**家居系统的轻量级快速碰撞检测的设计与实现**

摘 要

置业对每个家庭来说是一项重要的抉择，因此业主在购买前需要对该居所有一个全面深入的了解。在当今这个快节奏的社会中，为了能更为快捷直观地了解整个居所的细节，同时也能更好地融入自己的设计想法，在线漫游成为了一种主要途径。随着互联网的发展，人们对于网络虚拟空间的真实性要求越来越高，Web3D技术得以迅速发展。于是，碰撞检测对于提高三维图形程序的真实性来说就显得尤为重要。当前国内外存在着许许多多的碰撞检测算法，本论文在研究了一些经典的碰撞检测算法后，针对基于Flash3D的在线家居设计系统提出了一个较为完整的轻量级快速动态碰撞检测方法——首先，在2D场景下，采用基于AABB包围盒的四叉树实现粗略碰撞检测；然后，在3D模型下，采用基于OBB包围盒的八叉树实现精确碰撞检测。通过实验结果表明，对于较复杂的家居系统，该算法能较好地平衡计算消耗和检测精度之间的矛盾。最后本论文进行了总结，并提出了下一步的优化可能。

**关键字：**动态碰撞检测, Flash3D，四叉树，八叉树，AABB, OBB，家居系统

**Lightweight and Fast Collision Detection Algorithm for Home Design System**

**ABSTRACT**

Buying a property is a big decision to each family and therefore property buyers tend to have a thorough idea of those properties in advance. Considering the fast living pace in modern society, online browse is becoming a popular method for people to know all the details about those properties quickly and directly and meanwhile to better merge their own designs. With the development of Internet, people require the higher authenticity of virtual cyber space so that Web3D develops rapidly. As a result, collision detection is definitely significant for improving the authenticity of 3D graphic programs. Currently, there exist plenty of collision detection algorithms around the world. After some classic collision detection algorithms are studied, this thesis presents a lightweight and fast dynamic collision detection algorithm for an online home design system on Flash3D platform. Firstly, in the case of the scene in 2D, the algorithm implements the construction of quadtree based on axis aligned bounding box (AABB) to have rough collision detection. Then, in the case of the models in 3D, the algorithm implements the construction of octree based on oriented bounding box (OBB) to have accurate collision detection. The result of experiment illustrates that, for the rather complicated home design system, it can fairly balance the conflicts between computing efficiency and detecting accuracy. At last, this thesis makes a conclusion and put forward the possibility of optimization in the future.

**Key words：**dynamiccollision detection, Flash3D, quadtree, octree, AABB, OBB, home design system

目 录

1 引 言 5

1.1 背景和意义 5

1.1.1 碰撞检测 5

1.1.2 Web3D技术和Flash3D技术 5

1.2 研究现状 6

1.3 相关工作 6

1.4 论文原创性 7

1.5 小结 7

2 基本概念及理论 8

2.1 碰撞检测理论 8

2.1.1 碰撞检测的定义 8

2.1.2 碰撞检测的应用 8

2.1.3 碰撞检测的分类 8

2.1.4 碰撞检测算法的重要性 9

2.2 Flash3D技术 9

2.2.1 Flash3D简介 9

2.2.2 Flash3D技术的优点 9

2.2.3 Flash3D应用的特点 10

2.2.4 不同Flash3D引擎的比较 10

2.3 包围体技术 10

2.3.1 包围体的定义 10

2.3.2 包围体的使用 11

2.3.3 包围体的常见类型 11

2.4 四叉树理论 12

2.4.1 四叉树的定义 12

2.4.2 四叉树划分平面场景的原理 13

2.5 分离轴测试 13

2.6 八叉树理论 15

2.6.1 八叉树的定义 15

2.6.2 实现八叉树的原理 16

2.7 小结 16

3 碰撞检测算法实现 17

3.1 2D粗略碰撞检测 17

3.1.1 AABB包围盒的构建 18

3.1.2 四叉树的创建 20

3.1.3 AABB包围盒的动态相交测试 23

3.2 3D精确碰撞检测 26

3.2.1 OBB包围盒的构建 27

3.2.2 八叉树的构建 30

3.2.3 分离轴测试 31

3.3 碰撞响应 34

3.4 小结 35

4 实验与性能分析 36

4.1 实验环境 36

4.2 四叉树构建的性能 36

4.3 八叉树构建的性能 37

4.4 粗略碰撞检测的效果 37

4.5 精确碰撞检测的效果 38

4.6 算法性能的比较 38

4.7 小结 40

5 总结与展望 41

5.1 总结 41

5.2 展望 41

参考文献 43

附 录 45

谢 辞 55

# 1 引 言

## 1.1 背景和意义

### 1.1.1 碰撞检测

目前，虚拟现实等系统领域的真实性研究主要集中在视觉、听觉和触觉等几个方面。例如，采用几何建模方法来表示虚拟环境中的实体，然后使用纹理映射、消隐和光照计算等方法生成逼真的环境。然而，虚拟现实光有逼真的外观是远远不够的，还需要有逼真的状态变化。例如，虚拟现实中的实体需要具有运动逼真性，即实体的位置、方向和形状等状态变化规律与真实世界中相同或相似。碰撞检测和响应则是虚拟现实中的一项不可或缺的技术，是其基本特性之一——“沉浸感”的重要保证。

碰撞检测的主要作用是判断物体间的重叠性和相交性，其两个重要约束条件是实时性和精确性。针对虚拟环境中的视觉显示要求，碰撞检测的速度至少要达到24Hz，而针对虚拟环境中的触觉显示要求，碰撞检测的速度则要至少达到300Hz，这样才能维持触觉交互系统的稳定性，要达到1000Hz才能获得平滑的效果。一个好的碰撞检测算法有助于创建真实的行为模拟和仿真环境，否则由于虚拟场景复杂度的提高、数据规模的增加，将严重影响碰撞检测的速度和精度，无法满足用户对碰撞检测实时性的需求。例如，网络游戏中一个打斗画面通常会出现几十个游戏角色，虚拟手术系统中存在很多器械的移动，像这种大而复杂的协同环境以及大规模的物理仿真环境就需要进行多物体之间的碰撞检测。因此，如何提高碰撞检测速度同时确保检测的精度是一个十分重要的课题。

### 1.1.2 Web3D技术和Flash3D技术

Web3D又称网络三维，该技术的出现最早可追溯到VRML。VRML（Virtual Reality Modeling Language）即虚拟现实建模语言，开始于20世纪90年代初期。1998年，VRML组织把自己改名为Web3D组织，同时制订了一个新的标准，Extensible3D (X3D)。到了2000年春天，Web3D组织完成了VRML到X3D的转换。X3D整合正在发展的XML、JAVA、流技术等先进技术，包括了更强大、更高效的3D计算能力、渲染质量和传输速度。

Flash3D技术是一种当今主流并且较为成熟的Web3D技术。Flash主要运行在网页浏览器端，依靠着轻量化的模型与数据传输，使得用户只需要打开网页就能进行三维体验，这对于计算机远程教育，网上产品展示和虚拟社区等网络应用都有很大的价值，是未来网络技术的发展方向之一。 2011年Adobe公司推出的Molehill渲染引擎以及AGAL（Adobe Graphics Assembly Language）图形汇编语言，借助GPU强大的浮点运算能力实现开创新的Flash3D技术。各种第三方Flash3D引擎也都更新换代，支持GPU计算，使得渲染的效率更高，能够渲染的场景也更大了。

如今，最广为人知并且使用广泛的Flash3D引擎有：Flare3D、Away3D、 Alternativa3D以及Papervision3D 等等。但是这些第三方Flash3D引擎对于碰撞检测的功能依然比较原始。例如，Flare3D只支持基于包围球和相交射线的碰撞检测，而Alternativa3D也只提供使用椭球的碰撞检测函数。因此，在Flash平台上实现一套低功耗且高精度的快速碰撞检测算法是十分有必要的。

## 1.2 研究现状

由于碰撞检测算法的重要性，目前有许多关于这方面的研究，这对虚拟现实技术的迅猛发展起到了推动作用。模型之间的碰撞检测算法大致可分为三类：几何方法、空间分解法和层次包围盒方法。

几何方法主要分析模型的拓扑结构，通过计算模型间的最近特征对，如模型的点、边、面，来进行碰撞检测。其中典型的算法有Lin-Canny算法和Enhanced GJK算法。Lin-Canny算法通过计算两个物体间最邻近特征的距离来确定他们是否相交。GJK算法通过计算凸体之间的最短距离来判断它们是否发生碰撞。该类算法在物体结构较复杂或者运动较快的情形下，算法效率显著降低，无法满足实时性要求。

空间分解法先将虚拟空间划分为体积相等的单元格，确定空间中各个对象所处的一个或多个单元格，对占据同一单元格或相邻单元格的所有对象对进行碰撞检测。其中比较典型的空间划分方法有k-d树、八叉树、BSP树、四面体网和规则网格等。采用空间分解法可以进一步提高算法的深度，但算法通常只适用于稀疏的环境中分布比较均匀的几何对象间的碰撞检测。

层次包围盒是碰撞检测算法中使用最广发的一种方法，在计算机图形学的许多应用领域中得到深入的研究。将虚拟空间中的对象以层次方式（即包围盒树）组织起来，初步检测时根据包围盒之间的相交测试可以快速排出没有发生碰撞的对象，对包围盒发生相交的对象再进行精确的碰撞检测。

此外，基于图像空间的碰撞检测算法也开始应运而生。该算法利用图形硬件对物体的二维图像采样和相应的深度信息来判别两物体之间的相交情况。这类算法的优势在于能有效利用图形硬件加速技术来减轻CPU的计算负荷，从而达到提高检测效率的目的。算法由于其检测结果的不精确性和对硬件支持的依赖，一直发展比较缓慢。但是随着计算机图形硬件性能的迅速增长，基于图像空间的碰撞检测算法进入了一个新的快速发展阶段。

就总体而言，碰撞检测的主要问题在于准确性和实时性的矛盾。为了解决该问题，以效率换取精度的碰撞检测方法成为目前碰撞检测领域的一个热点。目前有两类典型的方法：第一类，采用基于层次包围盒树的检测方法进行相交测试；第二类，当物体逼近到一定程度，在两个物体上通过采样选取一定量的特征作为碰撞对集合进行相交测试。

在不同的应用场合，碰撞检测对实时性和准确性的要求也不尽相同，因此大部分碰撞检测算法都是面向具体应用场合设计的，没有一种算法可以适用所有的情形。

## 1.3 相关工作

通过查阅相关的资料，发现基于八叉树的碰撞检测算法的提出，对于碰撞检测问题的研究有很大的贡献。但是八叉树的构建需要大量的计算，因此会占用大量时间，对Flash程序的运行效率会有一定的影响，所以本论文将对该算法进行优化，提出一套针对于家居系统的轻量级快速碰撞检测算法——首先，在2D场景下，采用基于AABB包围盒的四叉树实现粗略碰撞检测；然后，在3D模型下，采用基于OBB包围盒的八叉树实现精确碰撞检测，以减轻八叉树的构建对程序执行上的开销，提高碰撞检测的效率，同时确保较高的精度。

## 1.4 论文原创性

鉴于家居系统的复杂性，为了减少模型之间进行碰撞检测的次数，筛选需要进行碰撞检测的对象就成为了算法的关键。本论文对于快速碰撞检测的对象选择主要有以下几点来实现：

（1）对场景的2D俯视面进行四叉树划分，排除距离较远的检测对象。

（2）对剩余对象进行动态相交测试，排除距离较近但是相对运动不会导致碰撞的对象。

考虑到家具的规则性以及上述动态碰撞检测只是预处理阶段，所以将对模型构建AABB包围盒来实现粗略碰撞检测。

在如今的家居设计中，家具的布局有时候比较独特，如果继续使用AABB包围盒进行接下来的3D模型下的精确碰撞检测，那么误差将会比较大。所以，当碰撞检测对象确定之后，为了提高检测的准确性，将对模型构建OBB包围盒并对其进行八叉树的划分，以减少需要进行碰撞检测的模型空间。最后进行逐层扫描，对节点做分离轴测试来确定模型之间是否发生碰撞。

通过分别使用粗略碰撞检测和精确碰撞检测的策略，可以省去许多不必要的计算，弥补已有Flash3D引擎中碰撞检测函数不足的效率和精度。

## 1.5 小结

本章节主要介绍了Web3D技术和Flash3D技术的发展，同时也介绍了碰撞检测的重要意义。之后又对当今国内外已有的一些主要的碰撞检测算法进行了介绍。最后还介绍了本论文所涉及到的关于算法优化的相关工作以及其原创性。

# 2 基本概念及理论

## 2.1 碰撞检测理论

### 2.1.1 碰撞检测的定义

碰撞检测所要解决的是如下看似简单的问题：检测两个（或多个）物体是否相交，并且将确定两个物体是否，何时以及在何处发生碰撞。“是否碰撞”将产生一个布尔值，并计算两个物体是否相交。“何时碰撞”将额外测定碰撞发生的时刻。“何处碰撞”将解释物体如何形成碰撞。一般来讲，以上述给定顺序分析这3类问题是一个比较复杂的过程。

### 2.1.2 碰撞检测的应用

碰撞检测计算往往是各种相关应用程序的基本原理，包括计算机游戏、物理仿真（如计算机动画）、机器人技术、虚拟样机技术以及工程仿真。

在计算机游戏中，碰撞检测将保证真实世界的正确虚拟化（例如，禁止人物穿越墙壁，或者防止人物坠落至地板之下）。同时，碰撞检测还提供了某种视线查询，通知敌人是否发现玩家并发动相应的攻击。

在计算机动画（例如布料的实现）中，碰撞检测常用于约束布料的物理模拟，确保布料以更加真实的方式运动。同时，还可以防止人物运动时产生“脱离”现象。在机器人模拟程序中，碰撞检测还可以用于路径规划，从而使机器人能够避开障碍物。在虚拟样机技术中，物理模型进行批量生产之前，碰撞检测常用于修正计算。另外，碰撞检测还可用于撞击测试和其他类型的工程仿真。某些应用程序，例如路径规划和动画渲染，并不需要其碰撞检测系统的实时特性；而有些应用程序，特别是计算机游戏，则十分强调碰撞检测系统的实时性能。一些基于计算机或游戏机平台的动作类游戏所涉及的模拟计算数据量，则要求以每秒30帧～60帧的帧速率进行计算。在这种实时状态以及游戏和物理引擎中碰撞检测占优的状态下，碰撞检测系统将占用游戏一帧中的大量时间消耗。因此，在计算机游戏中，拙劣的碰撞系统设计将导致其瓶颈的产生。

