GPU PROGRAMMING FOR REAL-TIME WATERCOLOR SIMULATION

A Thesis

by

JESSICA STACY SCOTT

Submitted to the Office of Graduate Studies of
Texas A&M University
in partial fulfillment of the requirements for the degree of

MASTER OF SCIENCE

December 2004

Major Subject: Visualization Sciences

GPU PROGRAMMING FOR REAL-TIME WATERCOLOR SIMULATION

A Thesis

by

JESSICA STACY SCOTT

Submitted to Texas A&M University
in partial fulfillment of the requirements
for the degree of

MASTER OF SCIENCE

Approved as to style and content by:

Donald House (Chair of Committee)	Frederic Parke (Member)
Jianer Chen (Member)	Phillip Tabb (Head of Department)

December 2004

Major Subject: Visualization Sciences

ABSTRACT

GPU Programming for Real-Time Watercolor Simulation. (December 2004)

Jessica Stacy Scott, B.A., Williams College

Chair of Advisory Committee: Dr. Donald House

This thesis presents a method for combining GPU programming with traditional programming to create a fluid simulation-based watercolor tool for artists. This application provides a graphical interface and a canvas upon which artists can create simulated watercolors in real time. The GPU, or Graphics Processing Unit, is an efficient and highly parallel processor located on the graphics card of a computer; GPU programming is touted as a way to improve performance in graphics and non-graphics applications. The effectiveness of this method in speeding up large, general-purpose programs, however, is found here to be disappointing. In a small application with minimal CPU/GPU interaction, theoretical speedups of 10 times may be achieved, but with the limitations of communication speed between the GPU and the CPU, gains are slight when this method is used in conjunction with traditional programming.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

CHAPTER I

INTRODUCTION

Watercolor is a beautiful and fascinating medium, which has been used by artists for hundreds of years. Watercolor paint, mixed with water and applied to paper, produces translucent layers which are quite visually different from the opaque layers obtained from other types of paint. The superposition of many such layers, called glazes, gives watercolor paintings their characteristic appearance. Working with watercolor provides a spontaneity unparalleled by other paint media; the sometimes unpredictable way in which pigment is transported through water and wet paper can provide the artist with delightful surprises. This unpredictability is also a source of difficulty in creating digital tools to emulate watercolor painting.

Enabling the creation of digital fine art is an important research direction in the field of computer graphics. There are many tools available to artists who wish to work digitally, many of which attempt to in some way mimic traditional artists' media. However, these tools sometimes do not behave in the way that the artist expects them to, leading to confusion and increasing the time that the artist must spend learning new software at the expense of time spent in creative endeavors.

Computer scientists, meanwhile, spend copious amounts of time in attempting to capture and recreate the physical processes of the real world. Physics-based modeling, especially given recent advances in computer hardware, has become a popular method of producing believable artistic effects in an automated manner. This technique is used in such varied settings as movies, games, scientific visualization, and engineering simulations. One type of physics-based modeling, fluid simulation, is applicable to

The journal model is *IEEE Transactions on Visualization & Computer Graphics.*

the problem of creating a watercolor tool.

The computational complexity of physics-based modeling has, in the past, limited its use primarily to applications which either do not run in real time or use simplified and inaccurate mathematical models. Previous work has been done with physics-based modeling for the simulation of traditional artists' media, but hardware limitations have limited the usefulness of the method. However, with recent hardware advances in computer graphics cards, a technique has become available for greatly increasing the speed of the computations required for physics simulations. The GPU, or Graphics Processing Unit, can handle many computations quickly and in parallel, which makes it ideal for applications which must run in real time but which are computationally expensive.

In this thesis, a combination of traditional and GPU programming is used to simulate watercolors for digital painters. Through physics-based fluid simulation, I have created a tool that behaves in a way which is believable and natural to artists already familiar with watercolor techniques.

CHAPTER II

BACKGROUND

The watercolor medium

Physically, watercolor paint consists of small particles of pigment mixed with water, binder, and surfactant; the binder and surfactant allow water and pigment to be absorbed into the paper [3]. Special watercolor paper is generally used with watercolors; this paper is highly textured and coated with cellulose called sizing, which prevents the water from being absorbed into the paper too quickly. Artists use several different techniques when painting with watercolor. A painter may choose to paint with wet-on-dry, wet-on-wet, or dry-brush strokes. Wet-on-dry strokes will not spread because of the sizing on the paper, and water evaporation from the edges of a wet-on-dry stroke results in a darkened line around the stroke's edge. When paint is applied to wet paper, however, pigment will spread in random-seeming patterns through capillary action within the paper. Dry-brush painting produces a rough-looking pattern on the paper, since the almost-dry brush used by the artist only applies paint to the raised areas of the paper.

Computer simulation of fluids

The most appropriate way to reproduce the behaviors of watercolor for a digital painting application is through the use of fluid simulation. Fluid simulation has long been a topic of interest to computer graphics researchers, and can be used to reproduce the distinctive behavior of watercolor. A convincing visual representation of fluid is very difficult to achieve without mathematical simulation, although fluid is a very common element in our surroundings. Therefore, many computer graphics researchers have turned to physics and physics-based simulation in an attempt to create accurate and visually pleasing representations of fluid and fluid motion.

The mathematical underpinnings of fluid simulation are the Navier-Stokes equations for the behavior of incompressible fluid [15]:

$$\begin{aligned} \frac{\partial u}{\partial t} &= -(u \cdot \nabla)u - \frac{1}{\rho}\nabla p + \nu \nabla^2 u + f && (2.1) \\ \nabla \cdot u &= 0. && (2.2) \end{aligned}$$

This is a compact formation of the equations in which ν is the fluid's viscosity, ρ is its density, p is pressure, f is an external force, and ∇ represents a vector of spatial partial derivatives ($\partial/\partial x$, $\partial/\partial y$ in two dimensions, for example) and $\nabla^2 = \nabla \cdot \nabla$ [15]. Equation (2.1) is derived from the conservation of momentum, taking into account the force of gravity, acceleration resulting from differences in pressure, and drag from the viscosity (thickness) of the fluid [5]. Equation (2.2) represents the conservation of mass.

Computer graphics approaches use grid-based numerical approximations to these equations, since it is not possible to attempt closed form solutions. In Foster and Metaxas, a finite difference approximation to the equations is used [5]. They divide a fluid area into a grid of regular cells, assigning to each cell a pressure and a velocity in each direction (vertical and horizontal velocities would be assigned for a 2D grid, for example). A sample simulation grid cell is pictured in Figure 1.

