Pick Your T-shirt – Style 1

- Piccolo Souvenir T-shirt
  - Style 1
  - Piccolo Digital World Series

  - Key elements
  - Piccolo Logo P surrounded by Piccolo codes, representing the digital world created by Piccolo

Game-Engine Explorer

Pick Your T-shirt – Style 2

- Piccolo Souvenir T-shirt
  - Style 2
  - Piccolo Pro Series

  - Key elements
  - Code between Programmers (DDDD) +Equation of quaternion function, representing professional

Game-Engine Explorer

Get Your Own T-shirt

- How to get?
  - Please comment under《Lecture16》@Bilibili before 10:00 am next Monday 7/25
  - we will give out 10 T-shirts for the best comments

- Reward List for Naming Event
  - otaku小许、张茂、Frozen、核桃、王十一、李同学、蝈蝈、塞伦斯、鄙姓金名聪、家子颜

Q&A

- Q1: Is component-based architecture best suited for gameplay logic?
- Q2: How could Blueprint System support collaborative editing?
- Q3: Will Event System consider priority of events ?

Lecture 16

Gameplay Systems

Basic Artificial Intelligence

WANG XI

GAMES 104

2022

Outline of Artificial Intelligence Systems

01.

AI Basic

- Navigation
- Steering
- Crowd Simulation
- Sensing
- Classic Decision Making Algorithms

02.

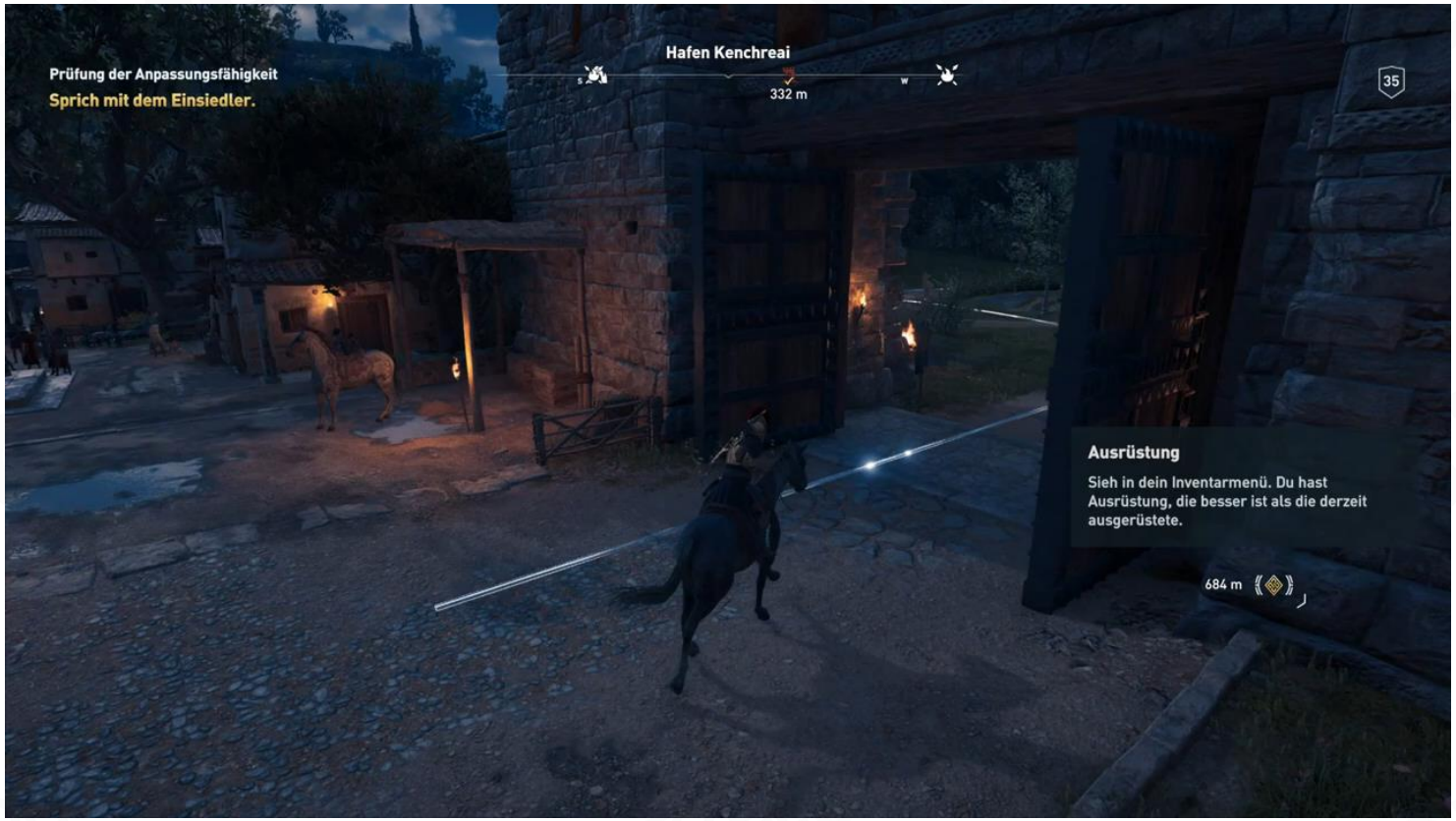
Advanced AI

- Planning and Goals
- Machine Learning

Navigation

Navigation In Games

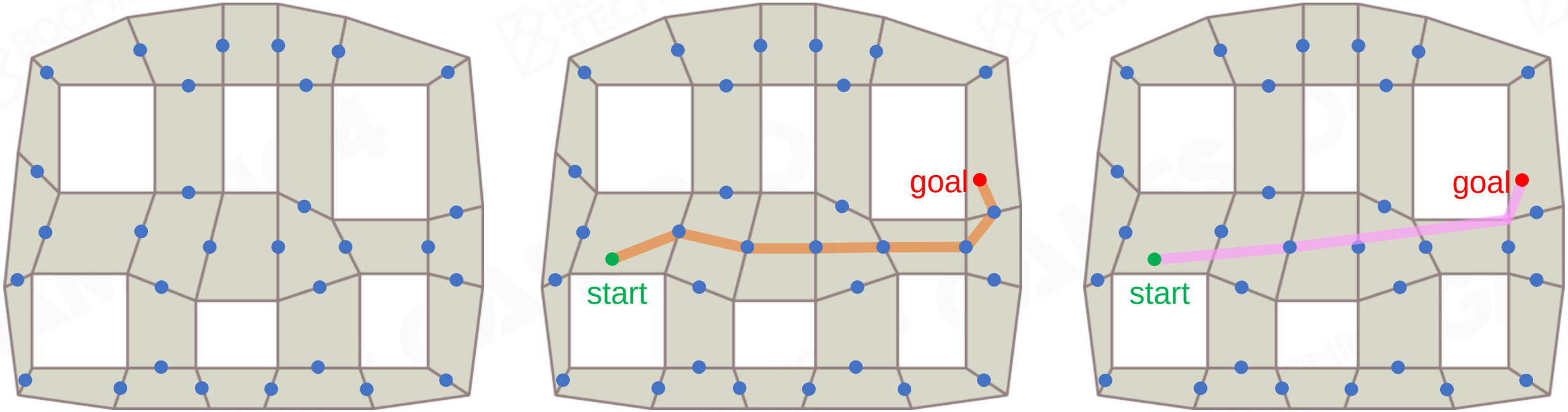
Find paths from a location to another in an automatic manner



Assassin's Creed: Odyssey

Navigation Steps

Map representation → Path finding → Path smoothing



Map Representations – Walkable Area

- We need to tell AI agents where they can walk – Walkable area
- Walkable area of players is determined by character motion capabilities
  - Physical Collision
  - Climbing slope/height
  - Jumping distance
  - …
- Simulating movement of AI agents as players costs too much
- AI agents are still expected to have the same walkable area as players

Map Representations - Formats

- Waypoint Network
- Grid
- Navigation Mesh
- Sparse Voxel Octree

Waypoint Network (1/3)

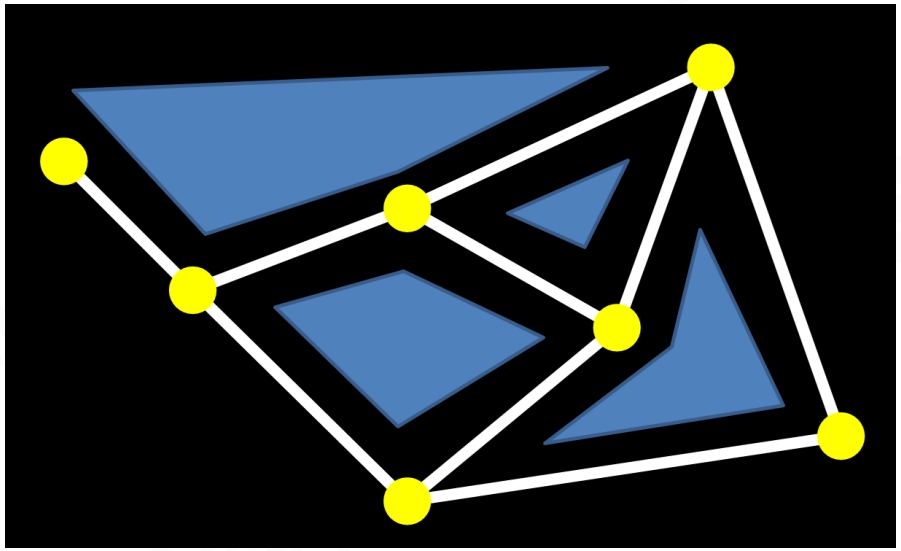
- Network connecting critical points (waypoints) from the map
- Waypoint sources:
  - Designed important locations
  - Corner points to cover walkable area
  - Internal points to connect near-by waypoints adding flexibility to navigation

World of Warcraft

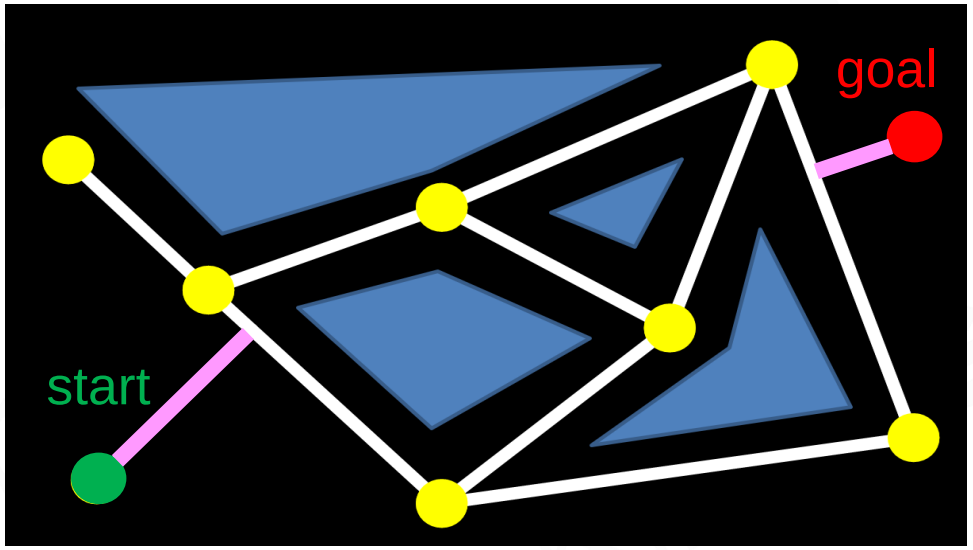
Waypoint Network (2/3)

Usage of waypoint network is similar to subway system

- Find the nearest points to get on and off the network
- Plan the path on the waypoint network



waypoint network



path on waypoint network

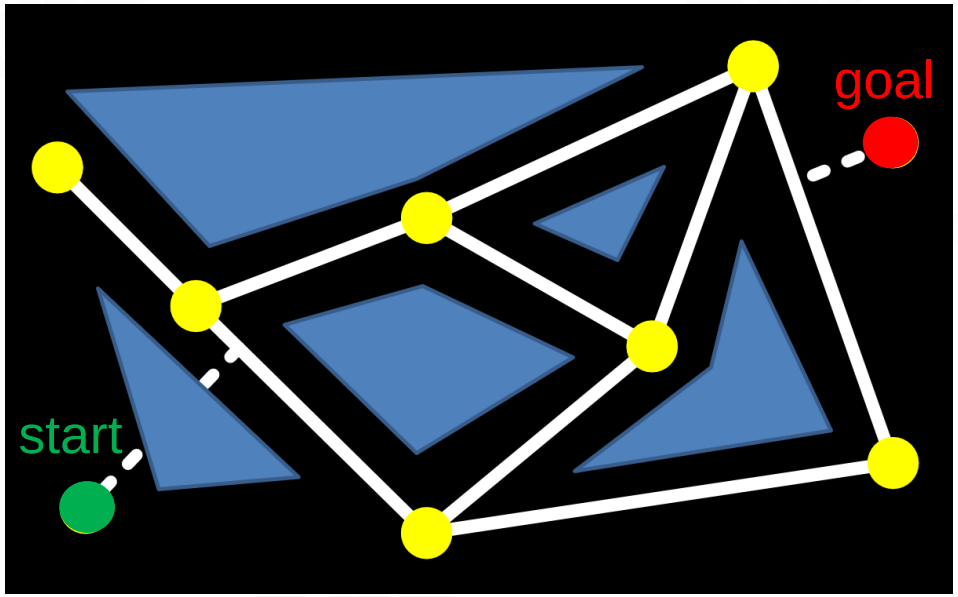
Waypoint Network (3/3)

Pros:

- Easy to implement
- Fast path finding, even for large maps

Cons:

- Limited flexibility: must go to the nearest point in the network before navigation
- Waypoint selection requires manual intervention



Grid (1/3)

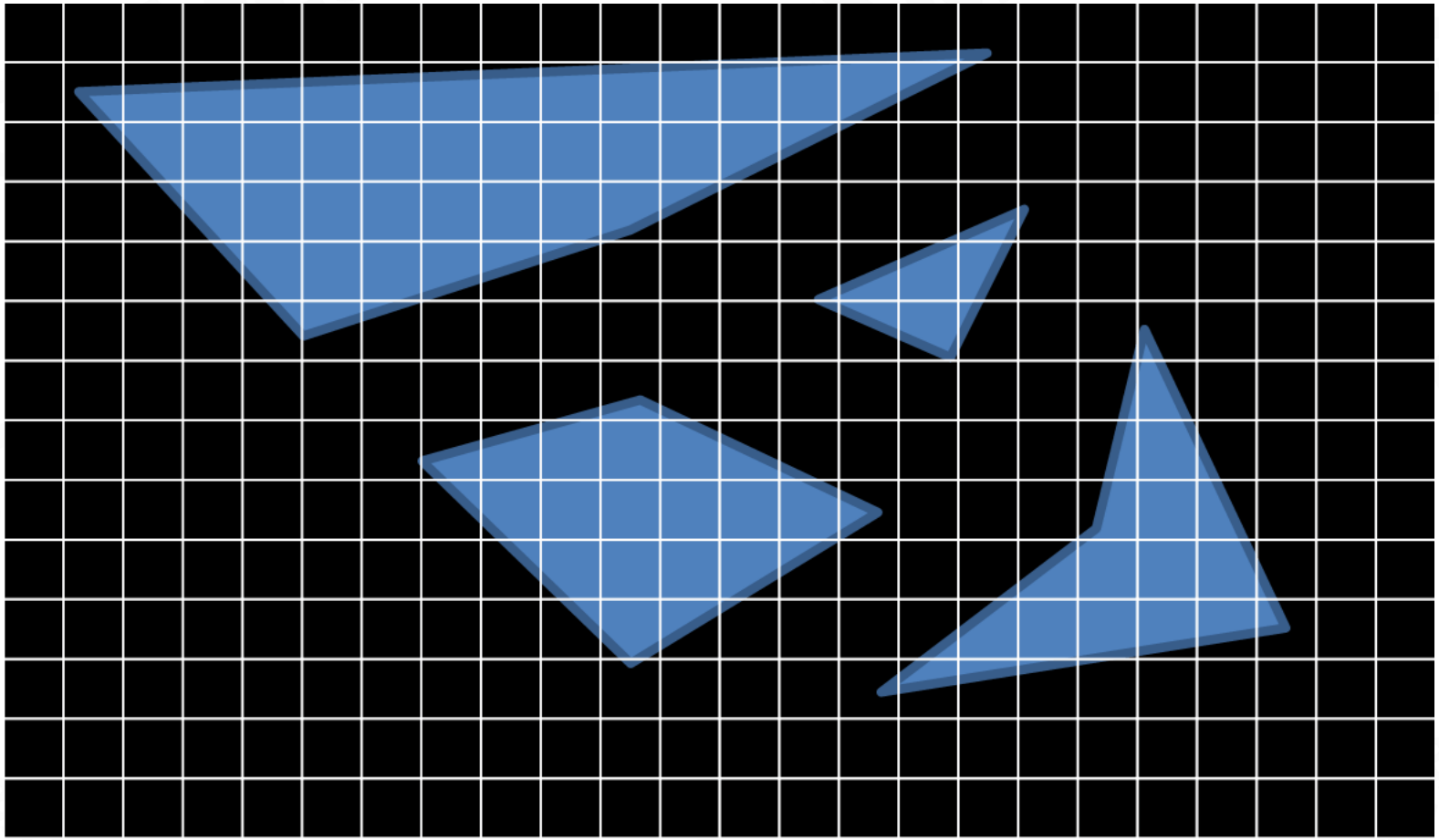
- Intuitive discretization of map
- Uniform subdivision into small regular grid shapes
- Common grid shapes
  - Square
  - Triangle
  - Hexagon

Sid Meier's Civilization IV

Sid Meier's Civilization V

Grid (2/3)

Grid property could be modified in runtime to reflect dynamic environmental changes



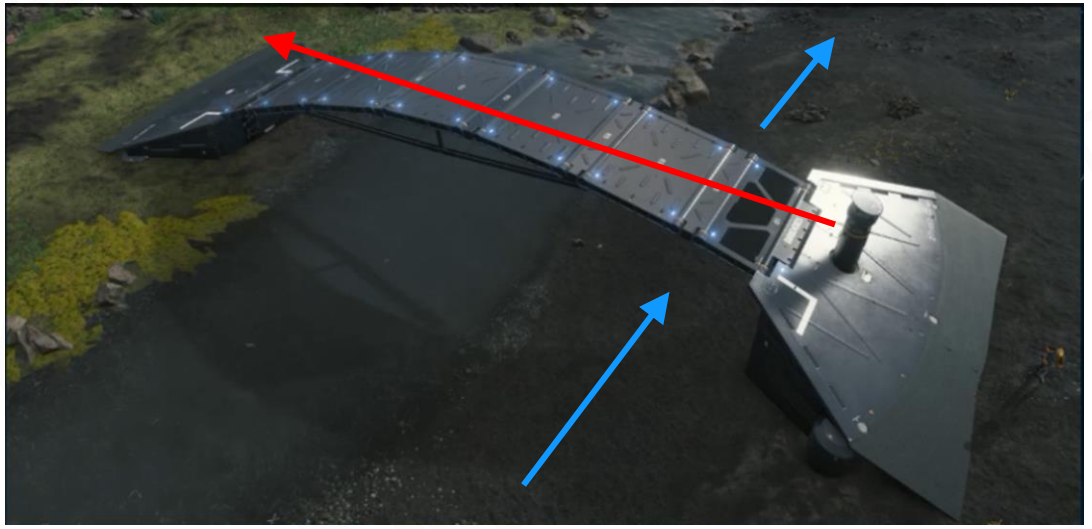
Grid (3/3)

Pros:

- Easy to implement
- Uniform data structure
- Dynamic

Cons:

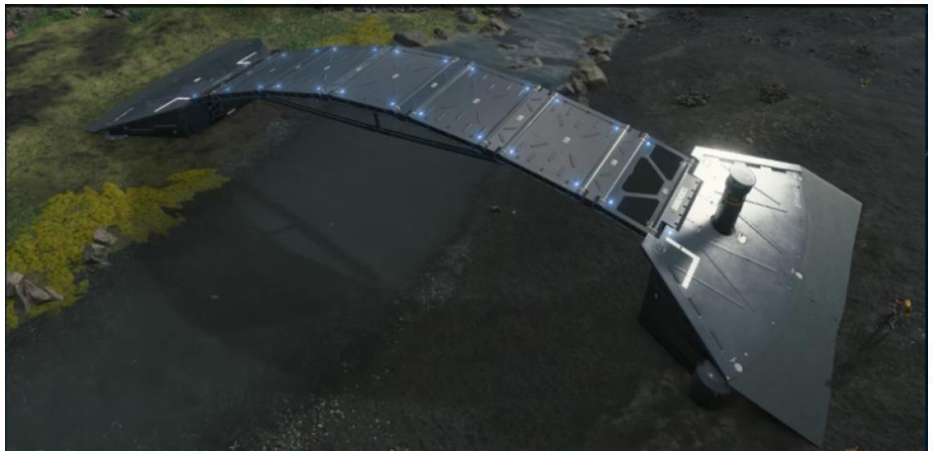
- Accuracy depends on grid resolution
- Dense grid lowers pathfinding performance
- High memory consumption
- Hard to handle 3D map



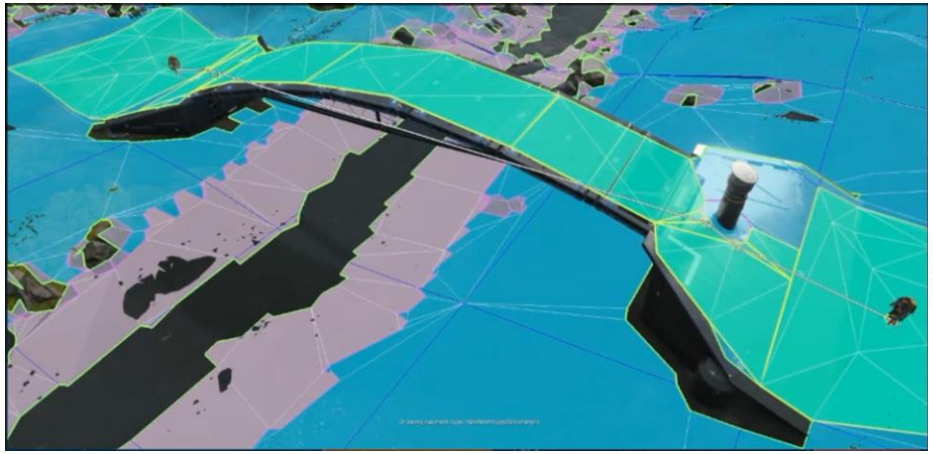
3D overlapping walkable surface

Navigation Mesh (NavMesh)

- Solves the problem of representing overlapped walkable areas
- Approximates the walkable area of character controller based on physical collision and motion capabilities
- Lowers network density to boost pathfinding performance



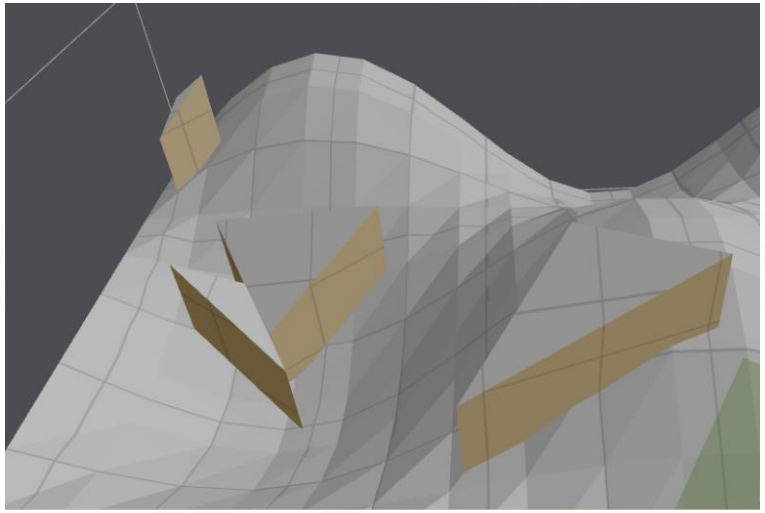
Original 3D bridge



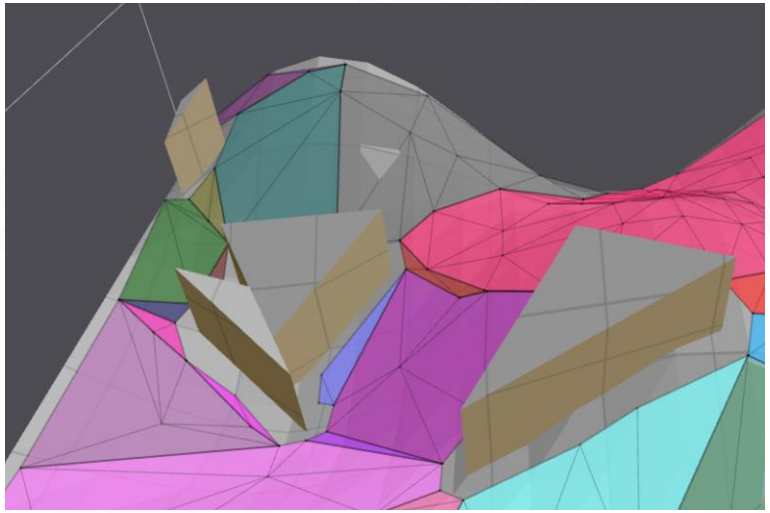
3D bridge on NavMesh

NavMesh Example

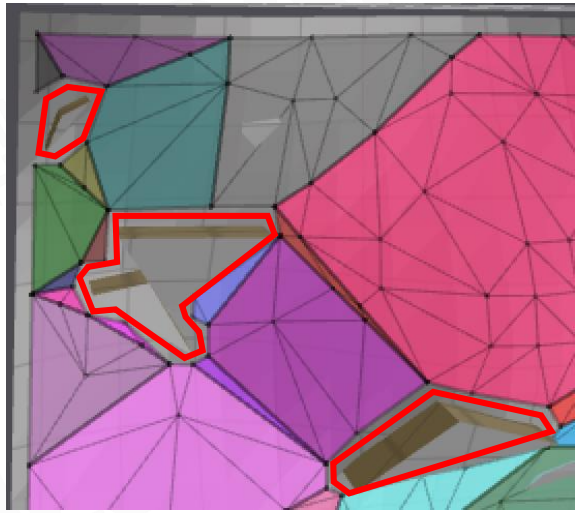
Neighboring 3D convex polygons to represent walkable areas



original mesh



NavMesh

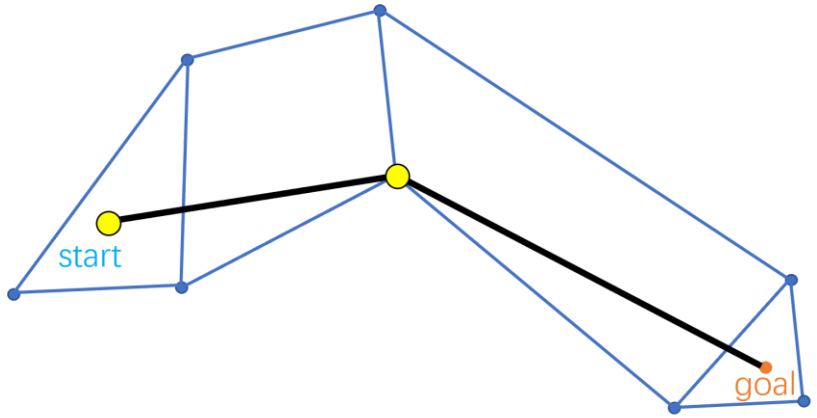


Top-down view of NavMesh

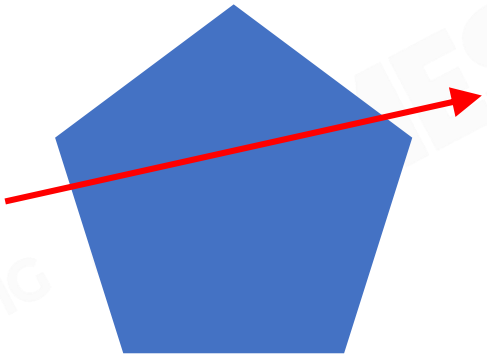
Convex Polygon of NavMesh

Why convex polygon?

