University of Dublin

TRINITY COLLEGE

Animation Inverse kinematic

Student Name: Hao Guan
Student Number: 14305641
Submission Date: 13/11/2014
Course Module: Real-time Animation

School of Computer Science and Statistics

O'Reilly Institute, Trinity College, Dublin 2, Ireland

Summary

The goal of this Inverse kinematic lab is to develop a human full arm and hand with bone structure to simulate Inverse kinematic base on the CCD algorithm, and few feature was added as extra, such as joint limitation, hand grabbing, manual control and cubic spline interpolation with keyframe.

Code explanation

Bone class:

```
class Bone
{
public:
    Bone(void);
    ~Bone(void);

    int id;
    glm::vec3 pos;
    glm::mat4 offset;
    glm::mat4 localTransformation;
    glm::mat4 globalTransformation;
    Bone* children[1];
    Bone* parent;
    float damp_width;
    int chilrenSize;
    float max_rx;
    float min_rx;
    float max_ry;
    float min_ry;
    float max_rz;
    float min_rz;
    float currentXPos;
    float currentYPos;
    float currentZPos;

    Bone* createBone(int ID, glm::vec3 pos, float damp_width);
    void addChild(Bone* child);
    void addParent(Bone* parent);
    int getID(Bone* bone);
    glm::vec3 getPos(Bone* bone);
    void setJointLimit(float max_rx, float min_rx, float max_ry, float min_ry, float max_rz, float min_rz);
};
```

For this lab, the new maximum rodation on each axis is added, also the current bone roation degree of each axis, this will allow calclate the each joint limition.

```
void Bone::setJointLimit(float max_rx, float min_rx, float max_ry, float min_ry, float max_rz, float min_rz)
{
    this->max_rx = max_rx;
    this->min_rx = min_rx;
    this->max_ry = max_ry;
    this->min_ry = min_ry;
    this->max_rz = max_rz;
    this->min_rz = min_rz;
}
```

```
    handNode[0]->setJointLimit(10.0f,-179.0f,90.0f,-10.0f,45.0f,-179.0f);
    handNode[1]->setJointLimit(135.0f,0.0f,90.0f,0.0f,0.0f,0.0f);
    handNode[2]->setJointLimit(75.0f,-75.0f,0.0f,0.0f,45.0f,-45.0f);
```

Joint limitation for upper arm, lower arm and wrist.

Skeleton class:

```
class ArmSkeleton
{
public:
    ArmSkeleton(Setup* m_setup);
    ~ArmSkeleton(void);

    Bone* bone;
    Bone* handNode[19];
    Cylinder* cylinder[19];
    Cylinder* armTarget;
    Setup* m_setup;

    double lastTime;
    double currentTime;
    float deltaTime;
    bool timeFlag;
    float whatsoever;
    float speed;
    glm::vec3 armTargetPos;
    glm::mat4 armTargetTransformation;
    float joint2TargetPos;
    float endToTargeDistance;
    float currentXPos,currentYPos,currentZPos;
    float DOF_x,DOF_y,DOF_z;
    float deltaT;
    bool targetLoops;


    void createArmNode();
    void drawArmMesh(GLuint shaderProgramID);
    void updateArmMesh(GLuint shaderProgramID);
    void updateArmTarget(GLuint shaderProgramID);
    void calculateInverseKinematics();
    void calcGlobalTransformation();
    glm::quat calcJointLimit(Bone* bone, glm::vec3 angles);
    void calcEffectorToTargetDistance();
    glm::vec3 interpolateCubic(float deltaTime, glm::vec3 beingPos, glm::vec3 point1,  glm::vec3 point2, glm::vec3 endPos);
};
```

In skeleton class I have implement few more new functions, such as calculate IK with CCD, calculate joint limitation, calculate the effector to target distance and cubic spline interpolation.

The code for calculate IK with CCD steps as follow:

Get the position from effector position, current bone position and target position. From this 3 position we can get the cos angel from dot product, after that we will get the rotation from cross product, once we get both of them, we can rotation quaternion and then transfer to matrix to apply to bone local transformation.

