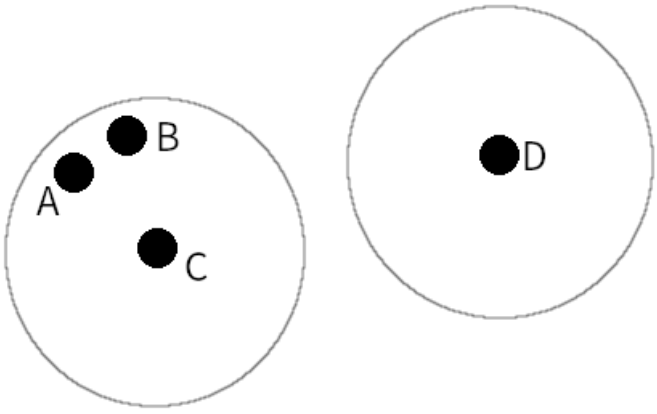
游戏服务器AOI的实现



在一个场景里，怪物A攻击了玩家B，玩家B掉了5血量。玩家B反击，怪物A掉了10血量。玩家C在旁边观看了这一过程，而在远处的玩家D对这一过程毫无所知。这是MMO游戏中很常见的一情景，从程序逻辑的角度来看，把它拆分成以下几部分

1. 怪物A感知玩家B在攻击距离内，释放了技能，并把整个过程广播给附近的玩家B、玩家C
2. 玩家B感知怪物A在攻击距离内，释放了技能进行反击，并把整个过程广播给自己(玩家B)、附近的玩家C
3. 玩家D因为离得太远，无法感知这个过程

可以看到，整个逻辑都是以位置为基础来进行的，玩家需要知道周边发生了什么。通常把玩家周边的这块区域叫做玩家感兴趣的区域，即AOI(Area Of Interest)，其大小即玩家的视野大小，上图画出了C、D这两个玩家的AOI。玩家AOI区域里的视觉变化（如攻击、掉血、移动、变身、换装等等），都需要通知玩家。而不在区域内的变化，比如上面的玩家D的AOI不包含A、B，就不需要通知他。怪物是不需要知道这些视觉变化的，因此一般来说怪物是没有AOI的。

AOI的核心是位置管理，其作用一是根据AOI优化数据发送（离得太远的玩家不需要发送数据，减少通信量），二是为位置相关的操作提供支持（例如玩家一个技能打出去，需要知道自己周围有哪些怪物、玩家，这些都是通过AOI来查询）。

PS: 场景中的玩家、怪物、NPC等统称为实体，下面有用到。

Interest列表

AOI的作用之一是优化数据发送，哪到底这个要怎么实现呢。以上面的情景为例，怪物A攻击时，是如何知道要把数据发送给B、C，而不发给D呢？最简单的办法是把场景里的玩家遍历一次，计算一下位置。但在实际中，一次攻击可能会下发4到5个数据包（攻击、掉血同步、怪物死亡、击退等等），现在有些游戏喜欢做成一刀打一片怪，那数据包可能要到10个以上了，每次都计算一下显然是不太现实的。因此一般每个实体上都有一个列表，所有对该实体感兴趣的玩家（即AOI包含该实体的玩家），都在列表上，一般把这个列表叫做Interest列表，或者观察者列表、目击者列表。例如，C在A、B的Interest列表里，D不在，所以A、B攻击时，把数据发给了玩家C，没发给D。

每当位置变化时，需要维护这个列表，这个处理起来还挺麻烦，后面再细说。

AOI区域的形状与大小

理想情况下，AOI区域是圆形的，因为现实生活中人在各个方向的视野大小都是一样的。不过用来玩游戏的手机、显示器可不是圆形的，因此为了方便，很多时候AOI是做成了方形的。一来AOI区域的大小并不需要很严格，大点小点一般没问题，二是判断点是否在圆内，需要计算平方，而判断是否在正方形内，只需

1. 更改debian的软件源sources.list(32167)
2. 解决Qt5.7.0 cannot find -lGL(24997)
3. sem_timedwait的用法(24329)
4. CMake优先链接静态库(22064)
5. 多线程中的信号处理(16723)

