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

第三版

05/05/2025

目录

第三版前言

前两版的技术文档深入剖析了 Minecraft 与 LittleTiles 模组的底层技术原理，但由于配图和描述不够严谨，导致读者在理解时存在一定歧义。因此，第三版将对内容进行更为细致和准确的讲解，并配合程序代码示例详细说明。

本技术文档将与 GitHub 项目 Minecraft-LittleTiles-Reader[1]（v2 主分支）配合讲解，旨在提高代码的可读性。文档还将同步在 Inception 工作室[2]官网一并发布。

此外，由于第一版的 Minecraft-LittleTiles-Reader（v1 分支）在其它电脑上配置运行环境相对困难，作者正在开发第二版。现已完成第一版功能的重构，支持白模分区块导出 obj 模型。该版本采用 CMake 进行项目依赖和构建管理，从而简化配置流程。在此，特别感谢 GitHub 用户 suzakuwcx 对该项目的贡献，尤其是在 CMake 部分的工作。

在查阅此文档之前需先具备基本计算机素养与知识储备，如基本的二进制及其位操作。同时读者也需至少了解一种编程语言。

*本资料 Minecraft 部分参考 Minecraft Wiki[3]。

[1] https://github.com/Gaksy/Minecraft-Littletiles-Reader
[2] https://inception.work
[3] https://zh.minecraft.wiki/

此页留白

Chapter 1: Minecraft 的基础知识

1.1. Minecraft 中的坐标系

在 Minecraft 会有多种不同的坐标系，但通常都使用右手坐标系。我们在该文档中会涉及到**世界坐标**、**方块坐标**、**区块[4]坐标**、**子区块[5]坐标**和**区块方块坐标**等。

1.1.1. 世界坐标

世界坐标

世界坐标是由三个绝对坐标 (x, y, z) 组成的一个有序的实数三元组。

- x 表示该点在世界原点 $(0,0,0)$ 沿 x 轴方向上的有向距离；
- y 表示该点在世界原点 $(0,0,0)$ 沿 y 轴方向上的有向距离；
- z 表示该点在世界原点 $(0,0,0)$ 沿 z 轴方向上的有向距离。

这三个坐标值共同确定了该点相对于世界原点的绝对空间位置。

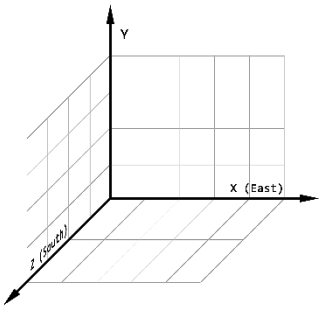

图 1 右手坐标系示意图

坐标系

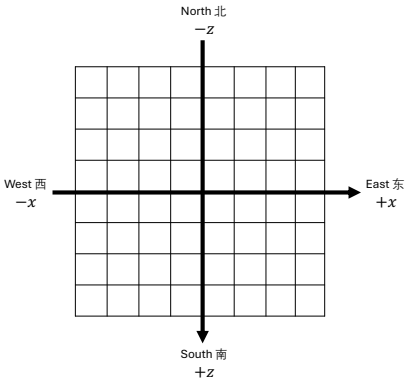

图 2 坐标系轴向俯视示意图

世界坐标基于一个由相互垂直且交于一点（即原点）的三条坐标轴形成的网格，即一个空间直角坐标系。**坐标系的单位长度为一个方块长**。

- x 轴的正方向为东，反方向为西，其坐标反映了距离原点在东（+）西（−）方向上的距离。
- z 轴的正方向为南，反方向为北，其坐标反映了距离原点在南（+）北（−）方向上的距离。
- y 轴的正方向为上，反方向为下，其坐标反映了距离原点的距离。

三条坐标轴形成了右手坐标系（拇指为 x 轴，食指为 y 轴，中指为 z 轴）。

[4] 区块 *Chunk*
[5] 子区块 *Sub chunk/Chunk Section*

方块坐标

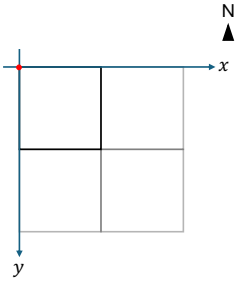

图 3 方块坐标示意图

一个方块的坐标实际上是这个方块的西北下角那一点的坐标，即方块内的坐标向下取整得到的整数坐标。在游戏中，一个小数坐标通常需要通过向下取整转换成整数坐标，这个整数坐标称为原坐标的方块坐标。

