Capitolo 3

Creazione di Oggetti 3D

3.1. Gli Oggetti 3D

Gli **oggetti** di cui parliamo sono oggetti 3D (tridimensionali), cioè le cose che poi diventeranno reali una volta stampati.

In **FreeCAD** abbiamo una serie abbastanza completa di oggetti e possiamo addirittura definirne altri molto più complessi usando varie tecniche.

Usando lo scripting, abbiamo anche la scelta di usare molte vie, ho scelto di usare principalmente un modo, quello di usare la funzione:

```
DOC.addObject()
```

alla prova dei fatti questo metodo è risultato meno problematico di altri, e meno involuto, questa funzione crea direttamente oggetti che vanno nel documento aperto, senza poi doverli esplicitamente visualizzare richiamando una funzione come si fa con altri metodi.

Quando gli oggetti creati con questo metodo sono usati come componenti di una costruzione complessa, semplicemente diventano parte dell'albero dell'oggetto.

Faremo anche largo uso del linguaggio Python creando metodi e poi chiamandoli al bisogno, alla prova dei fatti si è dimostrato il modo più maneggevole per la costruzione di elementi anche molto complessi.

Il concetto base per costruire un oggetto complesso, assomiglia al vecchio indovinello "come fa una formica a mangiare un elefante?", la risposta ovvia, una volta che la si è sentita è "a piccoli pezzi".

Partiremo illustrando un paio di concetti, ma procederemo con esempi "concreti", non essendo un **Manuale** dove è necessario essere sistematici, ma piuttosto una **Guida** cercheremo appunto di guidare il lettore partendo dal "semplice" per arrivare al "complesso".

Faremo spesso riferimento a "listati" dei programmi che sono presenti nell'Appendice A, se li avessimo riportati nel testo, avrebbero occupato a volte più di una pagina e avrebbero interrotto il filo del discorso, forniremo all'interno del testo le porzioni di codice rilevanti e le aggiunte graduali.

Questa guida andrebbe seguita scrivendo (o scaricando il codice quando verrà reso dispo-

nibile) inizialmente lo **schema base**, Listato A.1 a pagina 45, salvandolo ed aggiungendo via via le porzioni di codice illustrate nel testo, risulterà utile salvare le versioni intermedie con il nome che preferite, in modo da poterle riprendere poi se necessario.

Non possiamo fare a meno ora di richiamare alcuni concetti relativi alla geometria 3D.

3.1.1. Spazio tridimensionale

Nella figura 3.1 qui accanto potete vedere una rappresentazione convenzionale dello spazio tridimensionale, ogni punto nello spazio 3D è definito da una tripla di coordinate (X, Y, Z).

Le convenzioni sono le stesse usate da **FreeCAD**, cioè come le vedete nella vista 3D.

La figura mostra una schematizzazione dello spazio 3D, con raffigurato un cubo, e gli otto punti che lo definiscono.

Ogni punto è definito da una tripla di numeri che riportano le coordinate, alcuni punti li potete ricavare seguendo le linee punteggiate sul grafico, ad esempio il punto **A** ha coordinate (5,3,0), mentre il punto **B** ha coordinate (8,3,0), non sorprendentemente il punto corrispondente ad **A** sulla faccia superiore del cubo **E** avrà coordinate (5,3,3) mentre **F** che corrisponde a **B** (8,3,3).

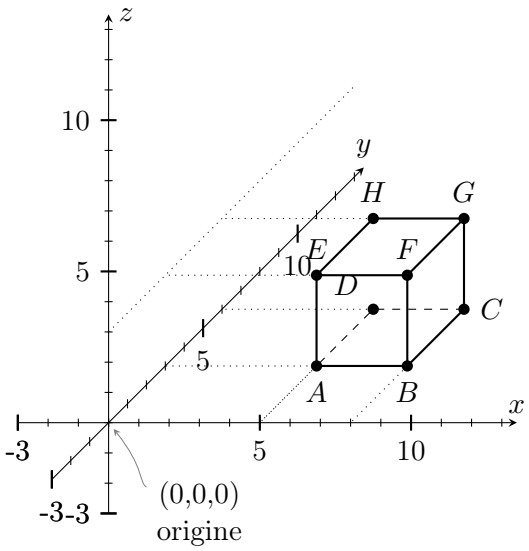
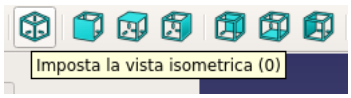


Figura 3.1.: Lo spazio 3D

Bozza

3.1.2. Proiezioni e viste

Per poter rappresentare uno spazio tridimensionale usando due dimensioni è necessario usare le cosiddette proiezioni, **FreeCAD** ne possiede diversi tipi, alcune accessibili direttamente attraverso il menù **Visualizza**, ad esempio **Vista Ortografica**, oppure **Vista Prospettica**, altre accessibili come voci del menù **Visualizza** ⇒ **Viste Standard**.



