基于 Unity3D 的 Dots 寻路算法

王 霞[1]
WANG Xia

摘 要 自动寻路 A* 算法的时间复杂度和空间复杂度较高，且对资源消耗较大。通过在 Unity3D 引擎中结合 Dots 面向数据的技术堆栈技术，采用 ECS+JOBS 的工作模式，将过程多线程化，使用 JobSystem+Job 多线程运行逻辑，对自动寻路的 A* 算法进行改进，采用修改地图点密度的方法来降低计算量，性能节省非常明显。为了同时考虑性能问题和寻路表现，再次改进算法，根据寻路对象的个数来动态地改变点与点之间的间隔，并比较两种不同的方案触发寻路，选出最优方案，有效缩短寻路时间，项目可以稳定运行数千个实体角色。

关键词 自动寻路；A*；单核多线程；ECS；Dots

doi：10.3969/j.issn.1672-9528.2024.05.030

0 引言

Unity 中自动寻路功能模块采用高效的 A* 算法，目前很多游戏的寻路功能底层大部分都是采用 A* 算法 [1]。把游戏世界某块区域的地图数据化，每个点即作为一个对象，存储着当前的位置状态 *F*、*G*、*H* 和上一个路径点等属性。其中，*F*=*G*+*H*[2]，*G* 为寻路对象的起点到当前点的一个预估消耗，可理解为权重，该值随寻路对象的位置变化而变化；*H* 表示该点到寻路目标的一个固定的消耗值，与 *G* 值不同的是，*H* 是一开始计算完成就不会改变。将开始节点放入到 open 列表，然后在循环中得到一个最小的 *F* 值，选取这个点。放入 close 列表，表示已经确定属于路径上的一点，选取它的 8 个方向点，筛选出不存在 close 列表的点且没有障碍物的点来放入 around 列表中。遍历这些周围点，如果当前的点到这个周围点的 *G* 值比之前的 *G* 值要小，就要更新 *G*、*F* 值和上一个路径点。最后加入开启列表，进入下一次的循环。剩下的以前面的方式继续迭代，直到 open 列表中出现终点为止。

在 A* 算法基础上，国内学者提出了一些改进和优化。比如，钟瑛提出使用二叉堆进行优化 OPEN 表结构 [3]，缩短搜索节点的耗时；周小镜提出运用将单个物体路径搜索和算法相结合的分级路径搜索方法来搜索寻路 [4]；陈素琼提出采用向量夹角余弦值作为启发函数来减少考察无用节点数量等 [5]。理论上来说，A* 算法能扩展较少的节点，最快地找到答案，但 A* 算法是对资源空间消耗很大的算法 [6]。学者们研究的是二维地图的寻径算法的改进，而三维地图的路径搜索各方面更为复杂 [7]。在这里使用 Dots 多线程开发，使游戏在多核处理器上运行得更快，节省性能。

1 Dots 框架下的寻路系统

A* 算法的时间复杂度和空间复杂度都是蛮高的，游戏中有成百上千的敌人，那么就不可能用一般的方法来进行寻路，将坐标世界数据化。每一个点看作一个对象存放着该点的指标和父节点，利用 A* 算法，每次运算时得到一条没有障碍的最优解。普通的组件式开发，光就一帧生成几百的游戏对象，都会造成程序卡死，更别提让几百只敌人同时寻路了。在这里使用 Dots 多线程开发，Dots 是 Unity 游戏引擎中基于 ECS 架构开发的一套包含 Burst Complier 技术和 JobSystem 技术面向数据的技术栈 [8]。Dots 可使出色的游戏在多核处理器上更快运行，而不带来繁重的编程负担。为什么 Dots 可以使游戏在多核处理器上运行得更快，关键核心在 ECS。ECS 即实体（Entity）、组件（Component）、系统（System）[9]，其中 Entity、Component 皆为纯数据向的类，System 负责操控它们，这种模式会一定程度上优化代码速度 [10]，同时提升代码的可读性、易维护性和延展性。

