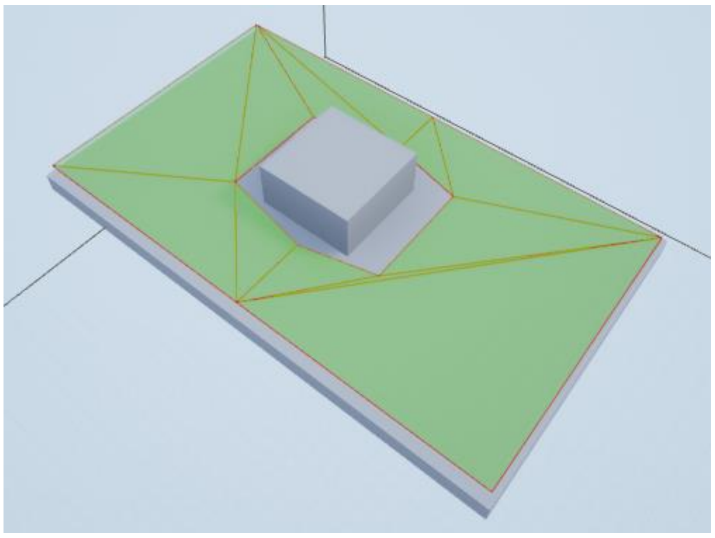
Navigation Mesh Generation

In this document I write about my progress in creating a custom navmesh for Unreal Engine and the decisions I have made. The goal is to learn about navigation mesh generation and to create a good piece for my portfolio. One of the biggest improvements I want to implement is 3-axis navmesh generation, Recast(Unreal's navmesh generator) can only generate navigation meshes on 2 axis. The 3 axis navigation mesh would be able to generate navigation meshes on the walls or ceiling. In this project I will probably use algorithms that are easier to implement at the cost of performance. I can always optimize when the generator is implemented.

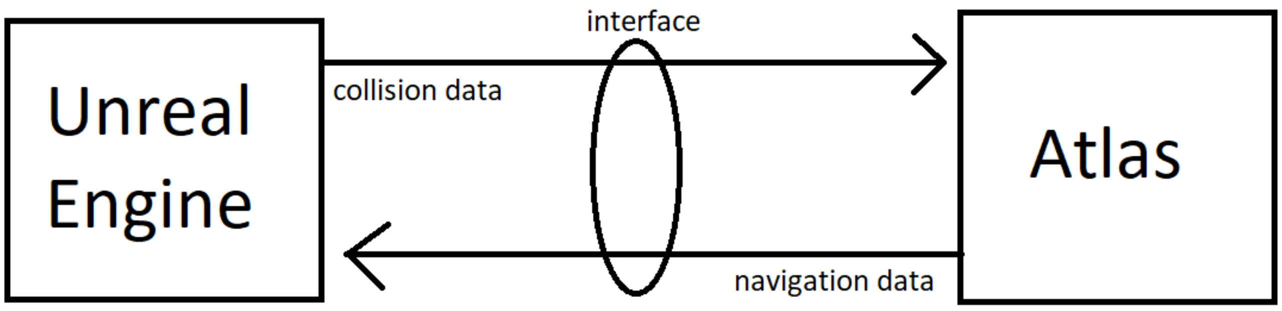


Breda University of Applied Sciences

Javid Ladhani
140629

Plan

My first step is to create a connection between Unreal Engine and my library(Atlas) using an interface. Atlas should receive collision data and give back a navigation mesh. The interface between the engine and Atlas makes sure that Atlas is not dependable on Unreal Engine and thus can be easily used for other projects or Engines.



After the connection has been made I can start implementing Atlas.

1. Implement a Heightfield Structure
2. Voxelize collision to the Heightfield.
3. Generate Open Heightfield
4. Generate a distance field and use it to build regions
5. Generate a contour mesh
6. Generate convex polygon mesh
7. Generate Detailed mesh

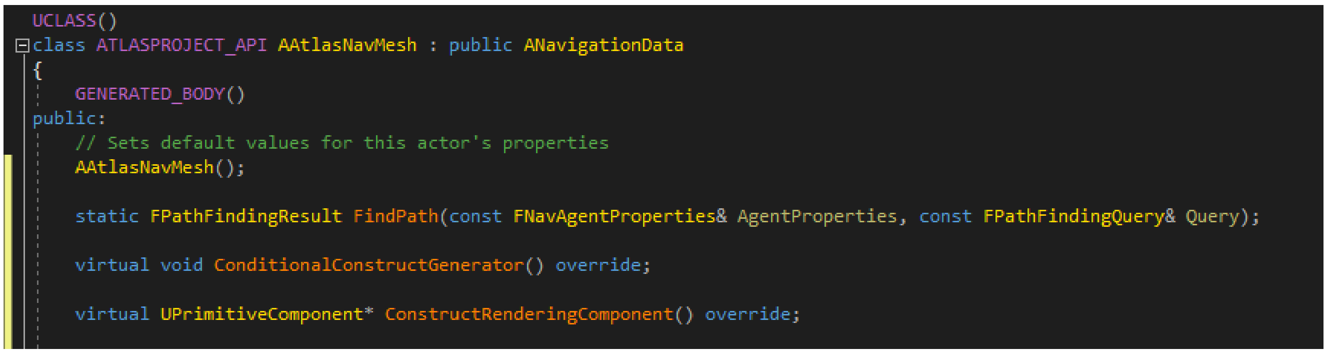
I start with a generator that uses 2 axis because I have no experience with building a navmesh Generator. When the generator with 2 axis is finished I will start implementing a 3 axis generator.

It may be necessary to build a pathfinder for Atlas since the pathfinder of Unreal Engine uses the Recast navmesh.

The implementation

The interface

In Unreal's settings a navigation generation system can be set for an agent type. This system has to be derived from ANavigationData. The class I created is called AAtlasNavMesh .



```
UCLASS()
class ATLASPROJECT_API AAtlasNavMesh : public ANavigationData
{
    GENERATED_BODY()
public:
    // Sets default values for this actor's properties
    AAtlasNavMesh();

    static FPathFindingResult FindPath(const FNavAgentProperties& AgentProperties, const FPathFindingQuery& Query);

    virtual void ConditionalConstructGenerator() override;

    virtual UPrimitiveComponent* ConstructRenderingComponent() override;
```

Code Snippet 1

AAtlasNavMesh has to inherit these three functions:

- **FindPath**

The navigation system calls this function whenever a agent tries to find a path.

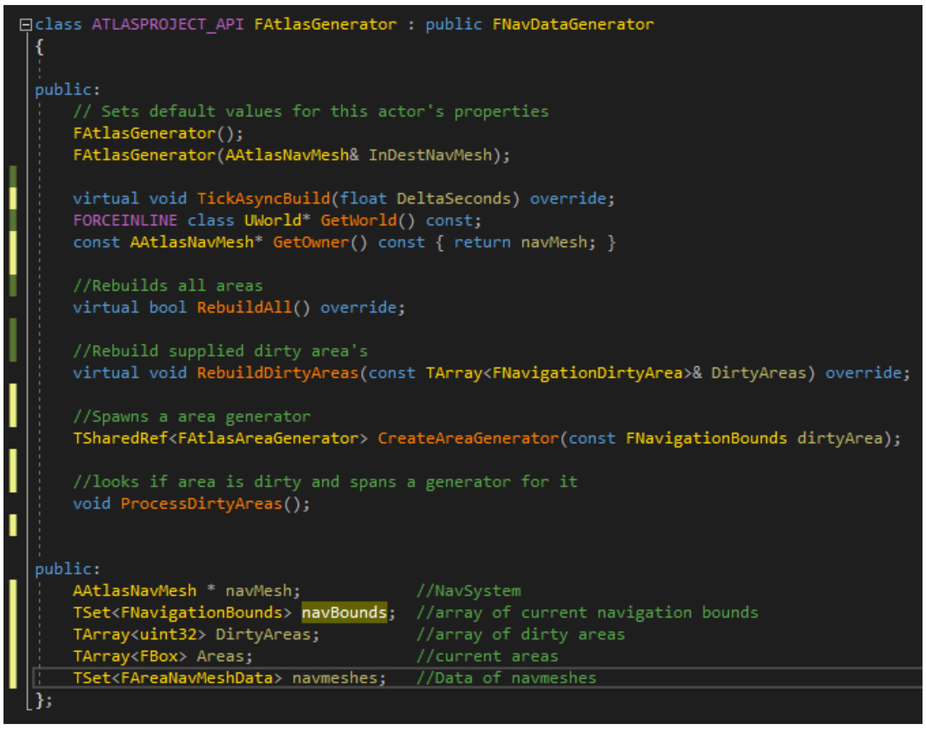
- **ConditionalConstructGenerator**

Constructs and sets the generator which generates the navmesh. This function is called at the start of the game or editor or whenever the generator needs to be refreshed. It is conditional because sometimes there is no need to spawn a generator, for example when runtime navigation generation is not supported it should not spawn a generator when not in the editor. The generator should be an object derived from FNavDataGenerator and is set to a variable called NavDataGenerator in ANavigationData.

- **ConstructRenderingComponent**

constructs a component that contains the debug rendering of the navmesh. This is called every time the rendering flag of the navigation system is set to dirty.

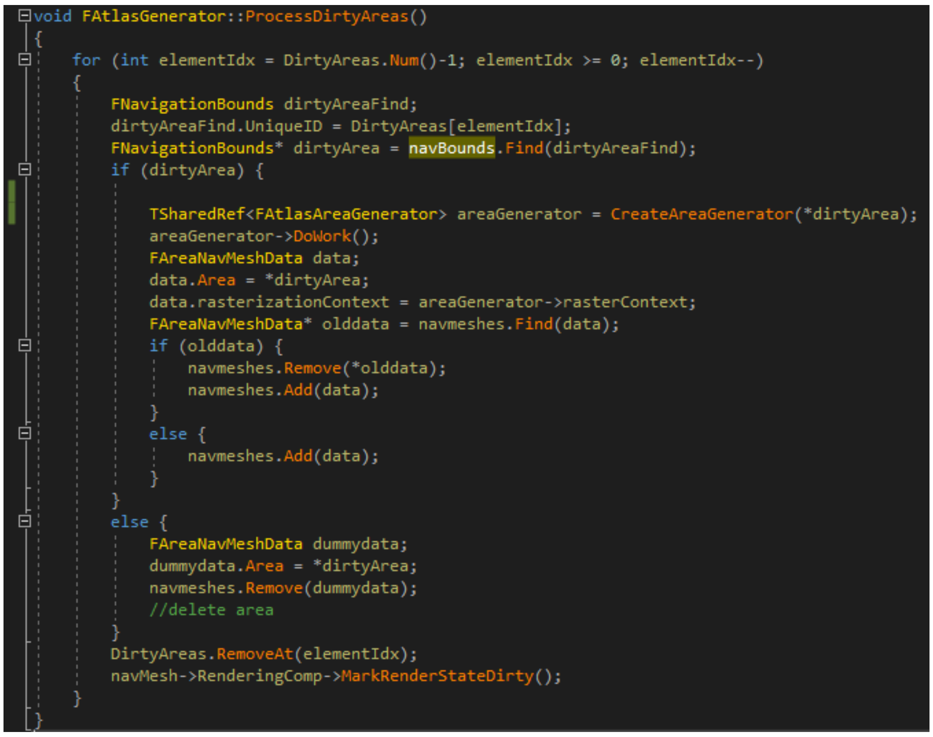
The most important function is ConditionalConstructGenerator because it spawns the generator which will eventually ask Atlas to generate a navigation mesh. I created FAtlasGenerator as seen in Code Snippet 2 and there are 2 ways currently implemented to rebuild areas. RebuildAll which gets called if the button Rebuild Navigation in the editor gets pressed and RebuildDirtyAreas which gets called when one or more navigation areas are moved.



```cpp
class ATLASPROJECT_API FAtlasGenerator : public FNavDataGenerator
{

public:
    // Sets default values for this actor's properties
    FAtlasGenerator();
    FAtlasGenerator(AAtlasNavMesh& InDestNavMesh);

    virtual void TickAsyncBuild(float DeltaSeconds) override;
    FORCEINLINE class UWorld* GetWorld() const;
    const AAtlasNavMesh* GetOwner() const { return navMesh; }

    //Rebuilds all areas
    virtual bool RebuildAll() override;

    //Rebuild supplied dirty area's
    virtual void RebuildDirtyAreas(const TArray<FNavigationDirtyArea>& DirtyAreas) override;

    //Spawns a area generator
    TSharedRef<FAtlasAreaGenerator> CreateAreaGenerator(const FNavigationBounds dirtyArea);

    //looks if area is dirty and spans a generator for it
    void ProcessDirtyAreas();


public:
    AAtlasNavMesh * navMesh;            //NavSystem
    TSet<FNavigationBounds> navBounds;  //array of current navigation bounds
    TArray<uint32> DirtyAreas;          //array of dirty areas
    TArray<FBox> Areas;                 //current areas
    TSet<FAreaNavMeshData> navmeshes;   //Data of navmeshes
};
```

Code Snippet 2

In both RebuildAll and RebuildDirtyAreas the DirtyAreas array gets filled with the dirty areas. Every tick ProcessDirtyAreas(Code Snippet 3) is called which picks up these dirty area's and spawns a sub generator for it.

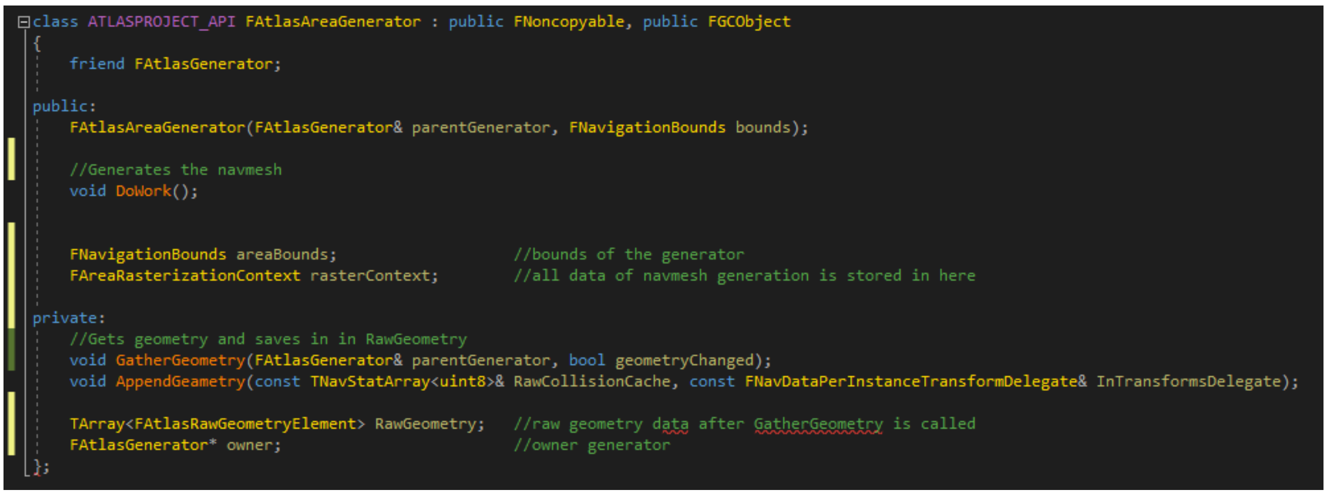


```cpp
void FAtlasGenerator::ProcessDirtyAreas()
{
    for (int elementIdx = DirtyAreas.Num()-1; elementIdx >= 0; elementIdx--)
    {
        FNavigationBounds dirtyAreaFind;
        dirtyAreaFind.UniqueID = DirtyAreas[elementIdx];
        FNavigationBounds* dirtyArea = navBounds.Find(dirtyAreaFind);
        if (dirtyArea) {

            TSharedRef<FAtlasAreaGenerator> areaGenerator = CreateAreaGenerator(*dirtyArea);
            areaGenerator->DoWork();
            FAreaNavMeshData data;
            data.Area = *dirtyArea;
            data.rasterizationContext = areaGenerator->rasterContext;
            FAreaNavMeshData* olddata = navmeshes.Find(data);
            if (olddata) {
                navmeshes.Remove(*olddata);
                navmeshes.Add(data);
            }
            else {
                navmeshes.Add(data);
            }
        }
        else {
            FAreaNavMeshData dummydata;
            dummydata.Area = *dirtyArea;
            navmeshes.Remove(dummydata);
            //delete area
        }
        DirtyAreas.RemoveAt(elementIdx);
        navMesh->RenderingComp->MarkRenderStateDirty();
    }
}
```

Code Snippet 3

It was not necessary to create a sub-generator for each dirty area. I could've just called Atlas code from here. But I want to extent this to multithreading later in the progress, in which I want to put each area generation in a different thread. This will speed up the generation and the editor will not get stuck when it is generating the nav meshes.