Le viste accessibili tramite il menù **Visualizza** ⇒ **Viste Standard** sono le viste "dall'alto", "dal basso", "da destra", "da sinistra", "di fronte" e "da dietro", si trovano anche nella Barra degli strumenti Viste visualizzate come un cubo con la faccia evidenziata relativamente alla posizione della vista richiesta. possiede anche altre tre viste nel **Visualizza** ⇒ **Viste Standard** ⇒ **Axonometric**.

Tra queste trovate anche una vista importante, la vista **Isometrica** che possiede anche una icona nella Barra degli strumenti Viste come potete vedere nella figura qui a fianco.

3.1.3. I Vettori

Un **vettore** in **FreeCAD** lo possiamo concepire come un contenitore che contiene 3 elementi, in genere le coordinate 3D di un punto nello spazio definite come valori in virgola mobile (float) di X, valore di Y e valore di Z in questo modo `Vector(val_x, val_y, val_z)`, è un concetto abbastanza comune a tutti i modellatori 3D.

A volte contiene anche cose diverse da un punto, ad esempio gli angoli di rotazione per ogni asse.

Perché un **vettore** e non qualche altra cosa, perché sui **vettori** si può operare in caso di necessità con opportune operazioni e perché ci permette di fare riferimento a quella terna di valori con un unico nome.

3.1.4. Topologia

La (Topologia), cioè della branca della geometria che studia le proprietà delle figure.

Usando **FreeCAD** non possiamo esimerci da introdurre alcuni concetti:

- **Vertice** (**Vertex**) Un elemento topologico corrispondente ad un punto. Esso non ha dimensioni.
- **Bordo** (**Edge**) Un elemento topologico corrispondente ad una curva limitata. Un **bordo** è generalmente delimitato dai **vertici**. Ha una dimensione.
- **Polilinea** (**Wire**) Una serie di **bordi** (una polilinea) collegati tra di loro nei **vertici**. Può essere aperto o chiuso, secondo se i **bordi** sono interamente concatenati oppure no.
- **Faccia** (**Face**) In 2D è una parte di un piano; in 3D è una parte di una superficie. La sua geometria è vincolata (delimitata/tagliata) dai suoi **bordi**.
- **Guscio** (**Shell**) Una serie di **facce** connesse nei loro **bordi**. Una **shell** (**guscio**) può essere aperta o chiusa. solid Una parte di spazio limitato da shell. E' tridimensionale

Bozza

3.1.5. Geometria e Vista

FreeCAD è stato inizialmente creato per lavorare come applicazione a riga di comando, senza la sua attuale interfaccia utente. Di conseguenza, quasi tutto al suo interno è separato in una componente "geometria" e una componente "vista".

In **FreeCAD**, quasi tutti gli oggetti possiedono due parti distinte, una parte **Object** che contiene i dati della geometria e una parte **ViewObject** che contiene i dati di visualizzazione, ad esempio il colore e la trasparenza.

Nello **schema base** a pagina 45 potete vedere un utilizzo della parte **ViewObject** nel metodo **setview()**.

3.1.6. Il riferimento di costruzione

Una delle particolarità di **FreeCAD** che all'inizio può destare qualche perplessità è il riferimento in base al quale viene costruito l'oggetto.

Nella tabella 3.1, elenchiamo i punti di riferimento dei vari oggetti.

Geometria	Punto di riferimento
Part::Box	vertice sinistro (minimo x), frontale (minimo y), in basso (minimo z)
Part::Sphere	centro della sfera (centro del suo contenitore cubico)
Part::Cylinder	centro della faccia di base
Part::Cone	centro della faccia di base (o superiore se il raggio della faccia di base vale 0)
Part::Torus	centro del toro
Part::Wedge	spigolo di Xmin Zmin

Tabella 3.1.: punti di riferimento

3.1.7. Il primo oggetto

Semplicemente partiamo definendo un oggetto, ad esempio un cubo, cioè un oggetto di tipo **Part::Box**, creiamo quindi un metodo in questo modo:

```
def cubo(nome, lung, larg, alt):
    obj_b = DOC.addObject("Part::Box", nome)
    obj_b.Length = lung
    obj_b.Width = larg
    obj_b.Height = alt

    DOC.recompute()

    return obj_b
```

Il metodo restituisce un oggetto cubo (anche se in realtà è un parallelepipedo), il perché deve restituire qualcosa diventerà chiaro più avanti.

Il metodo prevede 4 parametri:

1. **nome** il nome assegnato all'oggetto, questo nome apparirà nella **vista combinata** nella parte **vista ad albero**.
2. **lung** la misura della lunghezza
3. **larg** la misura della larghezza
4. **alt** la misura dell'altezza

estendiamo il nostro schema base aggiungendo la parte sopra, e un paio di altre righe, ottenendo il programma **cubo di prova** a pagina 46 che sorprendentemente e magicamente caricato e lanciato in **FreeCAD**, visualizzerà un cubo (questo sarà proprio un cubo) nella **vista 3D**.

