Ross Adelman
15468 A

Collision detection in the cloud

For project 7, I decided to reorganize my rigid body simulator from project 4 into a client/server system.

I removed the GPU version of the collision detection code for this project -- instead, all the collision detection is done on the CPU. I did, however, add optimizations that made the CPU version faster, including the sort and sweep algorithm.

Before jumping into implementing the client/server system, I did some back of the envelope calculations to determine how much data I would have to send back and forth between the server and a single client. Suppose the server is simulating 1000 bodies at 30 frames per second. Each body is represented by a position (a vector) and an orientation (a matrix) -- this amounts to 48 bytes per body, 48 kilobytes per frame, and 1.4 megabytes per second. That is a lot of data to move around!

Here is how the client/server system works.

The server, located on one machine, handles all the calculations involved in the simulation, including the force accumulator, the integrator, and the collision detection code.

Whenever a client connects to the server, the server immediately sends to the client information about the world, such as the number of bodies. In addition, it sends information about each of the bodies that will not change from timestep to timestep, such as type, geometric properties, and color.

When launched, the server is fed the number of clients that are going to connect, and it waits for that many connections before continuing on with the simulation. This way, the server can establish connections before the simulation begins, and not have to check for new connections while the simulation is in progress.

Once all the clients have connected, during each timestep, after calculating the new positions, orientations, and so on of each body in the simulation, the server sends to each client a copy of the updated state of the world. Upon receiving the updated state of the world, the client renders the scene using standard OpenGL routines.

I originally coded the server and client to send and receive data in ASCII. However, I eventually added support for sending and receiving data in binary. How the client and server communicate can be adjusted when they are launched on the command line.

I measured transfer rates and frame rates on the server and the clients for different numbers of clients using both ASCII and binary. Below are six graphs that show how frame rates and transfer speeds (on both the server and the clients) are affected by adding more clients to the mix. The top three graphs show the frame rates, transfer rates on the clients, and transfer rate on the server using ASCII encoding. The bottom three graphs show the same things as the top three, except binary encoding was used instead of ASCII encoding.

I collected this data while running a moderately complicated simulation -- the simulation consisted of 1350 blocks falling on nine stationary blocks. The timestep was 0.05 seconds. This is why on many of the plots, some lines appear to end earlier than others. I stopped collecting the data as soon as the simulation reached a certain point (the blocks had all fallen), so the simulations that had higher frame rates completed faster than the others. Also, the frame rates tend to drop half way through -- this is when the most collisions are occurring.

There are some interesting trends visible in the graphs.

As the number of clients increases, the frame rate decreases. This happens because the server begins to spend more and more time sending data to the clients and less and less time calculating the upcoming frame.

Also, as one would expect, the amount of data the server sends increases as the number of clients increases. However, this increase is not linear. In fact, the transfer rate on the server begins to saturate at around 80 MBps, indicating that the Ethernet card on the machine is nearing its hardware, probably around 100 MBps on the Gates machines.

I only attempted to run five clients simultaneously, no more. However, even at five clients, the frame rates are still quite high. Using binary encoding, frame rates never dropped below 130 or so frames per second. I imagine I could have added up to ten more clients before the frame rate dropped below 30 frames per second, which is where the animation would begin to lose its smoothness. After that, the server's Ethernet card would probably saturate to the point where adding another client would be too expensive.

In the future, I would like to investigate better methods for sending the updated state of the world to the clients.

For example, instead of having the server send an entire copy to each client separately, a better method would be to send a different part to each client, and then have the clients forward them onto the others. This way, the server (and the clients) would only send in total one copy each of the world to the clients.