ECS 面向数据的编程思想比面向对象的编程思想更加契合，架构在执行逻辑时，只会操作需要操作的数据，而 E 和 C 这两者的配合把相关数据紧密排列在一起，并且通过 Fliter 组件过滤掉不需要的数据，这样就减少了缺页中断次数，且 System 只会对含有指定数据的 Entity 进行操作，这

1. 福建船政交通职业学院信息与智慧交通学院
福建福州 350002

[基金项目] 职业院校服务全民终身学习实践研究课题（2022001ZZ52）

在整体上提高了程序效率。ECS 会在内存中对带有相同组件（Component）集的所有实体（Entity）进行组合。ECS 把这类组件集称为原型，每个原型都有一个 Chunks 块列表，用来保存原型的实体。在通过 System 对 Entity 进行操作时，会循环所有块，并在每个块中对紧凑的内存进行线性循环处理，以读取或写入组件数据。

得益于 ECS 的优点，所有的组件都是通过结构体的方式创建直接在栈中开辟空间，不需要系统再去堆中申请空间和 GC 回收。由此，使每个敌人的组件都可以从栈中快速得到，且寻路运算可以分布在不同的线程，可以实现万人同时寻路的效果。

2 修改地图点密度的寻路

使用 Dots 多线程开发，可以实现万人同时寻路，但随着游戏中寻路对象的增多，性能消耗开销愈发明显，如 100×100 的地图，单就一个敌人最高可能需要执行 1 万次才能得到结果。于是，采用修改地图点密度的方法来降低计算量，即将之前每一个单位距离一个点的方案改为 x 个单位距离一个点为一个地图点，性能节省非常明显。三个单位的间隔就能多出上千个寻路对象。比如，起初 800 个寻路对象同时寻路出现掉帧，采用方案后 2000 个寻路对象还基本保持在 50 帧以上。对比分析图如图 1 ～图 4 所示。

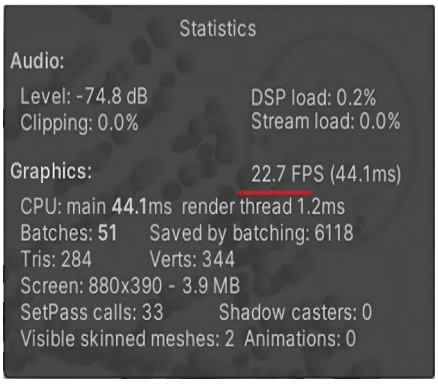

图 1 一个单位间隔的帧率图

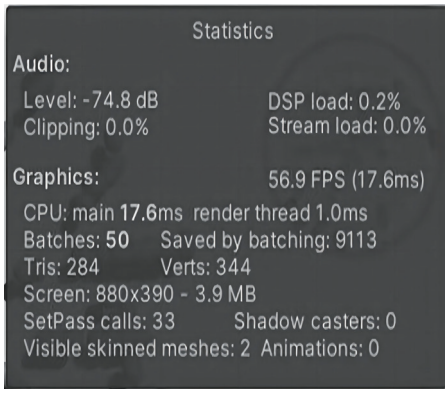

图 2 三个单位间隔的帧率图

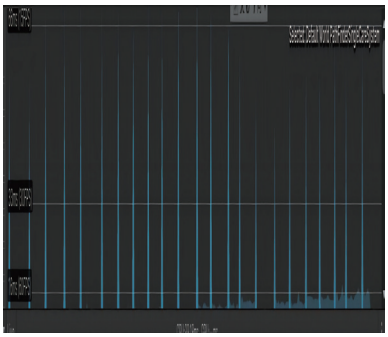

图 3 一个单位的峰值图

图 4 三个单位间隔的帧率峰值图

通过上面的比较，可以明显看出前后性能的提升。三个单位帧率更稳定，峰值比前者低，通过峰值图的横向也可以看出两者算法的调用量的差别。

这个方案虽然解决了性能问题，但是寻路对象的跨步减少了，走的路径也就更笔直了。也就是说，寻路对象的寻路表现没有小单位间隔点的寻路表现好。为了同时考虑性能问题和寻路表现，再次改进算法，在游戏中根据寻路对象的个数来动态地改变点与点之间的间隔。这样就能实现寻路对象少的时候，表现力更强，但寻路对象过多时，只能牺牲寻路表现力来降低开销。

