

计算机动画的算法与技术——基于 GPU 的碰撞检测算法

田正祺 2020080095 软件 01

1 编译

此项目在 Windows 11 操作系统使用 Visual Studio Community 以及 Microsoft (R) C/C++ Optimizing Compiler Version 19.37.32825 for x64 编译器进行编译。可以使用 CMake 以及 CMakeLists.txt 生成 Visual Studio 的工程文件。理论上所有代码依赖都在 libs/ 中并会自动编译与链接，无需手动安装额外依赖。编译与运行可以使用 build.Ps1 脚本。此程序是在 AMD 6800H 处理器（包含 AMD Radeon 680M 集显）上开发测试的。

2 运行

bin/resources/collidables.txt 中定义了所有碰撞体。一行表示一个碰撞体，格式为：


```
s p.x p.y p.z v.x v.y v.z m c l [w] [h]
```

其中：

- s: 形状，可以为球形 s 或长方体 c
- p: 初始位置三元向量
- v: 初始速度三元向量
- m: 重量
- c: 弹性系数
- l, w, h: 若是球形，则只填写 l，表示半径，否则填写 l, w, h，表示长方体的长、宽、高

建议使用 bin/resources/test.py 生成此文件。

运行 bin/gpu-collision-detection.exe 进入仿真动画，漫游方法如下：

- | | |
|--|---|
| •  : 向前移动 | •  : 向上移动 |
| •  : 向左移动 | •  : 向下移动 |
| •  : 向后移动 | • 鼠标左/右移动: 向左/右看 |
| •  : 向右移动 | • 鼠标上/下移动: 向上/下看 |
| | •  : 进入菜单 |

在菜单中可以调节漫游速度、动画速度，重置场景，以及暂停仿真。

3 代码结构

以下的路径都在 `src/` 目录下

- `kernels/`: 在 GPU 上运行的函数
 - `kernels/defines.cl`: 基本结构体（碰撞体、树节点等）的定义
 - `kernels/morton.cl`: 计算结构体的莫顿码
 - `kernels/construct.cl`: 构建层级包围体树
 - `kernels/aabb.cl`: 计算树中每个节点的 AABB
 - `kernels/traverse.cl`: 遍历树进行宽相位和窄相位碰撞检测，并计算碰撞后的速度
 - `kernels/physics.cl`: 根据万有引力更新碰撞体位置以及速度
 - `kernels/matrices.cl`: 计算每个碰撞体的 OpenGL 模型矩阵
- `shaders/`: OpenGL 着色器
- `libs/`: 第三方库代码
- `resources/`: 资源文件
- `headers/`: C++ 头文件，每个文件的名称对应着其中声明的类
 - `camera.hpp`: 相机类，让用户在场景中漫游
 - `collidable.hpp`: 碰撞体的属性与创建
 - `gpu_collision_detector`: 基于 OpenCL 的 GPU 加速碰撞检测
 - `model.hpp`: 用来在 OpenGL 动画中表示碰撞体的模型
 - `shader.hpp`: 编译，使用 OpenGL 着色器
- `sources/`: C++ 源文件，每个文件/目录的名称对应着其中定义的类

4 运行流程

1. 初始化使用 OpenGL 时需要的各个部件 (GLFW, GLAD 等) (`sources/main.cpp`, `sources/scene/misc.cpp`), 并进入渲染函数 (`sources/scene/render.cpp`)
2. 加载着色器、模型节点数据, 进入渲染循环
3. 若需要重置, 从文件加载碰撞体的信息, 创建包围所有碰撞体的箱子
4. 计算 FPS、处理用户的键盘数据, 即视角的移动
5. 更新碰撞体的物理属性 `sources/gpu_collision_detector/update_physics.cpp`
 - (a) 计算莫顿码 `kernels/morton.cl`
 - (b) 根据莫顿码排序碰撞体
 - (c) 构建层级包围体树 `kernels/construct.cl`
 - (d) 计算树中每个节点的 AABB `kernels/aabb.cl`
 - (e) 遍历树进行宽相位和窄相位碰撞检测, 并计算碰撞后的速度 `kernels/traverse.cl`
 - (f) 根据万有引力更新碰撞体位置以及速度 `kernels/physics.cl`
6. 计算表示碰撞体的模型的模型矩阵 `sources/gpu_collision_detector/update_physics.cpp`, `kernels/matrices.cl`
7. 渲染所有模型
8. 渲染 GUI
9. 重复以上渲染循环内的步骤

5 算法

5.1 背景

碰撞检测 (collision) 是判断两个物体是否碰撞的算法, 在计算机图形学、计算物理学、电子游戏等等多个领域中使用。碰撞检测通常分为宽相位 (broad phase) 碰撞检测和窄相位 (narrow phase) 碰撞检测^[1]。宽相位碰撞检测用于快速粗略地排除完全没法碰撞的物体, 然后对可能碰撞的物体进行更精确的但通常更慢的窄相位碰撞检测。由于仿真的物体 (球, 正/长方体) 比较简单, 此项目会使用简单的窄相位碰撞检测算法, 而聚焦在宽相位碰撞检测。

5.2 算法概述

朴素的算法会将 n 个物体中的每一个物体与其他 $n - 1$ 个物体进行碰撞检测, 其时间复杂度为 $O(n^2)$ 。对于几万物体, 这个算法的速度不够快。

为了降低时间复杂度, 此项目使用的数据结构是层级包围体树。一个包围体必须完全包围它的物体或子包围体的所有点。为了加快计算, 通常会使用最小轴对齐包围盒 (minimum axis-aligned bounding box), 即最小的, 各个边与三个坐标轴平行的长方体包围体。包围体可以形成树结构: 若叶节点都表示物体, 而内部节点表示包围体, 并且任意一个内部节点必须是它的子节点的包围盒, 则形成一个层级包围体树。在遍历层级包围体树时, 若一个物体与某一个节点表示的包围体没有碰撞, 则物体不会与此包围体的子节点表示的子包围体/物体碰撞, 因此可以跳过子节点。对于理想构建的树而言, 时间复杂度可以降低到 $O(n \log n)$ 。

