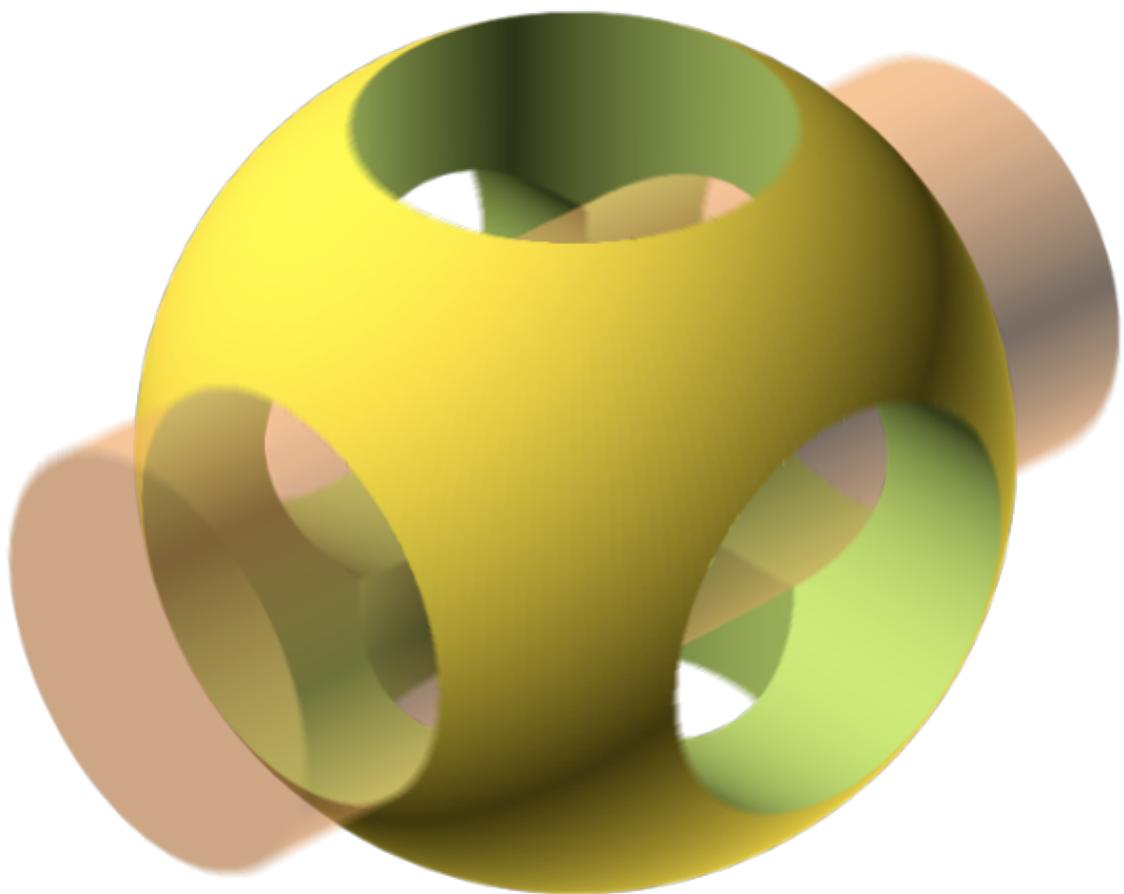


OpenScad

Modélisation 3D pour programmeurs

Rudi Giot - Dernière révision le 9 novembre 2020



**Attribution - Pas d'Utilisation
Commerciale - Pas de Modification 3.0
non transposé (CC BY-NC-ND 3.0)**

Table des matières

1. Introduction	5
1.1. Installation	6
1.2. Interface	6
1.3. Primitives et syntaxe de base	7
1.3.1. Sphere	7
1.3.2. Cube	8
1.3.3. Cylindre	9
1.3.4. Polyèdre	12
1.4. Résolution	16
1.5. Compilation	21
1.6. Console	22
2. Variables et Mathématiques	23
2.1. Déclaration et affectation	23
2.2. Utilisation	24
2.3. Opérations mathématiques	24
2.4. Types	25
3. Transformations	27
3.1. Translation	28
3.2. Rotation	32
3.3. Redimensionnement	36
3.4. Couleurs	37
3.5. Symétrie	40
4. Algèbre de Boole	41
4.1. Union	42
4.2. Différence	44
4.3. Intersection	48

5.	Opérations géométriques	55
5.1.	Somme de Minkowski	55
5.2.	Convex Hull	57
6.	Modules	60
6.1.	Fonction	61
6.2.	Module	62
6.3.	Include et Use	64
6.4.	Importation	68
7.	Boucle et Condition	69
7.1.	Boucle For	69
7.2.	Intersection For	73
7.3.	Conditions	74
8.	Extrusion	75
8.1.	Primitives 2D	75
8.1.1.	Rectangle	75
8.1.2.	Cercle	76
8.1.3.	Polygone	76
8.1.4.	Redimensionnement	77
8.1.5.	Texte	78
8.1.6.	Surface	78
8.2.	Extrusion	79
8.2.1.	Extrusion linéaire	79
8.2.2.	Extrusion rotationnelle	82
9.	Animations	84
9.1.	Translation animée	85
9.2.	Rotation animée	85
10.	Exercices récapitulatifs	87
	Annexes	89

Préface

Ce document est destiné aux programmeurs qui désirent s'initier à la 3D grâce à un environnement proche de leurs outils de développement ainsi qu'aux personnes qui conçoivent déjà des modèles 3D mais qui désirent concevoir des pièces difficilement réalisables avec les logiciels « classiques » tel que *Blender*.

Ce syllabus a été réalisé en se basant sur la documentation officielle du site *OpenScad* <http://www.openscad.org/documentation.html> et de « tutoriels » sur *Youtube* et autres blogs :

- https://www.youtube.com/channel/UCz8_pJ9-DI8SLINVnMxsYow
- <http://static.fablab-lannion.org/tutos/openscad/#/pageDeGarde>
- <https://www.simonlefort.be/blog/openscad-1.html>
- <https://blog.bandinelli.net/index.php?q=openscad>

Tous les éléments de ce document sont originaux sauf certains passages ou illustrations qui sont alors référencés. Ce document est mis à disposition sous licence Attribution - Pas d'Utilisation Commerciale - Pas de Modification 3.0 non transposé. Pour voir une copie de cette licence, visitez <http://creativecommons.org/licenses/by-nc-nd/3.0/>.

Bonne lecture.

1. Introduction

OpenScad est un logiciel multiplateforme (*Linux/UNIX, Windows et MacOS*) de **CAO** (Conception Assistée par Ordinateur) qui permet de créer des modèles 3D. Contrairement à la plupart des logiciels gratuits de modélisation 3D, comme *Blender*, il n'a pas pour but de créer des modèles destinés au cinéma d'animation ou au jeu vidéo, il se focalise sur l'aspect **CAO**, destiné à fabriquer des pièces pour de la construction de machines ou de mécanismes. Cependant, certaines personnes l'utilisent de manière créative pour réaliser des impressions 3D destinées à la décoration, au jeu de plateau, ...

OpenScad n'est pas un modeleur interactif comme *Blender*, il nécessite l'écriture de *scripts* (code informatique) qui décrivent les objets 3D qui composent le modèle. Cette « originalité » vous permet de créer des formes « algorithmiques » qui peuvent être reproduites et modifiées en changeant quelques paramètres.

OpenScad peut également importer des fichiers *DXF* et exporter les modèles créé aux formats *STL* et *OFF*.

OpenScad est un logiciel idéal pour apprendre certains principes de base de l'algorithmique, des mathématiques et de la programmation.

OpenSCAD est un logiciel gratuit réalisé sous « General Public License version 2 », il est disponible à l'adresse <http://www.openscad.org/> et est maintenu à jour par *Marius Kintel*.

Ce syllabus sera divisé en deux parties qui correspondent aux deux techniques de modelage que *OpenScad* propose :

- La première partie, la plus longue, consiste en l'addition/soustraction de formes 3D de base (**Constructive Solid Geometry - CSG**)
- La seconde consiste en l'extrusion de formes 2D

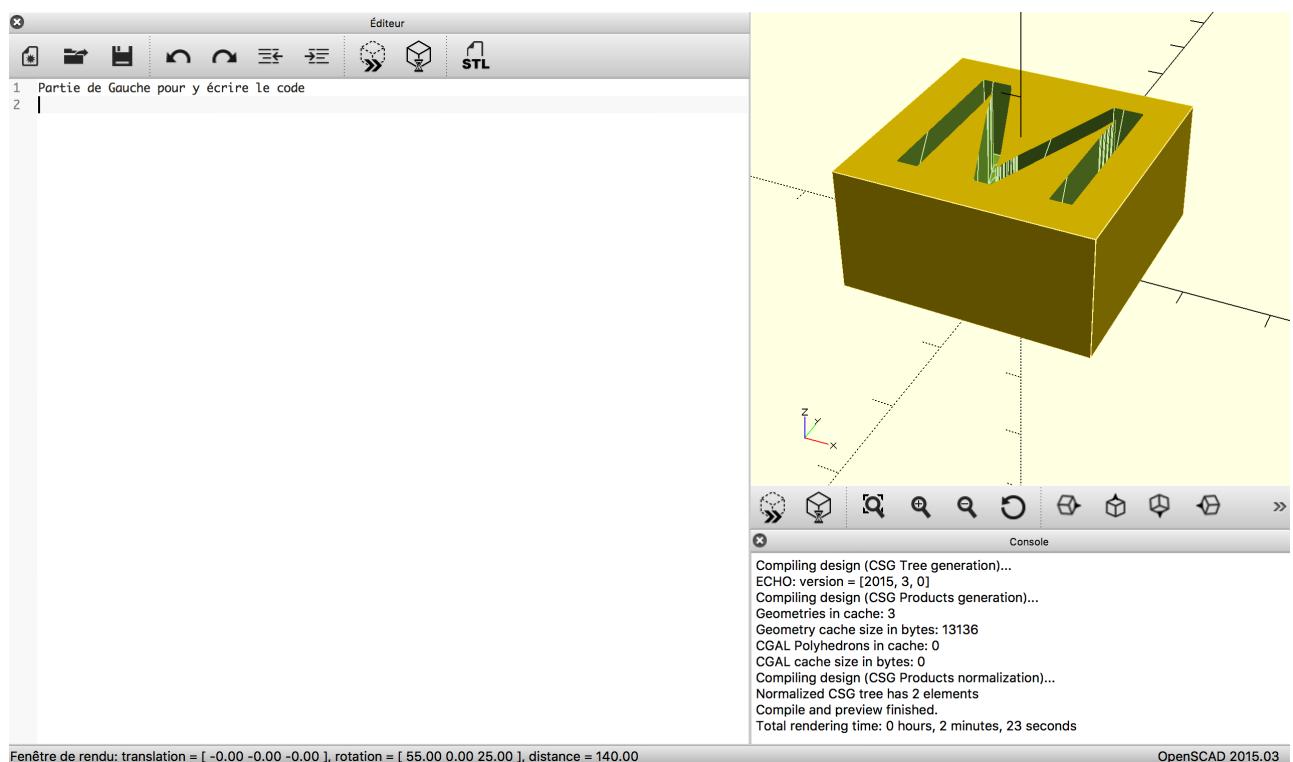
Les exemples et exercices de ce syllabus sont interprétés dans la version **2015.03-3** de *OpenScad*.

1.1. Installation

Pour installer le logiciel reportez-vous à la marche à suivre spécifiée à l'adresse <http://www.openscad.org/downloads.html> en fonction de votre système d'exploitation. Exécutez ensuite le logiciel pour découvrir son interface.

1.2. Interface

L'interface d'*OpenScad* est divisée en trois parties. Sur la gauche, on peut saisir le code, dans la « fenêtre de script », qui sera compilé pour composer le modèle 3D que l'on visualisera dans la sous-fenêtre en haut à droite.



Interface de OpenScad

En bas, à droite, on trouve la console dans laquelle on visualisera tous les messages de compilation et les erreurs éventuelles.

Chaque fenêtre est redimensionnables en fonction de votre projet et de la taille de votre écran de travail.

1.3. Primitives et syntaxe de base

La *Constructive Solid Geometry* (CSG) est une branche de la modélisation 3D qui permet de créer des modèles comme combinaison d'objets simples appelés primitives (sphère, cylindre, cube et polyèdre) à l'aide d'opérateurs géométriques booléens (union, intersection et soustraction). Nous allons donc voir d'abord comment créer chacun des éléments de base pour ensuite les combiner. Nous verrons en même temps les règles de base de la syntaxe du langage d'*OpenScad*.

1.3.1. Sphere

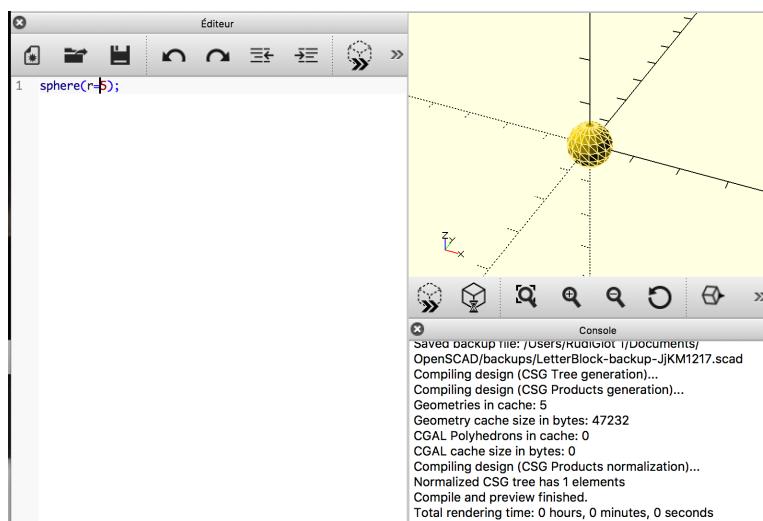
Vous pouvez écrire votre premier code dans la « fenêtre de script » :

```
sphere(r=5);
```

La fonction *sphere()* prend un paramètre, le rayon (*r*) à qui l'on donne la valeur 5 (l'unité de mesure dans *OpenScad* est le **millimètre**). Comme dans beaucoup de langages informatiques, chaque ligne de code se termine par un « ; ». Vous pouvez ensuite « compiler » votre programme, ce qui va se traduire par un affichage de la sphère dans la fenêtre de droite. Pour réaliser cette compilation vous pouvez soit :

- faire *Menu -> Conception -> Aperçu*
- pressez la touche F5
- cliquer sur l'icône

Votre écran devrait ressembler à ceci :



« Compilation » d'une Sphère

Vous remarquerez dans la fenêtre de visualisation qu'il est possible de zoomer/dézoomer (avec la molette de la souris), tourner autour de la sphère avec un click gauche/déplacement de souris et se déplacer avec un click droit (ou touche CTRL)/déplacement de souris.

Vous pouvez aussi visualiser cette fenêtre depuis des points de vue définis dans le *Menu -> Vue -> Haut, Dessus, Gauche, ...* Si vous ne voyez plus rien parce que vous vous êtes trop déplacés, vous pouvez faire un *Menu -> Vue -> Réinitialiser la vue*. Vous pouvez aussi réaliser ces opérations en utilisant les icônes qui se trouvent en dessous de la fenêtre de visualisation du modèle :

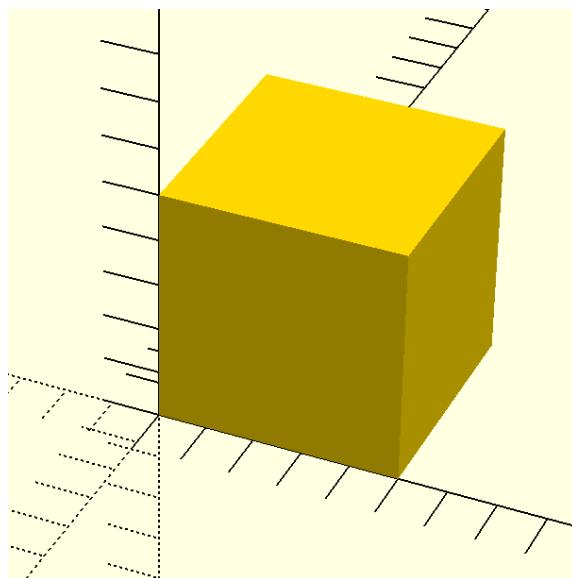


Icônes pour les raccourcis

1.3.2. Cube

Vous pouvez maintenant expérimenter d'autres formes de base (**primitives**). Remplacez la ligne précédente (*sphere*) avec l'instruction suivante (le paramètre *size* représente la longueur du côté du cube) et visualisez le résultat :

```
cube(size=5);
```



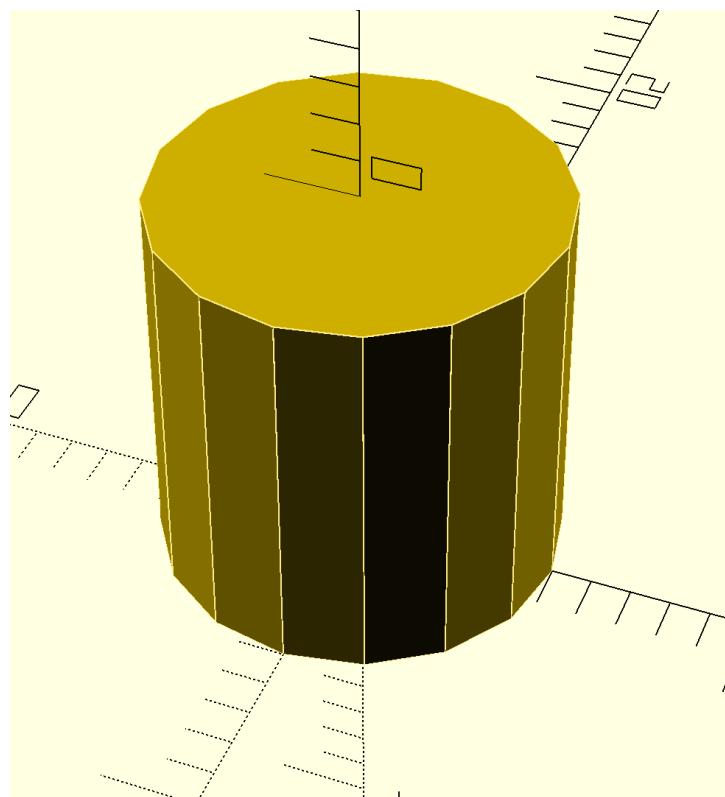
La primitive « Cube »

1.3.3. Cylindre

Vous pouvez ensuite essayer la primitive « cylindre » qui utilise la syntaxe suivante (les paramètres h et r représentent respectivement la hauteur et le rayon de la base circulaire) :

```
cylinder(h=10, r=5);
```

Le résultat est le suivant :



La primitive « Cylindre »

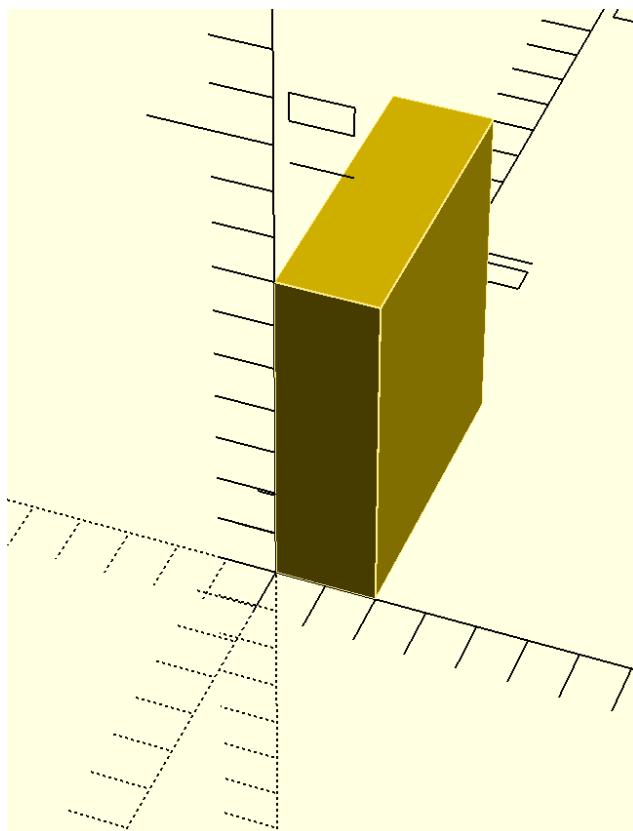
Pour insérer des commentaires dans un script, il suffit, comme dans la plupart des langages de programmation, d'utiliser le double *slash* (//) :

```
// Commentaire sur une seule ligne
```

Tout ce qui suit les « // » jusqu'à la fin de la ligne est ignoré par le compilateur.

On peut, à partir des formes de base, construire de nouvelles formes :

```
// Pour un parallélépipède rectangle on fait :  
cube(size=[2, 6, 7]);  
// Les trois valeurs entre les crochets représentent  
// respectivement la taille des côtés en x, y et z
```



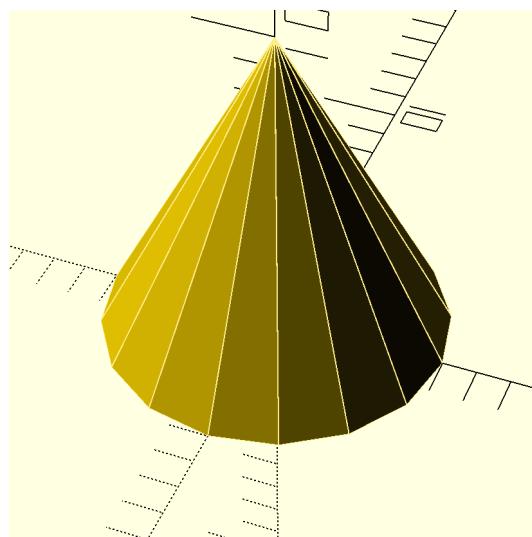
Parallélépipède rectangle à partir d'une primitive « Cube »

Si on désire commenter plusieurs lignes dans du code sans devoir mettre un double slash en début de chaque ligne, on utilise « /* » pour signaler le début jusqu'au « */ » qui clôture le commentaire. Tout ce qui se trouve à l'intérieur est ignoré par le compilateur.

```
/* Mon commentaire  
sur plusieurs  
lignes */
```

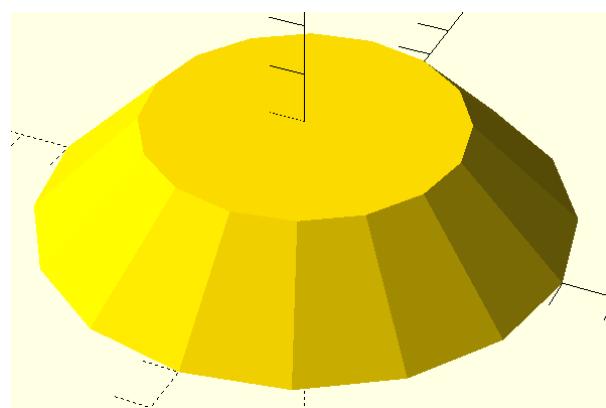
A partir d'un cylindre, on peut réaliser des cônes avec les syntaxes suivantes :

```
/* Réalisation d'un cône à partir d'un cylindre.  
Le h représente la hauteur  
Le r1 représente le rayon de la base  
Le r2 représente le rayon du sommet  
*/  
cylinder(h=10, r1=5, r2=0);
```



Cône à partir de la primitive « Cylindre »

```
cylinder(h=2, r1=5, r2=3);
```



Cône tronqué à partir de la primitive « Cylindre »

1.3.4. Polyèdre

La quatrième forme de base, le **Polyèdre**, est plus complexe et nécessite beaucoup plus de lignes de codes. Il faut, en effet, définir dans l'espace, les points des sommets (*vertices*) qui vont déterminer des triangles qui vont représenter les « facettes » du maillage (*mesh*) qui formera notre polyèdre.

Au niveau de la syntaxe, il faut d'abord écrire que nous souhaitons créer un polyèdre avec l'instruction :

```
polyhedron( ... );
```

Entre les parenthèses, on va énumérer la liste des sommets du polyèdre, chaque sommet étant décrit par ses coordonnées x , y et z :

```
polyhedron (
    points=[[-10,-10,0], [-10,10,0], [10,10,0]]
    ...);
```

Attention, les points ainsi définis sont ordonnés et possèdent chacun un indice qui commence à zéro. Concrètement, dans l'exemple précédent, le point d'indice 0 a pour coordonnées [-10,-10,0] , celui d'indice 1 a pour coordonnées [-10,10,0] et le point d'indice 2 a pour coordonnées [10,10,0]. Ceci est important car pour former les triangles (facettes) ce sont ces indices que nous allons utiliser.

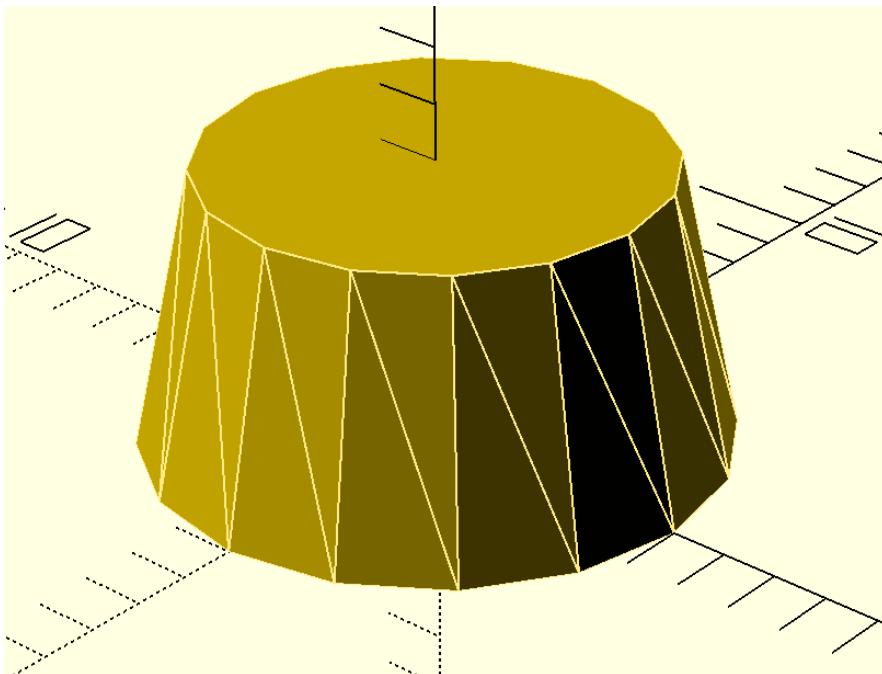
Nous allons donc décrire chaque facette (*faces*) en précisant l'indice de ses trois sommets.

```
polyhedron (
    points=[[-10,-10,0], [-10,10,0], [10,10,0]],
    faces=[[0,1,2]]
);
```

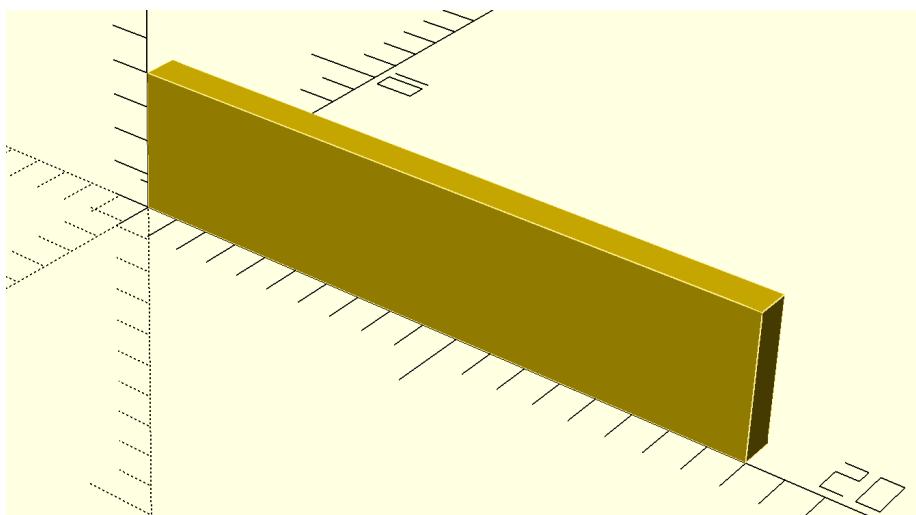
Dans cet exemple, nous dessinons un seul triangle. Vous pouvez, dès lors, imaginer la complexité que peut prendre le code lorsqu'on veut dessiner une forme complexe avec des centaines de faces.

