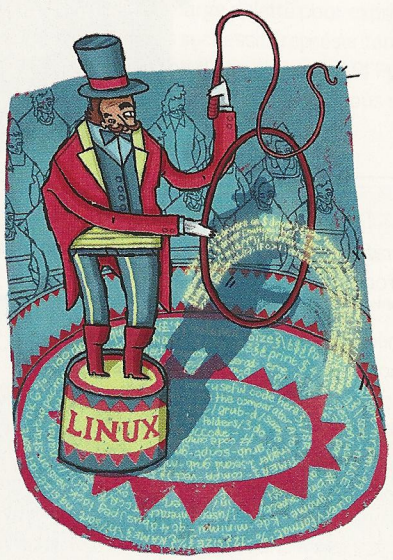
Minecraft : créer un trébuchet avec le Pi

Préparez l'offensive en catapultant des projectiles dans les airs avec du code pyrotechnique, une pincée de TNT et une machine de guerre d'un autre âge.



Maintenant que vous connaissez les bases de l'API, il est l'heure de faire preuve de créativité. Construire une maison est difficile, n'est-ce pas ? Faux. En quelques lignes de Python, la résidence de vos rêves vous tend les bras. Enfin, si elle repose essentiellement sur le kit de construction standard, il faut bien l'avouer. Si vous nourrissez de plus grands rêves, il vous suffira de développer davantage de code. Vous n'aurez jamais à vous soucier du permis de construire, de la connexion, des réparations ou des contributions et du creusage accidentel d'un cimetière néolithique (à moins de l'avoir construit vous-même au préalable).

Il ne pleut jamais dans Minecraft Pi, un toit plat convient parfaitement à nos besoins. Nous commençons par définir les deux bords de la maison : v1 est le bloc à nos côtés dans la direction x, à un bloc au-dessus de notre altitude actuelle, tandis que v2 se trouve à une distance que l'on estime suffisante :

```
pos = mc.player.getTilePos()
v1 = minecraft.Vec3(1,1,0) + pos
v2 = v1 + minecraft.Vec3(10,4,6)
Now we create a solid stone cuboid
```

Nous créons à présent un cuboïde entre ces deux verticales, en l'évidant à l'aide d'un cuboïde plus petit contenant de l'air frais :

```
mc.setBlocks(v1.x,v1.y,v1.z,v2.x,v2.y,v2.z,4)
mc.setBlocks(v1.x+1,v1.y,v1.z+1,v2.x-1,v2.y,v2.z-1,0)
```

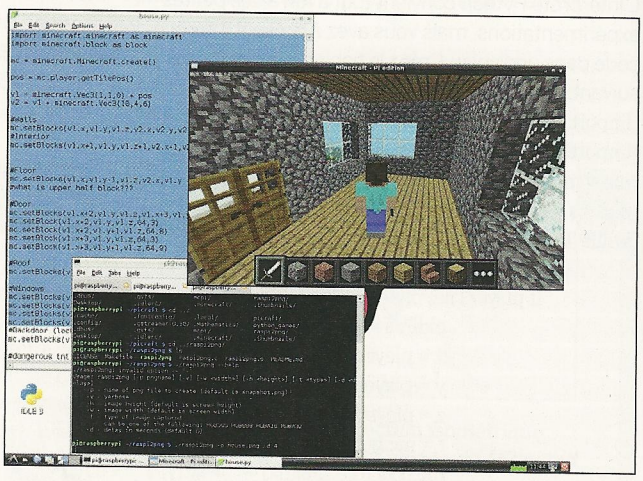
C'est parfait, sauf que notre seul point d'entrée reste cette lucarne plutôt généreuse et on apprécierait tout particulièrement un plancher de bois. Si vous vous tenez sur une zone plate, vous remarquerez que les murs de votre maison se tiennent à un bloc au-dessus du niveau du sol. C'est là où nous allons ajouter le plancher. Si la topographie n'est pas si plate, votre maison se tient peut-être sur une colline, ou partiellement dans les airs, mais ne vous inquiétez pas – les ultimes ajustements à la gravité locale s'effectueront par la suite. Nous créons notre plancher de bois rustique :

```
mc.setBlocks(v1.x,v1.y-1,v1.z,v2.x,v1.y -1,v2.z,5)
```

