Collision checking and pathfinding

Keeping your game optimized

By

Fco Borja Lozano

Universidad Complutense de Madrid

September 22, 2019

Contents

Pseudo-Code

Code

1 Introduction

1.1 The importance of efficient collision checking

In a game, there are usually thousands, if not millions, of particles and objects at the same time. This makes finding a fast way of controlling their collisions and trajectories efficiently a big priority.

For example, the game franchise Battlefield differentiates itself from other games as being a war-simulator, featuring large online multiplayer battles. There are dozens of players shooting each other and destroying terrain in real-time.

Keeping track of all objects and characters on the map at the same time becomes a great problem. To check whether a bullet has hit something, we can't iterate over all the objects on the map checking if there has been a collision because that would be too time consuming.

And this is where spatial partitioning comes into play. It will allow us to decrease the number of checks necessary to greatly improve the speed of calculations needed.

In the next section we will visit the most used spatial partitioning methods in the industry, together with some methods to decide whether two objects are colliding. At the end you will see a possible implementation of a generic KD tree (click here).

1.2 The importance of efficient pathfinding algorithms

In the previous example, trajectories are either simple and predefined, as a bullet that will just move with an acceleration and speed till it hits something, or controlled by a player, as are the characters in the game. But what happens when we want an object to decide on its own which path to choose? There is a destination, and the character has to go there.

Take, for example, Ultimate Epic Battle Simulator, a game in which we set the factions and have them fight one another. Each character chooses a path, and there can be dozens of thousands at the same time. And that is why pathfinding algorithms come into play.

The majority of the pathfinding algorithms used in videogames are variations of Dijkstra's algorithm. We will explore some of them and explanations on how to implement them. At the end you will find a possible implementation of a Weighed A* algorithm (click here).

2 Collision Detection

2.1 How to detect a collision

Every object has a certain size and shape, but to detect perfect collisions between two very complex objects, we would have to go pixel by pixel to check whether they intersect or don't. A way of simplifying this is by creating a virtual object with simple shapes which contains the real one. Next, we say that the two real objects collide if the virtual ones collide. We only have to create a virtual object once and move it together with the real one. The Minimum Bounding Box is the simplest way of doing this.

2.1.1 Minimum Bounding Box

There are two types of Minimum Bounding Box (MBOX): the AABB and the AOBB. The AABB or Axis-Aligned Minimum Bounding Box is the smallest hyperrectangle whose sides are parallel to the axis. This is simple to do, but imprecise. The AOBB or Arbitrarily Oriented Minimum Bounding Box, which is the smallest hyperrectangle without constraints of alignment, is more precise. There exist also minimum bounding hyperspheres, which are faster to check (we only have to measure the metric distance between centres), but they are usually less precise than AOBB's.

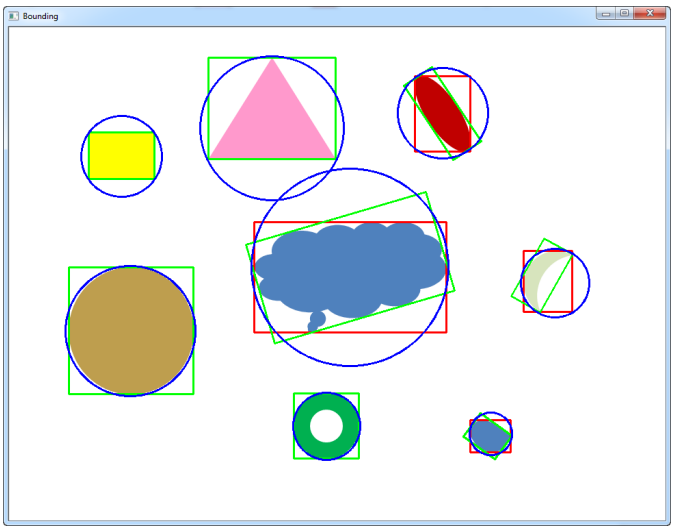

Figure 1: The blue circles represent the minimum bounding hypersphere. The red rectangle the AABB and the green one, the AOBB.

2.2 How to check every collision - Spatial partitioning

The pseudo-algorithm for brute force collision detection is pretty simple:

Algorithm 1 CollisionDetection

```
1: procedure DETECTCOLLISIONWITH(object)
2:     collision ← false
3:     for o in objects do
4:         if o ≠ object then
5:             if o.intersects(object) then
6:                 collision ← true
7:                 break
8:     return collision
```

But this is very inefficient because you have to check for every object that exists. The most intuitive way to solve this is by only checking the collision with objects that are close; for this, we use Spatial Indexing.

