

1 Youtube link

<https://www.youtube.com/watch?v=mbqRKKrZFag>.

2 Required feature 1: Simple 2D IK in 2D

The simple 2D IK in 2D was implemented with analytic method. Only 2 bones are used in this case (see Figure 1). The following code shows how the IK is calculated.

```
1 // when the target is over the distance the two bones can reach to
2 if (std::sqrt(positionX * positionX + positionY * positionY) > (
3     bone1_length + bone2_length))
4 {
5     if (positionY > 0) bone1_angle = std::acos(positionX / std::sqrt(
6         positionX * positionX + positionY * positionY));
7     else bone1_angle = -std::acos(positionX / std::sqrt(positionX *
8         positionX + positionY * positionY));
9
10    bone2_angle = 0;
11 }
12 // when the target is inside the area the two bones can reach to
13 else if (std::sqrt(positionX * positionX + positionY * positionY) < std::
14     abs(bone1_length - bone2_length))
15 {
16     if (positionY > 0) bone1_angle = std::acos(positionX / std::sqrt(
17         positionX * positionX + positionY * positionY));
18     else bone1_angle = -std::acos(positionX / std::sqrt(positionX *
19         positionX + positionY * positionY));
20
21    bone2_angle = M_PI;
22 }
23 // to avoid the bones flip when go beyond x_axis or y_axis, devide them to
24 // 2 cases
25 else if (positionY > 0)
26 {
27     angle_T = std::acos(positionX / std::sqrt(positionX * positionX +
28         positionY * positionY));
29     angle_1 = std::acos((bone1_length * bone1_length + positionX * positionX
30         + positionY * positionY - bone2_length * bone2_length) / (2 *
31         bone1_length * std::sqrt(positionX * positionX + positionY * positionY
32         )));
33     bone1_angle = angle_1 + angle_T;
34     bone2_angle = -std::acos(-(bone1_length * bone1_length + bone2_length *
35         bone2_length - positionX * positionX - positionY * positionY) / (2 *
36         bone1_length * bone2_length));
37 }
38 else
39 {
40 }
```

```

27     angle_T = std::acos((bone1_length * bone1_length + positionX * positionX
28         + positionY * positionY - bone2_length * bone2_length) / (2 *
29             bone1_length * std::sqrt(positionX * positionX + positionY * positionY
30             )));
31     angle_1 = std::acos(positionX / std::sqrt(positionX * positionX +
32         positionY * positionY));
33     bone1_angle = -(angle_1 - angle_T);
34     bone2_angle = -std::acos(-(bone1_length * bone1_length + bone2_length *
35         bone2_length - positionX * positionX - positionY * positionY) / (2 *
36             bone1_length * bone2_length));
37 }
```

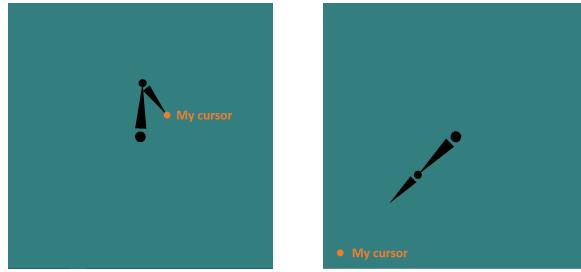


Figure 1: IK on 2 bones in 2D (analytic method).

3 Required feature 2: Multi-bone IK in 3D

In my work, CCD was used for calculate IK in 3D. For demonstrate this feature, a diamond was created. I only calculated the IK for its left arm. The left arm consists of 6 bones. Among the last 3 pyramids, only the centre one is used for IK and its tail end is where the end effector is. Figure 2 are screenshots of moving the target position. You can check the YouTube video (the link is in Section 1) for more demonstration. Besides IK, for better appearance, skybox and a ground were added to make the scene looks better. Blinn-Phong model was used for rendering.

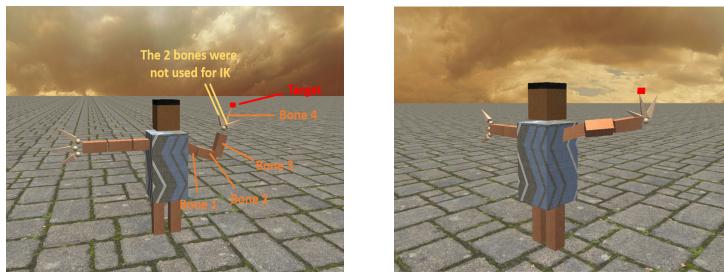


Figure 2: IK on multiple bones in 3D (CCD).

