Hierarchical Dynamic Pathfinding for Large Voxel Worlds

Benoit Alain
Lead Programmer & CTO, Sauropod Studio

Benoit Alain, Lead Programmer
Sauropod Studio Inc.

Benoit Alain, Lead Programmer
Sauropod Studio Inc.

Benoit Alain, Lead Programmer
Sauropod Studio Inc.

Benoit Alain, Lead Programmer
Sauropod Studio Inc.

Benoit Alain, Lead Programmer
Sauropod Studio Inc.

Benoit Alain, Lead Programmer
Sauropod Studio Inc.

Quick Feature Breakdown

- Many agents: avoidance, flocking
- Dynamic obstacles: doors, movable barrels
- Castle mechanics: stairs, block climbing
- Voxel mechanics: deformable terrain, buildable blocks

Performance constraints

- Large scale
- Lots happening at once
- Updates to pathfinding are seamless
- Characters react to changes immediately

Let's Get Started!

Talk overview

1. The Problem
2. Building our data structure
3. Hierarchical Pathfinding
4. Gameplay examples
5. CPU and memory performance
6. Conclusions

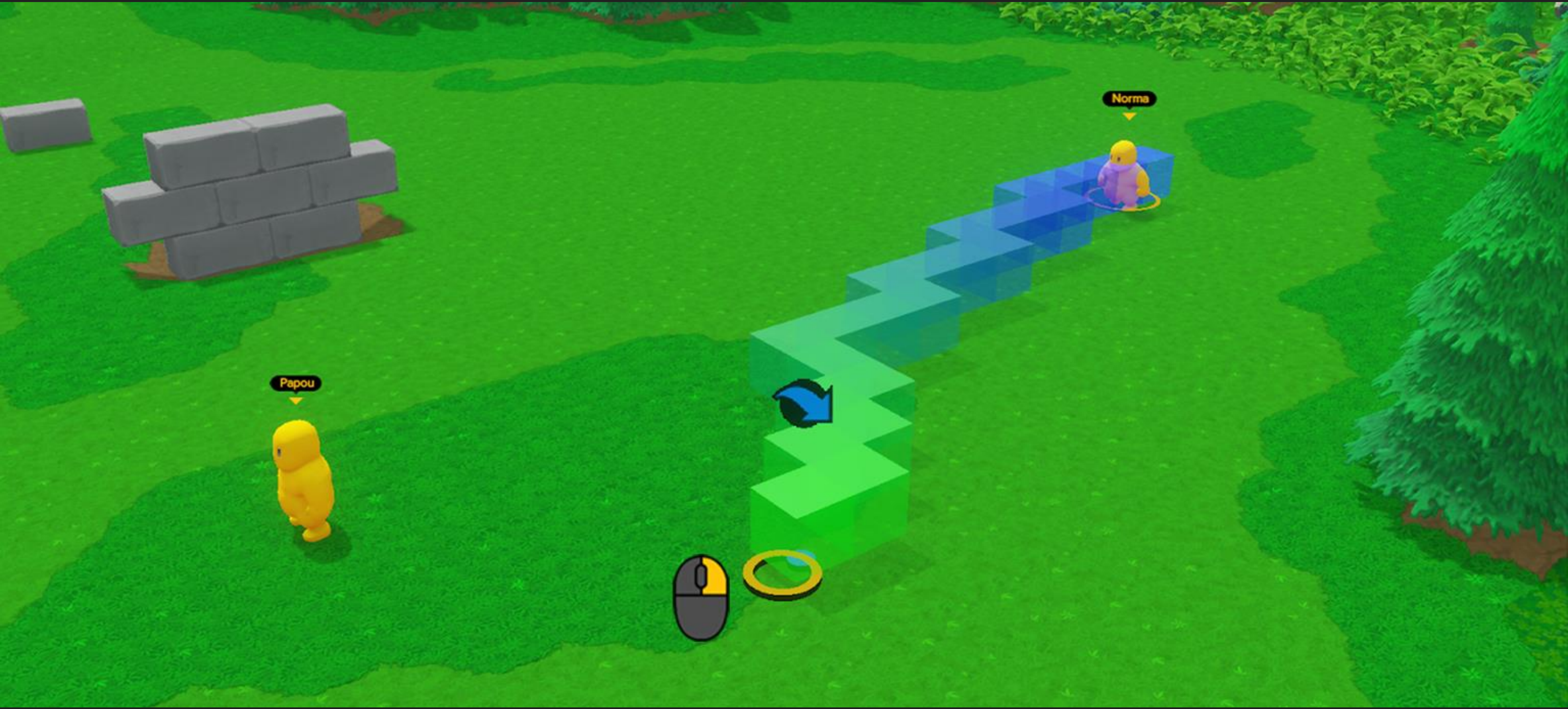
Benoit Alain, Lead Programmer
Sauropod Studio Inc.


Norma

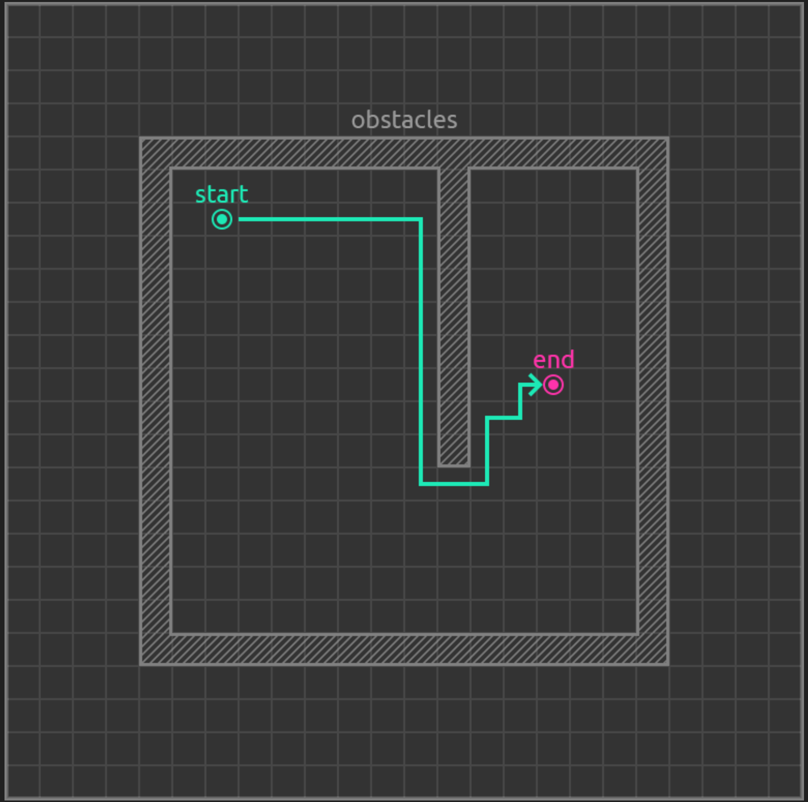

Norma

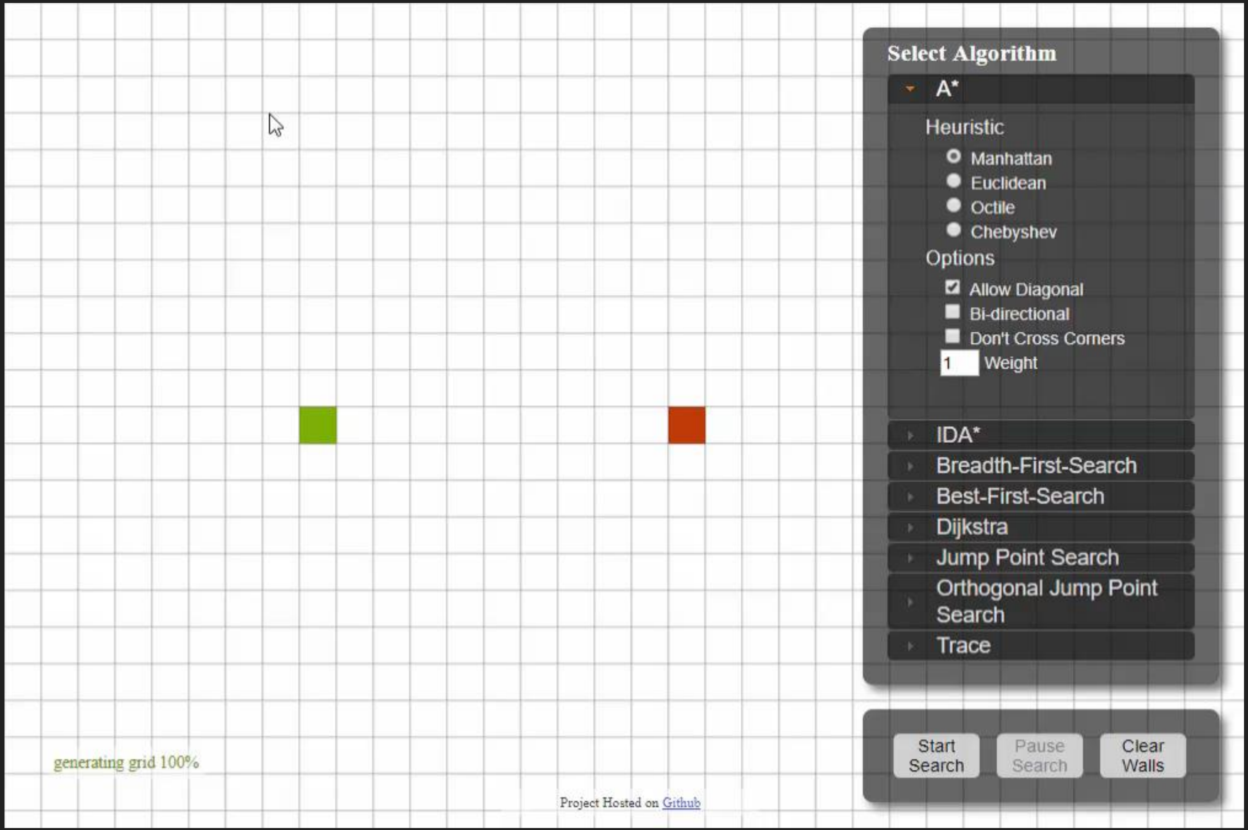

1
Norma


Norma
Papou


Norma
Papou

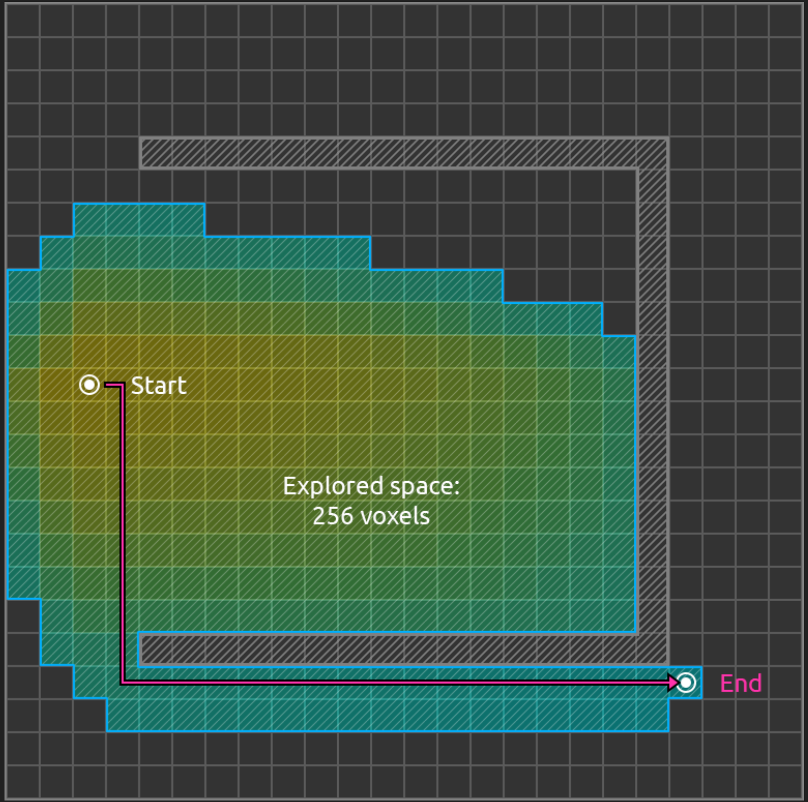

Papou
Norma


obstacles
start
end



https://qiao.github.io/PathFinding.js/visual/

A* Traps.



They are common.

Very common.

We never know how far

we need to explore

If there's a path,

we can finally stop.

If not,

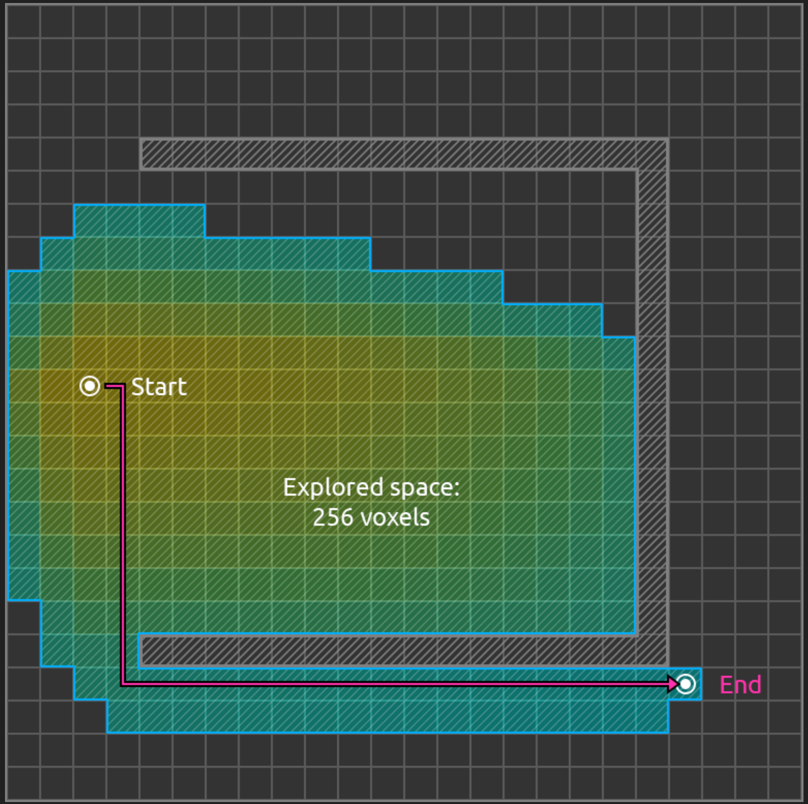
we explore the entire map

):

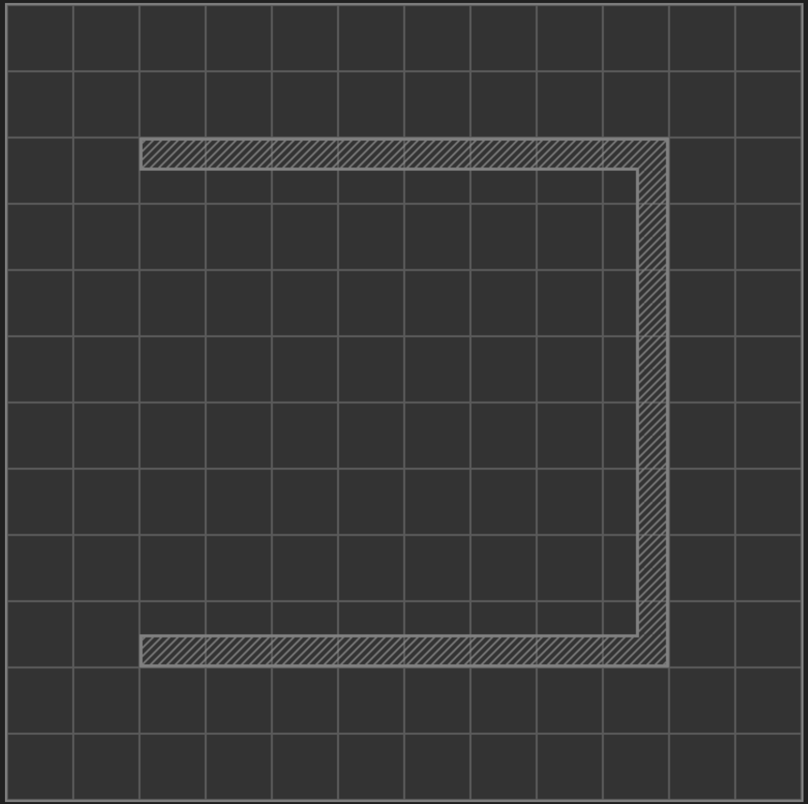
Maps with
1 million
walkable voxels

can take
minutes to
explore!

