Minecraft 与 LittleTiles
导出 Obj 基础技术研究文档

08/14/2024 V2 草稿

目录

Chapter 1　《Minecraft》的基础知识

1.1　《Minecraft》中的坐标系

在《Minecraft》中坐标系分很多种（更多的是基于技术上的坐标系转换之用）。在游戏中，通常都使用右手坐标系，如图 *Figure 1* 所示。

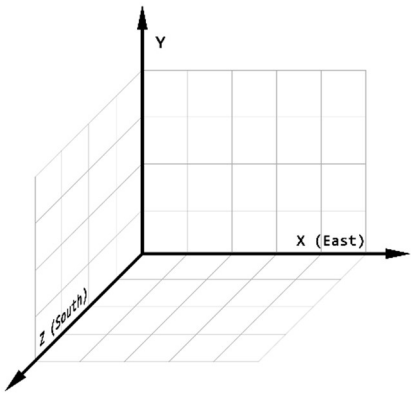


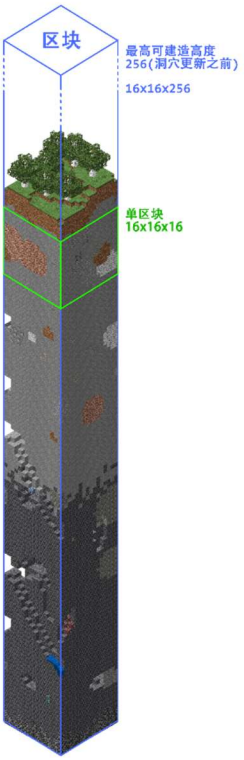
Figure 1 右手坐标系示意图

玩家坐标将会称为“**世界玩家坐标**[1]”，它使用三个 32 位浮点值来分别表示 $\{x, y, z\}$ 三个轴向的坐标。

方块坐标将会称为“**世界方块坐标**[2]”，它使用三个 32 位的有符号整数来分别表示 $\{x, y, z\}$ 三个轴向的坐标。

对于**区块坐标**[3]，它使用两个 32 位有符号整数来分别表示 $\{x, z\}$ 两个轴向的坐标。每个区块都由 $16 \times 16 \times 256$ 的区域构成；也就是说，它把整个世界按 16×16 的大小进行划分，然后单独对该区块在世界中的位置进行标识。

在一个区块中，会按高度进行进一步划分，我们将其称为“**单区块坐标**[4]”，它通过两个 32 位有符号整数来分别表示 $\{x, y, z\}$ 三个轴向的坐标。它的 x 和 z 坐标，与其所在区块的 x和 z 是相同的，y 则是在区块内部，以 16 为单位进行划分标识。



[1] 世界玩家坐标 *World Player Coordinates*
[2] 世界方块坐标 *World Block Coordinates*
[3] 区块坐标 *Chunk Coordinates*
[4] 单区块坐标 *Single Chunk Coordinates*

1.2 《Minecraft》中存档的存储方式

在 1.2 版本之后，《Minecraft》以 Anvil 格式存储游戏存档，即 mca 文件。在上一节中我们了解到区块，在一个 mca 文件中，它可以存储 32×32 个区块数据。

所以，实际上《Minecraft》将整个游戏世界以 32×32 个区块为单位进行划分，我们将其称为区域。我们将每个区域都存储到一个对应的 mca 文件中。

对此我们在游戏外部也要提出一个坐标系，它是区域坐标，它通过两个 32 位有符号整数来分别标识区域坐标的 x 和 z。

1.2.1 Anvil 格式

Anvil 格式用于指导如何存储一个地图块数据，其 mca 文件名用于指出该 mca 文件的区域坐标，格式为 $r.x.z.mca$。

在 Anvil 格式中，前 8KiB 被称为 8KiB 区，它用于存储区块的索引数据和区块最后修改时间，剩余部分是用于存储区块已压缩数据的多个扇区。每个扇区的大小为 4096 字节，一个区块可以占用多个扇区。

