三维网格模型的实心体素化与实时渲染

《计算几何》课程大作业报告

郑少锟　　孙志航
2020310830　　2020312646

日期：2021 年 6 月 18 日

1　引言

问题描述

类似二维图像中的像素（Pixel），对三维空间进行量化划分的最小单位被称为体积像素（Volume Pixel），或简称体素（Voxel）。

与同样可以表示三维物体的网格（Mesh）模型不同，体素可以用来描述物体的实体体积，而非仅仅对物体的表面进行建模。同时，体素规则的三维排布使得在其上进行物理模拟、渲染等计算任务时相比网格更加高效。此外，对体素模型的可视化与渲染往往呈现出强烈风格化的视觉效果，带来独特的艺术体验。图 1 展示了流行的体素风格游戏 Minecraft[1]。

将三维模型转化为体素表示的过程称为体素化。在本次大作业中，我们将网格模型转化为体素表示。具体地，我们完成了如下工作：

- 利用空间加速结构计算位于模型内部的体素线段；
- 利用八叉树结构实现对体素线段的稀疏体素化，减少空间占用、提高；
- 对得到的稀疏体素模型进行实时渲染与可视化，并提供了可交互界面；
- 使用线程池技术对算法进行了高效实现，并在渲染过程中对光线与八叉树求交进行 GPU 加速。

组员分工

在本项目中，郑少锟同学主要负责工程项目搭建与打包、网格模型加载与加速结构构建、体素线段计算、线程池与 GPU 并行化等性能优化、光线追踪渲染等工作；孙志航同学主要负责八叉树的构建与 CPU 求交、GUI 界面编写、性能测试与鲁棒性实验，以及 Demo 视频制作、最终报告与文档撰写等工作。

[1]https://www.minecraft.net

图 1: 体素化风格游戏 Minecraft

2 算法概述

计算内部线段

为将模型进行实心体素化，即对于每个在模型内部的格点都生成一个体素，我们投射若干条与坐标轴平行的射线来穿过每一个格点：选取分辨率 N^3，我们在相机空间中以 $(0,0,0),(0,1,0),\ldots,(N-1,N-2,0),(N-1,N-1,0)$ 这 N^2 个点为起点，向 z 轴负方向投射 N^2 条射线。我们由远及近地求出每条射线与模型的所有交点，通过一个初始化为 0 的计数器记录光线在模型内外的状态：如果光线击中模型正面，则计数器自增；如果光线击中模型背面，则计数器自减。从而，使计数器从 0 增至 1 和从 1 减至 0 的一对交点即标记了一条“穿入模型”和“穿出模型”的格点线段，我们认为其表示了一条位于模型内部的体素“线段”。此外，考虑到实际的网格模型可能并非闭合流型（例如存在孔洞或单面建模），我们对穿入穿出不匹配的情况进行了特殊处理以增强算法的鲁棒性。

我们分别尝试了使用 KD-Tree 和 Intel Embree[2]提供的 BVH 来加快射线与模型的求交过程。两者原理相似：通过将空间划分为多个子空间来快速过滤不必要的遍历过程，减少射线与其中面片的相交测试次数。这一优化显著减少了运算量。在最后的展示实现中，基于 Embree 提供的高质量层次包围盒加速结构，我们使用自行实现的线程池并行化方案实现了对网格模型的实时光线追踪渲染。

