2020 CA HW2 Report

一、 作業目標

本次作業目的在於初步了解人物動畫的製作流程，而我們只需要實作其中的兩部分，分別為 Forward Kinematics 的 Forward Solver 與 Time Warper。

二、 環境

IDE: Visual studio 2017 / 2019
Platform: Windows
Graphics API: OpenGL
OpenGL Loading Library: glad2
OpenGL Toolkit: glfw
UI Library: dear imgui
Math Library: Eigen

三、 作業內容

1. 實作 Forward Solver
   將 ASF File 的骨骼架構與 AMC File 的動作模型連接起來。
2. 實作 Time Warper
   Translation 部分使用 linear interpolation，Rotation 部分使用 spherical linear interpolation。

四、 實作

1. 實作 Forward Solver

此 Function 最需要注意的就是，Rotation 究竟該如何轉換。

每個關節點都有各自的 Rotation，每個 Bone 也都有各自原本的方向，並且每根 Bone 的座標系都還會受到 Parent Bone 所影響。

非實作部分：

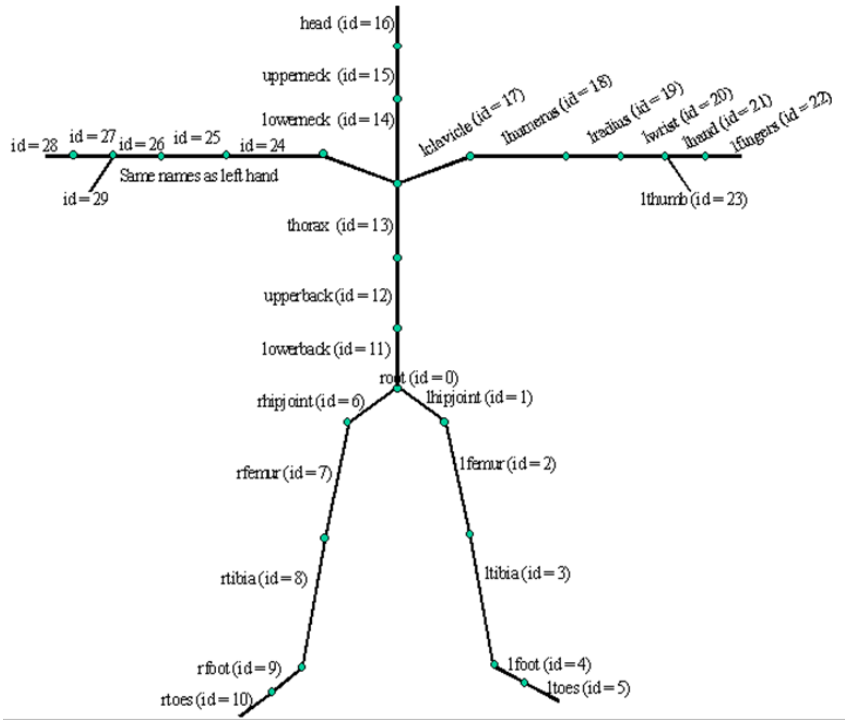
ASF File 所讀取到的 Skeleton 模型主要是以 Tree 來呈現，可以從 Root node 一路 Traverse 到每一根骨頭。並且還有一個額外的 Vector 可以快速得到任意一根 Bone 的參數。

Tree 的建立方式是：

Root Node 的建立，沒有 Parent。

讀取到一根新的骨頭，會依照該 ASF File 找到他的 Parent Node 的 ID，設定該 Parent Pointer。

若要連接新的 Child Node，會先看他自己有沒有已經有 Child Node，若沒有則將之設定為 Child。若有，則循著 Child 設定 Sibling。



AMC File 讀取到的 Motion 檔案其實只包含了兩個東西，一個是當前描述的骨頭名字，另外會有最高達到六個值，代表該 Bone 的六維參數（移動三軸、轉動三軸），此六軸都是計算從骨架原本的狀態到該動作的新的狀態的 delta 參數，並且我們並不在乎先前的狀態究竟為何，因為都會被記錄在各根 bone 中，只需要直接提取值即可。

