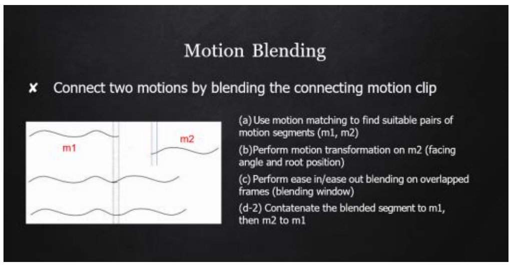
1. Introduction

本報告旨在探討 motiongraph 和 transform & blend 的實作，其中包含一個片段的尾部及頭部的對其，動作的變換和混和來確保其能平滑連接，以及繪製 motiongraph 邊的決定權重的方法，我們用 C++和 Visual Studio 來實作，分析了這些功能與動畫流暢性的相關連性，主要用以下資料來實作。



Motion Blending

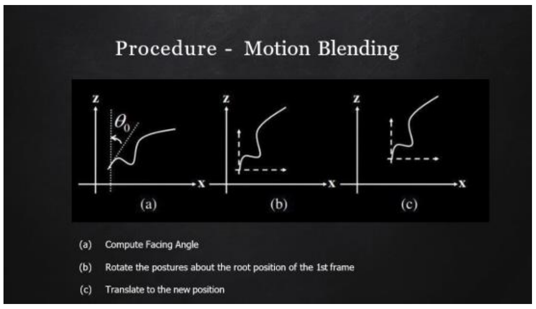
✘ Connect two motions by blending the connecting motion clip

(a) Use motion matching to find suitable pairs of motion segments (m1, m2)

(b)Perform motion transformation on m2 (facing angle and root position)

(c) Perform ease in/ease out blending on overlapped frames (blending window)

(d-2) Contatenate the blended segment to m1, then m2 to m1

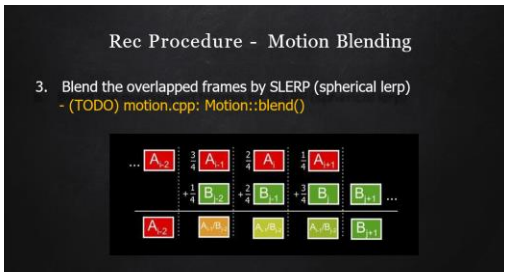


Procedure - Motion Blending

(a) Compute Facing Angle

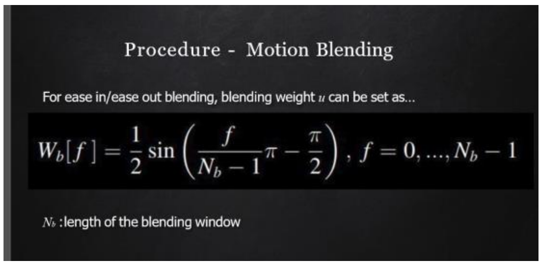
(b) Rotate the postures about the root position of the 1st frame

(c) Translate to the new position



Rec Procedure - Motion Blending

3. Blend the overlapped frames by SLERP (spherical lerp)
   - (TODO) motion.cpp: Motion::blend()

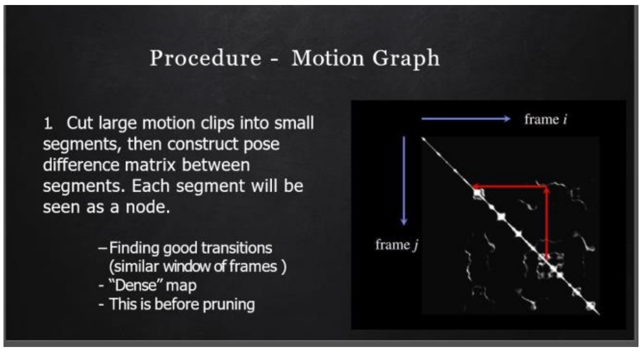


Procedure - Motion Blending

For ease in/ease out blending, blending weight u can be set as...