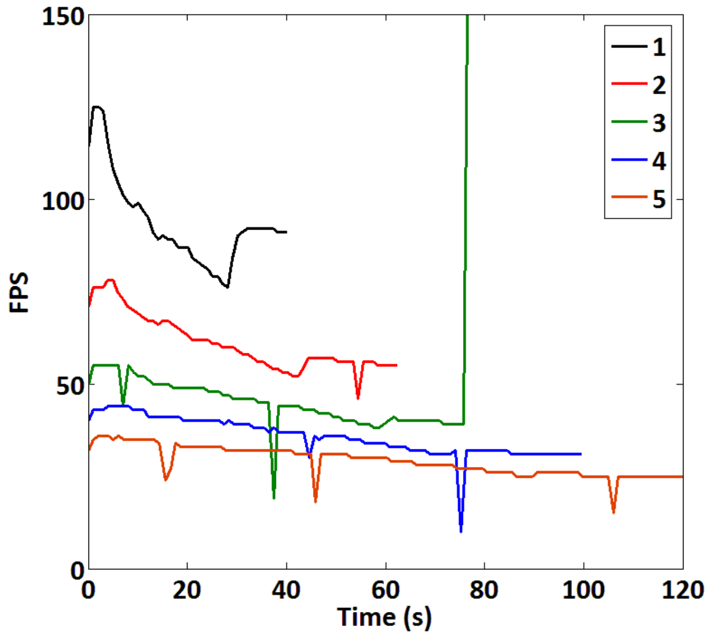

Figure 1. Frame rates on the clients (data shown is from the clients) for different numbers of clients using ASCII encoding.

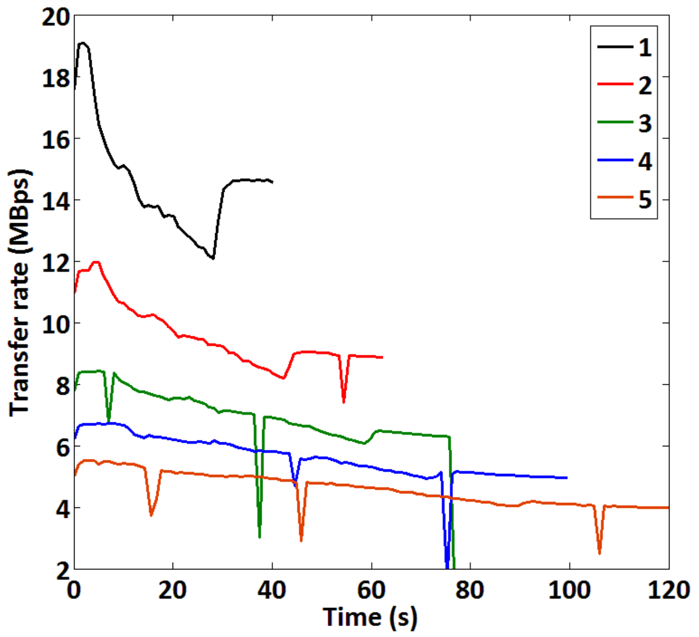

Figure 2. Transfer rates on the clients (data shown in from one of the clients) for different numbers of clients using ASCII encoding.

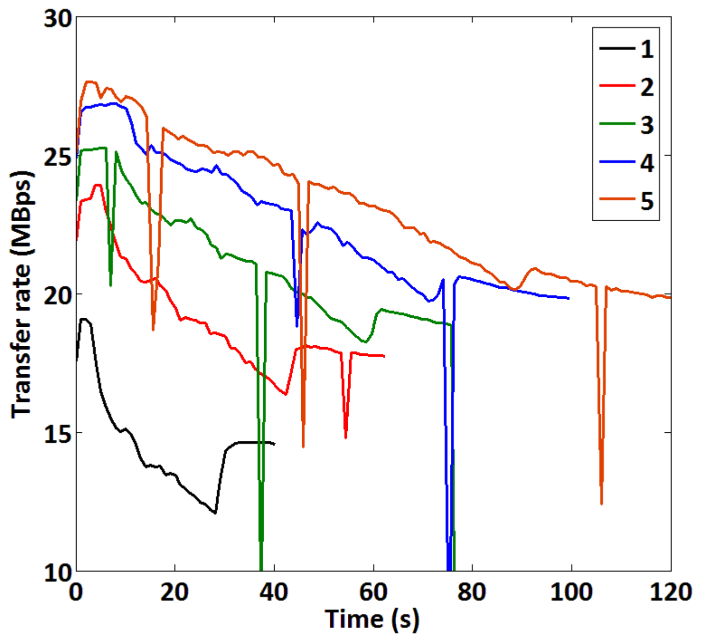

Figure 3. Transfer rate on the server for different numbers of clients using ASCII encoding.