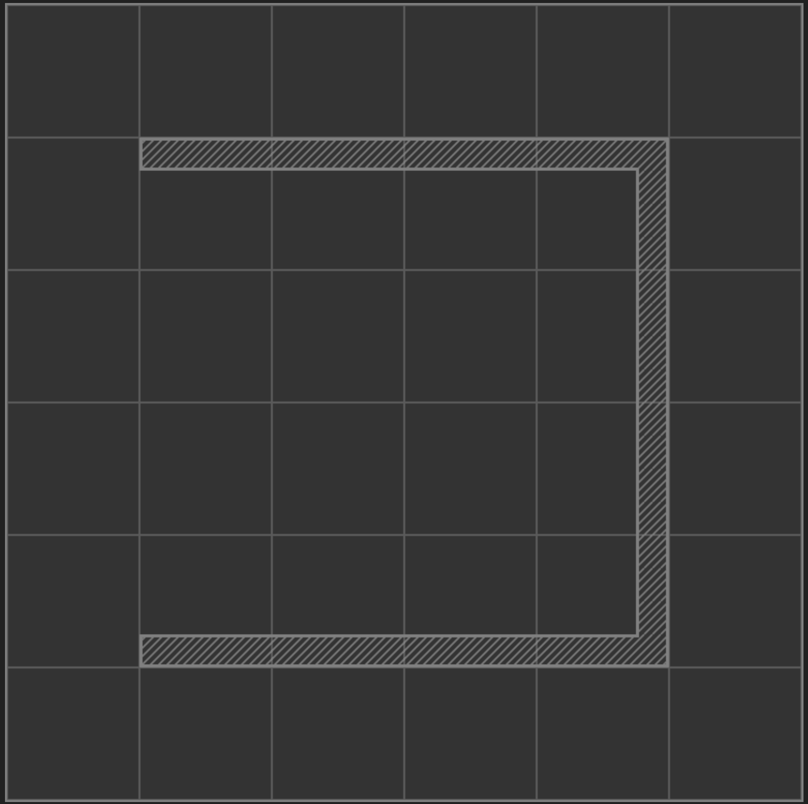
The problem with A* is the number of cells to explore.



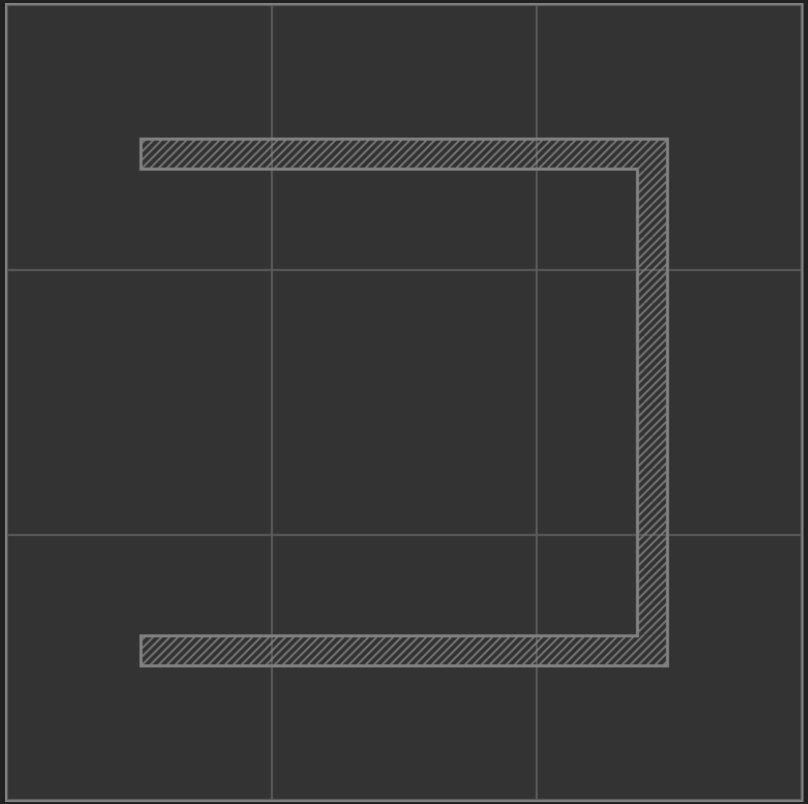
If we could combine cells...



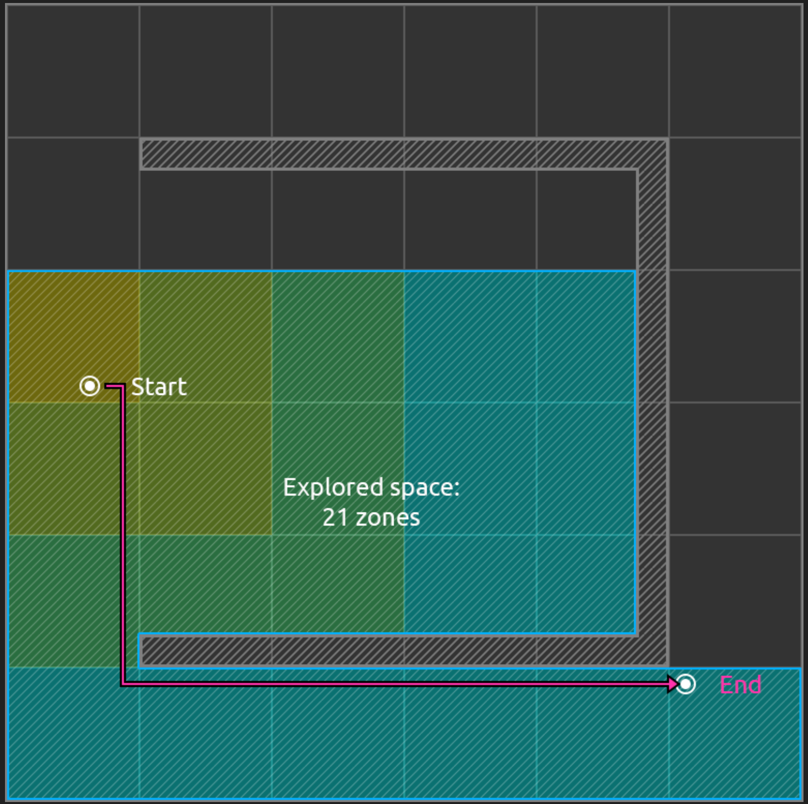
Into larger cells...



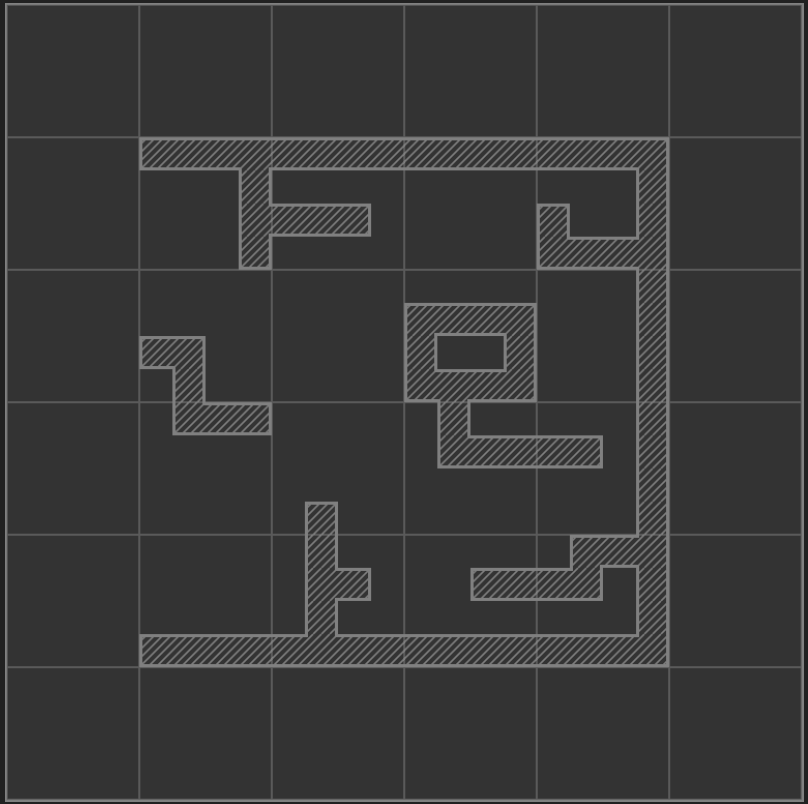
Until our world is simple...



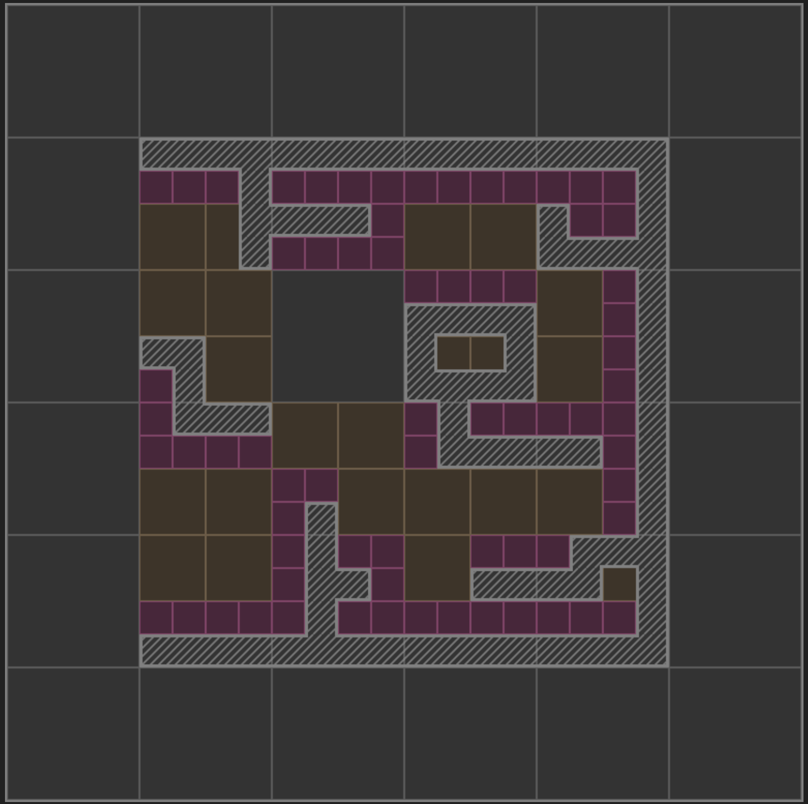
We could simply fill our dead ends.



If we add a bit of detail, however…



Regularity dissolves quickly.

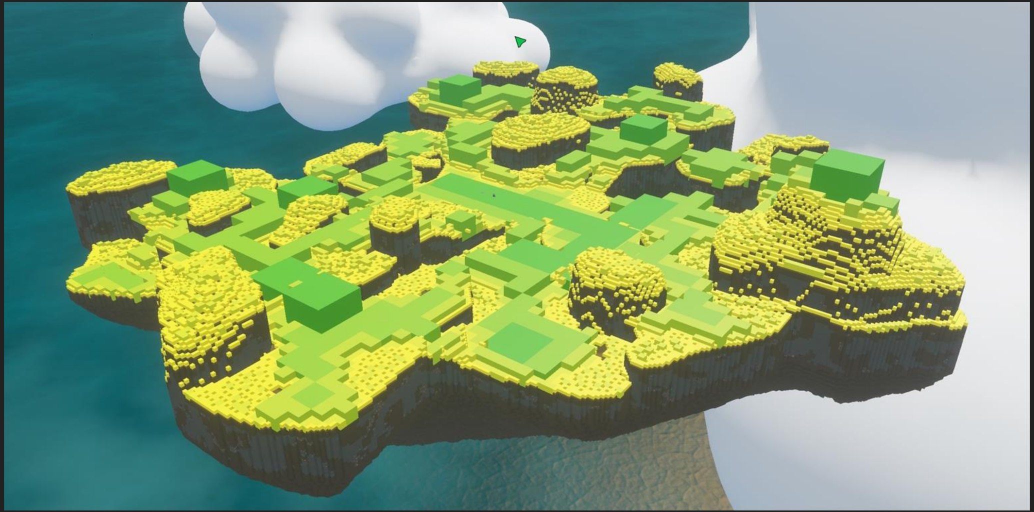


Benoit Alain, Lead Programmer
Sauropod Studio Inc.

When There Is No Path

Regular grouping isn't enough.

Just testing if a path Exists costs too much!

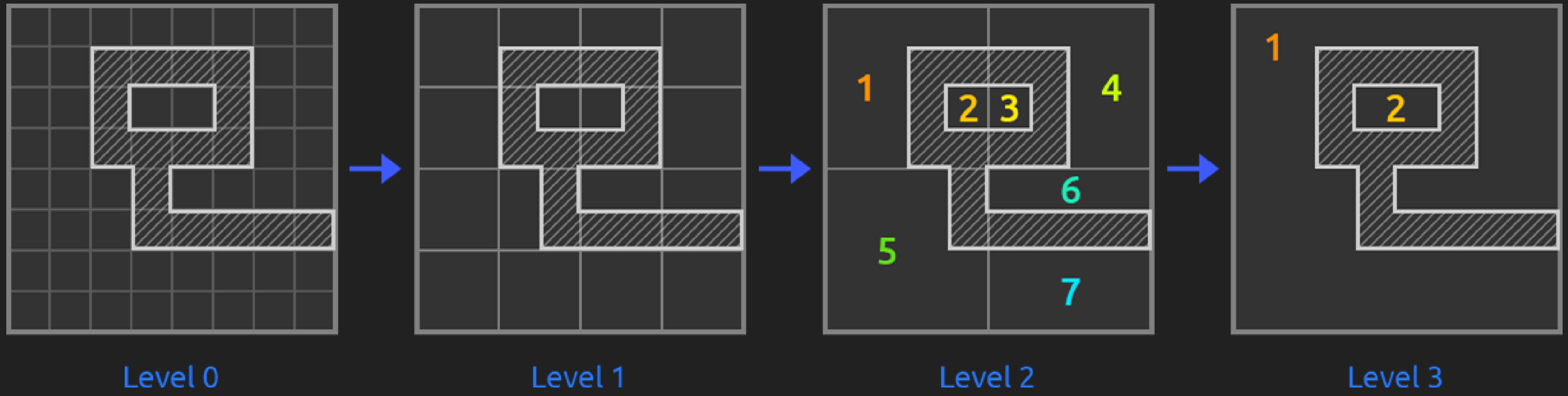


Let's start by solving
path existence…

Our Idea

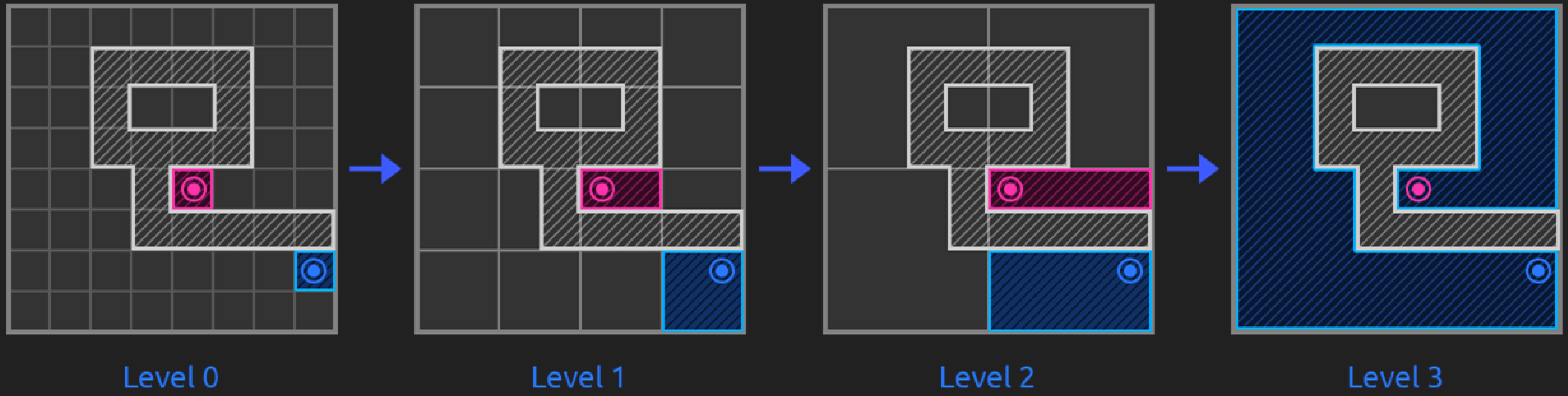
Group regions by Increasing Local Connectivity:

(merge regions if they can be connected without leaving their parent cell)



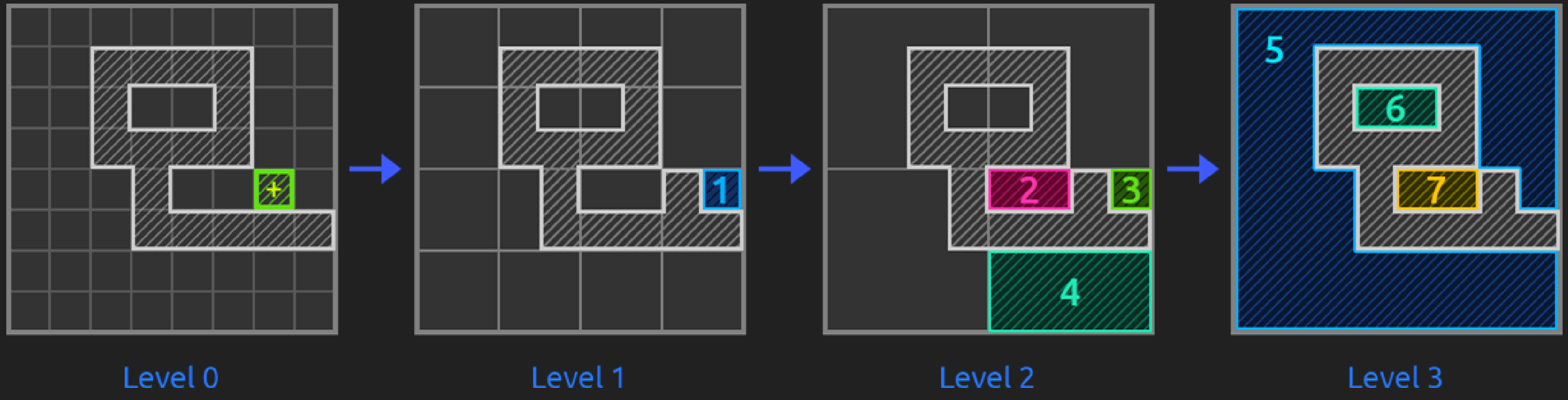
Path Existence

In this hierarchy, connected voxels always share a parent.



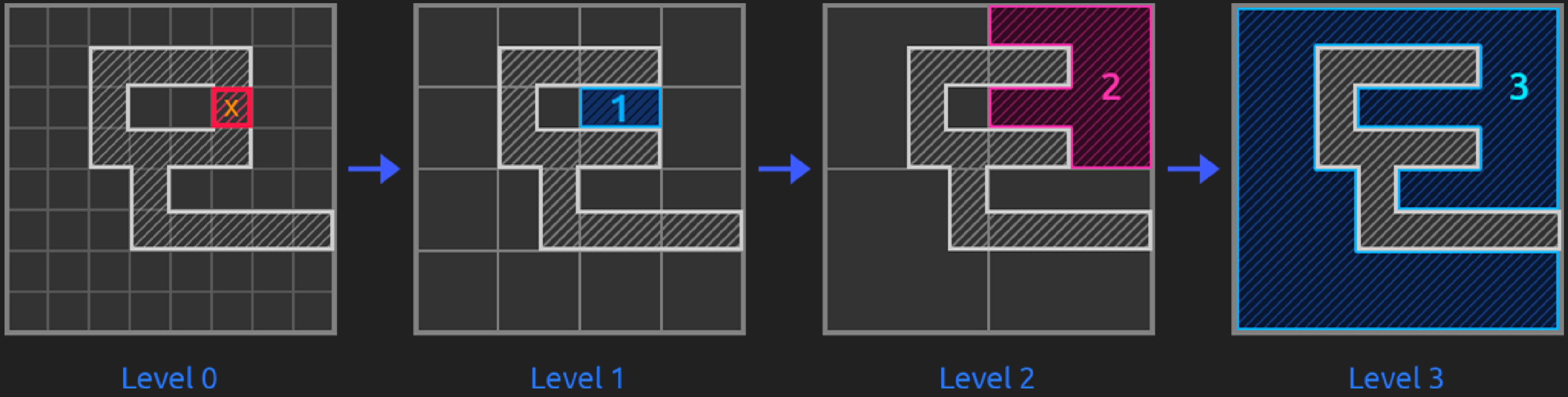
Terrain Modification

The number of regions to update is roughly constant at each level.



Terrain Modification

Local changes remain local throughout the hierarchy,
but their reach is exponential.



Benoit Alain, Lead Programmer
Sauropod Studio Inc.

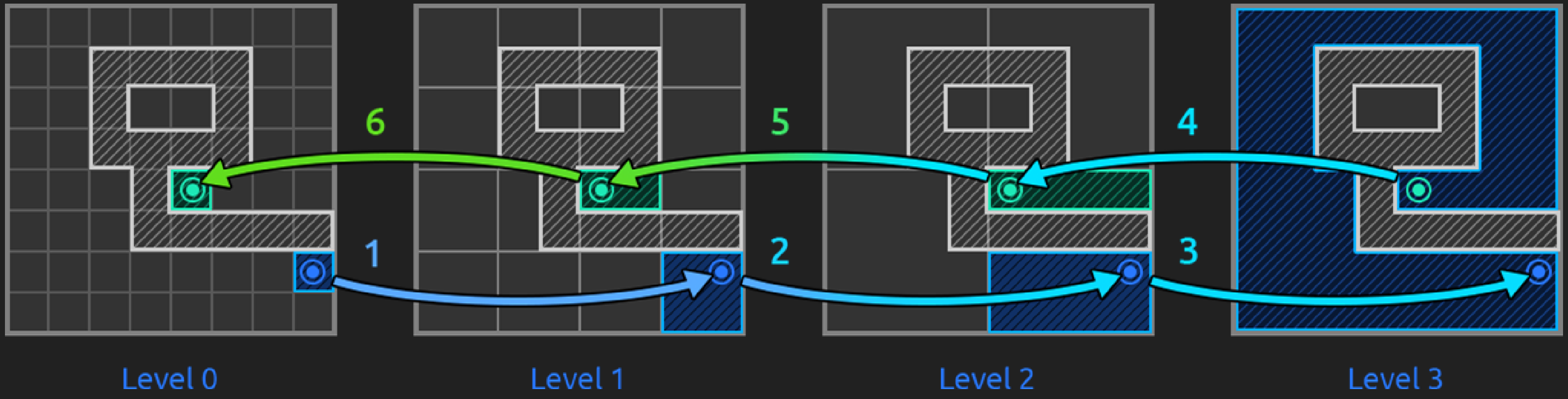
OK, *path existence*
can be solved efficiently.

Now can we find a *path*?

Yes

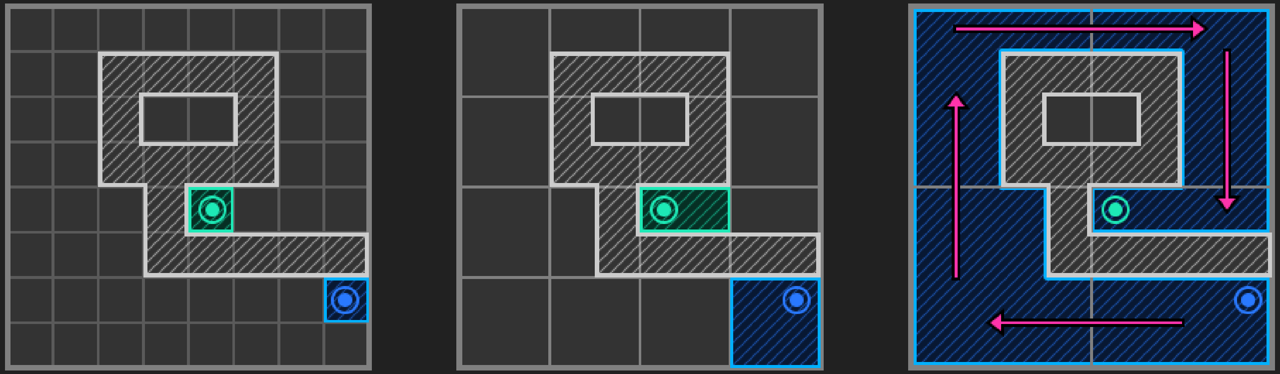
Building a Walkable Path

Find a common parent



Building a Walkable Path

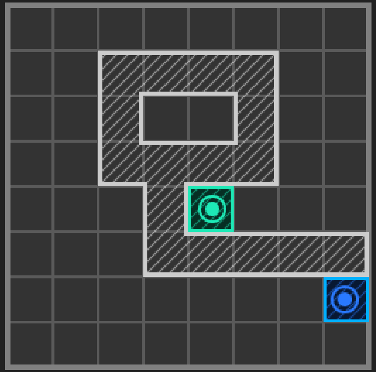
Refine the top layer



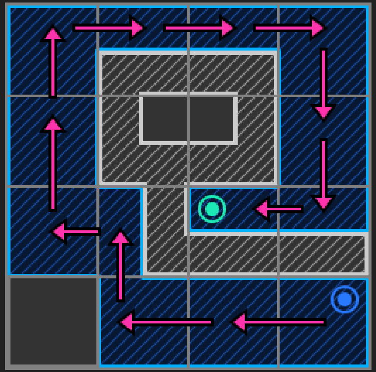
Level 0 | Level 1 | Level 2

Building a Walkable Path

Use existing path to refine even more



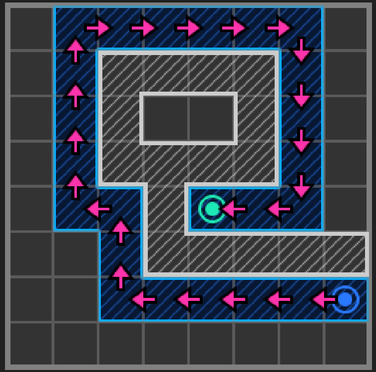
Level 0



Level 1

Building a Walkable Path

Until we have a complete walkable path

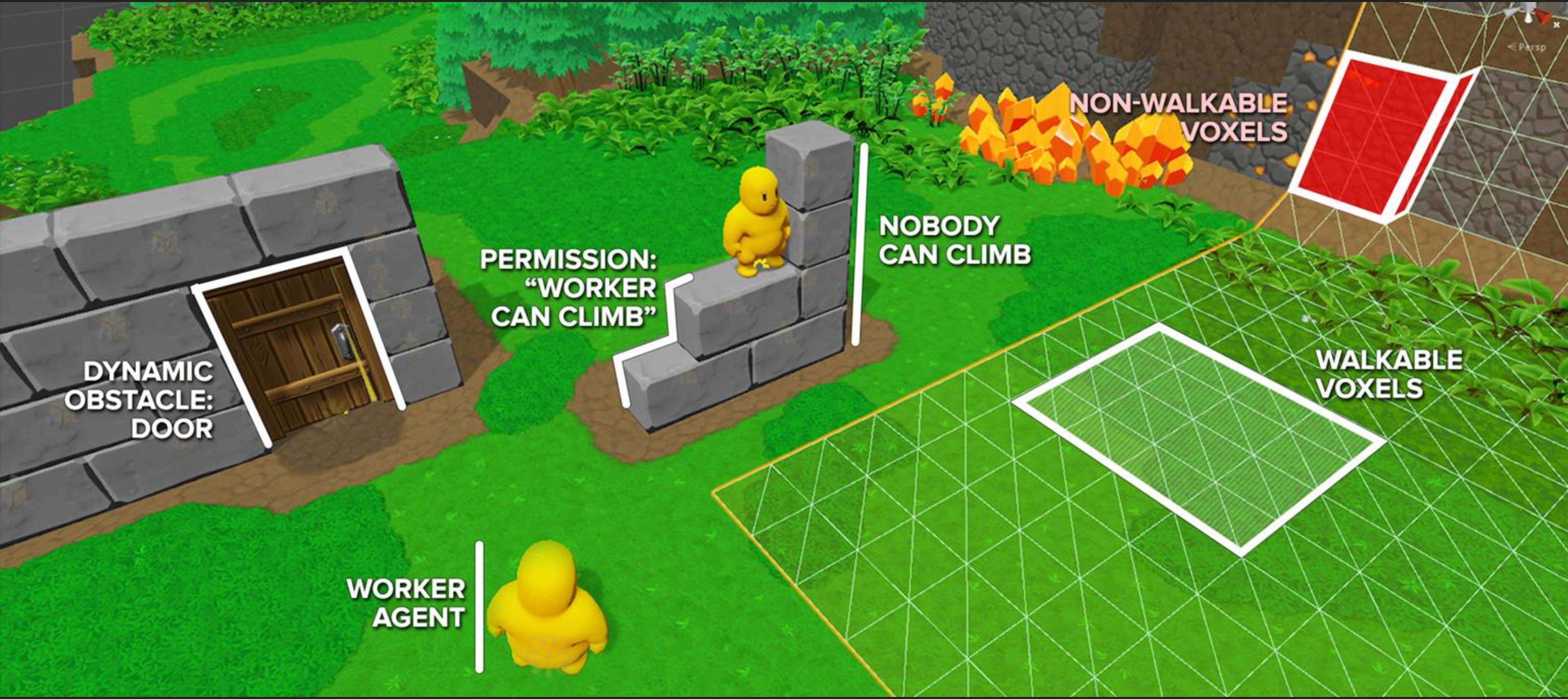


Level 0

Implementation

Implementation (Overview)

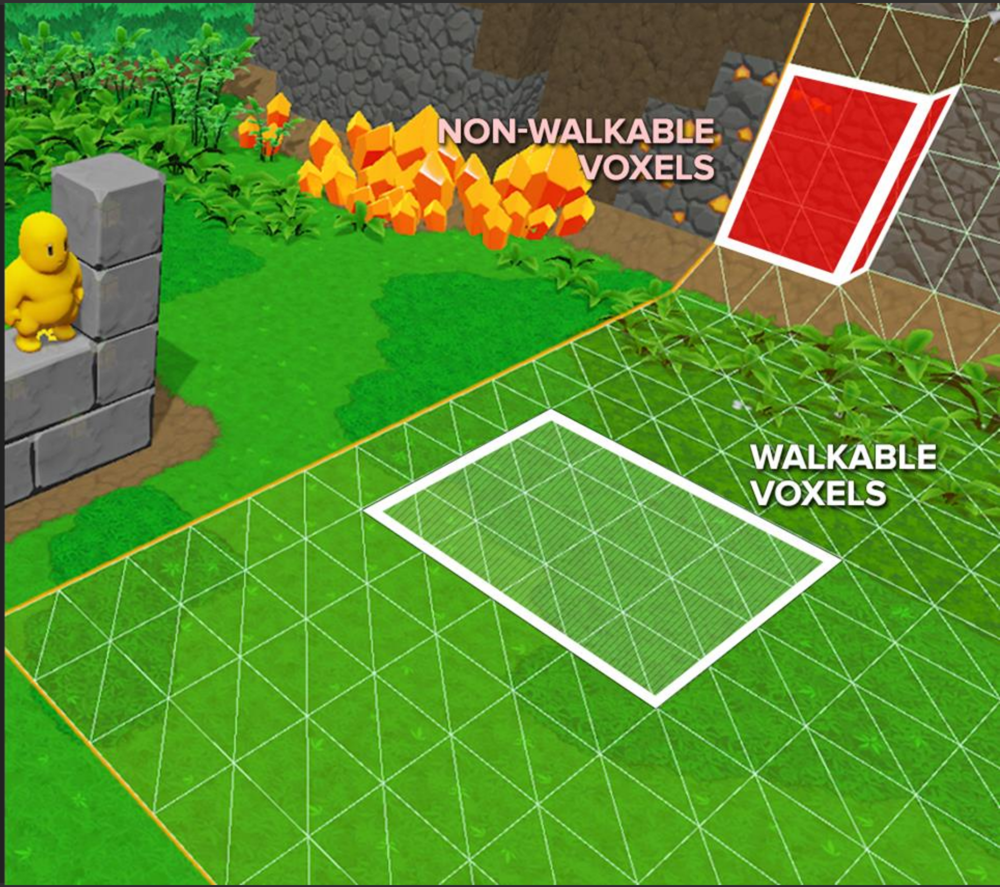
1. Building the hierarchy
2. Updating the hierarchy
3. Exploring the hierarchy


NON-WALKABLE VOXELS
NOBODY CAN CLIMB
PERMISSION: “WORKER CAN CLIMB”
DYNAMIC OBSTACLE: DOOR
WALKABLE VOXELS
WORKER AGENT

Building the Hierarchy

Step 1: Create a Node on each *walkable voxel.*

Walkable voxels are usually all Blocks and Terrain Voxels that have an exposed upper face.



