REAL-TIME RENDERING OF BURNING OBJECTS IN VIDEO GAMES

Dhanyu Eshaka Amarasinghe

Dissertation Prepared for the Degree of

DOCTOR OF PHILOSOPHY

UNIVERSITY OF NORTH TEXAS

August 2013

APPROVED:

Ian Parberry, Major Professor
Armin Mikler, Committee Member
Robert Renka, Committee Member
Robert Akl, Committee Member
Paul Tarau, Committee Member
Barrett Bryant, Chair of the Department of Computer Science and Engineering
Costas Tsatsoulis, Dean of the College of Engineering
Mark Wardell, Dean of the Toulouse Graduate School

Amarasinghe, Dhanyu Eshaka. Real-Time Rendering of Burning Objects in Video Games. Doctor of Philosophy (Computer Science), 82 pp., 27 figures, references, 57 titles.

In this research, I focus on fire simulations and its deformation process towards various virtual objects. In most game engines model loading takes place at the beginning of the game or when the game is transitioning between levels. Game models are stored in large data structures. Since changing or adjusting a large data structure while the game is proceeding may adversely affect the performance of the game. Therefore, developers may choose to avoid procedural simulations to save resources and avoid interruptions on performance.

I introduce a process to implement a real-time model deformation while maintaining performance. It is a challenging task to achieve high quality simulation while utilizing minimum resources to represent multiple events in timely manner. Especially in video games, this overwhelming criterion would be robust enough to sustain the engaging player's willing suspension of disbelief. I have implemented and tested my method on a relatively modest GPU using CUDA. My experiments conclude this method gives a believable visual effect while using small fraction of CPU and GPU resources.

Copyright 2013

by

Dhanyu Eshaka Amarasinghe

ACKNOWLEDGMENTS

I dedicate my dissertation work to my family with a special feeling of gratitude to my loving parents, late Mr. Somaratne Amarasinghe and Kamala Amarasinghe for their proper guidance and encouragement to show me the importance of proper education.

I extend my dedication to my sister, Maulie Happawana and her husband Professor Gemunu Happawana who never left my side and influence me for the higher education. Including my brothers, Professor Rajee Amarasinghe, Dr. Nuditha Amarasinghe and their families for the unconditional support and guidance for my success. Not to forget my late brother Sujai Amarasinghe.

I also dedicate this dissertation to my loving wife Dr. Nishani Amarasinghe and the baby 'Panda' to come with full of excitement.

I had a perfect opportunity to work with Professor Ian Parberry, the best adviser that one could wish for these long years of doing research. His excellent guidance, patience and support are truly an inspiration to succeed in this challenging task.

I would like to thank Professor Armin Mikler who convince me to proceed with PhD by clarifying the worth of such achievement. I also like to grasp this opportunity to thank Professor Robert Renka, who taught me the first graphics class and made me falling love with the world of graphics programming. I like to thank rest of the committee and their support to fulfill my achievement.

I would like to thank all my colleagues at Laboratory for Recreational Computing (LARC) and all the friends and members of the Sri Lankan Students association of University of North Texas (SLSA).

TABLE OF CONTENTS

Page

ACKNOWLEDGMENTS iii

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER 1 INTRODUCTION 1

1.1. Hypothesis 3

1.2. Contribution 3

1.3. Overview 4

CHAPTER 2 GPU PROGRAMMING 6

2.1. Hardware Performance and Comparison 6

2.2. GPU Memory Architecture 8

2.3. OpenGL Interoperability with VBO 10

2.4. GPU Implementation with CUDA 12

CHAPTER 3 REAL-TIME RENDERING OF BURNING OBJECTS 13

3.1. Overview 13

3.2. Previous Work 14

3.3. The Heat Boundary 15

3.4. Internal Deformation 17

3.5. Structural Deformation 20

3.5.1. Model Subdivision 21

3.5.2. Rotate Around the Midpoint 22

3.5.3. Vertex Mapping 24

3.5.4. Surface Removal 25

3.6. Flame Distribution 26

3.7. Optimization 26

3.7.1. Heat Boundary Optimization 27

3.7.2. Time Division Optimization 27

3.7.3. Block Division Optimization 27

3.7.4. Pre Loading Optimization 28

3.7.5. CUDA Optimization 28

3.8. Results 29

CHAPTER 4 REAL-TIME RENDERING OF BURNING LOW-POLYGON OBJECTS 31

4.1. Overview 31

4.2. Previous Work 32

4.3. Introduction to GAMeR 33

4.4. Subdivision Techniques 34

4.4.1. Refinement Patterns and Properties 34

4.4.2. Barycentric Points and Heat Boundary 36

4.4.3. Deformation 39

4.5. Level Sets and Distance 40

4.6. Experiments and Results 41

CHAPTER 5 REAL-TIME RENDERING OF BURNING SOLID OBJECTS 43

5.1. Overview 44

5.2. Previous Work 44

5.3. Internal Deformation 45

5.3.1. The Heat Boundary 45

5.3.2. The Deformation Process 46

5.3.3. Inward Contraction Displacement 47

5.3.4. Vertex Displacement 49

5.4. Structural Deformation 50

5.5. Results and Optimization 51

CHAPTER 6 REAL-TIME RENDERING OF MELTING OBJECTS 53

6.1. Overview 53

6.2. Previous Work 53

6.3. Internal Deformation 55

6.3.1. The Heat Boundary 56

6.3.2. The Deformation Process 57

6.3.3. Polygonal Melting Properties 57

6.3.4. Base Point Movement 60

6.4. Structural Deformation 61

6.5. Results 62

CHAPTER 7 CONCLUSION 63

7.1. Summary 63

7.2. Hypothesis 64

APPENDIX A BASIC DATA STRUCTURES 65

APPENDIX B GPU CODE STRUCTURES 69

APPENDIX C CUDA GPU IMPLEMENTATION 72

BIBLIOGRAPHY 77

LIST OF TABLES

Page

Table 3.1. Designer set constants. 18

Table 4.1. Adaptive refinement pattern (ARP) attribute set. 35

Table 4.2. Frame rate of fully subdivided model versus my approach. 41

Table 6.1. Constant attribute set. 60

LIST OF FIGURES

Page

Figure 2.1. Performance of CPUs and GPUs over the last few years. 7
Figure 2.2. Index of multi-dimensional thread IDs and block IDs. 10
Figure 2.3. Representation of OpenGL supported serialized VBO data layout. . . . 11
Figure 3.1. Animation frames for burning the Stanford bunny. 14
Figure 3.2. Approximated heat boundary expansion. 16
Figure 3.3. The division of the heat boundary. 17
Figure 3.4. The deformation coordinates of a single triangle. 19
Figure 3.5. Model subdivided into blocks. 21
Figure 3.6. Object deformation with and without structural deformation. 23
Figure 3.7. Deformation with surface removal. 26
Figure 3.8. Animation frames of burning models. 29
Figure 4.1. The deformation of a low polygonal model. 32
Figure 4.2. The level subdivision of a single triangle. 35
Figure 4.3. Heat boundary areas and barycentric point sets. 37
Figure 4.4. The refinement hierarchy and deformation. 38
Figure 4.5. Real-time computation of LOD for a burning object. 40
Figure 5.1. Shell model deformation (left) vs solid model deformation (right). . . . 43
Figure 5.2. The combustion of a solid model and the spread of procedural fire. . . . 45
Figure 5.3. Heat boundary with different levels of boundary. 46
Figure 5.4. Categorized model triangles. 47
Figure 5.5. The deformation coordinates of a single triangle. 49
Figure 5.6. Level of detail (LOD). 51
Figure 6.1. A wax like solid model (left) and it's melting effects taken place (right). 54
Figure 6.2. The melting of a solid model and the spread of procedural fire. 55

Figure 6.3. Categorized model vertices. 58
Figure 6.4. The deformation coordinates of a single triangle. 59
Figure 6.5. Melting sequence of a wax form solid model. 62

CHAPTER 1

INTRODUCTION

In recent years there has been growing interest in limitless realism in computer graphics applications. Among those, my foremost concentration falls into the complex physical simulations and modeling with diverse applications for the gaming industry. The advances in computing hardware and 3D graphics rendering techniques have driven research and development intended at closing the gap between the naturalization and visualization of virtual spaces. These visual simulations play an increasingly important role as a part of game industries, modeling of natural systems, applications such as pilot training, war simulations and medical imaging etc. Simulation is also used to show the eventual real effects of alternative conditions and it is very useful when the real system cannot be engaged, because it may not be accessible, or it may be dangerous to engage in real life. Furthermore, most different simulations have been virtually successful by replicating the details of a physical process. As a result, some were strong enough to lure the user into believable virtual worlds that could destroy any sense of attendance.

There are many natural phenomena that most of us encounter in daily life. The singularities related to water, wind, smoke and fire are the most common among all. With rapidly developing graphics hardware, there is a cumulative attention for simulation and visualization of natural phenomena within the computer graphics community. However, maintaining a believable realism of a simulation which similar to event that everyone is familiar with is quite a challenging task. Especially, in gaming industry, most of these require visually compelling but not sternly accurate imitations. Accuracy is demarcated as how well the simulation matches with the believable realism of such event. Even though, most of these are not usually addressed in scientific simulations, it has to be capable of capturing the visual physiognomies of the phenomenon. In the case of visual simulation, we would like to match what we see in the real world occurrence. Each natural phenomenon may require different simulation models that are incompatible with each other. Therefore, in

this research I have focused on fire simulations and its deformation process towards various objects specified in gaming environment.

In most game engines model loading takes place at the beginning of the game or when the game is transitioning between levels. Game models are stored in large data structures. Since changing or adjusting a large data structure while the game is proceeding may adversely affect the performance of the game. Therefore, developers may choose to avoid procedural simulations to save resources and avoid any interruptions on performance. I introduce a process to implement real-time model deformation while maintaining performance. It is a challenging task to achieve high quality simulation while utilizing minimum resources to represent multiple events in timely manner. Unlike many studies that focus on exact simulation of reality, I have focused on the fundamental physical changes of the simulation process. This would be adequate to feel the realistic essence of the virtual world. Especially in video games, this overwhelming criterion would robust enough to sustain the engaging player's willing suspension of disbelief.

I have implemented and tested these methods on a relatively modest Graphics Processing Unit (GPU) using Compute Unified Device Architecture (CUDA). My experiments suggest that these methods give believable rendering of the effects of fire while using only a small fraction of Central Processing Unit(CPU) and GPU resources.

1.1. Hypothesis

The hypothesis of this dissertation is as follows:

One can simulate different aspects of deformation and consumption of virtual objects by procedural fire and achieve visually plausible results at interactive rates on current hardware. Where the terms describe as;

- *Different aspects*: Features such as burning solid objects, burning shell objects and melting objects.
- *Deformation and consumption*: Most objects experience shape changes when it burns due to the ingesting process of fire.
- *Virtual objects*: Objects that create using computer graphics.
- *Procedural fire*: Fire simulation that take place and behave dynamically upon the situation and interaction.
- *Visually plausible*: Looks realistic and direct towards the believable realism.
- *Interactive*: Maintains sufficient frame rate in current hardware for user to make changes in parameters and gain immediate results.

1.2. Contribution

Today, the video games and computer graphics have become a part of the main stream culture of the modern society. The objective of this work is to improve interactive entertainment and visual effects in video games. This is a substantial contribution to bring up the gaming environments one step closer to the real world phenomenon. Furthermore, my aim is to extend this work to benefit the computer graphics community with visual effects and related fields.

This research mainly covers the simulations related to fire, specifically a decomposition and deformation of burning shell models and burning solids. In addition this framework also extends to the low polygonal object deformation and solid object melting. This work tries to bridge the gap between interactive methods and physically based simulations. Video games related interactive simulations like deformation of burning object have not being

achieved before. My work is strictly aimed at computer graphics and virtual environments. The techniques presented in this work address the visualization of physically based simulation under general circumstances. Special cases such as combustion under high winds or burning of a moving object etc. may have to address independently. In addition, this is not necessarily identical to real world situation where the theoretical or experimental data and scientific calculations involved. However, my approximated simulations are visually compelling and much comparable to the real world phenomenon.

This dissertation discusses unified simulation framework addressing the several proceedings related to deformation of burning objects such as:

- Heat boundary approximation
- Deformation and decomposition of shell based models
- Deformation and decomposition of low polygonal models
- Deformation and decomposition of solid based models
- Structural deformation of burning objects
- Melting objects
- Flame distribution of a burning object

1.3. Overview

The rest of the dissertation is organized as follows. Chapter 2 consists basic introduction to GPU programming. Here, I have briefly addressed the General-Purpose Graphics Processing Unit (GPGPU) programing with OpenGL. This programming method has been used in CUDA implementation. In chapter 3, I propose a framework for real-time rendering of burning objects in video games. Here, it is mainly focused on deformation of general shell based polygonal models. Real-time rendering of burning low polygon models in video games is discussed in chapter 4. In this chapter, I have discussed a systematic polygonal refinement method that suitable for my criteria. In chapter 5, I address real-time rendering of burning solid objects in video games. Usually, 3D polygonal models loaded as shell based structures in computer games. However, some objects must behave like natural solids. Here, I am focusing on how to overcome this issue when deformation takes place. Chapter 6, introduces a

method involved in real-time rendering of melting objects in video games. In this chapter, I am introducing an efficient method to simulate viscoelastic fluid like behavior into polygonal meshes. Chapter 7, I conclude the findings and justify the hypothesis. Appendix A contains the data structures and VBO framework that has used in my implementation. Appendix B shows my implementation of calculating Normals in the GPU. This code structure can be followed to modify the color changes of the burning model as well as altering texture effects. Appendix C contains complete GPU side code snippet showing how to apply fluid like behavior to the polygon model. This will guide the user to understand the implementation aspect of my framework.

CHAPTER 2

GPU PROGRAMMING

With the evolution of high computational performance of the modern graphics hardware, programmable flexibility turned out to be a bonus. Many researchers apply this technology to resolve problems previously attempted to solve using CPUs. The GPU consists of a number of streaming multiprocessors (SMs). Each SM has a set of execution units, a set of registers and a shared memory, designed to obtain the best performance with graphics computing. Although, the GPU programming for general purpose (GPGPU) are becoming more popular because of its promise of massive parallel computation abilities. The modern GPUs consider as highly multithreaded architecture which is very suitable for solving data-parallel problems and does it with increasingly higher performance rate. Furthermore, CUDA is basically a new programming approach that improves the unified shader model of the most current GPUs from NVIDIA. The unified shader models are capable of executing any type of shader when graphics hardware supports the assignment. This Chapter contains a brief introduction of fundamentals of the GPU programming using CUDA and OpenGL.

2.1. Hardware Performance and Comparison

This section committed to present a performance analysis and GPU development of the last few years. The idea of presenting relevant data is to illustrate the potential of this current hardware and evidencing the purpose of my research leading to this direction. In addition, my hypothesis laid on performance of current existing hardware. There are many research carried out in past few years with various comparison criteria between CPUs and GPUs with different assignments. Out of many convincing references with positive conclusions about GPU performances, I list few relevant ones as follows. General purpose molecular dynamics simulation work present by Joshua A. Anderson a and Chris D. Lorenz [5] claim that their single GPU implementation provides a performance equivalent to that of fast 30 processor core distributed memory cluster. Ronan Amorim and Gundolf Haase [4] present their work comparing CUDA and OpenGL implementations for the Jacobi iteration with

CPU performance. By benchmarking GPUs to tune dense linear algebra by Vasily Volkov James W. Demmel [53] claims matrix-matrix multiply routine (GEMM) runs up to 60% faster than the vendors implementation and approaches the peak of hardware capabilities. Furthermore, Victor W Lee, Changkyu Kim [28], Naga K. Govindaraju, Jim Gray [19], Nadathur Satish, Mark Harris [44], Chi-Keung Luk, Sunpyo Hong [31], Ian Buck, Tim Foley [10] and Glenn A. Elliott, James H.Anderson [13] have present work with executing various datasets and algorithms on modern GPUs. In addition, Shane Ryoo and Christopher I. Rodrigues [43], Buatois, Luc and Caumon [9] present optimizing techniques that useful in memory management with GPUs using CUDA.

The following figure 2.1 summarize the development of the GPU technology for the past few years by illustrating floating-point operations per second (FLOPS) for the CPU and GPU (*left*) and Memory Bandwidth for the CPU and GPU (*right*).

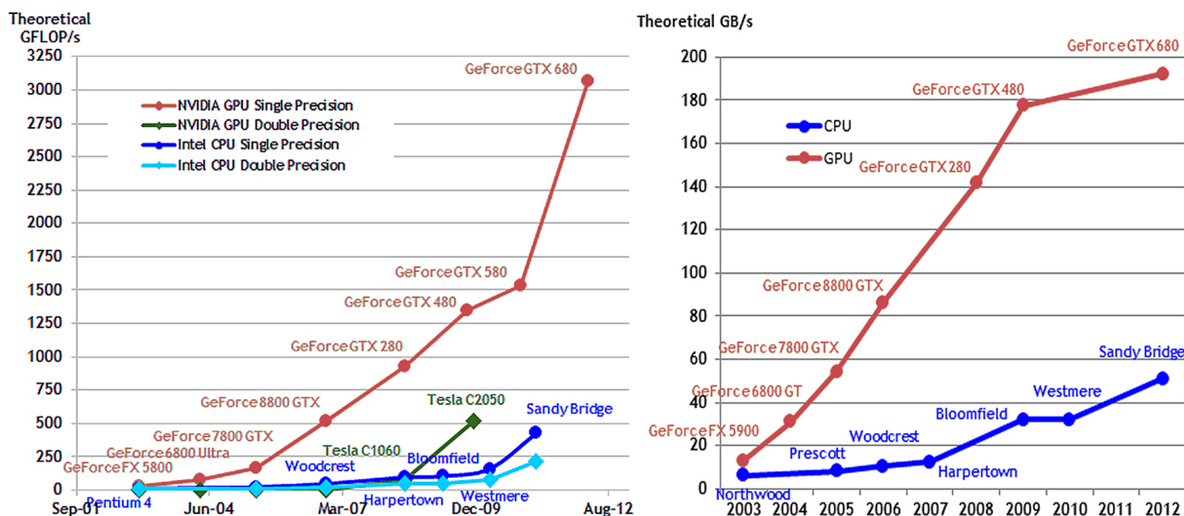

FIGURE 2.1. Performance of CPUs and GPUs over the last few years. Figure courtesy of NVIDIA, and adapted from Ref. [39]

Since, GPUs are designed for highly parallel computations and its transistors are devoted to data processing rather than data caching and flow control, the above results are obvious. However, CPU and GPU comprehend with different characteristics of design criteria to serve its purpose. Generally, CPU core is designed to execute one set of instructions when GPUs are designed to execute multiple parallel set of instructions per cycle. The CPU uses

cache to improve performance by reducing the latency of memory accesses as GPU uses cache to amplify bandwidth using shared memory. CPUs support one or two threads per core while latest CUDA supported GPUs support up to 2048 or more threads per streaming multiprocessor. Therefore, the GPU is especially well suited to address problems that can be expressed as data-parallel computations than CPU. This observation is not conclude GPUs are much efficient than CPUs in every aspect. However, it is clear that could improve performance of a given task by consuming combination of both CPU and GPU. In order to manipulate the usage of these devices we must have general understand of its structural design. The following Section presents a brief overview of memory architecture of CUDA supported GPUs.

2.2. GPU Memory Architecture

The structure of GPU memory architecture contains most die area with thousands of cores consist with Arithmetic Logic Units (ALUs) and relatively small caches. On the contrary, CPUs consist of a few cores that most optimized for serial processing. GPUs designed to execute number of threads within cores to support data-parallel processing. Data-parallel processing can be performed on applications that process large data sets such as arrays, sets of pixels and vertices etc. Data-parallel programming model often used in 3D rendering applications due to its ability to speed up the computations. This data-parallel processing executes by mapping data elements to parallel processing threads. When programmed through CUDA, the GPU capable of executing a very high number of threads in parallel. GPUs always operate as a coprocessor to the CPU. Therefore, GPU identify as a *compute device* when CPU identify as *host*. In running application, compute-intensive portions of the host could load to the compute device as it to process in data-parallel mode. This data transaction between host and device must be proceeding through storage spaces. To accomplish these criteria, both units maintain their own Dynamic random-access memory (DRAM), referred to as *host memory* and *device memory*. CUDA provides a mechanism to copy data from one DRAM to the other using optimized Application Programming Interface (API) calls that utilize the devices high-performance Direct memory access (DMA). Having

an overview of this structure is always helpful when its come to allocate the memory spaces for mapping the data structures that has used in the implementation.

Furthermore, CUDA provides the flexibility of manipulating threads to attain the most optimum data-parallel procedure under given operation. The independent threads are organized into blocks which can contain anywhere from 32 to 2048 threads each in modern CUDA supported GPU. The blocks are completely independent and each block is given a small area of shared memory that exists on the multiprocessor. Threads in a single block will be executed on a single multiprocessor, sharing the software data cache, and can synchronize and share data with threads in the same block [42]. On the otherhand, threads in different blocks may be assigned to different multiprocessors concurrently. CUDA provides its own synchronization mechanism to keep all the threads in order. Any thread in the block is delayed at this synchronization point until all the other threads in the block complete its task. Any thread in the block can access this shared memory area without much latency. Each thread is identified by its *thread ID* and each block identified by its *block ID*.