由於在 ASF File 中已經事先設定好所有骨頭的 dof（Degree of freedom），因此在讀取 AMC File 時可以避免讀取到一大堆用不到的數據，兩者基本上必須具有高度的配合性。

實作部分：

$$ {}_{i+1}^{\square}T = {}_{0}^{i}RV_i + {}_{i}^{\square}T $$

${}_{i}T$ 代表第 i 個關節點的世界座標，也對應到該關節點前一根骨頭的終點，與後一根骨頭的起點。

${}_{j}^{i}R$ 代表從 i 座標系轉換到 j 座標系的轉換矩陣。

V_i 代表在 i 座標系中的任意向量，在此次實作中代表骨頭的起點到終點的（local）向量。

此公式目的在計算第 i 根骨頭的終點。

${}_{i}T$ 與 ${}_{i+1}T$可以理解為第 i 根骨頭的起點與第 i 根骨頭的終點（同樣也是第 $i+1$ 根骨頭的起點）。

$$ {}_{0}^{i}R = {}_{0}^{1}R\,{}_{1}^{2}R \cdots {}_{i-1}^{i}R $$

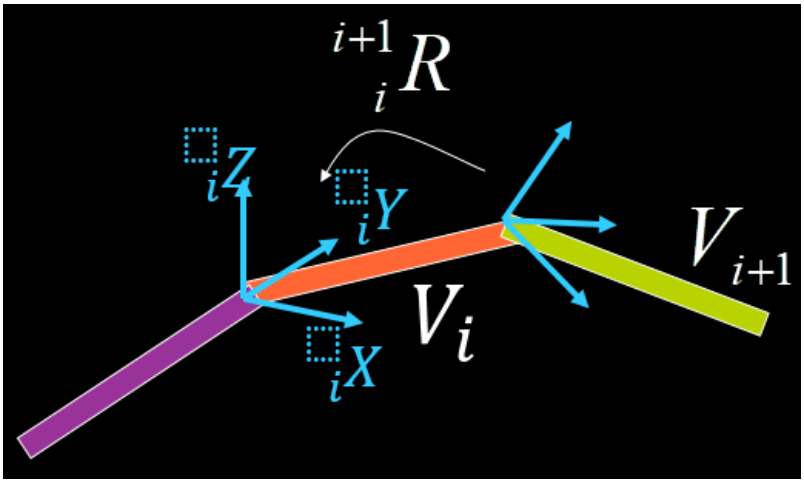
可以得知，要想知道第 i 根骨頭（從 local root 到該骨頭的起點）的轉換矩陣，需要累積他的所有 parent 的轉換矩陣。

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

而每一根骨頭（本身座標系）的旋轉矩陣，來自於從檔案直接讀取進來的 R_{asf} 與 R_{amc}。結合所有 parent 的轉換矩陣，可以得到該骨頭從 root 到算出終點的整體的旋轉矩陣，並且此旋轉矩陣也需要提供給下一根骨頭所用。

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

在本次實作中，V_i 代表的是骨頭本身起點到終點的 local 向量，由單位向量與其長度所得出。



Code 解析：

Start position 的計算

雖然先前描述的公式中沒有提到任何移動矩陣 translation matrix，但可以很容易理解移動座標必須被施加在起點上，畢竟只有在起點移動過後並到了定點後，才能以此點為軸開始做各式各樣的旋轉，以得到終點，因此
若沒有 parent bone，起點即為 translation 本身；若是有 parent bone 則起點從 parent bone 的終點開始計算，並加上 translation offset。

```
bone->start_position = posture.bone_translations[bone_idx];
if (parentBone != nullptr) {
    bone->start_position = parentBone->end_position + posture.bone_translations[bone_idx];
}
```

Rotation 的計算

此值的目的在於，一方面讓自身骨頭可以直接從 local 坐標系旋轉到 globla 座標系，另一方面讓 child bone 可以拿到，以達到旋轉矩陣疊加的目的。

與起點的計算相同，若沒有 parent bone，則旋轉直接設定為預設的 rot_parent_current（Rotation Parent to Current）乘上 delta rotation；若有 parent bone 則前面需要再乘上 parent 的 rotation。

```
bone->rotation =
    bone->rot_parent_current * Eigen::Affine3d(util::rotateDegreeZYX(posture.bone_rotations[bone_idx]));
if (parentBone != nullptr) {
    bone->rotation = parentBone->rotation * bone->rotation;
}
```

