VOLUMETRIC PARTICLE MODELING

A Dissertation

by

BRENT MICHAEL DINGLE

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

DOCTOR OF PHILOSOPHY

May 2007

Major Subject: Computer Science

VOLUMETRIC PARTICLE MODELING

A Dissertation

by

BRENT MICHAEL DINGLE

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

DOCTOR OF PHILOSOPHY

Approved by:

Chair of Committee,	John Keyser
Committee Members,	Donald House
	Frank Shipman
	Peter Stiller
Head of Department,	Valerie Taylor

May 2007

Major Subject: Computer Science

ABSTRACT

Volumetric Particle Modeling.

(May 2007)

Brent Michael Dingle, B.S., Bradley University;

M.S., Texas A&M University

Chair of Advisory Committee: Dr. John Keyser

This dissertation presents a robust method of modeling objects and forces for computer animation. Within this method objects and forces are represented as particles. As in most modeling systems, the movement of objects is driven by physically based forces. The usage of particles, however, allows more artistically motivated behavior to be achieved and also allows the modeling of heterogeneous objects and objects in different state phases: solid, liquid or gas. By using invisible particles to propagate forces through the modeling environment complex behavior is achieved through the interaction of relatively simple components. In sum, 'macroscopic' behavior emerges from 'microscopic' modeling.

We present a newly developed modeling framework expanding on related work. This framework allows objects and forces to be modeled using particle representations and provides the details on how objects are created, how they interact, and how they may be displayed. We present examples to demonstrate the viability and robustness of the developed method of modeling. They illustrate the breaking and fracturing of solids, the interaction of objects in different phase states, and the achievement of a reasonable balance between artistic and physically based behaviors.

ACKNOWLEDGMENTS

I would like to thank all those who feel they should be thanked, for whatever help they provided. While I am certain I will miss a few, these people include:

Daphne B., Kendra B., Michelle B., Tami D., Amy H., Cathy H., Vicki H., Sarah H., Traci H., Rain J., Elizabeth J., Kristine K., Nicole K., Jennifer K., Stephanie M., Patrice P., Randi R., Jennifer S., Nicole S., all the guys who wouldn't want their names mentioned anyway, such as Tracy C., Joey C., David E., Kelly F., Alan J., Jamie M., Jacob M., Doug R., Frank S., Jens W., Jeff W., Fletcher, Jonez, and all the others for reminding me and advising me…

Continued thanks to my wife, Stephanie and all my family.

Special thanks to my committee members: Donald House, Frank Shipman and Peter Stiller.

An extreme amount of thanks to my committee chair: John Keyser.

Final nods to all the online websites that keep computer science free and abundant in source code, images, models and imagination.

Smiles and laughs to the many science fiction and fantasy writers, producers, actors, cartoons and whatever other descriptions they claim. Science would be nothing without such inspiration and dreams.

But most of all, Thanks and Glory to God.

TABLE OF CONTENTS

Page

ABSTRACT ... iii

ACKNOWLEDGMENTS ... iv

TABLE OF CONTENTS ... v

LIST OF FIGURES ... vii

LIST OF TABLES ... xi

1. INTRODUCTION ... 1

1.1 Dissertation Statement ... 1
1.2 Accomplishments ... 1
1.3 Motivation ... 2
1.4 Application Areas ... 4
1.4.1 Animation and Game Development ... 4
1.4.2 Dynamic Object Reconstruction ... 5
1.4.3 Parallel Computing and GPUs ... 6
1.5 Challenges and Solutions ... 6
1.5.1 Object Representation ... 7
1.5.2 Object Creation ... 7
1.5.3 Engine Implementation ... 7
1.5.4 Object Display ... 8
1.5.5 Collision Processing ... 8

2. BACKGROUND ... 10

2.1 Particle Systems ... 10
2.1.1 Flocks and Herds ... 13
2.1.2 Growing Patterns ... 13
2.1.3 Surfaces and Textures ... 14
2.1.4 Solid Objects ... 14
2.1.5 Molecular Dynamics ... 15
2.2 Splatting Techniques ... 15
2.2.1 Splatting in General ... 16
2.2.2 Basic Splatting Algorithm ... 18
2.3 Previous Work on Fracture and Breaking ... 19
2.3.1 Spring Method ... 21
2.3.2 Finite Element Method ... 22

3. VOLUMETRIC PARTICLE OBJECTS (VOLPARS) ... 24

3.1 Definitions ... 24
3.2 Particle Bindings ... 24

Page

3.3 Particle Shapes 25
3.4 State Phases 26
3.5 Gas Particles 27
3.6 Liquid Particles 28
3.7 Solid Particles 29
3.7.1 Rigid Breakable Objects 30
3.7.2 Elastic Deformable Objects 31
3.7.3 Plastic Deformable Objects 32
3.8 Object Conversion 33
3.8.1 Grid Based Filling, Cubic Representation 34
3.8.2 Packing More Particles, Filled Representation 36
3.8.3 Packing Density 40
3.9 Grouping Particles 42
3.10 Space Partitioning 45

4. FORCE AND ACTION REPRESENTATION 48

4.1 Impact Forces, Impulse Divided 49
4.2 Damaging Connections 51
4.3 Wave Propagation 55
4.3.1 Blast Waves 55
4.3.2 Impact Waves 57
4.4 Action and Force Particles 58
4.4.1 Motivation from Wave Propagation 59
4.4.2 Definition Expansion 60
4.4.3 Basic Examples 61
4.4.4 Vortex Effects 64
4.4.5 Event Driven, Cross-Media Effects 66
4.4.6 Coupled Force Particle Systems 69
4.5 Gas Pressure 71

5. OBJECT DISPLAY 74

5.1 Billboarding 75
5.2 Surfels 75
5.2.1 Breakable Surface Identification 76
5.2.2 Calculating Surface Normals 77
5.2.2.1 Moving Least Squares 77
5.2.2.2 Outward Direction 80
5.2.3 Applying Surfels 82
5.3 Other Options 83

6. IMPLEMENTATION EXAMPLES AND RESULTS 84

6.1 Keyframing 85
6.1.1 Basic Method, Position to Position 86
6.1.2 Density to Density Keyframes 89
6.1.3 Boundary to Boundary Keyframes 92

Page

6.1.4 Plausibility 93
6.1.4.1 Plausibility Criteria 94
6.1.4.2 Distance Plausibility 95
6.1.4.3 Viability Plausibility 95
6.1.4.4 Velocity Plausibility 96
6.1.5 Potential Refinements 97
6.2 Basic Rigid Objects 98
6.3 Fracture and Breaking 100
6.3.1 Connection Strengths 100
6.3.2 Material Fracture Patterns 101
6.3.3 Material Fracture Patterns, DLA Grown 102
6.3.4 Scorelines 104
6.3.5 Fracture Patterns as Scorelines 105
6.3.6 Fracture Tip Propagation 106
6.3.7 Implementation Details 110
6.3.8 Visual Results 113
6.3.9 Results Summary 118
6.4 Dirt, Water, Mud 120
6.5 Melting 124

7. CONCLUSION 129

REFERENCES 130

APPENDIX A 141

APPENDIX B 165

APPENDIX C 174

APPENDIX D 176

APPENDIX E 179

APPENDIX F 181

VITA 184

LIST OF FIGURES

FIGURE		Page
1	Chess pawn renderings	8
2	Artifacts of splatting	17
3	Spring object model	19
4	Refining the tetrahedralization of objects	22
5	Generalized form of a Lennard-Jones force potential equation	25
6	Graph of LJ-potential with an extended rest interval	32
7	Stanford Bunny transformed from a mesh object to a volpar object	34
8	Inside/outside vertex marking using grid rays	35
9	Surface boxes as discovered for a 2D object	35
10	Particle neighbor relations	36
11	Bull mesh being filled with particles	37
12	2D result of filling a cubic representation	39
13	Packings of unit spheres for side length, s, of 4.05, 8.95, and 5.40 units	42
14	A demonstration of grouping internal particles	45
15	A particle in a basic grid	46
16	Grid cells overlap each other by the maximum diameter of all the particles	46
17	Perpendicular and parallel projection of impact force	52
18	Lack of particle motion damages the connection	53
19	Pressure front passing over two connected voxels	56
20	Pressure curve of a blast wave using a modified Friedlander equation	57
21	Visually odd cause and effect	59
22	Particles representing a conic wave front	60
23	Bubble of air generated by a force particle displacing water particles	61

FIGURE Page

24 Heat transfer from fire using action particles 62
25 Force particles cause ripples to form in a pool of water 63
26 Smoke particles where only a linear wind current is applied 64
27 Smoke particles influenced by a linear current and vortex force particles 65
28 Fan blowing confetti via wind force particles 66
29 Confetti blown by force particles in a circular pattern against a wall 67
30 Smoke, wind, fan and force particles example 68
31 A tornado modeled using a coupled force particle system 69
32 Wireframe, triangulated surface, particle spheres and surfel display of a chair 74
33 A solid object partially fractured 76
34 Two possible surface particle normals for a given particle 80
35 Surface normal for a particle at a narrow point 82
36 Two calculated forces acting on a particle 87
37 Motion by scaled force sum for i^{th} time step 87
38 A picture composed of particles dissolving under gravity 88
39 An imaging forming via keyframed forces 88
40 Motion based on center of mass 90
41 Dispersive external forces drive the particles too far apart 91
42 Adaptive subdivision corrects dispersion 91
43 A square morphing into a triangle 93
44 Block volpar objects colliding 98
45 Stanford bunnies collide 99
46 Surfel rendered Stanford bunny, bounces around 99
47 Generic glass-like material fracture pattern 105

FIGURE Page

48 Generic three and four prong material fracture patterns 106

49 Fracture tip propagation by application of a material fracture map 107

50 Overhead view and volpar view of fracture program 112

51 Simple scoreline fracture 114

52 A plate struck twice 114

53 A chess pawn scored, breaks into two 115

54 Glass-like fracture pattern, first impact 115

55 Glass-like fracture pattern, second impact 116

56 Four prong bunny fracture 116

57 Heterogeneous brick wall 117

58 Brick wall with a 3 prong material fracture pattern applied 117

59 A demolition ball hits the brick wall 118

60 Water mixes with dirt particles to form mud particles 121

61 Mud drying and cracking 122

62 Ice melting displayed using spheres and cubes 127

63 Raytraced ice cube melting 128

LIST OF TABLES

TABLE Page

1 Improved update process for a particle system 12

2 Splatting pseudo-code 18

3 Description of how state phases affect particle behavior 26

4 Pseudo-code for 2-pass filling-over method 40

5 Results of filling tests 41

6 Group frame rates 43

7 Description of the grouping algorithm 44

8 Description of the graph creation algorithm 44

9 Pseudo-code for constraining force particle location 71

10 Algorithm to calculate normals of surface particles 78

11 Structures used in implementing fracturing and breaking 110

12 Classes used in implementing fracturing and breaking 111

13 Keyboard interface options for the fracture and breaking program 112

1. INTRODUCTION

This introductory section presents the hypothesis of the dissertation, the motivation behind it, the relevant application areas, and the challenges faced while working on it. In sum, a general description of a modeling framework and what it is to achieve is presented. The details are left to later sections.

1.1 Dissertation Statement

It is the hypothesis of this dissertation that a modeling framework based strictly on a particle representation of objects, and forces, is feasible and robust. To prove this assertion, the design and details of such a framework must be presented.

The following pages contain a description of such a framework. This material includes the abstraction for representing the objects and the modeling engine. In later sections, examples of implementations are described. Throughout this the desired result is a combination of art and physics where the user, via modeling parameters, controls the realism of the model. By default the behavior of objects is only acceptable for animation and is not precise enough for engineering based simulation. While the framework provides options for increasing accuracy, leaving it low allows the overall speed of the modeling engine to increase and offers better blending of artistic with realistic effects.

1.2 Accomplishments

Fundamentally a particle based framework changes the way objects and forces are modeled. No longer are there objects, but compositions of particles. Because these compositions are derived from the same basic particles, the behavior of every object is governed by the same set of rules.

This dissertation follows the style of *Computer Graphics Forum*.

The rules are applied to the basic particle elements, not individual objects. This allows an engine to be designed with few special cases leading to a diverse number of applications, and offers significant gains in many aspects of modeling. These are shown in later sections.

To reiterate, ***the major accomplishment of this dissertation is the design and demonstration that a modeling framework based strictly on a particle representation of objects, and forces, is feasible and robust.*** The feasibility of the framework is shown by presenting a modeling abstraction that is based on particles, consistent within itself, and capable of modeling the motion and interaction of significantly complex objects. The robustness of the framework is proven using examples demonstrating the use of particles for object representation and force propagation. To accomplish this a variety of methods related to particle systems are modified, adapted and brought into a unifying paradigm. Among the notable accomplishments of this are:

- Methods to convert existing models into volumetric particle representations.
- Methods of displaying objects with topologically changing surfaces.
- Methods for fracturing and breaking objects with a stress memory.
- Application of short range, long range and collision forces to particle objects.
- Application of space partitioning to move and display particle objects.
- Application of graph theory to particle connectivity, object coherence, and fracture detection.
- Use of force particles to simulate temporary, event drive effects and force propagation.
- Development and application of scorelines and material fracture patterns.
- Use of keyframing and physically based techniques to artistically, yet realistically, guide particle systems into a given state.

1.3 Motivation

In 2002 the paper, "Algorithmic issues in modeling motion," was published in the Journal of ACM Computing Surveys [1]. In this paper four major issues within physical simulation are presented. The first challenge is in the combination of different state representations to solve interactions between objects in

different phases (solid, liquid, gas). The second challenge is to create a modeling system which can select appropriate motion models and algorithms for a given situation. The third is to design simulation algorithms capable of being implemented in parallel processing environments, such that the simulation scales with the number of processors. The fourth challenge is focused on animation environments with the challenge being to give more control of the simulation to the user while maintaining physical plausibility.

The modeling framework presented within this dissertation is based solely on representing objects with particles. This addresses the first challenge as it allows the interaction of objects in various phases. In addition it allows the modeling of heterogeneous substances and allows material properties to be represented. To an extent the second challenge is addressed as all interactions are based on particle interactions, so choices are limited. The third challenge is met as the use of particle representations has a natural capacity to be implemented in parallel environments [2], especially when using graphic processing units (GPUs) of current video cards. The fourth challenge is overcome in a variety of ways. One specific triumph is illustrated in the keyframing abilities of particle based modeling (section 6).

It should be understood that this dissertation is not created as a response to the paper. This dissertation began in 2001 with the only desire being to design and implement a modeling framework based on particles. However, as a particle framework addresses the requests presented in the 2002 paper, it is worth mentioning to illustrate need.

Motivation is based on the current shortfall of object representations. As the majority of objects are now modeled, to allow them to change form or break apart requires separate models of their altered forms to be created, maintained and displayed when appropriate. This requires special effects and transitions between forms to be hard coded or performed by hand for each case [3]. Besides being time consuming and a waste of memory, this tactic removes all aspects of physically based reality. To remedy this, a new representation and modeling engine is needed.

This dissertation presents such a modeling framework. To demonstrate this, in the examples, focus is given to physically based shape and state changing effects such as breaking, fracturing, melting, cracking, etc. User controllability of such physical effects is also demonstrated.

While such features are clearly useful in the multi-media industry, they also have use in other areas. This is shown in section 1.4. So, what is motivation is also potential use.

1.4 Application Areas

The target area of application for the results of this dissertation is in the entertainment industry. The techniques and methods are developed with the specific intent of animation. However, several other areas of application and research presented themselves during the development of the framework. These alternate areas of interest include dynamic object reconstruction, robotic motion planning, and parallel computing. A discussion of each is presented in the following sections.

1.4.1 Animation and Game Development

Game development is always a demanding area of performance and expectation. As computing power increases game players continue to want more from their games. In the last decade 2D and isometric games have been discarded in preference for 3D games. Sprites have been replaced by full 3D models. Planar movement has yielded to six degrees of freedom. Games are changing. Players expect the ability to interact with their environment. If players injure an alien, they expect to see blood. If they shoot a wall, they expect bullet holes. If they wreck a vehicle, they expect it to be damaged. If they are walking through water, they expect to see waves. The demand for realistic behavior of all the objects in the gaming environment is strong. Methods for attaining such realistic behavior are being sought.

Similar expectations are being placed on the animation industry. The 2D hand drawn cartoons of the past, while still appreciated, are being merged with, if not replaced by, computer animation. Computer

characters and special effects are taking a strong hold on audiences. Their usage is becoming almost common in recent movies.

The use of a particle based modeling framework applies directly to animation and game development. In these areas it allows more realistic behaviors to be incorporated into previously and newly designed objects. It allows user specified artistic effects to be achieved while applying realistic physical laws. It also allows the easy incorporation of objects in different state phases. Thus it is possible to create scenes where smoke, water and solids naturally interact with each other with little effort from the animator or game designer.

1.4.2 Dynamic Object Reconstruction

Recently, as range scanning devices have become more prevalent, the desire to recreate objects from a set of surface points has increased. Further propelling this demand are medical and geophysical imaging techniques, where densities of scanned objects at given points in space can be measured. The advantages of being able to create and display meaningful representations of these objects in a graphical fashion are obvious.

Creating objects from particles is not limited to just modeling the surface of objects. The methods proposed here can be used to create volumetric objects directly from scanned density data or from reasonable assumptions about the internal structure of surface scanned data. Specifically, with regard to scanning methods where the density values of the scanned object change within the object, it is possible to represent each level of density with a different type of particle (e.g. particles of bone, blood, skin, granite, lava, clay, etc.). Thus a *dynamic*, volumetric model of the scanned object may be created as a particle representation.

This is a variation of traditional particle density representations where the number of particles indicates the density. While such a representation remains feasible, with the object representation described here it is

possible to create a system where *the types of particles reflect the nature of the materials* defined by different densities. Thus appropriate material properties for such things as the elasticity of skin and the breakability of bone, can be set. Once such a model is created from a scan, it can be animated. This allows "what if" scenarios to be carried out, which include dynamic motion such as blood flow.

1.4.3 Parallel Computing and GPUs

In recent years video cards of home computers have acquired their own graphics processing unit (GPU). This unit is specifically designed to work well with matrix transforms on a large number of 3D points. These cards perform vector and matrix computations in a very efficient manner and research has been done to offload such computations from the CPU to the GPU. Such endeavors demonstrate that the use of the GPU can greatly improve the performance of basic algorithms. Among the list of such algorithms are those of sorting and searching [4, 5] and a variety of numerical methods [6].

While the majority of the work completed for this dissertation uses the CPU, it is known that particle systems work well within parallel environments [2]. So in designing the particle framework, effort is made to make it amenable to implementation in parallel environments, such as that offered by current GPUs. For the moment such implementations are reserved for future work, but are certainly of interest.

1.5 Challenges and Solutions

In the design of a modeling framework to allow objects of various types, phases and shapes to interact and change within their environment two major problems need to be solved. The first is how to represent and create the objects. The second is how to implement the modeling engine. In both of these challenges theoretical and application issues must be addressed. Of concern is keeping the theory simple and robust, and keeping the implementation easy enough to be done by one person in a short amount of time.

1.5.1 Object Representation

The first major obstacle to overcome is to determine a way to represent objects that is general enough to allow a wide variety of objects to be modeled. While there are several choices for solid objects, there are very few for liquids and even fewer for gases. In the end, representing the objects as a composition of basic elements allows a great diversity in use. So a basic element must be chosen.

The selection made is a spherical particle. This proves to be a better choice than shapes such as cubes, tetrahedrons or ellipsoids because the theory will still apply to the other choices. Further, spheres have no inherent direction to be connected, and thus, can approximate a larger number of forms without creating inappropriate sharp edges or faces.

1.5.2 Object Creation

Once the basic representation of objects is decided, it is necessary to devise a way to create objects. Optimally, such a method needs to convert already created models into particle representations and the particles are not to be arranged in any pattern. Meeting this challenge is not as simple as it first seems, but it can be done. The details of this are presented in section 3.8.

1.5.3 Engine Implementation

The second major obstacle to overcome is to correctly describe the abstraction behavior. This problem is one of controlling the objects. In designing the engine things such as how to display the objects and how collision detection and response occur are two fundamental points of concern. Further, as objects can fracture and break, transmitting collision forces through objects is a major concern. To complicate things more, objects in various states need to interact correctly. So the major challenges, in terms of engine design, fall into three subcategories: object creation, display and collision processing.

1.5.4 Object Display

There are various methods that could be used for object display. A surfel based approach is very particle-like. To apply such a method, surface particles and surface normals need to be calculated from the particle representations. Details of this are presented in section 5. An example of the result is shown in figure 1.

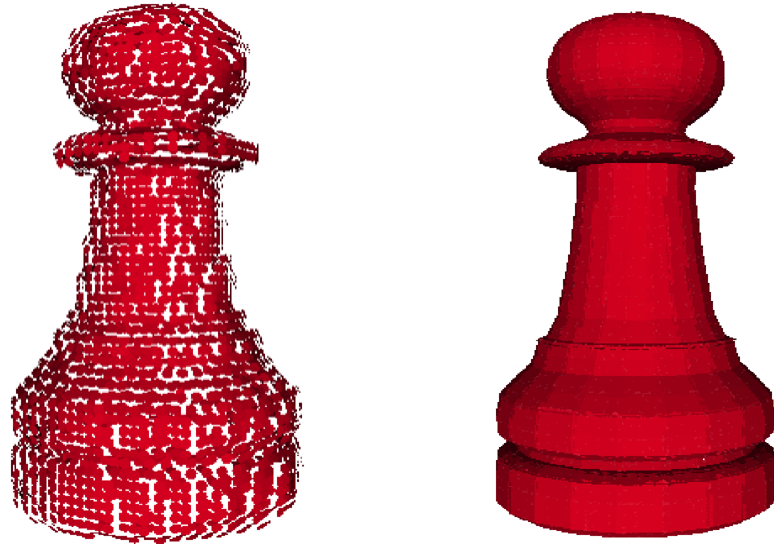


Figure 1: Chess pawn renderings.
A chess pawn rendered using surfels and blended surfels.

1.5.5 Collision Processing

In terms of collision detection it is necessary to create a method that allows objects in different phases to interact with one another as well as solids to correctly impact each other. During these impacts it is necessary to create and maintain fractures and to recognize broken pieces as separate objects. A point of concern is to be certain particles on either side of a partial fracture recognize they are part of the same object. Yet, at the same time, two particles in different pieces of a broken object must recognize they are no longer in the same object.

Also in the scope of collisions is the need to propagate forces through a solid object and determine if any fractures or breaks occur from a given impact. Further, some form of memory of damage already done to

the internal structure must be maintained. Of course, a way to determine what phase state an object is in must also be available, since liquids and gases do not fracture. The resolution of these challenges is found in sections 3 and 4.

2. BACKGROUND

Within this section is a summary of background material viewed as necessary to understand and appreciate the results of this dissertation. Focus is given to the material with a direct relation to the development of a particle based modeling framework. Further background is given in Appendix A.

In section 2.1 is a brief presentation on the history and current usage of particle systems. This is not the complete history, nor is every usage of particle systems discussed. It is just a sampling, as a full coverage would take far too many pages. In section 2.2 an overview of surfel and splatting techniques is offered. These methods are later reapplied to be used within our framework (section 5). Section 2.3 presents a brief summary of fracture and breaking methods. Focus is given to those commonly used in computer graphics. The background presented relates to the fracture and breaking example found in section 6 and suggests a possible motivation for describing solid objects as particles.

While the methods described within our framework certainly relate to finite differencing methods and finite element methods, the background in section 2.3 is sufficient for general purposes. A significantly more thorough presentation of such methods can be found in Appendices A, B and C.

2.1 Particle Systems

Particle systems are wonderful creations designed to allow the modeling of amorphous objects. These objects are usually those more freeform in behavior or composed of items which may behave in a loosely organized fashion. The most often cited particle system paper is Reeves' [7]. It was this paper and the use of such systems in movie effects, that established particles as viable for modeling fuzzy things. It also set the general framework for many of the particle systems used today. A point of interest is that the majority of particle systems are used to represent masses, rarely are they used to model force or energy.

In general, current particle systems are sets of individual primitive objects. In most cases these primitives are represented as spheres, circles, squares or triangles. Each particle is given a set of physical attributes and a set of governing rules is applied to make the entire particle system move. These systems are usually developed to model things without a clearly defined structure such as gases and liquids [8-11]. Particle systems have also been used to model things with a changing structure such as cloth [12-14] and other deformable objects [15-17]. Surfaces [18] and the generation of surface textures [19] have also been modeled using particle systems.

Within these systems a tradeoff is made. In all cases the particle systems are governed by a simple set of rules. This allows a number of effects to be produced automatically, but offers only a limited amount of control over the entire effect. Another difficulty of particle systems has been the restriction on the number of particles that can be employed. The larger the number, the slower the system. It is known that parallel processing environments can be used to improve particle system performance [2]. It has also been demonstrated the GPU can likewise improve performance [5].

In the basic particle engine, each particle is minimally given a position, velocity, color and lifetime. The motion of the particles is driven by: global forces such as gravity, contact forces such as collisions and friction, physical interactions with each other such as spring or repulsive forces, and interior, self-contained forces similar to artificial intelligence or random deviations. The system itself is given a generator. This generator is specified as a 2D or 3D shape placed in world space. Within this generator shape the new particles emerge. In some cases the generator itself is allowed to move within world space.

Once the particle properties have been declared and the forces acting upon them determined, an engine is designed to move the system. This motion is performed across small time steps or frames. During each time step the general tasks performed are:

- remove particles that have exceeded their lifetime
- generate new particles

- initialize the new particles' attributes
- apply the forces to each particle as appropriate
- update each particle's position and velocity
- render the scene

In the update process, the forces for each particle are calculated before any change in position of velocity is made to any particle. This maintains consistent behavior between particles. For example, if particle *A* reacts to particle *B*, then particle *B* should react to particle *A*. To maintain this consistency a copy of the state of each particle is made before calculating forces. This update process is shown in table 1.

Table 1: Improved update process for a particle system.

Name	Update
Parameters	dt, this is the time step size to be applied to the system.

Pseudo-Code	void UpdateImproved(double dt)
	{
	  cur_time += dt;
	  RemoveOldParts(part_list, cur_time);
	  GenNewParts(part_list); *// assume initial values given also*
	  part_list.CopyTo(copy_list);
	  for (i = 0; i < copy_list.NumParts; i++)
	  {
	    copy_list.CalcForce(i, part_list); *// Forces based on part_list*
	    copy_list.UpdateState(i, dt, copy_list); *// Changes made in copy_list*
	  }
	  copy_list.CopyTo(part_list);
	}

The display routine of most particle systems is straightforward billboarding. Sometimes, to allow blending, the particles will be sorted based on the distance from the viewer. In almost every system there is no explicit boundary of the object being modeled. Thus the display of the "particle object" is a perceived shape, formed by the individual particles.

While most particle systems are used for special effects such as dust clouds or simple explosions, they can be used for more complicated tasks. To fully illustrate the use of particle systems, brief descriptions of various applications are offered in the following sections.

2.1.1 Flocks and Herds

In 1987 Reynolds showed particle systems can model flocks or herds of animals. In his systems each particle represents a bird (boid) object [20]. The motion of each boid is governed by a set of rules. The rules prevent boids from running into obstacles and each other and cause the boids to form clusters. Cumulatively these rules create particle systems that appear to move as a flock of birds. However, the application of these rules increases the complexity of the implementation. Because the boids are aware of their environment, an abstraction of their environment and positions relative to one another must be maintained. This interaction with and dependence on each other increases the complexity of the code and can decrease the runtime speed.

Another notable difference in Reynolds' system is each particle is displayed as a set of polygons rather than a simple primitive object. Also, the number of particles remains fixed throughout the simulation, thus there is no generator and the lifetime of each particle is infinite.

2.1.2 Growing Patterns

Particle systems can also model static or instantaneous things. They have been used to model plants and fracture patterns [21, 22] as well as lightning, frost, ice or snowflakes [23-29].

These methods directly or indirectly relate to diffusion limited aggregation (DLA) methods [30, 31]. In sum, the particles generated in DLA methods go on random walks. When a particle encounters an attractor or another particle, it sticks. Should the particle wander too far from the area of interest then it is killed. Once a particle is stuck or killed, another particle is generated. This is repeated until a given number of

particles have been generated. More on DLA methods is found in section 6. There they are used to create fracture patterns to be applied to solid objects.

2.1.3 Surfaces and Textures

In the majority of cases, particle systems are designed to represent objects without a well defined surface. However, in the early 1990s Szeleski and Tonnesen introduced a particle system designed to *create* a surface [32]. They termed these systems as "oriented particle systems." These systems are composed of particles capable of organizing themselves to represent the surface of a given object. This is achieved using attractive and repulsive forces between particles.

Also in the early 1990s, Turk [19, 33] designed a particle system based on reaction diffusion to create surface textures using particle systems. These are similar in behavior to oriented particles, but have extra guiding properties based on the color associated with the particle. The textures generated by these particle systems are similar to color patterns found in nature, such as color markings on animals. More information on reaction diffusion textures can be found in the papers by Witkin and Kass [34]. Further information on its mathematical concepts can be found in the work of Turing [35].

2.1.4 Solid Objects

In 1998, Tonnesen devised a way to create volumetric objects from particles for purposes of realistically sculpting objects [36]. This, like the oriented surfaces, is based on attractive properties of particles. In sum, he applies short and long range forces to attract particles together, but not overlap. Thus an object coherency is maintained by interparticle forces. *However the particles are not explicitly connected.* This description allows objects to be created that bend under gravity or repulsive collisions with environmental objects. He also developed heat transfer equations that cause the attractive forces between particles to dissipate as the temperature of the object is increased. Thus his objects can melt.

Some weaknesses in his method are noticeable. For example, his objects are singular and only interact with the environment. So, *no method for distinguishing two separate objects is made*. He also *does not create objects from existing object representations*. His examples are all basic geometric shapes. Also, his forces are all attractive, or repulsive, in nature. *Force propagation has no bearing in his framework*.

This should not be taken as minimizing his work. It is quite impressive. As his major goal was to create an artistic development environment based on the conceptual model of clay, these limitations are not a concern to his work. The above points are made to make it apparent his work is significantly different from the goal, design and methods of this dissertation.

2.1.5 Molecular Dynamics

Particle systems have been used in the field of molecular dynamics to model volumetric objects [37, 38]. In these applications the microscopic behavior of atoms and molecules is of extreme importance. These simulations are designed to be as physically accurate as possible. They are based on physical, chemical and biological properties. The goal of such applications is to study the interaction of microscopic behaviors. *These simulations are not designed to run in real time, nor are they designed to model the macroscopic interaction of multiple objects. Further, they have little interest in the surface display of objects* and almost all use an atomic display, where each particle is displayed as a sphere. While they certainly are similar in design to some of the work in this dissertation, they are significantly more interested in microscopic interaction behaviors.

