凸包围多面体生成算法及应用

（申请清华大学工学硕士学位论文）

培 养 单 位：软 件 学 院
学　　科：软 件 工 程
研　究　生：唐　磊
指 导 教 师：雍　俊　海　教　授

二〇一五年五月

Convex Bounding Polyhedron Construction and its Application

Thesis Submitted to

Tsinghua University

in partial fulfillment of the requirement

for the degree of

Master of Science

in

Software Engineering

by

Tang Lei

Thesis Supervisor : Professor Yong Junhai

May, 2015

关于学位论文使用授权的说明

本人完全了解清华大学有关保留、使用学位论文的规定，即：

清华大学拥有在著作权法规定范围内学位论文的使用权，其中包括：（1）已获学位的研究生必须按学校规定提交学位论文，学校可以采用影印、缩印或其他复制手段保存研究生上交的学位论文；（2）为教学和科研目的，学校可以将公开的学位论文作为资料在图书馆、资料室等场所供校内师生阅读，或在校园网上供校内师生浏览部分内容。

本人保证遵守上述规定。

（保密的论文在解密后应遵守此规定）

作者签名：_____　　导师签名：_____

日　　期：_____　　日　　期：_____

摘 要

在计算机辅助设计、计算机动画和计算机图形学等领域中，包围盒的应用十分广泛，根据其相交测试比原始模型更简单这个性质，常用于在模型之间的相关计算（如几何求交、光线跟踪或者碰撞检测等）之前进行预判剪枝，以提高整体算法的效率。凸包围多面体作为包围盒的推广，对于一般不规则形体，可达到比包围盒更好的紧致程度，因而能够更好地进行预判剪枝以提高算法的整体效率。

本文提出了一种能够快速构造给定点集的指定 k 面的紧致凸包围多面体（k -Convex Bounding Polyhedron，简称 k-CBP）的方法。该方法首先利用一个线性算法对输入点集构造一个近似内凸包，然后根据该近似凸包的面片法向通过 k-means 聚类算法生成构造凸包围多面体的 k 个截面法向，再依次沿各个法向搜索切点构造构成凸包围多面体的截面，最后通过截面求交构成 k-CBP。在搜索截面的过程中，各个法向之间的搜索过程相互独立，可以方便地进行并行搜索，本文分别就基于 OpenGL 着色语言（GLSL）和 计算统一设备架构（CUDA） 两种平台提供了并行加速的方案。在截面求交过程中，本文利用计算几何中的对偶映射技术加快求交过程。实验结果表明，与同类算法相比，本文方法能够更快地构造给定点集更加紧致的凸包围多面体。

碰撞检测算法一直是计算机动画和计算机图形学领域中的研究热点，本文在层次结构包围盒树的基础上，利用构造模型的 k-CBP 进行预判剪枝，提出了一种基于 k-CBP 的碰撞检测算法。该方法首先构造模型的包围盒，通过包围盒相交检测的模型须通过 k-CBP 的相交检测才能进行真实模型的相交检测。本文在 k-CBP 之间的相交测试过程中，利用了一种基于包围盒树的方法和一种基于计算凸体模型之间的最近距离的方法进行相交测试，实验结果表明该方法在静止或运动场景的碰撞检测环境中均达到良好的剪枝效果，有助于提高碰撞检测算法的效率。

关键词：凸包围体；近似凸包；并行计算；碰撞检测

Abstract

Bounding box is widely applied to computer-aided design, computer animation, computer graphics and related fields. Basing on the property that bounding box is simpler than the original model, the calculations between original models can be pruned if the pre-computed bounding boxes do not intersect, so that the efficiency of the algorithms, such as geometry intersection, ray tracing, collision detection and etc., can be increased. Convex bounding polyhedron, a generalization of bounding box, can approximate a model with higher tightness compared to the corresponding bounding box in usual cases, which can be used to prune more calculations.

This paper proposes a method to construct the convex bounding polyhedron which is made of given k faces (k-Convex Bounding Polyhedron, k-CBP for short) from a point set. At first, an algorithm with linear time complexity is used to get an approximate convex hull from the input point set, then the algorithm generates k normals by k-means algorithm from the normals of the approximate convex hull, moreover, it searches the tangent point to generate a cutting plane along each normal, and at last it constructs the convex bounding polyhedron by getting the intersection of the cutting planes. It is easy to use GPU to accelerate the searching process of the cutting planes which is independent between each other parallelly. This paper provides methods on both OpenGL Shading Language(GLSL) and Compute Unified Device Architecture(CUDA) platform in order to parallelize the proposal. Technology of duality mapping in computational geometry is used to accelerate the process of getting the intersection of cutting planes. Experiments show that the algorithm can construct tighter convex bounding polyhedron faster in comparison to the related algorithms from the given point set.

Collision detection has always been a hot topic in the field of computer animation and computer graphics. This paper proposes a method in which the constructed k-CBP is used to avoid redundant operations between models if their k-CBPs do not intersect with each other based on the bounding volume hierarchies. This method first constructs the bounding box of models, the models that run the intersection test should first pass the pruning intersection test of bounding boxes and then the intersection test of k-CBPs. Bounding box hierarchy and a method used to calculate the minimized distance between two convex objects are used during the process of intersection test between k-CBPs. Ex-

periments illustrate that this method can achieve good pruning effects in both static and dynamic environments so that the method will be able to speed up the process of collision detection.

Key words: Convex bounding volume; Approximate convex hull; Parallel computing; Collision detection

目 录

主要符号对照表

BV	包围体 （Bounding Volume）
BVH	层次结构包围体 （Bounding Volume Hierarchies）
CH	凸包 （Convex Hull）
τ	包围体的紧致程度
rtt	渲染到纹理 （Render To Texture）
k-DOP	离散方向 k 面体（k Discrete Orientation Polytope）
k-CBP	凸包围 k 面体（k Convex Bounding Polyhedron）

第 1 章 引言

本章将介绍本文的相关背景，常用的凸包围体种类及相应的应用，常见的碰撞检测算法分类及基于包围体的碰撞检测算法相关流程，最后介绍了本文的结构安排。

1.1 相关背景

随着计算机软硬件相关技术的不断升级和发展，人们的衣食住行都越来越离不开计算机。计算机图形学、计算机动画和虚拟现实等相关技术已经融入人类的日常生活当中，成为人们生活的重要组成部分，如医学领域的虚拟手术、设计领域的三维设计建模辅助造型技术，日常生活中的娱乐活动如电影、游戏等等。

凸包围体技术在计算机图形学领域里的各种算法中发挥着重要作用，如优化渲染和建模过程，加速求交、碰撞检测等算法。以求交算法为例，如果两个模型相交，则对应的凸包围体一定相交，若凸包围体不相交则其对应的原始模型一定不相交。凸包围体作为原始模型的近似，通常情况下，判断凸包围体是否相交比判断原始模型相交更简单，因此利用这个性质就可以加速模型之间的相交检测。计算机图形学领域里常见的包围盒和计算几何领域里的凸包都是凸包围体。凸包围体有多种应用，主要是利用其“凸”的性质和可用来近似被包含模型的特征，应用于模型化简、碰撞检测等。在几何计算过程中，包围体也用于相交等操作中进行预判和剪枝以提升整体算法的运行效率。

碰撞检测问题是计算机图形学、虚拟现实等领域中的研究热点，是计算机模拟真实环境中不可或缺的技术，在物理仿真及游戏领域里应用十分广泛。例如在游戏中，碰撞检测技术增强了游戏的真实性，游戏中的角色行走不可穿墙、角色中弹而亡等等都离不开碰撞检测技术。

下面将分别介绍凸包围体相关技术和碰撞检测相关算法。

1.2 凸包围体

对于“凸”定义[1]如下（以二维欧式空间为例）：给定点集 $S = \{p_1, ..., p_n\} \subseteq E^2$，如果有 $\lambda = (\lambda_1, ..., \lambda_n)^T \in R^n, \lambda_1 + ... + \lambda_n = 1$ 且 $min\{\lambda_1, ...\lambda_n\} \geq 0$，我们称点 $p = (p_1, ..., p_n)\lambda = \lambda_1 p_1 + ... + \lambda_n p_n$ 为 S 的一个凸组合。一个点集 $P \subseteq E^2$ 为凸集合，当且尽当 P 的子集的凸组合仍然是 P 的子集，集合 P 的凸包就是包含 P 的最

小的凸组合。更形象地讲，如果一个二维欧式空间的多边形是凸的，那么连接任意多边形内的两点构成的线段仍然在这个多边形的内部。

凸包围体能用于在原始模型之间的相关计算（遮挡测试、相交测试等）之前进行预处理判断和裁剪的理论基础正是基于此。通常情况下（无特别强调，本文不考虑包围体比原始模型还复杂的情况），凸包围体之间的计算比其包含的原始模型计算更简单，当凸包围体之间没有相交或遮挡时，原始模型一定没有相交或遮挡。

根据具体的应用场景的不同有不同形状的凸包围体，常见的如下。

1.2.1　AABB 包围体

沿坐标轴方向的包围盒（Asix Aligned Bounding Box，简称 AABB）包围体是最常见最简单的包围体，俗称包围盒，因其方向始终沿着坐标轴方向[2] 而得名。在平面中就是包含二维模型的矩形，如图 1.1 所示，为平面图形 Bunny 的 AABB 包围体（二维情况称包围矩形更合适，但为了统一，这里统称包围体，后同）。对应到三维空间中就是沿坐标轴方向包含模型的最小的长方体。

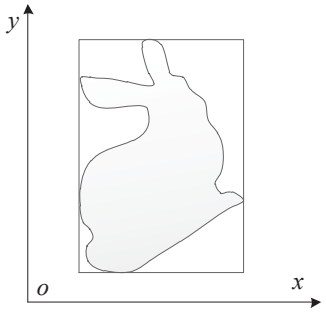


图 1.1　AABB 包围体

表达 AABB 包围体的数据结构通常有 3 种方式：

(1) 存储 AABB 包围体对角线上的两个极点，$Point_{min}$ 和 $Point_{max}$，如著名的计算几何库 CGAL① 以此种方式存储；

(2) 存储 AABB 包围体的中心点和各个方向的半径，$Point_{center}$ 和 $Vector_{radius}$，如碰撞检测库 SOLID[2]；

(3) 存储 AABB 包围体的极小点和各个方向的延伸长度[3]，$Point_{min}$ 和 $Vector_{extent}$。

一方面 AABB 包围体之间的相交测试较简单，以数据结构为存储两个极点为例，只需要比较各个坐标值之间的大小即可。但另一方面，通常情况下与原始模

① CGAL, Computational Geometry Algorithms Library, http://www.cgal.org

型的近似性较差，因其包围体的边沿着坐标轴方向可能会留较多的空白空间。

1.2.2　OBB 包围体

方向包围盒（Oriented Bounding Box，简称 OBB）可看作是 AABB 包围体沿任意方向转动一定角度后构成的包围体，因其方向更加灵活，因此通常情况下， OBB 较 AABB 而言可以更紧致地逼近原始模型。如图 1.2 所示为平面图形 Bunny 的 OBB 包围体。

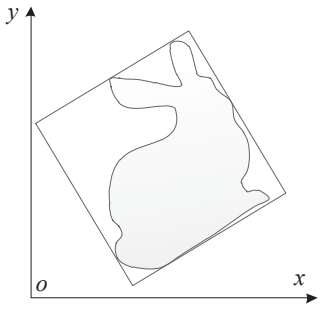


图 1.2　OBB 包围体

与 AABB 类似，OBB 包围体的数据存储方式也可以有多种，最常用的是存储中心点，局部坐标架以及各个方向上的半径[4]。因为 OBB 包围体的方向是任意的，有不少学者研究如何得到更加紧致的 OBB。最早可追溯到由 O'Rourke[5] 提出的 $O(n^3)$ 算法，文献 [6] 中提出了一种方法理论上可以在 $O(n+1/\epsilon^{4.5})$ 复杂度内计算出一个近似最小 OBB（ϵ 为近似误差，即得到的 $V \leq (1+\epsilon)\times V_{min}$，其中 V 表示 OBB 的体积，V_{min} 为最小 OBB 的体积），实际应用中通常实现的算法时间复杂度为 $O(nlogn+n/\epsilon^3)$。C.K.Chan [7] 提出了另外一种迭代的算法可以计算出给定误差范围内最小的 OBB 包围体，该算法适用于点数量较多（超过 1 万个点）的模型。

与 AABB 相比，两个 OBB 包围体之间的相交测试稍复杂，需将二者转换到同一坐标系下进行计算，但其能更好的逼近原始模型。

1.2.3　Sphere 包围体

球形（Sphere）包围体，也称包围球（二维情况下即为包围圆，下同），即用一个半径尽量小的球包围住给定所有点，该问题的求解是计算几何中的经典问题最小包围球的变种。如图 1.3 所示为平面图形 Bunny 的 Sphere 包围体。

表达包围球的数据结构较简单，只需要球心和半径。一种最简单的计算一个包围球的方法是将相应 AABB 包围体的中心点作为球心，半径是为 AABB 各方向半径的最大值，这种方法虽然计算较快，但得到的包围体往往不够紧致。Ritter[67]

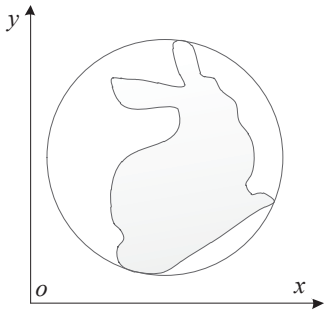


图 1.3　Sphere 包围体

提出了一种线性算法可计算得到更紧致的包围球，该算法简单且较容易实现，在实际应用中较广泛。为了得到最小包围球，E.Welzl[8] 提出了一种线性的随机算法求出容差范围内的最小包围球。T.Larsson[9] 提出了一种快速简单的算法，该算法基于选择 k 个极点生成 $k/2$ 个方向构造包围球，因而称为 EPOS （Extremal Points Optimal Sphere ）算法，用户可以自定义 k 的值，能够给执行时间和包围盒紧致性之间进行调整折衷，提高了灵活性。

一般而言，球形包围体的紧致程度较差，但测试两个 Sphere 包围体是否相交比较简单，只需判断两个 Sphere 包围体的球心之间的距离是否大于其半径之和即可。

1.2.4　k-DOP 包围体

离散方向多面体（Discrete Orientation Polytope，简称 k-DOP）包围体是一个由 $k/2$ 对固定方向的半空间相交构成的凸多面体，其思想最早来源于 TL.Kay[10] 等人提出的用于解决光线追踪的问题，k-DOP 术语是由 J.Klosowski 等人[11] 在 1998 年用于解决碰撞检测问题时提出的，有学者也称 k-DOP 为固定方向凸包（k-Fixed Directions Hull，简称 k-FDH）[12]。在二维（三维）平面上，k-DOP 就是一个由 $k/2$ 对平行的固定方向的边（面）围成的凸多边形（多面体），其中 $k \in \mathbb{N}$，$k \geq 4$（$k \geq 6$）。如图 1.4所示，为平面图形 Bunny 的 8-DOP。

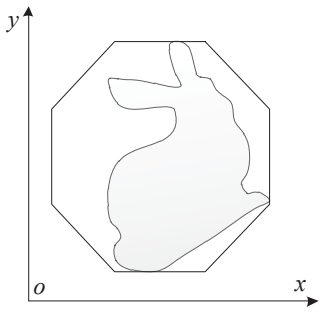


图 1.4　k-DOP 包围体（$k = 8$）

对于 k-DOP 中的每一个方向，可以由相应边（面）的法向决定，其数据结构一般用 $k/2$ 对半平面的法向，再加上模型在各个方向上的投影的极值即可。因为 k-DOP 的方向固定，k 值确定后，相应的法向也随即确定，所以只需要存储各个方向的极值即可。当 k-DOP 用于碰撞检测或可视化时，还需要存储包围体的各个顶点。

算法 1 k-DOP 相交测试算法

输入: 两个 k-DOP 节点 dop_1, dop_2 和 k
输出: 两个 k-DOP 是否相交

```
1: function CheckIntersection(dop1, dop2, k)
2:     for i = 0 → k/2 do
3:         if dop1.min[i] > dop2.max[i] or dop1.max[i] < dop2.min[i] then
4:             return False
5:         end if
6:     end for
7:     return True  // 所有的区间都有重叠，k-DOP 一定相交
8: end function
```

k-DOP 之间的相交测试如算法 1 所示[3]，与 AABB 包围体的相交测试相似，不同点在于 AABB 包围体是针对 x, y, z 供 3 个轴方向进行判断是否有交叉重叠，而 k-DOP 是针对平行于 $k/2$ 对方向的每个方向上检测是否含有重叠交叉。

与其他几种包围体相比较而言，k-DOP 是取包围体存储计算和相交测试耗费时间的一个折衷，且可以通过修改 k 的值来提高包围体的灵活性。在不同的碰撞检测环境中可能选择不同的 k 值能达到更优的结果，文献 [11] 中综合对比了 $k \in \{6, 14, 18, 26\}$ 的实验效果，得出 $k = 18$ 时能达到效率最优的结论，而文献 [13] 从构造时间、性能和存储空间角度等方面进行实验，发现在 $k \in \{6, 8, 14\}$ 中，$k = 6$ 与 $k = 14$ 时能达到基本相当的实验结果，事实上对于不同的碰撞检测测试环境，不同的模型而言，达到最优的结果选取的 k 值可能不都相同，文献 [14] 在综合测试比较后又得出 $k = 24$ 时更容易得到更优的结果。

本文提出的 k-CBP 与 k-DOP 不同之处在于：

(1) k-DOP 中 k 值通常仅局限于少数几个，例如 $k = 6, 8, 14, 18, 20, 26$ 等[11]（文献 [14] 最大支持 $k = 46$），且 k-DOP 中方向是成对平行的，k 值是偶数，而 k-CBP 中的 k 理论上可以是任意的，奇偶都可以；

(2) k-DOP 中的凸多面体方向是固定的即 k 值确定之后，不管输入模型的点集分布如何， k-DOP 的方向始终保持一致，本文则提出了一种能够自适应模型的算法使得构造的凸包围多面体足够紧致；

(3) k-CBP 中 k 取值灵活，必须找出一种算法生成 k 个法向，比只有少数几个可直接通过枚举方向的 k-DOP 取值更复杂。

通过更加紧致的 k-CBP 去近似原始模型，能够达到更好的精度，且输入 k 值可以自由控制，为其能够用于碰撞检测等应用提供足够大的灵活性。

1.2.5　Convex hull 包围体

凸包（Convex hull）是在计算几何中常用的概念，被定义为包含点集模型的最小凸集[1]，因此 Convex hull 包围体是最紧致的凸包围体，能够更好地近似模型本身。包围体的紧致程度以凸包为衡量标准，一个包围体的紧致性 τ 按照如下公式计算：

$$\tau = \frac{V(CH)}{V(BV)}, \tag{1-1}$$

其中，$V(CH)$ 为凸包的体积，$V(BV)$ 为包围体的体积（二维情况 V 表示相应的面积），按照此公式计算出来的 τ 值的大小越接近 1 表明其越紧致，凸包最紧致，其紧致性值为 1。

图 1.5 为 Bunny 二维凸包的示例，从中可以看出，对比以上几个包围体，凸包空白区域是最少的。常用的凸包构造算法[1] 有 Gift Wrapping，时间复杂度最坏

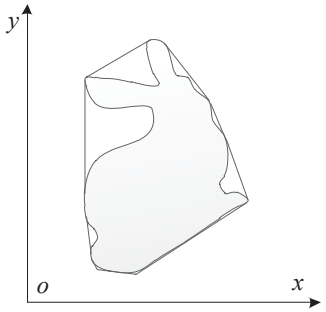
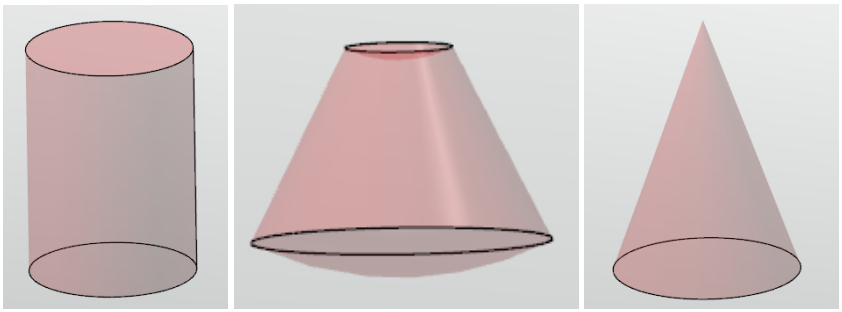
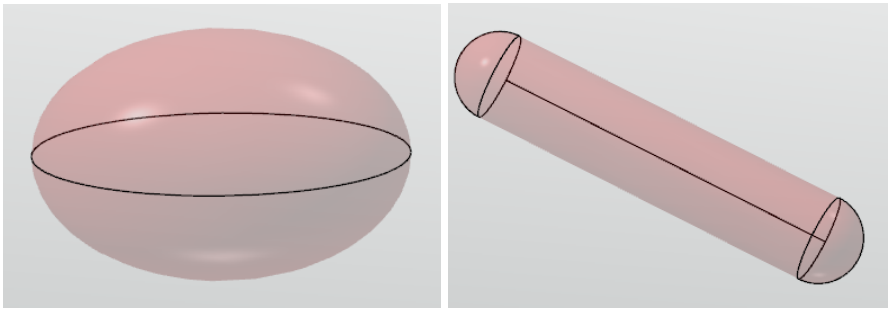


图 1.5　Convex hull 包围体

情况下为 $O(n^2)$，构造凸包的算法时间复杂度下限为 $O(n\log n)$，如 Preparata 提出的分治算法。当模型点集较大时，包围体涉及到的面片数量太多，做相交测试等相关计算时耗费时间较久，同时也耗费较多存储空间。因此 Convex hull 包围体不太适用于大模型。为此有学者在一些应用中提出了近似凸包的概念，近似凸包可分为三类：近似外凸包、近似内凸包和近似凸包[15]，近似内凸包完全在精确凸包内部，近似外凸包完全包含精确凸包，近似凸包即与精确凸包有相交的区域。文献 [16,17] 介绍了两种计算近似内凸包的算法，文献 [18] 介绍了一种计算二维近似外凸包算法，本文算法在生成法向时就借鉴了文献 [16] 中的近似内凸包的算法。



(a) 圆柱形包围体　(b) Sphere-shell 包围体　(c) 圆锥形包围体



(d) 椭球形包围体　(e) Swept-sphere 包围体

图 1.6　其他包围体

1.2.6　其他包围体

除了以上几种常见的凸包围体外，不同学者在特定领域里也研究出以下另外几种包围体：

(1) **Tribox** 可以看作是 k-DOP 的一种特例，其中，二维 Tribox 中的 $k=8$，三维中 $k=18$。文献 [19] 中提出的方法可以方便构造 Tribox 层次结构，并应用于模型分解，当投影方向固定时，对于运动对象，更新其 Tribox 包围体也比较方便。

(2) **Swept-sphere** 是由 Eric.Larsen 等人[20] 提出的一种用于查询两个物体之间精确和近似最近距离的解决方案，因而也用于碰撞检测的应用当中，该包围体有多种变种。原文中给出了球沿直线做拉伸构成的 Line-swept-sphere（如图1.6(e)所示）以及沿矩形拉伸构成的 Rectangle-swept-sphere 等阐述，并提出了构造层次结构包围体的算法，还对哪些包围体合适做碰撞检测，哪些包围体合适做最近距离计算做了分析和说明，更多详细的内容可以参考文献 [20]。

(3) **Sphere-shell** 文献 [21] 给出了一种“球壳”（Sphere-shell）包围体的定义，该包围体由两个同球心不同半径的球中间围成的壳与锥点为球心的锥面相交部分构成，如图1.6(b)所示。这种球壳包围体有利于非结构化的模型、多边形集合（Polygon soups ）模型之间（如简单多边形，样条曲面构成的模型）的最近距离计算或者进行碰撞检测。

(4) **Zonotopes**　Leonidas J.Guibas[22] 等人在 2003 年提出了一种利用对偶的方

法用线段表示三维包围盒的隐式表达法，称其为 Zonotopes（二维称为 Zonogon），这种方法不用显示描述记录构成包围盒的多面体的每个面，节省存储且同样能够在较快的时间内进行碰撞检测等。

(5) **其他**还有如图 1.6(a) 所示的圆柱形[23]、图 1.6(c) 所示的圆锥形[24] 和如图 1.6(d) 所示的椭球形[25] 等等包围体。

1.2.7　包围体的应用

包围体常见的应用领域有真实感渲染如可见面判别、视域剔除[26]，光线追踪[27] 等。在光线追踪算法中，包围体用于检测光线是否与物体相交，如果光线没有与包围体相交，则肯定不会与包围体内的物体模型相交，进而就不用渲染显示该物体。另外更常见的应用就是碰撞检测[28]，大多数碰撞检测算法都用到了各式各样的包围体以加速，如 [29–34] 等。

有不少文献将包围体和模型简化技术结合起来，例如 Kai Huebner 等人[35] 在利用文献 [6] 中提出的构造最小 OBB 包围体算法的基础上将物体分解成多个 OBB 包围体，用这些 OBB 包围体近似原始模型用于机器人抓取应用中，如图 1.7(a) 所示。类似的还有 Sphere 包围体的分解[36]（如图 1.7(b)），Tribox 的分解[19] 等。Jyh-Ming Lien[37] 等人在 06 年提出一种方法可以将给定的多边形（包括

(a) 模型简化成 OBB 包围体[35]　　(b) 模型简化成 Sphere 包围体[36]

图 1.7　包围体应用于模型简化

含有孔洞的）分解成多个凸多边形，并提供不同种精度的分解，可以应用于多层次细节（Level of Detail，简称 LoD）表达，并在 07 年将其扩展到三维[38]，即提出了一种方法将原始实体模型近似为层次结构的凸包围多面体，如图 1.8 所示，对图中的原始模型进行准确凸分解（Exact Convex Decompositions，简称 ECD）将得到 726240 个组成部分，而近似凸分解（Approximate Convex Decompositions，简称 ACD）仅得到 98 个组成部分，同时保持了原始模型的大致形状，这大大加速了渲染过程。这种分解还可以应用于多种应用，例如运动规划（Motion planning），

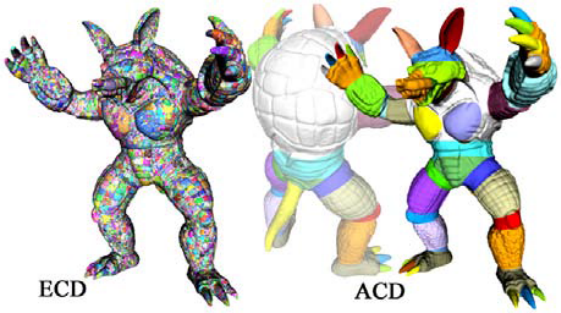


图 1.8　层次结构凸包围多面体的精确构造和近似构造[38]

网格生成（ Mesh generation ），点定位（ Point location ）问题等等，更多内容可以参考 [39] 以及 Jyh-Ming Lien 的博士学位论文[40]。

文献 [41] 利用一种类似的方法将原始 3D 模型分解成一个具有层次结构的模型（图 1.9(a)所示），最顶层将是整个模型的凸包。将该算法应用到 3D 编辑环境中，能够更加方便地让用户选择模型的某个部分进行交互设计。效果如图 1.9(b) 所示，用户选择模型的头部，并进行拖拽能够较快速得到模型新的外观。①

(a) 层次结构的分解　　(b) 区域选择进行交互设计

图 1.9　层次结构的凸包围体及其应用于交互设计[41]

随着计算机软件技术和硬件的发展，有不少并行算法来计算包围体。文献 [42] 利用 Intel 单指令多数据流（Single Instruction Multiple Data，简称 SIMD） SSE 指令集和 OpenMP 实现了对 AABB， OBB 和 *k*-DOP 的计算。文献 [43] 提出了在 统一计算设备架构（Compute Unified Device Architecture，简称 CUDA，后同） 平台上基于 GPU 构造 AABB 包围体树的方法并应用于光线追踪。

