PHY 350/CS 460 Framework
Technical Manual

By: Tommy A. Brosman IV
tbrosman@digipen.edu
Fall 2009

Major Revision Summary

October 8, 2009 – Tommy Brosman
Created initial document.

October 10, 2009 – Tommy Brosman
Added info about alternate rendering modes.

October 31, 2009 – Tommy Brosman
Assignment 2 info.

November 22, 2009 – Tommy Brosman
Assignment 3 info.

December 10, 2009 – Tommy Brosman
Assignment 4 info.

Assignment 4 Implementation Overview

Object Modeling

For the physically based modelling project, I chose to do a constrained dynamics engine. More information is in the process document.

Constraint-Based Animation

The integration step is Forward Euler as described in the process document.
See physics.cpp - Physics::DampedEulerIntegrate

Visualization

The third project is fully integrated with projects 1, 2, and 3.
Changing the Init/Update functions to run an older project should behave as expected
Skinning is not used, since all bodies are rigid
As with Assignment 3, SPACE turns picking on and freezes the camera (hitting it again turns it off)
The camera controls are the same as described under Instructions

Assignment 3 Implementation Overview

Object Modeling

The IK chain has a length of 7 (see Character Schematic)
The target object can be picked and moved (hit SPACE to toggle pick mode, drag with mouse)
The character moves close to the target when out of bounds (using steering behavior)
    Note that the character does NOT use the curve-following from HW 2
    Curve following caused visual artifacts with turning when creating a new path
    Animation/movement speed synching is implemented, but does not use ease in/out
    See GameLogic::Update() for the implementation
All the IK information/etc is generated in GameLogic::InitCS460HW3()

Inverse Kinematics

For a description of the inverse kinematics algorithm, see IK Algorithm.
The algorithm described is implemented in AnimationManager::SolveIK()

Visualization

The third project is fully integrated with projects 1 and 2
The rendering speed is clamped to 60 Hz in source/system.cpp
The “floor” is rendered as a grid in source/graphics.cpp
SPACE turns picking on and freezes the camera (hitting it again turns it off)
The camera controls are the same as described under Instructions

