# Proland 文档 - 地形插件

## 1. 简介

Proland 地形插件提供了一些专门的类来生产地形高程数据，以及通用栅格数据（可以是任何表现形式，如反射率，土地覆盖分类，法线贴图，地平线图，环境闭塞地图等）

> 该文档中，producer 意为编程中的处理类，生成类，将统一翻译为 类

## 2. 地形插件类

本节介绍了 Proland 地形插件提供的预定义类。其中一些是地形高程数据专用的，其他的则被设计用于普通栅格数据（可以是任何表现形式，如反射率，土地覆盖分类，法线贴图，地平线图，环境闭塞地图等）

### 2.1. 残差类

生成高程**块**的一种直截了当的方法是从硬盘中载入预计算的块。这种方案通过把块数据看作高程，使用正交类来实施。这种方案下每个块可以被独立处理。一个问题是在生成比预计算块的精度更高的高程块的时候，如果使用邻域或线性插值去生成新的海拔高度，结果产生的的地形将看起来很不真实。

> tile，意为一块，一片，一景，该文档中应指程序处理的数据对象，为n*n矩阵形式，将统一翻译为 块

这也是为什么Proland也建议另一种方案，基于"多分辨率细分表面"的方法：除去根以外，在每个四叉树水平上通过对父块的**上采样**来计算高程块，同时加上高程残差。之后高程块自残差块中生成。这些残差块可以预计算并存储在硬盘上。它们也可以过程化：随着四叉树水平的降低，随机残差的振幅也降低，使用这些随机残差并通过上采样和叠加的过程，就可以获得一个分形地形。这个方法的一个优点是，通过仔细选择上采样的方案，在多次上采样操作后，有限的表面会被平滑（$C^\infty$）。因此地形可以在比预计算的残差更高的分辨率尺度上生成，并且仍然是平滑的，如有缺失的尺度则使用空残差（或者在小尺度上使用程序化噪声增加分形细节）。另一个优点则是比起原始高程，残差可以更好的被压缩存储。

> upsampling, 在该文档中指对于一个四叉树来说，当前树结点是从父结点采样而来，将统一翻译为 上采样

下图描述了高程块和残差块之间的关系：

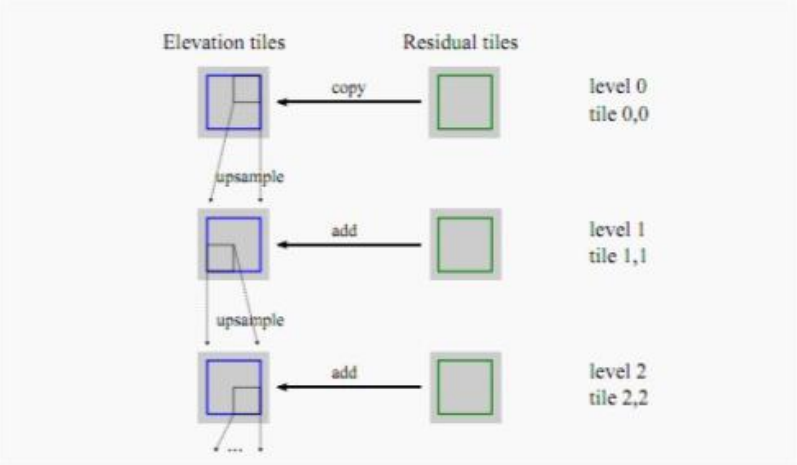


**图1：高程块和残差块的关系**

注意，高程和残差块都有边界（图中蓝色和绿色方框外面的灰色部分）。这个边界等于2个像素宽。根高程块是残差块的拷贝。其他高程块是通过对其父块的四叉树之一的上采样获得，并且叠加相应的残差块得到结果。每个块的内部（不含边界）是对其父块内部的四叉树之一做上采样获得。

上采样滤波器被用于生成高程块，也可以用于（预）计算残差块。 实际上，在上采样后获得给定地形所需的残差，显然是由上采用滤波器决定的。该滤波器定义见下图。其能确保得到平滑的有限表面。

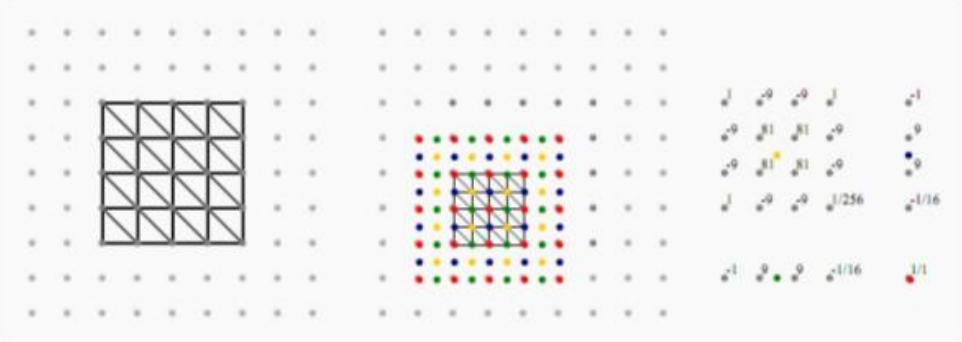

**图2：用于生成高程块的上采样滤波器**

上图示例假设高程块内部为 5*5 的样点，与边界构成 9*9 的样点。每个样点都假定作为一个 4*4 个四边形的格网网眼的顶点（如左图）。一般情况下，对一个 n*n 个四边形的格网网眼来说，高程块必须有 (n+5)*(n+5) 个样点，包含边界，n为整数。左图中，高程块的左下角由 3*3 个内部样点组成，被上采样为 5*5 的新的内部样点（含边界为 9*9 个），并组成更好的 4*4 个四边形的网眼，在中图中以彩色点表示。这些新样点是通过对父样点加权平均计算得来，根据右图中所示，新样点的位置决定了加权算法：

- 红色样点和其同位置的父样点完全相等（上采样滤波器是一种插值滤波，而不是近似滤波器）
- 绿色样点等于其同一水平位置上四个邻近父样点的加权平均值，权重分别为 -1/16，9/16，9/16，-1/16
- 蓝色样点定义与绿色一样，使用的是垂直位置上的四个邻近父样点
- 黄色样点定义为与它邻近的16个父样点的加权平均，权重定义为右图中权重的张量积，如1/256，-9/256 等

