. 118.

Fig. 118: Final geometry.

7 Curves and Curved Surfaces

Objectives:

- Learn the concept of reference domains and reference maps.
- Learn to map curves and surfaces in 3D space.
- Learn about Bézier curves.
- Create Bézier curves and surfaces.
- Create coons patches.
- Create rotational surfaces.
- Learn how to solidify a surface.
- Create ruled surfaces.
- Learn to span surfaces between curves.
- Create cylindrical and conical surfaces.
- Create profile product surfaces.
- Create cubic Hermite surfaces.

PLaSM provides extensive support for curved surfaces including Bézier surfaces, coons patches, rotational surfaces, ruled surfaces, cylindrical and conical surfaces, Hermite surfaces, splines, Cartesian products of curves, etc. These techniques will be discussed in the following, but first let us explain the underlying concepts of *reference domains* and *maps*.

7.1 Reference domains

In a standard way, curves in PLaSM are parameterized from an interval, and surfaces from a rectangle. These domains are called *reference domains*. Reference interval $(0, L)$ that is subdivided into N equally-long pieces is created using the command `INTERVALS`:

```
ref_domain = INTERVALS(L, N)
```

Reference rectangle $(0, L_1) \times (0, L_2)$, where the first and second directions are subdivided into N_1 and N_2 pieces, respectively, can be created using the `PRODUCT` command:

```
dom1 = INTERVALS(L1, N1)
dom2 = INTERVALS(L2, N2)
ref_domain = PRODUCT(dom1, dom2)
```

7.2 Mapping curves

Every curve is the graph of a *function* P that is defined in a reference interval $(0, L)$ and for every value t in this interval it returns a 3D point $P(t)$. The function P is called *reference map*. Let us show a few examples.

Example 1: The map $P(t) = [1 + t, 1 + t, 1 + t]$ which in Python is written as

```
def P(t): return [1 + t, 1 + t, 1 + t]
```

renders for t in $(0, 1)$ a curve that is a straight line connecting the points (1, 1, 1) and (2, 2, 2), as shown in Fig. 119.

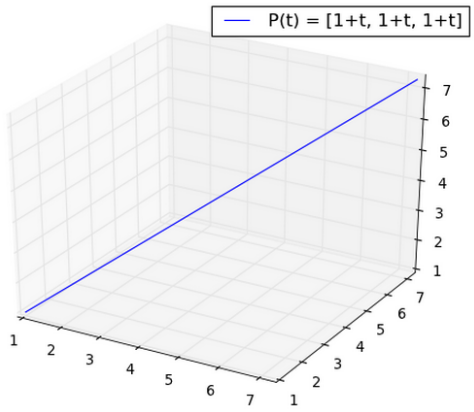


Fig. 119: Graph of the reference map $P(t) = [1 + t, 1 + t, 1 + t]$.

How can we verify that this graph is correct? Insert the limit values $t = 0$ and $t = 1$ into the formula for the map P to see that the end points are (1, 1, 1) and (2, 2, 2), respectively. The fact that the curve is a straight line follows from the linearity of the map P in all three components.

Example 2: The map $P(t) = [\cos(t), \sin(t), 0]$ whose Python form is

```
def P(t): return [cos(t), sin(t), 0]
```

maps the reference interval $(0, 2\pi)$ to a circle in the xy-plane, as shown in Fig. 120.

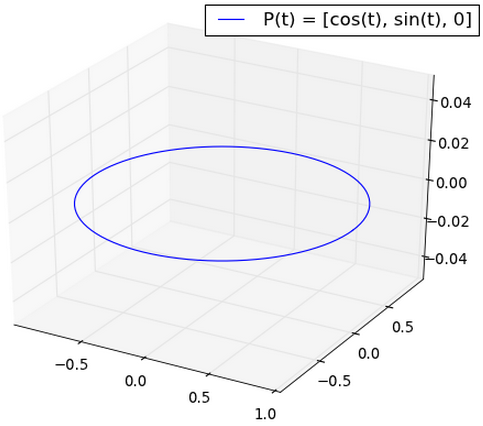


Fig. 120: Graph of the reference map $P(t) = [\cos(t), \sin(t), 0]$.

Example 3: The map $P(t) = [0, \cos(t), \sin(t)]$ a.k.a.

```
def P(t): return [0, cos(t), sin(t)]
```

maps the reference interval $(0, 2\pi)$ to a circle in the yz-plane, as shown in Fig. 121.

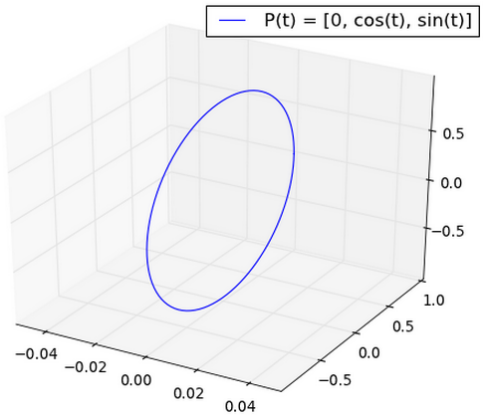


Fig. 121: Graph of the reference map $P(t) = [0, \cos(t), \sin(t)]$.

Example 4: The map

$$P(t) = [\cos(t), \sin(t), t]$$

or

```
def map(t):
    return [cos(t), sin(t), t]
```

maps the reference interval $(0, 2\pi)$ to a spiral whose xy-plane projection is the unit circle and in the z-direction spans the interval $(0, 2\pi)$, as shown in Fig. 122.

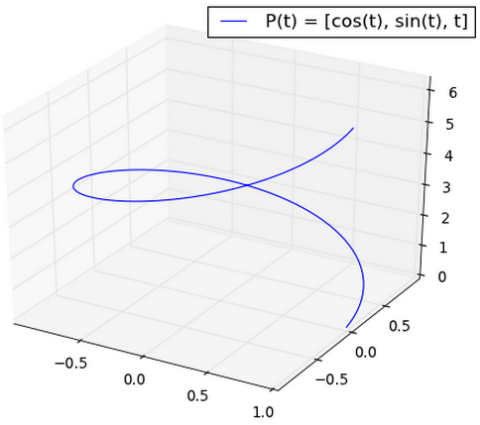


Fig. 122: Graph of the reference map $P(t) = [\cos(t), \sin(t), t]$.

7.3 Mapping surfaces

Now that the reader understands the parameterization of curves, surfaces will follow easily. In fact the only difference is that the reference map $P(t_1, t_2)$ is a function of two variables – it still returns 3D points. Let us show a few examples.