2.2.1 Quad trees - 2D

We divide the space into tiles and only check the objects that happen to be in the same tile. In the case that it overlaps the edges of the tiles, we will also check the objects in the adjacent tiles. In a two dimensional space, we usually use the quad tree method.

First, we establish a maximum number of objects that can be in the same tile, and establish the initial tile as the entire space. When the number of objects in a tile surpasses it, we subdivide the tile into four tiles, which are usually the same size, and insert in each sub-tile the objects that occupy that subspace.

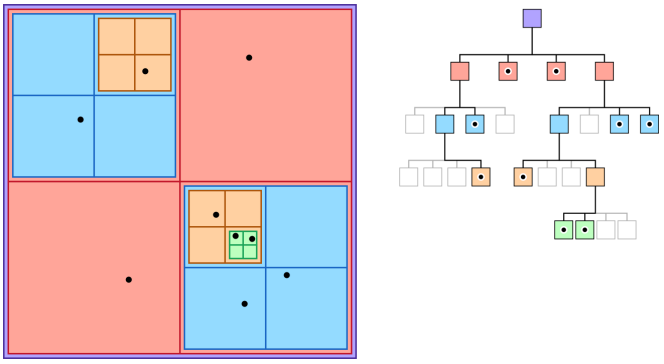

Figure 2: Example of a quad tree.

To do this we use a quad tree data structure where each node has:

- a way of knowing the start and end of the tile
- a list of the objects that are contained
- the four children

Usually, we will store the coordinates of the left upper point of the tile and the size of its sides. Let's see how the operations in this tree would look. Sometimes many objects can happen to be in the same point, that's why we usually set a minimum size for the tiles so that it doesn't divide unnecessarily. This is possible because we can set that minimum size to the minimum size of the objects that can exist, and thus all objects that are in that cell are colliding and don't have to be checked.

Algorithm 2 Subdivision

```
1: procedure SUBDIVIDE(node)
2:     children.lowerleft ← new Node
3:     children.lowerright ← new Node
4:     children.upperleft ← new Node
5:     children.upperright ← new Node
6:     for o in node.list do
7:         insert(node, o)
```

Algorithm 3 Insertion

```
1: procedure INSERT(node,object)
2:     if node.children ≠ null then
3:         if object.position.x ≤ node.coordinates.x + node.size then
4:             if object.position.y ≤ node, coordinates.y + node.size then
5:                 node.children.upperleft.insert(object)
6:             else
7:                 insert(node.children.lowerleft, object)
8:         else
9:             if object.position.y ≤ node.coordinates.y + node.size then
10:                insert(node.children.upperright, object)
11:            else
12:                insert(node.children.lowerright, object)
13:    else
14:        if node.list.size = M then
15:            subdivide(node)
16:            insert(node, object)
17:        else
18:            node.list.insert(obect)
```

2.2.2 Octrees - 3D

The octree method is the three-dimensional variation of the quad tree method. Instead of subdividing the 2D tile into four 2D tiles, we subdivide the 3D tile (cell) into eight cells. We can implement it with an array instead of with a tree, this is called a linear octree.

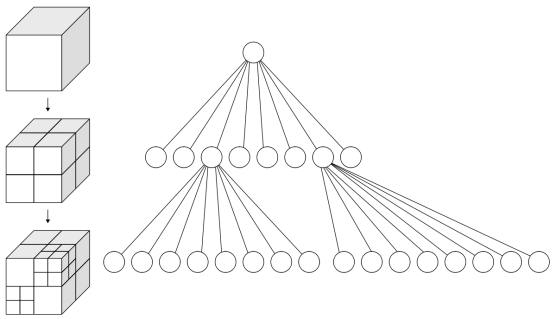

Figure 3: Example of an octree.

The problem with these methods is what to do when the objects overlap the borders of the cell. Sometimes it is solved by inserting it in the list of the parent cell so that it is not overlapping the border of the tile it is in, and sometimes by inserting it in every tile, it overlaps with. Another possible solution is the loose octree.

