INSTALL, UNDERSTAND, FLY

Quick Start

LCM Nav3D 2.2.0 • part of the complete User Manual

Contents

1 Tutorial 01: Installation

TL;DR: Put the plugin in `YourProject/Plugins/LCMNav3D/`, open the project and let it compile, enable **Show Plugin Content** in the Content Browser, then open a demo map and press Play. If the demo map works, your install is good.

Time: ~10 minutes (plus one compile).

1.1 Before you start

Requirement	Detail
Unreal Engine	5.8
Platform	Windows 64-bit or Linux

Requirement	Detail
Project type	**C++ project** for a manual install (see below)
Compiler	Visual Studio 2022 with the *Game development with C++* workload (Windows)

The plugin ships as source, so something has to compile it once. Which "something" depends on how you install it.

1.2 Method A: Installed from Fab / Epic Games Launcher

This is the simplest path and works with **Blueprint-only projects**.

1. Install the plugin from the Launcher's **Library** → **Fab Library** section. It lands in `<Engine>/Plugins/Marketplace/LCMNav3D/` and is already compiled for 5.8.
2. Open your project.
3. **Edit** → **Plugins**, search "LCM Nav3D", tick **Enabled** if it isn't already.
4. Restart the editor when prompted.

Skip to Verify the install.

1.3 Method B: Manual install into your project

Use this when you have the plugin as a folder (for example from a direct purchase or a version you re-targeted yourself).

1. Close the editor.
2. Copy the `LCMNav3D` folder into your project so the layout is:

```
YourProject/
  YourProject.uproject
  Plugins/
    LCMNav3D/
      LCMNav3D.uplugin
      Source/
      Content/
      Config/
      Shaders/
```

> Create the `Plugins` folder if your project doesn't have one. The folder must be named exactly `LCMNav3D`, because the content mount `/LCMNav3D/` depends on it.

3. **If your project is Blueprint-only, convert it to C++ first.** A Blueprint-only project has no build pipeline and cannot compile a source plugin. The easiest conversion: **Tools** → **New C++ Class** → **None** → **Create Class**. The editor generates a `Source/` folder and restarts. You never have to write any C++ afterwards.
4. Right-click `YourProject.uproject` → **Generate Visual Studio project files**.
5. Open the generated `.sln` and build **Development Editor | Win64**, or open the `.uproject` and click **Yes** when the editor offers to rebuild.
6. The plugin is `EnabledByDefault`, so it turns itself on. Confirm under **Edit** → **Plugins**.

1.4 Verify the install

Do all three of these. Each one catches a different kind of failure.

1. The plugin loaded

Edit → **Plugins** → **AI Navigation** category. You should see **LCM Nav3D**, enabled, version 2.2.0.

2. You can see the plugin's content

Plugin content is hidden by default in the Content Browser:

Content Browser → Settings (the gear icon) → tick "Show Plugin Content".

A **LCM Nav3D Content** folder appears in the tree. Without this you will not find any demo map, and it is by far the most common "the plugin didn't install" false alarm.

3. A demo map actually runs

Open **`LCM Nav3D Content → Demo → BehaviourTree → InfniteMaps → Vertical_Skyscraper_BT`** and press **Play**.

You should see agents flying through the building: over floors, through gaps, not along the ground. Give it a second or two on first play: the navigation volume builds at `BeginPlay`.

> **`Demo → MainMenu`** is a gallery of every demo in the plugin, if you would rather browse than open maps by name.

If that works, **your install is correct** and you can move on to Tutorial 02: Core Concepts.

1.5 The engine plugins it turns on

`LCMNav3D.uplugin` declares these as dependencies, so the engine enables them for you. You do not need to touch them, but it is worth knowing they are on:

Plugin	Why it's needed
`StateTree`, `GameplayStateTree`	The StateTree task/condition/evaluator set
`MassEntity`, `MassGameplay`, `MassAI`	The Mass crowd layer
`SmartObjects`	Smart Object behaviour definitions
`DataValidation`	In-editor validation of illegal Mass trait combinations

> **If the plugin fails to load with `GetLastError 126`**, one of the Mass plugins got disabled. `MassGameplay` in particular is required at runtime. Re-enable it and restart.

1.6 Common install problems

"The following modules are missing or built with a different engine version: LCMNav3D" The plugin hasn't been compiled for your engine build. Click **Yes** to rebuild. If the rebuild fails, you are on a Blueprint-only project, so go back to Method B step 3.

