# Proland 文档 - 河流插件

> sprites，原意为小精灵，此处理解为附在粒子上的构成波动的小碎块，统一翻译为 小图块

河流插件基于论文 “Scalable Real-Time Animation of Rivers”, Qizhi Yu, Fabrice Neyret, Eric Bruneton and Nicolas Holzschuch, Eurographics 2009. 它基于图形和粒子系统实时渲染动态河流。*该插件需要地形和图形插件。*

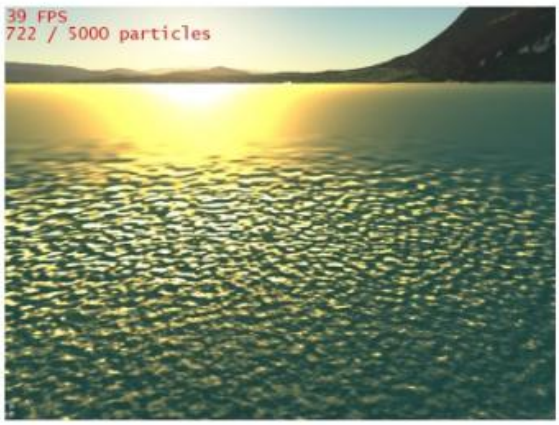


**图1：一条河流范例**

我们在这里重温以上算法的主要原理（细节见论文）：

- 河流表面的渲染用一个小图块集合完成。每个小图块用一个小的**波块**贴图来表现河流表面特征。这些小图块在渲染中被融合到一起得到一个起伏的连续河流表面。
- 河流表面的动作来自两件事。其一，波块纹理是生动的。因此表面波动是生动的即使河流没有流动（例如湖面）。其二，最重要的，小图块被附在由河流输送的**粒子**上。这意味着每个小图块都在一个粒子中心，而粒子的位置基于河流的流速场在每个画面中都被更新。

该算法的可量测性来自两个重要的特性：

- 第一，粒子没有覆盖整个河流表面，而只是给定时间内可见的部分。而且离相机远的地方比近处用的粒子更少更大。这保证了粒子的数量（由于渲染由粒子数直接影响，所以也直接影响了表现效果）是有限的，即使河流网络非常大。这些属性由管理几乎整个**屏幕空间**的粒子保证。特别是，一个粒子处理类在屏幕空间保持粒子常量的大小和密度。这自动保证只有可见的部分被覆盖，为远处部分使用更大的粒子（在世界空间）覆盖。世界空间计算仅用于粒子的输送。
- 第二个特性是给出一个可扩展算法用于计算河流流速场（自己用于世界空间粒子的输送）。没有使用计算流体力学算法，那样需要模拟整个河流网络，Qizhi Yu 的算法使用了程序化流速场，可以在每个点上独立计算，仅基于该点到最近的**河岸**（世界空间内一个有

限半径的圆盘）的距离，以及沿着河岸的流函数（流函数也称为 势，因为流速场是从该方程的导数计算得到）的值。

proland 模块被划分为几个部分，对应于以上算法的逻辑步骤。从算法的输入到输出，这些部分如下（接下来的章节会介绍更多细节）：

- 图形框架提供类来描述算法的输入，称为河流网络。该网络是由描述河岸的曲线组成。图形框架可以描述图形和曲线，但是没有河流算法需要的每条曲线的流函数的值，也称为势。为此，图形模块提供了一个图形框架的扩展，为描述河流网络给曲线添加了必要的数据。
- proland::HydroFlowProducer 和 proland::HydroFlowTile 类实施了程序化流速场的算法，使用描述为河流图的河流网络做输入。
- proland::particles 框架提供了类来管理屏幕空间中的粒子，特别是保持一个不变的粒子密度。该框架的类也计算新粒子在世界空间的初始化位置。这些初始化位置由 proland::HydroFlowProducer 计算的流速场输送。
- 最后，河流的保留类在前一个粒子以及生动的波动块纹理辅助下来绘制和融合小图块，为了渲染一个生动的河流表面。

