A New Architecture for Cloud Rendering and Amortized Graphics

David Klionsky

CMU-CS-11-122

August 2011

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

Thesis Committee:
Adrien Treuille
Srinivasa Narasimhan

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

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

Keywords: amortized graphics, view-independent rendering, rendering architectures, cloud rendering

Abstract

High quality graphics and realism are essential features of modern games. Until recently, players needed to own expensive consoles or outfit their PCs with the latest hardware to play games with cutting-edge graphics. This has changed with the introduction of "cloud gaming" companies, which require only a good internet connection and minimal hardware to play the latest games. These ventures suggest that the gaming industry is moving to a cloud-based model, in which computation-intensive rendering and simulation is offloaded to remote servers and only the resulting images are streamed back to the client. However, present cloud gaming services still implement a one-console-per-user model, in which across-user computation, or computation that is independent of any particular user, is repeated for all users, rather than computed only once and then shared among all users in the same scene. We present a new architecture which amortizes the cost of across-user rendering, simulation, and memory. This architecture frees additional resources for rendering and allows for improved graphics at no additional hardware cost. To analyze the performance and generality of this architecture we implemented two applications: a light field and a fluid simulation. We also devised a general means of measuring the performance of amortized algorithms in this architecture.

Acknowledgments

First I would like to thank Eric Butler. This project is half his, and much of the design of this architecture is thanks to him. This project would not have been possible without him.

I'd like to thank Yantong Liu and Wei-Feng Huang for their extensive contributions to this project early on. I'd also like to thank Jake Poznanski, who worked briefly on this project before our subsequent work on networked physics.

I'd like to thank my advisor, Adrien Treuille, for his seemingly boundless energy and enthusiasm for this project. We couldn't have done it without his guidance and experience.

Finally I'd like to thank Srinivasa Narasimhan for being on my committee, David Andersen for answering all of our silly networks questions, and Deb Cavlovich and Catherine Copetas for coordinating my degree.

Contents

Chapter 1

Introduction

Modern games are inherently limited by the hardware on which they run. Game state size, processing speeds, and rendering capabilities are all determined by the hardware that is available, and affordable, to an individual user. Even with state of the art consoles and modern PC hardware, today's games do not approach the vast complexity and detail present in the real world. Even at the present rate of hardware innovation, these problems will not be solved soon. For example, high resolution fluids would require 11 doublings in processor speed to be simulated in real time on a single desktop [13].

In recent years a collection of new companies, foremost among them OnLive [17], Gaikai [16], and Otoy [18], have taken steps toward solving this problem by offering *cloud gaming* services, in which the game is run on a remote server and fully rendered images are streamed back to the user in real time. This eliminates the need for the user to own any hardware besides a controller and monitor, and relocates expensive or bulky hardware to a datacenter.

While cloud gaming is a paradigm shift in the way we play games, these early incarnations still follow a restrictive one-console-per-user model. Cloud consoles are virtualized, which allows for economic efficiencies to be achieved from maintaining less hardware, but the games themselves are still limited by the capabilities of the virtual hardware for which they were designed. Therefore, present day cloud services do not actually improve

the quality of graphics attainable in games.

Figure 1.1: Frame from the popular first person shooter *Crysis*. Many phenomena in this scene, such as global illumination, shadows, physical simulations, and character animation, are across-user phenomena, since they do not depend on the location of the viewer. [15]

Another limitation of this model is that it results in a large amount of wasted resources. Suppose we are rendering the frame in Figure 1.1 from the popular first person shooter, *Crysis* [15]. Some of the work required to render this frame is **across-user** or **view-independent** work, since it does not depend on either the location of the user or any specific information about the user. Examples of across-user phenomena in the frame include the moving trees, global illumination, and physical simulations in the frame. The rest of the work is **per-user** or **view-dependent** and does depend on this particular user, such as camera transformations and clipping operations. In principle, across-user work must only be done once, and can then be shared among all users, but in modern games and

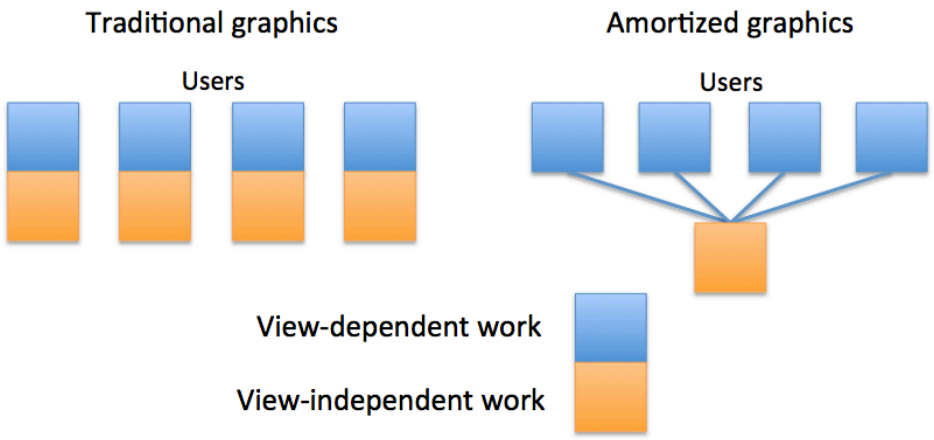

Figure 1.2: Comparison of amortized and non-amortized architectures. Assume that rendering one frame for one user requires one unit of per-user work and one unit of across-user work (bottom). A non-amortized architecture (left) duplicates all computation for each user, while an amortized architecture (right) computes view-independent data once.

present cloud architectures, both per-user and across-user data are computed by *everyone*, a significant waste of both computation time and memory (Figure 1.2).

We present an architecture that achieves the economic efficiencies of cloud computing while more importantly improving the graphics that are attainable in games. This is achieved by *amortizing* the cost of across-user computation and memory among all users, rather than multiplying it by all users in the manner of existing architectures. The building blocks of our architecture are computational nodes, or programs assigned specific tasks. "Frontend" nodes represent the users, and connect over a network to "backend" nodes, which act as servers for view-independent data. Frontend nodes perform any view-dependent work, and send fully rendered frames back to the user over a network.

This system amortizes the cost of across-user computation, since backend nodes work independently of the frontend nodes and are accessible by all frontend nodes. It also allows for larger simulations and increases in memory that were not previously attainable in consoles or cloud-based games. We call this branch of graphics that is focused on the separation of per-user and across-user work *multi-viewpoint graphics*, or *amortized*

graphics.

To test and demonstrate the generality of our architecture, we built two applications, a light field and fluid simulation. We found that while the light field amortizes memory well and could not have been run on conventional hardware, it does not amortize well computationally, and its expensive view-dependent operations prevent it from scaling well to many users. The fluid amortized well in both computation and memory and scaled well for multiple users.

We begin by discussing related work in amortized graphics and across-user rendering (§2). We then discuss the theory for what kinds of phenomena are well suited to this architecture (§3) as well as a new means of measuring the performance of applications in this architecture (§3.1). We then describe the architecture in more detail, independent of any particular rendering or simulation algorithm (§4). We describe the theory and construction behind our two test applications, the light field (§5.1) and fluid simulation (§5.2), as well as their results. Finally, we discuss future work that could be done to improve this system and what other kinds of applications could best exploit this new architecture (§6).

Chapter 2

Related Work

Existing graphics pipelines such as OpenGL [28] and Direct3D [3], are designed for per-user computation. For example, per-user operations such as clipping, viewport transformations, and rasterization are all parts of the original fixed-function pipelines in both systems. Due to the popularity of these systems, even some across-user phenomena have been designed to be computed in these heavily per-user pipelines. For example, Pharr and Fernando perform global illumination on the GPU with rasterization [26], and Mueller, et al. use polygonal meshes in screen space to smoothe the appearance of fluids [24].