I can't find the demo maps. You haven't enabled *Show Plugin Content*. See verification step 2.

The plugin is missing from the Plugins window entirely. The folder is in the wrong place or misnamed. It must be `YourProject/Plugins/LCMNav3D/LCMNav3D.uplugin`. Check you didn't end up with a doubled folder like `Plugins/LCMNav3D/LCMNav3D/`.

Compile errors mentioning StateTree or Mass types. An engine plugin dependency was disabled. Re-enable the table above and regenerate project files.

More in Troubleshooting.

1.7 Other engine versions

This copy targets **Unreal Engine 5.8**. The plugin ships as a separate copy for each supported engine version, **5.2 through 5.8**, and each one is built and verified against that engine.

Install the copy that matches your project rather than re-targeting this one. Engine APIs move between versions, and the per-version copies already carry those differences.

Next: Tutorial 02: Core Concepts. What the navigation manager is actually doing, and the three settings that decide whether it works well in your level.

2 Tutorial 02: Core Concepts

TL;DR: LCM Nav3D carves your level's **empty air** into a tree of boxes (a Sparse Voxel Octree) and paths through the empty ones. You place one **`LcmNavigationManagerSVO`** actor per level, tell it how big your world is and how small the tightest gap is, and it does the rest.

Time: ~10 minutes reading. No project changes. This is the chapter that makes every later setting make sense.

2.1 1. Why not just use Unreal's navmesh?

Unreal's Recast navmesh is a **surface**. It answers "where can I stand?" by flattening your level onto walkable polygons. That is exactly right for a soldier and completely wrong for a dragon, because the interesting space for a flyer is the part with nothing in it.

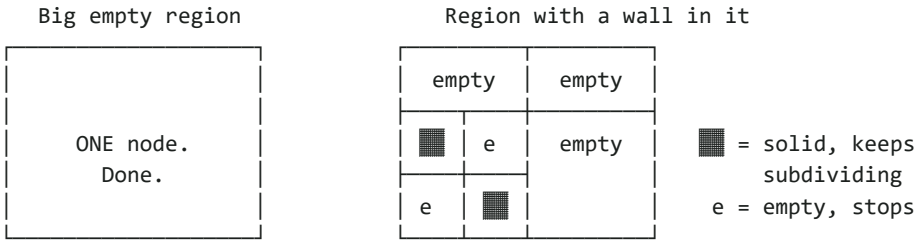
LCM Nav3D answers a different question: **"where is there empty air?"** It represents the volume, not the floor. That's the whole idea. Everything else is engineering around making that fast.

2.2 2. The Sparse Voxel Octree, in one picture

Imagine a giant cube around your level. Now:

1. Is the cube completely empty? **Yes** → done, one big box of free space.
2. **No**, something's in it? → cut it into 8 smaller cubes and ask each one the same question.
3. Repeat until the cubes reach your minimum size.

That's it. That's the octree.



"Sparse" is the important word. Open sky costs almost nothing: one node covers a huge volume. Detail only appears where geometry actually is. This is why a 2 km map doesn't cost 2 km worth of memory: most of it is air, and air is cheap.

What this means for you: the cost of your navigation data is driven by *how much clutter you have*, not by how big your level is.

2.3 3. The one actor you must place

`LcmNavigationManagerSVO`: drop one into your level. One per level, that's the rule.

It owns the octree, runs the pathfinder, and answers every query. Every agent talks to it. If it isn't in the level, nothing navigates.

Its Details panel is deliberately organised in the order you should think about it:

Category	What it's for
1. Setup	Finite or infinite world, the biggest decision
2. Voxels	Size and resolution
3. Collision	Which geometry counts as an obstacle
4. Performance	Frame-budget knobs

Category	What it's for
5. Debug	Visualisation
6. Runtime State	Read-only live info

Full field-by-field detail is in the Settings Reference. Here we only cover the concepts behind them.

2.4 4. Finite vs Infinite: the decision that shapes everything

This is **`1. Setup → Infinite World`**, and it changes which other settings even appear.

Finite (default, `Infinite World` unticked)

One fixed box of navigation, built once. You set **`WorldExtent`** (default **10000 cm** = a 200 m cube, since extent is measured from the centre outward).