They then choose a timestep and move the simulation forward by calculating new velocities from finite difference equations approximating equation (2.1). They calculate new pressures for each cell and modify velocities according to pressure differentials across cells, a relaxation scheme designed to satisfy equation (2.2). In this

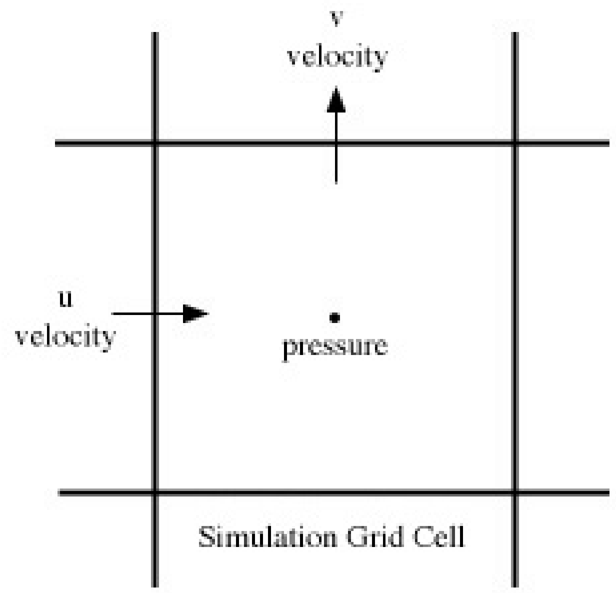


Fig. 1. A Simulation Grid Cell

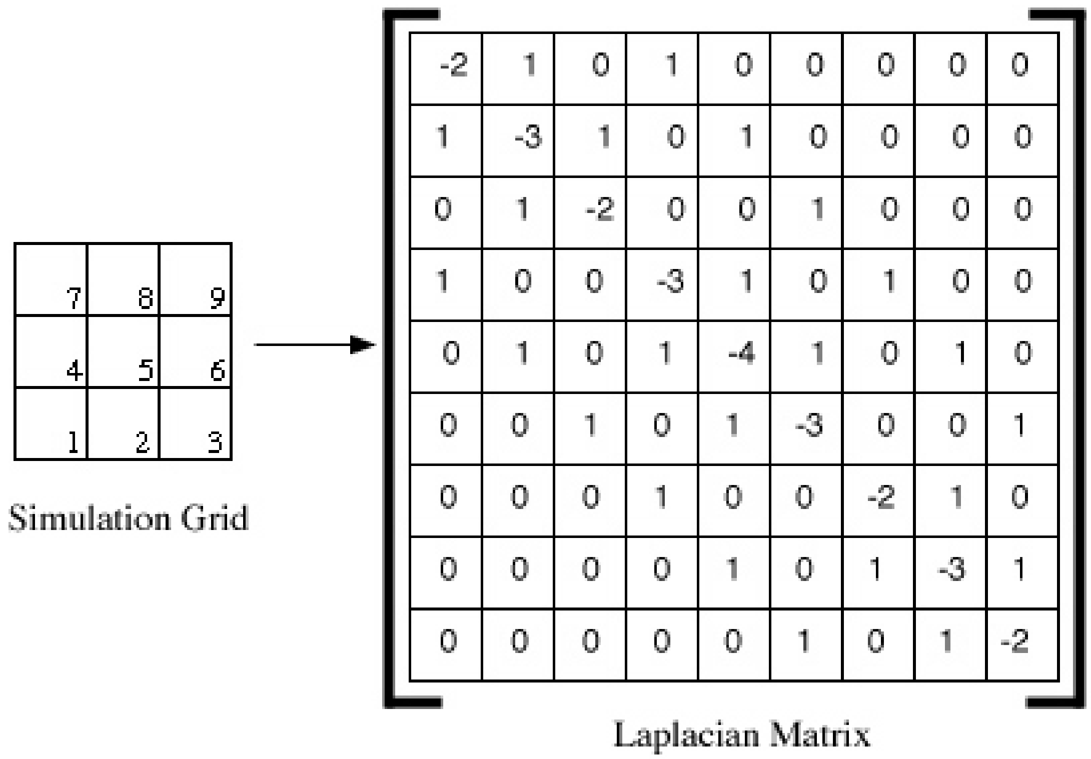
process, the divergence $\nabla \cdot u$ is calculated, which consists of the velocity differential across each cell. Divergence is, therefore, a measure of the rate at which mass is "disappearing". Foster and Metaxas assure that divergence is 0 through an iterative process; each grid cell's pressure is changed in proportion to its divergence, and then the measure of the divergence over all of the cells is taken. This process is repeated until each cell has a divergence less than some specified small amount [5].

The work of Foster and Metaxas provides a solid foundation for much of the later research into fluid simulation techniques; their basic simulation setup has been endorsed by most of the subsequent papers on the topic. However, their solution method can lead to problems; for example, if the timestep is set to too large a value, the entire simulation can become unstable.

Stam makes some improvements on [5], introducing a method for maintaining a stable simulation with a smaller timestep [15]. The most important of the improvements in [15] accounts for fluid advection, which is the way that disturbances propagate through a fluid. Advection is represented by the $-(u \cdot \nabla)u$ term in the Navier-Stokes equations. Stam proposes that this be solved by tracing particles in the fluid back through time, changing the velocity at a particle's current position to the velocity at its last location [15]. This guarantees conservation of momentum in the fluid, and is much more stable than previous methods. Stam's advection method directly replaces Foster and Metaxas' finite difference solution step. In addition, Stam replaces the iterative divergence relaxation of Foster and Metaxas with a direct solution method.

The steps followed by Stam at each timestep of the simulation are as follows: add velocity contributions from external forces such as gravity, calculate current velocities through advection, apply the effects of diffusion, and project the results onto a divergence-free field [15]. In Foster and Metaxas, the effect of diffusion (the $\nabla^2 u$ term in equation (2.1)) is accounted for through the finite difference equations, but here it is a separate, implicit calculation. The final projection step makes the simulation divergence-free by finding new pressures for all of the simulation cells and then using these pressures to modify the velocity field. Instead of taking Foster and Metaxas' iterative approach, Stam proposes the use of a Poisson solver to directly solve for the correct pressures. For a more thorough discussion of Stam's methods, please see [15].

Solving for new pressures in a fluid simulation is both interesting and mathematically complex. The fluid simulation can be represented by a large system of linear equations, in which each cell of the simulation has a small local area of effect. This system, in turn, can be represented as a Laplacian matrix, which is a sparse matrix with a banded structure. Each cell of the simulation grid is represented as a row in the matrix; the diagonal entry represents how many neighbors each cell has, and off-diagonals indicate which cells are adjacent. For example, in this illustration, the first row of the matrix indicates that cell 1 has two neighbors, cell 2 and cell 4. 1's are placed in column 2 and column 4 to indicate these connections, and a -2 in the diagonal to indicate the number of connections, as shown in Figure 2.



$$\begin{bmatrix} -2 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & -3 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -2 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & -3 & 1 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 1 & -4 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 & -3 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & -2 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & -3 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 & -2 \end{bmatrix}$$

Fig. 2. Construction of a Laplacian Matrix

There are several solution techniques which can be applied to sparse linear systems; one of the most commonly applied is the conjugate gradient method. The conjugate gradient algorithm is an iterative solution method for systems of linear equations in the form $A \cdot x = b$, where A is an NxN matrix representing the system of equations, b is a vector of size N, and x is a vector of unknowns. The algorithm generates an initial solution and iteratively improves it until the error is within some user-specified tolerance [13]. The algorithm attempts to minimize the function

$$f(x) \;=\; \frac{1}{2}x \cdot A \cdot x - b \cdot x,$$

where

$$\nabla f \;=\; A \cdot x - b.$$

When the function's gradient, ∇f, is 0, $A \cdot x = b$ and the solution has been reached; this is guaranteed to happen within N iterations.

The physics of shallow-water flows are another necessary component of this work [16]. When creating a fluid simulation to simulate watercolor, the impact of paper height on the fluid simulation cannot be ignored, but a simple 2D simulation will not take it into account. The addition of a third dimension to a fluid simulation would significantly slow down the processing; fortunately, this can be avoided through the application of the shallow-water equations. These equations describe shallow-water flows, in which the depth of the fluid is very small in comparison to its horizontal extent. Typical examples of systems to which the shallow water equations may be applied include tidal flow, oceanic currents, river flows, and atmospheric flows. A complicated simulation such as tidal flow would require consideration of multiple fluid layers, water salinity, atmospheric pressure gradient, wind stress, bottom height, fluid surface level, and fluid depth. For a watercolor simulation, only a few of these need to be considered in addition to the 2D Navier-Stokes equations; bottom height, fluid surface level, and fluid depth.