1.3　碰撞检测算法

碰撞检测算法是许多应用的基础，例如在 3D 游戏，物理仿真，机器人，虚拟现实等领域中。本节将就碰撞检测问题的分类和基于包围体树的算法进行阐述。

① 详情可参考视频：http://www.readcube.com/articles/10.1111/j.1467-8659.2008.01271.x

1.3.1 碰撞检测算法的分类

碰撞检测算法按照不同的分类标准可以有不同的分类方法。根据其利用的加速结构不同大致可以分为两类，一类是空间划分树（Spatial Partition Tree，例如四叉树、八叉树和 KD 树等），如文献 [44] 就提出了一种基于 BSP 的算法应用于 3D 游戏中的碰撞检测。另外一种就是层次结构包围体（Bounding Volume Hierarchies，简称 BVH，也称包围体树），如在第 1.2 节中提到的 AABB，OBB 等包围体树。包围体树跟常见的空间划分树最大的区别就是，在 BVH 中的两个或多个包围体可以包含相同的空间，而空间划分树的每个划分结构是分离的，且在 BVH 中，父节点不一定必须完整包含子节点，只需要包含子节点对应原始物体的那部分即可[3]。根据参与碰撞检测的物体模型的表现形式，又可以分为凸体模型或凹体模型，多边形或者三角网格模型，构造实体几何（Constructive Solid Geometry，简称 CSG）模型，隐式方程或者参数表达曲面模型等，例如文献 [45] 就讨论了计算机动画系统中基于 CSG 模型的碰撞检测算法，GJK（Gilbet Johnson Keerthi） 算法[46,47] 是用来计算凸体模型之间的距离，因而常用于针对凸体模型之间的碰撞检测算法。根据碰撞检测的模拟环境划分，又可以分为成对（Pair Processing）碰撞检测和多体（Nbody Processing）碰撞检测，刚体和柔性模型的碰撞检测以及静态和动态连续碰撞检测[48]。连续的碰撞检测算法多应用于在物理仿真领域，检测到模型发生碰撞后还要根据碰撞点等信息作出逼近真实情况的响应，如布料模拟[49,50]，头发模拟[51,52] 等。文献 [53] 从碰撞检测的解决策略上做了系统的分类和总结，文中提到目前的碰撞检测都是从几何或代数的方法进行求解，其中几何方法主要利用投影、采样或者二者的结合的技术进行处理。

随着 3D 扫描仪的出现，近年来也出现了一些基于点云的碰撞检测算法。Klein Jan 等人[54] 首次在 EuroGraphics 04 上提出了点云的碰撞检测概念，同样利用了层次结构包围体进行加速检测，并提出了一个给定容差的判断点云模型之间是否相交的算法。文献 [55] 将 AABB 包围体和 R 树结合起来，并从 OAABB（Overlapped AABB）中找出是否含有一个模型的点在容差范围内逼近另外一个点云模型以此判断是否发生相交。随着计算机图形卡的产生，又不断涌现出基于 GPU 的碰撞检测并行算法，如文献 [56,57] 等，利用 GPU 多线程技术提高碰撞检测的效率。

1.3.2 基于包围体树的碰撞检测算法

在第 1.2 节中介绍了在对原始几何模型进行求交判断之前先用其包围体判断是否相交有助于提高求交过程的性能，通过将包围体组织成树形结构即层次结构包围体（BVH）能够将包围体预判阶段从理论上降低到对数时间复杂度（要求树

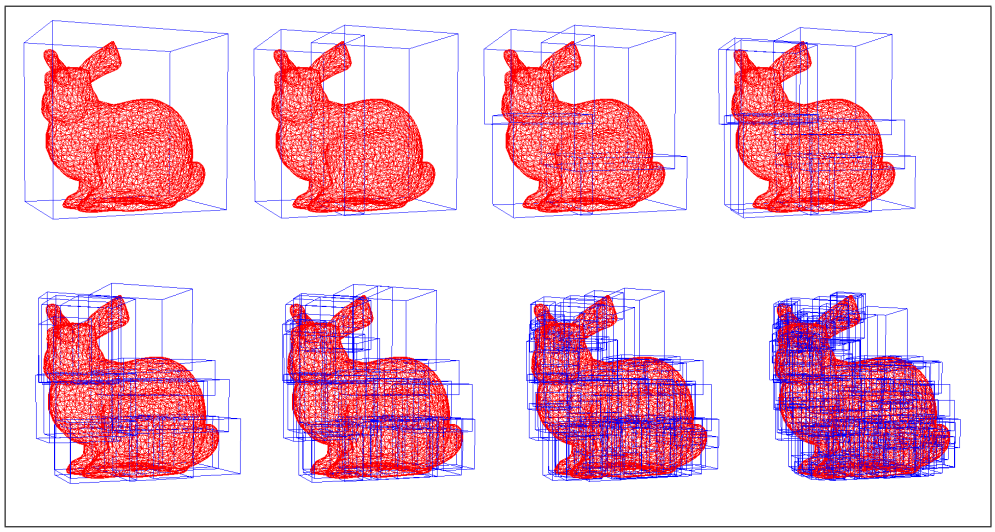


图 1.10　八层 BVH 示例

是平衡的)。当构造原始复杂模型的包围体树时，复杂模型会被拆分成多个部分，每个部分用某种包围体近似，这构成了树型结构的叶子节点，这些节点会按照某种分组策略进行合并成更大的包围体分别构成其父节点，如此递归，最终形成一颗树，树的根节点是一个包含住原始物体的最粗糙的包围体，如图 1.10 所示，为一个 8 层 AABB 包围体的 BVH，也可以看作是 8 层 6-DOP 的 BVH。

在碰撞检测算法中，一般会对模型进行预处理建立起各自的 BVH 树，当两个模型进行相交检测时，自顶向下遍历两棵 BVH 树，若父节点没有相交，则没有必要进行子节点的相交测试，自顶向下的层次遍历算法如算法 2 所示，当检测到任意一个叶子节点包含的部分原始模型发生相交时，即可终止遍历原始模型发生碰撞。

从算法 2 可以看出，基于 BVH 的碰撞检测算法性能的好坏关键在于两个操作，一是分别来自两个 BVH 内部节点包围体之间的相交测试，二是 BVH 最底层叶子节点内的原生几何相交测试，常常利用公式 (1-2) 来衡量单次相交测试的所付出的代价

$$T_{cost} = n_v * C_v + n_p * C_p, \tag{1-2}$$

其中，n_v 和 n_p 分别表示参与包围体节点相交测试的数量和参与原始几何相交测试的数量，C_v 和 C_p 则表示相应的平均测试耗费的代价[11]。当运动场景或连续碰撞的模型之间的碰撞检测时，往往还需要考虑到因物体模型旋转平移等变换产生的包围体的更新所付出的代价，即上述代价函数变成

$$T_{cost} = n_v * C_v + n_p * C_p + n_u * C_u, \tag{1-3}$$

算法 2 自顶向下层次遍历 BVH

输入: 两个 BVH 树的根节点 $node_1$，$node_2$
输出: 模型是否相交

```
1: function TraverseBVHTree(node1, node2)
2:     if node1.bv ∩ node2.bv = ∅ then
3:         return False // 包围体重合测试, 包围体不相交直接返回
4:     else
5:         if node1.children = ∅ then
6:             if node2.children = ∅ then
7:                 // 最底层叶子节点原生几何相交测试
8:                 return CheckIntersection(node1.primitives, node2.primitives)
9:             else
10:                for all child ∈ node2.children do
11:                    TraverseBVHTree(node1, child) // 递归调用
12:                end for
13:            end if
14:        else
15:            for all child ∈ node1.children do
16:                TraverseBVHTree(child, node2) // 递归调用
17:            end for
18:        end if
19:    end if
20: end function
```

其中 n_u 和 C_n 就是模型旋转或者运动后包围体更新的数量和更新的代价。希望提高碰撞检测算法的整体性能，就得想办法减小 T_{cost} 的值。

不同的包围体，上述代价函数各个值不同，对于不同的应用场景，可能需要选择不同的包围体进行加速碰撞检测。文献 [34] 从包围体的构造难度，包围体的紧致性，包围体之间相交测试复杂性及包围体在运动过程中的更新代价几个方面对五种常见的包围体进行了比较，如表 1.1 所示。可以看出，各种包围体有各自的优缺点和适用场景，因此，很多研究人员充分利用各种包围体的优点，将多种包围体结合在一起组成新的组合包围体。文献 [32] 利用 OBB 包围体相对的紧致优点和球形包围体相交测试简单性的优点进行组合，构造 OBB-Sphere 包围体，在进行相交测试时，先利用球形包围体进行测试，如果相交再用 OBB 进行测试。类似的，文献 [34] 同时利用更加紧致的 k-DOP 和 Sphere 包围体构造 k-DOP-Sphere 包围体。

表 1.1 碰撞检测中常用凸包围体比较

凸包围体	构造代价	相交测试简单性	紧致性	更新代价
AABB	1	2	4	3
OBB	4	4	3	2
Sphere	2	1	5	1
k-DOP	3	3	2	4
Convex hull	5	5	1	5

本文算法在基于 AABB 包围体树的算法基础上，通过更加紧致的 *k*-CBP 来过滤更多不相交的模型，通过减少公式 (1-2) 及公式 (1-3) 中 n_p 的值来达到提高碰撞检测算法的效率。

1.4　本文主要内容

本文着重讨论如何快速地生成紧致性可控的凸包围体以及将此包围体应用于静止和运动场景的碰撞检测。

第一章首先介绍了当前各种凸包围体的特征及应用，然后再介绍了碰撞检测问题的背景和本文研究的基于凸包围体的碰撞检测算法。

第二章提出了紧致性可控的凸包围多面体（*k*-Convex Bounding Polyhedron，简称 *k*-CBP） 的生成算法，紧致性可控主要通过凸包围体的平面数量 *k* 来调节。该算法法首先利用近似内凸包和 *k*-means 聚类算法生成构成凸包围多面体 *k* 个截面的法向，然后根据输入点集沿各法向搜索切点构成截面，最后由这些截面通过对偶映射的方式求交构成凸包围多面体。在搜索截面过程中，提出了两种并行策略以加速搜索过程。实验结果表明，与同类算法相比，该方法能够更快地构造给定点集更紧致的凸包围多面体。

第三章提出了基于本文提出的 *k*-CBP 包围体的碰撞检测算法，在包围体之间的相交测试时分别用 AABB 树的方式和基于 GJK 算法的两种方式进行。实验结果表明本文提出的凸包围体能够有效加速碰撞检测算法。

第四章是对本文的研究工作进行总结，以及未来可以改进的方向。

第 2 章　凸包围体生成算法

从第 1.2 节可以看出包围体的紧致程度直接影响相应算法的效率，对于不规则形体，常见的包围体往往不够紧致，凸包紧致但往往包含过多的面片数而增加算法的复杂性。

本文提出一种构造凸包围多面体的方法，该方法首先利用近似内凸包和 k-means 聚类算法生成构成凸包围多面体 k 个截面的法向。然后根据输入点集沿各法向搜索切点构成截面，最后由这些截面通过对偶映射的方式求交构成凸包围多面体。搜索截面过程中，各个法向的搜索过程相互独立互不影响，因此可以方便地利用 GPU 进行加速。本文提出的方法的主要优势是：(1) 对给定点集可构造紧致的包围体；(2) 利用 GPU 加速，能够快速构造包围体；(3) 通过参数 k 调节凸包围多面体的简单性和紧致性，可适用于不同的应用场景。

由 k 个截面构成的凸包围多面体称为凸包围 k 面体 (k-Convex Bounding Polyhedron，简称 k-CBP), 可通过 k 个半空间定义：

$$\begin{cases} k\text{-CBP} = \bigcap_{i=1}^{k} \boldsymbol{H_i} \\ \boldsymbol{H_i} = \left\{ \boldsymbol{p} \in \mathbb{R}^3 \middle| \boldsymbol{n_i} \cdot \boldsymbol{p} \le w_i, w_i \in \mathbb{R} \right\}, \end{cases} \tag{2-1}$$

其中，$\boldsymbol{n_i}$ 是半空间 $\boldsymbol{H_i}$ 的法向，方向指向包围体外部，w_i 是输入点集中沿 $\boldsymbol{n_i}$ 方向投影的最大值。如图 2.1 所示是 Bunny 模型的凸包围 34 面体（34-CBP）。

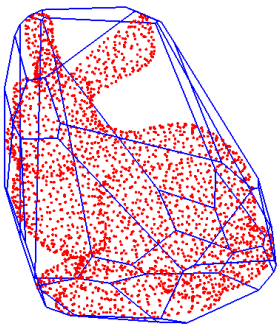


图 2.1　Bunny 模型的 34-CBP

本文算法的主要流程如图 2.2 所示，首先利用近似内凸包和 k-means 聚类算法生成构成凸包围多面体 k 个截面的法向，然后根据输入点集在 GPU 中沿各法向搜索切点构成截面，最后由这些截面通过对偶映射的方式求交构成凸包围多面体。搜索截面需要多次扫描输入点集，在 CPU 中计算较耗时，因此用 GPU 加速使得算法整体性能得以提升，其他步骤在 CPU 计算即可。

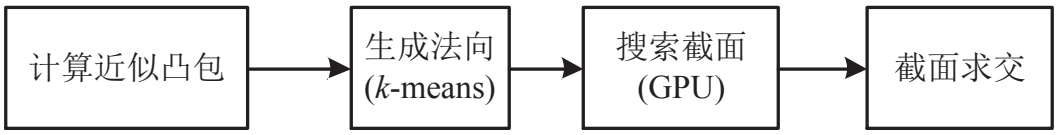


图 2.2　构造 k-CBP 算法流程图

图 2.3 从左至右分别显示了 Bunny 模型的 26-CBP、精确凸包和近似内凸包。近似凸包是精确凸包的一种近似，与精确凸包外观相似但其构造复杂度降低，不少研究者利用该性质解决计算机图形学中很多问题[15]。

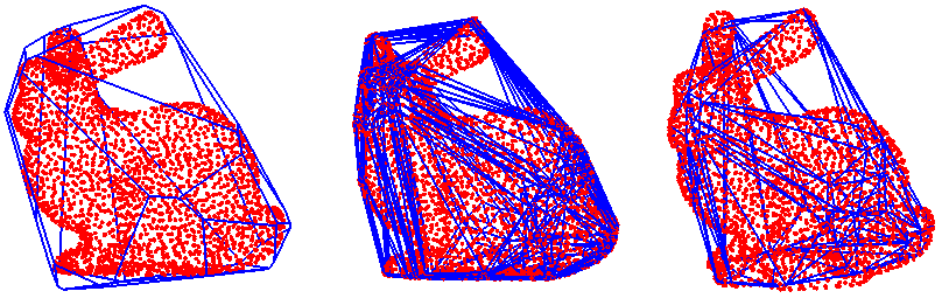


图 2.3　Bunny 模型的 26-CBP、凸包和近似内凸包

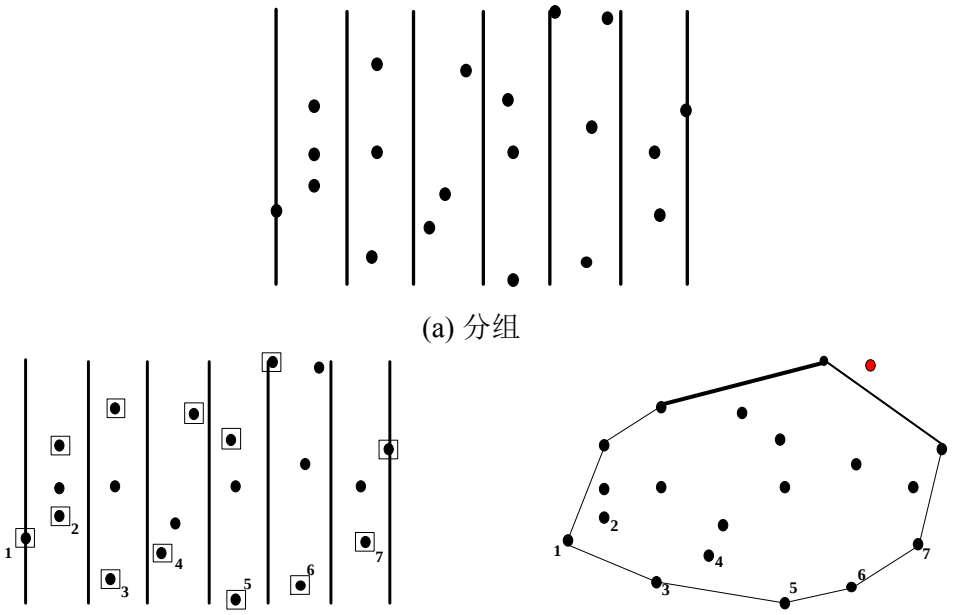
本文就利用了近似内凸包与精确凸包的相似性，从众多近似凸包面片的法向中通过 k-means 聚类算法生成构成凸包围多面体的 k 个法向。本章后续部分将按步骤详细介绍算法的实现。

2.1　截面法向的生成

截面法向的选定与多面体的紧致程度密切相关。k-DOP[11] 预先定义 $k/2$ 对法向，且 k 值局限于 6、14、18 或 26 等少数几个，对于不同的模型其方向始终一致，导致其生成的多面体不能自适应模型，对于不规则形体的模型不够紧致。与 k-DOP 相比，本文算法 k 值取值更灵活，因此可以根据不同需求不同应用场景更加灵活地选择需要的凸包围体。根据经验，为了使凸包围多面体尽可能逼近凸包，在多面体数量一定的情况下需尽量保留凸包中面积较大的面。本文生成凸包围多面体的法向的流程是先通过一种线性算法[16] 构造近似内凸包，然后从近似内凸包里选取面积最大的 a 个面的法向，最后用聚类算法从剩下的面中生成 $k-a$ 个法向。下面将介绍截面法向生成的主要步骤。

2.1.1　近似内凸包生成聚类样本法向集

以平面点集的近似内凸包为例说明初始法向集的生成。



(a) 分组

(b) 选极值点　　(c) 连线

图 2.4　二维近似内凸包的构造[16]

如图 2.4 所示，整个算法分为如下 3 个步骤：

(1) **分组：** 首先根据 x 轴将输入点集均分 ξ 组，如图 2.4(a)所示，ξ 值可以根据输入点集数量以及希望得到的近似凸包的近似程度进行选取，图示中分成了 6 组。

(2) **选极值点：** 然后在每组中找出沿 y 轴的最大最小坐标值的点，得到每组的极值点，如图 2.4(b) 所示，在边界上的点可直接当作极值点。

(3) **连线：** 最后按条件连接每组的最值构成近似内凸包，如图 2.4(b)所示，以下半圈为例，在选定的极值点中先选定最左边的点 1，下一个候选点 2，连接 13 发现候选点 2 在 13 的左边，丢弃候选点 2 直接连接 13，同理连接 35，当查看候选点 6 时，发现点 6 在 57 连线的右边，因此候选点 6 保留，依次类推可得整个下半圈为凸包，上半圈同理可得。最后合并上下半圈得到如图 2.4(c)所示的结果。

此算法复杂度为 $O(n+\xi)$。得到近似凸包后，因较长的边（图 2.4(c) 中的粗边）对最后凸包围多边形影响较大，因此可以保留较长的 a 条边，其他 $n-a$（n 为近似内凸包边数）进行聚类得到 $k-a$ 类，最后得到边对应的垂直向外的方向作为法向。在三维空间里，可按照 x,y 轴最多划分成 $(\xi+2)\times(\xi+2)$ 个正方体网格，每个正方体网格取 z 轴的最大最小坐标值，因此所有网格含最多含有 $2(\xi+2)^2$ 个极值点，其凸包可以在 $O(\xi^2\log\xi^2)=O(\xi^2\log\xi)$ 内求得，因此整个算法时间复杂度为 $O(n+\xi^2\log\xi)$，关于此算法更详细的细节可参考文献 [16]。实验过程中，为了更快地构造近似内凸包，ξ 值通常取得较小（例如取 $\xi=10$）。

假设点 A,B,C 为近似凸包的某个平面 P_i 上逆时针方向上的 3 个顶点，则该平

面 P_i 的法向 $\boldsymbol{n_i} = \overrightarrow{AB} \times \overrightarrow{AC}$，这些法向就是参与聚类算法的所有样本法向集。

2.1.2　聚类初始点的选择

聚类算法是数据挖掘领域里的研究热点之一，k-means 是最流行和最简单的基于划分的聚类算法[58]。其中，k-means 算法最初的步骤就是初始聚类中心的选择。本文算法将采取如下的策略生成：给定一个单位球，将其按照等面积划分成 k 份即 k-means 中的参数 k，连接球心到每份中心点构成的方向作为法向，其效果如图 2.5 所示（$k = 26$）。

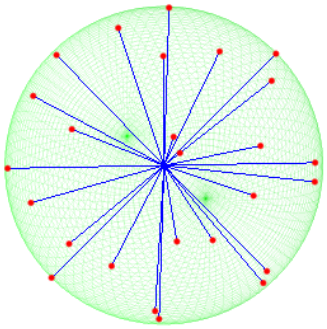


图 2.5　初始聚类法向的生成

初始点生成算法具体如算法 3 所示① 。另外，亦可使用如文献 [59] 中的数学统计方法在球面上生成若干个点使其满足均匀分布。

算法 3 初始法向的生成

输入： 初始法向数量 k
输出： k 个法向 *normals*

```
1:  function GenerateInitNormal(k)
2:      area ← 4π/k
3:      m_v ← ROUND(π/√area)
4:      d_v ← π/m_v
5:      d_phi ← area/d_v
6:      for m = 0 → m_v − 1 do
7:          v ← πm/m_v
8:          m_phi ← ROUND(2π sin v/d_phi)
9:          for n = 0 → m_phi − 1 do
10:             φ ← 2πn/m_phi
11:             normals ← normals ∪ VEC3(sin v cos φ, sin v sin φ, cos v)
12:         end for
13:     end for
14:     return normals
15: end function
```

① http://www.cmu.edu/biolphys/deserno/pdf/sphere_equi.pdf

2.1.3　聚类确定法向

聚类算法将相似程度高的变量聚集到一类，距离度量是 *k*-means 中的关键，采用不同的距离计算函数可能得到不同的聚类结果，依据数据的不同性质可选用不同的距离度量方法，常用的有欧式距离，曼哈顿距离和卡方距离等等。本文采用余弦距离度量，将方向相近的点归聚到一类。*k*-means 本质上是一个迭代贪心算法，每一次迭代需要重新计算每一类的中心点，直到两次迭代后中心点相差在给定的容差范围内为止。完整的算法如算法 4 所示。

算法 4 *k*-means 确定法向

输入： 初始中心点 *init*, 聚类数量 *k*, 聚类样本法向集 *points*
输出： *k* 个聚类后的法向 *result*

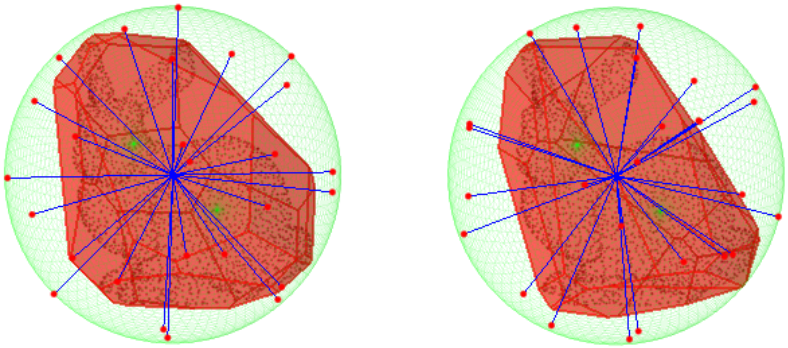
```
1:  function KMEANSCLUSTER(init, k, points)
2:      for all p ∈ points do
3:          for all c ∈ init do
4:              φ ← c · p  // 计算余弦距离, 并记录每个聚类变量属于哪一类
5:          end for
6:          for all c ∈ init do
7:              result ← UPDATE(c)  // 按照公式 (2-2) 更新中心点
8:              if CHECKSTATE(result, init, iter) then
9:                  return result
10:                 // 检测迭代条件是否满足，中心点容差范围内不在变化或迭代次数超过最
    大迭代次数
11:             else
12:                 init ← result
13:                 iter ← iter + 1
14:             end if
15:         end for
16:     end for
17: end function
```

算法 4 中第 7 行 $UPDATE$ 方法作用是聚类迭代更新每类的中心点，本文更新中心点时将每个法向对应的面片的面积作为权重即

$$\boldsymbol{c_i} = \frac{\sum_{i=1}^{i=n} \omega_i \cdot \boldsymbol{n_i}}{\sum_{i=1}^{i=n} \omega_i}, \tag{2-2}$$

其中，$\boldsymbol{c_i}$ 为第 i 类的中心点，ω_i 为法向 $\boldsymbol{n_i}$ 所在面片对应的面积，这样使得生成的法向尽量靠近原始近似凸包面积较大的面片的方向。

通过用聚类方法得到的法向生成的凸包围多面体比直接用初始法向生成的包围体更加紧致。如图 2.6 所示，其中图 2.6(a) 为按照初始方向为 Bunny 模型生成的凸包围多面体，其中 $k = 26$。聚类后其法向及其生成的凸包围多面体如图 2.6(b) 所示，后者的紧致程度比前者高 14.98%。



(a) 初始法向生成 k-CBP　　(b) 聚类法向生成 k-CBP

图 2.6　初始法向和聚类确定法向生成 k-CBP 对比 ($k = 26$)

2.2　搜索截面

在确定构成 k-CBP 的法向后，只需要搜索空间上的一个点即可确定 k-CBP 的截面，搜索截面等效于寻找沿着法向上的最大投影值，即对每个法向 $\boldsymbol{n_i}$，从输入模型的所有点中寻找最大投影值的点作为切点进而确定法向 $\boldsymbol{n_i}$ 对应的截面。具体算法如算法 5 所示，该过程的时间复杂度为 $O(k \cdot n)$，其中 k 为法向数量，n 为模型点的数量。

算法 5 搜索截面串行算法

输入: 截面法向 *normals*, 模型点集 *points*
输出: 多面体截面 *planes*

```
1:  function SearchCuttingPlanes(normals, points)
2:      for all n ∈ normals do
3:          for all p ∈ points do
4:              proj ← p · n  // 计算投影值
5:              max_proj ← MAX(proj, max_proj)  // 更新最大投影值
6:          end for
7:      end for
8:      for all n ∈ normals do
9:          max_point ← n · max_proj  // 切点
10:         planes ← planes ∪ PLANE(n, max_point)  // 构造平面
11:     end for
12:     return planes
13: end function
```

从算法 5 可以看出该搜索过程中各法向的计算相互独立，互不影响，因此可方便地借助 GPU 并行加速。本文将采用两种平台进行 GPU 加速，一种平台是基于 OpenGL 着色语言，另外一种是 CUDA 平台，下面将分别介绍这两种平台的实现。

2.2.1　基于着色器的并行算法

OpenGL 提供了可编程管线，它的轻便性、跨平台性和被广泛硬件厂家所支持使得 OpenGL 着色语言（OpenGL Shading Language，简称 GLSL）被广泛应用，在基于 GPU 的通用计算（General Purpose Graphic Process Unit，简称 GPGPU）中也发挥着重要作用。下面将分别介绍本文提出的基于深度缓冲 (Z Buffer) 和乒乓技术的算法。

2.2.1.1　基于深度缓冲的算法

在 OpenGL 中，当渲染 3D 模型时，深度测试是在片段着色器工作后的一个重要测试操作，当两个片段在相同的位置时，深度测试的目的是决定哪个片段将要保留下来，默认情况下，最前面即有较小（或较大，可设置）的 z 坐标值的片段将通过测试被保留下来。每个片段的深度信息保存在深度缓冲中，当新的片段通过测试后将更新缓冲中的值。本文充分利用了 OpenGL 的这种机制，算法每走一遍渲染流程将找出一个方向上的切点，在顶点着色器中，将所有点的 x, y 坐标值设为一样，并将该点在法向上的投影作为 z 值即深度值，所有点经过深度测试后将只有 z 值最大的点被保留下来，该点就是沿着这个法向的切点。该过程的算法流程图如 2.7 所示。

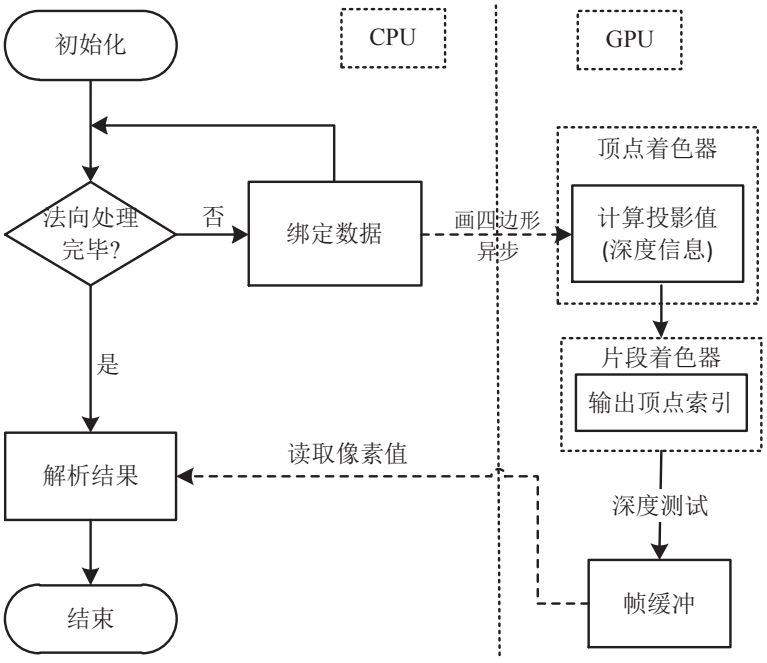


图 2.7　基于 Z Buffer 算法流程图

在实际实现过程中需要注意默认情况下 OpenGL 处理 x 和 y 坐标值的范围是 $[-1, 1]$，而深度值需要映射到 $[0, 1]$。我们利用一张 $k \times 1$ 大小的纹理保存结果，将

