基于层次包围盒的碰撞检测算法的存储优化

金汉均[1],王晓荣[1],王 萌[2]
JIN Han-jun[1],WANG Xiao-rong[1],WANG Meng[2]

1.华中师范大学 计算机科学系,武汉 430079
2.广西工学院 计算机工程系,广西 柳州 545006
1.Department of Computer Science,Huazhong Normal University,Wuhan 430079,China
2.Department of Computer Engineering,Guangxi University of Technology,Liuzhou,Guangxi 545006,China
E-mail:ccnuxnxs@163.com

JIN Han-jun,WANG Xiao-rong,WANG Meng.Memory-optimized of collision detection algorithm based on bounding-volume hierarchies.Computer Engineering and Applications,2007,43(16):61-63.

Abstract: A method of memory-optimized is presented for collision detection algorithm based on bounding-volume.Collision detection algorithm based on *AABB* tree is improved from a space perspective.From the constructing process of *AABB* tree,the amount of byte of *AABB* bounding-volume for internal node is reduced.We wipe the bounding-volume out from leaf nodes structure based on a fast triangle-triangle intersection test algorithm,and then wipe the leaf nodes out.We optimize the storage of *AABB* bounding-volume and leaf node at the same time.The direct result is that it can save a large amount of space and speed up the algorithm.

Key words: *AABB* bounding-volume;memory-optimized;intersection test;collision detection

摘 要: 介绍了基于层次包围盒的碰撞检测算法的存储优化方法。该方法从存储空间的角度来改进基于*AABB*树的碰撞检测算法。根据*AABB*树的构造过程,减少内部节点的*AABB*包围盒的存储字节数;基于快速三角形相交测试算法,从叶节点结构里去掉包围盒信息,将叶节点从存储结构中删除。实验表明,利用*AABB*包围盒和叶节点的存储优化,既减少了算法的存储空间又加快了算法的执行时间。

关键词:*AABB*包围盒;存储优化;重叠测试;碰撞检测
文章编号:1002-8331(2007)16-0061-03 **文献标识码**:A **中图分类号**:TP391

1 引言

虚拟环境中由于物体的交互和运动,物体间经常会发生碰撞。系统必须能及时检测到这些碰撞,以避免产生穿透等不真实的现象。随着虚拟现实技术和分布式仿真技术的兴起,碰撞检测问题成为一个研究的热点。

层次包围盒法是进行物体间碰撞检测的常用方法。初检测阶段通过物体的包围盒相交测试来排除不可能相交的物体对,再对包围盒相交的物体对进行精确的相交测试。由于求包围盒的交比求物体的交简单的多,所以可以快速排除很多不相交的物体,从而加速了算法。但是,虚拟环境中大量的物体以及物体的复杂形状使得碰撞检测模块占去大量的存储空间和处理时间,往往导致系统处理的瓶颈。提高碰撞检测的速度就成为很多学者关注的问题。大部分研究都关注于选择合适的包围盒、采用合适的方法构造包围盒树来提高碰撞检测的速度,没有太多关注内存需求问题。包围盒树中存储每个三角形通常需要一个不可忽略的字节数,这从根本上引发了我们对包围盒树存储需求的关注[1]。

初期用于减少包围盒树存储需求的方法主要有两种。方法一是使得包围盒树每个叶节点含有多于一个的三角形;方法二根据包围盒树节点的左右孩子节点的存储地址总是相邻的,使得一个孩子节点的索引信息变为隐藏信息。两种方法都减少了包围盒树的内存需求,但也一定程度上增加了检测碰撞的时间。

文献[2]对*AABB*包围盒进行了存储压缩,减少了存储整棵*AABB*树所需的字节数。本文在文献[2]所提的存储优化方法的基础上进行了改进。根据*AABB*树的构造方法来优化内部节点的存储结构,这一操作减少了存储需求但是却引发了时间问题。为了加速算法,利用基于Möller提出的两种有关三角形的快速检测方法[3,4]来优化叶节点的存储。实验证明,内部节点和叶节点的存储优化减少了包围盒树的内存需求,同时也提高了碰撞检测的速度。

2 存储需求分析

2.1 树的构造

在分析采用*AABB*树的碰撞检测算法的存储需求之前,首

基金项目:广西省自然科学基金(the Natural Science Foundation of Guangxi Province of China under Grant No.0481016,桂科自0640034)。
作者简介:金汉均(1963-),男,博士,副教授,研究领域为虚拟现实,遗传算法;王晓荣(1978-),女,硕士研究生,研究领域为虚拟现实。

