Navigation Mesh based Pathfinding for large crowds

Robert Lindner

Introduction

This paper is about methods that can be used to get as much performance as possible while simulating the pathfinding of large crowds. For my research I focused on implementing pathfinding using the A* (A-Star) algorithm and found a method that builds a library of paths while the game is running so that the amount of path calculations is brought down to a minimum, and at some point will completely stop.

To demonstrate the algorithm in a reasonable context I applied it in a small zombie game where the player controls a crowd of humans that need to stay alive as long as possible, while there is a horde of zombies chasing them. It can just as easily be used on any type of AI that requires pathfinding though, the benefits of which I will discuss in the conclusion of the paper.

Given this context, I realized that the method would also have to be compatible with the "Boid" (also known as "Birdoid" or Flocking algorithm).

To begin with I will explain my implementation of Navigation Meshes for A* and follow it up with my implementation of the Path Library.

Navigation Meshes

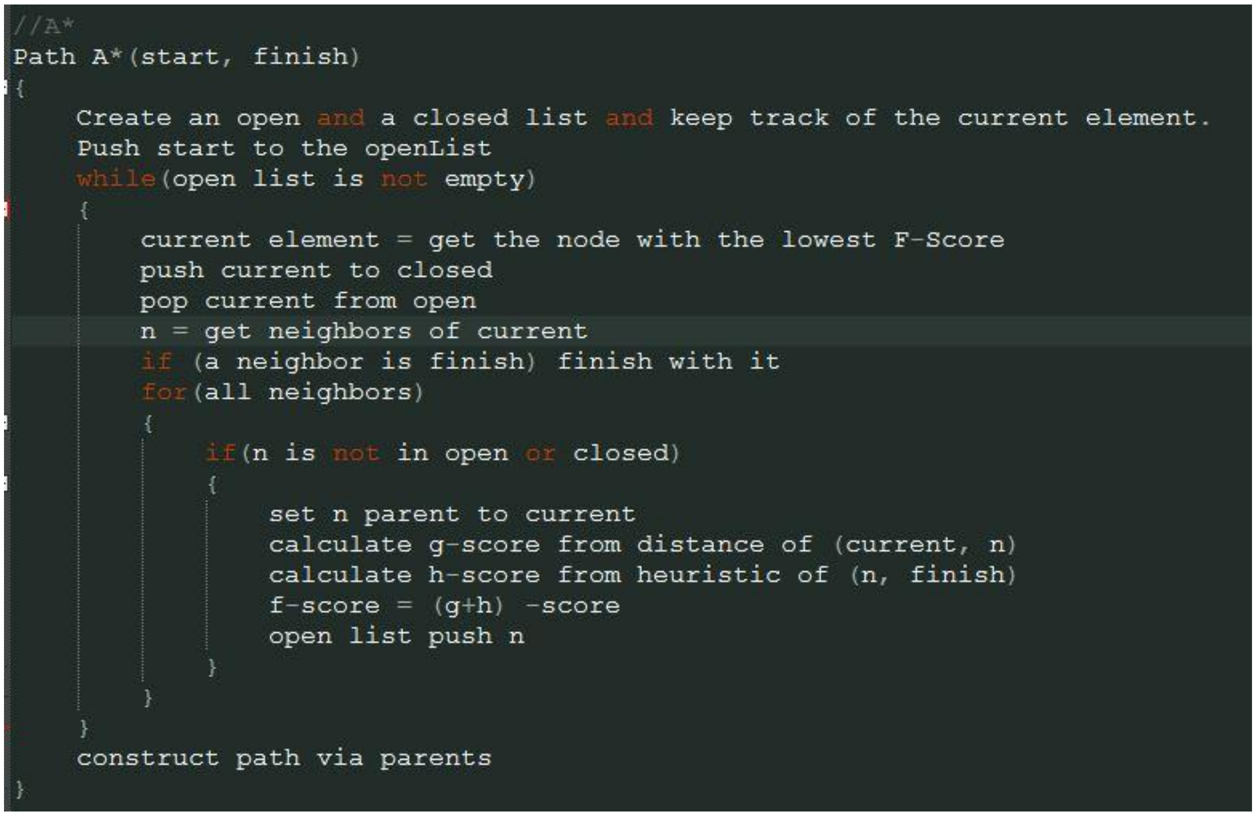
A-Star

In order to understand my Navigation Mesh implementation it is necessary to understand the way the A* algorithm works. Abstractly explained it calculates a path over a structure of nodes, similarly to the Djikstra algorithm, but instead of making sure it returns the shortest path, it estimates which nodes are more likely to lead to the target based on a Heuristic and stops as soon as it finds any solution.