1.1.2. 区块

区块宽 16，长 16，且水平方向上与 16 整数倍的位置对齐。由于我们所探讨的版本在洞穴更新之前，故高度为 256。

区块还可以再更加细分为子区块，子区块高 16 格。

对于区块坐标，由于没有高度区分，所以由两个 **32 位有符号整数**[6]表示 (x, z) 两个轴向的坐标。子区块坐标使用三个 32 位有符号整数来分别表示 (x, y, z) 三个轴向的坐标。

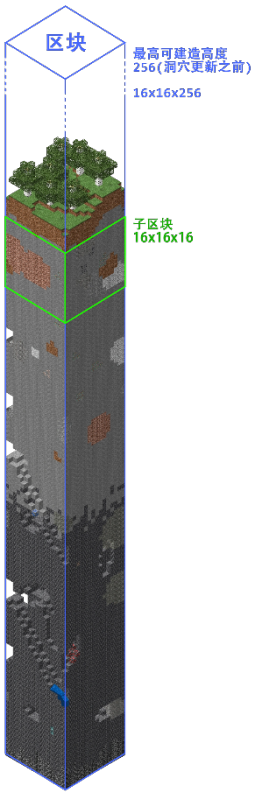

图 4 区块与子区块示意图

1.2. Anvil——区域文件格式

在 1.2 版本之后，Minecraft 以 Anvil 格式存储区块数据，即 mca 文件（区域文件）。

一个区域文件存储的范围被称为区域[7]（也称为地图块）。一个区域的大小是 32×32 区块，并且区块坐标与 32 对齐。

所以区域坐标也由两个 32 位有符号整数来分别表示 (x, z) 两个轴向的坐标。

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 文件的文件名用于指出该 mca 文件所属的区域坐标，如果区域的坐标为 (x, z) 则其文件命名格式为 r.***x***.***z***.mca。

区块坐标换算为区域坐标需要除以 32 并向下取整（或右移 5 位）：

```
galib::Minecraft:: ChunkCoordToRegionCoord
static_cast<std::int32_t>(std::floor(kOreChunkCoord.x / 32.0))
static_cast<std::int32_t>(std::floor(kOreChunkCoord.z / 32.0))
```

图表 1 区块坐标转区域坐标

[6] 在 *C++*中 32 位有符号整数定义在 `cinttype` 头文件中，类型为 `std::int32_t`。
[7] 区域 *Region*

既然区块被存储在区域文件中，虽然区块有其世界区块坐标，但为了方便在区域文件中存储，他还拥有**区域内的局部坐标**。

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. 区域内的局部坐标换算

局部坐标由于涉及对负数部分的处理，所以转换过程需要多几步计算。我们将涉及到负数部分的坐标做一个变换，将其视为正数处理[8]，随后我们就可以对其通过取余 32 得到对应的局部坐标。

需要注意的是，如果我们将负数轴镜像到正数轴，需要对其-1，因为负数轴是从-1 开始计数，而正数轴是从 0 开始的。

在*图表 2* 中，`ore_x` 是 x 或 z 坐标。（因为 x 与 z 都经过相同的计算）`desc_x` 是计算所需要的中间变量，也是最终结果。

```
galib::minecraft::CoordSwap
NumericType desc_x;
if (ore_x < 0) {
    desc_x = -ore_x - 1;
    desc_x %= 32;
    desc_x = (32 - 1) - desc_x;
} else {
    desc_x = ore_x % 32;
}
return desc_x;
```