These thread IDs are designed as thread index within the block. For example:
For a two-dimensional block of size (D_x, D_y), the thread ID of a thread of index (x, y) is $(x + yD_x)$.
For a three-dimensional block of size (D_x, D_y, D_z), the thread ID of a thread of index (x, y, z) is $(x + yD_x + zD_xD_y)$.

Usually, block ID specified by a grid as a two-dimensional array of blocks using a 2-component index. For the block of size (D_x, D_y), the block ID of a block of index (x, y) is $(x + yD_x)$. The following figure 2.2 illustrates the diagram of imaginary structure for two dimensional (*left*) and three dimensional (*right*) thread ID index alone with block ID index.

The CUDA provides the flexibility to apply necessary dimension for the thread indexing depending on the application. For example, two dimensional indexing more suitable for matrix multiplication, texture mapping procedures etc. On the other hand, three dimensional threads indexing mostly used in image processing and video streaming applications. The number of threads per block and number of required blocks may decide depending on

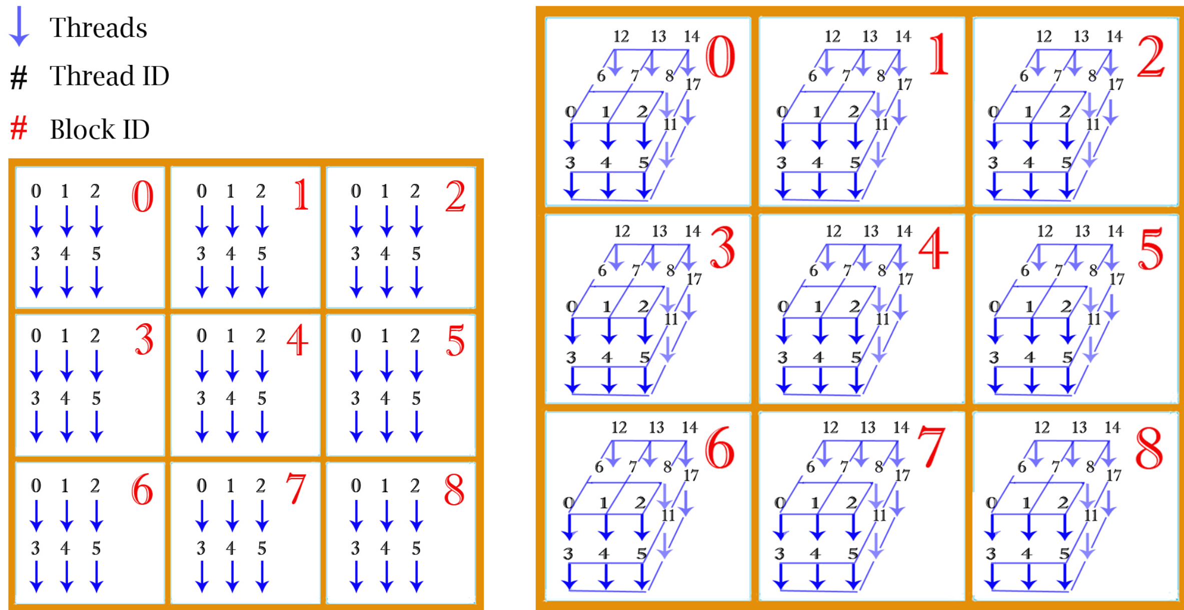

FIGURE 2.2. Index of multi-dimensional thread IDs and block IDs.

the application. Usually, developer must have a basic approximation about the total number of data elements to be scheduled with data-parallel processing. Less number of threads and blocks for large number of data elements may cost the performance of the application. Therefore, proper optimization measure must to be addressed depending on the circumstances. This will help the cores to be productive and there will be enough parallelism to keep them busy. Furthermore, maximum number of threads supported by each block may vary depending on the hardware specification. Rest of the chapter consists with a quick overview of required the data structures and mapping techniques that followed to achieve the prompted task.

2.3. OpenGL Interoperability with VBO

In this section, we will look into making host side data structures that suit enough to precede with device side data-parallel operations. A vertex buffer object (VBO) is an ideal feature that OpenGL provides. VBO is a powerful method that allows us to store certain data in high-performance memory of the host. This method facilitate for uploading data such as vertex, vertex normals, color, etc. to the video device prior to immediate

rendering. The basic idea of this mechanism is to encapsulate data into memory buffer in object state. Each data element will be available through identifier pointers. CUDA and OpenGL interoperability follows when CUDA maps a data buffer into its own memory space. When a deformation occurs on a burning object, its color and shape subjected to be changed. In order to illustrate these features, I have loaded vertices, color map and list of normals coordinates into the VBO and map to the CUDA. The following figure 2.3 shows the imaginary view of serialized VBO memory data structure that I have used in the implementation. Each pointer has access to the appropriate data set and provides it service in each call.

More information related to function calls and detailed description can be found in NVIDIA CUDA programming Guide [39]. Next we must map this serialized data structure into data-parallel processing structure described in section 2.2.

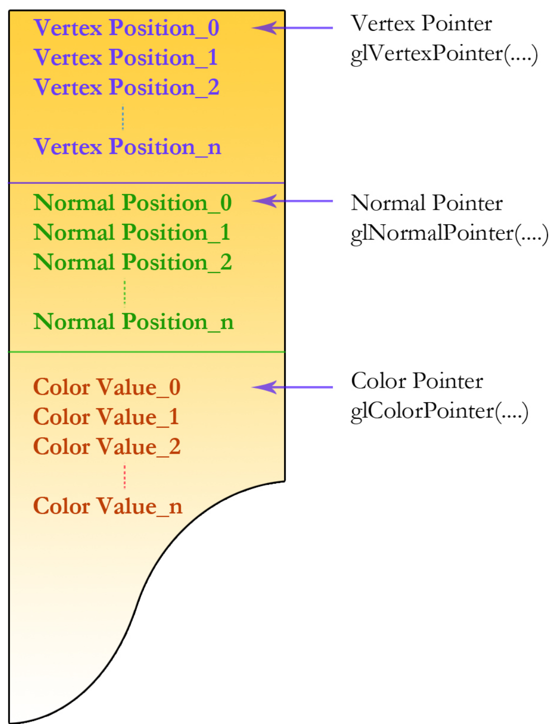

FIGURE 2.3. Representation of OpenGL supported serialized VBO data layout.

2.4. GPU Implementation with CUDA

To this point, I have discussed different memory structures that have being used to manipulate the model data. When it comes to object deformation, modifications of color and normal coordinates must be situated in the corresponding vertices. Therefore, it is vital to gain precise control over GPU mapped data and its correspondents. However, the VBO contains a serially arranged data set and GPU must achieve data-parallel approach. In this transition, one primary criterion is to keep track of each data element that to be processed in separate threads. The key to succeed from this problem is to find a way to map these data into the threads without breaking its serialized order. CUDA facilitate the user to choose threads and block size based on the size of the data sets using in the procedure. Here, user may decide the dimension of the grid including number of blocks and number of threads per block. These values will be passing to the GPU on the CUDA initializing process. When data mapping begins CUDA assigned each element with an index number (*idx*). I have calculated this idx value as follows.

*idx = block ID * Number of threads per block + thread ID*

This assignment presents one to one alignment between *idx* value and the serialized index while mapping the data into each thread. For a given vertex $DATA[idx]$, the corresponding color value can be found in location at $DATA[idx{+}number\ of\ vertices]$ and the Normal coordinates will be located at $DATA[idx{+}number\ of\ vertices{+}Number\ of\ color\ elements]$. CUDA supports different idx assignments. User can define idx depending on the application, the required memory structure and its dimension.

The implementation code snippets related to the discussion in this chapter can be found in Appendix A. In order to gain extended knowledge of this procedure, it is important to understand CUDA fundamental function type qualifiers such as __device__, __global__, __host__ and Variable Type Qualifiers such as __device__, __constant__, __shared__. Detailed description related to these topics and additional memory types and management techniques are listed in NVIDIA CUDA programming guide [39].

CHAPTER 3

REAL-TIME RENDERING OF BURNING OBJECTS

This chapter presents fundamental framework for emulating the deformation and consumption of polygonal models under combustion while generating procedural fire. I concentrate on introducing this framework aiming to facilitate video games. One way a game can make an impact on a player is by increasing realism. Replicating the details of a physical process can increase the believability of virtual worlds, and draw the player into the virtual environment. My focus is on achieving the best visual effects possible while maximizing computation speed so that the processing power is available for other tasks in video games. In this chapter, I describe my representation framework for the internal deformation and the key features of such a strategy including useful optimization techniques.

3.1. Overview

When an object burns, its geometry and topology changes as heat spreads. There will be multiple internal chemical reactions at various stages of the process, during which its properties may change from solid to liquid and from liquid to gas due to volumetric expansion caused by weakening bonds at the molecule level. Modeling an object undergoing combustion also includes heat boundary expansion, flame distribution, fuel consumption, and shape deformation over time. Using a unified representation of all the properties of a given object's deformation and consumption during combustion in a gaming environment may not effective due to the limitations of resource consumption and the suitability of specific representations. Here, I have focused on most fundamental changes of the physical process of such event. The changes in the bond strength between molecules disturb the stability of the internal forces. Therefore, these unstable molecules will start moving to find its firm position in the available space. This causes the changes in shape to the object's affected areas. I show how to mimic this molecular behavior into vertex displacement. In this chapter I have mainly discuss heat boundary expansion, internal and structural deformation alone with the flame distribution of exposed simulation.

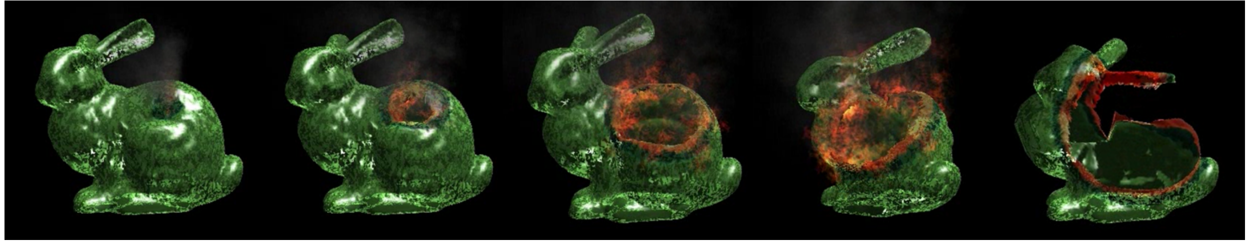

FIGURE 3.1. Animation frames for burning the Stanford Bunny.

3.2. Previous Work

To the best of my knowledge, a simulation of procedural deformation and consumption of objects due to combustion has never been used in gaming environments. Instead, developers appear to use model swapping techniques. Therefore, this is the first attempt to introduce the modelling of combuston to the field of gaming in a generalized form.

Melek and Keyser [32, 33] discuss techniques that were used in selected object deformation due to fire, however, these methods are not designed for game playing environments in which a certain amount of realism can be traded for performance. Sederberg and Parry [46] and Hsu, Hughes and Kaufman [23] introduce some adaptive techniques of model deformation. Müller and McMillan [36] discuss real-time rendering techniques for deformation focussing on selected materials in spacial cases. Toivanen [52] is quite adaptive in deformation of gaming models. Nguyen and Fedkiw [38] introduce high quality flame simulations, but do not address object deformation. Wei and Zhao [57] use an approach similar to Melek, defining solids as a volumetric implicit field, but also do not discuss object deformation. Wei and Li [55] use splatting techniques that help to increase visual impact. Moidu, Kuffner, and Bhat [34] demonstrate an attempt to animate deformable materials, but they do not introduce complex heat transfer models. Although they discuss the spring-mass model technique to simulate combusting surfaces, their main focus is on selective materials such as paper and cloth. Fuller [17] has a useful method for generating procedural volumetric fire in real time using curve-based volumetric free-form deformation.

3.3. The Heat Boundary

In the real world, the temperature of an object changes over time and space during combustion. I assume that the elevated temperature generated in the model due to fire effects have a strong influence on the mechanical behavior of the object and that the mechanical behavior influences the thermal response. Calculating actual heat transfer in a finite element is considered to be a composite process and depends on many parameters including the impact of environmental factors such as humidity.

$$R^2 = |\sin(\pi\Theta/\Delta r) + \sin(\pi\Theta) + \rho((x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2)|,$$

where $R = r + \Delta r$, the radius r is incremented by Δr in each Δt time period. The angle Θ is a random value in order to avoid uniformity of the expanding heat boundary. The heat index value is represented by ρ. The value of ρ depends on the size of the triangles used in the model and the material that the model is made from. The location of the heat source is (x_0, y_0, z_0).

The above boundary function creates a roughly spherical but irregular heat boundary around the heat source. Burning objects contain different intensities depending on flammability of the material. Some were burn rapidly when others burn in relatively slower. The intensified flames produce more heat that flow through the burning surfaces. The rate of heat transfer through a given surface defined as *heat flux* or *thermal flux* [8, 29]. This causes a heat flux determind the deformation speed of the burning object. To maintain the simplicity and the visual effectiveness of the simulation, I let the heat flux determined by the changes of the Δr. If the flames are intense, the increment of Δr in time Δt will be increase by suitable approximation. I let the developer to choreograph the simulation with proper measures depending on the game environment where the replication is engaged in.

Figure 3.2 illustrates the similarity of the approximated heat boundary expansion for single versus multiple heat sources. The multiple source heat boundary expands throughout the model with behavior similar to single heat source approximation implemented using the above function. In the real world, heat sources reproduce throughout the burning object

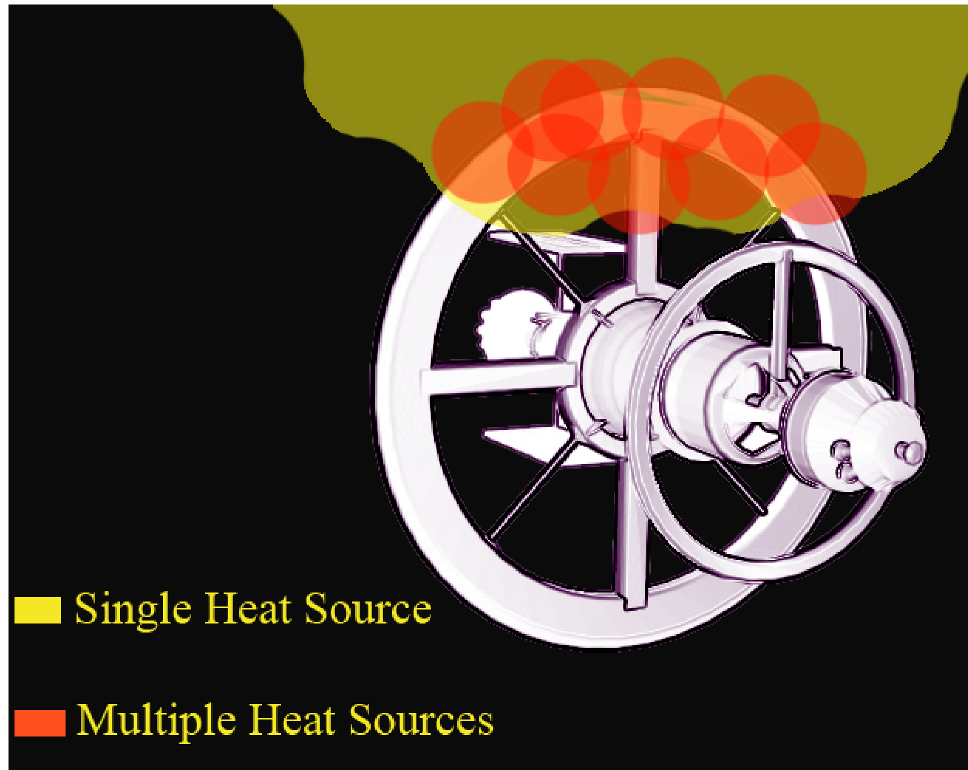

FIGURE 3.2. Approximated heat boundary expansion in case of single vs. multiple heat sources.

as flames distribute over time. Each flame capable of generate heat that flow throughout the object. In order to determine the temperature dispersal in a medium, it is necessary to solve the suitable form of the heat equation. However, the solutions would depend on the physical conditions of the medium. Therefore, determination of the authentic heat boundary expansion is computationally expensive. I believe that the use of a single source heat boundary expansion is a viable alternative for use in video games.

In object combustion process, there will be various physical changes taking place. In addition to heat boundary expansion, color diffusion, thermal expansion and deformation are the few sequential events that may occur in affected areas. Some of these events are fairly noticeable than the others. In a dynamic simulation, these sequential occurrences must handle carefully. Therefore, when representing a unified framework, the deformation process must perform on different stages. Deforming a polygonal model must take many cautious measures to maintain realism of the performance. First we must gain enough control over

the sequential events folding in the simulation. As a solution, I divide the heat boundary into different areas and performed different systematic operations.

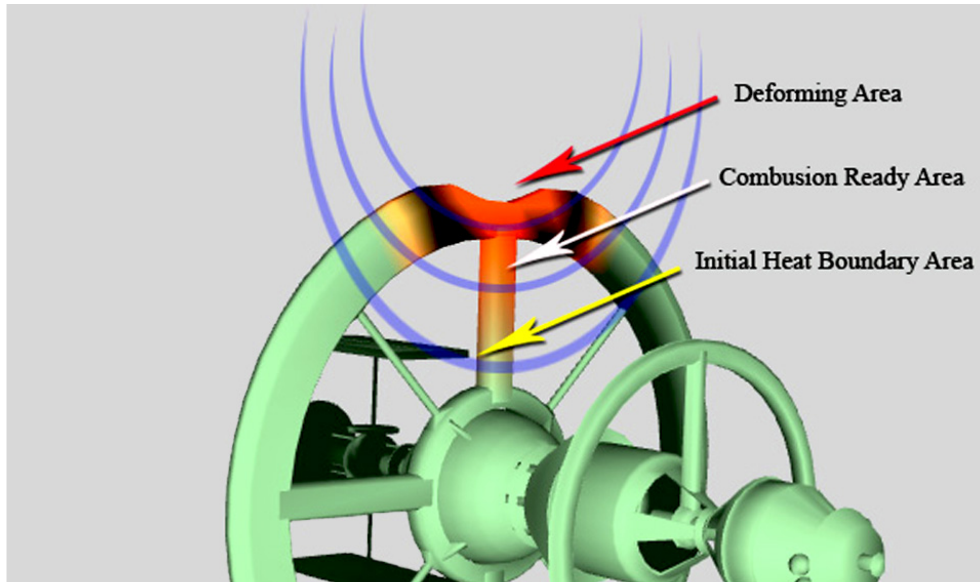

FIGURE 3.3. The division of the heat boundary.

Figure 3.3 shows the heat boundary division consist with three different areas, the *initial heat boundary area* in which combustion is actively taking place and vertices are being deformed, the *combustion-ready area* in which ignition starts, and the *deformed area* which has been burned and is ready for surface removal. By this proceeding, I was able to apply the color effects to the model as it is shown here. First, the vertices within the area under the initial heat boundary are directed to the deformation process. Especially, the initial Heat boundary used to apply structural deformation. Secondly, division is also useful when it comes to flame distribution throughout the model. The combustion ready area is considered as the area where ignition starts in the model. Finally, the actual internal deformation and surface removal process undergoes on the deformed area.

3.4. Internal Deformation

Some events like volumetric expansion due to the heat of an object are barely visible to the naked eye in a real world physical simulation, and can easily be ignored in given virtual world. Internal deformation is achieved by displacement of the vertices of the model mesh.

The position of each vertex will depend on three properties: vertex distance, gravitational force, and material index. While I may assume that material is a constant over large areas of the model, vertex distance and gravitational force are more complicated, and will be examined next. Here, I consider each vertex of the object to be analogous to an molecule, and I take the distance between vertices of a given triangle as the strength of the bond between vertices. The bond between two vertices of a triangle is taken to be inversely proportional to the distance between them. Vertex displacement is inversely proportional to bond strength, that is, directly proportional to distance, and scaled by material index.

I make use of the following designer-set constants, L, ε, β, ρ, and ϕ described in Table 3.1. The first is an integer, the remainder are real-valued constants between zero and one. All of the values can be made constant for the entire model, but in principle L can be different for each vertex, ε can be different for each edge, and β, ρ, and ϕ can be different for each block.

Name	Type	**Description**	**Level**
L	Integer	Flammability	Vertex
ε	$0 < \varepsilon < 1$	Meltability	Edge
β	$0 < \beta < 1$	Displacement scale	Block
ρ	$0 < \rho < 1$	Material density	Block
ϕ	$0 < \phi < 1$	Bond strength	Block

TABLE 3.1. Designer set constants.

Figure 3.4 illustrates the coordinates and parameters used in these equations. The values λ and μ are the displacement amounts of each triangle due to the effect of heat on the vertex. The values used for λ, μ will be discussed further below. The lengths of BC and BA are d_1 and d_2 respectively. The points (x_1, y_1, z_1) and (x_2, y_2, z_2) are a μ and λ fraction of the length along the edges (respectively BC and BA) of the triangle.