Despite the dominance of OpenGL and Direct3D, architectures designed with a focus on across-user rendering have also been proposed. The PixelView architecture stores world-space rendering results, such as transformed vertices and geometry, in a 4D ray buffer, allowing new views to be synthesized from this buffer in real-time without recomputing any world-space transformation work [31]. The PixelView architecture differs from OpenGL in that it handles view information as late as possible in the pipeline, but it is still designed to function on a single machine. In contrast, our architecture is designed to allow for arbitrary arrangements of programmable pipeline stages. Our architecture is also not built around a ray-casting approach to rendering, and allows for any rendering algorithm to be used in the pipeline.

Much investigation has been done concerning parallel rendering [7, 36, 6]. Rendering

on large clusters has also been studied [27, 14, 9], and this work motivated our experiments in splitting a fluid simulation across multiple GPUs in our cluster. Using the parallel rendering taxonomy developed by Molnar et al. [23], our architecture could be described as using "sort-middle" techniques (since objects are distributed among backend nodes for rendering after some geometry processing) as well as "sort last" techniques (since fully rendered images are composited last on the frontend).

Our approach to passing data between pipeline stages was strongly inspired by the GRAMPS architecture of Sugerman, et al. [32], which allows for both fixed function and programmable pipeline stages to exchange data via queues. Their system also allows for arbitrary pipeline topologies, though ours are acyclic. The primary difference in our contributions is GRAMPS is designed to run on a single chip for one user, whereas our system is intended for multi-user applications running on a cluster of machines.

Rendering of across-user or view-independent phenomena has been well studied in computer graphics. Many across-user lighting techniques have been proposed, such as pre-computed radiance transfer [29], radiosity [11] and photon mapping methods [20]. There has also been work done on billboard approaches to rendering [8] and volumetric or voxel-based rendering [5], though these approaches primarily amortize memory and still require significant per-user computation. Most relevant to our system is the work on light fields, or lumigraphs [21] [12], which render by querying a database of precomputed light rays. Light fields make excellent candidates for amortized rendering because they trade amortizable memory for non-amortizable rendering time. For these reasons we implemented a light field as one of our two test applications (§5.1).

Physical simulations are also good targets for amortization. They can require large amounts of amortizable memory or state and updating that state is also amortizable, since the results can be used for any viewpoint. There has been significant work on physical simulation in graphics, especially in fluid simulations and character animation [1, 19, 2, 35, 22, 33, 34]. In particular, we adapt Stam's approach to real-time fluids to our architecture [30] to study the performance of a physical simulation (§5.2).

Distributed architectures for games include local area network (LAN) games and massively multiplayer online games (MMOs). LAN games are multiplayer games played

among a relatively small number of co-located machines on a network. While LANs can achieve low latency interactions, the game state and processing power available to a user are limited by that individual user's hardware. MMOs hold the game state on remote servers which players connect to over the internet and therefore have fewer memory limitations, but they are still limited by the rendering capabilities of the client machines. Our architecture has the advantages of both LAN and MMO approaches to games, allowing for both large game states and high quality rendering.

Several new companies are leading the way in relocating graphics computation to the cloud. OnLive [17], Gaikai [16], and Otoy [18] all offer cloud-based gaming services, in which all rendering and computation for games is done in a remote cluster. NVIDIA has also recently proposed RealityServer, a service offering high quality graphics for web applications through the cloud [4]. All of these offerings emphasize a separation between the expensive (and physically large) graphics hardware and the client device, which is typically a monitor or web browser. However, they do not exploit amortizable aspects of the graphics pipeline or innovate significantly in that pipeline. They also do not share significant data among multiple users viewing the same scene.

Chapter 3

Amortized Rendering

The basic principle behind amortized graphics is the fact that any graphics algorithm can be decomposed into per-user and across-user computation. Per-user computation by definition must be done for each user, while across-user computation only needs to be performed once. Therefore, if across-user work is done once and then shared among all users, the cost of that computation can be amortized by the number of users.

3.1 Measuring the Performance of Amortized Rendering

To distinguish between computation and memory amortization we establish two variables: α, the fraction of *computation* that is across-user work, and β, the fraction of *memory* that is per-user work. Any algorithm used in our architecture can be described in terms of these variables. Higher values for both of these variables are more desirable, since that means a greater fraction of the algorithm is amortizable.

In order to estimate α along with the added utility of using an amortized algorithm we have to measure three values:

- c_i: The across-user portion of the computation.
- c_b: The per-user portion of the computation that is computed on the backend (server).

- c_f: The per-user portion of the computation that is computed on the frontend (client).

These values refer to the fraction of computation required to render one frame for one user. So, if computation is measured in unit c, their units are c/user. We estimate c in seconds, though in principle it would be better to measure it directly in units of computation, such as floating point operations, which are independent of hardware speeds.

Given these variables, we can define

$$\alpha = c_i/(c_i + c_b + c_f)$$

Assume that c_i, c_b, and c_f are identical in both the amortized and "normal", unamortized versions of the algorithm. A single user must perform $c_i + c_b + c_f$ units of computation for one frame. The amount of computation required to support M users under the normal system is therefore

$$M(c_i + c_b + c_f)$$

In an amortized environment, c_b and c_f must be computed for ever user but c_i must only be computed once. Therefore the amount of computation required to support N users in the amortized version of the algorithm is

$$N(c_b + c_f) + \mathbf{1}c_i$$

The $\mathbf{1}$ in the above equation refers to a "unit user". It exists in the equation to make the units match between the two operands so we can reasonably add them.

For the sake of comparing amortized and non-amortized algorithms, assume that the total computation available to us is fixed, i.e. we cannot add or remove any hardware from our system, and all computation can be applied to any kind of work. Using the normal algorithm, we can support M users, and under the amortized algorithm we can support N users, without increasing or decreasing the available computation. Substituting our definition of α into the above equation, we can derive the relationship between M and N:

$$M = N(1 - \alpha) + \mathbf{1}\alpha$$

As the number of users becomes very large, we can ignore the constant α attached to the unit user:

$$M = N(1 - \alpha)$$

So, for every user supported by the original algorithm, the amortized algorithm can support approximately $1/(1 - \alpha)$ users. As α increases, more users can be supported compared to the non-amortized algorithm. This is a rough approximation, and for a small number of users the $\mathbf{1}c_i$ constant is non-negligible. For a large number of users, however, there are certainly gains from using the amortized algorithm.

A parallel argument can be made that applies to memory usage in the normal and amortized cases. Instead of α we use $\beta = m_i/(m_i + m_f + m_b)$, indicating the fraction of across-user memory used. We can then draw similar conclusions about how many more users an amortized algorithm could support when the amount of memory available is held constant.

Chapter 4

Architecture

4.1 Architecture Overview

At the highest level our architecture consists of a cluster of compute units which are partitioned into **frontend nodes** and **backend nodes**. This is designed to separate amortizable across-user work from non-amortizable per-user work.

Frontend nodes perform primarily per-user computation. There is typically one frontend node for each user, although each user does not require an individual or dedicated piece of hardware (i.e. nodes can be virtualized). Frontend nodes also track the state of the user in the world, such as the user's position and camera parameters. Frontend nodes receive input directly from the user in the form of button presses, mouse or joystick moves, and data from other input devices. Users connect to frontend nodes over the internet through a thin client, such as a web browser.

Backend nodes perform primarily across-user tasks, and hold any global or across-user state. Frontend nodes communicate with backend nodes via queues, and backend nodes are interconnected to share data when necessary (for example, in a networked simulation application). Since all backend and frontend nodes are co-located, it is possible to interconnect nodes with extremely high bandwidth, low latency connections.

