# Customizable Pathfinding Unreal Engine 5 plugin

(Wtyczka do Unreal Engine 5 - Customizable Pathfinding)

Dominik Trautman

Praca inżynierska

**Promotor:** dr Łukasz Piwowar

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

1 lutego 2023

**Abstract**

Pathfinding is a process of finding an ordered list of points in a virtual space, such that after connecting all the points in order, there is a continuous path from the first to the last point. This process is extensively used in video games where autonomous agents have to traverse around the game world. The most common algorithm for this purpose is A*. Pathfinding is often limited to two dimensional space. However, there are cases where that's not enough. For example, swimming under water or flying.

This thesis is about my implementation of 3D pathfinding as an *Unreal Engine 5* plugin. By default, the engine only supports ground pathfinding on the Z-Axis.

My graph is based on an octree structure which ensures decent space management and allows for total parallelism, as each tree can be generated independently.

One of the main features separating my solution from other plugins is the possibility to regenerate parts of the graph around selected objects during runtime, allowing for dynamically changed environments.

Each 'Find Path' request is by default executed on a separate thread, leaving the main game thread unencumbered.
The algorithm used for pathfinding is a slightly modified A*, with post-process node reduction.

Pathfinding (wyszukiwanie ścieżki) odnosi się do procesu znajdowania optymalnej ścieżki dla postaci lub obiektu poruszającego się w grze. Często jest używany w grach, w których występują postacie sterowane przez sztuczną inteligencję. Pozwala im na wiarygodne przemieszczanie się po otoczeniu. Algorytmy wyszukiwania ścieżki biorą pod uwagę takie czynniki jak: ukształtowanie terenu, przeszkody, oraz zdolności postaci do wspinania się, skakania, latania czy pływania. Najpopularniejszy algorytm wykorzystywany w tym celu to A*. W silnikach do gier zazwyczaj jest zaimplementowany jedynie w dwóch wymiarach (2D). Istnieją jednak przypadki, gdzie potrzebna jest ścieżka w trójwymiarowej przestrzeni, na przykład pływanie pod wodą lub latanie.

W pracy opisuję własną wtyczkę do silnika Unreal Engine 5, która implementuje rozwiązanie tego problemu, także w 3D. Domyślnie, UE5 nie wspiera takiej możliwości.

Strukturą użytą do wyszukiwania ścieżki jest graf oparty o octree (drzewo ósemkowe). Umożliwia ona stworzenie w miarę wydajnej reprezentacji trójwymiarowej przestrzeni, oraz pozwala na zrównoleglenie procesu generacji grafu na wiele wątków.

Dodatkową funkcjonalnością, odróżniającą moje rozwiązanie od innych, pełniących podobną funkcję, jest możliwość regeneracji grafu w czasie rzeczywistym. Pozwala to na zastosowanie w dynamicznie zmieniających się przestrzeniach.

Domyślnie, każde polecenie wyszukania ścieżki jest wykonywane na osobnym wątku. Dzięki temu główny wątek, na którym wykonuje się większość gry, jest całkowicie nieobciążony. Algorytm użyty do szukania ścieżki to lekko zmodyfikowana wersja klasycznego A*. Końcowym etapem wykonywania algorytmu jest optymalizacja znalezionej ścieżki (wygładzanie i usuwanie zbędnych punktów).

# Contents

# Chapter 1

# Introduction

## 1.1 Problem description

Unreal Engine is a popular software that serves as a code base and graphics editor for developing video games. Unreal Engine features many tools, with one of them being an existing pathfinding solution.
However, the in-built implementation is limited in many ways:

- works only on the ground, so it cannot be used for flying, jumping, or wall climbing
- accessible by overwriting the in-built *Pawn* class. It is impossible to get a path consisting of points without modifying the engine itself
- not usable for 2D games, unless they use a top-down view and all AIs move on the ground
- cannot be used for pathfinding on a discrete grid
- the graph is not generated in runtime, so it is not suitable for dynamically generated levels

Fortunately, Unreal Engine allows developers to extend the engine's functionality by plugins.
My plugin, *Customizable Pathfinding*, developed for this thesis, aims to fill the gaps listed above.

## 1.2 Main ideas

If the plugin is supposed to work in 3D space, it needs to describe empty space efficiently. The graph for pathfinding is represented by an array of *octrees* with variable depth. This approach allows for heavy parallelization of tasks, as the octrees can be generated independently of one another.
Maximum octree depth is exposed to the user. It enables a normal grid setup, since with *maximum depth* set to 0, our graph will just be an array of voxels.

Two core functions of the plugin are overridable. One determines the priority of a node while searching for a path, the other one is invoked during generation. By overriding the latter, user can implement conditions to tell whether a node in graph is free or occupied, and optionally generate additional data, accessible during pathfinding. This, combined with customizable generation area and octree depth, introduces great flexibility.

## 1.3 Other implementations

Pathfinding in 3D space is a relatively new topic. The need for it has greatly increased ever since drones became widely popular. However, most companies keep their solutions private. Another obvious application is flying entities in video games. Most other implementations I found share a lot of similarities. The main difference is space representation and the pathfinding algorithm used. Octree seems to be the most common choice for space representation. There are two common ways of traversing it:

- **Points in centers of cubes** - This is perhaps the simpler approach, and the one I have chosen. We search for a path between centers of different-sized cubes in the graph.

- **Cube corners** - In this version, edges in the graph are represented by the actual edges of each cube in the octree. The biggest downside of this approach is that in order for an agent to be able to traverse along the edge, all cubes sharing the edge must be free.

Another simplistic approach is to manually create *waypoints*, and only search for paths between them. This only works for completely static environments, and requires additional work from the user. The only positive is that it is incredibly fast, and in some cases where there is no need for a precise dynamic solution, this one is the most optimal.

As for alternatives available for a potential user of this plugin, there is the in-built engine solution. In section **1.1**, I have listed reasons as to why it is insufficient in

many scenarios. I have also found one other plugin worthy of mentioning, specifically designed for 3D pathfinding - ***Flying Navigation System*** [**13**]. It has been released during the development of my plugin. It also uses an octree structure and allows for the selection of a pathfinding method. However, it does not seem to have many use cases apart from 3D pathfinding in continuous space and it only works in fully static environments.

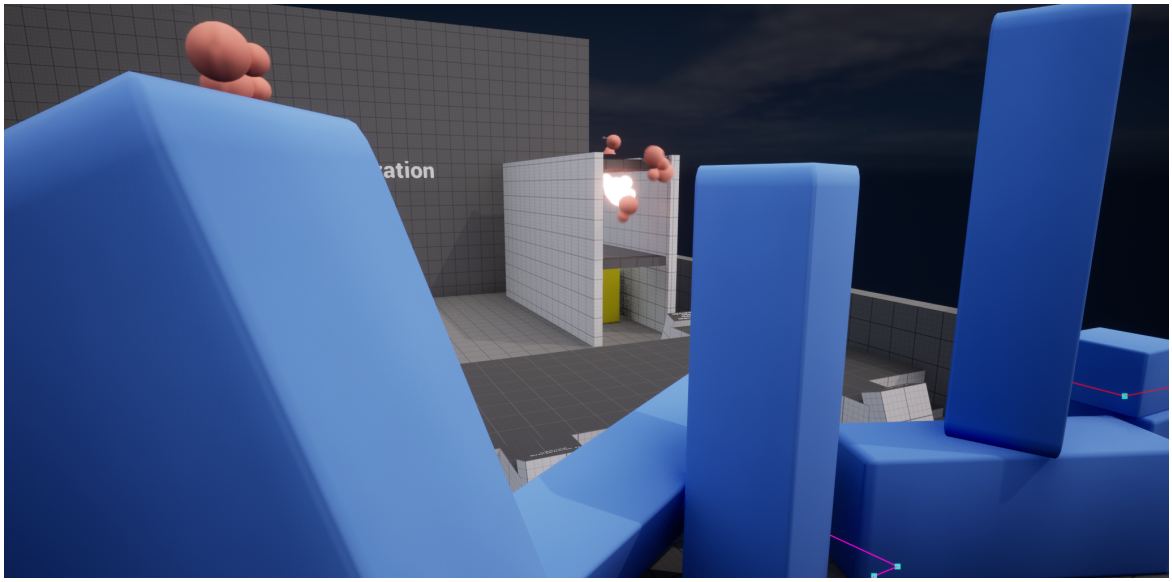

*Fig. 1.1* The first example level in Unreal Engine 5. On the screenshot we can see blue dynamic obstacles, and pink flying pawns navigating to the bright target where they end their journey. The blue obstacles can be moved, and the graph regenerates around them, allowing the pawns to avoid them.

