我们以前曾发现，在三角光线投射的情况下，短堆栈比全堆栈要慢一些，而在体素情况下，这种更为明显。原因是在这两种情况下，堆栈的用途不同。

对于三角形光线投射器来说，堆栈就像是要访问的节点列表，每次弹出都会移除最上面的一项。而对于体素，堆栈用于存储通往当前体素的路径，弹出的层级取决于射线在遍历时穿过的边界。例如，八叉树中间的轴对齐平面。当射线穿过其中任何一个平面时，我们需要一直弹出到堆栈的第一个条目，即最底层。与通常的弹出操作相比，这种随机访问方式更不适合短栈方法。

重新使用光线投射堆栈。这一优化与二次射线有关。由于光线投射堆栈总是包含通往当前体素的路径，因此可以将主光线的最终堆栈用作次光线的起点，前提是次光线的起点是主光线的终点。

在理想的情况下，这可以按原样工作，但在实际操作中，有必要将辅助射线的起点稍微移动一下，以考虑到误差。事实证明，只要偏移方向与射线方向的坐标八度相同，就可以解决这个问题，但偏移到其他八度就比较复杂了。因此，唯一实用的偏移方向是次射线的方向，幸运的是，这在大多数下都没有问题。不过，偏移量越大，堆栈的重复使用就越少。

我们仍不能完全确定堆栈重用为何不能加快渲染速度，但我们观察到，虽然辅助光线的迭代次数确实减少了，但整体性能却下降。一个可能的原因是，辅助光线都从自己的堆栈开始，这降低了整体的一致性。此外，体素层次结构中的初始下降非常连贯，而且成本很低。这样看来，即使为处理次射线起点偏移所需的额外代码，也是不值得的。

属性插值。我们花费了大量精力，试图以 GPU 的常规图形流水线处理纹理的方式，重新移动表面的块状结构。最初的插值属性获取实验令人沮丧--寻找邻域、解码和插值属性比实际的光线投射更昂贵。

由此可见，属性存储需要一定的冗余度，这样才能避免昂贵的邻居搜索。经过多次实验，我们最终确定了一种方案，即每个非叶子体素都有一个 $3 \times 3 \times 3$ 的属性点网格，每个网格都可能有人或无人。这些点对应于子体素的角，而父体素面上的数据点会在相邻体素之间复制，因此无需查找邻居。

这种方案每个体素产生大约 4.5 个属性点，这是一个相当大的扩展因子。如果使用智能压缩方案，例如尽可能每个体素使用少于 8 个角，就有可能降低这一系数，但即使在最佳情况下，似乎也至少需要两倍于理想数据量的数据。

在这种结构中，属性查找速度相当快，因为不需要进行邻域搜索。不过，基于块的纹理压缩并不容易使用，因为压缩的自然单位是一组 8 个子节点，其中可能会占用多达 27 个属性点。最后，属性的构建变得非常困难。要保证相邻属性点之间的连续性并不难。

但要在不同的体素层次之间实现连续性却很困难。

目前看来，使用后处理模糊来平滑曲面要比执行插值属性查找更好。在不进行插值的情况下，内存中可以容纳更多的数据，而且在很多情况下，这种方法比插值更能提高质量。后处理模糊的另一个优点是可以平滑大模糊半径区域之间的轮廓，而这是插值无法实现的。

轮廓模糊在轮廓线出现之前，我们曾试图通过简单的模糊处理来掩盖剪影块状感。事实证明，这比最初想象的要困难得多。一个明显的问题是，模糊只能从表面向外进行，因为由于遮挡，我们无法知道轮廓后面是什么。这样做的后果是，当物体离摄像机越来越近时，就会显得越来越厚。对轮廓周围进行模糊处理还会导致本应非常清晰的细节变得模糊，例如在轮廓外就能看到的远处几何体。

我们尝试了各种方法来消除这些不必要的影响，但都没有找到合适的解决方案。最终，轮廓线以整洁、内存友好的方式消除了隐藏轮廓块的需要，同时提供了比立方体体素更好的几何分辨率。

7 数据管理

由于我们正在处理的数据集可能会大于 GPU 内存可容纳的容量，因此我们的数据格式被设计为适合按需进行局部分辨率调整。如第 4 节所述，当与摄像机的距离增加时，分辨率要求会迅速降低，我们希望能够利用这一点。

磁盘存储我们的八叉树数据是以*切片*的形式存储的，这些切片保存在一个文件中，该文件一个切片目录。一个切片包含八叉树特定立方体部分中一个分辨率级别的体素数据。为便于按需流式传输，我们将一个切片可包含的数据量限制在 1 MB 左右。切片是按层次结构组织的，如果一个切片的下一个分辨率层次所消耗的内存超过了允许的容量，该切片就会在空间上被分割，使其拥有多个子切片。图 20a 展示了在磁盘上以切片形式存储八叉树的情况。

没有切片内容的切片层次结构很小，可以保存在内存中。因此，在特定空间区域找到包含所需分辨率数据的切片非常有效。在我们的实现中，所有数据更新都由 CPU 执行。

切片的内容与内存中块的内容有很大的不同，因为每当我们要加载一个切片时，我们的内存中已经有了它在八叉树中的所有上层体素。特别是，我们已经知道切片将包含哪些体素，因为父级包含了这些信息。因此，我们不需要在切片所包含的体素中存储任何空间位置、指针或索引。

内存存储。如第 5 节所述，八叉树的加载部分*以块*的形式存储在 GPU 内存中。实际上，由于分辨率的动态变化，我们需要两种块。最高级别的八叉树存储在*主干中*，主干是一个可以轻松分配和释放子描述符的池。主干以块的形式存储，这样光线投射器就不需要对主干中的体素进行不同的处理。