第 i 个法向的切点保存在纹理的 $(i,0)$ 坐标处，这样做的好处是每绘制一遍不用清除深度缓冲，最后可以一次性读取这个 $k\times 1$ 大小的纹理得到结果。如代码 2.1 的着色器代码所示，OpenGL 中利用齐次坐标处理渲染流程，因此我们利用前三个分量保存法向的 x,y,z 坐标值，第 4 个分量 w 保存法向的索引，在片段着色器中，只需要简单地输出切点的在输入点数组的索引即可。通过绘制 k 遍，我们从大小为 $k\times 1$ 的纹理中读出点索引以此就能确定 k 切平面，每绘制一遍得到 1 个切平面，当 k 值较大时，这个过程仍然比较耗时，从实验结果也能看出，这种算法适用于 k 值相对较小的情况。

代码 2.1　基于 Z Buffer 算法着色器代码

```
//顶点着色器
layout(location = 0) in vec3 inPosition;
flat out vec4 result;
uniform vec4 normal; //xyz: 坐标值, w: 法向索引
uniform float length;
uniform int normal_count;

void main()
{
    float distance = dot(inPosition, normal.xyz);
    float depth = distance / length; //放缩到 [-1, 1]
    depth = depth * 0.5 + 0.5; //映射到 [0,1]
    vec2 coord = vec2(-1 + normal.w * 2.0 / normal_count, 0.5);
    gl_Position = vec4(coord, depth, 1.0);
    result = vec4(gl_VertexID, 0, 0, 0);
}

//片段着色器
flat in vec4 result;
out vec4 output;

void main()
{
    output = vec4(result.x, 0, 0, 0);
}
```

2.2.1.2　基于乒乓技术的算法

渲染到纹理（Render To Texture，简称 RTT）是一个非常重要的图形可视化技术，它能够帮助快速地渲染很多漂亮的结果，同时也是 GPGPU 重要组成部分。在基于 OpenGL 的 GPGPU 中，通常要将参与计算的数据通过 CPU 传送至 GPU 的特定大小的纹理中，通过绘制一个与保存数据有着同样大小的四边形来调用着色器中的算法[60]，将结果输出到纹理中。该过程中，绘制同样大小的四边形是为了避免纹理插值对输入数据的影响。RTT 的算法一般是在片段着色器（Fragment

Shader）中完成，有时通过一遍的绘制得不到最终结果还需要把前一次运算结果传递给下一次运算用来作为后继运算的输入，此时可以利用乒乓技术来解决。

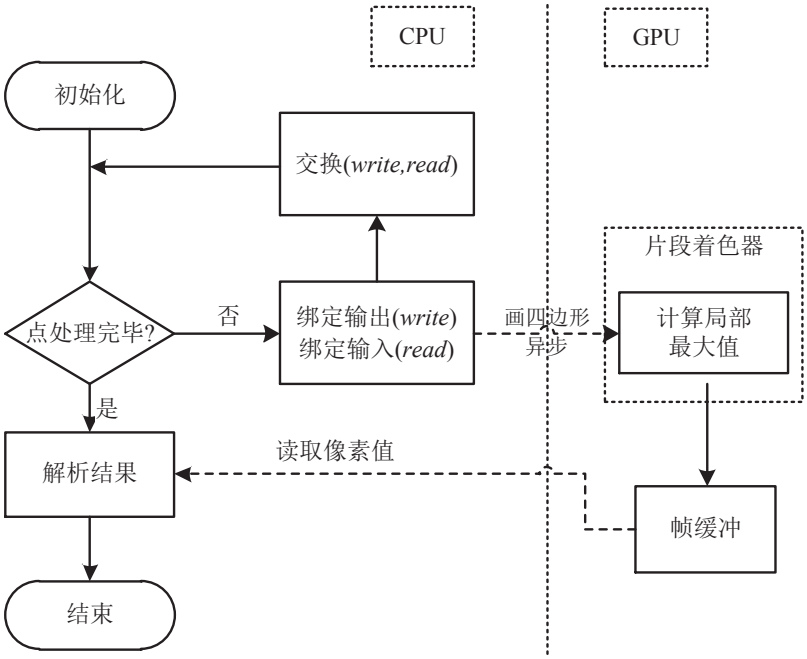


图 2.8　基于乒乓技术算法流程图

图 2.8 为基于乒乓技术的算法的流程图，算法将所有法向保存在纹理 *read* 中并利用纹理 *write* 缓存中间结果，第 m 次渲染进行计算时，纹理 *read* 作为输入，将渲染即计算的结果缓存在纹理 *write* 中，在第 $m+1$ 次渲染时将读取缓存在纹理 *write* 中的数据作为输入，而此时纹理 *read* 又作为输出，如此进行交替，每步交换只更改沿着每个法向的局部最大值。如代码 2.2 中的片段着色器代码所示，用一个像素值保存法向的 *xyz* 坐标值和沿着该法向投影得到最大值，另外该像素的第四个分量保存该最大值点在输入点数组中的索引。算法将输入点分成 x 份，每次只处理一份，在当前渲染过程中，找出在该批点集中沿着所有法向投影值最大的点，当处理下批点集即下一次渲染时，将与上次局部最大投影值进行比较，若有新的投影最大值就更新，如此反复 x 次，当所有点都被处理完毕后得到沿着所有法向投影最大值的点即切点。最后再通过 CPU 一次性从帧缓冲中读取所有法向的切点。

与 Z Buffer 算法相比，基于乒乓技术的算法在进行一次绘制操作能够得到沿着 k 个法向的局部（所有处理过的点集）最大值。为了充分利用 GPU 的并行计算能力，将每次处理点的数量设置为 GPU 硬件所支持的 OpenGL 中统一块（uniform block）所能容纳数据的大小，即需要反复绘制的次数为 $x = n/b$，其中 n 为点集大小，b 为统一块能容纳点的数量，值与显卡具体型号有关。

代码 2.2　基于乒乓技术算法着色器代码

```
//片段着色器
uniform sampler2DRect  imageSampler;
uniform sampler2DRect  indexSampler;
uniform int curPointCount;
uniform int curPointStart;

layout(packed) uniform Points
{
    //$MAX_POINT_SIZE$编译前会被替换
   vec4 curPoints[$MAX_POINT_SIZE$];
};

void main()
{
    vec4 normal_t = texture2DRect(imageSampler, gl_TexCoord[0].
       xy);
    float index_float = texture2DRect(indexSampler, gl_TexCoord
       [0].xy).w;
    int index = int(index_float);
    vec3 normal = normal_t.xyz;
    float max = normal_t.w;

    for(int i = 0; i < curPointCount; i++)
    {
        float t = dot(curPoints[i].bgr, normal); //BRGA格式
        if(t > max)
        {
            max = t;
            index = curPointStart + i;
        }
    }
   gl_FragData[0] = vec4(normal, max);
   gl_FragData[1] = vec4(0, 0, 0, index);//index为全局索引
}
```

2.2.2　基于 CUDA 的并行算法

CUDA 是显卡公司 NVIDIA 推出的通用的并行计算架构平台，能够利用 GPU 解决并行计算问题，并提供了多种语言的编程接口。CUDA 程序可以由一系列的主机程序组成，主机程序能够并行地在 GPU 设备上运行，GPU 运算单元将并行的线程分解成线程块，每个线程块又由若干线程组成，GPU 的流水处理器以线程块为单位进行调度。将问题划分成子问题提交给 GPU 让多线程同时处理这些子问题，这样可以提升算法的效率[43]。

如图 2.9 所示，最大投影值的计算可采用如下规约方式：将输入点交给数量为 t 的线程计算点积得到投影值，线程 i 和 $i+t/2$ 比较选取较大者，经 $\log_2 t$ 次比

较可得最大值。

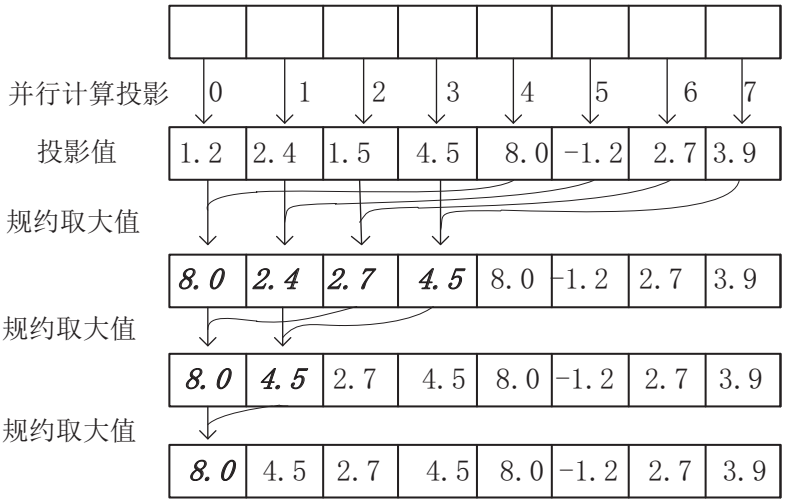


图 2.9　并行规约求最大投影值

图 2.9 所示的示例中，假设有 8 个点与某法向点积得到的投影值为 $(1.2, 2.4, 1.5, 4.5, 8.0, -1.2, 2.7, 3.9)$，有 4 个线程同时比较第 i 个投影值与 $i+4$ 投影值得到新的较大投影值 $(8.0, 2.4, 2.7, 4.5)$，并将新的较大的投影值放入线程 i 中，再以同样的方式规约得到较大的两个投影值 $(8.0, 4.5)$，最后再通过一次比较得到最大的投影值 8.0 。文献 [61] 介绍了更多更详细规约优化技术。

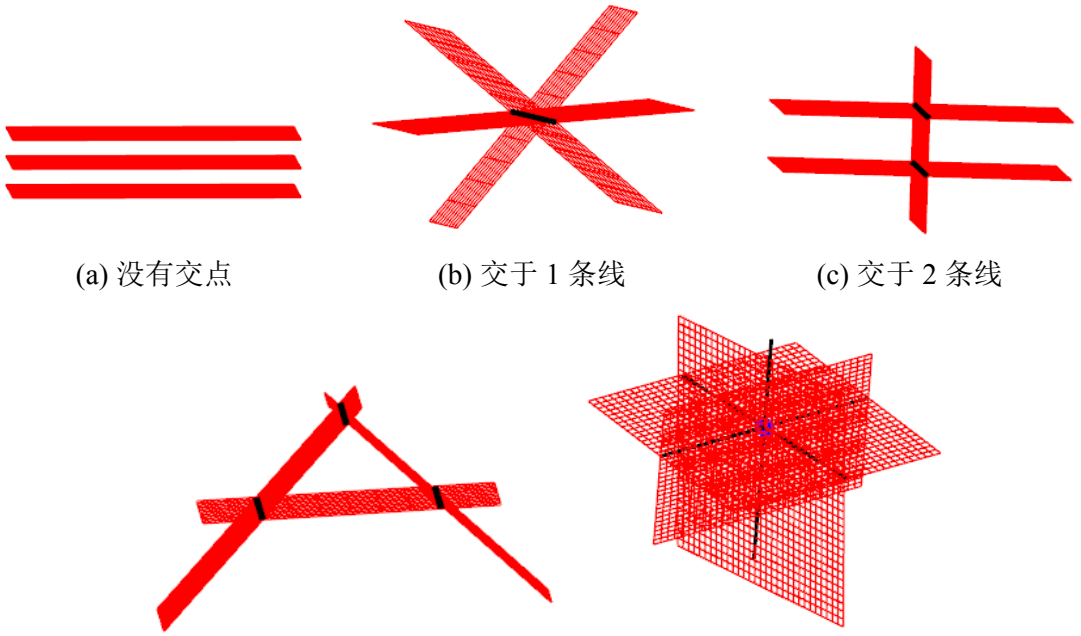
2.3　截面求交算法

确定 k-CBP 各个截面后，直接求得所有平面的交点并排除在平面外部的交点即可得到 k-CBP 的顶点。该问题即转化为：给定 k 个平面及其法向，平面及法向相当于一个半空间 $\boldsymbol{H}$，即已知集合 $\{\boldsymbol{H_1}, \boldsymbol{H_2}, \cdots, \boldsymbol{H_k}\}$，求 $\boldsymbol{H_1} \cap \boldsymbol{H_2} \cap \cdots \cap \boldsymbol{H_k}$。该问题是一个线性规划（Linear Programming）问题，文献 [1] 详细介绍了在二维线性规划下的相关算法，本文将分别介绍一种直观的枚举算法和利用对偶映射的方法求得 k-CBP 的交点。

2.3.1　枚举法

空间中的平面位置情况如图 2.10 所示，可分为如下几种情况：(1) **无交点**，若 3 个平面互相平行则没有交点，如图 2.10(a) 所示；(2) **1 条交线**，如图 2.10(b) 所示，3 个平面相交于 1 条交线；(3) **2 条交线**，当有两个平面互相平行，另一个平面与其相交时有 2 条交线，如图 2.10(c) ；(4) **3 条交线**，如图 2.10(d) 所示，3 个平面交于 3 条交线；(5) **1 个交点**，最后一种情况是三个平面相交于 1 点的情况，如图 2.10(e) 所示。

一个凸多面体的每一个顶点都可看作是至少 3 个平面的交点。现在只需要枚举出所有 3 个平面相交于 1 点的所有情况即可得到 k-CBP 的顶点，值得注意的是，并非所有交点都是多面体的顶点，交点在某个半空间的正方向上即在半空间相交区域的外部须排除。



(a) 没有交点　(b) 交于 1 条线　(c) 交于 2 条线

(d) 交于 3 条线　(e) 交于 1 个点

图 2.10　空间中 3 个平面的相交情况

假设满足如图 2.10(e) 所示的三个平面的法向分别是 $\boldsymbol{n_1}, \boldsymbol{n_2}, \boldsymbol{n_3}$，平面上一点分别为 $\boldsymbol{p_1}, \boldsymbol{p_2}, \boldsymbol{p_3}$，则三个平面的交点须满足如下方程：

$$\begin{cases} \boldsymbol{n_1} \cdot \boldsymbol{x} = \boldsymbol{n_1} \cdot \boldsymbol{p_1} \\ \boldsymbol{n_2} \cdot \boldsymbol{x} = \boldsymbol{n_2} \cdot \boldsymbol{p_2} \\ \boldsymbol{n_3} \cdot \boldsymbol{x} = \boldsymbol{n_3} \cdot \boldsymbol{p_3} \end{cases} \tag{2-3}$$

令 $(\boldsymbol{n_1} \cdot \boldsymbol{p_1}, \boldsymbol{n_2} \cdot \boldsymbol{p_2}, \boldsymbol{n_3} \cdot \boldsymbol{p_3}) = (y_1, y_2, y_3) = \boldsymbol{y}$，则求解式 (2-3) 相当于解线性方程组 $\boldsymbol{Ax} = \boldsymbol{y}$ 即 $(\boldsymbol{n_1}, \boldsymbol{n_2}, \boldsymbol{n_3})^T \cdot \boldsymbol{x} = \boldsymbol{y}$，令 $\boldsymbol{n_i} = (a_{i1}, a_{i2}, a_{i3}), i = \{1, 2, 3\}$，则有：

$$\begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{pmatrix} \cdot \boldsymbol{x} = \begin{pmatrix} y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

根据克莱姆法则（Crammer's Rule），上述方程有唯一解，则 $|\boldsymbol{A}| \neq 0$，且解 $x_i = \frac{|\boldsymbol{A}_i|}{|\boldsymbol{A}|}, i = \{1, 2, 3\}$，其中 $|\boldsymbol{A}_i|$ 是矩阵 $\boldsymbol{A}$ 中将第 i 列替换成 $\boldsymbol{y}^T$ 之后的行列式，在

计算 $\boldsymbol{A}_i$ 和 $|\boldsymbol{A}|$ 时为了避免一些重复计算，将其展开有：

$$\begin{cases} |\boldsymbol{A}| &= a_{11}\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12}\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13}\begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} \\ |\boldsymbol{A}_1| &= y_1\begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} - a_{12}\begin{vmatrix} y_2 & a_{23} \\ y_3 & a_{33} \end{vmatrix} + a_{13}\begin{vmatrix} y_2 & a_{22} \\ y_3 & a_{32} \end{vmatrix} \\ |\boldsymbol{A}_2| &= a_{11}\begin{vmatrix} y_2 & a_{23} \\ y_3 & a_{33} \end{vmatrix} - y_1\begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} + a_{13}\begin{vmatrix} a_{21} & y_2 \\ a_{31} & y_3 \end{vmatrix} \\ |\boldsymbol{A}_3| &= a_{11}\begin{vmatrix} a_{22} & y_2 \\ a_{32} & y_3 \end{vmatrix} - a_{12}\begin{vmatrix} a_{21} & y_2 \\ a_{31} & y_3 \end{vmatrix} + y_1\begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} \end{cases}$$

令

$$\begin{cases} b_1 &= \begin{vmatrix} a_{22} & a_{23} \\ a_{32} & a_{33} \end{vmatrix} = a_{22}a_{33} - a_{32}a_{23} \\ b_2 &= \begin{vmatrix} a_{21} & a_{23} \\ a_{31} & a_{33} \end{vmatrix} = a_{21}a_{33} - a_{31}a_{23} \\ b_3 &= \begin{vmatrix} a_{21} & a_{22} \\ a_{31} & a_{32} \end{vmatrix} = a_{21}a_{32} - a_{31}a_{22} \\ b_4 &= \begin{vmatrix} y_2 & a_{23} \\ y_3 & a_{33} \end{vmatrix} = y_2a_{33} - y_3a_{23} \\ b_5 &= \begin{vmatrix} y_2 & a_{22} \\ y_3 & a_{32} \end{vmatrix} = y_2a_{32} - y_3a_{22} \\ b_6 &= \begin{vmatrix} a_{21} & y_2 \\ a_{31} & y_3 \end{vmatrix} = a_{21}y_3 - a_{31}y_2 \end{cases}$$

则有：

$$\begin{cases} x_1 = \frac{|\boldsymbol{A_1}|}{|\boldsymbol{A}|} = \frac{y_1b_1 - a_{12}b_4 + a_{13}b_5}{a_{11}b_1 - a_{12}b_2 + a_{13}b_3} \\ x_2 = \frac{|\boldsymbol{A_2}|}{|\boldsymbol{A}|} = \frac{a_{11}b_4 - y_1b_2 + a_{13}b_6}{a_{11}b_1 - a_{12}b_2 + a_{13}b_3} \\ x_3 = \frac{|\boldsymbol{A_3}|}{|\boldsymbol{A}|} = \frac{-a_{11}b_5 - a_{12}b_6 + y_1b_3}{a_{11}b_1 - a_{12}b_2 + a_{13}b_3} \end{cases} \tag{2-4}$$

其中，$|\boldsymbol{A}| \neq 0$。

完整的算法如算法 6 所示，枚举遍历所有平面的组合，搜索出三个平面交于一点的情况，排除在半空间外部的交点即可得到 k-CBP 的顶点，算法复杂度为 $O(k^3)$，此算法思想简单易于理解和实现，但当平面数量较多（即 k 值较大）时，

算法 6 枚举算法

输入: 平面 *planes*
输出: 凸包围多面体 *k*-CBP

```
1:  function ConstructKCBP(planes)
2:      for all p1 ∈ planes do
3:          Intersection ← ∅
4:          for all p2 ∈ planes do
5:              for all p3 ∈ planes do
6:                  if p1 ∦ p2 ∦ p3 then
7:                      if |A| ≠ 0 then
8:                          P = vec3(x1, x2, x3)  // 按照公式 (2-4) 计算得到交点坐标值
9:                          if validate(P) then
10:                             // 验证交点 P 是否都在半空间负方向上
11:                             Intersection ← Intersection ∪ P
12:                         end if
13:                     end if
14:                 end if
15:             end for
16:         end for
17:         if Intersection.size ≥ 3 then
18:             k-CBP ← k-CBP ∪ polygon(p1, Intersection)  // 满足条件，加入到结果集
19:         end if
20:     end for
21:     return k-CBP
22: end function
```

会耗费很多时间，与枚举法相比基于对偶映射的算法实现较复杂但其时间复杂度为 $O(k\log k)$，效率更高。

2.3.2　对偶映射算法

正如 *k*-CBP 的定义所知，*k*-CBP 可看作是多个半空间的交集，一个半空间可由法向 $\boldsymbol{n}(a,b,c)$ 及离原点距离 d 确定，转换为线性不等式即为

$$a_i x + b_i y + c_i z \le d_i, \quad i = 1, 2, \cdots, k, \tag{2-5}$$

其中 a_i, b_i, c_i, d_i 是不同时为 0 的实数。

求解 *k*-CBP 可将问题转换为求 k 个线性不等式的解。利用对偶映射的方式可在 $O(k\log k)$ 的方式解决，具体方法分为三个步骤[62]：

(1) 首先，将上述线性不等式对偶映射成一个欧式空间上的三维点，$a_i x + b_i y + c_i z = d_i \rightarrow \boldsymbol{p}(-a_i/d_i, -b_i/d_i, -c_i/d_i), d_i \neq 0$，其中 (a_i, b_i, c_i) 为第 2.2 节中得到的截面法向 $\boldsymbol{n}$，$d_i = \boldsymbol{n} \cdot max\_point$ 为截面法向与截面上的投影点 *max_point* 的点积，此步骤的算法复杂度为 $O(k)$；

(2) 其次，对对偶映射得到的欧式空间 k 个点求凸包，此步骤的算法复杂度为 $O(k\log k)$，可用第 1.2.5 节中的分治算法；

(3) 得到凸包后，再利用相同的对偶变换将凸包平面方程映射回欧式空间三维点，这些点即为原始半空间的交点，此步骤的算法复杂度仍为 $O(k)$。

综上，该方法总体复杂度为 $O(k\log k)$ 。在利用这种对偶变化时需要注意约束条件即上述不等式中的 $d_i > 0$，此时原点 $O(0,0,0)$ 始终满足不等式 (2-5)，体现在算法输入上即原始模型需要包含原点。当原始模型不包含原点时，可以先对模型做一定平移或者也可用另外的对偶变换进行求解，文献 [63] 对此问题进行了详述的阐述。

2.4　实验结果及分析

本文将针对不同点集规模的模型进行测试①，从两个角度对生成的 k-CBP 进行实验对比，一方面对生成 k-CBP 的速度进行效率上的对比，主要对比了串行算法、基于着色器和 CUDA 环境的并行算法和文献 [42] 的并行算法，其详细结果如 2.4.1 节所示；另一个方面对生成 k-CBP 的质量即紧致性上进行了对比实验，主要围绕着凸包、文献 [14] 中实现的 k-DOP 进行对比，详细结果见第 2.4.2 节。

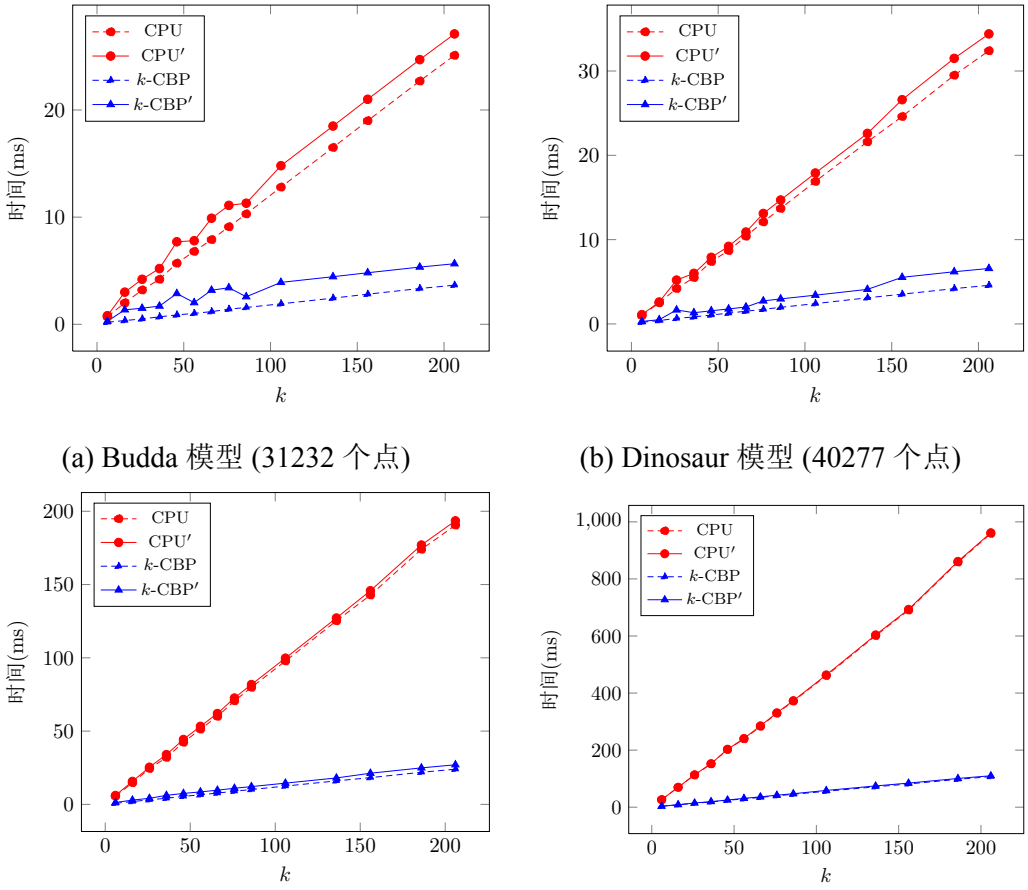
2.4.1　凸包围多面体生成效率

如图 2.11 所示为本文算法（CUDA）与传统的 CPU 算法对比结果，图中横纵坐标分别代表多面体面数和运行时间，其中虚线代表搜索截面的过程，实线为构造凸包围多面体总体耗时。当模型点数量较大时，搜索截面的过程占据了算法绝大多数时间，且随着凸包围多面体的面数 k 值增加而线性增长，这与搜索截面时间复杂度 $O(k\cdot n)$ 一致，截面求交过程利用第 2.3.2 节中的对偶映射法，其时间复杂度为 $O(k\log k)$，当点数量极大时，如图 2.11(d) 所示，实线虚线几乎重合即求交等步骤耗时相比整体算法而言几乎可忽略。当输入模型的点数量越大，本文算法的优势越明显。如 Budda 模型点数量 31k 左右，加速比约为 3 ~ 6 倍，而含有 224k 个点的 Alice 模型和 1010k 个点的 Bugatti 模型，能够加速 7~9 倍。

图 2.12 为着色器算法根据不同模型构造不同面数的包围体所耗费的时间对比，横坐标表示多面体面数 k，纵坐标为运行时间，曲线 cpu、zbuffer 和 rtt_pp 分别表示基于 CPU、深度缓冲（Z Buffer）和基于乒乓技术的算法的运行时间。

从实验结果可以看出，当模型规模不大时，如图 2.12(a) 所示的含有 8 千多个点的 Apple 模型，Z Buffer 算法和传统的 CPU 算法差别不是很大，因此在实际应用中当模型规模较小时可直接用 CPU 计算即可。随着输入模型所含有点的数量规模的增加，CPU 和 GPU 运行时间之间的差距也越来越大。当多面体面数增加

① 运行时环境为：Intel(R) Core(TM) i5-2320 CPU @ 3.2GHz 8G RAM NVIDIA GeForce GTX 650。



(a) Budda 模型 (31232 个点)　(b) Dinosaur 模型 (40277 个点)

(c) Alice 模型 (224291 个点)　(d) Bugatti 模型 (1010815 个点)

图 2.11　基于 CUDA 并行算法运行时间

即 k 的增大时，在 Z Buffer 算法中，需要更多的绘制次数，因此其运行时间也有所增加，而在基于乒乓技术的算法中，当点规模一定时，k 的变化对最后运行时间影响不明显，因此当较大的 k 时，这种算法更快。

表 2.1 详细的展示了基于着色器的两种算法应用于 Alice 和 Bugatti 模型的构造时间及相应的加速比。从中可看出，Z Buffer 算法适合相对较小的 k 值，而基于乒乓技术的算法在较大 k 值时能达到更大的加速比。

本文实现了文献 [42] 中的并行算法并进行对比实验，表 2.2 为实验统计结果，表中数据均为搜索截面耗时，因为二者其他步骤均相同且从 2.11 可得搜索截面时间占用整体绝大部分时间，其中 k 为多面体面数，列 SSE 和列 k-CBP 分别为文献 [42] 中的算法和本文提出的基于 CUDA 算法的运行时间（单位毫秒）。

