Self-Balancing Artificial Intelligence

Jaden Williams

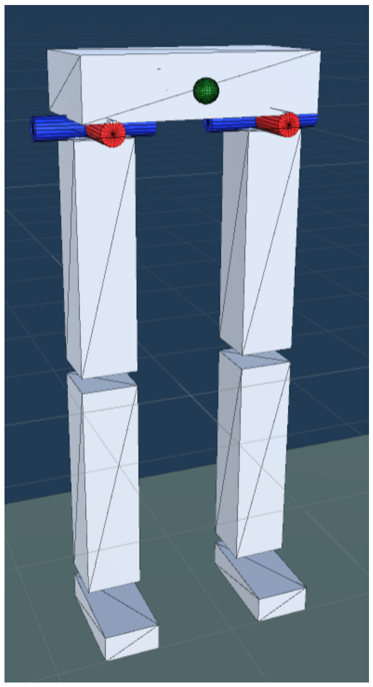
4/22/2020

For the past six months, I, Jaden Williams, have been fantasizing about the concept of teaching an AI to learn how to stand up straight in a virtual physics engine using only humanoid appendages.

The first thing I must clarify is that all of the experimentation and testing was done through the "Unity" engine, a highly powerful physics and rendering engine used mostly for videogame development. To clarify, most virtual experimentations use videogame engines because of the tools at-the-ready for them such as user interface control and design. I chose this engine, unlike other such as the "Unreal Engine" or "Godot" because of it's known physics capabilities and my already previous work with it. The Unity engine uses the C# coding language, unlike the others, as its foundation for programming scripts which is my most familiar language I work with.

The inspiration and reasoning behind the idea came from the wanting for a new idealistic approach to videogame development. Most high end "triple-A" video games companies, meaning major publishers in the industry, use a majority of the time, coordinate-based movement animations that go from one set of 3D vector coordinates to another with no physical attributes being affected besides the position of the "game object" over a period of fixed time. With this system, it allows for no interaction between the user controlling the character and the environment around it. For instance, hypothetically, if a character or "player" were to walk forward and there was a virtual brick thrown at them, the player would not be affected in the slightest because of the lack of physics interaction implemented. There would be no change in the player's velocity, center of mass, and etcetera. A majority of the time in videogames, those things aren't affected because none of it was implemented. The only thing you could really expect is for that brick to maybe bounce off the character if it wasn't going to phase through it already. And there are reasons for this as well. Most of the time it's to save processing power for the computer or console because physics simulations come with costs. But what if you started with physics and built up from there? I wanted to create a self-balancing character that used a joint-based walking and standing system, much like how us humans have in real life to interact with the environment around us. If the character were to walk down a staircase, the hips would move, the center of mass would change, the magnitude of velocity would increase, all sorts of things that you wouldn't have in a triple-A game would be happening in yours.

To get an idea of the layout of the character, the player has several different appendages and several different joints that only rotate on one axis per joint. Each appendage also comes with a collider or "box-collider" matched with the rendered edges of the body part. The character has two feet with two ankle joints, two lower leg portions that I refer to as the "legs" with two knee joints, two upper leg portions that I refer to as the "thighs" with two upper-leg joints, and two "hip" joints parented to those upper leg joints that move in the perpendicular axis to the upper-leg joints. The ankles rotate forward and backwards, the knees rotate more backwards than forwards because of the humanoid-limits that prevent our knees from bending forward, the upper-leg joints rotate forwards and backwards, and the hips rotate left and right. The combination of the upper-leg joints and the hip joints, which are both right on top of each other at the same "vector3" position, or 3D vector position in world space, allow for the simulation of one joint that rotates in two axis much like how a human's upper-leg rotates. At the top of the body, all joints are connected back together by the time they reach the waist game



object, which is primarily used for holding the two legs together. Lastly, all joints and game objects come with a set of two vector3 coordinates: A vector3 position and a vector3 Euler-angle/rotation. I was able to work with quaternion rotations but I wasn’t familiar enough with it at the time.

In the early tests, prior to using AI, I tried to program into the character that when the waist game object began to fall forward that it would bring one leg at a time forward to try to catch itself. To clarify, the local X axis relative to the character in the positive direction is always in front of him. The Y axis in the positive direction is always aiming up out of his head, and the Z axis is always facing to the left of him in the positive direction. The rotation axis is always rotating around that very position axis. For instance, if you were to want the character to do a cartwheel to the right, which would make his head go more negative on the local Z axis on his way down, you would be wanting him to rotate along the local X axis in the negative direction.