Ora complichiamo la faccenda, il cubo è fatto, ma dov'è, nel senso in che **posizione** si trova?

Questo è uno dei segreti di **FreeCAD**, e uno dei suoi aspetti più complicati, della teoria e delle particolarità parleremo più avanti a tempo debito.

Per avere una piccola idea di dove si trova l'oggetto, possiamo attivare la visualizzazione dell'origine nella finestra grafica, usando il menu **Visualizza** ⇒ **Origine degli assi**, oppure usando aggiungendo al metodo **setview()**

```
FreeCAD.Gui.activeDocument().activeView().setAxisCross(True)
```

in questo modo si visualizza l'origine degli assi e le direzioni positive dei tre assi X, Y e Z.

ora potete giocare con i valori della riga:

```
obj = cubo("cubo_di_prova", 5, 5, 5)
```

e vedere a cosa corrispondono i valori lung, larg e alt, relativamente agli assi X, Y e Z.

Bozze

3.1.8. Posizionamento

Dalla tabella 3.1 a pagina 20 potete vedere che il cubo non usa il centro come riferimento di costruzione.

In una costruzione modulare, in genere, per evitare problemi, è necessario adottare una tecnica abbastanza rigorosa.

Una delle tecniche "salvavita" è quella di riportare tutti gli oggetti creati in una posizione convenzionale.

La posizione convenzionale più comoda è quella con la faccia inferiore centrata rispetto `Vector(0, 0, 0)`

Per fare questo è necessario modificare la posizione del cubo creato.

vettore è usato molto da **FreeCAD** in operazioni come la rotazione per contenere anche gli angoli di rotazione per i tre assi X, Y, Z oppure per definire usando i valori (0 ed 1) l'asse a cui si riferisce una certa operazione.

Vedremo fra poco questi due utilizzi.

La scelta di lavorare attraverso esempi impone che al momento di introdurre nuovi "costrutti" dobbiamo affrontare anche qualche argomento di teoria, per cui interrompiamo

momentaneamente la spiegazione per introdurre le direttive *import* di Python

La proprietà Placement

Per modificare la posizione abbiamo come al solito diversi metodi, secondo la mia opinione il metodo che porta minori problemi è quello di impostare direttamente la proprietà **Placement**.

La proprietà **Placement** può essere espressa in diversi modi, una delle "scritture" è:

```
FreeCAD.Placement(
    Vector(pos_X, pos_Y, pos_Z),
    FreeCAD.Rotation(Vector(0,0,0), 0)
    )
```

La trovate abbastanza spesso in giro, per cui spieghiamola con un certo dettaglio.

Riflette anche le proprietà che trovate nella **vista combinata** nella parte **editore delle proprietà** come potete vedere nell'immagine 3.2 a pagina 22 che visualizza la vista **Dati**.

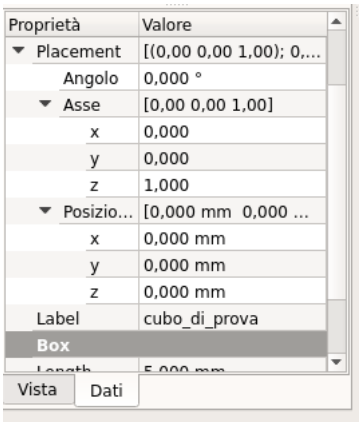


Figura 3.2.: L'editor delle proprietà

L'ordine delle voci varia rispetto a quello del codice, vediamo comunque che **Placement** possiede una freccia sulla sinistra che se premuta espande la vista come nella figura 3.2 e mostra 3 sotto proprietà:

Nome	Descrizione
Angolo	angolo di rotazione in gradi
Asse	asse di riferimento **Vettore** con valori 0 o 1 che selezionano l'asse di riferimento, sono ammessi più valori
Posizione	Posizione X Y e Z

Tabella 3.2.: proprietà Placement

Sorprendentemente questa "scrittura" non è quella che otteniamo aggiungendo questa linea al codice dello **schema base** a pagina 45.

```
print(obj.Placement)
```

subito prima della chiamata alla funzione **setview()**.

Se lanciamo il programma vedremo apparire (se non sono presenti errori) nella **finestra dei rapporti**.

```
Placement [Pos=(0,0,0), Yaw-Pitch-Roll=(0,0,0)]
```

Possiamo facilmente riconoscere un **Vettore** chiamato `Pos`, seguito da un secondo **Vettore** chiamato `Yaw-Pitch-Roll`.

Questo tipo di posizionamento usa gli **angoli di Eulero o di Tait-Bryan**, i nomi delle rotazioni (imbardata, rollio e beccheggio), sicuramente ricorderanno a qualcuno i termini tipici della navigazione navale o aeronautica:

Nome	Descrizione	Angolo
Yaw (imbardata)	rotazione rispetto a Z	**Psi ψ**
Pitch (beccheggio)	rotazione rispetto a Y (alzare o abbassare il muso)	**Phi φ**
Roll (rollio)	rotazione rispetto a X (dondolare le ali)	**Theta θ**

Tabella 3.3.: Angoli di Eulero

Bozza

Il modo più comodo per manipolare la rotazione di un oggetto è quello di usare (ed immaginare) la rotazione attorno al centro geometrico dell'oggetto, a meno di non avere delle ragioni per usare altri punti di riferimento.