This is done by adding the neighbors of any node that is calculated to a container which will hold the elements of potential paths. Every neighbor is assigned the current node as a parent, so that once the target node is found the path can be recreated by backtracking along the parents.

From the container of "open" nodes the one is selected which the heuristic estimates to be most likely to lead to the target, based on the sum of the distance between the current node and the neighbor node and the result of the heuristic between the neighbor and the target node. This "score" is added to the already existing score of the current node.

Every current node is marked as calculated by putting it in a container of "closed" nodes in order to prevent paths from looping, and the next node to be calculated is selected from the container of open nodes.



```
//A*
Path A*(start, finish)
{
    Create an open and a closed list and keep track of the current element.
    Push start to the openList
    while(open list is not empty)
    {
        current element = get the node with the lowest F-Score
        push current to closed
        pop current from open
        n = get neighbors of current
        if (a neighbor is finish) finish with it
        for(all neighbors)
        {
            if(n is not in open or closed)
            {
                set n parent to current
                calculate g-score from distance of (current, n)
                calculate h-score from heuristic of (n, finish)
                f-score = (g+h) -score
                open list push n
            }
        }
    }
    construct path via parents
}
```

For the Heuristic function I found that Euclidian worked best for me, which simply calculates the distance of the vector between the two nodes with a square root.

Navigation Element (Node)

From the algorithm above we can determine the requirements an Abstract NavElement needs are:

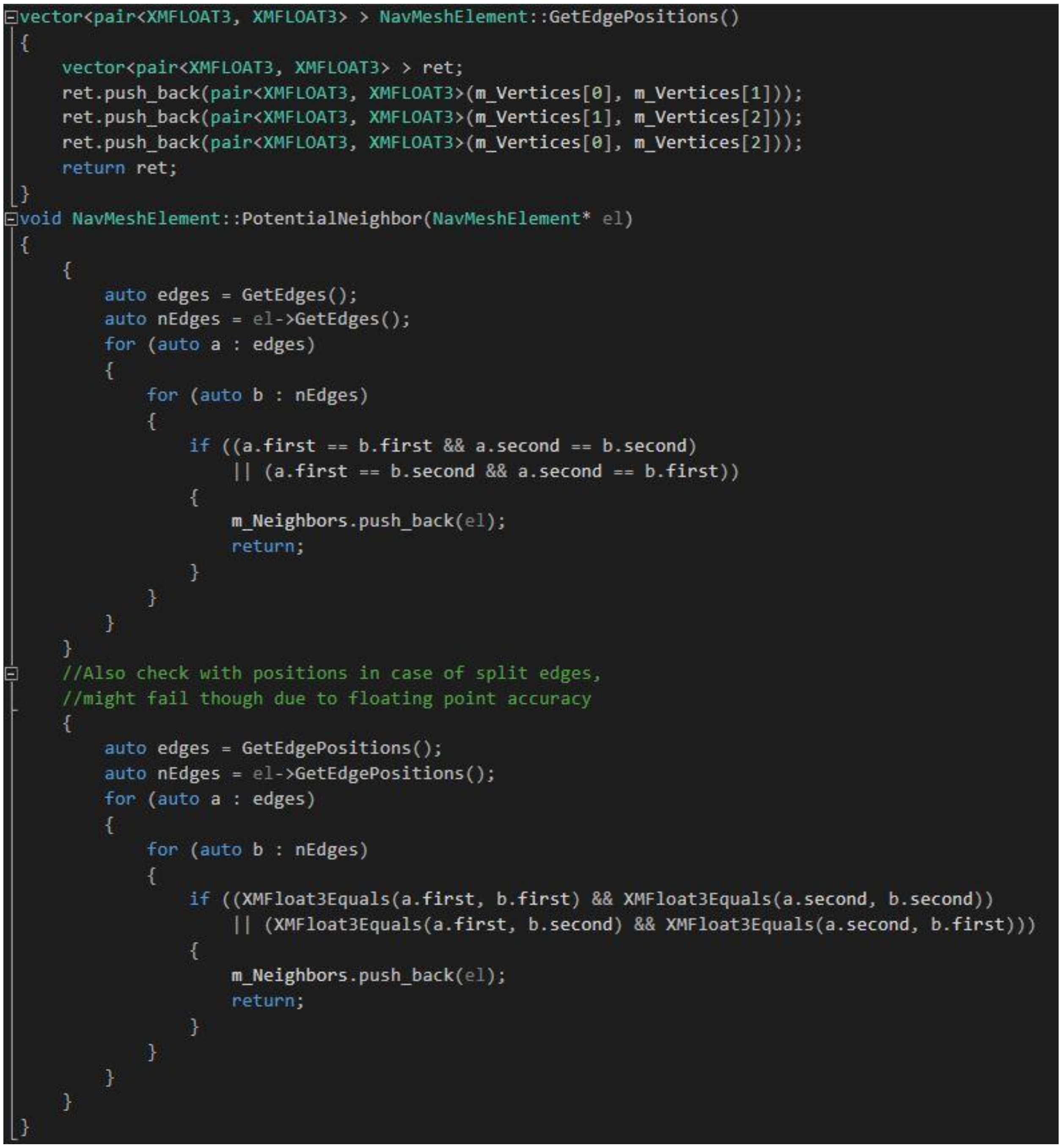
- Values that hold the G, H and F Score
- A Pointer to the parent NavElement
- A Position for the heuristic calculations
- A way to determine its neighbors. This could either be calculated at runtime or precalculated at the beginning, one having a heigher calculation cost and the other possibly bloating memory. Since I am trying to allow the calculation of as many actors as possible and am willing to sacrifice that for a less detailed navigation geometry, I opt for precalculation and store pointers to the neighbors in a structure