与文献 [42] 中的算法相比，本文算法优势明显。当用于点数量较小 Apple 模型时，能够提高 3~4 倍速度，模型变大，加速比也更大， Bugatti 模型的提速达到 4~8 倍。

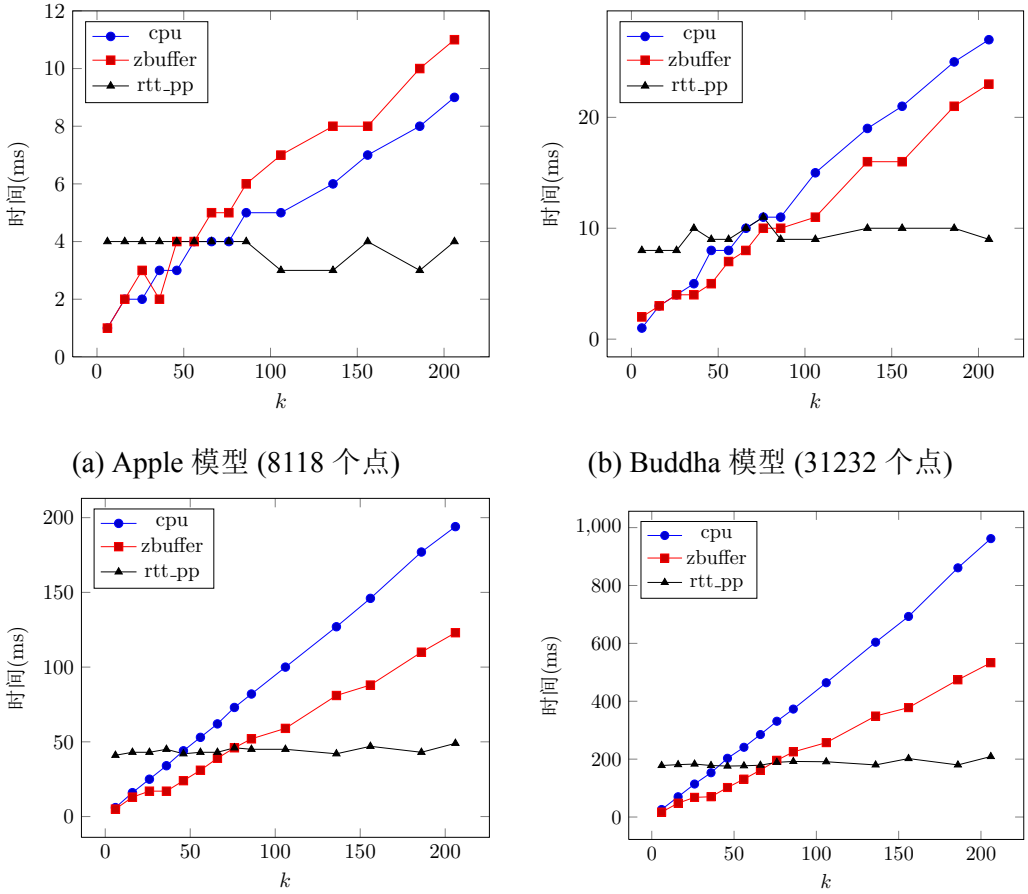
表 2.1　基于着色器并行算法加速比

k	Apple 模型 (8118 个点)				Bugatti 模型 (1010815 个点)			
	cpu (ms)	zbuffer (ms)	rtt_pp (ms)	加速比 (a)	cpu (ms)	zbuffer (ms)	rtt_pp (ms)	加速比 (a)
6	6	5	41	1.20	26	17	178	1.53
16	16	13	43	1.23	70	48	181	1.46
26	25	17	43	1.47	114	68	183	1.68
36	34	17	45	2.00	153	70	178	2.19
46	44	24	42	1.83	203	102	176	1.99
56	53	31	43	1.71	241	131	177	1.84
66	62	39	43	1.59	285	162	179	1.76
76	73	46	46	1.59	331	196	189	1.75
86	82	52	45	1.82	373	225	192	1.94
106	100	59	45	2.22	464	257	191	2.43
136	127	81	42	3.02	604	349	180	3.36
156	146	88	47	3.11	693	378	202	3.43
186	177	110	43	4.12	861	474	180	4.78
206	194	123	49	3.96	962	533	209	4.60

(a): 加速比 = cpu/*min*(zbuffer, rtt_pp)

表 2.2　本文算法与文献 [42] 的并行算法对比

k	Apple 模型 (8118 个点)			Bugatti 模型 (1010815 个点)		
	SSE[42](ms)	k-CBP(ms)	加速比	SSE[42](ms)	k-CBP(ms)	加速比
6	0.4	0.12	3.20	24.2	3.20	7.56
16	0.9	0.26	3.43	44.5	8.44	5.27
26	1.4	0.41	3.38	66.5	13.65	4.87
36	1.9	0.52	3.65	91.1	18.34	4.97
46	2.5	0.67	3.74	119.5	24.13	4.95
56	2.9	0.79	3.66	138.4	28.86	4.80
66	3.5	0.95	3.69	170.6	34.10	5.00
76	4.0	1.08	3.70	197.1	39.85	4.95
86	4.5	1.22	3.69	219.8	45.08	4.88
106	5.4	1.49	3.62	267.8	55.52	4.82
136	6.8	1.92	3.54	342.9	71.24	4.81
156	7.7	2.17	3.55	411.3	81.18	5.07
186	9.3	2.60	3.58	479.4	97.39	4.92
206	10.5	2.85	3.68	523.0	106.87	4.89



(a) Apple 模型 (8118 个点)　(b) Buddha 模型 (31232 个点)

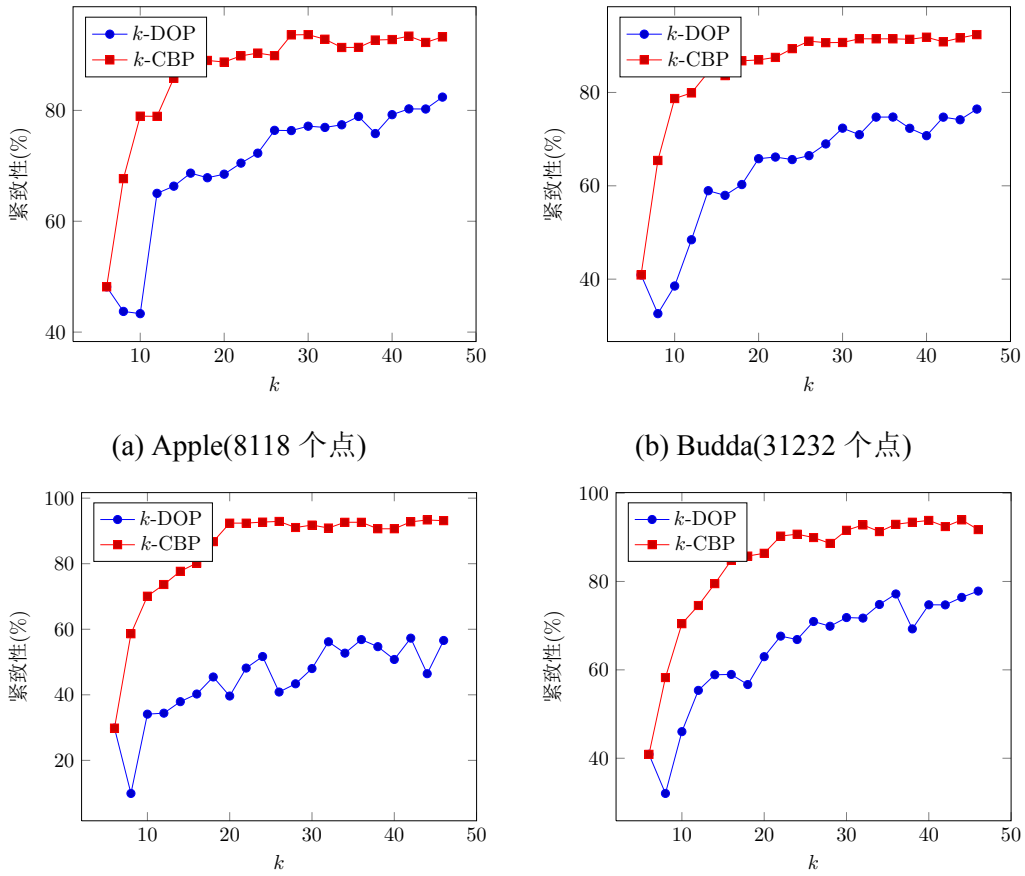
(c) Alice 模型 (224291 个点)　(d) Bugatti 模型 (1010815 个点)

图 2.12　基于着色器的并行算法运行时间

2.4.2　凸包围多面体紧致程度

凸包围多面体的质量用公式 (1-1) 衡量即通过凸包与凸包围多面体的体积之比来量化包围体的紧致程度。文献 [14] 是基于 k-DOP 实现的层次结构的包围体，其顶层的 k-DOP 与本文算法生成的 k-CBP 的紧致程度对比如图 2.13 所示，图中横坐标 k 为凸包围多面体的面数，纵坐标为紧致程度，曲线 k-DOP 和 k-CBP 分别为文献 [14] 和本文的方法，由图可知，对于不同模型，本文构造的凸包围多面体紧致程度均有所提升，如 Alice 模型提升了 12.08%， Dinasour 模型提升了 34.0%。以 $k \in \{20, 38\}$ 为例，相应模型的 k-DOP 和 k-CBP 效果可见图 2.14。

本文算法利用近似凸包与精确凸包的相似性，从近似凸包的众多面片对应的法向中通过 k-means 聚类算法生成 k 个法向。表 2.3 为利用 Alice 模型的近似凸包和精确凸包分别聚类生成法向构造 k-CBP 的紧致程度对比。可以看出，一方面，利用精确凸包构造的 k-CBP 并不一定比近似凸包构造的结果更紧致，但由于近似凸包与精确凸包的外观的近似性因此二者得到的结果相差并不大 (5% 以内)；另一



(a) Apple(8118 个点)　(b) Budda(31232 个点)

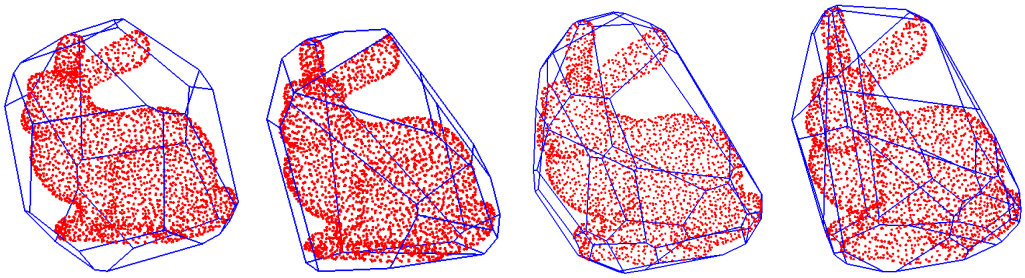
(c) Dinosaur(40277 个点)　(d) Alice(224291 个点)

图 2.13　紧致程度对比：k-DOP 为文献 [14] 的算法, k-CBP 为本文算法

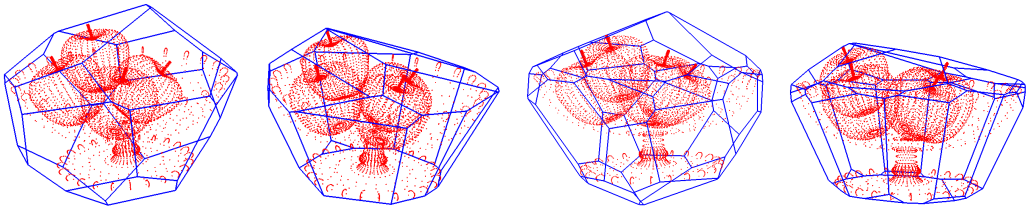
方面，因近似凸包的时间复杂度为线性，而精确凸包为 $O(n\log n)$，因此本文利用近似凸包聚类生成法向。示例中构造近似凸包耗费时间仅为 0.88ms ，而精确凸包为 79.31ms 。

表 2.3　用近似凸包与精确凸包构造 k-CBP 紧致程度对比

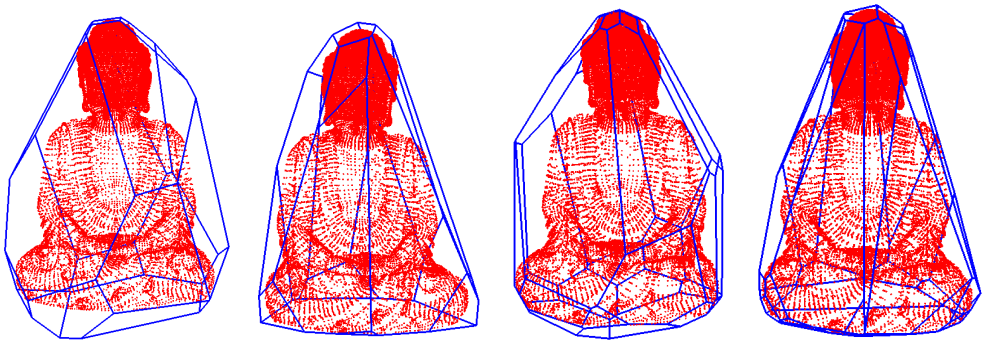
k	τ(ACHull)(%)	τ (CHull)(%)	k	τ (ACHull)(%)	τ (CHull)(%)
10	70.44	65.56	26	89.93	94.45
12	74.56	75.63	28	88.61	92.24
14	79.50	82.49	30	91.58	92.35
16	84.79	85.20	32	92.78	93.78
18	85.75	90.09	34	91.28	93.22
20	86.38	90.36	36	92.93	94.68
22	90.27	92.48	38	93.41	93.20
24	90.70	93.09	40	93.81	94.70



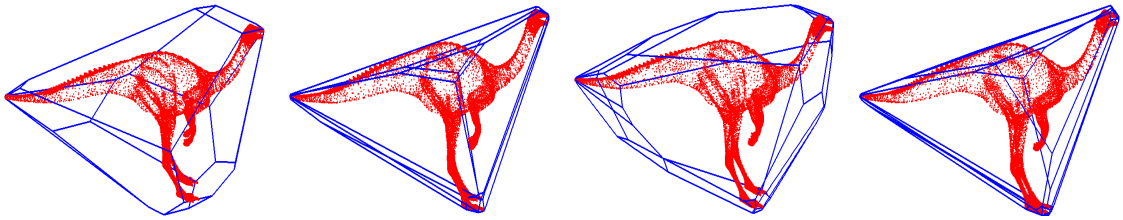
(a) Bunny(20-DOP)　(b) Bunny(20-CBP)　(c) Bunny(38-DOP)　(d) Bunny(38-CBP)



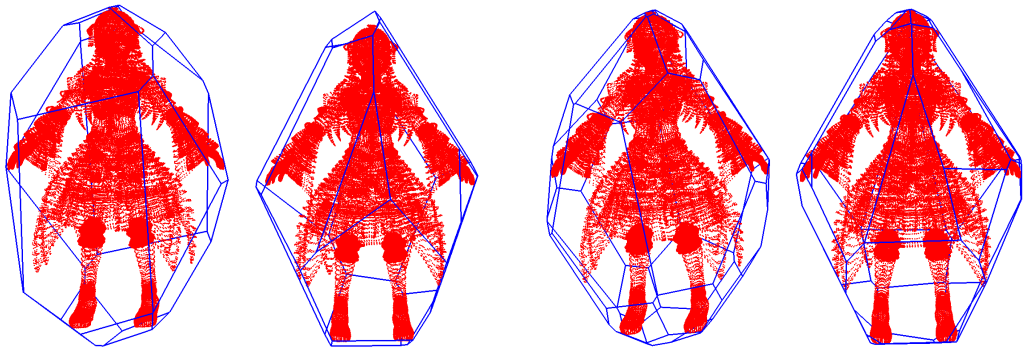
(e) Apple(20-DOP)　(f) Apple(20-CBP)　(g) Apple(38-DOP)　(h) Apple(38-CBP)



(i) Budda(20-DOP)　(j) Budda(20-CBP)　(k) Budda(38-DOP)　(l) Budda(38-CBP)



(m) Dinosaur(20-DOP)　(n) Dinosaur(20-CBP)　(o) Dinosaur(38-DOP)　(p) Dinosaur(38-CBP)

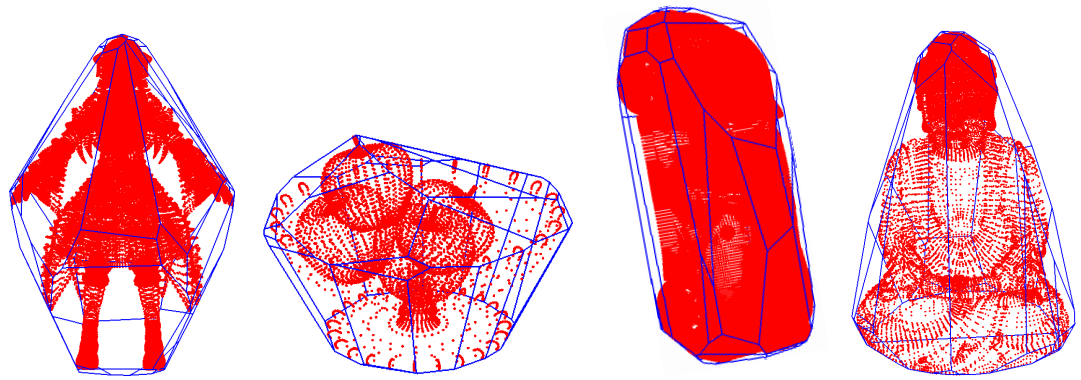


(q) Alice(20-DOP)　(r) Alice(20-CBP)　(s) Alice(38-DOP)　(t) Alice(38-CBP)

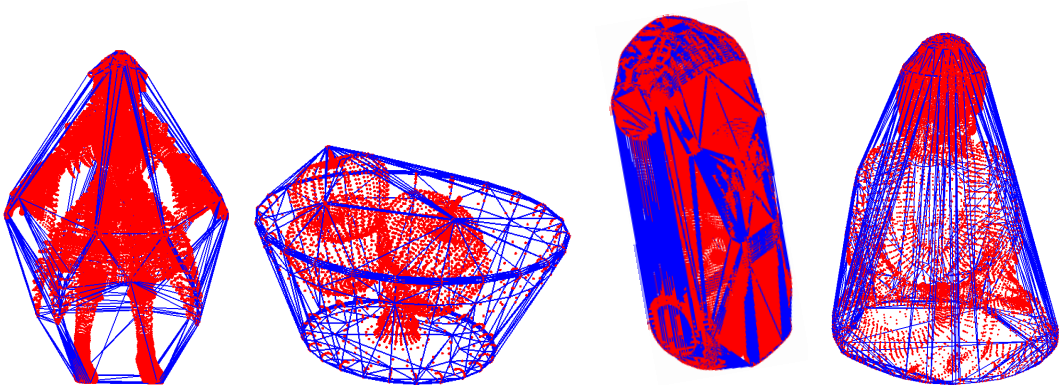
图 2.14　*k*-CBP 与 *k*-DOP 对比

表 2.4　*k*-CBP 与 QuickHull 凸包算法比较

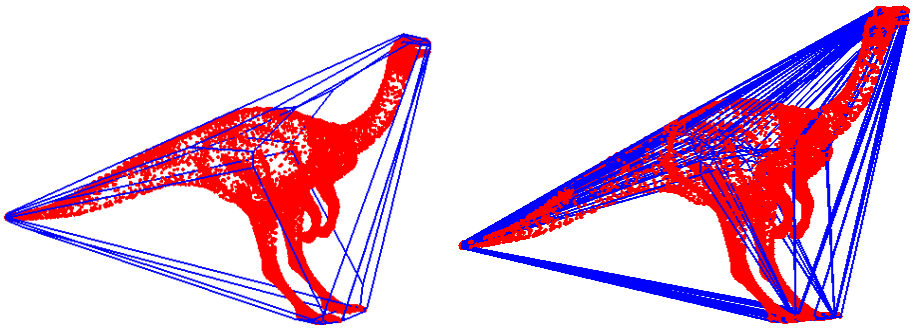
Model	*f*(CHull)	*f*(*k*-CBP)	τ (*k*-CBP)(%)	*t*(CHull)(ms)	*t*(*k*-CBP)(ms)
Apple	499	30	93.67	5.5	1.30
Budda	1608	46	92.39	21.3	2.86
Dinosaur	1240	44	93.34	22.6	1.99
Alice	1332	44	93.92	85.8	8.47
Bugatti	24654	44	95.06	688.7	25.41



(a) Alice(44-CBP)　(b) Apple(30-KCBP)　(c) Bugatti(44-CBP)　(d) Budda(46-CBP)



(e) Alice(CHull)　(f) Apple(CHull)　(g) Bugatti(CHull)　(h) Budda(CHull)



(i) Dinasour(44-CBP)　(j) Dinosaur(CHull)

图 2.15　*k*-CBP 与凸包对比

本文算法与 CGAL 库中利用 QuickHull 算法构造的凸包进行比较的结果如表 2.4 所示，其中 f(CHull) 和 f(k-CBP) 分别表示凸包的面数和凸包围多面体的面数，τ(k-CBP) 为凸包围多面体的紧致程度，t(CHull) 和 t(k-CBP) 分别表示凸包和 k-CBP 构造所花费的时间。相应模型的可视化结果如图 2.15 所示。与凸包相比，本文算法在大大简化包围体平面数量的同时能保持较好的紧致程度，例如 Apple 模型的凸包有 499 个平面，本文算法仅用 30 个平面就能达到 93.67% 的紧致程度，而对 Bugatti 模型，仅用了其凸包平面数量的 0.17% 就达到 95.06% 的紧致程度，且构造速度快了 27 倍。

2.5　本章小结

本章主要介绍了凸包围多面体 k-CBP 的生成算法，算法主要分为 3 个步骤，首先确定生成的 k-CBP 的法向，为了得到更加紧致的凸包围多面体，本文利用 k-means 对构造的近似内凸包的法向进行聚类；然后多次扫描模型点集得到每个法向的切点进而得到截面，该过程各方向计算相互独立互不影响，因此利用了 GPU 进行加速；最后通过截面求交得到 k-CBP 的各个顶点。本章最后通过实验从凸包围多面体的生成效率和紧致程度两个角度与现有算法进行对比，说明本文算法能够快速构造更加紧致的凸包围多面体，较现有算法相比效率上平均能够提高 3 ~ 8 倍，且较 k-DOP 提高了近 10% ~ 40% 的紧致程度。

第 3 章　基于 *k*-CBP 碰撞检测算法

碰撞检测算法是计算机图形学、计算机动画等领域里必不可少的基础算法之一。本章提出了基于 *k*-CBP 的碰撞检测算法，算法首先对输入的网格模型进行预处理，构造模型的包围盒即 AABB 包围体、*k*-CBP，当进行碰撞检测时，首先判断包围盒是否相交，若相交再进行 *k*-CBP 之间的相交测试，再次相交再进行实际模型的相交测试。在进行实际模型的相交测试时，利用了 AABB 树形结构进行判断剪枝，在 *k*-CBP 之间分别用 AABB 树的方式和基于 GJK 算法两种方式进行，实验结果表明本文提出的方法能够有效加速碰撞检测算法。

本章后续部分的内容组织如下：第一节介绍了两种 *k*-CBP 之间的相交测试算法，分别是基于 AABB 树的算法和基于 GJK 的算法；第二节介绍了三角网格之间的相交测试算法；第三节介绍基于 *k*-CBP 的碰撞检测算法总体的流程；然后是实验结果的分析以及小结。

3.1　*k*-CBP 之间的相交测试算法

在所有基于包围体的碰撞检测算法中，都是利用了包围体的相交测试比直接用原始模型相交测试更简单以提升算法的整体效率，包围体的相交测试是非常重要的一个步骤。与其他基于包围体的碰撞检测算法一样，本文基于 *k*-CBP 的算法也是先进行 *k*-CBP 的相交测试，若 *k*-CBP 相交，再进行原始模型的相交测试。*k*-CBP 之间的相交测试以两种方法实现，一种是将构造的 *k*-CBP 进行空间划分，构造 *k*-CBP 的 AABB 树，再基于 AABB 树进行相交测试，详细划分原则等算法见第 3.1.1 节；另一种是基于计算凸多面体之间的最近距离的算法 GJK，详细算法见第 3.1.2节。

3.1.1　基于 AABB 树的算法

AABB 包围体是碰撞检测算法过程中一种最常用的包围体，构造模型的 AABB 包围体树能够有效提高模型的求交或碰撞检测过程。本文将 *k*-CBP 视为普通的三角网格模型，采取一种自上而下的构造 AABB 包围体树的方法，顶层包围体为所有三角网格的包围体的并集，然后按照包围体跨度最大的维度进行划分成两个子节点，然后再对每个子节点进行递归划分，具体算法如算法 7 所示。

为了使生成的 AABB 包围体树更加平衡，因此划分策略为两个子节点包含相同数量的三角网格。划分轴采用沿着坐标轴方向节点跨度最大的轴进行划分，在

算法 7 AABB 树的构造

输入: 原始三角网格数组 *primitives* 和对应数组下标 $first, last$
输出: AABB 树的根节点 *root*

```
1:  function ConstructAABBTree(primitives, first, last)
2:      root ← constructNode()
3:      root.primitives ← primitives
4:      for i = first → last do
5:          root.box ← root.box ∪ primitives[i].box  // 求每个三角网格的 AABB 的并集
6:      end for
7:      size ← primitives.size
8:      if size = 1 then
9:          return root
10:     end if
11:     axis ← longestAixs(root.box)  // 计算包围体跨度最大的轴
12:     sort(primitives, axis)  // 按照 axis 对 primitives 排序
13:     if size = 2 then
14:         root.left ← constructNode()
15:         root.left.primitives ← primitives[0]
16:         root.left.box ← primitives[0].box
17:         root.right ← constructNode()
18:         root.right.primitives ← primitives[1]
19:         root.right.box ← primitives[1].box
20:         return root
21:     end if
22:     half ← size/2
23:     root.left ← ConstructAABBTree(primitives, first, half)
24:      // 前一半作为其中一个叶子节点，继续递归构造
25:     root.right ← ConstructAABBTree(primitives, half + 1, size)  // 后一半继续递归构造
26:     return root
27: end function
```

对三角网格进行排序时，通常可选择三角网格中心位置的某轴向坐标值进行排序，因为本文的策略为平衡二叉树策略，两个孩子节点包含三角网格数量一致，因此该点的选择不会对总体划分产生较大影响，只会影响划分轴边缘的三角网格，因此本文仅仅简单选择三角网格第一个坐标点的轴向坐标轴进行排序。

算法 7 的叶子节点为单个的三角网格，根据实际需要可以设置 AABB 树的最大深度，将达到最大深度的全部三角网格结合构成一个叶子节点。假设算法 7 的时间复杂度为 $T(n)$，则有

$$T(n) = O(n\log n) + 2T(\frac{n}{2}), \tag{3-1}$$

公式 (3-1）中 $O(n\log n)$ 为算法中根据某坐标轴排序的耗费，根据主定理得此构造包围体树的整体算法时间复杂度 $T(n) = O(n\log^2 n)$，文献 [3] 中提到可以用一种 $O(n)$ 的算法替代其中的排序操作，使得整体复杂度为 $O(n\log n)$。

生成两个 *k*-CBP 的 AABB 包围体树后，当进行碰撞检测时，将采用算法 8 的迭代算法进行遍历判断。从顶层 AABB 节点开始，若两个节点的 AABB 包围体相

交，则进行深度优先遍历其孩子节点，当到达叶子节点时，再进行原生几何（本文中的 k-CBP 多边形网格，为了统一转化成与输入模型一致的三角网格）进行相交测试的判断，k-CBP 相交后，进行真实模型的相交测试也通过此方法进行。

