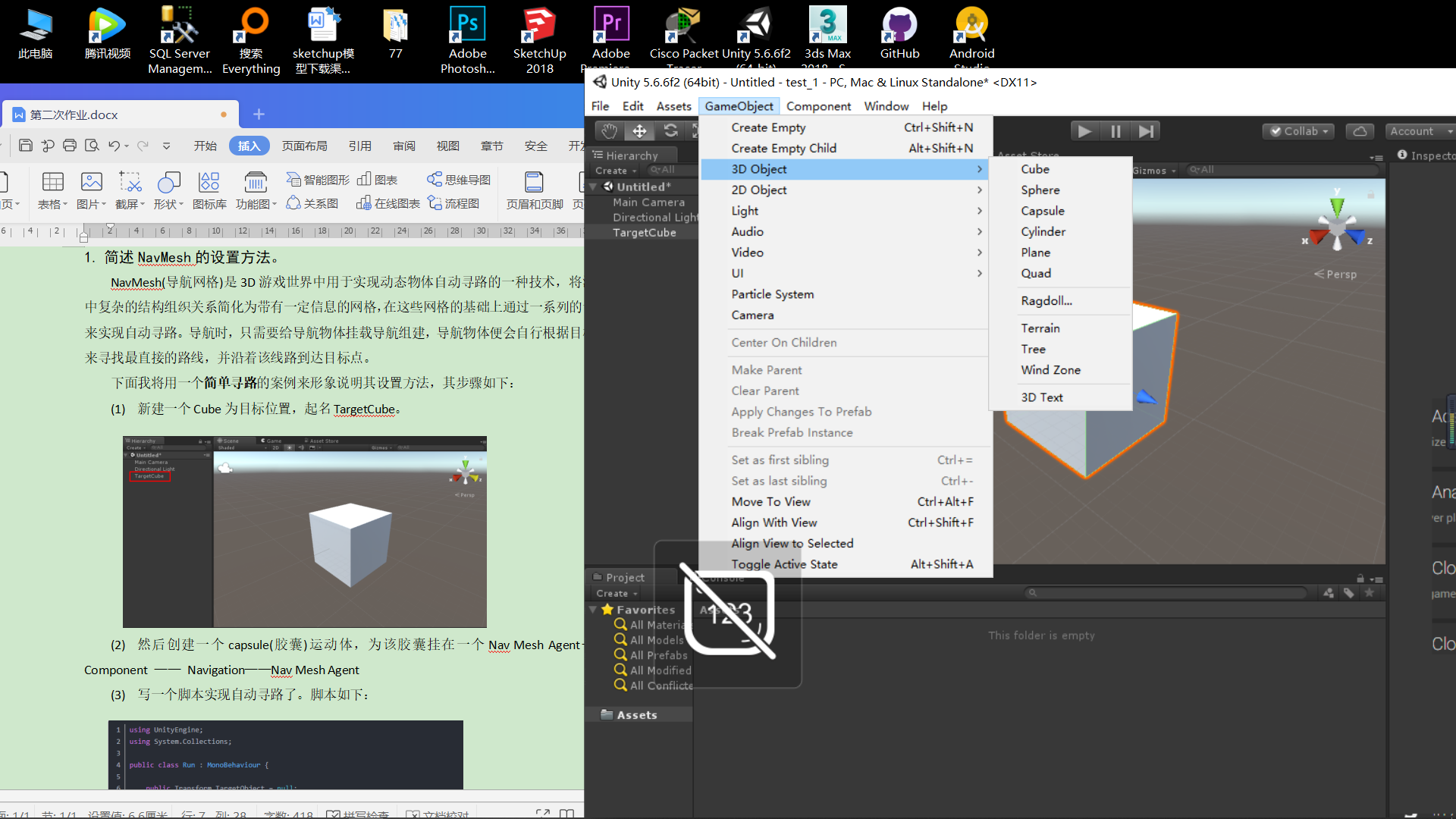
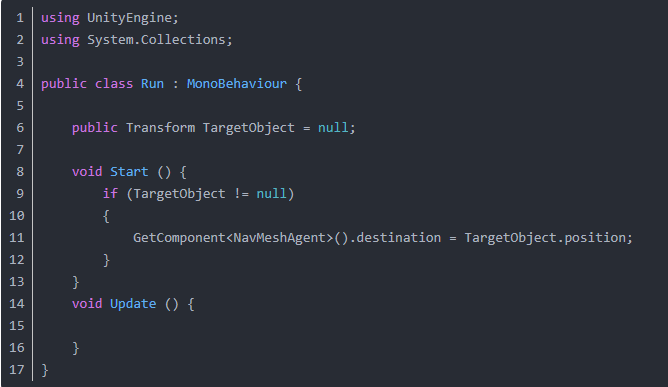
1. 简述NavMesh的设置方法。