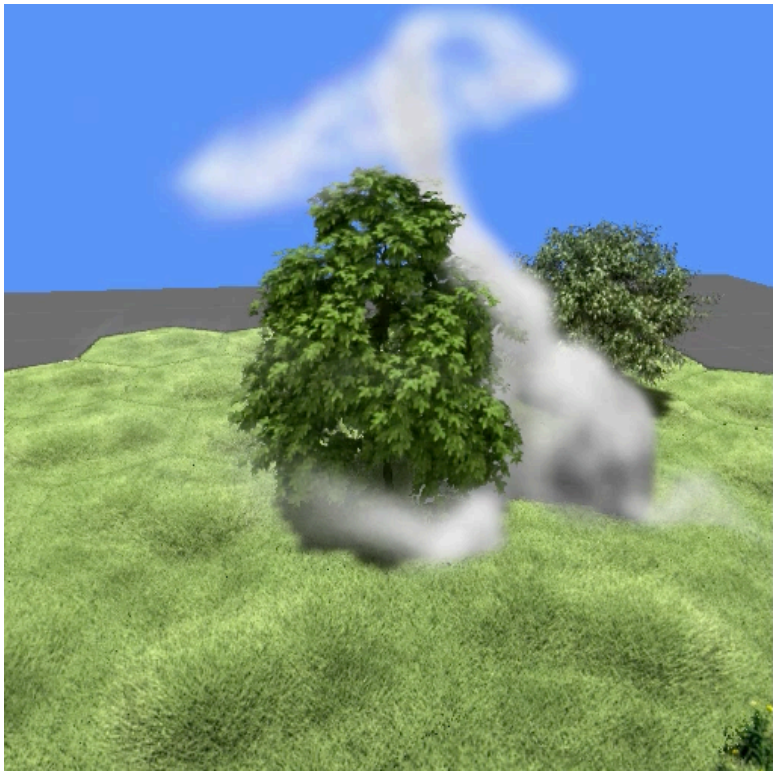

Figure 4.1: An example scene rendered using our architecture. The two trees, smoke, and hexagonal patches of grass are all separate objects, rendered by different backend nodes and composited into this final image by a frontend node.

The virtual world is decomposed into a set of objects. For example, in the scene in Figure 4.1, the tree, hexagonal patches of grass, and the smoke are all separate objects. Each object is owned by a particular backend node, and that node is responsible for maintaining the object's state and responding to requests for that object. A request can be for data, such as the positions of the particles in a particle system, or for an image, such as a rendering of a tree from a particular camera position. In the former case, the backend node is considered to be a **simulation node**, and in the latter the node is considered to be a **rendering node**. In some cases a group of nodes governs a single object, such as a networked physical simulation that is distributed among multiple simulation nodes. In general, backend nodes can be compared to web servers, since they are tasked with rapidly

responding to a large volume of requests from many different clients, or frontend nodes.

4.2 The Pipeline and Stages

Backend and frontend nodes are implemented as pipelines, within which individual stages communicate via queues, as shown in Figure 4.2. Since all stages share a common interface, it is not difficult to add new stages or replace existing ones with new implementations. For example, a backend node that renders a tree might initially be implemented with a light field, but we could later replace the rendering stage of the node with another implementation, such as volumetric billboards, without changing any other stages or the output type of the node.

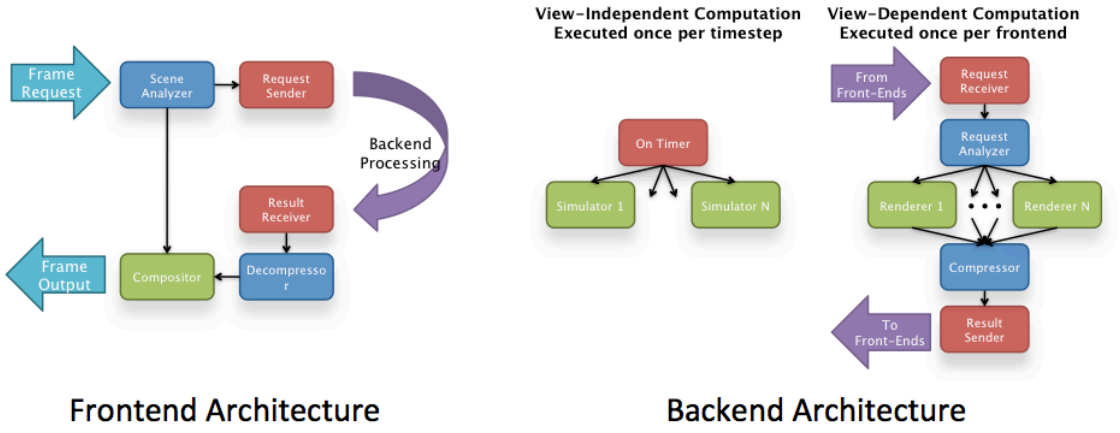

Figure 4.2: Layout of the pipeline stages in the frontend (left) and backend (right) components of our architecture.

The typical pipeline for rendering one frame for one user is as follows:

1. **Scene analysis**: In the frontend, the Scene Analyzer stage examines all objects in the virtual world and generates a list of the objects that need to be rendered for this frame. This is currently implemented by intersecting bounding boxes for objects with the camera's frustum. The Analyzer also sorts the visible objects in depth

order (back to front) for the Compositor, and then sends its sorted list of objects to both the Request Sender and the Compositor.

2. **Sending requests**: The Request Sender constructs a request for each object and sends each request to the backend node responsible for that object type. A request contains any information necessary for the backend node to satisfy the request. For example, if the backend node is a rendering node, then the Request Sender includes camera information in the request.

3. **Receiving requests**: Backend nodes handle new requests in a Request Receiver stage, which unpacks the request and sends the request data along to either a rendering or simulation stage, depending on the node type.

4. **Rendering**: If the backend node is a rendering node, it renders an image using the parameters in the request, and outputs an image. A rendering node can output multiple images, depending on the nature of the request.

5. **Simulation**: If the backend node is a simulation node, it outputs simulation data, such as particle positions or fluid velocities. It may also perform an update of its simulation, although usually updating the simulation occurs on a separate thread independent of frontend requests.

6. **Compression**: Result data sent from a backend can be compressed in this stage. Our current implementation employs a fast Gzip compression [10] to help reduce the amount of data sent between nodes.

7. **Sending results**: The backend completes its portion of the pipeline by sending its results back to the frontend node that sent it the original request.

8. **Receiving results**: The frontend node receives the data returned from the backend and marks that request as fulfilled.

9. **Decompression**: If the backend data was compressed, the frontend decompresses it in this stage.

10. **Compositing**: This stage commences once the result receiver marks all requests for this frame as fulfilled. At this stage all requests to backend nodes are complete, and the frontend node now has all the data it needs to render a frame for the user. In the first stage in the pipeline, the Scene Analyzer sent the Compositor a depth sorted list of the objects in the frame. The Compositor now uses the depth sorted list along with the result data from the backend to composite rendered images of the objects in the scene. If a backend simulation node returned simulation data, the Compositor performs the rendering for the simulated object. If the backend send depth data as well as color data, then the compositor can do more complex depth sorting.