Building the Hierarchy

Step 2: Build Neighbor connections

Note: Some connections can require special Permissions

- *Climb up*, *jump down*, etc.

Building the Hierarchy

Step 3: Divide into a
Larger Voxel Grid
(using XYZ position)

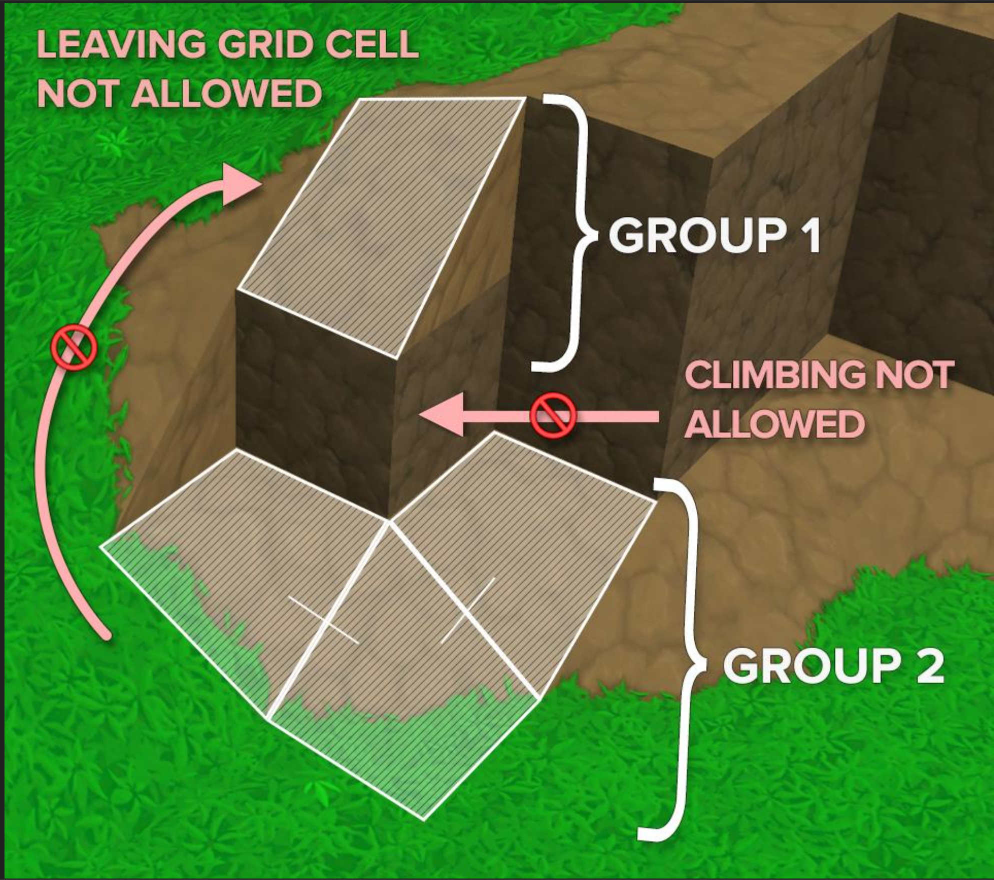
Building the Hierarchy

Step 4: Form Groups of locally interconnected voxels.

Create a Parent Node for each group.

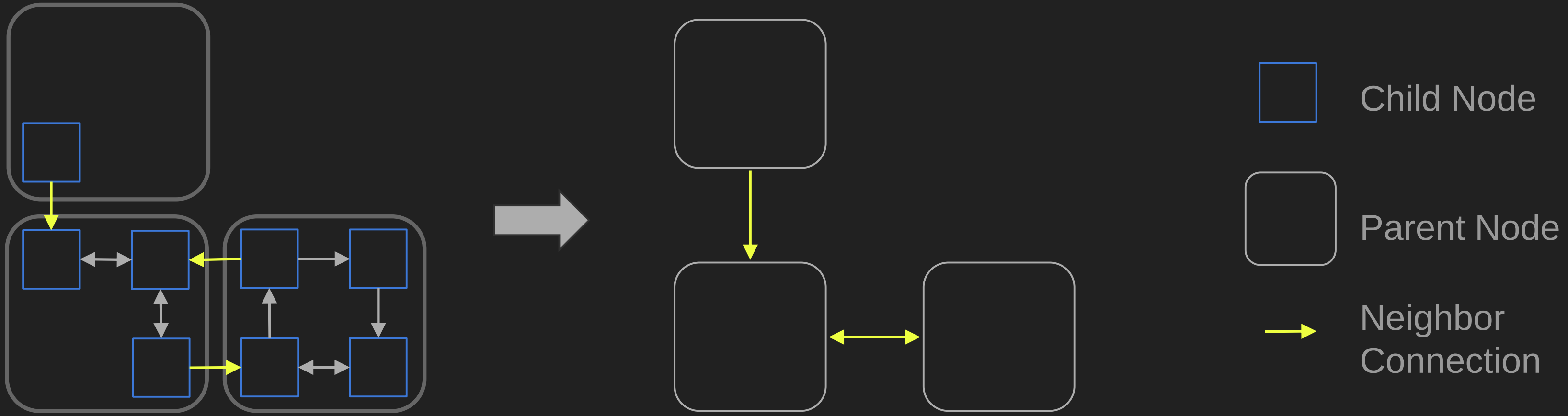
Implementation Tips:

- You can use *flood fill*
- Neighbor *permissions* shouldn't be mixed
- No need to be optimal; in doubt, be conservative



Building the Hierarchy

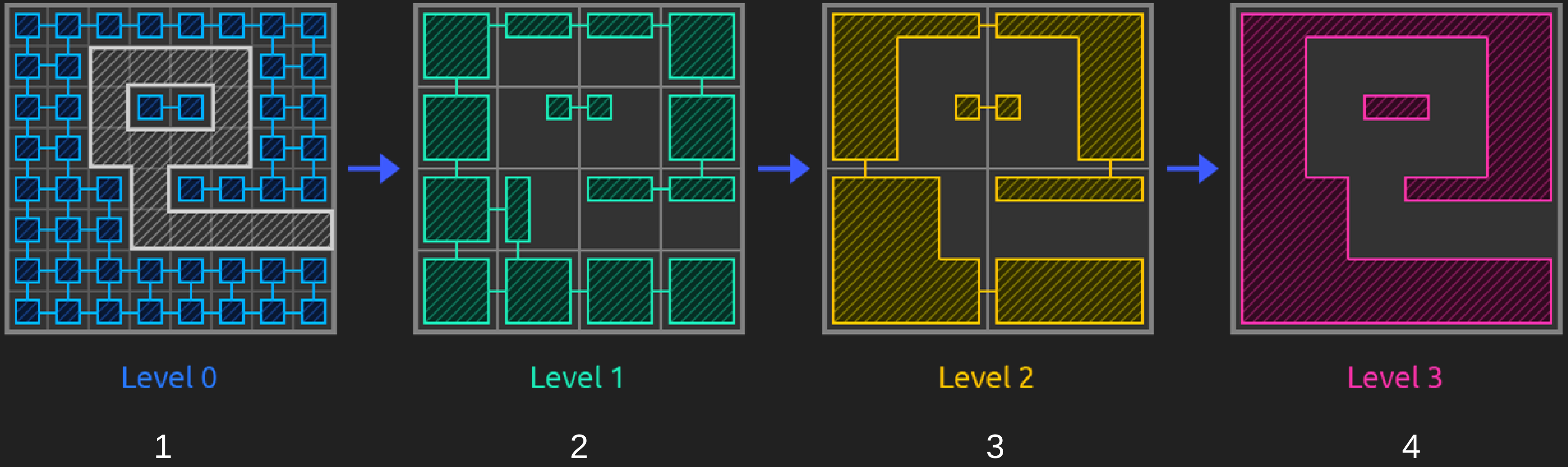
Step 5: Create Parent Neighbor Connections by combining children's connections that leave their group.



Note: neighbor connections aren't necessarily symmetrical.

Building the Hierarchy

Step 6: Form groups of *parent nodes* until you have L levels.



Updating the Hierarchy

Benoit Alain, Lead Programmer
Sauropod Studio Inc.

Updating the Hierarchy

Step 1: Identify the Altered voxels

- *Walkability* changed
- New neighbor *connections*

Create new *nodes* and Destroy old ones.

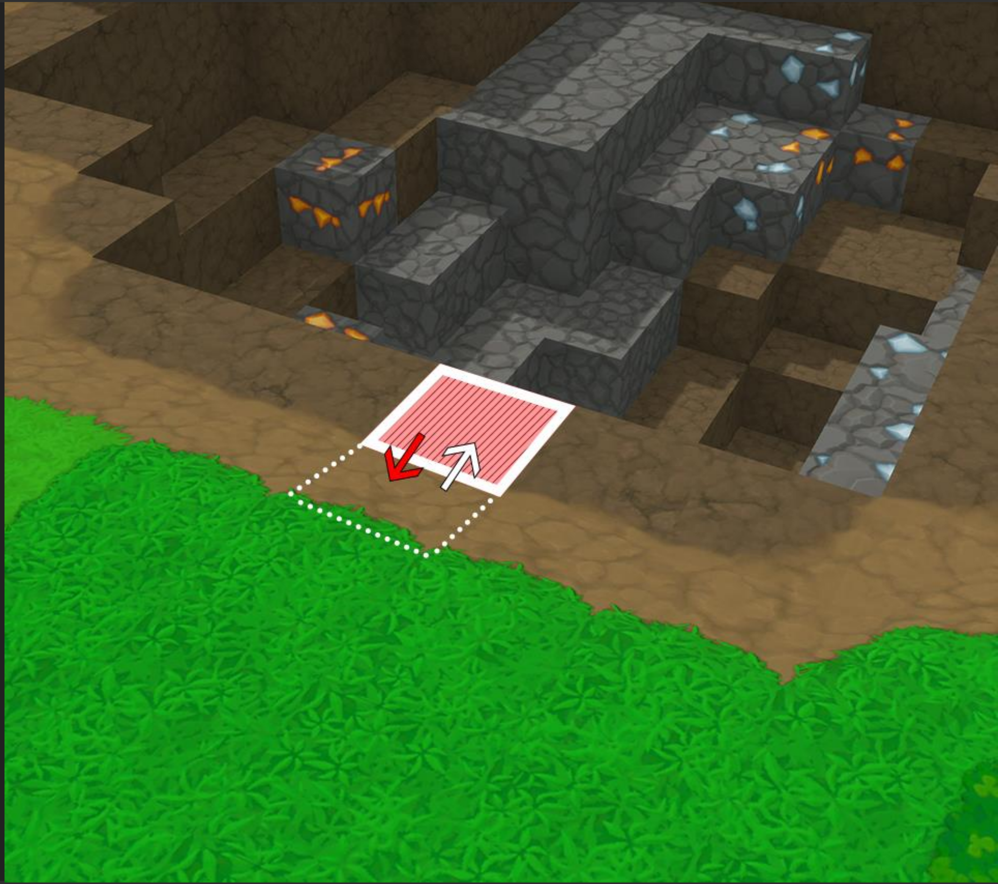
Updating the Hierarchy

Don't count Dynamic Obstacles
for now

- Loose blocks
- Collapsed structures

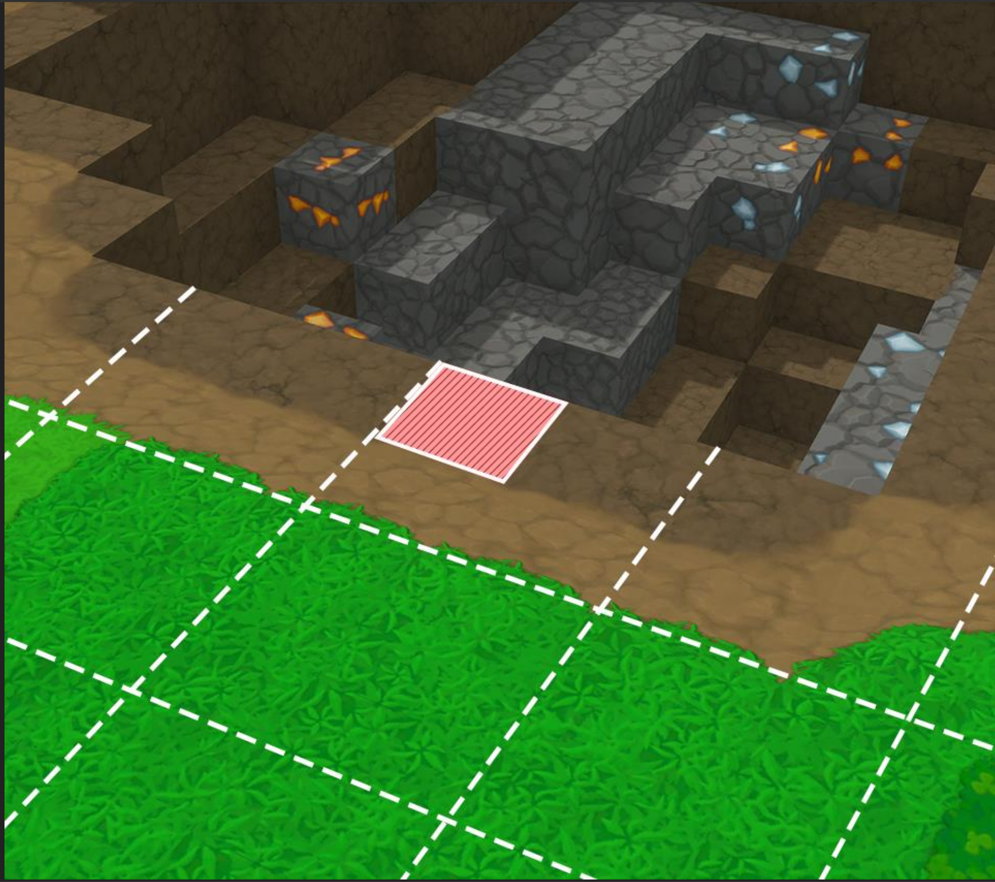
Updating the Hierarchy

Step 2:
Rebuild neighbor connections
To and From altered voxels.