2.2 Splatting Techniques

Of special interest to the development of a particle based representation of objects is the recent effort to display raw point set data used to describe objects. There are two keywords which describe these efforts: surfels and splatting. The word surfel stands for surface element and was brought to light in the paper by Pfister et al. in 2000 [18]. Some also refer to these as splats. This second term is derived from the process of splatting as described in the papers by Rusinkiewicz et al. and by Zwicker et al. [39, 40]. It should be

noted that the conceptual idea of splats comes from others [41], but was not fully recognized until around 2000.

Splatting works with objects represented as a dense surface set of oriented points. Rendering is accomplished by displaying an oriented disk for each surface point and blending the disks together. In mathematical terms this means the object's surface is represented as a superposition of oriented 2D Gaussian basis functions (or similar) which are truncated to provide local support. The disks are referred to as surfels. Surfel attributes comprise depth, texture color, surface normals, and other features of the object. So when performing splatting the object is represented as a collection of dense, oriented, surface points, most often derived from a range scanning device. Rendering objects using surfels allows for complex shape representation, low rendering cost, and high image quality. This, combined with the point set surface representation, makes them well suited for application to particle based modeling.

Because of this nice fit, a brief background of the process follows. This description is based on the work of Pfister, Zwicker, Rusinkiewicz et. al [18, 39, 40]. To apply these methods to a dynamic environment of volumetric particle representations requires some modification and extraction of data. The details of such adaptations can be found in section 5.

2.2.1 Splatting in General

The surface point representation of an object needed for splat based rendering is often obtained using 3D scanners. However, the points may also be derived by point sampling a pre-created surface representation of the object. Traditionally such scanned point sets have been converted into a polygonal surface representation of the object. However, as the use of such technology expands, alternative methods for displaying such data is being developed. Splatting has become one of the more popular methods. This is primarily for the high quality rendering it can produce, but it also offers other benefits such as permitting stochastic sampling to be employed, which allows effects such as motion blur to be possible [42]. Further,

shadows are supported by using a z-buffer algorithm [43] or by ray-tracing [44]. Other lighting effects are also achieved using ray-tracing methods.

Splatting is also simpler to implement than polygon render systems and, as it requires no connectivity information, it parallelizes better than a polygon renderer. The rendering times it offers are comparable to those of non-antialiased polygon rendering times. These features make it probable that hardware implementations of such rendering schemes will become prevalent in the future. All of these advantages synchronize well with particle based representations of objects.

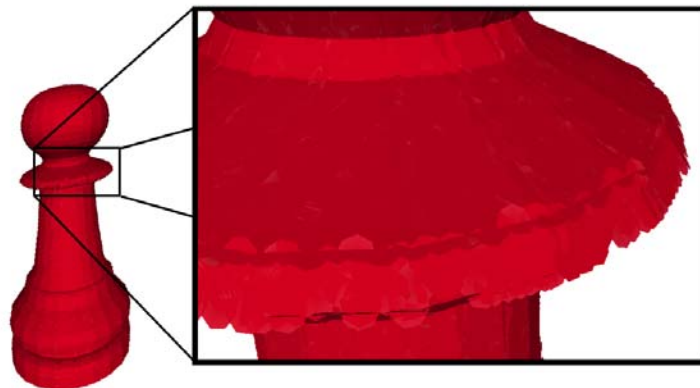


Figure 2: Artifacts of splatting.
If not well blended, splatting methods may produce artifacts. The above image is a surfel based rendering of a chess pawn represented using volumetric particles.

While useful, splatting is not perfect. As it is basically a bunch of overlapping circles, proper blending is required to prevent artifacts, as shown in figure 2. These artifacts while disguised at a distance, become more evident when the object nears the viewing plane. This small hindrance is circumvented by advanced blending techniques and dense surface sampling. However, this is only necessary when an object will be closely examined. Thus the artifacts are more a level of detail (LOD) problem than an issue with the display technique itself. Similar LOD problems exist in all object renderings.

2.2.2 Basic Splatting Algorithm

The full details of splatting algorithms are available in various papers [18, 39, 40, 45]. However, a brief summary is presented here because this method is the basis for the one described in section 5.

Splatting begins with a point sample of an object's surface. For each point this sample includes a position, a normal, a color and a variance matrix. The variance matrix determines the average distance to neighboring samples and also determines the shape and size of the reconstruction filter (i.e., the size and shape of the ellipse). To reconstruct the point samples an oriented reconstruction filter is placed on each sample point. The reconstruction filters are then transformed and projected into screen-space to form the object. The rendering primitive used to display each such filter is a splat.

The standard splatting algorithm is broken into two passes. For the first pass, each splat is transformed and rasterized into screen space. The rasterization of each splat is stored in an a-buffer. This may simplistically be thought of as a z-buffer as it is also used to remove hidden surfaces [46]. During the second pass, each pixel is assigned a color derived from the rasterizations stored in the a-buffer. A general description of the code for this is shown in table 2.

Table 2: Splatting pseudo-code.

First Pass:	For every splat
	{
	    For each pixel of the splat's transform and rasterization to screen space.
	    Store its color, distance and weight (call this the splat's fragment).
	}

Second Pass:	Sort the stored fragments by distance.
	For each pixel
	{
	    Find the fragments effecting it.
	    Blend/select the color of the pixel (weights are used for blending).
	}

While there are numerous details omitted from this two-pass process, it is sufficient to understand the process. The important features to note are: the object is represented only as a set of surface points and the surface normal (splat orientation) must be known at these points.

2.3 Previous Work on Fracture and Breaking

Given the usefulness of being able to model fracture and breaking, it is not surprising that a significant amount of work has already been done on the subject. Much of this has been performed in the engineering fields [47, 48]. However, we are more interested in the graphical development and expression of fractures and focus our background summary to the field of computer graphics.

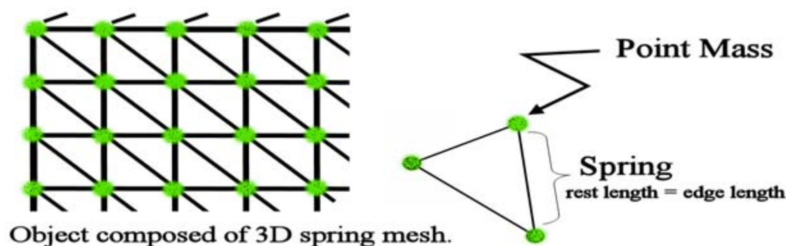


Figure 3: Spring object model.
To model fracturing, objects composed of a spring mesh can be used.
The vertices are considered point masses and the edges are springs.

Some of the earliest work on fracture modeling (in computer graphics) was done in 1991 by Norton et al. [49]. In their method they represent an object as cubical masses connected by springs. When a spring is stretched beyond a pre-specified limit, the spring breaks. The broken springs represent fractures within the object. The basic layout of this modeling structure is described in figure 3. More information is given in section 2.3.1.

A more accurate method is presented by O'Brien et al. [50, 51]. In these papers the authors represent the object as a tetrahedral mesh. Using a finite element method (FEM) applied to the tetrahedrons they

develop a technique to fracture and break the object based on the material properties of the object. The results are outstanding. More on this method is presented in section 2.3.2.

Other work on fracture and breaking includes the modeling of inelastic deformation done in 1988 by Terzopoulos and Fleischer [52]. In this paper viscoelastic and plastic deformations as well as fracture are investigated. In 1999, Neff and Fiume presented a technique to simulate the breaking of a window under a spherical blast wave [53]. Their model is based on the theory of rapid fracture propagation. In this method a small initial fracture is introduced in the object. This initial fracture then propagates through the rest of the material. To create a more rugged fracture pattern various parameters in the simulation are randomized. Also in 1999, Mazarak et al. [54] offered a method to model the effect of a blast wave on nearby objects. In this method the objects are represented as a rigidly connected set of voxel elements. Fractures occur when the rigid connection between two voxels encounter a pressure force beyond a predefined threshold.

In 2000 Smith et al. presented a realtime brittle fracture method based on constraint dynamics [55]. This method, like others, models its objects using tetrahedral meshes. It also offers the user limited control of the size and number of fragments by adjusting the constraints. However, there is no control of the actual direction or shape of the fracture pattern. A visible limitation of this method is that the fractures occur only along predefined element boundaries. Another realtime method was presented in 2001 by Müller et al. [56]. This method models deformation and fracture. It also uses objects composed of tetrahedral components and applies finite element methods. Using static analysis of impact events, it achieves an increase in speed performance over similar methods.

In all of these papers some common threads emerge. Among these is the representation of the objects as a composition of smaller objects. To examine this detail we look more closely at the techniques proposed by Norton et al. [49] and O’Brien and Hodgins [50]. Focus is given to these two methods because, between them, the basic methods used by others is covered.

2.3.1 Spring Method

In Norton's method, the fundamental unit of an object is a cube connected by a set of internal springs. To model object curvature the cube structure is modified to also curve and the spring settings are adjusted to approximate this curvature. The motion of the object is based on force transmission through the connecting springs. The breakage pattern is determined by springs breaking when they are overextended. The general collision detection and response routines are based on repulsive methods. This model of breaking objects works well and is straightforward to implement; however, it suffers from several limitations. Among these are the time necessary to run the simulation, the jaggedness of the breakage, and the quantity of "dust" cubes produced.

The time needed to run the simulation becomes enormous if the system of equations created by the springs is too stiff. This stiffness requires exceptionally small time steps. This can be partially overcome by increasing the cube size and thus decreasing the number of springs. This does not increase the time step but reduces the number of items being updated, or rather reduces the size of the system, which reduces the number of computations needed. With fewer computations necessary, the overall speed of the simulation increases.

However, if the cube size is increased then the fracture and breakage becomes extremely jaggy in appearance. This jaggedness is the result of allowing fractures only to occur between cubes. Thus fractures appear as stair-like structures and if the cube size is large this jaggedness becomes more apparent. It is also noted this method of fracturing produces a large number of cubic dust particles, as individual cubes have a tendency to break away from the object. So when the cube size is increased these dust particles become larger and more visible.

Another criticism of trying to use this method is the development and creation of the objects is extremely user intensive. Specifically every object must be hand crafted from a set of cubes. Each cube must then have its springs manually adjusted to properly set the rest lengths needed to achieve any curvature within

the object. Once each individual building piece is created it is then necessary to put the pieces together and make certain a stable object results. After accomplishing this, the spring settings must be calibrated to achieve the desired breakage behavior. While this is time consuming, it is not that bad when only one or two objects are needed. However, it is a criticism.

2.3.2 Finite Element Method

In O'Brien and Hodgins' method the objects are represented as a set of tetrahedral elements. To implement fracture and breaking, they develop equations to transmit impact forces across these elements using a finite element (FEM) approach. As the force propagates through the elements each element can potentially fracture based on the direction the force tip is inclined to go. This method is likely the most accurate method described for computer graphics use. The results it is capable of producing are remarkable; however, it is extremely complicated and difficult to implement correctly. It also suffers from a slow runtime and is exceptionally difficult to use in a dynamic environment containing a large number of breakable objects. This is clearly true in its requirement to repeatedly re-tetrahedralize objects as they fracture. An example of globally retretrahedralizing an object is shown in figure 4.

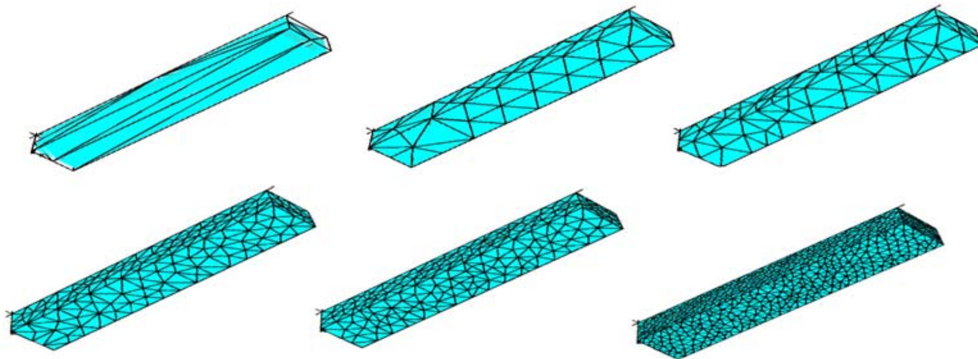


Figure 4: Refining the tetrahedralization of objects.
This is time consuming, even when done locally.
The above illustrates a simple bar undergoing global refinement.

O'Brien and Hodgins' method, like Norton's, transmits the entire impact force through the object on an element by element basis. Yet, it significantly improves on the accuracy of the force propagation and allows fractures to occur anywhere within the solid by re-tetrahedralizing the object as necessary. While this is a gain in visual appearance, this method of propagation results in a slow implementation due to the small time steps required to maintain stability and accuracy.

In sum, this method is complicated to correctly implement and requires extensive and dynamic re-meshing of the object. It also is not designed to run in a realtime environment and even today takes hours, if not days, to completely process a single impact. Another implementation concern is that no specification on how to obtain the initial tetrahedralization is offered. But more importantly any restrictions, if any, on the tetrahedrons are unstated. However, the technology to accomplish tetrahedralizations has improved since the original date of these papers and it is likely a program such as TetGen [57] would suffice.

3. VOLUMETRIC PARTICLE OBJECTS (VOLPARS)

This section describes how particles are used to represent objects of substance. Several definitions are provided followed by a description of particle bindings, shapes and state phases. For each state phase a description of the behavior of particles when used to represent objects in the given phase is provided, followed by a discussion on how existing models (solid objects) are translated into particle representations. The conclusion of this section describes particle grouping methods applied to solid objects and a space partitioning method useful for any type of object.

3.1 Definitions

Within the modeling framework, the representation of material substances is particle based. VOLumetric PARticle representation is abbreviated ***volpar***. A volpar object is defined as: ***A set of particles such that the particles fill the majority of the represented object's volume and reflect the geometry and topology of the object***.

This means the objects are a distinct type of particle system, where a particle system is defined as: ***A collection of particles such that each particle has a set of attributes defining its state***. Changes in a particle's state are governed by a set of rules. These rules are usually responses described by reactions to Newtonian forces within the particle's environment. The state of a particle includes attributes such as position, velocity, acceleration, shape, size, mass, color, etc.

3.2 Particle Bindings

A given particle may be bound to other particles. The nature of this binding reflects the material substances the particles represent. This binding also reflects the state phase of the materials the particles represent. Particle bindings also are referred to as intra-particle, or intra-object, forces. These bindings are divided into two categories: ranged and connected.

Ranged bindings between particles are similar in form to Lennard-Jones (LJ) force potentials. The generic form of an LJ force potential is shown in figure 5. The conceptual model of these forces is to establish:

- a repulsive force between two particles if they are too near each other
- an attractive force if they are "somewhat" near each other
- no force between them if they are far from each other

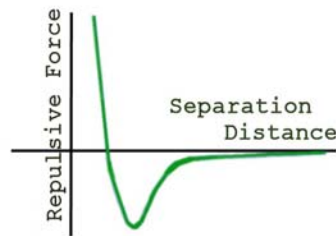


Figure 5: Generalized form of a Lennard-Jones force potential equation.

Connected bindings are more explicit than ranged bindings. These bindings are modeled as rigid rods, or springs. In the case of rigid rods, each binding is given a strength rating. In the case of springs, each binding has a maximum "stretch" length. In both cases the binding may be broken.

Assuming there are *n* particles in a system, to maintain these bindings and their effects can require a magnitude of $O(n^2)$ operations. To avoid this, with respect to ranged bindings, space partitioning reduces the complexity to $O(kmn)$, where *k* is the number of neighboring cells to be considered and *m* is the maximum number of particles in each cell. With respect to objects of connected particles, most forces are applied as if the object is a rigid body. To propagate forces through the object, wave-based propagation is used. This reduces the complexity to $O(n)$. Details of this are presented in sections 3.10, 4.2, and 4.3.

3.3 Particle Shapes

In most cases particles are presented as spheres. This is done for consistency of discussion, but particles may be of different shapes and sizes. When not spheres, they remain associated with a bounding sphere.

This association is done to provide a consistent mechanism for distance calculations, collision detection, neighbor determination and surface identification. Section 6 contains examples using particle shapes such as cubes and cylinders. Yet within the modeling framework, they are still considered spherical for many calculations. So particles are not required to be spherical, but making them spherical, or bounding them with spheres, simplifies and maintains consistency of calculations.

3.4 State Phases

Particles represent material substances in various states. Sets of particles are identified as being part of a given object in a given state. Most objects are solid objects composed of a set of explicitly connected particles. However, a set of particles may also represent a body of liquid, such as a water puddle. In such cases, the puddle of water is an object and every water particle is part of that object. In such amorphous objects, the particles are not explicitly bound together, but experience ranged bindings. They may also experience ranged repulsion, or attraction, with other objects. Such relations are needed, for example, in the case of oil and water objects. Oil particles do not attract water particles, and may mix with them. However, given time, the oil will eventually “float” on the water.

To model objects in various state phases, particle behavior must reflect objects in each state phase. This behavior must accommodate the interaction of particles representing an object in a given state phase with particles representing objects in the same and other state phases. This behavior is summarized in table 3.

Table 3: Description of how state phases affect particle behavior.

State Phase	Binding	Collides With
Gas	None	Solid, liquid
Liquid	Ranged	Solid, liquid, gas
Solid, Breakable	Rigid Bar	Solid, liquid, gas
Solid, Deformable, Elastic	Spring, Rigid bar	Solid, liquid, gas
Solid, Deformable, Plastic	Ranged, Rigid bar	Solid, liquid, gas

These relations are presented in general terms. The exact details are parameter driven, based on the materials being represented. Such parameters are specified when a given object is created or loaded from a file. This specification may be hard coded or user defined.

3.5 Gas Particles

Gas particles represent a gaseous substance or substance appearing to be a gas. Substances appearing to be gas are included in this to improve efficiency and better reflect how people interpret what they see. For example, smoke is technically not a gas; however, the majority of people believe smoke is a gas. By modeling only the smoke as a gas the region of modeled air is isolated. This reduces the number of particles required. If necessary, all the air could be modeled and burning particles would float therein. But this is unnecessary when appearance is the only priority.

When initially created (born) each gas particle is given a randomized velocity such that the average velocity of all the particles is around a specified vector. In most cases this is the zero vector; however, this does not mean most velocities are zero. In fact, the magnitude of each velocity ranges from a minimum to a maximum. This range guarantees none are zero. So, if a large number of gas particles are created in the same location, they will disperse. If the gas particles are not bounded by a containing object they will disperse to such an extent as to no longer be within the modeling environment. If they are bounded, then they will bounce against the containing particles.

In terms of particle to particle behavior, gases experience no binding forces. They experience no collisions with other gas particles; however they do experience collisions with solid and liquid particles. Thus when implementing collision routines gas particles are not individually checked. They are only noticed if a solid or liquid particle detects them as a potential collision. These collisions, for the gas particles, are viewed as perfectly elastic. So collisions do not reduce the speed of gas particles.

Without a reduction in speed, the gas particles, cumulatively, can establish a pressure force capable of moving solid or liquid objects. With this pressure, obviously, is the concept of temperature increasing the velocity of the gas particles. Further, the non-collision between gas particles reduces the number of calculations needed to model the gas without any significant loss in observed results. The explanation for this observational result is intuitive. In effect, the direction of motion of the gas particles is a random distribution. Thus for any large number of particles, the total effect of all the collisions would, on average, result in an equally random distribution. So with perfectly elastic collisions, no significant variation of results would be apparent. Thus, there is no reason to waste the calculations needed to model the collisions. Additional results of these elastic collisions will be discussed later in terms of force and pressure (section 4).

So in the modeling framework, gas particles have no binding forces, experience elastic collisions with solids and liquids and do not collide with each other. This does not adversely affect the overall behavior of a gas substance. Further, in terms of displaying substances, such as smoke, the non-collision between gas particle allows them to overlap. This allows blending effects to be used to visually represent the density of the visible gases (or gas-like substances).

3.6 Liquid Particles

Liquid particles have no explicit connections defined between them. However in the real world, liquids exhibit various amounts of cohesion. Thus liquid particles experience a variable amount of attractive force between each other. This attractive force is not limited to immediately adjacent particles but may extend across short to medium distances. Thus liquid particles experience ranged binding forces. These attractive forces and the associated distances vary based on the type of liquid being modeled. The exact values for these forces are parameter driven and may be hard coded or user specified.

In many of our applications these ranged forces were scaled based on gravity, g. For example a given liquid particle may experience an attractive force of $0.5g$ towards another liquid particle when they are

within 2.5r of each other, where r is the radius of each particle. This scaling of forces with respect to gravity is useful when the visible behavior of the particles is the primary concern. For more realism, these ranged forces may be set based on physically based attributes.

Liquid particles experience collisions with each other and solid objects. These collisions are inelastic. This also means liquid particles repel each other if they are too close together. However in many cases, liquid particles are allowed to overlap to a small extent. This behavior allows for a small amount of compression within a liquid, as well as an appearance of joining together.

In some cases, liquid particles experience attractive forces to solids. For example, water tends to stick to solids. Liquids may also experience various attractive or repulsive forces with other liquids. For example, water and oil separate even when thoroughly stirred. To simulate these interactions other ranged bindings can be specified for liquid particles of a given type towards solids and liquids of other types.

So in our framework, liquids experience ranged bindings. They collide with solids and gases. They may experience limited binding forces with other liquids or solids. These ranged bindings may be implemented, in form, as LJ-force potentials. This representation of liquid does have restrictions. A notable point is they model a compressible fluid. So, pressure changes on one side of a liquid object will require time to propagate to the opposite side.

3.7 Solid Particles

Solid objects are composed of explicitly connected particles. They are sub-typed into three categories of objects: rigid, elastic deformable and plastic deformable. The type of the object determines the type of the binding connections between particles. The details of these connection arrangements will be given shortly.

Not all solid objects are composed of homogeneous material. To model heterogeneous objects particles of different material substances are connected together. For example, a brick wall may be composed of brick

particles and mortar particles. Such compositions mean the connections between particles of different material types are different than connections between particles of the same material. Continuing the example, the binding connection between a brick particle and another brick particle is stronger than one between a brick particle and a mortar particle.

Key in each type of solid object, is the notion of a breakable, or deformable, connection. How these connections are used is one of the novelties of this work. Unlike previous work the connections are not used to model the motion of the solid body. Instead, as described in section 4, most forces move solid objects using rigid body transforms. The connections between compositional particles is only significant when handling impact forces and possible fracture or breakage. Details regarding the forces are in section 4 and only the connections are described here.

When an object breaks, a connection must have broken. So during any time step of the simulation in which a connection breaks, it must be determined if the object has broken into two or more pieces. To accomplish this, each connection is aware of its end particles. Each particle is aware of the object in which it is contained. Each particle is also aware of every connection to which it belongs. Thus, when a connection breaks, a depth first search is performed on the object containing the broken connection.

This search begins with one of the connection's end particles. If the depth first search encounters the other end particle of the connection, then both particles are still part of the same object. So the object remains whole. If the depth first search does not find the connection's other end particle, then the object has broken into two pieces. The search from the first particle identifies which particles compose one of the pieces. To identify the "newly" created piece a second search is performed from the connection's other end particle.

3.7.1 Rigid Breakable Objects

Rigid solids are the basis of our solid object representations. These objects are assumed to be breakable. To allow them to break, the binding connections between their particles must have a strength. This

strength decreases when the connection is damaged and increases with the passage of time (to a maximum). The initial strength of the binding can be a relational value or based on the fracture toughness rating of the material [58]. In either case, objects with a high strength value are more difficult to break.

In implementation relational strength values are more useful. For example, assume that object A is twice as difficult to fracture as object B. Then the connections in object B may be given a strength of ten and the connections in object A may be given a strength of twenty. Of course, this is not necessarily physically accurate; however, it is behaviorally accurate. Further, if it is necessary, these values can be derived from scientifically determined material strengths. The modeling framework still functions correctly and the accuracy is relegated to user interests.

3.7.2 Elastic Deformable Objects

Elastic deformations are such that the material deforms, but returns to its original shape. To model elastic deformations a constrained spring based connection is used. This means the general motion of the particles is spring based with a restriction placed on how far a spring is allowed to move in one time step.

While it is possible to use springs for every connection in the object, such a system results in one very similar to that used in the work of Norton et al. [49]. These systems are subject to numerical instabilities when the system of springs is too stiff or the time step size is too large. To minimize the likelihood of this problem the number of springs is reduced involved by only using spring connections between particles near the surface of the object. The interior particles of the object are bound using rigid connections. Performance is also gained by heavily damping the spring motion. Further gains are achieved by the impact force separation, as discussed in section 4. This force separation reduces the forces traveling through the springs, so their motion does not propagate very far. Yet the end result is an object whose surface can deform and spring back.

Deformable objects may also break if the springs are compressed or stretched too much. They will also break if the interior rigid connections receive too much damage. In application this rarely occurs; however when it does a rather interesting difficulty arises. When such objects break, the surface particles must be re-identified and their connections must be set to springs. While identifying the surface particles is straightforward, the alteration of binding type can be complicated and slows the simulation.

In sum, these objects work well as long as they do not break. While this could be a problem, they do not break very often. In simple terms, the forces required to get past the deformation process and break enough of the connections is extremely large. In the end, the focus is on breaking the more rigid solids and exploration of breaking deformable objects is left for future work.

3.7.3 Plastic Deformable Objects

Plastic deformations are such that the material deforms, but does not fully return to its original shape. To model such a deformation a method to allow the solid particles to overlap with each other is required. As in the elastic deformation this was first done by changing the connection type of the surface particles while leaving the interior particles rigidly connected. This works, but proves unnecessary. In the end applying a ranged based binding to all the connections is sufficient. These ranged connections, while similar to those of liquids also have a strength rating. However, the strength of the connection is not reduced until the connection has been fully compressed. Once fully compressed, it functions as a rigid bar.

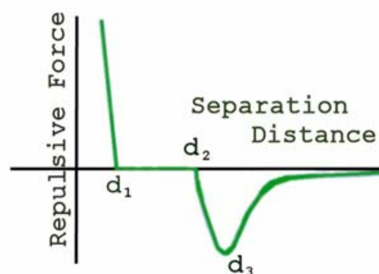


Figure 6: Graph of LJ-potential with an extended rest interval.
This interval allows plastic deformations to be achieved in particle representations of solids.

Mathematically the initial ranged bindings can be modeled using a LJ-force potential with an extended rest length, as illustrated in figure 6. Such a description allows particles to remain at rest when the distance between them is from d_1 to d_2. Should the separation distance become less than d_1 then a repulsive force is exerted to correct the distance to be d_1. When the separation distance is greater than d_2 then the particles exert an attractive force which attempts to bring them to a distance of d_2 of each other. This attractive force is viewed as maximal when the separation distance of the particles is d_3.

This is a good mathematical description; however, in implementation the details are slightly different. Initially the particles are at a distance of d_2 from each other. The particles experience no forces when they are separated by a distance of d_1 to d_2. If the distance becomes less than d_1, then the connection reverts to a rigid bar. If the distance is between d_2 to d_3, then there is a small attractive force exerted. This allows the motion of other particles to bounce compressed particles to a non-compressed state (i.e. a little wiggle room). When the distance becomes greater than d_3 the particles are moved back to be exactly a distance of d_3. This last effect almost never happens, it is included as a precaution.

3.8 Object Conversion

Most objects in computer graphics are modeled using B-reps or CSG representations. To use volumetric particle representations (volpars) a method to create or translate objects is required. While models can be generated by hand, this requires an excessive amount of time, produces a limited number of models and is not very useful to anyone else. So instead an automated method to convert existing B-rep models (of solid objects) into volpar representations is used.

Conceptually the translation process is composed of two tasks. The first is to fill the B-rep object with particles and the second is to create the various connections between the particles. The first task may be achieved in a variety of ways. Here a two step method is used, which is relatively fast and achieves near maximally dense packings. In brief this method divides the object into 3D cubes, places a particle in each cube, and then applies a physically based filling to increase the density of the packing and randomize the

particle locations. The second task is accomplished in a straightforward manner. Once the object is filled with particles, connections are formed based on the adjacency of particles. The strength of the connections is set by a user driven parameter based on the assumed material of the object.

3.8.1 Grid Based Filling, Cubic Representation

There exist many ways to achieve a 3D cubic (voxel) representation of an object [59, 60]. This method is straightforward and is designed to convert B-rep objects into volpar representations. It begins with a B-rep of an object stored in a simple mesh file (smf) as defined by Garland [61]. Almost any B-rep may be expressed in this fashion and most CSG objects can be converted to this format by already existing programs. The conversion from B-rep to volpar is visually demonstrated in figure 7.

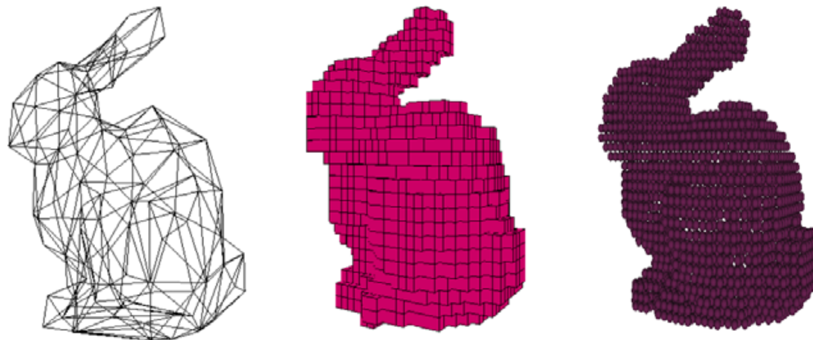


Figure 7: Stanford Bunny transformed from a mesh object to a volpar object.

The first step is to read the desired object into memory from a file. In memory the object is represented as a set of vertices, edges and faces. Once the object is loaded, a bounding box is placed around the object to be transformed. The bounding box is then partitioned into a set of cubes with edge length h. The sides of the bounding box may be increased to be evenly divisible by h if necessary.

For the next step, each vertex of each cube is marked as inside or outside the object. This is done by ray intersection tests with the object surface. One way to accomplish this is to align each ray with the edges of the partitioning cubes. This is not the only way and others exist that can directly generate a face centered

cubic sphere packing. For demonstration the aligned rays are used. Thus the first intersection point with the object is found. All vertices on the ray to that point are outside the object. The next intersection point with the object is then found. All vertices on the ray between the intersection points are inside the object. This repeats until the ray no longer intersects the object (i.e. goes outside the bounding box). A demonstration of this process is shown in figure 8. Special consideration is given to rays that are parallel to any face making up the surface of the object. In most cases this is resolved by jittering the ray slightly in a direction perpendicular to the face. Consideration is also given to rays going through a vertex of the object. Again this is resolved by jittering the ray.