Bottom height is dealt with simply, as it is analogous to the height of the watercolor paper at any given point. Fluid depth is then calculated by subtracting the bottom surface height from the fluid surface height. The rate of change of the surface level h is given by

$$\frac{\partial h}{\partial t} = -a\nabla \cdot u$$

where $\nabla \cdot u$ is the divergence and a is the fluid depth [16].

Additionally, during each fluid simulation step, the newly calculated cell pressures must be modified according to the fluid height gradient. This models the idea that a change in pressure in a cell will change the height of fluid in that cell at each timestep. The new pressure in a cell is given by

$$\begin{aligned} p_s &= \rho g(h - h_s), \\ p &= p_s + \rho g(h_s - z) \end{aligned}$$

in which g is gravity, p_s is surface pressure, ρ is density, h is the current surface height, and z is depth [16]. For these equations to be applicable, one must be able to make the assumption that the surface can be replaced with a fixed boundary; this is called the rigid-lid approximation. The height of this fixed surface is h_s. For watercolor simulation, we are not interested in carefully modeling surface behavior, so this assumption is appropriate.

An existing fluid simulation-based watercolor tool

The 1997 paper by Curtis et al. [3] describes the implementation of a watercolor tool for artists, upon which the present work is heavily based. Their implementation uses at its base a fluid simulation, modeled upon the work of Foster and Metaxas in [5]. Additionally, they describe the ways in which pigment responds to the fluid simulation and cover many other important characteristics of a watercolor application, such as paper representation and a color model for accurately representing pigments.

The Curtis et al. fluid simulation consists of three conceptual fluid layers; the shallow-water layer, where traditional fluid simulation takes place, the pigment-deposition layer, in which pigment is transferred into and out of paper, and the capillary layer, where pigment and water travel through capillary action. Like most fluid simulations, their simulation is discretized over a grid of cells. The primary difference between this simulation's methods and the Foster and Metaxas methods stems from the fact that watercolor is a very thin layer of fluid over a surface; this required the incorporation of ideas from shallow water-specific fluid dynamics into their simulation. For example, fluid velocity is strongly affected by the local slope of the paper surface, while small changes in the shape of a boundary are discounted in a larger-scale fluid simulation. For the most part, however, the Foster and Metaxas techniques apply with only minor changes. There are, however, some additional considerations; specifically, pigment must be moved through the fluid simulation, and mimicking the visual characteristics of watercolor requires other modifications.

Pigment migration through fluid is fairly straightforward; a small amount of a cell's pigment is moved to the appropriate surrounding cells according to the velocities calculated in the simulation. Pigment absorption and desorption, however, are slightly more complicated; the rate at which pigment is absorbed and desorbed is affected by properties of the pigment as well as the fluid simulation. Another important modification to the simulation attempts to reproduce the edge-darkening effect prominent in actual watercolor. This darkening results from water evaporation from the edge of a pool of water, and Curtis et. al model this by simply removing a small amount of water from cells at the edge of the simulation at each timestep.

Considerations other than fluid simulation also become important when dealing with a watercolor tool; the programmer must consider such things as paper texture, pigment display, mixing, and compositing, and user interface. In an actual watercolor, fluid runs off high areas of the paper and pools in low areas, resulting in a distinctive appearance. Curtis et al. choose to represent paper simply as a height field, discretized at the same scale as their fluid simulation grid. Pigment representation is a bit more complicated; the pigment model used by Curtis et al. includes not only color but also density, granulation, and staining power. These three parameters represent the heaviness of pigment particles in watercolor, the tendency of pigment particles to clump together, and the ability of a pigment to adhere to the paper. Changes to these parameters directly affect the simulation, specifically the simulation step which involves absorption and desorption of pigment.

Curtis et al. also decided to use a complicated color model to represent pigments for added realism; instead of using simple RGB color, they follow the Kubelka-Munk color model. This color model allows pigments to have different appearances over dark and light colors, which realistically models actual watercolor behavior. Some pigments are more opaque than others, and would therefore appear more vividly colored over black than would more translucent pigments. [3] In the Kubelka-Munk color model, pigments are associated with sets of numbers representing the pigment's light absorption and light scattering properties. These numbers, called the absorption and scattering coefficients, are actually samples from functions of wavelength. From these numbers and information about the thickness of a pigment layer, one can calculate reflectance and transmittance of light through the layer by

$$\begin{aligned} a &= 1 + \frac{K}{S}, \\ b &= \sqrt{a^2 - 1}, and \\ R &= \frac{1 - p_0(a - b\coth bSh)}{a - p_0 + b\coth bSh} \end{aligned}$$

where h is the thickness of the layer, S is the scattering coefficient, K is the reflectance coefficient, and p_0 is the reflectance of the material underneath the paint layer [6].

Sampled at red, green, and blue wavelengths, these equations provide values suitable for output to RGB displays such as computer monitors. Curtis et al. provide reflectance and scattering coefficients for a number of common watercolor pigments, which they obtain by specifying pigment colorations over black and white backgrounds [3].

GPU programming

The GPU, or Graphics Processing Unit, is the processor which resides on the graphics card in a computer. A programmer, through one of several methods, can write programs which can be run on the GPU in combination with regular CPU programs. Most GPU programs are created to enhance or speed up the display of 3D scenes, but in recent years, many computer graphics researchers and professionals have turned to graphics cards to provide them with extra computing power. The advantage to using the GPU for general-purpose computation is speed; the GPU is a fast, parallel processor, and can provide extra speed for either artistic renderings or mathematical calculations.

Until quite recently, the only programmable interface to these cards required developers to write in GPU assembly code, which is difficult and time-consuming. NVIDIA, in collaboration with Microsoft, has recently developed a language called Cg, which allows developers to harness the computing power of graphics cards without the necessity of programming in assembly language. The features of this language are set forth in [10]. For the most part, Cg is used as a shading language, giving artists control over the surfaces and appearances of computer-rendered objects. However, there has been much recent interest in using Cg and other GPU languages as vehicles for general-purpose computation [9, 2].

Cg resembles both the C programming language [8] and the RenderMan shading language [1]. A shading language is used specifically to modify the appearance of surfaces in graphics; the designers of Cg decided, however, to avoid specialization and create a general-purpose language like C. Therefore, Cg lacks some of the features that make RenderMan easy to use, such as built-in lighting calculations and surface characteristics. Cg does, however, share one crucial feature with RenderMan which it does not share with C; it uses a stream processing model. In the stream processing model, the programmer writes a program which is executed not once, as in C, but once per unit of input data. The GPU is designed to run short programs on many data elements in parallel. Syntax in Cg is C-like, and the designers of Cg hoped to incorporate some of C's portability and performance into their language [10].

A Cg program must be run within the context of a larger program using a graphics API; both OpenGL [12] and Microsoft's DirectX [14] provide language bindings for Cg users. Since the present work utilizes OpenGL in conjunction with Cg, discussion will focus on using Cg with OpenGL rather than DirectX. Before discussing the graphics hardware pipeline, however, some terminology must be introduced. The *frame buffer* is the memory area in which the image to be displayed is assembled;

for systems with graphics cards, it is part of the graphics hardware. A *vertex* is a defining point on a piece of 2D or 3D geometry; vertices are defined in the OpenGL program. *Primitives* are basic geometric shapes, defined in the OpenGL program as a set of vertices and vertex grouping information. A *fragment* is the name for the set of information necessary to update a *pixel*, which is one element of the frame buffer [4].