Example 1: The map

$$P(t_1, t_2) = [t_1, t_2, t_1 + t_2]$$

whose code is

```
def P(t1, t2):
    return [t1, t2, t1 + t2]
```

maps the reference square $(0, 1) \times (0, 1)$ to a planar surface that spans four points $(0, 0, 0)$, $(1, 0, 1)$, $(0, 1, 1)$ and $(1, 1, 2)$. No subdivision in any direction was applied to

the reference square to generate the wireframe plot shown in Fig. 123. In other words, only the four vertices of the reference domain were transformed using the map, and the resulting 3D points were interpolated linearly. This was enough in this case since the resulting surface is linear.

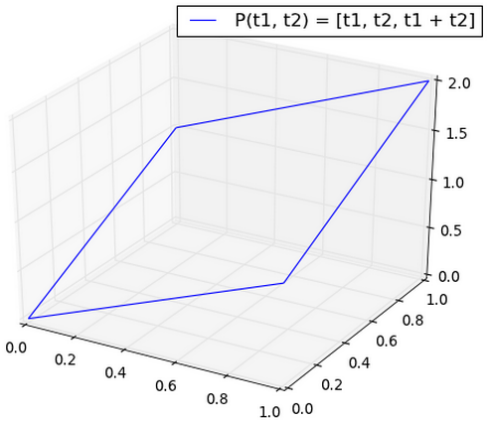


Fig. 123: Graph of the reference map $P(t_1, t_2) = [t_1, t_2, t_1 + t_2]$.

Example 2: Next we render a saddle surface using the reference square $(-1, 1) \times (-1, 1)$. The map is defined as

$$P(t_1, t_2) = [t_1, t_2, t_1 t_2]$$

or

```
def P(t1, t2):
    return [t1, t2, t1*t2]
```

The reference square was subdivided into 10 intervals in each direction, i.e., into 100 square cells. This means that 11^2 points were transformed via the map $P(t_1, t_2)$. The corresponding wireframe plot with 100 linearly interpolated cells is shown in Fig. 124.

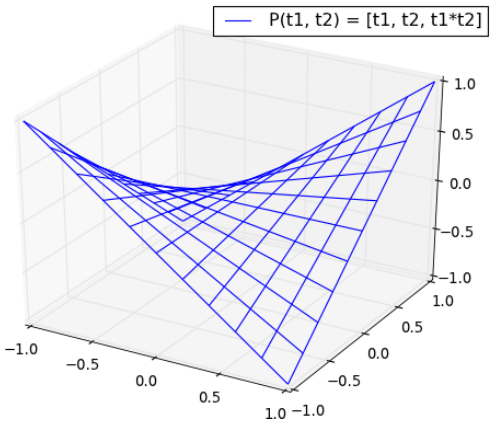


Fig. 124: Graph of the reference map $P(t_1, t_2) = [t_1, t_2, t_1 t_2]$.

Example 3: Now let us render a cylindrical surface with radius 1 and height 1 using the reference rectangle $(0, 2\pi) \times (0, 1)$. The interval $(0, 2\pi)$ represents angular direction and $(0, 1)$ the z-direction. The map is defined as

$$P(a, z) = [\cos(a), \sin(a), z]$$

or

```
def P(a, z):
    return [cos(a), sin(a), z]
```

The reference domain was subdivided into 32 equally-long subintervals in the angular direction, and it was not subdivided in the z-direction. Hence the wireframe plot shown in Fig. 125 consists of 32 linear surfaces.

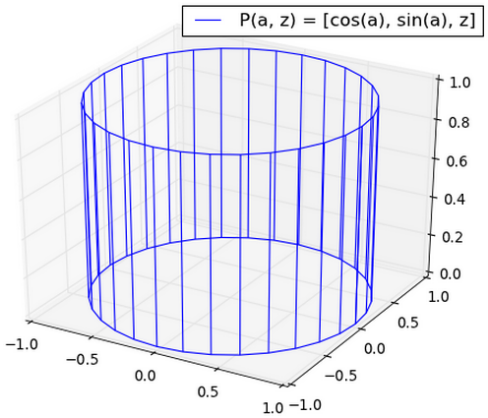


Fig. 125: Graph of the reference map $P(a, z) = [\cos(a), \sin(a), z]$.

In fact, PLaSM uses the same map to render cylindrical surfaces, and that's is where the optional subdivision parameter in the `CYLINDER` command comes from. In PLaSM, its default value is 64.

7.4 Three ways to map a sphere

Usually there is more than one way to map a given surface. The maps will differ in their properties and some of them will be better than the others. Let us illustrate this using the spherical surface as an example.

Map 1: Let us begin with the simplest map based on parameterizing the angular and radial directions. The reference rectangle is $(0, 2\pi) \times (0, 1)$. where $(0, 2\pi)$ represents the angle and $(0, 1)$ the radius. The map is given by the formula

$$P(a, r) = \left[r\cos(a), r\sin(a), \sqrt{1 - r^2}\right]$$

or

```
def P(a, r):
    return [r*cos(a), r*sin(a), sqrt(1 - r**2)]
```

We subdivide the reference rectangle into 32 pieces in the angular direction and into 10 pieces in the radial one. This means that the piecewise linear wireframe plot shown in Fig. 126 has 320 cells.

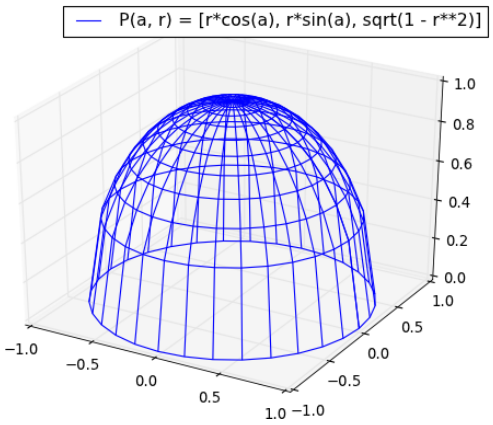


Fig. 126: Graph of the reference map $P(a, r) = \left[r\cos(a), r\sin(a), \sqrt{1-r^2}\right]$.

Notice that many points are accumulated near the pole while the lower portion of the surface is approximated rather crudely.

Map 2: This map uses the angle and the z-direction. The reference rectangle is formally the same as before, $(0, 2\pi) \times (0, 1)$, but the interval $(0, 1)$ has a different meaning. Also the map is different,

$$P(a, z) = \left[\sqrt{1-z^2}\cos(a), \sqrt{1-z^2}\sin(a), z\right]$$

or

```
def P(a, z):
    return [sqrt(1 - z**2)*cos(a), sqrt(1 - z**2)*sin(a), z]
```

The wireframe plot shown in Fig. 127 is based on precisely the same subdivision of the reference rectangle as in the previous case.

