Path Finding

Content

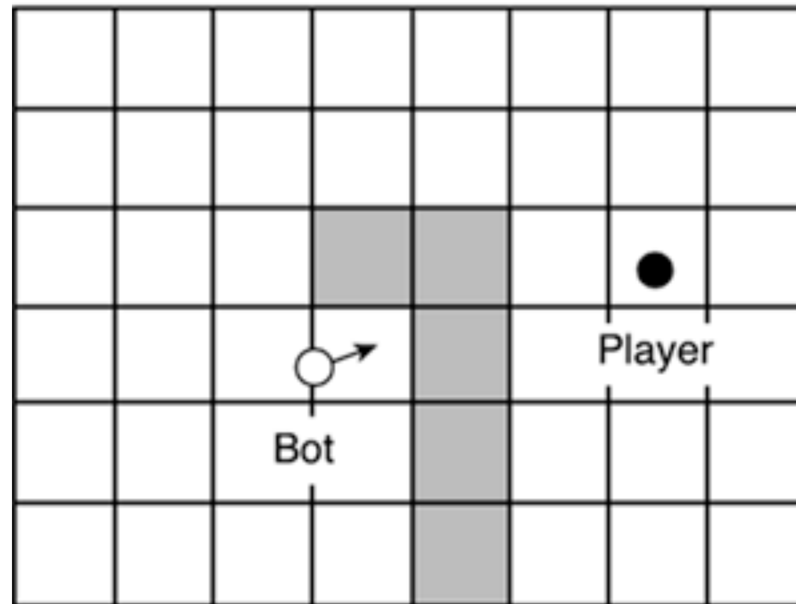
1. Path-Finding Basics
2. Some Initial Path-Finding Attempts
3. Basics of the A* Algorithm
4. Applying the A* Algorithm
5. Using the A* Algorithm with a BSP Tree
6. Generic Path Finding
7. Making a PathBot
8. Enhancing the A* Search
9. Summary

1. Path-Finding Basics

- Path finding can be reduced to answering the question, "How do I get from point A to point B?" Generally, the path from a ninja (or, in this case, a bot) to another location could potentially have several different solutions, but ideally, you want a solution that solves these goals:
  - How to get from A to B
  - How to get around obstacles in the way
  - How to find the shortest possible path
  - How to find the path quickly
- Some path-finding algorithms solve none of these problems, and some solve all of them. And in some cases, no algorithm could solve any of them—for example, point A could be on an island completely isolated from point B, so no solution would exist.

2. Some Initial Path-Finding Attempts

- If the environment is flat with no obstacles, path finding is no problem: The bot can just move in a straight line to its destination. But when obstacles occur between point A and point B, things get a little hairy. For example, in the figure, if the bot just moves straight toward the player, it will end up just hitting a wall and getting stuck there.



Not the best solution: The bot (white circle) just runs straight toward the player (black circle), even if there are obstacles in the way.

2. Some Initial Path-Finding Attempts (2)

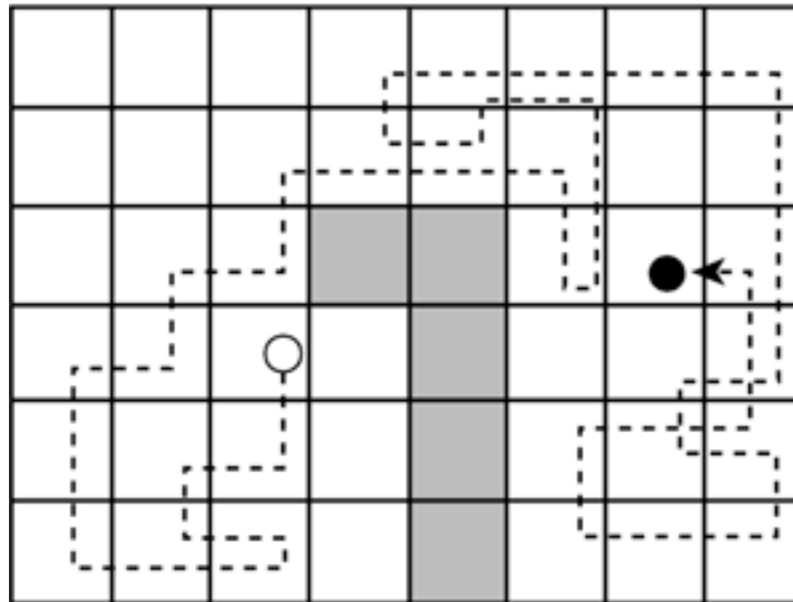
- This behavior might be fine for some games because it doesn't really matter what the bots do; it matters only what they appear to do. In a 3D world from the player's perspective, it might look like the bot is just waiting behind the wall for the player instead of being stuck.

- A bird's-eye perspective is a different story. Path finding is virtually required for games with bird's-eye perspectives, such as real-time strategy games or for games like The Sims. You wouldn't want to tell a team of fighters to attack something but have the team not be able to do it because the destination was on the other side of the wall! Come on now, use the door.

- Also, making a bot smart enough to find a path around an obstacle makes it more interesting for the player and harder to "trick" the bot into doing something or moving a certain way.

2. Some Initial Path-Finding Attempts (3)

- Several techniques can be used to get around obstacles. A randomized algorithm finds a solution eventually, if there is one. This technique makes the bot walk around randomly, like in this figure, until it finds what it is looking for.



The bot (white circle) moves around randomly until the player (black circle) is found.

2. Some Initial Path-Finding Attempts (4)

- Of course, the randomized algorithm doesn't give the best-looking results or the shortest path. Who wants to watch a bot move around randomly like it's missing a circuit or two?

- This random algorithm could be modified to move randomly only occasionally and at other times move directly toward the player, which would make the bot's movement appear slightly more intelligent.

- Another solution can be found when the environment is a simple maze with no loops, like in the next figure.

2. Some Initial Path-Finding Attempts (5)

- Here, the path-finding algorithm is the "right hand on the wall" algorithm (or the "left hand on the wall" algorithm, if you're into that sort of thing). Just keep your right hand on the wall as you move through the maze, and eventually you'll find the destination.

The bot (white circle) moves in a maze, keeping its right hand on the wall until it finds the player (black circle).

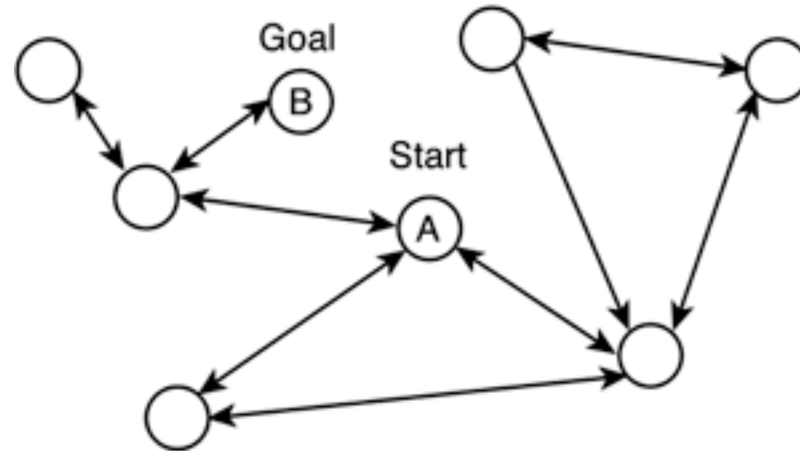
2. Some Initial Path-Finding Attempts (6)

- If a bot follows the "right hand" algorithm literally, it will rarely find the shortest path from A to B. Alternatively, the path could be calculated ahead of time, before the bot makes any moves. Then any backtracking (such as when venturing into the lower-right hall in the figure) could be removed from the path, leaving a shortest-path solution.