### 2.1.3 碰撞检测的分类

按检测方式分类，碰撞检测可以分为离散点的碰撞检测和连续碰撞检测（CCD continuous collision detection）。离散点的碰撞检测是指定某一时刻T的两个静态碰撞体，看它们之间是否交迭，如果没有交迭则返回它们最近点的距离，如果交迭则返回交迭深度，交迭方向等。连续碰撞检测则是分别指定在T1、T2两个时刻两个碰撞体的位置，看它们在由T1运动到T2时刻的过程中是否发生碰撞，如果碰撞则返回第一碰撞点的位置和法线。连续碰撞检测是最为自然的碰撞检测，可以大大方便碰撞响应逻辑的编写，可以很容易避免物体发生交迭或者穿越。离散点的碰撞检测则没有那么友好，当检测到碰撞时两个物体已经发生了交迭，如果其中有三角形网格对象那么已经有许多三角形发生了交迭，如何将两个交迭的对象分开并按合理的方式运动是一个挑战。虽然连续碰撞检测是最自然的方式，但它的实现非常复杂，运算开销也很大，所以目前大部分成熟的物理引擎和碰撞检测引擎还是采用了基于离散点的碰撞检测，为了避免物体交迭过深或者彼此穿越，大多都要采用比较小的模拟步长。

### 2.1.4 碰撞检测算法的重要性

碰撞检测过程包括碰撞检测阶段和碰撞响应阶段。碰撞检测阶段确定精确碰撞时间以及位置，如果存在相交图元就进入碰撞响应阶段，也就是发生碰撞后物体所采取的反应。虚拟场景主要依赖精确的碰撞检测来为用户的交互行为、视景中物体的移动提供碰撞避免和交互信息。随着场景变得越来越复杂，场景中物体增加的越来越多，碰撞检测系统需要在不影响帧速的情况下提供精确以及实时的碰撞检测任务，这也就要依赖于一个好的碰撞检测算法来完成。

## 2.2 Flash3D技术

### 2.2.1 Flash3D简介

Flash3D是指基于网页flash播放器播放的并且可交互的实时三维画面信息。开发人员通常可以用ActionScript 3.0进行编程，在flash中创建一个真实的3D环境，用以制作更富有表现力的界面和交互式内容。一个运用Flash3D技术的实例如下：



图2.1 基于Flash3D的家居系统

### 2.2.2 Flash3D技术的优点

（1）基于Flash的Flash3D技术，可以完美的展示3D效果，且无需安装插件。

（2）Flash3D可以在缩放、改变横宽比时，不降低显示的图片质量。

（3）Flash3D采用模型加贴图的方式，各模型只采用一张，使得图片容量下，加载快。

（4）Flash3D交互性强，可以在网页、应用程序和后台程序等进行交互通讯，并可作为程序的一部分嵌入运行。

### 2.2.3 Flash3D应用的特点

（1）实现了多种模式的浏览模式。

（2）拖动模式（场景旋转）适合于浏览纵观全局的浏览方式。

（3）走路模式可模拟在场景中的走动和障碍物的碰撞。

（4）顶视（俯视）图模式适合浏览3D状态下的2D俯视形式。

（5）系统的可扩展性强，支持动态导入3ds、dae模型文件，容易增加各种操作，方便后期系统的扩展。

（6）支持用户自定义模型位置的调整。

（7）支持多级联动。如浏览房产时，进入房间内部浏览，进入物品的详细浏览等。

（8）支持Flash文件、模型文件、图片文件的加密处理，保障文件的安全。

### 2.2.4 不同Flash3D引擎的比较

大部分的第三方Flash3D引擎都支持碰撞检测功能，但是其实现的功能比较简单，都只是支持包围盒之间或者包围球之间的碰撞检测。各大主流Flash3D引擎之间的比较如下：

表2.1 不同Flash3D引擎的比较

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Flare3D | Alternativa3D | Papervision3D | Away3D |
| 是否支持碰撞检测 | 是 | 是 | 是 | 否 |
| 精度 | 包围球 | 椭球 | 包围盒 | / |
| 效率 | 高 | 低 | 高 | / |
| 代码量 | 不开源 | 不开源 | 小 | / |

如表2.1所示，Flare3D只支持基于包围球和相交射线的碰撞检测，而Alternativa3D也只提供使用椭球的碰撞检测函数，而且这两个引擎都不开源。对于开源的Papervision3D，碰撞检测的方法就是运用了基本的包围盒技术。此外，Away3D虽然有包围体的支持，但是需要用户自己编写碰撞检测的脚本。

## 2.3 包围体技术

### 2.3.1 包围体的定义

在[计算机图形学](http://zh.wikipedia.org/wiki/,6;S+j=,@%3EA%17)与[计算几何](http://zh.wikipedia.org/wiki/,6;S,(*%3E)领域，一组物体的包围体就是将物体组合完全包容起来的一个封闭空间。将复杂物体封装在简单的包围体中，就可以提高几何运算的效率。通常简单的物体比较容易检查相互之间的重叠。

一组物体的包围体也是包含一个物体及周围相关环境的封闭空间，因此可以用它来表示一个非空、有限的单一物体。

### 2.3.2 包围体的使用

包围体经常用于加速一些特定的检验过程。

在光线跟踪中，包围体用于光线相交检验，在许多渲染算法中，它又用于视体的检验。如果光线或者视体与包围体没有交叉，那么就不会与包围体内的物体相交。通过这样的相交检验，就可以生成需要显示的物体列表。

在碰撞检测中，如果两个包围体没有相交，那么所包含的物体也就不会碰撞。由于包围体的几何形状较为简单，而物体通常是多边形或者简化为多边形近似的数据结构所组成，所以对于包围体的检验通常要比对于物体本身的检验速度更快。

为了得到复杂物体的包围体，一种常用的方法是用场景图或者如OBB树这样更加专业的包围体层次工具对物体或者场景进行分解。这种做法的基本思想就是将场景组织成一个树状结构，根节点是整个场景，每个叶节点包括一个简单的局部。

### 2.3.3 包围体的常见类型

包围体有许多种不同的构建形式，常见的有以下几种：

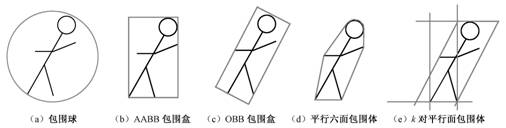


图2.2 2D情况下不同种类的包围盒



图2.3 3D情况下不同种类的包围盒

来源：http://www.cnblogs.com/iamzhanglei/archive/2012/06/07/2539751.html

包围球是一个包容物体的球面。在二维图形中，这是一个圆，包围球就用圆心及半径进行表示。包围球的碰撞检测速度非常快：当两个球心距离不超过半径之和时就会相交。这样包围球就可以用于物体可以向任意方向移动的场合。

包围柱是包容物体的圆柱，在多数应用程序中圆柱的轴与场景竖直方向相同。圆柱是只能绕着纵向轴旋转三维物体，而不能绕着其它轴线旋转，或者只能进行平移。两个轴线为竖直方向的圆柱只有当它们在竖直轴上的投影（即两条线段）以及在水平方向的投影（即两个圆）同时相交的时候，它们才相交。在视频游戏中，包围柱经常作为直立人物的包围体使用。

包围盒是一个包容物体的立方体或者二维长方形。在动态仿真中，对于大致是立方体的物体需要相当高的相交检验精度时，包围盒就优于包围球或者包围柱。这样做的效果是很明显的，例如地面上停放的汽车这种一个物体放在另外一个物体上的场合：用包围球处理就会得到汽车与地面相交的结果，但是经过更加复杂的对于汽车模型的检验就会发现事实并非这样；包围盒立即就可以发现汽车与地面并不相交，这样就可以节省花费较大的检验。

在许多应用程序中，包围盒按照坐标系的坐标轴进行排列，这被称为按坐标轴排列的包围盒（AABB）。为了将AABB 与通用的情况进行区分，有时将任意的包围盒称为定向包围盒（OBB）。AABB检验物体的相交要比OBB更加简单，但是它的缺点是当模型旋转的时候无法随之旋转，而必须重新进行计算。

表2.2 几种典型包围盒的比较（1为最优）

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 构造难度 | 存储量 | 紧密性 | 相交测试复杂度 | 旋转时包围盒更新计算量 |
| AABB | 1 | 2 | 3 | 2 | 3 |
| Spheres | 2 | 1 | 4 | 1 | 1 |
| OBB | 4 | 3 | 2 | 4 | 2 |
| K-DOP | 3 | 4 | 1 | 3 | 4 |

如表2.2所示，AABB包围盒相对来说容易构建且存储量小，但是其紧密性较差且旋转后包围盒的更新计算量较大。而对于OBB包围盒来说，其相交测试的精确度较高，但是包围盒的构造难度和相交测试的复杂度都比较大。所以进行碰撞检测时，就需要根据系统的特性选择合适的包围盒。

一个应用程序中的包围体类型的选择受到不同的因素影响：计算物体包围体所需的计算开销、物体移位、形状或者尺寸发生变化时在程序中更新所需开销以及相交检验所需精度。通常使用几种类型的组合，例如用来快速、大致检验的便宜方法与精确费用较高的方法组合在一起使用。

## 2.4 四叉树理论

### 2.4.1 四叉树的定义

四叉树是一种树形数据结构，它的每个节点下至多可以有四个子节点，每一个节点代表一个区域，每一个区域又可划分为四个小区域，这四个小区域作为四个子节点所代表的区域。区域可以是正方形、矩形或是任意形状。四叉树的二维空间结构和存储结构示意图如下：

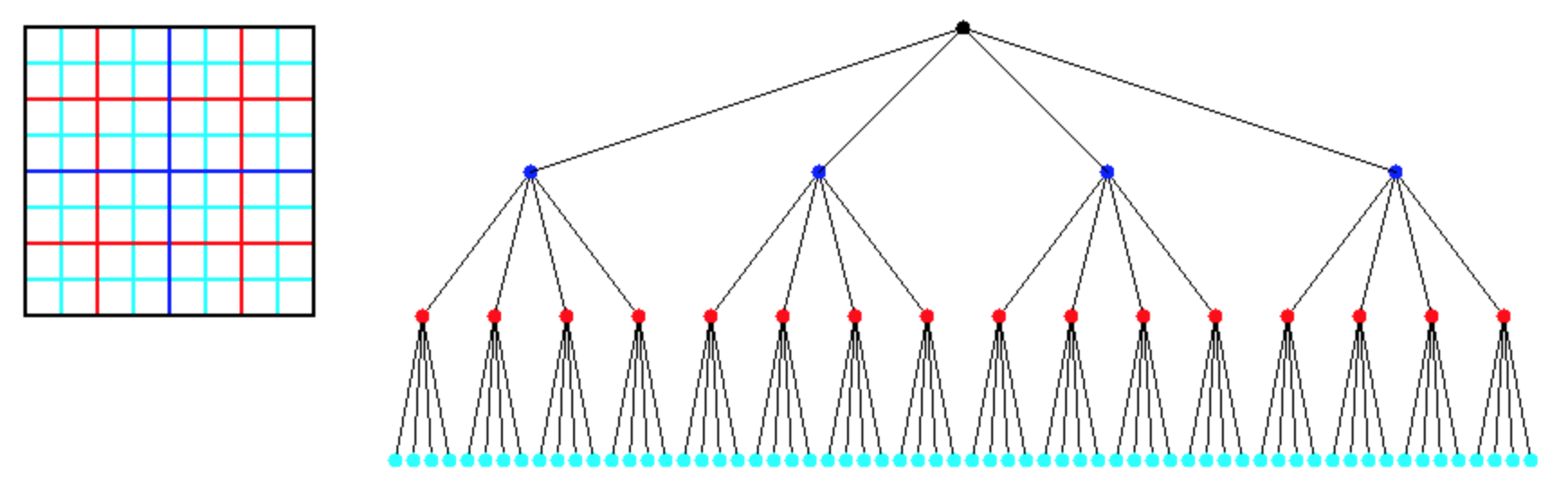


图2.4 左：四叉树的二维空间结构 右：四叉树的存储结构

四叉树是在二维图片中定位像素的唯一适合的算法。因为二维空间中，平面像素可以重复的被分为四部分，树的深度由图片、计算机内存和图形的复杂度决定。

### 2.4.2 四叉树划分平面场景的原理

假设在一个矩形区域里有N个对象，如下左图一个黑点代表一个对象，每个对象的坐标位置都是已知的。通过将所有黑点插入四叉树的方法去划分平面场景，用四叉树的一个节点存储一个对象，构建成右图所示的四叉树。

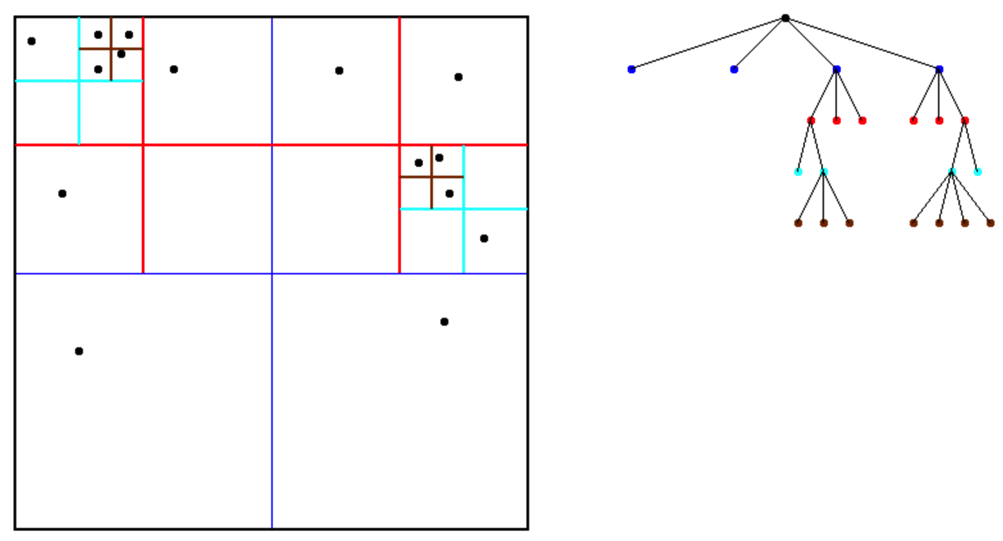


图2.5 四叉树划分平面场景

四叉树的结构比较简单，并且当空间对象分布比较均匀的时候，具有比较高的空间插入和查询效率。

## 2.5 分离轴测试

分离轴测试(Separate Axis Testing)是空间碰撞检测中一个很常用、很有效的判断方法。它的原理来源于集合分析中的“分离超平面定理”（separating hyper-plane theorem）：如果两个集合A和B不相交，那么必定存在一个分离超平面P，并使得A和B分别位于P的不相同的两侧。此定理对于空间数据中的凸集合和凹集合均适用。当数据集是凸集时这里的分离超平面就是一个空间的平面；当数据集是凹集时，这里的分离超平面则不再是一个空间的平面，而是一个空间的曲面。