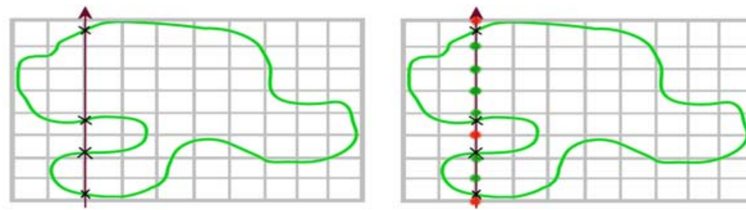


Figure 8: Inside/outside vertex marking using grid rays.
The x's denoted the intersection points of the ray and object.
Green vertices are inside the object and red are outside.

Each cube of the bounding box is now assigned a state of interior, exterior or surface. A cube is interior if all its vertices are inside the object. A cube is exterior if all its vertices are outside the object. A cube is a surface cube if some of its vertices are inside the object and some are outside. This process is illustrated in figure 9. This surface identification is mentioned and used again in section 5.

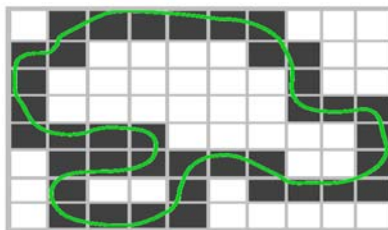


Figure 9: Surface boxes as discovered for a 2D object.

Within each cube a particle-sphere is placed. The radius of the sphere is half the side length of a cube. Each sphere has the same interior-exterior-surface state as its containing cube and is assigned neighbors based on the cube relations. In the simplest case, the maximum number of neighbors for a given sphere is six, with one above, below, left, right, forward and behind. Another possible arrangement of neighbors is one with a maximum of twenty six, with nine above, nine below and eight around. Both arrangements are shown in figure 10.

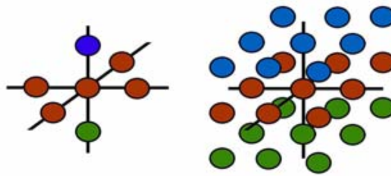


Figure 10: Particle neighbor relations.
On the left the center particle has 9 neighbors. On the right it has 26.

Another interesting neighbor arrangement is based on sphere tangencies, where the maximum number of neighbors is twelve, with three above, three below and six around. This last neighbor arrangement is referred to as a honeycomb or bricklayer arrangement and is associated with optimal packings as proposed by Kepler. Regardless of how many neighbors are maintained for each particle, each neighbor relation is considered to be a connection between particles.

Once the particle neighbor relations are determined, the surface normal for each surface particle is calculated for display purposes. The details and requirements for the display methods being used are found in section 5. As such details are not inherently necessary for understanding our filling methods we will, for the moment, assume any such information is calculated and available when needed.

3.8.2 Packing More Particles, Filled Representation

While cube-like fillings of objects are nice, natural objects are not often so well structured. To introduce randomness into the structure and achieve a more dense packing, an additional step is performed after the

cube-like form is achieved. This second step is referred to as a ***filling-over method***. An earlier presentation of this method can be found in [62]. A visualization of this method is shown in figure 11.

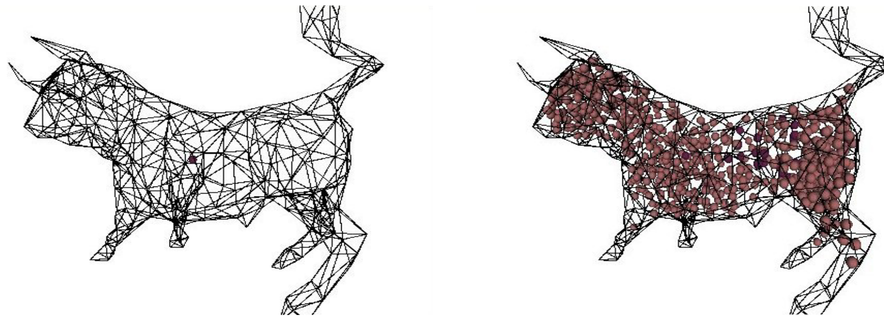


Figure 11: Bull mesh being filled with particles.
In this example the bull was not initially divided into cubes.
When completely full we have a volpar representation of the bull.

To test this method we began without initially dividing the objects into cubic cells. Omitting that initialization greatly increases the time necessary, but the filling process remains the same. So in implementation we begin the filling after we have a cubic representation. However, it is possible to begin the process without such an initialization.

The filling begins with the selection of a point within the object to pour particle particles into. This pouring is a physically based particle simulation. The state of each particle is updated in incremental time steps based on the conservation of momentum and energy with respect to velocity.

Approximately every tenth time step a new particle is introduced into the system with a random velocity. The system of particles is bounded by the object being filled and the particles apply a repulsive force to each other. This force is similar in shape to a LJ-force potential, or ranged binding force, with an extended rest interval. These forces, rather than hard collisions, are used between particles to encourage a dense packing of the object [63]. LJ forces are effective at packing molecules in the real world, so they are expected to be effective at packing spheres into modeled objects. The LJ forces also provide a way to keep

the neighbors of particles from changing too frequently. These properties assist in the overall performance of the method. Without LJ forces, and with only hard collisions providing forces, the particles tend to bounce around and do not easily form themselves into anything. Simply stated, the attractive forces are literally the glue that maintains a semblance of coherence. While the object boundary will eventually force this behavior, the particles do not tend to be so nicely arranged without such a glue.

The extended rest interval of the LJ based forces is not necessarily required, but allows an overlap between the particles. The overlap allows a small amount of shuffling to occur even when the spheres are tightly packed. In practice this provides slightly better results then without it. However, the difference is usually not more than five spheres more per one hundred for any packing.

As an object is being filled, a given particle also experiences impulse based collisions between itself and the faces of the object. This collision response is damped by a coefficient of restitution. So when a particle collides with a face it loses roughly 50% of its velocity. This coefficient of restitution is arbitrary but necessary to reduce the velocities within the system.

To determine when the object is full of particles, the total kinetic energy of the system at each time step is monitored. This measurement is estimated based on the change in position of the particles from one time step to the next. As particles are continually added into the system, eventually there are more than can fit into the object. Thus particles may be forced out of the object. When this happens they are re-spawned back at the pouring point, plus or minus a random epsilon. This epsilon avoids placing two balls in exactly the same location. This re-spawning causes a significant increase in the kinetic energy estimation which allows detection of an overfull state.

Once an overfull state is detected, the number of particles currently in the system is recorded. This number is the *estimated cap*. One or more particles is then removed from the object and the simulation is allowed to run for a few more time steps without the addition of any more particles. This is the *settling period*.

After this period passes, the addition of particles resumes. It is expected an overfull state will again occur when the number of particles is near the estimated cap. If this happens several particles are again removed and another settling period is established. After this second settling period the simulation ends (or may be redundantly tested again). If it does not reach an overfull state a second time, then things continue as though nothing ever happened, since the first overfull must have been a “false positive.”

Such false positives require this double checking to be done. They occur because the system of particles sometimes becomes overexcited. This overexcitement is caused by the repulsive collision detection method being used. This method sometimes allows particles to overlap too much, or get pushed through each other, because the time step is too large. When this happens the repulsive force becomes extremely large. This induces a large velocity and a large amount of motion in one time step. When this happens, a sharp jump in kinetic energy and a false overfull state is detected. The chance of this happening is small when there is a large amount of space for the particles to move and increases as the free space decreases. Or rather, the probability of this happening twice at the same number of particles is largest when the object is nearly full. So this apparent error works to correctly identify when the object is full. Figure 12 shows a cubic representation and the resulting representation after filling.

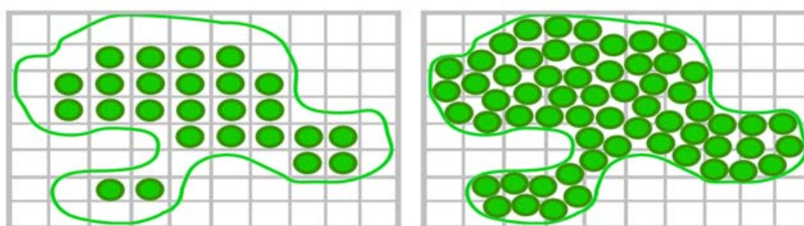


Figure 12: 2D result of filling a cubic representation.
On the left is a cubic representation. On the right is the result of a filling-over method.

Once an object is considered full, the particles are frozen in place. The neighbors of each particle are then determined and surface particles are identified and assigned normals. A two-pass version of the algorithm for the ***filling-over method*** is summarized in table 4.

Table 4: Pseudo-code for 2-pass filling-over method.

Name	FillingOver()

Pseudo-code	Set number of particles to zero.
	While (not over-full)
	    Add a particle to the system.
	    Update the system.
	End While
	Set Cap = Current Number of Particles minus 1.
	Reset Number of Particles to zero.
	While (number of particles < Cap)
	    Add a particle to the system.
	    Update the system.
	End While
	While (system not at rest)
	    Update the system.
	End While

3.8.3 Packing Density

Even though we are not directly interested in the maximal number of spheres that fit into a given object, a dense packing better represents the volume of the object. Because of this, it is necessary to determine how much the filling method improves the density of the packing.

In abstraction, particle systems have been used by mathematicians to determine the optimal packing of various objects with spheres (or circles in 2D). In fact, packing is a well known and mathematically explored problem. However in terminology, particle systems are rarely mentioned.

Historically the packing problem originated in the seventeenth century [64, 65]. This eventually led Kepler, in 1611, to suggest a face-centered cubic packing to be the most efficient way to pack spheres in the smallest amount of space. In 1998, Thomas Hales proved it. However, as the problem was unsolved for so long, and is mathematically interesting, a multitude of papers have been published on sphere packing [66-69]. It is also known, when properly stated, these problems are related to Voronoi diagrams and medial axis representations [70-72].

In sum, there are many approaches to achieving dense packings. Some methods pose the problem as an optimization problem [73]. However, in many cases the approach used is equivalent to growing spheres inside an object or shrinking the object around a set of spheres [67, 69, 74].

To analyze the efficiency of our filling method we examine the densities we achieve in the 2D case of packing circles into squares. This approach is chosen as the results of these packings have been mathematically proven and physically verified by others. The distinction between proven and verified being described as: "we know how many can fit, but we do not always know exactly how or where."

Using such a comparative analysis demonstrates the results are close to the "maximal" number of circles that can fit within the squares of various sizes, as documented in the papers by Peikert et al., Friedman, and Weisstein [66, 75, 76]. Notice in addition to verifying these numbers, an actual placement of circles within the square is determined.

Table 5: Results of filling tests.
Shown are the side lengths, *s*, of our test squares and the number, *N*, of spheres our algorithm placed in the square. The minimum known side length which can contain *N* is also given. Blank values indicate we could not achieve a side length which would contain *N* spheres.

s	N	s_{min}		s	N	s_{min}
4.05	4	4.00		7.20	11	7.03
4.85	5	4.83		----	12	7.15
5.40	6	5.33		7.60	13	7.47
5.80	7	5.74		7.85	14	7.74
----	8	5.87		----	15	7.87
6.05	9	6.00		8.05	16	8.00
6.90	10	6.75		8.95	17	8.54

The results of the filling method applied to several test cases are found in table 5. In these tests, the radius of each circle is fixed at one. Through multiple runs the side length, s, is determined. This side length is of the smallest square into which our method consistently fit N such circles. Also shown in the table is the proven minimum side length s_{min} (rounded to two decimal places) that allows the given N circles to fit in it. So all of our s values must be greater than or equal to the s_{min}. The closer they are the better the result.

Notice in several cases our algorithm does not fit the maximum number of circles in the given square size. For example, for all side lengths from 5.75 to 6.05 it consistently fits only 7 circles into the square. Similar difficulties arise for the $N = 12$ and 15 scenarios. However this is not too bad, for in performing one hundred test runs for any given s, roughly 95% returned the resulting number of circles used to fill the square as being two less, one less, or exactly the proven optimal number. In the remaining 5% of the runs the result was more than two under the minimum; in no case did it overestimate. From this it is reasonable to speculate that the filling algorithm obtains near optimal results for any object. This is further validated by visual inspection of the packings that result. Where possible, these form near hexagonal stacking patterns which are known to be maximal [77].

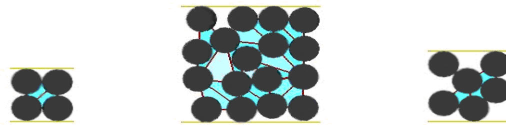


Figure 13: Packings of unit spheres for side length, s, of 4.05, 8.95, and 5.40 units.

In sum, we are able to obtain near optimal circle packings in cases where an optimal packing is known. Several such packings are shown in figure 13. This indicates our method should achieve good, dense, results at packing arbitrarily shaped objects (where computing optimal packings is not possible).

3.9 Grouping Particles

Using particles to represent everything in a modeling system quickly requires a large amount of memory. Further, a large number of particles requires a large number of updates to be performed. While parallel, or GPU, implementations may alleviate some of this burden, the problem of large numbers will still remain.

In our applications we use between ten and one hundred fifty thousand particles for solid objects, between twenty and two hundred thousand for gaseous objects and from thirty to one hundred thousand for liquid objects. These numbers, usually, keep our frame rates above one frame per second.

Table 6: Group frame rates.
Grouping of interior particles into larger particles provides a minimal increase in performance.

Number of Particles	Number of Groups	Group Size	Frames per Second
54 000 (30x30x30)x2	0	0	6
54 000	5940	3	6
54 000	1980	9	7
128 000 (40x40x40)x2	0	0	0.5
128 000	14080	3	0.5
128 000	4693	9	0.6
128 000	1564	27	0.7
250 000 (50x50x50)x2	0	0	0.05
250 000	27500	3	0.08
250 000	9167	9	0.09
250 000	3055	27	0.09

To alleviate some of the memory swapping issues, and reduce the calculations occasionally particles are grouped into sets. Thus a set of particles becomes one particle. This is primarily applied to the interior particles of solid objects, as they are the least dynamic. Because the number of particles in the test applications is small, this grouping methodology is not used often. However, in a limited set of test cases it improves the performance speeds, as measured in frames per second. This is illustrated in table 6. These cases model two solid, rigid objects and their interactions. No breaking or deforming occurs. One frame represents about 0.05 seconds of simulated time (i.e. step size is 0.05 seconds).

To group the particles, two simple algorithms (tables 7 and 8) are used. The minimum distance from surface particles is arbitrary. The larger the distance is, the fewer the number of grouped particles. The surface particles should not be grouped; they are needed to maintain a smooth surface and to allow deformations to occur if necessary. The interior particles can be grouped. They are only significant when an impact occurs. Notice the particles can be ungrouped during impact events, but this will slow the simulation if impacts frequently occur. This can be circumvented if the particles remain grouped except when impacts strong enough to break multiple connections occur.

Table 7: Description of the grouping algorithm.

Algorithm	Grouping Particles
Description	1. Consider the particles which compose a given solid. 2. Measure distances in terms of shortest path length between two particles. 3. Select a particle which is at least a distance of 3 from any surface particle. Give preference to one exactly 3 away from impact. 4. Select two neighbors of this particle, which are themselves neighbors. 4a. If unable to find two then select one. 4b. If unable to find one goto step 6. 5. Mark the connections between these three particles as joined and flag them as a grouped node. 6. Find an ungrouped particle at least a distance of 1 from one of the previous groups of particles and at least a distance of 3 from any surface particle. 7. Repeat from step 4 until step 6 fails to find such a particle.

Notice the grouping can also stop when a specific number of particles has been grouped. Our test cases used three. Thus a second run to grouped sets of nine particles and a third run to grouped sets of twenty-seven. Similar results can be achieved by directly adjusting the stopping number to the desired amount.

Table 8: Description of the graph creation algorithm.
This readjusts the solid object to include groups of particles, not just single particles.

Algorithm	Group Graph Creation
Description	1. Consider the set of grouped particles. 2. For each group of particles, set the group node to have the combined neighbors of the particles within the group. Note if a connection goes to a particle in another group make the connection go to that particle's group node, not to the particle. 3. Adjust the strengths and properties of the group connections to reflect any duplication of connections. 4. Add all ungrouped particles and their connections into the group graph.

Once the particles are grouped the objects must be redefined so they include the groups of particles rather than individual particles. This is accomplished using the algorithm described in table 8.

This grouping method is illustrated in 2D in figure 14. Notice the surface particles are first identified, then the interior particles are placed in groups of two or three. Once they are grouped the connectivity graph of the object is recalculated to include the groups instead of the individual particles.

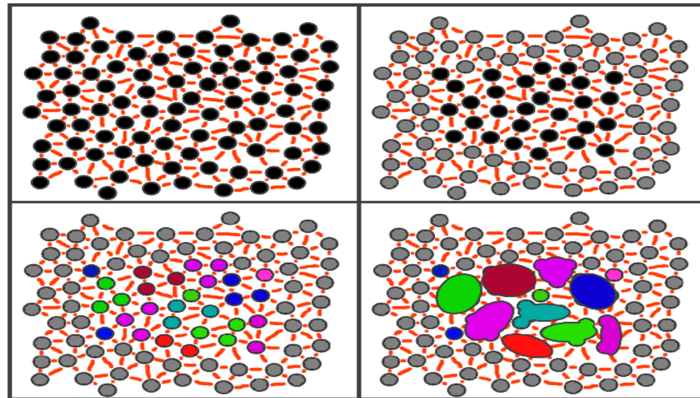


Figure 14: A demonstration of grouping internal particles.

3.10 Space Partitioning

A necessary component of our modeling framework is the efficient calculation and maintenance of particle neighbor relations. While this is easy to do for solid objects, gases and liquids are significantly more dynamic. To address this issue a grid based partitioning is applied to the entire modeling environment. There are many ways to implement such a grid. In this section focus is given to one which is useful when working with particles. It is presented here as no such description was found elsewhere. There are many other ways to improve this grid based scheme, but none are necessarily specific to the use of particles, which is the focus of this work.

Originally we believed a staggered grid representation of space would be most useful in application with particles. However, such application requires too much memory. Instead an overlapping grid partition is used. The effectiveness of this is inherent to the nature of using similar sized particles to model all objects. To illustrate this, consider the standard grid shown in figure 15, looking at cell[7].

1	2	3	4	5
6	7	8	9	10
11	12	13	14	15
16	17			
21				

Figure 15: A particle in a basic grid.
Assume the particle is in cell[7]. It would then need neighbor calculations done with every particle in cells: 1, 2, 3, 6, 7, 8, 11, 12, and 13. This an unnecessary number of calculations.

A particle in cell[7] may have neighbors in cell[7] and the eight adjacent cells denoted cell[1], [2], [3], [6], [8], [11], [12], and [13]. Assume each cell contains 100 particles. Then to calculate possible collisions for a given particle in cell[7] requires 99 + 8*100 comparisons, or rounded, 900 comparisons. This is clearly unnecessary in every case. Most particles are not near the edges, so the majority of particles only need to be compared with the 99 other particles in the same cell. Further, in the worst case of a particle being in a corner, say the upper right, it would only need to be compared to 99 + 3*100, or roughly to 400 other particles, and even that would be overkill. This number of calculations can be reduced.

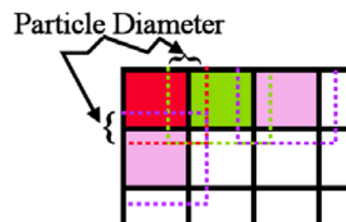


Figure 16: Grid cells overlap each other by the maximum diameter of all the particles.

Because every basic object is a particle, and all particles have about the same radius, the grid cells can overlap. Specifically, they can overlap by the maximum diameter of all the particles, as shown in figure 16. This overlap reduces the number of comparisons required. More accurately, particles in a given grid cell only need to be compared to all the particles within the same grid cell. The overlapping of the grids accounts for any particles near the boundaries. In 2D this results in almost eight times fewer comparisons.

There are several things to understand when this is done. The first is this technique is only useful if all the basic objects (particles) are small with respect to the cell sizes, and are all about the same size. The particle size relative to the cell size is important because if the overlap is too large then there is no reduction of the number of comparisons. Likewise, if the particles are not nearly all the same size, the overlap needs to be large enough to cover the largest object. For example, if the largest particle is ten times the size of the smallest, then the overlap causes the number of potential objects in a given cell to increase from n^3 to $(2*10 + n)^3$. Whereas, if all the particles are approximately the same size, it only increases to $(2 + n)^3$.

The second item of note is that particles may be contained within multiple cells. This means collision routines need to be certain not to have two objects in collision collide more than once. For example, suppose objects *A* and *B* are both in cells 1 and 2. Also suppose the collision detection routine discovers they are in collision when examining cell 1. Then the routine also discovers they are in collision when examining cell 2; however, the collision response should only be applied once. In most cases this is not difficult to achieve.

In sum, the overlapping grids function well with particle based objects, do not overly increase the memory requirements, and offer speed increases by reducing the number of distance comparisons. Because of this, most of our implementations have some form of overlapping grid partitioning for neighbor relations, collisions, surface display, and other calculations.

4. FORCE AND ACTION REPRESENTATION

Before discussing how to model force, it is necessary to define what force is. In the majority of this section the Newtonian definition of $\mathbf{F} = m\mathbf{a}$, is used. However, in implementation the rules applied to the particles are categorized into actions, where ***actions are anything capable of changing the state of a particle or particle's bindings***. This allows the changing of the color, shape, size, temperature or energy of a particle to be accomplished by applying an action. So Newtonian forces are one type of action in a rule system and an action is "that which brings change." While concentration is given to forces and force propagation, the methods also apply to actions and the propagation of actions.

In all particle systems each particle's motion is governed by a rule system. Usually these rules are defined to affect particles in a static volume of space (e.g. gravity). An exception to this appears to be found in flocking systems, where the motion of the flock is influenced by the motion of "leader" particles [20]. However, this is not an exception but a variation of abstraction and implementation. Specifically, these behavior rules are mobile regions associated with specific particles capable of changing the state of other particles. Further study of particle rule systems leads to a categorization of actions.

There are constant, static, actions that are always "on" and always influence the same region of space. Yet there also exist mobile, temporary, actions. These actions are applied conditionally and influence different regions of space. These temporary actions, or forces, include things such as gusts of wind, impact forces, vortices in a current, etc.

In this section focus is given to temporary forces because the constant forces are well described in other pieces of literature. The first topic is impact force (impulse) and how to divide it into two components. One component used for motion and the other for damage calculation. The second topic is how force damages particle connections. This leads to the third topic describing how impact forces propagate across connections. In these discussions action (force) particles are introduced. A brief presentation on how they

accomplish various temporary and event driven effects is offered. This section concludes with a description of gas particles and how, cumulatively, they establish pressure.

4.1 Impact Forces, Impulse Divided

In simulating dynamic environments, time is divided into time steps. The integration method, applied to model the motion of the objects, assumes the motion is continuous and differentiable. However, collisions cause instantaneous changes in velocity. This creates a non-differentiable path of motion. To overcome this, most existing modeling systems use an impulse force when dealing with collisions.

Impulses are not real. They are based on known results of physics. They simulate subtle properties of objects and their surfaces. In effect, they simulate a large force in a small time step, though when applied no time passes. They produce the desired, and expected, change in velocity at the cost of possibly having objects briefly overlap.

Significant to this work is the concept of a coefficient of restitution as used with impulse force calculations. The coefficient of restitution of a given object is a measured quantity and is a material property. It is primarily used to model the dissipation of energy during a collision. In modeling it is used to dampen impact results, thus producing inelastic collisions. For a collision with a stationary object the coefficient of restitution is defined as:

$$e = \mathrm{v}_{\text{final}} / \mathrm{v}_{\text{initial}}$$

From this we see if e is one then the collision is perfectly elastic, as the new velocity must equal the old. Likewise if e is zero then the collision is perfectly inelastic, as the object sticks to the whatever it hit.

If we consider two colliding objects, say particle A and particle B, we can calculate the normal, $\mathbf{n}$, of their separating plane [78]. If we denote the new velocity of particle A to be $\mathbf{v}^{A}_{new}$, the new velocity of particle B

to be $\mathbf{v}^{B}_{new}$, the old velocity of particle *A* as $\mathbf{v}^{A}_{old}$, and the old velocity of particle *B* as $\mathbf{v}^{B}_{old}$ then the coefficient of restitution, *e*, is defined such that:

$$(\mathbf{v}^{A}_{new} - \mathbf{v}^{B}_{new}) \cdot \mathbf{n} = -e\,(\mathbf{v}^{A}_{old} - \mathbf{v}^{B}_{old}) \cdot \mathbf{n}$$

We now denote an impulse force to be $\boldsymbol{j}$. Using a simple Eulerian update of $\mathbf{v}_{new} = \mathbf{v}_{old} + \mathbf{a}t$ and taking time, t, to be 1, we have:

$$\mathbf{v}^{A}_{new} = \mathbf{v}^{A}_{old} + (\boldsymbol{j} / m^{A})\mathbf{n}$$

and

$$\mathbf{v}^{B}_{new} = \mathbf{v}^{B}_{old} + (\boldsymbol{j} / m^{B})\mathbf{n}$$

Where m^{A} is the mass of particle *A* and m^{B} is the mass of particle *B*.

Using these three equations we derive by substitution:

$$\mathbf{v}^{A}_{old} + (\boldsymbol{j} / m^{A})\mathbf{n}\cdot\mathbf{n} - \mathbf{v}^{B}_{old} + (\boldsymbol{j} / m^{B})\mathbf{n}\cdot\mathbf{n} = -e(\mathbf{v}^{A}_{old} - \mathbf{v}^{B}_{old}) \cdot \mathbf{n}$$

And solve for $\boldsymbol{j}$:

$$\boldsymbol{j} = (m^{A}\, m^{B})(-e(\mathbf{v}^{A}_{old} - \mathbf{v}^{B}_{old}) \cdot \mathbf{n} - \mathbf{v}^{A}_{old} + \mathbf{v}^{B}_{old}) \,/\, (\,(\mathbf{n}\cdot\mathbf{n})\,(m^{A} + m^{B})\,)$$

This derivation leads to how an impact force can be divided into two components. One component is applied to the object as a normal rigid body transform. The other is used to damage the connections within the object. This division is based on the premise the coefficient of restitution models energy dissipation. This lost energy in the real world produces heat, sound, deforms the objects, damages the objects, etc. Thus in no way are is the modeling going beyond the realm of physics.

In every collision an impulse is calculated using a material's known *e* value. This result is denoted ***j1***. This force is then applied to the object to move it correctly based on the impact. In our framework, a second impulse, ***j2***, is also calculated with *e* equal to one. This determines how much force would be applied to the object if the collision were perfectly elastic.

Letting ***j3*** be the difference of ***j2*** minus ***j1***, the force used to break connections is calculated. This force is propagated through the object. It does not influence the motion of the object. Thus we have used the entirety of the impact force. Most of the impact energy moves the object. However, the energy normally dissipated by the coefficient of restitution is instead used to damage the internal structure of the object. This is significantly different than most modeling systems, yet consistent with physics.

4.2 Damaging Connections

To demonstrate how a force damages a connection between particles, assume a force is acting on particle *A* which is rigidly connected to particle *B*. The force acting on *A* is transmitted to particle *B* through the rigid connection, **R** = pos(*B*) – pos(*A*). Assume the angle between **F** and **R** is θ. Then the component forces exerted on *B* can be deduced from the projection of **F** onto **R**. Mathematically this is defined as:

$$\text{proj}_{\mathbf{R}}\,\mathbf{F} = \left(\frac{\mathbf{R}\cdot\mathbf{F}}{\|\mathbf{R}\|^2}\right)\mathbf{R}$$

And the magnitude of the projected force is thus:

$$\text{comp}_{\mathbf{R}}\,\mathbf{F} = \frac{\mathbf{R}\cdot\mathbf{F}}{\|\mathbf{R}\|}$$

Key in this analysis is the observation that the particles do NOT change position relative to one another (i.e. in the coordinate system where the object's center of mass is the origin, the relational position of *A* to *B* does not change). An explanation of why they do not change position can be found in numerous physics

books, for example the work presented in Marion and Thornton's book [79]. Further recall, as described in section 4.1, impact forces are divided into two components. One component is used for motion, the other for damaging the connections. Thus the force applied in this section is the portion of the total impact force already designated to damage the connections.

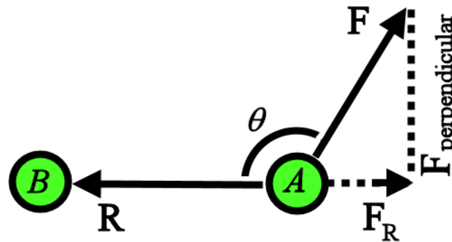


Figure 17: Perpendicular and parallel projection of impact force.
$F_{perpendicular}$ should induce a rotation around A and B's center of mass. It does not, so a counter force on B must prevent it. Thus, locally, there are two forces working in opposite directions. One at A and another at B. This strains (bends and tears) the connection between A and B.
F_r produces tearing or compression damage on the same connection.

Because the particles do not change relative position, the connection binding them must absorb any force that would cause them to move apart. This absorption is assumed to damage the connection. Note there are only two general directions the particles can move with respect to each other: together or apart. This means the connection may experience damage from compression or damage from tearing. To better understand this consider figure 17.

The connected particles do not rotate about their center of mass. This is stated in physics as the total internal torque between particles is zero. Thus the projection of the force onto a vector perpendicular to the connection, as shown in figure 17, is destructive to the connection. This is because the connection must be acting on *B* to prevent the particles from rotating about their center of mass. Such opposed forces strain the connection between the particles, as shown in figure 18.

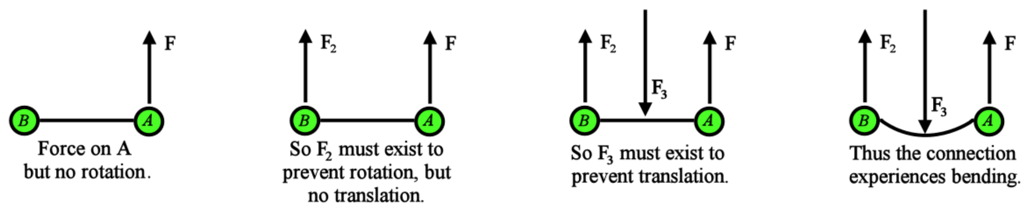


Figure 18: Lack of particle motion damages the connection.
This bending damage is similar to tearing.

We also see the force projected onto the vector parallel to the connection damages the connection (figure 17). Again, this damage results from the connection acting as a force on *B* to prevent the separation, or compression, of the two particles.

To better illustrate these damaging effects consider two people pulling on either of your arms. Both people pull in opposite directions, but are stuck together because you are holding onto them (you are the connection between them). As long as you have the strength to hold on, neither person will separate from the other. However, the more they pull, the more tired you become (the connection is damaged). Eventually you become too tired to maintain your grip, and let go (the connection breaks).