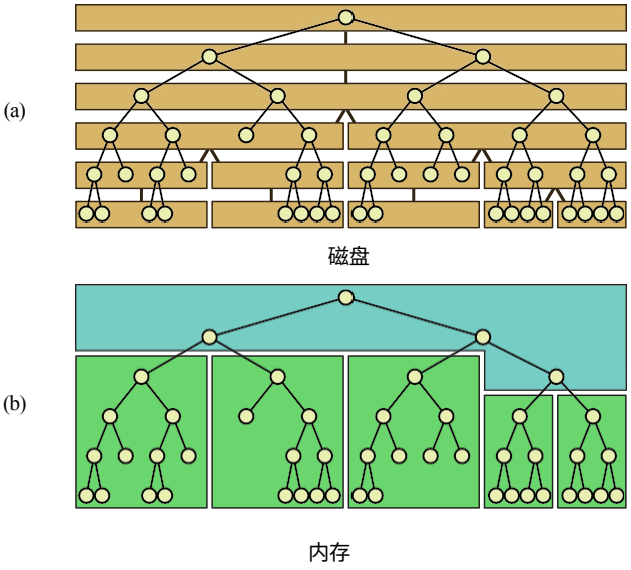

图 20：*磁盘和内存中的八叉树数据。(a) 八叉树以片段集的形式存储在磁盘上。每个方框代表一个片段该片段最多只能包含给定数量的数据。方框之间的线表示片的层次结构。(b) 在内存中，八叉树以一组块的形式存储。顶部是主干，底部的方框代表叶块，在叶块中可以添加新的切片。当叶块需要拆分时，其部分内容会被移至主干。*

在分配主干中的子描述符时，它们总是为远指针和所有可能的属性留出空间。这使得主干块非常灵活，因为加载和卸载分片往往需要对主干块进行小幅修改。由于分支因子高、块大，任何时候内存中都只有少量的主干块。

绝大多数体素数据都存储在普通块或叶块中。叶块中的体素在内存中按深度优先顺序排列，使其尽可能紧凑和高效。图 20b 展示了八叉树在内存中以块的形式存储的情况。

加载切片让我们考虑这样一种情况：内存中有一棵八叉树，我们希望通过增加一个额外的解析级别来扩展一个给定的块。现在，让我们假设切片没有被分割，即只有一个子切片。我们首先找到包含八叉树所需部分的切片，然后从磁盘加载切片数据。我们还将相关数据块从 GPU 传输到主机内存。然后，我们将数据块与来自切片的数据合并，生成包含所需额外分辨率的结果块。最后，我们将结果块传回 GPU 内存。

在任何时候，我们都要在内存中的非主干块和磁盘上的下一级切片之间保持一一对应的关系。为了实现这一点，我们必须能够在加载片时分割块。如果加载的片段有多个子片段，我们会为每个子片段构建一个结果块，从而分割块。这样可以确保每个结果块都有一个与下分辨率相关的切片。当一个区块被分割成多个区块时，原始区块根部附近会有一个或多个体素不在任何结果区块中。这些体素将被转移到主干中。请参见图 21。

动态加载。我们使用简单的启发式方法来动态选择要加载到内存中的片段。根据摄像机到图块的距离和图块的分辨率，我们可以大致确定要加载到内存中的片段。

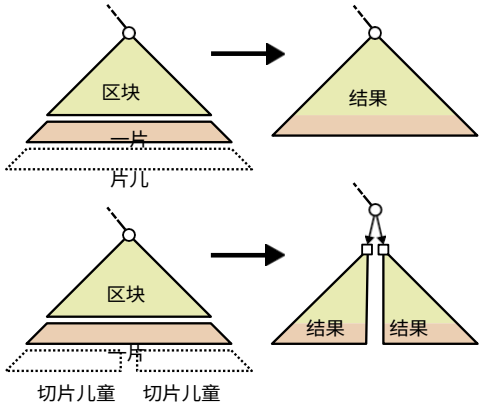

图 21：*加载片段的过程。上图：当加载一个只有一个子块的片段时，会用片段数据对块进行扩充，并生成一个结果块。下图如果片段有多个子片段，则每个片段生成一个结果块。原始块根会被移至主干。*

交配屏幕上叶子体素的大小。我们的启发式方法试图利用所有可用的内存资源，保持恒定的体素与像素比例。对于内存和磁盘中的每个切片，我们都能计算出一个分数，近似表示该切片对当前摄像机位置的质量影响有多大。切片按照分数决定的顺序加载。当 GPU 内存已满时，只要能提高总分，就会从内存中卸载切片，为新切片腾出空间。异步 I/O 用于将多个请求流式传输到，从而提高总数据吞吐量。

我们使用定制的内存管理器来分配 GPU 内存的大舞台，并处理块的分配和释放。由于加载和卸载大小可变的块，内存空间将逐渐变得支离破碎。当碎片导致内存分配无法成功时，就会对内存块进行组合，以获得更大的连续空闲空间。为避免出现重大故障，压缩器在任何时候都只移动必要的数据。

7.1 体素层次结构构建

八叉树的层次结构是以自上而下的方式一个切片一个切片地构建的。我们在切片层次结构中需要继续构建的地方插入特殊的*未构建*切片。未构建切片包含构建切片体素所需的所有数据，即构建*数据*。切片之间不存在任何依赖关系，因此可以同时使用所有 CPU 内核进行构建。构建数据包含每个体素的以下信息：标志（REFINE-GEOMETRY，REFINE-ATTRIBUTES）、体素位置、与体素相交的输入三角形和位移原点（第 7.4 节）的索引，以及最重要的祖先轮廓。标记表示体素的形状或属性是否超出了给定的误差容限。因此，在启用可变构建分辨率时，这些标志中至少有一个总是处于激活状态，否则体素就不会有问题，也不需要进一步细分。

建构开始时，它会获取输入网格，并创建一个覆盖所有输入三角形的未建构根切片。当生成器处理一个未构建的切片时，它会读取构建数据，并用实际的体素数据替换切片的内容。一个或多个子切片也会根据各自的构建数据被创建。对建构数据中单个体素的处理包括过滤三角形和分解三角形。