Exercices :

Réaliser les formes suivantes en commentant votre code.

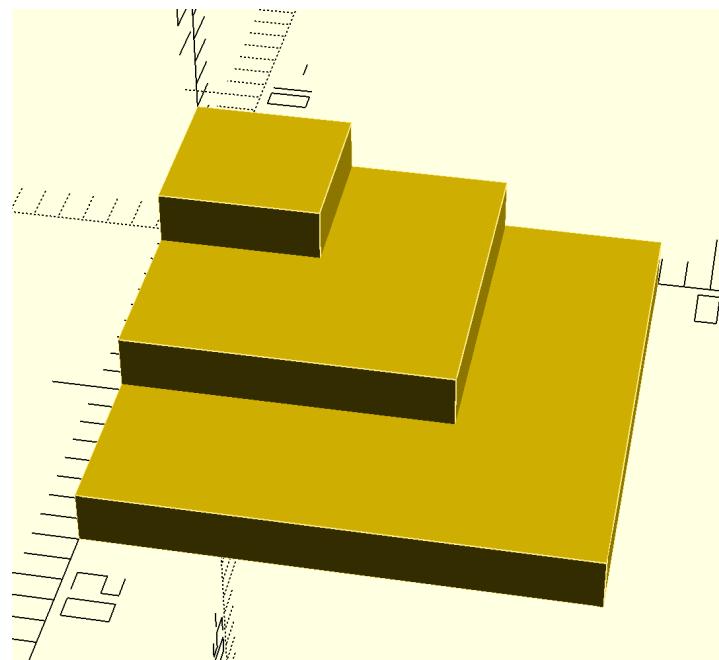


Tabouret pour lion au cirque



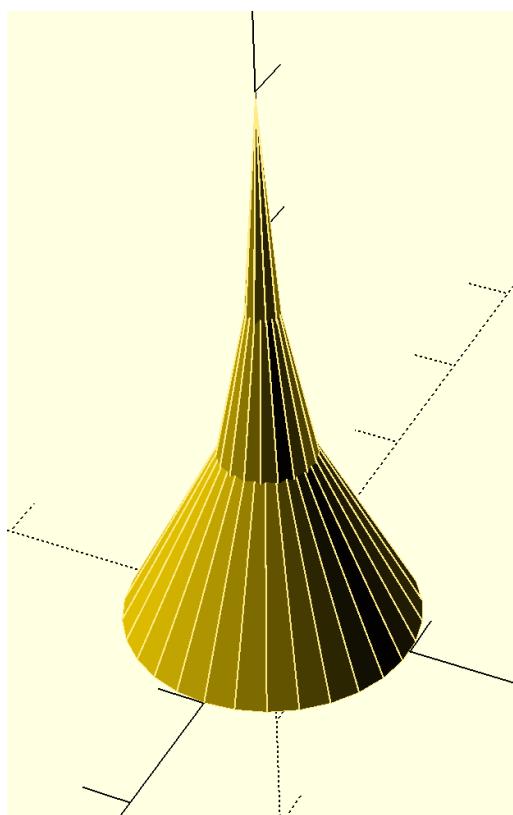
Un élément de Kapla

Un escalier sur un coin :



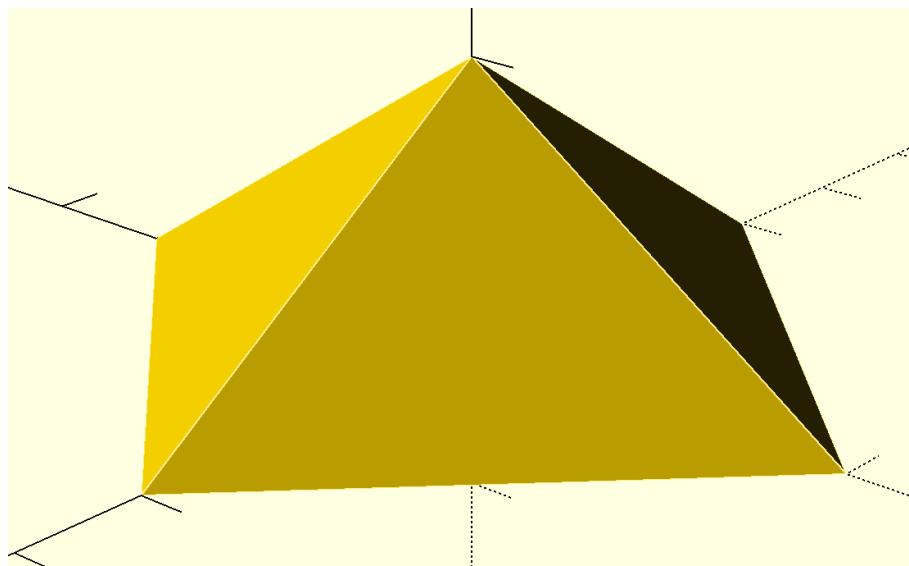
L'escalier de coin

Un petit toit en forme de cône :



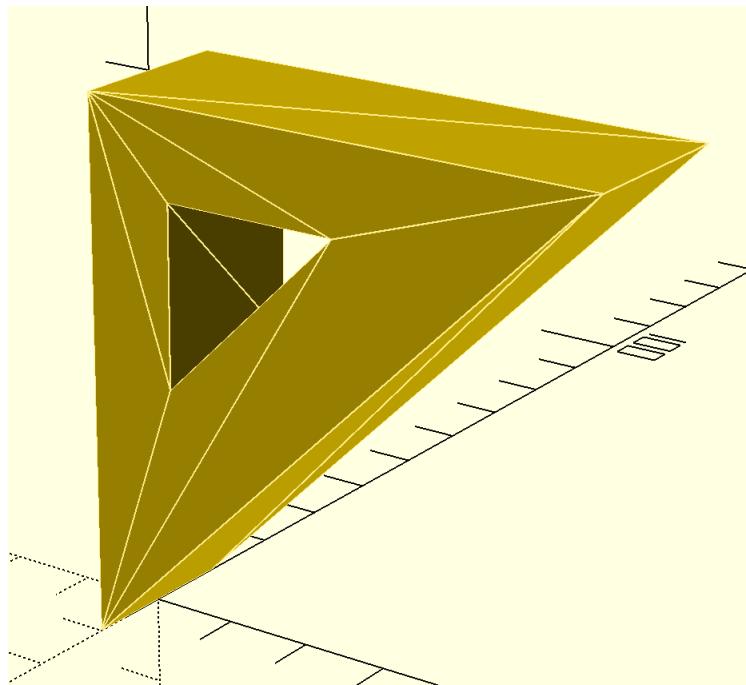
Toit conique

Réalisez une pyramide à partir de l'instruction *polyhedron()*



Pyramide égyptienne avec polyhedron()

Si vous êtes courageux ou téméraire, essayez de réaliser la forme suivante :

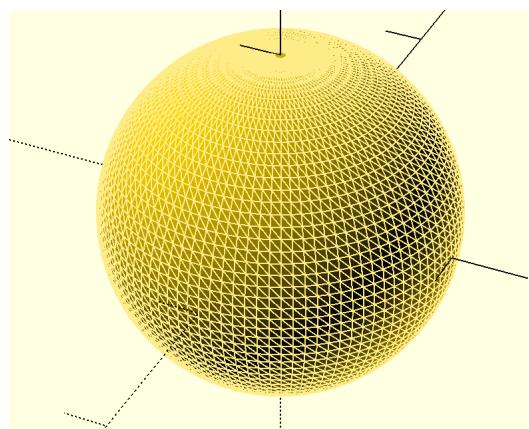
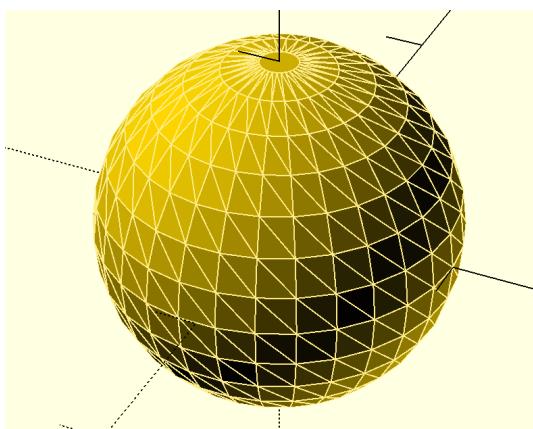


Équerre à partir de polyhedron()

1.4. Résolution

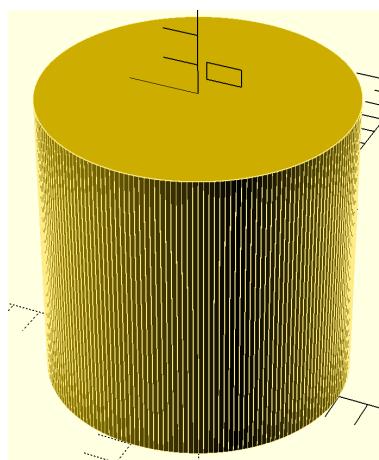
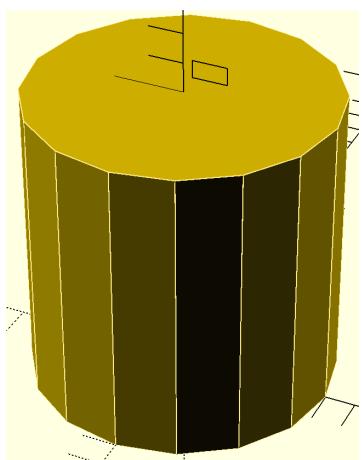
Vous avez certainement remarqué que les sphères et les cylindres que nous avons dessinés n'étaient pas très « lisses ». C'est un problème de résolution, elle est réglable en utilisant le paramètre `$fn` dans les formes de base. Par exemple, essayez les lignes suivantes :

```
sphere(r=10, $fn=100);
```



Sphère avec à gauche une résolution par défaut et à droite une résolution de 100

```
cylinder(h=10, r=5, $fn=200);
```



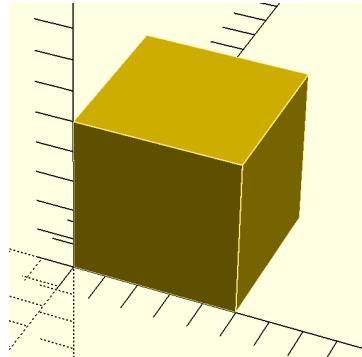
Cylindre avec à gauche une résolution par défaut et à droite une résolution de 200

Le soucis avec la résolution, c'est que plus vous l'augmentez et plus votre modèle sera complexe et donc « lourd » à l'exportation (plus de temps de calcul et plus d'espace disque).

Remarques :

- La résolution n'est pas applicable au cube puisqu'elle compliquerait inutilement le modèle sans y apporter visuellement une différence, le code suivant n'a donc pas vraiment de sens :

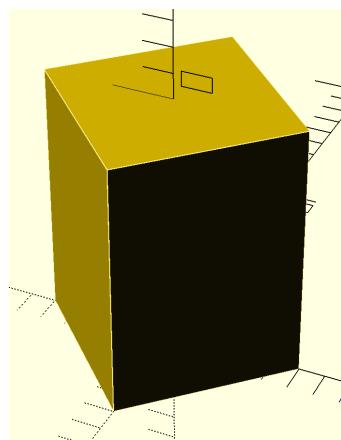
```
cube(5, $fn=100);
```



Avec une résolution de 100 un cube reste un cube

- Si on réalise un cylindre avec une résolution de 4 on obtient un parallélépipède rectangle à base carrée

```
cylinder(h=10, r=5, $fn=4);
```

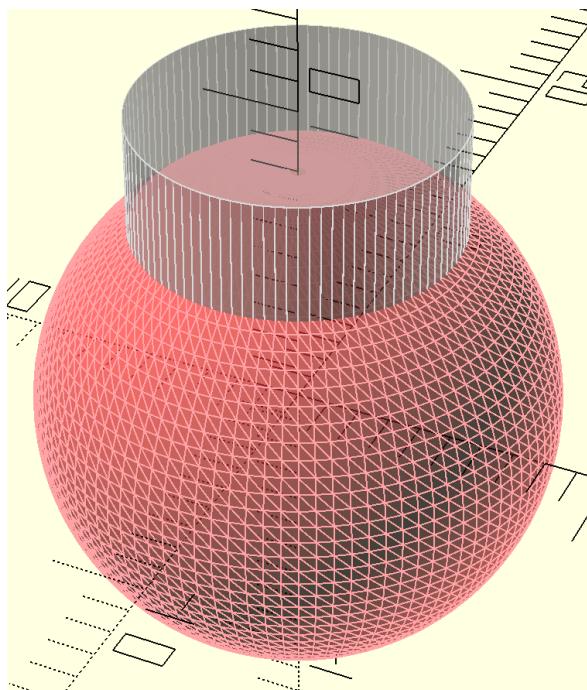


Parallélépipède obtenu à partir d'un cylindre

- Lorsqu'on affiche plusieurs formes on peut, pour savoir quelle instruction affiche quelle forme, faire précéder l'instruction d'un dièse (#), d'un pour-cent (%), d'un point-d'exclamation (!) ou d'une étoile (*). La forme est alors affichée seule (!), invisible (*), en rouge(#) ou en transparence (%).

Par exemple :

```
#sphere(r=8, $fn=100);  
%cylinder(h=10, r=5, $fn=100);
```



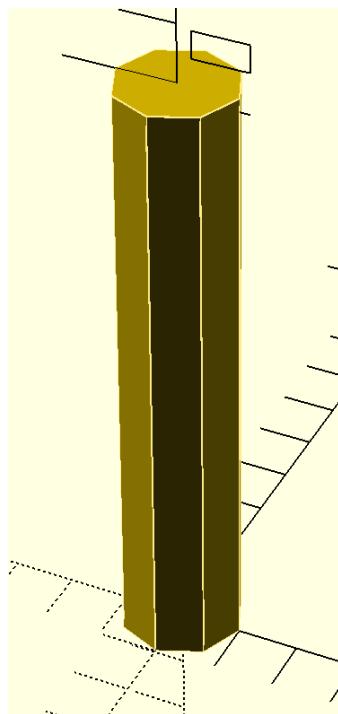
Utilisation des modificateurs # et %

Exercice :

Testez les différents modificateurs (#, %, ! et *) pour visualiser leur effet respectif.

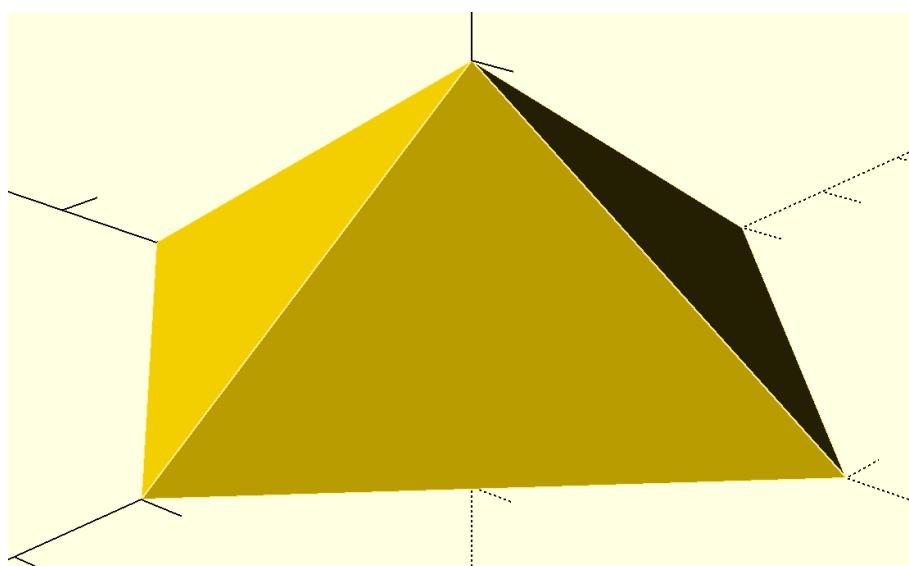
Exercices :

Réalisez à partir de l'instruction *cylinder()* un parallélépipède rectangle de 10mm de hauteur à base octogonale. Cet exercice, aussi simple soit-il, est très intéressant car il est à la base de réalisations complexes à base de boulons, par exemple.



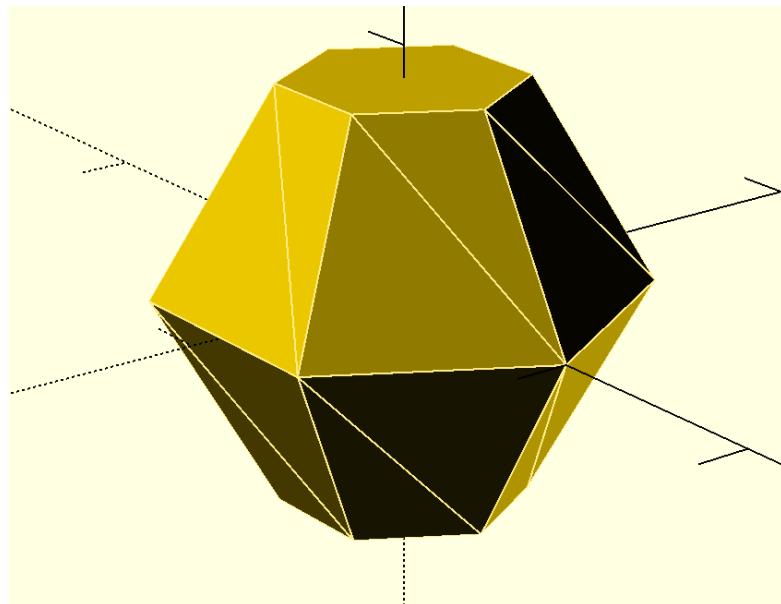
Parallélépipède rectangle à base octogonale

Réalisez une pyramide égyptienne **sans** l'instruction *polyhedron()* :



Pyramide égyptienne sans polyhedron

Réalisez une sphère à faible résolution pour obtenir le solide suivant :

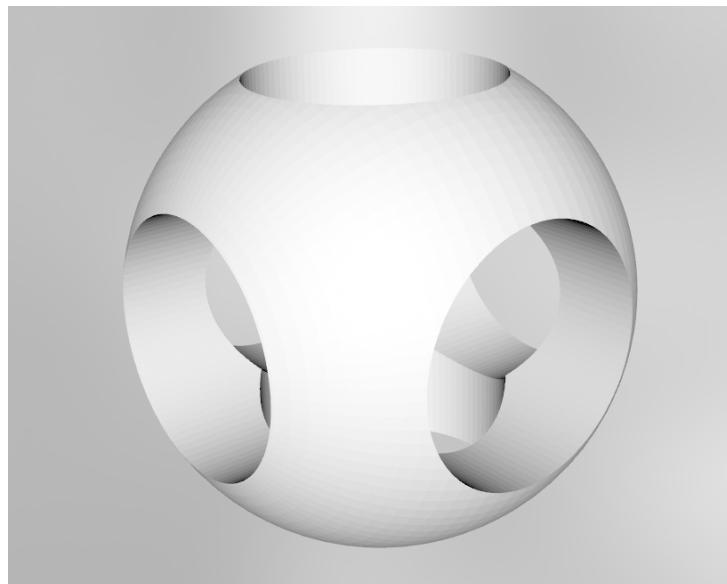


Sphère à faible résolution

1.5. Compilation

La *compilation* peut prendre plusieurs formes. Pour l'instant, nous avons juste réalisé un « *aperçu* » du résultat dans notre fenêtre (raccourci F5). Si vous désirez « concrétiser » votre réalisation sur une imprimante 3D, il faut alors faire un « *rendu* » (raccourci F6). Ce type de compilation consomme plus de ressources machines (surtout si votre résolution est élevée). Pour des modèles complexes avec une résolution élevée (plus de 100), la compilation peut durer plusieurs heures. Ceci dit, plus la résolution est élevée et plus votre modèle imprimé sera « *lisse* ». Cette résolution fait donc l'objet d'un compromis « *temps de calcul* » vs « *qualité du rendu* ».

Une fois ce rendu obtenu vous pouvez exporter votre modèle au format *STL*. Ce format est le plus courant et le plus populaire pour être envoyé à des imprimantes 3D.



Logo OpenScad exporté en STL

Remarque :

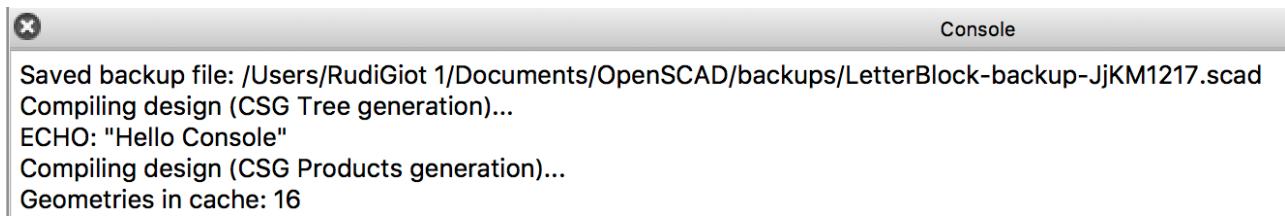
Il n'est pas toujours nécessaire d'avoir une résolution élevée à la compilation. En effet, si l'imprimante 3D que vous utilisez n'est pas très précise, vous pourrez adapter la résolution à sa précision. Le paramètre résolution sera donc toujours dépendant de la machine sur laquelle le modèle sera imprimé.

1.6. Console

La *console* est l'endroit où l'on visualise les erreurs de compilation mais elle peut également servir pour le débugage. En effet, nous pouvons, par exemple, y afficher des messages intermédiaires lors de l'interprétation du code. Par exemple, vous pouvez taper le code suivant dans la fenêtre de script et faire *F5* :

```
echo("Hello Console");
```

Vous voyez alors apparaître dans la console le message « Hello Console ».



The screenshot shows a window titled "Console". Inside, the following text is displayed:

```
Saved backup file: /Users/RudiGiot 1/Documents/OpenSCAD/backups/LetterBlock-backup-JjKM1217.scad
Compiling design (CSG Tree generation)...
ECHO: "Hello Console"
Compiling design (CSG Products generation)...
Geometries in cache: 16
```

Message dans la console

Si vous avez des notions d'*HTML* vous pourrez même mettre votre texte en forme avec des balises. Par exemple, pour mettre en gras :

```
echo("<strong>Hello Console</strong>");
```

C'est surtout pour visualiser l'état des variables que cet affichage est intéressant. C'est dans le chapitre suivant que nous allons aborder cette notion de variable, savoir à quoi elles servent et comment on peut les utiliser.

2. Variables et Mathématiques

Les **variables** représentent une des notions primordiales de *OpenScad*, c'est en effet grâce à elles que nous pourront facilement modifier des paramètres d'un modèle mais aussi faire évoluer ces paramètres dans des algorithmes élaborés pour générer des modèles complexes.

2.1. Déclaration et affectation

Les variables sont à la base de la majorité des langages de programmation, elles permettent de stocker des valeurs dans une case mémoire à laquelle est associée un label (le nom de la variable).

Une variable est déclarée simplement de la manière suivante :

- la variable porte un label (un nom) qui est unique
- ce nom comporte des caractères alphanumériques mais commence par une lettre
- idéalement le nom de la variable est explicite (préférez *rayon* à *r*)

L'attribution (**affectation**) d'une valeur à une variable se fait grâce à l'opérateur égal :

```
rayon = 5;  
hauteur = 10;
```

Dans la première ligne, on déclare une variable « rayon » à laquelle on affecte la valeur 5. Dans le deuxième exemple, on déclare une variable « hauteur » à laquelle on affecte la valeur 10.

2.2. Utilisation

Pour exploiter une variable, c'est très simple, il suffit d'utiliser son nom à la place d'une valeur, par exemple, comme paramètre d'une fonction :

```
rayon = 5;  
hauteur = 10;  
echo(hauteur);  
echo("Le Rayon = ", rayon);
```

Vous remarquerez que l'on peut faire précéder l'affichage de la valeur de la variable par un message spécifié entre guillemets.

Il est évidemment plus intéressant d'utiliser les variables comme paramètre d'une fonction qui dessine un modèle de base.

Par exemple, pour un cylindre :

```
rayon = 5;  
hauteur = 10;  
resolution = 100;  
cylinder(h=hauteur, r=rayon, $fn=resolution);
```

2.3. Opérations mathématiques

Le plus intéressant avec les variables, c'est de pouvoir les utiliser dans des opérations algébriques (+, -, *, /).

Par exemple, si vous voulez dans votre programme spécifier la circonference de la base du cylindre plutôt que son rayon, vous pouvez écrire le code suivant :

```
circonference = 10;  
rayon = circonference / (2 * 3.14159);  
echo(rayon);  
cylinder(h=10, r=rayon, $fn=100);
```

2.4. Types

Vous déclarez implicitement le type de la variable au moment de son assignation. Dans *OpenScad*, on trouve les types suivants :

- nombre (pas de différence entre un nombre entier et un nombre réel)
- booléen (vrai ou faux)
- chaîne de caractère
- intervalle
- vecteur

En voici quelques illustrations :

```
var = 25;           // type nombre ou number
rayon = 2.5;        // type nombre ou number
logic = true;       // type booléen ou boolean
MyString = "Hello"; // type chaîne de caractère ou string
a_vector = [1,2,3]; // type vecteur ou vector
xx = [0:5];         // type intervalle ou range
```

Le *vector* est un type qui permet de stocker un tableau de plusieurs nombres, en général trois, puisque nous allons travailler en 3D. Les vecteurs seront utilisés essentiellement pour donner des coordonnées de points dans l'espace ou pour déterminer des translations ou des rotations.