3 寻路触发方案

在游戏的开发中发现，如果在游戏对象生成的一瞬间就开启寻路，那么这一处的 CPU 峰值会非常高。于是，前后采用两种方案来触发寻路，对比分析图如图 5 和图 6 所示。

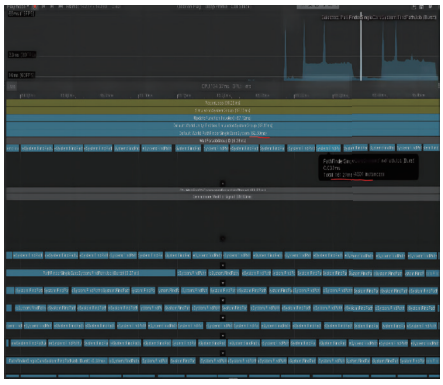

图 5 方案 1 的性能分析

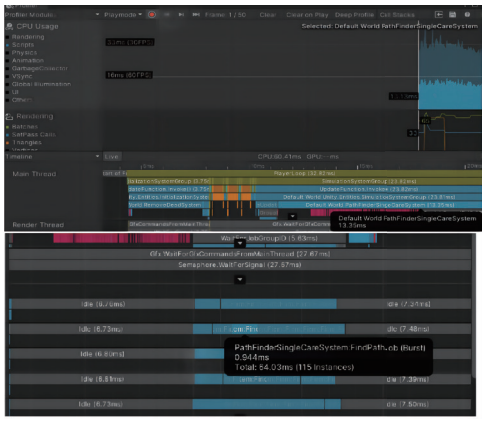

图 6 方案 2 的性能分析

方案 1 根据主角的移动距离触发寻路：

```
if(math.distance(player.Value, recordPlayerLastPosData.lastPos)>1f)
    {
     recordPlayerLastPosData.lastPos = player.Value;
    }
    else
    {
      return;
     }
```

方案 2 根据主角离寻路对象的距离触发寻路：

```
if (math.length(offset) < 16f)// 控制怪在什么范围会追击玩家
{
    if (EntityManager.HasComponent<AutoPartolTagComponent>(entity))
    {
     EntityManager.RemoveComponent<AutoPartolTagComponent>(entity);
    }
     EntityManager.AddComponentData<TargetPointData>(entity, new TargetPointData
    {
     startPosition = new int2 { x = startX, y = startY };
endPosition = target;
}
  }
```

由上分析可见，方案 2 的性能远远高于方案 1。每个个体只会在主角接近它的时候才开启寻路。5000 个敌人中，这一帧开启寻路的只有 115 个，而方案 1 里，5000 个敌人，这一帧开启寻路 4601 个。且方案 1 里 5000 个寻路对象同时寻路时，帧率为 29.7，而方案 2 的帧率还保持在 40.5 帧，如图 7 和图 8 所示。综合考虑后，方案 2 更适合本案例的场景，且性能更优。

