Creating a Custom Deep Reinforcement Learning Environment

Design a custom RL environment using modern game development software and Deep Reinforcement Learning(A2C, PPO, TD3, ACER, DQN)

Aaron Krumins Nov 18, 2020 · 14 min read

In Wikimedia Commons. Retrieved November 17, 2020 wiki.unrealengine.com

https://www.youtube.com/watch?v=kNmNKnDQvgo

While the scope of reinforcement learning (RL) is likely to soon extend far beyond computer simulation, today the main location for training RL agents is within the digital environment. In the world of artificial intelligence, simulators are often the environments in which an algorithm functions.

For humans, we are born directly into our simulator and it requires no effort on our part to go on functioning. We call this simulator the universe and it exists whether we believe in it or not. Similarly, the laws of physics apply whether you acknowledge them or not. They require no effort or acquiescence on our part. A synthetic agent possessed of reinforcement

learning such as a robot or video game character, however, has no direct access to any world except that which we provide it through artificial sense organs. By itself, it exists as little more than a complex set of instructions for building a learning agent, not unlike a genetic code. It has no body or senses, no environment in which to act. Initially, at least, it is far easier for such agents to interact in virtual worlds within a computer than our own physical world since their instructions are comprised of software rather than organic compounds like DNA.

In a virtual world, the agent can easily gain access to all sorts of information about its simulator that is difficult to discover in our own physical world. If it wants to know about the physics of its virtual world, it can simply query the variables in which the programmer stored this information. In the physical world, there is no lookup table in which this information is stored. Instead, such properties must be ascertained the hard way, through learning. It is also far safer to have these reinforcement learners confined to virtual worlds during their exploration period since when they make a "mistake," it has limited repercussions, at worst causing the computer on which they run to crash(an exception are algorithms with access to the Internet, like some computer viruses or stock trading bots which have far greater capacity for real world damage) . Artificial agents can also learn much faster within virtual worlds since they are not limited by many of the environmental variables of our physical world. This is crucial when the reinforcement learning process requires thousands or millions of training iterations to get a sense of the desired associations for it to reach its strategic goal.

The simulator is therefore a pivotal additional element to the reinforcement learning equation. In this article we will explore how to bridge the gap between algorithm and simulator — creating a custom reinforcement learning environment in the process. Until recently OpenAI Gym was one of the best candidates for experimenting with RL agents; however this had its limitations. One can only watch an agent learn to play Atari Pong so many times before the crudeness of the pixelated environments begins to grate upon the nerves.

Today there are better options — with a free plugin called MindMaker, one can deploy the same cutting edge RL Algorithms that work in OpenAI Gym on the most graphically realistic simulation environment known to humankind. Yes, I'm talking about Unreal Engine 4 from the makers of Fortnight. Thanks to blueprint integration you can customize your DRL algorithm with an easy to use GUI rather than needing to dive into the

code. So without further preliminaries, lets get started creating a custom RL environment in Unreal Engine 4.

