110550047 HW3

Introduction/Motivation

This project implement inverse kinematic mechanism. By holding and dragging the target ball to adjust bones' corresponding position and rotation, which is a crucial technique in the field of animation and computer graphics. This project simulate a standard humanoid bones with setting its body part to be the end bone that need to touch the target, and provide the real-time feedback on changing viewpoint and bone motion.

Fundamentals

Inverse Kinematics

In simpler terms, inverse kinematics is the process of determining the required joint angles or positions for a robot to reach a specific target location in space. It is the opposite of forward kinematics, which determines the position of an end-effector given the joint angles.

Inverse kinematics algorithms use complex mathematical calculations to find the required joint angles that will move the end-effector to its desired position. These calculations take into account the constraints and limitations of the robot's joints, such as their range of motion and physical limitations, as well as external factors like obstacles or target positions.

$$FK : f(\theta) = P$$
$$IK : \theta = f^{-1}(P)$$

DOF Jacobian

$$\theta = f^{-1}(P)$$
$$V = J(\theta)\hat{\theta}$$
$$\hat{\theta} = J^{-1}(\theta_k)V$$
$$J = \begin{bmatrix} rx_x & rx_y & rx_z & \dots \\ ry_x & ry_y & ry_z & \dots \\ rz_x & rz_y & rz_z & \dots \\ 0 & 0 & 0 & \end{bmatrix}$$

Implementation

SVD

```
// J(theta) = target
// theta = (J^+)*V
// SVD : J = U E V^T, J^+ = V E^+ U^T
Eigen::VectorXd pseudoInverseLinearSolver(const Eigen::Matrix4Xd&
Jacobian, const Eigen::Vector4d& target) {
Eigen::MatrixXd pinv =
 Jacobian.completeOrthogonalDecomposition().pseudoInverse();
    deltatheta = pinv *  target;
    xxxxxxxxxx Eigen::MatrixXd pinv =
 Jacobian.completeOrthogonalDecomposition().pseudoInverse();
 deltatheta = pinv *  target;
}
```

IK

```
// Get all influenced bones
while (current != start_bone && current != root_bone)
{
    bone_num++;
    boneList.emplace_back(current);
    current = current->parent;
}
if (current == root_bone) {
    current = start_bone;
    while (current != root_bone)
    {
        bone_num++;
        boneList.emplace_back(current);
        current = current->parent;
    }
}
bone_num++;
boneList.emplace_back(current);

// Call SVD
for (long long i = 0; i < bone_num; i++) {
            int startCol = i * 3;
            Eigen::Vector3d tempVector = (end_bone->end_position -
boneList[i]->start_position).head(3);
            Eigen::Vector3d head = boneList[i]-
>rotation.matrix().col(0).head(3);
            Eigen::Vector3d temp = head.cross(tempVector);
            Eigen::Vector4d result = Eigen::Vector4d(temp.x(), temp.y(),
temp.z(), 0);
```

```
            if (boneList[i]->dofrx)
                Jacobian.col(startCol) = result.normalized();

            head = boneList[i]->rotation.matrix().col(1).head(3);
            temp = head.cross(tempVector);
            result = Eigen::Vector4d(temp.x(), temp.y(), temp.z(), 0);
            if (boneList[i]->dofry)
                Jacobian.col(startCol + 1) = result.normalized();

            head = boneList[i]->rotation.matrix().col(2).head(3);
            temp = head.cross(tempVector);
            result = Eigen::Vector4d(temp.x(), temp.y(), temp.z(), 0);
            if (boneList[i]->dofrz)
                Jacobian.col(startCol + 2) = result.normalized();
        }
        Eigen::VectorXd deltatheta = step *
pseudoInverseLinearSolver(Jacobian, desiredVector);


// Update rotation and limit rotation
for (long long i = 0; i < bone_num; i++) {
            double newValue = posture.bone_rotations[boneList[i]-
>idx].x() + step * deltatheta(i * 3) * 180 / 3.14;
            if (newValue > boneList[i]->rxmin && newValue < boneList[i]-
>rxmax)
                posture.bone_rotations[boneList[i]->idx].x() = newValue;
            newValue = posture.bone_rotations[boneList[i]->idx].y() +
step * deltatheta(i * 3 + 1) * 180 / 3.14;
            if (newValue > boneList[i]->rymin && newValue < boneList[i]-
>rymax)
                posture.bone_rotations[boneList[i]->idx].y() = newValue;
            newValue = posture.bone_rotations[boneList[i]->idx].z() +
step * deltatheta(i * 3 + 2) * 180 / 3.14;
            if (newValue > boneList[i]->rzmin && newValue < boneList[i]-
>rzmax)
                posture.bone_rotations[boneList[i]->idx].z() = newValue;
        }
```

Result and Discussion

How different step and epsilon affect the result

Touch the target or not

Check norm of the vector between end_bone and target position. Return true means stable(touchable), return false means unstable(untouchable).

```
if ((end_bone->end_position - target_pos).norm() < epsilon)
   return true;
else {
   posture = original_posture;
   return false;
}
```

Least square solver

Use the easiest implementation called SVD.
The notion is that you use linear algebra feature to generate a pseudo inverse matrix J^{+1}.

$$J(\theta) = target$$
$$\theta = (J^{+})V$$
$$SVD : J = UEV^{T}, J^{+} = VE^{+}U^{T}$$

Bonus

Rotation Limitation

```
for (long long i = 0; i < bone_num; i++) {
            double newValue = posture.bone_rotations[boneList[i]-
>idx].x() + step * deltatheta(i * 3) * 180 / 3.14;
            if (newValue > boneList[i]->rxmin && newValue < boneList[i]-
>rxmax)
                posture.bone_rotations[boneList[i]->idx].x() = newValue;
            newValue = posture.bone_rotations[boneList[i]->idx].y() +
step * deltatheta(i * 3 + 1) * 180 / 3.14;
            if (newValue > boneList[i]->rymin && newValue < boneList[i]-
>rymax)
                posture.bone_rotations[boneList[i]->idx].y() = newValue;
            newValue = posture.bone_rotations[boneList[i]->idx].z() +
step * deltatheta(i * 3 + 2) * 180 / 3.14;
            if (newValue > boneList[i]->rzmin && newValue < boneList[i]-
>rzmax)
                posture.bone_rotations[boneList[i]->idx].z() = newValue;
        }
```

Stability Determination

```
if ((end_bone->end_position - target_pos).norm() < epsilon)
        return true;
    else {
        posture = original_posture;
        return false;
    }
```

Conclusion

In this project, we simulate a humanoid bone with inverse kinematic. This project also provide the function of setting end bone and start bone. It is really interesting that we can inspect different motions in order to reach the point. This method use a lot of math knowledge, another method of example-based method is much limited, but can be more efficient to use.