Le type *range* permet de stocker une série de nombre avec une valeur minimum, une maximum et un incrément (par défaut, s'il n'est pas spécifié sa valeur = 1). Un intervalle se déclare avec les crochets de la manière suivante :

```
variable_de_type_range = [minimum: incrément: maximum];
```

Par exemple :

```
x = [0:3];      // stocke les valeurs 0, 1, 2, 3
y = [0:2:8];    // stocke les valeurs 0, 2, 4, 6, 8
```

Remarque :

Si une variable est utilisée dans un script alors qu'elle n'a pas encore été assignée à une valeur, elle est dite *undefined* (non définie) et portera la valeur *undef*.

```
echo(a); // Donne dans la console ECHO: undef
```

Attention :

Les scripts *OpenSCAD* sont compilés et les variables sont calculées au moment de la compilation et pas au moment de l'exécution. Concrètement, cela signifie que le code suivant ne fera sans doute pas ce que vous espériez qu'il fasse :

```
a = 0;  
echo(a);  
a = 3;  
echo(a);  
a = 5;  
echo(a);
```

En effet, si vous essayez de l'exécuter, vous voyez dans la console :

```
ECHO: 5  
ECHO: 5  
ECHO: 5
```

Le compilateur conservera la dernière valeur affectée à la variable *a* et c'est cette dernière valeur (5) qui sera affichée trois fois.

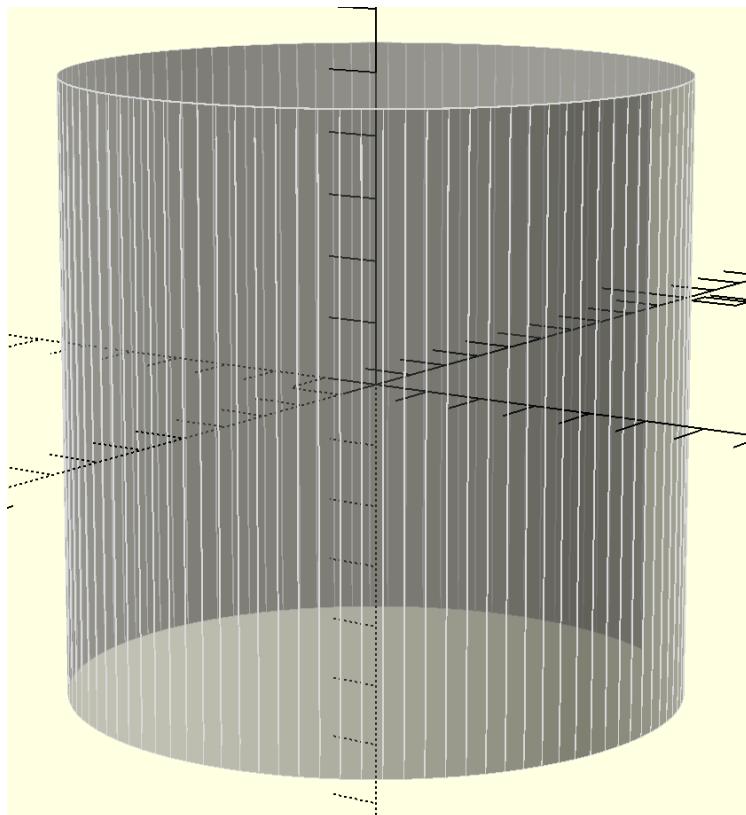
Bien que cette règle ne paraisse pas très naturelle (surtout aux programmeurs qui ont l'habitude d'utiliser ces variables pour changer régulièrement de valeur), elle permet des possibilités intéressantes comme, par exemple, d'inclure une bibliothèque extérieure pour laquelle vous pouvez « écraser » (*override*) les valeurs avec vos propres données. Nous verrons cela plus loin.

3. Transformations

Les volumes que nous avons dessinés jusqu'ici sont placés dans l'espace à une position par défaut. On peut les centrer sur l'origine du référentiel en utilisant la paramètre *center* (Booléen) dans les fonctions *sphere()*, *cylinder()* et *cube()*.

Par exemple :

```
%cylinder(h=10, r=5, $fn=100, center=true);
```



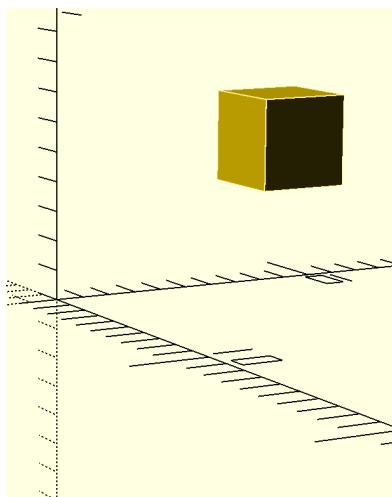
Cylindre centré sur l'origine du repère

Si vous voulez les déplacer à des positions différentes vous allez devoir réaliser des translations et des rotations.

3.1. Translation

La translation permet de déplacer une forme ou un ensemble de formes le long d'un vecteur. L'instruction `translate()` prend donc comme paramètre un vecteur à trois valeurs (correspondants aux trois dimensions x , y et z). Cette instruction doit être immédiatement suivie (sans point-virgule) de la forme à dessiner après cette translation.

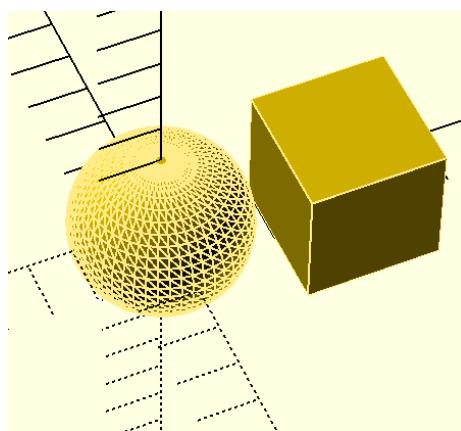
```
translate([2, 5, 4]) cube(3);
```



Cube translating

Dans le code suivant, seul le cube est affecté par la translation :

```
translate([2, -3, 0]) cube(3);
sphere(2, $fn=50);
```

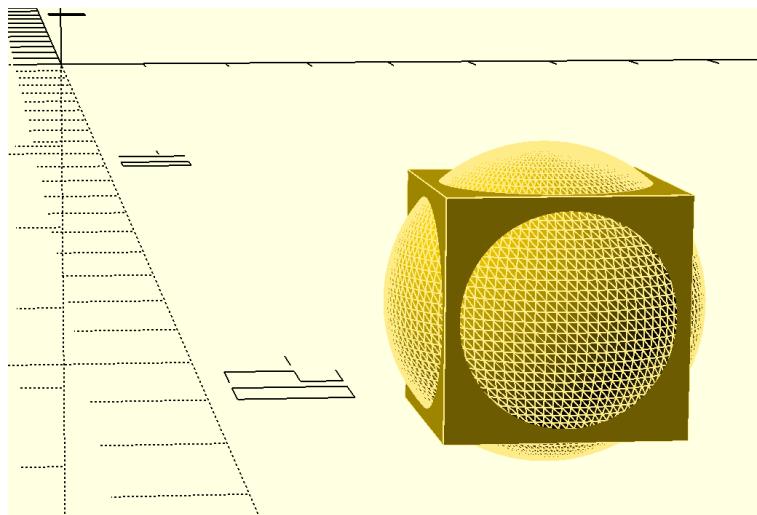


Sphère centrée, cube translating

Pour que les deux formes soient traduites, il faut utiliser les accolades :

```
translate([6, 0, -3]) {  
    cube(3, center=true);  
    sphere(2, center=true, $fn=100);  
}
```

En effet, la syntaxe impose que pour qu'une traduction porte sur un ensemble de primitives, elles doivent être placées entre un couple d'accolades (ouvrante/fermante).



Translation de deux primitives

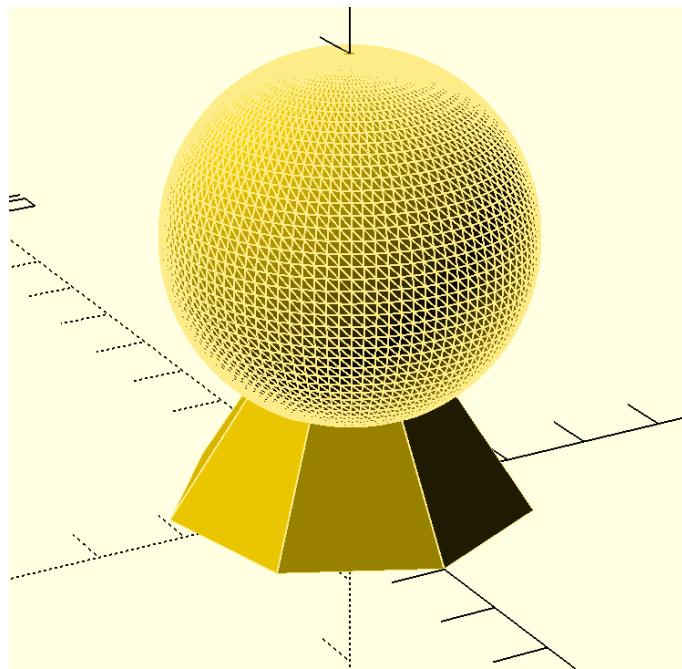
Remarque :

On aurait pu écrire le même code en utilisant une variable pour le vecteur de traduction :

```
a = [6, 0, -3];  
translate(a) {  
    cube(3, center=true);  
    sphere(2, center=true, $fn=100);  
}
```

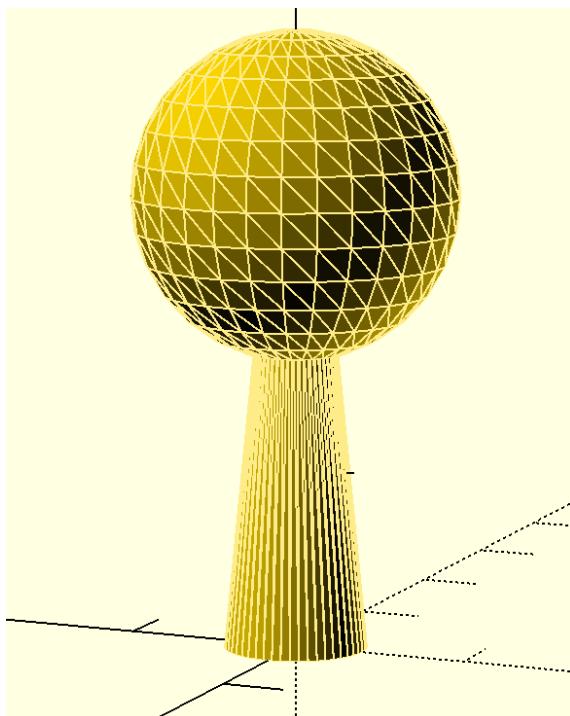
Exercices :

Réalisez une boule de cristal sur son support avec des variables comme paramètres :



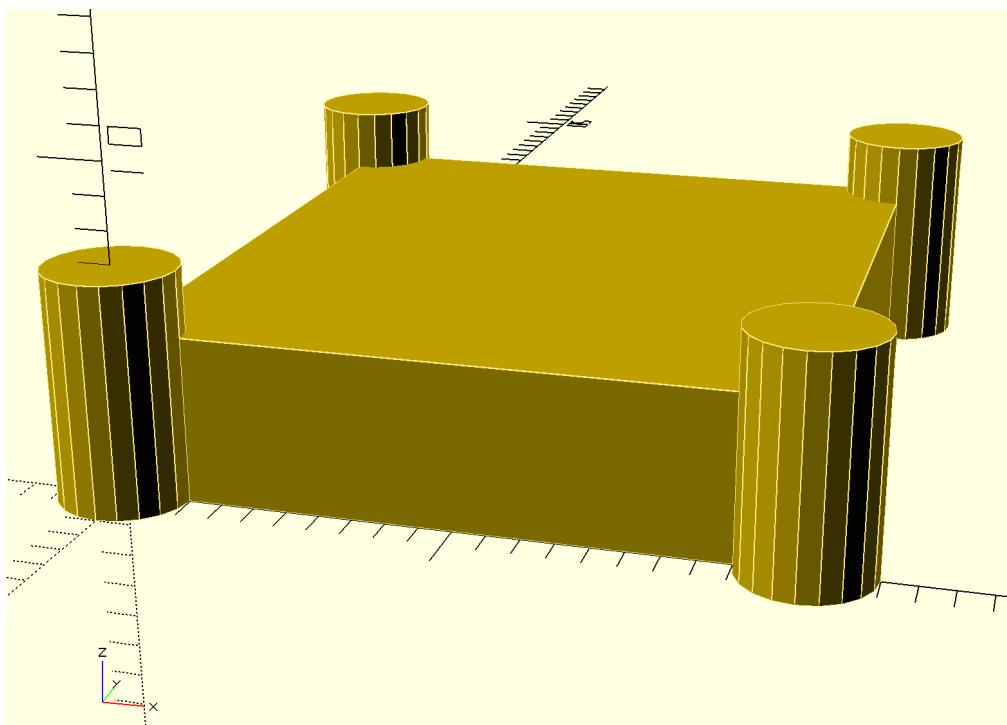
Boule de cristal

A partir de cette boule de cristal sur son support, changer juste les valeurs des variables pour avoir deux « maracas » :



Maracas

Réalisez le squelette d'un château-fort (avec 4 paramètres : hauteurChateau, largeurChateau , longueurChateau et largeurTour) qui pourrait ressembler à l'illustration suivante :

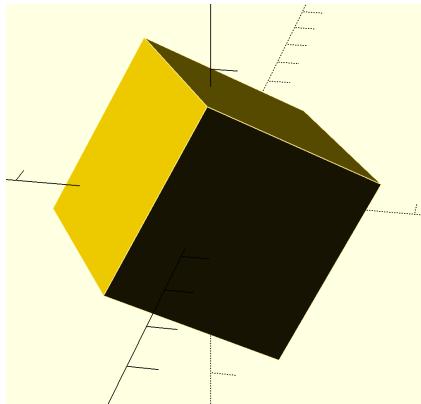


Chateau fort

3.2. Rotation

La rotation est très complémentaire à la translation. En effet, avec ces deux transformations, vous pouvez construire quasiment n'importe quel assemblage. La fonction `rotate()` s'utilise avec un paramètre, vecteur à trois dimensions (l'unité est le degré) :

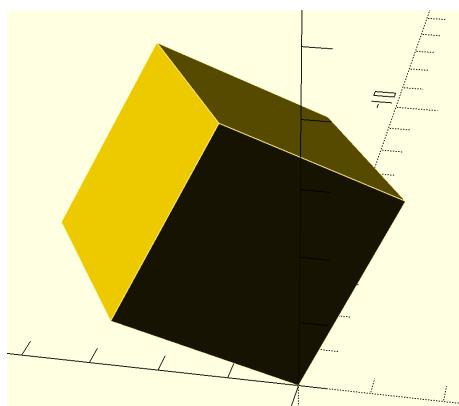
```
rotate([30, 0, -60]) {  
    cube(3, center=true);  
}
```



Cube centré et rotation

Si, comme dans l'exemple précédent, dans `cube()`, vous spécifiez `center=true`, le centre de rotation est le centre du cube, sinon c'est son sommet à l'origine :

```
rotate([30, 0, -60]) {  
    cube(3);  
}
```



Cube non centré et rotation

La fonction `rotate()` s'utilisent parfois avec deux paramètres. Dans l'exemple suivant, on réalise une rotation du cube de 60° autour de l'axe formé par le vecteur (1, 0, 0) :

```
rotate(60, [1, 0, 0]) cube(3, center=true);
```

Ce qui est équivalent au code suivant :

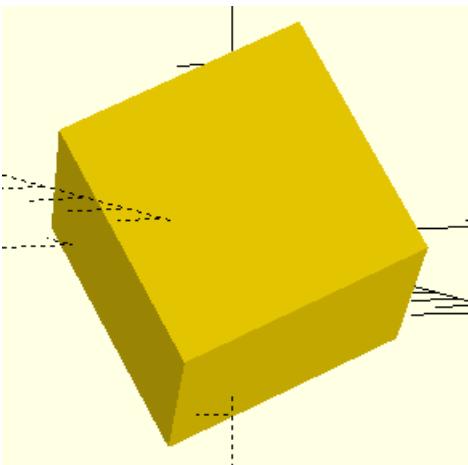
```
rotate([60, 0, 0]) cube(3, center=true);
```

Par contre, les deux codes suivant ne donnent pas le même résultat :

```
rotate(60, [1, 1, 0]) cube(3, center=true);  
rotate([60, 60, 0]) cube(3, center=true);
```

En effet, la première ligne fait tourner le cube de 60° autour de l'axe définit par le vecteur (1, 1, 0) alors que la seconde ligne fait tourner le cube de 60° autour de l'axe X et de 60° autour de l'axe Y. Cette seconde ligne est alors équivalente à :

```
rotate([0, 60, 0]) rotate([60, 0, 0]) cube(3, center=true);
```

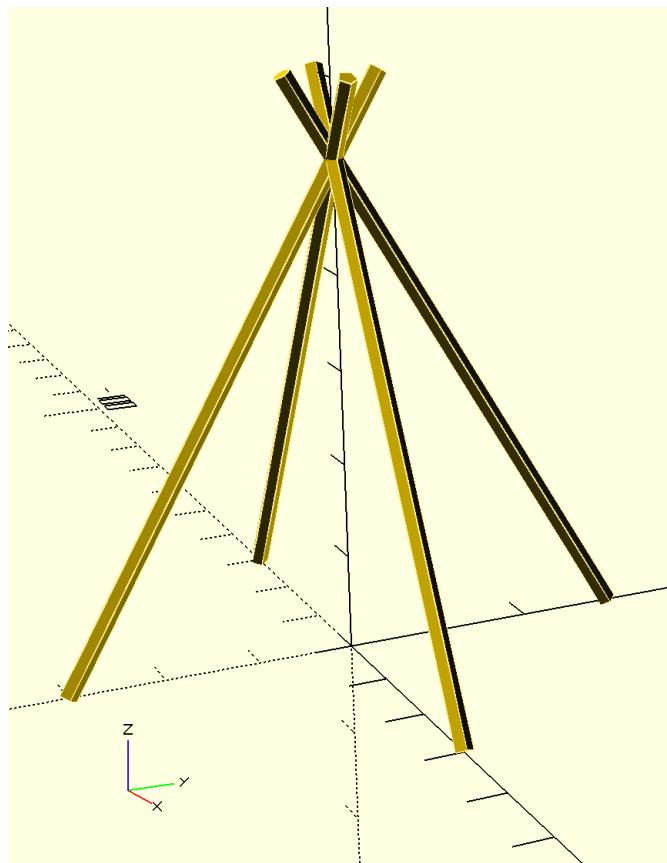


Combinaison de rotations

Nous voyons avec ce dernier exemple que l'on peut « enchaîner » les transformations. Cependant, faites très attention à l'ordre dans lequel vous écrivez les fonctions `rotate()`, il est souvent primordial.

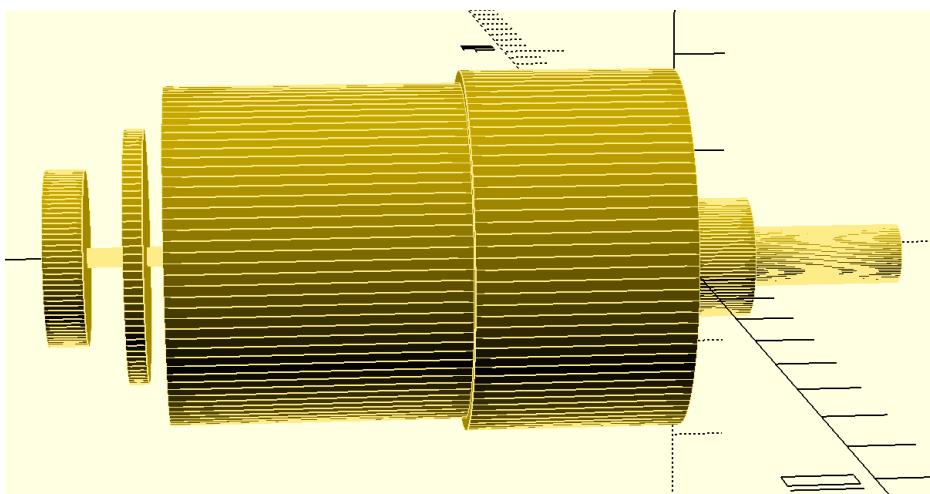
Exercices :

En combinant les translations et les rotations, réalisez le squelette d'un tipi indien (avec 4 poteaux) qui ressemble à l'illustration suivante :



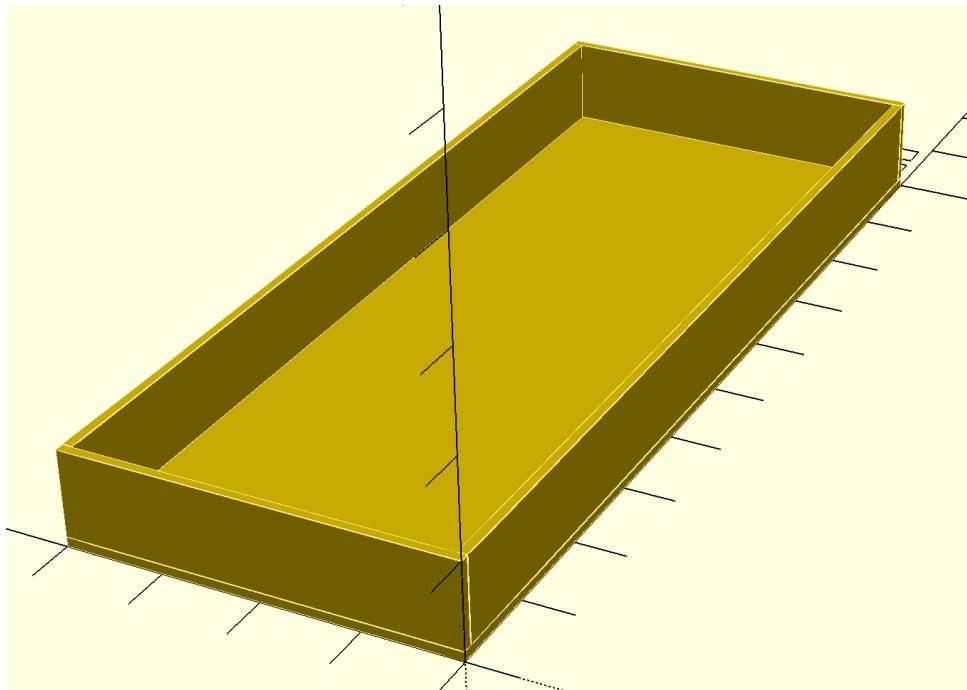
Squelette d'un tipi indien

Plus complexe, réalisez la maquette d'un petit moteur :



Maquette d'un moteur

Essayez de construire à partir de quatre paramètres (largeur, longueur, hauteur, épaisseur), une petite boîte rectangulaire similaire à celle-ci (attention, si vous changez un paramètre, la boîte doit toujours être une boîte) :



Une boîte paramétrable

3.3. Redimensionnement

La redimensionnement permet de modifier la taille d'un objet ou d'un ensemble d'objets. C'est la fonction `scale()` qui va être utilisée avec un paramètre de type vecteur. Vous pouvez donc par exemple écrire :

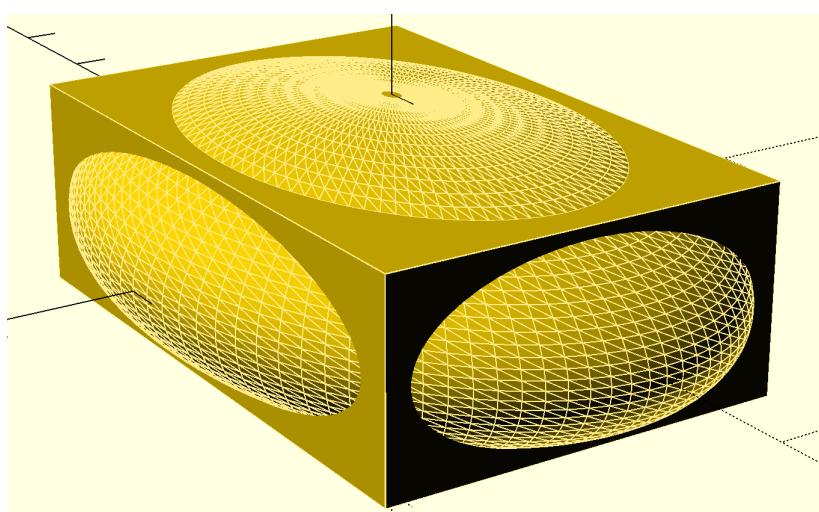
```
scale( [10, 0.5, 1] ) cube(1);
```

Cet exemple n'est pas très intéressant puisque nous pouvons avoir le même résultat avec :

```
cube( [10, 0.5, 1] );
```

Le `scale()` est surtout intéressant sur des groupes d'objets et s'utilise de la même façon que `translate()` ou `rotate()`, par exemple :

```
scale([3, 2, 1]) {  
    sphere(1, $fn=100);  
    cube(1.5, center=true);  
}
```



Mise à l'échelle de deux primitives

3.4. Couleurs

Les couleurs sont obtenues avec la fonction `color()` à laquelle il faut spécifier quelle couleur (en *RGB*) appliquer au modèle. Par exemple, on peut dessiner un cube vert avec les instructions suivantes :

```
color([0, 0.9, 0]) {  
    cube(1);  
}
```

Les quantités de *Red*, *Green* et *Blue* sont spécifiées entre 0 et 1.

On peut également donner un deuxième paramètre qui correspond à l'alpha (la transparence). Si on désire une transparence moyenne de 0.5, le code devient alors :

```
color([0, 0.9, 0], 0.5) cube(1);
```

Remarque :

L'alpha peut également être intégré dans la couleur (*RGBA*) :

```
color([0, 0.9, 0, 0.5]) cube(1);
```

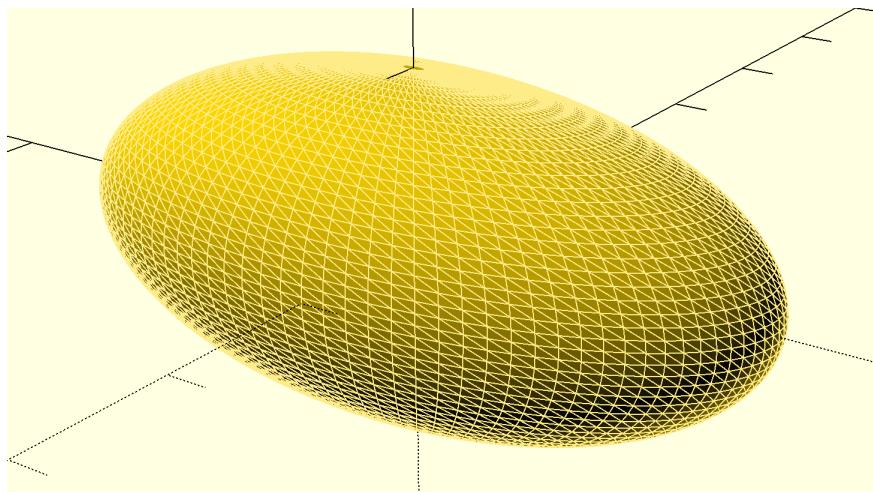
Certaines couleurs, telles que « *Blue* », « *Red* », ... sont définies dans *OpenScad*, la liste complète peut être obtenue à l'adresse https://en.m.wikibooks.org/wiki/OpenSCAD_User_Manual/Transformations#color .

