University of Nevada
Reno

Wildfire Simulation on the GPU

A thesis submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

by

Roger Viet Hoang

Dr. Frederick C. Harris, Jr., Thesis Advisor

December, 2008

University of Nevada, Reno
Statewide • Worldwide

THE GRADUATE SCHOOL

We recommend that the thesis
prepared under our supervision by

ROGER V. HOANG

entitled

Wildfire Simulation On The GPU

be accepted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

Frederick C. Harris, Jr, Advisor

Sergiu M. Dascalu, Committee Member

Timothy J. Brown, Graduate School Representative

Marsha H. Read, Ph. D., Associate Dean, Graduate School

December, 2008

Abstract

The environmental, social, and economic effects of wildfires have led researchers to develop various models to study the behavior of this phenomenon. These models vary widely in terms of complexity, with some models simulating the basic spread of surface fires and other more complex models simulating the progression of fire through tree crowns and lofting embers. The computational requirements of these more complex models limits their use in prediction and interactive applications, and increasing the parallel computational power through the addition of more CPUs is not always cost-effective. At the same time, the increase in computational power of relatively inexpensive graphics cards has led to their use as parallel general purpose processors. This thesis examines the viability of harnessing the power of GPUs to simulate fire spread. A fire spread model that incorporates the effects of surface fire, crown fire, and fire acceleration is developed. A mapping of this model to GPU concepts is presented, and the results of an implementation are discussed.

Acknowledgments

This work is funded by the STTC CAVE Project (ARO# N61339-04-C-0072) at the Desert Research Institute.

Thanks to my committee: Dr. Frederick C. Harris, Jr., Dr. Sergiu M. Dascalu, and Dr. Timothy J. Brown.

Also, thanks to my family for putting up with me and my rampant use of their electricity.

Finally, thanks to all my friends and co-workers at UNR and DRI.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Every year, wildfires destroy millions of acres of land and costs millions if not billions of dollars to control. From 2000 to 2002, over 18 million acres were burned, over 2,000 structures were destroyed, and over 3.4 billion dollars were expended just for suppression efforts. Beyond the immediate damage caused by wildfires, there are also lingering effects that are not only environmental, but social and economical as well [24].

To better combat wildfires, scientists have developed models to predict the spread of fire. These models may be used to identify and reduce potential fire risks. Areas determined to be highly combustible or susceptible to crowning may be candidates for controlled burns or forest thinning. Prediction of fire growth given some set of conditions could be used to more efficiently allocate suppression resources. Integration of these models into training exercises could increase the effectiveness of such exercises.

For these reasons, several simulators have been created over the years that incorporate various fire models. More complex simulators tend to take extremely long times to execute. As such, their effectiveness at real-time prediction and their ability to react to unexpected changes in the environment are reduced. Increasing computational power by increasing the number of CPUs may prove uneconomical and impractical for more mobile applications. At the same time, the computation power offered by commodity graphics hardware has increased at a faster rate than CPUs. The advent of the programmable shader allowed not only for elaborate graphical effects to be implemented but also for the GPU to be used as a general purpose processor with a large number of processing cores.

Applications of using the GPU as a general purpose processor have been successful in a wide range of simulation applications; however, their applicability towards the simulation of large scale wildfire growth has not been examined. This work addresses this problem by developing a wildfire simulator that can be executed in real-time on commodity graphics hardware. Two approaches were taken. The first allows for GPU computation of fire spread where the costs of spreading a fire to adjacent cells remain the same for the span of the time step. The second approach allows for GPU computation of fire spread where the costs of spreading are not known until a cell is actually ignited.

The rest of this thesis details the development of a GPU fire simulator. Chapter 2 examines the background information on fire modeling, computational simulation of wildfires, and the history of GPUs and their applicability to general purpose computation. Chapter 3 motivates the development of a GPU wildfire simulator. Chapter 4 discusses the design and underlying components of the software developed. Chapter 5 details the model used to simulate fire spread and its mapping to the GPU. Chapter 6 presents the results. Chapter 7 concludes and offers several directions for future work.

Chapter 2

Background

This chapter delves into the history of work done on fire models and computer simulators of these models. It then details the nature and evolution of the GPU from a graphics processor to a general-purpose processor.

2.1 Fire Models

Research into parameterizing various aspects wildfires has been ongoing for decades. This section highlights components of this research that are relevant to this work. For a more complete treatment of fire models, the reader is referred to [41].

2.1.1 Taxonomy of Fire Models

Both [41] and [42] outline three types of fire models: theoretical, empirical, and semi-empirical. Theoretical models are models that rely solely on physical principles. The advantage of such reliance is that these models are based on known properties. On the other hand, the utility of these models in practice is questionable due to the difficulty of obtaining the appropriate inputs. On the other end of the spectrum are empirical models, which are generally statistics that can be used to predict fire behavior under certain conditions. Beyond this set of conditions, purely empirical models have had little success. Between these two extremes lies semi-empirical or semi-physical models that rely on some theoretical principles which are adjusted with some experimental data. Because of their reliance on experimental data, some calibration may be needed in order to apply these models to different conditions.

2.1.2 Fire Shape

Most work done in determining the shape of a fire as it spreads has concluded that under homogeneous conditions, fire will spread in roughly an elliptical shape of some sort. [13] compared the effectiveness of a simple ellipse, a double ellipse, an ovoid, and a rectangle and found that while any of those shapes could adequately fit the contours of various observed fires, the ellipse and double ellipse were found to fit best under homogeneous conditions. As wind speed increased, the length to width ratio of the ellipse would likewise increase [1].

Spread of a fire tends to be modeled using Huygens's Principle, which assumes that every point along a fire perimeter will ignite another fire that grows elliptically according to the environmental characteristics of that point. At the end of a time step, the new fire perimeter is given by the outline of all of these new ellipses [43].

2.1.3 Model Subsystems

[41] further classifies models by the aspect of wildfire that a model is attempting to parameterize. This classification divides fire models into surface fires, crown fires, and spotting.

Surface Fire

Surface fires are characterized by the burning of fuels that are less than two meters in height. While numerous models for surface fire spread have been developed, few have been applied to real-world applications. The most successful of these models is a semi-empirical model developed by Rothermel in 1972 [41]. In [44], Rothermel found that the forward rate of spread for a wildfire was approximated by the equation

$$R = \frac{I_R \xi (1 + \phi_w + \phi_s)}{\rho_b \varepsilon Q_{ig}}$$

where I_R is the reaction intensity, ξ is the propagating flux under zero wind and zero slope conditions, ϕ_w is a coefficient resulting from wind, and ϕ_s is a coefficient resulting from slope.

The previous equation approximates spread in one dimension with both the direction of maximum wind and the direction of maximum slope oriented along this axis. [45] presents a method for finding a two dimensional spread vector if the slope and wind directions are not aligned. The maximum spread rate is found using only the slope in the direction of maximum slope and no wind; likewise, the maximum spread rate is found using only the wind speed and direction and no slope. These two vectors are then added together to yield a vector describing the maximum spread rate.

Crown Fire

Crown fires are characterized by the spread of fire into the crowns of trees. Modeling of such a phenomenon attempts to determine how a surface fire transitions to a crown fire and how such a crown fire would modify the spread of fire [41].

In [50], Van Wagner classifies crown fires into three categories and details conditions required for each type. The conditions are tied to the surface fireline intensity and the surface fire spread rate. A passive crown fire characterized by the torching of trees occurs when the fireline intensity crosses a threshold intensity required for the crowns to be ignited, but the surface fire spread rate is less than the spread rate required for an active crown fire. Should the surface rate cross this threshold value, the fire is then considered an active crown fire. Such a fire spreads through both the surface fuels and crown fuels simultaneously. The surface fire aids in igniting the crowns while the crown fire increases the heat radiated to the surface fuels in front of the fire. If the horizontal heat flux required to ignite the crowns can be supplied completely by the burning crowns, then the fire is considered to be independently crowning, spreading without being linked to the surface fire below.

While Van Wagner outlined various crown fire types and the conditions for each, the quantitative effects of these fires had to be measured and calibrated for particular situations. In [45], Rothermel provides one such set of values. The average spread rate of a crown fire was found to be roughly 3.34 times the spread rate computed for fuel model 10 with the wind reduced by a factor of 0.4. Despite its purely empirical nature and relatively high standard deviations, these values have been applied to other situations [41].

Spotting

Spotting occurs when burning embers are carried into the air and land somewhere in the landscape, possibly igniting it. This presents a slew of problems ranging from limiting the efficacy of fire barriers to altering the shape, size, and progression of a fire [41]. As an example, spotting was observed as far away as ten kilometers from the fire front [4].

Major models in this cateory have been primary theoretical in nature and have focused on determining the maximum spotting distance. Albini proposed a set of models to determine the maximum spotting distance from torching trees, fuel piles, and surface fires [41]. In the torching tree case, embers are lofted vertically to some maximum height and then fall horizontally with the wind field [11]. While Albini's models computed the movement of cylindrical particles that ignored wind parameters as they were being lofted upwards, more recent research has been focused on other shapes such as spherical particles being lofted [52] and propagated [53] and disk-shaped particles moving through a three-dimensional plume that accounts for wind during the lofting stage [46].

2.2 Fire Simulators

As computer technology advanced, fire models were incorporated into various software applications. In the one-dimensional case, several applications were developed that could compute various aspects of wildfires under some set of conditions [2, 3]. With the increase in availability of spatial landscape data, several simulators have been developed that account for both spatial and temporal variations in landscape characteristics. Such simulators can be divided into two classes: vector-based and raster-based.

2.2.1 Vector-based Approaches

Vector-based approaches more strictly follow the idea elliptical wave propagation; that is, the fire shape after some time step can be found by generating ellipses along the previous fire shape and determining the new outline. Simulation is done on a continuous space. While greater in accuracy when compared to raster-based approaches, their complexity and time requirements are also greater [41]. Compared to the number of raster-based ones, few simulators use this approach including SiroFire [7] and Prometheus [6], and of them, the most commonly known is FARSITE.

FARSITE

Utilized globally [41], FARSITE is a vector-based wildfire simulator. It incorporates a number of fire models, including Rothermel's surface spread model and Van Wagner's crown fire model. Spotting is implemented via Albini's spotting model for torching trees with an adjustable percentage reduction in the number of brands that actually ignite new fires. Fire acceleration is also accounted for. Acceleration addresses changes such as a fire heating nearby fuels and increasing the potential for spread as well as to prevent instantaneous jumps in spread rate when the perimeter enters an area with a different fuel type [11]. More recent work on FARSITE includes the integration of a post-frontal combustion model [12]. The simulator has been used in a number of fuel treatment assessment applications both by itself [51] and in conjunction with FlamMap [48].

2.2.2 Raster-based Approaches

Raster-based approaches, or cellular approaches, propagate fire through some set of rules across a uniform grid. While faster and simpler to implement, they lack precision when compared to vector-based approaches [41]. Depending on the number of paths that a fire can travel across, distortion is possible [11].

HFire

HFire is a cellular fire spread simulator designed to compute the spread of surface fire in chaparral fuels. It allows fire to spread from cell to cell in eight directions, four orthogonal and four diagonal. A fire ellipse is computed using Rothermel's spread rate equation and then spread rates in each of these spread directions are derived from the result. The simulator assumes that fire spreads at the maximum rate (no acceleration is accounted for) and mitigates the effects of distortion due to the limited spread directions by using an adaptive time step and finite fractional distances. The adaptive time step determines the minimum time required for a fire to spread from at least one burning cell to another burning cell. Using such a mechanism allows the simulation to quickly simulate large time steps when a fire moves slowly and vice-versa. In conjunction with an adaptive time step, finite fractional distances allow a fire to spread some partial distance to other cells within a time step. Should a fire spread farther than the distance separating two cells, the extra portion of that spread is contributed to the partial distance burning out of the newly burning cell [23].

FlamMap

While FARSITE computes the spread characteristics of a fire over time, FlamMap assumes that the entire landscape is ignited and computes the spread characteristics for each cell of the terrain. It also provides functionality to compute the minimum time required for a fire to spread from an ignition point to any other point on the landscape given constant temporal conditions [10]. This particular capability is detailed in [21]. In essense, every node in the grid is connected to each other. The cost in time of traversing any connection is computed accounting for changes in fuels and the length of the connection across each fuel type. Given this information, a shortest path algorithm is executed to determine the minimum travel time to every node in the simulation space. Optimizations such as stopping the search if no travel times are updated within a certain range can be used to speed up this algorithm.

2.2.3 Computation Time

While vector-based approaches may be more precise, they can also be rather time-consuming. While [48] computes the spread of fire and a few other characteristics using FARSITE, another set of characteristics were computed with FlamMap instead since a FARSITE simulation could run for hours while FlamMap output was almost instant. On a set of trial runs, HFire was found to run roughly 92 times faster than FARSITE running with several of FARSITE's capabilities disabled to more correctly compare the outputs [23].

2.3 GPU Computation

Because of their intended purpose, graphics processing units(GPU) are designed to process a massive number of vertices and pixels in parallel. The introduction of the programmable shader allowed for an array of new graphical effects to be implemented. At the same time, the programmability of the GPU allowed for the exploitation of its resources for non-graphical purposes [22].

