Ultimate Godot Game AI for Beginners

Game AI - An Introduction

Don't skip this!

Welcome, welcome!

Welcome! I'm so glad you are here. I'm Adrian and I will be your coach through your AI journey using Godot Engine.

You are now on your way to properly understand Artificial Intelligence in Godot. By the end of the course, you will be able to implement your own AI system in Godot, regardless of your experience level.

But there is more - by finalizing this course you will understand how the AI systems work under the hood, how to tweak them, how to customize them to your liking and more.

Course Structure

Now, let's look at how you can get the most out of this course. This course has 7 modules as following:

- Introduction - which you are currently watching
- Understanding the Environment - is about how the AI Agent perceives the environment it can move in
- Finite State Machines - are the concepts, implementation and a small example of one of the most common ways to implement AI decision making
- Pathfinding - is about the algorithms involved in finding a path from A to B using the information from the environment
- Sensors - are all about how the AI sees and feels the targets, items and what's around it
- And the final project - which will be a full implementation of an AI system
- There is an extra module, the Bonus module, which has an introduction to GDscript and Data Structure Concepts

Besides these modules, the course contains PDF booklets for every chapter in the modules. There is also a project that has all the code, assets and everything you need to browse the examples and the final project.

How can you get the most out of this course

So, how can you get the most out of this course?

Beginner - if you are a beginner, I strongly suggest you finish the welcome module and then head over to the bonus one to do the introduction to GDScript and the Data Structures. After this, you can come back and start with module 2. Try not to skip anything as the modules are related and you might need information presented in a previous one.

Medium - if you already know GDScript and Godot, start directly from module 2. You will be able to get a solid grasp on all the AI concepts to get you started.

Advanced - if you know Godot and some AI, feel free to browse through the course and watch the modules that seem interesting to you. You can also just head straight to the final project.

What is game AI and why should you care

Here we are, in 2022 and most of the top games are multiplayer. A legit question that everyone has is “why should I learn AI when I can make a multiplayer-only game?”. And this is a very good question as well.

The main issue with making multiplayer games only is that all of them compete on a finite resource and that’s not money. It’s time. Which is limited. Most of the big multiplayers already saturate this market.

People might get bored and change, right. But when their friends are still in that game, when they invested hundreds of dollars in items, the decision to let the game go is even harder. This is called sunken cost fallacy and it’s real.

Let’s say your multiplayer is successful. In that case, a skyrocketing game will face a totally different set of issues. From the need to implement lag mitigation techniques to maintaining a lot of servers and handling cheaters, as your game grows in popularity things don’t get easy.

In most cases, an indie multiplayer-only game will become a dead on arrival game. That means that the players at launch won’t be enough to make the world feel alive and eventually get bored and leave before new players join. When the new ones join, they will be greeted with a lot of tumbleweeds.

I know this best since I experienced it first-hand.

How can AI, or Artificial Intelligence, help your game?

If your dream is to create a multiplayer game, here’s how AI can give your game a strong competing chance. By implementing AI Agents that will fill in for the players, the game will become alive.

The players will replace them one by one when joining and those already in will experience a world that has a lot of opportunities.

If you are working on a singleplayer game, then AI can help your game A LOT. Of course, not all single player games do need it. For example puzzle games. But, the majority does benefit from a proper AI implementation.

Different types of Game AI

When people discuss AI, automatically they think about Enemies. But that's only one way the players experience AI systems in modern games.

What are the most common types of AI?

AIs as Enemies - this is the most found implementation in games. Usually, everything that comes to you and shoots to kill is most likely an Enemy. Enemies come in different flavors to spice up the gameplay: aggressive, defensive, tactical.

AIs as Companions - is there a Non-Player Character following you? Then, most likely you got yourself a companion. Be it either for a specific mission or a squad-based game, companions play an important role. Games like Mass Effect or Gears of War have a lot of mechanics focused on companions.

Do it right, and your game will benefit a lot. Have companions going through walls, dying randomly and failing the mission - and your game's player will get a bad experience.

AIs as your own units - are you issuing orders to a squad of units in a Real-Time Strategy game? Then, most likely, you are ordering smart AI Agents a tactical decision and they handle the movement, patrolling, attacking or any other action they have.

AIs as your own character - are you playing a top-down role playing game where your character moves around a dungeon? Most likely the AI is controlled by a pathfinder that decides where to move exactly, based on your specified move location.

AIs as invisible helpers - what if I told you that a game that has 0 non-player characters and in fact, any other moving parts - has AI? It's true. I'm referring to Jonathan Blow's "The Witness" where the AI was used to debug the map to find out the walkable areas.

AIs can take many forms, the limit being the imagination of the developer. My hope is that by getting all the information I provide in this course you will be on your way to make Great AI Systems that will improve your game.

Next, let's look at how an AI can handle decisions.

Decision Systems

Before we can discuss decision systems we must first look at what makes an AI work - what are its parts.

This is possible because of the following systems:

- **The Brain**: it's responsible with the decision making
- **The Senses** (or sensors): are the AI's way of detecting enemies, obstacles and more. They are linked with the brain to provide the accurate information in real time
- **The Body**: receives actions from the Brain and executes them
- **The Legs**: are used to move the character. They implement special pathfinding algorithms that compute the route from A to B for the AI to take

Let's focus on the brain of the AI - the decision systems.

Finite State Machines - This AI is used a lot. Actions are mapped to states and the AI can be in a particular state at a specific time. The states are interconnected together with transitions that get triggered when a particular change happens to the agent.

Games: DOOM, Quake

Behavior Trees - BTs are very common in most game engines, having direct integration in Unreal Engine. Their main selling point is that a designer can create an AI system without having to interact with any code piece.
They are also very flexible to use.

Games: HALO

Planning - This one is very similar to the State Machines, but the main difference is that the states are not connected between them. This gives total freedom for the AI to pick any available option.