Having established both the perpendicular and parallel components of an impact force can damage the connections to *A*, we develop a representation to apply this damage. In this the projection of the impact force onto the direction of the connection reflects damage from compression when the angle between $\mathbf{F}$ and $\mathbf{R}$ is less than 90 degrees. The damage is tearing based when the angle is greater than 90 degrees. The perpendicular component of the impact force also induces tearing, or bending, damage on the connection.

Once the perpendicular component, $\mathbf{F_{perp}}$, and the parallel component $\mathbf{F_R}$ have been determined, any number of relations can be established to reflect the damage done on a connection by a given impact force. For our implementations we use the following to estimate such damage:

$$\text{base damage} = \alpha\,|\mathbf{F}_{\mathbf{perp}}| + \beta\,|\mathbf{F}_{\mathbf{R}}|$$

where α and β reflect the material properties of the object. To prevent compressional damage, β can be set to zero when $\mathbf{F}_{\mathbf{R}} \cdot \mathbf{R} > 0$. Likewise, the bending and tearing damage caused by the perpendicular component can be removed by setting α to zero.

Another factor influencing the amount of damage done to a connection is its distance, d, from the initial impact point. Obviously, the greater the distance, the less the damage. To achieve this a damage scaling constant, s, is used. In conjunction with s is a function, h, which determines the maximum distance an impact can travel through a given object. This function is used to reflect material properties of an object, such as how well it dampens internal vibrations. The function, h, is also dependent on the magnitude of the original impact force. This h function is only used for wave based propagations. With force particles it is obfuscated, as the strength of the force particle is instead reduced as it moves. Details of each propagation method are given later, however both are equivalent in form. So the scaling function is:

$$s = (1 - d / h), \text{ for } d \le h.$$

Thus the damage for a given connection is:

$$\text{damage} = s * (\alpha\,|\mathbf{F}_{\mathbf{perp}}| + \beta\,|\mathbf{F}_{\mathbf{R}}|)$$

In sum, to allow an object to fracture and break into pieces, the connections between particles of the object are given strength ratings. These connections suffer damage as forces act on their connected particles. The extent of damage suffered by a given connection depends on the direction of the force. Based on the angle between the force direction and the connection, the damage is scaled. This scaling reflects how much of the force is compressing the connection and how much is tearing the connection. The damage is also

scaled based on the distance the initial impact is from the given connection. More on this is presented in section 4.3.2.

4.3 Wave Propagation

Force may be propagated in various ways. When dealing with solids represented as small pieces of a larger whole, the force is usually transmitted from one piece to the next. This technique describes the process used in finite differencing methods (FDMs) and finite element methods (FEMs). Examples of such methods can be found in the work of others [49, 50].

We use a different idea, based on wave propagation. The distinction of this method is the force does not travel from one piece of the object to the next along a connection. Instead, it washes over the entire object.

Force waves, in general, have been well studied in physics [80-82] and have been used in computer graphics [53, 54, 83]. While wave propagation is an exceptionally interesting and diverse topic area, we only consider basic properties. In this we emphasize that our use of waves is in a simplified and approximate form. However, conceptually, the application of such can be extended to be significantly more complex and accurate.

4.3.1 Blast Waves

Blast waves are generated by explosive detonations. A detailed discussion of this process can be found in the theses by Neff [84] or by Roach [83], as well as in shorter articles [53, 54, 85]. The related physics may be investigated in various books and articles [80-82]. Presented here is a summary of the process. This is offered to demonstrate it can be applied to volpar systems and to set the stage for the development of impact waves.

In sum, blast waves can be used with voxels to model the breaking of objects as if they have been hit by an explosive force [53]. The voxels are connected through shared faces (adjacency). The damage incurred by

their connections is based on the shared area of their faces and a modified Friedlander equation describing the pressure force. This is shown in figure 19.

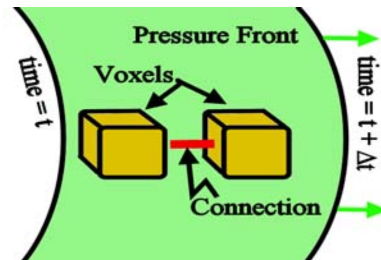


Figure 19: Pressure front passing over two connected voxels.
Applying Pressure = Force / Area, it is easy to see the force experienced between two connected voxels can be calculated using the Friedlander equation and the area shared between the voxels.

The specific form of the modified Friedlander equation is:

$$p(t) = p_0 + P(1 - t/T)e^{-bt/T}$$

where p_0 is the initial (ambient) pressure, P is the increase in pressure at the time of the shock front's arrival, t_a is the time of the shock front's arrival, and T is the time for the pressure to first return to p_0. The constant b allows further adjustment to the curvature and duration of the effect. In circumstances where accuracy is important, the variables: b, P and p_0 are based on experimental results measured from TNT explosions.

This modeled pressure, of course, is based on the shock wave front. It focuses on the pressure changes induced on an object upon a wave's arrival. This is best described by the illustration shown in figure 20, which graphs the pressure profile of a blast wave at a given point. This graph is based on a modified Friedlander equation, which is known to accurately represent the pressure curve [81].

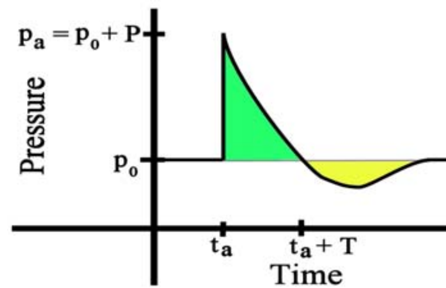


Figure 20: Pressure curve of a blast wave using a modified Friedlander equation. Here p_0 is the initial pressure, t_a is the time of the shock front's arrival, P is the pressure increase at the t_a, and T is the time for the pressure to first return to p_0.

The significance of this blast wave modeling is that the pressure wave breaks connections between voxels as the wave passes over the connections. This is an efficient force propagation method because the force applied to the connections is not transmitted from one element to another. Instead the pressure washes over each connection. As the wave encompasses each connection, a damaging force is applied to the connection. This eliminates the need for a system of complex equations (e.g. system of spring equations) to propagate damaging force.

4.3.2 Impact Waves

An impact force can be transmitted across an object's connections just as an explosion's shock wave front pressure can be. However to accomplish this, a description of the force carried with the impact wave must be defined and accurately reflect the impact's capability to do damage.

The majority of this definition is found in sections 4.1 and 4.2. In section 4.1 the impact force is divided into two components. The portion of interest here is the portion which damages the connections. Section 4.2 describes how that force damages connections based on distances and relative angles. However, these descriptions are not complete as the force experienced also depends on how many connections have been encountered and how many have been broken by the impact wave before it encountered the current

connection. This behavior is modeled within the wave's propagation. Specifically, the force contained by the wave is reduced each time it encounters a connection and each time it breaks a connection.

For example, assume a wave has an initial magnitude of F_0. Also assume it travels five units in a given time step. During this time step it encounters three connections. Each connection experiences a force of $s_i F_0$, where s_i is the scaling factor for the ith connection. The new magnitude of the force becomes F_1, where $F_1 = F_0 - 0.1 s_1 F_0 - 0.1\, s_2 F_0 - s_3 F_0$. Assuming F_1 is greater than zero the propagation continues.

In the next time step the wave encounters only one connection, but breaks it. Assume the broken connection has a maximum strength of q, then the wave's force is reduced by $0.4q$:

$$F_2 = F_1 - 0.4q$$

This reflects the amount of energy needed to break the connection apart. This behavior is similar to that of the specific heat of water (i.e. more energy is needed to transform ice at 0° C into water at 0° C).

The percentage of force lost when encountering a connection and breaking a connection is yet another parameter of the material. Stronger or more elastic materials may use up more energy as the impact travels through the object. The above usage of 0.1 and 0.4 are merely examples.

4.4 Action and Force Particles

In this section action and force particles are introduced. The description and usage of action particles in a volpar system is unique to this dissertation. Action particles are defined as ***invisible particles with an associated region of influence.*** Force particles are a subclass of action particles, specifically related to Newtonian based forces.

This section begins by describing one of the motivations, related to wave propagation, that led to using force particles. We then describe the full breadth of use action particles have. To demonstrate this breadth basic examples are presented in this section. In section 6, much richer examples are presented. Those found here focus on action particles. Those in section 6 demonstrate their use within volpar systems.

4.4.1 Motivation from Wave Propagation

The motivation for using particles to represent force came from the development of impact wave propagation. During this process the breaking patterns observed were too global – damage was appearing too far from the actual point of collision. The cause of this odd visual behavior is the global effect of the propagation. The collision force is propagated through the entire object, and if any connections are excessively weak, they break first. From a visual, “cause and effect” perspective this can appear wrong (figure 21). To correct this behavior a method to localize, or at least better direct, the impact propagation is needed.

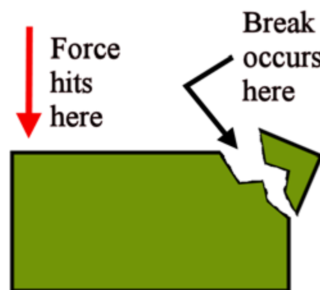


Figure 21: Visually odd cause and effect.

One way to localize the wave’s effect is to limit it to a cone shaped volume, where the center of the cone is in line with the direction of the collision’s impact. This works, but is difficult to implement and involves timely calculations to determine which connections are affected. It also is not as “particle-based” as we desire. However, such a cone-like wave front can be represented using particles. This is accomplished in, at least, three ways as shown in figure 22.

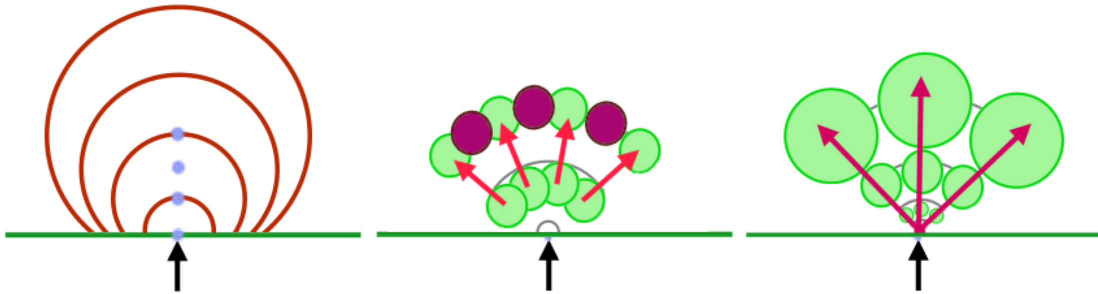


Figure 22: Particles representing a conic wave front.
On the left, one particle with an expanding radius is used. In the middle, particles split as they become more distance from each other. On the right, particles with expanding radii are used.

4.4.2 Definition Expansion

Action, and force, particles are invisible particles with an associated region of influence, or force field. Contained within that brief definition is significantly more information.

The more complete definition is that they are invisible particles capable of accomplishing a variety of effects. These particles, like all others, have a position in world space. These particles do not collide with any other particles and may or may not be affected by other forces. They may or may not influence each other. Force particles may move through space or be stationary. Their motion may be independently driven or linked to another particle to move with it through space.

Action particles may have lifespans, so they can be born and die. In most situations only a small number of action particles are alive at the same time.

Generators of action particles may be user defined, linked to specific types of world forces, linked to objects, or be spontaneously created as the result of an event. In the case of spontaneous creation, the event may be specified by the user or defined within the modeling engine. For example, turning on a fan may produce a temporary force particle generator.

Action particles may also be coupled with each other. Thus, as mass particles form volumes of mass (objects), action particles form volumes of action (action objects). The interactions of such coupled action particles may be quite complex and time oriented. The coupling may be viewed as a force effect, or constraint, induced by each particle on the other coupled particles.

It is also possible to create mass particles with action particle properties. Another possibility is to use action particles to implement localized, moving, grid-based (Eulerian or semi-Lagrangian) effects within the particle's region of influence. Within this definition a variety of effects, many previously thought of as special cases, is described.

In sum, action particles allow a large number of modeling effects to be achieved. The significance of this is the abstraction encapsulation of such effects and the ease of implementation this allows (e.g. object oriented inheritance and expansion).

4.4.3 Basic Examples

The simplest design of an action particle can be seen in the guiding nature of "leader" particles in flocking systems. However, our intent for using action particles goes beyond this. They may be used for other effects and may be quite complicated. In almost all circumstances, they have a range of effect.

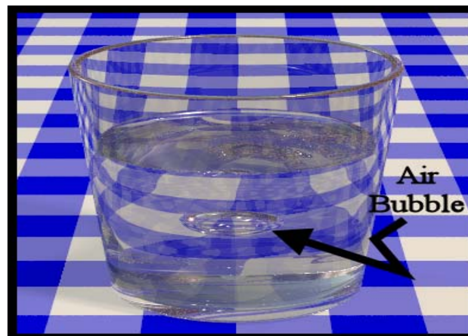


Figure 23: Bubble of air generated by a force particle displacing water particles.

By using action and force particles it is possible to create a plethora of unique effects which may be artistically or realistically motivated. One such effect is shown in figure 23. In this example an action particle is simulating a bubble of water rising through a massive number of liquid particles. Such an effect may be needed if the water is to boil.

The extensibility of using action particles becomes obvious given the ***definition of action as: anything that changes the state of a particle or particle's bindings***. This allows action particles to represent the transmission of such things as heat energy, as shown in figure 24. They may also be used to temporarily increase or decrease the friction of a surface. For example, if a substance is spilled on a surface, force particles may be spread with the substance to alter the frictional force experienced by objects touching that surface. This effect remains, even after the spilled substance vanishes. Thus modeling the nature of substances, such as evaporated sugar water or oil residues. Action particles also allow event driven effects to occur. These include things such as bursts of wind when an object moves or the generation of a whirlpool after an object sinks into a liquid.

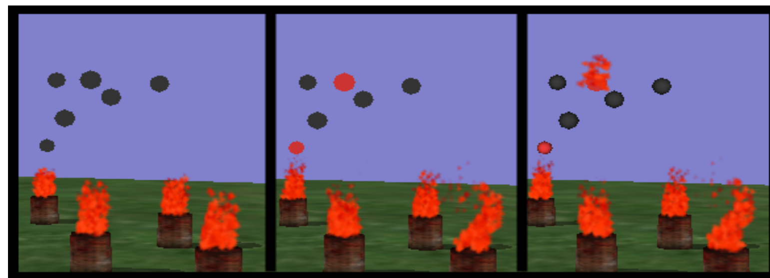


Figure 24: Heat transfer from fire using action particles.
Invisible heat particles transmit heat from the fire. Thus the flammable balls hanging above the fires burst into flames when sufficiently warmed (though never directly touched by flames).

Action particles also offer advantages to other modeling systems. For example, as they can model force transmission through non-represented media and can be generated automatically, it is possible to use them in a boundary surface representation (B-rep) based environment to model force propagation. As B-rep

objects are hollow there is no obvious way to transmit an impact force from one side of an object to the other. Yet, this can be accomplished using force particles, as shown in figure 25. This is done by making some assumptions regarding the interior of the B-rep object, such as homogeneity and applying the force particles in an appropriate fashion.

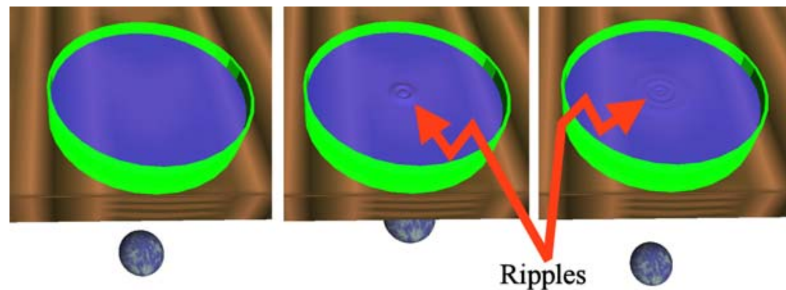


Figure 25: Force particles cause ripples to form in a pool of water.
Even though the table is a B-rep object, when the ball-hammer strikes the underside of the table, force particles are still able to transmit force through it to cause the surface of the water to ripple.

While the application of force particles to B-rep objects is interesting, and important, we are more interested in their usage with volumetric particle objects. To continue demonstrating their abstraction ability, several more detailed examples of implementations are found in the next sub-sections.

The first example demonstrates using force particles to model the swirling behavior of smoke. The second shows the modeling of forces through non-represented media. The third example illustrates coupled force particles. Each of these examples are demonstrations of the extensibility of action particles. Individually they are not dramatically significant, but collectively they illustrate the power of action particle representations. Further examples are given in section 6.

In the end, the interaction between action, and force, particles allows very complicated systems of actions to develop from very basic representations. These complex systems are easily computed, as action effects are added together, and the results appear very natural. Further, the action particles offer an intuitive and

obvious means for a user to design actions and encapsulate the ideas and methodologies of others into one modeling system. Thus action particles unify a variety of special cases and ad-hoc practices. They also serve to better describe how apparently non-related particle based systems, such as flocking and water simulation, relate to each other conceptually in action and force abstraction.

In sum, the use of action particles extends beyond our motivated need to propagate force through solids. The following examples demonstrate such extension.

4.4.4 Vortex Effects

To illustrate the usefulness of force particles consider a set of particles designed to represent smoke. Smoke billows and swirls seemingly at random. This is difficult to achieve using only predefined forces because the random appearance and motion of the swirling areas is difficult to explicitly model. In scenarios such as this, force particles are very useful. To accomplish the desired effect, assume there is a continuous stream of smoke being pushed by a global wind force (a current). With nothing more, this appears as a stream of particles moving in one direction, as seen in figure 26.

Figure 26: Smoke particles where only a linear wind current is applied.

To make the smoke swirl, vortex-force particles are born and pushed by the wind. These particles affect each other and have variable lifespans. They are generated across random time intervals at the same location as the smoke particle emitter. The region and magnitude of effect that each vortex-force particle exhibits is random. The direction of rotation they induce is likewise random. The results of such an implementation are shown in figure 27.

Figure 27: Smoke particles influenced by a linear current and vortex force particles.

In 2D smoke implementations, such as that pictured in figure 27, the vortex particles are given a cylindrical volume of influence. The cylinder is aligned with one of the major coordinate axes. For this example it is aligned with the positive z-axis and the spin is counter-clockwise. With this arrangement, the velocity induced by the vortex at coordinate (x, y) is in the direction of the tangent line at that point. Thus if a point is on the unit circle with (x, y) = (cos(θ), sin(θ)), then the tangential velocity, is (-sin(θ), cos(θ)), or simply (-y, x).

Letting r be the radius of the circular base of the cylinder and v be the maximum velocity of the vortex, allows the velocity of the vortex to be scaled based on how close a point is to the center of the circle. Assuming the point is at (x, y), then its scaled velocity is ($-vy/r$, vx/r). Note this assumes the center of the base is at (0,0) and induces a faster spin velocity at the outer portions of the vortex. To summarize:

Vortex Center	= Force Particle's Position = (c_x, c_y)	
Mass Particle's Global Position	= (m_x, m_y)	
Mass Particle's Position in Field	= ($m_x - c_x$, $m_y - c_y$)	= (x, y)
Target Velocity of Mass Particle	= ($-vy/r$, vx/r)	

The term *target velocity* is used to indicate this velocity may be applied in conjunction with any other velocity force fields in which the particle is contained. The net sum of such velocity effects is then scaled

by the size of the current time step and applied to the particle's current velocity. Thus the particle may not actually achieve the field's velocity and so the term *target velocity* is used.

In sum, this example demonstrates how force particles introduced into a simple global field induce complicated motion. This motion is difficult to explicitly model, however, using this emergent methodology it becomes almost trivial to implement and remains visually plausible. While this example is in 2D it extends in an obvious fashion to 3D. As such, it is implemented in other examples.

4.4.5 Event Driven, Cross-Media Effects

Another effect achievable using force particles is the transmission of currents through substances such as water or air. Force particles offer an advantage in modeling this effect because they are automatically generated and do not require the transmission media to be modeled explicitly. To demonstrate this, several examples of a fan blowing various materials through the air are presented. The key feature within this section is the transmission media, the air, is not being represented. So force particles allow two objects to influence each other without sharing a physical connection and this interaction is event driven and not continuous.

Figure 28: Fan blowing confetti via wind force particles.
Wind force particles are shown as wire spheres. The fan is the small rectangle on the left. The confetti pieces are blown upward and right. Pieces beyond the range of the force particles fall straight down.

The first example is a fan blowing pieces of confetti. The image in figure 28 is captured from the simulation. Notice this is intended as a proof of concept, so little concentration is given to the visual aspects of the environment (i.e. no shadows, no textures, etc).

During the simulation, the user is able to turn the fan off or on. When the fan is on, its blades generate wind force particles. These particles move away from the fan. When they encounter pieces of confetti, the pieces are lifted and carried with the wind force particles.

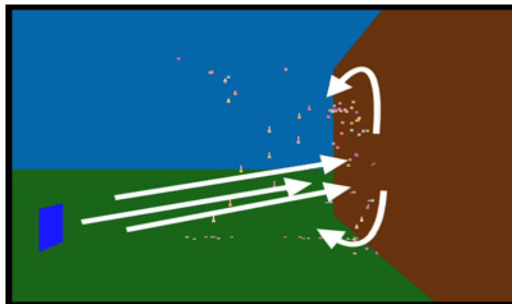


Figure 29: Confetti blown by force particles in a circular pattern against a wall.

The second example is also a proof of concept implementation, the results are shown in figure 29. In this example the force particles are given the ability to bounce off a wall, thus the their direction and the direction of their force changes. This allows effects such as wind hitting the floor and blowing up a wall to be created and produces a swirling motion.

The wind blowing up the wall is achieved by altering the force particles angle of reflection. In other words, the bounce of the force particles is not the traditional bounce of “angle of incidence equals angle of reflection.” Instead the bounce is altered so the angle of reflection is reduced based on the amount of wind force the particle is exhibiting. For example the following may be used:

$$\beta = Random(1 - V/V_{max}) * \alpha$$

Where β is the resulting angle of reflection, α is the angle of incidence, V is the particle's target wind velocity, V_{max} is the maximum allowed wind velocity, and the function *Random*(k) returns a random number, normally distributed, from zero to k. This equation and these settings are sufficient for our purpose of simple demonstration. To better model such forces a fuller development of wind velocities and effects is required. While not related to force particles, the work of Shinya and Fournier [86] or that of Sakas and Westermann [87] might prove useful in better developing the model for wind.

Continuing with force particles, as the particle's direction of movement is altered, its direction of wind force also changes. This, with their modified bounce directions, gives the wind force particles a tendency to crawl along surfaces. This is further amplified by other incoming force particles pushing them against the surfaces.

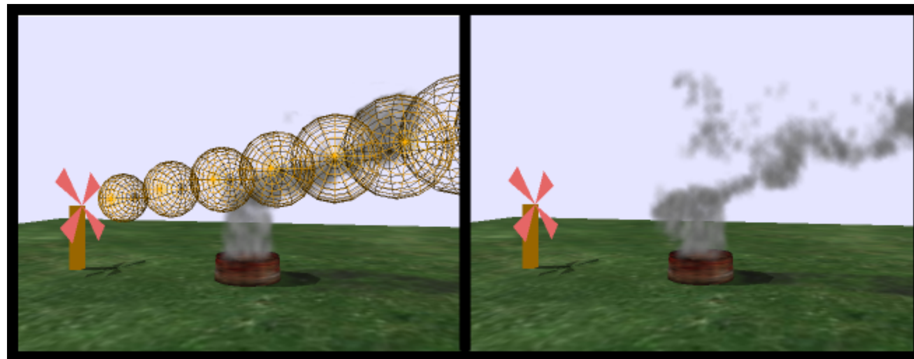


Figure 30: Smoke, wind, fan and force particles example.
A fan, when turned on, generates wind-force particles, visualized in the left image as wire frame spheres made invisible in the right image. The smoke is also subject to 3D vortex-force particles.

The third example models smoke emanating from a fire pit with a fan blowing nearby (figure 30). When the fan is switched on the smoke is blown to the side. When the fan is switched off the motion of the smoke returns to an upward direction. This example also uses vortex particles within the smoke. This is an extension of the 2D effect discussed in section 4.4.4.

Three examples of wind force particles have been presented. In these examples are three significant features. The first is the event driven nature of the particles which allows for user controlled and environment responsive events. The second is the transmission of force across a non-represented media. Thus objects may influence objects at a distance without requiring a chain of object connections. The third feature is the redirection capabilities of the force particles which allow the effect of the particles to be partially dependent on the environment. This makes the nature of the force particles adaptive and not entirely predefined or hard-coded.

4.4.6 Coupled Force Particle Systems

So far it has been shown force particles can be used to influence other force fields and can be used to transmit force through non-represented media in an event driven manner. In this section force particles are coupled together to create a dynamically changing region of influence. Such modeling is necessary when the boundary of influence of a force is not clearly defined, is changing, or cannot be discretely modeled by the position of a single point in space. By coupling force particles, they combine their effect to produce a shape beyond simple spheres, yet they are still allowed to move independently. In the example below a spline based on the particle locations defines a region of influence. To maintain a reasonable spline-funnel the particles are constrained to stay within a defined proximity of each other.

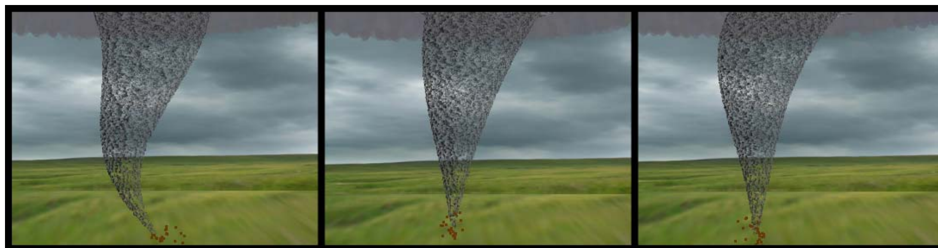


Figure 31: A tornado modeled using a coupled force particle system.
Note how the volume of influence changes.

To demonstrate a coupled force particle system a tornado is modeled, the results are shown in figure 31. In this system a set of three force particles define a cubic spline. Each particle is given a height above ground

level which does not change. Each particle is also given a maximum distance it can travel from the other two particles. A force field is defined by the three particles. The center of this field is the spline curve described by the position of the force particles, it is the center line of the tornado. The force experienced by a particle within the field is similar to the vortex field described in section 4.4.4, where the radius of influence is a function of height. Into this field are placed a large number of visible dust particles. Thus if a dust particle at height h, is within distance r(h) of the curve, then the particle is subjected to a force which will cause it to spin around the curve. These dust particles visually represent our tornado and are not allowed to leave the volume of influence. Note, however, as the tornado moves, it is not limited to affecting just the dust particles, but can also affect other objects.

In general terms this setup is expressed as follows:

Position of top force particle = (0, CloudHeight, 0)

Position of middle force particle = (0, 0.5 * (CloudHeight – GroundHeight), 0)

Position of bottom force particle = (0, GroundHeight, 0)

$r = \mathrm{r}(h) = (5/3) * h$

Target velocity of affected particle = $(-vz/r, 0, vx/r)$,

Where (x, h, z) is the affected particle's position, relative to the curve's coordinates of (cx, h, cz).

Key in this coupling of force particles is the fact the particles can move, which in turn moves the tornado. This is denoted by:

Velocity of top force particle = (vtx, vty, vtz)

Velocity of top force particle = (vmx, vmy, vmz)

Velocity of top force particle = (vbx, vby, vbz)

Of course the distance between the force particles is constrained by a maximum distance to keep the funnel looking like a tornado. For example the topmost particle is limited by applying an algorithm similar to the one in table 9.

Table 9: Pseudo-code for constraining force particle location.

Algorithm Name	Position Constraint

Pseudo-code	NewPositionTop = (position + (*vtx*, *vty*, *vtz*) * Δ*t*)
	NewPositionMid = (position + (*vmx*, *vmy*, *vmz*) * Δ*t*)
	while (distance(NewPositionTop, newPositionMid) > MaxDist)
	{
	    Δ*t* = Δ*t* * 0.9
	    NewPositionTop = (position + (*vtx*, *vty*, *vtz*) * Δ*t*)
	    NewPositionMid = (position + (*vmx*, *vmy*, *vmz*) * Δ*t*)
	}

While this is a simple example, it can be expanded to achieve more realistic behavior. However, our intent is to show the coupling of three force particles can be used to define a dynamically changing region of influence. Specifically in this example, each particle is experiencing a random force in the xy-plane. The total effect these random forces have on the shape is constrained by the maximum distance the particles are allowed to separate from each other. Thus, in this section, it has been shown coupled force particles can be used to model regions of dynamically changing influence.

4.5 Gas Pressure

Force particles are useful for many things. However, in modeling a gas, the gas itself should possess the ability to apply pressure to other objects. How this is accomplished using gas particles is briefly mentioned in section 3.5. A more thorough description is offered here.

As stated previously every gas particle has a velocity. These velocities reflect the ambient temperature of the particles' environment. Gas particles also experience perfectly elastic collisions with solid and liquid particles. However, even though they are moving and collide with solids and liquids, they are not very

massive. So a single gas particle cannot move a solid or liquid object. However, a large number of gas particles can.

This is possible because when a gas particle collides with a solid (or liquid) particle, the gas particle is allowed to push against the solid. This alters the solid's velocity a very small amount. Most of the time, this alteration is not noticeable. There is an oddity in this collision. As stated, collisions from the gas particle's perspective are perfectly elastic, so the gas particle does not lose any speed. Technically this is inaccurate. However, this is justified by assuming the velocity lost by the gas particle is instantaneously regained due to the ambient temperature. If this is not true, the gas particles would not be moving.

Because of the velocity of the gas particles, and their collision behavior with solids and liquids it is possible for a sufficient number of gas particles to create a pressure (force) on the surface of a solid or liquid. In most cases this pressure does not produce any observable effects. However, there are circumstances when gas pressure can be observed. For example, if a large number of gas particles are placed in an elastic solid, something like a balloon, they can inflate the solid. This is accomplished by increasing the temperature within the balloon. This increases the velocity of the gas particles. The increased velocities increase the frequency of collisions with the balloon's interior surface and increase the "push" the gas particles exert on the solid. This causes the balloon to inflate. A similar effect can be achieved by introducing more gas particles into the balloon.

Pressure effects can also be observed in a liquid environment. For this to happen, the liquid must be sticky enough (viscous enough) to prevent the gas particles from escaping the liquid. Effectively the liquid must be able to trap a sufficient number of gas particles within its interior to create a situation where the gas particles push the liquid particles out and form a bubble within the liquid.

In sum, gas particles create a pressure force on the surfaces of solids and liquids when they are confined by such objects. If the temperature of the gas increases then the pressure increases. If the quantity of gas

increases and the volume remains the same then the pressure increases. Likewise if the volume is decreased then the pressure exerted by the particles will increase. This behavior is consistent with physical behavior. However, to achieve a pressure force requires a significant number of gas particles and to observe the pressure effect requires specific scenarios to be modeled.