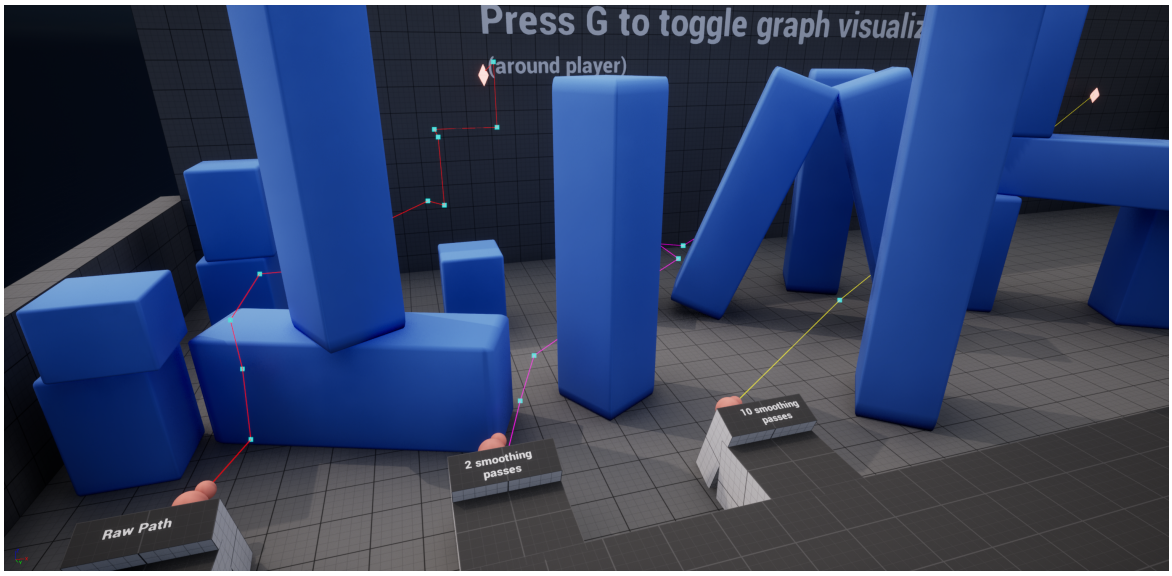


*Fig. 1.2* The same level, from another perspective. We can see FindPath calls generating paths around the dynamic obstacles with different settings.

# Chapter 2

# Implementation details

## 2.1 Octree

### 2.1.1 Memory representation

The most confined *octree* representation for this purpose would be an array of bits. An octree can be either *free* (bit 0) or *occupied* (bit 1). However, each tree's size would need to be static, relative to the maximum octree depth (MaxDepth). This means that even if there is a large empty area, there is data for every subtree in that area. For MaxDepth equals 3, that would be $8^3$ bits. Furthermore, this approach limits the data each tree may carry to just this one piece of information.

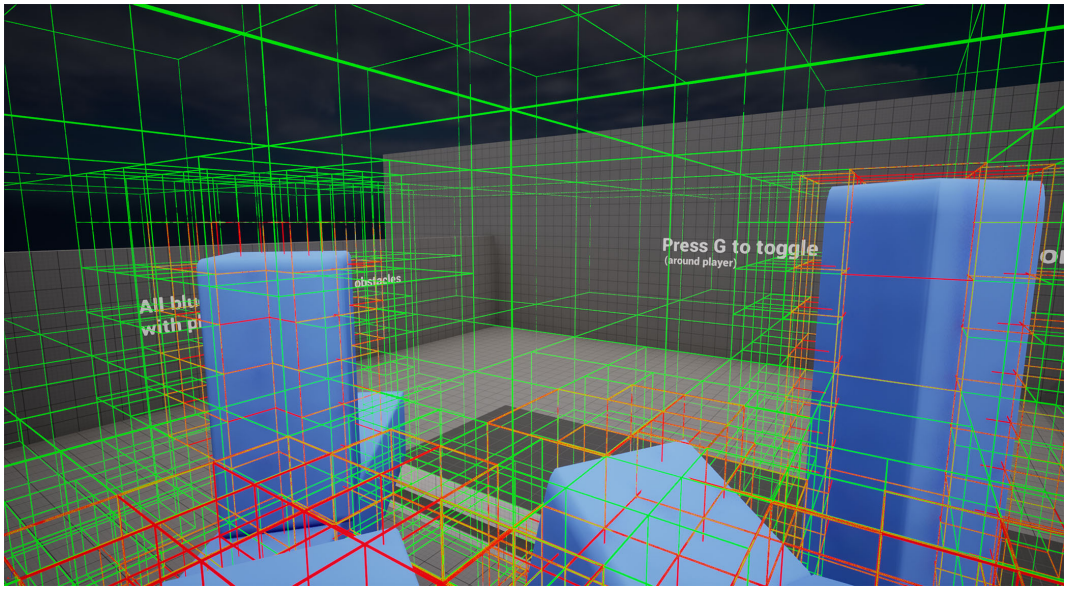


*Fig. 2.1* Visualization of generated octree in one of Unreal Engine 5's example levels

In my project, every *octree* is represented by a pointer to its children and a UserData variable. The UserData variable is a 32-bit integer. By default only the least significant bit is used, leaving 15 bits for any user data. If the pointer to children is a nullptr, the tree is a leaf.

There is room for improvement in my representation. There is no need for a whole 32-bit or 64-bit pointer to children. All children could be stored in a single array with custom memory management. The maximum size of this array could be, for example, $2^{26}$. This would impose a limit on the number of nodes in our graph, but over 67 million nodes should be more than enough for every scenario. Each tree could then be represented by a single 32-bit integer. Assuming that every address points to 8 children, we only need 18 bits for the address to a 26-bit sized array storing all the children:

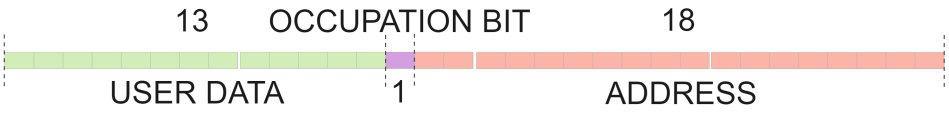


*Fig. 2.2* Visual representation of the optimized integer

### 2.1.2 Generation

The main structure in my program is named *volume*. A volume is a cuboid that may be placed in a level. Its size is set by the user. The volume consists of an array of octrees. Their count depends on the scaled size of the volume, the MaxDepth variable, and the octree cube extent at MaxDepth (from now on called VoxelExtent). Therefore, the number of octree roots in each dimension is calculated at the beginning with the following formula:

$$ceil(\frac{VolumeSizeInThisDimension}{VoxelExtent \cdot 2^{MaxDepth}})$$

One advantage of having multiple separate trees is that they can be generated independently. This allows for pretty much unlimited parallelism. By default, the plugin uses up to $N - 1$ threads for generation, where $N$ is the number of physical cores in the system. There are two stages of generation:

- **Initial generation** - First, a vector populated with empty octrees is created. The number of trees to generate is divided equally between each thread. After all threads finish their work, the graph is marked as initialized.