放置基元到 8 个子体素，为它们确定适当的轮廓（如果标志 REFINE-GEOMETRY 要求）和属性（如果设置了 REFINE-ATTRIBUTES），最后为子体素创建构建数据条目。

每个未构建的片段都包含如何将片段分割成子片段的说明。允许一个切片被分割成八个以上的子非常重要。否则，我们就需要对最坏情况下的数据扩展进行预处理，这反过来又会切片的平均尺寸非常小。举例来说，我们不分割切片的情况下提高了许多层面的体素分辨率，因为数据在大尺度上是低维的。因此，体素水平和切片分割水平之间存在很大差距，使得切片的空间尺寸大于其所包含的体素尺寸。现在，假设数据开始以容积方式表现，因此每个层面上的体素数量增长了八倍。考虑到数据是大尺度的低维数据，分割一次切片并不会像提高分辨率那样减少数据量。因此，每个切片的数据量可能会无节制地增长。

尽管这个例子听起来有点矫揉造作，但在实际操作中问题还是很严重的，除非允许分割到 8 个以上的子片中。需要注意的是，父分片必须在看到子分片建立后实际包含的数据量之前，决定其每个子分片的分割策略，因此有可能出现估算超限，生成的分片比希望的稍大。

7.2 轮廓构造

为了简化用等值线逼近给定曲面的工作，我们可以看到，逼近的结果并不一定要光滑。只要我们确保原始曲面被每条轮廓线完全包围，就能保证得到一个不含任何漏洞的近似曲面。虽然体素边界上的不连续性可能会在光线追踪中产生错误的自阴影或相互反射等问题，但这些问题通常可以通过少量偏移辅助光线的起始位置来解决。因此，构建过程可以定义为最小化每个体素的初始范围，而不考虑其相邻体素。值得注意的是，即使我们不强制要求平滑，所得到的轮廓表面也会相对平滑。

我们采用一种贪婪算法，以分层自上而下的方式为每个体素构建轮廓。构建的基础是给定体素内包含的原始曲面，以及已经确定的祖先轮廓。我们首先利用体素立方体与其每个祖先轮廓线之间的交点来构建多面体。然后，我们选取一些候选方向，确定多面体在每个方向上对原始表面的高估程度。如图 22 所示，高估的计算方法是沿给定的候选方向，多面体与原始表面的空间范围之差。最后，我们选择高估值最大的方向，并在该方向上构建一条与之垂直的等高线，使其尽可能紧密地包围原始表面。

由于构建过程的贪婪性质，所得到的近似结果的质量在很大程度上取决于所选择的候选方向集。由于我们的层次结构数量有限，我们希望避免选择那些只会局部缩小空间范围，而不会对叶体素的最终形状产生影响的方向。因此，我们将候选方向集限制为原始表面的正负方向以及表面边界的垂直方向，因为这些方向最有可能对最终形状做出贡献。在实践中，我们发现只需包含

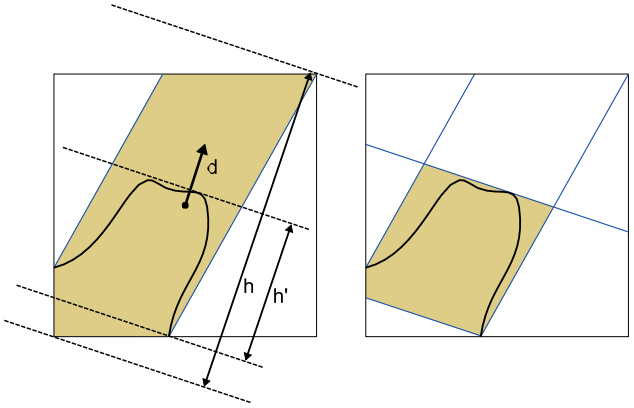

图 22： *合作导览构建中候选方向的评估。左图：一条轮廓线已经确定（蓝线），下一条轮廓线正在考虑候选方向* $\mathbf{d}$*。最外侧的虚线表示候选方向上的当前体素范围* h*，而内侧的虚线表示如果在该方向上插入一条轮廓线的结果范围* h'*。得分候选方向由差值* $h - h'$ *决定。右图：选择左边候选方向的情况。*

为了加快处理速度，我们只考虑这些方向中相对较小的一个子集。

除了构建轮廓外，我们还希望检测这样一种情况，即由其祖先轮廓定义的体素形状已经足够接近原始曲面。这种情况在平滑的输入几何图形中很常见，省略不必要的轮廓通常可以节省大量内存。进行测试的一种方法是检查多面体内每个点到原始曲面的距离是否低于一个固定的临界值。实际上，只需考虑多面体的顶点就能得到有效的近似值。轮廓质量阈值目前指定为有效的立方体八叉树级别，也就是说，如果几何误差小于给定级别上的立方体体素大小，则认为体素的形状足够好。

7.3 属性构建

属性值的计算方法是对体素内的所有输入表面进行积分。我们尝试过使用加权滤波器，如延伸到体素边界之外的吡啶中值滤波器，但结果并不比简单的方框滤波器好。支持度越大的滤波器越容易模糊内容。我们还尝试过取属性范围的中点，认为这样可以最小化误差指标，但结果质量很差。将体素内的所有表面视为同等重要的一个明显问题是，对于相邻的表面，即使是外部看不到的部分也会影响属性。正确的解决方法是找出 "最外层 "的表面，并只它们，但这似乎并不容易，今后的工作。

判断属性是否得到充分表达的方法是查看体素中的输入几何图形，并检查其中是否包含与编码值相差太远值。属性量化和压缩是在测试之前进行的。这样可以确保正确地到它们所造成的假象。请注意，不同的属性使用不同的误差容限。

纹理采样方式对结果质量有很大影响。我们绘制纹理的 mip 地图，并在体素内的每个三角形中提取一个三线性样本。这种方法并不特别精确，但效果令人满意。更精确的解决方案可能会在一定程度上提高质量，尤其是在使用各向异性很强的纹理参数时。

