HW3

110550085 房天越

1. Introduction

In this assignment, we need to implement the motion graph based on the kinematics implemented in last assignment. There are mainly three parts to implement, which are the transformations, the blending between two motions, and the construction and the distributions of the weights.

2. Implementations

I. Transform

The implementation of this part is based on the hints the TA’ s gave us. There are mainly 6 parts:

First, we take the start position and start facing of root bone in frame 0 by taking posture[0].bone_translations[0] and postures[0].bone_rotations[0].

Second, get the difference between new position and the start position.

Third, transform the start facing and the new facing to a 3*3 rotation matrix using rotateDegreeZYX and toRotationMatrix,

we also need to normalize the vector after using rotateDegreeZYX.

Fourth, we take the third column of the rotation matrices and take the third component of the taken vectors to calculate the angles rotated by the y-axis, and finally calculate the thetaY by the difference between the two angles.

Fifth, we calculate the 3*3 rotation matrix R using the theta with the given formula.

Finally, we traverse all the postures in the motion, take the current position and the current rotation of posture[i], and calculate the rotation and the position with the rotation matrix we calculated.

The code is as follows:

```cpp
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.

    Eigen::Vector4d start_position = postures[0].bone_translations[0];
    Eigen::Vector4d start_facing = postures[0].bone_rotations[0];

    Eigen::Vector4d diff = newPosition - start_position;

    Eigen::Matrix3d start_rot_mat =
        util::rotateDegreeZYX(start_facing).normalized().toRotationMatrix();
    Eigen::Matrix3d new_rot_mat =
        util::rotateDegreeZYX(newFacing).normalized().toRotationMatrix();

    double theta_start = atan2(start_rot_mat(0, 2), start_rot_mat(2, 2));
    double theta_new = atan2(new_rot_mat(0, 2), new_rot_mat(2, 2));
    double facing_angle = theta_new - theta_start;

    //Use facing_angle to get the rotation matrix rotated about the y-axis
    Eigen::Matrix3d R;
    R(0, 0) = cos(facing_angle);
    R(0, 2) = sin(facing_angle);
    R(1, 1) = 1;
    R(2, 0) = -sin(facing_angle);
    R(2, 2) = cos(facing_angle);
    R(0, 1) = 0;
    R(1, 0) = 0;
    R(1, 2) = 0;
    R(2, 1) = 0;

    //Traverse all the postures and apply the transformation
    for (int i = 0; i < postures.size(); i++) {
        Posture p = postures[i];
        Eigen::Vector4d current_pos = p.bone_translations[0];
        Eigen::Vector4d current_rot = p.bone_rotations[0];

        //Translate the root rotation
        //First make current_rot a 3*3 rotation matrix
        Eigen::Matrix3d current_rot_mat = util::rotateDegreeZYX(current_rot).normalized().toRotationMatrix();
        //Multiply the rotation matrix by R
        Eigen::Matrix3d new_rot_mat = R * current_rot_mat;
        //Convert it back to ZYX angles
        p.bone_rotations[0].head<3>() = new_rot_mat.eulerAngles(2, 1, 0).reverse() * 180 / 3.14159265;

        //Translate the root position
        Eigen::Vector3d r_diff = current_pos.head<3>() - start_position.head<3>();
        p.bone_translations[0].head<3>() =
            R * r_diff + start_position.head<3>() + diff.head<3>();

        postures[i] = p;

    }
}
```

II. Blend

In this part, we need to return a motion that is blended with the two given segments, first, we declare a motion that equals to bm1, than, traverse every frame, for the position, we simply calculate it with the weight. For the rotation, we calculate it using a technique called “Slerp”.

The equation for SLERP is listed as follows

$$Slerp(q_1, q_2; u) = \frac{\sin(1-u)\theta}{\sin\theta} q_1 + \frac{\sin(u\theta)}{\sin\theta} q_2$$

To use this technique, we first use rotateDegreeZYX to transform the rotations of the two postures to Quaterniond, then apply slerp.

Finally, convert it back to Euler angles and set it to the new posture’ s bone_rotation, then set the corresponding posture of the motion.

Finally return the motion.