Updating the Hierarchy

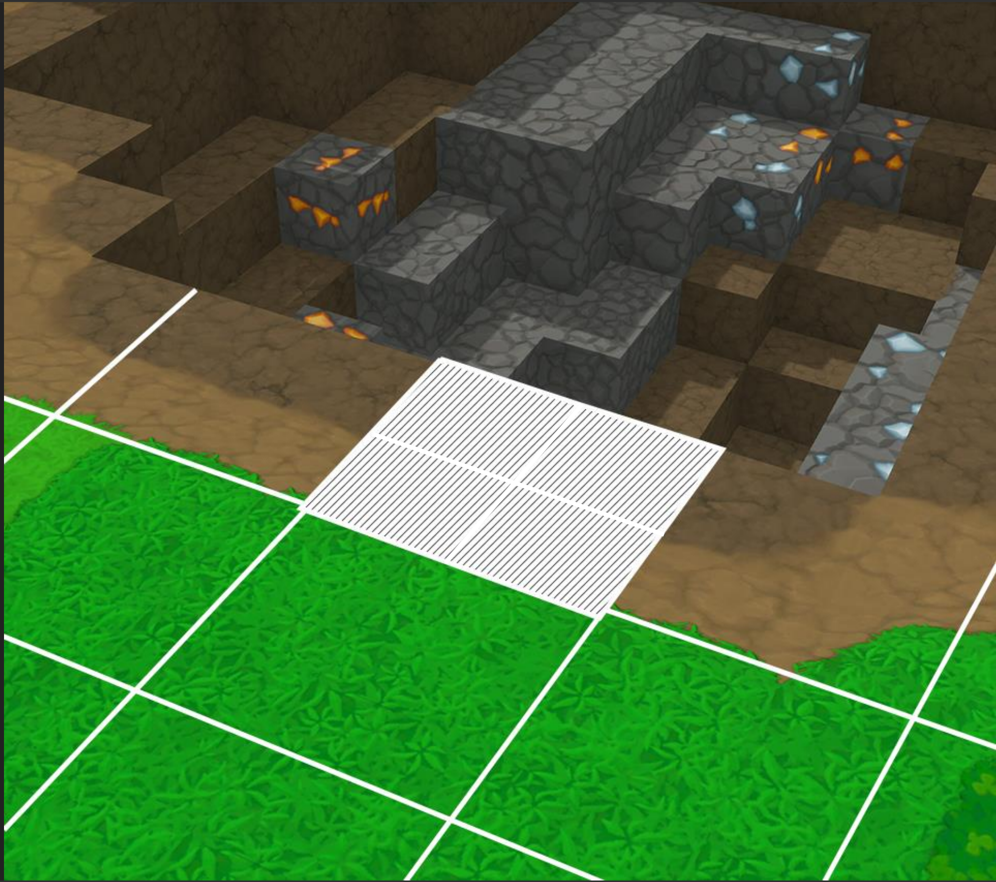
Step 3:
Map altered nodes on the
Larger Voxel Grid.



Updating the Hierarchy

Step 4: Form new Groups.

Create new Parent Nodes.

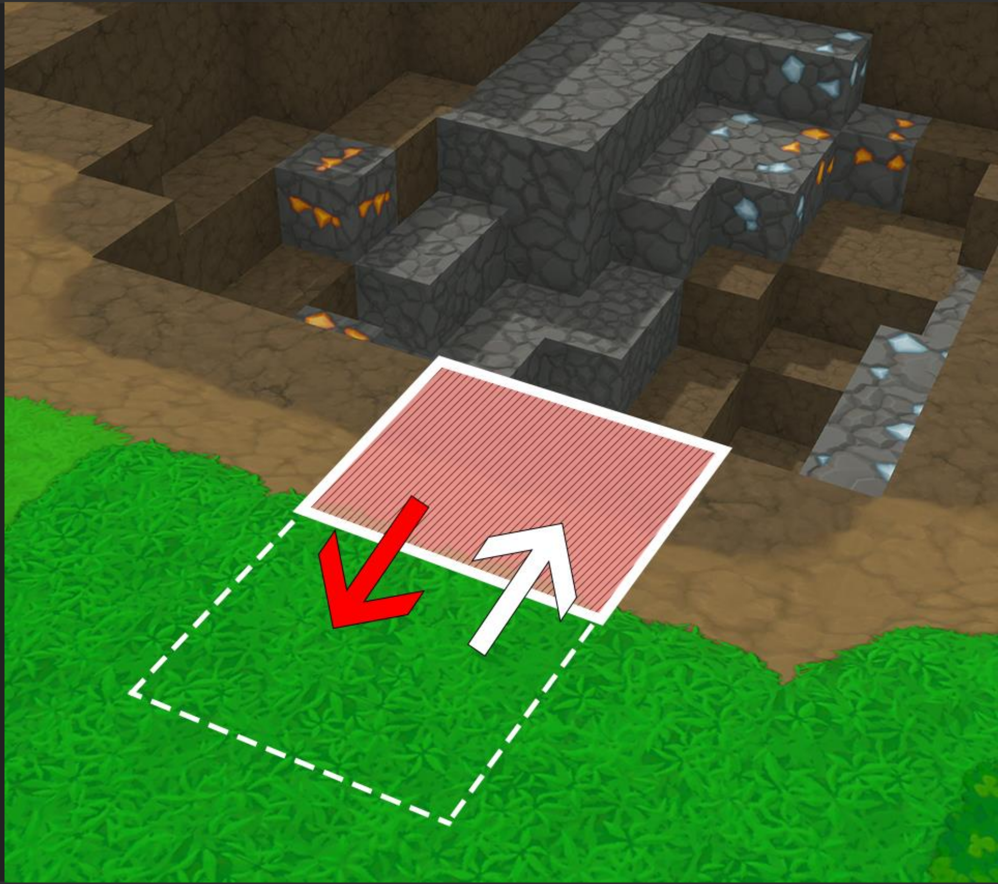


Updating the Hierarchy

Step 5: Rebuild Parent-Level connections to and from altered nodes.

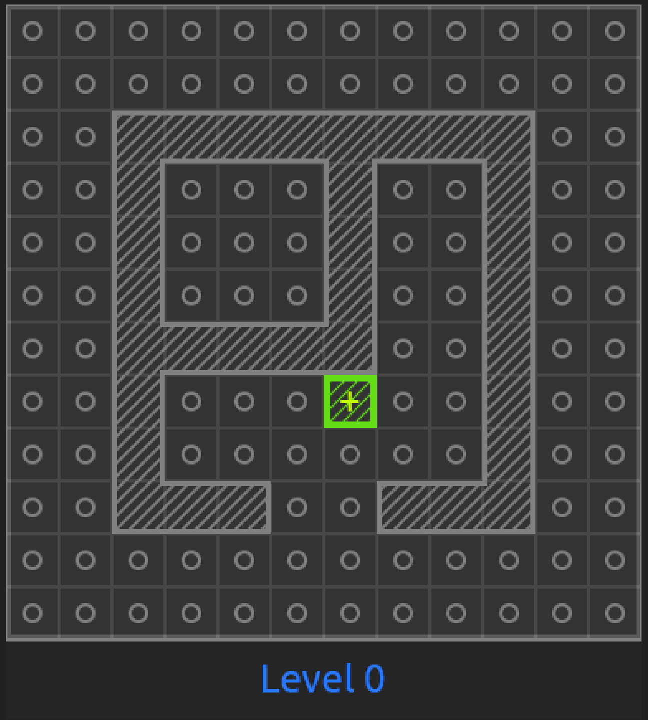
Implementation tips:

- Use a table of *grid position* to *parent nodes* to find potential parent neighbors

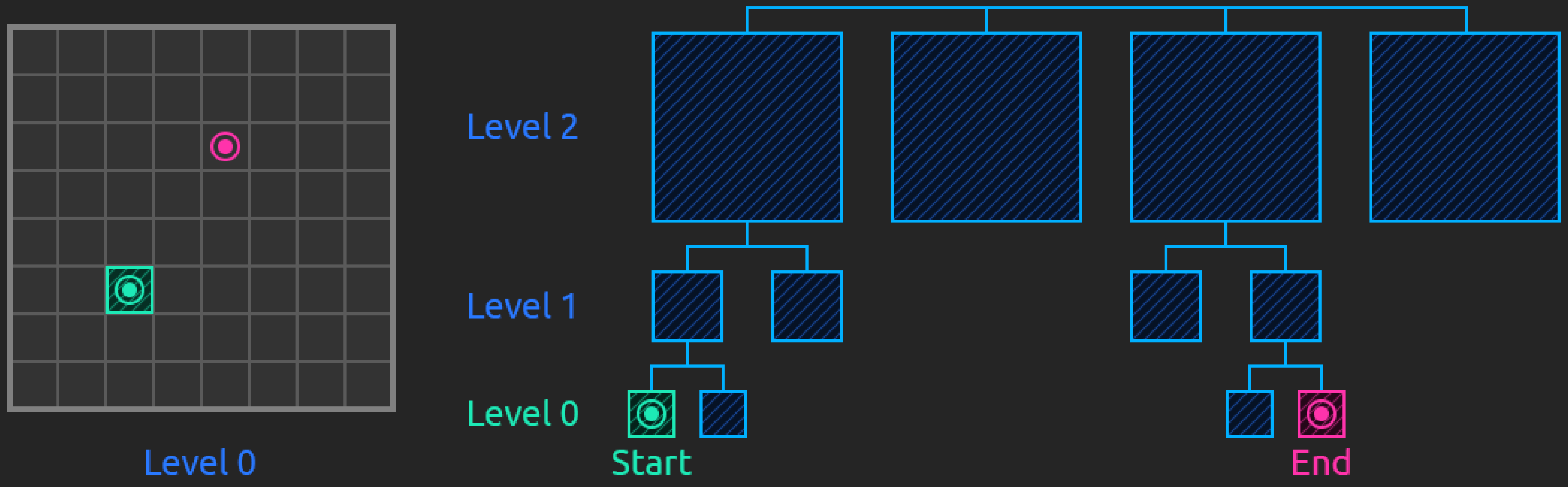


Updating the Hierarchy

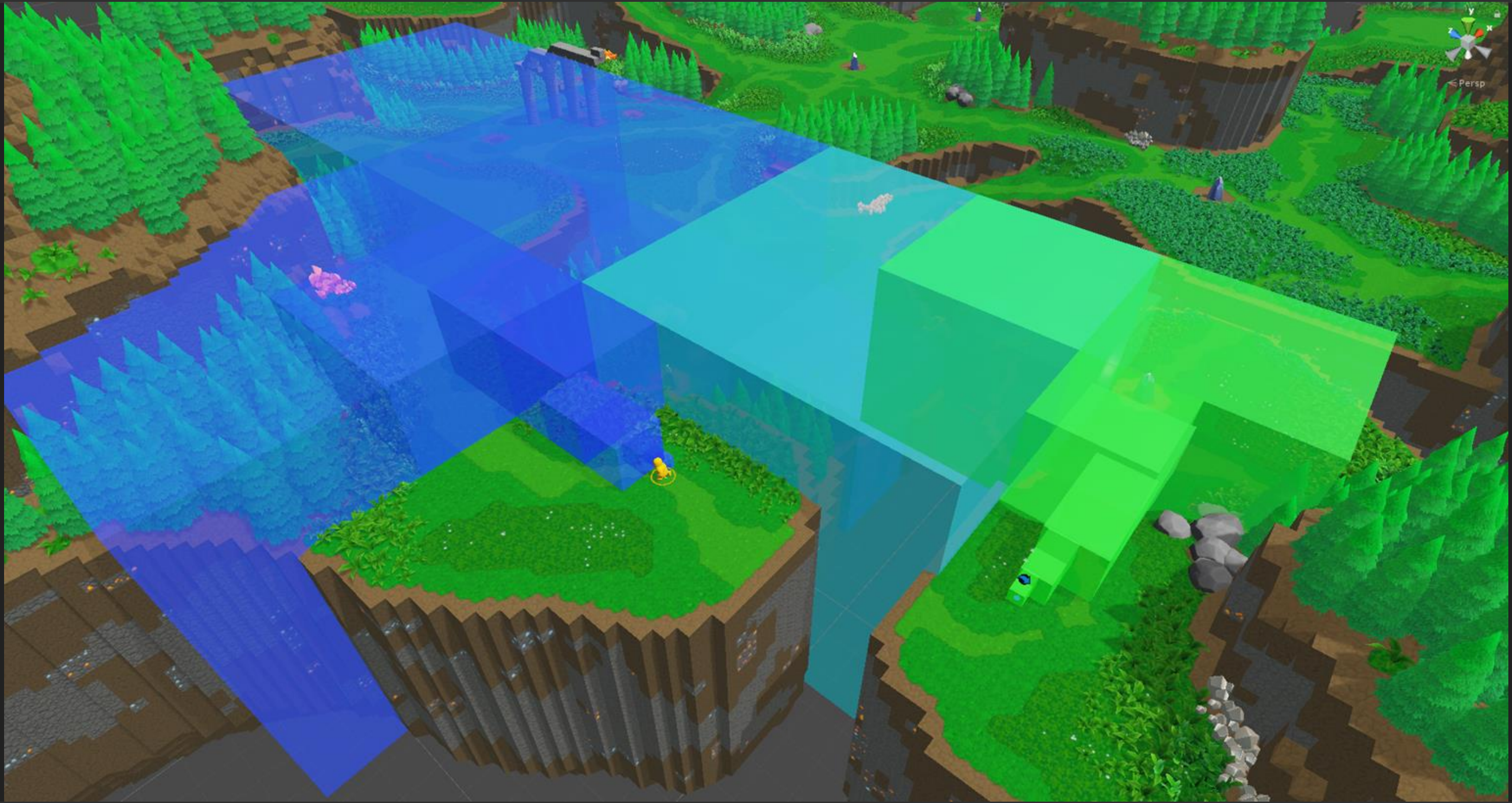
Step 6: Propagate changes in *parent nodes* until you have L levels.



Hierarchy Overview

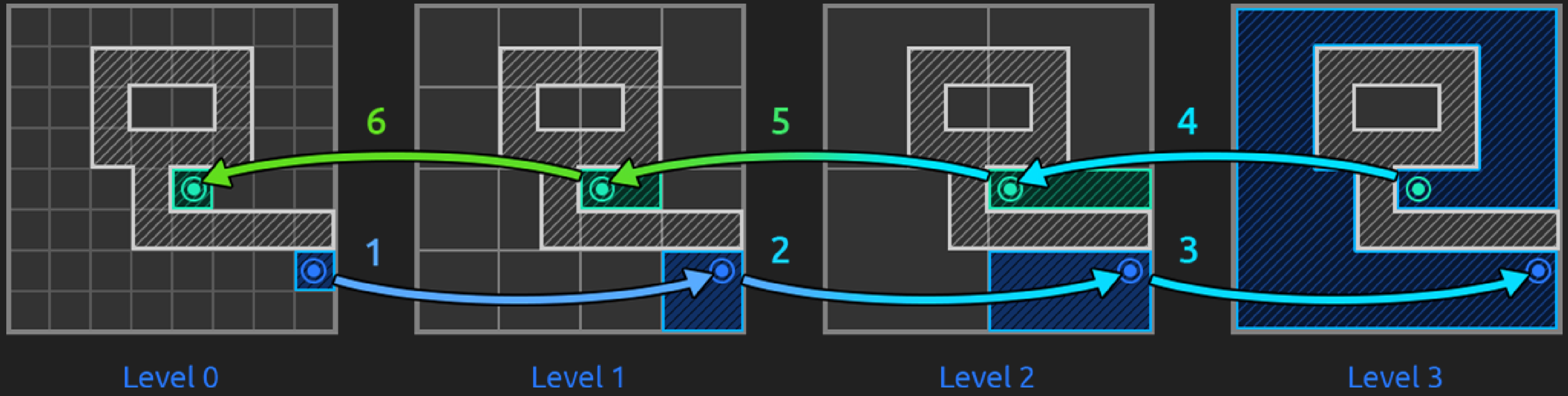


Hierarchical Pathfinding


Persp

Building a Walkable Path

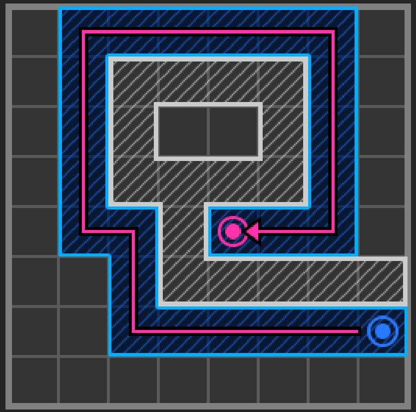
We know the *common parent* method:



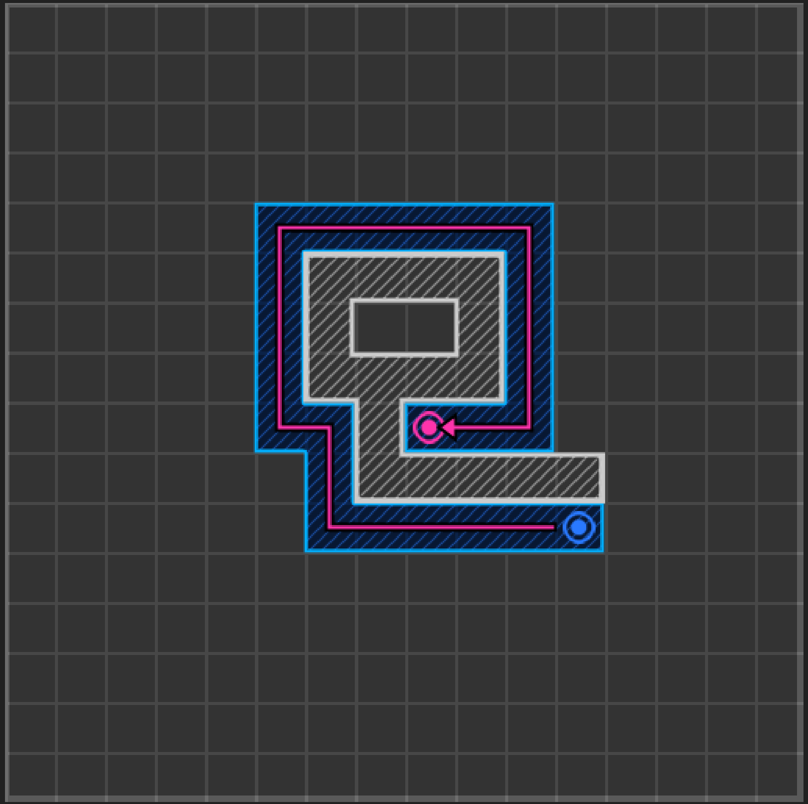
Building a Walkable Path

We can refine recursively to get a plausible path.