这四部分及其联系由下图 UML 图说明：

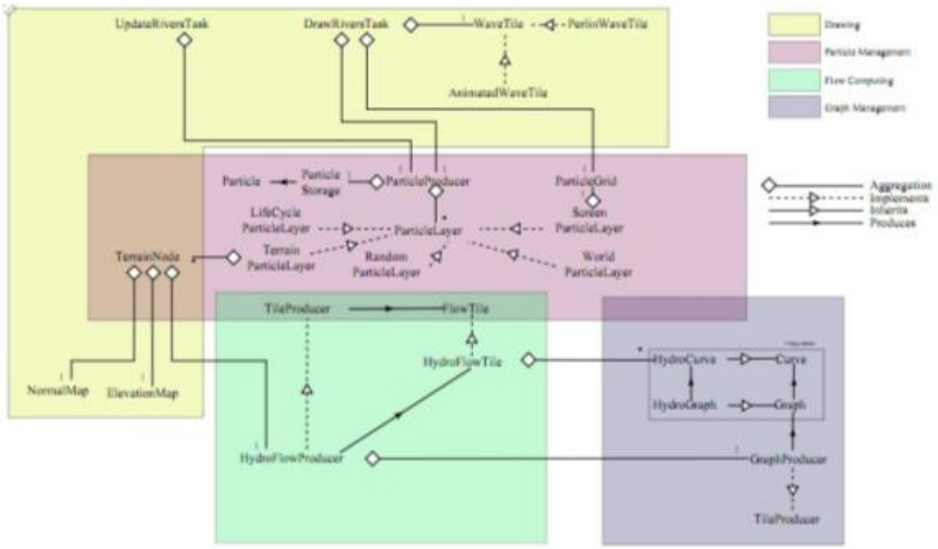


**图2：用于河流的 UML 图**

用于输入的河流网络称为水文图。**特别注意我们这里水文图的表现和 Qizhi Yu 的文章不是完全一致。**在该文章中，只有河岸和障碍被清楚的表现。在我们的表达里，我们也需要一个清楚的**河流轴线**的表达：其必须为一个相应河流中心轴线的曲线，宽度必须足够大以覆盖河岸。另一个不同是我们需要每条河岸都有相应河流轴线的一个引用（除了在该河岸上的流函数，如 Qizhi Yu 的文章一样）。

河流轴线曲线对 proland::WaterElevationLayer 是必需的（为了得到正确的河床要扁平化地形）。在河流模型中快速检查是否一个粒子在河流内部也是很有用的（见下节）。河岸和轴线之间的引用需要在程序化流速场算法中实施（见下节）。

下一个图显示了水文图是如何组织的。

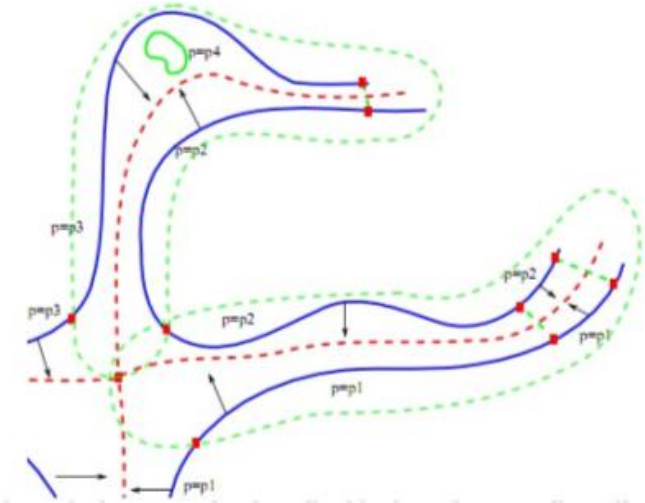


**图3：一个水文图。每个河流必须由它的轴线曲线（红色）和河岸（蓝色）描述。河流轴线曲线的宽度必须覆盖相应的河岸（该宽度这里用绿色虚线表示）。河岸曲线必须通过一个非河岸曲线链接（绿色块）形成河面。它们必须包含一个流函数值（作为势）和一个到它们相关轴线曲线的链接。只有一种可能，如果河岸曲线被河流汇合处的节点（红色点）断开，除此以外，与河岸曲线相连的节点必须具有同样的势值（避免不连续）。最后，一个河流图必须包含河流内部的障碍曲线来定义岛（这些曲线必须有一个势值，但它们不需要与河流轴线有链接 - 尽管它们必须被该轴线曲线覆盖）**