图表 2 局部坐标换算

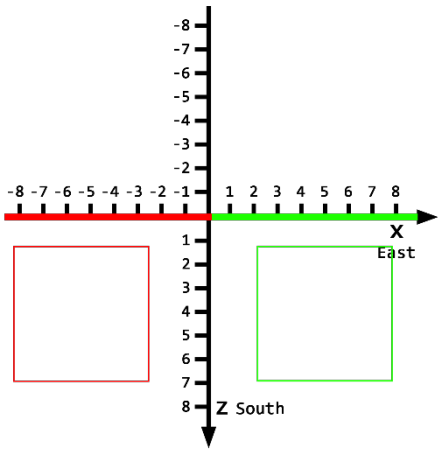

图 5 局部坐标换算示意图

`ore_x < 0` 用以检查是否在负数轴，如果在就需要将其镜像到正数轴，反之正常取 32 的余数得到局部坐标值。

`-ore_x - 1` 用以将值镜像到正数轴并 -1 。

随后对其得数取 32 的余数，由于我们的 `-ore_x - 1` 操作相当于镜像到了正数轴，所以我们需要再镜像回去，所以我们对结果进行`(32-1)-desc_x` 计算，最终我们得出了我们所需要的局部坐标值。

但通过位运算，我们可以很简单的得出结果。我们对 x 或 z 应用按位与运算 `0x1F` 即可得到结果，当然只适用于取 32 的余数的情况。

虽然按位与操作可以快速得出结果，但考虑到执行数量且需要计算取余 16 的情况，故该项目保留该计算方式，而非直接的按位与运算。

[8] 这样做不会影响最终局部坐标的正确性，因为我们计算的是相对坐标，这一步相当于将其平移到了正数部分方便我们计算。

1.2.3. $8KiB$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（使用大端序保存纪元秒，如果未创建过则为 0）。

两个 $4KiB$ 区都是以 4 字节为单位进行划分的，其分别存储 (x,z) 区块的偏移量和该区块的最后修改时间戳。其遍历循序应先从 x 轴开始遍历，再遍历 z 轴（x 与 z 都从 0 开始，最大值为 31）：

$$(0,0),(1,0),(2,0),\dots,(29,31),(30,31),(31,31)$$

前 $4KiB$ 区每组的前 3 个字节为偏移量，以大端序存储，单位为 4096 字节；最后一个字节为该区块所用扇区数，一个扇区为 $4KiB$[9]。

字节	1	2	3	4
描述	偏移量			扇区数

要计算 (x,z) 区块（世界区块坐标）在 $4KiB$ 区中的具体字节位置，可通过如下公式的到：

$$4\times\big((x\ mod\ 32)+(z\ mod\ 32)\times 32\big)$$

在编程中，可能会出现负数，所以需要使用 & 运算符而不是取模运算符来进行计算：

```
4 * ((x & 0x1F) + (z & 0x1F) * 32)
```

1.2.4. 获取区块索引与偏移量示例

`galib::minecraft::Anvil::getChunkConstIterator_()`

```
// Calculating chunk offset index
const ByteIndex chunk_offset_index = 4 * (kRegionChunkCoord.x + kRegionChunkCoord.z * 32);

// Calculating chunk offset
ByteIndex chunk_offset = 0;
chunk_offset = chunk_offset | static_cast<uint8_t>(kMcaData[chunk_offset_index]) << 16;
chunk_offset = chunk_offset | static_cast<uint8_t>(kMcaData[chunk_offset_index + 1]) << 8;
chunk_offset = chunk_offset | static_cast<uint8_t>(kMcaData[chunk_offset_index + 2]);
```

图表 3 计算区块索引与偏移

在项目中的“获取区块索引”的函数先计算区块在前 $4KiB$ 区的索引获取区块的偏移量[10]，即 `chunk_offset_index`。

`kMcaData` 为 mca 文件数据，按字节读取，每一个元素一个字节。在获取区块偏移量时将其转换为 uint8_t 是因为读取文件时是按 char 类型读取的，它可能是一个有符号类型，如果对其进行位移运算会发生意想不到的效果。所以这里将其转为 uint8_t 之后再进行位移处理。

计算区块偏移量时遵循 Anvil 文件规定使用大端序，且区块索引为三个字节。

[9] 由于占用扇区数被保存为一个字节，所以在区域文件中区块数据最多只能保存 $255\times 4KiB$ $(1020KiB)$ 如果超出了这个数字，游戏会使用额外文件辅助保存这个区块。
区域文件最多保存 1024 个区块，如果所有区块都保存了 $1020KiB$ 数据，那么可以得出区域文件最大大小为 $1044488KiB(0.996GiB)$。
[10] 这里没有用按位与是因为事前已经计算好了区域局部坐标。