2.2.3 Loose octree

In a loose octree, we relax the bounds so that we can insert the object in one cell. This method is useful when we know the maximum size of the objects. What we do is create a relaxed boundary that is half the size of the object, so that if the centre of the object is inside the cell, the object is inside the relaxed bounds. That way the object only has to be inserted in one cell and we don't have to check whether it overlaps or doesn't. In the collision checks we will have to check with the adjacent cells.

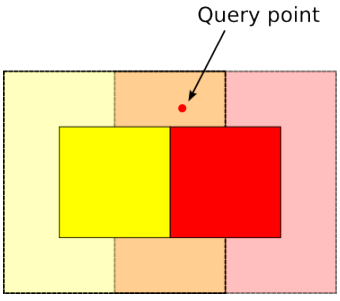

Figure 4: Example of loose tiles.

2.2.4 KD tree

A KD tree is the generic K dimensional equivalent of an octree. As there are many planes to split, the canonical KD tree does it as follows:

- Each time we have to split the plane we split it in one dimension.
- We split in the median of the positions of the objects in the tile. A fast way to do this is to pick some random objects and do their median.

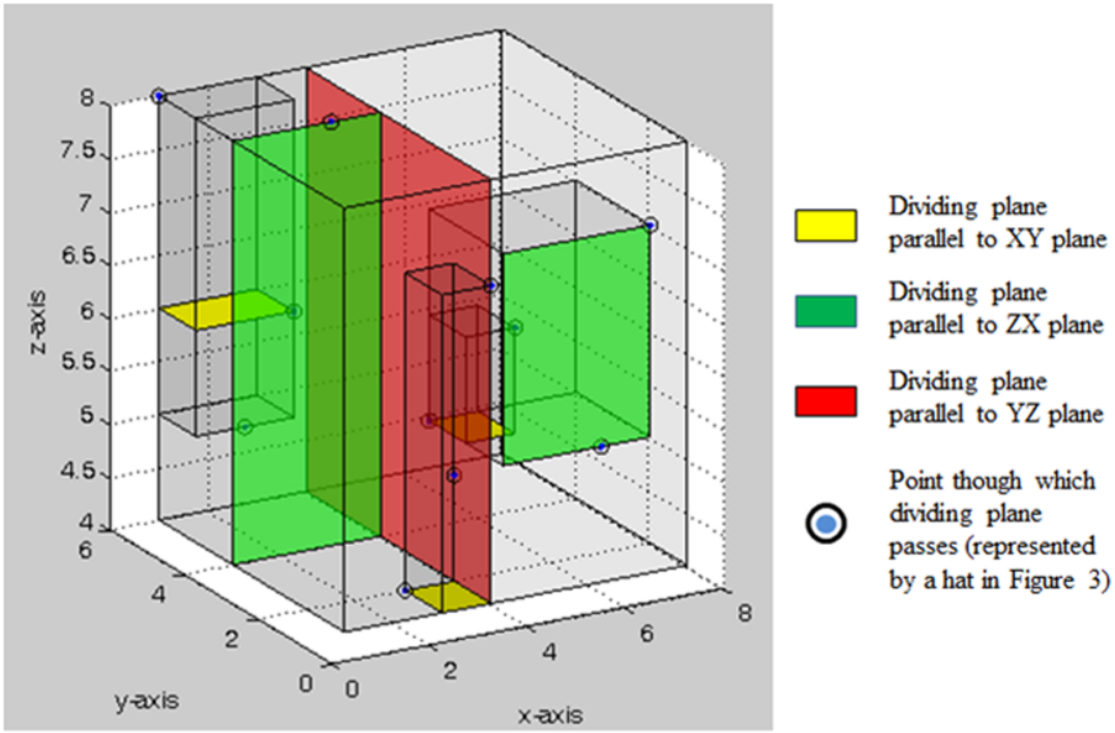

Figure 5: Example of a KD tree.

2.2.5 BSP tree

A BSP tree or Binary Space Partitioning tree is another way to subdivide the space. It is a generalization of the KD tree. The only difference is that the partition doesn't have to be parallel to an axis.

