Proland 文档 - 图形插件

> flattend 应指将图形平滑，拐角圆润，此处翻译为 扁平化
> 图11 网页打不开

1 简介

Proland 图形插件提供一些矢量数据专用的处理类，以及矢量数据栅格化。*该插件依赖地形插件*。

1.1 图形处理类

一个**图形**包含点，曲线和面组成的 2D 矢量数据。矢量数据可以用来表现线性的地貌如道路和河流，以及面状地貌如森林区域，城市区域等。它用于在正射影像上绘制这些地貌，为了在地形中正确插入这些地貌而修改高程，驱动地形上对象的过程演化（如沿道路种树）等。矢量数据的主要优点是其与分辨率无关，不像栅格数据。因此你可以用非常少的数据得到任何尺度上的高精度线性特征。

在 Proland 中，图形主要在**图层**中使用，它可以在任何基于图形内容的处理类上绘制数据。见**基于图层**的正射图形或**基于图层**的高程图形。

进一步讲，一个图形由节点，曲线和面组成：

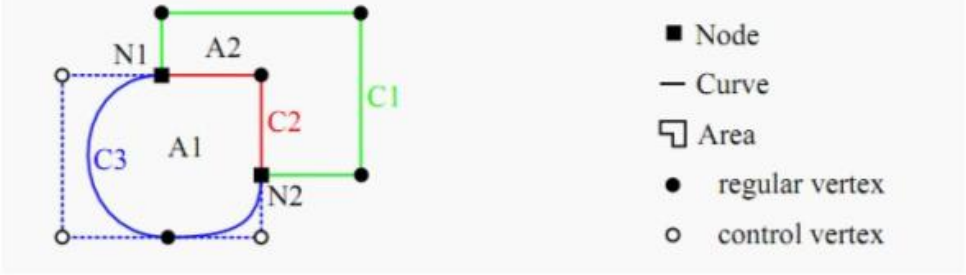

图1：一个图形由节点，曲线和面组成

- 一个节点是一个简单的 2D 点
- 一个曲线是通过一系列顶点链接一个起始节点和一个结束节点
- 一个面是一个曲线的集合，以逆时针方向形成一个循环。一个面可以递归地包含另一个图形，称为子图形

一个图形如此定义是因为它存储了这些元素之间的联系。一个节点存储了连接到它的曲线集合（上例中， N1 和 N2 都引用了 C1, C2 和 C3 ）。一个曲线在它的两端引用了两个节点，并且引用了包含该曲线的面（最多2个，曲线两侧一侧一个 - 例如 C2 被 A1 和 A2 引用，而 C3 仅被 A1 引用）。最后，一个面就是简单的曲线引用和它们方向的一个列表。

一个节点仅定义为两个 (x, y) 坐标。一个曲线定义为它的起始和结束节点以及通过的顶点。它也有类型和宽度，依你所需可以随意表达。顶点可以是*端点*可以是*控制点*。仅由端点组成的曲线就是一条折线（上例中的 C2）。但你可以在两个端点之间插入一个或（最多）两个控制点，来定义一条二次或三次平滑的贝塞尔补丁（上例中 C3 是由一个二次和一个三次贝塞尔补丁组成）。注意端点位于曲线上，但控制点不是（它们在端点上定义切线向量）。连接顶点的折线（上例中的虚线）被称为该曲线的控制曲线。

一个图形可以用添加或移除节点，顶点，曲线或面，移动节点或顶点等方式任意编辑。

一个图形也可以被裁剪，通过移除落在裁减区域的元素得到指定的矩形面。这里"元素"指曲线的线性，二次或三次贝塞尔补丁。意思是曲线仅能在这些补丁之间的端点上裁剪（我们没有精确计算曲线和裁剪区域的交集来提高性能）。裁剪后的曲线得到若干没有联系的小曲线。相反，裁剪后的面得到最多一个面（结构化）。为了得到闭合面我们有时需要引进新的直线来闭合裁减区域。

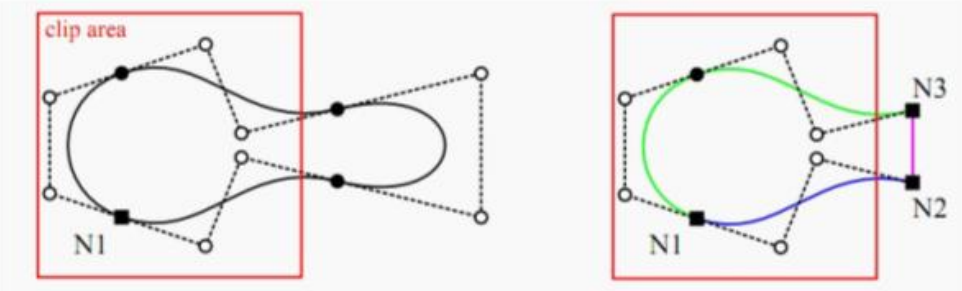

图2：一旦裁剪，左图中形成一个环的闭合曲线会给出新的节点（ N2 和 N3 ）和两条裁剪曲线（右图中蓝色和绿色）。注意曲线没有被裁减区域精确剪裁，而是在端点处剪裁。为了闭合裁减区域，N2 和 N3之间增加了一条直线。为了避免带有直线的裁减区域的转角问题，练习中裁剪分两步：首先是水平片，其次是垂直片

