Name: Jing Wang
ID: 21330849

Assignment 2

Module: CS7GV5
February 25, 2022

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.

```
// when the target is over the distance the two bones can reach to
if (std::sqrt(positionX * positionX + positionY * positionY) > (bone1_length + bone2_length))
{
  if (positionY > 0) bone1_angle = std::acos(positionX / std::sqrt(positionX * positionX + positionY * positionY));
  else bone1_angle = -std::acos(positionX / std::sqrt(positionX * positionX + positionY * positionY));

  bone2_angle = 0;
}
// when the target is inside the area the two bones can reach to
else if (std::sqrt(positionX * positionX + positionY * positionY) < std::abs(bone1_length - bone2_length))
{
  if (positionY > 0) bone1_angle = std::acos(positionX / std::sqrt(positionX * positionX + positionY * positionY));
  else bone1_angle = -std::acos(positionX / std::sqrt(positionX * positionX + positionY * positionY));

  bone2_angle = M_PI;
}
// to avoid the bones flip when go beyond x_axis or y_axis, devide them to 2 cases
else if (positionY > 0)
{
  angle_T = std::acos(positionX / std::sqrt(positionX * positionX + positionY * positionY));
  angle_1 = std::acos((bone1_length * bone1_length + positionX * positionX + positionY * positionY - bone2_length * bone2_length) / (2 * bone1_length * std::sqrt(positionX * positionX + positionY * positionY)));
  bone1_angle = angle_1 + angle_T;
  bone2_angle = -std::acos(-(bone1_length * bone1_length + bone2_length * bone2_length - positionX * positionX - positionY * positionY) / (2 * bone1_length * bone2_length));
}
else
{
```

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

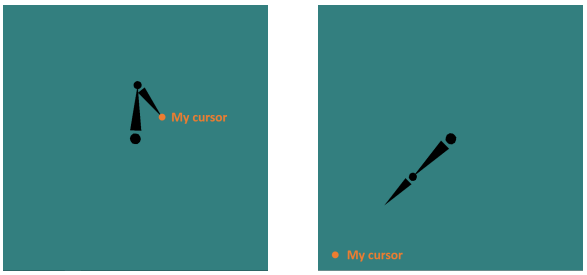

Figure 1: IK on 2 bones in 2D (analytic method).

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.

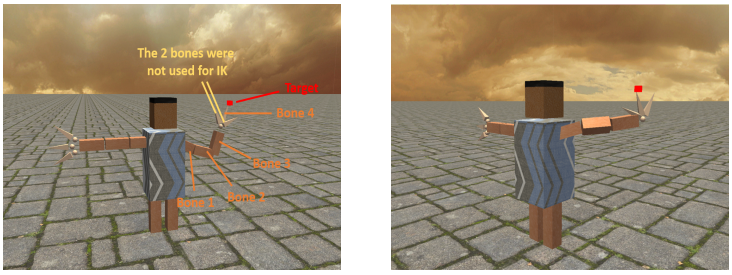

Figure 2: IK on multiple bones in 3D (CCD).

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.

```
bool CCDoverCondition(glm::vec3 _target_position, glm::vec3 _limb1_position, float _humanchain_length, glm::vec3 _humanchain_endPosition)
{
  bool over = false;

  if (glm::length(_target_position - _limb1_position) <= 0.95 * _humanchain_length)
  {
    if (_humanchain_length && glm::length(_target_position - _humanchain_endPosition) < 0.1)
      over = true;
  }
  else if (glm::length(_target_position - _limb1_position) <= _humanchain_length)
  {
    if (_humanchain_length && glm::length(_target_position - _humanchain_endPosition) < 0.2)
      over = true;
  }
  else
  {
    glm::vec3 p2e = _humanchain_endPosition - _limb1_position;
    glm::vec3 p2t = _target_position - _limb1_position;
    glm::vec3 e2t = _target_position - _humanchain_endPosition;
    float angle = std::acos(glm::dot(glm::normalize(p2e), glm::normalize(p2t)));

    if (angle < 0.01 && glm::length(e2t) < 0.5)
      over = true;
  }

  return over;
}

// at most update 5000 iterations per frame
int counter = 5000;
float clip_angle = 0.05;

trans = glm::toMat4(limb1_quat);
limb2_position = glm::mat3(trans) * glm::vec3(limb1_length, 0.0f, 0.0f) + limb1_position;
trans *= glm::toMat4(limb2_quat);
limb3_position = glm::mat3(trans) * glm::vec3(limb2_length, 0.0f, 0.0f) + limb2_position;
trans *= glm::toMat4(limb3_quat);
finger1_position = glm::mat3(trans) * glm::vec3(limb3_length, 0.0f, 0.0f)
```

```
    + limb3_position;
trans *= glm::toMat4(finger1_quat);
humanchain_endPosition = glm::mat3(trans) * glm::vec3(finger1_length, 0.0f
    , 0.0f) + finger1_position;

// calculate the postition and orientation of the bones one by one
while (--counter)
{
  if (CCDoverCondition(cube_position, limb1_position, humanchain_length,
    humanchain_endPosition))
     break;

  glm::vec3 p2e;
  glm::vec3 p2t;
  float angle;
  p2e = glm::normalize(humanchain_endPosition - finger1_position);
  p2t = glm::normalize(cube_position - finger1_position);
  angle = std::acos(std::max(std::min(glm::dot(p2e, p2t), 1.0f), -1.0f));

  if (angle >= 0.005)
  {
    if (angle > clip_angle) angle = clip_angle;

    rotAxis = glm::normalize(glm::cross(p2e, p2t));
    rot = glm::angleAxis(angle, rotAxis);
    finger1_quat = rot * finger1_quat;

    trans = glm::toMat4(limb1_quat);
    limb2_position = glm::mat3(trans) * glm::vec3(limb1_length, 0.0f, 0.0f
    ) + limb1_position;
    trans *= glm::toMat4(limb2_quat);
    limb3_position = glm::mat3(trans) * glm::vec3(limb2_length, 0.0f, 0.0f
    ) + limb2_position;
    trans *= glm::toMat4(limb3_quat);
    finger1_position = glm::mat3(trans) * glm::vec3(limb3_length, 0.0f,
    0.0f) + limb3_position;
    trans *= glm::toMat4(finger1_quat);
    humanchain_endPosition = glm::mat3(trans) * glm::vec3(finger1_length,
    0.0f, 0.0f) + finger1_position;
  }
  if (CCDoverCondition(cube_position, limb1_position, humanchain_length,
    humanchain_endPosition))
     break;

  p2e = glm::normalize(humanchain_endPosition - limb3_position);
  p2t = glm::normalize(cube_position - limb3_position);
  angle = std::acos(std::max(std::min(glm::dot(p2e, p2t), 1.0f), -1.0f));

  if (angle >= 0.005)
  {
```