On écrit alors simplement :

```
color("Cyan") cube(1);
```

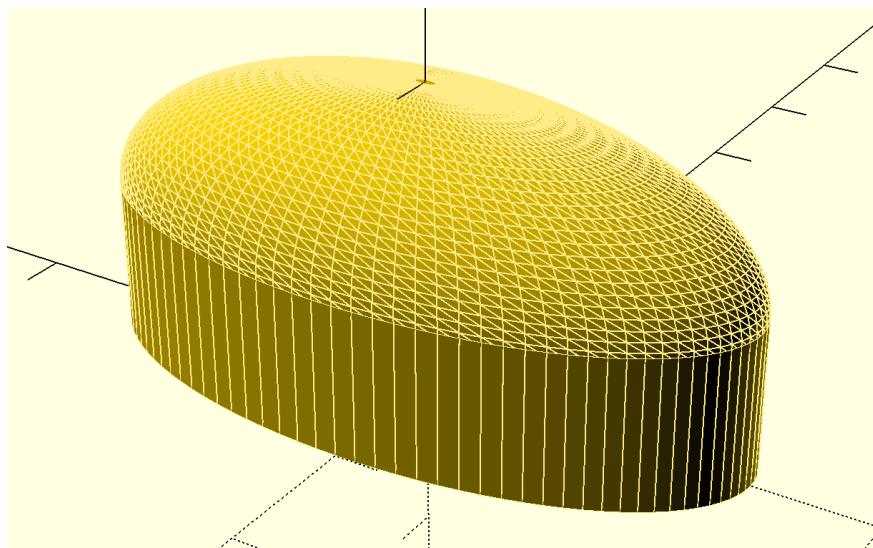
Exercices :

- Avec la fonction `scale()`, dessinez un ballon de rugby dans le genre de l'image ci-dessous.



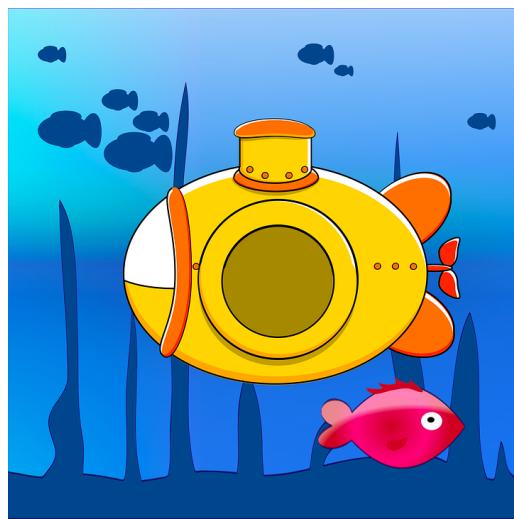
Ballon de Rugby

- Avec les fonctions `scale()` et `translate()`, réalisez cette fois, le stade de rugby.



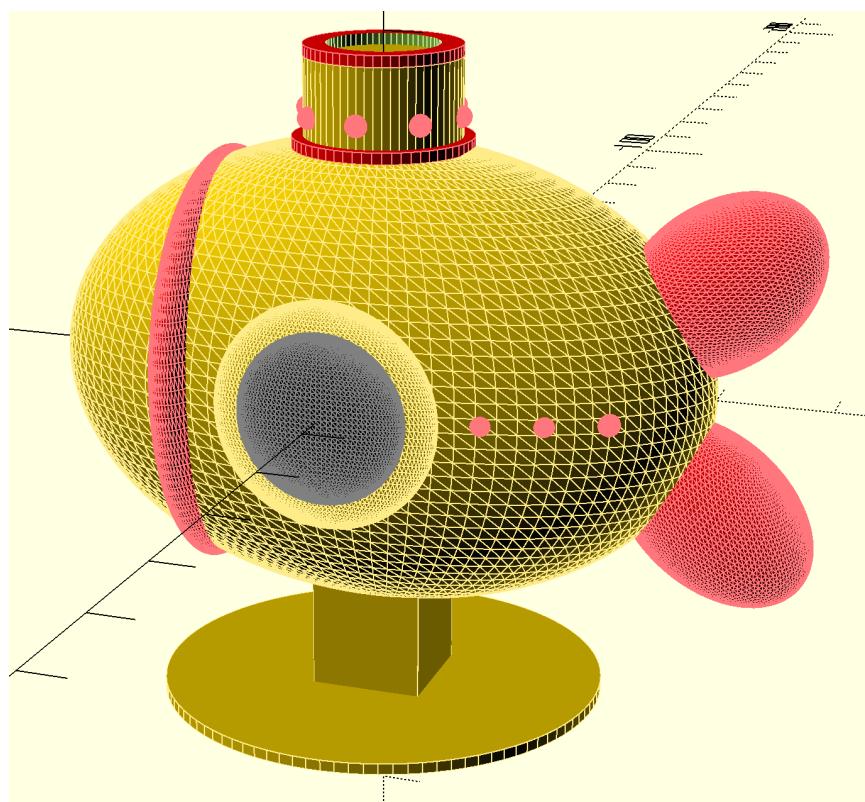
Stade de Rugby

-
- En combinant tout ce qui a été vu pour l'instant, réalisez un modèle qui pourrait ressembler à un sous-marin jaune. Vous pouvez vous inspirer de l'image suivante :



Yellow Submarine

La solution de *Laury Hughes* :



Le Yellow Submarine en 3D

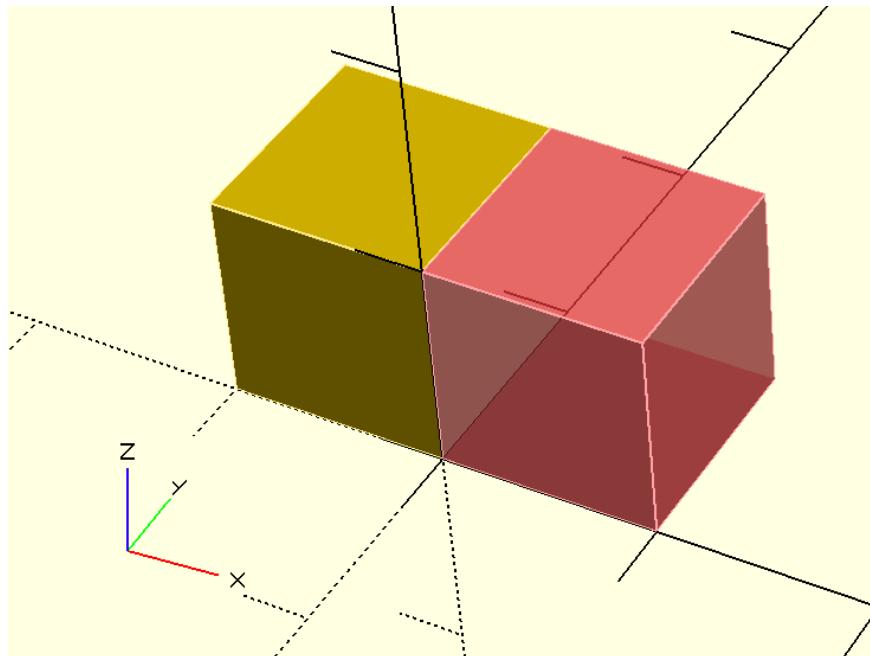
3.5. Symétrie

Pour réaliser un « effet miroir » on utilise la fonction *mirror()*. Elle va déplacer un objet ou un ensemble d'objets symétriquement par rapport à un plan déterminé par son axe (un vecteur normal) et passant par l'origine. Le vecteur normal doit être passé en paramètre à la fonction *mirror()*.

Pour visualiser cette symétrie, dans l'exemple suivant, nous allons dessiner d'abord le cube original (en rouge grâce au #) et ensuite le cube en « miroir » par rapport au plan oYZ perpendiculaire à l'axe des X (vecteur normal $[1, 0, 0]$) et passant par l'origine :

```
#cube(1);  
mirror([1, 0, 0]) {  
    cube(1);  
}
```

Le résultat est le suivant :



Symétrie appliquée à un cube

4. Algèbre de Boole

Les opérations booléennes sont appliquées entre des formes primitives pour réaliser d'autres formes plus complexes. Nous allons voir les trois opérations **UNION**, **DIFFERENCE** et **INTERSECTION**. Pour appliquer une de ces opérations, il suffit de spécifier son nom et entre parenthèses les formes sur lesquelles elle doit s'appliquer :

```
OPERATION () { forme1; forme2; }
```

Les opérations **UNION** et **INTERSECTION** sont commutatives (l'inversion des opérandes ne change pas le résultat).

```
union () { forme1; forme2; }
```

est équivalent à

```
union () { forme2; forme1; }
```

```
intersection () { forme1; forme2; }
```

est équivalent à

```
intersection () { forme2; forme1; }
```

La **DIFFERENCE** n'est pas commutative, il faudra donc bien faire attention à l'ordre des opérandes.

```
difference () { forme1; forme2; }
```

n'est pas équivalent à

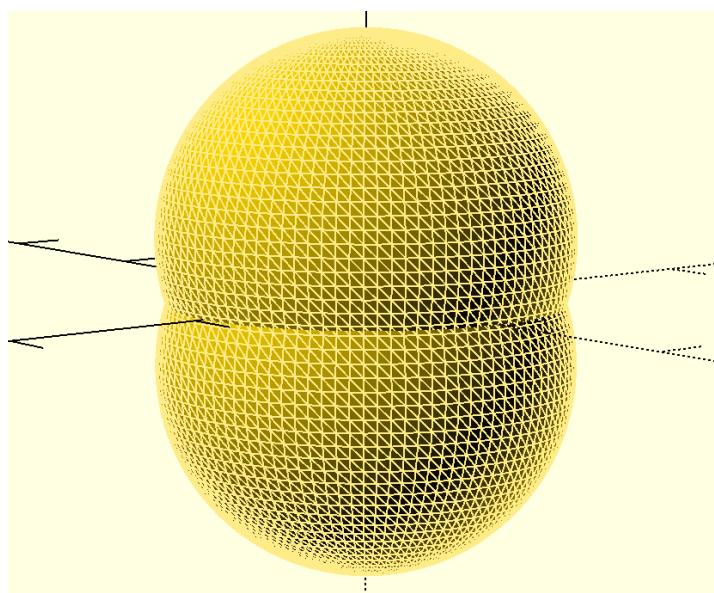
```
difference () { forme2; forme1; }
```

4.1. Union

L'union de deux formes va donner un nouveau modèle qui reprend l'ensemble des éléments des deux formes. Par exemple, l'opération :

```
union() {  
    translate ([0, 0, 2]) sphere(10, $fn=100);  
    translate ([0, 0, -2]) sphere(10, $fn=100);  
}
```

Donne le résultat suivant :



Union de deux sphères

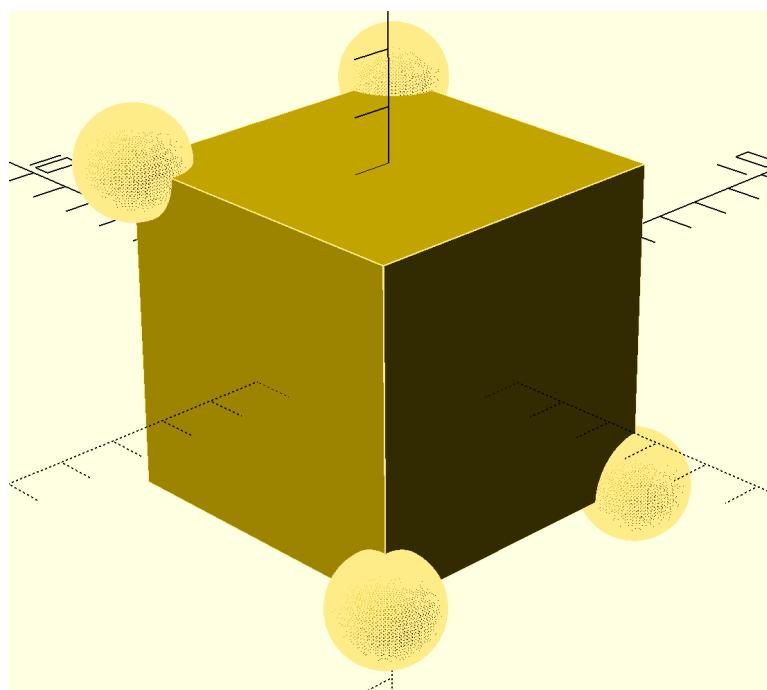
Remarques :

- Vous pourriez penser que le résultat ci-dessus est le même que si on n'avait pas fait l'*UNION*. Visuellement, c'est vrai, l'aperçu est le même, mais après un *UNION*, vous créez un nouveau modèle qui forme un tout alors que sans l'*UNION* vous avez juste deux objets superposés.
- On peut réaliser l'*UNION* de plus de 2 formes en une seule instruction :

```
union (forme1, forme2, forme3, ...);
```

Exercice :

Réaliser une nouvelle forme qui ressemble à ceci :



Le cube à boules

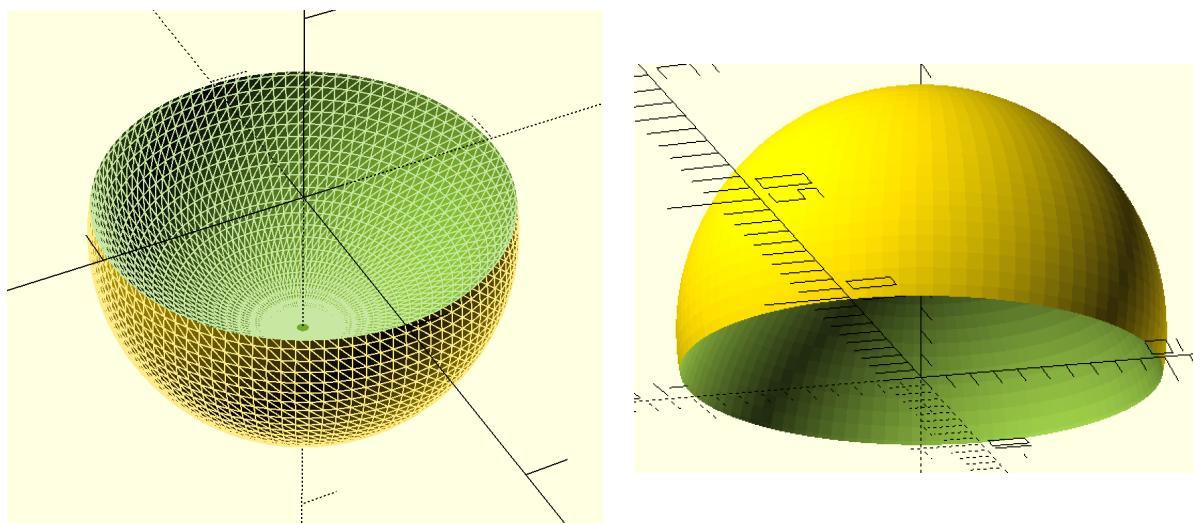
4.2. Différence

La différence entre deux formes donne un nouveau modèle qui reprend l'ensemble des éléments de la première forme de laquelle on retire tous les éléments de la seconde.

Par exemple, l'opération :

```
difference() {  
    translate ([0, 0, -2]) sphere(10, $fn=100);  
    translate ([0, 0, 2]) sphere(10, $fn=100);  
}
```

Donne le résultat suivant :



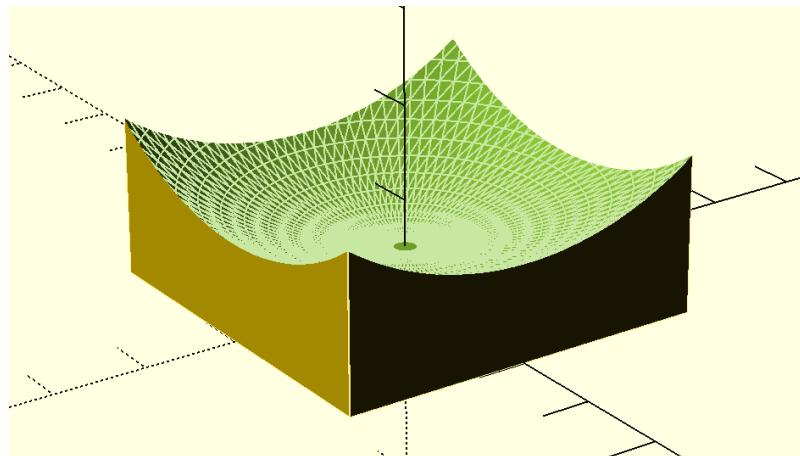
Différence de deux sphères

Remarque :

Comme nous l'avons déjà signalé la forme résultante dépend de l'ordre des opérandes. L'opération *difference (forme1, forme2)* ne donne pas le même résultat que *difference (forme2, forme1)*; Les résultats des deux opérations sont montrés ci-dessus.

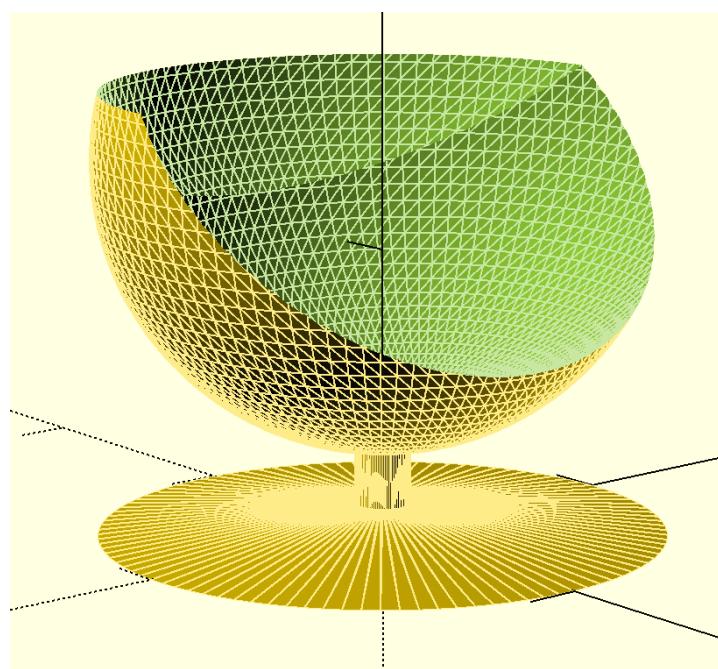
Exercices :

Réalisez une forme qui ressemble à un « cendrier » :



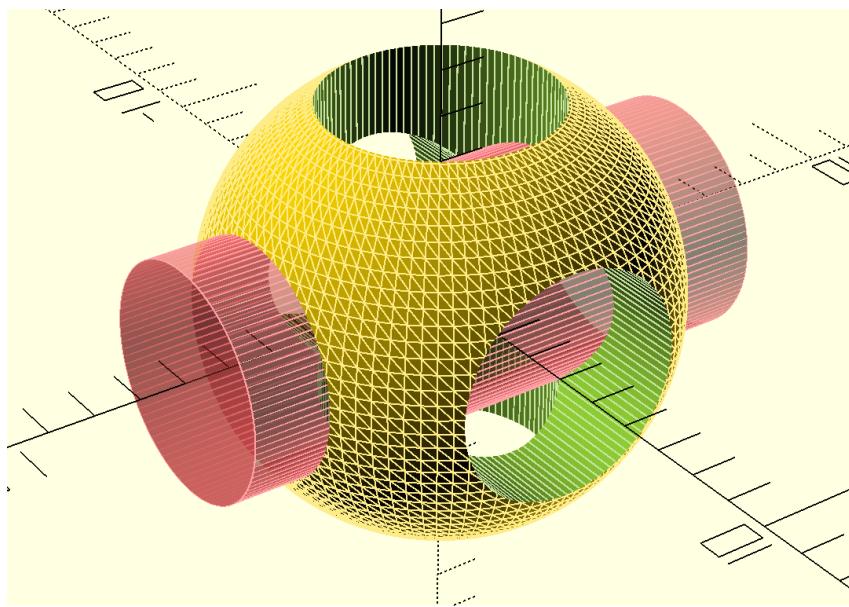
Le cendrier

Réalisez un modèle qui ressemble à un « siège de salon contemporain »



Le siège contemporain

Réalisez (sans tricher ;-)) le logo OpenScad :



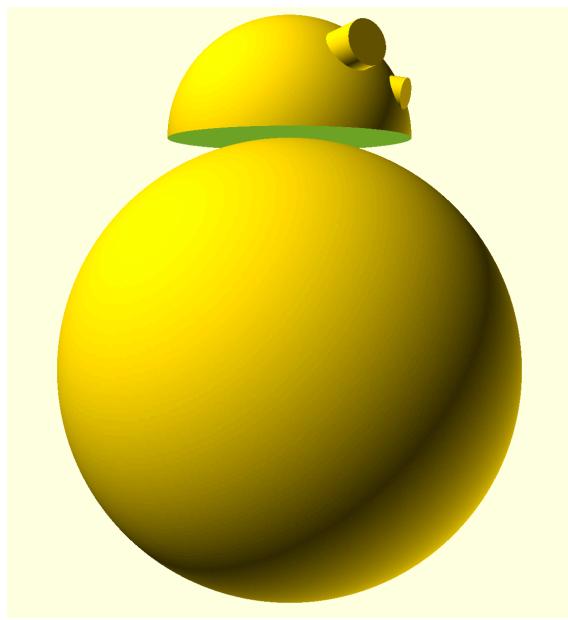
Logo OpenScad

Réaliser le petit robot de *Star Wars BB8* :



BB8¹

La solution de *Wouter Gordts* :



Une solution pour BB8

¹ https://www.pngkey.com/detail/u2q8o0i1a9e6r5y3_star-wars-bb8-clipart-star-wars-bb8-cartoon/

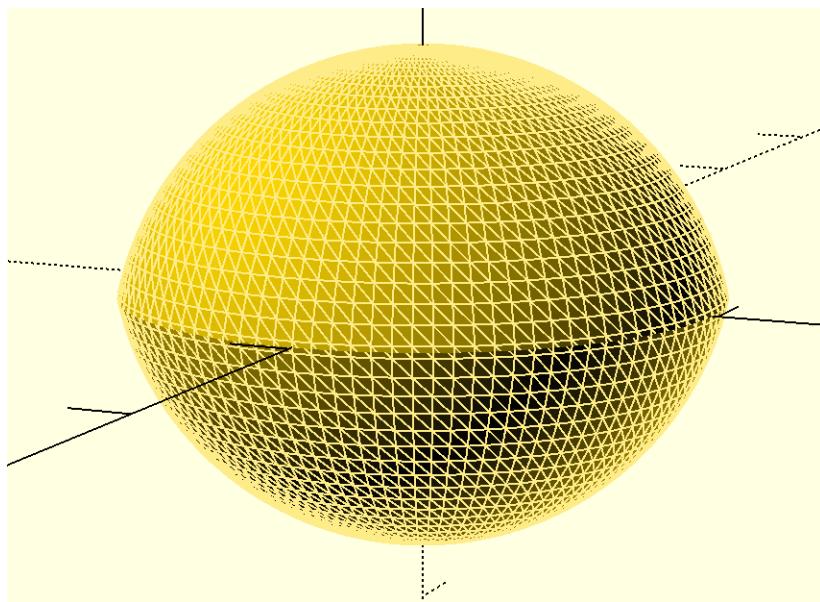
4.3. Intersection

L'intersection entre deux formes donne un nouveau modèle qui reprend l'ensemble de tous les éléments communs aux deux formes.