NavMesh(导航网格)是3D游戏世界中用于实现动态物体自动寻路的一种技术，将游戏中复杂的结构组织关系简化为带有一定信息的网格，在这些网格的基础上通过一系列的计算来实现自动寻路。导航时，只需要给导航物体挂载导航组建，导航物体便会自行根据目标点来寻找最直接的路线，并沿着该线路到达目标点。

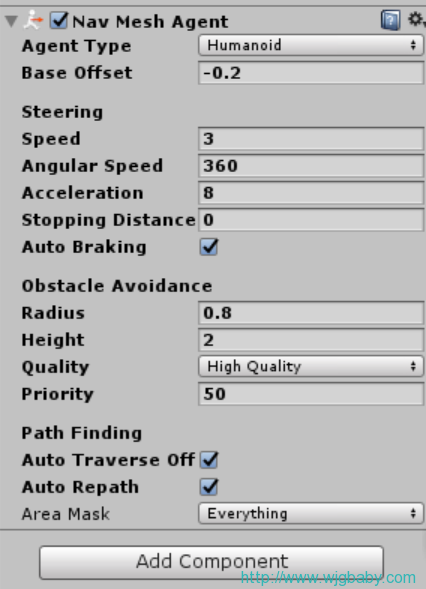
下面我将用一个**简单寻路**的案例来形象说明其设置方法，其步骤如下：

1. 新建一个Cube为目标位置，起名TargetCube。
2. 然后创建一个capsule(胶囊)运动体，为该胶囊挂在一个Component — Navigation—Nav Mesh Agent
3. 写一个脚本实现自动寻路了。脚本如下：
4. 将脚本挂载到胶囊体上
5. 将TargetCube赋予给胶囊体的Run脚本，运行场景。胶囊体会按照箭头的方向运动到Cube位置。

这样一个简单的自动寻路就完成了，如果要更精细的寻路，可根据相关参数进行调整。

1. 简述NavMeshAgent属性参数的使用方法。
2. Base Offset：碰撞模型和实体模型之间的垂直偏移量。
3. Speed：物体的行进最大速度
4. Augular Speed：行进过程中转向时的角速度。
5. Acceleration：物体的行进加速度
6. Stopping Distance：离目标距离还有多远时停止。
7. Auto Braking在行进某些原因中断后是否重新开始寻路。
8. Radius：物体的半径
9. Height：物体的高度。
10. quality:质量
11. priority:优先级
12. Auto Traverse Off Mesh Link：是否采用默认方式度过链接路径。
13. Auto Repath：在行进某些原因中断后是否重新开始寻路。
14. 简述寻路过程中路网烘培过程。

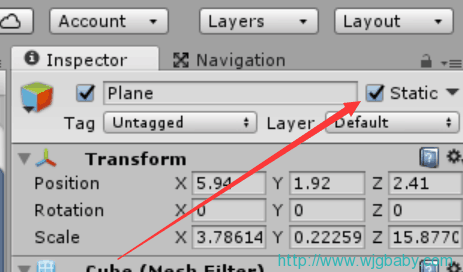
几个常用的属性：



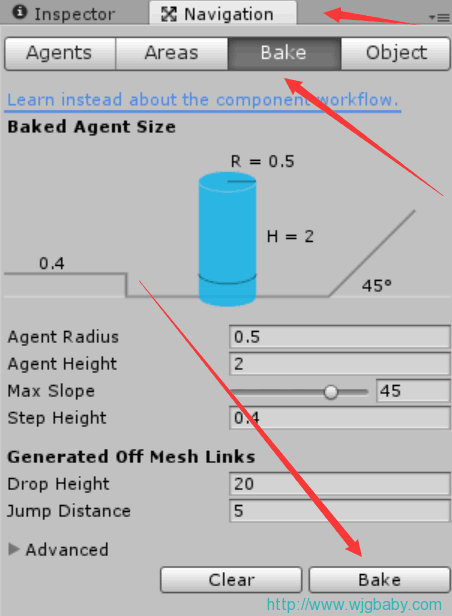
1. Radius[半径]：控制代理器这个“圆柱体”的半径；
2. Height[高度]：控制代理器这个“圆柱体”的高度；
3. Base Offset[基础偏移]：控制代理器这个“圆柱体”在垂直方向的偏移量；
4. Speed[速度]：导航的移动速度；
5. Angular Speed[转弯速度]：模型是Y 轴朝向目标点；如果不是，则转向；
6. Acceleration[加速度]：保持默认即可；
7. Stopping Distance[停止距离]：距离目的地多远的时候停止；
8. Auto Braking[自动停止]：保持默认勾选状态即可；