启用属性压缩后，属性将被编码到每个体素中。这是必要的，因为我们目前的实现方式不允许将 DXT 压缩块排除在外。此外，由于颜色和法线是一起编码的，因此也不可能省略其中一个。需要注意的是，压缩伪影会受到属性误差指标的保护，因为实际压缩值被用作参考。增加分辨率会使组合块在空间上更小，更有可能包含相似的颜色，因此使用可变分辨率编码自动修复出现的压缩伪影，除非达到最大级别限制。

7.4 位移绘图

处理位移映射三角形时，首先要将其转换为位移基元。处理方法与处理普通三角形相似，但在必要时可对其进行分割。位移基元表示为位移映射空间中的二次幂正方形和原始三角形。当体素被细分时，位移基元会被分割，直到它们在世界空间中的尺寸小于体素，而那些肯定在体素之外的基元则会被剔除。确定几何外延或对体素内容进行网格划分等操作都是近似的，因为它们是以保守的方式进行的。例如，几何外延是根据从 mip 地图获取的最小和最大位移值计算得出的。

8 成果

主要测试是在一台配备 2.5 GHz Q9300 Intel Core2 Quad CPU 和 4 GB 内存的 PC 上的 NVIDIA Quadro FX 5800 上进行的。操作系统为 64 位版 Windows XP 专业版。使用的是公共 CUDA 2.1 驱动程序和编译器。

其他渲染测试在 NVIDIA GeForce GTX 285 上进行，内存为 1 GB，电脑配备 2.66 GHz E6750 Intel Core2 Dual CPU 和 2 GB 内存。该电脑的操作系统为 32 位 Windows Vista Enterprise。

图 23 显示了本文中使用的测试场景及其三角形数量。由于普遍缺乏大规模的高分辨率体素数据集，我们所有的体素数据集都是由三角形网格建立的。

8.1 内存使用和构建时间

表 2 显示了使用不同体素级别的体素数据集所消耗的 GPU 内存量。我们可以看到，有轮廓的表示（下行）比没有轮廓的表示（上行）要紧凑得多。SIBENIK-D 和 HAIR- BALL 是例外，因为在这两个数据集中，等值线无法很好地表示表面，因此无法省略更精细的层次。当然，场景的等高线版本也更忠实地再现了原始网格。目前，立方体体素的编码会浪费每个体素大约一个字节，因为即使不启用等高线，我们也不会省略等高线指针。不过，这只占总内存使用量的 20%。

现场	10	11	12	13	14	15	16	字节数
城市	8	34	152	655	2724	-	-	4.99
	13	39	131	432	1368	-	-	5.44
锡贝尼克	33	132	530	2120	-	-	-	5.18
	41	141	440	1034	1857	-	-	5.68
西贝尼克-D	47	184	744	2734	-	-	-	5.52
	79	314	1192	2806	-	-	-	8.10
毛球	262	1157	-	-	-	-	-	5.27
	442	1552	-	-	-	-	-	7.48
仙女	10	36	145	606	2669	-	-	5.56
	11	35	99	239	376	639	1109	5.62
会议	21	89	362	1459	-	-	-	5.09
	17	40	96	220	512	1328	-	5.16

表 2： *不同八叉树深度测试场景的 GPU 内存使用量（MB）。对于每个场景，上面一行表示具有立方体体素（无轮廓）的数据集，下面一行表示具有轮廓的数据集。右侧的字节列显示了最大构建数据集中每个体素的平均内存消耗。*

现场	8 9		10	11	12	13	14	15	16
城市	5	6	8	16	45	141	505	-	-
	9	11	14	23	48	115	311	-	-
锡贝尼克	1	3	7	28	104	395	-	-	-
	2	4	10	31	90	194	336	-	-
西贝尼克-D	9	21	56	190	839	2079	-	-	-
	10	24	70	274	1342	2541	-	-	-
毛球	45	70	136	335	-	-	-	-	-
	77	132	294	628	-	-	-	-	-
仙女	1	2	4	12	42	153	592	-	-
	2	3	6	15	38	86	121	178	282
会议	1	2	4	14	55	222	-	-	-
	3	4	7	13	24	51	115	288	-

表 3： *不同层次计数的体素层次结构构建时间。数值以秒为单位，每个数值表示到所示层级的总构建时间。因此，构建单个层次的成本就是相邻数字之间的差值。对于每个场景，上面一行表示立方体体素化（无轮廓），下面一行表示轮廓体素化。*

表中最右边一列显示的是单个体素在最高分辨率下的平均字节数，包括所有开销。与理论数字（第 5.6 节）相比，并考虑到立方体浪费的一个字节，我们可以看到这些测量值在没有轮廓线的情况下比预期值高 0-0.5 字节，而在有轮廓线的情况下则高 0.2-0.7 字节。

造成差异的原因包括分支因子的差异、子描述符阵列的间隙以及轮廓数据量的差异。我们的属性压缩方法要求子体象素的每个跨度都以 128 位边界对齐，假设每个象素平均有 4 个子体，这会导致数据大小膨胀约 1/8。此外，页眉、远指针等也会造成轻微的开销，这些都包含在本图中。

表 3 列出了测试场景的构建时间。请注意，每个数字都是到所示级别的总构建时间，这些数字中不包括加载网格所需的时间。我们可以看到，在大多数场景中，有轮廓的构建时间都低于无轮廓的构建时间。这是因为当轮廓线开始足够逼近曲面时，体素不再细分。

CITY, 879K tris　SIBENIK-D, 77K tris+ disp.　HAIRBALL, 2.88M tris　FAIRY, 93K 三体　CONFERENCE, 283K tris

