Proland 文档 - 核心库

slot, 此处翻译为 快照

1 简介

Proland 是一个程序化景观渲染库。被设计用于实时渲染超大型景观，乃至整个星球。在该语义下，它不可能在 GPU 显存存储整个景观数据。数据必须为当前视野实时生成。可以从硬盘载入预计算数据实现，或用程序化方法生成。 Proland 的另一个目标是景观的实时编辑。这通过运行中景观数据的重生成实现，当视野移动时用同样的方法在运行中生成。

Proland 由核心库和一些扩展插件组成。核心库提供了处理框架，地形框架，和一个基本用户接口框架。

- 处理框架为所有数据处理类定义了一个通用接口，如 CPU 或 GPU 处理类，栅格数据或矢量数据处理类等。处理类可以使用其他处理类生成它们的数据然后用复杂方式组合（例如一个地形法线类可以通过高程类从高程生成中生成法线）。处理类框架也提供了一个一般缓存组件来存储生成的数据。这给临时粘性带来优点：感谢这些缓存，一个画面的数据处理类可以被接下来的画面所使用。
- 地形框架基于当前观测位置使用地形四叉树动态划分。它也提供了一个新的 GLSL 类型可以用于访问 GPU 上栅格数据的缓存，和一些方法来更新缓存（通过使用数据处理类）和绘制地形。它也为地图提供了变形使平地形变为其他形状，如球形（渲染球体）。
- 用户接口框架基于事件句柄。它为你能在 Proland 中显示的大场景的导航提供了基础。

Proland 插件基于此框架提供了一些预定义处理类：一些处理类是地形高程数据生成专用，一些设计于通用栅格数据（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），另一些生成矢量数据（也可以通过矢量数据到纹理的栅格化来生成栅格数据）。

Proland 有以下方式以 Ork 库为基础：

- 处理类框架使用 Ork 任务生成数据。因此景观数据可以并行生成，或者甚至在 Ork 计划任务框架预抓取特征之前。该框架也提供处理类任务之间的依赖。因此当一个数据由一个处理类编辑，所有直接或间接从其得到的数据，通过其他处理类，也都自动重计算
- 地形框架为着色器，格网等使用 Ork 渲染框架，并且扩展 Ork 场景图，用新方法更新 GPU 缓存和绘制地形
- 最后 Proland 为预定义处理类和地形组件，用新资源类型扩展 Ork 资源框架

以下章节解释了处理类框架，地形框架和用户接口框架：

2 处理类框架

处理类框架定义了景观数据如何生成，存储和缓存。使用这个框架可以定义一些处理类，每个处理类生成景观数据的一部分。例如有的生成地形高程，有的生成地形法线，有的生成地形反射率，有的生成河流数据，有的生成建筑模型等。每个处理类都可以使用硬盘上的数据，其他处理类的生成的数据，二者结合等作为输入参数。

处理类框架假设处理类生成的数据是被四叉树划分的。这意思是每个块，即每个四边形的数据，可以独立生成。一个块可以包含栅格数据，矢量数据，或任何其他数据。每个处理类都使用自己的四叉树组织这些数据，即，不同处理类的四叉树不需要有同样的特征（最大深度，块大小等）。然而，四边形和块总是使用相同的"坐标系统"识别的。实际上，一个四边形或块通过它们在四叉树中的 level （根是 0 ）和该层中的 tx, ty 坐标（tx 和 ty 位于 0 和 $2^{level}-1$ 之间，0, 0 是在下角）识别。这样 (level, tx, ty) 坐标称为逻辑坐标：

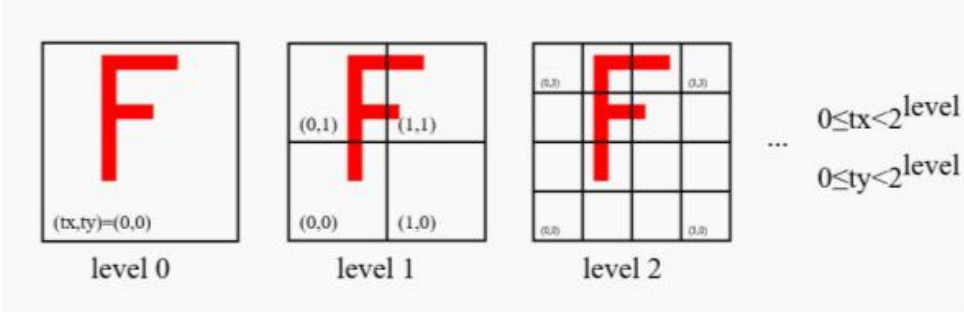

图1；逻辑四边形和块坐标

一个处理类的根块 (0,0,0) 在包含一个粗分辨率的相应于整个场景的数据。其他层次上的块包含该数据的一部分，但是分辨率更高（层次越高，分辨率越高）。处理类框架也使用物理坐标。这些坐标是一个固定参考系四边形的左下角的 ox, oy 坐标，原点为根四边形的中心，加上某个长度单位（如米）上边长 l 的大小。下图给出说明，假设根四边形的边长为 L：

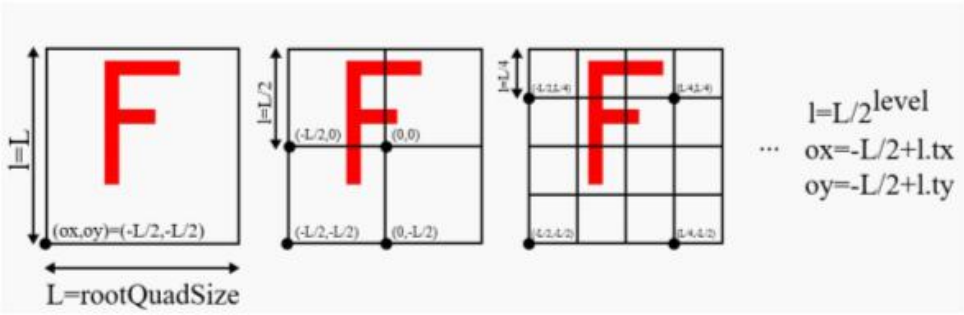

图2；物理四边形坐标 (ox, oy, l)

这些物理坐标只是局部坐标，像一个 Ork 场景图的每个场景节点的局部参考系。在渲染时场景可以用相应的变换，旋转和其他转换放在世界场景的任何地方。

注意：

- 在四边形和块之间有明显的区别。一个四边形是一个四叉树的节点，一个块是一个四边形相关的一些数据。逻辑坐标应用于二者，但物理坐标只用于四边形。实际上块可以包含它相关的四边形物理边界之外的数据。这时我们说块有一个非空边界。有边界的块在生成的数据中引入了一些冗余，但是这些冗余在避免纹理滤波的失真，生成邻居块等时也很有用。下图用包含栅格数据的块说明了这些不同：

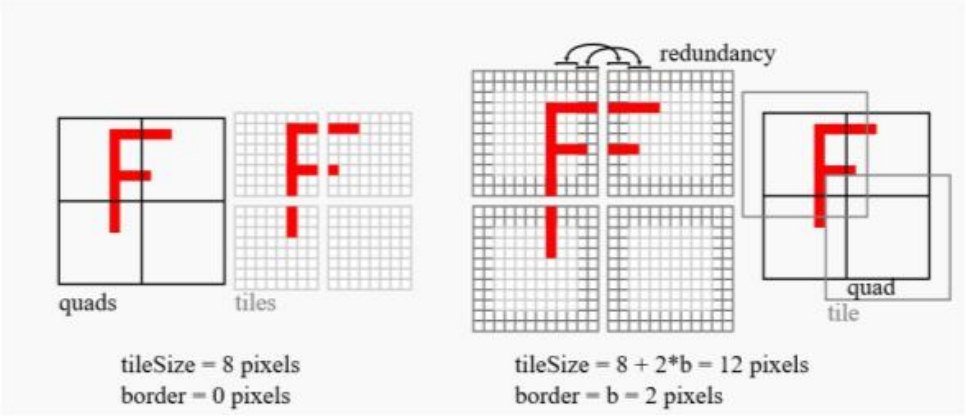

图3；块和四边形的不同：左图：没有边界的块和四边形精确对应。右图：有边界的块引入了冗余

块存储在*块存储*上。有 GPU 上的栅格数据块的块存储（使用纹理），有 CPU 上的栅格数据块的块存储（使用数组），也有矢量数据或其他 CPU 数据的块存储（使用 Ork::Object ）。也可以用 GPU 缓冲定义 GPU 上的块存储，例如顶点缓冲（例如植物处理类使用存储在一个单一 GPU 缓冲中的所有格网为每个四边形生成一个点格网，见 proland::PlantsProducer 源代码）。

一个块存储可以包含多个处理类生成的块。换句话说多个处理类可以使用同一个存储储存它们的块。一个块存储可以包含当前视野需要的块，但也可以包含前一个画面创建的但已经不需要的块。如果观测点回到前一个视野，那么这些块可以直接被再利用：它们将不需要再次生成。类似的一个块存储也可以包含预抓取的块，即，为未来画面提前生成。

哪个块在使用中，即，当前视野需要的，哪个不再使用（从前一帧缓存，或未来一帧抓取），由块缓存管理。一个块缓存也在块的逻辑坐标和存储坐标之间存储一个映像，即，它们在块存储中的位置。像块存储一样，块缓存也被块处理类共享。下图使用一个 GPU 块存储（存储的坐标格式和依赖使用的存储类型的意义。GPU 块存储使用纹理，它是一个层索引 - 纹理存储是一个 2D 数组纹理，一个层一个块）解释了块缓存和块存储之间的关系：