The following code shows how I implemented CCD. I update the orientation up to 5000 times per frame. I also check whether the bones are in a good orientation with *CCDoverCondition*

function. If the end effector is already close enough to the target position, it will stop update. These conditions reduced the oscillation which is a problem of CCD method. In addition, quaternion was used to rotate those bones.

```
1 bool CCDoverCondition(glm::vec3 _target_position, glm::vec3
2   _limb1_position, float _humanchain_length, glm::vec3
3   _humanchain_endPosition)
4 {
5   bool over = false;
6
7   if (glm::length(_target_position - _limb1_position) <= 0.95 *
8     _humanchain_length)
9   {
10     if (_humanchain_length && glm::length(_target_position -
11       _humanchain_endPosition) < 0.1)
12       over = true;
13   }
14   else if (glm::length(_target_position - _limb1_position) <=
15     _humanchain_length)
16   {
17     if (_humanchain_length && glm::length(_target_position -
18       _humanchain_endPosition) < 0.2)
19       over = true;
20   }
21   else
22   {
23     glm::vec3 p2e = _humanchain_endPosition - _limb1_position;
24     glm::vec3 p2t = _target_position - _limb1_position;
25     glm::vec3 e2t = _target_position - _humanchain_endPosition;
26     float angle = std::acos(glm::dot(glm::normalize(p2e), glm::normalize(
27       p2t)));
28
29     if (angle < 0.01 && glm::length(e2t) < 0.5)
30       over = true;
31   }
32
33   return over;
34 }
35
36 // at most update 5000 iterations per frame
37 int counter = 5000;
38 float clip_angle = 0.05;
39
40 trans = glm::toMat4(limb1_quat);
41 limb2_position = glm::mat3(trans) * glm::vec3(limb1_length, 0.0f, 0.0f) +
42   limb1_position;
43 trans *= glm::toMat4(limb2_quat);
44 limb3_position = glm::mat3(trans) * glm::vec3(limb2_length, 0.0f, 0.0f) +
45   limb2_position;
46 trans *= glm::toMat4(limb3_quat);
47 finger1_position = glm::mat3(trans) * glm::vec3(limb3_length, 0.0f, 0.0f)
```

```
    + limb3_position;
39 trans *= glm::toMat4(finger1_quat);
40 humanchain_endPosition = glm::mat3(trans) * glm::vec3(finger1_length, 0.0f
    , 0.0f) + finger1_position;
41
42 // calculate the position and orientation of the bones one by one
43 while (--counter)
44 {
45     if (CCDOverCondition(cube_position, limb1_position, humanchain_length,
        humanchain_endPosition))
46         break;
47
48     glm::vec3 p2e;
49     glm::vec3 p2t;
50     float angle;
51     p2e = glm::normalize(humanchain_endPosition - finger1_position);
52     p2t = glm::normalize(cube_position - finger1_position);
53     angle = std::acos(std::max(std::min(glm::dot(p2e, p2t), 1.0f), -1.0f));
54
55     if (angle >= 0.005)
56     {
57         if (angle > clip_angle) angle = clip_angle;
58
59         rotAxis = glm::normalize(glm::cross(p2e, p2t));
60         rot = glm::angleAxis(angle, rotAxis);
61         finger1_quat = rot * finger1_quat;
62
63         trans = glm::toMat4(limb1_quat);
64         limb2_position = glm::mat3(trans) * glm::vec3(limb1_length, 0.0f, 0.0f
    ) + limb1_position;
65         trans *= glm::toMat4(limb2_quat);
66         limb3_position = glm::mat3(trans) * glm::vec3(limb2_length, 0.0f, 0.0f
    ) + limb2_position;
67         trans *= glm::toMat4(limb3_quat);
68         finger1_position = glm::mat3(trans) * glm::vec3(limb3_length, 0.0f,
0.0f) + limb3_position;
69         trans *= glm::toMat4(finger1_quat);
70         humanchain_endPosition = glm::mat3(trans) * glm::vec3(finger1_length,
0.0f, 0.0f) + finger1_position;
71     }