Suppose B is a vertex to be displaced in triangle ABC, where $A = (x_a, y_a, z_a)$, $B = (x_b, y_b, z_b)$, and $C = (x_c, y_c, z_c)$. B is to be displaced to (X, Y, Z) as follows:

$$X = (x_1 x_2 (y_a - y_c) + x_1 x_a (y_c - y_2) + x_c x_2 (y_1 - y_a) + x_a x_c (y_2 - y_1))/$$

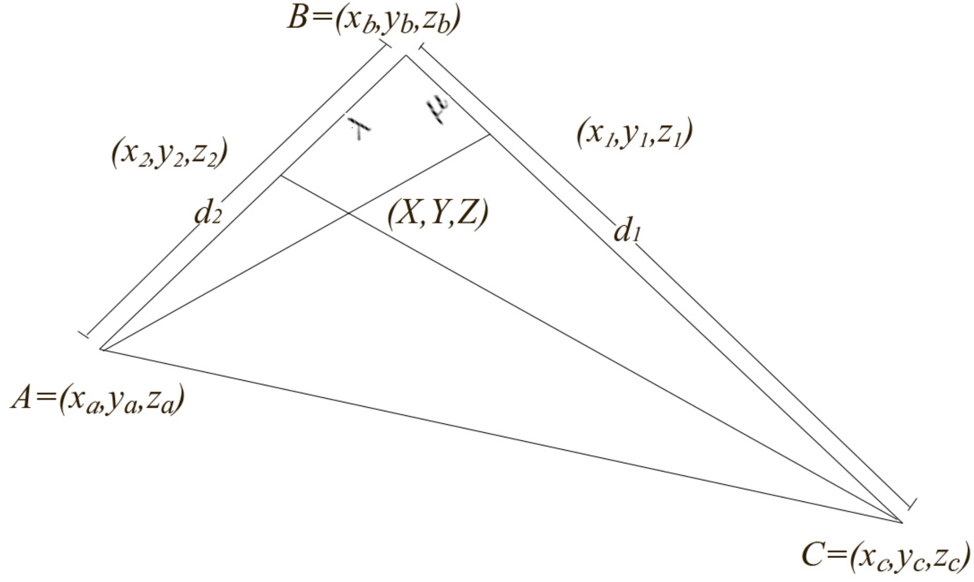

FIGURE 3.4. The deformation coordinates of a single triangle.

$$
\begin{aligned}
& ((x_a - x_2)(y_c - y_1) - (x_c - x_1)(y_a - y_2)) \\
Y &= (y_1 y_2 (x_a - x_c) + y_1 y_a (x_c - x_2) + y_c y_2 (x_1 - x_a) + y_a y_c (x_2 - x_1))/ \\
& ((y_a - y_2)(x_c - x_1) - (y_c - y_1)(x_a - x_2)) \\
Z &= (z_1 z_2 (y_a - y_c) + z_1 z_a (y_c - y_2) + z_c z_2 (y_1 - y_a) + z_a z_c (y_2 - y_1))/ \\
& ((z_a - z_2)(y_c - y_1) - (z_c - z_1)(y_a - y_2))
\end{aligned}
$$

where

$$(x_1, y_1, z_1) = \mu C + (d_1 - \mu)B$$

$$(x_2, y_2, z_2) = \lambda A + (d_2 - \lambda)B.$$

The values μ and λ are displacement parameters for vertex B. In addition I use a *displacement adjustment parameter* β to allow for the variation in triangle size from one model to another. The designer must set this value as part of the design process. Let ρ denote a *material density index* . When both vertices of an edge are inside the heat boundary, bond strength is weaker by a factor of ϕ than when one vertex is outside of the heat boundary. ρ and ϕ are real values between 0 and 1 that are again set by the model

designer. Placement(λ) is then defined to be $\beta\rho L/d_2$ if A is outside the heat boundary, and $\phi\beta\rho L/d_2$ otherwise, where L is the *flammability* of the vertex.

Burning objects are consumed by combustion, and combustion subsides when there is nothing left to consume. I model this with a flammability value at each vertex. The counter decreases each time vertex displacement is processed. After the level counter reaches zero, I consider that there is no consumable resources left at the vertex. The designer sets the initial flammability value for each vertex. This gives the designer the ability to vary flammability from place to place in the model, thus mimicking the effect of having the model constructed from different physical materials such as wood or metal. Among all of the external forces, gravity plays a major part in every physical based simulation. The effect of gravity is computed as follows:

$$Y = Y - \epsilon\vec{g},$$

where ϵ be a constant that represents the amount that the model melts due to heat, and $\vec{g}$ be the gravity vector.

3.5. Structural Deformation

Deformation of a burning object can be caused by factors such as the expansion and weakening of the internal bonds, and the relative weights of cantilevered parts of the object. Exact calculation of these complex processes is costly. Furthermore, accuracy of the real world simulation represents all the details of the physical phenomena, this is usually impractical when I consider the processing capability, and even when it comes to the chemical changes of the physical process. Therefore, I have usually target in this computer simulation is to simplify the physical model that enough to feel realistic essence. Here, I have eliminated certain secondary effects in order to achieve visually plausible results. By doing so I was succeeded achieving effective results in structural deformation. I simplify the process by considering only the weight of a given point of the structure. The weight changes of the burning structure will occur due to consumption of the object by fire. I have divide the object into uniform blocks and treat each block as a single unit. Changes within a block are propagated to neighboring blocks. This section is divided into two subsections. Section 3.5.1

describes the model subdivision into blocks in more detail. Section 3.5.4 describes the removal of surfaces that have been marked by having their vertices flammability value go to zero.

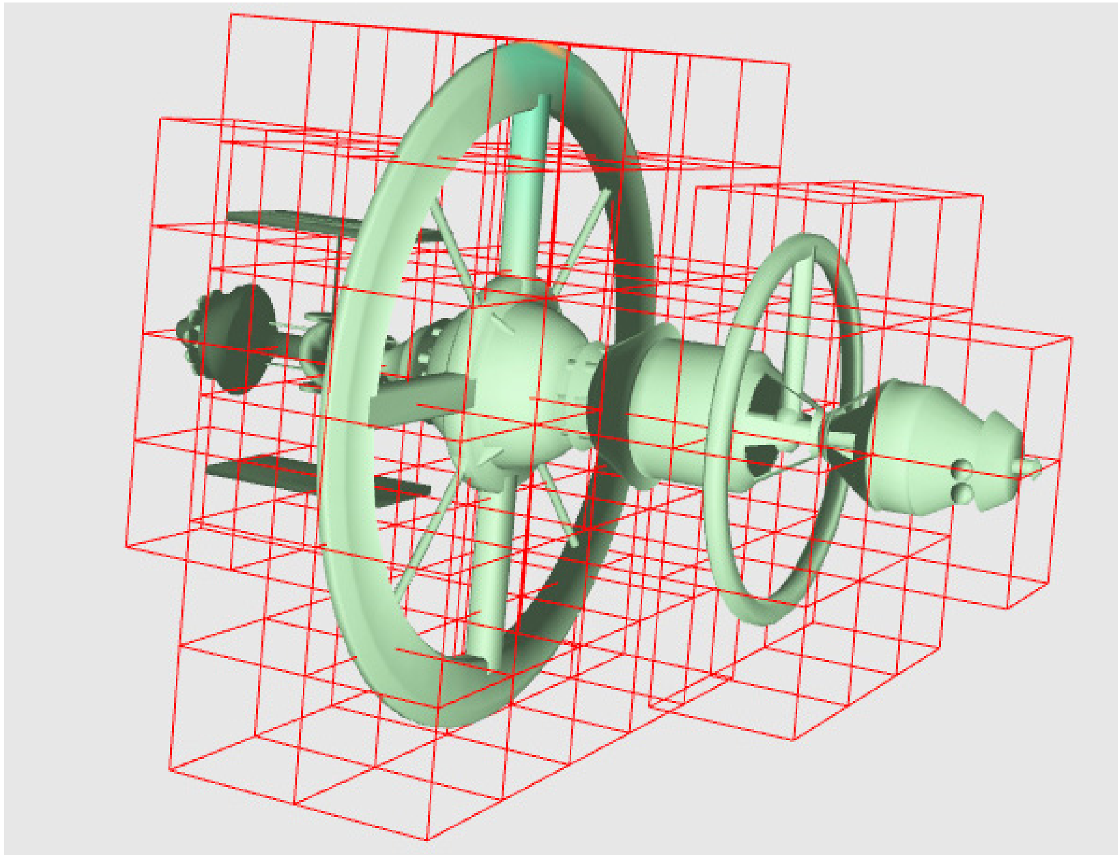

FIGURE 3.5. Model subdivided into blocks.

3.5.1. Model Subdivision

I start by constructing an axially aligned bounding box around the object, then decompose it into a grid of smaller axially aligned bounding boxes which I will call *blocks*. Deciding the number of blocks per model is up to the designer. Higher numbers of smaller blocks will make the simulation more realistic at the cost of lower performance. I weight the blocks according to the number of vertices in them, and discard the empty ones of weight zero. Figure 3.5 shows a model subdivided into blocks. Only nonempty blocks are shown. I store for each block the amount of rotation, midpoint of each box, number of vertices and the list of neighboring subunits. Since all the blocks are interlinked, a change to one block may affect all of the blocks in the model. To limit the required computation, I apply changes

to only immediate neighboring blocks, and rely on time to propagate the effects further. The initial condition of the box with its assigned weight (number of vertices) chanegs as vertices are removed. The change of the weight in the block is indicated by a slight rotation of the box around its midpoint. The direction of the rotation will be determined by the placement of the displaced vertex compared to the midpoint of the box. Interestingly, I have found applying a random rotation also gives satisfactory results of the structural deformation. In fact, it appears that for gaming applications the random rotation will be adequate since calculating position of displaced vertex can be costly. Secondly, stability will change due to the rotation of the immediate neighboring box. In order to cope with this I keep track of neighbors of each subunit by maintaining a data structure that contains neighbor indices, rotation amount, number of vertices etc.

3.5.2. Rotate Around the Midpoint

Over doing the rotation of each subunit of the model may cause an unrealistic disfigured structural deformation. However, to handle this situation I have consider that the total rotation must be very minute in degrees unless the subunit is not free from adjacent neighbors (two or three sides). Thus corner bending is possible effect of a deforming object under combustion. I consider the rotation amount of the subunit proportional to the change in number of vertices in each unit. If a given displaced vertex changed index of its corresponding subunit to one of its neighboring subunit indexes, then there is a break in the equilibrium of the original subunit due to weight change. In addition this vertex displacement is sufficient enough to decide the direction of the rotation as well as its orientation such as pitch, roll and yaw.

The following function is used to calculate the rotation amount(R) of each bounding box.

$$R_{(Pitch,Roll,Yaw)} = \gamma\Delta\Theta \qquad 0<\gamma<1, \tag{4}$$

where γ is a constant that controls the rotation amount depending on the number of subunits in the model. In this simulation γ is carrying an important role by controlling

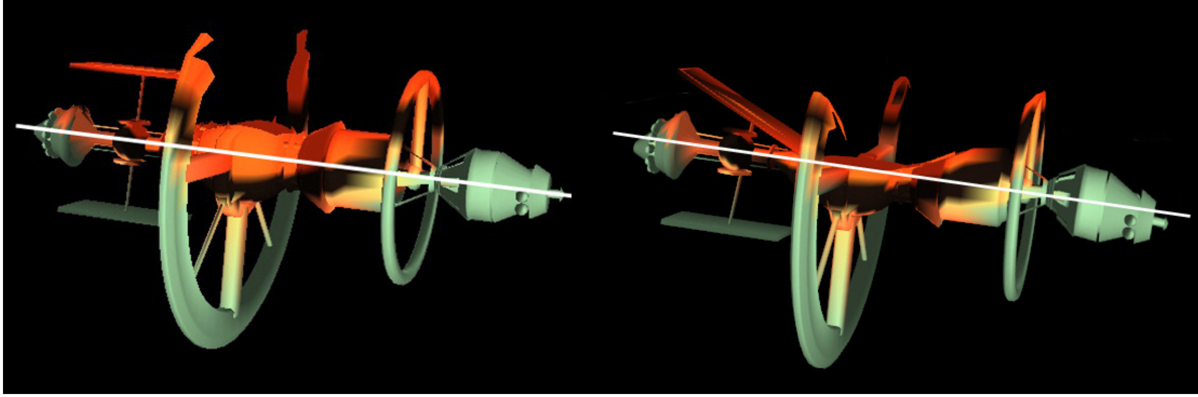

FIGURE 3.6. Object deformation without structural deformation vs. object with structural deformation.

rotation angles without over doing such operations. $\Delta\Theta$ is the change in angle by degrees due to a single vertex displacement in a given subunit. The change in angle per vertex displacement is something that the developer can assign due to the fact the triangles in a model are not carrying same size, number etc. These properties may change from one model to another. Therefore, the amount of rotation must be handled carefully in order to get a realistic simulation. This $\Delta\Theta$ value also may be define as a property of material density. In my simulation, I calculate the amount of rotation as per vertex change by: $\Delta\Theta = \rho\pi/N\varepsilon(5)$, where, N is the number of vertices per subunit, ε implies the number of neighbors per subunit and ρ is an approximated material density of the model. However, the γ was used to keep control of the overall rotational angle. As a result, I gained a very appealing structural deformation in this simulation. The following figure 3.6 illustrates difference between object deformation without structural deformation (*left*) and object deformation with structural deformation (*right*).

Futherfomre, one can notice the slumping in the last frames of the satellite and bunny in figure 3.8. As I have mentioned above, the first order rotation of the subunit due to the weight change will carry over to the immediate neighbors surrounding the subunit. The rotation of the immediate neighbors is determined by the following algorithm:

Get displaced vertex A
Get corresponding subunit X
Get mid point of the X
Find ø(Pitch, Roll, Yaw)
    by Function (4)
    Add ø(Pitch, Roll,Yaw) to X
    Get neighbor List Yn
    If Y_1 *at side of X*
        *-1 * æ* ø(Pitch, Roll, Yaw)*   *where 0<æ<1*
    Else if Y_1 *at up ||down of X*
        *+1 * æ* ø(Pitch, Roll ,Yaw)*   *where 0<æ<1*
Repeat for $Y_{1....n}$

3.5.3. Vertex Mapping

So far, I have calculated the box rotation of each subunit of the model. However, the rotation of the box does not demonstrates the actual structural deformation of the model. In order to gain the proper visualization, this displacement must take place in the vertex level of the model. Therefore, I have mapped the rotation of each subunit in to its vertices. Due to the limited resources that has consumed in this simulation, I have avoided continuous vertex mapping. In other words, this process does not change the vertices displacement whenever the subunit changes its position. The changes of subunits will be recorded into the data structure gradually. The vertex mapping will take part whenever the vertex is about to render at the graphic hardware level using CUDA technology. Therefore, this process

can be distinct near rendering simulation; due to the fact, the structural deformation is not completely dynamic in this simulation. It can be processed time to time while the simulation is going on.

3.5.4. Surface Removal

Due to the combustion, the resource consumption of the object may cause the mass of the subjected area to subside. This can be present in a graphical environment by removing the surface of the object's affected area. The model in this virtual world, there are few ways to illustrate this phenomenon. One traditional practice that has been carried so far is to illustrate the change in the model is by model swapping. In that case, the developer has to keep extra models in the storage space or loaded into memory.

One may suggest reconstructing the model mesh by removing unnecessary (burned) triangles. However, to carry out such a process in an ongoing simulation may cost more processing time. My approach is more procedural and efficient. Since this simulation builds influencing CUDA environment, I have gained an upper hand of controlling model rendering procedures. I simply identified render required triangles without harming the complete model mesh or orientation of the triangles. In order to determined the consumed triangles I have used a level counter that has discussed above in section 3.4. However, when it comes to the issue of surface removal the shape of the object has to be considered. For example, by removing the surface the object may appear as two different pieces. Sometimes, it will be an unrealistic behavior in the deforming model. I was able to prevent this sort of behavior using block division techniques that I also used in structural deformation 3.5.1

The technique I have used to prevent the model breaking into pieces in surface removal is fairly simple. If the subunit has no weight (no vertices) I simply consider it is a burned out unit. However, there cannot be any subunit that contains weight but no neighbors. If so, I have avoided applying the surface removal into that area; by doing so I have maintained the connectivity of the object. However, breaking the model into pieces or keeping the model intact is up to the choreography of the simulation. Figure 3.7 illustrates what my simulated model looks like after surface removal applied.

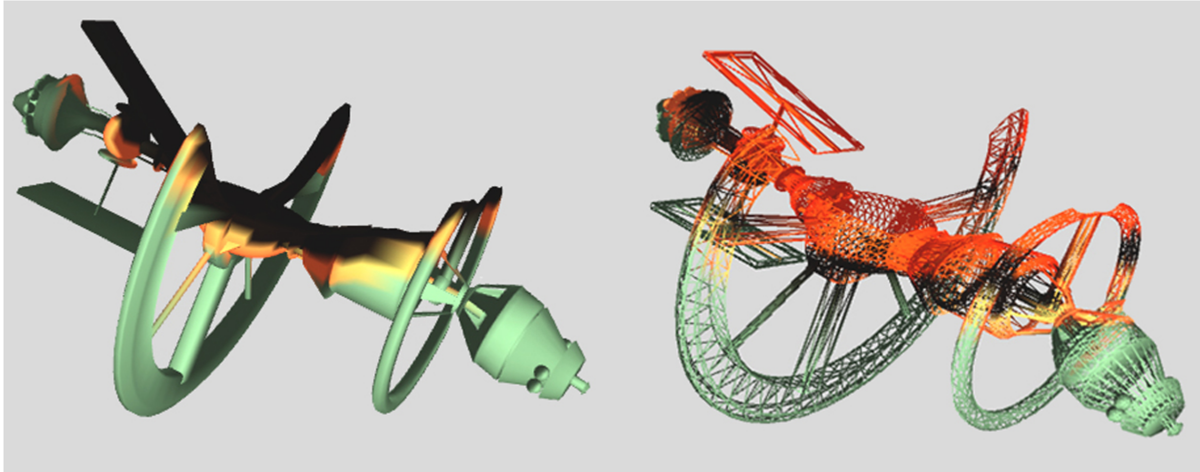

FIGURE 3.7. Deformation with surface removal.

3.6. Flame Distribution

When it comes to the realistic simulation of a burning object, the size of flame must change over time due to the richness of the resources it burns as well as maintaining the proper distribution. To manipulate the flame distribution properly I have taken advantage of the subdivided block properties described in Section 3.5.1. In this simulation, I have used a particles engine to generate visual fire. Flames will appear in each subunit that belongs to the combustion ready area of the heat boundary (Section 3.3). Treating set of particles as one group of flames make my process much easier. I have increase the number of particles in the given group according to the number of vertices covering the appropriate subunit. If the weight of the subunit becomes zero I disregard the unit and move the flame group to the adjacent neighboring unit that satisfies the combustion ready area of the heat boundary. Then follows the same principle by considering the number of vertices in the subunit and fluctuate the size of the flame group accordingly. By doing this I have gained a well controlled flame distribution in this simulation.

3.7. Optimization

I believe combusting an object in a gaming environment should not take place in a rapid time frame. This process can be manipulated to be complete in its own time frame without interrupting the game play. In order to avoid consuming too many hardware resources of

the targeted machine where the game play takes place, I suggest the following techniques to optimize the process in the areas of heat boundary expansion, time division, block division, pre loading and using CUDA. These optimization techniques will be useful when it come to combustion of average high polygonal models under procedural fire generation.

3.7.1. Heat Boundary Optimization

When heat boundary expansion takes place throughout the model over time, the number of vertices to be deformed increases (active vertices). Increasing the number of vertices to be deformed means the number of operations increases rapidly. However, when the vertex reaches the inactive stage due to the level adjustments (Section 3.4), the number of active vertices inside the heat boundary decreases. Therefore, we can control the number of vertices to be deformed in a given time. Whenever, the number of vertices in the processing group reaches the maximum level, the expansion of heat boundary can be halted until a certain number of vertices become inactive. By doing so, we can control the use of resources in game play mode. In other words, by controlling the heat boundary expansion over time, we can maintain a balance between burning and burned materials and gain an upper hand in the performance aspect.

3.7.2. Time Division Optimization

I believe that by manipulating time according to the processing speed of the hardware most of the overhead operation can be avoided. Time division parameters (parameter that adjust according to the processing speed) can be apply to the heat boundary expansion process. As a common practice these types of parameters can be determined according to the frame rate of the game running. By applying time division, the rapid expansion of the heat boundary can be control systematically which eases the combustion and deformation process into a suitable level.

3.7.3. Block Division Optimization

I have introduced a block structure for the given model that has being used for structural deformation (Section 3.5.1). By deciding the number of subunits a certain model can control

the number of operation that has been used in the system. Applying block division to a model depends on the size of the model. However, using more subunits gives a more realistic essence in the simulation. Coming up with the number of adequate subunits is totally up to the developer. In this case, I divided the model, which contains 15000 polygons into 336 subunits. However, I have experienced that even when the model divided into 3375 (15 x 15 x 15) subunits it gives satisfactory results slightly over 30 fps due to the fact that the operations take place in the fairly modest CUDA supporting graphical hardware. Also, I have come to the conclusion that even though the whole bounding box is divided into a certain amount of subunits, I operate with only the ones that carry a weight (Section 3.5). Therefore, the shape of the model must be an important factor when one decides the number of subunits to be used.

3.7.4. Pre Loading Optimization

I suggest that loading all the necessary static data into the graphical hardware prior to the simulation is useful. In this simulation the Level count parameters, material indexes, density indexes (in case of different materials being used) and data structure of the subunits (used for structural deformation) etc. have been loaded to the graphical processing unit(GPU) when the model loading take place. By doing so, I was able to limit the data transfer back and forth between CPU to GPU and vice versa. This allows to gain the upper hand t the performance level and avoid over use of the data busses. By following such optimization, I have gain impressive results for this simulation. The detailes of the data structures have been used in the framework can be found in Appendix B.