- However, this algorithm works only for the simple case of a maze with no loops. If a loop was in the path, the bot would veer to the right forever. Also, if rooms had wide spaces, the bot might never find its destination in the middle of a room because the bot is always next to a wall. Ideally, you'll probably want to create more intricate environments than this.

2. Some Initial Path-Finding Attempts (7)

- Instead of looking at the environment as a grid, we'll look at it as a graph, like in this figure. A graph is simply a bunch of nodes grouped by various edges.

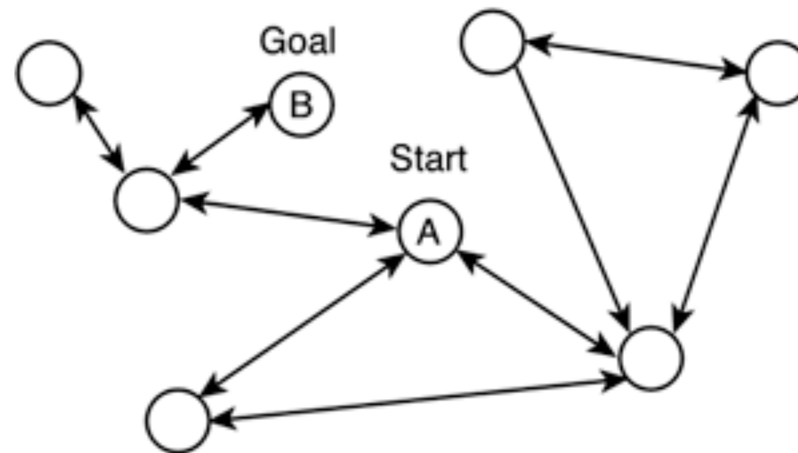


A simple graph.

- Each node of the graph could be anything. For example, a node could be a cell in a 2D grid. Or, a node could be a "city," with the edges of the graph representing highways. And remember, when finding the path from A to B, any node can be the start or goal node.

2. Some Initial Path-Finding Attempts (8)

- A graph is similar to a tree, as discussed in Chapter 10, "Scene Management Using BSP Trees," except that, instead of each node having up to two children, each node can have an indefinite number of children, or neighbors. Some graphs are directed, meaning that an edge between two nodes can be traversed in only one direction. Looking at the example, all the edges are bidirectional except for one. A unidirectional edge could be useful in such situations as when traveling involves jumping down a cliff that is too high to jump back up.



- In this example, you want to find the shortest path from node A to node B, or the fewest number of traversed edges to get to the goal. Looking at the figure, the solution is easy to determine, but how would you find the solution in a computer program? An easy solution is to use a breadth-first search.

Breadth-First Search

- Like traversing BSP trees, a breadth-first search involves visiting nodes one at a time. A breadth-first search visits nodes in the order of their distance from the start node, where distance is measured as the number of traversed edges.

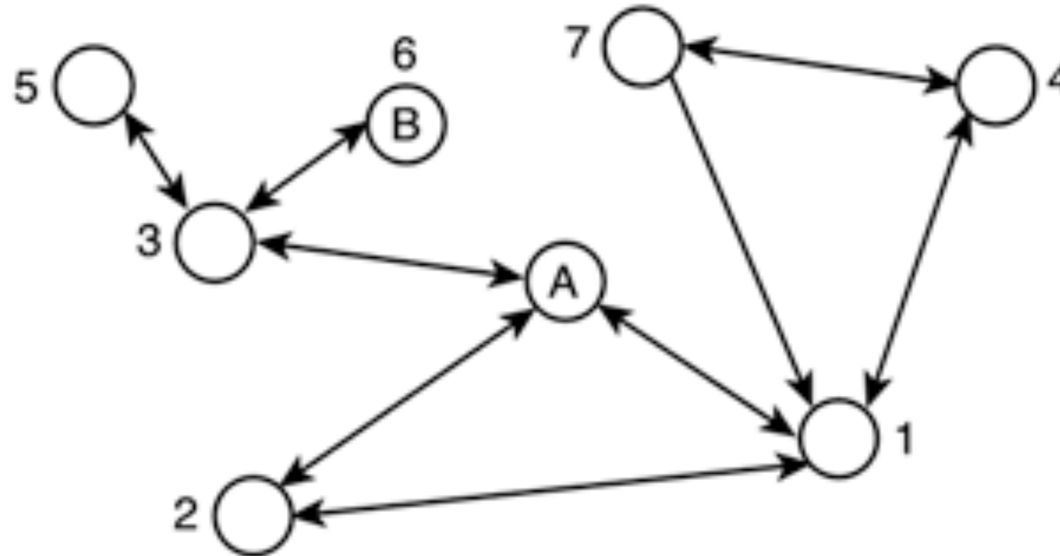
- So, with a breadth-first search, first all nodes one edge away from the goal are visited, then those two edges away are visited, and so on until all nodes are visited.

- This way, you find the path from the start to the goal with the minimum number of traversed edges.

- Another way to word it is like this: Visit the neighbor nodes, then the neighbor's neighbor nodes, and so on until the goal node is found.

Breadth-First Search (2)

- An example of a breadth-first search is in this figure, in which the nodes are numbered in the order they are visited.



- Later in this chapter, you'll use the **A*** search algorithm, which has several similarities to a breadth-first search. But first, to better understand A*, you'll implement the easy-to-understand breadth-first search algorithm.

Breadth-First Search (3)

- First, start with a basic node, which has references to all its neighbors:

```
public class Node {
    List neighbors;
    Node pathParent;
}
```

- The neighbors list is simply a list of all the nodes' neighbors.

- The **pathParent** node is used for searching only. Think of the path from the start node to the goal node as a tree, with each node having only one child. The pathParent node is the node's parent in the path tree. When the goal is found, the path can be found by traversing up the path tree from the goal node to the start node, like this:

```
private List constructPath(Node node) {
    LinkedList path = new LinkedList();
    while (node.pathParent != null) {
        path.addFirst(node);
        node = node.pathParent;
    }
    return path;
}
```

Breadth-First Search (4)

- Now you're ready to implement the breadth-first search algorithm. One thing to keep in mind is that you want to be sure to only visit each node once. For example, if A has neighbors B and C, and B has neighbor A, you don't want to visit A again or you'll end up in an infinite loop.

- So, you'll keep track of all nodes that have been visited by putting them in a "closed" list. If a node shows up in the search that's already in the closed list, you'll ignore it. Likewise, you'll keep track of all the nodes you want to visit in an "open" list. The open list is a first in, first out list, effectively sorting the list from smallest number of edges from the start goal to the largest.

