Natural Interaction with a Virtual World

by

Ilya D. Rosenberg

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

New York University

September 2013

Ken Perlin

Dedication

To my beautiful daughter Lilu, who reminded me of the incredible wonders possible in this universe and gave me the motivation I needed to get reinstated into the graduate program at NYU and complete my dissertation.

Acknowledgments

I would like to thank New York University, the Graduate School of Arts and Sciences, the Courant Institute and the Media Research Lab for accepting me into the graduate program and supporting my studies. I would especially like to thank Margaret Wright, the Director of Graduate Studies and Rosemary Amico, the Assistant Director of the CS department for helping me get reinstated in the program and putting up with my uncanny ability to do everything in the most difficult way possible and at the last minute.

I would like to thank my thesis advisor, Dr. Ken Perlin, who was flexible enough to let me work on what I wanted, yet caring enough to support me no matter what sort of issues I was having. Your insights and encouragement really laid the groundwork for my doctoral work and I think of you not only as a caring mentor but as a good friend.

I would like to thank all the other professors at the Media Research Lab, especially Dr. Denis Zorin and Dr. Davi Geiger for working with me on my fluid rendering and stereo vision projects. I learned a tremendous amount from the two of you, and will be forever grateful for your insights and guidance.

I would like to thank my teachers at Princeton, especially my advisor Dr. Brian Kernighan, who encouraged me to create technologies ten times better than what existed before, Dr. Thomas Funkhouser, who got me interested in computer graphics and Dr. Robert Sedgewick, who made me fall in love with algorithms and data structures.

ACKNOWLEDGMENTS

I would like to thank my office mate, Yotam Gingold, and the other students in the program including Adrian Secord, Matthew Grimes, Philip Davidson, Daniel Howe, Elif Tosun, Evgueni Parilov, Casey Muller, Denis Gueyffier, Denis Kovacs, Harper Langston, John Lee and Murphy Stein. Your warm friendship and brilliant but wacky minds made the Media Research Lab a very exciting place!

I would like to thank my co-founders and interns at Touchco, including Nadim Awad, Julien Beguin, Tomer Moscovich, Charles Hendee, Alex Grau and Merve Keles. Working with you was a wild ride, and it's truly incredible what we accomplished in the short nine months of Touchco's existence.

I would like to thank NYU's industrial liaison Robert Fechter for supporting the commercialization of IFSR and being such a pleasure to work with.

I would like to thank Alex Jaoshvili and Dr. Richard Brandt. Alex dragged me into the whole messy business of working with FSR due to his desire to improve the game of tennis and Richard supported and advised our first startup, SmartLines. Even though we didn't quite make it, I learned a tremendous amount working with the two of you.

I would like to thank Franklin Eventoff, the inventor of FSR, for showing me the ropes of FSR design and fabrication and building prototypes for us in his basement.

I would like to thank Jeff Han for introducing me to multi-touch interaction and using my lava-lamp app in a Ted talk!

Assembling hardware required custom manufacturing and support from many vendors. I would like to thank all the companies that have helped us with the manufacture of IFSR sensors including Sytek, Parlex and Nissha. I would especially like to thank Libing Zhang. I can't thank you enough for the countless varieties of experimental sensors that you built for us and for sharing your expertise in screen printing and manufacturing technology.

I would like to thank Ken Birdwell, Jeep Barnett, Jay Stelly, Chris Green, Brian Jacobson, Richard Lord, Steve Kalning, Lars Jensvold, Steve McClure, and everyone else at Valve for your enthusiasm and support of the RTPIE project. Working with you guys was a blast!

I would like to thank my college roommate and best buddy Rishi Sanyal for stumping me with various deep questions about photography and electronics and for always being there to talk about whatever was bothering me, and most of all, for taking as long to finish your PhD as I did.

I would like to thank everyone at Lab126 that supported IFSR and made working there fun, including Dave Buuck, Joe Hebenstreit, Aaron Zarraga, Connor Worley, John DePew, Brad Bozarth, Nidhi Rathi, Marcia Almanza-Workman, Tiffany Yun, Ed Liljegren, Reuben Martinez, Bob Olson, Chris Li, Sun Han, Ben Rosette, Omar Leung, Jano Banks and J.S. Yang.

I would like to thank my parents Yuri and Laura, brother Mark and grandparents Vadim, Lena and Yosif for their unconditional love and support. Mom and dad, thank you for your bravery in leaving the former Soviet Union and bringing me and my brother to the United States. Vadim, thank you for making me passionate about inventing new things, teaching me to take apart and fix almost anything, and instilling in me a curiosity for everything in the universe from a very early age. Yosif, thank you for asking me when I'm going to be done with my PhD every single time I came to visit. I hope I have made all of you proud.

Last but not least, I would like to thank my loving wife Fang for supporting all my projects and ideas, helping me start Touchco, and putting up with my unconventional work habits. I couldn't have done it without you!

Abstract

A large portion of computer graphics and human/computer interaction is concerned with the creation, manipulation and use of two and three dimensional objects existing in a virtual world. By creating more natural physical interfaces and virtual worlds which behave in physically plausible ways, it is possible to empower non-expert users to create, work and play in virtual environments. This thesis is concerned with the design, creation, and optimization of user-input devices which break down the barriers between the real and the virtual as well as the development of software algorithms which allow for the creation of physically realistic virtual worlds.

Table of Contents

TABLE OF CONTENTS

List of Figures

LIST OF FIGURES

Introduction

0.1 Motivation

At the time that I worked on the projects described in this thesis, most computer interaction was done via a keyboard, mouse or digital stylus. I envisioned that we would one day create technologies that would allow us to interact with computers by naturally using our hands and bodies in free space or on virtual surfaces, without having to hold or touch mechanical implements, and that the virtual worlds that we would interact with would be comprised of virtual particles which, while not quite as small as atoms, would be able to capture and simulate the behavior of physical matter in a realistic way.

Today, much of this vision has come true. The Microsoft Kinect [26] has revolutionized the way in which people interact with video games, using whole body motions to control virtual avatars, and the LeapMotion [27] allows 3D tracking of fingers, enabling natural interaction with 3D objects in free-space. Touch surfaces have become ubiquitous. Most people carry one in their pocket in the form a smartphone or in their backpacks/purses in the form of a tablet. Particle-based fluids, although not yet very common have been used in games such as Portal 2 [9], to create challenging fluid-based puzzles in a virtual world.

One common theme found in my work is the focus on algorithms and technologies that allow for real-time, highly accurate and highly responsive interaction.

Also, I have focused on avenues of research that could have broad commercial applicability in the hopes of having a real impact on the average user of computing systems. Although only a fraction of the work that I have done as part of my thesis has become commercially successful, I hope that my work has served to inspire others to further explore these interesting directions in graphics and human-computer interaction research.

0.2 Thesis Organization

This thesis is organized along three topics, roughly related to three avenues of research that I pursued during my time at New York University. The first topic (Chapter 1) is concerned with the creation of a real-time GPU-based algorithm to generate high-quality dense disparity maps from stereo images, which I call SGMGPU. Applications such as robot navigation, real-time manipulation of 3D objects, gaming and simulated camera effects such as depth-of-field are discussed.

The second topic (Chapter 2) is concerned with the creation of a real-time algorithm for rendering deformable objects that are comprised of particles, which I call RTPIE. Although this algorithm was initially developed to render fluids in unbounded virtual worlds, it has also been used to create virtual blobby creatures, a three-dimensional clay-sculpting tool, and an interactive touch-based lava lamp.

The third topic (Chapter 3) is concerned with the design and development of a technology called IFSR which enables novel pressure-sensitive multi-touch surfaces. The development, manufacture and evaluation of an IFSR based input pad, called the UnMousePad is discussed, as well as possible uses for the technology.

1 SGMGPU - Semi-Global Matching on a GPU

1.1 Introduction

This chapter describes my work in the field of computer vision. This work was originally motivated by the desire to give computers the ability to see depth the way humans do. Furthermore, I wanted to create an approach that would work in real-time, allowing the algorithm to be used for applications such as robot navigation, helping blind people to navigate unfamiliar environments and tracking the human body in order to enable natural human-computer interaction. My goal was to develop an algorithm with the following characteristics:

1. Real-Time - The algorithm should be parallelizable so that it can run efficiently on a SIMD processor or a programmable graphics card (GPU), and should have a consistent run-time regardless of the input.

2. Robust - The algorithm should behave consistently in different environments and lighting conditions, and should be able to detect areas where it is unable to compute depth with confidence.

3. Per-Pixel - The algorithm should be able to compute depth for every pixel in an image.

1 SGMGPU - Semi-Global Matching on a GPU

At the outset of the project, I developed an algorithm based on dynamic programming. I chose to use dynamic programming because the problem of finding optimal matches between two images along an epipolar line is very similar to finding a lowest energy path through a two-dimensional array. My first attempt at a solution treated each horizontal line of pixels separately, which created banding across the resultant depth image and performed poorly compared to more common windowed approaches because it couldn't make use of information from neighboring epipolar lines.

To reduce banding, I decided to add the ability to pass information between epipolar lines. However, although I was able to develop several heuristics similar to those described by Birchfield and Tomasi [3], I could not find a way to do this optimization in a way that would arrive at the global solution. I soon discovered that what I was attempting to do – dynamic programming in three dimensions – was known to be an NP-complete problem.

In searching for a solution, I discovered a dense stereo algorithm called Semi-Global Matching (SGM) [20] which avoided this problem and was ideally suited to implementation on a SIMD processor such as a GPU. Because it was based on dynamic programming, it's run-time was consistent no matter what the input was. Furthermore, the SGM algorithm performed better than most other algorithms on the "Middlebury Stereo Evaluation Version 2" [39] when run on natural scenes such as the Cones and Teddy data-sets.

Starting with a CPU implementation that I then optimized with SSE3 SIMD instructions, which could not achieve real-time performance, we set about porting the algorithm to run in real-time on a GPU, freeing the CPU for other tasks. Our resulting algorithm was able to take a live stereoscopic video feed and generate dense disparity maps in real-time. We also developed several shaders which allowed us to manipulate the generated video stream in real time using the extracted depth information to perform operations such as simulating depth of field, and performing edge extraction and object separation using depth information. This work was presented at SIGGRAPH 2006 [37].

1.2 Algorithm

Most GPU stereo algorithms aggregate errors across blocks, an approach which is known to wash out fine detail. In contrast, the SGM algorithm uses dynamic programming to capture detail. To avoid horizontal streaking, SGM performs dynamic programming along N directions (typically 16) in the xy plane of a (W*H*D) disparity volume. At each xy position, it sums up the costs along the z axis to find the minimum cost path that ends at pixel p and disparity d.

The most straight-forward implementation of the algorithm processes one direction at a time, starting with each xy position on the edge of the volume, thus performing approximately (N/2-1)*(W+H) passes. The computed costs are stored in a (W*H*D) error volume and the final disparity at each pixel is taken to be the disparity with least cost. Post processing steps are used to detect occluded regions as well as regions where disparity was computed incorrectly with a consistency check, to compute sub-pixel disparity, and to detect and fill occluded areas.

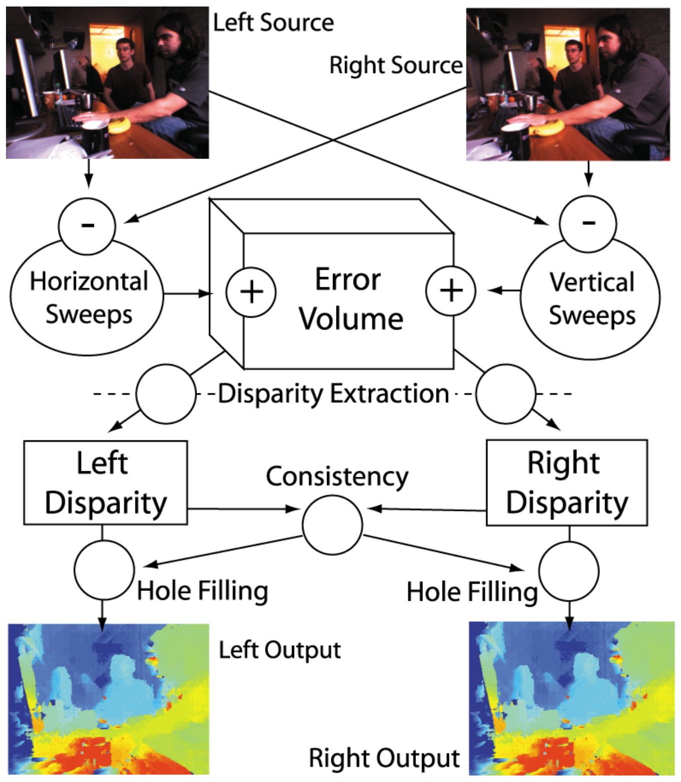


Figure 1.1: Simplified GPU data flow diagram. Differences between the left and right images are fed through the vertical and horizontal sweep shaders which add errors to a 320x240x64 error volume. Disparity is extracted for the left eye and right eye images and compared to generate a consistency map. Inconsistent pixels are filled in with depth from neighboring pixels and then a shader is used to generate a heat map based on the depth of the pixels in both left and right images.

1.3 GPU Implementation

The original SGM algorithm was not suitable for GPU architectures because it operated on a single column of the disparity volume at a time. Instead, we desire to make the minimum number of passes through the disparity volume, while in parallel, operating on the largest number of paths.

We accomplish this by storing dynamic programming costs in an RGBA sweep texture, which is as wide as the max disparity and as tall as the image height or width (depending on sweep direction). Each component of the texture contains temporary data for one direction, thus we perform dynamic programming for four directions in one sweep. We perform one horizontal and one vertical sweep per video frame, yielding 8 directions, giving sufficient quality for real-time purposes.

The accumulated costs are then packed and additively blended into the error volume which is stored as a 2D texture of 4x4 RGBA blocks where each component represents a cumulative error at a given depth, for a total of 64 depths. Next, an extraction step extracts disparity information from the error volume into two textures containing per-pixel disparity and match costs for the left and right depth maps. Finally, a consistency shader compares these to find occluded areas and a hole-filling shading fills them. (see Figure 1.1)

1.4 Results

Our demo runs on a modest desktop PC. The CPU is tasked with acquiring and rectifying stereo image pairs from a Point Grey Research Bumblebee stereo camera, and feeds them to a set of shaders running on an nVidia 7900GTX video card (Figure 1.2). The GPU interface was implemented using software we derived from the OpenVIDIA project [16]. The software invokes shaders which compute depth maps and render them to the display along with post-processing effects.

At a resolution of 320x240 and max disparity of 64, our implementation runs at 8fps. At 160x120x32, it runs at 13fps. This includes the time necessary to acquire

Figure 1.2: We used an nVidia 7900GTX video card to run the SGM algorithm and a Point Grey Research Bumblebee camera to acquire stereo images.

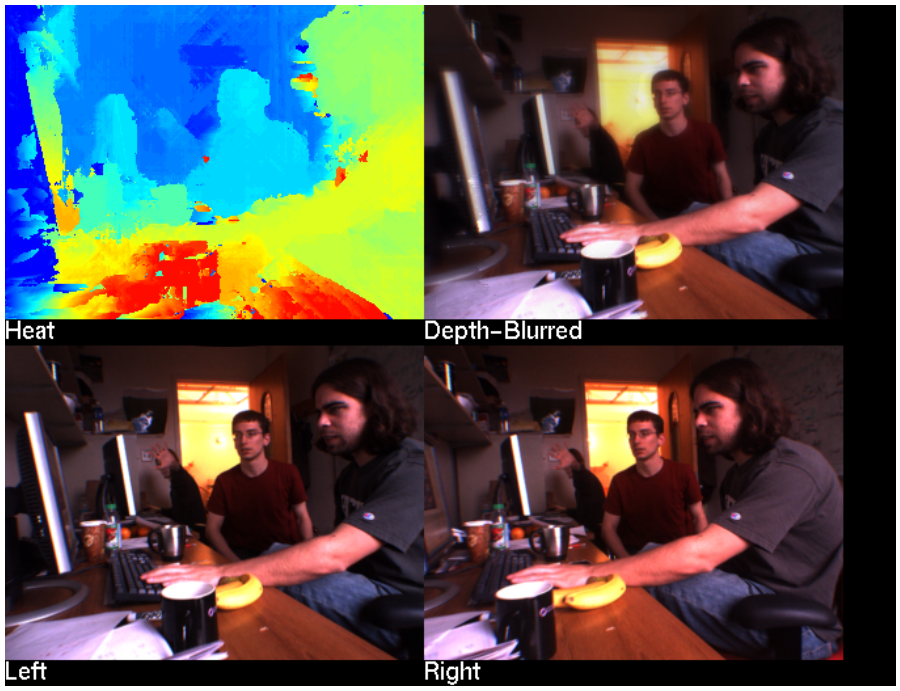


Figure 1.3: Screenshot of SGMGPU interactive program running with a 320 x 240 image resolution. The computed disparity map, shaded with a heat-map is in the upper left, and a real-time depth-of-field shader applied to the input video is shown on the upper right. The left and right input images are shown below.

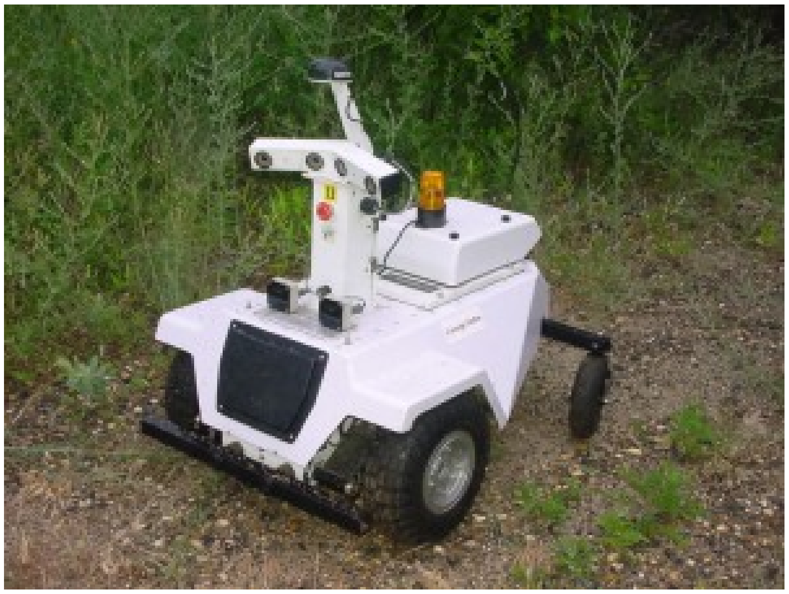


Figure 1.4: One of the autonomous LAGR robots. The SGM algorithm was used to generate ground-truth disparity data to train LAGR's neural networks for obstacle avoidance.

and rectify images, which we did not accelerate as part of the project. Without image acquisition and rectification, the demo runs at 10fps and 24fps respectively.

Even at these frame rates, there is significant room for improvement in data acquisition and shader implementation, and we believe that a 2x speedup is easily possible. Furthermore, tremendous improvements would be possible if the video card supported rendering to 3D texture slices, and taking minimums across a texture row efficiently. Although our implementation performed sweeps in only 8 directions, the quality of the depth maps was almost indistinguishable when compared to the original 16 direction algorithm.

In addition to the realtime interaction applications we described in our 2006 SIGGRAPH presentation, our algorithm was also used to generate ground truth

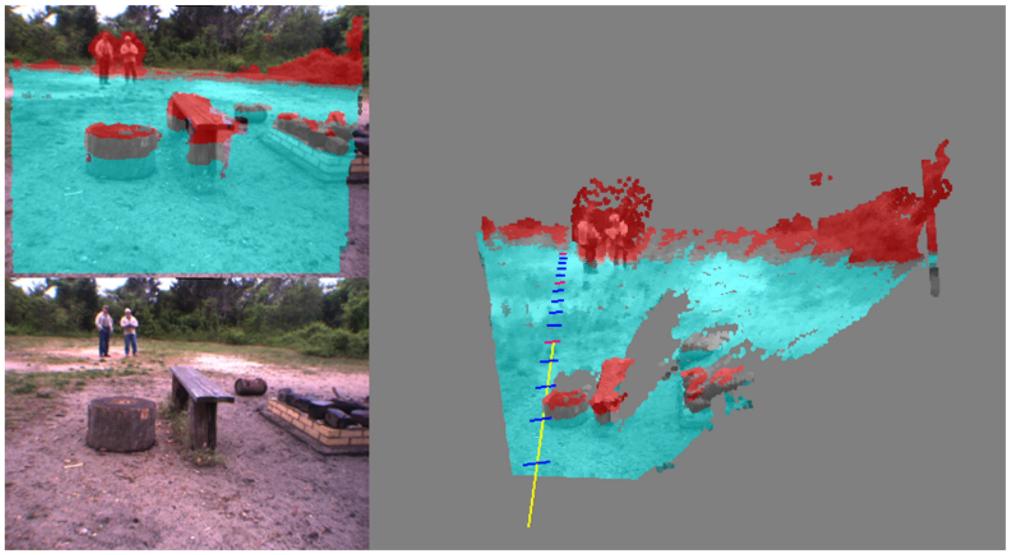


Figure 1.5: After training a set of neural network classifiers with depth information extracted using our algorithm, the LAGR robots were able to classify the space around them as navigable (blue) and obstacle (red).

depth information as part of the LAGR (Learning Applied to Ground Robotics) project [17]. The data was used to train a neural network classifier which was used for obstacle avoidance on the LAGR robots (Figures 1.4 and 1.5). Our implementation yielded significantly better disparity maps than the SAD (Sum of Absolute Differences) stereo algorithm used by the Triclops SDK [33] that came standard with the Point Grey Bumblebee cameras used on the LAGR robots.

Some comparison images of our algorithm operating on the LAGR dataset are presented in Figure 1.6 and Figure 1.7. These images are taken in highly unstructured outdoor environments which are a very difficult test case for most stereo algorithms. The images in the figures were generated using the same GPU algorithm described above, running at 320x240x64 resolution.

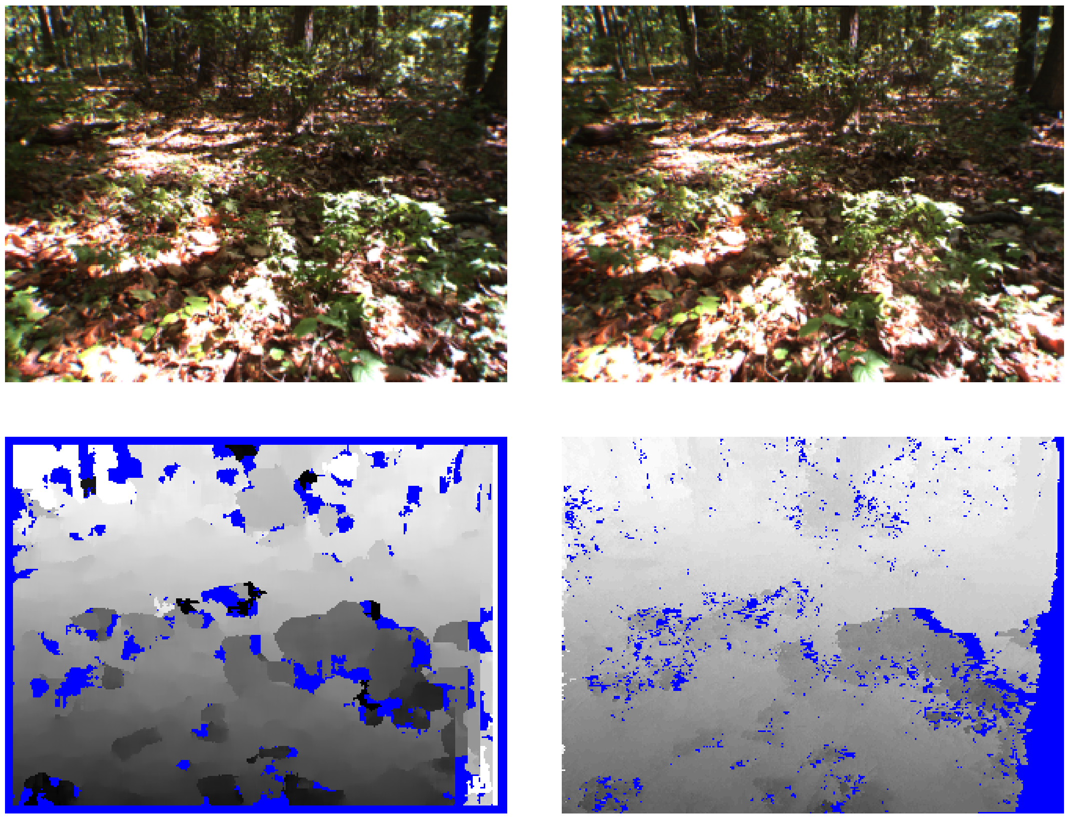


Figure 1.6: Comparison of SGMGPU with the Triclops SDK's algorithm on LAGR test data. The upper images are the left and right stereo input images. The bottom left shows the output of the Triclops SAD algorithm. The bottom right shows the output of our SGM algorithm. Note that our algorithm is much better at resolving the leafy branch in the right side of the images as well as the trees in the distance. Furthermore, we are able to extract depth for the entire image frame, and accurately report occluded areas in blue. In contrast, the SAD algorithm has an area all around the image for which depth is not computed and has several black and white splotches where depth was not computed correctly.

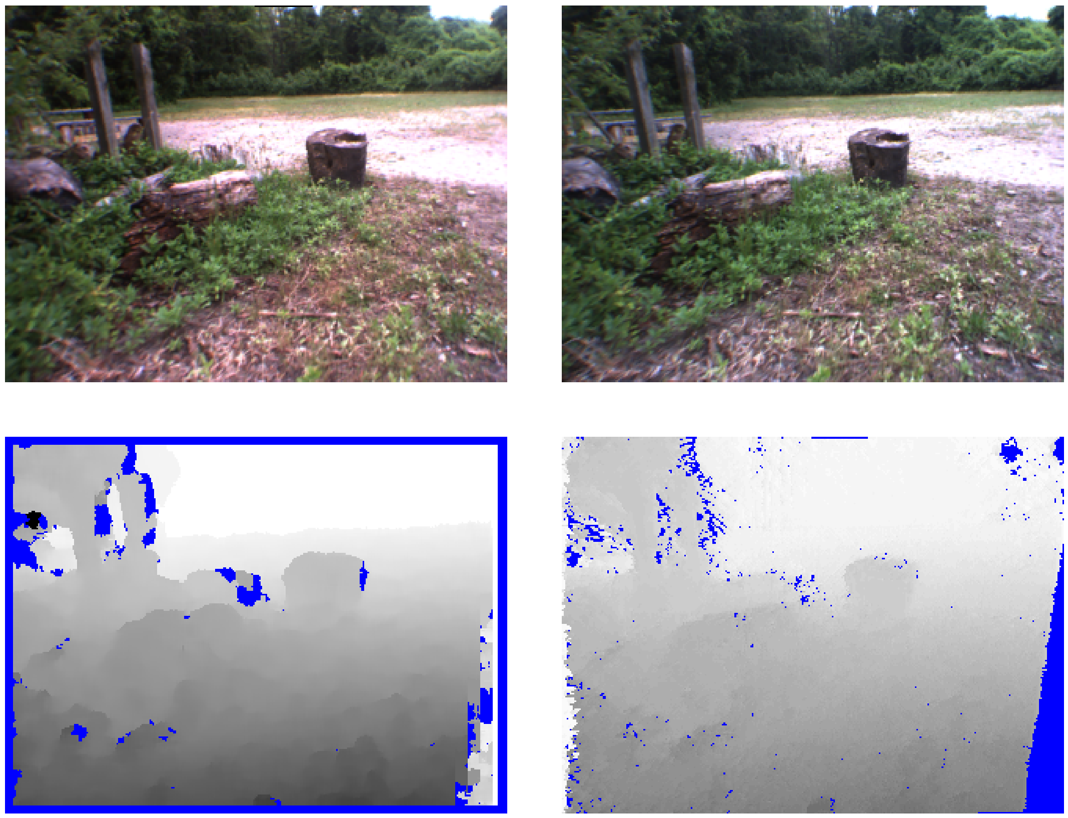


Figure 1.7: Second comparison of SGMGPU with the Triclops SDK's algorithm on LAGR test data. The upper images are the left and right stereo input images. The bottom left shows the output of the Triclops SAD algorithm. The bottom right shows the output of our SGM algorithm. Note that our algorithm is much better at resolving the shape of the tree stump, and produces a continuous depth gradient along the path, while the SAD algorithm exaggerates the size of the tree stump and outputs depth in splotchy discontinuous steps.