5. OBJECT DISPLAY

In this section we present a surfel based method for displaying volpar objects. This is a reapplication of existing methods [39, 40, 88]. This adaptation is not as trivial as it may seem, but at the same time is not dramatically surprising. It is unique in that it must accurately determine which particles are part of the object's surface, the phase state of a particle, the object to which it belongs, its changing orientation, and its visible properties. The advantage of using surfels with volpars is the technique can be applied for any type of object as the display reduces to displaying the particles, not the objects.

We first discuss the display of visible gas particles. We then describe how solids and liquids are displayed using surfel based methods. We conclude with other display options and various reasons why different display methods may better accommodate different situations.

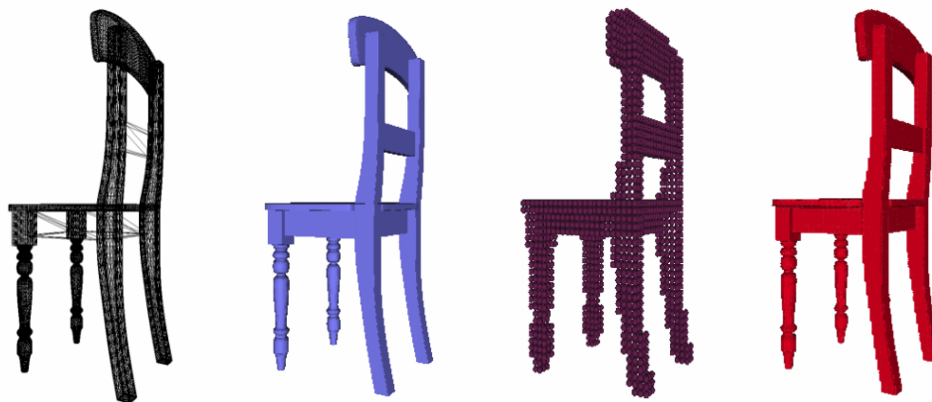


Figure 32: Wireframe, triangulated surface, particle spheres and surfel display of a chair.

To set the mood of this discussion there are four renderings of the same chair shown in figure 32. The first two images are rendered using common techniques, the third is a spherical display, and the fourth is the result of a surfel based approach. The latter two are using an underlying volpar representation.

5.1 Billboarding

Visible gas particles, such as smoke particles, are displayed using a billboarding method. For example, they can be displayed using a shaded, or textured square which is oriented to always face the viewer. These squares are then blended together to create the illusion of transparency and density.

While billboarding works well for objects like smoke, it is not very useful in the case of solids. For solids and liquids techniques such as marching cubes, iso-surfaces or level set methods can be used [89-91]. These methods are effective, but do not capture the essence of a strictly particle based system.

In sum, billboarding methods are used to display visible gas objects. They can be used to display liquids, but do not always achieve the desired result. In terms of solids, billboarding fails to achieve good results.

5.2 Surfels

To display a solid (and sometimes liquid) object it is necessary to determine where the surface of the object is. To accomplish this requires a method to identify surface particles of objects. Also needed is a method to find the nearest neighbors of a given particle. Such methods can be constructed using an overlapping grid as described in section 3.10. The neighbor identification is straightforward from the discussion in that section. The identification of surface particles is done using the same overlapping grid in conjunction with the surface identification method as described in section 3.8.1 (where it is more a byproduct of the grid based filling).

The following subsections present the methods necessary to apply a surfel based display to volpar objects. In sum the idea is simple. Once the surface particles are identified, a surface normal is calculated for each. With this information each solid object is displayed as a set of overlapping blended circles. This is very appropriate, and works well, within our volpar framework. The details of implementing this display, once the surface particle information is available can be found in various works [39, 40, 88].

5.2.1 Breakable Surface Identification

In the identification of surface particles a complication may occur when breakable objects are present. The difficulty appears when an object partially fractures. For example consider figure 33, where there is a fracture line of broken connections, but cells on either side of the fracture still contain particles. For the fracture line to be visible, those particles must be identified as surface particles. Fortunately, a mechanism is already in place to assist in this process.

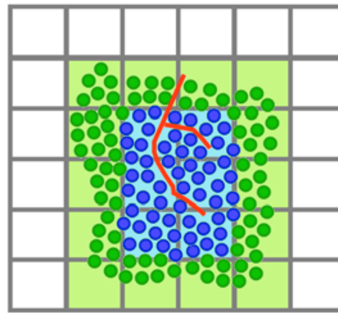


Figure 33: A solid object partially fractured.
The red line denotes the broken connections between particles. So particles immediately adjacent to the red line should be surface particles, but cannot be flagged as such based on the grid cells.

As described in section 3.7 solid particles within an object form a connected graph. Thus each solid particle knows to what object it belongs. This knowledge is in the form of an index number referring to the object. This indexing is updated whenever a connection breaks. Because it is maintained throughout the simulation this indexing allows particles on fracture surfaces to be identified.

Such identification is achieved by examining every broken connection. If the particles of a broken connection still have the same object index then both particles are flagged as surface particles. This is because their broken connection is part of a partial fracture. A complete fracture causes the particles to be in two different objects (i.e. different object indices).

5.2.2 Calculating Surface Normals

Once the surface particles are identified the surface normal for each surface particle must be calculated. For gas particles, where every particle is a surface particle, this normal points at the viewer. However for liquids and solids a more definitive surface is required and more calculations are performed. To calculate the surface normals for liquid and solid particles a method based on a moving least squares (MLS) algorithm is used [45, 92, 93, 94]. Previously, MLS methods have been used to approximate normals and display surfaces for point cloud data [95]. In a few cases they have been used to calculate normals for surface based deformable objects [96]. In few, if any, of these cases are they used in a physically dynamic modeling environment. More specifically, in each case the objects being modeled are singular in their worlds and are not moving. In the case of deformable objects the deformations are caused by the user in an effort to sculpt or mold the object into a desired shape.

A general description of our MLS based algorithm is found in section 5.2.2.1. More general information on least square methods can be found in Appendix D. A point worth noting in this process is discussed in section 5.2.2.2 where the volumetric representation of an object is used to identify the interior of the object. This makes the determination of the outward pointing normal easier, which is a known issue when dealing with surface-point only representations [97].

5.2.2.1 Moving Least Squares

In this section the MLS algorithm used in our implementations is described. We assume methods for identifying surface particles are in hand. We further assume a surface particle, p, has been selected for processing. With this particle we associate up to n nearest particles. These n particles are called the neighbors of the given particle. These neighbors, and p itself, are referred to as the neighborhood of p, denoted Nhbrhd(p).

The choice of n neighbors is arbitrary. In implementation we use between ten and fifty neighbors. In cases where a particle has fewer than ten identifiable neighbors, such as in areas of high curvature, errors in the

normal calculations become obvious. To correct this problem various re-sampling techniques may be used, however if there is only one particle, then there is still only one oriented circle. While this can be a problem, blending of the high curvature area minimizes the visibility of this error. Further, the motion of the object decreases the visibility of the error.

Table 10: Algorithm to calculate normals of surface particles.

Algorithm	Calculate Surface Normals
Description	For i = 1 to [number of surface particles] Let p[i] denote the ith surface particle. Let N[i] denote the k nearest neighbors of p[i] ($k \le 30$). Let Nhbrhd(p[i]) = p[i] $\cup$ N[i]. Let **o**[i] denote the center of Nhbrhd(p[i]). Let **n**[i] denote the normal of a tangent plane TP(p[i]). Calculate **n**[i] such that the tangent plane, TP(p[i]), is the least squares best fitting plane to Nhbrhd(p[i]). End for i.

Once the neighbors of each surface particle have been determined, a MLS based algorithm [98] is used to calculate the surface normal for each surface particle. The algorithm used to determine the normals for each particle is described in table 10. The more intricate details are described shortly.

To calculate each normal, **n**[i], the covariance matrix of Nhbrhd(p[i]) is created. This is a 3x3, positive, semi-definite matrix defined as:

$$\mathrm{CV}[i] = \sum_{\mathbf{q} \in \mathrm{Nhbrhd}(\mathbf{p}[i])} (\mathbf{q} - \mathbf{o}[i]) \otimes (\mathbf{q} - \mathbf{o}[i])$$

Where $\otimes$ denotes the outer product vector operator. So, if the vectors **a** and **b** have components a_i and b_j respectively then the matrix $\mathbf{a} \otimes \mathbf{b}$ has $a_i b_j$ as its $(j, i)^{th}$ entry. For example if $\mathbf{a} = [1\ 2\ 3]^T$ and $\mathbf{b} = [4\ 5\ 6]^T$, then

$$\mathbf{a} \otimes \mathbf{b} = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \otimes \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} = \begin{bmatrix} 1*4 & 2*4 & 3*4 \\ 1*5 & 2*5 & 3*5 \\ 1*6 & 2*6 & 3*6 \end{bmatrix}$$

Once the covariance matrix is calculated its eigenvalues and eigenvectors are calculated. The eigenvalues are denoted λ_1, λ_2, λ_3, with corresponding eigenvectors $\mathbf{v_1}$, $\mathbf{v_2}$, $\mathbf{v_3}$. The eigenvalues are also labeled such that $\lambda_1 \geq \lambda_2 \geq \lambda_3$. Thus by mathematical theorems $\mathbf{v_3}$ is the desired normal vector, $\mathbf{n}[i]$. A more detailed presentation of this is provided in the paper by Hoppe [97] and Appendix C.

In implementation the position of particle $\mathbf{p}_i$ is denoted $[x_i, y_i, z_i]^T$ and the mean position of the neighborhood, or centroid, of p_i is $[\mu_x, \mu_y, \mu_z,]^T$. The differences:

$$x'_j = x_j - \mu_x,\ y'_j = y_j - \mu_y, \text{ and } z'_j = z_j - \mu_z,,$$

are then calculated. These form the difference vectors:

$$\mathbf{d_j} = [x'_j, y'_j, z'_j]^T.$$

These difference vectors are then used to calculate the covariance matrix. Thus the covariance matrix is a sum of six products for each particle in the neighborhood. Stating this more clearly, if there are thirty particles in the neighborhood then the covariance matrix is calculated using 6·30 = 180 multiplies. Explicitly, for each particle, $\mathbf{q}_j$, in the neighborhood of $\mathbf{p}_i$ we calculate the products: $x'_j x'_j$, $x'_j y'_j$, $x'_j z'_j$, $y'_j y'_j$, $y'_j z'_j$, $z'_j z'_j$, and set CV[i] as follows:

$$\mathrm{CV}[i] = \sum_{\mathbf{d_j} \in \mathrm{Nhbrhd}(\mathbf{p}[i])} (\mathbf{d_j} \otimes \mathbf{d_j}) = \begin{bmatrix} \sum x'_j x'_j & \sum y'_j x'_j & \sum z'_j x'_j \\ \sum x'_j y'_j & \sum y'_j y'_j & \sum z'_j y'_j \\ \sum x'_j z'_j & \sum y'_j z'_j & \sum z'_j z'_j \end{bmatrix}$$

Where each sum is for j = 1 to [number of particles in Nhbrhd(p[i])].

5.2.2.2 Outward Direction

One of the weaknesses of MLS methods is they do not necessarily return a normal which points outward from the object. This is a known problem when dealing with representations that only contain "near surface" points [97]. This is primarily because there is no obvious way to decide where the interior of a possibly concave object is. In terms of display, this can be resolved by making each oriented display circle double sided. But that leads to problems in reflection, shading and transparency, so it is more desirable to have an outward normal.

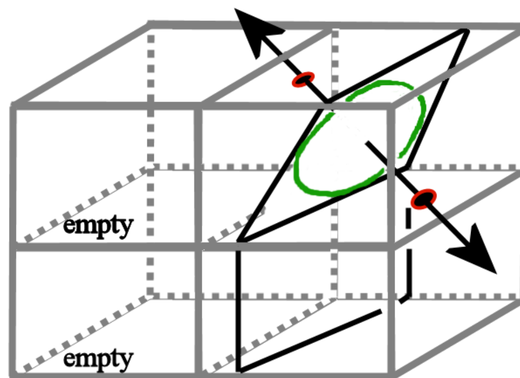


Figure 34: Two possible surface particle normals for a given particle.
Here the particle is the green circle and the upward-left pointing arrow is chosen as the surface normal because it enters an empty adjacent cell.

Fortunately, volpar representations not only identify which particles are surface particles, but also which are interior particles. Thus we rely on surface continuity and probability to direct the normal. The general idea behind this is illustrated in figure 34.

The process of selecting an outward pointing normal begins with the selection of a surface particle in a given surface cell. An MLS algorithm is then used to calculate a possible normal. However, this normal may point inward or outward.

It is assumed the majority of the empty cells adjacent to the containing surface cell are outside the object, or rather, are on the "outside" side of the tangent surface plane calculated for the given surface particle (i.e. the plane the oriented display circle is contained in). With this assumption, an outward normal should intersect an empty adjacent cell.

Thus we determine which two faces of the containing cell are intersected by the normal line through the particle being considered. In most cases, one of these faces is shared with an empty cell and the other is shared with an interior or surface cell. In these cases the normal is directed to point towards the face shared with the empty cell.

For cases where the normal goes through or nearly goes through an edge of the containing cell, all three adjacent cells are considered. If any are empty, then that is the direction of the outward normal. The normals of neighboring particles are then heuristically used to verify the decision (i.e. the normals of the neighbors should be in about the same direction). Similar is done in the case of intersections with vertices of the containing cell.

In the event both intersected faces are empty it is likely the cell is only surrounded by empty cells. If this happens then special consideration is given to the cell's contained particles. In most cases they are marked as excluded from display or are displayed as a single sphere or blob.

Another possible reason for both intersected faces to be empty is the object surface became exceptionally narrow or pointed within the containing cell. In this case only one neighboring cell face of the containing surface cell belongs to a non-empty neighbor. For this case, the point is rounded by setting the normal to be perpendicular to the shared face of the non-empty neighbor. This process is illustrated in figure 35.

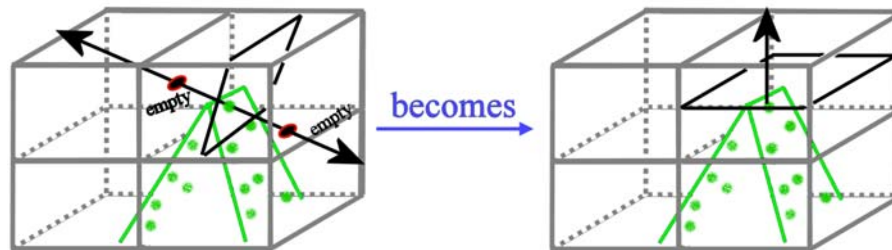


Figure 35: Surface normal for a particle at a narrow point.
In this case both possible normals enter empty cells, however only the bottom neighboring cell is actually non-empty. This the normal is select to point away from it (i.e. upward).

5.2.3 Applying Surfels

Once the outward pointing normals for each surface particle are calculated the display routine sorts the surface particles based on the distance from the viewer user. This allows proper blending and transparency effects to be applied. When sorting is complete, an oriented circle is rendered for each surface particle. Most of the examples shown in this dissertation intentionally leave the circles disconnected to better illustrate what is being done. Full implementations blend the circles and apply reflections, highlights, shadows, etc.

In sum the process is simple. Once the surface particles are identified, a surface normal is calculated for each. With this information each solid object is displayed as a set of overlapping blended circles. This is very appropriate, and works well, within our volpar framework. More complete details of implementing this display, once the surface particle information is available, can be found in various works [39, 40, 88].

5.3 Other Options

As mentioned billboarding and surfels are not the only other options for displaying volpar objects. For solids and liquids techniques such as marching cubes, iso-surfaces or level set methods can be used [89-91]. Each method has its own advantages and disadvantages. We have described the use of billboarding and surfels as they reflect “particle based thinking” and fit well into the volpar framework.

Beyond just using different rendering methods, offline options present possible ways to improve the modeled results. For example, as will be seen in section 6, the state of every particle may be recorded to a file for each time step of the simulation. These state files may then be given to another program. This secondary program can reformat the information to generate additional files to be used with advanced rendering systems employing raytracing methods.

Such offline rendering allows faster display methods to be used for proto-typing an effect. Once the effect has been “tweaked” to the user’s satisfaction, it is output to state files for the more time consuming process of raytracing. This type of process is common to multimedia organizations.

Another option in such methods is the possibility to use the second program to add in detail. These details may include texture effects or the addition of smaller details. For example, the particles might only model the formation of major crack lines (thus large particles are used). The secondary program then takes the results and introduces smaller crack lines extending from the major crack lines.

6. IMPLEMENTATION EXAMPLES AND RESULTS

The hypothesis of this dissertation is that a modeling framework based on a particle representation of objects and forces is feasible and robust. Previous sections have described such a framework and offered some basic examples of implementation. In this section more complete implementation examples are presented. These examples demonstrate various features of the framework and show a diverse range of modeling effects. Cumulatively they “prove” the feasibility and robustness of the framework.

A major advantage of a volpar framework is its ability to incorporate artistic as well as physically based effects into one engine. While several of the sections mention this to some degree, explicit details of how keyframing may be used within a physically based system are found in section 6.1.

Section 6.2 provides a brief demonstration of solid particle objects interacting with other solid objects. This is shown to illustrate the functionality of rigid body dynamics when applied to volpar objects. In this demonstration all the interactions are between particles. In particular, the collision detection and response are particle based. These methods of motion, in conjunction with the damaging effects illustrated in section 6.3, are the entirety of the hybridization of rigid body dynamics with particle based force propagation. Another feature demonstrated in this section is the application of the proposed surfel based rendering techniques for displaying volumetric particle objects in a dynamic modeling system.

One of the major concepts of this work is the fracture and breaking capabilities of solid objects. While the foundation for the development of this fracturing is presented in section 3, and further laid in section 4, a continued discussion and extensive examples focusing on fracturing is offered in section 6.3.

In section 6.4 and 6.5 we offer a demonstration of objects transitioning and interacting in different state phases. In these sections we provide examples of ice cubes melting, as well as, dirt combining with water to form mud. Within these examples further use of force particles and fracture patterns is demonstrated.

6.1 Keyframing

In this section a method allowing users to create artistic effects with particle systems is presented. This method, while giving artistic control to the user also maintains a degree of physical realism. To accomplish this, the position, velocity, density, orientation and color of the particles are keyframed to be at a specific state at a specific time. Between keyframes and when no keyframe is active, the particles are controlled by physically based forces. This combination of controlling forces, allows the user to create visual effects using the particles, while maintaining a degree of physical realism. Much of the material in this section was first presented in [99].

Discussion begins with a simple method of keyframing the position of every particle. This basic method is then expanded to achieve more robust keyframing using density to density and boundary to boundary methods. During this discussion two examples of application are illustrated.

The gains this keyframing method offers within the context of our volumetric particles include: increased user controllability and blending of natural and artistically motivated forces. However, it also functions with almost any particle system.

While effects similar to those presented in this section have been previously achieved, no general presentation of how they were achieved has been offered. There has been work done by others that might suggest such methods [100], but few have specifically applied or discussed the methods with respect to physically based particle systems.

Many rendering systems offer particle based effects. These systems often refer to keyframing particle effects. However, the keyframing they refer to does not involve the states of the individual particles, but rather the state and motion of the particle emitters. Effectively they are referring to sequencing the timing and positioning of particle effects. In a few of these systems it is possible to assign a goal for particles; however, it is only possible to assign one goal. Conditional forces, cannot be applied to the individual

particles. Further, the goals cannot be turned off or randomized for each particle. In the end, it may be possible to implement the keyframing methods we present on existing rendering systems, however the methods themselves are not inherent to any such system.

Related to the methods presented here is, the work done previously to keyframe the motion of smoke [101, 102]. Other, more general, keyframing techniques are also relevant [103, 104]. Yet, none of these applied keyframing directly to particle states. So while we are using similar ideas, we show how to apply them to particles in general. Also, no previous work was considering the possibility of using force particles in conjunction with the keyframing and certainly none were considering keyframing such force particles (as they have none).

6.1.1 Basic Method, Position to Position

The most straightforward method to keyframe particles is to specify each particle's initial and target position (or state). The difficulty is maintaining the natural behavior of the substance being modeled by the particles, while at the same time forcing it to do something very unnatural. With this in mind, if we are given an initial position and a final position for each particle and a set of external forces acting on the particle then it is possible to achieve an automatic 'in-betweening' of the keyframes using a simple process.

Assume the initial position of a particle, $\mathbf{p_0}$, occurring at time t_0, and the next position of the particle $\mathbf{p_n}$, occurring at time t_n is given, with a constant time step of $\Delta t = t_n - t_0$. Assume Δt is used throughout the simulation and a list of forces acting on the particle is known. With this information a force for each time step can be derived that applies the given forces to the particle ***and*** guides it to its final destination. It should be noted that keyframes are not always active, thus the particles may have periods of free motion.

For a given time step, i, let the current time be denoted t_i and let the sum of the forces acting on the particle be denoted by $\mathbf{F_i}$. For the same time step let $\mathbf{f_i}$ denote the force that if applied for the remaining

time of t_n - t_i = Δt_i would move the particle from its current position, $\mathbf{p_i}$, to its final location, $\mathbf{p_n}$, disregarding $\mathbf{F_i}$ (figure 36). Note the distinction between Δt_i and Δt. The first is the time remaining to the keyframe's end time and the latter is the time step of the current iteration.

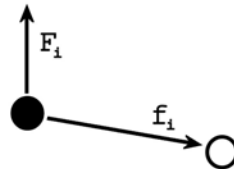


Figure 36: Two calculated forces acting on a particle.

This gives the total force acting on the particle to be: $(\mathbf{F_i} + \mathbf{f_i})\Delta t$. However, this does not guarantee that the particle ever reaches its target position $\mathbf{p_n}$. To make such a guarantee a scaling or weighting function, s_i, is introduced. Thus the force acting on the particle is: $(s_i\mathbf{F_i} + \mathbf{f_i})\Delta t$.

For simplicity we use a linear scaling function to define $s_i = \Delta t_i /(t_n - t_0)$. This is not the only function for all cases, and others can be used. This scaling term automatically diminishes the effect of $\mathbf{F_i}$, while the effect of $\mathbf{f_i}$ is inherently diminishing as the particle gets closer to its target. However, $\mathbf{f_i}$ may also be explicitly weighted if necessary. Further if to encourage each particle to behave differently a degree of randomness may be introduced by multiplying s_i by a random scalar $r_i \in (0, 1]$.

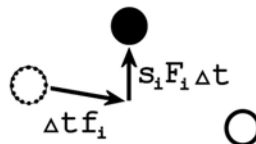


Figure 37: Motion by scaled force sum for ith time step.

Having calculated the forces $\mathbf{F_i}$ and $\mathbf{f_i}$ and the scaling term s_i, the particle for the ith time step is moved by applying a force of $(s_i\mathbf{F_i} + \mathbf{f_i})\Delta t$ (figure 37).

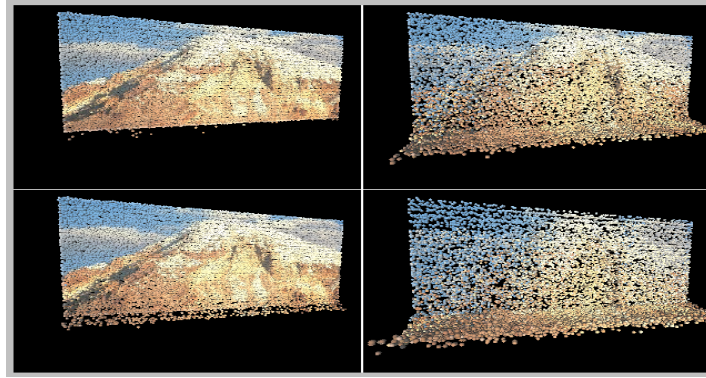


Figure 38: A picture composed of particles dissolving under gravity.

This position to position method is used to morph one image into another, as illustrated in figure 38 and figure 39. This morph begins by initializing the particles so they display their image. They then experience a variety of physically based forces, without any keyframing. After a certain amount of time passed, a keyframe becomes active guiding them back into a form that displays their contained image. During this morphing a timely change of particle coloring also occurs. This demonstrates a feature of this method by showing the particles need not move immediately from one keyframe to another.

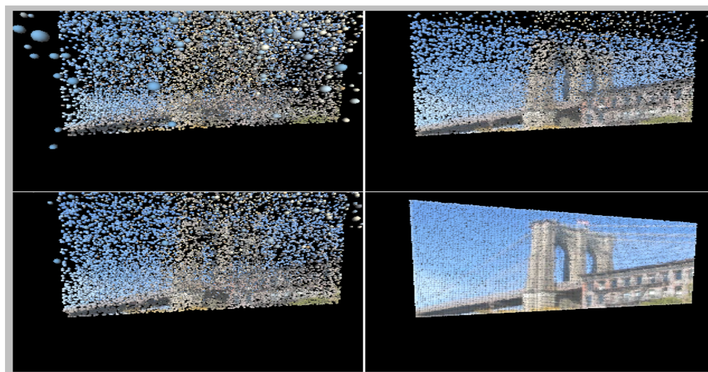


Figure 39: An image forming via keyframed forces.

6.1.2 Density to Density Keyframes

Because user specification of target goals for every single particle can become problematic explicit position to position keyframes are limited in their use. However, there are many algorithms that employ the use of density and velocity functions, for example those used to model fluids or gases [11, 105-107]. The effects of these methods are impressive but they can take some time to simulate and render, and do not inherently allow control of the visual effect. Recent effort has been made to remedy this. For example, it is now possible to control the motion of smoke and similar substances [101, 102] on grid based simulators but these are not particle based effects.

Because there are scenarios where the motion of the density of a substance is being considered, it is useful to have the ability to keyframe particles based on area, or volume, densities. From a user perspective, this means moving a given number of particles from one area to another. Exactly which particles get moved where is no concern as long as the necessary number of particles ends up in the correct area. While this can be from multiple areas to multiple areas, for a simple presentation the scenario is limited to a case of one source area and one or more target areas.

In this form of keyframing the shape of the source and target areas is of no concern. Only the number of particles is of any significance. While the word area is used in the descriptions the word volume equally applies. For additional simplicity we assume each mentioned area is of the same size so densities are more directly identified by the number of particles in each area.

For density to density keyframing we are not concerned with which particles go where. Rather we want to move a given number of them from one area to another to simulate a density flow pattern. To achieve this the task is broken into three parts: particle selection, destination calculation and path generation. To implement this a different function for each part of the task is used. To demonstrate these function some assumptions are made. Among these are: there exist no inter-particle relationships and there is only one source area. If these assumptions are not true then more complicated functions may be used, and even if

they are true, other functions may perform better in given scenarios. Yet the following worked well for our scenarios.

Before creating the paths of particles involved in density to density keyframes it is necessary to identify which source particles will move to which target areas. Assuming we have only one source and one target area, this is trivial. If there is one source and two target areas, where the density of each target is half that of the source, it is easy to randomly select half of the source particles and assign them to the first target area and the other half to the second target area. This readily extends to three or more target areas and can be adapted to work if there are multiple source areas. However, if there are relationships between the particles themselves or other such considerations, then other selection methods may be used. In all cases the source area must have a density great enough to support the target densities.

Once the source particles are selected and assigned to target areas it is necessary to determine where in the target area they will go. Assume there is only one source area and one target area. Obviously they could all go to the same location in the target area. However, that is usually not desired. To determine the target location of each particle a coarse estimate based on a center of mass concept is used (figure 40).

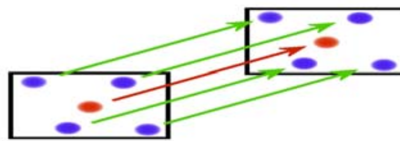


Figure 40: Motion based on center of mass.
In density to density keyframing, with no external forces, particles move as the center of mass.

Once the center of mass of the source area and target area is calculated, the simulation then runs forward. At each time step the center of mass of the particles is recalculated. The position to position keyframing is performed on the center of mass of the particles to obtain the $\mathbf{f_i}$ that will be applied to all the particles. The

$\mathbf{F_i}$ is still unique to each particle. This allows a performance gain by reducing the number of $\mathbf{f_i}$ calculations. However, if the particles are extremely dispersive in nature as in figure 41, or the relative size of areas involved is significantly different, the particles may not all end in the target area, which would require a refinement of the path. This need for refinement would require a detection method, such as the plausibility test described in section 6.1.4.

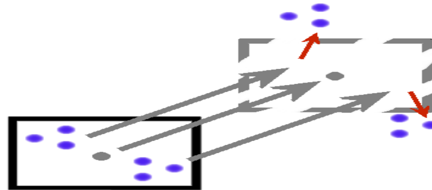


Figure 41: Dispersive external forces drive the particles too far apart.

To refine the path we subdivide the particles based on their locations at each time step, as shown in figure 42. This division is done using an adaptive kd-tree algorithm, where the divisions increase in number as the time approaches t_n. We then perform the same center of mass path planning as described above on each division. If necessary, this progresses down until each division contains only one particle. Thus, eventually, a path ending with the desired density in the target area is guaranteed.

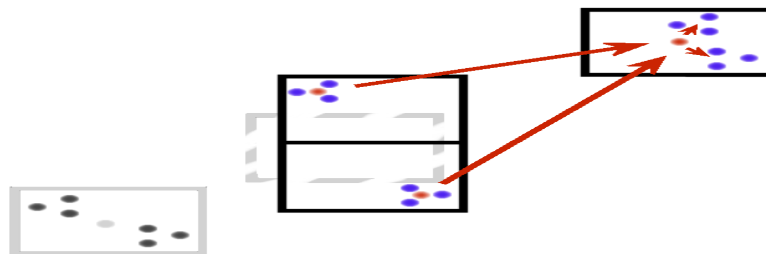


Figure 42: Adaptive subdivision corrects dispersion.

This center of mass concept may be skipped. However doing so may slow the simulation. Should that not matter the easiest method for density to density keyframing is again to rely on randomness, and assign each source particle a random location in its target area. Once each particle is assigned a destination, its path is generated as described in the position to position keyframing. Again, if there are interparticle relationships to be maintained, or other restrictive considerations, then other target assignment methods may be used, but the method of path generation stays the same.

6.1.3 Boundary to Boundary Keyframes

For boundary to boundary keyframing the steps are the same as used in density to density keyframing. Specifically, the task is again broken into three parts: particle selection, destination calculation and path generation.

For the particle selection process, particles from the source areas are assigned to go to specific target areas. For ease of presentation we assume there is only one source area and one target area. Should that not be the case random source to target area assignments may be performed, perhaps weighted by proximity to one another.

To calculate the destination of the particles we must emphasize that the particles are expected to visibly form the target area's shape. Thus they must fill the shape as much as possible. To accomplish this each source particle is assigned a specific target location. These target locations are generated as a uniform random sample within the target area.