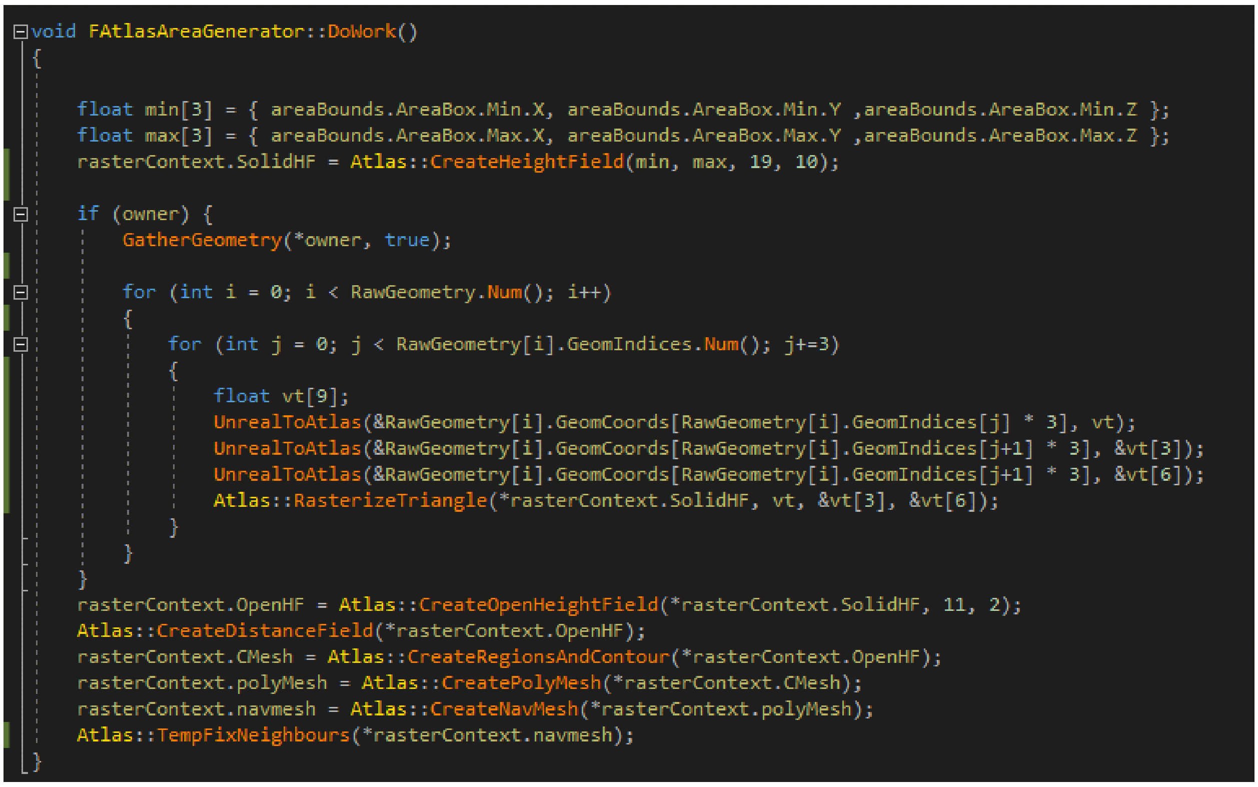
AreaGenerator is a simple class. Only data he needs is its NavigationBounds which is provided when constructing the object and a reference to the main generator. Then DoWork is called to generate the navigation mesh data which is then stored into rasterContext. The main generator can read the rasterContext when the AreaGenerator is done.



```cpp
class ATLASPROJECT_API FAtlasAreaGenerator : public FNoncopyable, public FGCObject
{
    friend FAtlasGenerator;

public:
    FAtlasAreaGenerator(FAtlasGenerator& parentGenerator, FNavigationBounds bounds);

    //Generates the navmesh
    void DoWork();


    FNavigationBounds areaBounds;                   //bounds of the generator
    FAreaRasterizationContext rasterContext;        //all data of navmesh generation is stored in here

private:
    //Gets geometry and saves in in RawGeometry
    void GatherGeometry(FAtlasGenerator& parentGenerator, bool geometryChanged);
    void AppendGeametry(const TNavStatArray<uint8>& RawCollisionCache, const FNavDataPerInstanceTransformDelegate& InTransformsDelegate);

    TArray<FAtlasRawGeometryElement> RawGeometry;   //raw geometry data after GatherGeometry is called
    FAtlasGenerator* owner;                         //owner generator
};
```

Code Snippet 4

The Atlas functions are called in the DoWork function which fill the rastercontext.



```cpp
void FAtlasAreaGenerator::DoWork()
{

    float min[3] = { areaBounds.AreaBox.Min.X, areaBounds.AreaBox.Min.Y ,areaBounds.AreaBox.Min.Z };
    float max[3] = { areaBounds.AreaBox.Max.X, areaBounds.AreaBox.Max.Y ,areaBounds.AreaBox.Max.Z };
    rasterContext.SolidHF = Atlas::CreateHeightField(min, max, 19, 10);

    if (owner) {
        GatherGeometry(*owner, true);

        for (int i = 0; i < RawGeometry.Num(); i++)
        {
            for (int j = 0; j < RawGeometry[i].GeomIndices.Num(); j+=3)
            {
                float vt[9];
                UnrealToAtlas(&RawGeometry[i].GeomCoords[RawGeometry[i].GeomIndices[j] * 3], vt);
                UnrealToAtlas(&RawGeometry[i].GeomCoords[RawGeometry[i].GeomIndices[j+1] * 3], &vt[3]);
                UnrealToAtlas(&RawGeometry[i].GeomCoords[RawGeometry[i].GeomIndices[j+1] * 3], &vt[6]);
                Atlas::RasterizeTriangle(*rasterContext.SolidHF, vt, &vt[3], &vt[6]);
            }
        }
    }
    rasterContext.OpenHF = Atlas::CreateOpenHeightField(*rasterContext.SolidHF, 11, 2);
    Atlas::CreateDistanceField(*rasterContext.OpenHF);
    rasterContext.CMesh = Atlas::CreateRegionsAndContour(*rasterContext.OpenHF);
    rasterContext.polyMesh = Atlas::CreatePolyMesh(*rasterContext.CMesh);
    rasterContext.navmesh = Atlas::CreateNavMesh(*rasterContext.polyMesh);
    Atlas::TempFixNeighbours(*rasterContext.navmesh);
}
```

Code Snippet 5

The heightfield Structure

A heightfield is formed when voxelizing an object and connecting all voxels connecting on top and bottom of a voxel to create spans. An example heightfield is shown in figure 1, a red span represents collision.

First I need to define the structure of a heightfield. It needs to include the size of the heightfield in world units and the size of the voxels. It also needs to hold the spans.

spans is an array of pointers to spans which size is the width times the depth of the field. If a pointer to a span is a nullptr then there is no span on that location. The spans have a pointer to a span on the same x and y position but higher in z.

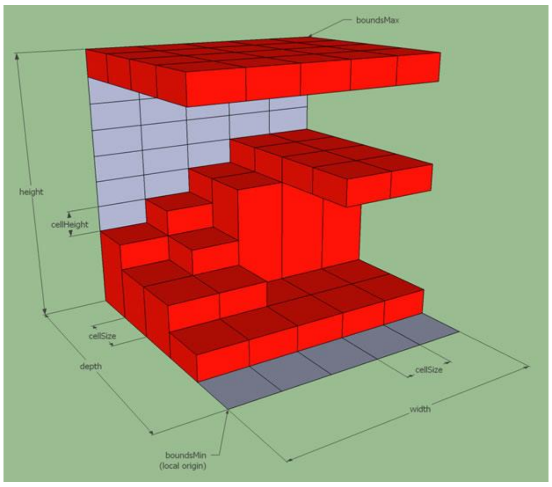
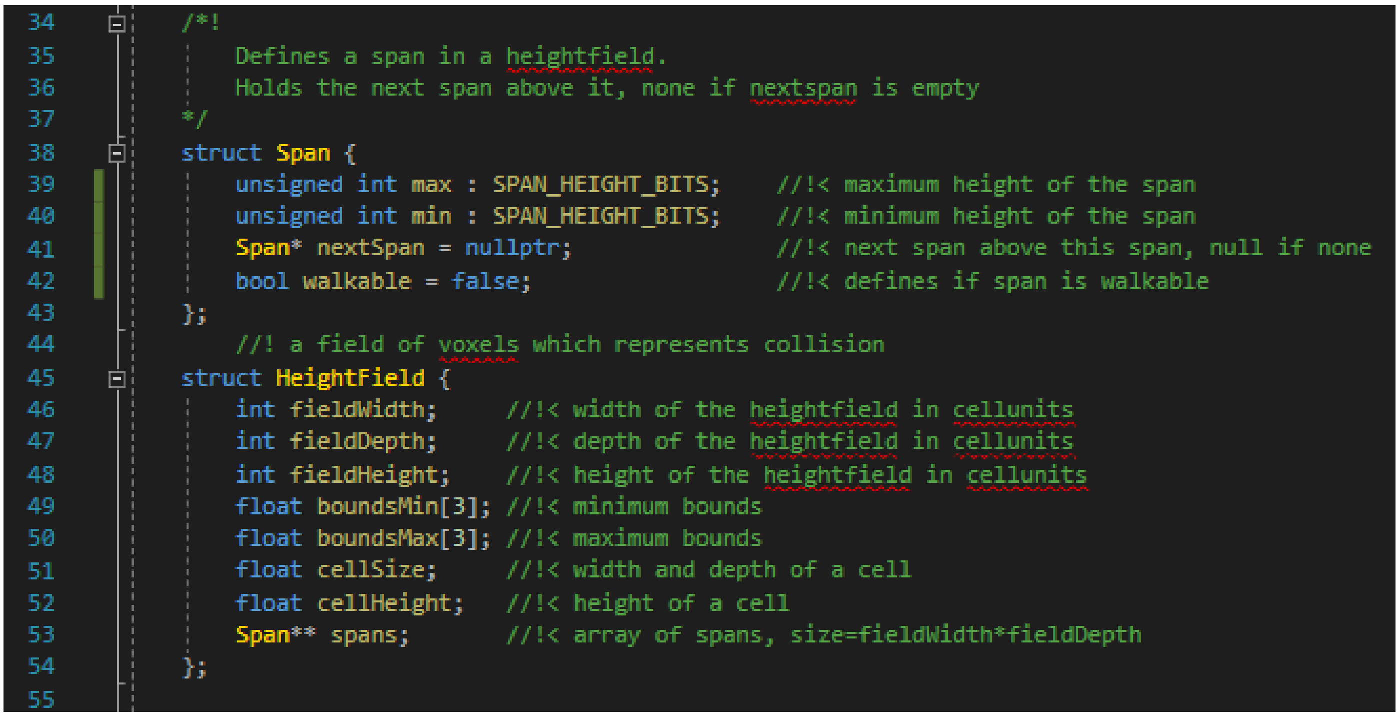


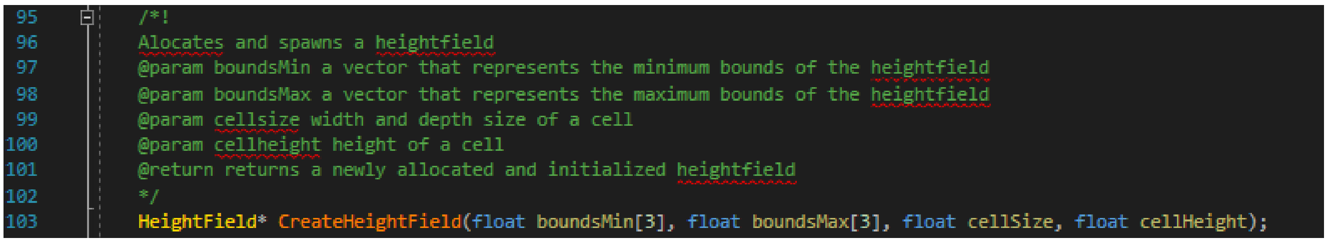
Figure 1. A Heightfield. Retrieved from http://www.critterai.org/projects/nmgen_study/heightfields.html



```
/*!
    Defines a span in a heightfield.
    Holds the next span above it, none if nextspan is empty
*/
struct Span {
    unsigned int max : SPAN_HEIGHT_BITS;    //!< maximum height of the span
    unsigned int min : SPAN_HEIGHT_BITS;    //!< minimum height of the span
    Span* nextSpan = nullptr;               //!< next span above this span, null if none
    bool walkable = false;                  //!< defines if span is walkable
};
    //! a field of voxels which represents collision
struct HeightField {
    int fieldWidth;     //!< width of the heightfield in cellunits
    int fieldDepth;     //!< depth of the heightfield in cellunits
    int fieldHeight;    //!< height of the heightfield in cellunits
    float boundsMin[3]; //!< minimum bounds
    float boundsMax[3]; //!< maximum bounds
    float cellSize;     //!< width and depth of a cell
    float cellHeight;   //!< height of a cell
    Span** spans;       //!< array of spans, size=fieldWidth*fieldDepth
};
```

Code Snippet 6. Link

The heightfield needs to be allocated and initialized with CreateHeightfield which sets the variables and allocates a chunk of data for the spans.



```
/*!
Alocates and spawns a heightfield
@param boundsMin a vector that represents the minimum bounds of the heightfield
@param boundsMax a vector that represents the maximum bounds of the heightfield
@param cellsize width and depth size of a cell
@param cellheight height of a cell
@return returns a newly allocated and initialized heightfield
*/
HeightField* CreateHeightField(float boundsMin[3], float boundsMax[3], float cellSize, float cellHeight);
```

Code Snippet 7. [Link]

Voxelizing

Voxelizing is converting a 3d model based on triangles to a 3d model based on cube volumes. Voxelizing an object is done by voxelizing every triangle in that object. There are many different methods to Voxelize an object but I am using the Sutherland-Hodgman algorithm to create the voxels.

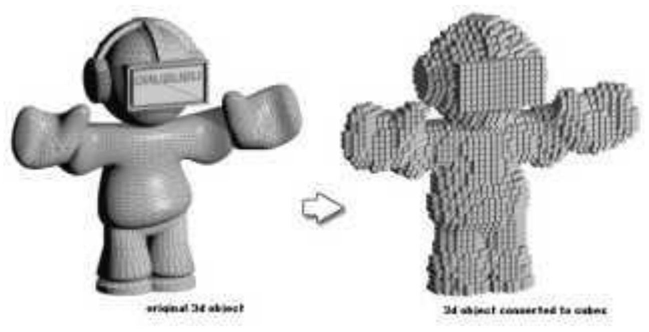


Figure 2. Voxelizing an object. Retrieved from https://static.giantbomb.com/uploads/scale_small/0/2891/804743-voxel_pixelart.jpg

The Sutherland-Hodgman algorithm works by clipping a polygon in a box looping though all sides of the box like seen in Figure 3. When a side of the box is being examined it loops though all the lines in the polygon and saves the new polygon. It follows these rules.

1. When the edge is inside, the second vertex is saved
2. When the edge is leaving, the intersection is saved
3. When the edge is outside, nothing is saved
4. When the edge is entering, save the intersection and the second vertex.

I am using the definition “inside” and “outside” in the ruleset. When examining the edge of the box, the side of the box is the inside. The other side is outside.

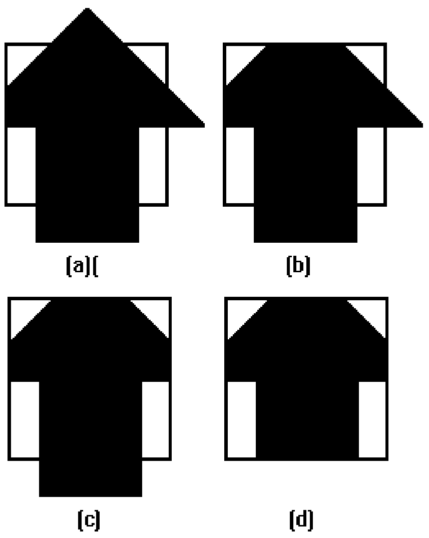


Figure 3. Sutherland-Hodgman algorithm. Adapted and retrieved from https://www.cs.helsinki.fi/group/goa/viewing/leikkaus/FIG2.GIF

I have expanded this algorithm to 3d. which clips a 3d triangle in a 3d box. I have replaced the line-line intersection with a line-plane intersection algorithm which is not exactly a line-plane algorithm. In my algorithm it can only calculate planes which are axis aligned to make the algorithm cheaper. Figure 4 shows an image from this video, in this video the red polygon is the resulted clipped polygon from the green triangle.

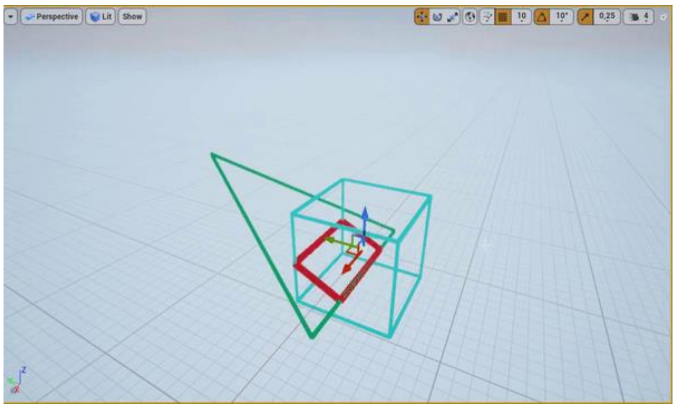


Figure 4. clipping triangle

So if we run this algorithm and there are no vertices left that means the triangle doesn't collide with the voxel? Correct but if we have to do this for every voxel it can be rather expansive and I could've used a much cheaper algorithm to determine if the triangle was hitting the box. The big benefit using this algorithm is that it is possible to get the top and bottom vertex from the clipped polygon. Instead of clipping the triangle to a voxel, I clip the triangle to a box which represents a column of voxels. Every voxel between the top and bottom vertex from the clipped triangle is a solid voxel like seen in figure 5.