$$W_b[f] = \frac{1}{2}\sin\left(\frac{f}{N_b-1}\pi - \frac{\pi}{2}\right),\ f = 0, \ldots, N_b - 1$$

N_b :length of the blending window



Procedure - Motion Graph

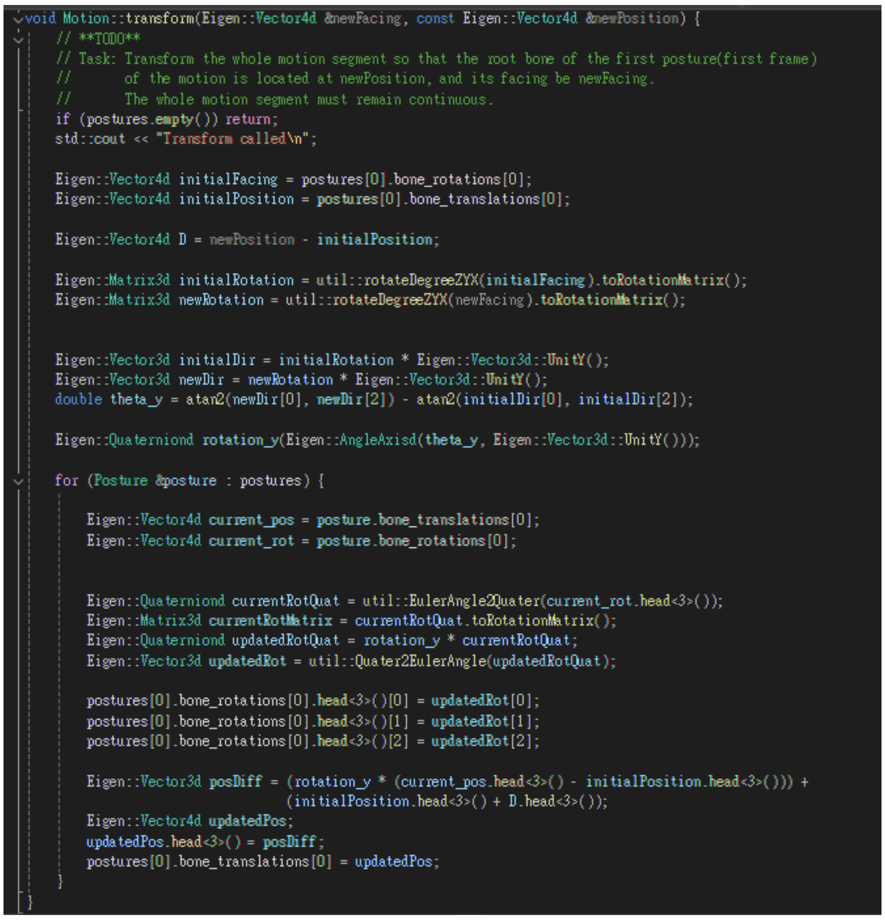
1. Cut large motion clips into small segments, then construct pose difference matrix between segments. Each segment will be seen as a node.

   – Finding good transitions (similar window of frames)
   - "Dense" map
   - This is before pruning

2. Implementation

Transform 的實作:

1. 先記下第一個 posture 的面向和位置
2. 利用參數給的新的面向和位置和剛剛記下的來計算旋轉矩陣
3. 計算新的方向向量和角度差 theta_y
4. 建構四元樹的旋轉
5. 把所有的 posture 更新 root bone 的位置和旋轉
6. 更新並反映其 rotaion 和 traslation



```
void Motion::transform(Eigen::Vector4d &newFacing, const Eigen::Vector4d &newPosition) {
    // **TODO**
    // Task: Transform the whole motion segment so that the root bone of the first posture(first frame)
    //       of the motion is located at newPosition, and its facing be newFacing.
    //       The whole motion segment must remain continuous.
    if (postures.empty()) return;
    std::cout << "Transform called\n";

    Eigen::Vector4d initialFacing = postures[0].bone_rotations[0];
    Eigen::Vector4d initialPosition = postures[0].bone_translations[0];

    Eigen::Vector4d D = newPosition - initialPosition;

    Eigen::Matrix3d initialRotation = util::rotateDegreeZYX(initialFacing).toRotationMatrix();
    Eigen::Matrix3d newRotation = util::rotateDegreeZYX(newFacing).toRotationMatrix();


    Eigen::Vector3d initialDir = initialRotation * Eigen::Vector3d::UnitY();
    Eigen::Vector3d newDir = newRotation * Eigen::Vector3d::UnitY();
    double theta_y = atan2(newDir[0], newDir[2]) - atan2(initialDir[0], initialDir[2]);

    Eigen::Quaterniond rotation_y(Eigen::AngleAxisd(theta_y, Eigen::Vector3d::UnitY()));

    for (Posture &posture : postures) {

        Eigen::Vector4d current_pos = posture.bone_translations[0];
        Eigen::Vector4d current_rot = posture.bone_rotations[0];


        Eigen::Quaterniond currentRotQuat = util::EulerAngle2Quater(current_rot.head<3>());
        Eigen::Matrix3d currentRotMatrix = currentRotQuat.toRotationMatrix();
        Eigen::Quaterniond updatedRotQuat = rotation_y * currentRotQuat;
        Eigen::Vector3d updatedRot = util::Quater2EulerAngle(updatedRotQuat);

        postures[0].bone_rotations[0].head<3>()[0] = updatedRot[0];
        postures[0].bone_rotations[0].head<3>()[1] = updatedRot[1];
        postures[0].bone_rotations[0].head<3>()[2] = updatedRot[2];

        Eigen::Vector3d posDiff = (rotation_y * (current_pos.head<3>() - initialPosition.head<3>())) +
                                  (initialPosition.head<3>() + D.head<3>());
        Eigen::Vector4d updatedPos;
        updatedPos.head<3>() = posDiff;
        postures[0].bone_translations[0] = updatedPos;
    }
}
```

blend 的實作:

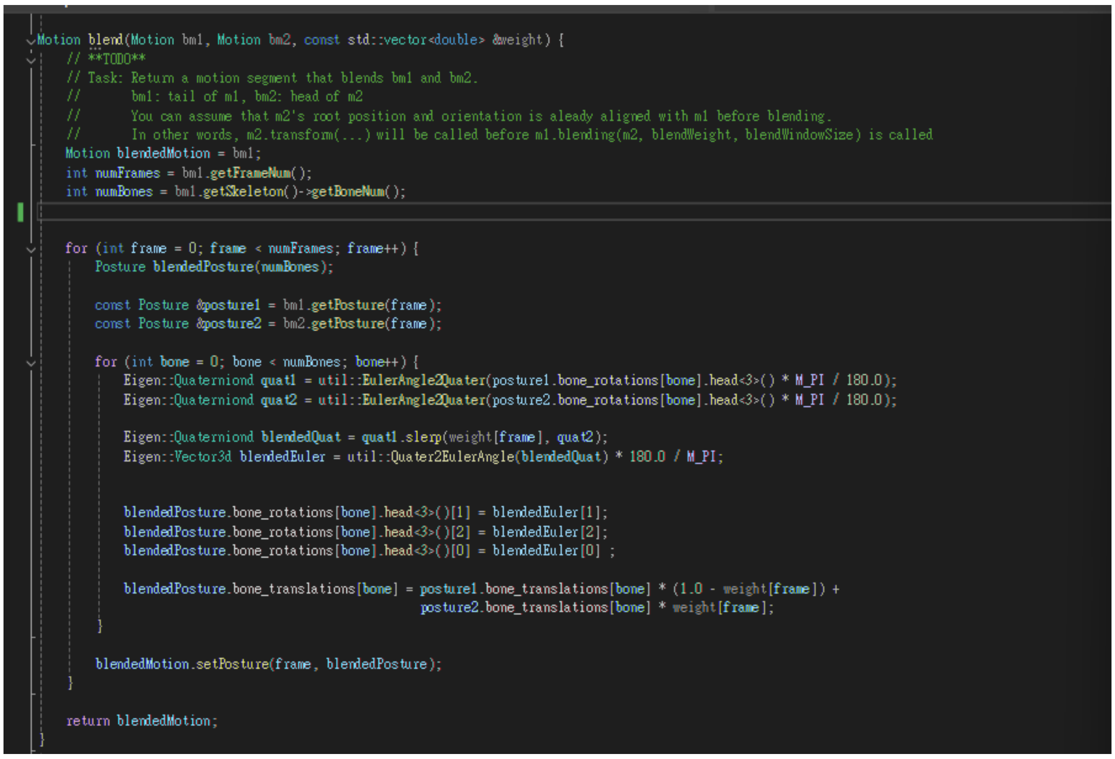
1 先初始化混和動作片段

2 對每一幀進行混和，分別除理位置和旋轉

3.用四元樹表示旋轉並且用 slerp 縣性差值還做混和比較穩定，將四元數轉回歐拉角來更新姿態

4 根據講義的公式來混合 bm1 和 bm2，bm1*(1-weight)+bm2*weight

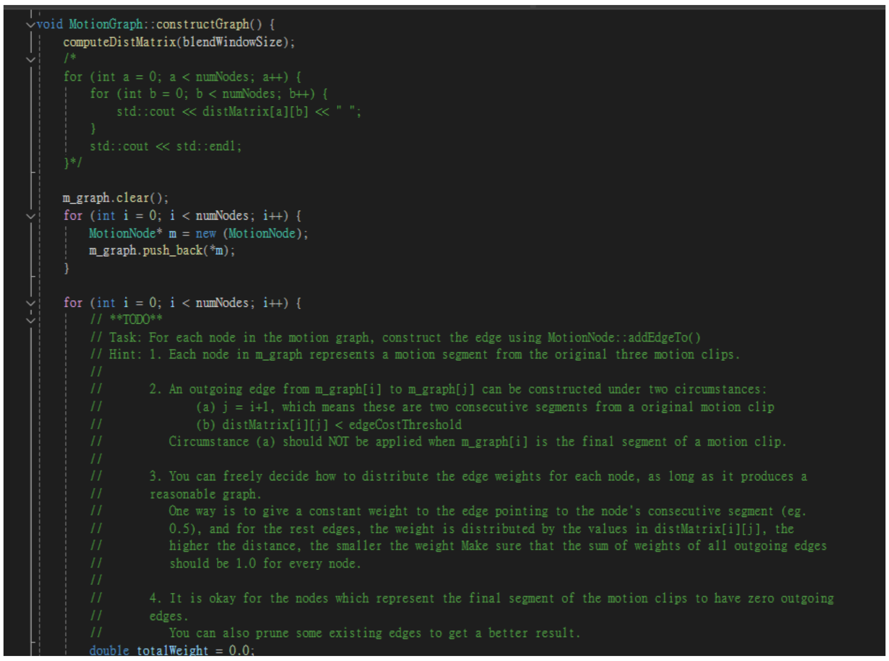
5 將混合過的設置到對應的幀的 posture 中，把結果設置回 blendedMotion 並回傳。



