Compression of Physical Simulations for Mobile Virtual Worlds

Eric Butler

CMU-CS-11-126

August 2011

School of Computer Science
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
Adrien Treuille, Chair
David Andersen

Submitted in partial fulfillment of the requirements for the degree of Master of Science.

This research was sponsored by NVIDIA Corporation, Intel Corporation, Alcatel-Lucent S.A., and the National Science Foundation.

Keywords: physical simulation, compression, real-time graphics, client-server design

Abstract

This paper introduces a general technique for streaming complex, interactive physical simulations to portable devices from a server. We present an implementation of this approach with an iOS client application that runs particle simulations with several thousand objects. We achive good compression by factoring the simulation in two parts: the portable device computes only the easy simulation steps that have a large impact on the result. Therefore, the server only needs to transmit information about the simulation steps that are hard to compute. The server can therefore transmit significantly compressed data while still allowing clients recover the full simulation state.

Acknowledgments

There are many people I'd like to thank for help with this project.

First, I would like to thank Adrien Treuille, a great advisor and mentor. His ideas and guidance have been invaluable for me, and he is a very enjoyable person to work with.

I would also like to thank David Andersen for advising our projects during the past year and for being on my committee.

David Klionsky has contributed an enormous amount of work on this project and deserves much of the recognition. We have worked together all this past year on many cloud-related projects of which this is but the most recent, and I would not have been able to do this or any others without him.

Additionally, I'd especially like to thank Yantong Liu, Wei-Feng Huang, and Jacob Poznanski, who have each done a tremendous amount of work the past year with David and me.

I'd like to give a special thanks to Deborah Cavlovich for help sorting through the administrative details.

Lastly, I'd like to thank my parents and brother for their support of my education over the years.

Contents

Chapter 1

Introduction

For portable devices such as smartphones, streaming information over the Internet allows for interactive applications otherwise impossible with such limited computational power. Complex, interactive physical simulations cannot be computed on portable devices, but streaming physics data from a server would enable such simulations on these machines. However, real-time streaming is impractical because of the large amount of data, and even standard compression techniques will not solve the problem. For example, in a particle simulation with 10,000 objects, the full simulation state for a single frame is over 200 kilobytes. At 30 frames per second, a program would require a bandwidth of nearly 6 megabytes per second to stream the data. An ideal system would enable reconstruction of the entire simulation state while transmitting only a fraction of this data. At the same time, such a system should be general to many types of simulation. To date, no streaming technique for interactive simulation captures these properties.

This work solves the problem by exploiting a key property of certain physical simulations. Consider particle simulation. Some steps of particle simulation, such as integration, are easy to compute yet affect a large number of particles. Others, such as collision detection, are very hard to compute but have a comparatively small effect on the simulation state. We exploit this to create a novel factorization of particle simulations, enabling real-time streaming of a large physical simulation from a server to a portable device. The client device computes the easy parts of the simulation so that the server only needs to transmit data resulting from the difficult simulation steps.

We have implemented our approach with a multi-user iOS application that allows interaction with a particle simulation of several thousand objects. Since particle simulations form the basis of many more complex simulations such as fluids or cloth, our method works with many simulation types. In fact, our method applies generally to any simulation which can be factored as described.

Our technique exhibits many unique properties for mobile devices. Our solution allows for much larger simulations with many more objects than otherwise possible on devices with such limited computational power. Furthermore, it enables multi-user interaction with very large scenes, which are not attainable even with desktop machines.

Sending physics data has several advantages over other approaches, such as streaming video. With physics data, the client is free to render at a frame rate independent of the rate in which updates are received, leading to smoother frame rates. More importantly, transmitting physics data gives the client flexibility to distribute parts of the program's updates and rendering computations on the client. For example, the client could run other independent, less computationally-intensive simulations itself.

This paper is organized as follows. Chapter 2 discusses related work. Chapter 3 describes the full and predicted simulations we use and the algorithm to create a compressed delta between the two. Chapter 4 describes the implementation details of our system as a mobile phone application. Chapter 5 shows results of the algorithm, such as the amount of data sent over the network.

Chapter 2

Related Work

We focus on particle simulations as a model problem. Particle simulations serve as the building blocks for more complex real-time simulations, such as cloth[Baraff and Witkin, 1998], fluids[Müller et al., 2003], or rigid bodies[Baraff and Witkin, 2001]. While our implementation uses the CPU to compute the simulation, even larger simulations can be run on graphics hardware[Green, 2008]. Particle-based cloth[Zeller, 2005] and fluids[Amada et al., 2004] have also been simulated in real-time on the GPU. Our method can be applied to any simulation that can be separated into a computationally-intensive portion such as collision detection and a portion that can be simulated on mobile devices.

Data compression is a widely studied topic. Specifically in computer graphics, mesh compression is relevant to our work. Mesh compression is typically done in two separate segments: lossless compression of connectivity data and lossy compression of geometry data[Peng et al., 2005]. Since the latter of these consists of compressing a sequence of 3D-vectors of floating-point data, it is applicable to our problem space. Geometry compression typically relies on first quantizing the data, then using a predictor allowing the compressed delta to be encoded rather than the full data. Examples of such predictors are the delta predictor[Deering, 1995], the linear predictor[Taubin and Rossignac, 1998], and the parallelogram predictor[Touma and Gotsman, 1998]. All of these approaches essentially generalize to predicting a vertex as some linear combination of nearby, connected vertices. Our work differs in that we don't necessarily have connected objects to use for prediction. Instead, we use a partial simulation as the predictor for object positions and velocities. However, these approaches could be used with simulations where particles are connected, such as cloth.