2.3.1 Graphics Pipeline

At its core, the graphics pipeline consists of four stages: vertex transformation, primitive assembly and rasterization, fragment transformation, and framebuffer operations. The first stage transforms primitives defined by a set of vertices by a set of transformation matrices before being clipped and converted into fragments (rasterization) in the next stage. Textures and other per fragment operations are performed in the third stage. In the fourth stage, before the fragment is sent to framebuffer, a set of operations can be applied to possibly discard the fragment due to depth testing or to blend the resulting fragment with the data currently stored in the target pixel [47]. Figure 2.1 shows a picture of this pipeline.

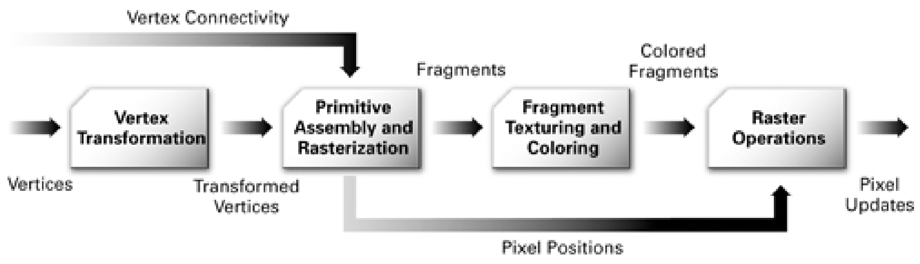

Figure 2.1: Traditional Graphics Pipeline [9]

Programmable Shaders

Over time, modifications to the standard pipeline have been made to allow for more developer control over it. Control over the vertex transformation and fragment transformation stages of the traditional pipeline was permitted through the use of vertex [40] and fragment shaders [38], respectively. A shader is essentially an executable that can be run on some programmable part of the GPU [39]. Figure 2.2 shows these modifications. In addition to vertex and fragment shaders, a geometry shader stage was added directly after the vertex transformation stage. A geometry shader takes whole primitives transformed in the vertex transformation stage and outputs a variable number of another primitive type [29].

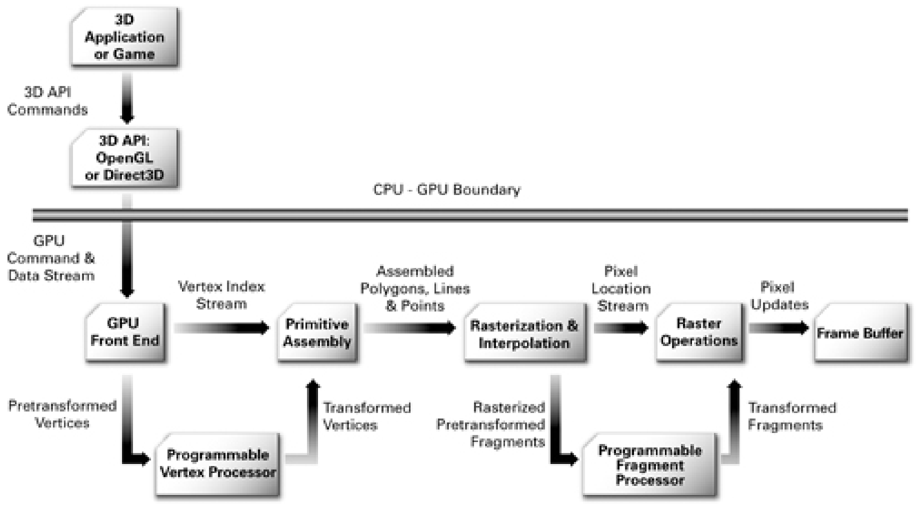

Figure 2.2: Programmable Graphics Pipeline [9]

OpenGL Extensions

Beyond the addition of shaders, several other extensions to OpenGL were made that extended the capabilities of GPUs and made general purpose computing on the GPU more viable. Textures were no longer limited to 8-bit integer representations with the introduction of floating point textures capable of storing both 32-bit and 16-bit floating point values [27]. Similarly, another extension brought the ability to store full 32-bit or 16-bit integers into textures [30], and shaders were updated to support integer computation [32]. The updating of these textures using shaders was made possible with the advent of the framebuffer object, allowing directing rendering access to the them [28].

With the addition of 32-bit floating point color textures, the ability to use 32-bit floating point depth buffer textures was also introduced. Using this sort of depth buffer no longer locks the depth range between 0 and 1 [26]. Query capabilities were also added, allowing one to obtain a count of the number of fragments actually drawn into the framebuffer [31].

2.3.2 SIMD Architecture

To process large numbers of vertices and primitives and because these objects can be processed without regard for other objects, GPUs adopt a single-instruction multiple-data (SIMD) parallel processing architecture, employing a large number of processors all executing the same set of instructions simultaneously [49]. Earlier architectures such as the one used in the NVIDIA 6000 series separated these processors into vertex processors and fragment processors that can execute their respective shader programs [16]. More recent GPU architectures take a unified approach, providing a set of stream processors that can be used for any sort of shader [20].

2.3.3 GPGPU

Given the parallel processing power of a GPU in conjunction with the availability of these processors in commodity hardware, a great deal of work has been done on moving computations generally done on the CPU to the GPU, something that has been dubbed General Purpose computations on the GPU (GPGPU). This is accomplished by storing data into graphics memory in textures and updating them through the use of shaders [19]. The proliferation of GPGPU has led to the creation of special cards such as the NVIDIA Tesla dedicated to GPGPU with a GPU onboard but no video outputs [25].

Applications

GPGPU has been applied to a broad spectrum of applications. More graphical applications include not only the rendering but also the creation, updating, and destruction of a massive number of particles [17] to the simulation of fluid effects such as fire and water [8]. On the less graphical side, n-body simulations [37] and computation of all pairs shortest paths [15] have been performed.

GPGPU Languages and Abstraction

A learning curve exists for those that wish to do work with GPGPU. Because GPUs were designed originally for graphics, knowledge of graphics and graphics APIs is necessary. To perform computations using shaders, an understanding of a shading language such as GLSL [14] is also necessary. Because of the GPU's architecture, programs must be written with a stream processing mentality.

To reduce the price of admission, work has been done to abstract away the graphics elements of GPGPU and provide direct access to the GPU. Some earlier work includes the development of Brook. Data-parallel sections of code can be denoted in a Brook program. These sections are then compiled into shader code while the remainder of the code is compiled into C++ code which executes the shader code using a run-time library. The underlying graphics calls are blocked from the developer's view [5]. Accelerator takes a similar approach by providing a library that developers can use to execute data-parallel operations over arrays [49].

While the previous two approaches block the graphics API from the developer's view, the graphics pipeline is still running in the background. More recent work removes the pipeline completely and allows for direct access to the GPU's stream processors. This idea is embodied in NVIDIA's Compute Unified Device Architecture (CUDA). Programs written with CUDA can execute thousands of simultaneous threads that run on the GPU. Unlike using a fragment shader, scattered writing into memory is possible [36]. Packs of threads called warps are executed together on a single stream processor in a model called single instruction multiple thread (SIMT). As a result, branching is localized, and applications can be written to reduce branching within warps [18].

Chapter 3

Real-time GPU-based Wildfire Simulation

While several fire simulators have been implemented in the past, many require excessive amounts of time to complete their simulations. Because of this, these simulators as they are are unsuitable for real-time purposes such as interactive training and prediction. For example, the value of the predictions made by these simulators decreases as the time it takes to actually compute these predictions increases. Modifications to the simulation data due to unexpected changes in winds can further increase delay.

To increase the effectiveness of fire simulation in a real-time setting, some relaxations or approximations of the model could be made. Alternatively, the simulator could be implemented in a manner that supports parallelism. While developing a simulator to run on multiple CPUs or multiple processing cores may bring a simulator to real-time speeds, it is not necessarily cost-effective to do so due to the price of CPUs. Additionally, while simulation is useful in itself, visualization of the results may be just as important. However, the continuous transfer of simulation data from system memory, perhaps even across systems, to the graphics card can become a bottleneck itself.

For these reasons and given the results obtained by other similar projects, moving the fire simulation onto the GPU warranted experimentation. The focus of this work is on such an experiment. Given the various stages of a fire simulation, a set of modifications and mappings were developed to allow what have been traditionally

CPU-based techniques to be performed on the GPU. In the end, a fire simulator will be produced that, other than the CPU calls used to initiate and control the computations as well as to feed in user interaction, resides on the GPU.

To do so, a raster-based fire spread approach will be developed that incorporates surface fire and crown fire models in addition to fire acceleration. The resulting model can be thought of as a compromise between several fire simulators, adopting the raster-based aspects of HFire for speed purposes while expanding its accuracy and applicability by incorporating various aspects of FARSITE. Rather than adapting timesteps and computing the transfer of burn progression, this model instead will compute all possible spread given some arbitrary timestep, a characteristic more akin to FlamMap's minimum travel time functionality. However, unlike FlamMap, temporal alterations to the system will be permitted through the use of discrete events. Time steps will then be divided to account for these changes. Partial burning will be introduced to reduce distortion caused by these smaller time steps.

This approach will then be mapped to the GPU. Discretized terrain properties such as topology and fuel models will be stored in video memory using textures. Updating of this data will be performed by using a number of shaders that read the data from textures and output the updated values into another texture. Looping will be accomplished by executing a shader repeatedly, and a method for determining a stopping condition will be established.

An interface will be designed that would allow for an application to control the flow of the simulation as well as to inject events that would alter its progression. In addition, the interface would provide mechanisms for the application to access the state of the simulation in order to provide feedback to the user. These mechanisms will also be designed to be usable in systems running with multiple video cards. Synchronization of data will be provided through co-simulation.

The resulting simulator is expected to perform in a manner that would allow it to run in coordination with some sort of visualization system at interactive rates at various time scales. While the overall model developed would be slower than HFire due to the incorporation of various FARSITE features, such slowdown is expected to be tempered by the parallel processing capabilities of the GPU. Timings will be performed to determine conditions under which the simulator could achieve interactive rates.

Chapter 4

Software Engineering

This chapter outlines the software design decisions made and the reasoning behind them. The underlying data structures are discussed, and the interaction between components of the system are examined.

4.1 Requirements and Use Cases

Because the goal of developing a real-time fire simulator is to allow for user interactivity, whether the user is a human or a computer, the functional requirements reflect these intentions. Furthermore, the notion of human interactivity implies a visualization of some sort in order for the user to understand the simulation. Hence, because the rendering environment cannot be predicted beforehand, requirements were developed to facilitate the possibility that the simulation would have to be replicated across multiple rendering contexts. These functional requirements are enumerated in Table 4.1. Because the simulator is designed to be a component in some larger package, the user in this case is the calling application. Nonfunctional requirements were also developed to assure speed and consistency through development. These requirements can be found in Table 4.2.

Figure 4.1 shows the use cases generated from the functional requirements.

F01	The simulator shall allow for multiple graphical contexts.
F02	The simulator shall allow for user-defined ignitions.
F03	The simulator shall allow for user-defined fuel modifications.
F04	The simulator shall allow for user-defined moisture modifications.
F05	The simulator shall allow for user-defined modifications to wind.
F06	The simulator shall allow user access to all graphical assets.
F07	The simulator shall simulate the spread of fire.
F08	The simulator shall allow for control of the forward flow of time.

Table 4.1: Functional Requirements

NF01	The simulator shall be written in C++.
NF02	The simulator shall run primarily on the GPU.
NF03	The simulator shall use OpenGL.
NF04	The simulator shall use shaders written in GLSL.

Table 4.2: Nonfunctional Requirements

4.2 Graphics Data Structures

Given that the bulk of the simulation will be computed on the GPU, a set of data structures was developed to facilitate more rapid development. Their relationships to each other are summarized in Figure 4.2. Their design and purpose are discussed in the following subsections.

4.2.1 Texture and TextureFormat

The Texture class was designed to abstract away details regarding texture creation and maintenance. Because textures with similar pixel formats, internal formats, mipmapping, and other texture settings are often created, these settings are separated from the Texture class itself and relocated into a TextureFormat class that may be used multiple times. Binding textures to various texture units is also handled through the Texture class. Spatially organized simulation data, such as the elevation map, fuel map, and moisture maps, are organized into regular two-dimensional rectangular lattices. As such, two-dimensional textures are used to store such data. To prevent loss of precision, simulation data is stored using 32-bit floating point textures.