**注意：**

- 计算上采样的内部样点时，我们需要在父块中有1个像素的边界。为了实施第二次上采样，我们需要在上一步的采样块中也有1个像素的边界。所以在父块中，需要2个像素的边界。所幸所需要的边界不会无限制的增长。而实际上2个像素宽的边界就足以递归应用任何数量的上采样步骤。

**proland::ResidualProducer** 类可以将硬盘上的预计算残差块载入CPU寄存器。该残差块必须存储于包含"金字塔"块的文件中，每个金字塔一个文件（这些文件可以用 proland::preprocessDem 和 proland::preprocessSpherialDem 类生成 - 具体请看"preprocess"示例）：

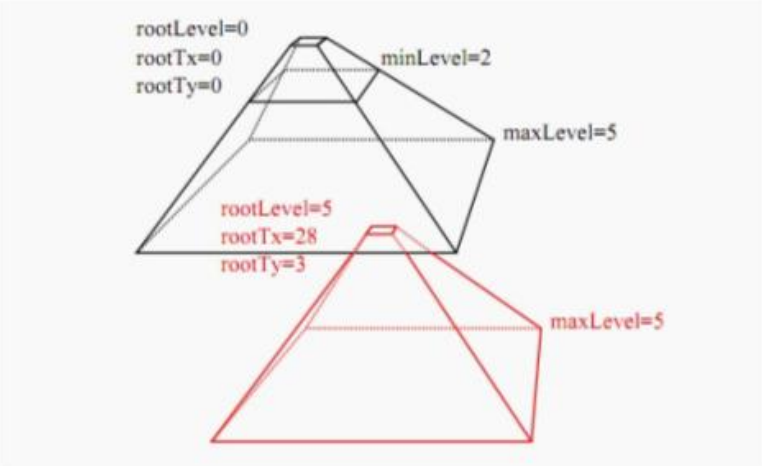


**图3：文件中残差块的存储**

至少要有一个文件，包含从根到最大四叉树水平的所有四叉树的残差块。其他文件则可以用来在一些地方提供更高的分辨率。这些文件也包含四叉树金字塔，但不是从根开始，而是从其他位置的块开始。在上图的示例中，主文件包含从上至下到第五级（包括第五级）所有的块。附加文件则包含了由块 (5, 28, 3) 开始，向下直到第 5+5=10 级的该块的所有子块。

每个块必须服从以下格式。文件应以一个由6个32位整型和一个32位浮点型数据组成的头文件开始：

- minLevel：如下
- maxLevel：该文件中数据块的最大级别，与起始块有关。

- tileSize：残差块的大小，以像素为单位，包含边界，最小值是5
- rootLevel：绝对四叉树中金字塔起始块的级别
- rootTx：绝对四叉树中起始块的tx坐标
- rootTy：绝对四叉树中起始块的ty坐标
- scale：该文件中残差值的尺度因子

minLevel 必须为 0，如果 rootLevel 不为 0，否则，它表示 tileSize 大小的金字塔的第一个块的级别。该块在第 minLevel-1, minLevel-2, …, 0级的大小为 tileSize/2, tileSize/4, …, tileSize/$2^{minLevel}$。尺度因子用于将整型的残差转换为在一些长度单位（如米）上的高度差。尺度为 1 则在高度上精度为 1 米。尺度为 0.1 则精度为 0.1 米，以此类推。

头文件后必须跟随 2.ntile 个 32 位的偏移量数组 offsets。（ ntile 等于金字塔中块的数量，即 minLevel+($2^{max(maxLevel-minLevel,0)\2+2}$-1)/3 。这些偏移量表示从该数组结尾开始，块数据在文件中剩余部分的位置。相对于起始块 (rootLevel, rootTx, rootTy) 来说，一个块的坐标 (level, tx, ty)，必须存储在 offsets[2*i*] 和 offsets[2*i*+1] 之间，其中 *i* 是块的索引， *i*=minLevel+tx+ty\$2^l$+($2^{2.l}$-1)/3, *l*=max(level-minLevel,0) (或 *i*=level 如果 level< minLevel)。在这2个偏移量之间，块必须以单一条带（single strip），每个像素 16 位，存储为TIFF格式。

**注意：**

- 不同坐标的块可以有相同的偏移量。这对于存储大量完全相同的块时非常有用（比如“空”块）

### 2.2.1. 残差类资源

残差类以以下格式（见“terrain4”具体实例）在 Ork 资源框架下载入：

```
<residualProducer name="myResidualProducer"
    cache="myCache" file="..." delta="0" scale="1"/>
```

cache 属性必须是缓存资源块的名字（其块的存储**必须**是 cpuFloatTileStorage，块大小为 tileSize+5 ）。 file 属性必须是包含残差块的文件的名字。选项 scale 属性可以用来在一些尺度水平上对残差值做拉伸。其默认值为 1，则不对残差做改变。最后，选项增量 delta 属性则被用来选择一个级别作为根从而代替“真实”的根。实际上，在残差生成过程中，getTile(level, tx, ty)函数从残差文件中载入了坐标为 (level+delta, tx, ty) 的块。因此 getTile(0, 0, 0) 载入了块 (delta, 0, 0) 。 delta 必须在 0（默认值）到 minLevel 之间。

**说明：**

- 具体而言，如果 delta 不为空， getTile(0, 0, 0) 返回的是应用于 0 级到 delta 级的块的上采样和叠加过程的结果。因此返回的块可以用作高程处理中的“根”块（见下节）。如果 minLevel 不为空，第一个残差块则和其他块大小不同。但是它们都返回一个自左下角起， (tileSize+5)(tileSize+5) 的数组：

**图4：残差块（红色部分，深灰色为边界）在 0, 1, 2, ... 级别的存储，假设 minLevel=2，tileSize=8。所有块都存储在 13*13 数组中（浅灰色部分）**

残差类可以使用存储在多个块文件中的多个块金字塔。做法如下：

```
<residualProducer name="myResidualProducer" cache="myCache" file="main">
    <residualProducer cache="myCache" file="region1"/>
    <residualProducer cache="myCache" file="region2">
        <residualProducer cache="myCache" file="region2-1"/>
        ...
    </residualProducer>
    ...
    <residualProducer cache="myCache" file="regionN"/>
</residualProducer>
```

以上处理对四叉树的第一级使用 main 文件，reginal1, reginal2, ..., reginalN 文件来为一些区域增加更高的分辨率。这个方案是可递归的，例如，reginal2-1 在 reginal2 的子区域增加了更高的分辨率。

