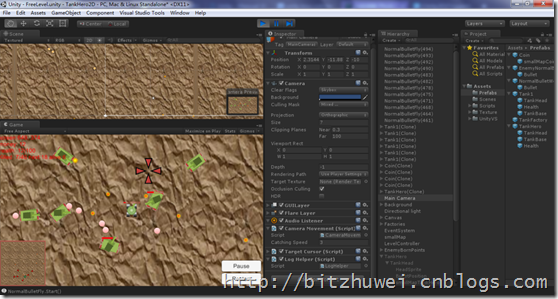
自制Unity小游戏TankHero-2D(4)关卡+小地图图标+碰撞条件分析

我在做这样一个坦克游戏，是仿照（<http://game.kid.qq.com/a/20140221/028931.htm>）这个游戏制作的。仅为学习Unity之用。图片大部分是自己画的，少数是从网上搜来的。您可以到我的github页面（<https://github.com/bitzhuwei/TankHero-2D>）上得到工程源码。

[](file:///C:\Users\DELL\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles119413CA\image%5b2%5d.png)

本篇主要记录关卡解析器、小地图图标和对碰撞的原理的探索，需要耐心分析。

# 关卡解析器

在一个关卡里，敌方坦克应该是一波一波地出现，每波敌人出现多少个，每个敌人是什么类型的坦克、出现在什么位置都应该是可配置的。这需要一个关卡解析器，把如下的文字解析为一个数据结构Level。

|  |
| --- |
| level  {  tank{0 0} |  tank{0 1} |  tank{0 2} |  tank{0 0} tank{0 1} tank{0 2} |  tank{0 0} tank{0 1} tank{0 2} tank{0 0} tank{0 1} tank{0 2} |  tank{0 0} tank{0 1} tank{0 2} tank{0 0} tank{0 1} tank{0 2} |  tank{0 0} tank{0 1} tank{0 2} tank{0 0} tank{0 1} tank{0 2} tank{0 0} tank{0 1} tank{0 2} |  tank{0 0} tank{0 1} tank{0 2} tank{0 0} tank{0 1} tank{0 2} tank{0 0} tank{0 1} tank{0 2} tank{0 0} tank{0 1} tank{0 2}  } |

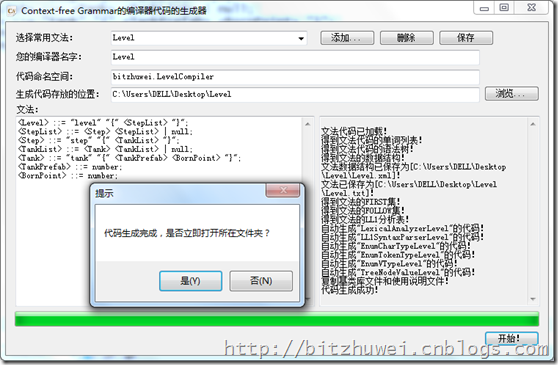
这段文字的意思是，第一波敌人是类型编号为0，出生位置编号为0的1个坦克；第二波敌人是类型编号为0，出生位置编号为1的1个坦克；。。。

这种东西我喜欢用编译原理解决，因为我在（<https://github.com/bitzhuwei/CGCompiler.git>）有一个自己写的自动生成词法、语法分析器的工具。关于这个工具的介绍可参考我博客里关于编译原理的文章（[在这里搜索“编译器”](http://www.cnblogs.com/bitzhuwei/p/bitzhuwei-cnblogs-com-index.html)）。

先总结一下关卡的文法

|  |
| --- |
| <Level> ::= "level" "{" <StepList> "}";  <StepList> ::= <Step> <StepList> | null;  <Step> ::= "step" "{" <TankList> "}";  <TankList> ::= <Tank> <TankList> | null;  <Tank> ::= "tank" "{" <TankPrefab> <BornPoint> "}";  <TankPrefab> ::= number;  <BornPoint> ::= number; |

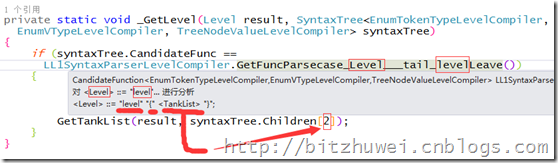
然后用工具生成词法语法解析器代码。

[](file:///C:\Users\DELL\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles119413CA\image%5b5%5d.png)