图 23： *测量中使用的测试场景。*SIBENIK *与*SIBENIK-D *相同，但采用平面三角形。*

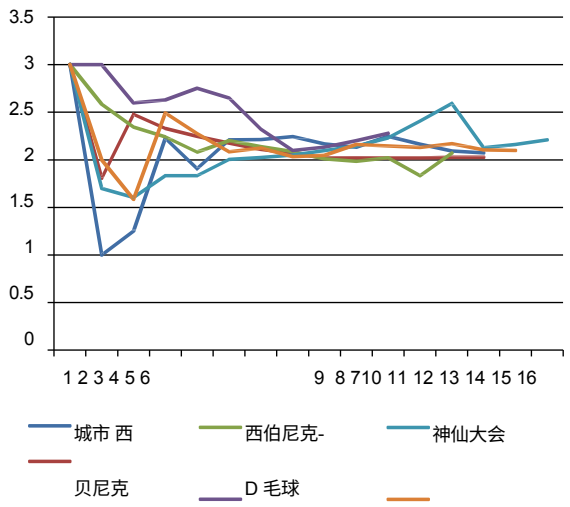

图 24： *测试场景在不同体素水平上的局部维度。我们可以看到，所有场景都趋向于二维设置。*

场景	决议	三角形脚轮	立方体象素	封闭观光	继续。带光束
城市	512x384	46.7	45.1	79.9	88.6
	1024x768	68.5	54.3	89.1	106.0
	2048x1536	77.1	63.9	97.4	123.8
锡贝尼克	512x384	64.3	38.7	80.0	82.5
	1024x768	94.1	46.5	94.1	103.6
	2048x1536	107.1	55.1	103.9	122.0
西贝尼克-D	512x384	-	24.8	32.6	37.5
	1024x768	-	30.2	38.7	48.3
	2048x1536	-	37.1	43.6	60.9
毛球	512x384	11.6	22.4	24.1	24.1
	1024x768	20.5	22.5	27.9	28.4
	2048x1536	31.2	29.2	36.5	38.2
仙女	512x384	63.9	62.1	128.2	132.6
	1024x768	125.1	69.4	145.4	150.9
	2048x1536	155.8	78.6	160.4	169.2
会议	512x384	69.1	35.8	97.9	104.4
	1024x768	111.9	43.8	110.3	124.3
	2048x1536	134.0	52.3	120.2	140.8

表 4： *不同屏幕分辨率下主光线的投射性能。数值单位为每秒数百万条光线。在体素测试中使用了 4 GB 容量的最大数据集。*

8.2　场景维度

图 24 展示了测试场景的局部维度 D_{local}（第 4 节），它取决于体素级别。维度曲线是通过对各层次的平均分支因子取 2 阶对数计算得出的。随着体素变小，场景大多趋向于二维设置，即平均每个体素约有 4 个儿童。在很长一段时间内，HAIRBALL 看起来几乎是体积的，直到单根毛发的表面结构开始占据主导地位。值得注意的是，在这种情况下，单根毛发的一维性质绝不会导致维度小于二维。这是因为在单根毛发表面结构还无法辨别的时候，物体的密度非常高，中心附近的体积状杂乱结构占据了主导地位。而在 FAIRY 中，在单个肢体的表面变得清晰可见之前，维度始终小于 2。

8.3　渲染性能

可以说，最有趣的信息是使用体素数据的渲染性能。在我们的案例中，最主要的因素是光线投射的效率，因为着色成本可以忽略不计，而且后处理过滤比光线投射要快一到两倍。表 4 总结了不同分辨率下测试场景的光线投射性能。表中的数值是以多个视点的平均值测得的，每帧重复多次以摊销启动和刷新延迟。因此，随着分辨率的提高，性能的提升完全可以用更好的光线连贯性来解释。

*三角形*光线投射*器*一栏指是我们之前的论文[Aila and Laine 2009]中描述的最快的 GPU 光线投射器。体素一栏显示的是立方体体素数据的体素光线投射性能，而*等高线*一栏显示的是包含的体素数据的结果。值得注意的是，这两个数据集在平均深度方面有所不同，因为等高线提供了更好的近似度，可以更积极地修剪层次结构。最后，最后一列显示了启用波束优化后的结果（第 6.2 节）。可以看出，在测试场景中，体素光线投射器的性能始终优于三角形光线投射器。

显然，三角形和体素射线投射性能的比较是苹果和橘子之间的比较，因为我们投射数据类型不同。基于三角形的表示法能够完美地辨别出每一条边和每一个角，而体素表示法在这些地方可能会不准确。另一方面，体素表示法在每个样本的基础上都包含独特的（即不重复的）颜色和法线信息，并能表示独特的高分辨率--使用三角形光线投射器无法渲染 SIBENIK-D。

值得注意的是，三角形光线投射器是为快速处理三角形而煞费苦心地进行操作的，稍微增加复杂度就会严重影响其性能。因此，增加对位移贴图的支持很可能会使其性能低于体素光线投射器。诚然，这种说法仍未得到证实，因为目前还没有这样的实施方案。

现场	八度 深度	MB	Quadro FX 5800	GeForce GTX 285	加速 %
城市	13	432	109.5	126.6	15.6
锡贝尼克	12	440	110.1	127.4	15.7
毛球	10	442	32.1	39.3	22.4
仙女	15	639	152.7	182.2	19.3
会议	14	512	124.4	144.8	16.4

表 5：*两块配备相同 GPU 的板卡之间的比较。数值单位为每秒百万条光线，测量方法是使用等高线和光束优化渲染 1024z768 主光线。为了公平比较，两次运行的内存使用量都限制在 GTX 285 显卡的 1 GB 可用内存范围内。这也解释了为什么 Quadro 的报告数字高于表 4 中的数字。*

我们假设，在几何图形极其精细的静态场景中，以体素层次结构而非三角形和位移贴图进行光线投射时，光线投射的性能会更高。，我们假设，在具有极其精细几何图形的静态场景中，如果采用体素层次结构而不是三角形和位移贴图进行光线投射，则光线投射性能会更高。