稀疏体素化

一般而言，实心体素化需要按分辨率将大量连续的体素堆积在模型内部，这是对空间极大浪费，也妨碍了快速求交的进行。因此，我们使用稀疏八叉树来提升体素模型的

[2]https://github.com/embree/embree

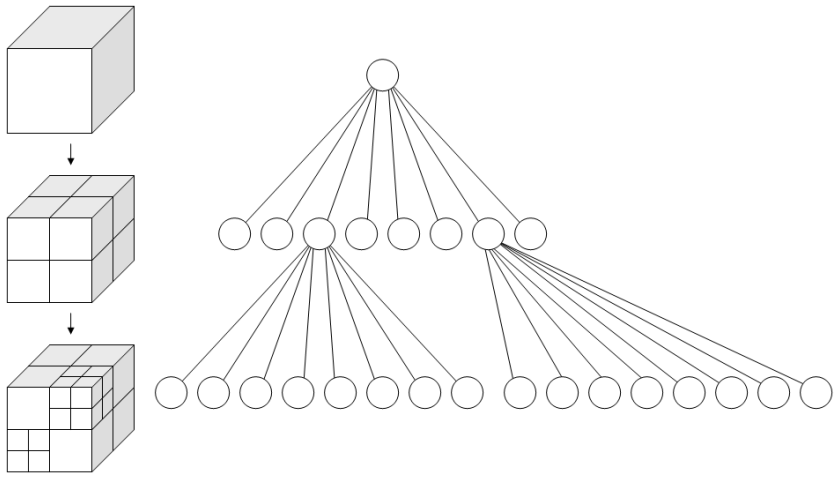


图 2: 八叉树结构示意图

存储及利用效率。图 2 展示了八叉树的示意图。

在我们的项目中，每个八叉树节点为格点组成的正方体，节点分为三种情况：

- 正方体中每个格点均有体素，称该节点是满的，满节点是叶子节点。
- 正方体中每个格点均无体素，称该节点是空的，空节点是叶子节点。
- 正方体中存在有体素的格点，也存在无体素的格点，称该节点是半满的，半满节点为非叶节点，需要继续细分（将该正方体按 $2 \times 2 \times 2$ 的方式分为 8 个子正方体，每个子节点代表其中一个）。

我们将通过求交得到的线段直接用于稀疏八叉树的构建。具体地，对于八叉树的每一层级，我们分别计算与其八个子空间相交的线段集合。这一过程被递归直到我们能够确认到达叶子节点（即全空或全满节点）；子节点满、空、半满的信息会返回父节点，并在其中通过位域进行标记，以支持后续的快速求交。

体素的渲染

我们分别在 CPU 和 GPU 上实现了光线与八叉树的求交计算，从而能以光线追踪的方式渲染体素模型。在八叉树表示的稀疏体素模型下，若光线穿过八叉树节点对应的方块，光线与正方体内体素的交点可以如下方式求得：

- 若节点是空的，则显然光线与此块无交点，我们可以直接返回以进行后续遍历；
- 计算光线与该块的交点，此时有如下三种情况：
  1. 若光线与方块无交点，则光线与该块子节点均不可能有交，我们可以返回以进行后续遍历；
  2. 若节点是满的，则光线与正方体的交点就是与体素的交点；
  3. 否则节点是半满的，则按照一定顺序对子节点进行递归遍历求交。

可以看出，用八叉树表示稀疏体素模型，不但节省了空间，还有希望在体积较大的高层节点处便快速确定或排除交点，简化了光线与体素求交的过程，显著减少了求交操

作的次数。

3 数据结构与模块设计

模型加载模块

输入：OBJ 或 PLY 格式的网格模型。

输出：顶点列表和对顶点进行索引的面片列表，其中每个面片（三角形）记录了顶点在顶点列表中的位置。

模型加载基于开源库 ASSIMP[3]实现；为方便后续处理，我们将加载得到的场景图进行扁平化以获得单一的顶点和面片列表，并将所有顶点归一化使其坐标均在 $[-1,1]^3$ 范围内。

线段求解模块

输入：面片列表，可以通过面片索引三角形三个顶点的位置。

输出：线段的列表，每条线段记录四个整数 $\{x, y, z_{\min}, z_{\max}\}$，以定点数形式（按要求的体素分辨率归一化），记录从 (x, y) 为起点的射线在模型内部的部分 $(x, y, z_{\min})$ 至 $(x, y, z_{\max})$，即一段平行于 z 轴的连续体素线段。

