* + [一、程序技术篇：算法和架构（Programming Algorithms & Architecture）](http://gulu-dev.com/post/2014-11-16-open-world#toc_0)
    - [1. 无限循环的平铺地图（Infinite Tiling）](http://gulu-dev.com/post/2014-11-16-open-world#toc_1)
    - [2. 可预测随机数和无限宇宙（Predictable Random）](http://gulu-dev.com/post/2014-11-16-open-world#toc_2)
    - [3. 精度问题解决方案](http://gulu-dev.com/post/2014-11-16-open-world#toc_3)
    - [4. 超大地形的处理 （Terrain Visualization）](http://gulu-dev.com/post/2014-11-16-open-world#toc_4)
      * [4.1 古典算法（从 GeoMipMapping，Progressive Mesh 到 ROAM）](http://gulu-dev.com/post/2014-11-16-open-world#toc_5)
      * [4.2 层次的艺术（Quadtree 和 Chunked LOD）](http://gulu-dev.com/post/2014-11-16-open-world#toc_6)
      * [4.3 以GPU为主的技术（从 Paging，Clipmap 到 GPU Terrain）](http://gulu-dev.com/post/2014-11-16-open-world#toc_7)
    - [5. id tech 5 的 megatexture （超大地表上的非重复性海量贴图）](http://gulu-dev.com/post/2014-11-16-open-world#toc_8)
    - [6. 过程式的内容 （Procedural Content Generation）](http://gulu-dev.com/post/2014-11-16-open-world#toc_9)
      * [6.1 过程式纹理（Procedural Texturing）](http://gulu-dev.com/post/2014-11-16-open-world#toc_10)
      * [6.2 过程式建模（Procedural Modeling）](http://gulu-dev.com/post/2014-11-16-open-world#toc_11)
  + [二、内容制作篇：设计和创造（Content Design & Creation）](http://gulu-dev.com/post/2014-11-16-open-world#toc_12)
    - [1. 随机地图类游戏 (Diablo II) 中地图的拼接](http://gulu-dev.com/post/2014-11-16-open-world#toc_13)
    - [2. 无缝大世界 (World of Warcraft) 中区域地图的拼接](http://gulu-dev.com/post/2014-11-16-open-world#toc_14)
    - [3. 卫星地质数据的导入，规整化和再加工（一些飞行模拟类游戏）](http://gulu-dev.com/post/2014-11-16-open-world#toc_15)
    - [4. 超大地图的协同编辑：并行操作，数据同步，手动和自动锁的运用](http://gulu-dev.com/post/2014-11-16-open-world#toc_16)
  + [三、异次元篇：我们的征途是星辰大海](http://gulu-dev.com/post/2014-11-16-open-world#toc_17)
    - [1. 终极沙盒（EVE）：当规模大到一定程度——宇宙级别的混沌理论与蝴蝶效应](http://gulu-dev.com/post/2014-11-16-open-world#toc_18)
    - [2. 打通两个宇宙（EVE & Dust）：发现更广阔的世界——宇宙沙盒游戏和行星射击游戏联动](http://gulu-dev.com/post/2014-11-16-open-world#toc_19)

**一、程序技术篇：**[**算法**](http://lib.csdn.net/base/datastructure)**和**[**架构**](http://lib.csdn.net/base/architecture)**（Programming Algorithms & Architecture）**

* + 无限循环的平铺地图（Infinite Tiling）
  + 可预测随机数和无限宇宙（Predictable Random）
  + 精度问题解决方案（Precision Problem Solving）
  + 超大地形的处理 （Terrain Visualization）  
    4.1 古典算法（从 GeoMipMapping，Progressive Mesh 到 ROAM）   
    4.2 层次的艺术（Quadtree 和 Chunked LOD）  
    4.3 以GPU为主的技术（Paging，Clipmap 到 GPU Terrain）
  + id tech 5 的 megatexture （超大地表上的非重复性海量贴图）
  + 过程式内容生成 （Procedural Content Generation）  
    6.1 过程式纹理（Procedural Texturing）  
    6.2 过程式建模（Procedural Modeling）

**二、内容制作篇：设计和创造（Content Design & Creation）**

* + 随机地图类游戏 (Diablo II) 中地图的拼接
  + 无缝大世界 (World of Warcraft) 中区域地图的拼接
  + 卫星地质数据的导入，规整化和再加工（一些飞行模拟类游戏）
  + 超大地图的协同编辑：并行操作，数据同步，手动和自动锁的运用

**三、异次元篇：我们的征途是星辰大海**

* + 终极沙盒（EVE）：当规模大到一定程度——宇宙级别的混沌理论与蝴蝶效应
  + 打通两个宇宙（EVE & Dust）：发现更广阔的世界——宇宙沙盒游戏和行星射击游戏联动

**一、程序技术篇：算法和架构（Programming Algorithms & Architecture）**

**1. 无限循环的平铺地图（Infinite Tiling）**

按照维度的不同，循环平铺有下面三大类：

a. 在一维方向上扩展的横版卷轴游戏（以动作类游戏为主）和纵版卷轴游戏（以射击类游戏为主）。这些类型的游戏里，为了避免循环平铺给玩家带来的重复的疲劳，卷轴游戏会添加一些随机或动态的元素，比如超级玛丽里的背景上云朵的位置，分出多个层次以不同速率卷动的背景层，等等。

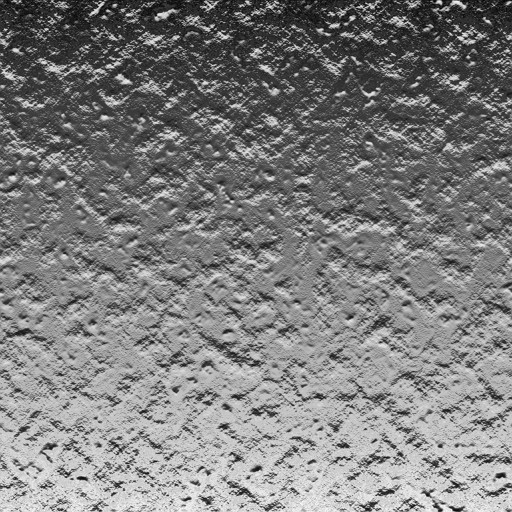


b. 在二维方向上循环平铺的固定视角2D游戏。这一类游戏里，比较典型的就是 Diablo。暗黑中的随机地图生成，在本质上，就是叠加了一定的随机性，约束和边界条件的循环平铺效果。



c. 在 3D 游戏里循环平铺高度图，形成连绵不断的地形效果。

找不到游戏内的图，拿下面这个高度图来凑数吧。请大家脑补一下，把下面这个灰度图立体化之后，一块一块像地砖一样循环平铺以后，3D渲染出来的连绵起伏的直抵地平线（好吧，直抵远裁剪面）的山脉的壮观效果吧。



