# Introduction to Computer Animation HW2 Report

109350008 張詠哲

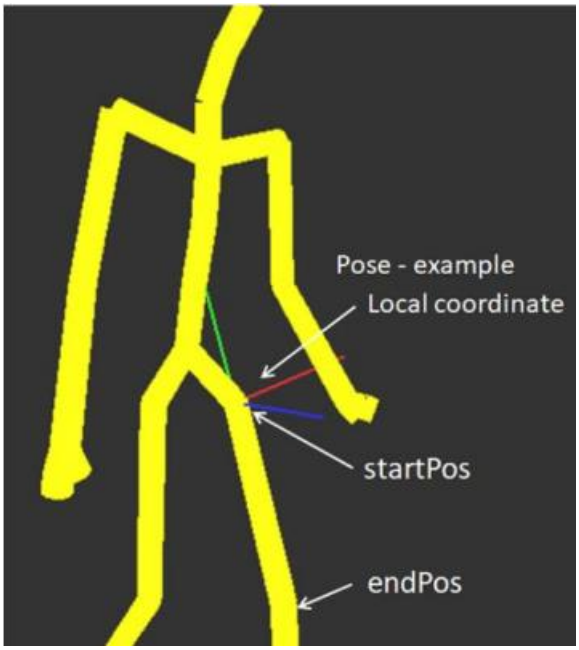
## • Introduction/Motivation

這次我們要實作的是模擬人體骨架(很多關節再加連接的骨頭)在跑步和投籃(丟球)來展示 Forward Kinematics 和 Time Warping。藉由現有的 Function 和 Eigen 提供的各種計算來實現旋轉矩陣和 Quaternion 的轉換和運算。我們會從骨骼的根部開始計算骨骼之間的起始位置、結束位置和旋轉，模擬整個人體形狀，以及其在運動中的姿勢。

## • Fundamentals

✧ **Local Coordinate:**

每個關節都有一個各自的坐標和自由度(角度)，是相對於上一個骨頭的末端點或某個關節定義的座標。示意圖:




✧ **Global Coordinate:**

Global Coordinate 就是指我們平常所常見的 3D 空間，用來描述整個場景的位置和姿態。示意圖:


✧ **Time Warping:**

是指對動畫時間軸上的時間進行壓縮或擴展。通過 Time warping，可以改變動畫中物體運動的速度和節奏，從而達到一些特殊的效果，例如慢動作、加速運動、時光倒流等等。

✧ **Forward Kinematics:**

是把物體的 local coordinate 轉到 global coordinate，意思就是說給定每個關節的角度後，可以算出各個關節的位置。但是上網也查到說，Forward Kinematics 有其局限性，像是無法很好地處理物體之間的碰撞和交互。這些問題需要使用更複雜的技術，例如 Inverse Kinematics 解決。

## • Implementation

✧ **ForwardSolver**

在這個部份我採用的是 DFS 去歷遍每個骨頭，接著去計算他們在 global coordinate 的起始位置、結束位置、旋轉。以下是一些 code 的解釋以及完整程式碼。

1. 將起始位置接到 parent 的結束位置

```
temp->start_position = temp->parent->end_position;
```

2. 將 asf 文件中的 child 對其 parent 的 rotation 與 amc 文件中的 rotation 相乘以獲得 global coordinate 的 child 相對其 parent 的 rotation。

$$ {}^{i+1}_{i}R = {}_{i}R_{asf} \cdot {}_{i}R_{amc} $$

```
Eigen::Quaternion rota =
    util::rotateDegreeZYX(posture.bone_rotations[temp->idx].x(),
                          posture.bone_rotations[temp->idx].y(),
                          posture.bone_rotations[temp->idx].z());

Eigen::Affine3d R = temp->rot_parent_current * rota;
```

3. 計算 local coordinate to global coordinate 的 quaternion

$$ {}_0^iR = {}_0^1R\,{}_1^2R \cdots {}_{i-1}^iR $$

```
for (acclaim::Bone* i = temp->parent; i != nullptr; i = i->parent) {
    Eigen::Quaternion i_rota =
        util::rotateDegreeZYX(posture.bone_rotations[i->idx].x(), posture.bone_rotations[i->idx].y(),
                              posture.bone_rotations[i->idx].z());

    R = i->rot_parent_current * i_rota * R;
}
```

4. 由於 asf 文件中的方向是單位向量，我們需要乘以長度才能得到正確的方向

$$ V_i = \hat{V}_i \cdot l_i $$

```
Eigen::Vector4d dir = temp->dir * temp->length;
```