因此 mca 文件大小一定为 4096 字节的倍数，《Minecraft》不接受非 4096 字节的倍数的 mca 文件。

1.2.2 8KiB 区

8KiB 区分为两个 4KiB 区，其中前者提供区块的索引相关数据，后者提供区块最后修改的时间戳。

两个 4KiB 区都是以 4 字节为单位进行分割的，其分别存储 $\{x, z\}$ 区块的偏移量和该区块的最后修改时间。

注意同样的，先从 x 轴开始遍历，再遍历 z 轴：

$$\{0,0\}, \{1,0\}, \{2,0\}, \dots, \{29,31\}, \{30,31\}, \{31,31\}$$

前 4KiB 区每组的前 3 字节为偏移量，以大端序存储，单位为 4096 字节，后 1 字节为该区块所用的扇区数量：

Byte	1	2	3	4
描述	偏移量			扇区数量

要计算 $\{x, z\}$ 区块在前后 4KiB 区中的具体字节位置，可通过如下公式得到：

$$4 \times \left((x \ mod \ 32) + (z \ mod \ 32) \times 32\right)$$

在编程中，如 Java/C/C++ 可能会出现负数，所以需要使用 & 运算符而不是取模来进行计算：

$$4 * ((x \ \& \ 31) + (z \ \& \ 31) * 32)$$

例如，区块 [3,12] 计算后为 1548 字节，即该区块的偏移量在前 4KiB 区的第 1548 字节的位置。最有以大端序读取偏移量，并将结果乘以 4096 即可得到该区块在 mca 文件中的字节位置，然后根据其扇区数量读取对应的扇区。

到这并没有真正获取到区块的数据，只是获取了该区块的压缩数据。

最后修改时间同理，在后 4KiB 的第 1548 字节处。

1.2.3 区块数据

在上一节中我们获取了指定区块在文件中对应的扇区位置，在每个区块第一个扇区的起始位置，有 5 个字节用于记录该区块压缩数据的信息，前 4 个字节存储有效数据的长度（单位为字节），后 1 个字节存储压缩类型标识符：

Byte	1	2	3	4	5
描述	有效数据长度				压缩类型

目前压缩类型定义了三种方案：

标识符（十进制）	方案
1	GZip （RFC1952）（未使用）
2	Zlib （RFC1950）
3 (1.15.1 之前的版本)	未压缩 （未使用）

当获取到有效数据长度时，可以提取其对应的压缩数据，并根据压缩类型对这段数据进行解压，即可得到 NBT 的二进制格式。

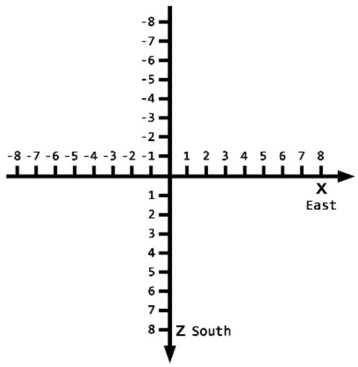
注意这里指的是原始二进制格式，而不是解析后的 SNBT 格式或解析后的树状 NBT 结构。