Il problema è che raramente in **FreeCAD** un oggetto viene creato con il centro geometrico a X = 0, Y = 0 e Z = 0, o se vogliamo scriverlo come **Vettore** `Vector(0, 0, 0)`, si renderà pertanto necessario modificare il suo posizionamento, modificando la proprietà **Placement**, che però non viene conservata ad esempio quando si sposta un oggetto in un'altra posizione.

Il **posizionamento** e la rotazione sono cose relativamente complicate, forse le più ostiche da comprendere nella modellazione 3D, possono essere gestite usando qualche "astuzia" che vedremo a tempo debito.

Capitolo 4

Modellazione avanzata

Un oggetto può essere creato in molti modi, **FreeCAD** offre molte funzioni per creare oggetti anche molto complessi, il suo arsenale di strumenti è potente e versatile.

In questo capitolo analizzeremo alcune operazioni sugli oggetti, che vanno padroneggiate in quanto sono le tecniche fondamentali per la creazione di oggetti complessi.

Abbiamo già visto una delle geometrie di base, il parallepipedo, altre le tratteremo nel proseguimento del discorso, per non complicare la discussione con inutili elencazioni di nozioni.

Ma se le geometrie che abbiamo a disposizione non riescono a produrre, la forma voluta, abbiamo a disposizione un set di strumenti per crearne di nuove.

4.1. Le operazioni booleane

Bozza

Creare oggetti singoli può essere un utile esercizio, ma lo scopo finale è quello di creare oggetti complessi.

Partendo da semplici geometrie e operando sulle stesse utilizzando le **operazioni booleane**, che nonostante il nome strano sono relativamente semplici.

Queste funzioni sono le basi per la modellazione, chi ha studiato gli insiemi, troverà delle analogie, e non è un caso.

Vediamo le **operazioni booleane**:

Nome	Descrizione
Unione	**Part::Fuse** oppure **Part::MultiFuse**
Sottrazione	**Part::Cut**
Intersezione	**Part::MultiCommon**

Cominciamo ad estendere lo **schema base** a pagina 45 aggiungendo le righe presenti nel Programma 1 che trovate a pagina 25.

Invochiamo il metodo creato, in questo modo per creare un oggetto **Part::Cylinder**, in questo modo:

```
obj_1 = base_cyl('primo cilindro', 360,2,25)
```

Illustriamo i "parametri" che forniamo al metodo:

1. `nome` Il nome che desideriamo abbia la geometria
2. `ang` cioè l'angolo che viene assegnato alla proprietà **Angle**.
   Possiamo creare solo una porzione di cilindro fornendo valori inferiori a 360°
3. `rad` il raggio che viene assegnato alla proprietà **Radius**.
4. `alt` l'altezza che viene assegnata alla proprietà **Height**

Ora dovremmo avere un programma che assomiglia al listato A.3 a pagina 47

che una volta lanciato dovrebbe mostrare nella **vista combinata**, qualcosa che assomiglia alla figura 4.1a.

Non è proprio una scultura del Canova, ma già possiamo notare alcune cose:

1. Il cubo è costruito con lo spigolo in `Vector(0, 0, 0)`
2. il cilindro è posizionato con il centro della faccia inferiore in `Vector(0, 0, 0)`

Niente di strano in quanto pienamente in accordo a quanto abbiamo elencato nella tabella 3.1 a pagina 20

4.1.1. Unione

Bozza

Adesso si comincia a fare sul serio.

Uniamo le due figure usando **Part::Fuse**, lo facciamo però in modo Pythonico cioè usando una bel metodo, che potete vedere nel Programma 2.

Questa porzione di codice la inseriamo alla riga 58 del nostro schema base, subito dopo al metodo `base_cyl`.

Lo usiamo come è dovrebbe essere diventato usuale invocando il metodo con gli appropriati parametri.

```
def base_cyl(nome, ang, rad, alt ):
    obj = DOC.addObject("Part::Cylinder", nome)
    obj.Angle = ang
    obj.Radius = rad
    obj.Height = alt

    DOC.recompute()
```

Programma 1: metodo cilindro

```
fuse_obj("cubo-cyl-fu", obj, obj1)
```

che posizioniamo "ovviamente" dopo aver creato gli oggetti, quindi subito dopo la riga 71 che crea l'oggetto `primo cilindro` invocando il metodo `base_cyl`.

Lanciando il programma apparentemente non otteniamo nulla, infatti il risultato è la figura 4.1b se non che nella **vista combinata** sono "spariti" i due oggetti e ne è apparso uno solo chiamato `cubo-cyl-fu`.