This way I can loop though all columns underneath the triangle like seen in figure 4 and use the algorithm to find the solid voxels.

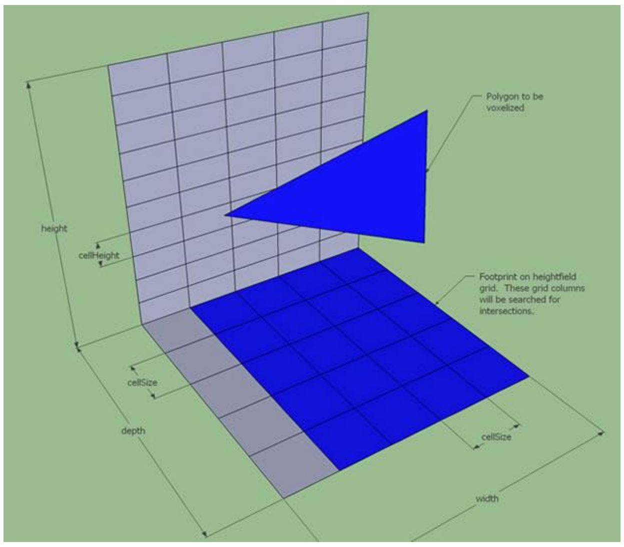

Polygon to be voxelized
height
cellHeight
Footprint on heightfield grid. These grid columns will be searched for intersections.
cellSize
depth
cellSize
width

Figure 6. Footprint of triangle. Retrieved from http://www.critterai.org/projects/nmgen_study/media/images/shfg_01_aabb.jpg

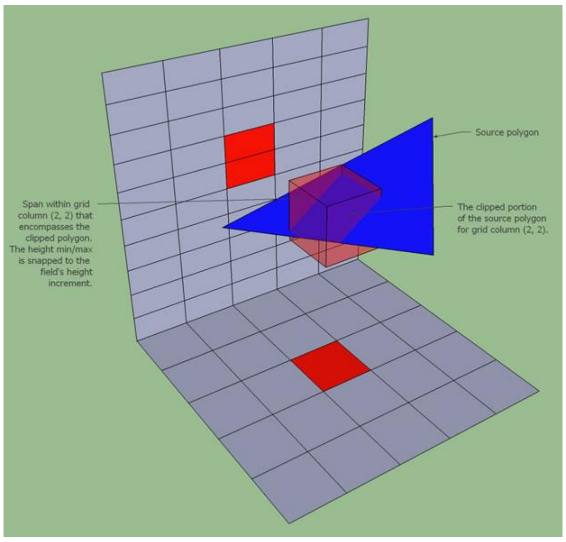

Source polygon
Span within grid column (2, 2) that encompasses the clipped polygon. The height min/max is snapped to the field's height increment.
The clipped portion of the source polygon for grid column (2, 2).

Figure 5. Voxels of triangle. Retrieved from http://www.critterai.org/projects/nmgen_study/media/images/shfg_02_clipping.jpg

In Figure 8 we see the result of a voxelized rotated cube. The green boxes are the combined voxels called spans. There is an empty space inside the cube because there is no mesh to make solid spans. There are still spans at the bottom of the cube which surface is inside the cube. There is no need to create a navigation mesh on these surfaces since we don't want any navigation inside the cube but these spans can't be deleted because they are needed in the open heightmap. The solution is to set their state to not walkable when the triangle which generated the voxel is facing down, this is represented by rendering the span red.

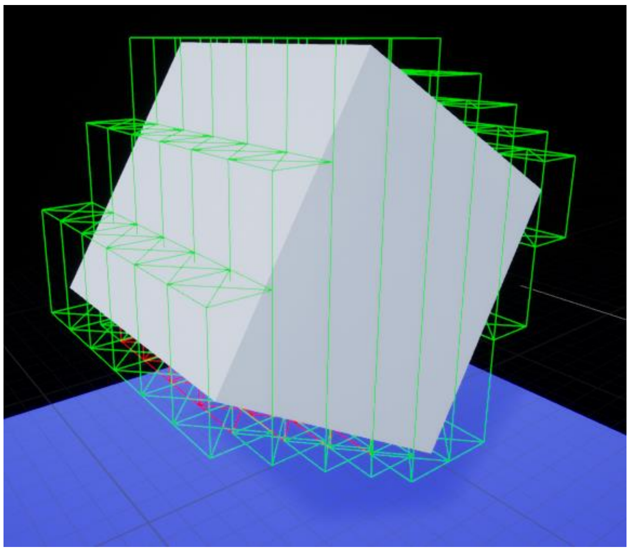


Figure 8. A voxelized rotated cube

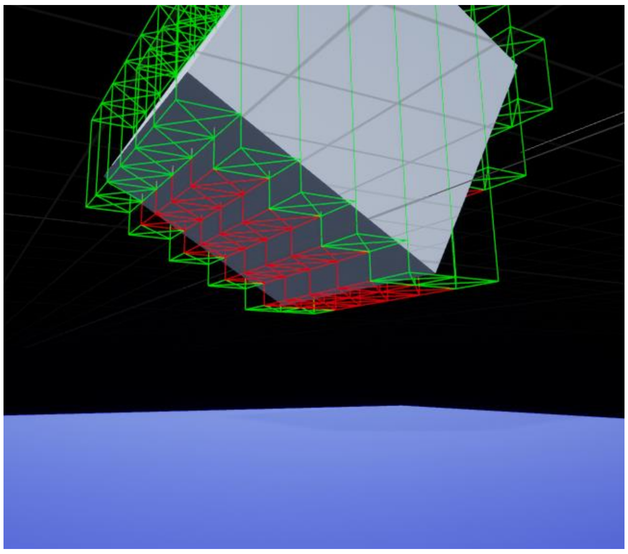
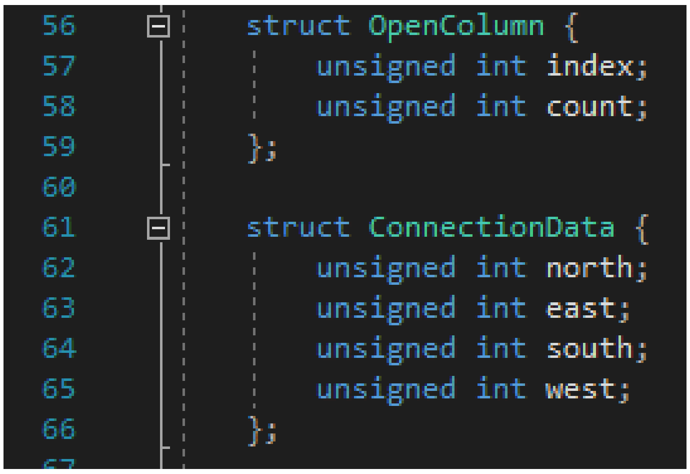


Figure 7. The bottom of a voxelized rotated cube showing the non-walkable spans

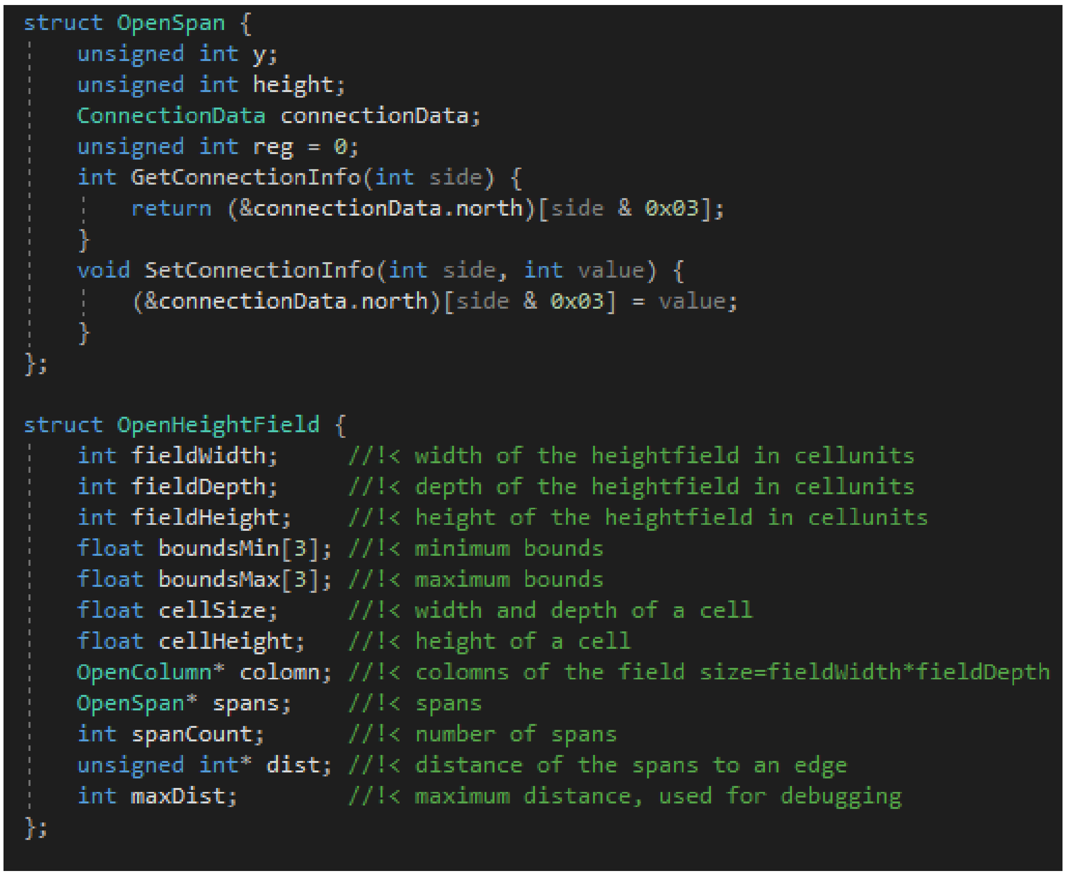
Open Heightfield

An open heightfield is the inverse of a heightfield. It defines the open spaces in an area. The structure is different from the normal Heightfield because it integrates neighbours into the spans.



```
struct OpenColumn {
    unsigned int index;
    unsigned int count;
};

struct ConnectionData {
    unsigned int north;
    unsigned int east;
    unsigned int south;
    unsigned int west;
};
```

Code Snippet 8



```
struct OpenSpan {
    unsigned int y;
    unsigned int height;
    ConnectionData connectionData;
    unsigned int reg = 0;
    int GetConnectionInfo(int side) {
        return (&connectionData.north)[side & 0x03];
    }
    void SetConnectionInfo(int side, int value) {
        (&connectionData.north)[side & 0x03] = value;
    }
};

struct OpenHeightField {
    int fieldWidth;      //!< width of the heightfield in cellunits
    int fieldDepth;      //!< depth of the heightfield in cellunits
    int fieldHeight;     //!< height of the heightfield in cellunits
    float boundsMin[3]; //!< minimum bounds
    float boundsMax[3]; //!< maximum bounds
    float cellSize;      //!< width and depth of a cell
    float cellHeight;    //!< height of a cell
    OpenColumn* colomn; //!< colomns of the field size=fieldWidth*fieldDepth
    OpenSpan* spans;     //!< spans
    int spanCount;       //!< number of spans
    unsigned int* dist; //!< distance of the spans to an edge
    int maxDist;         //!< maximum distance, used for debugging
};
```

Code Snippet 9. Link

In the Heightfield class the spans are stored in a double array which causes a lot of memory jumping when iterated though. But it was necessary to have a dynamic sizable list because it did not know how many spans are being generated. Now that we know the exact amount of spans it can be compressed into a single array.

I could have stored the location of the spans inside the spans itself but that makes the data bigger and it is harder to iterate though locations. We can iterate though location by getting the correct columns index (column[x+y*fieldWidth]). Inside a column the index of the first span in the spans array is stored along with the amount of spans. So for example column[5+2*fieldWidth] has index 5 and count 3, this means the spans[5] to spans[5+3] are the spans for that location.

The spans now also include the connection data to neighbours which are walkable from this span. The neighbour is considered walkable if the following conditions are met:

- The step up or down between the floors of the two spans is less than the set value(Figure 10)
- The gap between the two spans is big enough to be travers by an agent(Figure 10)

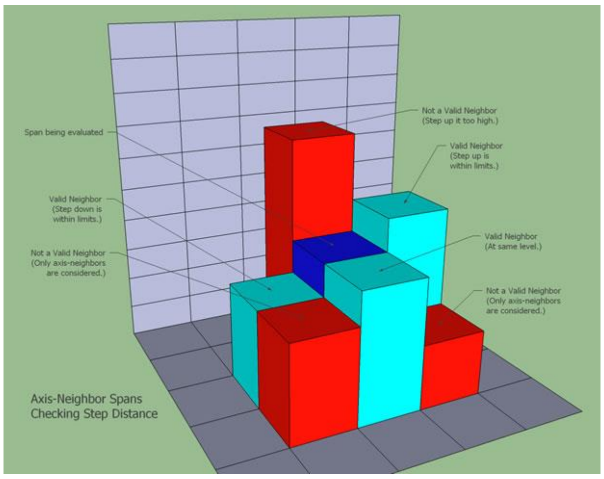

Span being evaluated
Not a Valid Neighbor (Step up it too high.)
Valid Neighbor (Step up is within limits.)
Valid Neighbor (Step down is within limits.)
Valid Neighbor (At same level.)
Not a Valid Neighbor (Only axis-neighbors are considered.)
Not a Valid Neighbor (Only axis-neighbors are considered.)
Axis-Neighbor Spans Checking Step Distance

Figure 10. step up/down neighbours. Retrieved from :http://www.critterai.org/projects/nmgen_study/media/images/ohfg_02_steplinks.jpg

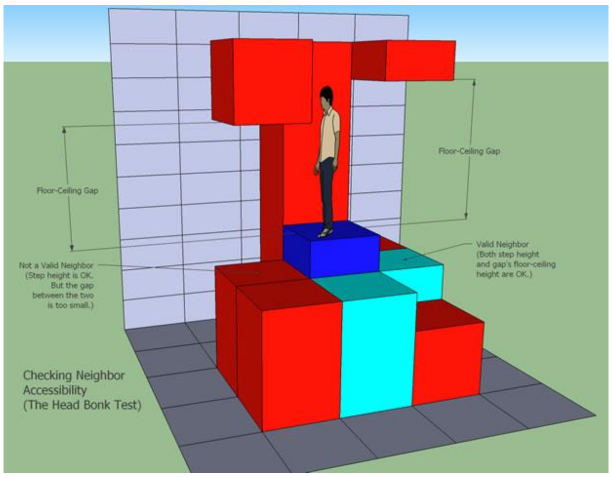

Floor-Ceiling Gap
Floor-Ceiling Gap
Not a Valid Neighbor (Step height is OK. But the gap between the two is too small.)
Valid Neighbor (Both step height and gap's floor-ceiling height are OK.)
Checking Neighbor Accessibility (The Head Bonk Test)

Figure 10. head bump. Retrieved from :http://www.critterai.org/projects/nmgen_study/media/images/ohfg_03_headbonk.jpg

In my further examples only the bottom of the open spans are shown in images. The spans still include the height but is easier to visualize when only its floor is drawn.

In Figure 11 we can see the open height field with a mannequin in it. Spans are generated on places that are actually high enough for the character to pass though.

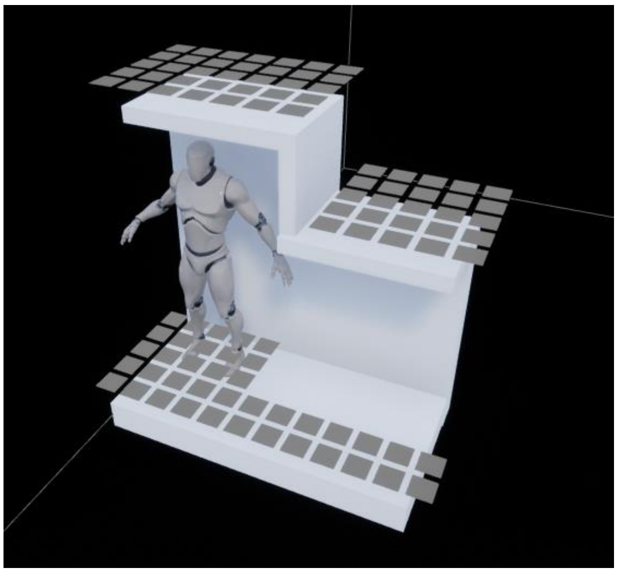
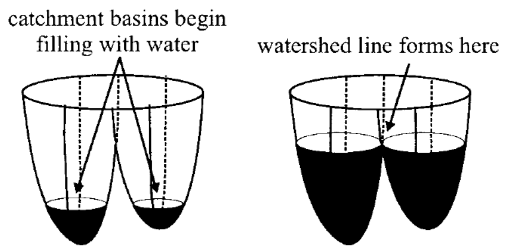


Figure 11. An open height field with mannequin

Region Generation

Now that a walkable area has been generated it needs to be transformed into a polygon mesh. To define the polygons the open heightfield needs to be divided into regions. I'll be using the watershed algorithm for this. This algorithm separates adjacent drainage basins by slowly flooding the area and build barriers where the different water sources meet.


catchment basins begin filling with water
watershed line forms here

Since the open heightfield doesn't have many height differences on a flat area I am using the distance to a border span as "flood height". A border span is a span with less than 4 neighbours. So border spans are the last to flood and the spans furthers away from the border spans are the first to flood.