**2. 可预测随机数和无限宇宙（Predictable Random）**

有个传说中的游戏叫 Elite ，不知道有没有同学玩到过。据说这游戏运行在32K内存的机器上，其中16K还是只读的ROM。这游戏据说拥有难以匹敌的游戏深度：近乎无限个行星，每一个都有各自的名字和特征。

可预测随机数本身是游戏内运用非常广泛的一个技术，这里我们着重谈一下它在为游戏提供（微观上）更丰富的细节和（宏观上）更广阔的世界的作用。这一技术的最重要原则是，为了在一个游戏世界中给出无限空间的幻觉，我们需要满足两个分解条件，可以把它们成为宏无限（macro-infinite）和微无限（micro-infinite）”。前者涉及到问题的空间规模，后者则任意一个对象所支持的最小细节层次级别。

从实现上来说，如何设定随机种子是这个技术的核心。由于给定一个随机种子，生成的随机序列是完全可预测的，那么根据游戏内的一些时空的设定，通过对随机种子进行一些定制，得到在游戏内任意某个时刻和某个空间点上完全可预测的行为就是可行的了。

最简单的是使用以下几个元素与随机种子配合计算：

* + 世界坐标（即 X Y Z 值，既可以表示空间中的某个点，也可以表示某个区域）
  + 系统时间（可以用真实时间，也可以用游戏内设计者定义的时间，如果是前者的话需要考虑离线时的处理）
  + 正态分布（在游戏里建一个查找表即可，这是最廉价的方案）

这些因素加上对应的随机序列，已经可以营造出宏观上比较有深度的一个宇宙空间了。理论上，如果所有的随机性都是由给定的随机种子产生，而这些随机种子要么是游戏定义的常量，要么是查表得到，要么是均匀流逝，要么是由更高层次的随机种子生成，那么这样一层一层上溯到尽头的话，任何一个游戏内的宇宙，都可以归因到一个初始的种子，这个种子，就是决定论中经典物理学的所谓的**第一推动**吧。其实如果真做到了这一点，我们大可以把这个种子交给玩家，在首次进入游戏的时候掷一个 2^64 骰子。这是真正的上帝创世的感觉，想象一下，上帝说，要有光，于是掷出了骰子，第一推动怦然落地，整个时空的巨大齿轮开始运转，在不同的时间点和空间点上，更多的随机序列被生成出来...

微观上本质上也是一样的，只是把发散的过程倒过来，变成了逐级收敛的过程。为了在某一个点上放大时，能得到尽可能细致，准确和一致的表现，我们需要对较低层次的世界定义更丰富的规则和约束，比如黑洞对周围的影响情况，双星体系的轨道，恒星的种类与行星个数之间的对应关系，不同恒星系结构下行星表面的大气构成，等等。这样才能较好地平衡多样性和独特性，带来更真实的模拟效果和更令人信服的游戏体验。

**3. 精度问题解决方案**

当足够大尺度的世界被创建出来时，就会自然而然地遇到精度的问题。同时这也是补充说明中提到的一个问题，这里我们简单介绍一下几种实践中的解决方案。

先描述一下问题吧，我们知道，IEEE754 中规定的32位浮点数中，有23位是用来表示尾数的。也就是说，对于接近1的浮点数，可能会带来 2E-24 左右的误差（大约是 0.0000001192）以现实单位计算的话，如果游戏世界是边长为100km的正方形，那么在这个正方形的最远角落里，我们的最小空间单位是约 7.8 毫米；而如果是中国这么大的面积的话，空间误差将达**半米**左右。那么可以想象一下，如果是宇宙级别的游戏，采用32位浮点数作为全地图精度，那么实际误差可能会有多么大。

在实践中，这种误差可能会带来以下这些影响：

* + **无法将相邻的对象对齐**。这种情况，场景美术（关卡设计师）应该会比较头疼，这对于游戏的编辑器来说是大问题了。物件没法在引擎编辑器里对齐；在不同的平台上，对齐也不一样；甚至不同的编译器，同一个编译器的不同版本编出来的引擎；对齐都不一样 ... 所以说处女座不要做 LD :P。
  + **模型间的穿插和裂缝** 本来封闭的墙角可能漏个洞，本来放在地上的石头变成了悬浮在空中。这实际上是上一种的变种。
  + **骨骼动画的抖动** 由于世界矩阵往往参与骨骼动画的运算，误差可能会被逐级放大，在那些最远离根骨骼的骨头上（也是玩家最容易注意到的地方），抖动可能会发生得非常剧烈。
  + **物理模拟失真** 一些柔体的模拟可能会直接失败，而刚体也可能会产生怪异的运动。碰撞系统也可能无法正常工作。

所有这些一旦发生，都是很容易觉察的。一旦你发现在一个很大的坐标上有这些问题，而接近原点处问题却消失了，那么很有可能就是精度在作怪。而需要注意的是，出现这种问题，只和游戏中出现的数字的规模和跨度有关，和游戏选择了什么样的长度单位（如用毫米还是公里做单位）无关。

那么怎样使用有限的坐标精度来描述较大尺度的世界呢？

最直接的方案是 **使用双精度浮点数** （64 位），如果这是可接受的选择，那么就不必费心引入后面讨论的复杂度了。

其次是 **区域划分法** 。我看到 Milo 同学已提到，不过这里出于完整性的考虑，再简单说一下。正如 Milo 同学所说，“把世界划分成不同的区域，在区域内的计算使用其局部坐标系统。”相对应的跨区访问，需要对应的“本地A -> 世界 -> 本地B”的坐标转换。

还有一个方案是 **节点中转法**。正如移动电话的基站用来承载和协调整个通信网络那样，我们在游戏的给定活动区域使用静态信标，所有的逻辑上与之相关的单位，都以该信标的坐标作为参考单位，这样也可以做到把数据访问局部化。相距足够远的两个物体（相当于上面的跨区访问）交互总是通过静态信标来完成（正如移动电话网络中发生的那样），这样的好处是相关的复杂度可以隔绝在这个中转系统的内部。