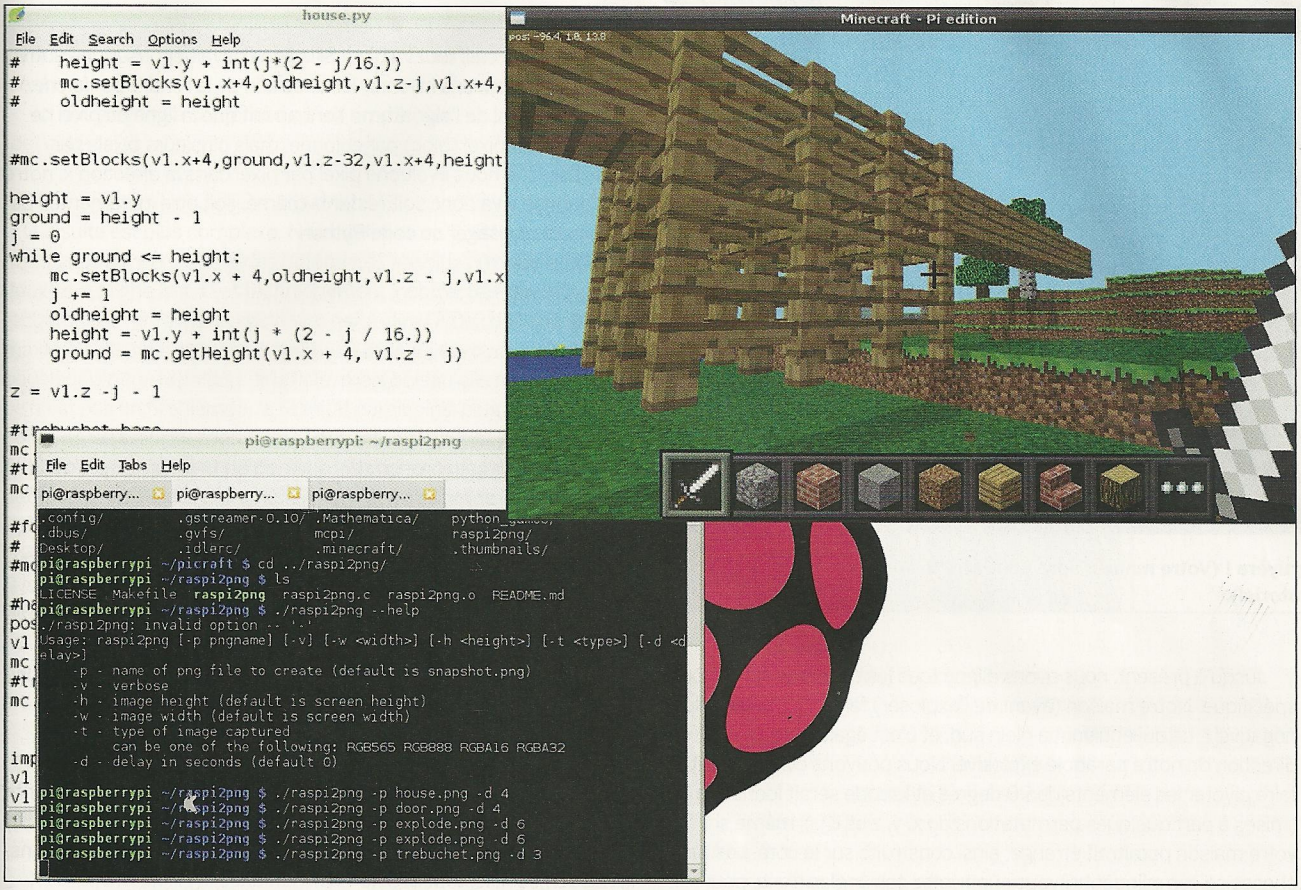
Les fenêtres ne sont qu'une variante du même thème :

```
mc.setBlocks(v1.x,v1.y+1,v1.z+1,v1.x,v1.y+2,v1.z+3,102)
mc.setBlocks(v1.x+6,v1.y+1,v1.z,v1.x+8,v1.y+2,v1.z,102)
mc.setBlocks(v2.x,v1.y+1,v1.z+1,v2.x,v1.y+2,v1.z+3,102)
mc.setBlocks(v1.x+2,v1.y+1,v2.z,v1.x+4,v1.y+2,v2.z,102)
```

Le toit utilise le demi-bloc spécial 44, qui présente des types différents. On définit le bloc en lui donnant le même bois que le plancher :