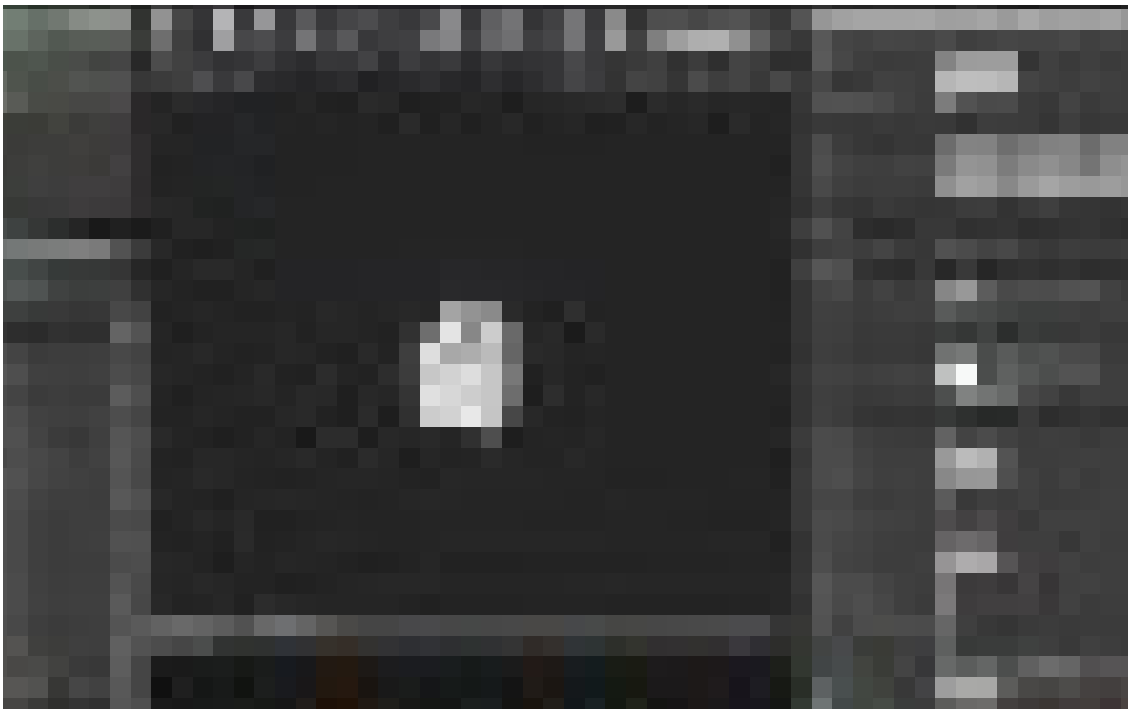
In this example we will go about recreating a classic RL training environment, the legendary cart pole task. The goal of the project is to teach a cart to balance a pole by moving side to side in the appropriate manner. It is similar to the balancing games played by children in which they learn to steady a broom on the tip of one's finger. The example has some interesting corollaries for such things as learning to use a steering wheel in the case of a self-driving car or teaching a robot to balance while standing upright. We will be creating this learning environment from scratch, demonstrating how easy it is to create a custom RL training task within UE4.

To begin download the the MindMaker Deep Reinforcement Learning package from the UE Marketplace. This contains the starter content that we will be using to create the cart pole environment (a finished version of the Cart-pole example is also included with the free MindMaker AI Plugin). Open the maps folder and locate the map called "MindMakerStarterContent". After opening this, navigate to the folder named "MindMakerStarterContent". This will look similar to the default 3rd person starter content provided by UE4, with the difference that several non-standard blueprints are included.

Before proceeding we need to make one change to the project settings, setting the frame rate to 30. This is to ensure that we get a smooth response from the physics engine and the cart doesn't get stuck hanging in mid air as it falls down. Go to the Edit drop down, and click project settings. Search for frame rate, its located under Engine General Settings, and change the Use Fixed Frame rate check box to true. Than set the frame rate to 30.

Now Rename BlankMindMakerActorBP blueprint, to "Cart_BP". Make sure to delete the other Mindmaker assets that won't be used before proceeding, specifically MindMakerCharacterControler and MindMakerCharacterBP which refer to the 3rd person character if you were building an RL agent based upon the 3rd person blueprints. Next open Cart_BP in the blueprints editor and navigate to the viewport tab. From the left side green button "add component menu" find 1M_Cube object and place one in the viewport window. Change its z scale to .2 and make sure the enable gravity checkbox in it's properties is set to true. Click the add component menu again and add a sphere over the cube so it sticks out the top. Change its z scale to .3. and the z location to 10. Hurrah, we now have our basic cart shape. Return

to the map and drag your Cart_BP blueprint onto the map so you see the combined shapes appearing there. Now change the x scale at the map level of your cart .3, this effect both the shapes that make up the cart.



Screen Capture by Author

Now we need a cylinder to balance on top of the cart. Create a new blueprint of class actor and name it Cyl_BP. Open it and from the left side green button “add component menu” click cylinder and drag it into the viewport. Change the scale of the cylinder so the X and Y dimensions measure .2 instead of 1. Got to the map level, select the cylinder and make sure that the simulate physics checkbox under properties is set to True as well as the enable gravity one. Change the linear dampening and angular dampening to 11. These are both under properties. Now go to the event graph and create two new float variables within the cylinder blueprint, called cylroll, cylpitch. Next create a variable of type Cart_BP, that will serve a reference to the cart since we will need to communicate with variables contained in Cart_BP. Return to the map, select your cylinder and from the Default menu you should now see the Cart_BP variable we just created. Click it and select the Cart_BP reference from under your map name. This is so that Cart_BP variable within the cylinder blueprint points to the Cart_BP blueprint. Return to the cylinder blueprint and create a bool variable called start which will function as an identifier for when the cylinder has tipped over and needs to be reset. Make the default value for Start set to true.