算法 8 基于 AABB 树碰撞检测迭代算法

输入: 两个 AABB 树的根节点 $rootA, rootB$
输出: 模型是否相交

```
1:  function TraverseDetection(rootA, rootB)
2:      p ← initStack(), q ← initStack()  // 初始化两个栈，用于记录待判断的 AABB 节点对
3:      p.Push(rootA), q.Push(rootB)
4:      while !p.empty() and !q.empty() do
5:          nodeA ← p.Pop()
6:          nodeB ← q.Pop()
7:          c ← Intersect(nodeA.box, nodeB.box)  // 判断两个节点的 AABB 包围体是否相交
8:          if c = False then
9:              Continue  // 节点 AABB 不相交，则过滤到这两个节点及其孩子节点
10:         end if
11:         if nodeA.isLeaf() then
12:             if nodeB.isLeaf() then  // 两个叶子节点的原始几何进行相交测试
13:                 for all p1 ∈ nodeA.primitives do
14:                     for all p2 ∈ nodeB.primitives do
15:                         if Intersect(p1, p2) = True then
16:                             // 按照第 3.2 节中的算法进行原始三角网格相交测试
17:                             return True  // 若相交就直接返回 True
18:                         end if
19:                     end for
20:                 end for
21:             else // nodeB 节点有孩子节点
22:                 p.push(nodeA), q.push(nodeB.left)
23:                 p.push(nodeA), q.push(nodeB.right)
24:             end if
25:         else // nodeA 节点有孩子节点
26:             if nodeB.isLeaf() then  // nodeB 是叶子节点
27:                 p.push(nodeA.left), q.push(nodeB)
28:                 p.push(nodeA.right), q.push(nodeB)
29:             else // nodeA 和 nodeB 都有叶子节点
30:                 p.push(nodeA.left), q.push(nodeB.left)
31:                 p.push(nodeA.left), q.push(nodeB.right)
32:                 p.push(nodeA.right), q.push(nodeB.left)
33:                 p.push(nodeA.right), q.push(nodeB.right)
34:             end if
35:         end if
36:     end while
37:     return False  // 遍历完毕也没有检测到原始三角网格相交，则返回 False
38: end function
```

算法 8 中，底层叶子节点包含的原始三角网格数量与树的高度相关，对于相同的模型，树的高度越低，叶子节点包含三角网格数量也就越多，且叶子节点测试的时间复杂度为 $O(m^2)$，m 为叶子节点包含的三角网格数量，一般而言在存储

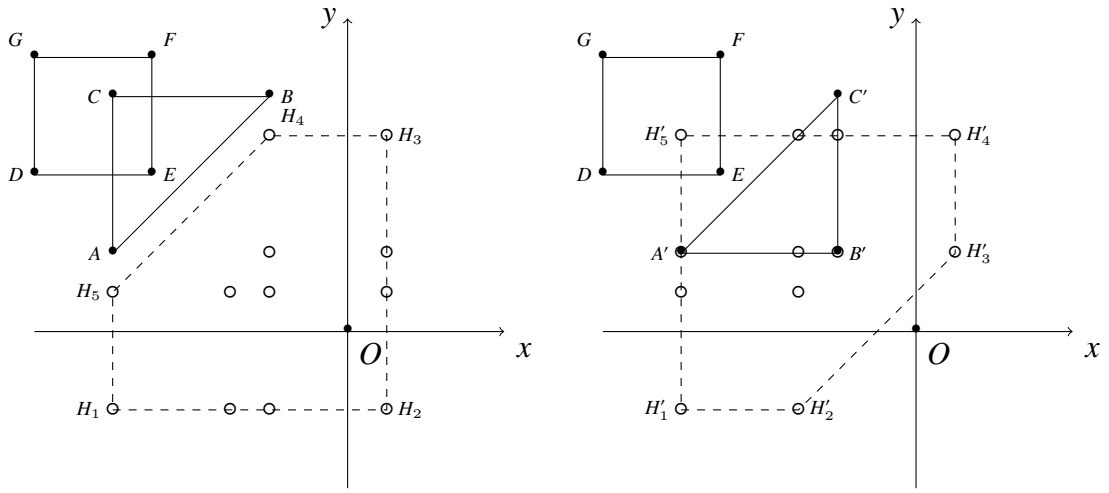
允许的情况下都尽量使得最底层叶子节点仅包含 1 个或少数几个三角形，以减少底层叶子节点三角网格两两测试的时间复杂度。

当在运动场景中的模型进行碰撞检测时，需要对 *k*-CBP 及模型的 AABB 包围体树进行更新，本文采用一种近似的算法进行计算，详细将在第 3.3 节中介绍。

3.1.2　基于 GJK 的算法

GJK 算法是 E.G.Gilbert，D. W. Jonhson 和 S.S. Keerthi（取三位作者姓名首字母作为算法名称的缩写）等人在 1988 年发表的一种用于计算三维欧式空间中两个凸集合点之间的最近距离的方法[46]，后常用于凸体之间的碰撞检测算法[47]。本文也利用此算法用于计算两个凸包围多面体 *k*-CBP 之间的相交测试。

GJK 算法需要用到闵科夫斯基和（Minkowski Sum）的概念如下定义：两个点集 $\mathbb{A}$ 和 $\mathbb{B}$，其 Minkowski 和为 $\mathbb{A}+\mathbb{B}=\{\boldsymbol{a}+\boldsymbol{b}|\boldsymbol{a}\in\mathbb{A},\boldsymbol{b}\in\mathbb{B}\}$，将相应的加号改为减号得到 Minkowski 差，即 $\mathbb{A}-\mathbb{B}=\{\boldsymbol{a}-\boldsymbol{b}|\boldsymbol{a}\in\mathbb{A},\boldsymbol{b}\in\mathbb{B}\}$，GJK 算法的核心基础在于若两个凸体相交，则其 Minkowski 差必包含原点，因为若 $\mathbb{A}$ 和 $\mathbb{B}$ 相交即 $\mathbb{A}$ 和 $\mathbb{B}$ 必含有公共交集，即至少含有一点同时属于 $\mathbb{A}$ 和 $\mathbb{B}$，该点的 Minkowski 差即为原点 $\boldsymbol{O}(0,0)$，更加详细的原理和证明过程可以参考文献 [46,47]。下面将以一个二维例子阐述此思想。



(a) 相交　　(b) 不相交

图 3.1　二维 GJK 算法示例

图 3.1(a) 中，四个点坐标 $D(-4,2),E(-2.5,2),F(-2.5,3.5),G(-4,3.5)$ 构成了四边形 *DEFG* 的四个顶点，三角形 *ABC* 的坐标分别为 $A(-3,1),B(-1,3),C(-3,3)$，其 Minkowski 差为 $\square_{DEFG}-\triangle_{ABC}=\{(0.5,1),(-1.5,0.5),(0.5,2.5),(-1,2.5),(-3,-1),(-1,-1),(0.5,0.5),(-1,1),(-3,0.5),(0.5,-1),(-1.5,-1),(-1,0.5)\}$，四边形 *DEFG* 与

三角形 ABC 相交，该 Minkowski 差构成的多边形 $H_1H_2H_3H_4H_5$ 包含原点 $\boldsymbol{O}$。而在图 3.1(b) 中，三角形坐标变为 $A'(-3,1), B'(-1,1), C'(-1,3)$，四边形 $DEFG$ 坐标不变，其 Minkowski 差为 $\square_{DEFG} - \triangle_{A'B'C'} = \{(0.5,1),(-1.5,1),(0.5,2.5),(-1.5,-1),(-1,2.5),(-3,2.5),(-1.5,2.5),(-3,1),(-1.5,0.5),(-3,0.5),(-1,1),(-3,-1)\}$，四边形 $DEFG$ 与三角形 $A'B'C'$ 不相交，该 Minkowski 差构成的多边形 $H'_1H'_2H'_3H'_4H'_5$ 不包含原点 $\boldsymbol{O}$。

通过该方法可将两个凸多边形或多面体是否相交转化为其 Minkowski 差是否包含原点，假设两个凸多边形或多面体分别有 m 和 n 个点，则直接计算其 Minkowski 差的时间复杂度为 $O(m \cdot n)$。GJK 算法并不直接计算其 Minkowski 差，而是采取一种迭代的算法计算其 Minkowski 差一步一步向原点逼近，在有限步内若包含原点则原凸多边形或凸多面体相交反之不相交，文献 [47] 证明了该迭代过程能在有限步内完成。

GJK 算法中定义支持映射（Support Mapping）函数为将给定的方向 $\boldsymbol{d}$ 映射为凸多边形或多面体中沿着 $\boldsymbol{d}$ 方向最远的点，该点称为支持点（Support Point），用符号 $Support(\boldsymbol{d})$ 表示，该概念与本文第 2.2节中搜索截面中的最大投影值点一致。设 $\mathbb{A}$ 和 $\mathbb{B}$ 的 Minkowski 差为 $\mathbb{D} = \mathbb{A} - \mathbb{B}$，则 $Support_{\mathbb{D}}(\boldsymbol{d}) = Support_{\mathbb{A}}(\boldsymbol{d}) - Support_{\mathbb{B}}(-\boldsymbol{d})$，这样能使得 Minkowski 差 $\mathbb{D}$ 围成的区域更大并尽可能的包含原点。

GJK 算法采用如下的流程[3] 进行迭代计算两个凸多面体 $\mathbb{A}$ 和 $\mathbb{B}$ 是否相交：

(1) 任选方向 $\boldsymbol{d}$ 计算 $Support_{\mathbb{D}}(\boldsymbol{d}) = Support_{\mathbb{A}}(\boldsymbol{d}) - Support_{\mathbb{B}}(-\boldsymbol{d})$，即任意从 $\mathbb{A}$ 和 $\mathbb{B}$ 的 Minkowski 差中选择一点，加入集合 $\boldsymbol{S}$；

(2) 计算在 $ConvexHull(S)$ 中，具有最小二范数的点即离原点最近的点 $\boldsymbol{D}$；

(3) 如果 $\boldsymbol{D}$ 是原点本身或 $ConvexHull(S)$ 包含原点，则表明 $\mathbb{A}$ 和 $\mathbb{B}$ 相交，停止迭代；

(4) 规约集合 $\boldsymbol{S}$，使得 $\boldsymbol{S}$ 中是满足 $\boldsymbol{D}$ 属于其凸包的最小的集合；

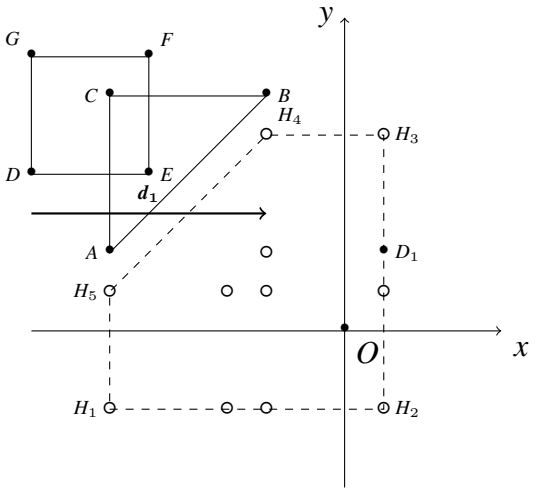
(5) 更新方向 $\boldsymbol{d} = -\boldsymbol{D}$，并计算 $Support_{\mathbb{D}}(\boldsymbol{d}) = Support_{\mathbb{A}}(\boldsymbol{d}) - Support_{\mathbb{B}}(-\boldsymbol{d})$；

(6) 如果有新的 $Support_{\mathbb{D}}(\boldsymbol{d})$ 产生，则添加至集合 $\boldsymbol{S}$ 并从第 (2) 步继续迭代，否则结束迭代，且得出 $\mathbb{A}$ 和 $\mathbb{B}$ 不相交的结论。

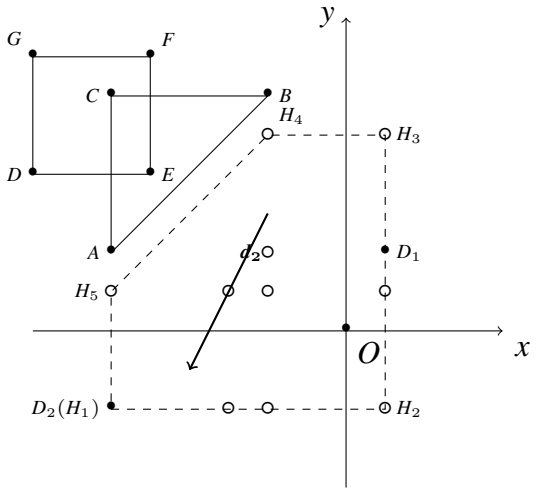
具体而言，仍以图 3.1(a) 为例，令四边形 $DEFG$ 为 $\mathbb{A}$，三角形 ABC 为 $\mathbb{B}$，迭代过程如下：

(1) 第一次迭代不妨设 $\boldsymbol{d_1} = (1,0)$，如图 3.2(a) 所示，$Support_{\mathbb{D}}(\boldsymbol{d_1}) = Support_{\mathbb{A}}(\boldsymbol{d_1}) - Support_{\mathbb{B}}(-\boldsymbol{d_1}) = E(-2.5,2) - A(-3,1) = D_1(0.5,1)$, 此时，$\boldsymbol{S} = \{D_1(0.5,1)\}$；

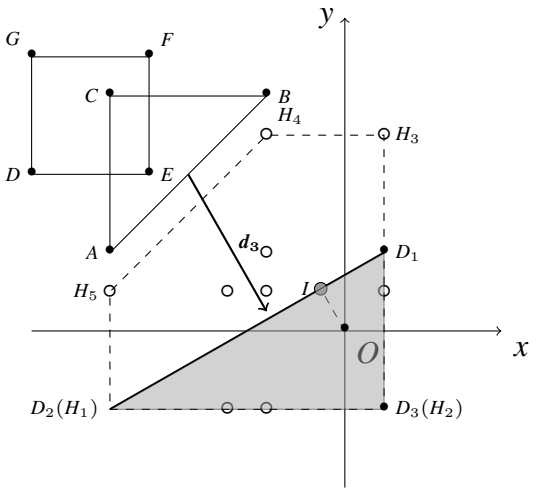
(2) 第二次迭代取 $\boldsymbol{d_2} = -\boldsymbol{D_1} = (-0.5,-1)$，如图 3.2(b) 所示，$Support_{\mathbb{D}}(\boldsymbol{d_2}) = Support_{\mathbb{A}}(\boldsymbol{d_2}) - Support_{\mathbb{B}}(-\boldsymbol{d_2}) = D(-4,2) - B(-1,3) = D_2(-3,-1)$, 此时，$\boldsymbol{S} =$



(a) 第一次迭代



(b) 第二次迭代



(c) 第三次迭代

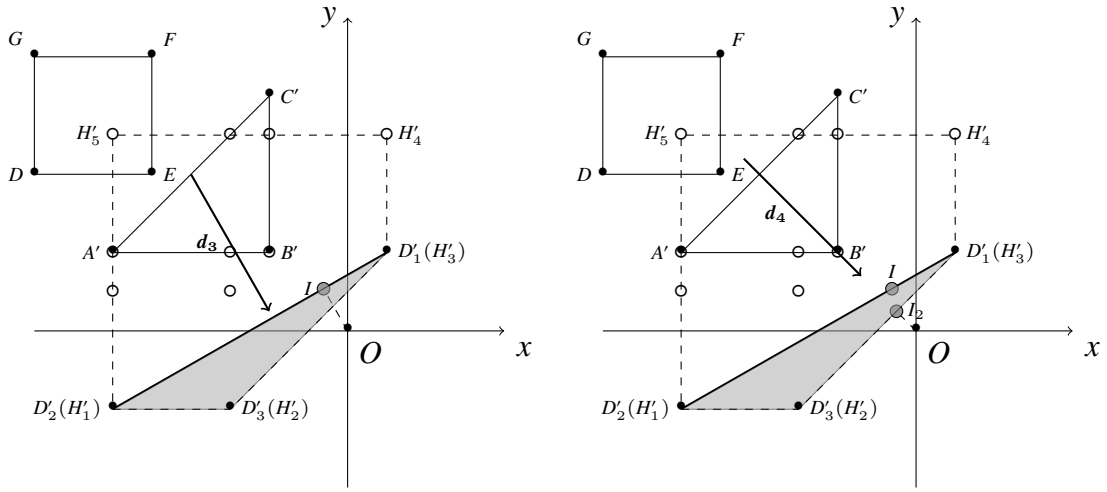
图 3.2　二维 GJK 算法迭代示例（相交情况）

$\{D_1(-0.5,1), D_2(-3,-1)\}$;

(3) 第三次迭代，计算 $ConvexHull(S)$ 中最小二范数的点为 $I(-4/13, 7/13)$，并令 $\boldsymbol{d_3} = -\boldsymbol{I} = (4/13, -7/13)$，$Support_{\mathbb{D}}(\boldsymbol{d_3}) = Support_{\mathbb{A}}(\boldsymbol{d_3}) - Support_{\mathbb{B}}(-\boldsymbol{d_3}) = E(-2.5,2) - C(-3,3) = D_3(0.5,-1)$, 此时，$\boldsymbol{S} = \{D_1(0.5,1), D_2(-3,-1), D_3(0.5,-1)\}$，如图 3.2(c) 所示，$ConvexHull(S)$ 为阴影部分覆盖区域，因其包含原点故结束迭代，因此，$\mathbb{A}$ 和 $\mathbb{B}$ 相交。

再以图 3.1(b) 为例，设三角形 $\mathbb{B}' = A'B'C'$，该例前两次迭代与上例相同，此处省略直接从第三次迭代起：

(1) 第三次迭代，此时 $\boldsymbol{S} = \{(D_1', D_2')\}$，令 $\boldsymbol{d_3} = -\boldsymbol{I} = (4/13, -7/13)$，$Support_{\mathbb{D}}(\boldsymbol{d_3}) = Support_{\mathbb{A}}(\boldsymbol{d_3}) - Support_{\mathbb{B}'}(-\boldsymbol{d_3}) = E(-2.5,2) - C'(-1,3) = D_3'(-1.5,-1)$, 此时 $\boldsymbol{S} = \{D_1'(0.5,1), D_2'(-3,-1), D_3(-1.5,-1)\}$，如图 3.3(a) 所示，



(a) 第三次迭代　　(b) 第四次迭代

图 3.3　二维 GJK 算法迭代示例（不相交情况）

与上例不同，此时 $ConvexHull(S)$ 并不包含原点，所以规约集合 $\boldsymbol{S} = \{(D'_1(0.5,1), D'_3(-1.5,-1)\}$；

(2) 第四次迭代，计算 $\boldsymbol{S}$ 中二范数最小的点为 $I_2(-1/4,1/4)$，并令 $\boldsymbol{d}_4 = -I_2 = (1/4,-1/4)$，$Support_{\mathbb{D}}(\boldsymbol{d_4}) = Support_{\mathbb{A}}(\boldsymbol{d_4}) - Support_{\mathbb{B}'}(-\boldsymbol{d_4}) = E(-2.5,2) - C'(-1,3)$ 或 $A'(-3,1) = D'_3(-1.5,-1)$ 或 $D'_1(0.5,1)$, 此时，并没有新的 $Support_{\mathbb{D}}(\boldsymbol{d})$ 产生，结束迭代，$\mathbb{A}$ 和 $\mathbb{B}$ 不相交。

在实际计算过程中，并不需要计算离原点最近即二范数最小的点的具体坐标值，只需选择垂直已有方向再进行判断选择方向即可，以图 3.2(c) 为例，此时 $\overrightarrow{D_1D_2} = D_2(-3,-1) - D_1(0.5,1) = (-3.5,-2)$，此时只需令 $\boldsymbol{d'_3} = \overrightarrow{D_1D_2} \times \overrightarrow{D_1O} \times \overrightarrow{D_1D_2} = -(\overrightarrow{D_1D_2} \cdot \overrightarrow{D_1O})\overrightarrow{D_1D_2} + (\overrightarrow{D_1D_2} \cdot \overrightarrow{D_1D_2})\overrightarrow{D_1O} = (5,-8.75)$ ①，$Support_{\mathbb{D}}(\boldsymbol{d'_3}) = Support_{\mathbb{A}}(\boldsymbol{d'_3}) - Support_{\mathbb{B}'}(-\boldsymbol{d'_3}) = E(-2.5,2) - C(-3,3) = D_3(0.5,-1)$, 得到的 $Support_{\mathbb{D}}(d'_3)$ 仍为 $D_3(0.5,-1)$，与选择二范数最小点 I 所代表的方向达到相同的结果，因为选择二范数最小点的坐标代表方向与该垂直方向相同，因此结果一致。

判断原点是否被阴影部分面积所覆盖的过程如下：

(1) 通过 $\boldsymbol{d}'_3$ 得到的支持点 $\boldsymbol{D}_3$ 中，$\boldsymbol{d}'_3 \cdot \overrightarrow{OD_3} = (5,-8.75) \cdot (0.5,-1) = 11.25 > 0$ 表明通过 $\boldsymbol{d}'_3$ 方向到支持点的方向覆盖了原点，即原点在线段 $\overline{D_1D_2}$ 线段的右下方；

(2) $\overrightarrow{D_3D_1}$ 垂直并指向阴影部分面积方向为 $\overrightarrow{P_{D3D1}} = \overrightarrow{D_3D_1} \times \overrightarrow{D_3D_2} \times \overrightarrow{D_3D_1} = (-14,0)$，此时，$\overrightarrow{P_{D3D1}} \cdot \overrightarrow{D_3O} = (-14,0) \cdot (-0.5,1) = 7 < 0$ 表明原点 $\boldsymbol{O}$ 在 $\overrightarrow{P_{D3D1}}$ 的正方向即线段 $\overline{D_3D_1}$ 的左边；

① $\boldsymbol{a} \times \boldsymbol{b} \times \boldsymbol{c} = -(\boldsymbol{c} \cdot \boldsymbol{b})\boldsymbol{a} + (\boldsymbol{c} \cdot \boldsymbol{a})\boldsymbol{b}$

(3) 同时，$\overrightarrow{D_3D_2}$ 垂直并指向阴影部分面积方向为 $\overrightarrow{P_{D3D2}} = \overrightarrow{D_3D_2} \times \overrightarrow{D_3D_1} \times \overrightarrow{D_3D_2} = (0, 24.5)$ 且 $\overrightarrow{P_{D3D1}} \cdot D_3O = (0, 24.5) \cdot (-0.5, 1) = 24.5 > 0$ 表明原点 $\boldsymbol{O}$ 在 $\overrightarrow{P_{D3D2}}$ 的正方向即线段 $\overline{D_3D_2}$ 的上边。

以上三个条件全部满足，即原点 $\boldsymbol{O}$ 被阴影部分所覆盖，即 $\mathbb{A}$ 和 $\mathbb{B}$ 的 Minkowski 差包含原点即可确定 $\mathbb{A}$ 和 $\mathbb{B}$ 相交，结束迭代。

相应的以图 3.3(a) 为例，其结果仍保持一致。$Support_{\mathbb{D}}(\boldsymbol{d'_3}) = Support_{\mathbb{A}}(\boldsymbol{d'_3}) - Support_{\mathbb{B}'}(-\boldsymbol{d'_3}) = E(-2.5, 2) - C'(-1, 3) = D'_3(-1.5, -1)$，图 3.3(a) 中，上一步通过 $\boldsymbol{d'_3}$ 得到的支持点为 $D'_3(-1.5, -1)$，且 $\overrightarrow{D'_3D'_1}$ 垂直并指向阴影部分面积方向为 $\overrightarrow{P_{D'_3D'_1}} \times \overrightarrow{D'_3D'_2} \times \overrightarrow{D'_3D'_1} = (-2, -2) * (-3) + 8 * (-1.5, 0) = (-6, 6)$，此时 $\overrightarrow{P_{D3'D1'}} \cdot \overrightarrow{D'_3O} = (-6, 6) \cdot (1.5, 1) = -3 < 0$ 表明下一次迭代方向需要以 $\overrightarrow{P_{D3'D1'}}$ 的反方向开始，此时点 D'_2 可以规约掉，即 $\boldsymbol{d'_4} = (6, -6)$，以此方向得到的支持点 $D'_1(0.5, 1)$，此时，$\boldsymbol{d'_4} \cdot \overrightarrow{OD_1} = (6, -6) \cdot (0.5, 1) = -3 < 0$ 表明通过 $\boldsymbol{d'_4}$ 方向到支持点覆盖的区域没有包含原点，即可排除相交。

具体的算法如算法 9 所示，文献 [47] 证明了集合 $\boldsymbol{S}$ 中在三维空间最多只包含 4 个点，且算法最终会收敛。在实际实现过程中可以根据需求设定最大迭代次数，达到后或者认为没找到相交点或者采取保守态度进行下一步验证。

算法 9 基于 GJK 的 k-CBP 相交检测算法

输入： 两个 k-CBP $k\text{-}CBP_1, k\text{-}CBP_2$
输出： k-CBP 是否相交

```
1:  function KCBPDetectionBasedOnGJK(k-CBP_1, k-CBP_2)
2:      d ← initNormal()
3:      D ← Support(k-CBP_1, k-CBP_2, d)
4:      S ← {p}
5:      iter ← 1, d ← −d
6:      while inter++ < MaxIter do
7:          D ← Support(k-CBP_1, k-CBP_2, d)
8:          if D · d < 0 then
9:              return False
10:         end if
11:         S ← S ∪ D
12:         contains ← CheckContainUpdate(S, d) // 检测是否包含原点，对集合 S 进行规约，并获取下一次迭代的方向 d
13:         if contains then
14:             return True // 包含原点，直接返回相交，否则继续迭代
15:         end if
16:     end while
17:     return False // 达到最大迭代次数，根据需求返回相交或者不相交
18: end function
```

算法 9 第 12 行 CHECKCONTAINUPDATE 子过称需检测 $\boldsymbol{S}$ 是否包含原点，必要时进行规约并获取下一次迭代方向，以上两个例子已经对二维情况进行分析，

具体三维实现可以参考文献 [47]。

3.2　两个三角网格的相交测试算法

所有针对三角网格模型的碰撞检测算法最终都离不开三角网格的相交测试，不管模型是用何种包围体采用几何或者代数的方法都需要进行三角网格的相交测试。本文将利用一种几何代数相结合的方法进行三角网格之间的相交测试。具体而言，假设两个不同面的三角形所在平面交于直线 $\boldsymbol{L}$，两个三角形位置关系分为相交或者不相交两种情况，如图 3.4 所示。

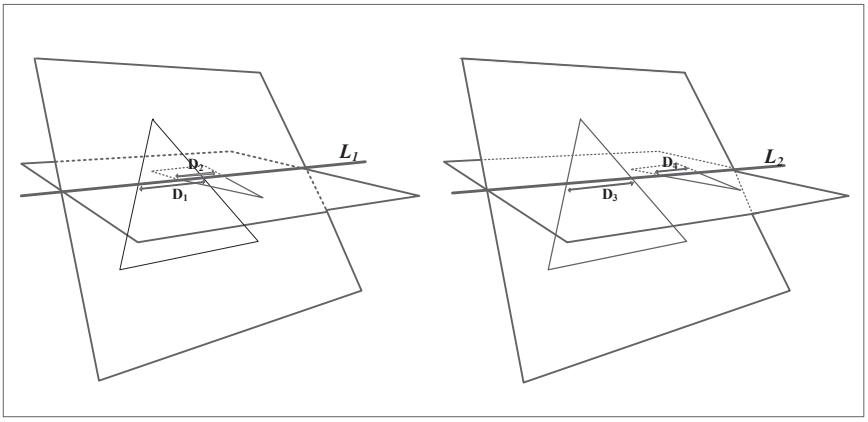


图 3.4　空间中两个非共面三角形的位置关系[64]

图 3.4 左图中，两个三角形所在平面交于直线 $\boldsymbol{L}_1$ 且与三角形公共部分为线段 $\boldsymbol{D}_1$，另一个三角形与 $\boldsymbol{L}_1$ 交于线段 $\boldsymbol{D}_2$，相应的右图两个三角形分别交公共交线 $\boldsymbol{L}_2$ 于 $\boldsymbol{D}_3$ 和 $\boldsymbol{D}_4$，其中 $\boldsymbol{D}_1$ 和 $\boldsymbol{D}_2$ 相交可以推出两个三角形相交，反之因 $\boldsymbol{D}_3$ 和 $\boldsymbol{D}_4$ 不相交因此三角形不相交，因此只需要判断两个三角形与平面交线的线段是否相交即可[64]，以下方法都基于此结论。

假设两个三角形 $T_1(\boldsymbol{U}_1, \boldsymbol{U}_2, \boldsymbol{U}_3), T_2(\boldsymbol{V}_1, \boldsymbol{V}_2, \boldsymbol{V}_3)$ 所在平面分别为 Π_1, Π_2，相应的法向分别为 $\boldsymbol{n}_1, \boldsymbol{n}_2$，假设平面方程

$$\Pi_1 = \boldsymbol{n}_1 \cdot \boldsymbol{x} + d_1 = (\boldsymbol{U_2} - \boldsymbol{U_1}) \times (\boldsymbol{U_3} - \boldsymbol{U_1}) \cdot \boldsymbol{x} + d_1, \tag{3-2}$$

其中将任意一点 $U_i, i \in \{1,2,3\}$ 带入公式 (3-2) 得 d_1，同理得到 Π_2。

然后将三角形 $\boldsymbol{T}_2$ 的三个顶点带入方程 (3-2) 得到 $\boldsymbol{T}_2$ 到 Π_1 的有向距离 $\boldsymbol{l}_{1i}, i \in \{1,2,3\}$，根据 $\boldsymbol{l}_{1i}$ 的值分为下面三种情况讨论：

(1) 若 $\forall \boldsymbol{i} \in \{1,2,3\}, l_{1i} = 0$，即三角形 $\boldsymbol{T}_2$ 的三个顶点到三角形 $\boldsymbol{T}_1$ 所在 Π_1 的距离都为 0，则两个三角形共面；