Par exemple, l'opération :

```
intersection() {  
    translate ([0, 0, -2]) sphere(10, $fn=100);  
    translate ([0, 0, 2]) sphere(10, $fn=100);  
}
```

Donne le résultat suivant :



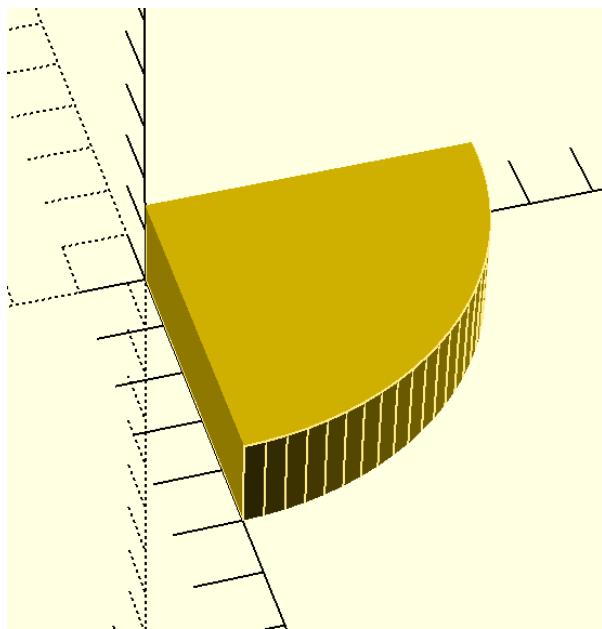
Intersection de deux sphères

Remarque :

On peut réaliser l'*INTERSECTION* de plus de 2 formes en une seule instruction :
intersection (forme1, forme2, forme3, ...);

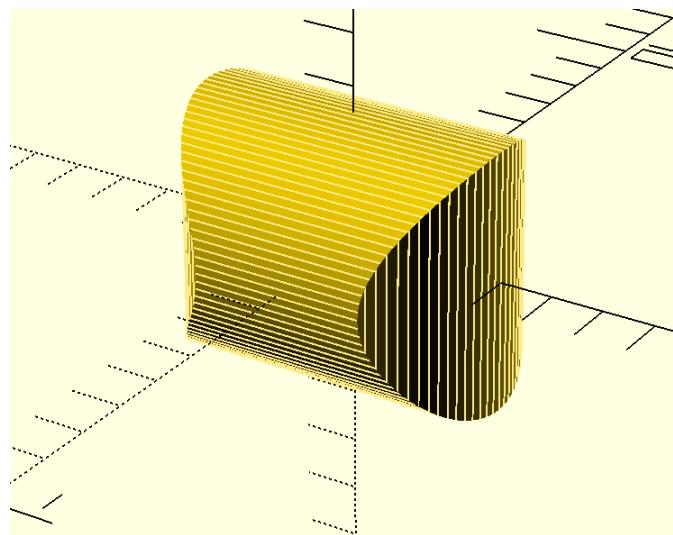
Exercices :

Réalisez une nouvelle forme en « quart de tarte » :



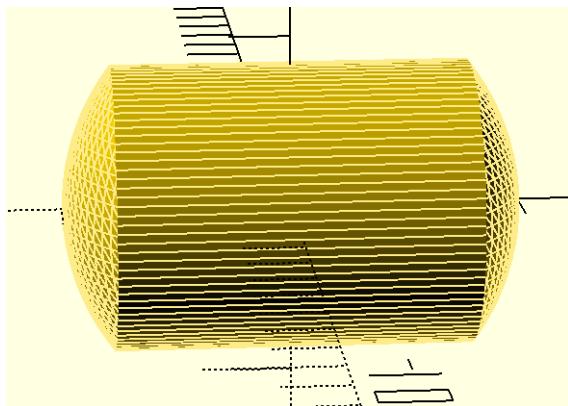
Le quart de tarte

Réalisez un petit « osselet » :



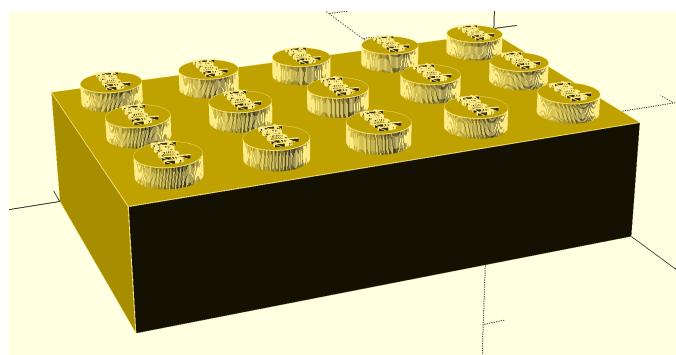
Le petit osselet

Réalisez la base d'une capsule spatiale, dans ce genre :



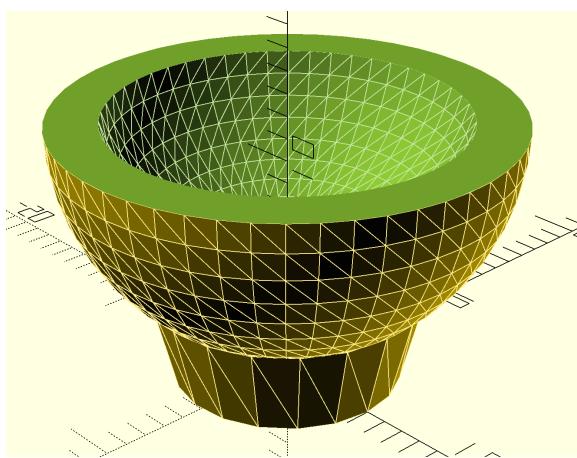
Base de la capsule spatiale

Exercice complémentaire (sans solution) :



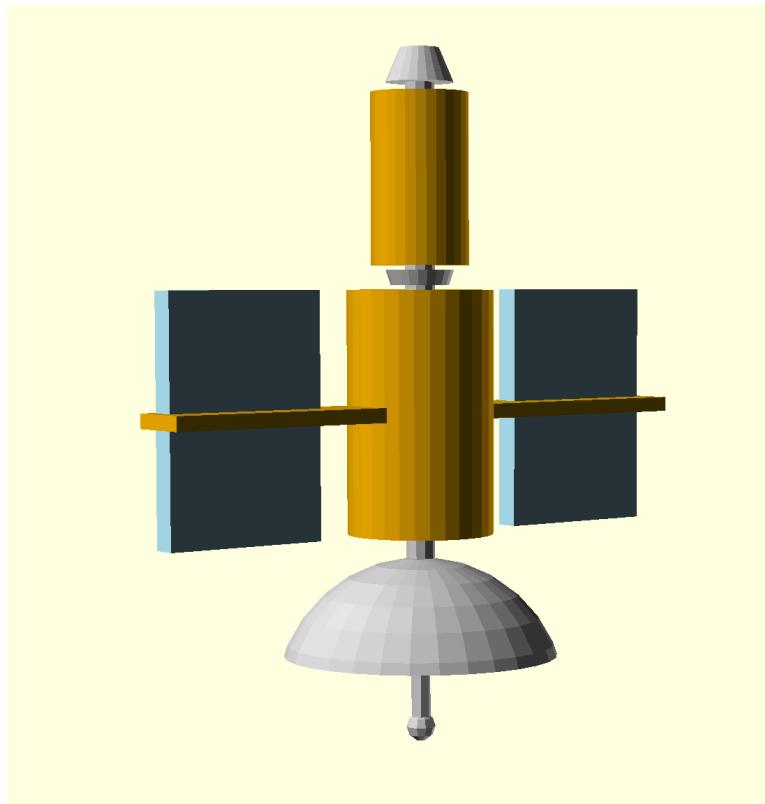
Fausse pièce Légo

Réaliser un petit bol tibétain :



Bol tibétain

Réaliser un satellite avec ses panneaux solaires :



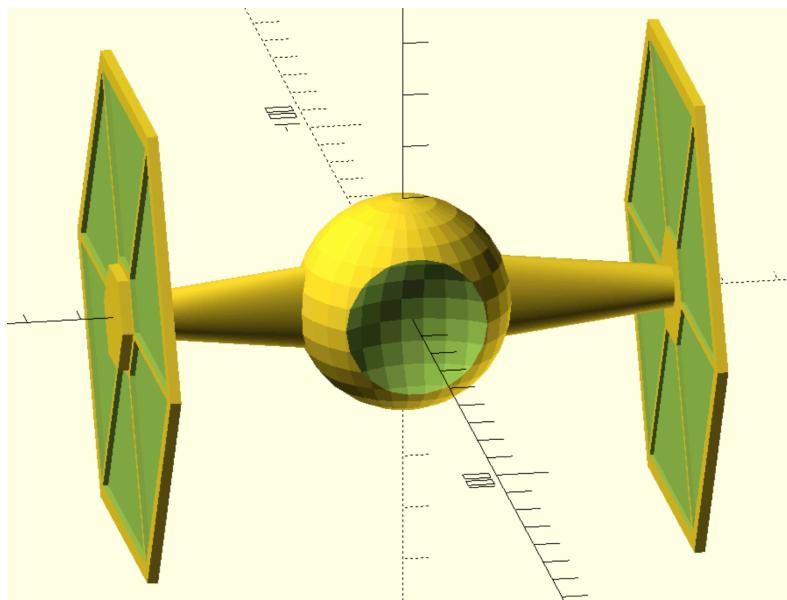
Satellite

Réaliser un des vaisseaux de « Star Wars » :



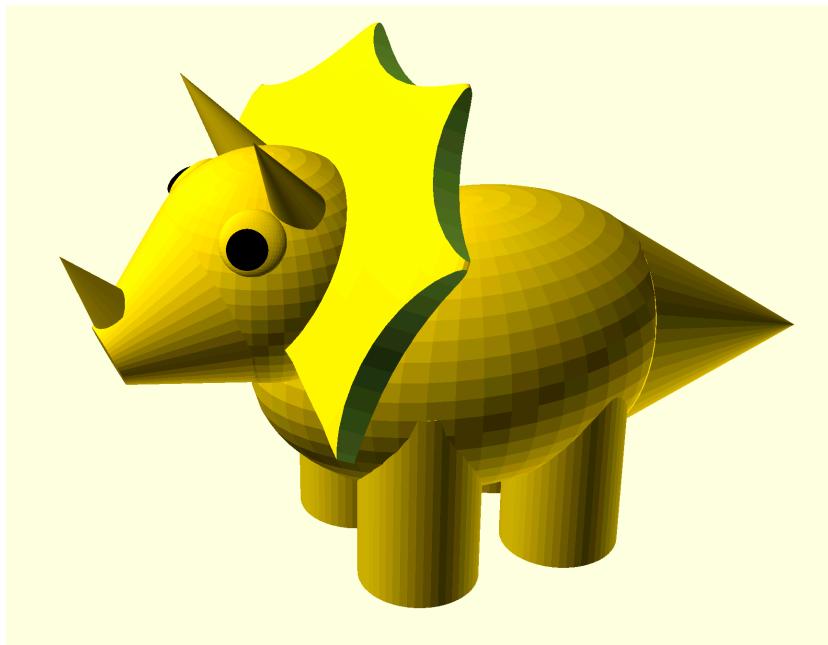
Vaisseau « Star Wars »

Ou alors :



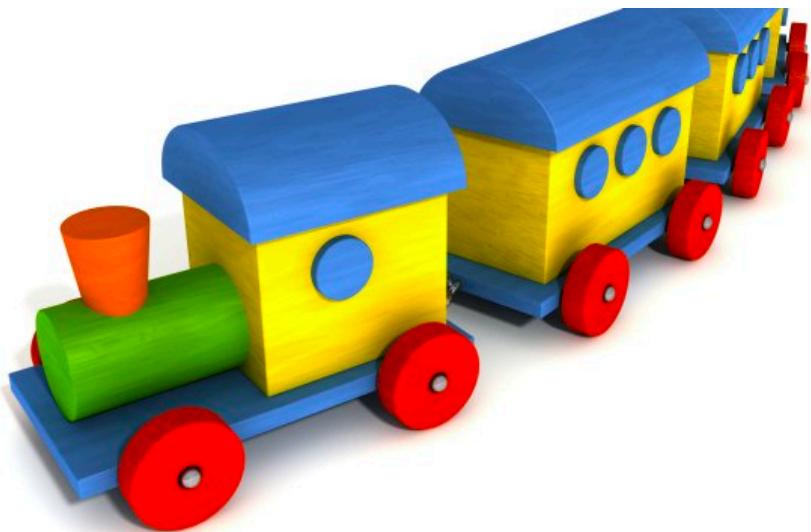
Autre vaisseau « Star Wars »

Réaliser un tricératops :



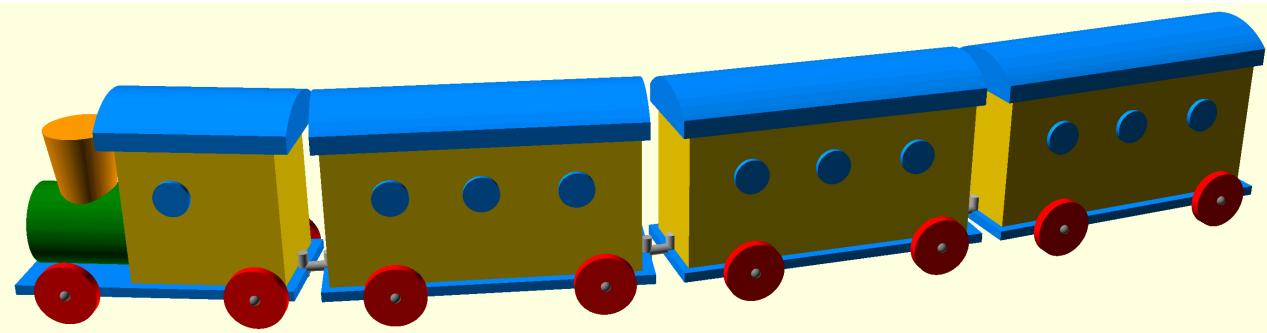
Tricératops

Réaliser un petit train (avec les wagons en option) :



Modèle pour le Petit train

Proposition de solution pour le petit train :



Solution du Petit train

Pour les plus ambitieux vous pouvez tenter un palais oriental :



Palais oriental

5. Opérations géométriques

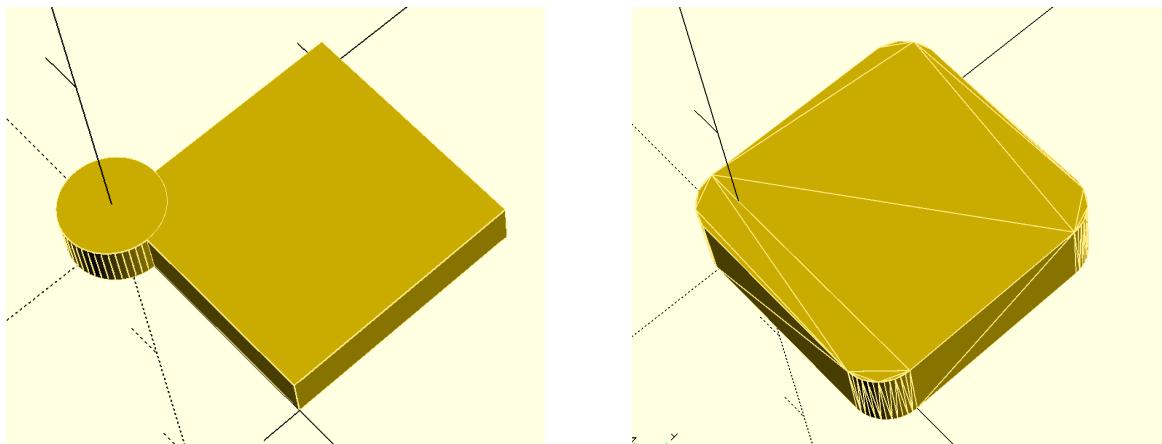
Il existe d'autres opérations qui permettent de créer des formes particulières à partir de la combinaison ou l'extension (aussi appelée dilatation) de formes élémentaires. Nous allons voir deux de ces « opérations géométriques », celle de *Hull* et celle de *Minkowski*.

5.1. Somme de Minkowski

La somme de *Minkowski* est relativement difficile à expliquer car elle nécessite des connaissances relativement avancées en mathématique. Nous allons donc l'aborder de manière pragmatique en visualisant son résultat à travers différents exemples.

Commençons simplement par dessiner un cube et un cylindre :

```
$fn=50;  
cube([10,10,2]);  
cylinder(r=2,h=2);
```



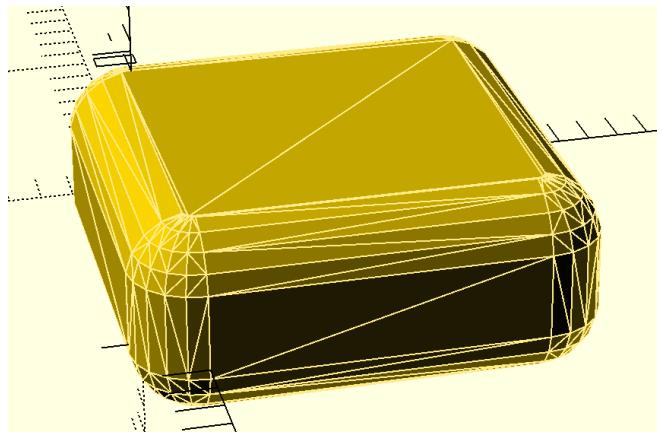
Avant/Après la somme de Minkowski

L'idée de l'addition de *Minkowski* est d'appliquer la forme du cylindre sur chacun des coins du cube. L'opération s'écrit de la manière suivante :

```
$fn=50;  
minkowski() {  
    cube([10,10,2]);  
    cylinder(r=2,h=2); }
```

En faisant la même chose mais avec une sphère on obtient des coins encore plus arrondis :

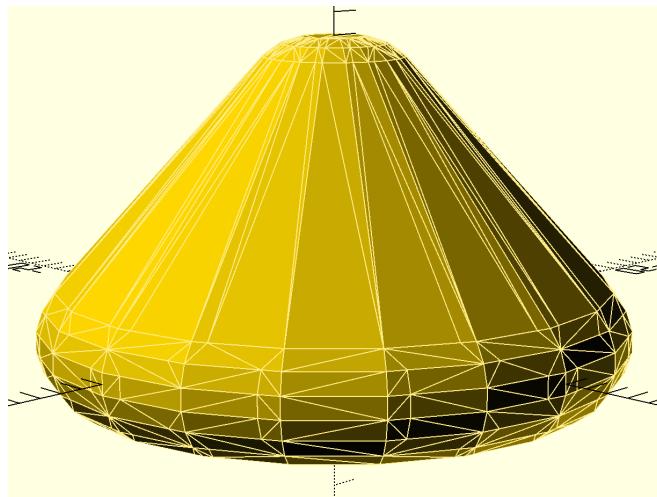
```
$fn=20;  
minkowski() {  
    cube([10,10,2]);  
    sphere(2); }
```



Somme de Minkowski avec une sphère

Ou encore avec un cylindre à la place du cube, on obtient :

```
$fn=20;  
minkowski() {  
    cylinder(10,10);  
    sphere(3); }
```

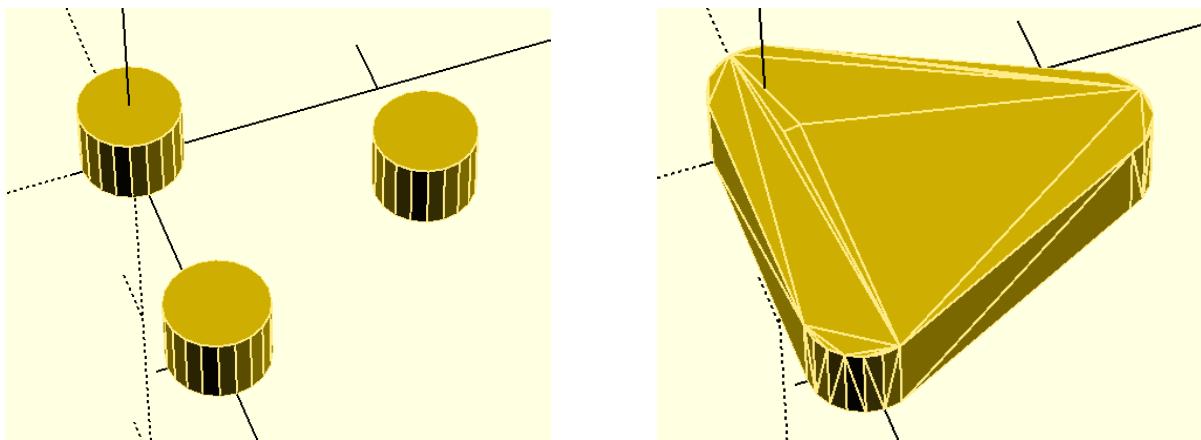


Somme de Minkowski avec une cylindre et sphère

5.2. Convex Hull

Nous allons de la même manière aborder les formes convexes de *Hull*. Vous positionnez plusieurs formes, éloignées l'une de l'autre et l'opération de *Hull* va les « rassembler en une seule forme. Par exemple, positionnez quelques cylindres distants les uns des autres :

```
$fn = 20;  
translate([0,0,0]) cylinder(r=2,h=3);  
translate([10,0,0]) cylinder(r=2,h=3);  
translate([5,10,0]) cylinder(r=2,h=3);
```



Avant/Après l'opération de Hull

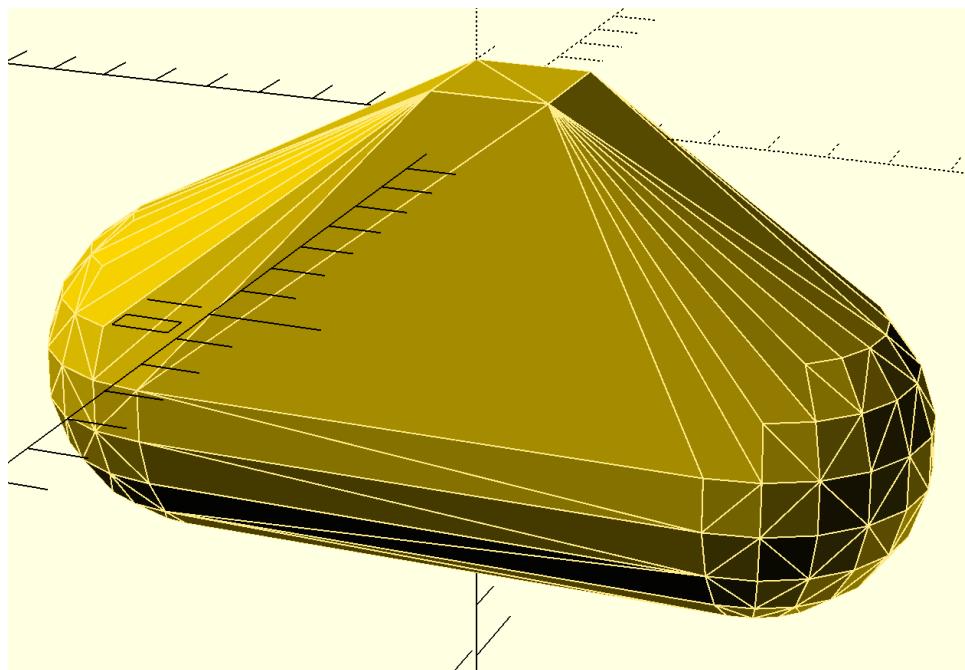
Appliquez ensuite la transformation convexe de *Hull* en écrivant :

```
$fn = 20;  
hull () {  
    translate([0,0,0]) cylinder(r=2,h=3);  
    translate([10,0,0]) cylinder(r=2,h=3);  
    translate([5,10,0]) cylinder(r=2,h=3);  
}
```

L'idéal pour apprêhender ces deux opérations est de les pratiquer, les tester pour visualiser leur effet et résultat à travers des exemples ou exercices.

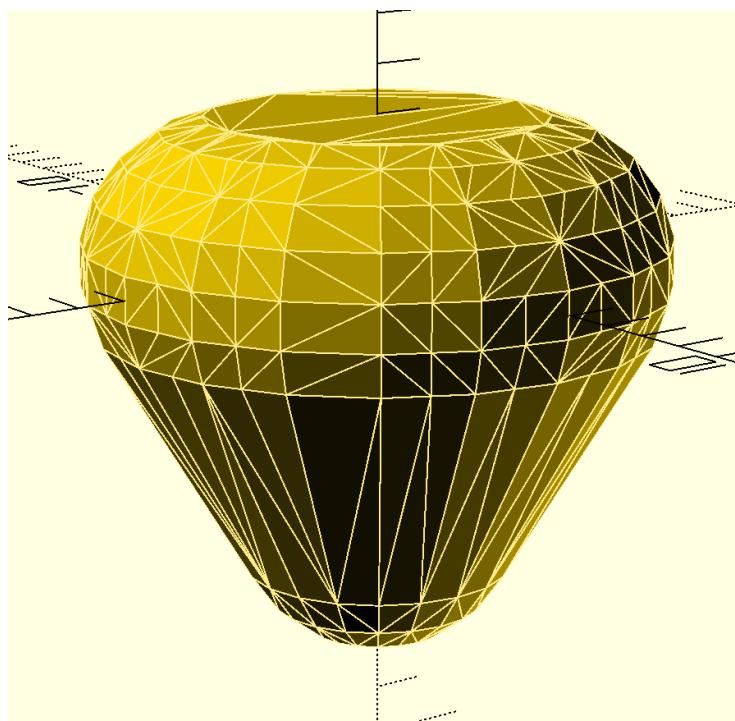
Exercices :

Réalisez la forme suivante avec la fonction *hull()*:



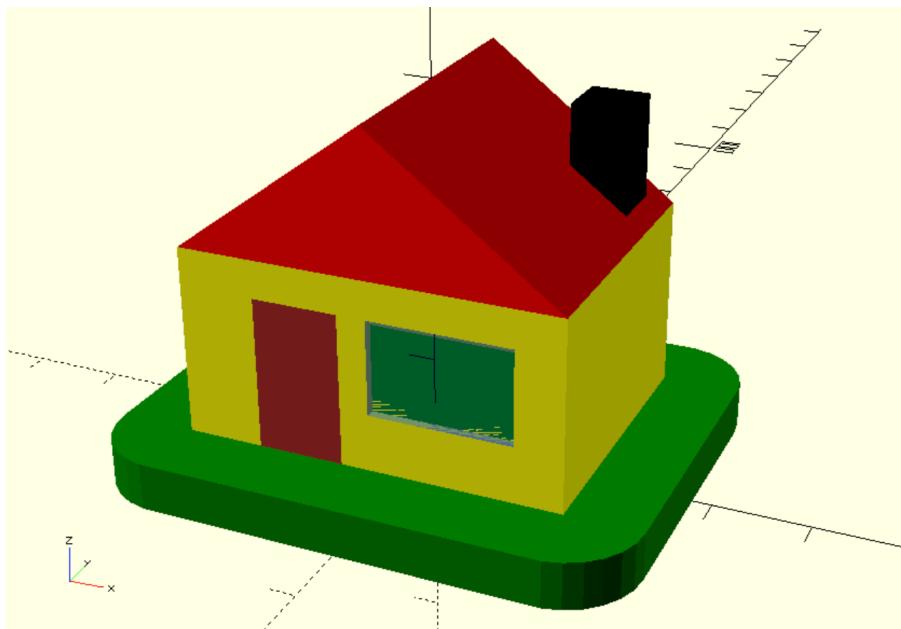
Donnez un nom à cet objet

Réalisez la forme de la montgolfière avec la fonction *minkowski()*, vous pouvez faire mieux encore en lui ajoutant une petite nacelle :



Montgolfière

Réalisez une petite maison sur son support :



Maison

6. Modules

Pour augmenter l'intérêt de ce langage et permettre la réutilisation de formes ou de codes existants, on peut créer des modules et des fonctions qui permettent de rendre vos programmes plus lisibles, plus courts et plus efficaces. On peut également importer des modèles « tout prêts » et les intégrer dans nos projets.

Les module et fonction ont une syntaxe et une utilisation très similaire, ils regroupent un ensemble d'instructions sous un nom (le nom de la fonction ou du module) qui est réutilisé dans un programme pour éviter de chaque fois devoir répéter un ensemble de lignes de code.

Ce qui différencie une fonction d'un module c'est son domaine d'application. Une fonction est utilisée pour réaliser des calculs, des opérations qui ne dessinent rien alors qu'un module va générer des modèles.

6.1. Fonction

Une *function* représente un ensemble d'instructions paramétrables et réutilisables dans un même programme et destinés à réaliser une opération qui ne réalisent pas de « rendu » (pas de générations d'objets 3D) mais qui renvoie un résultat. Par exemple, vous pouvez écrire une fonction qui réalise le calcul du rayon à partir de la circonference d'un cercle avec la syntaxe suivante :

```
function rayon(circonference) = circonference / (2*3.14150);
```

Pour appeler la fonction on peut ensuite simplement faire :

```
echo("Rayon = ", rayon(10));
```

On peut faire passer des vecteurs comme paramètre à une fonction, par exemple, pour calculer la distance qui sépare un point de l'origine :

```
function distance(point) = sqrt((point[0]*point[0]) +
                                  (point[1]*point[1]) +
                                  (point[2]*point[2]));
```

Et appeler ensuite la fonction :

```
echo("Distance = ", distance([1, 2, 3]));
```

On peut spécifier des valeurs de paramètres par défaut avec la syntaxe :

```
function rayon(circonference=1) = circonference / (2*3.14150);
```

Si l'utilisateur appelle la fonction sans spécifier de valeur c'est la valeur par défaut (1 dans notre exemple) qui sera utilisée.