- **Updating dynamic obstacles** - Each object in the world can be marked as a *DynamicObstacle*. In regular intervals, the volume instance creates a set of octrees to regenerate based on each DynamicObstacle's bounds. Object bounds are provided by the engine. The regeneration task is potentially divided between threads, just like the initial generation. However, more than

one thread is only used if there are many trees to regenerate. At the beginning of each update, finished threads are collected, and the set is initialized with trees added in the previous update.

## 2.2 Pathfinding

There are 3 most common algorithms for efficient pathfinding in 3D space:
**A***, **Theta*** and **Lazy Theta***. *Theta** and *Lazy Theta** are extensions of *A**, specifically designed to find more optimal paths in 3D space. A detailed description of these algorithms can be found here [**1**].

However, I have decided to implement *A**, for two main reasons:

- **Performance** - In Theta versions, visibility line checks need to be done for every considered node that is not a direct neighbor of the currently expanded node. In most scenarios, this decreases the performance by more than half, while producing a path that is barely $10 - 20\%$ more optimal.

- **Precision** - A simple line trace check does not guarantee that the Agent will be able to traverse along traced distance. For that to be the case, the line trace would need to be replaced by a sweep with user's chosen shape and dimensions.

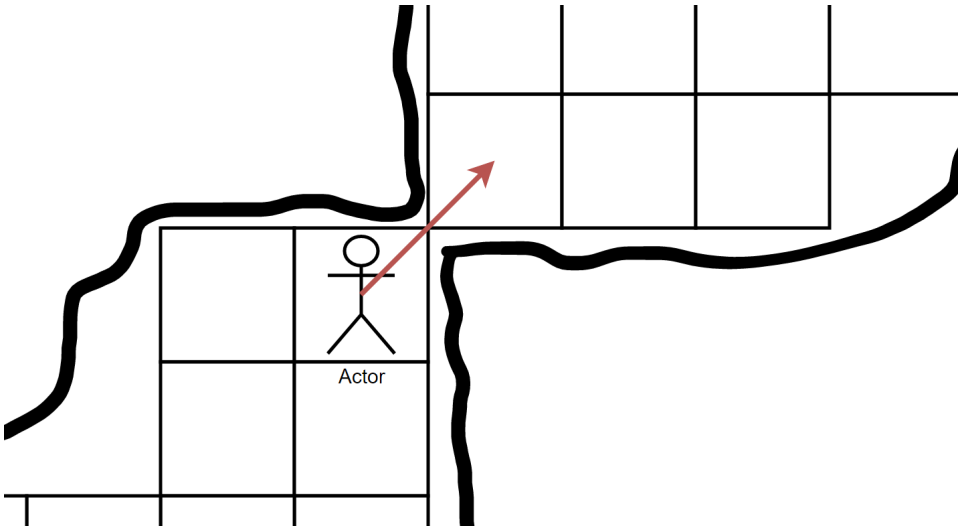


*Fig. 2.3* An example of the issue caused by considering diagonal neighbors

For the same reason as above, my implementation doesn't even consider diagonal nodes. Therefore, assuming that the neighbors are at the same octree depth, a node only has 6 neighbors. The downside of this approach is it potentially produces jagged paths, but that issue is resolved by post-processing.

## 2.3 Synchronization

Volume data can only be accessed while it's not being modified. The only way it can be modified after initial generation is through *Generation Threads*. There are two *atomic integers* per volume instance. One tracks how many generation tasks are running, the other how many pathfinding tasks are running. A pathfinding task can only start once there are no generation tasks running, and vice versa. However, when a generation task is requested, it instantly increments its thread count, so that no new pathfinding tasks may begin while it's waiting for the ongoing ones to finish.

# Chapter 3

# Documentation

## 3.1 Detailed graph representation

A previously mentioned volume is a class named ACPathVolume, that inherits from Unreal Engine's AActor class. Actor is an object that can be placed in the game *World*, so its lifespan is automatically managed. Actor's lifespan is also managed by Unreal's garbage collector. On top of that, inheriting from AActor allows me to attach *components* to it. The box itself, that lets user set the area of generation is represented by UBoxComponent. More about actors and components can be found in Unreal Engine 5 documentation [**2**].

### 3.1.1 Graph Generation

The graph is initialized by calling method ACPathVolume::GenerateGraph(). Here, all the necessary variables are set, including MaxGenerationThreads (unless it was overridden by user).
Initial generation is always performed by MaxGenerationThreads. This method also initializes the ACPathVolume::GenerationUpdate() timer. Timers are another feature of Unreal Engine, which invokes a method after specified delay, optionally in specified time intervals until explicitly stopped.

All generation is performed asynchronously. Threads are created using Unreal Engine's FRunnable and FRunnableThread objects. FCPathAsyncVolumeGenerator inherits from FRunnable. Given an array of indices, its task is to generate (or regenerate) octrees with those indices.

A single octree is generated by one recursive method -

```
bool RefreshTreeRec(CPathOctree* OctreeRef, uint32 Depth, FVector
    TreeLocation);
```

This method calls RecheckOctreeAtDepth() to check if the octree is finished at the

current depth. If it is, it returns true. If it's not, it creates eight children and calls itself on them. If no children returned true, all of them are deleted to save memory. Only in this case false is returned, since then this Octree has no free children and cannot be accessed in any way.

In my base implementation, function

```
virtual bool ACPathVolume::RecheckOctreeAtDepth(CPathOctree* OctreeRef,
    FVector TreeLocation, uint32 Depth)
```

performs a box overlap scan at user selected channel, with dimensions based on Depth and returns true if there was no collision. This method also updates the UserData variable in passed octree - by default it only sets the least significant bit to 1 if there was no collision. All the other bits may be utilized by the user to save some other data during generation. For example, a downward line-trace can be performed to check whether this node is close to the ground and saved on the second bit, or even full distance with 31-bit precision if this is the only required information.

Furthermore, since this is the only method that determines if the node in graph is free or not, by overriding this method it is possible to completely change how the graph is gonna behave. For example, one could check if the node is within some distance from a certain terrain, and only then make it free. This could be used for wall walking or to only allow access to certain areas.

ACPathVolume holds a *set* of tracked dynamic obstacles.
Since the UCPathDynamicObstacle is an actor component, every dynamic obstacle has to be an AActor.
All obstacles overlapping the ACPathVolume are added at BeginPlay() by default. They are also added upon leaving and entering the volume as the game goes on. To guarantee that each obstacle is only tracked once, the container is implemented using std::set.

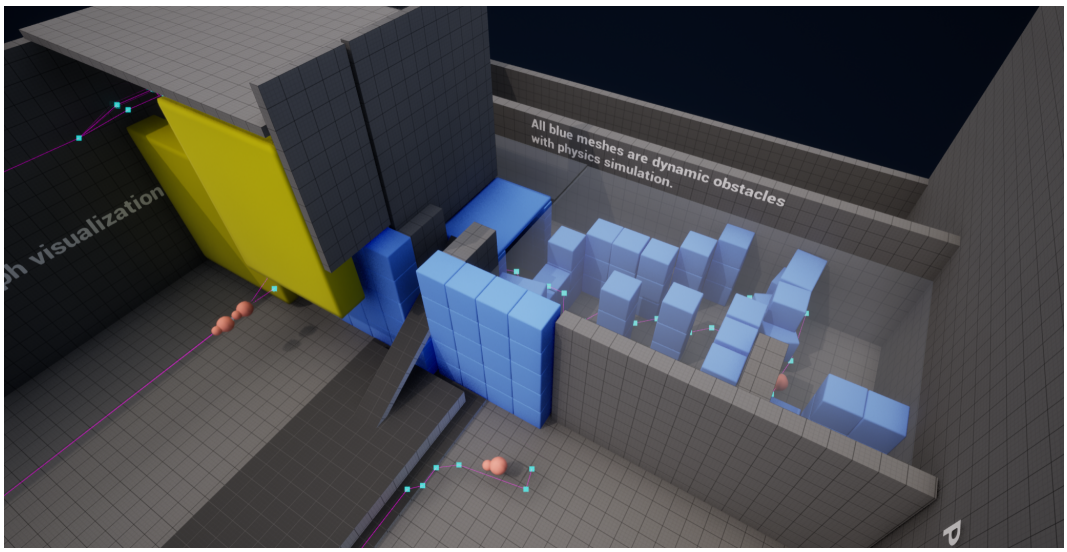


*Fig. 3.1* Screenshot of one of the example levels in Unreal Engine 5. The blue and

yellow objects are moving dynamic obstacles. We can see the small pink Pawns navigating through the maze of real-time updated maze by following the path obtained via calling FindPath few times per second.