› **Une maison. Faisons-la exploser, maintenant.**



› **Cela n'a l'air de rien, mais attendez un peu...**

```
mc.setBlocks(v1.x,v2.y,v1.z,v2.x,v2.y,v2.z,44,2)
```

La porte est plus complexe, reportez-vous à l'encadré pour de plus amples détails, mais les trois lignes suivantes suffisent à l'ajouter :

```
mc.setBlocks(v1.x+2,v1.y,v1.z,v1.x+3,v1.y,v1.z,64,3)
mc.setBlock(v1.x+2,v1.y+1,v1.z,64,8)
mc.setBlock(v1.x+3,v1.y+1,v1.z,64,9)
```

Après avoir consciencieusement bâti notre nouvelle résidence, l'étape suivante consiste à trouver des moyens inventifs de la détruire. Nous avons déjà évoqué la possibilité d'utiliser du TNT, avec un simple coup d'épée (ou quoi que ce soit), vous le ferez détonner. Il serait futile d'utiliser setBlocks pour remplir votre maison de dynamite, nous pouvons faire bien mieux. C'est là que le trébuchet entre en jeu.

Au lieu de simuler le déplacement d'un projectile dans l'espace, nous allons tracer sa trajectoire parabolique en la faisant suivre de TNT. En faisant détonner son extrémité, nous produirons une réaction en chaîne, qui conduira à l'explosion de votre maison dans un déluge de flamme. Nous devons tout d'abord nous attarder sur les mécanismes basiques en deux dimensions. En l'absence de frottements, un projectile décrit une parabole en fonction de la vitesse du lancement initial, de l'angle du lancer et de l'accélération gravitationnelle, qui est sur Terre de 9;81ms-2.

Pour bien démarrer, nous allons injecter ces constantes afin que la distance horizontale couverte par l'arc corresponde à 32 blocs, avec un sommet 16 blocs plus haut que l'altitude d'origine. Si les blocs étaient des mètres, ce lancer aurait ainsi une vitesse de 18ms-1, avec un angle de 60 degrés. Nous travaillons en deux dimensions, l'arc sera tracé le long de l'axe z avec la coordonnée x fixe, face à notre porte. La simple formule y = z(2 – z/16) le résume, que nous intégrons de cette manière :

```
for j in range(33):
        height = v1.y + int(j*(2 – j/16.))
        mc.setBlock(v1.x+4,height,v1.z-j,46,1)
```

Le dernier argument active le TNT, touchez-le avec votre épée et profitez du feu d'artifice. Ou au contraire, gardez-vous-en : les explosions vont, outre peser sur les ressources du Pi, faire tomber du TNT, ce qui interrompt la réaction en chaîne et préserve la maison. Ce n'est pas ce que nous voulons, utilisez le code suivant à la place :

```
height = v1.y
ground = height - 1
j = 0
while ground <= height:
        mc.setBlocks(v1.x + 4,oldheight,v1.z - j,v1.x + 4,height,v1.z -
        j,46,1)
        j += 1
        oldheight = height
        height = v1.y + int(j * (2 - j / 16.))
        ground = mc.getHeight(v1.x + 4, v1.z - j)
```

Vous vous assurez ainsi que la parabole ne présente pas de trou et vous évitez que l'arc-en-ciel de TNT n'explose à moitié dans les airs. Nous utiliserons à nouveau cette fonction **getHeight()** pour déterminer le niveau du sol à chaque point de l'arc, afin d'arrêter sa construction lorsqu'on l'atteint.

Notez que l'on doit appeler la fonction **getHeight()** avant de placer le dernier bloc de TNT, la hauteur du monde étant définie par l'objet non-air le plus élevé, même s'il flotte dans l'espace.

Si notre construction dépasse l'environnement de Minecraft, vous pouvez construire une maison dans un meilleur emplacement, ou vous pouvez remplacer **v1.z - j** par **max(-116,v1.z-j)** dans la boucle précédente, ce qui construit un totem vertical au bord du monde. Maintenant que nous avons notre trajectoire, nous ajoutons l'impitoyable machine de guerre :

```
z = v1.z -j - 1
mc.setBlocks(v1.x + 3, oldheight, z +10, v1.x + 6, oldheight +
2, z+ 7,85)
mc.setBlocks(v1.x + 4, oldheight +2, z +12, v1.x + 4,
oldheight +2, z +1, 5)
```

Astuce express

Le code du trébuchet nous a été inspiré par Martin O'Hanlon et ses formidables projets autour du Raspberry Pi, que vous retrouverez sur **www.stuffaboutcode.com.**

Un vrai morceau de gruyère ! (Votre maison aura peut-être besoin de réparations, après ce tutoriel).