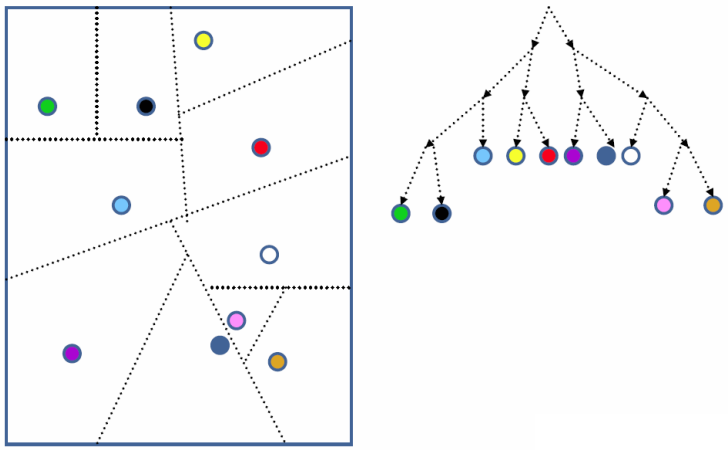

Figure 6: Example of a BSP tree.

2.2.6 Spatial hashing

This method of space partitioning is quite different from the ones before. We subdivide the space at the start and assign each cell a hash using a good hash function, forming a hash table. Then we store every object that exists within the cells in the hash table of cells we have created.

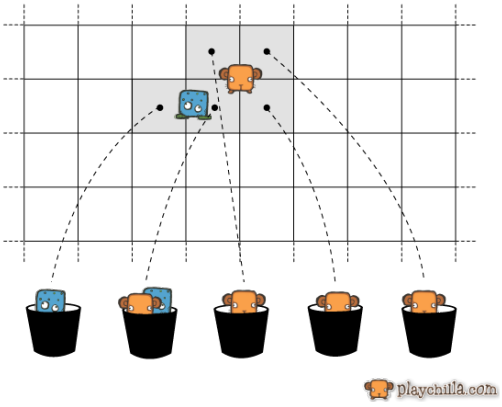

Figure 7: Example of hash partitioning.

The weak point of this method is that we have to set the size of the cells at the start and depending on that and the number and size of objects that it will contain, the memory cost might outweigh the fast collision checking.

2.3 Code

The following code is a generalization of a KD tree which splits along the middle instead of the median. The 'VirtualCoordinate' and 'VirtualDivider' have to be implemented once the dimension has been decided. The 'Object' class also requires to be implemented once we decide the collision method.

Code 1: Generic KD tree

```
#pragma once
#include <exception>
#include <utility>

struct VirtualCoordinates
{
    VirtualCoordinates();
    virtual VirtualCoordinates& operator-(VirtualCoordinates const& c) = 0;
    virtual VirtualCoordinates& operator+(VirtualCoordinates const& c) = 0;
    virtual void operator+=(VirtualCoordinates const& c) = 0;
    virtual void operator-=(VirtualCoordinates const& c) = 0;
    virtual bool operator<(VirtualCoordinates const& c) = 0;
};

class VirtualObject
{
public:
    virtual bool contains(VirtualCoordinates const& c) = 0;
    virtual bool collides(VirtualObject* o) = 0;
    virtual ~VirtualObject() = 0;
    virtual VirtualCoordinates const& get_position() = 0;
    virtual void set_position(VirtualCoordinates& coord) = 0;
};

class VirtualDivider
{
    /*
     * Returns the size of the partitions (which is the same), and their positions.
     * Each time the method is called, the current axis changes.
     */
    virtual VirtualCoordinates& operator()(VirtualCoordinates const& ini,
        ↪ VirtualCoordinates const& size, VirtualCoordinates& first,
        ↪ VirtualCoordinates& second) = 0;
};

template <class Coordinate, class Divider, typename std::enable_if<std::is_base_of<
    ↪ VirtualCoordinates, Coordinate>::value, Coordinate>::type,Coordinate,
    ↪ typename std::enable_if<std::is_base_of<VirtualDivider, Divider>::value,
    ↪ Divider>::type >
class KD tree
{
    class Hyperrectangle : public VirtualObject
    {
        Coordinate pos_, size_;
```

```
public:
    Hyperrectangle() : pos_(Coordinate()), size_(Coordinate())
    {
    }

    Hyperrectangle(Coordinate& pos, Coordinate& size) : pos_(pos), size_(size)
    {
    }

    Coordinate const& size() const { return size_; }
    Coordinate const& position() const { return pos_; }
};

class Node : Hyperrectangle
{
    static Coordinate min_size_;
    static Divider d_ = Divider();
    static int int max_n_obj;
    Node *l, *r;
    std::vector<VirtualObject*> list_;

public:
    Node()
    {
    }

    Node(Coordinate& pos, Coordinate& size)
    {
        this->pos_ = pos;
        this->size_ = size;
    }

    void insert(VirtualObject* o)
    {
        if (l == nullptr)
        {
            if (this->size() < min_size_)
                list_.push_back(o);
            else if (list_.size() == max_n_obj)
            {
                subdivide();
                insert(o);
            }
            list_.insert(o);
        }
        else
        {
            if (l->collides(o))
                l->insert(o);
            if (r->collides(o))
                r->insert(o);
        }
    }
```

```
        void remove(VirtualObject* o)
        {
            if (l == nullptr)
            {
                list_.erase([o](VirtualObject* aux)-> bool { return o == aux; });
            }
            else
            {
                if (l->collides(o))
                    l->remove(o);
                if (r->collides(o))
                    r->remove(o);
            }
        }

        void update(VirtualObject* o, Coordinate& c)
        {
            remove(o);
            o->set_position(c);
            insert(o);
        }

        void subdivide()
        {
            Coordinate pos1, pos2, size = d_(this->pos_, this->end_,pos1,pos2);
            l = new Node(pos1, size);
            r = new Node(pos1, size);
        }
    };

    Node* root_;

public:
    KD tree(Coordinate& size, Coordinate& min_size, int max_objects) : root_(new
        ↪ Node(Coordinate(), size))
    {
        Node::max_n_obj = max_objects;
        Node::min_size_ = min_size;
    }

    void insert(VirtualObject* o)
    {
        if (!root_->contains(o->get_position()))
            throw std::domain_error("The objects is outside the space assigned.");
        root_->insert(o);
    }

    void remove(VirtualObject* o)
    {
        if (!root_->contains(o->get_position()))
            throw std::domain_error("The objects is outside the space assigned.");
        root_->remove(o);
```

```
    }

    void update(VirtualObject* o, VirtualCoordinates& c)
    {
        if (!root_->contains(o->get_position()))
            throw std::domain_error("The objects is outside the space assigned.");
        root_->update(o, c);
    }
};
```