- Built at `BeginPlay` if **`Auto Build On Play`** is on (it is by default).
- Simple, fast, completely predictable.
- **Use this for:** arenas, levels, dungeons, single buildings, anything with a boundary.

Start here. Most projects never need anything else.

Infinite (`Infinite World` ticked)

The world is divided into **chunks** of **`ChunkSize`** (default **8000 cm**). Chunks are generated around your agents as they move and thrown away behind them.

- **Use this for:** open worlds, streaming levels, procedural terrain, anything without a boundary.
- Costs more complexity: chunks must generate before an agent can route through them.

> **`ChunkSize` is not a free dial.** It decides both memory *and* whether distant routes can be found at all. **8000 is the tested default. Change it only with a reason and re-test.** Larger chunks mean fewer, heavier generations; smaller means more, lighter ones, and more seams to cross.

2.5 5. Resolution: the two settings people get wrong

`MinVoxelSize` (default 100 cm)

How small the smallest box is allowed to get. **This is the single biggest driver of both quality and cost.**

- **Smaller** → finds tighter gaps, uses more memory, takes longer to build.
- **Larger** → cheaper and faster, but narrow openings vanish and agents route around them.

> ⚠ **It's a floor, not an exact size.** The octree halves its way down, so the real leaf size is the world (or chunk) size repeatedly divided by two until it's about your `MinVoxelSize`. Two consequences that surprise people:
>
> - Changing `MinVoxelSize` from 100 to 90 may change **nothing** (same number of halvings).
> - Changing it from 100 to 60 may cross a threshold and **double your memory** in one step.
>
> Tune it by testing, not by assuming the number is literal.

Rule of thumb: set `MinVoxelSize` to roughly **half the narrowest gap** your agents must fly through. A 3 m window needs about 150 cm or finer.

`ClearancePadding` (default 100 cm)

Inflates obstacles by this much when voxelizing, so agents don't clip walls.

> ⚠ **The classic mistake: setting this too high seals your level.** If padding approaches the size of your voxels, doorways and corridors fill in completely and the pathfinder reports "no route" through an opening you can plainly see. **Keep `ClearancePadding` well below `MinVoxelSize`.** If you need more clearance than that, lower `MinVoxelSize` too.

If agents refuse to path through an opening, this setting is the first thing to check.

2.6 6. How a path is actually found

For a short hop, it's a straight search through the octree. For anything longer, there are two levels:

```
1. MACRO: "which chunks/regions do I cross?"     (coarse, cheap, long-range)
                  ↓
2. MICRO: "which voxels inside them?"            (fine, detailed, short-range)
                  ↓
3. SMOOTH: turn the blocky voxel path into a nice line
```

You don't call these yourself; it's automatic. The reason to know is that it explains a common symptom: if a long route fails but a short one works, the problem is usually **macro**. Either chunks are not generated yet, or `ChunkSize` is interacting badly with your layout.

Choosing a solver

Solver	Character	Use when
A* *(Fastest, Grid Locked)*	Cheapest; paths follow voxel edges, so turns look square	You'll smooth the result anyway, or you need max throughput
Theta* *(Accurate, Any Angle)*	Most direct; cuts corners properly	Quality matters more than cost
Lazy Theta* *(Balanced)*	Nearly Theta* quality, near A* cost	**The sensible default**

Smoothing

Mode	Result
Raw Path	Straight out of the solver, blocky
Linear Shortcut	Removes redundant waypoints; straight segments
Curved Spline	Smooth curves: **what you want for flying things**

> Smoothing runs *after* pathfinding and is validated against geometry, so a smoothed path won't cut through a wall.

Where the work runs

Path solving runs on the CPU, on a worker pool, off the game thread. It is deterministic: same input, same output, every time - which is what makes replays, lockstep multiplayer and automated tests reproducible.

The GPU is used for **voxelization**: turning your geometry into the octree, where it is substantially faster. If no suitable GPU is present it falls back to the CPU automatically, so nothing breaks.

2.7 7. Things that move

Static geometry is baked into the octree. For things that move, add a **`LcmDynamicObstacleComponent`** to the actor. It restamps the octree as the actor moves, and agents reroute around it. That's Tutorial 05.