- Here's the source code:

Breadth-First Search (5)

```
public List search(Node startNode, Node goalNode) {
    // list of visited nodes
    LinkedList closedList = new LinkedList();

    // list of nodes to visit (sorted)
    LinkedList openList = new LinkedList();
    openList.add(startNode);
    startNode.pathParent = null;

    while (!openList.isEmpty()) {
        Node node = (Node)openList.removeFirst();
        if (node == goalNode) {
            // path found!
            return constructPath(goalNode);
        }
        else {
            closedList.add(node);

            // add neighbors to the open list
            Iterator i = node.neighbors.iterator();
            while (i.hasNext()) {
                Node neighborNode = (Node)i.next();
                if (!closedList.contains(neighborNode) &&
                    !openList.contains(neighborNode))
                {
                    neighborNode.pathParent = node;
                    openList.add(neighborNode);
                }
            }
        }
    }

    // no path found
    return null;
}
```

3. Basics of the A* Algorithm

- An A*—pronounced "A-star"—search works like a breadth-first search, except with two extra factors:
  - The edges have different "costs," which is how much it costs to travel from one node to another.
  - The cost from any node to the goal node can be estimated. This helps refine the search so that you're less likely to go off searching in the wrong direction.
- The cost between nodes doesn't have to be distance. The cost could be time, if you wanted to find the path that takes the shortest amount of time to traverse. For example, when you are driving, taking the back roads might be a shorter distance, but taking the freeway usually takes less time (freeways in Los Angeles excluded). Another example is terrain: Traveling through overgrown forests could take longer than traveling through a grassy area.

Basics of the A* Algorithm (2)

- Or, you could get more creative with the cost. For instance, if you want a bot to sneak up on the player, having the bot appear in front of the player could have a high cost, but appearing from behind could have little or no cost. You could take this idea further and assign a special tactical advantage to certain nodes—such as getting behind a crate—which would have a smaller cost than just appearing in front of the player.

- The A* algorithm works the same as breadth-first search, except for a couple of differences:

  - The nodes in the open list are sorted by the total cost from the start to the goal node. In other words, it's a priority queue. The total cost is the sum of the cost from the start node and the estimated cost to the goal node.

  - A node in the closed list can be moved back to the open list if a shorter path (less cost) to that node is found.

Basics of the A* Algorithm (3)

- Because the open list is sorted by the estimated total cost, the algorithm checks nodes that have the smallest estimated cost first, so it searches the nodes that are more likely to be in the direction of the goal. Thus, the better the estimate is, the faster the search is.

- Of course, you need to define the cost and the estimated cost. If the cost is distance, this is easy: The cost between nodes is their distance, and the cost from one node to the goal is simply a calculation of the distance from that node to the goal.

- Note that this algorithm works only when the estimated cost is never more than the actual cost. If the estimated cost were more, the path found wouldn't necessarily be the shortest.

Basics of the A* Algorithm (4)

```
/**
    The AStarNode class, along with the AStarSearch class,
    implements a generic A* search algorithm. The AStarNode
    class should be subclassed to provide searching capability.
*/
public abstract class AStarNode implements Comparable {

    AStarNode pathParent;
    float costFromStart;
    float estimatedCostToGoal;


    public float getCost() {
        return costFromStart + estimatedCostToGoal;
    }


    public int compareTo(Object other) {
        float otherValue = ((AStarNode)other).getCost();
        float thisValue = this.getCost();

        return MoreMath.sign(thisValue - otherValue);
    }


    /**
        Gets the cost between this node and the specified
        adjacent (aka "neighbor" or "child") node.
    */
    public abstract float getCost(AStarNode node);


    /**
        Gets the estimated cost between this node and the
        specified node. The estimated cost should never exceed
        the true cost. The better the estimate, the more
        efficient the search.
    */
    public abstract float getEstimatedCost(AStarNode node);


    /**
        Gets the children (aka "neighbors" or "adjacent nodes")
        of this node.
    */
    public abstract List getNeighbors();
}
```

- Source code. First, you'll create a generic, abstract A* search algorithm that can be used for any type of A* search.
- The idea is that you'll be able to use this generic abstract class for lots of different situations, and because it's generic, the code will be easier to read. You'll start with an A* node.

Basics of the A* Algorithm (5)

- The **AStarNode** class is an abstract class that needs to be subclassed to provide any search functionality. Like the node used for the breadth-first search, it contains the pathParent node used during the search process only. **costFromStart** and **estimatedCostToGoal** are also filled in during the search because these vary depending on where the start and goal nodes are.

- The **getCost(node)** abstract method returns the cost between the node and a neighbor node. The **getEstimatedCost(node)** abstract method returns the estimated cost between the node and the specified goal node. Remember, you have to create these functions, depending on what you want the cost to be.

- Now we'll implement the A* search.

`Source: AStarSearch.java`

Basics of the A* Algorithm (6)

- The **PriorityList** inner class is a simple LinkedList that adds nodes using an insertion sort. In this case, only AStarNode instances are added, keeping the list sorted from lowest cost to highest. Nodes are removed only from the front of the list (lowest cost).

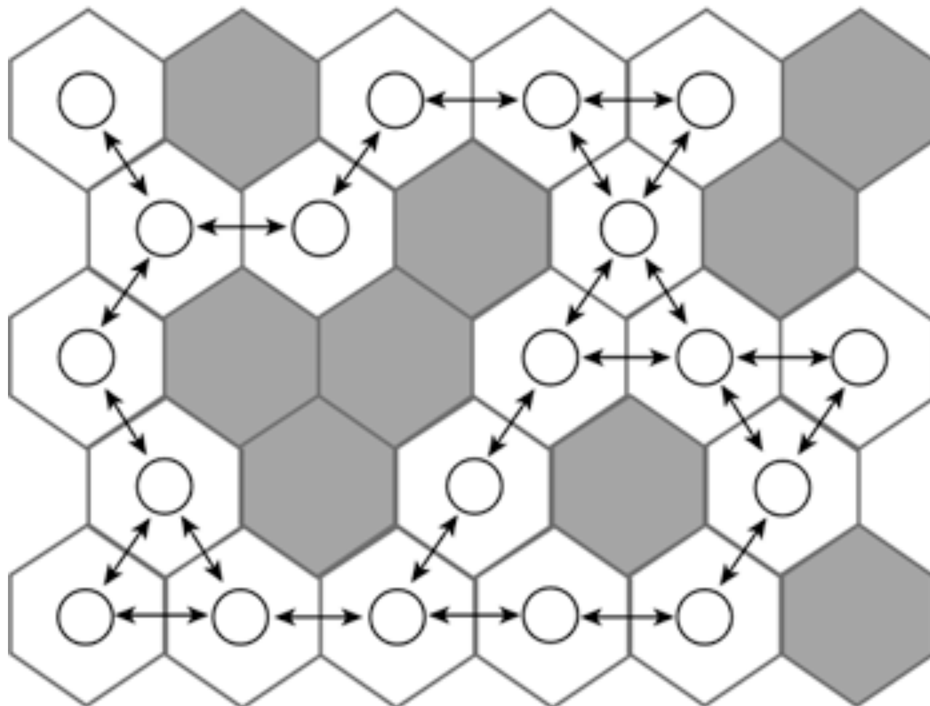
- The **findPath()** function is very similar to the breadth-first search implementation, except for a couple of changes. The costFromStart and estimatedCostToGoal fields are calculated as you go along. Also, a node is moved from the closed list to the open list if a shorter path to that node is found. Other than that, they're pretty much the same.

- That's it for the basics of the A* algorithm. The next step is to actually apply this algorithm in a game.