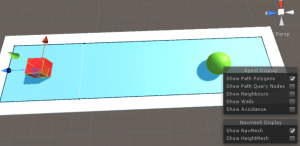
下面开始实战：

1. 新建一个plane，设置为static，即静态物体



1. 打开Window下的Navigation面板，选中Bake，点击Bake，烘焙导航场景，可以看到Plane上加了一片蓝颜色区域，这就是可移动区域。



1. [](http://wu.wjgbaby.com/wp-content/uploads/2017/08/17080462.png)新建一个Cube放在Plane上面，给Cube添加NavMeshAgent组件，新建一个Sphere，也放在Plane上面。
2. 新建一个C#脚本PlayerGo放在我们的Cube身上
3. 对于寻路过程中的障碍物绕行应该怎样处理？

常用的有以下几种方法：

1. **重新计算路径**

我们希望游戏世界随着时间改变。一条一段时间之前发现的路径，可能不再是现在的最优路径。用新的信息更新旧路径是值得的。下面列出的是一些用来判断决定是否需要重新计算路径的标准：  
　　每N步一次：这样保证用来计算路径的信息不会旧于N步。  
　　当额外的CPU时间可用时：这样可以实现路径质量的动态调整。即使使用了更多的游戏单位，或者是在一台较慢的电脑上运行游戏，每个游戏单位的CPU使用率都可以降低。  
　　当游戏单位转弯或者通过一个关键路径点的时候。  
　　当游戏单位附近的世界发生改变的时候。  
　　重新计算路线的主要缺点在于有很多路径信息被丢弃了。例如，如果路径长100步，并且每十步进行一次重新计算，那么路径的总步数是100+90+80+70+60+50+40+30+20+10 = 550。对于一个长M步的路径，总共大概进行了M^2步计算。因此，如果你想要得到很多条长路径，重新计算路线并不是一个好主意。重复使用路线信息，而非丢弃，这样会是更好的办法。

1. **路径剪接**

当一条路径需要被重新计算时，意味着世界正在改变。给定一个变化中的世界，地图上的邻近部分比远处的部分更好了解。  
　　路径剪接比重新计算路径明显地快了许多，但它对于路径的重大改变并不能很好地应对。不过很多这种情况可以容易地发现，并直接使用重新计算路径来代替路径剪接。它同样有几个可以调整的变量，比如M和进行新路径的寻找的时间，所以它可以被调整为适合不同的情况（即使是在运行的时候）。但是路径剪接并不能处理游戏单位需要确定位置进而来互相穿过的情况。

1. **监视地图的改变**

　 选择重新计算全部或部分路径在特定的时间间隔，是对地图的改变来触发重新计算。地图可以分成不同的区域，每个游戏单位可以在特定的区域表现出兴趣。（所有包括部分路径的区域都可能是感兴趣的，或者仅仅是邻近的包含部分路径的区域）无论障碍进入或离开某个区域，那个区域就标记为已经改变，然后所有对那个区域感兴趣的游戏单位都会被通知，所以路径可以在考虑障碍发生变化这一前提下被重新计算。  
　　这一技术有很多可能的变化。例如，我们可以仅仅在特定的时间间隔通知游戏单位而不是立即通知。并且多次改变可以被组合成一次通知，所以不再需要过多的进行重新计算路径。另一个变化是让游戏单位来查询地区的状态，而不是让地区来通知游戏单位。  
　　监视地图的改变，避免了游戏单位在障碍物没有发生变化的时候进行重新计算。

1. **预测障碍物移动**

如果障碍物的移动可以被预测，那就可以在进行寻路时把未来的障碍物位置纳入考虑。像A\*这类算法有一个代价函数，来决定通过地图上某点的困难程度。A\*可以被修改成实时更新到达一点所需要的时间（由当前的路径长度决定），这个时间也可以被传入代价函数中。代价函数就可以把时间纳入考虑，然后就可以使用在那个时刻的障碍物预测位置，来决定那个地图位置是否无法通过。但是这个修改并不完美，因为它不会考虑在某个点等待障碍物离开路径的可能性，另外A\*并不是设计用来区分相同路线上的路径，而是时间不同的点。

**具体实现代码如下：**

1、获取运动数据

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

// <summary>

//运动体 ，职责：定义运动数据

//</summary>

public class Vehicle : MonoBehaviour

{ //操控对象容器

    [HideInInspector]

    public Steering[] steerings;