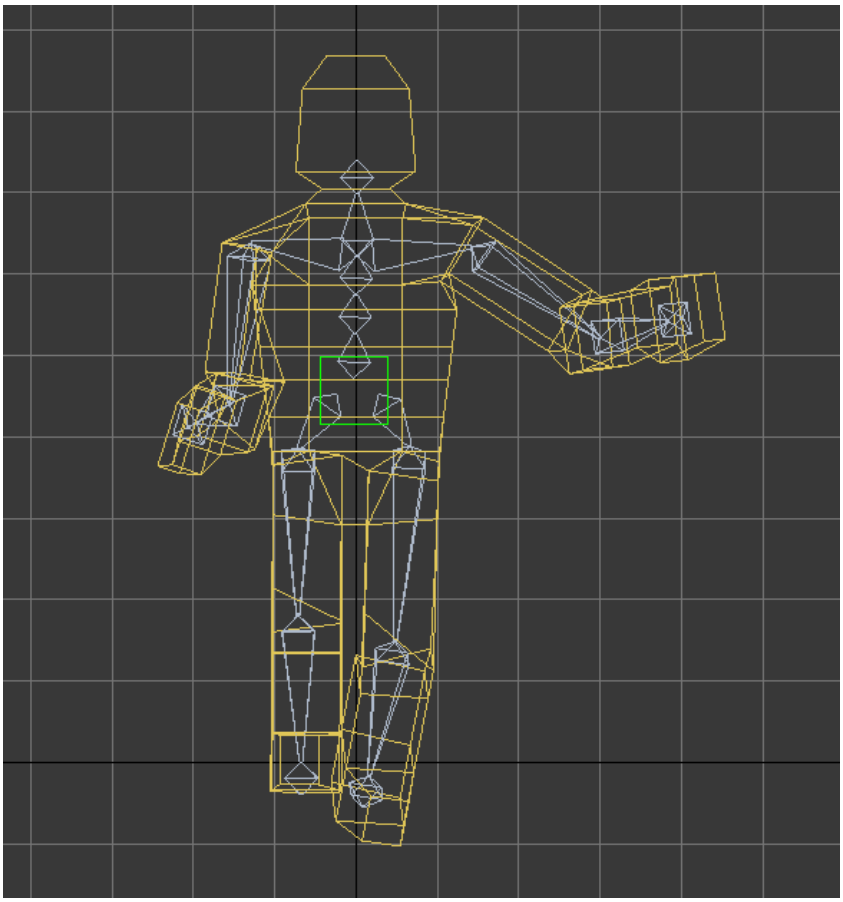
IK Algorithm

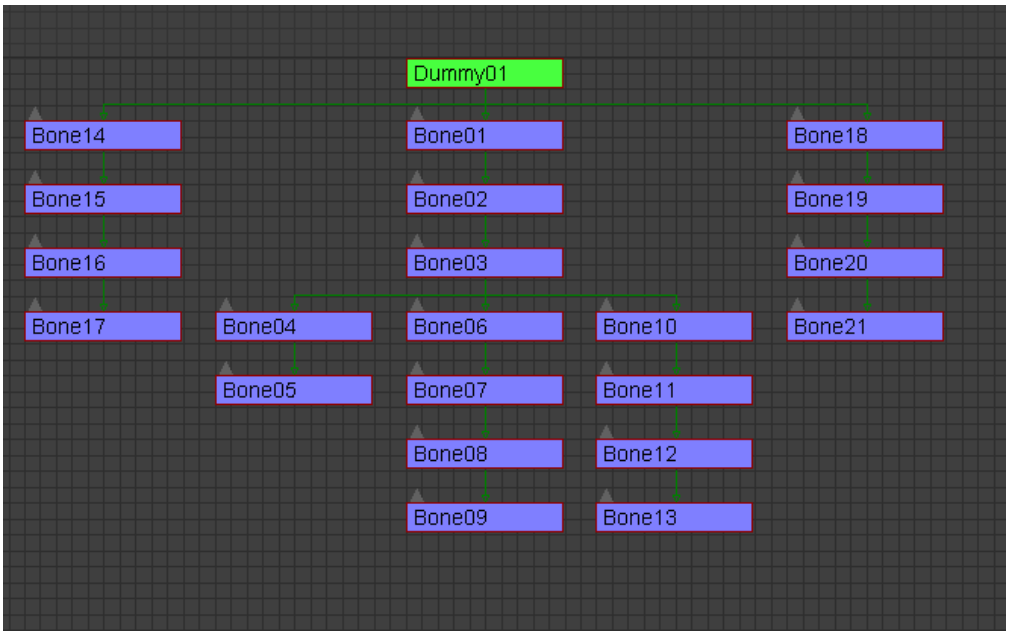
The algorithm implemented in CS460 HW3 is a modified version of CCD. The angle constraints are enforced by Slerping back to the maximum angle if the constraint is violated. Also, instead of interpolating results, a damping factor is multiplied by the angle. If the constraints and priority list are set to something reasonable, the algorithm provides smooth animation without the need for interpolation.

```
for i=0,...,k-1
  index = priorityList[i]

  destPos = destination position in joint[index]'s parent space
  endEffector = end effector position in joint[index]'s parent space
  jointPos = the current joint's translation (joint position in parent space)

  distance = |endEffector - jointPos|
  if(distance < eps)
    stop

  vcurr = (endEffector - jointPos) normalized
  vdest = (destPos - jointPos) normalized

  axis = vcurr X vdest
  angle = dampingFactor*acos(vcurr * vdest)

  tempRot = [cos(0.5*angle), sin(0.5*angle)*axis] * joint[index].currentRot

  // Slerp if the angle between the pose rotation and current rotation is
  // greater than the maximum angle constraint
  c = acos(tempRot dot joint[inded].poseRot)
  if(c > joint[index].maxAngle)
    float frac = joint[index].maxAngle/c
    tempRot = Slerp(joint[inded].poseRot, tempRot, frac)

  links[index].rot = tempRot
```

Character Schematic

This is the character used in CS 460 HW3 (“Low Poly Guy 2”). The IK chain animated by the program runs from Bone01 to Bone09.





Priority List for CCD

The priority list makes the arm and waist the most flexible, and the upper back the least flexible. The end effector is last.

Order:

- Bone08
- Bone07
- Bone06
- Bone01
- Bone02
- Bone03
- Bone09

Each bone has an angular constraint relative to its original pose. The maximum (unsigned) angle in any direction for each bone is:

Bone01 – 0.1
Bone02 – 0.1
Bone03 – 0.1
Bone06 – 0.2
Bone07 – 0.3
Bone08 – 0.6
Bone09 – 0.1 (end effector, never manipulated directly)

Assignment 2 Implementation Overview

Character Path

The points list can be found in assets/character_path_points.txt
All points in the list are in (x, y, z) form
The control points are interpolated using a piecewise cubic bezier curve with C1 continuity
The implementation for this can be found in source/geom/cubic_bezier.cpp and .h
BezierSpline contains multiple cubic Bezier segments with aligned tangent lines
A tangent factor of 0.15 is used instead of 0.5 for less distortion in small curves

Arc Length Calculation

The arc length table is built by BuildArcLengthTable() in source/geom/cubic_bezier.h
The lengths of finite differences between points on a sampled curve are used
If the angle between two segments is straight, the segments are concatenated
After the finite differences are taken, all lengths are normalized (time is already normalized)
Both the arc length function and its inverse can be evaluated within the CurveFollowComp
GetSegByTime and GetSegByDist in source/comp/curve_follow_comp.cpp are used
They perform a binary search either by time or by distance