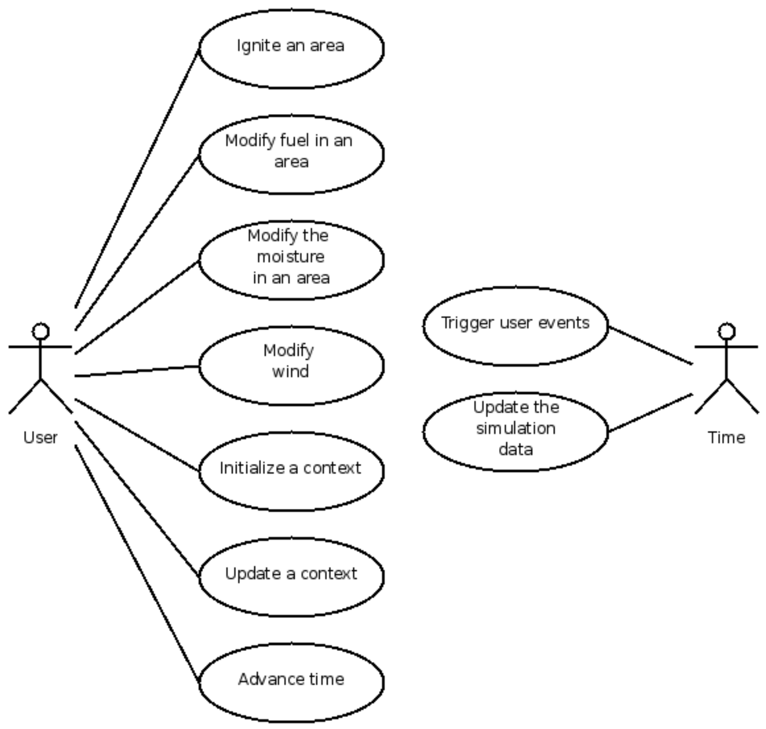

Figure 4.1: Use case diagram

4.2.2 Framebuffer

Textures containing simulation data are useless without a method of manipulating their contents. The `GL_EXT_framebuffer_object` extension [28] provides this functionality by allowing one or more textures to be attached to a framebuffer object (FBO) and subsequently written to by binding the FBO as the render target and rendering fragments into the texture. Blending functionality is retained while using an FBO, facilitating the partial burn component discussed later on. Depth testing can still be used by attaching a special texture to the FBO's depth attachment, which facilitates the spread algorithm also discussed later. The Framebuffer class itself manages the creation, binding, error checking, and destruction of these objects.

4.2.3 VertexBuffer and TextureBuffer

While textures provide an analogue for two-dimensional arrays, vertex buffer objects (VBO) [35] provide an analogue for one-dimensional arrays. Data stored in a VBO

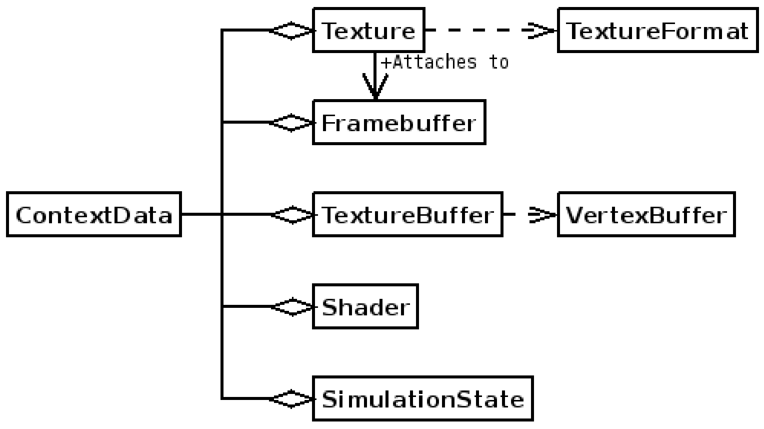

Figure 4.2: Graphics Data Structure Relationships

reside in graphics memory and can be used as vertex data such as position, texture coordinate, or element index. Later extensions increased the utility of VBOs, including the introduction of transform feedback [34], allowing transformed vertex data to be streamed out of either the vertex or geometry shader stages and into another VBO for later use. This particularly capability allows for the simulation of a variable number of particles, which were used to model ember flight and fire spotting. The VertexBuffer class facilitates the creation, destruction, memory management, and data uploading for these structures. Further extending the capabilities of the VBO was the addition of the `ARB_texture_buffer_object` extension [33], allowing VBOs to be bound as textures accessible by shaders. Because a large number of variables are constant for a particular fuel model, rather than storing these constants for every single cell, memory conservation was achieved by storing these constants into VBOs which were then converted to texture objects using the TextureBuffer class. Each cell instead only holds a fuel model number which is used to index into the VBOs.

4.2.4 Shader

The Shader class encapsulates details regarding the driving force behind the simulator. It is responsible for loading shaders from files, setting any necessary program parameters, and providing access to various uniform variables.

4.2.5 ContextData and SimulationState

As previously discussed, due to the intended purposes of real-time simulation, the need for visualization is often a side-effect. To this end, direct access to the data textures is allowed. However, information about the underlying visualization system may not be known at runtime. Given a system with multiple video outputs through separate graphics cards, the simulation data must be replicated in each graphics context's video memory. To facilitate this, the graphics data is kept separate from the simulator itself. This ContextData structure is created for and returned to each rendering context that requests it. When communicating with the simulator, a graphics context passes the structure in as an argument. It is the responsibility of the application programmer to assure that every ContextData instance is updated before advancing the simulation. Otherwise, synchronization issues may arise. Beyond graphics assets, the ContextData structure also holds a SimulationState object that stores information regarding the state of the simulation for that particular context. This object is the only data accessible by events and allows events to manipulate the simulation by queuing data modifications in the form of ignitions, moisture modifications, and fuel model modifications.

4.3 Simulator Components

While a single monolithic simulator component would have reduced code complexity, it would have also degraded modifiability and maintainability. Therefore, the simulator is broken up into several structures to better facilitate these properties. Figure 4.3 summarizes the class relationships of the simulator.

4.3.1 FuelModel and FuelMoisture

To allow for various file formats for fuel data, the simulator abstracts this data into the form of a FuelModel structure and a FuelMoisture structure. The FuelModel structure maintains data regarding an integer fuel model number such as fuel density

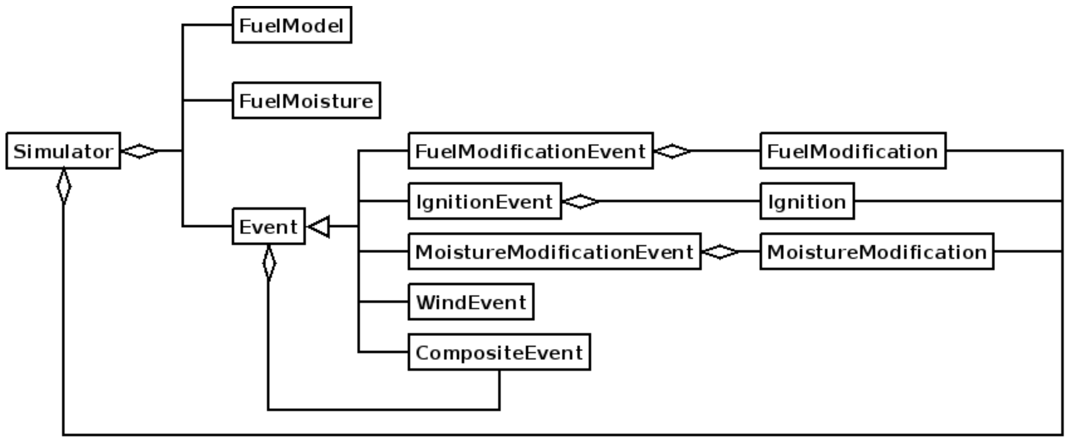

Figure 4.3: Simulator Class Hierarchy

and extinction moisture. This data is eventually uploaded and maintained in graphics memory in the form of TextureBuffers for spread rate calculations; on the other hand, the FuelMoisture structure's existence is far more ephemeral. As its name indicates, the FuelMoisture structure structure holds information regarding the moisture content for each fuel class of each integer fuel model number. These values, however, do not remain constant due to factors such as weather and human intervention; thus, they are only used to generate an initial fuel moisture map and then discarded.

4.3.2 Events

Events are the primary method of interacting with the simulation. Every event derives from an abstract Event class that contains a trigger time and a trigger function that can manipulate the SimulationState structure. As events are registered with the simulator, they are maintained in a list ordered by trigger time. When the event falls within the simulation step's time window, the event is moved to a different event list. As part of the updating stage for each context, every event in this second list is eventually triggered. To prevent some redundant rendering state setup, events in general enqueue some sort of action into the SimulationState structure. By doing so, all events of the same type, such as ignitions, can be processed together rather than, for example, setting the rendering state up for a single ignition, processing the ignition, setting the rendering state up for a fuel modification, processing that modification, and setting the rendering state up again for yet another ignition. To further increase performance, a set of flags are available to denote which pieces of data a particular event will modify. For example, an ignition event will merely modify time-of-arrival data and will have no effect on spread rates or the wind field; nothing else is recomputed. On the other hand, alteration of the wind properties will cause changes to the spread data, in which case, a flag will be raised requiring the simulator to recompute the spread data before proceeding.

In its present incarnation, four primary types of Events have been implemented. The first, an IgnitionEvent, is used to start fires. The second, a FuelModification event, is used to alter the fuel models of the landscape. Fire breaks can be approximated by altering the fuel model to some unburnable type, while the effects of other preventative measures such as forest thinning can be experimented with through the alteration of the fuel models to less dense models. The third subclass of Event is the MoistureModificationEvent, which allows for the alteration of moisture content in fuels. These events can be used to simulate climate changes as well as fire suppression methods. Finally, a WindEvent class allows for the alteration of wind properties, specifically the wind direction and wind speed. Another Event subtype, the CompositeEvent, is provided to ensure that a set of events will be triggered simultaneously.

4.4 Data Storage

Spatial inputs to the simulator come in the form of two-dimensional lattices. Because of this, they are stored as two-dimensional textures. Constants for each fuel model are stored in VertexBuffer objects that are bound as textures using a TextureBuffer. Due to the limited number of textures, these constants were packed into different components of four-component floating point buffers. They are retrieved by fetching the fuel model index for a cell and using the result to index into the buffers.

Chapter 5

Modeling and Implementation

This chapter discusses the method used to simulate the various aspects of fire spread and how they were performed on the GPU.

5.1 Fire Spread

For basic fire spread, the simulator is based on Rothermel's fire spread equations, which are based on Huygens's principle. Thus, fire spread is modeled as an ellipse expanding at some maximum rate R_{max} with some orientation ϕ and eccentricity ε based on wind and slope. Given this information, the spread rate in any arbitrary direction Θ can be computed using

$$r(\Theta) = R_{max} \frac{1.0 - \varepsilon}{1.0 - \varepsilon cos(|\phi - \Theta|)}$$

as described in [23]. Given this spread rate r, the time required for a fire to travel from one point to another point separated by a distance d is given by

$$t = d/r$$

Because the desired output is a set of rasters that can be visualized and analyzed, the simulation is modeled as a two-dimensional regular rectangular lattice in a fashion similar to that of HFire [23]. Essentially, the center of each cell is connected to the centers of each of its eight neighboring cells by a straight line of fuel. Fire can spread from a burning cell to any unburnt cell by burning the entire distance separating the cell centers. In the case of multiple lines burning towards the same center, the first line to completely burn is used to determine the time of arrival and other fire characteristics. The time of arrival logic can be summarized by Algorithm 5.1. It may be executed repeatedly until the resulting times of arrival no longer change. To account for changing spread conditions, times of arrival can only be computed up to the point when these conditions are altered; any arrivals after this point are simply discarded.

Algorithm 5.1 Time of arrival computation.

```
1: for all Surrounding Cells $c$ do
2:     $t$=timeOfArrival($c$) + spreadTime($c$, $thisCell$)
3:     timeOfArrival($thisCell$)=min(timeOfArrival($thisCell$), $t$)
4: end for
```

To achieve this logic on the GPU, the times of arrival are double-buffered with a pair of 32-bit floating point textures. A 32-bit floating point depth buffer also stores the time of arrival. An OpenGL extension allows for this depth buffer to be clamped to a range other than [0..1]. All times are initialized to the very end of the simulation. A fragment shader is executed over each cell of the simulation space. This shader executes a single iteration of Algorithm 5.1 and uses the resulting value as the fragment color as well as the fragment depth. During execution, one texture is used as the times of arrivals accessed while the other texture is used to collect the outputted values. After each iteration, the values are copied from the write texture to the read texture. Due to depth buffering, only fragments with a depth value, which contains the time of arrival, lower than the current pixel depth are actually written. As a result, only the earliest times will be written out, and only these written fragments are counted when querying the number of fragments drawn. Hence, to compute times of arrival up to a certain point in time, the acceptable depth range is set to the simulation's current time step window (earlier fragments need not be written again) and the fragment shader is executed repeatedly until no more fragments are written.

While some arrival times may exceed the threshold time and thus would not be recorded during that simulation step, that does not necessarily imply that nothing occurred between each pair of links. In fact, if that were the case, using a time step that is too short would result in no fire propagation at all since the arrival times would always exceed the time window. To prevent this problem, fractional burning is utilized. That is, while a fire may not be able to travel the entire distance between two links, it has at least burned through some of that distance; as a result, that distance is reduced by the product of the spread rate and the elapsed time.

$$\mathrm{d}=\mathrm{d}-\mathrm{r}\Delta \mathrm{t}$$

The result of this effect is that the distance will eventually be reduced to the point where the fire will spread across the link within the simulated time window. It also necessitates the modification of Line 2 of Algorithm 5.1 to

$$t = \max(\text{timeOfArrival}(c), \mathit{windowStartTime}) + \text{spreadTime}(c, \mathit{thisCell})$$

to account for the time spent partially burning the distance. Unlike HFire, link distances between each cell's neighbors is maintained rather than a normalized value representing the highest burned fraction.

On the GPU, implementation of partial burning is straightforward. The remaining distance from the center of each cell to each of its neighboring cells is stored in two textures, with the orthogonal distances in one four-component floating point texture and the diagonal distances in another. At the end of a simulation step, a fragment shader is executed with the write targets set to these distance textures. The fragment shader fetches from a set of textures the spread rates in each direction, multiplies them by the elapsed time, and outputs the result as the fragment colors. Subtractive blending results in the distances already stored in the write targets being reduced by the output fragment values.