The graphics hardware pipeline, as illustrated in Figure 3, consists of four steps; vertex transformation, primitive assembly and rasterization, fragment texturing and coloring, and raster operations [4]. Vertex transformation takes in the vertices defined by the program and results in vertex screen positions, texture coordinates, and colors. Primitive assembly processes the groupings given for these vertices, and rasterization determines which pixels will be covered by the primitives generated by the assembly process, generating a set of fragments. During fragment texturing and coloring, then, each fragment is updated with its correct color and other information. Lastly, raster operations turn fragments into pixels by determining whether or not each fragment is visible and should be displayed.

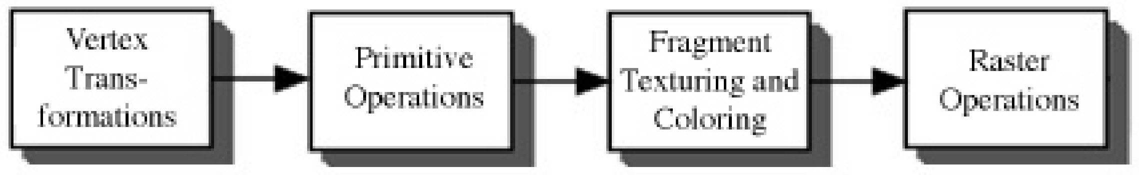


Fig. 3. Graphics Hardware Pipeline

GPU programming enters the graphics pipeline through two types of programs; vertex programs and fragment programs. Unsurprisingly, vertex programs operate on vertices and fragment programs on fragments. In Cg, vertex programs may output a vertex position, texture coordinates, and color, while a fragment program may only output a single color [4]. The graphics pipeline with GPU programs included is shown in Figure 4.

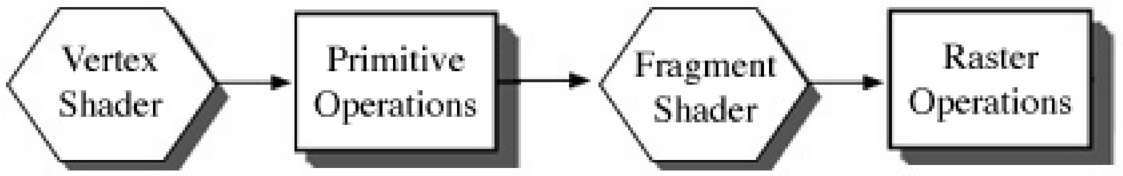


Fig. 4. Graphics Hardware Pipeline with Programmable GPU Elements

The use of GPU programming for increasing the performance of mathematically intensive programs has been the topic of several recent papers in the field of computer graphics. Krüger and Westermann [9], Bolz et al. [2], and Moravanszky [11] all present ways of using GPU programming to speed up mathematical solution techniques. All three of these works present similar techniques for conjugate gradient solvers, which are suitable for fluid simulation. In general, numerical solution methods using GPU programming follow the same basic steps; computational inputs are represented as 2D textures and sent to the GPU, and outputs are written to a buffer or to a texture and returned to the CPU.

The Krüger and Westermann paper [9] focuses primarily on the appropriate formatting of input data for GPU programs dealing with matrix and vector operations. Matrices are stored in multiple 2D textures, with one texture allocated for each diagonal of the matrix. For sparse and highly diagonal matrices, such as the Laplacian matrix used for fluid simulation, this storage method will only require sending a small part of the matrix to the GPU, saving time and space. The paper describes implementation methods for simple vector arithmetic, matrix-vector multiplication, and vector reduction. Using these operations as building blocks, they describe a GPU implementation of the conjugate gradient algorithm which provides significant speedup over the CPU version.

Bolz et al. [2] present similar information; they discuss matrix representation, matrix-vector operations, and a conjugate gradient solver. Similarly, Moravanszky [11] presents detailed information about matrix representation and linear algebra operations, though in a more detailed and practical tutorial format.

CHAPTER III

METHODS

Structure of the watercolor program

The watercolor simulation tool described in this paper is based on the principles and methods described in Curtis et al. [3], but updated to run in real-time with GPU programming. In the application created by Curtis et al., the user sets up a series of "glazes", for which the fluid simulation is calculated later. In my application, however, the user can paint onto a canvas and immediately see the behavior produced by the fluid simulation. Curtis et al. developed methods for handling many watercolor-specific issues, such as the representation of watercolor paper, the representation of pigments, and pigment interaction with the fluid simulation. The main differences between their application and the one described in this work are in the fluid simulation and the application of GPU code; therefore, I have used their methods for other parts of the application as far as possible. Fluid simulation particulars will be discussed in the next section.

A simple height field is used for the representation of watercolor paper. This can be generated randomly or from scanning actual watercolor paper. This height field is read in from an image file for ease of replacement. The Kubelka-Munk color representation system is used to describe pigment color. The watercolor application starts out with six pigments available to the user; these pigments are generated with the absorption and scattering coefficients set forth in the Curtis et al. paper for various common watercolor pigments. The user can then mix varying amounts of these base pigments to create new colors. One difference from the Curtis et al. paper, however, has to do with pigment compositing. Since my application is interactive, there are not well-defined layers as in the previous work. Therefore, instead of compositing colors together, I simply mix pigments according to their concentrations in a given grid cell, which also results in the correct appearance for multiple layers of pigment. This is illustrated in Figure 5.

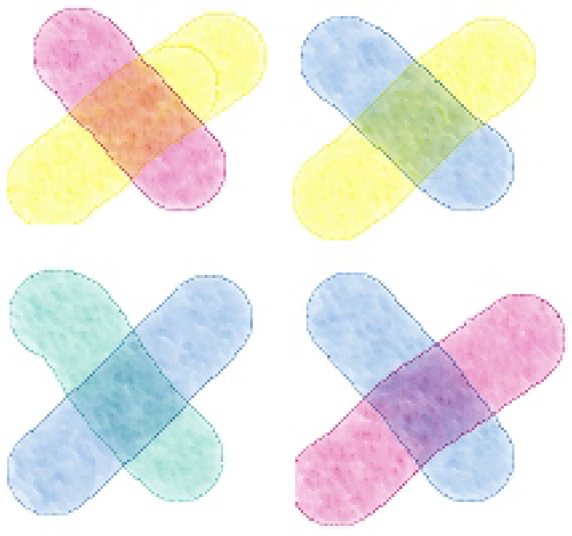


Fig. 5. Example of Application Color Mixing

The fluid simulation

A physics-based fluid simulation is at the heart of this watercolor application. However, where the Curtis et al. paper used the fluid simulation approach of Foster and Metaxas [5], I use Stam's approach [15].

There are four main steps in the fluid simulation loop; adding velocity, tracing back particles, solving for diffusion, and projection onto a divergence-free field. The first step is relatively simple; it is simply adding the effects of gravity to the output from the last fluid simulation step. For each cell, the height of the cell's fluid is compared to the the cells next to it, and the fluid's velocity is adjusted such that fluid flows from higher to lower areas.