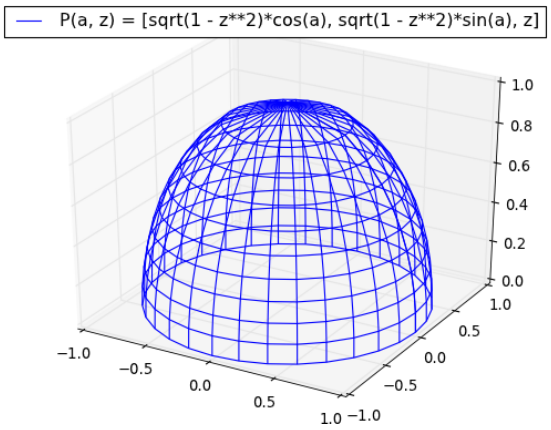


Fig. 127: Graph of the reference map $P(a,z) = \left[\sqrt{1-z^2}\cos(a), \sqrt{1-z^2}\sin(a), z\right]$.

The reader can see that while the lower portion of the surface is approximated better compared to the previous case, the accuracy suffers in the vicinity of the pole.

Map 3: The last map uses two angular directions a and b and a different reference rectangle $(0,2\pi)\times(0,\pi/2)$. The map is defined as $P(a,b) = [\cos(a)\cos(b), \sin(a)\cos(b), \sin(b)]$ or

```
def P(a, z): return [cos(a)*cos(b), sin(a)*cos(b), sin(b)]
```

The wireframe plot shown in Fig. 128 is also based on a 32×10 subdivision of the reference rectangle which results into the same number of 320 linear cells that we had in the previous two examples. However, the reader can see that this map yields clearly the best approximation.

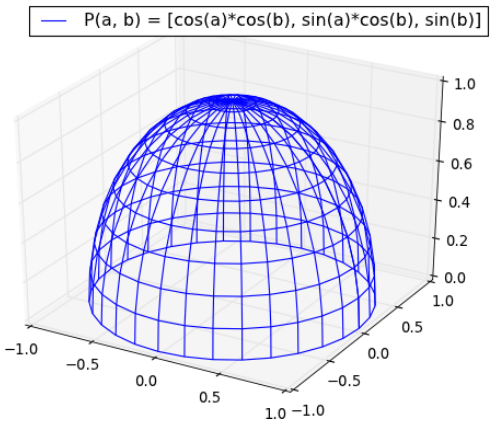


Fig. 128: Graph of the reference map $P(a,b) = [\cos(a)\cos(b), \sin(a)\cos(b), \sin(b)]$.

Therefore PLaSM uses precisely this map to render spherical surfaces. In fact PLaSM uses a reference rectangle $(0, 2\pi) \times (-\pi/2, \pi/2)$ to cover the entire sphere. The default subdivision for the spherical surface in PLaSM is 48×24 which corresponds to

```
s = SPHERE(R, [24, 48])
```

(the order of the two subdivisions is switched).

7.5 Primer on Bézier curves

Bézier curves are the most widely used curves in engineering design. Let us begin with the linear case. Linear Bézier curve is a straight line that connects two control points P_1 and P_2. All Bézier curves are parameterized from the interval $[0, 1]$, so in the linear case the exact mathematical definition is

$$B(t) = P_1 + t(P_2 - P_1)$$

You can easily check that the curve starts at P_1 (substitute $t = 0$ to the above formula), and that it ends at P_2 (substitute there $t = 1$).

Quadratic Bézier curves are defined using three control points P_1, P_2 and P_3. Again they are parameterized from $[0, 1]$, and their definition is

$$B(t) = (1-t)^2 P_1 + 2(1-t)tP_2 + t^2 P_3.$$

Cubic Bézier curves are defined using four control points P_1, P_2, P_3 and P_4. They are parameterized from $[0, 1]$ as usual, and their definition is

$$B(t) = (1-t)^3 P_1 + 3(1-t)^2 tP_2 + 3(1-t)t^2 P_3 + t^3 P_4.$$

7.6 Drawing Bezier curves

The command `DRAWBEZIER2D` takes a list of 2D points in the xy-plane, and it creates an object that can be visualized with the `SHOW` command. Besides the list of points, the `DRAWBEZIER2D` command has the following optional parameters: thickness of the curve (default value 0.02), radius of spheres that represent the points (default value 0.1), color of the curve and end points (default BLACK), color of the interior points (default BLUE), and two divisions (default 32 and 1). Here is an example of a curve given by two points (which is a straight line):

```
# List of XY plane points:
point_list = [[3, 0], [0, 6]]

# Show the Bezier curve:
c = DRAWBEZIER2D(point_list)
SHOW(c)
```

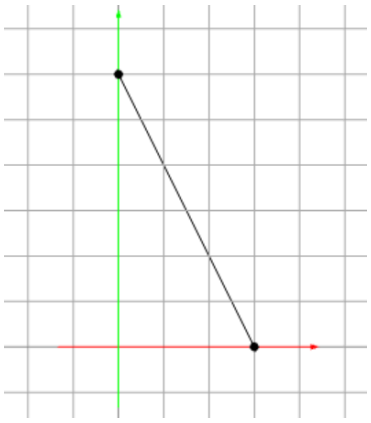


Fig. 129: Bezier curve (straight line) given by two points [3, 0], [0, 6].

Another example of a curve given by three points (which is a parabola):

```
# List of XY plane points:
point_list = [[3, 0], [3,4], [0, 6]]

# Show the Bezier curve:
c = DRAWBEZIER2D(point_list)
SHOW(c)
```

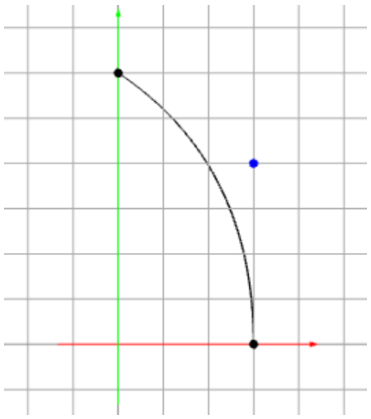


Fig. 130: Bezier curve (parabola) given by three points [3, 0], [3, 4], [0, 6].

7.7 2D object with one straight and one curved Bézier edges

For starters, let's create a 2D quadrilateral with one linear and one cubic Bézier edges. Bézier curves are created using the commands `BEZIER` and `BEZIER2` (that can be abbreviated as `BE` and `BE2`). The former is mapped on the x (horizontal) coordinate in the reference rectangle, the latter on the y (vertical) coordinate. For completeness, there is also `BEZIER3` that is mapped on the z coordinate of a reference 3D box. But we are going to stay with 2D surfaces for now.

