Spider agent on uneven terrains

Artificial Intelligence for videogames

Francesco Bultrini - 970277

Abstract

This report presents a spider-like agent that is able to move on uneven grounds. The agent's movement is based on procedural positioning of its legs according to the surface it is on; also the agent's body can avoid obstacles by rising or lowering its position. Two types of terrain will be tested: one generated procedurally using noise and one that uses meshes to create elevations and roughness on a flat ground. The goal of this project is to demonstrate how procedural techniques can be used to create believable and adaptive movement.

Introduction

Procedural animation is a branch of computer animation that is used for many purposes. In videogames, it is used to generate animations on the fly, rather than pre-animating them. This allows more flexibility and realism in the game's animation system.

One common use of procedural animation is in the movement of characters and creatures. For example, in a game featuring a wide variety of enemy types, each with its own unique movement patterns, it would be impractical to pre-animate every possible movement for every enemy. Instead, a procedural animation system can be used to generate the animation for each enemy based on its movement patterns and physical properties.

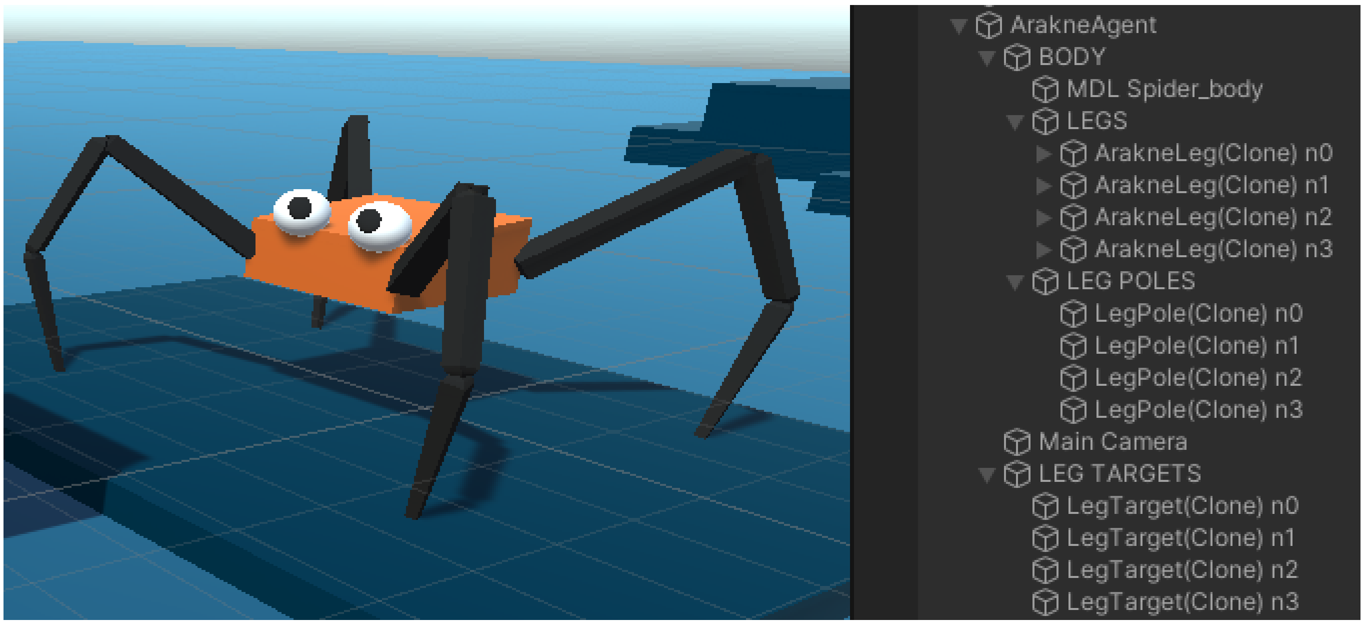
Procedural asset generation is used in video games to generate game assets, such as textures, models, and levels. It is used to reduce production time and space on disk, as well as to provide unique and complex assets without the need of manual creation.

In this project, we will see all the steps to implementat in Unity a spider-like agent that is able to navigate through a 3D environment using procedural animation techniques. The agent's movement is based on the positioning of its legs in response to the terrain beneath it. To achieve this, we have combined the use of Inverse Kinematics (IK) and procedural generation techniques to create believable and adaptive movement. The agent's ability to move on uneven terrain was made possible by the implementation of a procedural ground that changes dynamically based on noise algorithms. In this report, we will provide an overview of the technical challenges faced during the development of the agent, and we will detail the methods used to overcome these challenges. The goal of this project is to demonstrate how procedural techniques can be used to create believable and adaptive movement in 3D environments, and to showcase the potential of IK and procedural generation in game development.

Project Development

Introducing our agent

The spider-like agent for this Unity project will be portrayed by a gameobject named ***ArakneAgent*** and having the following structure:



Its main child gameobjects are:

- *BODY* : the parent Transform for most of the agent body parts, like:
    - the model of the body
    - *LEGS* : it is the container for the (custom amount of) instances of *ArakneLeg* prefab, created during the execution
    - *LEG POLES* : it is generated during execution and contains the pole vectors for the legs IK system
- *Main Camera* : the game camera, which will follow the agent while moving
- *LEG TARGETS* : it is generated during execution and contains the legTarget objects that each legHandle follows when taking a step

Two AI scripts are attached to the *ArakneAgent* gameobject:

1. **`ArakneAI`** : the main AI component of our agent. It handles the procedural initialization of the model, as well as the run-time execution of the procedural animations and the obstacle avoidance for the body. From the inspector, the user can set many parameters, like:

    - `bodyWidth` and `bodyLength`, to set the body size of the agent
    - `pairsOfLegs`, to decide the number of legs of the agent (can be changed at run-time)
    - `handleDistance`: the distance of the leg tips from the body
    - `poleDelta`: the offset for the legs pole vectors
    - `bodyDefaultHeight` and `bodyCurrHeight`, to raise the body from the ground

2. **`MoveAgent`** : the movement behavior that make our agent rotate and move towards the position of a given target position. From the inspector, the user can set the variables:

    - `moveSpeed`: the maximum speed at which the agent should move towards its target

  - `target`: the Transform that the agent should move towards
  - `slowDistance`: the distance at which the agent should start slowing down towards its target
  - `stopDistance`: the distance at which the agent should stop moving towards its target

Each leg prefab *ArakneLeg* has a hierarchical structure from root bone to leaf bone. The leaf bone (named "*Leg_[chainLength-1]_end*") of each leg has a **`FabrIK`** script that handles the position and rotation of the whole limb via an Inverse Kinematic solver.

Introduction to FK and IK

Forward Kinematics (FK) is the process of calculating the final position and orientation of an object or character's end effector, such as a hand or a foot, based on the position and orientation of its parent joints.

Inverse Kinematics (IK), on the other hand, takes the final position of the end effector and determins the position and orientation of its parent objects.

Fabrik algorithm

The FABRIK (*Forward And Backwards Reaching Inverse Kinematics*) algorithm is a popular method for solving IK in computer graphics and robotics. It uses an iterative algorithm that can handle chains of any length.

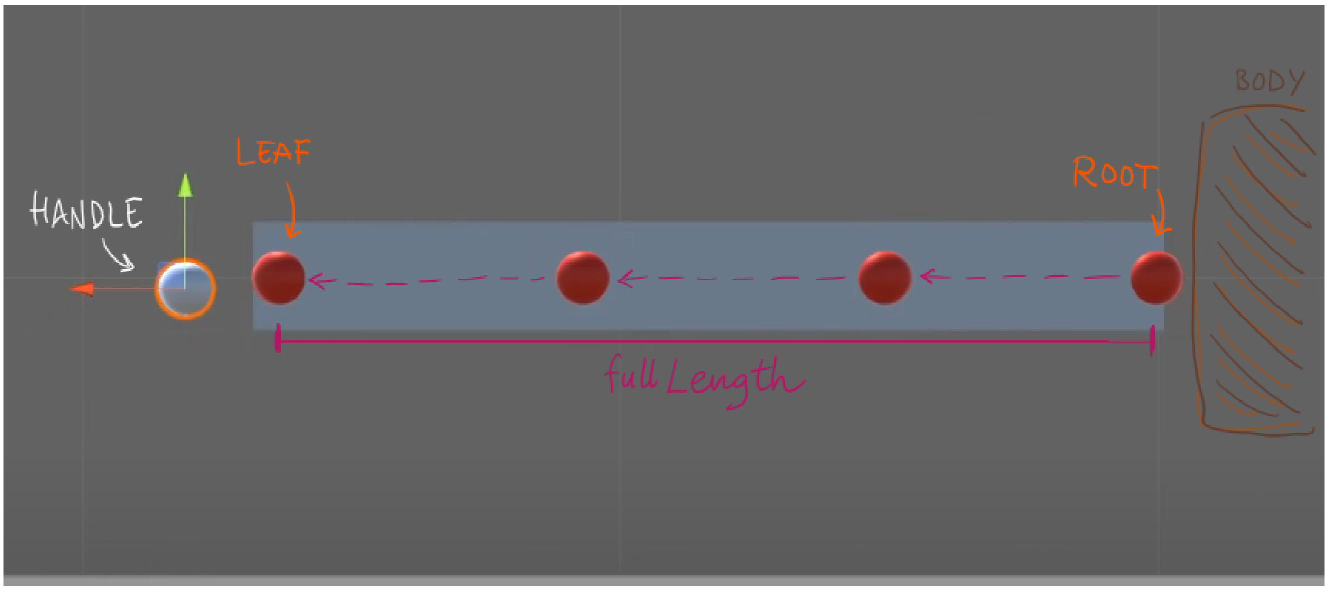
The basic idea behind FABRIK is to move the end effector of the kinematic chain towards the target position, while maintaining the constraints of the joints. The algorithm starts by moving the end effector towards the target position, and then iteratively moving each joint towards the position of its parent joint, while maintaining the constraints of the joints. This process is repeated until the end effector reaches the target position.