## 实现

图形框架不足以描述河流网络。因此扩展的河流图用来描述这些网络，称为水文图。

水文图由 proland::HydroGraph 和 proland::LazyHydroGraph 表现。分别是 proland::BasicGraph 和 proland::LazyGraph 的子类。与经典图形的主要不同是我们需要每个河岸多两个信息：一个流势和对轴线曲线的引用。这些新信息存储在 proland::HydroCurve （和 proland::LazyHydroCurve ）。

那么，proland::HydroGraph 扩展了 proland::BasicGraph 仅是创建水文图而不是基本图，从文件加载和保存每个水文曲线的附加数据。这些文件中每个曲线都必须有 5 个参数，而不是基本图中的 3 个。这两个新参数必须跟在默认的 3 个参数（大小，宽度和类型）之后：

- 顶点数（包含起始和结束点 - 整型）
- 宽度（浮点型）
- 类型（整型）

- **势值（浮点型）**
- **对应河流轴线的标识符（如果该曲线不是河岸则为 -1 ）（整型）**

注意这些文件与其他图形兼容。如果河流图用做标准图，那河流数据会被忽略。河流算法主要基于曲线类型，所以需要对每条曲线正确设置该值： HydroCurve::AXIS 值可以用于你想显示的河流轴线。一般来说，最好把河流轴线曲线的类型放入 HydroCurve::CLOSING_SEGMENT 中。该类型用于非显示曲线，可以链接上。它们可能在河流结尾使用闭合河岸，为了形成一个循环。最后， HydroCurve::BANK 用来创建河岸或障碍（岛，水中的移动物体等）。该类型的曲线必须定一个势值。

## 算法

下图再现了 Qizhu Yu 的算法原理，以程序化方法计算河流内部的流速场（细节见文章）：

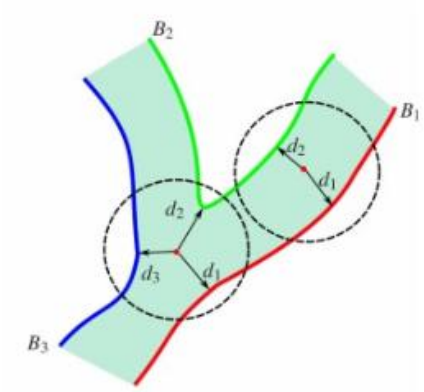


**图4：给定点的流函数是用附近河岸的流函数值插值计算得来，使用了该点到每个河岸的距离。只有该点周围有限半径的环形搜索区域里的河岸才被考虑。该点的流速场由该插值流函数的导数计算得来（这需要计算点周围四个插值来计算有限差的导数）**

**我们的算法实现和 Qizhu Yu 文章的描述有一点不同。**实际上我们制造使用了一个与河岸曲线相关的附加数据：与相应河流轴线曲线的链接（见上文）。

在 Qizhu Yu 的文章中，基本算法的改进是通过一个**加速数据结构**来快速找到给定点有限距离内的河岸来实现的。为了实时编辑，不会一次为所有河流图计算该数据结构。而是为块处理类框架生成的每个河流图块在运行时计算（即，当新块出现在视野时）。河流图每次被编辑都重新计算。

该加速数据结构包含一个河流图的简化版本，只引用了那些足够大在屏幕上显示的曲线（基于块大小参数，类似于正交处理类（应该是一样的，但可能用户显示的细节量大小而改变）。