2.8 8. Terms you'll meet in the rest of the docs

Term	Meaning
SVO	Sparse Voxel Octree, the navigation data
Voxel / node	One box in the octree; a **leaf** is a smallest one
Chunk	One tile of an infinite world
Portal	A connection between two chunks: how routes cross a seam
Macro / micro	Coarse long-range vs fine short-range pathfinding
Clearance	How much room an agent needs to fit
Flow field	One shared steering field many agents sample (for crowds)

2.9 Recap

- The octree stores **empty space**, and empty space is cheap.
- **One `LcmNavigationManagerSVO`** per level.
- **Finite** unless you genuinely have an open world.
- **`MinVoxelSize`** ≈ half your tightest gap; it snaps to powers of two.
- **`ClearancePadding`** below `MinVoxelSize`, or you'll seal your own level.
- **Lazy Theta* + Curved Spline** is a good default for a flyer.

Next: Tutorial 03: Your First Flying Agent. Build a working agent in your own level.

3 Tutorial 03: Your First Flying Agent

TL;DR: Three ingredients: an **`LcmNavigationManagerSVO`** in the level, a **pawn with an AIController**, and a Behavior Tree whose only task is **`LCM Fly To (SVO)`** pointed at a blackboard key. No C++.

Time: ~20 minutes. **You'll end with:** a pawn that flies through 3D space to a goal in your own level.

3.1 What we're building

```
Level
  ├── LcmNavigationManagerSVO        ← the brain (one per level)
  ├── BP_MyFlyer  (Pawn)             ← the body
  │     └── AIController: BP_MyFlyerAI
  │             └── runs BT_MyFlyer
  │                     └── Blackboard: BB_MyFlyer  (Vector key "GoalLocation")
  └── TargetPoint                    ← where we want it to go
```

3.2 Step 1: Add the navigation manager

1. In the **Place Actors** panel, search **`LcmNavigationManagerSVO`**.
2. Drag one into your level. Position doesn't matter much. It uses its own location as the centre of the navigation volume, so put it roughly in the middle of your playable space.
3. Select it and set, in the Details panel:

Category	Field	Value
1. Setup	`Infinite World`	**unticked** (finite, see Tutorial 02)
1. Setup	`Auto Build On Play`	**ticked**
2. Voxels	`World Extent`	large enough to cover your play space (default **10000** = 200 m cube)
2. Voxels	`Min Voxel Size`	**100** to start
2. Voxels	`Clearance Padding`	**100** to start

> `World Extent` measures **outward from the actor**, so 10000 gives you a 200 m cube centred on the manager. If your agents will fly outside that box, increase it: outside the volume there is no navigation data and paths will fail.

See what it built

Tick **5. Debug** → **`Draw Debug Volumes`** and press Play. You'll see the octree drawn over your level: lots of big boxes in open air, small ones packed around geometry. **This is the single most useful thing you can do while tuning.** If a doorway you care about shows no small voxels, the pathfinder can't see it either.

Turn it back off when you're done; it's expensive to draw.

3.3 Step 2: Make the pawn

You need something that can move in 3D without gravity.

1. **Content Browser** → **Add** → **Blueprint Class** → **Pawn**. Name it **`BP_MyFlyer`**.
2. Open it and add a **Floating Pawn Movement** component. (Add Component → search "Floating Pawn Movement".) This is what makes it fly rather than fall.
3. Select `FloatingPawnMovement` and set **Max Speed** to **600**. Match this to the task's `Flight Speed` later.
4. Add a **Static Mesh** component so you can see it. A cone or sphere is fine. Scale it down to something sensible.
5. In **Class Defaults**, set **`Auto Possess AI`** to **`Placed in World or Spawned`**.

> **Collision tip.** For your first test, set the mesh's collision to **No Collision**. Physics fighting the navigation is a very common source of "my agent jitters / gets stuck", and you want a clean first result. Add collision back once flying works.

3.4 Step 3: Blackboard and Behavior Tree

The blackboard

1. **Add** → **Artificial Intelligence** → **Blackboard**. Name it **`BB_MyFlyer`**.
2. Add one key:
   - **Type:** **`Vector`**, **Name:** **`GoalLocation`**

> The Fly To task also accepts an **Object** key holding an Actor. Use a Vector for now; the Actor form is what you'd use for chasing, covered at the end.