评论排行榜

1. ubuntu14.04折腾迅雷xware(6)
2. 解决Qt5.7.0 cannot find -lGL(4)
3. 测试clang-format的格式化效果(3)
4. 网络问题导致vscode无法运行测试用例(2)
5. Protocol Buffers与FlatBuffers效率对比(2)

推荐排行榜

1. ubuntu14.04折腾迅雷xware(6)
2. 多线程中的信号处理(4)
3. CMake优先链接静态库(3)
4. 解决Qt5.7.0 cannot find -lGL(3)
5. Qt Creator 修复The code model could not parse an include file(2)

最新评论

1. Re:测试clang-format的格式化效果

我补充下ClangFormat详细文档，方便后来者阅读。

--骇客技术

2. Re:C++使用mutable在const成员函数中加锁

原来如此

--witcherk

3. Re:测试clang-format的格式化效果

很需要这种clang format能力展示的博文，评论区提到clang-tidy，看来也可以用来检查命名风格。

--lingr7

4. Re:删除ubuntu旧内核

兄弟，你这个正则表达式连gcc等工具链都卸载了....

--蓝天上的云™

5. Re:测试clang-format的格式化效果

测试代码很棒啊。
最后提到的命名风格问题，可以

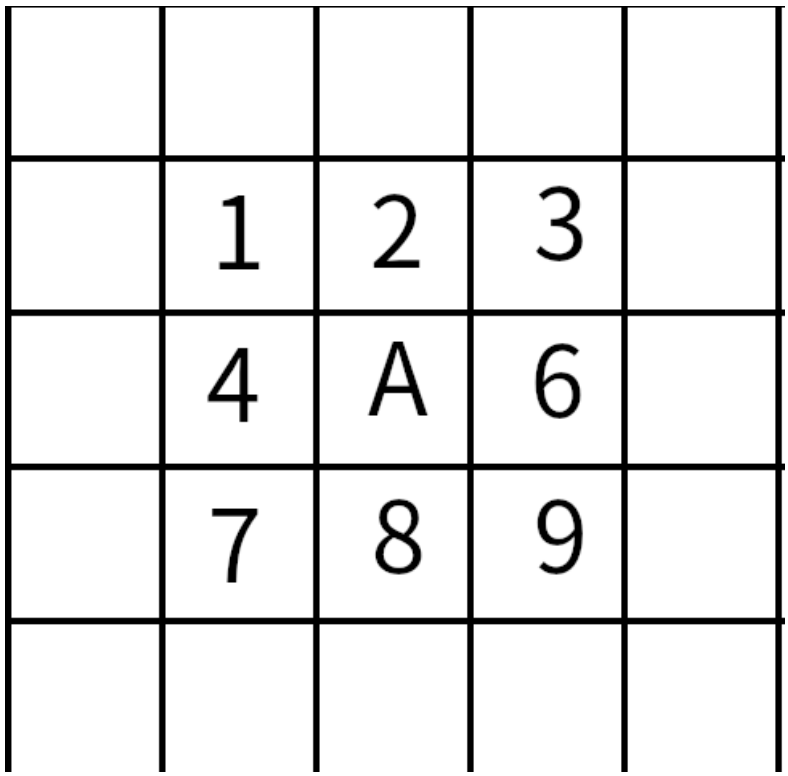
要判断大小，效率高一些。还有另一个原因就是有些AOI算法，不太好实现圆形区域（如下面的格子算法）。

虽然实体看得比较远，例如玩家可以看到很远的那座山。但很多游戏不会给你拉那么远的镜头的机会（看到的远处的山实际是装饰用的，走不到那个位置，和AOI无关），所以不少游戏的AOI都很小，只有几个格子，等同手机屏幕大小即可。折算到现实现生活中大概只有10多米，即只能看到旁边的那块石头。

AOI算法

AOI并没有什么特别优秀又通用的算法，甚至做一些同场景人数不多的游戏时（比如经典的传奇类游戏），简单的遍历或者全场景广播都比其他算法优秀。其他算法是各有各的特点，下面简单说下一些通用的AOI算法