这些曲线被分割成线段存储在覆盖整个块的格网中。注意每个河岸曲线总是和它相应的河流轴线在同一个格网单元里，反过来也一样。这可以快速提取给定点到每个河岸的最近距离。我们在每个线段周围添加了一个半径因子，然后将线段添加到半径中的单元里去。该半径取决于块中最大的河流。

当计算给定点的流速场时，我们必须首先检查是否该点在河流中。如果在，该点的流速场将由该点周围四个点的插值结果的有限差导数计算得到。

决定一个点是否在河流中，我们浏览存储在该点对应的格网单元的每个河流轴线。对每个河流轴线，我们只需要知道该点到最近曲线线段的距离。如果大于河流半径，我们就从势值计算中排除该河流。对每个留下的河流，我们得到对应的河岸，计算与其之间的单向距离。如果单向距离是负值，意味着该点在河岸外。如果一个河流轴线没有至少两个有正距离的对应河岸，该点认为在河流之外，河岸也被忽略。如果存在，结果值是一个保留下的河岸势值的插值。

河流轴线和河岸之间的链接用于确保我们不会使用不需要的河岸或河流来插值，这会是算法变慢并且产生异常值。

最后，为提高哪怕一点计算速度，我们添加一个数组来存储块中计算过流速场的像素。所以当需要一个临近点的流速场时，我们可以重复使用。因为流速场是四个值的导数，我们总是检查是否在计算前就已经具有这些。实际上，如果其中之一是负值，点在河流外，因此我们不需要重做这整个过程。格网的大小取决于 HydroFlowProducer 类的 potentialDelta 参数，以及当前块的层次。potentialDelta 是一个应用于当前块大小来决定缓存大小的比例。它最多和 displayTileSize 参数一样大，即，显示的块（和前面一样，应该和正交处理类的 tileSize 参数相等）。该参数用于仅需要一些细节的时候（远离视点），并且不需要像相机贴近地面时的那么多内存。这被用来显示层次以避免流速场的不连续（插值距离不一样，因此在层次 n 上，将显示一个两倍于层次 n-1 的速度）。默认值是 0.01 。同样的标准用于决定是否需要创建一个新的流块：在一些点上，两个连续的层只包含同样多的细节量，那么我们会用掉不需要的内存。我们可以决定出现这样的层并只使用需要的父块。

## 实现

以上算法和加速数据结构用于提高在 proland::HydroFlowTile 类中的计算速度。该类的一个实例包含了相应地形块的裁剪图形的加速数据结构（该剪裁图形没有存储在水文流块中而是水文图的 proland::GraphProducer 相关的块缓存中）。它的角色是测试是否给定点在河流内部，如果是，就计算该点的流速场。

这些水文流块由 proland::HydroFlowProducer 从 proland::GraphProducer 生成的水文图块生成（假设生成图的曲线是 proland::HydroCurve 的实例）。

**注意：**

- 除了继承于 proland::TileProducer ，proland::HydroFlowProducer 也是一个 proland::CurveDataFactory 。计算河流流速场时不需要，但渲染步骤需要（见下节）。

proland::HydroFlowProducer 可以由 Ork 资源框架载入，有以下格式：

```
<hydroFlowProducer name="myFlow1" cache="anObjectTileCache" graphs="myRiverGraph"
displayTileSize="192" slip="0.9" searchRadiusFactor="20" potentialDelta="0.01"/>
```

- cache 属性是存储该处理类生成的块的块缓存。链接到一个对象块存储

- graphs 属性是用于创建水文图的图形处理类。如果提供的图形不是水文图或懒水文图（或导出），出现一个断言
- displayTileSize 属性是显示的块的大小。应该与正交 GPU 块一样
- slip 属性决定在边界上的滑落条件（细节见 Qizhu Yu 的文章）
- searchRadiusFactor 属性决定为给定点寻找用于插值的河岸的搜索半径
- potentialDelta 属性决定用于插值流速场的点的位置，以及流速场缓存的大小（见前节）