3.7.5. CUDA Optimization

Since my primary objective is to come up with a method for deforming a polygonal model due to combustion while generating procedural fire for gaming applications; avoid utilizing most essential resources that are being used in the game play is important. Therefore, as I discussed throughout this chapter, I have occupied part of GPU and allowed the CPU to do more work in the game play. In terms of CUDA technology the developer has the upper hand

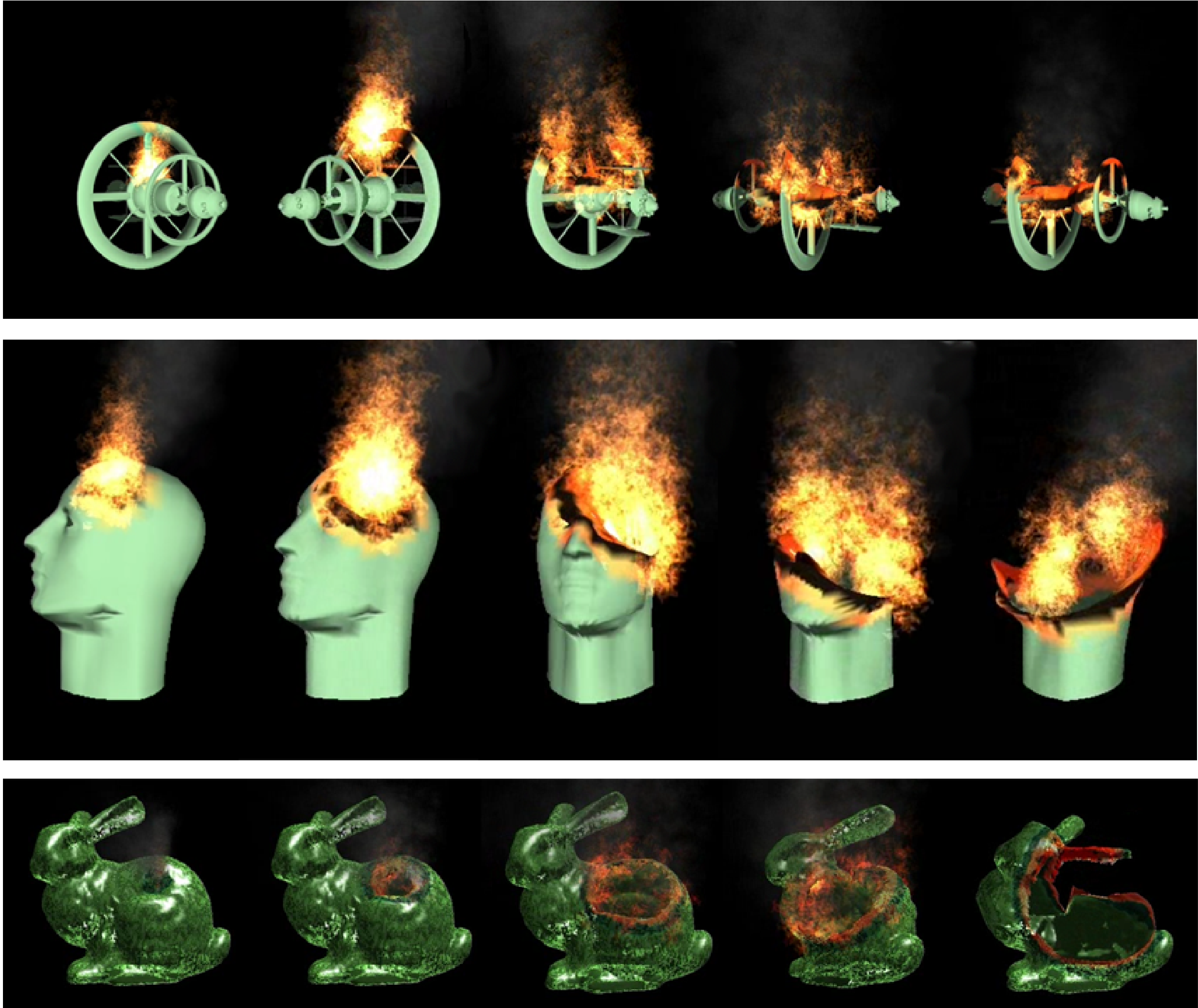

FIGURE 3.8. Animation frames for burning a model satellite (top),a head (center), and a bunny (bottom) using my algorithm in real-time.

of choosing optimized techniques to carry on such operation in the GPU. Its all depends on how to manage the memory as well as following the proper mapping techniques which are dependent on the process that are needed. I have followed some of the standard techniques described in CUDA in order to manipulate this simulation efficiently. I leave that to reader to figure out since the strategies and techniques vary for CUDA. The related code snippet supporting this framework can be found in Appendix B.

3.8. Results

Most of the images shown in this chapter were generated using the simulation described. Figure 3.8 shows five image sequence of animation of a deforming model objects. The flames

are generated using 3000 fire particles and 1800 smoke particles. However, the rendered particles at a given time are not up to the maximum. The model consists of 15949 triangles. The animation runs on 70fps (frames per second) on an Intel®Core™2 Duo CPU P8400 @ 2.26GHz processor with GeForce 9800 GTS graphical hardware unit. Results could be improved by tracking the sample particles over time using additional optimization techniques. This performance will be much better on the current graphical hardware since the performance of the parallel processing is quite high compared to a GeForce 9800 GTS GPU.

In this chapter, I have proposed a method for the real-time deformation and consumption of a polygonal model during combustion by procedural generated fire. I have focused on the performance with a reasonable amount of realism sufficient enough to attract the game player into the virtual world. I believe this method is the first of its kind. It takes into account a variety of physical properties including material density indexes, material indexes, heat distribution, gravity, structural and internal deformation, and flame distribution. My method described in this chapter, performs well on a model with fairly high polygon count and small triangles. It remains to apply my results to models with a low polygon count. I suggest that triangle subdivision is an intelligent first move. Most of the models used in video games are so called shell models. Deformation of shell models is different from solid models, and they should burn differently. In the rest of the chapters comprise with my proposal solutions to solve these problems.

CHAPTER 4

REAL-TIME RENDERING OF BURNING LOW-POLYGON OBJECTS

In this chapter I present a framework for modeling the deformation and consumption of low polygonal models under combustion while generating procedural fire. Many recent publications have shown variety of deformation techniques involved in computer graphics using the GPUs instead of using CPU alone. However, when it comes to low polygonal models, deformation is quite challenging since it is hard to maintain the realistic essence of such animation (Ex. Deforming a burning door consist of few triangles). On the other hand, use high polygonal models for every entity in the game environment may not be practical for most games due to the lack of resources and consumption of the processing power. Therefore, I suggest a method include mesh refinement in needy basis while maintaining both low and high end of polygonal aspects in balance. My focus is on trading realism for computation speed so that the processing power is available for other tasks, such as might arise in the current generation of video game.

4.1. Overview

In most of the video games, model deformation is essential to maintain the realism of the physical object behaviors. When it comes to maintaining the quality of video games, usage of high quality graphics is an inescapable necessity. However, due to the limitations of hardware resources and processing power in real time rendering, developers may have to choose detailed structure of game models carefully. One of the key features of such detailed structure is a number of polygons per model covered by pragmatic textures. As a trade-off between quality and performance, many game developers use low polygonal models for most of the flat surfaces in the game environment such as Doors, Windows, Walls etc. Due to this reason interaction between player and physical entities in the gaming world is restricted to some extent. Especially when deformation occurs, manipulating low polygonal models and maintaining the realism of the event is quite thorny. Therefore, when it comes to deformation of low polygonal models, model swapping techniques were commonly being used.

FIGURE 4.1. The consumption of a low polygonal model due to procedural fire.

Here, I am mainly focusing on general modeling of deformation and consumption of low polygonal models due to combustion. Fire simulations may be used effectively to increase the reality of visual effects in computer animations. Although, there is a little work has been done on model combustion and deformation. When it comes to the deformation of low polygonal models, triangle subdivision must be useful technique in many customs. However, complete model subdivision of each and every model is not a practical solution to accomplish my task. My main focus in this chapter is to introduce refinement method that could be used in deformation and real-time rendering of burning low polygonal models while maintaining the performance as well as realism in balance. My aim is to increase believability by a large amount with increasing computation sustain minimally while mesh refinement techniques have been used.

4.2. Previous Work

In the previous chapter 3, I have discussed techniques that were used object deformation due to fire, in gaming environments. However, this approach is appropriate mainly on the models that contain considerably high number polygons. When it comes to deformation and consumption of low polygonal models due to combustion, the related work published in this area is second to none. The obvious way to extend this technique to low-polygon models is to use real-time mesh refinement, subdividing triangles only when necessary. There are great amount of work has been done to optimize evaluation of various subdivision surface schemes in area of fast mesh refinement for real-time applications. Martin Wicke and Daniel Ritchie [56] introduces method of Mesh refinement to capture detailed physical behavior, fractures are simulated by subdividing mesh elements. Kai Hormann, UlfLabsik, [15, 22] discuss what kind of parameterizations are optimal for the purpose of remeshing. In geo-

desic remeshing using front propagation techniques introduce by Gabriel Peyré and Laurent Cohen [40], is quite similar to the method of processing adaptive mesh refinement while the heat boundary is expanding. The papers of Xiaohu Guo, Xin Li [20], and Lei He, Scott Schaefer [21] discuss subdivision parameterization of mesh surfaces. L. Giraud-Moreau1, H. Borouchaki1, A. Cherouat [6] introduces the real time application of deformation by applying remeshing to selective material. Denis Kovacs, Jason Mitchell [27] crease approximation contains useful information about surface subdivition. I have found Pierre Alliez1, Giuliana Ucelli [1] survey of recent developments in remeshing of surfaces quite useful to determined the recent advances in this area of research.

Conversely, little work has been published about hardware assisted implementation of subdivision schemes. T. Boubekeur and C. Schlick [6, 7] introduced useful mesh refinement techniques that has done using modern GPUs. Hsi-Yu Schive, Yu-Chih Tsai1, and Tzihong [45], uses the GAMeR technique using GPUs to introduce adaptive mesh refinement for Astrophysics. Fengtao Fan, Fuhua Cheng [14] and Christoph Fünfzig and Müller [47], discuss few techniques of subdivisions using modern GPUs.

4.3. Introduction to GAMeR

With the remarkable advancement of its computation power, GPUs are no longer limited only to scene rendering, but also GPUs have been used for other general purpose computing. Technology such as CUDA developed by NVIDIA is a distinctive platform for general purpose computation of GPUs that lead developers to utilize the computation power of modern generation of GPUs. However, as it mentioned by T. Boubekeur and C. Schlick [7], there are limitations when it come to data translation from CPU to GPU, since current graphic hardware is not able to generate more polygons than those sent through the graphics bus, by the application running on the CPU. Consequently, I have adopt T. Boubekeur and C. Schlick's idea of The generic adaptive mesh refinement (GAMeR) technique that will be used to virtually create additional inner vertices for a given polygon and further developed an algorithm that ensemble for my criteria. However, my framework is different from the GAMeR architecture described in T. Boubekeur and C. Schlicks paper [7]. Before I proceed

further, will look into the basic idea behind the GAMeR approach. GAMeR presents a generic vertex program that used to perform an adaptive refinement for meshes with arbitrary topology. For instance, starting from a stationary coarse mesh, this process replaces each triangle with a refined triangular patch that stored in GPU memory. As it mentioned in chapter 2, my framework is based on collaboration of CPU and GPU. First, the cores mesh of the model loaded into the CPU. Second, this serial data structure map into GPU prior to the immediate rendering. With CUDA supported GPUs, I was able to model the cores mesh with vertex placements and produced a totally different output. I adopt this concept and applied to the individual polygons of the mesh and advanced with an efficient subdivision scheme. The following Section presents the mesh refinement process and further details of this approach.

4.4. Subdivision Techniques

To this point, I have figured that GPU have ability to submit vertex placement prior to the screen render. In other words, GPUs are capable of scaling and placing polygons according to given calculations. GAMeR method has built upon this perception and gained rather fascinating results when it comes to mesh refinement process. The important factor in this procedure is that it is not required to have high number of polygons in the mesh to start with. However, GPU cannot produce additional polygons other than the ones in the graphic bus. Therefore, I have to provide demanded amount of polygons to support the refinement process. To fulfill these requirements, I have adopted additional mesh pool and bind required number of polygons from CPU side in needy basis. To make things easy on calculations, this pool of vertices could be arranged as a copy of ARP. CPU keeps ARP pool separate from the original VBO mesh and binds as a separate index buffer in order to load through the graphic bus. Extended description of the GAMeR method can be found in [7].

4.4.1. Refinement Patterns and Properties

Even though, the suggested approach is valid for other polygonal shapes, I only consider the usual case of triangular meshes in this Chapter. I have pre-computed the possible

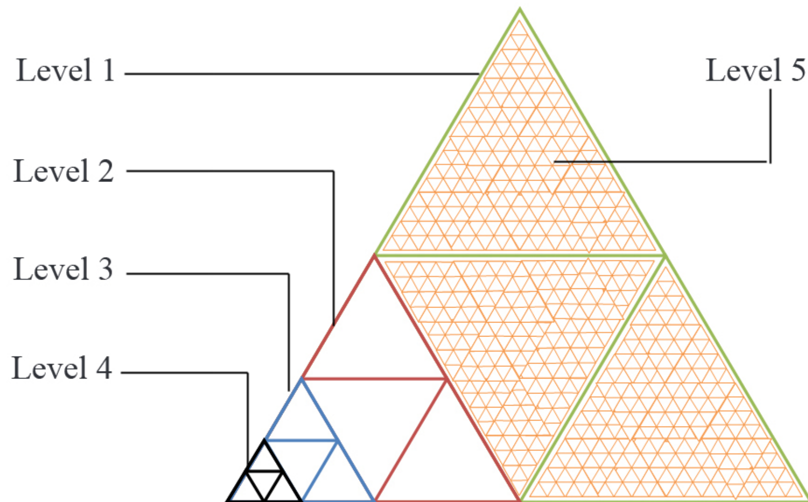

FIGURE 4.2. The level subdivision of a single triangle.

refinement configurations of a single triangle. The subdivision approach is processed using uniform decomposition where the subdivision takes place in all the cells recursively. I have use isotropic template which divided each vertex into half for five recursive levels in depth as it illustrate in figure 4.2.

Recall that my objective is not to subdivide each and every triangle in the object. My aim is to subdivide only when necessary, and prior to deformation. Therefore, we needed to gain more control over the refinement process before concede further. In order to manipulate much sophisticated refinement process I have declared some attributes to each of the subdivided triangles as it shown in Table 4.1.

Attribute	**Values**	**Description of each Attribute**
id	*Integer*	Track siblings and parent
SetLevel	$1, 2, 3, 4, 5$	Depth of the polygon division
Siblings	*Integer*	Each has three siblings
Parent	*Integer*	Parent id
Active	$-1, 0, 1, 2$	Status of the triangle

TABLE 4.1. Adaptive refinement pattern (ARP) attribute set.

One of the important attributes in this set is *status* of the triangle. In this value set -1, 0, 1, 2 represent triangle's status as *inactive*, *initial*, *active*, and *processed* respectively. The

renderer will draw only the final ARP of active and processed triangles generated by each coarse triangle. I discuss about these attributes further alone with my proposed algorithm in Section 4.4.3.

After loading ARP and its attributes to the vertex buffer, I needed to map ARP coordinates to the qualified coarse polygon using displacement maps similar to figure 4.2. Unlike T. Boubekeur and C. Schlick [7], I have recorded the final coordinates set into the GPU since my approach of deformation has lot more to proceed prior to rendering the final output. At this point I apply ARP to the coarse polygon only if is eligible to proceed to next level of subdivision. This eligibility is depends upon the location of the heat boundary relative to the triangle.

4.4.2. Barycentric Points and Heat Boundary

My objective is to perform an interactive, believable simulation of a low polygonal burning object. When a primary occasion is combustion, it is necessary to address heat boundary expansion of the object. As a fact, the temperature increase due to combustion influences the mechanical behavior of the object. Likewise, the thermal conductivity of the object influences the thermal response (Chapter 3). To speed up computation, I have decided to proceed with approximated single point heat boundary. I have followed the same functionality suggested in chapter 3. The approximated heat boundary expansion was given by:

$$R^2 = |\sin(\pi\Theta/\Delta r) + \sin(\pi\Theta) + \psi((x - x_0)^2 + (y - y_0)^2 + (z - z_0)^2)|,$$

where $R = r + \Delta r$, the radius r is incremented by Δr in each Δt time period. The angle Θ is a random value in order to make the expanding heat boundary irregular in shape. The location of the heat source is (x_0, y_0, z_0). However, in this approach, the value of heat index constant ψ, which is supposed to be a constant that depends only on the size of the coarse triangles of the model, is no longer fixed. Therefore, I let the designer set ψ depending on how many levels of subdivision are planned.

It remains to decide which coarse triangles are eligible for subdivision. This has to be

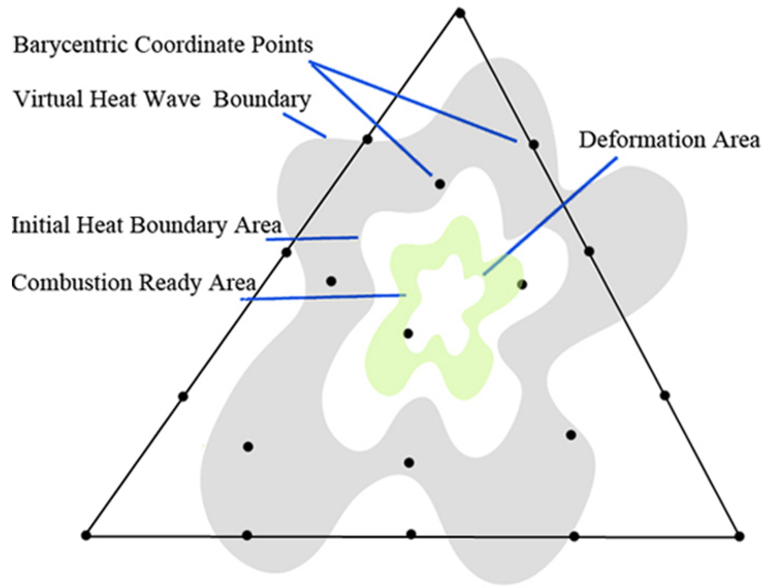

FIGURE 4.3. Heat boundary areas and barycentric point sets.

a function of the expanding heat boundary. Furthermore, the subdivision has to take place prior to the deformation process. My solution is to send a virtual heat wave through the model prior to the actual heat boundary expansion. This creates an area in addition to the three initial heat boundary areas described in chapter 3. Since the introduced boundary expansion takes place prior to the three original expanding boundaries (see Figure 4.3), I can proceed with the subdivision of qualified triangles before the deformation process begins. Since I am using a single source heat boundary, temperature at all points will depend on the distance from the heat source at (x0, y0, z0). If this point is in the middle of one of the coarse triangles, the triangle will not be eligible for subdivision until the virtual heat boundary hits one of its vertices. To avoid such issues I represent each triangle using barycentric coordinates as follows.

Suppose point $P = (x, y, z)$ is given by:

$$
\begin{aligned}
x &= \lambda_1 x_1 + \lambda_2 x_2 + \lambda_3 x_3 \\
y &= \lambda_1 y_1 + \lambda_2 y_2 + \lambda_3 y_3 \\
z &= \lambda_1 z_1 + \lambda_2 z_2 + \lambda_3 z_3,
\end{aligned}
$$

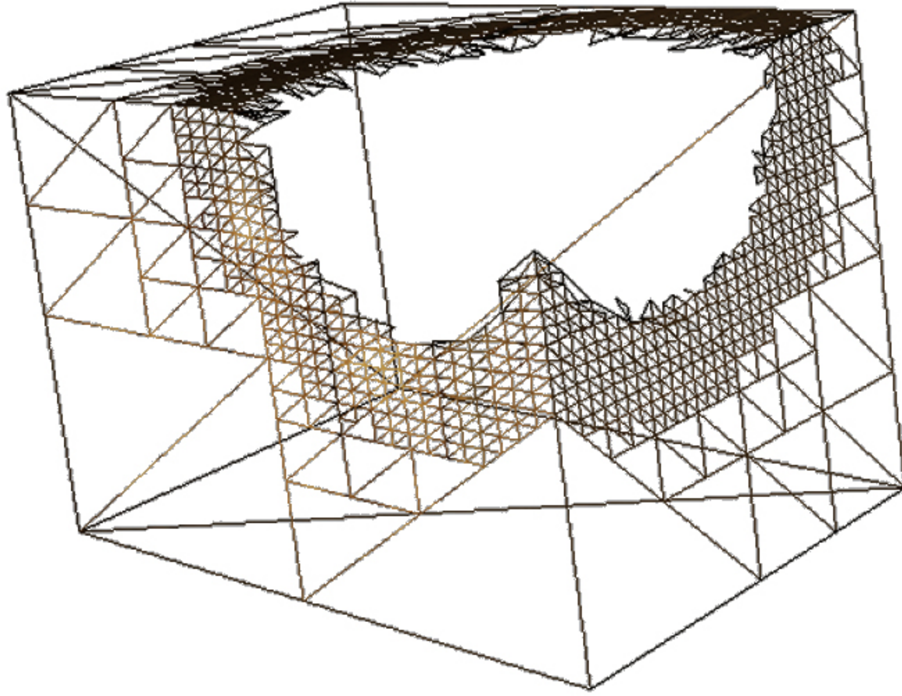

FIGURE 4.4. The refinement hierarchy and deformation applied to a 12-triangle model of a box.

where λ_1,λ_2 and λ_3 are area parameters such that $\lambda_1 + \lambda_2 + \lambda_3 = 1$.