With all this in mind, when he would try to rotate his leg forward to catch himself, he would be rotating his upper-leg joint in the positive local Z axis until his foot was aligned with the waist again on the global “world” X axis. This would prevent him from accidentally rotating his foot too far or not far enough. I also implemented a “buffer” area where if his foot was between 1 to 2 units of the waist on the local x-axis, it would mean it was close enough and would not do anything. This helped prevent a jittering problem because the rotations incremented in units of 1. For an example, if the waist was at the local X position of 5.1 and the foot was at 4.4, if the foot were to go forward again, it would then be at 5.4 which means its now passed over the wanted-x-axis position. This would then in turn make it try to rotate back again, causing a never-ending jittering cycle. So, giving it a small buffer area of a few units helped fix this.

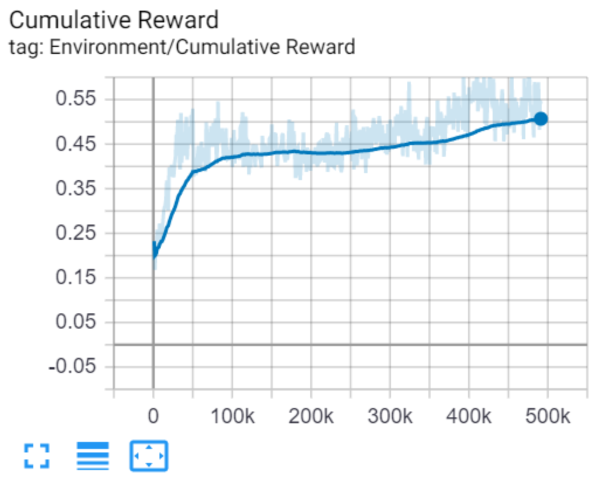
I programmed in several other algorithms similar to this on the hips and knees of the character. If the knee on the local axis was greater than 0, preventing the leg from looking broken, it was allowed to rotate forward if the waist was falling forward. It was also called at the same time as the upper-leg rotation to allow for both joints to rotate forward at the same time. This helped give a more human-like visual to the physics-based animation. So, not only was the upper leg rotating but now the lower leg was rotating as well, essentially doubling the rotating speed to reach the position underneath the waist at the world position. This all also worked for backwards, left and right. In addition, all joints had a minimum and maximum limit of -75 and +75 for the time being. Except for the knee joints, which has a maximum of 0 to, again, prevent the leg from visually looking broken.

The issue I was began to discover was that there was nothing preventing the waist from rotating when the legs rotated underneath it. When the leg rotated, the waist would in turn rotate in the opposite direction. Since the waist was not connected to anything, I did not have any hinge joint for it to rotate off of. So, what I unfortunately had to result in doing was force the waist to counter the change in rotation from the legs, using no physics at all- but forced change in the vector3 rotation. This as expected caused many physics issues. When the legs tried to rotate, the entire body began to slide across the ground because the waist was fighting the laws of physics and there was no proper friction at the time implemented in preventing the feet from sliding wherever. After a couple months of working on this flawed system, I began to realize I was just caking on algorithms over the broken ones to help fix them. In the end, it all resulted in the player jittering, sliding across the floor, and other humorous glitches like floating or body parts abruptly shooting across the world with an extremely high velocity.

After doing research for a month on different routes to take, I started to learn more about the use of artificial intelligence. More specifically, a new early-development built-in Unity system called ML-Agents, where the ML stood for machine learning. In essence, because of the complexity of AI, the summary of what it did was take into account several different choices it could make, see which choices would give it the most amount of “points”, and then try to recreate those decisions on the next simulation, and refining those decisions more and more per simulation, teaching it how to perform whatever task.

I began to teach the character or “agent” how to earn the most points to succeed. Obviously, the best success was to maintain a solid balance. I sent a ray-cast or ray straight down below the waist to the floor. I then grabbed the distance between the waist and the floor and told it to try to maintain a y-position distance of at least 8 to 10 units. If it were to go below 8 units, it would give the agent +1 points times the distance between the floor and a height of at least 8 units. This may seem odd to say that I would give it points if it’s underneath the wanted height level but subtracting points would mean that the closer to the floor it is, the less points it gets taken off. That’s why I needed to give it points rather than take them away. The taller he was, the more points he earned. I also made it that if he was between 8 to 10 units, that would be the best way to earn the most points. This made it aware that too low is not good and between 8 to 10 unites is the best. I implemented the same things for being too high above 10 units. Even though, technically that should never even be possible, I wanted to make sure that the agent didn’t try to build poor habits in trying to bounce to stay up-right.

After implementing the height “goal” algorithm, I needed to tell it what options it had available to change. For each joint, I gave it the decision of 3 options: It was allowed to move +1, -1, or 0 to stay in the rotation that it was at. If a person were standing up right, you don’t need to constantly be changing the joint rotation of your leg to balance. You just don't move your leg. That's why having an option of 0 is important. It needs to be able to not make a change to be able to balance if the waist isn't falling over in any direction. I also was able to give it the constant values of where each joint position was, where each body part position was, joint rotation, body part rotation, waist height, and the velocity on all body parts and joints. In addition, I added subtle constant force upward to help counter the force of gravity and soften the fall speed. Lastly, I added in the change in floor angle by a random value between -5 and 5 degrees on the X and Z axis. I then sent down 4 more rays on each side of the waist to the floor to help it calculate the angle of the floor below it so it could work with that and change how it approached its balancing when at an angle. This helped teach it the concept of floor angles.