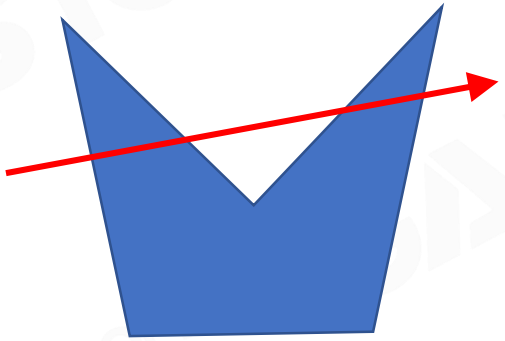
- Pathfinding generates a series of polygon (**Polygon Corridor**) need to walk through
- Convexity guarantees the final path is limited in the polygon and two adjacent polygons have only one common edge (**Portal**)



polygon corridor



convex polygon



concave polygon

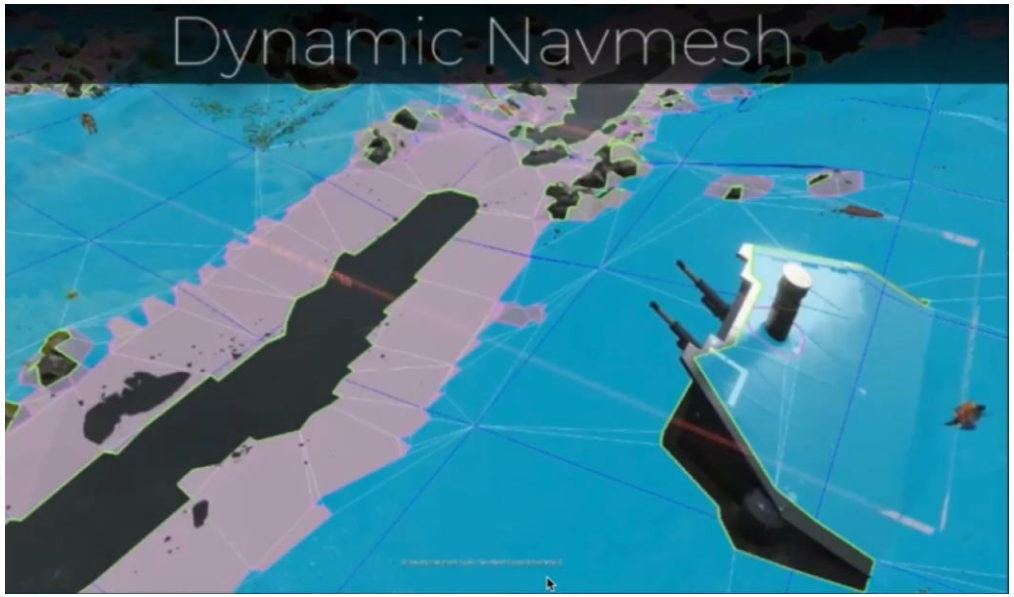
NavMesh Pros and Cons

Pros:

- Support 3D walkable surface
- Accurate
- Fast in pathfinding
- Flexible for selection of start/destination
- Dynamic

Cons:

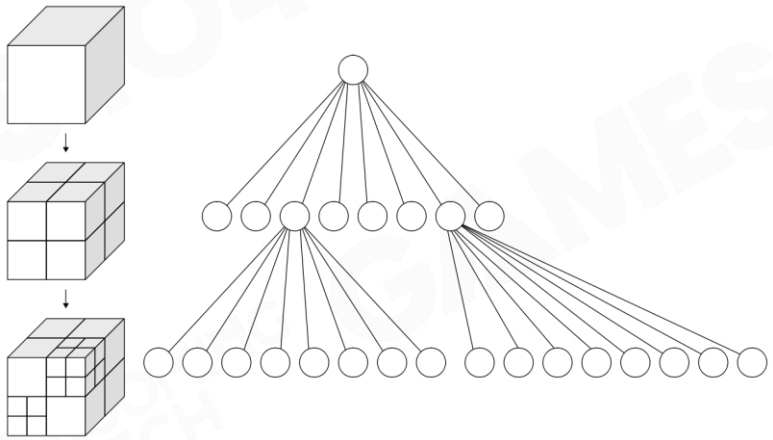
- Complex generation algorithm
- Not support 3D space

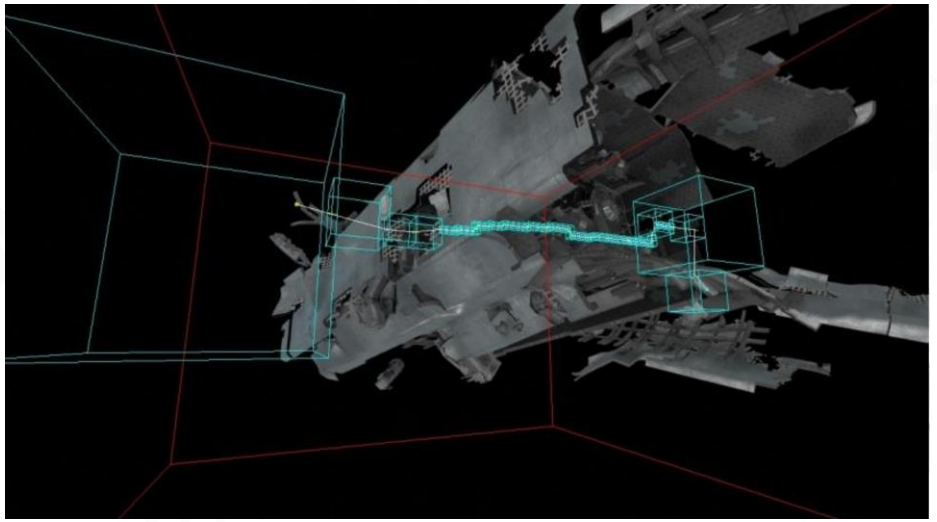


Death Stranding

Sparse Voxel Octree

- Represents “**flyable**” 3D space
- Similar to spatial partitioning
- Finest level voxels represents complicated boundary
- Coarser-level voxels represents uniform regions

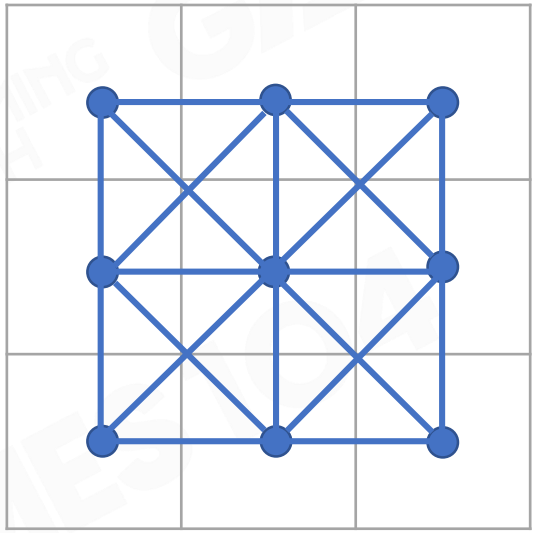




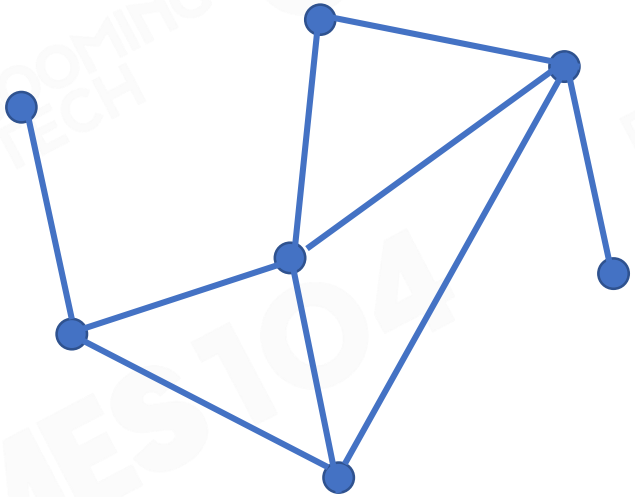
Warframe

Path Finding (1/2)

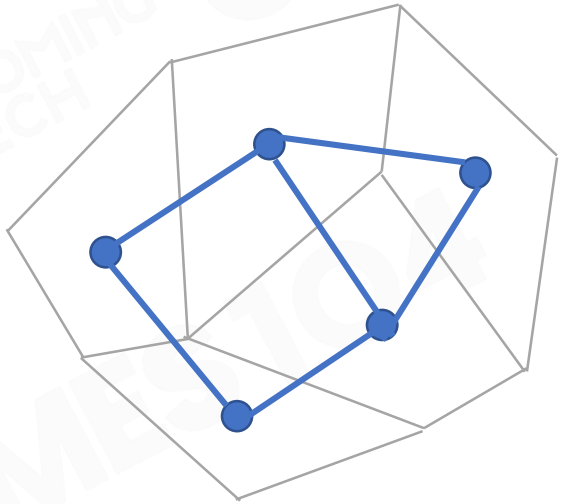
Distances in map representations can be abstracted as edge costs in graph



grid



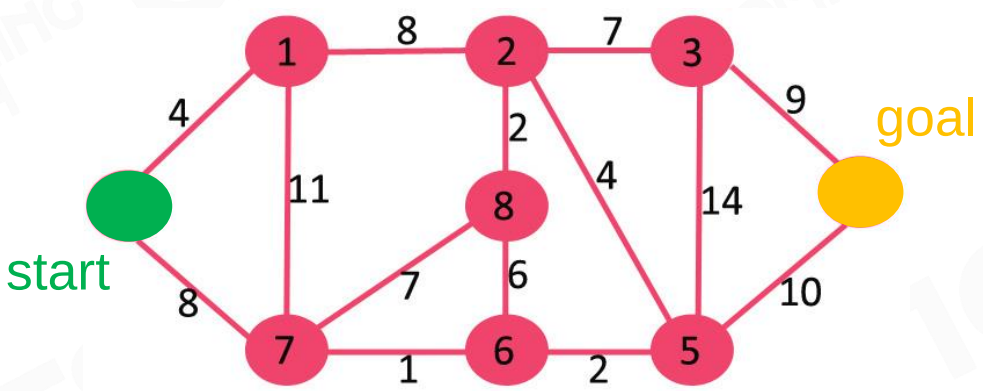
waypoint

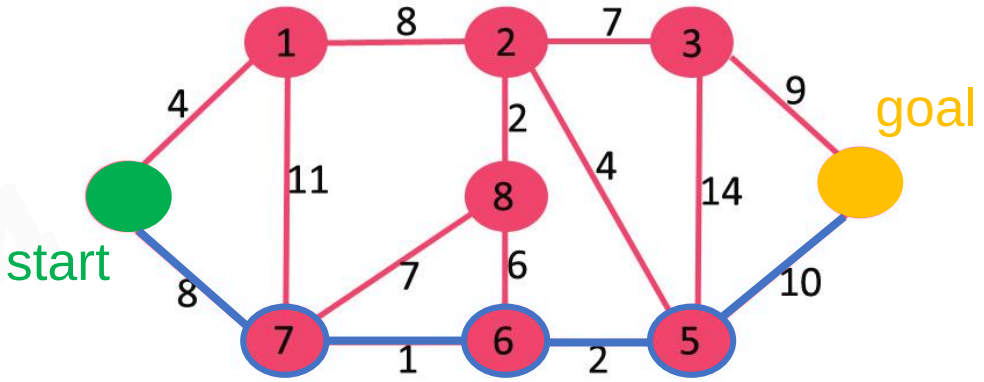


NavMesh

Path Finding (2/2)

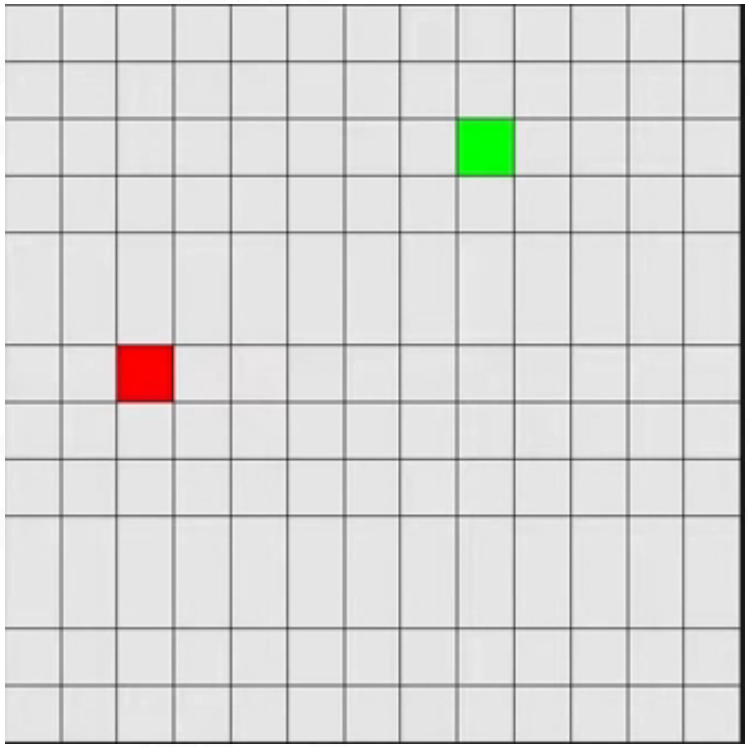
Pathfinding can be abstracted as shortest path problem in non-directional graph

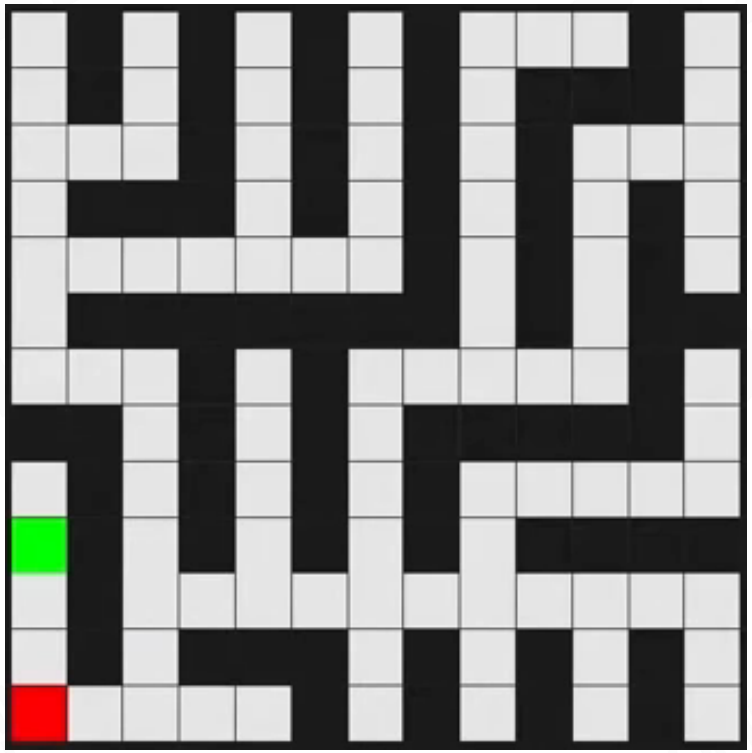




Depth-First Search

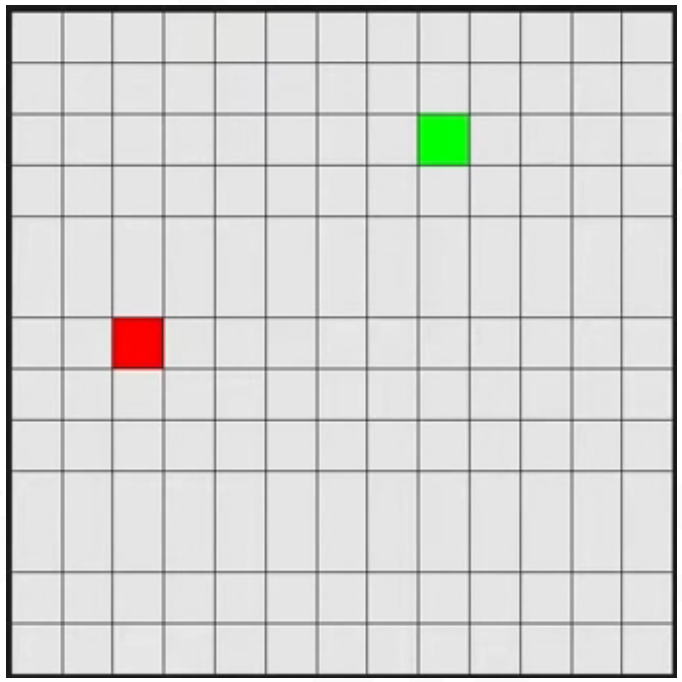
Expand most recently added

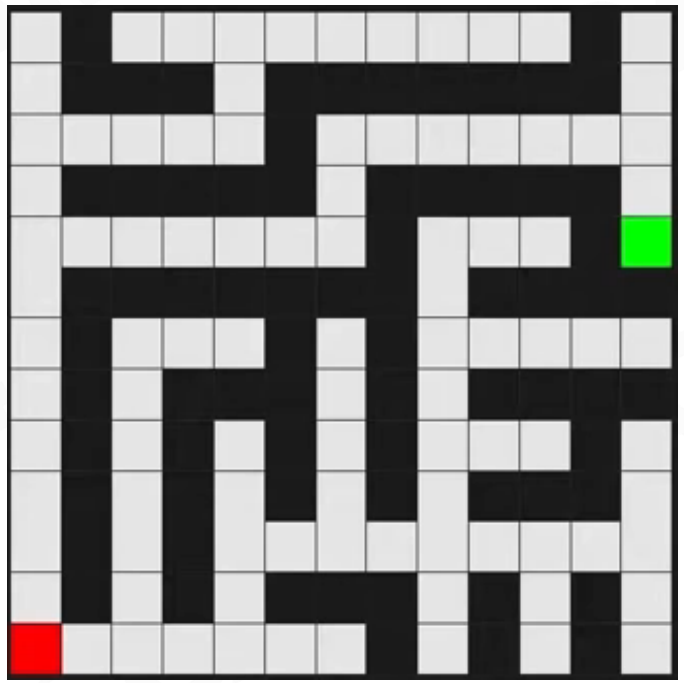




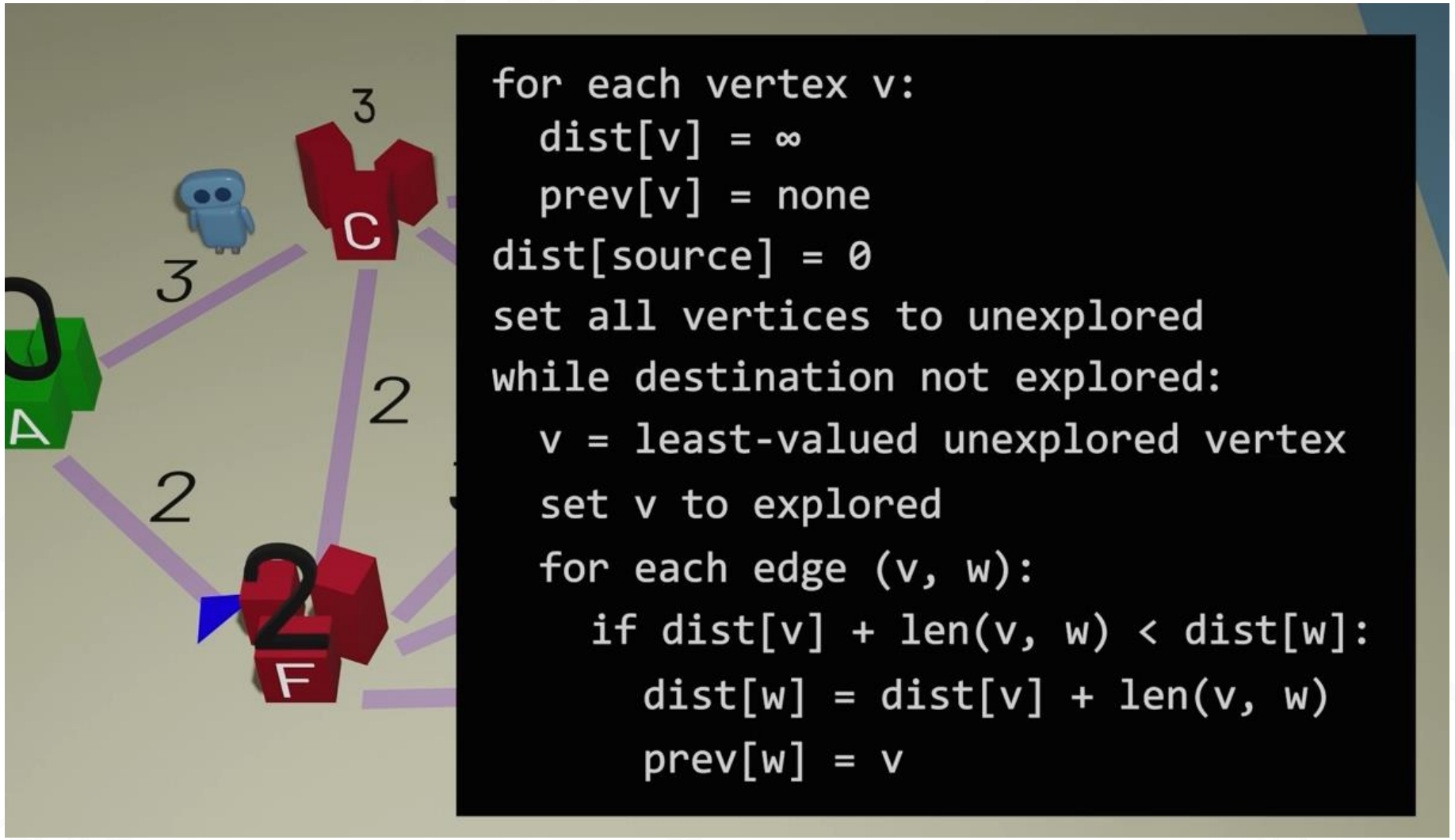
Breadth-First Search

Expand least recently added

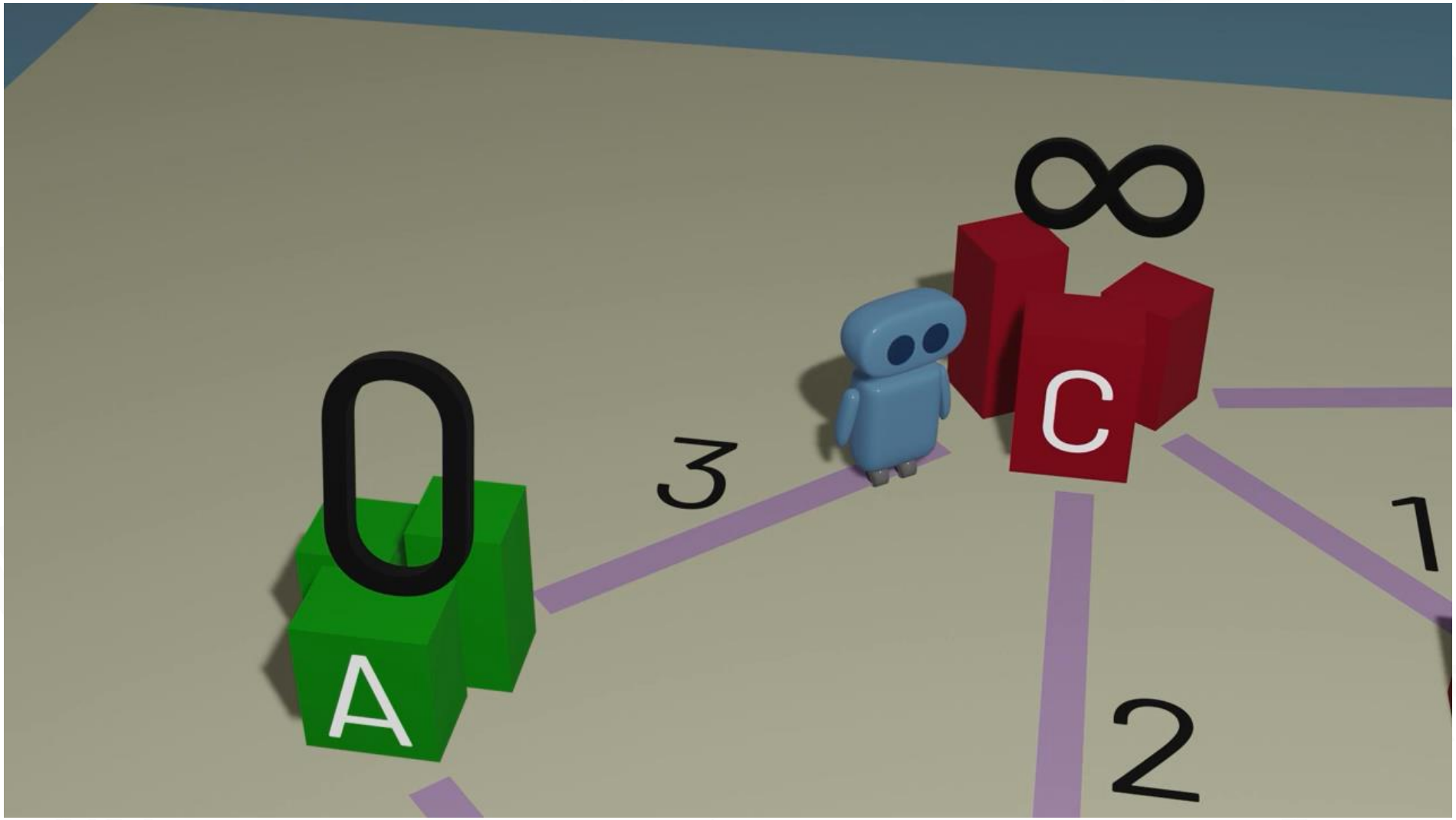




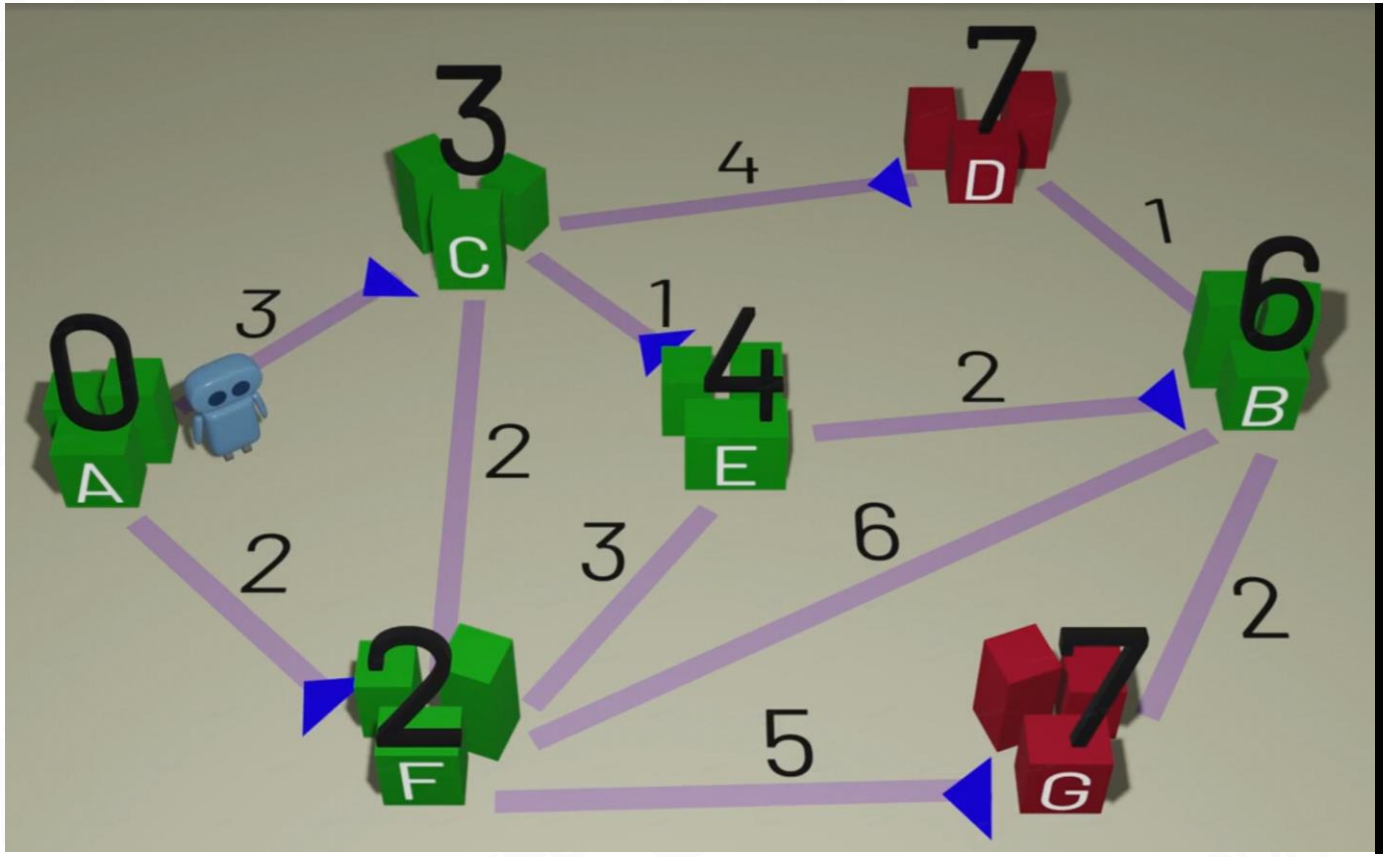
Dijkstra Algorithm (1/3)



Dijkstra Algorithm (2/3)



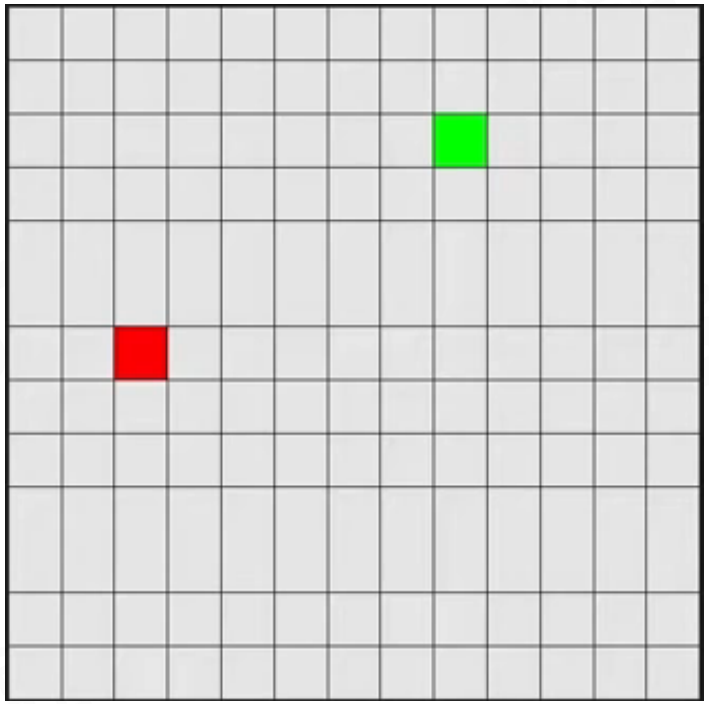
Dijkstra Algorithm (3/3)