Now we will add some logic that detects when the cylinder has tipped over and resets it. This can be done by adding an event tick node that will

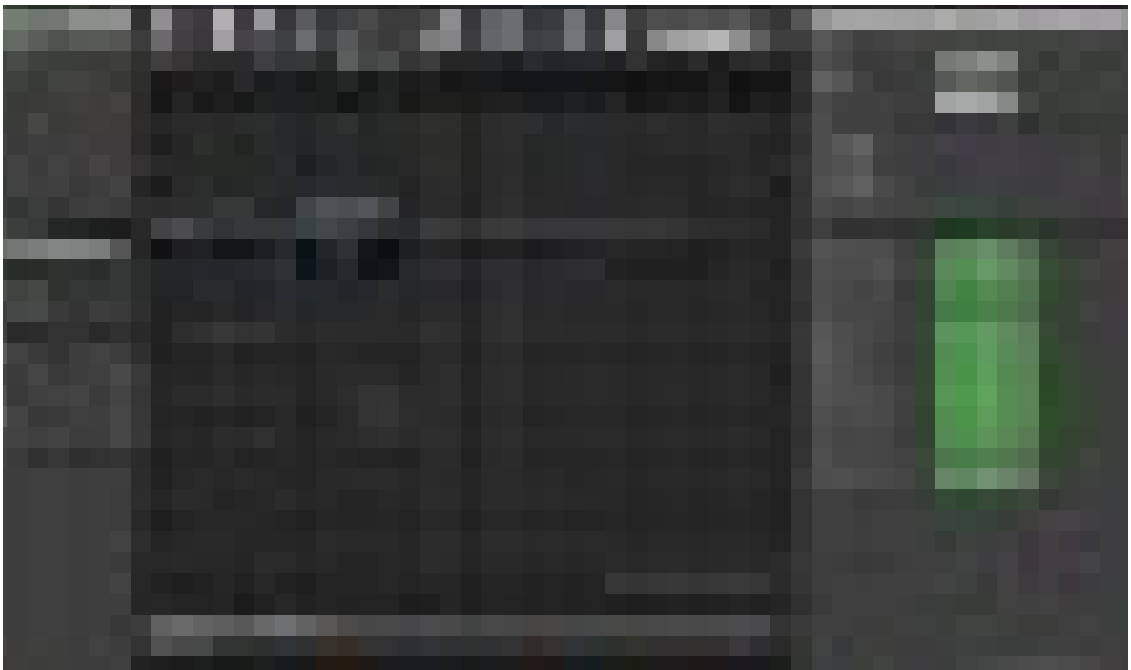
continually check to see if the cylinder has fallen. Connect the event tick to a branch node that will fire when any one of the variables cylroll, cylpitch, are inclined more than 30 degrees or less than 30 degrees. We will update these variables based on the physics of the cylinder at the map level blueprint. We also want the branch node to fire when the training period for the cart is complete, even if the cylinder has not fallen over yet, so that it can be reset when the number of training episodes is complete and we are ready to evaluate the AI. So we add a reference to the TrainingComplete variable contained in Cart_BP. TrainingComplete is one of the default MindMakerVariables that comes with the package. Whenever any of these conditions are True, we set the Start bool to false, essentially stopping training until the cart and pole are reset.

Screen Capture by Author

Screen Capture by Author

The next two nodes are for the logic of resetting the cart and pole to their original position using the set World location node and the move component node. The x, y and z coordinates for these will be somewhat particular to where exactly you placed the cart and pole on the map to begin with. You can find this info by clicking on them at the map and jotting down their x y and z coordinates. Last we need a node setting the Start bool to true, so that the MindMaker learning engine in Cart_BP will know that the cart has been reset and to begin training again. Next we set TrainingComplete variable to false, so that we don't continually reset the cart during any demonstration episodes that follow training.