Each AActor in Unreal Engine has *ActorBounds*. This is the smallest box shape that fits the entire actor, and it's recalculated each frame (also known as bounding box). Using this box, we can easily determine which octrees need to be regenerated. The container that holds indexes to regenerate is also a set, so each index is unique. Thanks to this, number of dynamic obstacles doesn't have as much performance impact - what matters is the number of octrees marked for regeneration, so the bigger an obstacle is, the more expensive it is to update.

### 3.1.2 Graph Traversal

The first node accessed when performing a FindPath() call is a leaf which contains the starting point, passed as a location vector in world space. This vector is transformed into the local space of ACPathVolume's collision box, and if it's not within it, an empty path is returned with a proper error value.
The array CPathOctree* Octrees is a flattened 3D array that contains all octree roots (depth 0). On that account, accessing the correct octree at depth 0 is as simple as dividing it by VoxelSize $*\, 2^{\text{MaxDepth}}$ and transforming it into this array's index. After that, we check if the octree has children. If it doesn't, it means that this is the leaf, if it does, we call a recursive function that returns the correct child tree based on location, until the tree has no children.

Since each octree has up to 8 children, index of a child can be saved using 3 bits. With MaxDepth predefined, we can keep the address of any subtree within our graph in a single integer. With MaxDepth equal to 3, we need 2 bits for the depth itself, 9 bits for children indexes, and we're left with 21 bits for the depth 0 index.

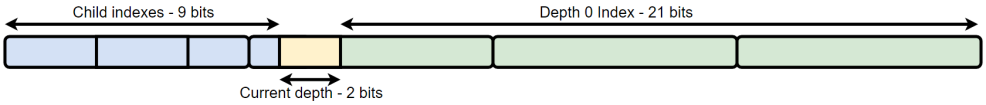


*Fig. 3.2* Tree index address packed into a 32-bit integer.

This format allows for quick access to particular nodes when we have a constructed index. It is especially helpful when accessing neighbors, since then in most scenarios we only need to change child indexes.

For this purpose, my code has many inline functions to modify or extract data from this TreeIndex, such as:

```
inline uint32 CreateTreeID(uint32 Index, uint32 Depth) const
inline uint32 ExtractOuterIndex(uint32 TreeID) const
inline uint32 ExtractDepth(uint32 TreeID) const
inline void ReplaceDepth(uint32& TreeID, uint32 NewDepth)
inline uint32 ExtractChildIndex(uint32 TreeID, uint32 Depth) const
inline void ReplaceChildIndex(uint32& TreeID, uint32 Depth, uint32
    ChildIndex)
```

And more similar ones but combined for optimization or ease of use. The other crucial traversal functionality is obtaining free neighboring leaves. The problem here is that the leaves may be at different depths. When looking for a neighbor on the side of a leaf, this neighbor might be:

- Child of the same parent - in this case, we need to access the parent and get a correct child.
- Child of a neighbor of a parent - go back in depth until we get to the parent of this neighbor or reach depth 0.
- Depth 0 - We get a neighbor from the octrees array by transforming the index.

Once we get the neighbor, it is not guaranteed that it's a leaf. There are two cases:

- The neighbor is the same size or larger and has no children (is a leaf) - we simply return it.
- The neighbor is the same size and has children - in this case, all four children on the proper side are neighbors. This needs to be recursive because children of this neighbor might also have children, then we'll get 16 neighbors, and so on.

To make this process faster and avoid unnecessary calculations, I have made some constant definitions. First of all, children of an octree are numbered like so:

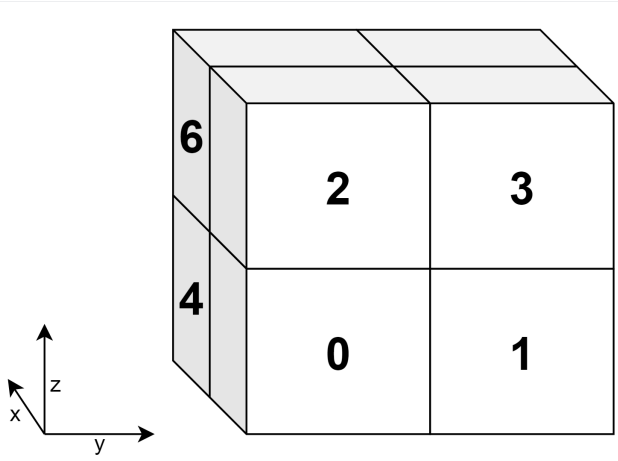

*Fig. 3.3* Octree's children numeration

I have also numbered each direction in which we can have a neighbor, to allow the existence of a readable enum:

```
enum EneighborDirection
{
        Left ,    // -Y
        Front ,   // -X
        Right ,   // +Y
        Behind ,  // +X
        Below ,   // -Z
        Above     // +Z
};
```

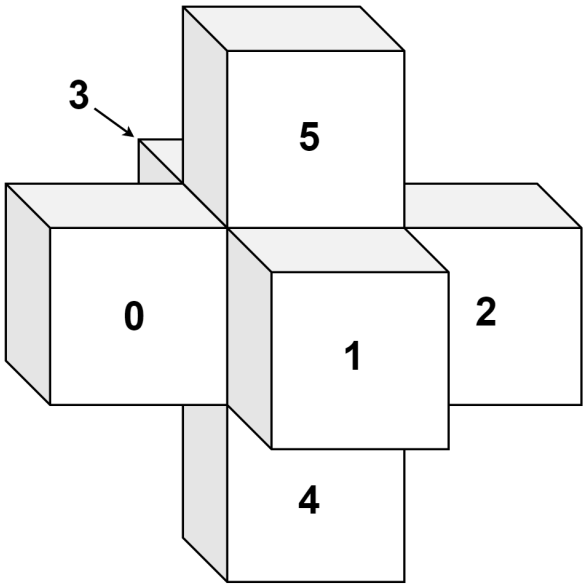


*Fig. 3.4* neighbor direction visualization

Thanks to this, I was able to create some constant lookup tables. I will describe two of the most important ones.

```
const int8 ACPathVolume::LookupTable_neighborChildIndex[8][6] = {
        {-2,-5, 1, 4,-3, 2},
        { 0,-6,-1, 5,-4, 3},
        {-4,-7, 3, 6, 0,-1},
        { 2,-8,-3, 7, 1,-2},
        {-6, 0, 5,-1,-7, 6},
        { 4, 1,-5,-2,-8, 7},
        {-8, 2, 7,-3, 4,-5},
        { 6, 3,-7,-4, 5,-6},
};
```

This table requires child index (values 0 to 7) of the child we want to get the neighbor of, and EneighborDirection. Positive numbers are children of the same

parent, negative numbers are children of a neighbor of this parent. So, for example, LookupTable_neighborChildIndex[0][2] == 1. We are looking for the Right neighbor of a child at child index 0. A quick look at figure 3.2 shows us that we obtained a correct child index, but if we're looking for the left neighbor of the same node, we would receive -2. Since this number is negative, we must first get the left neighbor of the parent of this child. Once we get it, we add 1, negate the negative number, and obtain 1, which is the correct child index, since it's adjacent to our current node.