I need to calculate the barycentric coordinates for non-eligible triangles only, where *eligible* triangles are those that are close to the heat boundary. The following algorithm returns `true` if coarse triangle T is eligible.

for each coarse triangle T

if T is not eligible **then** get barycentric point set P

for each barycentric point set P

if P is inside the heat boundary

return `true`

4.4.3. Deformation

After applying ARP to the eligible triangle, I next apply deformation techniques. Although my ARP arbitrarily contains triangles of five levels in depth (see figure 4.2), this number can be changed in the obvious fashion by the choreographer. Deformation applies only to the final level (in this case, fifth level) status *active* triangles. At this point, rendering all levels of triangles in the ARP will be costly and wasteful. Instead, I choose which triangles to render using the ARP attributes listed in Table 4.1.

The process can be described informally as follows. Initially, the coarse triangles of the model are considered *active* (status value 1) triangles, and all ARP triangles are initialized as *initial* (status value 0) triangles. Each subdivided triangle consists of three siblings and a parent. As my algorithm proceeds, if one of the child triangles turns *active*, then the parent will turn *processed* (status value 2) until all of its children also become status *active*. Once its children have all turned *active*, the parent triangle will change its status from *processed* to *inactive* (from status value 2 to -1). More specifically:

if triangle has SetLevel = 5 and Status = 0
 if triangle is inside the heat boundary
 Status := 1, Status of all siblings := 1
 Status of parent := 2

if SetLevel <5 and Status >0
 if sibling's Status >-1
 sibling Status := 1
 parent Status := 2

if all children have Status = 1 and parent has Status = 2
 parent Status := -1

In my high-polygon deformation algorithm (chapter 3), the displacement of each vertex depends on the surrounding vertices. Therefore, to apply proper calculation of deformation,

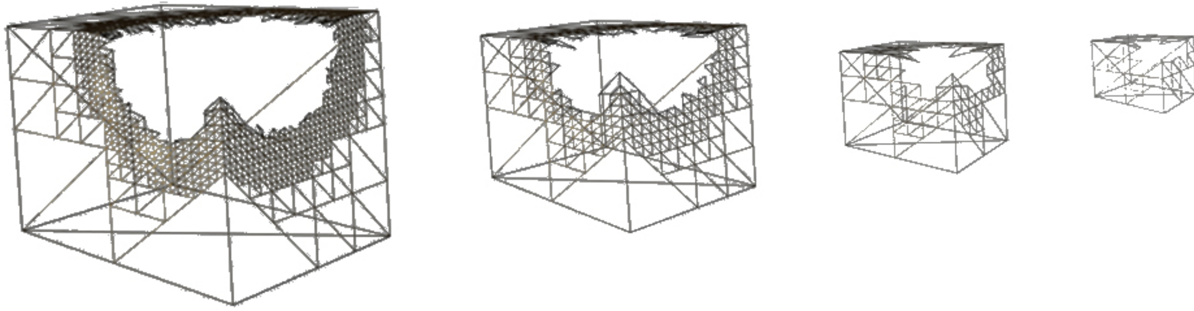

FIGURE 4.5. Real-time computation of LOD for a burning object.

we must let the subdivision proceed a few steps further before applying deformation to the mesh. By doing so, I was able to calculate the proper strength factors within the deforming triangles properly. Figure 4.4 illustrates the refinement hierarchy and deformation applied to a low-polygon model of a box.

4.5. Level Sets and Distance

In a game environment, objects located far from the viewer need not be rendered in as much fine detail as those close up. A significant speed-up can be obtained by having models stored at various Levels Of Detail (LOD) ranging from, for example, hundreds of triangles for objects in the far distance to tens of thousands for close-up objects. These variants of the model are usually created by the artist, although procedural methods do exist.

My algorithm allows us to implement LOD for burning objects by controlling the level of adaptive refinement of the coarse mesh triangles. I calculate the distance between object and the player in the CPU and pass it to the GPU as a parameter. Level adjustment is decided and passed to the appropriate ARP before rendering. Figure 4.5 illustrates the burning box model at different LODs.

For a solid object the level of refinement is directly proportional to the distance. However, surface removal and deformation of a burning object makes it slightly more challenging to maintain a smooth transition between level swaps. Define the *burn level* of a model as the number of triangles of the model that have been consumed by fire (chapter 3). Then use the following algorithm to determine whether to render triangle T.

if number of children of T with

higher burn level than T is ≥ 2

and SetLevel ≥ 5

then hide T

else show T

4.6. Experiments and Results

The images of a burning box shown in this paper are screenshots from a CUDA implementation of my algorithm applied to a model with 12 triangles. The flames are generated using 2000 fire particles and 500 smoke particles. The advantage of such a system is clear when comparing the resources required to deform a completely subdivided model versus deforming a low-polygon model using my method. Table 4.2 shows the frame rates of the animation when my algorithm is implemented in CUDA on relatively modest hardware; An Intel®Core™2 Duo CPU P8400 @ 2.26GHz processor with an NVidia GeForce 9800 GTS graphics card. This performance will of course be much better on the current generation of graphics hardware, but that is not my aim. My aim is to provide detail sufficient to trigger willing suspension of disbelief at a relatively low cost in computation load.

The outcome of these experiments shows that my method results in doubling the frame rate. Therefore, I believe this approach is a better alternative than subdividing the complete model when it comes to deforming low-polygon models.

Polygon Count	Fully Subdivided	Our Method	Speed-up Factor
10k	84fps	165fps	1.96
15k	76fps	159fps	2.09
20k	63fps	153fps	2.43
50k	48fps	60fps	1.25

TABLE 4.2. Frame rate of fully subdivided model versus my approach.

I have proposed a method for the real-time deformation and consumption of a low-polygon model during combustion by procedurally generated fire. By doing so, I have

extended my work in chapter 3 to low-polygon models. I have performed simulation of real-time deformation and consumption of any model regardless of the size of the triangles. My simulations have performed well on a model with low-polygon count and large triangles.

Most of the models used in video games appear to be shell models, with a hollow interior. Deformation of shell models is different from solid models. I intend to investigate the extension of my method to solid models.

CHAPTER 5

REAL-TIME RENDERING OF BURNING SOLID OBJECTS

Many cutting edge console and PC games compete to attract seasoned players by increasing realism over and above what they are accustomed to in other games. Replicating the details of a physical process such as fire can readily draw the player's attention. The typical object in 3D video game is represented by a polygonal mesh in the shape of its surface, which called a shell model. The problem of applying shape changes to a shell model by emulating solid object properties without overloading the available computational resources is a challenging one. Here, I am exploring how to manipulate any given shell structure by applying solid model characteristics under deformation circumstances. Solid objects do in fact burn rather differently than shell objects. My objective is to gain believable visual simulation that could clearly distinguish solid deformation from shell deformation. In this particular case, I propose to tackle the emulation of solid object deformation and consumption under combustion.

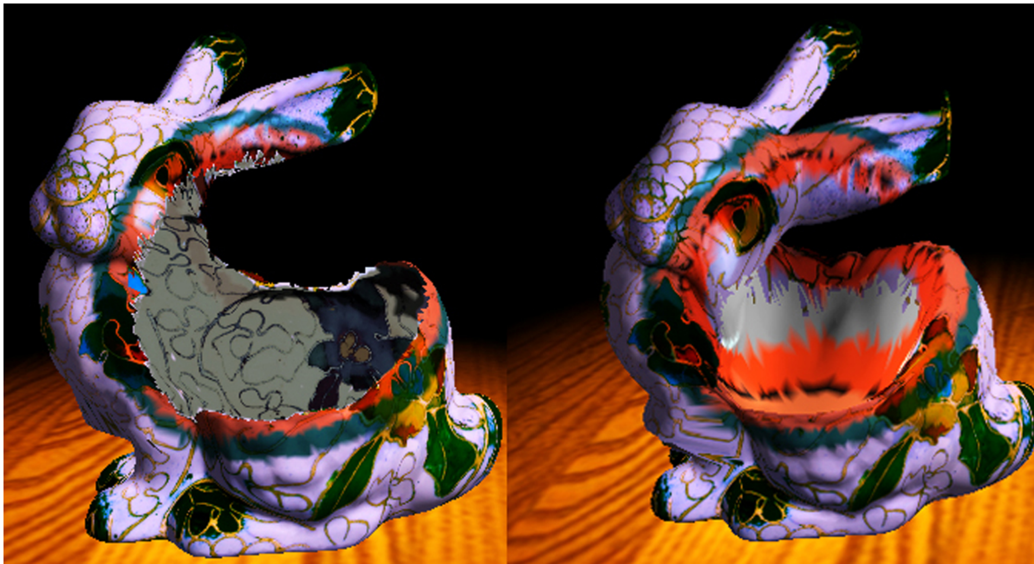

FIGURE 5.1. Shell model deformation (left) vs solid model deformation (right).

5.1. Overview

In this chapter, I describe a method for the real-time deformation and consumption of a solid model during combustion by procedurally generated fire, extending my previous work described in preceding chapters 3 and 4. Solid objects are expected to burn much differently than shells. Aside from the obvious difference of being able to see the inside of a burned-out shell (figure 5.1, left), a solid object will melt and deform under heat in a different way (figure 5.1, right).

5.2. Previous Work

This chapter extends my previous work on the emulation of burning objects in video games. Also as in publications, Amarasinghe and Parberry [3] laid down the foundation of my approach and demonstrated the ability to realistically burn in real time on a relatively slow GPU a high-polygon count shell model of a toy satellite. Amarasinghe and Parberry [2] extended this work to models with a very low polygon count by judicious use of procedural triangulation in the areas that are on fire, and demonstrated the ability to realistically burn on the same GPU a 12-triangle shell model of a door. This approach also lent itself easily to dynamic level of detail rendering. Model deformation is a popular topic in the computer graphics community. I single out the following papers as relevant and significant, but without exception they strive for realism at the cost of performance. Although they are more realistic than my approach, their methods are not real-time and are therefore more useful for offline applications such as motion pictures than for video games. Melek and Keyser [32, 33] discuss techniques that were used in selected object deformation due to fire. Demetri Terzopoulost and John Platt [50] introduce the theory of elasticity to describe deformable materials such as rubber, cloth, paper, and flexible metals. Sederberg and Parry [46] introduce a technique for deforming solid geometric models in a free-form manner. E. B. Tadmor and Rob Phillips [49] and Nealen *et al.* [37] use finite element methods to deform complex geometries. Toivanen [48] discusses free deformation of meshes. Finally, Nguyen Rasmussen and Fedkiwr [38, 41] introduce high quality flame simulations that I use in my experiments, but they do not address object deformation.

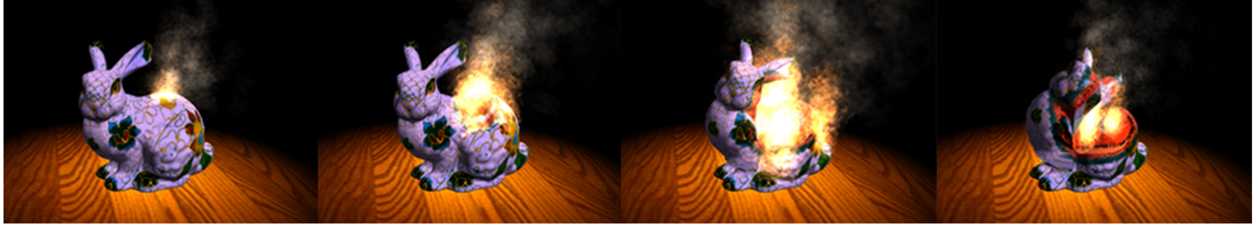

FIGURE 5.2. The combustion of a solid model and the spread of procedural fire.

5.3. Internal Deformation

According to Melek and Keyser [32], when an object burns there are assorted interior chemical reactions at various stages that lead its properties to change in a process called *pyrolysis*. Volumetric expansion of heated material is caused by weakening bonds at the molecular level. Internal forces are disturbed by the effect of heat on unstable bond structure, ultimately leading to the consumption of material. This causes changes in the shape of the object's affected areas. I begin by creating a simplified model of heat spread.

5.3.1. The Heat Boundary

As I discussed in previous chapter 4, I approximate the expansion of the heat boundary by calculating it around a fixed solitary point, using the same function placed as:

$$R^2 = |\sin(\pi\Theta/\Delta r) + \sin(\pi\Theta) + \psi((x-x_0)^2 + (y-y_0)^2 + (z-z_0)^2)|,$$

where $R = r + \Delta r$ indicates that the radius r is incremented by Δr in each Δt time period. The angle Θ is a random value in order to make the expanding heat boundary irregular in shape. The location of the heat source is (x_0, y_0, z_0). The heat index also can be approximated by a constant that depends on the size of the coarse triangles of the model.

In this chapter I address the combustion of solid models with an arbitrary number of polygons. If the targeted triangle is considerably larger than the rest of the triangles, I can always apply the subdivision techniques discussed in chapter 4. Thus, the designer can maintain a fixed heat index value that is suitable for the model and maintain the subdivision level accordingly.

As shown in figure 5.3, I divide the heat boundary into four different areas. The *Virtual*

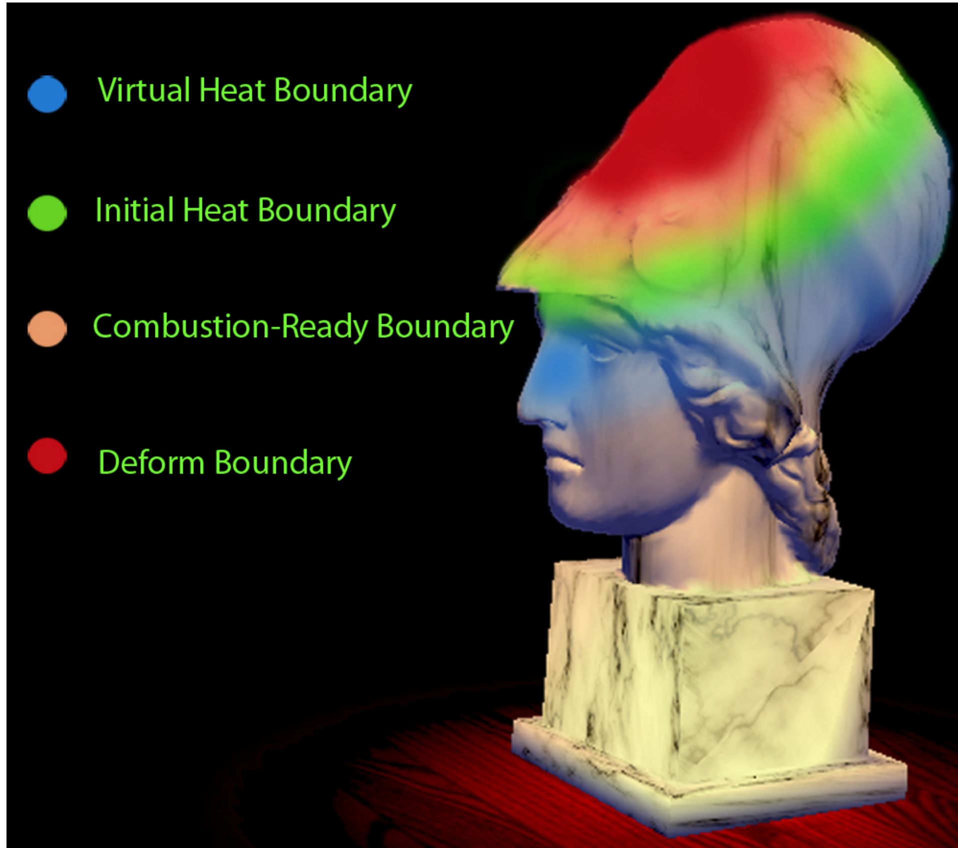

FIGURE 5.3. Heat boundary with different levels of boundary.

Heat Boundary is spread through the model prior to the actual heat boundary expansion and is used to amortize essential calculations that could apply to the qualified triangles before the deformation process begins. The other three boundaries are similar to those introduced in chapter 4; where the *Initial Heat Boundary* in which combustion is actively taking place and vertices are preparing to be deformed, the *Combustion Ready Boundary* where ignition starts, and the *Deform Boundary* consisting of material that has been burned.

5.3.2. The Deformation Process

Surface removal as practiced in the prior chapters is less useful in solid models than in shell models because the consumption of material in a solid model simply reveals more material just underneath it. Consequently solid models have more triangles to deform than shell models, and these need to be managed efficiently and effectively. In order to achieve this I categorize model triangles into three major types as shown in figure 5.4. Those are

called boundary qualified triangles, combustion qualified triangles, and deforming triangles. *Boundary qualified triangles* are the triangles located inside the virtual heat boundary. These can be completely or partially contained within the heat boundary, depending on the size of the triangle. If the latter is the case, triangle subdivision must take place. *Combustion qualified triangles* are the ones that are ready to take part in the first round of deformation.

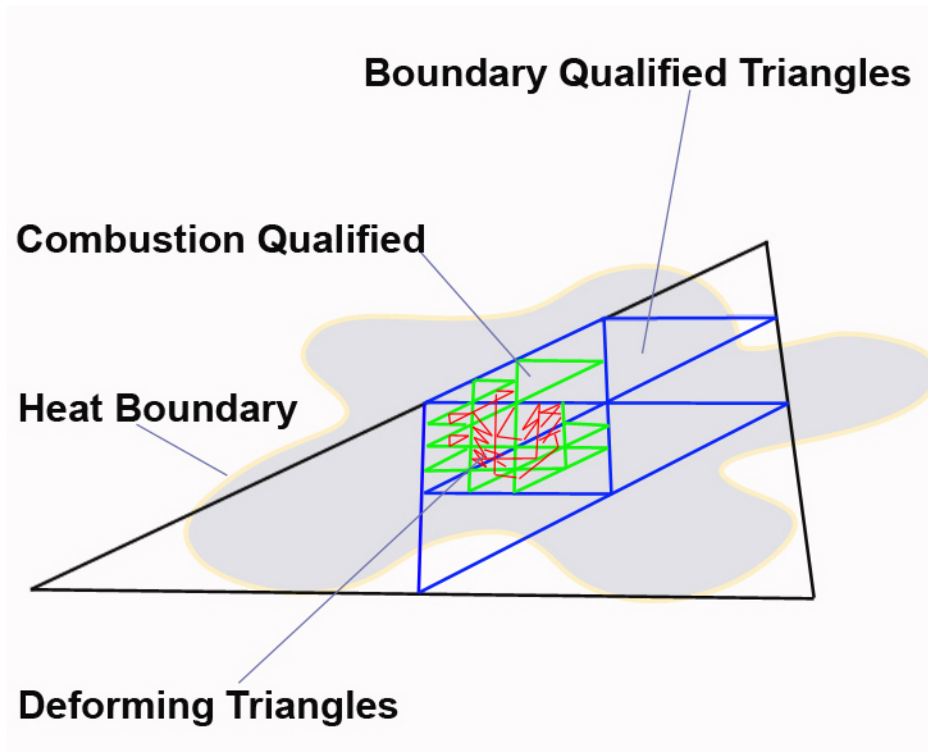

FIGURE 5.4. Categorized model triangles.

5.3.3. Inward Contraction Displacement

In shell models the heat-induced deformation of an object is achieved by displacement of the vertices of the model mesh (see chapter 3), where the position of each vertex depends on given properties such as vertex distance, gravitational force, and material index, and the internal forces work on the triangle pointing towards the direction of its vertices. However, when the model represents a solid object we must also apply *inward contraction forces* to the vertices. In burning objects, the extending heat waves weaken the bond strength between adjacent molecules. This weakening effect falls off as a function of the distance from the

heat source. As a result, surface molecules move towards the stronger bonds in order to find stable equilibrium between the acting forces. This results in contraction of the burning area of the object.

Melek and Keyser [32] also noted that due to multiple internal chemical reactions at various stages of the combustion process, material may change state from solid to liquid and from liquid to gas. Both these cause reduction of the mass in affected areas of the burning object. In most cases this will cause an inward concave shape in the consumed area. To illustrate this phenomenon in a simulation I have applied what I call the *Inward contraction displacement technique* to calculate the inward movement of the vertices of the deforming triangle. The idea of this technique is to identify for each triangle a virtual point covered by the affected polygonal boundary in distance (see figure 5.5) and use this to calculate the the local inward displacement.

First we must identify the inward direction of the combustion qualified triangle or the deforming triangle. Secondly, the distance of the virtual point must be proportional to the size of the qualified triangle. However, calculating random virtual points to meet the necessary requirements on continuously deforming polygons is not an efficient solution. Therefore, my best approach to succeed this task is to employ the face normal of the object and calculate the inverse directional coordinates. To maintain the proportional distance between the virtual point and the triangle surface, I factor the normal vector coordinate by the length of either side of the triangle (d_1 or d_2 in figure 5.5).

That is,

$$(X_{in}, Y_{in}, Z_{in}) = -D \cdot (X_{fn}, Y_{fn}, Z_{fn}),$$

where (X_{in}, Y_{in}, Z_{in}) is the inward contraction point and (X_{fn}, Y_{fn}, Z_{fn}) is the face normal of the targeted polygon. The distance of the either side of a polygon is represented by D. Deforming triangles are the triangles that actually performing the deformation of the burning object. The displacement of its vertices is addressed in the following subsection.