However, we never leave the Common Parent.

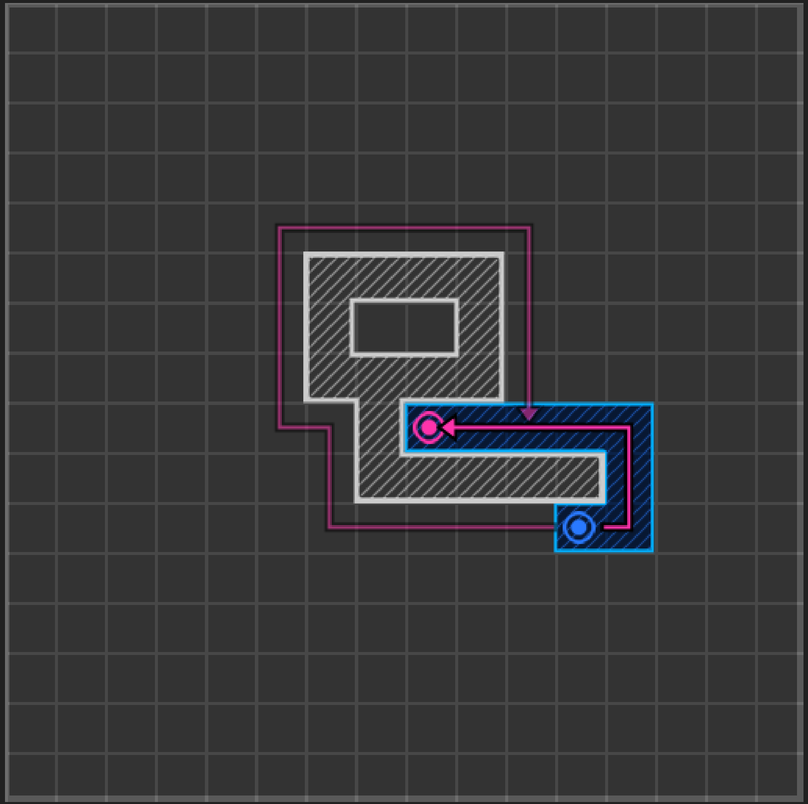


Are we sure we didn't miss the big picture?



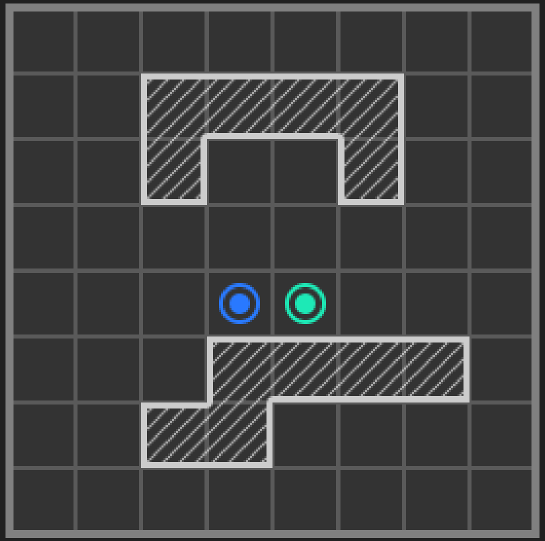
Not all paths are optimal.

):

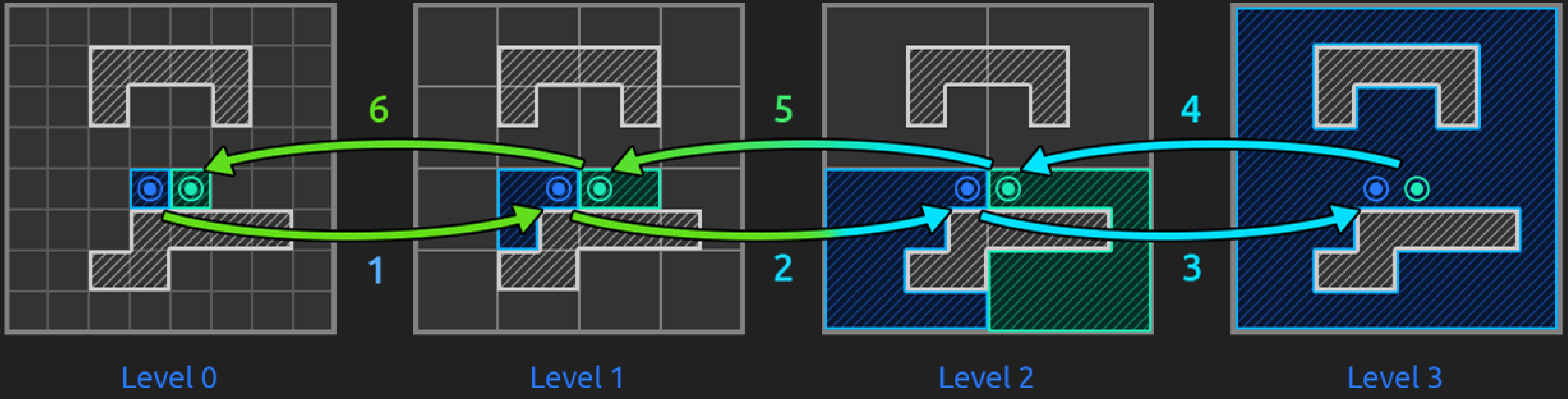


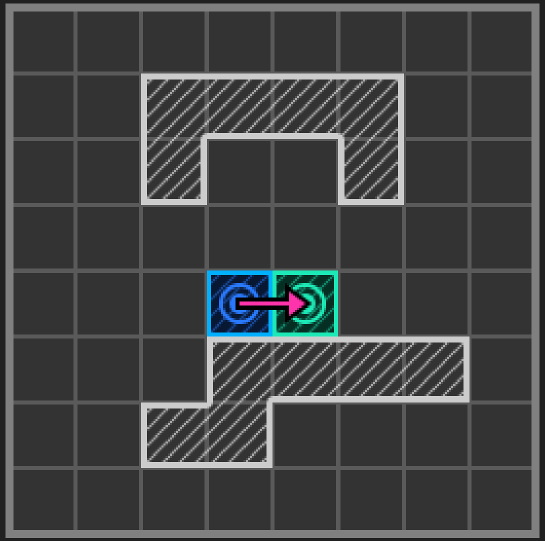
We need some rules

Rule #1: Stop Early



?


6
5
4
1
2
3
Level 0
Level 1
Level 2
Level 3

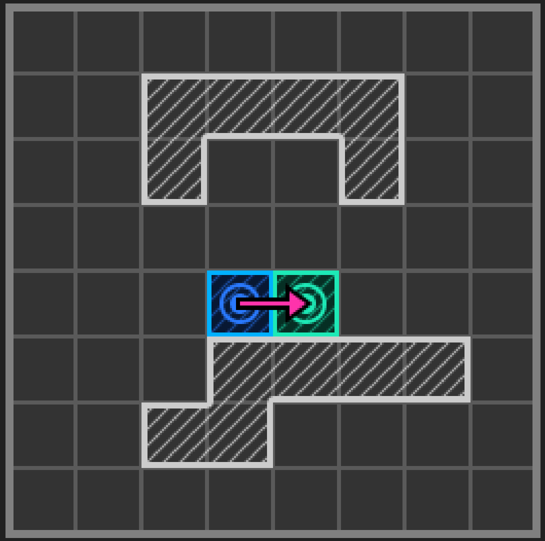


Rule #1: Stop Early

Explore Lower hierarchy levels
as you get closer to the Goal.

(Nodes should be smaller than the *goal distance*)

If there are multiple *goals*, use the
Minimal Distance as much as possible.



Rule #2: Be Pessimistic

(No false positives)

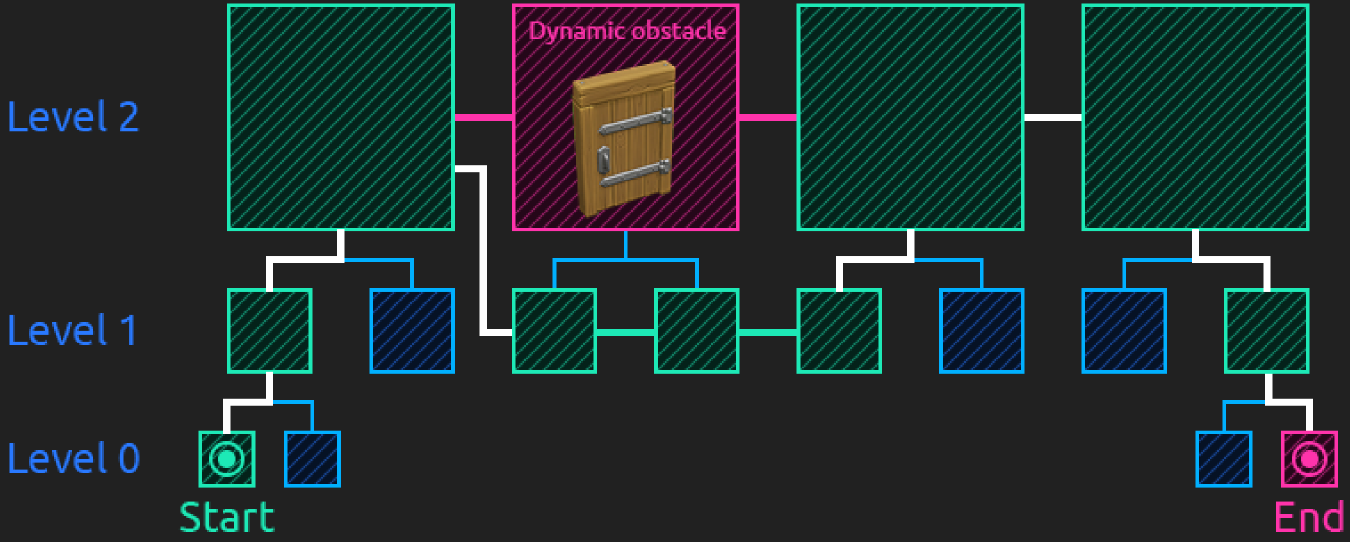

DYNAMIC OBSTACLE
BARREL
DYNAMIC OBSTACLE
DOOR

Rule #2: Be Pessimistic

Explore Lower hierarchy levels if they contain Dynamic Obstacle blockers

Implementation Tips:

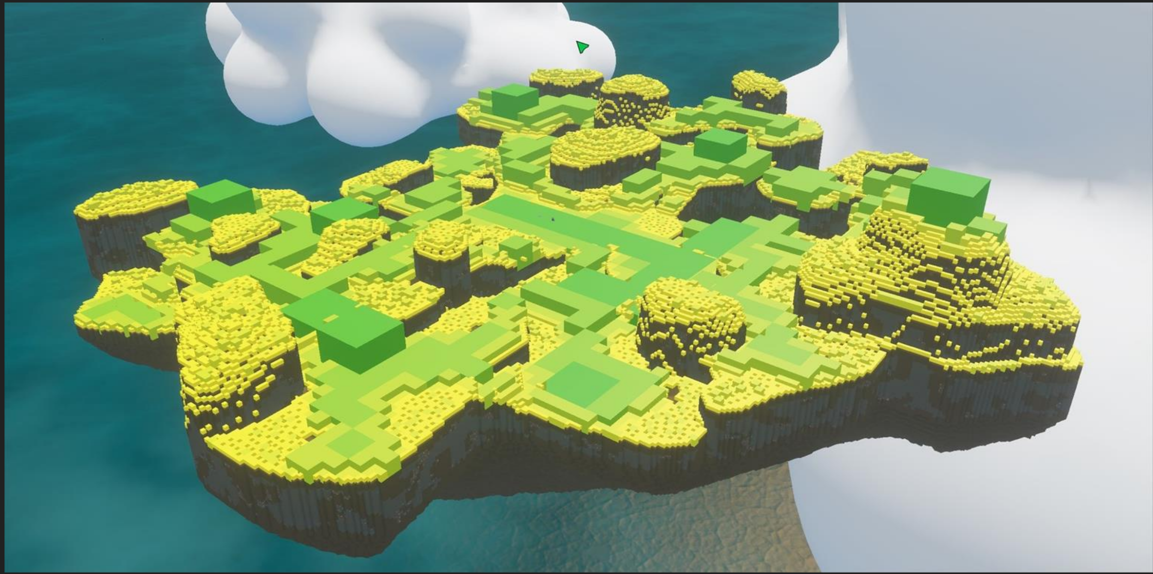
- Use an *obstacle count* variable for an early out
- Use a virtual method on each *dynamic obstacle* to decide if they allow passage to a given pathfinding request

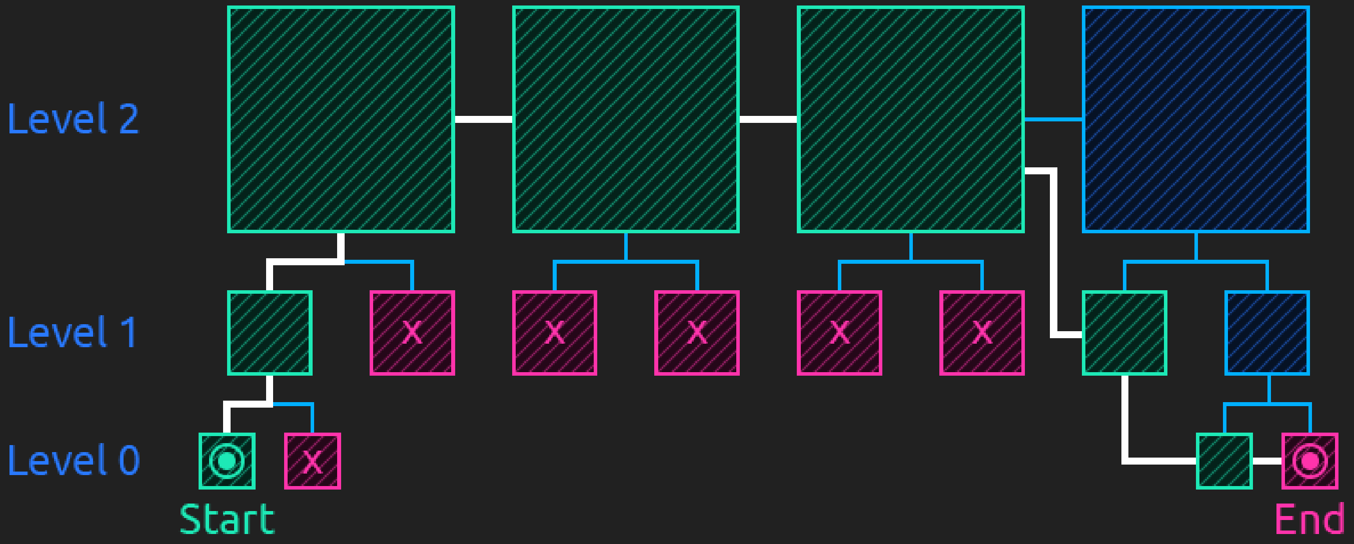

Level 2
Level 1
Level 0
Dynamic obstacle
Start
End

Rule #3: Converge Quickly

Rule #3: Converge Quickly

Don't explore Child Nodes
if you have already
explored any of its *parents*.