在此處 rot_parent_current 即為先前公式中的 R_{asf}

而從 posture 中取得的 delta rotation 即為R_{amc}

End positon 的計算

有了 rotation 之後，終點的計算就相當簡單了，只需要將該旋轉矩陣，乘上先前公式中的 V_i ，再加上自身的起點，就得到終點的世界座標。而 V_i 也相當容易：

```
bone->end_position = bone->start_position + bone->rotation * (bone->dir * bone->length);
```

骨頭原本的方向為 dir，長度為 length，相乘即可。

Traverse 所有骨頭

此部分未方便撰寫，我使用的是 DFS，並且 Traverse 順序為 self -> sibling -> child，以此來保證在抵達所有骨頭之時，他們的 parentBone -> rotaion 都已經計算完成。

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

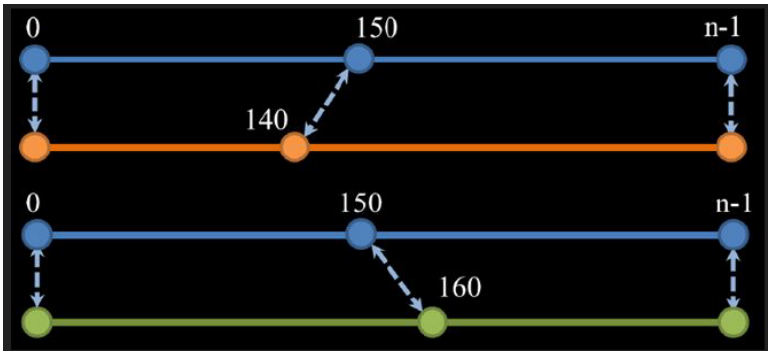
而 Recursive 的 terminate condition 就是當自身其實是個 null pointer 時。

```
if (bone == nullptr) { return; }
```

2. **實作 Time Warper**

Time Warper 的重點在於，了解如何實作 frame 與 frame 之間的內插。
本次實作需要分別做 linear interpolation 與 spherical linear interpolation。

Linear interpolation：



以此圖為例，藍色都是原本的時間線（keyframes）
上半部分： 新的 keyframe 每一格對應到舊的 keyframe 的 $\frac{150}{140} * \mathrm{i}$。
將原本的前 150 號 keyframe 對齊新的前 140 號 keyframe，由於無法完全對齊，因此從原本的 keyframe 使用內插法來得到新的 keyframe 的動作。

下半部分： 新的 keyframe 每一格對應到舊的 keyframe 的 $\frac{150}{160} * i$。
從這裡可以得到一個通式，新的第 i 個 keyframe 會得到舊的第 $\frac{old\_keyframe}{new\_keyframe} * i$ 個 keyframe。

```
double ratio = double(keyframe_old) / double(keyframe_new);
```

接下來就是取得舊的 keyframe 的前後兩個整數 keyframe 來做內插。

```
double oldKeyframeNum = ratio * i;
int lowerBound = oldKeyframeNum;
int upperBound = lowerBound + 1;
double ratioBetweenBoundary = oldKeyframeNum - double(lowerBound);
```

```
Eigen::Vector4d translationDifference =
    postures[upperBound].bone_translations[j] - postures[lowerBound].bone_translations[j];
new_postures[i].bone_translations[j] =
    postures[lowerBound].bone_translations[j] + translationDifference * ratioBetweenBoundary;
```

若範圍已經超出需要做內插的部分，則新的第 i 個 keyframe 全部取整數位移後的舊的 keyframe。例如從第一張圖來說，從新的第 160 號 keyframe 之後，全部對齊原本的第 $i - 10$ 個 keyframe。

```
if (i > keyframe_new) {
    oldKeyframeNum = i - difference;
}
```

```
if (ratioBetweenBoundary == 0) {
    new_postures[i].bone_translations[j] = postures[oldKeyframeNum].bone_translations[j];
    continue;
}
```