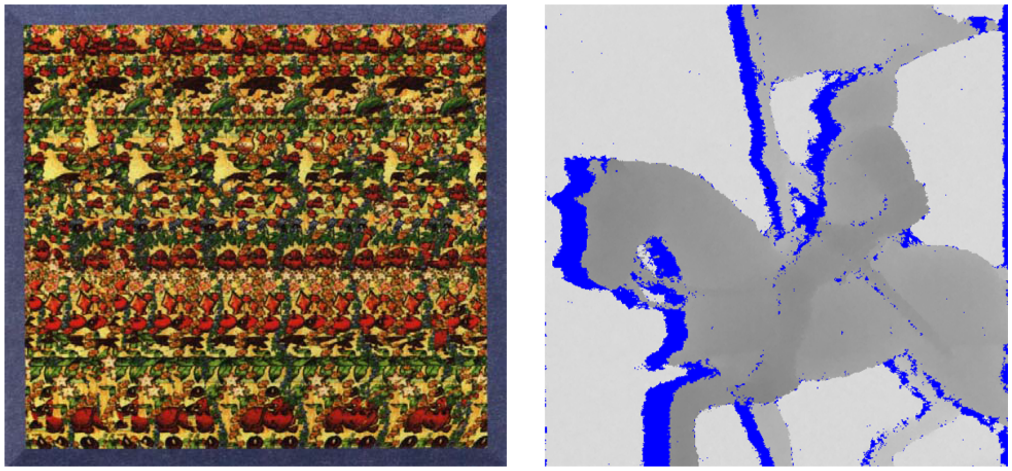


Figure 1.8: Extraction of a disparity map from a stereogram. No special modifications to the software were necessary for it to see a depth image within the stereogram. To create the input, we simply cropped two images from the original stereogram. For the left eye image, we removed one vertical band of the pattern from the right side of the source image, and for the right eye image, we removed a band of the same size from the left side of the source image.

Another application I explored was extracting depth out of stereograms, which are images with encoded stereo depth that can only be seen by crossing or uncrossing your eyes. Surprisingly, our algorithm was able to extract the hidden depth information without any modifications. All we needed to do was to crop the original image into a left eye and right eye version, and feed those two images into our algorithm. The result was a greyscale depth image showing the hidden picture (Figure 1.8).

1.5 Follow-up Work

In early 2008, we ported our algorithm to use CUDA which allowed us to greatly simplify the algorithm implementation and increase the number of directions used in the SGM algorithm to 16.

We did this by running through the directions one at a time, and using a separate CUDA thread to process each slice in a given direction (Figure 1.9). To maximize parallelism, all the threads for each direction were started simultaneously. For threads along the diagonal directions, to maintain memory locality and insure a consistent amount of work per thread, we had the threads reset their state and wrap around whenever they stepped past the edge of the error volume.

Although this new approach reduced the amount of parallelism by a factor of four compared to the shader-based algorithm, it made the memory access patterns much more consistent, which led to significantly faster performance. This increased the quality of our results, allowing us to match the quality of the CPU based version of SGM as described in the original SGM paper [20].

The resulting CUDA-based algorithm nearly doubled the frame rate of our shader-based algorithm to 14.8fps with 320x240x64 resolution and 16 directions, running on the same nVidia 7900GTX used in previous experiments. Although the results of this work were never published, we believe that this later implementation outperformed other GPU-based implementations of the SGM algorithm that existed at the time [14].

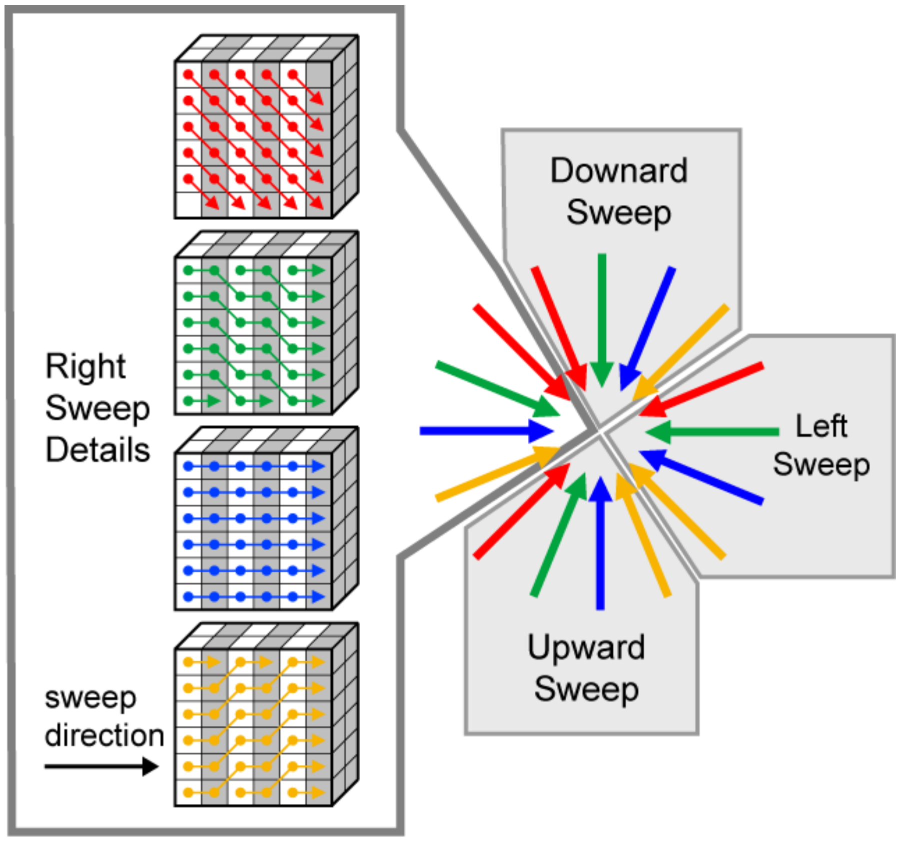


Figure 1.9: The 16 passes used in the CUDA implementation of the SGM algorithm. For the 4 out of 16 passes shown in detail, each set of connected dots and arrows represents the work done by a single thread. For the diagonal passes, threads loop around whenever they reach the top or bottom of the error volume.

2 RTPIE - Real-Time Particle Isosurface Extraction

2.1 Introduction

Particle-based methods are commonly used for simulation of fluid, gelatinous, and gooey substances. These methods are often used in interactive applications such as surgical simulation, surface modeling, and video games. While modern computers are easily capable of simulating thousands of particles in real time, in many cases, a surface must be generated over the particles in order to realistically render the output of such a simulation. This surface extraction step is often the bottleneck in such applications due to the high computational cost and/or large memory requirements of common surface extraction algorithms.

We present a new approach for fast, high quality polygonization of isosurfaces that can be used to render surfaces in real-time over thousands of particles in an unbounded spatial domain using a small amount of working memory, and compare it to existing algorithms. Furthermore, we extend our approach to generate polygon faces in back-to-front rendering order for transparent surfaces. Finally, we demonstrate the effectiveness of this new technique with several interactive scenarios showing complex interaction between fluid entities and dynamic objects in a virtual environment.

2 RTPIE - Real-Time Particle Isosurface Extraction

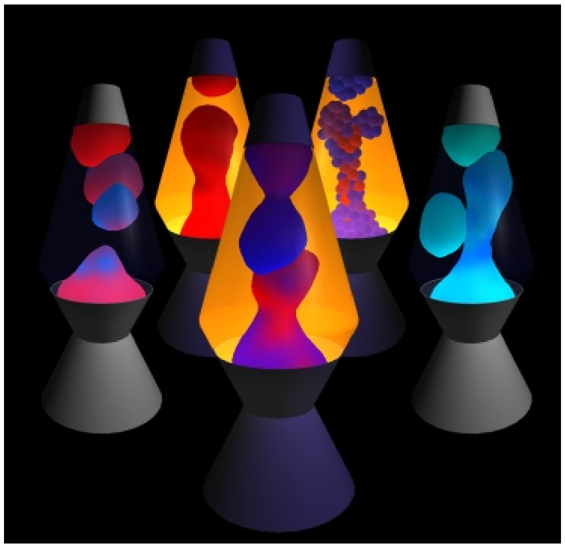


Figure 2.1: Several different lava lamps rendered in realtime with our algorithm.

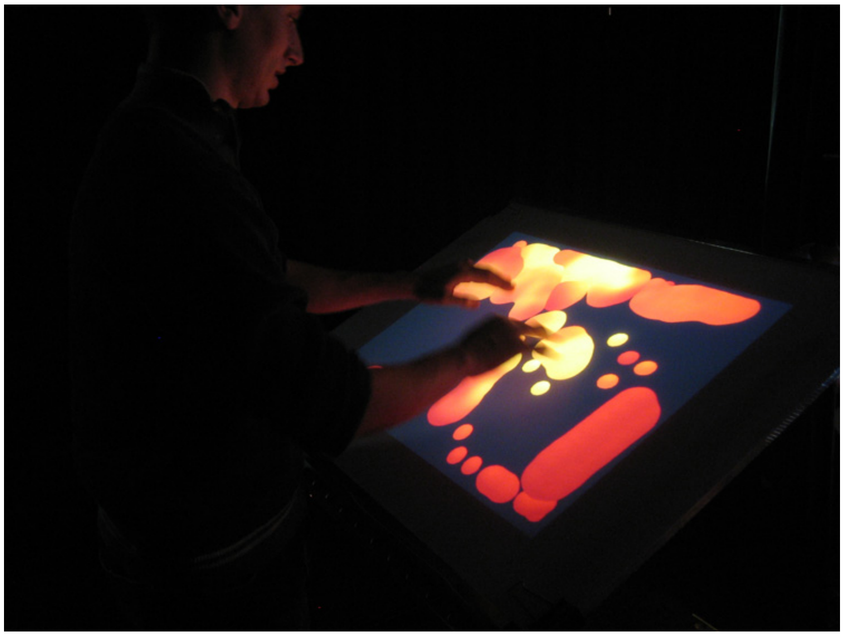


Figure 2.2: Interactive lava lamp simulation running on an FTIR table.

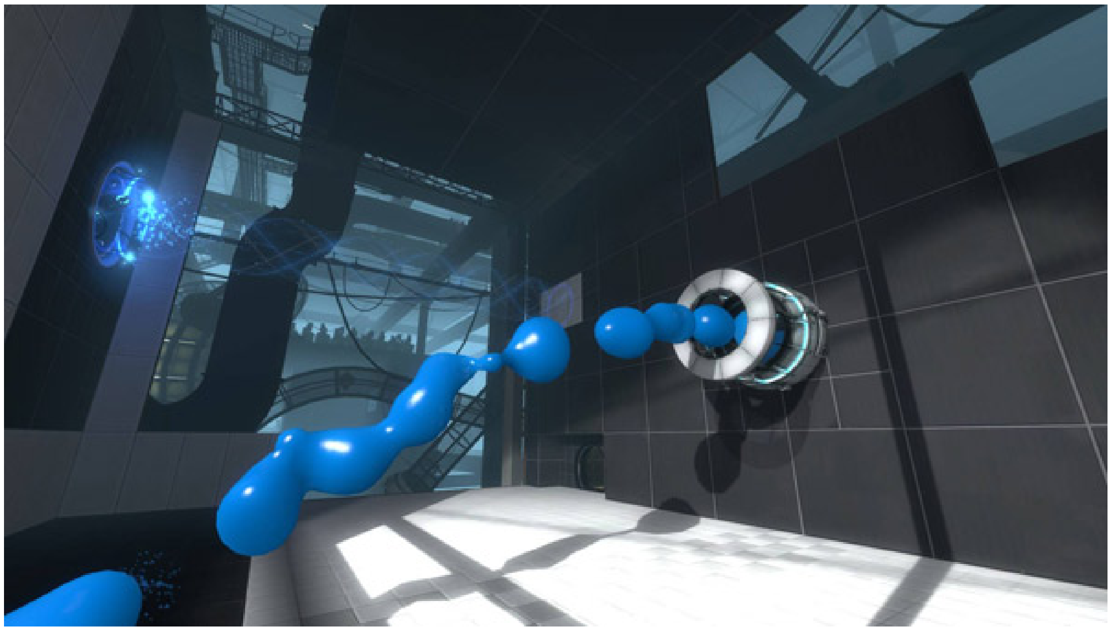


Figure 2.3: Blue gel in the Portal 2 game. Courtesy of Valve Corporation.

The development of this algorithm began in 2001 with the creation of a virtual lava lamp simulation for a graphics class at Princeton University. This was later released as a screen saver called LavaLamp3D (Figure 2.1). The algorithm was later used to create an interactive lava lamp for use with multi-touch displays, including Jeff Han's Perceptive Pixel display (Figure 2.2).

The lava table demo was seen by Valve Corporation's Ken Birdwell, who invited me two work with Valve on incorporating the algorithm into Valve's source engine. At Valve, I perfected the algorithm for use in real-time applications such as video games, solving many of the challenges created by going from a fixed volume simulation to fluids that could exist in a large virtual world and in incorporating the algorithm with the other components of a complex game engine.

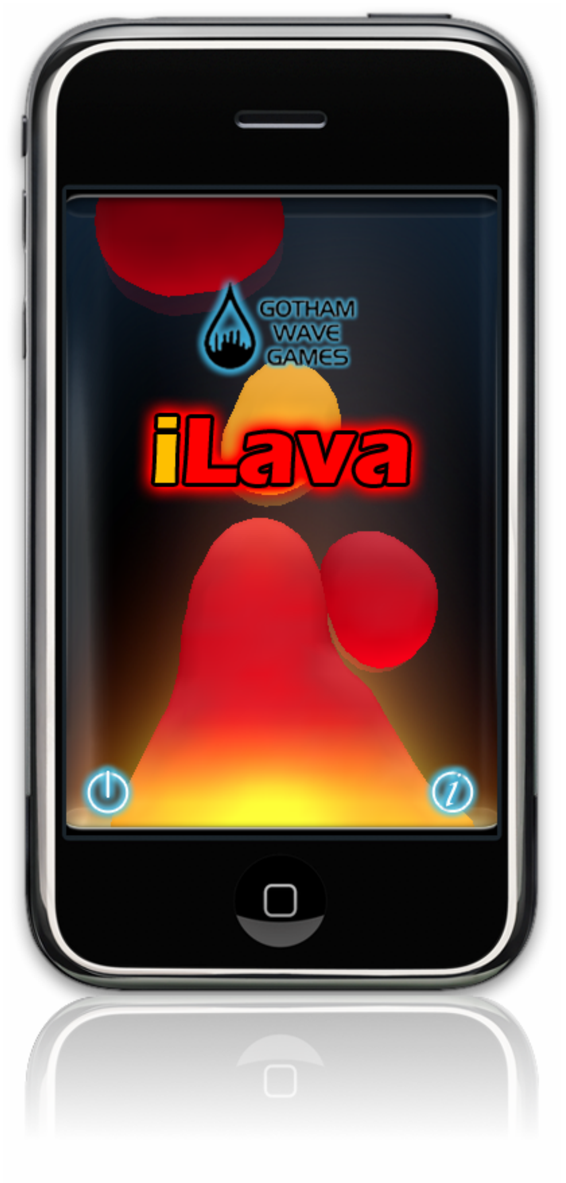


Figure 2.4: Screenshot of the iLava virtual lava lamp iPhone app.

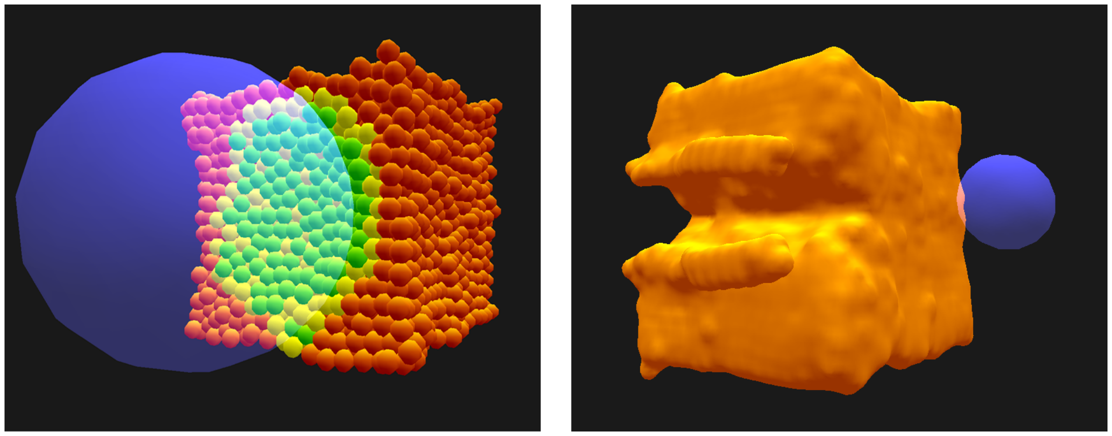


Figure 2.5: Virtual clay sculpting software. The underlying particle simulation is shown on the left. Skin is rendered over the sculpted particles using our method on the right.

As part of this work, I developed a complete, well documented and flexible isosurface rendering system called Blobulator and worked with a small team of talented artists and engineers to experiment with several different gameplay concepts.

With Valve's support, the technology was demoed at I3D in 2008 [36]. In April 2011, Valve released Portal 2, the first game to use the Blobulator engine. The engine was used to render "gels" which interacted with a complex virtual world to change it's behaviors, and could travel through portals to create incredibly complex and engaging puzzles (Figure 2.3).

The engine was also used to create iLava, a popular virtual lava lamp application for the iPhone and iPad (Figure 2.4) and a virtual clay sculpting application (Figure 2.5).

2.2 Related Work

Particles are a common and convenient representation for modeling physical entities, especially those that can undergo drastic changes in shape and topology. By adding inter-particle forces, a particle system can be made to act like a fluid, solid, and anything in between, or it can be procedurally animated to achieve a specific shape or motion. Furthermore, in interactive applications, particles can easily handle interaction with users and world geometry.

In order to create compelling simulations of particle-based phenomena, a large number of particles is usually necessary. To give the user the illusion that the simulated phenomena are composed of an inordinate number of particles, as they are in nature, it is often desirable to coat the particles with a smooth surface so that the individual particles are not visible. A common technique for generating such a surface is to define an implicit function over the particles in space, and to render the surface wherever that function is equal to a predetermined threshold value. Such a surface is commonly referred to as an implicit surface or isosurface.

Blinn [4] first proposed the use of implicit surfaces as a model for ray-tracing electron density maps over molecular structures and suggested the use of implicit surfaces as a general model for three-dimensional shapes. Reeves [34] proposed the use of particle systems as a technique for modeling and animating fuzzy objects and rendering particles simply by additively blending (splatting) them into a buffer. Sims [40] expanded upon Reeves' work by using particles for modeling other natural phenomena such as waterfalls, fire, and tornadoes.

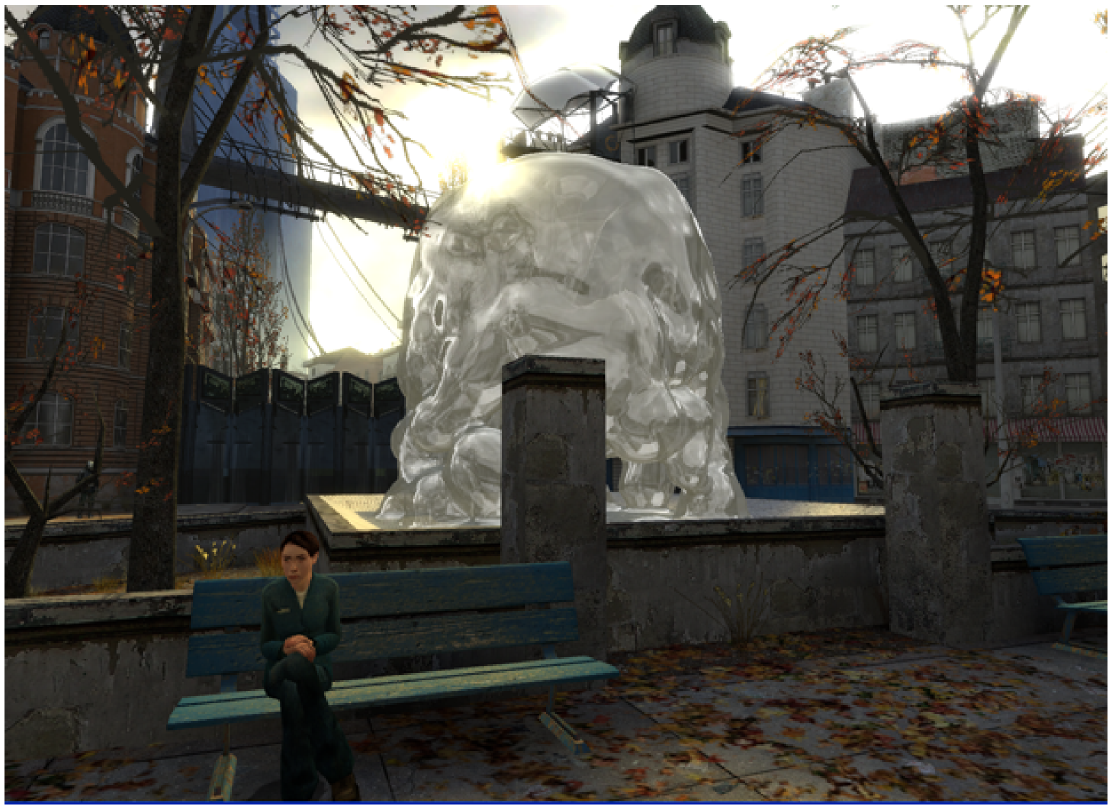


Figure 2.6: A fountain rendered in real-time using our technique in a fully interactive video game environment.

In more recent work, taking advantage of recent hardware, Adams *et al.* [1] proposed a technique where on the order of 100,000 particles are rendered as transparent sprites on the GPU. Because the number of particles is so large, the individual particles become less visible, creating the illusion of a smooth surface. However, the approach suffers from the same problem as earlier work since the illusion is broken at the edges of the surface where individual particles are still visible.

Szeliski and Tonnesen [43] and Witkin and Heckbert [50] proposed an alternate approach which uses a second class of oriented particles, known as surfels, to track implicit surfaces. More recently, Müller *et al.* [29] used a large number of surfels to track texture and surface detail over slowly deforming models. The drawback of these techniques is that for a small number of physically simulated particles (phyxels), several orders of magnitude more surfels must be simulated to cover the surface. These surfels must physically interact with both the phyxels and each other. Furthermore, the physical simulation requires frame coherence, otherwise surfels lose track of the surface. Finally, neither the ray-tracing, particle splatting nor surfel tracking approaches produce polygons as output. Thus, they can not fully take advantage of the pipeline used in the majority of interactive graphics applications where geometry is generated on the CPU, and the GPU is used for polygon based texturing, shading and rasterization.

For these reasons, a technique that can efficiently polygonize the isosurface, or in other words, generate a triangle mesh is preferable. Although there are techniques such as marching tetrahedra proposed by Bloomenthal [5] and marching triangles proposed by Hilton and Illingworth [19] which can polygonize surfaces, marching cubes proposed by Lorensen and Cline [24] is by far the most commonly used approach because is straightforward to implement, does not require coherence between frames, and is arguably the fastest way to extract surfaces from a 3D volume of scalar data.

In the marching cubes approach, a volume of space is divided into individual cubes. Isosurface field values and normals are calculated at the corners of the cubes, and an eight bit lookup is computed based on whether the field values are larger or smaller than the isosurface threshold. The lookup is used to index a static data structure which indicates how vertices on the edges of the cube are to be connected to form a set of triangles. Finally, the vertices are computed by linearly interpolating the positions and normals at the corners based on the field values.

The main drawback of marching cubes is that it requires a regular 3D volume of scalar field values as input. This is usually not a problem in applications such as medical imaging, where the data is naturally captured as a 3D volume. However, in interactive applications where the input is a set of procedurally generated particles, there is no volume of scalar data a-priori, and it is preferable to sample the field values defined by the particles judiciously by traversing, calculating isosurface values for, and polygonizing only the the cubes that contain segments of the isosurface.

To address this problem, Wyvill *et al.* [52] proposed a surface-following (continuation) approach which, starting at a point which contains a surface, continued by traversing neighboring cells which contain portions of the surface in a depth-first or breadth-first manner. Additionally, the paper noted that it is preferable to force particle fields to have a limited radius of influence, and suggested using a voxel grid to look up particles that have influence at particular locations in the volume. It mentioned that this lookup structure becomes more accurate as the voxel size shrinks to the size of the marching cube grid, but that the memory footprint of the structure and the cost of lookup/insertion increases as the grid size becomes smaller. It also noted that because cube corners can be shared by as many as eight cubes, storing these values in a 3D cache grid and re-using these values is much more efficient than re-calculating them from scratch each time they are needed. However, it did not suggest any way to find and eliminate unneeded cached values during the course of rendering a single frame. Finally, it proposed using a hash function that wraps each coordinate to keep it within the the dimensions of the 3D cache to allow rendering in an arbitrarily large unbounded volume.

Triquet *et al.* [47] detailed several important considerations for extending these techniques for fast isosurface polygonization of particles. These included an improved isosurface field function, finding start points (seeds) from which to traverse the surface by evaluating field values along a fixed direction starting at the center of each particle, and reusing vertex calculations at edges that are shared between cubes in addition to reusing corner calculations. This paper also gave a cursory description of the problem of looking up particles that contribute to a given isosurface field value, and came to the same conclusion as [52], that the voxel size for the lookup data structure must be coarser than the marching cubes grid, yet failed to provide an analysis of the optimal grid size. Furthermore, the paper noted that using a hash map for making isosurface extraction unbounded slows the algorithm, yet failed to provide an alternative.

Teschner *et al.* [45] proposed an alternate hash function along with analysis of its performance, and additionally provided an analysis of the cost of inserting/looking up elements in a volumetric grid and how it varies with grid size. Although they used tetrahedra as a primitive rather than spheres, their findings showed that the optimal performance of a grid based lookup occurs when the grid is approximately the same size as a tetrahedron's edge length.

In our experiments with the same sort of lookup, we found a similar relationship for particles, where the optimal performance occurred when the size of grid cells was approximately equal the radii of the field of influence of the particles. Unfortunately, at this grid size, this type of lookup cache is very imprecise, yielding approximately 6.5 times more particles than the number that actually contribute to the field value at a given point (See Section 2.7 for more details).

In this chapter, we present a novel real-time particle isosurface extraction technique that overcomes the deficiencies of previous approaches. Our technique consists of the following major contributions:

1. A spatial decomposition algorithm which divides the volume to be rendered into blocks while avoiding seams or inconsistencies in the surface between adjacent blocks. This naturally limits the upper bound of memory usage, eliminates the need for hashing, allows rendering of particles in an unbounded volume, and enables multi-threaded rendering on multi-core computers.

2. An algorithm for extracting the isosurface within a block which we refer to as

Marching Slices that avoids excessive growth of cached data by polygonizing the isosurface in a slice by slice fashion. Our algorithm guarantees a single visit to each cube intersecting the isosurface without keeping global information on all visited cubes by discarding slices of cached data that will not be reused in the future. Moreover, because our algorithm renders in slices, it improves the locality of memory accesses, and can be easily extended to output triangles in a back-to-front order when rendering transparent surfaces.

3. A fast and exact particle lookup technique which speeds up isosurface field value calculations by finding all the particles within a fixed influence radius that contribute to a sample point without finding any particles outside of that radius. In contrast, alternative lookup techniques return many particles that are outside of the influence radius, thereby wasting time in field value calculations.

Although blocking in order to subdivide large problems into smaller ones, processing of volumes one slice at a time, and lookup of particle influences by projecting them onto a 2D surface has been explored before in other contexts, we believe we are the first to combine these techniques into a unified approach which solves many, if not all of the practical real-world problems that one may encounter when using isosurface-extraction of particle data sets in interactive applications.

The rest of this chapter details how these three components work in concert to enable memory efficient, spatially unbounded real-time rendering. Furthermore, we present a detailed comparison between this new technique and other particle-based isosurface extraction approaches over multiple scenarios, and further demonstrate its performance in a complex interactive game environment. Due to space limitations, we do not address issues relating to particle simulation or animation other than mentioning that we use a separate set of data structures (which operate at a much coarser level than the data structures used for rendering) for inter-particle collision detection, and a commercially available physics engine for collisions between particles and the world.

