COMP7508 Report of Assignment 1

Ruitian Yang 3036410718

Task 1

Step 1: Software Preparation

For this assignment, I utilized Blender version 4.1. I chose this version because the latest release, Blender 4.2, has compatibility issues with the Rigify plugin, which is essential for creating and managing the skeleton for my mesh. Blender 4.1 provides a stable environment with full support for Rigify, ensuring a smoother workflow during the rigging and animation process.

Step 2: Mesh Preparation

I began by importing the provided mesh into Blender. The mesh served as the base model for the animation, and I ensured that it was correctly scaled and positioned within the Blender workspace. If I had opted to use a different mesh, I would have followed a similar import process, ensuring compatibility and readiness for rigging.

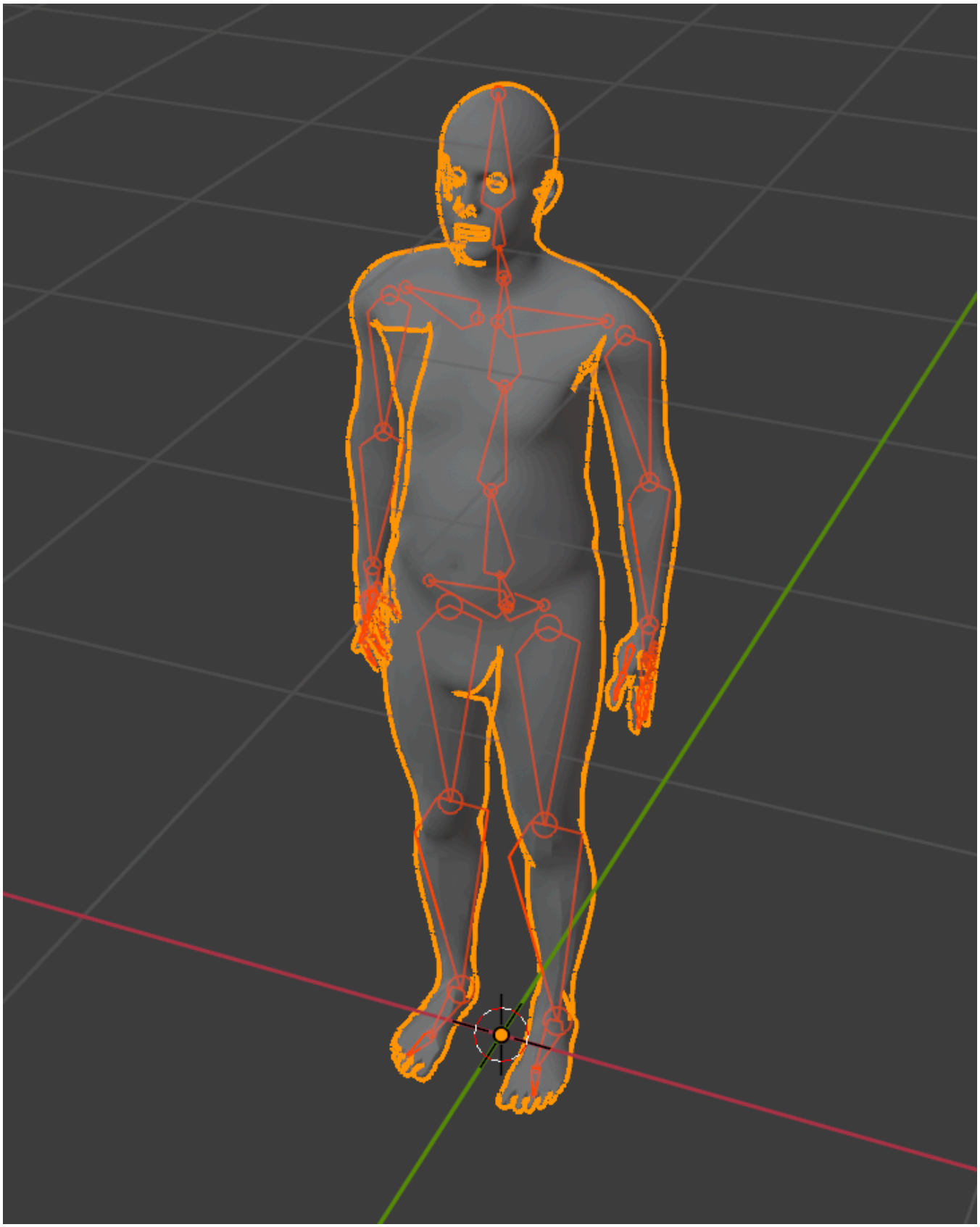
Step 3: Skeleton Preparation

Using Blender's Rigify plugin, I imported a basic skeleton structure. I carefully adjusted the bones to align perfectly with the mesh imported in Step 2 by using transformation tools such as scaling, rotating, and translating. To streamline the skeleton for my specific needs, I removed unnecessary bones, including the two located at the bottom of the feet, which were not required for the animation I intended to create.

Step 4: Rigging and Skinning

After establishing the skeleton, I proceeded to rig the mesh by binding it to the bones. Utilizing Blender's "Parent with Automatic Weights" feature, I linked the mesh to the skeleton, allowing the mesh to deform naturally with the skeleton's movements. This automatic

weighting provided a good initial distribution of weights across the mesh. To enhance the deformation quality, I manually adjusted the weight painting, especially in complex areas such as the joints and limbs. This fine-tuning ensured that the mesh responded accurately to bone movements, resulting in smooth and realistic animations.

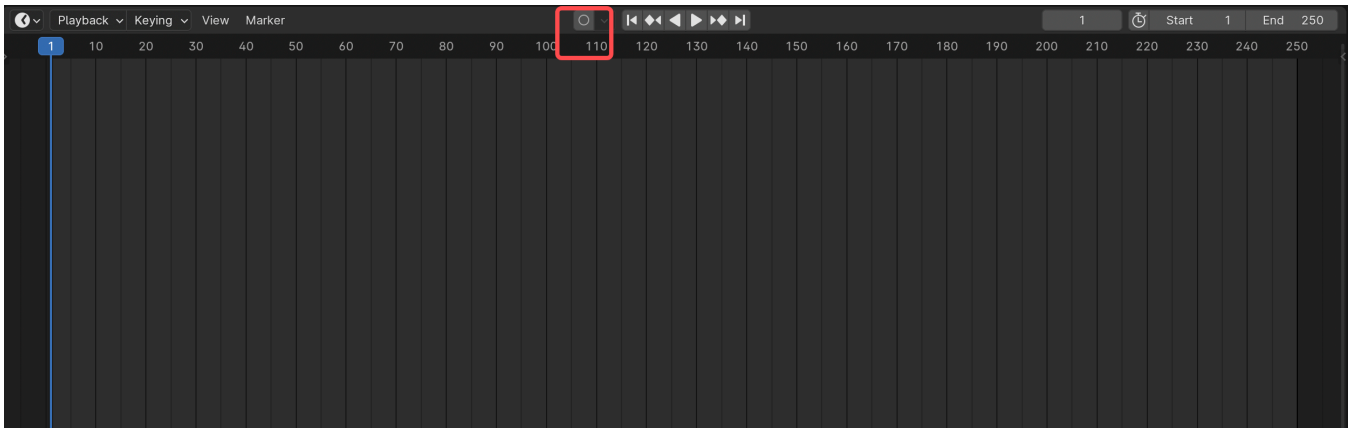


Step 5: Animation Design

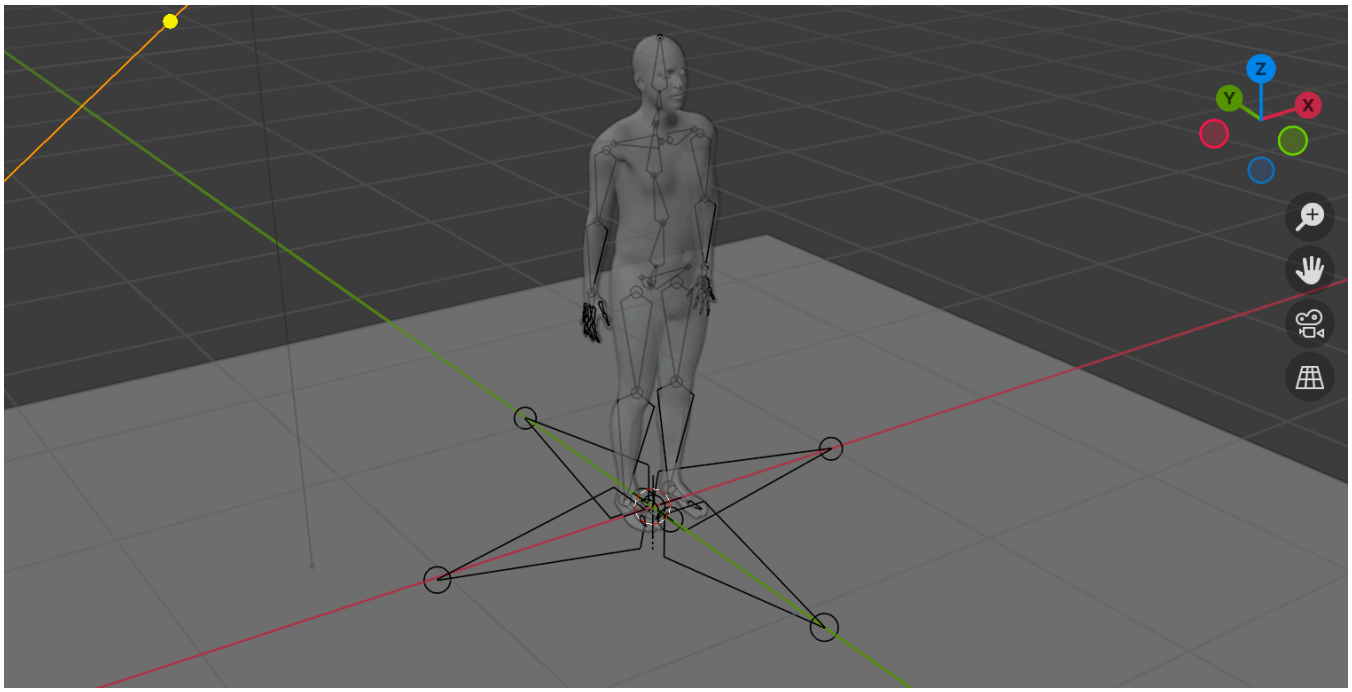
I conceptualized a simple yet dynamic animation where the character performs a single-leg jump on a trampoline. This idea allowed me to focus on both the character's movement and the interaction with the trampoline, providing a clear and engaging visual.

Step 6: Animation Implementation Using Keyframe Animation

Leveraging Blender's keyframe animation tools, I implemented the designed animation. By switching to Pose Mode, I selected various bones within the skeleton and inserted keyframes at specific frames to define the character's movements. To maintain consistency, I enabled auto-recording and set keyframes for all skeleton bones at frame 0. This ensured that any subsequent adjustments preserved the initial pose during video rendering.

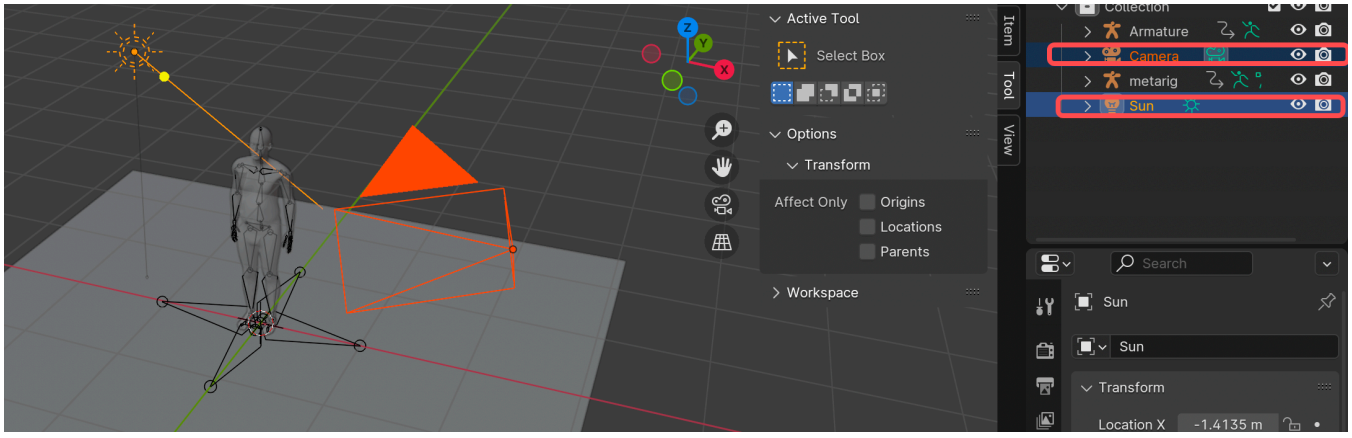


In addition to animating the character, I added a plane mesh beneath the character to represent the trampoline. I applied subdivision to the mesh in Edit Mode, increasing the number of vertices to allow for more detailed animations. I then added four bones to the trampoline and animated them using keyframes, creating realistic bounces and interactions between the character and the trampoline.