Tracing back particles is slightly more complicated; this function is modeled on Stam's particle traceback function, in which velocities are moved through the velocity field [15]. For each fluid cell, an imaginary particle at the center of the cell is traced backward in time, and the center velocity of the cell is updated with the velocity from the particle's "old" location. This propagates velocities smoothly through the velocity field and prevents errors from out-of-range values; the largest value possible in the new field is the largest value present in the old field.

The diffusion step solves for the impact of viscosity on the simulation; it is possible to solve the diffusion equation either directly or implicitly. Stam formulates the diffusion equation as

$$\frac{\partial w_2}{\partial t} = \nu \nabla^2 w_2,$$

where the current velocity field is w_2, viscosity is ν, and ∇^2 is the Laplacian matrix representing the connections within the fluid simulation grid [15].

Stam solves this equation implicitly, but a direct approach is simpler to implement and stable for all but very high viscosities [15]. Therefore, this paper follows the Foster and Metaxas method of straightforward solution. The Laplacian matrix is multiplied by a vector representing u or v velocities, timestep, and viscosity, and the resulting quantity is added to the u or v velocities of the field.

Projection of values onto a divergence-free field is the most complicated step in the fluid simulation process. First, the pressures of the grid cells must be calculated, and then the effects of the pressures must be applied to the velocity field. The method of applying the effects of the pressure calculations is described above; to calculate the pressures, a conjugate gradient solver is used to solve the equation $\nabla^2 q = \nabla \cdot w_3$, where ∇^2 is the Laplacian matrix, q is the quantity being solved for, ∇ is the divergence vector (a measure of differences in cell velocities), and w_3 is the current velocity field [15].

There are several versions of the conjugate gradient algorithm for different types of input matrices. In order to use the simplest conjugate gradient algorithm, the input matrix A (here, ∇^2) must be positive definite symmetric and nonsingular. Unfortunately, the Laplacian matrix for a grid with appropriate boundary conditions is inherently singular. This can be corrected without introducing significant error, however, by adding a small constant to each diagonal term of the matrix. In this work, the conjugate gradient algorithm described by Moravanszky is used [11]. Instead of iterating through the algorithm until error is below some minimum tolerance, however, a small constant number of iterations are performed. This is adequate to provide visual realism and prevent instability; the same technique was employed by Bolz et al. to speed up simulation [2].

GPU implementation

There are four main steps necessary for general-purpose GPU programming with Cg; representation of input data in C++ and OpenGL, passing textures and other inputs to the graphics card with OpenGL, writing Cg programs, and retrieving data from the graphics card. Since the object of GPU programming is a speed increase, all of these need to be handled not only correctly, but also quickly. The graphics card used in this work is the NVIDIA Quadro FX 3000, which supports the latest vertex and fragment profiles as well as full-precision floating point textures.

Handling input data

The inputs to a conjugate gradient solver consist of a large, sparse matrix A and a vector b. The size of b is N and the size of A is NxN, but for the particular type of matrix generated by a 2D fluid simulation, each matrix row has only five non-zero elements. Since N, for this application, is very large (65536 for a 256x256 simulation grid), the matrix representation should only store the non-zero matrix elements. There are many ways to accomplish this; the method set forth by Krüger and Westermann turned out to be the most efficient in terms of speed [9].

In this method, the matrix is stored in five separate arrays of size N. The first array contains the diagonal entries of the matrix, and the other arrays each contain one non-zero off-diagonal. This representation will only work for a specific matrix structure; the non-zero entries must all fall in five diagonal and off-diagonal rows.

Sending input to the GPU

The only method for sending large amounts of input data to the GPU is through the use of textures. Therefore, working with OpenGL, the five arrays representing the matrix and the array representing the vector must be bound into texture memory. Here, it is possible to add another level of compression to the data. Four data elements can be packed into each RGBA texture element, which will later be operated on in parallel. Traditionally, OpenGL textures have been square and power-of-two sized; there is now, however, an OpenGL extension by NVIDIA which allows for rectangular textures. Textures have a maximum size in any direction, determined by the capabilities of a given graphics card. Therefore, decisions must be made about the correct width and height for these textures.

For a 256x256 simulation grid, there will be 6 arrays of length 65536 which must be fit into textures. The maximum texture size for the NVIDIA Quadro 3000 FX is 2048; the simplest method of texture storage would be to fit 4 data elements into each texture element. At 8192 data elements per row, 8 rows will be necessary to store one array.

Due to the way the fragment shader is written, however, an 8x2048(x4, from the RGBA packing) storage structure is not the most logical choice. For each element of the result, the shader will need to retrieve five values from the b vector passed in with the matrix. For the entry in the Laplacian matrix which corresponds to a matrix row m (see Figure 2), vector elements m, $m-1$, $m-\sqrt{m}$, $m+1$, and $m+\sqrt{m}$ must be accessed. These elements will correspond to the columns containing non-zero numbers in the Laplacian matrix for all power-of-two sized grids. To facilitate this access, it is convenient to store the matrix such that entries $m-1$, $m+1$, $m-\sqrt{m}$, and $m+\sqrt{m}$ are adjacent to entry m. This can be accomplished by storing $\sqrt{m}$ entries per column, packed into four-component slots (m, in this application, is always a power of two). For a vector of length 65536, $\sqrt{m}=256$, so the texture would be of size 64x256(x4).

Writing a Cg shader

In general, it is helpful to split processing time equally between vertex and fragment shaders; unfortunately, only the Cg fragment shader can access textures. Since the input data for large matrix-vector computations is stored in textures, it is only possible to make use of fragment shaders in this application. The necessary fragment shader needs to perform matrix-by-vector multiplication, with the inputs described above.

Most of the work done in the shader consists of texture fetches; one RGBA texel is fetched from each of the textures representing the matrix, and five texels are fetched from the array representing the vector. Then, each matrix element is multiplied by the corresponding vector element and the results are added together. In Cg, operations on vectors of four floats are built into the language; shaders which process four elements at a time are as easily writable as those containing only single-element manipulation.

Retrieving data from the GPU

To actually run the Cg shader, the main program must make a call to glDrawPixels; instead of writing to the display, however, one must write to a hidden surface called a pbuffer. This is handled through a class called RenderTexture, developed by Mark Harris [7]. Each set of four results (since the shader operates on four elements at a time) is written into one RGBA pixel of the pbuffer, and the pbuffer can be either read out to an array with glReadPixels or bound as a texture for more GPU operations.

User interface

The user interface, shown in Figure 6, is designed to be as simple as possible; most of a user's time should be spent creating artwork, not searching through menus. Menus, therefore, contain only actions performed very rarely, such as saving, clearing the screen, and exiting the program. There are four main graphical user interface components; a color palette for mixing colors, a brush palette for selecting brush size, a current brush window for selecting the current water content and pigment concentration of the brush, and, of course, a paper area upon which the user paints.

