CS520 Assignment 3: Inverse Kinematics with Skinning

Due Monday April 17, 2023, by 11:59pm

Instructions

In this assignment, you will implement skinning, forward kinematics (FK) and inverse kinematics (IK) to deform a character. The character is represented as an obj mesh. We provide ASCII files for skinning weights and skeleton data. Our starter code can load the mesh, the skinning weights and the skeleton data, and render the mesh. Your task is to fill in the missing functions and code blocks to add skinning, FK and IK functionalities to the starter code.

Starter code and data

You can download the starter code and data here.

Code setup instructions

This assignment requires three external libraries: Eigen, ADOL-C and our old friend OpenGL. Eigen and ADOL-C have been provided in the starter code.

Eigen

Eigen is a C++ template library for linear algebra: matrices, vectors, numerical solvers, and related algorithms. For this assignment, we will mostly use Eigen to solve linear systems. Eigen is (mostly) a header-only library, which means you don't need to compile and install it before using. For basic Eigen usage, see https://eigen.tuxfamily.org/dox/GettingStarted.html. For using Eigen to solve linear systems, see the simple example file EigenSolveExample.cpp in the starter code.

ADOL-C

ADOL-C is a library for computing first and higher derivatives of vector functions. For Windows users, we have provided pre-compiled ADOL-C headers, libs and dlls in the starter code. We have also provided a VS2017 project that has included the paths to ADOL-C. So no need to do anything. For Mac and Linux users, we have provided the source code and students can compile them using the following instructions:

```
First, we need to install some necessary tools for compiling ADOL-C.
For MacOS (tested on 10.14.2), we recommend installing Homebrew.
        Homebrew is a MacOS software package management system that provides an easy way to use libraries on MacOS as if using them on Linux.
        Then, open Terminal, run
        $brew install autoconf automake libtool
        to install the tools.
For Linux, run
        $sudo apt-get install autoconf automake libtool
        to install the tools.

Next, enter the ADOL-C folder: <starter code folder>/adolc/sourceCode/, run command
$autoreconf -fi
to create a configure script. If no errors are reported, run
$./configure
to create a Makefile. If no errors are reported, run
$make
to compile the code.

Finally run
$make install
to install ADOL-C at <your account's home folder>/adolc_base/.
If you want to install ADOL-C at a different location, or if you want to customize, you can read <starter code folder>/adolc/sourceCode/INSTALL for more information.
On Linux, you also need to add the path to ADOL-C libraries to LD_LIBRARY_PATH.

For using ADOL-C, see https://core.ac.uk/download/pdf/62914383.pdf for a brief introduction.
We also provide a simple example file ADOLCExample.cpp in the starter code which includes all the ADOL-C functions we need in this assignment.
```

OpenGL

```
In this assignment, OpenGL is used for rendering.
For Windows and Linux users, no need to do anything for OpenGL.
For Mac users, it is a little bit tricky:
If you are using Mac OS X Mojave, make sure you update it to the latest version; otherwise, OpenGL errors can occur.
Next, use Homebrew to install freeglut, which is an implementation of GLUT:
$brew install freeglut
(Note that although macOS comes with a GLUT framework, it is now deprecated and may not be stable.)
Then, open Makefile, comment the line
OPENGL_LIBS=-lGL -lGLU -lglut
and uncomment
OPENGL_LIBS=-framework OpenGL /usr/local/Cellar/freeglut/3.0.0/lib/libglut.dylib
Now you should be able to compile the starter code in macOS.
```

