浙江大学

课程名称：计算机动画

姓　　名：刘佳润

学　　院：计算机科学与技术学院

专　　业：数字媒体技术

学　　号：3180105640

指导教师：金小刚

2021 年 1 月 9 日

浙江大学实验报告

课程名称：计算机动画 实验类型：综合

实验项目名称：基于循环坐标下降法的三维反向运动学模型

学生姓名：刘佳润 专业：数字媒体技术 学号：3180105640

同组学生姓名：无 指导老师：金小刚

实验地点：玉泉 实验日期：2021 年 1 月 9 日

一、 实验目的和要求

实现基于循环坐标下降法（CCD）的反向运动学（IK）求解。在学习和编程的过程中体会 IK 的方法与思想以及 CCD 方法的具体内容和优缺点，能够实现简单的 demo。

二、 实验内容和原理

实验内容：

1、进行三维建模模拟骨骼链以及关节动画；

2、指定目标点，使用循环坐标下降法控制骨骼的末端影响器到达目标点；

3、生成中间帧，得到一个比较完整流畅的关节动画。

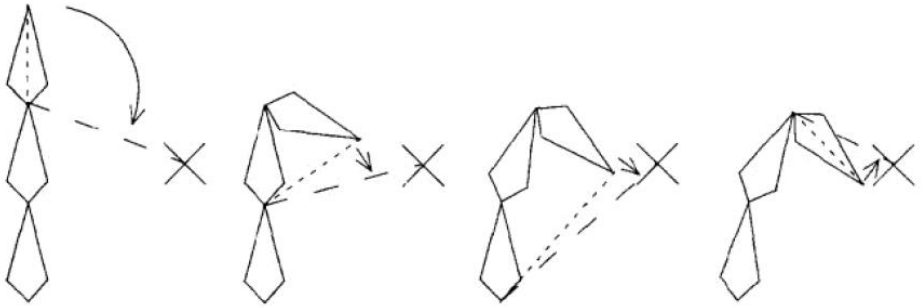
实验算法流程：

1、 从最末端的一节骨骼 B_n 开始，通过几何关系判断与调整，使得末端影响器 E 到 B_n 骨骼节的始端的连线所在直线通过目标点 T；

2、 回到该骨骼节的父骨骼 B_{n-1}，通过几何关系判断与调整，使得末端影响器 E 到 B_{n-1} 骨骼节的始端的连线所在直线通过目标点 T；

3、 以此类推，直到到达根骨骼 B_1，同样进行上述操作；

4、 在迭代过程中始终判断末端影响器 E 是否到达目标点 T。整个过程可能需要多次迭代以得到稳定的结果。



三、 实验器材

编译环境：VS2019 Release x86

编程语言：C++

调用开源库：OpenGL 相关（glew 库、freeglut 库、glm 数学库）、STL 库

四、 实验步骤

【本实验参考了一部分 OpenGL 场景搭建的内容，核心算法由自己编写与改善】

1、 骨骼对象的设定

定义了一个骨骼类 Bone，每个对象代表一段骨骼节。通过 Bone 之间的指针连结成的链状结构来表示整个骨骼链。在 Bone 类里对对象骨骼节的旋转角度限制以及旋转矩阵（用于进行旋转变换）。下图展示部分 Bone 类的成员变量。

```
class Bone {
public:
    Bone* parent;                   // 父骨骼
    std::vector<Bone*> bones;       // 骨骼组合
    float length;                   // 骨骼长度

    glm::mat4 M;                    // 变换矩阵
    glm::vec3 rotation;             // 当前旋转角

    glm::vec3 coordinates;          // 当前坐标
    glm::vec3 constraint[2];        // 角度限制，分别是最小和最大
```

在获取骨骼的时候，这份参考的代码用了一种很巧妙的思路：用一个多位

数来表示末端影响器，用它的每一个数位来表示对应层次的骨骼节。“1”代表根骨骼，以此类推。比如“231”顺序意味着顺序为“根骨骼-第三节骨骼-第二节骨骼”。这样，调用末端影响器的数字序列，我们就可以判断需要操作哪根骨骼进行旋转。那么对于 CCD 算法，我们从末端影响器逐级向上迭代的话，只需要设置每一位都是“1”即可。