Jusqu'à présent, nous avons aligné tous les éléments sur un axe spécifique. Notre maison (avant de l'exploser) fait face à la direction négative z, ce qui équivaut à plein sud, et c'est également la direction de notre parabole explosive. Nous pouvons évidemment faire pivoter les éléments de 90 degrés et le code serait identique (mises à part quelques permutations de x, y, z et +/-), même si votre maison paraîtrait étrange, ainsi construite sur le côté. Les choses se compliquent si vous supprimez ces angles droits pour travailler autour d'une construction plus exotique. La difficulté, c'est de calculer une ligne droite alors que nous avons une série de blocs et non de points.

La fonction en trois dimensions **drawline()** vous sera d'une aide inestimable dans vos prochaines créations, elle vous permet de créer de nombreux éléments, notamment des parallélépipèdes et des pentagrammes. Vous devez utiliser une version 3D du classique algorithme de Bresenham. Le gourou du Pi Martin O'Hanlon propose plusieurs projets incroyables autour de Minecraft Pi Edition sur son github, dont un canon d'où vient l'inspiration de ce projet pour débutants. Martin propose une classe complète de dessin en Python, qui comprend le tracé de lignes 3D évoqué précédemment, mais une fois que vous aurez compris la version 2D, il vous sera facile de généraliser.

Imaginons que le monde de Minecraft est plat et que nous souhaitons calculer la ligne dans le plan (x,y) reliant les points (-2,-2) et (4,1). Cette ligne correspond à l'équation y = 0,5x – 1. L'algorithme nécessite que le gradient de la ligne soit compris entre 0 et 1, dans ce cas, tout va bien. Si la ligne devait épouser une autre pente, nous pouvons inverser les axes afin de la rendre conforme. Le point crucial de l'algorithme tient au fait que la ligne de pixel ne remplit qu'un pixel (bloc) par colonne, mais plusieurs pixels par ligne. Lorsque nous évoluons pixel par pixel dans la direction x, notre coordonnée y va donc soit rester la même, soit être incrémentée de 1. On peut envisager ce code Python :

```
dx = x1 – x0
dy = y1 – y0
y = y0
error = 0
grad = dy/dx
for x in (x0,x1):
        plot(x,y)
        error = error + grad
        if error >= 0.5:
                y += 1
                error -= 1
```

où **plot()** est une fonction de traçage et **grad** est compris entre 0 et 1. On incrémente ainsi y à chaque fois que le taux d'erreur grandit suffisamment, et le résultat correspond à l'image ci-contre.

L'astuce de Bresenham consistait à réduire tous les calculs à des opérations d'entiers, bien plus à la portée du matériel des années 1960. Nous pouvons à présent effectuer des calculs en virgule flottante à la vitesse de la lumière, mais il est toujours agréable de profiter de ces astuces. Les variables en virgule flottante grad et error surviennent en raison de la division par **dx**, si nous multiplions toutes les valeurs par cette quantité et que nous réglons la mise à l'échelle, tout est ok.

Pour l'adapter en trois dimensions, il suffit de poursuivre l'abstraction. Il faut tout d'abord déterminer l'axe dominant (celui qui présente les plus nombreux changements de coordonnées) et déplacer les éléments en conséquence, en évoluant d'un bloc à la fois et en incrémentant les coordonnées des axes mineuses si nécessaire. Nous devons prêter attention au signe de chaque changement de coordonnée, que nous stockons dans la variable **ds.** La fonction **ZSGN()** retourne 1, -1 ou 0 si l'argument est respectivement positif, négatif ou nul. Nous vous laissons cet exercice. Nous utilisons largement la fonction **minorList(a,j)**, qui renvoie une copie de la liste a, sans la j-ème entrée. On le programme en une ligne grâce aux fonctions lamba et au découpage de liste :

```
minorList = lambda a,j: a[:j] + a[j +1:]
```

Notre function **getLine()** prend deux vertices, que nous représentons avec des listes de trois éléments, et retourne une liste de toutes les vertices dans la ligne 3D résultante. Nous nous basons ici sur le code de Mrtin. La première partie initialise la liste des

Astuce express

Vous pouvez saisir tout le code dans l'interprète, mais les erreurs de saisie sont frustrantes. Il vous sera ainsi plus simple de placer tout le code dans un fichier baptisé maison.py, que vous exécuterez ensuite avec la commande python maison.py pendant que Minecraft est en cours d'exécution.