为了在裁剪中考虑曲线宽度，裁剪区域可以使用一个边界系数自动扩展来避免曲线的中心线在裁减区域之外，而不是边界：

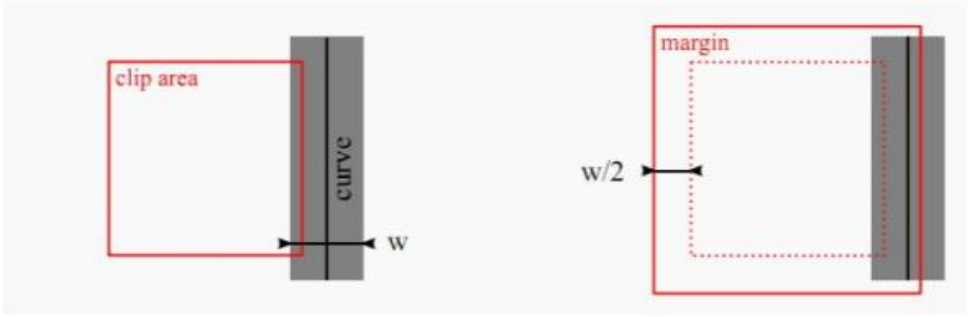

图3：边界系数避免了曲线的中心线在裁减区域之外，而不是边界。没有边界系数（左图）曲线被忽略了，这是不正确的。有一个等于曲线宽度一半的边界系数，该曲线才被捕捉（右图）

最后一个图形可以通过在曲线中引入新的顶点而扁平化，使得它们的控制曲线向平滑曲线会合。这在绘制一个更简单的控制折线代替曲线时很有用。如果控制折线足够接近平滑曲线，结果看起来就几乎一样。新的顶点由迭代贝塞尔曲线划分算法计算得到，直到曲线和其控制折线之间的距离最大值小于指定阈值：

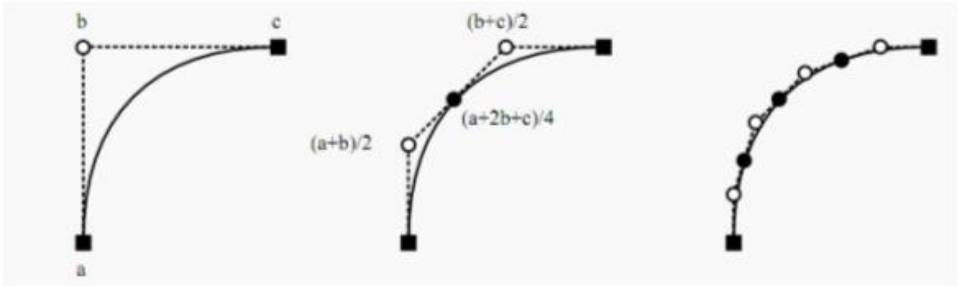

图4：通过旧顶点计算得到的新顶点来替换，直到曲线（实线）和其控制折线（虚线）之间的距离最大值小于指定阈值，一个图形逐步被扁平化

此过程中，我们沿曲线为每一个顶点计算一个伪曲线坐标，记为 *s*。初始这些伪曲线坐标为该曲线中的顶点索引。扁平化过程中，就顶点本身而言，使用相同的公式计算其新顶点的分坐标。如果需要并且即使没有足够扁平化（通过二分法查找，和"完全"扁平化的辅助曲线的帮助下），这些伪曲线坐标允许我们沿曲线（记为 *l*）快速找到**真实曲线坐标**：

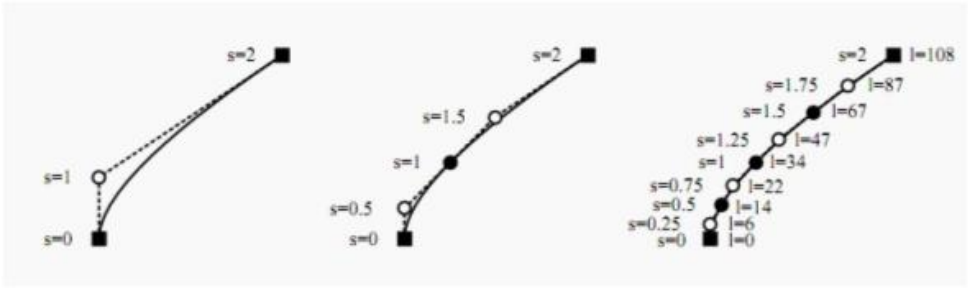

图5：伪曲线坐标 s 用于快速计算真实曲线坐标 l。注意 s 对曲线真实长度没有影响

当一个图形通过编辑，裁剪或扁平化修改时，这些变化会被该图形的**监听器**注意到。这些变化被描述为一个移除的曲线和面的集合，以及一个新加曲线和面的集合。换句话说，一个修改的曲线，例如可以描述为一个移除的旧曲线和一个新加的新曲线。这些变化可以被裁剪和扁平化操作的增量版本利用。**增量裁剪**操作用源图形，裁剪为某区域的相同图形，以及对源图形所做的变化作为参数。它输出一个更新的剪裁图形和对该剪裁图形的变化。它并没有对整个源图形做裁剪，而是使用源变化仅对剪裁图形的改变的元素做重计算。**增量扁平化**操作以类似方式工作。

一个**图形处理类**为每个四叉树之一生成图形块。其仅包含与该四边形（边界系数计算在内）相交的矢量数据，采用合适的阈值使得曲线可以在有较好精度的栅格块的辅助下绘制。根四边形的图形由简单的从硬盘载入图形"生成"。该图形下的任何其他四边形都是对其父四边形的*裁剪*生成，以及父图形使用的阈值的一半来*扁平化*。

注意：

- 在硬盘上预计算和存储给定四叉树水平 L 的剪裁图形也是可以的。这是没有图形从 0 级到 L-1 级生成。 L 级图形是由硬盘加载而"生成"。大于 L 级的图形是由其父图形的裁剪和扁平化生成。该选项对特大图形非常有用。
- 我们说父四边形的图形是子四边形的四个剪裁图形的**父图形**。如果有的话（比如，如果该图形由另一个图形剪裁生成），一个图形有一个相对其父图形的参考系。同样，一个曲线有一个相对其**父曲线**的参考系，如果该曲线由另一个曲线剪裁生成，一个面有一个相对其**父面**的参考系，如果该面由另一个面剪裁生成。一个图形，曲线或面的**祖先**是元素的父辈的父辈的父辈…，直到没有父辈。

一个图形处理类是其根图形的监听器。任何作用于根图形的变化都会被通知，例如通过 proland::EditGraphOrthoLayer 提供的图形化用户界面（见**预定义正交 GPU 处理类层**）。当除根以外的四边形的图形被检测到变化，那么使用增量裁剪和扁平化操作从上至下进行增量重计算。更多细节见下文。