Games: FEAR

Utility AI - The Utility AI does not have a particular structure, but rather a set of rules. It's mostly used to handle complex actions like an RPG character that has multiple skills and needs to pick one or another based on the current situation.

Games: Dragon Age: Inquisition

Understanding the environment

How does the AI navigate in the game world

The AI Agent is not only just a collection of decision-making algorithms. Those decisions need to be applied to specific contexts. For example, the AI needs to move around, avoid obstacles, go to certain targets, etc.

To achieve this, the AI need to be able to "understand" the environment it's in:

- Where it can move or cannot
- Where are the points of interest e.g. cover points, important locations
- Where is the enemy

While the player can perceive the environment by visual cues like 3D objects in forms of paths, obstacles the AI Agent needs to get a processed version of that in a language that it understands.

If we use the exact environment as-is, the AI will have a hard time computing everything because usually the environments are too complex. Even in 2D games, the complexity of the visual world is too much to handle for an AI instance. I'm not saying that one AI cannot handle it, most likely it can. But when we are discussing AI Agents, we always need to take into consideration the amount of CPU used when running 20 agents, 50 agents.

We need to use a simplified version of the world. Here are the most common ways to achieve this:

- **Grids** - a set of rows and columns that form a 2D table
- **Graphs** - a collection of points that are connected between them
- **Navmeshes** - a simplified environment mostly auto-generated by modern game engines

Let's look at these 3 and see their Pros and Cons.

Grids

Grids are the most common way to create 2D environments but don't be fooled by their 2D capabilities as they can also be used in 3D games that are limited to a flat surface.
A grid is a set of lines and columns that are put together to form a 2D table. Each point can be referenced by using a pair of row/column indexes.

If you want to learn how to implement grids, head over to the bonus module and look at Data Structures.

What are the PROs and CONs of using grids in your game?

PROs of a grid are:

- It's **easy to use**. Create a 2D array of elements and it will already work with the map structure (if the map was designed to support a grid in the first place).
- It **supports real time changes**. Does your game have actions that can change the map? By having a grid you can easily set what's walkable and what's not.
- Most game engines have grids implemented already, even with editing capabilities. They are called tilemaps and mostly used for 2D games. Godot also has a 3D alternative called the GridMap.

CONs of a grid:

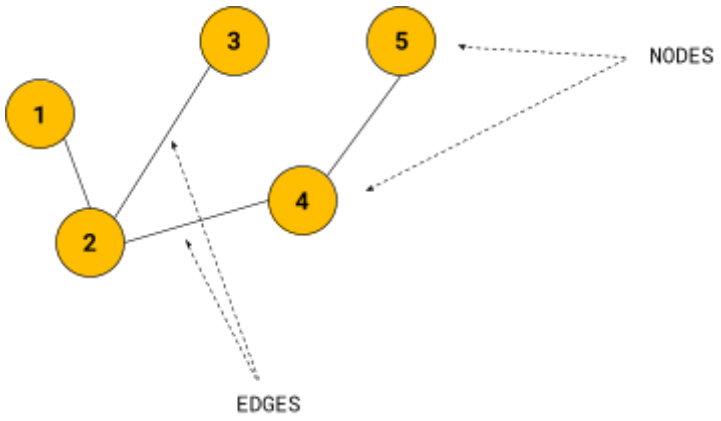
- It has a **fixed shape.** It will always be square shaped. Of course, you can disable the sides or skew it to give it more variation.
- It **cannot support height variations**. Does your game map have hills, valleys, etc? Then a grid might not be what you are looking for.
- It might **not create enough variation**. Having a grid city might not be fun. But there are cities with grid layouts like New York which are interesting.

If a gridlike structure sounds too restrictive for you, then you might want to try Graphs. Let's look at them in the next chapter.

Graphs

Do you want to make dynamic cities, maps that have height variations, even breakpoints? Then graphs might be the data structure you are looking for!

A graph is a data structure made out of a collection of nodes and edges. The nodes are the endpoints (containers) where the edges are the connections between them.



Using graphs we can create city streets, map regions or even grids.

To find out more about how they can be implemented and used, go to the graph section in the bonus module of this course.

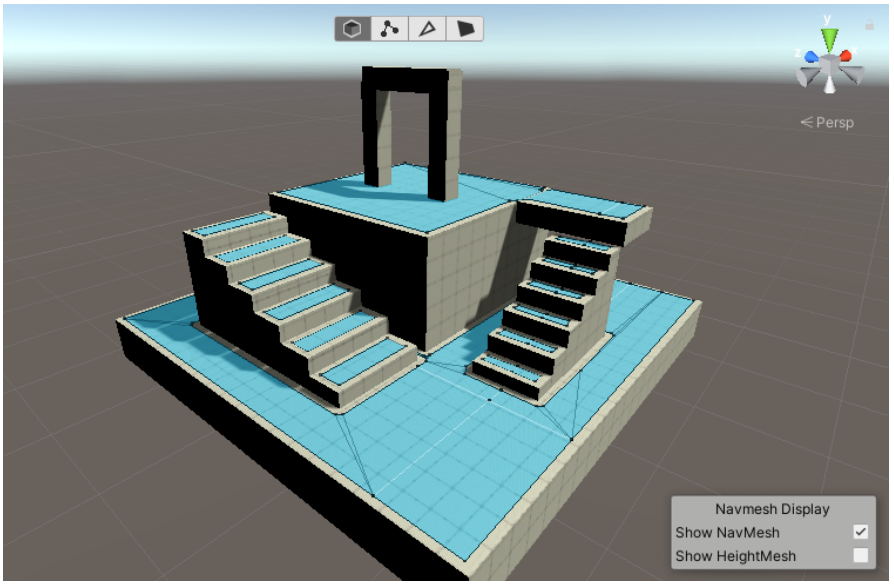
PROs of Graphs:

- Can be used to **model any type of terrain, height**, street intersections, even grids.
- Can be **easily updated** with new changes.