剩下的就是自己写一下从语法树到数据结构的转换。代码如下。

|  |
| --- |
| public static Level GetValue(this SyntaxTree<EnumTokenTypeLevelCompiler,  EnumVTypeLevelCompiler, TreeNodeValueLevelCompiler> syntaxTree)  {  if (syntaxTree == null) { return null; }  var result = new Level();  \_GetLevel(result, syntaxTree);  return result;  }  private static void \_GetLevel(Level result, SyntaxTree<EnumTokenTypeLevelCompiler, EnumVTypeLevelCompiler, TreeNodeValueLevelCompiler> syntaxTree)  {  if (syntaxTree.CandidateFunc == LL1SyntaxParserLevelCompiler.GetFuncParsecase\_Level\_\_\_tail\_levelLeave())  {  GetTankList(result, syntaxTree.Children[2]);  }  }  private static void GetTankList(Level level, SyntaxTree<EnumTokenTypeLevelCompiler, EnumVTypeLevelCompiler, TreeNodeValueLevelCompiler> syntaxTree)  {  if (syntaxTree.CandidateFunc == LL1SyntaxParserLevelCompiler.GetFuncParsecase\_TankList\_\_\_tail\_tankLeave()  || syntaxTree.CandidateFunc == LL1SyntaxParserLevelCompiler.GetFuncParsecase\_TankList\_\_\_tail\_or\_Leave())  {  var egg = GetTank(syntaxTree.Children[0]);  level.Add(egg);  GetTankList(level, syntaxTree.Children[1]);  }  else if (syntaxTree.CandidateFunc == LL1SyntaxParserLevelCompiler.GetFuncParsecase\_TankList\_\_\_tail\_rightBrace\_Leave())  {  //nothing to do  }  }  private static TankEgg GetTank(SyntaxTree<EnumTokenTypeLevelCompiler, EnumVTypeLevelCompiler, TreeNodeValueLevelCompiler> syntaxTree)  {  if (syntaxTree.CandidateFunc == LL1SyntaxParserLevelCompiler.GetFuncParsecase\_Tank\_\_\_tail\_tankLeave())  {  var tankPrefab = GetTankPrefab(syntaxTree.Children[2]);  var bornPoint = GetBornPoint(syntaxTree.Children[3]);  var result = new TankEgg(tankPrefab, bornPoint);  return result;  }  else if (syntaxTree.CandidateFunc == LL1SyntaxParserLevelCompiler.GetFuncParsecase\_Tank\_\_\_tail\_or\_Leave())  {  var result = new TankEgg(-1, -1);  return result;  }  return null;  }  private static int GetBornPoint(SyntaxTree<EnumTokenTypeLevelCompiler, EnumVTypeLevelCompiler, TreeNodeValueLevelCompiler> syntaxTree)  {  if (syntaxTree.CandidateFunc == LL1SyntaxParserLevelCompiler.GetFuncParsecase\_BornPoint\_\_\_numberLeave())  {  var result = int.Parse(syntaxTree.Children[0].NodeValue.NodeName);  return result;  }  return 0;  }  private static int GetTankPrefab(SyntaxTree<EnumTokenTypeLevelCompiler, EnumVTypeLevelCompiler, TreeNodeValueLevelCompiler> syntaxTree)  {  if (syntaxTree.CandidateFunc == LL1SyntaxParserLevelCompiler.GetFuncParsecase\_TankPrefab\_\_\_numberLeave())  {  var result = int.Parse(syntaxTree.Children[0].NodeValue.NodeName);  return result;  }  return 0;  } |

在VS2013里你可以获得Tip，便于coding。

[](file:///C:\Users\DELL\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles119413CA\image%5b8%5d.png)

解析器写好了，调用方式如下。

[](file:///C:\Users\DELL\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles119413CA\image%5b11%5d.png)

语法分析器的类型太长，只好用上图表示一下。

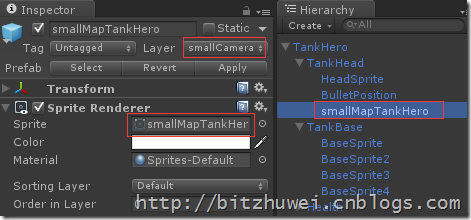
这样就有敌方坦克一波一波来袭的感觉了。

# 小地图图标

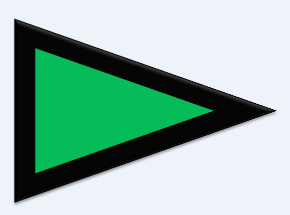
小地图上显示的坦克很不清晰，如果能显示出一个鲜艳的三角形就好了，尖头指向开炮的方向。如下图所示，绿色的为玩家坦克，红色的为敌方坦克。