8.4 GTX 285 实验

由于 Quadro 显卡通常比配备相同 GPU 的 GeForce 显卡速度稍慢，因此我们想测试一下在 GeForce GTX 285 显卡上测试结果会有多大不同。由于我们的 GTX 显卡只有 1 GB 内存，因此我们使用适合 1 GB 内存的较小数据集进行测试。例如，在 CITY 中，我们只能使用 13 个级别，而不是其他测试中使用的 14 个级别。

GTX 285 的核心时钟速度比 Quadro 显卡快约 14%，峰值显存带宽则高出 56%。表 5 总结了启用轮廓和光束优化后的渲染结果。根据结果，GTX 285 的渲染速度提高了 15-22%，平均速度提高了 18%。观察到的速度提升表明，光线捕捉器并没有受到内存带宽的严格限制，因为核心时钟速度的差异几乎足以解释这一结果。如果我们受到显存性能的严重限制，速度提升应该更高。

8.5 详细的执行概况

表 6 列出了使用等值线以 *1024z768* 分辨率渲染测试场景时的一些剖析计数器。除考虑整个画面的数值外，所有数值均为每条光线的平均值。通过将 Mrays/s 数值与表 4 对比可以看出，基准运行之间存在一些波动（小于 1%），我们无法消除这些波动。

最上面一行 *Mrays/s* 表示测量的渲染性能，单位为每秒数百万射线。下一行*模拟*显示的是通过计算每个代码块在整个翘曲范围内的执行次数以及相应的指令次数加权得到的模拟性能数据。这个数字包含了翘曲发散的影响，因此是光线投射最佳执行速度的上限。我们假定每一条指令都有可能双发，并且不存在内存延迟或其他任何干扰。同样的技术也曾用于估算 GPU 的理论光线投射性能[Aila 和 Laine，2009 年]。

下一行显示了我们在测量中达到的模拟性能。三角光线投射器的主要光线性能始终超过模拟性能的 80%，在没有光束优化的情况下，我们获得了与体素接近的数据。然而，启用光束优化后，我们的数据下降到 60-80% 的范围，甚至低于 FAIRY 中的数据。这表明

在某种程度上，波束优化会使我们的内存带宽受到限制。

内存带宽 *bw GB/秒的*计算方法是：将每条光线访问的全局和本地内存字节数相加，乘以每秒光线数，再除以 2^{30}。这个数字没有凝聚或任何其他实际内存子系统特性。我们可以看到，光束优化大大降低了带宽需求。顶部最后一行显示了渲染粗帧进行光束优化所花费的时间占总渲染时间的比例。

下一节将详细介绍代码各部分的执行次数。第一行显示的是每条射线平均执行的指令数，未考虑 SIMD 执行情况。表中未包括 SIMD 的总体效率，即平均每条光线有多少线程被启用。在不使用光束优化的情况下，效率通常在 70% 左右徘徊，而在启用光束优化的情况下，效率则为 60%。

*迭代*行显示光线投影器主循环的迭代次数，*交集行*显示执行了多少次体素-光线交集测试（图 15 伪代码中的 INTERSECT）。行 *push* 表示执行 PUSH 分支的次数，而*存储*则表示实际写入堆栈的次数。行 *advance* 和 *pop* 也对应于伪代码中各自的部分。所有数字都是按行计算的，SIMD 执行次数未计算在内。我们可以看到，到目前为止，最常见的情况是下降到子体素（*push*），最不常见的情况是在层次结构中上升（*pop*）。比较行*推送*和*存储*，我们可以看到，消除不必要堆栈写入的优化能够消除一半以上的堆栈写入。

表格底部列出了内存传输统计数据。第一行 *glob acc* 统计每条射线执行的全局内存取回指令总数，下一行 *glob bytes* 统计取回的字节数。这些数据未考虑凝聚。不过，下一行 *glob trans.* 会计算凝聚后的取数。这一数字的计算方法是：将所有并行执行的取指令计算在内，并计算这些取指令在应用 GPU 中的聚合逻辑后产生的请求数量。同样，总数除以射线数，就得到了每射线的数字。不难看出，取数的凝聚效果相当好，实际事务计数大多在取数指令数的 12% 至 16% 之间只有 HAIRBALL 的比率高达 37%。

接下来的三行显示了本地内存的相同统计数据。所有本地内存访问都是堆栈流量造成的，因为在光线投射内核中没有本地内存溢出。我们假设本地内存是条带化的，因此每个线程地址空间中的相同本地内存地址会连续存储在 GPU 内存中。按照这种方式计算，聚合可将请求次数减少约 13-20%。HAIRBALL 的每请求事务数比率为 44%，在同类产品中处于领先地位。

射线之间的高度一致性解释了由于凝聚而导致的低交易/请求比率。值得注意的是，这些数据仅针对原生光线测量，其他类型的光线可能会表现出较低相干性。不过，对于其他类型的长射线，每条射线的大部分数据也是相同或几乎相同的。