## 2.2. 高程类

**proland::ElevationProducer** 类从残差类生成的残差块中生成地形高程。如前描述，该GPU类上采样父高程块并叠加到其残差块上，使用GPU着色器。此类可以具有层，以便用来修改前一步所生成的高程。

### 2.2.1. 输入数据

**proland::ElevationProducer** 有多种输入。特别是它在有或没有残差类的情况下都能使用。

- 没有残差类时，伪随机决定残差将自动生成。这时用户必须为每个四叉树水平指定噪声振幅。这种模式可以用来生成随机分形地形。 “terrain1”, “terrain2”, “terrain3” 实例对该选项做出了说明。
- 有残差类时，在比残差类提供的更高的分辨率上生成细节时，有两个选项可用。默认情况下，结果为平滑表面时不生成细节。另一个选项是通过为没有残差块的四叉树水平指定噪声振幅来叠加分形细节。”terrain4” 实例对第一个选项做出了说明。

两种情况中，用户指定的"上采样和叠加"着色器，在相同的输入条件下也能得到不同的结果。例如一个基础着色器可以在地形的每个点上简单的使用同样的噪声振幅。一个较复杂的着色器可以基于局部海拔，坡度或地形的曲率调节振幅。第一种情况里，所有地方都生成统一的分形细节。第二种情况里，噪声可以自适应局部地形 - 例如在山区比在平原生成更多的噪声（这两种情况在 "terrain1" 和 "terrain2" 实例的 "upsampleShader" 着色器中有描述）。

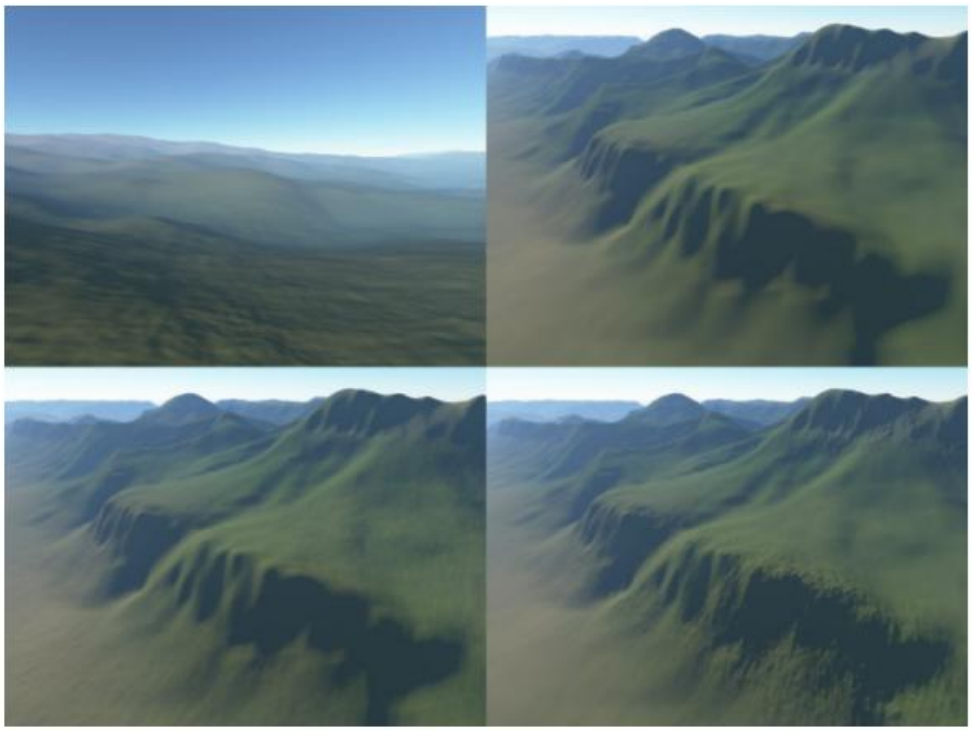

**图5：左上：随机残差生成分形地形。右上：预计算残差，没有叠加细节。下方：有均匀噪声（左）和没有均匀噪声（右）的同一地形**

### 2.2.2. 输出数据

生成的高程数据每个像元有三个分量，记作 (zf, zc, zm) - 如果没有使用层，则第三分量不用存储。第一分量是高程，由以上介绍的上采样和叠加过程生成。第二分量是粗高程，比如同一个点的父块的高程（此目的是为了表现渲染时 zf 和 zc 之间的插值，以避免四边形在突然划分时产生的爆音 - 见 sec-quadblend）。第三分量是校正高程。默认等于 zf ，但是层可以修改此值。

高程 zf 是使用前节介绍的上采样滤波从父块的未校正高程计算而来。但是粗高程 zc 是使用线性插值，从父块的校正高程 zm 计算得来。实际上 zc 是为了给出由平面三角组成的父网眼的高程（因此做线性插值）。因此我们需要知道哪些网眼会被用来渲染地形四边形。实际上高程块可以用多种大小的网眼渲染，高程块的大小是网眼的大小的倍数（注意：网眼对角线**必须**是从"西北"到"东南"）：