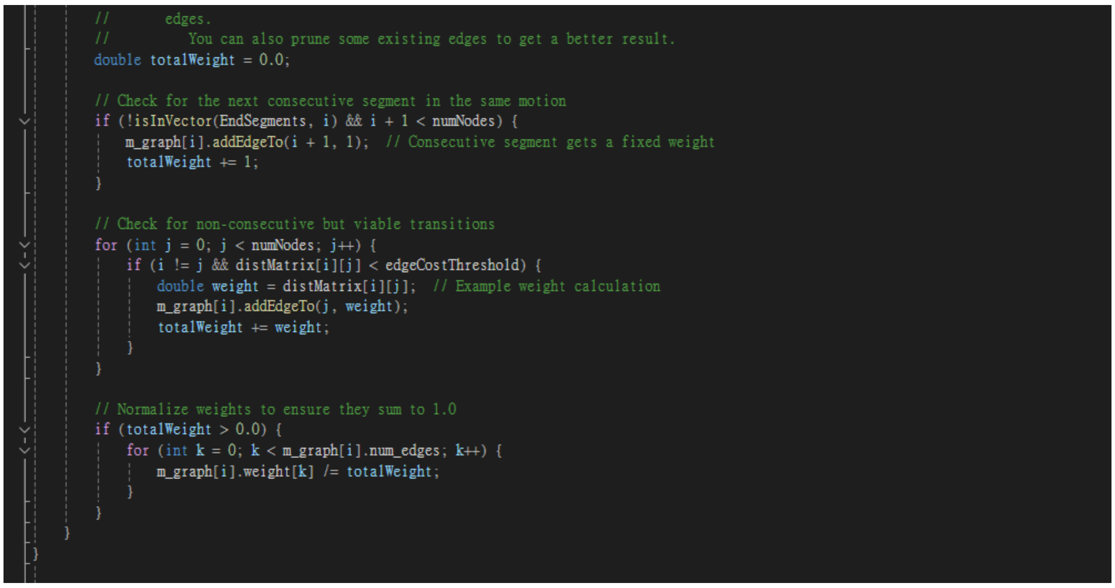
```cpp
Motion blend(Motion bm1, Motion bm2, const std::vector<double> &weight) {
    // **TODO**
    // Task: Return a motion segment that blends bm1 and bm2.
    //       bm1: tail of m1, bm2: head of m2
    //       You can assume that m2's root position and orientation is aleady aligned with m1 before blending.
    //       In other words, m2.transform(...) will be called before m1.blending(m2, blendWeight, blendWindowSize) is called
    Motion blendedMotion = bm1;
    int numFrames = bm1.getFrameNum();
    int numBones = bm1.getSkeleton()->getBoneNum();


    for (int frame = 0; frame < numFrames; frame++) {
        Posture blendedPosture(numBones);

        const Posture &posture1 = bm1.getPosture(frame);
        const Posture &posture2 = bm2.getPosture(frame);

        for (int bone = 0; bone < numBones; bone++) {
            Eigen::Quaterniond quat1 = util::EulerAngle2Quater(posture1.bone_rotations[bone].head<3>() * M_PI / 180.0);
            Eigen::Quaterniond quat2 = util::EulerAngle2Quater(posture2.bone_rotations[bone].head<3>() * M_PI / 180.0);

            Eigen::Quaterniond blendedQuat = quat1.slerp(weight[frame], quat2);
            Eigen::Vector3d blendedEuler = util::Quater2EulerAngle(blendedQuat) * 180.0 / M_PI;


            blendedPosture.bone_rotations[bone].head<3>()[1] = blendedEuler[1];
            blendedPosture.bone_rotations[bone].head<3>()[2] = blendedEuler[2];
            blendedPosture.bone_rotations[bone].head<3>()[0] = blendedEuler[0] ;

            blendedPosture.bone_translations[bone] = posture1.bone_translations[bone] * (1.0 - weight[frame]) +
                                                     posture2.bone_translations[bone] * weight[frame];
        }

        blendedMotion.setPosture(frame, blendedPosture);
    }

    return blendedMotion;
}
```

blend 的實作(我實作的部分):