Gli oggetti però non sono "spariti", sono diventati parte dell'oggetto `cubo-cyl-fu`, infatti cliccando sulla freccia ▶ essa diventa ▼ e appaiono "magicamente" i nomi dei due oggetti, in grigio chiaro, per indicare che sono i "componenti" dell'oggetto `cubo-cyl-fu`.

Soffermiamoci un attimo sul funzionamento della **vista combinata**, solo due parole, se selezionate un oggetto e premete la barra spaziatrice, essa si comporta come un interruttore per la **Visibilità** dell'oggetto.

Se ad esempio rendiamo invisibile l'oggetto `cubo-cyl-fu` e poi rendiamo visibile l'oggetto `primo cilindro`, possiamo controllare il suo posizionamento, ovviamente con solo due oggetti è quasi inutile, ma con molti diventa essenziale.

4.1.2. Sottrazione

La sottrazione, detta anche "Taglio" traducendo letteralmente la parola **Cut**, sottrae una geometria da un'altra.

Al solito creiamo un metodo come nel Programma 3.

Bozza

Questa porzione di codice la inseriamo alla riga 66 del nostro schema base, subito dopo al metodo `base_cyl`.

Lo usiamo come è dovrebbe essere diventato usuale invocando il metodo con gli appropriati parametri.

```
cut_obj("cubo-cyl-cu", obj, obj1)
```

```
def fuse_obj(nome, obj_0, obj_1):
    obj = DOC.addObject("Part::Fuse", nome)
    obj.Base = obj_0
    obj.Tool = obj_1
    obj.Refine = True
    DOC.recompute()

    return obj
```

Programma 2: Unione

Mettiamo il segno di # davanti all'invocazione di `fuse_obj`, copiamo la riga sopra e lanciamo il programma, ottenendo come risultato è la figura 4.1c.

Niente di eccezionale, notate però due cose, la struttura del metodo è sostanzialmente la stessa del metodo `fuse_obj`, ovviamente con il nome del metodo cambiato, e l'operazione **Cut** al posto di **Fuse** nella definzione dell'oggetto.

Il concetto importante è che la proprietà **Base** è la geometria da cui dobbiamo sottrarre l'oggetto che sarà contenuto nella proprietà **Tool**, e l'uso della proprietà **Refine** settata a `True` che cerca di "affinare" l'oggetto eliminando le facce inutili e le cuciture.

4.1.3. Intersezione

L'Intersezione, estrae la parte comune delle geometrie.

Creiamo questo metodo come nel Programma 4: Intersezione.

Questa porzione di codice la inseriamo alla riga 76 del nostro schema base, subito dopo al metodo `cut_obj`.

Lo usiamo come è dovrebbe essere diventato usuale invocando il metodo con gli appropriati parametri.

```
int_obj("cubo-cyl-is", obj, obj1)
```

Mettiamo il segno di # davanti all'invocazione di `cut_obj`, copiamo la riga sopra e lanciamo il programma, ottenendo come risultato è la figura 4.1d: .

Trovate il listato dell'esempio completo con il nome **Operazioni booleane - esempio completo** nell'Appendice A.4 a pagina 49.

Bozza

4.2. Estrusione

Finora abbiamo creato direttamente gli oggetti usando le forma base, per forme complesse esiste una tecniche potente chiamata "estrusione", che usa la funzione **extrude**.

```
def cut_obj(nome, obj_0, obj_1):
    obj = DOC.addObject("Part::Cut", nome)
    obj.Base = obj_0
    obj.Tool = obj_1
    obj.Refine = True
    DOC.recompute()

    return obj
```

Programma 3: Sottrazione

Questa tecnica consiste nel creare una forma in 2D e poi estruderla in 3D, più o meno come fa una stampante 3D.

Per utilizzare questa tecnica dobbiamo fare uso di alcuni concetti.

Non ci muoveremo dal modulo Part.

Creiamo questo metodo come nel Programma A.5: Estrusione - esempio completo a pagina 52.

Il programma contiene abbastanza carne al fuoco, analizziamolo in dettaglio:

```
def reg_poly(center=Vector(0, 0, 0), sides=6, dia=6,
             align=0, outer=1):
```

Con queste righe definiamo il metodo che creerà il nostro oggetto, la definizione e di conseguenza la chiamata possiede di molti parametri, per cui viene dotata di una nutrita docstring, costruita secondo i "canoni" di Python, contiene la descrizione della funzione, e `Keywords Arguments:`, seguito dalla linea 7 alla linea 12 dalla lista completa dei parametri con una spiegazione sommaria di quello che fanno.