- 九宫格



如图所示，九宫格AOI算法核心是把整个地图划分成大小相等的正方形格子，每个格子用一个数组存储在格子里的玩家，玩家的视野即上图中标了数字的九个格子（如果视野大小为2个格子，再往外扩一圈即可，依此类推）。九宫格的优点是效率高，拿到坐标后即可跳转到对应的格子，视野范围内需要遍历的格子也不多，配合经典的格子地图（tile map）再合适不过，都不需要把像素坐标转格子坐标。其缺点是占用内存有点大，因为必须为所有格子预留一个数组，即使是一个数组指针，长宽为1024的一个地图也要 `1024 * 1024 * 8 = 8M` 内存，这还不算真正要存数据的结构，仅仅是必须预留的。

我实现了一个格子的AOI算法用于测试：
https://github.com/changnet/MServer/blob/master/engine/src/scene/grid_aoi.hpp。

- 灯塔
  灯塔AOI是把整个地图划分成比较大的格子，每个格子称为一个灯塔，玩家视野一般涉及上下左右4个灯塔（之所以不是周边的9个而是4个是因为灯塔必须大于玩家的视野，因此偏向左下方就查左下方那4个格子即可，不用查9个，其他依此类推）。我觉得这个算法和九宫格没啥区别，无非就是格子变大了些，九宫格变成了四宫格，因此我没有实现这个算法。网易的pomelo有实现这个算法，可以参考一下。

- 十字链表
  把场景中的实体按位置从小到大用双向链表保存起来，X轴用一条双向链表，Y轴用一条双向链表，因为在画坐标时X轴和Y轴刚好呈十字，所以称十字链表（嗯，我觉得是这样，但找不到出处）。但是查资料的时候我发现，这个算法的实现几乎按55比例分成了两种

  1. 链表中保存的是一个点
     每个实体在链表中为一个节点，如 `a->b->c->d->f`

  2. 链表中保存的是一条线段
     每个实体在链表中为一个线段，包含(左视野边界AL、实体本身A、右视野边界AR)三个点，如下图

使用 clang-tidy，检查 readability-identifier-naming
--孤独行者

我不太理解第一种的算法，因为插入、移动实体时，都需要从其中一条链表当前实体分别向两边遍历到视野边界，才能维护interest列表，查找视野范围内的实体也是如此。既然是只遍历其中一条链表，为啥需要两条链表，只用X轴一条链表即可。有人认为需要两条链表是因为查找视野内实体是需要分别遍历x、y两轴，再求两轴的交集。我觉得遍历x轴，判断每个实体是否在视野内比求交集高效。

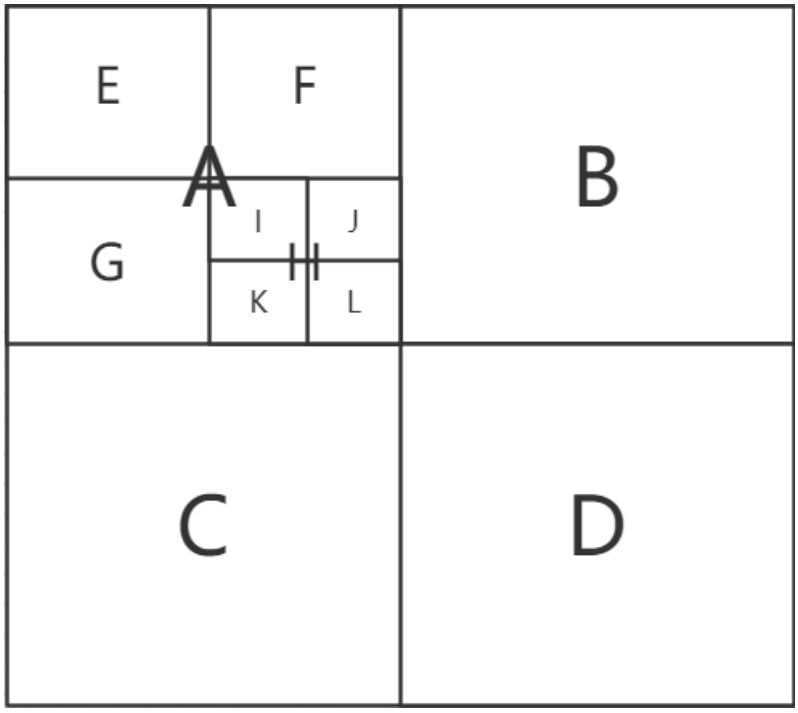
而对于第二种算法，每个实体为一条线段，线段起点为左视野边界，终点为右视野边界，中间还得加上实体本身，如上图中实体A为AL、A、AR，实体B为BL、B、BR。当插入、移动实体时，如果已方边界遇到对方实体，则表示对方进入或退出自己视野，如果对方边界遇到己方实体，则表示自己进入或退出对方视野。例如上图中，A在BL与BR之间，则表示A在B的视野范围内，而B不在AL与AR之间，则B不在A的视野范围内。当然，像怪物、NPC这种没有视野的实体，就可以优化成只需要一个点，按第一种算法处理。

