基于图像空间的刚体碰撞检测的 OpenGL 实现

黄祥建,田怀文

(西南交通大学 机械学院,四川 成都 610031)

摘要:本文针对三维场景的的运动模拟,详细讨论了在 OpenGL 中利用模板测试和深度测试进行碰撞检测的原理和方法。这种基于图像空间的碰撞检测方法,充分利用了图形硬件的性能并有效地降低了 CPU 的计算负担。由于充分结合了 OpenGL 的操作特点,因此本文具有一定的实践意义。

关键词:模板缓存;碰撞检测;图像空间

中图分类号:TP391　　文献标识码:A　　文章编号:1009-3044(2007)02-10439-03

Realization of OpenGL for Ragid-body Collision Detection Based on Image Space

HUANG Xiang-jian,TIAN Huai-wen

(Institute of Mechanical Engineering,Southwest Jiao Tong University,Chengdu 610031,China)

Abstract: The paper discusses the way of collision detection for rigid-body in image space and introduces the principles to implement collision detection by Visual C++ calling OpenGL based on stencil buffer and depth buffer.This method makes eficient use of graphics rendering hardware and reduces the computation overhead of CPU.Because of making use of advantages of OpenGL,this paper has important practice meanings.

Key words: Stencil buffer;Collision detection;Image-space

1 引言

碰撞检测问题在计算机图形学、仿真、动画及虚拟现实等领域中都是至关重要的问题之一。随着计算机软硬件及网络技术的高速发展,人们迫切要求对现实世界的真实模拟及与计算机的实时交互操作。要真实地模拟现实世界中物体的运动,关键之一是必须对场景中的所有物体进行实时碰撞检测。目前,虚拟环境中的三维物体模型越来越复杂,场景越来越巨大,而实时性的要求越来越高,这种苛刻的实时性要求迫使碰撞检测过程必须尽可能地快速完成。国内外有许多专家和学者已经针对碰撞检测问题开展了不少有重要价值的研究工作,并且将其理论成果实际应用到虚拟现实、动画和网络交互等各个领域中。

碰撞检测可分为基于物体空间的碰撞检测和基于图像空间的碰撞检测。基于物理空间的碰撞检测已有了较为充分的研究,但这些算法的效率极大地取决于物体模型的表示方法和场景的复杂程度。同时,巨大的计算量往往使系统不堪负重。而基于图像空间的碰撞检测方法一般是将三维物体通过向参考面投影而降维得到一个二维的图像空间,然后分析保存在图像空间的各类缓存信息,得出物体之间是否发生碰撞并进行下一步响应处理。

基于图像空间的碰撞检测方法最初由于不精确性和对图形硬件要求过高而一直发展缓慢。近年来,随着图形硬件的快速发展和研究人员的艰辛探索,使得基于图像空间的碰撞检测方法有了新的发展。在文献[1]、[2]、[3]、[4]、[5]中提出了几种使用图形硬件进行碰撞检测的算法,它们利用了图形处理器(GPU)的一些硬件特性,因此检测速度很快。但这些方法比较基础和抽象,没有特定的实现环境,离实践还有一定距离。本文拟立足于 3D 图形的 OpenGL实现,讨论刚体运动中的碰撞检测方法。

2 缓存及其应用

几乎所有的图形程序的共同目标都是进行屏幕绘图,所有的计算机动画也是由一系列静止的图像连续播放而形成的。而屏幕图形或图像可以看作是由象素点组成的矩形阵列,每个象素点都可以显示图像中的一小块区域。事实上,图形硬件为每个象素点存储一系列显示信息,而所有象素信息的存储就称为缓存,每一帧画面的缓存即帧缓存。

在 OpenGL 中,帧缓存由以下四部分组成[6]:

(1)颜色缓存:通常就是程序员绘图所用的缓存。它存储象素的颜色索引值或 RGBA 值,通常包括实现立体感的左右缓存和支持平滑动画的前后缓存。

(2)深度缓存:深度缓存即 z 缓存,它存储每个象素的深度值,用来实现隐藏面的消隐。

(3)模板缓存:模板缓存主要用于将作图限制在它指定的范围内,实际应用中,我们可以配合深度缓存实现一些特殊应用。

(4)累积缓存:通常用于图像的累加和合成。

OpenGL 的实际应用都采用一种相似的操作顺序,即渲染管道。顶点数据和象素数据光栅化后需经过片元测试,在测试通过后才能写入帧缓存以在屏幕上输出。测试和相关操作按以下次序进行:裁剪测试、alpha 测试、模板测试、深度测试、混合、抖动、逻辑操作。本文感兴趣的是模板测试和深度测试。模板测试对存储在模板缓存中的象素值与参考值进行比较,可以根据比较结果,对模板缓存中的值进行修改操作。两个相关的函数如下:

void glStencilFunc(GLenum func, GLint ref, GLuint mask),用于将参考值(ref)和模板缓存中的值进行比较,且比较仅适用于掩码(mask)的相应位为 1 的数位。参数 func 用于指定测试操作类型,取值范围包括:GL_NEVER、GL_ALWAYS、GL_LESS、GL_LEQUAL、GL_EQUAL、GL_GEQUAL、GL_GREATER、GL_NOTEQUAL。它们对应于:总不通过测试、总是通过测试以及参考值分别小于、小于等于、等于、大于等于、大于、不等于模板值时通过测试。例如,在 GL_LESS 的情况下,如果参考值小于模板缓存中的值,则片元通过测试,可以进行下一个函数指定的操作。

void glStencilOp(GLenum fail, GLenum zfail, GLenum zpass),此函数对模板缓存的数据进行相应的修正。参数 fail, zfail, zpass 分

收稿日期:2006-09-08

作者简介:黄祥建(1977-),男,四川自贡人,讲师,西南交通大学机械工程学院机电硕士,主攻三维计算机图形学;田怀文(1965-),男,博士,教授,重庆荣昌县人,西南交通大学机械工程学院从事教学工作,在概念设计、反求工程、计算机辅助工程(CAE)和企业生产管理领域进行研究开发工作。

别用于指定片元未通过模板测试、通过模板测试但未通过深度测试、模板测试和深度测试皆通过三种情况下的修正操作，取值范围包括：GL_KEEP、GL_ZERO、GL_REPLACE、GL_INCR、GL_DECR、GL_INVERT。它们对应于：保持当前值、以 0 替换、以参考值替换、增加该值、减小该值以及对其按位取反。

深度缓存通常用于隐藏面的消隐。如果有一种新的候选颜色用于某一象素的显示，则只有该象素在物体空间中新的对应点比它先前的对应点距离视点更近时，该象素才能以新颜色被画出。也就是说，只有未被其它物体遮盖的点、表面、物体才能被画出。对于屏幕上的每个象素，深度缓存时刻追踪视点与占据该象素的物体上的对应点之间的距离。如果通过了深度测试，输入的深度值就将取代深度缓存中的相应值。我们可以 GL_DEPTH_TEST 为参数调用 glEnable()、glDisable()函数来激活或关闭深度测试，同时可以用函数 glDepthFunc(Glenum func)来更改测试条件。func 的取值同函数 glStencilFunc()中的相应值

3 图像空间中刚体的碰撞检测算法

在计算机图形学中渲染复杂物体的一个简单有效的造型方法，是使用大量多边形(三角形无疑是一个好选择)构造物体的表面。OpenGL 就提供了类似的渲染方法，但由于图形硬件信息存储的限制，只能处理凸多面体，因此，必须将任意形状的物体分解成一系列凸体的组合。目前对凸分解技术的研究与应用已经较为成熟，出现了许多用于凸包计算的软件包.本文假定所提及的刚体均为处理后的凸体。

假设 Pxy(X)是刚体 X 在 XOY 平面的垂直投影区间，Pz(X)是刚体 X 在 Z 轴上所占据的区间。那么，两刚体 A 和 B 若发生碰撞，当且仅当下式成立：

$$\begin{cases} p_{xy}(A) \cap p_{xy}(B) \neq \phi \\ Pz(A) \cap Pz(B) \neq \phi \end{cases}$$

设 Z 轴对应深度方向，则 XOY 面对应模板区域，因此可以很好的利用深度缓存和模板缓存。那么，对图像空间中两个相对运动的刚体 A、B 而言，与某一象素点相应的 A、B 上的对应点的深度重叠关系仅有以下八种情况，见图 1。

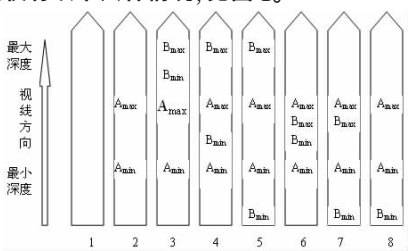

图 1　深度区间重叠关系图

在这八种关系中，4、5、6、7 为发生碰撞的情形，针对这几种深度关系，我们将三维物体空间的碰撞检测通过向屏幕投影，降维到二维图像空间，并通过模板测试及深度测试来实现碰撞检测。算法流程如图 2 所示。

算法的伪代码如下：

```
bool CollisionDetection(A,B)
{glColorMask(0,0,0,0);//屏蔽颜色缓存，只做检测
Projection(A,B);//利用投影矩阵将 A, B 进行正交投影
And(A,B);//求 A, B 在投影面(屏幕)上的重叠区域
if(重叠区域的像素为 0)return false;
glClearDepth(0.0);
glClear (GL_DEPTH_BUFFER_BIT |GL_STENCIL_BUFFER_BIT);
glStencilFunc(GL_ALWAYS,0x1,0x1);//将模板缓存置 1`
```