4.3 Frontend Requests and Pipeline Utilization

In order to achieve the highest possible frame rate, a frontend node will send out requests for multiple frames without waiting for earlier frames to be completed. In Figure 4.3, the frontend waits until one full frame is completed before sending requests for the next frame. This is inefficient, since all stages of the pipeline can occur in parallel. Figure 4.4 shows a frontend which sends out multiple requests for frames before waiting for the first frame to complete. As a result, the pipeline is far better utilized, and the resulting frame rate is much higher. In principle it is possible for all stages to be running at 100% utilization at all times. If some stages are not being fully utilized, the ones that are fully utilized are bottlenecks and should be either parallelized or optimized. In our implementation, we refer to the number of frames that are requested before waiting for finished frames as the number of "frames in flight".

4.4 Issues and Limitations

We faced various challenges while designing this architecture.

Load balancing In a world with many users, an individual backend node often has to respond to a large number of requests every frame. This can also happen if there are

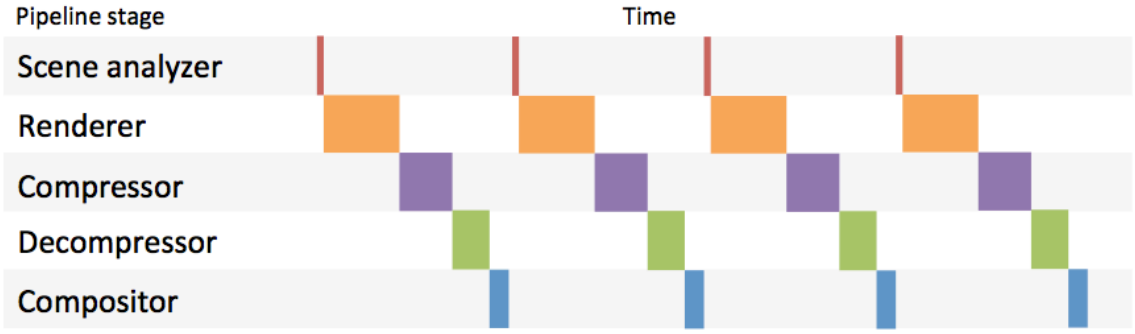

Figure 4.3: An example timeline showing active pipeline stages when there is only one frame in the pipeline at any given time. Sending only one frame at a time severely under-utilizes all stages and results in a lower frame rate. Stages that take negligible time, such as frontend request sender, backend request receiver, etc., have been removed to simplify the figure.

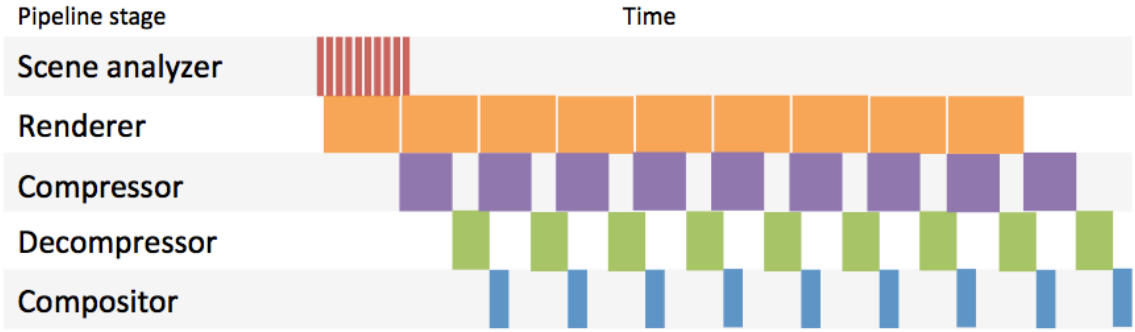

Figure 4.4: An example timeline in which the frontend requests multiple frames at a time without waiting for frames to complete. This timeline shows much better utilization and a higher frame rate than the previous timeline, since more work is being done in parallel. The bottleneck in this timeline is the stage with the highest utilization, rendering.

few users in the world but there is some object (owned by one backend node) that is duplicated in the scene many times. If a single backend receieves too many requests, the pipeline becomes bottlnecked by render time, reducing overall utilization. To prevent this scenario, backend nodes need to have some kind of load balancing,

similar to what is done with webservers. One solution involves constructing multiple backend nodes with duplicate data and splitting requests among them evenly. We did not have time to implement our own load balancing, so this has been delegated to future work.

Receiving data out of order Backend nodes make no guarantees about how long they take to fulfill requests. As a result, if the frontend sends requests for multiple frames to multiple backend nodes, one backend might return results for future frames before the other backends return results for earlier frames. In our implementation, we handle this by storing all backend results in a table of size requests by frames in flight. This problem is more difficult to solve when there are more objects in the scene or more frames in flight, since the frontend may run out of memory for storing results in this table.

Compression time Backend and frontend nodes are connected by high bandwidth connections, but bandwidth is never infinite, and it is usually necessary to compress result data sent by the backend. Unfortunately, compression and decompression can end up taking a significant amount of time, so choosing a good compression algorithm is of high priority.

per-user rendering Some backend rendering nodes render an object when given a set of camera parameters. Since this rendering uses frontend camera data, this is a per-user computation occurring on a backend node. In our implementation, this operation is performed on the backend because our light field rendering requires a large amount of amortizable memory. In contrast, the simulation step of a backend simulation node is fully across-user and amortizable in both computation and memory.

Algorithms must amortize well If the algorithm on a backend node does not have a large amortizable component, then there are no gains from this architecture. Not all algorithms are well suited to this system.

Chapter 5

Applications and Results

As a demonstration of the generality of our architecture, we implemented two algorithms: A light field for rendering large, complex objects in real-time, and a fluid simulation. The pipeline and all application stages were implemented in C++ and CUDA. We tested our algorithms on a cluster of 8 virtual backplane nodes, which resided on NVIDIA Tesla C2050 processors spread across two Silicon Mechanics Hyperform HPCg machines, each with 4 Intel Xeon E5620 Quad-Core 2.40GHz processors. Frontplane nodes consisted of a Dell Precision Workstation with on Dual Core Intel Xeon Processor, a 512MB NVIDIA Quadro FX 580 graphics card, and a Dell FX100 Remote Access device to enable remote rendering. With an InfiniBand interconnect and PCI 8 busses, all internal communication between CPUs and between CPUs and GPUs occured at 2 GB/sec.

5.1 Light field Rendering

Light fields simplify the problem of rendering meshes with high polygon counts or complex rendering effects, but are not used in many applications due to their tremendous memory requirements. Those memory requirements make them well suited to our architecture, since our backend nodes can collectively hold a significant amount of memory. Light fields make good candidates for amortization because these memory requirements are also fully

amortizable, i.e. all users viewing the same scene can query the same light field. Using the variables defined in section §3, light fields can be classified as having a very high β.

Our light field application renders several types of trees and different patches of grass, all in real-time with local lighting effects such as self-shadowing and subsurface scattering. The patches of grass can be tiled, allowing the grass and trees to be assembled in many unique ways.

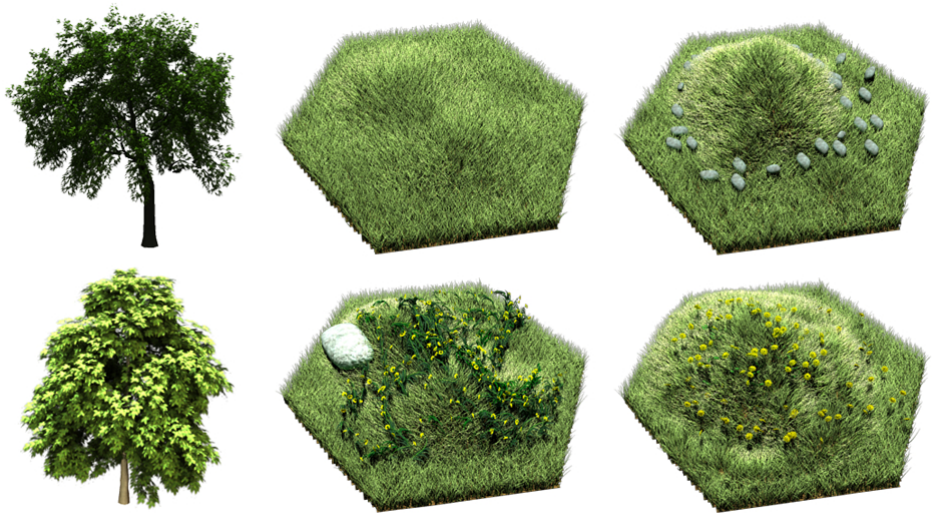

Figure 5.1: We created various organic objects for our light field application, a sample of which are shown here. The patches of grass were different in appearance but identical at the edges, allowing us to rotate and tile them when building the scene. Each object's original model contained several million triangles.