4. Applying the A* Algorithm

- To use the A* algorithm in a game, you'll have to interpret the game environment as a graph—or, in other words, decide what the nodes and edges represent.

- In a top-down, tiled-based world, this is easy: The nodes represent the tiles, as in the example. In this figure, the game environment is laid out on a hexagonal grid, and you can travel from one tile to any of its six neighboring tiles (unless one or more of the neighboring tiles is an obstacle).



A tiled-based game can easily be interpreted as a graph.

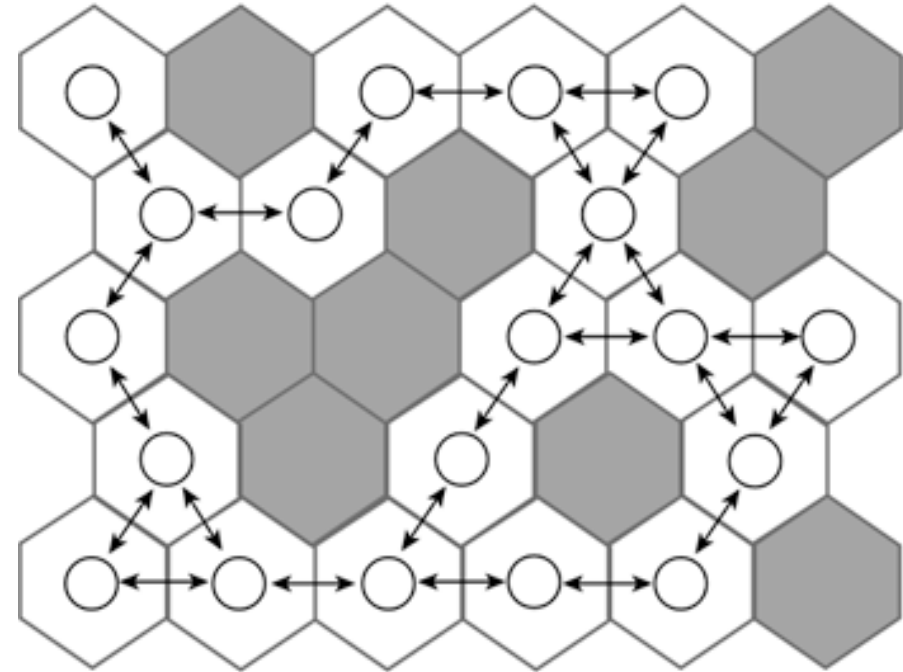
Applying the A* Algorithm (2)

- In this case, the distance between neighbor nodes is the same, but the cost between neighbor nodes doesn't have to be. As mentioned before, you could make traveling on some tiles, such as muddy sand, take longer than on other tiles, such as sidewalks.

- If the tiles were square, traveling diagonally would have a greater cost than traveling straight north, south, east, or west because the diagonal distance between two square tiles is greater.

- One benefit of a top-down, tile-based world is that it's easy to dynamically modify the graph. If the player blasts a tile, changing it from an obstacle to a grassy area, the graph can be easily updated.

- However, tiled-based worlds aren't the only situation in which you can use the A* algorithm.



5. Using the A* Algorithm with a BSP Tree

- Let's consider a typical floor plan for a 3D world like you used in previous chapters. In this world, what could your nodes be?

- First, the nodes could be placed by hand. All you would have to do is make sure the edges between neighbor nodes have a clear path (no obstacles in the way) and that there are enough nodes to cover the entire map.

- However, this can be a bit tedious. Who wants to sit around defining where path-finding nodes are on a map? So let's try to come up with an automatic solution for node placement.

Portals

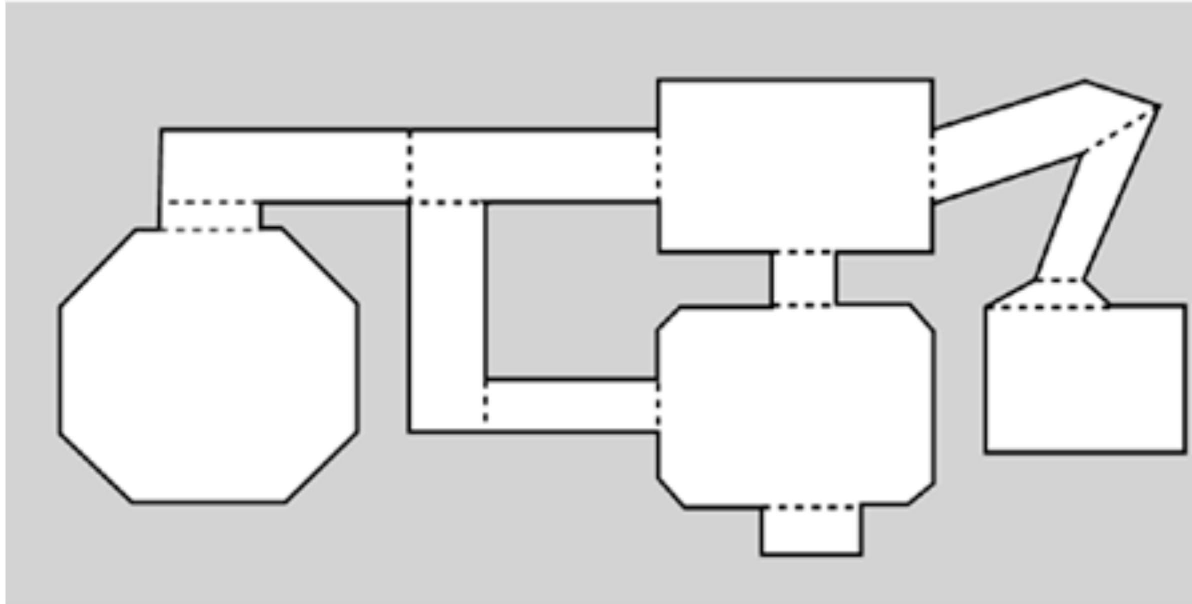
- “Rooms" in the floor plan are convex. This means a bot traveling from one entryway of the room to another will never hit any walls, as long as the two entryways aren't collinear.

- If two entryways are collinear (they exist on the same line in space) and a wall was between them (think doorways in a dorm hall), then a bot could collide with the wall between them. So, extra care may be needed to define room boundaries so that collinear entryways in the same room don't occur.

- Entryways between convex rooms are often called portals. Portals are generally an invisible polygon (or, in a 2D world, an invisible line) that separates two convex areas.