5.2 Surface Fire

The basic spread characteristics of a wildfire are computed using Rothermel's model. Spatial information such as fuel model and terrain characteristics are stored in two-dimensional textures. Various fuel model properties such as packing ratio and fuel bed density are stored in one-dimensional TextureBuffers that are accessed using the integer fuel model used for each cell. Computation of the spread characteristics is performed by executing a fragment shader over the simulation space with the write target set to a four-component floating point texture. The outputted fragments contain the maximum spread rate of fire including the contributions of slope and wind, the eccentricity of the ellipse, the direction of maximum spread, and an intensity modifier value

$$IntensityModifier = \frac{12.6 I_R}{60.0\sigma}$$

where I_R is the reaction intensity and σ is the ratio of the fuel bed's surface area to volume. The modifier is used later in computing crowning.

5.3 Fire Acceleration

A newly ignited fire does not immediately begin to spread at its maximum rate. Instead, it accelerates towards its maximum rate over time. Fire acceleration in the GPU simulator uses a modified model of that used by FARSITE [11]. Given the maximum rate R_{max}, the spread rate at time t is given by

$$R(t) = R_{max}(1.0 - e^{-a_a t})$$

where a_a is an acceleration constant. From this equation, the time t_{max} required to achieve the maximum spread rate given the current spread rate R is given by

$$t_{max} = \frac{1.0 - \frac{R}{R_{max}}}{a_a}$$

During the fire acceleration phase, the spread rate for every burning cell is increased by

$$dR = \frac{dt}{t_{max}}(R_{max} - R_{current})$$

A fire started from an ignition will have an initial spread rate of zero and will steadily increase to its maximum rate. As the fire spreads from cell to cell, the cell inherits the current spread characteristics of the cell that ignited it. If the spread rates exceed the maximum spread rates of the newly burning cell, they are clamped instantaneously to their maximums. If, however, they are slower than the cell's maximum rates, they are accelerated by the difference dt given by

$$dt = timeOfArrival(thisCell) - max(baseTime, timeOfArrival(ignitingCell))$$

where $baseTime$ is the last time that the burn distances were updated.

The addition of fire acceleration on the GPU simulator introduces a problem which cannot be handled by the previously described depth buffering method. That method of computing spread can be reduced to a problem of determining the lowest cost (least time) to each node in a graph. The method makes a few assumptions, some of which are invalid with fire acceleration. First, the cost of each linkage traversed during the previous simulation step remain constant. This remains valid as the path traversed to that point in time was actually taken and burned. The second assumption is that the cost of each linkage is constant during a simulation step. This assumption becomes invalid for the following reason: because a newly ignited cell takes on the spread characteristics of the igniting cell, the costs going out of that cell depend on the path taken to get to the cell.

Figure 5.1 illustrates the problem. Consider the two burning cells with spread properties a and b in (a). The resulting spread characteristics after one pass through the propagation algorithm is shown in (b). The upper left cell burns into the upper middle cell before the upper right cell can, so the upper middle cell inherits a modified set of a's spread properties, a'. The following iteration (c) shows that the lower left cell will receive spread properties that originated from the upper left cell since the upper middle cell will burn to it before the lower middle cell can. However, in the next iteration (d), it is determined that the lower middle cell will burn into the upper middle cell (from b to b' to b'' to b''') before the upper left cell can burn directly into it. As a result, the cell now contains properties inherited from b, but that cell's spread rate towards the bottom left cell may be slower than the spread rate used to compute a''. Consequently, the value in the lower left is invalid. If such an event occurs using

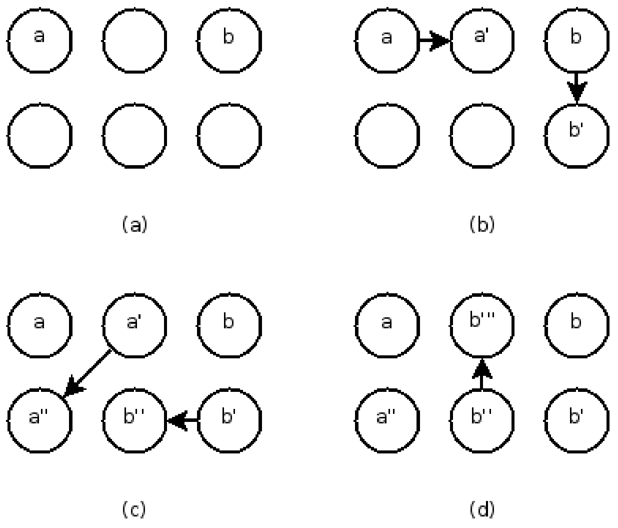

Figure 5.1: Problem introduced by fire acceleration

the depth buffering algorithm previously described, the depth value written due to a'' will be lower than the depth value written for b''', resulting in that cell incorrectly retaining a'''s characteristics.

To remedy this problem and implement fire acceleration, the depth buffering method was abandoned. Maximum spread rate properties are still only computed when some event is triggered that would alter these properties. Given these computed properties stored in a texture, the maximum spread rate in each of the eight spread directions is computed and stored in a set of two textures, one for the orthogonal directions and another for the diagonal directions. A seperate pair of textures is used to maintain the current spread rates for each cell and is initialized at the beginning of the simulation to be all zeroes. At the end of every simulation step, all burning cells have their spread rates accelerated as previously discussed.

Propagation based on burn times and times of arrival is still used, although it was modified to account for changing spread rates as a result of acceleration and rate inheritance. Spread rates are double-buffered like the time of arrival textures, and a copy from write texture to read texture for each is done at the end of the iteration. The propagation shader was modified to output the inherited and accelerated rates from the igniting cell.

Three additional textures were added. The first two store integer time stamps that denote the last time a cell was updated. If any cells surrounding the current cell were updated in the last round, the current cell is checked again to see if a lower time of arrival is possible. These textures are swapped in terms of reading and writing at the end of each iteration; no compositing is necessary since only cells that changed in the previous iteration are of concern. At the beginning of the propagation phase, all stamps are cleared to zero to force all cells to at least check once with new spread times. Cells that are updated write the next time stamp out to the write buffer. The third texture contains the texture coordinate of the cell that ignited the current cell. If the data in that cell becomes invalid, that is, if the spread properties of that cell have changed and the time stamp is equal to the previous iteration's time, the current cell's time of arrival is invalidated and the original time of arrival used at the beginning of the simulation step is used to proceed.

Because depth buffering can no longer be used, fragments must be explicitly discarded in the shader to ensure that zero fragments are eventually written in a single pass, signaling the end of the propagation step. In the fragment shader, a fragment is discarded if the current time of arrival is earlier than the start of the simulation step, if the resulting time of arrival is later than the end of the simulation step, if the resulting time of arrival is later than the current time of arrival, or if no surrounding cell was updated in the previous iteration.

5.4 Crown Fire

Whether a fire will spread into and through the crowns of trees depends on a number of factors. As with FARSITE's implementation [11], active crown fires have a different maximum spread rate than surface and passive crown fires approximated by

$$R_{max_{crown}} = 3.34 R_{10}$$

The actual maximum spread rate of a crown fire depends on the crown fraction burned, given by the equation

$$CFB = 1 - e^{-a_c(R-R_o)}$$

where

$$a_c = \frac{-ln(0.1)}{0.9(RAC - R_o)}$$

and

$$R_o = I_o\frac{R}{I_b}$$

I_o is the reaction intensity of the fire which can be obtained by multiplying the current spread rate by the intensity modifier outputted from the maximum spread property shader. I_b is the threshold intensity for a crown fire to occur and is given by

$$I_o = (0.010CBH(460 + 25.9M))^{\frac{3}{2}}$$

where CBH is the crown base height and M is the foliar moisture content. RAC is the threshold spread rate at which a crown fire is promoted from passive to active. Given the crown bulk density CBD, this threshold is given by

$$RAC = 3.0/CBD$$

Crowning is modeled in the simulation as an increase in maximum spread rate in the case of an active crown fire. Canopy height is stored in a texture, and crowning is only considered if the canopy height is greater than zero. RAC is computed with a shader and stored in a texture. Modification of the CBD texture results in the recomputation of the RAC texture. I_o is also computed with a shader and is only updated if the CBH texture is modified. Before accelerating spread rates, the current spread rates are used to test whether the fire is now actively crowning, and if so, the maximum spread rate is adjusted accordingly. This computation is done on a per direction basis.

5.5 Simulation Progression

At the beginning of the simulation, all directional spread rates are initialized to zero. Times of arrival are cleared to a user-defined end time. Burn distance textures are set to the complete distance to each neighbor.

Algorithm 5.2 outlines the flow of the simulation. Because spread properties can change due to alterations caused by events, simulation progresses in substeps between events.

Algorithm 5.2 Simulation Flow

```
1: $endTime = currentTime$
2: $startTime = lastUpdateTime$
3: while $startTime$ != $endTime$ do
4:     $stepTime = \min(endTime, nextEvent.time)$
5:     Update corrupted data()
6:     Propagate fire($startTime$, $stepTime$)
7:     Burn distances($startTime$, $stepTime$)
8:     Accelerate($startTime$, $stepTime$)
9:     Trigger next event()
10:    $startTime = stepTime$
11: end while
12: $lastUpdateTime = endTime$
```

5.5.1 Texture Summary

Table 5.1 summarizes the purposes of every texture used in the system with fire acceleration incorporated. Constant textures such as the TextureBuffers used to store constant data are not enumerated. Several of the textures have a notation of `[2]` denoting that there are actually two copies of this texture.

Texture	Purpose
slopeAspectElevation	Slope, aspect, and elevation of the terrain
fuel	Integer fuel model numbers
deadMoistures	Dead fuel moisture content (1-hour, 10-hour, 100-hour)
liveMoistures	Live fuel moisture content (herbaceous, woody)
wind	Wind direction and magnitude
spreadData	Maximum spread rate, direction, eccentricity, intensity modifier
burnDistance[2]	Length of unburned fuel in orthogonal and diagonal directions
burnRate[2]	Current spread rates in orthogonal and diagonal directions
maxBurnRate[2]	Maximum spread rates in orthogonal and diagonal directions
sourceData	Coordinate of cell that ignited this cell
timeOfArrival	Earliest ignition times
originalTimeOfArrival	Time to revert to if source data becomes corrupted
crownThreshold	Threshold intensity for crowning
crownBaseHeight	Base height of the crown (CBH)
crownBulkDensity	Density of crown bulk (CBD)
crownActiveRate	Threshold spread rate for crown spread (RAC)
canopyHeight	Height of the canopy
timeStamp[2]	Last integer iteration cell was updated during

Table 5.1: Summary of Textures

Chapter 6

Results

In order to examine the benefits and shortcomings of the GPU simulator, a set of test cases were generated. Simple conditions were used to examine the basic properties of fire spread while more complex conditions were setup to examine performance under more realistic conditions. For all figures, times of arrival are colored with a gradient mapping violet to the time of ignition and red to the simulation time at which the image was produced. The black lines indicate some time milestone, and images within the same figure use the same time milestones.

6.1 Simple Conditions

Figure 6.1 shows the simulated spread progression of an ignited fire in a completely homogeneous environment. The slope of the entire landscape was set to zero, the fuel map contained a single fuel model and crowning was disabled. Fuel moistures were also kept uniform. With zero wind, it would be expected that such conditions would result in fire spreading in a perfect circle. However, the simulation result shows fire spread in the shape of an octagon. This is explained by the limited number of directions that fire can spread in. Points lying on these directions relative to the ignition point have correct times of arrival since the distance traveled to reach them is equal to the distance between the cell and the ignition point. Every other point, however, can only be reached by traversing a set of links oriented in the allowable spread directions. In other words, the actual distance burned to get to these points is

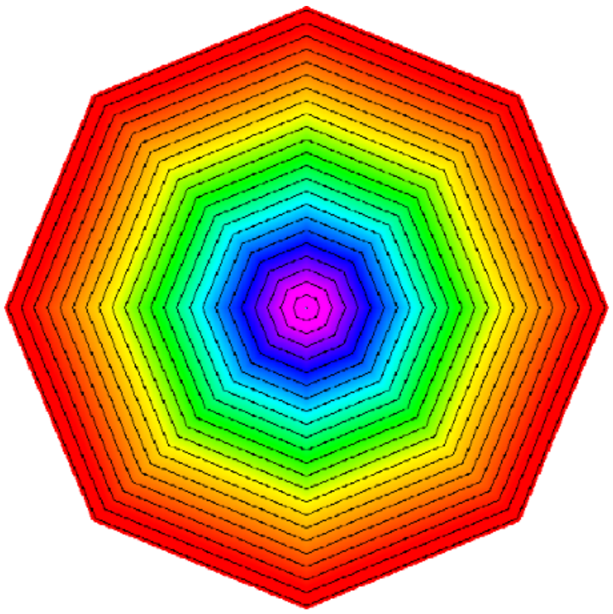

Figure 6.1: Simulated fire spread on terrain with no slope, no wind, and a uniform fuel map.

longer than the length of a line connecting the ignition point with the cell. As such, it may be concluded that this particular model will underestimate the spread of fire.

Figure 6.2 summarizes the results of simulation of fire given various uniform slopes facing west. Again, fuels and moistures were kept consistent and no wind was added. As the slope increases, the eccentricity of the fire ellipse increases as expected. Additionally, the overall spread rate increased. Fire spread upslope was relatively faster than fire spread downslope, and the difference between the two increased with the slope percentage. At the same time, the underprediction due to the limited spread directions becomes more apparent.