1.3 坐标系的换算

1.3.1 重要思想

对于从一个坐标系换算到另一个坐标系，我们不采用矩阵的方式来进行计算。而是对要换算的东西做一些抽象的理解。

由于在该项目中，我们不需要对高度也进行划分，所以实际上我们所涉及的换算还只是二维平面的坐标系。



对于二维坐标轴的第一象限（在这里是右下角），对它进行坐标系换算相对来说是比较容易的。

从世界玩家或方块坐标换算到其所处的区块坐标、区域坐标只需要简单的除法即可运算，也无需区分正负轴。但如果要计算所处区域文件中第几个区块的坐标就要考虑很多事情。

对于这类问题我们在计算第一象限的时候，只需对 x 和 z 值进行取余运算即可。但是如果到了 x 或 z 的负数轴上就有点棘手了。

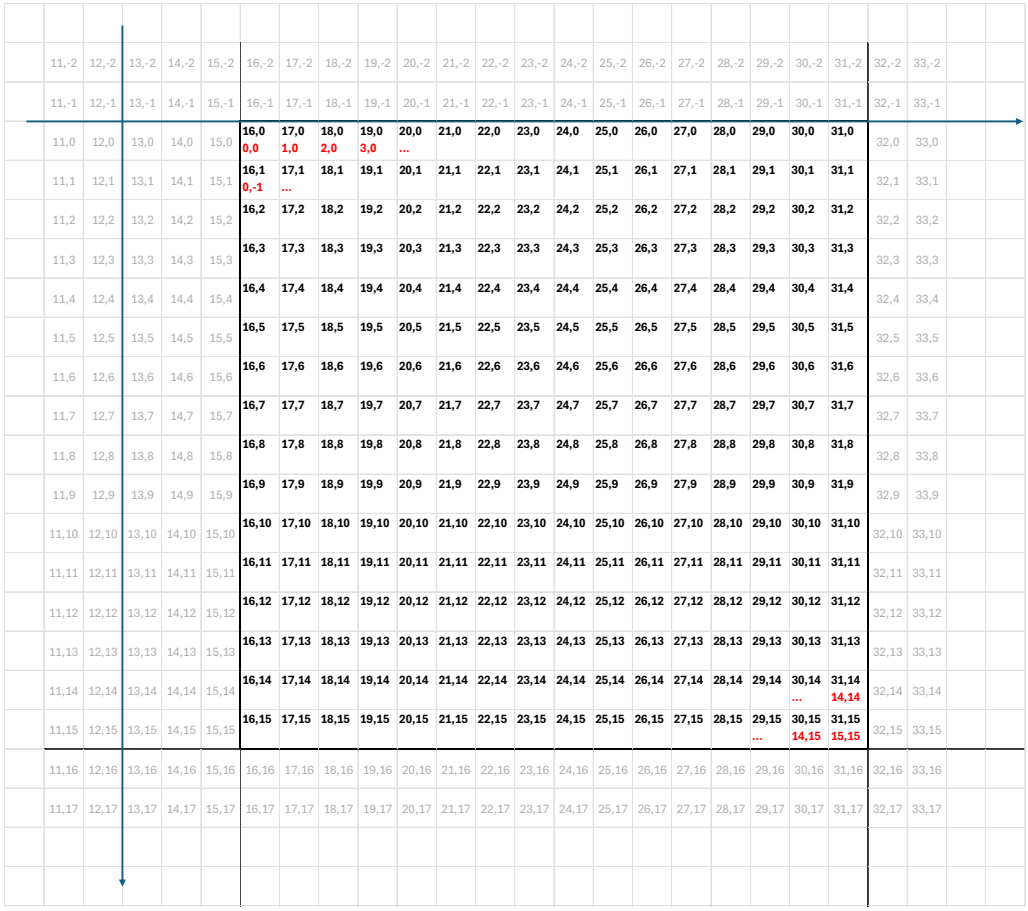
对此我的思想是将负数轴转变为正数轴；然后因为负数是从 1 开始计数，而正数轴是从 0 开始计数的，所以我们需要对其 -1 偏移一格。

