電腦動畫與特效 HW3 110612117 張仲瑜

I. Introduction/Motivation

在這個作業中，我們想要透過建立關於多個 motion 片段間的 motion graph，作為選擇連接 motion segment 的依據，最終建立出順暢且長度足夠的 motion sequence.

II. Fundamental

A. Motion Matching

透過定義的函式，來計算兩個 motion segments 之間的距離，此相似度後續會轉換成比重，作為連接兩個 segments 的優劣評斷，常見的定義方法為量化 motion 中每個骨架中節點的角度、位置差異等。

B. Motion Graph

透過 Motion Matching 計算 segment 與其他 segments 之間的距離，並根據距離的反比或其他衡量方式給定權重（切換的機率），並記錄成表格，作為執行時切換到特定 segment 的依據。

C. Motion Transform and Blending

在從 segment 1 切換到 segment 2 時，為了在給定的 frame 數中完成流暢的轉換，我們需要對後續連接的 segment 進行朝向以及位置的調整，對齊切換前的朝向以及位置。在對齊後，使用 sperical lerp 來產生逐漸切換的比例，並以此比例混合兩個 segment 中的 frame 來完成轉換過渡。

III. Implementation

A. Motion Transform

```
75  void Motion::transform(Eigen::Vector4d &newFacing, const Eigen::Vector4d &newPosition) {
76      // **TODO**
77      // Task: Transform the whole motion segment so that the root bone of the first posture(first frame)
78      //       of the motion is located at newPosition, and its facing be newFacing.
79      //       The whole motion segment must remain continuous.
80      Eigen::Vector3d root_pos = postures[0].bone_translations[0].head<3>();
81      Eigen::Vector4d root_rot = postures[0].bone_rotations[0];
82      Eigen::Vector3d translationDiff = newPosition.head<3>() - root_pos;
83
84      Eigen::Matrix3d ini_rot = util::rotateDegreeZYX(root_rot).toRotationMatrix();
85      Eigen::Matrix3d new_rot = util::rotateDegreeZYX(newFacing).toRotationMatrix();
86
87      Eigen::Vector3d ini_dir = ini_rot.col(2);
88      Eigen::Vector3d new_dir = new_rot.col(2);
89
90      double thetaY = std::atan2(new_dir[0], new_dir[2]) - std::atan2(ini_dir[0], ini_dir[2]);
91      Eigen::Matrix3d rot_along_y;
92      rot_along_y = Eigen::AngleAxisd(thetaY, Eigen::Vector3d::UnitY());
```

80~81: 取得 rootbone 於 frame0 的初始位置及方向

82: 計算 newPosition 與初始位置位置差

84~85: 轉換初始方向與新方向為 3*3 旋轉矩陣

87~92: 取出第三行的單位向量，再藉由 atan2 得出 thedaY，最後再透過 AngleAxised 取出 RotationMatrix R

```
94      for (auto &posture : postures) {
95          Eigen::Vector4d current_rot = posture.bone_rotations[0];
96          Eigen::Matrix3d current_rot_mat = util::rotateDegreeZYX(current_rot).toRotationMatrix();
97          Eigen::Matrix3d new_rot = rot_along_y * current_rot_mat;
98          posture.bone_rotations[0].head<3>() = new_rot.eulerAngles(2, 1, 0);
99          posture.bone_rotations[0] = util::toDegree(posture.bone_rotations[0]);
100
101         Eigen::Vector3d current_pos = posture.bone_translations[0].head<3>();
102         Eigen::Vector3d new_pos = rot_along_y * (current_pos - root_pos) + root_pos + translationDiff;
103         posture.bone_translations[0].head<3>() = new_pos;
104     }
105 }
```

95~99: 將 current_rot 轉成 rotation matrix 後，與 R 相乘再轉回 Degree 以完成旋轉

101~103: 平移部分，將 cuurent_pos 與 root_pos 乘上 R 旋轉後，再加回初始位置(root_pos)與加上 newPosition 與初始位置位置差。

B. Motion Blending

```
114     int numBlendedFrames = static_cast<int>(weight.size());
115     Motion blendMotion(bm1);
116
117     for (int i = 0; i < numBlendedFrames; i++) {
118         Posture &p1 = bm1.getPosture(i);
119         Posture &p2 = bm2.getPosture(i);
120         Posture pb;
121
122         pb.bone_rotations.resize(p1.bone_rotations.size());
123         pb.bone_translations.resize(p1.bone_translations.size());
124         int numBones = p1.bone_rotations.size();
```