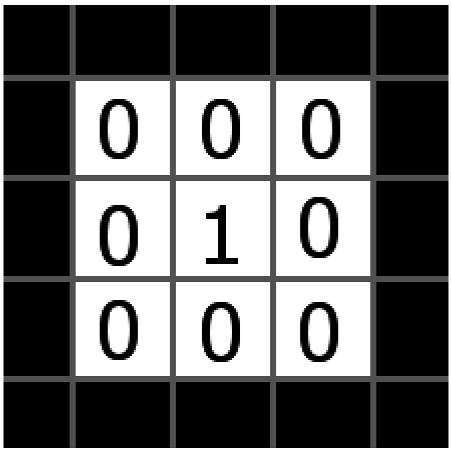


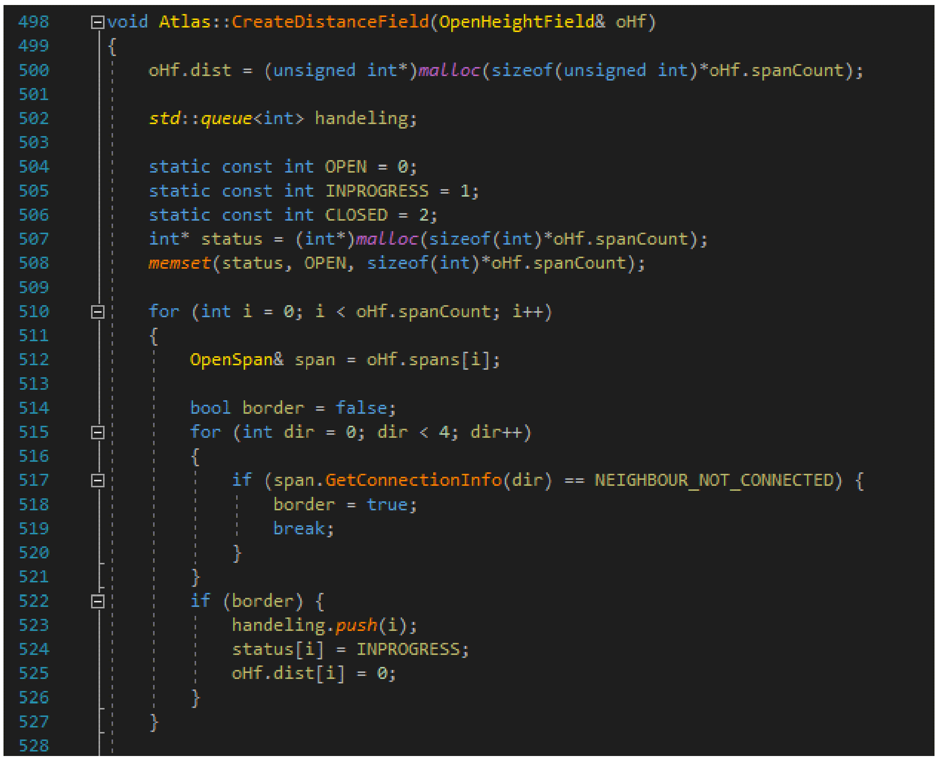
Figure 12. Distance field example

Distance field generation

A distance field is generated from the open heightfield. It represents how far each span from the open heightfield is from the border. A span is considered a border when it has at less than 4 neighbours.

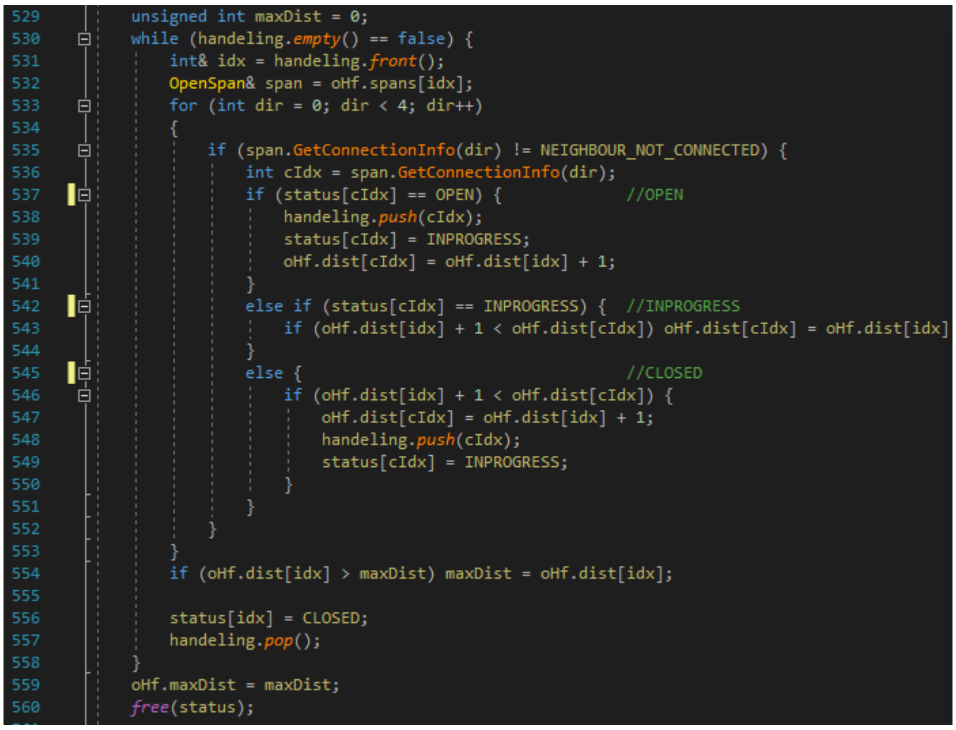
First I need to get the border spans. A span can have on of 3 states: OPEN,INPROGRESS,CLOSED. All the spans start open and they should all be closed at the end of the algorithm.

First it needs to get the border spans by checking every span if they have less than 4 neighbours. Their status is set to INPROGRESS and they are put into the handling list.



```
void Atlas::CreateDistanceField(OpenHeightField& oHf)
{
    oHf.dist = (unsigned int*)malloc(sizeof(unsigned int)*oHf.spanCount);

    std::queue<int> handeling;

    static const int OPEN = 0;
    static const int INPROGRESS = 1;
    static const int CLOSED = 2;
    int* status = (int*)malloc(sizeof(int)*oHf.spanCount);
    memset(status, OPEN, sizeof(int)*oHf.spanCount);

    for (int i = 0; i < oHf.spanCount; i++)
    {
        OpenSpan& span = oHf.spans[i];

        bool border = false;
        for (int dir = 0; dir < 4; dir++)
        {
            if (span.GetConnectionInfo(dir) == NEIGHBOUR_NOT_CONNECTED) {
                border = true;
                break;
            }
        }
        if (border) {
            handeling.push(i);
            status[i] = INPROGRESS;
            oHf.dist[i] = 0;
        }
    }

```

Code Snippet 10



```
    unsigned int maxDist = 0;
    while (handeling.empty() == false) {
        int& idx = handeling.front();
        OpenSpan& span = oHf.spans[idx];
        for (int dir = 0; dir < 4; dir++)
        {
            if (span.GetConnectionInfo(dir) != NEIGHBOUR_NOT_CONNECTED) {
                int cIdx = span.GetConnectionInfo(dir);
                if (status[cIdx] == OPEN) {             //OPEN
                    handeling.push(cIdx);
                    status[cIdx] = INPROGRESS;
                    oHf.dist[cIdx] = oHf.dist[idx] + 1;
                }
                else if (status[cIdx] == INPROGRESS) {  //INPROGRESS
                    if (oHf.dist[idx] + 1 < oHf.dist[cIdx]) oHf.dist[cIdx] = oHf.dist[idx]
                }
                else {                                  //CLOSED
                    if (oHf.dist[idx] + 1 < oHf.dist[cIdx]) {
                        oHf.dist[cIdx] = oHf.dist[idx] + 1;
                        handeling.push(cIdx);
                        status[cIdx] = INPROGRESS;
                    }
                }
            }
        }
        if (oHf.dist[idx] > maxDist) maxDist = oHf.dist[idx];

        status[idx] = CLOSED;
        handeling.pop();
    }
    oHf.maxDist = maxDist;
    free(status);
```

Code Snippet 11

Then the main part of the algorithm gets the first span out of the handling queue sets it to closed and applies the following rules to its neighbours:

- If it is **OPEN** it will be placed in handling and it's distance will be updated.
- If it is **INPROGRESS** then the distance may be updated if the new distance is smaller.
- If it is **CLOSED** and it's distance needs to be updated because it is smaller than the distance will be updated and the neighbour is put in the handling list again because it's neighbours may require updating.

This continuous until the handling queue is empty

With this algorithm a distance field is generated(Figure 13). The spans closer to the edge are darker then spans further away from the edge

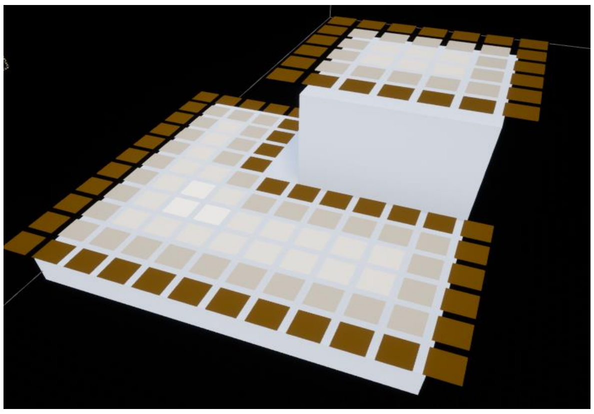
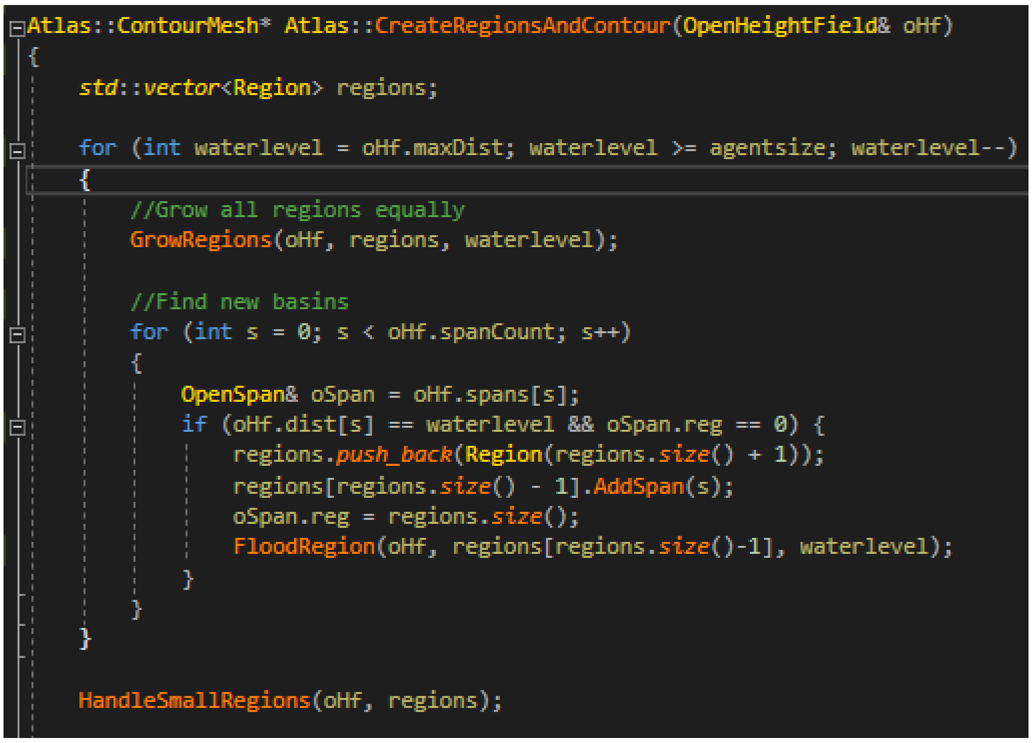


Figure 13. A distance field

The watershed algorithm

To create regions the distance field will be slowly flooded with the following rules each iteration:

1. Flood current basins equally
2. Find new basins with the current water height and fill them.
3. Repeat



```
Atlas::ContourMesh* Atlas::CreateRegionsAndContour(OpenHeightField& oHf)
{
    std::vector<Region> regions;

    for (int waterlevel = oHf.maxDist; waterlevel >= agentsize; waterlevel--)
    {
        //Grow all regions equally
        GrowRegions(oHf, regions, waterlevel);

        //Find new basins
        for (int s = 0; s < oHf.spanCount; s++)
        {
            OpenSpan& oSpan = oHf.spans[s];
            if (oHf.dist[s] == waterlevel && oSpan.reg == 0) {
                regions.push_back(Region(regions.size() + 1));
                regions[regions.size() - 1].AddSpan(s);
                oSpan.reg = regions.size();
                FloodRegion(oHf, regions[regions.size()-1], waterlevel);
            }
        }
    }

    HandleSmallRegions(oHf, regions);
```

Code Snippet 12

Code Snippet 12 loops though all waterlevels from on the distance field from high to low. It starts at the maximum waterlevel and ends at charactersize. Characters have a certain width and there is no need to generate a navigation mesh next to the wall when the character can't walk there.

For every waterlevel GrowRegions is called which grows regions to the new level. This needs to be done equally if not then a region can have domination over another region based on who gets updated first as seen in Figure 13. After aall regions are grown equally it searches for spans with that height and if they have no region yet a new region will be created. This new region floods its neighbours with the same height until there are no neighbours with that height left anymore.

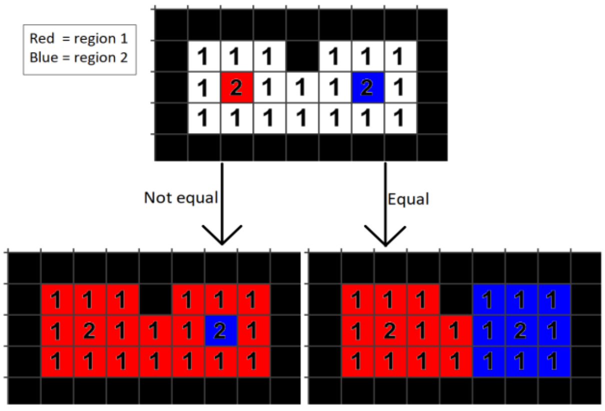


Figure 14. Equal region regeneration

After all spans have a regions HandleSmallRegions is called. Which tries to merge small regions with its neighbours regions. If it can't merge regions it will delete them.