Portals (2)

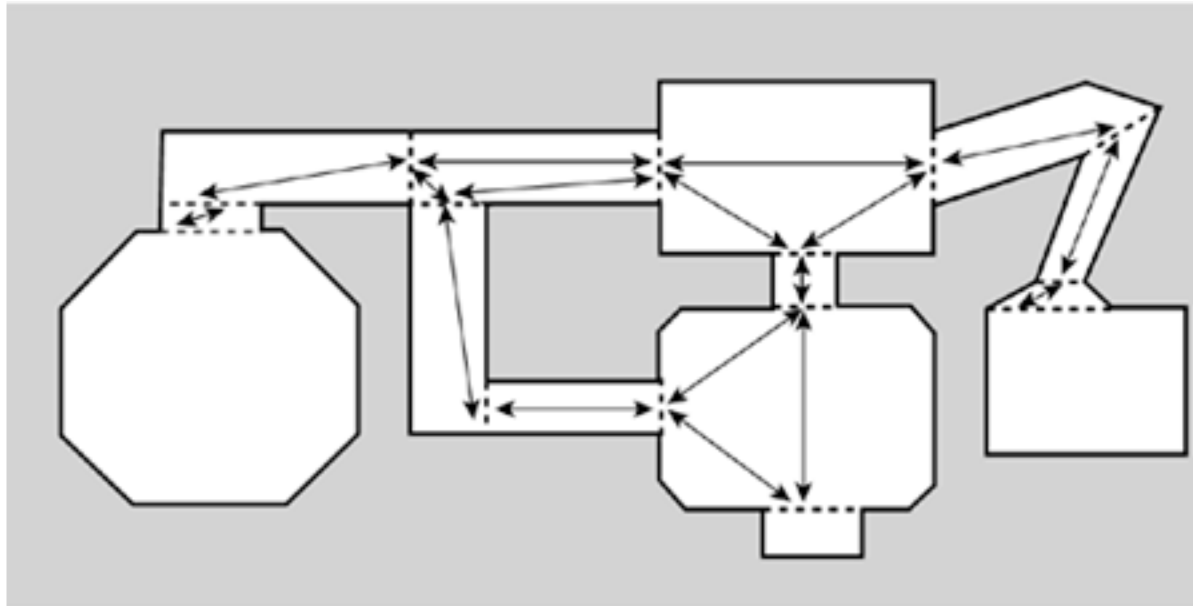
- See example: The dotted lines represent the portals.



- Portals can also be used as a rendering technique. With portal rendering, you start by drawing all the polygons in the room that the camera is in. If any portals are in the view, the room on the other side of the portal is drawn, and then so on until the view is filled.

Portals (3)

- Using portals as nodes means you can easily create a graph for traveling room to room.

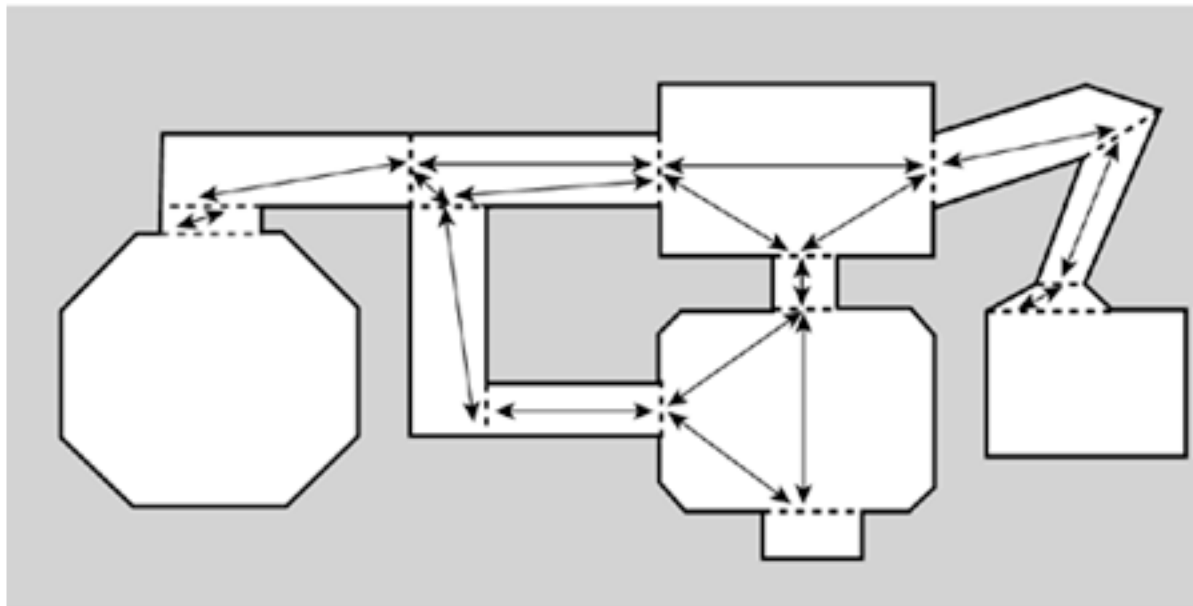


- The only issue is that the start and goal locations are within a room, not on a portal. However, because rooms are convex, a bot can move in a straight line from any location in a room to one of the room's portals without running into any walls.

Portals (4)

- So, when you are doing an A* search, you can make temporary nodes that represent the start and the goal. These node's neighbors are all the portals of the room they are in.

- Note that using portals as nodes takes only the world into account—obstacles based on game objects aren't considered. So, if you have a door or other game object in the way, you'll need some extra code to move around those objects or at least consider them in the search.



Implementing Portals

- In a BSP tree, portals can be stored in the leaves of the tree because the leaves represent convex areas. This way, you can easily get all the portals for a room. Here you just add a list of portals to the BSP tree leaf:

```
public static class Leaf extends Node {
    public float floorHeight;
    public float ceilHeight;
    public Rectangle bounds;
    public List portals;
}
```

- Finding the portals is easy. Because a leaf in a 2D tree represents an area, just check each edge of that area to see if there's a "solid wall" on it. If there is not, you have a portal. The same is true for a 3D BSP tree, except that every face of the leaf volume is checked.

Implementing Portals (2)

```
/**
    Gets the Leaf in front of the specified partition.
*/
public Leaf getFrontLeaf(BSPLine partition) {
    return getLeaf(root, partition, BSPLine.FRONT);
}


/**
    Gets the Leaf in back of the specified partition.
*/
public Leaf getBackLeaf(BSPLine partition) {
    return getLeaf(root, partition, BSPLine.BACK);
}


protected Leaf getLeaf(Node node, BSPLine partition, int side)
{
    if (node == null || node instanceof Leaf) {
        return (Leaf)node;
    }
    int segSide = node.partition.getSide(partition);
    if (segSide == BSPLine.COLLINEAR) {
        segSide = side;
    }
    if (segSide == BSPLine.FRONT) {
        return getLeaf(node.front, partition, side);
    }
    else if (segSide == BSPLine.BACK) {
        return getLeaf(node.back, partition, side);
    }
    else { // BSPLine.SPANNING
        // shouldn't happen
        return null;
    }
}
```

- When you have a portal, you need to know what leaf is on either side of it. You can find out by doing a simple tree traversal, shown in Listing 12.3 with the **getFrontLeaf()** and **getBackLeaf()** methods of BSPTree.

Methods of BSPTree.java

Implementing Portals (3)

- Finally, you'll make a Portal class, as shown in Listing 12.4. This class is a subclass of **AStarNode**.

```
Source: Portal.java
```

- The neighbors of a portal are the portals in the front and back leaves. The **getNeighbors()** method fills the neighbors in a list.

- The getEstimatedCost() estimates the cost between portals as the distance between the midpoints of each portal. Note that for two adjacent nodes, getEstimatedCost() is the same as the actual cost.

- Finally, you just need one more type of AStarNode that is a location anywhere within a convex area defined by the leaf. These nodes are generally just temporary because they are used only for the start and goal nodes of a specific path search.