72     if (CCDOverCondition(cube_position, limb1_position, humanchain_length,
        humanchain_endPosition))
73         break;
74
75     p2e = glm::normalize(humanchain_endPosition - limb3_position);
76     p2t = glm::normalize(cube_position - limb3_position);
77     angle = std::acos(std::max(std::min(glm::dot(p2e, p2t), 1.0f), -1.0f));
78
79     if (angle >= 0.005)
80     {
```

```
81     if (angle > clip_angle) angle = clip_angle;
82
83     rotAxis = glm::normalize(glm::cross(p2e, p2t));
84     rot = glm::angleAxis(angle, rotAxis);
85     limb3_quat = rot * limb3_quat;
86
87     trans = glm::toMat4(limb1_quat);
88     limb2_position = glm::mat3(trans) * glm::vec3(limb1_length, 0.0f, 0.0f
89 ) + limb1_position;
90     trans *= glm::toMat4(limb2_quat);
91     limb3_position = glm::mat3(trans) * glm::vec3(limb2_length, 0.0f, 0.0f
92 ) + limb2_position;
93     trans *= glm::toMat4(limb3_quat);
94     finger1_position = glm::mat3(trans) * glm::vec3(limb3_length, 0.0f,
95 0.0f) + limb3_position;
96     trans *= glm::toMat4(finger1_quat);
97     humanchain_endPosition = glm::mat3(trans) * glm::vec3(finger1_length,
98 0.0f, 0.0f) + finger1_position;
99 }
100 if (CCDOverCondition(cube_position, limb1_position, humanchain_length,
101 humanchain_endPosition))
102     break;
103
104 p2e = glm::normalize(humanchain_endPosition - limb2_position);
105 p2t = glm::normalize(cube_position - limb2_position);
106 angle = std::acos(std::max(std::min(glm::dot(p2e, p2t), 1.0f), -1.0f));
107
108 if (angle >= 0.005)
109 {
110     if (angle > clip_angle) angle = clip_angle;
111
112     rotAxis = glm::normalize(glm::cross(p2e, p2t));
113     rot = glm::angleAxis(angle, rotAxis);
114     limb2_quat = rot * limb2_quat;
115
116     trans = glm::toMat4(limb1_quat);
117     limb2_position = glm::mat3(trans) * glm::vec3(limb1_length, 0.0f, 0.0f
118 ) + limb1_position;
119     trans *= glm::toMat4(limb2_quat);
120     limb3_position = glm::mat3(trans) * glm::vec3(limb2_length, 0.0f, 0.0f
121 ) + limb2_position;
122     trans *= glm::toMat4(limb3_quat);
123     finger1_position = glm::mat3(trans) * glm::vec3(limb3_length, 0.0f,
124 0.0f) + limb3_position;
125     trans *= glm::toMat4(finger1_quat);
126     humanchain_endPosition = glm::mat3(trans) * glm::vec3(finger1_length,
127 0.0f, 0.0f) + finger1_position;
128 }
129 if (CCDOverCondition(cube_position, limb1_position, humanchain_length,
130 humanchain_endPosition))
```

```
121     break;
122
123     p2e = glm::normalize(humanchain_endPosition - limb1_position);
124     p2t = glm::normalize(cube_position - limb1_position);
125     angle = std::acos(std::max(std::min(glm::dot(p2e, p2t), 1.0f), -1.0f));
126
127     if (angle >= 0.005)
128     {
129         if (angle > clip_angle) angle = clip_angle;
130
131         rotAxis = glm::normalize(glm::cross(p2e, p2t));
132         rot = glm::angleAxis(angle, rotAxis);
133         limb1_quat = rot * limb1_quat;
134
135         trans = glm::toMat4(limb1_quat);
136         limb2_position = glm::mat3(trans) * glm::vec3(limb1_length, 0.0f, 0.0f
137 ) + limb1_position;
138         trans *= glm::toMat4(limb2_quat);
139         limb3_position = glm::mat3(trans) * glm::vec3(limb2_length, 0.0f, 0.0f
140 ) + limb2_position;
141         trans *= glm::toMat4(limb3_quat);
142         finger1_position = glm::mat3(trans) * glm::vec3(limb3_length, 0.0f,
143 0.0f) + limb3_position;
144         trans *= glm::toMat4(finger1_quat);
145         humanchain_endPosition = glm::mat3(trans) * glm::vec3(finger1_length,
146 0.0f, 0.0f) + finger1_position;
147     }
148 }
```