The color palette tool, shown in Figure 7, attempts to mimic an actual watercolor palette; clicking on an empty square brings up a box in which the user can mix colors. This works in much the same manner as mixing actual watercolors; a colored box shows the new color, to which the user can select multiple colors and concentrations

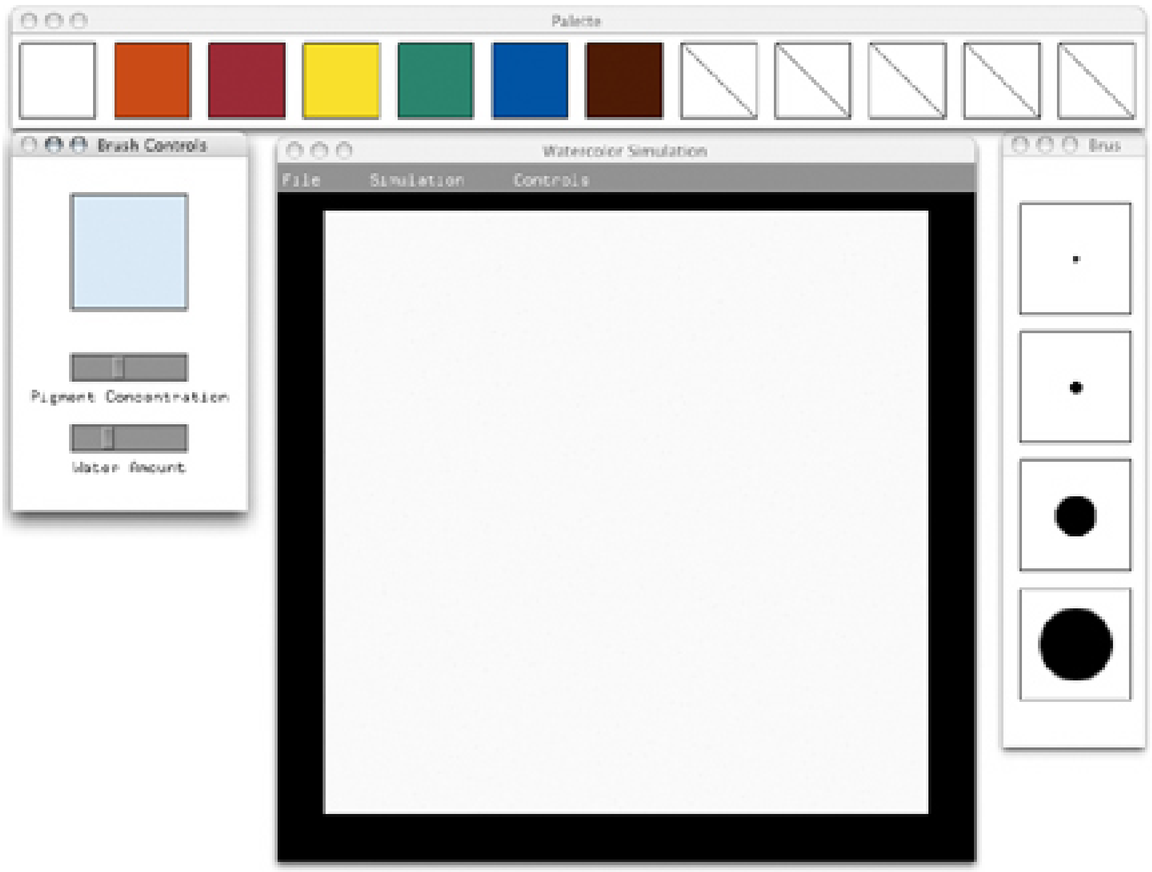


Fig. 6. Watercolor Application User Interface

to add.

The brush palette is similar to those in numerous commercial painting and image editing programs; it contains boxes with images of brush profiles. A user clicks upon the brush of his or her choice, and that brush is used in the paper area until the user selects a different brush. To keep the user interface as simple as possible, a limited number of brush shapes and sizes are available; improvements upon the brush model would be one possible extension of this work. It is also necessary for a user to know exactly what color will appear when he or she applies the brush to the paper. The current brush window provides this functionality with a colored area representing the

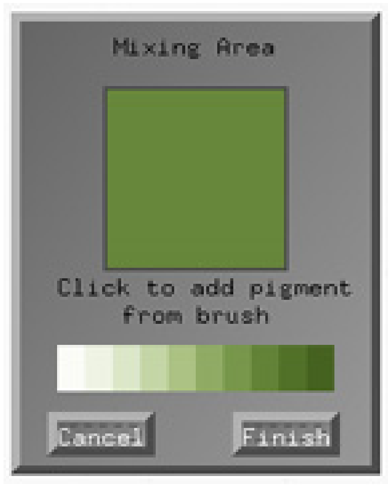


Fig. 7. Color-Mixing Dialog Box

current pigment concentration and color, as well as including sliders to control the concentration of pigment and the amount of water on the brush.

Through the application menus, located at the top of the paper window, the user can perform several rarely used but important functions. The “File” menu offers the standard “Save”, “Clear”, and “Exit” options, as well as “Clear Wet”, an option which clears only wet watercolor, and “Import Drawing”, which allows the user to import a drawing as a background for the paper area. The “Simulation” menu offers options to pause and restart the simulation or to dry the paint, which prevents further brush strokes from affecting the paint already in place. The “Controls” menu allows the user to reset the palette, clearing it of all user-defined colors and making room for more new pigments.

CHAPTER IV

EVALUATION

Usability and output

The watercolor program effectively simulates several types of watercolor strokes, as illustrated in Figure 8. The unpredictability of watercolor makes it an interesting medium with which to work; its basis in fluid simulation allows this application to demonstrate some of the same erratic qualities. The strokes on the left are real and simulated dry-brush, strokes in the middle are examples of edge darkening, and the strokes on the right are wet-on-wet painting. The simulated strokes are similar to the real strokes; they show the "paper" texture convincingly, and demonstrate darker edges and the spreading of paint through wet paper accurately. However, they do still betray their digital origins; the edges on the drybrush stroke are too defined, and the edges of the middle stroke are pixelated. These problems could be resolved through an improved brush model and the ability to run the simulation on a larger grid.

The illustrations in Figure 9 were created for the purpose of comparing the visual appearance of actual watercolors to the output of the watercolor tool; for each pair, a drawing was created and scanned, and painted first by hand and then with the watercolor program.

The digital "watercolors" are reasonably similar in appearance to the actual watercolors; there are some slight differences, but the overall effect of the digital paintings is that of watercolor. The relative coarseness of the fluid simulation grid is partially to blame for these differences; the size of the physical watercolors provided much more opportunity for detail and nuance than the paper size of the watercolor tool. Since the fluid simulation can only run on, at best, a 256x256 grid, the level of

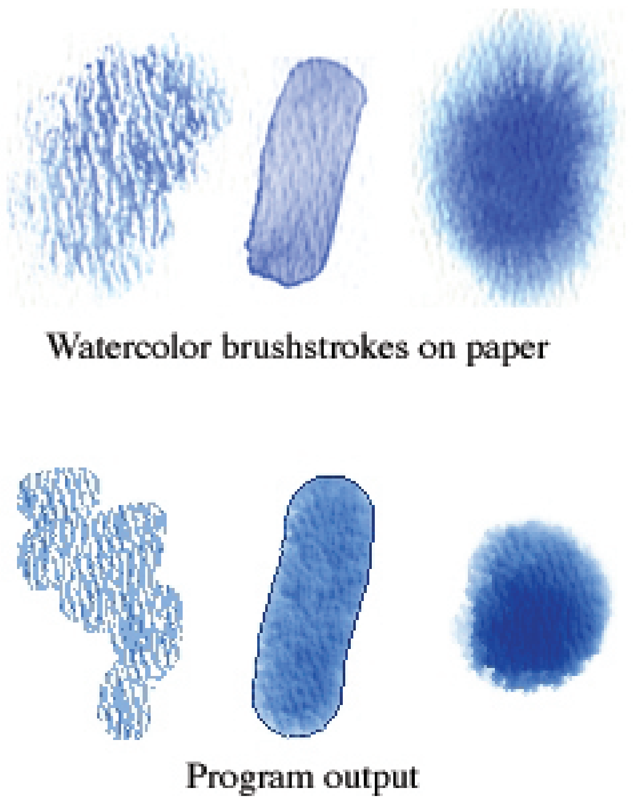


Fig. 8. Comparison of Watercolor Brushstrokes with Program Output

detail possible with a physical watercolor cannot be achieved. Despite this limitation, however, the paintings created through the use of the tool are visually similar to actual watercolor paintings. There are many existing commercial programs which have some watercolor-like capabilities, but these do not simulate the physical processes underlying brushstrokes.