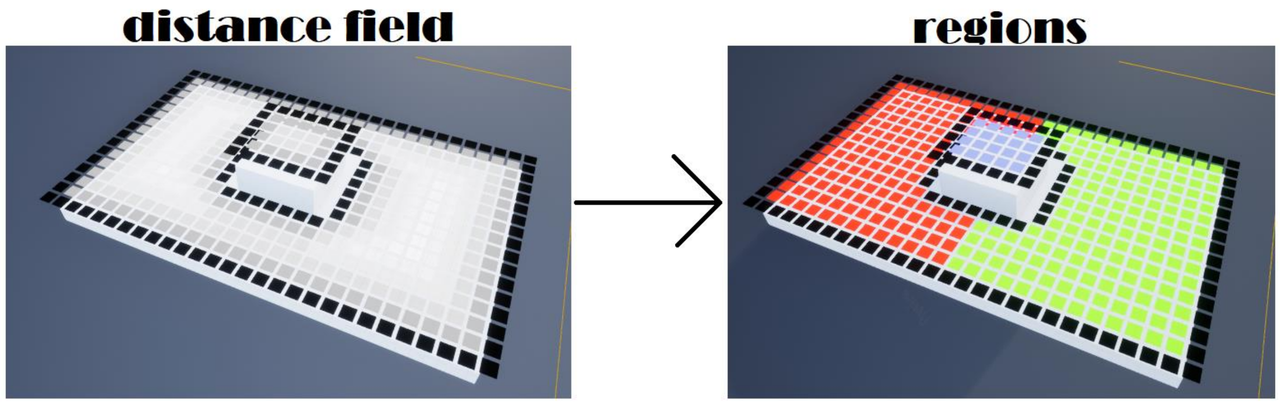
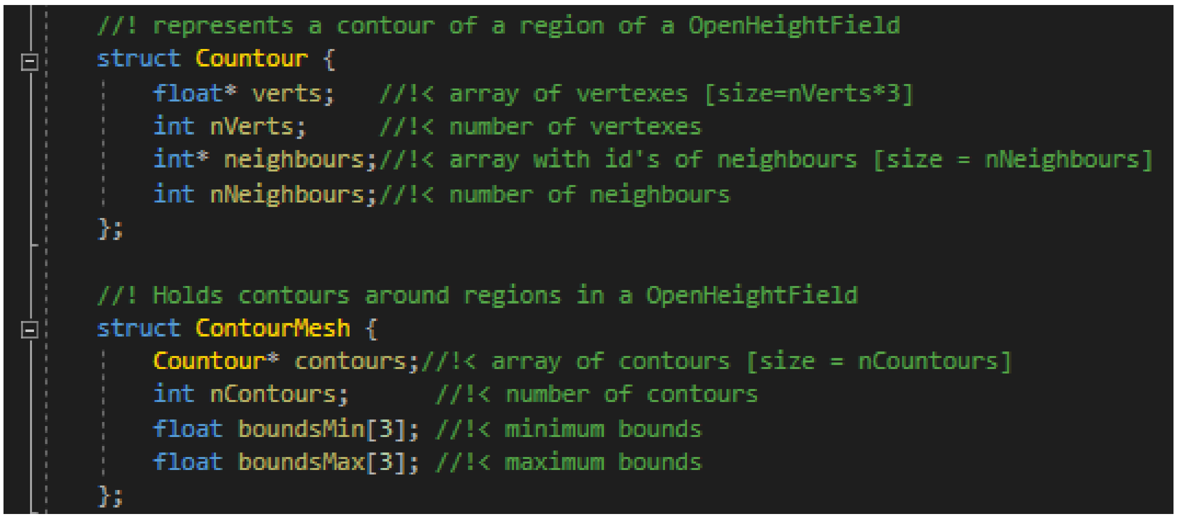


Figure 15. Region generation

Contour Generation

The regions can now be used to generate a contour mesh which has a contour in it for each region. The contourmesh has an array of Contours in it. Contours have an array of vertices and an array of neighbours.



```
//! represents a contour of a region of a OpenHeightField
struct Countour {
    float* verts;   //!< array of vertexes [size=nVerts*3]
    int nVerts;     //!< number of vertexes
    int* neighbours;//!< array with id's of neighbours [size = nNeighbours]
    int nNeighbours;//!< number of neighbours
};

//! Holds contours around regions in a OpenHeightField
struct ContourMesh {
    Countour* contours;//!< array of contours [size = nCountours]
    int nContours;      //!< number of contours
    float boundsMin[3]; //!< minimum bounds
    float boundsMax[3]; //!< maximum bounds
};
```

Code Snippet 13

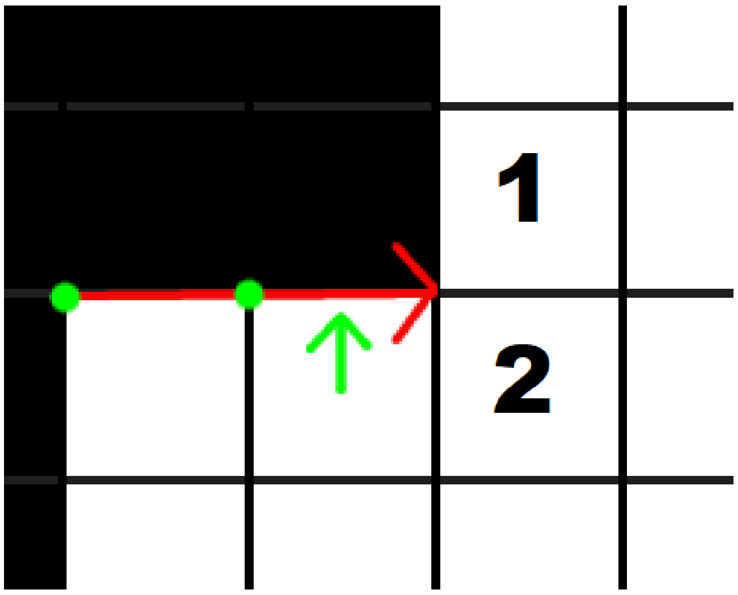


Figure 16. Region edge walking

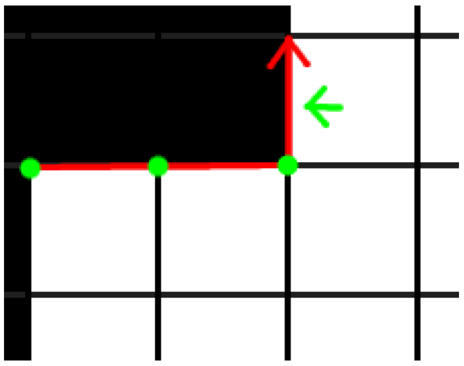


Figure 17. Edge walking case 1

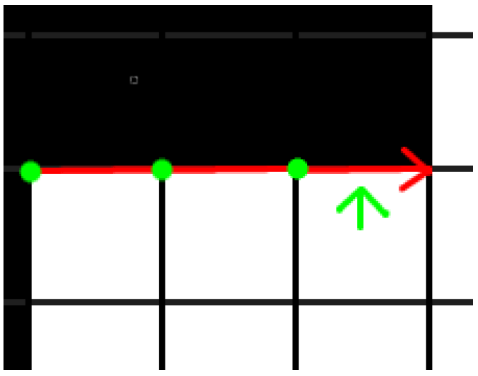


Figure 18.Edge walking case 2

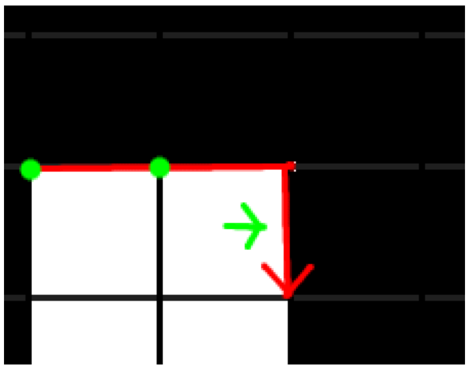


Figure 19.Edge walking case 3

A contour is generated by walking over the edge of a region as seen in **Fout! Verwijzingsbron niet gevonden.**. Here we see edge walking in progress. Currently our last edge we walked on is facing right and has a normal facing up because that is the side which has a span that not belongs to the region. There are 2 important spans which are defined as span 1 and span 2, it depends on these two spans where our next edge should be.

1. If both spans are owned by the region then our next edge is going outwards, in this case up.(Figure 17)
2. If span 1 is not owned by the region but span 2 is then the edge continues on its path, in this case right.(Figure 18)
3. If both spans are not owned by the region then our next edge is going inwards.(Figure 19)

Now that an contour* is generated it still needs to be simplified. There are now way too many vertices in the contour which we don't need. Even on straight lines there is a vertex for each span.

The only mandatory vertices are the vertices where there is a change in region connection. Between these mandatory vertices are portals. Portals can either have a region-region connection or a region-null connection.

Both portals are treated differently when it's simplified. Region-region portals are simplified by discarding all the vertices between the mandatory vertices. Region-null portals are a bit trickier. These portals need to keep their shape.

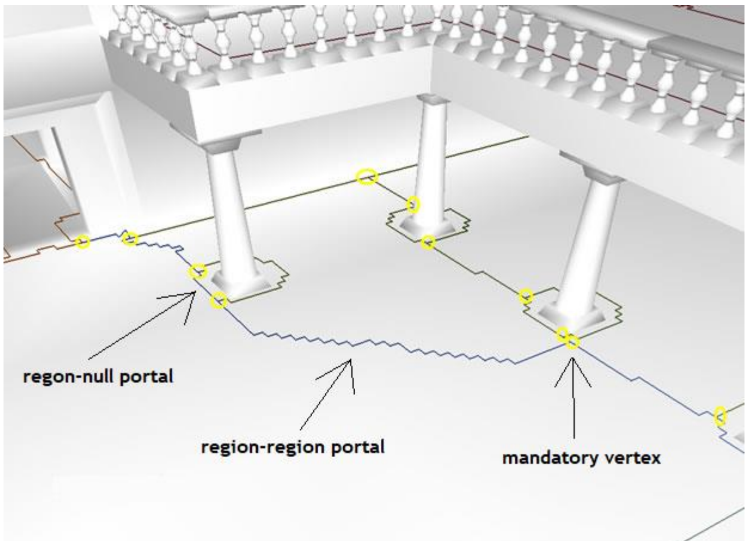


Figure 20. portals

The goal of the algorithm is to keep the shape of the edge between the mandatory vertices with as few vertices as possible. I use the Douglas-Peucker algorithm to achieve this. It starts off with just the mandatory vertices(Figure 21) and it searches for the vertex that is furthest away from the simplified edge. If the distance between the edge and vertex is bigger than maxdeviaton then the simplified edge splits in 2 with the vertex as middle point(Figure 22). It repeats this progress until all vertices have a smaller distance to the simplified edge then maxdeviation(Figure 23).

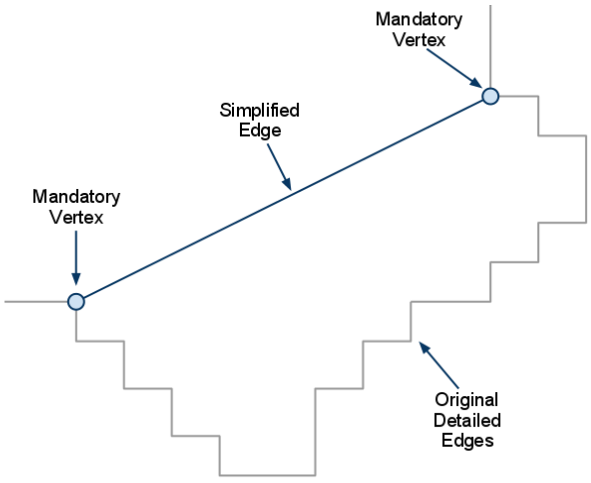


Figure 21. douglas-peucker algorithm

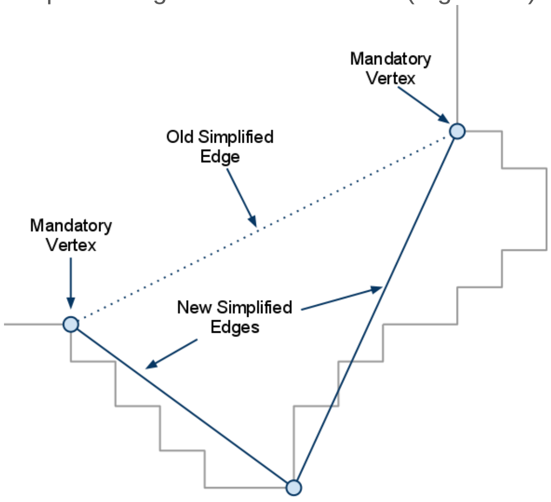


Figure 22. douglas-peucker algorithm 2

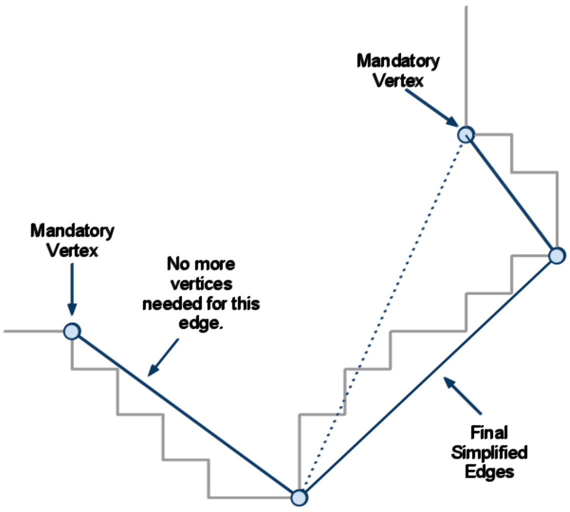
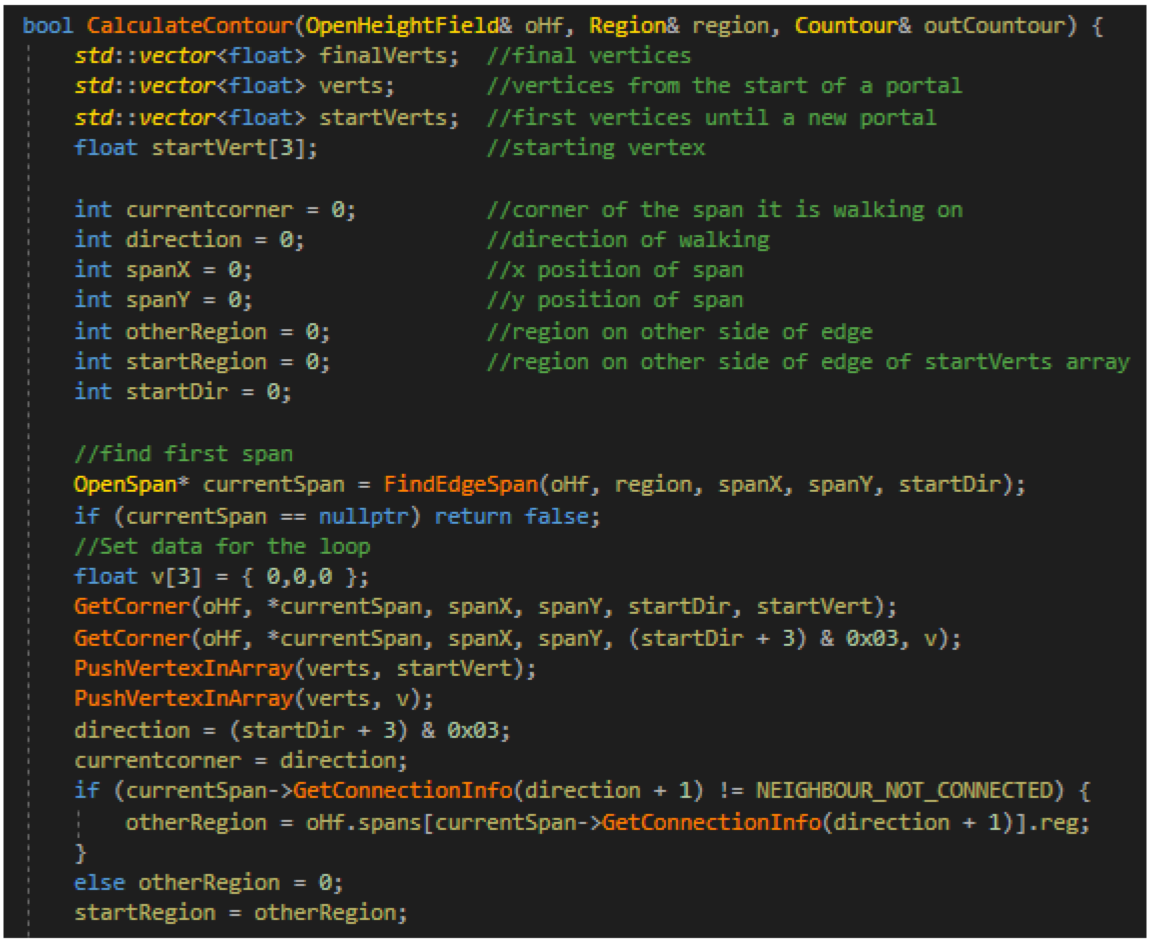


Figure 23. douglas-peucker algorithm 3

The algorithm is rather big so I divide it into 4 parts:

- setup
- edge walking
- portal cleaning
- finalizing

All the variables in code are initialized and set. It searched for a span that lies next to a other region or no region and saved as the current span. The first 2 vertices are saved into the verts array and other variables are set based on the currentspan.