Step 8: Lighting and Camera

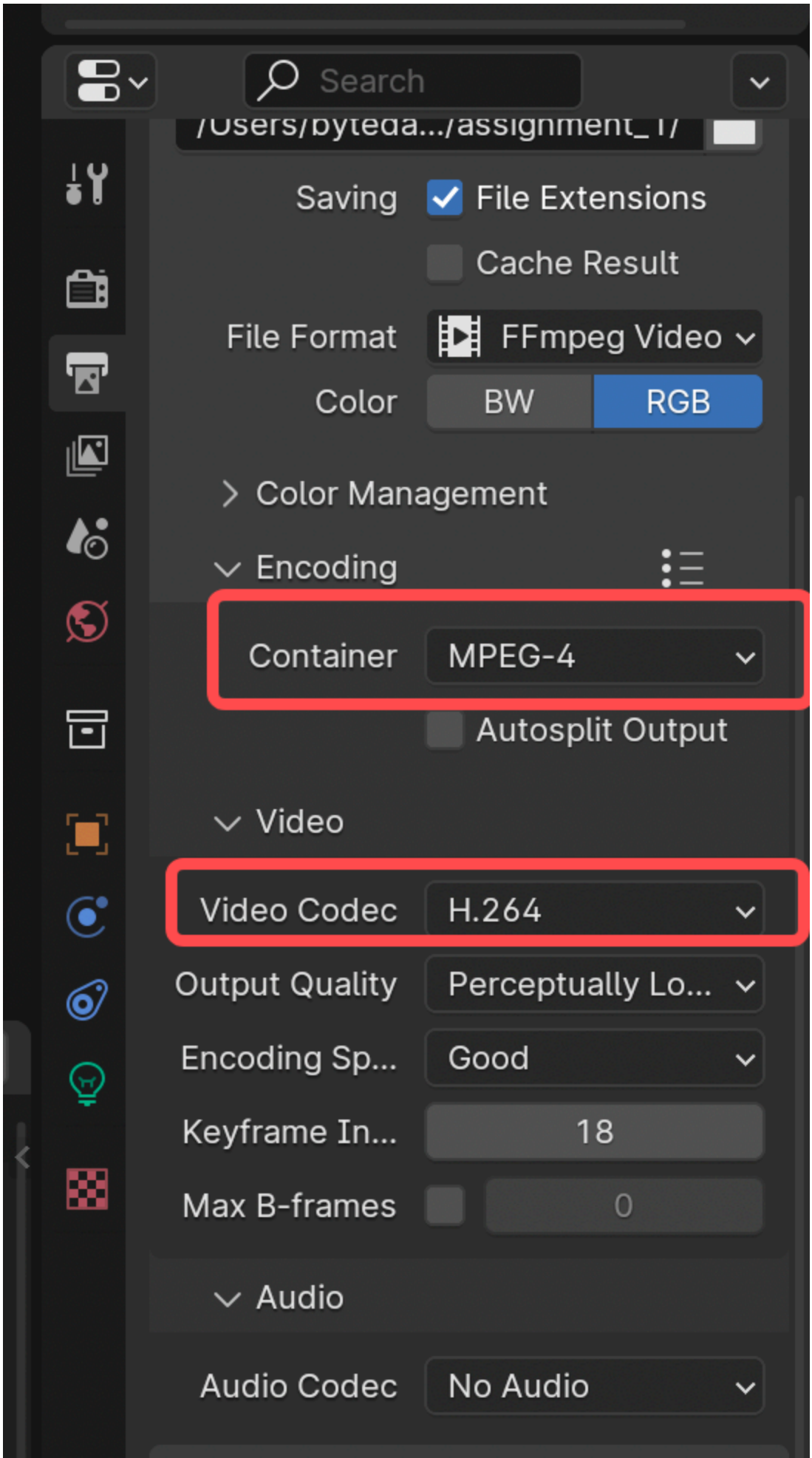
To enhance the visual appeal of the animation, I added a sun light source and positioned the camera to capture the scene effectively. The lighting setup provided natural illumination, highlighting the character and the trampoline, while the camera angle was chosen to showcase the full range of motion during the jump.

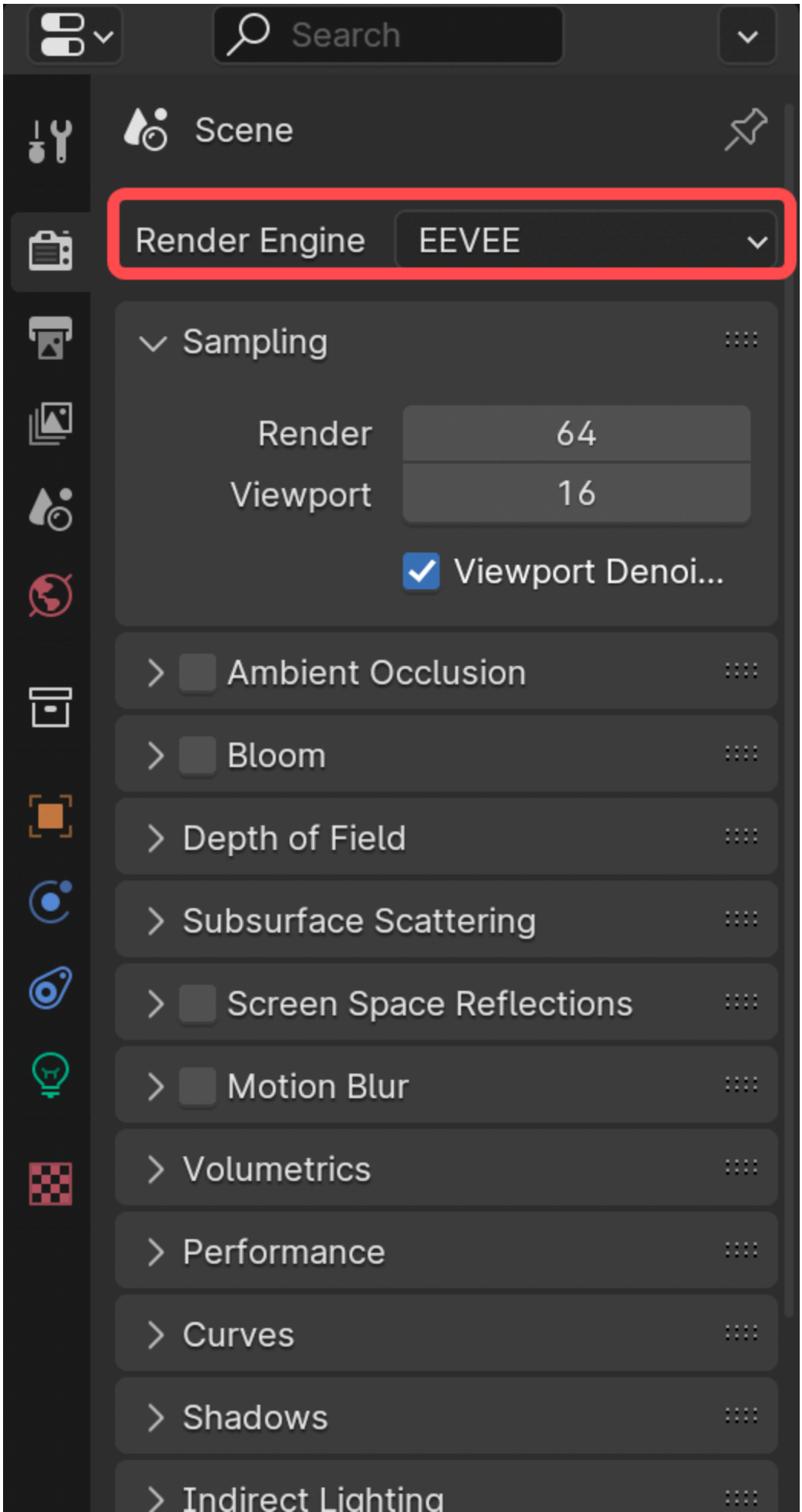


Step 9: Final Rendering

For the final rendering, I configured the output settings to produce an MP4 animation file that reflected the keyframes defined during the animation process. I opted to use the EEVEE rendering engine due to its efficiency and speed, which was suitable for the relatively simple

scene. With the settings in place, I initiated the rendering process by selecting "Render Animation," resulting in a smooth and visually coherent animation.


Search
/Users/byteda.../assignment_1/
Saving
File Extensions
Cache Result
File Format
FFmpeg Video
Color
BW
RGB
Color Management
Encoding
Container
MPEG-4
Autosplit Output
Video
Video Codec
H.264
Output Quality
Perceptually Lo...
Encoding Sp...
Good
Keyframe In...
18
Max B-frames
0
Audio
Audio Codec
No Audio


Search
Scene
Render Engine
EEVEE
Sampling
Render
64
Viewport
16
Viewport Denoi...
Ambient Occlusion
Bloom
Depth of Field
Subsurface Scattering
Screen Space Reflections
Motion Blur
Volumetrics
Performance
Curves
Shadows
Indirect Lighting

Task 2

Implementation of `show_T_pose` Function

The `show_T_pose` function is responsible for displaying the skeleton in a T-shaped pose. To achieve this, I pass a list of joint names, their corresponding parent indices, and an array of local offsets as input parameters. The function iterates through each joint, calculating its global position by adding the local offset to the global position of its parent joint. This hierarchical calculation ensures that each joint is accurately positioned relative to its parent, resulting in a coherent T-pose. Understanding the skeleton hierarchy and correctly computing the global positions are crucial for accurately visualizing the T-pose.

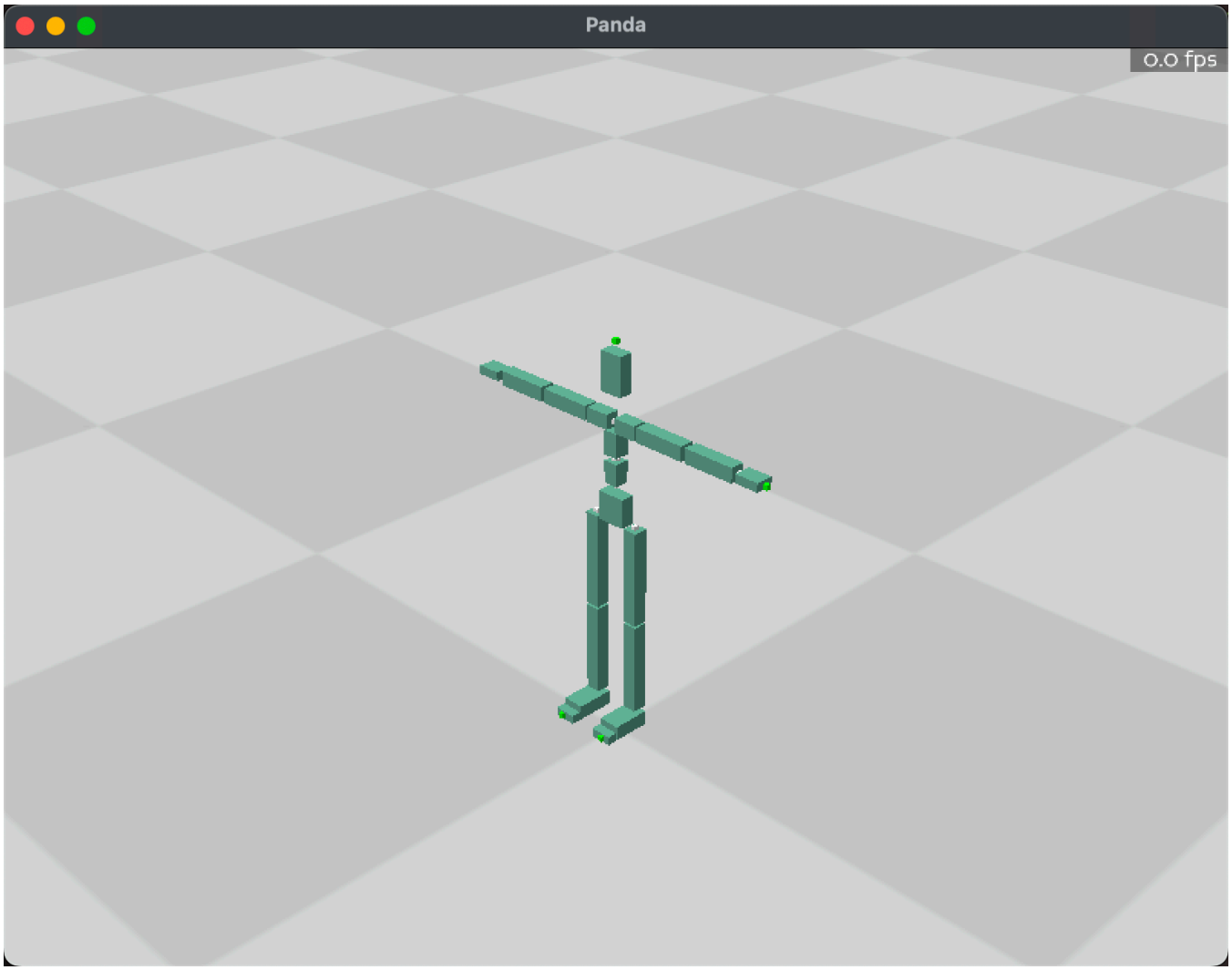
For Root Joint (No Parent)

If `parent_idx == -1`, the joint is a root joint. The global position is manually set, in this case, shifted upward by 1 unit along the Y-axis (`[0, 1, 0]`). This arbitrary shift ensures the skeleton is positioned adequately within the viewer's frame.