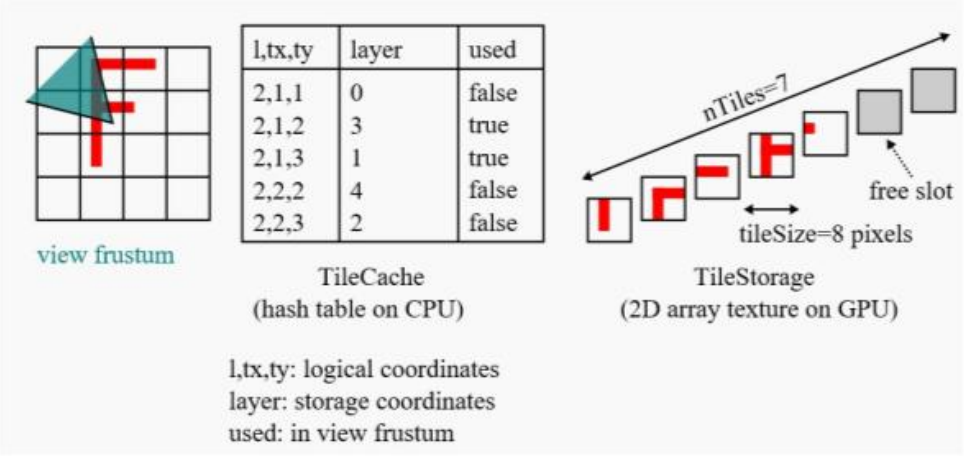

图4；块缓存和块存储之间的关系

该例中，块缓存指明块 (2,1,2) 和 (2,1,3) 在使用中，存储在块存储的 3 和 1 层。它也指明在块存储中的其他 3 个块可用但当前未使用，即，当前视野不需要。该块存储在一个 7 层的 2D 数组纹理中存储块。每个层是一个 8*8 的 2D 纹理，称为一个快照。当前只有 5 个快照被分配，其余快照可以存储其他块。注意一个分配的快照（在存储水平）可以与一个未使用块一致，也可以和使用中的块一致（在缓存水平）。

2.1 块存储

一个块存储用 proland::TileStorage 类表现。这个抽象类有 3 个子类，proland::GPUTileStorage, proland::CPUTileStorage 和 proland::ObjectTileStorage ，分别对应 GPU 栅格数据， CPU 栅格数据，和 CPU 矢量和其他数据（你可以在自己的子类中实现，见 proland::PlantsProducer 源代码示例，基于一个顶点缓冲对象定义了一个 GPU 块存储）。每个块存储有一个容量，是该块存储中的快照的数量，每个快照能存储一个块。一个块存储的容量是固定的不能在运行时被改变。每个快照要么为空要么被分配。一个空快照不能包含任何块，一个分配了的快照只能拥有一个块（要么使用要么不使用）。

该容量由 proland::TileStorage::getCapacity 提取。空快照的数量由 proland::TileStorage::getFreeSlots 给出。空快照可以由 proland::TileStorage::newSlot 获取。返回的快照被认为是可分配的，可以存储一个块。相反一个分配了的快照可以用 proland::TileStorage::deleteSlot 被返回到空快照池。

2.1.1 GPU 块存储

proland::GPUTileStorage 是一个在 GPU 上存储栅格数据的块存储。它使用 2D 纹理或 2D 数组纹理存储块。这样一个块存储可以用 Ork 资源框架创建，如下：

```
<?xml version="1.0" ?>
<gpuTileStorage name="myGpuStorage"
tileSize="196" nTiles="512"
internalformat="RGBA8" format="RGBA" type="UNSIGNED_BYTE"
min="LINEAR_MIPMAP_LINEAR" mag="LINEAR" minLOD="0" maxLOD="1"
tileMap="false"/>
```

该例中每个快照创建为 196*196 像素的存储块（此大小必须包含块边界，如果有的话）。快照的总数是 512 。换句话说该存储分配了一个 196*196*512 2D 数组纹理。该纹理使用 RGBA8 内部格式（71.2 MB）。纹理滤波器和最大最小 LOD 像纹理资源一样被指定。tileMap 属性在地形框架一节解释。

通过 GPU 块存储管理的快照由 proland::GPUTileStorage::GPUSlot 类描述。该类描述了快照在块存储中的位置。它也提供方法来复制一个帧缓冲的一部分或纹理的一部分到快照中。

注意：

- 如果你使用一个明细贴图滤波器，那么每次快照内容变化，你都必须调用 proland::GPUTileStorage::notifyChange （这用于自动更新存储纹理的明细水平）。实际上你不得不这么做，除非重写你自己的处理类。

2.1.2 CPU 块存储

proland::CPUTileStorage 是一个在 CPU 上存储栅格数据的块存储。它使用数组存储块（每个块存储在自己的数组中）。这样一个块存储可以用 Ork 资源框架创建，如下：

```
<?xml version="1.0" ?>
<cpuByteTileStorage name="myCpuStorage"
tileSize="196" channels="4" capacity="1024"/>
```

该例中存储可以储存 196*196 像素组成的 1024 个块，每个像素 4 个字节。使用 cpuFloatTileStorage 属性的话，每个像素则为 4 个浮点型。

一个 CPU 块存储管理的快照由 类描述。该类给出了访问包含块栅格数据的数组的入口。

2.1.3 对象块存储

proland::ObjectTileStorage 是一个在 CPU 上存储任意数据块的块存储。每个快照存储一个指向 ork::Object 的指针。不像 GPU 和 CPU 块存储，这里每个快照的数据不由块存储分配（因为它是任意的）。而是你必须在每次获取一个新快照时手动分配数据，也必须在删除一个快照时手动删除数据。这样一个块存储可以用 Ork 资源框架创建，如下：

```
<?xml version="1.0" ?>
<objectTileStorage name="myObjectStorage" capacity="1024"/>
```

一个对象块存储管理的快照由 proland::ObjectTileStorage::ObjectSlot 类描述。该类通过指针访问块数据。

2.2 块缓存

一个块缓存由 proland::TileCache 类表现。因为一个块缓存不能存储块本身（是由块存储做的），而只能存储和管理逻辑块坐标和块存储的快照之间的映射，一个单一类可以用于所有类型的块。

一个块缓存有相应的块存储。它管理逻辑块坐标和块存储的快照之间的映射。一个块缓存被一个或多个块处理类使用。每次一个块处理类被创建，相应有一个块缓存，它从该块缓存获取一个局部处理类标识符，并且块缓存保留一个对该类的引用。该引用用于当一个来自该处理类的新块从缓存中请求，为了生成它时。

块缓存管理的块可以是正在使用的也可以是未使用的（使用中的块一般是对应于那些需要用来渲染当前景观视野的块）。具体说一个块缓存保持对每个块的使用者数量的追踪。用户使用 proland::TileCache::getTile 方法获取块，用 proland::TileCache::putTile 方法释放块。因此第一个方法增加请求块的用户的计数器，第二个减少该计数器。当计数器为 0 时块变成未使用。

使用中的块是“锁定的”，即，它在块存储中的快照不能用于存储其他块，只要该块在使用中。相反，一个未使用块可以在任何时候从缓存中收回，它的快照可以被用于存储其他块。它发生在，特别是当需要使用一个新块时，而所有快照都是被分配的。一个收回的块如果在未来再次需要，则需要被重新生成。为了最小化一个块被重生成的次数，块缓存优先收回那些长时间不使用的块（该启发式算法称为**最近使用法则** - 或 LRU - 启发式缓存）。

除了 getTile 和 putTile ，块缓存中的主要方法还有：

- proland::TileCache::findTile 方法用于在缓存中查找一个块。该方法不改变返回块的用户数量。该块可以在使用中的块列表中查找，或块缓存管理的所有块中查找，无论是否在使用。
- proland::TileCache::prefetchTile 方法用于请求生成一个未来画面需要的块。该方法立即返回。该块将作为未使用块生成。如果没有空快照存储此预抓取的块，那么一个未使用块将被收回。
- proland::TileCache::invalidateTiles 方法用于强制生成该块的类重新生成该块。所有的块在块存储中保留它们的当前快照，但快照内容在使用前将重计算（意思是使用中的块将立即重计算，未使用的块在使用前重计算）。

当一个块被 getTile 请求时有两种情况：

- 如果该块在缓存中，它的用户数量增加 1 。如果该块未使用，则变为使用中
- 否则该块在一个空快照中或一个先前收回块（未使用）的快照中被生成。实际上该块不是立即生成。而是返回一个生成块的 ork::Task 任务（在 proland::TileCache::Tile 内部）。该任务在该块数据使用之前被 ork::Scheduler 执行。

2.2.1 块缓存资源

一个块缓存可以用 Ork 资源框架载入，有以下格式（嵌套存储资源当然也可以是一个 cpuXxxTileStorage 或一个 objectTileStorage ；它描述与块缓存相关的块存储）：

```
<?xml version="1.0" ?>
<tileCache name="myCache" scheduler="myScheduler">
<gpuTileStorage .../>
</tileCache>
```

2.3 块处理类

一个块处理类由 proland::TileProducer 类表现。该抽象类有很多具体子类，在处理类一节中介绍。一个块处理类与一个块缓存相关，用于缓存它生成的块（由 proland::TileProducer::getCache 给出）。一个块处理类只与一个块缓存相关，但一个块缓存可以与多个块处理类相关（如果生成的块是同一类型）。为了区分不同处理类生成的块，每个处理类在块缓存中有一个唯一的局部识别符，处理类创建时自动由 proland::TileProducer::getId 赋予。

一个块处理类的主要方法是 proland::TileProducer::doCreateTile 抽象方法。它实施块生成算法，即，它定义了块如何生成。然而该方法从不直接调用，而是通过 proland::TileProducer::getTile 方法调用。如果一个请求的块不在缓存中， getTile 不会立即生成块。而是返回一个生成该块的 ork::Task 任务。

一个“基本”的块处理类返回一个“基本”的 ork::Task 任务来生成块，即，一个不依赖其他任务的任务。相反，使用另一个处理类生成的块作为输入参数的块处理类返回一个 ork::TaskGraph 生成块。该任务图包含生成块的任务，依赖于生成输入数据的任务。例如一个法线处理类，从高程处理类生成的地形高程来生成一个高程法线。那么我们有如下处理，缓存和存储：