4 Required feature 3: Scripted animation

Cubic spline was used to created scripted animation. I specified 4 points to made a cubic spline. In addition, ease-in ease-out was used to make the motion better. For the ease-in ease-out effect, I made use of arctan function, because arctan function changes slowly when the absolute value of input is large, while changes fast when the input value is near to zero. This feature is suitable to make ease-in ease-out motion. Figure 3 are 4 screenshots at different time. You can check the YouTube video (the link is in Section 1) for the whole animation. The following code is how I calculated cubic spline and made ease-in ease-out effect.

```
1
2 // create basis matrix
3 glm::mat4 B(
4     0.0f, 1.0f, 8.0f, 27.0f,
5     0.0f, 1.0f, 4.0f, 9.0f,
6     0.0f, 1.0f, 2.0f, 3.0f,
7     1.0f, 1.0f, 1.0f, 1.0f
8 );
```

```
9
10 B = glm::inverse(glm::transpose(B)); // transpose is because the order in
   OpenGl is already transposed
11
12 // create geometry matrix
13 glm::vec4 spline_p1(1.0f, 5.5f, -1.0f, 0.0f);
14 glm::vec4 spline_p2(2.0f, 3.f, 1.0f, 0.0f);
15 glm::vec4 spline_p3(3.0f, 5.5f, -1.0f, 0.0f);
16 glm::vec4 spline_p4(4.0f, 3.0f, 0.5f, 0.0f);
17 glm::mat4 G(spline_p1, spline_p2, spline_p3, spline_p4);
18
19 std::vector<glm::vec3> splinePoints;
20
21
22 // use arctan function to map the time
23 for (int i = 0; i <= 30; i++)
24 {
25     float time = (std::atan(5 * (0.1 * i - 1.5)) + M_PI / 2) / M_PI * 3.0;
26     glm::vec4 t(std::pow(time, 3), std::pow(time, 2), time, 1);
27     glm::vec3 p = glm::vec3(G * B * t);
28     splinePoints.push_back(p);
29
30     std::cout << p.x << " " << p.y << " " << p.z << std::endl;
31 }
32
33 for (int i = 29; i >= 0; i--)
34 {
35     float time = (std::atan(5 * (0.1 * i - 1.5)) + M_PI / 2) / M_PI * 3.0;
36     glm::vec4 t(std::pow(time, 3), std::pow(time, 2), time, 1);
37     glm::vec3 p = glm::vec3(G * B * t);
38     splinePoints.push_back(p);
39
40     std::cout << p.x << " " << p.y << " " << p.z << std::endl;
41 }
42
43 while (!glfwWindowShouldClose(window))
44 {
45     float time = glfwGetTime();
46     cube_position = splinePoints[int(time * 10) % 61];
47     ...
48 }
```

Name: Jing Wang
ID: 21330849

Assignment 2

Module: CS7GV5
February 25, 2022

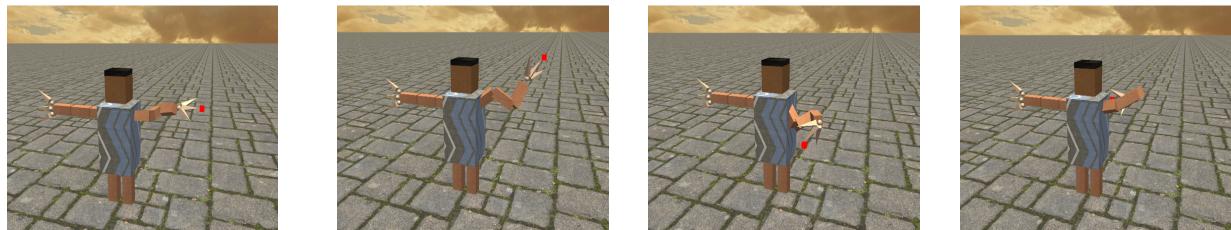


Figure 3: Scripted animation using cubic spline. IK is using CCD.