    //操控合力

    protected Vector3 steeringForce;

    //是否在平面

    public bool isplane = true;

    //质量

    public float mass = 1f;

    //加速度

    protected Vector3 acceleratedSpeed;

    //最高速度

    public float maxSpeed;

    //操控力的上线

    public float maxForce;

    [HideInInspector]

    public Vector3 velocity;

    //合力计算的间隔时间

    public float intervalComputerForce = 0.2f;

    public void Start()

    {

        //取得运动体上所有的操控对象

        steerings = GetComponents<Steering>();

        //按时间间隔计算操控合力

        InvokeRepeating("ComputerFinalSteeringForce", 0, intervalComputerForce);

    }

    //计算合力

    public void ComputerFinalSteeringForce()

    {

        steeringForce = Vector3.zero;

        //循环所有的操控组件，产生操控合力(每个单一操控的叠加)

        foreach (var item in steerings)

        {

            steeringForce += item.GetForce() \* item.weight;

        }

        //控制操控力的上限

  steeringForce=Vector3.ClampMagnitude(steeringForce, maxForce);

        if (steeringForce == Vector3.zero)

        {

            velocity = Vector3.zero;

        }

        //根据质量计算运动算需要的加速度

        acceleratedSpeed = steeringForce / mass;

    }}

**2、处理运动数据**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

//<summary>

//运动控制类

//</summary>

public class LocomtionController : Vehicle

{

    //转向速度

    public float rorationSpeed = 0.5f;

    //移动组件

    [HideInInspector]

    public CharacterController ch;

    //动画组件

    [HideInInspector]

    public Animator anim;

    //位移(操控力)

    public void Start()

    {

        base.Start();

        ch = GetComponent<CharacterController>();

        anim = GetComponent<Animator>();

    }

    public void Update()

    {

        Movement();

        Rotation();

        PlayAnim();

    }

    public void Movement()

    {

        //当前速度+加速度

        velocity += acceleratedSpeed \* Time.deltaTime;

        //当前速度不要超上限

        if (velocity.magnitude > maxSpeed)

        {

            velocity = velocity.normalized \* maxSpeed;

        }

        //是否是平面

        if (isplane) velocity.y = 0;

        //移动

        if (ch != null)

        {

            ch.SimpleMove(velocity);

        }

        else

        {

            transform.position += velocity \* Time.deltaTime;

        }

    }

     //动画

    public void PlayAnim()

    {

        if (anim != null)

        {

        }

    }

    //转向

    public void Rotation()

    {

        if (velocity != Vector3.zero)

        {

Quaternion targetRoration = Quaternion.LookRotation(velocity, Vector3.up);

transform.rotation = Quaternion.Lerp(transform.rotation, targetRoration, rorationSpeed);}

    }

}

**3、运动物体基类**

using System.Collections;

using System.Collections.Generic;

using UnityEngine;

//<summary>

//运动体

//</summary>

public class Steering : MonoBehaviour

{

    //目标

    public Transform target;

    public Vector3? targetPosition = null;

    //希望速度

    protected Vector3 expectationVelocity;

    //最大速度(逃跑，徘徊的速度可能不同)

    public float maxSpeed;

    //运动体

    public Vehicle m\_vehicle;

    //权重

    public float weight=1;

    public void Start()

    {

        m\_vehicle = GetComponent<Vehicle>();

        if (m\_vehicle != null && maxSpeed == 0)

        {

            maxSpeed = m\_vehicle.maxSpeed;

        }

    }

    //<summary>

    //实现具体操控的算法

    //</summary>

    //<returns></returns>

    public virtual Vector3  GetForce()

    {

        if (target != null)

        {targetPosition = target.position;  }

         return Vector3.zero;  }}

**4、运动物体**

 using System.Collections;

using System.Collections.Generic;

using UnityEngine;

[RequireComponent(typeof(LocomtionController))]

public class SteeingForColliderObstacle : Steering

{ //触角的长度

    public float maxSeeAhead = 5f;

    //障碍物所在的层

    public LayerMask mask;

    //推力的放大系数

    public float expandRate = 10f;

    //射线发射点

    public Transform sendPos;

    //碰撞物体的中心

    public Transform PusnPos;

    public void Start()

    {  base.Start();

        if (m\_vehicle != null)

        {

            if (expandRate > m\_vehicle.maxForce)

            {

                expandRate = m\_vehicle.maxForce;   } } }

    public override Vector3 GetForce()