Another useful lookup table:

```
const int8 ACPathVolume::LookupTable_ChildrenOnSide[6][4] = {
        {0, 2, 4, 6},
        {0, 1, 2, 3},
        {1, 3, 5, 7},
        {4, 5, 6, 7},
        {0, 1, 4, 5},
        {2, 3, 6, 7}
};
```

This is used to get child indexes on a requested side of an octree. Example of use:

```
uint8 ChildrenOnLeftSide[4] = LookupTable_ChildrenOnSide[
    EneighborDirection::Left];
```

## 3.2 Pathfinding

Three classes perform the pathfinding process:

- **CPathAStar**
- **UCPathAsyncFindPath** - The 'U' prefix is enforced by Unreal Engine API for every class inheriting from UObject.
- **FCPathRunnableFindPath** - Unreal Engine API also enforces the 'F' prefix here, since it extends the FRunnable class.

### 3.2.1 UCPathAsyncFindPath

This class inherits from UBlueprintAsyncActionBase, which lets it appear as an *Async* node in every blueprint. Async nodes are called through the normal blueprint execution pins, however, they may have multiple exit execution pins that fire asynchronously - invoked through C++ delegates. Each exit pin may also return different values for the return output.

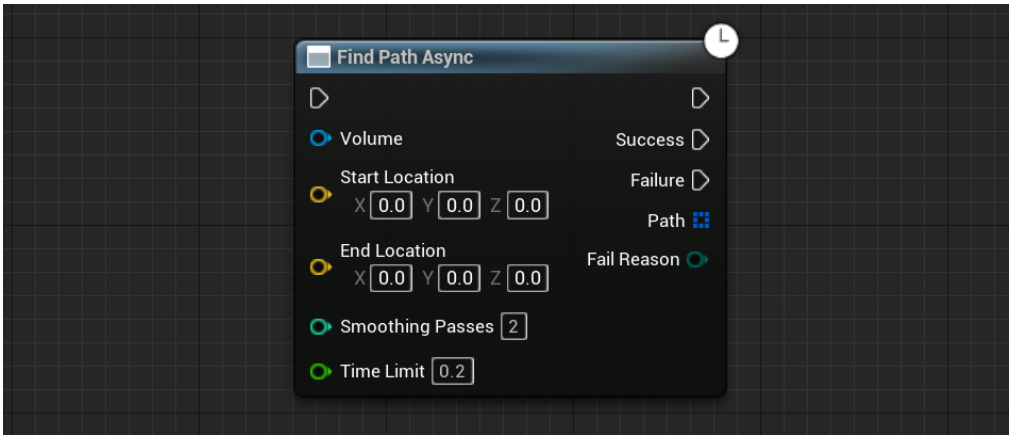


*Fig. 3.5* Find Path Async node in a blueprint

This class creates FCPathRunnableFindPath and runs it on a new thread. It also creates an instance of CPathAStar and passes it to the thread.

### 3.2.2 CPathAStar

This class performs the actual pathfinding. It uses a pointer to ACPathVolume to traverse the graph. First, it finds the closest leaves to start and target position. If suitable leaves are found, it searches for a path between them using the $A*$ algorithm. Each visited leaf is wrapped into a CPathAStarNode structure. This structure contains a hash method, compare operators, a pointer to the node it was discovered from, its fitness score, and tree index of the leaf it describes. Each visited node is inserted into an unordered set for quick "contain" checks. Nodes taken from priority queue are inserted into std :: vector so that the previous nodes can be referenced while obtaining a path and safely deleted along with the CPathAStar instance.
This class is also responsible for path post-processing.

### 3.2.3 Path post-processing

There are two steps performed after finding a path.

- **Smoothing** - The user can set how many smoothing passes should be performed after each FindPath() call. A smoothing pass attempts to remove half of the path's nodes. For each of three consecutive nodes $A, B$, and $C$, a *sweep* collision check is performed, from $A$ to $C$. If there is no collision, $B$ is removed. This could be optimized by first removing nodes in straight lines, as these do not require collision checks.
- **Transforming to user path** - First, all nodes on the same line are removed. After this pass, only nodes that make the path turn by more than Alpha degrees remain. Alpha has a default value of 3. During this process, CPathAStarNode s are transformed into a structure exposed to Unreal Engine's blueprints -

FCPathNode, inserted to Unreal Engine's TArray in the correct order and returned.

### 3.2.4 FCPathRunnableFindPath

This class is run on a dedicated thread, and its job is to manage CPathAStar. It waits if the graph is generating, correctly interrupts pathfinding if it has been explicitly stopped, and increments the running pathfinder thread count so that generating can't be started until it's finished.

# Chapter 4

# User guide

The user guide can be accessed in this document [**5**]. It was created this way to make it easily adaptable and accessible to users.

## 4.1 Installation

Before using this plugin, Unreal Engine 4.27, 5.0, or 5.1 must be installed [**4**].

Since this is a Code plugin, Microsoft Visual Studio 2017 (or newer) is required, along with its and Unreal's prerequisites [**6**].

After that, there are multiple ways to install the plugin:

- **Through Epic Games Launcher** - After installing the Epic Games Launcher [**3**], follow the previously mentioned User Guide.
- **Getting my project from GitHub** - Download or clone the repo and launch the project. Link in bibliography [**7**]
- **Manually adding plugin files to an existing project** - Get CPathfinding folder from the repo, and copy it to *YourUE5Project/Plugins/* Link [**8**]

## 4.2 Parameter tuning and explanation

Following parameters can be tuned in any Blueprint extending the ACPathVolume class.

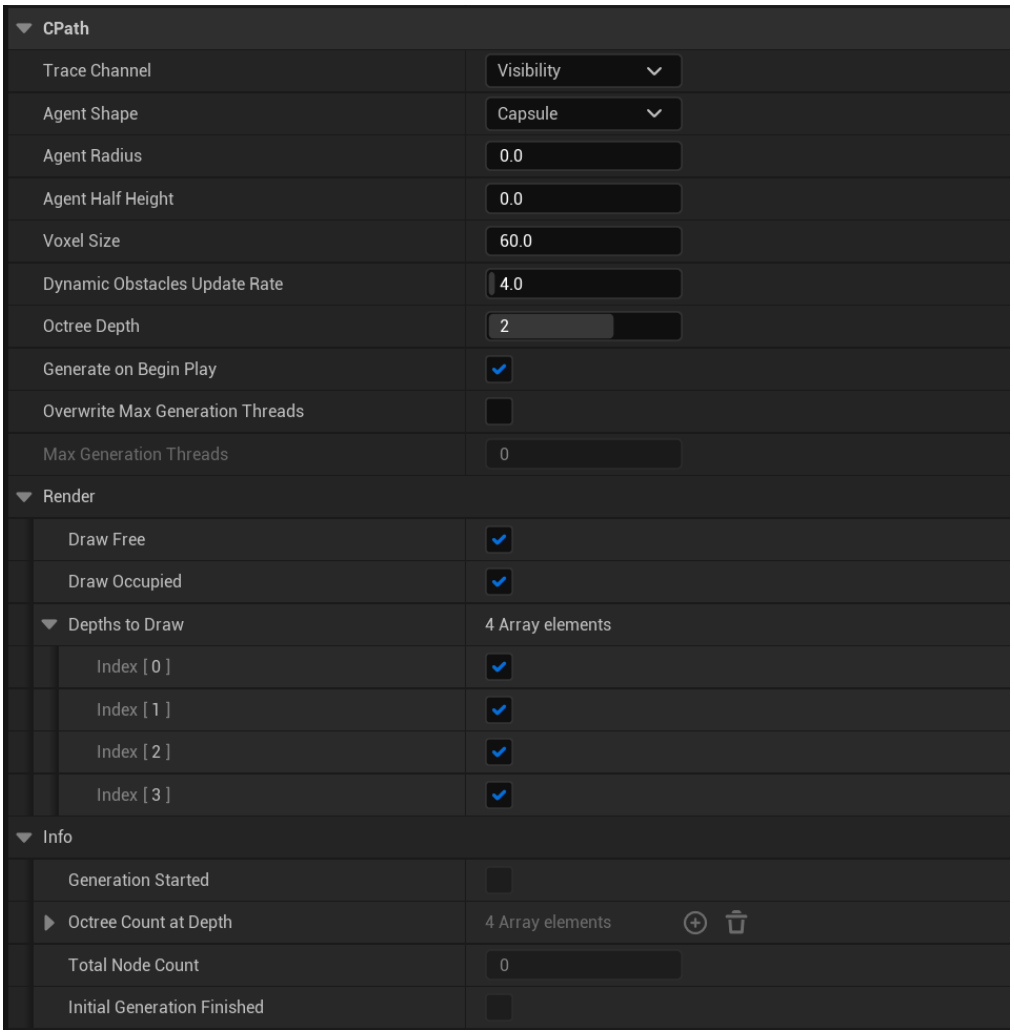


*Fig. 4.1* CPathVolume based blueprint, details panel.