The behavior tree

1. **Add** → **Artificial Intelligence** → **Behavior Tree**. Name it **`BT_MyFlyer`**.
2. Open it, and in the Details panel set **Blackboard Asset** to `BB_MyFlyer`.
3. Drag from **ROOT** and add a **Sequence**.
4. Drag from the Sequence and add the task named **`Fly To`**; it appears in the tree as **`LCM Fly To (SVO)`**.
5. Select that task and set:

Field	Value
Blackboard Key	`GoalLocation`
Acceptance Radius	`100`
Flight Speed	`600`

Leave everything else at defaults for now.

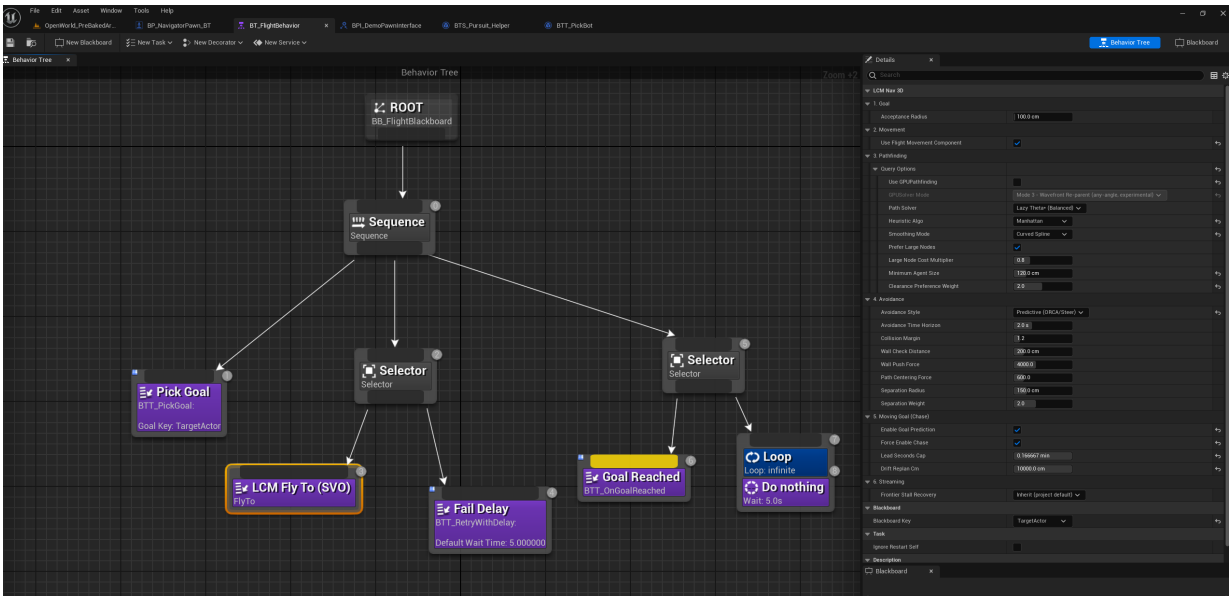


Figure 03.1 - Where `LCM Fly To (SVO)` sits. This is the shipped `BT_FlightBehavior` demo, one step past what you have just built: the same `Fly To` task, with a goal picker in front of it and a retry-with-delay branch beside it so a failed plan does not kill the tree. Yours needs only the highlighted node to start flying.

The FlyTo settings, in full

You will not need most of these on day one, but it is worth knowing the shape of the surface before you go looking for a specific knob.