Additionally an element could hold information about having higher cost of walking on it or not being passable at all, but it is not necessary.

NavMesh

There are quite a few structures that can hold NavElements other than Navigation Meshes, like for instance a square grid or a hexagon grid, but they come at the disadvantage of usually being limited to 2D space (unless you are willing to sacrifice tremendous amounts of computing data for a 3d grid) and being less intuitive to construct for your scene, and surprisingly Navigation meshes are not a lot more complicated than any other structure on the A star side of things. The concept is simple: every triangle represents a NavElement, and its neighbors are determined by its edges.

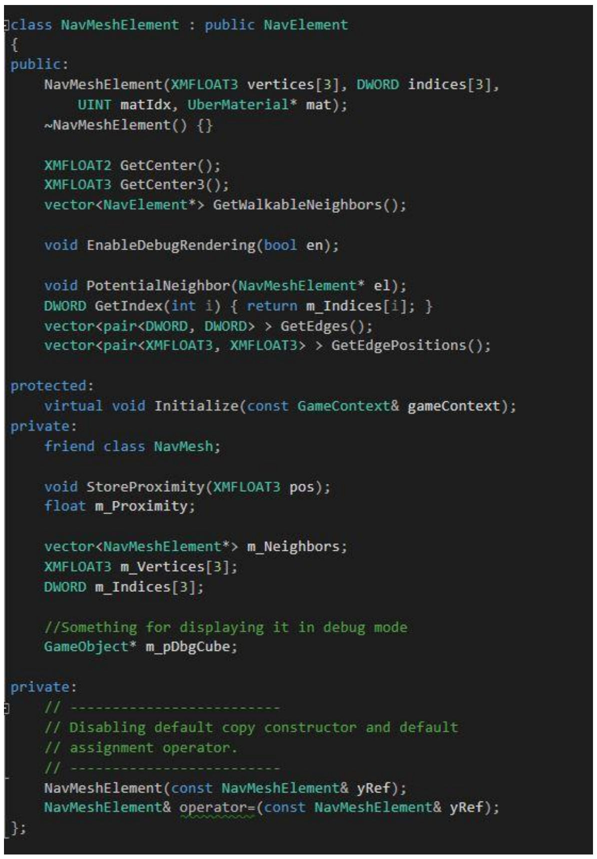
If two triangles have two matching indices each, they are neighbors, which is calculated in a nested for loop on initialization:



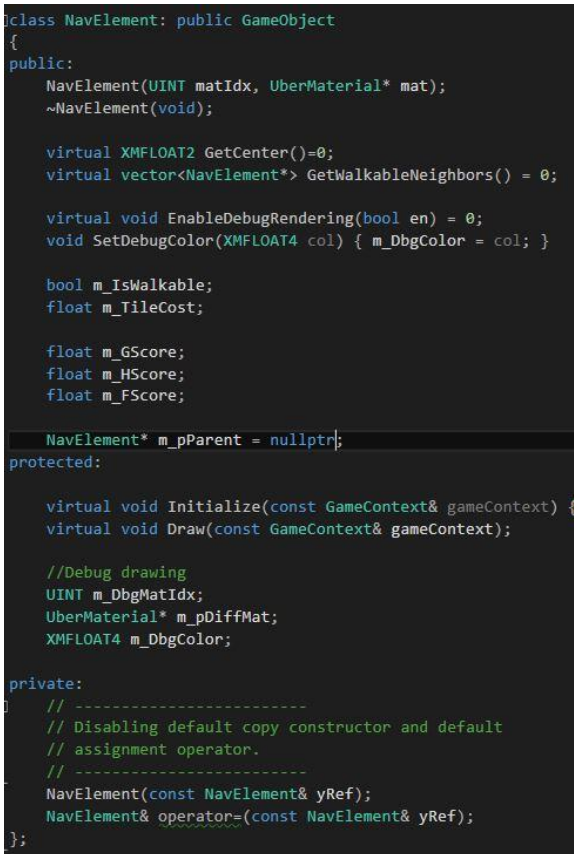
```cpp
vector<pair<XMFLOAT3, XMFLOAT3> > NavMeshElement::GetEdgePositions()
{
    vector<pair<XMFLOAT3, XMFLOAT3> > ret;
    ret.push_back(pair<XMFLOAT3, XMFLOAT3>(m_Vertices[0], m_Vertices[1]));
    ret.push_back(pair<XMFLOAT3, XMFLOAT3>(m_Vertices[1], m_Vertices[2]));
    ret.push_back(pair<XMFLOAT3, XMFLOAT3>(m_Vertices[0], m_Vertices[2]));
    return ret;
}
void NavMeshElement::PotentialNeighbor(NavMeshElement* el)
{
    {
        auto edges = GetEdges();
        auto nEdges = el->GetEdges();
        for (auto a : edges)
        {
            for (auto b : nEdges)
            {
                if ((a.first == b.first && a.second == b.second)
                    || (a.first == b.second && a.second == b.first))
                {
                    m_Neighbors.push_back(el);
                    return;
                }
            }
        }
    }
    //Also check with positions in case of split edges,
    //might fail though due to floating point accuracy
    {
        auto edges = GetEdgePositions();
        auto nEdges = el->GetEdgePositions();
        for (auto a : edges)
        {
            for (auto b : nEdges)
            {
                if ((XMFloat3Equals(a.first, b.first) && XMFloat3Equals(a.second, b.second))
                    || (XMFloat3Equals(a.first, b.second) && XMFloat3Equals(a.second, b.first)))
                {
                    m_Neighbors.push_back(el);
                    return;
                }
            }
        }
    }
}
```

Notice that I double check with the Vertex Positions too incase the imported mesh has unintentional split edges. Since this calculation is not done at runtime there is no downside except for slightly higher loading times for this. On that note, when creating NavMeshes, make sure the edges are not split.

For your convenience, I am including the c++ headers for the Navigation Element



```cpp
class NavMeshElement : public NavElement
{
public:
    NavMeshElement(XMFLOAT3 vertices[3], DWORD indices[3],
        UINT matIdx, UberMaterial* mat);
    ~NavMeshElement() {}

    XMFLOAT2 GetCenter();
    XMFLOAT3 GetCenter3();
    vector<NavElement*> GetWalkableNeighbors();

    void EnableDebugRendering(bool en);

    void PotentialNeighbor(NavMeshElement* el);
    DWORD GetIndex(int i) { return m_Indices[i]; }
    vector<pair<DWORD, DWORD> > GetEdges();
    vector<pair<XMFLOAT3, XMFLOAT3> > GetEdgePositions();

protected:
    virtual void Initialize(const GameContext& gameContext);
private:
    friend class NavMesh;

    void StoreProximity(XMFLOAT3 pos);
    float m_Proximity;

    vector<NavMeshElement*> m_Neighbors;
    XMFLOAT3 m_Vertices[3];
    DWORD m_Indices[3];

    //Something for displaying it in debug mode
    GameObject* m_pDbgCube;

private:
    // -------------------------
    // Disabling default copy constructor and default
    // assignment operator.
    // -------------------------
    NavMeshElement(const NavMeshElement& yRef);
    NavMeshElement& operator=(const NavMeshElement& yRef);
};
```



```cpp
class NavElement: public GameObject
{
public:
    NavElement(UINT matIdx, UberMaterial* mat);
    ~NavElement(void);

    virtual XMFLOAT2 GetCenter()=0;
    virtual vector<NavElement*> GetWalkableNeighbors() = 0;

    virtual void EnableDebugRendering(bool en) = 0;
    void SetDebugColor(XMFLOAT4 col) { m_DbgColor = col; }

    bool m_IsWalkable;
    float m_TileCost;

    float m_GScore;
    float m_HScore;
    float m_FScore;

    NavElement* m_pParent = nullptr;
protected:

    virtual void Initialize(const GameContext& gameContext) {
    virtual void Draw(const GameContext& gameContext);

    //Debug drawing
    UINT m_DbgMatIdx;
    UberMaterial* m_pDiffMat;
    XMFLOAT4 m_DbgColor;

private:
    // -------------------------
    // Disabling default copy constructor and default
    // assignment operator.
    // -------------------------
    NavElement(const NavElement& yRef);
    NavElement& operator=(const NavElement& yRef);
};
```

One complication NavMeshes come with that is finding out which element is responsible for a certain position in 3D space. On a grid it is easy to tell because you can simply convert the position into a set of indices.