先要知道 *AABB* 树的构造过程。*AABB* 树的构造有自顶向下和自底向上两种方法。这里采用自顶向下的方法来构造 *AABB* 树,算法如下:

(1)根据根节点 *V* 所包含的基本几何元素的坐标值求出各元素的表现点;

(2)求出节点 *V* 的 *AABB*;

(3)沿着当前的 *AABB* 的最长轴,按三角形的质心划分为两个子集;

(4)把两个子集分别作为根节点,返回步骤(2)。直到每个基本几何元素的 *AABB* 都是叶子节点。

由此得到的 *AABB* 树是一棵完全的二叉树,每个叶子节点仅含有一个三角形。

2.2 存储需求函数

根据 *AABB* 树的构造过程,引入如下存储需求函数 *M*:

$$M=\sum_{i=1}^{m} BI_i+\sum_{j=1}^{n} BL_j+\sum_{k=1}^{n} T_k+\sum_{l=1}^{n} D_l \qquad (1)$$

其中,*M* 为总的内存需求字节数。BI_i 和 BL_j 分别表示存储 *AABB* 树的一个内部节点和叶节点所需的字节数;T_k 和 D_l 分别表示存储一个三角形的顶点值和顶点索引所需的字节数;*m* 和 *n* 分别表示 *AABB* 树的内部节点和叶节点的个数。完全 *AABB* 树中三角形的数目和叶节点的数目相同。

3 存储优化

3.1 *AABB* 包围盒优化

AABB 树的内部节点的常规存储结构包括 *AABB* 包围盒、左孩子索引和右孩子索引 3 个域。物体的 *AABB* 包围盒是包围该物体并平行于坐标轴的最小的六面体,由其基本几何元素集合中每个元素的顶点坐标在 *x*、*y*、*z* 轴上的最大值和最小值来确定。即对物体 *O* 而言,$AABB(O)=(x_{\min}(O),x_{\max}(O),y_{\min}(O),y_{\max}(O),z_{\min}(O),z_{\max}(O))$。$AABB(O)$中每个值由一个 4 字节的浮点数来表示,则存储一个 $AABB(O)$需要 24 个字节。另外,*AABB* 树中给每个内部节点添加了两个 4 字节的分别指向左右孩子节点的索引来增加检索的效率。因此 *AABB* 树中存储每个内部节点共需要 32 个字节。

假定 T_L 和 T_R 和分别为 *AABB* 树中某内部节点 *T* 的左右孩子节点。若 $AABB(T_L)=(-9.8,3.3,4.1,8.6,4.3,10.2)$,$AABB(T_R)=(2.3,11.2,-8.9,4.9,3.5,9.6)$,由 2.1 节 *AABB* 树的构造算法可知,$AABB(T)=(-9.8,11.2,-8.9,8.6,3.5,10.2)$。$AABB(T_L)$与 $AABB(T)$的某些值相同,对于 $AABB(T_R)$也一样。两个孩子节点的 *AABB* 包围盒与父节点的 *AABB* 包围盒最多有 6 个取值不同[2],孩子节点与父节点间存在数据冗余。为了减少冗余数据的重复存储,在父节点中增设一个字节的标志位,使得冗余的数据只在父节点中存储一次。但 *AABB* 树的根节点包围盒的存储例外,其包围盒信息存储完整的 6 个坐标值。根节点结构如表 1 所示。

表 1 根节点结构

范围值(4×6=24 Byte)									
左子节点索引 (4 Byte)	右子节点索引 (4 Byte)	T_{xmin}	T_{xmax}	T_{ymin}	T_{ymax}	T_{zmin}	T_{zmax}	T_{lchild}	T_{rchild}
		标志位(1 Byte)							

标志位的每一位取值为 0 或者为 1。其中 T_{lchild} 和 T_{rchild} 表示节点的左右子节点是否为叶节点,若是则取值为 1,否则取值为 0。标志位前 6 位的取值对应着左右子节点的包围盒的范围值,按照 $x_{\min}$,$x_{\max}$,$y_{\min}$,$y_{\max}$,$z_{\min}$,$z_{\max}$ 的顺序,如果范围值与左子节点的范围值相同,则对应标志位为 1,如果与右子节点的范围值相同,则对应标志位为 0。上例中父节点的标志位前 6 位取值为(1,0,0,1,0,1)。一般而言,标志位前 6 位的 0 和 1 的个数是相同的。但是有两种情况除外,左右子节点在某个轴上有相同的范围值或者一个子节点的包围盒完全包含了另一子节点的包围盒。这两种情况下可以调整子节点包围盒的值使得 0 和 1 的个数相同。