Screen Capture by Author



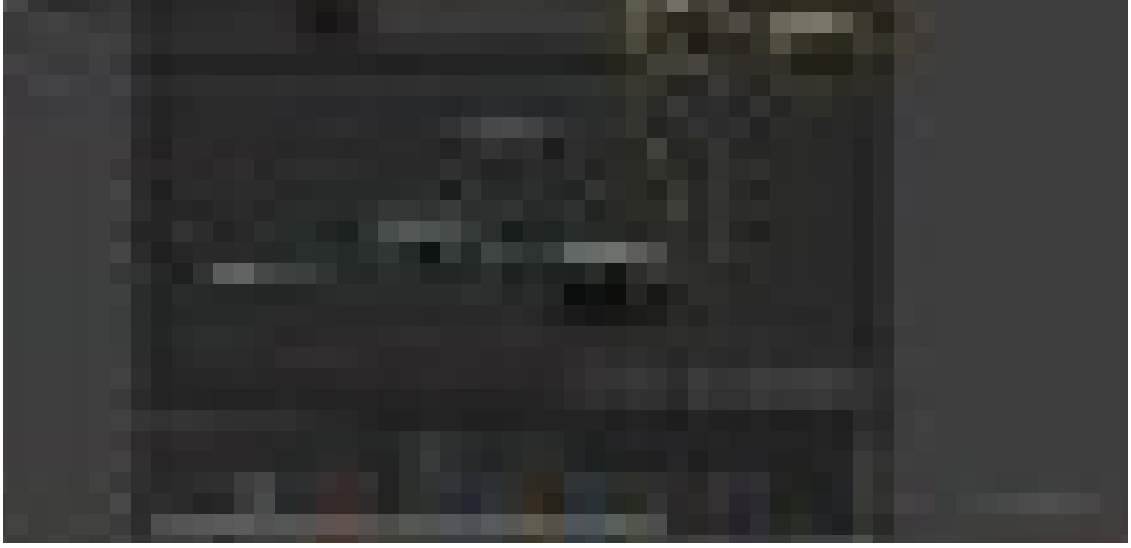
Screen Capture by Author

This completes the cylinder blueprint. Return to the map and drag in CYL_BP so that the cylinder appears roughly on top of the cart.

Screen Capture by Author

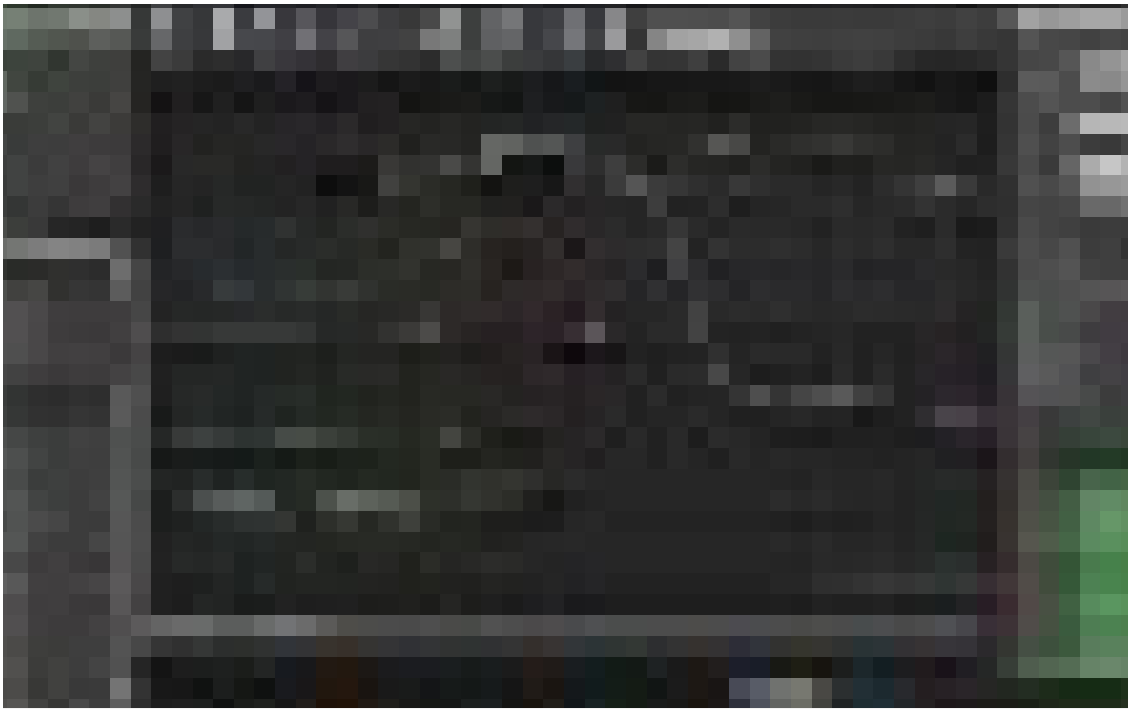
Getting updates on our cylinder physics can be a tricky and must be done at the map level. To do so FIRST CLICK THE CYLINDER ON THE MAP SO IT IS SELECTED AND OUTLINED IN YELLOW. Than open the map level blueprint. Now right click on the event graph and you will see an option “Create a Reference to Cyl_BP”. This is only there because you first selected the cylinder in the map window, and cannot otherwise be accessed.