For Child Joints

For joints with a valid parent (`parent_idx != -1`), the global position is computed by adding the local offset (`joint_offsets[joint_idx]`) to the parent's global position (`global_joint_position[parent_idx]`). This cumulative addition ensures that each joint's position is relative to its parent, adhering to the hierarchical structure of the skeleton.

Results



Implementation of `part2_forward_kinematic` Function

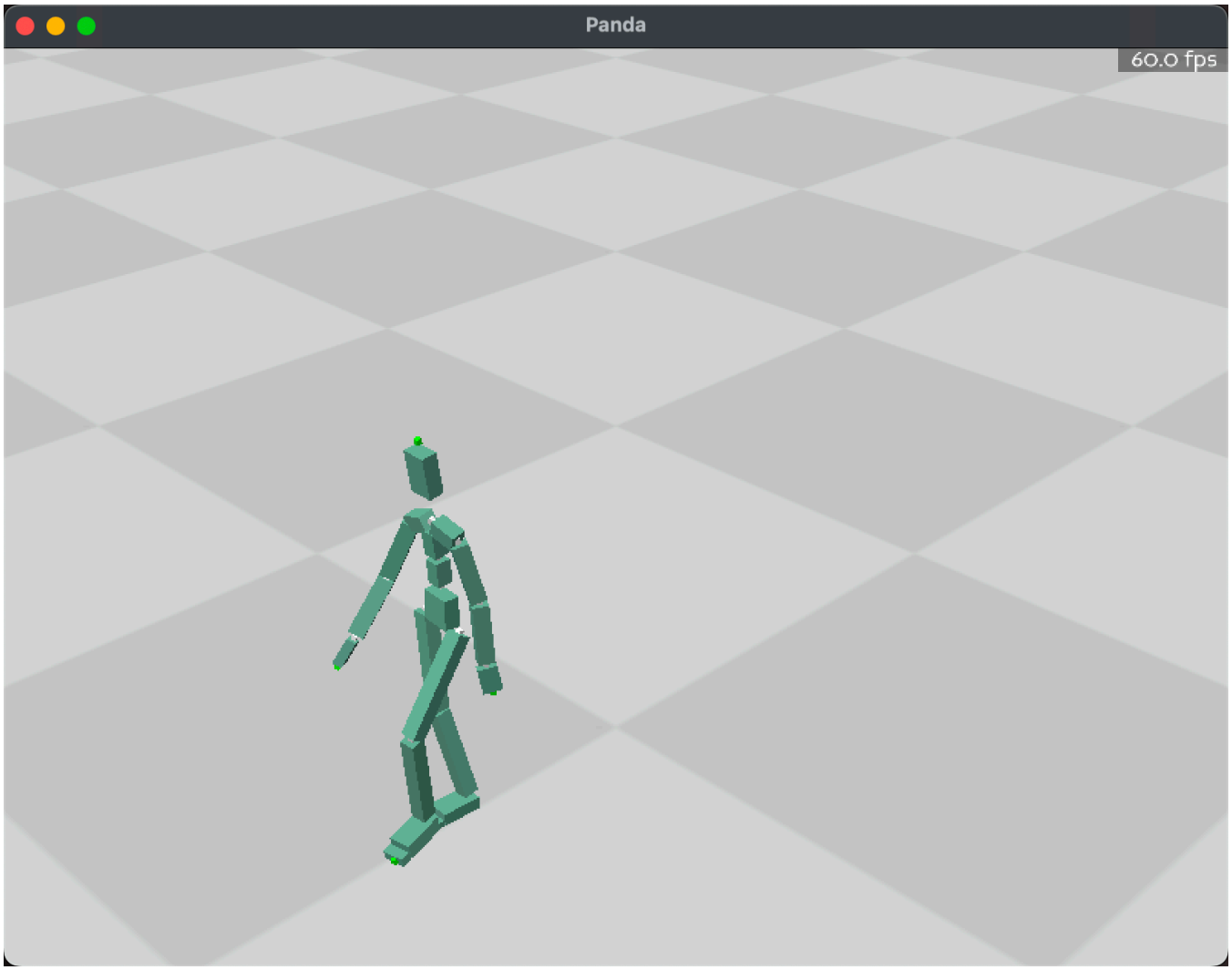
The `part2_forward_kinematic` function calculates the global positions and orientations of joints using forward kinematics. This function takes in the joint names, parent indices, local offsets, local joint positions, and joint rotations as input parameters. It systematically traverses each joint in the hierarchy, applying rotations and translations based on the parent joint's orientation and position. By sequentially updating each joint's global position and orientation, the function ensures that the entire skeleton moves cohesively. Mastery of forward kinematics and a clear understanding of the joint chain structure are essential to implement this function effectively.

Forward Kinematics Computation:

- **Iterating Through Frames and Joints**:
    - The function uses nested loops: the outer loop iterates over each frame (`f`), and the inner loop processes each joint (`j`) within that frame.
- **Processing Each Joint**:
    - **Local Rotation**: Converts the quaternion rotation for the current joint in the current frame into a Scipy `Rotation` object using `R.from_quat` .
    - **Root Joint**:
        - If `parent_idx == -1` , the joint is a root joint.
        - **Global Position**: Directly assigned from `joint_positions[f, j]` .
        - **Global Orientation**: Set to the local rotation (`local_rot.as_quat()`).
    - **Child Joints**:
        - **Parent's Global Position and Rotation**:
            - Retrieves the global position (`parent_pos`) and orientation (`parent_rot_quat`) of the parent joint.
            - Converts the parent's quaternion rotation into a Scipy `Rotation` object.
        - **Rotated Offset Calculation**:
            - Applies the parent's rotation to the local offset (`joint_offsets[j]`) using `parent_rot.apply` .
            - This rotated offset represents the displacement from the parent joint to the current joint in the global coordinate system.
        - **Global Position**:
            - Computed by adding the rotated offset to the parent's global position.
        - **Global Orientation**:
            - Determined by combining the parent's rotation with the joint's local rotation (`parent_rot * local_rot`).

- This ensures that the joint's orientation is relative to its parent's orientation, maintaining the hierarchical relationship.

Results



Task 3

Implementation of CCD Inverse Kinematics

In this task, I implemented the Cyclic Coordinate Descent (CCD) algorithm for inverse kinematics (IK). The CCD algorithm iteratively adjusts each joint in the IK chain to bring the end effector closer to the target position. Here's a breakdown of my implementation process:

1. **Understanding CCD Algorithm**: I began by studying the CCD algorithm, which works by adjusting each joint in the chain from the end effector towards the base until the end effector reaches the target position or the maximum number of iterations is reached.
2. **Iterative Adjustment**: I set the number of iterations to 20 to allow sufficient adjustments for the joints. For each iteration, I looped through the joints in reverse order, starting from the joint just before the end effector and moving towards the base.
3. **Vector Calculations**: For each joint, I calculated two essential vectors:
    - **Vector to End Effector**: The vector from the current joint to the end effector.
    - **Vector to Target**: The vector from the current joint to the target position.
4. **Calculating Rotation**: I computed the angle between these two vectors using the dot product and clamped the result to avoid numerical issues. Additionally, I determined the rotation axis by taking the cross product of the two vectors and normalizing it.
5. **Applying Rotation**: Using the calculated angle and rotation axis, I created a rotation vector and updated the current joint's orientation. This adjustment aimed to align the end effector closer to the target.
6. **Updating Chain Positions**: After rotating a joint, I updated the positions and orientations of all subsequent joints in the chain to ensure consistency and realism in the movement.
7. **Optimization Considerations**: Throughout the implementation, I considered potential optimizations, such as increasing the number of iterations for greater accuracy or adjusting the order of joint updates to enhance convergence speed.

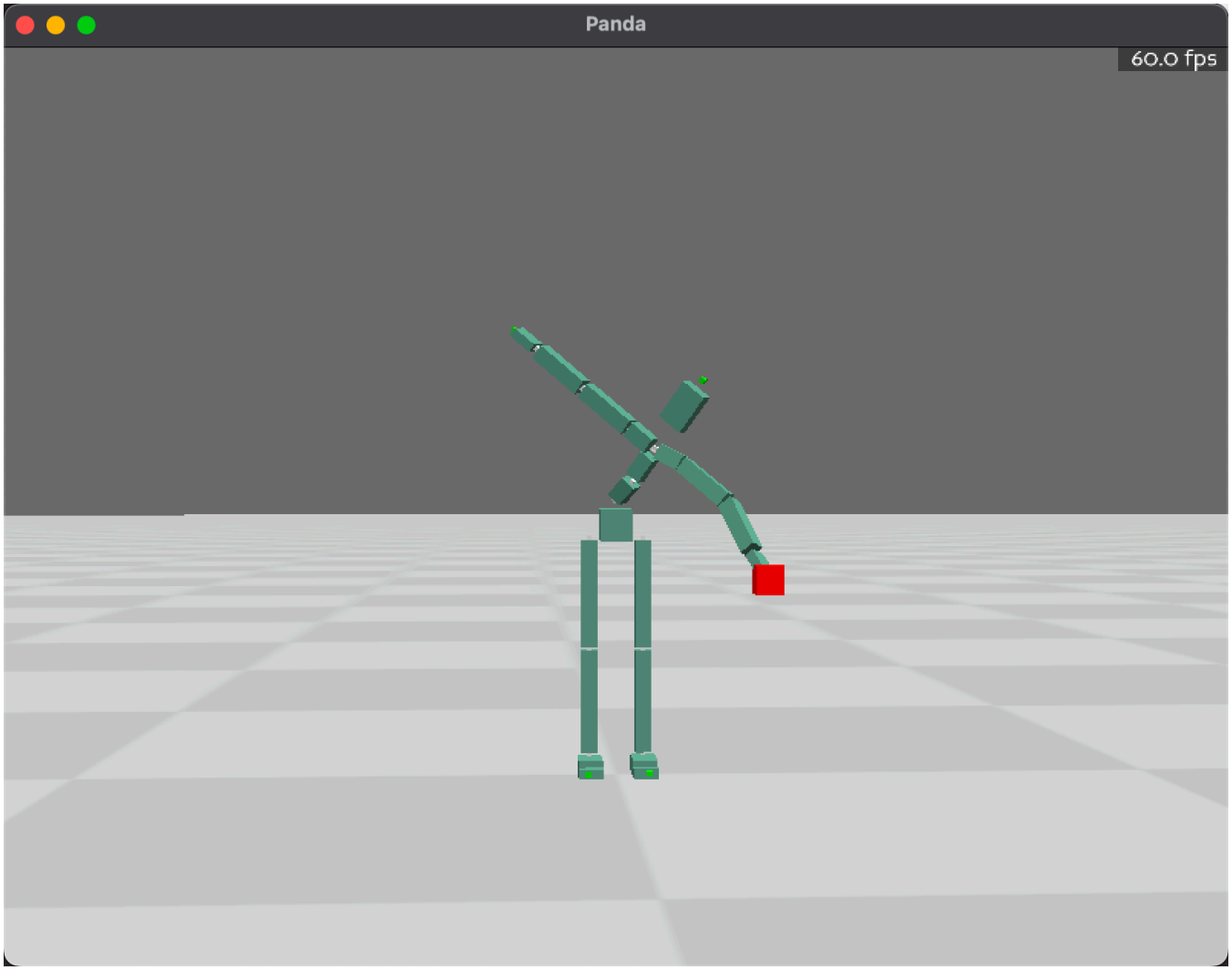
By following these steps, the CCD algorithm effectively adjusted the joints, enabling the end effector to reach the desired target position through iterative optimizations.