Figure 6.3 drops the slope back to zero and instead varies the wind speed moving eastward. The effects are similar to the effects of increasing slope in that an increase in wind speed increased the eccentricity of the fire ellipse as well as the spread rate. Again, underestimation can be observed, although this effect seems to be mitigated by oscillating the wind direction by some amount during each update. Figure 6.4 shows the effect of oscillating the wind vector by various amounts using different periods.

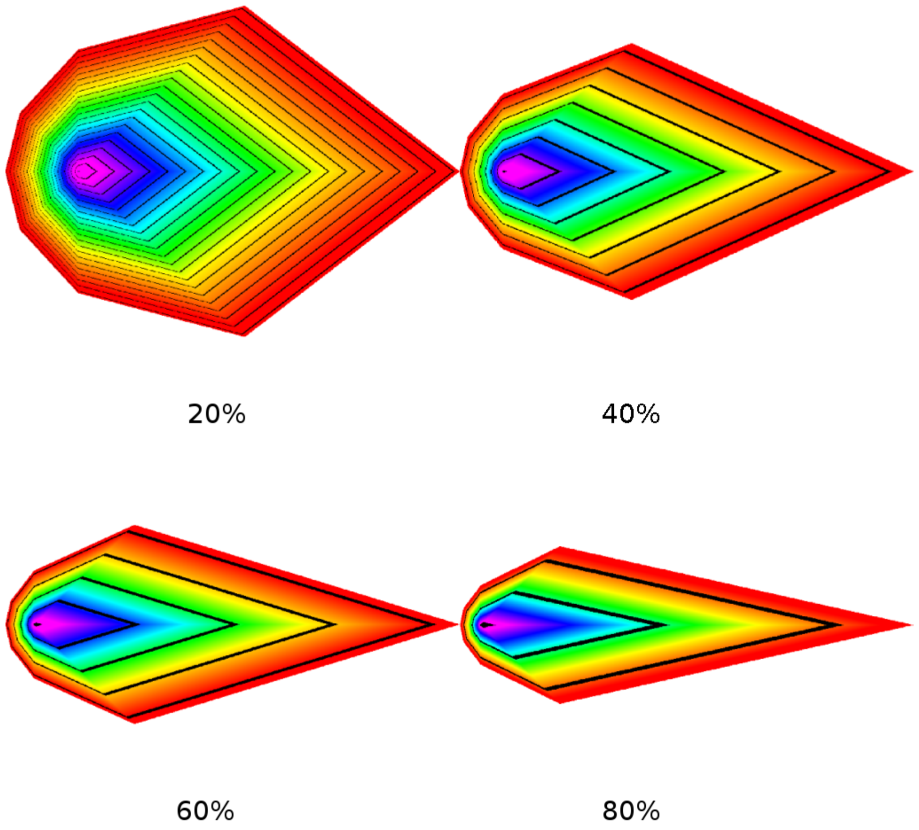

Figure 6.2: Simulated fire spread on terrain with various slopes, no wind, and a uniform fuel map.

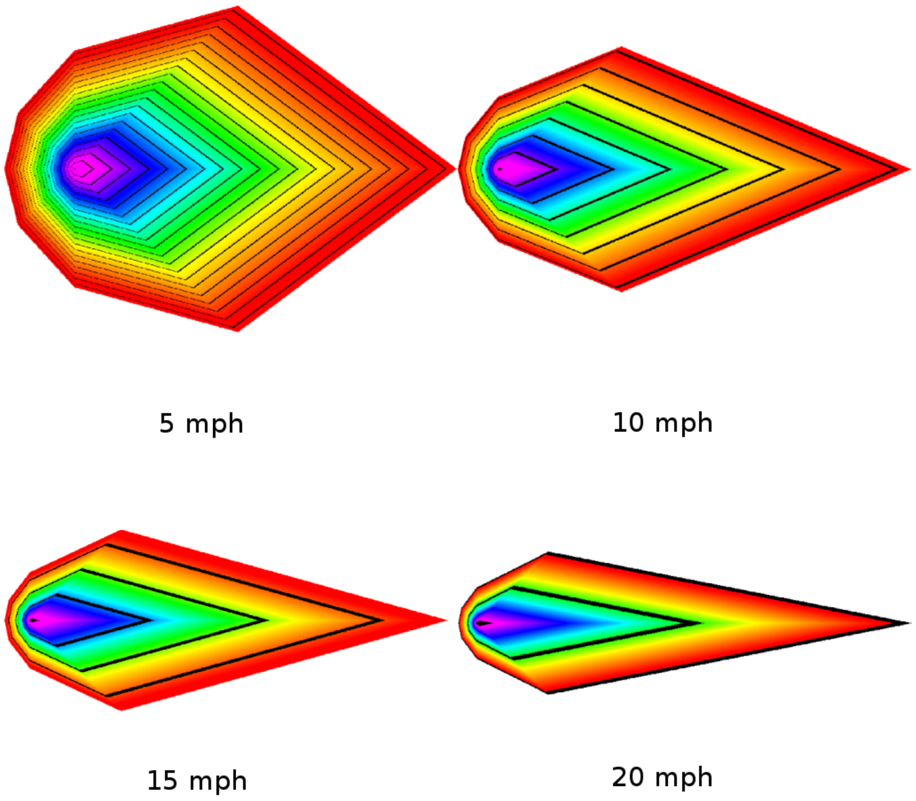

Figure 6.3: Simulated fire spread on terrain with no slopes, various wind speeds, and a uniform fuel map.

The shapes on the left side were generated by allowing the wind to oscillate with a period of 4000π seconds while those on the right used a period of 8000π seconds. The shapes on top allow the direction to oscillate up to 20 degrees in either direction while the lower shapes allow an absolute difference of 40 degrees. These variables were inputted into a sine function to determine the actual oscillation at any point in time.

The effects of fire acceleration is illustrated in Figure 6.5. The upper left fire used an acceleration constant a_a of 0.1 while the upper right used a constant of 0.01. The lower left and lower right fires used constants of 0.001 and 0.0001, respectively. At ignition, a 10 mph is set blowing east. The direction is rotated 90 degrees coun-

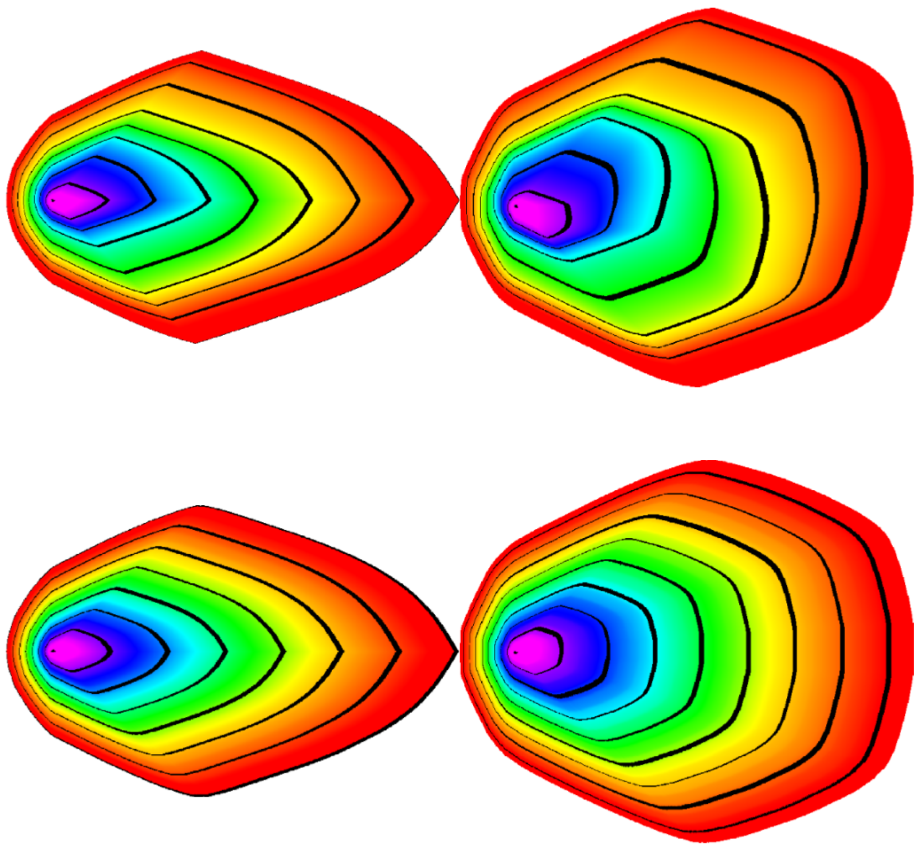

Figure 6.4: Simulated fire spread on terrain with no slopes, a steady wind speed, an oscillating wind direction, and a uniform fuel map.

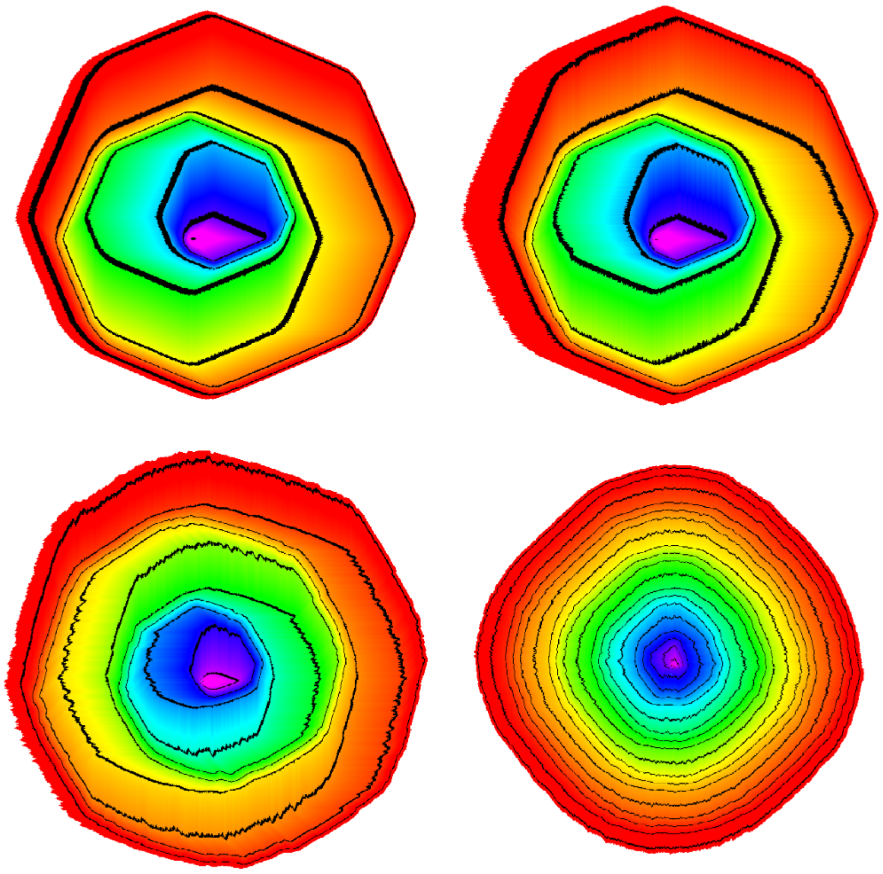

Figure 6.5: Effects of fire acceleration on flat terrain with a rotating wind using a uniform fuel map.

terclockwise every 3000 simulation seconds. As a_a decreases, the time required to accelerate towards the maximum spread rate increases. As a result, the effects of the wind direction change become less apparent as a_a decreases.

While all of the previous results assumed a single fuel model across the entire simulation space, in reality this is rarely the case. When a fire progresses from an area with one fuel model to an area with a different fuel model, spread rates change, altering the overall progression of the fire. These effects were examined by generating a fuel map containing a single fuel model in all areas except for a large rectangle of some other fuel type. Figure 6.6 shows the effects of an unburnable fuel type being inserted into the simulation space. Fire spread stops completely at the boundary of

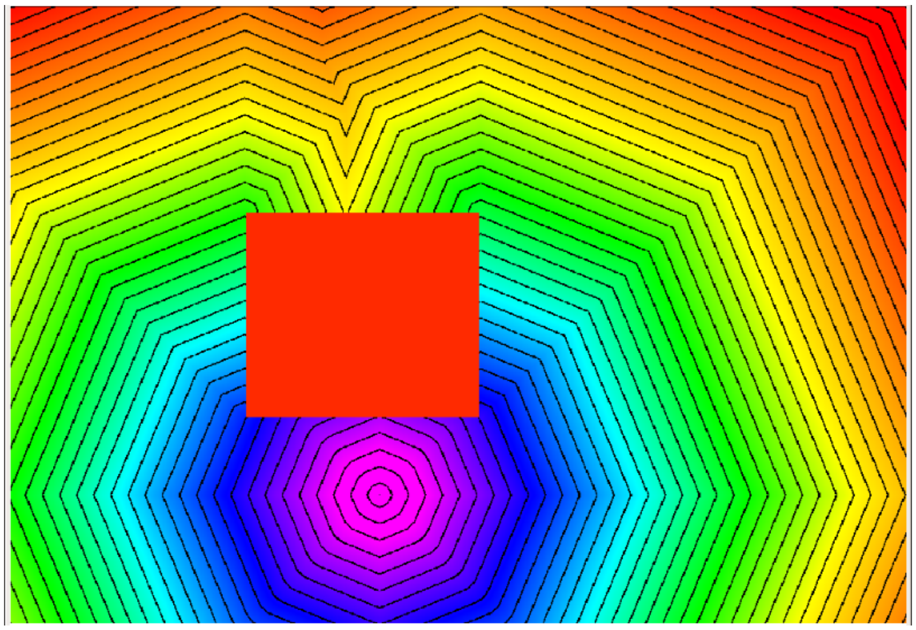

Figure 6.6: Simulation of fire spread around an unburnable area.

the fuel and must spread around it, delaying its spread to the other side of the barrier.