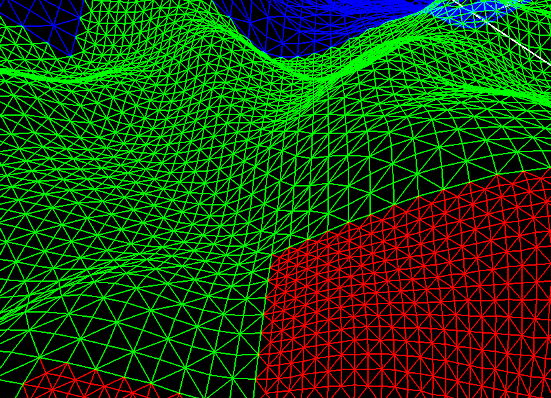
此外凯丁同学提到了一个 **坐标转换法**，“所有位置信息都以角色位置为中心做一次转换”。这正是 《Game Programming Gems IV》 [“4.0 Solving Accuracy Problems in Large World Coordinates”](http://www.gameenginegems.net/gemsdb/article.php?id=280) 文中的方案。这个方案可以解决部分问题（主要是渲染相关的问题），但是仍有一些问题需要考虑，比如：1. （上面提到的）编辑器内操作的物体，在序列化到文件中时，精度丢失的问题。2. 大部分物理模拟通常需要一个角色无关，摄像机无关的全局坐标系。等等。

**4. 超大地形的处理 （Terrain Visualization）**

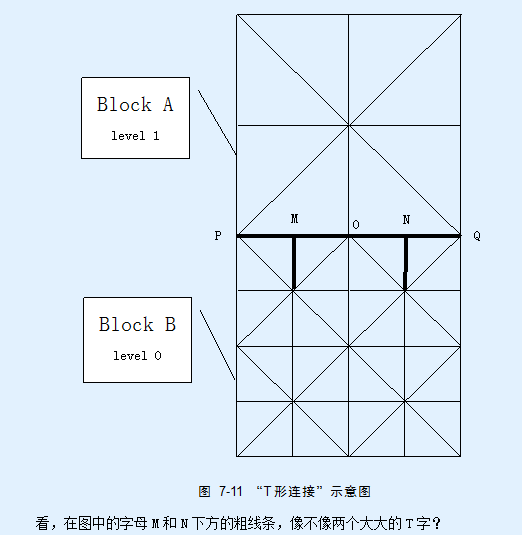
整个过程大致可以分成三个阶段：一开始，GPU处理顶点能力很弱，这个时期的各种精巧算法（如一些VDPM和后期的ROAM），**尽力在用CPU来降低顶点的总量**，避免一不留神压垮图形系统；后来，图形系统的能力上去了，人们开始更多地考虑到**把地形系统融入到通用的场景管理**中去，如四叉树八叉树什么的就是在这个阶段被广泛应用的；再往后，GPU已经很强大了，CPU由于要承担更复杂的游戏逻辑，越来越成了整个系统的瓶颈，这个阶段，人们琢磨的更多的是，怎么**利用GPU给CPU减负**了，一直到如今，由 GPGPU 带动起来的异构计算，也都是这个路数。

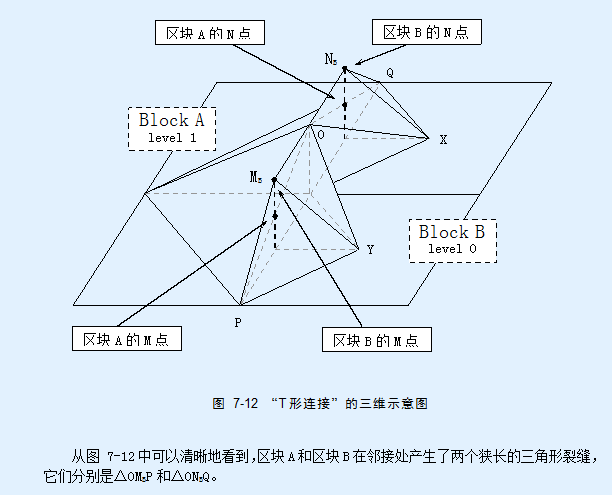
**4.1 古典算法（从 GeoMipMapping，Progressive Mesh 到 ROAM）**

**GeoMipMapping** 是从纹理的 MipMapping 技术演化来的一个地表处理技术，原理上是根据任何一小块地形在屏幕上显示的实际尺寸（主要跟与摄像机的距离和起伏程度有关）来选择对应密度的网格，然后把不同分辨率的网格之间以某种方式拼接起来（没有这一步的话就会有裂缝），本质上是一种比较粗糙的区域 LOD 算法。顺便说一下，由于那时候针对顶点级的处理很多，导致这种T型裂缝很常见，以至于有个专门的名字叫“T-Junction”，针对这个的处理在当时也有很多方案。



这是俺刚刚到老硬盘里刨出来的大三时写的 GeoMipMapping 代码，编了一下居然还能跑起来。有点土，别介意：P 可以看到不同的 MipMap 级别是用不同的颜色渲染的，也可以看到接头处 T 型裂缝的处理。唉，这代码勾起了俺的青葱回忆啊，那就顺便再来两张 T 型裂缝的示意图和消除过程吧。





**Progressive Mesh** 是后来很流行的技术 Simplygon 的前身，原理上基本也是一致的，就是以某种方式渐变性地化简某个给定的 Mesh。

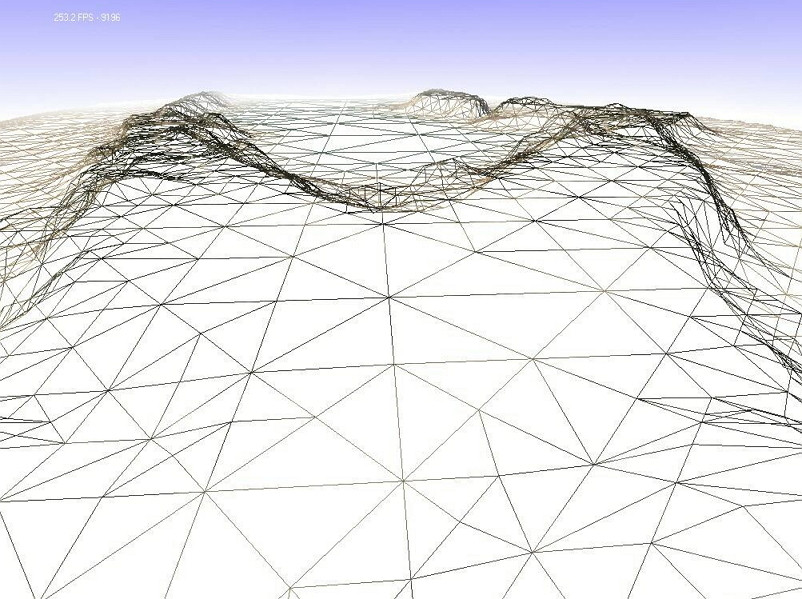
渐进式网格有两种：视点无关的 （View-Independent Progressive Mesh，VIPM） 和视点相关的 （View-Dependent Progressive Mesh，VDPM）。两者的区别就是，前者预先离线生成好所有的渐变过程，运行时直接用就行（也是后来 Simplygon 采用的方案），而后者随着摄像机的位置和角度的变化，生成对应的简化模型。两相对比，VIPM的好处是运行时运算开销低，简化模型的效果好，缺点是费内存（因为数据都存下来了，当然后来增量的方式能省一些），而VDPM在当时是不错的选择，因为跟VIPM相比不用费额外的内存，而且对于视点（就是摄像机）变化不剧烈的应用，不需要每帧处理和更新对应的简化模型（普通的MMO类的一般一秒一次就够了），此外由于一些简单的遮挡剔除和背面剔除，能够比VIPM裁掉多得多的顶点（一般能多裁1/3到一半吧，在当时这可是头等大事）。