In this implementation, the CCD algorithm iteratively adjusts each joint's orientation to minimize the distance between the end effector and the target position. By carefully updating the chain's orientations and positions, the algorithm ensures that the character's movement remains natural and aligned with the intended target.

Results

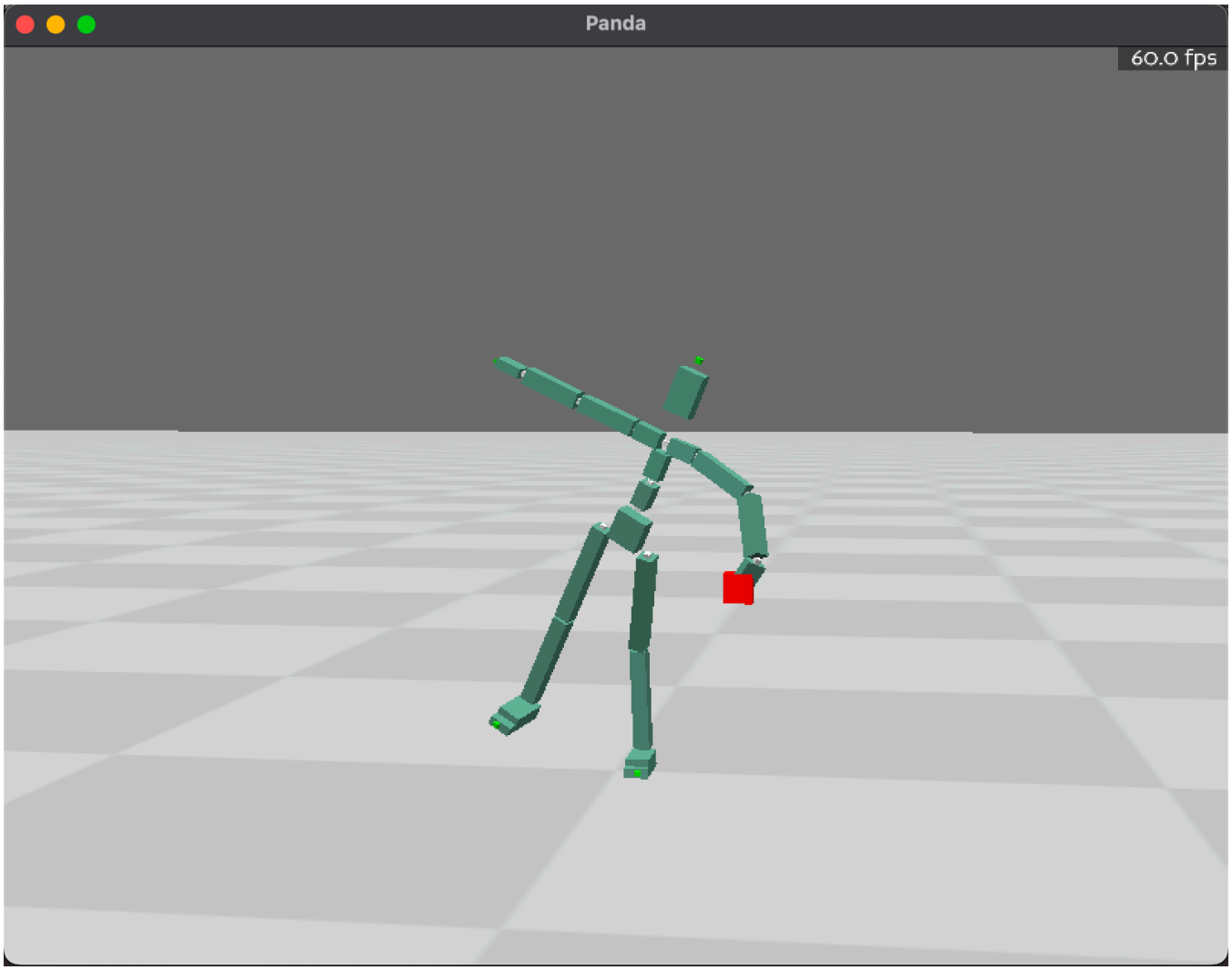
1

```
IK_example(viewer, np.array([0.5, 0.75, 0.5]), 'RootJoint', 'lWrist_end')
```



2

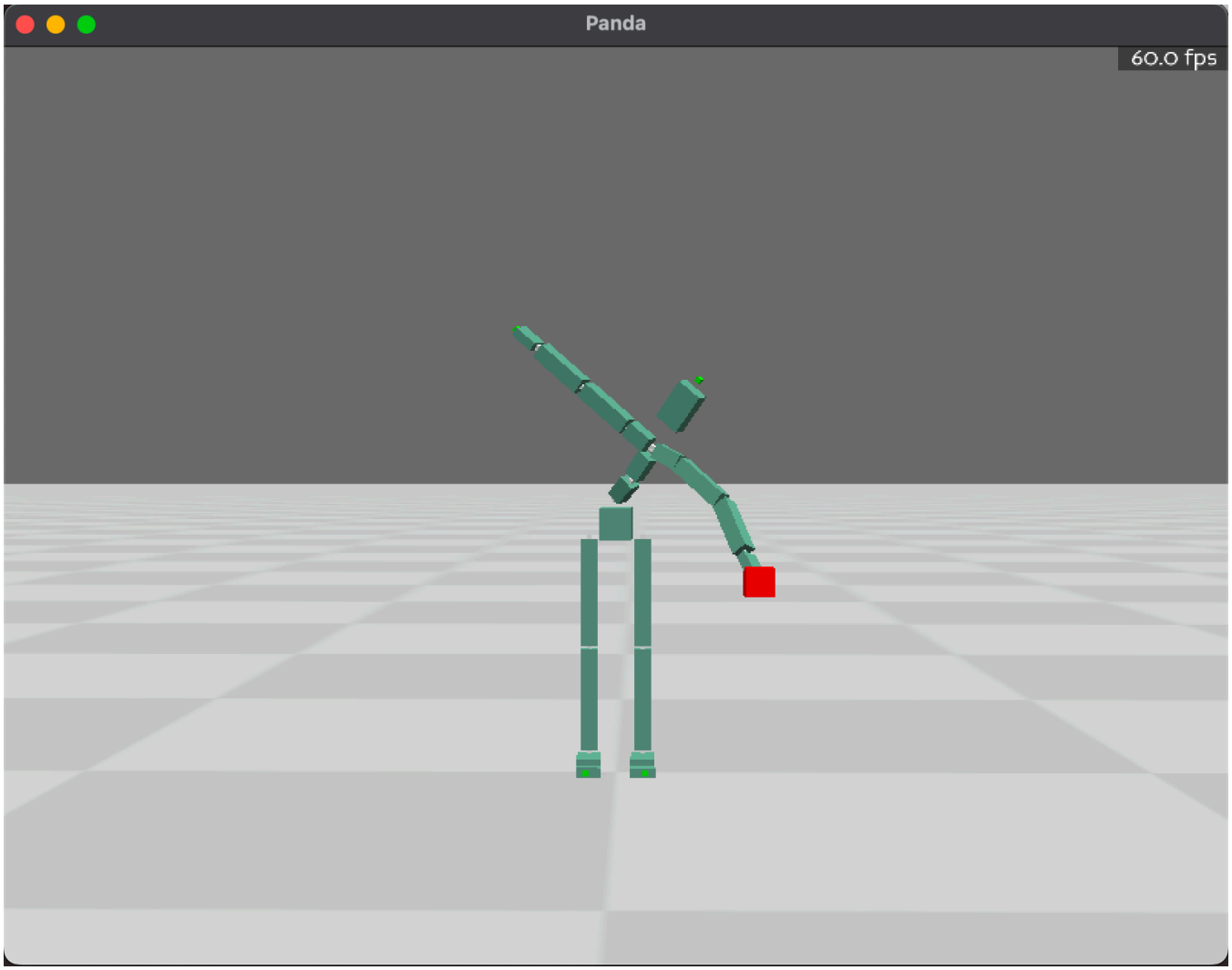
```
IK_example(viewer, np.array([0.5, 0.75, 0.5]), 'lToeJoint_end', 'lWrist_end')
```



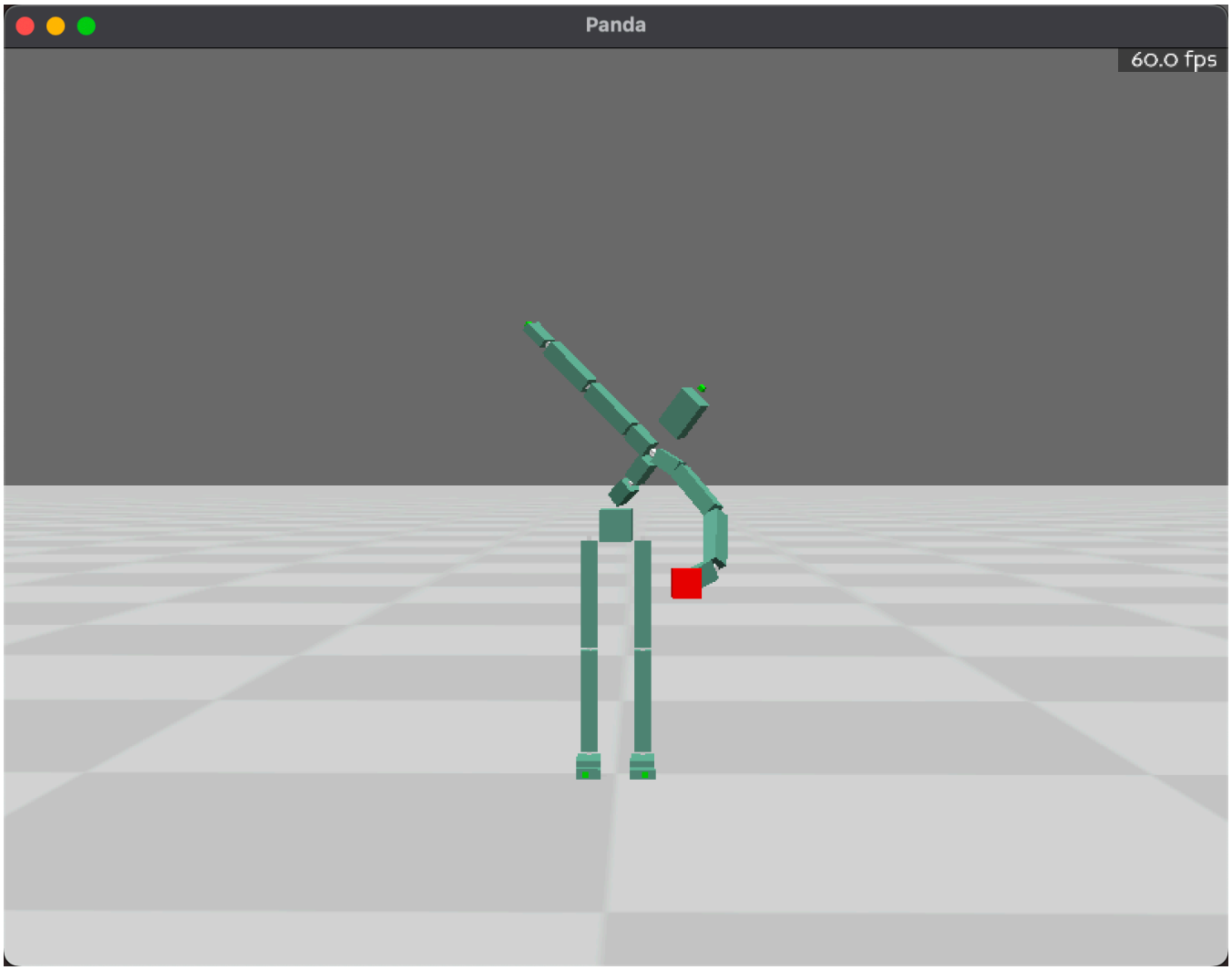
3

```
IK_interactive(viewer, np.array([0.5, 0.75, 0.5]), 'RootJoint', 'lWrist_end')
```