Per chi non legge bene l'inglese, cerchiamo di elencarli:

- `center` = un Vettore che contiene il centro del poligono
- `sides` = il numero dei lati del poligono da un minimo di 3 a quello che volete voi
- `dia` = il diametro del **cerchio base** con una specificazione che può trattarsi o dell'apotema (diametro interno) o del diametro esterno
- `align` = allineamento (0 oppure 1), allineare il poligono all'asse (X)
- `outer` = 0 = apotema, 1 diametro esterno, cioè specifica che di che tipo è la misura fornita con il parametro `dia`

Bozza

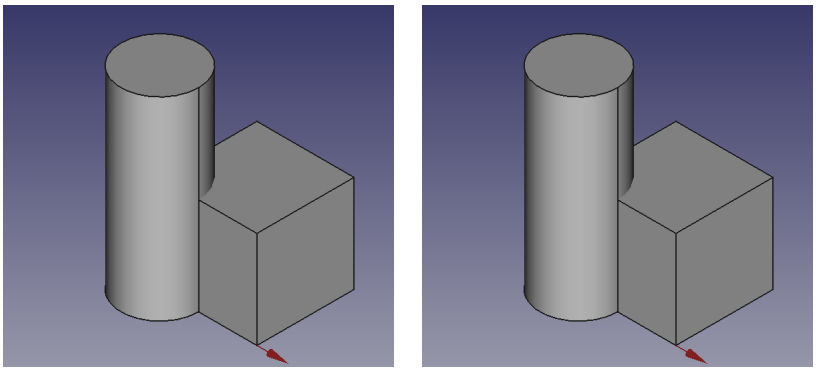
Il parametro `outer` merita una spiegazione in merito alla parola apotema, L'apotema è il **raggio della circonferenza interna** del poligono, nel programma è usato in modo improprio. [1]

[1] Non possiamo scrivere un trattato in una docstring per cui dobbiamo semplificare, la scelta della parola "apotema" risiede nella sua compattezza per indicare che il dato è relativo al **diametro del**

```
def int_obj(nome, obj_0, obj_1):
    obj = DOC.addObject("Part::MultiCommon", nome)
    obj.Shapes = [obj_0, obj_1]
    obj.Refine = True
    DOC.recompute()

    return obj
```

Programma 4: Intersezione



(a) cubo e cilindro. (b) Unione

(c) Sottrazione. (d) Intersezione.

Figura 4.1.: Le operazioni Booleane.

Bozza

Segue poi la "magia nera" del calcolo dei punti di vertice del poligono e della loro assegnazione alla **lista** `vertex`.

La linea:

```
    vertex = []
```

si occupa di creare la **lista** `vertex` creandola vuota, la parentesi quadra aperta e chiusa creano la lista vuota.

```
        vertex.append(Vector(vpx, vpy, 0))
```

si occupa di aggiungere `append()` è il metodo per aggiungere alla lista `vertex` qualcosa.

Il qualcosa è il Vettore che contiene il punto di vertice appena calcolato dal ciclo `for`, ma potete notare che all'interno delle equazioni per calcolare i punti di vertice ci siamo riferiti alla variabile `center` usando delle parentesi quadre, questo costrutto, si chiama **indicizzazione**, in pratica dice prendi l'elemento numero x dalla lista `center`, ma se ricordate bene `center` è un Vettore non una lista.

cerchio interno rispetto al **diametro del cerchio esterno**, usando la definizione corretta, si cade facilmente in confusione, perché il nostro cervello confonde molto facilmente due concetti che differiscono in pratica per solo due lettere (in/es)terno.

In Python molte cose possono essere rappresentate come **sequenze** e quindi indicizzate: `center[0]` è l'elemento 0 della sequenza Vettore, quindi il valore di X, ovviamente 1 è il secondo elemento, quindi il valore di Y, `center[2]` sarebbe Z, ma qui non ci è servito.

Veniamo ora alla parte sostanziosa:

```
obj = Part.makePolygon(vertex)
```

Assegniamo alla variabile `obj` il risultato della chiamata al metodo `Part.makePolygon()`, a cui abbiamo passato l'elenco dei vertici, questo elenco deve essere ordinato in senso orario o antiorario, ma comunque ordinato, stiamo passando l'elenco dei vertici di un polygono e le linee non devono "incrociarsi".

```
wire = Part.Wire(obj)
```

Trasformiamo l'oggetto poligono in un oggetto di tipo **Polilinea** (**Wire**)

```
poly_f = Part.Face(wire)
```

Trasformiamo l'oggetto **Polilinea** (**Wire**) in una **Faccia** (**Face**), ricordiamo dalla definizione che la **Faccia** è una parte di piano delimitato dal una **Polilinea** chiusa.

Questa funzione è stata creata per una funzione eminentemente pratica, creare esagoni, lo si capisce dai valori di default, e dalla scelta di quali parametri passare al metodo.

Gli esagoni sono molto usati in quanto sono la base per la costruzione di dadi, o di alloggiamenti per gli stessi, cosa molto comune nella stampa 3D.

Internamente, il metodo utilizza correttamente il raggio del poligono, i vertici infatti sono situati sul cerchio esterno del poligono, noi al metodo forniamo invece diametro interno e centro.