2.3 Algorithm Overview

The input to our algorithm is a list of particles containing (x, y, z) coordinates for position and the output is a list of triangles represented by a vertex and index buffer. The algorithm requires that the implicit field produced by each particle is radially monotonic, continuous and has a limited radius of influence which we call the cutoff radius R_c. We employ the following function because it can be computed efficiently with a few simple operations (this function is similar to the one used by Triquet [47], but we prefer this form because it requires less operations to compute):

$$f(r) = \left\{ \begin{array}{ll} (1 - (r/R_c)^2)^2 & \text{if } r < R_c \\ 0 & otherwise \end{array} \right\}$$

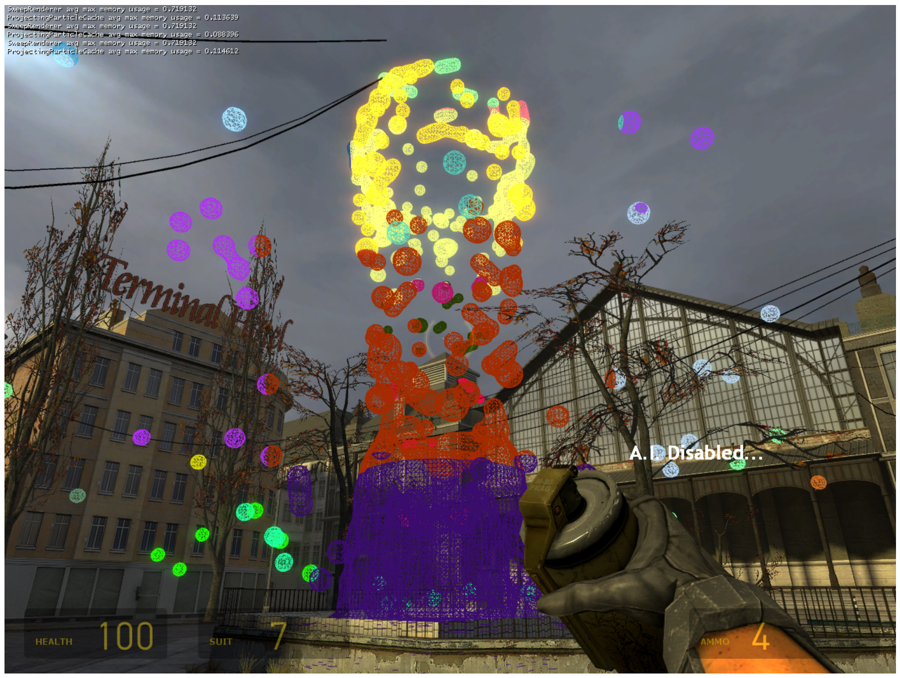
.

In addition to R_c and a list of particles, our algorithm takes the size of a cube S, the threshold value where surfaces are generated T, and possibly other user-defined data. Without user-defined data, a field calculation routine calculates field values and normals which are then interpolated by a vertex calculation routine to output vertex positions and normals. The user-defined data can be used along with custom user-defined isosurface field and vertex calculation routines to generate vertices with additional data such as colors, or texture coordinates.

Once all the input data is specified, the algorithm proceeds by subdividing the render volume into a list of blocks. Proceeding one block at a time, it builds a particle lookup cache for accelerating field value calculations, and then finds the seed cubes at which polygonization will begin. It then proceeds to use the marching slices algorithm to polygonize the cubes inside each block slice by slice, using field and vertex calculation routines to generate vertices and writing the output into a user-specified vertex and index buffer.

We would like to point out that at all stages of the algorithm, we take great care to avoid duplicate computation by using caches to recycle computation results. We also focus on keeping memory usage low and reasonably bounded as this is very important for real time applications. Because of the memory efficiency of our algorithm, a majority of our data structures can fit into L2 or L3 cache on most modern computers, which is often several orders of magnitude faster than main memory. Finally, in a real-time system, such as a video game, our rendering algorithm needs to coexist with other game components on a wide range of platforms, including consoles, which often have limited memory. For this reason, our algorithm is ideally suited to real-time particle rendering in real-time systems.

2.4 Block Subdivision



```
SweepRenderer avg max memory usage = 0.719132
ProjectingParticleCache avg max memory usage = 0.113639
SweepRenderer avg max memory usage = 0.719132
ProjectingParticleCache avg max memory usage = 0.088396
SweepRenderer avg max memory usage = 0.719132
ProjectingParticleCache avg max memory usage = 0.114612
```

Figure 2.7: This simulated fountain is too big to render in a single marching cubes volume, so it is seamlessly split into multiple blocks, which are shown here in different colors.

We divide the volume containing particles into disjoint blocks, which can be processed independently, calculating the surface sequentially using only the memory footprint for a single block, while producing a seamless final result (Figure 2.7). Starting from the global set of particles, we build a list of blocks, each of which keeps its own list of particles. Each block is only responsible for rendering the cubes inside it, but to maintain consistency with adjacent blocks, it must consider all particles beyond its boundary that may have a field contribution in the interior

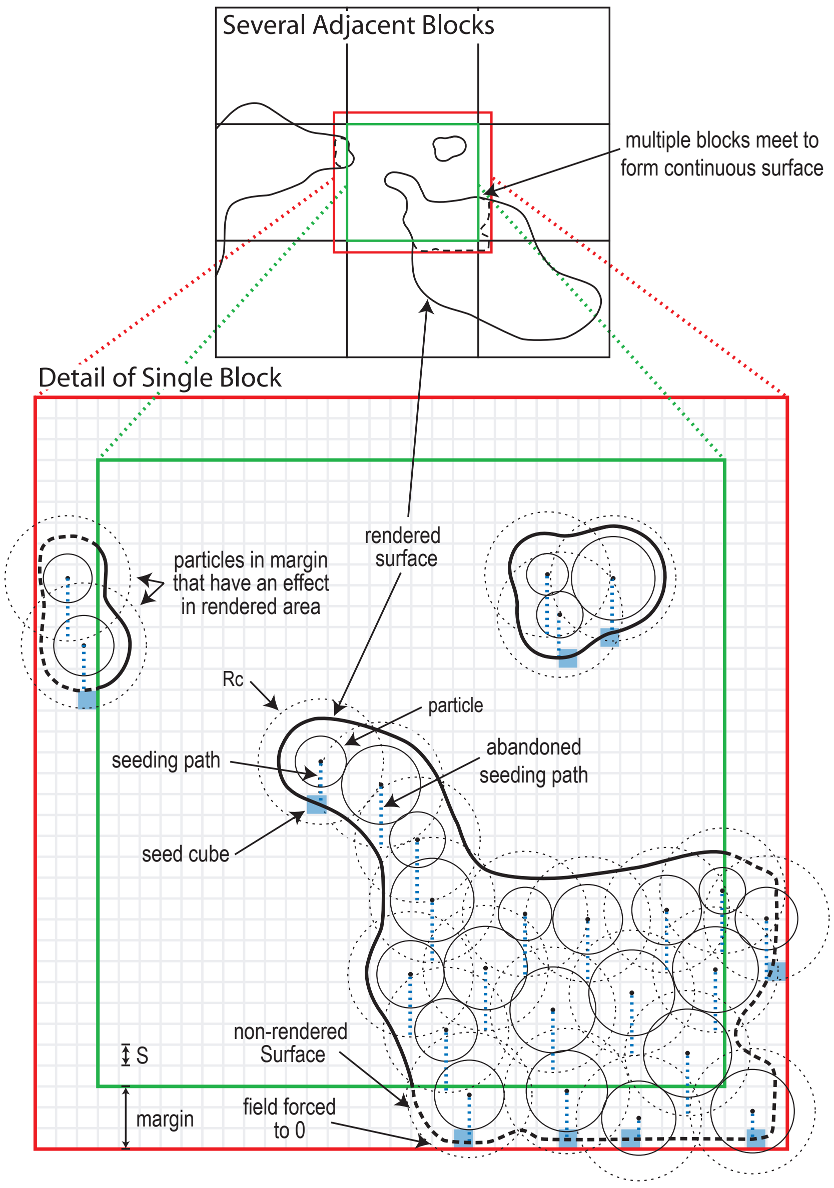


Figure 2.8: 2D illustration of block subdivision. The green box contains the polygonized surface and red box outlines the margin.

of the block. Thus, we additionally insert all particles within a distance of R_c from the block into the list (Figure 2.8).

We expand the original dimensions of the block by a margin of cubes to enclose a distance of R_c, the radius of influence of a particle. Marching slices will operate over the expanded volume, but will not output mesh triangles for cubes in the margin. This guarantees extraction of the isosurface of all particles including those located in the margin. Because the marching slices algorithm (Section 2.5) relies on the assumption that the isosurface is closed in order to guarantee a traversal of the entire surface, we force field values to disappear at sample points on the outer margin boundary by instrumenting our lookup cache in such a way that it does not return particles for samples on the boundary (Section 2.7).

The overhead of our block subdivision approach is the cost of traversing the cubes that are in the margin. For a typical $100 \times 100 \times 100$ cube block, with a margin that is 5 cubes wide, 25% of the cubes in the full $110 \times 110 \times 110$ volume are margin cubes. Since no vertices are generated for margin cubes, and full field calculations are not necessary (only the sign of the field is needed), the cost of traversing a cube in the margin is approximately half that of a normal cube. Thus, in the case that particles are randomly distributed within the volume, the extra overhead of blocking is approximately 12% of total work. This overhead is small compared to the cost of hashing, the alternate approach for rendering unbounded volumes, as hashing must be done both for corner retrieval and for particle lookups (Section 2.7). Furthermore, block subdivision decreases memory usage by limiting the number of corner values and particles kept in cache.

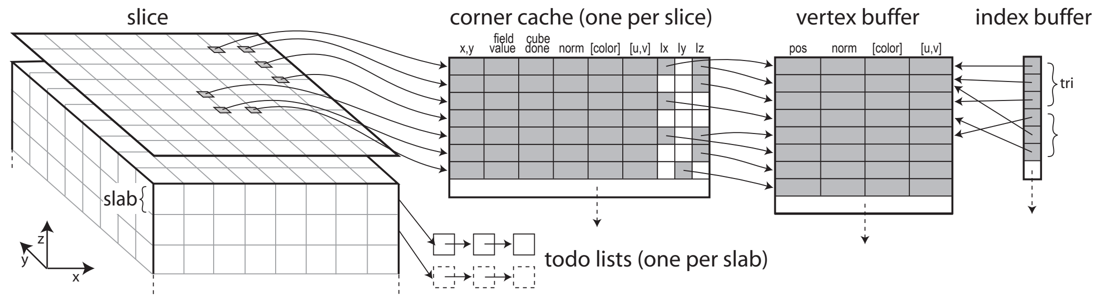


Figure 2.9: Data structures used in the Marching Slices algorithm.

Block subdivision is also beneficial for applications such as surface modeling, where only a small subset of particles move between frames. Here, the blocks with no moving particles can be redrawn without polygonization simply by re-rendering old vertex and index buffers. By skipping the rendering of blocks that are not visible from the camera, block subdivision can be used for visibility culling, and can also be used for controlling level of detail by varying S between blocks, although care needs to be taken to avoid visible seams. Finally because each block is rendered independently of other blocks, it is straightforward to parallelize the algorithm by rendering each block in a separate thread on a multi-core processor.

2.5 Marching Slices

The intuition for the marching slices algorithm is that memory usage would be greatly reduced if it were possible to polygonize isosurfaces one slice at a time, caching only the data necessary for the current slice. However, this poses two serious challenges. The first is that the field values are not known a-priori since we only calculate field values on demand. Thus, it is not possible to simply march back-to-front through the volume. Instead, it is necessary to march along connected cubes containing the isosurface starting at a seed cube in a fashion where all the connected cubes in the current slice are traversed before moving on to the next slice, with the ability to march through the volume arbitrarily in both forward and reverse order depending on how the cubes are connected. The second challenge is that in order to realize a reduction in memory usage, we need to discard cached data that will not be reused and ensure that there are no redundant visits to completed cubes.

2.5.1 Data Structures

To keep memory usage to a minimum, we use a sparse set of data structures over the volume (Figure 2.9). We treat each block as a vertical array of N_z *slabs* of size $N_x \times N_y \times 1$ cubes. We refer to the vertices of a cube as *corners*, reserving the term *vertex* for a vertex that is output by the algorithm for rendering. The slabs are numbered from 0 to N_z where z increases upwards. Each slab is bounded above and below by a two-dimensional slice, with adjacent slabs sharing slices. The slices are numbered from 0 to $N_z + 1$. For each of the N_z slabs, we store a linked list, called the *todo list* containing integer (x, y) tuples which represent cubes in the slab that may need to be rendered.

For any slab which is being polygonized, we allocate two *slice caches* for the slice above and the slice below the slab to store data that may be reused by an adjacent slab. A slice cache consists of a 2D *corner index array* of size $N_x + 1 \times N_y + 1$, and a dynamically resizing vector of corner values called a *corner cache*. Each element of the corner index array consists of an index which can be used to retrieve a computed corner from the corner cache (or 0 if the corner has not been computed), and two boolean "done" flags to indicate whether the cubes above and below the slice at that location have been rendered.

For corners that have already been visited, the corner cache array stores an (x, y) tuple corresponding to the location of the corner (which is later used for clearing the corner index array), the calculated floating point field value and normal, an optional user-defined structure for data such as colors and texture coordinates and three vertex indices (I_x, I_y, I_z). Each non-zero vertex index refers to an entry in a vertex buffer for the three possible vertices located between the corner at location (x, y, z) and the three other neighboring corners at $(x+1, y, z)$, $(x, y+1, z)$ and $(x, y, z+1)$. Each entry in the vertex buffer stores an (x, y, z) position, normal and optional information such as color, and texture coordinates. Finally, we keep an index buffer which references vertices in the vertex buffer and defines the set of triangles to be drawn in the generated mesh.

2.5.2 Seeding

Because the marching slices algorithm only traverses cubes that contain the isosurface, one or more starting points which we refer to as *seed cubes*, or simply *seeds*, must be found for each disconnected surface. To ensure that the isosurface surrounding each particle is found, seed cubes are generated for all particles in a block. This is done by snapping to the corner closest to the particle position, evaluating the field value, and then stepping down in the negative z direction one corner at a time until a corner with a field value below the threshold is found. An entry is then pushed into the todo list of the slab that contains the transition in field values with the (x, y) position of the cube that the seed particle was in.

The steps are taken in the negative z direction because we prefer to render slices from bottom to top and would like to find seed points as close to the bottom as possible. If seeding traverses farther than R_c down from a particle center without finding an isosurface, the search is terminated, as such a particle must have another particle below it which will find the surrounding surface. If the seeding algorithm steps all the way down to slice 0, a seed cube will be found because field values at the boundary of a block are forced to 0 so that all surfaces are closed, as discussed in Section 2.4. It is important to note that the grid resolution must be high enough such that the corner closest to each particle is within that particle's radius. Otherwise, this procedure may fail to find the isosurface around a particle.

2.5.3 Execution

Once seeding completes, the marching slices algorithm begins to polygonize the isosurface (Algorithm 1). The algorithm repeatedly polygonizes the bottom-most slab containing an entry in its todo list, until all todo lists are empty.

Before entering a slab, the algorithm ensures that the slice caches and corner caches have been allocated for the slices above and below the slab. It then proceeds to polygonize the slab using a variant of surface-following marching cubes that visits only the connected cubes in the current slab.

To polygonize a slab, the algorithm repeatedly pops a cube from the slab's

Algorithm 1 The Marching Slices Algorithm

```
$slabs[].todo\_list \leftarrow findAllSeedCubes()$
**while** one or more slabs has a non-empty todo list **do**
    $z \leftarrow getZOfLowestSlabWithTodoItems()$
    $slab_{current} \leftarrow slabs[z]$
    $slab_{below} \leftarrow slabs[z-1]$
    $slab_{above} \leftarrow slabs[z+1]$
    $slice_{below} \leftarrow slices[z]$
    $slice_{above} \leftarrow slices[z+1]$
    $ensureAllocated(slice_{below})$
    $ensureAllocated(slice_{above})$
    **while** $!slab_{current}.todo\_list.empty()$ **do**
        $(x, y) \leftarrow slab_{current}.todo\_list.pop()$
        **if** $slice_{below}[x, y].done_{above}$ *or* $slice_{above}[x, y].done_{below}$ **then**
            **continue**
        $slice_{below}[x, y].done_{above} \leftarrow$ **true**
        $slice_{above}[x, y].done_{below} \leftarrow$ **true**
        $corner_0 \leftarrow lookupOrEval(slice_{below}[x, y])$
        $corner_1 \leftarrow lookupOrEval(slice_{below}[x+1, y])$
        $corner_2 \leftarrow lookupOrEval(slice_{below}[x, y+1])$
        $corner_3 \leftarrow lookupOrEval(slice_{below}[x+1, y+1])$
        $corner_4 \leftarrow lookupOrEval(slice_{above}[x, y])$
        $corner_5 \leftarrow lookupOrEval(slice_{above}[x+1, y])$
        $corner_6 \leftarrow lookupOrEval(slice_{above}[x, y+1])$
        $corner_7 \leftarrow lookupOrEval(slice_{above}[x+1, y+1])$
        **if** cube is not in margin **then**
            $polygonize(corner_{0-7})$
        $(do_{above}, do_{below}, do_{left}, do_{right}, do_{front}, do_{back}) \leftarrow cubesToDo(corner_{0-7})$
        **if** $do_{above}$ *and* $!slice_{above}[x, y].done_{above}$ **then**
            $slab_{above}.todo\_list.push(x, y)$
        **if** $do_{below}$ *and* $!slice_{below}[x, y].done_{below}$ **then**
            $slab_{below}.todo\_list.push(x, y)$
        **if** $do_{left}$ *and* $!slice_{below}[x-1, y].done_{above}$ **then**
            $slab_{current}.todo\_list.push(x-1, y)$
        **if** $do_{right}$ *and* $!slice_{below}[x+1, y].done_{above}$ **then**
            $slab_{current}.todo\_list.push(x+1, y)$
        **if** $do_{front}$ *and* $!slice_{below}[x, y-1].done_{above}$ **then**
            $slab_{current}.todo\_list.push(x, y-1)$
        **if** $do_{back}$ *and* $!slice_{below}[x, y+1].done_{above}$ **then**
            $slab_{current}.todo\_list.push(x, y+1)$
    **end while**
    **if** $slab_{below}.todo\_list.empty()$ **then**
        $deallocate(slice_{below})$
    **if** $slab_{above}.todo\_list.empty()$ **then**
        $deallocate(slice_{above})$
**end while**
```

todo list and makes sure it hasn't been polygonized by checking the done fields in the slice above and below. It then marks the cube as visited and polygonizes the cube. If polygonization indicates that unvisited neighboring cubes contain the isosurface, they are added to the todo list of their respective slabs. This procedure repeats until the todo list of the current slab is emptied.

Once the todo list in a slab is emptied, a check is performed to see if the cached data for the top or bottom slice can be cleared. The top slice's cache can be cleared when the todo list in the slab above is empty, and similarly, the bottom slice's cache can be cleared when the todo list in the slab below is empty. All cleared slices' corner index arrays are reset by clearing nonzero indices and done flags using the (x, y) entries in their respective corner caches and are returned to a common memory pool.

The algorithm completes once there are no more todo list items for any slab in the block, at which point the index and vertex buffers that have been generated for the current block can be flushed out for rendering to the GPU.

2.5.4 Algorithm Correctness

To prove that each cube is visited only once, we need only show that the slice caches (storing the cube-done flags) above and below a slab are only deallocated once the visited cubes in a slab can't possibly be re-visited. A slice cache is allocated when corner values are needed, that is, when a slab above or below is traversed. The done flags in the slice cache prevent revisitation of processed cubes in the current slab. They also prevent the addition of cubes that have already been processed to the todo list of an adjacent slab. A slice is deallocated only when the slabs above and below have empty todo lists, at which point, all the connected cubes in both slabs must have been traversed. When this condition is met, there is no danger in deallocating a slice because it no longer contains cube-done values and cached data that can be reused by unvisited cubes. This ensures that each cube can only be visited once, and that the algorithm will always complete, for any possible input.

2.5.5 Memory Requirements and Performance

We can estimate how many slices need to be kept in memory at the same time for the purposes of predicting overall memory usage. We know that if the seeding algorithm were able to find all the local minimum points in the isosurface and seed there, we could traverse the slabs monotonically from bottom to top using only two cache slices to render the entire block.

However, there are rare cases where the seeding algorithm does not find all the local minima (Figure 2.10). As a result, marching slices sometimes needs to step downwards to render connected cubes that were missed in slabs below. Each time the step direction reverses, an additional slice is needed to store the abandoned frontier until the algorithm returns to that slice. Fortunately, we have found that the typical number of slices needed to render a block is three, and situations where more than three slices are necessary are rare and short-lived.

Compared to the equivalent surface-following marching cubes algorithm which would have kept all $N_z + 1$ slices and associated corner values in memory, our approach uses much less working memory. It is also faster than surface-following

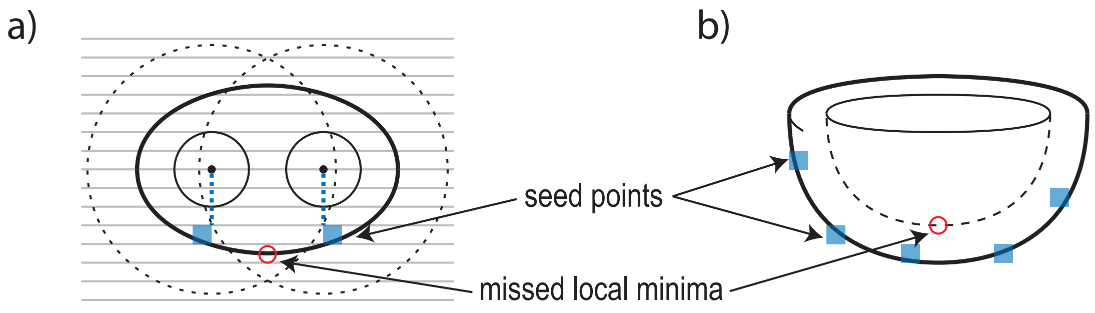


Figure 2.10: Cases where seeding fails to find local minima a) at the bottom of a surface, seeding is sometimes off by a single slab b) inside of an upright bowl shape there may be no seeds.

due to an increase in the locality of memory accesses. In fact, the amount of cache memory used by our approach with a $110 \times 110 \times 110$ grid is below 1MB (See Results). Additionally, while working on a slice, memory accesses become 2D rather than 3D, and various positional calculations can be reused, further improving performance. Our approach also runs much faster than hash map based approaches by avoiding the cost of hash map lookups and uses significantly less memory because the hash-based approaches keeps all corner values in memory, while ours keeps corner values only for the currently active slices (See Results).

Because of its small memory footprint, marching slices is able to coexist well with other components of an application such as AI, physics, networking and world rendering in a graphics engine. This is critical in interactive applications where virtual memory paging will degrade performance and frustrate users. Finally, this is important for multi-core hardware which shares cache memory between multiple processes, and hardware with relatively small amounts of working memory, such as mobile devices and game consoles.

2.6 Transparency and Culling

In some situations, it is advantageous to generate isosurfaces in a front-to-back or back-to-front order. For example, in order to properly render transparent surfaces on most graphics hardware, it is necessary to draw triangles in a back-to-front order. When drawing opaque surfaces, it is usually faster to draw in a front-to-back order to take advantage of per-fragment early-z-culling available on most graphics hardware. Because our algorithm generates surfaces in a slice-by-slice fashion, either can be achieved nearly for free with a few simple modifications.

The first step is to rotate the rendering volume perpendicularly to the viewer so that the sweep happens in the desired direction, and to rotate the particles in the opposite direction so they remain in the same place. Secondly, the block subdivision routine must be modified to render blocks in the desired order according to distance from the viewer. Within a block, because of the structure of the marching slices algorithm, slabs are polygonized in an approximately unidirectional order. The order is only broken when the algorithm steps back to do work on a slab "below" the current slab, which happens when seeding fails to find the bottommost portion of surface enclosing a particle. Because this is rare and the polygons that are incorrectly sorted rarely occlude each other, in practice the algorithm as described above is good enough for use in interactive applications without any visible artifacts (as can be seen in our demo videos).

In cases when absolutely perfect back-to-front sorting is required, instead of a global index buffer, separate index buffers can be allocated per slab and flushed out in a back-to-front order for each rendered block. Within an individual cube, we can guarantee that polygons are output back-to-front by preprocessing the static lookup tables which are used by the polygonization routine. Here, we can also ensure that polygons are output with consistent winding to facilitate back-face culling. Because both of these optimizations modify static lookup tables, there is no run-time penalty for their use.

In cases where it is necessary to keep the rendering volume aligned with a fixed basis vector rather than rotating it with respect to the user, front-to-back or back-to-front rendering can still be achieved. In these cases, the algorithm must be modified to perform a sweep along the axis that is most closely aligned to the user. In this approach, if the viewer has a wide field of view, they may be able to look edgewise through a slice. In this case, the cubes within each slice must be cached and individually sorted by distance to viewer before they are rendered. We leave it as an exercise to the user of the algorithm to determine the right set of tradeoffs between perfect Z-ordering of triangles and performance.

2.7 Particle Lookup Cache

Field value calculations are in the inner loop of both the marching cubes and marching slices algorithm and are generally the place where these algorithms spend the majority of computing time. This makes them the major target for optimization, and is the reason why we cache and reuse these values at sample points.

Because the field of influence of the particles is limited to a fixed cutoff radius, it is possible to optimize the field calculations by iterating only over the particles within R_c from a given sample point using a spatial data structure which we refer to as a *particle lookup cache.* Observing that lookups are only performed at field corners, we relax the requirements for the particle cache by only requiring it to know which particles influence the field at a corner, and not at any arbitrary point in space. At the same time, in order to avoid wasting time in the field calculation routine, we require that the particle cache give an exact solution, without returning any particles that are outside of R_c.

A simple way to implement such a particle cache would be to build a linked list at each corner referencing all the particles that are within R_c of the corner. However, the insertions spanning a sphere with radius R_c, would be computationally expensive and consume a considerable amount of memory. We can greatly improve upon this approach by observing that the linked list traversal during field calculation can serve double duty by simultaneously performing a one dimensional search through 3-space. This means that instead of marking a spherical 3D region, we only need to mark a 2D region for each particle in the shape of a disk.

2.7.1 Data Structure and Algorithm

Our approach, which we call a *2D projection 1D lookup* operates as follows: We allocate a single 2D array of dimension $Nx + 1$ by $Ny + 1$ called the projection slice. Each element in the array is simply a pointer to a doubly linked list. A list pointed to by element (x, y) in the projection slice will contain one entry for each of the particles with influence on the corners in column (x, y). Each element in the list contains the integral values min_z and max_z specifying the range of slices

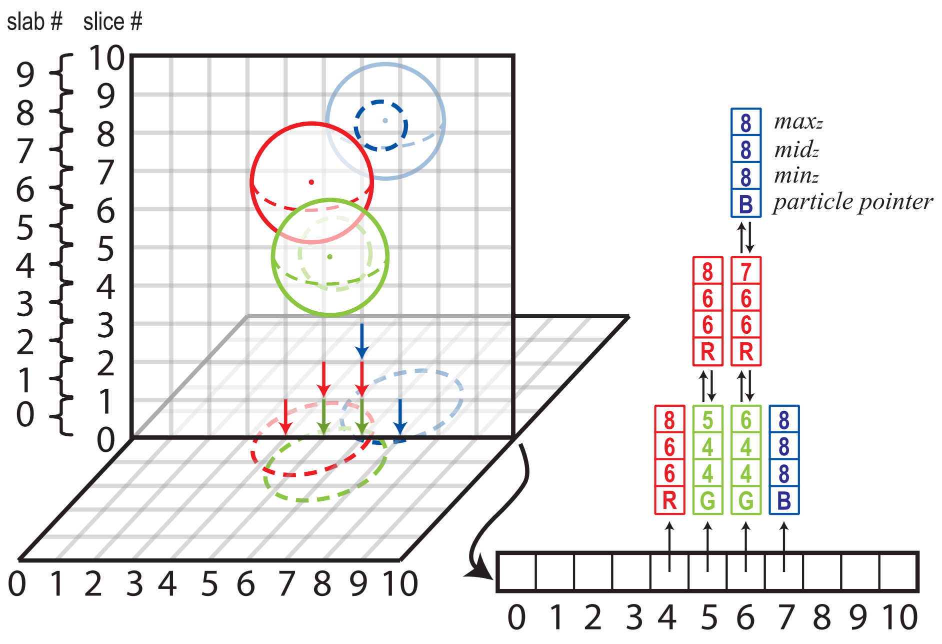


Figure 2.11: Cutaway diagram of insertion into particle cache.

in which the particle has influence, an integer mid_z specifying the slab containing the center of the particle, and a pointer to the particle (Figure 2.11).

The lists are built by quick-sorting all particles from smallest to largest z coordinate, and then, for each particle, inserting a list element into each (x, y) column where it has influence. In order to force the field values to 0 along the boundaries of a block as described in section 2.4, we avoid inserting elements into columns that are on a boundary, and clamp min_z and max_z to values between 1 and N_z. Thus, there is no additional cost for enforcing this constraint during field value calculation. For particles with a field of influence smaller than R_c, it is straightforward to modify the insertion step to shrink the radius in which the particle will be found and ensure an exact lookup.

Whenever a lookup is performed, the current corner location (x, y) is indexed in the projection slice to access a list containing all particles that have influence in that column. Next, to find the starting point, the list is searched for the bottommost entry with $mid_z \geq z - \lceil R_c/S \rceil$. The list is then traversed upwards to the topmost entry with $mid_z \leq z + \lceil R_c/S \rceil$. This traversal effectively finds every particle within a cylinder of radius R_c and height $2R_c$ around the lookup point.