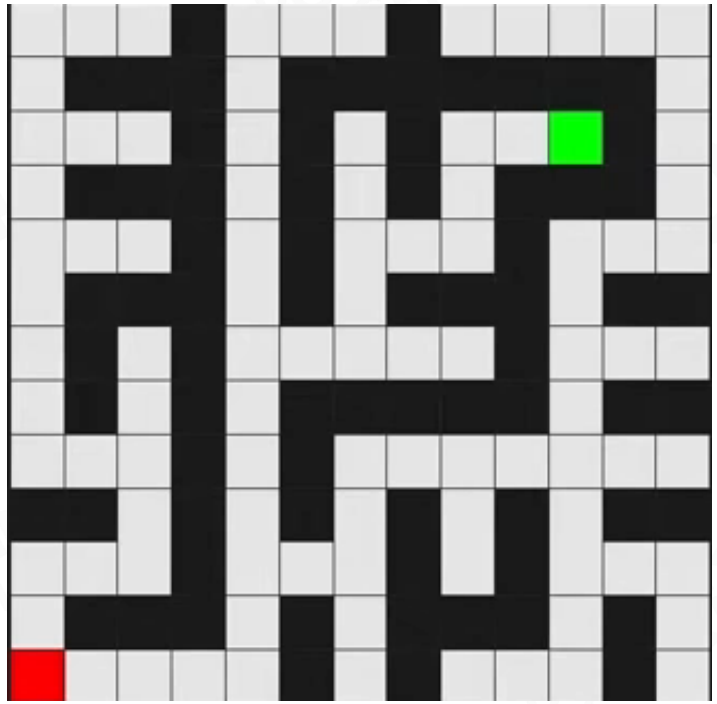



```
for each vertex v:
  dist[v] = ∞
  prev[v] = none
dist[source] = 0
set all vertices to unexplored
while destination not explored:
  v = least-valued unexplored vertex
  set v to explored
  for each edge (v, w):
    if dist[v] + len(v, w) < dist[w]:
      dist[w] = dist[v] + len(v, w)
      prev[w] = v
```

A Star (A*)

- Expand lowest cost in list
- Distance is known distance from source + heuristic
- Greedy: stops when reaches the goal

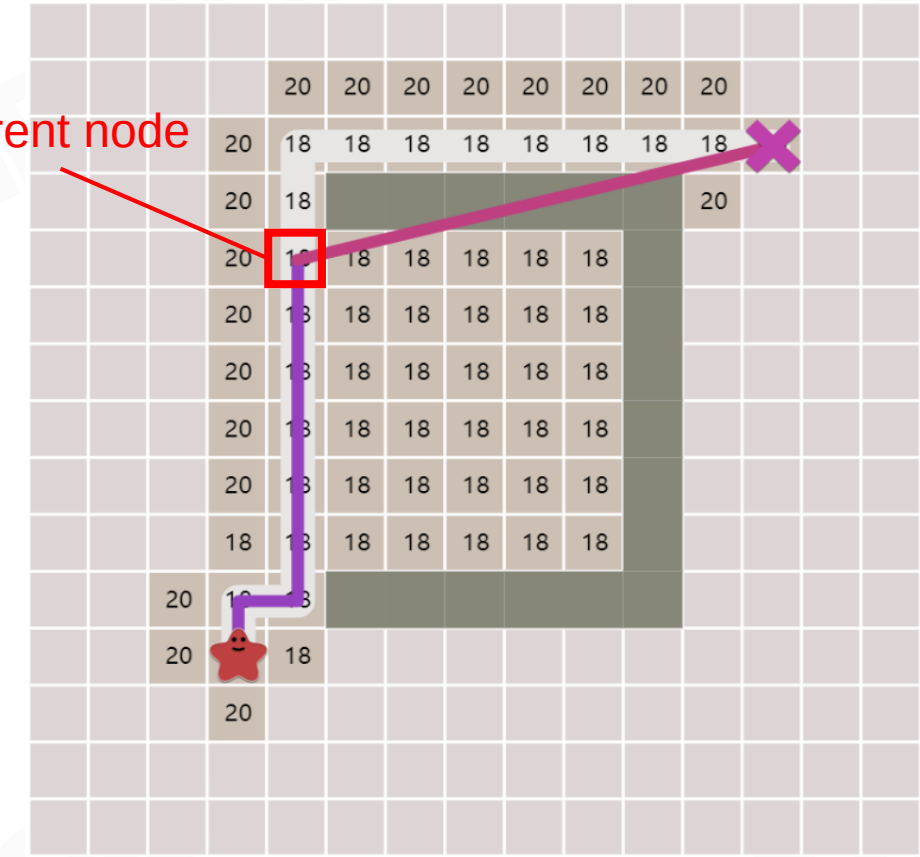




A* – Cost calculation

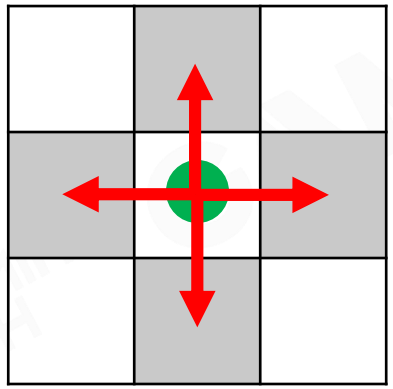
Cost calculation: $f(n) = g(n) + h(n)$

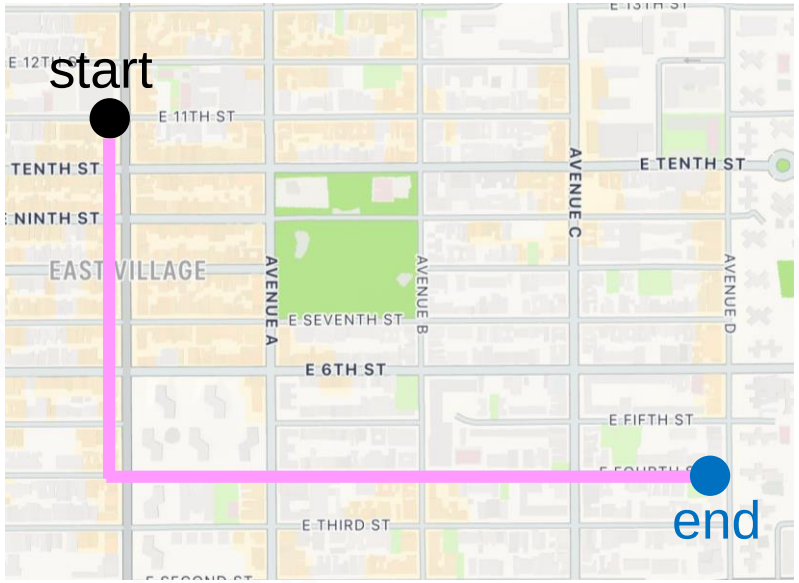
- $g(n)$: the exact cost of the path from the start to node n
- $h(n)$: the estimated cost from node n to the goal

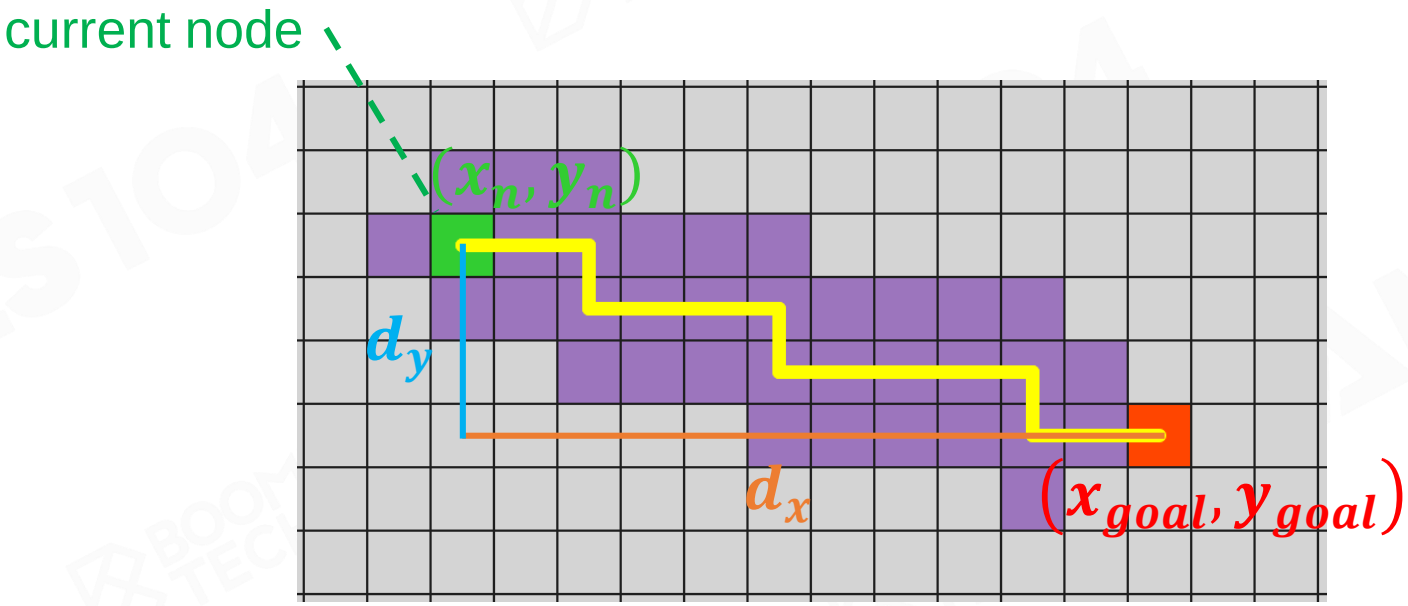


A* – Heuristic On Grids

- For 4 directions of movement, we can use Manhattan distance
- D_1: cost for moving to the adjacent node
- $h(n) = D_1 \cdot (d_x + d_y)$
  - $d_x = |x_n - x_{goal}|$, $d_y = |y_n - y_{goal}|$



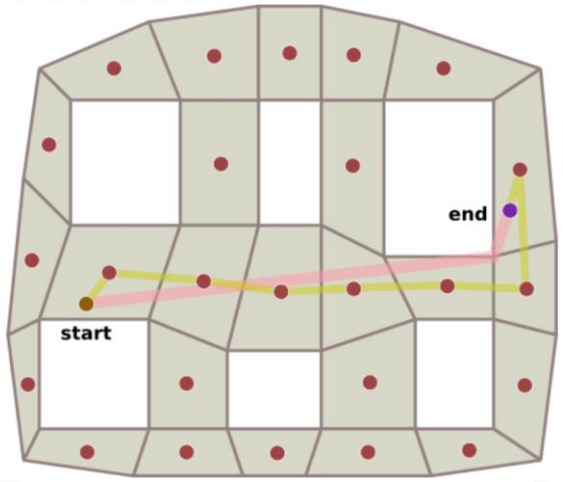




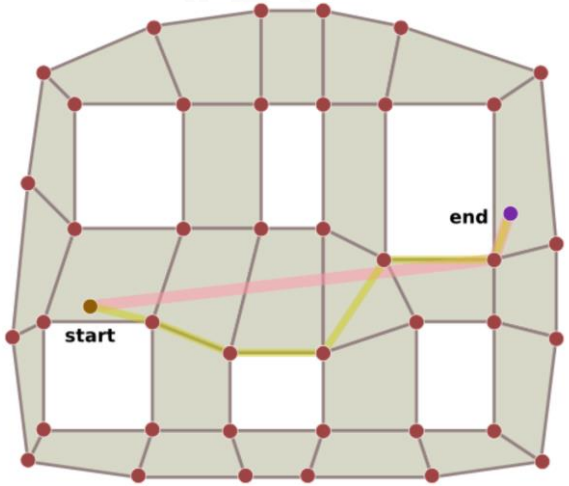
A* – Heuristic On NavMesh (1/2)

Multiple choices when evaluating cost on NavMesh

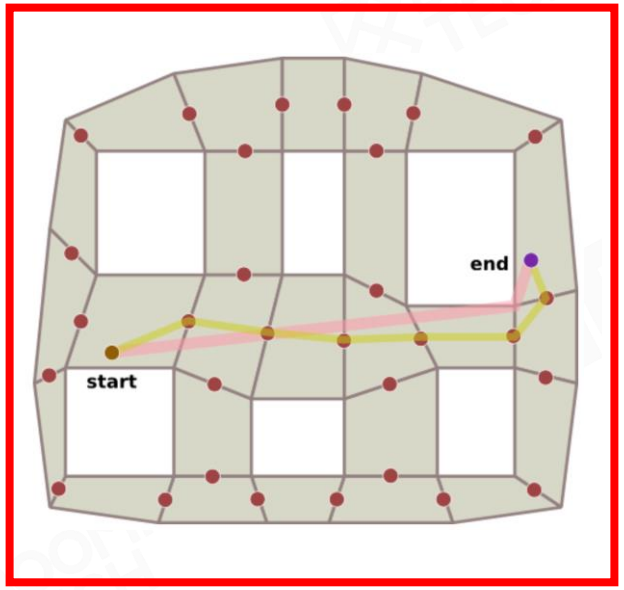
- Using polygon centers or vertices usually over-estimate the cost
- Using hybrid method introduces too many points to check
- Midpoints of edges – a good balance



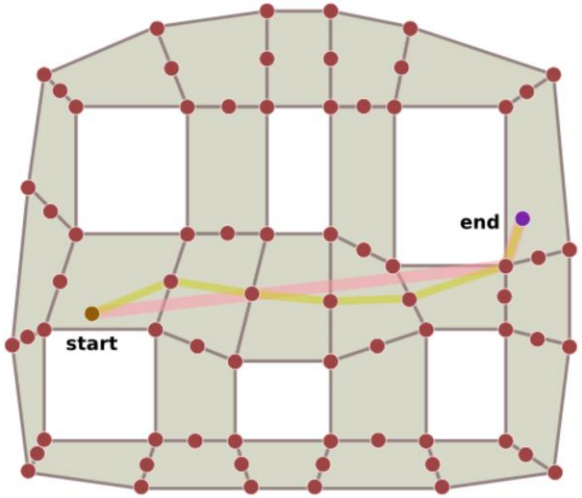
polygon centers



polygon vertices



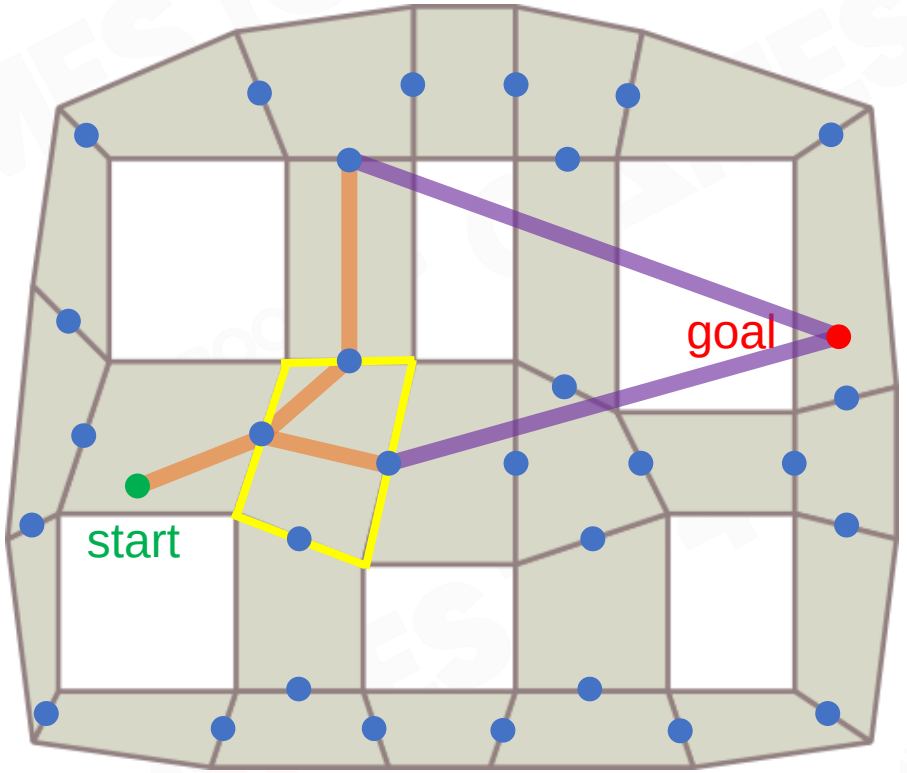
midpoints of edges



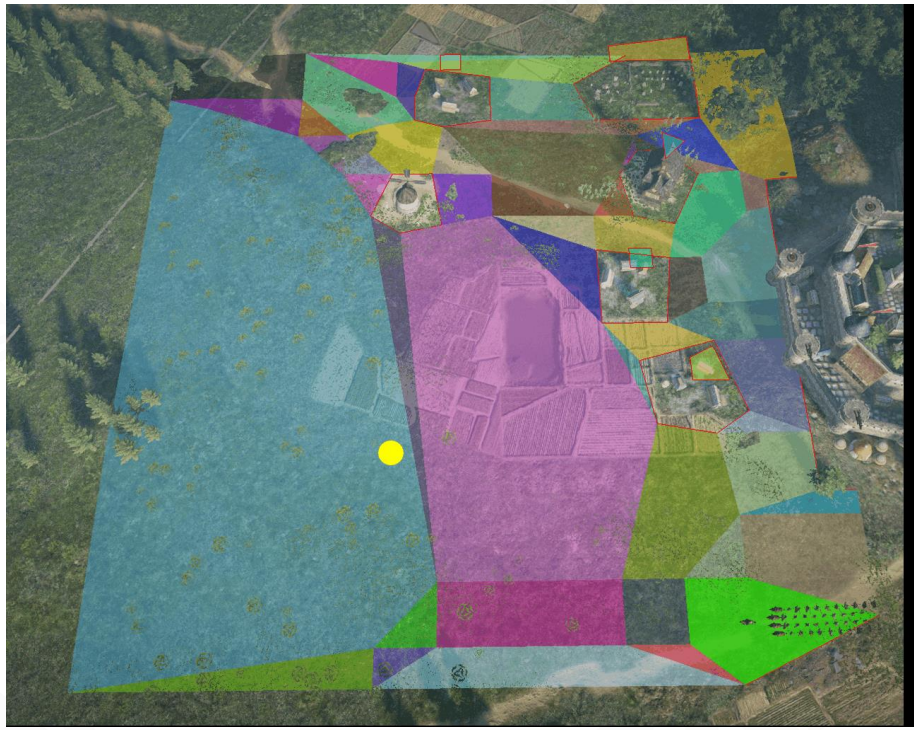
hybrid

A* – Heuristic On NavMesh (2/2)

- On a navigation mesh that allows any angle of movement, use a straight line distance
- Use midpoint of the edge entering the current node as node cost calculation point
- D: the cost for moving unit distance in any direction
  - $h(n) = D \cdot \sqrt{d_x \cdot d_x + d_y \cdot d_y}$
  - $d_x = |x_n - x_{goal}|, d_y = |y_n - y_{goal}|$



A* – NavMesh Walkthrough



A* algorithm

Graph: Explore Neighbours

Greedy: Stop once reached

Least Cost First Explore

TotalCost
=
Cost + Heuristic

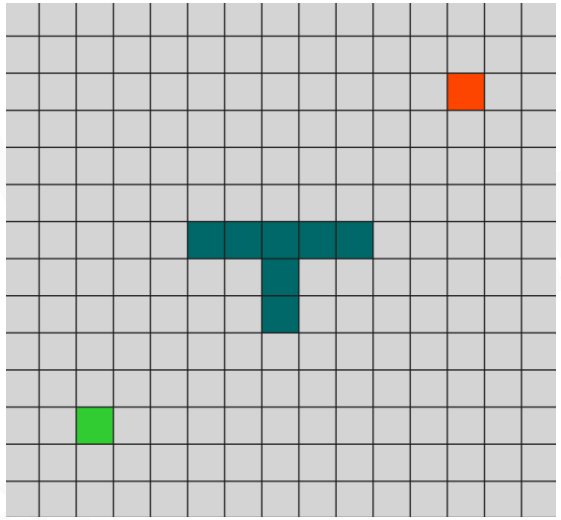
Raw Path

Algorithm 1 A*

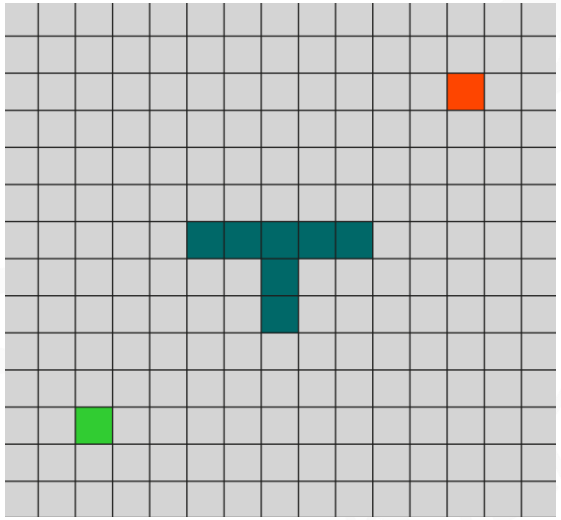
```
1:  openList ⇐ empty list of nodes
2:  closedList ⇐ empty list of nodes
3:  startNode.f ⇐ 0
4:  add startNode to openList
5:  while openList is not empty do
6:      currentNode ⇐ the node with the least f value
7:      remove currentNode from openList
8:      add currentNode to closedList
9:      if currentNode is goal then
10:         backtrack to get path, terminate
11:     end if
12:     children ⇐ the adjacent nodes
13:     for all child in children do
14:         if child is in closedList then
15:             continue
16:         end if
17:         g ⇐ currentNode.g + distance(child, currentNode)
18:         if child is in openList and a then
19:             if g > child.g then
20:                 continue
21:             end if
22:         else
23:             add child to openList
24:         end if
25:         child.g ⇐ g
26:         child.h ⇐ heuristic of distance(child, goal)
27:         child.f ⇐ child.g + child.h
28:     end for
29: end while
```

A* – Heuristic

- $h(n)$ controls A*'s behavior:
  - With 100% accurate estimates, get shortest paths quickly
  - Too low, continue to get shortest paths, but slow down
  - Too high, exit early without shortest path
- Balance between pathfinding speed and accuracy



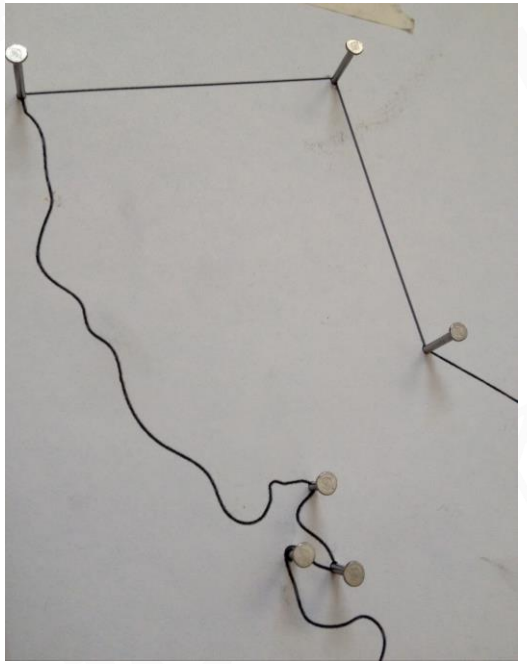
high $h(n)$

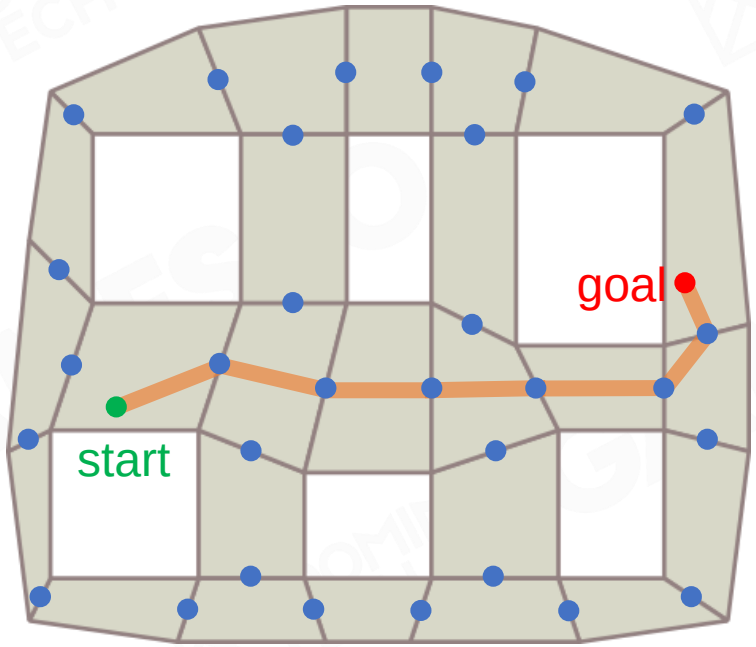


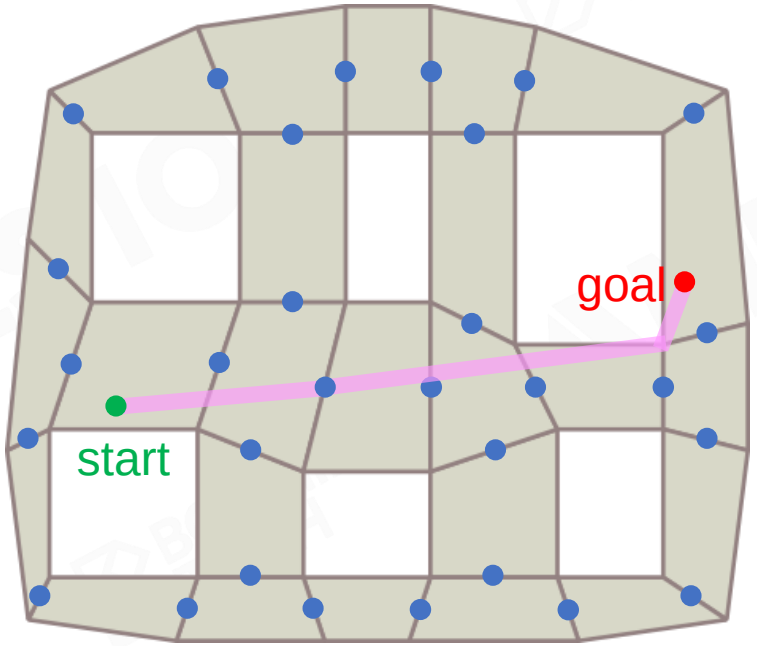
low $h(n)$

Path Smoothing

- Why we need path smoothing
  - Zigzag, many unnecessary turns
- “String Pulling” – Funnel Algorithm

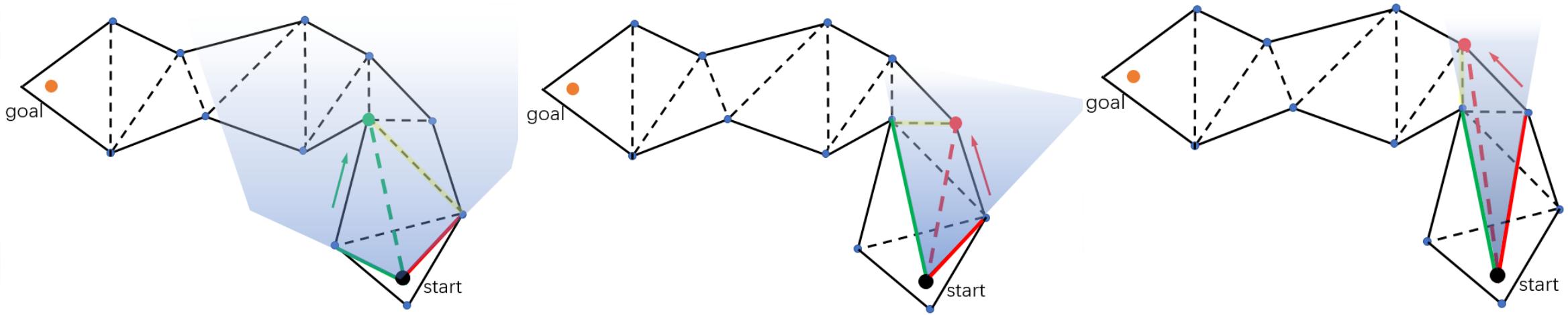






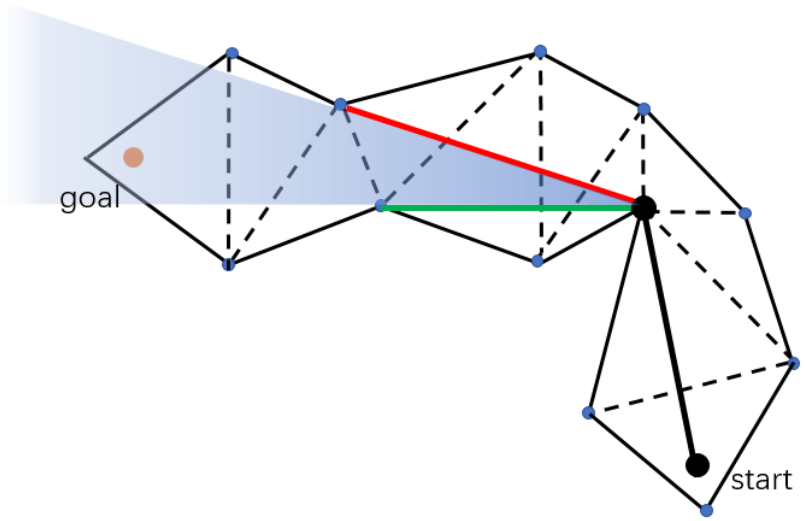
Path Smoothing – Funnel Algorithm (1/2)