First let us create two Bézier curves, one linear and one cubic:

```
# Linear Bezier curve:
c1 = BEZIER([0, 0], [0, 4])

# Cubic Bezier curve:
c2 = BEZIER([2, 0], [4, 1], [0, 2], [1, 3])
```

These are the left vertical edge, and the right (curved) vertical edge in Fig. 131.

The surface between these two curves needs to be treated as any other curved surface, although in this case it is a subset of the XY plane. It needs to be mapped on a reference rectangle. For simplicity we will use the unit square.

Fig. 131: 2D object with one straight and one curved Bézier edge.

```
# Reference domain with 30 x 30 subdivision:
refdomain = UNITSQUARE(30, 30)

# The 2D surface:
surf = BEZIER2(c1, c2)
out = MAP(refdomain, surf)
```

Notice that `BEZIER2` has `c1, c2` as arguments. For any fixed value of x between x_{min} and x_{max} these become just a pair of points. Since there are just two of them, the `BEZIER2` curve connecting them is linear in y. When $x = x_{min}$, then it is a straight line connecting the beginning points of the curves `c1` and `c2`. When $x = x_{max}$, then the `BEZIER2` curve is a straight line connecting their end points. Similarly, the `BEZIER2` curve is a linear function of y, connecting just two points in the XY plane, for any fixed value of x between x_{min} and x_{max}. The output is displayed in Fig. 131.

Remarks:

– As we already saw in a different context before, a point list such as `L = [[0, 0, 0], [0, 4, 0]]` can be passed into the `BEZIER` and `BEZIER2` functions as well, with an asterisk in front of it, i.e., as `BEZIER(*L)` or `BEZIER2(*L)`.

– Notice that the commands accepts general 3D points, they do not have to lie in the xy-plane as they are in this example.

7.8 2D object with two curved Bézier edges

This example is very similar to the last one, and we are showing it only to illustrate that both the Bézier edges can be curves. Concretely, the curve `c1` is made parabolic by adding one more control point to it.

```
# Quadratic Bezier curve:
c1 = BEZIER([0, 0], [-1, 1.5], [0, 4])

# Cubic Bezier curve:
c2 = BEZIER([2, 0], [4, 1], [0, 2], [1, 3])

# Reference domain with 30 times 30 subdivision:
refdomain = UNITSQUARE(30, 30)

# The 2D surface:
surf = BEZIER2(c1, c2)
out = MAP(refdomain, surf)
```

The output is displayed in Fig. 132.

Fig. 132: 2D object with two curved Bézier edges.

7.9 3D surface defined via four Bézier curves

Next let us construct a Bézier surface defined via four Bézier curves.

```
# Bezier curves:
c1 = BEZIER([0, 0, 0], [5, -2, -5], [10, 0, 0])
c2 = BEZIER([0, 2, 0], [8, 3, 0], [9, 2, 0])
c3 = BEZIER([0, 4, 1], [7, 5, -1], [8, 5, 1], [12, 4, 0])
c4 = BEZIER([0, 6, 0], [9, 6, 3], [10, 6, -1])

# Reference domain with 30 times 30 subdivision:
refdomain = UNITSQUARE(30, 30)

# The surface:
surf = BEZIER2(c1, c2, c3, c4)
out = MAP(refdomain, surf)
```

All four curves are parameterized by x (the horizontal variable) on the reference rectangle. Three of them are parabolas and one is cubic (that's `c3`). All of them are genuinely 3D, with the exception of `c2` which lies in the XY plane. Since there are four of them, in this case `BEZIER2` is a cubic function (of y on the reference rectangle). The output is displayed in Fig. 133.

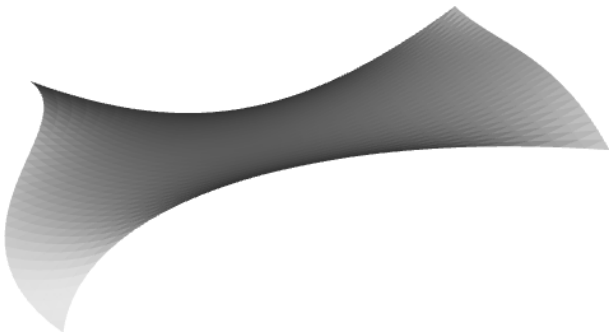


Fig. 133: 3D surface defined via four curved Bézier edges.

7.10 Coons patch

Coons patch is a technique that forms a Bézier patch from four Bézier edges connected by their end points. The center control points are calculated by a blend of two linear

interpolations and one bilinear interpolation.

```
u1 = BEZIER([0, 4, 0], [2.5, 3, 6], [5, 0, -6], [7.5, 0, 6],
[10, 0, 0])
u2 = BEZIER([0, 6, 0], [2.5, 7, 6], [5, 10, -6], [7.5, 10, 6],
[10, 10, 0])
v1 = BEZIER2([0, 0, 0], [-3, 3, 3], [-3, 7, 3], [0, 10, 0])
v2 = BEZIER2([10, 0, 0], [15, 6, 0], [10, 10, 0])

# The coons patch:
out = COONSPATCH(u1, u2, v1, v2)
```

The reference domain is the unit square, divided by default into 32x32 intervals. These divisions can be changed using a pair of optional parameters that comes after the four curves. The curves `u1` and `u2` are mapped on the bottom and top horizontal edges of the reference unit square. The curves `v1` and `v2` are mapped on the left and right vertical edges. The output is displayed in Fig. 134.

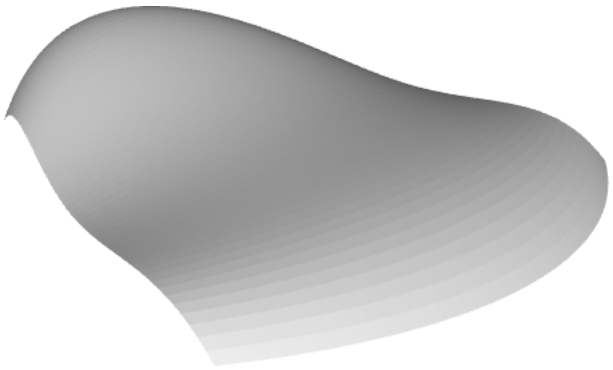


Fig. 134: Coons patch.

