# 2.5 操控行为——避开障碍

obstacle avoidance行为操控交通工具避开路上的障碍。障碍物是任何一个近似圆周（在3D中，将是球体）的物体。我们保持长方形区域（检测盒，从交通工具延伸出的）不被碰撞，就可以躲避障碍了。检测盒的宽度等于交通工具的包围半径，它的长度正比于交通工具的当前速度（它移动得越快，检测盒就越长）。

深入讲解处理过程之前，先看一下图，图中有一个交通工具、多个阻挡物和计算中用到的检测盒。



**用于躲避障碍操控行为的信息**

## 寻求最近的相交点

检查与障碍物的相交点的过程相当复杂，所以下面逐步讲解。

A．交通工具只考虑那些在检测盒内的障碍物。最初，避开障碍算法迭代游戏世界中所有的障碍物，标记那些在检测盒内的障碍物以作进一步分析。

B．算法然后把所有已标记的障碍物转换到交通工具的局部空间。转换坐标后，那些x坐标为负值的物体将不被考虑，所以问题就变得简单多了。

C．接下来，该算法必须测试障碍物是否和检测盒重叠。使障碍物的包围半径扩大检测盒（交通工具的包围半径）宽度的一半。然后测试该障碍物的y值是否小于这个值（即障碍物的包围半径，加上检测盒宽度的一半）。如果不小于，那么该障碍将不会与检测盒相交，所以不予继续讨论。

下图中列出了前面3步。图中障碍物上的字母A、B、C和前面的描述相对应。



**步骤A、B、C**

D．此时，只剩下那些会与检测盒相交的障碍物了。接下来，我们找出离交通工具最近的相交点。我们将再一次在局部空间中计算，C步骤扩大了障碍物的包围半径。我们用简单的线-圆周相交测试方法可以得到被扩大的圆和x轴的相交点。如图所示，有两个相交点（我们不用担心和交通工具相切的情况，交通工具将擦过障碍物）。注意如下情况，在交通工具前面可能有个障碍物，但和交通工具相交在交通工具的后部，如图中的障碍物A，和交通工具的后部有个交点。该算法将不处理这种情况，只考虑存x轴上的交点。



**相交点**

此算法测试所有剩下的障碍物，从中找出一个有最近的（正的）相交点的障碍物。

在讲解如何计算操控力之前，先列出实现躲避障碍算法中从步骤A到步骤B的相关代码。

//---------------------- ObstacleAvoidance -------------------------------

//

// 给定一个障碍物集合，这个方法返回一个操控力，

// 这个力让智能体不与最近的障碍物发生碰撞。

//------------------------------------------------------------------------

public Vector2 ObstacleAvoidance(List<BaseGameEntity> obstacles)

{

 // 检测盒长度正比于智能体的速度

 float \_dBoxLength = Global.MinDetectionBoxLength +

 (\_vehicle.Speed / \_vehicle.MaxSpeed) \* Global.MinDetectionBoxLength;

 // 标记在检测盒半径之内的障碍物

 \_vehicle.World.TagObstaclesWithinViewRange(\_vehicle, \_dBoxLength);

 // 保存最近的相交障碍物(CIB)

 BaseGameEntity ClosestIntersectingObstacle = null;

 // 到最近相交障碍物的距离

 float DistToCIB = float.MaxValue ;

 // 保存最近相交障碍物在局部空间中的坐标

 Vector2 LocalPosOfClosestObstacle=Vector2 .Zero ;

 foreach(BaseGameEntity curOb in obstacles )

 {

 // 如果该障碍物被标记在范围内

 if (curOb.IsTagged)

 {

 // 计算这个障碍物在智能体的局部空间中的位置

 Vector2 LocalPos = Vector2.Transform(curOb.Pos,

 Matrix.Invert(Matrix.CreateRotationZ((float)(Math.Atan2(\_vehicle.Heading.Y, \_vehicle.Heading.X)))

 \* Matrix.CreateTranslation(new Vector3(\_vehicle.Pos, 0))));

 // 如果局部空间中的X为负，那么障碍物在智能体后方，可以忽略。

 // 所以我们只考虑X不为负的情况。

 if (LocalPos.X >= 0)

 {

 // 如果障碍物到x轴的距离小于它的半径+检测盒宽度的一半，

 // 则有可能相交。

 float ExpandedRadius = curOb.BRadius + \_vehicle.BRadius;

 if (Math.Abs(LocalPos.Y) < ExpandedRadius)

 {

 // 进行线/圆周相交测试。圆周中心坐标用(cX, cY)表示，

 // 相交点在局部空间x轴上的坐标值由公式x = cX +/-sqrt(r^2-cY^2)给出，此时y=0。

 // 我们只需要x的最小正值，它就是最近的相交点。

 float cX = LocalPos.X;

 float cY = LocalPos.Y;

 float SqrtPart = (float)Math.Sqrt(ExpandedRadius\*ExpandedRadius - cY\*cY);

 float ip = cX - SqrtPart;

 if (ip <= 0.0)

 {

 ip = cX + SqrtPart;

 }

 // 测试是不是最近点，如果是则保存这个障碍物和的局部坐标

 //record of the obstacle and its local coordinates

 if (ip < DistToCIB)

 {

 DistToCIB = ip;

 ClosestIntersectingObstacle = curOb;

 LocalPosOfClosestObstacle = LocalPos;