The main advantage of the this algorithm is that it is able to handle chains of any length, and can find solutions for both simple and complex kinematic chains. Additionally, it can handle multiple constraints and can work well with high degrees of freedom. In video game development, it is commonly used for animation, physics-based simulations, and character control.

The FABRIK algorithm has some limitations, for example it can have problems when dealing with very long chains and the solution may not always be the most optimal one.

FabrIK implementation

Let's abstract a limb as a chain made up of a customizable number of joints (we call this number *`chainLength`*) connecting a serie of *chainLength+1* nodes together, arranged in a straight line and parented one to the other from root to leaf.



Schematic representation of a IK chain having chainLength = 3

We use an array to store the lengths of each joint, that is the distances between one node and the next in the chain. The sum of all these distances will give us the overall length of the limb; we call it *`fullLength`*.

Then let's add a separate Transform called *`handle`* to act as the target for the tip of our limb. So now, basically, each time the *handle* is moved we can have two possible situations:

- A. the distance between handle and root is equal to or greater than the complete length of the limb;
- B. the distance between handle and root is less than the complete length of the limb.

The first one is the easiest case: we just need to start from the root and align each bone along the straight line from root to handle.

In the second case we would like the leaf bone to copy the taget location and the rest of the limb bend accordingly. So the leaf is put in the same position of the handle, then it tries to adjust the position of its parent accordingly, which tries to adjust its parent position consequently, and so on up to the root bone with a backwards cascading effect. Then, the same process is applied forward from root to leaf (the root is skipped because we want to keep it in place). This produces an approximation of the ideal position for each bone in the chain. The process can be repeated a desired number of times (*iterations*) to improve the approximation; also we can stop the computation as soon as the leaf bone and the handle are closer than a desired *delta*.

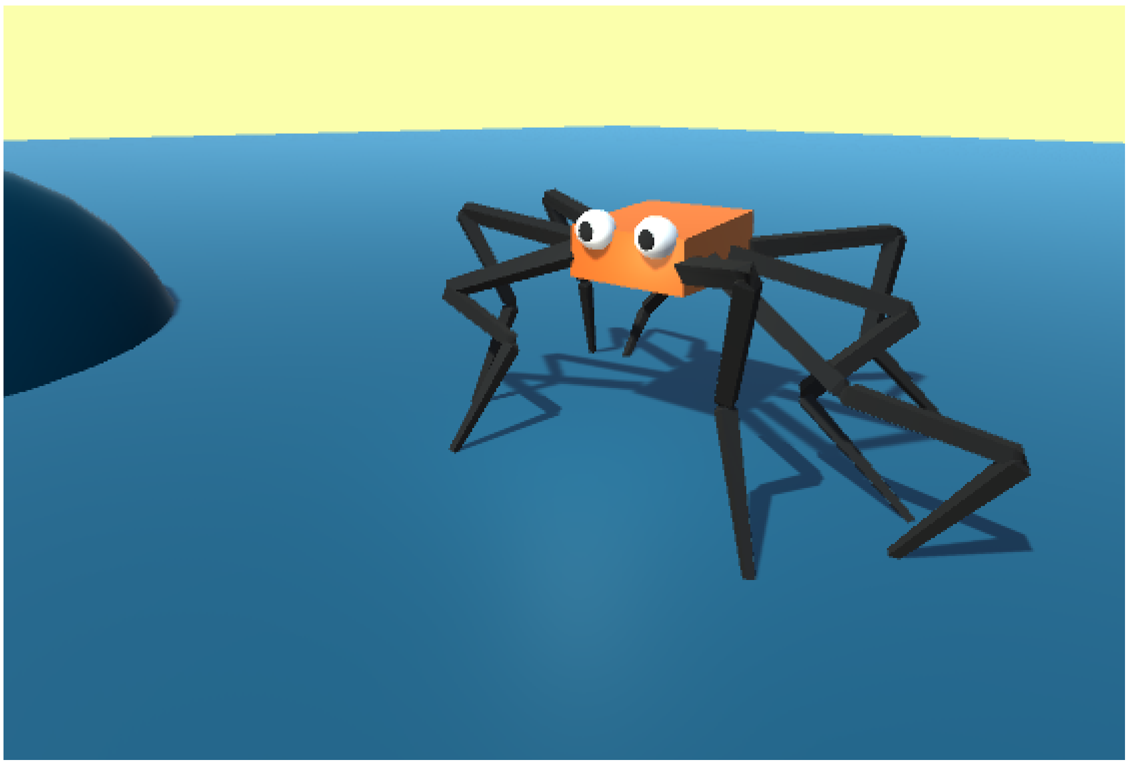
The following is an abstraction of the implemented algorithm inside our **`FabrIK`** Script:

- For each iteration, we perform the main steps:
  1. Backward step:
     - we set the position of the last bone in the chain (the "leaf" bone) to the target position;
     - Then, for each bone in the chain (except for the root bone), we place it along the line connecting it to its child bone, using the bone's length to determine the exact position;
  2. Forward step:
     - For each bone in the chain (starting from the second bone), we set the bone's position to be along the line connecting it to its parent bone, using the parent bone's length to determine the exact position;

3. Check: after each iteration, we check if the distance between the target and the leaf bone is smaller than *delta*. If so, it breaks out of the loop, as the bone positions are considered close enough to the target.