7.11 Rotational surface

Rotational surfaces can be created using the `ROTATIONALSURFACE` command (possibly abbreviated as `ROSURF`). The command takes a list of 2D points that define a Bezier curve in the first quadrant of the xy-plane. The curve is then rotated about the y-axis:

```
# List of points in the first quadrant of
# the XY plane, that define a Bezier curve:
points = [[0, 0], [2, 0], [3, 4]]

# Create the rotational surface:
out = ROSURF(points, 360, 32, 64)
```

The last three parameters are optional - an angle of evolution greater than zero and less or equal to 360 degrees, and two divisions. The output of the above code is displayed in Fig. 135.

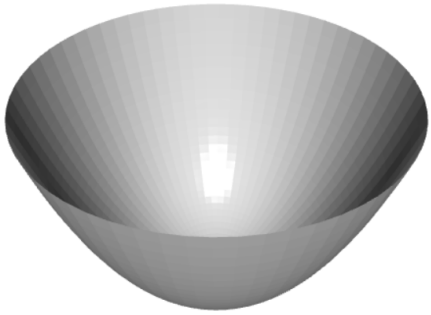


Fig. 135: Rotational surface created using a quadratic Bézier curve.

7.12 Solidifying a surface

Surfaces can be solidified using the command `JOIN`. The result of this operation will always be a convex hull. Adding to the last script:

```
# Solidify the surface:
out = JOIN(out)
```

This operation may take a while since PLaSM creates a new set of volume maps from the existing set of surface maps. The output is displayed in Fig. 136.

7.13 Rotational solid

Rotational solids can be created using the `ROTATIONALSOLID` command (possibly abbreviated as `ROSOL`). The usage is analogous to `ROTATIONALSURFACE`, except that the result is a 3D volumetric object instead of a surface. Also, there is one additional parameter `rmin` that comes after the angle of evolution – with `rmin = 0` the solid will

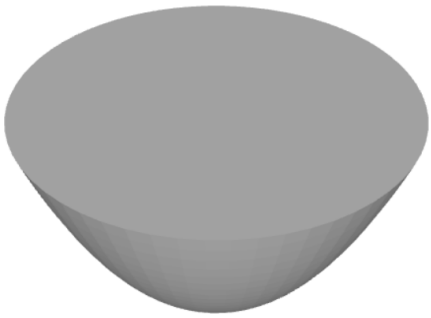


Fig. 136: Solidifying the surface from Fig. 135.

extend all the way from the axis of rotation to the curve in the radial direction. If `rmin > 0` then there will be a cylindrical void about the axis of rotation:

```
# List of points in the first quadrant of the XY plane,
# that defines a Bezier curve:
points = [[0, 0, 0], [2, 0, 0], [3, 0, 1], [0, 2, 0],
          [3, 0, 4]]

# Create a rotational solid about the y-axis:
out = ROSOL(points, 360, 0, 32, 64)
```

The last four parameters are optional - an angle of evolution greater than zero and less or equal to 360 degrees, `rmin`, and two divisions. The output of the above code is displayed in Fig. 137.

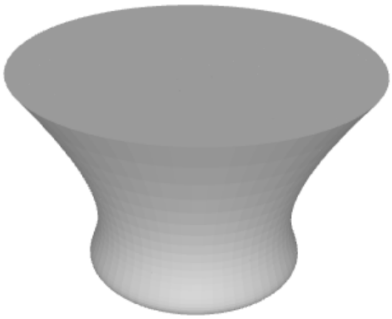


Fig. 137: Creating a rotational solid.

7.14 Ruled surface - an introduction

In geometry, a surface s is called *ruled* (or *scroll*) if it is "formed by straight lines". More precisely, if through every point of s passes a straight line that lies in s. The most familiar examples are the plane, cylinder, and cone. The standard definition of the ruled surface is

$$s(x, y) = p(x, y) + yq(x, y)$$

where $p(x, y) = p(x)$ and $q(x, y) = q(x)$ are maps defined in a reference rectangle $A \times B$. None of them depends on the y-coordinate, just on x. For every x in A, $p(x, y) = p(x)$ returns a 3D point on the surface s. The map $q(x, y) = q(x)$ returns for every x in A a direction vector (in the form of a 3D point). This is the direction of the straight line that lies on the surface s. The values of y are in the interval B.

In the first example, the map $p(x, y) = p(x)$ is defined as

$$p(x, y) = (x, 0, 0)$$

and thus the curve lying on s is a straight line connecting the points $(0, 0, 0)$ and $(1, 0, 0)$.

```
def p(point):
    x = point[0]
    return [x, 0, 0]
```

The map $q(x, y) = q(x)$ is given by

$$q(x, y) = (0, 1, x).$$

and it defines a direction vector that changes gradually from (0, 1, 0) to (0, 1, 1) as x goes from 0 to 1.

```
def q(point):
    x = point[0]
    return [0, 1, x]
```

Reference domain is the unit square covered with a 32×32 division:

```
# Unit square covered with a 32x32 Cartesian grid:
refdomain = UNITSQUARE(32, 32)
```

The RULEDSURFACE command takes the two maps p, q as arguments. It can be abbreviated as RUSU:

```
# Creating the ruled surface:
surf = RUSU(p, q)
out = MAP(refdomain, surf)
```

The output is displayed in Fig. 138.

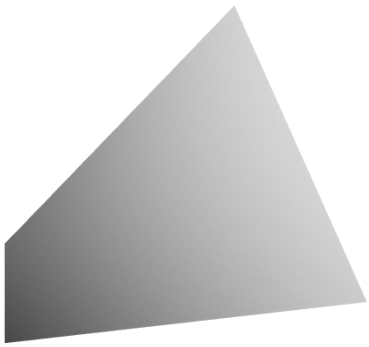


Fig. 138: Ruled surface given by the maps p and q.

7.15 Ruled surface - spiral

In this example the map $p(x, y) = p(x)$ moves the 3D point along the z-axis and the direction vector $q(x, y) = q(x)$ is horizontal and rotates. The resulting surface S is a spiral.

```
# Import sine and cosine from Numpy:
from numpy import sin, cos

def p(point):
    x = point[0]
    return [0, 0, 0.5*x]

def q(point):
    x = point[0]
    return [cos(x), sin(x), 0]

# Reference domain:
refdomain = REFDOMAIN(8*PI, 5, 128, 16)

# Creating the ruled surface:
surf = RUSU(p, q)
out = MAP(refdomain, surf)
```

The output is displayed in Fig. 139.

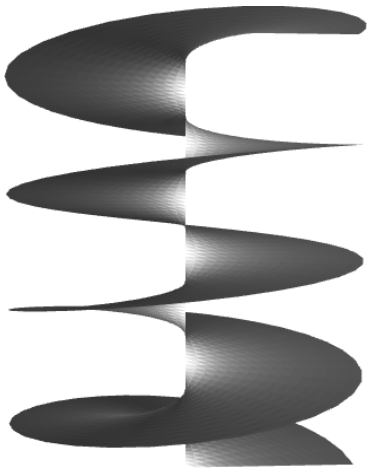


Fig. 139: Spiral as a ruled surface.

