# 2.1 什么是自治智能体

在20世纪80年代晚期，BBC Horizon一期纪录片，主要讲述了经典的计算机图形和动画。片中呈现的内容精彩纷呈，令人兴奋，其中最生动的莫过于一群鸟的群集行为。它的原理其实十分简单，但看上去的确很自然逼真。节目的设计者叫Craig Reynolds。他称群鸟为“boids”，称那些简单原理为操控行为（Steering Behaviors）。

从那以后，Reynolds发表了几篇关于研究不同操控行为的文章，均受到好评。他所介绍的大多数操控行为与游戏直接关联，下面的文章会让你了解如何使用它们，以及如何用代码加以实现。

对于自治智能体的定义，有许多不同的版本，但是如下这个可能是最好的：

**一个自治智能体是这样一个系统，它位于一个环境的内部，是环境的一部分，且能感知该环境对它有效实施的作用，并永远按此进行，为未来的新感知提供条件。**

本章提到的“自治智能体”指拥有一定自治动作的智能体。倘若一个自治智能体偶尔发现意外的状况，如发现一堵墙挡住了去路，马上会做出反应，根据情况调整动作。比方说，你可能会设计两个自治智能体，一个可以像兔子那样行为，另一个能像狐狸。假设兔子正在愉快地享受一片湿润的嫩草，忽然看见了狐狸，它会自治地逃走，同时狐狸也会自治地追捕兔子。所有的动作都是自行完成，中间不需要设计者介入。一旦开始后，自治智能体都可以简单地自行处理。

这并不是说自治智能体绝对可以应付任何情况（虽然那可能是你的目标之一），但它可以非常有效地应付许多情况。例如，在编写寻路代码时经常出现处理动态障碍的问题。动态障碍就是那些在游戏世界里游来游去、不时更换位置的物体，如其他的智能体、滑门等等。如果存在一个合适的环境，将正确的操控行为安插到游戏角色上，就能很好地避免为处理动态障碍而编写特定的代码（自治智能体总能在必要的时候处理这些问题）。

自治智能体的运行过程可以分解成以下三个小环节。

* 行动选择：该部分负责选定目标、制定计划。它告诉我们“到这来”和“做好A、B，然后做C”。
* 操控：该环节负责计算轨道数据，服务行动选择环节制定的目标和计划。由操控行为执行。操控行为产生了一个操控力，它决定智能体往哪移动及如何快速移动。
* 移动：最后环节。主要表现一个智能体运动的机械因素，即如何从A到B。比如，如果你掌握了骆驼、坦克和金鱼的机械学原理，并命令它们向北走，它们会依据各种不同的力学方法来产生动作，即使它们有相同的意向。将移动环节与操控环节区分开来，就很有可能以相同的操控行为来完成迥然不同的移动，而几乎不需要修正。

Reynolds在他的论文“Steering Behaviors for Autonomous Characters”中，运用了一个精彩的类比阐述以上三环节的各自作用。

“打个比方，设想一群牛仔在山间牧牛。有一头牛走出牛群，老板让一个牛仔去把它找回来，牛仔对他的马说了一声“驾”，骑着马来到走开的牛的旁，他在途中可能还要绕开障碍。在这个例子里，老板就是行动选择环节：注意到世界情形己改变（牛离开了牛群），于是选定—个目标（找到牛）。操控部分由这个负责寻牛的牛仔来担当，他将目标分解成几个子目标来完成（靠近牛，绕开障碍，牵牛回来）。一个子目标要符合牛仔和马团队的操控行为。运用多样的控制信号（语音指令，策马前进、勒紧缰绳），牛仔骑上马朝目标出发。一般来说，这些信号代表一些含义，比如，速度快点、慢点、向右走、向左走等。而马执行移动环节。它被输入了牛仔给它的控制信号，朝着指示方向运动，这种运动是马的官能相互复杂作用的结果，这些官能包括的视觉感知、平衡感，还有使马的腿关节活动的肌肉。”

自治智能体并不容易实现，操控行为的执行会给程序员带来无数的困难。一些行为会带来繁重的参数微调工作，还有一些其他行为则需要细心编码从而避免占用大量的CPU时间。当我们组合使用行为时，一定要注意两个或两个以上的行为有可能会排斥对方。虽然大多数的问题都有对应的方法来解决（微调除外，但是微调的过程是有趣的），而且通常操控行为的益处远多于它带来的弊端。

## 交通工具模型

在讨论各个操控行为之前，先介绍交通工具模型（运动）的编码和类设计。MovingEntity是一个基类，所有可移动的游戏智能体都继承于它。它封装一些数据，用来描述作为质点的基本交通工具。代码如下：

public abstract class MovingEntity : BaseGameEntity

{

protected Vector2 \_velocity; // 实体的速度向量

protected Vector2 \_heading; // 归一化的速度向量，即实体的朝向

protected float \_mass;

protected float \_maxSpeed; // 实体的最大速度

protected float \_maxForce; // 实体所能受到的最大力

public MovingEntity(Vector2 position, float radius, Vector2 velocity, float max\_speed, Vector2 heading,

float mass, Vector2 scale, float max\_force)