Needing a Pole vector

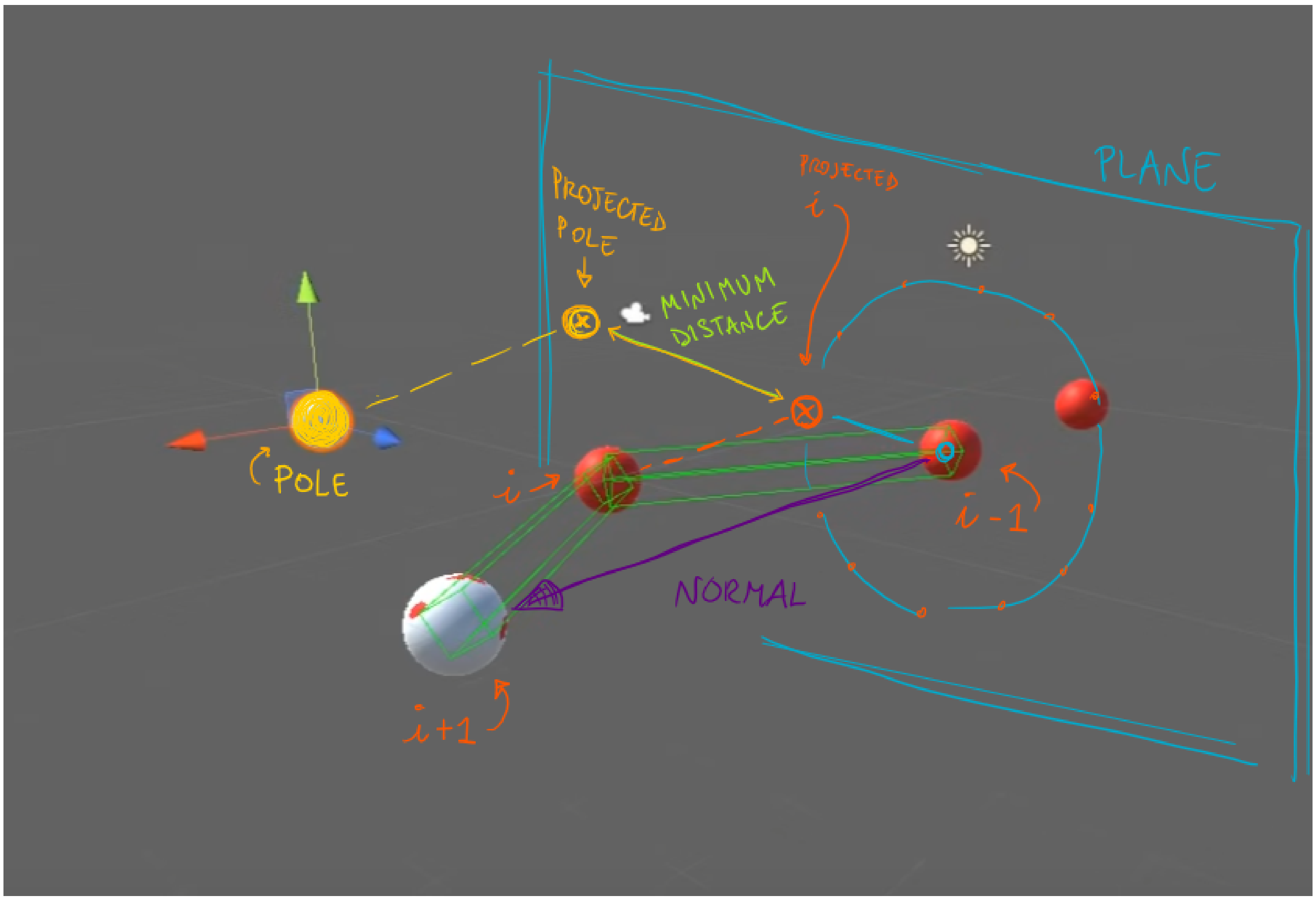
In some cases the obtained solution could appear unnatural, resulting in a "*broken leg*" effect.



Not a great leg positioning, unless we want a mosquito agent

In order to address this issue, a further geometric computation is added: we'll take into account also another point in the space, that is the position of a given `pole` Transform. So the algorithm iterates through the bones of the limb, excluding the root and leaf bones, and proceeds as follows:

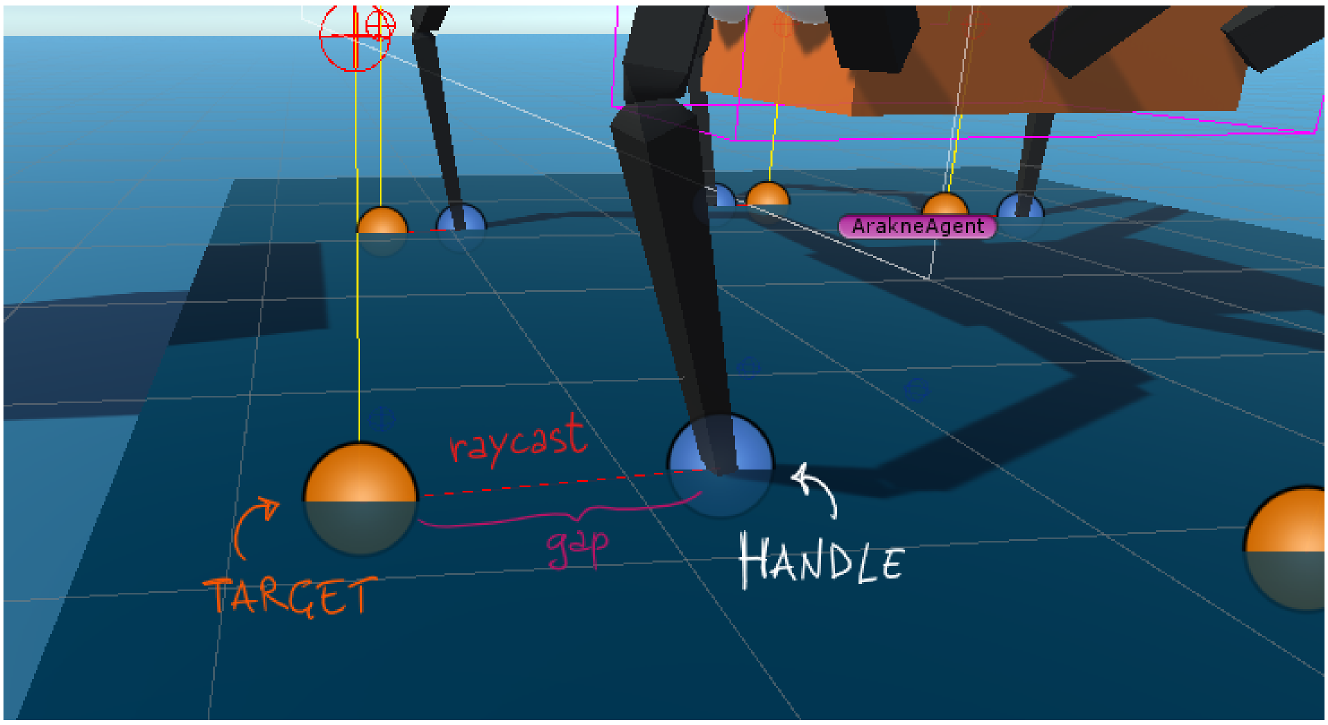
1. For each internal bone, it creates a plane having as normal the vector from the parent bone to the child, and passing through the position of the parent bone.
2. It then projects the position of the pole and the current bone onto this plane, obtaining the projected pole and projected bone positions.
3. It calculates the angle between the line connecting the projected bone position to its parent and the line connecting the projected pole position to its parent.
4. It rotates the current bone position around the normal of the plane by this angle and updates the current bone position to the new position.