为了能够取代 Qizhu Yu 的程序化流速场算法，可能不能基于图形，我们提供了一个抽象接口 proland::particles::FlowTile ，用于实现 proland::HydroFlowTile 类。该接口的主要方法是 getType 和 getVelocity ，都是用一个 2D 坐标为输入参数，并生成一个**数据类型**和一个流速（只有第二种方法）。粒子处理类（见下节）使用该接口代替具体的 proland::HydroFlowTile 实现，为了用其他实现取代它。

为了表现流体系统的运动，一个知名的技术就是使用粒子。粒子实际上表现为一个使用单一位置对象（ 2D 或 3D ，取决于我们需要什么），以及若干属性来决定它的行为。一般来讲，我们需要创建，更新，当不需要时删除。它们可以用来处理大量的类似对象。

如前所述，在 Qizhu Yu 的算法中，粒子首先在屏幕空间生成然后输送到世界空间。它们的屏幕坐标必须更新到对应的世界坐标。粒子的生成基于屏幕空间上的泊淞距离分布，通过增加或移除粒子，该分布必须在每个画面中检查。粒子也有最大生命长度，加上一个渐入和渐出的状态，避免当它们消失时产生爆音效应。

算法如下（每个画面都调用）：

- 世界空间粒子的输送：依赖我们有的河流更新粒子的世界坐标
- 计算新的屏幕空间坐标
- 检查是否一些粒子必须销毁，要么离得太紧要么太旧
- 如果泊淞距离分布中有洞则添加新粒子
- 计算新添加粒子的世界坐标

## 实现

Proland 提供了使用粒子所需的所有工具：

首先，我们有一个 proland::ParticleStorage ， proland::ParticleStorage::Particle 缓存可以创建和删除粒子，不需要调用新建/删除方法（它在程序开始时回收内存，该程序取决于粒子包含的内容）。该粒子存储由 proland::ParticleProducer 使用，然后处理该粒子。有一个 proland::ParticleLayer 的列表，每个为一个给定任务专用，为其调用以下三个方法：

- proland::ParticleLayer::moveParticles：在每个空间更新粒子的坐标
- proland::ParticleLayer::removeOldParticles：由于每个粒子有一个寿命，假设在某一点上死去。这时，当在每层上发生时该方法允许添加指定的行为
- proland::ParticleLayer::addNewParticles：用于新的添加的粒子

每个图层添加它需要的内容到粒子上。如此，就要包含一个这些元素需要的结构。粒子处理类将决定用 proland::ParticleLayer::getParticleSize 方法为每个粒子请求的内存大小。每个图层都能将一个粒子转换为包含特别数据的特别粒子类。

提供了一些预定义图层：

- LifeCycleParticleLayer：处理粒子生命循环：它存储每个粒子的产生事件，然后可以决定它们的强度。也可以决定是否一个粒子在产生时渐进或渐出
- ScreenParticleLayer：以泊淞距离分布在屏幕空间创建粒子。对每个粒子来说，它也包括它们的屏幕空间坐标和一个原因参数，用来决定为什么屏幕粒子层杀死这个粒子（仅用于该粒子当前正渐出时）。它也用场景管理器作为参考，将粒子坐标从屏幕空间转换到世界空间。为了存储和快速访问粒子，屏幕粒子层使用一个 proland::ParticleGrid 类。窗口被划分为正方形格网，格网的每个单元包含覆盖它的粒子。这使得能够快速获取一个给定粒子的邻居，也能加速 GPU 对粒子的渲染（见下节）
- WorldParticleLayer：为每个粒子添加世界空间 3D 坐标以及世界流速。在每个移动粒子调用时，更新依赖于它们的流速数据内容的粒子的坐标。它也有一个暂停状态，可以暂停粒子更新
- TerrainParticleLayer：该图层存储了场景节点列表，能够决定一个粒子在哪个上面。它也存储了地形空间坐标和一个粒子状态。在移动粒子调用时，它更新依赖于地形节点创建的流块的内容的粒子坐标。（地形节点必须有一个生成流块的块处理类）。它也更新移动粒子的世界流速
- RandomParticleLayer：对每个粒子来说，只包含一个给定界限里的随机 3D 向量