我们实现了 KD-Tree 永于加速射线与三角网格模型的求交计算，即先将面片组织为 KD-Tree 形式，再对于前述的每一条射线与 KD-Tree 求交。此外，我们也对 Embree 进行了适配，以提供更高性能的光线求交运算。

设加速结构大小为 N，每一像素与模型相交的平均线段数量为 S，体素分辨率为 R^3，该阶段时间复杂度为 $O(R^2 S \log N)$，所需空间为 $O(N + R^2 S)$。

稀疏体素化模块

输入：线段的列表，每条线段表示一段连续的体素。

输出：体素的八叉树表示。

该模块递归构建八叉树：若一个节点对应的方块充满体素或不存在体素，则是叶节点；否则将其分割为 8 个子块，即建立 8 个子节点。为减少内存消耗和提高访存局部性，我们将节点存储在数组中，同组 8 个子节点连续存储，并将每个子节点控制在 8 字节内，其中包括 32 位的首个子节点索引偏移、8 字节子节点是否为空的位域标记和结点本身是否为满的标记。

具体构建算法可见上一节内容。设线段数量为 M，体素分辨率为 R^3，则该阶段时间复杂度为 $O(M \log R)$，所需空间为 $O(M + R)$。

[3]https://github.com/assimp/assimp

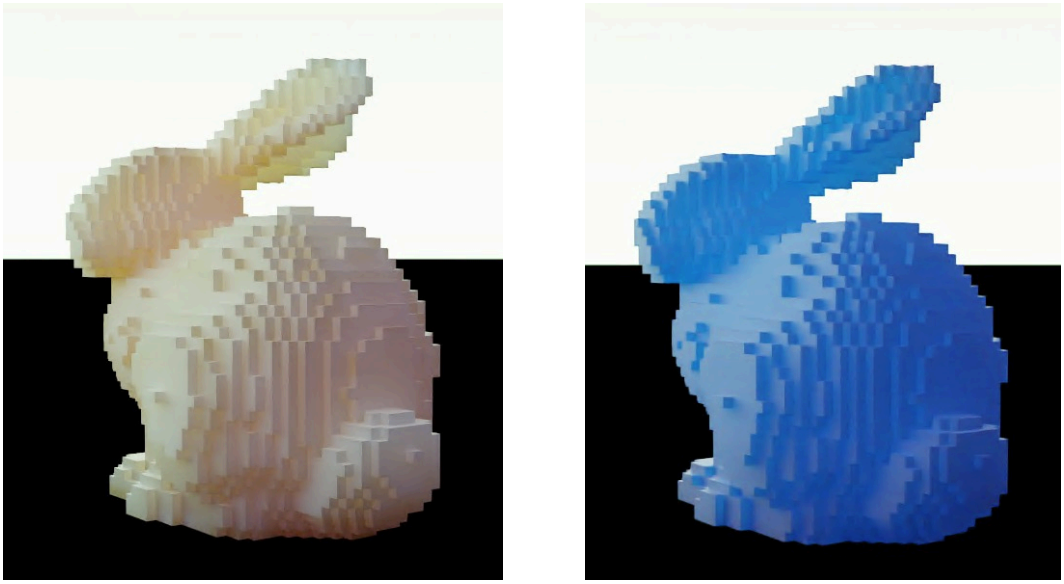


图 3: 使用路径追踪渲染器对导出模型进行体渲染的效果

此外，在这一模块中我们也实现了将稀疏体素模型导出为 OBJ 模型格式的功能，从而能够将其方便地用于其他任务中，例如图 3 展示了利用路径追踪渲染器对导出模型进行体渲染的效果。

渲染模块

输入：网格模型的加速结构表示，体素模型的八叉树表示，相机参数，光源参数等。

输出：图片。

该模块采用光线追踪算法，为确定图片上每个像素的颜色，从相机发出光线与模型求交，如存在交点，则根据光源和模型颜色参数以漫反射模型进行着色。为消除图像采样率不足造成的锯齿，每帧我们使用 Halton 采样器对像素点位置进行随机扰动，并在场景参数未改变的情况下将多帧结果累积以消除噪声。