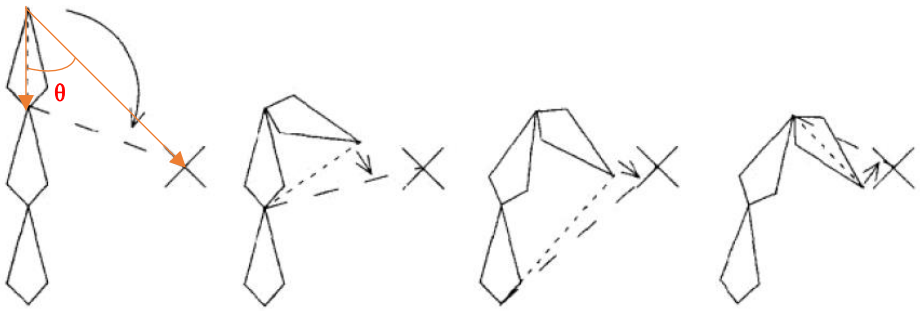
```
/**
* Find Bone using its unique identifier. On every joint there
* may be maximum 8 bones identified using this method. Should
* be used once in the while to obtain the pointer to the bone.
*
* The rightmost digit represents the current joint index
* (i.e. 1 will return first bone, 4 will return 4th bone - vector index 3).
* If id is longer then one digit, it is read recusively
* from right to left (i.e. 31 will return 3rd bone on 1st bone joint,
* 231 will return 2nd bone on previously mentioned bone joint).
*
* @param id Bone identifier (every digit must be [1:9])
* @return Bone
*/
// 根据末端影响器的编号来找到骨骼
Bone* bone(unsigned long long id);

unsigned long long id();
```

```
////////////////////////////////////////////////////
Bone* Bone::bone(unsigned long long id) {
    if (id < 10) {
        // 一位数——直接返回对应的骨骼
        return bones[id-1];
    }
    else {
        // 两位数——按数位取
        return bones[id % 10 - 1]->bone(id/10);
    }
}
////////////////////////////////////////////////////

unsigned long long Bone::id() {
    if (parent == NULL)
        return 0;
    else
        return parent->id()*10 + (std::find(parent->bones.begin(), parent->bones.end(), this) - parent->bones.begin()) + 1;
}
```

2、 核心算法：循环坐标下降法 CCD

CCD 的核心代码是一个双层循环。在判断尚未到达目标点并且在限定的迭代次数之内时，进行 CCD 的内循环——从末端影响器所在的骨骼节直到根骨骼为一次迭代。具体的思路在上面已经阐释。

实现细节：如图所示，通过两个归一化向量的点积得到图中所示的夹角，即为当前骨节需要旋转的角度 θ



因为是在三维空间上，所以需要构造四元数来进行三维旋转，并转换到欧拉角，存储三个方向的旋转分量，作为当前骨骼的最大旋转角。下面给出核心算法代码：

```
void ccd(Bone* endEffector, vec3 target, int iterations=3000)
{
    bool found = false;
    while (!found && iterations--)                  // 未找到且在迭代次数（1000？）之内
    {                                               // 开始迭代
        Bone* currentBone = endEffector;            // 当前骨骼——末端影响器
        while (currentBone->parent != NULL)         // 当前骨骼不是根
        {
            vec4 endPosition = endEffector->getEndPosition();               // 得到骨骼链的末端
            vec4 currentEndPosition = currentBone->getEndPosition();        // 得到当前骨骼的末端
            vec4 startPosition = currentBone->parent->getEndPosition();     // 得到当前骨骼的前端（即父骨骼的末端）
            vec3 toTarget = normalize(vec3(target.x - startPosition.x, target.y - startPosition.y, target.z - startPosition.z));              // 得到骨骼前端到目标点的向量——模长归一化
            vec3 toEnd = normalize(vec3(endPosition.x - startPosition.x, endPosition.y - startPosition.y, endPosition.z - startPosition.z));   // 得到骨骼链前端到末端的向量——模长归一化

            float cosine1 = dot(toEnd, toTarget);   // 点乘两个单位向量得到夹角
            if (cosine1 < 1)                        // cos(0)=1，也就是如果该夹角在0~360度
            {
                vec3 crossResult = cross(toEnd, toTarget);      // 得到一个垂直于这两个向量的向量

                float angle = glm::angle(toTarget, toEnd);      // 得到这两个向量夹角的数值

                quat rotation = angleAxis(angle, crossResult);  // 构造四元数来进行三维旋转：旋转的角度即为angle，旋转的轴是垂直向量
                rotation = normalize(rotation);                 // 归一化四元数
                vec3 euler = glm::eulerAngles(rotation);        // 转化到欧拉角，存储三个方向的旋转分量
                currentBone->rotateMax(euler.x, euler.y, euler.z);  // 设置该骨骼的最大旋转限度
            }

            // 判断找到目标点
            vec3 temp = vec3(endEffector->getEndPosition());
            temp.x -= target.x;
            temp.y -= target.y;
            temp.z -= target.z;
            if (dot(temp, temp) < 0.0000001)            // 同向量点乘得到距离
            {
                found = true;
                cout << "reached" << endl;
            }
            // 迭代
            currentBone = currentBone->parent;
        }
    }
```