### 4.2.1 Agent Settings

There are three settings representing the agent that will be following our generated paths:

- **Agent Shape** - Determines the shape of the agent's collision. It can be a *capsule*, a *sphere*, or a *box*. Select one that represents your agent's actual collision as closely as possible.

- **Agent Radius** - Radius for *capsule* and *sphere*, $X$ and $Y$ extent for the *box*.

- **Agent Half Height** - Half height ($Z$) for *capsule* and *box*. If *Agent Shape* is set to sphere, this is not editable.

During graph generation, if either of these dimensions are larger than the voxel size at the currently checked depth, an additional trace in the middle of the voxel is performed:

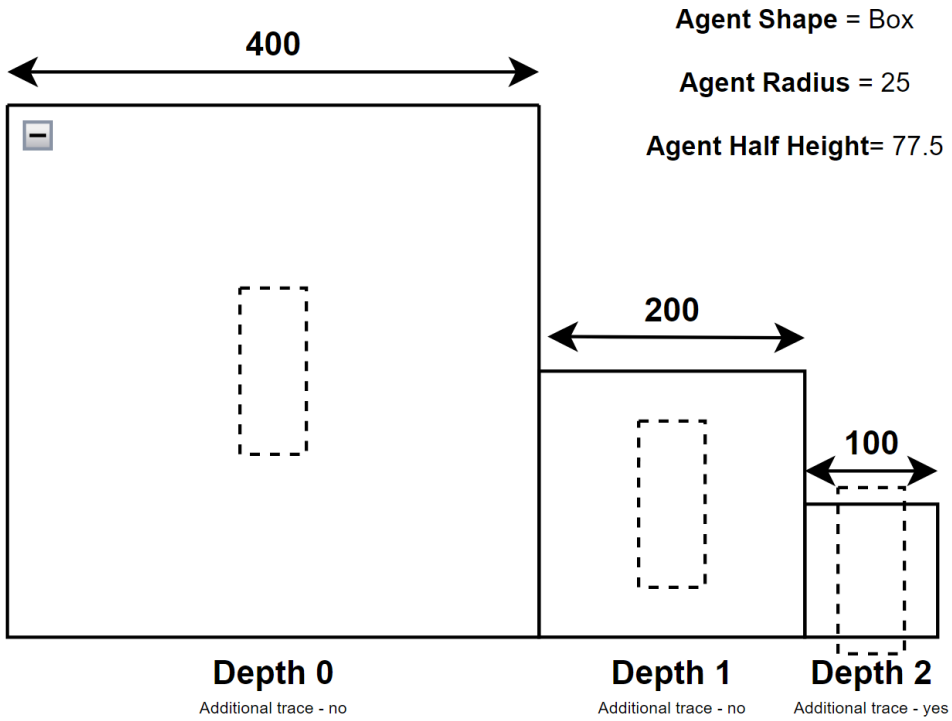


*Fig. 4.2* Agent settings visualization.

### 4.2.2 Octree Depth

This parameter sets how deep each octree can be. If you want the graph to be an unified grid, set it to 0. This is one of the most critical settings, performance wise. The default value of 2 is optimal in most cases. However, if the pathfinding space consists of a large open space, with few obstacles and walls, setting it to 3 will be more optimal. On the other hand, if your space is a maze with very few open spaces, or a dense building with many rooms, hallways, furniture, etc. - it should be set to 1. The highest depth is 3 and the reasoning for that can be read in the documentation chapter.

Developing an algorithm to determine an optimal depth based on geometry present on the level is a problem for another thesis. For now, the best way to optimize that is by comparing pathfinding search times from the same locations with different OctreeDepth values.

### 4.2.3 Voxel Size

This parameter sets the edge size of a voxel at OctreeDepth (so, the size of the smallest voxel). For example, in *figure 3.2*,
VoxelSize = 100 and OctreeDepth = 2. For better performance, this should be as large as the level design allows it to be. For precise pathfinding, it should be set to *half* of the smallest dimension of the *tightest* space you want your agent to traverse. If that condition is satisfied, your agent can go anywhere it needs to go. There is no point in making it even smaller in 3D space. For 2D side-scrolling, it may be set to much smaller values since the search space will be much smaller overall.

### 4.2.4 Dynamic Obstacles Update Rate

As the name suggests, this setting only has meaning if any dynamic obstacles within the volume are present. Generally, this should have as low value as possible. It means how many times per second the graph should be regenerated around dynamic obstacles. Suppose you have very few dynamic obstacles and you're using this option as an obstacle avoidance system on top of pathfinding, which in that case you may set it to higher values like 10, or even 30 for an accurate real-time obstacle avoidance. Keep in mind that while the graph is being generated, pathfinding calls and debug rendering calls are *waiting*. Therefore, you should only increase this value if you know what you're doing. Otherwise, there may be some unwanted behavior such as high CPU usage, pathfinding calls getting delayed by multiple frames, and Debug Drawing not updating fast enough to keep up with graph regeneration.

### 4.2.5 Generate on Begin Play

Setting it to true means that the graph will start generating as soon as it's spawned in the world. It may be unchecked, but then the GenerateGraph() method has to be called manually before any FindPath() call will execute. It might be helpful if there is some geometry that is loaded procedurally.

### 4.2.6 Overwrite Max Generation Threads

Checking this allows you to set a custom limit on concurrent threads spawned to generate the graph. By default, it uses up to PhysicalSystemCoreCount−1 threads. Splitting the generation into more threads brings minimal benefit and might cause the main game thread to be overwhelmed as well.

### 4.2.7 Render settings and info

When drawing debug graph, you may filter which subtrees should be visible. The Info tab provides read-only statistics about the generated graph.

### 4.2.8 *FindPath* Smoothing Passes

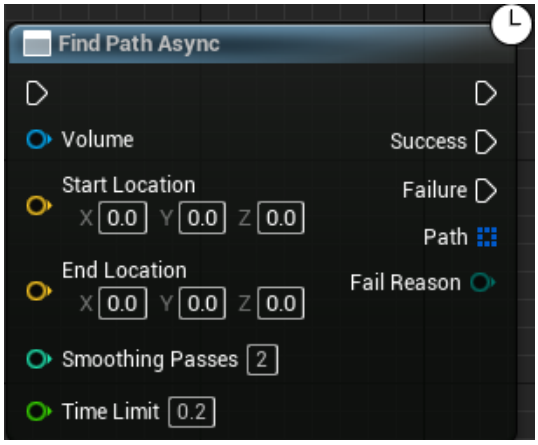


*Fig. 4.3* Find Path Async node.

For every smoothing pass, the amount of nodes in a path gets potentially halved. It performs collision checks but if you have some custom pathfinding fitness function with conditions other than just plain euclidean distance, setting this to anything higher than two leads to a potential data loss. In Figure 3.4, we can see a comparison of the same path with different Smoothing Pass parameters.