Here is the code of this part:

```
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

    std::cout << "Blending"<<std::endl;

    Motion bm = bm1;

    for (int i = 0; i < bm1.getFrameNum(); i++) {
        Posture p1 = bm1.getPosture(i);
        Posture p2 = bm2.getPosture(i);

        Posture new_p = p1;


        new_p.bone_translations[0] = p1.bone_translations[0] * (1 - weight[i]) + p2.bone_translations[0] * weight[i];

        for (int j = 0; j < new_p.bone_rotations.size(); j++) {
            Eigen::Quaterniond q1 = util::rotateDegreeZYX(p1.bone_rotations[j]);
            Eigen::Quaterniond q2 = util::rotateDegreeZYX(p2.bone_rotations[j]);

            Eigen::Quaterniond new_q = q1.slerp(weight[i], q2);
            new_p.bone_rotations[j].head<3>() = new_q.normalized().toRotationMatrix().eulerAngles(2, 1, 0).reverse() * 180 / 3.14159265;
        }

        bm.setPosture(i, new_p);


    }

    return bm;
}
```

III. ConstructGraph

The way to construct graph is according to the based on the given hint, we first normalize the edges using the sum of 1/distMatrix[i][j], the way to use reciprocal is because the farther it is, the lower the weight should be.

Then, if the sum is zero, we give the edge from i to i+1 a weight = 1. If not, we give the edge from i to i+1 a weight = 0.5, then distribute the weights according to their distances with a total weights = 0.5.

The code is here:

```
double sum = 0.0;
for (int j = 0; j < numNodes; j++) {
    if (j == i + 1 && i != numNodes - 1) {
        continue;
    }
    else if (distMatrix[i][j] < edgeCostThreshold) {
        sum += 1.0 / distMatrix[i][j];
    }
}

double certi_sum = 0.0;

for (int j = 0; j < numNodes; j++) {

    if (j == i + 1 && i != numNodes - 1) {
        if (sum == 0.0) {
            m_graph[i].addEdgeTo(j, 1.0);
            certi_sum += 1.0;
            continue;
        }
        else {
            m_graph[i].addEdgeTo(j, 0.5);
            certi_sum += 0.5;
        }
    }
    else if (distMatrix[i][j] < edgeCostThreshold) {
        m_graph[i].addEdgeTo(j, 0.5 * ((1.0 / distMatrix[i][j]) / sum));
        certi_sum += 0.5 * ((1.0 / distMatrix[i][j]) / sum);
    }
}
```

IV. Effects of parameters

1. The Blending Window Length

   The blending window length is the number of frames over which two motion segments are interpolated to create a smooth transition. If the length is longer, the transition would be smoother and more gradual. If the length is shorter, the transitions might be more of a sudden, but could also preserve more original motion data. If the length is longer, the computational cost would be also higher.

2. The Segment Length

   The segment length is the number of frames that make up each segment of motion within the graph.

   If the segments are shorter, the control over the motion graph would be more flexible and fine-grained, but the transitions would be more frequent and the result might be noticeably artificial. The computational cost would also be higher if the segment lengths are shorter.

3. Result and Discussion

I. The Names of the .AMC Files to Use to Test the Code

I did not modify the AMC files, so it is OK to use the original

three .AMC files, which are:

walk_fwd_circle.AMC

walk_fwd_curve1.AMC

walk_fwd_curve2.AMC

II. The Result of Motion

We can see that the result is quite nice, the transformation and the blending part are not obvious, if we do not look at the terminal or the blocks on the screen, we would not know that there is a jump.

III. Some problems I met

The first problem is that the way to implement the transform function is too difficult, before the hints are given, I spent more than one day figuring out how I should implement that.

The second problem is that it is very difficult to make sure whether I have implemented the functions correctly because I need to finish all parts to know it. When I was in the state of try-and-error, I found some problems about the positions and rotations, the character would flip around or suddenly be at another places, and I found it difficult to know which part I

messed up.

4. Conclusion

In this assignment, I learned how to implement the functions about the motion graph, and understood how the transformation and blending between two motions are realized. Also, I learned how to use some functions like slerp. I also learned how to construct graph based on the distance between motion segments. This is an interesting and important experience. I learned a lot from it!