To cull this list down to the sphere of radius R_c, quick integer comparisons are performed for every element to test that $min_z \leq z$ and $max_z \geq z$. If so, the element has influence at the lookup point, and is handed to the field calculation routine. Since we are likely to do lookups above and below recent lookups, to speed up finding of the starting point, the pointer at location (x, y) in the projection slice is updated with the first element found by each particle lookup in that column.

2.7.2 Memory Usage and Performance

Because the cylindrical volume covered by the list traversal is efficiently culled down to a spherical volume with two integral tests, the 1D search is only slightly less efficient than a simple linked list traversal, yet allows for a significant reduction in the time and memory cost of particle insertion. Furthermore, due to the nearly unidirectional traversal of the marching slices approach, the doubly linked list at a column is usually traversed only once while looking for starting points. Thus, the work done to find starting points is small and proportional to the number of element inserted into the particle cache.

To estimate the memory consumption of our approach, we note that approximately $\pi R_c^2/S^2$ elements are inserted for each particle. With a cutoff radius $R_c = 3$ and cube size $S = 0.8$, approximately 45 elements are inserted per particle. If each of these is 16 bytes in size, the cache consumes 720 bytes per particle, with a cache of 1,000 particles consuming a reasonable 720KB of data.

For situations where many more than 1,000 particles may be inserted into a block, we can reduce memory usage by keeping only the necessary z range of particles in cache. This can be done by removing particles that fall outside the useful z range or by keeping multiple projection slices, each one storing only the elements on a single slab.

Compared to the well known class of approaches used by Wyvill *et al.* [52, 51], Müller *et al.* [28, 30, 31] and Triquet [47], which use either a point insertion and a 3D lookup or a 3D insertion and point lookup, our approach is the only one that does an exact lookup, without returning any particles outside of R_c. This is because these approaches use a separate grid for particle lookups with cube size R_c (much coarser than the marching cubes/slices grid) in order to keep the number of cubes traversed during insertion or lookup down to a reasonable $3 \times 3 \times 3$ volume. Thus, these approaches return 6.5 times more particles than our approach by looking in a $(3R_c)^3$ volume, rather than a $(4/3)\pi R_c^3$ volume, which they must then cull by performing floating point distance calculations for each particle. With these approaches, if the lookup grid resolution is increased to reduce the number of false positives, the insertion/lookup costs quickly grow and dominate overall isosurface extraction costs, especially when each cube access requires a hash map lookup as suggested by previous approaches for unbounded rendering.

In our benchmarks, the memory usage of our approach was never significantly higher than with previous approaches because we build and destroy our particle cache on a per block basis, and can be further reduced by removing unneeded particles from the cache (as explained previously). In terms of time, although our approach must perform a 2D projection for each particle, it performed significantly faster than previous approaches in all of our benchmark tests because the number of lookups done per block is typically much greater than the number of insertions, and because our lookups are fast and don't return false positives.

2.8 Results

In order to accurately measure the performance of our technique and compare it to alternative approaches, we created simple particle simulations of a fountain, an amorphous blob, and a particle explosion (Figure 2.12). We use these as benchmarks to compare memory usage and rendering speed across different rendering approaches on a 3.2Ghz Intel Core 2 Duo processor with 4MB of L2 cache (only one core was used for computation). The approaches we compared are:

1. Our new approach: Marching slices with block subdivision and a 2D projection 1D lookup particle cache.

2. Surface-following marching cubes with blocking algorithm to allow for unbounded rendering, and a point insert, 3D lookup cache.

3. Surface-following marching cubes using a hash map to store corners, and a point insert, 3D lookup cache also using a hash map for particle lookup.

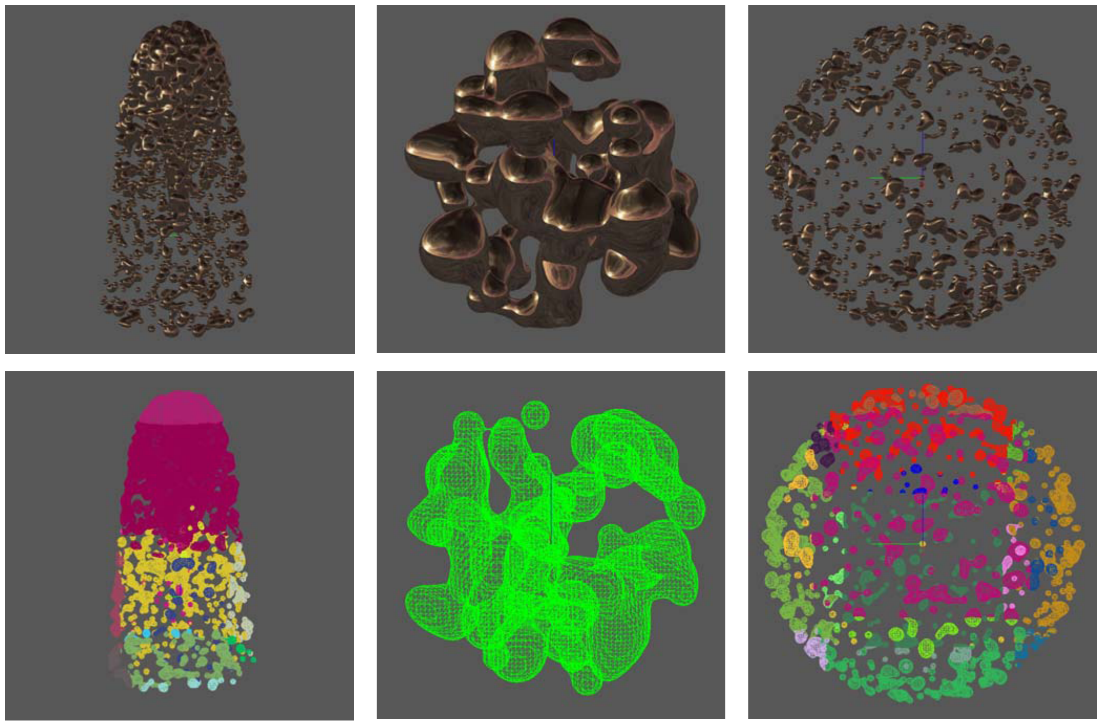


Figure 2.12: Screenshots of simulations used for performance comparison. Top row from left to right: fountain, blob, and explosion simulations. Bottom row: wireframe views of simulation with blocks identified by different colors.

Fountain (3000 particles, Rc = 3.0, T = 0.2, S = 1.0)

Approach	Cubes / Sec	Tris / Sec	Render Mem	Cache Mem	FPS
1	1,636,312	3,171,223	0.517	0.894	46.6
2	947,294	1,836,490	7.474	8.020	27.2
3	462,378	898,280	4.756	0.368	13.1

Fountain (3000 particles, Rc = 3.0, T = 0.2, S = 0.75)

Approach	Cubes / Sec	Tris / Sec	Render Mem	Cache Mem	FPS
1	1,859,083	3,657,317	0.741	1.548	29.1
2	1,033,835	2,033,716	10.513	8.022	16.2
3	477,330	939,837	8.611	0.364	7.4

Fountain (3000 particles, Rc = 3.0, T = 0.2, S = 0.5)

Approach	Cubes / Sec	Tris / Sec	Render Mem	Cache Mem	FPS
1	1,706,667	3,388,940	0.749	0.408	11.7
2	968,568	1,924,154	8.827	8.002	6.5
3	451,350	899,031	2.051	0.346	3.0

Blob (100 particles, Rc = 3.0, T = 0.2, S = 0.3)

Approach	Cubes / Sec	Tris / Sec	Render Mem	Cache Mem	FPS
1	2,145,530	4,288,108	0.246	0.137	77.7
2	1,228,188	2,458,939	8.655	8.000	49.3
3	559,283	1,118,880	3.179	0.022	22.5

Explosion (1000 particles, Rc = 3.0, T = 0.2, S = 0.6)

Approach	Cubes / Sec	Tris / Sec	Render Mem	Cache Mem	FPS
1	1,967,465	3,811,094	0.437	0.099	33.2
2	790,634	1,533,241	8.495	8.001	13.3
3	298,943	586,777	8.211	0.052	4.8

Figure 2.13: Performance comparison (memory is measured in MB). R_c is the cutoff radius, T is the threshold (a threshold of 0.2 corresponds to an isosurface of radius 1 for a single particle), S is the size of the cube grid.

All of these approaches were instrumented to measure frame rates, memory usage, and cubes and triangles rendered per second (Figure 2.13). We measured memory usage separately for the corner caches, which we refer to as *Render Mem* and for the particle lookup caches which we refer to as *Cache Mem*. These fields only measure active memory which is both written to and read from during algorithm execution, and do not measure the memory used for the vertex and index buffers, since these are statically allocated, and in our benchmarks, are the same size (960KB) for all three approaches.

We first compare Approaches 2 and 3 to evaluate the costs and benefits of using the block subdivision component solely versus the hashing approach. We see that over all the test scenarios, frame rates, cubes/sec and tris/second are approximately doubled when using block subdivision instead of hashing. However, memory usage is higher, mainly due to the large 3D arrays used for lookups of field values and nearby particles.

By replacing surface-following marching cubes with marching slices, and using a 2D insertion 1D lookup particle cache along with the block subdivision approach, we realize a significant performance improvement over the other approaches, with our algorithm performing 1.5 to 2.5 times faster than marching cubes with block subdivision and 3.5 to 6.9 times faster than the hashing approach across our benchmarks. We also significantly improve upon the biggest drawback of surface-following marching cubes, reducing memory usage by a factor of 8 to 43, and improving upon the memory usage of the hashing approach by a factor of 2 to 15 across our benchmarks. However, we notice that our particle cache memory footprint, while only a fraction of overall memory usage, is still 2 to 5 times larger than that of the hashing approach. This is a reasonable tradeoff given the increase in overall performance, and can be improved greatly in cases where large quantities of particles are rendered by using the approaches mentioned in section 2.7 for keeping particles cached only over the active slices.

Because of the small memory footprint of our approach, it can reside fully in the main memory of any modern computer along with other components of an interactive application (which in games include AI, physics, networking and world rendering). This is critical in interactive applications where virtual memory paging is unacceptable because it causes intermittent stalls, degrades performance and frustrates users.

Furthermore, because the memory footprint is so small, the algorithm can effectively take advantage of processor level 2 cache, avoiding cache misses which can stall the pipeline of a modern processor for as much as 500 cycles. Thus, we get a similar benefit as in out-of-core algorithms, but at one level higher on the memory hierarchy! This characteristic is especially beneficial on multi-core hardware which shares cache memory between multiple processes and hardware with small amounts of working memory such as hand held devices and cell processors.

We demonstrate the suitability of this algorithm in complex, fully interactive virtual environments. For our first experiment, we constructed a fountain simulated with $1,300$ particles and placed it in an outdoor video-game environment with virtual characters (Figure 2.6). The player can interact with the simulation by throwing objects and explosives into the fountain. The demo uses back-to-

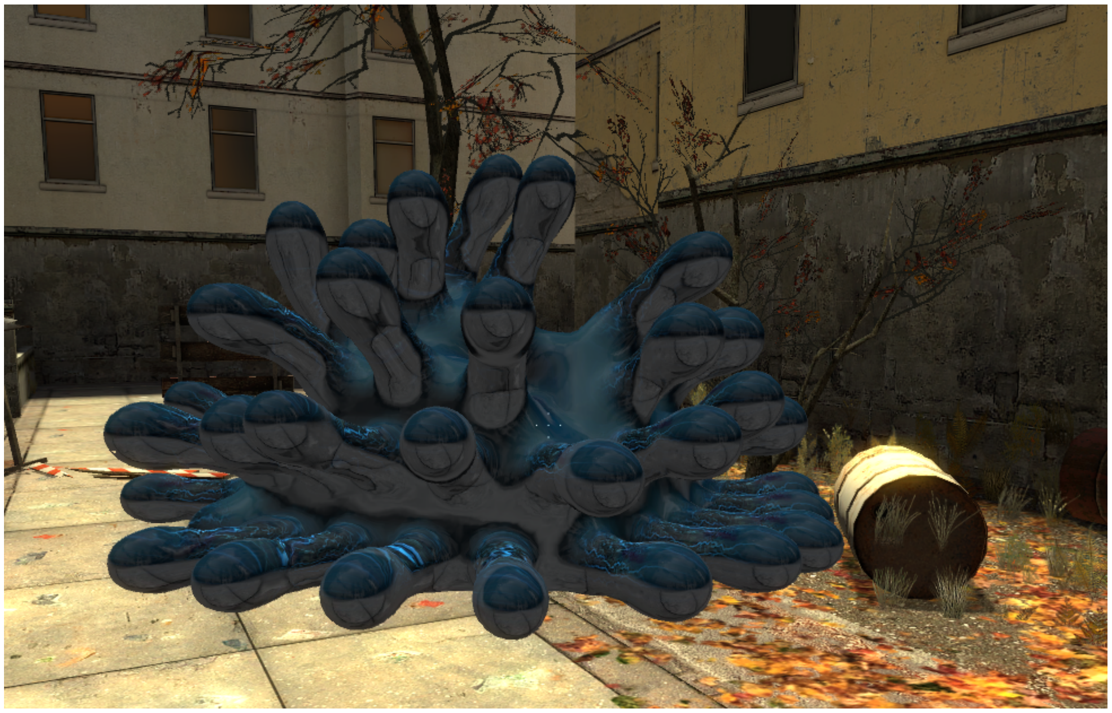


Figure 2.14: Here the blob monster is shown in the game environment with textured skin and fully extended tentacles.

front sorted rendering of polygons, and a refractive and reflective water shader for transparency, and runs interactively at a frame rates between 50fps at 60fps.

Our second experiment uses our algorithm to animate an amorphous particle monster (Figure 2.14). The monster is composed of 250 particles which are animated with several procedural behaviors such as emerging from a sewer, growing tentacles, forming a ball, coating a wall, and chasing players.

The tentacles, when grown, are cylindrically textured by interpolating per particle basis vectors and length coordinates onto the surface using a custom field calculation routine. These are then used to generate u, v texture coordinates into

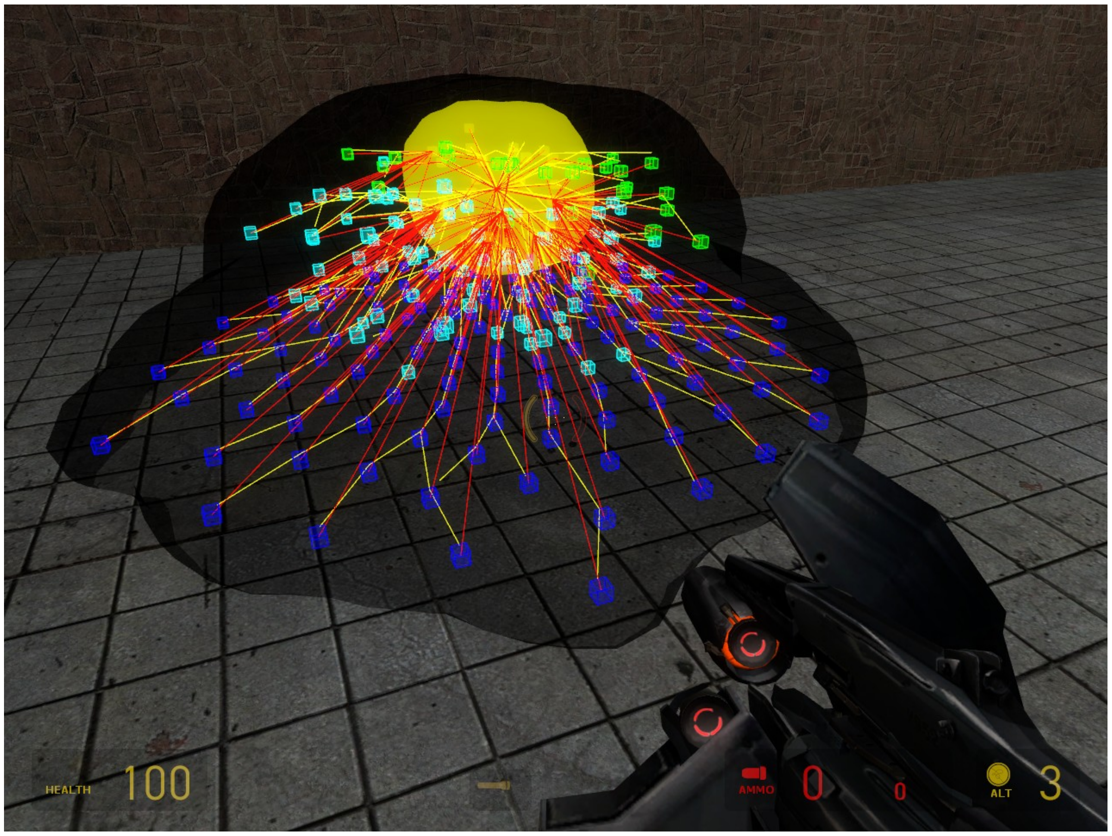
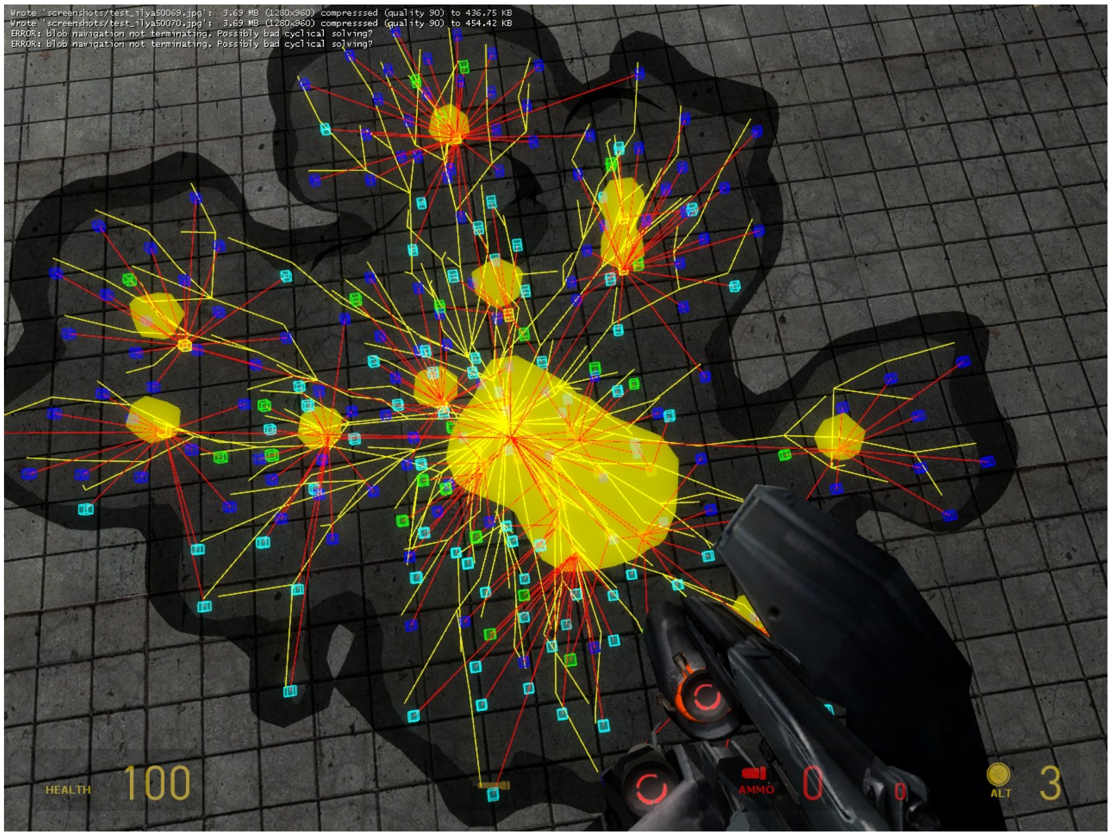


Figure 2.15: Procedural animation of blob monster. Squares represent particles, and the color of the square represents the particles function (yellow particles belong to the brain, light blue particles are structural, dark blue particles are feet, and green particles are arms. Yellow lines show the tree structure of the blob, and red lines represent “tendons” which apply forces to hold the blob together.

a preloaded texture by a custom vertex calculation routine. We also dynamically control the radius of a tentacle along its length by modulating the strength of field functions of the tentacle particles.

The monster is scripted to actively move around the scene, interact with objects, and chase the player. Furthermore, it can be blown to bits with explosive,



```
Wrote 'screenshots/test_ilya50069.jpg':  3.69 MB (1280x960) compresssed (quality 90) to 436.75 KB
Wrote 'screenshots/test_ilya50070.jpg':  3.69 MB (1280x960) compresssed (quality 90) to 454.42 KB
ERROR: blob navigation not terminating. Possibly bad cyclical solving?
ERROR: blob navigation not terminating. Possibly bad cyclical solving?
```

Figure 2.16: Here, the blob monster reassembles after being blown apart by an explosion. Note that it formed several mini monsters which are merging back together to form a big monster.

and will automatically reassemble into its original shape, using a set of rules to build a tree data structure which represents an internal skeleton (Figures 2.15 and 2.16). This demo runs interactively at 50-60fps depending on the number of physics objects the monster is colliding with and at 60-70fps during replay. Demo videos for both of these experiments along with various screenshots are included as supplementary materials in our I3D publication [36].

2.9 Conclusion and Future Work

In this chapter, we have presented a technique for real-time particle isosurface extraction. This approach consists of three novel components: a block subdivision algorithm which divides the render volume into seamless blocks, marching slices which renders isosurface in a slice-by-slice manner, and a 2D insert 1D lookup particle cache which enables fast and accurate lookups for field contributing particles. Using our technique, we demonstrated significant memory reductions and speed improvements over existing approaches for unbounded particle isosurface extraction. We also employed this approach in a video game environment, showing how our technique allows for rendering of dynamic particle-based entities with quality approaching that of offline techniques and has the potential to bring novel experiences to users of interactive applications.

Although we presented this algorithm running in a single thread, we have experimented with parallelized algorithms running on a quad core machine. We found that there is potential for a nearly linear speedup by parallelizing at the block-level for particle distributions that span more than one block, where incidentally, performance improvements are most important. Given that the latest generation of computing and game platforms have multiple cores, this is an important direction for future exploration. Furthermore, because this algorithm operates with 2D data structures, it should also be amenable to implementation on programmable graphics hardware, which is optimized for 2D texture lookups, and where we can expect tremendous performance gains, and many new and exciting applications.

3 IFSR - Interpolating Force Sensing Resistance

3.1 Introduction

This chapter describes my work in the area of multi-touch interaction. From the time that I started my research in this field and invented IFSR to the time that I sat down to write this thesis, multi-touch input has gone from being an active area of research, with limited commercial success to a technology that most people carry in their pocket, purse, or book in the form of a touch-screen phone or tablet.

Although multi-touch input has been an active area of research for nearly three decades [6], it still suffers from the absence of an easily available high-quality touch input device that is accurately responsive to pressure and can scale inexpensively to large or small form factors.

Sensors based on Interpolating Force Sensing Resistance (or IFSR) enable inexpensive multi-touch pressure acquisition. They can accurately measure entire images of pressure with continuous bilinear interpolation, permitting both high-frame-rate and high-quality imaging of spatially variant pressure upon a surface.

Though the use of force-variable resistors at multiple points of contact is not new [25], previous work in this area has focused mainly on arrays of discrete and independent sensors. The key difference between our technology and what existed previously is the newly developed principle of Interpolating Force Sensing Resistance (IFSR), which closely mimics the multi-resolution properties of human skin, in which the position of a touch can be detected at finer scale than the discrimination of multiple touches.

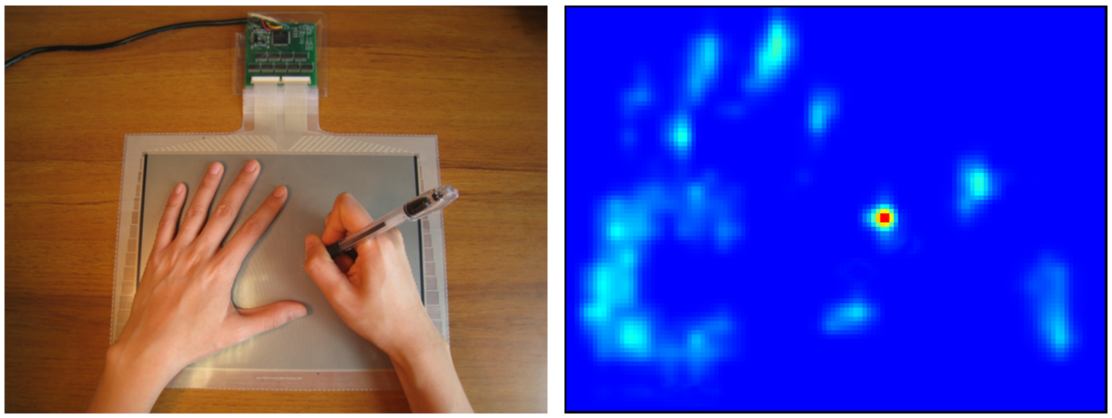


Figure 3.1: Writing on an UnMousePad and the resulting force image (warmer colors represent greater pressure). The red dot corresponds to the high pressure point created by the pen tip.

The development of IFSR based sensors and an improved understanding of their electrical properties enhances the type and quality of information that may be obtained in situations where entire images of pressure need to be captured and continuously tracked (Figure 3.1). IFSR based sensor technology is inherently unobtrusive, inexpensive, and very durable. It has a very wide range of potential applications in many sectors of society, primarily because it enables multi-touch pressure imaging at a low cost in a wide variety of form factors.

3.1.1 The Invention of IFSR

I was introduced to multi-touch interaction through Jeff Han's display wall project which was based on an optical touch sensing technology called FTIR [18]. As a member of that project, I developed a demo application consisting of an interactive virtual lava lamp based on the real time fluid rendering technology described in the previous chapter. The software allowed multiple users to move, squish and separate glowing blobs of lava by using swipe, pinch, and finger spreading gestures. Furthermore, because the display walls were pressure sensitive in addition to being touch sensitive, users had a third dimension of input – they could heat the lava, causing it to become more buoyant and float upwards, by applying pressure.

The major drawback of Jeff's display walls was that they required a complex and volume-consuming setup, including projectors and cameras behind the display. One day, after spending hours in the NYU black-box working with a massive FTIR display unit, I thought to myself that it would be revolutionary if it were possible to create something similar to an FTIR display that was inexpensive and compact enough to fit on one's desk. I realized that the FSR materials and manufacturing methods that we had used earlier (in a previous startup called SmartLines) could be adapted to create an array of tiny FSR sensors for touch interaction.

However, the main challenge with FSR was that the base materials were not transparent. I had been struggling to think of a way to get around this problem, when my advisor Ken Perlin suggested a very straightforward solution – an opaque sensor could still be extremely useful in scenarios where the sensor and display are separate. The sensor could sit on the table-top, potentially replacing a keyboard and mouse, and the user would look ahead at a display instead of looking down at their fingers. We called this user interaction modality LOTUS, which stood for Look Outwards, Touch Upon Surface.

With this as our vision, in early 2008, I set out to manufacture a force-sensing array using FSR. At first, I looked at standard FSR arrays, such as those built by companies like Tekscan [25]. However, these types of sensors could not achieve the level of accuracy that was necessary for touch interaction. The solution to this problem, which led to the invention of IFSR, is discussed in later chapters.

3.1.2 Birth of Touchco

In October 2008, my advisor and I, along with several students from NYU's Media Research Lab demonstrated some of the first sensors based on IFSR at the UIST conference in Monterey, California. Almost instantly, we were inundated with commercial inquiries from companies such as Wacom, Sony, Samsung, Nokia, Apple, Google, Microsoft and Amazon, just to name a few. We realized that we were onto something significant, and decided that it would be a good idea to start a company to commercialize the technology.