Calculate IK with CCD

```
void ArmSkeleton::calculateInverseKinematics()
{
    int NumConter,linker;

    NumConter = 0;
    linker = handNode[2]->id;
    while (endToTargeDistance > IK_POS_THRESH && NumConter < MAX_IK_TRIES)
    {
        glm::vec3 effectorPos = glm::vec3(handNode[3]->globalTransformation[3][0],
            handNode[3]->globalTransformation[3][1], handNode[3]->globalTransformation[3][2]);
        glm::vec3 bonePos = glm::vec3(handNode[linker]->globalTransformation[3][0],
            handNode[linker]->globalTransformation[3][1], handNode[linker]->globalTransformation[3][2]);

        glm::vec3 endVector = effectorPos - bonePos;
        glm::vec3 endVectorNor = glm::normalize(endVector);

        glm::vec3 targetVector = armTargetPos - bonePos;
        glm::vec3 targetVectorNor = glm::normalize(targetVector);

        float cosAngle = glm::dot(targetVectorNor, endVectorNor);

        if ((endVectorNor != endVectorNor)||(targetVectorNor != targetVectorNor))
        {
            break;
        }

        if (cosAngle >= 1)
        {
            break;
        }

        if (cosAngle <= -1)
        {
            break;
        }

        glm::vec3 crossResult = glm::cross(endVectorNor, targetVectorNor);
        crossResult = glm::normalize(crossResult);
        crossResult = glm::vec3(glm::mat3(glm::inverse(
            handNode[linker]->globalTransformation)) * crossResult);

        float turnAngle = glm::acos(cosAngle);  // GET THE ANGLE
        float turnDeg = glm::degrees(turnAngle); // COVERT TO DEGREES

        glm::quat eulerRot = glm::angleAxis(turnDeg, crossResult);
        glm::vec3 angles = glm::eulerAngles(eulerRot);
        eulerRot = calcJointLimit(handNode[linker], angles);
        glm::mat4 rot =  glm::toMat4(eulerRot);

        handNode[linker]->localTransformation *= rot;

        calcGlobalTransformation();
        calcEffectorToTargetDistance();

        NumConter++;
        linker--;
        if (linker < 0)
        {
            linker = handNode[2]->id;
        }
    }
    calcEffectorToTargetDistance();
}
```

Calculate joint limitation

```
        glm::vec3 angles = glm::eulerAngles(eulerRot);
        eulerRot = calcJointLimit(handNode[linker], angles);
```

```
glm::quat ArmSkeleton::calcJointLimit(Bone* bone, glm::vec3 angles)
{
    bool fx = false;
    bool fy = false;
    bool fz = false;
    if (angles.x > 0 && bone->currentXPos == bone->max_rx)
    {
        angles.x = 0;
        fx = true;
    }
    if (angles.y > 0 && bone->currentYPos == bone->max_ry)
    {
        angles.y = 0;
        fy = true;
    }
    if (angles.z > 0 && bone->currentZPos == bone->max_rz)
    {
        angles.z = 0;
        fz = true;
    }

    if (angles.x < 0 && bone->currentXPos == bone->min_rx)
    {
        angles.x = 0;
        fx = true;
    }
    if (angles.y < 0 && bone->currentYPos == bone->min_ry)
    {
        angles.y = 0;
        fy = true;
    }
    if (angles.z < 0 && bone->currentZPos == bone->min_rz)
    {
        angles.z = 0;
        fz = true;
    }
```

If the current angles equals maximum rotation axis, 0 degree will be return, as this no rotation allow in this case.

```cpp
    if((angles.x + bone->currentXPos) <= bone->max_rx && (angles.x + bone->currentXPos) >= bone->min_rx && fx == false)
    {
        bone->currentXPos = bone->currentXPos +  angles.x;
    }
    else
    {
        bool swicher = false;
        if ((angles.x + bone->currentXPos) > bone->max_rx)
        {
            angles.x = bone->max_rx - bone->currentXPos;
            bone->currentXPos = bone->max_rx;
            swicher = true;
        }

        if ((angles.x + bone->currentXPos) < bone->min_rx && swicher == false)
        {
            angles.x = bone->min_rx - bone->currentXPos;
            bone->currentXPos = bone->min_rx;
        }
    }
```