1.2.5. 区块的读取

起始扇区偏移是从文件开始计算的，即第一个区块数据的偏移为 2 （扇区），剩余的区块数据偏移也至少为 2 （扇区）。

如果起始扇区偏移和占用扇区数都为 0，则代表对应区块不存在或未创建。

每个区块数据的前五个字节为区块元信息，其中前四个字节存储有效数据的长度（单位为字节），最后一个字节存储压缩类型标识符：

字节	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[11] 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。

1.2.6. 获取区块二进制 NBT 示例

```
galib::minecraft::Anvil::getChunkConstIterator_()
// Get chunk length
const ByteIndex chunk_length = static_cast<uint8_t>(kMcaData[chunk_offset_index + 3]);
```

图表 4 获取扇区大小

获取区块扇区大小，以确定区块数据范围。

```
galib::minecraft::Anvil::getChunkConstIterator_()
// Get desc chunk iterator index
const ByteIndex chunk_begin_index = chunk_offset * 4096;
const ByteIndex chunk_end_index = chunk_begin_index + chunk_length * 4096;

const ByteIndex chunk_valid_begin_index = chunk_begin_index + 5;
```

图表 5 获取区块数据范围

通过区块偏移量乘以 $4KiB$ 可获得区块起始位置（包区块元信息），然后根据扇区大小可以确定区块结束位置。

最后的区块有效起始位置就是区块起始位置增加 5 字节（区块元信息）的位置。

[11] NBT 格式说明：https://zh.minecraft.wiki/w/NBT%E6%A0%BC%E5%BC%8F

galib::minecraft::Anvil::getChunkConstIterator_()

```
ByteIndex chunk_valid_end_index = 0;

chunk_valid_end_index = chunk_valid_end_index | static_cast<uint8_t>(kMcaData[chunk_begin_index]) << 32;
chunk_valid_end_index = chunk_valid_end_index | static_cast<uint8_t>(kMcaData[chunk_begin_index + 1]) << 16;
chunk_valid_end_index = chunk_valid_end_index | static_cast<uint8_t>(kMcaData[chunk_begin_index + 2]) << 8;
chunk_valid_end_index = chunk_valid_end_index | static_cast<uint8_t>(kMcaData[chunk_begin_index + 3]);
ByteIndex chunk_valid_size = chunk_valid_end_index;
chunk_valid_end_index += chunk_valid_begin_index;
```

图表 6 获取区块有效数据结束位置

由于区块有效数据长度为四个字节，并以大端序存储，所以我们同样应用位操作获得有效数据长度去计算最终的区块有效数据结束位置（有效数据起始位置+有效数据长度）。

Chapter 2: LittleTiles 技术原理

2.1. LittleTiles 的基本单位

在 LittleTiles 中，一个方块可以被细分为多个方块，每个小方块我们称为 Tile。每个 Tile 实际上是一个立方体，即使它拥有一些斜面，它由八个顶点组成。

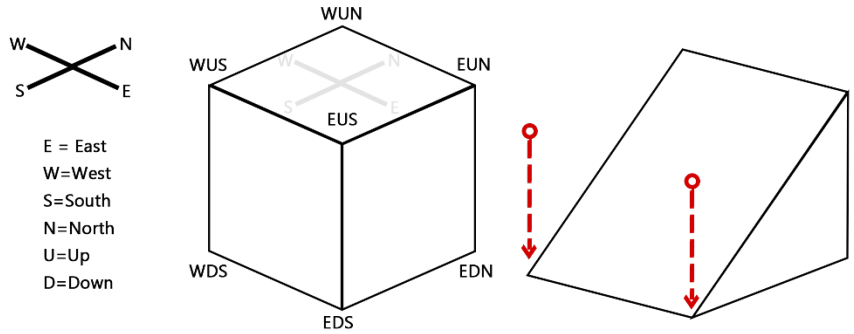

图 6 Tiles 顶点示意图

这里的每个顶点的命名都与其对应的方向有关，该方向为游戏世界中的东南西北。我们通过这八个顶点即可表示出一个立体矩形（不必是标准的正方体），随后如果需要表示斜面，小方块通过对定点施加偏移，从而创造斜面。