As each particle has a destination generated for it, the path generation for each particle is identical to that described for position to position cases. This means an $\mathbf{f}_i$ is calculated for each particle.

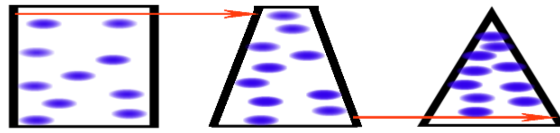


Figure 43: A square morphing into a triangle.

An expansion of this method also exists using current morphing techniques. Assuming the source and target boundaries are both closed then it is possible to create a morph between them [108]. From this an intermediate boundary shape for each time step is obtained. The movement of the particles is then confined to stay within, or near, these intermediate boundaries (figure 43). This confinement is enforced by the plausibility testing described in the next section. This results in a smoother transition.

6.1.4 Plausibility

In all three keyframing methods there must be a concern for the plausibility of the paths the particles follow. Thus, once a path is generated its plausibility is measured. This concept is explained well in [100], and we will be modifying and adding to their results, applying them to physically based systems.
While we could approach plausibility as an optimality problem and apply techniques similar to those in [109, 110] for visual effects we do not necessarily want the most optimal solution and will likely desire some randomness to remain or be introduced in the motion. To accomplish this we will stray from the optimality methods and introduce a random scalar $r_i \in (0, 1]$ and offer a change of the equation presented in section 3, where the total force was: $(s_i\mathbf{F}_\mathbf{i} + \mathbf{f}_\mathbf{i})\Delta t$, it now becomes: $(r_i s_i \mathbf{F}_\mathbf{i} + \mathbf{f}_\mathbf{i})\Delta t$. Other randomization techniques may also be applied or specified by the user. Likewise the $\mathbf{f}_\mathbf{i}$ term could be randomly scaled, however that removes the guarantee of hitting the target positions. This randomness allows multiple paths to be generated from the same algorithm. This should change the paths enough that some will be better than others.

It should be noted that technically the entire path from one keyframe to the next must be calculated to truly judge the plausibility of the path. To do this, speculative paths must be generated fast enough to not delay

the visual display. However the activation and duration of keyframes is user specified. To reduce the runtime we do not always judge the plausibility of the entire path, but just a small subsection going only a few time steps ahead of the current time. The exact number of time steps is left as a parameter to the user. If that number amounts to a time greater than the largest active keyframe duration, then the plausibility of the entire path will be performed for all keyframes. While that should generate better paths it is not required and may slow the runtime performance.

The process of generating paths and testing their plausibility is performed until a user specified level of plausibility is achieved or a given number of attempts is exhausted.

6.1.4.1 Plausibility Criteria

For our simulation methods, the plausibility is a measure based on:

- d = the distance of particles from their target positions,
- p = the viability of the particle positions,
- v = the ratio of the magnitudes of the velocity of the particles between time steps.

This plausibility is comparative in nature so the first path generated will always be accepted, but may be replaced by successively generated paths. To express this we will follow notation similar to that presented in [100]. However we will be testing individual particle paths, not all the paths all at once. So, letting g(*candidate path*) be the plausibility rating of a newly generated path and g(*current path*) be the plausibility rating of the currently chosen path then the probability of choosing the new path over the currently chosen path is:

$$P_{accept} = g(\textit{candidate path}) / g(\textit{current path})$$

This allows the new path to be chosen if P_{accept} is greater than a user specified value.

For a given path we will define three functions; g(d), g(p) and g(v) such that g($path$) = g(d)*g(p)*g(v). The details of each of these functions is described below.

It is important to understand these are only suggested criteria. Other measures of plausibility may be used as needed. Notice also each plausibility test is across only a small number of time steps, possibly one or perhaps the entire time from initial state to keyframe state. The number of time steps being considered will determine how reliable the plausibility test is.

6.1.4.2 Distance Plausibility

The distance plausibility of a path, g(d) is a measure of how close the particles are to their target states. In density to density keyframes this may be applied to the center of mass rather than individual particles. In every method it is defined as:

$$g(d) = \frac{1}{\sigma_d \sqrt{2\pi}} e^{-\|Pos(part[i]) - Dest(part[i])\| / 2\sigma_d^2}$$

where $part[i]$ is the particle indexed by i, Pos($part[i]$) is the current position of $part[i]$, Dest($part[i]$) is the target position of $part[i]$ and σ_d is a small user defined constant, 0.1 to 0.5 should work. This is similar to the center of mass measure presented in [100], though it is being used a little differently here. Other measures should also work. Such measures should have near zero values when the particle is far from its destination and go towards one as the particle nears its destination.

6.1.4.3 Viability Plausibility

The plausibility of the viability of the ending particle position of a path is a measure of whether the particle can be or should be in that location. Thus it is defined as two functions:

$$g(p) = h_c * h_s.$$

The “can be” part of the measure, h_c, is a Boolean function. If at any time of the path being considered, the particle is sitting somewhere that it cannot be, such as inside another object, $h_c = 0$, otherwise $h_c = 1$. Other criteria for this may be used, and the function need not be Boolean, however for our purposes this was sufficient.

The “should be” part of the measure, h_s, only applies in the case of a boundary to boundary morphing keyframe and is a function of the square distances of the particles from their temporary target locations. This is defined as:

$$h_s = e^{-k*sqrdist(part[i])}$$

where k is a user supplied constant and sqrdist($part[i]$) is the distance squared from $part[i]$ to its target destination. Values between 5 and 20 work well for k. This is similar to the shape measure presented in [100]. However the h_c term is unique to this paper and our points of distance measure for h_s are different. Other functions for h_s may be used. Such alternate functions work best if they are near zero most of the time and go sharply towards one as the particle nears its destination. The desired behavior of h_c and h_s is to strongly discourage “impossible” paths while still giving preference to paths that bring the particle closer to any intermediate destination it may have been assigned.

6.1.4.4 Velocity Plausibility

The plausibility of the magnitude of the velocity of the particles, g(v), is necessary to achieve a visual smoothness in motion. This measure is unique to this paper. For one time step

$$\mathrm{f}(v) = e^{\frac{-c*\left\|vel_c(part[i])-vel_p(part[i])\right\|}{\left\|vel_p(part[i])\right\|}}$$

where $vel_p(part[i])$ is the velocity of *part*[*i*] on the previous time step, $vel_c(part[i])$ is the current velocity and c is a user defined constant. Values near 1 should work well for c. Other functions may be used for f(v) as long as they are near one when the magnitude of the change in velocity is near zero and tend towards zero as the magnitude tends to infinity.

From this g(v) is the product of all f(v) across all the time steps used to generate the path:

$$g(v) = \prod_{all_time_steps} f(v)$$

This discourages sudden, large velocity changes across time steps.

6.1.5 Potential Refinements

While the above examples illustrate artistic effects mostly involving gas or liquid-like substances the described keyframing techniques may be used to accomplish other goals. For example they may be applied to solid objects to move them in a specific direction or in a specific way. This might be useful to make objects temporarily disobey the forces within the system or to guarantee two objects collide (or don't). Advancements in these keyframing techniques may also be used to morph solid objects into new forms. However this would also require solid connections to be broken and reestablished, which was not discussed in the above description.

In the end it should be seen these keyframing techniques offer a mere glimpse into the possibilities allowed within our modeling paradigm to combine the artistic with the physically plausible. It is hoped such effects would inspire others to develop them to yet grander implementations.

6.2 Basic Rigid Objects

The examples in this section demonstrate the basic motion of solid volpar objects. Within these examples are cases where objects collide and interact with each other and the surfaces of a containing room. Very little technical information is offered in this section as the goal is merely to illustrate a working implementation of our methods was actually created.

Also in this section we demonstrate the viability of our surfel based rendering. While examples of this have previously been shown, they have not been emphasized. So here we offer them to demonstrate they can, indeed, be used within a dynamic modeling environment.

The first example is one of simple block objects. In figure 44 we see a stack of blocks about to be hit by another couple of blocks. We also see the results after the collision. While still images are not the best to demonstrate a dynamic process, they provide evidence that our described methods can be implemented with believable results. This particular run of the program achieves a speed of about 15 frames per second (FPS), where the time step within the simulation is 0.01 seconds. For reference all of these simulations were run on a Dell Inspiron 5150 laptop with a 3 GHz processor, 512 MB RAM, and an nVidia GeForce FX Go5200 graphics card.

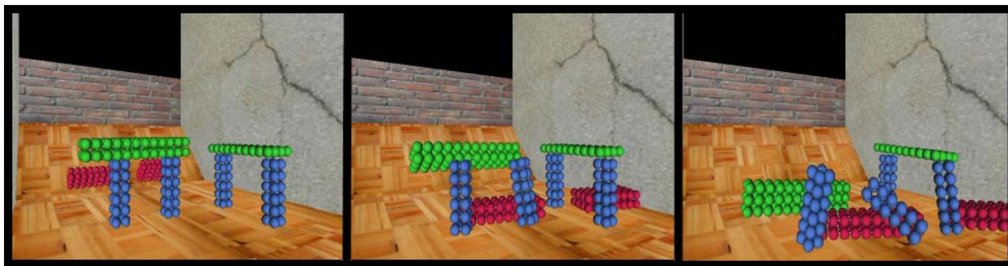


Figure 44: Block volpar objects colliding.

In our next run of the program, as illustrated in figure 45, we introduce Stanford bunnies. This demonstrates particle based, rigid body dynamics work correctly when applied to non-basic objects. Again

we see the objects before and after they collide. The large number of particles in this system, causes an excessive amount of memory swapping, so the runtime lagged to about one frame every five seconds.

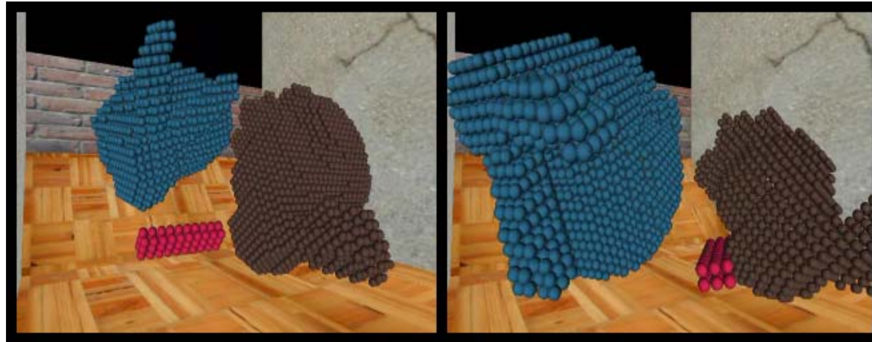


Figure 45: Stanford bunnies collide.

For the third run of our program, as shown in figure 46, we have turned on surfel based rendering. Here we again have a Stanford bunny, but this time rendered using a surfel based display. While in this display mode the program decreases speed to about one frame every fifteen seconds. This rendering is done using the methods described in section 5.

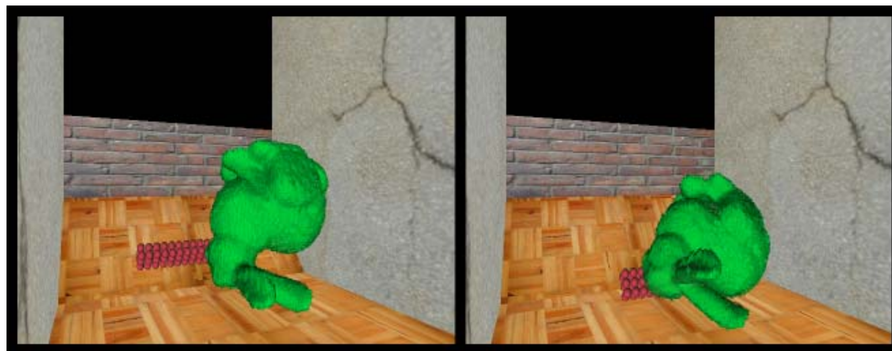


Figure 46: Surfel rendered Stanford bunny, bounces around.

6.3 Fracture and Breaking

The implementation of a fracture and breaking routine is one of the major test scenarios for determining the viability of using particles to model solid objects. Our method is designed to *reflect and approximate* the result of an impact on a particle based solid. The *results of this impact are an approximation* and are designed for animation purposes only. The material settings are intended to approximate effects and are not designed for rigorous use. However, in concept, volpar methodologies can be adapted to meet more demanding levels.

Before trying to understand the material in this section, some background is required. A brief description of the more popular, existing methods of fracturing and breaking objects is found in section 2. The background and details concerning force propagation can be found in section 4 and a description of connection strengths is found in section 3.

Also before getting to the examples of fracturing some additional material is required. This material is presented in this section as it directly applies to fracturing. In sum, connection strengths are relative in meaning (section 6.3.1). Many modelers are not engineers and are not interested in setting connection strengths to accurately model a fracture. Yet, most want a realistic fracture to occur. To address this need we develop the use of fracture patterns (section 6.3.2). This requires a way to create them (section 6.3.3). Once created they must be applied. To accomplish that, a scoring method is developed (section 6.3.4). This leads back to fracture patterns and fracture tip propagation (sections 6.3.5 and 6.3.6).

With this additional background established we present some of the implementation details, followed by a variety of examples (sections 6.3.7, 6.3.8). We conclude with a short summary of the results (6.3.9).

6.3.1 Connection Strengths

In sections 3 and 4 it is shown force particles and wave propagation can be used to damage the connections between particles. This is on the assumption the connection strengths have been set to reflect

the strength of the material composing the object. While this is useful, and can increase the accuracy of the fracture and breaking, it is also daunting and intimidating for some users. To help remedy this, we allow the option to set the connection strength to a default value. In simplest terms this setting can be described as a scale of values, where 0.01 means easy to break and 100.00 means difficult to break. It is assumed these values relate to the known fracture toughness of a compositional material. A discussion of such properties and values for various materials is available in a number of engineering texts, such as Callister's book on materials science [58]. We also note the exact numeric values are not as critical as the relation between the numbers. For example, a material with an average connection strength of five will break much easier than a material with an average connection strength of eighty.

6.3.2 Material Fracture Patterns

In conjunction with the connection strengths of a material, a fracture pattern can be selected for a given object. Thus if a user is modeling a glass object then he would likely set the general strength of the material to something low, such as 5.0 and select a "glass material fracture pattern." Of course this results in a glass-like fracture pattern ***developing*** when the object is struck, but there are some details to discuss. The concept of associating fracture patterns with a given type of material appears in a variety of literature [111-113]. One of the more interesting sources is the work of Desbenoit et al. [114, 115]. It is in this work an "atlas" of fracture patterns was developed, and applied to various objects by subtracting volumes from them based on the fracture pattern. These fracture patterns were selected by the user based on material type, and previous to application to the object they could be modified by the user (e.g. rotated, skewed, lengthened, narrowed, etc). This is the concept we have taken and applied to our volumetric particle systems. It should be noted, that Desbenoit, did not intend his method to be used with volumetric-type objects, nor in conjunction with physically based modeling. This is made obvious in [115] as their method is being compared against voxelized, or tetrahedralized, objects as well as against physically based techniques.

So our adaptation of this method to volpar objects (or anything similar) was not something he or his colleagues were considering. It is also important to note, the actual methods are quite different. The only commonality is the classification of the material fracture patterns and making them available to the user. Specifically, while Desbenoit's method carves out the fracture from existing, B-rep or CSG, objects, ours weakens the connections of the object at the time of impact. So our method does not force the exact fracture pattern to occur, but rather, encourages a pattern similar to that requested to emerge. This means this is a randomized effect. Even if exactly the same pattern is applied the result is dependent on the connection strengths, the impact force, and the location of impact.

6.3.3 Material Fracture Patterns, DLA Grown

While observationally based material fracture patterns are clever, and an atlas of such patterns can be established [115], there are other methods for generating fracture patterns. In this discussion we move away from the material specific patterns and towards a more artistically driven, though scientifically motivated, method. This method is chosen as it too functions using particles as its primitive objects.

An advantage in using this method is it allows a sequence of images to be produced as the pattern grows. This was useful for a time when we wanted to accurately model the fracture tip propagation of a crack. Thus we could apply a sequence of patterns to an object as it was hit multiple times. We have since advanced to a better method to encourage fracture growth. However, using a sequence of images led to our final method and we believe this sequencing idea is useful by itself.

Diffusion limited aggregation (DLA) was introduced in 1981 in the paper by Witten and Sander [30] to model irreversible colloidal aggregation. It was quickly realized the method they were proposing could be applied to a great many things, and it has been. In general this technique creates patterns which are similar in appearance to snowflakes or lightning. We use it to create a 2D surface map. This map represents a surface fracture pattern. While there are many other ways to create this map, DLA is particle based and, as such, demonstrates yet another use of particles. The DLA method we describe is not unique, however we

use its results in a unique way. Specifically, as the method "grows" a pattern it allows us to take snapshots of it at various moments in time. This allows us to obtain a sequence of images that we use to model and display an impact fracture growing on the surface of an object. This process is done offline and its results are stored for later use within the actual simulation.

The general concept of DLA is to place a seed particle somewhere in 2-space. Another particle is then added "far away" from the original particle. This second particle performs a random walk. If it goes too far away from the first particle it dies and is replaced. This should not happen very often. Instead, what is hoped, is that it eventually comes close to the original particle. When this happens it is frozen in space and a third particle is introduced. This third particle then begins its own random walk. This process repeats until a given number of particles have become stuck.

While this seems to be a very trivial algorithm it has been studied in various fields for over twenty years. For our purposes we are content that it creates a nice random radial pattern and it does so in a way that makes the pattern appear to grow. As we implemented this process, the user was allowed to specify a distance d to determine when to take snapshots of the growth. Specifically whenever a particle freezes at a distance of $k \cdot d$ from the original particle a snapshot is taken. In this, k begins at zero and increments each time a snapshot is taken. This process continues until a specified number of snapshots have been taken or a given number of particles have been frozen. Within each snapshot the endpoints of each major branch of the image are recorded. This requires the use of two files, an image file and an associated meta-file. The image file is literally the image. The meta-file defines a graph used to apply the pattern within the fracturing and breaking program.

Other methods may also be used to generate these sequenced images of growing fractures. As stated previously we have chosen to use a DLA particle method for two reasons. The first is it literally grows the fracture pattern. The second is because it is a particle based method, and we wish to further illustrate the usefulness of particles.

6.3.4 Scorelines

In general scorelines are defined by a startpoint and endpoint. These are 2D (co-planar) in nature and are on the surface of an object. From these two points the average surface normal between the points is calculated. For example, assume the object is a cube and the scoreline is drawn on the topmost surface. Assuming +y is the up direction, then the average surface normal between the points would be <0, 1, 0>. Once this average normal is calculated, a plane containing the scoreline's points is calculated such that its normal is perpendicular to that of the average surface normal (e.g. <1, 0, 0>). Stated simply, the points of the scoreline define a plane which slices through the object.

When a scoreline is drawn on an object its plane "cuts" through the object. In this "cutting" the connections near the plane are weakened. The exact amount connections are weakened is a user specified, or material based, parameter. This parameter reflects the effectiveness of scoring on a given material. For example, a piece of wood is not influence by scoring as much as a clay brick. Regardless of the exact amount the material is weakened, when the object is struck, after being scored, it is most likely to break along the scoreline's plane. This might seem to always cut an object too cleanly. However, the particles within the object are not in any specific pattern and connection strengths are randomized, so the probability of such a clean cut is unlikely. Thus, should the object break along the scoreline, the volpar representation inherently introduces a jaggedness to the breakage.

In implementation scorelines are bounded or unbounded. Most drawn by the user are unbounded, or rather, they are extended at least to the edge of the object. This behavior is usually the desired behavior because the scoreline is used to cut the object into pieces. To accomplish that task, the scoreline must extend to the edges of the object. Bounded scorelines, however, influence a smaller region. They start and stop at the specified endpoints, so rather than defining a cutting plane, they more accurately define a cutting rectangle, finite in two directions (x and z) and infinite in the third (y).

From an engineering, or physics, perspective a scoreline is the introduction of a near continuous line of small surface fractures. Thus, as objects tend to break where fracturing has already begun, the object tends to break along the scoreline. To the best of our knowledge no other volumetric fracture method has made use of such scorelines. Also, while others have allowed lines to indicate fracture areas, we know of no other physically based fracture methods designed to allow the user to use scorelines, as described here, to specify where fracturing and breaking should occur. Specifically, the physically based nature of the scoreline applied to volumetric modeling is unique to this dissertation.

6.3.5 Fracture Patterns as Scorelines

Scorelines also play an important role in applying material fracture patterns. A fracture pattern can be viewed as a set of bounded scorelines. As stated before, we use the term bounded to imply the effect, of the scoreline, does not extend (far) beyond its endpoints.

Having established what a bounded scoreline is, we use them to apply fracture patterns described in 2D. For example, a glass-like fracture pattern is defined as series of circles of increasing radius formed from line segments, where each line segment is a bounded scoreline. Further additional line segments are added that extend between the circles. This fracture pattern, if associated with a given object, can then be applied to the object at the time of impact, centered at the impact point. An example of this circular fracture pattern is shown in figure 47.

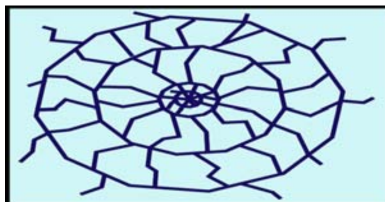


Figure 47: Generic glass-like material fracture pattern.

In a similar fashion *n*-prong fracture patterns can be applied using scorelines. To accomplish this, each line segment of each prong is treated as a scoreline. Examples of three and four prong fracture patterns are presented in figure 48.

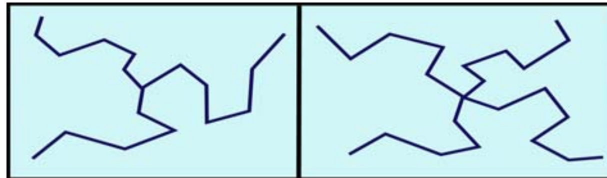


Figure 48: Generic three and four prong material fracture patterns.

6.3.6 Fracture Tip Propagation

There is no continuity in random, at least not in the way we assign strengths to the connections of our solids. This leads to a difficulty in fracture development. Specifically, all fracturing observes what is termed "fracture tip propagation." What this means is that if an object has a fracture and is struck, then the object will tend to fracture along the pre-existing fracture. Of course, the existing fracture may fork to create new fracture tips. But even these are usually very near the pre-existing fracture. Because the strengths of our connections within a solid are uniformly random, it is unlikely any continuity in strength exists between adjacent connections. What this means is when an object is struck, and force particles propagate through it, the connections which break are random. With a sufficient number of impacts this may cause the object to break into pieces, but there is no guarantee these pieces will not be just every individual particle (i.e. the object crumbles into dust).

To better ensure our objects break into larger chunks, and to assist in the visual appearance of fracture tip propagation, we use fracture patterns. While this may initially seem as though we are explicitly specifying how an object will break and circumventing the connectivity strengths, we are not. The fracture patterns only suggest a pattern. The results of the pattern depend on the initial strength and position of the connections. Further, the patterns are applied at run-time, based on where impacts occur. Thus, in no exact

way, is the breakage of the object predetermined. Also, this is using the strengths of the connections, so we are not neglecting them nor overriding them. In essence a fracture pattern only *suggests* what will happen after an impact occurs, if and when there is sufficient force to take the fracture to completion (extend the fracture tip through the entire object).

To better understand how we came to use fracture patterns, we must step back to where we began, which was with DLA processes. Using these processes we created a sequence of fracture pattern images. In this sequence of images we literally can watch a fracture pattern grow. Thus, if each pattern is applied sequentially, the tips of the fracture grow outward and the desired fracture tip behavior is achieved.

In the end, though, we discarded the image sequencing method as simply applying the last image in the sequence is sufficient. This is because the strength of our force particles decreases, not only as a function of distance traveled, but by connections broken. So as long as the fracture pattern extends farther than any of the force particles will travel, there is no need to have multiple patterns. Basically the force particles will continue the fracture as appropriate because they can go farther after each hit. This is illustrated in figure 49. This force degradation was discussed with respect to impact waves in section 4.3.2. We extend that discussion to force particles and apply it to fracture tip propagation here.

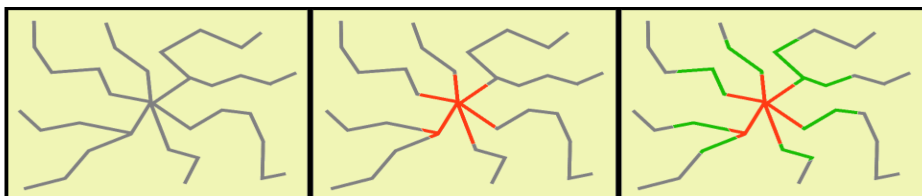


Figure 49: Fracture tip propagation by application of a material fracture map.

To explain figure 49, on the left, the entire fracture pattern is displayed in light gray, where the initial impact occurred near the center of the pattern. After the first impact, the force particles propagate outward far enough to break particles near the portion of the fracture pattern displayed in red, as shown in the

center image. After the second impact, the force particles are able to travel farther, and propagate far enough to break particles near the green portion of the fracture pattern, as shown in the right image. On successive impacts the force particles might eventually be able to propagate far enough to complete the fracture pattern. Notice, key in this effect, is the application of a fracture pattern only on the first impact. Other fracture patterns might be applied on successive impacts, if the distance from the original is sufficiently far away.

To explain this fracture continuation, let us reduce the scenario to 2D. Assume each connection has a strength of one hundred. Assume a simple scoreline reduces all affected connections to a strength of one (on average). The object is struck near one end of the scoreline, with a force strength of forty. Assume on average each particle has a diameter of d, and if the force particle travels a distance of d it automatically loses one point of strength and does one point of damage to every connection encountered. So, assuming this was all that happened, the force particle would always travel a distance of $40d$ from the point of impact. However, also assume when a connection breaks, a little more force (energy) is expended in the process. Specifically, we require the force particle to lose one point of strength for each connection it breaks. So if it breaks one, its maximum distance is $39d$. If it breaks two, its maximum distance is $38d$, and so on.

This strength reduction process is simple enough, but does not seem extremely useful until the results are observed. For the sake of argument, we assume the scoreline affects 30 particles from one side of the object to the other. By no coincidence we strike the object at exactly one end of the scoreline and for some reason only one force particle is generated and it travels precisely along the scoreline path. Clearly it only travels a distance of $20d$, breaking twenty connections along the way. This is insufficient to break the object but creates a fracture line. According to fracture mechanics, if the object is struck again the fracture line should continue to grow. So we strike the object a second time. This time the particle, when it reaches a distance of $20d$, has not fractured any connections. So it still has a strength of twenty remaining and travels farther. In fact, it travels another $10d$ units, breaking ten more connections, and the object breaks

into two pieces. Thus we see the fracture tip is contained within the fracture pattern. Of course there still remains a problem if our fracture pattern is too “short.” So in implementation effort must be made to prevent that.

Other possibilities exist for developing fracture propagation and achieving larger chunks when the object breaks. For example we might group a random number of adjacent particles together, giving the connections between them a specific strength. If this is done such that every particle belongs to a group, and the connections between groups are weaker than within the groups, then fracturing should occur between the groups. When we implemented this as an early prototype, it did achieve larger chunk sizes, but did not help much with fracture tip propagation. The grouping methods used were similar to those described in section 3.9.

Another possibility is to weaken all the connections near any connection that breaks. In the beginning, we believed this would be useful. However, in implementation, it proved only to cause a large amount of dust particles to be formed where breaks first occur. When extended, it effectively causes all the connections to fail (i.e. more crumbling and dust). However, we believe it might still function, but is highly dependent on the relative strengths of the connections. If they are all basically the same value, then only dust is likely. However, if implemented with the above grouping idea, it could produce better results, as the fracturing would “want to” extend along the weaker connections, as they would fail first. This latter idea is similar in nature to the brick wall example that is given in section 6.3.9.

A third possibility, and one we did not try, is using the dual graph of the connection graph. Since our connection graph maintains a record of which connections existed and broke, we are able to construct a graph of broken connections. Notice every connection also has a direction based on its end particles. If we created a tree of broken connections with a root node at the impact point, then the leaves of the tree would be likely candidates for fracture tip propagation. So, if we consider the leaf nodes, and the planes perpendicular to their broken connections, we would have possible fracture planes. These planes might

somehow be useful in extending the fracture tip. For example, we might look for connections near the leaf nodes that are cut by these fracture planes and weaken them. This, of course, is all speculative as we did not implement this option.

So, in the end, we have chosen to use fracture patterns and the dissipative nature of our force particles to mimic fracture tip propagation. In this we have deviated from a strictly physically based approach, as we are “peeking” into the future when we apply the final fracture pattern, instead of letting it develop from time step to time step. However, its appearance is still restricted to time steps, and whether it appears at all depends on how hard and where the object is struck. As stated this is not perfectly physically based, but is the most natural extension to accomplish fracture tip propagation within the context of our fracturing procedures.

6.3.7 Implementation Details

In this section we describe the layout of our implementation. This includes a description of the preliminary processes involved, the program layout and design, and the results produced. There are undoubtedly some details left unmentioned, but the more significant points are covered. This information is offered to tie the abstract concepts from previous sections to an actual implementation.

Table 11: Structures used in implementing fracturing and breaking.

Structure Name	Variables and Methods	
Particle	Boolean	mb_alive
	Integer	ml_component_ID
	Float Array	md_position[]
	Float Array	md_velocity[]
	Connector Pointer Array	mp_connection[]
Connector (all rigid here)	Boolean	mb_alive
	Boolean	mb_broken
	Particle Pointer	mp_first
	Particle Pointer	mp_second
	Float	md_strength
	Float	md_maximum_strength

The first item of consideration is data structure. After a multitude of versions, the data structures which best suit this paradigm are the structures and classes as described in table 11 and 12.

Table 12: Classes used in implementing fracturing and breaking.

Class Name	Variables and Methods	
Scoreline	Void Void	Initialize(Startpos[], EndPos[]) Apply(ConnectorList[], ListSize)
FracturePattern	Void Void Void	SetType(Type, Filename) SetPosition(Pos[]) Apply(Connector List, List Size)
ForceParticle	Void Void	Initialize(Pos[], Vel[], Dir[], ObjectHitID, Strength) Update(ConnectorList[], ListSize)
World	Integer Void Void Void Void Void	InitializeFromINI(filename) InitScore(Starpos[], EndPos[]) SelectMaterialFracturePattern(PatternID) ApplyImpact(pos[], force, direction[]) Update(TimeStep) Display()

Using the structures described in table 11 and the classes described in table 12, the update process was developed within the world class to function using an existing graphics library. The details of this update process have already been described in general terms throughout this dissertation, so are not repeated.

Using these structures and classes the flow of the program begins by initializing the world class as determined by the contents of an INI file. Within this INI file, parameters are given to determine: what the object filename to load is, how to rotate and translate the object, what default material properties to use for the object, and how to display the object. Other options may be specified, this is just a list of some common features.