In early 2009, I assembled a team, registered a company named Touchco, and found some office space near NYU. Our goal was to develop the technology for inclusion into a commercial product, and we did this by developing both the hardware and software that was necessary to create a product based on IFSR. This included bringing up new vendors and finding suitable materials for IFSR production, im-

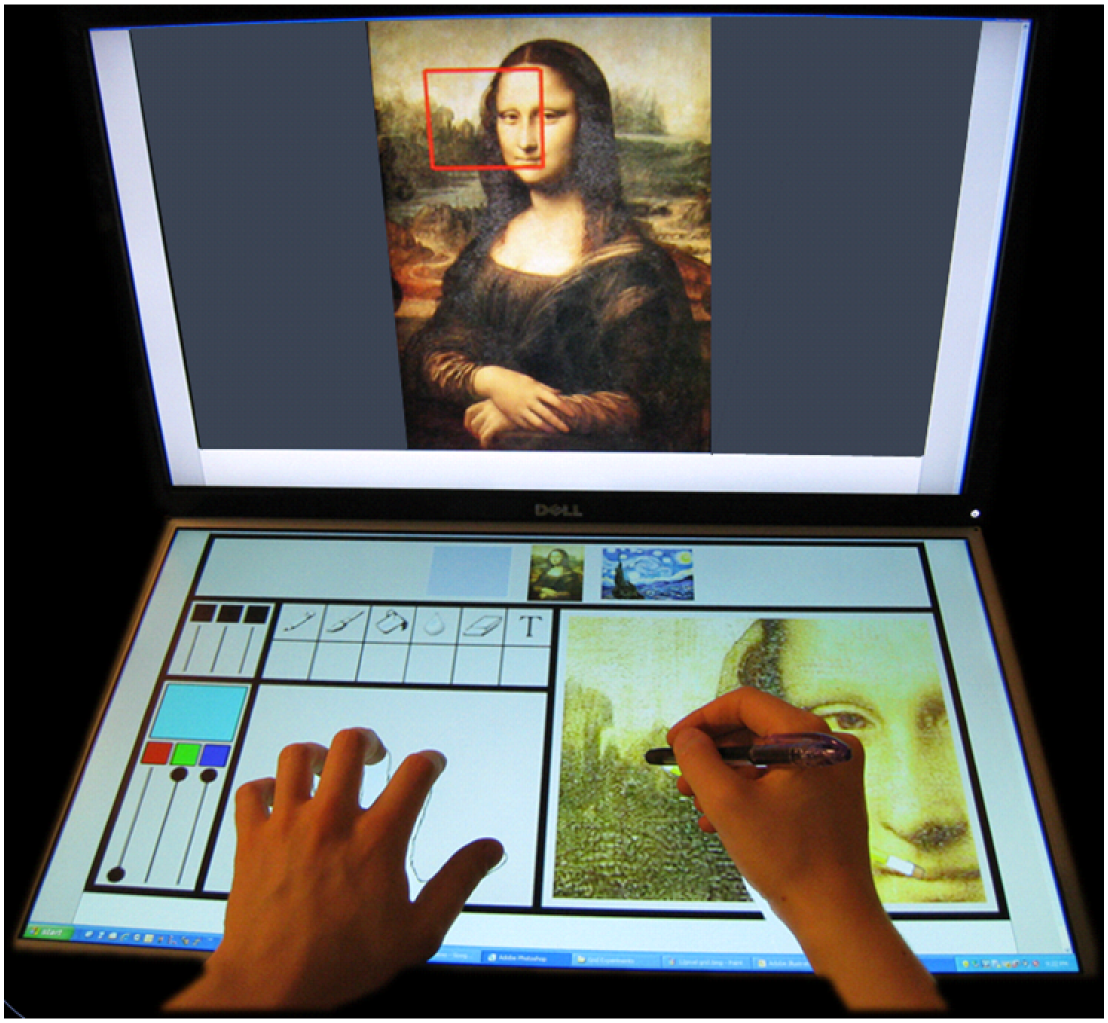


Figure 3.2: The desktop of the future concept device which was developed at Touchco combined a 24" transparent touch surface on top of an LCD with a second forward facing 24" LCD. The bottom LCD was used to display custom user interface controls. Here we show a drawing application, where the bottom touch-display shows controls and a detail view while the main display shows an overview of the final image.

proving the manufacturing and assembly process for our sensors, performing reliability testing, developing low cost scanning electronics, developing algorithms and firmware, creating drivers and demo applications for multiple operating systems, and creating custom prototypes and development kits.

We received an incredible amount of external interest in a wide range of applications of the technology. Some of the applications that we decided to pursue included sensors for measuring foot pressure distribution, measuring how people sat on seats, wrapping around pens, adding touch interaction to drawing tablets, creating drums, keyboards and new musical instruments, and using IFSR as a touch sensor below flexible displays. We also made significant progress on a transparent version of IFSR which was used to create a prototype device we called Desktop of The Future (Figure 3.2). This and several other concept devices were demonstrated in Emerging Technologies at SIGGRAPH 2009.

3.1.3 Acquisition

In mid-2009, we were approached by one of our customers with an offer to acquire Touchco. We believed that working with a major consumer electronics company would speed the development of our technology and that together, we would create some truly iconic, world-changing devices, especially because our technology worked so well with electronic paper displays, which rely on reflected light for readability and thus would appear much more paper-like if the touch sensor could be placed below the display. Although we were able to significantly advance the performance and manufacturing of the IFSR technology, due to internal politics and a series of events outside of our control, the technology was not commercialized, and the founders of the company went off to work on other projects.

Although IFSR was not commercially successful, I believe that the technology has inspired a host of other pressure sensitive touch technologies including Synaptics' ForcePad [41] and Tactonic's Interpolating sensors [44]. I also believe that it's not the end of the road for the technology, and that one day, there will be a renewed interest and commercial products based on IFSR.

3.2 Related Work

Multi-touch input technologies fall mainly into three broad categories: optical, capacitive and resistive. Each of these affords a particular set of advantages and disadvantages.

3.2.1 Optical Sensing

Optical multi-touch sensing generally incorporates a digital video camera behind the touch surface to form an image of a user's fingers and possibly hands. Microsoft Surface [11] integrates images captured simultaneously from several digital video cameras to obtain both touch and proximity information. In systems based on Frustrated Total Internal Reflectance [18] a finger or hand touch disrupts the path of infrared light that is undergoing total internal reflectance inside a glass surface. The scattered light is captured by a digital video camera placed behind the glass. The ability to determine touch pressure by incorporating a soft light-scattering layer over the touchable FTIR surface has also been demonstrated [10]. Because of the use of a camera at some distance behind the touch surface, many of the early optical multi-touch technologies tended to be bulky and sensitive to lighting conditions.

More recently, attempts have been made to incorporate optical sensing into the bezels or into the display surface of the display itself. Companies such as RPO, Neonode, ELO Touch Systems, Flatfrog and RAPT have developed touch sensors which work by detecting interruption of IR beams fired either slightly above a display or within a display's cover-glass layer. This type of touch technology, which was originally developed by Bell Labs [7], has proliferated since the patents expired. Attempts have also been made to incorporate optical touch technologies into display backlights [46, 8] and into display pixels themselves [12]. Although these technologies are very promising, they require tradeoffs and complex integration between display and touch technologies and have thus not been broadly incorporated into commercial products.

3.2.2 Capacitive Sensing

Recently, multi-touch capacitive sensing has become the most common technology used in touch-screens. One of the best known and iconic implementations of a capacitive multi-touch system is the touch-screen of the Apple iPhone [21], a successful adaptation of technology developed by FingerWorks [49]. Other early multi-touch systems included the DiamondTouch system [13], developed by Mitsubishi Electric Research, which was able to distinguish between different users by incorporating each user's body into a unique capacitive circuit, and SmartSkin

[35], developed by Sony Computer Science Laboratories, which was able to detect several different gestures, including hover above the touch surface.

The majority of multi-touch capacitive systems are based on a technology called mutual capacitance, or projected capacitance. To detect touches, a touch controller drives a series of transmit electrodes with an electrical signal at a known frequency. That frequency is then received by a series of receive electrodes, which form a grid with the transmit electrodes. The presence of a finger absorbs some of the transmitted electric fields, reducing the capacitive coupling between the transmit and receive electrodes. This reduction of capacitive coupling is then interpreted as a touch.

One of the biggest advantages of capacitive touch technologies, is that there are no moving parts, and unlike optical technologies, they are insensitive to lighting conditions. Furthermore, they can easily be made transparent using transparent conductors such as ITO. However, capacitive touch is highly susceptible to electrical noise. Noise can come from a variety of sources including chargers, fluorescent lights, and displays. Noise increases the power usage of capacitive sensors since multi-sampling and many extra layers of analog and digital filtering are necessary to deal with the noisy signals. Furthermore, the signal strength varies strongly based on the user's skin conductance and the amount of capacitive coupling between the user, the device and a ground reference. All of these factors make capacitive touch sensing incredibly complex and difficult to implement well in practice.

Another drawback of capacitive touch systems is that they can only measure the surface area of a contact, and not the level of force applied. Furthermore, that measurement is often imprecise due to variations in skin conductance between individuals. This limits touch interaction to a vocabulary of taps and swipes, and misses out on our ability to apply varying levels of force. To address this issue, companies such as Pressure Profile Systems (PPS) have developed a capacitive force sensing array technology [42]. Their sensors are created by sandwiching a compressible material such as silicon between a set of row and column electrodes. Pressure applied to the sensors deforms the silicon, bringing the plates closer together and increasing their capacitive coupling. This change in capacitive coupling can be detected by electrical circuitry and interpreted as a touch. Today, this technology has primarily found use in medical and industrial applications, as the electronics are too complex and expensive for most consumer electronics.

Another approach for creating a force-sensing touch pad is to suspend a capacitive touch sensor on top of four force sensors located at the corners. One such technology is Synaptic's ForcePad technology [41]. The drawback of this technology is that it is mechanically complex and that it can only measure the centroid of the force applied, and can not distinguish the force applied by the individual touches.

A further limitation of capacitive touch technology is its reliance on the conductive properties of the water within the human body – most capacitance-based systems are only able to track a finger, not a stylus or other non-conductive object. One exception is the N-trig DuoSense [32] technology, which is able to track both fingers and a stylus, by making use of an active pen. These special pens use a battery to power an integrated circuit which transmits a radio signal through a pressure sensitive tip. However, the pens used in these systems are expensive, and the accuracy is often poor due to electrical noise and because the sensor needs to triangulate the position of the stylus based on the strength of radio signals rather than directly sensing the contact point of the pen tip.

3.2.3 Resistive Force Sensing

Multi-touch sensors based on arrays of resistive sensors can have a flat form factor, are inherently inexpensive, use little power and can measure applied force. Such devices are typically based on the principle of Force Sensing Resistance [15]. An FSR device is a continuous electrical switch in which electrical conductivity increases gradually as external force is applied. In one common configuration, which is often referred to as "through" mode, two conductors which have both been coated with FSR ink are placed into mutual contact.

The FSR principle is often used to create discrete sensor arrays. An inherent limitation of such arrays is their inability to track at resolutions finer than the spacing between successive sensing elements, and their inability to track a stylus when it enters the dead zones between sensels. For this reason, the JazzMutant and Stantum sensors [22] are limited to applications such as music remixing that do not require high precision, whereas Tekscan sensors [25] cost thousands of dollars due to the high cost of electronics required to scan a high-density array. Another approach to creating multi-touch pressure sensors is to combine multiple buttons that can each sense a position and a force [48]. However, these devices still suffer from the inability to track the position of styli and fingers between sensor cells.

To improve tracking, one could put a thick sheet of rubber over existing devices, blurring the force image, and thereby creating a form of positional interpolation. However, per our experiments, such a layer of rubber needs to be approximately half an inch thick to have any appreciable effect. Besides the packaging, weight and aesthetic issues that this creates, the blurring makes it impossible to distinguish a stylus press from hand/finger input. In addition, the damped signal has higher latency and greatly reduced temporal resolution. Recently, a company called Tactonic Technologies [44], has developed a structured material which solves some of the issues associated with mechanical interpolation. However, the Tactonic sensor still requires and additional force sensing layer above the sensor to operate. Our approach allows for high-resolution, high-speed, force-sensitive positional tracking without the expense of a high-resolution discrete FSR array, or the signal losses associated with a mechanical blurring layer.

3.3 Our Contribution

An all-purpose pressure-sensitive multi-touch input device needs to do at least two quite different things: (i) track a stylus with high resolution, and (ii) distinguish between different fingers of a human hand (Figure 3.3). These are very different requirements. Fingertips are relatively large, and their centers never get much closer to each other than about half an inch. In contrast, a stylus tip generally needs to be tracked to within a fraction of a millimeter. In addition, while the distance between nearby fingers may be large, fine position sensing is useful for detecting subtle finger movements, such as leaning and wiggling.

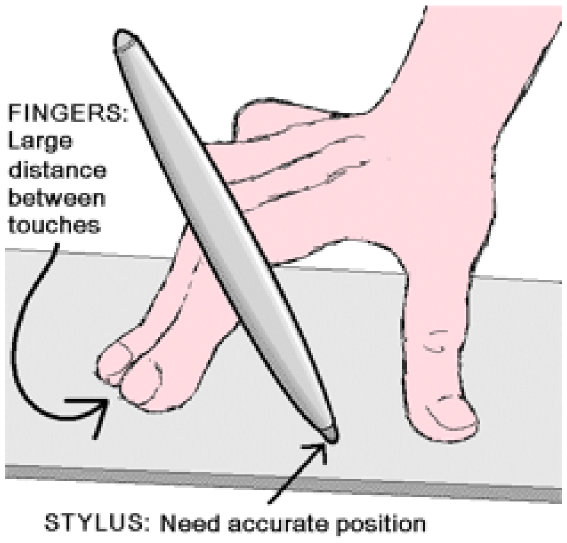

FINGERS: Large distance between touches
STYLUS: Need accurate position

Figure 3.3: Finger touches remain far apart, but a stylus must be tracked precisely.

What has not been shown previously is a multi-touch input technology that provides, simultaneously, a flat and flexible/bendable form factor, the option of transparency, ability to inexpensively scale up to large sizes, accurate pressure sensing, self-calibration, interpolation between touches, low power and low cost. We have achieved this by revisiting the use of FSR materials in a novel configuration. In the remainder of this chapter, we discuss the following key contributions:

1. The general principle underlying the sensor.
2. Construction of IFSR sensors.
3. Methods for electrically scanning IFSR sensors.
4. Methods for improving accuracy.
5. Methods for interpreting device output in software.
6. Applications of IFSR sensors.

3.3.1 Operating Principle

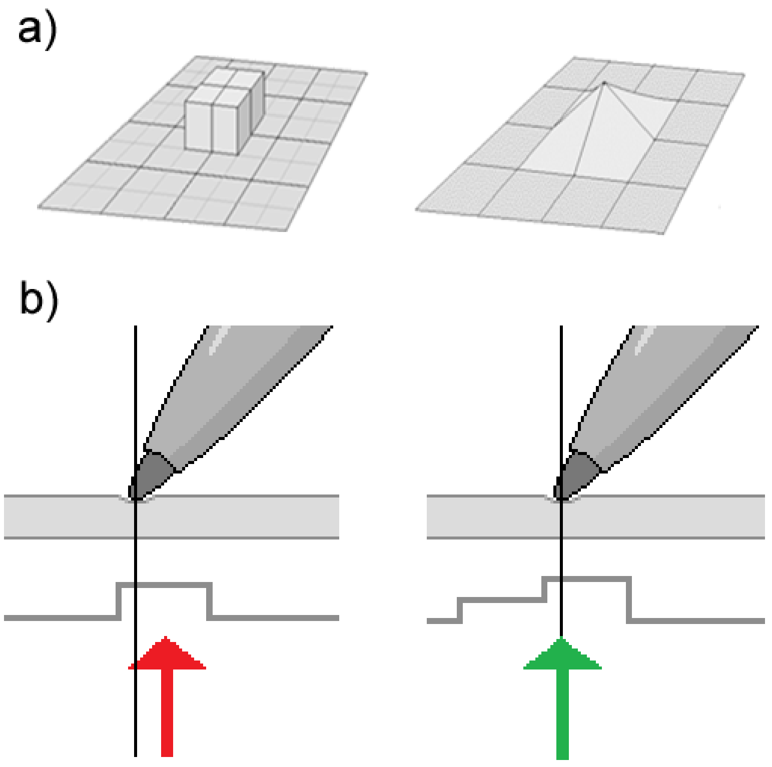


Figure 3.4: Discrete versus bilinear sampling. a) Area response of one sensel of a discrete FSR sensor (left) vs. an IFSR sensor (right). b) Reconstruction of pen position with a discrete FSR sensor yields an error in position (left), while an IFSR sensor properly reconstructs position with minimal error (right).

Interpolating Force Sensing Resistance (IFSR) is a new and cost-effective method we have developed for using FSR to capture images of pressure upon a surface. With pre-existing arrays of discrete FSR sensors, it is not possible to correctly reconstruct the position of a point touch with a low-resolution grid. In contrast, the IFSR has sensels with overlapping regions of sensitivity (see Figure 3.4). The spatial drop-off in sensitivity at each sensel of an IFSR is near-linear on both X and Y axes, resulting in a piecewise bilinear response kernel. The bilinear kernel allows the position of any touch, even a very small point touch, such as that of a stylus tip, to be accurately interpolated by calculating the voltage-weighted average of the sensel positions (see Equation 3.1). Effectively, the IFSR yields an anti-aliased image of the pressure applied to it.

$$(X, Y) = \left(\frac{\sum_i \sum_j i * V_{out}(i,j)}{\sum_i \sum_j V_{out}(i,j)}, \frac{\sum_i \sum_j j * V_{out}(i,j)}{\sum_i \sum_j V_{out}(i,j)} \right) \tag{3.1}$$

As a result, the IFSR has two notions of resolution. The first is how close two touches can be before they can no longer be distinguished from each other. We refer to this as *grid resolution*. The second measures the much finer resolution at which a single point can be tracked. We refer to this as *positional resolution*. An IFSR allows grid resolution to be much lower than positional resolution. Because a sensor with lower grid resolution has less active electrodes, compared to a discrete FSR sensor, an IFSR sensor with similar positional resolution can be manufactured with relatively inexpensive drive and A/D conversion electronics. Because of the lower grid resolution, IFSR sensors also require less bandwidth to process and transmit the acquired data.

3.3.2 UnMousePad Construction

We have built IFSR sensors in various form factors including credit-card sized sensors, large disk-shaped sensors, transparent sensors, and 12” x 16” sensors for

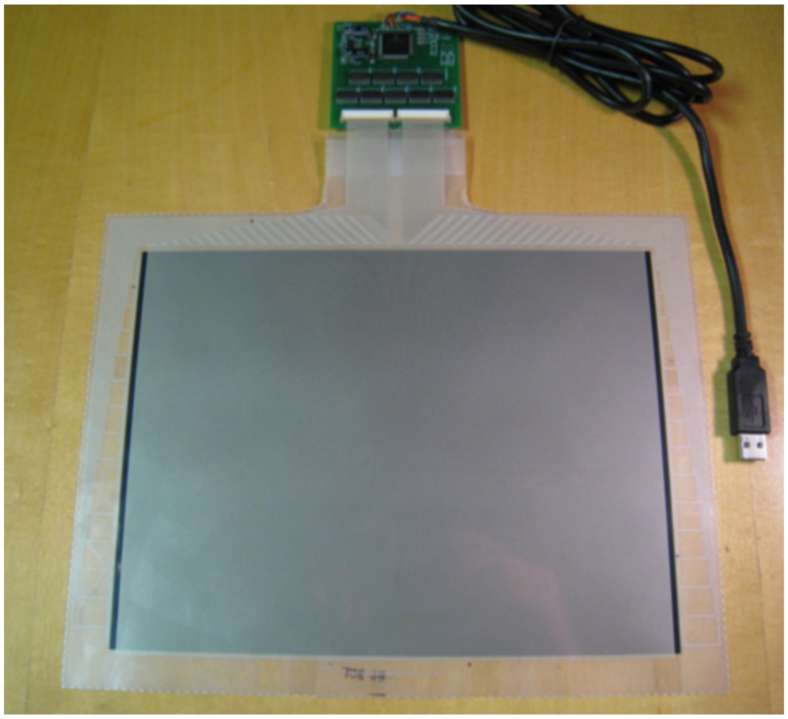


Figure 3.5: 8.5" x 11" UnMousePad with drone wires

two-handed operation. We affectionately call our 8.5" x 11" form factor IFSR device (shown in Figure 3.5) the *UnMousePad*. It is conveniently sized like a page of standard letter paper. It has an 6.85" x 9.21" active sensing area consisting of a 40x30 grid of sensels spaced at 6mm intervals – sufficient to obtain two samples per finger width, and therefore to reliably distinguish between two fingers even when those fingers are very close together.

The UnMousePad consists of two paper-thin 8.5" x 11" sheets of PET plastic attached together at the edges. On the inner side of the top sheet is a circuit pattern consisting of 40 parallel active column electrodes spaced at 6mm (with non-active drone electrodes in-between at 1mm intervals). The circuit pattern is coated with

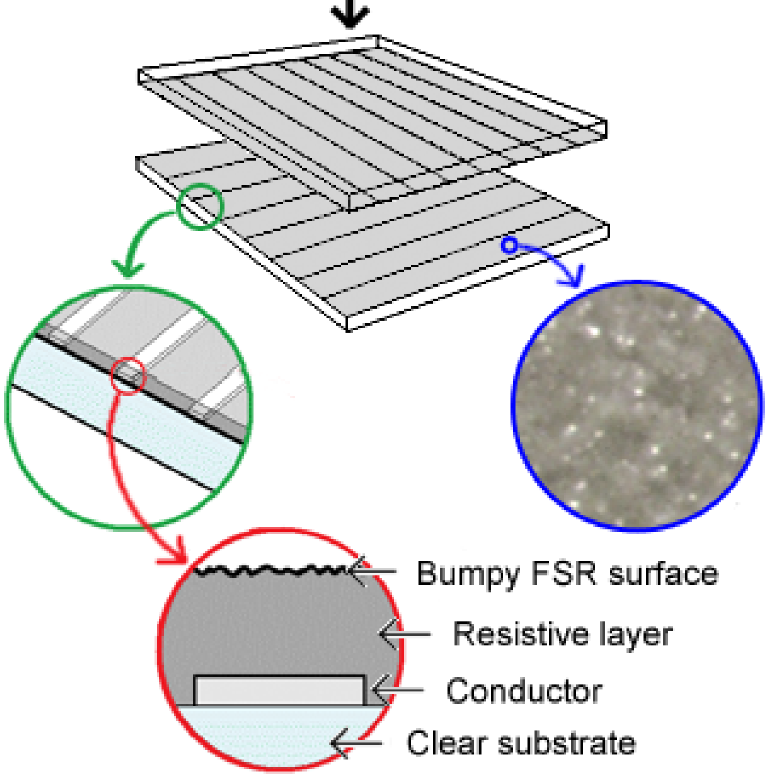


Figure 3.6: The UnMousePad is constructed by sandwiching together two sheets with electrodes at right angles. The electrodes are covered with a thin layer of FSR ink. A contrast-enhanced image of FSR material at 20x magnification is provided.

a thin, solid layer of FSR ink. As this ink dries, its exposed surface hardens to form microscopic bumps (see Figure 3.6). Because the sensor uses a solid layer of ink, it is easy to align it with the electrode layer, reducing manufacturing cost. A printed wire runs from each electrode to a connector area that is provided on one side for interfacing with electronics. The inner side of the bottom sheet has a similar pattern with 30 row electrodes which are perpendicular to the column electrodes.

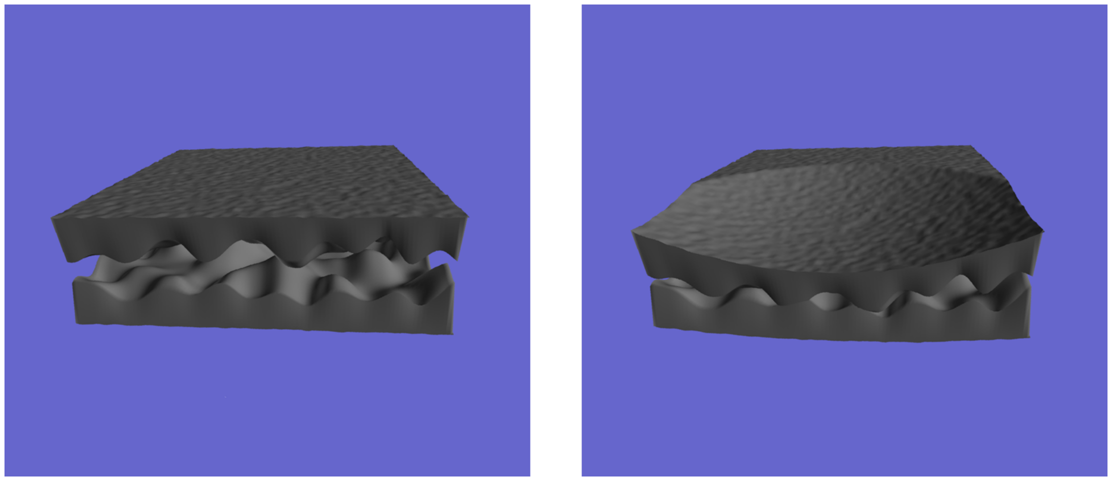


Figure 3.7: As pressure is applied to the sensor, the FSR on the top and bottom layer intermeshes, increasing conductivity.

Force sensitivity results from the upper and lower bumpy FSR layers intermeshing with each other as pressure is applied (Figure 3.7). Each point of contact creates an additional current path. These current paths can be thought of as parallel resistors. Because the conductivity of parallel resistors sums together, the number of contact points increases as pressure is applied, which results in a proportional increase in conductivity.

A circuit board, with off the shelf electronics and a micro-controller, is connected to the top and bottom layers. The circuit board reads values from the UnMousePad and sends pressure images to a computer via USB. In our implementation, we drive the column electrodes and read out signals on the row electrodes. But, it is worthwhile to mention that the sensor is bidirectional – it can just as well be driven on the row electrodes and read out on the column electrodes.

3.3.3 Scanning the UnMousePad

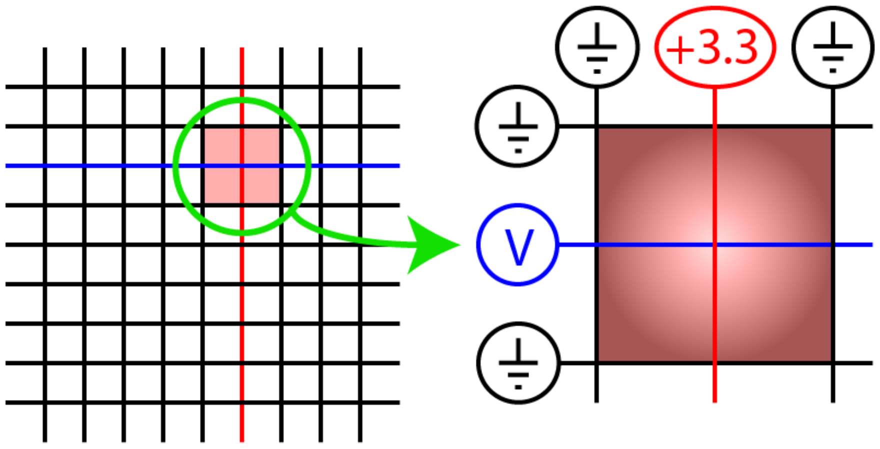


Figure 3.8: One time-slice during a sensor scan. The vertical red line indicates a powered column electrode, the horizontal blue line indicates a metered electrode, and black lines indicate grounded electrodes. The pink area illustrates bilinear sensitivity in the metered region which is centered around the powered column and metered row electrodes and bounded on all sides by grounded electrodes.

A micro-controller on the circuit board scans all row-column intersections in succession. At any given moment of the scan, some column i is connected to a positive voltage source (+3.3V for our micro-controller) and some row j is connected to an A/D input port on the micro-controller, which measures output voltage. Meanwhile, all other rows and columns are connected to ground (see Figure 3.8).

Consider the highlighted (pink) region in Figure 3.8. If the sensor is not being touched in this region, a steady stream of current will flow from the positive voltage source to the neighboring grounded electrodes to the left and right along the top FSR surface, but no current will flow between the top and bottom surfaces.

When a user touches the UnMousePad in the highlighted region, current is able to flow through to the bottom surface. Some of this current goes to the nearest grounded electrodes on the bottom surface above and below, and some of it goes through the metered electrode to the circuitry that is measuring voltage (Figure 3.8). As the position of the touch moves nearer to the intersection between the positive voltage source and the metered output line (i.e. toward the center of the highlighted region), the measured voltage increases.

3.3.4 Avoiding False Positives

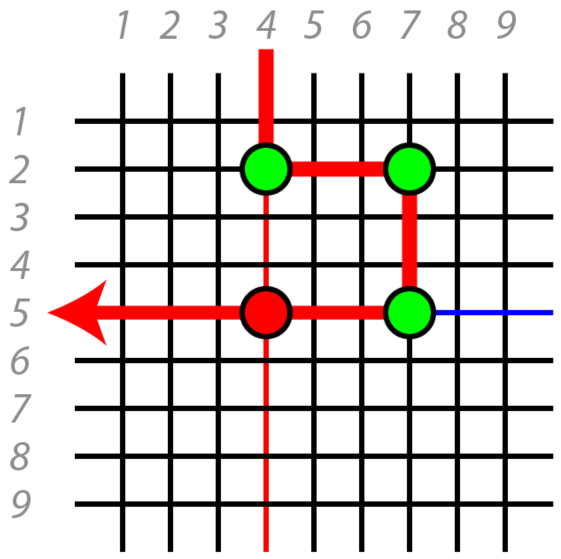


Figure 3.9: False positives are avoided by grounding all non-active electrodes. Green dots indicate real touches. The thick red line indicates a current path that would result in the detection of a phantom touch (red dot) if the non-active electrodes weren't grounded.