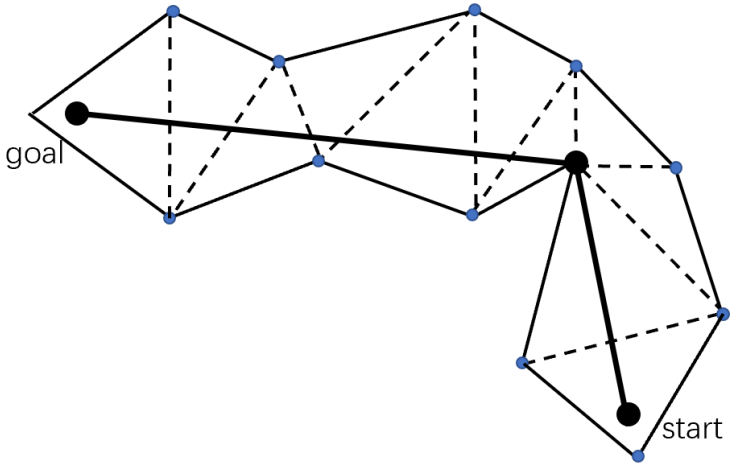
- The scope of the funnel is the possible scope of the path
- Narrow the funnel if necessary to fit the portal



Path Smoothing – Funnel Algorithm (2/2)

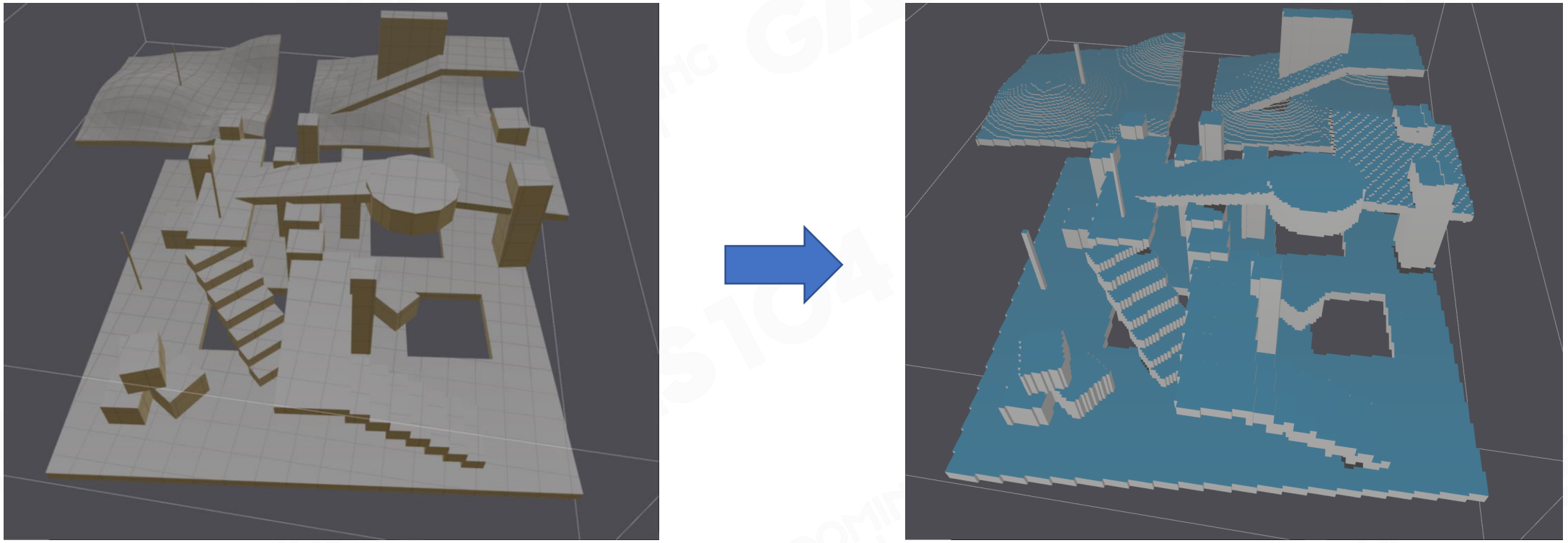
Terminate when the goal is in the funnel





NavMesh Generation – Voxelization

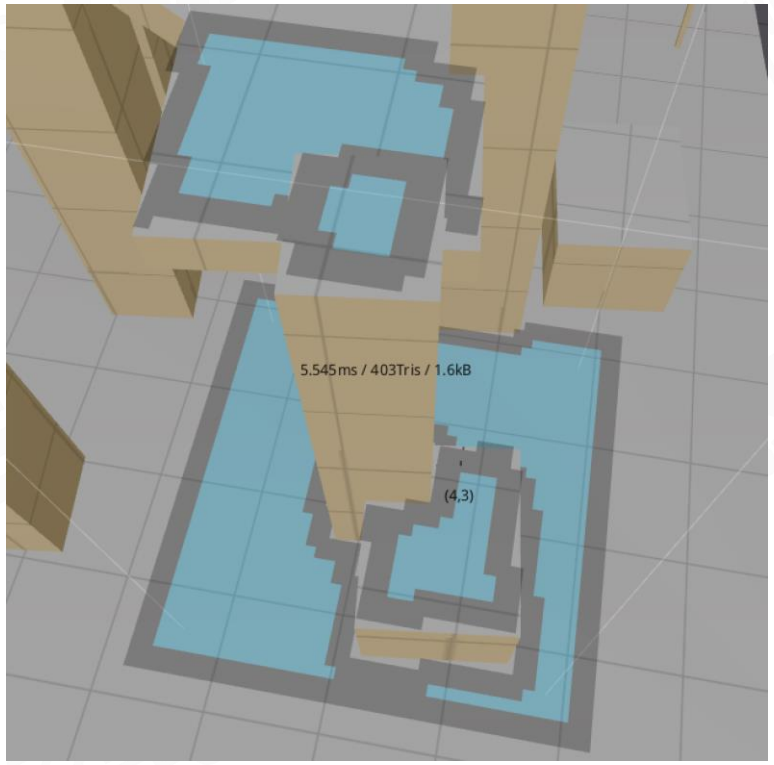
Sample collision scene by voxelization

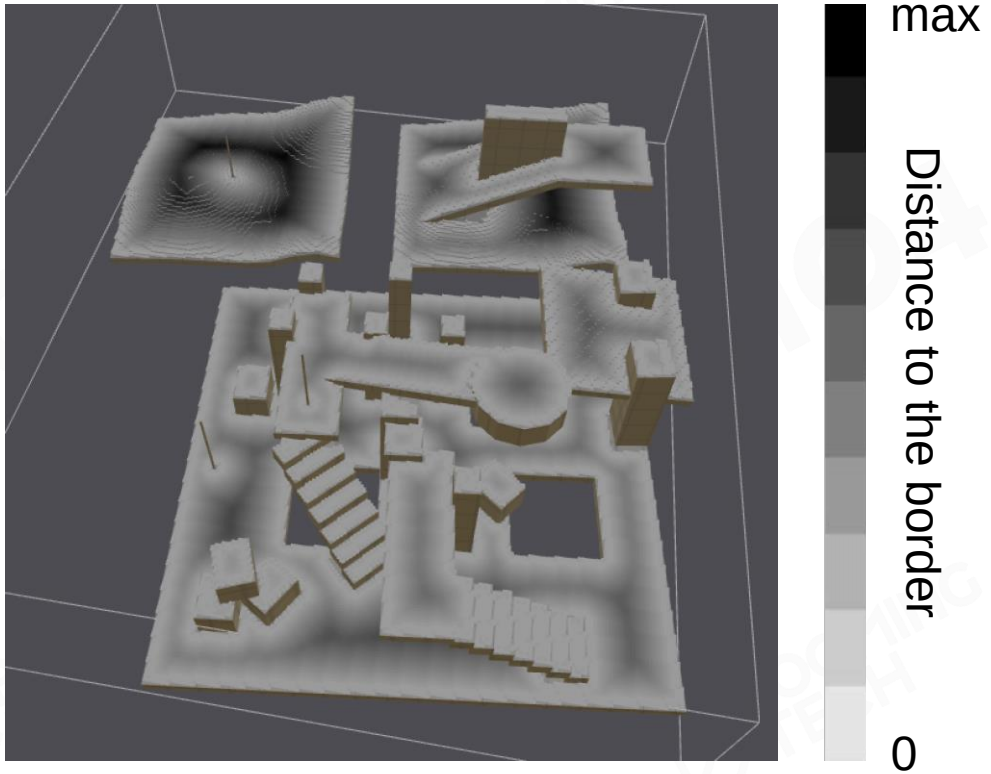


NavMesh Generation – Region Segmentation (1/4)

- Calculate the distance of each voxel to border
- Mark border voxels by AgentRadius to avoid clipping

Mark border voxels

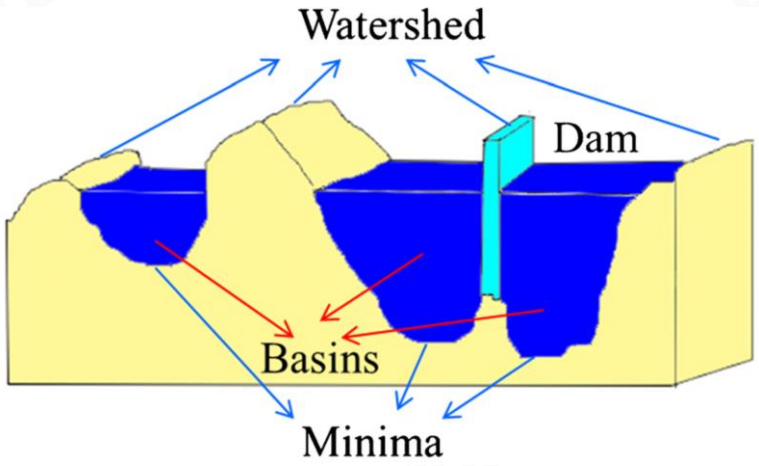


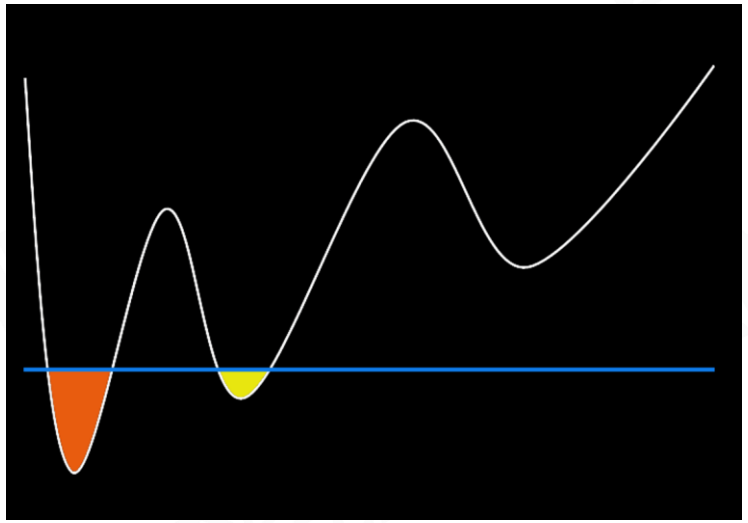


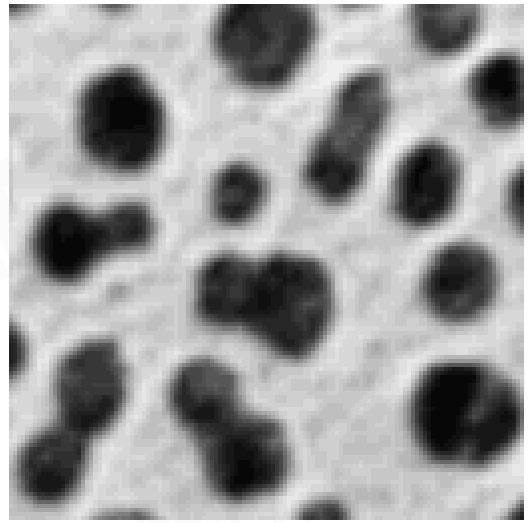
NavMesh Generation – Region Segmentation (2/4)

Watershed Algorithm

- Gradually “flood” the “terrain”
- Form “watershed” (dividing ridge) when “pools” meet

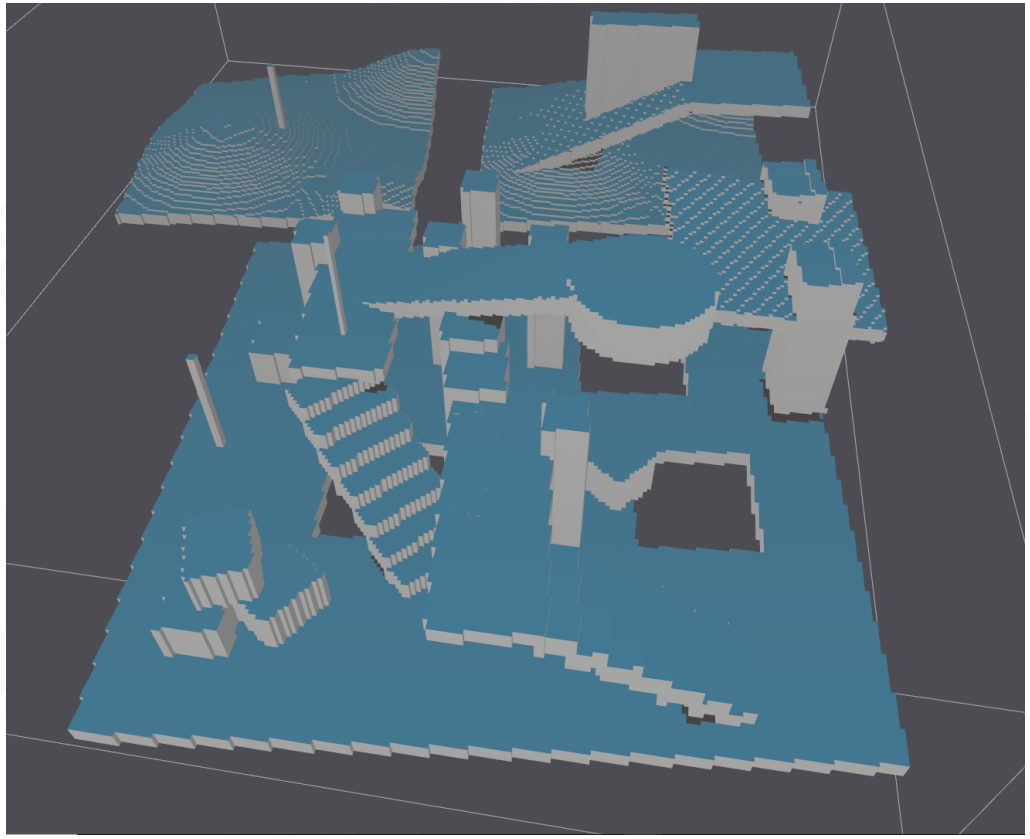


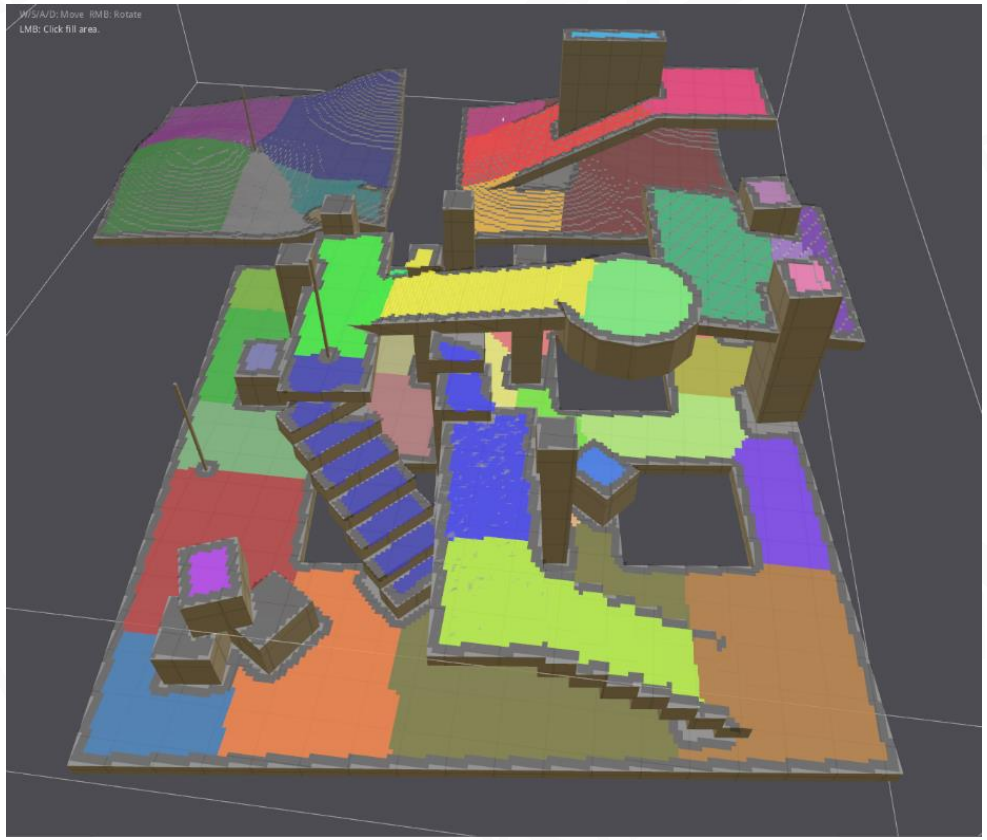




NavMesh Generation – Region Segmentation (3/4)

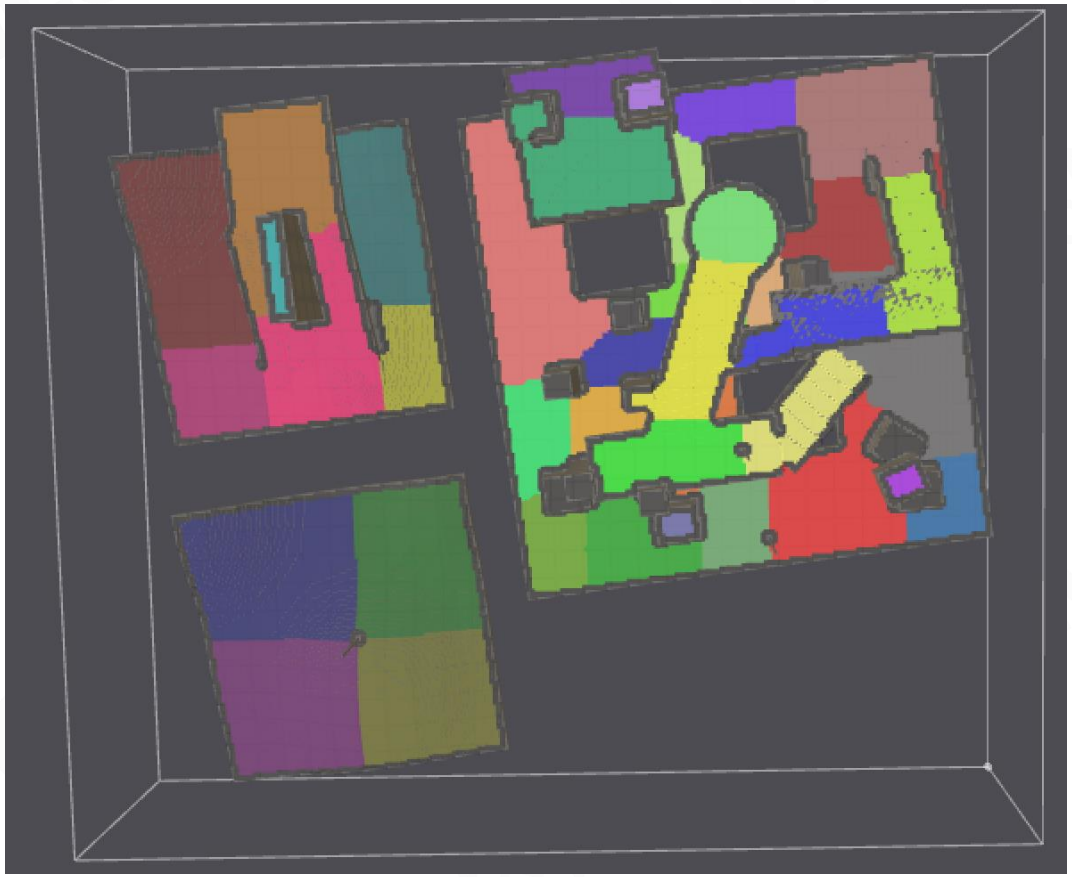
Segment the “neighboring” voxels into regions to provide a good basis for polygon mesh





NavMesh Generation – Region Segmentation (4/4)

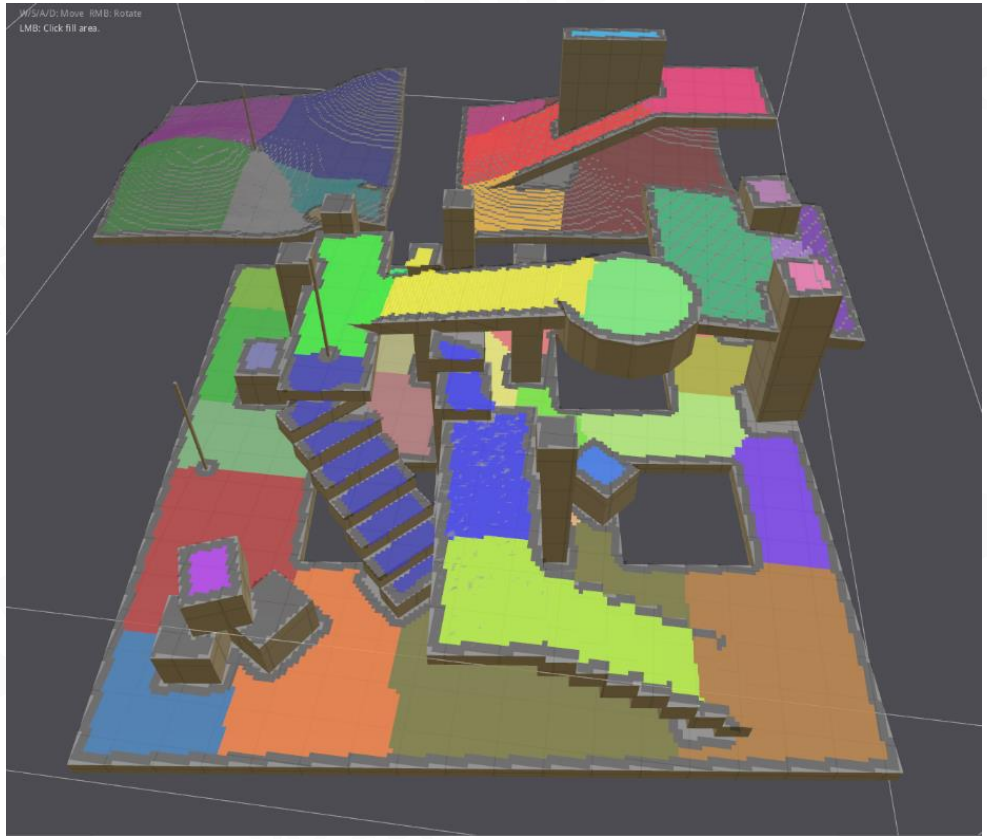
Regions don’t have overlapping voxels in 2D

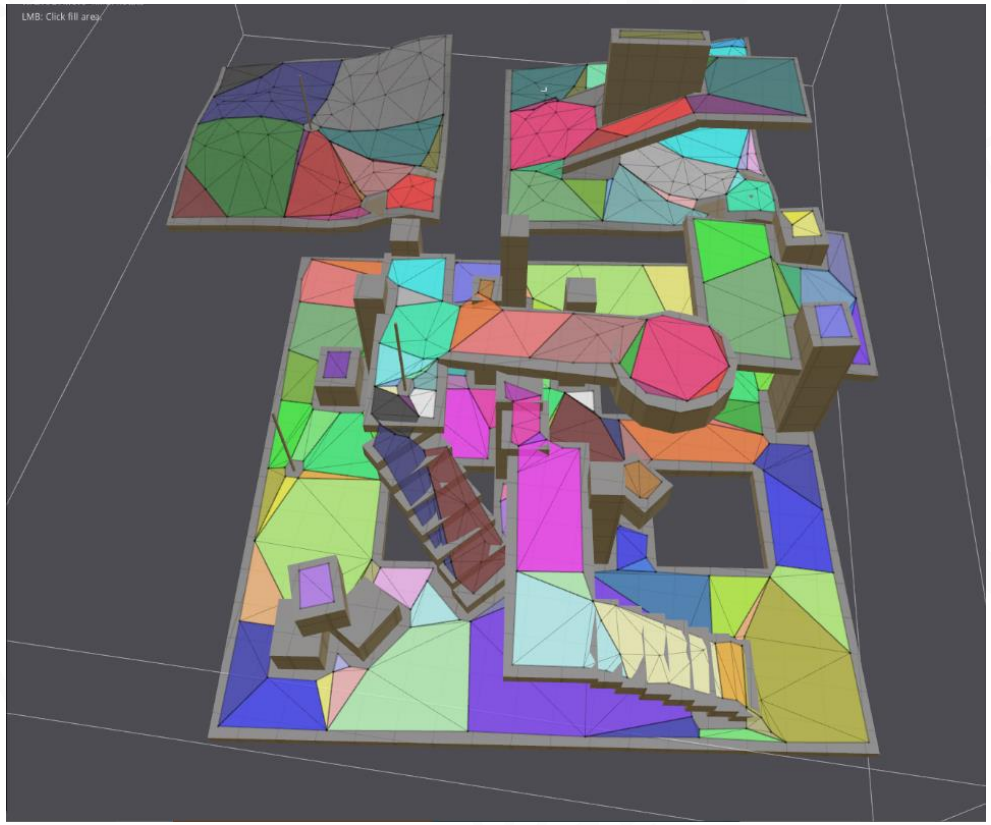


Top-down view of regions

NavMesh Generation – Mesh Generation

Generate NavMesh from segmented regions

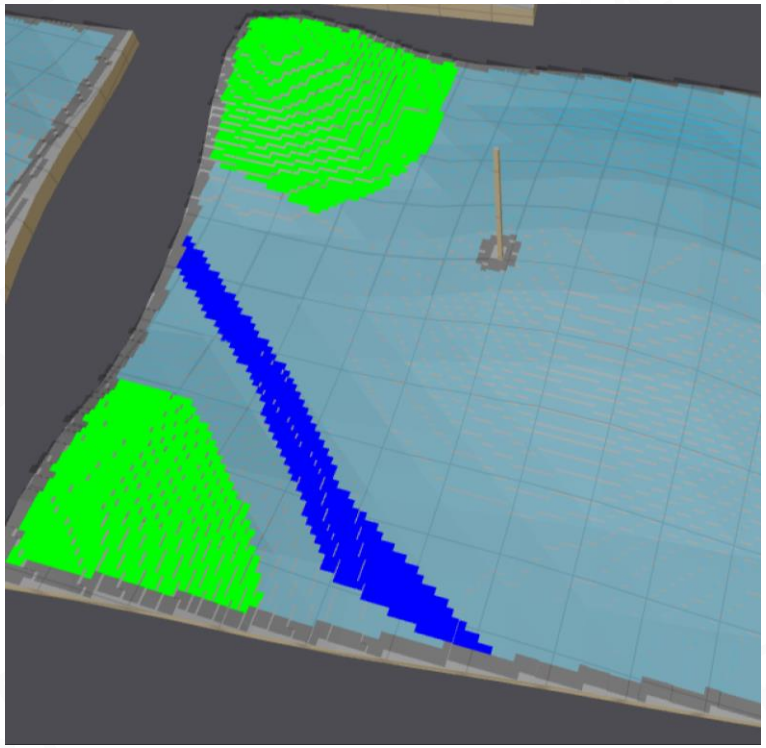


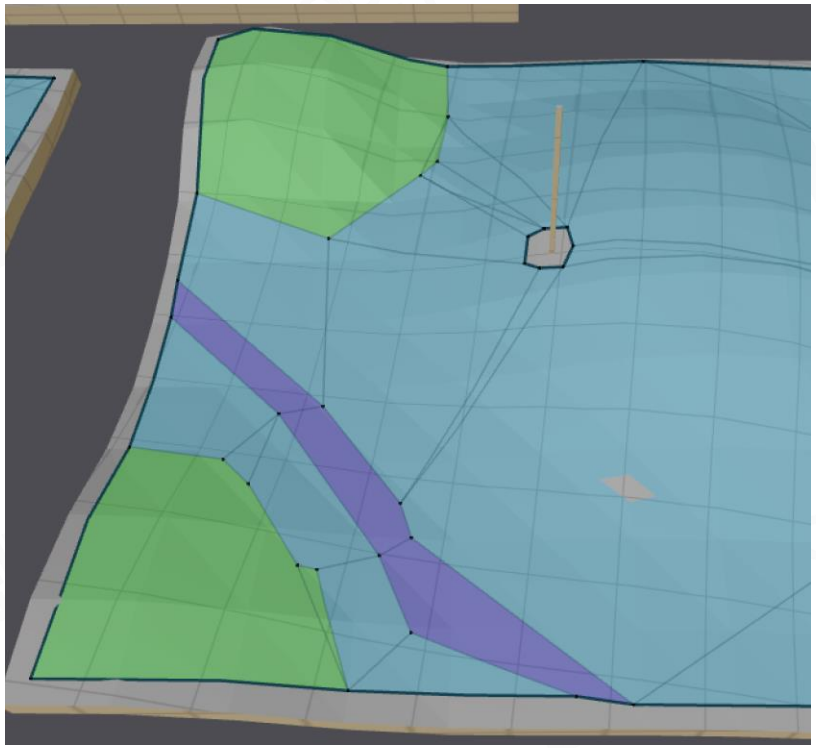


NavMesh Advanced Features – Polygon Flag

Useful for marking terrain types: plains, mountain, water, etc.