Visualization of pole and bones and their projections onto the plane

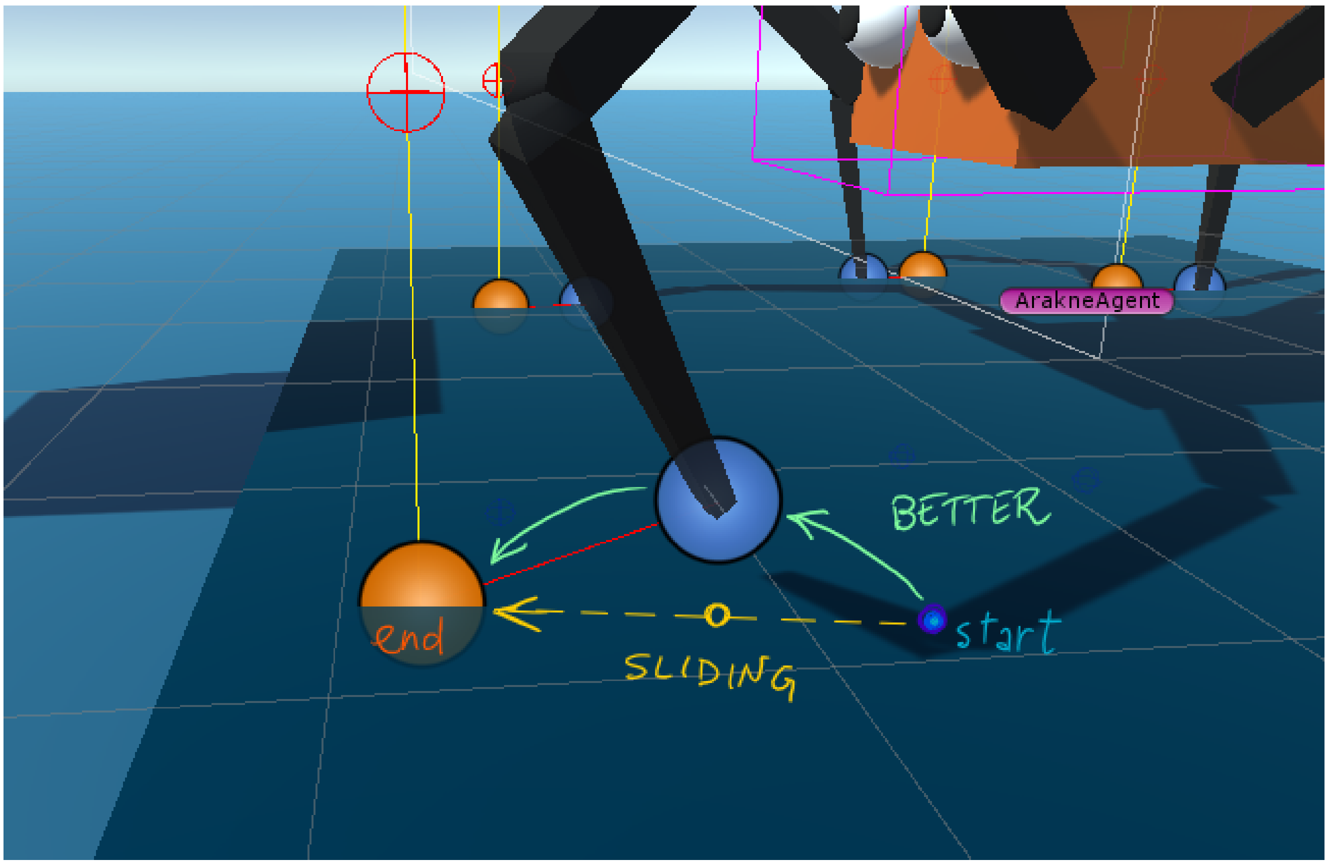
Taking a step

In order to allow our IK leg to take a step, we associate a *`LegTarget`* Transform to the leg handle of each leg. This target is parented to the agent, so it will move forward along with the body, while the handle remains stationary in position. Only when the distance between the handle and the target exceeds the `stepGap` threshold, then the relative leg enters the `hasToMoveLeg` state: the handle will then be moved and will tend to reach the target, resulting in the animation of a step.



Raycast from handle to target

For the step animation, a first solution could be to linearly interpolate in time between the position of the handle and the target (given a `legSpeed`). The result is acceptable but the effect is that of a leg that slides along the ground, without lifting off. For a more realistic step, we would like the leg to lift off the ground, tracing some curve, and then touch the ground only once it reaches the target position. Therefore, for the first half of the path, we direct the handle forward towards the target but adding an upward component; only after the half of the path, once the leg tip will be in the air, we will interpolate in time its position with the actual position of the target.



Raising up the handle instead of just sliding towards the target

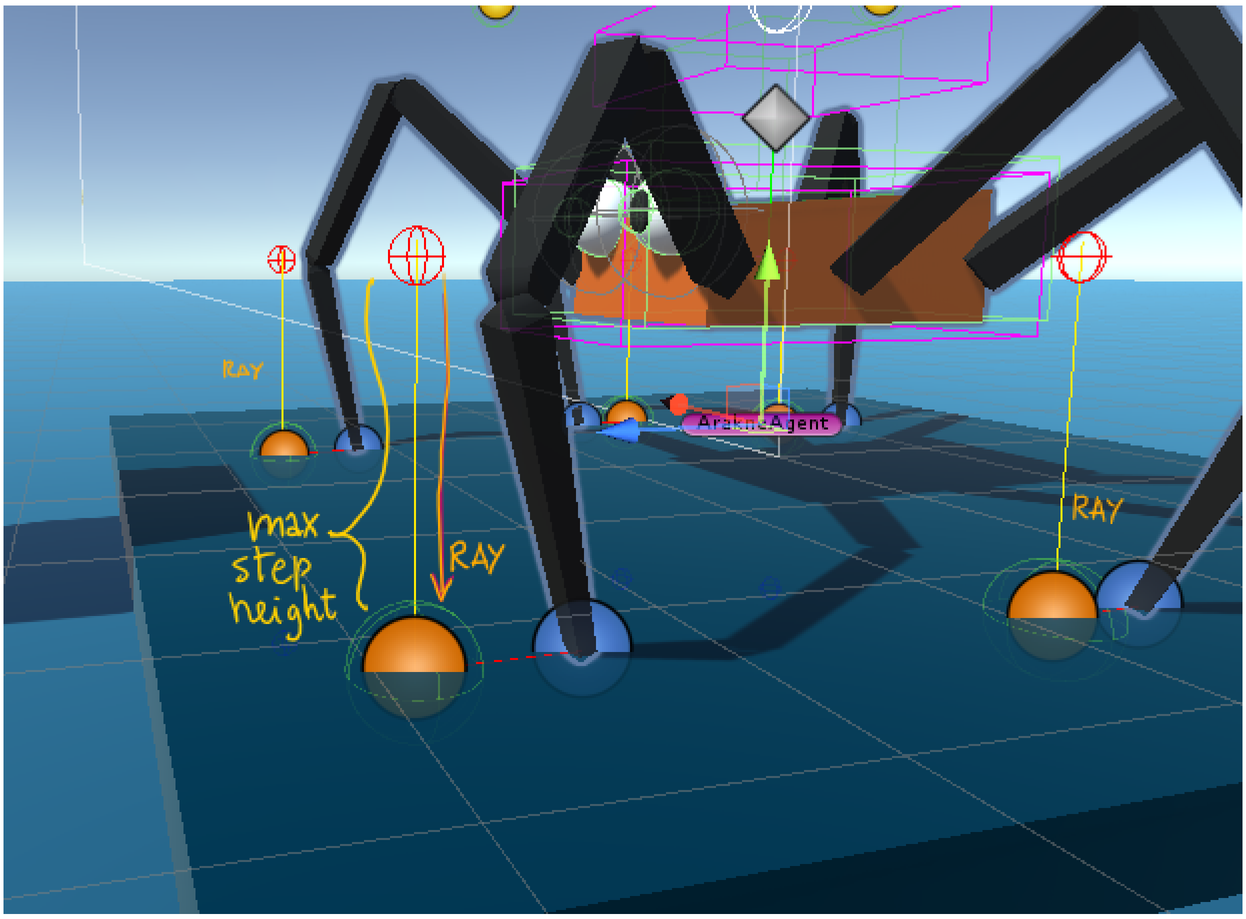
It should be noted that, in the event that too much distance has accumulated between the handle and the target, the step animation is accelerated; in extreme cases, the handle is instantly teleported into final position, to preserve the organicity of the model rather than the realism of the step.

Make legs follow the surface

We want to ensure that, during execution, each leg touches the ground correctly, following the shape of the terrain. So, we define a Unity layer called "Ground": we will assign it to every gameobject on which we want our agent to be able to stand on. Then, in our **AraknеAI** Script we add a `whatIsGround` LayerMask, which we will set to "*Ground*" from the Inspector.

At run-time, our approach will be to perform a raycast every frame starting from each legTarget position and pointing downwards, looking for some Collider belonging to the "*Ground*" layer. Then we place the legTarget at the collision point. So we ensure that every legTarget is grounded and it will be a valid next position for its relative legHandle. The only issue is that is that we cannot give for granted that such a raycast always hits the ground, since it may occur that the legTarget is below the terrain. The downwards raycast does not work in this case. We can't even cast a ray coming from the sky, since it may encounter floating obstacles above the ground.

In order to do so, we need to slightly offset the origin of the raycast along the vertical axis by a custom value, which we can tune according to our level design. Therefore, we add a variable called *maxStepHeight* to our script for this purpose; we will use it to regulate the maximum altitude difference that our agent will be able to handle.



Rays casted from above the targets towards the ground

Now, let's add an extra check since we want to ensure that not only the position at the next step is correct, but also the current position of each legHandle. This is because the ground that our agent is standing on may change in real-time under its feet. So, inside `CheckLegHandle()`, we can use the same raycast approach to find the closest point on the ground for each legHandle and put it in the correct position. Be careful to do it only if the relative *hasToMoveLeg* is false, beacause we do not want to stick the leg on the ground while performing the step animation.

Result: our happy agent going on a hike

Generate a spider with variable legs

In order to generate the desired number of limbs, our ArakneAI handles the instantiation (and the destruction) on demand of the leg prefabs and all the others gameobjects associated, like legHandles, legPoles and legTargets.

