Parallelizing Skeletal Animation and Skinning

Melody Chang

30 April, 2018

Abstract

This project is to parallelize the joint transformation and skinning function provided from CS 4600 Computer graphics course. I wanted to decrease the time to recalculate the values of each joint and vertices when animating a 3D model. This will improve the render time and decrease the chance for animation to lag. The result turned out well, the overall time seems to be about the same but the CPU percentage dropped a significant amount.

1 Introduction

Computer animation has become a big part of our life. I have taken the course CS4600 Computer graphics at University of Utah taught by professor Ladislav Kavan. During one of the lectures we talked about animation with Joint transformation and Skinning. During this class we have done a project on learning matrix transformation to animate a character, using skeletal animation and Linear blend transforming. I will be using this project as the base and improve the time of rendering by parallelize the joint transformation and skinning function. I will be using Visual Studio Community 2017 as my IDE, TBB to parallelize the functions, OpenGL to render the animation. All of these will be done on a 8 cores machine. I will time the result using the timer provided by TBB and the visual studio profiler to get the total CPU percentage.

For this section and all following sections: If you refer to an equation, previous result or theory that is not regarded as common knowledge, then cite the source (article or book) where you found this. For example, you can cite the Nano 3 Lecture notes [1].

1.1 Objectives

The Objective of this project is to parallelize the calculation of joint transforming and skinning to reduce the time spent in both functions while animating. This will decrease the run time when rendering and animating the model it will also reduce chances to have lags in the animation.

1.2 Potential impact

If the parallelization succeeded and the runtime has been successfully reduced. Animators will be able to spend less time waiting for their models to be animated which will also speed up their working time and make the animation smoother.

It will help animators finish their project faster also will reduce the chance of lagging when running on a slower machine.

2 Background

2.1 Animation

Animating a 3D model using only its vertices will cause a lot of degree of freedom (DOF) therefore will be difficultly to control. So we chose to use skeletal animation and linear blend skinning to animate the 3D model to improve the run time, decrease DOF and memory usage.

2.2 Skeletal Animation

A skeleton is made up a tree of nodes and edges which represents joints and bones. To animate a skeleton we will compute the joint transformation by using this equation.

$$F(j) = F(p(j))R(j)T(j) \tag{1}$$

$$F(0) = R(0)T(0) \tag{2}$$

Where F is the result matrix which consist the animation pose by the calculation. p is the parent matrix, it keeps tracks the parent node for every joints. The root joint does not have a parent joint so in the parent matrix the value will be -1. R as the rest pose matrix or offset matrix, it stores the relative transformation between individual joints (joint information in rest pose). Values in this matrix does not change. T as the transformation or local matrix, which includes the rotation and translation of each joint, values in this matrix changes.

2.3 Linear Blend Skinning

Now that we have made the skeleton moved, we will want to move the skin with it. To do that I am using Linear Blend Skinning method. It takes the computed transformed matrix from joint transformation and compute the deformed vertex position with the following equation:

$$v' = \sum_{i=1}^{m} wi * F(ji) * A^{-1}(ji) * vi \tag{3}$$

Where j represents joints j1, ..., jm. W as the weights matrix that includes describes the amount of influence of bone j on vertex i. A common requirement is that weights must be convex, i.e.

- `w1+ ... + wm = 1`
- `w1>=0, ..., wm>=0`

weights are usually edited manually.
F is the result from joint transformation. `A^(-1)` is the inverse of the joint translation position at rest post. v is the vertex position at rest pose. v' is the vertex position in the animation pose. j is the joint which vertex v is attached to. For each vertex we calculate the sum of the weights multiply by

the joint transformation and the inverse joint translation position at rest post then multiply the sum by the position of vertex in rest pose to get the position of this vertex in animation pose.

3 Overview

The base code that I am using we produced by professor Ladislav Kavan. There are a few different parts that were put together for the model to become successfully animated.

