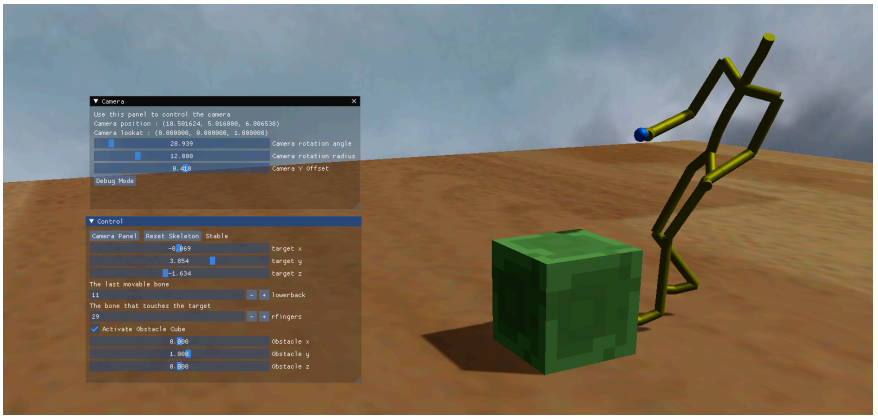
Computer Animation & Special Effects: HW2 Report

Author: 111550113 謝詠晴

1 Introduction

此次作業旨在實作一個基於Inverse Kinematics (IK) 的系統，使角色骨架可操控不同的 bones 來接觸目標物。使用者可以指定起始骨骼 (Start Bone) 和終端骨骼 (End Effector) 去觸碰目標物，且會同時考慮避開障礙物的限制。

在實作過程中，將運用 Forward Kinematics (FK) 來推算骨骼的位置，並透過數值方法求解 Jacobian 的逆矩陣以控制角色姿勢。另外，也加上考慮關節限制和是否與目標物成功碰觸的判斷。透過本次作業，能夠更加理解 FK 和 IK 的實務上差異與應用，並加強對動畫角色控制與姿態模擬的掌握。



2 Fundamentals

2.1 Forward Kinematics (FK)

- 定義：已知每個關節的旋轉角度，計算得出骨架的絕對位置和方向。
- 用途：
  (1) 在 IK 過程中，每次調整完 posture.bone_rotation (骨骼旋轉角度) 後，需要重新計算整個骨架的實際空間位置及和旋轉矩陣，這些資訊才能拿來計算 end effector 和更新 Jacobian 的 arm vectors。
  (2) 把局部的角度變化傳遞成整體空間動作的過程，例如：上臂彎曲 → 整條手臂連動。
- 方法：根據骨架的父子關係，先計算 root 節點的位置和旋轉，再由父節點推到子節點，直到遍歷完整條 Bone 串鏈。

2.2 Inverse Kinematics (IK)

- 定義：給定一個目標位置，反推每節 bone 應該旋轉的角度。

- 用途：根據使用者指定的目標位置 (球的位置)，計算角色骨架各個關節應該如何旋轉才能移動到該目標處。
- 方法：因為 IK 通常沒有閉式解，所以多透過數值近似的方法。用 Jacobian 矩陣計算每次微調的旋轉量 (deltatheta)，一步步調整，想辦法讓 end_pos 接近目標位置。

2.3 Jacobian Matrix

- 定義：
  (1) 描述「每個關節的小角度變化」和「末端位置的變化」之間的線性關係。
  (2) 每一列對應 x, y, z 軸的移動方向。
  (3) 每一欄對應到某個關節的一個自由度。
  (4) 依照以下公式計算每一欄要填入的值

$$\frac{\partial \mathbf{p}}{\partial \theta_i} = \mathbf{a}_i \times (\mathbf{p} - \mathbf{r}_i)$$

2.4 Pseudo-inverse and Least Squares Solution

- 目的：因為 Jacobian 通常不是方陣，不能直接計算反矩陣，所以要用最小平方法解。
- 方法：使用 SVD (Singular Value Decomposition) 來求解 pseudo-inverse。
- $J^{+}V = \dot{\theta}$ (J^{+} 是 J 的 pseudo-inverse)

2.5 Obstacle Avoidance

- 目的：角色在移動時若碰到障礙物，應該避開而非穿透。
- 方法：”Repulsion Mechanism”若骨骼中心靠近障礙物，對目標方向加一個反向推力。
- 公式：$repulse = normalize(obsDiff) \times (threshold - dist)$