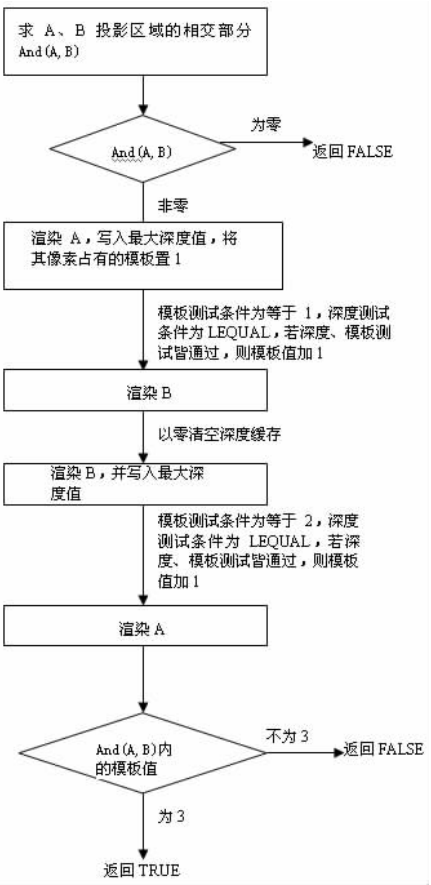

图 2

```
glStencilOp(GL_REPLACE,GL_REPLACE,GL_REPLACE);
glDepthFunc(GL_GREATER);//将最大深度写入深度缓存
Draw(A);//绘制 A 物体
glStencilFunc (GL_EQUAL,0x1,0x1);//模板缓存为 1 时通过测试
glDepthFunc(GL_LEQUAL);//深度植小于等于缓存中的值时通过测试
glStencilOp(GL_KEEP,GL_KEEP,GL_INCR);//模板缓存和深度缓存都通过，模板值加 1
Draw(B);//绘制 B 物体
glClearDepth(0.0);
glClear(GL_DEPTH_BUFFER_BIT);
glDepthFunc(GL_GREATER); //将最大深度写入深度缓存
Draw(B);//绘制 B 物体
glStencilFunc(GL_EQUAL,0x2,0x1);
//模板缓存为 2 时通过测试
glDepthFunc(GL_LEQUAL);
//深度植小于等于缓存中的值时通过测试
glStencilOp(GL_KEEP,GL_KEEP,GL_INCR);
//模板缓存和深度缓存都通过，模板值加 1
Draw(A);//绘制 A 物体
检测重叠区域内像素的模板值 StencilValue;
if(StencilValue==3)return true;
return false;}
```

如果两物体发生碰撞，那么它们的模板缓存值一定相同，且深度关系有图 1 中的四种。先将 A 物体的模板置 1。按照 A, B 的顺序渲染，如果 B 能够通过模板缓存和深度缓存，那么重叠区域内像素的模板值将变为 2。再按照 B, A 的顺序渲染，如果 A 能通

过模板缓存,那么重叠区域内像素的模板值将变为 3。如果两次渲染都通过了,那么物体发生了碰撞,屏幕上的 A,B 重叠区域内的模板值为 3。因此最终只需检测 A,B 重叠区域内是否出现了模板值为 3 的像素,如果出现了,那就说明发生了碰撞

本文作者基于这种方法,在微机上实现了基于物理模型的两个相对运动刚体的自动碰撞检测,可以不必时刻跟踪两刚体的空间坐标,同时避免了复杂的三维求交运算。

4 结束语

基于图像空间的碰撞检测方法,充分利用了图形硬件的性能,有效减轻了 CPU 的工作负荷。并且不管物体的复杂程度度,其检测时间波动不大,具有相当的平稳性。用 OpenGL 的深度测试和模板测试来实现这种碰撞检测更是具有很强的实践意义。

参考文献:

[1]G.Baciu,S.Wong,and H.Sun.Recode: An Image- Based Collision Detection Algorithm[J]. Proc.Pacific Graphics,pp.497- 512,1998.

[2]D.Knott and D.K. Pai.CInDeR: Collision and Interference Detection in Real- Time Using Graphics Hardware[J].Proc.Graphics Interface,pp.73- 80,2003.

[3]K.Myszkowski,O.G. Okunev,and T.L. Kunii.Fast Collision Detection between Complex Solids Using Rasterizing Graphics Hardware[J].The Visual Computer,vol.11,no.9,pp.497- 512,1995.

[4]J.Rossignac,A.Megahed,and B.Schneider.Interactive Inspection of Solids: Cross- Sections and Interferences[J]. Proc. ACMSIGGRAPH,pp.353- 360,1992.