7.16 Ruled surface - straight cylinder

In this example the map $p(x, y) = p(x)$ moves the 3D point on a circle that lies in the xy-plane, and the vector $q(x, y) = q(x)$ it a unit vector in the vertical direction. Hence the resulting ruled surface is a cylinder.

```
# Import sine and cosine from Numpy:
from numpy import sin, cos

# Cylinder radius:
r = 3.0

# Cylinder height:
h = 10.0

def p(point):
    x = point[0]
    return [r*cos(x), r*sin(x), 0]

def q(point):
    x = point[0]
    return [0, 0, 1]

# Reference domain:
refdomain = REFDOMAIN(2*PI, h, 64, 1)

# Creating the ruled surface:
surf = RUSU(p, q)
out = MAP(refdomain, surf)
```

The output is displayed in Fig. 140.

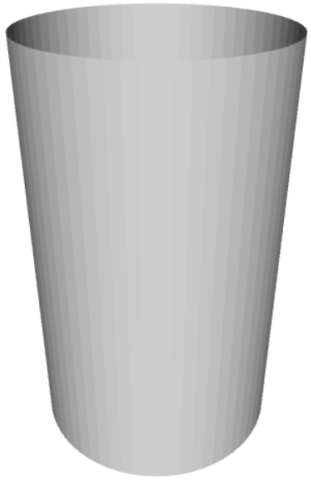


Fig. 140: Cylinder as a ruled surface.

7.17 Ruled surface - curved cylinder

This example is similar to the last one, except that the directional vector $q(x, y) = q(x)$ is not vertical. Instead, it points to a point on the upper circle of the cylinder that has an angular shift α. The script below uses a shift of 0.7π. When increased towards π, the curved surface becomes thinner in the middle portion.

```
# Import since and cosine from Numpy:
from numpy import sin, cos

# Cylinder radius:
r = 3.0

# Cylinder height:
h = 10.0

# Angular shift:
alpha = 0.7 * PI

def p(point):
    x = point[0]
    return [r*cos(x), r*sin(x), 0]
```

```
def q(point):
    x = point[0]
    return [r/h * (cos(x + alpha) - cos(x)),
            r/h * (sin(x + alpha) - sin(x)), 1]

# Reference domain::
refdomain = REFDOMAIN(2*PI, h, 64, 32)

# Creating the ruled surface:
surf = RUSU(p, q)
out = MAP(refdomain, surf)
```

The output is displayed in Fig. 141.

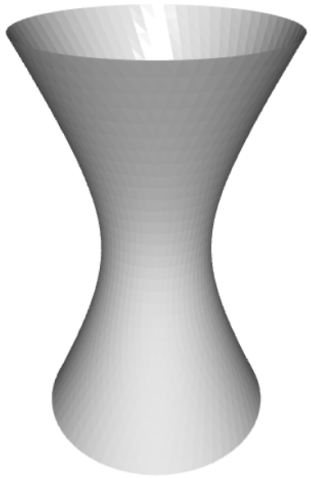


Fig. 141: Curved cylinder.

7.18 Ruled surface - spanning arbitrary 3D curves

This is one of the most useful application of ruled surfaces. We will show two examples. In the first one we consider a $2\pi \times 2\pi$ square, define a sine function and a parabola on the opposite edges, and connect them via a ruled surface:

```
# Import sine from Numpy:
from numpy import sin

# Define two 3D curves:
def c1(x):
    return [x, 0, sin(x)]

def c2(x):
    return [x, 2*PI, (x/PI - 1)**2]

def p(point):
    x = point[0]
    return c1(x)

# Return a vector pointing from one curve to the other:
def q(point):
    x = point[0]
    return [c2(x)[0] - c1(x)[0], c2(x)[1] - c1(x)[1],
            c2(x)[2] - c1(x)[2]]

# Reference domain:
refdomain = REFDOMAIN(2*PI, 1, 64, 1)

# Creating the ruled surface:
surf = RUSU(p, q)
out = MAP(refdomain, surf)
```

The output is displayed in Fig. 142.

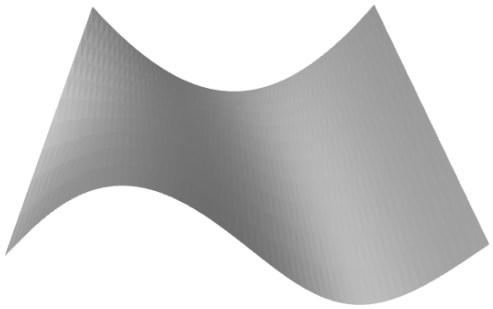


Fig. 142: Spanning the graphs of a sine function and a parabola.

In the next example we define two curves that start at (1, 0, π) and (-1, 0, π), respectively, and descend to the xy-plane describing a spiral trajectory:

```
# Import sine and cosine from Numpy:
from numpy import sin, cos

# Define two 3D curves:
def c1(t):
    return [cos(t), sin(t), PI - t]

def c2(t):
    return [cos(PI + t), sin(PI + t), PI - t]

def p(point):
    x = point[0]
    return c1(x)

# Return a vector pointing from one curve to the other:
def q(point):
    x = point[0]
    return [c2(x)[0] - c1(x)[0], c2(x)[1] - c1(x)[1],
            c2(x)[2] - c1(x)[2]]

# Reference domain:
refdomain = REFDOMAIN(PI, 1, 64, 30)

# Creating the ruled surface:
surf = RUSU(p, q)
out = MAP(refdomain, surf)
```

The output is displayed in Fig. 143.

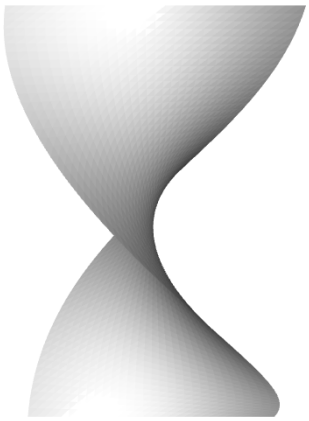


Fig. 143: Spanning a pair of spirals.