114: 設定疊合 frame 數為與 weight 等 size

115: 宣告並初始化疊合的 motion

118~123: 在每個疊合 frame 中，取得兩個 motions 在當下的 posture，並初始化疊合 posture 的參數長度與骨頭數

```
125     for (int j = 0; j < numBones; j++) {
126         //Eigen::Quaterniond q1(util::rotateDegreeZYX(p1.bone_rotations[j]));
127         Eigen::Quaternion q1 = util::EulerAngle2Quater(util::toRadian(p1.bone_rotations[j]).head<3>());
128         // Eigen::Quaterniond q2(util::rotateDegreeZYX(p2.bone_rotations[j]));
129         Eigen::Quaternion q2 = util::EulerAngle2Quater(util::toRadian(p2.bone_rotations[j]).head<3>());
130         Eigen::Quaterniond interpolation = q1.slerp(weight[i], q2);
131         Eigen::Vector4d v;
132         v << util::Quater2EulerAngle(interpolation), 0;
133         pb.bone_rotations[j] = util::toDegree(v);
134         // std::cout << "weight:" << std::endl << weight[i] << std::endl;
135         // std::cout << "p1:" << std::endl << p1.bone_rotations[j] << std::endl;
136         // std::cout << "p2:" << std::endl << p2.bone_rotations[j] << std::endl;
137         // std::cout << "pb:" << std::endl << pb.bone_rotations[j] << std::endl;
138         /*pb.bone_rotations[j] = p1.bone_rotations[j] * (1 - weight[i]) + p2.bone_rotations[j] * weight[i];
139         pb.bone_translations[j] = p1.bone_translations[j] * (1 - weight[i]) + p2.bone_translations[j] * weight[i];*/
140         pb.bone_translations[j] = p1.bone_translations[j] * (1 - weight[i]) + p2.bone_translations[j] * weight[i];
141     }
142     blendMotion.setPosture(i, pb);
```

127~130: 針對每個骨頭的旋轉進行 slerp 內插

131~133: 將結果轉換回 degree 後存回疊合 posture

140: 對平移的部分做內插

142: 將該 frame 的 posture 存進疊合 motion 中

C. Motion Graph

```
192         double sumWeight = 0.0;
193         std::vector<std::pair<int, double>> edges;
194         for (int j = i + 1; j < numNodes; j++) {
195             bool isConsecutive = (j == i + 1);
196             bool isBelowThreshold = distMatrix[i][j] < 200;
197             if (isConsecutive) {
198                 m_graph[i].addEdgeTo(j, 0.5);
199             } else if (isBelowThreshold) {
200                 double weight = 1.0 / distMatrix[i][j];
201                 edges.push_back({j, weight});
202                 sumWeight += weight;
203             }
204         }
205         if (sumWeight > 0.0) {
206             for (auto& edge : edges) {
207                 edge.second /= sumWeight;
208                 m_graph[i].addEdgeTo(edge.first, edge.second * 0.5);
209             }
210         }
211     }
212 }
```

將 threshold 設為 200，只有距離小於 200 的狀況才會切換 Node

將有連續的 Node 的權重設為 0.5，並將其他沒有連續根據距離反比正規化。

IV. Result and Discussion

A. Effect of parameters

i. Motion graph threshold

太小的 threshold 會導致可以連接的 node 數太小(Edge 很少)而一直走連續的 motion segment，然而太大的 threshold 會提升頻繁切換的可能性(不穩定度)與增加計算量，因為需要對齊相差更遠的 posture.

ii. Motion graph weight formula

若給連續的 node 的權重過大的話，容易導致其他正規化的權重相對下過小，這樣會讓整個 motion graph 的多樣性減少，因為系統會偏向選擇權較大的連續 segment，而忽略了其他可能的切換路徑。

iii. Blending window length

Blending window length 決定了在切換不同 motion segment 時，進行內插過渡的時間長度。若 blending window 太短，則過渡可能會顯得突兀，導致動畫不自然；反之，則會增加計算複雜度並可能導致過渡過程中動作不明顯。而且過長的 blending window 可能會使內插時間過長而失去原本兩個 motion 各自的特別動作。

iv. Segment length

Segment length 決定了每個 motion segment 的長度。如果 segment 過短，則會導致系統需要頻繁地切

換 segment，增加了計算複雜度和不穩定性；反之則會限制動作的靈活性。因為系統必須走完整個 segment 才能切換到下一個動作，這可能會導致動作的多樣性降低，並且無法及時切換動作。

V. Conclusion

這次的實作相當有趣，尤其在拿不同的 motion 的串聯結果與各自的 motion files 來切換比較時，趣味感油然而生。然而這次的角度、弧度；四元數與各種的切換讓我吃足了苦頭，花了巨量的時間來修正型別錯誤與錯誤單位的套用修正，其中也不乏許多有趣的人物大風車 frame 出現，或是瞬間移動等等，幸好最後都逐一解決，也讓我對使用大專案內提供的 functions 更加熟悉，希望下次還有機會可以接觸類似的專案。