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

```
    break;

  p2e = glm::normalize(humanchain_endPosition - limb1_position);
  p2t = glm::normalize(cube_position - limb1_position);
  angle = std::acos(std::max(std::min(glm::dot(p2e, p2t), 1.0f), -1.0f));

  if (angle >= 0.005)
  {
    if (angle > clip_angle) angle = clip_angle;

    rotAxis = glm::normalize(glm::cross(p2e, p2t));
    rot = glm::angleAxis(angle, rotAxis);
    limb1_quat = rot * limb1_quat;

    trans = glm::toMat4(limb1_quat);
    limb2_position = glm::mat3(trans) * glm::vec3(limb1_length, 0.0f, 0.0f
   ) + limb1_position;
    trans *= glm::toMat4(limb2_quat);
    limb3_position = glm::mat3(trans) * glm::vec3(limb2_length, 0.0f, 0.0f
   ) + limb2_position;
    trans *= glm::toMat4(limb3_quat);
    finger1_position = glm::mat3(trans) * glm::vec3(limb3_length, 0.0f,
   0.0f) + limb3_position;
    trans *= glm::toMat4(finger1_quat);
    humanchain_endPosition = glm::mat3(trans) * glm::vec3(finger1_length,
   0.0f, 0.0f) + finger1_position;
  }
}
```

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.

```

// create basis matrix
glm::mat4 B(
    0.0f, 1.0f, 8.0f, 27.0f,
    0.0f, 1.0f, 4.0f, 9.0f,
    0.0f, 1.0f, 2.0f, 3.0f,
    1.0f, 1.0f, 1.0f, 1.0f
  );
```

```

B = glm::inverse(glm::transpose(B)); // transpose is because the order in
    Opengl is already transposed

// create geometry matrix
glm::vec4 spline_p1(1.0f, 5.5f, -1.0f, 0.0f);
glm::vec4 spline_p2(2.0f, 3.f, 1.0f, 0.0f);
glm::vec4 spline_p3(3.0f, 5.5f, -1.0f, 0.0f);
glm::vec4 spline_p4(4.0f, 3.0f, 0.5f, 0.0f);
glm::mat4 G(spline_p1, spline_p2, spline_p3, spline_p4);

std::vector<glm::vec3> splinePoints;


// use arctan function to map the time
for (int i = 0; i <= 30; i++)
{
  float time = (std::atan(5 * (0.1 * i - 1.5)) + M_PI / 2) / M_PI * 3.0;
  glm::vec4 t(std::pow(time, 3), std::pow(time, 2), time, 1);
  glm::vec3 p = glm::vec3(G * B * t);
  splinePoints.push_back(p);

  std::cout << p.x << " " << p.y << " " << p.z << std::endl;
}

for (int i = 29; i >= 0; i--)
{
  float time = (std::atan(5 * (0.1 * i - 1.5)) + M_PI / 2) / M_PI * 3.0;
  glm::vec4 t(std::pow(time, 3), std::pow(time, 2), time, 1);
  glm::vec3 p = glm::vec3(G * B * t);
  splinePoints.push_back(p);

  std::cout << p.x << " " << p.y << " " << p.z << std::endl;
}

while (!glfwWindowShouldClose(window))
{
    float time = glfwGetTime();
  cube_position = splinePoints[int(time * 10) % 61];
    ...
}
```

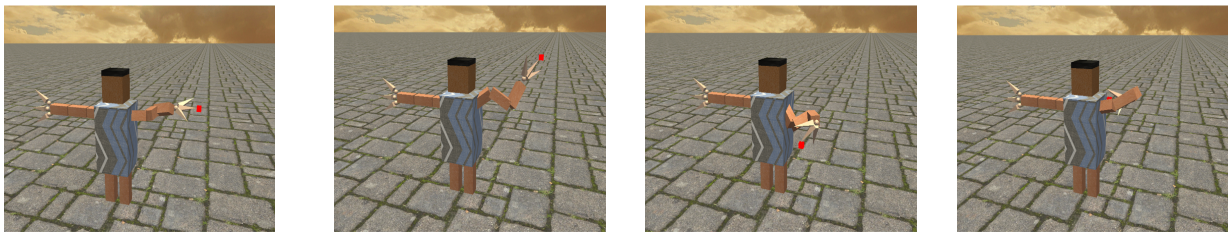

Figure 3: Scripted animation using cubic spline. IK is using CCD.