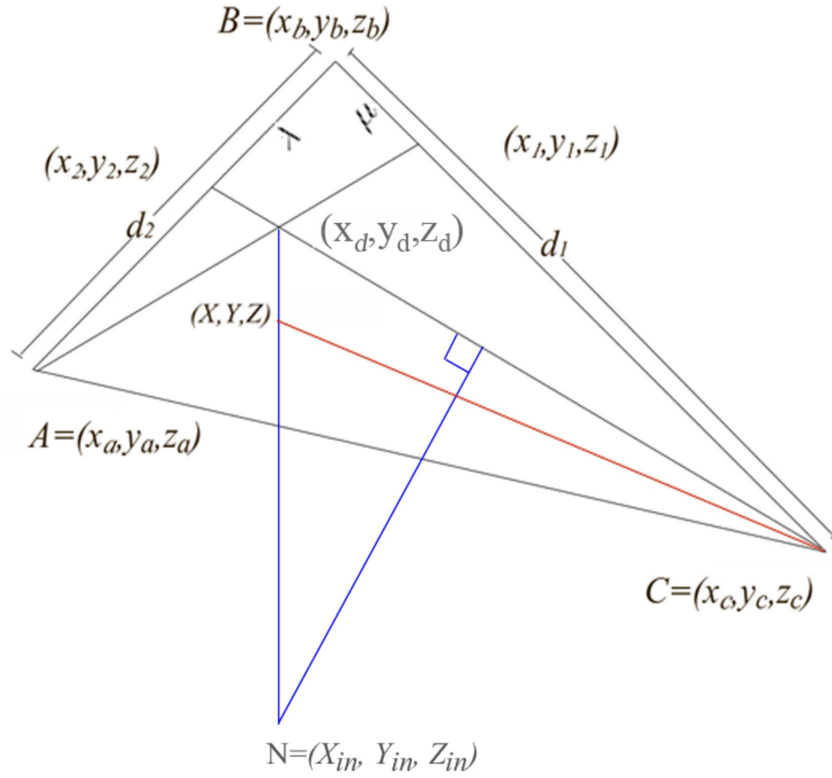

FIGURE 5.5. The deformation coordinates of a single triangle.

5.3.4. Vertex Displacement

Suppose B is a vertex to be displaced in triangle ABC, where $A = (x_a, y_a, z_a)$, $B = (x_b, y_b, z_b)$, and $C = (x_c, y_c, z_c)$. B is to be displaced to (X, Y, Z), as follows:

$X = (x_1x_2(y_a - y_c) + x_1x_a(y_c - y_2) + x_cx_2(y_1 - y_a) + x_ax_c(y_2 - y_1))$
$/((x_a - x_2)(y_c - y_1) - (x_c - x_1)(y_a - y_2))$ (similarly for Y and Z as it described in chapter 3), where
$(x_1, y_1, z_1) = \mu C + (d_1 - \mu)B$ and $(x_2, y_2, z_2) = \lambda A + (d_2 - \lambda)B$.

Figure 5.5 illustrates the coordinates and parameters used in these equations. A detailed description of above function and its parameters can be found at section 3.4 of chapter 3.

The final displacement value of X, Y, Z can be calculated as follows:

$$X = \lambda_b \cos(\Theta) \sin(\alpha)$$
$$Y = \lambda_b \sin(\Theta)$$
$$Z = \lambda_b \cos(\Theta) \cos(\alpha)$$

where

$$\alpha = \tan^{-1}(X_d - X_{in}/Z_d - Z_{in})$$
$$\Theta = \tan^{-1}(Y_d cos\alpha/Z_d - Z_{in})$$

λ_b is either value of λ or μ depending on the corresponding distance that taken for D as d_1 or d_2. Furthermore, Let ε be a constant that represents the amount that the model melts due to heat, and $\vec{g}$ be the gravity vector. Then the effect of gravity is computed as: $Y = Y - \varepsilon\vec{g}$.

5.4. Structural Deformation

As I described in The structural changes in a burning object are the result of various factors including the expansion and the weakening of the internal bonds, and the relative weights of cantilevered parts of the object 3.5. The precise calculation of these complex processes is costly. Therefore, I adopt the *block sampling method*method discussed in chapter 3 as a computationally less expensive solution to maintaining realism while performing systematic structural change. The block sampling method divides the object into uniform blocks and treats each block as a single unit, propagating changes to neighboring blocks.

The difference with burning solids is that there are no surface removal techniques associated with burn level adjustments. Furthermore, the weight changes of each block are not significant enough without the effect of level adjustment. As a solution for these concerns, I maintain a counter to monitor the time of combustion per each block. Weights of the blocks are decided according to the number of vertices factored with the counter. The empty ones

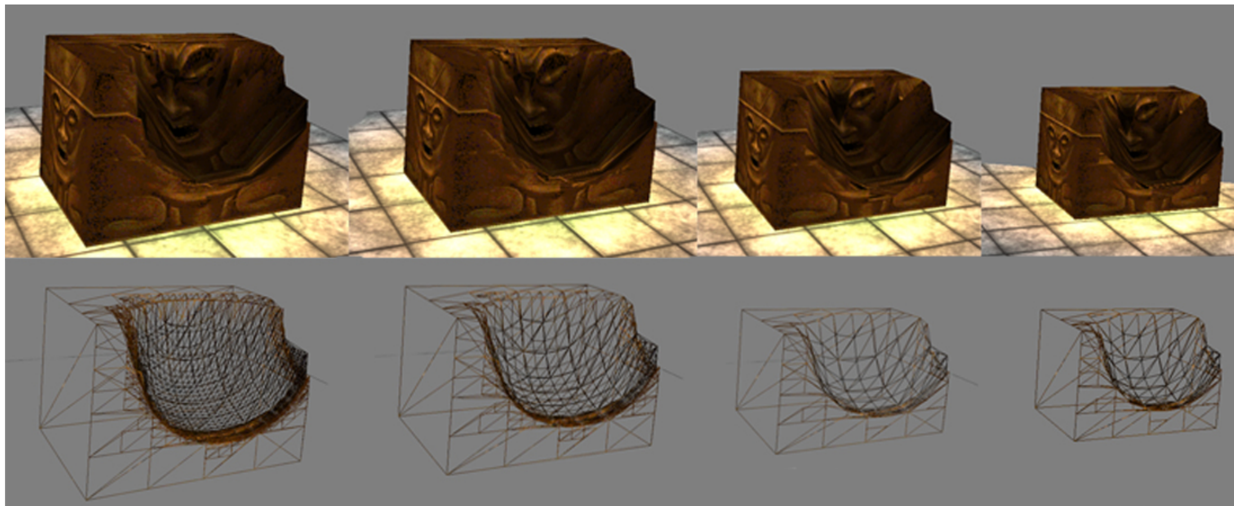

FIGURE 5.6. Level of detail (LOD).

of weight zero are discarded.

I keep track of the orientation of each block as a triple of Euler angles. The change in roll angle R (pitch and yaw are similar) for a block is: $R = \gamma\rho\pi/NM$, where γ is a scaling factor chosen by the designer, ρ is a measure of the material density of the model in that block, N is the number of vertices in the block, and M is the current number of nonempty neighboring blocks.

5.5. Results and Optimization

I have implemented automatic LOD rendering into my simulation using techniques presented in chapter 4. Figure 5.6 illustrates my LOD algorithm applied to the burning of a solid block of wood. The images shown in this chapter are from a CUDA implementation of my algorithm applied to different models. Since there is no strict time line for the combustion of the model, I can always control complexity of the simulation by limiting the number of deforming triangles at a time. Optimization is possible since the deformation is always applied mostly to the affected areas of the object. The continuous deformation of given polygon can be controlled by parameter settings such as the *flammability* value L. In particular, the shape and size of a deforming triangle can be drastically changed. Overly-exaggerated deformation reduces realism. In order to maintain efficient simulation without heavy resource usage, once a deforming triangle's flammability value L exceeds some limit I remove

the polygon from the group of deforming triangles and add more from the set of combustion qualified triangles into the group. By following this practice I gained more control over the simulation with better performance while maintaining realism.

I used approximately 2000 fire particles and 500 smoke particles to demonstrate the visual effects. The algorithm was implemented in CUDA on relatively modest hardware; An Intel®Core™2 Duo CPU P8400 @ 2.26GHz processor with an NVidia GeForce 9800 GTS graphics card. I was able to maintain 60fps frame rate up to 45k triangle model with balanced settings (quality vs. performance) in the graphic card. This performance will of course be much better on the current generation of graphics hardware, and thus able to run in parallel with other rendering tasks and game-related computation. I was able to successfully perform my simulation on models of various mesh resolution and topology on less than cutting-edge hardware. These simulations perform well on various models ranging from a dozen to hundreds of thousands of triangles. When it's come to incineration of objects, the matter of melting cannot be ignored. I will discuss about possible solution criteria for this issue in the next chapter.

CHAPTER 6

REAL-TIME RENDERING OF MELTING OBJECTS

I present a method for simulating the melting and flowing of material in burning objects fast enough to be of use in video games where most of the graphical and computational resources are needed elsewhere. One of the major goals in animation research is to be able to simulate the behavior of real-world materials in practical situations. The standard practice of using particle engines or fluid dynamics for melting are far too costly for use in this environment. Furthermore, in the game industry we require simulations that are visually compelling but not necessarily fully accurate imitations. My aim is to achieve a visual effect that is as close as possible to that of scientific simulations, but at a small fraction of the computational cost. I demonstrate that my method, which is based on systematic polygonal expanding and folding, uses only a fraction of the computational power available by implementing the computation on a very modest GPU using CUDA.

6.1. Overview

I commit this chapter to propose a method for emulating how solid objects melt under combustion. Each object in a 3D video game is represented by a polygonal mesh in the shape of its surface, which I will call a *shell model*. The challenge is to maintain a natural fluid-like behavior in this polygonal structure. Unlike any other free form defamation, melting simulations are required careful monitoring measurements to maintain its realism. Especially, when it comes to viscoelastic fluid like behavior in certain polygons in a rigid mesh. In this chapter, my goal is to introduce a polygon refinement method that can apply to any shell model (Figure 6.1, left), and support compelling visual replication of the melting process through to a burned-out and melted version of the object at the end (Figure 6.1, right).

6.2. Previous Work

Model deformation is a popular topic in the computer graphics community. I single out the following papers as relevant and significant, but without exception they strive for

realism at the cost of performance. Although they are more realistic than my approach, their methods are not real-time and are therefore more useful for offline applications such as motion pictures than for video games.

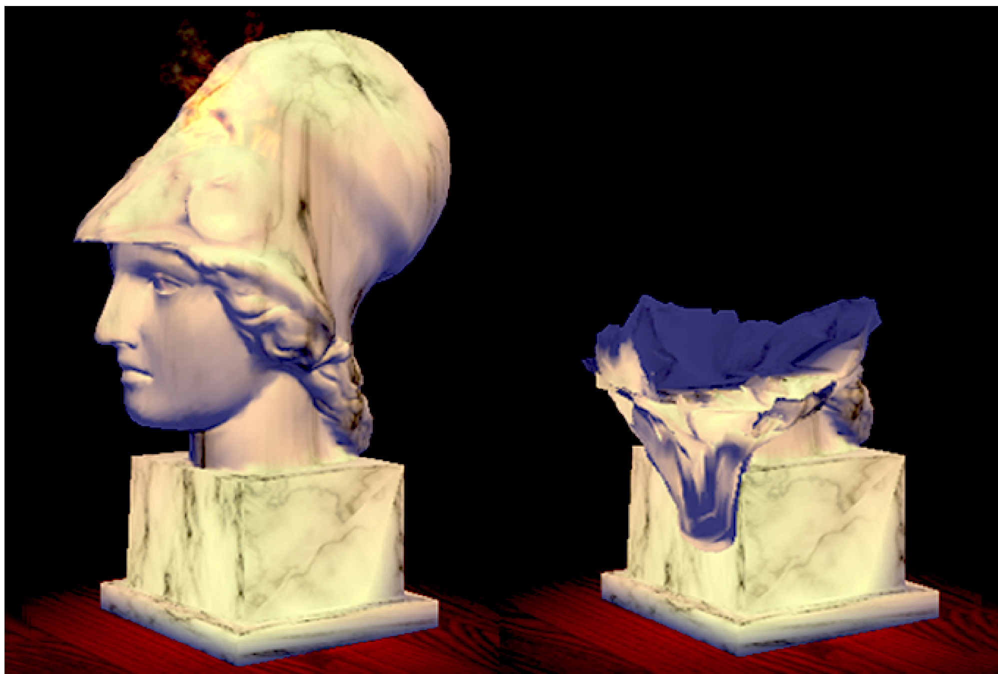

FIGURE 6.1. A wax like solid model (left) and it's melting effects taken place (right).

Particle simulations are most popular techniques used to achieve fluid-like effects in computer graphics. These use a high number of small particles. Examples include Iwasaki [24], Foster and Fedkiw [16], Carlson, [11], Clavet [12], Keiser and Adams [26], and Goktekin and Bargteil [18]. Losasso and Irving [30] introduce a method for liquid or gas simulation using grid-based techniques including vortex confinement and the particle level set method. Müller, Keiser, *et al.* [35] laid a point-based framework for the volume and the surface representation, allowing arbitrarily large deviations from the original shape. Wei and Li [54] introduce a 3D cellular automata approach for melting objects. Terzopoulos and Platt [51] discuss deformable models featuring non-rigid dynamics governed by Lagrangian equations of motion and conductive heat transfer governed by the heat equation for non-homogeneous, non-isotropic media.

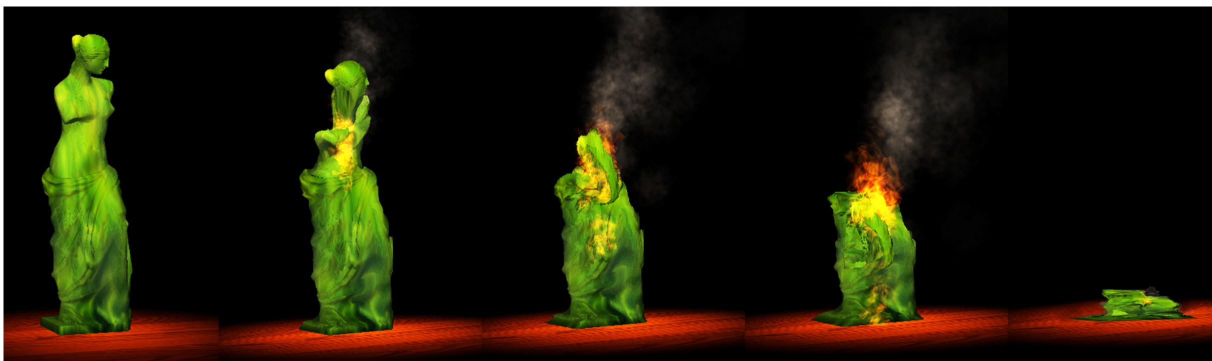

FIGURE 6.2. The melting of a solid model and the spread of procedural fire.

Melek and Keyser [32, 33] discuss techniques that were used in selected object deformation due to fire. Terzopoulost and Platt [50] introduce the theory of elasticity to describe deformable materials such as rubber, cloth, paper, and flexible metals. Sederberg and Parry [46] introduce a technique for deforming solid geometric models in a free-form manner. Tadmor and Phillips [49] and Nealen *et al.* [37] use finite element methods to deform complex geometries. Toivanen [48] discusses free deformation of meshes.

Finally, Rasmussen and Fedkiwr [38, 41] introduce high quality flame simulations that I use in the screenshots and videos, but they do not address the topic of object deformation.

6.3. Internal Deformation

Numerous chemical reactions take place in the interior of a burning object caused by the heat of combustion. Melek and Keyser [32] use the general term *pyrolysis* to describe this process. Volumetric expansion of heated material is caused by weakening bonds at the molecular level. Internal forces are disturbed by the effect of heat on unstable bond structure, ultimately leading to the consumption of material. Both thermal flow and the latent heat during the phase change bring the model to a state of melting (Jones, M.W. [25]). This causes changes in the shape of the object's affected areas.

The remainder of this section is divided into three subsections. Section 6.3.1 begins with a review of a simplified model of heat spread from chapter 3. Section 6.3.2 gives an overview of the deformation process and defines three new vertex classes, Flow True Vertices, Fixed Point Vertices, and Base Point Vertices. Section 6.3.3 investigates the melting process at the

triangle level using Flow True and Fixed Point Vertices. Section 6.3.4 describes the motion of Base Point Vertices.

6.3.1. The Heat Boundary

I have used the same approximated heat boundary expansion presented in chapter 3. In this chapter I address the combustion of models with a large number of polygons. If a particular targeted triangle is considerably larger than the rest of the triangles, I can always apply the subdivision techniques discussed in chapter 4. Thus, the designer can maintain a fixed heat index value that is suitable for the model and maintain the subdivision level accordingly.

I divide the heat boundary into four areas described in Chapter 5.

(1) The *Virtual Heat Boundary* is spread through the model prior to the actual heat boundary expansion and is used to amortize essential calculations that could apply to the qualified triangles before the deformation process begins.

(2) The *Initial Heat Boundary* is the area in which combustion is actively taking place and vertices are preparing to be deformed.

(3) The *Combustion Ready Boundary* is where ignition starts.

(4) The *Deform Boundary* consists of material that has been burned.

In order to implement melting, it will needed to add more computation within these boundaries as described in the remainder of this paper. With composite simulations such as melting, one major challenge is to maintain the relative size of triangles so as to avoid empty triangles or sliver triangles. I will adapt the subdivision procedure introduced in previous chapters to the areas affected. These calculations will be applied to the essential triangles inside the virtual boundary.

6.3.2. The Deformation Process

The position of each vertex depends on given properties such as vertex distance, gravitational force, viscosity damping force (Wei and Li [54]), and the internal forces work on each triangle pointing towards the direction of its vertices. However, when the model starts to melt I must also consider the effect of fluid-like behavior on the movement of the vertices.

Although in particle based fluid dynamics particle overlapping is tolerable, in mesh deformation vertex overlapping would result in unacceptable visual artifacts. Therefore, the movement of vertices must be systematic and restricted. I will use three new vertex categories (see Figure 6.3).

(1) *Flow True Vertices* are vertices defined inside the Combustion Ready Boundary that are granted free movement within the given space.

(2) *Fixed Point Vertices* are vertices located in the non-melting region of the mesh but carry an association with a polygon that contains Flow True Vertices.

(3) *Base Point Vertices* are vertices in charge of guiding the flow of the fluid.

6.3.3. Polygonal Melting Properties

Heat weakens the bond strength between adjacent molecules of burning objects. This weakening effect falls off with distance from the heat source. As a result, surface molecules move in the direction of stronger bonds in order to find stable equilibrium between the forces that act on them. This results in stress on the burning area of the object. Melek and Keyser [32] also note that due to multiple internal chemical reactions at various stages of the combustion process, material may change state from solid to liquid. Material under change may exhibit both fluid and solid characteristics. These are often referred to as *viscoelastic fluids*, which have the property that the material initially responds to strain elastically like a solid, but when subjected to increasingly large stresses it flows like a fluid (see Clavet, S. [12]). The melting stage takes place as sub-volume transformation (see Wei and Li [54]), in which the vertices perform free movement on limited space.

Since one of the prime effects of the melting process is to smear viscoelastic-fluid-like

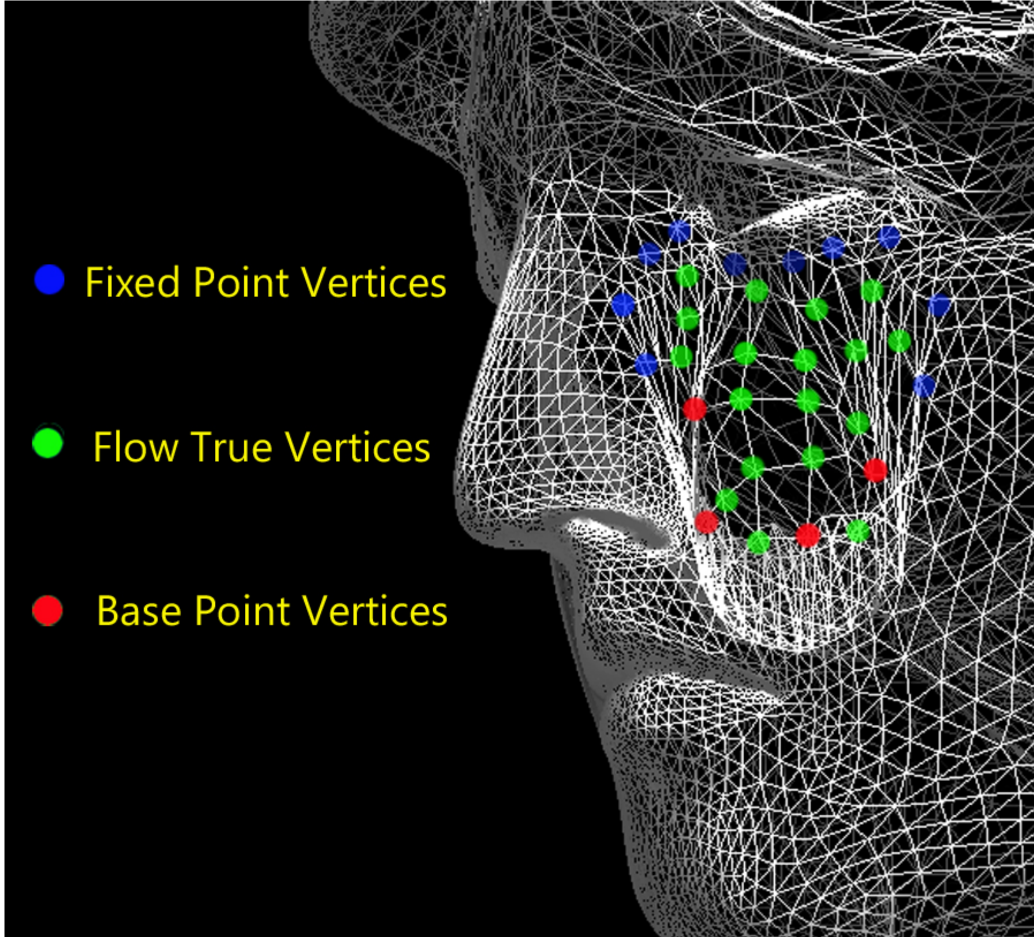

FIGURE 6.3. Categorized model vertices.

characteristics into the flow true vertices, my framework models the free movement, velocity and viscosity of the material. I achieve a fluid-like movement to a set of vertices by implementing a polygon folding procedure that I call *Fractional Folding Method*. The following approach contributes a fluid like behavior to the vertices in the targeted area by eliciting a chain reaction among the flow true vertices towards a stable placement. This method will result in polygons maintaining a certain range in sizes during the simulation.