算法二的实现比较复杂，其优点是移动的时候，遍历的数量比较少。例如：实体从(1, 100)移动到(1, 101)，必须找出视野范围内的玩家。对于算法一，没有什么变量能确定是遍历X轴还是遍历Y轴，因此只能随意选择一个。假如选择X轴，极端情况下，场景所有实体X坐标都在1，但Y轴都不一样，但这种算法就变成了遍历所有实体。对于算法二，由于X轴不变，因此X轴不需要移动，把Y轴向右移动1，在移动的过程中，根据“如果已方边界遇到对方实体，则表示对方进入或退出自己视野，如果对方边界遇到己方实体，则表示自己进入或退出对方视野”这个规则来处理遍历的实体即可。

但我的疑问是，算法二会导致链表长度大增加，其插入、移除的复杂度都高于算法一，仅仅是移动所带来的好处能抵消吗?
目前我用算法二实现了一个十字链表
https://github.com/changnet/MServer/blob/master/engine/src/scene/list_aoi.hpp。

另外，十字链表这算法都是很怕聚集的，例如大部分实体的X坐标都在2，另一个实体从1移到3就需要遍历大量的实体了。

- 四叉树
  AOI的核心是对空间进行管理，格子太耗内存，链表遍历太耗CPU，那四叉树是一个比较合适的方案。四叉树是把地图分成4块，每一块里再分4小块，根据场景中实体的数量不断地递归划分直到最小值（比如一个实体的视野范围）。盗用别人的图演示一下



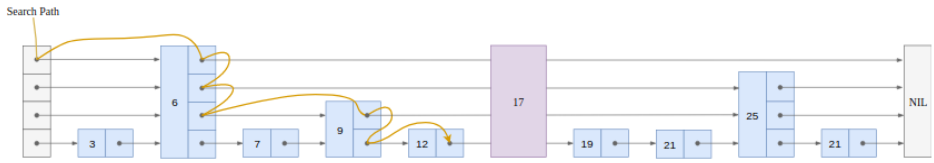
假如一个实体的坐标在L区域，那么需要从 `A->H->L` 这条路线来查询，遍历也不算太多。但是这个算法有一个缺点，就是视野不好处理，没法直接搜索相邻的实体。假如上图中的L区域右边为B区域，但是在四叉树查询B区域的实体是走 `B->?` 的，和L区域的完全不一样。

由于我对四叉树不太熟悉，也没在实际项目中用过，因此不太清楚一些具体的细节是怎么处理的，暂时没有实现。不过别人实现了一个，可以参考一下。

- 跳表
  我原本并没有考虑这个算法，但在对比九宫格和十字链表的性能后，我对自己实现的十字链表性能很不满意，但是九宫格效率虽高，却不适合大地图、可变视野、三轴坐标，说到底还是没有实现一种比较通用高效的算法，心有不甘。用callgrind看了十字链表半天后，CPU都耗在链表的遍历、插入、移动，因为它的链表实在太长了，而且有三条链表，最终没有找到什么办法来优化，放弃了。九宫格如果改用unordered map，性能会下降一些，加上三轴，需要遍历的格子多了，再降一些，实现可变视野后，继续再降一点，这么多缺点我连尝试的动力都没有了。而我到现在也没想明白四叉树是怎么搜索相邻的实体，如果非得从树根遍历，再加上三轴和可变视野，那我觉得性能不会太好看。