Implementing Portals (4)

```
/**
    The LeafNode class is an AStarNode that represents a
    location in a leaf of a BSP tree. Used for the start
    and goal nodes of a search.
*/
public class LeafNode extends AStarNode {
    BSPTree.Leaf leaf;
    Vector3D location;

    public LeafNode(BSPTree.Leaf leaf, Vector3D location) {
        this.leaf = leaf;
        this.location = location;
    }

    public float getCost(AStarNode node) {
        return getEstimatedCost(node);
    }

    public float getEstimatedCost(AStarNode node) {
        float otherX;
        float otherZ;
        if (node instanceof Portal) {
            Portal other = (Portal)node;
            otherX = other.getMidPoint().x;
            otherZ = other.getMidPoint().z;
        }
        else {
            LeafNode other = (LeafNode)node;
            otherX = other.location.x;
            otherZ = other.location.z;
        }
        float dx = location.x - otherX;
        float dz = location.z - otherZ;
        return (float)Math.sqrt(dx * dx + dz * dz);
    }

    public List getNeighbors() {
        return leaf.portals;
    }
}
```

- The LeafNode class is fairly straightforward. The neighbors of a LeafNode are all the portals of the BSP tree leaf the node is in. The estimated cost is calculated as the distance between the node location and a portal's midpoint.
- Now you have all the little pieces you need to perform an A* search on a BSP tree.
- Next, you'll put all the little pieces together in a wrapper that implements a generic path-finding interface.

6. Generic Path Finding

- The A* search function you created returns a list of AStarNodes, which is great for generic implementation but isn't too friendly when you need something more concrete. Really, you want something like a list of locations, as in the PathFinder interface.

```
/**
    The PathFinder interface is a function that finds a path
    (represented by a List of Vector3Ds) from one location to
    another, or from one GameObject to another. Note that the
    find() method can ignore the requested goal, and instead
    give an arbitrary path, like patrolling in a set path or
    running away from the goal.
*/
public interface PathFinder {

    /**
        Finds a path from GameObject A to GameObject B. The path
        is an Iterator of Vector3Ds, not including the start
        location (GameObject A) but including the goal location
        (GameObject B). The Vector3D objects may be used in
        other objects and should not be modified.
        Returns null if no path found.
    */
    public Iterator find(GameObject a, GameObject b);


    /**
        Finds a path from the start location to the goal
        location. The path is an Iterator of Vector3Ds, not
        including the start location, but including the goal
        location. The Vector3D objects may be used in other
        objects and should not be modified. Returns null if no
        path found.
    */
    public Iterator find(Vector3D start, Vector3D goal);
}
```

Generic Path Finding (2)

- In this interface, two methods are provided: one to find a path between two points, and another to find a path between two objects.

- So, if you use PathFinder, it doesn't matter what the underlying algorithm is, whether it is A*, random, or some other idea. Also, you can use this interface for other types of paths, such as simple paths that follow the perimeter of a room or a flocking bird pattern as a background effect.

- Let's create an implementation of the PathFinder interface for finding the shortest path between two points in a BSP tree using the A* algorithm. Well, at least the **AStarSearchWithBSP** class in the next listing is straightforward.

`Source: AStarSearchWihBSP.java`

Generic Path Finding (3)

- The actual searching in the **find()** method is just like we said it would be: First, you create temporary nodes representing the start and goal nodes, add them to the graph, and then perform the search. After the search, you clean up by removing those temporary nodes from the graph. Also, if the start location and the goal location are in the same leaf of the BSP tree, no search needs to be performed, so the method just returns an Iterator over the goal location.

- The **convertPath()** method converts a list of AStarNodes to actual locations. Here, you use the midpoint of each portal as the location to use. Another idea instead of using the midpoint is to use a point within the portal that is closest to the location on either side of it, as long as you leave room for the size of the object. But midpoints work well, so you can stick with that simpler solution.

7. Making a PathBot

- Here's something you might have already thought about: Let's say you have a bot that finds a path to the player and starts to follow it, but by the time the bot reaches the goal location, the player is long gone. Kinda pointless, eh?

- You could recalculate the path every frame, but what a waste of resources! Performing an A* search makes quite a few temporary objects and takes up a chunk of processor power (including several Math.sqrt() calls), so you don't want a bot to recalculate the path every frame. Also, the player or any other goal location isn't likely to change much from frame to frame, so the path wouldn't need to change much, either.

- Instead, you can just keep a calculated path around for a while and recalculate it occasionally, every few seconds or so. This way, the path is fairly up-to-date but doesn't hog the processor or waste memory.

Making a PathBot (2)

- Another thought along those lines is, what if you have several bots calculating their path at the exact same time? You might see little drops in the frame rate of the game as this happens because several bots are hogging the processor at once. To alleviate this, you can just take the easy route and make sure each bot starts calculating a path after a random amount of time. In this case, you'll make sure bots don't calculate the path until 0 to 1000 milliseconds have passed.

- Another idea is to cache common A* searches from portal to portal (not including start and goal nodes). This way, two bots in the same room can share a path, and a path that's already been calculated wouldn't need to be calculated again.

- The PathBot follows the path given to it from a PathFinder. It has a few default attributes, such as its speed, how fast it can turn, and how high above the floor it "flies." The turn speed is purely cosmetic—the bot will turn to face the direction it's facing, but the turn speed doesn't affect how fast a bot travels along a path.

`Source: PathBot.java`

Making a PathBot (3)

- PathBot is fairly uncomplicated. It keeps track of how much time is left until the path is recalculated again, and if that amount of time has passed, the path is recalculated in the update() method. The rest of the update() method simply follows the path in this manner:
  - The bot's transform is set to move to the first location in the path.
  - When the bot has finished moving, the next location in the path is chosen.
  - If there are no more locations in the path, the bot stops until the path is recalculated.
- Also note that the bot turns to look at the location specified by the facing vector. If this value is null, the bot just looks at the direction it's traveling. You'll use this later when you want a bot to focus its attention on, say, the player.

Making a PathBot (4)

- The last thing that PathBot does is handle collisions. If a collision with a wall or an object is detected, the bot backs up, stops, and then waits a few seconds before recalculating the path.

- That's it for path finding! The source code with this book also includes the PathFindingTest class, which creates a few pyramid-shape PathBots in the same 3D map used in the last couple of chapters. The PathBots follow the player around, no matter where the player is.

8. Enhancing the A* Search

- When you run the **PathFindingTest** demo, you might notice some strange behavior in the bots related to the portals. Because the BSP building process splits polygons, sometimes a few portals can exist within a convex room. So, when the bot travels from portal to portal, it occasionally won't travel in a straight line from the bot to the player, even though it looks like it should.

- You can fix this in a few ways. One way is to do extra checks at runtime to see if you can skip a portal in the path. For instance, instead of traveling from portal A to B to C, you would just travel from A to C if there was nothing in the way to block you.

- Another way is to just define the portals yourself by hand, like we mentioned before.

Enhancing the A* Search (2)

- A third way to fix this is what we implement in the next chapter. In that chapter, if a bot can "see" the player, it can just head straight toward it. This isn't a perfect solution, considering things such as small holes the bot could see though but not travel though, but combined with other AI abilities, it works well.

- Another idea is to enhance a path by smoothing out the corners using a Bézier curve. This way, instead of making sharp turns, the bot gradually turns around a corner, presenting a more natural movement.

- Finally, some larger worlds might be so big that doing an A* search between frames could cause a visible slowdown. What you could do is create a low-priority "worker" thread which has the sole job of calculating paths. A bot could make the request to the worker to calculate a path, and later the worker would signal to the bot that the path was found.