粒子也与 GLSL 着色器兼容：粒子处理类能够转换粒子数据到格式化可用的到 GPU glsl 着色器。这通过 ParticleProducer#copyToTexture 方法完成。该方法使用 ork::TextureRectangle 做参数并且回调函数将决定在纹理中设置哪些内容。结果包含在提供的纹理中。粒子在粒子存储内存空间中按它们的位置被排序。

proland::ParticleGrid 也能转换它的数据到一个 GPU glsl 着色器可用的格式化数据，也通过它的 copyToTexture 方法。这个复制了每个粒子在纹理中的的索引作为输入。每个单元的最大粒子数量可以被决定，因为为每个像素浏览太多粒子会非常耗时间。该纹理作为一个间接表工作，并且它将公平简单的提取覆盖像素所需的粒子。

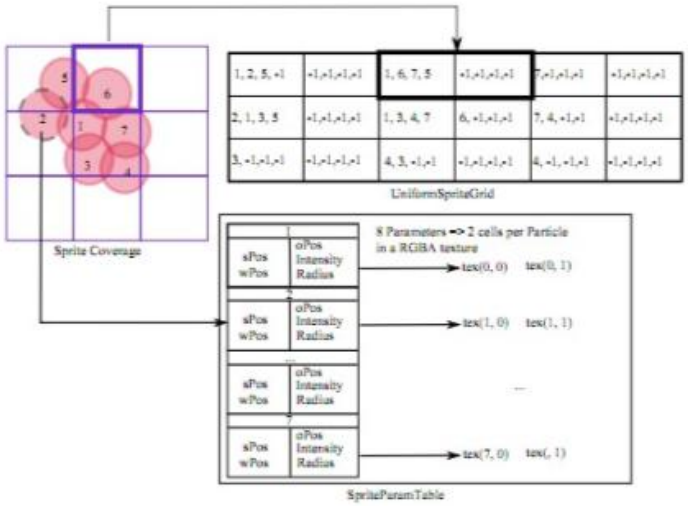


**图5：用于传递粒子到 GPU 的结构。粒子格网设置为每个格网单元处理 8 个粒子，使用 RGBA 纹理；这是为什么每个单元对应 2 个纹理坐标。如果处理的粒子数太高，将失去很多性能。如果太低，格网边缘将不连续。 -1 值指示空纹理值**

粒子框架以 Ork 资源框架载入：

首先，我们需要一个粒子存储来储存粒子。

```
<particleStorage name="particleStorage1" capacity="5000" pack="true"/>
```

可用的参数有：

- capacity：该存储可以创建的最大粒子数
- pack：决定粒子在内存空间中如何组织。如果为真，创建和删除粒子会使用较长时间，但用于访问它们的时间会减少而且每个粒子在内存中都是相接的

然后，我们创建粒子处理类和其中的图层：

```
<particleProducer name="particles1" storage="particleStorage1">
<terrainParticleLayer name="terrainLayer"
terrains="terrainNode1/riverFlow,terrainNode2/riverFlow"/>
<worldParticleLayer name="worldLayer" speedFactor="1.0"/>
<screenParticleLayer name="screenLayer" radius="30"/>
<lifeCycleParticleLayer name="lifeLayer" fadeInDelay="5" fadeOutDelay="0.5"
activeDelay="30" unit="s"/>
<randomParticleLayer name="randomLayer" xmin="-1" ymin="-1" zmin="-1"/>
</particleProducer>
```