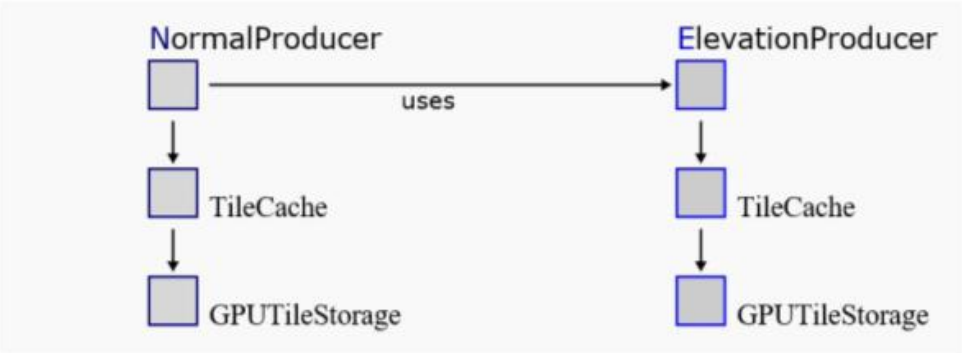

图5：使用另一个处理类生成的块做输入的处理类

那么法线处理类的 getTile 方法返回的任务图如下：

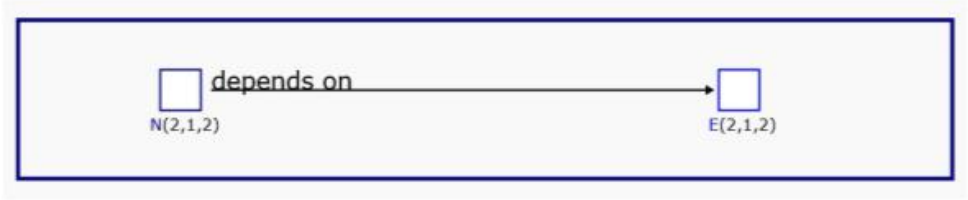

图6：使用另一个处理类生成的块作为输入的任务图

生成法线块 (2, 1, 2) 的任务 N(2, 1, 2) 有一个对任务 E(2, 1, 2) 的依赖，其为同一个四边形生成高程，二者都被放入任务图中。

特别是高程处理类不是一个"基本"类：它使用残差处理类生成的数据块作为输入（见处理类一节）。它也递归的使用自己：实际上一个高程块通过上采样和叠加残差数据从父高程生成。一个高程处理类可能也使用图形处理类生成的矢量数据（通过图层 - 见下文），也使用自己迭代。所以实际上我们在法线，高程，残差和图形处理类之间有如下关系（这我们没有给出相关的缓存和存储）：

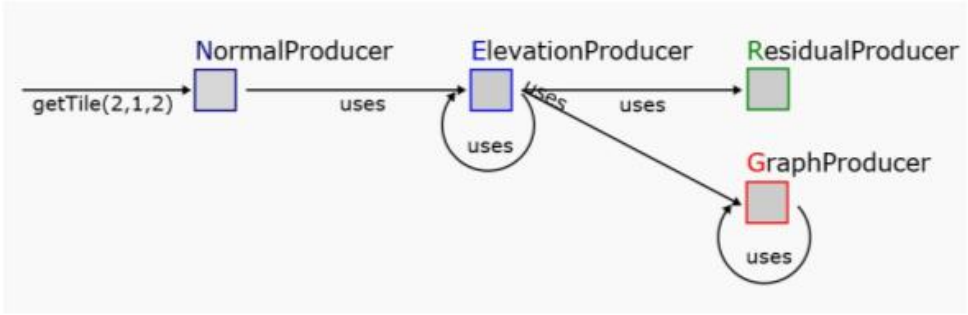

图7：法线，高程，残差和图形处理类之间的关系

生成法线块 (2,1,2) 的任务 N(2,1,2) 是一个复杂任务图（现实中该图甚至更复杂，因为一个法线处理类也使用自己作为迭代，像高程和图形处理类一样）：

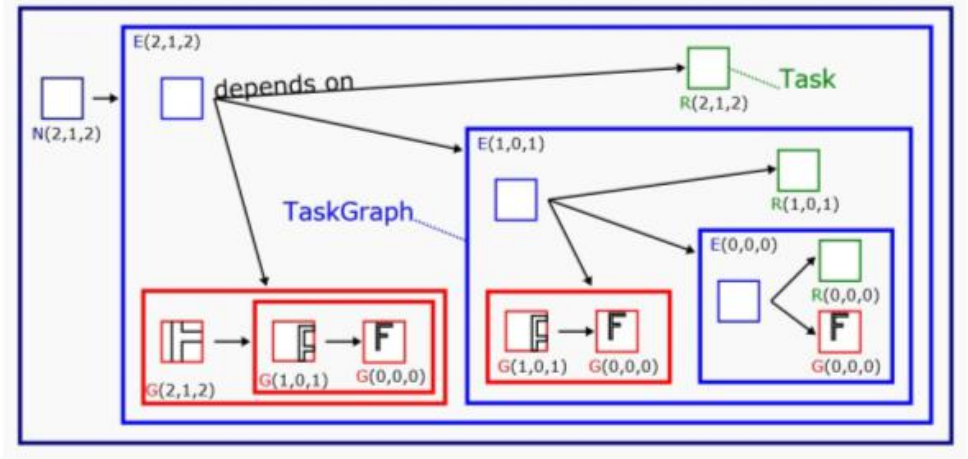

图8：法线块 N(2,1,2) 的任务图

在最高级我们仍然有一个嵌套任务 N(2,1,2) 和 E(2,1,2) 的任务图，像前一个例子一样互相之间有依赖。因此尽管 E(2,1,2) 任务本身就是一个任务图。注意整个图的"片段"：来自高程处理类对自己的迭代使用，和图形处理类对自己的迭代使用。也要注意此双重迭代导致了出现在多个任务图的任务，即，在任务图之间共享（每个共享任务只有一个实例）。

为同一个块连续调用 getTile 方法，总是返回同一个 ork::Task 实例。因此一个任务实例只执行一次（见任务图），上图中，一旦调度，将不会导致它包含的任务的执行。实际上多数任务可能已经执行过，所以不会被执行。然而，如果没有任务被明确重调度（这发生在是否相关块是 proland::TileProducer::invalidateTiles 无效的），那么 Ork 框架将自动重调度与其直接或简介依赖的任务。因此所有直接或简介依赖于无效块的块都会自动重计算。

2.3.1 主要方法

proland::TileProducer 类提供一些一般方法，这些方法提供了它能生成的块的信息：

- proland::TileProducer::isGpuProducer 指出是否该处理类在 GPU 上生成栅格数据块。一个 GPU 处理类被假定使用 proland::GPUTileStorage 相关的块缓存。
- proland::TileProducer::getRootQuadSize 给出该处理类管理的四叉树的根四边形的物理大小，该大小可以由 proland::TileProducer::setRootQuadSize 设置。
- proland::TileProducer::getBorder 方法指出是否该处理类生存的块有边界。具体说它指出这些边界的大小，像素为单位（假设该块包含栅格数据）。默认返回 0（即，没有边界），但可以重写它。
- proland::TileProducer::hasTile 方法指出是否该处理类能生成由它的逻辑坐标指定的块。默认该方法总是返回真，但是你可以重写它。例如，常见的有为处理类限制一个最大分辨率，即，四叉树的最大级别。
- proland::TileProducer::hasChildren 方法指出是否处理类可以生成一个由它的逻辑坐标指定的块的子块。处理类框架假定如果一个块的子块可以生成，那么该块的四个子块将被生成。因此如果指定块的左下角块可以被生成，该方法返回真（返回 hasTile）。

也有 proland::TileProducer::update 方法，每个画面调用一次（通过 proland::UpdateTileSamplersTask 和 proland::TileSampler::update ）。该方法默认什么都不做，但是需要的话你可以重写它使一个块无效（即，如果一些输入数据用于生成一个已经改变的块 - 如果该输入数据由其他处理类生成那么你就不需要这么做，它会通过 Ork 任务框架自动完成）。或者，如果生成数据必须被激活，你也可以通过此方法修改每个画面中以及生成的内容。

最后，proland::TileProducer 类也提供一些方便的方法在与它相关的块缓存上调用相应的方法。这些方法有：

- proland::TileProducer::getTile
- proland::TileProducer::putTile
- proland::TileProducer::findTile
- proland::TileProducer::prefetchTile

- proland::TileProducer::invalidateTiles

2.3.2 块处理类层

一些块处理类可以由图层定制。一个图层可以修改"原"处理类或前一个图层生成的数据。例如你可以设想一个图层在一个卫星图片（"原"数据）上从矢量数据绘制道路，或一个图层基于同样的道路矢量数据修改地形高程，在地形中生成道路轨迹。像处理类一样，一个图层可以使用其他处理生成的数据做输入（前例中，图层使用图形处理类的矢量数据做输入）。一个块处理类的图层由 proland::TileProducer::getLayerCount, proland::TileProducer::getLayer 和 proland::TileProducer::addLayer 方法管理。

2.4 用户自定义处理类

你可以通过扩展 proland::TileProducer 类自定义块处理类。本节演示了如何这样做，以一个使用 CPU 处理类生成的块作为输入的 GPU 处理类为例：

```
class MyProducer : public TileProducer
{
public:
MyProducer(Ptr<TileCache> cache, Ptr<TileProducer> input, ...) :
TileProducer("MyProducer", "MyCreateTile")
{
init(cache, input, ...);
}
virtual ~MyProducer()
{
}
protected:
MyProducer() : TileProducer("MyProducer", "MyCreateTile")
{
}
void init(Ptr<TileCache> cache, Ptr<TileProducer> input, ...)
{
TileProducer::init(cache, true);
this->input = input;
...
}
```