Networked video games are closely related to our problem and are an obvious application of our method. Due to their proprietary nature, comprehensive information on the implementation details of networked games is difficult to find, so our information is based on a few sources[Fiedler, 2010, 2011] and the best of our knowledge. There are a few main models of multi-user networked games. One is where clients all run the full simulation in lock step, sharing only input events. This is typically used when the simulation state is too large to transmit or when input latency is not of concern. Another is to have an authoritative server that sends the full simulation state to every client. This suffers the same problems already discussed, specifically large latency with user input, even with small state size. Most games using a client/server model overcome this by having clients use predictive simulations to hide the latency of communication with the server. Clients do not know other users' input events, so their predicted simulations are incorrect. On each update, the server transmits the true simulation state to each client, and each client fully replaces its prediction with the server version. Compression is often used; for example, servers typically employ delta compression by sending only changes from the previous state instead of the entire state. Other techniques employed include quantization of floating-point data or entropy encoding such as Huffman coding. But to the best of our knowledge, in contrast to our method, games do not exploit the client's prediction when compressing the server's data to any significant degree. The most a typical game may do is avoid sending data for objects under the client's control, such as the user's avatar. Usually, predictions are just discarded and overwritten with the correct data. In further contrast to our method, games do not use the client/server model to enable to use of complex physical simulations on low-end devices but rather simply for enabling a multi-user application. Clients' predictions in games with complex physical simulations are typically of comparable computational intensity as the server's simulation.

Chapter 3

Overview

The overall goal of our method is for a set of mobile devices or other low-end machines to display a complex, interactive, multi-user simulation. Since the simulation is very complex, these machines do not compute the entire simulation. Instead, each client connects to a central server, which computes the full simulation and transmits the result to all clients. Our method is to use a second, partial simulation as a prediction for the full simulation. If this prediction is close to the original, the server can significantly compress the data it sends to the client. We applied our technique to a particle simulation.

It is important to note that we do not try to precisely match the output of the full simulation, but rather we attempt have an output that is perceptively identical to it. For example, it's important that particle positions in the client's simulation are very close to those in the full simulation, but they can differ very slightly if it is not noticeable to a user. This leniency allows for significant lossy compression of the delta.

3.1 Client and Server Simulations

We refer to the algorithm that computes the entire simulation as the *full simulation*. The desired output is displaying the result of executing the full simulation. *Predicted simulation* refers to the algorithm that partially computes the desired simulation, for the purpose of serving as a prediction for compression. In our implementation, the predicted simulation is a particle simulation without collision detection. Our algorithm's goal is for each client to execute only the predicted simulation, but adjust the result each frame using data from the server so that the client's simulation closely matches the full simulation. Figure 3.1 shows a high-level picture of how the system should behave.

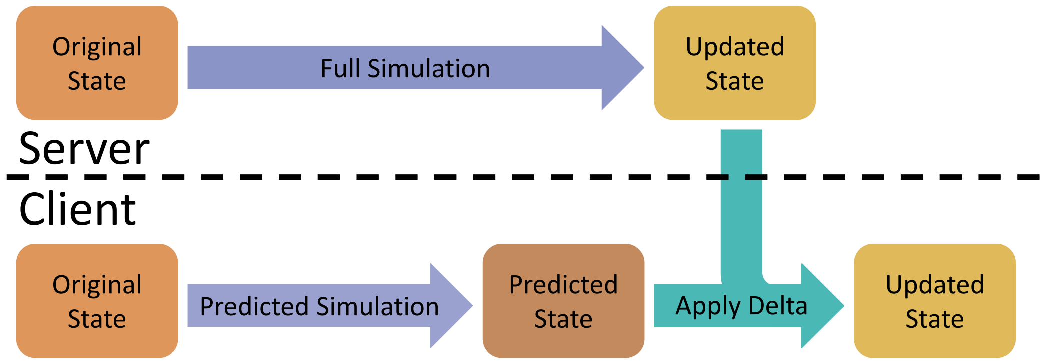


Figure 3.1: High-level picture of how the system should function. The client computes only the predicted simulation but recovers the full simulation state using data from the server.

We realize this goal by keeping track of two simulations, the server state and the client state. The *server state* is the authoritative simulation state, run by the server and updated with the full simulation. The *client state* is only updated with the partial simulation, and exists on both server and clients. The client state is updated identically on the server and on every client.

Each update step, both the server and each client update the original client state using the predicted simulation, resulting in a predicted client state. The server also updates the original server state using the full simulation, then computes a compressed *delta* between the updated server state and predicted client state. This delta is sent to all client machines. Finally, the server and every client apply the delta to the predicted client state to get the fully-updated client state. This updated client state is then displayed to users. Figure 3.2 illustrates the algorithm in detail. Note that due to lossy compression, the fully-updated client state will not precisely match the updated server state.

3.2 Simulation Details

We focus on particle simulation with our implementation. To create the predicted simulation, we remove collision detection, since it is very expensive to compute but does not have a large impact on the result.