1.1.1 图形类

以上概念通过**图形**模块中的类实现。一个图形是一个 proland::Graph 的实例。该类定义了所以可以在图形上实施的函数，如载入和保存至硬盘，获取它们的节点，曲线和面，移动，编辑和移除节点，曲线和面，是否增量裁剪和扁平化图形，等等。它也管理着通知图形变化的监听器。该类用于任何你需要操纵图形的地方，*除了*实例化一个图形。实际上该例是抽象类。你只能实例化它的两个子类， proland::BasicGraph 和 proland::LazyGraph 。第一个一直在内存中保留整个图形，直到图形被删除。第二个仅当使用时才从硬盘载入图形元素（亦即*懒惰的*），并且在它们不再使用时从内存删除。它用于超出 CPU 寄存器容量的非常大的图形。

proland::Node 类表达一个节点。它存储节点的 (x, y) 坐标，连接到该节点的曲线的引用，和它所属的图形的引用，称为它的*拥有者*。

proland::Curve 类表达一个曲线。它存储该曲线的起始和结束节点（以及它们的伪和真曲线坐标 s 和 l ），曲线的顶点（每个顶点的 x, y, s 和 l 坐标，加上一个区分端点和控制点的布尔变量），曲线类型和宽度，曲线每侧的面的引用（如果有的化），父曲线的参考（如果有的话），以及拥有者图形的引用。

proland::Area 类表达一个面。它存储了该面引用的每个曲线及其方向（一个指示曲线是否必须遵循从起点到终点或反过来 - 这些有方向曲线必须以逆时针形成闭合环）。它也存储父面的引用（如果有的话），和拥有者图形的引用。

proland::BasicGraph 使用智能指针引用它的节点，曲线和面。因此直到图形对象本身被删除这些元素才能被删除，即使这些节点，曲线和面已经不引用自己（除了被图形重调，智能指针可以看作"强指针"，防止在源之前目标被删除；相反普通指针可以看作"弱指针"，目标可以在任何时候删除）。曲线和面也使用智能指针，来引用它们的父元素。然而，因为智能指针绝不能形成循环，节点，曲线和面使用普通指针引用它们的拥有者图形：

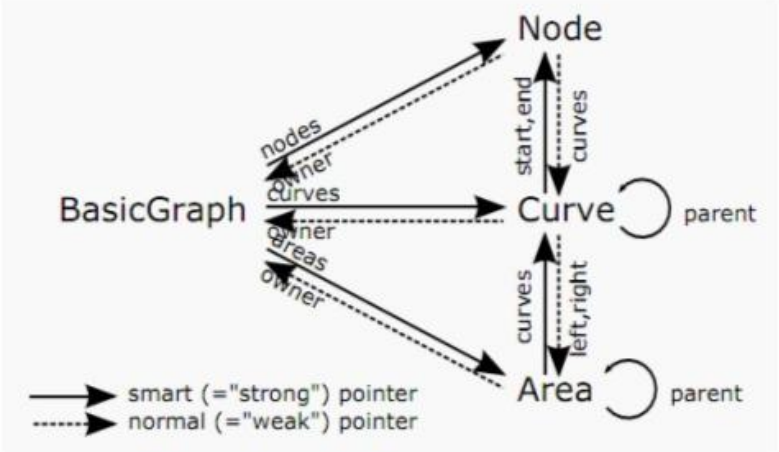

图6：一个基础图形使用智能指针引用它的元素，而元素本身使用普通回指针（避免循环引用）

proland::LazyGraph 使用普通指针引用它的元素。然而普通指针只能在内部使用。在外部仍使用智能指针。结果是，当一个元素返回到"客户端"变得不再被任何智能指针引用后，该元素被删除（用 ork::Object::doRelease 方法）。实际上不是这样，因为该释放方法在 proland::LazyNode 类，
以及 proland::LazyCurve 和 proland::LazyArea 子类中被重写了。该重写方法将未引用元素放入缓存（大小由用户在 proland::LazyGraph 中指定）。如果这些元素在短期内再次使用，它们将从缓存中再启用而是重新构建。因此缓存防止了元素永久被删除和重建。

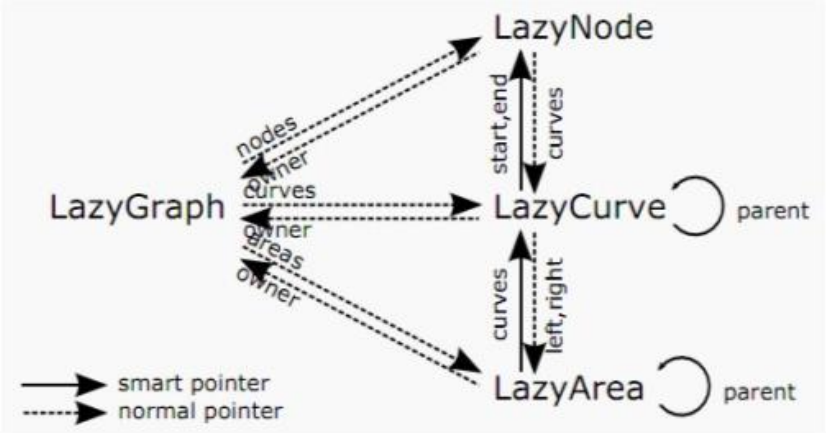

图7：内部，懒图使用普通指针引用它的元素。外部使用智能指针。因此当不再被引用时，元素可以被删除

实际上， proland::LazyGraph 中的每个引用都由一个标识符和一个普通指针组成。该标识符指示，通过存储在图形文件中的索引图，指定元素在硬盘的什么位置存储。如果引用元素还没被载入内存，或者不在内存中了，则指针为空。这时，如果元素被请求，标识符用于从硬盘载入它并且创建它的内存再现。这些标识符被客户端用于在只需要指明而不使用时，指定图形元素。为了统一化基本图和懒图的标识符（对于基本图，“标识符”就是指定元素的一个指针），proland::NodeId, proland::CurveId, 和 proland::AreaId 类型是一致的，可以包含一个整型或一个指针。