6.2. Module

Comme la fonction, le **module** représente un ensemble d'instructions paramétrables et réutilisables dans un même programme mais cette fois avec pour but de créer des objets 3D. Par exemple, vous allez écrire une instruction qui réalise un cube d'une certaine taille (trois paramètres) avec un déplacement (translation) sur l'axe z (un paramètre) :

```
translate([0,0,deplacementZ]) cube([tailleX, tailleY, tailleZ]);
```

Cette instruction ne sert à rien puisque les quatre paramètres ne sont pas définis. Nous allons donc inclure cette instruction dans la définition d'un module avec la syntaxe suivante :

```
module cubeTranslate(tailleX, tailleY, tailleZ, deplacementZ){  
    translate([0,0,deplacementZ]) cube([tailleX, tailleY,  
                                         tailleZ]);  
}
```

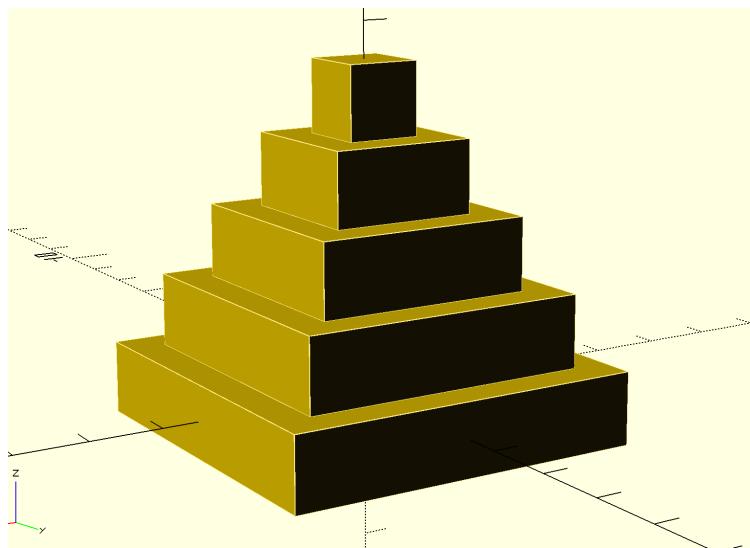
Vous remarquez que le module doit avoir un nom (*cubeTranslated* dans l'exemple) qui sert à pouvoir l'appeler dans la suite du programme. Par exemple :

```
cubeTranslated(2, 3, 4, 5);
```

Cette appel au module *cubeTranslated* va dessiner un cube de taille [2, 3, 4] et translaté de 5 mm suivant l'axe Z.

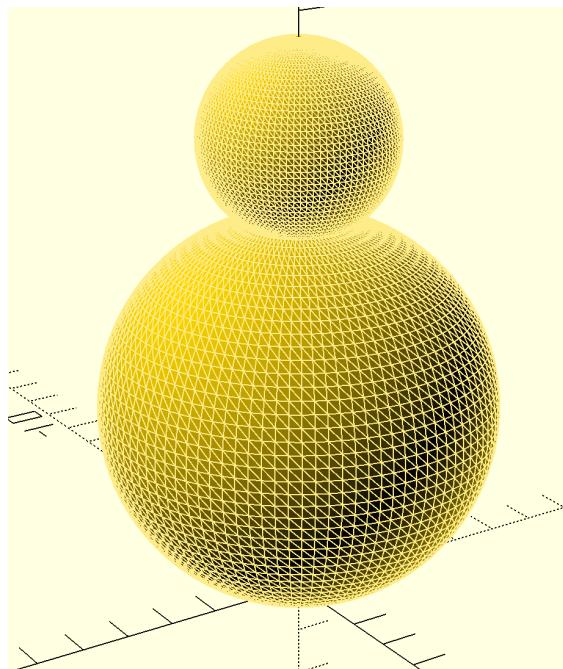
Exercices :

Modifiez légèrement le code du module *cubeTranslated* et ajoutez quelques instructions au programme pour afficher la pyramide aztèque :



Pyramide aztèque

Réalisez un module *sphereTranslated* (comme dans l'exemple ci-dessus) et créez ensuite un bonhomme de neige:



Bonhomme de neige

6.3. Include et Use

Comme dans la plupart des langages de programmation modernes, vous pouvez sauvegarder vos codes dans des fichiers qui seront réutilisés dans d'autres projets. Il suffit de sauvegarder votre module ou ensemble de modules dans un fichier qui porte un nom qui rappelle l'utilité du ou des modules réalisés. Ensuite dans un nouveau projet, il suffira d'inclure cette librairie afin de profiter des modules qui s'y trouvent. Pour inclure cette librairie sauvegardée dans votre nouveau projet, il y a deux commandes possibles :

- *include <nom_du_fichier>* : agit comme si le contenu du fichier (librairie) était copié dans le fichier du nouveau projet
- *use <nom_du_fichier>* : importe les modules et fonctions de la librairie mais n'exécute aucune commande autre que les définitions

Les fichiers « librairies » sont recherchés d'abord dans le répertoire courant (même répertoire que le fichier sauvegardé) et ensuite (si rien n'est trouvé) dans le répertoire d'installation des librairies d'*OpenScad*.

Vous écrivez donc soit :

```
include <NomDeLaLibrairie.scad>;
```

Soit :

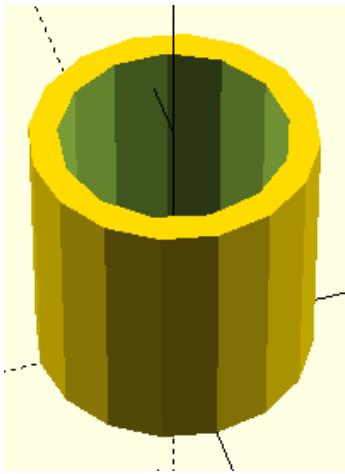
```
use <NomDeLaLibrairie.scad>;
```

Si les explications ci-dessus ne sont pas suffisamment explicites, nous allons montrer leur différence avec un exemple concret. Réalisons d'abord un fichier `ring.scad` qui contient le code suivant :

```
module ring(r1, r2, h) {
    difference() {
        cylinder(r = r1, h = h);
        translate([ 0, 0, -1 ]) cylinder(r = r2, h = h+2);
    }
}

ring(5, 4, 10);
```

Le résultat de l'exécution est le suivant :

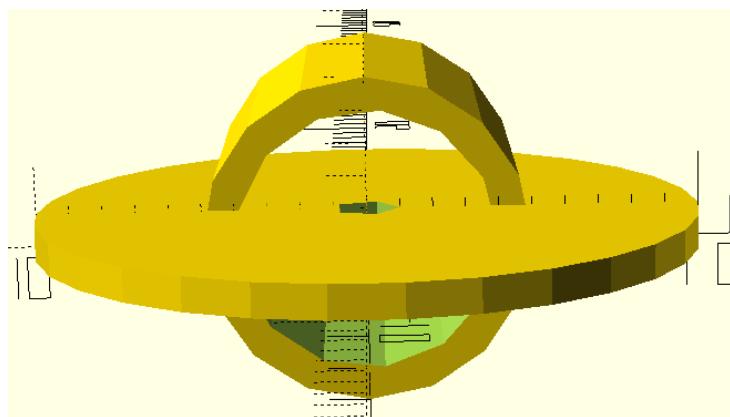


Exécution du module ring.scad

Si on écrit ensuite dans un nouveau fichier OpenScad :

```
include <ring.scad>;  
rotate([90, 0, 0]) ring(10, 1, 1);
```

Le résultat est le suivant :



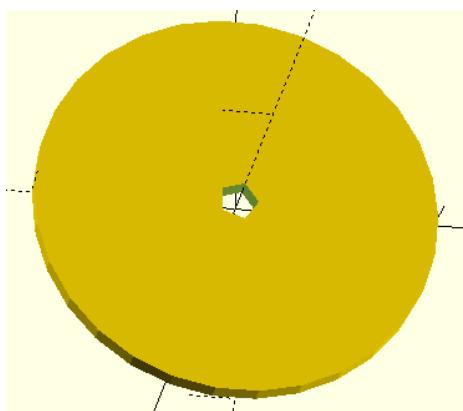
Utilisation du ring.scad avec « include »

Avec *include* le fichier *ring.scad* a complètement été intégré au nouveau fichier y compris l'instruction *ring(5, 4, 10)*. Nous avons donc deux formes résultantes : une venant du nouveau fichier et une venant de *ring.scad*.

Par contre si on écrit dans un nouveau fichier le code :

```
use <ring.scad>;
rotate([90, 0, 0]) ring(10, 1, 1);
```

Le résultat est alors le suivant :

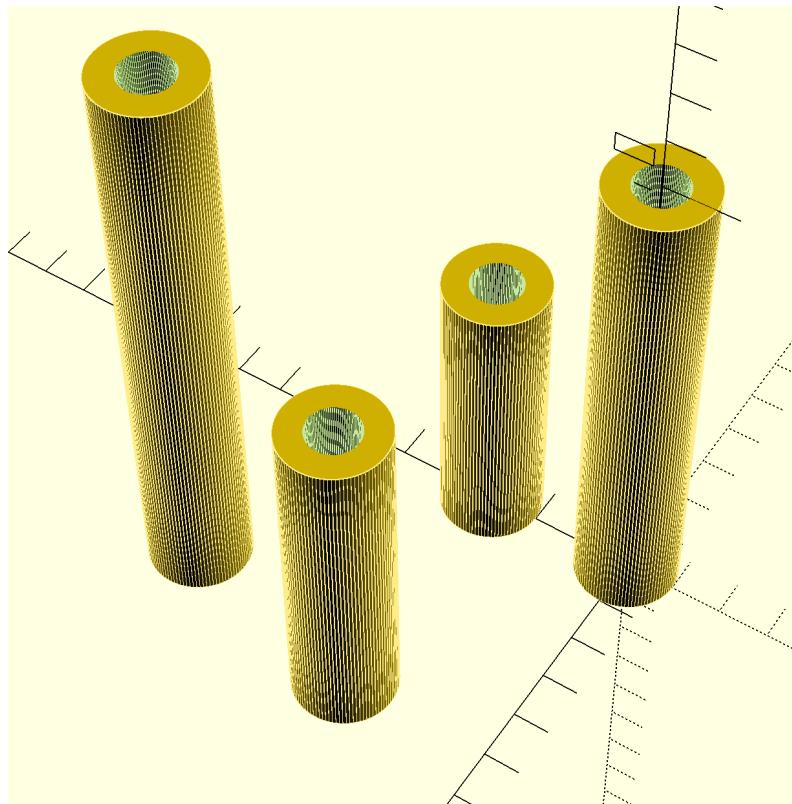


Utilisation du ring.scad avec « use »

Avec *use* seul le module défini dans *ring.scad* a été intégré dans le nouveau fichier sans l'instruction *ring(5, 4, 10)*. Nous avons donc comme résultat, une seule forme correspondant *rotate([90, 0, 0]) ring(10, 1, 1)*.

Exercice :

Réalisez un module qui réalise un tube (avec quelques paramètres), sauvegardez ce module dans un fichier que vous nommez *tube.scad* et créez ensuite un nouveau projet dans lequel vous réutilisez cette librairie (qui contient un seul module) pour dessiner quelque chose dans le genre de « la forêt de tubes » ci-dessous :



La forêt de tubes

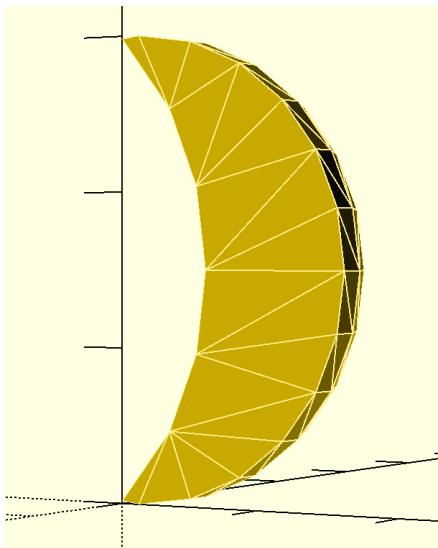
6.4. Importation

On peut dans *OpenScad* importer des fichiers *STL*, *DXF*, ... externes. Il suffit d'utiliser l'instruction *import* et de spécifier (avec le chemin) le nom du fichier.

Par exemple, pour importer le fichier *Moon.stl*² :

```
import ("/Path/Moon.stl");
```

Pour visualiser :



Importation du fichier Moon.stl

² Fichier disponible à l'adresse : <http://forums.reprap.org/file.php?88,file=354,filename=Moon.stl>

7. Boucle et Condition

Que serait un langage de programmation sans boucle et sans condition. Ces instructions vont permettre de réaliser des motifs très complexes par itérations (boucles) en y insérant (ou pas) des exceptions (conditions).

7.1. Boucle For

La boucle *for* permet de boucler sur un ensemble de valeurs spécifiées de plusieurs manière différentes :

```
for(variable = [start : increment : end])
for(variable = [start : end])
for(variable = [vector])
```

Par exemple, le code suivant :

```
for (a =[3:5])
```

Va assigner les valeurs 3, 4 et 5 à la variable *a* en trois itérations.

```
for (a =[3:0.5:5])
```

Va assigner les valeurs 3, 3.5, 4, 4.5 et 5 à la variable *a* en 5 itérations.

```
for (a =[3,1,5,8])
```

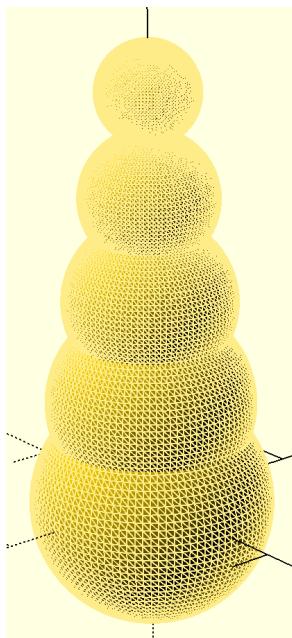
Va assigner les valeurs 3, 1, 5 et 8 à la variable *a* en 4 itérations.

Il est bien évident que l'utilisation de la boucle présente un intérêt dans le cadre de la réalisation d'objets 3D qui présentent une répétition de motifs. Par exemple, le « bonhomme de neige » que vous avez réalisé plus haut aurait pu être écrit à partir d'une itération du genre :

```
for(i=[0,1]) {  
    translate([0, 0, i*5]) sphere(r=4-2*i, $fn=100); }
```

Ce qui nous permettrait de créer un énorme bonhomme de neige sans écrire une ligne de code supplémentaire :

```
for(i=[0:4]) {  
    translate([0, 0, i*12]) sphere(r=14-2*i, $fn=100); }
```



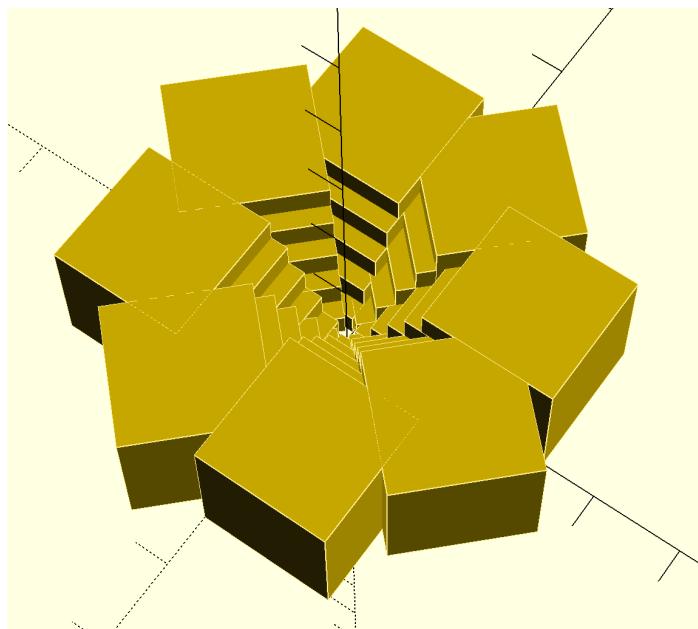
Super Bonhomme de neige

On peut évidemment encapsuler les boucles, comme dans cet exemple :

```
for(z=[-180:45:+180])  
    for(x=[10:5:50])  
        rotate([0,0,z]) translate([x,0,0]) cube(1);
```

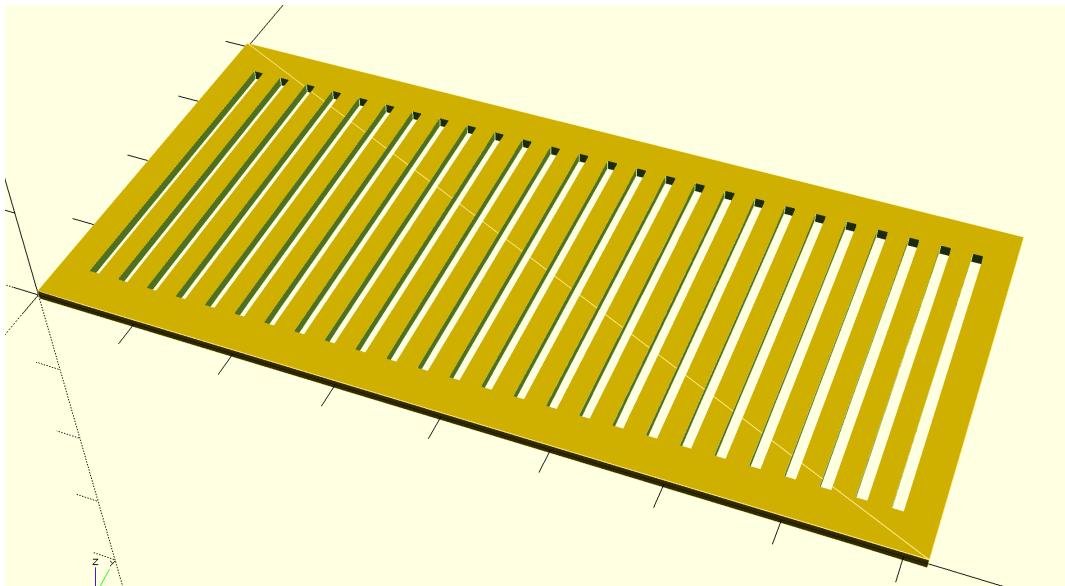
Exercices :

A partir du code ci-dessus, réalisez cet « escalier un peu fou » :



Escalier fou

Réalisez une petite grille d'aération :



Grille d'aération

Réaliser un « cube de Turner » :



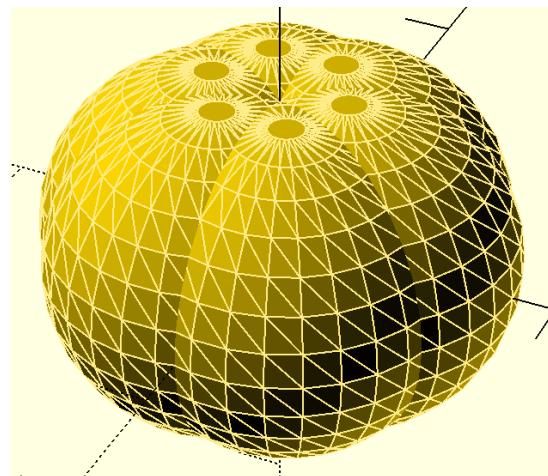
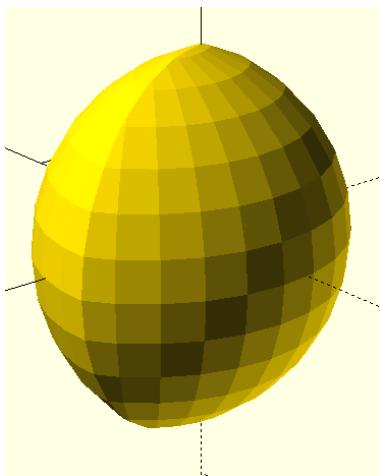
Le cube de Turner

7.2. Intersection For

On peut avec l'instruction intersection_for() itérer sur un intervalle de valeurs ou de vecteurs pour créer à chaque passe une intersection. Par exemple, le code suivant :

```
intersection_for(n = [1 : 5]) {
    rotate([0, 0, n * 36]){
        translate([4,0,0])
        sphere(r=15);
    }
}
```

Donne la forme de gauche :



Forme obtenues avec un intersection_for et intersection ... for

Alors que le code suivant donne la forme de droite :

```
intersection() {
    for(n = [1 : 6]) {
        rotate([0, 0, n * 60]) {
            translate([5,0,0])
            sphere(r=12);
        }
    }
}
```

7.3. Conditions

Le couple d'instructions *if - else* réalise un test logique pour déterminer si l'action ou le groupe d'instructions qui suivent doivent être exécutées ou pas. La syntaxe est la suivante :

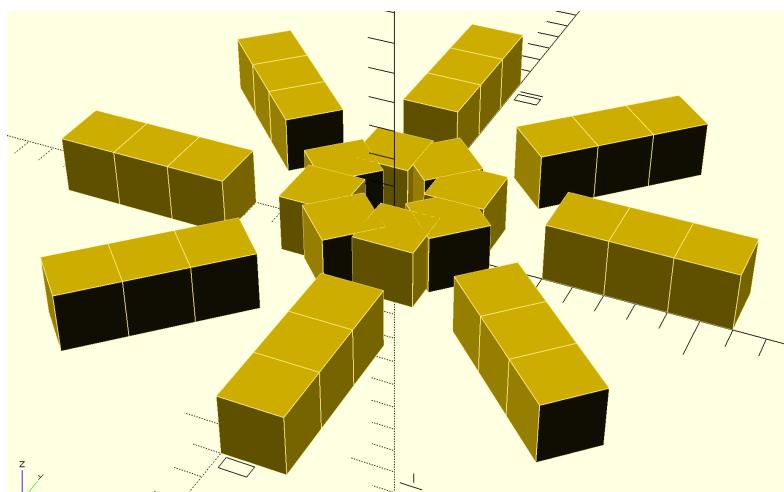
```
if (test) action  
if (test){groupe d_instruction}  
if (test) action1 else action2  
if (test){groupe d_instruction1} else {groupe d_instruction2}
```

Pour le test booléen (vrai ou faux), ce sont les opérateurs logiques traditionnels qui sont utilisés :

- == pour vérifier l'égalité
- > et < pour plus grand et plus petit
- && et || pour les ET et OU logique
- ! pour le NOT

Par exemple, on pourrait écrire :

```
for(z=[-180:45:+180])  
    for(x=[1:2:10])  
        if (x!=3) rotate([0,0,z]) translate([x,0,0]) cube(2);
```



Forme créée avec une condition

8. Extrusion

Jusqu'ici, nous avons réalisé des modèles 3D à partir de formes 3D de base. On peut obtenir des pièces 3D autrement que par « combinaison » de primitives. On peut, en effet, d'abord dessiner des formes en 2D et ensuite les « **extruder** », c'est à dire les « étirer » pour en faire des formes 3D. Un peu comme un potier part d'une forme plate et ronde en terre glaise pour ensuite l'étirer par extrusion.

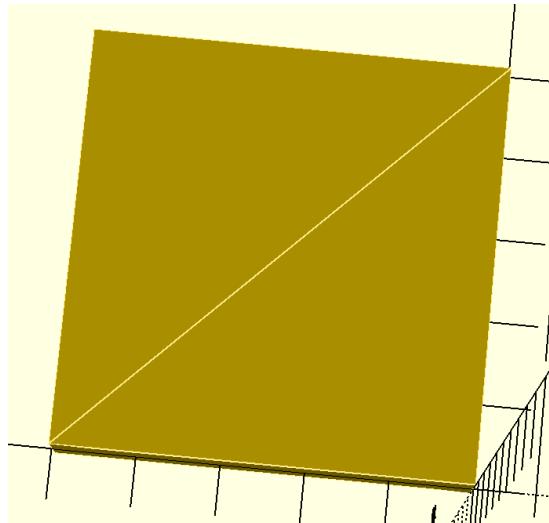
8.1. Primitives 2D

Comme en 3D, tout commence par les primitives (2D dans ce cas) : le rectangle, le cercle et le polygone.

8.1.1. Rectangle

Pour créer un rectangle on utilise l'instruction *square()* qui dessine un rectangle d'une certaine taille en x et y, centré à l'origine (ou pas).

```
square(size = [x, y], center = true/false);  
square(size = x , center = true/false);
```

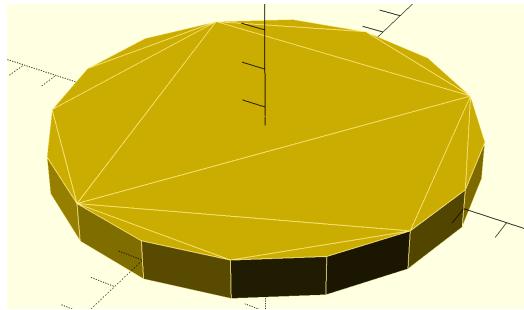


square(size = 5 , center = false);

8.1.2. Cercle

Pour dessiner un cercle centré à l'origine, on utilise l'instruction *circle()* avec comme paramètre le rayon ou le diamètre :

```
circle(r=rayon | d=diamètre);
```



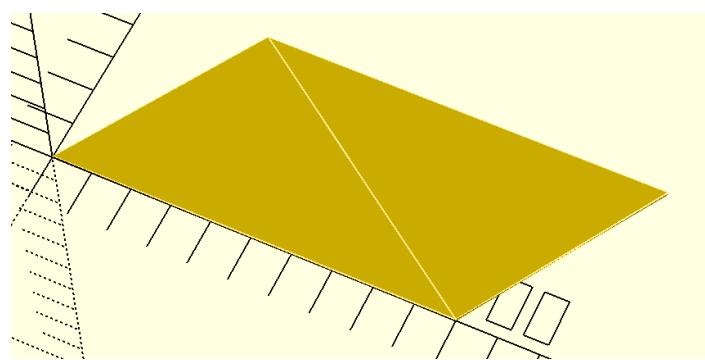
circle(r=5);

8.1.3. Polygone

Pour dessiner un **polygone**, on a deux possibilités: soit utiliser un cercle avec une résolution particulière, soit utiliser la fonction *polygon()*. La fonction *polygon()* nécessite d'abord une liste des *points* qui constituent le polygone et ensuite le chemin (*path*) qui relie ces points entre eux. On peut même placer des trous dans la forme créée. Si le *path* n'est pas précisé les points sont reliés dans l'ordre dans lequel ils sont spécifiés dans *points*.

```
polygon(points = [ [x, y], ... ],  
         paths = [ [p1, p2, p3...], ... ], convexity = N);
```

Par exemple pour réaliser un parallélogramme :



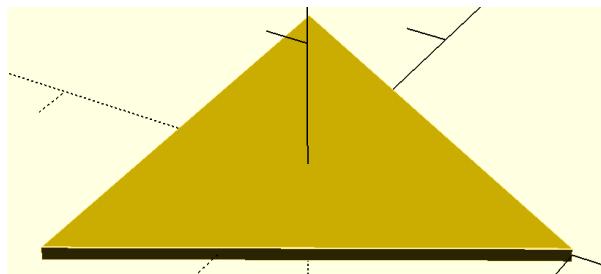
Parallélogramme à partir de la fonction polygon()

On peut écrire au choix une des deux lignes suivantes :

```
polygon(points=[[0,0],[100,0],[130,50],[30,50]]);  
polygon([[0,0],[100,0],[130,50],[30,50]], paths=[[0,1,2,3]]);
```

Pour dessiner une forme régulière comme un triangle, un pentagone, un octogone, un dodécagone, ... généralement on procédera plutôt différemment. En effet, nous avons déjà vu plus haut qu'en dessinant une forme on pouvait spécifier sa résolution (paramètre \$fn). Si nous dessinons un cercle avec une résolution de 3, nous obtiendrons donc un triangle :

```
circle(20, $fn=3);
```

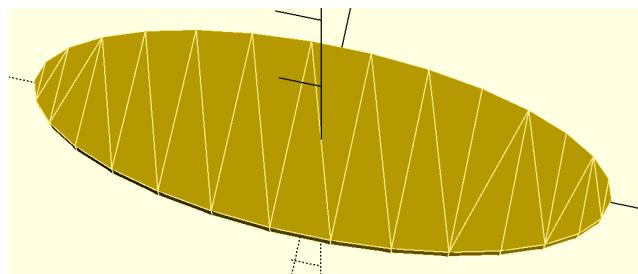


Triangle obtenu à partir d'un cercle

8.1.4. Redimensionnement

Ces trois primitives peuvent être re-dimensionnées avec la fonction *resize()*. On peut, par exemple, créer une ellipse à partir d'un cercle avec le code suivant :

```
resize([30,10]) circle(d=20);
```



Redimensionnement d'un cercle pour faire une ellipse

8.1.5. Texte

On peut également considérer le texte comme étant une primitive 2D, il est obtenu en utilisant l'instruction :

```
text("Votre texte ici");
```

Cette fonction nécessite de charger d'abord une *font* vectorisée et possède beaucoup de paramètres. Reportez-vous à la documentation officielle si vous êtes intéressés.