7.19 Generalized cylindrical surface

By generalized cylindrical surface we mean a surface that is obtained via extrusion of a curve lying in the xy-plane in a given constant direction. This can be done using the command `CYLINDRICALSURFACE` (that can be abbreviated using `CYS`) which is a simple application of `RULEDSURFACE`. If we use a circle as the underlying curve, and the directional vector is vertical, we obtain the traditional cylinder. Here is an example with a cubic Bézier curve:

```
# Cubic Bezier curve in the xy-plane:
c = BEZIER([1,1,0], [-1,1,0], [1,-1,0], [-1,-1,0])

# Directional vector:
vector = [0, 0, 2]

# Product geometry:
out = CYSU(c, vector, 64, 1)
```

The last two parameters are optional divisions. Their default values are 32 and 32. The output is displayed in Fig. 144.

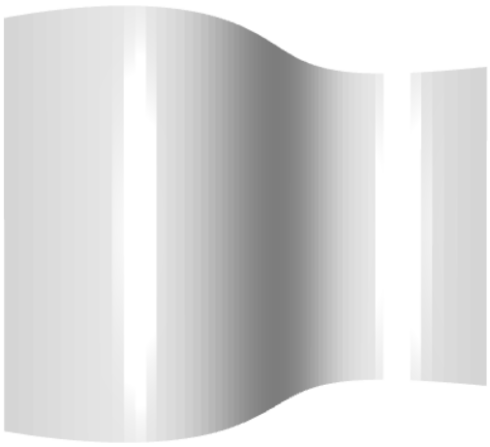


Fig. 144: Generalized cylindrical surface.

7.20 Generalized conical surface

Generalized conical surface (`CONICALSURFACE`, `COSU`) is similar to the generalized cylindrical one, except that the "rays" from the curve all go to just one point that lies above the midpoint of the curve (tip of the generalized cone). If we use a circle as the underlying curve, and the tip is above the center of the curve, we obtain the traditional cone. Let us use again a cubic Bézier curve as an example:

```
# Bezier curve in the xy-plane:
c = BEZIER([1, 1, 0], [-1, 1, 0], [1, -1, 0], [-1, -1, 0])

# Apex of the cone:
point = [0, 0, 2.0]

# Conical surface:
out = COSU(c, point, 64, 8)
```

The last two parameters are optional divisions - their default values and 32 and 32. The output is displayed in Fig. 145.

7.21 Profile product surface

Profile product surface (`PROFILEPRODSURFACE`, `PPSU`) is the Cartesian product of a curve c_1 in the z-direction with another curve c_2 in the xy-plane. The former determines

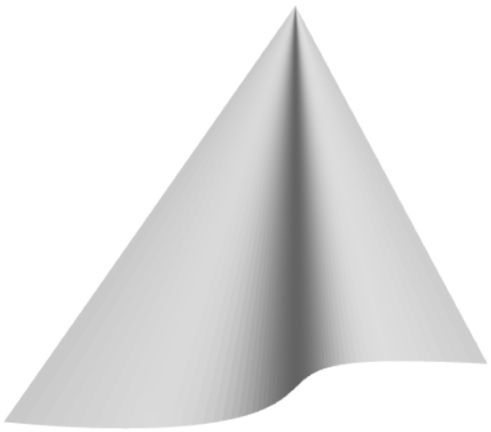


Fig. 145: Conical surface.

the magnitude and the latter the shape. In other words, when we cut the resulting product surface horizontally at some vertical coordinate z_0, then we obtain a curve similar in shape to c_2. The size difference between the cutline and the original curve c_2 is given by $c_1(z_0)$. Let us show two examples. In the first one, c_1 will be a straight line and c_2 a quadratic Bézier curve:

```
# Vertical shape:
c1 = BEZIER([0, 0, 0], [2, 0, 5])

# Base shape in the xy-plane:
c2 = BEZIER2([-1, 0, 0], [0, 3, 0], [1, 0, 0])

# Reference domain:
refdomain = UNITSQUARE(64, 64)

# Profile product surface:
surf = PPSURFACE(c1, c2)
out = MAP(refdomain, surf)
```

The output is displayed in Fig. 146.

In the next example we will construct the product of two cubic Bézier curves.

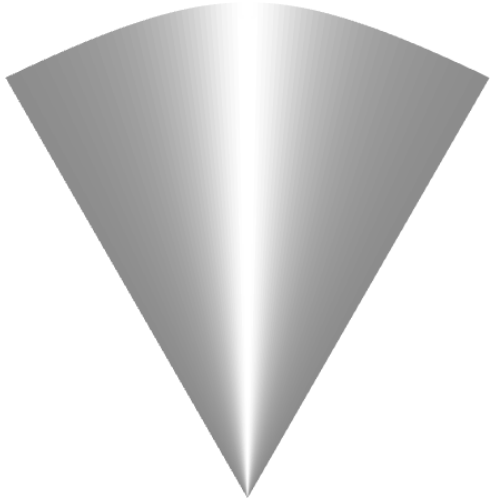


Fig. 146: Product of a straight line in the z-direction with a quadratic Bézier curve in the xy-plane.

```
# Vertical shape:
c1 = BEZIER([0, 0, 0], [2, 0, 0], [0, 0, 4], [1, 0, 5])

# Base shape in the xy-plane:
c2 = BEZIER2([0, 0, 0], [3, 0, 0], [3, 3.5, 0], [0, 3, 0])

# Reference domain:
refdomain = UNITSQUARE(64, 64)

# Profile product surface:
surf = PPSURFACE(c1, c2)
out = MAP(refdomain, surf)
```

The output is displayed in Fig. 147.

7.22 Cubic Hermite curves

A cubic Hermite curve can be defined using the commands `CUBICHERMITE` and `CUBICHERMITE2` that can be abbreviated via `CH` and `CH2`, respectively. The meaning of the suffixes `1` and `2` is the same as for Bézier curves.

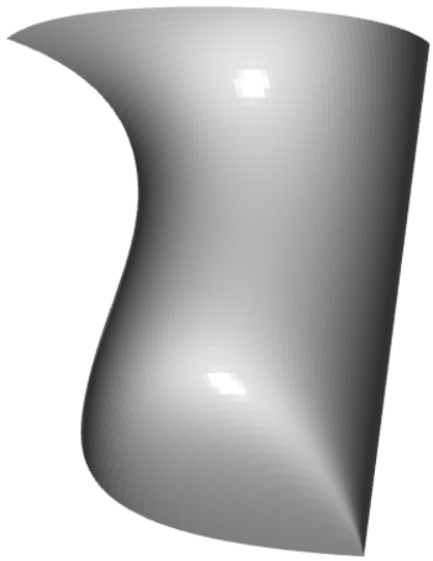


Fig. 147: Product of two cubic Bézier curves.