1. 建構圖的邊
2. 假設是連續段落的話我在這兩個邊加一個固定的權重(1000)，因為我有去打印出來 disMatrix 的值大部分都 3~500 左右，所以我就讓到下一個的權重必其他高但是比例不要高太多(就一值順著跑下去就失去這次實作的意義了。
3. 假設是不連續的段落的話我就假設其 `distMatrix[i][j]<edgeCosThreshold` 的話代表兩個過渡是可行的所以就加了
4. 我在家 edge 權重的時候都有累加起來所有的 edge 都建完之後就統一除以 totalweight，讓他的權重和為 1。



```
void MotionGraph::constructGraph() {
    computeDistMatrix(blendWindowSize);
    /*
    for (int a = 0; a < numNodes; a++) {
        for (int b = 0; b < numNodes; b++) {
            std::cout << distMatrix[a][b] << " ";
        }
        std::cout << std::endl;
    }*/

    m_graph.clear();
    for (int i = 0; i < numNodes; i++) {
        MotionNode* m = new (MotionNode);
        m_graph.push_back(*m);
    }

    for (int i = 0; i < numNodes; i++) {
        // **TODO**
        // Task: For each node in the motion graph, construct the edge using MotionNode::addEdgeTo()
        // Hint: 1. Each node in m_graph represents a motion segment from the original three motion clips.
        //
        //       2. An outgoing edge from m_graph[i] to m_graph[j] can be constructed under two circumstances:
        //              (a) j = i+1, which means these are two consecutive segments from a original motion clip
        //              (b) distMatrix[i][j] < edgeCostThreshold
        //          Circumstance (a) should NOT be applied when m_graph[i] is the final segment of a motion clip.
        //
        //       3. You can freely decide how to distribute the edge weights for each node, as long as it produces a
        //       reasonable graph.
        //          One way is to give a constant weight to the edge pointing to the node's consecutive segment (eg.
        //          0.5), and for the rest edges, the weight is distributed by the values in distMatrix[i][j], the
        //          higher the distance, the smaller the weight Make sure that the sum of weights of all outgoing edges
        //          should be 1.0 for every node.
        //
        //       4. It is okay for the nodes which represent the final segment of the motion clips to have zero outgoing
        //       edges.
        //          You can also prune some existing edges to get a better result.
        double totalWeight = 0.0;
```



```
        //       edges.
        //          You can also prune some existing edges to get a better result.
        double totalWeight = 0.0;

        // Check for the next consecutive segment in the same motion
        if (!isInVector(EndSegments, i) && i + 1 < numNodes) {
            m_graph[i].addEdgeTo(i + 1, 1);  // Consecutive segment gets a fixed weight
            totalWeight += 1;
        }

        // Check for non-consecutive but viable transitions
        for (int j = 0; j < numNodes; j++) {
            if (i != j && distMatrix[i][j] < edgeCostThreshold) {
                double weight = distMatrix[i][j];  // Example weight calculation
                m_graph[i].addEdgeTo(j, weight);
                totalWeight += weight;
            }
        }

        // Normalize weights to ensure they sum to 1.0
        if (totalWeight > 0.0) {
            for (int k = 0; k < m_graph[i].num_edges; k++) {
                m_graph[i].weight[k] /= totalWeight;
            }
        }
    }
}
```

3.Result and Discussion

改 blendWindowSize 看他的差別

我分別輸入

1. blendWindowSize:一般的情況過度平滑但是在某些快速動作的場景會造成不自然的拖延感或者會閃現。

2. blendWindowSize/2 會沒那麼平滑，但是閃現變少了，也比較沒有那麼拖延了。

3. blendWindowSize/3 會比較少閃現也比較沒有拖延感但是場景切換會一直暴衝。

改 segmentSize 看他的差別

我分別輸入

1. segmentSize: 計算沒那麼多但是就沒有第二個那麼高的自然度和精確性。

2. segmentSize/2 需要計算的段數會更多，會增加計算的負擔。

3. segmentSize/3 會出錯。

改變權重值來優化 motion graph，前面有說到接到下一個 frame 的 weight 值太高的話會一直連續下去，就跟一個 AMC 按順序跑一樣了，但是如果接到下一個 frame 的 weight 值太低，就會亂跳，所以 weight 值要找中間值。

4.Conclusion

在這項研究中，我們通過實作動作圖、動作變換與混合深入探討了如何實現動作的平滑過渡，並使用 C++和 Visual Studio 作為開發工具。實驗過程中，我們發現混合窗口長度、段長以及邊的權重對於動作的自然度和流暢性有顯著的影響。透過精確的數學計算和四元數處理，我們成功地實現了不同動作之間的自然且流暢的過渡。雖然有助教有提示，但是熬夜三天的努力還是有些問題沒有辦法解決，如動作的爆轉和閃現問題，這些問題讓我意識到自己在程式開發技能上還有提升的空間。儘管如此，我認為自己已盡力而為，希望未來能進一步提升自己的程式設計能力，更好地解決類似的技術問題。