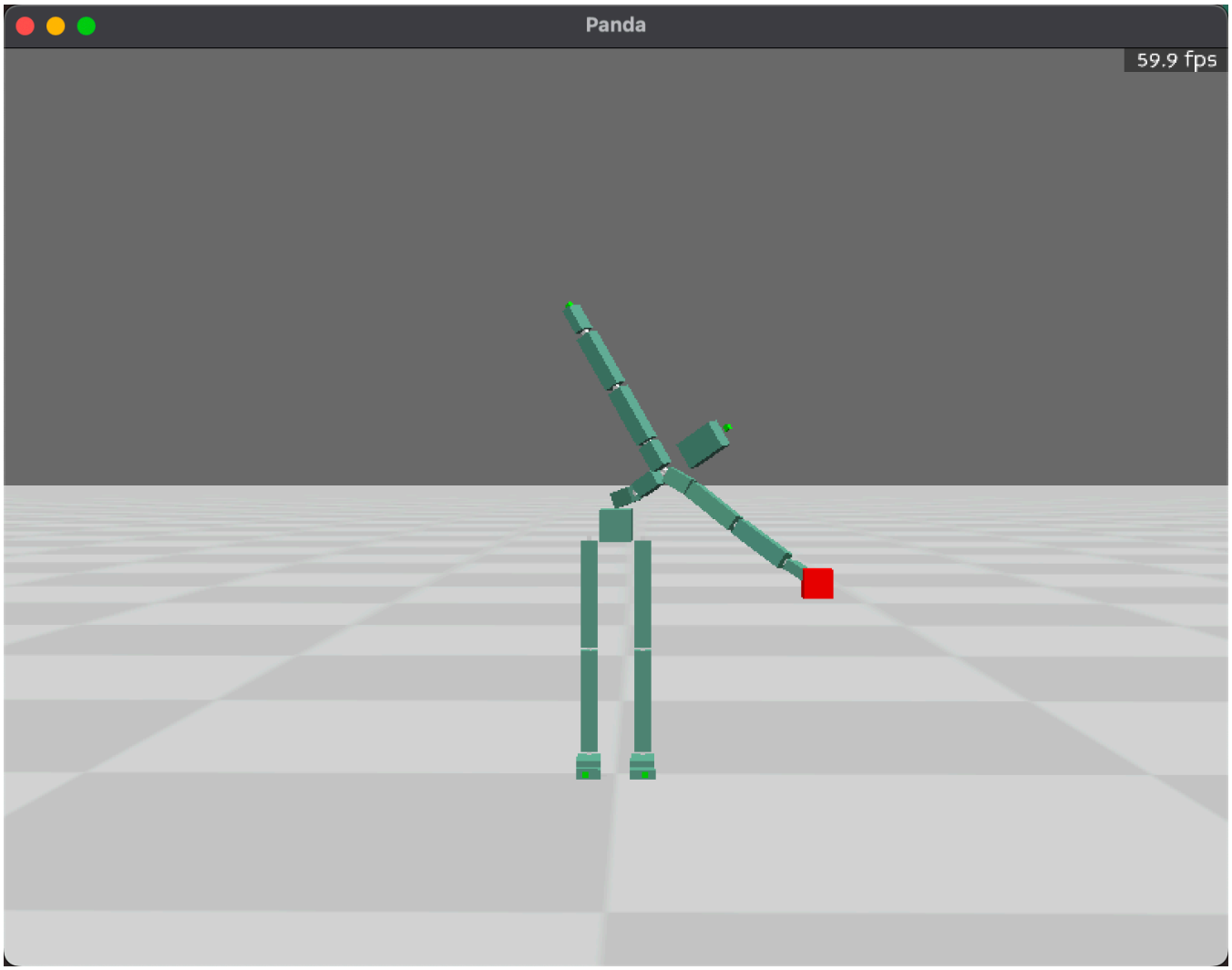
Initial position



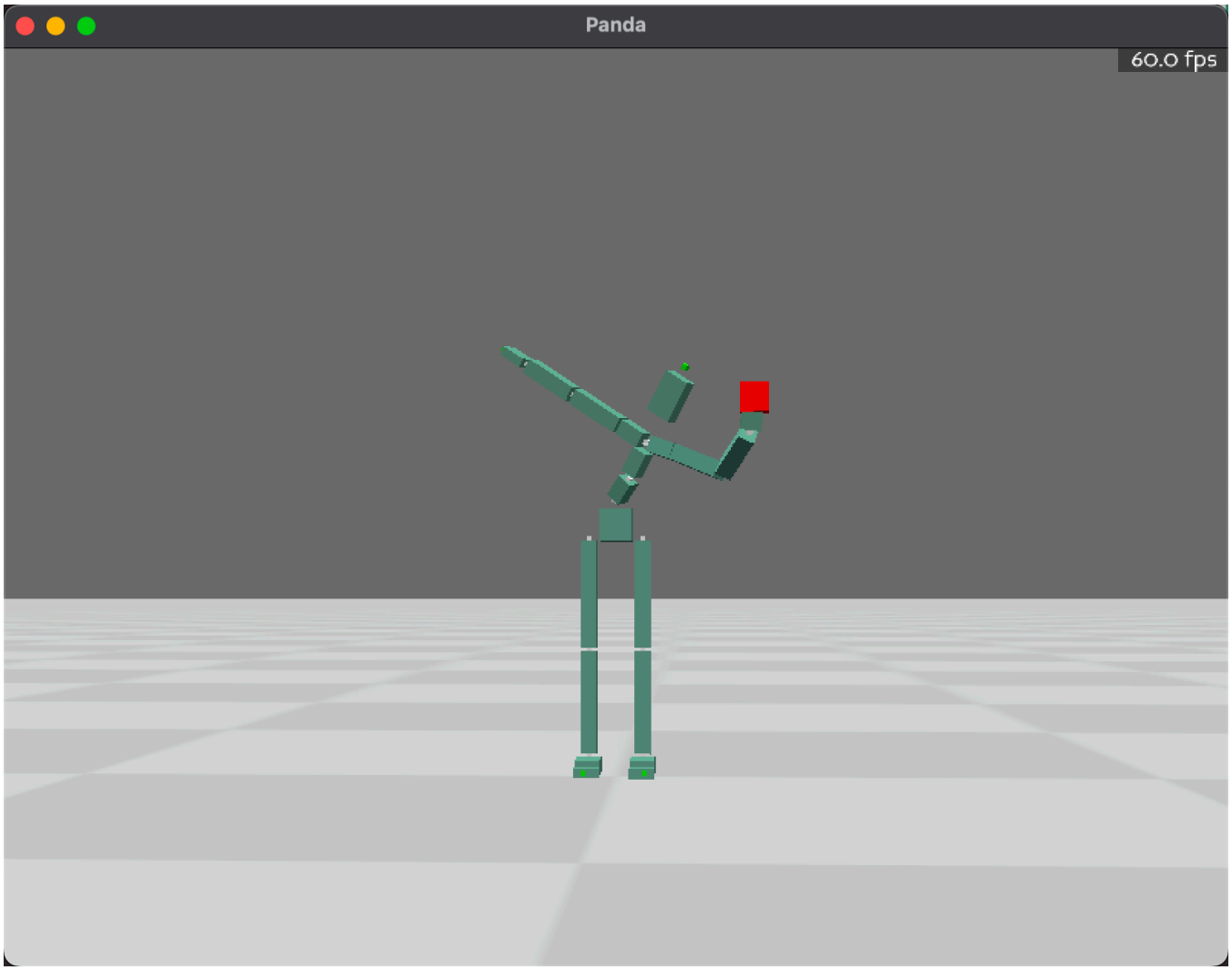
By pressing A from the starting position



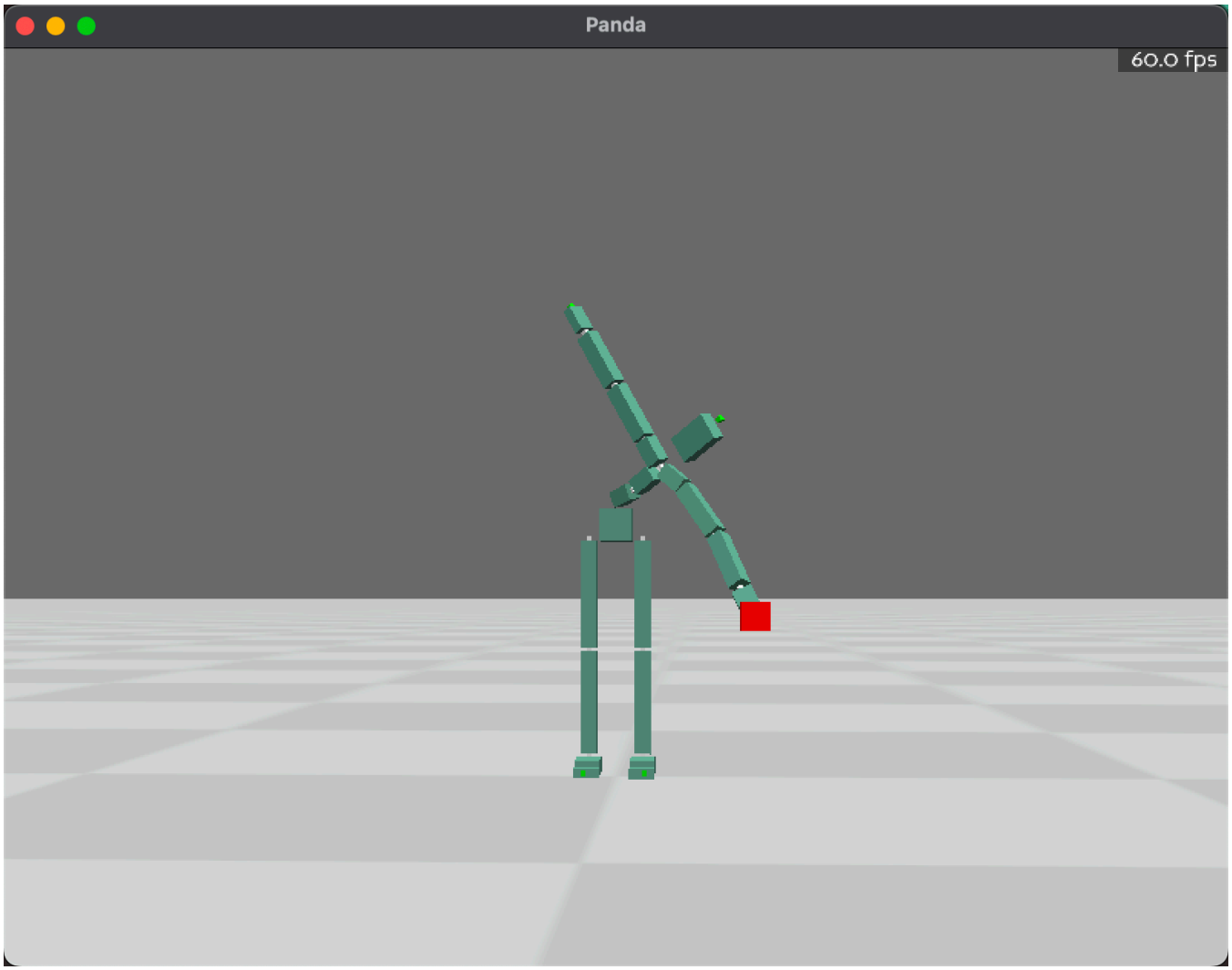
By pressing D from the starting position