CONs of Graphs:

- **Harder to work with than grids** - the game needs to have a way to map all the points on a graph.

Navmesh



The navigation mesh is one of the most common ways for the AI Agent to navigate on the map. Its implementation is found in most modern game engines such as Godot, Unity and Unreal.

A navmesh creates a simplified model of the 3D objects that are walkable using simple polygons and connects them. Then, when queried for a path, it quickly determines a way using a pathfinding algorithm such as A*.

PROs of a navmesh:

- **Already implemented** in most modern game engines.
- **Easy to use** - one button click.
- **Optimized** for efficiency.

CONs of a navmesh:

- **Supports only a particular type of map** - either made of continuous meshes or a terrain. Does not support grid, hexagon, or a totally different map concept
- **Not customizable**. All the logic of how the movement path is computed is inside of it.

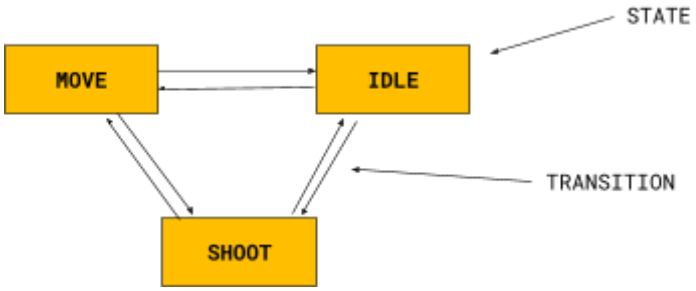
Finite State Machines

General concepts

A Finite State Machine is a system that uses States and Transitions to describe complex patterns that can be used in creating AI behavior, animations and more.

A FSM is one of the common ways to implement decision logic for the AI.

Each state represents an action (in the case of an AI) or an animation. They are interconnected by transitions and to move from one to another state there needs to be a fulfilled condition.



Behind the scenes, a FSM is a graph-like structure where its states are the nodes and its transitions, the graph's edges.

Finite State Machines are one of the first ways to implement AI Agents that appeared. It's commonly used today as it fits a lot of cases. It does have PROs/CONs.

PROs:

- Easy to use & understand
- Fast

CONs:

- Can get out of hand quickly if a lot of states are added
- It is hard to maintain for complex structures
- It is not reusable, as its parts are tightly linked together

There are ways to mitigate these issues, using Hierarchical State Machines. The main difference between the classic ones is that the Hierarchical ones have nested states inside states. Let's call them state ception.

For the final project we will use a finite state machine from a library but for the next example we will implement a simple one from scratch.

It's important to implement a FSM to understand its inner workings. Let's find them out right now.

How to implement a FSM in Godot

Now that we have covered the basics of a Finite State Machine, let's look how one can be implemented using GDScript.

Note: there are many ways to implement a Finite State Machine in GDScript. This is one of them.

We will have 2 types of scripts:

- The main FSM - which is responsible for changing and running the states and also keeping track of the values in a small database and also
- The others - are the actual state implementations

Let's look first at the finite state machine.

```
extends Node

class_name CustomStateMachine

export(String) var entry_state

var current_state

var db = {}
var state_list = {}

func _ready():
	for state in get_children():
		state.fsm = self
		state_list[state.name] = state

	change_state(entry_state)

func _process(delta):
	if current_state == null:
		return

	if current_state.has_method("process"):
		current_state.process(delta)

func _physics_process(delta):
	if current_state == null:
		return

	if current_state.has_method("physics_process"):
```

```
          current_state.physics_process(delta)

func change_state(new_state):
     if current_state != null:
          if current_state.has_method("exit_state"):
               current_state.exit_state()

     if !state_list.has(new_state):
          return

     current_state = state_list[new_state]

     if current_state.has_method("enter_state"):
          current_state.enter_state()

func set_value(value_name, value):
     db[value_name] = value

func get_value(value_name):
     if db.has(value_name):
          return db[value_name]

     return null
```

This script can be attached to a node that will contain all the states in the FSM.

Next, let's look at the structure of a state.

```
extends Node

var fsm:CustomStateMachine

func enter_state():
     print("Entered ", name)

func exit_state():
     print("Exited ", name)

func process(delta):
     check_conditions()

func check_conditions():
     pass
```

For each state, we can create individual children nodes and create scripts similar to this one, but having specific commands.

In the next video let's look at a small example that uses this implementation.

Mini-example with the created one

In this FSM example we will use 3 states: **IdleState**, **MoveState** and **FireState**. Every state will be connected with each other and will trigger transitions when specific conditions are met.

Let's configure the FSM example script first. Here, besides updating the UI, we need to provide the values is_moving and is_firing to the custom state machine. These will be used to trigger the transitions.

Now we need to create 3 new nodes, IdleState, MoveState and FireState. Each with its own script. The scripts themselves will be very similar to the one implemented earlier.

In the idle state, we need to check the values of the finite state machine for is_walking or is_firing. If one is active, the fsm needs to trigger a state change. It's good to break the code execution using return. Otherwise, there might appear issues with both conditions being true at the same time.

In MoveState and FireState the conditions are a little different. As long as their initial trigger is TRUE, we need to keep the current state up. But, as soon as the flag turns false, it's time to change it to the proper state.

With this you have completed the Finite State Machine module. Congrats! While FSMs can be easy to use, they are powerful and can handle a lot of AI types.

Let's find in the next module how to get an AI from position A to position B using pathfinding!

Pathfinding

How pathfinding works

Finding the path, or the sequence of steps, from A to B might not be as easy for an AI Agent as it is for a human. Luckily, we have pathfinding to save the day.

Pathfinding is the method of searching the shortest path from a point to another. At the core, pathfinding searches a graph (see the bonus section, data structures) starting from one of its nodes and, from neighbor to neighbor.