Create two of such references. From the first one, drag off a GetActorRotation node. From its return value, find a “Break Actor Rotator” node. Now go to the other Cyl_BP reference and drag off a “Set Cylroll” node. This is getting a reference to CylRoll variable your created in Cyl_BP. Do the same for CylPitch as well. Now connect all these to their corresponding break actor rotator nodes. Now add an event tick node and connect the two Set variable nodes to it. With that done, your variables in Cyl_BP will be updated in real time with values about the roll and pitch of the cylinder.



Screen Capture by Author

Next we move to the Cart_BP that we first created, as this is where the machine learning will take place. It already will contain many default variables and functions that are used with MindMaker because it was copied from the BlankMindMakerActorBP.



Screen Capture by Author

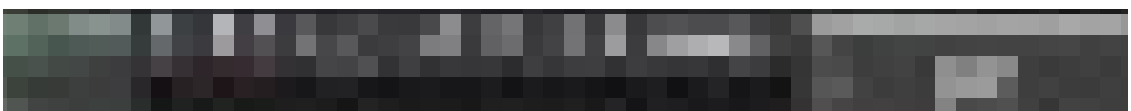
The first step in customizing Cart_BP will be to open it and click on the function on the left side called “CheckReward”. Rewards are the building blocks of Reinforcement Learning and we will need to tell the agent precisely what it is trying to optimize through this reward function. In this case we want it to be rewarded for each “episode” of the game in which it keeps the cylinder upright. To do this will use a branch statement, the equivalent of an “if than else”. We have several conditions to check so we will attach the branch node to an OR node. Next create a variable called cylinder of type Cyl_BP, which will be used to access our cylinder variables.