于是我打算实现单链表（类似十字链表的第一种算法，但没了y轴链表），对比一下是否会有更好的表现。不过单链表很明显的一个问题就是插入太慢，于是我打算加上索引。链表加上索引，那不就是跳表么。

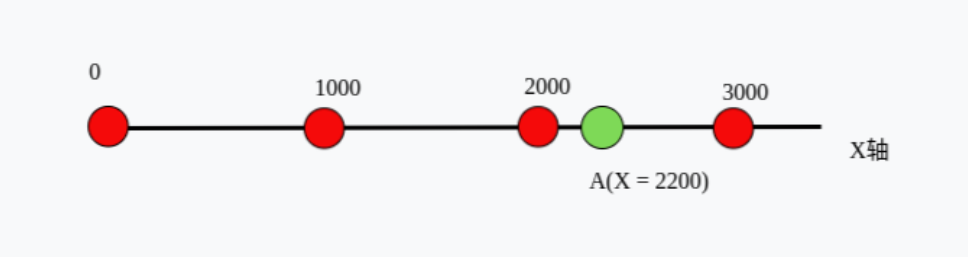
参考别人的blog



从上图中可以看到，跳表需要在链表中加上多层索引，然后根据索引跳跃式搜索。不过我觉得对于AOI来说，多层索引过于复杂，维护这些索引费时费力。那就用一层？用一层的话遍历索引也很费力，效率提升不大。链表节点变化时，还得更新索引，麻烦。

后来想用多链表来实现，即像九宫格那样，把x轴平均分段，每段是一个链表，用数组管理，访问时直接用 `x/index` 计算出数组下标。但是这样的话要查询相邻的实体可能要查询两条链表，而且实体移动需要跨链表时也需要额外的处理。

这里我忽然想到，我为啥不用静态索引跳表呢？对于一个通用的跳表而言，它存什么数据是未知的，数据的分布是未知的，它的索引理想的情况应该是平均分布的，这样查找的时候效率才高，因此需要维护索引。但对于AOI而言，它存的就是坐标，而且创建AOI的时候，肯定是知道地图的大小的。把x轴平均分段，每一段起点插入一个特殊的节点当作索引，然后用数组管理索引，访问时直接用 `x/index` 计算出数组下标。



红色为固定的索引节点（索引分段为1000），在创建AOI时就建立好，然后存到一个数组里。插入实体A(X=2200)时， `2200/1000=2` ，所以直接取索引节点2（索引从0开始）开始搜索合适的位置。

和原生的跳表相比，这种实现简单而且搜索效率高，不用维护索引。缺点是当实体聚集（比如所有实体坐标都在[0,1000]）时索引命中非常低。

可变视野与飞行、跳跃

绝大多数MMO游戏，尤其是武侠类的游戏，基本上都所有实体的视野都一样的。不过随着一些跳跃、飞行玩法的加入，飞行中或者跳到高处的玩家，视野更大。九宫格、灯塔之类的算法其实不太适合做这个。例如九宫格原本只需要遍历九个格子，假如有了可变视野，那只能按最大视野范围遍历，那就不止九个了，而绝大部分玩家的视野都是9个格子，徒增一些无效的遍历。

而用链表实现的AOI，视野变化只是遍历链表长度不一样，对现在的逻辑没有任何影响，都不需要改任何代码。

三轴AOI

越来越多的游戏开始使用3D地形，不过一般来说，地形对于武侠类游戏的服务器几乎没有影响，依然可以使用二轴AOI。一般是忽略高度，在高处的玩家和低处的玩家对于服务端来说是一样的，如果技能释放的时候有要求，那特殊处理一下也行。比如TrinityCore使用的是三坐标，但对于AOI来说只有二坐标。

当然想要做得细致一点也是可以的。九宫格需要多出一条轴，就变成27宫格了，而一张长宽高均为1024个格子的地图预留的内存就变成 `1024 * 1024 * 1024 * 8 = 8G` 。当然没人会给一张地图分这么多内存，