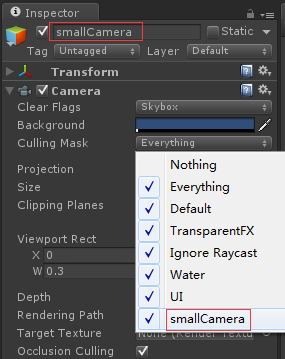
首先给坦克的prefab增加一个显示箭头的子对象。

[](file:///C:\Users\DELL\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles119413CA\image%5b14%5d.png)

给子对象smallMapTankHero增加Sprite Renderer组件，在组件的Sprite属性里赋予下面的图片，并把此对象的Layer属性设置为自定义的”smallCamera”（Layer的名字无所谓）。

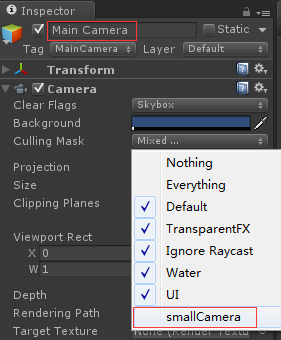


Hierarchy里的smallCamera对象是用来显示小地图的。设置其Culling Mask属性如下图所示，勾选smallCamera。这样，箭头就会显示在smallCamera里。

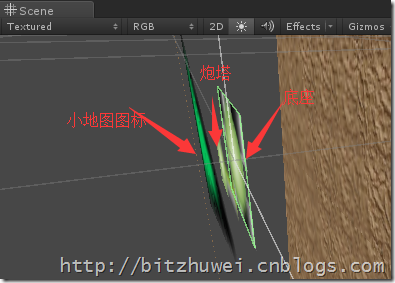


相应的，在主摄像机Main Camera里，设置Culling Mask属性如下图所示，取消勾选smallCamera。这样，箭头就不会显示在smallCamera里。

有点平行世界异次元的感觉。



小地图里仍旧会显示原有的坦克贴图，为了挡住坦克贴图，我们把小地图图标向上（靠近摄像机的方向）移动一点点。

[](file:///C:\Users\DELL\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles119413CA\image%5b20%5d.png)

当然，也可以通过把坦克对象的Layer设置为一个自定义的Layer（比如自定义为realworld），然后在smallCamera的Culling Mask属性中取消勾选realworld即可。不过这还需要把各种对象都放到自定义的Layer里去，太麻烦了。

# 碰撞

Unity中的碰撞，有Collision（物理碰撞）和Trigger（触发器碰撞）两种。

这里我用精心设计的试验分析出了碰撞的产生条件。

## 触发碰撞的基本条件

下面，假设有两个Hierarchy中的对象A和B，A和B都有Collider组件和Rigidbody组件。那么：

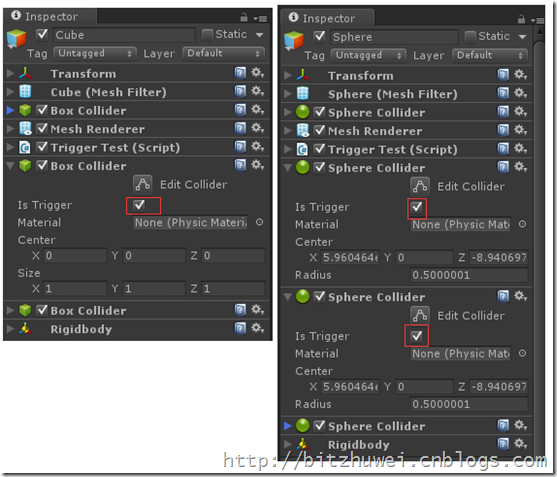
如果去掉其中任何一个的Collider，那么不会发生碰撞事件（只会穿透）。

如果去掉B的rigidbody，那么移动B去撞A时，不会发生碰撞事件（只会穿透）。

如果去掉B的rigidbody，那么移动A去撞B时，Collision和Trigger的碰撞都可以发生在AB身上，但B不会受物理引擎影响而移动。

## OnTriggerXXX的触发原则

下面，再假设A是Cube，B是sphere；A有3个Box Collider，设置第2个Box Collider的Is Trigger为true；B有4个Shpere Collider，设置第2、3个Sphere Collider的Is Trigger为true。