注意：

- 一个懒图句柄以一个特殊方式添加，修改和删除元素，为了避免从硬盘上重新载入一个元素的过期版本。实际上懒图用智能指针引用添加和修改的元素，以至于它们在图形本身之前不能被删除。被删除元素从用来从标识符载入元素的索引图中被移除，所以不会不慎从硬盘再次载入。

一个图形的监听器用 proland::Graph::addListener 和 proland::Graph::removeListener 方法添加和移除。每个监听器必须实现包含一个单一方法的 proland::GraphListener 抽象类。当图形发生改变时，它们存储在 proland::Graph::changes 字段，和 proland::Graph::Changes 类型，然后监听器被通知。注意 changes 字段的前一个值会丢失。因此监听器必须在其被新变化重写之前读取该值时必须小心。一个图形改变的次数存储在 proland::Graph::version 字段。这个整型初始化为 0 ，并且在每次改变之后，监听器被通知之前增加。

用于裁剪图形的边界系数由 proland::Margin 对象指定。该类在裁剪时计算了用于每个节点，曲线或面的边界系数（每个元素可以使用不同的系数）。它也可以指定一个来自裁剪区域本身的附加系数。这用于解释块边界（见 处理器框架“处理器框架”一节）。该类是抽象类，亦即，你必须重写它来定义系数如何被计算，取决于你想对图形怎样做。

图形可以容易的被扩展，为了展现特别的数据（例如为节点，曲线，面增加指定参数）。几乎所有的图形方法都实际上支持这点。用户自定义图形将需要重实施新节点，曲线，面的方法来创建正确的节点，曲线和面的扩展，就像创建孩子方法用来保证新子图被正确创建。为了正确的载入文件，载入和保存函数也将被重写。

那么用户可以改变几乎所有他想要改变的图形行为。

注意：

- 为了对基本图可重复使用，图形有一个参数统计了每种参数的数量（曲线参数，节点参数...的数量），写入'.graph' 文件，当载入文件时检查该参数：例如，你可以有一个将一个浮点型值添加到一个曲线中的图形。该值会在基本图和懒图中被忽略。

1.1.2 图形文件

图形可以以四种格式在硬盘上存储：基础 ASCII ，索引 ASCII ，基础二进制，索引二进制。文件的第一个字节指明了文件类型：0 表示基础二进制， 1 表示索引二进制， 48 （亦字符 '0' ）表示基础 ASCII ，和 49 （亦字符 '1' ）表示索引 ASCII 。 ASCII 格式容易手动编辑。索引格式含有预计算的索引图指示了文件中每个元素的偏移量。它们比未索引文件载入更快（未索引文件必须全部被载入为了在运行时构建索引图）。

以下 7 个参数用于决定是否文件中的图形适合我们想用于加载它的图形类：

- 节点参数的数量：默认为 2 (X, Y)
- 曲线参数的数量：默认为 3 （大小，宽度和类型）
- 面参数的数量：默认为 3 （大小，信息和子图布尔型）
- 曲线末端参数的数量：默认为 1 （节点数）
- 曲线顶点参数的数量：默认为 3 （X, Y, 是否为控制点）
- 面的曲线参数的数量：默认为 2 （曲线数，方向）
- 面的子图参数的数量：默认为 0

在这些数据信息字节之后，非索引文件必须为以下格式：

- 节点数（整型）
- 每个节点的描述
- 曲线数（整型）
- 每个曲线的描述
- 面数（整型）
- 每个面的描述
- 每个子图的描述

索引文件必须为以下格式：

- 文件中索引图的偏移量（长整型）
- 每个节点的描述
- 每个曲线的描述

- 每个面的描述
- 索引图

一个节点描述为以下格式：

- x 坐标（浮点型）
- y 坐标（浮点型）
- 连接到该点的曲线的数量（整型）
- 对每个连接的曲线：
  - 该曲线的标识符（整型）

一个曲线描述为以下格式：

- 顶点数（包括起始点和结束点 - 整型）
- 宽度（浮点型）
- 类型（整型）
- 起始点的标识符（整型）
- 对每个顶点：
  - x 坐标（浮点型）
  - y 坐标（浮点型）
  - 是否控制点（布尔型）
- 结束点的标识符（整型）
- “左侧”面的标识符（整型， -1 为空）
- “右侧”面的标识符（整型， -1 为空）
- 根图文件中祖先曲线的标识符 - 必须是索引文件（整型， -1 为空）

一个面描述为以下格式：

- 该面中曲线的数量（整型）
- 用户数据（整型）
- 如果该面有子图，设置为真的布尔值（整型）
- 对面中的每个曲线：
  - 曲线的标识符（整型）
  - 曲线方向（整型）
  - 根图文件中祖先面的标识符 - 必须是索引文件（整型， -1 为空）