在这里只讨论使用空间平面作为分离面的两个凸集合的分离测试，因为使用曲面的分离测试要比平面的分离测试复杂很多。对于分离两个凸集合的平面P，我们取平行于其法向量的一条直线作为这两个凸集合的分离轴。在这里，只是用分离面的法向量来限定了这条直线的方向，对于它的位置并末做限制，故而也可以将其看成是一个直线簇。

为了在分离轴上进行集合的相交测试，首先需要观察数据集合在分离轴上具有什么样的空间数学特性。如下图所示，设两个凸集的中心分别为CA、CB，分离它们的平面为P，相应的分离轴为L。首先计算得到两个集合中心之间的距离d，然后再分别求出两个凸集投影在在分离轴方向上的最大半径Ra、Rb（注意：这里的最大半径是向分离轴上投影而得到的，并非任意方向上的最大半径）。通过观察下图可以得到d、Ra、Rb之间的数学关系，即Ra + Rb < d。这即是两个凸集在分离轴上所具有的空间数学特性，也是下面我们利用分离轴进行相交性测试的依据。

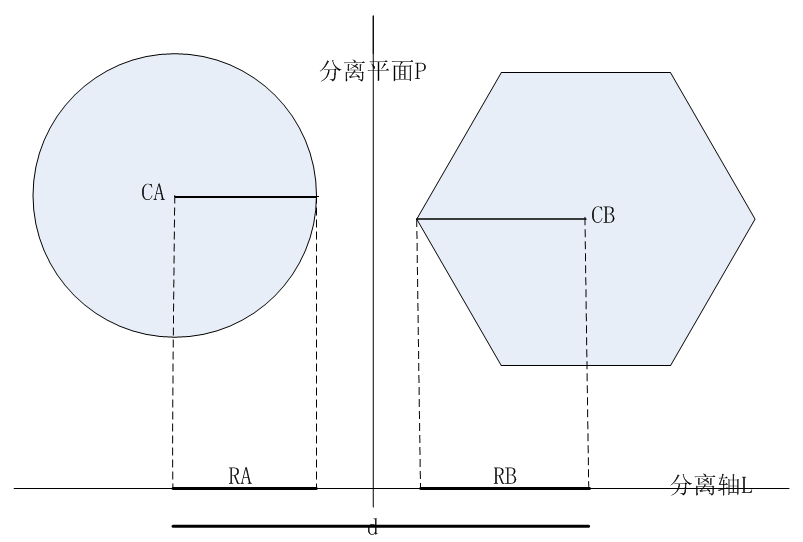


图2.6 分离轴测试方法

对于两个凸集，我们要判断它们的空间关系，只需要寻找它们之间潜在的分离轴，然后判断在分离轴上是否满足分离条件，进而就可以得之。如果两个凸集不相交，那么必定存在能将它们分离开来的平面，也即必定存在满足分离条件的分离轴。这时，我们要做的就是找到所有潜在的分离平面，即潜在的分离轴。对于两个相交的由多面体组成的凸集，它们之间的相交方式可以归结为以下几种：面－面、面－边、面－点、边－边、边－点、点－点相交，而对于其中的点，又可以将它们处理为退化的边，这样一来就可以简化两个凸集间的相交方式为：面－面、面－边、边－边相交。对于上述三种简化的相交方式，我们就可以逐条考虑进而找到所有潜存的分离轴，这样我们就可以得到两个凸集之间潜在的分离平面可能存在于以下位置：

（1）凸集A中每个多边形所在的平面。

（2）凸集B中每个多边形所在的平面。

（3）凸集A中的每条边和凸集B中的每条边之间的公共垂面。

而对应的分离轴的方向为：

（1）凸集A中每个多边形的法向量。

（2）凸集B中每个多边形的法向量。

（3）凸集A中的每条边和凸集B中的每条边之间的公共垂面的法向量，即两条边方向向量的叉乘方向。

假设凸集A中含有FA个面、EA条边，凸集中含有FB个面、EB条边，这样就存在FA + FB + EA \* EB条潜在的分离轴，某些情况下这个数量是巨大的，因而在实际情况中我们往往需要再根据具体几何体元的空间特征对其进行简化。比如，对于两个OBB之间的相交测试，由于OBB的六个面中两两相对的面是平行的，这样我们就可以将面－面之间的分离轴数目减少一半，同样也可以将边－边之间的分离轴数目减少一半，如此一来情况就简单了很多。

总结上述内容，可以得到两个凸集之间用分离轴测试进行相交判断主要有以下几个步骤：

（1）根据凸集的具体情况分析它们之间潜在的所有分离轴，并针对每条分离轴进行分离测试。

（2）对于每条分离轴L，做投影计算得到d、RA、RB，并根据它们之间的关系得到分离关系。若d > RA + RB则分离平面存在，即两个集合之间不相交，此时可以及时退出判断程序；反之，这两个集合在此分离轴下没有分离，但却也不能说明两个集合相交。

（3）若对于所有潜在的分离轴判断的结果不法证实分离平面的存在，那么这两个凸集必定相交。

一般来说，在上述操作中代价最大的就是计算d、RA、RB的值，因为这些值对于每条待测试的分离轴都需要进行更新，而且也是数学计算最多的环节。最简单的方法就是对于每个集合，将集合中的所有顶点向分离轴上进行投影、比较并求最值而得到RA、RB，但是计算的复杂度与集合的规模是线性相关的，故而代价较大。因此，在实现中我们同样需要根据具体的几何集合的空间特性来进行相应的分析而简化这一步骤的操作。比如具体到特殊的AABB、OBB、圆柱、包围球等，我们均可以做出相应的简化进而提高相交测试的效率。

## 2.6 八叉树理论

### 2.6.1 八叉树的定义

八叉树是一种用于描述三维空间的树状数据结构。八叉树的每个节点表示一个正方体的体积元素，每个节点有八个子节点，将八个子节点所表示的体积元素加在一起就等于父节点的体积。其结构如下图：

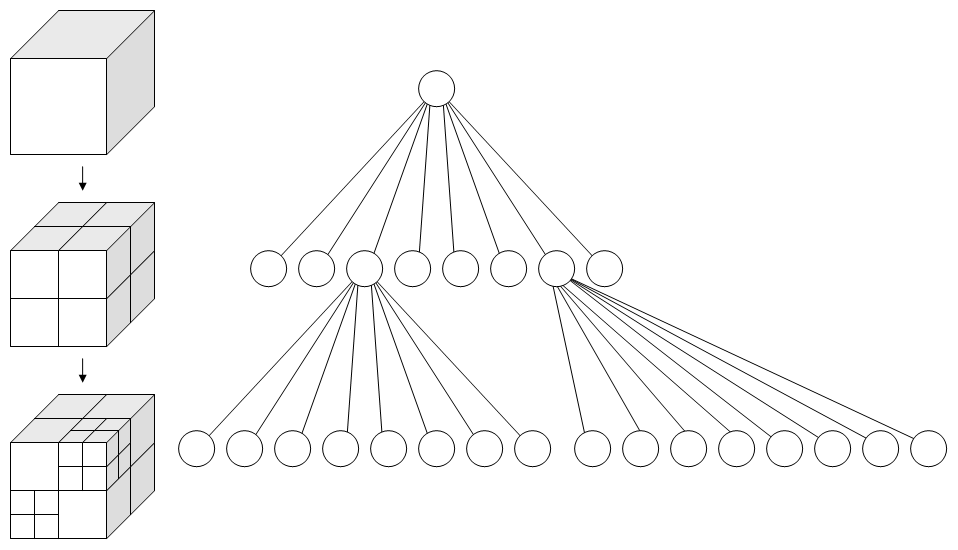


图2.7左：递归子切分一个立方体为多个卦限 右：对应的八叉树

### 2.6.2 实现八叉树的原理

对于一个场景而言，首先建立一个最大的包围盒，并设置为根节点。可直接将此包围盒定义为能包含整个场景的最小正六面体，每一个节点都会被细分成八个字节点，逐层进行下去，可以采用广度优先的遍历顺序进行节点操作。具体实现步骤如下：

（1）设定最大递归深度

（2）找出场景的最大尺寸，并以此尺寸建立第一个立方体

（3）依序将单位元元素丢入能被包含且没有子节点的立方体

（4）若没有达到最大递归深度，就进行细分八等份，再将该立方体所装的单位元元素全部分担给八个子立方体

（5）若发现子立方体所分配到的单位元元素数量不为零且跟父立方体是一样的，则该子立方体停止细分，因为根据空间分割理论，细分的空间所得到的分配必定较少，若是一样数目，则再怎么切数目还是一样，会造成无穷切割的情形。

（6）重复3，直到达到最大递归深度。

## 2.7 小结

本章节首先对碰撞检测以及Flash3D技术做了详细的介绍，之后又介绍了包围体技术、四叉树、八叉树和分离轴测试等概念以及其应用。这些重要的概念在本论文所提出的碰撞检测算法中是相当关键的，所以在此先对这些概念进行基本的介绍。

# 3 碰撞检测算法实现

本论文所提出的针对于家居系统的轻量化快速碰撞检测算法主要有以下两个部分组成：

（1）2D场景下，基于AABB包围盒的粗略碰撞检测。

首先，对场景中所有的模型构建AABB包围盒，并对这些包围盒做俯视面方向上的投影。然后，根据这些投影的位置信息，将平面场景进行四叉树划分并确定模型所处的节点。最后，对可能发生碰撞的包围盒进行AABB包围盒的动态相交测试。如果在平面内，AABB包围盒之间发生了碰撞，则继续进行模型间的碰撞检测。

（2）3D模型下，基于OBB包围盒的精确碰撞检测。

首先，对模型构建OBB包围盒。然后设定八叉树的最大深度，并对包围盒进行八叉树的划分，得到对应其节点的子包围盒。最后，层次遍历模型所有节点中的包围盒，并对包围盒进行分离轴测试。如果叶子节点中的包围盒发生碰撞，则确定模型之间就发生了碰撞。

在判断模型之间的碰撞情况之后，程序将对结果进行碰撞响应，即对模型进行相应的移动。如果不碰撞，则直接移动该模型；否则，需要对所有发生碰撞的模型进行分析，找出最先发生碰撞的模型，计算出新的坐标，然后才能移动模型。

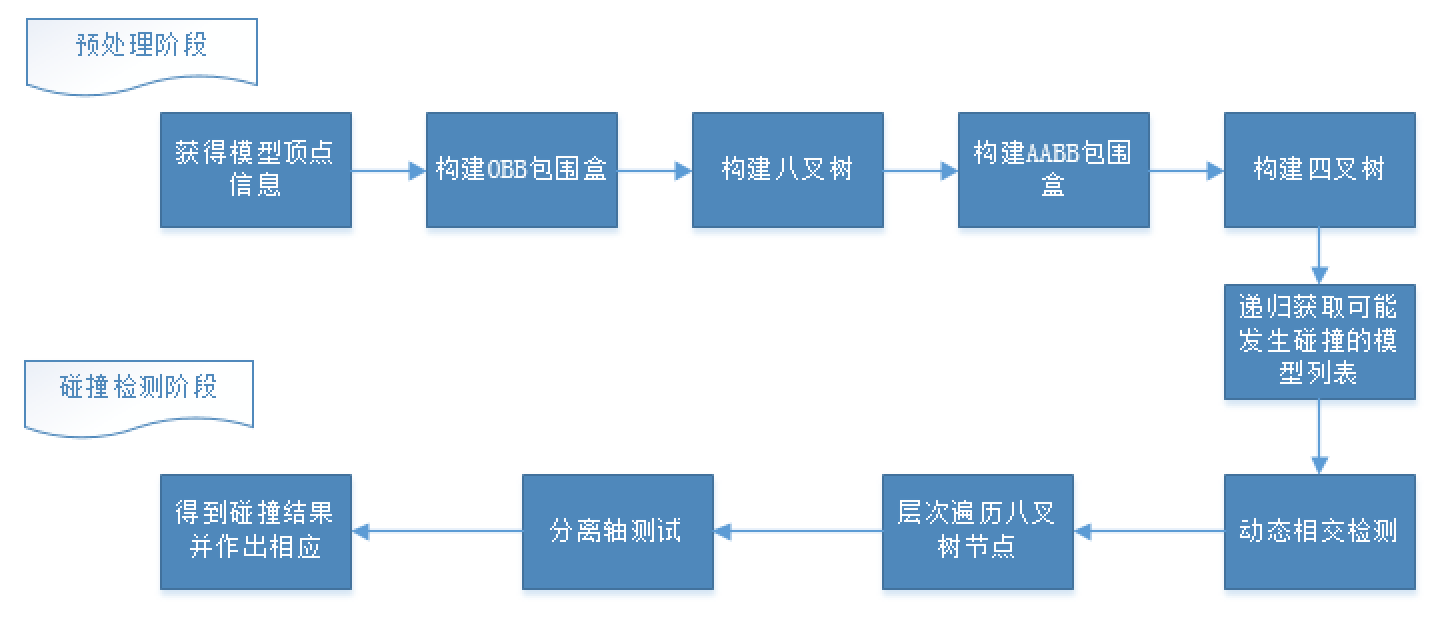


图3.1 碰撞检测算法的主要流程

算法的具体过程在接下来的内容中将会逐一进行详细的介绍。

## 3.1 2D粗略碰撞检测

采用2D碰撞检测的主要目的是为了快速排除不可能发生碰撞的对象，为了减少接下来3D碰撞检测时检测对象的个数，这样可以省去许多没有必要的计算。

考虑到家居系统中的家具模型较为规则，而且此阶段的动态碰撞检测过程只是一个预处理，相对来说对精确度的要求并不是很高，所以本论文选择对模型构建AABB包围盒。与此同时，家具的布局相对也比较整齐，所以选用四叉树进行平面场景划分也比较合理。