How to use the driver

The driver needs one argument to run. The argument is the filename of the configuration file <starter code folder>/armadillo/skin.config.
For Windows users, this argument has been added to the project properties. Launch the driver using the debug button in VS2017 and the correct configuration will be loaded.
For macOS and Linux users, execute <starter code folder>/run.sh to launch the driver with the proper configuration file.
After launching the driver, the model of Armadillo is rendered using OpenGL.
Drag right mouse button to change camera angle.
Hold middle mouse button and drag to zoom in or out.
Left-mouse click using button on the armadillo to display the closest IK handle on the armadillo.
The IK handle is displayed as a combination of three axis-aligned arrows colored red, green and blue. Not to be confused with the local coordinate frame displayed on each IK handles, which is made by three thick line segments in red, green and blue but not necessarily axis-aligned.
When one IK handle is displayed, left-mouse drag one of the arrows to manipulate the IK handle. If the assignment is finished without bugs, the armadillo should be deformed according to the movement of the IK handle.
In addition, the skeleton is displayed as yellow line segments connecting red circles. When left-clicking the mesh, the closest joint to the clicked location will be highlighted in red, and its descendent joints will be highlighted with smaller blue circles.

Important: When you compile the starter code, you won't actually see the skeleton yet. Instead, you will see a picture that looks like the following. The skeleton should show up properly once you've implemented the function FK::computeLocalAndGlobalTransforms() in FK.cpp. The transformations that you compute in this function will automatically be used by the skeleton renderer to transform the joints to the proper positions.

Algorithms

Skinning

The skeleton consists of several joints. Each joint has a parent joint, except the root joint. To implement skinning, each joint needs a skinning transform, which is a rigid transform: $p \to Rp + t$, where R is a 3×3 rotation matrix and t is a 3×1 vector. We can also use a 4×4 matrix to represent the rigid transform, and use homogeneous coordinates to represent a 3D point. In this way, transforming a 3D point can be simplified to multiplying a 4×4 transform matrix with the homogeneous coordinates of the point. We will use this representation for simplicity on this page. So for a mesh vertex i on the model, its position p_i is computed by skinning as:

$$p_i = \sum_j w_j M_j^s \bar{p}_i,$$

where $\bar{p}_i$ is the homogeneous coordinates of the rest (a.k.a undeformed) position of the vertex, j goes over all the joints that affect the vertex, w_j is the skinning weight of joint j to the vertex and M_j^s is the joint's **skinning** transform matrix. You will need to implement this equation in skinning.cpp in the starter code.

Forward Kinematics (FK)

Joints in the skeleton form a hierarchy. Each joint has a **local** transform. Using the hierarchy, we can compute the **global** transform of each joint. If joint "parent" is a parent of joint "child", then the global 4×4 transform matrix of "child" is computed from the local transform matrix of "child" and the global transform matrix of the parent,

$$M^g_{child} = M^g_{parent} M^l_{child},$$

where M_i^g is the global transform matrix of joint i and M_i^l is the local transform matrix of joint i. The local transform matrix is formed by its 3×3 rotation matrix and 3×1 translation vector. In this assignment, the local rotation matrix is computed using Euler angles, in the same way as in Autodesk Maya. All Euler angles in this assignment refer to the local rotation matrix between a parent and a child joint. Hence, we sometimes refer to them as the "local Euler angles", and sometimes simply as "Euler angles".

The starter code provides functionality to load the rest state Euler angles and rest local translation for each joint during initialization. It is written in FK.cpp. Here "rest" means the values assocated with the rest pose of the mesh. In this assignment, the local translations don't change at runtime, only the Euler angles. The input to the FK system are the Euler angles of each joint, and the output are the global transformations M_a^g of all the joints. You should first form local transformation matrices from the Euler angles, then traverse the joint hierarchy to form global transformation matrices from the local ones. You will need to implement this in FK.cpp.

Recall that skinning requires **skinning** transformation matrices for each joint. These are computed by FK too. For joint i, the equation is

$$M_i^s = M_i^g (\bar{M}_i^g)^{-1},$$

where $\bar{M}_i^g$ is the rest global transformation matrix. In the FK class constructor, you need to compute the rest global transformation matrices, and store them for re-use during runtime skinning.