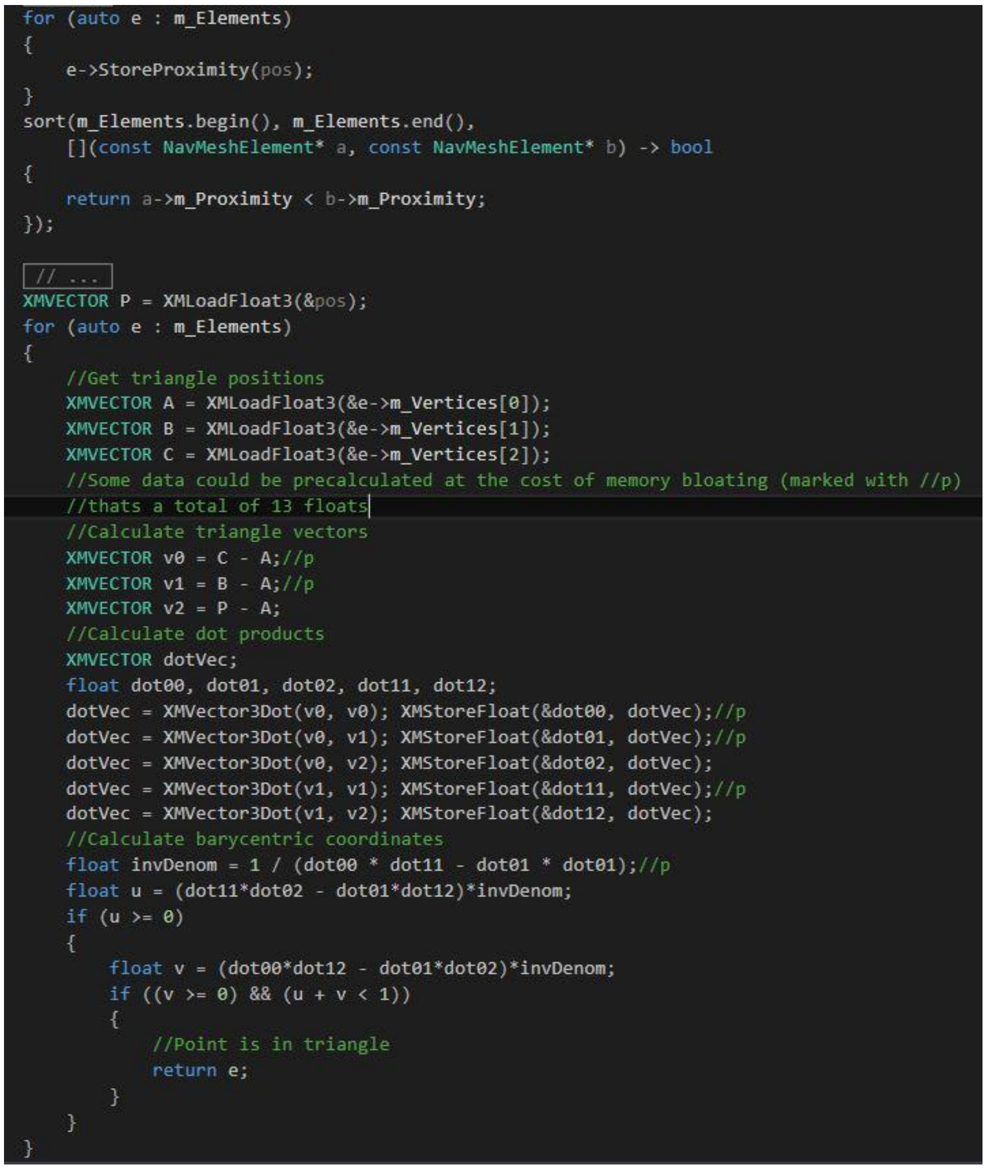
The method I came up with is the same used in raytracing applications for testing if a ray intersects with a triangle, which is calculating the barycentric coordinates. To be precise calculating those coordinates is the second part of the intersection test, namely a point in triangle test after the ray would be determined to intersect with the plane of the triangle. For this application, we only want the second part of that test since we are dealing with floating points in space.

In order to accelerate this, we first sort the list of elements by proximity to the point, so that we need to perform the calculations as little as possible. That comes with the benefit that incase none of the tests pass we can instead return the closest element to the point as a target, which is simply the first element of the already sorted list.

The coordinates are calculated using the dot products of the difference vectors of the triangle and an equation which is better explained than I would be able to in the link in the sources.