5.1.1 Construction

A light field is a function composed of thousands of views of a single object. When queried with a viewer's position and an orientation, it returns a new image of the object, generated by interpolating the precomputed views that are nearest to the requested view.

For our target objects, we acquired models of several types of trees from the Xfrog

Public Plants online database [25] and had an artist build several hexagonal patches of grass. We chose trees and grass because organic, natural objects have highly detailed appearances and are the most difficult to render realistically in real-time. The patches of grass differed in terrain and plant composition, but were identical along the edges, allowing us to tile them in our scene. Each of these models had on the order of several million triangles. Examples of some objects are shown in Figure 5.1. Using Maya, the trees and grass were modified to exhibit self-shadowing and subsurface scattering, as well as other global illumination effects. Some per-user phenomena were also be included, such as specular highlights and self-reflections. These effects add significantly to rendering time, but since this work was done as a precomputation step render time was not an issue.

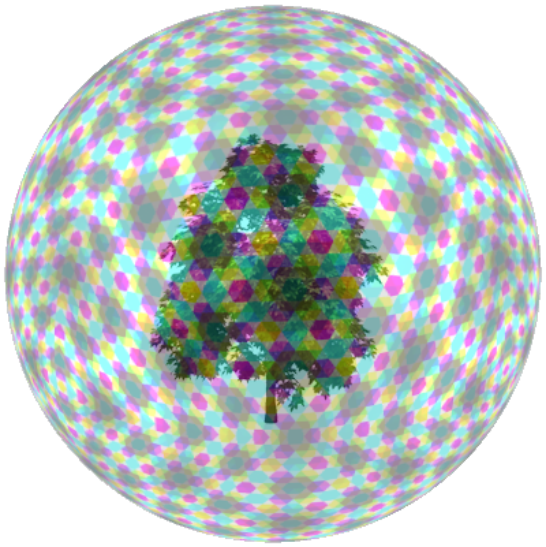

Figure 5.2: To generate images for our lightfield, we rendered views of each object from the vertices of a subdivided octahedron, shown here.

To generate a light field, one needs to sample the target object from a large number of different viewing positions around the object. To generate our sampling positions, we began with the vertices of an octahedron and spherically subdivided it for several iterations. We then took the vertices in the upper hemisphere, resulting in 8,321 final camera positions approximately in a sphere around the object Figure 5.2. Only the upper hemisphere

was used because this cut the memory required to store the light field in half, and we knew we would rarely be viewing light field objects from low angles. At each camera position, both a color image and a depth image were rendered at a 512x512 pixel resolution.

To synthesize a new image of the object at run time, the following is done for each pixel in the output image: rays are cast through the pixel and intersected with the sphere of precomputed camera positions. The precomputed images nearest the intersection point are queried, and the rays in those images that most closely match the intersecting ray are interpolated to produce a final color for the pixel.

5.1.2 Results

The light field is able to render high quality images of objects that could not otherwise be rendered in real-time. Several frames from our light field scene are shown in Figure 5.3. The trees exhibited subsurface scattering and interreflections of light among the leaves, as well as self shadows. If rendered using traditional methods, the scene would contain over 15 million triangles for the trees alone, and at least that many additional triangles for the grass. The scene also requires 24 GB of texture memory to render with light fields, which is not attainable on the hardware that would normally be available to a single user.

While the light field succeeded at doing what would otherwise be impossible, it proved to be far to per-user to be practical. The light field's strength is in memory amortization (a high β value), but rendering a single frame is *completely per-user work*, since the light field query uses the viewer's position. Rendering a single frame also requires a large number of texture lookups, exactly 32 per pixel (4 pixels from each of 4 textures for both color and depth data). GPUs are optimized to perform floating point operations but not memory reads, which decreased the benefits of running the light field on a GPU. Given these limitations we found that the backend could only process about 300 requests per second before rendering became the biggest bottleneck, which allowed four users to view a scene with 40 objects. This is an extremely small number of users and objects. As a result, the light field has too much per-user computation and does not scale well.

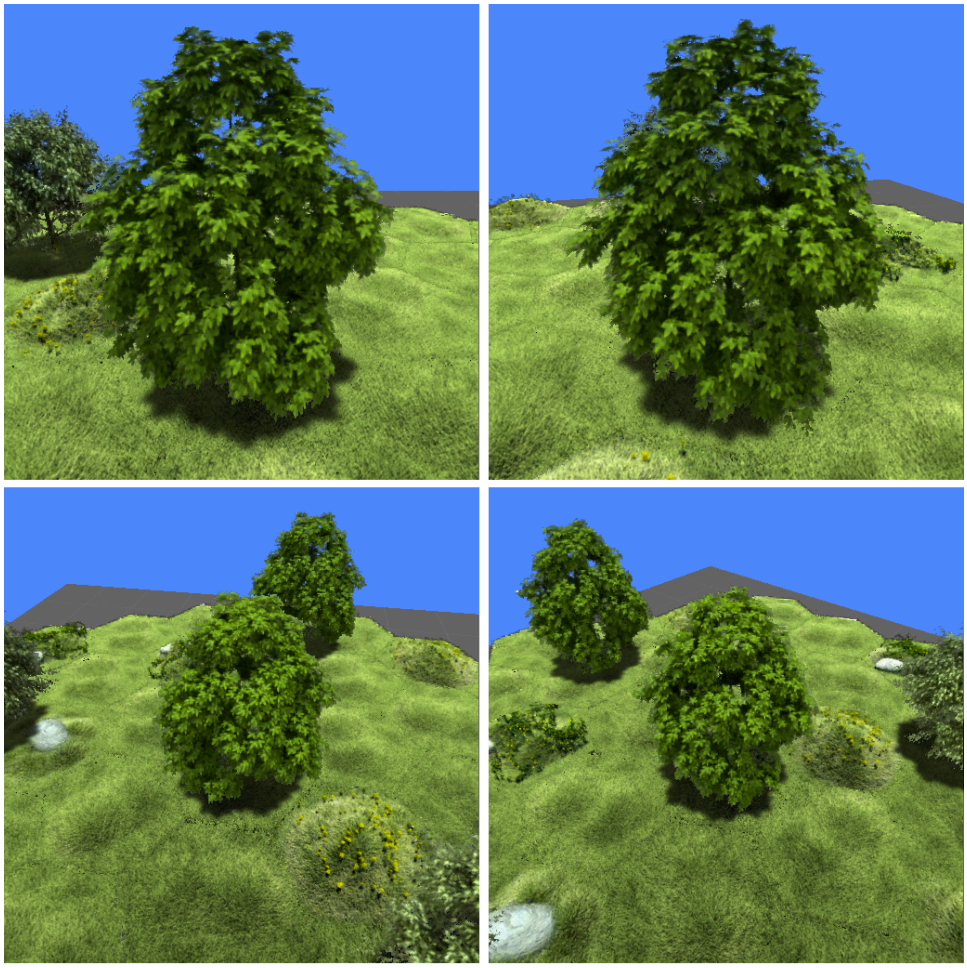

Figure 5.3: A light field scene as seen by four different players, who can view the world simultaneously.