冗余的数据在父节点中存储,对于子节点来说只需存储不相同的范围值。即每个内部节点的包围盒信息只需要用 3 个浮点数(按 *x*、*y*、*z* 轴)共 12 个字节来表示。上例中 $AABB(T_L)$就简化为(3.3,4.1,4.3)。加上 1 个字节的标志位,两个 4 字节的索引,存储一个内部节点共需要 21 个字节,其结构如表 2 所示。

表 2 内部节点结构

范围值(4×3=12 Byte)		
左子节点索引 (4 Byte)	右子节点索引 (4 Byte)	标志位 (1 Byte)

对于一棵有 *N* 个叶节点的 *AABB* 树来说,共有 *N*–1 个内部节点,忽略根节点添加的 1 个字节的标志位,可以节省(*N*–2)*(32–21)个字节。

3.2 叶节点优化

对物体 *A* 和物体 *B* 进行碰撞检测,主要是对两者的 *AABB* 树的节点进行重叠测试,遍历 *AABB* 树进行重叠测试的是叶节点与叶节点,叶节点与内部节点,内部节点与内部节点。

对两叶节点进行重叠测试时先测试叶节点所含的三角形的 *AABB* 包围盒是否相交,若不相交则测试终止;否则,再执行三角形与三角形之间的相交测试。包围盒的求交比三角形的求交简单,时间消耗少,所以先用包围盒求交剔除不相交的叶结点。但是对于三角形相交的叶节点而言,这一步操作增加了时间消耗。应用 Möller 提出的快速三角形与三角形相交测试算法,在对叶节点作重叠测试时我们直接进行三角形间的相交测试。这样就不需要使用其 *AABB* 包围盒信息。同样的,当对物体 *A* 的叶节点和物体 *B* 的内部节点(或者相反)进行重叠测试时,应用 Möller 提出的快速三角形与包围盒相交测试算法,也可以略过包围盒重叠测试的步骤,直接进行三角形与包围盒的相交测试而不增加算法的时间开销。

AABB 树叶节点的常规存储结构包括 *AABB* 包围盒信息和三角形索引两个域。物体 *A* 和物体 *B* 的 *AABB* 树重叠测试过程中涉及到叶节点的重叠测试都可以不用到包围盒信息而直接用三角形进行测试,这样就可以从叶节点的存储结构中删去包围盒信息。叶节点的存储结构中只有三角形的索引信息,此时就不必再用一个独立的节点表示叶节点。我们将三角形索引存放到父节点中,从包围盒树的存储空间中删除叶节点。叶节点的三角形索引在父节点中存放时不需要为父节点分配额外的存储空间,直接替换父节点的孩子索引域为孩子叶节点的三角形索引信息。此时,叶节点的父节点结构如表 3 所示。

表 3 叶节点的父节点结构

范围值(12 Byte)										
左子节点三角形索引 (4 Byte)	右子节点三角形索引 (4 Byte)	标志位(1 Byte)							1	1

图 1 所示为一棵含有 12 个叶节点的 *AABB* 树，需要存储 11 个内部节点,12 个叶节点。删除叶节点结构后,只需要存储 11 个内部节点(图 2),大大节省了存储空间。

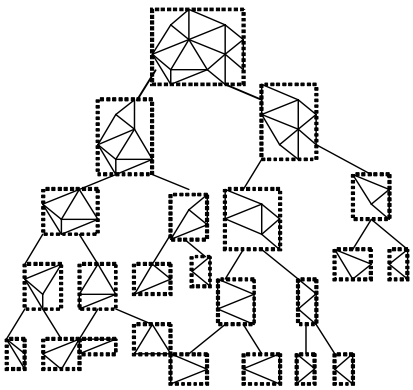

图 1　优化前的 *AABB* 树

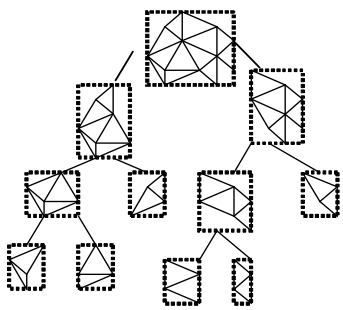

图 2　优化后的 *AABB* 树

一棵含有 N 个叶节点的完全 *AABB* 树共有 $2*N-1$ 个节点,存储 *AABB* 树时不需要为叶节点分配存储空间,则相当于从包围盒树的节点中删掉了一半的节点,这样大大节省了存储空间,减少了一半的内存需求。