为提高效率，除使用 CPU 并行求交和渲染外，我们也使用 CUDA 在 GPU 上实现了八叉树的求交计算。值得说明的是，由于 GPU 代码并不适合递归，我们利用栈将递归求交过程改写为循环过程。

交互模块

输入：用户发出的指令。

输出：GUI 图形界面。

该模块接受用户的输入并调用其他模块。当用户进行移动、旋转相机，改变相机朝向，改变光源位置，改变光强和漫反射系数（可对红绿蓝三种色光分别调整）等操作时会重新渲染，否则将积累渲染结果。此外，界面上也提供了重体素化、保存渲染图像、

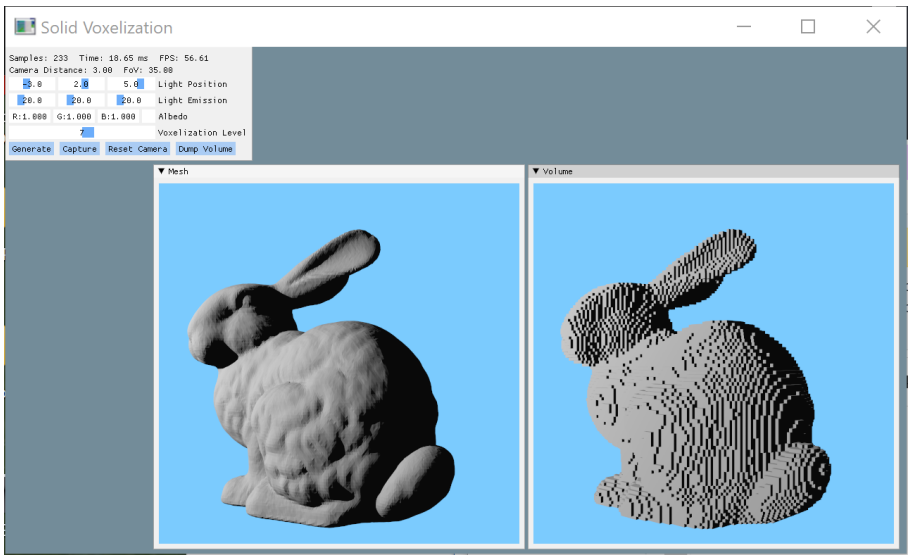


图 4: 图形用户界面

重置相机和导出稀疏体素模型文件等功能按钮。

GUI 界面基于开源库 Dear ImGui[4]实现。图 4 展示了我们程序运行时的图形用户界面。

4 效率测试

本节分别测试加速结构构建，体素化（包括线段计算与稀疏体素化）效率和渲染效率。所有测试均在一台配备 AMD Ryzen 7 4800H 与 NVIDIA GeForce RTX 2060，运行 CUDA 11.2 的笔记本电脑上完成。

预处理效率

预处理将组成网格模型的面片组织成求交加速结构，其用时与面片的数量正相关，对于计算机图形学中经典网格模型，预处理用时如表 1 所示。

表中结果显示，在面数足够大的情况下，预处理用时与面数大致成正比。由于预处理在整个算法中只执行一次，用时随面数的增长是可接受的。

体素化效率

体素化效率受加速结构复杂度和体素化分辨率影响，与生成线段数量成正比。对每个模型以不同分辨率执行体素化，用时如表 2 所示。

结果显示体素化用时主要和分辨率相关，而受面数的影响不大，这和我们给出的复杂度估计一致。在这些例子中，512^3 分辨率均达到了很好的视觉效果，且如非频繁重新

[4]https://github.com/ocornut/imgui

表 1: 网格模型预处理用时

模型名称	面数	用时 (ms)
Cow	5804	11
Bunny	69662	14
Forklift	451013	92
Bust	716802	140
Dragon	871414	154
Buddha	1087716	200