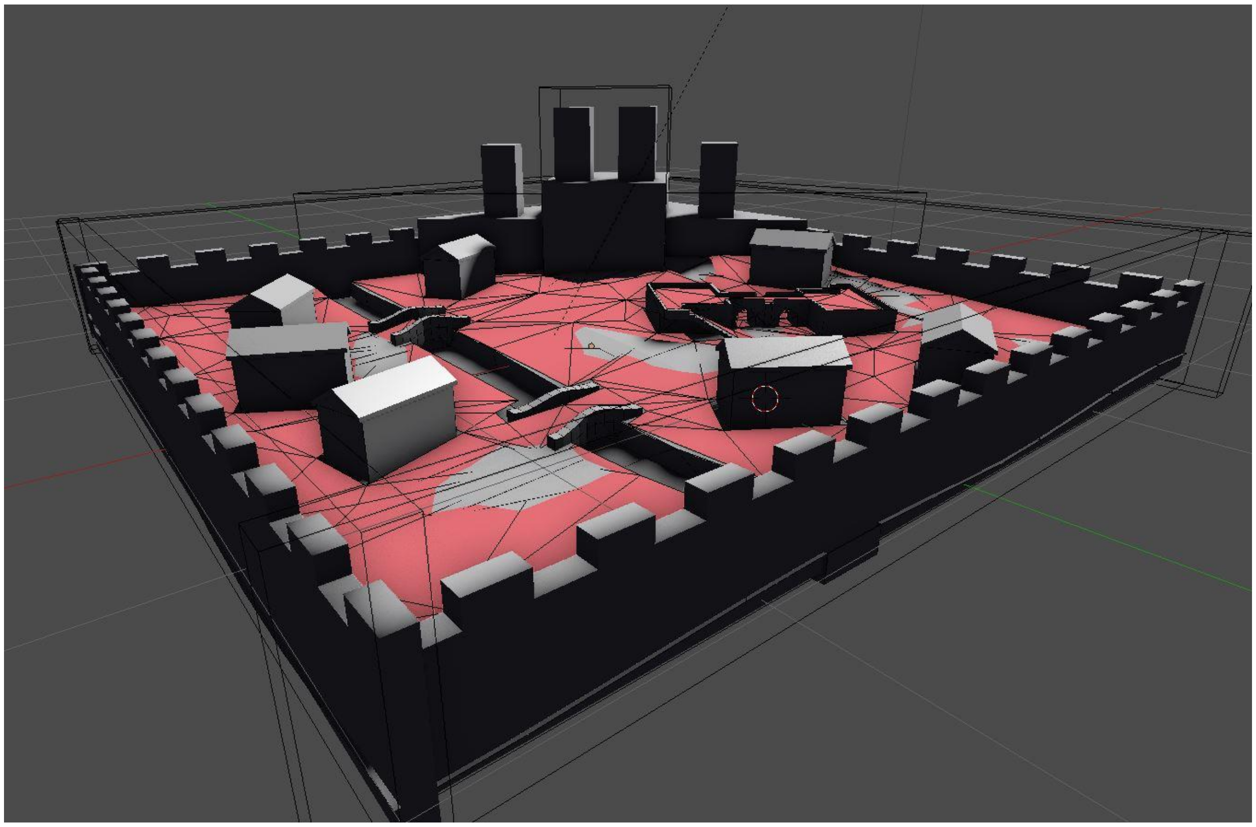
By convention the coordinates are labeled U and V since they can be used to calculate the actual Texture coordinates of the triangle.

If U and V are each larger than 0 and together smaller than 1, the point is in the triangle.



```
for (auto e : m_Elements)
{
    e->StoreProximity(pos);
}
sort(m_Elements.begin(), m_Elements.end(),
    [](const NavMeshElement* a, const NavMeshElement* b) -> bool
{
    return a->m_Proximity < b->m_Proximity;
});

// ...
XMVECTOR P = XMLoadFloat3(&pos);
for (auto e : m_Elements)
{
    //Get triangle positions
    XMVECTOR A = XMLoadFloat3(&e->m_Vertices[0]);
    XMVECTOR B = XMLoadFloat3(&e->m_Vertices[1]);
    XMVECTOR C = XMLoadFloat3(&e->m_Vertices[2]);
    //Some data could be precalculated at the cost of memory bloating (marked with //p)
    //thats a total of 13 floats
    //Calculate triangle vectors
    XMVECTOR v0 = C - A;//p
    XMVECTOR v1 = B - A;//p
    XMVECTOR v2 = P - A;
    //Calculate dot products
    XMVECTOR dotVec;
    float dot00, dot01, dot02, dot11, dot12;
    dotVec = XMVector3Dot(v0, v0); XMStoreFloat(&dot00, dotVec);//p
    dotVec = XMVector3Dot(v0, v1); XMStoreFloat(&dot01, dotVec);//p
    dotVec = XMVector3Dot(v0, v2); XMStoreFloat(&dot02, dotVec);
    dotVec = XMVector3Dot(v1, v1); XMStoreFloat(&dot11, dotVec);//p
    dotVec = XMVector3Dot(v1, v2); XMStoreFloat(&dot12, dotVec);
    //Calculate barycentric coordinates
    float invDenom = 1 / (dot00 * dot11 - dot01 * dot01);//p
    float u = (dot11*dot02 - dot01*dot12)*invDenom;
    if (u >= 0)
    {
        float v = (dot00*dot12 - dot01*dot02)*invDenom;
        if ((v >= 0) && (u + v < 1))
        {
            //Point is in triangle
            return e;
        }
    }
}
```