Speed and Orientation Control

A first order method is used for speed control instead of directly using a space curve
The curve itself is parabolic, Implemented in source/geom/parabolic_interp_curve.cpp
In AnimationManager::Update() (in source/animation_manager.h), the update is applied
The ease in/out curve's derivative's value at t is taken to be the velocity
This same value is also used as the time interval when updating bone positions (v0 = 1)
This eliminates most sliding and skidding, though it's not perfect
For orientation control, a center-of-interest approach with endpoint stabilization is used
The orientation control is implemented in CurveFollowComp::BuildTransform()

Visualization

The second project is fully integrated with project 1
To view project 1 from project 2, simply replace the 2 in InitCS460HW2() in GameLogic::Init()
The rendering speed is clamped to 60 Hz in source/system.cpp
The “floor” is rendered as a grid in source/graphics.cpp
Both the curve and its polyline are rendered as well
As with assignment 1, the bones can be displayed by hitting SPACE
The camera controls are the same as described under Instructions

Instructions

SPACE cycles the render mode between skin, bones only, and transparent skin with bones
W,A,S,D move the center/target of the camera around.
Dragging while holding the left mouse button pans the camera.
Dragging while holding the right mouse button zooms the camera.

Compiling and Running

Requirements

OpenGL	rendering version: OpenGL 2.0 with GLSL shader model 2.0 (at minimum) not included
GLEW	OpenGL extension management version: glew 1.5.1 included
GLFW	window management, input version: glfw 2.6 included
Boost	lots of stuff (hash tables, synchronization, etc) version: Boost 1.37.0 or later (up to Boost 1.40.0 works so far) not included
MSVS 2005	IDE/compiler version: Microsoft Visual Studio 2005 (IDE Version 8.0.50727.762) command line compiler version 14.00.50727.762 not included

Instructions

Make sure *phy350_cs460_framework* is the currently active project in the solution. Debug compile is working, Release compile not yet implemented (as of 10/8/2009).

File Structure

./glew32.dll	required to be in the working directory
./assets/	all art assets used by the engine, including those generated by dotx_to_binary
./shaders/	all shaders used by the engine
./source/	the sourcecode directory
./lib-msvc/	libraries for compiling in Microsoft Visual Studio
./lib-cygwin/	libraries for compiling under Cygwin (untested as of 10/8/2009)
./art_source/	the original model files, etc

Tools

3DS MAX 2010

Used to create an animate all models.

Paint.NET/GIMP/etc

Used to generate all textures.
all textures are in targa (.tga) format, 32-bit, uncompressed
the texture loader assumes standard targa format (image stored from bottom to top)

Panda DirectX Exporter

Used within 3DS MAX to export .x files.
all keyframes were saved in matrix form
left-handed coordinate frame was used for export
UVW modifiers were collapsed onto model in 3DS to deal with known bug
(only the skin modifier was kept in the modifier stack)

dotx_to_binary

Created for this project as a way to convert art .X files to an intermediate format
can be built from the main solution
is meant to be used as a command line tool
uses a binary I/O library that is designed to be (potentially) cross platform

Implementation Overview

The game engine implemented is split into multiple modules. A more detailed overview of the structure of the modules can be found in the TDD for “The Shaman Engines” (the game this architecture was originally developed for). The driver code is all in GameLogic, and imports the models, animations, and textures, as well as handling the camera movement and advancing the time for each animation set. The System module handles all debugging and logging, as well as timing for the engine. The animations are all time-based, not frame based. All game objects and the camera are stored in the ObjectManager module. The GraphicsComp holds all the mesh data as well as the animation skeleton. The Skeleton class contains the actual bones, as well as the animation tracks and intermediate transform buffers. The Graphics module renders the latest animation update by passing all relevant information to the bone shader.

When generating the latest set of bone matrices, the Skeleton class first computes all the current bone positions based on the time using linear interpolation for translation and Slerp for rotation. All bones are stored as quaternion-vector pairs, as are intermediate transformations (in the localTransformBuffer attached to the Skeleton class). This gives the bone to parent transformations, and the next step is to call the Skeleton::BuildBoneMatBuffer function to generate the final modelspace to bonespace to modelspace transformations. This function recursively builds the transformations using a stack, converting each of the final transformations to a matrix and concatenating it with the skin matrix (M = (all bone transformations, from root to current bone)*(skin matrix for the current bone)). Quaternion-vector transformations are used up until the point when the skin matrix needs to be concatenated.

The Graphics::RenderAnimatedMesh function then passes the bone matrix buffer to the shader, and for each vertex sends a set of 4 weights and 4 bone indices. The shader then weights these, producing the final matrix and concatenating it with the model to worldview matrix and the projection transform. All vertices are assumed to have exactly 4 indices, and any indices not used simply have a weight of 0.