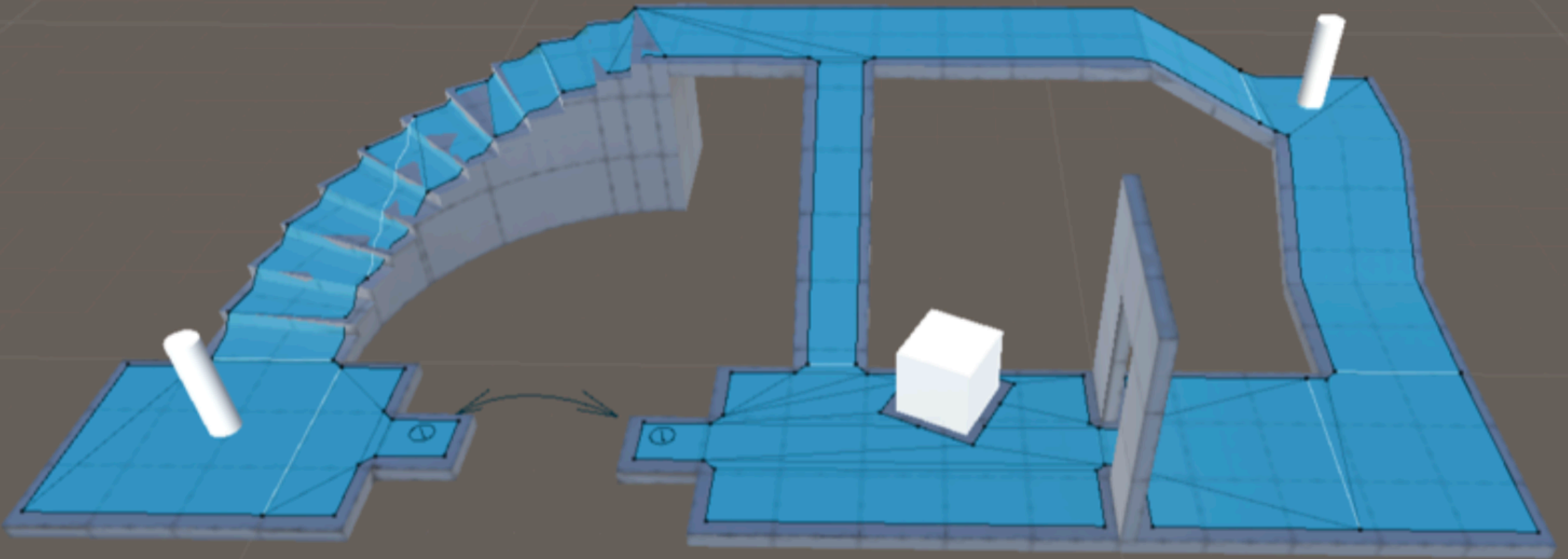
Enhancing the A* Search (3)

- Keep in mind that this worker thread would work only if there was just one of them. Only one path should be found at a time because various fields of AStarNode are changed during the searching process. Two threads modifying nodes at the same time will destroy the path. So, the worker thread could hold requests in a queue (such as a one-thread ThreadPool from Chapter 1, "Java Threads") and perform all path calculations in that thread.
- Also, remember that the PathBot knows nothing of the A* search—all it does is follow the path given to it by a PathFinder. As we mentioned earlier, this means you can modify your underlying search algorithm or give a bot other interesting paths to follow. You'll actually create several different path patterns in the next chapter.
-

9. Summary

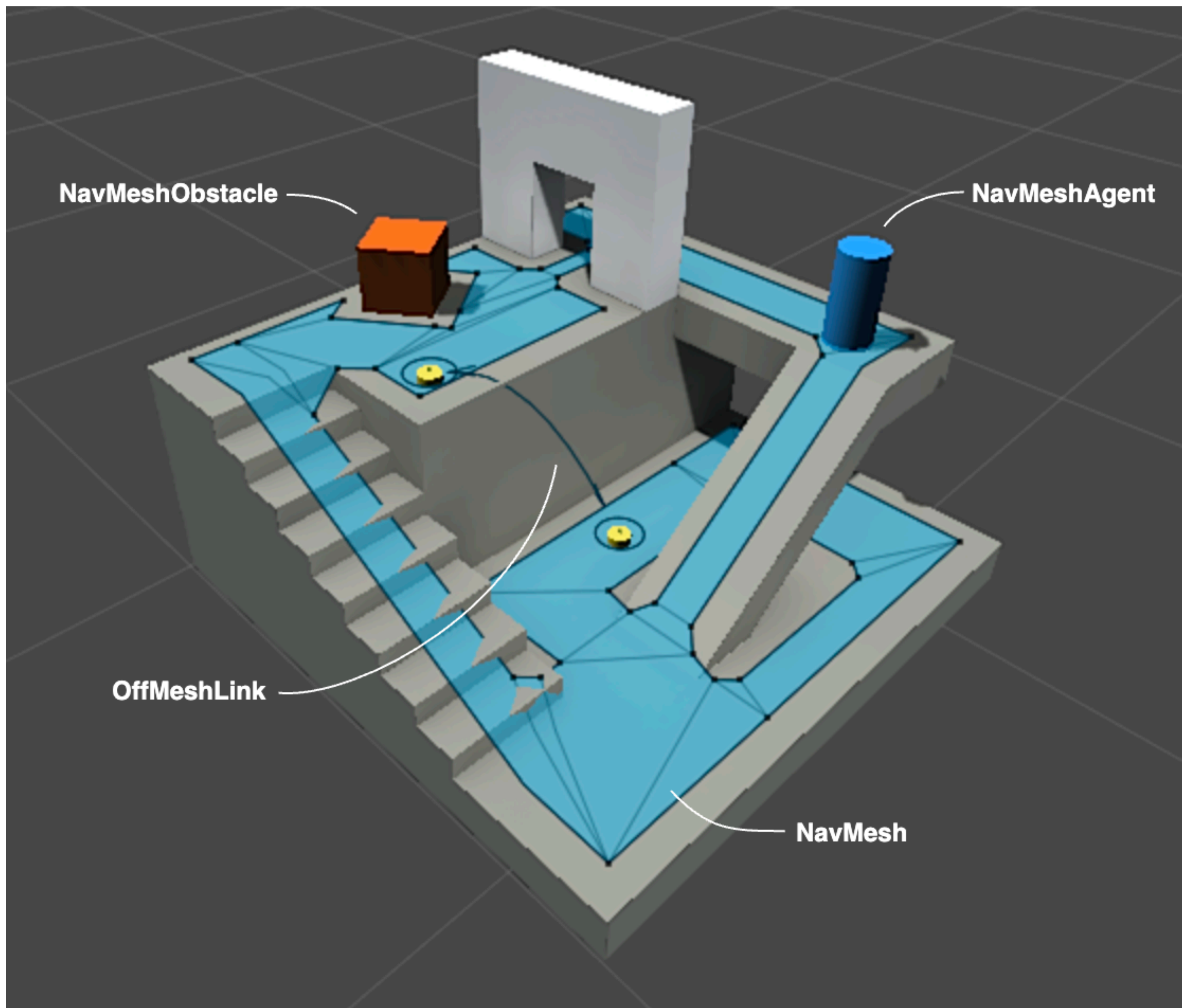
- In this chapter, we started with the basics: Ninjas.
- Actually, we started with some basic path-finding routines on a tile-based map, such as a randomized search and the popular "right hand on the wall" trick. Then we went to more advanced environments based on graphs and implemented a generic A* algorithm that could be used for graph- or tile-based environments (because tile-based worlds could be interpreted as a graph anyway).
- Finally, we merged your A* searching with the BSP tree we've been working with for the past couple of chapters, and we created a generic path bot that follows any type of path, whether it was created by an A* search or not. As usual, we made a demo.
- Path finding is one of those required elements of a game, and it's also the building block of a lot of artificial intelligence in games. Currently, your A* path bots are godlike, meaning they always "know" where the player is.
- Next, we'll use your path-finding capabilities to dumb down the bots' godlike powers, but we'll also give the bots a little more smarts about making decisions such as attacking, dodging, and chasing the player. In addition, we'll make the bots "strike back" so we can fight against our robots.

