CASE HW2

109550135 范恩宇

1. Introduction :

這次作業是要模擬人形生物跑步與投球的動畫，分別藉由要實作的 Forward Kinematics 和 Time Warping 實現。

2. Fundamentals :

- Forward Kinematics

  骨架中某個 Bone 做動作時，除了其 Translation 和 Rotation 會改變，它末端延伸到的 Bone 也會一起有所變化。過程先讀出 Root 的 Translation 和 Rotation，接著將它的末端接到下一個 Bone 的初始端然後算其 Translation 和 Rotation，最後 traverse 各個 Bone 並且做同前述的事情。

- Time warping

  即為時間的縮放，把原先的某個 Frame 對應到新的 Frame，但整體時間不變，使得新的 Frame 之前會加速播放然後減速，或是減速播放然後加速

- Local Coordinate

  僅直接看 Root Bone 的座標，以前者為準的每根 Bone 都有自己的 Local Coordinate。初始端為原點，當我們 translate 或 rotate 各個 Bone，會同時改變它們的 local coordinate。Local Coordinate 相較 Global Coordinate 更方便使用者操作不同物件，而不需考慮其他物體對當前操作的對象有何影響。

- Global Coordinate

  整個場景的座標系，由於每根 Bone 都在此座標系的特定位置，因此比 Local Coordinate 更容易讓我們看出它們之間的相對位置。然而我們沒辦法直接由 Global Coordinate 判斷某位置的是哪根 Bone，需要先由 Local Coordinate 算出 Bone 交互作用產生的動作，再把整個骨架貼到 Global Coordinate。

3. Implementation :

- forwardSolver

  先定好 Root Bone 的頭尾端以及 Rotation 值，Root Bone 因為沒有 parent，Rotation 不用像其他 Bone 多乘上 parent Rotation。再來，開一個長度 31 的 bool array(看程式執行的視窗說共 31 個 Bone)，接下來透過 BFS 來 traverse 所有的 Bone。

```
bool visited[31] = {};
// root
bone->start_position = posture.bone_translations[0];

bone->rotation = bone->rot_parent_current;
bone->rotation = util::rotateDegreeZYX(posture.bone_rotations[bone->idx]);

bone->end_position = bone->start_position;
bone->end_position += bone->rotation * (bone->dir.normalized() * bone->length)
//cout<<"x:"<<posture.bone_rotations[bone->idx].x()<<" y:"<<posture.bone_rotat
visited[0] = true;

BFS(posture, bone->child, visited);
```

Traverse 全骨架的過程中，若 Bone 的 sibling 尚未被標記為 visited 則向那 traverse。最後也檢查 Bone 的 child 是否有漏跑，有則一樣開跑 BFS。

每個 child 的起點都會是 parent 的終點，而且 Rotation 要多乘上 parent 的 delta rotation，從而帶動相連的 Bone 產生運動。Bone 之終點則如同 Root Bone 的算法，將 Rotation 值乘上該 Bone 的對應方向與長度。

```
void BFS(const acclaim::Posture& posture, acclaim::Bone* bone, bool visited[]){
    visited[bone->idx] = true;
    bone->start_position = bone->parent->end_position;

    bone->rotation = bone->parent->rotation * bone->rot_parent_current;
    bone->rotation *= util::rotateDegreeZYX(posture.bone_rotations[bone->idx]);

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

    acclaim::Bone* tmp = bone->sibling;
    while (tmp != nullptr){
        if (!visited[tmp->idx]){
            BFS(posture, tmp, visited);
        }
        tmp = tmp->sibling;
    }

    if (bone->child != nullptr) {
        if (!visited[bone->child->idx]) {
            BFS(posture, bone->child, visited);
        }
    }
}
```

- timeWarper

首先定義需要的各個變數，scaling 是新 Frame 對應舊 Frame 的比例，floor/top 為插值的下/上界，ratio 則是插值的比例。新的 Translation 藉由上下界的 Translation (type=Vector4d) 經 Linear Interpolation 得出；新 Rotation 則靠著上下界的 Rotation (type=Quaterniond) 經 Spherical Linear Interpolation 得出。最後將新的 Translation 和 Rotation 都 assign 至 new_poseture 的對應值，之後前者會被 push 進 new_postures 的 vector，完成更新。

```
double scaling = double(allframe_old) / double(allframe_new);
int floor = (i * scaling >= total_frames - 1) ? total_frames - 1 : i * scaling;
int top = (floor == total_frames - 1) ? floor : floor + 1;
double ratio = i * scaling - floor;

// translation
Eigen::Vector4d translation = postures[floor].bone_translations[j];
Eigen::Vector4d translation_new = postures[top].bone_translations[j];

// rotation
Eigen::Quaterniond rotation;
rotation.w() = postures[floor].bone_rotations[j].w();
rotation.x() = postures[floor].bone_rotations[j].x();
rotation.y() = postures[floor].bone_rotations[j].y();
rotation.z() = postures[floor].bone_rotations[j].z();

Eigen::Quaterniond rotation_new;
rotation_new.w() = postures[top].bone_rotations[j].w();
rotation_new.x() = postures[top].bone_rotations[j].x();
rotation_new.y() = postures[top].bone_rotations[j].y();
rotation_new.z() = postures[top].bone_rotations[j].z();

// final input
Eigen::Vector4d fin_translation = translation + (translation_new - translation) * ratio;
new_poseture.bone_translations[j] = fin_translation;
Eigen::Quaterniond fin_rotation = rotation.slerp(ratio, rotation_new);//取1-ratio會打回臉上
new_poseture.bone_rotations[j] = Eigen::Vector4d(fin_rotation.x(), fin_rotation.y(), fin_rotation.z(), fin_rotation.w());
```

4. Result and Discussion :

實作過程主要遇過以下問題：

1. 做 Forward Kinematics 的時候我一開始忘記讓 Root Bone 以外的 Bone Rotation 乘上 parent 的 rotation，以致最後骨架用一個很娘娘腔的姿勢在跑步，其實看起來蠻好笑的。後來是仔細把 bone.h 看過後

才想到要乘上 parent Rotation，並得到應該是正確的結果。

2. 算 scaling factor 時由於預設的分子分母都是 int，需要將它們都改為 double 才能得到需要的結果。
3. 插值的上下界，似乎都需要依據 scaling factor、total frame、floor 的相對大小來選擇，原先我直接給定值時都會讓程式無法執行。
4. 處理 Time Warping 時，一度遇上調某些 Post Keyframe 值時，球會無法丟出並打回臉上的情況。後來才想到 Translation 好像不該跟 Rotation 一樣用 Quaterniond 的資料型態，而且我 slerp 用的插值比例反了，調整完就看起來正常。

5. Conclusion :

這次作業除了要實作的量比較少，用到的公式與程式碼結構都沒上次複雜，尤其減少了許多「尋找有沒有某變數可用」、「判斷某變數用在那裡」、「猜測某變數怎麼用」的時間，整體輕鬆很多。而且綜合討論區、朋友與自己所遇到的問題，感覺基本上大同小異，可以順利快速排雷並完成作業。