In order to create the NavMeshes I simply modified the existing scene Geometry in Blender and exported it to the Engine Mesh format via Collada. For anyone else working with Blender, make sure all edges are smooth to avoid splitfaces.

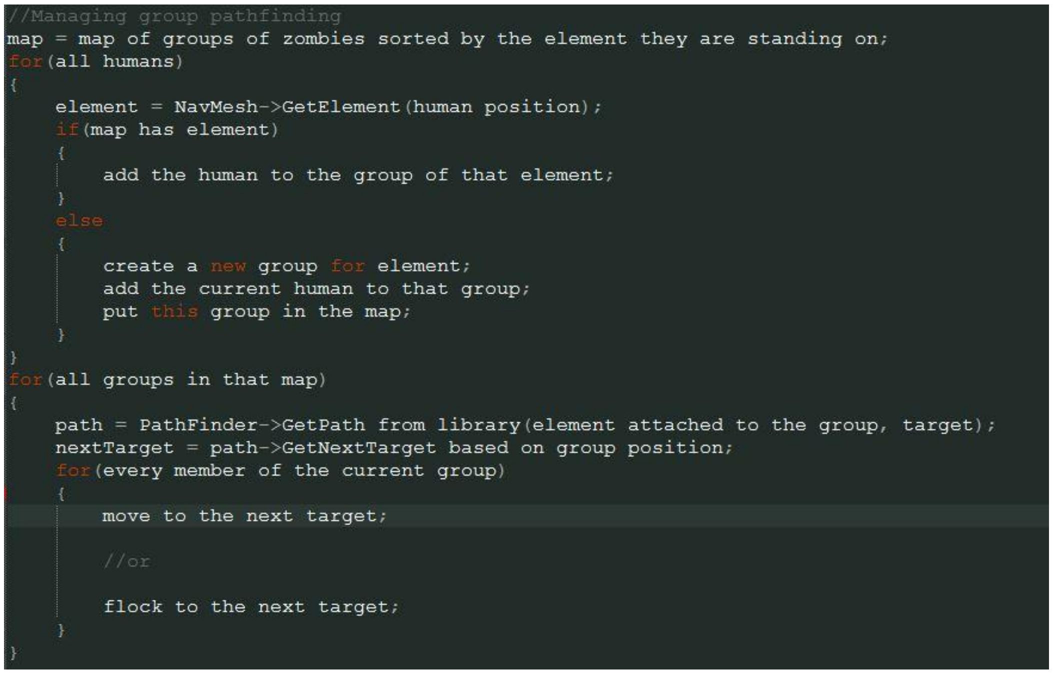


Path Library

This covers the NavMesh part of my research, and I will move on to explaining how I use this in a system that allows me to minimize the amount of path calculations.