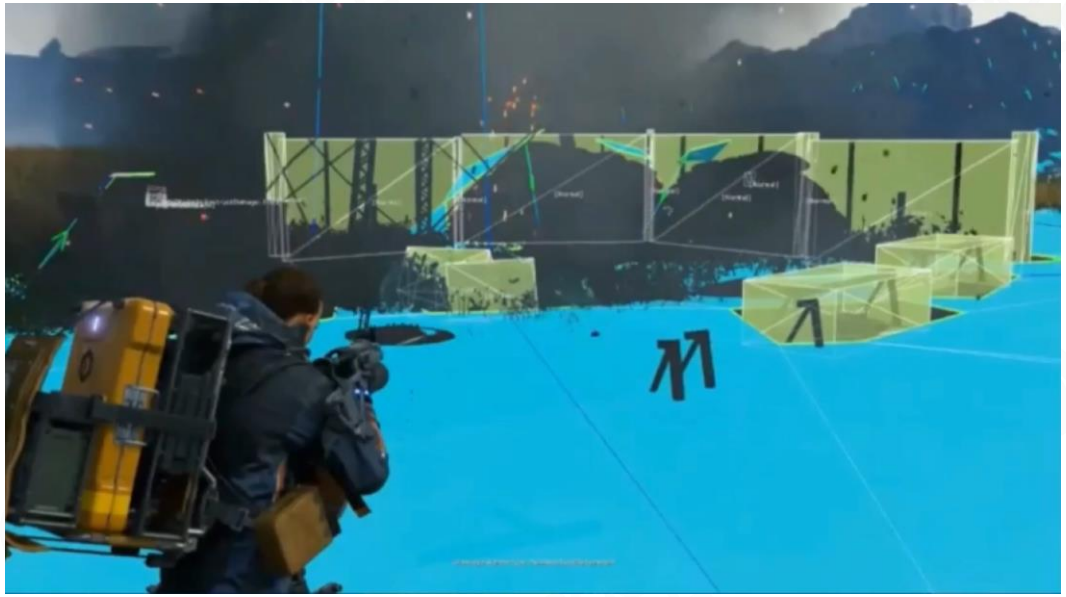
- “Paint colors” to add user-defined regions
- Polygons generated from user-defined regions have special flag





NavMesh Advanced Features – Tile

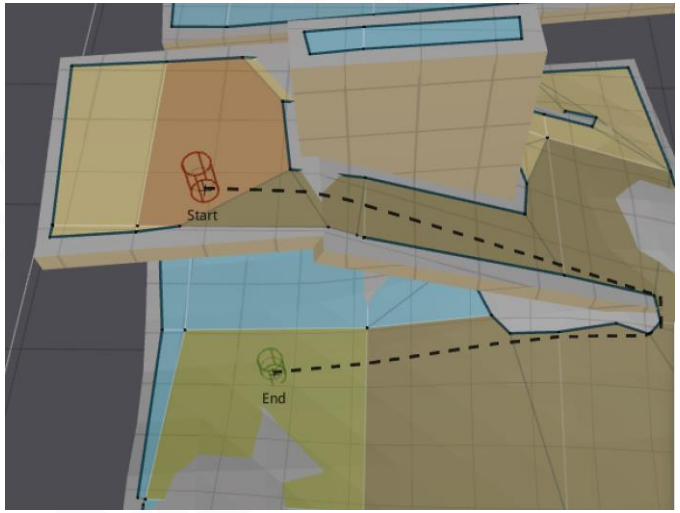
- Fast for responding to dynamic objects
- Avoid rebuilding the entire NavMesh
- TileSize – trade-off between pathfinding and dynamic rebuilding performance



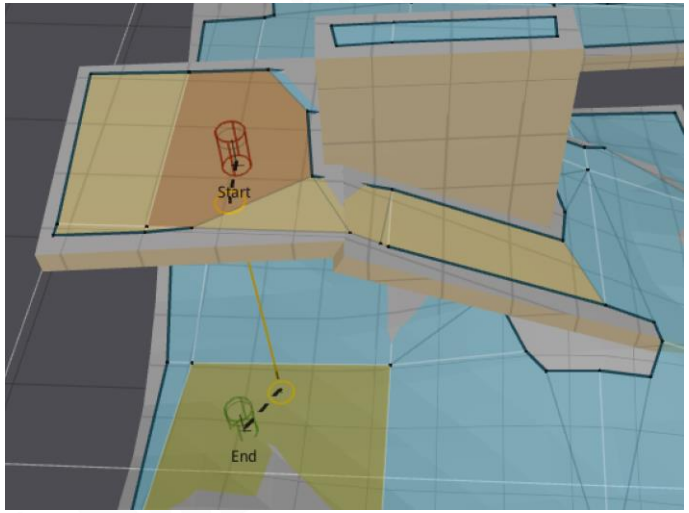
Death Stranding

NavMesh Advanced Features – Off-mesh Link

Allow agents to jump or teleport



without off-mesh link



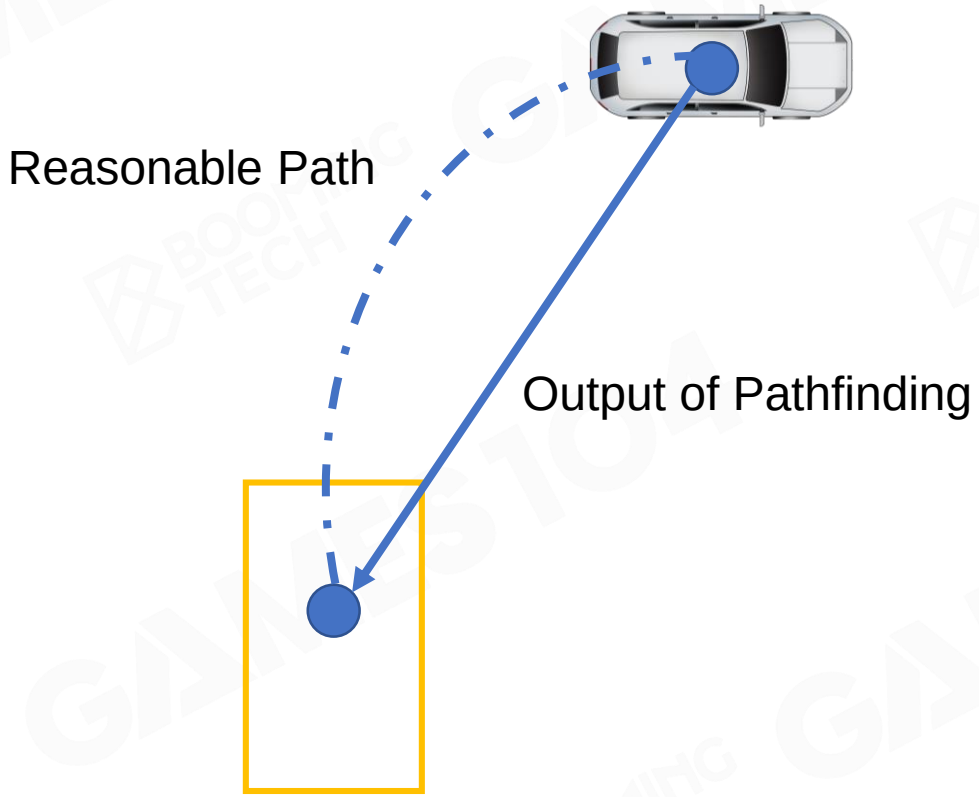
with off-mesh link

jump in Death Stranding

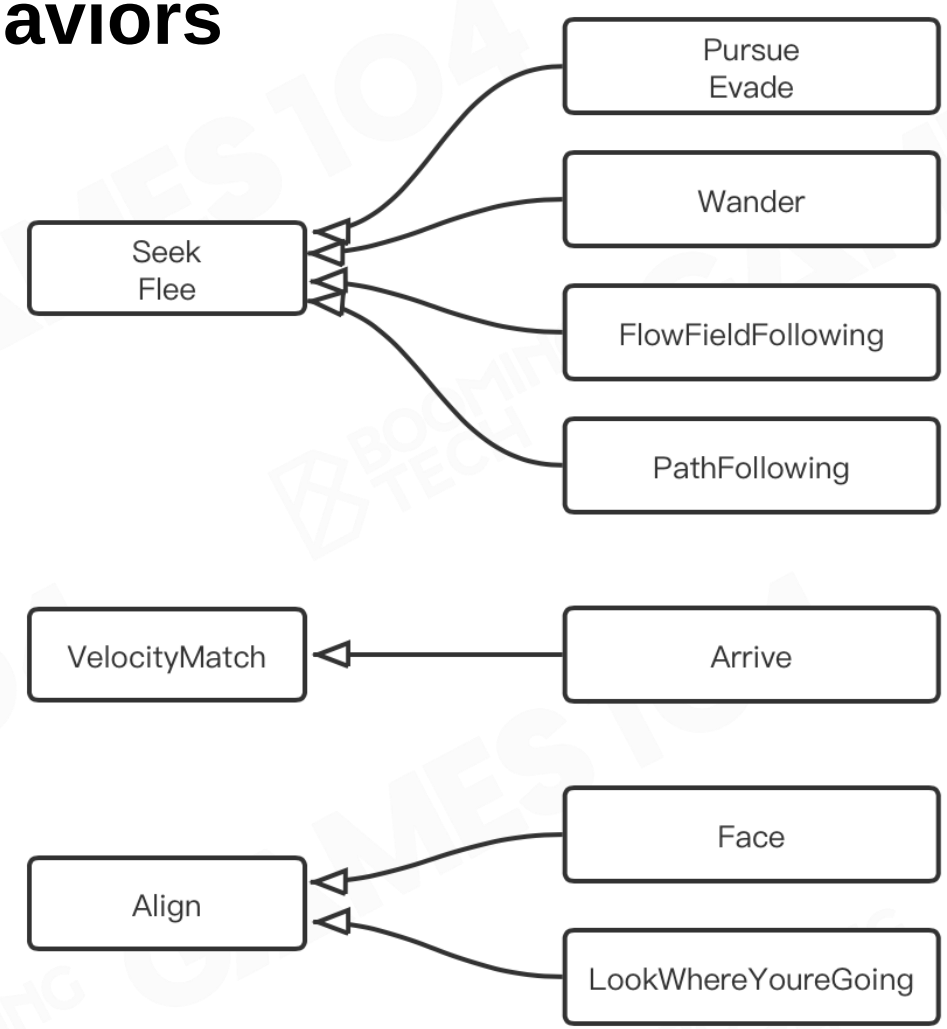
Steering

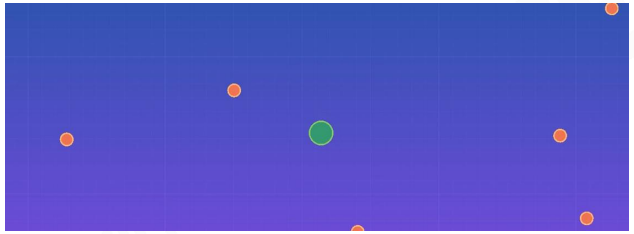
From Path to Motion

- Cars cannot follow planned path exactly
- Motion of cars are limited by theirs motion abilities:
  - Linear acceleration (throttle/brake)
  - Angular acceleration (steering force)
- Motion needs to be adjusted according to the limits



Steering Behaviors





Seek / Flee

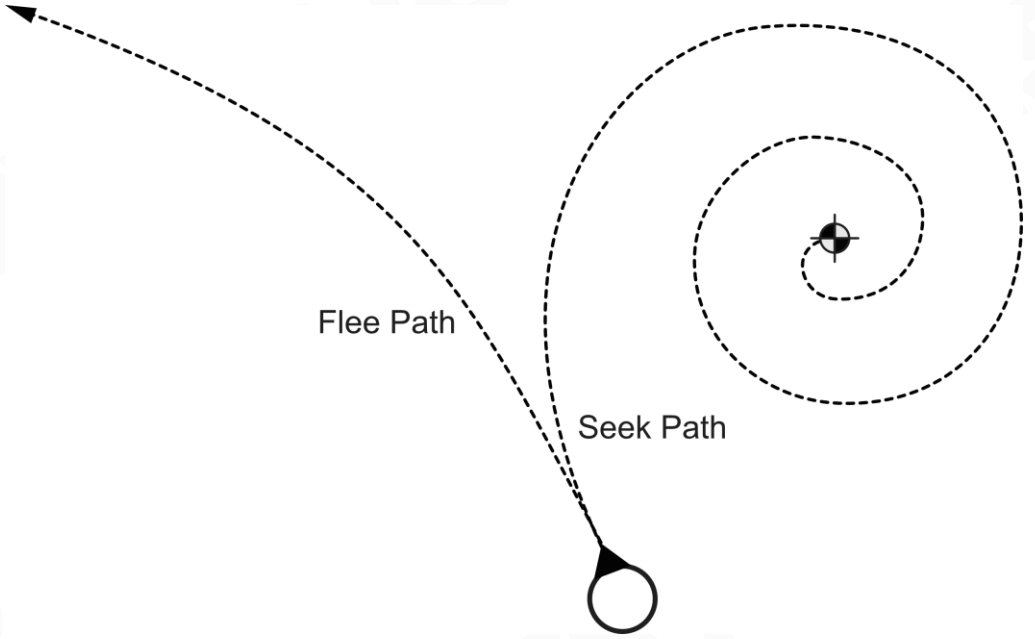
Velocity Match

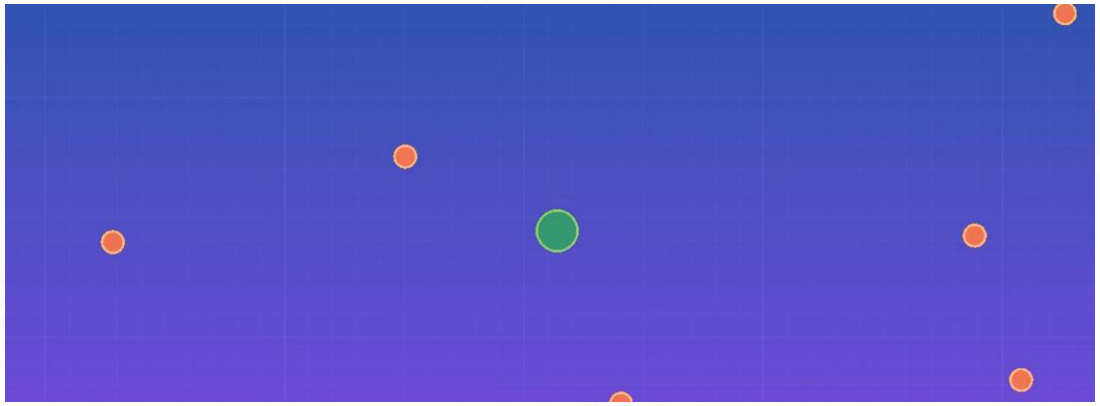
Align

Seek / Flee

Steer the agent towards / away from the target

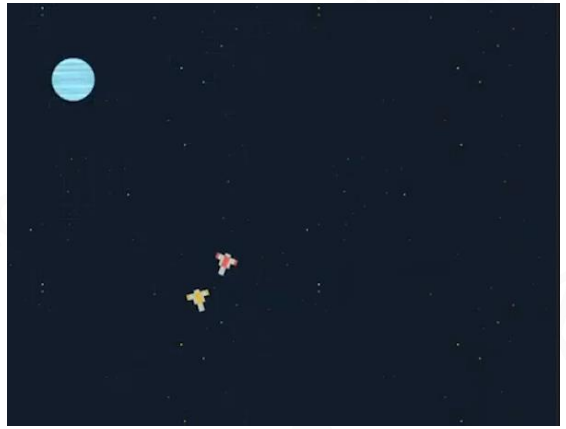
- Position matching in the nature
- Accelerate with max acceleration towards / away from the target
- Will oscillate around the target
- Input:
  - Self position
  - Target position
- Output:
  - Acceleration



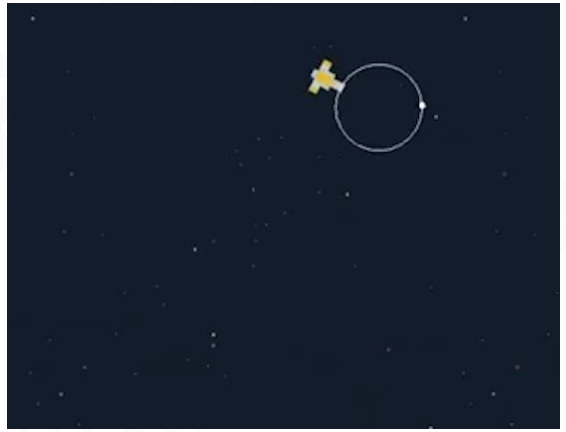


Seek / Flee Variations

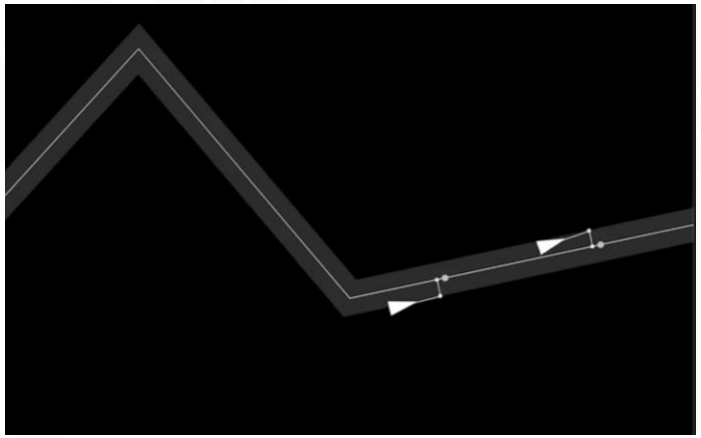
Modifying the target in runtime can generate new steering behaviors



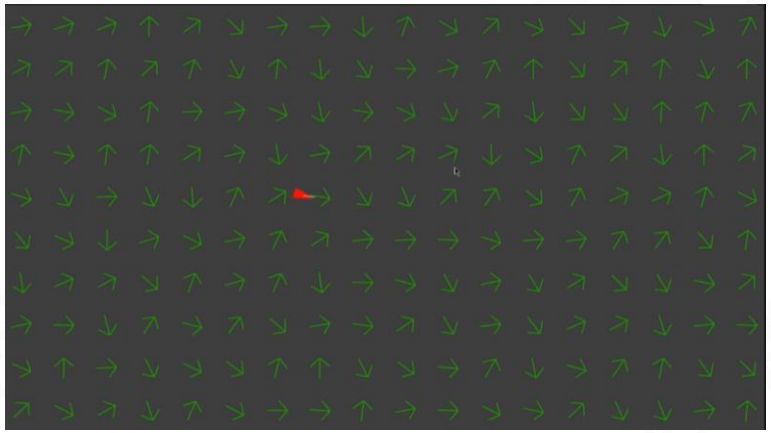
Pursue



Wander



Path Following

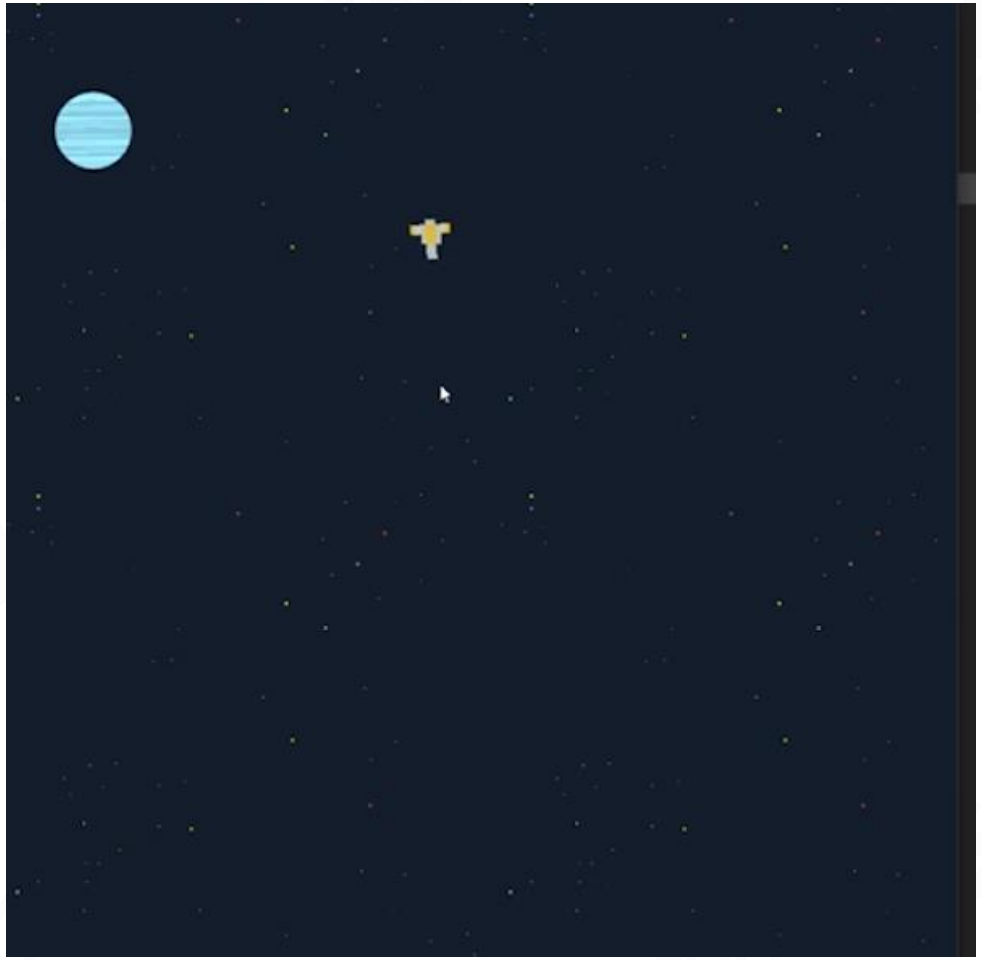


Flow Field Following

Velocity Match

Matches the target velocity

- Calculate acceleration from matching time and velocity differences
- Clamp the acceleration by maximum acceleration of agents
- Input:
  - Target velocity
  - Self velocity
  - Matching time
- Output:
  - Acceleration

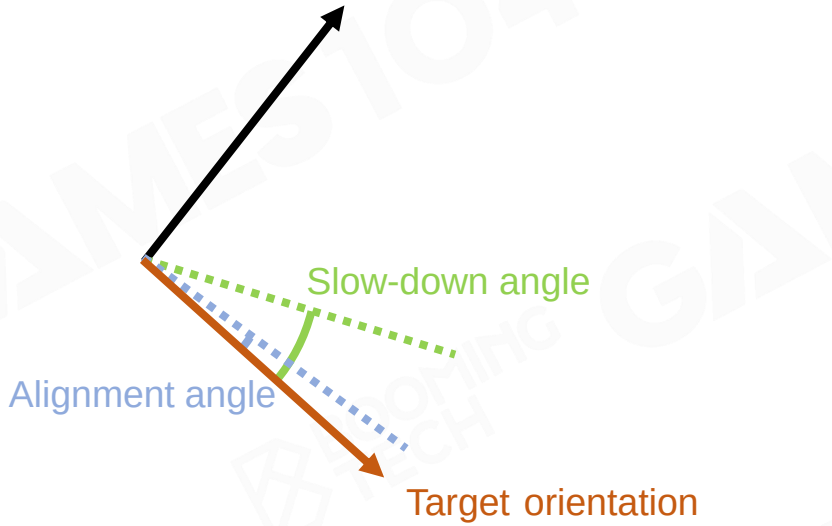


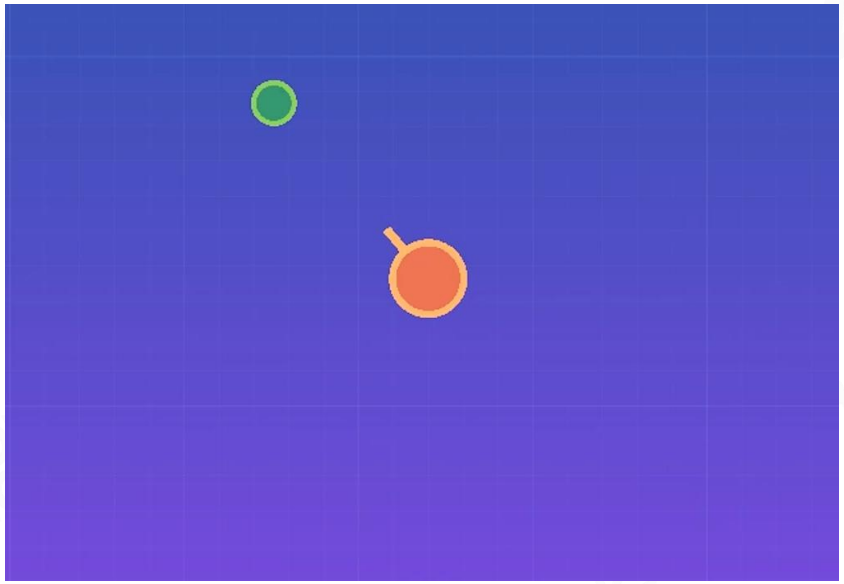
Arrive

Align

Matches target orientation

- Input:
  - Target orientation
  - Self orientation
- Output:
  - Angular acceleration





Crowd Simulation

Crowd

A large group of individuals share information in the same environment alone or in a group

- Collision avoidance
- Swarming
- Motion in formation
- ……



Crowd Simulation Models

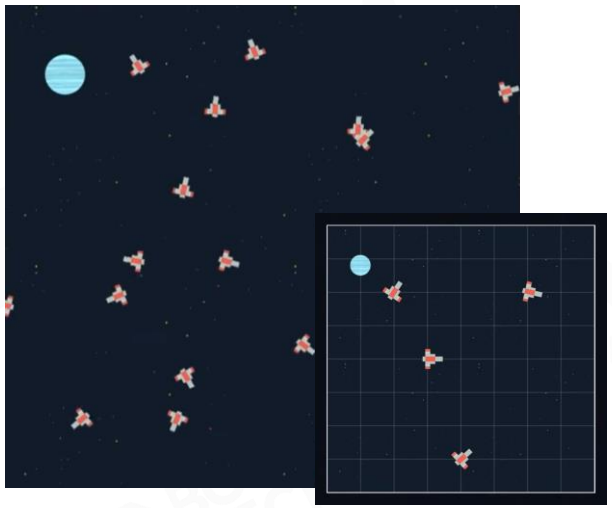
- Started from “Boids” system of Reynolds
- Three families of models:
  - Microscopic models
    - “Bottom-Up”
    - Focus on individuals
  - Macroscopic models
    - Crowd as a unified and continuous entity
  - Mesoscopic models
    - Divide the crowd into groups

C. W. Reynolds
Flocks, Herds, and Schools: A Distributed Behavioral Model

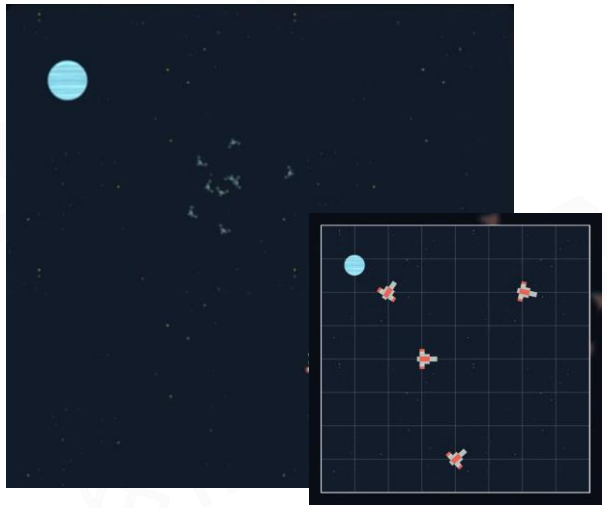
Microscopic Models – Rule-based Models

Flock dynamics of animal crowds as an emergent behavior by modeling motion of each individuals with simple predefined rules:

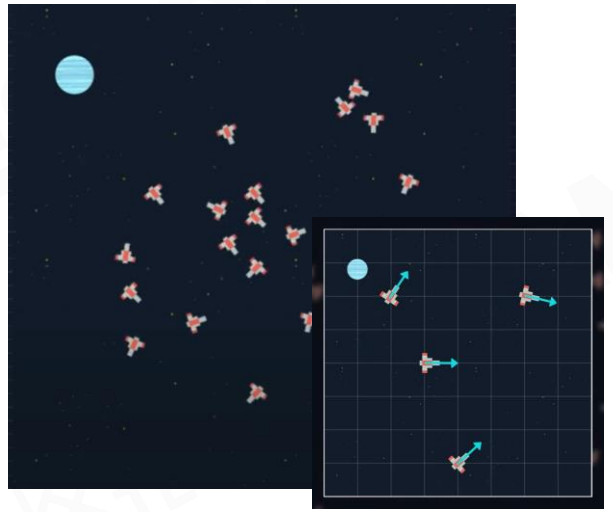
- Separation: to steer away from all of its neighbors
- Cohesion: to steer towards the “center of mass”
- Alignment: to line up with agents close by