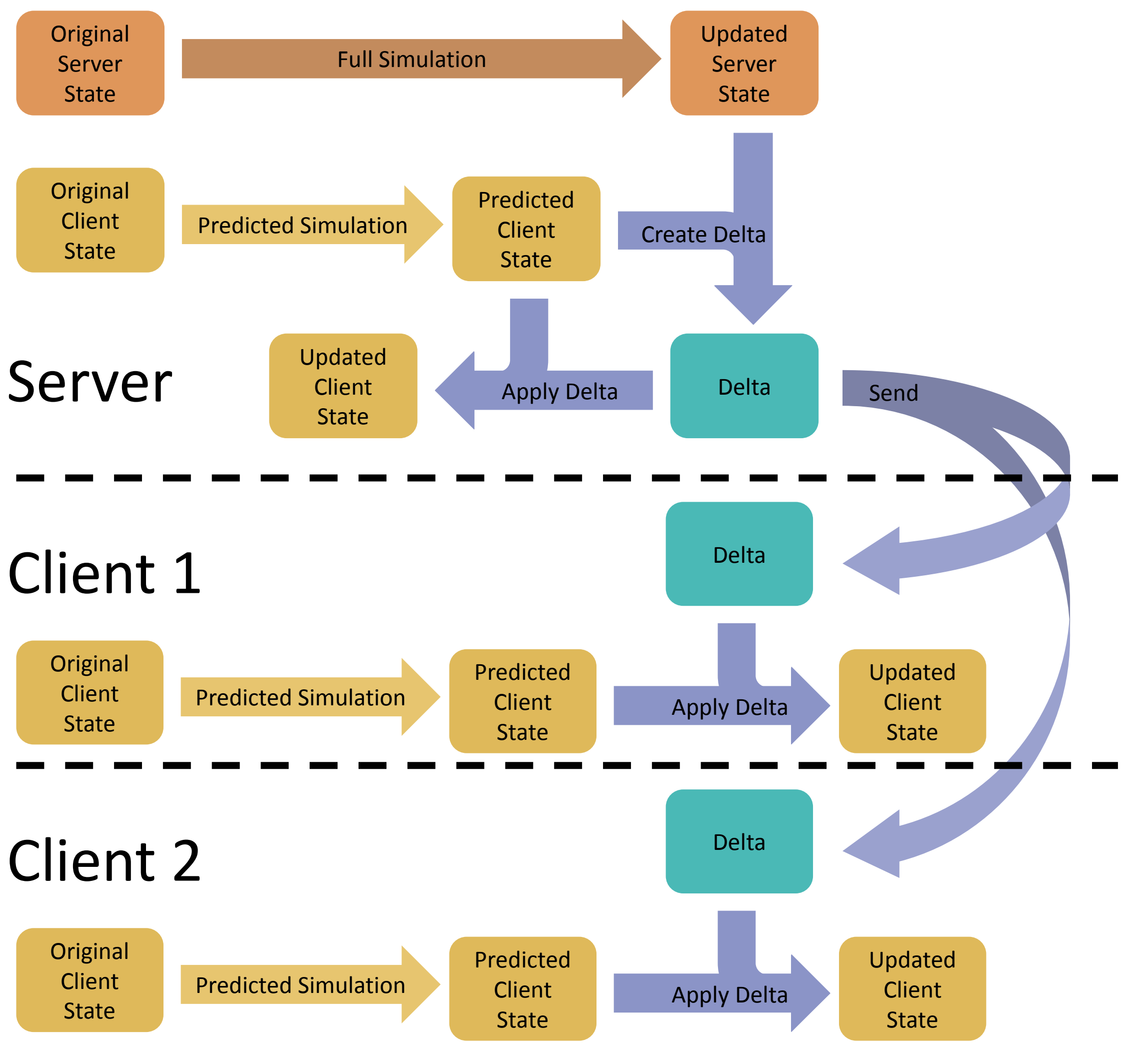


Figure 3.2: Detailed illustration of algorithm. Only the server has the *server state*, but the server and each client all have the *client state*. This illustration shows 2 clients, though an arbitrary number of clients is possible.

3.2.1 Full Particle Simulation

We implemented a standard particle simulation with penalty forces to prevent collisions. There are a number of particles, each of which consists of a 3-dimensional position and velocity. Orientation is not stored since all our objects are perfectly spherical, though our system could be extended to rigid body simulations by including orientation or constructing rigid bodies from multiple particles.

Every update step, forces (primarily gravity and velocity damping) are calculated, then each particle's position and velocity are updated using Euler integration. This is given by equation 3.1, where $\mathbf{p}_i^t$ and $\mathbf{v}_i^t$ are the position and velocity of particle i on frame t, respectively, $\mathbf{f}_i^t$ is the force applied to particle i at time t, and m_i is the mass of particle i, and δ is the change in time between two frames. We use Euler integration for simplicity and speed; other integrators with better stability or accuracy are also options.

$$\mathbf{v}_i^{t+1} = \mathbf{v}_i^t + \delta \mathbf{f}_i^t / m_i \qquad \mathbf{p}_i^{t+1} = \mathbf{p}_i^t + \delta \mathbf{v}_i^{t+1} \tag{3.1}$$

Following integration, intersections with the scene geometry and other particles are calculated, and penalty forces are applied to separate any intersecting particles. These forces are added to the force accumulator, to be applied on the subsequent update step. Our collision detection algorithm uses a spatial hash, based on an implementation by NVIDIA[Green, 2008]. The world space is partitioned in uniform cells, and before collision detection, each object is assigned to its nearest cell. Each object needs to be tested for collision with only other objects within its cell and adjacent cells.

3.2.2 Predicted Simulation

Since collision detection is by far the most computationally expensive portion of the simulation, it is the one removed to form the predicted simulation. The predicted simulation performs the same steps as the full simulation (e.g., gravity, Euler integration of position and velocity), but does not perform collision detection or resolution. Thus, the full and predicted simulations diverge when an object collides with scene geometry or with another object.

Collision detection is a good part of the full simulation to remove since it is both very expensive in computation but very sparse with its output. In many simulations, only a small percentage of particles are affected by collisions on any given frame. Even when there are a large number of collisions, a collision only changes the velocity of a particle, so a colliding particle in the predicted simulation will diverge slowly from the one in the full simulation. This means that after a single update step, relatively few particles in the predicted simulation will be incorrect, and those that are will be very close to where they should be.

3.3 Compression

We use the accuracy of the predicted simulation to significantly compress the simulation data sent by the server. This compression is what allows us to transmit the simulation state to the clients in real-time. The compression algorithm creates a delta that can be applied each frame to the predicted simulation to minimize the perceived difference between the predicted and full simulations. Since we allow output to deviate slightly from the full simulation, we can use lossy techniques to compress the delta. It is important that the compression algorithm does not modify the full simulation state, so the result of the simulation is not affected.

In our implementation, the data to be compressed are a sequence of floating-point numbers, six numbers per object. Our compression strategy consists of two major parts. First, we avoid sending any data for particles that are close enough to those the full simulation, correcting them only after a particle's error becomes noticeable. Second, for particles we do transmit, since the error is very small, we can encode that difference effectively using quantization and entropy-encoding. The entire compression technique presented here would apply to a rigid body simulation or any simulation consisting of floating-point or integer data.

3.3.1 Selective Correction of Particles

The largest portion of compression comes from only transmitting a small fraction of the erroneous particles each frame. Obviously, we do not need to send particles that have no error, but we also avoid transmitting particles with only minor error. We define particle error as the Euclidean distance of the position between the full and predicted simulation states.