The deformation coordinates and parameters of a single triangle is illustrate in Figure 6.4. To calculate the effect of melting on a triangle T, first identify the top, middle and bottom vertices (that is, sort them in order of Y coordinate, breaking ties at random). Let D be the given edge length of T. Compare D with the constant range $\omega Dist/\chi$ to $\omega Dist$, where ω is the viscosity of the material, $Dist$ is the average distance of a side of a triangle

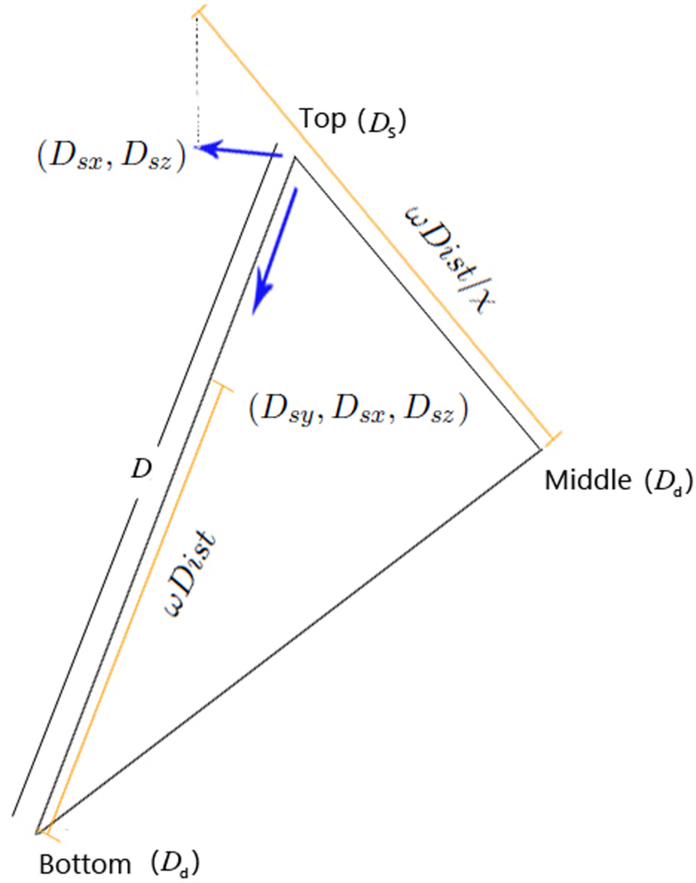

FIGURE 6.4. The deformation coordinates of a single triangle.

in original mesh (after subdivision takes place if required) and χ is a constant integer value that defines how small the smallest triangle should be.

Vertices are modified as follows:

$$\text{if}(D > \omega \text{Dist})$$

$$(D_{sx}, D_{sz}) = (D_{sx}, D_{sz}) -$$
$$((D_{sx}, D_{sz}) - (D_{dx}, D_{dz})) * \upsilon/(n+1)$$

$$\text{if}(D < \omega Dist/\chi)$$

$$(D_{sy}, D_{sx}, D_{sz}) = (D_{sy}, D_{sx}, D_{sz}) +$$
$$((D_{sy}, D_{sx}, D_{sz}) - (D_{dy}, D_{dx}, D_{dz})) * \upsilon/(n+1),$$

where υ is the velocity that determines how fast the vertices must move. If a flow true

vertex appears with n fixed point vertices in the same polygon, I factor the velocity of the movement by $n + 1$.

Since upward fluid motion would not make sense, when the contraction of the vertices ($D > \omega dis$) occurs, only x and z directional placement will be considered. After they reach their stable position, the motion of the vertices taking part in Fractional Folding will come to a halt until the next trigger occurs. The bodily flow of the melted substances needs a proper directional trigger to begin movement. I use the flow of base point vertices for this purpose. The table 6.1 contains the summary of the constant attribute set.

Attribute	**Values**	**Description**
$Dist$	$float$	Average distance
ω	$0 < \omega < 1$	Viscosity
χ	$int(1 - 10)$	Adjustment factor
υ	$0 < \upsilon < 1$	Velocity

TABLE 6.1. Constant attribute set.

6.3.4. Base Point Movement

I identify base point vertices by adapting the *Block Sampling Method* from chapter 3. This method divides the object into uniform blocks and treats each block as a single unit, propagating changes to neighboring blocks. I choose one base point vertex per block and keep them in an inactive state. They become active when they are reached by the combustion ready boundary. Since flow of the viscous fluid must pass over the solid surface of the object, I perform a collision check with the points in surrounding bounding boxes. In addition, usually fluid flows towards the bottom of the object due to gravitational influence. Here, first find the neighboring block that contains the lowest base point vertex, then, transform coordinates towards the lowest point. The transformation of the base point vertex is given by:

$(x, y, z) = (x, y, z - x_d, y_d, z_d)\omega + (x_d, y_d, z_d),$

where the (x_d, y_d, z_d) are the destination coordinates and ω is the viscosity.

A block absent one of its left, right, front or back neighbors is considered to be an *edge*

block. Whenever, an edge block has no bottom neighbor, Base Point Vertices fall under gravity:

$Y = Y - \omega\vec{g}/(Y_{crb} - Y)$,

where $\vec{g}$ is the gravity vector and Y_{crb} is the Y distance between the combustion ready boundary and Y. This will provide a deceleration when the vertices move away from the flames. Similarly, the rest of the Flow True Vertices will follow the Base Points.

6.4. Structural Deformation

In melting material the structural changes largely depend on the weight that is supported by the melting area. When combustion starts from the top of an area I assume that there is no weight supported and the structural changes can be ignored. When combustion starts from the middle or bottom, the weight effect causes a significant shape change. I have modeled this by applying the following calculations to the area outside of the combustion ready boundary and above the ignition point. I use the block sampling method described in chapter 3. I construct a bounding box around the object, and then decompose it into a grid of smaller axially aligned bounding boxes which I call *blocks*. Define the *weight* of a block to be the number of vertices inside it. I keep track of the orientation of each block as a triple of Euler angles. Weights of the blocks are decided according to the number of vertices inside it. Empty blocks of weight zero are discarded.

Although a change to one block may affect all of the blocks in the model, to reduce the computational load I apply changes to only immediate neighboring blocks, and rely on subsequent iterations to propagate the effects further. The change of the weight in each block results in a slight rotation of the box around its midpoint. The direction of the rotation will be determined by the placement of the displaced vertex compared to the midpoint of the box.

Stability will change due to the rotation of the immediate neighboring boxes. The change in roll angle R (pitch and yaw are similar) for a block is:

$R = \gamma\rho\pi/NM$,

where γ is a scaling factor chosen by the designer, ρ is a measure of the material density of the model in that block, N is the number of vertices in the block, and M is the current number of nonempty neighboring blocks.

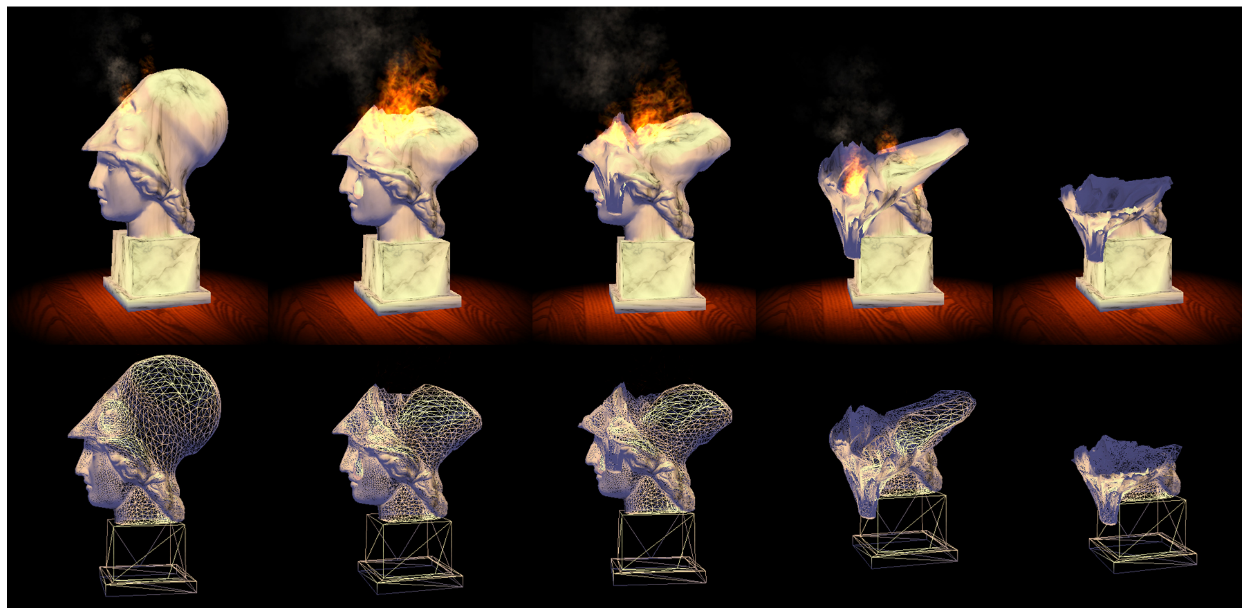

FIGURE 6.5. Melting sequence of a wax form solid model.

6.5. Results

I have described a method for the real-time melting of a wax type solid model during combustion by procedurally generated fire, extending my previous work described in preceding chapters. I was able to successfully perform this simulation on models of various mesh resolution and topology on less than cutting-edge hardware. My algorithm was implemented in CUDA on relatively modest hardware; An Intel®Core™2 Duo CPU P8400 @ 2.26GHz processor with an NVidia GeForce 9800 GTS graphics card. I was able to maintain 60fps frame rate up to 40k triangle model with balanced settings (quality vs. performance) in the graphic card. This performance will of course be much better on the current generation of graphics hardware, and thus able to run in parallel with other rendering tasks and game-related computation. I have used approximately 2000 fire particles and 500 smoke particles to demonstrate the visual effects.

CHAPTER 7

CONCLUSION

In this dissertation, I have proposed a method of real-time rendering of burning objects in video games. I have focused on deforming a polygonal model due to combustion while generating fire. I believe this is the first known method that could be used in gaming as modeling the deformation of a burning object. My work presents an extensive contribution to bring up the gaming environments one step closer to the real world phenomenon. The summary of my introduced framework generalize in the following section.

7.1. Summary

My research aimed specifically on decomposition and deformation of burning shell models and burning solids. In chapter 3, I have laid down the foundation and demonstrated the ability to realistically burn a high-polygon count shell model in real time on a relatively slow GPU. Chapter 4 extended this work to models with a very low polygon count by judicious use of procedural triangulation in the areas that are on fire, and demonstrated the ability to realistically burn on the same GPU. This approach also lent itself easily to dynamic level of detail (LOD) rendering, which is an important method for reducing the polygon count in games. In chapter 5, I have extended this work from shell models to solid objects by procedurally filling in the exposed interior of a burning object. I have described a method for the real-time melting of a wax type solid model during combustion by procedurally generated fire in chapter 6.

My findings and framework covers the basic theme of decomposition and deforming of a burning model. This introduced model can be improved further and there are open avenues for future research. For example, this framework works around single source heat boundary. Investigate a better approximation to heat boundary expansion with multiple heat sources still at large.

7.2. Hypothesis

My hypothesis stated that one can simulate different aspects of deformation and consumption of virtual objects by procedural fire and achieve visually plausible results at interactive rates on current hardware.

I have concluded my hypothesis as follows.

(1) I have successfully simulated deformation and consumption of burning objects.

(2) I was able to manipulate procedural fire distribute along the heat boundary using block sampling strategies.

(3) I was able to simulate objects with different aspects such as high polygonal models, low polygonal models, shell models and solid models. Furthermore, I was able to successfully address melting physiognomies of the burning objects.

(4) Almost all the deformation Figures included in this dissertation generated by my simulation framework. I believe these are adequate enough to lure the visually plausible results.

(5) I have designed and implemented my simulations for the video games. I have gain more than required interactive level frame rates in relatively modest GPUs.

APPENDIX A

BASIC DATA STRUCTURES

Here, we have laid out the basic data structures that have been used in our implementation and the useful code snippets castoff in OpenGL and CUDA interpolation.

```
#include <vector_types.h>  //Contains new data formats ex: float3, float4

typedef struct                /* Data structure for vertices */
{
   float4 CD;                 // Codinates
   float3 NM;                 // Normal coordinates
   float3 OLDNM;              // Keep record of original normal
   float4 OLDCD;              // Keep record of original cood
   int Cind;                  // Close point index
   int BID;                   // To hold the block number
   int TPC;                   // Structural deformation rotation controler
   int MinIdx;                // Keep track of Min point of associat triangle
} verticies;

typedef struct                /* Data structure for GPU Inds */
{
   int curr;                  // First vertex of the face
   int next;                  // Second vertex
   int last;                  // Last vertex
} GPUinds;

verticies *VertexGPU;         // Storage for vertices in GPU
GPUinds *IndsGPU;             // Store indecies in GPU
float4 *VertexData;           // Storage for vertices
```

```
                                                        // VBO Extension Function Pointers
PFNGLGENBUFFERSARBPROC glewGenBuffersARB = NULL;        // VBO Name Generation Procedure
PFNGLBINDBUFFERARBPROC glewBindBufferARB = NULL;        // VBO Bind Procedure
PFNGLBUFFERDATAARBPROC glewBufferDataARB = NULL;        // VBO Data Loading Procedure
PFNGLDELETEBUFFERSARBPROC glewDeleteBuffersARB = NULL;  // VBO Deletion Procedure

*************************************************************

                                                        // Create VBO
int bufferSize;                                         // Initialize Buffer size

glGenBuffersARB(1, vbo);                                // Initialize genaration of VBO
glBindBufferARB(GL_ARRAY_BUFFER_ARB, *vbo);             // Bind VBO to GL_Array buffer method
glBufferDataARB(GL_ARRAY_BUFFER_ARB,size,0,GL_DYNAMIC_DRAW);
                                                        // Copy vertex data arrays
glBufferSubDataARB(GL_ARRAY_BUFFER_ARB,0,(Number_of_vertices)
        *(sizeof(*VertexData)),VertexData);
                                                        // Copy vertex color arrays
glBufferSubDataARB(GL_ARRAY_BUFFER_ARB,(Number_of_vertices)
        *(sizeof(*VertexData)),(Number_of_vertices)
        *sizeof(*color_array),color_array);
glBufferSubDataARB(GL_ARRAY_BUFFER_ARB, (Number_of_vertices)// Copy vertex normal arrays
        *(sizeof(*VertexData))+(Numberofvertices)
        *sizeof(*color_array),(Number_of_faces)
        *sizeof(*NORMS),NORMS);
                                                        // Assign initial Buffer size
glGetBufferParameterivARB(GL_ARRAY_BUFFER_ARB, GL_BUFFER_SIZE_ARB, &bufferSize);

glBindBuffer(GL_ARRAY_BUFFER_ARB,0);                    // Set buffer set to start

                                                        // Register buffer object with CUDA
cudaGLRegisterBufferObject(*vbo);

*************************************************************
                                                        // Sample of GPU Index assignment
int idx = blockIdx.x * blockDim.x + threadIdx.x;        // General block thread distribution

*************************************************************

cudaGLMapBufferObject((void**)&dptr, vbo);              // Map OpenGL buffer object for CUDA

dim3 dimBlock(BLOCK_SIZE);                              // Invoke kernel
dim3 dimGrid(Numberofvertices*3/ dimBlock.x);           // Parrallel distribution processing
kernel<<<dimGrid, dimBlock>>> (dptr,--- variables to pass ---);  // Invoking Kernal

cudaGLUnmapBufferObject(vbo);                           // Unmap buffer object
```

```
unsigned int Size = ((Numberofvertices)*sizeof(*VertexData))             // Assign size of the serial data array
                  + ((Numberofvertices)*sizeof(*color_array))
                  + ((Number_of_faces)*sizeof(*NORMS));
                                                                          // Get Pointers To The GL Functions
glGenBuffersARB     = (PFNGLGENBUFFERSARBPROC)wglGetProcAddress("glGenBuffersARB");
glBindBufferARB     = (PFNGLBINDBUFFERARBPROC)wglGetProcAddress("glBindBufferARB");
glBufferDataARB     = (PFNGLBUFFERDATAARBPROC)wglGetProcAddress("glBufferDataARB");
glBufferSubDataARB = (PFNGLBUFFERSUBDATAARBPROC)wglGetProcAddress("glBufferSubDataARB");
glDeleteBuffersARB  = (PFNGLDELETEBUFFERSARBPROC)wglGetProcAddress("glDeleteBuffersARB");
glMapBufferARB      = (PFNGLMAPBUFFERARBPROC)wglGetProcAddress("glMapBufferARB");
glUnmapBufferARB =  (PFNGLUNMAPBUFFERARBPROC)wglGetProcAddress("glUnmapBufferARB");
glDeleteBuffersARB =  (PFNGLDELETEBUFFERSARBPROC) wglGetProcAddress("glDeleteBuffersARB");

************************************************************

                                                                          // CPU Display call
glEnableClientState(GL_NORMAL_ARRAY);                                     // Enable Normal Array
glEnableClientState(GL_COLOR_ARRAY);                                      // Enable Color Array
glEnableClientState(GL_VERTEX_ARRAY);                                     // Enable Vertx Array

glNormalPointer(GL_FLOAT,4*sizeof(float),                                 // Assign Normal Pointer
          (void *)((Numberofvertices)*sizeof(*VertexData)
          +(Numberofvertices)*sizeof(*color_array)));

glColorPointer(4,GL_FLOAT,0,                                              // Assign Normal Pointer
          (void *)((Numberofvertices)*sizeof(*VertexData)));
glVertexPointer(4,GL_FLOAT,0,0);                                          // VBO vertex pointer

for(int i = 0; i < Number_of_faces-2; i++)                                // Drawing process
{
  glBegin(GL_TRIANGLES);
     {
      glArrayElement(Indices[i]);                                         // Draw vertex element 1
      glArrayElement(Indices[i+1]);                                       // Draw vertex element 2
      glArrayElement(Indices[i+2]);                                       // Draw vertex element 3
      }
   glEnd();

   i=i+2;
 }

glDisableClientState(GL_COLOR_ARRAY);                                     // Close VBO Color Pointers
glDisableClientState(GL_NORMAL_ARRAY);                                    // Close VBO Normal Pointers
glDisableClientState(GL_VERTEX_ARRAY);                                    // Close VBO Vertex Pointers

glBindBufferARB(GL_ARRAY_BUFFER_ARB, 0);                                  // VBO Reset 0
```

APPENDIX B

GPU CODE STRUCTURES

In this section we list the code snippets for GPU side coding which include GPU Normal calculation. This structure of implementation can be used to demonstrate color, texture changes as well as midpoint rotation of the deforming model.

```
/**************************************************************************************
Function to calculate Normals on GPU
**************************************************************************************/
__device__ float3 GetNormal(float4 a, float4 b, float4 c)
{
	float NormalisationFactor, CombinedSquares;	//Parameters using to Normalize

	float3 x;
	x.x= b.x - a.x;					// Distance between vecter x components
	x.y= b.y - a.y;					// Distance between vecter y components
	x.z= b.z - a.z;					// Distance between vecter z components

	float3 y;
	y.x= c.x - a.x;					// Distance between vecter x components
	y.y= c.y - a.y;					// Distance between vecter y components
	y.z= c.z - a.z;					// Distance between vecter y components
							// Dot Product
	float3 n= make_float3(-(y.y*x.z- y.z*x.y), -(y.z*x.x- y.x*x.z), -(y.x*x.y - y.y*x.x));
							// Collect normals for normalized factor
	CombinedSquares = (n.x * n.x) + (n.y * n.y) + (n.z * n.z);
```

```
    NormalisationFactor = sqrt(CombinedSquares);    // Find normalized factor

    n.x /= NormalisationFactor;                     // Normalize final x component
    n.y /= NormalisationFactor;                     // Normalize final x component
    n.z /= NormalisationFactor;                     // Normalize final x component

    return n;
}

__global__  void kernel(--- Vriable data passing---)        // Main Kernel function
{

    int idx = blockIdx.x * blockDim.x + threadIdx.x;    // General block thread distribution

    if(s_data[inds[idx].curr].Chng)                     // If change in current vertex placement
    {
                                                        // Pass three verticies
    s_data[inds[idx].curr].NM = GetNormal(pos[inds[idx].curr], pos[inds[idx].next], pos[inds[idx].last]);
    s_data[inds[idx].curr].Chng= false;                 // Reset change flag

    }  // End checking if

    if(s_data[inds[idx].next].Chng)                     //  If change in next vertex placement
    {
                                                        // Pass three verticies
    s_data[inds[idx].next].NM = GetNormal(pos[inds[idx].curr], pos[inds[idx].next], pos[inds[idx].last]);
    s_data[inds[idx].next].Chng= false;                 // Reset change flag

    }  // End checking if

    if(s_data[inds[idx].last].Chng)                     // If change in third vertex placement
    {
                                                        // Pass three verticies
    s_data[inds[idx].last].NM = GetNormal(pos[inds[idx].curr], pos[inds[idx].next], pos[inds[idx].last]);
    s_data[inds[idx].last].Chng= false;                 // Reset change flag

    }  // End checking if
}
```

APPENDIX C

CUDA GPU IMPLEMENTATION

In this section, we listed a complete GPU side code sample that illustrates how to apply fluid like behavior to the polygonal model. Studying this may help understanding the complete implementation strategy behind our framework. In addition this section exemplifies how to use __device__ memory parameters and how to manipulate functions in GPU side coding using CUDA.