一个子图的描述与非索引文件相同（减去指示格式的第一个字节）。子图总是随基本图载入，即使图形本身以懒图载入。

索引文件中的索引图为以下格式：

- 节点数（整型）
- 曲线数（整型）
- 面数（整型）
- 子图数（整型）
- 对每个节点，它的描述在文件中的偏移量（长整型）

- 对每个曲线，它的描述在文件中的偏移量（长整型）
- 对每个面，它的描述在文件中的偏移量（长整型）
- 对每个子图，它的描述在文件中的偏移量（长整型）

一个节点的标识符简单的为节点列表的索引，为它们在文件中出现的顺序。曲线，面和子图也类似。

这里有一个非索引 ASCII 文件的例子（注释并*不*是文件的一部分）：

```
0 // Format: not indexed, ASCII
2 3 3 1 3 2 0 // Number of parameters for each type: Nodes, curves, areas,
curve extremities, curve vertices, area curves and subgraphs parameters.
5 // Number of nodes
-10000.000 -10000.000 2 0 1 // 1st node description
10000.000 10000.000 2 1 2 // 2nd node description
10000.000 -10000.000 2 2 0 // 3rd node description
150.000 13510.000 0 // 4th node description
500.000 400.000 1 3 // 5th node description
4 // Number of curves
2 1.000 0 // 1st curve description
0
1
0 -1 -1
2 1.000 0 // 2nd curve description
1
2
0 -1 -1
25 1.000 0 // 3rd curve description
2
-5634.000 -1208.000 0 // 3rd curve's vertices
[...]
-5224.000 -1678.000 0
-5244.000 -1698.000 0
-5864.000 -1228.000 0
0
0 -1 -1
4 1.000 0 // 4th curve description (a simple loop)
4
520.000 400.000 0
500.000 420.000 0
4
1 -1 -1
3 // Number of areas
3 1 0 // 1st area description
0 0
1 0
2 0
-1
1 1 1 // 2nd area description. This one has a subgraph
3 0
-1
// Begining of the subgraph descriptions
1 // Number of nodes
515.000 405.000 1 0 // 1st node description
1 // Number of curves
3 2.000 0 // 1st curve description
0
516.000 408.000 0
0
0 -1 -1
1 // Number of areas
0 1 // ?
-1 // ?
```

注意：

- .graph 文件**不**支持注释

1.1.3 图形编辑

图形是被设计为可编辑的。为这个目的， proland::Graph 类提供的每个函数都需要更新它们的内容，保持在系统中的稳定性，独立于图形类型（如果图形类是扩展的，并且如果你在节点，曲线，面之间添加了特别的链接，那么它们可能不得不被重新定义）。

每次添加/移除一个节点或曲线，图形类必须改变来保持一个一致性（亦即，如果曲线剪裁了一个面，它造成了两个面，如果曲线从一个面中删除了，那么这个面也应该被删除，但如果曲线有两个面，那么它们必须被重建）。

为避免浏览整个图花费太多的时间，添加和移除方法是智能的并且仅仅浏览相联系的曲线和面来检测该做什么。它检查如果你添加/移除一个曲线的地方存在循环，那么仅仅重建这些面。因为有**许多**特例，所以这个行为是强制的。当你移除一个节点时也一样，它要么删除周围的曲线，要么合并它们。如果被合并，则必须正确的被重建并添加到属于它们的面中。

进一步讲，对每个编辑函数来说，图形类是能够决定变化了什么，并在 proland::Graph::Changes 结构中返回它。如果一行中有多个变化，例如为了不删除同一个曲线两次，或避免刚移除就添加它，这些变化可以合并。这些变化结构在决定子图中什么该更新什么不该更新时非常有用，特别是对 proland::GraphProducer 类来说。

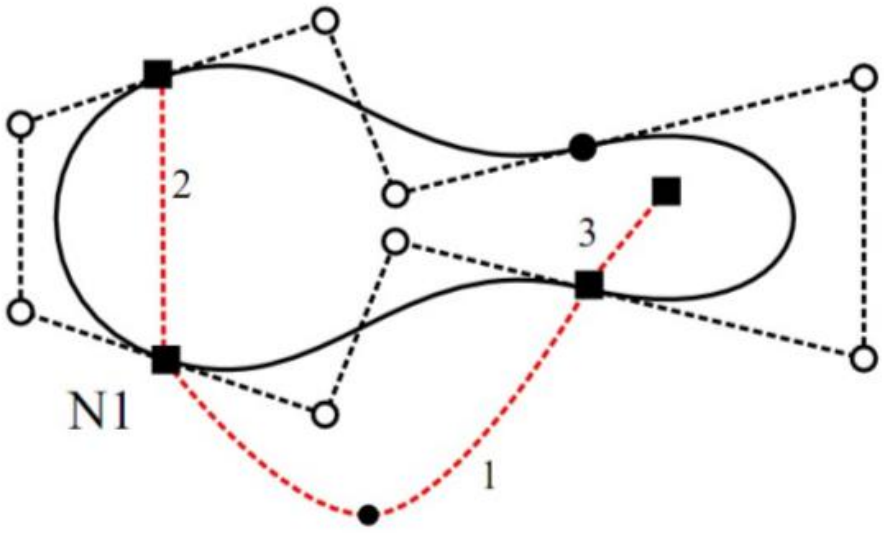

图8：当添加一个曲线时，系统检查是否该曲线创建一个新的面 (1) 或是否它分割了一个已有的面 (2)。如果是，面将通过浏览它们所链接的曲线来被重建。它也能够决定除了添加节点刷新曲线本身以外，添加行为 (3) 是否不影响面。当删除一个曲线时也要做同样的检查操作。删除曲线 (1) 将只删除一个面，但删除曲线 (2) 将合并它周围的两个面。最后，删除曲线 (3) 不影响任何曲线，因为它不是任何曲线的一部分（它也不是循环的一部分，因此应该在面建立过程中排除掉）

1.1.4 图形处理类

proland::GraphProducer 类可以从硬盘载入预计算图形，然后把它划分到相关子图的块中显示。其行为如下：

对根块来说，处理类简单的从文件载入图形（见图形文件一节载入和保存）。对每个其他块来说，如果没有存储在缓存中，相应图形从其父图形中裁剪。

因为图形处理类也是图形监听器，当一个图形变化时，它们能够决定。当此发生时，图形处理器对根块无效。根块不会被重计算，但会包含它上一步修改的变化，由于图形依赖于其父图形，需要的话，1 级的图形将不得不被检查然后更新。操作中如果有任何变化，1 级图的子图将变的无效，然后对其做同样的处理。图形仅当需要时才更新，即，当程序调用 GraphProducer#getTile 方法时。

智能更新机制由此启动：当为给定块计算一个图形时，图形处理类能决定是否该图形已经在缓存中并且无效。如果是，那么它需要被检查。否则，该图形就被剪裁（如果不在缓存中）或者直接就不更新（如果在缓存中并且仍然有效）。