3 Implementation

3.1 Forward Kinematics

(1) 我使用遞迴方式實作，從骨架的根節點開始依序遍歷整顆骨架樹。
(2) 首先，根節點會根據 posture 提供的位置和旋轉資訊進行初始化。
(3) 對於每個非根骨骼，先將其起始位置設為父節點的終點位置，並將父節點的全域旋轉、對齊矩陣和 local 旋轉角度相乘，得到此骨骼的全域旋轉。
(4) 最後，根據骨骼的方向和長度計算其中點位置。
(5) 遞迴呼叫子節點和 sibling 節點，以處理整個骨架。

```
if(bone->parent == nullptr) {
    bone->start_position = posture.bone_translations[bone->idx];
    bone->rotation = util::rotateDegreeZYX(posture.bone_rotations[bone->idx]);
} else{
    bone->start_position = bone->parent->end_position;
    bone->rotation = bone->parent->rotation * bone->rot_parent_current * util::rotateDegreeZYX(posture.bone_rotations[bone->idx]);
}
bone->end_position = bone->start_position + bone->rotation * (bone->dir.normalized() * bone->length);

if (bone->child) forwardSolver(posture, bone->child);
if (bone->sibling) forwardSolver(posture, bone->sibling);
```

3.2 pseudoInverseLinearSolver()

(1) 在給定 Jacobian 和目標位置向量的條件下，計算出一組角度變化量Δθ，使得Jacobian * Δθ 能夠近似 target。

(2) 使用了 SVD (Singular Value Decomposition) 來求解最小平方法問題，這邊我採用 Eigen 套件的 JacobiSVD 方法，並使用 .solve() 函式直接求得近似解。

```
Eigen::JacobiSVD<Eigen::MatrixXd> svd(Jacobian, Eigen::ComputeThinU | Eigen::ComputeThinV);
deltatheta = svd.solve(target);

return deltatheta;
```

3.3 Inverse Kinematics

(1) #TODO3-1：準備需要控制的 boneList

從目標骨骼 (end_bone) 開始，往上沿著骨架結構追溯直到起始骨骼 (start_bone)，將路徑上經過的所有骨骼依序儲存在 boneList 中。若無法從 end_bone 到達 start_bone，則會退回到 root 作為備用起點。

```
bool found = false;
while(current != nullptr) {
    boneList.push_back(current);
    if (current == start_bone) {
        found = true;
        break;
    }
    current = current->parent;
}

if (!found) {
    std::cout << "[Warning] Start bone is not reachable from end bone, using root instead." << std::endl;
}
bone_num = boneList.size();
std::reverse(boneList.begin(), boneList.end());
```

(2) #TODO3-2：建立 Jacobian 矩陣

在每次 IK 迭代中，會先呼叫 forwardSolver() 更新骨架的空間位置。接著，針對 boneList 中的每根骨骼，計算其此骨骼初始位置與末端骨骼間的 arm_vector，使用 cross product 計算旋轉對末端位置變化的影響，且針對具有自由度的軸才會填入 Jacobian 欄位的值。

```
for(size_t i = 0; i < bone_num; i++) {
    acclaim::Bone* bone = boneList[i];
    Eigen::Vector3d armVector = (end_bone->end_position - bone->start_position).head<3>();
    Eigen::Matrix3d rotation = bone->rotation.rotation().block<3, 3>(0, 0);

    if(bone->dofrx){
        Eigen::Vector3d axis_x = rotation.col(0);
        Jacobian.col(i * 3).head<3>() = axis_x.cross(armVector);
    }
    if(bone->dofry){
        Eigen::Vector3d axis_y = rotation.col(1);
        Jacobian.col(i * 3 + 1).head<3>() = axis_y.cross(armVector);
    }
    if(bone->dofrz){
        Eigen::Vector3d axis_z = rotation.col(2);
        Jacobian.col(i * 3 + 2).head<3>() = axis_z.cross(armVector);
    }
}
```

(3) #TODO3-3：處理避開障礙物的情形

計算每根骨骼中心點與障礙物中心點的距離 (obsDiff)，如果此距離小於設定的閾值，則計算一個指向遠離障礙物的排斥向量，並將此向量加到目前的目標向量。

(4) #TODO3-4：更新骨骼角度

使用 pseudoInverseLinearSolver() 函式求解 Jacobian 的 pseudo-inverse，計算出每個自由度應該調整的 deltaTheta。接著，將這些角度變化量依序套用到各個骨骼對應的本地旋轉角度上，這邊要注意要轉為 degree 為單位。

```
Eigen::VectorXd deltatheta = step * pseudoInverseLinearSolver(Jacobian, desiredVector);
```

```
for(size_t i = 0; i < bone_num; i++) {
    current = boneList[i];
    if(current->dofrx) {
        posture.bone_rotations[current->idx][0] += (deltatheta(i * 3) * 180.0) / EIGEN_PI;
```

