INTRODUCTION

Goal

What did we try to do? Who would benefit?

We tried to implement inverse kinematics and to get a simulated worm to automatically position its joints in order to get its mouth to eat Indiana Jones. In general applications, inverse kinematics uses kinematics equations to calculate the necessary joint parameters of a robotic arm to achieve a desired position of its end effector (e.g. a hand). This is the exact opposite of forward kinematics, which is the position of the end effector given the joint orientations.

Inverse kinematics is used widely in robotics to control robotic limbs. The application that is more related to computer graphics is its use in animation and the position of joints in 3D models. It allows animators to positions features like hands and feet, and the model will naturally position the other joints, such as the knees and elbows, to create the pose. Inverse kinematics is already fairly robustly implemented in industry applications, so the main benefit is to us, who will hopefully learn the fundamentals of IK.

Previous Work

What related work have other people done? When do previous approaches fail/succeed?

A number of different methods have been employed by people in inverse kinematics. These methods, which will be described and discussed below, include algebra, cyclic coordinate descent, and the Jacobian.

Algebra: calculate the joint angles by manually inverting the forward kinematics solution for the end effector position. For example, the end effector position for a system with 2 DOF can be expressed as $X_{EE} = \left(l_1 \cos\theta_1 + l_2 \cos\left(\theta_1 + \theta_2\right),\ l_1 \sin\theta_1 + l_2 \sin\left(\theta_1 + \theta_2\right)\right)$, giving the following expressions for θ_1 and θ_2: $\theta_1 = \dfrac{-\left(l_2 \sin\theta_2\right)x + \left(l_1 + l_2 \cos\theta_2\right)y}{\left(l_2 \sin\theta_2\right)y + \left(l_1 + l_2 \cos\theta_2\right)x}$, $\theta_2 = \cos^{-1}\dfrac{x^2 + y^2 - {l_1}^2 - {l_2}^2}{2 l_1 l_2}$. It is easy to see that this is really only practical for cases where the DOFs are limited to 2 or 3, as it gets prohibitively complicated past that. Additionally, given the domain of arccosine, the solution does not always exist, and if it does, it is not always unique nor continuous.

Cyclic coordinate descent: Minimization of an objective function $E(\theta) = \left(P - X(\theta)\right)^2$ is applied to each joint separately, with passes going from the most distant segment to the base segment. $E(\theta)$ is the error, P is the goal position of the end effector, and $X(\theta)$ is the current position of the end effector as a function of the joint angles. A number of passes are made to find the global minimum of the error equation. This method works fine for simple structures, but is not as effective for complicated ones. Also, when the change that must be done is near the base, a lot of wasted passes occur, which may slow down computation. Finally, the result is more like a chain than rubber, which may not be what is desired for figure animation. A few things this method excels at though are that it is free of singularities (where a function takes an infinite value), it does not include matrix inversion, and in most cases, only a few passes would be required, so computation is reasonably fast.

Jacobian matrix: The Jacobian is defined as the multidimensional extension to the differentiation of a single variable. Essentially, it shows how changes in the various joint angles affect the Cartesian position of the end effector via the formula $J\Delta\theta = \vec{e}$, where $\vec{e}$ represents the desired change in the position of end effector and $\Delta\theta$ is the amount to update the joint

angles at each joint. The variable we are most interested in is $\Delta\theta$, so the equation is transformed to $\Delta\theta = J^{-1}\vec{e}$, and the problem is reduced to finding the Jacobian and then inverting it.

Entries in the Jacobian are actually quite easy to calculate. If the j[th] joint is a rotational joint with 1 DOF, the joint angle is simply the scalar θ_j. Now let p_j be the position of the joint, v_j be the unit vector pointing along the axis of rotation (which will always be (0,0,1) in our 2D implementation), and s be the position of the end effector. The corresponding entry in the Jacobian is $\frac{\partial s}{\partial \theta_j} = v_j \times \left(s - p_j\right)$. Finding the inverse is the difficult part. First, the Jacobian matrix's dimension grows with the degrees of freedom, as does the cost of calculating the inverse. Furthermore, a matrix that is rectangular (such as any Jacobian matrix that does not have 6 DOF in 3D space) is not invertible. To combat this, two approximations, the transpose and the pseudo inverse are used instead.

The idea behind using the transpose is that, if the changes in angles calculated using the transpose are sufficiently damped (by multiplying by a factor $\alpha > 0$), the result will reasonably approximate those of the inverse.

The pseudo inverse, $J^{\dagger} = J^T \left(JJ^T\right)^{-1}$, is convenient in that it is defined for all matrices, even ones that are not square or not of full row rank, which would have made them non-invertible. Additionally, the pseudo inverse gives the best possible solution to the equation $J\Delta\theta = \vec{e}$ in terms of least squares. However, the pseudo inverse does have stability problems in the neighborhoods of singularities, which means that if the target position is not reachable or the arm is pulled taut, small changes in the target position will cause really big fluctuations in joint angles, which does not look natural.

Approach

What approach did we try? Under what circumstances do we think it should work well? Why do we think it should work well under those circumstances?