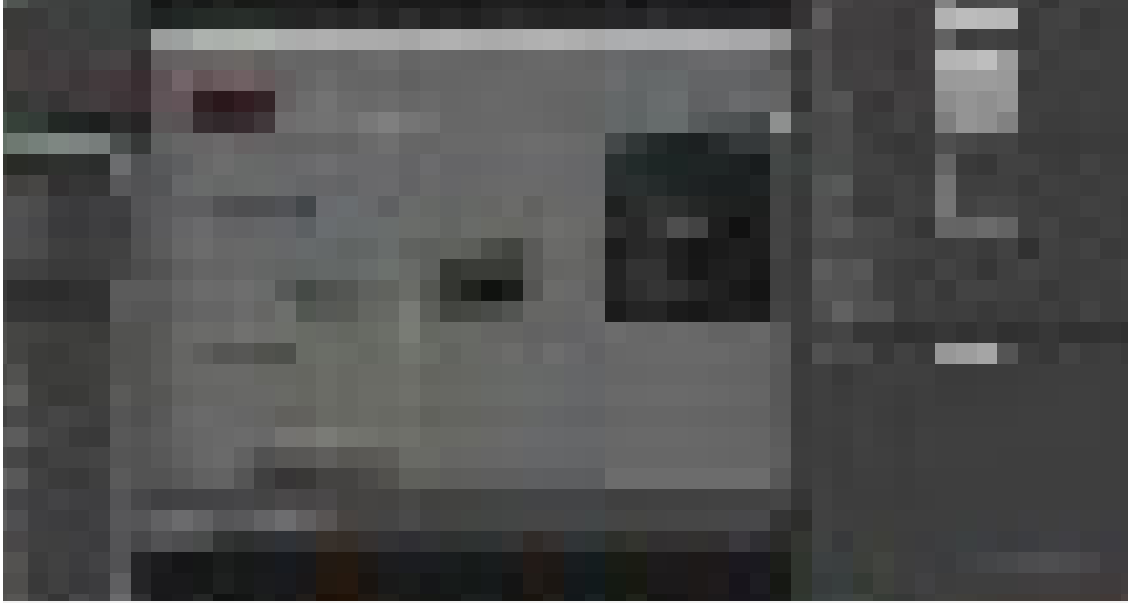
Go back to the map level blueprint, select the default tab, and you should see your Cyl_BP variable, click the dropdown and select the Cyl_BP object that should be listed under your map name. Again, this so that your Cylinder variable reflects values from Cyl_BP. Go back to the event graph for Cart_BP and from the OR node of your branch drag off conditionals for the pitch and roll of the cylinder, that is Cylpitch and Cylroll, and set them to greater than 30 and less than -30. Now from the True part of the branch, drag off a node setting the variable reward to itself plus one. This will increment the reward for every episode of the game in which the cylinder remains upright. If false, we set the reward variable to zero, resetting everything because the cylinder has fallen over. That's it for the reward function.

Next open the MakeObservations function in Cart_BP. We need to get a reference to the pitch of the cylinder, so that RL algorithm can learn to optimize around it. Similar to the reward function, grab the cylinder variable drag it in, and pull out the get cylpitch variable. Plug this in to the Append node of Set Observations.

We are almost done! Last we need to modify the display code, so that the environment gets updated with actions taken by the agent. Go to the main event graph and find the "Delay For Display" section of logic. Before the MakeObersvations function gets called, add a MoveComponentTo node and from the component reference on it, drag off a link to get "MindMakerStaticMesh". This is essentially the combined objects that make up the cart. From the same "MindMakerStaticMesh" node, drag off a "get relative location" node from the transform heading. Right click on the gold pin of that node and select "Split struct pin". Now we can alter the x y and z variables relating to our cart so that it moves depending on the action the RL algorithm has selected for it.

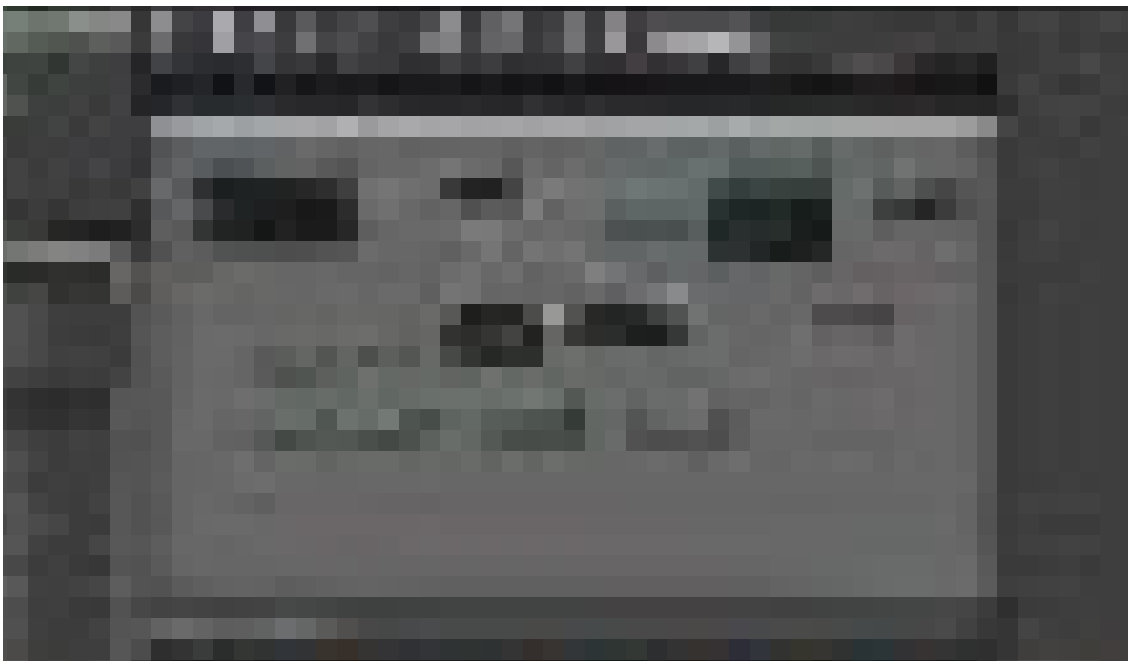
At this point in the blueprint, the action variable will already have been automatically updated in the "RecieveAction" function. MindMaker handles the generation of actions and they are received in UE via the ReceiveAction function so that afterwards, we move the agent according to the action selected. Now we just need to feed the Observations and Rewards from those actions back into the MindMaker RL algorithm until it generates better actions, in a kind of virtuous loop. This is the magic of reinforcement learning.