It might seem that a multi-touch device with passive junctions (as opposed to a transistor at every row-column intersection) would produce false positives. In

Figure 3.9, green dots indicate three touches where wires cross, and the red dot and path indicate an erroneous phantom touch that might result when input voltage is applied to column 4 and output voltage is measured at row 5.

The UnMousePad does not suffer from this problem because at any moment during the scan, all conducting lines that are not set to either the positive voltage source or the metered output are set to ground. In the above example, the current would simply drain to ground at both row 2 and column 7, rather than at row 5.

3.3.5 Interpolation Linearity

Ideally, the drop-off in response with respect to distance from a row/column intersection should be linear in both the X and Y directions. However, our first generation of UnMousePad prototypes had very poor interpolation behavior – sensitivity as a function of distance from the active intersection fell off much faster than expected.

In theory, this non-linearity could be compensated for in software, by use of a look-up table or by analytically solving for position of a point given the signal from the sensor. However, we found that a non-linear response reduced the sensor's signal-to-noise ratio, making this approach impractical. After careful analysis and experimentation, we found that there were two sources of error. We discuss each one in turn, together with methods to reduce them.

Radial Spread Non-Linearity

Touching where two wires cross (top of Figure 3.10) requires very little travel of current transversely through the high-resistance FSR layer, whereas touching far

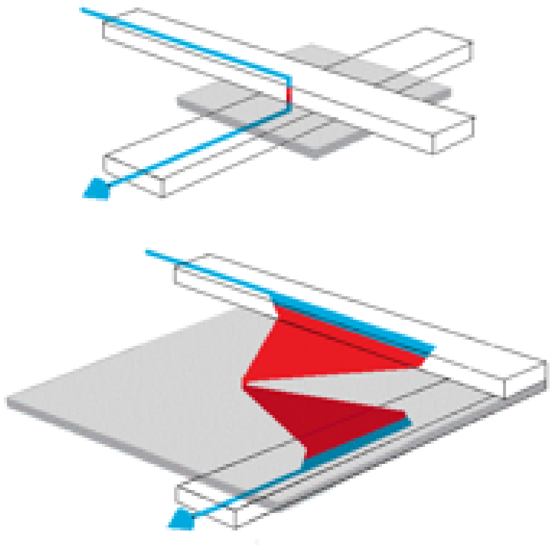


Figure 3.10: Touch sensitivity at corner (top) and center (bottom) of a 6mm cell. For a touch at the corner of the cell, current travels a short distance through the low-resistance metal trace, shown in blue, before traveling through the junction between the top and bottom FSR layers. For a touch at the center of the cell, current must travel transversely through a layer of high-resistance FSR ink (shown in red) before crossing the junction.

away from the nearest conductor (bottom of Figure 3.10) requires a significant distance of current travel within each FSR ink layer. Because the current spreads radially around the touch point, the resistance increases non-linearly as the touch point moves farther away from the area where conductors cross.

Using a field-solver that I implemented to precisely simulate the flow of current through the layers of the UnMousePad, we can see the radial pattern in the gradient of voltage around a touch point (top of Figure 3.11). This pattern occurs consistently with touches of varying force and size, and is greatest for a touch near the center of a cell.

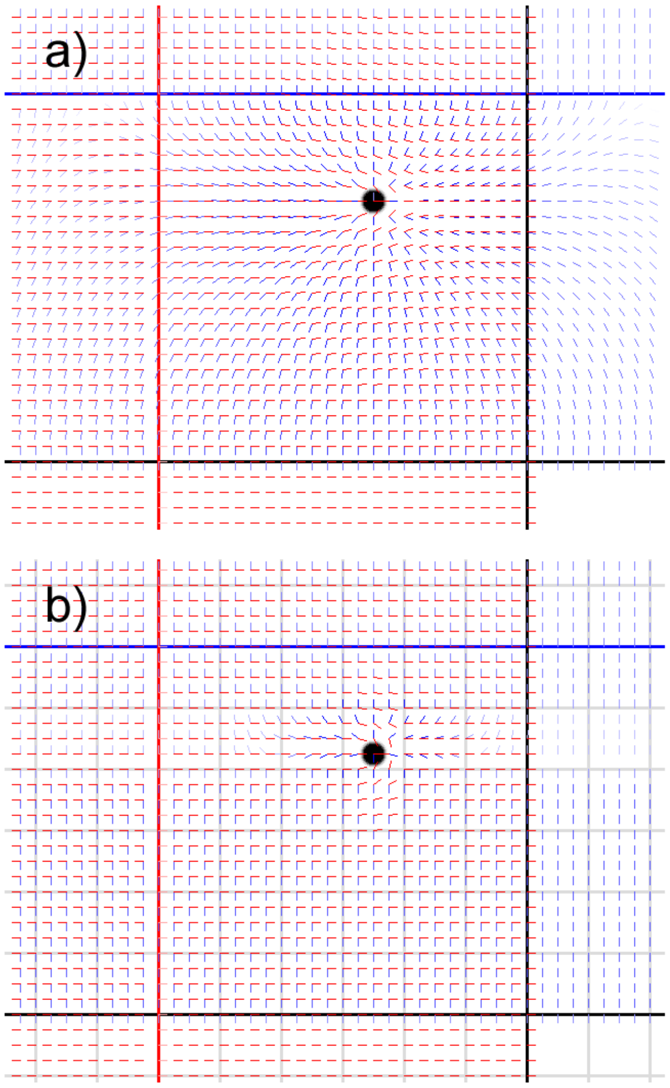


Figure 3.11: Potential field for a 100g point touch on a sensor with and without drone wires (computed with field solver). Thick red, blue and black lines represent powered, metered and grounded electrodes respectively. Red and blue vectors correspond to gradients on the top and bottom surfaces respectively. a) Without drones, potential field is curved, leading to non-linear response. b) With drones, linearity of potential field is greatly improved.

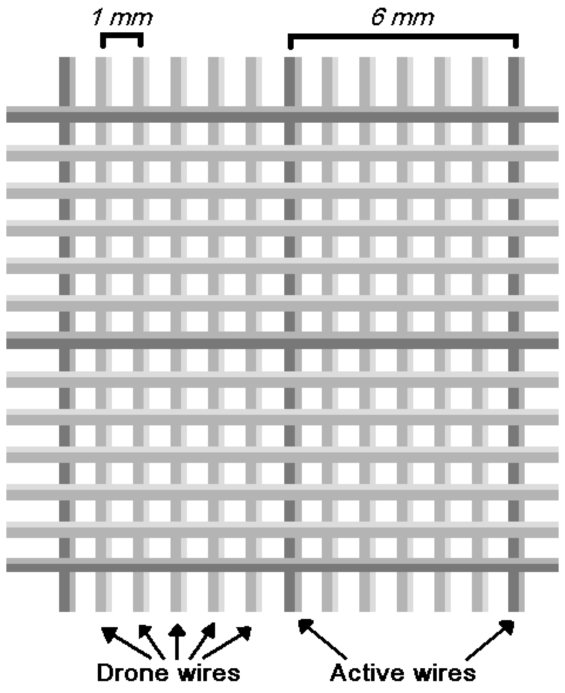


Figure 3.12: Drone wires are inserted between active row and column electrodes.

We addressed this issue by altering the printed conductor pattern to insert, between every pair of active wires, numerous parallel “drone wires” (see Figure 3.12). These passive wires have the effect of greatly shortening the maximum effective path that electric current needs to travel through the resistive FSR ink in one direction, effectively creating an anisotropically conductive surface. This shapes the voltage potential field to vary mostly along the direction perpendicular to the wires in a given layer.

As the number of drone wires increases, the scale at which the radial spread of current occurs proportionately decreases, and would approach zero for an infinitely fine drone conductor pitch. With our current manufacturing technique, we can reliably print electrodes with 1mm spacing. Thus, in the UnMousePad, we place five drone wires between each pair of active electrodes. This greatly improves the tracking resolution for fingers and styli and the accuracy of force measurements (see Section 3.4.5).

Voltage Divider Non-Linearity

While it is possible to completely eliminate the non-linearity due to the radial spread of current by introducing a large number of drone wires, there is a second form of non-linearity that results from the fact that when a touch occurs, the current flowing from the top layer of the sensor to the bottom layer disrupts the voltage gradient between electrodes on both the bottom and top layers of the sensor. This effect would not go away even if there were an infinite number of drone wires. Furthermore, this effect cannot be fully compensated for in software because error in position and error in force are difficult to distinguish.

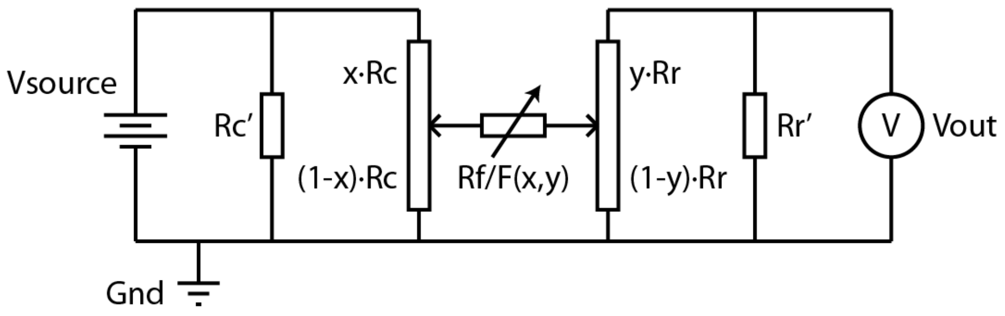


Figure 3.13: Schematic of effective circuit at one grid cell for a single point touch.

To understand the impact of this effect and find ways to reduce it, we created a simplified electrical schematic that models the behavior of a sensor cell in response to a single point touch (Figure 3.13). In this model, V_{source} is the voltage applied to a powered column. V_{out} is the voltage measured at the metered row. x and y are the positions of the touch and vary from (0, 0) to (1, 1). R_f is the resistance between the top and bottom layers of FSR material. We refer to this as the "through" resistance. $F(x, y)$ is the force applied at point (x, y). R_c is the resistance between two adjacent column electrodes of the sensor. R_r is the resistance between two adjacent row electrodes.

R_c and R_r depend on sensor design parameters which include the "transverse" resistance across the surface of the FSR material (R_{fsr}), the length and spacing of row and column electrodes (L_r, L_c, S_r, S_c), the density of drone wires (D), and the thickness of FSR material (T_{fsr}). R'_c and R'_r are the resistances of all other current paths to ground from a row and column electrode, respectively, and are equal to Rc and Rr in our sensor. We can estimate the values of these four constants with some simple math based on the sensor design parameters (Equations 3.2 and 3.3):

$$R'_c = R_c = \frac{R_{fsr} * S_c * (1 - D)}{L_c * T_{fsr}} \tag{3.2}$$

$$R'_r = R_r = \frac{R_{fsr} * S_r * (1 - D)}{L_r * T_{fsr}} \tag{3.3}$$

The following set of three equations (Equations 3.4, 3.5 and 3.6) result from solving for $V_{out}(i, j)$ for a given pressure distribution $P(x, y)$ over the modeled sensor area A:

$$V_{out} = \iint_A \frac{a * V_{input} * x * y}{b(x - x^2) + c(y - y^2) + R_f/P(x,y)}\, dx\, dy \tag{3.4}$$

$$b = \frac{1}{1/R_c + 1/R_c'} \approx R_c/2 \tag{3.5}$$

$$a = c = \frac{1}{1/R_r + 1/R_r'} \approx R_r/2 \tag{3.6}$$

We notice that the resulting equation has three constants, a, b, and c, which are dependent on R_c and R_r. Looking at the model, we see that two of these terms, b and c, control the linearity of the response – if b and c were both zero, V_{out} would be perfectly linear with respect to the magnitude and position of an applied pressure distribution. However, in practice, reducing these values results in decreased output voltage and increased consumption of power by the device. Thus, the best we can do is find a ratio of b and c that is low with respect to $R_f/P(x,y)$, and yet high enough to allow the external electronics to read the sensor.

Luckily, there are two factors that work to our advantage, producing a desirable ratio. The first is that in the useful range of operation for a human operator, the FSR inks we use tend to have a fairly high through resistance. In our prototype, this resistance varies between 1.2 M Ohms and 2.2 K Ohms for forces between 5 grams and 200 grams. The second is that with proper selection of FSR inks, we can attain values of b and c that are approximately 300 Ohms and 400 Ohms respectively. Thus, we have approximately an order of magnitude difference between the

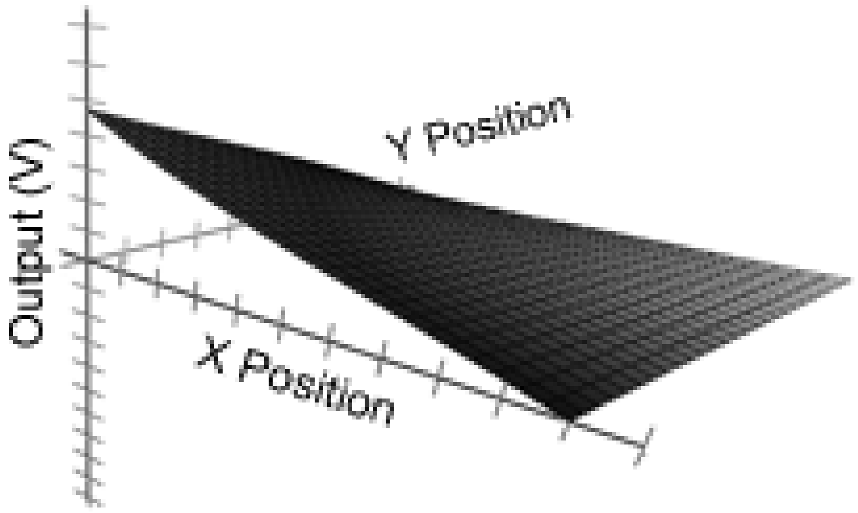


Figure 3.14: Calculated response of a single row/column intersection of the UnMousePad sensor resulting from a point touch of 100 grams at different (x, y) locations. Voltage decays in a near-linear fashion along X and Y axes.

"through" and "transverse" resistance. The result is a nearly linear sensitivity (see Figure 3.14), which increases the signal-to-noise ratio of our sensor and results in linear tracking of both finger touches and point contacts, such as those of a stylus.

3.4 UnMousePad Characteristics

3.4.1 Electronics

The main component on the UnMousePad circuit board is a PIC24H micro-controller produced by Microchip Technology, Inc. which uses five M74HC595 shift registers to power one column electrode at a time while grounding all others, then reads analog voltage values from each even row, followed by each odd row, while four other M74HC595 shift registers are used to alternately ground the odd or the even rows respectively. The micro-controller then switches to the next column and repeats this process for the remaining columns. The micro-controller converts the analog voltage values to digital values via an onboard 12-bit analog-to-digital converter. Finally, the micro-controller sends the complete frame of data to the host computer and starts the process all over to scan the next frame.

3.4.2 Scan Rate

The data stream from the UnMousePad consists of 1200 pairs of bytes forming a 40x30 image of force on the sensor. Each frame is followed by a terminating sequence. Whenever a run of successive zeros is encountered, a special code followed by the number of zeros is sent, effectively run-length-encoding long strings of zeros for areas of the sensor not being touched. The host computer receives, decompresses, and processes these pressure images at approximately 60 frames per second. The measured latency is 1/60th of a second. The rate-limiting factor is the USB transceiver chip, which restricts the data rate into the computer to 900kbits/sec. Because the A/D converter on the micro-controller is capable of up to a million samples per second, it is possible to attain sample rates of over 500 frames per second by switching to a faster USB transceiver.

It would appear that this method of scanning would not scale well to larger sensors, and would be insufficient for applications such as musical instruments where scan rates over 1000 Hz are necessary, or for mobile devices that need to draw as little current as possible. However, the IFSR architecture has a useful

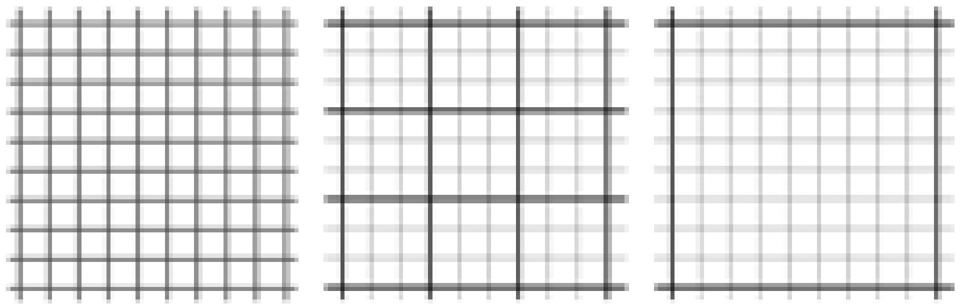


Figure 3.15: Scan resolution can be dynamically varied. Left: all active lines are scanned; Middle: every third active line is scanned, other lines effectively become drone conductors (gray); Right: sensor is scanned at the lowest possible resolution, effectively turning it into a single-touch sensor.

property which allows a dynamic trade-off between spatial resolution on the one hand, and faster scan rate and lower power consumption on the other.

Spatial resolution can be lowered simply by disconnecting sets of column or row electrodes, effectively turning them into drone conductors (this can be done using electronic logic that has a high-impedance mode). For instance, if we disconnect every other column and row electrode, the grid resolution is effectively reduced by a factor of two, and the scan rate goes up by a factor of four. Taking this further, if every column and row electrode except the first and last is disconnected, the sensor acts as a single bilinear cell which can only measure the centroid and sum of pressure exerted over the entire sensor surface (see Figure 3.15).

It is also possible to adaptively scan the sensor with finer detail only in areas where contact is made or where fine detail is required. This allows for the best of both worlds – providing high resolutions in areas where there is contact, while providing high speed and low power usage over areas with no contact.

This variable resolution property allows for the scanning of the sensor at many thousands of frames per second in order to detect very short duration impacts. It also permits a “sleep mode”, whereby battery-powered devices that need to conserve power can idle without drawing significant power as they wait for a touch event to awaken them. Finally, it permits us to build large, high resolution sensors without significantly impacting electronics cost.

3.4.3 Power Consumption

The shift registers mentioned above are used because they are capable of sourcing/sinking much more current than the micro-controller – up to 35mA. We found that the sensor could draw instantaneous currents of as much as 30mA when scanning a row/column intersection where the applied force was over 5kg. However, because 1200 points are scanned every frame, even when a great deal of force is applied to one point on the sensor, the average current draw during the span of an entire frame remains very low. We have found that during full speed operation, the sensor and shift registers which drive it typically consume less than 1mA.

Surprisingly, the bulk of the current used by the UnMousePad, as much as 80mA, is actually drawn by our micro-controller. In the future, this can be greatly improved by using a more efficient micro-controller, and optimizing the micro-controller’s code to allow it to go into low-power energy saving states during A/D conversion and when the sensor is not being touched. Even with the current power-hungry micro-controller, the total power consumption of the UnMousePad is low enough that it can be powered entirely from the USB bus.

3.4.4 Sensitivity

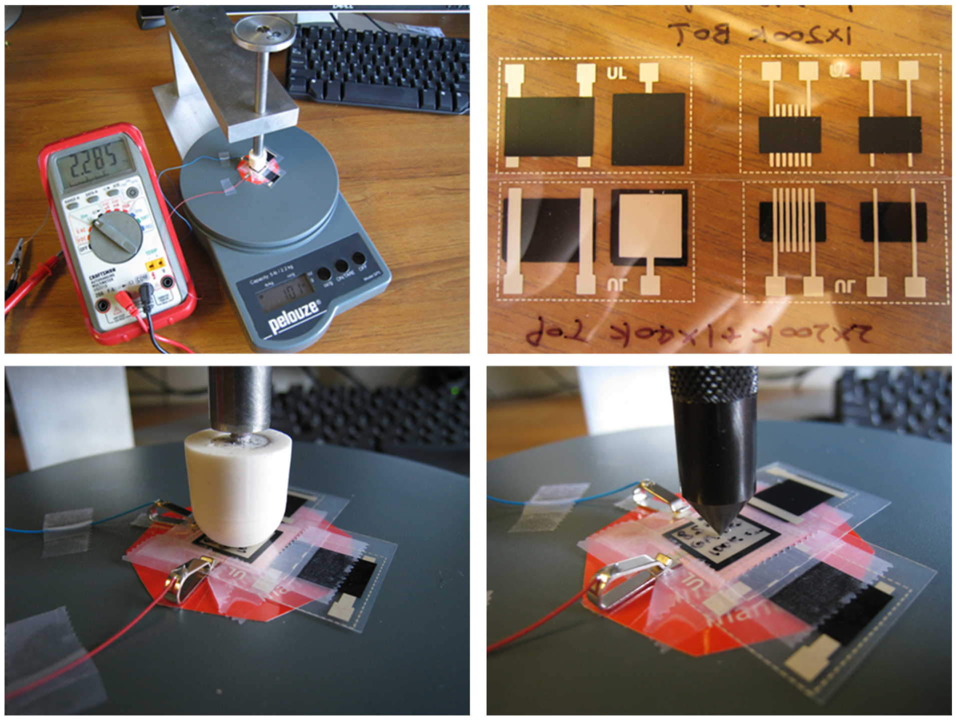


Figure 3.16: Sensitivity testing. Upper Left: setup showing sensor on scale, plunger, and ohm-meter connected to sensor; Upper Right: closeup of test areas cut from sensors; Lower Left: finger-shaped silicone rubber plunger; Lower Right: plunger fitted with pen-tip

Using a calibrated force gauge and weights (Figure 3.16), we found that the lightest force the UnMousePad can reliably detect is 5 grams, and the lightest touch that can be reliably tracked as it moves over the UnMousePad is in the range of 15-30 grams. By comparison, the force needed to press down a typical computer keyboard key is about 65 grams, and for a typical mouse button, is about 90 grams.

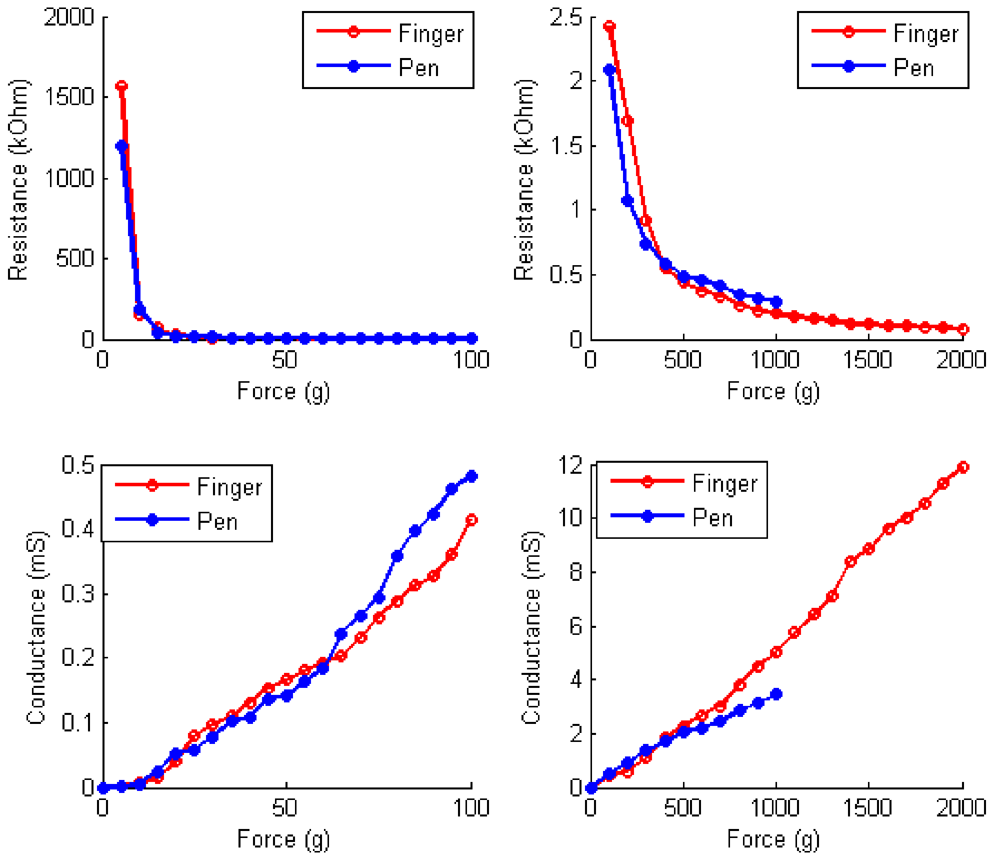


Figure 3.17: Force vs resistance (top) and force vs conductance (bottom) at various force ranges (0 to 100 grams on the left, and 0 to 2000 grams on the right) for a point touch and an area touch.

We also found that our sensor has a near-linear electrical conductance response to forces between 5 grams and 2 kg (Figure 3.17). Furthermore, comparing the response of a finger and pen touch, we see that the sensor measures force accurately for touches with different contact areas. The strongest measurable touch on the UnMousePad is around 7 kg - however, we could only verify the accuracy of forces up to 2 kg due to limitations of our test equipment. We suspect that 2 kg of force

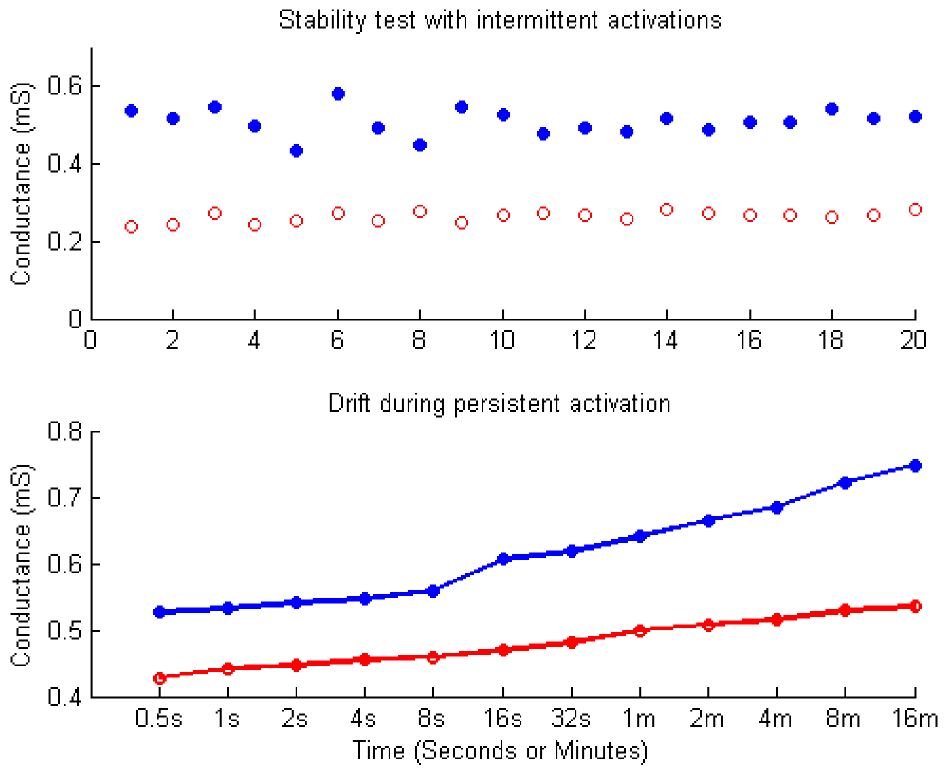


Figure 3.18: Sensitivity drift over time with intermittent activations at one-second intervals (top), and with persistent activation over a period of 16 minutes with a logarithmic time scale (bottom).

is a reasonable upper limit for most user interface applications, since we observed that pressing that hard with a finger was quite painful.

We also measured the repeatability of sensor force measurements (Figure 3.18). We found that with repeated activations, performed at one-second intervals, the measured force had a 5% standard deviation with very little drift. However, we found that with steady pressure at a single point, sensitivity increased over time. Although the increase was 5% over the first five seconds, we found that the rate of increase decayed exponentially with time. After removing the force and reapplying, the exact same pattern repeated. The authors suspect that this is due to the FSR surfaces intermeshing ever closer at a microscopic level over time, and that this effect can be compensated in software. In practice, we found that this small variation in force sensitivity over time was not perceivable by human operators.