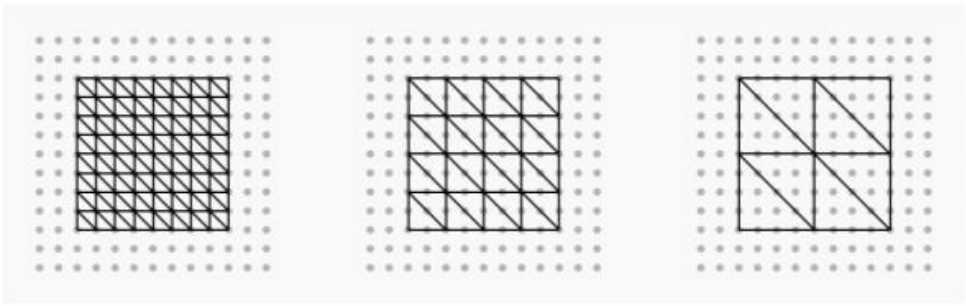

**图6：一个 9*9 的高程块（带边界是 13*13 ）可以用 8*8 , 4*4, 2*2, 或 1*1 的格网渲染**

一旦知道了网眼大小，我们可以找出哪些父样点被用于插值计算粗高程 zc ：

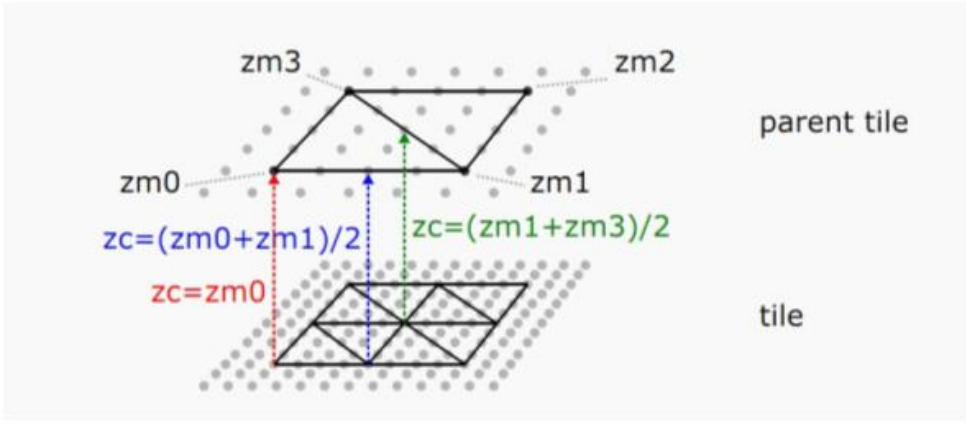


**图7：粗高程是由校正的父高程线性插值而来。使用的父样点由网眼大小决定。**

**注意：**

- 如果你想用"粗"网眼渲染地形，但用更高分辨率的法线地图为地形贴图，那么可以使用比高程块更小的网眼。例如，你可以使用 (192+5)(192+5) 的高程块，从中计算 (192+1)(192+1) 的法线。然后，对其使用 24*24 个四边形的格网来渲染， 但使用8倍于法线分辨率的 (192+1)(192+1) 的法线地图来贴图（以上图像用这些设置生成）。

### 2.2.3. 高程类资源

高程类以以下格式在 Ork 资源框架下载入：

```
<elevationProducer name="myElevationProducer"
    cache="myCache" residuals="myResidualProducer"
    upsampleProg="upsampleShader;" blendProg="blendShader;"
    noise="100,50,25,12,6,3,2,1" face="0"
    gridSize="24" flip="false"/>
```

该高程块由 cache 属性指定的缓存资源块生成。该缓存必须有一个关联的 gpuTileStorage。存储单元中块的大小定义了即将生成的高程块的大小（包含边界）。 residuals 残差属性指定了残差类的名字（不需要一定是 **proland::ResidualProducer**；任何在相同格式上生成浮点型数值的CPU类都可以使用）。此属性是可选项。如果没有给出， noise 属性则必须被定义（见下方）。

**注意：**

- 存储高程块的 gpuTileStorage 必须每个通道都为 16 或 32 位浮点型，但可以少于4个通道。例如，如果你没有使用地形变形，那么就不需要存储第四分量 w 。类似的，如果你没有使用层来校正地形高程，第三分量也不需要存储。
- 高程和残差块可以具有不同的大小。具体来说，残差块的大小必须是高程块大小的倍数。例如，如果残差块大小是 (192+5)(192+5) ，高程块的大小就可以是 (192+5)(192+5) ，(96+5)(96+5) ， (48+5)(48+5) ， (24+5)(24+5) 等（高程类会选择正确的"子块"，如果必要，残差块会直接用来生成高程块）。

upsampleProg 和 blendProg 属性指定了用来执行"上采样和叠加"操作的着色器，并且将层混入高程块。他们的默认值是 upsampleShader; 和 blendShader; 。如果没有使用层，blendProg 程序就不是必要的。Proland 示例解释了这些着色器如何执行（特别是在 "terrain1", "terrain2", 和 "graph1" 实例中）。

选项 noise 属性必须指定噪声振幅，每个四叉树级别一个值。这些振幅被用于生成分形地形或者给现有地形加入分形细节。选项 face 属性指定了类对应立方体的哪个面，如果6个类被用来建模一个球体（见 sec-deform）。面的数量必须是 1 到 6 （ 1 是北极面， 6 是南极面 - "terrain2" 实例演示了面是如何组织的）。如果类名字以 1 到 6 的数字结尾，那么面属性就不需要：它会从类名字中提取。此面标识符被用来产生立方体的面之间的接缝噪声。