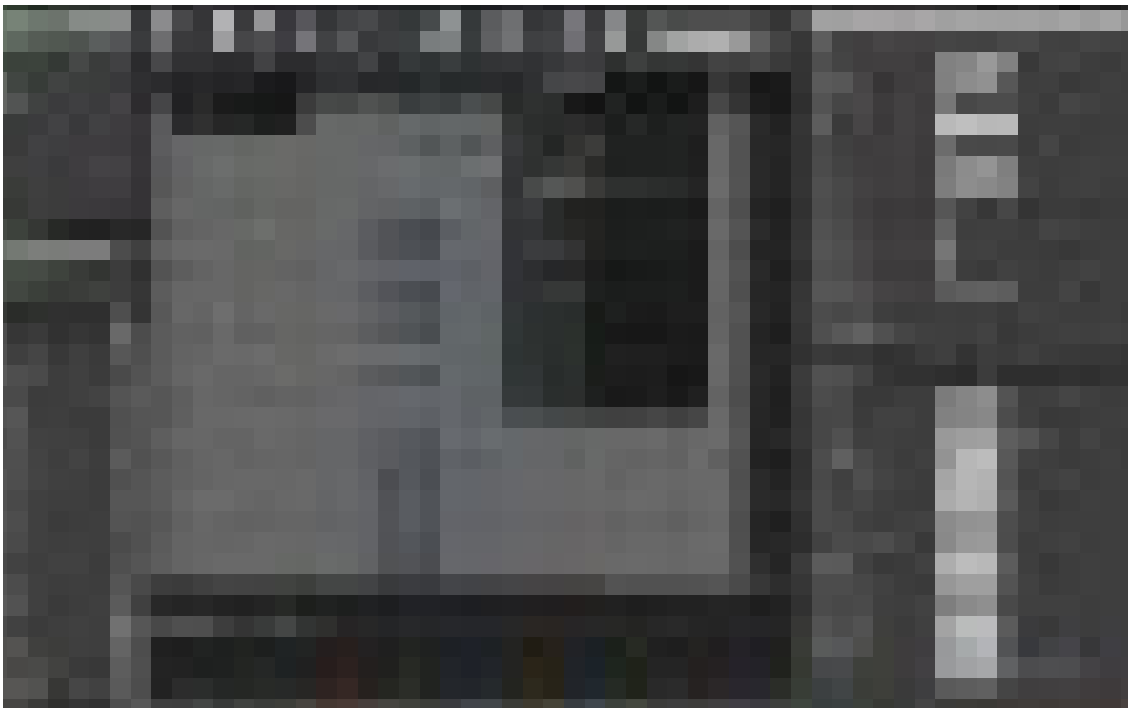
Screen Capture by Author

Coming back to the MoveComponentTo node, find the TargetRelativeLocation pin and drag of a MakeVector node. Connect the relative location Y and the relative location Z to their corresponding pins in the MakeVectorNode. The only one we want to change is the X value, since this is the plane in which the cart will move, ie side to side. So drag off an additional node from Relative X location and add the “actionselected” variable to it. Now that is complete, we need to add a delay so that when the cylinder is falling and gets reset, everything is temporarily paused and MindMaker doesn’t just continue generating actions “mindlessly”. To do this create a branch node after the set members in MindMakerCustomStruct node. The branch should check the status of the Boolean Start variable we created in our Cyl_BP . From the false pin, drag of a node to a delay call, setting this .1 seconds, and then reconnect it to the branch node, in a loop. From the true pin, it should just connect up to the Emit node that was there before.



Screen Capture by Author

Our last and final job is to configure the LaunchMindMaker function to work with the environment we have created. Fortunately, this is pretty simple. The first thing we need to do is choose which RL algorithm to train with. For this task we will use Proximal Policy Optimization, or PPO2 from the dropdown. Next we set the number of training episodes, lets try 1000 to begin with. For the number of evaluation episodes set this to 100. To make things more fun, lets allow the agent to use a continuous action space instead of a discrete set of actions. So rather than just moving the cart one space to the right or left, it can move it fractionally, making the action space infinitely larger and more difficult. In the action space section we need to specify the range over which the agent can take actions, let's do between 1 and -1. This is an extremely large space when you consider the decimal values between 1 and -1. This is entered in the format used by OPEN AI to define action spaces and should read “low=-1, high=1,shape=(1,)”



Screen Capture by Author

Now we need to set the observation space. The RL algorithm in the MindMaker learning engine will be receiving observations about the cylinders pitch, and for our purpose let’s say these will range from -100 to 120 so enter “low=np.array([-100]), high=np.array([120]),dtype=np.float32” This is again from the OpenAI format for observation spaces. Next, if we wanted to use any custom parameters for our RL algorithm, there’s a lot there that can be tweaked. I

include a screenshot of one setup that seems to work well for this task.