3 Path finding

The path finding problem is one of the most basic problems one has to face when doing a game. To be able to face this problem we need to create the graph equivalent of the terrain. This turns the problem into finding the shortest path in the graph. We must take into account that the more nodes we have per metric unit, the smoother the path will be, and the slowest the search will be.

There are two ways to solve the search:

- finding the shortest path.
- finding a short enough path.

3.1 Dijkstra's algorithm

This algorithms explores all reachable nodes from the closest to the furthest starting from the initial node. This continues till it has reached the destination or it has explored all reachable nodes. The original version of Dijkstra's algorithm uses the value of the edges to prioritize nodes, but in a game, the nodes store their coordinates and we use the module of the vector joining the nodes' coordinates.

Let's see how we implement this.

The nodes will have:

- the coordinates.
- the current parent node and shortest path to the origin.
- the adjacent nodes.
- the accumulative length of the best path to this node named as g. We will define it recursively as $g = parent.g + distance(this, parent)$.

Algorithm 4 Dijkstra's Path Finder

```
1:  procedure FINDPATH(origin, destination)
2:      current ← origin
3:      openSet ← origin
4:      closedSet ←
5:      while !openSetempty(). do
6:          current ← adjacent node with smallest g that isn't in the closedSet
7:          if current = end then
8:              break
9:          for nodes in current.adjacent do
10:             tentative_g ← current.g + distance(current, node)
11:             if node is in openSet then
12:                 if node.g > tentative_g then
13:                     node.parent ← current
14:                     node.g ← tentative_g
15:             else
16:                 node.parent ← current
17:                 node.g ← tentative_g
18:                 openSet.insert(node)
19:     return recreatePath(current)
```

3.2 A*

A* is a variation of Dijkstra's and currently is the most famous algorithm for finding the shortest path to an objective. The logic behind it is pretty simple. We start from the first node and move from it prioritizing using a heuristic (What is a heuristic?). This time, we will also need to store f. Where $f = g + heuristic(node, end)$.