5. 把剛剛計算的 rotation 乘上 dir 並加上起始位置以得到結束位置。

$$ {}_iT = {}_0^{i-1}RV_{i-1} + {}_{i-1}T $$

```
temp->end_position = R * dir + temp->start_position;
```

6. 如果是原點的話，因為 root 是沒有長度的，所以 root 的 start position 和 end position 會相同

```
else {
    temp->rotation = Eigen::Affine3d::Identity();
    temp->start_position = posture.bone_translations[temp->idx];
    temp->end_position = temp->start_position;
}
```

7. 全部的程式碼:

```
bone->start_position = Eigen::Vector4d::Zero();
bone->end_position = Eigen::Vector4d::Zero();
bone->rotation = Eigen::Matrix4d::Zero();

std::map<int, bool> vis;
std::stack<acclaim::Bone*> q;
vis[bone->idx] = true;
bone->start_position = posture.bone_translations[bone->idx];
bone->end_position = posture.bone_translations[bone->idx];
q.push(bone);

while (!q.empty()) {
    acclaim::Bone* temp = q.top();
    q.pop();
    if (temp->idx != 0) {
        temp->start_position = temp->parent->end_position;
        Eigen::Quaternion rota =
            util::rotateDegreeZYX(posture.bone_rotations[temp->idx].x(),
                                  posture.bone_rotations[temp->idx].y(),
                                  posture.bone_rotations[temp->idx].z());

        Eigen::Affine3d R = temp->rot_parent_current * rota;

        for (acclaim::Bone* i = temp->parent; i != nullptr; i = i->parent) {
            Eigen::Quaternion i_rota =
                util::rotateDegreeZYX(posture.bone_rotations[i->idx].x(), posture.bone_rotations[i->idx].y(),
                                      posture.bone_rotations[i->idx].z());

            R = i->rot_parent_current * i_rota * R;
        }

        Eigen::Vector4d dir = temp->dir * temp->length;
        temp->end_position = R * dir + temp->start_position;
        temp->rotation = R;

    }
```

```
    else {
        temp->rotation = Eigen::Affine3d::Identity();
        temp->start_position = posture.bone_translations[temp->idx];
        temp->end_position = temp->start_position;
    }

    for (acclaim::Bone* i = temp->child; i != nullptr; i = i->sibling) {
        if (vis.find(i->idx) == vis.end()) {
            vis[i->idx] = true;
            q.push(i);
        }
    }
}
```

✧ **Time Warping:**

實作這部分中的 low 和 high 是用來計算將 new frame 的每一幀 i 換算到 old frames 時會是在哪兩個 frame 中間。而 r 是$i \times \frac{old_frame}{new_frame} - low$，表示要內插在兩個 frame(low & high)中的地方。而以下是判斷式的解釋:

- 當 i == allframe_new - 1 時，表示到轉換後的最後一幀，此時直接將該幀設置為原始動作的最後一幀。
- 當 low == high 時，表示 r 為 0，不需做內插。直接將該幀設置為原始動作中對應的幀。
- 當 low != high 時，表示轉換後的當前幀落在原始動作的兩個相鄰幀之間，因此需要做插值。計算兩幀之間的平移變化使用線性插值，而球面線性插值(slerp)則是為了計算兩幀之間的旋轉變化。

以下為完整的程式碼:

```
new_posture.bone_translations[j] = Eigen::Vector4d::Zero();
new_posture.bone_rotations[j] = Eigen::Vector4d::Zero();

int low = floor(i * (double)allframe_old / (double)allframe_new);
int high = ceil((i * (double)allframe_old / (double)allframe_new));
double r = (i * (double)allframe_old / (double)allframe_new) - low;

if (i == allframe_new - 1) {
    new_posture.bone_rotations[j] = postures[allframe_old - 1].bone_rotations[j];
    new_posture.bone_translations[j] = postures[allframe_old - 1].bone_translations[j];
}

else if (low == high) {
    new_posture.bone_rotations[j] = postures[low].bone_rotations[j];
    new_posture.bone_translations[j] = postures[low].bone_translations[j];
}
```

```
    else {
        // translation
        new_posture.bone_translations[j] =
            r * postures[low].bone_translations[j] + ((double)1 - r) * postures[high].bone_translations[j];

        // rotation
        Eigen::Quaternion R1 =
            util::rotateDegreeZYX(postures[low].bone_rotations[j].z(), postures[low].bone_rotations[j].y(),
                                  postures[low].bone_rotations[j].x());

        Eigen::Quaternion R2 =
            util::rotateDegreeZYX(postures[high].bone_rotations[j].z(), postures[high].bone_rotations[j].y(),
                                  postures[high].bone_rotations[j].x());

        R1.slerp(r, R2);
        R1.normalize();
        Eigen::Vector3d temp = R1.toRotationMatrix().eulerAngles(2, 1, 0) * ((double)180 / pi);

        new_posture.bone_rotations[j][0] = temp[0];
        new_posture.bone_rotations[j][1] = temp[1];
        new_posture.bone_rotations[j][2] = temp[2];
    }
}
new_postures.push_back(new_posture);
```