(2) 若 $\forall \boldsymbol{i} \in \{1,2,3\}, l_{1i} > 0$ 或 $\forall \boldsymbol{i} \in \{1,2,3\}, l_{1i} < 0$，即三角形 $\boldsymbol{T}_2$ 的三个顶点到三

角形 T_1 所在 Π_1 的有向距离同号，则 T_2 在 Π_1 的同一侧，可立即排除相交；

(3) 其他情况，三角形 T_2 必交 Π_1 于一条线段。

同理可以根据三角形 T_1 到三角形 T_2 所在平面 Π_2 得到类似的情况。

针对情况 (1)，共面的两个三角形求交可以通过两个三角形中三条线段两两判定是否相交最多 9 次线段线段求交判定可得，或者通过我们在文献 [65] 中提出的方法进行，该方法对线段三角形的位置做了详细的分类可通过不超过 6 次线段线段求交判定；

针对情况 (2)，计算出有向距离同号后即可排除相交立即返回；

针对情况 (3)，如图 3.5 所示，不妨设在三角形 T_1 中，点 $\boldsymbol{V}_1, \boldsymbol{V}_2$ 在 Π_2 的一侧，$\boldsymbol{V}_3$ 在另外一侧，且点 $\boldsymbol{V}_1, \boldsymbol{V}_2$ 在平面上投影点分别为 $\boldsymbol{K}_1, \boldsymbol{K}_2$，线段 $\boldsymbol{V}_1\boldsymbol{V}_2, \boldsymbol{V}_2\boldsymbol{V}_3$ 与 $\boldsymbol{L}$ 分别交于点 $\boldsymbol{I}_1, \boldsymbol{I}_2$，点 $\boldsymbol{V}_1, \boldsymbol{V}_2$ 向直线 $\boldsymbol{L}$ 的投影点分别为 $\boldsymbol{P}_1, \boldsymbol{P}_2$，三角形 T_1 交于直线 $\boldsymbol{L}$ 与线段 $\overline{I_1I_2}$。

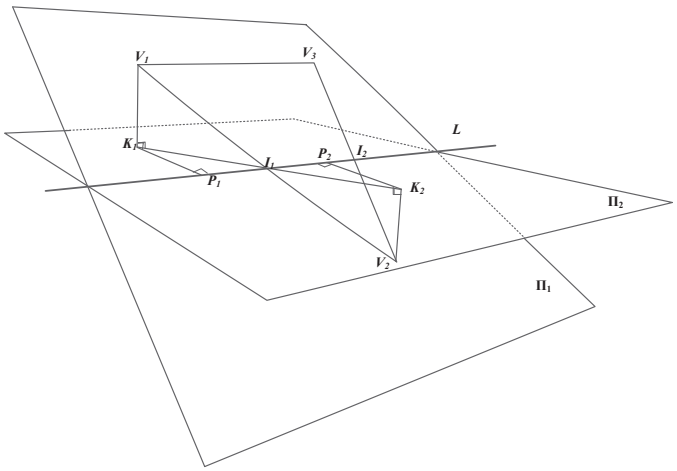


图 3.5　求非共面三角形区间线段示意图[64]

已知直线 $\boldsymbol{L}$ 的方向为 $\boldsymbol{n}_L = \boldsymbol{n}_1 \times \boldsymbol{n}_2$，设 $\boldsymbol{O}$ 为 $\boldsymbol{L}$ 上一点，则直线的参数方程为 $\boldsymbol{L} = t{\cdot}\boldsymbol{n}_L + \boldsymbol{O}$，假设交点 $I_1 = t_1{\cdot}\boldsymbol{n}_L + \boldsymbol{O}$，投影点满足 $p_1 = \boldsymbol{n}_L{\cdot}(\boldsymbol{V}_1 - \boldsymbol{O}), p_2 = \boldsymbol{n}_L{\cdot}(\boldsymbol{V}_2 - \boldsymbol{O})$，由图 3.5 可知，$\triangle K_1P_1I_1 \sim \triangle K_2P_2I_1, \triangle V_1K_1I_1 \sim \triangle V_2K_2I_1$，可得

$$\frac{t_1 - p_1}{p_2 - p_1} = \frac{l_{11}}{l_{11} - l_{12}} \Rightarrow t_1 = (p_2 - p_1)\frac{l_{11}}{l_{11} - l_{12}} + p_1, \tag{3-3}$$

同理可得 $\boldsymbol{I}_2$ 的参数 t_2，三角形 T_2 用相同的方法也能得到区间的参数，即可判断两个区间线段是否相交，进而可得该非共面的两个三角形是否相交。完整的算法如算法 10 所示。

算法 10 第 10 行 在方法 EDGEEDGETEST 进行边边测试子过程中，可通过点与有向线段的位置关系确定，如线段两个端点都在另外一个有向线段的一边说明

两线段不相交，反之则相交，不需要求解出具体的交点坐标。在实际实现过程中，往往需要引入容差以提高算法的稳定性。文献 [64] 介绍了更多的优化技巧。

算法 10 两个三角网格的相交测试算法

输入: 两个三角形的 6 个顶点 $T_1(\boldsymbol{U}_1, \boldsymbol{U}_2, \boldsymbol{U}_3), T_2(\boldsymbol{V}_1, \boldsymbol{V}_2, \boldsymbol{V}_3)$
输出: 三角形是否相交

function TRIANGLETRIANGLEDETECTION($\boldsymbol{U}_1, \boldsymbol{U}_2, \boldsymbol{U}_3, \boldsymbol{V}_1, \boldsymbol{V}_2, \boldsymbol{V}_3$)
    $\Pi_1 \leftarrow \boldsymbol{n}_1 \cdot \boldsymbol{x} + d_1$ // 按照公式 (3-2) 计算 T_1 所在平面方程
    for $i = 1 \rightarrow 3$ **do**
        $l_{1i} \leftarrow \boldsymbol{n}_1 \cdot \boldsymbol{V}_i + d_1$ // 计算 T_2 到 Π_1 的有向距离
    end for
    if ($l_{11} > 0$ **and** $l_{12} > 0$ **and** $l_{13} > 0$)**or**($l_{11} < 0$ **and** $l_{12} < 0$ **and** $l_{13} < 0$) **then**
        return False // T_2 在 Π_1 的同侧，排除
    end if
    if $l_{11} = 0$ **and** $l_{12} = 0$ **and** $l_{13} = 0$ **then**
        return EDGEEDGETEST($\boldsymbol{U}_1, \boldsymbol{U}_2, \boldsymbol{U}_3, \boldsymbol{V}_1, \boldsymbol{V}_2, \boldsymbol{V}_3$)
        // T_2 与 T_1 的共面，普通的边边相交测试
    end if
    $\Pi_2 \leftarrow \boldsymbol{n}_2 \cdot \boldsymbol{x} + d_2$ // 按照公式 (3-2) 计算 T_2 所在平面方程
    for $i = 1 \rightarrow 3$ **do**
        $l_{2i} \leftarrow \boldsymbol{n}_2 \cdot \boldsymbol{U}_i + d_2$ // 计算 T_1 到 Π_2 的有向距离
    end for
    if ($l_{21} > 0$ **and** $l_{22} > 0$ **and** $l_{23} > 0$)**or**($l_{21} < 0$ **and** $l_{22} < 0$ **and** $l_{23} < 0$) **then**
        return False // T_1 在 Π_2 的同侧，排除
    end if
    $t_1 \leftarrow$ CALPARAM, $t_2 \leftarrow$ CALPARAM() // 按照公式 (3-3) 计算线段 $\boldsymbol{D}_1$ 的参数区间
    $t_3 \leftarrow$ CALPARAM, $t_4 \leftarrow$ CALPARAM() // 类似的方法计算线段 $\boldsymbol{D}_2$ 的参数区间
    if OVERLAP(t_1, t_2, t_3, t_4) **then**
        return True // 区间交叉，表明三角形相交返回 **True**
    else
        return False
    end if
end function

3.3　基于 k-CBP 的碰撞检测算法

凸包围多面体可应用于加速相关算法的整体效率，图 3.6 为利用 Bunny 模型进行碰撞检测的示例，图中编号为 1 与 2 的模型、2 与 3 的模型的包围盒分别相交，而其 16-CBP 仅 1 与 2 相交，实际模型仅 1 与 2 相交。用 16-CBP 可排除模型 2 与 3 之间的碰撞检测，而仅用包围盒算法则无法排除，显然检测模型 2 与 3 的 16-CBP 是否相交比直接通过检测模型 2 与 3 是否相交更省时间。本文将从静止场景和运动场景两个角度来验证基于 k-CBP 的碰撞检测算法的有效性。

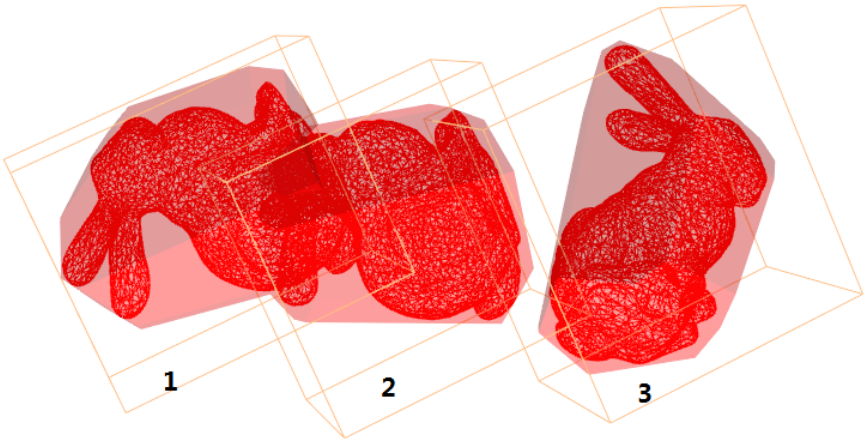


图 3.6　k–CBP 应用于碰撞检测示例

3.3.1　静止场景中的碰撞检测算法

模型的 k-CBP 相交后，会用模型的 AABB 树进一步对模型进行碰撞检测，模型的 AABB 树构造方法如第 3.1.1 节所述，图 3.7 是按照本文所采用的构造方法针对 Bunny 模型构造的 AABB 树形结构的顶上 4 层。

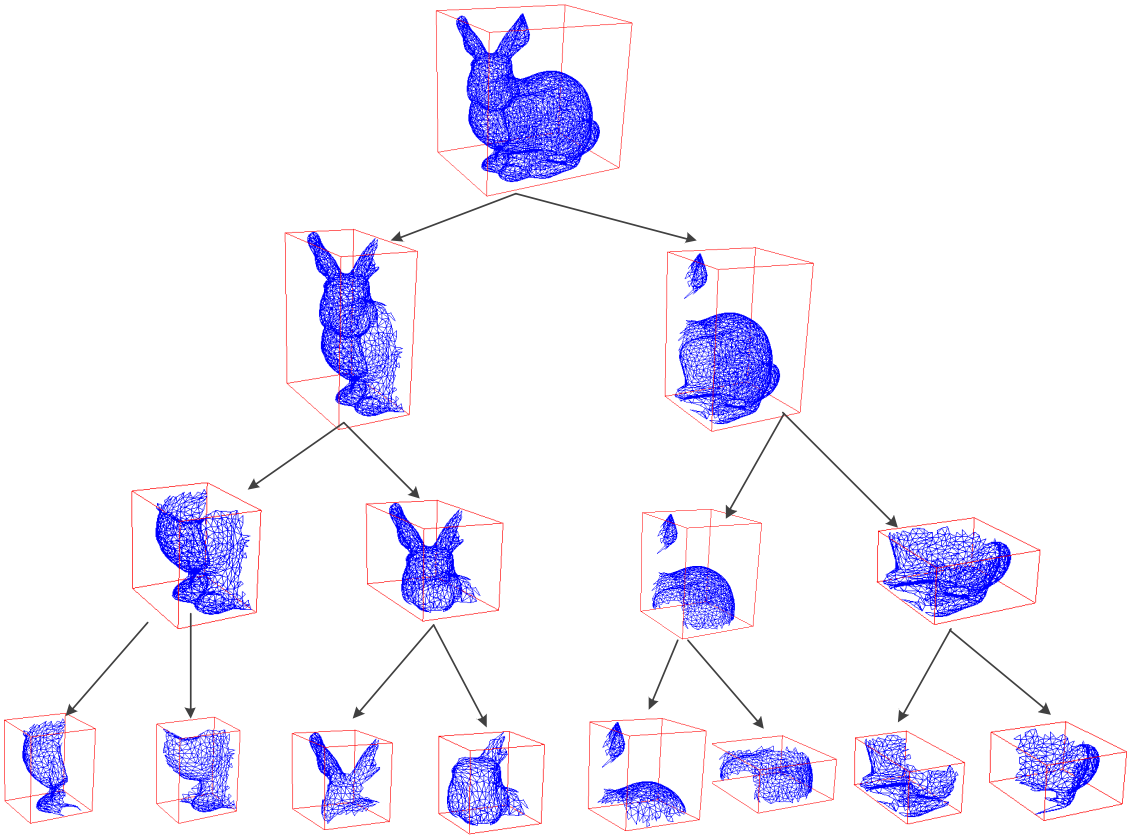


图 3.7　Bunny 模型的 AABB 树形结构 (部分)

整体的碰撞检测算法流程图如图 3.8 所示，首先扫描所有输入模型点集，对

每个模型计算其包围盒，然后计算要参与碰撞检测的模型的包围盒对进行相交测试，假设参与碰撞检测的模型的包围盒相交的对数为 n_1，再对这 n_1 对模型计算其 *k*-CBP 并进行初始化，如构造 *k*-CBP 的 AABB 树、初始化 GJK 算法，并计算 *k*-CBP 是否相交，此步骤后剩余模型对数为 n_2，最后再对这 n_2 对模型进行构造 AABB 树进而进行相交测试，真实模型相交对数为 n_3。整个流程中，包围盒的命中率 $r(\text{Box}) = n_3/n_1$，*k*-CBP 的命中率 $r(k\text{-CBP}) = n_3/n_2$。

图 3.8　基于 *k*-CBP 的碰撞检测算法流程图

3.3.2　运动场景中的碰撞检测算法

当在运动场景中的模型进行碰撞检测时，模型中的点坐标会更新，一种方法是重新计算模型中的所有点再以相同的流程和步骤对模型做碰撞检测，但当模型点数量较大时，耗时太久因此该方法不可取；另外一种算法是仍然利用静止场景中的 AABB 树形结构，仅重新计算将要碰撞的节点的坐标值，进而进行相交检测。节点包围盒坐标在运动过程中发生变化，精确的包围盒是重新计算该节点包含原始模型的点在变化后的点坐标值的包围盒，本文利用一种近似算法即仅对包围盒的 8 个顶点进行转换然后计算这 8 个顶点的包围盒，用这个近似包围盒进行遍历剪枝，当到叶子节点后，再重新计算网格模型的点的新坐标值用同样的方法进行相交检测。

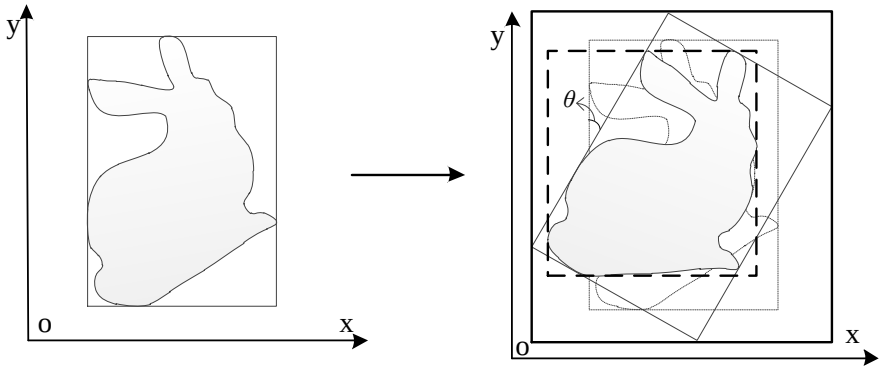


图 3.9　AABB 更新策略示意图

具体而言，以二维 Bunny 模型为例，如图 3.9 所示，当模型顺时针旋转 θ 后，本文利用近似算法得到新的包围盒如图中粗实线所示，而模型实际的包围盒为粗虚线所示，这样的结果会使得包围盒更不紧致，空白空间变得更大，但速度更快，

若包围盒相交后，会再用其 *k*-CBP 进行过滤，*k*-CBP 的顶点是精确顶点因而能更有效地进行过滤。

模型围绕任意过原点的轴 $\boldsymbol{n}(x,y,z)$ 旋转任意角度 θ 的变换矩阵如式 (3-4) 所示，其中 $\lambda = 1-\cos\theta$，详细的推导过程可以参考文献 [66]。

$$\begin{cases} \boldsymbol{R}(\boldsymbol{n},\theta) & = \cos\theta \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + \lambda \begin{pmatrix} \boldsymbol{n}_x^2 & \boldsymbol{n}_x\boldsymbol{n}_y & \boldsymbol{n}_x\boldsymbol{n}_z \\ \boldsymbol{n}_x\boldsymbol{n}_y & \boldsymbol{n}_y^2 & \boldsymbol{n}_y\boldsymbol{n}_z \\ \boldsymbol{n}_x\boldsymbol{n}_z & \boldsymbol{n}_y\boldsymbol{n}_z & \boldsymbol{n}_z^2 \end{pmatrix} + \sin\theta \begin{pmatrix} 0 & \boldsymbol{n}_z & -\boldsymbol{n}_y \\ -\boldsymbol{n}_z & 0 & \boldsymbol{n}_x \\ \boldsymbol{n}_y & -\boldsymbol{n}_x & 0 \end{pmatrix} \\ & = \begin{pmatrix} \cos\theta + \boldsymbol{n}_x^2\lambda & \boldsymbol{n}_x\boldsymbol{n}_y\lambda + \boldsymbol{n}_z\sin\theta & \boldsymbol{n}_x\boldsymbol{n}_z\lambda - \boldsymbol{n}_y\sin\theta \\ \boldsymbol{n}_x\boldsymbol{n}_y\lambda - \boldsymbol{n}_z\sin\theta & \cos\theta + \lambda\boldsymbol{n}_y^2 & \boldsymbol{n}_y\boldsymbol{n}_z\lambda + \boldsymbol{n}_x\sin\theta \\ \boldsymbol{n}_x\boldsymbol{n}_z\lambda + \boldsymbol{n}_y\sin\theta & \boldsymbol{n}_y\boldsymbol{n}_z\lambda - \boldsymbol{n}_x\sin\theta & \cos\theta + \boldsymbol{n}_z^2\lambda \end{pmatrix} \end{cases} \tag{3-4}$$

当模型平移时，点 $\boldsymbol{P}(x,y,z)$ 平移 $\boldsymbol{t}(t_x,t_y,t_z)$ 后的点坐标为 $\boldsymbol{P}'(x+t_x,y+t_y,z+t_z)$，可用 4×4 的变换矩阵 $\boldsymbol{T}$ 表示，即

$$\boldsymbol{T}(\boldsymbol{t}) = \begin{pmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3-5}$$

因此当模型平移旋转时产生的变换矩阵 $\boldsymbol{M}$ 为

$$\boldsymbol{M} = \boldsymbol{R}(\boldsymbol{n},\theta)\cdot\boldsymbol{T}(\boldsymbol{t}), \tag{3-6}$$

AABB 节点包围盒更新后的算法如算法 11 所示，算法输入为平移向量 $\boldsymbol{t}$ 及旋转方向 $\boldsymbol{n}$ 和角度 θ 或者直接传入变化矩阵 $\boldsymbol{M}$ 即可。

算法 11 AABB 节点包围盒更新算法

输入: AABB 包围盒 *box*，平移旋转变换矩阵 $\boldsymbol{M}$
输出: 变换之后的包围盒 *box*′

```
1: function TransformBox(box, M)
2:     vertices ← GetAABBVertices(box)
3:     box′ ← ∅
4:     for all v ∈ vertices do  // 遍历 box 的 8 个顶点
5:         v′ = M · v  // 计算变换后的点坐标
6:         UPDATE(box′, v′)  // 根据变换后的顶点 v 更新 box′
7:     end for
8:     return box′
9: end function
```

运动场景中模型的碰撞检测算法如算法 12 所示，假设参与碰撞检测的两个模型中第一个运动且变换矩阵为 $\boldsymbol{M}$，第二个静止，两个模型都运动时，可视第二个

算法 12 运动场景中基于 AABB 树碰撞检测算法

输入: 两个模型 AABB 树的根节点 $root_0, root_1$ 及 平移旋转变换矩阵 $\boldsymbol{M}$
输出: 模型是否相交

```
1: function MovingTraverseDetection(root_0, root_1, M)
2:     mBox ← TransformBox(M, root_0.box) // 按照算法 11 计算运动后的包围盒
3:     c ← Intersect(mBox, root_1.box)
4:     if c = False then
5:         return False // 包围盒不相交，直接返回 False
6:     end if
7:     if root_0.IsLeaf() then
8:         if root_1.IsLeaf() then // 两个叶子节点的原始几何进行相交测试
9:             for all p_1 ∈ root_0.primitives do
10:                for all p_2 ∈ root_1.primitives do
11:                    return Intersect(M, p_1, p_2) // 将 p_1 应用于变换矩阵 M 后采用算法3.2中
   进行三角网格相交测试
12:                end for
13:            end for
14:        else
15:            if MovingTraverseDetection(root_0, root_1.left) or
                  MovingTraverseDetection(root_0, root_1.right) then
16:                return True
17:            end if
18:        end if
19:    else
20:        if root_1.IsLeaf() then
21:            if MovingTraverseDetection(root_0.left, root_1) or
                  MovingTraverseDetection(root_0.right, root_1) then
22:                return True
23:            end if
24:        else // 两个节点都有孩子节点
25:            if MovingTraverseDetection(root_0.left, root_1.left) or
                  MovingTraverseDetection(root_0.left, root_1.right) or
                  MovingTraverseDetection(root_0.right, root_1.left) or
                  MovingTraverseDetection(root_0.right, root_1.right) then
26:                return True
27:            end if
28:        end if
29:    end if
30:    return False
31: end function
```

模型相对静止，将 $\boldsymbol{M}$ 设置为第一个相对于第二个模型的相对变换矩阵。

算法 12 是一个递归算法，从 AABB 树的根节点起向底层叶子节点进行深度优先遍历，当检测到运动后的模型的某两个叶子节点的包围盒相交时，遍历其三角网格的相交检测算法，在实际实现过程中，本文的叶子节点仅含一个三角网格。运动模型的三角网格应用变换矩阵得到一个新的三角网格，同样用 3.2 的算法对新的三角网格进行相交测试。除了用该递归算法外，也可对算法 8 进行稍许改动（只需改变包围盒检测和三角网格检测的函数）的迭代算法，此处不再赘述。

在基于 GJK 的 *k*-CBP 碰撞检测算法中，只需要改变算法 9 中 $Support$ 子过程，将变换矩阵应用到原始顶点进行计算得到新的支持点即可，而在连续运动的模型碰撞检测过程中，利用爬山法可以将支持点的搜索优化到常数时间复杂度[47]。

3.4　实验结果及分析

本章的实验主要分为 4 个部分，首先从 *k*-CBP 应用与碰撞检测的有效性上与仅用包围盒过滤算法进行对比，然后将 *k*-CBP 和其他主要几个包围体从构造时间、碰撞检测命中率上进行比较，最后分别从静止场景和运动场景中与文献 [14] 中的基于 *k*-DOP 树的算法对比。CollDet ① 是 Gabriel Zachmann 等人实现的一个碰撞检测库[14]，内含基于 *k*-DOP 树的实现，在与基于 *k*-DOP 树算法进行的实验对比结果中，本文在实验过程中均只通过 CPU 进行计算，针对碰撞检测实验环境中多个相同的实验模型，*k*-CBP 与 *k*-DOP 的构造均采用重新扫描扫描点集构造。

本章实验所采用的模型如表 3.1 所示，模型点集大小从 2.5k 到 225k 不等，三角面片的数量从近 5k 到 264k 不等，除了第 2 章的部分模型外，本章还用到了 CollDet 库所提供的部分模型② 。

表 3.1　实验模型数据

Model	Points(个)	Triangles(个)
Bunny	2503	4968
Apple	8118	9380
HappyBuddha	24810	50000
Dinosaur	40277	80554
Hand	64349	128314
Dragon	87257	174281
Alice	224291	264046

3.4.1　与包围盒过滤算法对比

下面的实验通过与包围盒过滤算法的对比来说明本文算法应用于碰撞检测的有效性，实验通过生成不同数量的模型（模型位置和旋转角度随机生成），碰撞检测时首先判断包围盒是否相交，然后判断凸包围多面体是否相交，最后再判断实际模型是否相交。该实验案例中模型和凸包围多面体是否相交都采用了普

① 其源码和文档均可通过 http://cgvr.cs.uni-bremen.de/research/colldet/ 下载得到。

② 实验中用到的 HappyBuddha、Dragon 和 Hand 等模型可从 http://cgvr.cs.uni-bremen.de/research/colldet_benchmark/下载。

通 AABB 树的方式进行判断，从如表 3.2 的实验结果可看出含有凸包多围体的模型之间的碰撞检测算法能显著提高整体应用的效率.