: base(position, radius)

{

\_velocity = velocity;

\_maxSpeed = max\_speed;

\_heading = heading;

\_mass = mass;

\_scale = scale;

\_maxForce = max\_force;

}

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

// 属性和公有方法

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

public Vector2 Velocity { get { return \_velocity; } }

public void SetVelocity(Vector2 NewVel) { \_velocity = NewVel; }

public float Mass { get { return \_mass; } }

public float MaxSpeed { get { return \_maxSpeed; } }

public void SetMaxSpeed(float new\_speed) { \_maxSpeed = new\_speed; }

public float MaxForce { get { return \_maxForce; } }

public void SetMaxForce(float mf) { \_maxForce = mf; }

public bool IsSpeedMaxedOut{get{return \_maxSpeed\*\_maxSpeed >= \_velocity.LengthSquared();}}

public double Speed { get { return \_velocity.Length(); } }

public Vector2 Heading { get { return \_heading; } }

public void SetHeading(Vector2 new\_heading) { \_heading = new\_heading; }

}

MovingEntity类继承于BaseGameEntity类，后者是一个定义了ID、类型、位置、包围半径和缩放比例的实体。

在本例中，交通工具的朝向向量将总是与速度一致（例如，火车的朝向和速度一致）。这些值将经常用在操控行为的算法中，而且每帧都被更新。

尽管这些数据足够表述一个可移动的对象，但是还需可以访问不同种类的操控行为。创建一个继承于MovingEntity的类Vehicle，它拥有操控行为类SteeringBehaviors的实例。

SteeringBehaviors封装了所有不同的操控行为，将在后面详细讨论。现在让我们看一下Vehicle类的代码。

public class Vehicle:MovingEntity

{

private GameWorld \_world; // 实体所在的GameWorld

private float \_timeElapsed; // 更新用时

private SteeringBehavior \_steering;

private Color color = new Color(0, 0, 255); // 三角形的颜色，默认为蓝色

// Vehicle实际上是一个三角形，顶点坐标如下所示

private Vector2[] vertexs = { new Vector2(-1.0f, 0.6f), new Vector2(1.0f, 0f), new Vector2(-1.0f, -0.6f) };

private Vector2[] transformedVertexs = new Vector2[3]; // 经过变换的三角形顶点坐标

Matrix transformMatrix; // 变换矩阵

// 属性

public GameWorld World { get { return \_world; } }

public float TimeElapsed { get { return \_timeElapsed; } }

public SteeringBehavior Steering { get { return \_steering; } }

public Color Color { get { return color; }}

public Vehicle(GameWorld world, Vector2 position, float rotation, Vector2 velocity, float mass,

float max\_force, float max\_speed, float scale):base(position,scale,velocity,max\_speed,

new Vector2 ((float)(Math.Sin(rotation)),-(float)(Math.Cos (rotation))),mass,new Vector2 (scale,scale) ,max\_fo

{

\_world = world;

\_steering = new SteeringBehavior(this);

}

// 更新实体的位置和朝向

public override void Update(float time\_elapsed)

{

\_timeElapsed = time\_elapsed;

// 通过SteeringForce类计算实体所受的操控类

//Vector2 SteeringForce = \_steering.Calculate();

Vector2 SteeringForce = Vector2.Normalize(new Vector2(250f, 250f) - this.\_pos) \* 50f;

// 由牛顿第二定律，加速度等于力除以质量

Vector2 acceleration = SteeringForce / \_mass;

// 更新速度

\_velocity += acceleration \* \_timeElapsed;

// 确保不超过最大速度

Global.Truncate (ref \_velocity,\_maxSpeed);

// 更新位置

\_pos += \_velocity \* \_timeElapsed;

// 如果速度不为零则更新朝向

if (\_velocity.LengthSquared() > 0.00000001)

{

\_heading = Vector2.Normalize(\_velocity);

}

// 让实体在屏幕中循环运动，而不是跑出边界就再也不回来

Global.WrapAround(ref \_pos,\_world.cxClient,\_world.cyClient);

}

public override void Render()

{

// 对三角形的顶点进行变换

transformMatrix =Matrix.CreateScale(new Vector3(\_scale,0.0f)) \*

Matrix.CreateRotationZ ((float)(Math.Atan2(\_heading.Y,\_heading.X)))\*

Matrix.CreateTranslation(new Vector3(\_pos, 0));

Vector2.Transform(vertexs, ref transformMatrix, transformedVertexs);