2D粗略碰撞检测的具体过程如下：

（1）对所有模型构建AABB包围盒。

（2）对所有包围盒做俯视面上的投影，得到投影的具体位置。

（3）根据位置信息，对平面场景进行四叉树划分，确定每个模型所处的节点。

（4）根据四叉树平面划分的结果，确定进行粗略碰撞检测的对象——同一区域中的所有对象，排除由于距离较远而短时间之内不会发生碰撞的对象。

（5）对这些可能发生碰撞的对象做俯视面坐标上的投影，进行AABB包围盒的动态相交测试，排除距离较近但是相对运动不会导致碰撞的对象。

如果这时候检测出AABB包围盒之间有碰撞，则进行3D下的碰撞检测。否则模型之间就不会发生碰撞。

### 3.1.1 AABB包围盒的构建

#### 3.1.1.1 AABB的定义

轴对齐包围盒（Axis-Aligned Bounding Box），是一个表面法向与坐标轴方向一致的长方体，可以用两个顶点amax和amin来表示物体A的AABB，其中a*i*min<a*i*max，*i*∈{*x*,*y*,*z*}。AABB包围盒的结构如下：

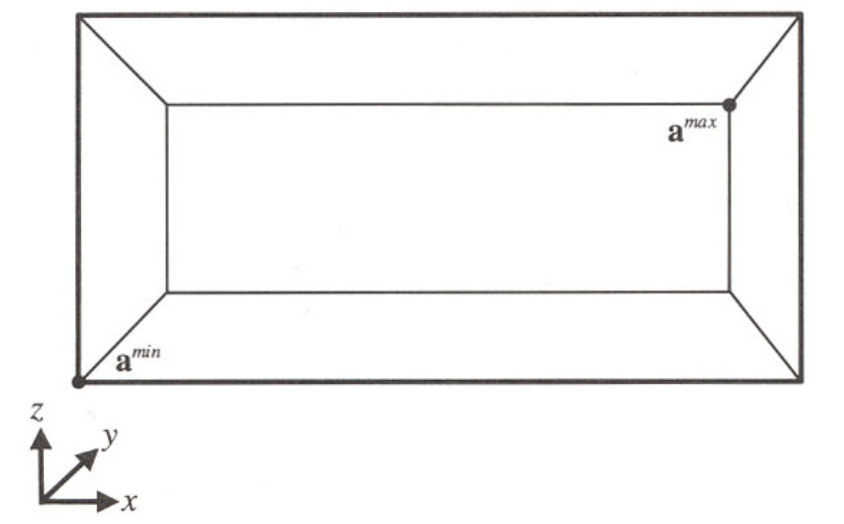


图3.2 AABB包围盒的结构

#### 3.1.1.2 AABB的创建方法

AABB包围盒的创建方法比较简单。首先，获取模型中所有的点的坐标信息。然后，遍历坐标信息并且进行一一比较，分别找到沿每条轴线方向顶点的最大值和最小值。有了这两个特别重要的顶点（minX, minY, minZ）和（maxX, maxY, maxZ）就可以构建一个模型的AABB包围盒了。其算法具体实现如下：

public function initAABB():void{

//获取所有点的坐标信息

var points:Vector.<Number> = mesh.geometry.getAttributeValues(VertexAttributes.POSITION);

var maxX:Number = points[0];

var maxY:Number = points[1];

var maxZ:Number = points[2];

var minX:Number = points[0];

var minY:Number = points[1];

var minZ:Number = points[2];

//寻找沿x轴方向的最大值和最小值

for (var i:int = 3; i < points.length - 2; i += 3){

if (points[i] > maxX)

maxX = points[i];

if (points[i] < minX)

minX = points[i];

}

//寻找沿y轴方向的最大值和最小值

for (var j:int = 4; j < points.length - 1; j += 3){

if (points[j] > maxY)

maxY = points[j];

if (points[j] < minY)

minY = points[j];

}

//寻找沿z轴方向的最大值和最小值

for (var k:int = 5; k < points.length; k += 3){

if (points[k] > maxZ)

maxZ = points[k];

if (points[k] < minZ)

minZ = points[k];

}

//构建AABB包围盒

mesh.boundBox.maxX = maxX;

mesh.boundBox.maxY = maxY;

mesh.boundBox.maxZ = maxZ;

mesh.boundBox.minX = minX;

mesh.boundBox.minY = minY;

mesh.boundBox.minZ = minZ;

}

此外，由于AABB包围盒是沿轴线方向构建的，所以只要知道minX、minY、maxX和maxY这四个点的值，就能很容易得知道俯视面方向上的投影以及俯视面坐标轴方向上的投影。

### 3.1.2 四叉树的创建

#### 3.1.2.1 四叉树划分场景的思想

本论文所构建的四叉树是自顶向下逐步划分的一种树状层次结构。其基本思想是将平面场景递归划分为不同层次的树结构，即将已知范围的空间等分成四个相等的子空间，如此递归下去，直至树的层次达到一定深度或者满足某种要求后停止分割。

四叉树划分平面场景的具体创建步骤如下：首先，设定四叉树最多能够划分的层数以及每个节点最多能够容纳的对象个数。四叉树的根节点就是待划分的最大场景；然后，根据模型对象的位置信息，判断其属于哪块区域，并将其插入该区域所代表的节点。在插入节点之前，需要先查看当前节点有无子节点，如果有就插入其对应的子节点中。运用递归函数一层层查看，直到没有子节点位置，把对象插入到该叶子节点中。在插入叶子节点之前，还需要确认该节点下的对象个数以及当前节点的层数。如果对象数量大于最多能够容纳的对象个数并且当前节点的层数小于最多能够划分的层数，则需要将节点划分出4个子节点。同时还要把当前节点下的对象删除并插入到对应的子节点中。等到所有模型对象都完成插入，四叉树划分场景的工作就结束了。

上述过程中还有一个细节问题，那就是当对象并不能完全属于某一块区域的情况。对于这个问题，本论文采取了特定的方法——并不是把此对象插入到该父节点中，而是将它分别插入它占据的区域所对应的子节点中。这样做会浪费一定的存储空间并且增加了需要进行碰撞检测对象的次数，但是提高了筛选的准确度，而且在之后的碰撞检测过程中也有可能会排除其碰撞的可能性。

当获取与该对象可能发生碰撞的对象列表时，也是通过递归的方法，将该对象所在节点的所有对象以及其子节点下左右对象全部加入到对象列表中。该对象自身可在之后的碰撞检测过程中剔除。

还有一点需要注意的是，由于本论文研究的是动态碰撞检测，所以对于场景下模型的位置信息需要实时更新，这也就使得划分场景的四叉树需要重新构建。于是在之后的构建过程之前，需要将之前的四叉树全部清空。

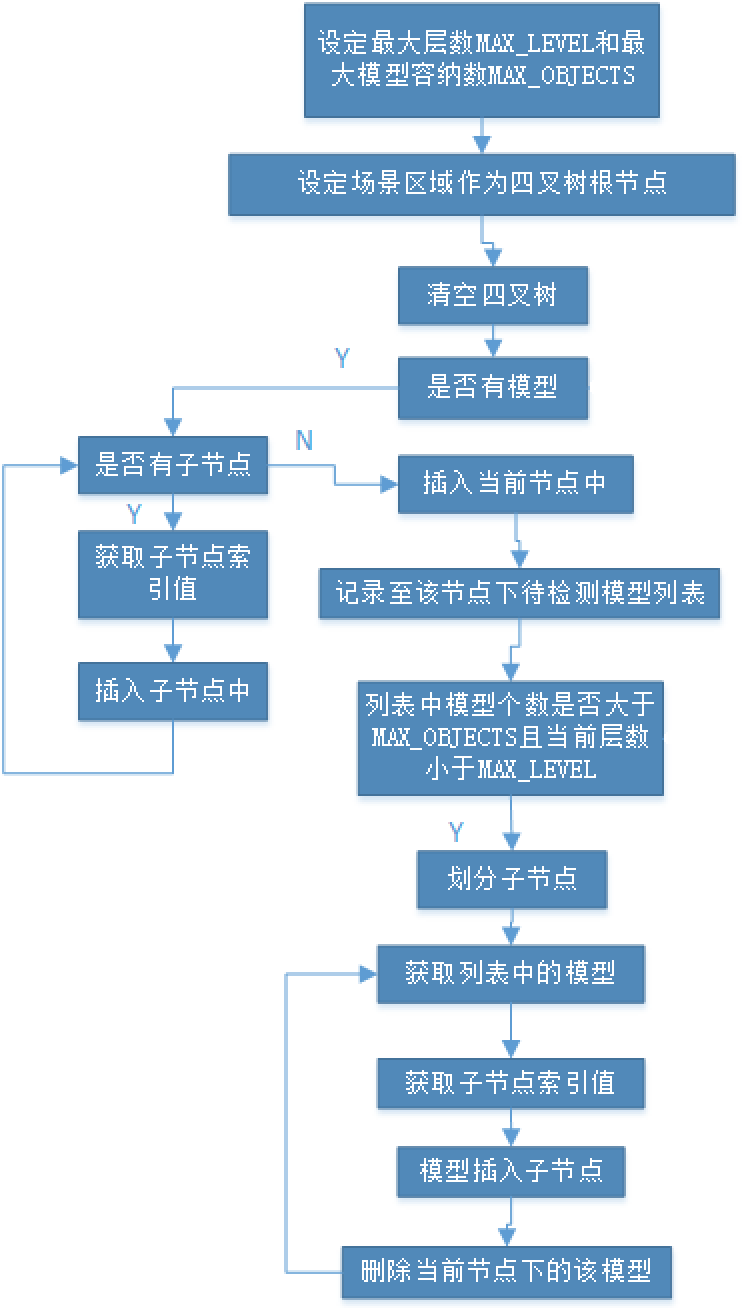


图3.3 四叉树划分场景的算法

#### 3.1.2.2 四叉树划分场景的实现

（1）四分区域的标识：

定义一个平面区域的四个子区域的索引号，左上为0，右上为1，左下为2，右下为3。

（2）四叉树节点数据结构：

public function QuadTree(level:int, bounds:Rectangle){

this.level = level; //当前节点的层数

this.bounds = bounds; //当前节点所表示的区域范围

objects = []; //当前节点内待检测的对象

nodes = new Vector.<QuadTree>(4); //4个子节点

}

（3）四叉树的场景划分：

private var MAX\_OBJECTS:int = 5; //每个节点最多能够容纳的对象个数

private var MAX\_LEVELS:int = 2; //四叉树最多能够划分的层数

//对象插入节点

public function insert(object:Object):void{

//判断当前节点有无子节点

if (nodes[0] != null){

//获取子节点的索引值

var index:Array = getIndex(object);

//对象插入子节点

for (var i:int = 0; i < index.length; i++){

if (index[i] != 0)

nodes[i].insert(object);

}

return;

}

//将该对象记录为当前节点内待检测的对象

objects.push(object);

//判断该节点下的对象个数以及当前节点的层数是否符合要求

if (objects.length > MAX\_OBJECTS && level < MAX\_LEVELS){

if (nodes[0] == null){

//划分子节点

split();

var i:int=0;

while(i < objects.length){

index = getIndex(objects[i]);

//把当前节点下的对象删除并插入到对应的子节点中

for (var j:int = 0; j < index.length; j++){

if (index[j] != 0)

nodes[j].insert(objects.splice(i, 1)[0]);

}

i++;

}

}

}

}

//获取待检测的所有对象

public function retrive(returnObjects:Array, object:Object):void{

var index:Array = getIndex(object);

if (nodes[0] != null){

//获取子节点中待检测的所有对象

for (var j:int = 0; j < index.length; j++){

if (index[j] != 0)

nodes[j].retrive(returnObjects, object);

}

}

for (var i:int = 0; i < objects.length; i++)

returnObjects.push(objects[i]);

}

### 3.1.3 AABB包围盒的动态相交测试

#### 3.1.3.1 动态相交测试的思想

对于包围盒之间的碰撞检测，主要就是运用了分离轴测试的方法。由于AABB包围盒与坐标轴方向相同，所以分离轴存在的情况只有是平面上两个坐标轴的可能。

在动态场景中，物体在时间轴上是连续运动的，用离散的方式很有可能会漏掉相交情形。虽然可以通过加密时间间隔进行相交测试，但是仍会错过部分相交测试，而且还会增加计算负荷。因此需要设计动态相交测试来处理这种情况，即判断物体之间的相对运动状态。对于AABB包围盒的动态相交测试，其基本思想如下：先将待检测的模型做俯视面坐标轴上的投影，然后在各坐标轴方向上分别做相对运动状态的分析。只要有一个轴上投影不相交，那么包围盒之间就不会发生碰撞，也就是说，如果有一个轴上投影重合，这时候还要判断其余轴上的情况。只有所有轴上的投影都重合，那么才能确定包围盒之间发生了碰撞。AABB包围盒在进行动态相交测试时的情景如下：

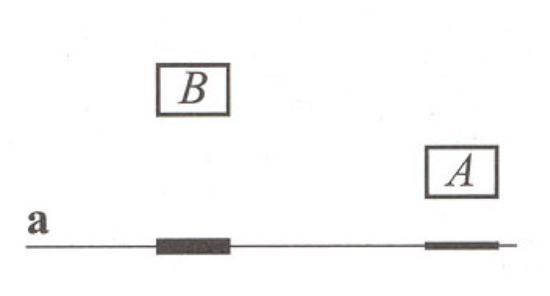


图3.4 投影在轴a上的动态相交测试

2D情况下，动态相交测试的具体步骤如下：首先，计算AABB包围盒之间的速度差（V = Vb – Va），这样就把问题转化成了一个静止的AABB包围盒A和一个运动的AABB包围盒B。然后对俯视面方向上的两个轴分别进行动态相交检测。其中，对每一个轴的检测都要分为v < 0、v > 0和v = 0这三种情况。由于v < 0、v > 0的情况正好相反，所以这里只说明v > 0的情况。另外，由于每个轴上的情况相同，所以这里只说明x轴上的情况，那么包围盒在x轴方向上的相对速度为v.x。