在*图 6* 右侧的立体图形就是通过将 `EUS` 与 `EUN` 这两点偏移而来。

2.2. Tiles（瓦片）

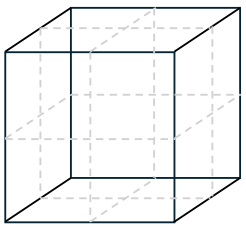

图 7 grid 为 2 时的网格

上面我们了解到一个方块可以包含多个 Tile 以形成更复杂的结构。接下来我们详细刨析它的几何结构是如何表示的。

在游戏中，我们经常去调整一个名为 `grid` 的参数以控制精细度，它控制的就是将方块分为多少等份。如 `grid=2` 时的网格如*图 7* 所示。

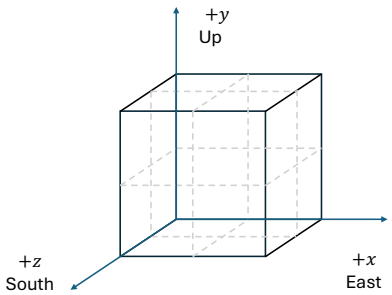

图 8 方块中的空间直角坐标系

此时的方块中就构成了一个空间直角坐标系，同样的它遵循我的世界的右手坐标系，其轴向与其保持一致，详见*图 8*。

2.2.1. Tile 的表示

当我们在方块中构建了一个 空间直角坐标系后，我们就可以利用该坐标系对 Tile 进行几何建模。前面我们提到，Tile 实际上是一个立方体，通过八个坐标点来分别表示立方体的八个顶点。

