计算几何文档(1)

DionysosLai 2014-4-24

计算几何主要研究解决几何问题的算法。在现代工程和数学领域，计算几何在图形学、机器人技术、超大规模集成电路设计和统计等诸多领域有着十分重要的应用。同样在游戏领域，计算几何对开发有特别的重要。在本文中，即接下来的文章，将对计算几何常用的基本算法做一个全面的介绍，希望对您了解并应用计算几何的知识解决问题起到帮助。同时，由于目前本人使用的是游戏引擎引擎是Cocos2d-x，因此使用的一些类名是基于Cocos2d-x的，如果你也正好是使用这个引擎，可以直接移植代码，但如果你不是使用此引擎，只要稍微更改一下代码也可以试用。

1. 矢量的叉积

设矢量P = ( x1, y1 )，Q = ( x2, y2 )，则矢量叉积定义为由(0,0)、p1、p2和p1+p2所组成的平行四边形的带符号的面积，即：P × Q = x1\*y2 - x2\*y1，其结果是一个标量。显然有性质 P × Q = - ( Q × P ) 和 P × ( - Q ) = - ( P × Q )。一般在不加说明的情况下，本文下述算法中所有的点都看作矢量，两点的加减法就是矢量相加减，而点的乘法则看作矢量叉积。

叉积方向：当使用右手坐标系时，可以使用右手负责来表示叉积方向。伸开右手掌，使除项目以外的4只手指指向P矢量的方向，再把4只手指指向内弯曲指向Q方向，那么拇指的方向便是叉积（P×Q）的方向。

根据，叉积方向定义，叉积的一个非常重要性质是可以通过它的符号判断两矢量相互之间的顺逆时针关系：

若 P × Q > 0 , 则P在Q的顺时针方向。

若 P × Q < 0 , 则P在Q的逆时针方向。

若 P × Q = 0 , 则P与Q共线，但可能同向也可能反向。

叉积代码如下：

double vectorProduct(double x1, double y1, double x2, double y2) // 行列式

{

return (x1\*y2-x2\*y1); ///< 叉积

}

1. 折线段的拐向判断

根据上文叉积性质，可以判断两条有公共顶点的线段方向。

原理如下：

折线段的拐向判断方法可以直接由矢量叉积的性质推出。对于有公共端点的p0p1和p1p2，通过计算(p2 - p0) × (p1 - p0)的符号便可以确定折线段的拐向：

若(p2 - p0) × (p1 - p0) > 0,则p0p1在p1点拐向右侧后得到p1p2。

若(p2 - p0) × (p1 - p0) < 0,则p0p1在p1点拐向左侧后得到p1p2。

若(p2 - p0) × (p1 - p0) = 0,则p0、p1、p2三点共线。

代码如下：

///@brief 判断折线的拐向，

///

///以线段P1为参考，；

///@param[in] p1--线段1，p2--线段2

///@pre P1、P1必须有公共端点

///@return 1--左边 0 -- 共线 -1---右边

///@retval

///@post

///@author DionysosLai,906391500@qq.com等

///@version 1.0

///@data 2014-04-10

int PolyLineDerection(const CCPoint p1, const CCPoint p0, const CCPoint p2)

{

float vectorProductResult = 0.0f;

vectorProductResult = (float)vectorProduct(p1.x-p0.x, p1.y-p0.y, p2.x-p0.x, p2.y-p0.y);

if (vectorProductResult < 1e-3 && vectorProductResult > -1e-3)

{

CCLOG("p1p0 and p2p0 is at the same dereciton.");

return 0;

}

else if (vectorProductResult >= 1e-3)

{

CCLOG("p2p0 is at p1p0's left dereciotn.");

return -1;

}

else if (vectorProductResult <= -1e-3)

{

CCLOG("p2p0 is at p1p0's right dereciotn.");

return 1;

}

}

1. 判断点是否在矩形内

原理：只要判断点的横坐标在矩形左右横坐标内，点的纵坐标在矩形上下纵坐标内即可。

代码如下：

///@brief 判断点是否在矩形内(点在边界也算)