Separation



Cohesion



Alignment

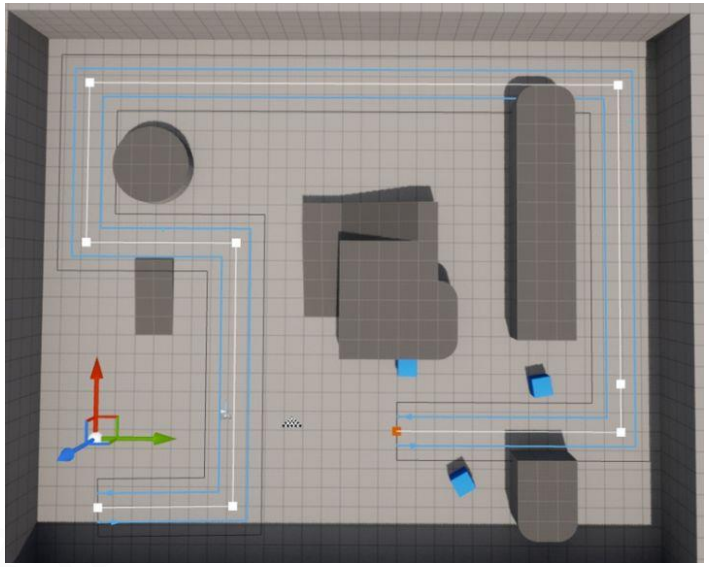
Microscopic Models – Rule-based Models

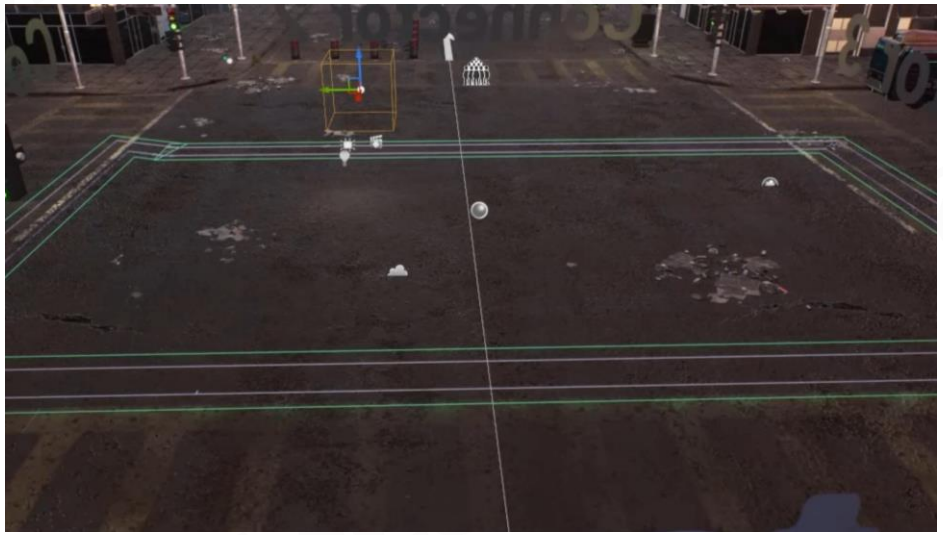
Easy to implement, but not suitable to simulate complex behavior rules

Macroscopic Models

Simulate crowd motion from a macro perspective

- Treat the crowd as a unified and continuous entity
- Control motions with potential field or fluid dynamics
- Does not consider interactions between individuals and the environment in individual level



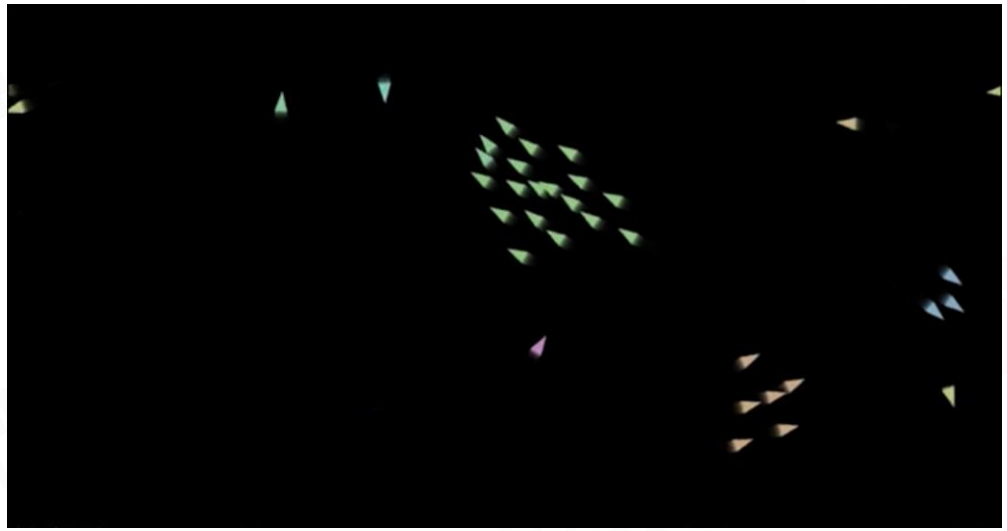


Flow Field In UE5 MassAI

Mesoscopic Models

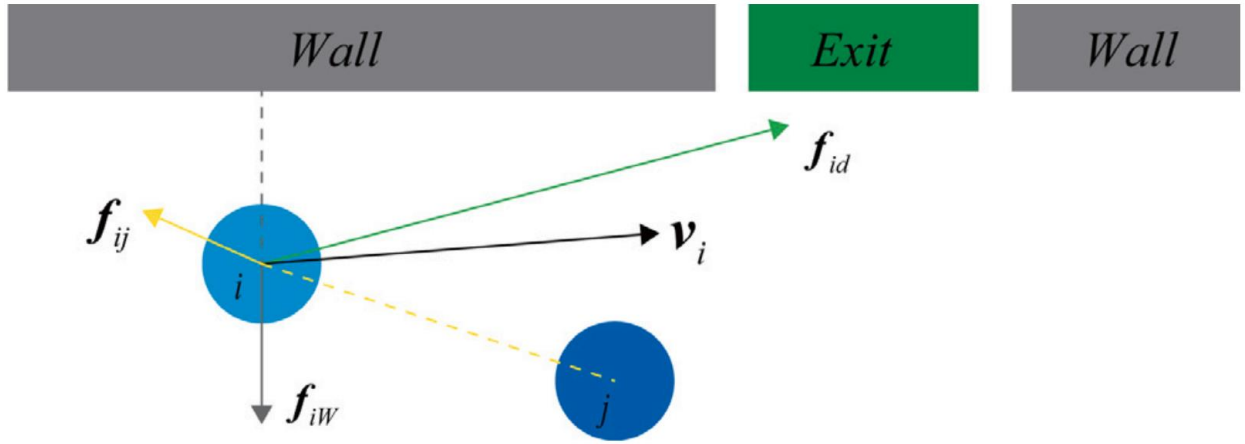
Simulate crowd motion taking care of both details and the whole

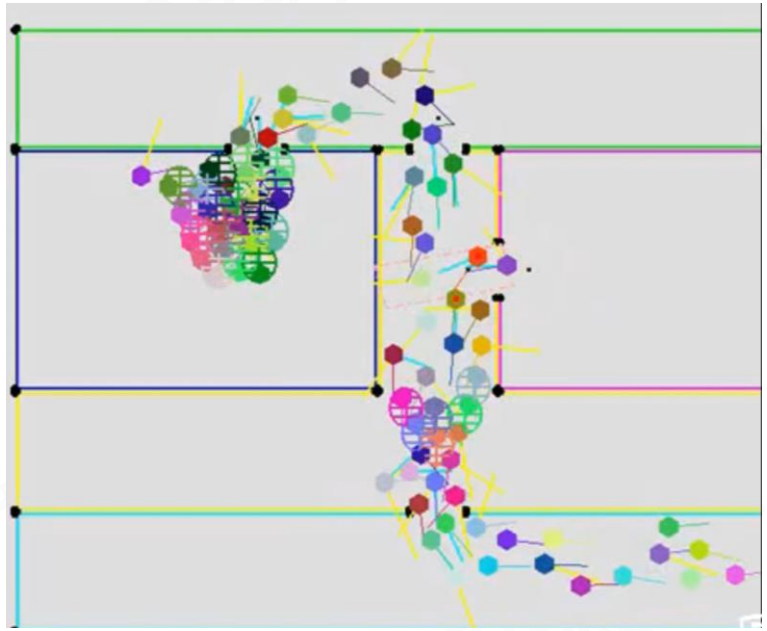
- Divide the crowd into groups
- Deals with interactions between groups and individuals in each group
- combinations of microscopic models and formation rules or psychological models



Collision Avoidance – Force-based Models

- A mixture of socio-psychological and physical forces influencing the behavior in a crowd
- The actual movement of an individual depends on the desired velocity and its interaction with the environment
- Can simulate dynamical features of escape crowd panic





Collision Avoidance – Force-based Models

Pros:

- can be extended to simulate more emergent behaviors of human crowds

Cons:

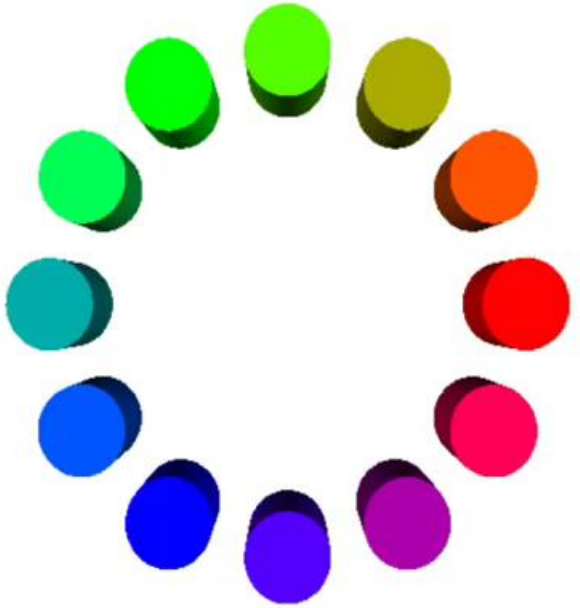
- Similar to physics simulation, simulation step should be small enough

Collision Avoidance – Velocity-based models

Consider the neighbor information to make decisions in velocity space

- able to simulate in local space
- applied to collision avoidance

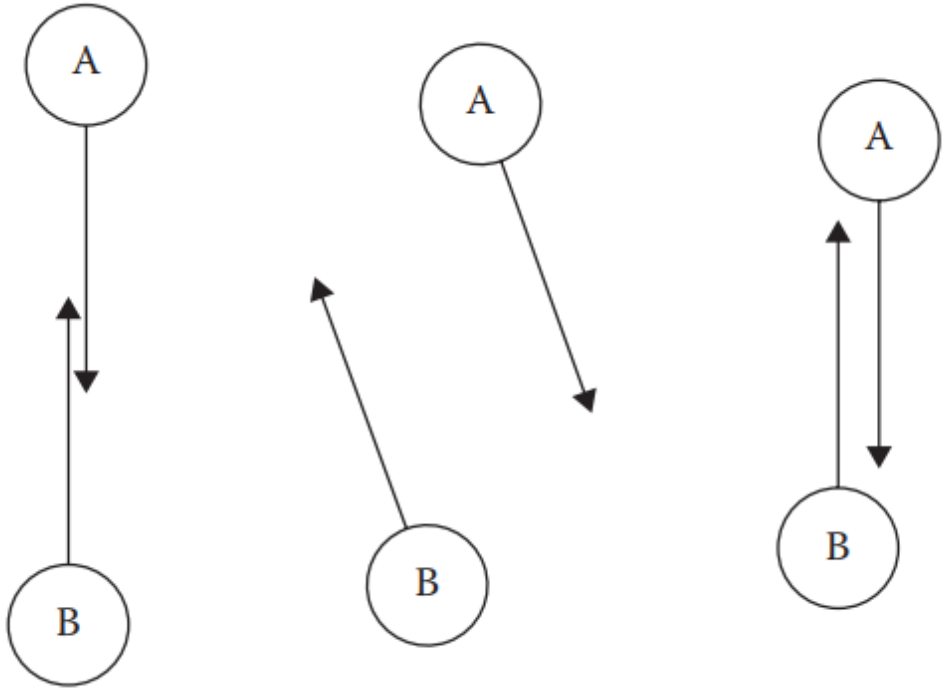
Reciprocal Velocity obstacle methods – Current standard collision avoidance algorithms



- Velocity Obstacle (VO)
- Reciprocal Velocity Obstacle (RVO)
- Optimal Reciprocal Collision Avoidance (ORCA)

Velocity Obstacle (VO)

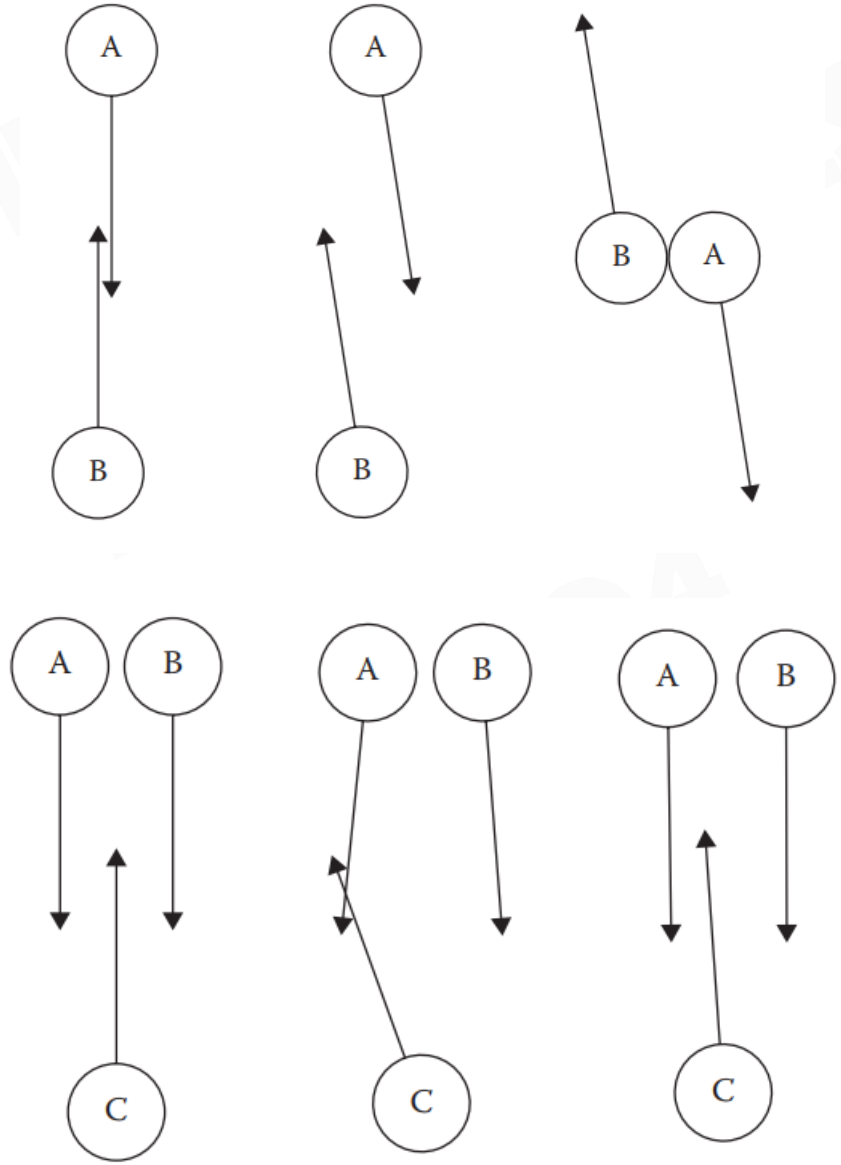
- Calculate its own dodge velocity, assuming other agent is unresponsive
- Appropriate for static and unresponsive obstacles
- Overshoot
- Causes oscillation between two agents attempting to avoid each other



Reciprocal Velocity Obstacle (RVO)

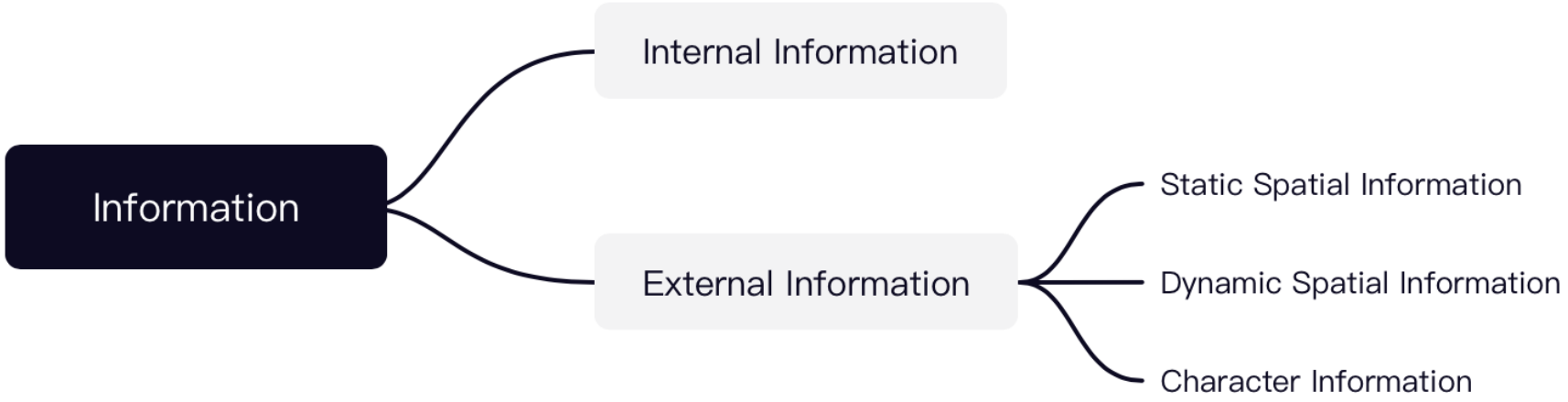
- Assuming the other agent is using the same decision process (mutually cooperating)
- Both sides move half way out of the way of a collision
- Only guarantees no oscillation and avoidance for two agents

Optimal Reciprocal Collision Avoidance (ORCA)



Sensing

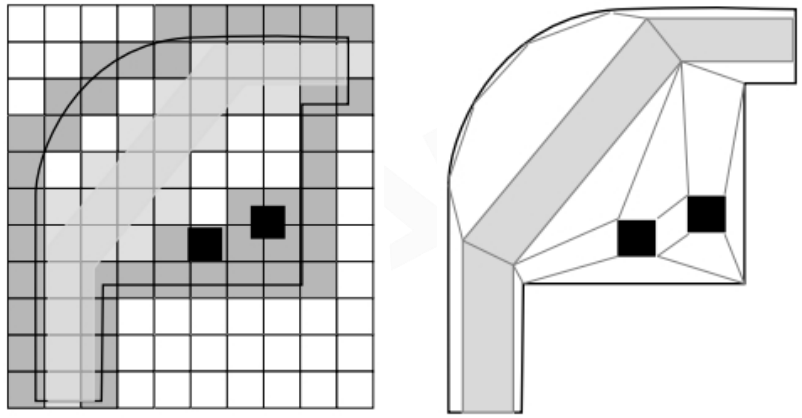
Sensing or Perception



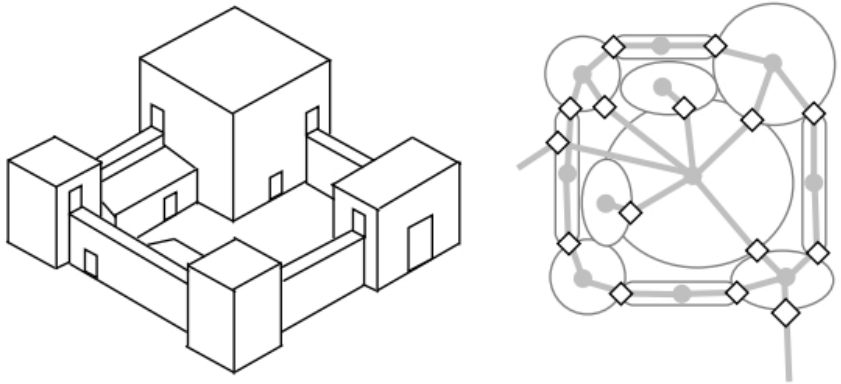
Internal Information

- Information of the agent itself
  - Position
  - HP
  - Armor status
  - Buff status
  - …
- Can be accessed freely

Static Spatial Information



Navigation Data

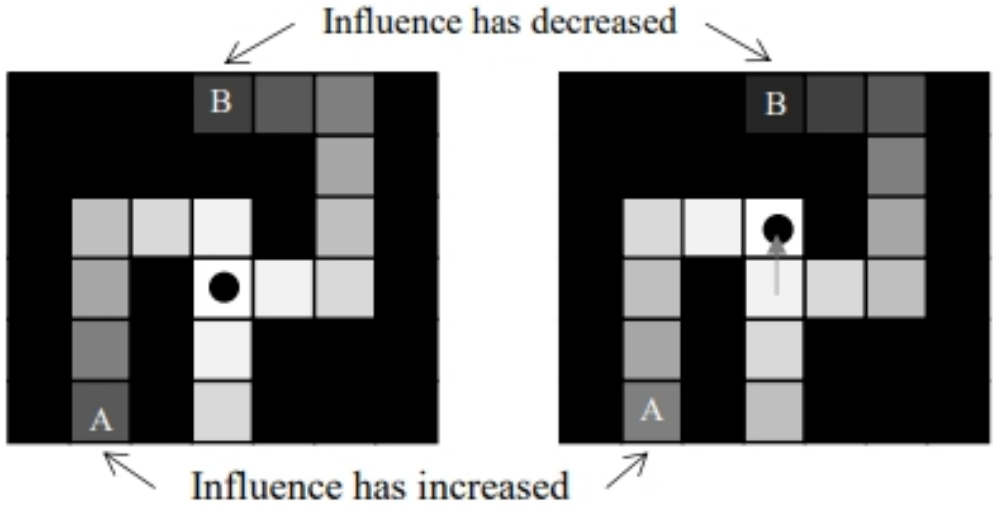


Tactical Map

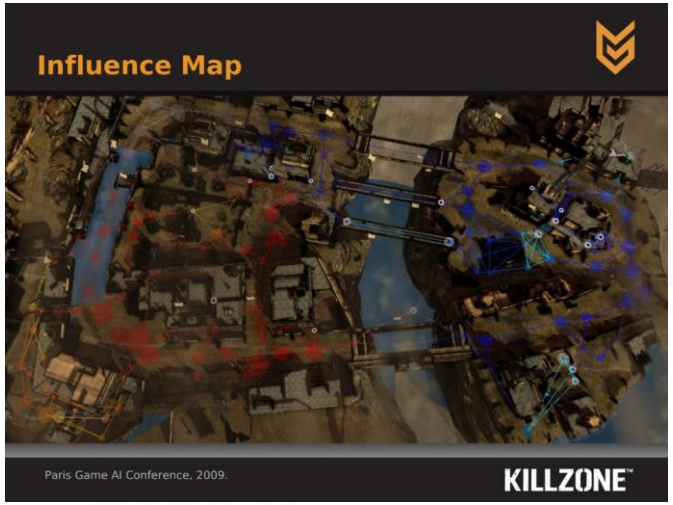
Smart Object

Cover Point

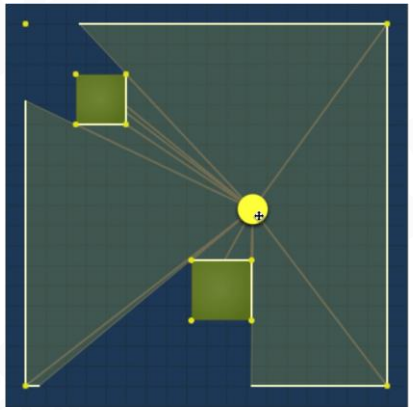
Dynamic Spatial Information (1/2) – Influence Map

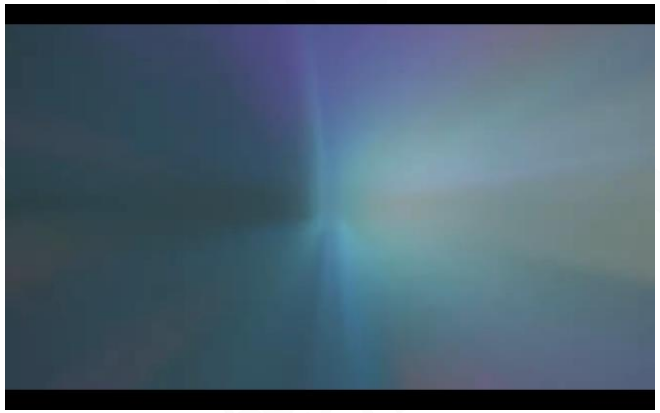


Marks on navigation data



Influence Map





Sight Area

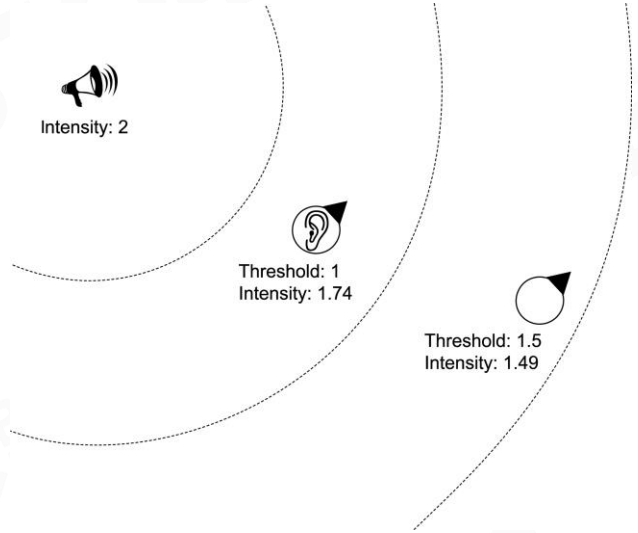
Dynamic Spatial Information (2/2) – Game Objects

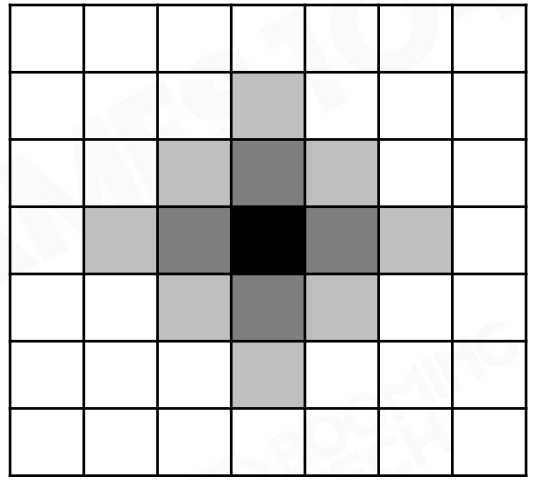
- Information being sensed from a character
- Multiple character information can exist for a single character

  as it can be sensed by multiple agents
- Usually contains:
  - Game Object ID
  - Visibility
  - Last Sensed Method
  - Last Sensed Position

Sensing Simulation

- Light, sound, and odor travels in space
- Have max traveling range
- Attenuates in space and time with different patterns
  - Sight is blocked by obstacles
  - Smelling ranges shrinks over time
- Radiating field can simulate sensing signals
  - Can be simplified as Influence Map
  - Agents covered by the field can sense the information





Classic Decision Making Algorithms

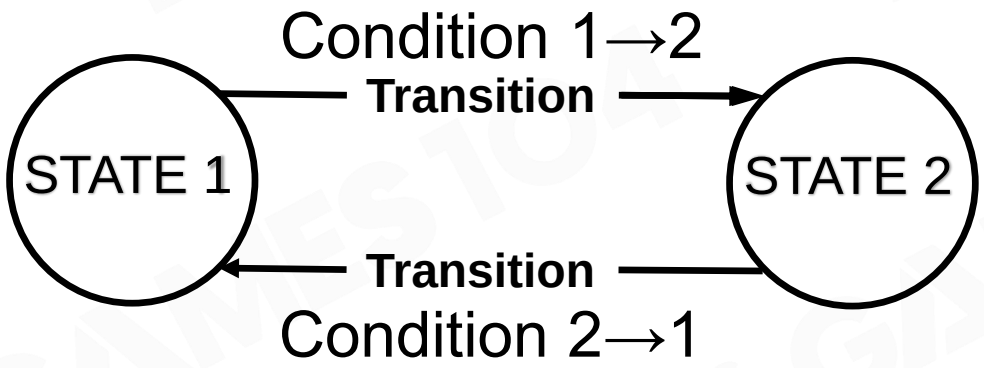
Decision Making Algorithms

- **Finite State Machine**
- **Behavior Tree**

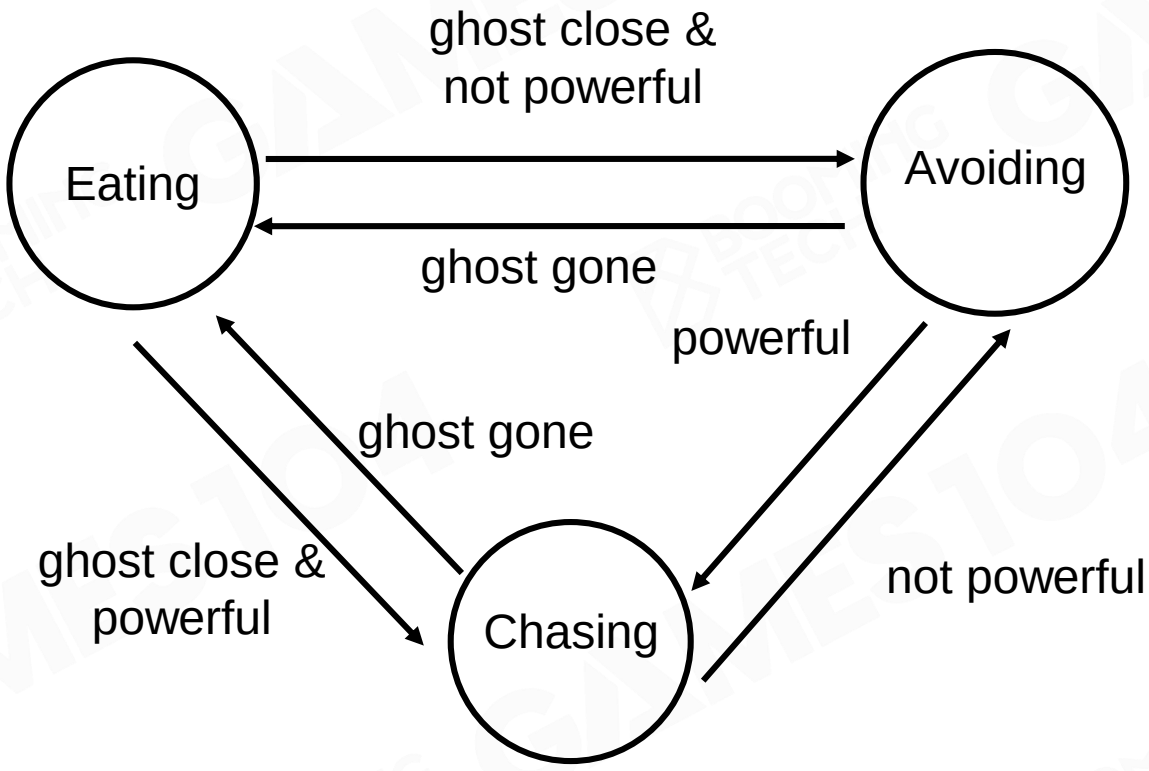
- Hierarchical Tasks Network
- Goal Oriented Action Planning
- Monte Carlo Tree Search
- Deep Learning