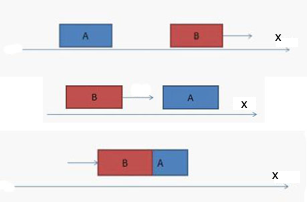


图3.5 当v > 0时，AABB包围盒在x轴上的三种位置情况

（1）当v = 0时，包围盒位置存在以下两种情况：

①当b.min > a.max和b.max < a.min时，投影分离，所以包围盒之间不发生碰撞。

②当b.max > a.min时，投影重合，需要考虑y轴的情况。

（2）当v > 0时，包围盒位置存在以下三种情况：

①当b.min > a.max时，投影分离，所以包围盒之间不会发生碰撞。

②当b.max < a.min时，投影有可能重合。这时需要计算A和B之间的碰撞时间tfirst，并且取最大值。因为如果x轴上的tfirst比y轴上的tfirst小，即使经过了x轴的tfirst，但是y轴上的时间没有超过tfirst，说明投影不重合，所以不发生碰撞。因此碰撞时间应该是tfirstmax。如果tfirstmax > 1，说明A和B之间的距离大于速度差v乘以帧数1得到的距离，所以包围盒之间不发生碰撞。如果tfirstmax < 1，则需要考虑y轴的情况。

③当b.min <= a.max时，投影重合，这时需要计算A和B之间的分离时间tlast，并且与其他轴上的tfirst作比较。如果tfirst > tlast，说明A和B在x轴分离之前，y轴上投影还没有重合，所以包围盒之间不发生碰撞。否则，还需要考虑y轴的情况。

当碰撞检测完成之后，得到的tfirstmax就是包围盒的运动时间。这样在之后移动模型时，模型之间就能紧贴。但是要留一个极小值，否则模型就会粘连。

#### 3.1.3.2 动态相交测试的实现

//动态相交测试，这里只对x轴进行说明，y轴完全相同

public function dynamicCollisionDetection(objectA:Object, objectB:Object, vA:Vector3D, vB:Vector3D, collisionTime:CollTime):Boolean{

//先进行静态相交检测

if (staticCollisionDetection(objectA, objectB)){

collisionTime.tfirst = 0;

collisionTime.tlast = 0;

return true;

}

var v:Vector3D = vB.subtract(vA); //相对速度

var deviation:Number = 0.001; //极小量

collisionTime.tfirst = 0; //碰撞时间

collisionTime.tlast = 1; //分离时间

//包围盒A在轴上的投影

var minXForObjectA:Number = objectA.x;

var maxXForObjectA:Number = objectA.x + objectA.width;

//包围盒B在轴上的投影

var minXForObjectB:Number = objectB.x;

var maxXForObjectB:Number = objectB.x + objectB.width;

//x轴上的情况

if (v.x == 0){

if (minXForObjectB > maxXForObjectA)

return false;

if (maxXForObjectB < minXForObjectA)

return false;

}

if (v.x > 0){

if (minXForObjectB > maxXForObjectA)

return false;

if (maxXForObjectB < minXForObjectA)

collisionTime.tfirst = Math.max((minXForObjectA - maxXForObjectB) / v.x, collisionTime.tfirst) - deviation;

if (maxXForObjectA >= minXForObjectB)

collisionTime.tlast = Math.min((maxXForObjectA - minXForObjectB) / v.x, collisionTime.tlast);

}

if (v.x < 0){

if (maxXForObjectB < minXForObjectA)

return false;

if (maxXForObjectA < minXForObjectB)

collisionTime.tfirst = Math.max((maxXForObjectA - minXForObjectB) / v.x, collisionTime.tfirst) - deviation;

if (maxXForObjectB >= minXForObjectA)

collisionTime.tlast = Math.min((minXForObjectA - maxXForObjectB) / v.x, collisionTime.tlast);

}

if(collisionTime.tfirst > collisionTime.tlast)

return false;

return true;

}

## 3.2 3D精确碰撞检测

采用3D碰撞检测的目的主要是为了对有可能发生碰撞的模型进行进一步的检测，确保碰撞检测的准确度。

由于OBB包围盒是模型的最小包围盒，相对于AABB包围盒来说，进行碰撞检测更为精确，所以本论文将会对模型再一次构建OBB包围盒。另外，在家居系统中，考虑到椅子与桌子之间位置的特殊性——可以一定程度的穿插，如果仅仅只对最外层的包围盒做碰撞检测，显然是不够的。于是，本论文将会采用八叉树的结构对模型进行层次划分，确定模型在包围盒中所占的空间位置，剔除没有模型占据的子包围盒发生碰撞的情况，进而得以提高检测精度。

3D精确碰撞检测的具体过程如下：

（1）对模型构建OBB包围盒。

（2）设定八叉树的最大深度DEPTH。

（3）对包围盒进行八叉树划分，确定模型在包围盒中所占的空间，即确定每个子包围盒所处的节点。

（4）通过递归，层次遍历模型所有节点下的包围盒，对包围盒之间进行分离轴测试。

（5）当八叉树遍历至最大深度时，如果叶子节点中的包围盒依旧会产生碰撞，则可以确定模型碰撞。

### 3.2.1 OBB包围盒的构建

#### 3.2.1.1 OBB的优点

在家居的设计中，家具的摆放位置有可能比较特别，并不和轴线方向相同，如果继续使用AABB包围盒，那么会导致包围盒比家具模型本身大得多。另外，对家具模型的操作并不局限于平移，有时候还会对其进行旋转的操作。由于AABB包围盒的特点是与世界坐标的坐标轴轴对齐，所以不能随着模型的旋转一同改变方向，而是需要进行额外的更新操作，这就会使得包围盒膨胀。由此可见，当上述两点情况发生，AABB包围盒都会导致碰撞检测的不精确。

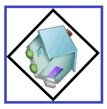


图3.6 模型经过旋转后AABB包围盒的膨胀现象

左图中的黑色框为旋转前的模型的AABB包围盒，当模型经过一定角度的旋转之后，如右图所示，其中的蓝色框为更新后的AABB包围盒，它是根据旋转后的黑色框建立的。可以很明显的发现，AABB包围盒的面积变大了，且随着模型的旋转，包围盒将膨胀得越来越厉害。这明显对碰撞检测精确度造成了很大的影响。

因此，为了提高碰撞检测的精度，在接下来的3D精确碰撞阶段，本论文将会对模型构建OBB包围盒，因为OBB包围盒的特点是方向任意，这个特点决定了OBB包围盒的紧密性较好，且随着模型的旋转而一起旋转。

#### 3.2.1.2 OBB的定义

有向包围盒（Oriented Bounding Box）是一个表面法向两两垂直的长方体，是一个可以任意旋转的AABB。可以用它的中心点bc、三个归一化向量bu、bv、bw以及半边长huB、hvB、hwB来描述。OBB包围盒的结构如下：

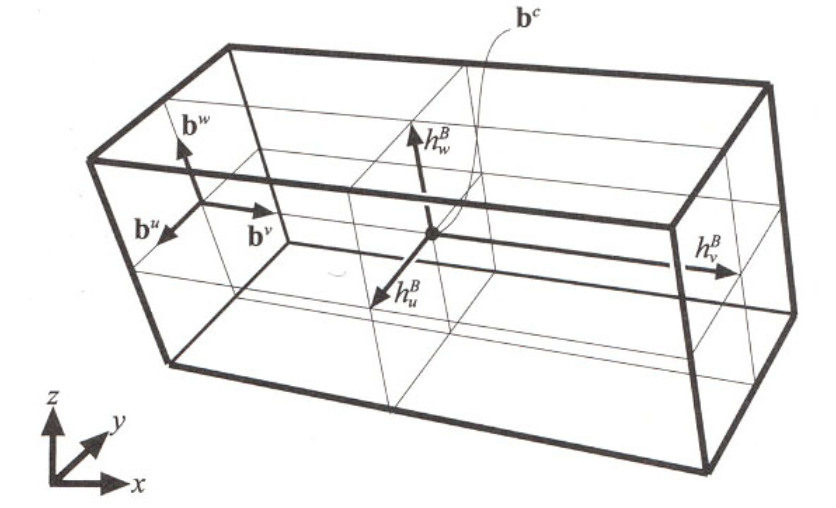


图3.7 OBB包围盒的结构

#### 3.2.1.3 OBB的创建方法

OBB包围盒的创建方法比较复杂。首先，获取模型所有的点的坐标信息；然后，计算协方差矩阵，求其特征向量并归一化，这样就得到了该包围盒的方向向量；最后，将所有点投影到各方向向量上，并求每个方向上的最大值和最小值。每个方向最大值和最小值的平均值决定了该方向的中心，最大值减去最小值的一半为该方向的半边长。这样就可以构建一个模型的OBB包围盒了。其具体算法实现如下：

public function initOBB():void{

//获取模型所有的点的坐标

var points:Vector.<Number> = mesh.geometry.getAttributeValues(VertexAttributes.POSITION);

//创建包围盒取向矩阵

var transform:Matrix4 = getOBBOrientation(points);

transform.createMatrix3D().transpose();

var v:Vector3D = new Vector3D(points[0], points[1], points[2]);

var vecMax:Vector3D = new Vector3D(transform.m00 \* v.x + transform.m01 \* v.y + transform.m02 \* v.z, transform.m10 \* v.x + transform.m11 \* v.y + transform.m12 \* v.z, transform.m20 \* v.x + transform.m21 \* v.y + transform.m22 \* v.z);

var vecMin:Vector3D = vecMax.clone();

//计算每个方向上的最大值和最小值

for (var i:int = 3; i < points.length / 3; i += 3){

var vec:Vector3D = new Vector3D(points[i], points[i + 1], points[i + 2]);

var vect:Vector3D = new Vector3D(transform.m00 \* vec.x + transform.m01 \* vec.y + transform.m02 \* vec.z, transform.m10 \* vec.x + transform.m11 \* vec.y + transform.m12 \* vec.z, transform.m20 \* vec.x + transform.m21 \* vec.y + transform.m22 \* vec.z);

vecMax.x = Math.max(vecMax.x, vect.x);

vecMax.y = Math.max(vecMax.y, vect.y);

vecMax.z = Math.max(vecMax.z, vect.z);

vecMin.x = Math.min(vecMin.x, vect.x);

vecMin.y = Math.min(vecMin.y, vect.y);

vecMin.z = Math.min(vecMin.z, vect.z);

}

transform.createMatrix3D().transpose();

axisX = new Vector3D(transform.m00, transform.m01, transform.m02);

axisY = new Vector3D(transform.m10, transform.m11, transform.m12);

axisZ = new Vector3D(transform.m20, transform.m21, transform.m22);

//计算包围盒的中心以及半边长

center = new Vector3D((vecMax.x + vecMin.x) / 2, (vecMax.y + vecMin.y) / 2, (vecMax.z + vecMin.z) / 2);

extents = new Vector3D((vecMax.x - vecMin.x) / 2, (vecMax.y - vecMin.y) / 2, (vecMax.z - vecMin.z) / 2);

center = new Vector3D(center.x \* transform.m00 + center.y \* transform.m10 + center.z \* transform.m20, center.x \* transform.m01 + center.y \* transform.m11 + center.z \* transform.m21, center.x \* transform.m02 + center.y \* transform.m12 + center.z \* transform.m22);

//向量归一化

axisX.normalize();

axisY.normalize();

axisZ.normalize();

//根据中心、半边长、方向向量构建OBB包围盒

var maxOBB:Vector3D = new Vector3D(center.x + axisX.x \* extents.x, center.y + axisY.y \* extents.y, center.z + axisZ.z \* extents.z);

var minOBB:Vector3D = new Vector3D(center.x - axisX.x \* extents.x, center.y - axisY.y \* extents.y, center.z - axisZ.z \* extents.z);

mesh.boundBox.maxX = maxOBB.x;

mesh.boundBox.minX = minOBB.x;

mesh.boundBox.maxY = maxOBB.y;

mesh.boundBox.minY = minOBB.y;

mesh.boundBox.maxZ = maxOBB.z;

mesh.boundBox.minZ = minOBB.z;

}

经过测试发现，由于家具的形状比较规则，所以构建的OBB包围盒就比较理想，和之前构建的AABB包围盒较为相似。为了减轻之后碰撞检测的计算量，对OBB包围盒的中心、方向向量和

半边长进行了如下的简化：

center = new Vector3D((obb.maxX + obb.minX) / 2, (obb.maxY + obb.minY) / 2, (obb.maxZ + obb.minZ) / 2);

extents = new Vector3D((obb.maxX - obb.minX) / 2, (obb.maxY - obb.minY) / 2, (obb.maxZ - obb.minZ) / 2);

axisX = new Vector3D(1, 0, 0);

axisY = new Vector3D(0, 1, 0);

axisZ = new Vector3D(0, 0, 1);

### 3.2.2 八叉树的构建

本论文构建八叉树的方法是采用了[[1]](#_参考文献)中提出的AABB八叉树构建方法，并在此基础上做了相应的改动，使其能对OBB包围盒进行八叉树的划分。该算法的思想大致如下：

首先，取得模型所有点，进而得到所有的面片。遍历所有面片，将其体素化——计算每个面片所占据的网格，转化成Morton码。接着，对所有的Morton码进行排序，由此建立一个分割信息表。最后，根据分割信息表来建立八叉树，确定面片在包围盒中的具体位置。

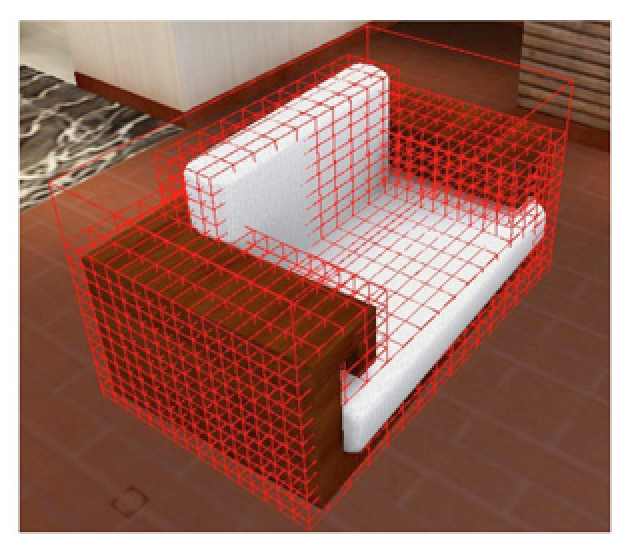


图3.8 模型划分八叉树后的效果

在进行两个OBB包围盒的精确碰撞检测时，需要对相应的两棵八叉树进行层次遍历。本论文采用的遍历顺序如下：八叉树A不动，先对八叉树B进行逐层递归，使其子节点下的包围盒分别和八叉树A的根节点下的包围盒进行碰撞检测。当八叉树B的叶子节点下的包围盒与八叉树A的根节点下的包围盒发生碰撞时，八叉树B叶子节点下的包围盒不动，这时候把八叉树A进行递归，按照相同的方式进行碰撞检测。如果叶子节点中的包围盒发生碰撞，则模型之间就发生了碰撞。