每个图形都包含一个版本号，使图形处理类能够决定哪些图形要更新哪些不更新。如果图形的版本号与其父图形相同，那么它已经被更新到最新版了。如果它的孩子随它更新到最新版，那么它也不需要更新。如果孩子的版本号是其父亲的前一个版本，我们就对当前块使用裁剪更新。否则，如果版本号都不一样，图形将不得不全部重新裁剪。当更新后，每个图形获得一个变化的列表，用于增量更新机制。这也帮助决定是否在当前块中有任何修改，因此来决定是否它的子图需要被更新。

变化列表由图形有添加和移除曲线和面的更新时创建。这在裁剪更新时仅允许更新这些变化的曲线和面。这是增量的，即，每次图形被更新时，它检查它的父图形的最新变化列表和版本，更新自己的曲线和面（如在当前图形中有的话，则更新变化列表中的子曲线和子面），和它自己的变化列表。如果没有元素被更新，那么只更新图形版本号，而且图形处理器不会使子块无效。否则，孩子将会无效，并且将轮到它们使用前一步计算的版本号和变化列表。

用户可以决定是否当前使用块的父块应该被存储（在层次之间加速转换，但是会使用更多的内存）。

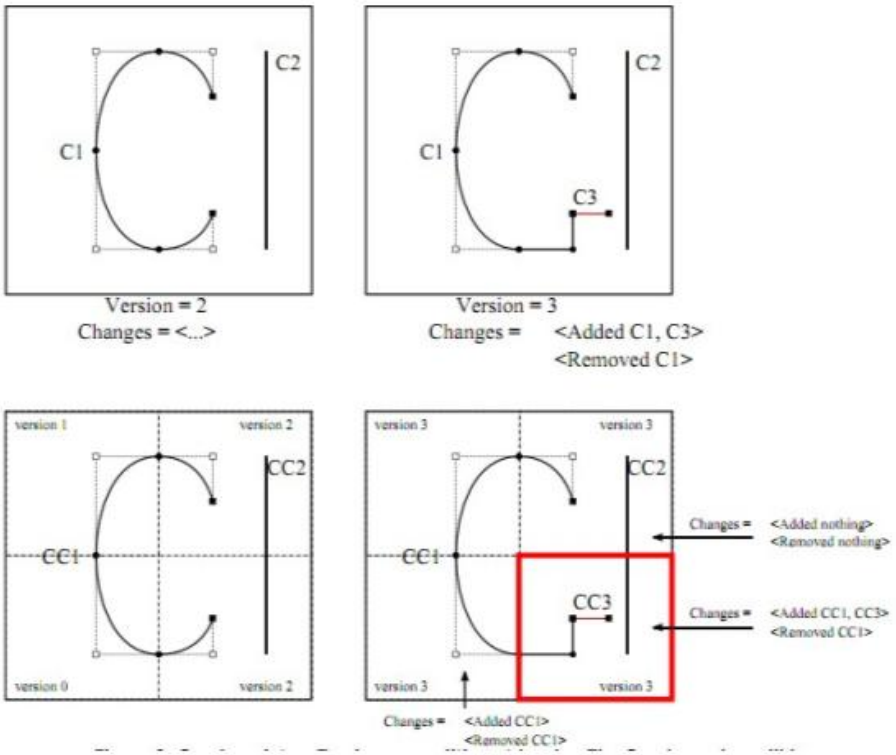

图9：图形更新。上图：在 n 级编辑。图形版本号将增加，并且包含了它的变化（增加一条曲线，去除一条曲线）。下图，在低一级发生的变化是：如果版本号不同于其父亲，块会被重新计算；左块被全部重计算，而右块只是被更新。更新后的图将包含所有的变化数据。那些被全部重计算的不需要存储变化数据，因为它们的孩子也被全部重新裁剪。左下块指明的变化列表包含是否它只是被更新。右上块没有任何变化因为在它的图中没有包含任何变化部分：曲线 C1 在其子块上实际被剪裁为更小的曲线，只有那些与变化部分相关的被更新。也要注意一个被移动或编辑的曲线（在曲线 C1 上，只有点变化的状态）将出现在移除曲线列表上，但是也出现在增加曲线列表上，所以它将在子图中被移除然后再添加

如下一些选项可用：

- 预计算图形：用户指定一个或多个预计算层次，为其提供相应的图形。这些图形必须以相同的名字存储在根图文件夹的子文件夹中。文件命名规则为：[图形名]_[层次]_[x 坐标]_[y 坐标].graph 。例如：roads_03_00_00.graph 为 3 级的左下块。如果一个或多个文件不存在，处理类将从根图创建它们（当需要时）。注意如果 doFlatten 选项设置为真，这些子图将不得不扁平化。那么，当需要时， proland::GraphProducer 将载入这些文件而不是从根图去裁剪获得。该选项允许避免未显示层次的无用计算，提高了载入速度。
- 边界系数：使用处理类的每个类都需要来自邻居块的附加数据（例如，显示沿块边界的一个道路，如果没有被考虑则将被切开）。所以，当创建后，这些类将通过 Margin() 和 removeMargin() 方法给图形处理类添加边界系数。最新的系数会被使用。
- 图形监听器：因为它们使用图形来生成其他图形，图形处理类必须知道什么时候图形被编辑。所以，图形处理类执行 proland::GraphListener::graphChanged() 方法：它通过智能

更新机制，使第一个引起变化蔓延到子图的块无效：只有被请求的块才被更新（如上文所述）。每次图形被编辑都调用此方法。

- 曲线数据：图形处理类也保留一个曲线数据对象的缓存。曲线数据类包含关于给定曲线的信息，如长度，类型等。例如，它在层中使用，来计算高度，长度，条带等。给缓存赋一个大小来避免内存过载。

1.1.5 图形处理类资源

proland::GraphProducer 可以用 Ork 资源框架载入，有如下格式（ “graph1” 例子说明如何使用图形处理类）：

```
<graphProducer name="myGraphProducer" cache="graphCache"
file="myGraphFile"
loadSubgraphs="true" doFlatten="true" precomputedLevel="3"
precomputedLevels="3,1:5"
nodeCacheSize="0" curveCacheSize="100000" areaCacheSize="100000"
dataCacheSize="-1" storeParents="true"/>
```