3.4.5 Resolution

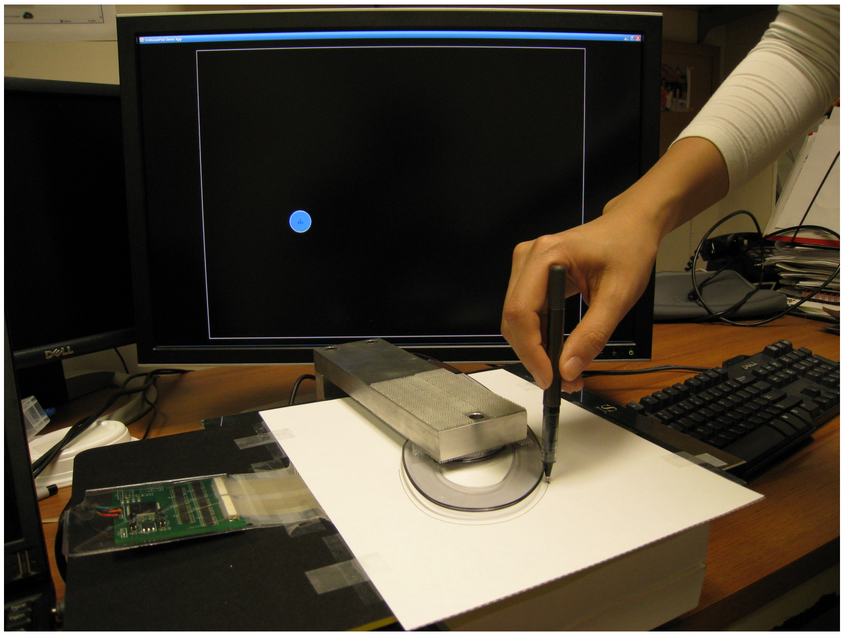


Figure 3.19: Resolution test setup. UnMousePad captures a curve as it's drawn on a sheet of paper attached to the top of the UnMousePad.

To experimentally determine the point-tracking resolution of the UnMousePad, we record the (X, Y) positions of our point tracking algorithm over time, as a curve is drawn on a sheet of paper placed over the UnMousePad. A "French Curve"

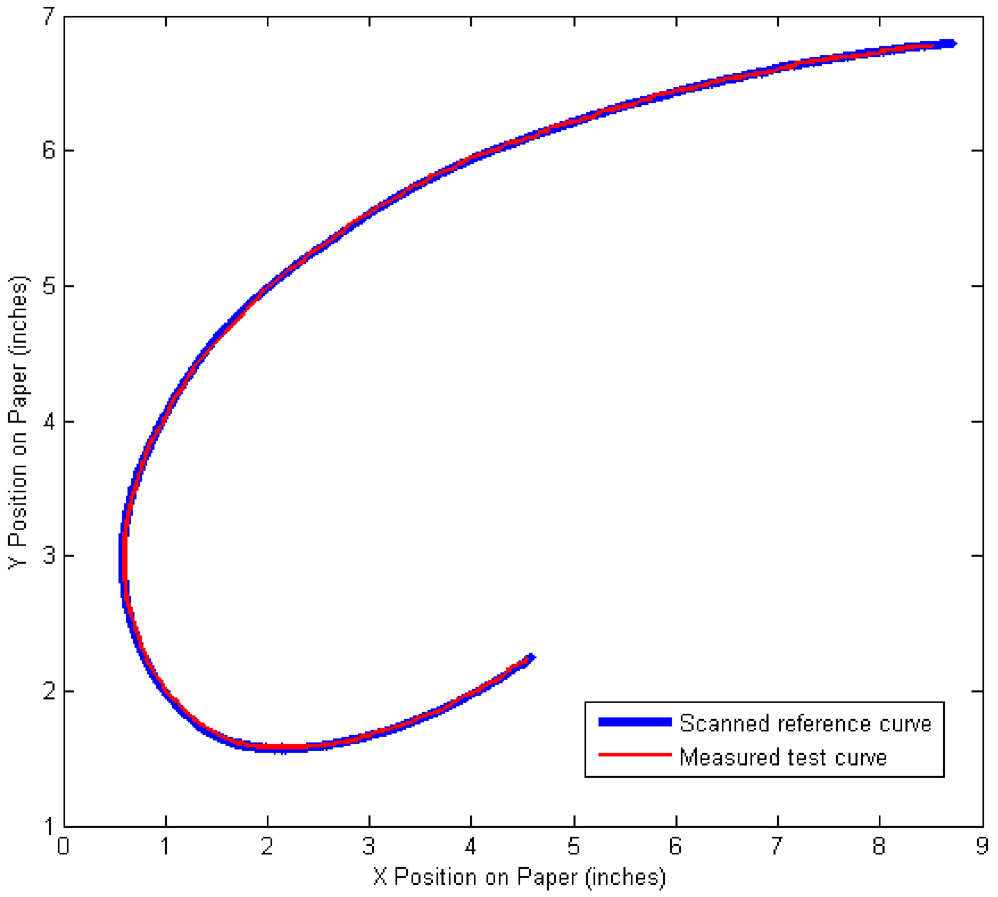


Figure 3.20: Resolution test result. Curve captured by the sensor is overlaid onto the thinned reference curve scanned from the paper.

mounted 1/4" above the sensors is used as a guide. Next, we scan the reference curve drawn on the paper with an optical scanner at 600dpi and use a thinning algorithm to find the line passing through the center of the curve. Finally, the curves are registered and the error is computed as the root mean square of the distance from each point in the test curve to the closest point on the reference curve.

Using this procedure, we determined that the UnMousePad can track a fine point such as a pen-tip with a standard deviation of 0.2934 mm (about 0.0115 inches, or 87 dots per inch). Note that this result was achieved with raw data, without applying any filtering, smoothing, or calibration.

3.5 Data Processing and Touch Tracking

For most graphics and HCI applications of the UnMousePad, rather than using raw images of pressure, it is often useful to work with a higher-level representation of the user's interaction with the device. In our driver-level code, we process the raw images to find touches. These touches are then tracked from frame to frame. Each touch consists of a unique identifier, an (x, y) position, the total force of the touch and the shape of the oval that encircles the touch. It also carries lower level data such as the pixels that comprise the touch.

A list of touches is provided to applications, as are change events such as "touch up", "touch down" and "touch moved". Furthermore, we can detect when two touches come so close together that they are no longer distinguishable (which can happen when a user pinches two fingers very close together) and when a single touch splits into two (the opposite of the first event). We call these two events "touch merged" and "touch split" respectively.

In order to get the full benefit of the UnMousePad's pressure imaging capability, the data must be properly processed. This processing consists of several stages of filters. Many of these filters operate on 2D images of force using standard image processing techniques. Because of the interpolating nature of the device, these

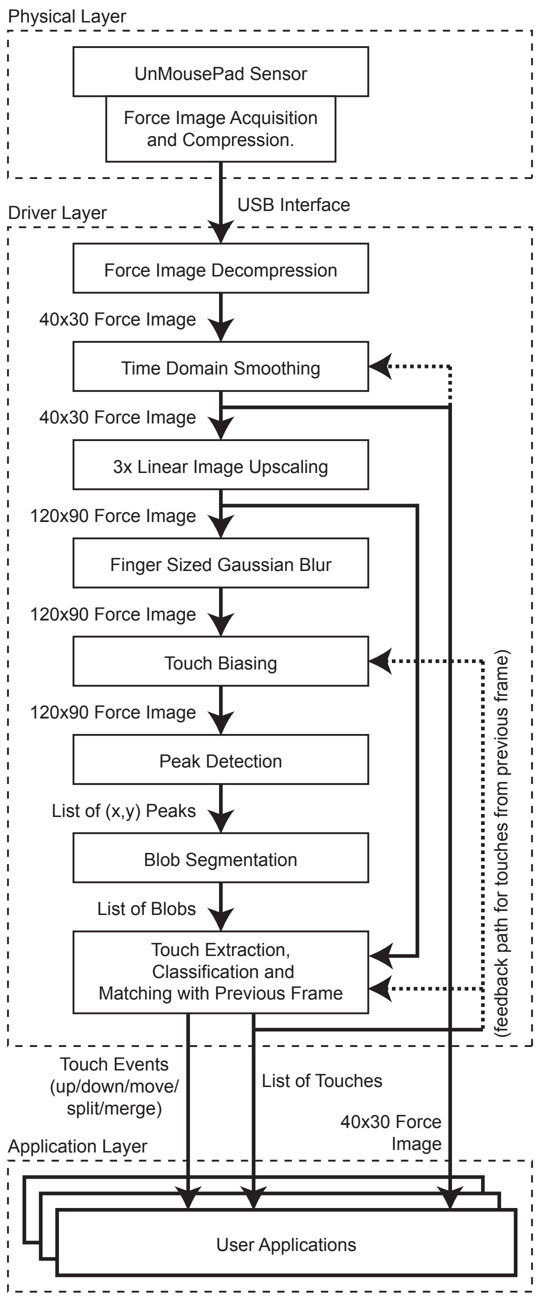


Figure 3.21: Illustration of data flow from sensor to application.

images are inherently low-resolution and can be processed efficiently in real-time on a micro-controller or a CPU using a small amount of processing power. The processing consists of the following steps:

1. *Force Image Acquisition and Compression:* Micro-controller captures a frame of data, and compresses it for transmission over a USB bus.

2. *Force Image Decompression:* Data is decompressed into a 40x30 image and normalized to grams.

3. *Time Domain Smoothing:* Noise is reduced by averaging current frame with last smoothed frame.

4. *Force Image Upscaling:* To enable accurate segmentation into touch events, upscale resolution by 3x. Reduce forces correspondingly by factor of 9.

5. *Gaussian Blur:* To improve finger tracking and remove singularities introduced by linear upscaling, perform a gaussian blur approximately the radius of a finger touch.

6. *Touch Biasing:* When force is near minimum touch threshold, or when touches merge or split, there can be oscillation between touch/not-touch in successive frames. To avoid these instabilities, bias blurred image by adding a small gaussian-shaped force signature wherever touches were detected in previous frame.

7. *Peak Detection:* Each pixel of biased image is compared against neighbors. Mark any pixel with a higher force than all neighbors as a peak.

8. *Blob Segmentation:* Segment image into areas around each peak by seeding each blob with peak location, then iteratively expanding each blob to encompass neighboring pixels with lower force value than pixels in blob. Stop growing at zero-valued pixels or when blob reaches a pre-set max size.

9. *Touch Extraction:* Compute position, size, total force and surrounding oval for each blob (using both blob data and unblurred image). Position is computed as the force-weighted average of the positions of pixels within each blob.

10. *Touch Classification:* Classify each touch as either *(i)* noise if below minimum touch force threshold, *(ii)* point touch (stylus) if ratio of force to area is above a predefined threshold or *(iii)* area touch (finger, palm or object) if ratio of force to area is below threshold.

11. *Matching with Previous Frame:* Match up touches with those in previous frame having corresponding position and size, giving each touch the same unique id as in previous frame. First time touches generate touch down events, and touches that disappear generate touch up events. When two touches of force a and b merge to a touch of force $a + b$, generate a merge event; vice versa for a split event.

12. *User Applications:* Send pressure images and touch events to user applications.

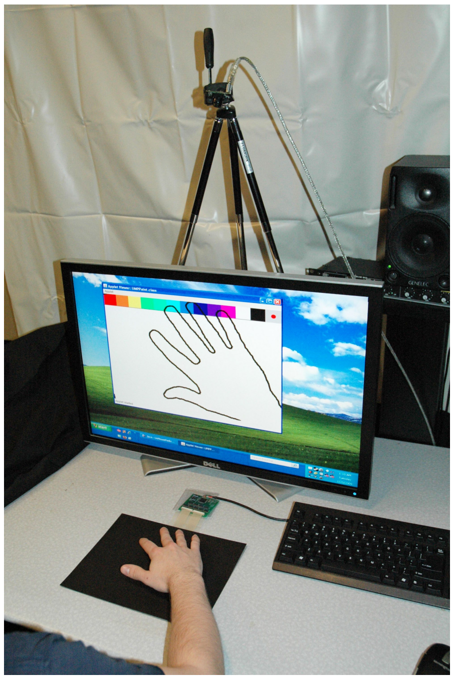


Figure 3.22: Setup for camera based hand visualization. The camera is mounted on a tripod above the IFSR sensor.

3.5.1 Camera-Based Hand Visualization

To enable tracking of the hand above the UnMousePad surface, we mount a Firefly MV digital video camera atop the computer monitor, pointing downward at the UnMousePad (Figure 3.22). The Firefly captures 320x240 images at 60fps. The UnMousePad is covered with black paper, so the user's hand will stand out in clear contrast. Infrared lighting and a camera which sees in IR can be used to reduce variations due to skin tone. We perform a small-radius Gaussian blur to remove any speckle, and then extract fragments of the silhouette edge with Marching Squares. A perspective correction matrix is applied to the two endpoints of each

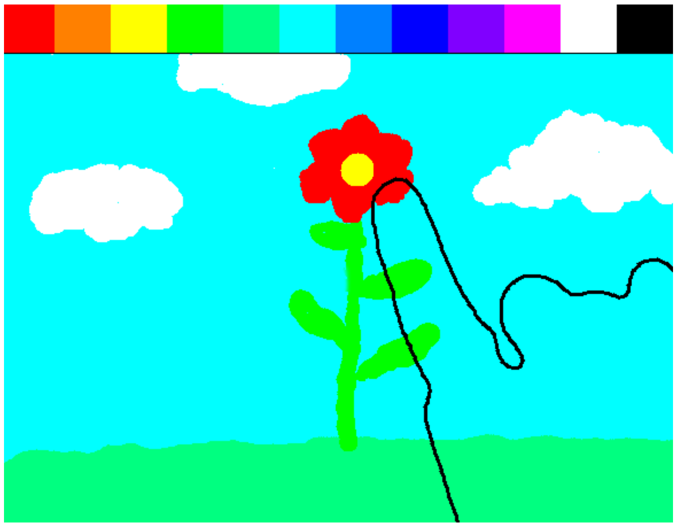


Figure 3.23: User's view of their own hand overlaid on application.

edge fragment so that the four corners of the UnMousePad are mapped to the four corners of the application window. The fragments are then displayed as thick black vector lines overlaid upon the application. To the user it appears as though their own hand is visible as a real-time outline on the computer display (Figure 3.23).

In addition, the hand silhouette is analyzed to provide the application software with a persistent identifier for each finger of the user's hand. In this way, applications that use two or more fingers can maintain a persistent ID for each finger, even before any fingers have physically touched the UnMousePad. Fingertips are identified by finding matches within the silhouette shape for circular arcs that are approximately one finger width in diameter. This is done by first initializing a weight image to zero. Then for each silhouette edge fragment, this weight image is incremented at locations perpendicular to the fragment at a distance of half a finger width. Centers of fingertips are where the highest weights are accumulated.

We found that silhouettes are extremely powerful because they enable hover and positioning of the hand in LOTUS applications, serving a similar purpose as the mouse cursor in traditional WIMP interfaces. In future work, we would like to replace the 2D camera with a 3D stereoscopic camera, such at that described in Chapter 1, which will enable even richer interactivity by combining in-air 3D gestures with precise surface interaction.

3.6 Applications

Because of IFSR's low cost, precision and ability to sense pressure, it is a versatile technology that can be used for a variety of applications that are not possible or practical with other technologies:

3.6.1 Bendable Sensors

Sensors based on IFSR technology are physically bendable. Their bending radius depends upon which material is used for the plastic – thinner is more bendable. We currently use either 3 mil or 5 mil thick plastic substrates, which permit bending radii of approximately 0.5" and 1.0", respectively. We have taken advantage of this flexibility to create applications that call for a sensor to be wrapped around a cylindrical object such as a mug (Figure 3.24). Uses such as wrapping the sensors around pens or around steering wheels have also been proposed. An interesting related feature of IFSRs is that they can be used as bend sensors.

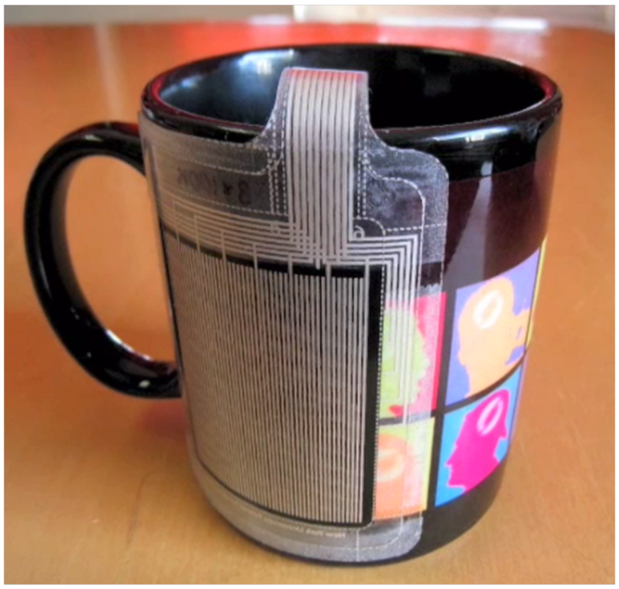


Figure 3.24: IFSR Sensor wrapped around a mug.

3.6.2 Touch on Back

We have created IFSR sensors that are extremely thin and compact enough to be wrapped around the back of electronic devices (Figure 3.25). Working with the creators of the NanoTouch [2] concept, we incorporated one of our sensors onto the back of a small device which has an OLED display on the front, to prototype what force sensitive touch interaction would feel like if used on the back of a small device such as a watch (Figure 3.26).

On such devices, screen real-estate is often very limited – thus, touching on the front of the device can be impractical. Because IFSR is force sensitive and extremely accurate, the user is able to use a light touch to move a cursor, and a harder touch to select items on the screen. A visualization, such as a circle

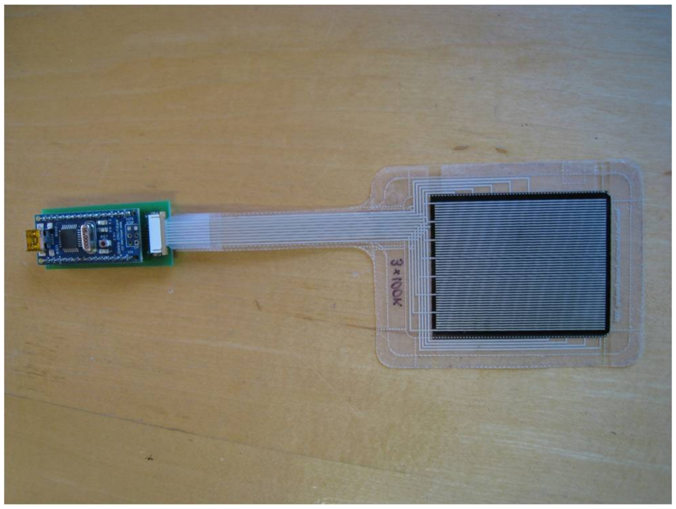


Figure 3.25: A compact and power efficient IFSR sensor for use on the back of small electronic devices. This sensor can also be used to replace computer track-pads.

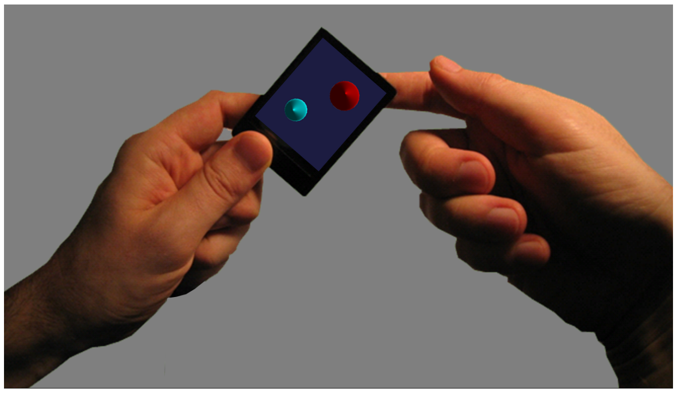


Figure 3.26: A concept of a device with touch-on-back interaction. Note how the position and force applied by the fingers is visualized on the display at the front of the device.

which changes color or size corresponding to pressure or a simulated deformation of the screen image, gives the user feedback on the amount of force applied and the position of their finger relative to on-screen items.

3.6.3 Medical and Industrial Pressure Imaging

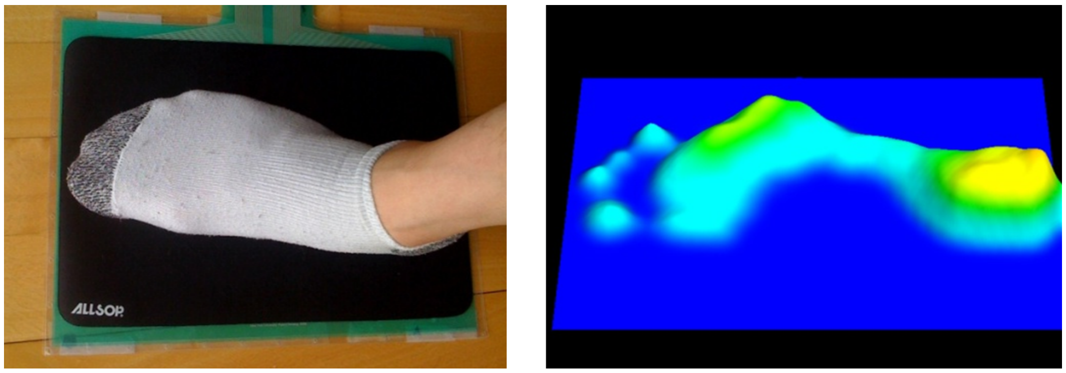


Figure 3.27: An IFSR sensor is used to measure the pressure distribution of a foot.

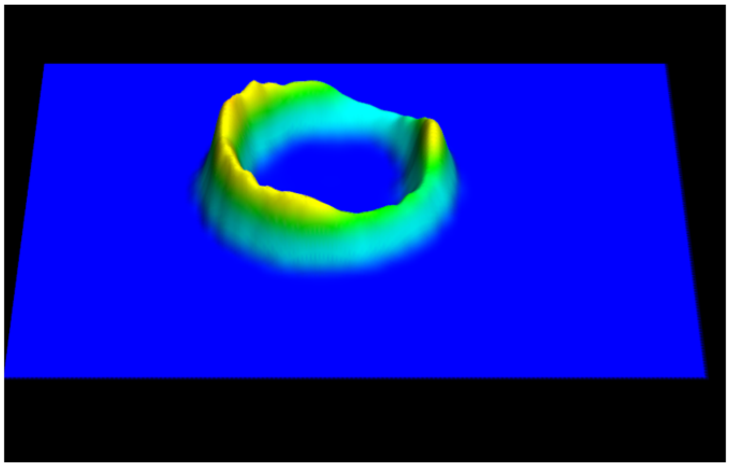


Figure 3.28: An IFSR sensor can be used to measure the pressure distribution of a solid object. In this case, the pressure distribution of a mug is visualized.

Sensors based on IFSR also have many applications in the medical field. For instance, they can be used to measure the pressure distributions of people's feet for the creation of custom orthotic inserts (Figure 3.27).

They can also be used to measure the shape and/or flatness of objects in manufacturing and industrial processes (Figure 3.28). One potential use for IFSR would be in the manufacturing of IFSR sensors themselves. The printing process for IFSR sensors involves the use of an automated screen-printing press, which uses a squeegee to push ink through a mesh. Without IFSR, there is no easy way to measure the pressure applied by the squeegee as it travels over the print surface. Using an IFSR placed below the object being printed to measure the distribution of squeegee pressure can improve the consistency of the screen-printing process, leading to more controlled manufacture of printed electronics such as IFSR sensors.

3.6.4 Sensing Through Paper and Flexible Displays

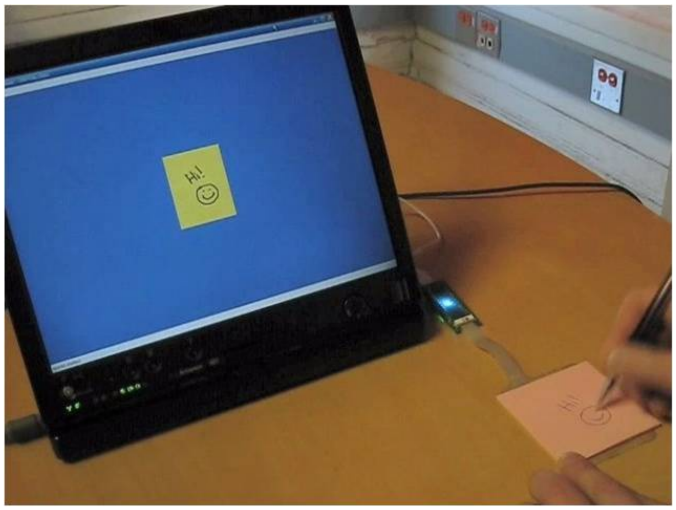


Figure 3.29: IFSR can track a stylus through several layers of paper.

We have found that the IFSR sensors can sense touches quite well through one or more sheets of paper. For example, when the UnMousePad is placed behind a pad of paper, the user can draw, write or scribble on the top sheet of the pad, and we can use the information that comes through to the UnMousePad to reconstruct the drawn image (Figure 3.29).

Similarly, the UnMousePad can read quite well through emerging thin flexible display technologies such as flexible OLED and e-ink. Because such displays are often manufactured on substrates similar to those used for the IFSR, our technology can readily pick up pressure through them. This allows for the addition of touch sensing without any loss of display brightness or contrast.

3.6.5 Touch Screens

In addition to opaque IFSR sensors, we have developed sensors with transparent FSR materials. Opaque FSR inks are typically composed of a carbon-infused polymer, whereas transparent FSR inks are made of a clear polymer infused with a transparent conductor. We have found transparent IFSR sensors to be well-suited for use over LCD displays.

A 24" diagonal transparent IFSR on top of an LCD display was demonstrated at SIGGRAPH Emerging Technologies in 2009. It was used to enable a multi-user multi-touch pressure sensitive drawing application (Figure 3.30). In this application, users could simultaneously use their left hand to switch the drawing color and drawing tool, and to control pan and zoom while using their right hand to draw on a canvass.

The conductive electrode lines can either be transparent or opaque. Transparent conductors are typically made of ITO (indium tin oxide). Unfortunately, due

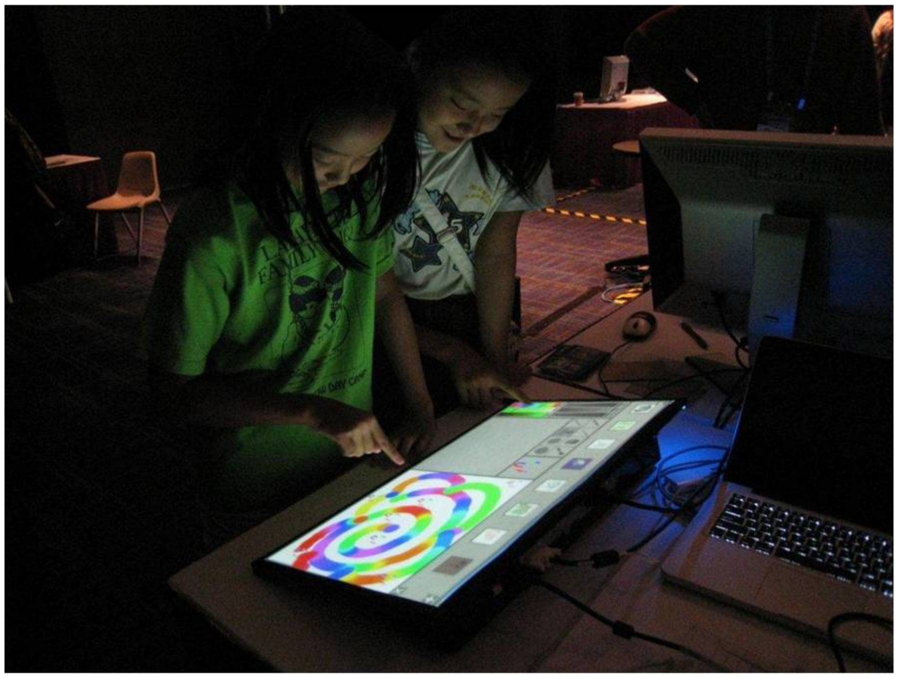


Figure 3.30: A 24” diagonal transparent IFSR prototype demonstrated at SIGGRAPH Emerging Technologies in 2009.

to the high resistance of ITO, IFSR sensors with transparent conductors cannot be made very large. Opaque conductors, made of silver, copper, aluminum, and even carbon nano-tubes, have much better conductance than any known transparent conductor, and can be made so thin as to be virtually invisible. They can also be patterned and/or aligned with the pixels of an LCD screen to make them even less visible. This was the approach we used in the 24” diagonal sensor prototypes.

We have also found that thin (6 mil) glass is suitable as a substrate for IFSR sensors, with no appreciable loss of spatial resolution. Glass can be advantageous over plastic for transparent touch-screen applications, since many people prefer the feel of glass.