Level 2
Level 1
Level 0
X
X
X
X
X
X
Start
End

Oh, One More Thing

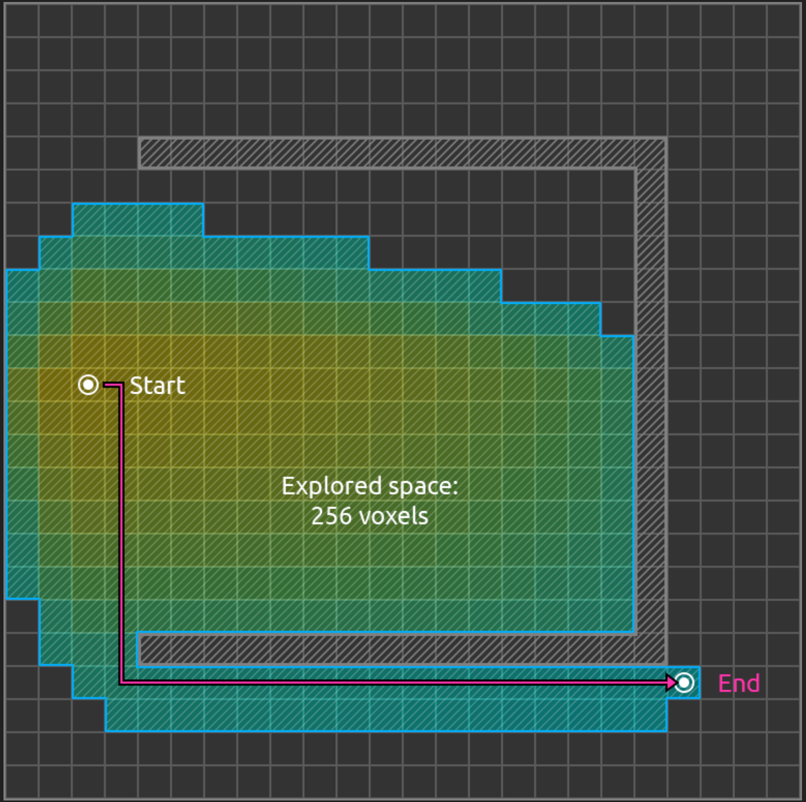
Path Prioritization

We're not just finding a *common parent* anymore.

We'd like to explore paths that go Towards the Goal first...

Our own version of

Let's use A*!



A* Algorithm (Pseudocode)

```
Queue.Insert(Source)
While (!Queue.IsEmpty())
    CurrentPath = Queue.PopPriority()
    If (Succeeds(CurrentPath))
        Return CurrentPath
    ForEach (SubPath In AvailableNeighbors(CurrentPath))
        Queue.Insert(SubPath)
Return Null
```

Parent, Neighbors, Children

Penalty Function

Nodes in the *priority queue* are ordered from a Penalty Function (lowest first)

Penalty =

k_A * CurrentPath.Length + ← Short paths

k_B * Distance(CurrentPath, Destination) + ← Towards the goal

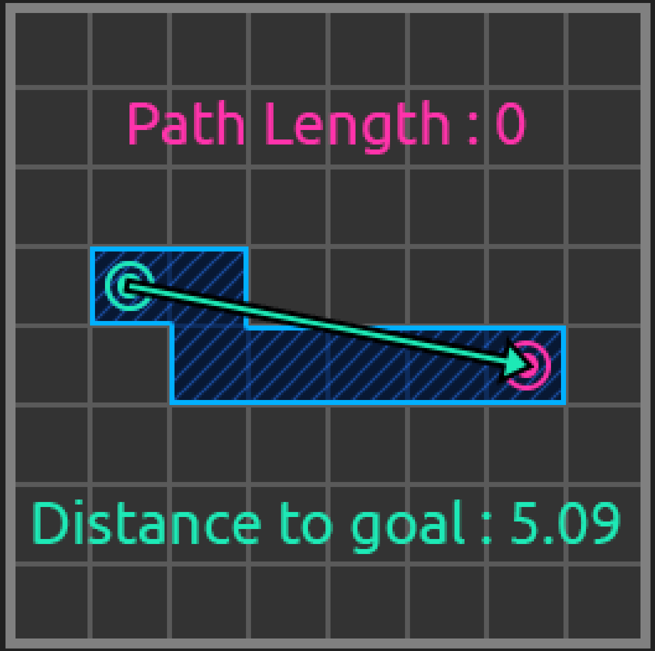
GameplayPenalty(CurrentPath)

Penalty Function

Priority queue penalty function requires
Path Length and Distance to Goal

(use *distance = 0* or *minimal distance*
if there are multiple goals)

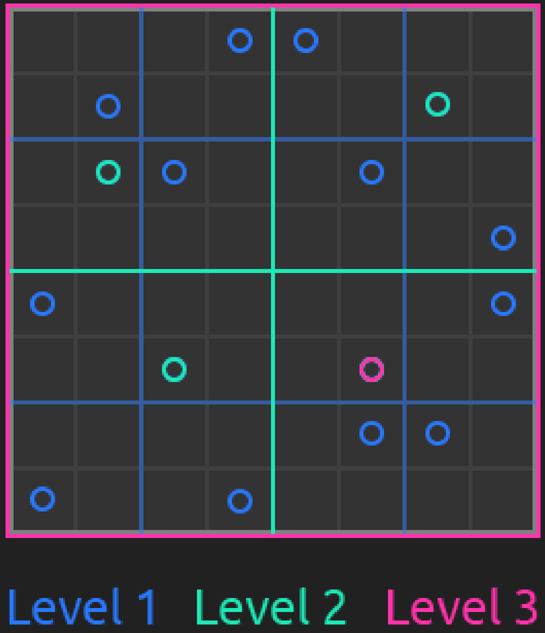
Can we generalize this to our *hierarchy*?



Representative Child

Idea: when building the node hierarchy, select a Representative Child for each Parent Node.

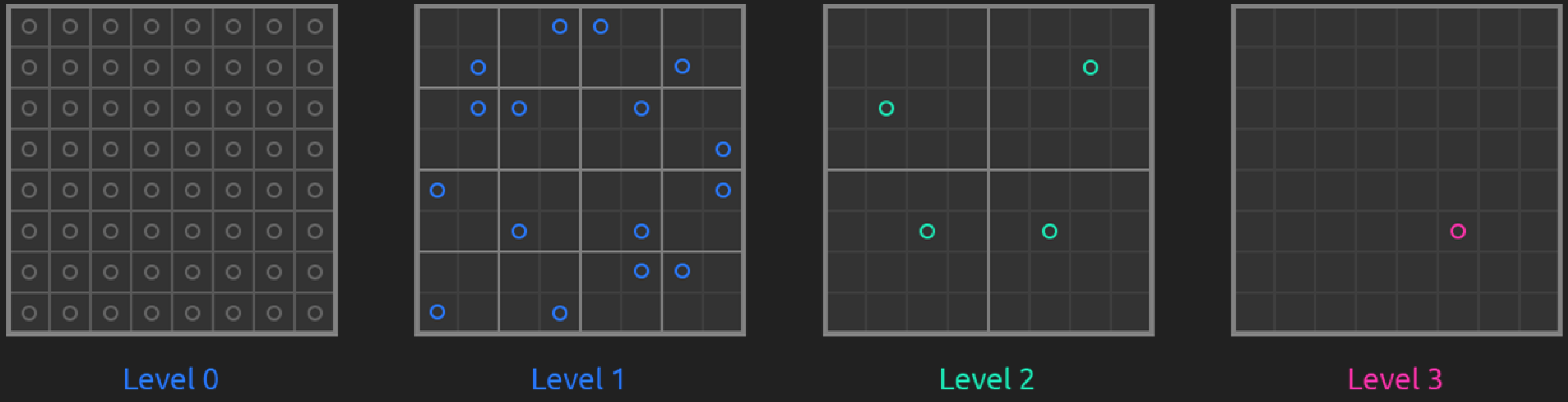
Recursively, Representative Children lead to a Representative Voxel.



Representative Child

Representative voxels can be used to estimate distance relations between nodes.

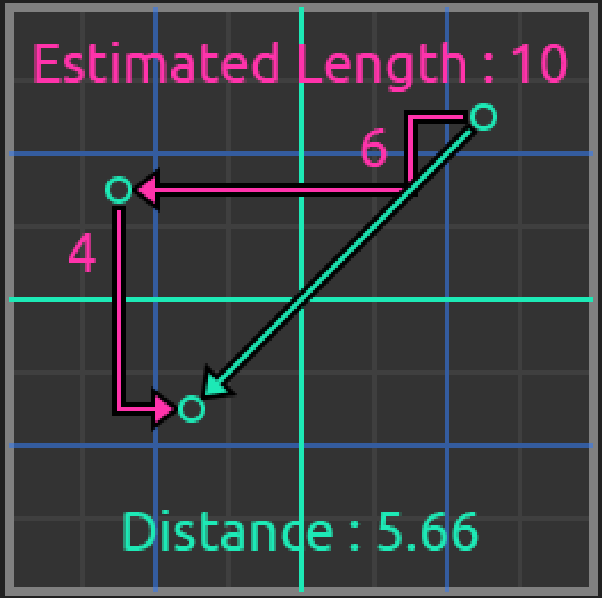
For best results, they should be close to the Center of the node.



Parent Nodes

Connection Weight =
Sum { Connections Weights between Representatives Children }

Distance to Goal =
Distance between Goal and Representative Voxel

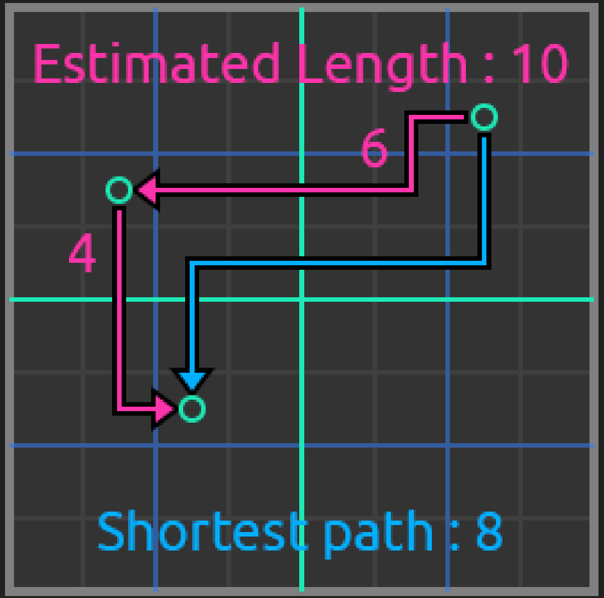


Path Optimisation

Base connection weights are Fine Tuned.

Parent connection weights are Approximations.

How optimal are *hierarchical pathfinding* results?

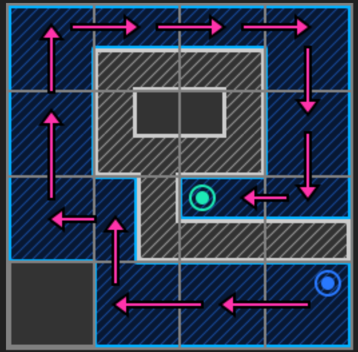


Path Refinement

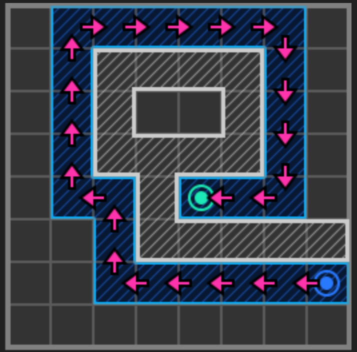
Important Note:

Path Refinement doesn't try to connect representative children!!

It just finds the Shortest subpath from node A to any child of node B



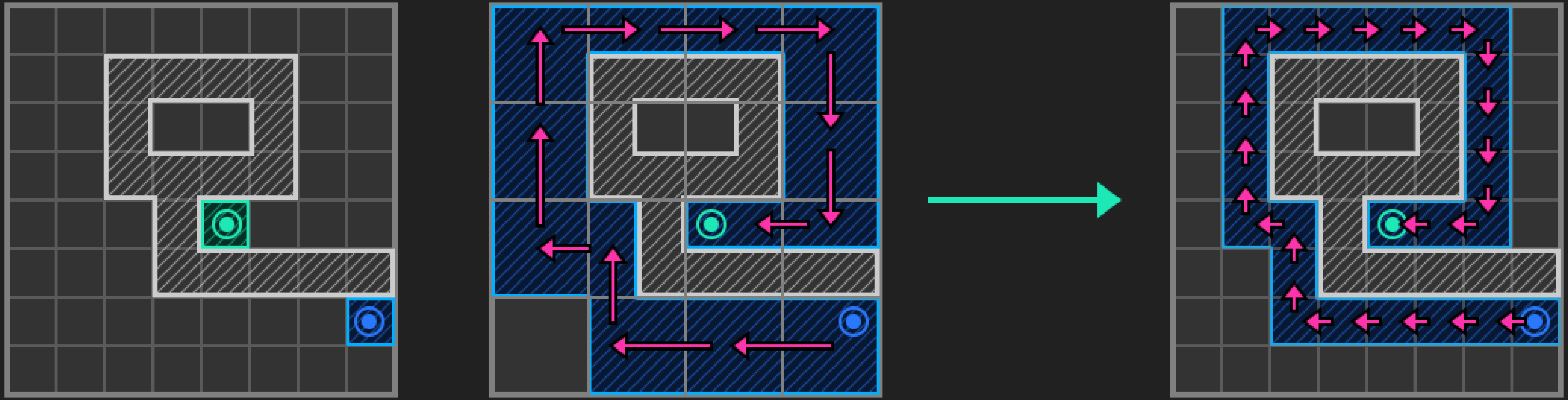
Level 1



Level 0

Path Refinement

Fine-tuned connection weights are used at the end of Path Refinement.



Path Optimality

Not an exact science!

Path Error =

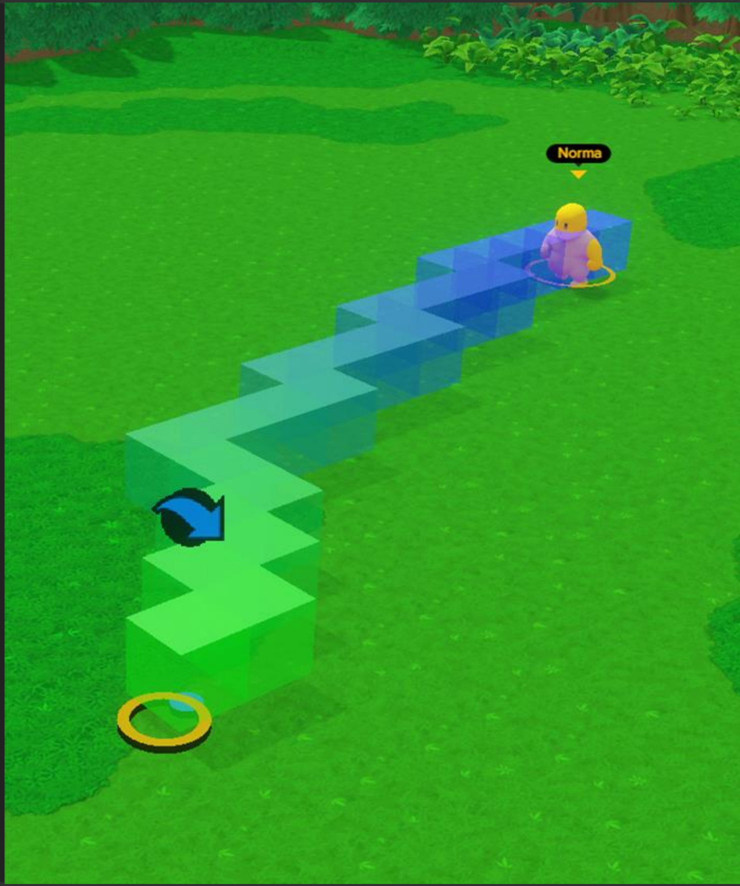
Hierarchical Path Length (approximation) -

Refined Path Length (precise)

If our path is Important (and short),

we can do a new pathfinding request

with a lower hierarchy constraint

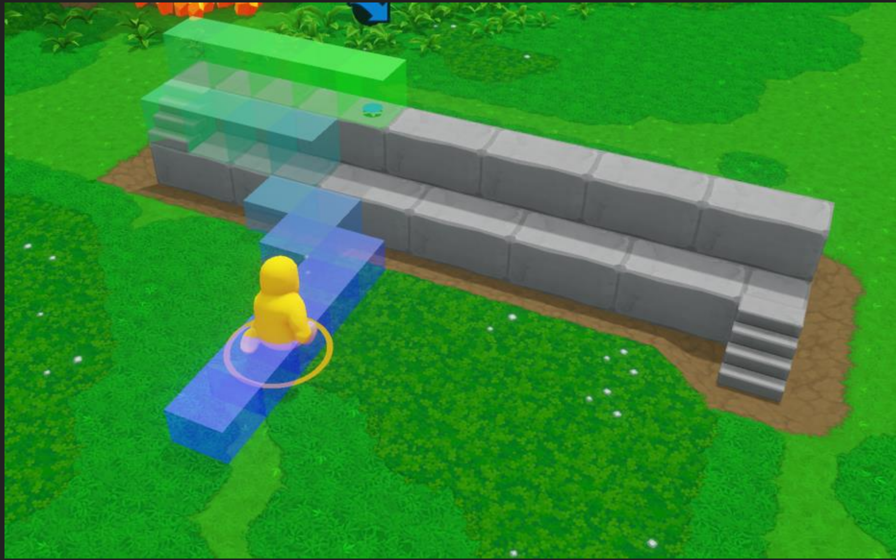


Path Optimality

Still not optimal?