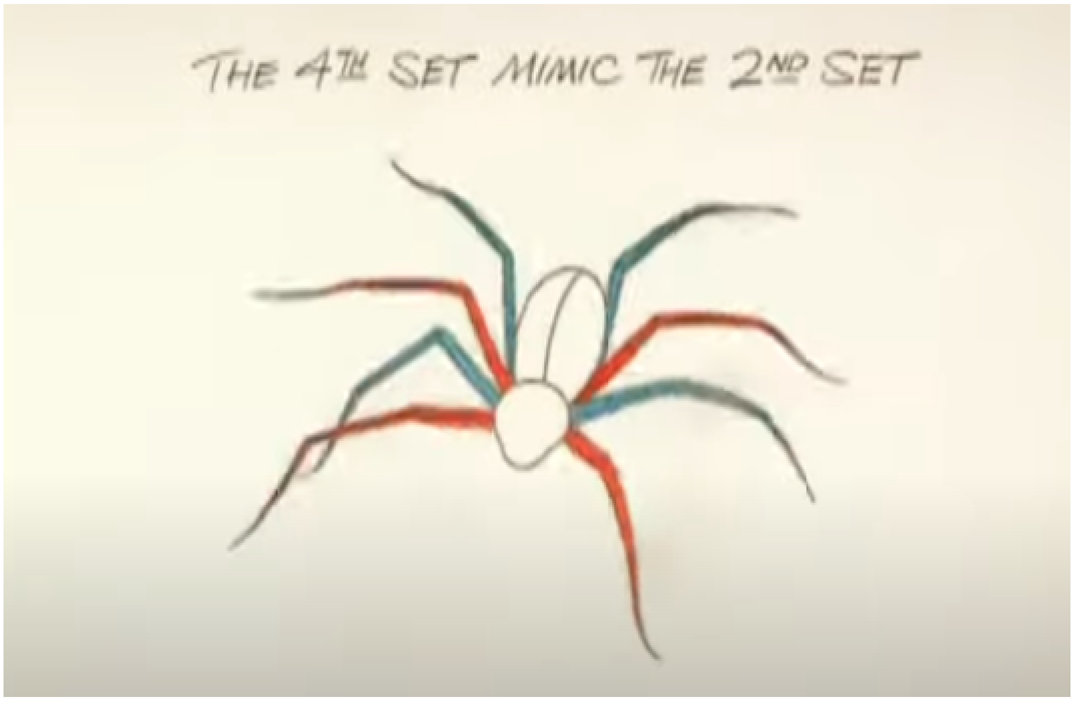
When the value of *pairsOfLegs* is changed by the user, the method `InitLegs()` is called. So the algorithm destroys all pre-existing gameobjects and resets the arrays that refer to legs. Then, new leg prefabs are instantiated.

To find the correct position for the leg roots and place them evenly spaced along the body length, we compute the legGap. It is proportional to bodyLength/(pairsOfLegs-1). To place the leg tips in a circular shape, we compute a direction for each leg. This direction is obtained normalizing the vector with origin in the body center and pointing to the leg root position. So we use this direction scaled by *handleDistance* to position the legHandle; then we add the *poleDelta* vector to position the legPole.

Make the agent take a step in a believable way

At this point, it might already be enough to apply a translation to our agent, and we would see that, at some point, when the distance between the handles and the targets exceeds the stepGap value, each leg performs the step animation. However, the problem now is that all legs move simultaneously.

Instead, we would like a more realistic behavior. Legs should move in a cyclic sequence, and no leg should step if the opposite is already performing a step.



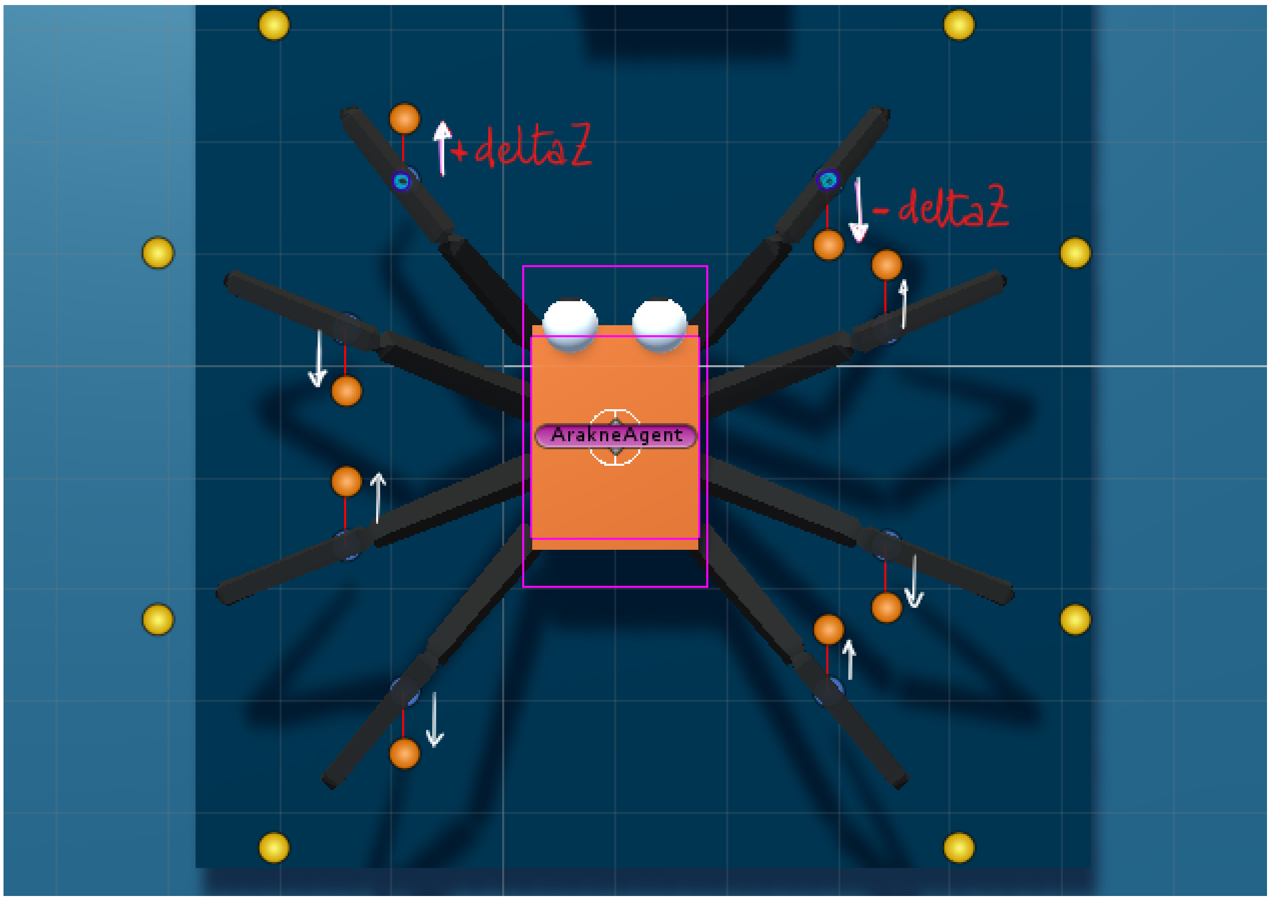
let's take this walking spider animation by Richard Williams as a reference

With that said, we can extend our code with the following improvements:

1. Leg targets are poisitioned in an organic zigzag pattern
2. Each leg is only allowed to take a step only if the opposite leg in the pair is grounded
3. Each leg is only allowed to take a step if the leg in front of it is on the ground.

Organic zigzag target displacement

In the `InitLegs()`, after our algorithm has instatiated and set up each leg, we add a last step so it loops through all of the pairs of legs in the character. For each pair, it moves one *legTarget* forward by a *deltaZ* displacement and the opposite *legTarget* backwards by the same amount. Then, for the next pair of legs, it flips the direction of *deltaZ* so that the next pair of legTargets will be mirrored. The consequence is that some legHandles will be ahead of their legTargets, while the others will be behind. This means that, when the agent will be moving, the step animation will be activated alternately for each pair of legs.



Alternating target displacement

We can add rules even while the program is running to ensure a more believable behavior even while moving.

Step only if opposite is grounded

At run-time, the `CheckLegHandle()` method also uses the *hasToMoveLegs* array of booleans to check the status of the opposite leg in the pair. If the opposite one is moving, the current one cannot. We add this rule so that two legs in the same pair are not in the air at the same time.

Step only if next is grounded

Let's add a further check in our `CheckLegHandle()` method so that two legs in a row cannot be in the air at the same time. So, for each leg (excluding the front legs), we use *hasToMoveLegs* array to check the status of the leg in front; if that one is moving, the current one cannot.

Adjust body height depending on legs average position

We want our agent to try to keep a fixed distance between the ground and its body (*bodyDefaultHeight*). This is possible in LateUpdate(). After updating each leg position with CheckLegHandle(), we evaluate the averageLegsHeight of all the legs by summing up every `legHandle[i].position.y` and then dividing the result by the number of legs. In this way we can use `averageLegsHeight + bodyDefaultHeight` as a candidate next position for our body.

Adjust body height if an upcoming collision is detected**

Before updating the final body position, we want to check if a collision is going to occur, so we can move the height of the agent's body accordingly.

The obstacle avoidance algorithm uses the Unity *Physics.CheckBox* function to check if there is an obstacle in the candidate next position of the spider's body. If there is no obstacle, the algorithm updates the last valid body position and returns the current body height. If an obstacle is detected, the algorithm enters a loop that tries to find free space for the body to move by checking for space both above and below the current position.