表 3.2　*k*-CBP 和包围盒应用于碰撞检测结果对比

n	CT(Box) (ms)	CT(16-CBP) (ms)	DT(Box) (ms)	DT(16-CBP) (ms)	*r*(Box) (%)	*r*(*k*-CBP) (%)	DP(Model) (对)
10	0.1	1.8	26.0	0.1	0.00	100.00	0
30	0.2	2.9	134.0	70.0	45.45	83.33	5
50	0.5	4.8	506.0	255.2	46.34	86.36	19
70	0.4	4.8	901.1	492.5	44.16	80.95	34
90	0.7	5.7	1324.0	734.7	41.82	73.02	46
100	0.7	7.8	1481.0	870.7	43.31	75.34	55
150	1.0	9.8	4153.1	2473.0	42.98	70.75	150
200	1.6	12.8	8049.3	4430.9	41.02	71.32	281

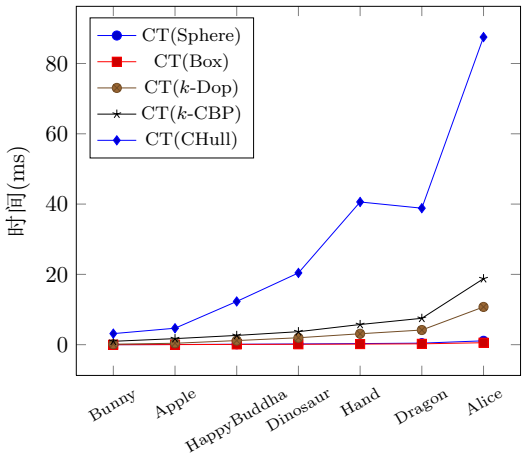
如表 3.2 所示，其中 *n* 表示场景中 Bunny 模型的数量，CT(Box)、CT(16-CBP) 分别表示模型包围盒的构造时间（Construction Time，简称 CT）和凸包围 16 面体的构造时间①，DT(Box)、DT(16-CBP) 分表表示用包围盒进行碰撞检测和利用凸包围 16 面体进行碰撞检测所耗费的时间（Detection Time，简称 DT），其中 *r*(Box)、*r*(16-CBP) 分别表示包围盒、16-CBP 的命中率（即用实际模型相交的数量除以包围体检测出来相交的数量），DP(Model) 检测到的模型实际相交的对数（Detection Pair，简称 DP），显然计算模型包围盒所耗费的时间要明显少于计算凸包围多面体的时间，但由于凸包围多面体比包围盒紧致，因而命中率比包围盒高，表中所示 16-CBP 比包围盒的命中率高 30% ~ 40%，能尽早排除更多本不相交的模型从而节省碰撞检测总时间，提高算法效率。该部分工作已经发表，详细内容见参考文献 [68]。

3.4.2　不同包围体实验对比

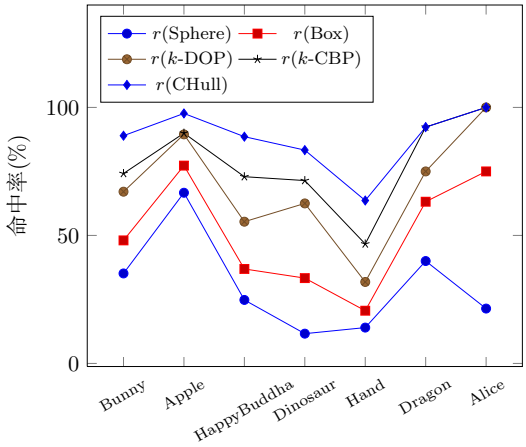
本文针对不同数量的模型分别用不同的包围体过滤参与碰撞检测，即对每个模型先构造模型的包围体，然后对包围体进行相交检测，包围体相交的模型最后再进行实际模型的相交检测，实际模型相交检测的算法为前文提到的基于 AABB 树的算法。

图 3.10 为在 2G 内存的限制下不同包围体应用于静止场景的碰撞检测实验结

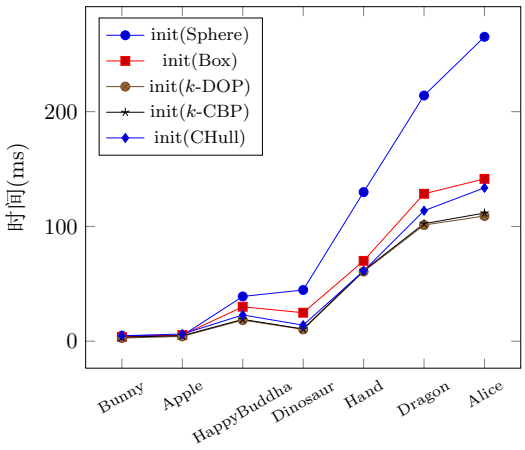
① 此处的时间为得到一个 *k*-CBP 后直接通过应用变换矩阵得到新的 *k*-CBP 总时间且是利用 GPU 搜索截面的时间，后文与 *k*-DOP 对比均采用 CPU 算法实现。



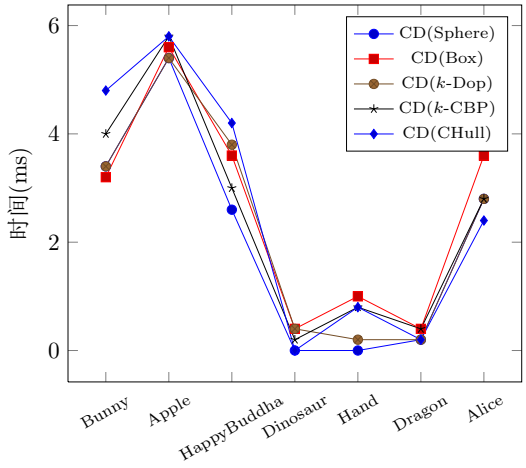
(a) 包围体构造时间对比



(b) 包围体命中率对比



(c) 包围体初始化时间对比



(d) 碰撞检测过程时间对比

图 3.10　不同包围体实验结果对比

果，其中 CT 表示单个模型包围体的平均构造时间，r 为包围体的命中率，Box、Sphere 和 CHull 分别是 AABB 包围体、Ritter[67] 算法计算出的包围球和 CGAL 计算的凸包，k-CBP 和 k-DOP 中的 k 值均为 16，且 k-DOP 构造过程中在此实验中没有计算其顶点，场景中随机放置的模型数量 Bunny、Apple 和 HappyBuddha 各为 100 个，Dinosaur 和 Hand 各 50 个，35 个 Dragon 和 25 个 Alice。从中可得，从包围体的构造时间上基本满足：凸包 > k-CBP > k-DOP > Sphere ≈ Box，碰撞检测中包围体的紧致程度和包围体的命中率满足正相关关系，与实验结果命中率保持一致，图示的实验结果表明各个包围体命中率基本满足：凸包 > k-CBP > k-DOP > Box > Sphere，在此实验中，一方面，命中率高的包围体可以减少不必要的真实模型之间的碰撞检测，并因此提高整体算法的效率；而另一方面命中率高的紧致包围体往往构造过程会更复杂，且包围体间的相交检测也更复杂。图 3.10(c) 中的 init 曲线所代

表初始化时间为包围体构造时间加上各个包围体相交检测后对相关模型进行构造 AABB 树的总时间，因为 Sphere 的命中率最低，因此需要构造 AABB 树的模型数量较多，因此初始化时间耗时更多，相应的凸包虽然其命中率高但其构造时间较长，因此其初始化时间也不是最少的，而 *k*-DOP 和 *k*-CBP 相比，虽然 *k*-CBP 命中率稍高但因其需要计算凸包围的的顶点因此构造时间稍长而 *k*-DOP 在此次实验中不需要计算顶点，因此综合来看二者初始化时间相差不大。图 3.10(d) 中碰撞检测过程中的时间对比，其中 *k*-CBP 和凸包之间的相交检测都是采用 GJK 算法，因为本实验采用的碰撞检测算法都是基于 AABB 树进行的，且包围体的碰撞检测只占用整个碰撞检测算法的一小部分，因此不同包围体在碰撞检测过程中耗时相差不大。

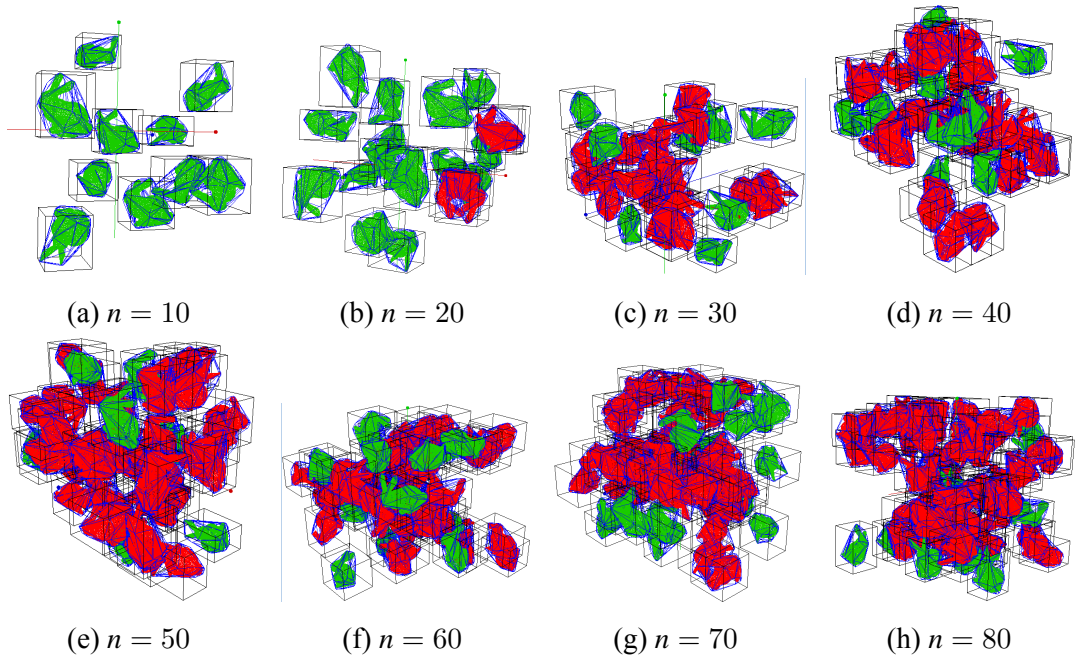
表 3.3　experiment model information

	Comparing File			Compared File		
No.	Name	Size(mb)	instances	Name	Size(mb)	instances
1	M7_R	0.591	11753	M7	0.425	8287
2	M8_A	1.040	24277	M8	1.257	26234
3	M9_R	2.519	42884	M9	3.511	68114
4	M10_A	9.817	215354	M10	4.364	91023

3.4.3　静止场景中与基于 *k*-DOP 树算法对比

在静止场景中的碰撞检测实验中，本文生成不同数量的模型，其位置和旋转角度随机生成（*k*-DOP 和 *k*-CBP 采用相同的随机生成的数据），图 3.11 为 Bunny 模型在模拟碰撞检测的实验示例。

表 3.4 为本文算法与 *k*-DOP 均采用 CollDet 中默认值 $k = 24$ 针对 Bunny 模型的实验结果。其中，n 表示场景中模型的数量，n(Box) 表示通过包围盒过滤后构造碰撞检测对的有效模型数量，$n - n$(Box) 的值表示场景中模型的包围盒不与任何其他模型包围盒相交的模型数量，CT(*k*-DOP) 和 CT(*k*-CBP) 分别表示初始化的时间，CT(*k*-DOP) 包括计算模型包围盒和计算 *k*-DOP 树，CT(*k*-CBP) 包括计算模型包围盒，*k*-CBP 以及用于实际模型碰撞检测算法的 AABB 树，DT(*k*-DOP)、DT(*k*-CBP)(*AABB*) 和 DT(*k*-CBP)(*GJK*) 均表示碰撞检测所耗费的时间，DT(*k*-CBP)(*AABB*) 为通过第 3.1.1 节中介绍的 AABB 方法对 *k*-CBP 进行相交检测（包括构造 *k*-CBP 的 AABB 树所耗费的时间），而 DT(*k*-CBP)(*GJK*) 表示通过第 3.1.2 节中介绍的 GJK 方法对 *k*-CBP 进行相交检测，相应的时间都包括 *k*-CBP 相



(a) $n = 10$　(b) $n = 20$　(c) $n = 30$　(d) $n = 40$

(e) $n = 50$　(f) $n = 60$　(g) $n = 70$　(h) $n = 80$

图 3.11　静止场景下 Bunny 模型碰撞检测示例

交后对模型进行 AABB 树相交检测时间。

表 3.4　静止场景下本文算法与基于 *k*-DOP 树结果对比（Bunny）

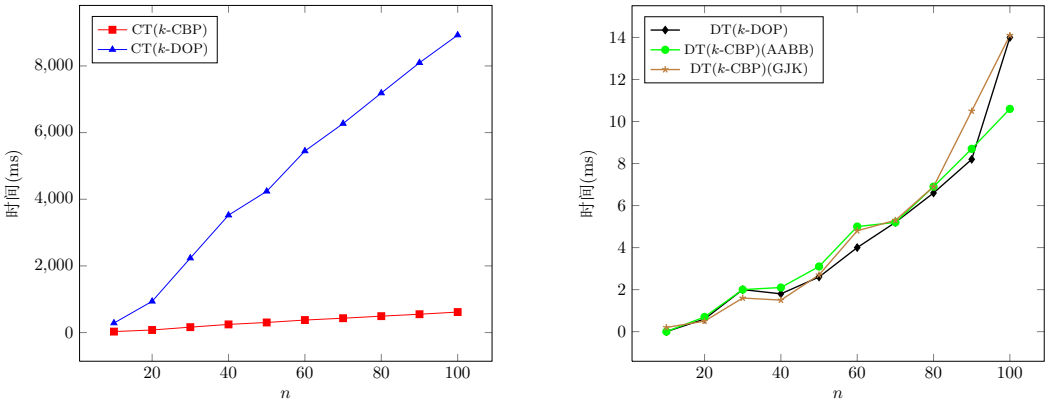
n	*n*(Box)	CT(*k*-DOP) (ms)	CT(*k*-CBP) (ms)	DT(*k*-DOP) (ms)	DT(*k*-CBP) (AABB) (ms)	DT(*k*-CBP) (GJK) (ms)
10	3	285.00	29.30	0.01	0.01	0.20
20	10	937.20	77.30	0.60	0.70	0.50
30	24	2233.60	162.20	2.00	2.00	1.60
40	38	3522.20	244.70	1.80	2.10	1.50
50	46	4238.20	302.90	2.60	3.10	2.70
60	59	5447.60	375.80	4.00	5.00	4.80
70	68	6271.00	430.40	5.20	5.20	5.30
80	78	7186.20	492.10	6.60	6.90	6.90
90	88	8096.20	550.30	8.20	8.70	10.50
100	97	8925.80	615.00	14.00	10.60	14.10

表 3.5 记录了更加详细的实验结果，其中 CT(*k*-CBP) 和 CT(AABB) 分别表示本文算法中构造 *k*-CBP 的时间和构造用于具体模型碰撞检测算法的 AABB 树的时间，DP(Box) 和 DP(Model) 表示包围盒相交的对数和实际模型碰撞的对数，这两组数据与 *k*-DOP 算法中相同，DP(*k*-CBP) 表示 *k*-CBP 相交的对数，从此表可以看出，在初始化过程中，构造模型的 AABB 树占用了较大的比重。结合二表可知，在前期初始化碰撞检测环境中，本文算法优势明显，当模型为 100 时，*k*-DOP 算

法构造 k-DOP 树需要近 9 秒的时间，而本文算法不到 1 秒，初始化环境后，进入碰撞检测环节，本文算法较 k-DOP 相差不大，而本文提出的基于 AABB 树算法和基于 GJK 算法中，对于较少三角网格的 Bunny 模型而言，AABB 算法在静态环境中较 GJK 更优，而从后文第 3.4.4 节也可以看出基于 GJK 算法在动态碰撞检测的环境中更优。

表 3.5　静止场景下本文算法实验结果（Bunny）

n	CT(k-CBP)(ms)	CT(AABB)(ms)	DP(Box)(对)	DP(k-CBP)(对)	DP(Model)(对)
10	13.60	16.60	2	2	0
20	23.80	53.40	10	4	2
30	35.00	128.00	42	31	23
40	42.40	199.40	42	21	19
50	51.00	244.20	66	43	35
60	58.00	313.40	121	62	49
70	66.80	360.20	144	77	61
80	72.20	412.80	167	101	85
90	81.00	463.80	261	119	84
100	92.00	514.20	335	200	161

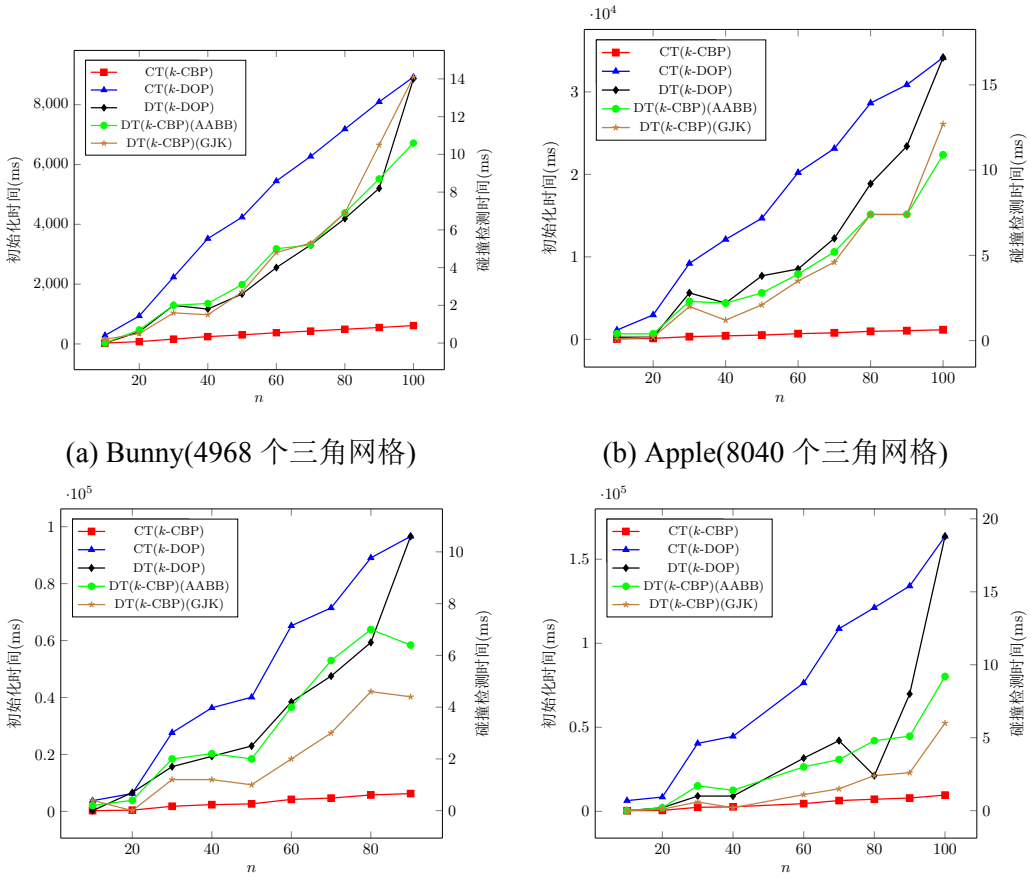


(a) 初始化时间对比　　(b) 碰撞检测时间对比

图 3.12　静止场景下本文算法与基于 k-DOP 树算法实验结果对比（Bunny）

图 3.12 为本文的两种算法和文献 [14] 中基于 k-DOP 树实现的实验结果的曲线展示图，从图 3.12(a) 可以看出，初始化构造 k-DOP 树和本文算法初始化所耗费时间与参与碰撞检测的模型的数量近线性关系，但明显本文算法所耗费时间更少，图 3.12(b) 为初始化碰撞检测环境后进行碰撞检测所耗费的时间，本文的两种算法和基于 k-DOP 的算法相比，碰撞检测所耗费的时间相差不大，总体来说针

对 Bunny 模型而言，基于 k-CBP 结合 AABB 树算法所耗费时间稍微较少，但三者相差不大，整体相差在 1 ~ 4 毫秒范围内。



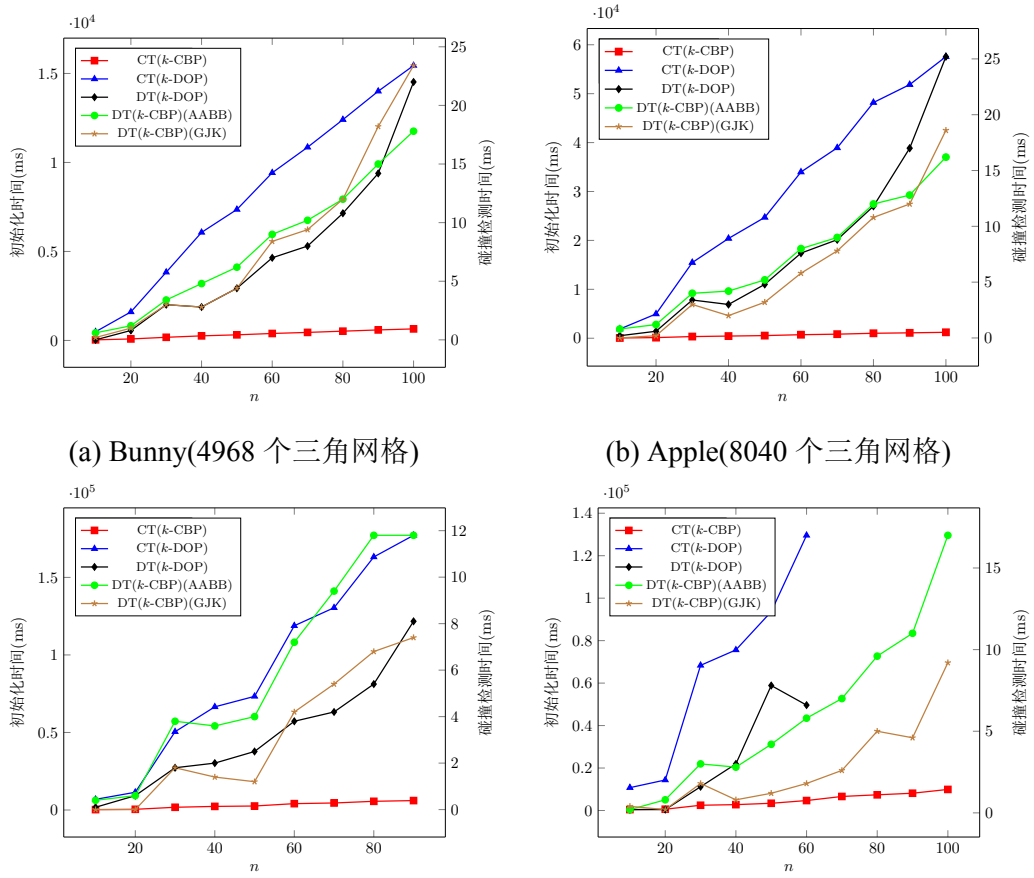
(a) Bunny(4968 个三角网格)　(b) Apple(8040 个三角网格)

(c) HappyBuddha(50000 个三角网格)　(d) Dinosaur(80554 个三角网格)

图 3.13　静止场景下本文算法与基于 k-DOP 树实验结果对比（$k = 24$）

图 3.13 记录了更多模型的实验结果，其中曲线三角形和正方形记录的是碰撞检测环境中的初始化时间，另外三条曲线分别是本文的两种算法和基于 k-DOP 的树算法碰撞检测所耗费的时间，左边纵坐标刻度为初始化时间，右边纵坐标刻度为碰撞检测的时间。从中可得，在静态碰撞检测实验中，本文算法与基于 k-DOP 树的算法相比，在初始化环境过程中占用更少的时间，而在碰撞检测过程中针对不同的模型可能有不同的结果。从整体来看，本文算法优于基于 k-DOP 树的算法，平均而言初始化过程的时间效率提高了 8 倍，碰撞检测过程的时间效率是基于 k-DOP 树算法的 0.8 ~ 3.2 倍。

对于更大的 k 值，构造层次结构的 k-DOP 树时会耗费更多的时间和存储空间，在模型较大且模型数量较多时，以文献 [14] 中最大的 k 值为 46 为例，32 位应用程序已经满足不了构造 70 个 Dinosaur 模型 的 k-DOP 树，且构造时间也更久，图 3.14 为 k-DOP 实现中的最大值 $k = 46$ 的实验结果。

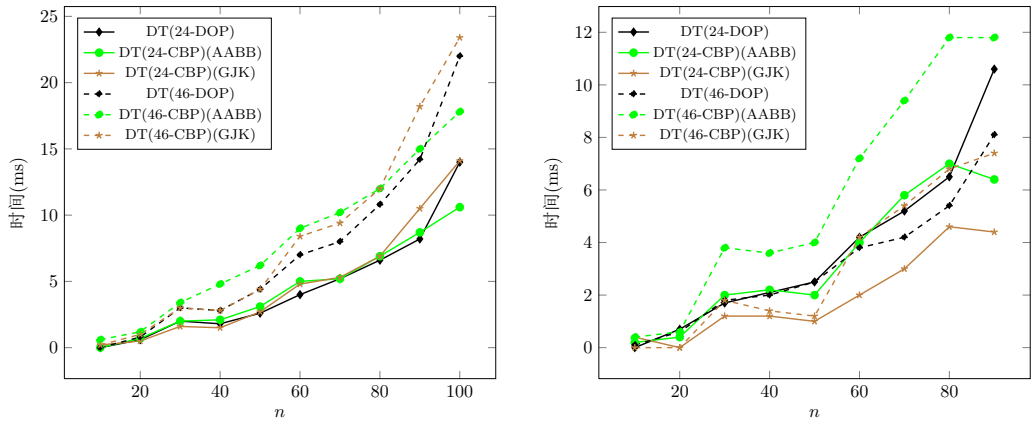


(a) Bunny(4968 个三角网格)　　(b) Apple(8040 个三角网格)

(c) HappyBuddha(50000 个三角网格)　　(d) Dinosaur(80554 个三角网格)

图 3.14　静止场景下本文算法与基于 k-DOP 树算法实验结果对比（$k = 46$）

更大的 k 值不一定能够使碰撞检测的效率提高，这与具体的碰撞检测环境有关，以 Bunny 模型和 HappyBuddha 模型为例，如图 3.15 为 $k = 24$ 和 $k = 46$ 的结果对比，其中，虚线的结果为 $k = 46$ 的结果，实线为 $k = 24$ 的结果，Bunny 模型中三种算法在 $k = 24$ 时耗时更少，而 HappyBuddha 模型在基于 k-DOP 树的算法中，$k = 46$ 的结果稍微优于 $k = 24$ 的结果，而基于 k-CBP 的两种算法中在 $k = 24$ 时碰撞检测所耗费的时间更少。因此在实际应用环境中应该根据具体的模型及具体的碰撞检测环境选择不同的 k 值。

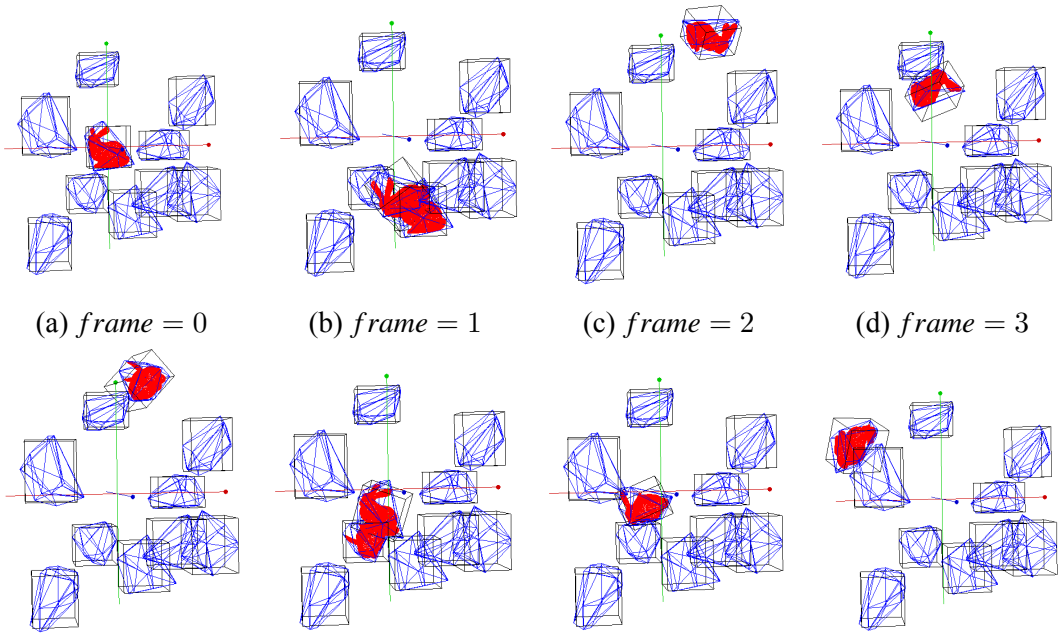


(a) Bunny(4968 个三角网格)　　(b) HappyBuddha(50000 个三角网格)

图 3.15　静止场景下不同 k 值实验结果对比

3.4.4　运动场景中与基于 k-DOP 树算法对比

对于运动场景中的碰撞检测实验，本文采取随机读入单个模型，并随机生成多个平移旋转变换构造多个模型，然后令其中一个模型产生随机平移旋转变换进行运动，检测该运动模型与其他所有的模型进行碰撞检测，图 3.16 为随机产生的 10 个模型，并让其中一个随机运动 7 步的示意图，示例中，对于没有和运动模型产生碰撞的模型，图中仅显示了其包围盒和 k-CBP。



(a) $frame = 0$　(b) $frame = 1$　(c) $frame = 2$　(d) $frame = 3$

(e) $frame = 4$　(f) $frame = 5$　(g) $frame = 6$　(h) $frame = 7$

图 3.16　运动场景下 Bunny 模型碰撞检测示例

表 3.6 为本文算法和 k-DOP 算法均采用 CollDet 实现中 $k = 24$ 默认值的初始

化时间对比，init(*k*-CBP) 所代表的时间表示构造 *k*-CBP 及模型的 AABB 树所耗费的时间，init(*k*-DOP) 为构造 *k*-DOP 所耗费的时间，*n* 的数量为动态碰撞检测场景中模型的总数量，从中可知本文在初始化时间上快 8 倍以上，同时对 10 个 Hand 模型进行初始化，*k*-DOP 算法需要近 40 秒的时间，而本文算法仅需 4.5 秒。

表 3.6　本文算法与基于 *k*-DOP 树算法初始化时间对比（$k = 24$）

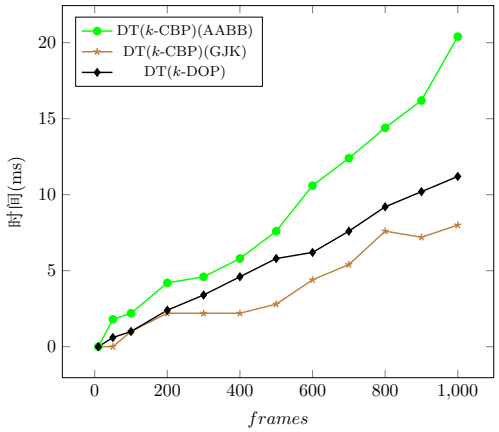
Model	$n = 2$		$n = 10$	
	init(*k*-CBP)(ms)	init(*k*-DOP)(ms)	init(*k*-CBP)(ms)	init(*k*-DOP)(ms)
Bunny	17.63	189.78	116.59	926.10
Apple	30.87	749.48	226.20	3732.43
HappyBuddha	160.58	2557.27	1361.57	12663.13
Dinosaur	250.55	4281.15	2230.88	21493.08
Hand	506.98	7926.88	4478.18	39538.70