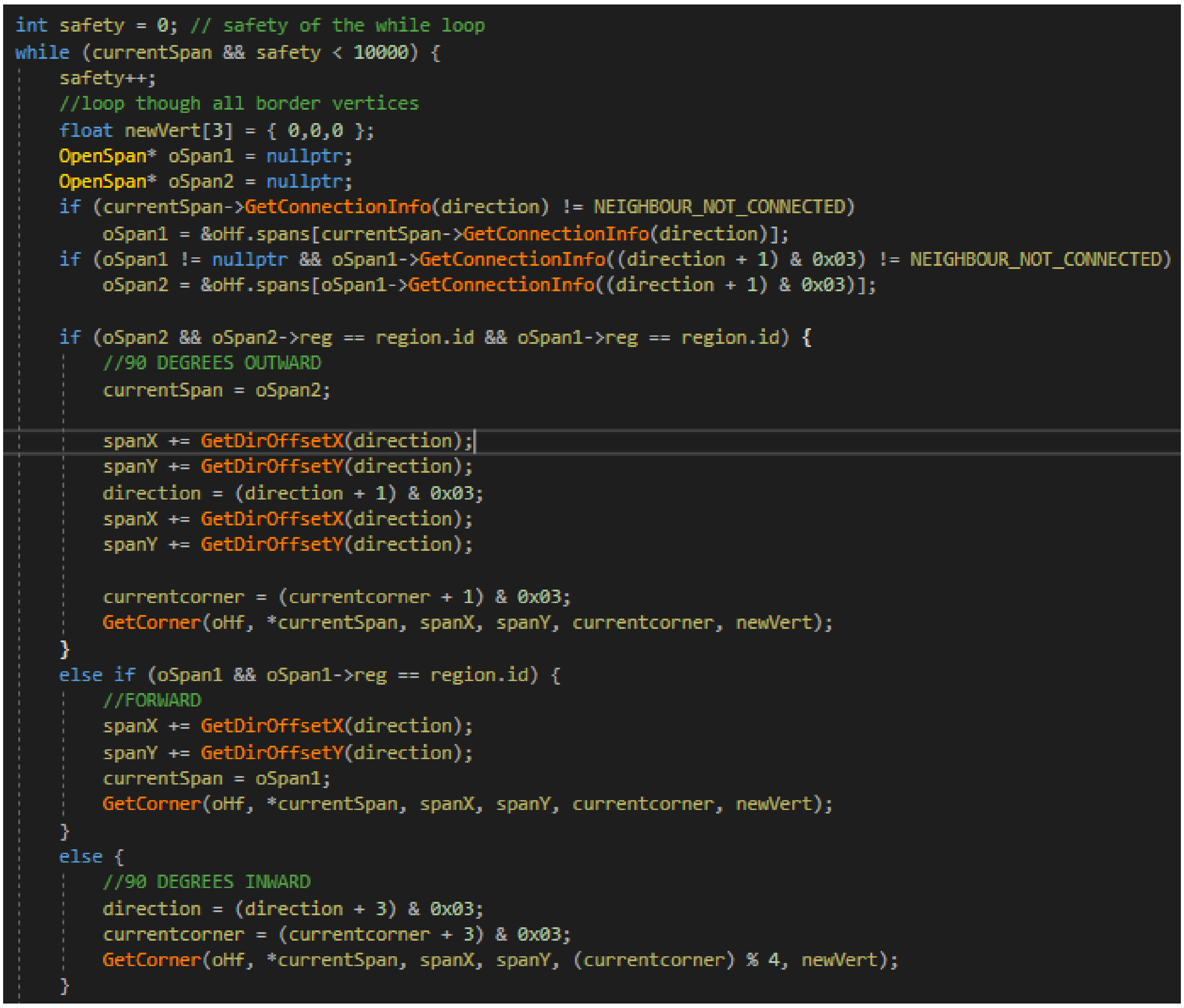


```
bool CalculateContour(OpenHeightField& oHf, Region& region, Countour& outCountour) {
    std::vector<float> finalVerts;  //final vertices
    std::vector<float> verts;       //vertices from the start of a portal
    std::vector<float> startVerts;  //first vertices until a new portal
    float startVert[3];             //starting vertex

    int currentcorner = 0;          //corner of the span it is walking on
    int direction = 0;              //direction of walking
    int spanX = 0;                  //x position of span
    int spanY = 0;                  //y position of span
    int otherRegion = 0;            //region on other side of edge
    int startRegion = 0;            //region on other side of edge of startVerts array
    int startDir = 0;

    //find first span
    OpenSpan* currentSpan = FindEdgeSpan(oHf, region, spanX, spanY, startDir);
    if (currentSpan == nullptr) return false;
    //Set data for the loop
    float v[3] = { 0,0,0 };
    GetCorner(oHf, *currentSpan, spanX, spanY, startDir, startVert);
    GetCorner(oHf, *currentSpan, spanX, spanY, (startDir + 3) & 0x03, v);
    PushVertexInArray(verts, startVert);
    PushVertexInArray(verts, v);
    direction = (startDir + 3) & 0x03;
    currentcorner = direction;
    if (currentSpan->GetConnectionInfo(direction + 1) != NEIGHBOUR_NOT_CONNECTED) {
        otherRegion = oHf.spans[currentSpan->GetConnectionInfo(direction + 1)].reg;
    }
    else otherRegion = 0;
    startRegion = otherRegion;
```

Code Snippet 14

In Code Snippet 15 the loop starts, it first calculates span 1 and 2 as described before and checks if the walking goes outward, forward or inward.

The new span location, the direction, currentcorner and newVert are saved.

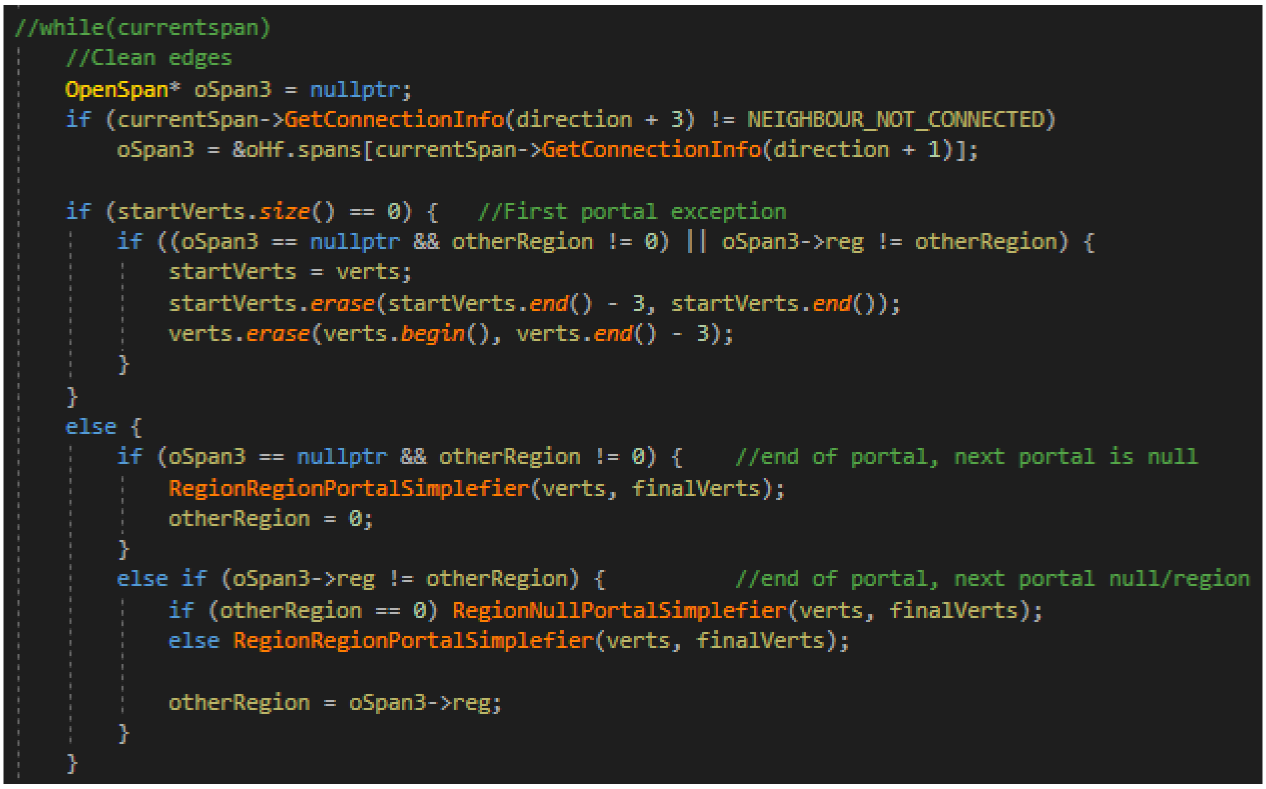


```
int safety = 0; // safety of the while loop
while (currentSpan && safety < 10000) {
    safety++;
    //loop though all border vertices
    float newVert[3] = { 0,0,0 };
    OpenSpan* oSpan1 = nullptr;
    OpenSpan* oSpan2 = nullptr;
    if (currentSpan->GetConnectionInfo(direction) != NEIGHBOUR_NOT_CONNECTED)
        oSpan1 = &oHf.spans[currentSpan->GetConnectionInfo(direction)];
    if (oSpan1 != nullptr && oSpan1->GetConnectionInfo((direction + 1) & 0x03) != NEIGHBOUR_NOT_CONNECTED)
        oSpan2 = &oHf.spans[oSpan1->GetConnectionInfo((direction + 1) & 0x03)];

    if (oSpan2 && oSpan2->reg == region.id && oSpan1->reg == region.id) {
        //90 DEGREES OUTWARD
        currentSpan = oSpan2;

        spanX += GetDirOffsetX(direction);
        spanY += GetDirOffsetY(direction);
        direction = (direction + 1) & 0x03;
        spanX += GetDirOffsetX(direction);
        spanY += GetDirOffsetY(direction);

        currentcorner = (currentcorner + 1) & 0x03;
        GetCorner(oHf, *currentSpan, spanX, spanY, currentcorner, newVert);
    }
    else if (oSpan1 && oSpan1->reg == region.id) {
        //FORWARD
        spanX += GetDirOffsetX(direction);
        spanY += GetDirOffsetY(direction);
        currentSpan = oSpan1;
        GetCorner(oHf, *currentSpan, spanX, spanY, currentcorner, newVert);
    }
    else {
        //90 DEGREES INWARD
        direction = (direction + 3) & 0x03;
        currentcorner = (currentcorner + 3) & 0x03;
        GetCorner(oHf, *currentSpan, spanX, spanY, (currentcorner) % 4, newVert);
    }
```

Code Snippet 15

Code Snippet 16 is still in the while loop. Every edge we walk it walks it checks if the other side of the edge changes region. oSpan3 is the span on the other side of the edge. If the oSpan3 is different from otherRegion then it is the end of the portal and it will clean that portal depending on the type of portal. The cleaned edge is saved in the finalVerts array.

There is an exception which is the first portal. The algorithm starts on a random edge span which can be in the middle of a portal. The first portal will be saved separately and merged with the last portal if they are the same type and share the same region.



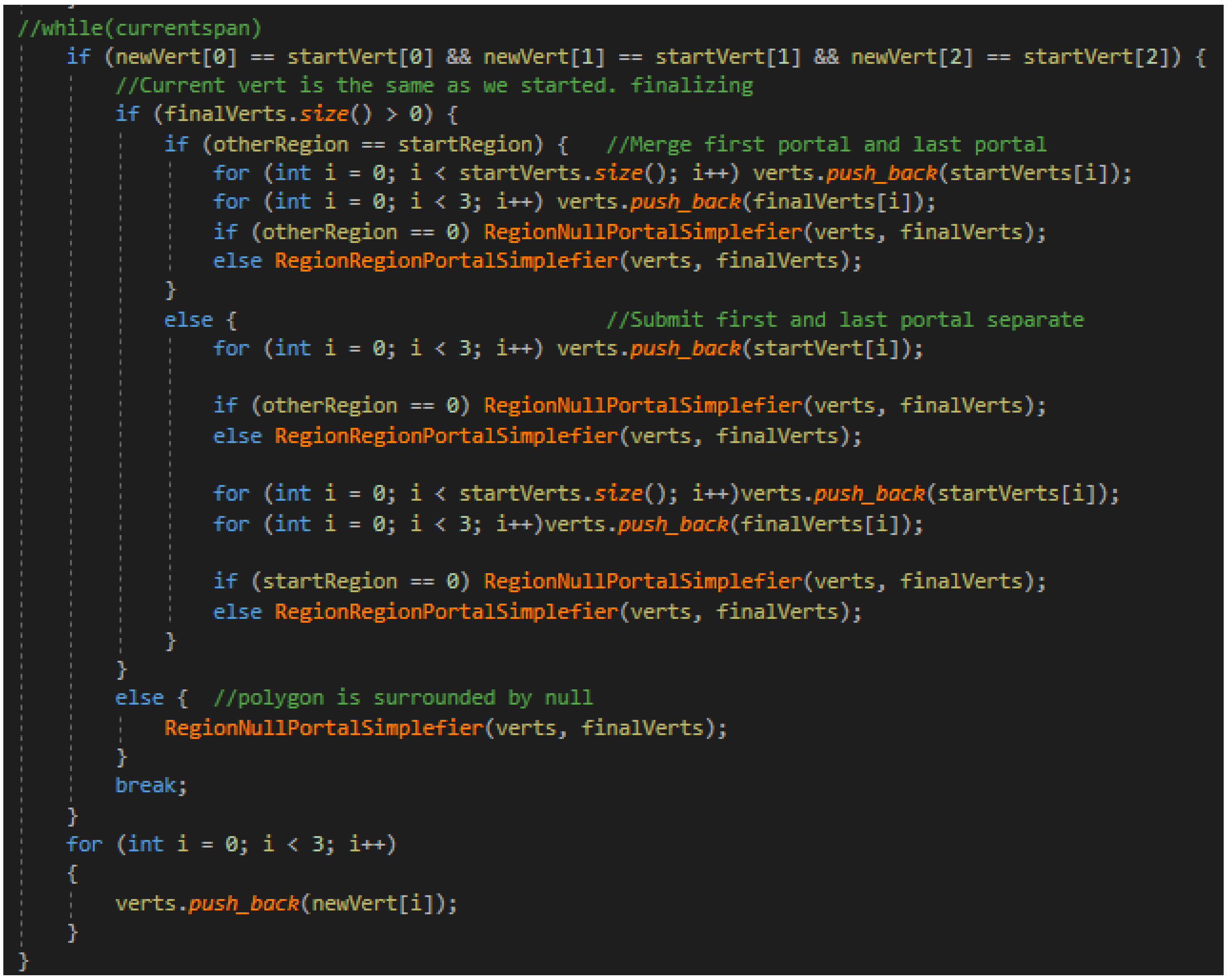
```
//while(currentspan)
    //Clean edges
    OpenSpan* oSpan3 = nullptr;
    if (currentSpan->GetConnectionInfo(direction + 3) != NEIGHBOUR_NOT_CONNECTED)
        oSpan3 = &oHf.spans[currentSpan->GetConnectionInfo(direction + 1)];

    if (startVerts.size() == 0) {   //First portal exception
        if ((oSpan3 == nullptr && otherRegion != 0) || oSpan3->reg != otherRegion) {
            startVerts = verts;
            startVerts.erase(startVerts.end() - 3, startVerts.end());
            verts.erase(verts.begin(), verts.end() - 3);
        }
    }
    else {
        if (oSpan3 == nullptr && otherRegion != 0) {    //end of portal, next portal is null
            RegionRegionPortalSimplefier(verts, finalVerts);
            otherRegion = 0;
        }
        else if (oSpan3->reg != otherRegion) {          //end of portal, next portal null/region
            if (otherRegion == 0) RegionNullPortalSimplefier(verts, finalVerts);
            else RegionRegionPortalSimplefier(verts, finalVerts);

            otherRegion = oSpan3->reg;
        }
    }
```

Code Snippet 16

Code Snippet 17 is still in the while loop. It checks if the current vertex is the same as the starting vertex. If that is true then it means we have done a full loop though the polygon. First thing it checks is if it already has submitted portals in finalVerts. If not then the polygon is completely surrounded by a null region and the whole polygon needs to be cleaned as a region-null portal.

If it has submitted portals is going to examine the first portal and last portal. If they are from the same region it going to merge and undergo a clean. If they a different from region they get submitted separately.

The while loop will break if the current vertex is the same as the start vertex. If not then the newVert which was calculated by walking is submitted to the verts array.



```
//while(currentspan)
    if (newVert[0] == startVert[0] && newVert[1] == startVert[1] && newVert[2] == startVert[2]) {
        //Current vert is the same as we started. finalizing
        if (finalVerts.size() > 0) {
            if (otherRegion == startRegion) {   //Merge first portal and last portal
                for (int i = 0; i < startVerts.size(); i++) verts.push_back(startVerts[i]);
                for (int i = 0; i < 3; i++) verts.push_back(finalVerts[i]);
                if (otherRegion == 0) RegionNullPortalSimplefier(verts, finalVerts);
                else RegionRegionPortalSimplefier(verts, finalVerts);
            }
            else {                              //Submit first and last portal separate
                for (int i = 0; i < 3; i++) verts.push_back(startVert[i]);

                if (otherRegion == 0) RegionNullPortalSimplefier(verts, finalVerts);
                else RegionRegionPortalSimplefier(verts, finalVerts);

                for (int i = 0; i < startVerts.size(); i++)verts.push_back(startVerts[i]);
                for (int i = 0; i < 3; i++)verts.push_back(finalVerts[i]);

                if (startRegion == 0) RegionNullPortalSimplefier(verts, finalVerts);
                else RegionRegionPortalSimplefier(verts, finalVerts);
            }
        }
        else {  //polygon is surrounded by null
            RegionNullPortalSimplefier(verts, finalVerts);
        }
        break;
    }
    for (int i = 0; i < 3; i++)
    {
        verts.push_back(newVert[i]);
    }
}
```

Code Snippet 17

The finalVerts array contains the final cleaned contour data and is saved into the outContour.