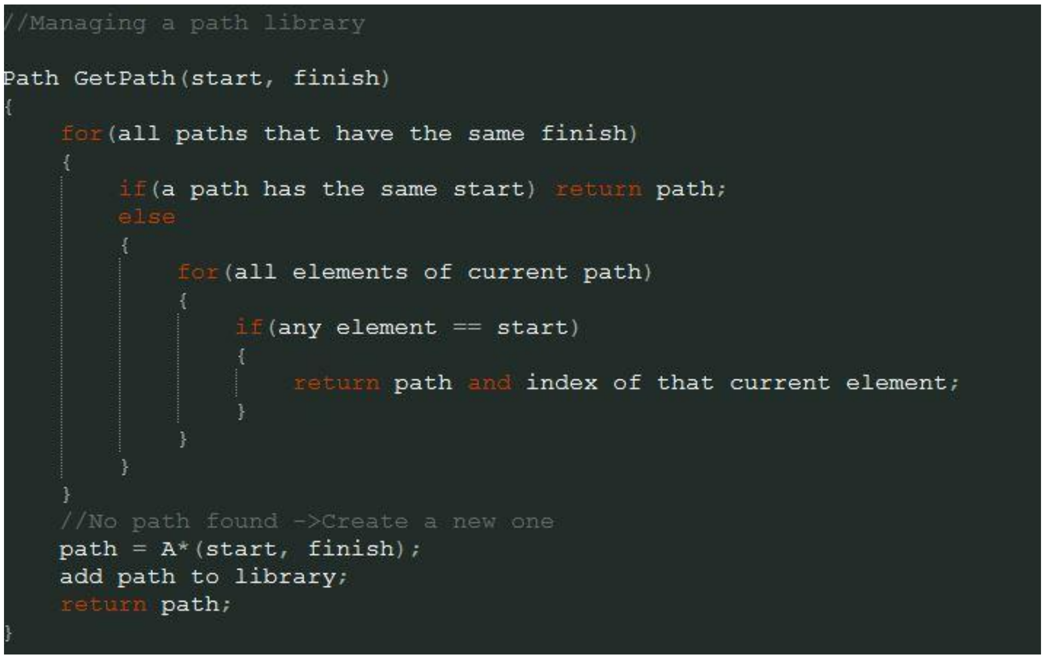
Abstractly speaking:

Instead of recalculating the path for every actor every frame, I sort the actors into groups based on what triangle they are standing on. That allows the amount of paths needed to be limited to the amount of inhabited nodes. The groups formed by this can then also be used for flocking.



```
//Managing group pathfinding
map = map of groups of zombies sorted by the element they are standing on;
for(all humans)
{
    element = NavMesh->GetElement(human position);
    if(map has element)
    {
        add the human to the group of that element;
    }
    else
    {
        create a new group for element;
        add the current human to that group;
        put this group in the map;
    }
}
for(all groups in that map)
{
    path = PathFinder->GetPath from library(element attached to the group, target);
    nextTarget = path->GetNextTarget based on group position;
    for(every member of the current group)
    {
        move to the next target;

        //or

        flock to the next target;
    }
}
```

Furthermore, instead of recalculating the paths every time, they are stored in a library that is indexed by their start and end node. If the library already has a path between those nodes, it does not need to recalculate the path. Even a path that is a subfragment of an existing path can be calculated that way. Theoretically if you let the game run long enough, every path will eventually be calculated and no more additional pathfinding is required



```
//Managing a path library

Path GetPath(start, finish)
{
    for(all paths that have the same finish)
    {
        if(a path has the same start) return path;
        else
        {
            for(all elements of current path)
            {
                if(any element == start)
                {
                    return path and index of that current element;
                }
            }
        }
    }
    //No path found ->Create a new one
    path = A*(start, finish);
    add path to library;
    return path;
}
```

On the downside, this will obviously increase memory load over time, so this solution is most suitable for levels with a low polycount on their navmesh. It did seem to help with the larger crowds though.

Conclusion

While the main goal of this project was to allow raising the maximum amount of actors that can be simulated per frame, this method could also be used for games where the AI does not require a very high accuracy, and the memory cost of storing the paths is not an issue.

Speaking about performance: Before I added Pathfinding to the game it ran at a little more than 60 fps, mainly due to the skinned animations on all the characters. Without skinned animations it was running at 300-600 before.

The naive pathfinding implementation brought the framerate down to 30 fps, and using the Path library gave me back approximately 5 – 10, so its running between 35 and 40 on my laptop most of the time. There are 100 actors in the game, and usually they are distributed onto around 5 elements. The navmesh has approximately 170 triangles.

That means currently every path calculation costs around 4-6 fps, but by splitting it onto elements and not actors, every actor needs around 1 frame every 5 seconds for the calclations. This is an acceptable cost to me, if the engine where faster at displaying animations it would allow a truly huge amount of actors to roam on the map.

There are always a few things that could be improved or changed though, and here are a few I can think of

- First, triangles could precalculate data for the point in triangle test, which I have commented in the code, again at the cost of memory. This could be optional in an engine.
- Second, it might be possible to limit the amount of paths that are calculated per frame to a fixed number and give every NavigationAgent a priority that determines which paths get recalculated first, and otherwise the NavigationAgent would run with old data.
- Third, it should be quite simple to add the option of OffMeshLinks by creating two points, calculating the relating nodes and using that to set those two nodes as neighbors. Once an agent reaches an off mesh link those two points are used in the path. It could even be implemented as a spline

Sources:

- http://www.blackpawn.com/texts/pointinpoly/
- https://github.com/Illation/LightWhat/blob/master/source/LightWhatRenderer/Geometry/Shapes/Mesh.cpp
- http://mathworld.wolfram.com/BarycentricCoordinates.html