3.6.6 Musical Instruments

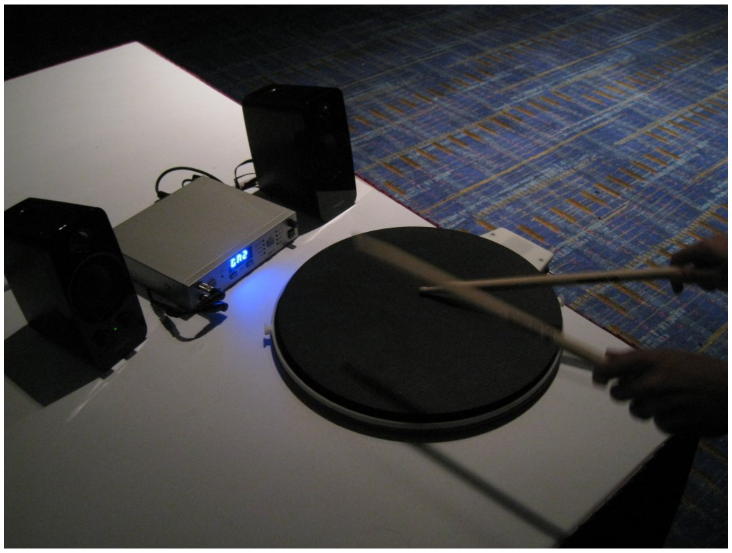


Figure 3.31: Drumming on a 13” diameter IFSR sensor. The MIDI sound synthesizer and speakers can be seen in the background.

Multi-touch position sensing, along with pressure sensitivity and the ability to scan at high frame rates make IFSR sensors ideal for musical instruments. We have used IFSR to build instruments such as pianos and drums (Figures 3.31).

To create the drum, we developed a circular sensor design that uses curved row and column electrodes (Figure 3.32). In this sensor, we chose to sacrifice multi-touch resolution to gain scanning speed. This allowed us to scan the sensor at approximately 1000Hz, which is essential for drumming. Instead of processing the signal with a computer, to avoid latency, we sent the output via MIDI directly to a sound synthesizer. The position on the drum-pad was used to vary pitch and timbre, creating a very rich and lifelike sound.

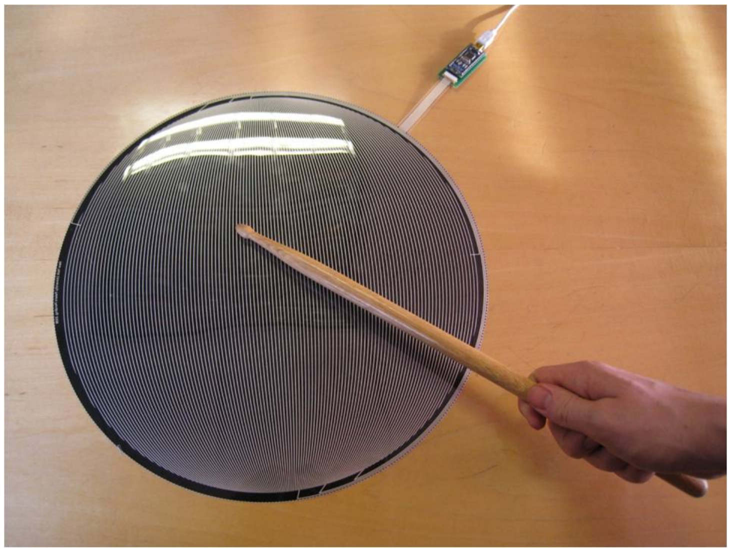


Figure 3.32: The drum sensor employs curved column and row electrodes.

IFSR was also used by the well known instrument designer Roger Linn to develop a musical array instrument called the LinnStrument [23].

3.6.7 Games

We have explored the use of IFSR sensors in gaming applications. In one of these applications, we used an UnMousePad as a giant force-sensitive track-pad to control a multi-user multi-touch Tetris game, which we affectionately called Touchtris (Figure 3.33). In this game, Touchtris pieces swirled in from the outside of the screen, as if they were falling into a black hole. Users had to work together to align the pieces and make them disappear before they filled up the whole screen. In the spirit of the original Tetris game, whenever users created a block that was 4x4 or bigger, it would be eliminated.

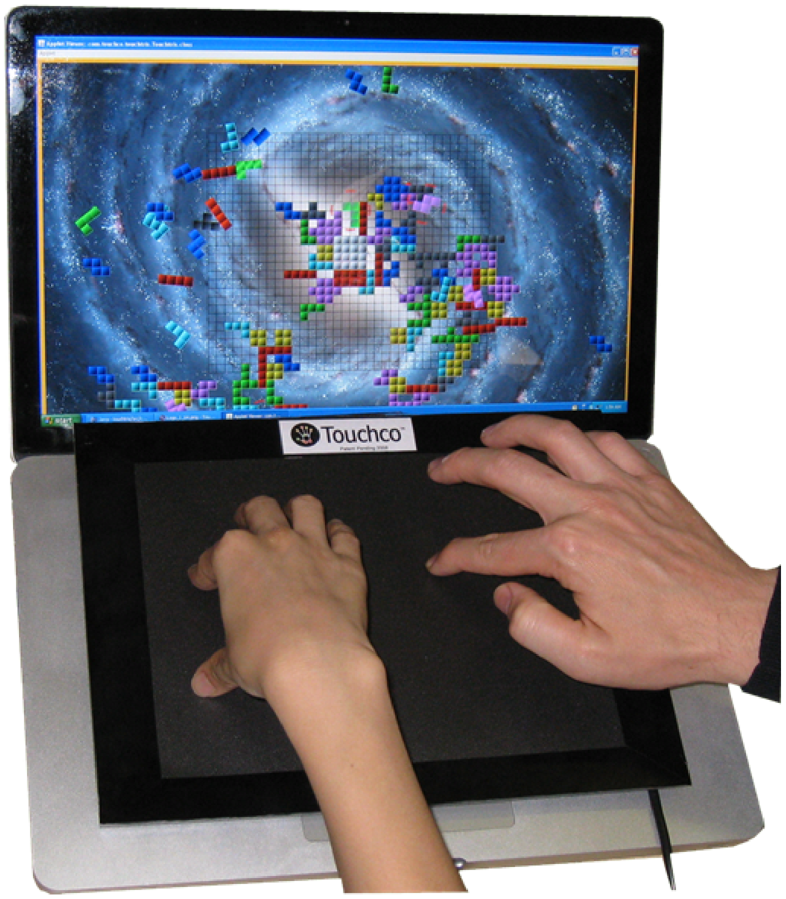


Figure 3.33: Here, two users can be seen using an UnMousePad to cooperatively play our Touchtris game, which is a 2D multi-touch analogue of Tetris.

In this game, the IFSR was used in the LOTUS configuration. When a user touched the UnMousePad lightly, a circle would show up on screen to indicate the position of their touch. Whenever force above a threshold was applied, the touch would engage with a Touchtris piece below it, allowing the user to translate and rotate the piece into place.

3.6.8 Design and Simulation

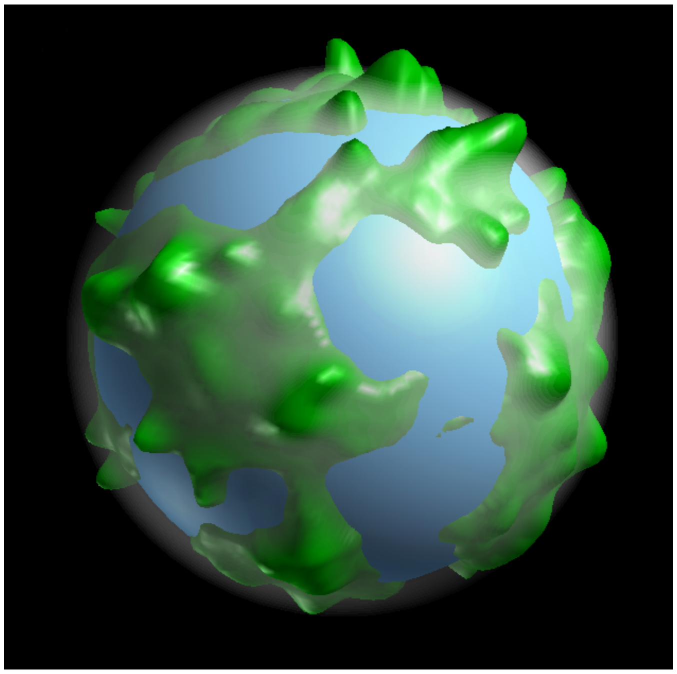


Figure 3.34: An application used to model 3D planets using an UnMousePad which we call WorldSculpt.

IFSR sensors are ideal for design and simulation applications which require rich multi-finger, multi-dimensional input. One of the design applications we have prototype is WorldSculpt (Figure 3.34), an application which allows a user to sculpt a planet with their hands. Users can simultaneously use their left hand to rotate the planet and select sculpting tools while using their right hand to sculpt. Sculpting tools include operations such as “push”, “pull” and “smooth”. IFSR is also an ideal input device for more complex sculpting tools such as the particle-based sculpting software presented in Chapter 1.

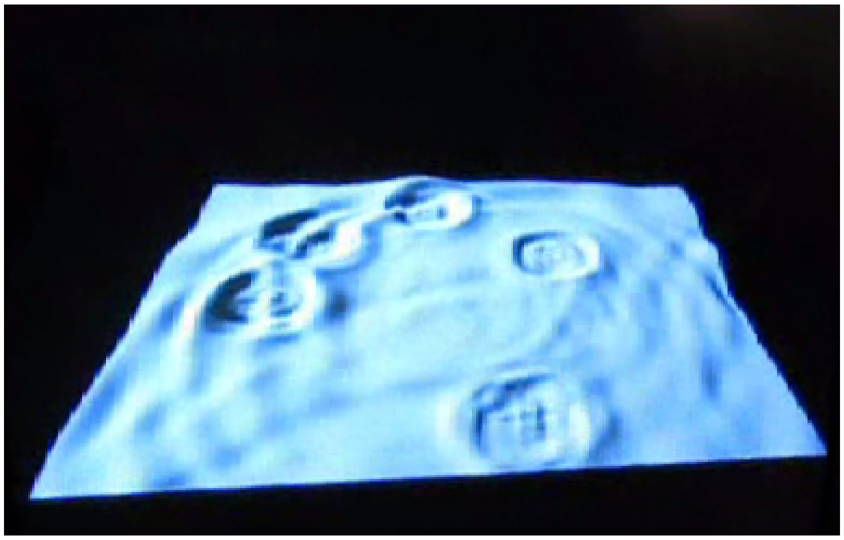


Figure 3.35: Users can interact with a water simulation in real-time using an UnMousePad.

We have also experimented with the use of an UnMousePad in simulation applications. For example, we have crated a water simulation where users can swipe, tap and push on the surface of the UnMousePad to apply forces to a virtual pool of water (Figure 3.35).

This type of user input device can be very useful for engineers for naturally inputting forces, velocities or other physical vector-based phenomena into a simulated system. For example, an aerospace engineer could use this to see what happens when air-flow disturbances, that she crates with her hands, interact with a simulated aircraft. This technology can also be useful for teaching physics to students. In fact, even our simple water simulation was able to attract hours of attention from curious students who used it to create waves and to see what happens when they crash into each other.

3.6.9 New Gestures

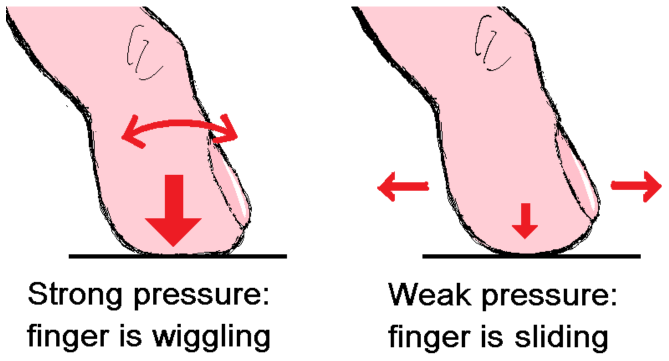


Figure 3.36: A heavy finger wiggles but a light finger slides.

The availability of fine resolution in pressure makes it possible to distinguish between gestures in interesting ways. This property distinguishes IFSR from capacitive input sensors such as the iPhone's screen which only measures contact area. Fine pressure resolution is needed to correctly interpret many pressure-dependent gestures. For example, lateral movement of a finger that is pressing down indicates wiggling, whereas the same lateral movement of a lightly touching finger indicates displacement by sliding (Figure 3.36).

We have also used pressure in a 3D manipulation task to distinguish between translation and scaling operations (light pressure) and rotation (heavy pressure). To complete the task, users had to align a series of 3D tea-cups with gray-colored teacup outlines (Figure 3.37). Users of this application reported that they found this new mode of interaction to be more intuitive and easier than using a separate modal input to switch between these operations.

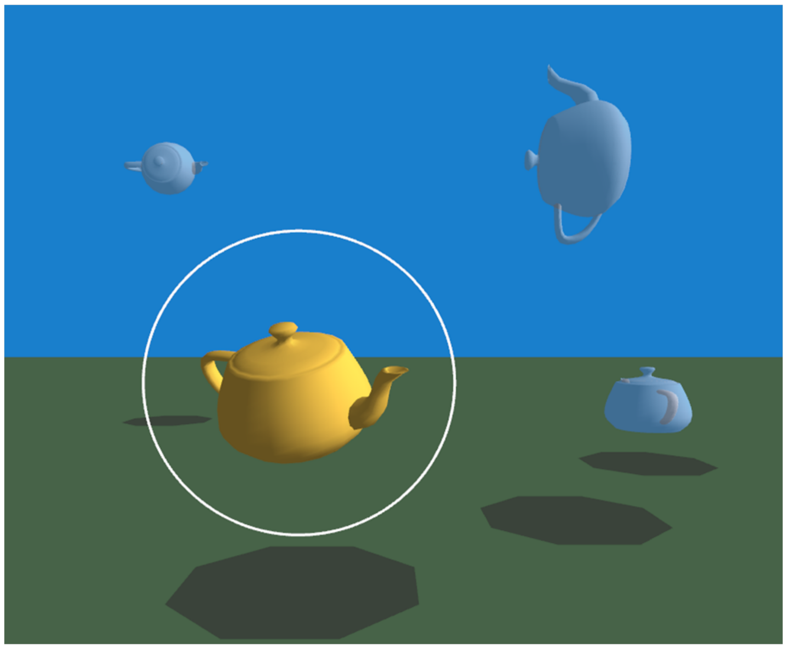


Figure 3.37: Manipulation of objects in 3D is intuitive with an UnMousePad. Multi-finger motion is used to move the tea-cup objects in the plane of the screen; pinching and spreading fingers move the objects in and out of the screen, and applying pressure switches into a rotation mode.

3.6.10 Isometric Control

Fine pressure resolution also permits highly accurate isometric control. For example, we have found that users of the UnMousePad quickly become proficient at manipulating virtual 3D objects with seven degrees of freedom by placing their fingers upon the pad to translate, rotate and scale an object. In another application we have used isometric pressure of fingers on the UnMousePad to simultaneously manipulate the eight faders of a BCF2000 MIDI controller, mapping pressure directly to fader settings (Figure 3.38).

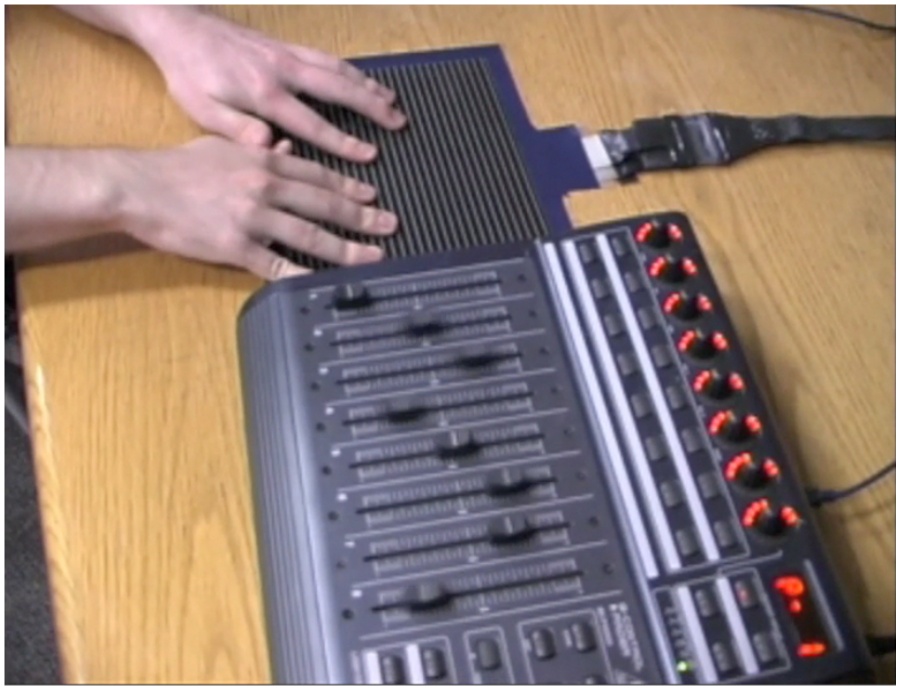


Figure 3.38: Simultaneous isometric control of 8 MIDI faders.

In all of the above examples, the control scheme can be effective only because we are using an input device capable of simultaneously tracking all finger touches with fine pressure resolution. These and other examples of unique applications of our device can be seen in the supplemental video to our SIGGRAPH paper [38].

3.7 Conclusion and Future Work

In later work, we planned to improve our driver-level algorithms to add useful features such as the ability to automatically distinguish the pressure profiles of palms, wrists, the side of the hand, and other objects, and to move more of the processing into the micro-controller to enable applications that don't require a host PC. We also planned to improve our transparent sensors for use over LCD screens, and to integrate the UnMousePad drivers with Microsoft Windows, which recognizes multi-touch devices. We also began researching combining IFSR with various forms of active haptic response, notably arrays of piezoelectric transducers.

We also planned to explore a variety of form factors, including flexible inserts for shoes and clothing, non-flat IFSR skins for such devices as game controllers, tennis racket grips and robots, as well as continuously extendable IFSR coverings for floors, tables and walls. We are particularly interested in capturing the subtle dynamics of human foot placement, as this capability opens up a wide range of applications, from dance to physical therapy. Furthermore, we believe that the core principle of the IFSR device – the anti-aliasing of physical phenomena before conversion to digital signals – is an exciting principle applicable to the transduction of sound, electro-magnetic waves and many other types of signals.

Unfortunately, due to the acquisition of Touchco, we do not know when, if ever, this further research will take place. We hope that one day, the original vision of Touchco will be realized and that IFSR will become available for use in these and many other applications that we haven't even thought of yet.

In conclusion, IFSR technology has the potential to revolutionize the field of multi-touch interaction by providing a low-cost, high-quality, flexible pressure-imaging device. As the availability of IFSR devices increases, and the cost of manufacturing flexible electronics continues to drop, we anticipate that they will find a myriad of new uses, and will come to play an important role in many aspects of everyday life.

Conclusion

Today, we are gradually moving into a world where digital technologies replace familiar physical implements. Some notable examples include tablets replacing pen and paper, the combination of CAD tools, 3D printing and other automated manufacturing methods replacing traditional design and craftsmanship, and computer games replacing play in the real world. As this trend progresses, users expect these digital technologies to offer high-fidelity interfaces and to behave in natural ways similar to the physical interactions that they replace.

This dissertation has presented three novel components. Chapter one presented a technique for generating 3D depth maps in real time from a stereo camera, and showed some applications including robot navigation, simulated camera effects, and tracking the human body for user interaction tasks. The second chapter presented a set of algorithms for real-time rendering of particle-based surfaces and showed applications including interactive fluids, virtual clay and procedurally animated characters based on particles. The third chapter presented a multi-touch force-sensitive surface based on interpolating force sensing resistance and showed applications such as 3D manipulation, virtual instruments, writing, and interaction with a simulated water surface.

These components can be thought of as pieces to a puzzle for realizing a larger vision, which combines 3D tracking of the user in space, responsive high-resolution touch surfaces, and software tools that mimic the real world to create immersive virtual environments. Seamless integration of these hardware and software components will allow users to naturally interact with a virtual world. The combination of these technologies will enable new and improved applications in the realms of work and play in areas such as art, music, design, modeling, scientific visualization, simulation and video games.

Although this thesis is by no means the last word in any of these three areas, I believe that my work has helped to illuminate a path for others to a world where user interaction technologies are more responsive and more accurate, fully capturing the speed, richness and subtlety of human touch and expression, and where software enables and encourages real-time interaction and fluid expression of the user's vision in whatever task they seek to accomplish in our increasingly digital world.

Bibliography

[1] Bart Adams, Toon Lenaerts, and Philip Dutré. Particle splatting: Interactive rendering of particle-based simulation data. *Technical Report CW 453, Katholieke Universiteit Leuven*, 2006.

[2] Patrick Baudisch and Gerry Chu. Back-of-device interaction allows creating very small touch devices. In *Proceedings of the 27th International Conference on Human Factors in Computing Systems*, pages 1923–1932. ACM, 2009.

[3] Stan Birchfield and Carlo Tomasi. Depth discontinuities by pixel-to-pixel stereo. In *ICCV*, pages 1073–1080, 1998.

[4] James F. Blinn. A generalization of algebraic surface drawing. *Computer Graphics and Interactive Techniques*, 1(3):235–256, 1982.

[5] Jules Bloomenthal. Polygonization of implicit surfaces. *Computer Aided Geometric Design*, 5(4):341–355, 1988.

[6] William Buxton, Ralph Hill, and Peter Rowley. Issues and techniques in touch-sensitive tablet input. *Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, pages 215–224, 1985.

[7] Frank John Clement and Barton Leigh Richardson. Coordinate detection system. *U.S. Patent No. 3764813*, 1973.

[8] Microsoft Corporation. Pixelsense. 2011. `http://www.microsoft.com/en-us/pixelsense/pixelsense.aspx`.

[9] Valve Corporation. Portal 2. 2011. `http://www.thinkwithportals.com`.

[10] Philip L Davidson and Jefferson Y Han. Extending 2D object arrangement with pressure-sensitive layering cues. *Proceedings of the 21st Annual ACM Symposium on User Interface Software and Technology*, pages 87–90, 2008.

[11] Isabelo de los Reyes, Nathanael Roberton, Brian Calvery, Timothy J. E. Turner, Adrian Chandley, Daniel Makoski, Paul Henderson, Egor Nikitin, Tarek Elabbady, and Phillip Joe. Function oriented user interface. *U.S. Patent Applications Publication No. US 2007/0198926 A1*, 2007.

[12] Willem den Boer and Adiel Abileah. Integrated light sensitive liquid crystal display. *U.S. Patent Application No. US20070109239 A1*, 2005.

[13] Paul Dietz and Darren Leigh. Diamondtouch: a multi-user touch technology. *Proceedings of the 14th Annual ACM Symposium on User Interface Software and Technology*, 2001.

[14] Ines Ernst and Heiko Hirschmüller. Mutual information based semi-global stereo matching on the GPU. In *Advances in Visual Computing*, pages I: 228–239, 2008.

[15] Franklin N Eventoff. Electronic pressure sensitive transducer apparatus. *U.S. Patent No. 4313227*, 1979.

[16] James Fung and Steve Mann. OpenVIDIA: Parallel GPU computer vision. In *Proceedings of the 13th ACM International Conference on Multimedia*, pages 849–852. ACM, 2005.

[17] Raia Hadsell, Pierre Sermanet, Ayse Erkan, Jan Ben, Jeff Han, Beat Flepp, Urs Muller, and Yann LeCun. On-line learning for offroad robots: Using spatial label propagation to learn long-range traversability. In *Proceedings of Robotics Science and Systems 07*, 2007.

[18] Jefferson Y Han. Low-cost multi-touch sensing through frustrated total internal reflection. *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology*, pages 115–118, 2005.

[19] Adrian Hilton and John Illingworth. Marching triangles: Delaunay implicit surface triangulation. *Technical Report CVSSP 01, University of Surrey*, 1997.

[20] Heiko Hirschmüller. Accurate and efficient stereo processing by semi-global matching and mutual information. In *Proceedings of IEEE Conference on Computer Vision and Pattern Recognition, vol. 2*, pages 807–814. IEEE Computer Society, 2005.

[21] Steven P Jobs et al. Touch screen device, method, and graphical user interface for determining commands by applying heuristics. *U.S. Patent Application No. 20080122796*, 2008.

[22] Pascal Joguet and Guillaume Largillier. Devices and methods of controlling manipulation of virtual objects on a multi-contact tactile screen. *U.S. Patent Applications Publication No. US 2007/0198926 A1*, 2007.

[23] Roger Linn. Linnstrument. 2010. `http://www.rogerlinndesign.com/preview-linnstrument.html`.

[24] William E. Lorensen and Harvey E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics and Interactive Techniques*, 21(4):163–169, 1987.

[25] Charles F Malacaria. A thin, flexible, matrix-based pressure sensor. *Sensors Magazine.*, 1998.

[26] Inc. Microsoft. Kinect. 2010. `http://www.xbox.com/en-US/kinect`.

[27] Leap Motion. Leap motion controller. 2013. `http://www.leapmotion.com`.

[28] Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. *Symposium on Computer Animation*, pages 154–159, 2003.

[29] Matthias Müller, Richard Keiser, Andrew Nealen, Mark Pauly, Markus Gross, and Marc Alexa. Point based animation of elastic, plastic and melting objects. *Symposium on Computer Animation*, pages 141–151, 2004.

[30] Matthias Müller, Simon Schirm, and Matthias Teschner. Interactive blood simulation for virtual surgery based on smoothed particle hydrodynamics. *Technology and Health Care*, 12(1):25–31, 2004.

[31] Matthias Müller, Simon Schirm, Matthias Teschner, Bruno Heidelberger, and Markus H. Gross. Interaction of fluids with deformable solids. *Computer Animation and Virtual Worlds*, 15(34):159–171, 2004.

[32] Haim Perski and Meir Morag. Dual function input device and method. *U.S. Patent No. 6762752*, 2002.

[33] PointGrey. Triclops SDK. 2006. `http://www.ptgrey.com/products/triclopsSDK/triclops.pdf`.

[34] William T. Reeves. Particle systems – A technique for modeling a class of fuzzy objects. *Computer Graphics and Interactive Techniques*, 17:359–376, 1983.

[35] Jun Rekimoto. Smartskin: An infrastructure for freehand manipulation on interactive surfaces. In *Proceedings of the CHI 2002 Conference on Human Factors in Computing Systems*, pages 113–120. ACM, 2002.

[36] Ilya D. Rosenberg and Ken Birdwell. Real-time particle isosurface extraction. In *Proceedings of the 2008 symposium on Interactive 3D graphics and games*, pages 35–43. ACM, 2008.

[37] Ilya D. Rosenberg, Phillip L. Davidson, Casey M. R. Muller, and Jefferson Y. Han. Real-time stereo vision using semi-global matching on programmable graphics hardware. In *Proceedings of ACM SIGGRAPH 2006*. ACM, 2006.

[38] Ilya D. Rosenberg and Ken Perlin. The unmousepad: An interpolating multi-touch force-sensing input pad. *ACM Transactions on Graphics - Proceedings of ACM SIGGRAPH 2009*, 28(3), 2009.

[39] Daniel Scharstein and Richard Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *International Journal of Computer Vision*, 47:7–42, 2001. `http://vision.middlebury.edu/stereo/eval/`.

[40] Karl Sims. Particle animation and rendering using data parallel computation. *Computer Graphics and Interactive Techniques*, 24:405–413, 1990.

[41] Synaptics. Forcepad. 2012. `http://www.synaptics.com/solutions/products/forcepad`.

[42] Pressure Profile Systems. Capacitive tactile sensing. 2008. `http://www.pressureprofile.com/technology-capacitive.php`.

[43] Richard Szeliski and David Tonnesen. Surface modeling with oriented particle systems. *Computer Graphics and Interactive Techniques*, 26(2):185–194, 1992.

[44] Tactonic. Sensor technology. 2010. `http://www.tactonic.com/technology.html`.

[45] Matthias Teschner, Bruno Heidelberger, Matthias Müller, Danat Pomerantes, and Markus H. Gross. Optimized spatial hashing for collision detection of deformable objects. *Vision, Modeling, and Visualization*, pages 47–54, 2003.

[46] Adrian R. L. Travis, Timothy A. Large, Neil Emerton, and Steven Bathiche. Wedge optics in flat panel displays. *Proceedings of the IEEE*, 101(1):45–60, 2013.

[47] Frédéric Triquet, Philippe Meseure, and Christophe Chaillou. Fast polygonization of implicit surfaces. *WSCG (Plzen, Czech Republic)*, 2:283–290, 2001.

[48] David Wessel, Rimas Avizienis, Adrian Freed, and Matthew Wright. A force sensitive multi-touch array supporting multiple 2-D musical control structures. *New Interfaces for Musical Expression*, pages 41–45, 2007.

[49] Wayne Westerman. *Hand Tracking, Finger Identification and Chordic Manipulation on a Multi-Touch Surface.* PhD thesis, University of Delaware, 1999.

[50] Andrew P. Witkin and Paul S. Heckbert. Using particles to sample and control implicit surfaces. *Computer Graphics and Interactive Techniques*, 28:269–277, 1994.

[51] Brian Wyvill, Craig McPheeters, and Geoff Wyvill. Animating soft objects. *The Visual Computer*, 2(4):235–242, 1986.

[52] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for soft objects. *The Visual Computer*, 2(4):227–234, 1986.