An algorithm that appears everywhere in gamedev is A*. In real-life use cases, this algorithm is mostly hidden behind layers upon layers of other systems that make use of it.

And it should be simple - because otherwise we will be reinventing the wheel over and over again. The most common issue here is that while modern game engines do provide a simplified process to it, it removes the proper understanding regarding this system.

Why is this a possible concern for a game developer? Because games usually need special cases like custom maps with hexagons, unique features and more. And just by using the tools at hand that task might be close to impossible.

Behind the pathfinder

At the core of any pathfinding solution, there are two main concepts:

DOMAIN + ALGORITHM

The Domain - Where it happens

Don't get tricked when thinking that the space means the terrain heightmap or the plane where the AI Agent needs to find its way. The space is the representation of that space in a way that is:

- **Fast** - having the full environment for running the algorithm over and over again is not really doable in a game project. We need a simplified version of it, similar to a High-Poly 3D Object's Low-Poly counterpart.
- **Reliable** - being fast but inaccurate is not ideal as well. Having AIs running through walls and other obstacles clearly diminishes the player's game experience.

- **Stored properly** - if we write the simplified version on a piece of paper it won't help our CPU process it. We need to use a proper data structure to hold all of this newly generated information.

The Algorithm - How it happens

There are many algorithms capable of doing pathfinding. Most of them work very similar to one another. The main difference is their implementation complexity and their CPU usage.

They can be categorized in two types:
- **UNINFORMED** -> does not know details about the map, deals with the situation as it unfolds (when an obstacle is encountered)
- **INFORMED** -> has additional information and does something extra for each search step

When speaking of pathfinding A* is the industry-standard because it produces fast results.

A* is an algorithm that belongs to the **INFORMED** type. That means, in order to properly run, A* does require more information about the map than just the obstacles.

In the next chapter let's look at a pathfinding algorithm, how it works and how it's implemented.

Anatomy of a Pathfinding Algorithm

How does an AI know how to get from point A to point B on a map? What is the code sequence that enables that? The code sequence is an algorithm, and there are many pathfinding algorithms out there.

In this chapter we will look at a simple pathfinding algorithm called Breadth-First Search or BFS. BFS is an **UNINFORMED** algorithm that deals with the situation at hand. Its end result is a path between a starting node A, and a final node B.

Let's deconstruct the algorithm and see how a pathfinder works!

The Space

First, we need a space (or a domain). We will use a grid because it's one of the easiest ways to implement it. Any position on the grid can be referred to using 2 indices: row index / column index.

The grid's elements will be marked with numbers as follows: 0 for empty space and -1 for wall.

The Queue

An integral part of the BFS algorithm, the Queue is a list that only allows elements to be added at the end of it and removed only from the beginning of it. You can find more about this data structure in the Bonus module - Data Structures.

For now, keep in mind that the queue acts like a real life cinema queue.

Movement Patterns

Our AI Agent will be able to move in 4 directions. This can be easily scaled up to eight directions, or even more.

As an example, let's say the AI is at position: 3, 3

To move:

- Up [3**-1**, 3] => [2, 3]
- Down [3**+1**, 3] => [4, 3]
- Left [3, 3**-1**] => [3, 2]
- Right [3, 3**+1**] => [3, 4]

Obs: for the same pair of values [3, 3] which represents the current position, if we add another set of numbers we can generate the end values which will result in the nearby positions.

For this, we can use direction lists. These are 2 lists that represent a X, Y pair for each direction UP, RIGHT, DOWN, LEFT. If we take each one and add it to the current position, we will end up with the next positions.

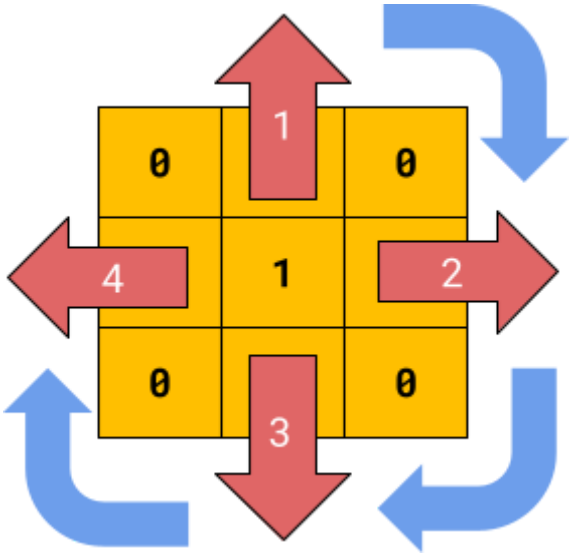
Direction List Table:

direction	up	right	down	left
dx	-1	0	1	0
dy	0	1	0	-1

Or, more specifically:

dx = [-1, 0, 1, 0]
dy = [0, 1, 0, -1]

Given an x / y coordinates, if we apply the following directions we will get the four possible moves: `UP`, `RIGHT`, `DOWN` or `LEFT` (in this order)



Breadth-First Search

Let's do a recap: we have the space which is a grid, we have the movement pattern. We also need a data structure called Queue.

Let's go through the steps performed by BFS and after, let's look at some examples.

Step	Action
1	add the starting position to the Queue
2	while the Queue is not empty
3	get the first position from the Queue
4	if the position is the destination, exit
5	get the positions possible from the combination of the current position / directions using dx/dy lists
6	fill the positions with the next step's value
7	add the valid ones back to the Queue
8	go to step 2

Running a full BFS will generate a filled grid with how many steps are needed from position A to reach that particular one. To get the steps sequence, we need to do one more step which is reconstructing the path.

Reconstructing the Path

The last step of the Breadth-First Search is to reconstruct the path so that we can use it later to move the AI Agent where we need it to be.

To do this, we need to use a recursive function. A recursive function is a function that calls itself in it. More about this in the Bonus section, introduction to GDScript.