3、 动画逻辑

OpenGL 的动画逻辑是刷新帧。

设置了 Animation 动画类，用一个 map 映射的数据结构 sequence 来储存骨骼对应的旋转角，并用一个类似的数据结构 moved 来记录骨骼已经旋转的角度，用以控制动画的过程。

```
class Animation {
private:
    // map:指定骨骼--对应角度，id:帧序号
    std::vector< std::map<unsigned long long, glm::vec3> > sequence;
    // 已经移动的距离
    std::vector< std::map<unsigned long long, glm::vec3> > moved;
    // 帧序号
    int position;
```

计算动画帧的逻辑如下：

- 接收到寻找目标点的指令，调用 CCD 函数，更新骨骼的旋转角；
- 更新动画队列，从根骨骼开始迭代，每次更新 sequence 数据结构中的对应骨骼的旋转角度，并将该状态储存为一张关键帧（将整个 sequence 扩容，在后面添加一帧来记录骨骼的状态）
- 从头开始遍历 sequence，将变化记录到 moved 里，直到到达序列的末尾，此时更新了 moved 动画序列，设定 animation_running 为 true

```
case 'p':
    // 建一个临时骨骼
    Bone *b = new Bone(*root);
    // 用CCD函数设定好每一个骨节的旋转角度
    ccd(b->bone(effector), target);
    // 构建一个新的动画序列
    delete animation;
    animation = new Animation();
    // 计算位置并增加动画帧
    animation->set(root)->keyframe();       // 根的位置确定，并增加这一帧
    for (int z = effector; z != 0; z /= 10) {
        // 根据影响器决定每根骨骼要到达的位置
        // 扩展动画序列
        animation->set(b->bone(z), b->bone(z)->rotation)->keyframe();
    }
    // 删除临时骨骼
    delete b;
    // 更新动画序列，设定开始播放
    animation->start()->next();
    animation_running = true;

    break;
}
```

然后在 glutIdleFunc 函数中调用 nextFrame 函数来显示下一帧。即检测到开始播放动画的指令后，按照速度生成 frame，直到到达序列末尾。

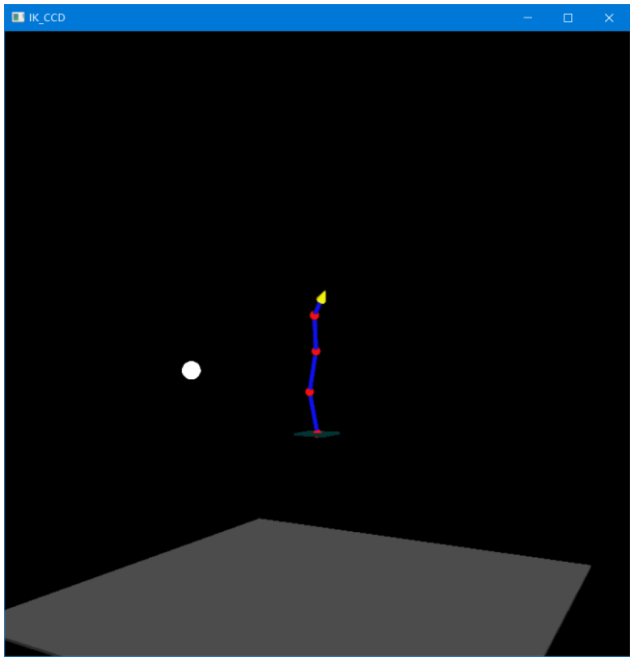
```
// 显示下一帧
void nextFrame(void) {
    //cout << "nextframe" << endl;
    int actTime = glutGet(GLUT_ELAPSED_TIME);
    float interval = actTime - lastTime;        // 时间间隔，其中lastTime初始化为0
    lastTime = actTime;
    angle_x += speed_x * interval / 500.0;      // 相机变化角度
    angle_y += speed_y * interval / 500.0;

    // 展示动画
    // 触发条件：在frame()函数中更新完动画序列之后，设置animation_running为true
    if (animation_running) {
        animation_fill += interval / 500.0;
        if (!animation->frame(interval / 500.0, root)) {
            // 当动画序列的下一帧生成失败，即停止
            animation_fill = 0.0;
            if (!animation->next()) {
                animation_running = false;
            }
        }
    }

    if (angle_x>360) angle_x -= 360;
    if (angle_x>360) angle_x += 360;
    if (angle_y>360) angle_y -= 360;
    if (angle_y>360) angle_y += 360;

    glutPostRedisplay();
}
```