粒子处理类只有一个参数：storage 。那么，它有一个对应于它的粒子层的无限制的孩子节点数。一些举例如下：

- 地形粒子层：它的输入参数是一个如此组织的 terrain 列表：首先，它需要包含生成流块的块处理类的地形节点。然后，以斜杠做分隔，它读取地形节点中块处理类的名字。如果场景包含多个地形，它们必须用逗号分隔
- 世界粒子层：包含世界空间中每一个粒子位移的 speedFactor
- 屏幕粒子层：在屏幕空间中，需要每一个生成的粒子的 radius
- 生命循环粒子层：生命循环延迟可以指定： fadeInDelay 和 fadeOutDelay 用于融合进/出粒子。 activeDelay 参数定义粒子的寿命。这些参数的单位默认是秒，但是可以通过 unit 参数修改（可以是 s 秒，ms 毫秒，us 微秒）
- 随机粒子层：用户可以指定随机生成坐标的 xmin, xmax, ymin, ymax, zmin 和 zmax 边界值。默认每个维度都是 0-1

现在我们来看渲染部分。

我们现在有了覆盖全屏幕空间的粒子，并能够输送它们。每个粒子与一个包含纹理的小图块相关。该纹理实际上是一个包含波浪状法线的大纹理的一个随机部分。为了使河流不那么静止，尤其是湖面，没有水流，该纹理必须通过时间更新。

那么我们只需要有效的融合并渲染这些小图块。为此，我们依靠前节介绍的间接格网（粒子格网内容）。那些格网只包含覆盖给定单元的粒子的索引，为避免冗余。每个粒子的参数都存储在不同的纹理中（见前节粒子处理类）。这些参数如下：

- 屏幕空间坐标
- 世界空间坐标
- 随机纹理坐标
- 粒子强度
- 屏幕空间中的粒子半径

算法如下：对每个像素，我们访问间接格网来查找所有覆盖该像素的小图块。对每个小图块，我们读取从参数纹理读取它的参数。然后我们计算每个小图块的贡献度并依据距离，强度等融合结果。

用于决定一个小图块的贡献度公式如下：

$$S_i(x) = T(x - p_i + u_i)$$

其中 i 是小图块的索引， $p_i$ 是小图块在世界空间中的坐标， $u_i$ 是一个随机纹理坐标，在整个小图块生命周期内是常量。

计算最终结果的公式如下：

$$F(x) = \sum_i (w_i \cdot S_i(x)) / \sum_i (w_i)$$

权重因子 $w_i$ 取决于到小图块中心的距离和小图块的强度。因为强度取决于粒子的生命循环（使它们渐进渐出），当小图块被删除是不会得到爆音效应。

那么我们可以应用普通着色器：凹凸映射，折射和反射，阴影...

另一点，我们以及注意到如何以一个高效的方式绘制河流：实际上，我们不需要从每一个可见块中绘制所有图形，像图形处理类一样：这花费太多时间在每一帧的重绘上（即使再小的图行，当放大时都有超过几千个顶点，特别是扁平化图形）。取而代之的是，我们决定浏览每一个显示的块和存储显示的曲线。我们也清理那些太小而不需要画的曲线（屏幕上宽度小于 1 个像素）。那么，我们浏览包含那些曲线的每个区域，作为 Tesselator 的输入参数，直接计算结果格网。最终，我们存储这些格网所以不需要在每一帧重复计算，但除过调用绘制方法时。那么，对于每一个顶点，我们访问了决定地面高程的高程表，所以河流可以沿着地形绘制。

不过，这个方法有一些不利方面，产生一些伪假象，如果高程不是常量，如果河流太大：Tesselator 可以创建非常长又细的三角形，如果高程与三角形的一个点和另一个点差别很大，这将在下一个图像中产生可见的缺陷。在地形上添加一个高程图层可以帮助较少这种效应。

# 实现

在 Ork 资源框架中，河流简单说就是一个节点。它有“对象”和“动态”标识符，为了使用框架能力在实际执行任务之前为每个画面去建立任务图。

在我们的情况中，更新方法调用使用了 proland::UpdateRiversTask 任务类，一个主张所有需要的资源都可用的任务。绘制方法调用使用了 proland::DrawRiversTask 任务类，更新粒子并调用纹理输送方法来绘制河流。注意只有绘制部分在 GPU 完成。