Inverse Kinematics (IK)

The problem of IK is: Given target positions of several "end effectors", find the set of Euler angles at all the joints so that the "end effectors" assume their target positions (or get as close as possible to the target positions). The IK problem is usually under-constrained, and hence we need to also impose some regularization condition, such as, for example, minimize the magnitude of the joint angles or similar. In general, "end effectors" can be mesh vertices; but in this assignment, "end effectors" are a user-chosen subset of the joints. We call a joint that serves as an end effector a "handle". The position of a handle is the global translation of the joint. Let's define $f(\theta)$ as the function whose input are all the Euler angles (in the entire joint hierarchy) and whose output are the handle joint global translations. Note that θ is a vector of size n, where n is $3\times$ the number of joints. The dimension of f is m, where m is $3\times$ the number of IK handles. We must have $m \leq n$. We need the Jacobian matrix of f for solving IK. In this assignment, the Jacobian matrix is computed using ADOL-C. Use J to represent the Jacobian matrix of f. The matrix J has m rows and n columns. We use Tikhonov regularization to perform IK. Formally, IK is computed by solving an optimization problem:

$$\min_{\Delta\theta} \frac{1}{2}\|J\Delta\theta - \Delta b\|^2 + \frac{1}{2}\alpha\|\Delta\theta\|^2,$$

where Δb is a $m \times 1$ vector representing the change of handle global positions, and $\Delta\theta$ is a $n \times 1$ vector representing the change of Euler angles we want to find. The term $\frac{1}{2}\alpha\|\Delta\theta\|^2$ is a regularization term to avoid changing Euler angles too much; this tends to stabilize solution, at the cost of somewhat not meeting the required handle positions. Parameter α determines how much regularization to add, and as such controls this tradeoff. Solving the optimization problem is equivalent to solving the following equation:

$$(J^T J + \alpha I)\Delta\theta = J^T \Delta b,$$

where I is a $n \times n$ identity matrix. You can use Eigen to compute the system matrix and the right-hand-side, and solve the linear system.

Implementation and Demos

In the starter code, you will see several places marked with "Students should implement this." (or similar). These are the places where you need to provide your implementation. You do not need to modify driver.cpp (for core credit).

The assignment ships with three demos: armadillo, dragon, hand. You can switch between them by modifying run.sh.