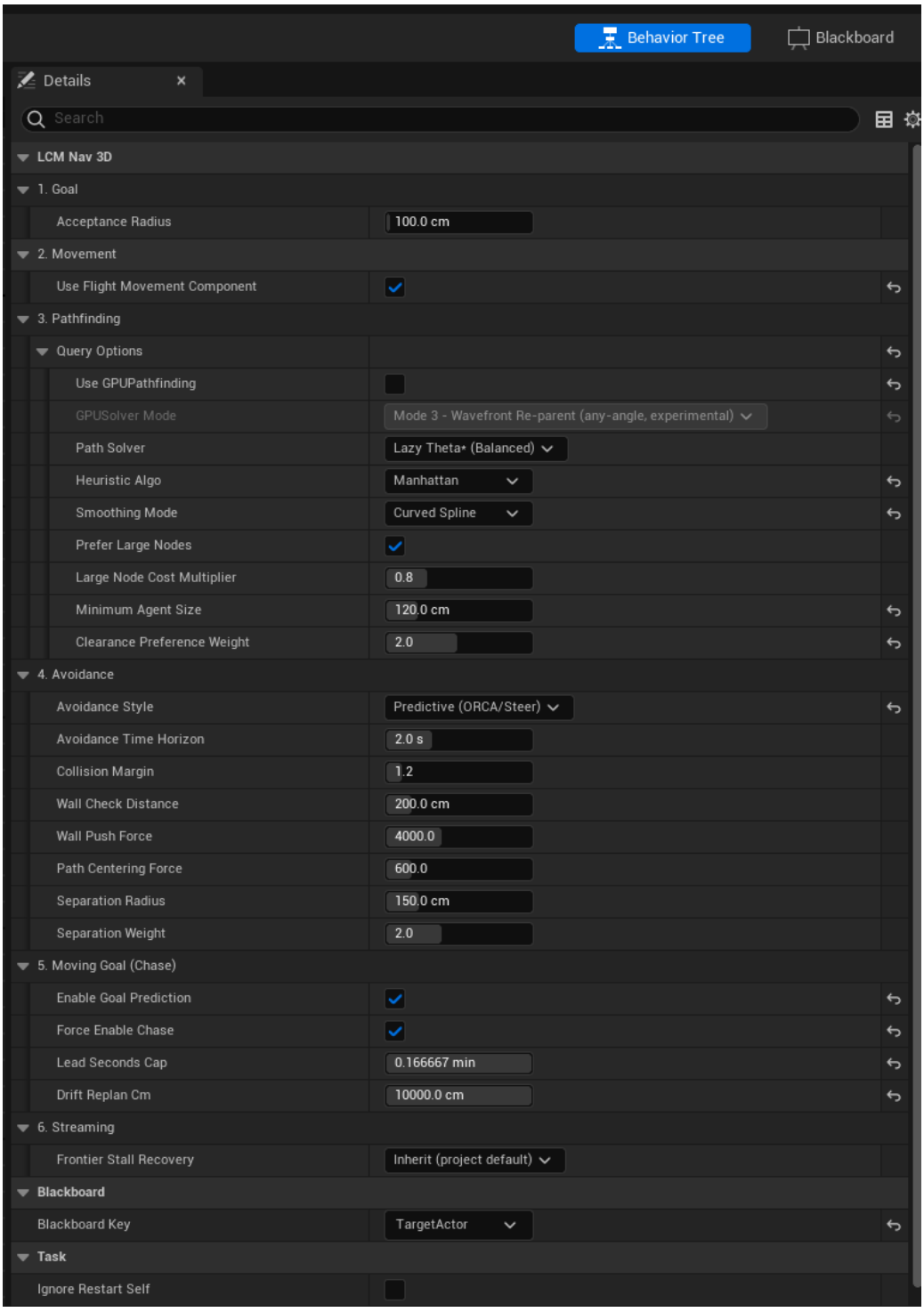


Figure 03.2 - The `LCM Fly To` task, all six sections. Read it as a pipeline: **1 Goal** is where you are going, **2 Movement** is what flies you there, **3 Pathfinding** is how the route is computed, **4 Avoidance** is how you get past other agents and walls, **5 Moving Goal** is only for chasing something that moves, and **6 Streaming** matters only in an infinite world.

Two fields decide the shape of everything below them:

- **`Use Flight Movement Component`** - ticked, the whole of section 2 collapses, because the movement component owns speed and dynamics instead of the task.
- **`Avoidance Style`** - set to `CBS`, the seven tuning fields under it hide; CBS is a planner-level solver and does not use them.

Sections you have not configured are not silently active. The panel hides any property the current configuration will ignore, so what you can see is what will run.

3.5 Step 4: The AI Controller

1. **Add → Blueprint Class → AIController**. Name it **`BP_MyFlyerAI`**.
2. Open it. On the **Event BeginPlay** node:
   - Add **Run Behavior Tree**, set **BTAsset** to `BT_MyFlyer`.
3. We need to put a goal in the blackboard. Still on BeginPlay, **after** Run Behavior Tree:
   - Add **Get Blackboard → Set Value as Vector**.
   - **Key Name:** `GoalLocation`
   - **Value:** for a first test, drag in a **Get Actor Of Class** node with **Actor Class** = **`TargetPoint`**, then **GetActorLocation** from it.

```
BeginPlay ─→ Run Behavior Tree (BT_MyFlyer)
          └→ Set Value as Vector (Key: GoalLocation,
                                  Value: GetActorOfClass(TargetPoint).GetActorLocation)
```

4. Compile and save.
5. Go back to **`BP_MyFlyer`** → Class Defaults → set **AI Controller Class** to `BP_MyFlyerAI`.

3.6 Step 5: Fly

1. Drag a **TargetPoint** into your level, somewhere your flyer has to climb or route around geometry to reach. Put it up high: that's the whole point.
2. Drag **`BP_MyFlyer`** into the level, somewhere with open air around it.
3. Press **Play**.