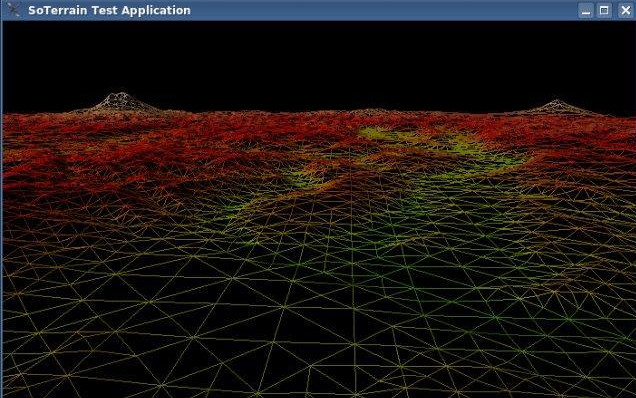
总的来说，至少在当时，两者的应用都比较广泛，而到了后来，显存越来越大，总线却越来越紧张，VDPM这种典型的刷顶点的算法（比较费总线带宽）就逐渐失去了市场，这是后话了。

大家可以在[这里](http://www.cbloom.com/3d/)看到一些 PM 在地形渲染上的应用。图咱就不上了，大家可以到 Simplygon 的网站上去看。

**ROAM** 可算作是上面提到的 VDPM 更进一步了。这个算法实际上借鉴了当时主流引擎的标配BSP的思路，想利用二叉树这个最简洁的空间描述[**数据结构**](http://lib.csdn.net/base/datastructure)，把（CPU端的）顶点消减发挥到极致。整个地表被组织成一个巨大的二叉树，有两个队列，一个是分割队列，一个是合并队列，分别用于处理摄像机移动时，增加进入视野的区域细节和消减退出视野的区域细节。精心设计的 ROAM 效果非常华丽（尤其是在线框模式下），你会看到在各种因素的影响（包括局部坡度，与摄像机的夹角，遮挡情况等等）下，各种三角形像魔术般的不断地变幻，生成和擦除超多的细节，效果非常魔幻。我印象很深的是当时连续打Quake3两个小时完全无感的我，调试这玩意的时候，每每不到十分钟眼就花了。

网上找了两张比较典型的 ROAM 大家感受一下吧。



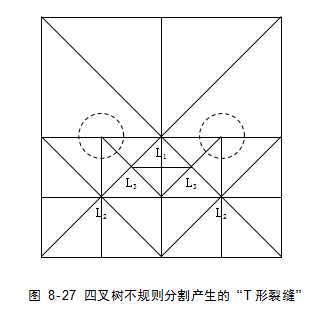


**4.2 层次的艺术（Quadtree 和 Chunked LOD）**

其实用于空间管理的树状结构有四叉树和八叉树（还有上面的二叉树），但地表通常以前者居多。是因为，从小范围来看，变化剧烈的地形是3D的，适合八叉树在xyz三个方向上扩展；但当尺度大到一定规模之后，地形通常退化为相对扁平的2D空间，就像摊平了的地球表面那样，在竖直的Z方向上变化相对不大，而XY平面则是可能无限延伸的。

**Quadtree** 四叉树很直白，具体的细节我就不讲了。值得一提的是四叉树往往也同时用于场景管理的快速剔除和查找，从理论上来讲，四叉树是一个平面上最迅速的用于剔除空间，定位一个物体，内存开销也是相对较低的数据结构。当用于地形渲染时，顶点剔除的效率也很高，我印象中仅次于高度优化的 ROAM。内存开销低主要是因为四叉树是可以完美展开到一个位数组里的，这样的话意味着整个树的利用率达到了百分之百——所有的空间都用来存储数据而不是维持结构。

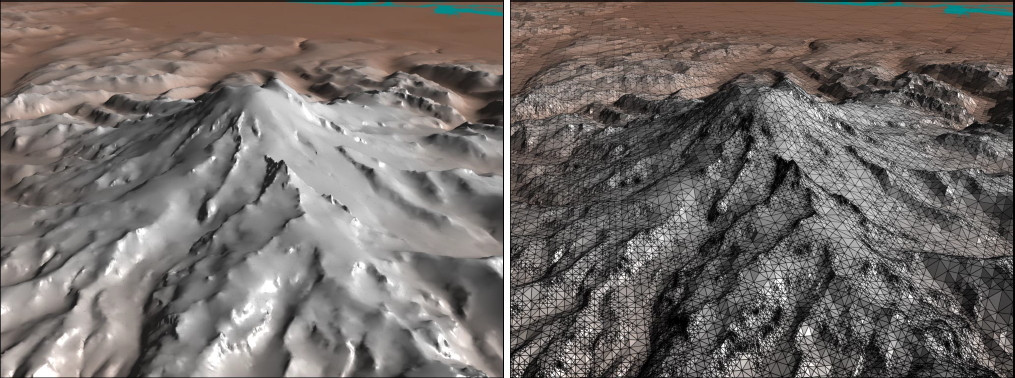
不过四叉树也不是啥都好，T型裂缝就比 GeoMipMapping 难处理，因为存在跨级的多段 T 缝，如下图：