Finite State Machine

- Change from one **State** to another according to some **Conditions**
- The change from one state to another is called a **Transition**



Finite State Machine

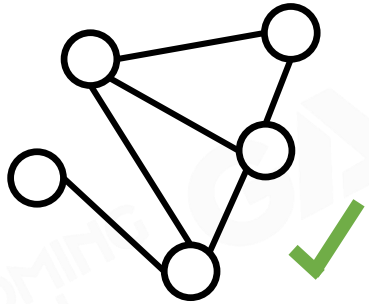


A FSM for Pacman

Finite State Machine – Pros & Cons

Pros

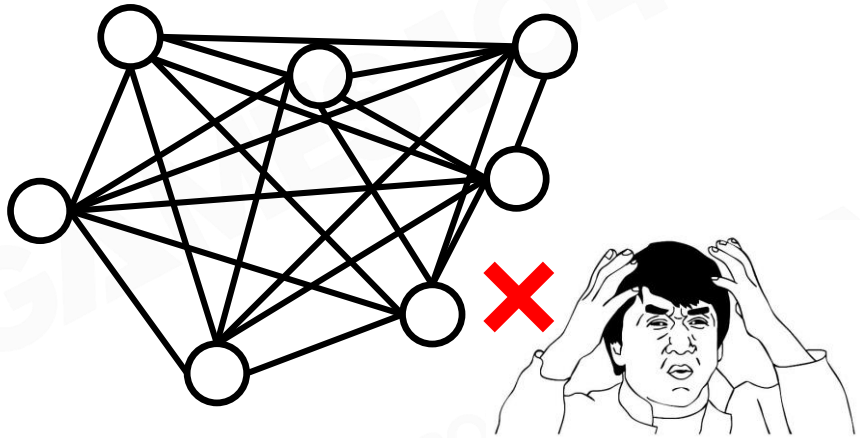
- Easy to implement
- Easy to understand
- Very fast to deal with simple case



For Simple Case

Cons

- Maintainability is bad, especially add or remove state
- Reusability is bad, can’t used in other projects or characters
- Scalability is bad, hard to modify for complicated case

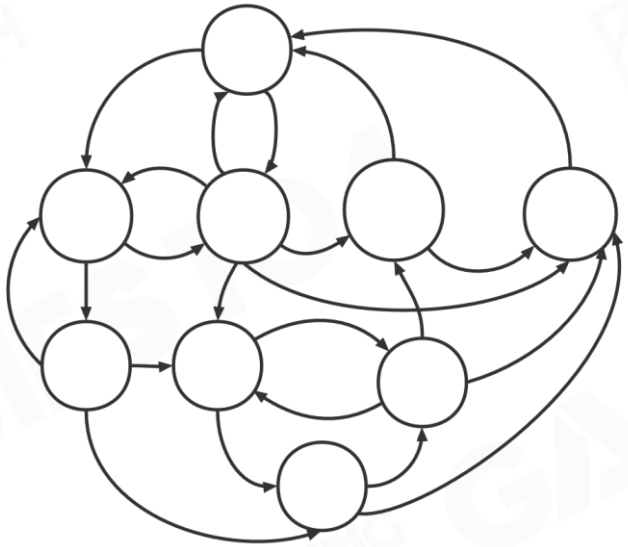


For Complicated Case

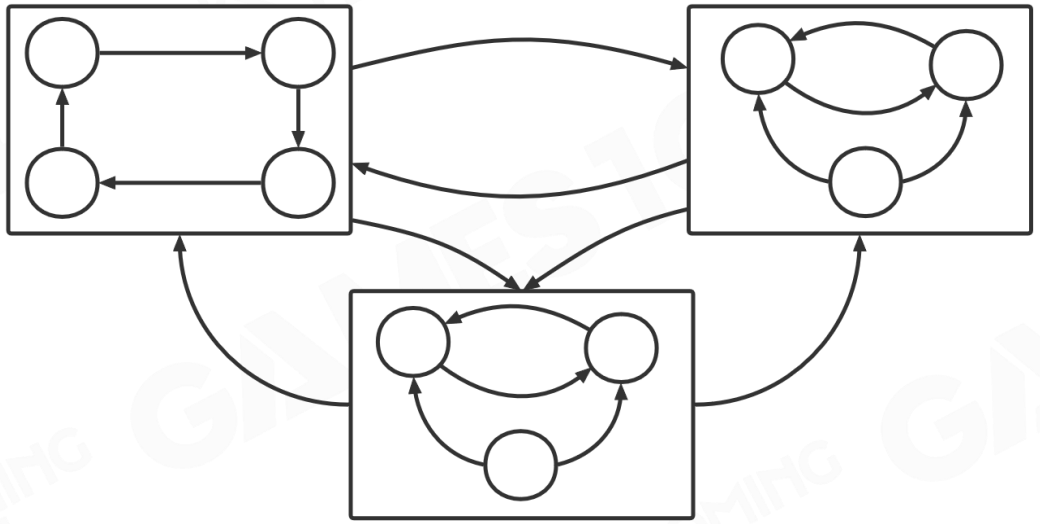
Hierarchical Finite State Machine (HFSM)

Tradeoff between reactivity and modularity

- Reactivity: the ability to quickly and efficiently react to changes
- Modularity:the degree to which a system's components may be separated into building blocks, and recombined



FSM with High Rreactivity

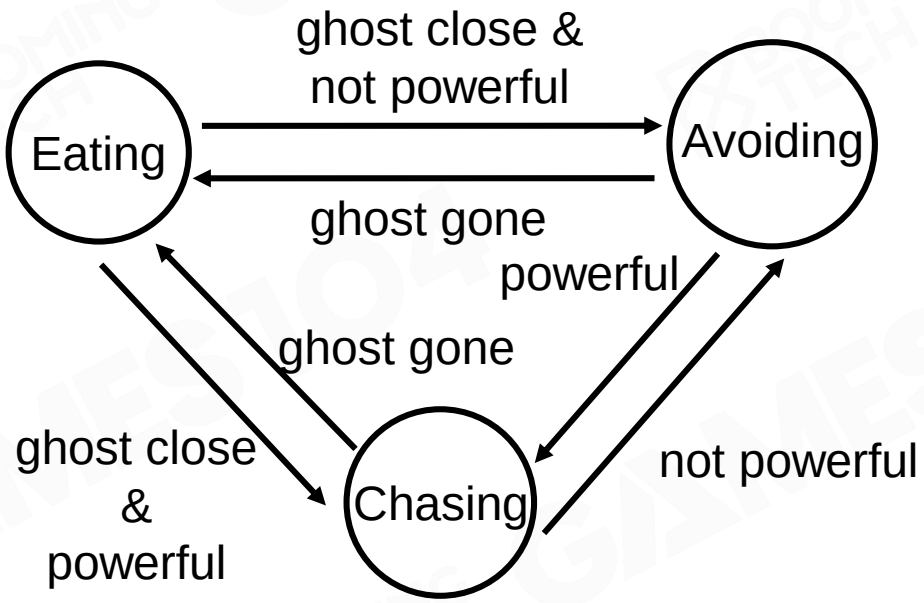


Hierarchical Finite State Machine (HFSM)

Behavior Tree (BT)

Behavior Tree

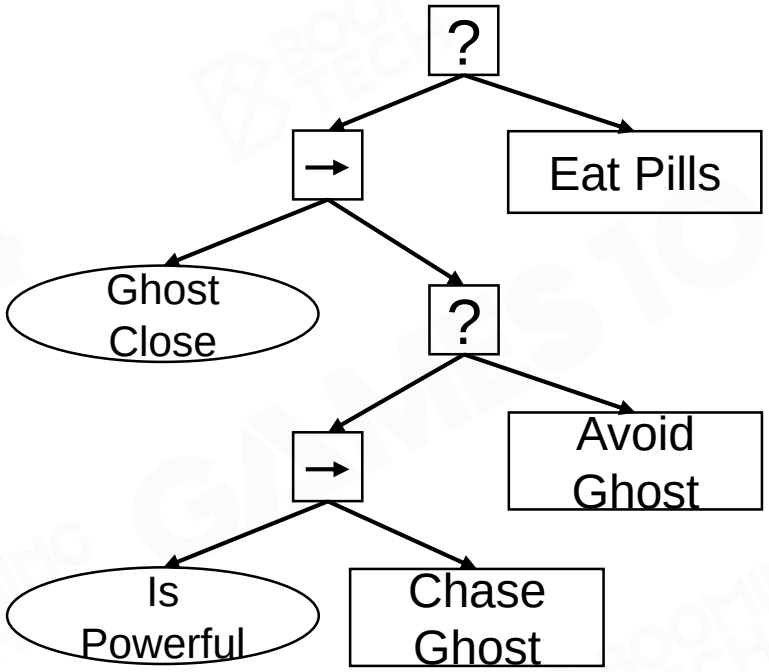
Focus on state abstraction and transition conditions



A FSM for Pacman

Similar to human thinking:

- If ghost close，run away
- But if I'm powerful, chase it
- Otherwise, eating

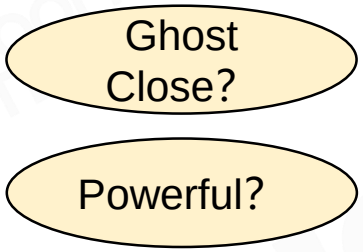


A BT for Pacman

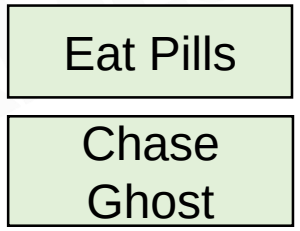
Behavior Tree – Execution Nodes

Execution node (leaf node)

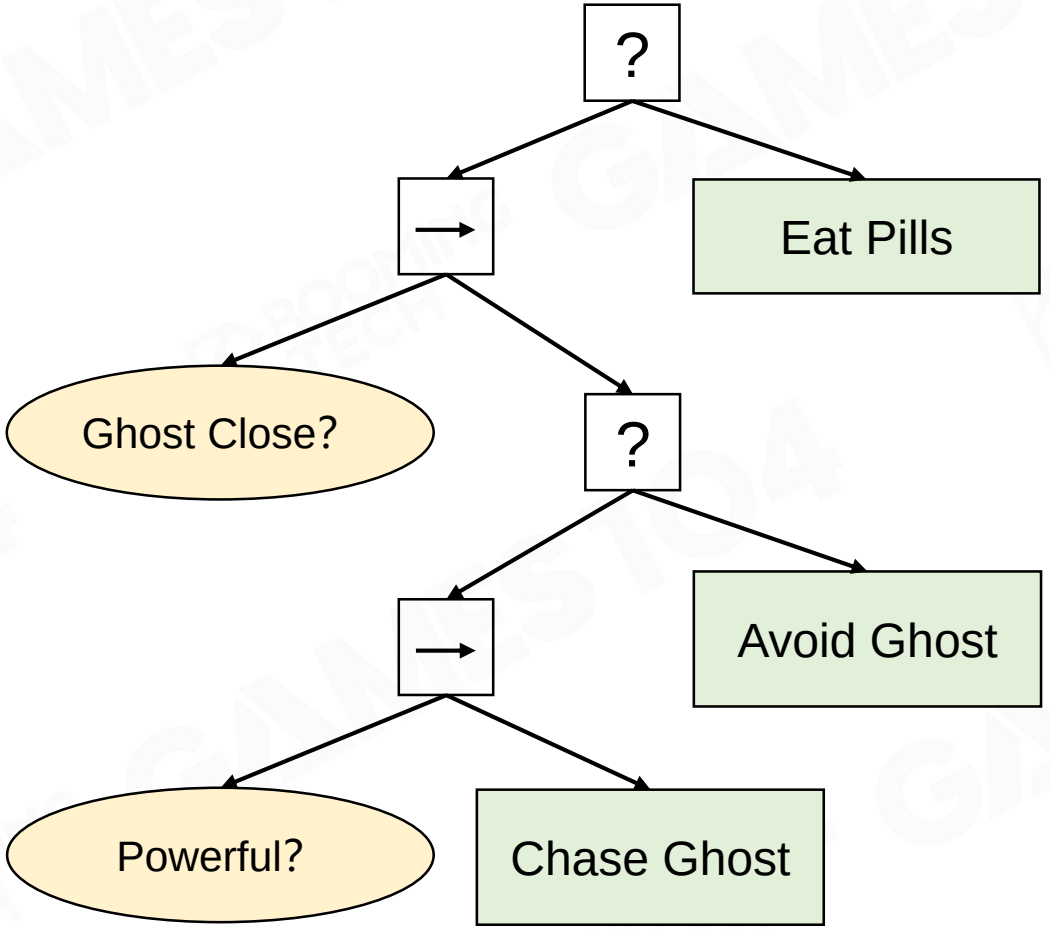
- **Condition node**
- **Action node**



Self states or perceptions
Return true or false



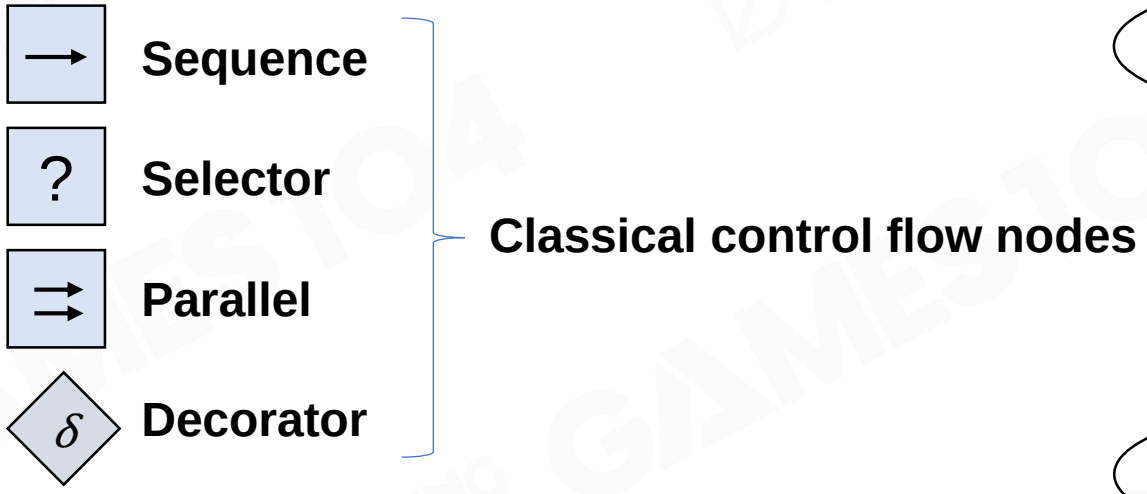
Actions
Return success or failure, running

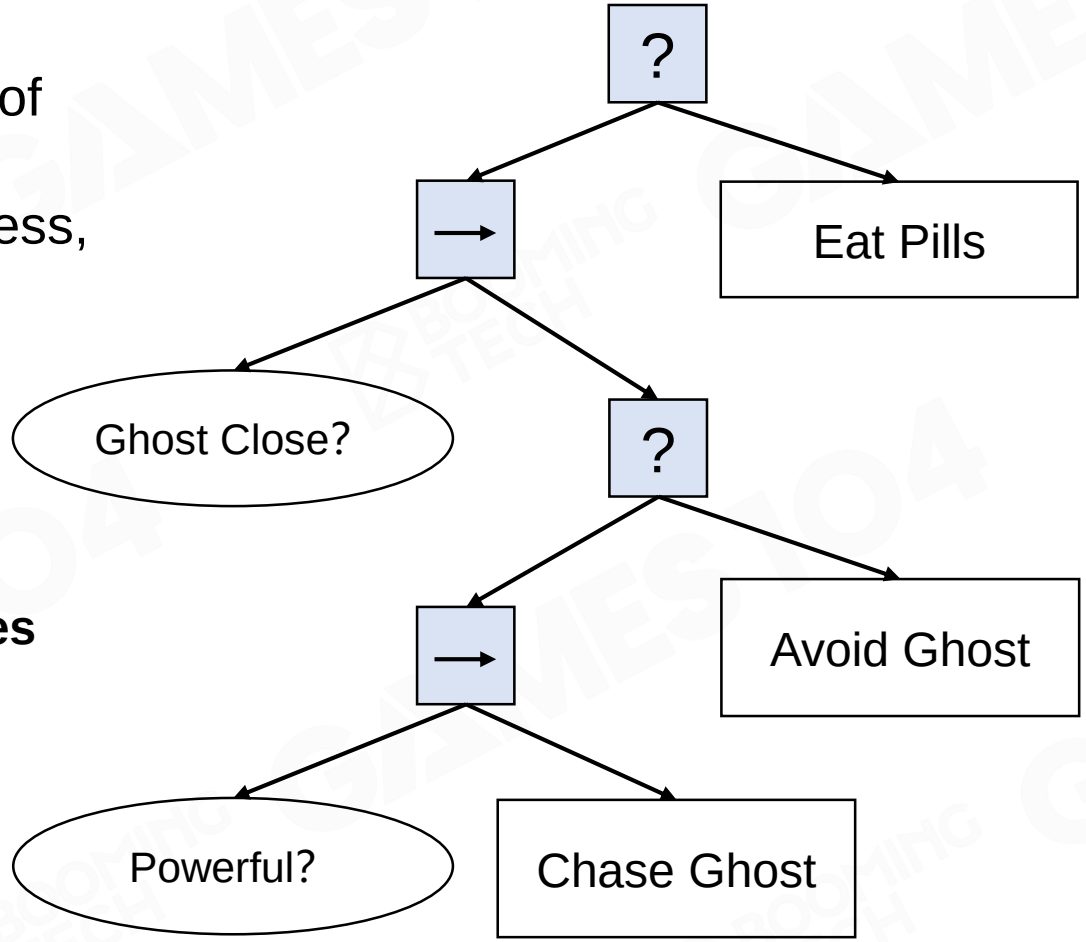


Behavior Tree – Control Nodes

Control flow node (internal node)

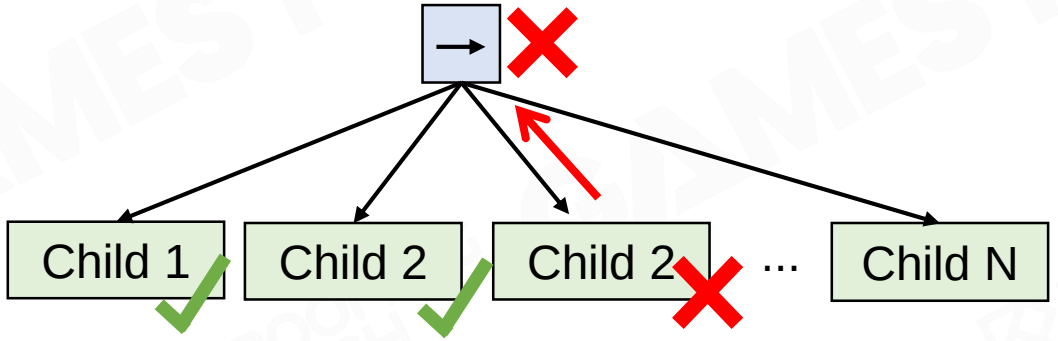
- Control flow determined by the return value of child nodes
- Each node has a return value which is success, failure or running





Control Node – Sequence (1/2)

- Order
  - Execute children from left to right
- Stop Condition and Return Value
  - until one child returns ***Failure*** or ***Running***, then return value accordingly
  - or ***all children return Success***, then return ***Success***
- If Stop and Return Running
  - the next execution will start from the ***running action***



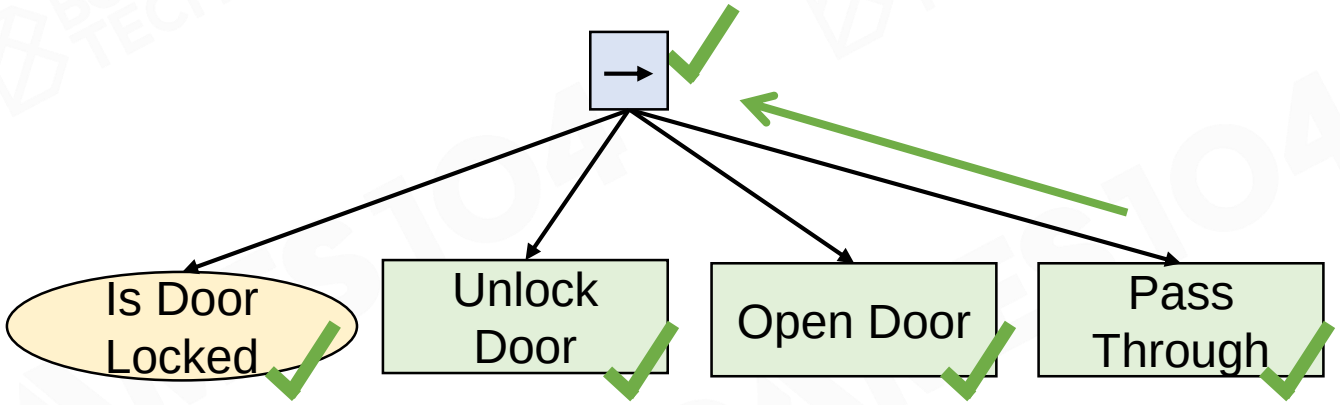
Algorithm 1 Sequence Node

```
1: for all child in children do
2:     childStatus ⇐ tick(child)
3:     if childStatus is Running then
4:         status ⇐ Running, terminate
5:     else if childStatus is Failure then
6:         status ⇐ Failure, terminate
7:     end if
8: end for
9: status ⇐ Success
```

Control Node – Sequence (2/2)

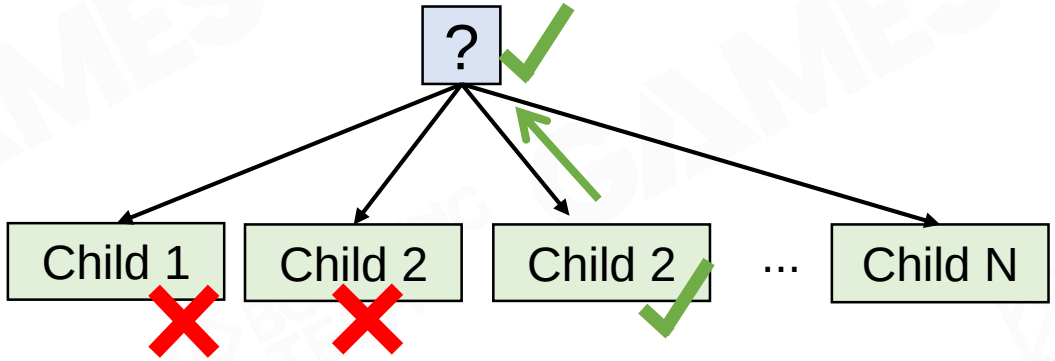
Sequence

- Allows designers to make a “**plan**”



Control Node – Selector (1/2)

- Order
  - Execute children from left to right
- Stop Condition and Return Value
  - until one child returns ***Success*** or ***Running***, then return value accordingly
  - or ***all children return Failure***, then return ***Failure***
- If Stop and Return Running
  - the next execution will start from the ***running action***



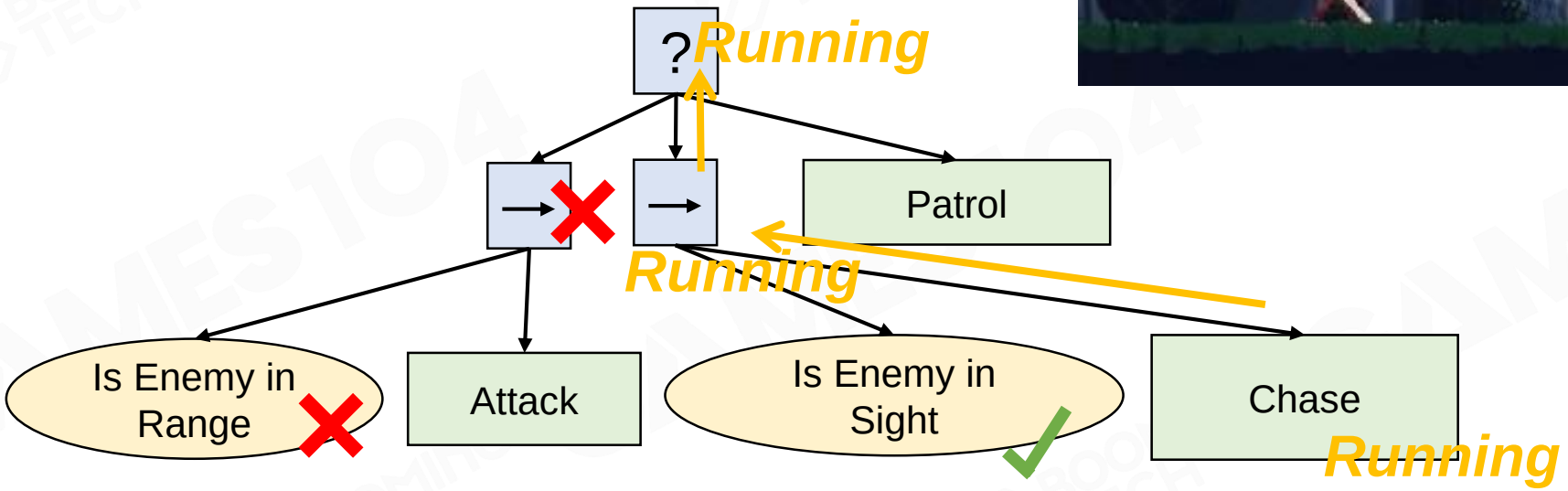
Algorithm 2 Selector Node

```
1: for all child in children do
2:     childStatus ⇐ tick(child)
3:     if childStatus is Running then
4:         status ⇐ Running, terminate
5:     else if childStatus is Success then
6:         status ⇐ Success, terminate
7:     end if
8: end for
9: status ⇐ Failure
```

Control Node – Selector (2/2)

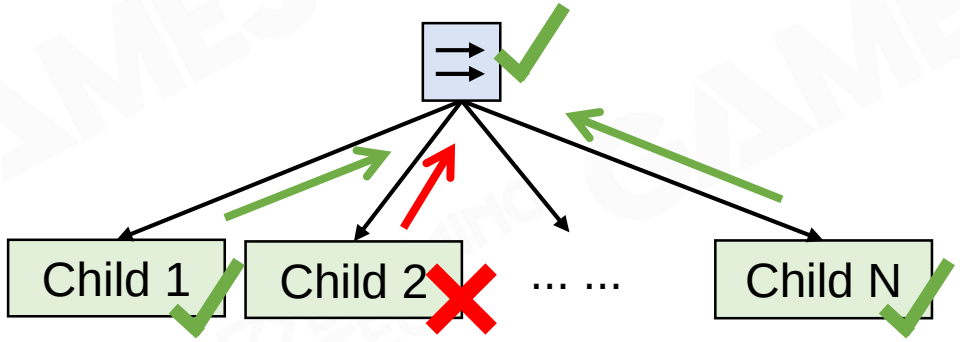
Selector

- Could **select** one action to do **response** to different environment
- Could do the right thing according to **priority**



Control Node – Parallel (1/2)

- Order
  - Logically execute all children simultaneously
- Stop Condition and Return Value
  - Return ***Success*** when at least **M** child nodes (between 1 and N) have succeeded
  - Return ***Failure*** when at least **N - M + 1** child nodes (between 1 and N) have failed
  - Otherwise return ***Running***
- If Stop and Return Running
  - the next execution will start from the ***running actions***



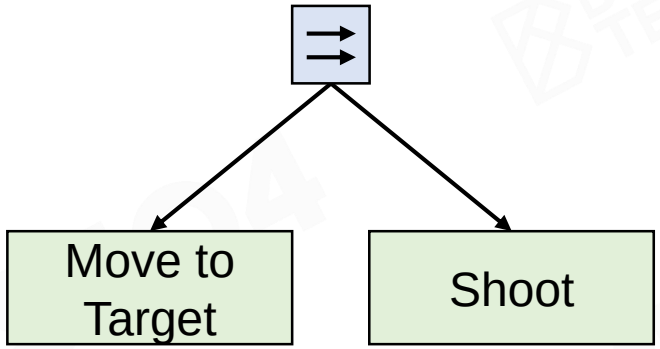
Algorithm 3 Parallel Node

```
1: for all child in children do
2:     childStatus ⇐ tick(child)
3: end for
4: if number of Success childStatus ≥ M then
5:     status ⇐ Success, terminate
6: else if number of Failure childStatus > N − M then
7:     status ⇐ Failure, terminate
8: end if
9: status ⇐ Running
```

Control Node – Parallel (2/2)

Parallel

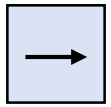
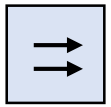
- Could do **multiple** things **"at the same time"**