///

///只要判断该点的横坐标和纵坐标是否夹在矩形的左右边和上下边之间即可。

///@param[in] p0 --- 要判断的点 r0,r1矩形对角线的两个端点

///@pre r0,r1必须可以形成一个矩形

///@return true---在矩形内 false ---不在矩形内；

///@retval

///@post

///@author DionysosLai,906391500@qq.com等

///@version 1.0

///@data 2014-04-10

bool HelloWorld::pointIsInRect(const CCPoint p0, const CCPoint r0, const CCPoint r1)

{

/// 首先判断r0 r1 是否可以组成一个矩形；----这个不用判断，不然点在线上就判断不出来

// if (r0.x == r1.x || r0.y == r1.y)

// {

// CCLOG("Rectangle cannot be maken up by the two points!");

// return false;

// }

/// 获取矩形的4条边界

float minX = r0.x > r1.x ? r1.x : r0.x;

float maxX = r0.x > r1.x ? r0.x : r1.x;

float minY = r0.y > r1.y ? r1.y : r0.y;

float maxY = r0.y > r1.y ? r0.y : r1.y;

/// 判断点是否在矩形左右、上下边界内

if ((p0.x >= minX && p0.x <= maxX) &&

(p0.y >= minY && p0.y <= maxY))

{

CCLOG("Point is in rectangle!");

return true;

}

else

{

CCLOG("Point is not in rectangle!");

return false;

}

}

1. 点在线段上  
    原理：先判断点是否在线段形成的矩形上，这个保证点不会在线段的延长线上；再次判断点与线段的2个端点是否共线。

代码如下：

///@brief 判断点是否在线段上

///

///

///@param[in] p0--点，p1,p2--线段两个端点

///@pre

///@return true---在线段上， false---不在线段上

///@retval

///@post

///@author DionysosLai,906391500@qq.com等

///@version 1.0

///@data 2014-04-10

bool LineDebug::pointIsAtLine(const CCPoint p0, const CCPoint p1, const CCPoint p2)

{

/// 先判断是否点在以p1 p2为对角线的矩形内

if (pointIsAtRect(p0, p1, p2))

{

/// 判断p1p0, p2p0是否共线

if (0 == PolyLineDerection(p0, p1, p2))

{

CCLOG("Point in at the line.");

return true;

}

CCLOG("Point isn't at the line.");

return false;

}

CCLOG("Point isn't at the line.");

return false;

}

1. 判断两个矩形是否相交

原理：两个矩形的中心点水平距离比两个矩形长的和的一半小；中心点垂直距离比两个矩形高的和的一半小。

代码如下：

///@brief 判断两个矩形是否相交

///

///

///@param[in] aa,bb--矩形1一个对角线端点， cc,dd--矩形2一个对角线端点

///@pre

///@return true---相交， false---不相交

///@retval

///@post

///@author DionysosLai,906391500@qq.com等

///@version 1.0

///@data 2014-04-10

bool LineMenu::isRectCollision(CCPoint aa, CCPoint bb, CCPoint cc, CCPoint dd)

{

CCPoint centre1 = ccp((aa.x + bb.x)/2.0f, (aa.y + bb.y)/2.0f); ///< 获得中点值

CCPoint centre2 = ccp((cc.x + dd.x)/2.0f, (cc.y + dd.y)/2.0f);

float lengthX = abs(centre1.x - centre2.x); ///< 获得两个矩形中心的距离

float lengthY = abs(centre1.y - centre2.y);

float lengthRect1X = abs(aa.x - bb.x); ///< 获得两个矩形长和宽

float lengthRect1Y = abs(aa.y - bb.y);

float lengthRect2X = abs(cc.x - dd.x);

float lengthRect2Y = abs(cc.y - dd.y);

/// 这里减去1是调整误差用的。

return (lengthX < (lengthRect1X + lengthRect2X)/2.0f-1 && lengthY < (lengthRect1Y + lengthRect2Y)/2.0f-1) ? true : false;

}