// 绘制三角形

ShapeRender.AddTriangle(transformedVertexs[0],transformedVertexs[1],transformedVertexs[2], color);

}

}

Vehicle的Update方法很重要，因为它是Vehicle类的核心，具体的原理其实就是高中物理中的牛顿第二定律，高中物理知识告诉我们，一个物体的运动情况是由它的初始状态（初始位置和初始速度）和所受力决定的，我们在实例化一个vehicle时指定了它的初始位置和初始速度，只需在每次更新时赋予它所受的力即可。更具体的介绍可以参见本网站中的[4．质点动力学——牛顿运动定律](http://shiba.hpe.cn/jiaoyanzu/wuli/showArticle.aspx?articleId=789&classId=4)及其系列文章。

由于目前我们还没有实现SteeringBehavior类，因此这里手动定义了一个力。这个力的大小为50，时刻指向屏幕中央。由高中物理知识可知，这个力叫做向心力，F＝mv2/r，本例中交通工具的质量为1，半径为200，速度为100，因此向心力为50，这样在屏幕上的三角形就会做匀速圆周运动。

MovingEntity有一个每帧都被更新的局部坐标系。因为交通工具的朝向总是与速度一致，所以需要更新，使其等于速度的标准向量。但是（这很重要）只有交通工具的速度大于一个很小的阈值时，才能计算出朝向。这是因为如果速度为0，程序将出现除0错误，如果速度不为0，但是很小，交通工具可能在停下来后还会不规则地移动几秒。

最后，显示区域是从上到下、从左到右是环绕的（wrap around），因此，我们要检查更新后的位置是否超出屏幕边界。如果是则位置将被相应地环绕。

GameWorld类包含智能体所在环境的所有数据和对象，例如墙、障碍物等（目前还没有实现），代码如下：

public class GameWorld

{

private List<Vehicle> \_vehicles=new List<Vehicle> (); // vehicle集合

private int \_xClient, \_yClient; // World大小

private Vector2 \_positionCrosshair; // 十字形位置

public List<Vehicle> Agents { get { return \_vehicles; } }

public int cxClient { get { return \_xClient; } }

public int cyClient { get { return \_yClient; } }

public Vector2 PositionCrosshair { get { return \_positionCrosshair; } }

public GameWorld(int cx, int cy)

{

\_xClient = cx;

\_yClient = cy;

// 在屏幕中央绘制十字形

\_positionCrosshair = new Vector2(\_xClient / 2.0f, \_yClient / 2.0f);

Vehicle vehicle = new Vehicle(this,

new Vector2 (450f,250f),

0.0f, // 初始旋转角度

new Vector2(0f,100f) , // 初速方向向下，大小100

Global.VehicleMass, // 质量

Global.MaxSteeringForce, // 最大力

Global.MaxSpeed, // 最大速度

10.0f); // 缩放

\_vehicles.Add (vehicle);

}

public void Update(float time\_elapsed)

{

// 处理鼠标和键盘输入

HandleInput();

// 更新所有vehicle

for (int i=0; i<\_vehicles.Count ;i++)

{

\_vehicles[i].Update(time\_elapsed);

}

ShapeRender.UpdateVertexBuffer ();

}

private void HandleInput()

{

// 获取当前的键盘和鼠标状态

KeyboardState keyboardState = Keyboard.GetState();

MouseState mouseState = Mouse.GetState();

// 如果点击鼠标左键，则将图片的左上角移动到光标的位置

if (mouseState.LeftButton == ButtonState.Pressed)

{

\_positionCrosshair = new Vector2(mouseState.X, mouseState.Y);

}

}

public void Render(GameTime gameTime)

{

// 绘制十字形

DrawCrosshair(\_positionCrosshair);

// 绘制所有vehicle

for (int i = 0; i < \_vehicles.Count; i++)

{

\_vehicles[i].Render ();

}

ShapeRender.Draw(gameTime);

}

// 绘制十字形

private void DrawCrosshair(Vector2 position)

{

ShapeRender.AddCircle(position, 4, 12, new Color(255,0,0));

ShapeRender.AddLine(position + new Vector2(-8.0f, 0.0f), position + new Vector2(8.0f, 0.0f), new Color(255, 0, 0));

ShapeRender.AddLine(position + new Vector2(-0.0f, 8.0f), position + new Vector2(0.0f, -8.0f), new Color(255, 0, 0));

}

}

从上面的代码可知，我们现在在屏幕中央绘制了一个十字形，在十字形右侧200像素的位置绘制了一个Vehicle，并指定了它的速度大小和方向。

## 渲染系统

要完成本程序屏幕绘制工作，只需提供画直线的方法，画三角形、圆形等其他图形的方法都可以在画直线的基础上获得。你可以使用GDI、flash、HTML5 Canvas等API实现，我用的是基于XNA的Silverlight 3D程序，原理可见[重用XNA的代码](http://shiba.hpe.cn/jiaoyanzu/wuli/showArticle.aspx?articleId=982&classId=4)。