Algorithm 5 A* Path Finder

```
1:  procedure FINDPATH(origin, destination)
2:      current ← origin
3:      openSet ← origin
4:      closedSet ←
5:      while !openSetempty(). do
6:          current ← adjacent node with smallest g that isn't in the closedSet
7:          if current = end then
8:              break
9:          for nodes in current.adjacent do
10:             tentative_g ← current.g + distance(current, node)
11:             if node is in openSet then
12:                 if node.g > tentative_g then
13:                     node.parent ← current
14:                     node.g ← tentative_g
15:                     node.f ← node.g + heuristic(node, end)
16:             else
17:                 node.parent ← current
18:                 node.g ← tentative_g
19:                 node.f ← node.g + heuristic(node, end)
20:                 openSet.insert(node)
21:     return recreatePath(current)
```

3.3 Weighted A*

Weighted A* is a faster way to find a path that may be sub-optimal. The only difference in the pseudocode is that $f = w1 * g + w2 * h$ where h is the heuristic and w1 and w2 are weights used to change the prioritization of the nodes. The bigger w1/w2 is, the closer to Dijkstra's it will be. The smaller it is, the more nodes closer to the end will be.

3.4 Anytime A*

The principle behind Anytime A* or Anytime Repair A* is simple. We have the algorithm find a fast suboptimal solution and then continually work on improving the solution until allocated time expires. As it will be run on a different thread, it will send the current best solution it has found. That way we have the benefits of both a fast suboptimal first and those of a longer optimal search.

3.5 Hierarchical path finding

This algorithm uses several layers at different abstraction levels to speed up the search. For example, if we can group up connected nodes inside pseudo-nodes, we can find the closest path between the pseudo-nodes before finding the shortest path inside the pseudo-nodes. This also provides the possibility of returning a part of the path before we have found the complete path.

3.6 Code

Code 2: Node

```
struct Coordinates
{
    float x, y;
    Coordinates() : x(0), y(0) {}
    Coordinates(float x, float y) : x(x),y(y) {}
};

class Node {
    Coordinates coord_;
    std::vector<Node*> adjacent_;
    bool is_obstacle_;
public:
    Node() : coord_(Coordinates()), is_obstacle_(false) {}
    Node(float const& x, float const& y, bool const& is_obstacle = false) : coord_(
        ↪ Coordinates(x,y)), adjacent_(std::vector<Node*>()), is_obstacle_(
        ↪ is_obstacle) {}
    std::vector<Node*> const& get_adjacent() const { return adjacent_; }
    void add_adjacent(Node* adj) { adjacent_.push_back(adj); }
    Coordinates const& get_coordinate() const { return coord_; }
    void set_coordinates(int const& x, int const& y) { coord_ = Coordinates(x,y); }
    bool is_obstacle() const { return is_obstacle_; }
    void set_obstacle(bool const& is_obstacle) { is_obstacle_ = is_obstacle; }
};
```

Code 3: Graph

```
class Graph {
    std::vector<Node*> nodes_;

public:
    Graph() {}
    void add_matrix(std::vector<Node*>& nodes, Matrix<bool>& m)
    {
        if(m.size() != nodes.size())
            throw std::domain_error("Matrix and nodes don't match.");

        for(int i = 0; i < m.size(); i++)
        {
            add_node(nodes[i]);
```

```
            for(int j = 0; j< m.size(); j++)
                if (m.at(i, j))
                    add_arch(nodes[i], nodes[j]);
        }
    }
    void add_node(Node* node)
    {
        nodes_.push_back(node);
    }
    void add_arch(Node* from, Node* to)
    {
        from->add_adjacent(to);
    }
    std::vector<Node *> Nodes() const& { return nodes_; }
};
```