例如，当 grid = 4 时，一个大小为 2 × 2 的立方体如*图 9* 所示。

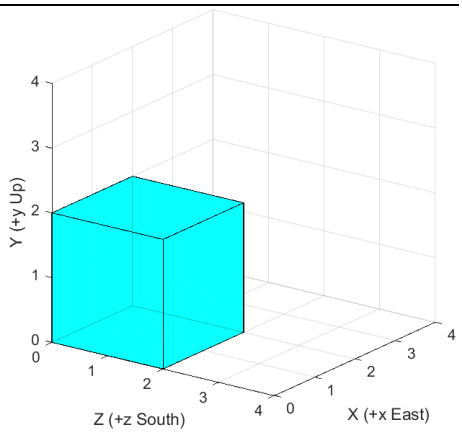

图 9 grid 为 4 时的立方体

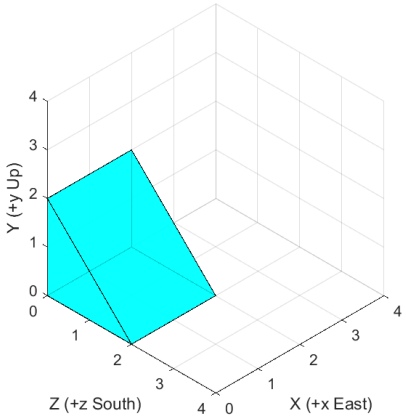

图 10 grid 为 4 且偏移后的立体图形

如果我们需要表示斜面，那么我们只需要更改这些点的坐标使其倾斜即可。

例如我们将 WUS 与 EUS 点分别将 Y 坐标 −2 以向下偏移，我们将得到如图 10 所示的立体图形。

2.2.2. 多个 Tiles 的表示

既然我们可以利用 grid 对方块进行细分并以此构建一个空间坐标系以表示几何结构，那么我们可以在该空间坐标系中表示多个 Tile。

我们有一个花盆示例、它周围是裹着粉红色的陶瓦。中心是混凝土，它的 grid 为 8，在游戏中具体如*图 11* 所示。

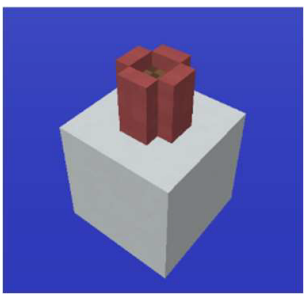

图 11 花盆示例

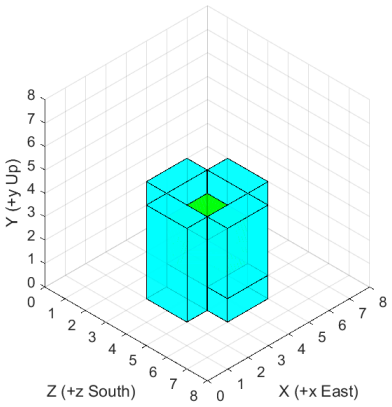

图 12 花盆空间坐标系几何结构示例

对此我们给出了该几何结构在空间坐标系中的表示，如*图 12* 所示。

即通过在一个方块中组合多个 Tile 即可在一个方块中表达更复杂的结构。

2.3. NBT 存储

LittleTiles 中的所有 Tile 都以方块为单位组织起来，这些数据被存入到第一章提到的区域文件中。当我们读取区域文件中的一个区块并解析为 **NBT 格式**[12]后，我们可以从根目录开始在 `/Level/TileEntities` 找到该区块中的所有 LittleTiles 方块。

`TileEntities` 是一个列表标签，其中的每一个元素都存储了 grid、方块 id、以及 tile 数据。其中每个 `TileEntities` 中的复合标签如下所示：

```
{
	content:{
		tiles: [
			{
				block: "minecraft:stained_hardened_clay:6",
				boxes: [
					[3,0,2,5,1,6],
					[5,0,3,6,4,5],
					[2,0,3,3,4,5],
					[3,1,5,5,4,6],
					[3,1,2,5,4,3]
				]
			},
			{
				block: "minecraft:dirt",
				box: [3,1,3,5,3,5]
			}
		]
	}
	grid: 8,
	// 方块坐标
	x: 0,
	y: 0,
	z: 0
}
```

图 13 花盆 NBT 示例

在该 NBT 结构中，content 标签为复合标签，其中的 tiles 为一个列表，其中每个元素都是复合标签，每个复合标签中都有 block，它标记了几何结构方块 ID（材质）。

前文我们提到所有的 Tile 本质上都是立体矩形，其斜面通过附加偏移来表示。所以所涉及到的坐标是两点坐标，在 boxes 或 box[13]的数组中前三个为 $pos_1(x_1, y_1, z_1)$，后三个为 $pos_2\ (x_2, y_2, z_2)$，由这两点构成一个立体矩形，boxes 表示有多种同材质的几何结构，box 则只有一个。

[12] NBT 格式说明：https://zh.minecraft.wiki/w/NBT%E6%A0%BC%E5%BC%8F
[13] 以下简称 boxes。

其中 pos_1 代表 WDN，pos_2 代表 EUS。最后 grid 代表网格大小，x、y、z 标签代表该方块的世界坐标。具体结构如*图 14* 所示（使用 NBTExporer[14]查看）。