If the angle plus current angle is in the limitation range, current position will replaced with new degree, and the angle for rotation is given a new suitable value base on the case of over the maximum of less than minimum degree.

```cpp
    if((angles.y + bone->currentYPos) <= bone->max_ry && (angles.y + bone->currentYPos) >= bone->min_ry && fy == false)
    {
        bone->currentYPos = bone->currentYPos + angles.y;
    }
    else
    {
        bool swicher = false;
        if ((angles.y + bone->currentYPos) > bone->max_ry)
        {
            angles.y = bone->max_ry - bone->currentYPos;
            bone->currentYPos = bone->max_ry;
            swicher = true;
        }


        if ((angles.y + bone->currentYPos) < bone->min_ry && swicher == false)
        {
            angles.y = bone->min_ry - bone->currentYPos;
            bone->currentYPos = bone->min_ry;
        }
    }

    if((angles.z + bone->currentZPos) <= bone->max_rz && (angles.z + bone->currentZPos) >= bone->min_rz && fz == false)
    {
        bone->currentZPos = bone->currentZPos + angles.z;
    }
    else
    {
        bool swicher = false;
        if ((angles.z + bone->currentZPos) > bone->max_rz)
        {
            angles.z = bone->max_rz - bone->currentZPos;
            bone->currentZPos = bone->max_rz;
            swicher = true;
        }

        if ((angles.z + bone->currentZPos) < bone->min_rz && swicher == false)
        {
            angles.z = bone->min_rz - bone->currentZPos;
            bone->currentZPos = bone->min_rz;
        }
    }

    angles = glm::radians(angles);

    return glm::quat(angles);
}
```

And do the same for y and x axis, after the new quaternion returned.

Calculate the effector to target distance

```
void ArmSkeleton::calcEffectorToTargetDistance()
{
    endToTargeDistance = glm::distance(glm::vec3(handNode[3]->globalTransformation[3][0],
        handNode[3]->globalTransformation[3][1], handNode[3]->globalTransformation[3][2]), armTargetPos);
}
```

Calculate hand grabbing based on effector to target distance, also update the global transformation after that.

```
void ArmSkeleton::calcGlobalTransformation()
{
    if (endToTargeDistance < 2.0f)
    {
        for (int i = 4; i < 19; i++)
        {
            handNode[i]->localTransformation = glm::rotate(handNode[i]->offset, deltaTime, glm::vec3(1,0,0));
        }
        deltaTime += 0.1f;
        if (deltaTime > 40.0f)
        {
            deltaTime = 30.0f;
        }
    }

    if (endToTargeDistance >= 2.0f)
    {
        for (int i = 4; i < 19; i++)
        {
            handNode[i]->localTransformation = glm::rotate(handNode[i]->offset, deltaTime, glm::vec3(1,0,0));
        }
        deltaTime -= 0.1f;
        if (deltaTime < 0.0f)
        {
            deltaTime = 0.0f;
        }
    }

    handNode[16]->localTransformation = glm::rotate(handNode[16]->offset, 25.0f, glm::vec3(0,0,-1));

    for (int i = 0 ; i < HAND_NODE_NUM ; i++)
    {
        if (handNode[i]->id == 0)
        {
            handNode[i]->globalTransformation = handNode[i]->localTransformation;
        }
        else
        {
            handNode[i]->globalTransformation = handNode[i]->parent->globalTransformation * handNode[i]->localTransformation;
        }
    }
}
```

Cubic spline interpolation with keyframe

```
glm::vec3 ArmSkeleton::interpolateCubic(float deltaTime, glm::vec3 beingPos, glm::vec3 point1,
                                        glm::vec3 point2, glm::vec3 endPos)
{
    glm::vec3 p1,p2,p3,p4;

    p1 = point2 - endPos - point1 + beingPos;
    p2 = point1 - beingPos - p1;
    p3 = endPos - point1;
    p4 = beingPos;

    return (p1 * pow(deltaTime,3) + p2 * pow(deltaTime,2) + p3 * pow(deltaTime,1) + p4);
}

glm::vec3 ArmSkeleton::interpolateCubic_2(float deltaTime, glm::vec3 beingPos,
                                          glm::vec3 point1,  glm::vec3 point2, glm::vec3 endPos)
{
    return pow(1-deltaTime,3)*beingPos+3*deltaTime*pow(1-deltaTime,2)*point1+3*
        pow(deltaTime,2)*(1-deltaTime)*point2+pow(deltaTime,3)*endPos;
}
```

And the following is when I calling this functions to draw two paths.