9 采用体素

在本节中，我们将从定量角度探讨体素作为通用几何图形表示法的可行性，并就剩余的瓶颈问题对未来进行粗略预测。

统计	城市		锡贝尼克		西贝尼克-D		毛球		仙女		会议	
	无梁	有梁	无梁	有梁	无梁	有梁	无梁	有梁	无梁	有梁	无梁	有梁
Mrays/s	89.1	106.0	94.1	103.6	38.7	48.3	27.9	28.4	145.4	150.8	110.5	124.3
仿真	96.5	133.8	107.1	142.2	46.4	73.5	38.9	41.3	182.4	268.0	122.2	163.0
模拟百分比	92.4	79.3	87.9	72.8	83.4	65.8	71.8	68.7	79.7	56.3	90.4	76.3
bw GB/s	33.8	25.5	32.1	23.2	25.6	16.3	12.0	10.3	26.7	14.5	32.3	23.1
粗略%	-	15.9	-	15.5	-	10.3	-	3.6	-	15.3	-	18.2
说明	2238	1370	1983	1256	3866	1953	2627	2220	1103	563	1766	1109
迭代	29.4	19.1	25.8	17.7	52.8	27.2	34.2	29.5	14.8	8.1	22.4	15.3
交叉	25.4	16.7	22.6	15.2	38.8	21.5	28.0	23.9	11.5	6.4	20.0	13.3
推动	19.5	14.7	17.0	13.4	30.6	18.3	18.4	16.7	8.0	5.6	14.1	11.2
店铺	8.8	7.3	8.0	6.7	12.3	8.3	7.6	7.0	3.5	2.6	6.1	5.0
前进	8.9	3.4	7.8	3.2	21.2	7.9	15.3	12.2	6.5	2.1	7.3	3.0
嚓	4.1	1.5	3.9	1.4	10.3	3.7	7.4	6.1	3.3	1.1	3.6	1.3
全球累计	51.4	33.1	45.8	31.0	89.2	47.6	58.7	50.3	23.9	13.5	40.4	27.1
glob bytes	303.9	192.6	270.9	179.5	528.9	272.5	340.0	289.4	142.2	77.1	236.2	153.8
glob trans.	6.0	4.5	6.0	4.9	16.3	11.4	19.7	18.7	3.2	2.5	4.8	3.9
本地累计	12.9	8.8	11.9	8.1	22.5	12.0	15.1	13.1	6.8	3.6	9.7	6.3
本地字节	102.8	65.7	95.5	60.6	180.4	90.3	120.5	101.9	54.6	26.4	77.8	45.7
当地转运。	1.7	1.4	1.7	1.5	4.7	3.5	6.0	5.8	1.0	0.8	1.3	1.1

表 6：*以 1024z768 分辨率运行的剖面分析统计数据。本表仅包含轮廓变体。对于每个场景，左列对应的是未进行光束优化的渲染，右列对应的是进行了光束优化的渲染。除顶部部分涉及整个画面外，所有数值均为每个射线的平均值。启用光束优化后，数值的计算方法是将粗帧和全分辨率帧的计数器相加，然后除以全分辨率帧中的射线数。请注意，代码执行统计是按每条射线计算的，由于 32 条射线的 SIMD 并行执行，实际值会更高。有关各行的说明，请参阅正文。*

9.1 渲染性能

根据基准测试结果，使用单个 GT200 级 GPU，我们每秒可在中等精细度场景（SIBENIK-D）中投射约 5000 万条主光线。假设着色成本可以忽略不计，这足以以每秒 25 帧的速度在 1080p 分辨率下进行首次渲染。

最简单、最美观的抗锯齿技术需要对每个像素取 4 个样本，根据我们的测试，这样做的成本大约是对每个像素取一个样本的三倍。此外，阴影射线和反射需要额外的射线。这些辅助光线的相干性比主差，但根据初步测试，它们的成本似乎与主光线差不多。不连贯的辅助光线无疑会更昂贵，但我们还没有进行测试。

假设平均每条主光线采样产生四条次光线，那么我们的采样成本将是仅首次命中情况下的五倍。强制抗锯齿使这一数字翻了三倍，每像素成本乘以 15。按照 GPU 性能每年增长 50%推断，使用这种渲染方式每秒渲染 25 帧需要 7 年左右的时间。

这一估算忽略了着色、后处理过滤以及数据传输等其他操作的成本。，算法的改进很可能会出现。

9.2 内存大小

考虑到我们可以在 4 GB 内存中容纳 600 多平方米 1mmz1mm 分辨率的表面数据（第 3.1 节），而且内存消耗量仅随观察距离的增加而呈对数增长（第 4 节），GPU 内存大小似乎并不像最初看起来那么令人担忧。然而，更复杂的着色模型会大幅增加每个体素的内存使用量。此外，我们也不能假设内存中的每个体素都会以精确的分辨率存储。，基于遮挡的按需流[Crassin 等人，2009 年]可以提供巨大的节省内存使用量。

，令人欣慰的是，与目前可用的内存大小相比，内存需求是非常合理的。GPU 内存大小似乎并不是体素渲染潜在应用的最大瓶颈。

9.3 介质尺寸和速度

虽然当今电路板的 GPU 内存容量相对较高，但用于部署数字内容的媒体却并非如此。一张双层蓝光光盘的容量为 50GB，仅为 GPU 内存容量的 12.5 倍。很明显，如果我们的目标是使内容非常详细，以至于渲染一个视角就需要使用价值几千兆字节的数据，那么数据总量就必须高得多。

毫不奇怪，人们正在研究更大的存储介质。*Holo- graphic Versatile Disc*（HVD）是一项由多家公司联合开发的技术，理论上它的存储容量将达到 6 TB。不过，1 TB 容量的首批产品要到 2016 年才计划发布。此外，计划中的传输速率仅为 125 MB/s，因此，在不浪费寻道时间的前提下，完全刷新一个 4 GB 视频存储器的内容需要 32 秒，如果到寻道时间，则需要更长的时间。

这一估算假设磁盘上的数据大小与 GPU 内存中的数据大小相同，因此忽略了对磁盘上的数据进行压缩的可能性。很难估计在降低数据质量的情况下可以达到什么样的（有损）压缩率。这是一个重要的实际问题，值得仔细研究。

考虑到目前的情况，唯一足以容纳合理数量内容的存储介质是硬盘驱动器目前购买 1 TB 存储容量的价格远低于 0.10 美元/GB。数据传输速率通常低于 100 MB/s，这仍然是个问题。固态硬盘的传输速率要高得多，因此如果价格大幅下降，这个问题可能会得到解决。

在不深入探讨存储技术问题的情况下，分发媒体的大小和速度似乎是数据密集型内容呈现的最大挑战之一。

9.4 远程渲染

一个有趣的可能性是，通过在专用服务器群中运行游戏，并将渲染图形流式传输到客户端，从而避开存储和媒体问题。在这个方向上有一些公司，如 OnLive、OTOY 和 Gaikai，它们承诺通过宽带连接提供高质量的游戏。尽管这种方法面临各种技术挑战，但通过在高性能 RAID 集群中存储巨大的资产，它们可以提供家庭游戏无法提供的功能。