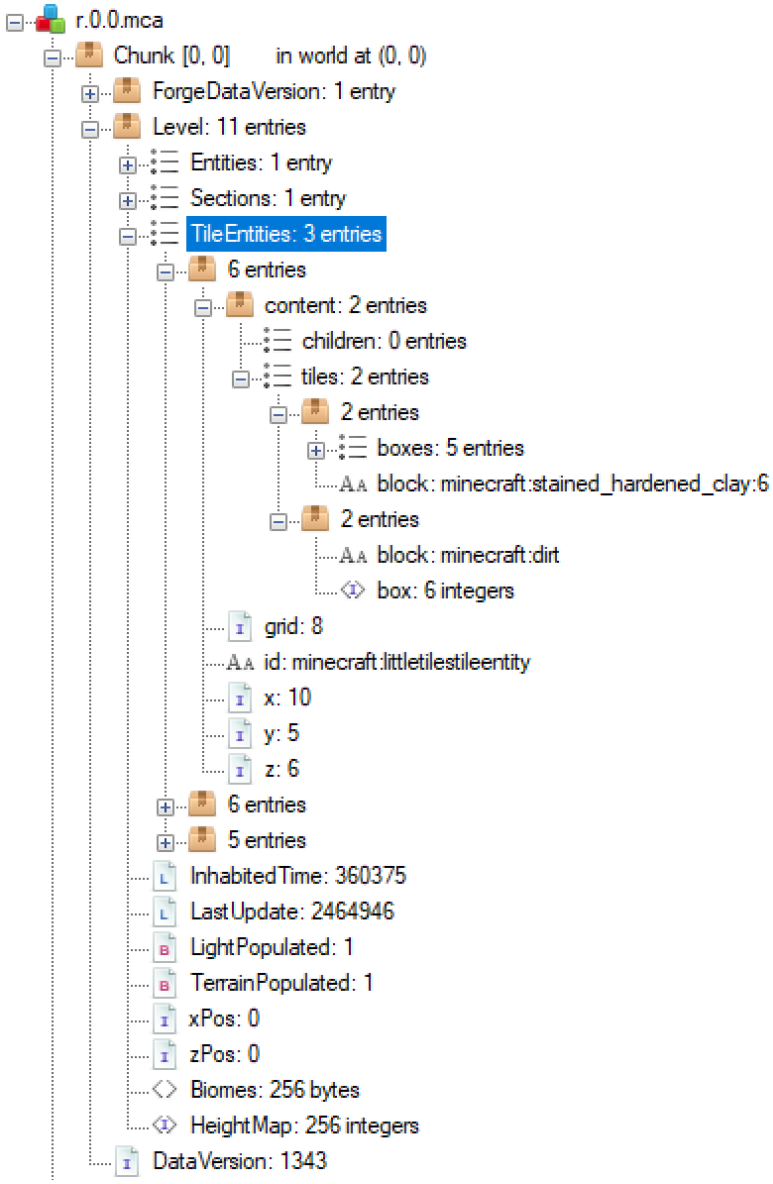

图 14 NBT 结构示例

[14] NBT Explorer：https://github.com/jaquadro/NBTExplorer/releases

2.4. Tiles 的偏移

2.4.1. 角的偏移

上文提到 Tile 有八个顶点，这八个顶点都可以进行偏移以形成更复杂的图形结构。在没有偏移的情况下，boxes 中的数组只有 6 个元素。

如果存在偏移，会在数组的最后添加一个数字表示角偏移状态，后续再紧跟偏移量。

$$boxes: [x_1, y_1, z_1, x_2, y_2, z_2, \text{angle\_state}, \text{offset\_value}]$$

Sign bit always negative (1)		Flipped						Vertex																							
								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

图 15 `angle_state` *位图（左侧为高位，右侧为低位）*

角偏移状态使用 32 位有符号整数，*图 15* 列出了每一个位的作用，其中最高位（31 位）永远为 1，该位也是符号位，这代表这个数永远为负数。

其中 Vertex 区域为我们这一节所关心的区域，它分为了八个区域代表每一个角，即矩形的八个顶点。每一个角有三个比特位每个比特位代表了三个轴向分别是否进行了偏移，如果是 1 则代表进行了偏移。

我们可以看到*图 15* 的 EUS 角和 EUN 角的 y 轴都进行了偏移，所以我们会看到其对应的比特位置为了 1。

当有 grid = 4，两点坐标为 $(1,1,1)_1$ $(3,3,3)_2$ 的立体矩形，且对 EUS 和 EUN 的 y 轴都向下偏移 2 个单位的几何结构如 *图 16* 所示。