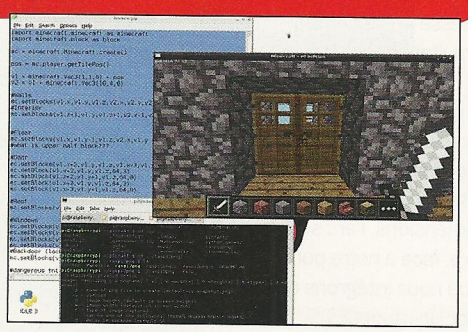
À propos d'une double porte

Placer des portes sur notre maison fut notre première rencontre avec le paramètre **blockData.** Il s'agit d'un entier entre 0 et 15, qui contrôle des propriétés complémentaires des blocs, comme la couleur du sol ou l'activation ou non du TNT. Notre porte occupe quatre blocs et se voit alignée avec l'axe x. Elle est légèrement en retrait des murs adjacents, elle est fermée et dispose de poignées en son centre. Ces propriétés sont contrôlées par les différents bits de **blockType.** Nous numérotons les quatre bits à partir du bit 0 le plus à droite vers le bit 3 le plus à gauche, si bien que dans cette notation 8 vaut 1000 en binaire. Le bit 3 est défini si le bloc appartient à la section supérieure d'une porte. Si c'est le cas, le bit 0 est le seul autre bit concerné – il définit la position des poignées. Les sections supérieures des portes ont donc un blocType 8 ou 9.

Pour les sections inférieures, nous assignons les bits suivants :

bit 3 ..... off
bit 2 ..... la porte est ouverte
bit 1 .... la porte est fermée
bit 0 .... alignement (off = x, on = z)

Les sections supérieures doivent être placées après les inférieures, car elles héritent leurs propriétés de ces dernières.



Les portes sont une bonne idée pour les grands claustrophobes.

vertices et traite le cas de figure trivial où elles sont toutes deux identiques...

Ici notre ligne ne comprend qu'un simple bloc :

```
def getLine(v1, v2):
        if v1 == v2:
                vertices.append([v1])
```

La suite est plus complexe. Nous définissons la liste précédente des signes **ds**, et une liste des différences absolues (multipliées par deux) **a**. La ligne **idx** = est techniquement incorrecte – nous recherchons notre axe dominant, c'est-à-dire l'index de l'entrée maximale de a. En utilisant la méthode **index()** avec **max,** nous bouclons à deux reprises sur la liste, mais puisqu'elle reste courte, ce n'est pas un problème – le résultat paraît ainsi plus clair. Nous pointons vers les coordonnées dominantes avec **X** et **X2**. Notre liste s est un réarrangement de **ds**, avec les coordonnées dominantes au début. On retrouve d'autres listes pour suivre la trace des erreurs. La variable **aX** correspond au signe du changement de coordonnées, le long de l'axe dominant.

```
else:
        ds = [ZSGN(v2[j] - v1[j]) for j in range(3)]
        a = [abs(v2[j]-v1[j]) << 1 for j in range(3)]
        idx = a.index(max(a))
        X = v1[idx]
        X2 = v2[idx]
        delta = a[idx] >> 1
        s = [ds[idx]] + minorList(ds,idx)
        minor = minorList(v1,idx)
        aminor = minorList(a,idx)
        dminor = [j - delta for j in aminor]
        aX = a[idx]
```

Une fois ces éléments en place, attaquons-nous à la boucle principale, dans laquelle on ajoute les vertices, on examine les différences le long des axes mineurs, on calcule les erreurs et on incrémente les coordonnées majeures. Nous retournons enfin une liste de vertices.