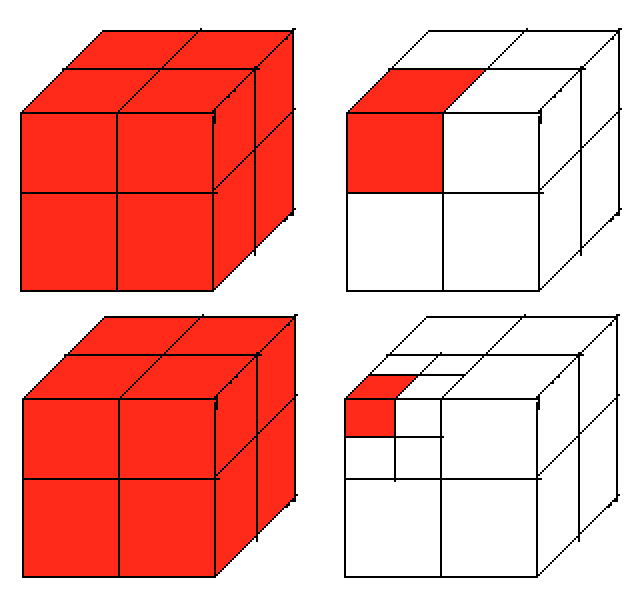


图3.9 两棵八叉树按顺序进行递归检测

对于OBB包围盒之间碰撞检测的方法，则依旧采用了分离轴测试，对于3D情况下的具体检测方法在接下来的小节中进行详细的介绍。

### 3.2.3 分离轴测试

#### 3.2.3.1 分离轴的选取

对于两个OBB包围盒A和B，根据分离轴定理，只要存在一条轴线，使得A和B在此轴线上的投影的“半长”没有重叠，那么这两个OBB包围盒就不会相交。在3D情况下，这样的轴线只可能存在于以下几种情况：

（1）3条取自A的表面，即包围盒A中3个表面的法向量，也就是包围盒A的方向向量。

（2）3条取自B的表面，即包围盒B中3个表面的法向量，也就是包围盒B的方向向量。

（3）9条取自A和B边的组合，即包围盒A的3条边分别与包围盒B的3条边两两组合形成的公垂面的法向量。

这样一共有15条轴线需要进行测试。

#### 3.2.3.2 分离轴测试的思想

首先，取包围盒A的一个面的法向量作为分离轴，然后获取包围盒A和包围盒B的顶点信息，计算在其轴上的最大最小投影值，根据投影值判断该分离轴上的投影是否重合。如果重合，就说明两个OBB包围盒在该轴上不存在分离轴。这样的过程重复三次，即取遍所有包围盒A的法向量进行测试。但是，其过程中一旦有投影不重合，就说明两个包围盒之间分离，该测试终止。如果包围盒A的方向向量都不是分离轴，则将对包围盒B采取以上相同的方法来判断两个包围盒是否发生碰撞。如果经过以上的测试，还是没有找到一条分离轴，那么将进行接下来的分离轴测试。首先，取包围盒A的任意一条边以及包围盒B的任意一条边，计算出两条边的公垂面的法向量，以此作为分离轴，然后计算两个包围盒的顶点在该轴上的最大最小投影值，并判断该分离轴上的投影是否重合。如果重合，就说明两个包围盒在该轴上不存在分离轴。这样的过程重复九次，即取遍所有两个包围盒各自三条边的组合形成的公垂面的法向量进行测试。同样，其过程中，一旦有投影不重合，则测试终止，因为两个包围盒之间有分离面。

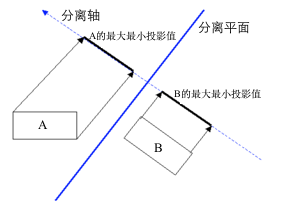


图3.10 取OBB包围盒B的一个面的法向量时，两个包围盒进行分离轴测试的情况

#### 3.2.3.3 分离轴测试的实现

public function collideDetection(objectA:Object, objectB:Object):Boolean{

var min1:Number;

var max1:Number;

var min2:Number;

var max2:Number;

//包围盒A的3个表面的法向量作为分离轴

for (var i:int = 0; i < 3; i++){

//计算包围盒A在某轴上的最大最小投影值

var value1:Array = getInterval(objectA, getFaceDirection(objectA, i), min1, max1);

min1 = value1[0];

max1 = value1[1];

//计算包围盒B在某轴上的最大最小投影值

var value2:Array = getInterval(objectB, getFaceDirection(objectA, i), min2, max2);

min2 = value2[0];

max2 = value2[1];

//判断分离轴上的投影是否重合

if (max1 < min2 || max2 < min1)

return false;

}

//包围盒B的3个表面的法向量作为分离轴

for (var i:int = 0; i < 3; i++){

var value1:Array = getInterval(objectA, getFaceDirection(objectB, i), min1, max1);

min1 = value1[0];

max1 = value1[1];

var value2:Array = getInterval(objectB, getFaceDirection(objectB, i), min2, max2);

min2 = value2[0];

max2 = value2[1];

if (max1 < min2 || max2 < min1)

return false;

}

//包围盒A和包围盒B的各三条边的两两组合形成的公垂面的法向量作为分离轴

for (var i:int = 0; i < 3; i++){

for (var j = 0; j < 3; j++){

//计算公垂面的法向量

var edge:Vector3D = new Vector3D();

edge = getEdgeDirection(objectA, i).crossProduct(getEdgeDirection(objectB, j));

//计算包围盒A在某轴上的最大最小投影值

var value1:Array = getInterval(objectA, edge, min1, max1);

min1 = value1[0];

max1 = value1[1];

//计算包围盒B在某轴上的最大最小投影值

var value2:Array = getInterval(objectB, edge, min2, max2);

min2 = value2[0];

max2 = value2[1];

if (max1 < min2 || max2 < min1)

return false;

}

}

return true;

}

## 3.3 碰撞响应

当碰撞检测阶段完成后，程序需要对检测结果作出相应的反馈，即进入碰撞响应阶段。当模型之间没有发生碰撞，只需要将模型之前的坐标加上移动速度就能得到新的坐标，然后对模型做移动的操作即可。但是如果模型之间有碰撞，则情况较为复杂。需要对所有发生碰撞的情况进行分析，找出最先发生碰撞的模型，计算出新的坐标，然后才能移动模型。

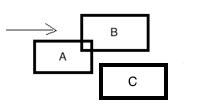


图3.11 碰撞检测顺序的不确定性导致模型之间发生穿透的现象

如上图所示，当模型A向模型B和模型C移动时，首先会发生碰撞的应该是B，然后A就不会再继续移动了。但是如果A和C进行碰撞检测发生在A和B进行碰撞检测之后，之前A运动的时间将被覆盖，那么A就会穿透B继续向C移动，而这种情况是不允许发生的。针对此情况，就需要在计算模型的新坐标的时候取最值。如果v > 0，取最小值；如果v < 0，则取最大值。在所有检测完成之后，根据最后得出的新坐标移动模型。

当模型之间没有碰撞，新坐标的计算方法如下：

newPosition.x = curMesh.x + vX;

newPosition.y = curMesh.y + vY;

newPosition.z = 0;

而如果模型之间有碰撞，则在每次进行碰撞检测的时候，新坐标的计算方式如下：

if(vX >= 0)

newPosition.x = Math.min(curMesh.x + vX \* collisionTime.tfirst, newPosition.x);

else

newPosition.x = Math.max(curMesh.x + vX \* collisionTime.tfirst, newPosition.x);

if(vY >= 0)

newPosition.y = Math.min(curMesh.y + vY \* collisionTime.tfirst, newPosition.y);

else

newPosition.y = Math.max(curMesh.y + vY \* collisionTime.tfirst, newPosition.y);

newPosition.z = 0;

这里要说明的是，在家居系统中，家具的移动都是水平方向的，所以z轴上的值不会发生变化。

## 3.4 小结

本章节对本论文所提出的轻量级快速碰撞检测算法进行了详细的介绍。不仅介绍了各项支撑技术的定义和优点，同时也介绍了技术的具体实现思想以及实现方法（关键方法在正文中已体现，相关方法则可参考附录部分），并配以图示，更清晰地解释了算法的内容。

# 4 实验与性能分析

## 4.1 实验环境

本论文的算法在如下配置的电脑上实现：

（1）操作系统：OS X Yosemite 10.10.3

（2）处理器：2.2 GHz Intel Core i7

（3）内存：16 GB 1600 MHz DDR3

（4）显卡：Intel Iris Pro 1536 MB

开发环境则为Flex Builder 4.6以及Alternativa3D 8.8。另外，进行性能测试所使用的Flash3D系统为图2.1中所示的家居系统。

## 4.2 四叉树构建的性能

（1）对四叉树的构建时间进行检测。影响构建时间的主要因素在于四叉树划分的深度以及子节点中最多能够容纳的模型数。

表4.1 不同深度、不同模型容纳数下四叉树的构建时间（单位：milliseconds）

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 深度\模型数 | 3 | 4 | 5 | 6 |
| 2 | 3 | 3 | 3 | 3 |
| 3 | 4 | 4 | 3 | 2 |
| 4 | 4 | 4 | 2 | 3 |

由表4.1可以发现，当模型容纳数较小时，四叉树的构建时间随着深度的增大而增大；当模型容纳数较大时，四叉树的构建时间基本上差不多。这是因为，家居系统中的家具模型分布比较均匀，当一个区域中的实际模型数超过了允许容纳的模型数，节点会分裂子节点，进而增加了构建时间。但是如果允许容纳的模型数比较多，区域内的实际模型数小于它，那么节点就不会继续分裂，四叉树的深度不会增加，所以建树时间也不会随着定义的深度的增加而增加。基于此原因，本论文在构建四叉树之前，定义深度为2，模型的最大容纳数为5。

（2）对获取可能发生碰撞的家具模型所需的时间进行检测。根据上面的解释，这里只检测深度为2时，不同模型下获取可能发生碰撞的模型列表的时间。

经过测试可以发现，床头柜、椅子以及电视柜等模型获取可能发生碰撞的模型列表的时间为1 milliseconds，而沙发、床、茶几等模型获取列表的时间几乎为0 milliseconds。这是因为前者的位置正好属于子节点的区域，而后者的位置都是占据多个子节点的区域。本论文对此采用的方法是将其分别放入所占区域的子节点中，而且获取列表的方式又是通过迭代的，这样势必会增加获取列表的次数，时间也就会相应的增加。但是由于四叉树深度不大，所以在时间上不会消耗太多。

## 4.3 八叉树构建的性能

本节对八叉树的构建时间进行检测。影响构建时间的主要因素在于八叉树划分的深度。

表4.2 不同深度下八叉树的构建时间（单位：seconds）

|  |  |
| --- | --- |
| 深度 | 构建时间 |
| 2 | 1 |
| 3 | 1 |
| 4 | 4 |
| 5 | 12 |

由表4.2可以发现，随着八叉树的深度增加，构建时间也会增加，而且增加的速度非常快。所以，如果对碰撞检测的精度要求不是太高的话，八叉树的深度为4较为理想，这样能保证一定的执行效率。

## 4.4 粗略碰撞检测的效果

本节对四叉树场景划分后需要进行碰撞检测的模型个数进行测试。对于该家居系统，如果不进行场景划分，那么一个模型需要进行碰撞检测的对象会有场景中的另外40个模型。接下来将在四叉树深度为2的情况下，测试不同模型需要进行碰撞检测的模型个数。

表4.3 四叉树深度为2时，不同模型需要进行碰撞检测的模型个数

|  |  |
| --- | --- |
| 模型 | 检测个数 |
| 沙发 | 12 |
| 床头柜 | 14 |
| 椅子 | 10 |
| 床 | 10 |
| 茶几 | 14 |
| 电视柜 | 12 |

由表4.3可以发现，不管是什么模型，需要进行碰撞检测的模型个数相比之前大幅减少。虽然对于有些模型占据多个子节点的区域时，该模型会被分别放入所占区域的子节点中，增加需要碰撞检测的模型个数，但是由于家具排布比较均匀，增加的个数相对来说并不是太多，而且通过之后的动态相交测试有可能也会进一步减少需要精确碰撞检测的个数。因此，四叉树划分场景来剔除不可能发生碰撞的模型，这个预处理的步骤是相当重要的。

## 4.5 精确碰撞检测的效果

本节对基于OBB包围盒的八叉树碰撞检测的效果进行测试。本论文在之前提过，如果不建立八叉树，只对最外层包围盒进行碰撞检测，那么桌子和椅子之间穿插的情况将无法实现，但是运用了八叉树划分模型，两者之间就可以在一定程度上穿插。所以接下来将会对此进行精度的测试。

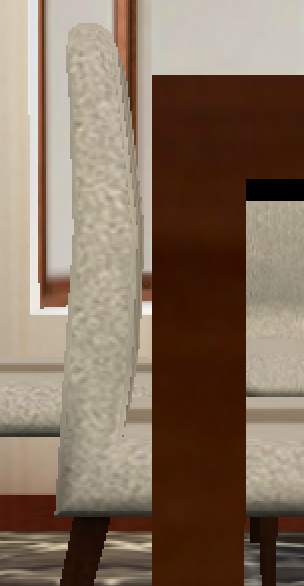
  

图4.1 八叉树深度依次为3、4、5时，椅子和桌子间的距离

由上图可以发现，八叉树的深度越大，椅子和桌子间的距离就越小，即检测精度越高，但是由4.3节所知，深度为5的八叉树构建时间较长。通过上图还可以发现，当八叉树深度为4的时候，虽然桌子和椅子之间的距离相对于八叉树深度为5时的距离要大，但是该距离还是比较近的。由此可见，八叉树深度为4较为理想，这样保证了一定的检测精度。