Don't worry if that sounds complex, we will take a look into it very soon!

```
reconstruct_path(current_x, current_y, start_x, start_y, map)
   if current_x == startx && current_y == start_y
      // we arrived at the start!
      return [{current_x,current_y}]

   for (dir_index = 0; dir_index < 4; dir_index++)
      if map[current_x+dx[dir_index]][current_y+dy[dir_index]] <
         map[current_x][current_y]
         return reconstruct_path(current_x+dx[dir_index],
             current_y+dy[dir_index], start_x, start_y, map) +
                   [{current_x,current_y}]
   return []
```

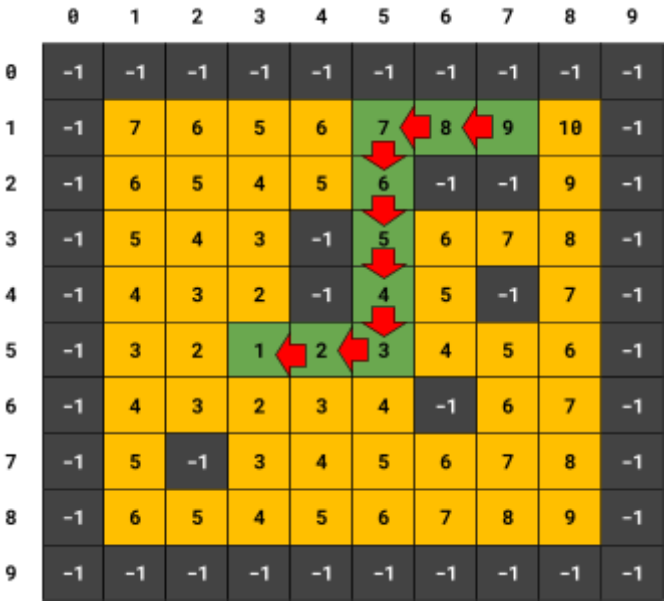
Our function will take the following parameters:

- **`current_x`** and **`current_y`** representing the current X and Y position in the map
- **`start_x`** and **`start_y`** representing the initial origin of the algorithm's start
- **`map`** representing the 2d array where the algorithm was applied

The first thing we need to check is if we have arrived at the destination. That is the end goal of this function. If we do reach the end, we return a list with a single pair: the `current_x` and `current_y` which are the same as `start_x` and `start_y`.

If not, then it means we are not done! For each possible direction, we need to check where the step decreases from our current one. When such a step is found, we can restart the algorithm from that point on.

This can only be done by returning the future list of steps combined with the current one.



PROs/CONs of Breadth-First Search

PROs

- Easy and fast to implement
- Easy to debug
- Easy to visualize
- Can be used in different scenarios, not only pathfinding

CONs

- Slower than A*
- Not quite useful for large datasets

BFS can be used for quick custom implementations. Alternatively, you can use the already created A* implementation in Godot. They do work similar but A* is more efficient.

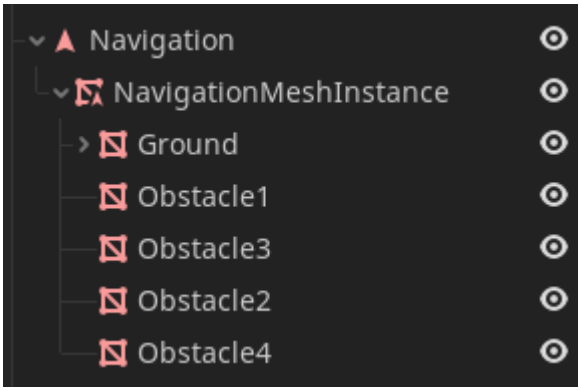
In the next chapter we will look at Godot's Navigation mesh implementation and how to use it.

Navigation mesh in godot

If you use 3D objects to define your terrain you are in luck! Godot provides a solution for you out of the box - enter the NavMesh.

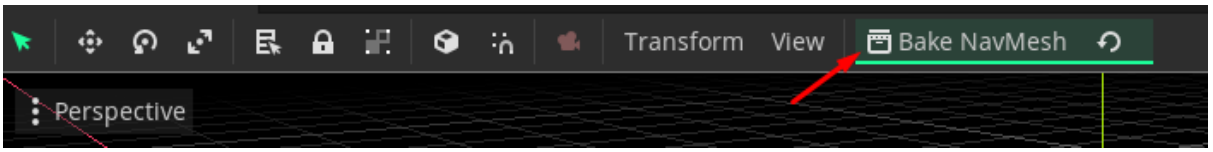
It's quite easy to create and use a navmesh.

Godot provides 2 key nodes for it: the Navigation and the NavigationMeshInstance.

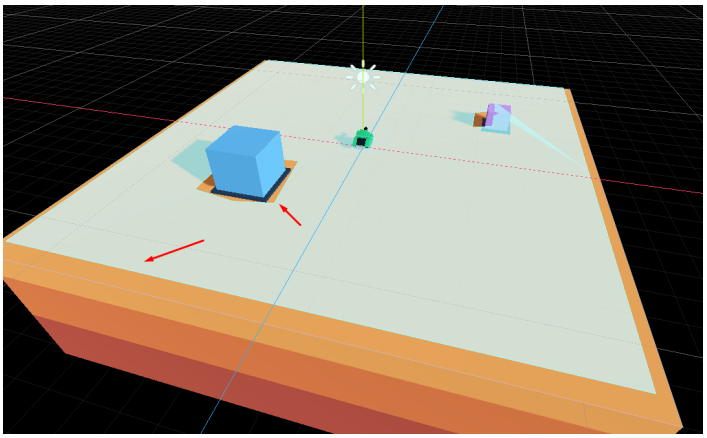


Once you have the NavigationMeshInstance parented to the Navigation node, you can include the map or the parts you want the AI to move on as children of the Nav mesh instance.

Once this step is completed, you need to select the navmesh instance and hit the bake navmesh.