 }

 }

 }

 }

}

## 计算操控力

计算操控力是容易的，通常分成两部分：侧向操控力和制动操控力，如图所示。



**计算操控力**

我们有许多方法可以计算侧向操控力，此处建议如下这种：障碍物包围半径减去其在局部空间中的y值。这会产生一个侧向操控力使其远离障碍物，它会随着障碍物到x轴的距离而减少。该力正比于交通工具到障碍物的距离（因为交通工具离障碍物越近，反应越快）。

操控力的另一部分是制动力。该力沿着图中的负x轴方向，其大小正比于交通工具到障碍物的距离。

最后，把操控力转换到世界空间，就是该方法的返回的结果。代码如下：

 // 如果我们找到了一个相交障碍物，则计算一个远离它的力

 Vector2 SteeringForce=Vector2.Zero ;

 if (ClosestIntersectingObstacle!=null)

 {

 // 智能体离物体越近，操控力就应越强

 float multiplier = 1.5f + (\_dBoxLength - LocalPosOfClosestObstacle.X) / \_dBoxLength;

 // 计算侧向力

 if( LocalPosOfClosestObstacle.Y>0)

 SteeringForce.Y = (-ClosestIntersectingObstacle.BRadius - LocalPosOfClosestObstacle.Y)

 \* multiplier;

 else

 SteeringForce.Y = (ClosestIntersectingObstacle.BRadius- LocalPosOfClosestObstacle.Y)

 \* multiplier;

 // 施加一个制动力，它正比于智能体到障碍物的距离

 const float BrakingWeight = 0.6f;

 SteeringForce.X = (ClosestIntersectingObstacle.BRadius - LocalPosOfClosestObstacle.X) \*

 BrakingWeight;

 }

 // 最后，将操控力向量从局部坐标转换到世界坐标

 return Vector2.Transform(SteeringForce,

 Matrix.CreateRotationZ((float)(Math.Atan2(\_vehicle.Heading.Y, \_vehicle.Heading.X))));

}

## 辅助方法

在上述ObstacleAvoidance方法中还用到了些辅助方法，这些辅助方法都位于GameWorld类中，代码如下：

// 创建一些随机大小的障碍物

private void CreateObstacles()

{

 for (int i=0; i<Global.NumObstacles; i++)

 {

 bool bOverlapped = true;

 // 不停尝试直至找到一个不与其他障碍物重叠的障碍物。

 int NumTrys = 0; int NumAllowableTrys = 2000;

 while (bOverlapped)

 {

 NumTrys++;

 if (NumTrys > NumAllowableTrys)

 return;

 // 随机产生一个介于MinObstacleRadius和MaxObstacleRadius之间的半径

 int radius = Global.random.Next((int)Global.MinObstacleRadius, (int)Global.MaxObstacleRadius);

 // 障碍物至少与边界距离10，障碍物之间至少距离20

 int border = 10;

 int MinGapBetweenObstacles = 20;

 Obstacle ob = new Obstacle(Global.random.Next(radius + border, \_xClient - radius - border),

 Global.random.Next(radius + border, \_yClient - radius - 40 - border), radius);

 if (!Overlapped(ob, \_obstacles, MinGapBetweenObstacles))

 {

 // 如果不重叠则添加到集合中

 \_obstacles.Add(ob);

 bOverlapped = false;

 }

 }

 }

}

//------------------------- Overlapped -----------------------------------

//

// 测试一个实体是否与另一个实体集合中的某个相交

//------------------------------------------------------------------------

private bool Overlapped(BaseGameEntity ob, List<BaseGameEntity> conOb, float MinDistBetweenObstacles)

{

 foreach (BaseGameEntity it in conOb)

 {

 if (TwoCirclesOverlapped(ob.Pos, ob.BRadius + MinDistBetweenObstacles, it.Pos, it.BRadius))

 {

 return true;

 }

 }

 return false;

}

//----------------------------- TwoCirclesOverlapped ---------------------

//

// 如果两个圆相交则返回true

//------------------------------------------------------------------------

private bool TwoCirclesOverlapped(Vector2 c1, float r1, Vector2 c2, float r2)

{

 float DistBetweenCenters = Vector2.Distance(c1, c2);

 if ((DistBetweenCenters < (r1 + r2)) || (DistBetweenCenters < Math.Abs(r1 - r2)))

 {

 return true;

 }

 return false;

}

// 给在range半径范围内的障碍物做标记

public void TagObstaclesWithinViewRange(BaseGameEntity vehicle, float range)

{

 TagNeighbors(vehicle, \_obstacles, range);

}

//----------------------- TagNeighbors ----------------------------------

//

// 在实体集合ContainerOfEntities中，若某个实体位于以指定实体entity为圆心，

// radius为半径的范围内，则对它打上标记

//------------------------------------------------------------------------

private void TagNeighbors(BaseGameEntity entity, List<BaseGameEntity> ContainerOfEntities, float radius)

{

 foreach (BaseGameEntity curEntity in ContainerOfEntities)

 {

 // 首先取消标记

 curEntity.UnTag();

 // 两个实体间的距离

 Vector2 to = curEntity.Pos - entity.Pos;

 // 还需考虑包围球半径

 float range = radius + curEntity.BRadius;

 // 若在范围内，则打上标记

 if ((curEntity != entity) && (to.LengthSquared() < range \* range))

 {

 curEntity.Tag();

 }

 }

}