There are two primary methods we use. One is to correct a particle only when the error exceeds a certain threshold. This threshold is chosen experimentally based on the scene to be as high as possible without making the corrections noticeable. We only consider position error rather than velocity error because while the position is easily visible to the user, errors in velocity are not visually apparent. If this error in velocity results in a large

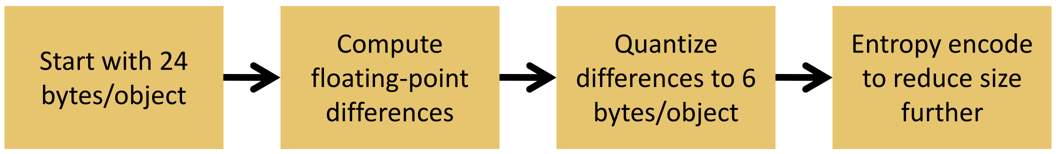

Start with 24 bytes/object
Compute floating-point differences
Quantize differences to 6 bytes/object
Entropy encode to reduce size further

Figure 3.3: An outline of how we compress individual particles.

error in position on a subsequent frame, the particle will be corrected at that time.

The second method is to use a rate limiter, forcing the delta to be smaller than a desired size. The algorithm sorts the particles by error from largest to smallest, and sends particles in that order until the desired size is exceeded. The remaining particles are left uncorrected.

Since relatively few particles need to be corrected each frame using these methods, we send a sparse list of the particles that need to be corrected. For each particle that needs to be corrected, we send its two-byte index followed by the necessary data. If the set of particles to send is dense enough, an alternative method is to send a single bit for every particle: 0 means the particle is not to be corrected, and 1 is followed by the data to correct the particle. This method is optimal when sending corrections for more than 1/16 of the total particles.

3.3.2 Compression of Individual Particles

Each object consists of 6 floating-point numbers and would require 24 bytes of correction data without additional compression. We further compress individual objects using quantization and entropy-coding. Figure 3.3 outlines the procedure.

Quantization of Floating-Point Differences

Since the predicted and full simulation are typically very close to each other, our algorithm quantizes the floating-point difference between the two simulations to reduce the size. Due to the nature of floating-point arithmetic and quantization, the full simulation's value will not be precisely recovered. However, as previously established, this is acceptable for the goals of the compression algorithm.

The range of the quantization is chosen experimentally based on the scene. The upper bound of the quantization range is chosen to slightly exceed the maximum error of experimental runs, with position and velocity data each using a different range. If a particle's error exceeds the quantization range, then the algorithm will take multiple frames to fully correct the error.

Our algorithm quantizes each floating-point value to an 8-bit integer, reducing each particle to 6 bytes of data (not counting the index).

Entropy Encoding of Quantized Values

The resulting 8-bit integers are skewed heavily toward small values, making them amenable to entropy-coding to further compress the data. Any well-known entropy-encoding method would be appropriate here. Our algorithm uses standard Huffman encoding, though it was chosen only for its simplicity, not because of optimality.

Chapter 4

System Details

This chapter discusses relevant details about our implementation. We implemented an iOS application using our technique to simulate several thousand objects in a interactive simulation. Users interact by changing gravity of the scene. The server runs on any typical desktop or machine of comparable capability.

4.1 Basic Program Architecture

The server is the authoritative controller of the simulation. The predicted simulation on the client and the server remain in synchronization for the compression algorithm to work, so the simulation updates at a fixed rate on all machines. Updates are tracked with frame numbers.

The clients send input data to the sever every update step. This message also notifies the server that the client has performed an update of the predicted simulation for a certain frame number. The server sends simulation data to all clients every frame. The data sent are the compressed difference between the full simulation and the predicted simulation as described in section 3.3.

Rendering on the client is decoupled from simulation updates, allowing for a smooth frame rate in spite of any interruptions in updates. Interactions such as panning the camera are therefore always smooth and responsive. This is a primary advantage or sending geometry data instead of streaming video.

4.2 Synchronizing Server and Client

Ideally, the server sends updates to every client every 33 milliseconds, and hears back from all of them a short time later. However, sometimes a client has a temporary interruption of communication with the server. Therefore, both the client and server are allowed to continue updating without communication from each other.

The server will continue to update at 30 updates per second, sending out simulation data to all clients regardless of how quickly clients respond. However, the server will stop if it does not hear back from a client after several updates. When this happens, server assumes there was some kind of minor network problem, and temporarily suspends updating until it either receives data from that client or disconnects from the client. If a single client suffers an brief interruption in communication with the server, the other clients can continue to receive updates, limiting the impact of the network problem to a single client, at least for a short while. In our implementation, we typically allowed to server to advance two frames ahead of the clients.

The server applies input events as soon as possible when receiving them from clients. Thus, input events suffer some latency from when they are executed by the user, from both the time for the event to travel from client to server and the number of frames the server is running ahead of client. Allowing the server to advance too far in front of the clients negatively impacts responsiveness of user input.

Likewise, the clients continue to update at 30 frames per second, sending events to the server. But if it too long a length of time passes with no response from the server, the client will temporarily halt until data are received.

The client stores simulation data for previous frames in a circular buffer whose length is large enough to hold data for any frame for which it may not have yet received data from the server. Once it receives a delta from the server, it will apply the delta to the appropriate frame, and then re-simulate from that frame until the current. However, since particles do not interact in the predicted simulation, the client needs to simulate only particles that were affected by the server data.

This of course means that the further ahead the server is allowed to go than the clients, the more noticeable the change will be once the delta is applied. In practice, we typically allow the client to get either zero or one frames ahead of the server to minimize this.

4.2.1 Synchronizing Input Events with the Clients

Certain input events, such as changing gravity, impact the predicted simulation as well as the real one. Therefore, all clients need to be notified about the change to the system so that their predicted simulations stay synchronized with the server. In practice, the server cannot apply such input events as soon as it receives them. It must choose the nearest frame in the future such that all clients will know about the input event before they simulate that frame. Increasing the time the client can advance without server data increases the distance to the nearest frame, thus impacting input event latency for such events.

4.3 Network Protocol

The client and server use TCP to communicate with each other. Due to the nature of the data, another protocol could possibly be developed on top of UDP that differs slightly from TCP, as many interactive applications such as games currently do, and would be a good avenue for future work.