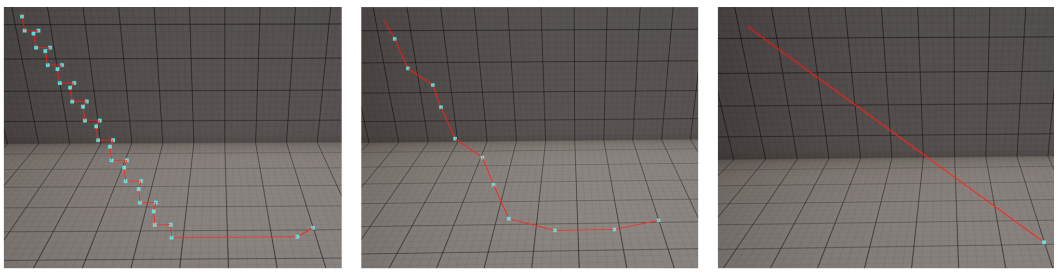


*Fig. 4.4* Three FindPath calls with the same coordinates but different *Smoothing Passes* parameters.

# Chapter 5

# Plugin after release

## 5.1 Installs

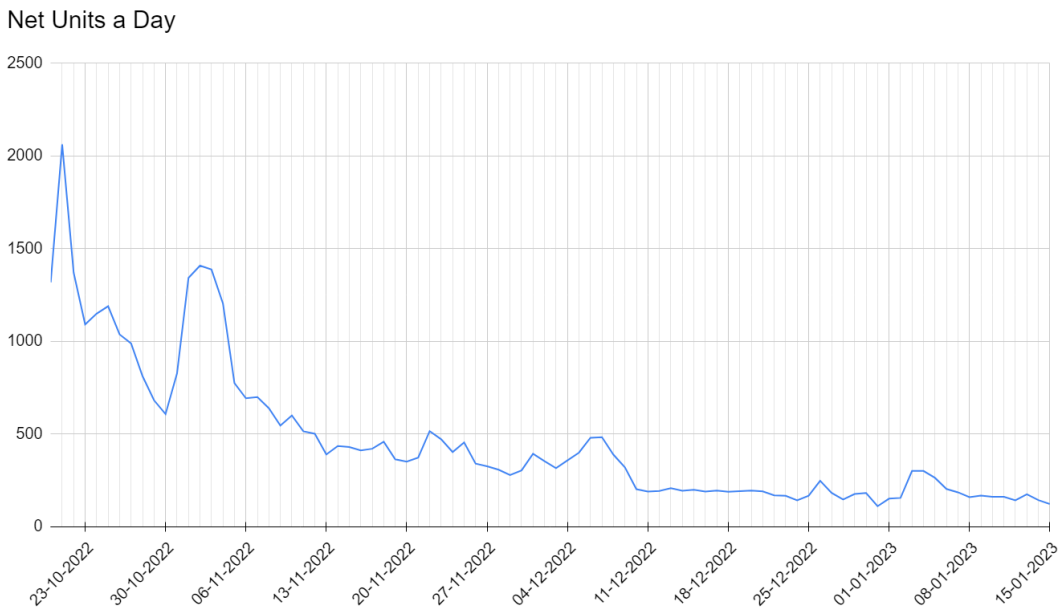


*Fig. 5.1* Daily user downloads during the first three months after release

The above chart represents plugin installs from Unreal Engine Marketplace, between October 20, 2022 and January 15, 2023. Surprisingly, without extensive advertising, the plugin got pretty popular at the beginning, averaging over one thousand downloads within the first 16 days.

After the initial influx, we can see a steady decline. There are two small spikes in December, corresponding to updates 1.01 and 1.02.

Overall, during this time period, 40,819 unique users have acquired the plugin.

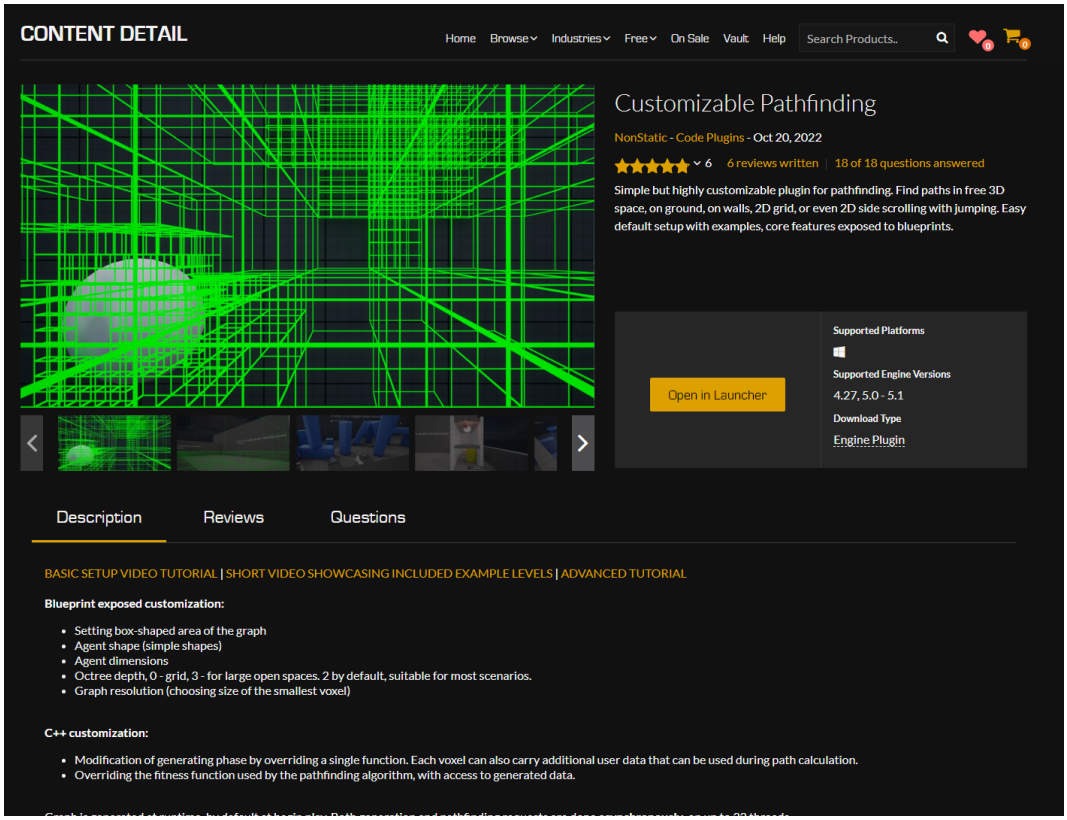


*Fig. 5.2* Fragment of the plugin page on the Marketplace

## 5.2 Marketplace Updates

### 5.2.1 Version 1.01

The first update mainly brought bug fixes. Some of the bugs were reported by users via email and the *questions* section in the marketplace. This update also added support for the UE 4.27 version. The code works the same on both Unreal Engine 4 and 5. However, the example content could not be transferred because of the editor differences. Therefore, I was forced to make a separate example level for the Unreal Engine 4 version from scratch.