可以考虑用unordered map，只是会慢一点而已。而十字链表，也需要多加一条链表。我上面实现的十字链表就是三轴可变视野的，而九宫格实现三轴的，我还没见过。

AOI的实现方式

有些项目做AOI时，是在AOI里定时去更新同步位置的。即更新位置时，不通知前端，而是在定时器里定更新位置，同步到前端。这种方式可能会更省一些资源，但极限情况下就需要特殊处理。例如释放技能时，把远处的玩家勾过来，再一脚踢飞出去，如果用定时器，那这个位置变化过程可能就没有同步到前端。当然特殊的问题可以特殊处理，这个可以手动同步一次，或者在技能那边处理即可。

有些甚至以一个独立的进程去实现的。即实体有变化时，通知另一个进程，由该进程定时同步位置到前端，云风讨论AOI模块时便是这个思路。从位置同步这一块来讲，这是没问题的。但是一般来说AOI兼顾技能的位置查询，以及一些外显数据的同步，不知道他们是怎么处理的。

另一种方式是AOI做实时，更新玩家位置时，立刻更新AOI中的位置，并同步到前端。而像移动这种，不是在AOI中做的，而是由定时器根据玩家移动速度定时计算出新位置，同步到AOI中。

我更趋向于第二种的，因为可以控制得更加细致，所以AOI是写成一个库。而采用第一种方式的，往往是把AOI直接写成一个独立的进程（或微服务之类的）。当然有了一个库，把它封装成一个微服务的也不算太难。

性能

别人的实现，因为接口、语言都不一样，因此我是没法测试的，不过我自己写的，可以对比一下
CPU: AMD A8-4500M APU@1.9GHz
OS: debian 10@VirtualBox

```
[T0LP01-24 13:49:21]Using filter: aoi
[T0LP01-24 13:49:21]test grid aoi
[T0LP01-24 13:50:51][  OK] base test (89210ms)
[T0LP01-24 13:50:55][  OK] perf test 2000 entity and 50000 times random
move/exit/enter (3902ms)
[T0LP01-24 13:51:01]actually run 1767
[T0LP01-24 13:51:01][  OK] query visual test 2000 entity and 1000 times visual
range (5980ms)
[T0LP01-24 13:51:01]list aoi test
[T0LP01-24 13:51:01][  OK] list_aoi_bug
[T0LP01-24 13:51:23][  OK] base list aoi test (21999ms)
[T0LP01-24 13:51:29][  OK] perf test no_y(more index) 2000 entity and 50000
times random M/E/E (6174ms)
[T0LP01-24 13:51:41][  OK] perf test 1 index 2000 entity and 50000 times random
move/exit/enter (11683ms)
[T0LP01-24 13:51:46][  OK] perf test 2000 entity and 50000 times random
move/exit/enter (5153ms)
[T0LP01-24 13:52:11]actually run 1978
[T0LP01-24 13:52:11][  OK] query visual test 2000 entity and 1000 times visual
range (24737ms)
[T0LP01-24 13:53:42]actually run 674000
[T0LP01-24 13:53:42][  OK] change visual test 2000 entity and 1000 times visual
range (90775ms)

[T0LP01-24 13:51:01]list aoi test
[T0LP01-24 15:32:15][  OK] list_aoi_bug (2ms)
[T0LP01-24 15:33:07][  OK] base list aoi test (52175ms)
[T0LP01-24 15:33:19][  OK] perf test no_y(more index) 2000 entity and 50000
times random M/E/E (11598ms)
[T0LP01-24 15:33:33][  OK] perf test 1 index 2000 entity and 50000 times random
move/exit/enter (14237ms)
[T0LP01-24 15:33:48][  OK] perf test 2000 entity and 50000 times random
move/exit/enter (14483ms)
[T0LP01-24 15:34:06]actually run 1952
[T0LP01-24 15:34:06][  OK] query visual test 2000 entity and 1000 times visual
range (17719ms)
[T0LP01-24 15:34:49]actually run 634000
[T0LP01-24 15:34:49][  OK] change visual test 2000 entity and 1000 times visual
range (43290ms)
```