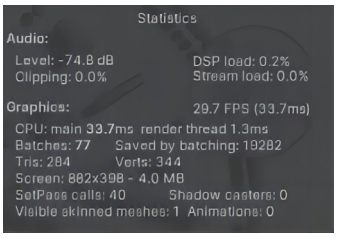

图 7 方案 1 帧率

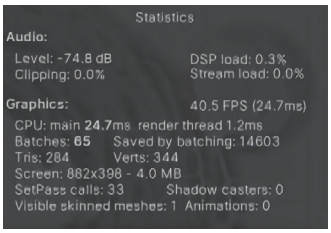

图 8 方案 2 帧率

得益于 Dots 的 JobSystem，在项目中无需关心线程冲突的问题，就可以完成多线程的开发，而且 Dots 的 Ecs 模式能更好地提高性能。通过上面两个概念，可以在 JobSystem 下通过 Entities.ForEach 根据组件、标签去遍历需要的数据对象。与以往开发不同的是，在 JobSystem 下行为和逻辑并不是依赖在游戏对象下的，而是单独把逻辑放到一个 System 类中，通过这个类，继承 JobSystem，根据组件遍历所需的对象。这样不需要对每个游戏对象都去添加行为组件，游戏对象只需要承载对应的数据即可，其他的逻辑交给 Jobsystem 处理，这就能使得不同类型游戏对象可以复用同样的逻辑，并不需要通过继承或复杂的设计模式来实现。这样降低了类的实例化，避免占用无用的字节。另外，在 Ecs 下 Component 是以值类型的形式挂载在每个游戏对象上，这是为了避免分配和 GC 花费的时间。在 JobSystem 中可以直接向栈中访问组件，通过 Ref 关键词修改栈中数据。

如下面移动转向的示例：

JobHandle jobHandle = Entities.WithAll<ZombieTag>().ForEach((DynamicBuffer<PathBuffPosition> buffPos, ref Translation translation, ref Rotation rotation, ref PointIndexData pathIndex) =>{ / / 具体代码 }。Schedule（arg）；

上面代码中使用 WithAll 去筛选包含 ZombieTag 标签的对象，然后在所有对象中再进行筛选去遍历委托中的参数（组件），最后通过 Schedule 交给 JobSystem 并行的执行代码。

在寻路系统下也是这么做的。通过组件 / 标签遍历所有的寻路对象，获取到寻路对象的当前位置和目标位置，执行 Job，得到一条路径。在 Job 算法中需要使用到在 JobSystem 中得到的地图数据，但又因为每个 Job 之间存在资源竞争，所以在 JobSystem 下不允许直接使用引用类型的原生数组来操作。为解决资源竞争关系，需要使用 Ecs 包下扩展的

Unity.Collections.NativeArray 数组，通过给每个 Job 发送一份数组的拷贝，就不会引用到主线程的数组，也就不会存在资源竞争关系。

有些场景下，必须拿到引用修改主线程的数组数据，如本案例下的寻路系统需要把得到的路径返回到主线程中，让移动系统（MoveUpdateSystem）能得到路径中的每一个点进行遍历移动。起初尝试直接在 Component 定义一个 NativeList 的成员，结果 Component 并不支持这些复杂的数据类型。为了突破这些问题，案例中使用 DynamicBuffer 来存储这些数据，并作为组件挂载到每个寻路对象上。

```
[InternalBufferCapacity(4)]
public struct PathBuffPosition: IBufferElementData
{
  public int2 pathPos;
}
```

[InternalBufferCapacity(4)] 限制动态缓存的最大容量。

通过 GetBufferFromEntity<PathBuffPosition>[Entity] 来获得到这些 Buffer 的引用。每次存放新的路径前，Clear 清空整个缓存，继续 Add 为 Buffer 添加元素，最后就可以在其他系统中访问到已经更新的路径数据。通过 ref PointIndexData PathIndex 获取到对应 DynamicBuffer 中的路径点，判断是否近似，如果近似对等，就对 PathIndex-1，并且返回给原内存地址的空间位置；否则，就继续朝向对应的路径点移动。在这个 MoveUpdate 中还存在一个值得思考的问题，就是最终目标点是否允许寻路对象重叠。在案例中，寻路对象的最终目标永远是人物的位置。为了避免人物与寻路对象发生碰撞，可以在得到的路径中不把最后一个目标点写入 DynamicBuffer。案例中是判断 PathIndex 的值是否满足定义的条件，并没有删除最后的目标点。因为通过 PathIndex 的值，就可以很方便地调整人物与寻路对象的间隔距离。