gridSize 属性必须指定为被用来渲染每个地形四边形的网眼的边的长短。这个长短大小需要用来正确计算粗高程 zc （见上文）。

最后，flip 属性表示，为了减少地形轮廓的几何失真，以当前网眼的每条"边"的四个角的高程为基准，是否需要地形渲染器动态的翻转地形格网的对角线：

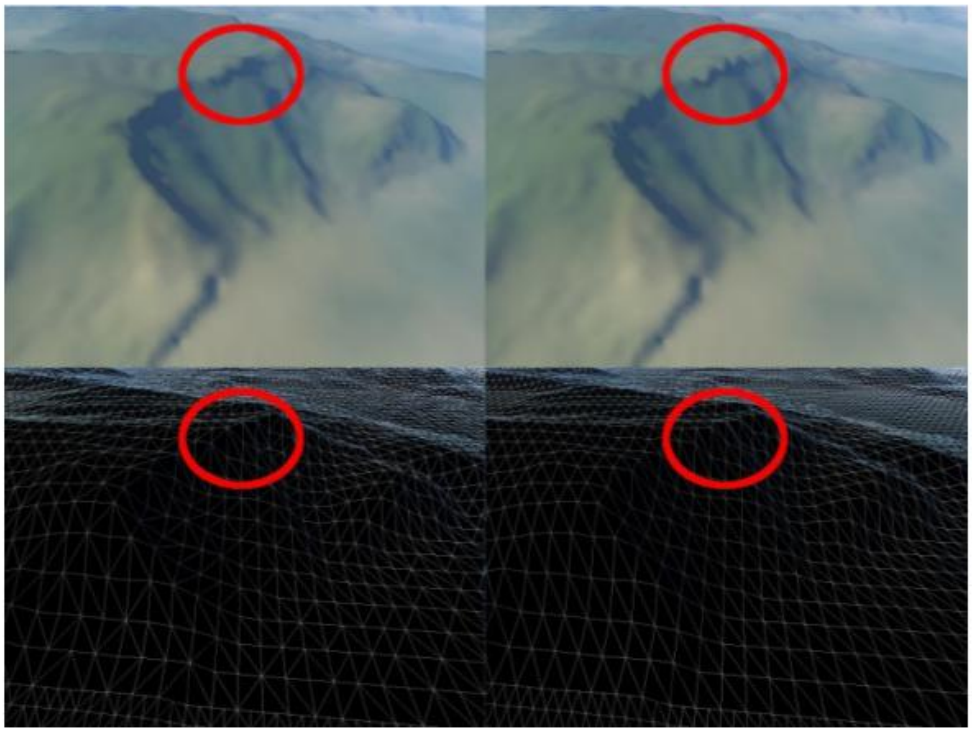

**图8：对比基本渲染（右图），动态翻转对角线渲染（左图）的几何失真更低**

在 upsampleProg 着色器中设置该选项，即使动态翻转，也仍然能计算正确的粗高程。“terrain4” 实例解释了该选项如何使用（见 helloworld.xml 文件， upsampleShader.glsl 和 terrainShader.glsl 代码，以及 quad.mesh 文件中的不同）。

## 2.3. 法线类

**proland::NormalProducer** 类是由高程类生成的高程块地形生成法线。该GPU类使用GPU的着色器计算有限差的法线。

### 2.3.1. 输入数据

**proland::NormalProducer** 类的输入数据是高程类生成的高程块。该高程类不需要一定是 **proland::ElevationProducer** 类：任何在相同格式（每个样点有 3 到 4 个分量，2 个像素的边界等）上能够生成浮点型高程数值的GPU类都可以使用。法线类用校正的高程 zm 计算法线（见高程类）。

### 2.3.2. 输出数据

生成的法线块每个像元有四个分量，记作 (nx, ny, ncx, ncy) ，没有边界（可以只存储前 2 个值）。前一个分量 (nx, ny) 给出当前点的法线的 x, y 分量（垂直分量 z 可以由 1-nx*nx-ny*ny 的平方根得到）。后一个分量 (ncx, ncy) 给出当前点的粗法线的 x, y 分量，比如，插值得到的当前点的父网眼的法线（假设法线是从顶点着色器获取并在每个格网三角面上线性插值 - 但也

可能使用生成的法线作为法线地图，比如，在片段着色器中获取）。目的是在渲染时，在法线和粗法线之间做插值，避免四边形划分时产生的爆音 - 见 sec-quadblend。就像在高程类中一样，用来渲染每个地形四边形的格网大小必须是已知的由粗法线正确计算而来。

**注意：**

- 法线是由位于块的中心的基准面的切线框架计算得来的（比如，单个切线框架用于块上所有的顶点）。默认基准面是水平的。如果使用了地形变形，则它可以是球面，柱面等。

### 2.3.3. 法线类资源

法线类以以下格式在 Ork 资源框架下载入：

```
<normalProducer name="myNormalProducer"
    cache="myCache" elevations="myElevationProducer"
    normalProg="normalShader;"
    gridSize="24" deform="none"/>
```

该法线块由 cache 属性指定的缓存资源块生成。该缓存必须有一个关联的 gpuTileStorage 。存储单元中块的大小定义了即将生成的法线块的大小（注意不含边界）。 elevations 高程属性指定了用于计算法线的高程类的名字（不需要一定是 **proland::ElevationProducer** ）

**注意：**

- gpuTileStorage 可以使用整型或浮点型存储法线块，每个像元使用 2 到 4 个通道。位于 -1 到 1 之间的法线坐标存储为浮点型。首先使用 0..1 的整型坐标来绘制（使用 (x+1)/2, (y+1)/2 ）。如果使用 2 个通道，则粗法线将被抛弃。总之，可用的格式有RGBA8，RGBA16F，RGBA32F，IA8，IA16F，IA32F。高程块和法线块的大小也必须相同。例如，如果高程块为 (192+5)(192+5) ，则法线块必须为 (192+1)(192+1) 。

选项 normal 属性指定了从高程计算法线的着色器。默认为 normalShader; 。 Proland 实例展示了这些着色器如何被实施（特别是在 “terrain1” 和 “terrain2” 实例中）。

gridSize 属性必须指定为用于渲染每个地形四边形的格网的边长。该边长大小用于正确计算粗法线 nc （见上文）。

最后 deform 属性必须指定为应用于地形的地形变形。目前仅提供 “none” 和 “sphere” 可选，分别表示地形不变形，或者变为球形（渲染球体）。该值控制了用于计算法线的切线框架（见上文）。

## 2.4. 正交CPU类

**proland::OrthoCPUProducer** 类用于将存储于硬盘的预计算块载入CPU寄存器。正交类不处理块的内容，内容可以是任何形式：反射率，土地覆盖类型，法线贴图，地平线图，环境闭塞地图等。和残差块一样，块必须存储在含有"金字塔"的文件中，每个金字塔一个文件（这些文件可以由 proland::preprocessOrtho 和 proland::preprocessSpherialOrtho 类生成 - 见 "preprocess" 实例）：