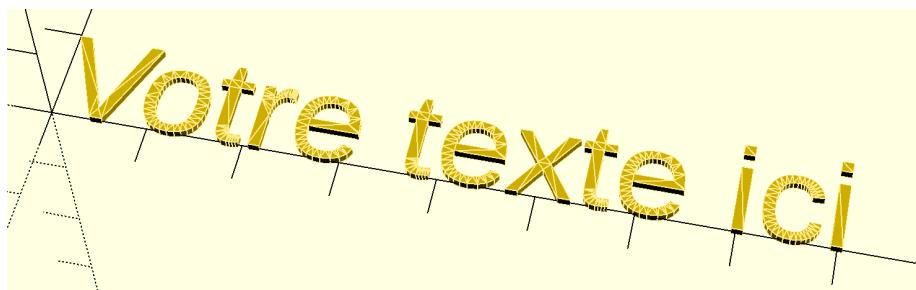
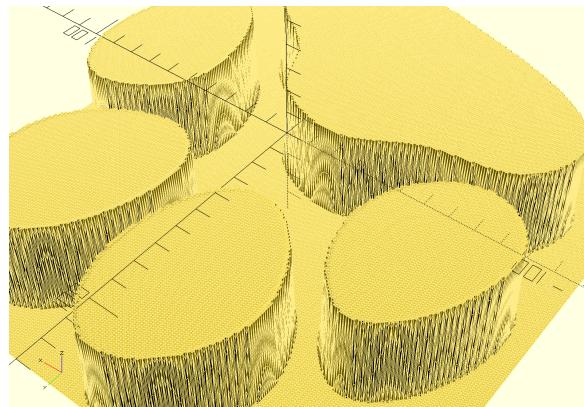


Illustration de la fonction text()

8.1.6. Surface

On peut également charger une image (avec l'extension .png) noir et blanc pour l'intégrer comme une surface avec l'instruction *surface()*. Par exemple :

```
surface(file = "/Path/image.png", center = true, invert = true);
```



Dog_Paw_Print.png³ importée comme une surface

³ https://upload.wikimedia.org/wikipedia/commons/c/c0/Dog_Paw_Print.png

8.2. Extrusion

Après avoir créé votre forme 2D il ne reste plus qu'à l'étirer en hauteur. *OpenScad* propose deux commandes pour créer de la 3D à partir de la 2D : *linear_extrude()* et *rotate_extrude()*.

8.2.1. Extrusion linéaire

L'extrusion linéaire est similaire à l'effet d'une défonceuse dans une presse avec un moule. Dans notre cas, le moule est représenté par la forme 2D, il ne reste qu'à préciser la « hauteur » d'extrusion (paramètre *height*):

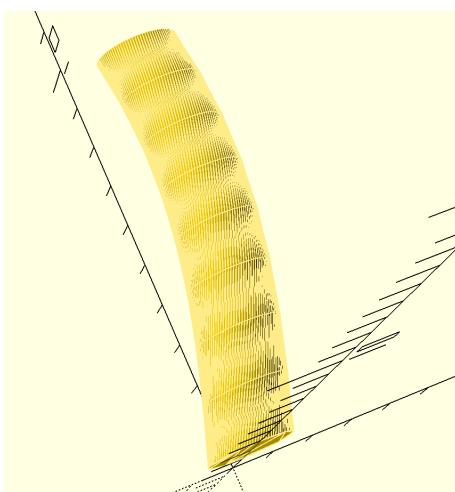
```
linear_extrude(height = 10) circle(r = 1, $fn=100);
```

Ce code permet de réaliser un cylindre, ce qui, dans ce cas précis, n'est pas très intéressant puisque nous aurions pu utiliser directement la primitive 3D *cylinder()*. Il y a d'autres paramètres, dont *twist*, qui est particulièrement intéressant.

Par exemple, le code :

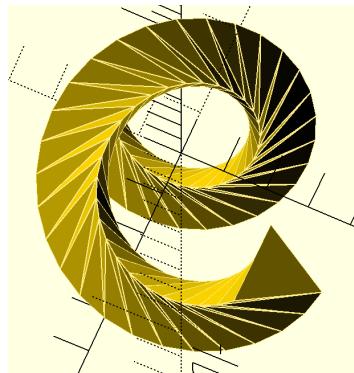
```
linear_extrude(height = 10, twist = -100)
    translate([2, 0, 0])
        circle(r = 1, $fn=100);
```

Permet de modéliser :



Extrusion linéaire d'un cercle

Ou encore un « e » en 3D :



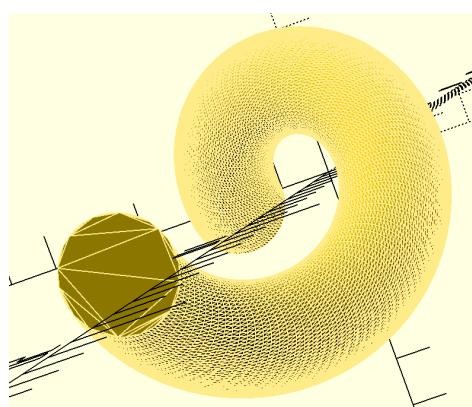
« e » en 3D avec `linear_extrude()`

Avec le code suivant :

```
linear_extrude(height = 10, center = true, twist = -500)
    translate([2, 0, 0])
        circle(r = 1, $fn=3);
```

Pour améliorer la « définition » on peut également utiliser le paramètre `$fn` dans l'extrusion :

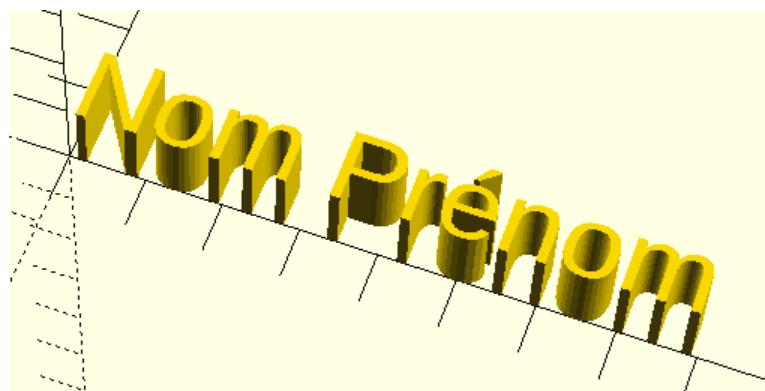
```
linear_extrude(height = 10, center = false, twist = 360,
                $fn = 100)
    translate([2, 0, 0])
        circle(r = 1);
```



Amélioration de la finition de l'extrusion avec le paramètre `$fn`

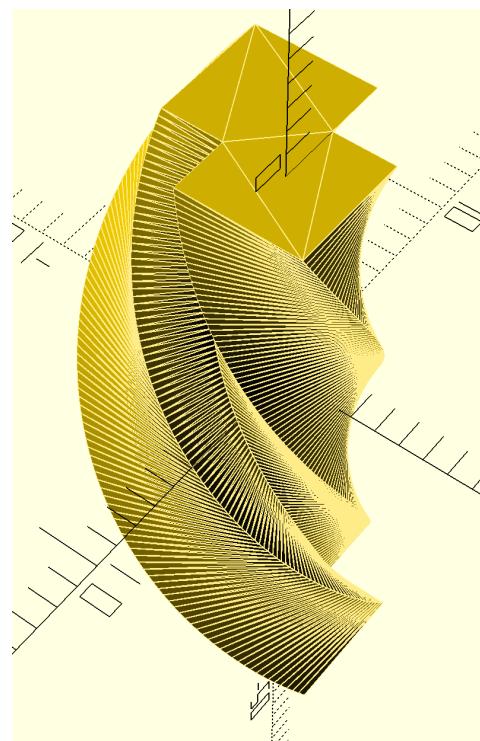
Exercices :

Réalisez la forme suivante avec une extrusion :



Vos nom et prénom extrudés

Réalisez un immeuble « comme à Dubaï »



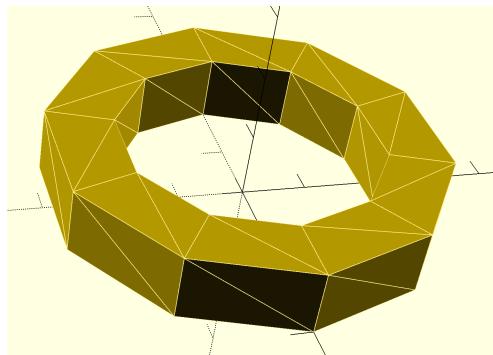
Un immeuble à Dubaï

8.2.2. Extrusion rotationnelle

L'extrusion rotationnelle produit comme son nom l'indique une extrusion en tournant autour de l'axe Z. Par exemple, le code :

```
rotate_extrude() translate([2, 0, 0]) square(1, $fn = 100);
```

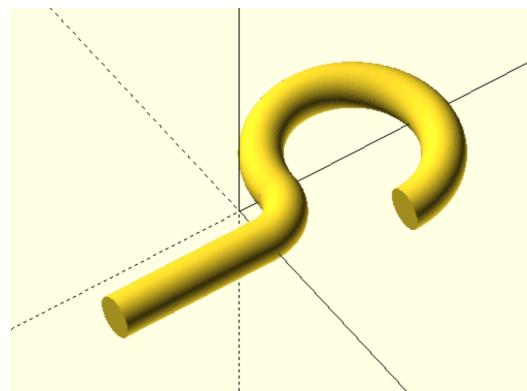
Produit le résultat suivant :



Forme obtenue avec un rotate_extrude()

Dans la dernière version (2017) d'*OpenScad*, on peut passer des paramètres à *rotate_extrude()* : *angle*, par exemple :

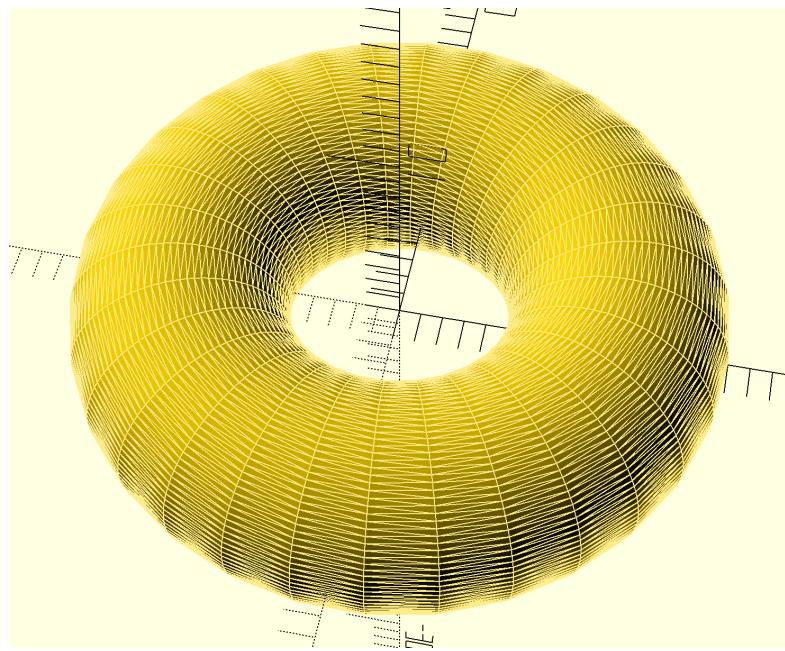
```
translate([0,60,0]) rotate_extrude(angle=270, convexity=10)
    translate([40, 0]) circle(10);
rotate_extrude(angle=90, convexity=10) translate([20, 0])
    circle(10);
translate([20,0,0]) rotate([90,0,0]) cylinder(r=10,h=80);
```



Le crochét

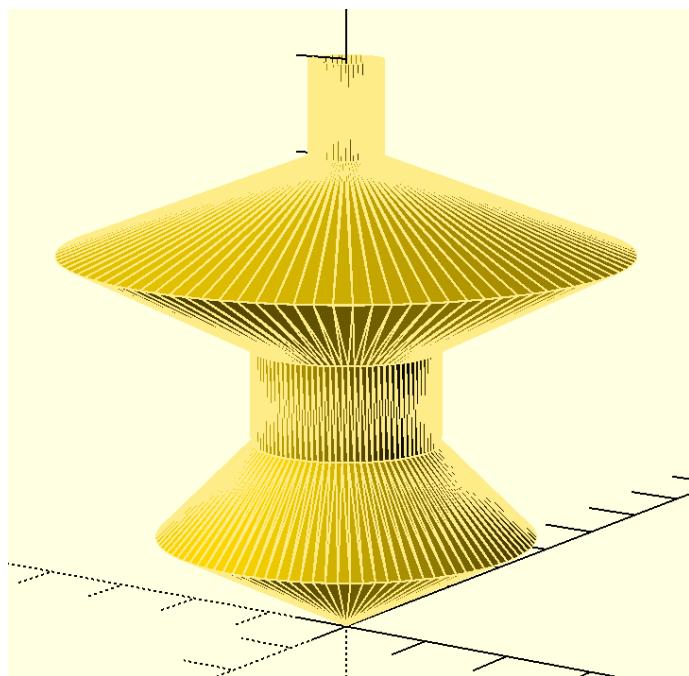
Exercices :

Réalisez la forme d'un Donut :



Le donut

Réalisez une petite toupie avec rotate_extrude() :



La toupie

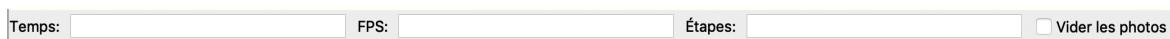
9. Animations

On peut, depuis la version 2015, réaliser dans *OpenScad* des animations simples. Pour les activer il faut aller dans le menu « *Vue* » et cliquer sur « *Animer* ».



Activer les animations

Une nouvelle barre de contrôle apparaît alors dans la fenêtre du logiciel.



La barre d'animation

Les animations sont contrôlées par la variable interne au système $\$t$. Cette variable va prendre des valeurs (comprises entre 0 et 1) qui sont calculées en fonction du paramètre *Etapes* que vous pouvez définir dans la barre de contrôle. Par exemple, si ce paramètre vaut 10, la variable $\$t$ prendra les valeurs 0, 0.1, 0.2, 0.3, 0.4, 0.5, 0.6, 0.7, 0.8 et 0.9, la valeur 1 n'est pas atteinte. À chacune de ces étapes une *frame* sera calculée (comme une pré-visualisation) et donnera donc l'impression d'une animation. La vitesse de l'animation sera donc contrôlée par le paramètre *FPS* (*frame per second*) dans la barre de contrôle. Par exemple, si vous mettez 10 pour le paramètre *Etape* et 1 pour le *FPS*, l'animation mettra 10 secondes pour se dérouler. Par défaut l'animation est bouclée, c'est à dire que lorsque la dernière valeur est atteinte, la variable $\$t$ reprend la valeur zéro et ensuite les valeurs suivantes. Avant de lancer l'animation, vous pouvez tester différentes valeurs pour $\$t$ en les tapant directement dans la barre de contrôle devant le paramètre *Temps*.

9.1. Translation animée

Pour faire bouger linéairement un objet, il suffit d'utiliser la variable `$t` dans la translation. Par exemple pour faire bouger une sphère suivant l'axe des `z` :

```
translate([0,0,$t*100]) sphere(10);
```

Exercice :

A partir de cet exemple, réaliser un petit train qui avance sur des rails.

9.2. Rotation animée

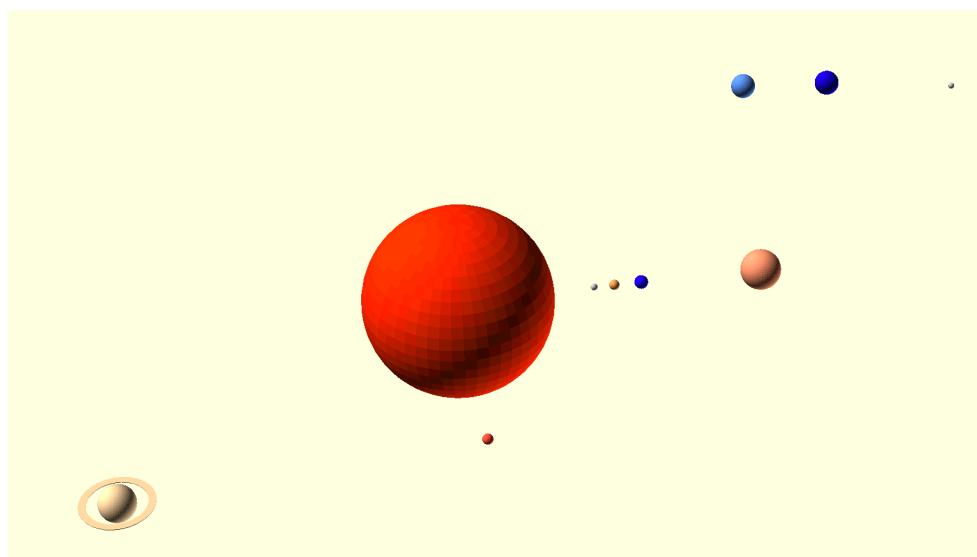
De la même manière, pour faire tourner un objet, il suffit d'utiliser la variable `$t` dans une rotation. Par exemple, pour faire tourner une sphère autour de l'axe des `z` :

```
rotate([0,0,$t*360]) translate([50,0,0]) sphere(10);
```

Vous remarquerez qu'ici on multiplie `$t` par 360 pour réaliser une rotation complète (360 degrés) autour de l'axe.

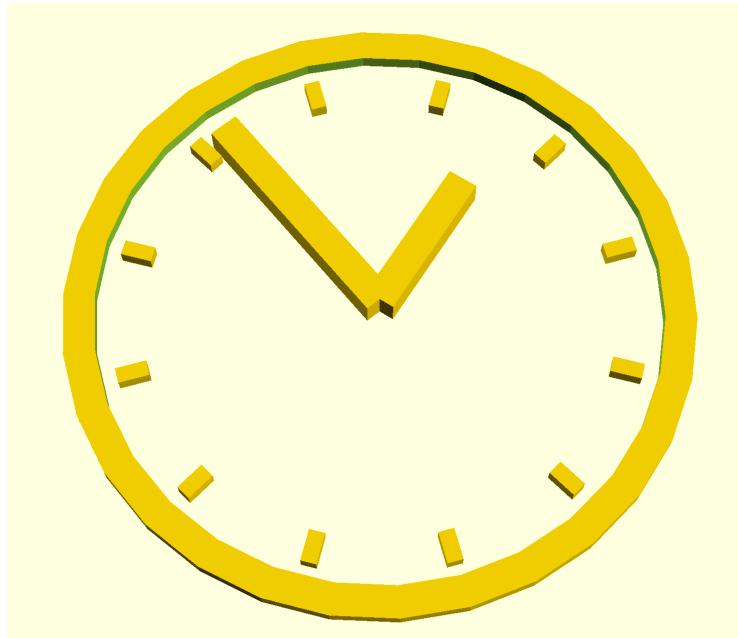
Exercices :

Réaliser une simulation des planètes du système solaire qui tournent autour du soleil.



Système solaire animé

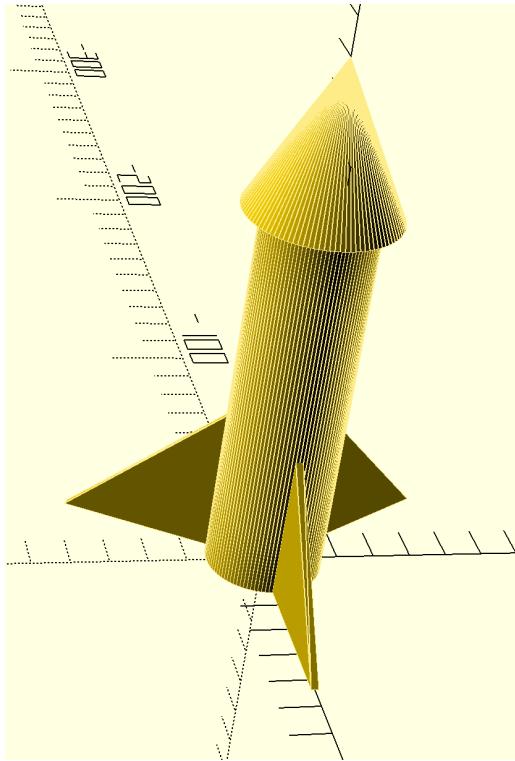
Réaliser une horloge analogique avec une petite et une grande aiguille qui tournent de manière logique (quand la grande aiguille fait un tour, la petite fait un douzième de tour).



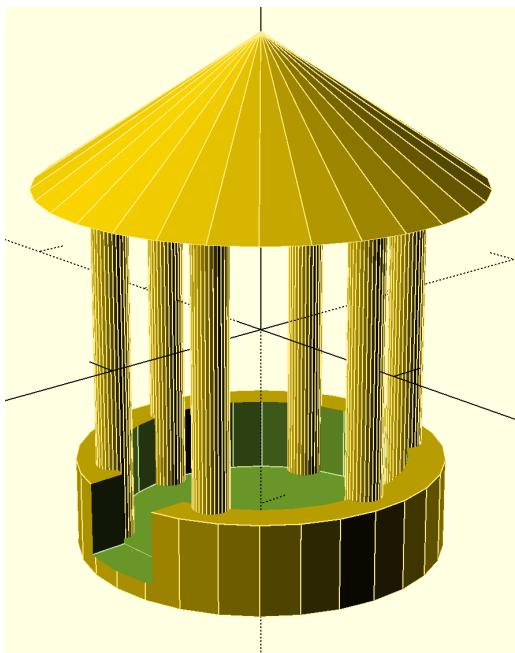
Horloge animée

10. Exercices récapitulatifs

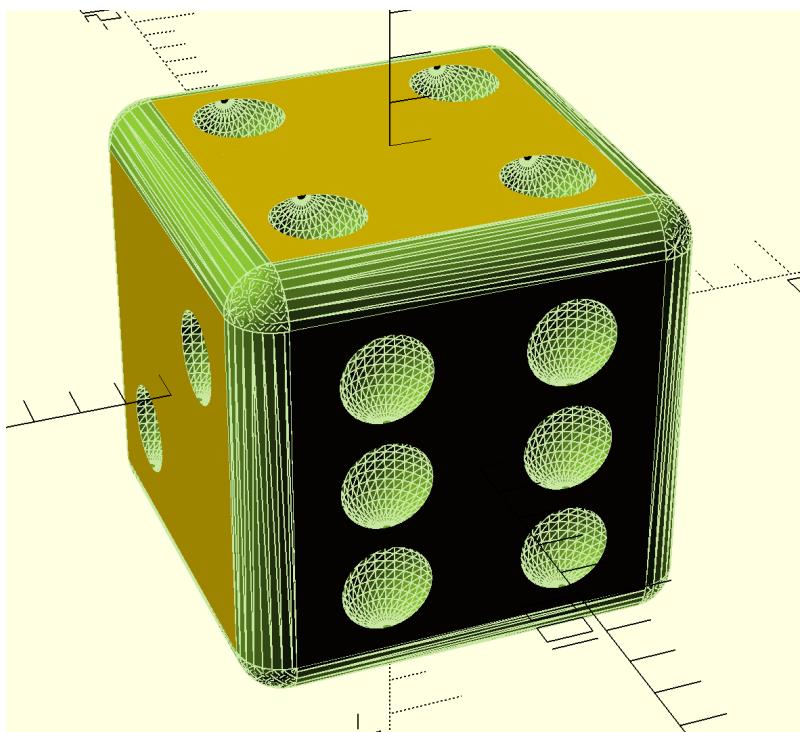
Voici quelques idées de réalisations sur lesquelles vous pouvez vous exercer pour progresser dans la réalisation de programmes OpenScad.