Note: Before executing kernel call to this code segment user must allocate GPU memory for the passing data structures (ex:for verticies *s_data and GPUinds *inds) using cudaMalloc().

```
/*****************************************************************************************************
Function to calculate Distance on GPU
*****************************************************************************************************/
__device__ float Distnce(float4 A, float4 B)
{
   return(sqrt(pow(A.x-B.x,2)+pow(A.y-B.y,2)+pow(A.z-B.z,2))); // Return distance
}
```

```
/*****************************************************************************************************
Function to Liquidise : keep vertex distance in certian distance.
keep moving like fluid
*****************************************************************************************************/
__device__ float4 Liqu(float4 A, float4 B, float visco, bool big)
{
   dis_max = 0.9;                                          // Max distance can be choreagraph
   dis_min = 0.4;                                          // Min distance can be choreagraph

   if(big)                                                 // If A and B larg in distance
   {                                                       // Compare distance with Max
      if(sqrt(pow(A.x-B.x,2)+pow(A.y-B.y,2)+pow(A.z-B.z,2))>dis_max)
      {
          A.x -= (A.x - B.x)*dis_min/visco;                //keep x distance intact of the polygon
          A.z -= (A.z - B.z)*dis_max/visco;                //keep z distance intact of the polygon
          A.y -= (A.y - B.y)*dis_max/visco;                //keep y distance intact of the polygon
      }
   }
   else
   {                                                       // Compare distance with Min
      if(sqrt(pow(A.x-B.x,2)+pow(A.y-B.y,2)+pow(A.z-B.z,2))<dis_min)
      {
         A.x += (A.x - B.x)*dis_min/visco;                 // keep x distance intact of the polygon
         A.z += (A.z - B.z)*dis_min/visco;                 // keep z distance intact of the polygon
         A.y += (A.y - B.y)*dis_max/visco;                 // keep z distance intact of the polygon
      }
   }     return A;
}
/*****************************************************************************************
//! Simple kernel to modify vertex positions for defomation
//! @param data  data in global memory
*****************************************************************************************/
__device__ float visco=300;          // Viscocity stord in the Device Memory (choreagraph value 300)

__global__  void kernel(float4* pos, float boundry, float4 start_point,
                 verticies *s_data, int *flags, float* PARA, float4 PTset, GPUinds *inds)
{

   // pos : CUDA data pointer allocate by mapping CPU to GPU
   // boundry: Heat Boundary expansion value. contains radial incriment of Delta(r)/Delta(t)
   // start_point: Location of the ignition point
   // s_data: Copy of Vertex data coordinates keep in GPU side
   // flags: size of the data pointers (length of #verticies) and boolean flags (stop burning/start burning)
   // PARA: usefull parameters used for changing of material indexes etc.
```

```
float disC    = 0;                          // Storing distance information
float MaxA = 0;                             // Keep track of the top vertex of the polygon
float MinB   = 0;                           // Keep track of the bottom vertex of the polygon

int minA_idx  = 0;                          // Index of the bootom vertex of a given polygon
int maxA_idx = 0;                           // Index of the top vertex of a given polygon
int midA_idx  = 0;                          // Index of the middle vertex of a given polygon

                                            // General block thread distribution
int idx = blockIdx.x * blockDim.x + threadIdx.x;

if(flags[2]==1)                             // Burn flag boolean value pass by CPU to stop/start
{
  if(idx< flags[1])                         // Keep mark of number of vertices.
  {                                         // Distance from ignition point to current vertex
    disC = Distnce(start_point, pos[inds[idx].curr]);
                                            // Find bottom vertex
    MinB = fmin(s_data[inds[idx].curr].CD.y, fmin(s_data[inds[idx].next].CD.y, s_data[inds[idx].last].CD.y));
                                            // Find top vertex
    MaxA = fmax(s_data[inds[idx].curr].CD.y, fmax(s_data[inds[idx].next].CD.y, s_data[inds[idx].last].CD.y));

    if(s_data[inds[idx].curr].CD.y == MaxA)          // If top point of the polygon
      maxA_idx=inds[idx].curr;                       // current is the top point
    else if(s_data[inds[idx].next].CD.y == MaxA)     // if top point of the polygon
      maxA_idx=inds[idx].next;                       // next is the top point
    else if(s_data[inds[idx].last].CD.y == MaxA)     // if top point of the polygon
      maxA_idx=inds[idx].last;                       // last is the top point

    if(s_data[inds[idx].curr].CD.y == MinB)          // if bottom point of the polygon
      minA_idx=inds[idx].curr;                       // current is the bottom point
    else if(s_data[inds[idx].next].CD.y == MinB)     // if bottom point of the polygon
      minA_idx=inds[idx].next;                       // next is the bottom point
    else if(s_data[inds[idx].last].CD.y == MinB)     // if bottom point of the polygon
      minA_idx=inds[idx].last;                       // last is the bottom point

                                            // If mid point of the polygon
    if(s_data[inds[idx].curr].CD.y != MinB && s_data[inds[idx].curr].CD.y != MaxA)
      midA_idx=inds[idx].curr;              // current is the mid point
                                            // If mid point of the polygon
    else if(s_data[inds[idx].next].CD.y != MinB && s_data[inds[idx].next].CD.y != MaxA)
      midA_idx=inds[idx].next;              // next is the mid point
                                            // If mid point of the polygon
    else if(s_data[inds[idx].last].CD.y != MinB && s_data[inds[idx].last].CD.y != MaxA)
      midA_idx=inds[idx].last;              // last is the mid point
```

```
                                                    // Keep track of the lowest pt
        if(s_data[maxA_idx].MinIdx==-1)s_data[maxA_idx].MinIdx=minA_idx;
        else if(s_data[s_data[maxA_idx].MinIdx].CD.y > s_data[minA_idx].CD.y )s_data[maxA_idx].MinIdx=minA_idx;
                                                    // Keep track of the lowest pt
        if(s_data[midA_idx].MinIdx==-1)s_data[maxA_idx].MinIdx=minA_idx;
        else if(s_data[s_data[midA_idx].MinIdx].CD.y > s_data[minA_idx].CD.y )s_data[maxA_idx].MinIdx=minA_idx;

    /**************************************************************************************************/

      if(boundry > disC-1.0 )                       // If inside the closing boundary limit
      {                                             // Call for fluid setting true = big; flase = small
        s_data[maxA_idx].CD= Liqu(s_data[maxA_idx].CD, s_data[minA_idx].CD, visco, true);
        s_data[maxA_idx].CD= Liqu(s_data[maxA_idx].CD, s_data[minA_idx].CD, visco, false);
        s_data[maxA_idx].CD= Liqu(s_data[maxA_idx].CD, s_data[midA_idx].CD, visco, true);
        s_data[maxA_idx].CD= Liqu(s_data[maxA_idx].CD, s_data[midA_idx].CD, visco, false);
        s_data[midA_idx].CD= Liqu(s_data[midA_idx].CD, s_data[minA_idx].CD, visco, true);
        s_data[midA_idx].CD= Liqu(s_data[midA_idx].CD, s_data[minA_idx].CD, visco, false);
      }

    /**************************************************************************************************/

      if(boundry> disC)                             // Inside the boudary
       {                                            // Gravitational flow effect
        s_data[maxA_idx].CD.y  -=  abs(s_data[maxA_idx].CD.y-s_data[minA_idx].CD.y)/visco;
      }
     }                                              // End if boudary
    }                                               // End if burn flags

    __syncthreads();                                // Syncronize threads

     pos[idx + 2*flags[0]] = make_float4(           // Update Normal values
                          s_data[idx].NM.x,
                          s_data[idx].NM.y,
                          s_data[idx].NM.z,
                          1.0);
                                                    // Update data values
      pos[idx] = make_float4(s_data[idx].CD.x,s_data[idx].CD.y,
                    s_data[idx].CD.z,s_data[idx].scale);
}
```

BIBLIOGRAPHY

[1] P. Alliez, G. Ucelli, C. Gotsman, and M. Attene. Recent advances in remeshing of surfaces. *Shape Analysis and Structuring*, pages 53–82, 2008. 33

[2] Dhanyu Amarasinghe and Ian Parberry. Fast, believable real-time rendering of burning low-polygon objects in video games. In *Proc. 6th Internat. North American Conf. on Intelligent Games and Simulation (GAMEON-NA)*, pages 21–26. EUROSIS, 2011. 44

[3] Dhanyu Amarasinghe and Ian Parberry. Towards fast, believable real-time rendering of burning objects in video games. In *Proc. 6th Annual Internat. Conf. on the Foundations of Digital Games*, pages 256–258, 2011. 44

[4] Ronan Amorim, Gundolf Haase, Manfred Liebmann, and R Weber dos Santos. Comparing cuda and opengl implementations for a jacobi iteration. In *High Performance Computing & Simulation, 2009. HPCS'09. International Conference on*, pages 22–32. IEEE, 2009. 6

[5] Joshua A Anderson, Chris D Lorenz, and Alex Travesset. General purpose molecular dynamics simulations fully implemented on graphics processing units. *Journal of Computational Physics*, 227:5342–5359, 2008. 6

[6] H. Borouchaki, P. Laug, A. Cherouat, and K. Saanouni. Adaptive remeshing in large plastic strain with damage. *International Journal for Numerical Methods in Engineering*, 63(1):1–36, 2005. 33

[7] T. Boubekeur and C. Schlick. Generic mesh refinement on GPU. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pages 99–104. ACM, 2005. 33, 34, 36

[8] Rodney Bryant, Carole Womeldorf, Erik Johnsson, and Thomas Ohlemiller. Radiative heat flux measurement uncertainty. *Fire and materials*, 27(5):209–222, 2003. 15

[9] Luc Buatois, Guillaume Caumon, and Bruno Lévy. Concurrent number cruncher: An efficient sparse linear solver on the gpu. *High Performance Computing and Communi-*

cations, pages 358–371, 2007. 7

[10] Ian Buck, Tim Foley, Daniel Horn, Jeremy Sugerman, Kayvon Fatahalian, Mike Houston, and Pat Hanrahan. Brook for gpus: stream computing on graphics hardware. In *ACM Transactions on Graphics (TOG)*, volume 23, pages 777–786. ACM, 2004. 7

[11] M. Carlson, P.J. Mucha, R.B. Van Horn III, and G. Turk. Melting and flowing. In *Proc. 2002 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 167–174. ACM, 2002. 54

[12] S. Clavet, P. Beaudoin, and P. Poulin. Particle-based viscoelastic fluid simulation. In *Proc. 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 219–228. ACM, 2005. 54, 57

[13] Glenn A Elliott, James H Anderson, et al. Globally scheduled real-time multiprocessor systems with gpus. In *Proceedings of the 18th International Conference on Real-Time and Network Systems*, pages 197–206, 2010. 7

[14] F. Fan and F.F. Cheng. GPU supported patch-based tessellation for dual subdivision. In *2009 Sixth International Conference on Computer Graphics, Imaging and Visualization*, pages 5–10. IEEE, 2009. 33

[15] M.S. Floater and K. Hormann. Surface parameterization: A tutorial and survey. *Advances in Multiresolution for Geometric Modelling*, pages 157–186, 2005. 32

[16] N. Foster and R. Fedkiw. Practical animation of liquids. In *Proc. 28th Annual Conference on Computer graphics and Interactive Techniques*, pages 23–30. ACM, 2001. 54

[17] Alfred R. Fuller, Hari Krishnan, Karim Mahrous, Bernd Hamann, and Kenneth I. Joy. Real-time procedural volumetric fire. In *I3D '07: Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games*, pages 175–180, New York, NY, USA, 2007. ACM. 14

[18] T.G. Goktekin, A.W. Bargteil, and J.F. O'Brien. A method for animating viscoelastic fluids. In *ACM Transactions on Graphics*, volume 23, pages 463–468. ACM, 2004. 54

[19] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: high

performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 325–336. ACM, 2006. 7

[20] X. Guo, X. Li, Y. Bao, X. Gu, and H. Qin. Meshless thin-shell simulation based on global conformal parameterization. *IEEE Transactions on Visualization and Computer Graphics*, pages 375–385, 2006. 33

[21] L. He, S. Schaefer, and K. Hormann. Parameterizing subdivision surfaces. *ACM Transactions on Graphics*, 29(4):1–6, 2010. 33

[22] K. Hormann, U. Labsik, and G. Greiner. Remeshing triangulated surfaces with optimal parameterizations. *Computer-Aided Design*, 33(11):779–788, 2001. 32

[23] W.M. Hsu, J.F. Hughes, and H. Kaufman. Direct manipulation of free-form deformations. In *Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, pages 177–184. ACM, 1992. 14

[24] K. Iwasaki, H. Uchida, Y. Dobashi, and T. Nishita. Fast particle-based visual simulation of ice melting. In *Computer Graphics Forum*, volume 29, pages 2215–2223. Wiley Online Library, 2010. 54

[25] M.W. Jones. Melting objects. *The journal of WSCG*, 11(2), 2003. 55

[26] R. Keiser, B. Adams, D. Gasser, P. Bazzi, P. Dutré, and M. Gross. A unified lagrangian approach to solid-fluid animation. In *Proc. Eurographics/IEEE VGTC Symposium on Point-Based Graphics*, pages 125–148. IEEE, 2005. 54

[27] D. Kovacs, J. Mitchell, S. Drone, and D. Zorin. Real-time creased approximate subdivision surfaces. In *Proceedings of the 2009 Symposium on Interactive 3D Graphics and Games*, pages 155–160. ACM, 2009. 33

[28] Victor W Lee, Changkyu Kim, Jatin Chhugani, Michael Deisher, Daehyun Kim, Anthony D Nguyen, Nadathur Satish, Mikhail Smelyanskiy, Srinivas Chennupaty, Per Hammarlund, et al. Debunking the 100x gpu vs. cpu myth: an evaluation of throughput computing on cpu and gpu. In *ACM SIGARCH Computer Architecture News*, volume 38, pages 451–460. ACM, 2010. 7

[29] MJ Lighthill. Contributions to the theory of heat transfer through a laminar boundary layer. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, pages 359–377, 1950. 15

[30] F. Losasso, G. Irving, E. Guendelman, and R. Fedkiw. Melting and burning solids into liquids and gases. *IEEE Transactions on Visualization and Computer Graphics*, 12(3):343–352, 2006. 54

[31] Chi-Keung Luk, Sunpyo Hong, and Hyesoon Kim. Qilin: exploiting parallelism on heterogeneous multiprocessors with adaptive mapping. In *Microarchitecture, 2009. MICRO-42. 42nd Annual IEEE/ACM International Symposium on*, pages 45–55. IEEE, 2009. 7

[32] Zeki Melek and John Keyser. An interactive simulation framework for burning objects. Technical Report 2005-03-1, Department of Computer Science, Texas A&M University, 2005. 14, 44, 45, 48, 55, 57

[33] Zeki Melek and John Keyser. Driving object deformations from internal physical processes. In *Proceedings of the 2007 ACM Symposium on Solid and Physical Modeling*, pages 51–59, New York, NY, USA, 2007. ACM. 14, 44, 55

[34] Sameer Moidu, James Kuffner, and Kiran S. Bhat. Animating the combustion of deformable materials. In *ACM SIGGRAPH 2004 Posters*, page 90, New York, NY, USA, 2004. ACM. 14

[35] M. Müller, R. Keiser, A. Nealen, M. Pauly, M. Gross, and M. Alexa. Point based animation of elastic, plastic and melting objects. In *Proc. 2004 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pages 141–151. Eurographics Association, 2004. 54

[36] Matthias Müller, Leonard McMillan, Julie Dorsey, and Robert Jagnow. Real-time simulation of deformation and fracture of stiff materials. *Computer Animation and Simulation 2001*, pages 113–124, 2001. 14

[37] A. Nealen, M. Müller, R. Keiser, E. Boxerman, and M. Carlson. Physically based deformable models in computer graphics. *Computer Graphics Forum*, 25(4):809–836,

2006. 44, 55

[38] Duc Quang Nguyen, Ronald Fedkiw, and Henrik Wann Jensen. Physically based modeling and animation of fire. In *Proceedings of the 29th Annual Conference on Computer Graphics and Interactive Techniques*, pages 721–728, New York, NY, USA, 2002. ACM. 14, 44, 55

[39] CUDA NVIDIA. C programming guide: `http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc.` *CUDA_C_Programming_Guide.pdf*, 2013. 7, 11, 12

[40] Gabriel Peyré and Laurent D Cohen. Geodesic remeshing using front propagation. *International Journal of Computer Vision*, 69(1):145–156, 2006. 33

[41] N. Rasmussen, D.Q. Nguyen, W. Geiger, and R. Fedkiw. Smoke simulation for large scale phenomena. *ACM Transactions on Graphics*, 22(3):703–707, 2003. 44, 55

[42] Diego Alejandro Rivera-Polanco. Collective communication and barrier synchronization on nvidia cuda gpu. 2009. 9

[43] Shane Ryoo, Christopher I Rodrigues, Sara S Baghsorkhi, Sam S Stone, David B Kirk, and Wen-mei W Hwu. Optimization principles and application performance evaluation of a multithreaded gpu using cuda. In *Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 73–82. ACM, 2008. 7

[44] Nadathur Satish, Mark Harris, and Michael Garland. Designing efficient sorting algorithms for manycore gpus. In *Parallel & Distributed Processing, 2009. IPDPS 2009. IEEE International Symposium on*, pages 1–10. IEEE, 2009. 7

[45] H.Y. Schive, Y.C. Tsai, and T. Chiueh. Gamer: A graphic processing unit accelerated adaptive-mesh-refinement code for astrophysics. *The Astrophysical Journal Supplement Series*, 186:457, 2010. 33

[46] T.W. Sederberg and S.R. Parry. Free-form deformation of solid geometric models. *ACM SIGRAPH Computer Graphics*, 20(4):151–160, 1986. 14, 44, 55

[47] V. Settgast, K. Müller, C. Fünfzig, and D. Fellner. Adaptive tesselation of subdivision

surfaces. *Computers & Graphics*, 28(1):73–78, 2004. 33

[48] JC Simo and F. Armero. Geometrically non-linear enhanced strain mixed methods and the method of incompatible modes. *Internat. J. for Numerical Methods in Engineering*, 33(7):1413–1449, 1992. 44, 55

[49] EB Tadmor, R. Phillips, and M. Ortiz. Mixed atomistic and continuum models of deformation in solids. *Langmuir*, 12(19):4529–4534, 1996. 44, 55

[50] D. Terzopoulos, J. Platt, A. Barr, and K. Fleischer. Elastically deformable models. *ACM SIGGRAPH Computer Graphics Quarterly*, 21(4):205–214, 1987. 44, 55

[51] D. Terzopoulos, J. Platt, and K. Fleischer. Heating and melting deformable models. *The Journal of Visualization and Computer Animation*, 2(2):68–73, 1991. 54

[52] J.I. Toivanen. A Non-Linear Mesh Deformation Operator Applied to Shape Optimization. 2006. 14

[53] Vasily Volkov and James W Demmel. Benchmarking gpus to tune dense linear algebra. In *Proceedings of the 2008 ACM/IEEE conference on Supercomputing*, page 31. IEEE Press, 2008. 7

[54] X. Wei, W. Li, and A. Kaufman. Melting and flowing of viscous volumes. In *Proc. 16th International Conference on Computer Animation and Social Agents*, pages 54–59. IEEE, 2003. 54, 57

[55] X. Wei, W. Li, K. Mueller, and A. Kaufman. Simulating fire with texture splats. In *IEEE Visualization*, pages 227–234, 2002. 14

[56] M. Wicke, D. Ritchie, B.M. Klingner, S. Burke, J.R. Shewchuk, and J.F. O'Brien. Dynamic local remeshing for elastoplastic simulation. *ACM Transactions on Graphics*, 29(4):1–11, 2010. 32

[57] Ye Zhao, Xiaoming Wei, Zhe Fan, Arie Kaufman, and Hong Qin. Voxels on fire. *Visualization Conference, IEEE*, page 36, 2003. 14