By pressing W from the starting position



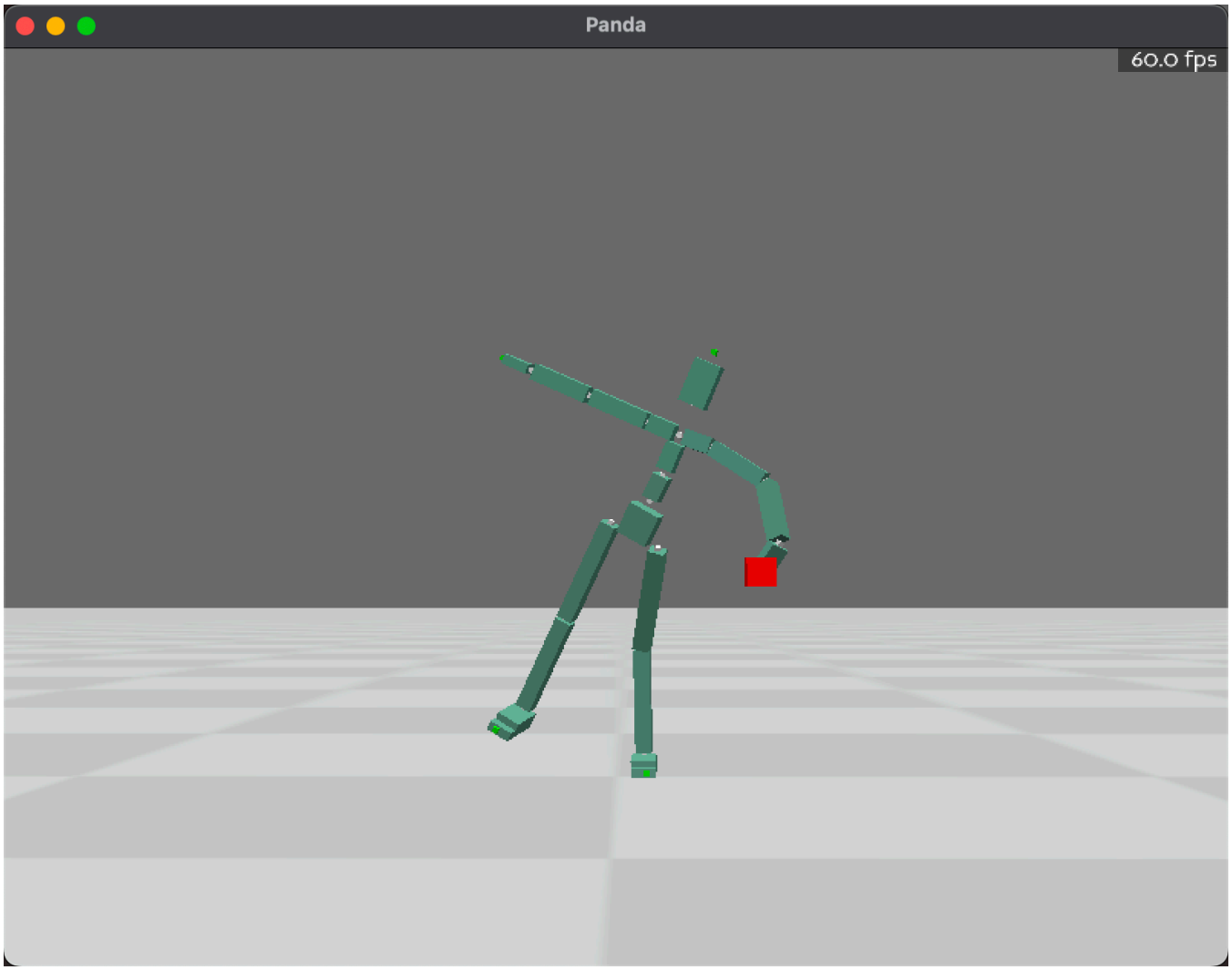
By pressing S from the starting position



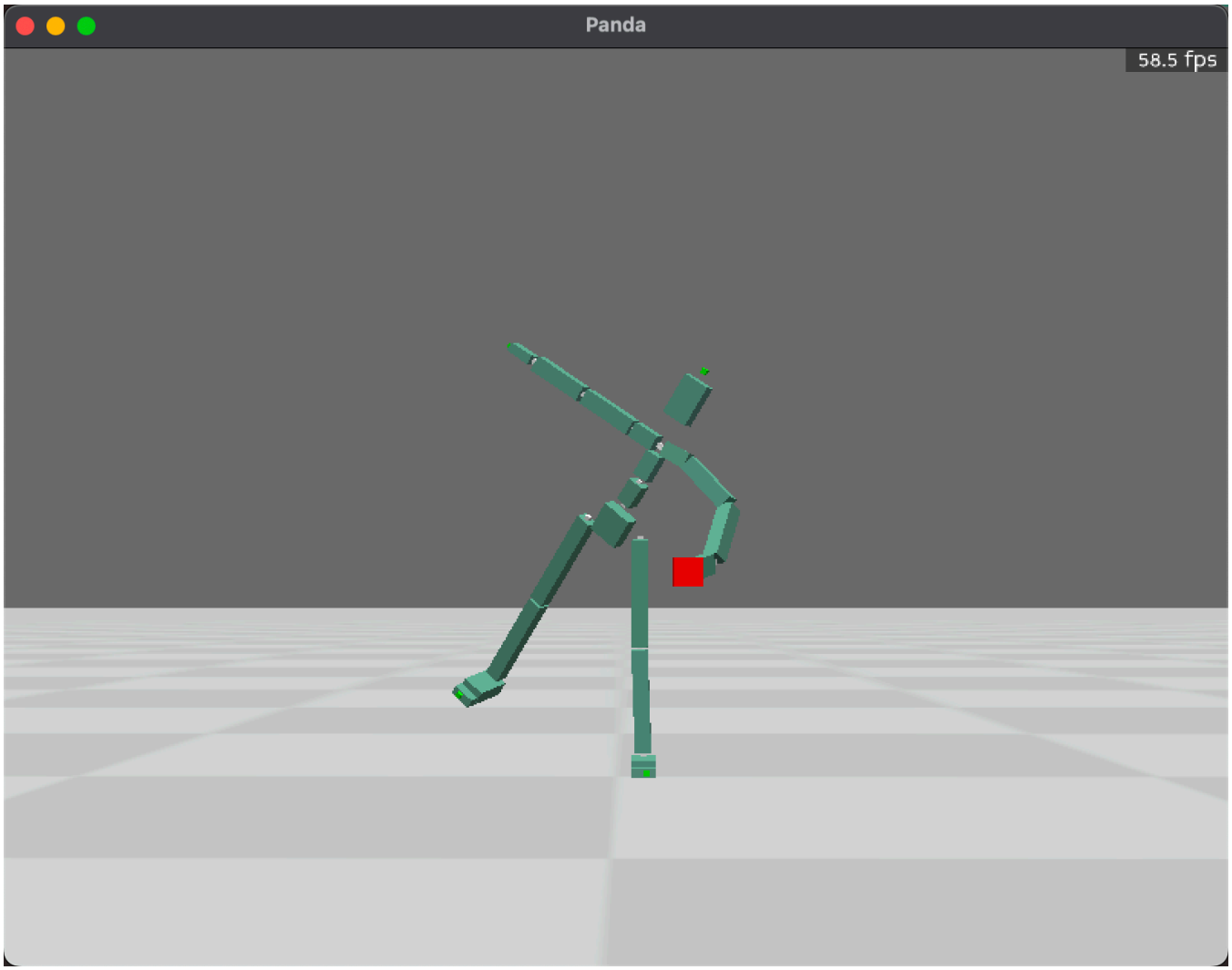
4

```
IK_interactive(viewer, np.array([0.5, 0.75, 0.5]), 'lToeJoint_end', 'lWrist_end')
```