Figure 6.7 shows the effects of fire spreading into an area that burns faster than the surrounding area. The fire shape becomes distorted as the fire burns through the area. Figure 6.8 shows the effect of swapping the fuel models. While the fire slowly progresses into the area from the south side, it also surrounds the entire slow area and slowly burns inward from all directions, forming a ring of fire.

6.2 Complex Conditions

Although the simple conditions are useful for examining the individual characteristics of the fire simulator, its performance cannot be realistically tested with such conditions. Therefore, performance measurements are done on a more complex example. A fire is simulated using data from Kyle Canyon, Nevada. A satellite image of the area in question is shown in Figure 6.9. A graphical representation of the fuel map used is shown in Figure 6.10.

Figure 6.11 shows the result of a run with no wind and no crown fire acceleration.

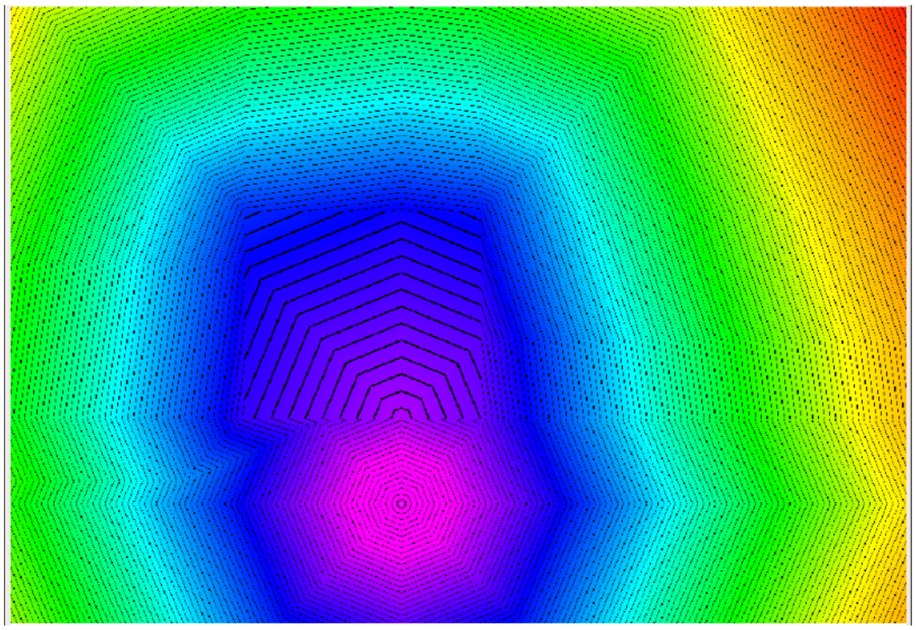

Figure 6.7: Simulation of fire spread through a faster burning area.

Even without these effects, the complex shape of the fire presents higher probabilities of the fire acceleration problem occurring and requiring additional propagation passes to correct it. Figure 6.12 shows the result of a run with an eastbound wind. Such winds increase the spread rates in various directions, increasing the probability that a fire may spread from one cell to two or more cells in a single simulation step, again requiring more propagation passes to account for. Figure 6.13 illustrates the effect of adding crown fire acceleration to the simulation without any wind. Again, the increased spread rates may result in additional propagation passes. Figure 6.14 shows the effect of both wind and crown acceleration on the result, increasing the spread rates even higher and possibly requiring more propagation passes.

For measurements, this complex data set was used for two experiments. For both experiments, an ignition is placed at the exact center of the simulation space. A 10 mph wind is used. Its orientation is rotated by 1 degree every 100 simulation seconds, and all of these wind events are initially enqueued. The timer is then started. The simulation is then updated with some time step until it has reached 100,000 simulation seconds. The timer is then stopped. Statistics on the number of updates and the

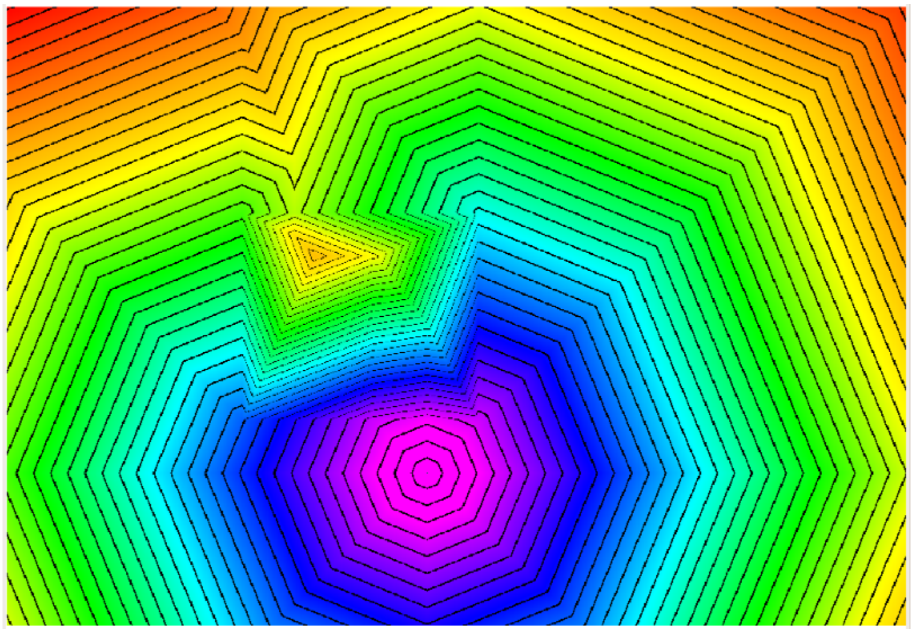

Figure 6.8: Simulation of fire spread through a slower burning area.

total simulation time are collected. The time per update is then computed as the total simulation time divided by the number of updates.

All experiments were executed on a computer with an Intel Core 2 Quad with four cores running at 2.4GHz and 3GB of memory. The graphics card used was an NVIDIA 9800GTX with 128 stream processors and 512 MB of video memory.

The first experiment keeps the number of cells constant at 576,752 cells. Time step size is varied from 50 seconds to 600 seconds in increments of 50 seconds. The results are shown in Figure 6.15. The higher execution times whenever the time step is not a multiple of 100 can be explained by the fact that a wind event occurs at every 100 second mark. Each of these events divides the simulation into substeps. When a time step is not a multiple of 100, these substeps will periodically be divided again, increasing the total number of substeps.

While the total execution time increases as the time step decreases, this increase may be worth it in order to allow for increased user interactivity. Figure 6.16 shows the average time per update. With smaller time steps, this time decreases, likely

Figure 6.9: Kyle Canyon, Nevada

due to fewer propagation iterations per update. This also suggests that there is a considerable overhead to calling the update function repeatedly.

The second experiment maintained a constant time step of 500 seconds while varying the number of cells in the simulation space. This value was varied by altering the size of each cell while maintaining the same simulation extents. It should be noted that some data, such as the fuel model texture, are not scaled according and are kept in their raw form. As a consequence, any gains or losses due to cache coherence is not accounted for.

Figure 6.17 shows the execution times of the second experiment. Increasing the number of cells appears to increase the execution time quadratically. The overall increase in execution time can be explained by several factors. First, while the number of cells increased, their distances decreased. As a result, the likelihood of a fire making more than one jump in a single iteration is increased given a static time step. Second, an increase in the number of cells increases the number of fragments that must be rasterized and processed. Finally, an increase in cells increases the size of the textures

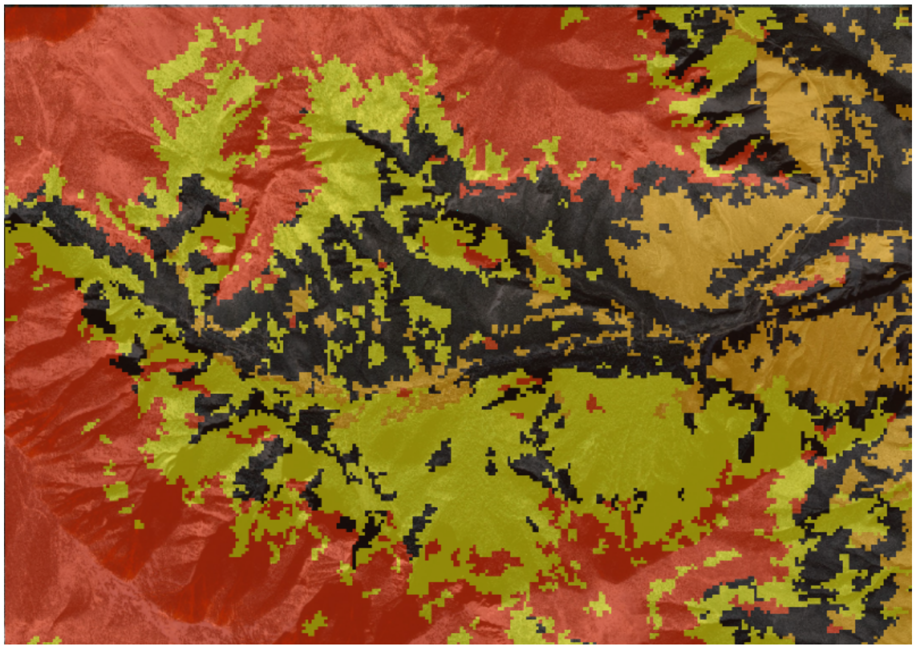

Figure 6.10: Fuel map used in the Kyle Canyon simulation

used to store the spread data which in turn may result in a higher frequency of cache misses. Unlike the first experiment, the computation time per update increases along with the total execution time since the number of updates remains constant due to the constant time step. Figure 6.18 reflects this.

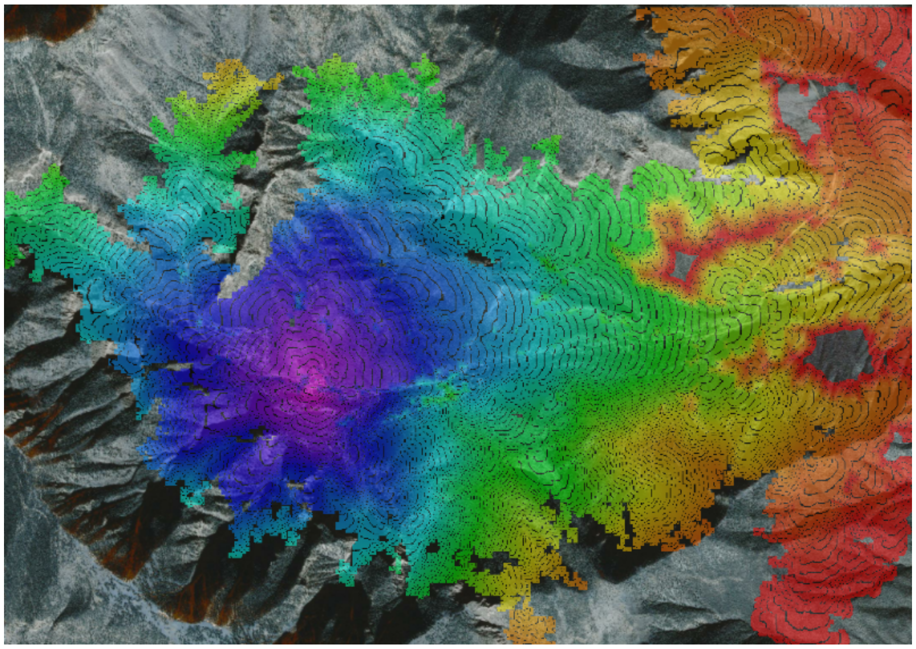

Figure 6.11: Simulation run with no wind and no crown acceleration.

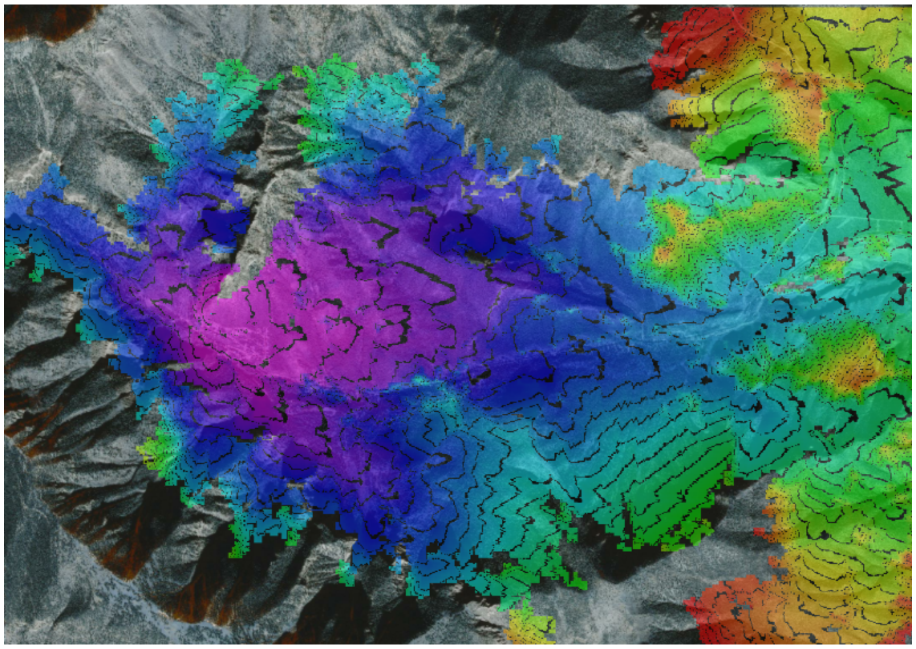

Figure 6.12: Simulation run with eastbound wind and no crown acceleration.