最后就像正数轴的处理方式一样，进行取余运算即可。

这一部分可能有点抽象，我们来看后面几节的示例演算。

1.3.2 世界方块坐标转区块方块坐标

在一个区块中，每个方块都有一个在这个区块中的坐标。用 32 位有符号整数表示三个轴向的坐标 $\{x, y, z\}$。



该图黑色方框框起来的区域就是一个区块的范围（16×16），红色坐标标识了改区块内的方块坐标。

要从世界坐标，比如 $\{18,0\}$ 转换到区块方块坐标，计算如下：

$$x = 18 \,\%\, 16 = 2$$

$$z = 0 \,\%\, 16 = 0$$

$16 \overline{)18}^{1}$ 余 2

$16 - 16 = 2$

所以最终的区块方块坐标为 $\{2,0\}$；那对于坐标 $\{25,12\}$ 呢？

$x = 25 \,\%\, 16 = 9$

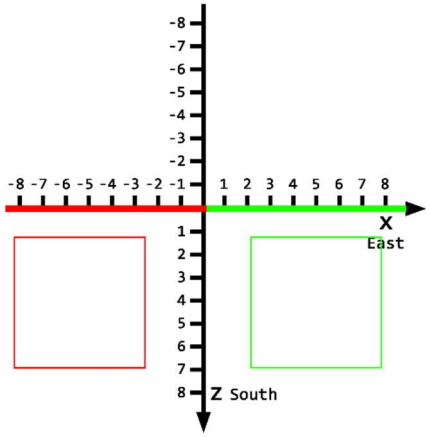
$z = 12 \,\%\, 16 = 12$

$\{9, 12\}$

$16 \overline{)25}^{1}$，$25 - 16 = 9$

$16 \overline{)12}^{0}$，$12 - 0 = 12$

这只是对于正轴的情况，如果出现负数轴呢？



如上图所示，假设我们要计算的区域在负数轴部分，我们只需要将其镜像到正轴，就可以像第一象限那样进行取余运算。

首先是取 x 的绝对值，然后对其-1，因为要将其向左移动一格。因为坐标是从 0 开始计数的，而负数轴则是从 1 开始，毕竟 (0,0) 不能同时属于两个区块吧。

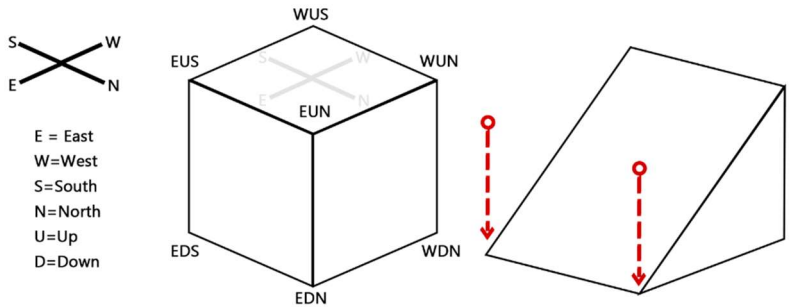
当然这里不是镜像，只是把他向右移动到了整数轴。但我们先进行取余运算，然后我们对最后得到的结果用 15 减去它，我们就完成了最后的镜像，同时我们也得到了该世界方块坐标在区块中的坐标。

对于 z 轴也同理，如果两个都为负，就将其都进行镜像到第一象限。

Chapter 2 《LittleTiles》的基础知识

2.1 Little tiles 的基本单位

LittleTiles 的 Tile 是以《Minecraft》中的一个方块为单位合并存储的。并将这些数据存储到区块文件中。每个 Tile 实际上是一个立体矩形，即使它是一个斜面；它由八个顶点组成：



上图可以看到这八个顶点（除了隐藏在后面的 WDS），它们的命名依照四个方向命名，在名字中间添加 U/D 以表示它是上面的顶点还是下面的顶点。

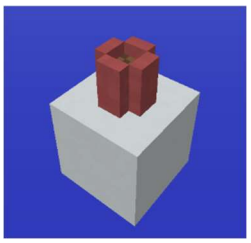
例如 EUS 就代表东南方向，上方，如图所见。

有时候你会选择去做一个斜面，打破《Minecraft》的限制。在制造斜面的时候，实际上只是把这八个顶点中的一些或全部进行了三个方向的移动（x,y,z）。

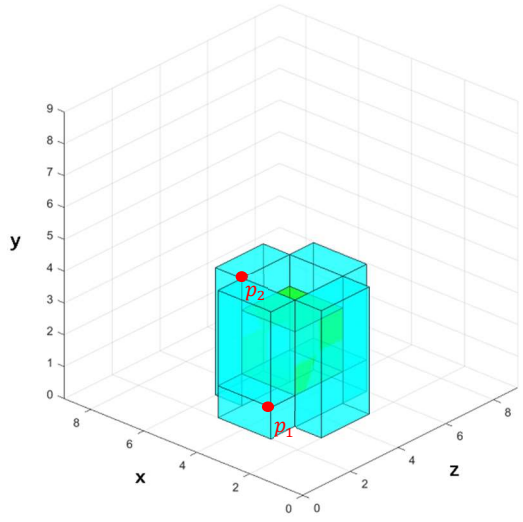
例如上图右侧的图形，只是把 EUS 和 EUN 沿着 y 轴向下移动，直到与 EDS 和 EDN 重合。具体的偏移数值，与 Tile 和方块的细分级别（grid）都有关联，后续介绍。