表 2: 体素化用时

模型名称	面数	128^3 用时 (ms)	256^3 用时 (ms)	512^3 用时 (ms)
Cow	5804	3	9	53
Bunny	69662	4	14	86
Forklift	451013	4	9	48
Bust	716802	3	9	50
Dragon	871414	3	12	61
Buddha	1087716	3	10	49

体素化，代价也是可接受的。

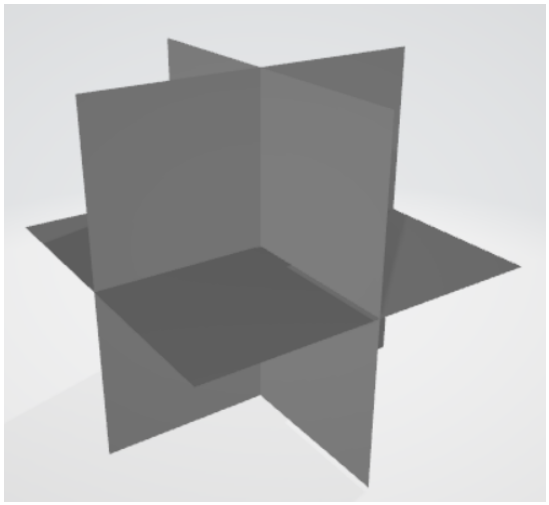
渲染效率

渲染效率受到体素分辨率影响。对每个模型以不同分辨率执行体素化后，渲染 512×512 大小最终图片的单帧用时如表 3 所示，其中体素模型的渲染使用了 GPU 路径。

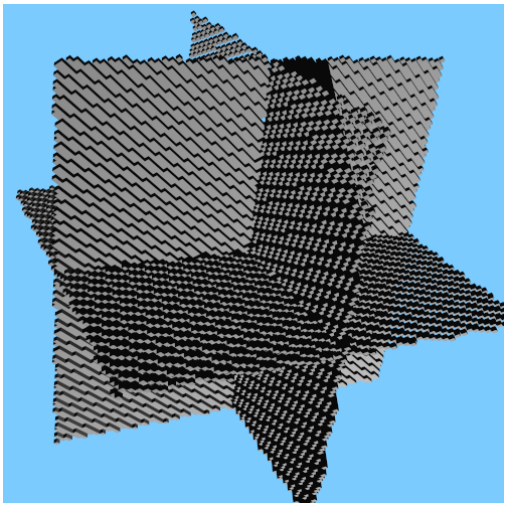
表 3: 渲染单帧用时

模型名称	128^3 用时 (ms)	256^3 用时 (ms)	512^3 用时 (ms)
Cow	17	19	21
Bunny	20	22	26
Forklift	19	22	26
Bust	20	21	23
Dragon	23	26	30
Buddha	19	23	26

结果显示渲染用时受体素分辨率影响并不显著（相对于体素化用时）。这是因为八叉树求交效率较高，渲染时间大部分花在了图形库生成图像上。然而，Dragon 相对其他



(a) 鲁棒性测试用例



(b) 鲁棒性测试结果

图 5: 鲁棒性测试

模型用时稍长，检查发现 Dragon 有较多的孔洞，生成的体素模型有较多小的不连续区块，影响了八叉树结构的有效性，这也是八叉树这一空间划分结构的问题之一。

5 鲁棒性测试

射线与模型求交的体素化算法依赖“穿入模型”和“穿出模型”的点对，因此令我们担忧的边界情况是它对无厚度的平面的处理。我们构造了如图 5a 所示的用例，它由三个无厚度的平面交叉而成。

结果如图 5b 所示，虽然不可避免产生锯齿，但考虑到它基本保持了平面的形状，这个结果是令人满意的，也说明我们对边界情形的处理是有效的。

6 结论

我们实现了一种基于稀疏八叉树的网格模型实心体素化方案。该方法通过使用射线求交计算位于网格模型内部的体素线段，从这些线段构建稀疏八叉树结构并进行实时光线追踪渲染。我们对该方法进行了基于线程池和 GPU 的并行化加速，在测试模型上得到了很好的效率与效果。对于异常模型的鲁棒性测试结果也证明了我们方法的有效性和稳定性。当然，由于时间和精力的限制，我们的算法和实现还存在一些有待提升之处，例如寻找对于带洞模型更鲁棒的体素化方案、提高八叉树构建和求交效率等等。