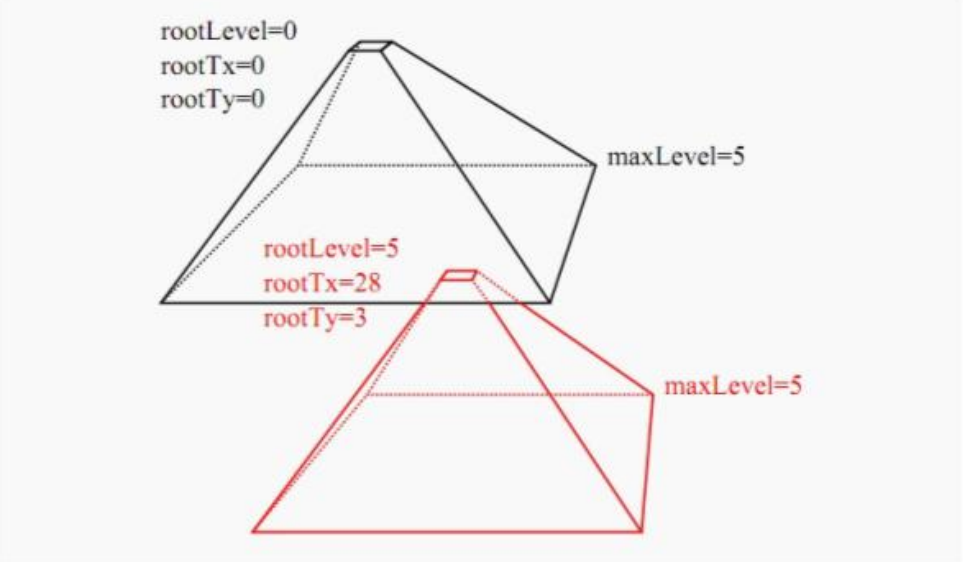


**图9：正交块在文件中的存储**

至少要有一个文件，包含从根到最大四叉树水平的所有四叉树的正交块。其他文件则被用于在一些地方提供更高的分辨率。这些文件也包含四叉树金字塔，但不是从根开始，而是从其他块开始。在上图的示例中，主文件包含从上至下到第五级（包括第五级）所有的块。附加文件则包含了由块 (5, 28, 3) 开始，向下直到第 5+5=10 级的该块的所有子块。

每个块必须服从以下格式。文件应由 7 个 32 位整型数据组成的头文件开始：

- maxLevel：该文件中数据块的最大级别，与起始块有关。
- tileSize：正交块的大小，以像素为单位，不含边界
- channels：每个像元分量的数量，位于 1 到 4 之间
- rootLevel：绝对四叉树中金字塔起始块的级别
- rootTx：绝对四叉树中起始块的tx坐标
- rootTy：绝对四叉树中起始块的ty坐标
- flags：表示块是否有边界和是否被压缩（见下文）

flags 字段是字节集。如果bit 0被设置，则块被存储为 DXT 格式（否则被存储为未压缩 TIFF，deflate，jpeg 等格式）。如果 bit 1 被设置则块具有 2 个像素的边界（否则没有边界）。

头文件后必须跟随 2.ntile 个 64 位的偏移量数组 offsets。（ ntile 等于金字塔中块的数量，即 $(2^{maxLevel*2+2}-1)/3$ 。这些偏移量表示从该数组结尾开始，块数据在文件中剩余部分的位置。相对于起始块 (rootLevel, rootTx, rootTy) 来说，一个块的坐标 (level, tx, ty)，必须存储在 offsets[2*i*] 和 offsets[2*i*+1] 之间，其中 *i* 是块的索引， $i=tx+ty*2^{level}+(2^{2.level}-1)/3$ 。在这 2 个偏移量之间，块必须以存储为 TIFF 或 DXT 格式（如 flags 所示 - 如果是 TIFF 格式，则必须存储为单一条带）。

### 2.4.1. 正交CPU类资源

正交CPU类以以下格式在 Ork 资源框架下载入：

```
<orthoCpuProducer name="myOrthoCpuProducer" cache="myCache" file="..."/>
```

cache 属性必须是缓存资源块的名字。其块存储必须是 cpuByteTileStorage ，大小为 tileSize ，如果有边界则为 tileSize+4 。块存储中的通道数量必须等于 channels 属性。 file 属性必须是包含上述格式的正交块的文件的名字。

正交类可以使用存储在多个块文件中的多个块金字塔。做法如下：

```
<orthoCpuProducer name="myOrthoCpuProducer" cache="myCache"
file="main">
    <orthoCpuProducer cache="myCache" file="region1"/>
    <orthoCpuProducer cache="myCache" file="region2">
        <orthoCpuProducer cache="myCache" file="region2-1"/>
        ...
    </orthoCpuProducer>
    ...
    <orthoCpuProducer cache="myCache" file="regionN"/>
</orthoCpuProducer>
```

以上处理对四叉树的第一级使用 main 文件，reginal1, reginal2, …, reginalN 文件来为一些区域增加更高的分辨率。这个方案是可递归的，例如，reginal2-1 在 reginal2 的子区域增加了更高的分辨率。“graph1” 实例说明了正交CPU类是如何使用的。

## 2.5. 正交GPU类

**proland::OrthoGPUProducer** 类把已经存储在主寄存器的块载入到 GPU 显存（如正交 CPU 类生成的块）。默认该块是简单的由 CPU 寄存器拷贝至 GPU 显存。但该类也允许有层，可以修改被拷贝到 GPU 之后的内容（使用 GPU 着色器）。

### 2.5.1. 预定义层

- EmptyOrthoLayer

**proland::EmptyOrthoLayer** 类继承于 proland::TileLayer ，其简单将块填充为常量颜色。用作为其他层提供背景色时的第一个层。

**proland::EmptyOrthoLayer** 类以以下格式在 Ork 资源框架下载入：

```
<emptyOrthoLayer name="background" color="255,255,255"/>
```

“river1” 实例说明了该图层如何被使用。

- TextureLayer

**proland::TextureLayer** 类继承于 proland::TileLayer ，其绘制了一个正交纹理顶部的四边形，用作输入其他正交 GPU 类生成的块（实际上任何使用 GPU 块存储的类都可以），并且在外部块混入生成的块之前转换它。

该图层的典型用法是添加层到正交类（见下文）。实际上一个 **proland::OrthoProducer** 不能拥有层本身。变通方案是，除此正交类之外，一个带有 **proland::TextureLayer** 类的正交 GPU 类用于输入正交类。这使得为正交 GPU 类添加图层成为可能。此方法在 “exercies2” 实例中有展示。

**proland::TextureLayer** 类以以下格式在 Ork 资源框架下载入：

```
<textureLayer name="background"
    producer="backgroundOrthoGpu1"
    renderProg="copyShader;"
    tileSamplerName="sourceSampler"
    equation="ADD" equationAlpha="ADD"
    destinationFunction="ZERO" sourceFunction="ONE"
    destinationFunctionAlpha="ZERO" sourceFunctionAlpha="ONE"/>
```

producer 属性必须含有用 GPUTileStorage 的块类的名字。它指定了将要被组合到的该图层所属的类所生成的块中的外部块。renderProg 属性指定了在被混入生成的块之前，用于转换 producer 的外部块的程序。tileSampleName 属性指定了用于接入外部块的 renderProg 的统一名称（见 sec-samplertileglsl ）。最后一个属性指定混合模式和混合转换后外部块的方程。