五、 实验结果分析

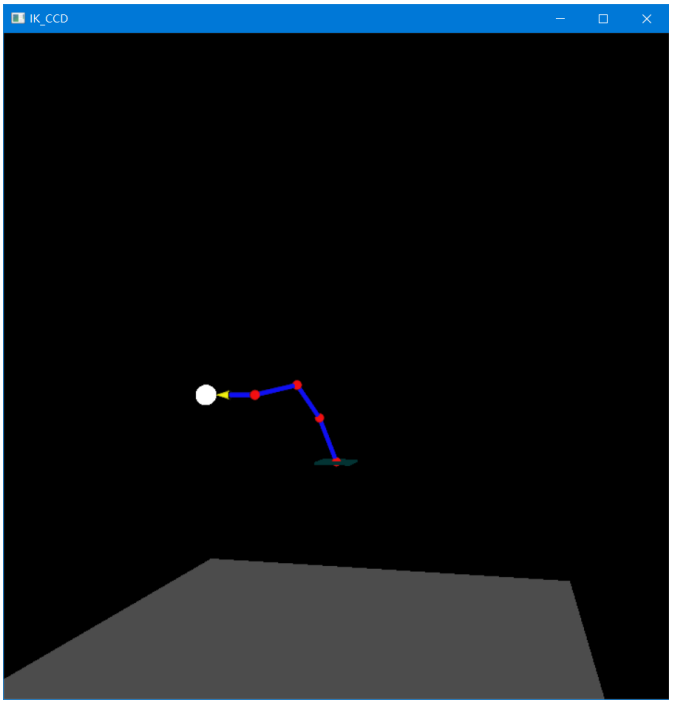
CCD 求解反向运动学是一种非常简便、迅速的计算方法，其优点显而易见——计算速度非常快，可以在很短的时间，比较少的迭代次数之内捕捉到目标点。但是其缺点也比较明显——一是会产生很多奇怪的中间帧，二是捕捉目标点的中间插值过程的状态与骨骼链的初始状态有很大关系。如果要解决这两个问题，需要对每一节骨骼的旋转角设定非常严格的限制。出于时间限制，本次实验仅仅模拟一个机械臂的寻找目标点过程，没有加旋转角限制。

另外，如果目标点超出了机械臂的可达到范围，机械臂仍会尽量地去找到该目标点，可能会在中间的迭代中出现多次远离目标点的尝试，但最终会形成一条指向目标点的直线并不再变动。

实验结果展示：



某一时刻的初始状态



找到目标点的最终状态

具体演示请参考 demo 视频。

六、 实验感想与心得

这次实验的动画播放类的编写和骨骼的末端影响器的编写思路是有所借鉴的，CCD 的算法和部分场景模型经过了自己的重写，动画播放逻辑也进行了调整。如果是从零开始搭建场景、编写基础部件的类和算法，我认为还是相当有难度的，也就是说“造轮子”的过程，其实还是最花时间的。光是为了适应参考工程中的基本部件与我的需要，这过程中不断 debug，改逻辑的工作量就蛮大的。反倒是把所有东西都考虑好以后，编写算法的思路只要正确，就会很顺畅。

实验是基于 OpenGL 的环境做的，因为去年图形学的课上曾经用过 glew 和 glm 数学库搭建过非常大的场景渲染，甚至还自己尝试写过 glsl，所以不算陌生。一些旋转矩阵的计算和旋转角的计算，借助 glm 的函数都可以完成。由于考虑到实现的是一个三维的 IK-CCD 场景，所以在构建旋转角的时候查阅了很多资料，如何来表示一个刚体的空间旋转角。之前在图形学实验中曾经使用过 Euler angle（欧拉角），但是通过单个参数来构建实在是感觉很奇怪，于是最终借助了之前

于老师讲过的四元数的方法，用 glm 的 Quaternion 类构造一个旋转四元数，再转换到欧拉角，解决了这个细节问题。

另外一个很有趣的 debug 细节——在用 glm::angle()函数从余弦值反推角度的时候，可能出现由于 C 的寄存器问题导致的“1 的溢出”，也就是本身 0° 的角计算出的余弦值为 1，但是由于内部储存的问题，变成了 1.000000001，导致这个角反求出来的结果是-Nan(ind)。后来加了限制条件解决了这个 bug，但是在找出这个 bug 的过程中费了不少周折。