10 未来的工作

向前迈出的明显一步是尝试真正的体积效应，如雾或部分透明材料。我们的数据结构很容易表示这些效果，而且我们认为在光线投射过程中积累消光系数或收集光照度也是相当有效的。

我们使用了简单的 Phong 阴影模型，因为可以获得这种形式的材质数据，在进行下采样时，我们通过求平均值的方法将多个基元阴影属性合并在一起。这在大多数情况下都能产生相当不错的结果，但在计算大体块的总体阴影属性时，最好能将遮挡因素考虑在内。此外，目前还不清楚 Phong 模型是否能很好地表现这类体素的外观。例如，Gobbetti 等人[2005]采用的方法--基于从不同方向对原始数据进行采样的通用阴影方法--似乎也非常适合我们的目的。

虽然所提出的数据结构在渲染方面效率很高，但仍然需要相当大的存储空间。目前 GPU 的内存容量足以满足渲染目的，但如果不进行某种形式的压缩，将大量高分辨率内容存储在光盘上或通过网络进行流式传输似乎是不可能的。找到高效的（可能是有损的）压缩算法将使基于体素的内容在实际应用中更加可行。

在我们的基准测试中，我们将整个场景以全分辨率加载到 GPU 内存中，而由于遮挡和分辨率要求随距离下降，渲染任何单张图像只需要一小部分内存。我们的系统已经支持基于摄像头距离的按需流式传输，但我们有兴趣看看基于可见度的流式传输（与 Crassin 等人的研究成果一致）能在多大程度上进一步减少内存占用。

致谢。感谢 Timo Aila 和 David Luebke 的讨论和有益建议。西贝尼克模型由 Marko Dabrovic 提供。仙女模型由犹他大学的 Ingo Wald 提供。会议室模型由劳伦斯伯克利实验室的 Anat Grynberg 和 Greg Ward 提供。

参考资料

AILA, T., AND LAINE, S. 2009.了解 GPU 上光线遍历的效率。*2009 年高性能图形学论文集*》，145-149 页。

AMANATIDES, J., AND WOO, A. 1987.用于光线追踪的快速体素遍历算法 *In In Eurographics 87*, 3-10.

ATI.2005.Radeon X800 ： 3Dc 白 皮 书 。 *http://www.ati.com/products/radeonx800/3DcWhitePaper.pdf* 。

CRASSIN, C., NEYRET, F., LEFEBVRE, S., AND EISEMANN, E. 2009.Gigavoxels：用于高效去尾体素渲染的光线引导流。In *I3D '09：2009 交互式 3D 图形与游戏研讨会论文集*》，15-22 页。

Dick, C., Kru¨ GER, J., and Westermann, R. 2009. GPU 用于可扩展地形渲染的光线投射。In *Proc.Eurographics 2009-Areas Papers*, 43-50.

费尔南多， R. 2005.Percentage-closer soft shadows.In *SIG- GRAPH '05: ACM SIGGRAPH 2005 Sketches*, ACM, New York, NY, USA, 35.

GOBBBETTI, E., AND MARTON, F. 2005.Far Voxels - a multires- olution framework for interactive rendering of huge complex 3D models on commodity graphics platforms.*ACM Transactions on Graphics 24*, 3, 878-885.

HORN, D. R., SUGERMAN, J., HOUSTON, M., AND HANRAHAN、P.2007.交互式 K-D 树 GPU 光线跟踪。In *I3D '07: Pro- ceedings of the 2007 symposium on Interactive 3D graphics and games*, 167-174.

Knoll, A., Wald, I., Parker, S. G., and Hansen, C. D. 2006.大型八叉树体的交互式等面光线跟踪。*2006 年电气和电子工程师学会交互式光线跟踪研讨会论文集*》，115-124 页。

Knoll, A. M., Wald, I., and Hansen, C. D. 2009.相干多分辨率等值面射线追踪。*Vis.Comput.25*, 3, 209-225.

Laine, S., and Karras, T. 2010.高效稀疏体素树。*ACM SIGGRAPH 2010 交互式 3D 图形与游戏研讨会论文集*》。

MUNKBERG, J., AKENINE-MO¨ LER, T., AND ¨M, J. 2006. 高质量法线贴图压缩*Proc.Graphics Hard- ware 2006*, 95-102.

MUNKBERG, J., OLSSON, O., STRO¨ M, J., AND AKENINE-Mo¨ ller, T. 2007.紧帧法线贴图压缩在 *Proc.图形硬件 2007*》，37-40 页。

RITSCHEL, T., ENGELHARDT, T., GROSCH, T., SEIDEL, H.-P., KAUTZ, J., AND DACHSBACHER, C. 2009. Micro-rendering 用于可扩展的并行最终收集。*ACM Trans.Graph.(Proc. SIGGRAPH Asia 2009) 28*, 5.

Robison, A., and Shirley, P. 2009.图像空间收集。在 *Proc. High Performance Graphics 2009*, 91-98.

SZIRMAY-KALOS, L., AND UMENHOFFER, T. 2008. Displace-GPU 上的 ment mapping - 技术现状。*Computer Graphics Forum 27*, 1.

van waveren, J. M. P., and castan˜ O, I. 2008.实时法线贴图 DXT 压缩。 *http://developer.nvidia.com/object/real-time-normal-map-dxt-compression.html*。

Weyrich, T., Heinzle, S., Aila, T., Fasnacht, D. B., Oetiker, S., Botsch, M., Flaig, C., Mall, S., Rohrer, K., Felber, N., kaeslin, H., and gross, M. 2007. A 曲面拼接的硬件架构。*ACM Transactions on Graphics (Proc. SIGGRAPH) 26*, 3 (Aug.).

ZWICKER, M., PFISTER, H., VAN BAAR, J., AND GROOM, M. 2001.Surface splatting.*SIGGRAPH '01: 第 28 届计算机图形和交互技术年会论文集*》，371-378 页。