Chapter 5

Results

This chapter discusses how well our algorithm performed in practice. We tested the program on several scenes with both single and multi-user scenarios. We looked primary at the quality of the compression and the achieved frame rate.

5.1 Evaluation of Compression Algorithm

We compared several different versions of the program to evaluate the impact of the compression algorithm. We analyzed both the network bandwidth used and the visual artifacts generated by compression. Each tested method used the same simulation but different methods of compression, as listed in table 5.1.

Table 5.2 lists the scenes on which we tested the algorithms. In each case, we used prescripted input events to ensure consistency for each run. The input events are simulated by clients as if they were regular user input.

The results are shown in table 5.3, which lists the average megabits per second required to run at 30 updates per second. Figures 5.1, 5.2, and 5.3 show bandwidth over time for some of the scenes, with accompanying images.

Discussion

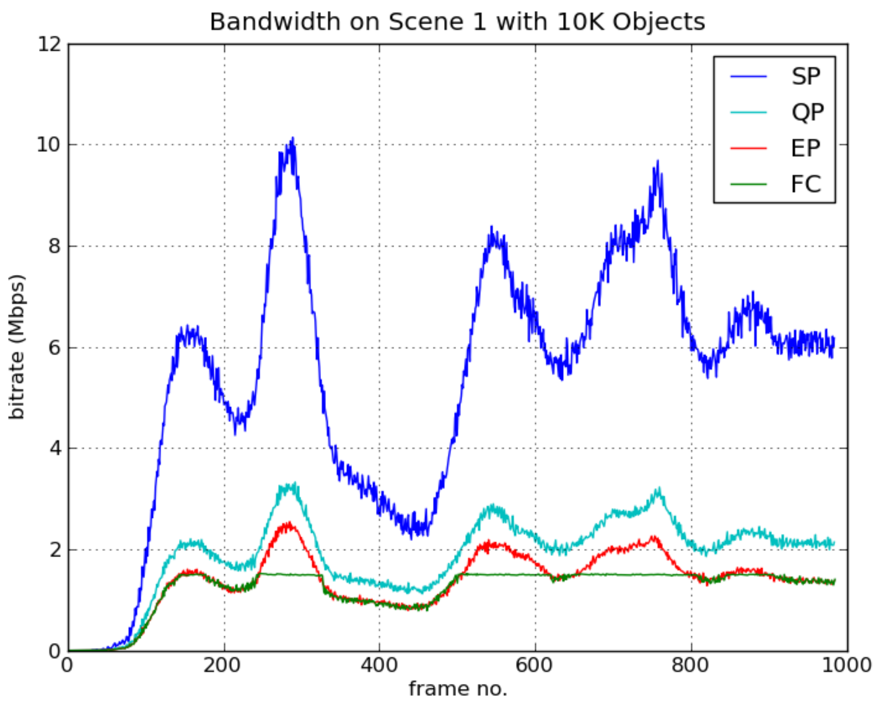
The effect of compression on data size is significant on most scenes. Even without the rate limiter, the compression shrunk the data to anywhere from about 2 to 10 percent of the original size. The data indicate that fast-moving objects do not compress as well as slow-

Algorithm	Description
No Compression (**NC**)	Sends the entire simulation state every frame.
Selective Particles (**SP**)	Uses error thresholds as described in 3.3.1, but does no compression on the resulting particles.
Quantized Particles (**QP**)	Same as **Selective Particles**, with the addition of quantization of particles as described in 3.3.2.
Entropy-Encoded Particles (**EP**)	Same as **Quantized Particles**, with the addition of using a Huffman encoder on the quantized particles as described in 3.3.2.
Full Compression (**FC**)	The entire algorithm, as described in the previous sections, including a rate limiter that prevents data from exceeding .05 Mbps. Is identical to **Entropy-Encoded Particles** with a rate limiter.

Table 5.1: Algorithms used to test bandwidth.

Scene ID	Description
1	Two diagonal ramps, forming a half-pipe. The objects start in a block above one pipe. Single user, with a few input events, and slow-moving objects.
2	A solid cube floating in the center. The objects start in a block above the cube. Single user, frequent input events, and slow-moving objects.
3	Four solid cubes floating in the center, arranged in a grid. The objects start in a block centered above the cubes. Single user, nearly continuous input events, and fast-moving objects.
4	Same as Scene 1, except with 2 users, each with frequent input events and slow-moving objects.
5	Same as Scene 1, except with 3 users, each with nearly continuous input events and fast-moving objects.

Table 5.2: Scenes for testing bandwidth. All use pre-scripted input events.