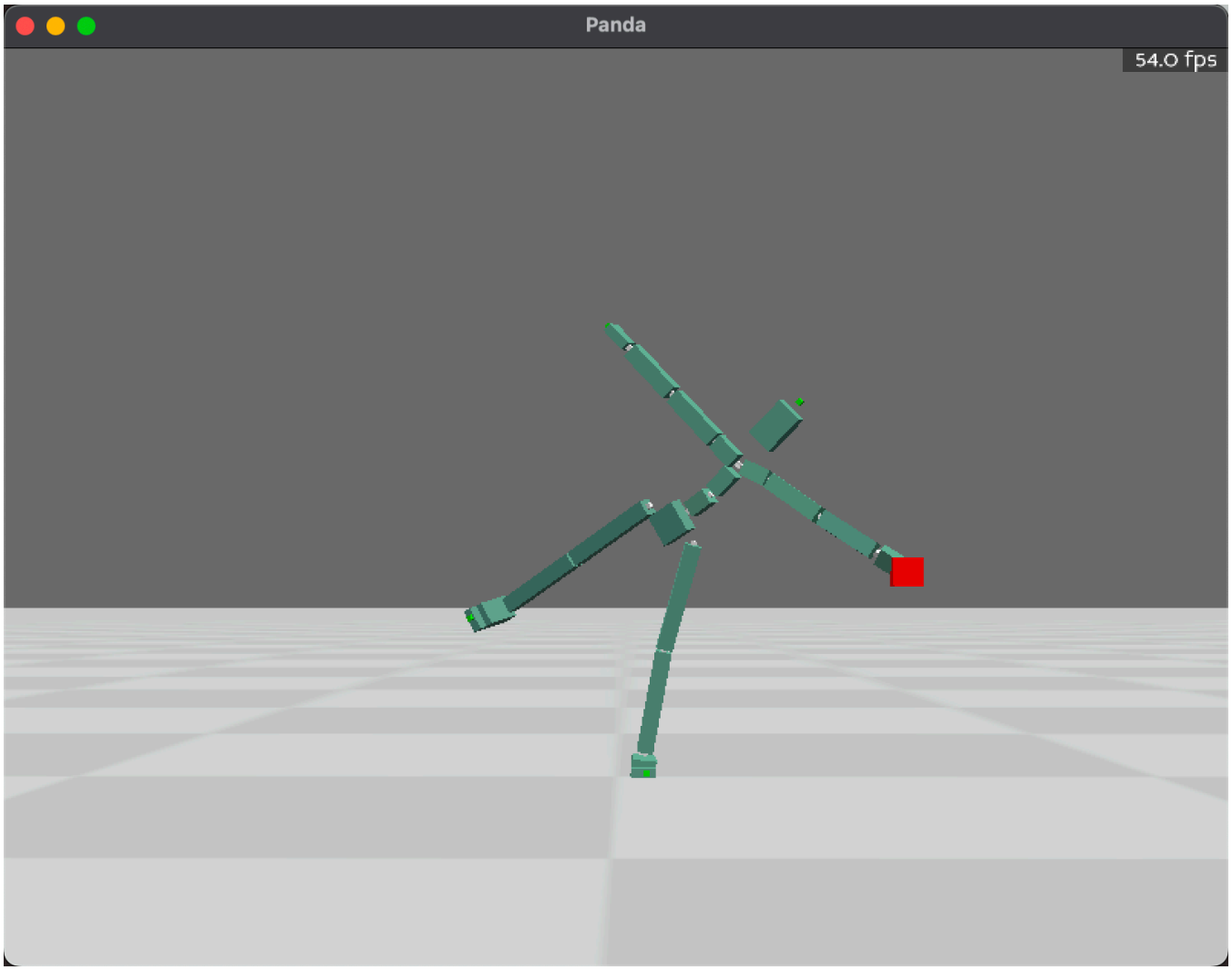
Initial position



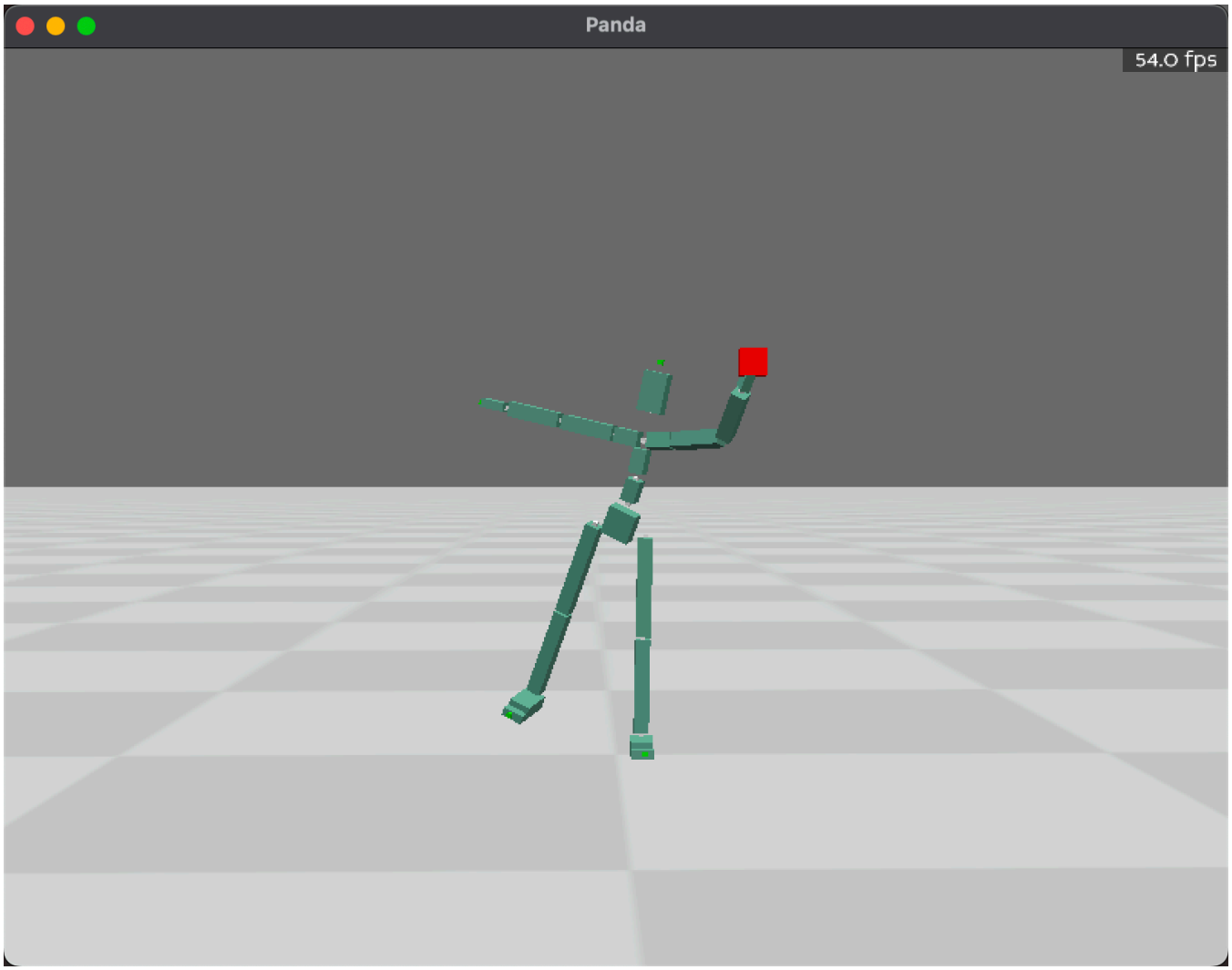
By pressing A from the starting position



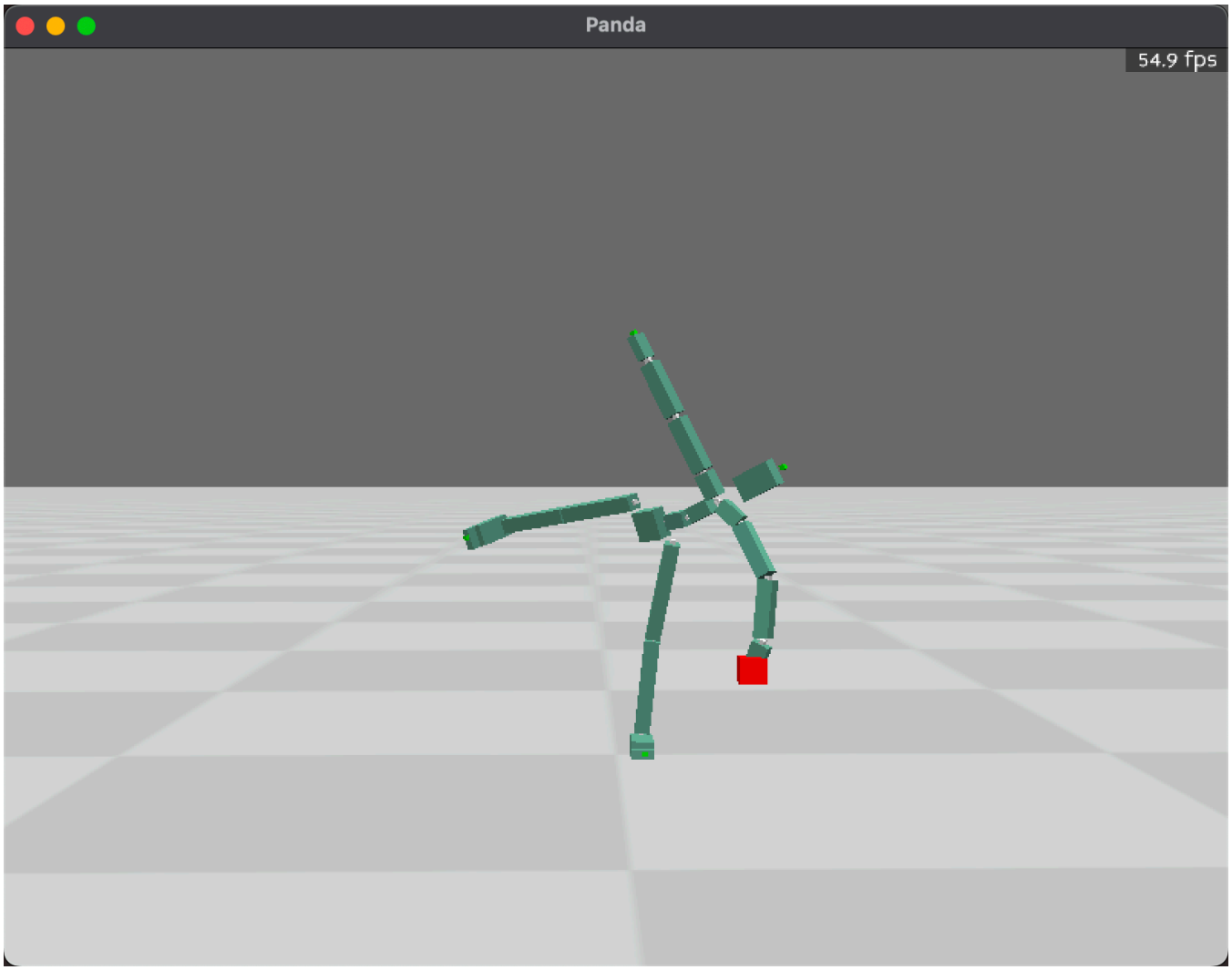
By pressing D from the starting position



By pressing W from the starting position



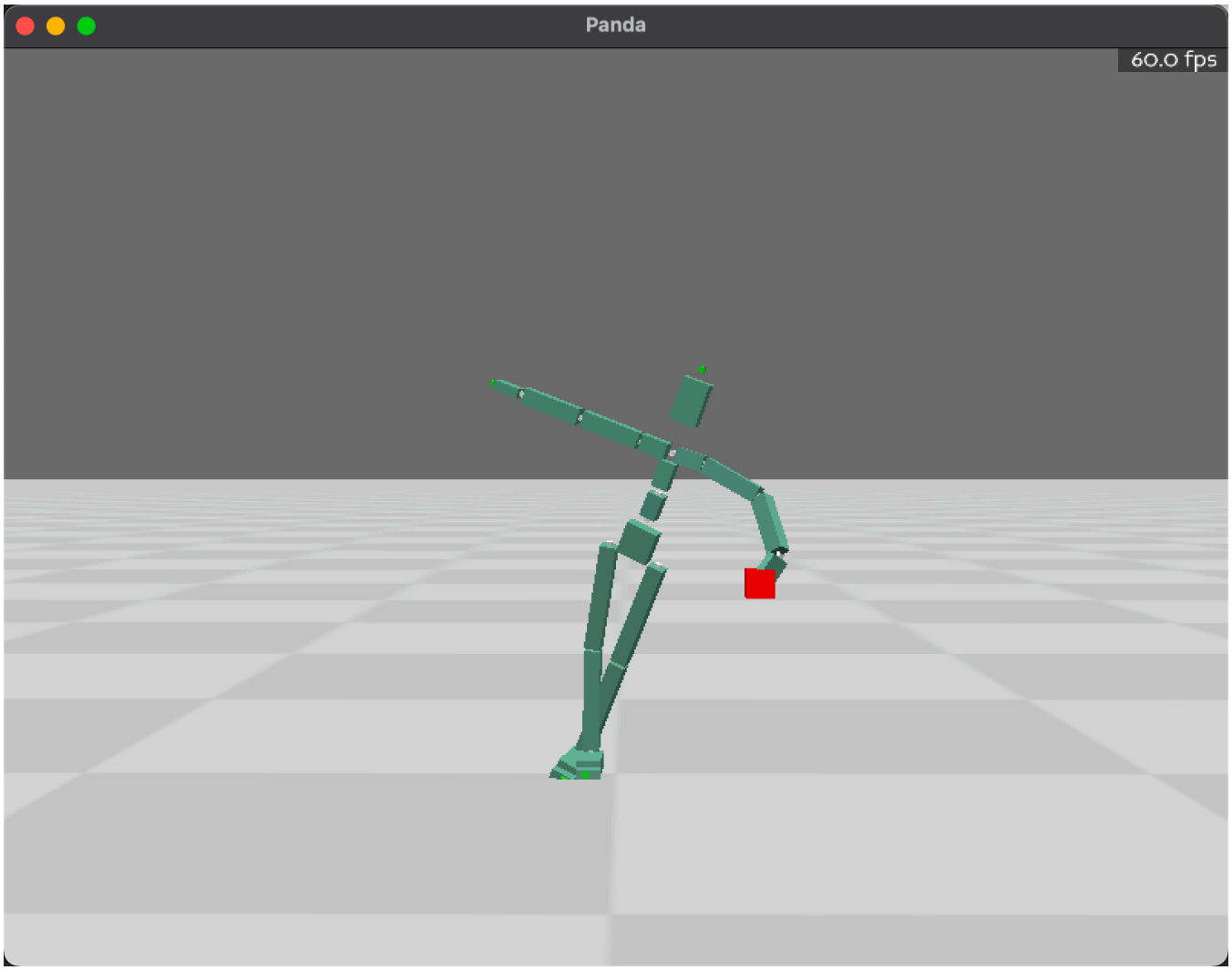
By pressing S from the starting position



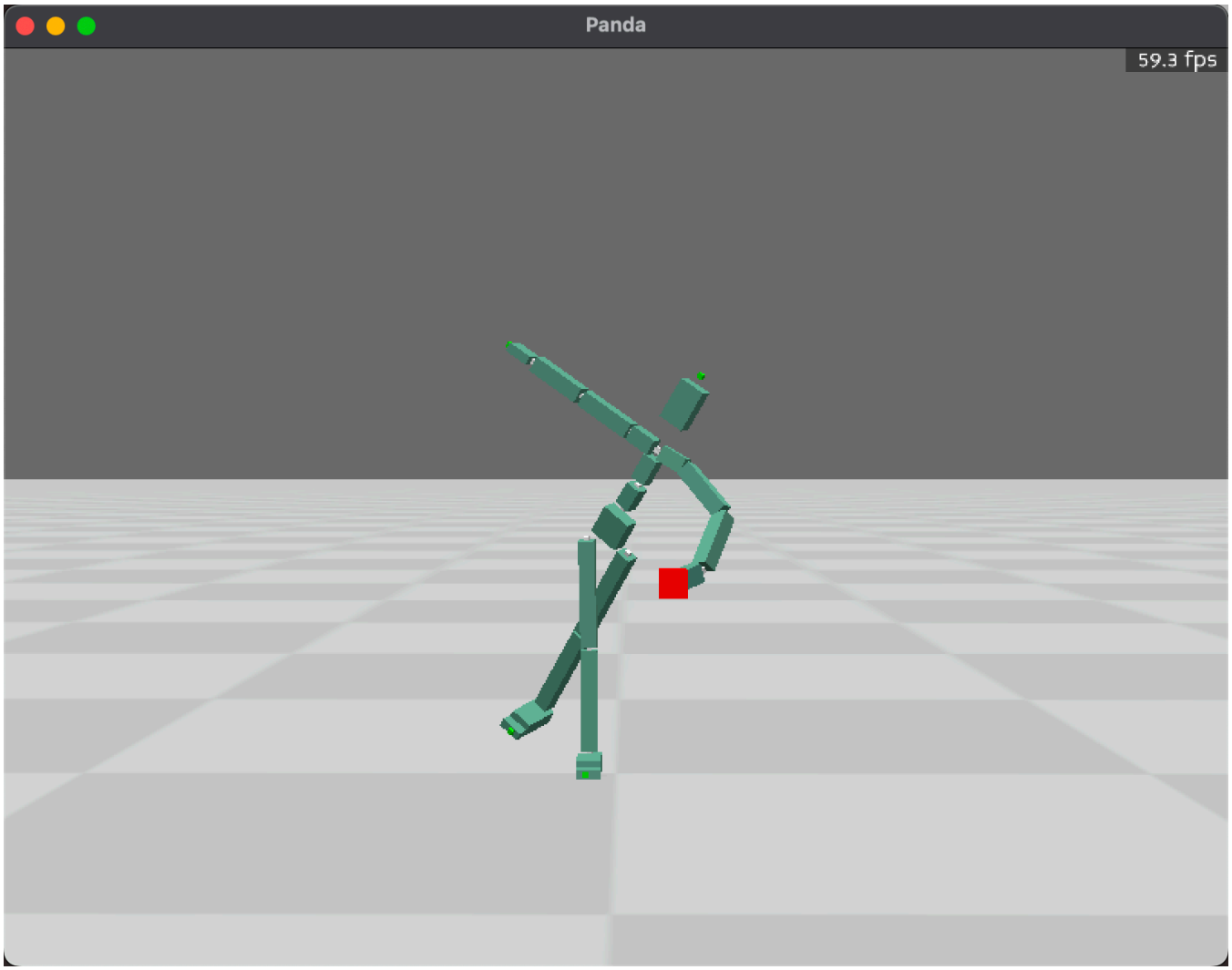
5

```
IK_interactive(viewer, np.array([0.5, 0.75, 0.5]), 'rToeJoint_end', 'lWrist_end')
```