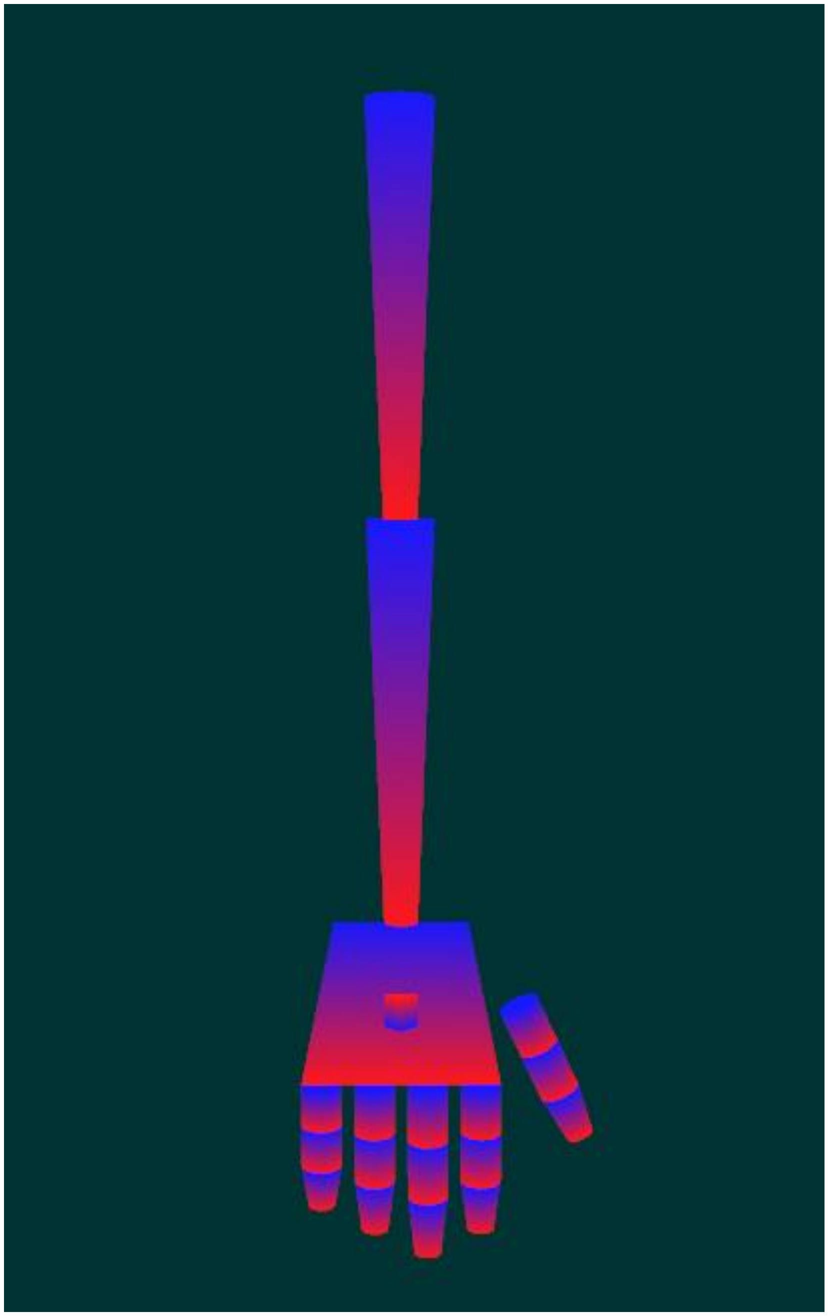
```
    if (targetLoops == false)
    {
        armTargetPos = interpolateCubic(deltaT, glm::vec3(20,50,15), glm::vec3(130,0,15),
            glm::vec3(-130,0,15), glm::vec3(-30,10,10));

        if (armTargetPos == glm::vec3(-30,10,10))
        {
            targetLoops = true;
        }
    }

    if (targetLoops == true)
    {
        armTargetPos = interpolateCubic(deltaT, glm::vec3(-30,10,10), glm::vec3(-20,-10,10),
            glm::vec3(15,-20,-10), glm::vec3(20,50,15));

        if (armTargetPos == glm::vec3(20,50,15))
        {
            targetLoops = false;
        }
    }
```