The algorithm starts by trying to move the body below the current position and checks for free space in increments of 0.15 units along the vertical. If it finds a free space, it updates the last valid body position and returns the new body height.

If no free space is found below the current position, the algorithm then tries to move the body above the current position and checks for free space in increments of 0.15 units along the y-axis. If it finds a free space, it updates the last valid body position and returns the new body height.

If no free space is found above or below the current position, the algorithm sets a flag that indicates that the spider is blocked and unable to avoid the obstacle, and returns the current body height. This flag can be used to prevent the spider from moving further and to trigger other actions such as playing an animation or a sound effect.

Make the agent move

So, we wanted to check out how our agent behaves when we let it roam around in the environment. Since it's mainly for testing, an easy approach is to use a kinematic algorithm like "arriving", which combines the simple "seek" behaviours.

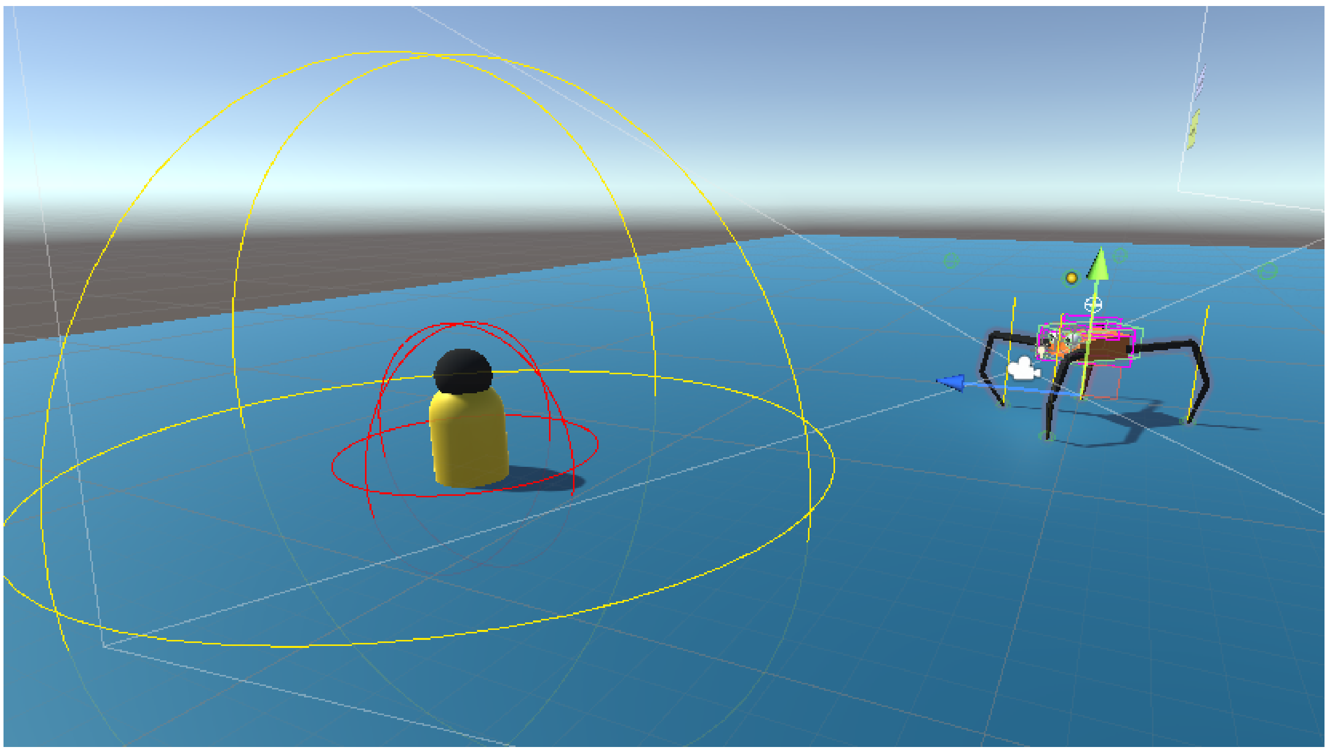
The Seek algorithm is used to make an agent move towards a target position. It computes the desired velocity by subtracting the agent current position from the target position and normalizing the result. This vector is then multiplied by the agent's maximum speed to obtain the final velocity. The agent's position is then updated by adding the velocity to it, so the result is that the agent will move in a straight line towards the target.

The Arriving algorithm is similar to Seek, but it includes the concept of slowing down as the agent approaches the target. It calculates the distance between the agent and the target, and if this distance is less than a certain threshold, the agent's speed is scaled down proportionally to the distance. This creates a smooth slowing down effect as the agent approaches the target.

Our custom script **MoveAgent** has several public variables that can be set in the Unity editor:

- *moveSpeed*: the maximum speed at which the agent should move towards its target
- *target*: the Transform that the agent should move towards
- *slowDistance*: the distance at which the agent should start slowing down towards its target
- *stopDistance*: the distance at which the agent should stop moving towards its target

The script also has a private variable, *currentSpeed*, which stores the current speed of the agent. Additionally, the boolean *isBlocked* is a flag controlled by ArakneAI and it is used to prevent the spider from moving when facing an insurmountable obstacle.



Our agent moving towards its target destination

Gizmos were used to make it easier to control the distances of the slowDownCircle (yellow) and the stopCircle (red).

The implementation inside MoveAgent works in the following way. At first, after checking if a target is assigned, it calculates the distance from the target in the xz plane. If the square of the distance is less than the square of stopDistance, the currentSpeed is set to 0. If the square of the distance is less than the square of slowDistance, the currentSpeed is linearly interpolated between 0 and moveSpeed based on the ratio of the square of the current distance to the square of slowDistance. If the square of the distance is greater than the square of slowDistance, the currentSpeed is just set to moveSpeed.

Then the agent's Transform is rotated with the `transform.LookAt()` method so it looks at the target's position. Finally, the agent is moved with `transform.MoveTowards()`, that translates it in the forward direction based on the currentSpeed.

Creating enviroments where letting the spider move

Environment 1: a Playground

A flat terrain consisting of a plane, to which 3d models of various shapes have been added; they are positioned in such a way as to serve as elevated spots or as obstacles (on the ground or suspended). Example: ladder.

The ground plane has a PlaneCollider component in order to be detected by raycasting, as well as any other model has a Collider matching to its mesh. These elements also are put in a specific physics Layer: "Ground". It is used by the agent AI during the raycasting operations to distinguish ground objects from body parts.

Environment 2: a Procedural, noise-based ground

For the second type of environment we are going to use noised-based terrain generation, that is a technique used in game development and other fields to procedurally create realistic and varied landscapes. The process makes use of mathematical noise functions - such as Perlin or Simplex noise - to generate a heightmap, which is then used to shape the terrain. This method allows for the creation of infinite, unique terrain with little manual input.

For the implementation, we use a custom approach. We are going to generate the flat ground mesh from scratch. Then, we will use an opensource script for generating optimized Simplex noise, that we use to perturbate the heights of our terrain mesh vertices.

The chosen approach is a re-working and adaptation of the implementation exposed by Sebastian Lague in his Procedural Planet Generation serie.

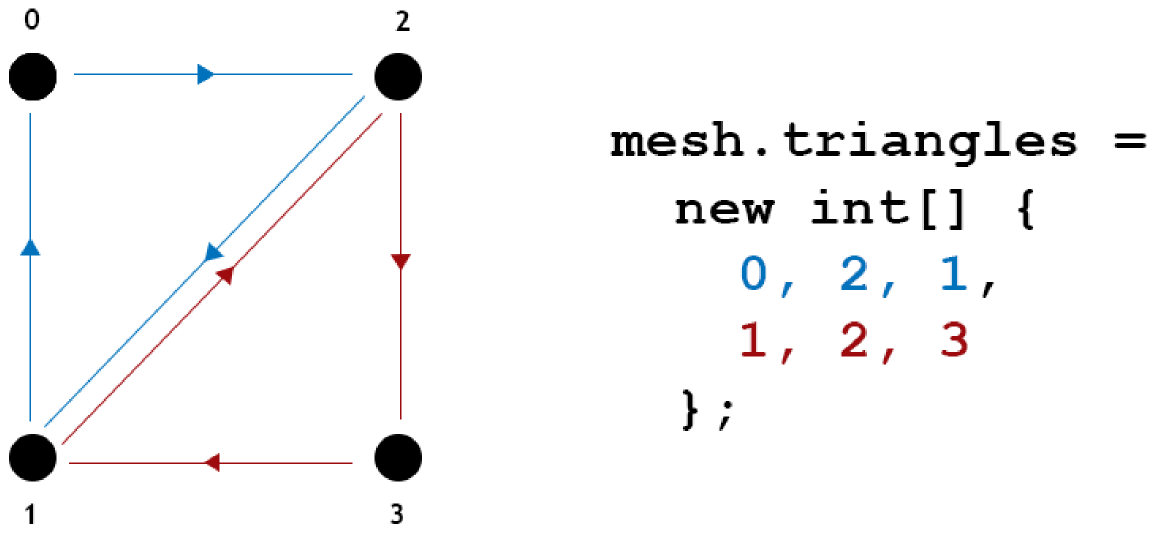
With the purpose to be modular and highly customizable from the Inspector, the resulting implemented system is quiet complex and I will try to unravel it in the following paragraphs.