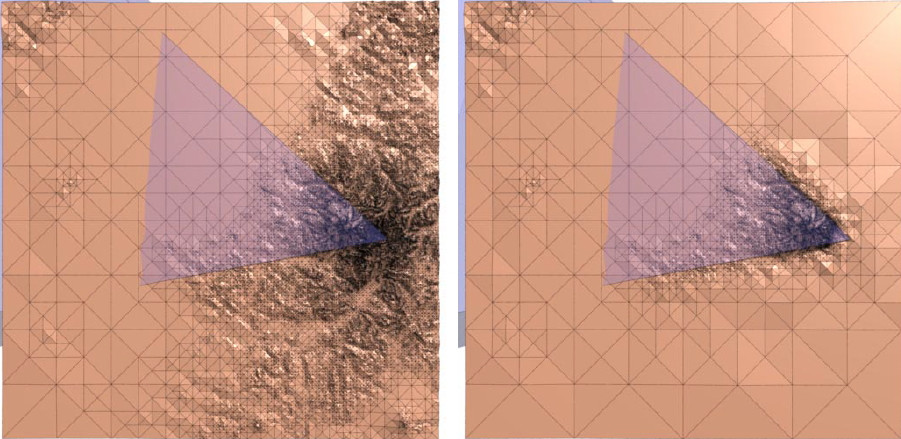
除此之外还有一些细节问题，这里就不一一说明了，地形的四叉树渲染还是有很多细节需要细心处理的，此处暂且放下不表。

**Chunked LOD** 是一种杂合改良的 LOD，其实糅合了上面说的不少细节，本质上是一种分区块地消减细节的技术。所谓 Chunk 是批量处理的一种方式，只是一种粒度划分的单位而已，跟现在 [**Java**](http://lib.csdn.net/base/javase) 的 GC 里分区回收概念上差不多。

下面是典型的 Chunked LOD 后的效果：



顶点多次过滤优化后的效果：

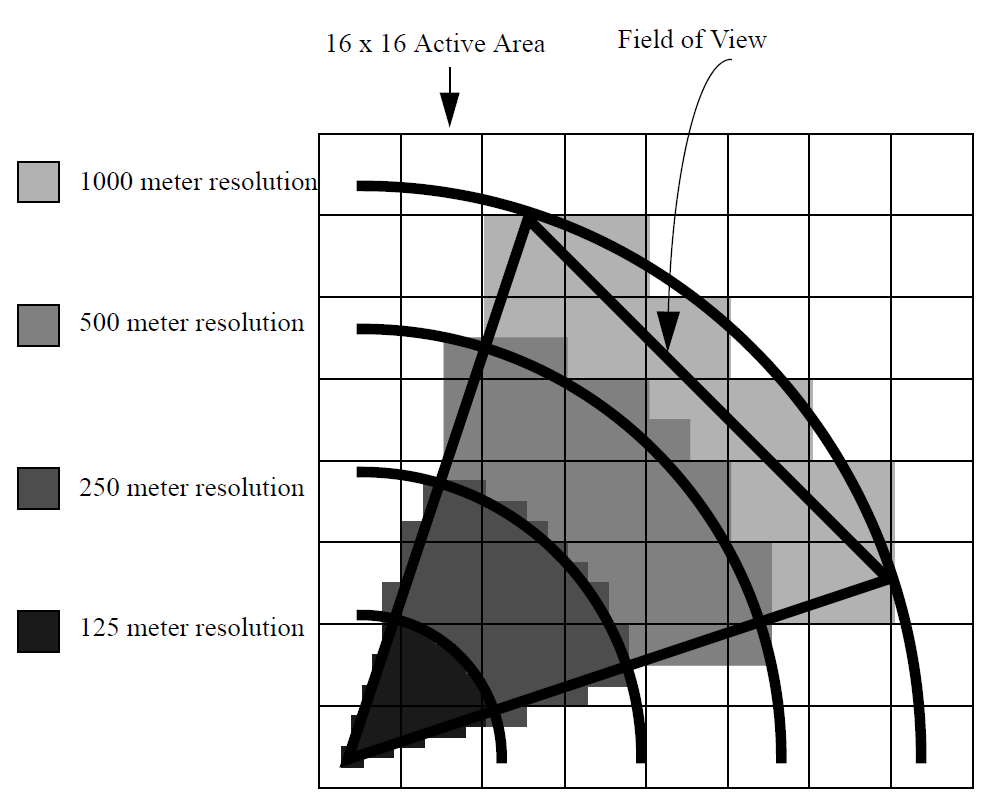


效果在当时还是很惊艳的。通常不到50k的渲染数据量已经能有非常逼真的效果了。

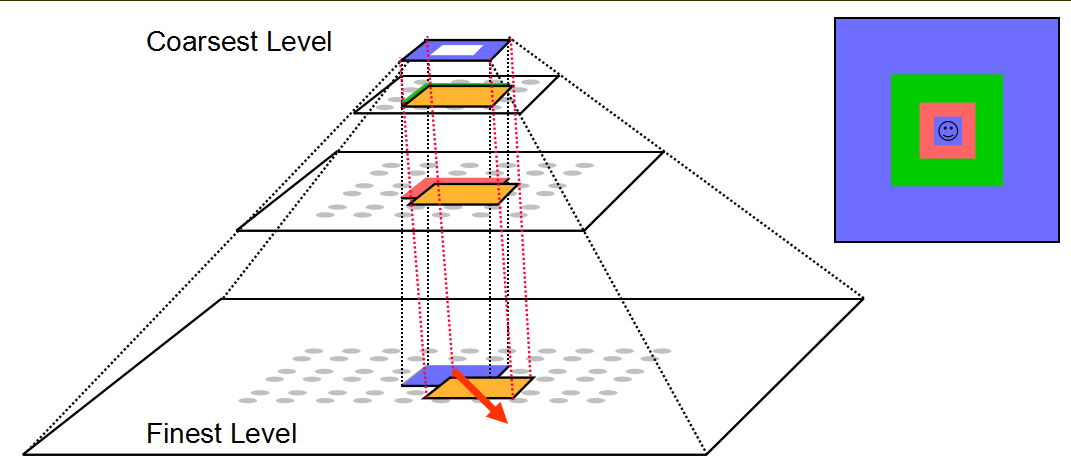
**4.3 以GPU为主的技术（从 Paging，Clipmap 到 GPU Terrain）**

上面的基本上都是传统方案，这一节我们将逐渐过渡并挨个介绍一下以 GPU 为运算主体的算法。

所谓**分页**（Paging）实际上是仿效虚拟内存的运行机制的一种方法。由于地表的顶点数据都是静态数据，适合常驻显存。当世界尺度较大时，显存没法一次放入所有数据，那么系统就像虚拟内存那样，把那些暂时没有用到的数据交换出去。随着游戏的进行，Paging In/Out 也在不断进行，辅以一定的异步机制，加载到显存的延迟可以被很好地掩盖。玩家的直观感受就是：哇，海量的细节。