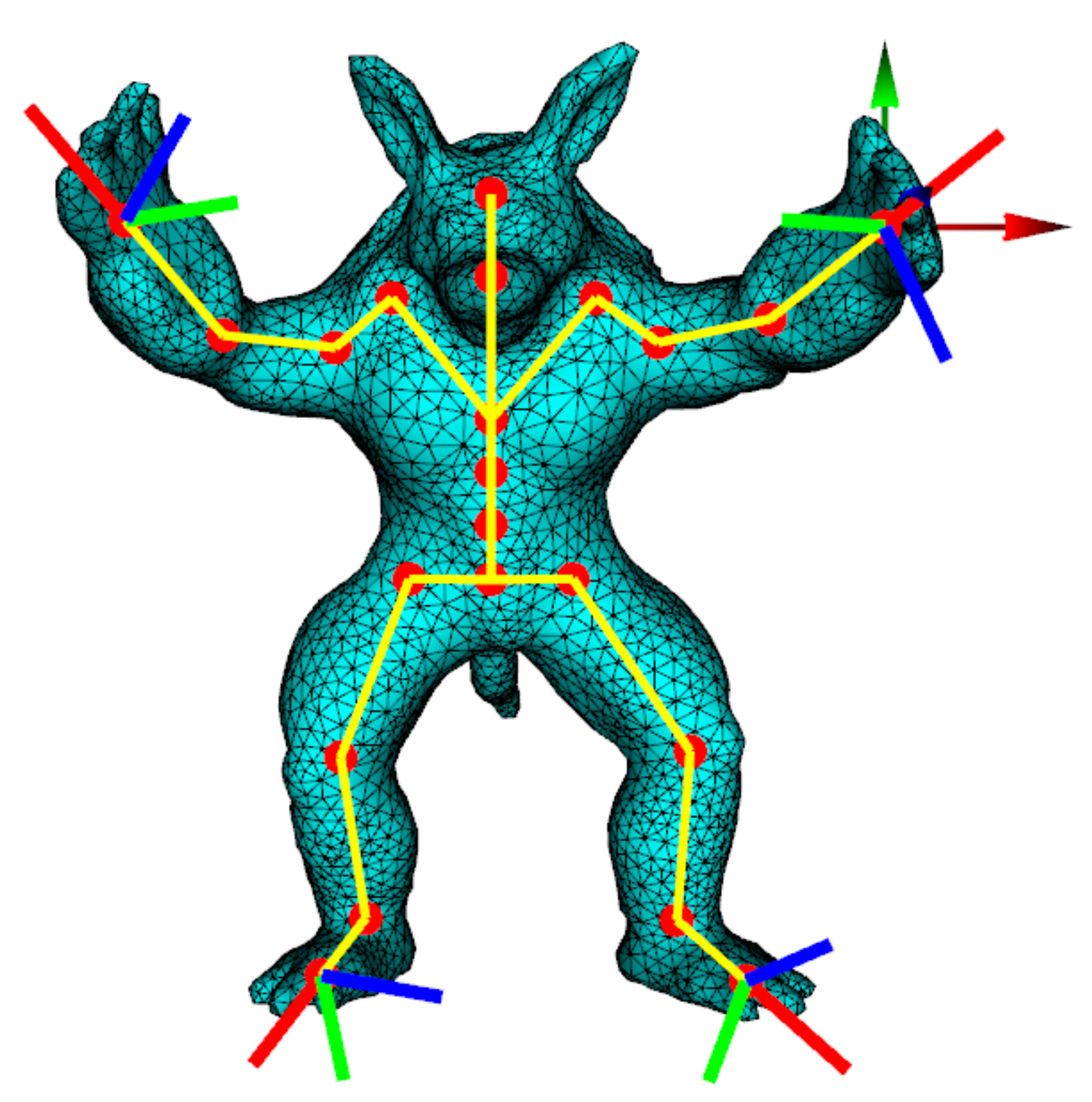
You can change the manipulated IK handles by editing skin.config, field "*IKJointIDs". To obtain the ID of a joint, click on it with the mouse and look into the Terminal window.