Unity: mesh from script

Even if Unity offers a built-in Terrain gameobject, it is actually an annoying asset to work with. That is because of many factors; its default size is very big and it is not easy to adjust. Also, the dedicated libraries

So we take this as an opportunity to generate a mesh from code in Unity. When we create a `Mesh`, we have to provide at least two arrays:

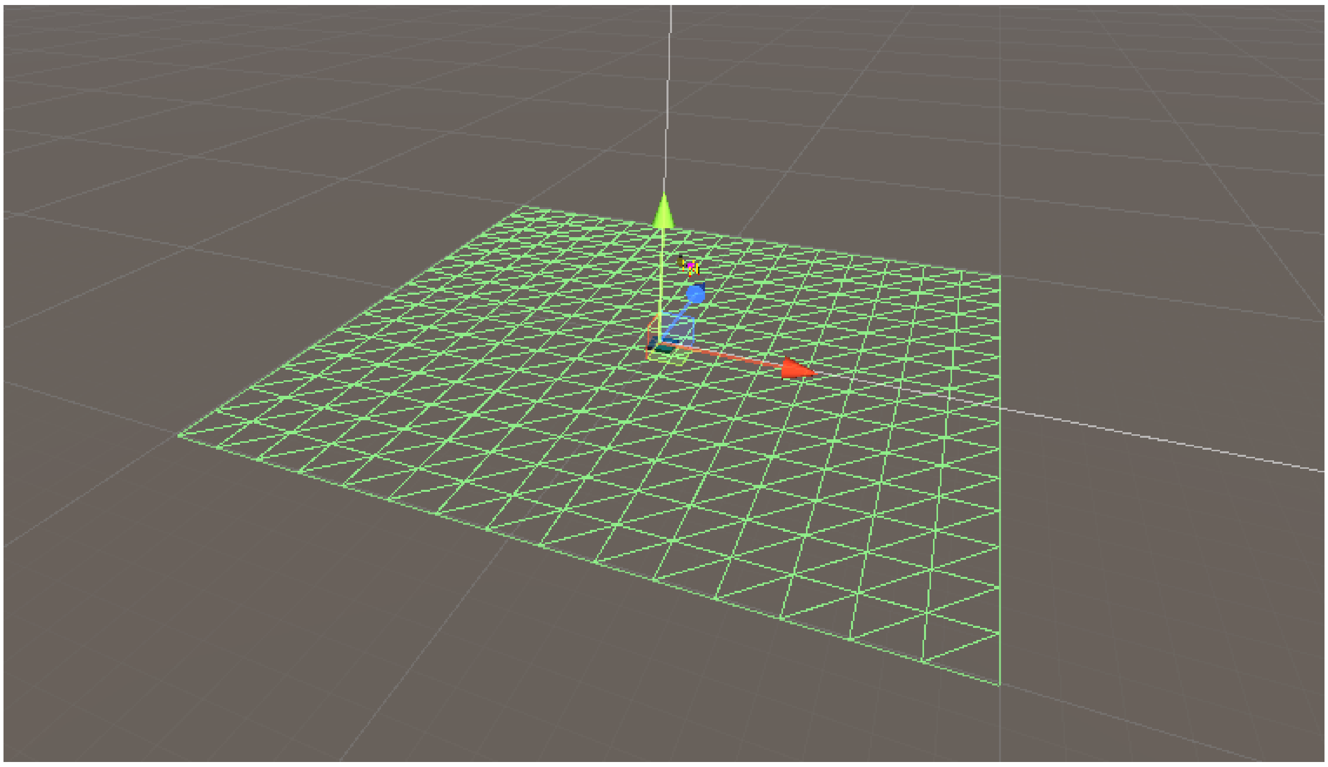
- one for the `vertices` which contains all the points (Vector3 objects) that make up the shape in space. Using these vertices, the individual triangles are defined, each of which represents a face of the mesh.
- and one for the `triangles`, that is made up of indices of the vertices array, which, grouped in threes, correspond to the faces of the mesh.



```
mesh.triangles =
  new int[] {
    0, 2, 1,
    1, 2, 3
  };
```

> N.B: The order of the vertices in these groups is important because it defines the direction of traversing the perimeter of each triangle and, consequently, the direction of the normal vector for that face.

In order to generate a custom flat lattice, we create the **PlaneMesh** class: it takes a desired `verticesDensity` and creates a 1x1 size mesh containing `verticesDensity*verticesDensity` evenly-spaced vertices. The generated square is centered in the origin, while the vertices array starts with the one in the bottom-left corner, having coordinates (-0.5, 0.0 ,-0.5).

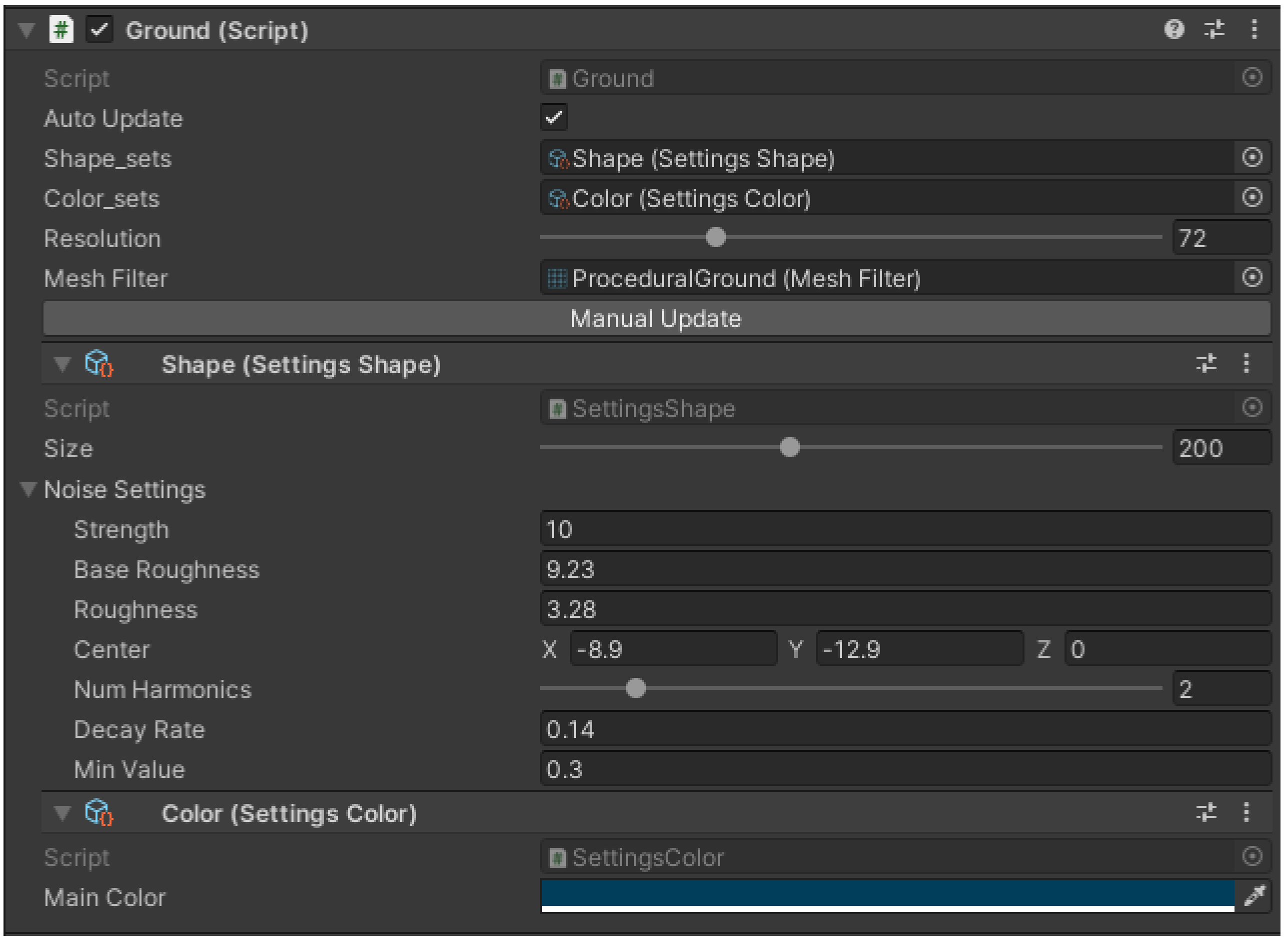


Wireframe of the generated flat lattice

PlaneMesh is used by Ground script, attached to the actual gameobject for the terrain, which is called 'ProcedutalGround'. Ground uses the *resolution* variable to control the *verticesDensity*. Ground also adds a MeshRender component (required by Unity for the rendering) and a MeshCollider (for physics collisions).