以上代码包含了创建处理类的初始化代码，使用该样式能容易的定义该类的一个 Ork 资源子类（见用户自定义资源）。构造函数使用一个块缓存做参数，用于缓存生成的块。该参数由超类的构造函数请求。该构造函数也使用作为参数的块的处理类作为参数。我们假设它是一个 CPU 处理类。

```
virtual Ptr<Task> startCreateTile(int level, int tx, int ty,
unsigned int deadline, Ptr<Task> task, Ptr<TaskGraph> owner)
{
Ptr<TaskGraph> result = owner == NULL ? new TaskGraph(task) : owner;
TileCache::Tile *t = input->getTile(level, tx, ty, deadline);
result->addTask(t->task);
result->addDependency(task, t->task);
return result;
}
```

startCreateTile 方法重写了超类的相关方法。它的角色是构造生成给定块的任务或任务图。这里我们的处理类使用另一个处理类生成的块做输入，所以生成我们的块的"基本"任务 - 自动传递给 task 参数并构建 - 必须对该输入任务有一个依赖（以至于在我们开始生成块之前执行）。这是为什么该方法创建了一个包含 task 的任务图，和任务 t 来生成输入。然后添加这些任务到任务图，并在它们之间创建一个依赖。注意 t 是有 getTile 获取的：这将锁定该输入块直到我们调用 putTile ，即，输入数据不会无故从缓存中收回。

```
virtual bool doCreateTile(int level, int tx, int ty, TileStorage::Slot *data)
{
CPUTileStorage<unsigned char>::CPUSlot *in;
GPUTileStorage::GPUSlot *out;
TileCache::Tile *t = intput->findTile(level, tx, ty);
in = dynamic_cast<CPUTileStorage<unsigned char>::CPUSlot*>(t->getData());
out = dynamic_cast<GPUTileStorage::GPUSlot*>(data);
...
getCache()->getStorage().cast<GPUTileStorage>()->notifyChange(out);
}
```

doCreateTile 方法重写了超类的相应方法。它的角色是生成所需的块。这里该方法首先获取所需的输入块。注意使用 findTile 完成：我们确定该块在缓存中因为它直到我们调用 putTile 才被收回（见上文）。然后它获取必须生成的块的快照。该快照在 data 中作为参数传递但必须抛为正确类型。一旦块被生成（通过"点"），处理类通知它的块存储该块的快照被改变，所以存储纹理的明细水平在需要时将自动被更新（见 GPU 块存储）。

```
virtual void stopCreateTile(int level, int tx, int ty)
{
TileCache::Tile *t = input->findTile(level, tx, ty);
input->putTile(t);
}
private:
Ptr<TileProducer> input;
};
```

stopCreateTile 方法重写了超类的相应方法。它的角色是清理在块生成时使用的资源。这里该方法在块被用做输入时调用 putTile ，因此该块的内容不再被需要。此调用的目的是解锁该输入块（如果它没有被其他用户锁定），使其可以在任何时候从它的缓存中收回。

2.4.1 用户自定义处理类层

你可以通过扩展 类自定义块处理类图层。该任务和块处理类的定义非常相似。特别是一个块处理类图层有相同的 startCreateTile, doCreateTile 和 stopCreateTile 方法，它们有相同的角色可以以相同的方式重写。

3 地形框架

地形渲染框架管理一个或多个地形，每个地形与一个块处理类集合相关。对每个地形来说，地形四叉树是基于当前观测位置动态划分的。当划分了新的四边形，与地形相关的处理类用于生成对应的块。地形框架也提供了新的 GLSL 一致变量，允许着色器像使用一个法线纹理一样使用纹理缓存的快照。因此你可以在你的着色器中像使用法线纹理一样使用生成的块。最后该框架提供了将一个平地形映射为其他形状的变形，如球形（渲染球体）。

注意：

- 这里我们讲"地形"但是实际上框架并不限制于地形本身。实际上与地形相关的块处理类可以生成任何类型的数据，包括渲染地形上的 3D 植被或建筑物（见 "trees1" 实例）。因此"地形"框架可以用于渲染整个景观。

3.1 地形变形

地形框架支持地形变形。这里一个变形不是局部地形修改。而是空间上的一个全局变形，例如转换一个平原到球面或圆柱面。地形变形用于生成球形星球，柱面地形等（例如一个旋转模拟重力的柱形太空飞船）。

一个变形转换一个局部空间的点到一个变换空间：

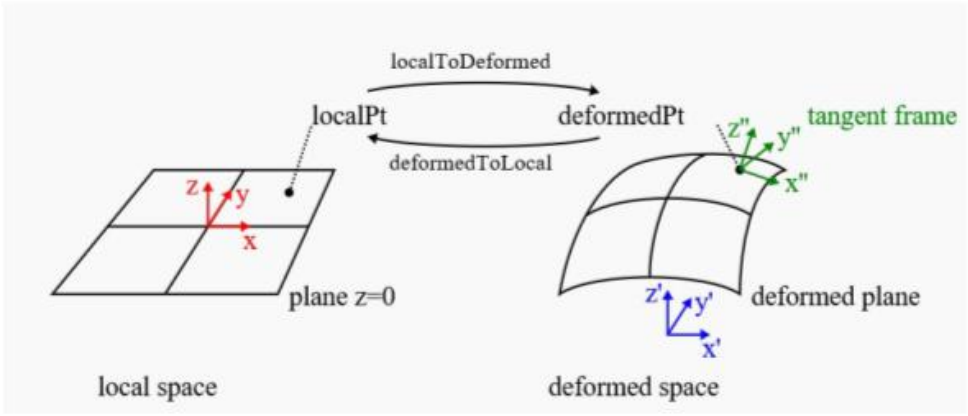

图9：地形变形注释

特别是局部空间是四边形物理坐标定义的空间 - 见处理类框架。在局部空间中"海平面"表面是平面 z=0 ，z 是垂直坐标轴。该平面可以变换为球面，柱面等。然而注意一个变形转换了整个 3D 空间，而不是一个单独的 2D 表面（这是转换海平面以上的点）。

proland::Deformation 类表现了一个地形变形。它定义了地形变形提供的方法，以及实施方法来辨别变形（即，没有变形）。 proland::SphericalDeformation 是一个子类，将水平面变换为球面。最后，proland::CylindricalDeformation 子类将水平面变换为柱面。注意你可以自定义自己的变换子类：

- proland::Deformation::localToDeformed ：变换一个局部空间的点到变形空间
- proland::Deformation::deformedToLocal ：变化一个变形空间的点到局部空间
- proland::Deformation::localToDeformedDifferential ：计算一些局部点上变形函数的微分。该微分给出了一个点周围变形的线性逼近：如果 p 接近 localPt ，那么相对于 p 的变形点可以用 localToDeformedDifferential(localPt) * (p - localPt) 估算。
- proland::Deformation::deformedToTangentFrame ：计算一个变换点的切线空间的正交法线参考系。该参考系的 xy 平面是变换点的切线平面，相对于局部平面 z=cste 的变换平面。该正交法线参考系不给出反变换函数的微分，一般不是一个正交法线变换。该切线参考系定义了地形法线计算出的切线空间。

proland::Deformation 类也用于设置需要在 GPU 变化地形顶点的 GLSL 着色器一致变量。使用 proland::Deformation::setUniforms 方法完成。有两种方法：第一个可以设置不取决与四边形的一致变量（如为球形变换设置球形半径），第二个可以为指定四边形设置一致变量。这些方法设置的 GLSL 一致变量取决于实际变换。

3.1.1 球形变形

proland::SphericalDeformation 把水平面变换为球面。它设计为渲染一个半径为 R 的球体（在海平面），使用位于一个 2R*2R*2R 大小的立方体的面上的 6 个地形，每个地形变换为球体的一部分：

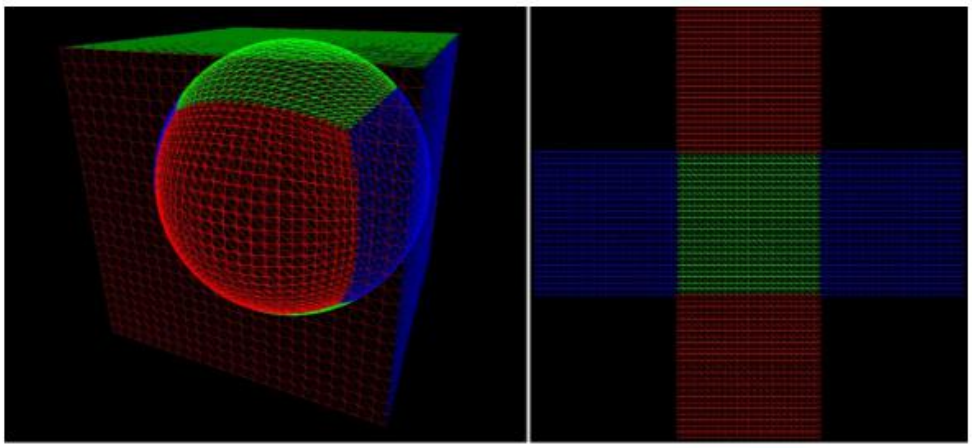

图10：左图：球形变换可以用 6 个变换地形形成一个球体。右图：映射球体到立方体的反变换

数学上，变换定义如下：从局部空间的一个点 p=(x,y,z) ，我们首先在球体参考系（原点为球体中心的参考系）中，在立方体"顶部"（上图中绿色部分）构建一个点 P=(x,y,R)。该点用于在球体参考系中定义变换点，如

$$q = (R+z)\ P\ /\ \|\ P\ \|,\ \text{where}\ P = (x,y,R)$$

该变形映射平面 z=0 到一个半球面。因此至少两个地形需要用于覆盖整个球面。为了限制变形，最好使用立方体面上的 6 个地形，如上所述。反变换在平面上（如对立方体映射 - 见上图）映射整个球面到立方体，除了南极"面"。对"北"面来说，反变换为：

$$p = (R\ q_x/q_z,\ R\ q_y/q_z,\ \|\ q\ \| - R)$$

变化点 q 的地形法线计算的切线参考系，由以下平面参考系的矢量单位定义：

$$u_x = (0,1,0) \times u_z\ /\ \|\ (0,1,0) \times u_z\ \|$$

$$u_y = u_z \times u_x$$

$$u_z = q\ /\ \|\ q\ \|$$

GLSL 一致变量

理论上 $q = (R+z)\ P\ /\ \|\ P\ \|$ 变换可以很容易在 GPU 实施。然后有两个精度问题，即使使用 32 位浮点型。它们与一个事实有关，即变换点是在一个原点位于球体中心的参考系中计算的。因此对于一个球体如地球，变换点的坐标非常大（地球半径约为 R=6360000 米）没有足够的位数左移来精确表现海拔。另一个问题是当这些坐标变换到相机参考系时，原点必须在平面坐标系中表达（减去两个互相接近的大数导致不准确的结果）。

为了解决这两个问题，思路是计算变形四边形的角并在 CPU 中使用双精度变换它们到相机参考系。结果是有"小"坐标的点，可以容易的在 GPU 中插值而没有精度问题。具体来讲，我们在 CPU
上计算四边形角 $c_i(i=1,\ldots4)$ 和这些角的顶点矢量 n_i，在相机参考系中，我们在 GPU 中计算。这个思路将变形顶点作为变形角的插值计算，取代顶点矢量的插值:

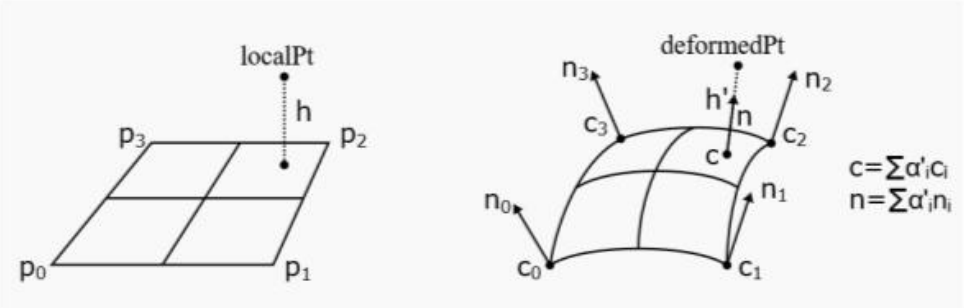

图11：一个变形点可以作为变形四边形 c 的插值计算，沿插值后的角垂直轴 n 翻转

我们注意局部空间中四边形的角 pi ，和相应的变形点 ci （ $c_i = R\,P_i / \|P_i\|$ ，平面参考系中 $P_i = (p_{ix}, p_{iy}, R)$ ）。也要注意变形垂直向量 ni （平面参考系中 $n_i = P_i / \|P_i\|$ ）。我们要计算一个局部点相应的变形点，定义为 $p = \sum \alpha_i p_i + (0,0,h)$ ($\sum \alpha_i = 1$)。我们表达变形点 q 为：

$$q = \sum \alpha'_i c_i + h' \sum \alpha'_i n_i$$

最终我们想要 $\sum \alpha'_i = 1$ ，以至于以上公式可以在任何参考系中使用。未知 α'_i 和 h' 可以通过在平面参考系中的以上关系计算，通过于该参考系中 q 的定义的比较， $q = (R+h) P / \|P\|$ ：

$$q = \sum \alpha'_i c_i + h' \sum \alpha'_i n_i = (R + h') \sum \alpha'_i P_i / \|P_i\|$$

$$q = (R + h) P / \|P\| = (R + h) \sum \alpha_i P_i / \|\sum \alpha_i P_i\|$$

$$\sum \alpha'_i = 1$$

可见有 $\alpha'_i = k \alpha_i \|P_i\| / \|\sum \alpha_i P_i\|$ ，前两行变成 $k.(R + h') = (R + h)$ 。我们就可以从 k 计算 h' ，使用第三个公式计算 k ，得到：

$$\alpha'_i = \alpha_i \|P_i\| / \sum \alpha_i \|P_i\|$$

$$h' = [h + R(1-k)] / k, \text{ where } k = \|\sum \alpha_i P_i\| / \sum \alpha_i \|P_i\|$$

我们在 CPU 上用双精度计算变形角和垂直向量 ci 和 ni ，直接在屏幕空间中表达（即，相机参考系中变换之后，和透视投影之后）。我们也在 CPU 上计算法线 $\|P_i\|$ 。proland::Deformation::setUniforms 方法传递这些值到 screenQuadCorners 和 screenQuadVerticals mat4 一致变量, 以及 screenQuadCornerNorms vec4 一致变量。着色器可以用以下代码计算四边形格网的变形顶点的屏幕空间坐标（见"terrain2"实例对该代码的具体使用 - 我们假设 zfc 变量包含了当前顶点的高程值 zf，zc，zm ）：

```
float R = deformation.radius;
mat4 C = deformation.screenQuadCorners;
mat4 N = deformation.screenQuadVerticals;
vec4 L = deformation.screenQuadCornerNorms;
vec3 P = vec3(vertex.xy * deformation.offset.z + deformation.offset.xy, R);
vec4 uvUV = vec4(vertex.xy, vec2(1.0) - vertex.xy);
vec4 alpha = uvUV.zxzx * uvUV.wwyy;
vec4 alphaPrime = alpha * L / dot(alpha, L);
float h = zfc.z * (1.0 - blend) + zfc.y * blend;
float k = min(length(P) / dot(alpha, L) * 1.0000003, 1.0);
float hPrime = (h + R * (1.0 - k)) / k;
gl_Position = (C + hPrime * N) * alphaPrime;
```

该代码首先计算 P 中 P= ∑ α i P i 。它在 alpha 中基于 xy 顶点坐标计算了 α i （假设四边形中不同于 0 和 1 ）。它然后在 alphaPrim 中计算了 α' i ，计算 h 和 k，在 hPrim 中计算了 h' ，最后用 alphaPrime 系数插值计算了结果。

proland::Deformation::setUniforms 方法也设置了一个 tangentFrameToWorld mat3 一致变量（使用以上定义的 ux, uy 和 uz ），用于转换四边形中心的切线参考系的地形法线到平面参考系中：

```
vec3 Ntangent = ...; // fetches normal in tangent space
vec3 Nworld = deformation.tangentFrameToWorld * Ntangent;
```

3.2 地形四叉树

3.2.1 基于距离划分

地形框架用一个基于当前视点位置动态划分的四叉树表现地形，为了提供观测者附近更多的细节。该划分是仅基于视点到四边形的距离，即，它不依赖于该四边形的数据块的"复杂性"。具体讲，如果它到视点的距离 d 小于它的边长 L ，那么四边形就划分，d 不是欧几里德距离，而是 max(dx, dy) 距离：

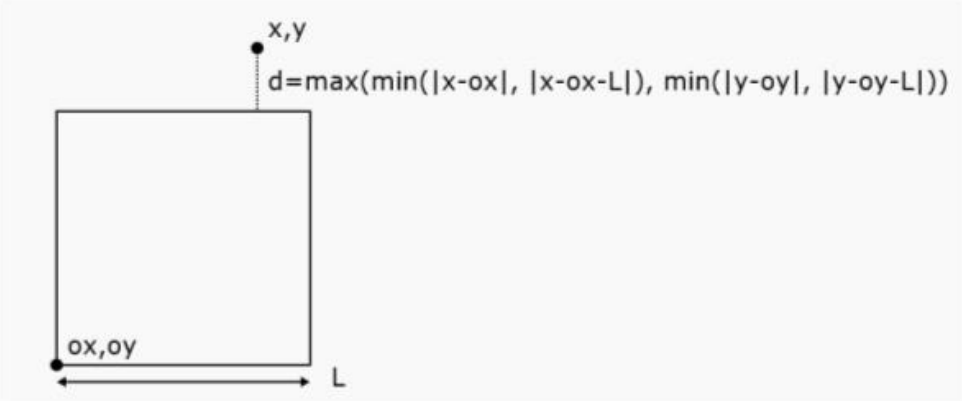

图12：如果它到视点的距离 d 小于它的边长 L ，那么一个四边形就被划分。 k 是距离划分因子

我们称 k 为距离划分因子。如果你想得到一个限制的四叉树，即，一个四叉树中两个相邻四边形的层次之间的差别总是 0 或 1，那么 k 必须大于 1 ：

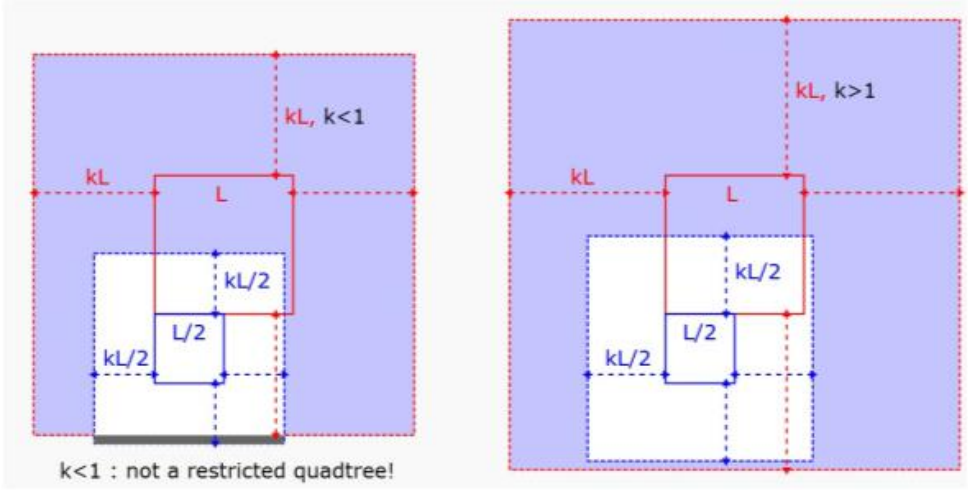

图13：距离划分因子必须大于 1 。左图：对灰色区域的观测点来说，蓝色四边形会被划分而不是红色那个，因此四叉树不受限制。右图：当 k > 1 这个问题就不会出现

k 值的增加意味着四边形不久被划分并且在屏幕上显得更小。因此你调整 k 值来得到屏幕上给定的分辨率。例如，如果每个四边形用 T*T 像素的纹理渲染，屏幕上这些纹理像素的投影大小将最多为 W/(2k.T.tan(fov/2)) ，其中 W 是像素为单位的屏幕宽度，fov 是视角的广度。下图给出了多个大于1 的 k 值的四叉树划分规则的结果的例子（见 “helloworld” 例子）：

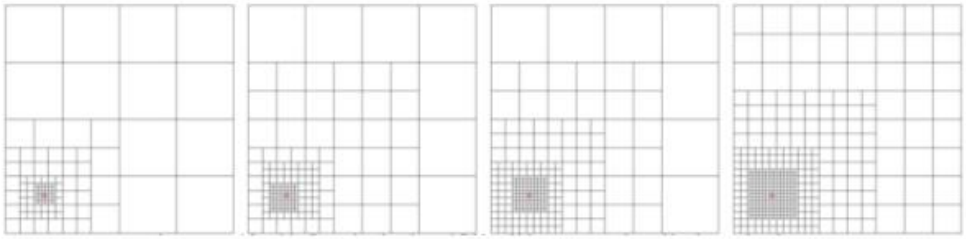

图14：从左至右：以红点为视点，基于 k=1.1, 1.5, 2.0, 2.8 的四叉树划分距离

注意：

- 特别是四边形和视点之间的距离也涉及海拔：d=max(min(|x-ox|,|x-ox-L|), min (|y-oy|,|y-oy-L|), z-groundz) ，其中 z-groundz 是地面之上的相机的高度。注意该距离在局部空间中计算，而不是变形空间（见上文）。为此相机位置从变形空间转换到局部空间。