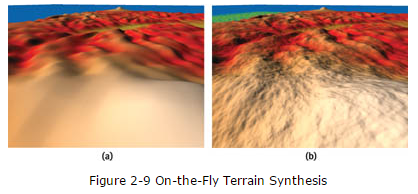


而 **Clipmap** 则比 Paging 更进一步，以金字塔的形式逐级把数据排列好，直接整体更新和渲染。从这里也可以看出 GPU 时代人们的思维方式的逐步变迁。从以前顶点级别（Vertex Level）的“锱铢必较”，到后来的一次多塞一点也无所谓，只要批次（Batch）少就 OK。下图可以看出 Clipmap 的基本思路。



所谓的 **GPU Terrain Rendering** 就是把高度图从内存里经由 2D Vertex Texture 搬到 VS 里去生成三角面，这样的好处是 CPU 和内存就被彻底解放出来了。只是访问上有一些限制，不像直接处理内存那样方便。具体的细节可以看这里：[GPU Gems 2: Terrain Rendering Using GPU-Based Geometry Clipmaps](http://http.developer.nvidia.com/GPUGems2/gpugems2_chapter02.html)

在 GPU 上做还有个巨大的好处是可以借助 Gaussian Noise 即时生成更多的细节了。直接拿一小块预生成的高斯噪点图在需要的时候叠加一下，就能在没太大额外开销的情况下，增加各种细节。如下图所示：



随着大家对 GPU 理解的深入，地形的处理又有很多的小技巧可以做，尤其是在 PS 里面，比如法线生成，动态uv展开，光照按需叠加/衰减什么的。不过呢据我所知没有什么非常别具一格的架构上的新思路了，所以就不再深入了。

**5. id tech 5 的 megatexture （超大地表上的非重复性海量贴图）**

**megatexture** 在当年（2007）是一个非常值得一提的技术。在这个技术出现之前，几乎所有的地表渲染用到的贴图都是若干张 blend 到一起后，以 tiling 的形式重复平铺在地表上的（包括比较典型的魔兽世界也是如此），这样的直接后果是图片的种类用多了耗资源，用少了又很容易感到单调和重复。而 megatexture 则是一张全局的超大贴图，从根本上避免了重复这个问题，理论上（实践上也是）能够生成非常壮丽和独特的地质风貌，是传统的刷地表无法创作出的效果。可以说这个技术让**真正的全景地貌**成为可能。

技术上的细节puzzy老师写过一个 pdf，强烈推荐感兴趣的同学搜来看一下（可以搜“ **ID Tech 5 中"Megatexture"针对地形的D3D9 基本实现原理 - 姚勇**”），珠玉在前，我就不啰嗦了。就来一张效果图吧（好吧我知道能坚持看到这儿的同学，这图基本上肯定都看过了）



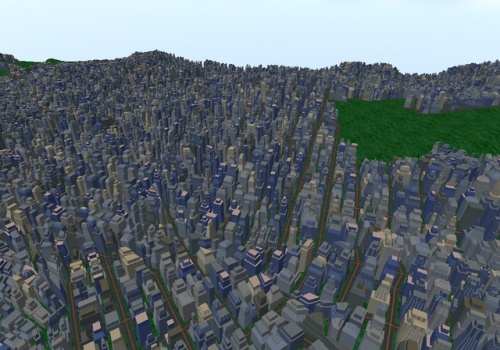
全局超大贴图对一个开放性世界的价值不言而喻。想象一下，跟拿乐高积木拼接出来的视觉效果（传统的 texture blending and tiling）相比，一幅万米画卷上，每个像素都可以随意描绘，是一种什么感受。 比如，你可以相对轻松地实现“整个世界的地貌瞬间被密集核弹蹂躏了一场之后”的效果。如果你想模拟整个生态环境的变迁，在不同粒度上的整体性修改更是无价之宝。

**6. 过程式的内容 （Procedural Content Generation）**

“过程式生成”是一个不是很恰当的翻译，实际上更贴近本意的说法是“以程序的手段生成”，这里我们简洁起见，仍使用过程式生成的字样吧。

过程式的内容生成是随机技术的在视觉效果上的一个重要衍生。这个技术虽然到最近才被广泛应用，但实际上从技术角度讲，在很久以前就已经有比较成熟的实现了。我手头的 2003 年出版的翻译版 Game Programming Gems III 中 就有 4.16 和 4.17 连续两篇文章以“程序手段生成的纹理”作为主题。这是构建超大规模的世界的一个重要的技术工具，尤其是与上面的 megatexture 技术结合起来，可以创造出非常令人震撼的视觉复杂度。

下面是 sourceforge 上一个开源的项目 [PCity - Procedural Modeling and Rendering of Cities](http://pcity.sourceforge.net/)



可以看出，对于过程式生成来讲，只要有非常小的初始数据集（meta-data），可以在宏观上达到很大尺度和复杂度的视觉效果。

过程式生成有两大分支，一个是过程式纹理，另一个是过程式建模（上面的 PCity 属于后者），下面我们分别来谈一谈。

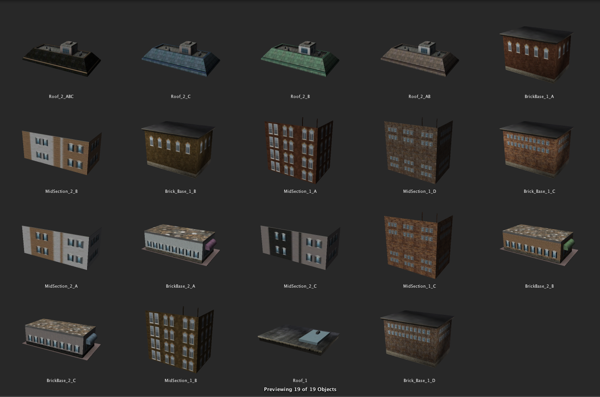
**6.1 过程式纹理（Procedural Texturing）**

人们发现，自然界中有很多视觉效果是可以用数学公式加上一些简单的随机性来模拟的，比如云彩，水流，火焰，木纹，大理石，草地，夜空，大气等等，程序生成的纹理效果大大丰富了普通纹理能表现的效果，就好像是物理引擎给游戏增加了活力一样。一个普通的噪点图，在不同的情境下，作为辅助参数来参与生成动态纹理，可以产生出近乎无穷无尽的变化。