Once the generation is done, it will be clearly visible in the editor view.



Congratulations! You just generated your first navmesh.

To generate a path sequence with a NavMesh, you must do the following:

- In the script of the AI you need a variable to point to the Navigation Node
- Use nav.get_simple_path

The code should look like this:

```
onready var nav = get_parent() # the AI needs to be parented to the Navigation

func get_custom_path(target_pos):
  var path = nav.get_simple_path(global_transform.origin, target_pos)
  return path
```

Now, if you specify a target value that is on the navmesh, this function will return a sequence of steps that the AI can take to reach it.

Currently Godot 3 does not support dynamic objects in a navmesh, only static, baked at edit time. Godot 4 on the other hand, has this planned so keep an eye on it.

This is the final lesson from this chapter! Congrats for reaching this far! You are halfway into the course already.

In the next section we will look at sensors, or how the AI can perceive what's happening around it.

Sensors

Sensing the Game World

In order for an AI Agent to take actions in the game there are two important factors: understanding the environment (which we discussed in a previous module) and sensing the threats/targets/items/etc.

There are many senses that can be implemented in an AI e.g. sight, hearing, hit detection.

In this module we will focus on two of the most important ones:

- Seeing the enemy. This includes detecting everything in the AI's range, filtering only for the cone of view and, lastly, using raycasts to detect objects that might occlude the target.
- The other sense is hit detection. This one is needed to detect when the player shoots at the AI Agent in order to retaliate.

Let's look at each one of the sensors, how they are implemented and how we can use them in a real scenario.

Implementing Range

Let’s start with range. First, we need to define what is range, what should the sensor do, what are its targets and how it can work.

Range can be visualized using a sphere that has its center point in the AI Agent. The sphere can be bigger or smaller. Based on this, the AI will have a bigger range or a smaller one.

Let’s limit the sensor to only enemies. But it can be extended very easily to support other points of interest such as pickable items, cover areas and more.

The sensor will use a list that will be constantly updated with the targets as they are coming in the range or moving out of it.

We will also use Godot’s event system called signals. They will be triggered when an enemy target has either entered the range or left it.

Let’s look at the code now.

If you run this example, you can see when the AI Agent detects the target and when it does not.

By only implementing this, the enemy will have eyes in its back as well. In most cases, we don’t want that. Enemies usually have a cone of view like a real person’s field of vision.

Let’s look at the next tutorial on how we can implement this functionality.

Field of View

Finding out what targets are around the AI Agent might not be ideal in most situations. We want to have AIs that behave like humans. Human's field of view is limited by the eyes.

Here comes the field of view. Basically, field of view or FOV, is an area specified by the forward direction of the agent and an angle. What is within it, it will be visible.

For this we will need to make use of a little math.

What we want to achieve is a function that receives a target and returns TRUE or FALSE if it is inside the field of view of the AI Agent.

We will need 4 parameters internally:

- The position of the AI Agent
- The forward vector for the AI Agent
- The target which we want to check
- And, The field of view in degrees

First we need to determine the direction from the Agent to the target. We can achieve this by calculating the difference between the target's position and the AI Agent's position. Once we have this, we can normalize its value. Vector normalization means that all the values are clamped to the [0,1] interval.

Once we have the direction, we need to calculate the dot product between the vector that represents the forward direction and the difference vector (or the direction).

We need to check if the value is greater than the cosine of the field of view angle.

Now we will make a second function that will be used from the AI Agent to filter the targets.

If you check the example provided you can see how the field of view detects or not the target by moving it around.

Now that we have the range detector and the field of view detector there is one final step for detecting targets. We need to determine if a target is behind an obstacle or not. The first two sensors will say that the target is visible so they are not enough.

In the next lesson, let's look at raycasts and how we can use them to determine if the AI Agent actually sees a target or not.

Raycasts

Now that we can determine if a target is in range and in the field of view, we also need to determine if the target is not occluded by an object like a box or other environment prop.

To do this, we will use raycasting. Raycasting is the process of generating a ray, a line, from the AI Agent towards the target. If the ray intercepts something in between, we will return FALSE and if the ray successfully hits its target, then the result will be TRUE.

Raycasts in Godot can be implemented in code but there is a nicer way, provided by the engine. Using the RayCast Node. This node will cast a ray from its position to the cast to point. If it's enabled, it will also detect collisions.

As for the raycast detector script, we need to get a target and set the ray to point on it. The next step is to check the collision and see if it's the same as the target or not.

Let's look at the demo provided and see how it works.

With all these sensors in place: range, field of view and raycasts our AI Agent can properly determine what target it can see.

The next step is to alert the AI if it's being fired upon from behind.

Hit Detection

What happens if the AI Agent gets attacked from behind? In this case it would be a good idea to retaliate as, otherwise, the AI might seem unresponsive.

The hit detection is a little different from the previous sensors. The main logic is for the projectile that hits a target to call a specific function inside it.

This will alert the AI Agent that hits are taken and it will prompt a defensive or offensive action.

For this we need to prepare a couple of things:

1. Create a bullet instead of the target
2. Use an Area to detect collisions
3. Attach a collision shape to match the bullet

In the bullet's code, we first need to set up the signal coming from the area, called body_entered. Now we know that a collision was detected. The next step is to find out if the object we hit has the function called "register_hit".

The collision is detected at the KinematicBody level so we need to get the parent which is the AI Agent. Next, we check if there is a function and call it.

In the AI Agent register it needs to be implemented as well.

Once everything is in place, let's test the project and see the result. Every time the projectile enters the AI's physics body it will trigger a hit event.

This can be further processed into decisions.

Now that we covered how to deal with sensors, state machines, let's head over to the last step in this course: creating a fully-functional AI Agent that will use all of this.

Project State Machine AI

- project overview

Top down hack slash where the enemies need to engage close range and patrol. There are a couple of different AI types from no-brainer to the one that can use smart objects against the player.