While this watercolor tool is not nearly as complex as a commercial painting program such as Adobe Photoshop, elements of its user interface can still be compared with those offered in commercial programs. In some ways, this application benefits from being small and single-purpose; user interfaces are easier to use when they are simple, as this one is. For example, in Photoshop, the proliferation of menu

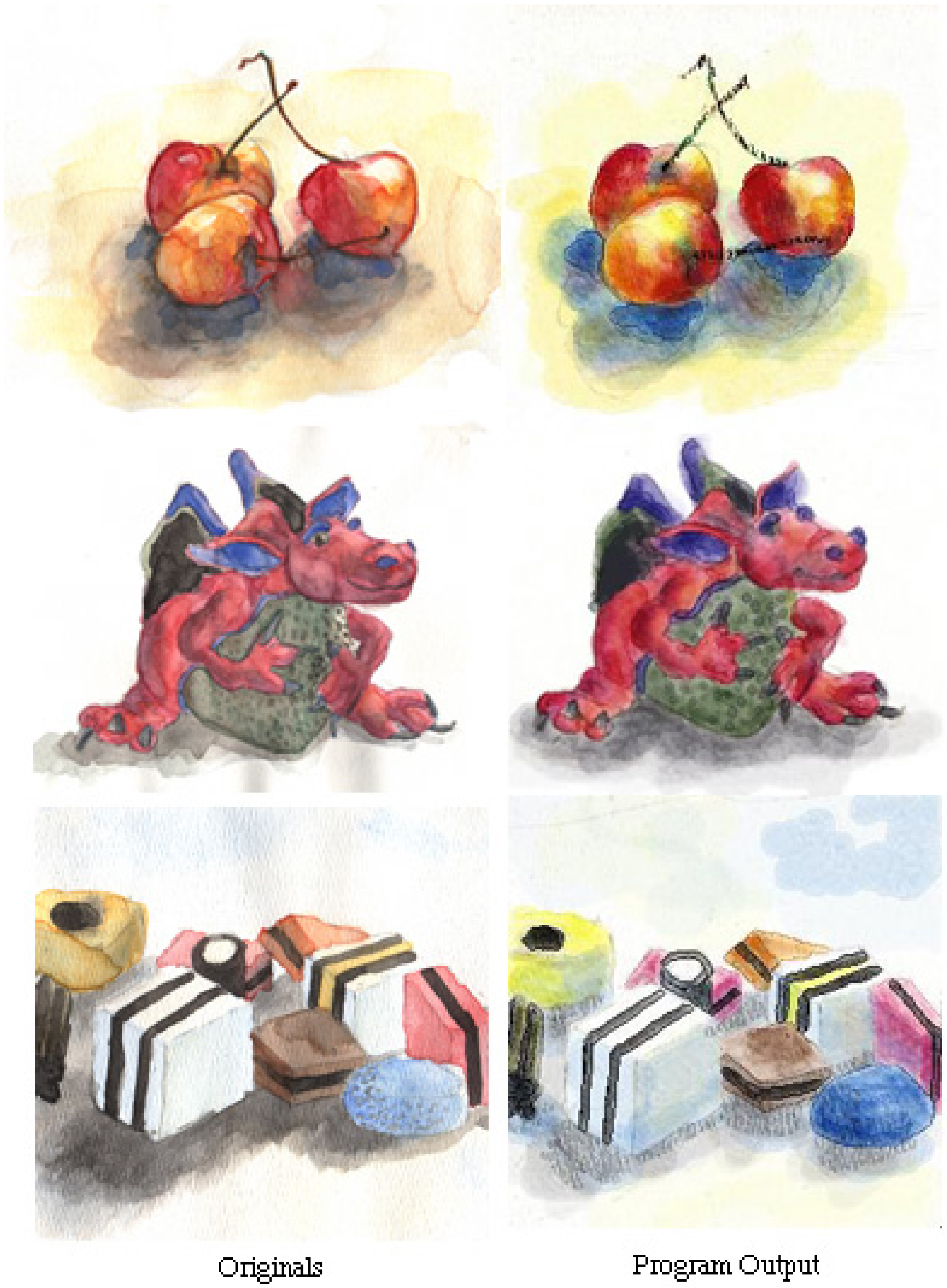


Fig. 9. Comparison of Watercolor Paintings with Program Output

options can often make finding a specific control quite difficult. Commercial programs, however, have the benefits of large development teams and time for usability testing, which result in more polished interfaces.

The color palette is a standard feature in commercial graphics programs; however, these programs often operate with RGB or HSV color spaces while this application used the Kubelka-Munk color model. The Photoshop color palette tool gives users a wider range of color choices than the palette in this application, but does not offer support for realistic pigment mixing. In all, I believe that the Photoshop tool is slightly easier to use but that the palette tool in this application offers a closer match to the process which occurs in creating an actual watercolor.

The method for controlling the shape and size of the brush is modeled upon the Photoshop technique for doing so; a user clicks on a graphic representing a brush shape, which is then used in painting. The main difference between the Photoshop implementation and that of this application is complexity; the Photoshop implementation offers much more control to the artist by providing a large range of brush shapes and sizes. I feel that this element is the weakest portion of the watercolor tool's user interface, and that it could benefit from further refinement.

The brush control window of this application has no precise analogue in Photoshop; it consists of an area showing the current brush color and two sliders to control the water content and pigment concentration of the brush. This user interface element seems to be necessary, adequately simple, and easy to use.

GPU speed and timing

The conjugate gradient step in the fluid simulation portion of this application is implemented through GPU programming; specifically, large matrix-vector multiplication takes place on the GPU. Table I is a comparison of timings for the same algorithm executed on the GPU and on the CPU for different simulation grid sizes; the size at which the program actually runs is 256x256, but other sizes are measured for the sake of comparison. The size of the matrix, however, is N^4, where N is the number of grid cells in one direction; for a 256x256 grid, the uncompressed matrix would contain 256^4, or 4,294,967,296, elements.

Table I. GPU Timing Results (milliseconds per operation)

	multiply		conjugate gradient	
	non-GPU	GPU	non-GPU	GPU
32x32	0.06	0.08	0.49	0.65
64x64	0.19	0.22	1.92	2.07
128x128	0.74	0.61	7.68	7.32
256x256	3.07	2.49	30.89	29.37
512x512	12.8	11.2	127.4	126.0

The results obtained from GPU timing tests were somewhat disappointing. While the addition of GPU programming provides some speedup, the increase is not as significant as those reported in other papers. In the paper by Krüger and Westermann, for example, the authors' GPU matrix-vector multiply was 12 to 15 times faster than an optimized CPU implementation [9]. In this application, however, GPU matrix-vector multiply is only approximately 1.25 times as fast as the CPU implementation. The overall increase in speed of the program is negligible; in both the version with GPU programming and the version without, the fluid simulation updates about 5 times per second and the display approximately 18. This is perfectly adequate for the purposes of real-time interaction, although greater speeds could provide a slight increase in the responsiveness of the application.