我们诚恳地感谢邓老师的精心细致的备课与耐心周到的讲解，感谢助教学长的辛勤工作与付出，也感谢每一位一起学习的同学，让我们在计算几何的世界里一起度过了难忘而富有启发的一学期。

参考文献

[1] LAINE S, KARRAS T. Efficient sparse voxel octrees[J]. IEEE Transactions on Visualization and Computer Graphics, 2010, 17(8): 1048-1059.

[2] GLASSNER A S. Space subdivision for fast ray tracing[J]. IEEE Computer Graphics and applications, 1984, 4(10): 15-24.

[3] DOMINGUES L R, PEDRINI H. Bounding volume hierarchy optimization through agglomerative treelet restructuring[C]//Proceedings of the 7th Conference on High-Performance Graphics. [S.l.: s.n.], 2015: 13-20.

[4] VINKLER M, BITTNER J, HAVRAN V. Extended morton codes for high performance bounding volume hierarchy construction[M]//Proceedings of High Performance Graphics. [S.l.: s.n.], 2017: 1-8.

[5] STRÖTER D, MUELLER-ROEMER J S, STORK A, et al. Olbvh: octree linear bounding volume hierarchy for volumetric meshes[J]. The visual computer, 2020, 36(10): 2327-2340.

[6] PARKER S G, BIGLER J, DIETRICH A, et al. Optix: a general purpose ray tracing engine[J]. Acm transactions on graphics (tog), 2010, 29(4): 1-13.

[7] WALD I, WOOP S, BENTHIN C, et al. Embree: a kernel framework for efficient cpu ray tracing[J]. ACM Transactions on Graphics (TOG), 2014, 33(4): 1-8.

[8] BENSON D, DAVIS J. Octree textures[J]. ACM Transactions on Graphics (TOG), 2002, 21(3): 785-790.

[9] CRASSIN C, GREEN S. Octree-based sparse voxelization using the gpu hardware rasterizer[J]. OpenGL Insights, 2012: 303-318.

[10] SCHWARZ M, SEIDEL H P. Fast parallel surface and solid voxelization on gpus[J]. ACM transactions on graphics (TOG), 2010, 29(6): 1-10.

[11] FOLEY T, SUGERMAN J. Kd-tree acceleration structures for a gpu raytracer[C]//Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware. [S.l.: s.n.], 2005: 15-22.

[12] KAUFMAN A, SHIMONY E. 3d scan-conversion algorithms for voxel-based graphics[C/OL]//I3D '86: Proceedings of the 1986 Workshop on Interactive 3D Graphics. New York, NY, USA: Association for Computing Machinery, 1987: 45–75. https://doi.org/10.1145/319120.319126.

[13] DANIELSSON P. Incremental curve generation[J/OL]. IEEE Transactions on Computers, 1970, C-19(9): 783-793. DOI: 10.1109/T-C.1970.223041.

[14] MOKRZYCKI W. Algorithms of discretization of algebraic spatial curves on homogeneous cubical grids[J]. Comput. Graph., 1988, 12: 477-487.

[15] KAUFMAN A. An Algorithm for 3D Scan-Conversion of Polygons[C/OL]//EG 1987-Technical Papers. Eurographics Association, 1987. DOI: 10.2312/egtp.19871015.

[16] KAUFMAN A. Efficient algorithms for 3d scan-conversion of parametric curves, surfaces, and volumes [J/OL]. SIGGRAPH Comput. Graph., 1987, 21(4): 171–179. https://doi.org/10.1145/37402.37423.

[17] COHEN D, KAUFMAN A. Scan-conversion algorithms for linear and quadratic objects[J]. IEEE Computer Graphics & Applications, 1991.