3.2.2 细节的连续级

当一个四边形被划分，因为突然被四个有着新数据的子四边形替代，会产生爆音。但愿，在 k>1 时，它可以在新子块数据内加快消退，并且在旧的父块数据外也相应消退。这用一个渐进的融合取代了突然的转换。该融合可以这样做：在四边形 (ox, oy, l) 的点 x,y 上，如果视点在 (cx, cy)， 那么融合系数定义为：

blend = clamp((d/l-k-1)/(k-1), 0, 1), where d=max(|x-cx|, |y-cy|)

用来混合旧的父块数据和新的子块数据：

$$x = blend * x_{parent} + (1-blend) * x_{child}$$

实际上，当视点位于到四边形的最小距离时， dmin = kl ， 我们得到 blend=clamp(-1/(k-1), 0, 1) = 0, 而且 x = x child 。反过来，当视点位于到四边形的最大距离时， d max =(2k+1)l ，我们得到 blend = clamp(k/(k-1), 0, 1) = 1 而且 x = x parent 。这就很容易在 发生时计算距离，给定了 blend=0 和 blend=1 之间转换区域的宽度。该宽度等于 (k - 1)l ，说明 k 越大，转换区域越大，可察觉越少。下图做了说明：

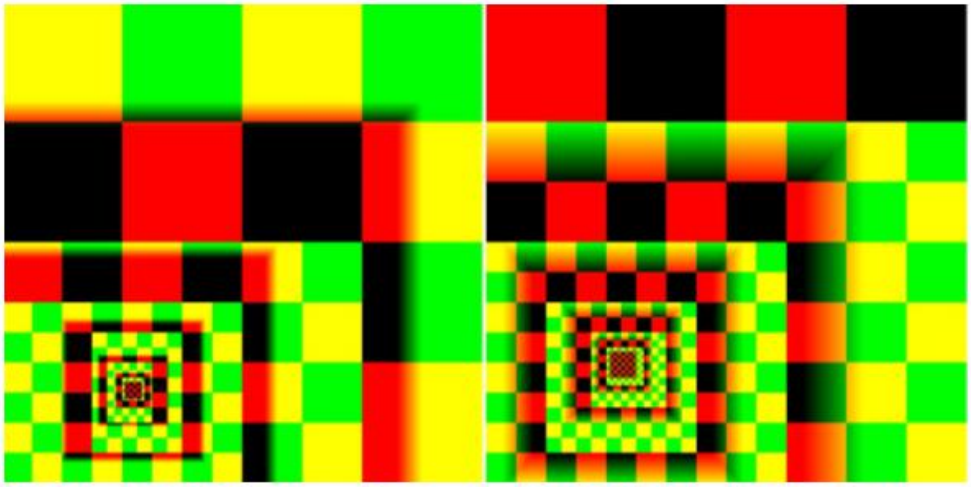

图15：四叉树顶部的融合系数（绿色）。k=1.2 （左图）的转换区域小于k=2 的（右图）

GLSL 一致变量

proland::Deformation::setUniforms 方法设置了两个一致变量，用来在 GPU 上计算上文的融合系数。 deformation.camera vec4 一致变量存储相机位置，相对于四边形左下角 ox, oy 和边长 l 。deformation.blending vec2 一致变量存储 k+1 和 k-1 。使用这些一致变量，融合系数可以按如下计算（四边形的顶点坐标假设在 0 和 1 之间）：

```
vec4 c = deformation.camera; // (cx-ox)/l, (cy-oy)/l, (cz-groundz)/l
vec2 k = deformation.blending; // k+1, k-1
vec2 v = abs(c.xy - gl_Vertex.xy);
float d = max(max(v.x, v.y), c.z);
float blend = clamp((d - k.x) / k.y, 0.0, 1.0);
```

3.2.3 地形类

地形四叉树由 proland::TerrainQuad 类的树表示。该类的每个对象提供如下字段：

- parent 给出指向父四边形的指针
- level, tx, ty 给出四边形的逻辑坐标
- ox, oy, l 给出四边形的物理坐标
- zmin 和 zmax 给出四边形高程的最大最小值
- children 是该四边形的四个子四边形的指针数组。它要么包含四个空指针，如果该四边形是一个叶子，或者如果四边形被划分，则包含四个指向子四边形的非空指针（左下，右下，左上和右上）
- visible 指示从视点来看，是否四边形是不可见的，部分可见的或全部可见的。该字段由 proland::TerrainQuad::update 方法更新

proland::TerrainNode 类表达一个地形。包含一个指向地形四叉树的根 proland::TerrainNode::root 的指针。它有如下字段：

- splitDist 是用于基于距离划分的划分因子 k（见上文）
- maxLevel 是划分必须停止的最大四叉树层次
- deform 是用于地形的变形（见上文）

proland::TerrainNode 存储当前视点位置和当前视锥面。这些值可以使用 proland::TerrainNode::getDeformedCamera, proland::TerrainNode::getDeformedFrustumPlanes, 和 proland::TerrainNode::getLocalCamera 从局部和变形空间中获取（见地形变形）。它们通过 proland::TerrainNode::update 方法更新，该方法使用一个定义了世界空间中地形位置的场景节点作为输入（从哪个相机位置也可以被获取）。

3.2.4 地形资源

proland::TerrainNode 可以用 Ork 资源框架载入，有以下格式：

```
<terrainNode name="myTerrain" size="6360000" zmin="0" zmax="10000"
deform="sphere" splitFactor="2" maxLevel="16"/>
```

该资源描述了一个地形，它的根四边形为 12720*12720 千米（ 12720=2*6360），高程在 0 到 10000 米之间，使用球形变形（当然长度单位是可以自己定义）。该地形四叉树将被 k=2 的划分因子（对一个 80 度的视场，视口宽度为 1024 像素。对于一个更小的视场和/或一个更大的视口，划分将自动在更大的距离上做，以至于一个四边形的边长在像素上或多或少保持不变）划分到 16 层（包括 16 层）。注意：当前 deform 选项仅支持 none 和 sphere 。在该球形变形中，球体半径设置为 size 。”terrain1” 和 “terrain2” 实例解释了平面和球面的地形节点如何使用。

3.3 地形块取样器

proland::TerrainNode 仅存储一个基于到当前视场位置距离划分的地形的当前四叉树。它不存储该四叉树的任何其他数据。实际上地形或景观数据是块处理类生存的，存储在块存储中，由块缓存管理（见处理类框架）。因此我们需要一个在地形和块处理之间的链接，使得当地形四边形被划分时，处理类被要求生存新块。该链接由 proland::TileSampler 类提供。

proland::TileSampler 与一个 GPU 块处理类相关。它的首要角色是当地形四边形被划分时，请求处理类生存新块。第二个角色是设置 GLSL 一致变量来允许着色器通过该处理类在块存储上获取一个纹理块。

第一个角色由 proland::TileSampler::update 方法实施。该方法以地形四叉树的根做输入参数。它在该四叉树和上一次调用该方法的前一个值之间比较。那么，对每个新的四边形，它使用 getTile 请求处理类生存相应的新块。相反，对于每个旧四边形（即，四边形不再是四叉树的一部分），它使用 putTile 通知处理类相应的块不再被使用。这确保了块数据在块存储中是"锁定的"（见块缓存），只要相应的四边形存在。

注意：

- 实际上 proland::TileSampler::update 方法返回一个包含生成所有必须生成的新块的任务的任务图。为了实际上生成块，该任务图必须调度来执行。

实践中它不总是需要为四叉树的每个四边形生成块。例如它只足够为叶子四边形生成块，即，那些以及被事实上渲染过，不需要子四边形的。它也只为可见四边形生成块，即，那些视锥面中完全或部分可见（见地形类）。为了指出是否必须为一个四边形生成块，你可以用以下配置方法：

- proland::TileSampler::setStoreLeaf 指出是否为一个叶子四边形生成块。默认是真
- proland::TileSampler::setStoreParent 指出是否为一个内部四边形生成块（即，非叶子）。默认是真
- proland::TileSampler::setStoreInvisible 指出是否需要为视锥外的四边形生成块。默认是真
- proland::TileSampler::setStoreFilter 添加一个任意块滤波器到滤波器列表。每个滤波器使用一个地形四边形作为参数，返回是否需要为它生成块。如果至少一个滤波器决定该块需要被生成，那么就生成它

最终一个块取样器可以用于两种模式的一种：同步或异步。默认为同步模式，更新方法对需要为新建四边形生成块的任务使用立即期限。意味着最终画面直到所有块生成才会显示。当块数据必须从硬盘加载时，是有高延迟的，这会在观测者移动时画面出现定格。

这可以用异步模式解决。该模式中每个生成块的任务不对当前画面设置期限。因此一个画面可以显示即使一些数据缺失。这时第一个祖先块被用于代替。这解决了延迟问题，但是当观测者移动太快时也会降低显示质量，当等待时新数据突然代替临时低分辨率数据，也会产生可见的

爆音失真（甚至导致在地形四边形之间的缝隙，因为用于显示的四叉树不需要是严格四叉树）。为了使用异步模式，必须配置好一些选项（"earth-srtm-async" 例子有说明 - 特别注意调度器的定义）：

- proland::TileSampler::setStoreParent 必须设置为真。这是为了保证如果一个数据没有准备好，至少能找到一个数据以及准备好的祖先
- proland::TileSampler::setAsynchronous 必须设置为真
- 最后，ork::Scheduler 必须支持任何类型的任务预抓取（ CPU 和 GPU ）。使用 ork::MultithreadScheduler ，这只有在指定预抓取率，或者一个固定的帧率时可以这样做

注意：

- 你可以混合同步模式中的块取样器和其他异步模式的取样器。因此一些块可以同步生成而另一些数据为异步生成。

3.3.1 GLSL 函数

如上所述，proland::TileSampler 的第二个角色是设置 GLSL 一致变量允许着色器在块存储中访问纹理块。该角色通过 proland::TileSampler::setTile 方法实施，该方法使用一个块的逻辑坐标作为输入参数。使用 findTile 在块存储中找到该块的位置。然后设置必要的 GLSL 一致变量从一个着色器访问该存储快照的内容。具体讲，如果请求的块没有找到，将使用它的父块。如果该父块也没找到，则使用该父块的父块，以此类推，直到找到请求块的祖先。那么必要的 GLSL 一致变量设置用于允许着色器访问相应请求块的祖先块的一部分。