经过上述优化之后碰撞检测算法的节点之间的重叠测试算法如图 3。

```
Intersecttest(P,Q)    //P,Q 为 AABB 树的两个内部节点
    {  if (overlap(P->BV,Q->BV))
       {  if (P0 is leaf)  //P0,P1,Q0,Q1 分别为节点 P 和 Q 的子节点
             if (Q0 is leaf) Perform triangle-triangle test
             else Perform triangle-box test
          else if (Q0 is leaf) Perform  triangle-box test
             else Perform  box-box test
          …  //repeat for P0Q1
          …  //repeat for P1Q0
          …  //repeat for P1Q1
       }
    }
```

图 3　节点重叠测试伪码

4　实验与结果分析

SOLID 是一个用于检测多个物体间碰撞情况的算法库,它采用的是 *AABB* 树算法。在 PC 机(CPU P4 1.6 GHz,内存 512 M,显卡 S3 Graphics ProSavageDDR,显存 128 M)平台上修改了代码。应用三角形之间的快速相交测试算法来删除叶节点,然后优化包围盒结构,使其仅需 21 个字节。这使得 *AABB* 比标准的 *AABB* 要小,而且仅需要一个标准树的一半节点。图 4 显示了修改后的算法与原算法的比较结果。

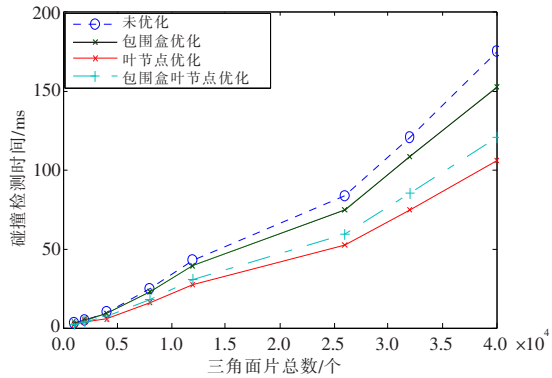

图 4　存储优化前后算法性能比较

包围盒存储结构的优化减少了存储需求,但是同时又使得内部节点包围盒的取值都要参考根节点的包围盒值,这就需要额外消耗一定的检索时间。所以当三角形的数目较少时,对包围盒优化后的算法(简称为算法 1)与未优化算法的执行时间相差无几。但是随着三角形数目的增加,算法 1 的优越性逐渐表现出来。在相同场景中,算法 1 的执行时间要快 10%左右。对叶节点优化后的算法(简称为算法 2)减少了树的遍历深度,节省了存储空间而没有增加额外的时间消耗，加快了算法的执行。由于算法 1 要花费时间查找相关信息,所以结合算法 1 和算法 2 得到的优化算法(简称为算法 3)的执行时间要略慢于算法 2。相同场景中,算法 2 所用的执行时间约为算法 3 所用时间的 0.88，而算法 1 所用时间约为算法 3 所用时间的 1.27 倍。算法 3 将两种优化操作结合起来使得算法在存储空间和执行效率上达到了平衡。

5　总结

碰撞检测算法中一个非常重要的过程就是从内存获取数据进行检测操作,简化算法的存储空间可以减少获取数据时出错的几率。本文对以 *AABB* 树为基础的碰撞检测算法进行了两方面的存储优化:*AABB* 包围盒的优化和叶结点的优化，减少了公式(1)的前两项的值,使得总的存储需求 M 减少。结合这两方面的优化,对一棵含有 N 个叶节点的 *AABB* 树来说,可以节省 $(N-2)\times(32-21)+N\times(24+4)\approx 39N$ 字节,大大减少了存储需求同时又加快了算法的执行。(收稿日期:2006 年 10 月)

参考文献:

[1] Terdiman P.Memory-optimized bounding-volume hierarchies[EB/OL].[2001].http://www.codercorner.com/Opcode.htm.

[2] 潘振宽,李建波.基于压缩的 AABB 树的碰撞检测算法[J].计算机科学,2005,33(2):213-215.

[3] Moller Tomas.A fast triangle-triangle intersection test[J].Journal of Graphics Tools,1997,2(2):25-30.

[4] Moller Tomas.Fast 3D Triangle-Box Overlap Testing[J].Journal of Graphics Tools,2002,6(1):29-33

[5] Gottschalk S,Lin M C,Manocha D.OBBTrees:a hierarchical structure for rapid interference detection[C]//ACM Computer Graphics (Proc of SIGGRAPH'96),1996:171-180

[6] Cohen J D,Lin M C,Manocha D,et al.I-COLLIDE an interactive and exact collision detection system for large-scale environments[C]//Symposium on Interactive 3D graphics,Monterey,CA USA,1995:189-196.

[7] Klosowski,Thomas J.Efficient collision detection for interactive 3D graphics and virtual environments[D],1998.