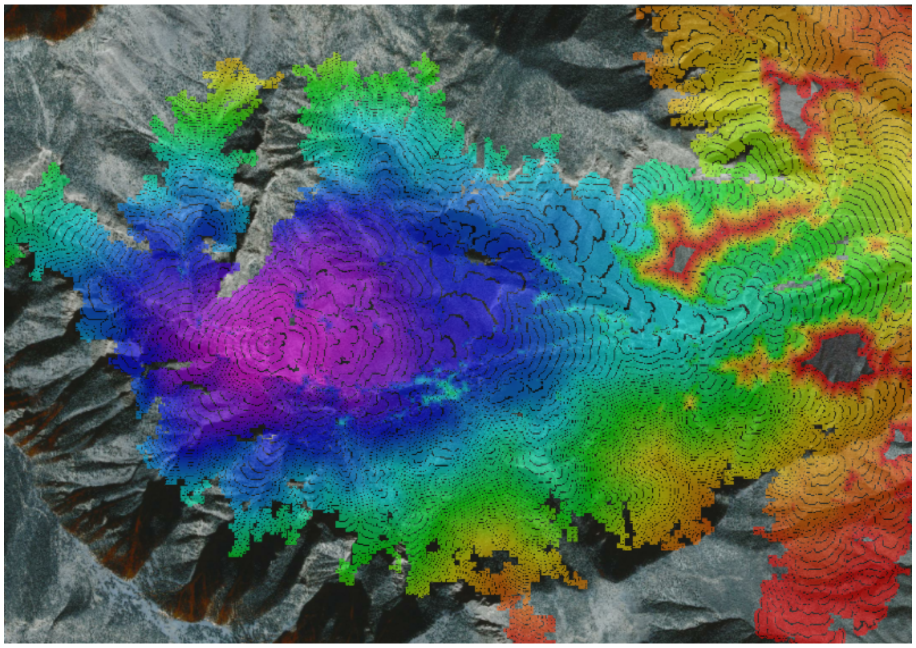

Figure 6.13: Simulation run with no wind and crown acceleration.

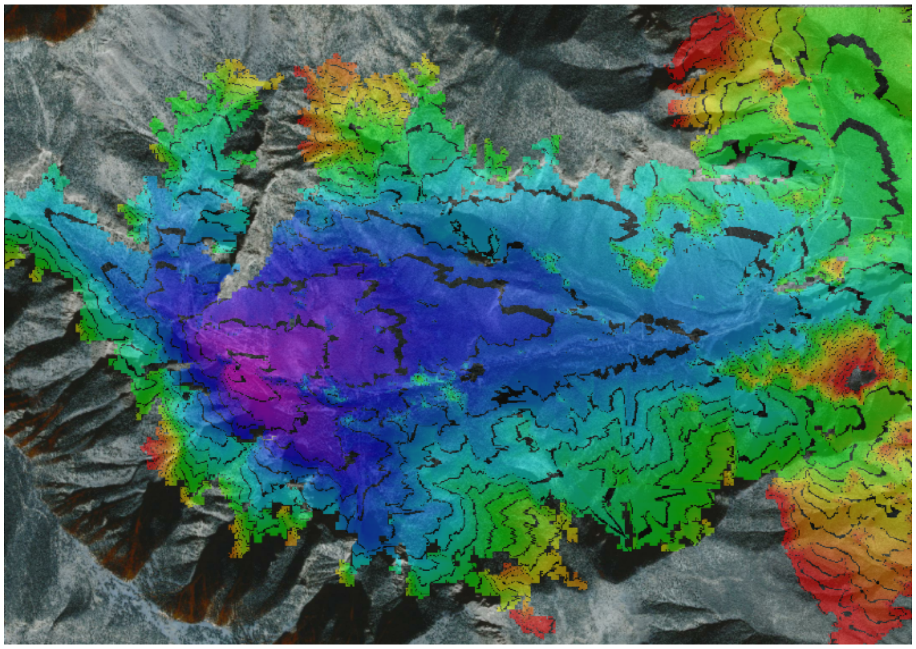

Figure 6.14: Simulation run with eastbound wind and crown acceleration.

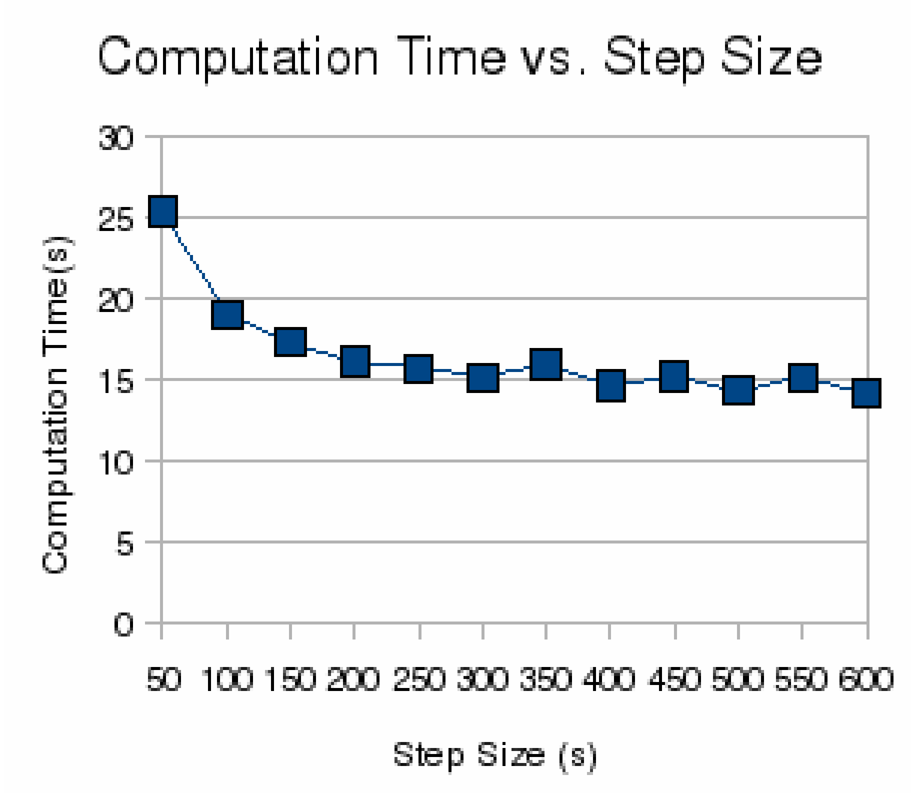

Figure 6.15: Execution times with 576,752 cells simulated.

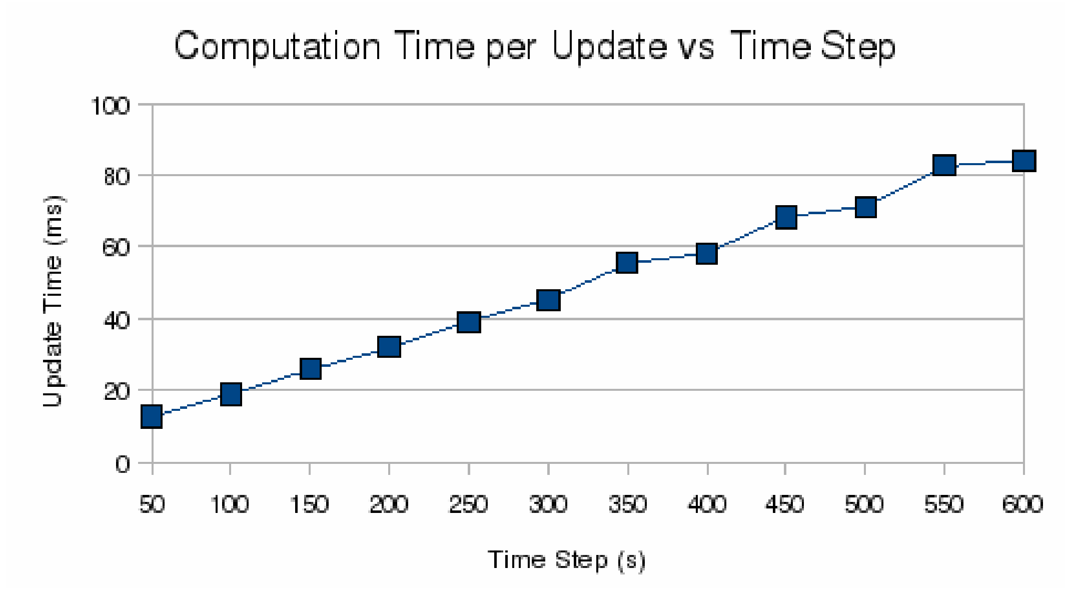

Figure 6.16: Average update time with 576,752 cells simulated.

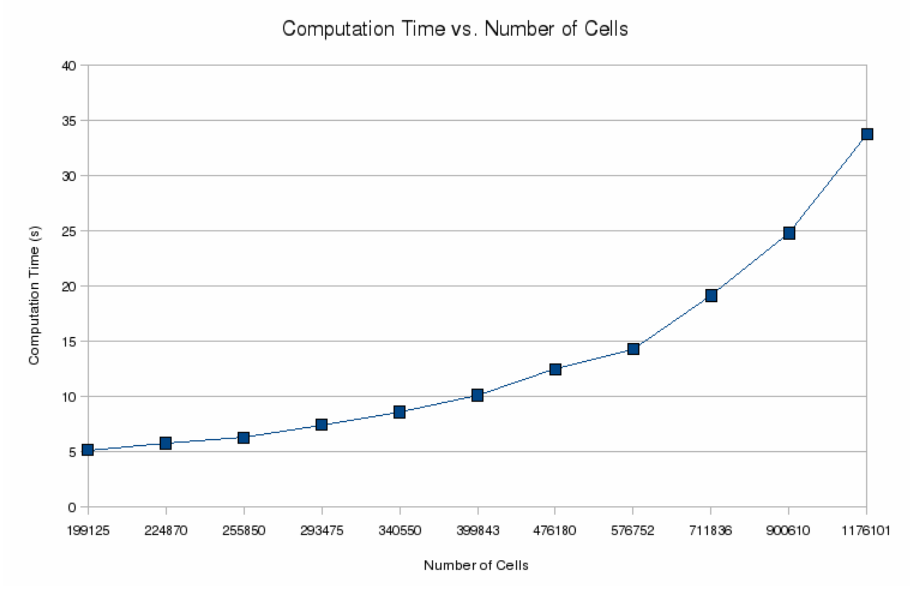

Figure 6.17: Execution times with a timestep of 500 seconds.

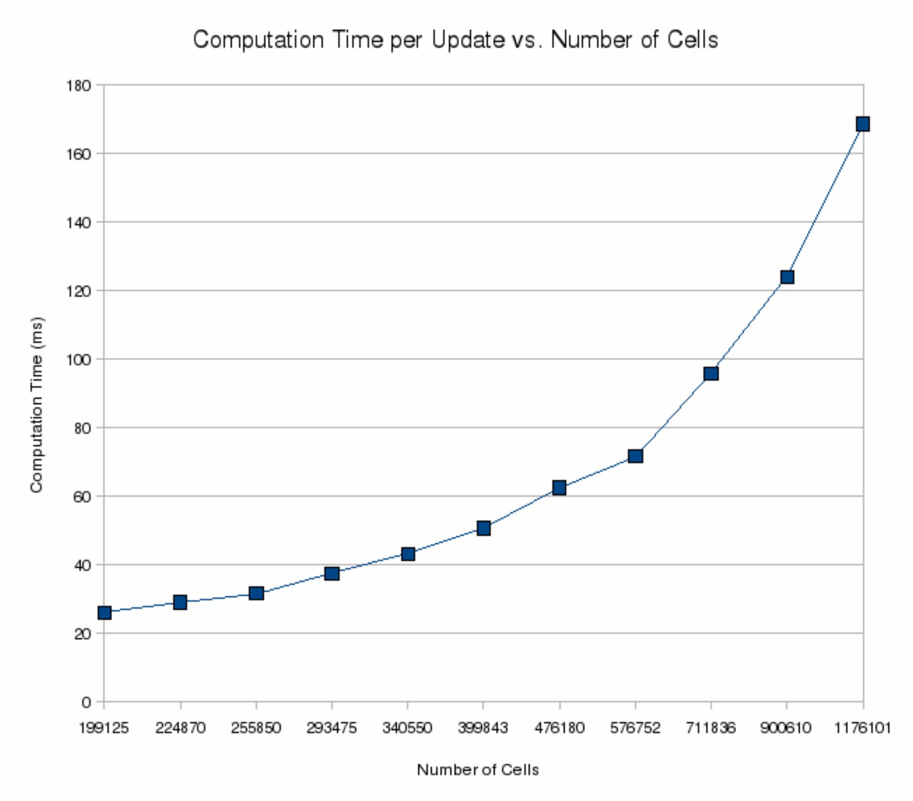

Figure 6.18: Average update times with a timestep of 500 seconds.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

Wildfire modeling is a field rife with complexities. At even the simplest level, simulation of wildfire can be a time-consuming endeavor, reducing its effectiveness as a predictive tool or a real-time training tool. Increasing the computational power by increasing the number of CPUs in a machine or machines in order to move towards this goal of rapid feedback may not be particularly cost-effective. On the other hand, the increasing power and low cost of commodity graphics cards offers a possible alternative. However, special techniques must be employed to harness this computational power.