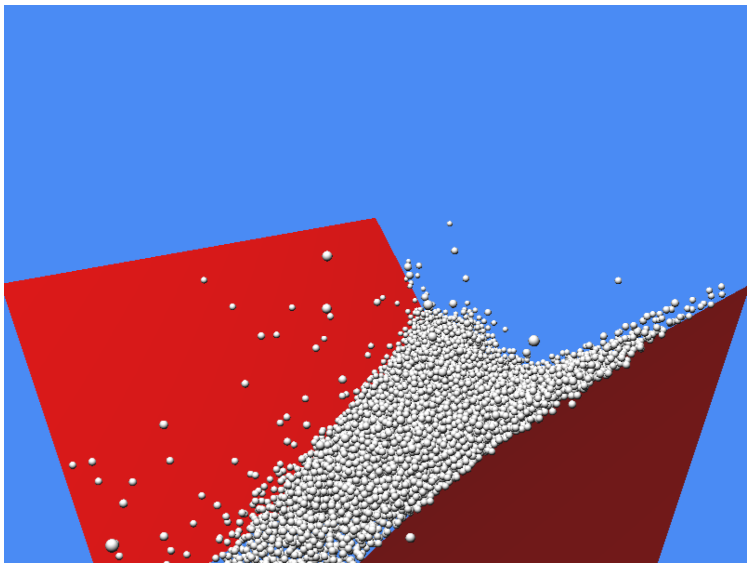
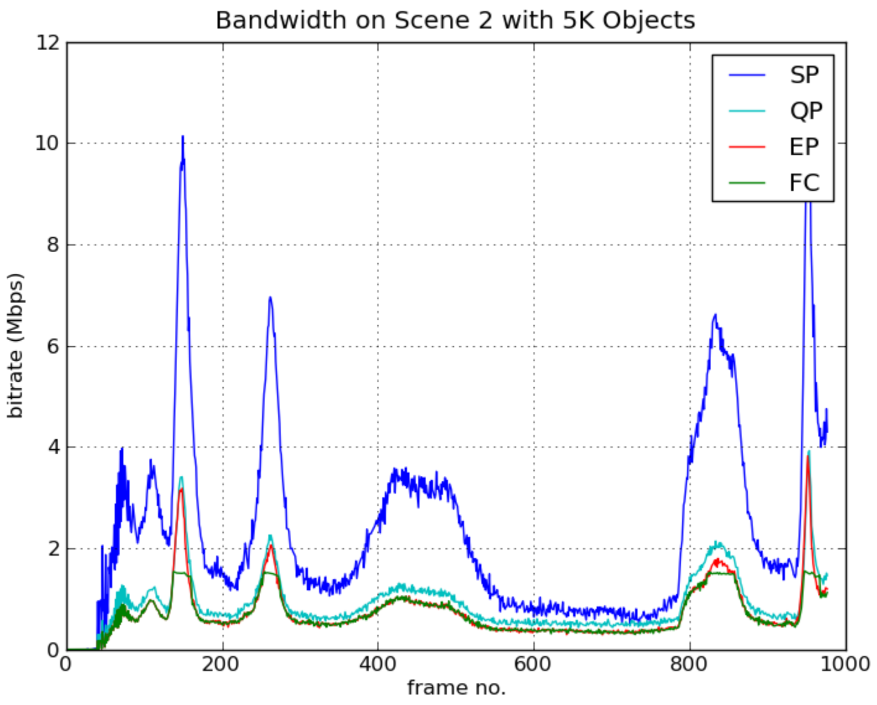


Figure 5.1: *Top*: Image from scene 1 with 10,000 objects. *Bottom*: Bandwidth over time for scene 1. Values for the **No Compression** algorithm are omitted both since it is constant and since the values are much larger than the other algorithms.



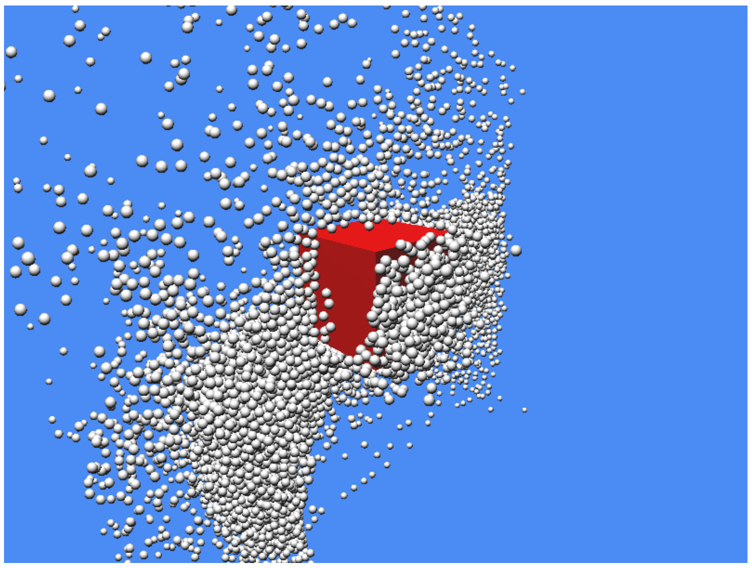
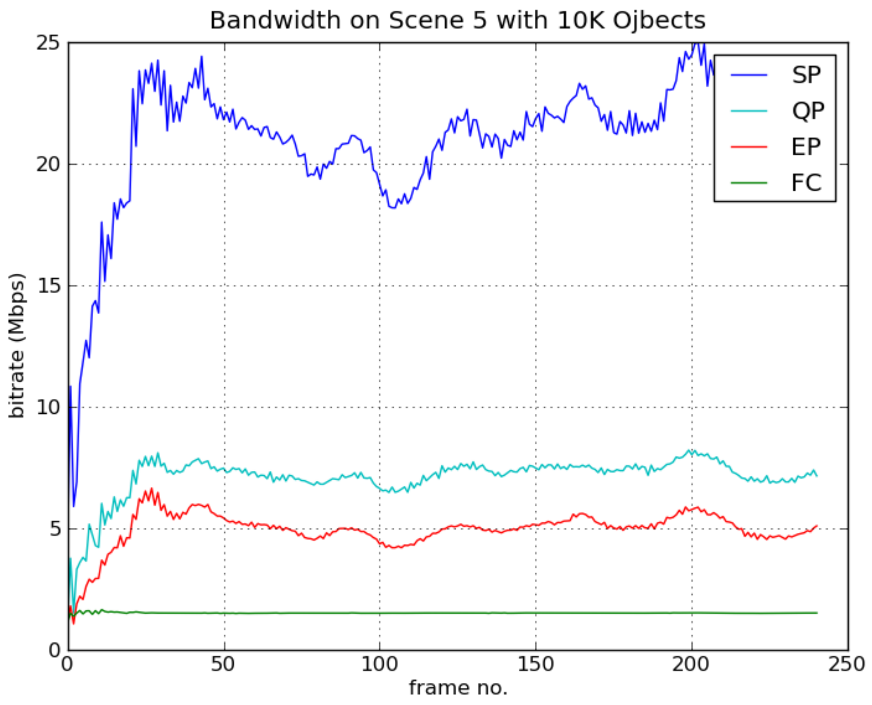


Figure 5.2: *Top*: Image from scene 2 with 10,000 objects. *Bottom*: Bandwidth over time for scene 2.