2.2 Little tiles 的块

上面我们了解到，Tile 存储于方块之中，并以此为单位存储在区块之中。那么在一个方块中，体素的存储我们有一个花盆示例，它周围包裹着粉红的陶瓦，中间是泥土。



它的精度为 8，具体如右图所示：



在左侧图中，青色部分就是粉红色陶瓦，而中间绿色的部分就是泥土。

我们可以清晰的看出这个三维图像在8x8x8（精度）的空间内组合了多个Tile，最终形成我们在游戏里的画面。

每个体素都有两个坐标，它定义了立体矩形在这个三维空间中的两个顶点。

例如最外面的那个 Tile，它在该块中的坐标就是:

$$p_1(3,1,2), p_2(5,4,3)$$

具体底层存储主要数据如下：

```
Tiles info (BLOCK):
        block coord: 1 5 2                  // 方块坐标
        grid: 8                             // 精度（细分等级）
        id: minecraft:littletilestileentity
        boxes count: 2

      Tile info:
            block id: minecraft:dirt        // Tile 的方块类型
            boxes:  [3, 1, 3, 5, 3, 5]      // 定义了立体矩形两顶点坐标
                                            // [x1, y1, z1, x2, y2, z2]

      Tile info:
            block id: minecraft:stained_hardened_clay:6
            boxes:  [3, 0, 2, 5, 1, 6]
                    [3, 0, 3, 3, 4, 5]
                    [5, 0, 3, 6, 4, 5]
                    [3, 1, 2, 5, 4, 3]      // 示例中提到的那个 Tile
                    [3, 0, 5, 5, 4, 5]
```

2.3 Tile 的具体表示

2.3.1 角的偏移

之前我们说 Tile 有八个顶点，这八个顶点在方块中可以通过立体矩形的两个坐标进行确定。所以表示八个顶点的位置，只需知道它在方块中的两个坐标即可确定。

所以你会在 SNBT 标签中看到一个数组，它的前 6 位表示体素在块中的坐标。

$$bBox: [I; x_1, y_1, z_1, x_2, y_2, z_2, \dots]$$

$$\begin{cases} p_1 = (x_1, y_1, z_1) \\ p_2 = (x_2, y_2, z_2) \end{cases}$$

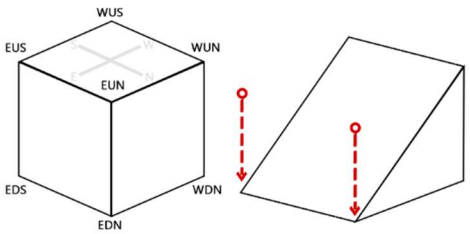
而后面有时候也会跟一堆数字，这些数字就是该 Tile 八个顶点的偏移数据。首先坐标后面紧跟着的第一个数字，它表示了哪些顶点进行了偏移，哪些没有。后面的数据则是按顺序列出的具体偏移量。

符号位永远为负(1)	空	Flipped						角																								
								WDS			WDN			WUS			WUN			EDS			EDN			EUS			EUN		
		E	W	S	N	U	D	Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X
1	0																															
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

上图从高位到低位有序列出了该 32 位有符号整数的每一个位的作用。符号位永远位 1，它代表负号。是 0 时，那它就成了正数（当然 Lt 不希望出现这样的情况）。

我们先从右往左看，每三个位代表一个顶点，它定义了一个顶点的 x,y,z 这三个轴。如果哪个轴移动了，它就会标识为 1，没有则是 0。

我们回到之前的例子，假设它的精度是 2。此时我对 EUS 和 EUN 都沿着 y 轴向下偏移了 2，那么它应该是右边的图形：



这张图的位设定是这样的：

符号位永远为负(1)	空	Flipped						角																								
								WDS			WDN			WUS			WUN			EDS			EDN			EUS			EUN		
		E	W	S	N	U	D	Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X	Z	Y	X
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	1	0	
31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	

我们可以看到 EUS 和 EUN 的 Y 都被设置成了 1，这说明这两个角都沿 y 轴进行了偏移。

这段二进制转换为十进制就变成了-2147483630。这也是为什么我们在导出 Lt 蓝图的时候，那串字符中可能出现许多很大的负数。

$$bBox: [I; 0,0,0,1,1,1, -2147483630, \dots]$$

既然我们知道了如何设定哪些角进行了偏移，那么后面的数字就代表了被偏移的角具体偏移了多少。它按照顺序排列，让我们先忽略掉哪些角，只看下一行的那些 xyz。其实这些 xyz 都表示是否沿着这个轴进行了偏移，所以我们也忽略掉这些 xyz，只看是否有轴被偏移。