To create a terrain mesh which can scale in size, we have to put the *size* parameter in our computation. Also, to create a rough terrain instead of a flat one, we have to perturbate the height of the vertices in our planar mesh. In order to do so, PlaneMesh calls for every vertex the method `FromUnitQuadToShape()` of the **ShapeGenerator** class. This method scales the 1x1 point multiplying its position by *size*; then, it applies a vertical offset computing and adding the *height* value. The value for the height is obtained by passing the 1x1 point to the `Evaluate()` method of another class, that is **NoiseTuner**. Here is where the different parameters for the noise function are used:

- the **amplitude** is used to vertically scale the noise values
- the **frequency** (`baseRoughness`) tells how rough the noise is
- the **octaves** (`numHarmonics`) tells how many noise function calls sum up per point
- the **persistence** (`decayRate`) scales down the amplitude for every additional octave
- the **lacunarity** (`roughness`) scales down the frequency for every additional octave
- the `minValue` is used to clamp the obtained noised value to allow flat low regions
- the `strenght` is an additional scale factor for the final output to grant more control


Ground (Script)

Script	Ground
Auto Update	✓
Shape_sets	Shape (Settings Shape)
Color_sets	Color (Settings Color)
Resolution	72
Mesh Filter	ProceduralGround (Mesh Filter)

Manual Update

Shape (Settings Shape)

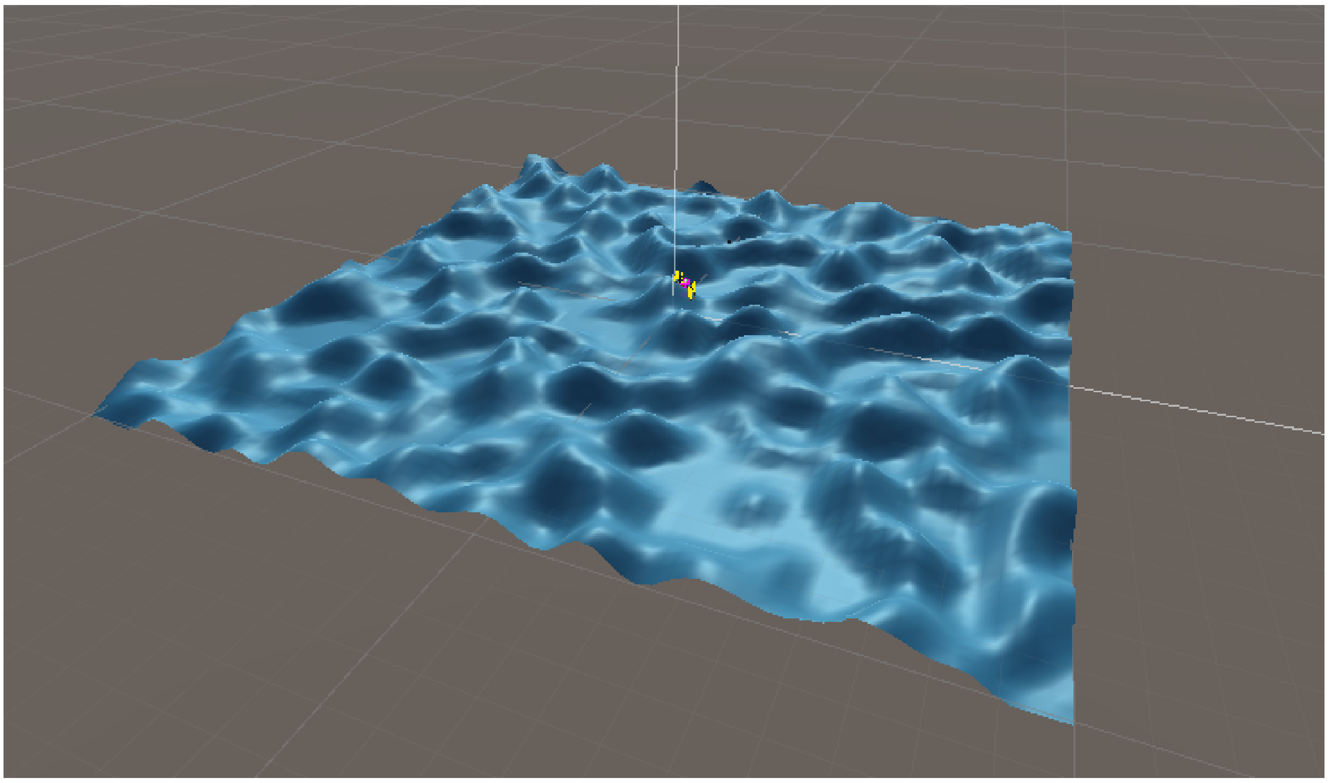
Script	SettingsShape
Size	200
Noise Settings	
Strength	10
Base Roughness	9.23
Roughness	3.28
Center	X -8.9 Y -12.9 Z 0
Num Harmonics	2
Decay Rate	0.14
Min Value	0.3

Color (Settings Color)

Script	SettingsColor
Main Color	🟦

The NoiseTuner gets the noise value for every octave from another method, which is `Noise.Evaluate()`. The **Noise.cs** script is part of libnoise-dotnet and it is ditributed under the GNU Lesser GPL. This is an opensource implementation of Simplex Noise and it is based on this paper by Stefan Gustavson. The Noise class has a method `Evaluate(Vector3 point)` that outputs a 3D Simplex Perlin noise.

It contains many precomputed factors that should increase the computation. So this script allows us to change the parameters of our procedural ground also in real-time, with a relative impact on the framerate.



Our final rough ground

Additional Scripts, like `NoiseSettings` and `GroundEditor`, as well as ScriptableObjects, like `SettingsShape` and `SettingsColor`, have been used just with the purpose to give more readability to the Inspector interface of our ProceduralGround.

Conclusions

In conclusion, the agent behavior presented in this project, although limited, could already be considered sufficient to be used in a video game. For example, it could be a believable robotic NPC that wanders within the game world. Of course, when dealing with an AI agent, some parameters may need to be manually set up on the specific needs, and some level design constraints must be considered.

On the other hand, if a more realistic agent is desired, then this model has some limitations, but is open to future expansion. One limitation is the obstacle avoidance system, which does not currently handle the edge case where the agent encounters an obstacle that is too high, causing it to stop. There are multiple solutions to this problem, depending on the need, such as creating and triggering a procedural jumping animation that allows the agent to overcome the obstacle; another alternative is to modify the movement behavior and make it a steering behavior to attempt to circumvent the obstacle on a horizontal plane, or expanding the movement system to allow the spider to move on vertical walls and climb the obstacle.

Depending on how realistic we want the agent to be, another problem could be that the collision of the legs is only handled on the root and on the tip. This means that, in some extreme cases, the model of the limb could penetrate a nearby obstacle. One possible solution is to expand our IK solving system so that it also uses raycasting from one bone to the next to check for the presence of any colliders; which in turn would require the ability to find an alternative solution for the IK chain. An alternative solution, faster to implement but less predictable, would be to use colliders for each segment of the leg (such as capsule colliders) and set appropriate constraints, leaving the final position of the bones to the physics engine.

External references

- Abstraction of procedural animation for a four-legged spider, by Codeer:
  - https://youtu.be/e6Gjhr1IP6w
- FastIK, by Daniel Erdmann:
  - video: https://youtu.be/qqOAzn05fvk
  - source code: https://github.com/ditzel/SimpleIK
- Procedural Planet Generation, by Sebastian Lague:
  - videos: https://youtube.com/playlist?list=PLFt_AvWsXl0cONs3T0By4puYy6GM22ko8
  - source code: https://github.com/SebLague/Procedural-Planets
- Simplex noise demystified, Stefan Gustavson:
  - http://staffwww.itn.liu.se/~stegu/simplexnoise/simplexnoise.pdf