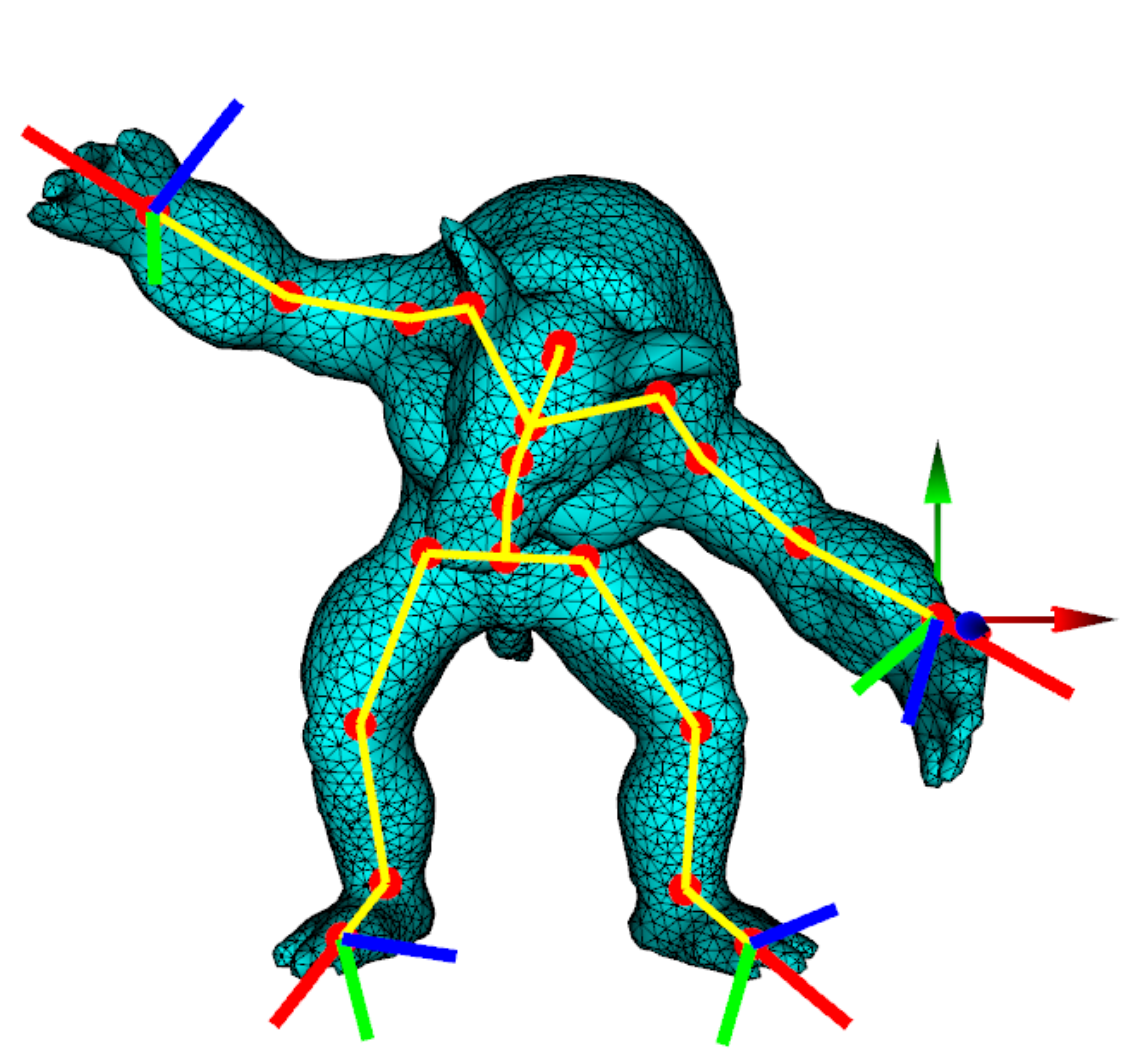
How to submit the assignment

Upload your entire solution as one zip file to the Blackboard. Don't forget to include your README file, the compiled executable (Windows or Mac, include all the required DLLs), the animation frames, and any other material required by the assignment writeup. For the animation, use the same format as with Assignment 1. Please submit JPEG frames (assumed frame rate is 15 fps), **at the 800x600 resolution** (or better). Do not exceed 600 frames.

Note: Blackboard has a limited upload bandwidth. To avoid uploading issues, please delete unnecessary files before submission. For example, Visual Studio will generate large redundant files. Remove them. Remove all object files (.obj, .o, or similar). Remove all intermediate compiling files (e.g. those in Debug/Release). **Be careful to not accidentally erase your actual code files.**

Example screenshots





Extra credit ideas

- Implement other skinning methods such as dual-quaternion skinning. Provide a comparison between linear blend skinning and dual quaternion skinning.
- Implement IK handles at mesh vertices instead of at skeleton joints. This means that the IK computation now needs to involve skinning.
- Implement and compare other IK algorithms, such as the pseudoinverse IK method.
- When the user moves the IK handle for a long distance, divide the IK process into several sub-steps to improve the solution, where each sub-step solves the IK problem on a portion of the original distance.
- Any creative extra credit contributions are encouraged.

Jernej Barbic, Yijing Li, USC