3.1 Loading Object and Initialization

In the project folder, there is a data folder which contains a capsule and a ogre folder. Each of the folder contains a mesh.obj, skeleton.bf (a matrix that represent the skeleton data), pose.dmat (a rest pose matrix and a transformed pose matrix), and weights.dmat (a list of number ¿1 as weights). At the start of the program these files are read and the variables such as the offset matrix, joint rotation matrix and joint parent matrix are being initialized, other variables are being resized according to the total number of joints. After the files are loaded, joint transformation is being called to calculate the exact joint position for the model at rest pose.

3.2 Render Loop

After the initialization the render loop function is being called. Before I have modified the function for recording time data, the function will be in a loop until the program has been terminated. During each loop the rotation of each joint is determine by a callback function where it sets the rotation matrix according to either 0, 1, or this function `std::fabsf(sinf(getTime()))` The program then computes a new joint transformation depending rotation and deformed vertices position based on joint transformation.

4 Implementation plan

Before deciding which part of the code U should parallelize, I run the original code with Visual Studio performance profiler on CPU runtime. The result showed that while the program was running it was spending 76% of the CPU runtime in the Skinning method.

4.1 Skinning

According to the CPU profiler the program is spending most of its time calculating the deformed vertices position which is this line of code

```
tmp += p_weights[i - 1][v] * p_jointTrans[i - 1] * p_jointTransRestInv[i - 1] * toHomog(p_vertices[v]);
```

To improve the performance I decided to parallelize the outer loop in the function which was iterating over all the vertices and calculating the deformed vertex position for each of the vertex. To do so I added a start and end variable in the function call to make it recursive. The function will call it self multiple times

on different threads to calculate the deformed vertices for different interval of vertices. The result will be store it in the deformedVertices matrix which is a global matrix.

4.2 Joint Transformation

Joint Transformation is another function that was called multiple times and since the serial code contains a for loop, I have decided to parallelize this function. My plan was to do the same thing I did with the skinning function and make this function recursive. The function will calculate the joint transformation for different interval of joints. The result will be store in the global joint transformation matrix. I was not able to get this function parallelized and experienced a data race. The equation to calculate the transformation of each joint requires the newly calculated transformation of the parent joint, so when I parallelized the function, it caused a data race.

4.3 Performance Goals

The goals of this project is to improve the runtime of the render loop function by half, if not one fourth of the original runtime. The function calls the skinning function which is the main function that I will be parallelizing.

5 Testing

5.1 Verification Plan

To verify the project have reached its goals, I will be performing Weak Scaling as well as Strong Scaling.
To get the Weak Scaling performance, I timed and increase the number of times that Skinning function has been called. The equation I used to increase the number is 2^n, where n = 1...14. I timed both the parallelized version and serial version as well as with a capsule model (small file) and a ogre model (large file). While getting the performance. I will also run the performance profiler with it to get the total and self CPU runtime percentage.
To produce the result of strong scaling, I will record the total time it takes to run the parallelized skinning function 5000 times with number of threads going from 1 to 8. I plan to change the number of threads initialized manually so I can run the performance profiler with the code to get total and self CPU runtime percentage for the skinning method.

5.2 Acceptance test

This project is actually created for class homework so there are no place for user to insert their desired input unless they have created a folder that contains a mesh.obj, skeleton.bf, pose.dmat, and weights.dmat files in it. The homework project have 2 folders in the data file, which contains a capsule and an ogre model. I plan to run the performance analysis and get the runtime on both model.

6 Project schedule

Date	Tasks
1/16/18	Project topic meeting
3/5/18	Meeting with professor to discuss project topic
3/12/18	Meeting with Saeed for suggestion on project set up
3/28/18	Profile which part of code takes most time
4/4/18	Finish setting up TBB on Visual Studio
4/11/18	Parallelize Skinning function
4/18/18	Parallelize Joint Transformation function
4/22/18	Gather runtime information
4/23/18	Finished Gather weak scaling runtime information
4/24/18	Gather String Scaling runtime information
4/25/18	Present

References

[1] Lecture slides from CS4600 Computer Graphics taught by Ladislav Kavan.