    {   base.GetForce();

       expectationVelocity = Vector3.zero;

        //检查障碍物(射线)

       RaycastHit hit;

       if (Physics.Raycast(sendPos.position, transform.forward,out hit, maxSeeAhead, mask))

       {

           //产生推力

           expectationVelocity = hit.point - PusnPos.position;

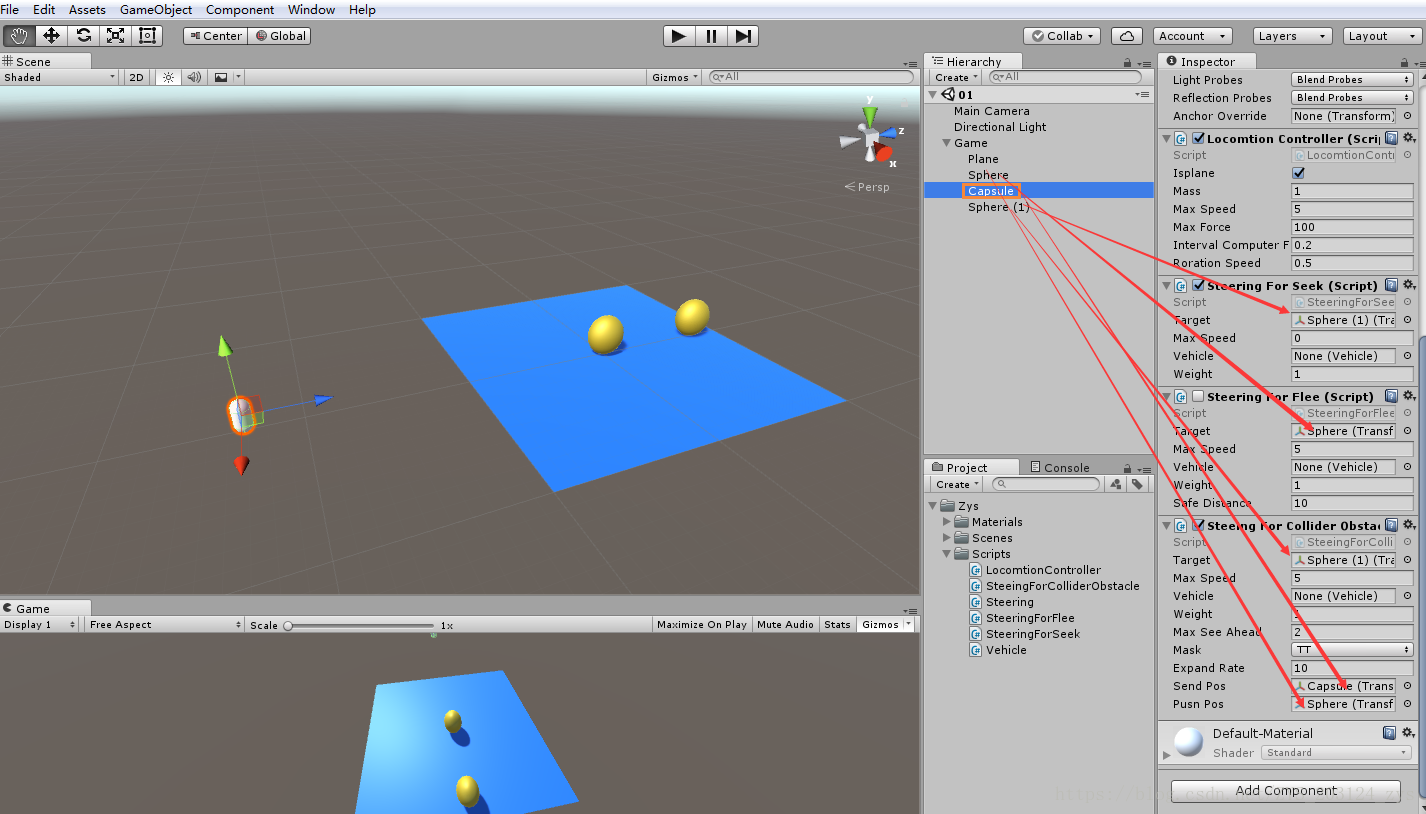
           //放大推力

           expectationVelocity \*= expandRate;  }

        //形成实际的操控力(实际速度)

       return expectationVelocity;  }}

1. 场景展示



1. NavMeshAgent组件的代理器移动到给定目标点需要利用哪个函数？该函数有几个参数？其含义分别是什么？



需要利用的函数：SetDestiantion函数；

该函数参数：便是你的目标点的位置；

含义：将物体移动到目标点所在的位置。