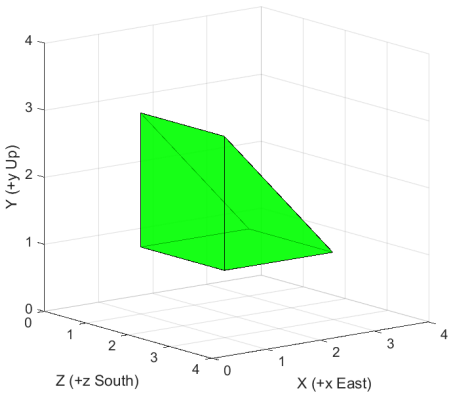

图 16 角偏移后的几何结构

2.4.2. 偏移量的表示

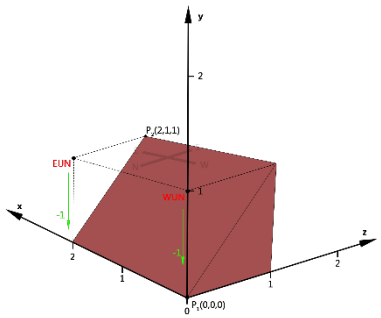

图 17 角偏移量示例

上一节我们说明了如何标记某个角是否有偏移，在此基础上我们还需要表示偏移量分别是多少。

在 boxes 数组中我们已经有了两点坐标、角状态，最后的 offset_value 表示了每个角的偏移量。offset_value 可以为多个，但比较特殊。

例如图 17 的示例，其中的 EUN 和 WUN 都沿 y 轴向下偏移了 1 格，即-1。

我们需要将其表示为 16 位的二进制反码表示法，即分别都为：$(1111\ 1111\ 1111\ 1111)_2$。

由于 NBT 标签只支持 32 位的整数，所以我们定义将他的低 16 位为第一个偏移量，高 16 位为第二个偏移量。

```
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（补码）表示为 11111111,11111111,11111111,11111111 (32bit)
 EUN y_offset （第一个偏移）来自高16位，即 11111111,11111111
 WUN y_offset （第二个偏移）来自低16位，即 11111111,11111111
 grid 代表该方块内的小方块精度是 2
```

图 17 由于启用了 WUN 与 EUN 对于 y 轴的偏移，所以角状态对应的比特位置为 1，由于二进制的性质，偏移的角应从右往左数。

第一个为 EUN 的 y 轴、第二个为 WUN 的 y 轴。

所以在角状态后紧接着的偏移量数据需要一个 32 位有符号整数来存储这两个偏移量，其中高 16 位是第一个（EUN 的 y 轴），低 16 位是第二个（WUN 的 y 轴）。

如果偏移有 3 个或更多，则继续向后追加 32 位有符号整数，并将其拆分为两个 16 位分别表示两个偏移量。

2.4.3. 特殊的偏移量

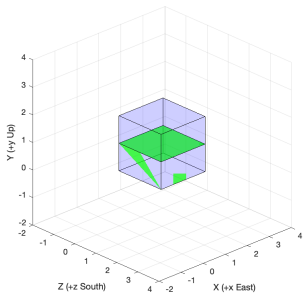

图 18 偏移前

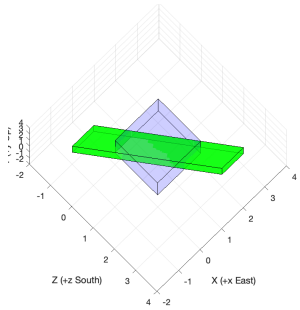

图 19 偏移后

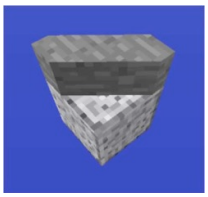

图 20 偏移后游戏表示

如 *图 18* 所示，这是一个 grid 为 2 在偏移前的一个几何结构。当我们对其应用偏移，进而导致它超出原先的几何结构边界，如*图 19* 所示（蓝色部分为方块边界）。

该图形会跟偏移前的几何结构做交集运算，仅保留在*图 18* 所示的几何图形的交集部分，所以实际的几何结构如*图 20* 所示。

2.5. Flipped

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

图 22 翻转位位说明

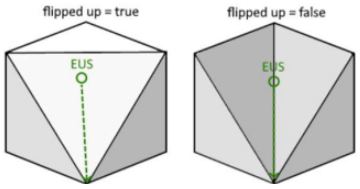

图 21 翻转示例

如*图 21*，我们只偏移了 EUS 角，但为了维持结构的完整性需要添加一条辅助线。但辅助线有两种生成方式，这两种不同的生成方式由 Fliped 表示，在位说明中（*图 22*），E、W、S、N、U、D 各自分别代表立体矩形的八个面，对应的二进制位表示辅助线的生成方式，如*图 21* 所示。