## 4.6 算法性能的比较

本节将对[[1]](#_参考文献_3)中碰撞检测算法（以下将称其为旧算法）与本论文的碰撞检测算法分别在精度和速度上进行性能的比较。

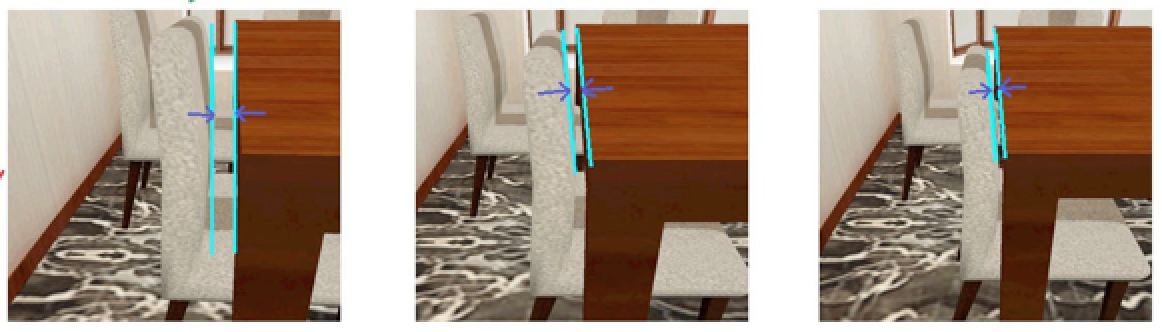


图4.2 旧算法中，八叉树深度依次为3、4、5时，椅子和桌子间的距离

由图4.1与图4.2可以发现，两种算法都能对于椅子和桌子间的情况进行较好的碰撞检测。随着深度的增加，检测的精确度都有很明显的增加。但是，通过对椅子和桌子之间的距离比较，本论文算法的检测精度相对来说更高一些。这是因为旧算法用了AABB包围盒，而本论文则用了OBB包围盒进行精确碰撞检测。

接下来，将对模型移动的实时效率进行测试。这可以用FPS（每秒的帧数，最大值为24）来衡量。

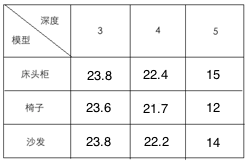


图4.3 不同八叉树深度下，模型移动时的帧速率

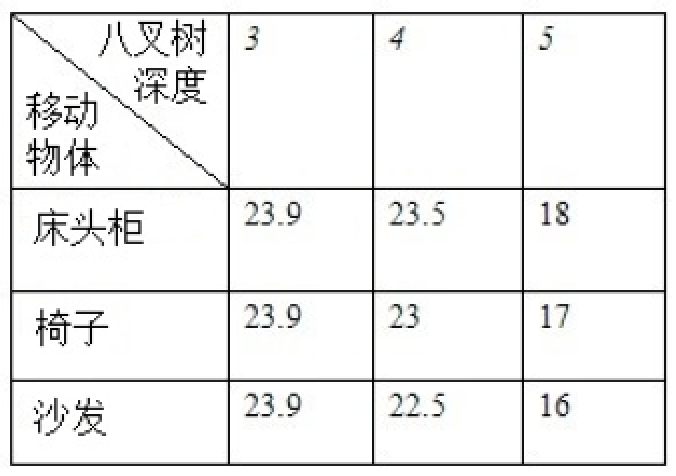


图4.4 旧算法中，不同八叉树深度下，模型移动时的帧速率

由图4.3和图4.4可以发现，两种算法在八叉树深度为3和4时，FPS都较高。但是，当八叉树深度为5时，由于划分的子包围盒过多，导致需要进行碰撞检测的次数增加许多，进而导致FPS骤降。所以，如果对精度要求并不是太苛刻的话，八叉树深度为4的效率是比较理想的。另外，通过对比还能发现，本论文算法测得的帧速率相对于旧算法来说要低一些，这是因为本论文在精确碰撞阶段运用了OBB包围盒的分离轴测试方法，每对包围盒之间都有可能会进行最多15次的检测，所以这样就会很耗时。但是，由于前期粗略碰撞检测阶段已经剔除了许多不需要检测的模型，所以需要进行精确碰撞检测的模型个数不多，计算量并不会骤加，这也就是为什么两个算法在帧速率上的差距并不是太大。所以本论文算法在速度上还是能接受的。

## 4.7 小结

本章节对四叉树和八叉树的构建性能进行了测试，也对粗略碰撞检测以及精确碰撞检测的效果进行了测试。通过测试分析，得出的结论是该算法在家居系统中是可行的。同时还对四叉树和八叉树的深度进行了比较分析，认为当四叉树深度为2，八叉树深度为4的情况下，碰撞检测的效率相对来说比较高，既减少了计算的时间，又确保了检测的精度。

# 5 总结与展望

## 5.1 总结

本论文展示了一种针对于在线家居设计系统的轻量级快速碰撞检测算法。该算法是对之前已有的算法和方法进行的一个改进，并在Flash3D下进行了实现。提出该算法的目的主要是因为构建八叉树进行模型间的碰撞检测需要相当大的计算量，所以有必要对此进行一定程度的优化。该算法的基本内容如下：首先，在2D场景下，用四叉树对场景中的模型位置进行管理，同时使用AABB包围盒进行动态相交测试。这一阶段是为了快速排除不可能发生碰撞的对象，减少之后进行精确碰撞的模型个数，所以对检测精度的要求并不是太高，实现了粗略碰撞检测的目的。之后，对可能发生碰撞的模型进行基于OBB包围盒的八叉树划分，并通过遍历对包围盒之间进行分离轴测试，这样能就能进一步的提高检测的精度。这里要说明的是，模型在包围盒空间内占据的位置是固定的，不会改变的，所以本论文把八叉树的建树过程放在了模型加载的阶段实现。在需要进行精确碰撞检测的时候，只要获取该模型的八叉树即可，这样在一定程度上提高了检测效率。最后，通过实验结果表明，对于较复杂的家居系统，该算法能较好地减少检测带来的计算时间上的消耗，同时也能保证较高的精确度。

## 5.2 展望

尽管本论文所提出的轻量级快速碰撞检测算法在执行效率和检测精度上都比较理想，但是对于该算法的细节部分还是有改进的空间。

（1）当每次模型移动时，场景的四叉树都要重新构建。但是在家居系统中，一般每次只会移动一个模型，其他模型的位置都不会发生改变，所以没有必要重新构建整个场景的四叉树。可以试着对模型在移动前后所处的节点的index做一个记录，分别为nowIndex和lastIndex，并且添加一个delete方法。这样当每次移动中的模型的所属区域发生变化时，只需将原节点下的模型删除，然后插入到新的节点中，同时更新index信息即可。

（2）对于模型并不完全属于某一块区域的情况，现有的做法是将该模型分别插入它占据的区域所对应的子节点中，这样做会一定程度上增加需要进行碰撞检测的对象的个数。可以考虑使用松散四叉树进行场景划分，将划分的区域面积进行一定的扩大，使得模型尽可能的归入其中一个子节点中，减少冗余。考虑到家居系统中，模型的大小基本相同，所以在一定程度上是可行的。

（3）本论文在构建OBB包围盒的时候，采用了计算协方差矩阵然后计算出特征向量的方法来得到包围盒的方向向量，其实这样的效率并不是太高。可以考虑使用Eberly方法，通过迭代计算来构建OBB包围盒。此外，本论文对OBB包围盒进行碰撞检测时使用了分离轴测试的方法，导致了每对包围盒之间最多可能进行15次测试，这样会非常耗时。相信对于OBB包围盒的碰撞检测会有更为简单快速的方法，值得进一步研究。

（4）本论文在构建八叉树的算法上依然沿用了[[1]](#_参考文献_1)中所介绍的方法。在该八叉树的构建时，依旧保留了子包围盒内不存在面片的子节点，这样就增加了子包围盒之间进行碰撞检测的次数。由于分离轴测试的计算量也不小，所以这样的情况会降低碰撞检测的效率。可以尝试使用稀疏体素八叉树来对模型进行划分，这样在构建八叉树时，就能把冗余的一些子节点删除，减少子包围盒之间进行分离轴测试的次数。

（5）当然，还可以尝试用GPU代替CPU去完成一些复杂的计算，例如Morton码的排序过程等。或者可以考虑对运行方式进行优化，即通过并行计算，将繁重的任务分成数个子任务同时计算，加快计算速度。

总之，任何改进的目的都是为了减少CPU的计算量，在确保较高检测精度的情况下尽可能地使检测速度加快，真正的达到低功耗的要求。

# 参考文献

1. 童心路. 基于八叉树的Flash3D碰撞检测算法的设计与实现[D]. 上海：同济大学，2011.
2. 廖若淮. 基于Flash3D的轻量级网上虚拟家居设计平台[D]. 上海：同济大学，2007.
3. 刘天慧. 实时碰撞检测算法技术[M]. 北京：清华大学出版社，2010.
4. 刘为，郝梅. Flash3D核心技术初探[J]. 科园月刊，2011，(3)：55-56.
5. 王晓荣，王萌， 李春贵，等. 基于AABB包围盒的碰撞检测算法的研究[J]. 计算机工程与科学，2007：59-61.
6. 王立文，刘璧瑶，韩俊伟，等. 一种改进AABB包围盒的碰撞检测算法[J]. 计算机工程与应用，2007，43：234-236.
7. 周俊玮，万宇，万旺根，等. 一种基于八叉树的OBB包围盒碰撞检测方法. 计算机软件与应用，2009，26(4)：75-77.
8. 赵波，边馥苓. 面向移动GIS的动态四叉树空间索引算法[J]. 计算机工程，2007，33(15)：86-87.
9. Garcia G, Le Corre J F. A New Collision Detection Algorithm Using Octree Models[C]. //Intelligent Robots and Systems '89. The Autonomous Mobile Robots and Its Applications. IROS '89. Proceedings., IEEE/RSJ International Workshop on. IEEE, 1989:93 - 98.
10. David Eberly. Dynamic Collision Detection using Oriented Bounding Boxes[DB/OL]. (2008-03-02)[2015-05]. http://www.geometrictools.com/
11. Jun Z. An Algorithm of Three Dimensional Medical Collision Detection Based on OBB[J]. Journal of Lanzhou Jiaotong University, 2008.
12. Fan Z, Wan H, Gao S. IBCD: a fast collision detection algorithm based on image space using OBB[J]. The Journal of Visualization and Computer Animation, 2003, 14(4):169–181.
13. Johnny Huynh. Separating Axis Theorem for Oriented Bounding Boxes[DB/OL]. (2009-09-15)[2015-05]. www.jkh.me
14. 陈尚飞. 基于分离轴理论的有向包围盒重叠测试算法[J]. 广西科学院学报，2005，21(3)：196-198.
15. 高博. 基于特征点的碰撞检测算法的研究[D]. 中国石油大学，2008.
16. Bédorf J, Gaburov E, Portegies Zwart S. A sparse octree gravitational N -body code that runs entirely on the GPU processor[J]. Journal of Computational Physics, 2012, 231(7):2825–2839.
17. Chittaro L, Ranon R. Web3D technologies in learning, education and training: Motivations, issues, opportunities[J]. Computers & Education, 2007, 49(1):3–18.
18. 王雁. 基于FLASH的WEB3D应用研究[D]. 北京林业大学，2009.
19. 徐妲，钟绍春，马相春. 基于Flash3D技术的小学立体几何教学平台的设计与实现[J]. 现代教育技术，2013，23(4):119-124.
20. Sulaiman H A, Bade A, Abdullah M H. Computing Distance using Internal Axis-Aligned Bounding-Box for Nearly Intersected Objects[J]. Aip Conference Proceedings, 2014, 1602(1):343-349.
21. Wang W. Study of axis aligned bounding box algorithm[J]. Network Security Technology & Application, 2013.
22. Laine S, Karras T. Efficient Sparse Voxel Octrees[J]. Visualization & Computer Graphics IEEE Transactions on, 2010, 17(8):1048-1059.
23. Baert J, Lagae A. Out-of-core construction of sparse voxel octrees[J]. Computer Graphics Forum, 2013, 33(6):220–227.
24. Cameron S. Collision Detection By Four-Dimensional Intersection Testing[J]. IEEE Trans on Robotics & Automation, 1990, 6(3):291--302.
25. Zhang D, Yuen M M F. Collision Detection for Clothed Human Animation[C]. //Computer Graphics and Applications, Pacific Conference on. IEEE Computer Society, 2000:328.
26. Kim D J, Guibas L J, Shin S Y. Fast Collision Detection among Multiple Moving Spheres \*[J]. IEEE Computer Society, 1997, 4(3):1 - 7.
27. Zou C, Tang Z. The Collision Detection Algorithm Based on the Combination of Two-Dimensional and Dynamic Octree[J]. Environmental Science & Information Application Technology .esiat .international Confer, 2009, 1:470-473.
28. Jian-Bo LI, Pan Z K. The Collision Detection Algorithm Based on Combination of Bounding Volumes and Space Division[J]. Computer Science, 2005.
29. Wang W, Jun MA, Liu W. Research and Application of Collision Detection Based on Oriented Bounding Box[J]. Computer Simulation, 2009, 26(9):180-183.
30. Shen Y, Sun X. Research And Improvement Of Collision Detection Based On Oriented Bounding Box In Physics Engine[C]. //Communication Software and Networks (ICCSN), 2011 IEEE 3rd International Conference on. 2011:73-76.
31. Shi X S, Qiao L H, Zhu Z W. Algorithm of Collision Detection Based on Improved Oriented Bounding Box[J]. Journal of Hunan University, 2014.
32. Wang Z, Jianzhong FU. Oriented Bounding Box and Octree Based Interference Detection Simplified Algorithm of Free-form Surfaces Machining[J]. Machine Tool & Hydraulics, 2006.
33. Wan W G, Zhou J W, Tang J Z. Real-time rendering of 3D large-scale scene based on improved quadtree algorithm[J]. Journal of Computer Applications, 2007, 27(9):2095-2094.
34. He H, Zhang Y, Liu H. A Method Used To Dynamic Scene Management Based On Octree[C]// 2013 10th International Conference Computer Graphics, Imaging and Visualization. IEEE Computer Society, 2007:16-21.
35. CAO Tong, College S H, University B U, et al. Quick Encoding Compression Algorithm of Octree for Modeling Three Dimensional GIS[J]. Journal of Image & Graphics, 2002, 7(1):50-54.