5.1.3 Other Challenges

Ghosting Since the light field interpolates between multiple precomputed views to produce a new view, parts of the object that were visible from some views and not from others become "ghosted", or partially visible in the final image Figure 5.4. As a result, the rendered object looks like several faded objects layered on top of each other, instead of a single, distinct shape. The naïve solution to ghosting is to increase

Figure 5.4: When nearest color rays are interpolated to produce a final image, the blending of multiple views results in ghosting artifacts (left). Using depth data to interpolate between rays that strike a particular point on the model eliminates nearly all of these artifacts (right).

the number of camera positions around your object, but due to memory constraints this was not feasible. Instead we used depth images in addition to color to estimate the distance to points on the object. This estimated distance was used in the color interpolation, and all but eliminates ghosting artifacts.

Tiling artifacts Due to a large amount of noise in the depth estimate at the edge of an object, tiling of objects, like patches of grass, did not work perfectly. The seams along adjacent tiles were extremely visible, and required a lot of filtering though shaders on the frontend to reduce their effect. This edge noise would be reduced by an improved depth estimate, which could come from either a better depth estimation algorithm or higher resolution depth images.

Limited resolution Each of our backend GPUs had approximately 3 GB of texture memory, so all precomputed data for the light field had to fit within that limit. As a result,

we could only hold 8,321 raw PNG images for each object. Since each image was only 512x512 pixels, the light field looked best at that resolution, but not higher. This memory limit also put a bound on the number of camera positions that the light field could use. The more precomputed positions available, the higher the quality of the resulting image and the fewer ghosting artifacts that are visible.

Static world Due to memory constraints, our light fields do not contain a dimension for time. This causes the objects to be completely static. I.e. they can be translated and rotated within the scene but the objects themselves cannot change in any way. They are fixed to appear as they did when they were rendered in the offline precomputation step. For a natural scene with trees and grass, this detracts from the realism. Trees should sway back and forth, and leaves and flowers in the grass should flutter in the wind for the scene to look realistic. Even if a dimension for time could somehow be added, a looping animation of 30 to 60 frames (requiring a 30x to 60x increase in memory) would not be enough to produce realistic motion.

5.2 Fluid Simulation

For our second application we implemented a grid-based fluid, rendered on the frontend with particles. A physical simulation is much better suited to our architecture than a light field, since both the simulation step and complete state of a simulation are fully amortizable. Therefore, the fluid should have both high α and β values.

5.2.1 Construction

Our fluid simulation is based on Jos Stam's Stable Fluids implementation [30]. The state of the simulation is held in a 3D grid, with each cell containing a velocity for the fluid in that cell. We used a grid size of 64x64x64 cells for our implementation. In a simulation step, velocities are advected, pressure is calculated, and velocities are adjusted again to be divergence-free. To visualize the fluid, thousands of particles are injected into the flow and

advected by the velocities in the grid. The exact number of particles present varies over the course of the simulation, but it peaks at about 10,000. The backend returns these particle locations and velocities as its result for every request, making it a simulation node rather than a rendering node. The fluid also supports collisions with voxel data, so the fluid can interact with voxelized light field objects.

On the frontend, particles are rendered as gaussian sprites to create the appearance of smoke. The particles are also warped in the direction of their velocity (requiring velocity data to be sent from the backend as well) to obscure the boundaries between adjacent particles. Particles are shaded based on the number of particles above them to simulate shadows, so particles at the bottom of a column in the grid are much darker than particles near the top. Since the light field generates depth data, particles can be depth composited with the trees and grass in the scene by the frontend.

5.2.2 Results

Shown in Figure 5.5 are several frames from our fluid simulation. The simulation runs in real-time, and can interact with the other voxelized light field objects in the scene, as can be seen in Figure 4.1. The simulation amortizes well, since simulation steps happen on the backend independently of all frontends, and responding to a frontend request merely requires sending packets with particle position data. Therefore its scaling properties are good, and the only bottleneck is sending simulation data to frontends.

5.2.3 Analysis and limitations

Sorting If any kind of shading is applied (such as shadows), then particles on the frontend must be drawn back to front in order to be composited correctly. This sort can be time consuming and is ideally done on the frontend since it is a per-user computation. In our implementation this sort was done on the backend.

Rendering artifacts The only significant rendering artifact is a flicker when the sorting direction changes. Fluid particles are depth sorted along one of six cardinal direc-

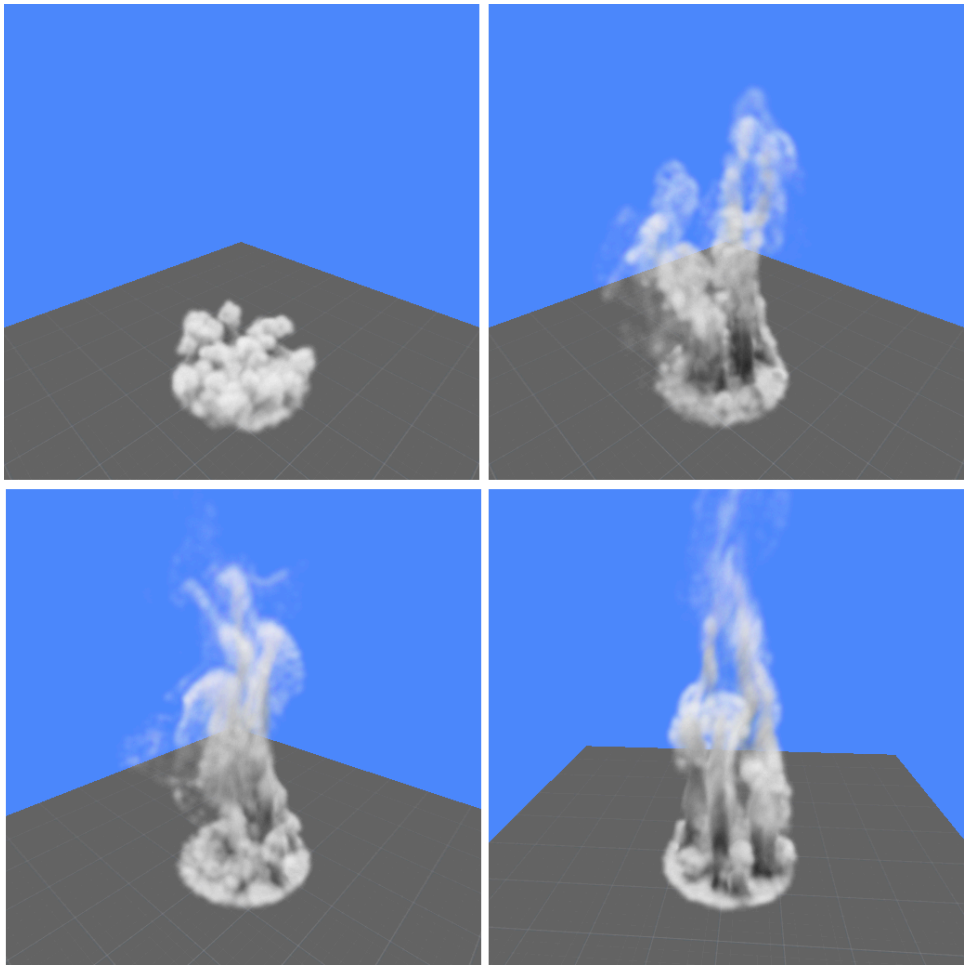

Figure 5.5: Four frames from our fluid simulation. The backend runs a grid-based fluid with particles injected into the flow. The backend sends the positions and velocities of these particles to the frontend, which renders them with gaussian sprites.

tions depending on the position of the viewer. When the viewer rotates around the fluid, the sorting direction will change, creating a visible pop or flicker among some particles which are no longer hidden by the shadows of their neighbors. This could be solved by a different rendering approach (such as with voxels instead of sprites) or by depth sorting the particles from the viewer's position, rather than from one of six cardinal directions and switching when the viewer moves enough.