为了使块存储中的纹理块更容易使用，一个块存储被看作一个新型的纹理。和 1D, 2D, 2D 数组或 3D 纹理一样，在 GLSL 中用 sampler1D, sampler2D, sampler2DArray 或 sampler3D 声明，用 texture1D(), texture2D(), texture2DArray(), 或 texture3D() 函数使用，我们为存储在块存储的纹理定义一个新的 samplerTile 类型和一个新的 textureTile 函数。它们的定义由 textureTile.glsl 文件提供。

因此由 GPU 块处理类 p 生成的块可以如下访问。我们首先用该处理类创建一个 TileSampler ：

```
Ptr<TileSampler> u = new TileSampler("mySamplerTile", p);
```

名字 “mySmaplerTile” 是 samplerTile 一致变量的名字，将用于在着色器中访问块。 update 方法被调用后，在它返回的任务被执行后，我们设置 “mySamplerTile” 一致变量的值为指定块，在当前选择的 GLSL 程序中（见 ork::SceneManager::getCurrentProgram ），使用：

```
u->setTile(level, tx, ty);
```

注意一致变量的类比：

```
Ptr<Uniform3f> v = new Uniform3f("myUniform");
v->set(vec3f(1.0, 0.0, 0.0));
```

然后我们渲染相应的四边形。在 GLSL 代码中，该块可以如下访问（见 “terrain1” 例子）：

```
#include "textureTile.glsl"
uniform samplerTile mySamplerTile;
void main() {
...
vec4 v = textureTile(mySamplerTile, uv);

}
```

其中四边形中的 uv 坐标必须在 0 到 1 之间（注意 textureTile 没有对块的边界采样，只是内部部分：[0…1] 范围映射到块的内部）。

注意：

- 如果 GPU 存储以 NEAREST 模式使用纹理，你仍可以在着色器中使用 textureTilelinear 代替 textureTile 做线性插值（该函数调用 textureTile 四次来插值得到结果）。

3.3.2 块映射

使用上文方法， samplerTile 一致变量只能在一个着色器中一次访问一个块。当然你可以为了同时访问多个块（在同一个存储或者不在）而在着色器中声明多个 samplerTile 。但仍是有限的选择一个固定的块的数量，绘制相关四边形，选择另一个块集合，绘制相关四边形，以此直到所有四边形。然而，有时需要同时访问一个（或多个）处理类的所有块。这可以使用块映射来实现：一个块映射是一个 GPU 中的间接结构，它指出每个块在块存储中的存储位置。由于一个存储可以储存多个处理类的块，所以你可以访问这些处理类的所有块。

使用

一个块映射通过 TileSampler 使用。但重要的是用来访问块映射的 TileSampler 在四边形被划分时不会请求处理类生成新块。换句话说它只是访问数据，但不生成。因此有必要使用一个“伴生” TileSampler, 没有块映射但是与使用相同块存储的块处理类相关，使得块能有效生成（实际上你可能需要多个这样的伴生取样器）。

一个法线 TileSampler 可以改变用来访问一个块映射：

- 第一步是在与 TileSampler 相关的 GPU 处理类中声明块映射，使用 tileMap=”true” 属性（见 GPU 块存储）
- “伴生”取样器相关的地形节点必须用 proland::TileSampler::addTerrain 声明

在着色器中使用 textureQuadtree 函数之前，该块映射可以通过调用 proland::TileSampler::setTileMap 方法使用（”terrain5” 例子中的 “terrainShader.glsl” 文件）：

```
#include "textureTile.glsl"
uniform samplerTile myTiles;
void main() {
...
vec4 v = textureQuadtree(myTiles, xy, 0.0);

}
```

该函数使用 x, y 物理坐标（在 -L/2 和 L/2 之间变化， L 是地形尺寸，即，根四边形边长 - 见处理类框架；第三个参数，这里是 0.0, 为处理类 id）作为输入参数。它首先找出包含该点的叶子四边形的逻辑坐标，然后使用块映射在存储中找到相应的块。最后在请求位置返回该块的内容。

算法

第一步找到包含物理坐标为 (x,y) 的点 p 的叶片四边形 q 的逻辑坐标 (level,tx,ty) ，假定大小为 L 的四叉树用划分距离因子 k>1 划分，视点为 (cx,cy) 。一旦知道了层数 level ，就很容易找到 tx, ty （实际上 $tx=|2^{level}(x/L+1/2)|$, ty 也一样）。所以主要问题就是计算 level 。

我们注意 p 和视点之间的距离 d=max(|x-cx|,|y-cy|) ，和四边形 q 与视点的距离（未知） d_q 。我们有 $d_q < d < d_q+L/2^{level}$ 。通过假设 q 没有被划分，那么：

$$kL/2^{level} < d_q < d$$

通过再假设 q 的父四边形被划分了，记作 r ，于是 $d_r < kL/2^{level-1}$ 。有 $d_r < d < d_r+L/2^{level-1}$ ，得到：

$$d < (k+1)L/2^{level-1}$$

那么我们考虑两种情况：$d < kL/2^{level-1}$ 或 $d > kL/2^{level-1}$ 。第一种情况我们由第一个关系 $kL/2^{level} < d < kL/2^{level-1}$ ，给出 $level = |1+\ln_2(kL/d)|$。第二种情况我们由第二个关系 $kL/2^{level-1} < d < (k+1)L/2^{level-1}$ 得到。在一些重写之后，给出 $1+\ln_2(kL/d) < level < 1+\ln_2(kL/d)+\ln_2(1+1/k) < 2+\ln_2(kL/d)$。总结以上，两种情况下， $1+\ln_2(kL/d) <= level <= 2+\ln_2(kL/d)$ 。所以我们这样计算 level ：首先计算 $l = |1+\ln_2(kL/d)|$ ，从中推出 tx 和 ty ，测试是否该四边形的距离 d_q 小于 $kL/2^l$ 。基于此结果，我们直到 level 要么是 l 要么是 l+1 。

一旦我们有了逻辑块坐标，第二步必须找到该块在块存储的位置。这是块映射的工作，为每个块在块存储中存储了快照（如果它存在于块存储中的话）。实际上对每一个潜在的块，该映射都没有一个入口：对一个 16 级的四叉树，将有超过 4^{16} 个潜在块，即，超过四十亿的入口！一个解决方案是在 GPU 中存储四叉树编码。但是查找一个块将请求遍历整个树。另一个解决方案是用 GPU 上的哈希表，但它很难避免冲突来保证最大效率。我们使用另一个解决方案，确保了固定时间访问（不遍历树，没有冲突，小的内存需求）。我们实际中在每一个四叉树层次都这样做，可以同时存在的叶子块的数量是有限且与层次独立的。

一个块 (l,tx,ty) 的父块如果没有划分，那它就不能存在。如果视点在 (cx,cy) ，包含视点的父块 (l-1, | 2 l-1 (cx/L+1/2) |, | 2 l-1 (cy/L+1/2) |) 被划分。l-1 层上距离小于 $kL/2^{l-1}$ 的所有块也要被划分。这个给出父块周围最多 k 个这样的块，即，l-1 层上最多 $(2|k|+1)^2$ 个块。因此层次 l 上同时最多有 $(4|k|+2)^2$个叶子块，无论 l 的值是多少。对 k<2 来说，这样给出一个 10^2.depth 大小的块映射，如，最大层次 16 有 1600 个入口（而不是四十亿个!）。

总的来说，CPU 通过存储每个叶子四边形的快照更新块映射纹理（在每个画面，取决于当前 cx, cy 的值），在索引层次上：

$$i = ix + iy.(4 | k | + 2) + l.(4 | k | + 2)^2$$

其中：

$$ix = tx - 2 | 2^{l-1}(cx/L+1/2) | + | k |$$
$$iy = ty - 2 | 2^{l-1}(cy/L+1/2) | + | k |$$

GPU 上，一旦物理坐标 (x,y) 相应的 (l,tx,ty) 坐标被找到，则计算索引 i ，索引中块映射的值从快照位置上取出，最终该快照上的纹理块被采样得到结果。 “terrain5” 中的 “terrainShader.glsl” 文件包含以上算法的具体实现。

3.3.3 纹理块取样器资源

proland::TileSampler 可以用 Ork 资源框架载入，有如下格式（”terrain1” 例子）：

```
<tileSampler sampler="mySamplerTile" producer="myProducer"
storeLeaf="true" storeParent="false" storeInvisible="false"/>
```

sampler 属性指定 GLSL samplerTile 一致变量的名字，该一致变量由 setTile 设置。 producer 属性是 GPU 块处理类资源的名字。 storeLeaf, storeParent 和 storeInvisible 属性是可选的，指定是否为一个给定四边形生成块（见上文）。使用 tileSamplerZ 代替 tileSampler 来创建 tileSampler 的子类，来将块数据读回 GPU，假设为高程块，使用该数据更新了地形四边形的 zmin 和 zmax 字段（见地形类），以及在 proland::TerrainNode::groundHeightAtCamera 中相机下的地形高度。

proland::TileSampler 以如下方式载入来访问一个块映射（”terrain5” 例子）：

```
<tileSampler sampler="mySamplerTile" producer="myProducer"
terrains="myTerrain1,myTerrain2,myTerrain3"/>
```

其中 terrain 属性间接通过地形节点资源指定“伴生” proland::TileSampler（最多可以指定 6 个地形）。

3.4 地形任务

提供了三个 ork::AbstractTask 子类来更新一个地形节点，更新一个纹理块取样器，最终绘制地形。

3.4.1 更新地形任务

proland::UpdateTerrainTask 在一个地形上简单的调用 proland::TerrainNode::update 方法。基于一个新的当前相机位置，更新地形四叉树。 proland::UpdateTerrainTask 可以由 Ork 资源框架创建，有以下格式i（ "helloworld" 例子有说明 ）：

```
<updateTerrain name="this.terrain"/>
```

name 属性指定必须被更新的地形节点，它有以下形式：

- name: 这时地形节点是名字为 name 的地形节点资源
- this.name, $v.name, flag.name：这时地形节点是目标场景节点 this, $v 或 flag （见方法）的 name 字段（见场景节点）的值