Manual control for the target

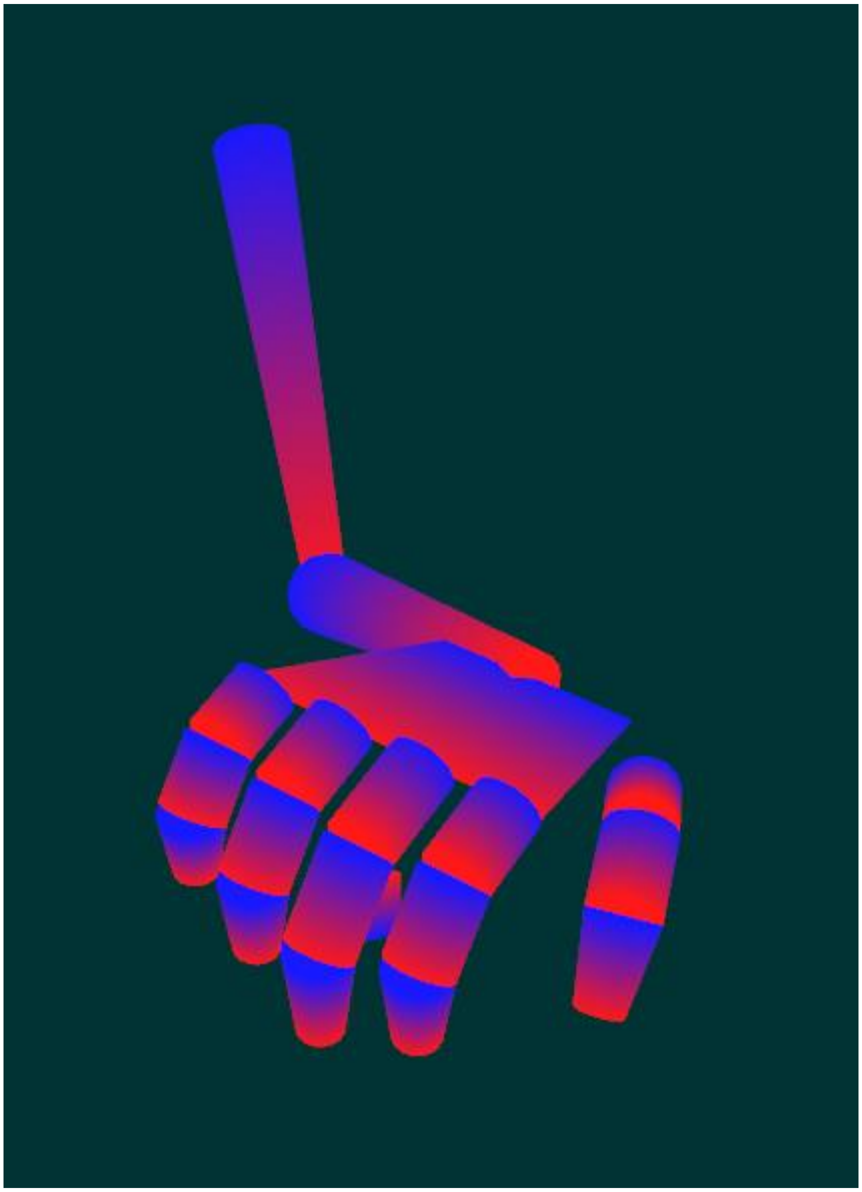
```cpp
void ArmSkeleton::updateArmTarget(GLuint shaderProgramID)
{
    //Move y up
    if (glfwGetKey( m_setup->getWindow(), GLFW_KEY_UP ) == GLFW_PRESS){
        armTargetPos.y += 0.3;
    }

    // Move y down
    if (glfwGetKey( m_setup->getWindow(), GLFW_KEY_DOWN ) == GLFW_PRESS){
        armTargetPos.y -= 0.3;
    }

    // Move x right
    if (glfwGetKey( m_setup->getWindow(), GLFW_KEY_RIGHT ) == GLFW_PRESS){
        armTargetPos.x += 0.3;
    }

    // Move x left
    if (glfwGetKey( m_setup->getWindow(), GLFW_KEY_LEFT ) == GLFW_PRESS){
        armTargetPos.x -= 0.3;
    }

    // Move z forward
    if (glfwGetKey( m_setup->getWindow(), GLFW_KEY_I ) == GLFW_PRESS){
        armTargetPos.z -= 0.3;
    }

    // Move z backward
    if (glfwGetKey( m_setup->getWindow(), GLFW_KEY_K ) == GLFW_PRESS){
        armTargetPos.z += 0.3;
    }
```

```cpp
    armTargetTransformation = glm::translate(glm::mat4(1),glm::vec3(armTargetPos.x,armTargetPos.y-3,armTargetPos.z));
    armTarget->update(armTargetTransformation, shaderProgramID);
    armTarget->draw();
```

Hand model



Initial position



Hand grabbing

Hand grabbing in steps.