To get the benefits of easy calculability and convenience provided by the pseudo inverse, but to also reduce the effects of singularities, we used what is called the damped least squares method. Instead of directly finding the solution to $J\Delta\theta = \vec{e}$, this method tries to find $\Delta\theta$ such that it minimizes the equation $\left|\left|J\Delta\theta - \vec{e}\right|\right|^2 + \lambda^2\left|\left|\Delta\theta\right|\right|^2$, where λ is a non-zero damping coefficient. This is equivalent to solving $\Delta\theta = J^T\left(JJ^T + \lambda^2 I\right)^{-1}\vec{e}$. Through trial and error, our λ ended up needing to be 4 to achieve smoothness. By using this method, our program works well for any number of joints and still behaves in the vicinity of singularities, making it look smooth for essentially any target position.

METHODOLOGY

What pieces had to be implemented to execute my approach?

Our approach required implementation of the following pieces: representation of the joints/arms, calculation and updating of the joint angles, rendering of the arms.

For each piece…

Were there several possible implementations? If there were several possibilities, what were the advantages/disadvantages of each? Which implementation(s) did we do? Why? What did we implement? What didn't we implement? Why not?

Representation of the joints/arms

We decided that it was easiest to implement the joints and arms as an array of Three.js Line objects, which made it easy to iterate through each arm's vertices in their intended order. We were then able to use the Three.js library to easily calculate angles between the different arm segments and update the positions of the joints by starting from the base and iterating outwards. We also kept an array of radii, which we were able to use to add or remove elasticity from the arm segments. We chose this instead of explicitly storing, for example, each arm segment individually because this was more efficient, easier to manipulate, and sufficiently expressive for our purposes. Storing each arm segment individually would have also introduced doubling up of vertices, although given the scale of our arms, this was not really an issue that would have mattered much anyway.

Calculation and updating of the joint angles

For our Jacobian matrix, we used a library called Sylvester, which is a powerful math library that allows for a number of matrix operations, such as matrix augmentation and inversion. Augmentation was useful for creating the actual matrix, as we didn't need to worry about accurately sizing/resizing the matrix, and inversion was especially useful for calculating the pseudo inverse, which would have been nontrivial if we had needed to implement it ourselves.

The joint angles were recorded as an array of floats, and were incremented by the calculated angle deltas from the Jacobian pseudo inverse. A constraint of $\pi/2$ was placed on the angles to make the simulation more realistic. These angles were used in a rotation matrix to position the joints after each update step.

Rendering of the arms

The arms were represented as a combination of cylinders and spheres. These were used instead of lines for example, because they were able to be textured and made to look like the cool alien worms that we used for our final demo rendering. These were also easy to create geometry for and render based on our arm representations; simply taking the vertex array and creating the shapes from that.

RESULTS

How did we measure success? What experiments did we execute? What do my results indicate?

We measured success on how well the arms were able to follow the target, in terms of how close they came to the target, how responsive they were to changes in the target position, and how well they reacted to edge cases, such as the aforementioned singularities. We also wanted to test how natural and smooth the movements of the arms looked, which is especially important for IK in an animation application. To test the success of our project, we simply created an arm and observed how well it tracked the position of our cursor. Overall, the results were very good, and the arms followed the target very smoothly and convincingly. Singularities, which occur when the target is out of reach and usually cause massive fluctuations in the arm position, were handled elegantly, with the arms remaining outstretched in the direction of the target. This was made possible by a relatively high damping coefficient, which unfortunately meant that responsiveness to changes in the target position was subpar. This was especially evident when

the target was in reach, in which you can see that the end effector only moves closer to the target and does not actually reach it. However, you can get the end effector to reach the target by wiggling it around in the same area, so it isn't a huge loss. Another issue is that if the arm ends up curling up, which happens when you track a spiral around the pivot, the arm does not return to normal unless you retrace the spiral in the reverse direction. This was a side effect of the angle constraints, because the end effector is constantly trying to fold over the arm to stretch it out, but the constraints prevent it from doing so, resulting in it getting stuck.

DISCUSSION

Overall, is the approach we took promising? What different approach or variant of this approach is better? What follow-up work should be done next? What did we learn by doing this project?

Overall, we think the approach we took is promising. For most cases, the inverse kinematics perform very well and create some high quality, smooth animations. There are a few augmentations that could have been implemented to improve the performance and address a few of the issues mentioned above. For example, *selectively* damped least squares is a method that only damps when it is needed, which eliminates the responsiveness problem caused by handling singularities. Our system also currently only supports rotational joints (where the degree of freedom is described as rotation about an axis), and does not allow for translational joints (where the joint is free to move in Cartesian space). This would have allowed for a slightly more dynamic arm, which would not need to be bolted down at its pivot position. A possibility that would have been opened up by implementing this is to use our inverse kinematics to create a very simplistic model of a worm or snake, that can move across the screen. We also could have figured out a better method of unfurling the stuck arms, so that additional consideration would not be needed by the user to restore the simulation to a working state. For future work, we could have also incorporated our inverse kinematics logic to something like a game to get more interesting interaction.

CONCLUSION

Through this project, we learned a lot about the fascinating world of inverse kinematics. Solving the inverse kinematics problem is actually something that we all do every second of our lives when we reach for something or try to move any part of our body anywhere. However, it is very interesting to see how this translates to mathematics and algorithms, and the difficulty that comes with it. It was a lot of fun trying to make this, and even though there is plenty of additional work that needs to be done to get to an industry-grade implementation, we are very happy with what we have accomplished.