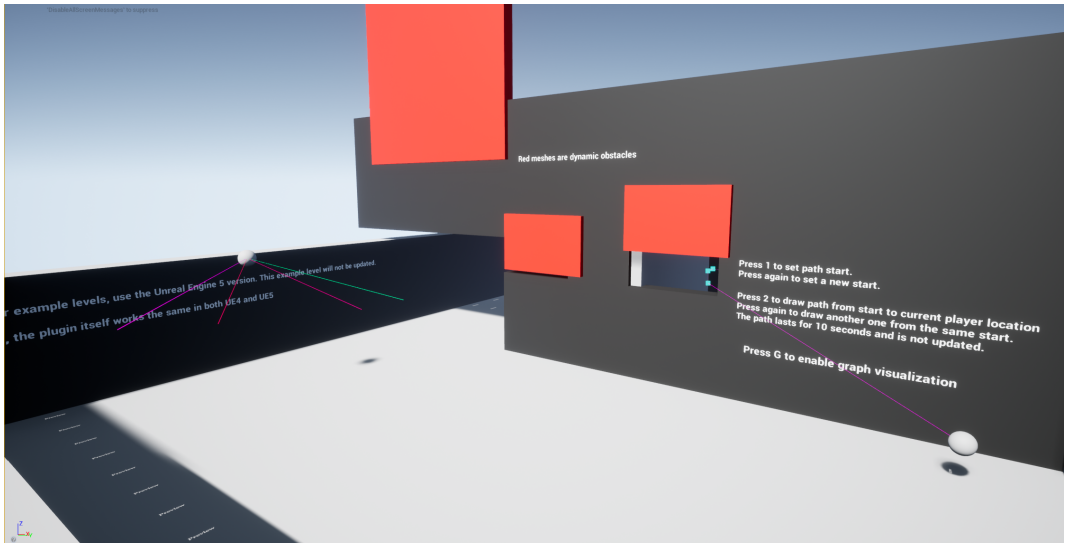


*Fig. 5.3* Screenshot of the UE 4.27 example level

Right after this update, I also created and uploaded two videos to YouTube with one of them being a short demonstration of the plugin **[10]** and the other one being an essential setup guide **[11]** which covers pretty much the same content as the document guide. Links to both videos are present on the marketplace page. Few users seem to click on them, since the videos have respectively about 1000 and 400 views (as of January 16, 2023).

| Video | Visibility | Restrictions | Date ↓ | Views | Comments | Likes (vs. dislike... |
|---|---|---|---|---|---|---|
| CPathfinding tutorial 02 - 2D grid and ... Marketplace link: https://www.unrealengine.com/marketpl... (12:05) | Public | None | Nov 26, 2022 Published | 314 | 2 | 100.0% 8 likes |
| CPathfinding Basic 3D setup guide Recorded using: - plugin version 1.02 - Unreal Engine version 5.1.0 Marketplace... (10:05) | Public | None | Nov 18, 2022 Published | 469 | 14 | 100.0% 7 likes |
| Customizable Pathfinding - Example le... Marketplace Link - https://www.unrealengine.com/marketpl... (5:19) | Public | None | Nov 1, 2022 Published | 1,091 | 0 | 100.0% 38 likes |

*Fig. 5.4* Performance of the YouTube videos created for this plugin as of January 16, 2023

### 5.2.2 Version 1.02

This update featured some convenience changes, cleaned up variable naming, and added a feature to pass an additional parameter while searching for a path. This can be utilized to change the pawn's pathfinding priorities based on some property - terrain preference, weight, stamina, etc.

The update introduced a third plugin version - for the newest Unreal Engine 5.1. This build was possible without code or content changes from the 5.0 version. I also created another YouTube video and released it with this update. The video **[12]** covers an example overriding the user exposed C++ methods, and using the

plugin for 2D platrofmer pathfinding. Within a month, it gained about 300 views, which is a bit better than the previous, basic guide.

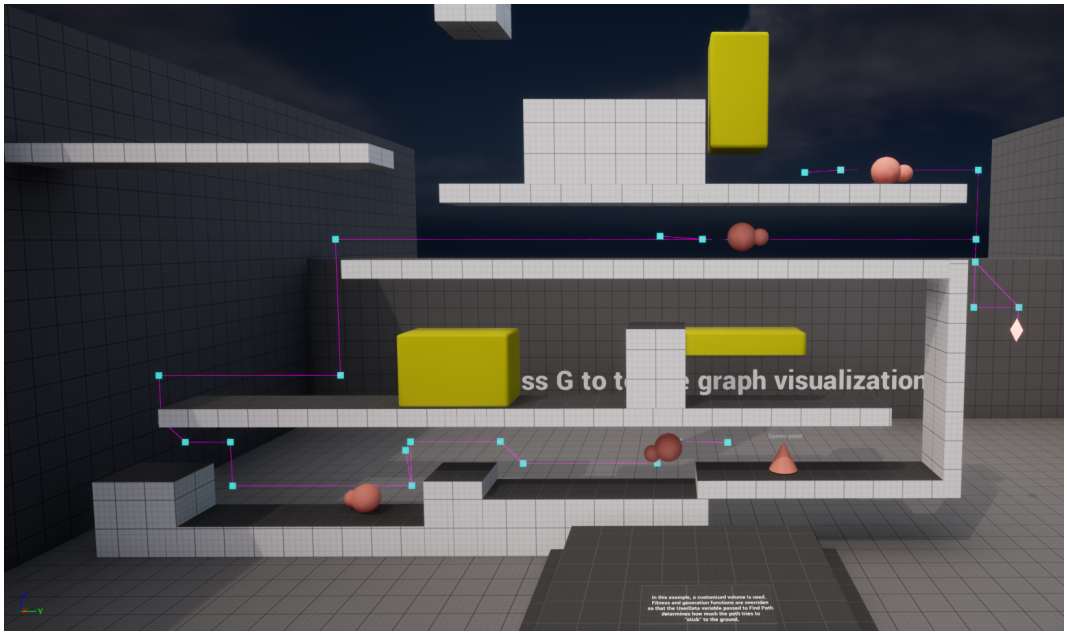


*Fig. 5.5* Screenshot of the 2D example level

## 5.3 User feedback, conclusions

Generally, the user feedback was very positive. As of January 16, 2023, there are six reviews on the marketplace, all positive. There are also 18 answered questions with some turning into back-and-forth conversations. Throughout these months I also got multiple emails asking about help with the setup or the future of this plugin, as well as some positive comments under the YouTube videos.

The plugin seems to serve its function of providing functionality that cannot be found in other plugins and Unreal Engine itself. Multiple users have already used it in their projects and even integrated it to other related plugins on the marketplace.

All in all, it got way more attention than I anticipated, and I'm very happy with the result.

...

# Bibliography


[1] http://idm-lab.org/bib/abstracts/papers/aaai10b.pdf - **Lazy Theta*: Any-Angle Path Planning and Path Length Analysis in 3D** by Alex Nash, Sven Koenig and Craig Tovey*

[2] https://docs.unrealengine.com/5.0/en-US/ - **Unreal Engine 5 Documentation***

[3] https://store.epicgames.com/en-US/download - **Epic Games Launcher download***

[4] https://www.unrealengine.com/en-US/download - **Unreal Engine 5 download***

[5] https://docs.google.com/document/d/1j3NQCgR5H0Uz8qd1RfNvZPzv JEcxNIapOkz0MESesWY/edit - **User guide document***

[6] https://visualstudio.microsoft.com/downloads/ - **Visual Studio download***

[7] https://github.com/NonStaticGH/CPathHostProject - **GitHub repo***

[8] https://github.com/NonStaticGH/CPathHostProject/tree/main/CPath_UE5HostProject/Plugins/CPathfinding - **Plugin files direct download***

[9] Hart, P.; Nilsson, N.; and Raphael, B. 1968. A formal basis for the heuristic determination of minimum cost paths. IEEE Transactions on Systems Science and Cybernetics - **Original A-Star document**

[10] https://youtu.be/HhH-_kPN7j4 - **Plugin demonstration video***

[11] https://youtu.be/GFOeX_Xd0Z8 - **Basic setup guide video***

[12] https://youtu.be/3wz0_jjlWFU - **Advanced features video***

[13] https://www.unrealengine.com/marketplace/en-US/product/flying-navigation-system - **Flying Navigation System plugin, Marketplace link***

[14] https://www.unrealengine.com/marketplace/en-US/product/c533ad0234c540fb81bdfa7bd55d3514 - **Customizable Pathfinding, Marketplace link***


**All links accessed and valid as of January 16th 2023*