所以我们知道，最多需要 24 个值来表示这 24 个偏移量/是否偏移（8 × 3）。但是每次把没有偏移的角都在后面进行存储，会非常浪费空间。所以我们只存储偏移过的角。

在例子中我们偏移了 EUN 的 Y 和 EUS 的 Y，所以我们会在后面需要两个值来表示。它的顺序就是从右往左依次写入偏移值的。

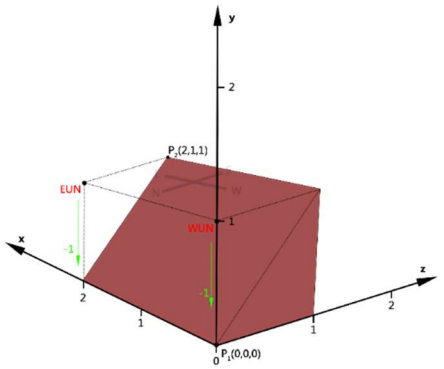
所以先写入 EUN 的 Y，然后才是 EUS 的 Y。读取的时候也是根据 Tile 坐标后面的那个很大的负数，查看哪些角有偏移。然后读取后面的数据，按照表从右往左的顺序读入，去定义后面的值都是哪些轴的偏移量。没有偏移的不进行读取，直接设为 0。

但是，这些角偏移量是以 32bit 为单位存储的，但是偏移量只有 16bit，所以在读取的时候，我们需要将 32bit 的有符号数拆成两份 16bit 的有符号数，它们分别都代表了一个偏移量。

所以数组中的第 8 及以后的数，它实际上每个数都代表了两个偏移量。

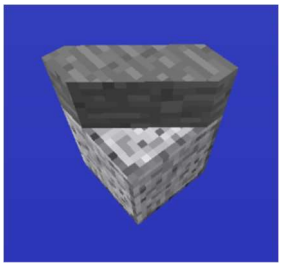
另一个更直观的例子是：

```
Tiles info (BLOCK):
    block coord: 15 4 10
    grid: 2
    id: minecraft:littletilestileentity
    boxes count: 1
    Tile info:
        block id: minecraft:stained_hardened_clay:6
        boxes:  [0, 0, 0, 2, 1, 1, -2147475454, -1]
        Angle change state (from -2147475454):
        |FLIPPED|WDS|WDN|WUS|WUN|EDS|EDN|EUS|EUN|
        | EWSNUD|zyx|zyx|zyx|zyx|zyx|zyx|zyx|zyx|
        | 000000|000|000|000|010|000|000|000|010|
        Angle offset (from -1):
        WUN:  y_offset: -1
        EUN:  y_offset: -1
tips:
  P_1(0,0,0) 来自 [0,0,0,2,1,1,...]
  P_2(2,1,1) 来自 [0,0,0,2,1,1,...]
  -1 的二进制表示为 11111111,11111111,11111111,11111111 (32bit)
  WUN y_offset 来自前16位，即 11111111,11111111
  EUN y_offset 来自后16位，即 11111111,11111111
  grid 代表该方块内的小方块精度是 2
```

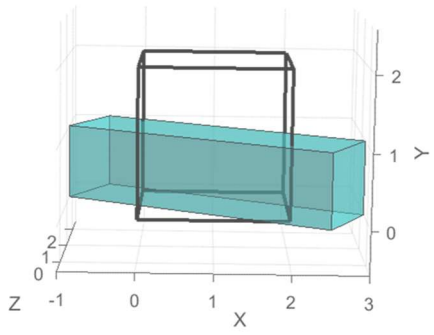


对于一些其它斜面，它的偏移量可能会超出方块的边界，例如右侧的小方块，它的精度为 2，各顶点的偏移量为：

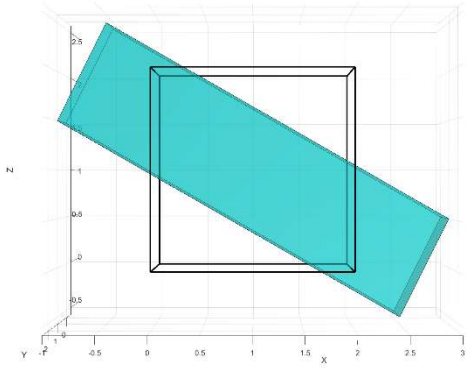
WDS:	x_offset: -2	z_offset: 2
WDN:	x_offset: 3	z_offset: -2
WUS:	x_offset: -2	z_offset: 2
WUN:	x_offset: 3	z_offset: -2
EDS:	x_offset: -3	z_offset: 2
EDN:	x_offset: 2	z_offset: -2
EUS:	x_offset: -3	z_offset: 2
EUN:	x_offset: 2	z_offset: -2