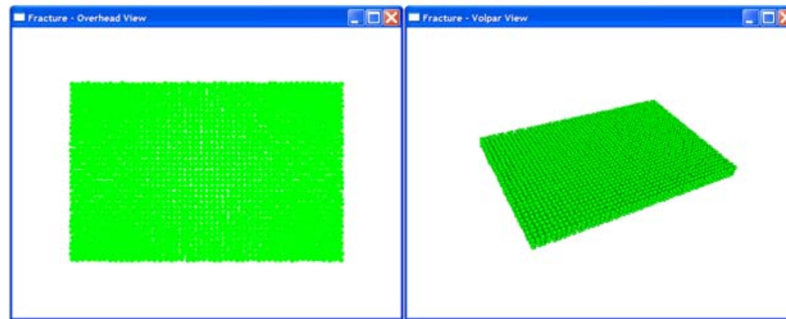


Figure 50: Overhead view and volpar view of fracture program.

Once the program is running, the user is presented with two views. Examples of these views are shown in figure 50. One of the views is the “shadow” of the object projected onto the xz-plane, where the “sun” would be placed at infinity along the +y axis. This view is termed the overhead view. To orient it as desired requires the INI file to specify the correct rotation. Most of the user interface operates based on interaction with this view. The second view is a 3D view of the object. This view may be oriented however the user wishes, allowing the results of operations to be better viewed. This second view is termed the volpar view.

Table 13: Keyboard interface options for the fracture and breaking program.

Key Pressed	Resulting Option
[s]	Toggles full/partial scoreline breakage.
[r]	Toggles bounded/unbounded scoreline extensions.
[j]	Toggles jagged/unjagged scoreline.
[m]	Selects glass like fracture pattern to be applied.
[3]	Selects a 3-prong fracture pattern to be applied.
[4]	Selects a 4-prong fracture pattern to be applied.
[f]	Selects a file-based (pregenerated) fracture pattern to be applied.
[+]	Increases impact strength.
[-]	Decreases impact strength.
[o]	Toggles between point and circle based impacts.
[spacebar]	Reinitializes the system from the INI file.
[ESC]	Ends the program.

The interface used within our program is 'hot key' and mouse based. While this is simple in design, it allows the greatest flexibility in development. Some of the options offered through the keyboard are listed in table 13.

The mouse is also used to interface with the program. Its functionality depends on which window it is over. If it is over the volpar view it alters the viewing orientation of the object, either translating or rotating depending on which mouse button is held down. If it is above the overhead view, left clicking and dragging draws a scoreline on the loaded object. This is similar in effect to scoring a clay brick. The reason for scoring an object is to encourage it to break along that scoreline.

To fracture the object, once all the options have been duly set, the object must be struck. To accomplish this, the user right clicks on the object as displayed in the overhead. This causes an impact to occur on the object. This impact is then propagated through the object via force particles. As the force particle moves, connections within the object are damaged. As the connections break, various fracturing occurs. What exactly happens is determined by the object connection strengths, the location of impact, the selected material fracture pattern, and any scorelines drawn on the object.

More accurately a force particle (or set of force particles) is generated at the user specified point of impact. From this point the force particles radiate outward as a wave front. As they encounter connections they damage them. As they continue to move outward, the force they carry with them weakens, eventually becoming zero. In this weakening process the majority of damage done to the object is nearest the point of impact. To actually break the object requires a rather strong initial force, or repeated impacts.

6.3.8 Visual Results

To illustrate the effectiveness of this implementation we now examine some of the visual results. With each result we offer a brief description of how the result was obtained, and what it is demonstrating.

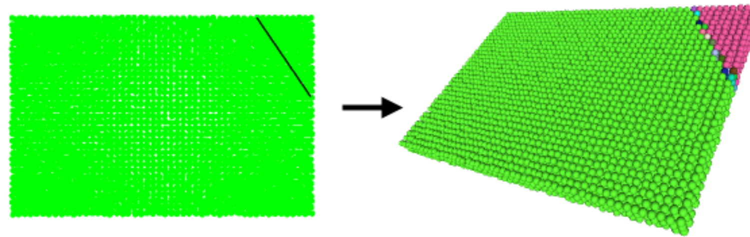


Figure 51: Simple scoreline fracture.

We begin with a simple example, as shown in figure 51. Here we start with a basic thin plate, or wall. We apply a score line to the upper right corner and strike the corner. The average strength of the particle connections is numerically twenty. The scoreline is set to reduce the strength of the connections on average by 80%. The strength of the impact force begins as ten, and travels through about one fourth of the plate. So the visual breakage that occurs is consistent with the given numbers.

One the challenges faced in using our fracturing method, is proper fracture tip propagation. The details of overcoming this challenge are present in section 6.3.6. The results of this are shown in figure 52.

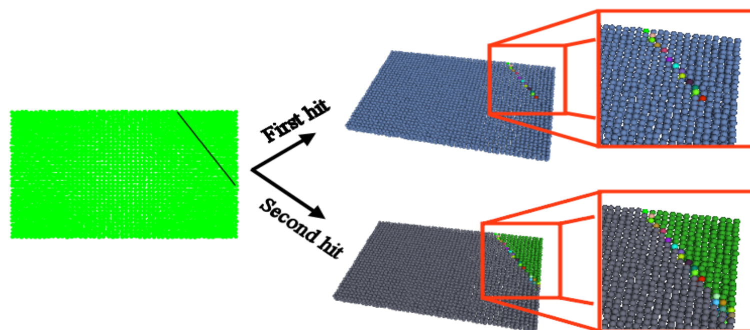


Figure 52: A plate struck twice.
The first impact only creates a fracture line. The second continues along the same fracture path, breaking the object. The color changes in the object are deliberately random.

The next example, shown in figure 53, demonstrates we can load pre-created volpar objects and fracture them. In this case we have loaded a chess pawn and scored it for breaking along its "neck." As in the case of the thin plate the average strength of the particle connections is numerically twenty. The scoreline is set to reduce the strength of the connections on average by 80%. The strength of the impact force begins as ten, and travels through slightly less than one fourth of the pawn. So the visual breakage that occurs is consistent with the given numbers.

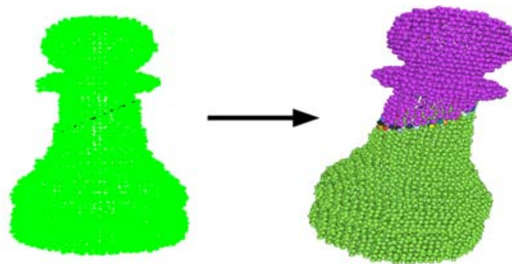


Figure 53: A chess pawn scored, breaks into two.

In our next example, as shown in figure 54, the application of a glass-like fracture pattern to a thin plate is demonstrated. This is the first non-scoreline example, however it functions much the same way as the scoreline. From a user standpoint the only difference is a material property is selected before applying the impact, and there is no line drawing.

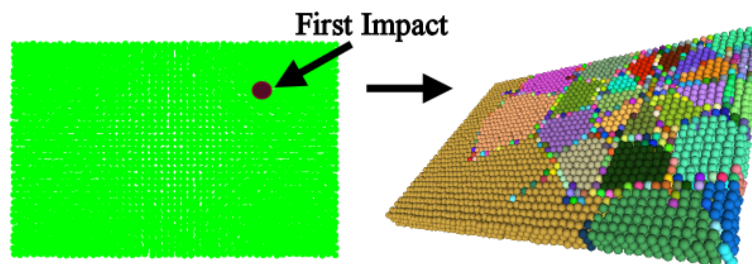


Figure 54: Glass-like fracture pattern, first impact.

In this example, we again demonstrate the fracture tip propagation. Looking back at figure 54, we see towards the bottom-left there is a fracture line extending towards the bottom edge, but not reaching it. If we strike the object a second time, in the left corner, we should be able to complete that breakage. In fact, that is exactly what happens as shown in figure 55.

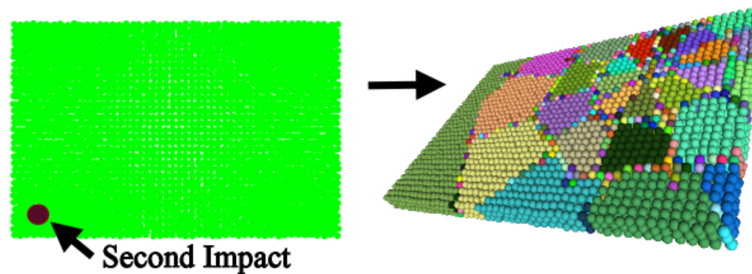


Figure 55: Glass-like fracture pattern, second impact

While plates are interesting, users may desire to fracture other objects. In the next example, shown in figure 56, we see we can also fracture the Stanford bunny. To accomplish this we apply a four-prong fracture pattern near its neck.

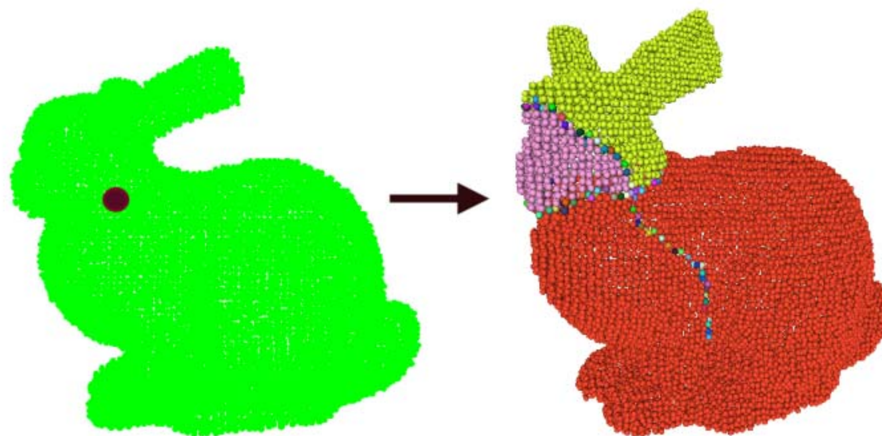


Figure 56: Four prong bunny fracture.

So far, all our objects have been homogeneous compositions. To demonstrate the robustness of our particle modeling we now show an example of a brick wall. In this, we give the bricks a connection strength of ninety, whereas the mortar only has a strength of ten. Thus if we strike the wall, without any scoreline or fracture pattern, the mortar joints break, and the bricks loosen. This is seen in figure 57.

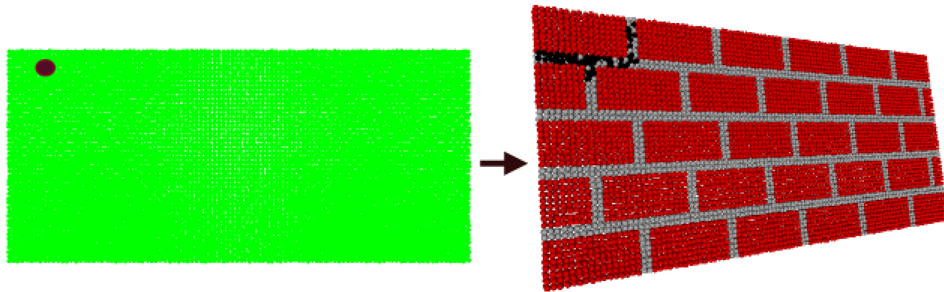


Figure 57: Heterogeneous brick wall.
No material patterns or scorelines have been used, just connection strengths.

Using this same model we can also apply material fracture patterns to encourage the bricks to also break apart. For example, if we apply a three prong material pattern and strike the wall, the bricks and the mortar break. This is shown in figure 58.

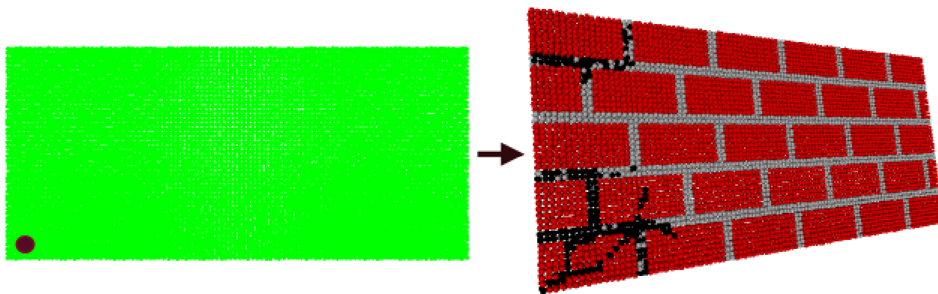


Figure 58: Brick wall with a 3 prong material fracture pattern applied.

In both of the brick wall examples, the robustness of our modeling system should become clear, as few other systems allow for heterogeneous objects. Further we have demonstrated the fracture patterns are not the only factor in determining the breakage patterns. As the mortar is significantly weaker than the brick, and fails as expected, we have shown the connection strengths also play a significant role in the outcome of a forceful impact.

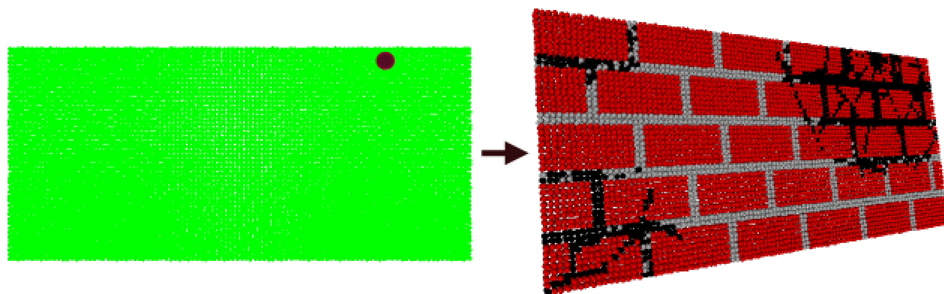


Figure 59: A demolition ball hits the brick wall.

As an extra bonus we show, in figure 59, a third impact against the same brick wall. This time we have chosen to apply a glass-like fracture pattern. This would be similar to striking the wall with a large demolition ball.

6.3.9 Results Summary

In terms of implementation, the application of our force particle method with solid, volpar objects, has advantages in speed, stability and ease of implementation. When compared to other methods [49, 50] our method avoids relying on particle to particle transmission of forces. This improves the stability and speed of the method. To accomplish this some accuracy is lost. However, the visual results are still good. Further when we consider our representation also allows objects of heterogeneous composition to be modeled, and is capable of providing interaction with liquids and gases, it seems well worth the small amount of loss. Beyond this our method is significantly easier to implement than O'Brien and Hodgins' method. While it

is of comparative difficulty to Norton et. al.'s method, the construction of objects is much more diversified and the results are comparable if not better.

In terms of abstraction our force particle propagation method has advantages in speed, stability and ease of understanding. When compared to other connected particles systems, specifically spring based systems our hybrid, rigid-particle solid representation avoids almost all the stability problems inherent to spring only representations. It also avoids the tweaking of individual spring settings. It is also as easy to understand. When compared to finite element methods our representation simplifies the process by allowing the forces to transmit themselves across the material representation. This removes the explicit calculations necessary in the development of finite element methods. In both cases our method has implementation speed advantages as it, in general, may use larger time steps. Thus the more basic integration methods may be applied rather than requiring an adaptive or implicit method. So, again, it is easier to understand in abstraction.

We have also introduced the concept of a scoreline into our modeling system. Using this we are able to apply fracture patterns. These fracture patterns can be derived from an atlas of images of real-world fractures, be generated by DLA methods, be generated by procedural methods, and can be further modified or specified by the user. These fracture patterns are not repetitive in nature. So even if the same pattern is applied to multiple objects, it will have variations. Thus the pattern itself should not be recognized as repeated. In sum, this allows for a rather diverse set of results ranging from realistic to artistic.

In particular the artistic and user driven options are significant. Many fracturing methods are either all artistic or all engineering based. Ours deviates and allows the user to decide how accurate they want things to be. So, while it is possible to set every connection within an object to have a strength determined from statistical analysis of the material an object is made from, this is cumbersome if the user does not possess an engineering background. To alleviate this, as mentioned, there is an option of also associating a

material based fracture pattern with any object. This is a significant deviation from previous physically based methods. But stays true to such methods as the intent is to use fracture patterns derived from the scientific study of materials (such as that done on glass [116-118]). Yet, again, there is no inherent requirement the fracture pattern be so rigorously studied and so artistic desires are met.

In terms of fracture tip propagation, our method is inaccurate. However, we are able to overcome this by using to fracture patterns in conjunction with the force dissipation of our force particles. In the end this allows us to produce a reasonable effect to simulate such propagation. While it may lack in accuracy, it is sufficient for most entertainment-based scenarios.

With regard to speed, our fracture method runs in near realtime. Specifically, all the results shown within this section took between several seconds to ten minutes to complete. This was accomplished running the program on a Dell Inspiron 5150 laptop, with a 3 GHz Pentium 4 CPU and 512 MB of 1.6 GHz RAM.

In total, the results of our fracturing method using volpar objects and force particles is comparable to other methods. It also provides greater capabilities than previous methods and runs in near real-time. All of which make our fracturing method using volpar objects and force particles a significant contribution to the field of computer science.

6.4 Dirt, Water, and Mud

In this section we present an example which uses three basic particles. The particle types are: dirt, water, and mud. Using these three types we demonstrate how dirt and water particles combine to produce a mud particle. We then demonstrate the mud can dry and return to dirt. To create a cracking pattern in the drying mud, we revisit our fracture patterns and show how they can be associated with force particles. In all of this we are providing another example demonstrating the robustness and possible usage of a modeling framework based on particle masses and force particle representations.

The setup for this example is straightforward. We create three particle types. The first is water. This particle type experiences short range repulsive forces, medium range attractive forces, and no intra-particle forces at long range. For this implementation short range is a distance from 0 to *r*, where *r* is the radius of the particle. Medium range is a distance from *r* to 5*r*, and long range is a distance greater than 5*r*. Water particles are subject to gravity, and will lose mass if the user turns on evaporation (i.e. global heat is applied to the entire environment).

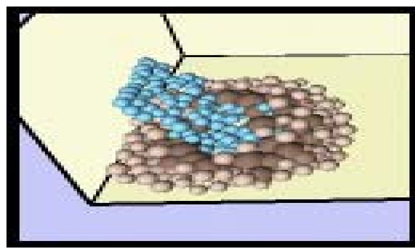


Figure 60: Water mixes with dirt particles to form mud particles.

The second type of particle is a dirt particle. These experience no attractive forces and experience rigid collisions between other dirt particles (i.e. no overlapping). They also experience a significant amount of friction, so the do not move easily. However, what makes them different from any other particle, presented so far, is their ability to drain mass from water particles. This happens whenever a water particle comes in contact with a dirt particle. The amount of water drained depends on how long the particles are in contact. On average the a dirt particle drain 10% of a water particle's mass in 0.1 seconds. As the dirt particles drain mass from water particles, their own water content increases. When this content exceeds 25% of their total mass they become mud particles. An example of dirt particles becoming mud particles is shown in figure 60.

Mud particles are the third type of mass particle used in this example. These particles experience attractive and repulsive forces similar to those of water particles, however at a slightly lower magnitude. Mud particles also have the ability to drain water from water particles. On average they drain 5% of a water

particle’s mass in 0.1 seconds. This drainage ability stops when the mud particle’s water content becomes greater than 75%. Mud particles may lose water mass to dirt particles and other mud particles. This loss is at the rate of 10% to dirt particles, and 5% to other mud particles. However, the loss only occurs if the mud particle has a greater water content than the particle with which it is in contact. The last feature of mud particles is their ability to dry out. They, like water particles, will lose water mass if the user turns on an evaporation process within the environment. The exact rate of water loss will depend on the strength of the evaporation.

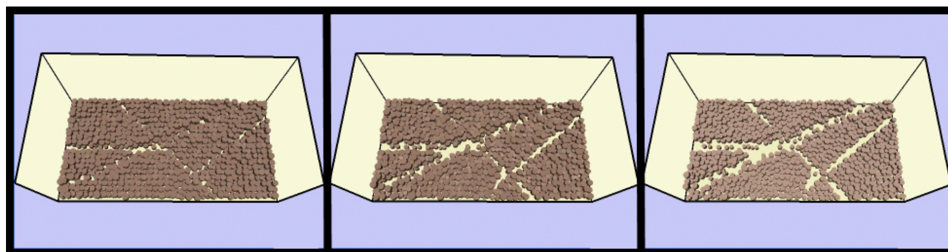


Figure 61: Mud drying and cracking.

As mud particles dry out, they will return to dirt particles when their water content drops below 20%. In this process it is expected the surface of the mud will fracture. To achieve this effect it is necessary to introduce some mechanism to separate the particles. An example of this effect is illustrated in figure 61.

To implement this cracking process it was hoped various repulsive or attractive forces could be used to force the drying mud particles to separate. However, as the mud particles dry at roughly the same rate, any induced forces tend to be symmetric among the totality of the particles. So if a particle tries to force away its neighbor on the right, there is almost always a particle to the right of it pushing it back to the left. Thus none of the particles separate.

To overcome this symmetry we introduced a force particle which is associated with a “mud” fracture pattern. These fracture-force particles are born randomly when mud particles begin drying. By random we

mean, about one out of one hundred mud particles will produce a fracture-force particle when it first drops below 20% water content. These force particles are locked in place – they will not move.

Once a fracture-force particle is born, it starts emitting a repulsive force. This force is based on a fracture pattern composed of line segments. These line segments do not extend very far, on average, no more than $10r$, where r is the average radius of any given particle. In effect any particle which is near a line segment is gently pushed away from it. As the mud dries, more and more of these fracture-force particles are generated. Thus the fracture lines begin to connect, and patterns emerge.

An advantage of implementing the cracking this way is it allows artistic effects to be introduced into the pattern. But it also allows patterns actually observed in the real world to be applied as well. Further it is non-deterministic as the exact placement of the fracture patterns cannot be determined ahead of time, as the water content of any given particle is randomized (though the user may manually place fracture-force particles if desired).

To prevent the fracture pattern from becoming permanent, the fracture-force particles are removed if they come into contact with any water particles. Thus, if after the mud dries and cracks, more water is introduced into the system, the cracks disappear as the dirt becomes mud and water particles remove the fracture-force particles.

Another item of note in this example is the display method is once again altered. While dirt and water particles are shown as spheres, the mud particles can be displayed as cylinders. This option is presented to better reflect the smoothness of the mud surface. Also, the astute observer may notice the particle sizes are not all the same. In fact, the particle size is indicative of the mass. Thus as a particle gains or loses mass, its radius increases or decreases proportionately. With these size changes, the color of the particles also changes. As dirt particles gain in water content they become darker. Likewise as mud particles lose water content they become lighter.

In sum, the dirt, water, and mud example illustrates the unique opportunities working with volpar systems provides. It also demonstrates yet another example of using force particles and another way to apply fracture patterns. Further it illustrates a unique way solids and liquids may interact when a volpar representation is used. Few, if any, other representation so easily transitions between solid and liquid form, where shape and form are dynamically determined. In fact, no other representation lends itself so easily to combining objects.

As a final remark, while the dirt never actually is bound into a rigid object, it is a simple task to determine the particle neighbors and establish particle connections to form it into a one. A possible use of this would be to give the user tools to mold the mud into a form and bake it into a rigid object. With the possibility of introducing fracture lines into the baking object, this would truly be unique.

6.5 Melting

Melting, by itself is rather easy to accomplish when given an object represented by particles. The general technique is to allow the connections between the particles to weaken. As the connections fail the particles roll off the object. In concept this is where we begin. However, as always, there are some details.

To melt ice, we must first have ice. This is accomplished by creating a cube of connected particles. The connections themselves are the same as in any other volpar object. However, each particle has a temperature value. When a particle reaches a certain temperature all of its connections are broken, and it transitions from a solid particle to a liquid particle. Aside from these small changes, our ice cube is the same as any other volpar object. While in this example it does not move, fracture, break, or explode, it could, using the same techniques described in previous sections.

With an ice cube created, we need a way to model the water it is to become. To accomplish this we use particles that are repulsed at close distances, attracted at medium distances, and indifferent to each other at long ranges. For this simulation, close range is defined as the radius of the particle, r. Medium range is a

distance from r to $5r$, and long range is beyond $5r$ units away. With these distances defined, we define the repulsive force F_r, to be a constant:

$$F_r = 1.05*m*g.$$

This is a deviation from more common repulsive techniques, but we are not very concerned if water particles overlap. In similar fashion, we define the attractive force, F_a, as:

$$F_a = 0.5*m*g,$$

where m is the mass of the particle and g is the global force of gravity. While this too is a linear function, we find it performs well enough and see no reason to introduce a quadratic.

Also, the relation to gravity might seem odd. It is used because we discovered relating our forces to gravity is useful in adjusting other attractive and repulsive forces. It should be understood, these forces, as well as the default value for gravity can be arbitrary, but are best selected near reality. This of course also requires the mass and radius of each particle to be set appropriately.

Once we have a representation of ice and water, we still need a method to introduce heat. The scene we want to model is one of an ice cube sitting near a desk lamp. This means the ice will never be in direct contact with the lamp, and thus the heat will need to transfer by convection, not conduction which is more common in a large number of heat transfer problems.

To model heat flow we use heat (force) particles. These particles are generated at the position of the lamp and given a velocity towards the ice cube. This velocity could be randomly set, but as there are no other objects for them to interact with, it is more pragmatic to at least nudge them in the direction we want them to go. Each heat particle is given a random amount of heat in the form of a temperature. As it travels, this

heat value (temperature) decreases. When it encounters an ice or water particle it transfers some of its heat. Exactly how much heat is transferred is based on Newton's Law of Cooling:

$$\mathrm{d}T/\mathrm{d}t = -k(T - T_a)$$

where k is a positive constant, T is the heat particle's temperature, and T_a is the temperature of the object encountered by the heat particle. The generalized solution to this is known to be:

$$T(t) = T_a + (T_0 - T_a)e^{-kt}$$

In applying this equation we measure temperature in degrees Celsius and time in seconds. We approximate heat loss by applying a change in temperature as described in the cooling law. For this, when the particle is traveling through the air we use a k value of -0.2776. When the heat particle encounters a water particle, we use a value of 0.005 and for an ice particle we use a value of 0.0005. These values produce the behavior we desire from the heat particles, and are within the realm of possible.

Another feature we introduce in this melting example, is an attractive property between the liquid particles and the ice particles. In general this is needed to obtain the observed effect of water clinging to the ice. To accomplish this we make the ice particle exert an attractive force, F_a, on any water particles between a distance of r and $5r$, where r is the radius of the ice particle. This force is defined to be:

$$F_a = 0.3*m*g,$$

Once all the above framework was established, we discovered in our early attempts the ice particles could not just become water particles. It produces a change in mass that is too sudden in appearance. To prevent this we allow an ice particle to melt in small pieces. So when its temperature first goes over 0° C, it loses one third of its mass and drops back to 0° C. The next time its temperature goes over 0° C, it loses half of

its mass and again drops back to 0° C. The third time, it actually melts completely. So, in the end, each ice particle becomes three separate water particles.

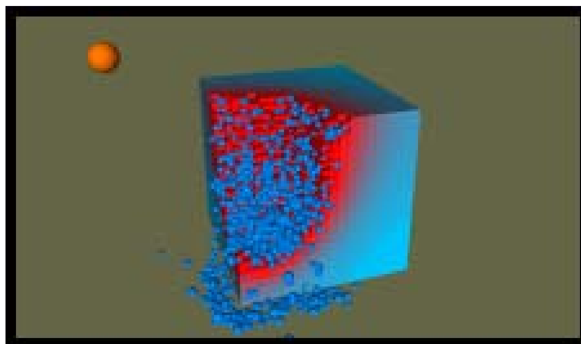


Figure 62: Ice melting displayed using spheres and cubes.

Another point to note in this example implementation is the choice of rendering. Specifically, we begin with a primitive based rendering. In this the ice particles begin rendered as cubes, so when they are stacked they form a larger cube. As they melt, and become liquid particles, we begin rendering them as spheres. An example of this rendering is found in figure 62.

While the primitive rendering is adequate for initially testing a simulation, there are many who will find it inadequate. To demonstrate our particle methods are capable of producing competitive images, we developed a method to save the state of each particle to a file for every frame. When the simulation completes we have a sequence of files, each representing a single frame of simulation.

Using these output state files, we load them into another program we created, which translates the states into a form that can be used to render the scene in a raytraced environment. In this instance we chose to use a public domain raytracer. An example of these renderings can be found in figure 63.

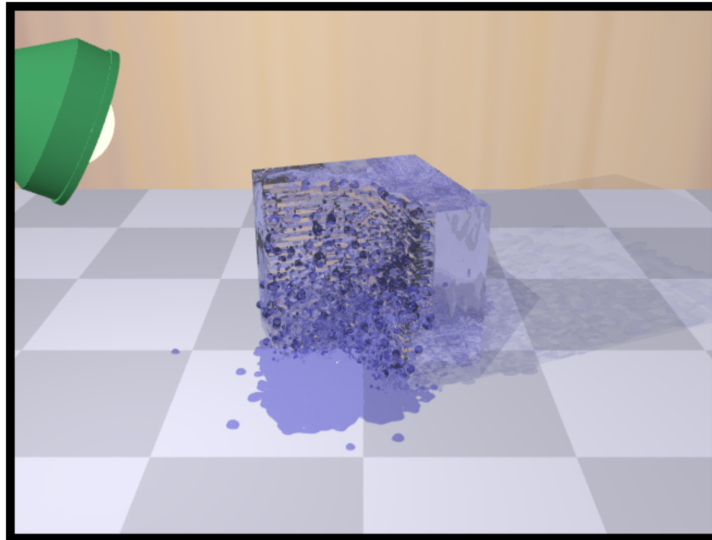


Figure 63: Raytraced ice cube melting.

With regard to timing issues it takes approximately the simulation about thirty to forty minutes to completely melt the ice cube. This was achieved using a Dell Inspiron 5150 laptop with a 3 GHz processor, 512 MB RAM, and an nVidia GeForce FX Go5200 graphics card. The raytraced images however took anywhere from thirty seconds to ten minutes to render per frame. In this process, of state storage and rendering, over 20 GB of disk space was used.

In sum, the results of the melting program are very good. It demonstrates a variety of points. Among these are the use of heat particles and the interaction of liquid with solid particles. It also shows the rendering of our particle objects is not limited to by the capabilities of the graphics library being used, and with proper planning may be exported with the intent to apply more photo-realistic methods.

7. CONCLUSION

So a theoretical framework for modeling objects and forces as particles has been presented. Details of abstractions and implementations have been provided. Examples of implementation have been shown. In the end, we have achieved what we sought to accomplish, demonstrating the features we sought to obtain.