这是过程式生成的云，出处在[这里](http://www.indigorenderer.com/content/cloud-render)。



这是过程式生成的外观，使用了 Allegorithmic 公司的 Substance Designer，出处在[这里](http://blog.mediarain.com/2013/03/procedural-textures-using-allegorithmic-substance-designer/)



这里是一些分解材质，相当于过程式纹理的图素，出处在[这里](http://www.3dtotal.com/team/Tutorials/yann_vaugne/alientut_1.asp)。





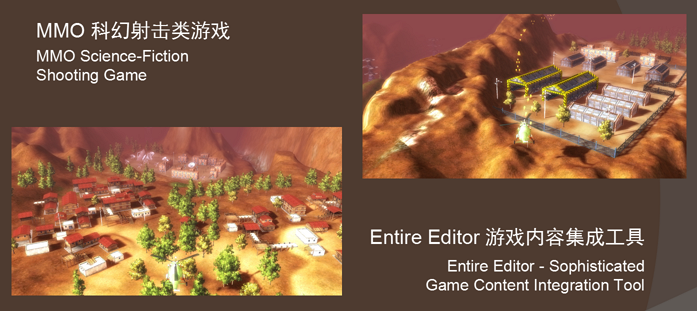
**6.2 过程式建模（Procedural Modeling）**

过程式建模特指以程序的手段动态建模。这是一个更大的话题，现在比较成熟的中间件的代表是 Speedtree，比如下面这个效果：



完全不同风格的纹理，模型的任意杂合，随意生成，效果也非常真实，非常适合做恐怖游戏。在 Speedtree 的网站上还可以看到长成茶壶的树之类的奇葩。我还记得有一年的GDC，在 IDV 的 Speedtree 的展台看到的一段华丽视频，就是各种藤蔓植物在几秒钟之内长满了一个峡谷内的整个大坝，电影级的效果非常震撼，不知道现在网上是否还能找到。

过程式建模是一项非常迷人的技术，我本人也曾被深深吸引，在上面投入过一段时间的精力。2010年时，我在开发一款飞行射击类的 MMO，当时接触到了 [Gamr7](https://www.linkedin.com/company/557710?trk=prof-exp-company-name) 的过程式建模技术，感觉很不错，在飞行类游戏中，地面物体的建模可以完全以程序方式生成，这个对当时的我来说吸引力太大了。那时我花了一个月把 Gamr7 的技术集成到自己的框架里，并在上海世博会期间，与 [Bernard Légaut 先生](https://www.linkedin.com/profile/view?id=518127) 一起在世博会的法国企业馆展示了合作成果。摘两张当时的 PPT 吧。



截图中的素材基本上都是使用了过程式自动生成的（不是美术手放上去的），树是用 speedtree 生成的。



总得来说，过程式建模是一项**潜力远远没有得到释放**的技术，现有的工具还处于比较原始的阶段。在当年 PPT 的技术展望（Beyond the Tech）一节中，我写到“（过程式建模带来的）**更高级的抽象使我们可以控制更高的复杂度，从而带来更丰富的细节** (Higher abstraction makes much more details and complexities manageable) ”时至今日，受限于技术的发展，这仍只在某个特定的主题（如 Speedtree 的植被模拟和一般的城市模拟）内有效。对于随机性的粒度，我们仍缺乏有效的手段去控制。当年展望时的两大 Expectation（一个是建立起模式和库抽象从而满足不同层次上的复用需求，另一个是如何统一过程式技术中的无序和有序，有效地控制随机性的粒度），现在据我所知仍是所缺甚多，颇为渺茫。当然了，对有志之士，这也不失为一大探索方向。

**二、内容制作篇：设计和创造（Content Design & Creation）**

聊完了程序方面的内容，我们来简单讲讲超大规模世界在设计和制作方面的一些情况。这方面因为我不是专家，只是做一下简单的介绍，不足之处，还请专业人士指正。

**1. 随机地图类游戏 (Diablo II) 中地图的拼接**

在暗黑二，暗黑三和类似的游戏“火炬之光”等游戏中，通过巧妙的拼接，理论上可以通过组合，生成近乎无限大的地图。

这是暗黑三第二章里所有地图的可能的部件形状：



这是拼接之后的样子：



除了拓扑结构上可以任意排列组合以外，在每一个分片上预留足够多的通用接口也很重要。比如一扇拱门，可以是闭合不可交互的状态，也可以通向下一个直角走廊，也可以是通往一个副本的入口。



要注意标准化的组件如果太多也会让玩家觉得单调和重复感过强，火炬之光在这一点上就做得不错。下图是火炬之光生成好的地图样貌：



可以看到效果还是很不错的，一眼看过去已经不太有重复感了。

**2. 无缝大世界 (World of Warcraft) 中区域地图的拼接**

无缝世界类游戏的区域拼接和上面的随机生成类游戏的区域拼接是很不一样的。



可以看出，不同的区域之间有着很长的接壤线和完全不规则的边缘。在这种情况下，为了简化制作，大部分边界区域以高山作为阻隔。你几乎不怎么会看到有建筑横跨两个不同的区域，原因也是在此。

在沙盘制作时，通常的做法是在整个世界地图（对应的世界编辑器）中规划好每个区域的范围，也就是分区分块。每个区域由不同的设计师在场景编辑器中分开制作。在制作当前场景时，与该场景接壤的几个场景的最新版本都会加载上来，编辑器中可以提供一些方便的工具，便于在接壤处对齐。主要是高度的对齐和贴图的融合。前者通常的选择是高度上用 Smooth 工具平滑过渡到邻接场景，后者通常最简单的处理方法是真正的过渡点两边使用同一种贴图即可，实际的美术风格过渡（如果需要的话）在邻接地图的接壤线附近完成。

一些贯穿不同地图的元素（如河流等）可能需要世界编辑器的参与来指定水平面的高度和区域范围之类的参数，但这一类元素通常不会太多，因为它们没有明显的 Gameplay 贡献，反而加剧了不同场景之间的耦合。

如果游戏有动态的天气/环境氛围系统，那么不同场景之间需要做一些平滑的过渡，这些程序用普通的插值实现就可以了，设计师一般只用关心当前场景内的表现即可。当然有的游戏这个过渡做的不好，玩家体验非常生硬，也是有的。



总得来说，这一类无缝大地图的复杂度主要是在编辑器的协同方面（后面我们会再提到）,实际的创作复杂度较前面的随机地图生成为低。

**3. 卫星地质数据的导入，规整化和再加工（一些飞行模拟类游戏）**

超大规模的开放性世界地图，还有一类是直接使用卫星地质数据以加强整个世界真实性的。据我所知，育碧出品的 Tom Clancy's H.A.W.X. I & II （就是国内翻译的 鹰击长空 I & II）就是使用了 GeoEye 的商业级高分辨率卫星地图。