Behavior Tree

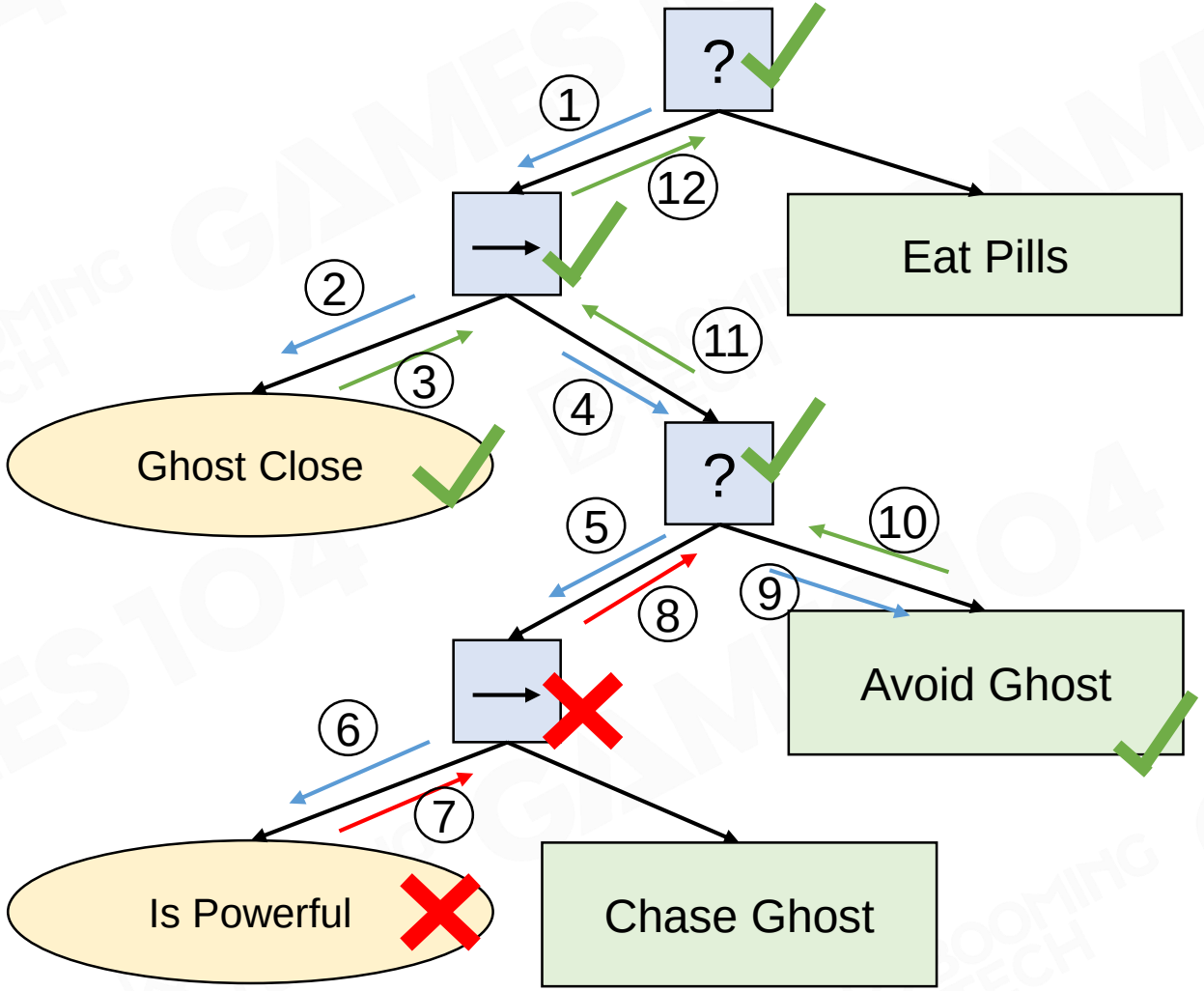
Execution nodes
- Action
- Condition

Control flow nodes
- Sequence
- Selector
- Parallel

Node Type	Symbol	Succeeds	Fails	Running
Sequence		If all children succeed	If one child fails	If one child returns Running
Selector		If one child succeeds	If all children fail	If one child returns Running
Parallel		If ≥ *M* children succeed	If > *N* - *M* children fail	else
Condition		Upon completion	If impossible to complete	During completion
Action		If true	If false	Never

Tick a Behavior Tree

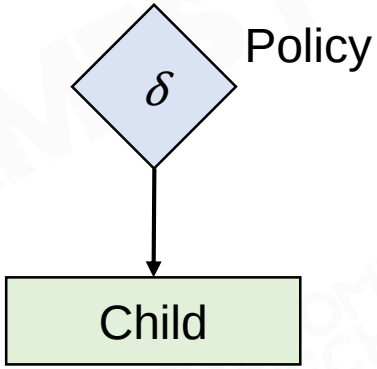
- The tick of BT is like thinking
- Every tick start from root node
- Go through different nodes from up to down, left to right
- Each node must return failure, success or running



Behavior Tree – Decorator (1/2)

Decorator

- A special kind of control node with a single child node
- Usually some behavior pattern which is commonly used



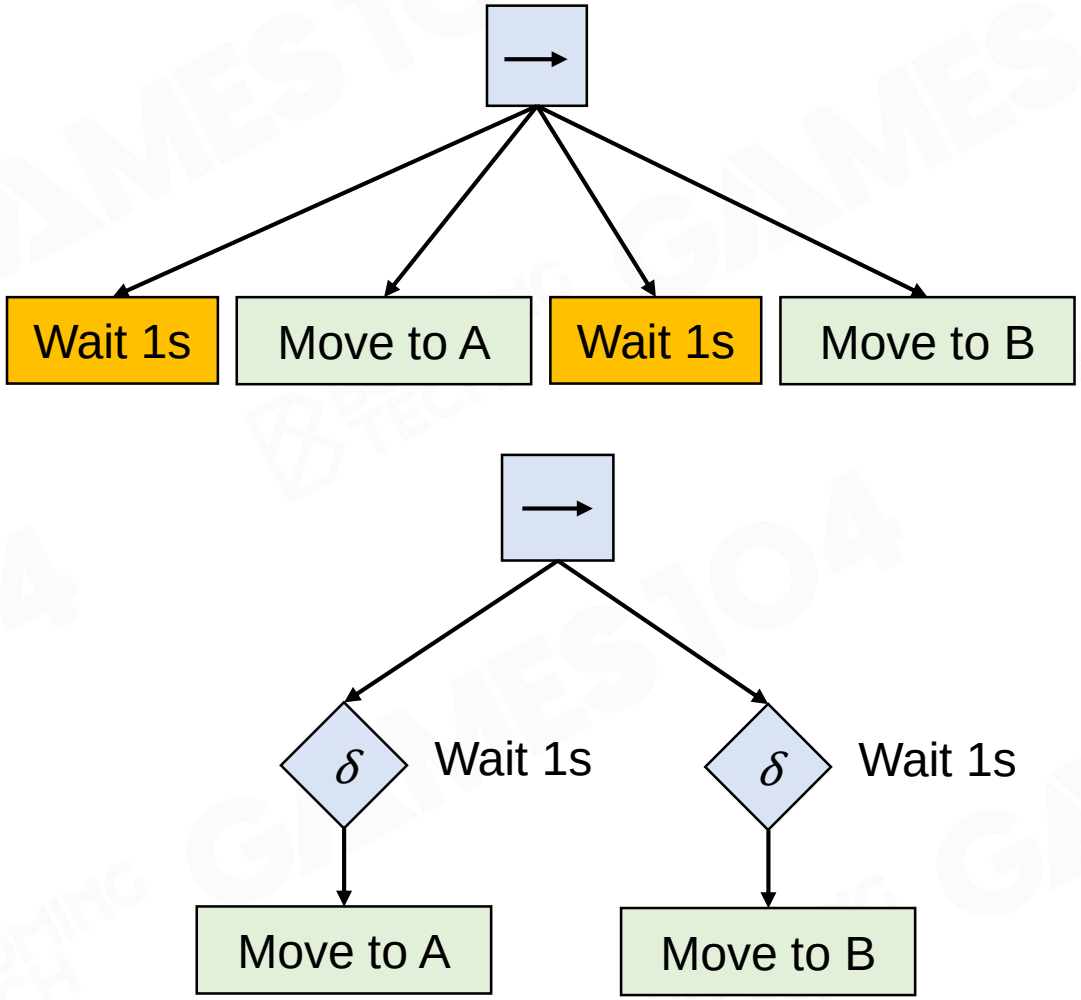
- For example, some common policies:
  - Loop execution
  - Execute once
  - Timer
  - Time Limiter
  - Value Modifier
  - Etc.

Examples of Decorator Nodes

Behavior Tree – Decorator (2/2)

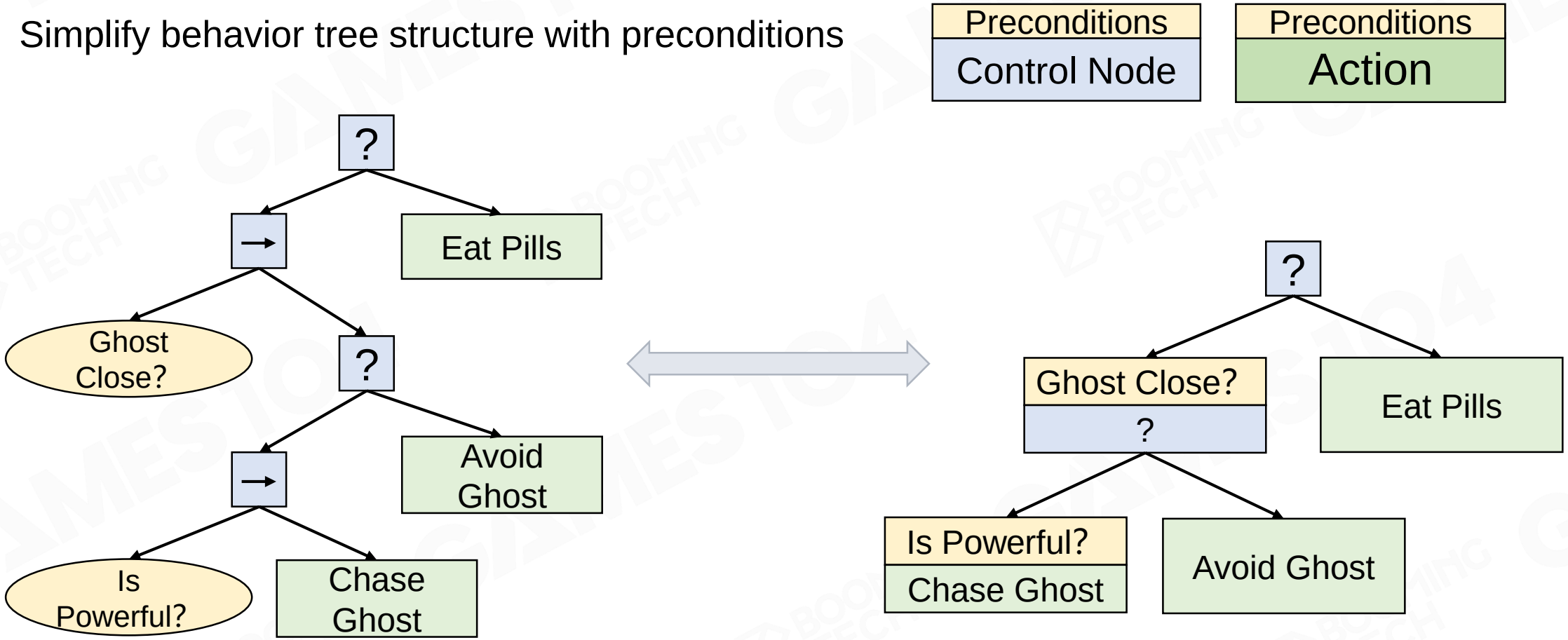
Decorator

- Example: Use timer to implement "patrol"



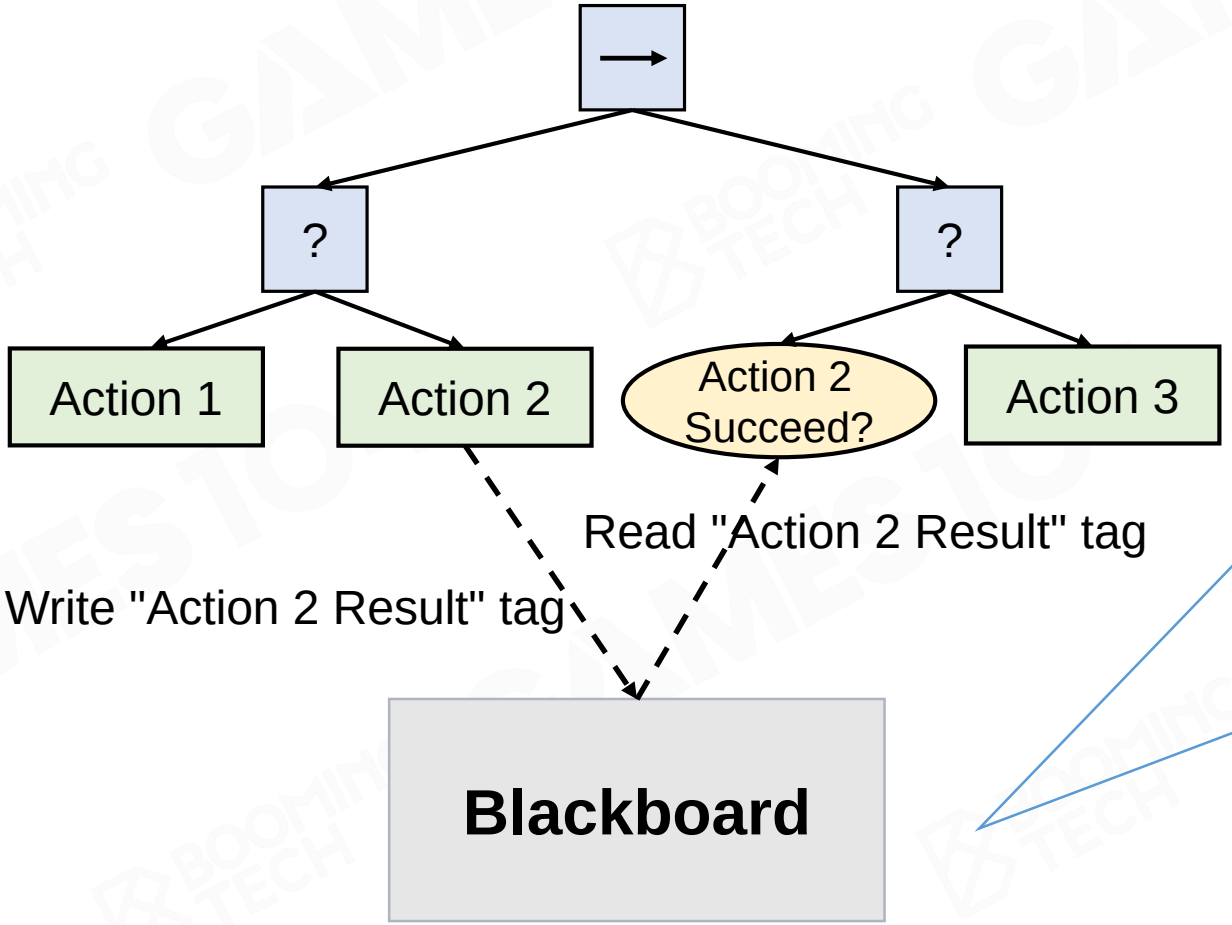
Behavior Tree – Precondition

Simplify behavior tree structure with preconditions



Behavior Tree – Blackboard

Blackboard : the memory of behavior tree

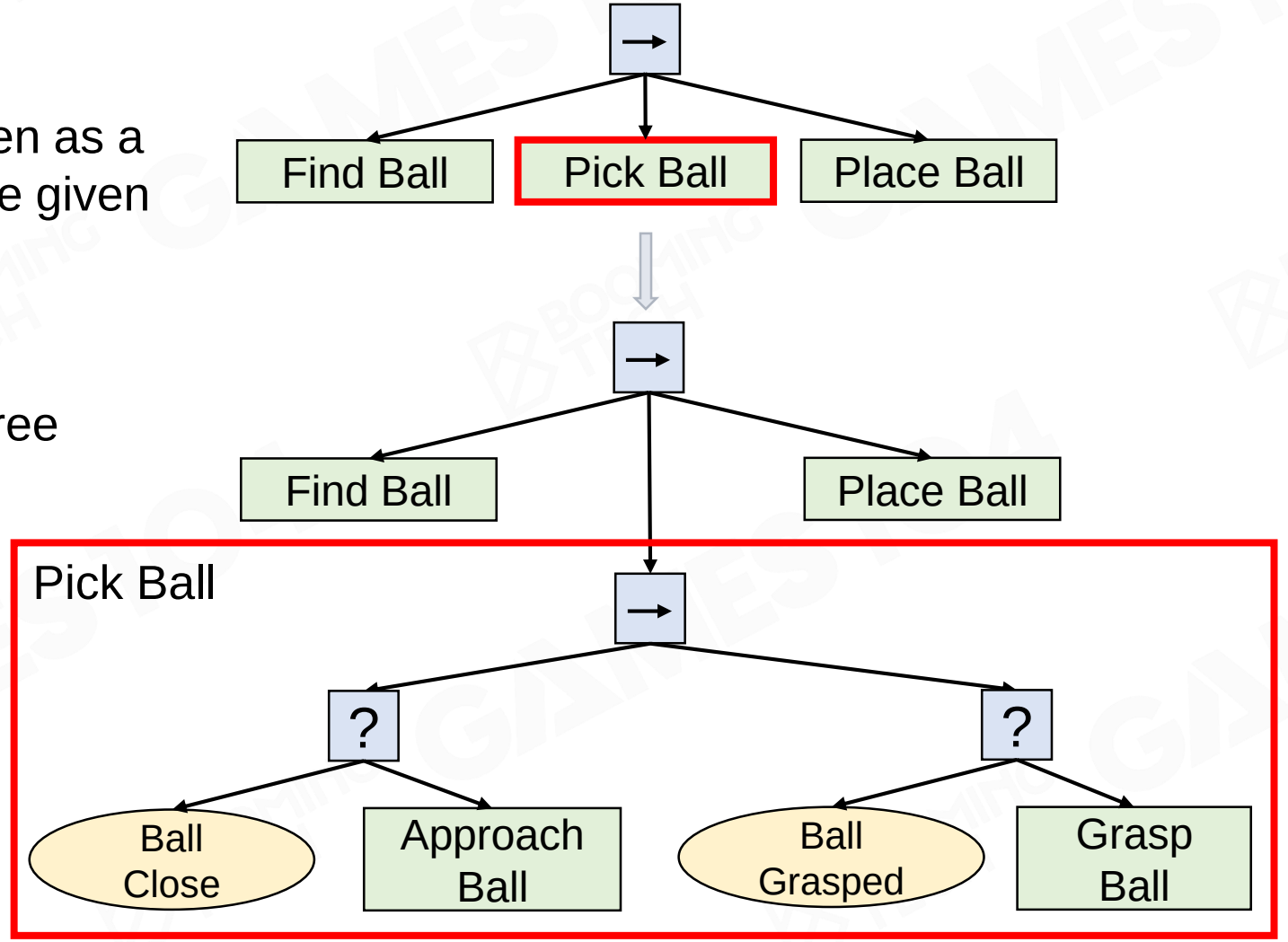


Blackboard

Key	Value
“Action 1 Result”	False
“Action 2 Result”	True
“Action 3 Result”	False
... ...	

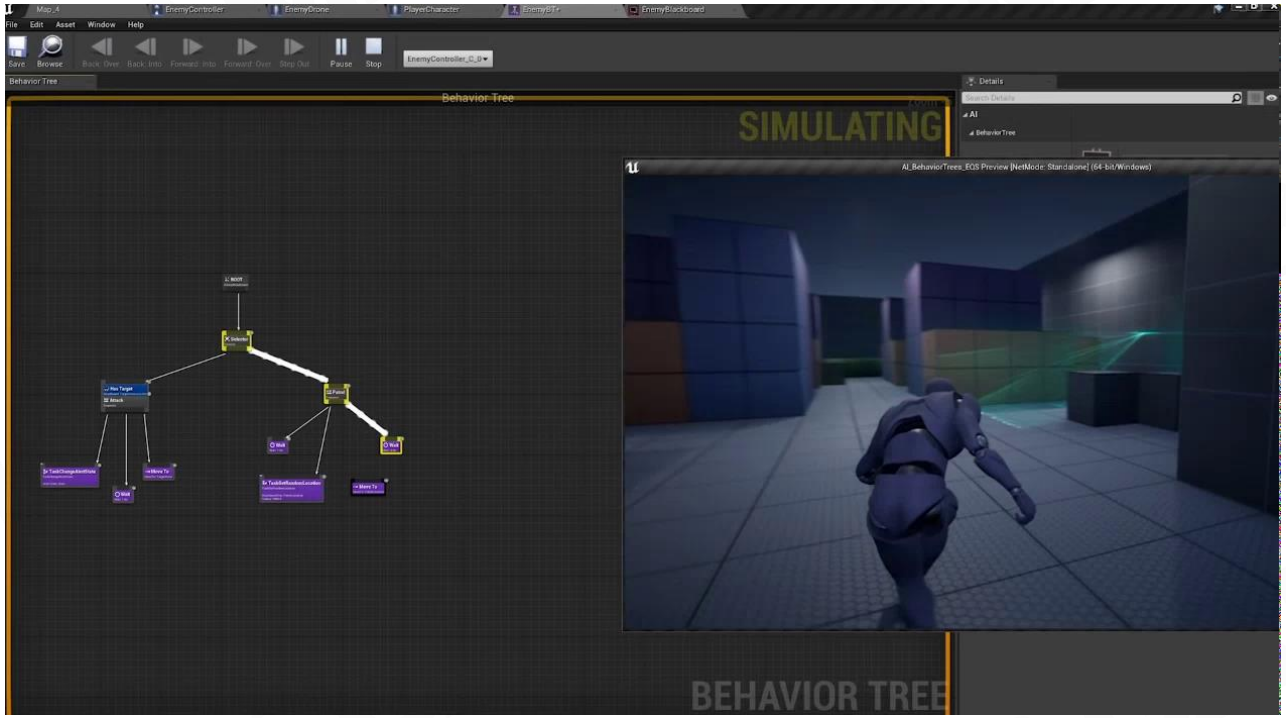
Behavior Tree – Pros (1/2)

- Modular, Hierarchical organization
  - **each subtree** of a BT can be seen as a **module**, with a standard interface given by the **return statuses**
- Human readable
- Easy to maintain
  - Modification only affect parts of tree



Behavior Tree – Pros (2/2)

- Reactivity
  - Think every tick to quickly change behavior according to environment
- Easy to Debug
  - Every tick is a whole decision making process, so taht it is easy to debug



Behavior Tree – Cons

Cons

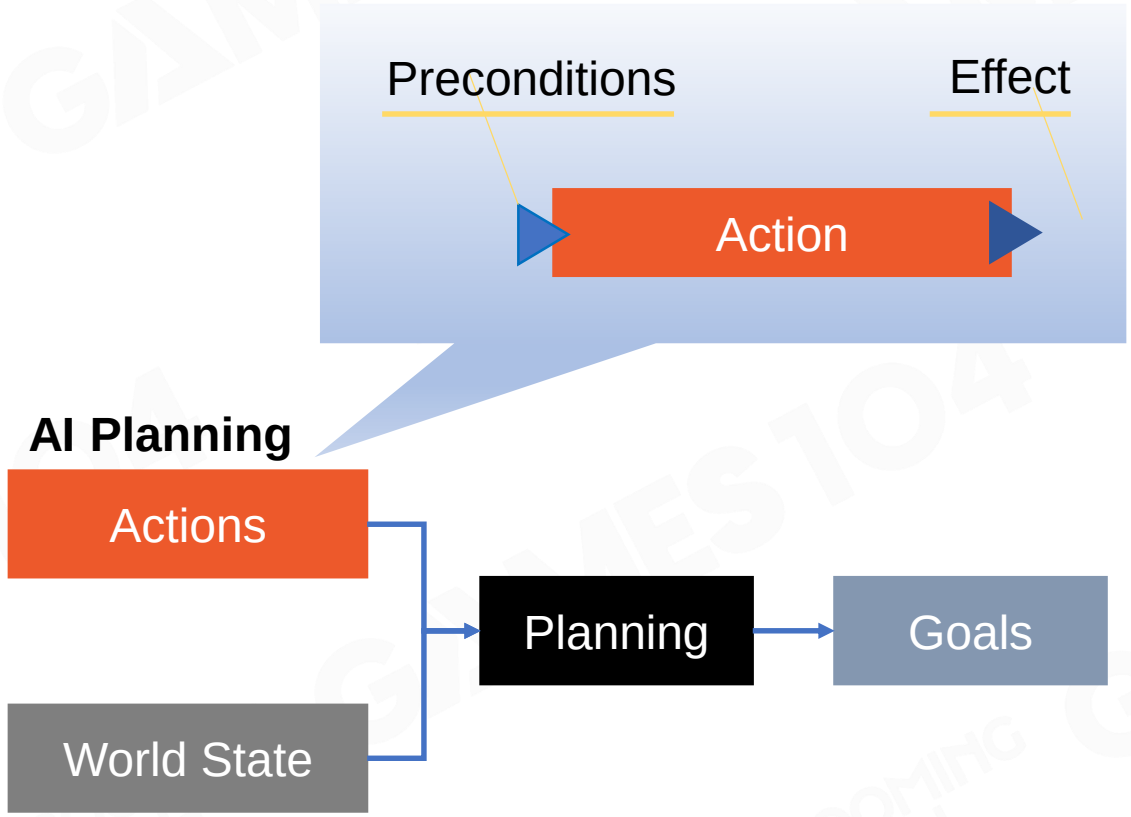
- Each tick starts from root node which costs much more
- The more reactive, the more condition to be checked and the more costs per tick

Upcoming: AI Planning and Goals

To make the AI more deliberative, game designers introduced the **AI Planning technique** to improve the planning ability of AI

AI Planning:

- Manage a set of actions
- A planner make a plan according to the initial world state



References

Navigation

- Amit Patel's fabulous website with animated demonstration of pathfinding algorithms: http://theory.stanford.edu/~amitp/GameProgramming/
- In-depth explanation of Detour Recast Navmesh generation algorithm: http://critterai.org/projects/nmgen_study/
- AI Summit: 'Death Stranding': An AI Postmortem, Eric Johnson, Kojima Productions, GDC 2021: https://gdcvault.com/play/1027144/AI-Summit-Death-Stranding-An
- Getting off the NavMesh: Navigating in Fully 3D Environments, Dan Brewer, Digital Extremes, GDC 2015: https://gdcvault.com/play/1022016/Getting-off-the-NavMesh-Navigating
- Mikko Mononen, Author of Recast Detour, blog on Funnel Algorithm : http://digestingduck.blogspot.com/2010/03/simple-stupid-funnel-algorithm.html?m=1

Steering & Sensing

- Steering Behaviors For Autonomous Characters, Craig W. Reynolds, Sony Computer Entertainment America: https://www.researchgate.net/publication/2495826_Steering_Behaviors_For_Autonomous_Characters
- AI For Games, Ian Millington, 3rd Edition, CRC Press, 2019
- Knowledge is Power, an Overview of AI Knowledge Representation in Games, Daniel Brewer: http://www.gameaipro.com/GameAIProOnlineEdition2021/GameAIProOnlineEdition2021_Chapter04_Knowledge_is_Power_an_Overview_of_AI_Knowledge_Representation_in_Games.pdf

Crowd Simulation

- A review on crowd simulation and modeling, Shanwen Yang, Tianrui Li, Xun Gong, Bo Peng, Jie Hu: https://www.sciencedirect.com/science/article/abs/pii/S152407032030024 2
- Forced-Based Anticipatory Collision Avoidance in Crowd Simulations, Stephen Guy, Ioannis Karamouzas, University of Minnesota, GDC 2015: https://www.gdcvault.com/play/1022465/Forced-Based-Anticipatory-Collision-Avoidance
- RVO and ORCA How They Really Work, Ben Sunshine-Hill: https://www.taylorfrancis.com/chapters/edit/10.1201/9780429055096-22/rvo-orca-ben-sunshine-hill
- RVO2 Library: Reciprocal Collision Avoidance for Real-Time Multi-Agent Simulation, Jur van den Berg, et al.: https://gamma.cs.unc.edu/RVO2/
- Documentation of AI systems in Unreal Engine 5: https://docs.unrealengine.com/5.0/en-US/artificial-intelligence-in-unreal-engine/

Classical Decision Making Algorithms

- Behavior Trees in Robotics and AI: an Introduction, Michele Colledanchise and Petter Ögre, 1st Edition, CRC Press, 2018, https://www.researchgate.net/publication/319463746_Behavior_Trees_in_Robotics_and_AI_An_Introduction
- The Behavior Tree Starter Kit, Alex Champandard, Philip Dunstan, Game AI Pro, 2013: http://www.gameaipro.com/GameAIPro/GameAIPro_Chapter06_The_Behavior_Tree_Starter_Kit.pdf
- FSM: Finite-State Machines: Theory and Implementation, https://gamedevelopment.tutsplus.com/tutorials/finite-state-machines-theory-and-implementation--gamedev-11867
- Managing Complexity in the Halo 2 AI System, Damian Isla, Moonshot Games, GDC 2005: https://www.taylorfrancis.com/chapters/edit/10.1201/9780429055096-22/rvo-orca-ben-sunshine-hill

Lecture 16 Contributor

- Olorin
- 新之助
- 大喷
- 普普
- 喵小君
- 一将
- Hoya

Q&A

Enjoy ;)
Coding

Course Wechat

Follow us for
further information