以下会讨论构造与遍历层级包围体树的算法的实现细节, 以及可以利用 GPU 加速的地方。

5.3 树的构建

此项目的碰撞检测对象是动态的物体, 即每次物体的位置更新后需要更新层级包围体树并做一次碰撞检测。目前存高效的算法, 可以一次性构建一棵树并每次更新同一棵树^[2]。但是考虑在此场景中, 物体的位置可能会有巨大的变化, 因此选择每次进行碰撞检测时重新构建全新的树。

此项目采用了线性层级包围体算法^{[3][4]}, 将所有物体排序, 并划分区间, 将同一个区间的物体放在同一个包围体中。由于更小的包围体能够更精确地拟合物体, 则目标是将距离较小的物体放在同一个包围体中。为了将三维位置信息映射到一维空间并且保留位置的局部性, 可以使用 Z 阶曲线 (Z-order curve, 也称 Morton order/code 莫顿码)^[5]。一个三维位置的莫顿码此是位置的三个坐标轴的取值的二进制表示中的每个比特的交错。比如:

```
x = 0.00101010 -> 0.0 0 1 0 1 0 1 0
y = 0.10100101 -> 0. 1 0 1 0 0 1 0 1
z = 0.10101010 -> 0. 1 0 1 0 1 0 1 0
Z(x, y, z) = 0.0110001110001010101010
```

我的初始实现对 100,000 个随机位置计算莫顿码需要 4.8ms, 其中包括了数据拷贝的时间。

计算到 Z 值后需要进行排序, 而排序是一个可并行化的操作。^[6] 中测试了多种并行排序算法, 其中对于此场景 (parallel, 32/64-bit, (key, value) pairs), 并行归并排序似乎是最快的。此项目将中测试多种算法。

排序后, 根节点覆盖这 $[0, n - 1]$ 区间内的物体, 而它的左子树和右子树覆盖的区间分别为 $[0, i]$ 和 $[i + 1, n - 1]$, 其中 i 是某一个合适的分割点。问题是自顶向下的递归方法会限制树的构建的可并行性, 因此计划使用^[7] 中提到的算法。

节点的包围体是自底向上计算的。叶节点的包围体是物体的包围体，而内部节点的包围体是它的两个子节点的包围体的并。如果使用包围盒，可以用长方体的对角线上的两个点表示。为了简化计算，可以取 x, y, z 分别最小和分别最大的两个角。在 GPU 上可以为每个叶节点创建一个线程，计算完了一个节点的包围体就计算父节点的包围体。为了避免重复地计算以及保证两个子节点的包围体也计算完毕，可以使用一个原子标志，使得第一个处理此节点的线程直接返回，而让第二个处理此节点的线程真正进行计算。

5.4 树的遍历

有了层级包围体树后，仅仅需要对每一个物体遍历树进行包围体的碰撞检测，如概述中所述。但是，由于 GPU 与 CPU 有类似的缓存的时间和空间局部性，以下是一些加速技巧^[8]：

- 用手动，迭代性的遍历，而不是直接递归。这样每个线程运行的是同一个循环，减少运行指令以及数据的发散。
- 为了增加时间局部性，由于相邻的叶节点在三维空间中也会比较近，则以叶节点的顺序进行搜索。
- 每个碰撞会被检测两次。为了避免进行冗余的计算，设定若 A 与 B 进行碰撞，则在树中 A 必须在 B 的前面。需要为每个内部节点加一个指针，表示此内部节点最靠右的叶节点。

参考文献

- [1] Thinking Parallel, Part I: Collision Detection on the GPU | NVIDIA Technical Blog — developer.nvidia.com[EB/OL]. <https://developer.nvidia.com/blog/thinking-parallel-part-i-collision-detection-gpu/>.
- [2] WALD I, IZE T, PARKER S G. Fast, parallel, and asynchronous construction of bvhs for ray tracing animated scenes[J/OL]. Computers & Graphics, 2008, 32(1): 3-13. <http://dx.doi.org/10.1016/j.cag.2007.11.004>.
- [3] Thinking Parallel, Part III: Tree Construction on the GPU | NVIDIA Technical Blog — developer.nvidia.com[EB/OL]. <https://developer.nvidia.com/blog/thinking-parallel-part-iii-tree-construction-gpu/>.
- [4] LAUTERBACH C, GARLAND M, SENGUPTA S, et al. Fast bvh construction on gpus[J/OL]. Computer Graphics Forum, 2009, 28(2): 375-384. <http://dx.doi.org/10.1111/j.1467-8659.2009.01377.x>.
- [5] Z-order curve - Wikipedia — en.wikipedia.org[EB/OL]. https://en.wikipedia.org/wiki/Z-order_curve.
- [6] BOZIDAR D, DOBRAVEC T. Comparison of parallel sorting algorithms[A]. 2015. arXiv:1511.03404.
- [7] KARRAS T. Maximizing parallelism in the construction of bvhs, octrees, and k-d trees[C]//EGGH-HPG'12: Proceedings of the Fourth ACM SIGGRAPH / Eurographics Conference on High-Performance Graphics. Goslar, DEU: Eurographics Association, 2012: 33-37.
- [8] Thinking Parallel, Part II: Tree Traversal on the GPU | NVIDIA Technical Blog — developer.nvidia.com[EB/OL]. <https://developer.nvidia.com/blog/thinking-parallel-part-ii-tree-traversal-gpu/>.