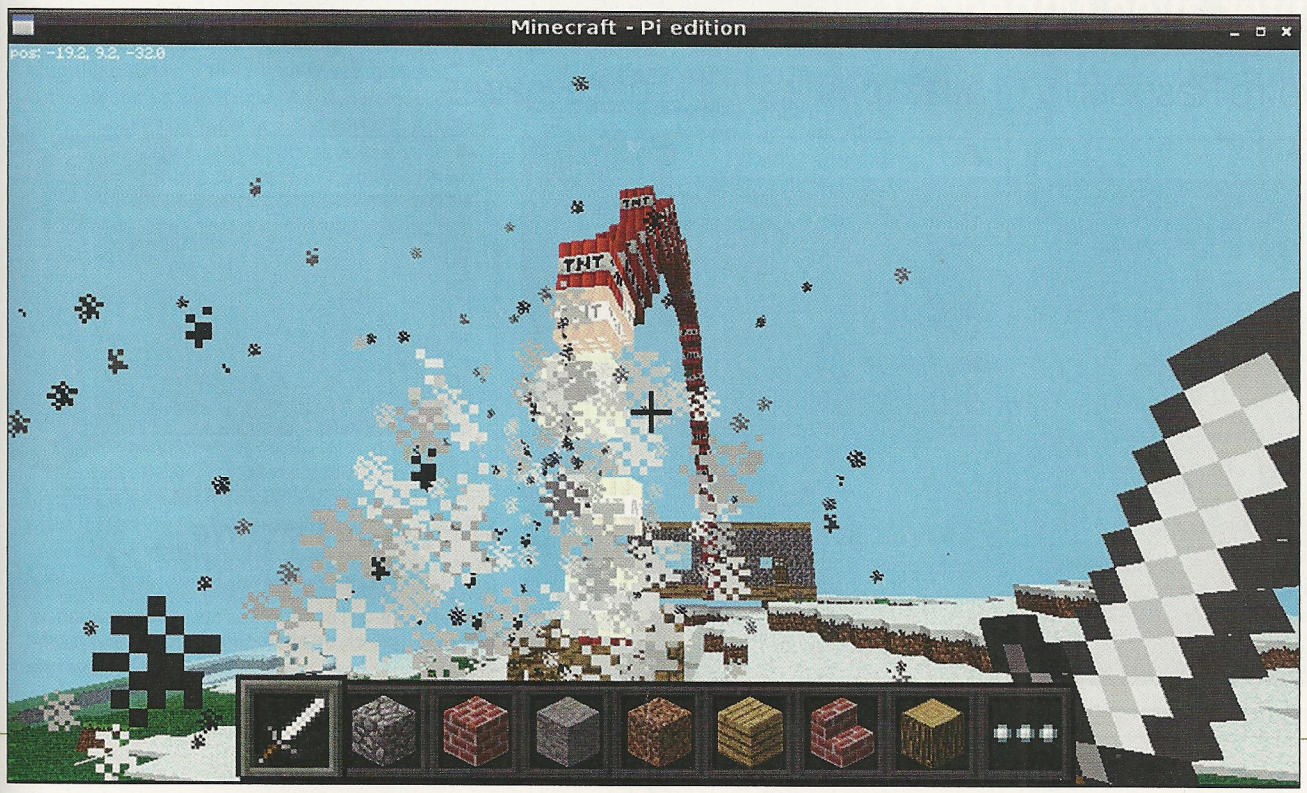
```
loop = True
while(loop):
        vertices.append(minor[:idx] + [X] + minor[idx:])
        if X == X2:
                loop = False
        for j in range(2):
                if dminor[j] >= 0:
                        minor[j] += s[j + 1]
                dminor[j] -= aX
        dminor[j] += aminor[j]
        X += s[0]
return vertices
```

› **Nous pouvons désormais nous affranchir de la grille et construire selon les angles de notre choix.**

Pour conclure le projet, nous testons la fonction en ajoutant un tas de bois à côté de notre position, comme un ultime témoignage du dur labeur conduit ce jour.

```
v1 = mc.player.getTilePos() + minecraft.Vec3(1,1,0)
v1 = minecraft.Vec3(1,1,0) + pos
v2 = v1 + minecraft.Vec3(5,5,5)
bline = getLine([v1.x,v1.y,v1.z],[v2.x,v2.y,v2.z])
for j in bline:
        mc.setBlock(j[0],j[1],j[2],5)
```

Et voilà ! Minecraft Pi Edition illustre à merveille les joies de la programmation en Python. Vous aurez tant de plaisir que vous ne réaliserez même pas que vous êtes en train d'apprendre !



› **N'essayez pas ça à la maison, les enfants.**