- 九宫格
  地图X最大6400,Y最大12800，格子边长64，视野半宽3 * 64,视野半高4 * 64，即这里实现的不是九宫格，

而是视野宽高不对等的格子。2000玩家、怪物、NPC随机进入地图，然后随机执行50000次退出、进入、移动，耗时3902ms，最终场景里还剩下1767个实体，对每个实体执行1000次查询视野范围内的实体，耗时5980ms

- 跳表（固定索引）
  地图X最大6400,Y最大19200，Z最大12800，视野半径256。2000玩家、怪物、NPC随机进入地图，然后随机执行50000次退出、进入、移动，耗时6174ms，最终场景里还剩下1978个实体，对每个实体执行1000次查询视野范围内的实体，耗时13243ms

当只采用一个索引时，这个就退化成单链表，我测了下，随机执行50000次退出、进入、移动，耗时11683ms，如果测多次，还是略好于十字链表的。

- 十字链表
  地图X最大6400,Y最大19200，Z最大12800，视野半径256。2000玩家、怪物、NPC随机进入地图，然后随机执行50000次退出、进入、移动，耗时10656ms，最终场景里还剩下1967个实体，对每个实体执行1000次查询视野范围内的实体，耗时17719ms

可以看到十字链表的性能并不是很理想，虽然算下来*单个实体的单次操作*（移动、进入、退出、视野变化）都在1ms以下，但是相对于九宫格还是太慢了，只能说够用。用跳表实现的介于两者之间，即支持三轴，也支持可变视野，性能又不太差，算是一个比较通用的AOI。

另外，这些测试数据有些异常，例如跳表的可变视野耗时基本是高于十字链表的，但从逻辑来看它们应该是差不多的，估计哪里有bug，但又没找到证据。

其他方案

- 云风的六边形

```
   __    __    __
  /  \__/  \__/  \
  \__/  \__/  \__/
  /  \__/  \__/  \
  \__/  \__/  \__/
```

云风用六边形做了一个灯塔AOI，相比四边形的灯塔只需要查询3个灯塔（灯塔设计得比视野大，虽然被7个灯塔包围，但是偏向哪边就查对应那边的3个灯塔即可）。不过我觉得多边形的运算太过于复杂（假如实体进入AOI时，需要判断属于哪个多边形，这个比灯塔、九宫格复杂。而且，这个要怎么实现三轴啊）。

- kbengine
  kbengine的AOI是三轴十字链表，支持可变视野。在查资料的时候，看过他的实现，这里记录一下。

`CoordinateSystems` 是AOI的主核心，三条链表都放这个类里。 `CoordinateNode` 是链表节点的基类，`EntityCoordinateNode` 是实体在链表中的节点， `RangeTriggerNode` 是视野左右边界在链表中的节点，通过 `COORDINATE_NODE_FLAG_POSITIVE_BOUNDARY` 和 `COORDINATE_NODE_FLAG_NEGATIVE_BOUNDARY` 这个flag来区分。

实体进入场景时，走 `Entity::installCoordinateNodes -- CoordinateSystems::insert` 把实体插入链表。接着初始化实体的视野会调用 `Witness::setViewRadius` ，这里会创建 `ViewTrigger` 并分别把左右视野边界插入链表。

当新节点插入或者位置有变化时，都会通过 `CoordinateSystem::update -- coordinateSystem::moveNodeX -- RangeTrigger::onNodePassX` 调整链表中的节点， `onNodePassX` 是一个多态函数，不同类型的CoordinateNode做不同的处理，触发实体进入、离开视野。

总体看下来，这个AOI运算量还是挺大的。这个模块没有单独出来，也没法直接放到我的代码里一同测试，性能如何不太清楚。

- 其他
  AOI的实现在英文资料非常少，想参考一下都不行。只搜索到一篇论文，测试了各种奇奇怪怪的AOI算法

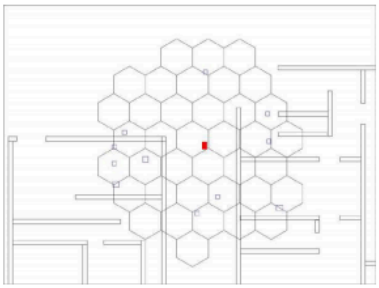


Figure 5: Hexagon tiles of area size 1.0 with interest radius of 2.0.