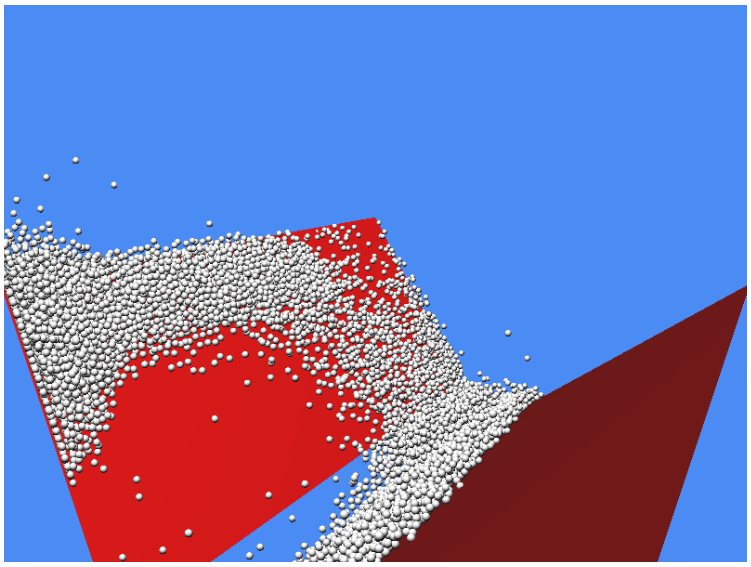


Figure 5.3: *Top*: Image from scene 5 with 10,000 objects. *Bottom*: Bandwidth over time for scene 5.

Scene			Average Bandwidth (Mbps)				
ID	# Particles	Avg. Collisions/Frame	**NC**	**SP**	**QP**	**EP**	**FC**
1	5,000	2,792	27.47	2.41	0.89	0.65	0.65
1	10,000	6,309	54.93	5.26	1.87	1.35	1.20
2	5,000	1,635	27.47	2.32	0.89	0.71	0.66
2	10,000	4,771	54.93	5.66	2.07	1.60	1.15
3	5,000	3,966	27.47	7.46	2.67	1.86	1.37
3	10,000	8,476	54.93	15.84	5.60	4.00	1.43
4	5,000	2,549	27.47	2.07	0.80	0.57	0.57
4	10,000	5,676	54.93	4.29	1.62	1.15	1.06
5	5,000	4,586	27.47	9.46	3.26	2.23	1.44
5	10,000	9,436	54.93	20.07	6.82	4.73	1.45

Table 5.3: Average bandwidth required to run at 30 updates per second. Averages are over the first 1000 update steps. Values are average megabits per second. Average collisions per frame lists the average number number of objects affected by collisions each update step.

moving objects, likely because faster speeds means particles diverge more quickly when collisions occur, so objects must be corrected more often. As expected, data size correlates highly with the number of collisions occurring in the scene. When all the objects are in the air, the bandwidth is very low, and likewise bandwidth spikes briefly when many objects collide simultaneously.

While the evaluation on how the compression algorithm perceptively affects the output was informal, we can make several observations. The rate limiter successfully reduces the size of the data, but at a noticeable cost. When the rate limiter prevents too many particles from being corrected, error builds up for a long while, and the eventual large corrections are quite obvious. To minimize the effect, the rate limiter should affect only a small number of objects or have affect only for a short period of time. It therefore works effectively for scenes that might have periodic spikes in data size but, on average, stay close to the limit.

Choosing effective thresholds for quantization bounds and deciding which particles to send also have noticeable impact. With a slightly too large threshold, objects corrections are plainly visible. In our scenes, we chose these values experimentally, though a good avenue for future research would be to have the program select these values automatically based on the data. Quantization bounds that are too large prevent the program from fully

Algorithm	Description
Client Only (**CO**)	Computes the full simulation on the client. The server is not used.
No Compression (**NC**)	The server computes and sends the entire simulation state every frame.
Entropy-Encoded Particles (**EP**)	Our method, as described in the previous sections, except without a rate limiter.
Full Compression (**FC**)	Our method, as described in the previous sections, including a rate limiter that prevents data from exceeding .05 Mbps.

Table 5.4: Algorithms used to test frame rate.

correcting all objects. On the other hand, quantization bounds that are too large create noticeable jitter since particles cannot be finely adjusted. In particular, objects at rest are very noticeable when they are corrected too roughly.

5.2 Evaluation of Frame Rates

The best evaluation of our method's success is how well it performs as an interactive application. We measured the achieved updates per second of our method (both with and without the rate limiter), as compared to two naïve solutions: using just the client machine or sending the entire simulation state from the server every frame. Table 5.4 describes the tested algorithms.

We ran the server on a desktop machine and ran the client on an iPod Touch connected to a consumer wireless router with broadband Internet. Due to having only two mobile devices for testing, headless desktop machines were used to simulate additional clients for the multi-user tests. As with the bandwidth tests, all user input events were scripted. Table 5.5 lists the test scenes. In multi-user tests, all clients are connected to the same router and consumer Internet connection.

Table 5.6 shows the results as average updates per second of the phone client. Since the client can only go a couple frames ahead of the server, and vice versa, this number is an accurate measure of the speed of the slowest component in the entire system.

Scene ID	Description
1	Two diagonal ramps, forming a half-pipe. The objects start in a block above one pipe. Single user, with a few input events and slow-moving objects.
2	A solid cube floating in the center. The objects start in a block above the cube. Single user, frequent input events and slow-moving objects.
3	Four solid cubes floating in the center, arranged in a grid. The objects start in a block centered above the cubes. Single user, nearly continuous input events, and fast-moving objects.
4	Same as Scene 1, except with 4 users, frequent input events and slow-moving objects.
5	Same as Scene 1, except with 4 users, nearly continuous input events, and fast-moving objects.

Table 5.5: Scenes for testing frame rate. All use pre-scripted input events.

Discussion

Our method greatly outperforms both naïve methods, anywhere from 4 to 20 times faster than the next-best method. In nearly all test scenes, both naïve methods performed more slowly than required for a smooth frame rate, while our method performed at or above interactive frame rates. For methods involving a server, frame rates are roughly correlated with data size, indicating that the network is the performance bottleneck. Even with 4 users on the same router, the program still ran at interactive frame rates. The bottleneck in the multi-user scenes was the shared router, which likely means that the server can support even larger numbers of users if they connect on different Internet connections. The data show that our method allows for much larger simulations that possible with more naïve methods.