La fusée



La gloriette



Le dé

Annexes

Cheat Sheet	90
Solutions des exercices	91

Cheat Sheet

OpenSCAD CheatSheet v2015.03

Syntax

```
var = value;
module name(..) { ... }
name();
function name(..) = ...
name();
include <...scad>
use <...scad>
```

2D

```
circle(radius | d=diameter)
square(size,center)
square([width,height],center)
polygon([points])
polygon([points],[paths])
text(t, size, font,
      halign, valign, spacing,
      direction, language, script)
```

3D

```
sphere(radius | d=diameter)
cube(size, center)
cube([width,depth,height], center)
cylinder(h,r|d,center)
cylinder(h,r1|r2,d2,center)
polyhedron(points, triangles, convexity)
```

Functions

```
concat
lookup
str
chr
search
version
version_num
norm
cross
parent_module(idx)
```

Transformations

```
translate([x,y,z])
rotate([x,y,z])
scale([x,y,z])
resize([x,y,z],auto)
mirror([x,y,z])
multmatrix(m)
color("colorname")
color([r,g,b,a])
offset(r|delta,chamfer)
hull()
minkowski()
```

Boolean operations

```
union()
difference()
intersection()
```

Modifier Characters

*	disable
!	show only
#	highlight / debug
%	transparent / background

Mathematical

```
abs
sign
sin
cos
tan
acos
asin
atan
atan2
floor
round
ceil
ln
len
let
log
pow
sqrt
exp
rands
min
max
```

Other

```
echo(..)
for (i = [start:end]) { ... }
for (i = [start:step:end]) { ... }
for (i = [...,...]) { ... }
intersection_for(i = [start:end]) { ... }
intersection_for(i = [start:step:end]) { ... }
intersection_for(i = [...,...]) { ... }
if (...) { ... }
assign (...) { ... }
import("...stl")
linear_extrude(height,center,convexity,twist,slices)
rotate_extrude(angle,convexity)
surface(file = "...dat",center,convexity)
projection(cut)
render(convexity)
children([idx])
```

List Comprehensions

```
Generate [ for (i = range|list) i ]
Conditions [ for (i = ...) if (condition(i)) i ]
Assignments [ for (i = ...) let (assignments) a ]
```

Special variables

\$fa	minimum angle
\$fs	minimum size
\$fn	number of fragments
\$t	animation step
\$vpr	viewport rotation angles in degrees
\$vpt	viewport translation
\$vpd	viewport camera distance
\$children	number of module children

Solutions des exercices

Tabouret pour lion au cirque :

```
// Le tabouret pour lion féroce  
cylinder(h=5, r1=5, r2=4);
```

Un élément de Kapla :

```
/* Pour réaliser un élément de Kapla  
il suffit de faire : */  
cube(size=[18, 1, 4]);
```

Un escalier sur un coin :

```
cube(size=[18, 18, 2]); // la base  
cube(size=[12, 12, 4]); // le second élément  
cube(size=[6, 6, 6]); // l'élément du dessus
```

Le toit conique :

```
cylinder(h=20, r1=10, r2=0); // la base  
cylinder(h=30, r1=6, r2=0); // le deuxième élément  
cylinder(h=40, r1=3, r2=0); // le dessus pointu
```

Équerre :

```
polyhedron(points = [[0, -10, 60], [0, 10, 60], [0, 10, 0],  
[0, -10, 0], [60, -10, 60], [60, 10, 60], [10, -10, 50],  
[10, 10, 50], [10, 10, 30], [10, -10, 30], [30, -10, 50],  
[30, 10, 50]],  
faces = [[0,3,2],[0,2,1],[4,0,5],[5,0,1],[5,2,4],[4,2,3],  
[6,8,9],[6,7,8],[6,10,11],[6,11,7],[10,8,11],[10,9,8],  
[3,0,9],[9,0,6],[10,6, 0],[0,4,10],[3,9,10],[3,10,4],  
[1,7,11], [1,11,5], [1,8,7],[2,8,1],[8,2,11],[5,11,2]]);
```

Pyramide avec Polyhedron:

```
polyhedron(  
    points=[[10,10,0],[10,-10,0],[-10,-10,0],[-10,10,0],[0,0,10]],  
    faces=[[0,1,4],[1,2,4],[2,3,4],[3,0,4],[1,0,3],[2,1,3]]);
```

Parallélépipède à base octogonale :

```
cylinder(h=10, $fn=8);
```

Pyramide égyptienne :

```
cylinder(h=20,r1=30, r2=0, $fn=4);
```

Sphère à faible résolution :

```
sphere(r=10, $fn=6);
```

La boule de cristal :

```
cylinder(h=3,r1=2, r2=0);  
translate([0,0,3]) {  
    sphere(2, center=true, $fn=100);  
}
```

Une « maraca » :

```
cylinder(h=20,r1=4, r2=2, $fn=50);  
translate([0,0,25]) {  
    sphere(9, center=true, $fn=30);  
}
```

Château-fort avec 4 paramètres :

```
hauteurChateau = 5;
largeurChateau = 20;
longueurChateau = 22;
largeurTour = 2;
cube([largeurChateau, longueurChateau, hauteurChateau]);
cylinder(h=hauteurChateau+2, r=largeurTour, $fn=20);
translate([largeurChateau, longueurChateau, 0])
cylinder(h=hauteurChateau+2, r=largeurTour, $fn=20);
translate([0, longueurChateau, 0]) cylinder(h=hauteurChateau+2,
r=largeurTour, $fn=20);
translate([largeurChateau, 0, 0]) cylinder(h=hauteurChateau+2,
r=largeurTour, $fn=20);
```

Structure d'un tipi indien :

```
translate([0, 30, 0]) rotate([30, 0, 0]) cylinder(h=70, r=1);
translate([0, -30, 0]) rotate([-30, 0, 0]) cylinder(h=70, r=1);
translate([-30, 0, 0]) rotate([0, 30, 0]) cylinder(h=70, r=1);
translate([30, 0, 0]) rotate([0, -30, 0]) cylinder(h=70, r=1);
```

La maquette du moteur :

```
$fn=100;
h1 = 2.2;
h2 = 3;
h3 = 1.2;

rotate ([0,90,0]) {
    translate([0,0.7,-2.2]) cylinder(h=1.55, r=0.3);
    translate([0,0.7,-0.65]) cylinder(h=0.65, r=0.6);
    cylinder(h=h1, r=1.85);
    translate([0,0,h1]) cylinder(h=h2, r=1.725);
    translate([0,0,h1+h2]) cylinder(h=h3, r=0.1);
    translate([0,0,h1+h2+0.2]) cylinder(h=0.2, r=1.3);
    translate( [0,0,h1+h2+0.8] ) cylinder(h=0.4, r=0.9);}
```

La boite paramétrable :

```
largeur = 100;
longueur = 150;
hauteur = 15;
epaisseur = 1;

cube([largeur,longueur,epaisseur]);
translate([epaisseur,longueur-epaisseur,epaisseur]) {
    cube([largeur-2*epaisseur,epaisseur,hauteur-epaisseur]);}
translate([0, 0, epaisseur]) {
    cube([epaisseur,longueur,hauteur-epaisseur]);}
translate([largeur-epaisseur, 0, epaisseur]) {
    cube([epaisseur,longueur,hauteur-epaisseur]);}
translate([epaisseur,0,epaisseur]) {
    cube([largeur-2*epaisseur,epaisseur,hauteur-epaisseur]);}
```

Le ballon de rugby :

```
scale([1, 2.2, 1]) sphere(1, $fn=100);
```

Le stade de rugby :

```
scale([1, 2.2, 1]) {
    translate([0, 0, 1]) sphere(1, $fn=100);
    cylinder(1, $fn=100);
}
```

Le sous-marin jaune de *Laury Hughes* :

```
union(){

scale([1,0.5,0.7])sphere(30,$fn=100);
color([1,0,0])translate([0,0,20])cylinder(h=1,r=8,$fn=60);
difference(){
    color([1,0,0])translate([0,0,28])cylinder(h=1,r=7,$fn=60);
    translate([0,0,26])cylinder(h=5,r=5,$fn=60);}
```

```

translate([0,0,20])cylinder(h=8,r=7,$fn=60);
translate([0,15,0])scale([1,0.1,1])sphere(10,$fn=100);
color([0,0,0])translate([0,16,0])scale([0.75,0.1,0.75])
    sphere(10,$fn=100);
translate([0,-15,0])scale([1,0.1,1])sphere(10,$fn=100);
color([0,0,0])translate([0,-16,0])scale([0.75,0.1,0.75])
    sphere(10,$fn=100);
color([1,0,0])translate([-30,0,10])rotate([0,40,0])
    scale([1,0.4,0.7])sphere(10,$fn=100);
color([1,0,0])translate([-30,0,-10])rotate([0,-40,0])
    scale([1,0.4,0.7])sphere(10,$fn=100);
color([1,0,0])translate([0,0,20])cylinder(h=1,r=8,$fn=60);
color([1,0,0])translate([15,0,0])scale([0.2,0.8,1])
    sphere(19,$fn=100);
color([1,0,0]) translate([7,0,23]) sphere(1,$fn=100);
color([1,0,0]) translate([-7,0,23]) sphere(1,$fn=100);
color([1,0,0]) translate([0,7,23]) sphere(1,$fn=100);
color([1,0,0]) translate([0,-7,23]) sphere(1,$fn=100);
color([1,0,0]) translate([5,5,23]) sphere(1,$fn=100);
color([1,0,0]) translate([-5,-5,23]) sphere(1,$fn=100);
color([1,0,0]) translate([5,-5,23]) sphere(1,$fn=100);
color([1,0,0]) translate([-5,5,23]) sphere(1,$fn=100);
color([1,0,0]) translate([-13,13,0]) sphere(1,$fn=100);
color([1,0,0]) translate([-18,11.5,0]) sphere(1,$fn=100);
color([1,0,0]) translate([-23,9.5,0]) sphere(1,$fn=100);
translate([0,0,-20])cube(size=[10,10,20], center=true);
translate([0,0,-30])cylinder(h=1,r=20,$fn=100);
color([1,0,0]) translate([-13,-13,0]) sphere(1,$fn=100);
color([1,0,0]) translate([-18,-11.5,0]) sphere(1,$fn=100);
color([1,0,0]) translate([-23,-9.5,0]) sphere(1,$fn=100);
}

```

Le cube avec 4 boules aux sommets :

```
union() {
    cube(6, center=true);
    translate ([3, 3, 3]) sphere(1, $fn=100);
    translate([-3, -3, -3]) sphere(1, $fn=100);
    translate([-3, 3, 3]) sphere(1, $fn=100);
    translate ([3, -3, -3]) sphere(1, $fn=100);
}
```

Le « cendrier » :

```
translate ([0, 0, 1.5])
difference() {
    cube([4, 4, 3], center=true);
    translate ([0, 0, 3]) sphere(4, $fn=100);
}
```

Le siège de salon :

```
translate ([0, 0, 15]) {
    difference() {
        translate ([0, 0, -2]) sphere(10, $fn=100);
        translate ([0, 0, 3]) sphere(10, $fn=100);
        translate ([2, 0, 0]) sphere(10, $fn=100);
    }
}
translate ([0, 0, 1]) cylinder(h=2, r=1, $fn=100);
cylinder(h=1, r1=10, r2=1, $fn=100);
```

Le logo OpenScad :

```
difference() {
    sphere(5, $fn=100);
    cylinder(12, r=2.5, center=true, $fn=100);
    #rotate ([0, 90, 0]) cylinder(12, r=2.5, center=true,
                                    $fn=100);
    rotate ([0, 90, 90]) cylinder(12, r=2.5, center=true,
                                    $fn=100);
}
```

Le bol tibétain :

```
translate (v=[0,0,12]) {
    difference() {
        difference() {
            sphere(r=10, $fn=50);
            translate(v=[0,0,2]) {
                sphere(r=8, $fn=50);
            }
        }
        translate(v=[0,0,5]) {
            cube(size = [20,20,10], center = true);
        }
    }
    translate(v=[0,0,-10]) {
        cylinder (h=4, r1=5, r2=6, center = true);
    }
}
```

Nom à déterminer :

```
$fn = 20;  
hull () {  
    translate([0,5,5]) sphere(r=3);  
    translate([0,0,4]) rotate([180,0,0]) cube([2,2,5]);  
    translate([0,-5,5]) sphere(r=3);  
}
```

La montgolfière :

```
$fn = 20;  
minkowski () {  
    sphere(r=3);  
    rotate([180,0,0]) cylinder(r1=3, r2=0, h=5);  
}
```

Le quart de tarte :

```
intersection(){  
    cylinder(3, r=5, center=true, $fn=100);  
    cube(5); }
```

Le petit « osselet » :

```
intersection(){  
    cylinder(20, r=3, center=true, $fn=100);  
    rotate ([0, 90, 0]) cylinder(12, r=2.5, center=true, $fn=100);  
}
```

Le vaisseau :

```
intersection(){  
    sphere(r=4, center=true, $fn=100);  
    rotate ([0, 90, 0]) cylinder(h=12, r=2.5, center=true,  
        $fn=100);  
}
```

La pyramide aztèque :

```
module cubeTranslate(tailleX, tailleY, tailleZ, deplacementZ){  
    translate([0,0,deplacementZ]) cube([tailleX, tailleY,  
                                         tailleZ], center=true);  
  
}  
  
cubeTranslate(5, 5, 1, 0);  
cubeTranslate(4, 4, 1, 1);  
cubeTranslate(3, 3, 1, 2);  
cubeTranslate(2, 2, 1, 3);  
cubeTranslate(1, 1, 1, 4);
```

Le bonhomme de neige :

```
module sphereTranslate(rayonBoule, hauteurBoule){  
    translate([0, 0, hauteurBoule]) sphere(rayonBoule, $fn=100);  
  
}  
  
sphereTranslate(4, 4);  
sphereTranslate(2, 9.5);
```

Le tuyau (sous forme de module) :

```
module tube(hauteur, diametre) {  
    difference(){  
        cylinder(hauteur, d = diametre, $fn=100);  
        translate([0, 0, -1]) cylinder(hauteur+2,  
                                         d=diametre-1, $fn=100); } }
```

La forêt de tuyau :

```
include<Tube.scad>;  
  
tube (10, 2);  
translate([3,0,0]) tube (6, 2);  
translate([3,5,0]) tube (7, 2);  
translate([7,4,0]) tube (12, 2);
```

L' « escalier fou » :

```
for(z=[-180:45:+180])
    for(x=[1:2:10])
        rotate([0,0,z]) translate([x,0,0]) cube((x+1)*1.5);
```

La grille d'aération :

```
difference() {
    square([82,40]);
    for ( i = [1 : 26] ){
        translate([i*3,4,0]) {
            square([1,32]);
        }
    }
}
```

L'immeuble de Dubaï :

```
linear_extrude(height=20,twist=180,slices=100,center=true) {
    square(5);
    square(5,true);
}
```

Le donut :

```
rotate_extrude()
translate([10, 0, 0])
circle(r = 5, $fn = 100);
```

La toupie :

```
rotate_extrude($fn=100)
polygon( points=[[0,0],[2,1],[1,2],[1,3],[3,4],[0.4,5],[0.4,6]] );
```

Le satellite :

```
color("Goldenrod"){

    // corps, partie haute
    cylinder(h=50, r=15);

    // corps, partie basse
    translate([0,0,-40]){ cylinder(h=35, r=10); }

    // barre transversale
    translate([0,0,25]){ cube(size=[125,13,3], center=true); }

}

color("Silver"){

// rondelle au milieu
translate([0,0,-4]) { cylinder(h=3,r1=7, r2=6); }

// barre verticale
translate([0,0,-45]) { cylinder(h=100,r=3); }

// bout
translate([0,0,-49]) { cylinder(h=7,r1=4, r2=7); }

}

// panneaux solaires
color("LightBlue") {

    translate([25,0,0]) { cube(size=[35,5,50]); }
    translate([- (25+35),0,0]) { cube(size=[35,5,50]); }

}

// antenne
color("Silver"){

    translate([0,0,85]){

        difference(){
            sphere(r=30);
            translate([0,0,5]) { sphere(r=32); }

        }
    }

}

translate([0,0,90]) { sphere(r=3); }
translate([0,0,60]) { cylinder(h=30,r=2); }

}
```

Le vaisseau « star wars » (solution Florence Dawagne):

```
height=10;
body=10;
wingSpan=25;

scale([1,0.8,1]){
    translate([0,0,-height/2]){
        cylinder(h=height,r=body);
        translate([0,0,height]) {cylinder(h=height/2,r1=body,r2=body/2);}
        translate([0,0,-height/2]){
            cylinder(h=height/2,r1=body/2, r2=body);
        }
    }
    // Barre transversale
    rotate([0,90,0]){
        translate([0,0,-wingSpan*0.9])
        cylinder(h=wingSpan*2 * 0.9,r=body/3); }

    // ailes
    translate([0,0,-body]){
        difference(){
            cylinder(h=body*2, r=wingSpan);
            translate([-4,0,-2]) { cylinder(h=body*2+5, r=wingSpan+1); }
        }
    }

    rotate([0,0,180]){
        difference(){
            cylinder(h=body*2, r=wingSpan);
            translate([-4,0,-2]){
                cylinder(h=body*2+5, r=wingSpan+1);
            }
        }
    }
}
```

Le cube de Turner (solution de Wouter Gordts) :

```
module my_cube(a) {
    difference() {
        cube(size=a, center=true);
        sphere(r=((a/2) + (a/7)));
    }
}

for (i=[0:20:100]) {
    my_cube(i);
}
```

BB8 (Solution de Wouter Gordts) :

```
body_size = 130;
head_size = body_size / 1.9;

$fn=500;

// body

sphere(r=body_size);

// head

module head () {
    translate([0, 0, body_size]){
        difference () {
            sphere(r=head_size);
            translate([0, 0, - head_size / 2]) {
                cube([head_size* 2, head_size *2, head_size],
center=true);
            }
        }
    }
    translate([0, 0, body_size + (head_size / 2)]) {
        rotate([80, 0, 0]) cylinder(r=body_size / 10,
h=head_size);
        rotate([100, 0, 30]) cylinder(r=body_size / 15,
h=head_size + (body_size / 100));
    }
}

head();
```

Le tricératops (solution Sophie Bonnet) :

```
$fn = 50;

translate([0,0, 70]){
scale([1,1.5,1]){
    sphere(49);
}

//tête
translate([0, 68, 40]){
    scale([1,1.05,1]){
        sphere(36);
    }
}

//eyes
translate([-24, 82, 60]){
    rotate([0,35,-25]){
        scale([0.4,1,1]){
            sphere(10);
        }
    }
}

translate([-27, 85, 60]){
    rotate([0,35,-25]){
        scale([0.4,1,1]){
            color("black"){
                sphere(6);
            }
        }
    }
}

translate([24, 82, 60]){
    rotate([0,-35,25]){
        scale([0.4,1,1]) { sphere(10); }
    }
}
```

```

translate([27, 85, 60]){
    rotate([0,-35,25]){
        scale([0.4,1,1]){
            color("black"){
                sphere(6);
            }
        }
    }
}

//cornes
translate([20, 66, 65]){
    rotate([-30, 20, 0]){
        cylinder(h=35, r1=8, r2=0);
    }
}

translate([-20, 66, 65]){
    rotate([-30, -20, 0]){
        cylinder(h=35, r1=8, r2=0);
    }
}

translate([0, 118, 35]){
    rotate([-35, 0, 0]){
        cylinder(h=25, r1=7, r2=0);
    }
}

translate([0, 120, 20]){
//colette
    rotate([68,0,0]){
        cylinder(h=35, r1=10, r2=30);
    }
}
}

```

```

difference() {
    difference() {
        difference() {
            difference() {
                difference() {
                    translate([0, 63, 35]) {
                        rotate([35,0,0]) {
                            scale([1,0.1,1]) {
                                sphere(85);
                            }
                        }
                    }
                    translate([80, -25, 90]) {
                        rotate([-62,0,0]) {
                            cylinder(h=80, r=50);
                        }
                    }
                }
                translate([115, 0, 10]) {
                    rotate([-62,0,0]) {
                        cylinder(h=80, r=50);
                    }
                }
            }
        }
    }
}
translate([-115, 0, 10]) {
    rotate([-62,0,0]) {
        cylinder(h=80, r=50);
    }
}
translate([-80, -25, 90]) {
    rotate([-62,0,0]) {
        cylinder(h=80, r=50);
    }
}

```

```

        }
    }

}

translate([0, -30, 120]){
    rotate([-62,0,0]){
        cylinder(h=80, r=50);
    }
}

translate([0, 125, -180]){
    rotate([-62,0,0]){
        cylinder(h=80, r=210);
    }
}

}

//queue
translate([0, -47.5, -5]){
    rotate([103,0,0]){
        cylinder(h=90, r1=37.5, r2=0);
    }
}

}

//pattes
translate([24.5, -25, 0]){
    cylinder(h=70, r=20);
}

translate([-24.5, -25, 0]){
    cylinder(h=70, r=20);
}

translate([24.5, 28, 0]){
    cylinder(h=70, r=20);
}

translate([-24.5, 28, 0]){
    cylinder(h=70, r=20);
}

```

Petit train de :

```
$fn=100;

color ("Green") translate ([0,0,15]) rotate([90,0,0]) cylinder
(h=25, r=8);

color ("Orange") translate ([0,-10,20]) cylinder (h=15, r1=5,
r2=7);

color ("DodgerBlue") translate ([0,-25,6]) scale ([12,27,1]) cube
(size=2, center=true);

translate ([0,-35,20]) scale ([10,15,15]) cube(2, center=true);

color ("Red") translate ([12,-8.5,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Red") translate ([-14,-8.5,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Red") translate ([-14,-45,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Red") translate ([12,-45,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Gray") translate ([14,-8.5,6]) sphere(r=1, $fn=100);
color ("Gray") translate ([-14,-8.5,6]) sphere(r=1, $fn=100);
color ("Gray") translate ([-14,-45,6]) sphere(r=1, $fn=100);
color ("Gray") translate ([14,-45,6]) sphere(r=1, $fn=100);

color ("DodgerBlue") translate ([-11,-30,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([10,-30,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") scale ([1,1.2,1]) translate ([0,1,35]) rotate
([0,0,270])intersection () {

    translate ([30,0,-8.5]) rotate ([0,90,0]) cylinder (h=30,
r=16, center=true);

    translate ([30,0,12]) cube ([30,22,25],true);}

translate ([-10,0,0]) rotate ([0,0,10]) union () {
color ("DarkGrey") translate ([0,-52,5]) cylinder (h=5, r=1);
color ("Silver") translate ([0,-57,5]) cylinder (h=5, r=1);
color ("DarkGrey") translate ([0,-52,7]) rotate ([90,0,0])
cylinder (h=5, r=1);

translate ([0,-87.5,20]) scale ([10,30,15]) cube(2, center=true);
```

```

color ("DodgerBlue") translate ([-11,-87,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([10,-87,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([-11,-70,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([10,-70,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([-11,-105,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([10,-105,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([0,-88,6]) scale ([12,32,1]) cube
(size=2, center=true);

color ("DodgerBlue") translate ([0,-24.5,35]) scale ([1,2.1,1])
rotate ([0,0,270]) intersection () {
    translate ([30,0,-8.5]) rotate ([0,90,0]) cylinder (h=30,
r=16, center=true);
    translate ([30,0,12]) cube ([30,22,25],true);
}

color ("Red") translate ([-14,-110,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Red") translate ([12,-110,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Red") translate ([-14,-70,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Red") translate ([12,-70,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Gray") translate ([14,-70,6]) sphere(r=1, $fn=100);
color ("Gray") translate ([-14,-70,6]) sphere(r=1, $fn=100);
color ("Gray") translate ([14,-110,6]) sphere(r=1, $fn=100);
color ("Gray") translate ([-14,-110,6]) sphere(r=1, $fn=100);};

translate ([-8,-69,0]) rotate ([0,0,20]) union () {
color ("DarkGrey") translate ([0,-52,5]) cylinder (h=5, r=1);
color ("Silver") translate ([0,-57,5]) cylinder (h=5, r=1);
color ("DarkGrey") translate ([0,-52,7]) rotate ([90,0,0])
cylinder (h=5, r=1);

translate ([0,-87.5,20]) scale ([10,30,15]) cube(2, center=true);

```

```

color ("DodgerBlue") translate ([-11,-87,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([10,-87,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([-11,-70,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([10,-70,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([-11,-105,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([10,-105,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([0,-88,6]) scale ([12,32,1]) cube
(size=2, center=true);

color ("DodgerBlue") translate ([0,-24.5,35]) scale ([1,2.1,1])
rotate ([0,0,270]) intersection () {
    translate ([30,0,-8.5]) rotate ([0,90,0]) cylinder (h=30,
r=16, center=true);
    translate ([30,0,12]) cube ([30,22,25],true);
}

color ("Red") translate ([-14,-110,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Red") translate ([12,-110,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Red") translate ([-14,-70,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Red") translate ([12,-70,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Gray") translate ([14,-70,6]) sphere(r=1, $fn=100);
color ("Gray") translate ([-14,-70,6]) sphere(r=1, $fn=100);
color ("Gray") translate ([14,-110,6]) sphere(r=1, $fn=100);
color ("Gray") translate ([-14,-110,6]) sphere(r=1, $fn=100);
};

translate ([10,-134,0]) rotate ([0,0,25]) union () {
color ("DarkGrey") translate ([0,-52,5]) cylinder (h=5, r=1);
color ("Silver") translate ([0,-57,5]) cylinder (h=5, r=1);

```

```

color ("DarkGrey") translate ([0,-52,7]) rotate ([90,0,0])
cylinder (h=5, r=1);

translate ([0,-87.5,20]) scale ([10,30,15]) cube(2, center=true);
color ("DodgerBlue") translate ([-11,-87,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([10,-87,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([-11,-70,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([10,-70,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([-11,-105,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([10,-105,25]) rotate ([0,90,0])
cylinder (h=1, r=3.5, $fn=100);

color ("DodgerBlue") translate ([0,-88,6]) scale ([12,32,1]) cube
(size=2, center=true);

color ("DodgerBlue") translate ([0,-24.5,35]) scale ([1,2.1,1])
rotate ([0,0,270]) intersection () {
    translate ([30,0,-8.5]) rotate ([0,90,0]) cylinder (h=30,
r=16, center=true);
}

translate ([30,0,12]) cube ([30,22,25],true);
}

color ("Red") translate ([-14,-110,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Red") translate ([12,-110,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Red") translate ([-14,-70,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Red") translate ([12,-70,6]) rotate ([0,90,0])
cylinder(h=2, r=6);

color ("Gray") translate ([14,-70,6]) sphere(r=1, $fn=100);
color ("Gray") translate ([-14,-70,6]) sphere(r=1, $fn=100);
color ("Gray") translate ([14,-110,6]) sphere(r=1, $fn=100);
color ("Gray") translate ([-14,-110,6]) sphere(r=1, $fn=100); };

```

Système solaire animé (solution de Florence Dawagne) :

```
module makePlanet(size, place, rotation, coloring)
{
    rotate([0,0,rotation])
    translate([place,0,0])
    color(coloring)
    sphere(r=size, $fn=50);
}

module makeSaturn(place, rotation) {
    rotate([0,0,rotation])
    translate([place,0,0])
    color("NavajoWhite")
    difference() {
        cylinder(h=2,r=100);
        translate([0,0,-1])cylinder(h=5,r=80);
    }
    makePlanet(50,place,rotation,"NavajoWhite");
}

time = $t*36000;

//sun
color("OrangeRed")sphere(r=280,$fn=50);
//mercury
makePlanet(10,400,time/10,"Silver");
// venus
makePlanet(15,460,time/30,"SandyBrown");
// earth
makePlanet(20,540,time/60,"Blue");
// mars
makePlanet(15,620,time/80,"Tomato");
```

```
// asteroid belt
/*for(i = [0:360]){
    rotate([0,0,i])
    translate([730 + 30*sin(i),0,0])
    color("Silver")
    sphere(r=5, $fn=50);
}*/

// jupiter
makePlanet(60,900,time/60,"LightSalmon");
// saturn
makeSaturn(1200,time/100);
// unarus
makePlanet(40,1500,time/500,"CornflowerBlue");
// neptune
makePlanet(40,1700,time/600,"MediumBlue");
// pluto
makePlanet(10,2000,time/800, »Silver");
```

Horloge animée (Solution Florence Dawagne) :

```
difference() {
    cylinder(h=5,r=100);
    translate([0,0,-5])
        cylinder(h=15,r=90);
}

rotate([0, 0, $t*360*12])
    translate([0,-5,0])
        cube(size=[80,10,10]);

rotate([0, 0, $t*360])
    translate([0,-5,0])
        cube(size=[50,10,10]);

for(i=[0:30:360]){
    rotate([0,0,i])
        translate([0,80,0])
            cube(size=[5,10,5], center=true);
}
```