3.4.2 更新块取样器任务

proland::UpdateTileSamplersTask 在一个纹理块取样器集合上简单调用 proland::UniformSamplerTask::update 方法。为该任务上次执行后出现的新四边形生成块。proland::UpdateTileSamplersTask 可以使用 Ork 资源框架创建，有如下格式（ "terrain1" 例子有说明 ）：

```
<updateUniform name="this.terrain"/>
```

name 属性指定必须被更新的地形节点，它有以下形式：

- name: 这时地形节点是名字为 name 的地形节点资源
- this.name, $v.name, flag.name：这时地形节点是目标场景节点 this, $v 或 flag （见方法）的 name 字段（见场景节点）的值

该任务更新了与 Ork 方法执行的任务所属的场景节点有关的所有纹理块取样器。实际上一个场景节点可以有相关的一致变量，包括 proland::TileSampler （ ork::Uniform 的一个子类 ）。

3.4.3 绘制地形任务

proland::DrawTerrainTask 使用当前选择的程序为地形的每一个叶子四边形绘制了格网。通常格网是一个正方形格网，被 GLSL 程序变换，缩放，代替，在地形的合适位置绘制每个四边形。在绘制之前，该任务用 proland::Deformation::setUniforms 来设置一个一致变量用来变形

地形。也使用 proland::TileSampler::setTile 或 proland::TileSampler::setTileMap 设置一个一致变量来访问该四边形的块（对每个与 Ork 方法执行的任务所属的场景节点有关的所有纹理块取样器）。

proland::DrawTerrainTask 可以使用 Ork 资源框架创建，有如下格式：

```
<drawTerrain name="this.terrain" mesh="this.grid" culling="true"/>
```

name 属性指定必须被更新的地形节点，它有以下形式：

- name: 这时地形节点是名字为 name 的地形节点资源
- this.name, $v.name, flag.name：这时地形节点是目标场景节点 this, $v 或 flag （见方法）的 name 字段（见场景节点）的值

mesh 属性是一个格网资源的名字（见格网）。它指定了用于绘制每个叶子四边形的格网。有如下形式：

- name.mesh：这时格网是名字为 name.mesh 的格网资源
- this.name, $v.name, flag.name：这时格网是目标场景节点 this, $v 或 flag （见方法）的格网名字

culling 属性指定是否所有的叶子四边形都要被绘制，或者仅绘制那些在视锥中出现的。默认指示否，意味着所有的叶子四边形都被绘制。

4 用户接口

Proland 的整个接口都是基于事件句柄的。为了用户友好性和快速可用，它为在 Proland 中显示的大场景的导航提供了基础。

UI 分为两部分：事件句柄（如导航，编辑…）和 TweakBars，用于为一些选项提供可视化帮助。TweakBars 也是事件句柄，可以响应键盘，鼠标和 OpenGL 事件。

4.1 默认句柄

4.1.1 视野句柄

当使用如 Proland 这样的库时，一个友好的导航系统是必须的。 Proland 提供了一个称为 proland::BasicViewHandler 的系统。它的行为非常直接：当没有其他事件句柄捕捉鼠标键盘事件时，它使用它们来导航。默认导航系统为： PageUp 和 PageDown 来向前向后移动（z 轴）。鼠标左键沿 x y 轴移动相机，而右键使太阳绕着场景节点转动。 CTRL+单机旋转相机。鼠标滚轮和 PageUp PageDown 作用一样。

proland::BasicViewHandler 为了更好的工作请求一个 proland::BasicViewHandler::ViewManager。该视野管理器提供对 ork::SceneManager ，proland::TerrainViewController 和对屏幕向世界的转换的访问。基本视野事件直接在每幅画面中计算新位置，设置为地形视野控制器的相机位置。

proland::TerrainViewController 控制相机位置和方向。默认使用平面地形作为根节点实现，但是 Proland 提供了球体和柱体的实现。

基本视野句柄可以用 Ork 资源框架载入：

```
<basicViewHandler name="myViewHandler" viewManager="myWindow"
next="anOptionnalEventHandler"/>
```

- viewManager: 处理导航 UI 的 proland::BasicViewHandler::ViewManager 对象
- next: 一个可选的事件句柄，用来提取没被视野句柄捕捉到的事件

“terrain3” 实例说明了该视野句柄如何在没有 Ork 资源时使用（特别是当与 “terrain2” 比较时）。 “ocean1” 实例说明了如何在 Ork 资源下使用。

4.1.2 事件记录器

用户可能想要记录一系列动作然后在稍后重放，或者创建一个操作视频。proland::EventRecorder 就可以做这些。当按下 F12 时，它启动记录每个事件直到 F12 再次被按下（所有键盘，鼠标和 OpenGL 事件都会在它们发生时被记录）。那么，用户可以使用 F11 重放。当按下 Shift+F11 时，画面将以每秒 25 帧的速率被记录在硬盘上（使用每个事件的原始日期，而不是重放的时间，重放的时间会被花在保存画面的时间打乱）。这允许用户在之后创建视频。也可以保存和载入记录的事件。

事件记录器记录了 Recordable 对象提供的事件。它在事件句柄中像透明层一样工作，即，它代替了一个 UI 管理器的位置（例如视野句柄），记录所有事件，并传递到真实的 UI 管理器，除了播放一段视频时。

事件记录器可以用 Ork 资源框架载入：

```
<eventRecorder name="myEventRecorder" recorded="myWindow"
videoDirectory="\home\myVideos\" cursorTexture"cursor" next="myBasicViewHandler"/>
```

- recorded 该事件记录器可记录的资源
- videoDirectory 用于保存视频画面的文件名格式
- cursorTexture 重放时显示光标位置的光标纹理
- next 该事件记录器记录和重放的事件的句柄

4.2 TweakBars

除了控制器以外，UI 的图像部分也很重要，用户必须能够在不知道所有热键的情况下快速使用。以及，对开发者来说，信息必须清楚的显示和容易管理。 Philippe Decaudin 开发了一个具有以上优点的工具栏框架：AntTweakBars 。 Proland 工具栏基于此框架。

为了避免在屏幕上显示太多工具栏， Proland 包含了一个 proland::TweakBarManager ，可以从任何 poland::TweakBarHandler 添加内容，使用/禁止它们。当停用后，它们可以禁止与它们相联系的事件句柄，如果有的话（如编辑器）。 TweakBarHandlers 可以有三种类型：长期型（一直被激活），独有型（不能作为其他的独有句柄而同时可用）或标准型（可以打开或禁止）。

对事件记录器来说，TweakBarManager 是给定 UI 管理器上的透明层。它也是一个事件句柄，因此可以捕捉事件并传递到它们的 TweakBarHandlers 。那些将决定数据显示后是否有变化；如果有，管理器将使用更新后内容重新创建 tweakbar 。

TweakBarManager 可以用 Ork 资源管理框架载入：

```
<tweakBarManager name="myTweakBarManager" minimized="false" next="myViewManager">
<editor id="myEditor1" bar="myTweakBarEditor"
exclusive="true" permanent="false" key="r"/>
<\tweakBarManager>
```

- minimized：决定 TweakBarManager 是否以最小化启动
- next：必须处理未使用事件的句柄
- bar：一个将它的内容添加到 TweakBarManager 的 TweakBarHandler
- exclusive：决定是否 TweakBarHandler 是专有的。同一时间只有一个句柄可以被激活
- permanent：决定 TweakBarHandler 是否可用
- key：一个可选的热键来启用/禁止 TweakBarHandler

一些默认可用的 TweakBars：

- proland::TweakResource：一个直接用 xml 文件描述的 TweakBar，像其他 Ork 资源一样
- proland::TweakSceneGraph：用于控制场景图。使用 proland::SceneVisitor 对象浏览场景图并在场景中显示每个节点。它允许用户启用/禁止几乎任何节点
- proland::TweakViewHandler：控制一个基本视野句柄。包含一次点击访问的预定义位置 i。也可以在当前位置上显示。

Proland 实例说明了这些 tweak bars 是如何使用的（特别是 “edit1”, “edit2”, “edit3” 和 “edit4” ）。下图显示了 proland::TweakSceneGraph 接口：tweak bar 给出一个树形表现的场景图，每个场景节点都可以扩展（允许显示/隐藏该节点，查看它的生产者，使它的块无效等）。这个工具栏也给出了一个可以显示的内容纹理的列表（右边我们看到正交处理类块缓存使用的纹理）。最后它还显示了关于块缓存的统计（容量，使用中或未使用的块的数量等）。

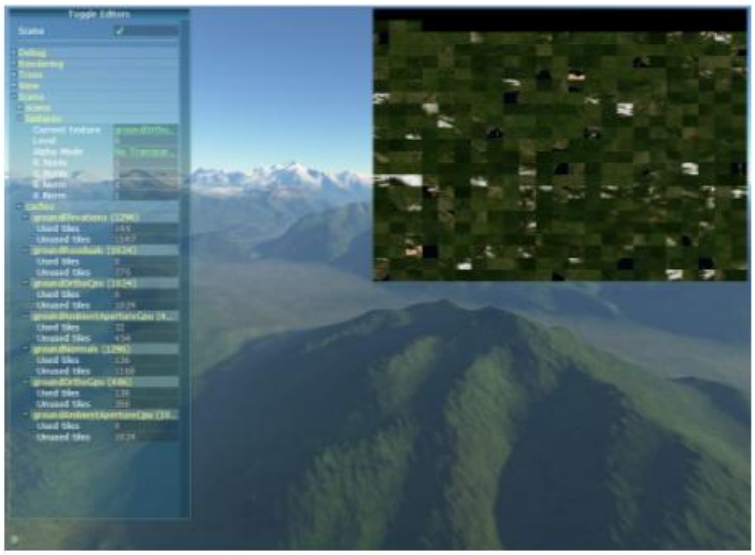

图16：proland::TweakSceneGraph tweak bar 界面。可以用鼠标移动和缩放窗口，用 SHIFT+鼠标拖拽移动内容。用 CTRL 鼠标拖拽放大，用鼠标滚轮放大缩小。右击恢复视窗设置