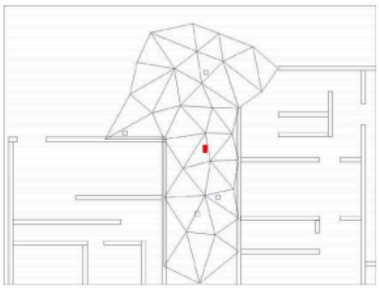


Figure 8: Tile visibility algorithm with interest radius of 2.0.

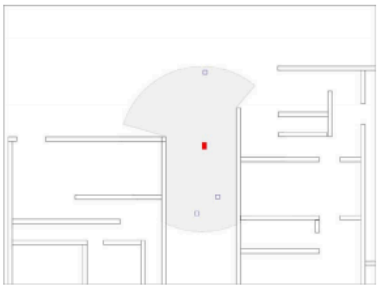


Figure 6: Ray visibility algorithm with interest radius of 2.0.

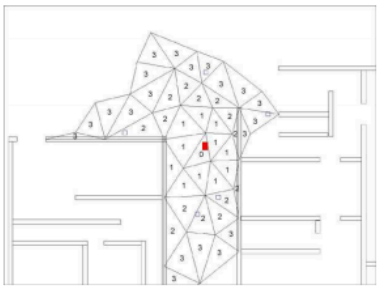


Figure 9: Tile neighbor algorithm with a depth of 3 neighbors. The numbers indicate the depth from the player's tile.

times and found that the largest variation in number of messages between experiments was of 0.4%.

4.3 Random Traces

Two random player movement traces were generated to compare with Orbius data. Both random traces were designed to send the same number of movements over the same period of time as the Orbius trace. Each of 28 clients sends one random move of length 0.5 in a randomly-chosen direction, at a constant rate of one move every 740 milliseconds; these parameters were derived from the average number of message sent by players divided by the experiment length. The main difference between the two random traces is the starting position of players. In the first trace players were all initialized in the space outside of buildings, similar to the way Orbius players were initialized; in the second trace 4 players were initialized outside and 24 inside of buildings—14 were initialized within the same building.

There are two main reasons for considering these variations in random traces. From our observations of random movements players initialized outside buildings are unlikely to end up inside of buildings, and symmetrically players initialized inside a building are unlikely to exit from that building. This has the potential to make a significant difference in interest management performance—players inside a building have many opportunities for occluded sight, and thus may be able to take better advantage of visibility-based algorithms. Relative distribution is another important property with respect to region-based IM; the second trace thus also allows us to examine a scenario in which half of the players are located in one place rather than being more uniformly distributed.

4.4 Measurements

To evaluate our implementations we considered two main types of messages between the server and clients: *content messages* and *update messages*. Content messages are used when an object is discovered by a player that has no copy

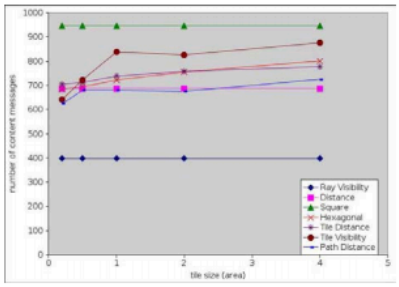


Figure 11: Average number of content messages received by a client for varying tile areas.

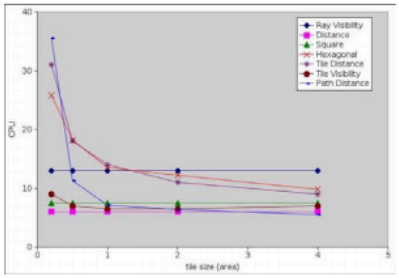


Figure 12: CPU consumption of the server for different tile areas.

但是这看起来并没有什么实际应用价值。唯一看到过真实应用的是TrinityCore，这个只是用了一个九宫格的AOI。

分类: coding

coding my life
粉丝 – 34 关注 – 0
+加关注

« 上一篇： CMake优先链接静态库
» 下一篇： win下添加Notepad3右键菜单

posted on 2021-01-24 16:19 coding my life 阅读(5374) 评论(0) 编辑 收藏 举报