What we will be creating (5min)

Defining the AI (5m)

Creating the FSM (10m)

Hooking up the FSM with the code (10min)

Implementing Actions (20min)

Implementing Cover (15min)

Putting everything together (20m)

Bonus

Quick introduction to programming

If you are a beginner in programming, don't skip this!

Since we will be using code to implement various systems, it's always a good idea to get up to speed. All the code will be in GDScript, Godot's native scripting language.

Let's look at the core building blocks and how to use them to create more complex functionalities.

Variables - they are used as data containers. They can store numbers, strings, even more complex things as objects, materials and whatever the developer needs.

They do come in more shapes as sizes. Usually, when creating a new variable it can store only one thing at a time. This is good for values like Hitpoints, Damage, Speed, etc.

When we need a list of elements, let's say a list of words, we can use **Lists**. Lists or arrays are also variables. The main difference between them is the fact that they can store multiple values, chained one after another. They are good to store groups of items: Inventories, enemies, projectiles.

Operators - with variables alone we can't do much. Not even store data. Not because they cannot, but because we cannot put data inside of them without the use of "=" equals operator.

Let's create 3 variables A, B and C and assign them 1, 2 and, lastly, a sum of both A and B, in C. Notice we have used another operator to do that, +.

Here are some other common operators used in GDScript: -, *, /, and/&&, or/||

You can find the full list in the link down below. But this is a good start.

Now that we can store and manipulate values, we need to talk about the execution of the code. This happens from top to bottom, line by line. But having a list of instructions doesn't really make anything useful.

Here is where the **flow control** comes into play. In order to have branching code that does not do the same thing every time we execute it, we need to make use of instructions such as **if**, **else** and **match**.

A simple example would be to check the value of a number and show different answers based on that. If we run the code and change the value, we can see that the response gets changed accordingly.

The match instruction offers more flexibility. We need to provide an expression and then, based on the single value it returns, to get a code branch.

Let's take for example a string that can be of different types.

Flow control does not stop at if, else and match! The code can also be repeated using **while** and **for** instructions.

While can be used to decrease values. Let's see an example.

For can be used to enumerate lists of elements for numbers. Let's enumerate a list and numbers.

Once you start writing code, you will find out that parts of the code repeat themselves. This is normal in programming. When you happen to find this kind of pattern, you can make it reusable by creating a **function**.

A function is a separate piece of code that can be called on demand, by specifying its name in the execution. Here is an example of add(x, y).

Good to take note here: Function and method are used interchangeably in most cases, but there is a difference: functions can be stand alone while methods are bound to a specific class. But what is a class?

What happens when we need a different data type combined with specific functionality. Let's say a player that has a function to take damage. In this case, we can use a **class**.

A class can be defined by simply the class instruction and a name on top of any script.

We can make use of a class in another script. Another advantage of classes is that they can be inherited. That means that all their variables and functions trickle down to their children while adding more functionality.

Let's take the simple example of a base enemy which has health and take damage. Then, we can proceed and inherit from that to create a soldier, a big boss and whatever enemy type we need.

Now that we know what a function is, let's take it one step more: enter recursive functions. When a function has calls to itself it's called a recursive function.

But having calls to itself might lead to infinite loops! This is a common issue when implementing recursive functions. Luckily, there are ways to mitigate this. One of them is using what is known as "exit conditions". They provide a way to exit the recursive loop.

Let's look at this small example of a function that prints 5 numbers.

That concludes the programming primer! For more about GDScript, I've linked some great resources below.

Data Structure Concepts

What are Data structures

In order to implement Game AI we need ways to store information. And sometimes variables and lists are not enough.

Arrays or Lists are data containers of the same type, chained together. This creates the idea of a list. E.g. `[1, 2, 3, 4, 5]` is a list of 5 elements: 1 2 3 4 5. A list can also be a string "Hello World". They have a lot of use-cases in general purpose programming and also in developing games.

The next step is to take it to another dimension: 2D Arrays are basically grids of elements of the same type.
E.g.

```
[
    [ 1, 2, 3 ]
    [ 4, 5, 6 ]
    [ 7, 8, 9 ]
]
```

This is very similar to a keypad. This can definitely work as a keypad implementation!

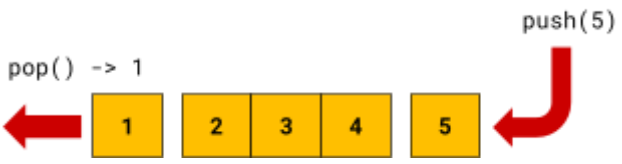
The Queue

Another important piece of the puzzle is the data structure Queue. This one has a particularity: it is an **abstract data type**. An abstract data type means that it can be implemented in multiple different ways.

A Queue is a List (an Array) with 2 or 3 functionalities:

- You can *only add elements at the end of it* (`push()`)
- You can *only take the first element* (`pop()`)
- You can only see what is the first element (`peek()`)

It is very similar to a real-life cinema queue.



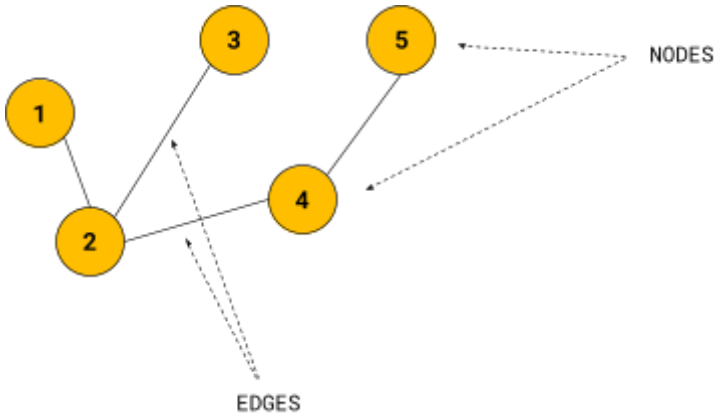
Why would we need such a restrictive structure? The answer is simple: some algorithms make use of it in their implementation because they need sequenced values that also have a particular order.