既然用了真实的卫星地质数据，这一类游戏同样能生成非常震撼的效果，也就不用多说了。找两张截图大家感受一下吧：



这一类的缺点是不能模拟真实世界中没有的环境（当然拿去再创作的不算）。



这种真实数据的数据源就没什么好说的了，我简单说一下导入后的处理。通常导入后的贴图需要美术在色彩和明暗上二次加工一下，得到和游戏匹配的整体氛围。需要提供比较精确的工具给美术进行高度图和高分辨率纹理的拟合。此外通常这一类地质数据是没有 NormalMap 的，需要提供工具生成一下。

还有就是，河流和湖泊这一类水体的处理，3D游戏通常在渲染效果方面对水面特效照顾得比较多。如何生成跟真实环境相匹配的河流和湖泊是一大难点。因为一般游戏里是有一个绝对水平的特效面片的，而如果给真实环境中得来的高度数据上配一个这种特效面片，通常无法跟真实的贴图相吻合（尤其是在山脉和峡谷等地形中的水体）如何提取真实的高分辨率贴图中的水面信息，自动生成对应的3D水面，也是一大话题。当然，如果不怕费事，也可以由美术直接做出来了事。

**4. 超大地图的协同编辑：并行操作，数据同步，手动和自动锁的运用**

现在咱们回过头来聊一聊在 wow 这一类超大地图里，如何在多人团队内协同编辑的问题。

对于美术（这里专指负责场景的设计师）来说，常见的最简单做法是每人一块（或多块）区域地图，团队内维护一个公共的物件和贴图库。（物件和贴图可以由专门同事制作，需要时也可外包）在这种情况下，美术的并行化程度很大程度上取决于团队自身能力，对场景编辑器没有特殊的技术上的需求。

超大地图的场景编辑器在加载周边邻接的区域地图时，需要显式地标示出其版本和上次修改日期，这样可以把邻接区域的修正工作量降到最低。最好的做法是能够通过版本管理软件，在有同事修改了邻接区域以后自动更新并重新加载（当然可以稍有延时，不用那么即时），这样的并行工作效率可以达到最高。

真正的难题通常发生在策划那边，当需要编辑跨区任务或事件时，如果对 Ownership （也就是场景实体的所有权问题）管理不善。跨区系统可能会产生各种层出不穷的 bug。比如同一个 npc 承担了多个跨区职责时，其中的状态就可能会互相干扰，在杀掉某个 npc 这一类任务中更易出现，造成难以重现的 bug。这就需要提供明确的所有权管理机制，明确跨区访问的一般规则，提供简单的全局状态检测工具，在设计时就能提示出绝大多数潜在的冲突。当然，这些的先决条件是所有的区域数据，要么提供中央[**数据库**](http://lib.csdn.net/base/mysql)管理，要么工具做到在团队网络内实时同步。

最后我们来说一下真正有趣的实践，就是**“真正的”协同编辑**。也就是任意个美术和任意个策划可以工作在任意个区域里。是的，你没看错，这才是终极的开发自由度。其实如果这是一个如典型的 WOW 那样的 MMORPG 的话，这就跟“所有的玩家登录到同一个服务器一起游戏”是同一个概念了。所不同的是，这里的“玩家”实际上全部是开发团队的成员，而且他们是有能力创造和修改这个游戏世界的。

只要想通了这个概念，实践上并不像想象中那么复杂。由于美术和策划对同一个地图关注的焦点不同，我们只要把他们工作有交集的部分处理好，他们就能一起愉快地玩耍了。实践上来看，两者的交集通常是 a. 整个区域的逻辑高度图和 b. 所有的相对碰撞关系。也就是说，美术和策划的工作内容里，只要不涉及到这两者，都可以随便搞，但如果影响到了这两者，编辑器需要有能力提示正在修改的人会影响到什么（或按默认行为自动处理），通常是不难做到的。举个例子，策划放好 npc 后，美术去调整高度，把 npc 站的广场弄成一个山坡，那么要么提示美术这么干可能会影响到策划的设计，要么自动把对应的 npc 都重新调整位置，吸附在新的地表高度（一般编辑器允许设置为吸附到地表）。

当两个美术在修改同一区域时，就涉及到锁的问题了。锁有两种，一种是在编辑时自动触发，场景地表以区域为单位，物件以 Instance 为单位，当编辑时自动把 Owner 设为当前编辑者，提交改动到服务器时可以选择继续锁或是释放锁。一种是手动锁，就是美术框住一片区域，主动加锁，这样有些时候更方便。编辑器制作者需要考虑的一些细节有：锁住的区域在其他开发者的机器上，需要比较显眼的提示信息；保险起见总是多锁一定的范围，以方便地表平滑等工具编辑时对周边区域的影响，等等。

**三、异次元篇：我们的征途是星辰大海**

**1. 终极沙盒（EVE）：当规模大到一定程度——宇宙级别的混沌理论与蝴蝶效应**

对于开放式世界来讲，如果没有真正与这个世界的尺度相配的开放式的交互，那么仍然是一个死气沉沉的世界。EVE，正是一个为了开放式交互而打造的超大的沙盒宇宙。

在这个宇宙中，玩家拥有很高的自由度去探索，创造，建设，摧毁（针对自然环境而言），配合，领导，同盟，背叛（针对社会环境而言）。这游戏我就不展开介绍了，有兴趣的同学可以去看一下 [EVEWiki](http://evewiki.tiancity.com/)。有趣的是，当沙盒大到一定程度时，它会在很多方面展现出一种自平衡的性质，就像经济学中那只“看不见的手”，自然生态学中地球这个大型生态系统的自我调节和自我修复。在我看来，这也是开放式游戏的最大的魅力之一，也让系统的复杂度进一步接近真实世界。

**2. 打通两个宇宙（EVE & Dust）：发现更广阔的世界——宇宙沙盒游戏和行星射击游戏联动**

EVE 的制作商 CCP，是一个来自冰岛的很有趣也很有追求的工作室。在 EVE 的大尺度宇宙成功地运行了若干年后，他们选择了一个更大的挑战——设计另外一个大型多人在线游戏，把新老两个宇宙之间联系起来，让两个游戏内的玩家可以互动，相互交谈，配合，雇佣，指派任务，火力支援或其他的互动，最终打通两个宇宙，让两个大型多人在线游戏之间达到有机的协同和交互。