The conjugate gradient algorithm contains not only matrix-vector multiplication but also other vector operations; therefore, speedups from faster matrix-vector multiplications will provide smaller percentage speedups when the entire algorithm is timed. For the timing test, five iterations were performed for each cycle of the conjugate gradient algorithm. For grid sizes 128x128 and larger, the GPU implementation shows some speedup over the CPU implementation; if it were possible to perform all of the operations on the GPU without switching back to the CPU, a much larger speedup could be realized. The method used by Krüger and Westermann to achieve this, unfortunately, is a trick which works on some graphics cards but not others. In their paper, they state that they used a texture simultaneously as an input and output buffer [9]; this may have worked on the graphics card cited in the paper but does not work on the NVIDIA Quadro FX 3000, nor on many other graphics cards.

The benefits of GPU programming for matrix-vector multiplication increase with the size of the matrix involved; for smaller grids, the CPU algorithm is actually faster. This occurs due to the overhead necessary for transferring data to and from the GPU. Binding textures, the method used for sending large amounts of data to the GPU, is a time-intensive operation, as is retrieving output with glReadPixels. Due to this overhead, for grid sizes smaller than 128x128, CPU code is faster than GPU code.

The main difference between the Krüger and Westermann implementation and the implementation described in this paper is necessitated by the complexity of the watercolor application; the Krüger and Westermann test application never writes data back from the GPU to the CPU. Without writing data back to the CPU, the average speed for one matrix-vector multiply for the 256x256 simulation grid in this application was 0.9 ms, increasing the speedup from 1.25 times to more than three times. Unfortunately, most complex applications require significant CPU processing, and cannot run all mathematical calculations on the GPU; while the theoretical speedups available from the GPU are quite large, considerations such as this make them unachievable in practice.

When a timing experiment was run with code which did not write data back to the CPU, much more impressive results were seen. In addition, removing texture binding from the timing loop provided even better timing numbers. Table II compares the average speed of one matrix multiply under four conditions: without GPU, with GPU, with GPU but without readback to the CPU, and with GPU but without any CPU interaction whatsoever. Speedups similar to those of Krüger and Westermann can be seen when there is no CPU interaction; unfortunately, these speedups are not available to this application due to its need to frequently send data back and forth between GPU and CPU.

Table II. GPU Timing Results without CPU Interaction (milliseconds per multiply)

	no GPU	GPU	GPU w/o readback	GPU w/o CPU interaction
32x32	0.06	0.08	0.03	0.02
64x64	0.19	0.22	0.07	0.03
128x128	0.74	0.61	0.21	0.08
256x256	3.07	2.49	0.83	0.28
512x512	12.8	11.2	4.89	1.30

For large applications, GPU programming is not yet an appropriate technique for significant performance enhancement. The gains which are seen in theoretical studies do not match those which can be achieved in practice within the context of a larger application, and the amount of processing which must currently occur on the GPU to see significant speedup is almost unmanageably large. GPU programming for general-purpose computation, however, is still a developing field of study, and hardware and software improvements may eventually remedy these problems.

CHAPTER V

CONCLUSIONS AND FUTURE WORK

Conclusions

In this work, techniques for combining GPU programming and fluid simulation to create a watercolor tool have been presented. The resulting application allows users to create watercolor-like paintings in real time. The images produced through the use of the program resemble actual watercolors, and the tool is both simple and enjoyable to use. While the use of GPU programming did not provide as much speedup as could have been hoped, valuable information has been gained about the practical limitations of this method and its applicability within the context of a larger application.

Future work

This thesis provides several directions for possible future work. The watercolor program itself could be improved; specifically, an improved brush model would enhance the usability of the application as an artists' tool. Ideally, some sort of physically-based brush model could be developed and integrated with the fluid simulation to provide an even more realistic experience. The most significant direction for future work, however, would be enabling the program to run on larger fluid simulation grids.

GPU programming for general-purpose computation is an evolving field; currently, at least one new OpenGL extension (GL_EXT_render_target) is under development which will alleviate some of the current problems in achieving practical speedups. When this is released, the application could be revisited and hopefully improved. Additionally, more work could be done with GPU programming in its current state to further improve the program speed. Currently, a large amount of time is spent calculating and displaying pigment mixtures for each cell of the fluid simulation grid; moving these calculations and the screen display to the GPU could provide additional speedup.

REFERENCES

[1] A. A. Apodaca and L. Gritz, *Advanced RenderMan: Creating CGI for Motion Pictures*. Morgan Kaufmann Publishers, 1999.

[2] J. Bolz, I. Farmer, E. Grinspun, and P. Schröder, "Sparse matrix solvers on the GPU: conjugate gradients and multigrid," *ACM Trans. Graph.*, vol. 22, no. 3, pp. 917–924, July 2003.

[3] C. J. Curtis, S. E. Anderson, J. E. Seims, K. W. Fleischer, and D. H. Salesin, "Computer-generated watercolor," *Computer Graphics*, vol. 31, no. {Annual Conference Series}, pp. 421–430, Aug. 1997.

[4] R. Fernando and M. J. Kilgard, *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*. Addison-Wesley, 2003.

[5] N. Foster and D. Metaxas, "Realistic animation of liquids," *Graphical Models and Image Processing*, vol. 58, no. 5, pp. 471–483, 1996.

[6] A. S. Glassner, *Principles of Digital Image Synthesis*, vol. 2. Morgan Kaufmann Publishers, Inc., 1995.

[7] M. Harris, "Render texture class." http://www.markmark.net/misc/rendertexture.html, August 2004.

[8] B. W. Kernighan and D. M. Ritchie, *The C Programming Language*. Prentice-Hall, Second Edition, 1988.

[9] J. Krüger and R. Westermann, "Linear algebra operators for GPU implementation of numerical algorithms," *ACM Trans. Graphics*, vol. 22, no. 3, pp. 908–916, 2003.

[10] W. R. Mark, R. S. Glanville, K. Akeley, and M. J. Kilgard, "Cg: A system for programming graphics hardware in a C-like language," *ACM Trans. Graphics*, vol. 22, no. 3, pp. 896–907, 2003.

[11] A. Moravanszky, "Dense matrix algebra on the GPU," in *ShaderX2: Introductions and Tutorials with DirectX 9.0* (W. Engel, ed.), Wordware Publishing, Inc., 2003.

[12] J. Neider, T. Davis, and M. Woo, *OpenGL Programming Guide.* Addison-Wesley Publishing, Second Edition, 1997.

[13] W. H. Press, S. A. Teukolsky, W. T. Vetterling, and B. P. Flannery, *Numerical Recipes in C: The Art of Scientific Computing.* Cambridge University Press, 1992.

[14] M. D. Root and J. Boer, *DirectX Complete.* McGraw-Hill Companies, 1998.

[15] J. Stam, "Stable fluids," *Proc. SIGGRAPH '99*, pp. 121–128, Los Angeles, Calif., Aug. 1999.

[16] C. B. Vreugdenhil, *Numerical Methods for Shallow-Water Flow.* Kluwer Academic Publishers, 1994.

VITA

Jessica Stacy Scott

104 Marlboro Street

Quincy, MA 02170

Education

M.S. Visualization Sciences, Texas A&M University, College Station, TX, 2004.

B.A. Computer Science; Studio Art, Williams College, Williamstown, MA, 2001.