# 附 录

（1）关于四叉树构建的部分代码如下：

//划分子节点

public function split():void{

var subWidth:int = bounds.width/2;

var subHeight:int = bounds.height/2;

var x:int = bounds.x;

var y:int = bounds.y;

nodes[0] = new QuadTree(level + 1, new Rectangle(x, y + subHeight, subWidth, subHeight));

nodes[1] = new QuadTree(level + 1, new Rectangle(x + subWidth, y + subHeight, subWidth, subHeight));

nodes[2] = new QuadTree(level + 1, new Rectangle(x, y, subWidth, subHeight));

nodes[3] = new QuadTree(level + 1, new Rectangle(x + subWidth, y, subWidth, subHeight));

}

//获取对象索引值

private function getIndex(object:Object):Array{

var index:Array = [];

var rectangle:Rectangle = new Rectangle(object.x, object.y, object.width, object.height);

var midX:Number = bounds.x + bounds.width/2;

var midY:Number = bounds.y + bounds.height/2;

var isBottom:Boolean = rectangle.y + rectangle.height <= midY;

var isTop:Boolean = rectangle.y >= midY;

var isLeft:Boolean = rectangle.x + rectangle.width <= midX;

var isRight:Boolean = rectangle.x >= midX;

if (isTop){

if (isLeft)

index = new Array(1, 0, 0, 0); //左上

else if (isRight)

index = new Array(0, 1, 0, 0); //右上

else if (!isLeft && !isRight)

index = new Array(1, 1, 0, 0); //左上、右上

}

else if (isBottom){

if (isLeft)

index = new Array(0, 0, 1, 0); //左下

else if (isRight)

index = new Array(0, 0, 0, 1); //右下

else if (!isLeft && !isRight)

index = new Array(0, 0, 1, 1); //左下、右下

}

else if (!isTop && !isBottom){

if (isLeft)

index = new Array(1, 0, 1, 0); //左上、左下

else if (isRight)

index = new Array(0, 1, 0, 1); //右上、右下

else if (!isLeft && !isRight)

index = index = new Array(1, 1, 1, 1); //左上、左下、右上、右下

}

return index;

}

//清空四叉树

public function clear():void{

objects = [];

for (var i:int = 0; i < nodes.length; i++){

if (nodes[i] != null) {

nodes[i].clear();

nodes[i] = null;

}

}

}

（2）关于动态相交测试的部分代码如下：

//静态相交测试

public function staticCollisionDetection(objectA:Object, objectB:Object):Boolean{

var minXForObjectA:Number = objectA.x;

var maxXForObjectA:Number = objectA.x + objectA.width;

var minYForObjectA:Number = objectA.y;

var maxYForObjectA:Number = objectA.y + objectA.height;

var minXForObjectB:Number = objectB.x;

var maxXForObjectB:Number = objectB.x + objectB.width;

var minYForObjectB:Number = objectB.y;

var maxYForObjectB:Number = objectB.y + objectB.height;

if (maxXForObjectA < minXForObjectB || maxXForObjectB < minXForObjectA){

return false;

}

if (maxYForObjectA < minYForObjectB || maxYForObjectB < minYForObjectA){

return false;

}

return true;

}

（3）关于OBB包围盒创建的相关代码如下：

//计算取向矩阵

private function getOBBOrientation(points:Vector.<Number>):Matrix4{

//生成协方差矩阵

var cov:Matrix4 = getConvarianceMatrix(points);

var evecs:Matrix4 = new Matrix4();

var evals:Vector3D = new Vector3D();

//获取特征向量

evecs = getEigenVectors(evecs, evals, cov);

evecs.createMatrix3D().transpose();

return evecs;

}

//计算协方差矩阵

private function getConvarianceMatrix(points:Vector.<Number>):Matrix4{

var cov:Matrix4 = new Matrix4();

var count:Number = points.length / 3;

var s1:Array = new Array(0.0, 0.0, 0.0);

var s2:Array = new Array(new Array(0.0, 0.0, 0.0), new Array(0.0, 0.0, 0.0), new Array(0.0, 0.0, 0.0));

for (var i:int = 0; i < points.length; i += 3){

s1[0] += points[i];

s1[1] += points[i + 1];

s1[2] += points[i + 2];

s2[0][0] += points[i] \* points[i];

s2[1][1] += points[i + 1] \* points[i + 1];

s2[2][2] += points[i + 2] \* points[i + 2];

s2[0][1] += points[i] \* points[i + 1];

s2[0][2] += points[i] \* points[i + 2];

s2[1][2] += points[i + 1] \* points[i + 2];

}

cov.m00 = (s2[0][0] - s1[0] \* s1[0] / count) / count;

cov.m11 = (s2[1][1] - s1[1] \* s1[1] / count) / count;

cov.m22 = (s2[2][2] - s1[2] \* s1[2] / count) / count;

cov.m10 = (s2[0][1] - s1[0] \* s1[1] / count) / count;

cov.m21 = (s2[1][2] - s1[1] \* s1[2] / count) / count;

cov.m20 = (s2[0][2] - s1[0] \* s1[2] / count) / count;

cov.m01 = cov.m10;

cov.m02 = cov.m21;

cov.m12 = cov.m20;

return cov;

}

//计算特征向量

private function getEigenVectors(evecs:Matrix4, evals:Vector3D, cov:Matrix4):Matrix4{

var n:int = 3;

var nrot:int = 0;

var v:Matrix4 = new Matrix4();

var b:Vector3D = new Vector3D();

var z:Vector3D = new Vector3D();

var d:Vector3D = new Vector3D();

v.createMatrix3D().identity();

for(var ip:int = 0; ip < n; ip++){

setElement(b, ip, getValue(ip + 4 \* ip, cov));

setElement(d, ip, getValue(ip + 4 \* ip, cov));

setElement(z, ip, 0.0);

}

for(var i:int = 0; i < 50; i++) {

var sm:Number = 0.0;

for(var ip:int = 0; ip < n; ip++){

for(var iq:int = ip+1; iq < n; iq++)

sm += Math.abs(getValue(ip + 4 \* iq, cov));

}

if(Math.abs((sm)) < epsilon) {

v.createMatrix3D().transpose();

evecs = v;

evals = d;

return evecs;

}

var tresh:Number = 0.0;

if (i < 3)

tresh = 0.2 \* sm / (n \* n);

for(var ip:int = 0; ip < n; ip++){

for(var iq:int = ip + 1; iq < n; iq++) {

var g:Number = 100.0 \* Math.abs(getValue(ip + 4 \* iq, cov));

var dmip:Number = getElement(d, ip);

var dmiq:Number = getElement(d, iq);

if(i > 3 && (Math.abs(dmip) + g == Math.abs(dmip)) && (Math.abs(dmiq) + g == Math.abs(dmiq)))

setValue(ip + 4 \* iq, cov, 0.0);

else if (Math.abs(getValue(ip + 4 \* iq, cov)) > tresh){

var h:Number = dmiq - dmip;

var t:Number;

if (Math.abs(h) + g == Math.abs(h))

t = getValue(ip + 4 \* iq, cov) / h;

else{

var theta:Number = 0.5 \* h / getValue(ip + 4 \* iq, cov);

t = 1.0 / (Math.abs(theta) + Math.sqrt(1.0 + theta \* theta));

if (theta < 0.0)

t = -t;

}

var c:Number = 1.0 / Math.abs(1 + t \* t);

var s:Number = t \* c;

var tau:Number = s / (1.0 + c);

h = t \* getValue(ip + 4 \* iq, cov);

setElement(z, ip, getElement(z, ip) - h);

setElement(z, iq, getElement(z, iq) + h);

setElement(d, ip, getElement(d, ip) - h);

setElement(d, iq, getElement(d, iq) + h);

setValue(ip + 4 \* iq, cov, 0.0);

for(var j:int = 0; j < ip; j++)

rotate(cov, j, ip, j, iq, s, tau);

for(var j:int = ip + 1; j < iq; j++)

rotate(cov, ip, j, j, iq, s, tau);

for(var j:int = iq + 1; j < n; j++)

rotate(cov, ip, j, iq, j, s, tau);

for(var j:int = 0; j < n; j++)

rotate(cov, j, ip, j, iq, s, tau);

nrot++;

}

}

}

for(ip = 0; ip < n; ip++){

setElement(b, ip, getElement(b, ip) + getElement(z, ip));

setElement(d, ip, getElement(b, ip));

setElement(z, ip, 0.0);

}

}

v.createMatrix3D().transpose();

evecs = v;

evals = d;

return evecs;

}

private function rotate(cov:Matrix4, i:Number, j:Number, k:Number, l:Number, s:Number, tau:Number):void{

var g:Number = getValue(i + 4 \* j, cov);

var h:Number = getValue(k + 4 \* l, cov);

setValue(i + 4 \* j, cov, g - s \* (h + g \* tau));

setValue(k + 4 \* l, cov, h + s \* (g - h \* tau));

}

（4）关于分离轴测试的部分代码如下：

//获得顶点坐标信息

private function getCorners(object:Object, corners:Vector.<Vector3D>):Vector.<Vector3D>{

corners[0] = new Vector3D(object.min.x, object.max.y, object.max.z);

corners[1] = new Vector3D(object.min.x, object.min.y, object.max.z);

corners[2] = new Vector3D(object.max.x, object.min.y, object.max.z);

corners[3] = new Vector3D(object.max.x, object.max.y, object.max.z);

corners[4] = new Vector3D(object.max.x, object.max.y, object.min.z);

corners[5] = new Vector3D(object.max.x, object.min.y, object.min.z);

corners[6] = new Vector3D(object.min.x, object.min.y, object.min.z);

corners[7] = new Vector3D(object.min.x, object.max.y, object.min.z);

return corners;

}

//求面的法向量

private function getFaceDirection(object:Object, index:int):Vector3D{

var corners:Vector.<Vector3D> = new Vector.<Vector3D>(8);

getCorners(object, corners);

var faceDirection:Vector3D = new Vector3D();

var v0:Vector3D;

var v1:Vector3D;

switch(index){

case 0: //与z轴平行的矢量

v0 = new Vector3D(corners[7].x - corners[6].x, corners[7].y - corners[6].y, corners[7].z - corners[6].z);

v1 = new Vector3D(corners[5].x - corners[6].x, corners[5].y - corners[6].y, corners[5].z - corners[6].z)

faceDirection = v0.crossProduct(v1);

faceDirection.normalize();

break;

case 1: //与x轴平行的矢量

v0 = new Vector3D(corners[1].x - corners[6].x, corners[1].y - corners[6].y, corners[1].z - corners[6].z);

v1 = new Vector3D(corners[7].x - corners[6].x, corners[7].y - corners[6].y, corners[7].z - corners[6].z)

faceDirection = v0.crossProduct(v1);

faceDirection.normalize();

break;

case 2: //与y轴平行的矢量

v0 = new Vector3D(corners[1].x - corners[6].x, corners[1].y - corners[6].y, corners[1].z - corners[6].z);

v1 = new Vector3D(corners[5].x - corners[6].x, corners[5].y - corners[6].y, corners[5].z - corners[6].z)

faceDirection = v0.crossProduct(v1);

faceDirection.normalize();

break;

}

return faceDirection;

}

private function getInterval(object:Object, axis:Vector3D, min:Number, max:Number):Array{

var corners:Vector.<Vector3D> = new Vector.<Vector3D>(8);

var val:Array = new Array();

getCorners(object, corners);

min = max = projectPoint(axis, corners[0]);

for (var i:int = 1; i < 8; i++){

var value:Number = projectPoint(axis, corners[i]);

min = Math.min(min, value);

max = Math.max(max, value);

}

val.push(min);

val.push(max);

return val;

}

//计算最大最小投影值

private function projectPoint(point:Vector3D, axis:Vector3D):Number{

var dot:Number = axis.dotProduct(point);

return dot \* point.length;

}

//求边的向量

private function getEdgeDirection(object:Object, index:int):Vector3D{

var corners:Vector.<Vector3D> = new Vector.<Vector3D>(8);

getCorners(object, corners);

var tmpLine:Vector3D;

switch(index){

case 0: //x轴方向单位向量

tmpLine = new Vector3D(corners[5].x = corners[6].x, corners[5].y - corners[6].y, corners[5].z - corners[6].z);

tmpLine.normalize();

break;

case 1: //y轴方向单位向量

tmpLine = new Vector3D(corners[7].x = corners[6].x, corners[7].y - corners[6].y, corners[7].z - corners[6].z);

tmpLine.normalize();

break;

case 2: //z轴方向单位向量

tmpLine = new Vector3D(corners[1].x = corners[6].x, corners[1].y - corners[6].y, corners[1].z - corners[6].z);

tmpLine.normalize();

break;

}

return tmpLine;

}

# 谢 辞

岁月如梭，时光荏苒。当年高考结束后拿到同济大学录取通知书的情景还依旧历历在目，然而一眨眼的功夫，四年的大学本科生活就将画上句号。虽然时间短暂，但是却过得非常充实。在这四年里，不仅学习了软件工程的专业知识，而且还有过难忘的交换生活以及公司实习的经历，这些都将成为我宝贵的回忆。

毕业设计可以说是对大学本科生活交出的最后一份答卷。这份答卷能够顺利提交，并不是我一个人的功劳，而是凝聚了许多人的力量。在这里，我要对他们表达我最深的谢意。

感谢我的导师——贾金原老师，同时也要感谢谢宁老师。从课题的选择到论文的修改，两位老师为我提供了许多专业知识上的指导以及一些富有创造性的建议。他们一丝不苟的作风和严谨求实的态度让我深受感动。另外，由于我的专业方向是企业计算，所以对计算机图形图像没有任何的研究。在此，我要感谢葛一波同学。多亏他不厌其烦地解答我的疑惑，我才能对该领域有一个初步的了解。如果没有得到这些帮助，我也不会顺利得完成毕业设计。

感谢所有教过我课的老师。是你们让我对这个专业有了更全面的了解，也产生了浓厚的兴趣。

感谢我的朋友们。我们在一起分享喜怒哀乐，也漫无目的地交流过各种话题，人生、爱好、困惑等等。正是这些琐碎的小事，能够让大家成为彼此的羁绊，这份友谊将永远铭记在心。

最后还要感谢我的父母。是你们在背后一直默默地支持着我。曾经我有过迷茫，遇到过挫折，没有你们的鼓励，也就没有今天的我。

随着年龄的增长，阅历也越来越丰富，参悟的事也会越来越多。正是这些让我更加坚定地选择了保研，希望将来研究生的生活更加丰富多彩。