Unity Pathfinding



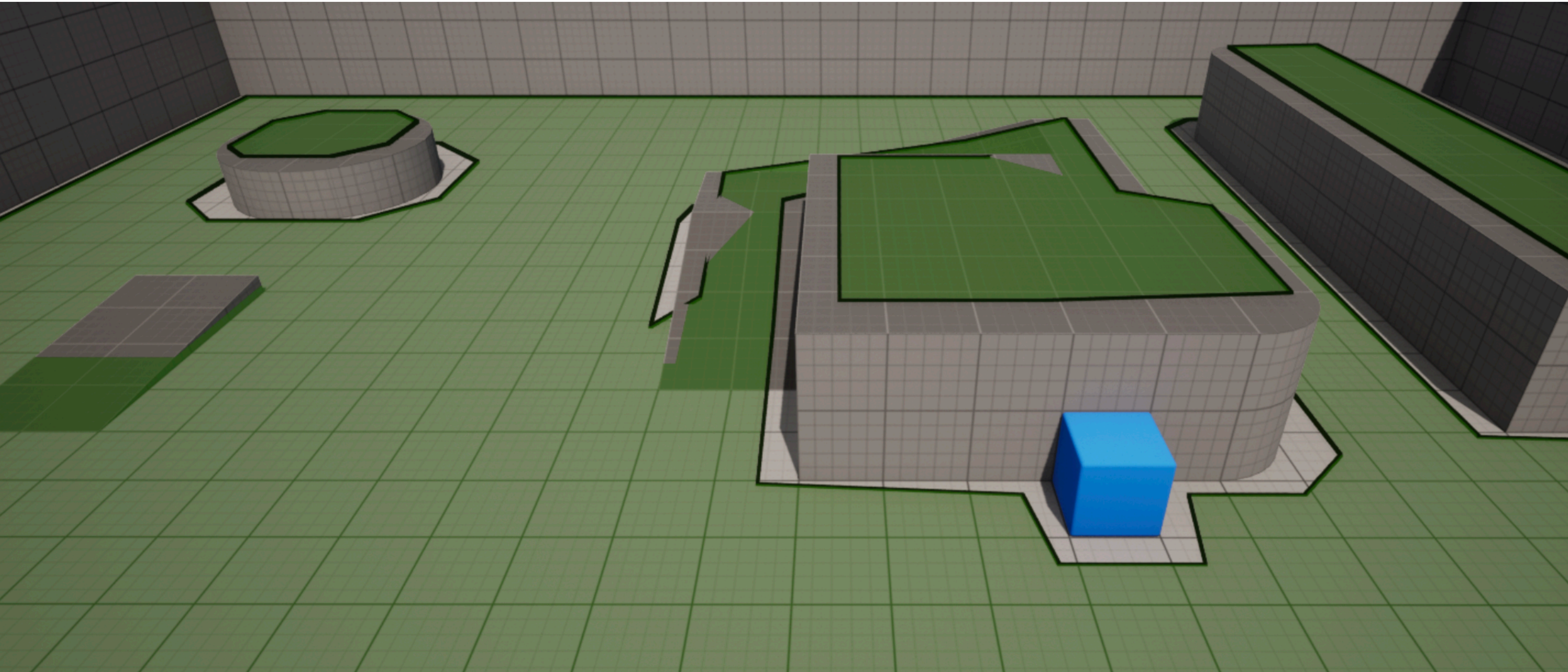
Source: https://docs.unity3d.com/Manual/Navigation.html

Unity Pathfinding



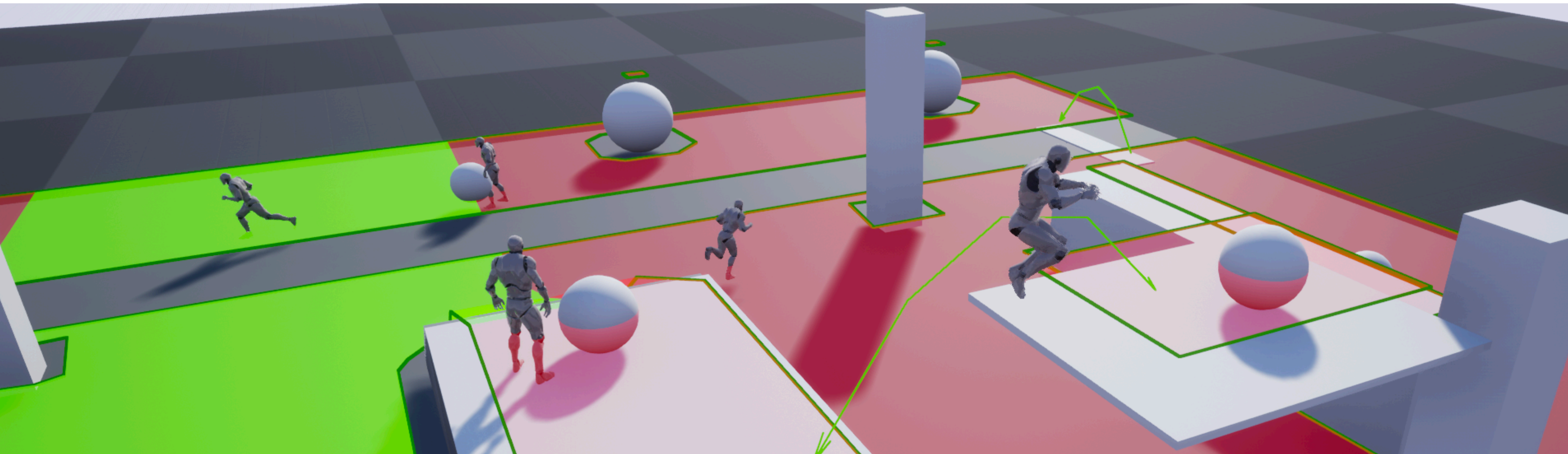
Source: https://docs.unity3d.com/550/Documentation/Manual/nav-NavigationSystem.html

Unreal Engine



Source: https://docs.unrealengine.com/5.1/Images/making-interactive-experiences/artificial-intelligence/navigation/basic-navigation/basic-navmesh-visualize-a.png

Unreal Engine



Source: https://docs.unrealengine.com/5.1/Images/making-interactive-experiences/artificial-intelligence/navigation/basic-navigation/basic-navmesh-visualize-a.png