Techniques to achieve wildfire simulation on the GPU were presented. A method for essentially determining shortest paths in a lattice with static costs using depth buffering was discussed and applied to spreading fire with no acceleration. Fire acceleration was examined along with the problem that comes with adding it to the simulator. A solution to that problem was presented, allowing one to find the shortest path in the same lattice with dynamic path costs.

The resulting simulator is capable of simulating the spread of surface and crown fires. It incorporates the concept of fire acceleration and allows the calling application to introduce events that alter the progression of the simulation. The simulator does the bulk of its work on the GPU and provides the calling application with access to its data textures for visualization purposes. Mechanisms for co-simulation across multiple GPUs is also provided.

Initial results showed that fire propagation using this lattice setup resulted in underestimation. Some of the underestimation in the case of winds seemed to be mitigated by oscillating the wind direction, which suggests that these distortions may be resolved by introducing some sort of noise. Still, fire spread behaved as expected with varying winds, slopes, and fuel models, and performance measurements indicate that depending on the resolution and step size, the simulator may be used as a component of a real-time simulation application. That being said, there are several areas where this work can be expanded.

7.2 Future Work

Spotting

As a wildfire progresses, burning embers can be lofted into the air by wind and smoke. Should these embers land on some sort of burnable object, these embers can ignite more fires. This effect is particularly of interest since these embers can jump over fire barriers that would have stopped a surface fire.

Preliminary work suggests that this effect can be modeled by sweeping the data textures using a geometry shader and emitting embers into the air which can then be controlled by several existing models, by some wind model, or some combination of the two.

Weather Simulation

The fuel moistures in the simulation are currently static. Weather patterns can have an impact on these moistures as well as other spread characteristics. Integration of a weather simulator would increase the utility and accuracy of the simulation. Such a simulator could also account for weather and moisture changes caused by the wildfire spreading in some sort of feedback loop.

Validation

While some basic properties of the simulator's fire spread model were discussed in this work, comparisons of the results to actual fires or the results of other simulators was not performed. Such comparisons would be necessary to not only validate the model but also determine any necessary refinements.

GPU Clustering and CUDA

Although a single GPU provides a considerable amount of processing power, it may not be enough. Extending the simulator to run on multiple GPUs in parallel is a logical path to obtaining the necessary power. Such an effort would likely introduce considerable complications in regards to the amount of data that would need to be shared and synchronized across GPUs.

While the simulator runs on the GPU, it currently does so by employing a set of tricks through the OpenGL API to force computation of the simulation. NVIDIA's CUDA [36] may provide speed increases by providing more direct access to a GPU's processing cores. At the same time, some of the graphics functionality, such as interpolation, employed by the simulator may become unavailable as the result of such a switch and must also be resolved.

Improved Fractional Burning

The limited number of direct cell connections can cause a rather large amount of underprediction. To a point, this can be ameliorated by increasing the number of direct connections at the cost of memory and computational complexity. A more scalable solution would involve determining any possible contribution of burned distances to links other than the one presently being burned; that is, a burning ellipse expanding into another cell horizontally will at least burn away some component in every other direction.

Interactive Simulation Tool

The ability to rapidly set up a simulation environment and try various scenarios would be of great use not only for model development but also for training purposes. The simulator itself can run in real-time, but a user-friendly tool geared towards training firefighters to orchestrate fire suppression strategies is needed to truly extend its applicability. In the same vein, user studies would need to be done to evaluate the effectiveness of such tools compared to more traditional training methods.

Bibliography

[1] H. E. Anderson. Predicting wind-driven wild land fire size and shape. Technical report, US Forest Service, 1983.

[2] P. L. Andrews. BEHAVE: fire behavior prediction and fuel modeling system - burn subsystem, part 1. Technical report, U.S. Forest Service, 1986.

[3] P.L. Andrews. BehavePlus fire modeling system: Past, present, and future. In *Proceedings of 7th Symposium on Fire and Forest Meteorological Society*, page J2.1, 2007.

[4] P. Billing. Otways fire no 22 – 1982/83. Technical report, Department of Sustainability and Environment, 1983.

[5] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for GPUs: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.

[6] Canadian Interagency Forest Fire Centre. PROMETHEUS. `http://www.firegrowthmodel.com/index.cfm`, May 2008. Accessed December 5th, 2008.

[7] J.R. Coleman and A.L. Sullivan. A real-time computer application for the prediction of fire spread across the australian landscape. *Simulation*, 67(4):230, 1996.

[8] Keenan Crane, Ignacio Llamas, and Sarah Tariq. Real-time simulation and rendering of 3D fluids. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 30. Addison Wesley Professional, August 2007.

[9] Randima Fernando and Mark J. Kilgard. *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[10] M. A. Finney. An overview of FlamMap fire modeling capabilities. In *Fuels Management - How to Measure Success: Conference Proceedings*, pages 213–220, 2006.

[11] M.A. Finney. FARSITE: Fire area simulator-model. development and evaluation. Technical report, USDA Forest Service, 1998.

[12] M.A. Finney. Spatial modeling of post-frontal fire behavior. Technical report, Rocky Mountain Research Station, 1999.

[13] D. G. Green, A. M. Gill, and I. R. Noble. Fire shapes and the adequacy of fire-spread models. *Ecological Modelling*, 20(1):33 – 45, 1983.

[14] Gold Standard Group. OpenGL Shading Language. `http://www.opengl.org/documentation/glsl/`. Accessed December 4th, 2008.

[15] Gary J. Katz and Jr Joseph T. Kider. All-pairs shortest-paths for large graphs on the GPU. In *GH '08: Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics hardware*, pages 47–55, Aire-la-Ville, Switzerland, Switzerland, 2008. Eurographics Association.

[16] Emmett Kilgariff and Randima Fernando. The GeForce 6 series GPU architecture. In *SIGGRAPH '05: ACM SIGGRAPH 2005 Courses*, page 29, New York, NY, USA, 2005. ACM.

[17] Lutz Lata. Building a million particle system. In *Proceedings of the Game Developers Conference 2004*, pages 54–60, 2004.

[18] E. Lindholm, J. Nickolls, S. Oberman, and J. Montrym. NVIDIA Tesla: A unified graphics and computing architecture. *Micro, IEEE*, 28(2):39–55, 2008.

[19] David Luebke, Mark Harris, Jens Krüger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley, and Aaron Lefohn. GPGPU: general purpose computation on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, page 33, New York, NY, USA, 2004. ACM.

[20] David Luebke and Greg Humphreys. How GPUs work. *Computer*, 40(2):96–100, 2007.

[21] Finney M.A. Fire growth using minimum travel time methods. *Canadian Journal of Forest Research*, 32:1420–1424(5), 2002.

[22] M. Macedonia. The GPU enters computing's mainstream. *Computer*, 36(10):106, 2003.

[23] M. E. Morais. Comparing spatially explicit models of fire spread through chaparral fuels: A new algorithm based upon the rothermel fire spread equation. Master's thesis, University of California, Santa Barbara, 2001.

[24] Douglas C. Morton, Megan E. Roessing, Ann E. Camp, and Mary L. Tyrrell. Assessing the environmental, social, and economic impact of wildfire. Technical report, The Global Institute of Sustainable Forestry, 2003.

[25] Nvidia Corporation. NVIDIA Tesla: GPU Computing Technical Brief. `http://www.nvidia.com/docs/IO/43395/Compute_Tech_Brief_v1-0-0_final__Dec07.pdf`. Accessed December 3rd, 2008.

[26] Nvidia Corporation. OpenGL floating point depth buffer extension. `http://developer.download.nvidia.com/opengl/specs/GL_NV_depth_buffer_float.txt`. Accessed December 3rd, 2008.

[27] Nvidia Corporation. OpenGL floating point texture extension. http://developer.download.nvidia.com/opengl/specs/GL_ARB_texture_float.txt. Accessed December 3rd, 2008.

[28] Nvidia Corporation. OpenGL framebuffer object extension. http://www.nvidia.com/dev_content/nvopenglspecs/GL_EXT_framebuffer_object.txt. Accessed December 3rd, 2008.

[29] Nvidia Corporation. OpenGL geometry shader extension. http://developer.download.nvidia.com/opengl/specs/GL_EXT_geometry_shader4.txt.

[30] Nvidia Corporation. OpenGL integer texture extension. http://developer.download.nvidia.com/opengl/specs/GL_EXT_texture_integer.txt. Accessed December 3rd, 2008.

[31] Nvidia Corporation. OpenGL occlusion query extension. http://www.nvidia.com/dev_content/nvopenglspecs/GL_ARB_occlusion_query.txt. Accessed December 3rd, 2008.

[32] Nvidia Corporation. OpenGL shader model 4 extension. http://developer.download.nvidia.com/opengl/specs/GL_EXT_gpu_shader4.txt. Accessed December 3rd, 2008.

[33] Nvidia Corporation. OpenGL texture buffer object extension. http://developer.download.nvidia.com/opengl/specs/GL_EXT_texture_buffer_object.txt. Accessed December 3rd, 2008.

[34] Nvidia Corporation. OpenGL transform feedback extension. http://developer.download.nvidia.com/opengl/specs/GL_NV_transform_feedback.txt. Accessed December 3rd, 2008.

[35] Nvidia Corporation. OpenGL vertex buffer object extension. http://www.nvidia.com/dev_content/nvopenglspecs/GL_ARB_vertex_buffer_object.txt. Accessed December 3rd, 2008.

[36] Nvidia Corporation. *NVIDIA CUDA Compute Unified Device Architecture - Programming Guide*, 2007.

[37] Lars Nyland, Mark Harris, and Jan Prins. Fast n-body simulation with CUDA. In Hubert Nguyen, editor, *GPU Gems 3*, chapter 31. Addison Wesley Professional, August 2007.

[38] OpenGL.org. OpenGL fragment shader extension. http://www.opengl.org/registry/specs/ARB/fragment_shader.txt. Accessed December 3rd, 2008.

[39] OpenGL.org. OpenGL shader object extension. http://www.opengl.org/registry/specs/ARB/shader_objects.txt. Accessed December 3rd, 2008.

[40] OpenGL.org. OpenGL vertex shader extension. http://www.opengl.org/registry/specs/ARB/vertex_shader.txt. Accessed December 3rd, 2008.

[41] E. Pastor, L. Zrate, E. Planas, and J. Arnaldos. Mathematical models and calculation systems for the study of wildland fire behaviour. *Progress in Energy and Combustion Science*, 29(2):139 – 153, 2003.

[42] G.L.W. Perry. Current approaches to modelling the spread of wildland fire: a review. *Progress in Physical Geography*, 22(2):p222 – 245, 1998.

[43] G.D. Richards. An elliptical growth model of forest fire fronts and its numerical solution. *International Journal for Numerical Methods in Engineering*, 30(6):1163–1179, 1990.

[44] R. C. Rothermel. A mathematical model for predicting fire spread in wildland fuels. Technical report, U.S. Forest Service, 1972.

[45] R. C. Rothermel. Predicting behavior and size of crown fires in the northern rocky mountains. Technical report, U.S. Forest Service, 1991.

[46] Nicolas Sardoy, Jean-Louis Consalvi, Bernard Porterie, and A. Carlos Fernandez-Pello. Modeling transport and combustion of firebrands from burning trees. *Combustion and Flame*, 150(3):151 – 169, 2007.

[47] Mark Segal and Kurt Akeley. The design of the OpenGL graphics interface. Technical report, Silicon Graphics Computer Systems, 1994.

[48] R. D. Stratton. Assessing the effectiveness of landscape fuel treatments on fire growth and behavior. *Journal of Forestry*, 102(7):32–40.

[49] David Tarditi, Sidd Puri, and Jose Oglesby. Accelerator: using data parallelism to program GPUs for general-purpose uses. In *ASPLOS-XII: Proceedings of the 12th international conference on Architectural support for programming languages and operating systems*, pages 325–335, New York, NY, USA, 2006. ACM.

[50] C.E. Van Wagner. Conditions for the start and spread of crown fire. *Canadian Journal of Forestry Research*, 7(1):23–40, 1977.

[51] J.W. Wagtendonk. Use of a deterministic fire growth model to test fuel treatments. *Sierra Nevada Ecosystem Project: Final Report to Congress*, II, 1996.

[52] J. P. Woycheese and P. J. Pagni. Brand lofting above large-scale fires. In *International Conference on Fire Research and Engineering Proceedings*, pages 137–150, 1998.

[53] John P. Woycheese, Patrick J. Pagni, and Dorian Liepmann. Brand propagation from large-scale fires. *Journal of Fire Protection Engineering*, 10(2):32–44, 1999.