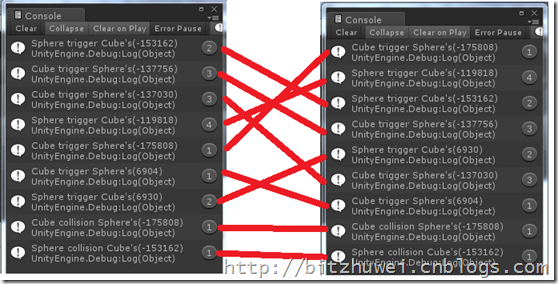
[](file:///C:\Users\DELL\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles119413CA\image%5b23%5d.png)

然后，给A和B分别添加如下的脚本组件：

|  |
| --- |
|  |
| public class TriggerTest : MonoBehaviour  {  // Use this for initialization  void Start()  {  }  // Update is called once per frame  void Update()  {  }  void OnTriggerEnter(Collider other)  {  Debug.Log(string.Format("{0} trigger {1}'s({2})", this.name, other.name, other.GetInstanceID()));  }  void OnCollisionEnter(Collision collision)  {  Debug.Log(string.Format("{0} collision {1}'s({2})", this.name, collision.transform.name, collision.collider.GetInstanceID()));  }  } |

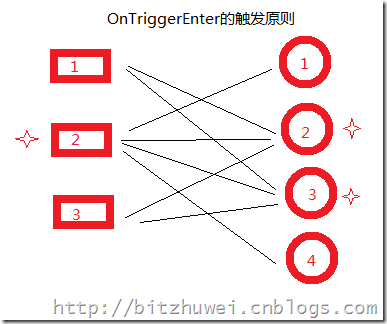
这样就可以记录自己碰撞了对方的哪个Collider。

我们先让A撞B，再让B撞A，分别得到如下图左右所示的结果。

[](file:///C:\Users\DELL\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles119413CA\image%5b26%5d.png)

分析发现，虽然引发的碰撞事件的先后顺序有所不同，但碰撞事件是相同的那16个Trigger事件和2个Collision事件。

据此我总结出Trigger事件的触发原则，如下图所示。

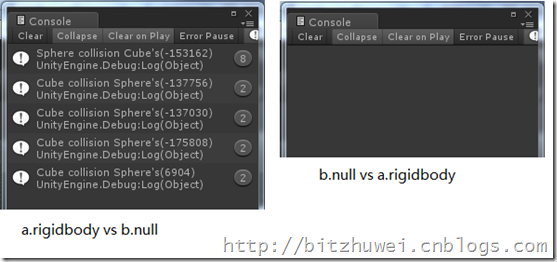
[](file:///C:\Users\DELL\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles119413CA\image%5b29%5d.png)

当A的第2个Collider设置为Is Trigger=true时，此Collider会与B的各个Collider都引发一次Trigger碰撞事件。B也同理。不过两边均为Is Trigger=true时，只会引发一次。数一下上图，可以看到有8条线，每条线在AB两方各引发一次Trigger碰撞事件，所以就是上文的16次。另外，每个Collider引发的次数也与上文的图示吻合。

## OnCollisionXXX的触发条件

下面，我把A和B的所有Collider的Is Trigger都设为false，来研究Collision的触发条件。

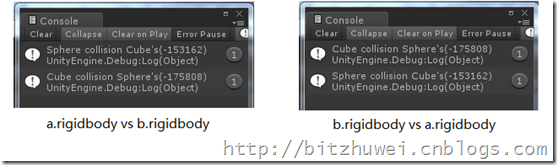
下图展示了去掉B的Rigidbody后，分别用A撞B和用B撞A的情况。

[](file:///C:\Users\DELL\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles119413CA\image%5b32%5d.png)

可以看到，没有Rigidbody的B撞A，什么都不会发生，B直接穿透了A。

再用含有2个Collider的Cube和含有2个Collider的Sphere试验（此处略）就可以知道，A撞B则会使A的第1个Collider依次与B的各个Collider引发Collision碰撞事件。所以上图左侧会有1个A的Collider与4个B的Collider引发的Collision事件。

继续看下图，是同时具有Rigidbody的A和B相互碰撞的结果。

[](file:///C:\Users\DELL\AppData\Local\Temp\WindowsLiveWriter1286139640\supfiles119413CA\image%5b35%5d.png)

可以看到A和B都只有1次Collision事件。

根据上述试验，我认为Unity3D引擎在处理Collision事件时，是让Collider找Rigidbody，如果找到了就让它执行物理引擎，然后此次碰撞就告结束，不再引发此次A和B的Collision事件。

# 总结

您可以到我的github页面（<https://github.com/bitzhuwei/TankHero-2D>）上得到工程源码。

请多多指教~