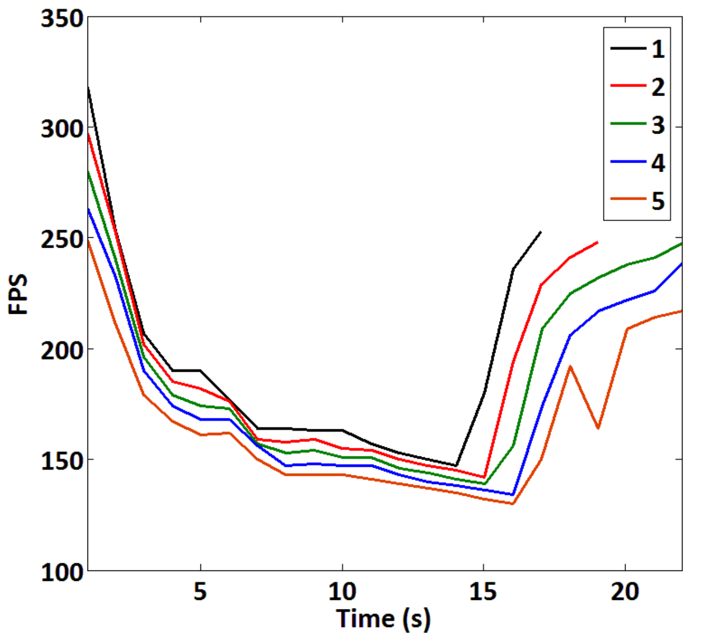

Figure 4. Frame rates on the clients (data shown is from the clients) for different numbers of clients using binary encoding.

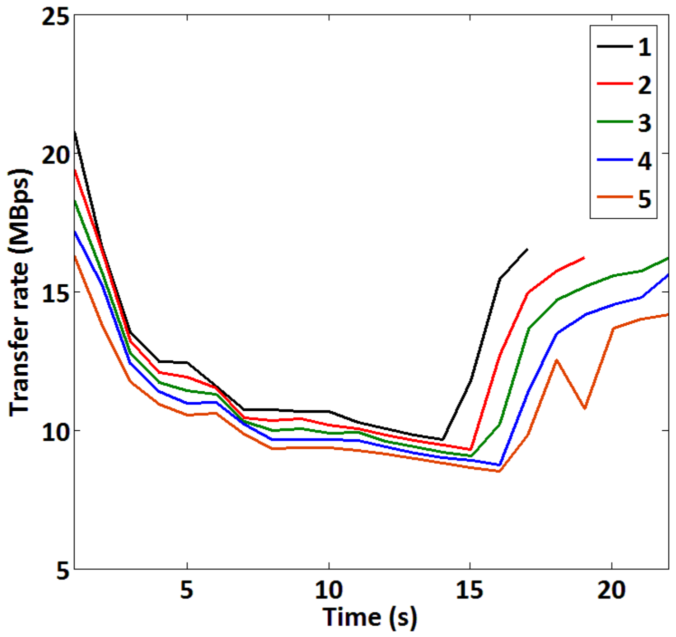

Figure 5. Transfer rates on the clients (data shown in from one of the clients) for different numbers of clients using binary encoding.

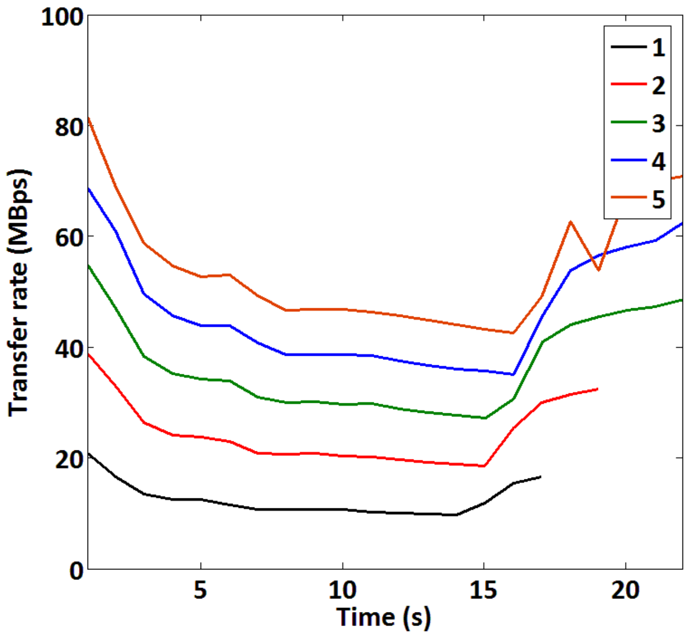

Figure 6. Transfer rate on the server for different numbers of clients using binary encoding.