该图形由 cache 属性指定的块缓存资源生成。该缓存必须有一个辅助的 ObjectTileStorage 。根图是从 file.graph 文件 载入，*文件*是 file 参数的值。如果 precomputedLevel 和 precomputedLevels 被设置，预计算剪裁图形就从*文件*文件夹载入。在该文件夹中，每个剪裁图形必须命名为 file_level_tx_ty.graph， level, tx, ty 是逻辑四叉树坐标。如果这些预计算文件不存在，那么在第一次使用时创建。预计算层次的使用必须用逗号分隔开每个层次。你可以通过在开始和结束层之间放置一个克隆来指定一个层次范围。 precomputedLevel 只允许 1 指定预计算层次；它这里主要为了与其他版本号共存。用户也可以指定是否在每次剪裁之后使用扁平化方法。如果你使用存在的预计算剪裁图形，它们必须与这些选择一致（即如果你选择 doFlatten='true'，它们就必须扁平化）。最后，用户也有一个选项来存储当前使用块的父块（ storeParents ）。如前所述，该选项允许在层次之间加速转换，但必须保留在内存中，一个在高层次的块会占用很大的内存。

懒图缓存大小可以用 nodeCacheSize / curveCacheSize / areaCacheSize 属性设置。最后，缓存曲线数据的最大数可以用 dataCacheSize 设置。 0 意味着没有缓存， -1 为无限制缓存。

1.1.6 曲线数据

proland::CurveData 类包含关于曲线的数据。一个曲线和它的孩子共享同样的曲线数据；这使得在不同层次的细节上保持了一致性。

它存储了由图形处理类创建生存的根曲线的扁平化版本。基本上，它包含了曲线的一般信息：起始和结束端点长度，边界，曲线长度和伪曲线坐标，以及其他任何你想添加的数据。所有这些数据只在需要时计算一次，然后存储起来为之后使用。

例如， ElevationCurveDatas 存储了高程曲线的摘要。曲线数据的主要目的是避免在不同块的不同层次采样同一个点，这会导致不连续性（非对齐或切断条带）。曲线数据信息在仅依赖曲线宽度的层次上计算，来避免层次之间的任何爆音效应。对曲线坐标和端点长度来说，一个友好的使用情况是道路（见下图）：它们帮助在结束节点添加特别的行为但都沿着该曲线。

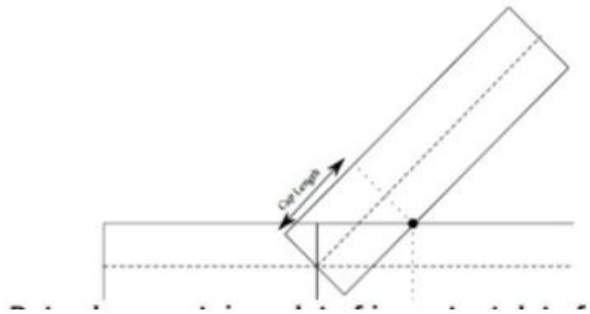

图10：曲线数据类包含很多用来绘制不连续自由图形的信息。端点长度允许决定曲线与其他与它相连的曲线在哪里相交。当结合了为每个点存储的曲线长度参数，它将很容易决定是否一个条带应该被绘制。而且，为避免在块边界上的不连续性，可以使用同样的曲线长度参数，以至于我们总是知道当前点在哪个曲线中

曲线数据由 proland::CurveDataFactory 处理。默认情况下，该工厂应该是图形处理类本身，但是如果你想扩展曲线数据，它可以被改变（见 proland::ElevationGraphLayer 和它的子类：只有 newCurveData() 方法可以被重写）。

该工厂工作与块处理类很相似：提供 3 个主要方法：

- getCurveData(): 如果曲线数据不存在，它将被创建。否则，它的使用计数器增加。它应该在你的处理类（或在你的任务的获取任务中）的开始创建块部分使用。
- findCurveData(): 只返回曲线数据，不改变使用计数。其应该在任务执行时使用，如果曲线数据不存在则返回空
- putCurveData(): 使用计数器减少。当曲线数据不被使用时（puts() 数等于 gets() 数），它被删除。该函数在你的处理类（或任务结尾）的终止生成块部分使用。

曲线数据工厂也会告知哪个曲线数据在给定块中被使用，这可以在该块中为每个曲线数据直接调用 put() 方法。两种方法定义如下：

- addUsedCurveDatas(): 决定块中的曲线数据
- releaseCurveData(): 在给定块为每个曲线数据调用 put 方法

最后，曲线数据工厂也是一个图形监听器，因此也能够检测曲线数据中的曲线发生变化和更新。

像块处理类一样，你可能想要为曲线数据增加依赖，这可以在图形层中请求。proland::GetCurveDatasTask 任务为此而建。它取决于给定的图形，在该图形中为每个曲线调用 getCurveData() 方法。

但是有时候，用户可能需要关于所有沿着给定曲线的数据，和与该曲线交叉的每个块的数据。通常，需要两步，即，两个框架（一个计算图形，一个计算与图形中曲线交叉的块），这不适于一般图形应用。

获取曲线数据任务能够处理这个：它在**执行时**改变图形任务，在图形创建后以及需要"随机"块的任务执行前。它强制处理类在需要的任务执行前创建裁剪它的曲线数据的块。

图11：高程曲线数据的例子：在覆盖蓝色区域（曲线）的每个块上，高程层必须知道沿着曲线的高程剖面。但是为避免不连续，曲线数据也需要来自高程类的高程，但不是必须全部计算（例如屏幕外）

注意：

- 记住，使用获取曲线数据任务时，你应该在你的任务末尾**一直**手动调用 releaseCurveDatas() 方法，否则内存将因为曲线数据没有被删除而饱和。