Internamente sono usate le formule per trasformare l'apotema in raggio del cerchio esterno, e i diametri vengono trasformati in raggi moltiplicando per 0.5. [2]

Un dado si distingue in base al diametro e al passo della filettatura interna, ad esempio "dado 3MA", ma la misura che più mi interessa è il diametro della chiave per manovrarlo che per inciso nei dadi 3MA è da 5,5 mm, da qui la scelta del parametro come diametro del cerchio interno.

Ottenuta con questi metodi la parte di piano possiamo finalmente **estruderla**, per farlo scriviamo un metodo:

[2] Nel programma ci sono molte "divisioni per due" scritte come "moltiplicazioni per 0.5" la ragione è un "trucco salvavita" da programmatore.

Alcuni linguaggi interpretano la divisione in modo strano (fortunatamente non è il caso di Python) per cui se io divido un valore **float**, cioè un **numero con la virgola** per 2 che è un **intero** l'interprete o il compilatore convertono l'operazione in una divisione di **interi**, restituendo un **intero** invece che un **float**, facendo diventare poi matti a trovare l'errore nel codice, usando la moltiplicazione si moltiplica un **float** per un altro valore **float** evitando ogni possibile errore di conversione implicita.

Una seconda ragione è che in genere la moltiplicazione è meno costosa in termini di risorse elaborative di una divisione.

Bozza

```
def dado(nome, dia, spess):
    polyg = reg_poly(Vector(0, 0, 0), 6, dia, 0, 0)

    nut = DOC.addObject("Part::Feature", nome + "_dado")
    nut.Shape = polyg.extrude(Vector(0, 0, spess))

    return nut
```

Analizziamolo in dettaglio:

```
polyg = reg_poly(Vector(0, 0, 0), 6, dia, 0, 0))
```

Questa linea di codice si occupa di recuperare la nostra parte di piano, chiamando la funzione VNMreg_poly

Ora creiamo il nostro oggetto, però non avendo una forma definita, dobbiamo usare la classe base, cioè un contenitore generico:

```
nut = DOC.addObject("Part::Feature", nome + "_dado")
```

L'oggetto **Part::Feature**, è la classe generica degli oggetti 3D, a cui dobbiamo fornire è una proprietà **Shape** attraverso:

```
nut.Shape = polyg.extrude(Vector(0, 0, spess))
```

Notate la "magia nera" della funzione **extrude** usata in `polyg.extrude()`, ad essa dobbiamo solo fornire un vettore che indica il punto finale dell'estrusione, nel nostro caso estrudiamo l'oggetto fino al punto con `Vector(0, 0, spess)`, ricordardando che l'esagono era costruito con il centro in (0, 0, 0), in pratica estrudiamo in direzione Z di `spess`.

Dovreste ottenere il risultato mostrato nella figure 4.2a a pagina 32.

Potete liberamente modificare il valore di questo vettore per vedere che risultati ottenete.

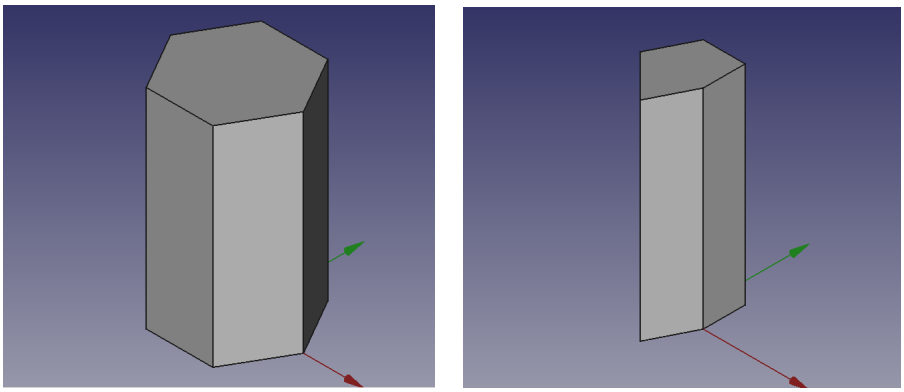
L'oggetto vero e proprio va come siamo abituati a fare creato invocando il metodo, in questo modo:

```
dado("Dado", 5.5, 10)
```

Una volta averlo creato noterete che nel'**editore delle proprietà** della **vista combinata**, il nostro oggetto avrà solo la proprietà **Base** e la proprietà **Placement**, più che sufficienti per poter operare sull'oggetto.

Usando la sola interfaccia grafica non potremmo modificare l'oggetto, pensate di dover definire tutti i punti a mano, trovando la posizione e inserendo ogni punto con click del mouse, poi estruderlo e infine assegnare la forma ad un oggetto. Se fate un errore dovete rifare tutta la trafila, a meno di non aver salvato le forme intermedie.

Usando lo scripting, basta editare un paio di parametri e rilanciare il programma, decisamente più comodo.



(a) Dado. (b) Prisma.

Figura 4.2.