Distributed simulation Our fluid runs on a single backend GPU which limits its size. It would be better if multiple backend nodes, or multiple GPUs on the same backend node, could each maintain a small section of a larger simulation and communicate with their neighboring simulations when information like fluid velocity or pressure must be exchanged. This would allow our fluid simulations to scale to virtually any size. We experimented with this but found that for a grid-based fluid, too much communication was required within each frame for this to be feasible in real-time. The main problem was in the projection step. In the stable fluids algorithm, projection is an iterative step that ensures the fluid is divergence free (i.e. the quantity of fluid entering each cell is equal to the quantity of fluid leaving each cell). Because this step is iterative, a single simulation node has to share pressure data with potentially 26 neighbors (in a 3D grid) each iteration. The node and all of its neighbors must communicate and perform the simulation in sync with each other as well fro this to be possible. These network costs, even over the fast connections between backend nodes, proved too expensive for a grid based distributed simulation to be practical. We believe a particle approach to fluids or a rigid body simulation would work much better, since the per-frame communication between adjacent simulations is much lower.

5.3 Performance

In section §3 we discussed how one could describe algorithms in our architecture with two variables: α, the fraction of computation that is across-user, and β, the fraction of memory that is across-user.

5.3.1 Estimating β

First let's consider the memory used in the two applications. In the light field, the full light field takes up approximately 3 GB of texture memory on the GPU. In a single frame, one user queries this lightfield for a single view of the object. This view is at most a

512x512 pixel image, which uncompressed at 4 bytes/pixel is 1 MB. We need two of these images, one for color and one for depth. As described earlier, computing each of these images requies 16 texture lookups per pixel, so the equivalent of 32 additional images are queried to generate the color and depth images. This data is per-user memory, but the rest of the memory used in the process — which is to say the light field structure itself — is view-indpendent, since it is used by all other clients making requests. Since the light field structure itself is nearly 3 GB, β for the light field is approximately $3 \text{ GB}/(3 \text{ GB} + 2 \text{ MB} + 32 \text{ MB}) = 3 \text{ GB}/3.033 \text{ GB} = 0.989$.

For the fluid simulation, the state of the fluid consists of a velocity for every cell in the grid, along with the positions and velocities for smoke particles. Since a simulation step updates the fluid for all users, and all users request the same set of smoke particles, this memory is entirely across-user . If the velocity in a cell is 12 bytes (1 float each for x, y, z), and a particle's position and velocity together are is 24 bytes (2 sets of 3 floats), the total across-user memory is $64^3(12 \text{ bytes}) + 5000(24 \text{ bytes}) = 3145728 + 120000 = 3265728 \text{ bytes} = 3.114 \text{ MB}$. per-user memory consists of the memory needed on the frontend to render particles. Since there are approximately 5000 particles, this equals all particle positions and velocities, or $5000(24 \text{ bytes}) = 120000 \text{ bytes} = 0.114 \text{ MB}$. Therefore β is approximately $3.114/(3.114 + 0.114) = 0.965$.

5.3.2 Estimating α

Though the light field has a β of nearly 1, its α is not nearly as high. This is because there is essentially no across-user computation done when rendering a frame from the light field. Rendering a frame requires firing rays from the client's camera position into their viewing plane, why by definition is per-user . No major sources of computation can be shared when rendering a light field, so the light field's α value is near 0.

In the fluid simulation, updating the fluid, advecting the particles, and calculating shadows are all completely across-user and amortizable, since all users can share the results from those stages. Depth sorting particles, rendering particles, and compressing the results are the major per-user sources of computation. In our implementation, the task of

depth-sorting particles was delegated to the backend, since this sort could be done more efficiently on a powerful GPU. As a result, when multiple frontends make requests for the same particles, the same set of particles must be sorted and *compressed* multiple times. In an ideal implementation, the per-user depth sorting would occur on the frontend, and particles would only have to be expensively compressed on the backend once. Given our actual implementation and this hypothetical ideal implementation, we can make two different estimates of α, shown in Figure 5.6. The first, α_0, assumes the ideal implementation, and doesn't consider compression time. The second, α_1, includes the compression time, which increases dramatically when users are added. By our estimates, α_0 is approximately 0.88, while α_1 drops rapidly as users are added, being as low as 0.48 with four users.

	100 frames		**500 frames**	
	1 frontend	**4 frontends**	**1 frontend**	**4 frontends**
c_i	67.11 ms	49.14 ms	59.88 ms	63.94 ms
c_{b_0}	0.76 ms	1.61 ms	0.61 ms	2.72 ms
c_{b_1}	17.95 ms	37.09 ms	14.58 ms	59.32 ms
c_f	9.56 ms	4.98 ms	7.77 ms	9.69 ms
α_0	0.87	0.88	0.88	0.84
α_1	0.71	0.54	0.73	0.48

Figure 5.6: Estimates of α for our fluid simulation in different scenarios. α_0 uses c_b from the renderable stage of the pipeline, while α_1 uses c_b from the compressor stage of the pipeline. $\alpha = \frac{c_i}{c_i + c_b + c_f}$.

Chapter 6

Conclusions

Modern games are heavily view-dependent and contain much per-user work. They are therefore limited in the graphics they can produce in real time. We have shown that by introducing a new architecture which amortizes the cost of computation and memory, we can achieve higher quality graphics and more complex worlds than were previously possible. Furthermore, we can do this with little or no increase in hardware cost.

In addition to presenting a new architecture, we also devised a new way to measure how well rendering and simulation algorithms amortize. Using the variables α and β, along with out results from light field and fluid simulation applications, we can now make better judgements about ideal amortized rendering algorithms. Physical simulations are excellent candidates because they have both high α and β. Algorithms that have high β and also require tremendous amounts of memory are ideal because they allow for graphics that were not possible on traditional hardware. Light fields, for example, require several gigabytes of texture memory for every object and allow for arbitrarily complex objects and lighting effects to be rendered in real-time. Motion graphs also require large amounts of memory and produce fluid, realistic motion. Both of these applicaitons are infeasible on normal hardware but are possible in our amortized architecture.

Algorithms which require a tremendous amount of *static* memory have an advantage since they can be used by users that are not in the same scene. For example, a light

field or motion graph can be queried by any user in any part of the virtual world, but a physical simulation can only exist in a particular part of the world and is therefore only visible to a subset of users. This limits the amount of amortization we can achieve with the simulation, despite its ostensibly high α. Another important factor to consider is how well the algorithm can be parallelized. A single users machine may only have several cores available for computation, but a large cluster could have many high end GPUs, each with many additional cores. The cluster is therefore capable of parallelizing an algorithm to its full potential.