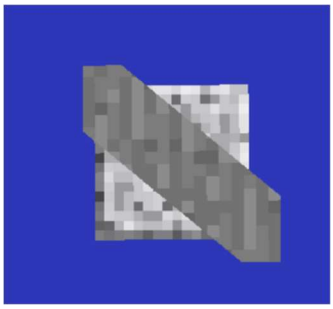


我们可以发现，它的顶点均超出了方块边界，它的实际效果是这样：



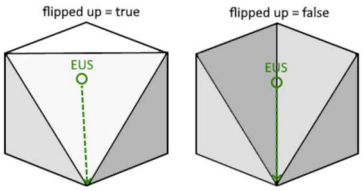
我们在游戏中看到的只是根据方块边界裁剪后的立体图形：





2.3.2 Flipped 位的作用

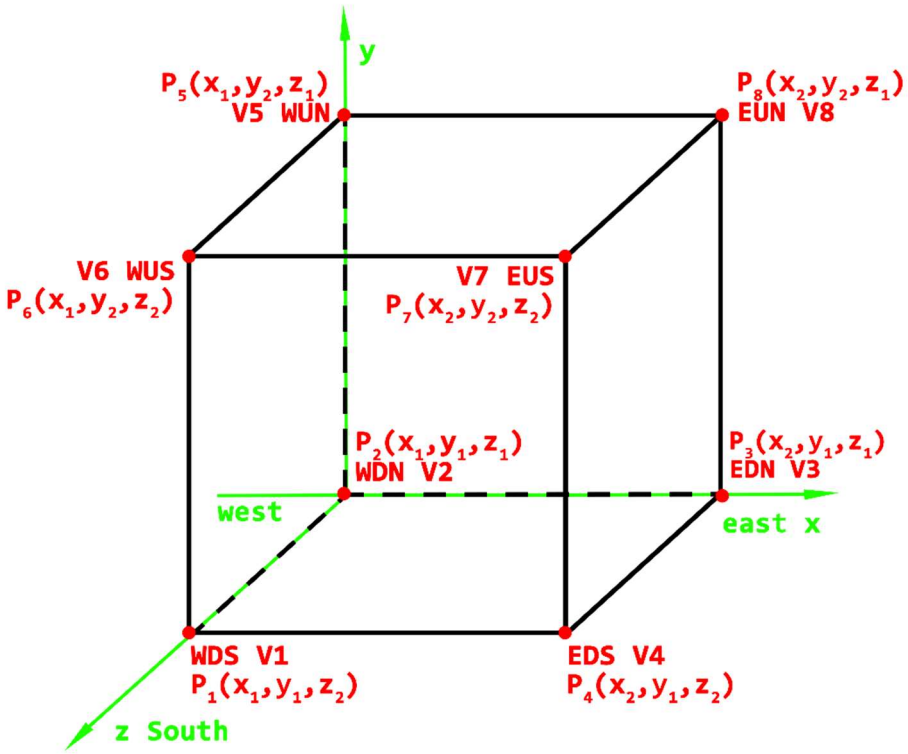
Flipped					
E	W	S	N	U	D
0	0	0	0	0	0
29	28	27	26	25	24



Flipped 位在上图中可以看出，E、W、S、N、U 和 D 都代表了一个立体矩形六个面中的一个面。

每个面都有一个比特位，该比特位用于定义三角形的切割方式，一个面有两种切割方式。该位仅对会产生三角面的偏移有效。

Chapter 3 附录 A：方块的点、坐标、面与轴方向示意图



DOWN V1 V2 V3 V4

UP V5 V6 V7 V8

NORTH V8 V3 V2 V5

SOUTH V6 V1 V4 V7

WEST V5 V2 V1 V6

EAST V7 V4 V3 V8