HW3

111550088 張育維

1. Introduction

This homework let us choose the better motion and blend them together. We should try to choose the reasonable motion segment, and blend them in a fluent way.

2. Implementation

transform

```
if (postures.empty()) return;

Eigen::Vector4d initPosition = postures[0].bone_translations[0];
Eigen::Vector4d initRotation = postures[0].bone_rotations[0];
Eigen::Matrix3d initDirection = util::rotateDegreeZYX(initRotation).normalized().toRotationMatrix();
Eigen::Matrix3d newDirection = util::rotateDegreeZYX(newFacing).normalized().toRotationMatrix();
Eigen::Matrix3d matrix = newDirection * initDirection;
double angle = atan2(matrix.row(0)[2], matrix.row(2)[2]);
//double angle = -atan2(newDirection(2, 2), newDirection(0, 2)) + atan2(initDirection(2, 2), initDirection(0, 2));
Eigen::Matrix3d R = Eigen::AngleAxisd(angle, Eigen::Vector3d::UnitY()).toRotationMatrix();

for (int i = 0; i < postures.size(); i++) {
    Eigen::Vector4d &cur_rot = postures[i].bone_rotations[0];
    Eigen::Matrix3d cur_rot_mat = util::rotateDegreeZYX(cur_rot).normalized().toRotationMatrix();
    Eigen::Vector3d newAngle = util::Quater2EulerAngle(Eigen::Quaterniond(R * cur_rot_mat));
    cur_rot.head<3>() = newAngle;

    Eigen::Vector4d &cur_pos = postures[i].bone_translations[0];
    cur_pos.head<3>() = R * (cur_pos - initPosition).head<3>() + newPosition.head<3>();
}
```

In this part, I store the initial data of root bone of the first frame, and use the function ’rotateDegreeZYX’ to make the rotation matrix of ‘newFacing’ and the root bone of the first frame, called ‘newDirection’ and ‘initDirection’, respectively. I calculate the angle along y-axis by ‘newDirection’ * ‘initDirection’, and puting the result’s [0][2] and [2][2] into function ‘atan2()’. Next, I generate the rotation matrix of this angle by changing it into ‘Eigen::AngleAxisd’ and use the function ‘toRotationMatrix()', which called ‘R’. After that, I trace all postures in the vector called ‘postures’. I change ‘bone_rotations[0]’ by ‘R’ * the matrix generated by root bone facing of this posture, and then use function ‘util::Quater2EulerAngle to change it back to ‘Eigen::Vector3d’. For ‘bone_translations[0]’, I change ‘bone_translation[0]’ by ‘R’ * the difference of new position and the frame 0’s initial position + the new position.

blending

```
int window = bm1.getFrameNum();
Motion r = bm1;
for (int i = 0; i < window; i++) {
    r.getPosture(i).bone_translations[0] = bm1.getPosture(i).bone_translations[0] *  weight[i] +
                                           bm2.getPosture(i).bone_translations[0] * (1.0 - weight[i]);
    /*Eigen::Quaterniond t1(bm1.getPosture(i).bone_translations[0]);
    Eigen::Quaterniond t2(bm2.getPosture(i).bone_translations[0]);
    Eigen::Quaterniond inter = t1.slerp(weight[i], t2);
    Eigen::Vector3d trans = util::Quater2EulerAngle(inter);
    r.getPosture(i).bone_translations[0].head<3>() = trans;*/
    Eigen::Quaterniond r1 = util::rotateDegreeZYX(bm1.getPosture(i).bone_rotations[0]);
    Eigen::Quaterniond r2 = util::rotateDegreeZYX(bm2.getPosture(i).bone_rotations[0]);
    Eigen::Quaterniond interpolate = r1.slerp((1-weight[i]), r2);

    Eigen::Vector3d rotate = util::Quater2EulerAngle(interpolate);
    // r.getPosture(i).bone_rotations[0] = bm1.getPosture(i).bone_rotations[0] * weight[i] +
    // bm2.getPosture(i).bone_rotations[0] * (1.0 - weight[i]);
    r.getPosture(i).bone_rotations[0].head<3>() = rotate;
}
return r;
```

In this part, I use the linear interpolation to do the position part. I use the root bone's position of this frame * weight of this frame + the root bone's position of corresponding frame * (1-weight of this frame), and store the result into 'r', which is a motion. For rotation part, I change the rotation data into 'Eigen::Quaterniond' and use the function 'slerp' to calculate SLERP. After that, I use the function 'util::Quater2EulerAngle' to change it back to 'Eigen::Vector3d' and put it into the root bone's rotation of motion called 'r'. After tracing all frames, return the 'r'.

construct Graph

```
double sum = 0.0;
std::vector<std::pair<int, double>> edges;
if (i != numNodes - 1) {
    edges.push_back({i + 1, 0.5});
}
for (int j = 0; j < distMatrix[i].size(); j++) {
    if (i != j && distMatrix[i][j] < edgeCostThreshold && i +1 != j) {
        edges.push_back({j, 1.0 / distMatrix[i][j]});
        sum += 1.0 / distMatrix[i][j];
    }
}
if (edges.size() == 1) {
    m_graph[i].addEdgeTo(i + 1, 1.0);
    continue;
}
if (edges.size() == 0) {
    continue;
}
double test = 0;
for (const auto& edge : edges) {
    if (edge.first == i + 1) m_graph[i].addEdgeTo(i + 1, edge.second);
    else m_graph[i].addEdgeTo(edge.first, edge.second / sum * (1.0 - 0.5));
}
```

First, I define a vector called ‘edges’ to store the index and distance of node that I want to add edge, and define a double called ‘sum’ to store the value. If the index doesn’t equal to ‘numNodes’, which means ‘i+1’ must exist, then ‘edges’ pushes back {i+1, 0.5}. Next, I trace the whole index. If the index doesn’t equal to this one, this one + 1, and the distance from the node to this one is smaller than ‘edgeCostThreshold’, then edges pushes back{the node’s index, 1/distance}, and ‘sum’ add 1/distance. After that, if ‘edges’ only has one component, then use function ‘addEdgeTo’ to add one edge between two nodes, and weight equals to 1 and skip the following code. If there is no component in ‘edges’, skip directly. Otherwise, I trace the whole ‘edges’ and use ‘addEdgeTo’ to make edge between two nodes, and the weight of the consecutive one is always 0.5, and the weight of others depend on ‘1/distance(store in vector ‘edges’) / sum * (1-0.5), which means that others share the rest 0.5, and the sum of all weight will be one.

3. Result and Discussion

blending window length:

Because it refers to the length of the transition region during blending, when it is longer, the result will be more smoothy, which can avoiding sudden motion changes and making motion transitions more seamless. However,

increasing the blending window length may cause computational overhead during blending.

segment length:

Because it refers to the length of each motion segment in the motion sequence, longer segment lengths provide smoother motions as they capture more motion details and variations. Also, it can decrease the computational cost because of the decrease of the blending needing.

4. Conclusion

In this homework, although my skeleton still has some weird movement, I think it is fluent enough. I would like TAs use ‘walk_fwd_circle.amc’ to test my code.