## • Result & Discussion

### ✧ Result:

ForwardSolver 中我們利用 amc file 和 asf file 中的資訊去計算每個關節的旋轉角度，以及其在世界坐標系下的位置。

TimeWarper 中我們計算了對時間軸上的時間進行壓縮或擴展。並對平移變化做線性插值，對旋轉做球面線性插值(slerp)。

✧ **Problems Encountered:**

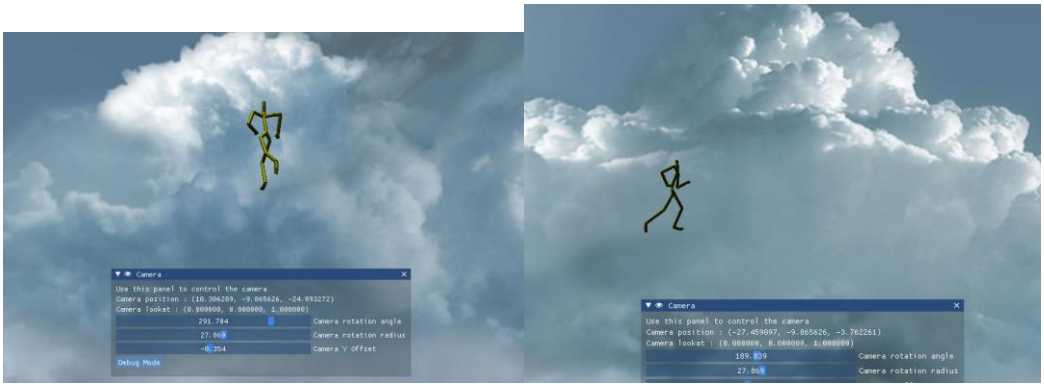
1. 起初剛開始的時候要搞懂那些旋轉式怎麼操作的花了一些功夫，常常一進到執行畫面看到骨架歪七扭八，就直接跳出來不跑了。還好經過多方嘗試以及理解後，最後大致上有實作出來。歪七扭八大概像這樣:


2. 再來是在做 timewarper 部分時，剛開始寫的時候很常跳出未載入符號檔，雖然最後不太確定是為什麼會這樣，但似乎是因為有些地方計算錯誤才會這樣。還有就是不知道為甚麼在高幀數的時候(大約>550)，球就會丟不出去。還好看到討論區有人提出了解法: 將 ball.cpp 中的第 48 行的>=改成>就可以了。有可能是因為差距太小以至於判定丟出去的速率為 0。

```
45
46      if (catched) {
47
48          if (time > thrown_frame && !thrown) {
49              thrown = true;
50              thrown_pos = center;
51              throw_velocity = thrown_pos - pre_pos;
52          }
```

## • Bonus

我還是先去材質包那裡看看有沒有甚麼酷酷的材質包可以選，發現好像只能換天空的 skin，因此我就順便換了一下。大概長這樣:




## • Conclusion

此次要實作的部分有 2 個 ForwardSolver 以及 TimeWarper 來模擬人體骨架在跑步和投籃(丟球)。我們從骨骼的根部開始計算骨骼之間的起始位置、結束位置和旋轉，模擬整個人體形狀，以及其在運動中的姿勢。

在 ForwardSolver 中我們利用 amc file 和 asf file 中的資訊去計算每個關節的旋轉角度，以及其在世界坐標系下的位置。TimeWarper 中我們計算了對時間軸上的時間進行壓縮或擴展。並對平移變化做線性插值，對旋轉做球面線性插值(slerp)。

也對於找到了為什麼在高幀數下球丟不出去的大致原因(有可能是因為差距太小以至於判定丟出去的速率為 0)，也找到了解法: 將 ball.cpp 中的第 48 行的 >=改成>。

因為有修改 ball.cpp(丟球)以及 main.cpp(加分)檔的關係，因此也會額外在繳交作業的地方附上這兩個檔案。