A cubic Hermite curve is given by its endpoints and tangential vectors at the endpoints - four parameters total. These four parameters define a unique Hermite cubic spline (i.e., cubic polynomial). For example, in the definition

```
point1 = [0, 0, 0]
point2 = [5, 0, 0]
vector1 = [1, 0, 0]
vector2 = [1, 0, 0]
c = CUBICHERMITE1(point1, point2, vector1, vector2)
```

both vectors are parallel to the line connecting the endpoints, and therefore the result is a straight line. The same happens with

```
point1 = [0, 0, 0]
point2 = [5, 5, 5]
vector1 = [1, 1, 1]
vector2 = [1, 1, 1]
c = CUBICHERMITE1(point1, point2, vector1, vector2)
```

If one or both vectors are not parallel to the line connecting the endpoints, the result is not a straight line anymore. For example, with

```
point1 = [0, 0, 0]
point2 = [5, 5, 5]
vector1 = [0, 0, 1]
vector2 = [1, 1, 0]
c = CUBICHERMITE1(point1, point2, vector1, vector2)
```

we obtain a cubic curve that ascends verticaly at $(0, 0, 0)$ with a tangential vector $(0, 0, 1)$ and ends up flat at $(5, 5, 5)$ with tangential vector $(1, 1, 0)$.

7.23 Cubic Hermite surfaces

Cubic Hermite surface spanning two curves differs from a ruled or Bézier surface in that it allows the user to prescribe slope vectors along the two curves. We show two examples. First, for simplicity, the two curves are the opposite edges of the unit square. Both of them are straight lines:

```
# First curve:
c1 = CUBICHERMITE1([0, 0, 0], [0, 1, 0], [0, 1, 0], [0, 1, 0])

# Second curve:
c2 = CUBICHERMITE1([1, 0, 0], [1, 1, 0], [0, 1, 0], [0, 1, 0])
```

Both slope vectors lie in the xz-plane and their angle to the xy-plane is $\pi/4$. This results into a wave-like cubic surface.

```
# Slope vector for the first curve:
s1 = [1, 0, 1]

# Slope vector for the second curve:
s2 = [1, 0, 1]

# Reference domain:
refdomain = UNITSQUARE(32, 32)

# Cubic Hermite surface:
surf = CUBICHERMITE2(c1, c2, s1, s2)
out = MAP(refdomain, surf)
```

The output is displayed in Fig. 148.

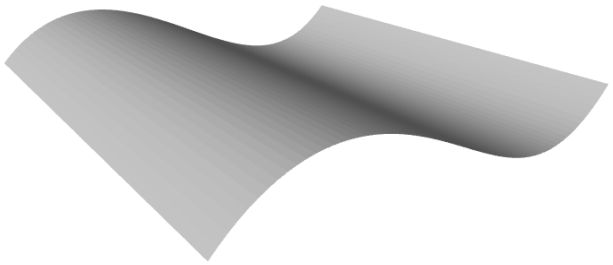


Fig. 148: Cubic Hermite "wave", constant in the y-direction.

Next let us deform the left edge by changing the tangential vectors at endpoints to (1, 1, 0) and (-1, 1, 0):

```
# First curve:
c1 = CUBICHERMITE1([0, 0, 0], [0, 1, 0], [1, 1, 0],
                   [-1, 1, 0])
```

The rest of the script stays the same. The output is displayed in Fig. 149.

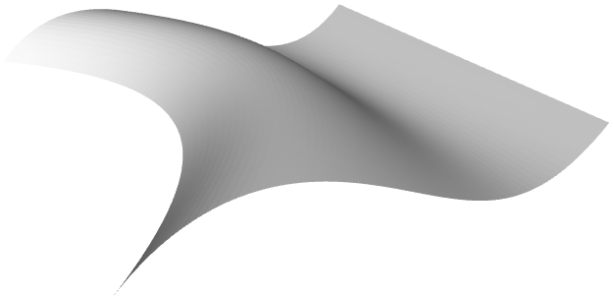


Fig. 149: Cubic Hermite surface after deforming the left edge.

We can also change the right edge to a cubic wave by adjusting the tangential vectors at endpoints to (0, 1, 1) and (0, 1, 1):

```
# Second curve:
c2 = CUBICHERMITE1([1, 0, 0], [1, 1, 0], [0, 1, 1],
                   [0, 1, 1])
```

The rest of the script remains the same. The output is displayed in Fig. 150.

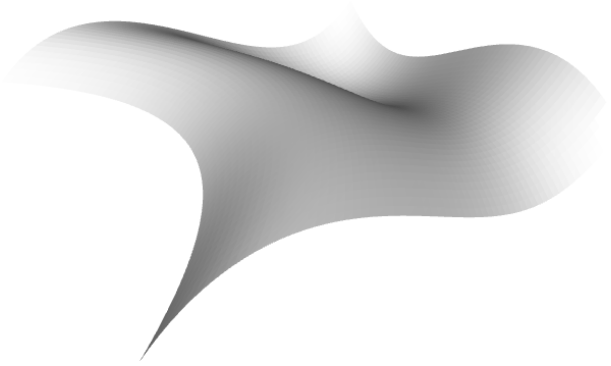


Fig. 150: Cubic Hermite surface after deforming also the right edge.

Last we create a cubic Hermite surface between two arcs:

```
# First (outer) curve:
c1 = CUBICHERMITE1([1, 0, 0], [0, 1, 0], [0, 3, 0],
                   [-3, 0, 0])

# Second (inner) curve:
c2 = CUBICHERMITE1([0.5, 0, 0], [0, 0.5, 0], [0, 1, 0],
                   [-1, 0, 0])

# Slope vector for the outer curve:
s1 = [0, 0, 1]

# Slope vector for the inner curve:
s2 = [-1, -1, -1]

# Reference domain:
refdomain = UNITSQUARE(32, 32)

# The 3D object:
surf = CUBICHERMITE2(c1, c2, s1, s2)
out = MAP(refdomain, surf)
```

The output is displayed in Fig. 151.

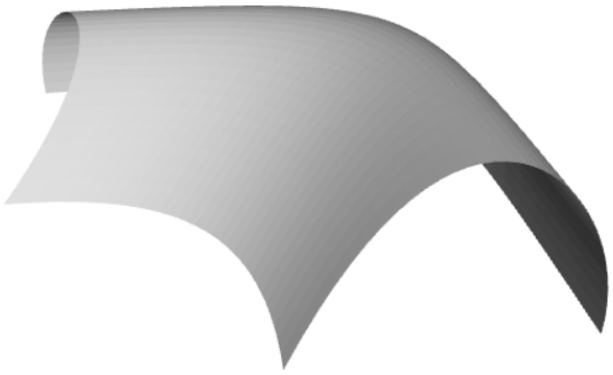


Fig. 151: Cubic Hermite surface.

Index