### 2.5.2. 正交GPU类资源

正交GPU类以以下格式在 Ork 资源框架下载入：

```
<orthoGpuProducer name="myOrthoGpuProducer"
    cache="myCache" backgroundCache="myCache"
    ortho="myOrthoCpuProducer"/>
```

cache 属性必须是块缓存资源的名字（其块存储必须是 gpuTileStorage ）。

选项 backgroundCache 属性必须是块缓存资源的名字（其块存储必须是 gpuTileStorage ），其可能和 cache 相同。该属性仅当层被使用时才是必要的（见 “graph1” 实例）。

选项 ortho 属性必须是生成CPU块的类的名字，该块由该 GPU 类拷贝到 GPU （不需要一定是 **proland::OrthoCPUProducer** ）。如果该类没有图层，则该属性是必要的。否则，它可以被忽略。此时块仅由图层生成（ “river1” 实例解释了该选项。 “graph1” 实例解释了第一个选项）。

**注意：**

CPU 和 GPU 块的大小必须相等。例如，如果 CPU 类生成了 192*192 大小的块，有 2 个像素边界，那么正交 GPU 类的 gpuTileStorage 必须使用 (192+2*2)*(192+2*2)=196*196 大小的 GPU 块。然而 gpuTileStorage 在每个像元上可以使用不同数量的通道，并且通道的格式是自由的（ 8 bits ， 16 bits 浮点型， 32 bits 浮点型等）。

## 2.6. 正交类

**proland::OrthoProducer** 类由残差类（不是 **proland::ResidualProducer** 类，但是，比如含有预计算颜色残差的 **proland::OrthoCPUProducer** 类）生成的残差块来生成地形纹理。

### 2.6.1. 输入数据

**proland::OrthoProducer** 类有多种输入。尤其是它可以使用也可以不使用残差类：

- 没有残差类时，伪随机决定残差将自动生成。这时用户必须为每个四叉树水平指定噪声振幅。这种模式可以用来生成随机分形纹理。 “terrain3” 实例对该选项做出了说明。
- 有残差类时，两个选项可以在比残差类提供的更高的分辨率上生成细节。默认情况下，结果为平滑纹理时不生成细节。另一个选项是通过为没有残差块的四叉树水平指定噪声振幅来叠加分形细节。

两种情况中，用户指定的“上采样和叠加”着色器，在相同的输入条件下也能得到不同的结果。如同 **proland::ElevationProducer** 。

### 2.6.2. 输出数据

正交类上采样父颜色块并叠加至其残差块上，如残差类中所描述，但是使用了不同的上采样滤波器。该滤波器如下定义（它更适应于颜色纹理，采样点位于每个纹理的中心，然而为高程纹理设计的滤波器，采样点为奇数，对应于网眼顶点）。

**图10：用于生成正交块的上采样滤波器**

图中示例假设纹理块有 4*4 个内部样点，连同边界为 8*8 。每个样点都位于纹理中心（纹理表示为图中黑色方框）。左图中颜色块的左下角由 2*2 的内部样点组成，上采样为 4*4 的内部样点（带边界为 8*8 ）后，并组成更好的 4*4 的内部纹理，在中图中以彩色点表示。新的样点由父样点的加权平均计算而来。如右图，规则取决于它们的位置（所有的权重都除以 16 ，之和为 1 ）。

相比于 **proland::OrthoGPUProducer** 类， **proland::OrthoProducer** 类的优点是在多次上采样操作后，有限纹理仍然是平滑的（$C^\infty$）。因此纹理能够通过为缺失的尺度使用空残差，在比预计算残差所描述的更高的分辨率上生成，并且仍然是平滑的。另外一个优点是该类支持实时编辑（见 sec-edit-ortho ）。不利的方面是该类比 **proland::OrthoGPUProducer** 类需要更多的 GPU 显存（为了更好的表现，一个块的所有祖先也都必须出现在 GPU 中）。

### 2.6.3. 预定义图层

一般来说， **proland::OrthoProducer** 类可以**不**含有层。实际上，任何层都可以修改上一步"上采样和叠加"步骤的结果，这些修改将会在下一级四叉树水平上被上采样，从而产生无用的结果（高程类可以含有层的原因是一个高程需要一个单一通道，而层修改一个独立的通道，不会被用于上采样；但是在纹理类中不行，因为一个颜色需要 3 到 4 个通道。然而，对于一个单色/黑白正交类，如果层修改独立通道，那么是可以使用的，使用方法和高程类中相同）。

该变通方案在 **proland::TextureLayer** 类中使用过（见上文）。

### 2.6.4. 正交类资源

正交类以以下格式在 Ork 资源框架下载入：

```
<orthoProducer name="myOrthoProducer"
    cache="myCache" residuals="myOrthoCpuProducer"
    scale="1" maxLevel="16" upsampleProg="upsampleOrthoShader;" hsv="true" cnoise="1
0,20,30" noise="200,180,160,140,120,100"
    face="0"/>
```

cache 属性必须是块缓存资源的名字（其块存储必须是 gpuTileStorage ）。 residuals 属性必须是生成该正交 GPU 类使用的 CPU 残差块的类的名字（不需要一定是 **proland::OrthoCPUProducer** ）。此属性是可选项。如果没有给出，则 noise 属性必须被定义（见下文）。

**注意：**

- CPU 块和 GPU 块的大小必须相等。例如，如果残差类生成的块大小是 192*192 ，有 2 个像素的边界，那么正交类的 gpuTileStorage 必须使用 (192+2*2)*(192+2*2)=196*196 大小的 GPU 块。然而 gpuTileStorage 在每个像元上可以使用不同数量的通道，并且通道的格式是自由的（ 8 bits ， 16 bits 浮点型， 32 bits 浮点型等）。

选项 scale 属性是应用于在叠加到上采样块之前的残差的尺度因子。maxLevel 属性用于为该类生成的块设置最大级别（默认无限制）。

选项 noise 属性必须指定为噪声振幅，每个四叉树级别有一个。这些振幅用于生成分型形纹理或者叠加分形细节到已有纹理上。这些噪声颜色通过 cnoise 属性指定，而 hsv 属性会指出该颜色是使用 HSV 空间还是 RGB 空间。

选项 face 属性指定了类对应立方体的哪个面，如果 6 个类被用来建模一个球体（见 sec-deform）。面的数量必须是 1 到 6 （ 1 是北极面， 6 是南极面 - “terrain2” 实例演示了面是如何组织的）。如果类名字以 1 到 6 的数字结尾，那么面属性就不需要：它会从类名字中提取。此面标识符被用来产生立方体的面之间的接缝噪声。