[5]M.Shinya and M.C.Forgue.Interference Detection throughRasterization[J].The J.Visualization and Computer Animation, vol.2,no.4, pp.131- 134,1991.

[6]Dave Shreiner,Mason Woo,OpenGL Programming Guide, POSTS & TELECOM PRESS[J].

(上接第 417 页)

```
List<Object> ObjectList = StringList; //2
```

用普通的继承思路认为一个 String 的 List 是一个 Object 的 List。

```
ObjectList.add(new Object()); // 3
String s = StringList.get(0); // 4: 试图把 Object 赋值给 String
```

代码中使用 ObjectList 指向 StringList, 可以插入任意对象进去。结果是 StringList 中保存的不再是 String,当从中取出元素的时候,会得到意外的结果,第 2 行会导致一个编译错误。

总之,如果 B 是 A 的一个子类型(子类或者子接口),而 F 是某种泛型声明,那么 F<B>是 G<A>的子类型并不成立。

5 泛型的通配符

泛型的继承机制使 Collection<Object>不再是所有类型的 collections 的父类,取而代之的是 Collection<?>,即是一个集合,它的元素类型可以匹配任何类型,它被称为通配符。考虑下面的代码:

```
void printCollection(Collection<?> co) {
for (Object o : co) {
System.out.println(o);}}
```

可以使用任何类型的 collection 来调用它。而 co 中的元素仍然能读取,其类型是 Object。这永远是安全的,因为不管 collection 的真实类型是什么,它包含的都是 objects。但是将任意元素加入到其中不是类型安全的:

```
Collection<?> co = new ArrayList<String>();
co.add(new Object()); // 编译时错误
```

因为不知道 co 的元素类型, 所以不能向其中添加对象。add 方法有类型参数 E 作为集合的元素类型。传给 add 的任何参数都必须是一个未知类型的子类。但不能确定那是什么类型,所以无法传任何东西进去。唯一的例外是 null,它是所有类型的成员。另一方面,可以调用 get()方法并使用其返回值,因为它是一个未知的类型,但它总是一个 Object,因此可以把 get 的返回值赋值给一个 Object 类型的对象或者放在任何希望是 Object 类型的地方。

6 泛型方法

方法的声明也可以被泛型化——就是说,带有一个或者多个类型参数:

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c){
for (T o : a) {
  c.add(o); // correct}}
```

可以使用任意集合来调用这个方法,只要其元素的类型是数组的元素类型的父类。

```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
fromArrayToCollection(oa, co);// T 指 Object
```

需要注意的是,在这里并没有传送真实类型参数给一个泛型方法。编译器根据实参为推断类型参数的值。它通常推断出能使调用类型正确的最明确的类型参数。

另一个问题:什么时候使用泛型方法,时候使用通配符类型

看看 Collection 库中的几个方法:

```
public interface Collection<E> {
boolean containsAll(Collection<?> c);
boolean addAll(Collection<? extends E> c);}
```

也可以使用泛型方法来代替:

```
public interface Collection<E> {
 <T> boolean containsAll(Collection<T> c);
 <T extends E> boolean addAll(Collection<T> c);}
```

但是, 在 containsAll 和 addAll 中, 类型参数 T 都只使用一次。返回值的类型既不依赖于类型参数也不依赖于方法的其他参数。也就是说类型参数被用作多态,它唯一的效果是允许在不同的调用点,可以使用多种实参类型。如果是这种情况,应该使用通配符。通配符就是被设计用来支持灵活的子类化的。泛型方法允许类型参数被用来表示方法的一个或多个参数之间的依赖关系,或者参数与其返回值的依赖关系。如果没有这样的依赖关系,不应该使用泛型方法。

7 总结

Java 泛型语法并未提供新关键词,而是以角括号(<>)表示类型参数。Java 编译器将泛型语法还原为旧式的非泛型语法。此与 C++面对泛型所采用的膨胀法不同。泛型 Java 给予程序员在类型检验上的协助,使程序员的工作更轻松,更不易出错。泛型编程是面向对象的进一步补充与扩展, 近年来越来越受人们的关注,泛化的结果是带来了更大规模的复用、更好的通用性、更高抽象程度。GP 本身也处在一个不断发展的阶段, 尚有极大潜力有待挖掘。

参考文献:

[1](美)奥斯腾,著,侯捷,译.泛型编程与 STL[M].北京:中国电力出版社, 2003.

[2]孙斌.面向对象,泛型程序设计与类型约束检查[C].计算机学报, 2004, 27(11): 1492.

[3]孙斌.扩展面向对象编程(XOOP)的理论和方法[J].计算机学报, 2001, 24(3): 266- 280.