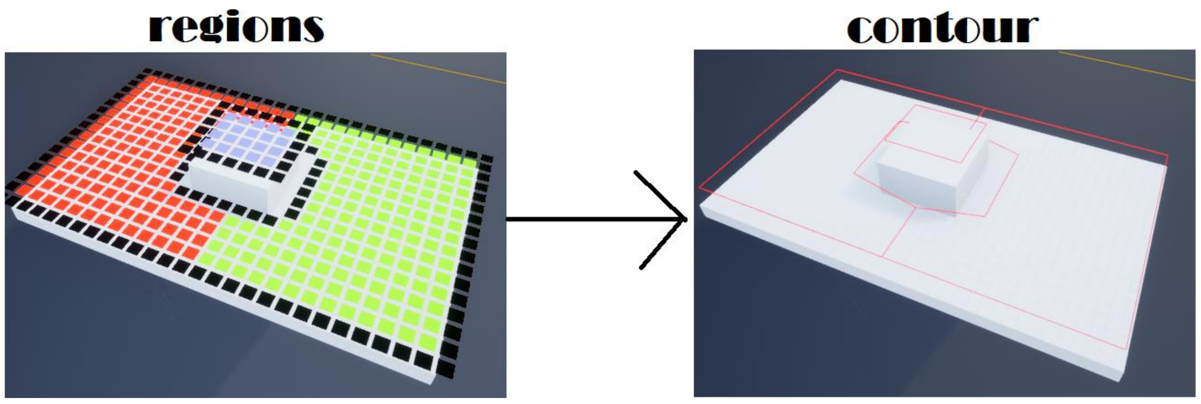
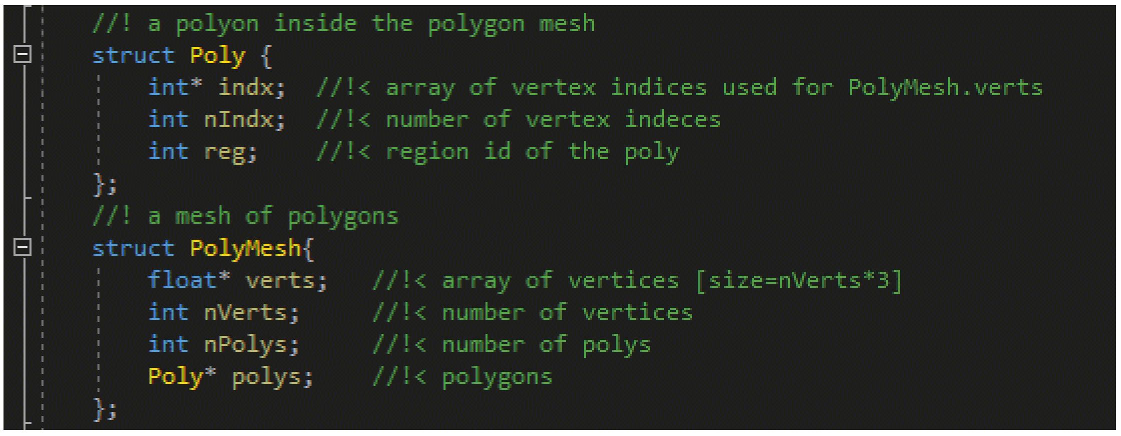


Figure 24.Contour Generation

Polygon Mesh

The contour mesh is now needs to be compressed into a polygon mesh. Vertices are not saved in a central location so there are duplicates when vertices in 2 separate regions have the same position.

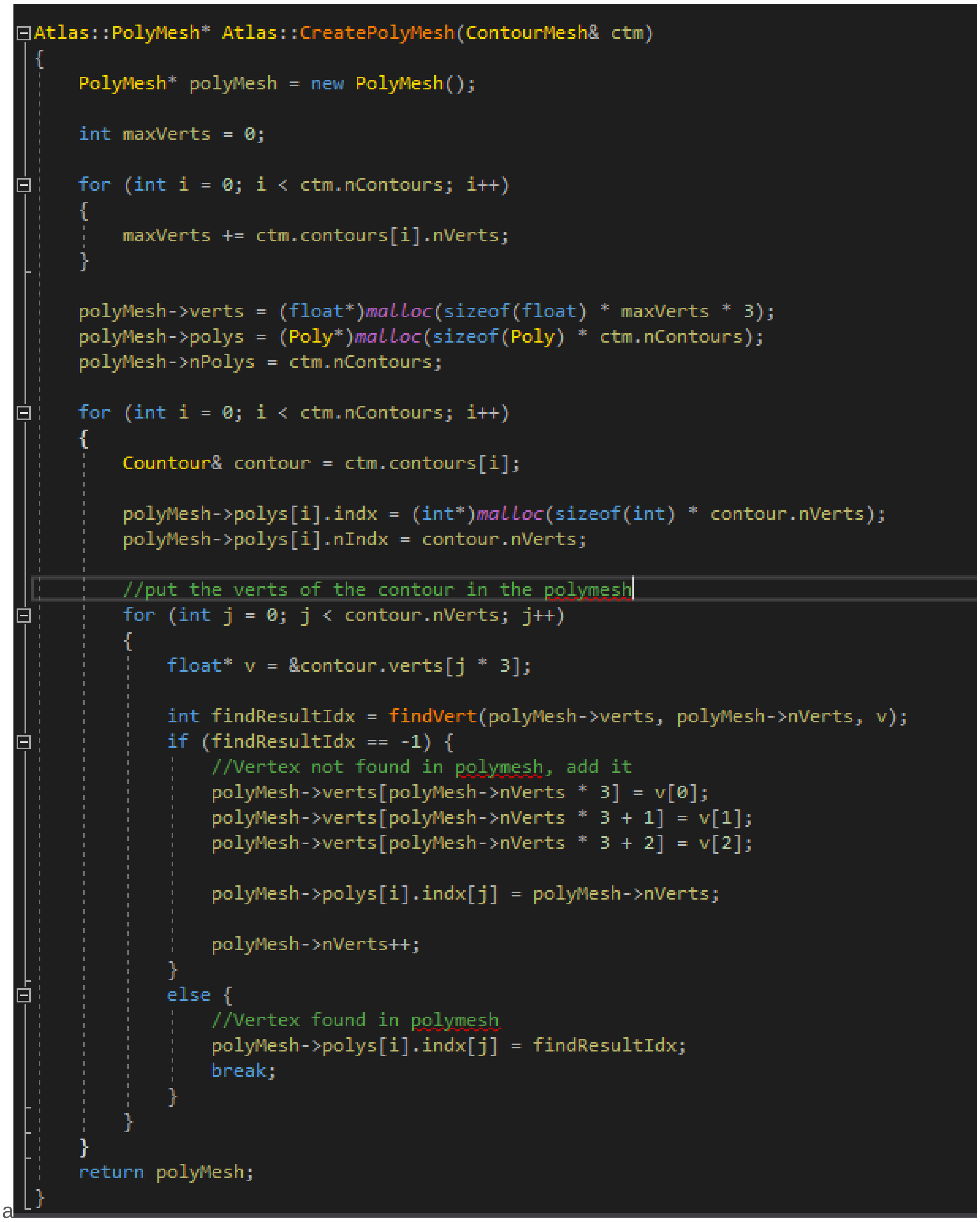
Here we see the data structure of the PolyMesh which consists of an array of vertices and an array of polygons. A polygon has an array of indexes of vertices in the polymesh.



```
//! a polyon inside the polygon mesh
struct Poly {
    int* indx;  //!< array of vertex indices used for PolyMesh.verts
    int nIndx;  //!< number of vertex indeces
    int reg;    //!< region id of the poly
};
//! a mesh of polygons
struct PolyMesh{
    float* verts;   //!< array of vertices [size=nVerts*3]
    int nVerts;     //!< number of vertices
    int nPolys;     //!< number of polys
    Poly* polys;    //!< polygons
};
```

Code Snippet 18

Code Snippet 19 loops though all the contours in the contour mesh and copies over the vertices to the poly mesh if they are not already in there.



```cpp
Atlas::PolyMesh* Atlas::CreatePolyMesh(ContourMesh& ctm)
{
    PolyMesh* polyMesh = new PolyMesh();

    int maxVerts = 0;

    for (int i = 0; i < ctm.nContours; i++)
    {
        maxVerts += ctm.contours[i].nVerts;
    }

    polyMesh->verts = (float*)malloc(sizeof(float) * maxVerts * 3);
    polyMesh->polys = (Poly*)malloc(sizeof(Poly) * ctm.nContours);
    polyMesh->nPolys = ctm.nContours;

    for (int i = 0; i < ctm.nContours; i++)
    {
        Countour& contour = ctm.contours[i];

        polyMesh->polys[i].indx = (int*)malloc(sizeof(int) * contour.nVerts);
        polyMesh->polys[i].nIndx = contour.nVerts;

        //put the verts of the contour in the polymesh
        for (int j = 0; j < contour.nVerts; j++)
        {
            float* v = &contour.verts[j * 3];

            int findResultIdx = findVert(polyMesh->verts, polyMesh->nVerts, v);
            if (findResultIdx == -1) {
                //Vertex not found in polymesh, add it
                polyMesh->verts[polyMesh->nVerts * 3] = v[0];
                polyMesh->verts[polyMesh->nVerts * 3 + 1] = v[1];
                polyMesh->verts[polyMesh->nVerts * 3 + 2] = v[2];

                polyMesh->polys[i].indx[j] = polyMesh->nVerts;

                polyMesh->nVerts++;
            }
            else {
                //Vertex found in polymesh
                polyMesh->polys[i].indx[j] = findResultIdx;
                break;
            }
        }
    }
    return polyMesh;
}
```

a

Code Snippet 19

The polygon mesh looks exactly the same as the contour mesh when drawn but the polygons are now connected

Ear Clipping

It is already possible to use a pathfinder to find a path between points on the polygon mesh but it is cheaper to find path on a triangle mesh. The polygon mesh has to be triangulated. I first used an ear clipping algorithm. A polygon with 4 or more vertices always has at least 2 ears. An ear is a vertex which is non-reflex, reflex meaning the angle of the vertex of 180 degrees or above. This resulted in some unexpected results, a triangle mesh was generated but it wasn't the most optimal mesh. It had weird edges which probably would result in a higher error in the pathfinder. (Figure 25 and Figure 26)

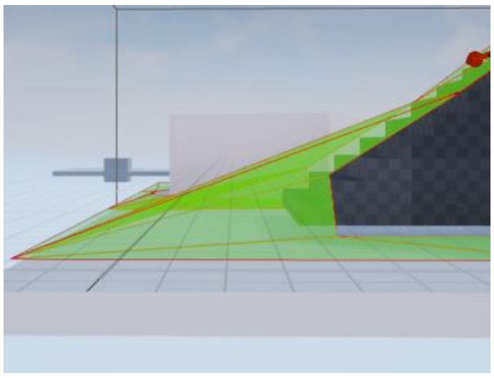


Figure 26. Triangulation error side

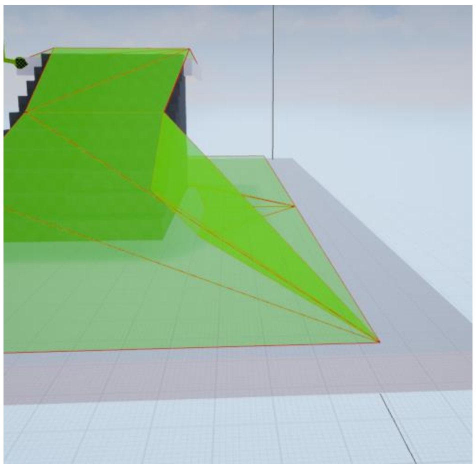


Figure 25. Triangulation error front

Min Area

Ear Clipping is an easy to implement algorithm but generates a low quality navigation mesh. Another algorithm always guarantees that the mesh has the minimal surface area. It uses the following expression:

$$W(D) = \min_{t \in T_D}(w(t) + W(D_1) + W(D_2))$$

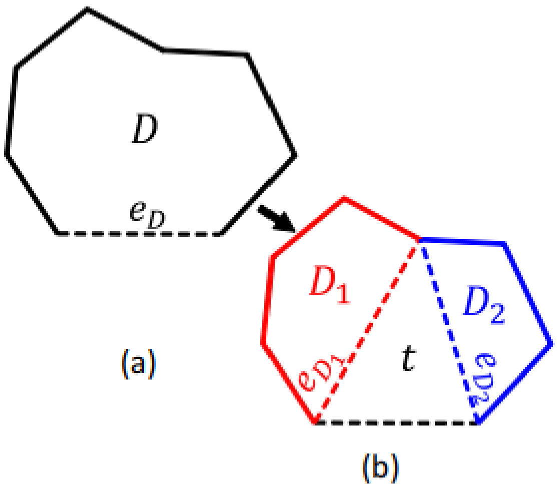


Figure 27

The polygon D has 2 vertices who are connected which form the access edge. Any other vertices in D can join the access vertices to form triangle t. in D there are 6 possible ways to form t.

The area of D is the area of t + the area of D1 + the area of D2. This is calculated for all t's and the one with the smallest area is the optimal mesh. D1 and D2 are calculated with the same equation so this is recursive.

This gives a really good mesh but is quite heavy because it generates all possible ways to form a triangle mesh out of the polygon mesh

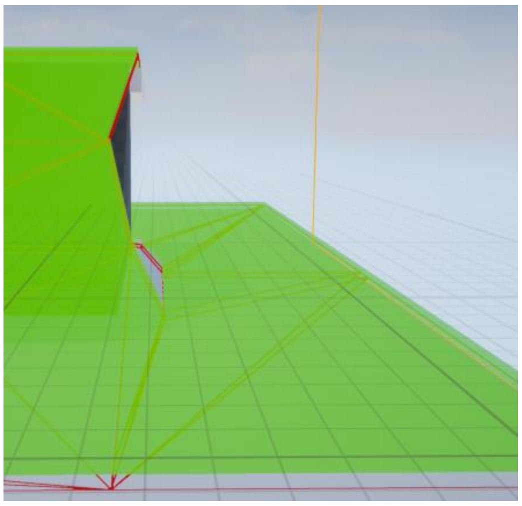
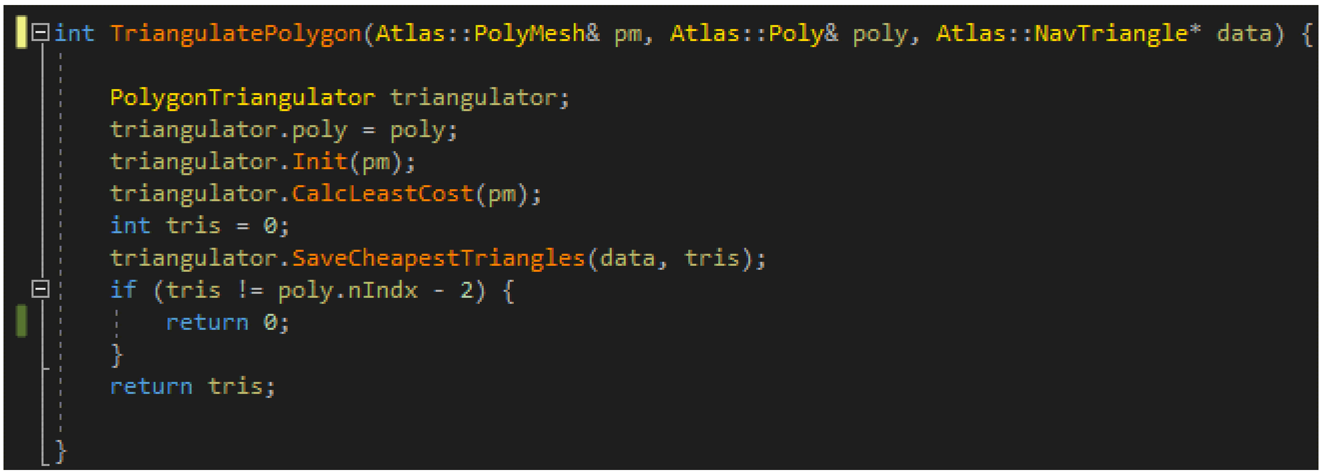