因为本文关注的人工智能，因此只对渲染代码做简单介绍。负责渲染的是静态类ShapeRender，代码如下：

public class ShapeRender

{

private static GraphicsDevice graphics;

private static SilverlightEffect LineEffect;

private static DynamicVertexBuffer vertexBuffer;

private static List<VertexPositionColor> vertList = new List<VertexPositionColor>();

private static int lineCount = 0;

// 初始化渲染程序

public static void Initialize(GraphicsDevice graphicsDevice, Microsoft.Xna.Framework.Content.ContentManager content,

int screenWidth, int screenHeight)

{

if (graphics != null)

throw new InvalidOperationException("Initialize can only be called once.");

graphics = graphicsDevice;

vertexBuffer = new DynamicVertexBuffer(graphicsDevice, VertexPositionColor.VertexDeclaration, 1000, BufferUsage.None);

LineEffect = content.Load<SilverlightEffect>("LinearRendering");

LineEffect.Parameters["screenWidth"].SetValue (screenWidth);

LineEffect.Parameters["screenHeight"].SetValue (screenHeight);

}

// 添加线段

public static void AddLine(Vector2 a, Vector2 b, Color color)

{

Vector3 va = new Vector3(a, 0.0f);

Vector3 vb = new Vector3(b, 0.0f);

vertList.Add(new VertexPositionColor(va, color));

vertList.Add(new VertexPositionColor(vb, color));

lineCount += 1;

}

// 添加三角形

public static void AddTriangle(Vector2 a, Vector2 b, Vector2 c, Color color)

{

AddLine(a, b, color);

AddLine(b, c, color);

AddLine(c, a, color);

}

// 添加圆形

public static void AddCircle(Vector2 pos, float radius, int circleResolution, Color color)

{

float step = MathHelper.TwoPi / circleResolution ;

for (float a = 0f; a < MathHelper.TwoPi; a += step)

{

AddLine(pos + new Vector2((float)Math.Cos(a) , (float)Math.Sin(a))\* radius,

pos + new Vector2((float)Math.Cos(a + step) , (float)Math.Sin(a + step))\* radius, color);

}

}

// 更新动态顶点缓存

public static void UpdateVertexBuffer()

{

int vertexCount = vertList.Count;

if (vertexCount > 0)

{

if (vertexBuffer.VertexCount < vertexCount)

{

vertexBuffer.Dispose();

vertexBuffer = new DynamicVertexBuffer(graphics, VertexPositionColor.VertexDeclaration,

vertexCount, BufferUsage.None);

}

vertexBuffer.SetData(vertList.ToArray(), 0, vertexCount, SetDataOptions.Discard);

lineCount = 0;

vertList.Clear();

}

}

public static void Draw()

{

LineEffect.CurrentTechnique.Passes[0].Apply();

if (lineCount > 0)

{

graphics.SetVertexBuffer(vertexBuffer);

graphics.DrawPrimitives(PrimitiveType.LineList, 0, lineCount);

}

}

}

由于在今后的程序中我们会绘制几百个三角形，因此一个一个绘制三角形是效率极低的方法，更好的做法是在一个批次中绘制所有线段，由于三角形的位置坐标会发生频繁的变动，因此使用的是动态顶点缓存而不是通常的顶点缓存。

使用的自定义Effect文件LinearRendering.slfx代码如下：

float screenWidth;

float screenHeight;

struct VertexOutput

{

float4 PositionPS : SV\_Position;

float4 Color : COLOR0;

};

VertexOutput LineRendering2DVS(float4 Position:SV\_Position,float4 Color:COLOR0)

{

VertexOutput vout;

// 坐标的映射

Position.x=-1.0+Position.x\*2/screenWidth;

Position.y=1.0-Position.y\*2/screenHeight;

vout.PositionPS = Position;

vout.Color = Color;

return vout;

}

float4 LineRendering2DPS(float4 Color:COLOR0) : SV\_Target0

{

return Color;

}

technique LineRendering2D

{

pass PassFor2D

{

VertexShader = compile vs\_2\_0 LineRendering2DVS();

PixelShader = compile ps\_2\_0 LineRendering2DPS();

}

}

其实使用框架自带的BasicEffect也可，但对于绘制如此简单的2D线条来说就显得大材小用了。

这个自定义的Effect要简单得多，注意顶点着色器中是如何映射顶点坐标的。因为屏幕空间的左下角为(-1.-1)，右上角为(1,1)，而我们习惯的2D坐标系坐标原点在左上角，右下角坐标为(屏幕宽度,屏幕高度)，因此在程序中指定的是2D坐标系，需要在shader中将它们转换到屏幕坐标系。