Code 4: PathFinder

```
template<typename Distance = D, typename Heuristic = H>
class PathFinder
{
    struct PathNode
    {
        const Node* n;
        PathNode* par;
        float g;
        float f;
        PathNode(const Node* n) : n(n),par(nullptr),g(0),f(0) {}
    };

    Graph g_;
    Heuristic h_;
    Distance d_;
    int w1_, w2_;

    Node* find_closest(Coordinates const& c)
    {
        Node* node = nullptr;
        auto min = std::numeric_limits<float>::max();
        for (auto n : g_.Nodes()) {
            if (n->is_obstacle())
                continue;

            const float dist = d_(c,n->get_coordinate());
            if (dist < min)
            {
                node = n;
                min = dist;
            }
        }
        return node;
    }
    static std::vector<const Node*> reconstruct_path(PathNode* pn)
    {
```

```
        std::vector<const Node*> path;
        do{
            path.push_back(pn->n);
            pn = pn->par;
        } while (pn != nullptr);
        return path;
    }
public:
    PathFinder(Graph& g, int const& w1, int const& w2) : g_(g),h_(Heuristic()), d_(
        ↪ Distance()),w1_(w1), w2_(w2) {}
    std::vector<const Node*> find_path(Coordinates const& from, Coordinates const&
        ↪ to)
    {
        Node *orig = find_closest(from), *end = find_closest(to);
        std::unordered_map<Node*,PathNode*> openSet, closedSet;

        auto start = new PathNode(orig);
        openSet.insert({ orig, new PathNode(orig) });
        std::vector<const Node*> path;

        while (!openSet.empty())
        {
            auto min = std::numeric_limits<float>::max();
            std::pair<Node*,PathNode*> current;
            for (auto n : openSet)
                if (n.second->f < min)
                {
                    current = n;
                    min = n.second->f;
                }

            if (current.second->n == end) {
                path = reconstruct_path(current.second);
                break;
            }

            openSet.erase(current.first);
            closedSet.insert(current);

            for (auto n : current.first->get_adjacent())
            {
                if (n->is_obstacle() || closedSet.find(n) != closedSet.end())
                    continue;

                float tentative_g = current.second->g + d_(current.first->
                    ↪ get_coordinate(), n->get_coordinate());

                typename std::unordered_map<Node*, PathNode*>::const_iterator iter
                    ↪ = openSet.find(n);

                PathNode* pn;
                if (iter == openSet.end())
                {
```

```
                    pn = new PathNode(n);
                    openSet.insert({ n,pn });
                }
                else if((*iter).second->g >= tentative_g){
                    pn = (*iter).second;
                }else
                    continue;

                pn->g = tentative_g;
                pn->f = tentative_g*w1_ + h_(n->get_coordinate(), end->
                    ↪ get_coordinate())*w2_;
                pn->par = current.second;

            }
        }

        for(auto pn : closedSet)
        {
            delete pn.second;
        }
        return path;
    }
};
```

Here we have a test run of this code:

```
#include "PathFinder.h"
#include <random>
#include <iostream>

int main()
{
    auto nodes = new Node*[10];
    for (auto i = 0; i < 10; i++)
        nodes[i] = new Node[10];

    for (auto x = 0; x < 10; x++)
        for (auto y = 0; y < 10; y++)
            nodes[x][y].set_coordinates(x,y);

    Graph g;
    for (auto x = 0; x < 10; x++)
        for (auto y = 0; y < 10; y++) {
            g.add_node(&nodes[x][y]);
            if(x < 9)
            {
                g.add_arch(&nodes[x][y], &nodes[x + 1][y]);
            }
            if(y < 9)
            {
                g.add_arch(&nodes[x][y], &nodes[x][y+1]);
            }
            if(x < 9 && y < 9)
```

```
            {
                g.add_arch(&nodes[x][y], &nodes[x + 1][y + 1]);
            }
        }

    PathFinder<D,D> pf(g,10,1);
    std::vector<const Node*> path = pf.find_path(Coordinates(0, 0), Coordinates(10,
        ↪ 10));
    return 0;
}
```

References

[1] Collection of Game algorithms
https://www.reddit.com/r/gamedev/comments/1aw8am/collection_of_game_algorithms/

[2] Best algorithm for efficient collision detection between objects
https://stackoverflow.com/questions/7107231/best-algorithm-for-efficient-collision-detection-between-objects

[3] Loose Octrees
https://anteru.net/blog/2008/11/14/315/index.html

[4] Spatial Hashing
http://matthias-mueller-fischer.ch/publications/tetraederCollision.pdf

[5] Heuristics
http://theory.stanford.edu/~amitp/GameProgramming/Heuristics.html#S7

[6] A*
https://en.wikipedia.org/wiki/A*_search_algorithm

[7] When to use each type of Spatial Partitioning
https://stackoverflow.com/questions/99796/when-to-use-binary-space-partitioning-quadtree-octree

[8] KD Tree
https://en.wikipedia.org/wiki/K-d_tree

[9] Quadtrees vs Hashing Partitioning
http://zufallsgenerator.github.io/2014/01/26/visually-comparing-algorithms/

[10] Comparative Analysis of Spatial Partitioning Algorithms
https://otik.uk.zcu.cz/bitstream/11025/10652/1/Li.pdf