- Extreme weights are hard to accommodate
- We can force lower hierarchy levels but only to a certain point

Optimality ⟷ Pathfinding budget

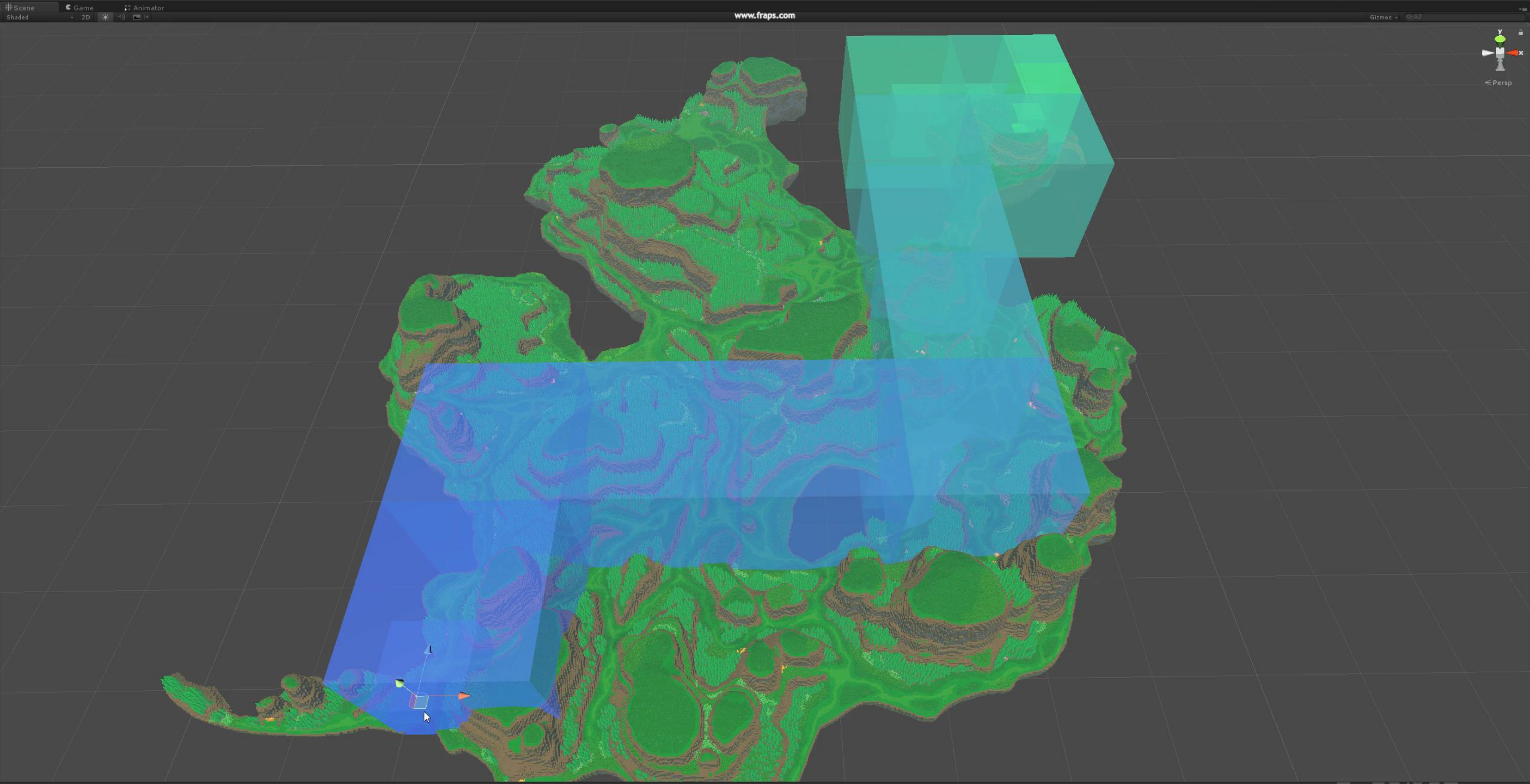


Path Unrolling

Path Unrolling

Characters don't need to know their entire path to move around.

We can defer Path Refinement to the last minute.


Scene
Game
Animator
Shaded
2D
www.fraps.com
Gizmos
Persp

Path Unrolling

Animations need 2-3 nodes of Look-Ahead

Path Error can be computed Partially

Best Case:

- Log(N) steps to get the Path
- Spread the cost of refinement over the execution time!

Path Validation

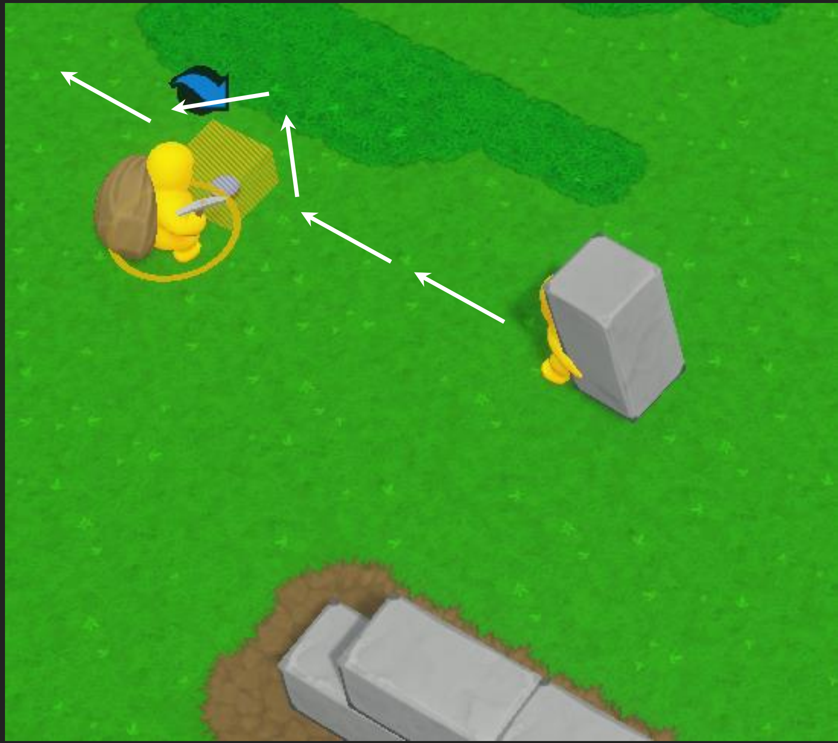
Moving Agents

Finding a path takes a few milliseconds.

Navigating a path takes seconds/minutes.

Paths can become invalid during navigation.

- Terrain is modified
- Other agents/objects moved in the way



Path Validation

Characters need to
Validate their path
as they are walking

When their path is Invalid,
they can stop moving and
find a new one…

):

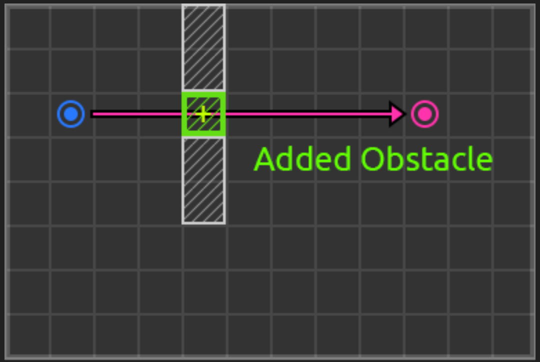
Benoit Alain, Lead Programmer
Sauropod Studio Inc.

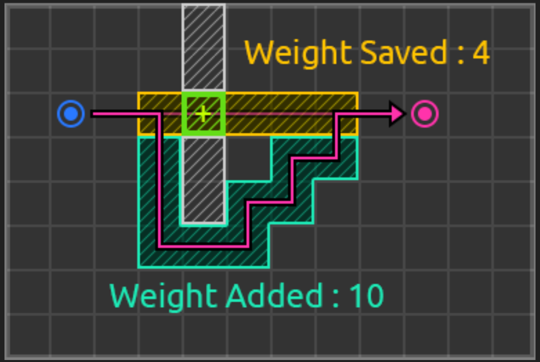
Path Mending

Compute a Patch subpath (if possible)

- Reconnect within distance limit
- Custom A* penalty =
  Weight Added - Weight Saved
- Avoid/penalize voxels where other agents are about to move

Otherwise, Stop Moving and find a different path.





Moving Agents

(:

More Implementation Tips

- Agents are treated as Dynamic Objects by other agents.
  - Enemy dynamic objects trigger the combat AI when met on the field, so they are usually safe to ignore during pathfinding.

- Destroyed terrain nodes can be Pooled and reused.
  - Compare node's *birth timestamp* with *path's timestamp* to know if it has Outdated nodes

- Branch Permissions are used to filter the `AvailableNeighbors` in the A* algorithm.
  - A bit mask can be used to encode branch permissions

Further Optimizations

Gain Performance (simpler cases)

- Replace *priority queue* by regular Queue
  (breadth-first or exhaustive search)
- Don’t test for Dynamic Obstacles
- Don’t store actual Paths (check only reachability)
- Fail after a number of Iterations
- Remain inside a fixed Volume
- Limit the Length of explored paths

Gain Flexibility

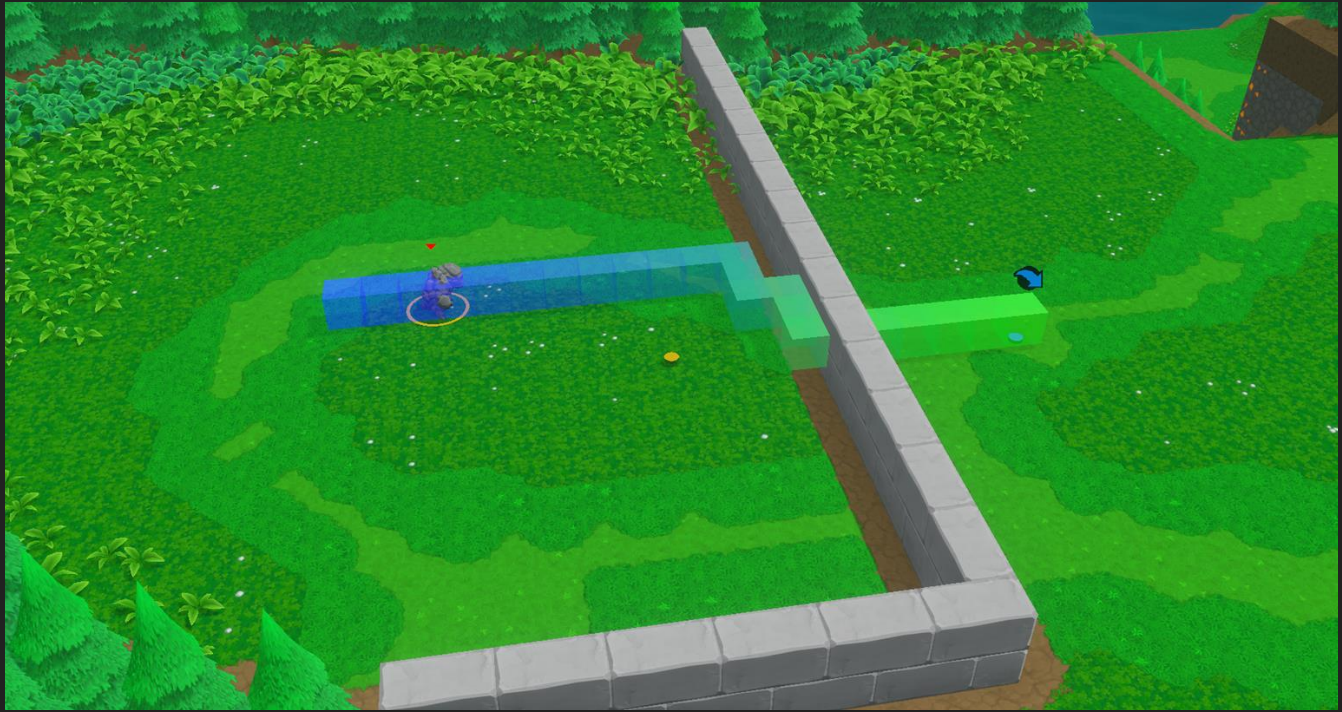
- Store Fallback Paths to intermediate destinations
- Use more precise nodes
  - As distance to Goal decreases
  - In areas where branch weights are fine-tuned
- Use non-standard A* Penalty
  - Position-dependent (diagonals)
  - Gameplay-dependent (fear markers)

Extra Gameplay Examples

Running From Enemies

Mapping Construction Goals

Breaking Through Enemy Walls



CPU and Memory Performance

Memory Usage

	Regular A*	Hierarchical Pathfinding
Nodes (largest maps)	1 million	1.5 million*
Memory per node (avg.)	80 B	100 B
Total memory	80 MB	150 MB

* Assuming 6-8 levels of hierarchy, 3.5 child per parent node on average, castles with ≤ 10,000 bricks

CPU Performance: Construction

- Built From Scratch at game start, from surface and block voxels
  - Base *nodes* and *neighbors* are translated directly from the map data
  - Hierarchical *group formation* explores each node exactly once
  - Total build time is O(N) for N walkable voxels
- Other optimizations were possible, but unnecessary
- Total initialization ≤ 5 s (≈ 20% of map loading time)

CPU Performance: Terrain Modification

- Update time ≤ 0.1 ms for 1 modified voxel, typically
  - Grows linearly in the amount of Hierarchy Levels (6 to 8 levels is a good number)
- Large terrain modifications can be Batched together
  - Computations simplify quickly in the higher hierarchy levels
  - Typically saves 80% to 90% of the update time
- Impact on global performance is Negligible

CPU Performance: Search

	Regular A*	Hierarchical Pathfinding
Nodes (largest maps)	1 million	1.5 million
Explored nodes (worst case)	1 million	1000-3000 *
Explored nodes (best case)	O(N)	O(Log(N))
Average time (worst case)	2-3 minutes **	0.5 seconds

* Depending on the number of *dynamic obstacles* (usually less than a few hundreds).

** Assuming a limit of 100 ticks per frame @ 60 fps.

Successes

- Simple concepts
- No precomputation (procedural world generation?)
- Fast init & maintain
- Lightweight
- Flexible, few core classes
- Beats regular A* by a factor of 100-1000 (!!)

Limitations

- Path optimality failures: weights are too extreme, too subtle
- Voxel-based. Hackish diagonals, what about other curves?
- Still a performance bottleneck
- Larger units: possible, but at a cost
- Digging units: don't fill the entire terrain with nodes!
- Flying units: requires new rules, don't place nodes everywhere in the sky!

Further Work

- Threading
- Infinite worlds
- Portals
- Public transportation?

Thank you for listening!

Special thanks to Faviann Di Tullio for spending a full weekend brainstorming and reviewing code with me before this was an official project.

Special thanks to Germain Couët for reviewing the visual support and providing graphics for this presentation.

Questions?

More questions? Please email me at
benoit@sauropodstudio.com or come see us in Montréal!