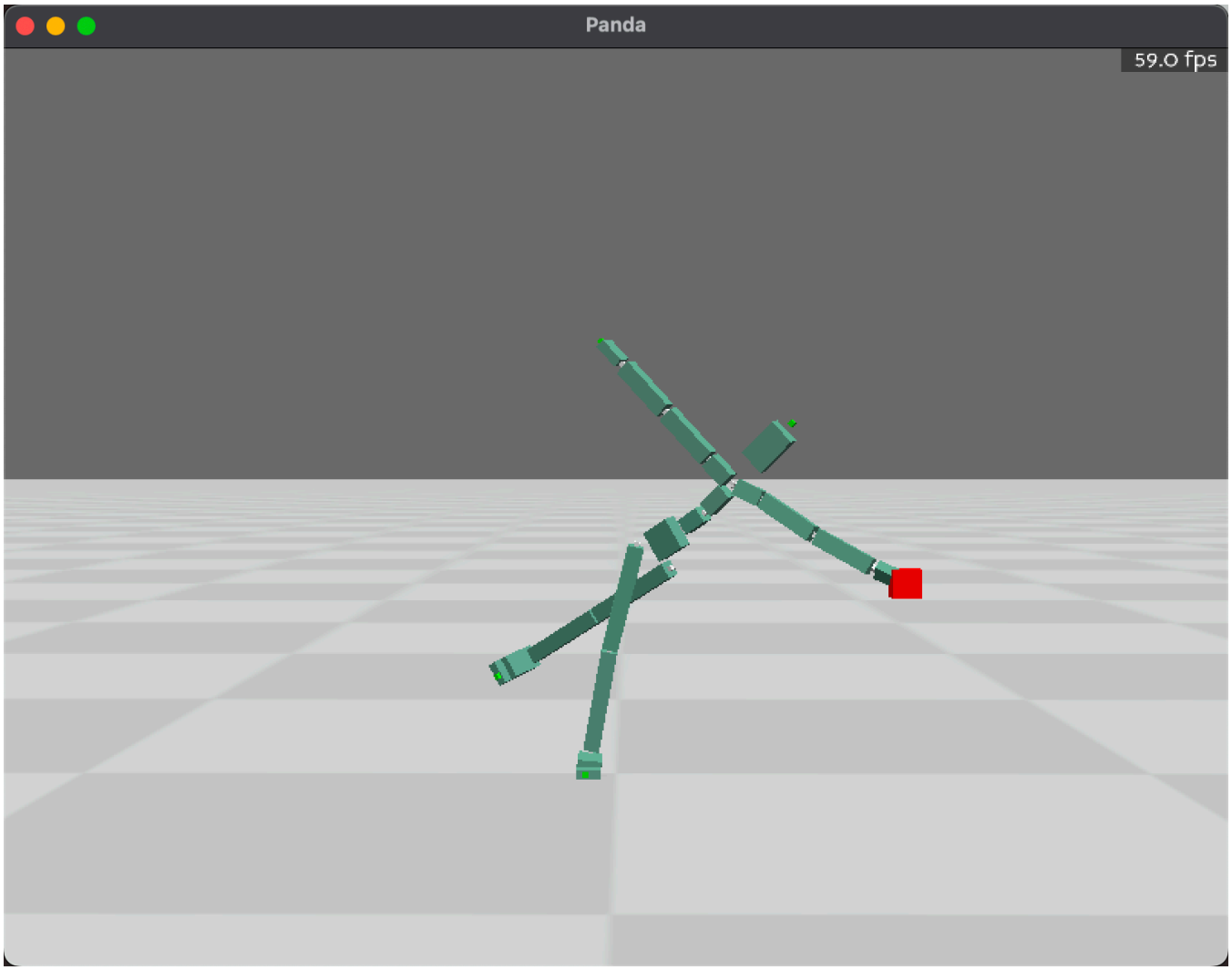
Initial position



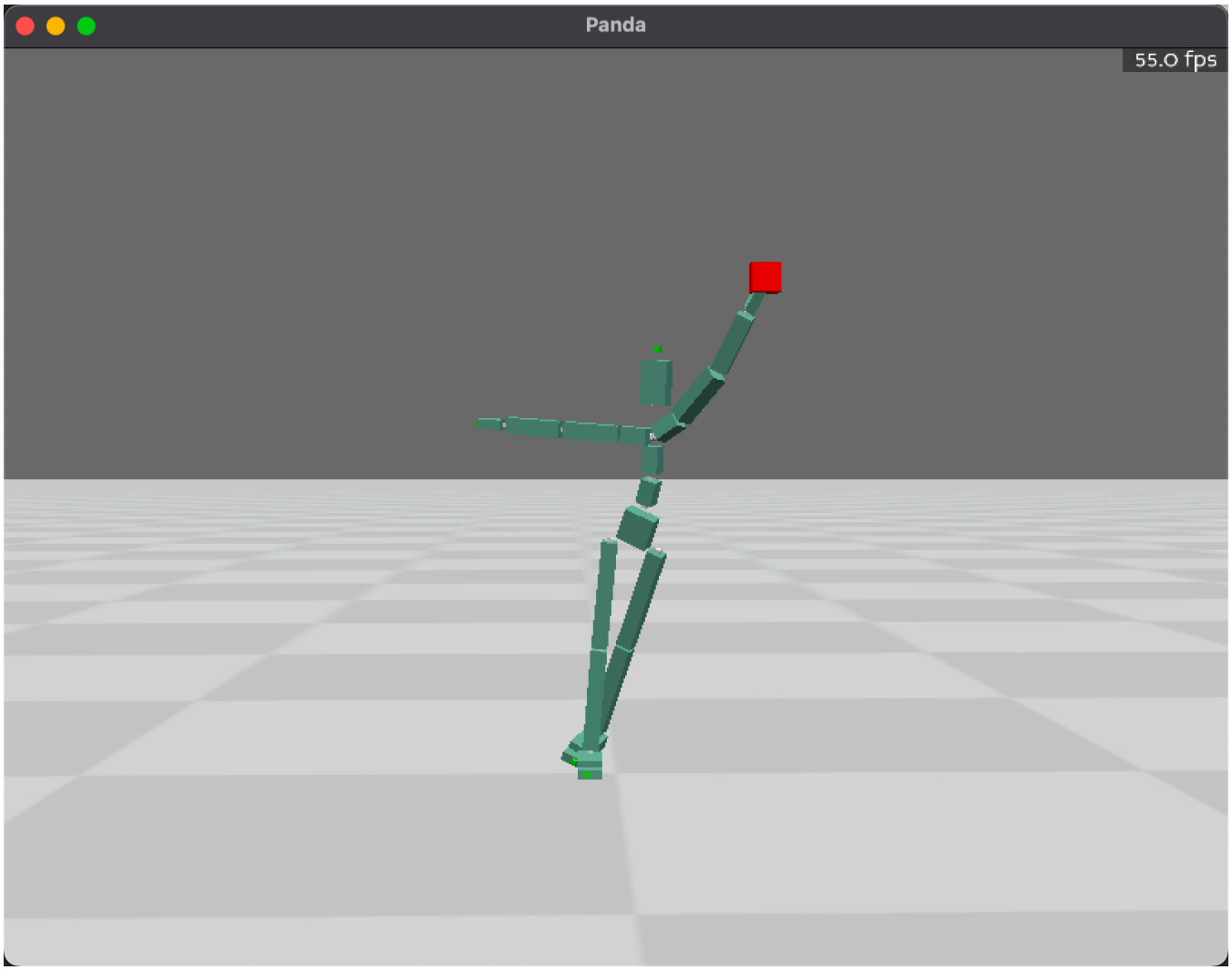
By pressing A from the starting position



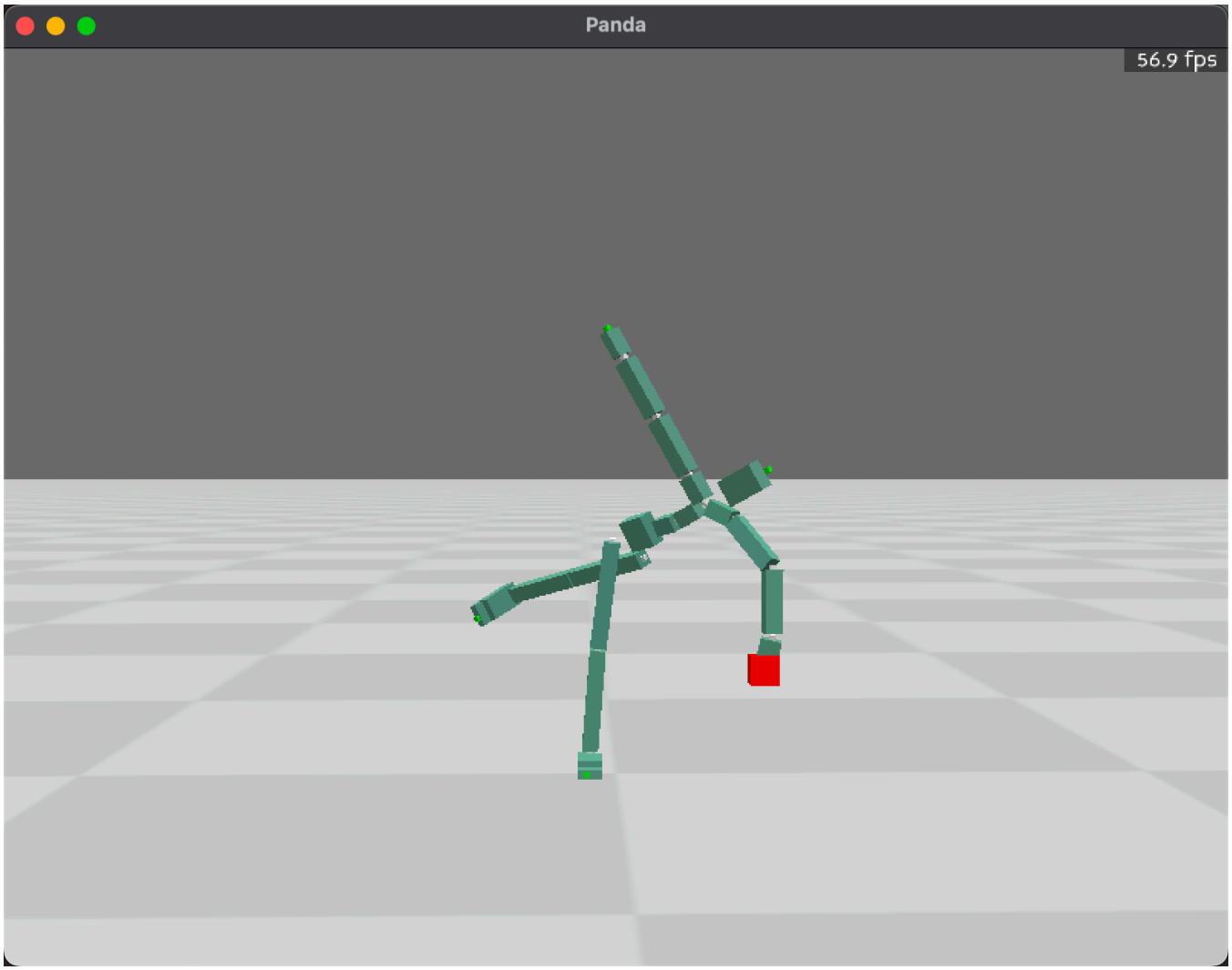
By pressing D from the starting position



By pressing W from the starting position



By pressing S from the starting position


Panda
56.9 fps