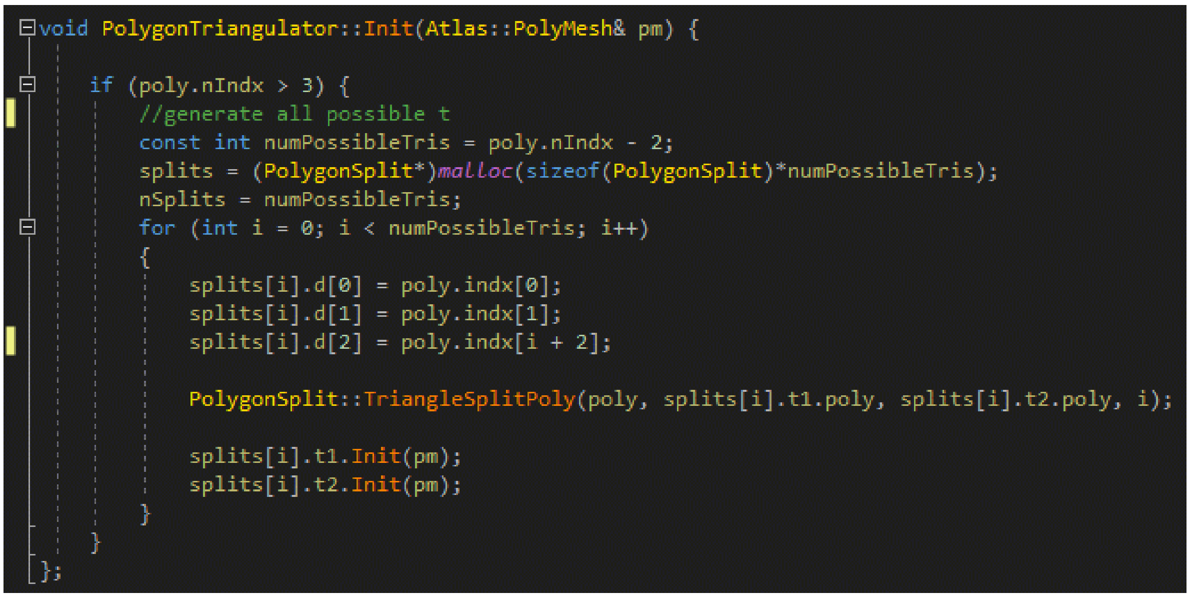
Figure 28



```
int TriangulatePolygon(Atlas::PolyMesh& pm, Atlas::Poly& poly, Atlas::NavTriangle* data) {

    PolygonTriangulator triangulator;
    triangulator.poly = poly;
    triangulator.Init(pm);
    triangulator.CalcLeastCost(pm);
    int tris = 0;
    triangulator.SaveCheapestTriangles(data, tris);
    if (tris != poly.nIndx - 2) {
        return 0;
    }
    return tris;

}
```

Code Snippet 20

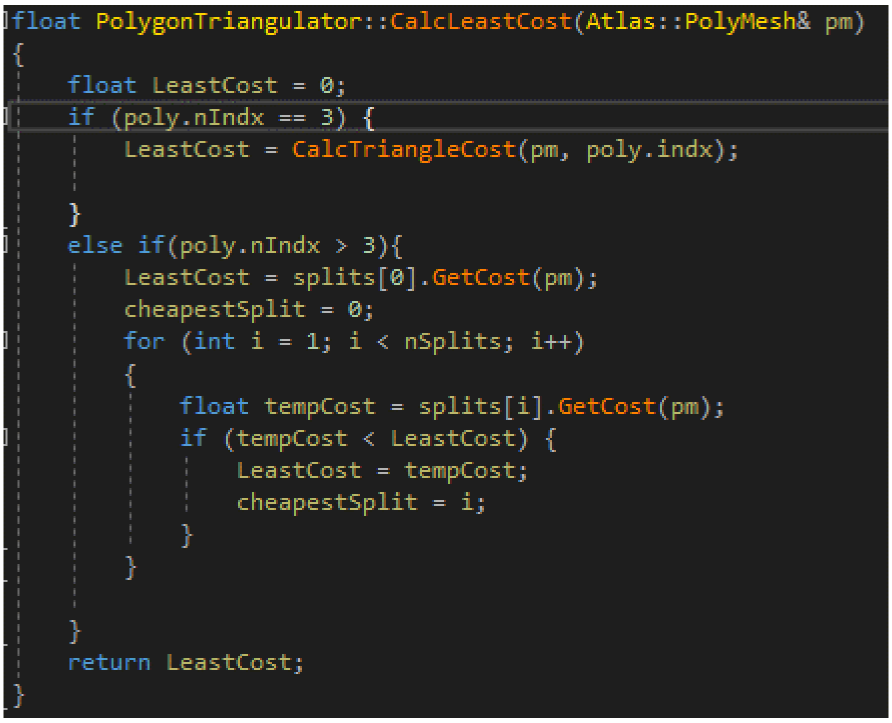
First I set up the triangulator by feeding it the polygon with Init(). Then it calculates the t's with the least costs with CalcleastCost(). After this a tree with all possible meshes is generated and the cheapestmesh is saved with SaveCheapeastTriangles().



```
void PolygonTriangulator::Init(Atlas::PolyMesh& pm) {

    if (poly.nIndx > 3) {
        //generate all possible t
        const int numPossibleTris = poly.nIndx - 2;
        splits = (PolygonSplit*)malloc(sizeof(PolygonSplit)*numPossibleTris);
        nSplits = numPossibleTris;
        for (int i = 0; i < numPossibleTris; i++)
        {
            splits[i].d[0] = poly.indx[0];
            splits[i].d[1] = poly.indx[1];
            splits[i].d[2] = poly.indx[i + 2];

            PolygonSplit::TriangleSplitPoly(poly, splits[i].t1.poly, splits[i].t2.poly, i);

            splits[i].t1.Init(pm);
            splits[i].t2.Init(pm);
        }
    }
};
```

Code Snippet 21

In the Init function it generates all possible t's as splits.

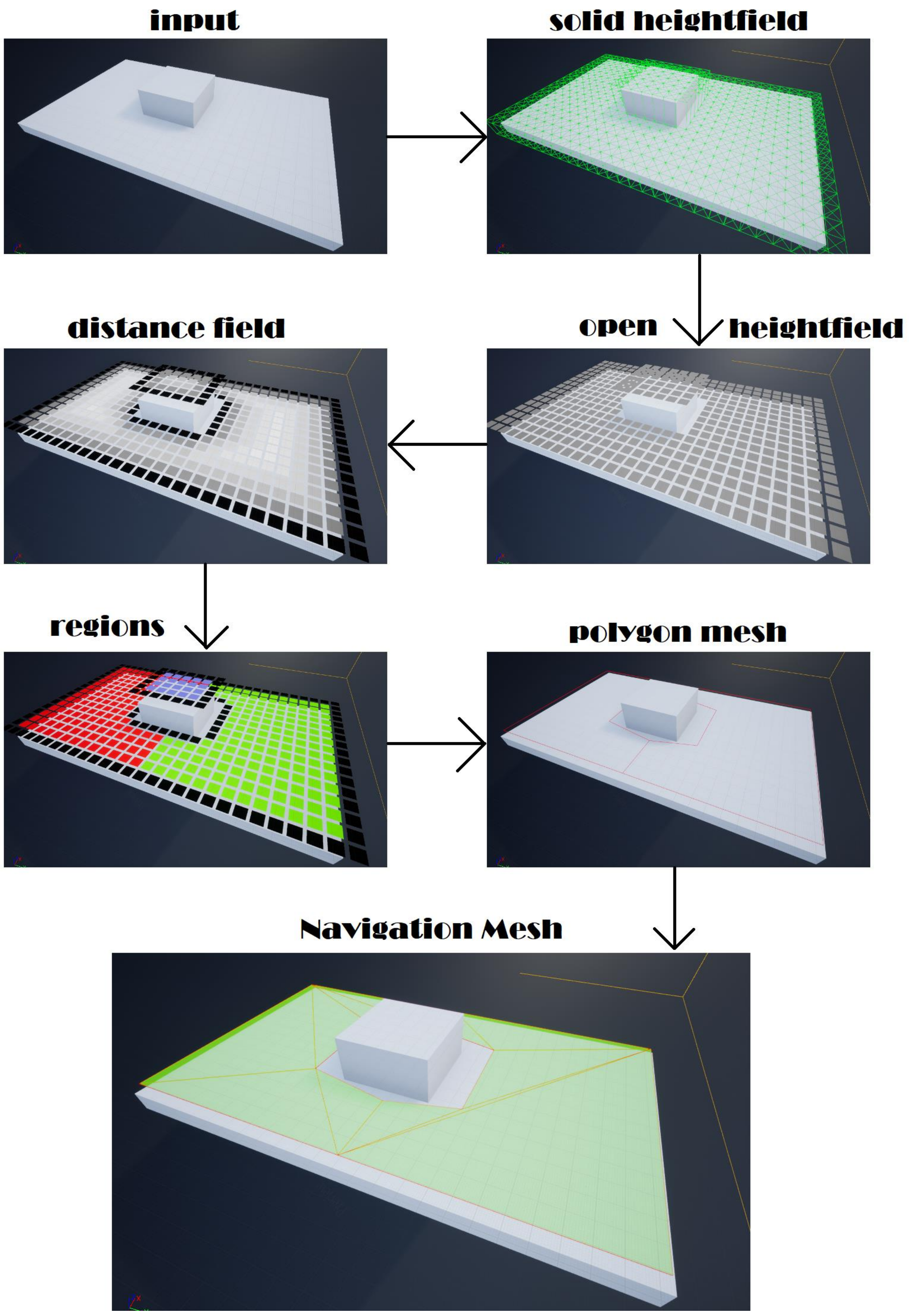


```
float PolygonTriangulator::CalcLeastCost(Atlas::PolyMesh& pm)
{
    float LeastCost = 0;
    if (poly.nIndx == 3) {
        LeastCost = CalcTriangleCost(pm, poly.indx);

    }
    else if(poly.nIndx > 3){
        LeastCost = splits[0].GetCost(pm);
        cheapestSplit = 0;
        for (int i = 1; i < nSplits; i++)
        {
            float tempCost = splits[i].GetCost(pm);
            if (tempCost < LeastCost) {
                LeastCost = tempCost;
                cheapestSplit = i;
            }
        }

    }
    return LeastCost;
}
```

Code Snippet 22

The CalcLeastCost function calculates the cheapest polygon from bottom of the tree to the top and is thus recursively solved. GetCost of a split calculates how big the area is with t+D1+D2.

The Result


input
solid heightfield
open heightfield
distance field
regions
polygon mesh
Navigation Mesh

3 Axis Navigation Mesh

There are multiple ways to implement a 3 axis navigation mesh but because I have created a 2 axis navigation mesh already I will be modifying it to support 3 axis. If a 2 axis Navigation mesh is generated 6 times, 1 time for each axis positive and negative with a supported angle of 45 degrees then the 6 meshes could be merged together to 1 big navigation mesh.

Not every step in the 2 axis navigation mesh generator has to be performed 6 times to generate 6 navigation meshes. Some can be modified.

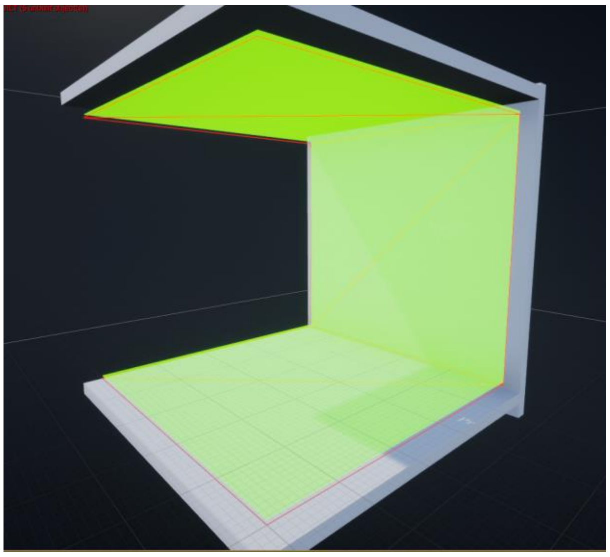
.1 The voxelizing
voxelizing can be done in 1 step because the voxels aren't going the change when changing axis but because it is easier for now I will voxelize once for each axis because there is a need of a top and a bottom on each span.

.2&3 Region generation & contour generation
regions and contours have be generated 6 times for each axis positive and negative.

.4 Polygon Generation
polygons can be generated using all the contour meshes and merged to 1 polygon mesh

.5 Navigation mesh
Is generated from the single polygon mesh

This approach did not work, it only works when the navigation mesh is 90 degrees from each other. Because on slopes the generator will generate multiple navigation meshes on top of each other.



A solution for this is to merge these overlapping meshes or to change a region generation to a 3d region generation which can also change direction.

Another solution is to change the heightfields to voxelfields. The openheightfield has surfaces with are all the same orientation, with voxelfield I can change the orientation and generate a region on the floor and walls(Figure 29).

With this 3d region I can generate the polygon mesh fairly easy which can be converted into a 3 axis navigation mesh.

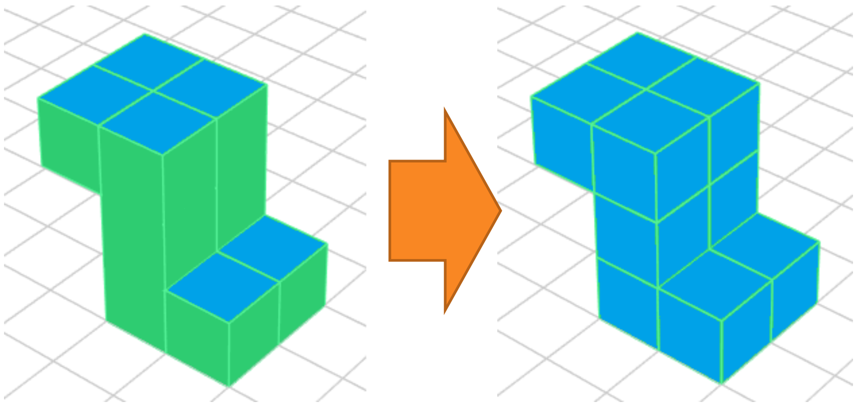


Figure 29

Testing

I didn't had any time to test extensively yet but I have made plans for testing. The navmesh should be tested on the following:

- Usability
- Speed of Generation
- Robustness

Usability

Atlas should be used in a developer environment so it's best to test it that way. There are 2 test, the usability of developers that use Atlas which is already implemented into an Engine and the usability of developers that has to implement Atlas into an Engine.

The first case is testable using the already implemented Atlas into Unreal Engine. I need to ask at least 10 developers which have experience with creating navmeshes in Unreal to use my system. The testers need to fill in a form and I make notes while observing them.

The second case is a bit harder to test since not many people have the time or the engine to implement Atlas. If I don't find a tester I have to implement Atlas myself into another Engine.

Speed of Generation

Every step of the generation can be measured in the number of operations. This should be tested multiple times on different maps. A simple geometry and a detailed geometry which also both have different sizes. This way I also test scalability. This data should be used in further optimizations.

Robustness

Robustness can be described as the program being to cope with errors during execution and is hard to test. One of the methods to test this is fuzz testing, where invalidated data is used, the outcome would be an invalidated nav mesh but the program should not crash. Another test is to input a large amount of data to check tests when the program runs out of memory.

I tested ona small case. 2 developers implemented atlas in a dummy environment because we didn't have a custom game engine available. It took them about 1 hour to implement Atlas using the doxygen documents. I didn't provide any help in the implementation, I only observed. After implementation they used Unreal Engine to generate a navigation mesh. They said the navigation mesh looks nice but they had some crashes which means Atlas isn't very robust yet.

Reflection

This project was very interesting with something new with every step. To create the navmesh I had to implement 7 systems as described in the plan at the start of this document. With the start of each step I had to research, implement and bug fix it. The hardest steps were:

- Voxelizing
- Contour generation
- Triangulation

Voxelization was an easy algorithm but because of the large amount of data hard to debug. I was also working with the engine which had to provide the collision data which was hard to obtain and which had to be transformed into data that Atlas would understand. Because I didn't know how Unreal Engine delivers the collision data I spend an a lot of time dissecting Recast which is implementing in Unreal Engine. I was stubborn because I wanted to it myself while I also could've asked someone else who has that knowledge. After I completed the project I joined a Unreal Dev Group which also contains several people who work for Epic.

Contour generation was also a simple algorithm but it was a big algorithm around 400 lines of code. This also made it hard to debug and the more code the more chance on a bug. If I would write this algorithm again it would probably not be this large and less prone to errors. Maybe in the future when I have more programming experience I can see these problems before I start writing an algorithm.

Triangulation was a hard algorithm and hard to debug. I had a hard time choosing a algorithm while researching triangulation. I never done 3d triangulation before so I wanted to start with an easy algorithm which probably wasn't the optimal mesh. I choose to adapt the ear clipping algorithm which is used in 2d triangulation. This adaptation worked and the mesh was triangulated but it was far from the optimal mesh, it had many errors and had to be replaced. Unfortunately I didn't had enough time to replace it.

I did underestimated almost every step of the implantation how much time it would take. I already knew that it very step would take quite some time so I planned extra time for it but I still went over it. This is due to the fact that I had never created something this big before and most of the content in this project was new for me.

The 3 axis generator does work but only for 90 degree angles, I have to improve the algorithm in the future to make it fully work. Unfortunate I don't had any time to complete the pathfinder, this means the characters in-game can't use the navigation mesh yet.

I am quite happy with the results of the project. I wanted to do more like optimizing the code and comparing it with Recast but I got at least Atlas working. My goal was to learn about navigation meshes and create a nice piece for my portfolio, both are a success. I learned a lot from this project as can be seen in this document and I created a nice video which shows all the steps in the generation of the navigation mesh. Next to learning about generation a navigation mesh I also got better at programming because I programmed a lot and I know now that certain things work and don’t work.