Given our analysis of the light field and fluid simulation, we now have a better idea of what algorithms would serve as good avenues for future research. One application that lends itself well to amortization is a motion graph. Like a light field, the motion graph has a high β value, since it requires a large database of motion data that could not realistically be stored on a normal user's machine. It could also have a higher α value than the light field since many more lookups in the motion graph could be cached. A distributed physical simulation, in which multiple backend nodes communicated every frame to update the state of a much larger simulation, would be much better than our current simulation, since its state and size could be much larger. This is another example of a phenomenon that would not be possible without an amortized graphics architecture, though it would require a way to minimize communicate along the boundaries of simulation nodes. A major component in real-time video compression is motion estimation. After a frame is broken up into blocks of pixels, a motion vector is estimated for each block, and motion data is sent to the client for subsequent frames, rather than color data. This can be a time consuming step in compression. In our architecture, we have access to more information than typical video applications, since the frontend can have perfect knowledge of the layout of the scene. In principle, the frontend could use this information to compute motion vector data, rather than estimate it, allowing for much faster video compression.

Bibliography

[1] Jernej Barbič and Doug James. Real-time subspace integration for St. Venant-Kirchhoff deformable models. In *Proc. SIGGRAPH '05*, 2005. 2

[2] Jernej Barbič and Doug L. James. Time-critical distributed contact for 6-dof haptic rendering of adaptively sampled reduced deformable models. In *2007 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, August 2007. 2

[3] David Blythe. The direct3d 10 system. *ACM Trans. Graph.*, 25:724–734, July 2006. 2

[4] NVIDIA Corporation. Realityserver: The future of 3d web applicaitons. `http://www.nvidia.com/object/realityserver.html`, August 2011. 2

[5] Cyril Crassin, Fabrice Neyret, Sylvain Lefebvre, and Elmar Eisemann. Gigavoxels : Ray-guided streaming for efficient and detailed voxel rendering. In *ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games (I3D)*, Boston, MA, Etats-Unis, feb 2009. ACM, ACM Press. to appear. 2

[6] Thomas W. Crockett. An introduction to parallel rendering. *Parallel Computing*, 23(7):819 – 843, 1997. Parallel graphics and visualisation. 2

[7] Thomas W. Crockett and Tobias Orloff. A parallel rendering algorithm for mimd architectures. Technical report, 1991. 2

[8] Philippe Decaudin and Fabrice Neyret. Volumetric billboards. *Computer Graphics Forum*, 28(8):2079–2089, 2009. 2

[9] John Eyles, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, Nick England, and Lee Westover. Pixelflow: the realization. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware*, HWWS '97, pages 57–68, New York, NY, USA, 1997. ACM. 2

[10] Jean-Loup Gailly. The gzip home page. `http://www.gzip.org`, August 2011. 6

[11] Cindy M. Goral, Kenneth E. Torrance, Donald P. Greenberg, and Bennett Battaile. Modeling the interaction of light between diffuse surfaces. In *SIGGRAPH '84: Proceedings of the 11th annual conference on Computer graphics and interactive techniques*, volume 18, pages 213–222, New York, NY, USA, July 1984. ACM Press. 2

[12] Steven J. Gortler, Radek Grzeszczuk, Richard Szeliski, and Michael F. Cohen. The lumigraph. In *SIGGRAPH '96: Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, pages 43–54, New York, NY, USA, 1996. ACM Press. 2

[13] Eran Guendelman, Andrew Selle, Frank Losasso, and Ronald Fedkiw. Coupling water and smoke to thin deformable and rigid shells. *ACM Transactions on Graphics*, 24(3):973–981, August 2005. 1

[14] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordan Stoll, Matthew Everett, and Pat Hanrahan. Wiregl: a scalable graphics system for clusters. In *Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, SIGGRAPH '01, pages 129–140, New York, NY, USA, 2001. ACM. 2

[15] Electronic Arts Inc. Crysis. `http://www.ea.com/crysis-1`, August 2011. 1.1, 1

[16] Gaikai Inc. Gaikai is the open cloud gaming platform. `http://www.gaikai.com/about`, August 2011. 1, 2

[17] OnLive Inc. Cloud gaming: A faster, easier way to play. http://www.onlive.com/service/cloudgaming, August 2011. 1, 2

[18] Otoy Inc. Otoy: Movies and games rendered in the cloud. http://www.otoy.com, August 2011. 1, 2

[19] Doug L. James and Kayvon Fatahalian. Precomputing interactive dynamic deformable scenes. In *Proc. SIGGRAPH '03*, 2003. 2

[20] Henrik Wann Jensen. Global illumination using photon maps. In *Proceedings of the Seventh Eurographics Workshop on Rendering*, pages 21–30, 1996. 2

[21] Marc Levoy and Patrick M. Hanrahan. light-field rendering. In *Proceedings of SIGGRAPH 96*, Computer Graphics Proceedings, Annual Conference Series, pages 31–42, August 1996. 2

[22] Antoine McNamara, Adrien Treuille, Zoran Popović, and Jos Stam. Fluid control using the adjoint method. *ACM Transactions on Graphics (SIGGRAPH 2004)*, 23(3):449–456, August 2004. 2

[23] Steven Molnar, Michael Cox, David Ellsworth, and Henry Fuchs. A sorting classification of parallel rendering. *IEEE Computer Graphics and Applications*, 14:23–32, 1994. 2

[24] Matthias Mueller, Simon Schirm, and Stephan Duthaler. Screen space meshes. In *2007 ACM SIGGRAPH / Eurographics Symposium on Computer Animation*, pages 9–16, August 2007. 2

[25] University of Konstanz. Xfrog public plants. http://graphics.uni-konstanz.de/plantslib/, August 2011. 5.1.1

[26] Matt Pharr and Randima Fernando. *Gpu gems 2: programming techniques for high-performance graphics and general-purpose computation*. Addison-Wesley Professional, 2005. 2

[27] Rudrajit Samanta, Thomas Funkhouser, Kai Li, and Jaswinder Pal Singh. Hybrid sort-first and sort-last parallel rendering with a cluster of pcs. In *2000 SIGGRAPH / Eurographics Workshop on Graphics Hardware*, pages 97–108, August 2000. 2

[28] M. Segal and K. Akeley. *The OpenGL Graphics System: A Specification (Version 1.2.1)*. Silicon Graphics, Inc., April 1999. 2

[29] Peter-Pike Sloan, Jan Kautz, and John Snyder. Precomputed radiance transfer for real-time rendering in dynamic, low-frequency lighting environments. In *Proc. SIGGRAPH '02*, 2002. 2

[30] Jos Stam. Stable fluids. In *Proceedings of SIGGRAPH 99*, Computer Graphics Proceedings, Annual Conference Series, pages 121–128, August 1999. 2, 5.2.1

[31] J. Stewart, E. P. Bennett, and L. McMillan. Pixelview: a view-independent graphics rendering architecture. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*, HWWS '04, pages 75–84, New York, NY, USA, 2004. ACM. 2

[32] Jeremy Sugerman, Kayvon Fatahalian, Solomon Boulos, Kurt Akeley, and Pat Hanrahan. Gramps: A programming model for graphics pipelines. *ACM Transactions on Graphics*, 28(1):4:1–4:11, January 2009. 2

[33] Adrien Treuille, Yongjoon Lee, and Zoran Popović. Near-optimal character animation with continuous control. *ACM Transactions on Graphics*, 26(3):7:1–7:7, July 2007. 2

[34] Adrien Treuille, Andrew Lewis, and Zoran Popović. Model reduction for real-time fluids. *ACM Transactions on Graphics (SIGGRAPH 2006)*, 25(3):826–834, July 2006. 2

[35] Adrien Treuille, Antoine McNamara, Zoran Popović, and Jos Stam. Keyframe control of smoke simulations. *ACM Transactions on Graphics (SIGGRAPH 2003)*, 22(3):716–723, July 2003. 2

[36] Scott Whitman. *Multiprocessor methods for computer graphics rendering*. A. K. Peters, Ltd., Natick, MA, USA, 1992. 2