Spherical linear interpolation：（簡稱 Slerp）
此內插法專門使用在旋轉矩陣上，並且使用的對象為四元數。
四元數（Quaternion）包含四個元素，分別為一個實數與三個虛數。三個虛數可以被用來當成空間中的轉動，對應三維中的三個轉軸 XYZ。
在本次實作中，由於原本的 bone_rotation 儲存的都只有角度資訊（單位為度），因此需要把他們轉換成四元數才能使用 Slerp。

```
Eigen::Quaterniond q1;
q1 = Eigen::AngleAxisd(postures[lowerBound].bone_rotations[j][0] * M_PI / 180, Eigen::Vector3d::UnitX()) *
     Eigen::AngleAxisd(postures[lowerBound].bone_rotations[j][1] * M_PI / 180, Eigen::Vector3d::UnitY()) *
     Eigen::AngleAxisd(postures[lowerBound].bone_rotations[j][2] * M_PI / 180, Eigen::Vector3d::UnitZ());

Eigen::Quaterniond q2;
q2 = Eigen::AngleAxisd(postures[upperBound].bone_rotations[j][0] * M_PI / 180, Eigen::Vector3d::UnitX()) *
     Eigen::AngleAxisd(postures[upperBound].bone_rotations[j][1] * M_PI / 180, Eigen::Vector3d::UnitY()) *
     Eigen::AngleAxisd(postures[upperBound].bone_rotations[j][2] * M_PI / 180, Eigen::Vector3d::UnitZ());
```

在 Slerp 之後，還需要將四元數轉換回角度，並且單位也同樣是度，幸好 Eigen::Quaternion 中都有可以使用的函式可以直接使用，但是需要先轉換成旋轉矩陣，再轉換成角度，並且轉換成角度的函式回傳的單位是徑度，因此又需要再手動轉回度。

```
Eigen::Vector3d eulerAngles = q1.slerp(ratioBetweenBoundary, q2).normalized().toRotationMatrix().eulerAngles(0, 1, 2);
new_postures[i].bone_rotations[j] = Eigen::Vector4d(
    eulerAngles[0] * 180 / M_PI, eulerAngles[1] * 180 / M_PI, eulerAngles[2] * 180 / M_PI, 0);
```

並且與 Translation 一樣，在所計算的 frame 值已經超出需要內插的範圍之後，也是直接取 difference 後的原本的 frame。

```
if (ratioBetweenBoundary == 0) {
    new_postures[i].bone_translations[j] = postures[oldKeyframeNum].bone_translations[j];
    continue;
}
```

而整體的 Time Warp 結構，只需要在決定 frame_old 與 frame_new 後，逐個 frame 逐個 bone 慢慢內插即可。

```
for (int i = 0; i < total_frames; ++i) {

    for (int j = 0; j < total_bones; ++j) {
```

五、 Bonus

無

六、 難點（疑問）與解法

本次實作中，難點有二：

1. 並不知道 Affine3d、Vector4d、Quaterniond 的轉換，原因在於並不了解他們裡面究竟儲存了什麼樣的值。例如一開始根本不知道 bone_rotations 中所儲存的 Vector4d 其實是角度，並且單位是度。Time Warp 中需要將 Vector4d 中的值轉換成 Quaternion，算完 slerp 又要轉換回來，實在是很難入門。但是在入門過後，一切都變得簡單許多了，比起自己建立 Mat44 然後手動計算角度與四元數，有這些方便的 Library 的我們實在是有夠幸福。
2. 在計算 Forward Solver 的 Rotation 與 End_position 時候，一直不知道他們之間的關係，以及 local 到 global 的轉換實在是搞到暈頭轉向都還弄不懂。直到一步步拆解後，開始覺得對於整個結構有更清晰的感覺，沒想到做出來的 Motion 還是錯的，十分絕望。最後發現，原來 rot_parent_current 並不是一個需要改動的值，甚至他也必須跟著乘上去，才終於將這個花了兩天都寫不出來的小小 function 給解決掉。

七、 結語

再次體驗了走這條路可能會碰到一大堆線性代數、微分方程。對於大一就沒學好線性代數的我來說，還是有點令人恐懼。但每次做出成品都會覺得自己又離數學物理的真理又更進一步了，非常開心。實在是很喜歡這種將現實生活中的東西，在電腦上實作出來的手感，希望第三次作業與期末都能更加得心應手。