Your pawn should take off and fly to the target point, routing around geometry in full 3D.

3.7 If it doesn't move

Work down this list in order. These are the actual causes, in the order they occur.

Symptom	Cause	Fix
Output Log: **"No SVO Manager found in the level!"**	No manager actor	Step 1
Nothing happens, no log errors	AI never possessed the pawn	`Auto Possess AI` = *Placed in World or Spawned*, and `AI Controller Class` is set
Task fails instantly	Goal is outside the navigation volume	Increase `World Extent`, or move the target inside it
Task fails instantly	Goal (or start) is **inside** geometry	Move them into open air
Agent flies but ignores walls	Draw Debug Volumes shows no detail	`Min Voxel Size` too large
Agent refuses an obvious doorway	Opening sealed by padding	Lower `Clearance Padding` (see Tutorial 02 §5)
Agent jitters or sticks	Physics fighting navigation	Set mesh collision to No Collision for the test

Full list: Troubleshooting.

3.8 Step 6: Make it look good

Defaults are functional, not beautiful. Two changes transform the motion:

On the `LCM Fly To (SVO)` task:

Field	Try	Effect
`Flight Style`	**Physics (Drones/Ships)**	Momentum and inertia instead of instant direction changes. *(This is already the default.)*
`Flight Style`	**Linear (Biological/Magic)**	Instant, weightless. Good for spirits, insects, magic
`Banking Amount`	`4.0`	Rolls into turns like an aircraft
`Max Banking Angle`	`45`	How far it's allowed to roll
`Deceleration Distance`	`800`	Starts slowing this far out instead of stopping dead
`Max Acceleration`	`2000`	Lower = heavier, more sluggish feel

Path shape: a raw path is blocky. In Tutorial 02 we met the smoothing modes; **Curved Spline** is what makes flight look natural rather than like a series of ruler lines.

3.9 Step 7: Several agents at once

Duplicate `BP_MyFlyer` a few times and press Play. They'll fly the same route and crowd each other. Two settings fix that, both on the Fly To task:

Field	Default	Purpose
`Separation Radius`	`150`	How close before they push apart
`Separation Weight`	`2.0`	How hard they push

And **`Avoidance Style`**:

Style	Behaviour
None (Ghost/Ignore)	They fly through each other
Reactive (Boids/Push)	Push apart on contact: cheap, and the default
Predictive (ORCA/Steer)	Anticipate and steer around *before* colliding: smoother, costlier
Context-Based Steering (SVO)	Geometry-aware steering

For a handful of agents, **Reactive** is fine. For dozens sharing a corridor, **Predictive** looks markedly better.

> Past a few hundred agents, stop adding pawns and move to Tutorial 06: Mass Entity.

3.10 Step 8: Chasing a moving target

To chase something instead of flying to a fixed point:

1. Change the blackboard key `GoalLocation` to type **Object**, with **Base Class** = **`Actor`**.
2. Set it to the actor you want chased.
3. On the Fly To task, tick **`Enable Goal Prediction`**.

With prediction on, the agent aims where the target *will be* rather than where it is, so it intercepts instead of trailing. `Lead Seconds Cap` (default `2.0`) limits how far ahead it will extrapolate, and `Drift Replan Cm` (default `400`) controls how far the target must move before the path is recomputed.

3.11 Recap

- **One manager**, finite world, sensible `World Extent`.
- **Pawn + Floating Pawn Movement + Auto Possess AI**.
- **Blackboard key → `LCM Fly To (SVO)` → done.**
- **`Draw Debug Volumes` is your best debugging tool.**
- Tune feel with `Flight Style`, banking and deceleration; tune crowds with avoidance and separation.

Next: Tutorial 04: The Same Agent in StateTree, or jump to Tutorial 05: Dynamic Obstacles to make the world move.