在运动环境中，本文算法与 *k*-DOP 算法初始化过程与在静态环境中一样，*k*-DOP 需要构造 *k*-DOP 树也因此耗费很多时间，本文需要构造 AABB 树，也是初始化过程中占用时间较多的步骤。图 3.17 为运动场景下本文算法与 *k*-DOP 树的实验结果对比，其中凸包围多面体面数 $k = 24$。横坐标表示其中一个模型运动的次数 $frames$，纵坐标表示随机运动 $frames$ 次碰撞检测所耗费的时间，星形所代表曲线为前文提出的基于 GJK 算法实现，菱形所代表的曲线为基于 *k*-DOP 树的实现，圆形所代表的曲线为前文提出的基于 AABB 树实现。本文基于 *k*-CBP 的两种碰撞检测算法步调一致，因为两种算法在 *k*-CBP 相交后，都是利用 AABB 树进行对真实模型的碰撞检测，算法主要依赖与 AABB 包围盒节点之间的相交检测。

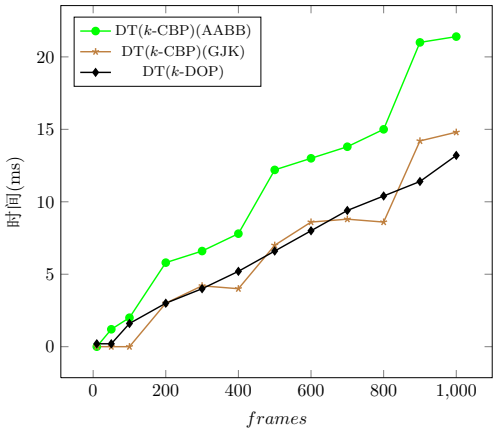
在 Bunny 模型中，本文基于 GJK 算法耗时最少，基于 *k*-DOP 算法耗时最久，本文基于 AABB 算法居中，总体趋势为 $frames$ 越大，耗时也越久，表 3.7 为 2 个 Bunny 模型在随机运动 $frames$ 步和 *k*-DOP 算法的比较结果，当真实模型相交数量较多时，三种算法所耗费的时间也更多。

随着三角网格增多，三种算法所耗费时间均有所增加，总体而言，基于 *k*-DOP 的算法比较稳定，而本文算法随着模型不同有一定波动。

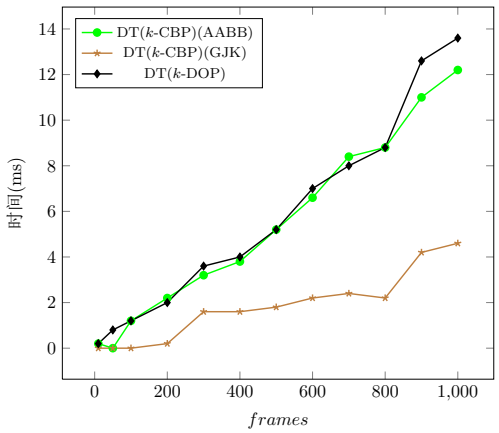
当模型数量 *n* 的值增大时，三种算法整体趋势仍能保持一致，如图 3.18 所示为相同 *k* 值，相同运动路径（包括平移距离旋转角度以及运动步数）下 10 个模型下的碰撞检测结果。从中可看出，当模型三角网格数量较少时，本文基于 *k*-CBP 结合 AABB 树的算法耗时最多，而随着三角网格数量增大，此算法和基于 *k*-DOP 树的算法耗时相差不大，但都多余本文基于 *k*-CBP 结合 GJK 的算法。整体来看，在运动场景的碰撞检测环境中，本文基于 *k*-CBP 结合 GJK 的算法在一定程度上有



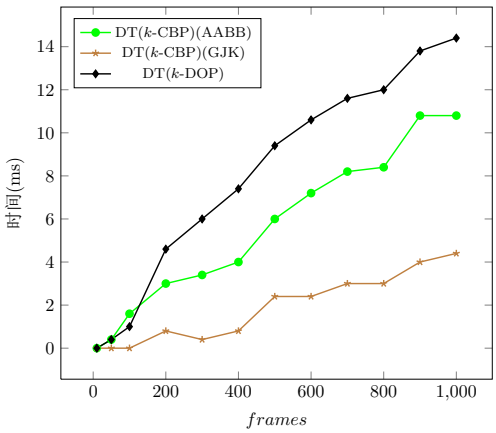
(a) Bunny(4968 个三角网格)



(b) HappyBuddha(50000 个三角网格)



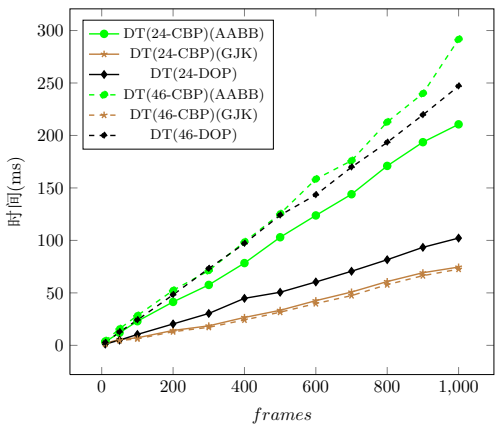
(c) Dinosaur(80554 个三角网格)



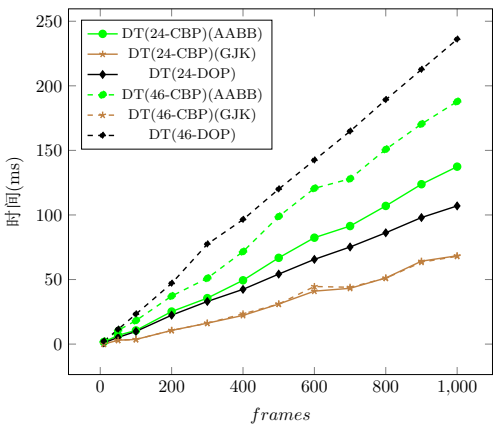
(d) Hand(128314 个三角网格)

图 3.17　运动场景下本文算法与基于 k-DOP 树实验结果对比（$k = 24, n = 2$）

一定优势，从实验结果来看平均而言是基于 k-DOP 树算法的 0.8 ~ 5.6 倍，而基于 k-DOP 算法相对较稳定，但其初始化时间很长。



(a) Bunny(4968 个三角网格)



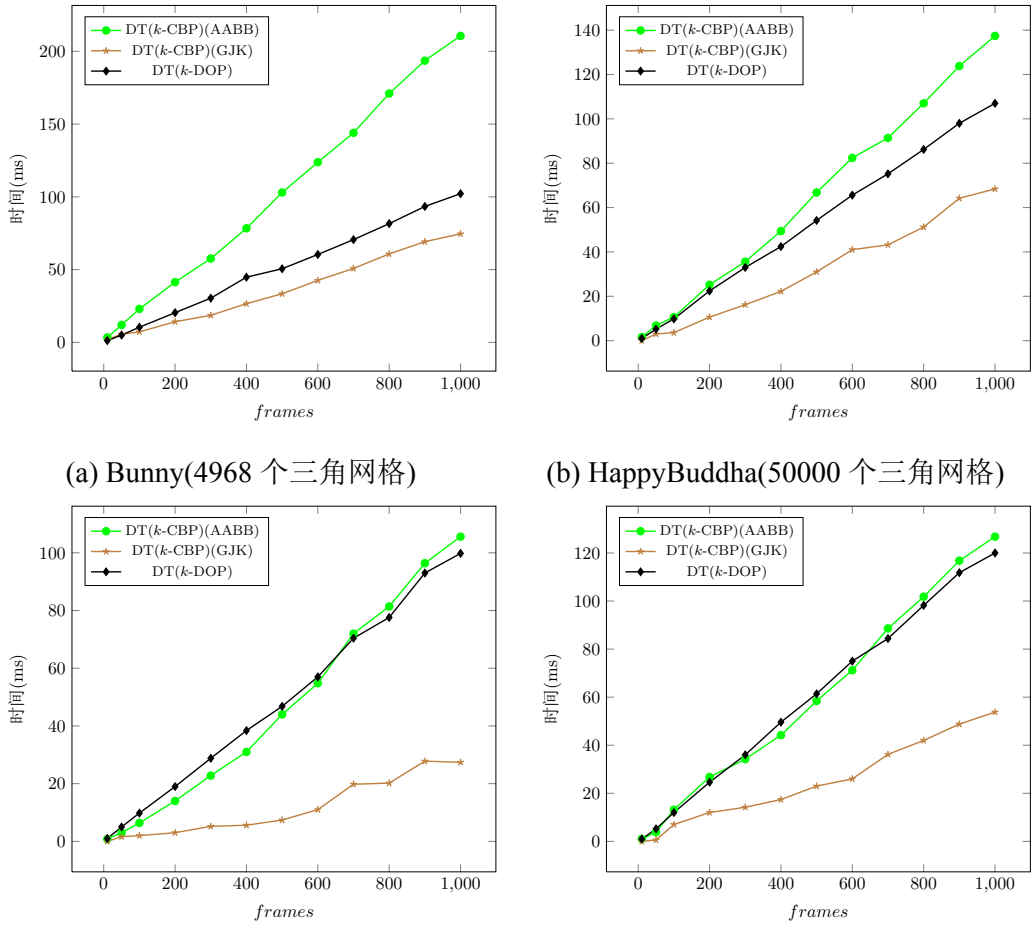
(b) HappyBuddha(50000 个三角网格)

图 3.19　运动场景下不同 k 值实验结果对比

表 3.7　运动场景下本文算法与 k-DOP 实验结果（Bunny）

$frames$	DT(k-CBP)(AABB) (ms)	DT(k-DOP) (ms)	DT(k-CBP)(GJK) (ms)	DP(k-CBP) (对)	DP(Model) (对)
10	0.00	0.00	0.00	0	0
50	1.80	0.60	0.00	1	1
100	2.20	1.00	1.00	3	2
200	4.20	2.40	2.20	7	6
300	4.60	3.40	2.20	9	8
400	5.80	4.60	2.20	12	10
500	7.60	5.80	2.80	14	12
600	10.60	6.20	4.40	16	13
700	12.40	7.60	5.40	19	15
800	14.40	9.20	7.60	21	17
900	16.20	10.20	7.20	25	21
1000	20.40	11.20	8.00	26	21

与在静止场景的碰撞检测环境一样，在运动场景的碰撞检测环境中，更大的 k 值也不一定能够使碰撞检测的效率提高，仍以 Bunny 模型和 HappyBuddha 模型为例（模型数量 $n = 10$），如图 3.19 为 $k = 24$ 和 $k = 46$ 的结果对比，其中，虚线的结果为 $k = 46$ 的结果，实线为 $k = 24$ 的结果，实验结果表明本文基于 k-CBP 的算法在运动场景中二者相差不大，而另外两种算法在 $k = 24$ 时碰撞检测所耗费的时间更少。因此在实际应用环境中应该根据具体的碰撞检测环境选择不同的 k 值。



(a) Bunny(4968 个三角网格)　(b) HappyBuddha(50000 个三角网格)

(c) Dinosaur(80554 个三角网格)　(d) Hand(128314 个三角网格)

图 3.18　运动场景下本文算法与 k-DOP 实验结果对比（$k = 24, n = 10$）

3.5　本章小结

本章主要介绍了基于 k-CBP 的碰撞检测算法，该算法主要利用包围盒和 k-CBP 对模型进行过滤，尽早排除不可能相交的模型。本文利用基于 AABB 树的方法和 GJK 的算法进行 k-CBP 之间的相交检测，在模型的 k-CBP 相交后，继续利用模型的 AABB 树对原始模型进行碰撞检测。本章基于 k-CBP 的碰撞检测实验从三个角度进行，一方面与仅用包围盒过滤算法进行对比，另外分别从静止和运动场景中与基于 k-DOP 的算法进行对比，实验结果表明平均而言本文算法初始化过程提高了至少 8 倍，在静止场景下，本文基于 k-CBP 结合 GJK 的算法是基于 k-DOP 树算法的 0.8 ~ 3.2 倍，而运动场景下是基于 k-DOP 树算法的 0.8 ~ 5.6 倍，因此本文算法能够有效加速碰撞检测过程。

第 4 章　总结与展望

4.1　总结

凸包围体在计算机图形学、计算机动画等领域中处于重要位置，常常作为原始模型的近似被广泛应用于光线跟踪、碰撞检测等算法中。常见的有沿坐标轴方向的包围盒，包围球、凸包等，包围体的紧致程度直接影响着相应算法的效率，对于一般不规则形状的模型，包围盒往往不够紧致而凸包很紧致但因其含有过多的面片数量导致算法复杂性增加。本文提出了一种构造凸包围 k 面体（k-CBP）的方法，该方法扫描输入模型的点集，然后构造一个粗糙的近似内凸包，利用这近似内凸包面片的法向通过 k-means 聚类算法生成指定 k 个构造凸包围多面体的法向；进而利用这些法向搜索原始模型点集中的切点构成凸包围多面体的截面；最后由这些截面求交构成凸包围多面体。在搜索截面的过程中，需要根据每个法向多次扫描点集，本文提出了两种方案进行并行加速，一种是基于 OpenGL 着色语言进行并行加速，该方法在点集较小的模型中利用深度缓存算法根据 OpenGL 渲染机制中深度裁剪自动提取切点，在点集较大的模型中利用乒乓技术能够有效对多次迭代数据进行缓冲进而能够较快获得切点；另一种方案是基于现代显卡较为通用的 CUDA 并行计算架构平台进行并行规约加速。实验结果证明本文提出的方法能够有效提高构造凸包围多面体的构造速度，与现有算法相比平均能够加速 3 ~ 8 倍。本文提出的 k-CBP 可根据不同应用场景选择不同参数 k 得到不同紧致程度的凸包围多面体。

本文提出的构造 k-CBP 方法较 k-DOP 而言能够得到更加紧致的凸包围多面体，对于一般不规则模型在 $20 \le k \le 40$ 时都能达到约 90% 的紧致程度，较 k-DOP 能够提高约 10% ~ 40% 的紧致程度。更加紧致的凸包围多面体能够使碰撞检测算法更有效率。本文提出一种基于 k-CBP 过滤的碰撞检测算法，该算法在包围盒相交后进行 k-CBP 的相交检测，若两个模型的 k-CBP 相交后再进行模型的相交测试。在进行模型的 k-CBP 相交测试中，本文利用了两种算法进行，一种是构造 k-CBP 的 AABB 树，将 k-CBP 看作是传统的网格模型进行相交检测判断，另外一种是利用 GJK 算法对 k-CBP 进行相交检测判断，实验结果表明当模型点集较小时，基于 AABB 树的方案较优，而当模型点较大或运动场景的碰撞检测算法中，基于 GJK 算法较优。总体看来，本文提出的基于 k-CBP 的碰撞检测算法在应用与静态或运动场景的碰撞检测算法中，基于 AABB 树方法或基于 GJK 算法能够

保持同步，基于 *k*-DOP 在初始化步骤中需要计算层次结构 *k*-DOP 耗时太久，因此本文算法在静态碰撞检测环境中都优于基于 *k*-DOP 的方法；在运动场景的碰撞检测算法中，排除初始化时间，较基于 *k*-DOP 的碰撞检测算法相比，本文算法还不够稳定，因为在 *k*-CBP 相交后，模型进一步相交检测须依赖于基于 AABB 树的碰撞检测算法，在真实模型相交的情况下，基于 *k*-DOP 树的算法优于本文采用基于 AABB 树的算法。从整体角度来看，本文算法在碰撞检测初始化过程效率上提高了至少 8 倍，而静止场景的碰撞检测过程中本文算法是基于 *k*-DOP 树的算法的 0.8 ~ 3.2 倍而在运动场景中是基于 *k*-DOP 树的算法的 0.8 ~ 5.6 倍。

4.2　展望

凸包围多面体在计算机辅助造型设计、计算机图形学等领域里有多种应用，碰撞检测是最其中重要的应用之一。本文提出的基于 *k*-CBP 算法在真实模型相交时需要用模型 AABB 树进行相交检测，因此算法依赖于模型 AABB 树的划分和遍历，在未来的工作中可以考虑如何摆脱对 AABB 树的依赖，通过快速构造的 *k*-CBP 在碰撞检测过程中加快过滤过程，或者可考虑将此 *k*-CBP 应用于近似碰撞检测算法中，用模型的多层次结构的 *k*-CBP 树替换精确的模型。同时在未来工作可考虑如何将 *k*-CBP 应用于如机器人抓取、路径规划等其他应用领域中。

参考文献

[1] 邓俊辉. 计算几何-算法与应用. 北京: 清华大学出版社, 2005.

[2] Bergen G v d. Efficient collision detection of complex deformable models using aabb trees. Journal of Graphics Tools, 1997, 2(4):1–13.

[3] Ericson C. Real-time collision detection. San Francisco, CA: Morgan Kaufmann Publishers, 2005.

[4] Gottschalk S, Lin M C, Manocha D. Obbtree: a hierarchical structure for rapid interference detection. the 23rd annual conference on Computer graphics and interactive techniques. ACM, 1996. 171–180.

[5] O'Rourke J. Finding minimal enclosing boxes. International journal of computer & information sciences, 1985, 14(3):183–199.

[6] Barequet G, Har-Peled S. Efficiently approximating the minimum-volume bounding box of a point set in three dimensions. Journal of Algorithms, 2001, 38(1):91–109.

[7] Chan C, Tan S. Determination of the minimum bounding box of an arbitrary solid: an iterative approach. Computers & Structures, 2001, 79(15):1433–1449.

[8] Welzl E. Smallest enclosing disks (balls and ellipsoids). Results and New Trends in Computer Science. Springer-Verlag, 1991. 359–370.

[9] Larsson T. Fast and tight fitting bounding spheres. The Annual Swedish Computer Graphics Association Conference(SIGRAD), 2008. 27–30.

[10] Kay T L, Kajiya J T. Ray tracing complex scenes. ACM SIGGRAPH Computer Graphics, volume 20. ACM, 1986. 269–278.

[11] Klosowski J T, Held M, Mitchell J S, et al. Efficient collision detection using bounding volume hierarchies of k-dops. IEEE Transactions on Visualization and Computer Graphics, 1998, 4(1):21–36.

[12] 魏迎梅, 王涌, 吴泉源, 等. 碰撞检测中的固定方向凸包包围盒的研究. 软件学报, 2001, 12(7):1056–1063.

[13] Zachmann G. Rapid collision detection by dynamically aligned dop-trees. Virtual Reality Annual International Symposium, Atlanta, Georgia: IEEE Computer Society, 1998. 90–97.

[14] Trenkel S, Weller R, Zachmann G. A benchmarking suite for static collision detection algorithms. International Conferences in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG), Plzen, Czech Republic: Union Agency, 2007.

[15] Hossain M Z, Amin M A. On constructing approximate convex hull. American Journal of Computational Mathematics, 2013, 3:11–17.

[16] Bentley J L, Preparata F P, Faust M G. Approximation algorithms for convex hulls. Communications of the ACM, 1982, 25(1):64–68.

[17] Kavan L, Kolingerova I, Zara J. Fast approximation of convex hull. the 2nd international conference on Advances in computer science and technology(ACST), 2006. 101–104.

[18] Zunié J. An outer approximation of the convex hull for finite grid point sets. Novi Sad Journal of Mathematics, 1992, 22(2):177–185.

[19] Crosnier A, Rossignac J. Tribox bounds for three-dimensional objects. Computers & Graphics, 1999, 23(3):429–437.

[20] Larsen E, Gottschalk S, Lin M C, et al. Fast proximity queries with swept sphere volumes. Technical report, Department of Computer Science, University of North Carolina, Chapel Hill, North Carolina, 1999.

[21] Krishnan S. Spherical shell: A higher order bounding volume for fast proximity queries. Third International Workshop on Algorithmic Foundations of Robotics, Houston, Texas: A. K. Peters Ltd, 1998.

[22] Guibas L J, Nguyen A, Zhang L. Zonotopes as bounding volumes. the fourteenth annual ACM-SIAM symposium on Discrete algorithms. Society for Industrial and Applied Mathematics, 2003. 803–812.

[23] Schömer E, Sellen J, Teichmann M, et al. Smallest enclosing cylinders. Algorithmica, 2000, 27(2):170–186.

[24] Held M. Erit - a collection of efficient and reliable intersection tests. Journal of Graphics Tools, 1997, 2(4):25–44.

[25] Wang W, Choi Y K, Chan B, et al. Efficient collision detection for moving ellipsoids using separating planes. Computing, 2004, 72(2):235–246.

[26] Assarsson U, Moller T. Optimized view frustum culling algorithms for bounding boxes. Journal of Graphics Tools, 2000, 5(1):9–22.

[27] Wald I, Boulos S, Shirley P. Ray tracing deformable scenes using dynamic bounding volume hierarchies. ACM Transactions on Graphics (TOG), 2007, 26(1):1–18.

[28] 王志强, 洪嘉振, 杨辉. 碰撞检测问题研究综述. 软件学报, 1999, 10(5):545–551.

[29] Larsson T, Akenine-Möller T. A dynamic bounding volume hierarchy for generalized collision detection. Computers & Graphics, 2006, 30(3):450–459.

[30] Madera F, Day A, Laycock S D. A hybrid bounding volume algorithm to detect collisions between deformable objects. Second International Conferences on Advances in Computer-Human Interactions(ACHI). IEEE, 2009. 136–141.

[31] Vogiannou A, Moustakas K, Tzovaras D, et al. Enhancing bounding volumes using support plane mappings for collision detection. Computer Graphics Forum, volume 29. Wiley Online Library, 2010. 1595–1604.

[32] Chang J W, Wang W, Kim M S. Efficient collision detection using a dual obb-sphere bounding volume hierarchy. Computer-Aided Design, 2010, 42(1):50–57.

[33] Tang M, Manocha D, Tong R. Fast continuous collision detection using deforming non-penetration filters. the 2010 ACM SIGGRAPH symposium on Interactive 3D Graphics and Games. ACM, 2010. 7–13.

[34] Zhigang F, Jianxun J, Jie X. Efficient collision detection using a dual k-dop-sphere bounding volume hierarchy. 2010 International Forum on Information Technology and Applications (IFITA), volume 3. IEEE, 2010. 185–189.

[35] Huebner K, Ruthotto S, Kragic D. Minimum volume bounding box decomposition for shape approximation in robot grasping. IEEE International Conference on Robotics and Automation(ICRA). IEEE, 2008. 1628–1633.

[36] Hubbard P M. Approximating polyhedra with spheres for time-critical collision detection. ACM Transactions on Graphics (TOG), 1996, 15(3):179–210.

[37] Lien J M, Amato N M. Approximate convex decomposition of polygons. Computational Geometry, 2006, 35(1):100–123.

[38] Lien J M, Amato N M. Approximate convex decomposition of polyhedra. the 2007 ACM symposium on Solid and physical modeling. ACM, 2007. 121–131.

[39] Lien J M, Amato N M. Approximate convex decomposition of polyhedra and its applications. Computer Aided Geometric Design, 2008, 25(7):503–522.

[40] Lien J M. Approximate convex decomposition and its applications[D]. College Station, Texas: Texas A&M University, 2006.

[41] Attene M, Mortara M, Spagnuolo M, et al. Hierarchical convex approximation of 3d shapes for fast region selection. Computer graphics forum, volume 27. Wiley Online Library, 2008. 1323–1332.

[42] Karlsson M, Winberg O, Larsson T. Parallel construction of bounding volumes. The Annual Swedish Computer Graphics Association Conference(SIGRAD), 2010. 65–69.

[43] Lauterbach C, Garland M, Sengupta S, et al. Fast bvh construction on gpus. Computer Graphics Forum, volume 28. Wiley Online Library, 2009. 375–384.

[44] Melax S. Dynamic Plane Shifting BSP Traversal. Graphics Interface, 2000. 213–220.

[45] Zeiller M, Purgathofer W, Gervautz M. Efficient collision detection for general csg objects. Computer Animation and Simulation. Springer Vienna, 1995. 66–79.

[46] Gilbert E G, Johnson D W, Keerthi S S. A fast procedure for computing the distance between complex objects in three-dimensional space. IEEE Journal of Robotics and Automation, 1988, 4(2):193–203.

[47] Bergen G v d. A fast and robust gjk implementation for collision detection of convex objects. Journal of Graphics Tools, 1999, 4(2):7–25.

[48] Lin M, Gottschalk S. Collision detection between geometric models: A survey. The Institute of Mathematics and its Applications(IMA) Conference on Mathematics of Surfaces, volume 1, 1998. 602–608.

[49] Brochu T, Edwards E, Bridson R. Efficient geometrically exact continuous collision detection. ACM Transactions on Graphics, 2012, 31(4):1–7.

[50] Wang H. Defending continuous collision detection against errors. ACM Transactions on Graphics (TOG), 2014, 33(4):122:1–123:10.

[51] Kaufman D M, Tamstorf R, Smith B, et al. Adaptive nonlinearity for collisions in complex rod assemblies. ACM Transactions on Graphics (TOG), 2014, 33(4):123:1–123:12.

[52] Chai M, Zheng C, Zhou K. A reduced model for interactive hairs. ACM Transactions on Graphics (TOG), 2014, 33(4):124:1–124:11.

[53] Jiménez P, Thomas F, Torras C. 3d collision detection: a survey. Computers & Graphics, 2001, 25(2):269–285.

[54] Klein J, Zachmann G. Point cloud collision detection. Computer Graphics Forum, volume 23. Wiley Online Library, 2004. 567–576.

[55] Figueiredo M, Oliveira J, Araújo B, et al. An efficient collision detection algorithm for point cloud models. 20th International conference on Computer Graphics and Vision, volume 43, 2010. 30–37.

[56] Xinyu Zhang Y. Interactive collision detection for deformable models using streaming aabbs. IEEE Transactions on Visualization and Computer Graphics, 2007, 13(2):318–329.

[57] Bing H, Yangzihao W, Jia Z. An improved method of continuous collision detection using ellipsoids. International Conference on Computer-Aided Industrial Design and Conceptual Design. IEEE, 2009. 2280–2286.

[58] Jain A K. Data clustering: 50 years beyond K-means. Pattern Recognition Letters, 2010, 31(8):651–666.

[59] Wong T, Luk W, Heng P. Sampling with Hammersley and Halton points. Journal of graphics tools, 1997, 2(2):9–24.

[60] 仇德元. GPGPU 编程技术. 北京: 机械工业出版社, 2011.

[61] Harris M. Optimizing parallel reduction in cuda. NVIDIA Developer Technology, 2007. 1–37.

[62] Preparata F P, Shamos M I. Computational geometry: an introduction. New York: Springer-Verlag, 1985.

[63] Preparata F, Muller D. Finding the intersection of n half-spaces in time O(n log n). Theoretical Computer Science, 1979, 8(1):45–55.

[64] Moller T. A Fast Triangle-Triangle Intersection Test. Journal of Graphics Tools, 1997, 2(2):25–30.

[65] 林建立, 唐磊, 雍俊海. 多边形网格的非流形封闭三角形网格正则化. 计算机辅助设计与图形学学报, 2014, 26(10):1557–1566.

[66] Dunn F. 3D Math Primer for Graphics and Game Development. Plano, Texas: Wordware Publishing, 2002.

[67] Ritter J. Graphics gems. chapter An Efficient Bounding Sphere, 301–303. San Diego, CA, USA: Academic Press Professional, Inc., 1990: 301–303.

[68] 唐磊, 施侃乐, 雍俊海, 等. 模型适应的凸包围多面体并行生成算法. 中国科学: 信息科学, 2014, 44(22):1515–1526.

致　谢

衷心感谢我的导师雍俊海教授对本人学习及工作的精心指导。雍老师工作勤奋、治学严谨令我非常敬佩，他严谨的学术精神以及勤奋忘我的工作态度给我留下了深刻的印象，让我终生难忘。

感谢施侃乐老师对本人的指导和帮助，GEMS 8 研发团队在本人求学过程中的提供了不少关心和帮助，特此表示衷心感谢。

感谢家人的理解和支持。

声　明

本人郑重声明：所呈交的学位论文，是本人在导师指导下，独立进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容外，本学位论文的研究成果不包含任何他人享有著作权的内容。对本论文所涉及的研究工作做出贡献的其他个人和集体，均已在文中以明确方式标明。

签　名：_____ 日　期：_____

个人简历、在学期间发表的学术论文与研究成果

个人简历

1989 年 12 月 30 日出生于重庆市石柱土家族自治县。

2008 年 9 月考入中南大学软件学院软件工程专业，2012 年 7 月本科毕业并获得工学学士学位。

2012 年 9 月免试进入清华大学软件学院攻读工学硕士学位至今。

发表的学术论文

[1] **唐磊**, 李春平, 杨柳. 统计策略序列模式挖掘及其在软件缺陷预测中的应用. 计算机科学, 2013, 40(5): 164-167.

[2] Shi KanLe, Yong JunHai, **Tang Lei**, et al. Polar NURBS surface with curvature continuity, Computer Graphics Forum. 2013, 32(7): 363-370.

[3] **唐磊**, 施侃乐, 雍俊海等. 模型适应的凸包围多面体并行生成算法. 中国科学：信息科学, 2014, 44(12): 1515-1526.

[4] 林建立, **唐磊**, 雍俊海等. 多边形网格的非流形封闭三角形网格正则化. 计算机辅助设计与图形学学报,2014,26(10):1557-1566.