在每个画面，更新河流任务检查是否有新块显示。如果是，处理类被请求释放那些不再可见的块。它也检查是否自上一个画面块被改变，这时如果有就必须更新（任务图也一样）。它处理流处理类和用于给定粒子处理类的河流图形处理类。它也强制为法线和高程块生成一个块映像。该块映像将允许我们直接在 GPU 中在给定地形空间坐标中快速获取高程和法线，而不需要从 CPU 中抓取。块映像在地形框架章节中介绍。

绘制河流任务使用来自每个可见块的数据绘制河流。首先，它更新粒子（ParticleProducer#updateParticles ），以及 gpu 结构。然后，取决于一些内部参数，它将显示：

- 河流，可用的不同显示类型有：
  - 不显示
  - 只显示格网
  - 输送法线
  - 非输送法线（应用一个简单纹理）
  - 粒子覆盖层
  - sprite格网显示
- 粒子
- 流速
- 太阳效应，由海洋框架提供
- 雾效应（为减少光晕效应，不管明细贴图是否这里仍需要）

为了绘制河流，我们使用了图形图层中一样的方法：大河（我们想绘制的）用面表示。如前所述，我们不绘制每条曲线，而且最多只绘制一条曲线一次。对每个场景节点，我们应用上文的方法，只在新块出现在屏幕上时重新计算它，或者当一个块更新时（要么是流体要么是图形）。这使我们可以容易的实时编辑河流。

为了绘制河流，我们多数时候使用凹凸映射的方法，使用上文介绍的融合函数的结果（F 函数）。我们将函数结果看作新的法线，在凹凸映射部分使用它。

波动纹理称为 proland::WaveTile 对象。它们包含了一个 2D 纹理，用来通过时间 timeStep 方法更新。

提供了 3 种实现：

- WaveTile：第一个仅使用一个 2D 纹理做参数，不需要修改。在整个执行期间它都保持一致
- PerlinWaveTile：该波动块在启动时生成一个单一柏林噪声纹理，像 WaveTile 一样使用
- AnimatedPerlinWaveTile：创建了 N 个 PerlinWaveTile 并有规律在它们中间做修改。这创建了一个流动的感觉即使粒子并没有移动

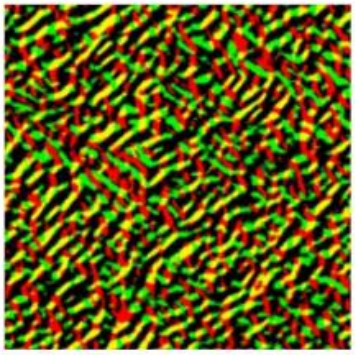

**图6：一个包含像波动一样的法线纹理片段的示例**

所有这些对象都可以用 Ork 资源框架载入：

- WaveTile：

```
<animatedWaveTile name="myWaveTile1" texture="mytexture" tileSize=1
gridSize=256 waveScale=1.0 timeLoop=32/>
```

有一些默认值。以下参数对 3 个已有的 WaveTile 对象通用，但它们不是都使用基本类。

```
- tex 包含波动剖面的 2D 纹理。仅用于 WavaTile
- gridSize 纹理的大小
- tileSize 块的大小
- waveScale 波的大小
- timeLoop 一个波循环的帧数
```

- UpdateRiversTask：

```
<updateRiversTask name="myUpdateTask" particles="myRiverParticleManager"
timeStep=1/>
```

```
- particles 一个粒子处理类
- timeStep 每个画面的时间步长。改变河流的速度
```

- DrawRiversTask:

```
<drawRiversTask name="myDrawRiversTask" particles="myRiverParticleManager"
drawParticles="true" particleRadius=0.01 timeStep=1 texture="myWaveTile1"
fogStart=0 fogEnd=5/>
```

有一些默认值：

```
- particles 用于创建流的粒子处理类
- timeStep 每个画面的时间步长。改变河流的速度
- drawGrid 决定是否绘制屏幕小图块格网，主要用于调试
- drawParticles 决定是否绘制粒子，粒子绘制为彩色的点
- texture 用于输送法线的纹理
- fogStart 和 fogEnd 烟雾距离（x：近，y：远）
```