## 2.7. 一个示例

一个由存储在 DEM.dat 文件中的残差块和存储在 ORTHO.dat 文件中的纹理块描述的单一地形场景，可以被描述为如下的 Ork 档案文件：

```
<multithreadScheduler name="defaultScheduler" nthreads="3" fps="0"/>
<tileCache name="dzCache" scheduler="defaultScheduler">
    <cpuFloatTileStorage tileSize="197" channels="1" capacity="1024"/>
</tileCache>
<residualProducer name="dz" cache="dzCache" file="DEM.dat"/>
<tileCache name="zCache" scheduler="defaultScheduler">
    <gpuTileStorage tileSize="29" nTiles="1600"
        internalformat="RGBA32F" format="RGBA" type="FLOAT"
        min="NEAREST" mag="NEAREST"/>
</tileCache>
<elevationProducer name="z" cache="zCache" residuals="dzCache"/>
<tileCache name="nCache" scheduler="defaultScheduler">
    <gpuTileStorage tileSize="25" nTiles="1600"
        internalformat="RGBA32F" format="RGBA" type="FLOAT"
        min="NEAREST" mag="NEAREST"/>
</tileCache>
<normalProducer name="n" cache="nCache" elevations="z"/>
```

以上资源使用 DEM.dat 文件定义了一个残差类，块大小为 (192+5)*(192+5) 。在一个缓存块大小为 (24+5)*(24+5) 的高程类中使用，存储为 1600 个快照，而该高程类则在一个具有相同大小但没有边界的法线类中使用，大小为 (24+1)*(24+1) ，存储为 1600 个快照。

```
<tileCache name="rgbCpuCache" scheduler="defaultScheduler">
    <cpuByteTileStorage tileSize="196" channels="3" capacity="1024"/>
</tileCache>
<orthoCpuProducer name="rgbCpu" cache="rgbCpuCache" file="ORTHO.dat"/>
<tileCache name="rgbGpuCache" scheduler="defaultScheduler">
    <gpuTileStorage tileSize="196" nTiles="400"
        internalformat="RGB8" format="RGB" type="UNSIGNED_BYTE"
        min="LINEAR_MIPMAP_LINEAR" mag="LINEAR" minLOD="0" maxLOD="1"/>
</tileCache>
<orthoGpuProducer name="rgbGpu" cache="rgbGpuCache" ortho="rgbCpu"/>
```

以上资源使用 ORTHO.dat 文件定义了一个正交 CPU 类，块大小为 (192+4)*(192+4) 。在一个具有相同块大小的正交 GPU 类中使用，缓存在 400 个快照中。

```
<terrainNode name="myTerrain" size="50000" zmin="0" zmax="5000"
    splitFactor="2" maxLevel="16"/>
<node name="myTerrainNode" flags="object,dynamic">
    <bounds xmin="-50000" ymin="-50000" zmin="0"
        xmax="50000" ymax="50000" zmax="5000"/>
    <field id="terrain" value="myTerrain"/>
    <tileSamplerZ id="z" sampler="zSampler" producer="z"
        storeInvisible="false"/>
    <tileSampler id="n" sampler="nSampler" producer="n"
        storeInvisible="false"/>
    <tileSampler id="rgb" sampler="rgbSampler" producer="rgb"
        storeParent="false" storeInvisible="false"/>
    <mesh id="grid" value="grid25.mesh"/>
    <method id="update" value="updateTerrainMethod"/>
    <method id="draw" value="drawTerrainMethod"/>
    <shader id="material" value="terrainShader"/>
</node>
```

以上资源为一个大小为 100 km * 100 km 的地形定义了一个地形节点，高程为 0 到 5000 m，设置 2 为距离划分因子，划分地形四叉树到 16 级（包括 16 ）。为该地形定义了一个场景节点资源：它通过"地形"字段引用地形， 3 个纹理块取样器从地形着色器中获取高程，法线和正交纹理块。地形场景节点也指定了用于绘制地形四边形的格网，用来更新地形节点的方法，绘制的方法，以及绘制地形的着色器。

```
<node name="myScene">
    <node flags="camera">
        <uniformMatrix4f id="cameraToScreen" name="cameraToScreen"/>
        <shader id="material" value="cameraShader"/>
        <method id="draw" value="cameraMethod"/>
    </node>
    <node name="terrain" value="myTerrainNode"/>
</node>
```

最终，一个场景图形被定义，包含一个摄像头和先前的地形场景节点。 updateTerrainMethod

资源被定义如下：

```
<?xml version="1.0" ?>
<sequence>
    <updateTerrain name="this.terrain"/>
    <updateTileSamplers name="this.terrain"/>
</sequence>
```

换句话说它更新了地形四叉树 “this.terrain” 和场景节点所属的纹理块取样器的方法。在我们的例子中， “this.terrain” 引用了 “myTerrainNode” 的 “terrain” 字段，而其本身引用自 “myTerrain” 。并且纹理块取样器是这些 “myTerrainNode” 场景节点，比如， “zSampler”, “nSampler”, 和 “rgbSampler” 取样器。 drawTerrainMethod 定义如下：

```
<?xml version="1.0" ?>
<sequence>
    <setProgram>
        <shader name="this.material"/>
    </setProgram>
    <drawTerrain name="this.terrain" mesh="this.grid" culling="true"/>
</sequence>
```

为了首先设置程序使用 “this.material” 着色器（在我们的例子中引用自 “myTerrainNode.material” ，而其本身引用自 “terrainShader” 着色器），在绘制每个可见叶片四边形之前使用 “this.mesh” 格网（在我们的例子中引用自 “grid25.mesh” ）。

最后，为了获取当前四边形的高程，法线和正交块， terrainShader 着色器应该遵照以下形式：

```
#include "textureTile.glsl"
uniform samplerTile zSampler;
uniform samplerTile nSampler;
uniform samplerTile rgbSampler;
#ifdef _VERTEX_
layout(location=0) in vec4 vertex;
out vec2 uv;
void main() {
    ...
    uv = vertex.xy;
    vec4 z = textureTile(zSampler, uv);
    vec4 n = textureTile(nSampler, uv);
    ...
}
#endif
#ifdef _FRAGMENT_
in vec2 uv;
layout(location=0) out vec4 color;
void main() {
    ...
    vec4 rgba = textureTile(rgbSampler, uv);
    ...
}
#endif
```