The Graph

Before we can understand Graphs better, let's look at some other common data storage options for coding:

- **Variables**: they can contain one value of a data type (number, character)
- **Lists** (or arrays): they contain a series of variables of the same data type
- **Multidimensional List**: contains a grid of variables of the same data type

A node in the graph represents a container - either for data, numbers, actions, etc - and a connection (or edge) represents a connection between two different nodes in a graph.



What can the Graph be used for?

- State Machines
- Mapping the streets in a city
- Creating regions on a map

Implementing Graphs

Graphs are an abstract data type - in practice, that means that they don't have a "fixed" implementation and there are multiple ways to create one.

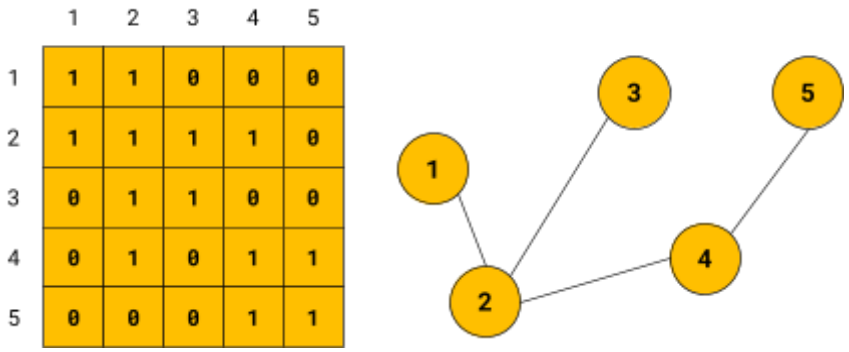
As described before, graphs contain two lists:

- The nodes
- The links between (edges)

Let's look at some of the most common graph implementations out there.

2D Array implementation

For this one, we will make use of a 2d list of elements (grid) with the following specification: each row and column number represents an unique node. If the variable contained there is true, then that means there is a connection between them.



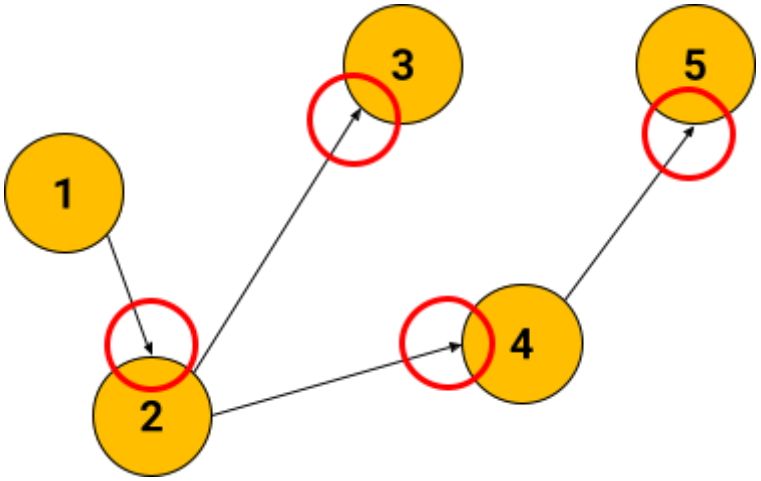
Godot by default does not support 2D Arrays. Luckily, there is a free addon called GodotNext which provides this exact functionality.

```
var grid:Array2D = Array2D.new()

func _ready():
  grid.resize(6, 6)

  for i in range(6):
    for j in range(6):
      grid.set_cell(i, j, 0)

  grid.set_cell(1, 2, 1)
  grid.set_cell(2, 1, 1)

  grid.set_cell(2, 3, 1)
  grid.set_cell(3, 2, 1)

  grid.set_cell(2, 4, 1)
  grid.set_cell(4, 2, 1)

  grid.set_cell(4, 5, 1)
  grid.set_cell(5, 4, 1)
```

Directed graph

A special type of Graph - The Directed Graph

Sometimes when we traverse a graph, we don't want to go back the same way we came - or even not go back at all. For this, another concept is needed: a graph that has its edges one-way. This allows more control regarding what nodes can be visited and when.



Graph traversal methods

A graph is a great data structure that can be used to store a lot of information. In order to access and use that information, we need to use what is called a traversal method, or a way to get from element to element.

The most common ways to achieve this are Breadth-First Search (BFS) and Depth-First Search (DFS).

Before we dive into each method, let's create a simple graph to work with. This is how it looks. We will make use of a grid representation for this implementation.

BFS

The Breadth-First search focuses on expanding at the same time in all directions. You can think of it as water ripples that have a starting point and move outwards.

If we were to run a BFS search on this graph, the result will be: 1, 2, 3, 4, 5

In order to implement a BFS we will make use of a Queue. If you don't know what a Queue is, then revisit the chapter "Data Structures". Implementation and examples are included there.

For the full BFS implementation, check the Pathfinding module.

DFS

to traverse a graph means for a given point to enumerate the connections from neighbor to neighbor. A matrix implementation can be traversed as following:

```
current_point = 1
visited = []

parse_graph(current_point)
   log(current_point) // display the current point
   visited[current_point] = true
   for (i = 0; i < max_nodes; i++)
      if (i == current_point)
         continue // skip the current node
      if (graph[current_point][i] == true && visited[i] == false)
         parse_graph(i)
```

This is one of the standard ways to implement a Graph. Its advantage is that, in only one instruction we can determine if there is a connection between a pair of nodes.

This method of traversing a Graph is called **Depth-First Search** and uses recursion. Recursion when a function calls itself. This can create an infinite loop, so we need to make sure that there are exit conditions. In this case, there will be a point where all the neighbors will be visited and the function won't get called again.