2 地形处理类

前节图形处理类可以通过层在地形插件处理类中使用，为了基于矢量数据生成或修改地形纹理或形状（如在地形中绘制或插入道路）。这些层在下文展开介绍。

2.1 高程处理类

2.1.1 基于层的图形

- 道路高程图层

proland::RoadElevationLayer 类继承于 proland::TileLayer 类，在高程纹理上绘制道路。它修改地形高程以至于道路交叉部分是水平的，以及道路高程剖面是平滑的，保证在道路轨迹区域的原始地形得到连续过渡。

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

```
<roadElevationLayer name="roadOrtho1" graph="roadGraph1"
cpuElevations="groundElevations1" renderProg="roadLayerElevationShader;"
level="3"/>
```

graph 属性必须包含需要绘制的图形处理类的名字。 cpuElevations 属性必须包含在 CPU 上计算地形高程的块处理类的名字。 renderProg 属性必须包含用于渲染道路的 ork::Program 。level 属性表示开始显示的层的第一级。 "graph1" 例子演示了该层是如何使用的。

- 水体高程图层

proland::WaterElevationLayer 类继承于 proland::TileLayer 类，在高程纹理上绘制水体。它修改地形高程以至于水体交叉部分是水平的，以及水体高程剖面是平滑的和总是降低的（水不能向上流），保证在水体轨迹区域的原始地形得到连续过渡。

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

```
<waterElevationLayer name="waterOrtho1" graph="waterGraph1"
cpuElevations="groundElevations1" renderProg="waterElevationOrthoShader;"
fillProg="fillShader;" level="3"/>
```

graph 属性必须包含需要绘制的图形处理类的名字。 cpuElevations 属性必须包含在 CPU 上计算地形高程的块处理类的名字。 renderProg 属性必须包含用于渲染小水体的 ork::Program 。level 属性表示开始显示的层的第一级。 “river1” 例子演示了该层是如何使用的。

2.2 正交 GPU 处理类

2.2.1 基于层的图形

- 线正交图层

proland::LineOrthoLayer 类继承于 proland::TileLayer 类，在一个正射纹理上绘制图形。该图形以一个像素宽的 OpenGL 线绘制。

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

```
<roadOrthoLayer name="lines" graph="myGraph"
renderProg="myShader;" level="3"/>
```

graph 属性必须包含需要绘制的图形处理类的名字。 renderProg 属性必须包含用于渲染线的 ork::Program 。level 属性表示开始显示的层的第一级。

- 掩膜正交图层

proland::MaskOrthoLayer 类继承于 proland::TileLayer 类，在一个正射纹理上绘制图形。该图形通过使用正确的宽度栅格化它的曲线和面，然后绘制。许多选项可以使用，用来混合此结果到当前块中，只在一些通道中绘制，忽略一些曲线，等等。该图层的一个典型使用是调节一个密度贴图纹理，为了排除定义在图形中的一些面。例如，在道路和水体的地方设置草地或树木密度为 0 （避免在道路和水体中出现实例化的草叶和树）。 “river1” 例子演示了该层如何使用。

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

```
<maskOrthoLayer name="mask" graph="myGraph"
renderProg="myShader;" level="3"
withFactor="1" color="255,0,0"
ignore="1,2," deform="false"
equation="ADD" equationAlpha="ADD"
destinationFunction="ZERO" sourceFunction="ONE"
destinationFunctionAlpha="ZERO" sourceFunctionAlpha="ONE"
blendColor="0,0,0,0" channels="r"/>
```

graph 属性必须包含需要绘制的图形处理类的名字。 renderProg 属性必须包含用于渲染线的 ork::Program 。level 属性表示开始显示的层的第一级。 withFactor 属性是一个在绘制曲线之前应用于曲线宽度的一个因子， color 属性是用于绘制曲线和面的颜色。 ignore 属性指定了一个绘制图形时必须跳过的曲线类型的列表。最后， deform 属性必须指定哪种地形变形应用于地形。当前只有 “none” 和 “sphere” 值可以使用，意思是不变形和球形变形（渲染球体）。最后一个属性指定了混合模式以及将图层混入块的公式，以及只在某些通道用于写入掩膜（r, g, b, 和 / 或 a）。

- 道路正交图层

proland::RoadOrthoLayer 类继承于 proland::TileLayer 类，在一个正射纹理上绘制道路。道路绘制依赖于显示分辨率。如果道路宽度小于一个像素，则用线绘制，否则使用三角带。另外，为了决定是否绘制道路带，十字路口等，需要设置一个质量参数。”graph1” 例子演示了该层是如何使用的。

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

```
<roadOrthoLayer name="roadOrtho1" graph="roadGraph1"
renderProg="roadLayerOrthoShader;" level="3"
color="64,64,64" dirt="154,121,7" border="43,68,20"
border_width="1.2" inner_border_width="2.0"
deform="false"/>
```

graph 属性必须包含需要绘制的图形处理类的名字。 renderProg 属性必须包含用于渲染道路的 ork::Program 。level 属性表示开始显示的层的第一级。 color 属性是道路的颜色，RGB 格式。 dirt 属性包含小路的颜色（宽度为 1 ），RGB 格式。 border 属性包含道路边界的颜色，RGB 格式。最后， border_width 和 inner_border_width 是绘制时添加到道路的边界（内界和外界）的大小（ deform 属性和上文相同）。

- 水体正交图层

proland::WaterOrthoLayer 类继承于 proland::TileLayer 类，在一个正射纹理上绘制静态河流。该层中， proland::Area 表现大河大湖， proland::Curve 表现小河。

水体正交图层可以用 Ork 资源框架载入，有以下格式：

```
<waterOrthoLayer name="waterOrtho1" graph="waterGraph1"
renderProg="waterLayerOrthoShader;" level="3"
color="255,0,255" deform="false"/>
```

graph 属性必须包含需要绘制的图形处理类的名字。 renderProg 属性必须包含用于渲染河流的 ork::Program 。level 属性表示开始显示的层的第一级。最后， color 属性是河流的颜色，RGB 格式。 该例中，河流是紫色的。 deform 属性和上文相同。