Scene			Average Updates/Second			
ID	# Particles	Avg. Collisions/Frame	**CO**	**NC**	**EP**	**FC**
1	5,000	2,780	8.9	19.2	100.1	118.1
1	10,000	6,276	4.4	8.0	77.9	86.7
1	20,000	13,648	2.6	4.6	40.4	46.1
2	5,000	1,641	6.6	16.4	107.4	109.3
2	10,000	4,755	3.4	8.5	73.0	85.0
2	20,000	10,973	2.3	8.3	40.0	43.0
3	5,000	4,003	2.6	18.0	77.2	77.3
3	10,000	8,485	1.3	8.7	39.1	41.27
3	20,000	17,607	0.9	4.3	18.5	19.1
4	5,000	2,307	N/A	4.2	68.9	81.2
4	10,000	5,766	N/A	2.1	46.2	60.5
5	5,000	4,665	N/A	3.9	26.9	37.2
5	10,000	9,392	N/A	2.1	18.7	25.8

Table 5.6: Average updates per second of the server on the test scenes. Multi-user scenes have no data for the **Client Only** algorithm.

Chapter 6

Conclusion and Future Work

We have presented a general approach to running complex, interactive physical simulations on small portable devices and implemented the specific example of a large particle simulation running on iOS devices. By factoring particle simulation into two parts, we were able to run an interactive simulation with several thousand objects on a mobile device. Clients compute the easy integration step that has a large impact on the result, while a server computes the hard collision detection step that affects comparatively few particles. The server can therefore transmit a significantly compressed delta, allowing clients recover the full simulation state.

Our implementation works well in practice, achieving interactive frame rates with a mobile phone application over a consumer broadband connection. Many users can interact simultaneously with the simulation, using devices that could not compute the simulation on their own. Our method runs at frame rates several times higher than the naïve methods of running the simulation fully on the client or sending uncompressed simulation state. Since the client receives geometry data, clients' frame rates are decoupled from the system update rate. This leads to smoother frame rates and the flexibility of allowing client-side prediction in the event of temporary network problems. The main drawbacks to our system are some combination of additional user input latency and visual artifacts of the result.

The compression algorithm is far from optimal. There is likely much room for improvement with the data size, and the current implementation has several apparent artifacts. These artifacts typically manifest as objects noticeably snapping to new positions as their positions are corrected. Currently, the parameters for our compression algorithm (e.g., quantization range, limit for the rate limiter) are hard-coded constants, decided by experimentation. The program could instead automatically adjust those values during runtime.

Looking forward, we hope to apply this technique to other algorithms, Since our technique applies to simulations that are extensions of particle simulation, rigid-body, fluid, or cloth simulations would be ideal targets for our technique. More generally, our method applies to any simulation or computation which can be factored appropriately. Any portion of the computation that is easy to compute yet high in impact can be moved to the client, enabling more complex simulations than otherwise possible.

Though computing power will increase, portable devices will always remain behind desktop machines in computational ability. However, the predicted simulation will need to adopt more complex techniques than merely removing the expensive portions of the full simulation. There are many exciting avenues of research to explore in this area. Predicted simulations could run lower-fidelity versions of the full simulation, such as a grid-based fluid with different grid sizes on the client and server. Methods could be devised that automatically factors the simulation at runtime based on the capabilities of the client machine. Our technique even extends beyond physical simulation and can be used for any real-time computation on mobile devices that can be effectively partitioned. In general, we believe that even as computer power grows, our technique will remain effective in allowing portable devices to share the same interactive simulations as more powerful computers.

Bibliography

T. Amada, M. Imura, Y. Yasumuro, Y. Manabe, and K. Chihara. Particle-based fluid simulation on gpu. In *ACM Workshop on General-Purpose Computing on Graphics Processors and SIGGRAPH 2004 Poster Session*, 2004. 2

David Baraff and Andrew Witkin. Large steps in cloth simulation. In *Proceedings of the 25th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '98, pages 43–54, New York, NY, USA, 1998. ACM. ISBN 0-89791-999-8. doi: http://doi.acm.org/10.1145/280814.280821. URL `http://doi.acm.org/10.1145/280814.280821`. 2

David Baraff and Andrew Witkin. Physically based modeling. In *ACM SIGGRAPH Course Notes*, 2001. 2

Michael Deering. Geometry compression. In *Proceedings of the 22nd annual conference on Computer graphics and interactive techniques*, SIGGRAPH '95, pages 13–20, New York, NY, USA, 1995. ACM. ISBN 0-89791-701-4. doi: http://doi.acm.org/10.1145/218380.218391. URL `http://doi.acm.org/10.1145/218380.218391`. 2

Glenn Fiedler. Networked physics. In *Game Developers Conference*, 2010. 2

Glenn Fiedler. Networking for physics programmers. In *Game Developers Conference*, 2011. 2

Simon Green. Cuda particles. In *NVIDIA Whitepaper*. June 2008. 2, 3.2.1

Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics symposium on Computer animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. ISBN 1-58113-659-5. URL `http://portal.acm.org/citation.cfm?id=846276.846298`. 2

Jingliang Peng, CHange-Su Kim, and C.-C. Jay Kuo. Technologies for 3d mesh compression: A survey. *Journal of Visual Communication and Image Representation*, 16(6): 688–733, December 2005. 2

Gabriel Taubin and Jarek Rossignac. Geometric compression through topological surgery. *ACM Trans. Graph.*, 17:84–115, April 1998. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/274363.274365. URL `http://doi.acm.org/10.1145/274363.274365`. 2

Costa Touma and Craig Gotsman. Triangle mesh compression. In *Proceedings of Graphics Interface*, pages 26–34, 1998. 2

Cyril Zeller. Cloth simulation on the gpu. In *ACM SIGGRAPH 2005 Sketches*, SIGGRAPH '05, New York, NY, USA, 2005. ACM. doi: http://doi.acm.org/10.1145/1187112.1187158. URL `http://doi.acm.org/10.1145/1187112.1187158`. 2