Nel caso di esempio l'estrusione è stata fatta con un oggetto semplice, se componete profili complessi, ovviamente il risultato sarà più complesso, un esempio semplice semplice è dato dal metodo `estr_comp` presente nel listato citato sotto che permette di ottenere il prisma mostrato nella nella figure 4.2b a pagina 32.

La parte rilevante è la definizione del profilo base da cui viene estruso il prisma, basta creare una lista ordinata di punti, in questo modo:

```
def estr_comp(nome, spess):
    vertex = ((-2,0,0), (-1, 2, 0), (1, 2, 0), (1, 0, 0),
              (0, -2, 0), (-2,0,0))
```

In realtà abbiamo creato una **tupla** contenente tante **tuple** quante sono i punti del poligono.

Ci sono alcune regole da rispettare:

- Il punto iniziale e il punto finale devono coincidere, perché il poligono deve essere **chiuso**.
- punti devono essere ordinati tra di loro (per convenzione si usa l'ordinamento antiorario).

Per contenere i punti invece di usare una **lista** abbiamo usato una **tupla** che in pratica è una lista immutabile, nel senso che una volta creata non può essere modificata.

Il suo utilizzo deriva dal fatto che risparmia memoria, notate che la **tupla** è identificata come abbiamo già accennato nella sezione **Le condizioni** a pagina 2.2.4 dall'uso delle "parentesi tonde" nella sua definizione, la **lista** usa le "parentesi quadre".

Se però avete dei vettori vanno bene lo stesso, la funzione `Part.makePolygon()`, accetta:

- Una lista di Vettori

- Una lista di tuple
- Una tupla di tuple

Notate che non accetta una **lista di liste**, per cui comunque i punti vanno racchiusi tra parentesi tonde.

L'utilizzo della funzione ci permette di ottenere forme molto complesse in maniera molto semplice, ed efficace, sovrapponendo o disegnando una forma su di un foglio di carta millimetrata e ricavandone il contorno, come una serie di coordinate X e Y, potete velocemente riprodurre una forma in una decina di linee di codice.

La trasformazione della lista in una faccia avviene nelle linee:

```
obj = Part.makePolygon(vertex)
wire = Part.Wire(obj)
poly_f = Part.Face(wire)
```

con cui otteniamo una **Faccia** che poi estrudiamo nella stessa maniera usata nel metodo `dado`.

Trovate il listato dell'esempio completo con il nome **Estrusione - esempio completo** nell'Appendice A.5 a pagina 52.

4.3. Rivoluzione

Bozza

In questa sezione non parliamo parliamo del fenomeno sociale, in quanto stiamo trattando di tecniche di creazione di oggetti 3D.

Una funzione molto potente è la funzione **revolve** che fornisce la possibilità di usare una parte di piano per definire un oggetto mediante la rotazione, da un triangolo possiamo ottenere un cono, da un parallelepipedo un cilindro, da un cerchio un toro, ecc.

Ovviamente avendo già a disposizione queste figure nell'arsenale di **FreeCAD**, useremo figure diverse, ad esempio l'esagono creato prima.

```
face = reg_poly(Vector(0, 0, 0), 6, 5.5, 0, 0)
```

Fino a qui nulla di eccezionale, e nemmeno nella parte finale del codice:

```
obj = DOC.addObject("Part::Feature", nome)
obj.Shape = face.revolve(pos, vec, angle)
```

Creiamo un oggetto base e assegniamo come forma quella ottenuta facendo ruotare la porzione di piano scelta.

Il succo sono i parametri che passiamo alla funzione **revolve**, analizziamoli in dettaglio:

```
# base point of the rotation axis
pos = Vector(0,10,0)
```

Indica il ”punto base” della rotazione, nel nostro caso il la faccia ha centro in (0, 0, 0) e scegliamo un punto base di (0, 10, 0) cioè a 10mm dal centro del poligono.

```
# direction of the rotation axis
vec = Vector(1,0,0)
```

Questo vettore fornisce l’asse di riferimento in questo caso l’asse X

```
angle = 180 # Rotation angle
```

Questo è ovviamente l’angolo di rotazione 180°

Il risultato sarà una cosa come quella in figura 4.3.

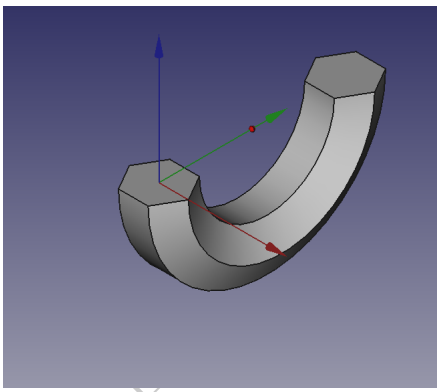


Figura 4.3.: Rivoluzione

Al solito nulla di eccezionale, solo un esempio, l’unico limite è la vostra fantasia.

Trovate il listato dell’esempio completo con il nome **Rivoluzione - esempio completo** nell’Appendice A.6 a pagina 52.