4 结语

A* 算法解决了游戏中单个非玩家角色的智能化寻路问题，成为目前游戏中使用最广泛的寻路算法，同时也在城市交通、车辆调度、镶嵌性提取、停车场寻路、士兵作战中有重大应用。A* 算法解决了游戏中单个非玩家角色的智能化寻路问题，成为目前游戏中使用最广泛的寻路算法之一。本文结合 Dots 框架，采用了 ECS+JOBS 的工作模式，通过 Jobs 将过程多线程化，并通过优化 Job 结构体中的 C# 代码，即在寻路算法中动态地改变点与点之间的间隔以及寻路触发的条件，从而减少了进程的执行时间。项目可以稳定运行数千个实体角色，大大地提升了脚本的执行速度。为了能给体验者更加极致的体验，未来的 VR 项目或许会用到大量的游戏单位模拟世界，但性能的消耗是非常关键的，而通过 Dots 可以很好地解决这个问题，通过 Dots 中的 Subscene，可以高效率地生成地图，大大提高游戏的性能，且在模拟世界中的人群、鱼群、鸟群等需要大量游戏对象的场景时，Dots 也可以轻松实现。

Dots 支持移动端应用的开发，对于未来 VR 设备的普及，势必少不了移动端的助力。通过 Dots 技术在移动端的应用，也可开发出高性能的 VR 程序，通过这项技术，甚至可以在手机上做人群的模拟、粒子群等。将来这项技术势必会更加成熟，为人们带来更加流畅的游戏体验。考虑到现在 VR 设备大多是通过手机来运行，如果将这项技术运用到手机上，原本需要电脑才可以流畅运行的一些程序，在手机上也可收放自如，甚至不会输给一些 PC 端的应用，那么，将大大提升手机 VR 模拟的体验感。

参考文献：

[1] 欧阳星昱 . 不完全标记的十五行为跟踪问题研究 [D]. 株洲 : 湖南工业大学 ,2012.

[2] 易威环 .A* 算法在 coco2d-x 游戏开发中的研究和应用 [J]. 信息与电脑 ,2019(17):36-40.

[3] 钟瑛 . 改进 A* 算法在游戏地图路径搜索中的应用研究 [J]. 网络安全技术与应用 ,2013(8):54-56.

[4] 周小镜 . 基于改进算法的游戏地图寻径的研究 [D]. 重庆 : 西南大学 ,2014.

[5] 陈素琼 . 基于改进 A* 算法的地图游戏寻径研究 [D]. 重庆 : 重庆师范大学 ,2016.

[6] PATRICK L. A* 路径损招算法入门 [EB/OL]. 孙璨 , 译 . (2010-07-19)[2024-01-23]. https://blog.csdn.net/cnfnjatmzx/article/details/5774360.

[7] 李恬 , 张树美 , 赵俊莉 . 即时战略游戏的智能流场寻路算法设计与实现 [J].JI 计算机应用 ,2020,40(2):602-607.

[8]KRAWCZYK B, MINKU L L, GAMA J, et al.Ensemble learning for data stream analysis: a survey[J].Information Fusion,2017,37:132-156.

[9] 李响 . 基于 Unity3D 的 ECS 框架的设计与实现 [D]. 上海 : 上海交通大学 ,2019.

[10]WIEBUSCH D, LATOSCHIK M E. Decoupling the entity-component-system pattern usingsemantic traits for reusable realtime interactive systems[C]// Software Engineering and Architectures for Realtime Interactive Systems.Piscataway:IEEE,2017:25-32.

【作者简介】

王霞（1983—），女，福建福州人，硕士，副教授，研究方向：虚拟现实技术应用。

（收稿日期：2024-03-22）