(5) #TODO3-5：收斂判斷

每次更新後，檢查終端骨骼目前的位置與目標位置之間的距離。如果誤差低於設定的容許範圍，則認為 IK 解算成功並結束迭代，最後回傳一個布林值 stable，表示 IK 是否成功且結果穩定。

```
bool stable = (target_pos - end_bone->end_position).norm() < epsilon;
if (!stable) {
    posture = original_posture;
    forwardSolver(posture, root_bone);
}
return stable;
```

4 Result and Discussion

4.1 How different step and epsilon affect the result

(1) 當 step 設太小，每次更新的幅度只有一點點，會導致收斂非常緩慢，需要更多次的迭代才可接近目標。而 step 設太大，則可能在接近目標時出現震盪，無法精確留在目標位置。

(2) 經過我嘗試不同 step 發現步長大小也會影響角色手臂的最遠伸展範圍，原本預設 step=0.1，角色手臂在 x 方向是可以完全伸直，都還是 stable 的，但當 step = 0.01，角色手臂可伸展到的範圍變小了，無法完全伸直手臂。這可能是因為步長太小時，每次角度更新幅度有限，在有限的最大迭代次數內無法累積足夠的變化量來達到目標。

(3) epsilon 調大時，會發現角色改變姿勢的速度變快了，因為可以加速收斂，但是終端骨骼和目標球的位置可能會有微小誤差。

(4) 整體而言，在嘗試調整不同的步長 (0.01 ~ 0.8) 與收斂容許範圍 (1e-1 ~ 1e-4) 時，觀察到的結果在肉眼上並未產生明顯的差異。

4.2 Touch the target or not

在一開始實作時，我遇到了一個問題：角色的終端骨骼無法在初始狀態下就觸碰到目標球，當目標球的位置被調整到某些特定區域時，角色才有辦法跟著移動，但移動過程中的姿勢也顯得不自然且異常。經過檢查後，發現問題出在計算 arm_vector 時，誤用了 bone->end_position 作為計算基準，正確應該是使用 end_bone->end_position。修正這個細節後，角色姿態的穩定性明顯改善。這次經驗讓我體會到，即使是非常細微的錯誤，也可能對整體角色動作產生顯著影響。

4.3 Least square solver

在解最小平方法的部分，我選擇使用 SVD 來求解，主要是因為：

(1) 透過 Eigen 函式庫內建的 JacobiSVD 類別，可以直接進行奇異值分解並使用 .solve() 方法求得最小平方法解，實作上簡單方便，且不易出錯。

(2) SVD 的穩定性高，即使在數值不穩定的情形下也可以找到最佳近似解。

(3) 相比於 Normal Equation 或 QR 分解方法，SVD 在數值精度上表現更佳。

而 SVD 的主要缺點是其運算成本相對較高，特別是在處理大型系統，可能會影響效能。

5 Bonus

5.1 Stability Check

(實作方法在 Implementation 的#TODO3-5 有說明)

為了避免 IK 解失敗時角色骨架會產生異常的姿勢，當最終誤差大於設定的容許範圍，我會將角色的姿勢還原為最初儲存的原始姿勢 (original_posture)，並再呼叫 forwardSolver() 更新骨架的空間位置，避免出現手臂亂揮或異常扭曲的情形。

5.2 Joint Limit

使用 std::clamp() 將旋轉角度限制在骨骼所定義的最小值 (rxmin, rymin, rzmin) 與最大值 (rxmax, rymax, rzmax) 之間，防止角色在運動過程中出現不自然的動作。

```
if(current->dofrx) {
    posture.bone_rotations[current->idx][0] += (deltatheta(i * 3) * 180.0) / EIGEN_PI;
    posture.bone_rotations[current->idx][0] = std::clamp(posture.bone_rotations[current->idx][0], (double)current->rxmin, (double)current->rxmax);
}
```

6 Conclusion

起初由於過去沒有接觸過角色骨架與姿勢操作的經驗，因此預期這次作業會相對困難。然而在實際實作過程中，發現只要按照助教提供的 spec 說明、清楚的 TODO 指引，以及課堂簡報中的公式推導，逐步完成各個部分，實際上並沒有想像中困難。

不過，在實作中仍遇到一些需要特別注意的小細節，例如一開始角色的姿勢會出現異常的擺動或不穩定的旋轉，發現實作加上 bone limit 和 return stable 處理後，角色的動作就改善許多。