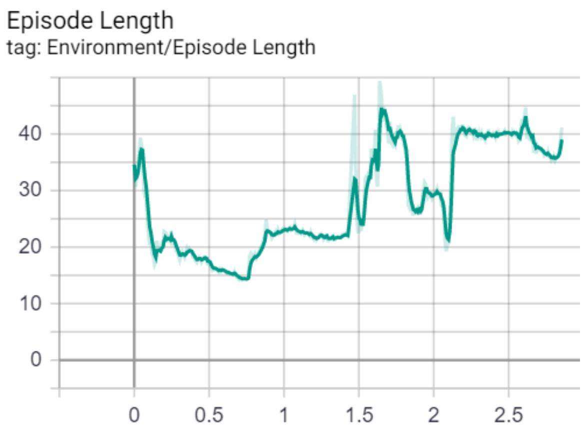


Though, after spending some time on refining the code and adding more algorithms to help optimize the learning, though the agent was indeed learning, he wasn't learning at a fast pace. At this point in testing, he was able to stand up for only a few seconds before falling over. This is then what I came to the realization of: After spending months on ending trying to figure out how to make the code more optimized to help the agents learn more quickly, I was missing the biggest optimization of all- Though I gave the agent only 3 options to choose from when balancing, though that is indeed small, I wasn't thinking about the amount of combinations of degrees it was deciding from. If the ankle could go from -30 to 30, the knee from 0 to 60, the thigh from – 60 to 60, and the hip from -60 to

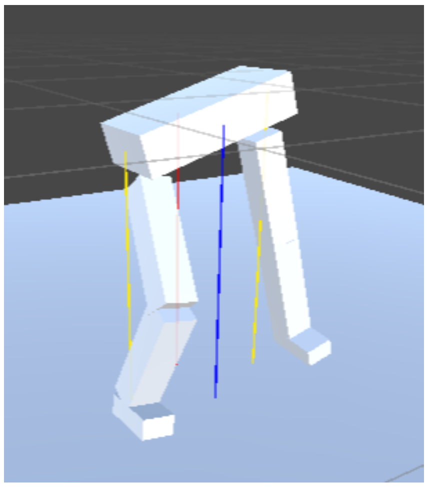
60 that was about 52 million combinations of joint degrees that it was having to choose from. That was quite high.

I decided to take a different approach in the decision-making process that it used. Instead of choosing between going forward, backward, or staying still, I made it choose from a small array of degrees. As an example, I allowed the agents to choose to make the foot angle -30, -20, -10, 0, 10, 20, and or 30 degrees every 2 seconds. The 2 seconds was important because I was now only letting it decide every 2 seconds instead of every frame. This was huge in helping optimize the learning but it also helped prevent jittering. Within those 2 seconds of it not deciding, I made the rotations of the legs increment to the next wanted rotation instead of it just popping abruptly into that expected angle. This was good for visual animation and again, helped stop any jittering because as we all know our legs don't magically pop into the position that we're wanting them to be in.

The next thing I added was the prevention of making the leg go in the wrong direction. If the waist were to be falling over, there would be no reason for the leg to rotate backwards. So dependent on the rotation of the waist, the agent was allowed to make a limited set of which degree angle they wanted their ankle, knee, upper leg, and hip to be at. If the ankle, knee, upper leg and hip could only make at the most 6 or so decisions at time, dependent of the angle of the waist, I went from 52 million combinations to about 1,300. Quite a difference. Even after that, I completely removed the decision-making process of the ankle and forced the foot to be at the same angle of the floor beneath it using ray-casts to



check the distance of the back of the foot compared to the front of the foot. This brought down the amount of decisions by a couple hundred.



In the end, once I began using AI, I went from about 3 seconds before it hit the floor to about 30. Yes, half a minute was the amount of time I reached over half a year of working on it. But I was proud of it. I overcame many obstacles and got that much closer to thinking like an AI. Physical interactions in a virtual environment is one of the more difficult things you can do to make an AI learn. I've always been constantly working on it and have always worked on optimizing it and making it more efficient. I think the biggest thing I learned out of all of it is truly comprehending the difference between a human brain and a computer. A computer doesn't understand what physics is- and may not ever. It doesn't know what a person is. It doesn't understand the concept of balancing or walking. But it does help enlighten even our own understanding of it all. The student is only as good as his teacher and that agent is the student. If I don't understand every concept of the anatomical movement of a human being, the computer won't either.