That’s it, we're ready to roll! Return to the map level viewport and hit play to begin training. You should see the MindMaker learning engine begin manipulating the cart. The size of the of movements it makes, essentially the random actions it is performing to learn how to balance the cart, will stream across the screen. After a thousand repetitions or so, it should have learned to balance the cart. In many cases, this may be done by taking very small movements that don’t cause the cart to fall.

In summary, we have seen how plugins such as MindMaker are extending the scope of reinforcement learning, opening up new frontiers in both video game AI and machine learning. This extends far beyond toy problems such as the one we have encountered here, impacting both scientific and technical fields including robotic simulation, autonomous driving, generative architecture, procedural graphics and much more. In this way, plugins like MindMaker bridge an important gap between simulator, algorithm and reality.

As an example, recently one of the founders of the company DeepMind expressed an interest in discovering a room temperature superconductor using reinforcement learning. For such a goal, the issue of creating the right simulator could be incredibly difficult since the simulator would have to capture all the properties of physics and chemistry that interact in superconductors. While the reinforcement learning algorithm needed to solve such a task could be represented in a few pages of code and only require a laptop computer to run, the simulator necessary to model the

environment in which superconductors operate could be so complicated that it would requires years of programming and giant supercomputers to run on. Many of the most obscure properties of physics interact in superconductors and one would need to capture a sizeable portion of these in order to be confident that the solution obtained by the reinforcement learner applied to our physical reality and not just the reality of the simulator. It is therefore crucially important when working with RL algorithms to have access to top of the line simulators such as Unreal Engine.

The use of Unreal Engine's blueprint functionality also decreases the barriers to entry in the field of machine learning. Developers with little to no coding experience can dive in and start using the same algorithms as machine learning professionals, all without needing to get their hands dirty tinkering with Python or Tensorflow code.

If you have questions or inquiries, feel free to leave them here or reach out to me personally. In upcoming articles, I will explore related topics such as the dark art of reward engineering — more specifically, how one sculpts suitable reward functions to use with their custom RL agent.

Reinforcement Learning | Deep Learning | Videogames | Machine Learning | Artificial Intelligence