HW3 REPORT

110511010 楊育陞

1 Introduction

In this homework, we implemented motion blending and motion graph algorithms. Motion blending is a technique to blend the transition between two motions. Motion graph is to create a graph of motions and transitions between them. By using these techniques, we can create new motions by blending existing motions.

2 Implementation

2.1 Motion Matching

To decide which motion to blend, we need to determine the distance between two motions, which means how similar the two motions are. In this homework, the distance is calculated by the difference of the root positions and orientations.

2.2 Motion Transforms

To blend two motions, we need to transform the motion first. We can transform the motion by aligning the root position and orientation. In this homework, we implemented the rotation along one axis and the translation on 2D plane. The steps are as follows:

1. Compute the angle between the two orientations.
2. Rotate the motion by the angle.
3. Translate the motion by the difference of the root positions.
4. Repeat the above steps for all frames.

The code is as follows:

```
void Motion::transform(const Eigen::Vector4d &newFacing, const Eigen::Vector4d &newPosition) {
    Eigen::Vector4d oldFacing = postures[0].bone_rotations[0];
    Eigen::Matrix3d oldRot = util::rotateDegreeZYX(oldFacing).toRotationMatrix();
    Eigen::Matrix3d newRot = util::rotateDegreeZYX(newFacing).toRotationMatrix();
    double oldAngle = atan2(-oldRot(2, 0), oldRot(0, 0));
    double newAngle = atan2(-newRot(2, 0), newRot(0, 0));
    Eigen::Matrix3d rotMat = util::rotateRadianZYX(0, newAngle-oldAngle, 0).toRotationMatrix();
    Eigen::Vector4d oldPosition = postures[0].bone_translations[0];
    const int frameNum = getFrameNum();
    for (int i = 0; i < frameNum; i++)
    {
        Eigen::Vector4d &curRot = postures[i].bone_rotations[0];
        Eigen::Matrix3d curRotMat = rotMat * util::rotateDegreeZYX(curRot).toRotationMatrix();
        Eigen::Vector3d euler = curRotMat.eulerAngles(2, 1, 0) * util::rad2deg;
        curRot.head<3>() = Eigen::Vector3d(euler[2], euler[1], euler[0]);
        Eigen::Vector4d &curPos = postures[i].bone_translations[0];
        Eigen::Vector3d curPosVec = (curPos - oldPosition).head<3>();
        curPos.head<3>() = (rotMat * curPosVec + newPosition.head<3>());
    }
}
```

First, I created the rotation matrix of the old and new orientations. Since the bone rotations are in euler angles of order zyx, I converted them to rotation matrices using the util functions rotateDegreeZYX. Then I calculated the rotation angle projected on the xz plane using atan2 function. After that, I calculated the rotation matrix to rotate the motion. Finally, I rotated and translated the motion for all frames. Since the bone rotations are in euler angles of order zyx, I reordered them after the rotation.

2.3 Motion Blending

After transforming, I blended the two motions by interpolating. Translations are interpolated linearly, and rotations are interpolated by slerp. The code is as follows:

```
Motion blend(Motion bm1, Motion bm2, const std::vector<double> &weight) {
    const int frameNum = bm2.getFrameNum();
    for (int i = 0; i < frameNum; i++) {
        Posture& posture1 = bm1.getPosture(i);
        Posture& posture2 = bm2.getPosture(i);
        const double w2 = weight.at(i), w1 = 1 - w2;
        posture2.bone_translations[0] = posture1.bone_translations[0] * w1 + posture2.bone_translations[0] * w2;
        Eigen::Quaterniond q1 = util::rotateDegreeZYX(posture1.bone_rotations[0]);
        Eigen::Quaterniond q2 = util::rotateDegreeZYX(posture2.bone_rotations[0]);
        q2 = q1.slerp(w2, q2);
        Eigen::Vector3d euler = q2.toRotationMatrix().eulerAngles(2, 1, 0) * util::rad2deg;
        posture2.bone_rotations[0].head<3>() = Eigen::Vector3d(euler[2], euler[1], euler[0]);
    }
    return bm2;
}
```

Since the bone rotations are in euler angles, I converted them to quaternions using the util functions rotateDegreeZYX. Then I interpolated the translations linearly and the rotations by slerp. Finally, I created a new motion by blending the two motions.

2.4 Motion Graph

In the construction of the motion graph, we need to decide the edges between the nodes. For one node, I first traverse among the distances between the node and all other nodes. If the distance is less than the edge cost threshold, I add an edge between the two nodes. Also if the node is the next node of the motion of the current node, I add an edge between the two nodes. Furthermore, I pruned the edges inside the same motion. The weights of the edges are determined by the distances between the two nodes, calculated in the motion matching step. The code is as follows:

```
for (int i = 0; i < numNodes; i++) {
    bool isEnd = isInVector(EndSegments, i);
    int outEdges = 0;
    std::vector<int> outEdgeIdx;
    double distSum = 0.0;
    std::vector<double> distVector;
    for (int j = 0; j < numNodes; j++) {
        if (i == j || j == i + 1 || motionSrc == getMotionSrc(EndSegments, j)) {
            continue;
        }
        if (distMatrix[i][j] < edgeCostThreshold) {
            outEdges++;
            outEdgeIdx.push_back(j);
            distSum += distMatrix[i][j];
            distVector.push_back(distMatrix[i][j]);
        }
    }
    if (outEdges == 0) {
        if (!isEnd) {
            m_graph[i].addEdgeTo(i + 1, 1.0);
        }
    } else {
        if (isEnd) {
            for (int j = 0; j < outEdges; j++) {
                m_graph[i].addEdgeTo(outEdgeIdx[j], distVector[j] / distSum);
            }
        } else {
            m_graph[i].addEdgeTo(i + 1, 0.5);
            for (int j = 0; j < outEdges; j++) {
                m_graph[i].addEdgeTo(outEdgeIdx[j], 0.5 * distVector[j] / distSum);
            }
        }
    }
}
```

3 Result and Discussion

3.1 Result

3.1.1 Testing amc files

- walk_fwd_circle.amc
- walk_fwd_curve1.amc
- walk_fwd_curve2.amc

3.1.2 Demo link

`https://youtu.be/koIaQvi4g2c?si=Yu2uwCMFFseZUWr0`

3.2 Discussion

3.2.1 Effect of the Edge Cost Threshold

The edge cost threshold is the threshold to decide whether to add an edge between two nodes. If the distance between two nodes is less than the threshold, we add an edge between the two nodes. If the threshold is large, the graph will be more connected, creating more transitions between motions.

3.2.2 Effect of the Blending Window Length

The blending window length is the number of frames to blend the two motions. If the length is small, the blending will be more abrupt. If the length is large, the blending will be smoother, but the transition will be slower.

4 Conclusion

In this homework, I implemented motion blending and motion graph algorithms. The hardest part for me was the transformation of the motion. I had to deal with the rotation among different coordinate systems, which was unfamiliar to me at first.