Putting all the features and accomplishments together, we conclude a volpar framework allows the interaction of a variety of substances. These substances can be gaseous, liquid, solid, heterogeneous, or homogenous. The user is offered a variety of parameters to blend artistic desires with physically real behavior. All of this simplifies the process of creating animations where the behavior of the objects must be visually realistic, yet quickly accomplished. It allows fast proto-typing of such effects, with options for more advanced and time consuming renderings. Clearly, the volpar framework is useful in various multi-media fields. But more significantly, it has been shown ***a modeling framework based strictly on a particle representation of objects, and forces, is feasible and robust***.

So we have proven the hypothesis of this dissertation is true. However, a volpar modeling framework is not the end. It has some weaknesses and other modeling frameworks have some advantages, but it should prove useful to the modeling community and there are directions more research can be performed. Some possible directions to explore include: Improving the interface for commercial use. Implementing the framework to take advantage of the GPU. Applying action particles to other object representations.

In final conclusion, the hypothesis has been proven. This has been accomplished through theoretical development and proof of application. With successful completion, we now end.

REFERENCES

[1] Pankaj K. Agarwal, Leonidas J. Guibas, Herbert Edelsbrunner, Jeff Erickson,Michael Isard, Sariel Har-Peled, John Hershberger, Christian Jensen, Lydia Kavraki, Patrice Koehl, Ming Lin, Dinesh Manocha, Dimitris Metaxas, Brian Mirtich, David Mount, S. Muthukrishnan, Dinesh Pai, Elisha Sacks, Jack Snoeyink, Subhash Suri, Ouri Wolefson. Algorithmic issues in modeling motion. *ACM Computing Surveys (CSUR)*, 34(4):550-572, 2002.

[2] Karl Sims. Particle animation and rendering using data parallel computation. In *SIGGRAPH 90: Proceedings of the 17th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 405-413, 1990.

[3] Jon A. Bell and Scot Tumlin. Extremely explosive: Explosive effects. *3D Design*, 4(8):74-80, 1998.

[4] T. Purcell, C. Donner, M. Cammarano, H. Jensen, and P. Hanrahan. Photon mapping on programmable graphics hardware. In *Proceedings of the 2003 ACM SIGGRAPH/ EUROGRAPHICS Workshop on Graphics Hardware*, pp. 41-50, 2003.

[5] P. Kipfer, M. Segal and R. Westermann. UberFlow: A GPU-Based Particle Engine. In *Proceedings of the 2004 ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, pp. 115-122, 2004.

[6] T. R. Hagen, K. A. Lie and J. R. Natvig, Solving the Euler equations on graphical processing units. In *Proceedings of ICCS06, Lecture Notes in Computer Science (LNCS)*, 2006.

[7] William T. Reeves. Particle Systems - A technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91-108, 1983.

[8] D. S. Ebert, W. E. Carlson and R.E. Parent. Solid spaces and inverse particle systems for controlling the animation of gases and fluids. *The Visual Computer*, 10:179-190, 1994.

[9] Jos Stam. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 121-128, 1999.

[10] G. D. Yngve, J. F. O'Brien and J. K. Hodgins. Animating Explosions. In *Proceedings of ACM SIGGRAPH 2000*, pp. 29-36, 2000.

[11] Ronald Fedkiw, Jos Stam and Henrik Wann Jensen. Visual simulation of smoke. In *Proceedings of SIGGRAPH'01, Annual Conference Series*, pp. 15-22, 2001.

[12] David E. Breen, Donald H. House and Michael J. Wozny. Predicting the drape of woven cloth using interacting particles. *Computer Graphics (Proc. SIGGRAPH '94)*, 28(4):365-372, 1994.

[13] David Baraff and Andrew Witkin. Large Steps in Cloth Simulation. In *SIGGRAPH '98: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 43-54, 1998.

[14] Kwang-Jin Choi and Hyeong-Seok Ko. Stable but responsive cloth. *ACM Transactions on Graphics*, 21(3):604-611, 2002.

[15] Demetri Terzopoulos and Andrew Witkin. Physically-based models with rigid and deformable components. In *Proceedings Graphics Interface*, pp. 146-154, 1988.

[16] J. Christensen and J. Marks and J. T. Ngo. Automatic motion synthesis for 3D mass-spring models. *The Visual Computer*, 13:20-28, 1997.

[17] A. Barr, M. P. Cani, G. Debunne and M. Desbrun. Adaptive simulation of soft bodies in real-time. In *Computer Animation 2000 Proceedings*, pp. 15-20, 2000.

[18] Hanspeter Pfister, Matthias Zwicker, Jeroen van Baar and Markus Gross. Surfels: surface elements as rendering primitives. *In Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*. pp. 335-342, 2000.

[19] Greg Turk. Generating textures on arbitrary surfaces using reaction-diffusion. *Computer Graphics (Proc. SIGGRAPH '91)*, 25(4):289-298, 1991.

[20] Craig Reynolds. Flocks, herds and schools: A distributed behavioral model. *Computer Graphics (Proc. SIGGRAPH'87)*, 21(4):25-34, 1987.

[21] William T. Reeves and Ricki Blau. Approximate and probabilistic algorithms for shading and rendering structured particle systems. In *SIGGRAPH '85: Proceedings of the 12th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 313-322, 1985.

[22] T. Martin, Pep Español, M.A. Rubio and I. Zúñiga. Dynamic fracture in a discrete model of a brittle elastic solid particle dynamics. *Phys. Rev. E*, 61(6):6120-6131, 2000.

[23] L. Niemeyer, L. Pietronero and H. J. Wiesmann. Fractal dimension of dielectric breakdown. *Physical Review Letters*, 52:1033-1036, 1984.

[24] Tamas Vicsek. Pattern Formation in Diffusion-Limited Aggregation. *Physical Review Letters,* 53(24):2281-2284, 1984.

[25] J. Nittmann J. and H. E. Stanley H. E. Non-deterministic approach to anisotropic growth patterns with continuously tunable morphology: the fractal properties of some real snowflakes, *Journal of Physics A*, 20(17):L1185-L1191, 1987.

[26] Y. Taguchi. Aggregation of particles which move on deterministic trajectories with fractal dimension two. I. A simple and new model for DLA. *J. Phys. A: Math. Gen*, 21(22):4235-4240, 1988.

[27] Todd Reed and Brian Wyvill. Visual simulation of lightning. In *SIGGRAPH '94: Proceedings of the 21st Annual Conference on Computer Graphics and Interactive Techniques*, pp. 359-364, 1994.

[28] W. Wen and K. Lu. Electric-field-induced diffusion-limited aggregation. *Phys. Rev. E*, 55(3):R2100-R2103, 1997.

[29] B. Sosorbaram, T. Fujimoto, K. Muraoka and N. Chiba. Visual simulation of lightning taking into account cloud growth. In *Proceedings of Computer Graphics International 2001*, pp. 89-95, 2001.

[30] T. Witten and L. Sander. Diffusion-limited aggregation, a kinetic critical phenomenon. *Physical Review Letters*, 47(19):1400–1403, 1981.

[31] Paul Meakin. Diffusion-controlled cluster formation in two, three, and four dimensions. *Physical Review A*, 27(1):604-607, 1983.

[32] Richard Szeliski and David Tonnesen. Surface modeling with oriented particle systems. In *SIGGRAPH '92: Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 185 - 194, 1992.

[33] Greg Turk. Generating synthetic textures using reaction diffusion. *Technical Report TR-90-018*. University of North Carolina, Chapel Hill, 1990.

[34] Andrew Witkin and Michael Kass. Reaction-diffusion textures. *Computer Graphics (Proc. SIGGRAPH'91)*, 25(3):299-308, 1991.

[35] Alan Turing. The chemical basis of morphogenesis. *Philosophical Transactions of the Royal Society (B)*, 237:37-72, 1952.

[36] David Tonnesen. Dynamically Coupled Particle Systems for Geometric Modeling, Reconstruction, and Animation. Ph.D. Thesis. Department of Computer Science, University of Toronto, Toronto, Canada, 1998.

[37] Henrik Bekker. Molecular Dynamics Simulation Methods Revised. Ph.D. Thesis. University of Groningen, Netherlands, 1996.

[38] Dennis C. Rapaport. *The Art of Molecular Dynamics Simulation*, 2nd Edition. Cambridge University Press, Cambridge, 2004.

[39] Szymon Rusinkiewicz and Marc Levoy, QSplat: a multiresolution point rendering system for large meshes. In *SIGGRAPH '00: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 343 - 352, 2000.

[40] Matthias Zwicker, Hanspeter Pfister, Jeroen van Baar and Markus Gross. Surface splatting. *Proceedings of the 28th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 371-378, 2001.

[41] Lee Westover. Footprint evaluation for volume rendering. *Computer Graphics (Proc. of SIGGRAPH'90)*. 24(4):367-376, 1990.

[42] Michael Potmesil and Indranil Chakravarty. Modelling motion blur in computer-generated images. *Computer Graphics (Proc. of SIGGRAPH'83)*, 17(3):389-399, 1983.

[43] Lance Williams. Casting curved shadows on curved surfaces. *Computer Graphics (Proc. SIGGRAPH'78)*, 12(3):270-274, 1978.

[44] Gernot Schaufler and Henrik Wann Jensen. Ray tracing point sampled geometry. *In Rendering Techniques 200: 11th EUROGRAPHICS Workshop on Rendering*, pp. 319-328, 2000.

[45] Marc Alexa, Johannes Behr, Daniel Cohen-Or, Shachar Fleishman, David Levin and Claudio T. Silva. Point set surfaces. In *Proceedings of the Conference on Visualization '01*, October 21-26, 2001.

[46] Loren Carpenter. The a-buffer, an antialiased hidden surface method. *Computer Graphics (Proc. of SIGGRAPH'84)*, 18(3):103-108, 1984.

[47] Brian Lawn. *Fracture of Brittle Solids, 2nd edition*. Cambridge University Press, England 1993.

[48] Ted L. Anderson. *Fracture Mechanics: Fundamentals and Applications, 3rd edition*. CRC Press, Boca Raton, Florida, 2005.

[49] Alan Norton, Greg Turk, Bob Bacon, John Gerth and Paula Sweeney. Animation of fracture by physical modeling. *The Visual Computer: International Journal of Computer Graphics*, 7(4): 210-219, 1991.

[50] James F. O'Brien and Jessica K. Hodgins. Graphical modeling and animation of brittle fracture. In *Proceedings of ACM SIGGRAPH'99*, pp 137-146, 1999.

[51] James F. O'Brien, Adam Bargteil and Jessica K. Hodgins. Graphical modeling and animation of ductile fracture. In *Proceedings of ACM SIGGRAPH'02*, pp. 291-294, 2002.

[52] Demetri Terzopoulos and Kurt Fleischer. Modeling inelastic deformation: Viscoelasticity, plasticity, fracture. In *Computer Graphics (SIGGRAPH '88 Proceedings)*, 22:269–278, 1988.

[53] Michael Neff and Eugene Fiume. A visual model for blast waves and fracture. In *Proceedings of the 1999 Conference on Graphics Interface '99*, pp.193-202, 1999.

[54] Oleg Mazarak, Claude Martins and John Amanatides. Animating exploding objects. In *Proceedings of the 1999 Conference on Graphics Interface '99*, pp. 211-218, 1999.

[55] Jeffrey Smith, Andrew Witkin and David Baraff. Fast and controllable simulation of the shattering of brittle objects, *Computer Graphics Forum*, 20(2):81-91, 2001.

[56] Matthias Müller, Leonard McMillan, Julie Dorsey and Robert Jagnow. Real-time simulation of deformation and fracture of stiff materials. In *Proceedings of the Eurographic Workshop on Computer Animation and Simulation*, pp. 113-124, 2001.

[57] Si, Hang. TetGen A Quality Tetrahedral Mesh Generator and Three-Dimensional Delaunay Triangulator. http://tetgen.berlios.de/, TetGen software, Research Group of Numerical Mathematics and Scientific Computing, Weierstrass Institute for Applied Analysis and Stochastics, Mohrenstr. 39, 10117 Berlin, Germany, 2007.

[58] William D. Callister Jr. *Materials Science and Engineering: An Introduction*, 7th Edition. John Wiley and Sons, Inc., New York, 2007.

[59] D. Cohen and A. E. Kaufman. 3D Line Voxelization and connectivity control. *IEEE Computer Graphics & Applications*, 17(6):80–87, 1997.

[60] Y. K. Liu, B. Žalik and H. Yang. An integer one-pass algorithm for voxel traversal, *Computer Graphics Forum*, 23(2):167–172, 2004.

[61] Michael Garland and Paul S. Heckbert. Surface simplification using quadric error metrics. In *SIGGRAPH '97: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 209-216, 1997.

[62] Brent M. Dingle. Obtaining Fuzzy Representations of 3D Objects. *Technical Report TR-2005-11-06*, Computer Science Department, Texas A&M University, College Station, 2005.

[63] S. Torquato, B. Lu and J. Rubinstein. Nearest-neighbour distribution function for systems on interacting particles. *J. Phys. A: Math. Gen.*, 23:L103-L107, 1990.

[64] Trevor Johnson. After 300 years, computers facilitate solution to Kepler Stacking Problem. http://www.wsws.org/articles/1999/jan1999/math-j06.shtml, World Socialist Web Site, International Committee of the Fourth International, 1999.

[65] Thomas C. Hales. Cannonballs and honeycombs. *Notices of the AMS*, 47(4):440-449, 2000.

[66] R. Peikert, D. Würtz, M. Monagan and C. de Groot. Packing Circles in a Square: A Review and New Results, System Modelling and Optimization, In *Proceedings of the 15th IFIP Conference*, pp. 45-54, 1992.

[67] David Eppstein. Faster circle packing with application to nonobtuse triangulation. *Int. J. Computational Geometry & Applications*, 7(5):485-491, 1997.

[68] Gary. L. Miller, Shang-Hua Teng, William Thurston and Stephen A. Vavasis. Separators for sphere-packings and nearest neighbor graphs. *Journal of the ACM*, Vol. 44(1):1-29, 1997.

[69] David W. Boll, Jerry Donovan, Ronald L. Graham and Boris D. Lubachevsky. Improving dense packings of equal disks in a square. *The Electronic Journal of Combinatorics*, 7:R46, 2000.

[70] Harry Blum. A transformation for extracting new descriptions of shape. In *Models for the Perception of Speech and Visual Form*, W. Dunn, Ed., M.I.T. Press, Cambridge, MA, 1967.

[71] J. W. Brandt. Convergence and continuity criteria for discrete approximation of the continuous planar skeletons. *CVGIP: Image Understanding*, 59:116-124, 1994.

[72] Nina Amenta and Marshall Bern. Surface reconstruction by Voronoi filtering, In *SCG '98: Proceedings of the 14th Annual Symposium on Computational Geometry*, pp. 39-48, 1998.

[73] Ron Graham, B.D. Lubachevsky, K.J. Nurmela and P.R.J. Ostergard. Dense packings of congruent circles in a circle. *Discrete Math*., 181:139-154, 1998.

[74] B. D. Lubachevsky, Ron Graham and F.H. Stillinger. Patterns and structures in disk packings, *Periodica Mathematica Hungarica*, 34:123-142, 1997.

[75] Erich Friedman. Circles in squares. http://www.stetson.edu/~efriedma/cirinsqu/, 2004.

[76] Eric W. Weisstein. Circle packing. http://mathworld.wolfram.com/CirclePacking.html, MathWorld - A Wolfram Web Resource, 2004.

[77] Herbert Meschkowski. *Unsolved and Unsolvable Problems in Geometry*. Oliver & Boyd Ltd., Edinburgh and London, 1966.

[78] Brent M. Dingle. Volumetric Particle Separating Planes for Collision Detection, *Technical Report TR-2004-12-04*, Computer Science Department, Texas A&M University, College Station, 2004.

[79] Jerry B. Marion and Stephen T. Thornton. *Classical Dynamics of Particles and Systems, 4th Edition*, Saunders College Publishing, Harcourt Brace College Publishers, Fort Worth, 1995.

[80] R. Courant and D. Hilbert. *Methods of Mathematical Physics, Vol II*, Wiley Interscience, New York, 1962.

[81] W. E. Baker. *Explosions in Air*, University of Texas Press, Austin, 1973.

[82] Vladimir I. Arnold. *Mathematical Methods of Classical Mechanics*, Springer-Verlag, 1997.

[83] Matthew D. Roach. *Physically Based Simulation of Explosions*. M.S. Thesis, Visualization Sciences, Texas A&M University, College Station, 2005.

[84] Michael Neff. *A Visual Model for Blast Waves and Fracture*. M.S. Thesis, Department of Computer Science, University of Toronto, Toronto, Canada, 1998.

[85] Claude Martins, John Buchanan and John Amanatides. Visually believable explosions in real time. In *Proceedings of Computer Animation 2001*, pp. 237-247, 2001.

[86] Mikio Shinya and Alain Fournier. Stochastic motion—motion under the influence of wind. *Computer Graphics Forum*, 11(3):119-128, 1992.

[87] Georgios Sakas and Rudiger Westermann. A functional approach to the visual simulation of gaseous turbulence. *Computer Graphics Forum*, 11(3):107-117, 1992.

[88] Mario Botsch, Alexander Hornung, Matthias Zwicker and Leif Kobbelt. High-quality surface splatting on today's GPUs. *Eurographics Symposium on Point-Based Graphics*, pp. 17-24, 2005.

[89] W. E. Lorensen and H. E. Cline. Marching cubes: A high resolution 3D surface construction algorithm. *Computer Graphics (Proc. SIGGRAPH'87)*, 21(4):163-169, 1987.

[90] Nick Foster and Ronald Fedkiw. Practical animation of liquids, In *Proceedings of SIGGRAPH'01*, pp. 15-22, 2001.

[91] Douglas Enright and Ronald Fedkiw. Robust treatment of interfaces for fluid flow and computer graphics, In *Hyperbolic Problems: Theory, Numerics, Applications*. Edited by T. Hou and E. Tadmor, pp. 153-164, Springer-Verlag, New York, 2003.

[92] P. Lancaster and K. Salkauskas. Surfaces generated by moving least squares methods. *Math. Comp.*, 37(155):141-158, 1981.

[93] David Levin. *Mesh-independent surface interpolation, Geometric Modeling for Scientific Visualization*. Edited by Brunnett, Hamann and Mueller, Springer-Verlag, 37-49, Heidelberg, Germany, 2003.

[94] Nina Amenta andY. J. Kil. Defining point-set surfaces. *ACM Transactions on Graphics (Proc. of SIGGRAPH'04)*, 23(3):264–270, 2004.

[95] M. Alexa, J. Behr, D. Cohen-Or, S. Fleishman, D. Levin and C. T. Silva. Computing and rendering point set surfaces. *IEEE Transactions on Visualization and Computer Graphics*, 9(1): 3-15, 2003.

[96] Mark Pauly , Richard Keiser, Leif P. Kobbelt and Markus Gross. Shape modeling with point-sampled geometry. *ACM Trans. Graph.*, 23(3):641 - 650, 2003.

[97] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald and Werner Stuetzle. Surface reconstruction from unorganized points. In *SIGGRAPH '92: Proceedings of the 19th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 71-78, 1992.

[98] M. C. Lin and S. Gottschalk. Collision detection between geometric models: a survey. In *IMA Conference on Mathematics of Surfaces*, 1:602-608, 1998.

[99] Brent Dingle and John Keyser. Keyframing particles of physically based systems. In *TPCG05: Eurographics UK Chapter Proceedings*, pp. 11-18, 2005.

[100] Matt Anderson, Eric McDaniel and Stephen Chenney. Constrained animation of flocks. *Eurographics Symposium on Computer Animation*, pp. 286-297, 2003.

[101] Adrien Treuille, Antoine McNamara, Zoran Popovic and Jos Stam. Keyframe Control of Smoke Simulations, In *Proceedings of SIGGRAPH'03*, 22(3):716-723, 2003.

[102] R. Fattal and D. Lischinski. Target-driven smoke animation. In *Proceedings of SIGGRAPH'04,* 23(3):264-270, 2004.

[103] Scott N. Steketee and Norman I. Badler. Parametric keyframe interpolation incorporating kinetic adjustment and phrasing control. In *Proceedings of SIGGRAPH'85*, pp. 255-262, 1985.

[104] John Lassester. Principles of traditional animation applied to 3D computer animation, In *Proceedings of SIGGRAPH'87*, pp. 35-44, 1987.

[105] D. S. Ebert and R.E. Parent. Rendering and Animation of Gaseous Phenomena by Combining Fast Volume and Scanline A-buffer Techniques, In *Proceedings of SIGGRAPH'90*, 24(4):357-366, 1990.

[106] G. Sakas. Fast Rendering of Arbitrary Distributed Volume Densities. In *Proceedings of EUROGRAPHICS '90*, pp. 519-530, 1990.

[107] Nick Foster and D. Metaxas. Realistic animation of liquids. *Graphical Models and Image Processing*, 58(5):471-483, 1996.

[108] Thomas W. Sederberg and Eugene Greenwood. A physically based approach to 2-D shape blending. In *Proceedings of SIGGRAPH'92*, pp. 25-34, 1992.

[109] Lynne Shapiro Brotman and Arun N. Netravali. Motion interpolation by optimal control. *Computer Graphics*, 22(4):309-315, 1988.

[110] Michael F. Cohen. Interactive spacetime control for animation, *Computer Graphics*, 26(2):293-302 , 1992.

[111] P. Federl, P. Prusinkiewicz. A texture model for cracked surfaces, with an application to tree bark. In *Proceedings of Western Computer Graphics Symposium*, pp. 23-29, 1996.

[112] Jan Astrom and Jussi Timonen. Fracture of a brittle membrane. *Physics Review Letter*, 79(19):pp. 3684-3687, 1997.

[113] David Mould. Image-guided fracture. In *GI '05: Proceedings of the 2005 Conference on Graphics Interface*, pp. 219 - 226, 2005.

[114] Aurelien Martinet, Eric Galin, Brett Desbenoit and Samir Hakkouche. Procedural modeling of cracks and fractures. *Shape Modelling International*, pp. 346-349, 2004.

[115] Brett Desbenoit, Eric Galin and Samir Akkouche: Modeling cracks and fractures. *The Visual Computer*, 21:717-726, 2005.

[116] S. P. McJunkins and J. I. Thornton. Glass fracture analysis: A review. *Forensic Science*, 2(1):1-27, 1973.

[117] E. F. Rhodes and J. I. Thornton. The interpretation of impact fractures in glassy polymers. *Journal of Forensic Sciences*, 20:274-282, 1975.

[118] T.A. Michalske and B. C. Bunker. The fracturing of glass. *Scientific American*, 12:122-129, 1987.

[119] S. Gibson and B. Mirtich. A Survey of Deformable Modeling in Computer Graphics, *Technical Report TR-97-19*, Mitsubishi Electric Research Lab., Cambridge, MA, November 1997.

[120] T. W. Sederberg and S. R. Parry. Free form deformation of solid geometric models, *Computer Graphics*, 20(4):151-160, 1986.

[121] G. Camacho and M. Ortiz. Adaptive Lagrangian modelling of ballistic penetration of metallic targets. *Computational Methods in Applied Mechanics and Engineering,* 142:269–301, 1997.

[122] Yi Wu, Daniel Thalmann, Nadia Magnenat Thalmann. Deformable surfaces using physically-based particle systems. In *Proceedings of Computer Graphics International 1995*, pp. 205-216, 1995.

[123] S. Osher and R. Fedkiw. *Level Set Methods and Dynamic Implicit, Surfaces*. Springer-Verlag, New York, 2002.

[124] Demetri Terzopoulos, John Platt, Alan Barr and Kurt Fleischer. Elastically deformable models. *Computer Graphics*, 21(4):205-214, 1987.

[125] M. Müller , J. Dorsey, L. McMillan, R. Jagnow and B. Cutler, Stable real-time deformations,. *In Proceedings of ACM SIGGRAPH Symposium on Computer Animation (SCA) 2002*, pp. 49-54, 2002.

[126] John H. Heinbockel. *Introduction to Tensor Calculus and Continuum Mechanics*. Victoria, B.C., Canada, Trafford, 2002.

[127] Lin, Ming C., Lecture Notes for Comp 259. http://www.cs.unc.edu/~lin/COMP259-S05, Department of Computer Science, University of North Carolina, Chapel Hill, 2005.

[128] Demitri Terzopoulos, John Platt and Kurt Fleischer. Heating and melting deformable models (from goop to glop). In *Proceedings of Graphics Interface '89*, pp. 219–226, 1989.

[129] Mathieu Desbrun and Marie-Paule Cani. Simulation adaptative en temps et espace pour la simulation de matériaux très déformables. *RR-3829*, Rapport de recherche de l'INRIA-Rhone-Alpes, Equipe : IMAGIS, December 1999.

[130] David Baraff and Andrew Witkin. Large steps in cloth simulation. In *SIGGRAPH '98: Proceedings of the 25th Annual Conference on Computer Graphics and Interactive Techniques*, pp. 43-54, 1998.

[131] Zoran Kačić-Alesić, Marcus Nordenstam and David Bullock. A practical dynamics system, In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/EUROGRAPHICS Symposium on Computer Animation*, pp. 7-16, 2003.

[132] Xavier Provot. Deformation Constraints in a Mass-Spring Model to Describe Rigid Cloth Behavior. In *Proceedings of Graphics Interface '95*, pp. 147-154, 1995.

[133] Trina M. Roy. Physically Based Fluid Modeling Using Smoothed Particle Dynamics. M.S. Thesis, University of Illinois at Chicago, 1995.

[134] M. Müller, D. Charypar, M. Gross. Particle-based fluid simulation for interactive applications. In *SCA '03: Proceedings of the 2003 ACM SIGGRAPH/EUROGRAPHICS Symposium on Computer Animation*, pp. 154 - 159, 2003.

[135] G. Miller and A. Pearce. Globular dynamics: A connected particle system for animating viscous fluids. *Computers & Graphics*, 13(3):305–309, 1989.

[136] Olivier Genevaux, Arash Habibi and Jean-Michel Dischler. Simulating fluid-solid interaction. *Graphics Interface*, pp. 31-38, 2003.

[137] Mark Carlson, Peter J. Mucha and Greg Turk. Rigid fluid: animating the interplay between rigid bodies and fluid. *ACM Transactions on Graphics*, 23(3):377-384, 2004.

[138] M. Mueller, S. Schirm, M. Teschner, B. Heidelberger and M. Gross. Interaction of fluids with deformable solids. *Journal of Computer Animation and Virtual Worlds*, 15(3-4):159-171, 2004.

[139] James D. Foley, Andries Van Dam, John F. Hughes and Steven K. Feiner, *Computer Graphics: Principles and Practice in C*, Second Edition, Addison-Wesley, Boston, 1995.

[140] Brian Mirtich and John Canny. Impulse-based simulation of rigid bodies. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pp. 181-188, 1995.

[141] M. H. Overmars. Point location in fat subdivisions. *Inform. Proc. Lett.*, 44:261-265, 1992.

[142] S. Gottschalk, M. Lin and D. Manocha. OBB-Tree: A hierarchical structure for rapid interference detection. In *Proceedings of SIGGRAPH'96*, pp. 171-180, 1996.

[143] M.C. Lin and J.F. Canny. Efficient algorithms for incremental distance computation. *IEEE Conf. Robot. Autom.*, pp. 1008-1014, 1991.

[144] B. Mirtich. V-Clip: Fast and robust polyhedral collision detection, *ACM Trans. Graph.*, 17:177-208, 1998.

[145] M. Lin and S. Gottschalk. Collision detection between geometric models: A survey. In *Proceedings of IMA Conference on Mathematics of Surfaces*, 1998.

[146] P. Jiménez, F. Thomas and C. Torras. 3D Collision Detection: A Survey. *Computers and Graphics*, 25(2):269-285, 2001.

[147] P. M. Hubbard. Real-time collision detection and time-critical computing, In *Proceedings of the First ACM Workshop on Simulation and Interaction in Virtual Environments*, 1:92-96, 1995.

[148] I. J. Palmer and R. L. Grimsdale. Collision detection for animation using sphere-trees. *Computer Graphics Forum*, 14(2):105-116, 1995.

[149] Jonathan D. Cohen, Ming C. Lin, Dinesh Manocha and Madhav Ponamgi, I-COLLIDE: An interactive and exact collision detection system for large-scale environments. In *Proceedings of the 1995 Symposium on Interactive 3D Graphics*, pp. 189-198, 1995.

[150] Gino Van Den Bergen. Efficient collision detection of complex deformable models using AABB trees. *J. Graph. Tools*, 2(4):1-13, 1997.

[151] A. Garcia-Alonso, N. Serrano and J. Flaquer, Solving the collision detection problem, *IEEE Comput. Graph. Appl*., 14(3):36-43, 1994.

[152] P. M. Hubbard. Interactive collision detection. In *Proceedings of IEEE Symposium on Research Frontiers in Virtual Reality*, 1:24-31, 1993.

[153] S. Bandi and D. Thalmann, An adaptive spatial subdivision of the object space for fast collision detection of animating rigid bodies. In *Proceedings of EUROGRAPHICS'95*, pp. 259-270, 1995.

[154] Kedem Fuchs and Naylor. On visible surface generation by a priori tree structures. In *Proceedings of SIGGRAPH '80*, pp. 124-133, 1980.

[155] B. F. Naylor, J.A. Amatodes and W. C. Thibault. Merging BSP trees yields polyhedral set operations. *Computer Graphics (Proc. SIGGRAPH'90)*, 24:115-124, 1990.

[156] L. Greengard and V. Rokhlin. A fast algorithms for particle simulations. *Journal of Computational Physics*, 73:325-348, 1987.

[157] B. Carnahan, H. A. Luther and J. O. Wilkes. *Applied Numerical Methods*. Wiley, New York, 1969.

[158] J. J. Monaghan. Smoothed particle hydrodynamics. *Annual Review of Astronomy and Astrophysics*, 30:543-574, 1992.

[159] M. Desbrun and M. P. Gascuel. Animating soft substances with implicit surfaces. *Computer Graphics*, 29:287-290, 1995.

[160] M. Desbrun and M. P. Gascuel. Smoothed particles: A new paradigm for animating highly deformable bodies. In *Proceedings of EG Workshop on Animation and Simulation*, pp. 61-76, 1996.

[161] M. Muller, B. Solenthaler, R. Keiser and M. Gross. Particle-based fluid-fluid interaction. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, pp. 237-244, 2005.

[162] R.A. Gingold and J. J. Monaghan. Smoothed particle hydrodynamics: theory and application to non-spherical stars. *Mon. Not. R. Astron. Soc.*, 181, 1977.

[163] L. B. Lucy. A numerical approach to testing of the fission hypothesis. *The Astronomical Journal*, 82(12):1013-1024, 1977.

APPENDIX A

EXTENDED BACKGROUND

A.1 Object Representations in Modeling

Within this section is an examination of existing object representations. It will be seen, none of the basic representations inherently consider the material properties of the object. Thus modeling such things as fracturing or breaking of objects is not easily achieved. It will also be noted, most of these representations fail to allow for “fuzzy” or deformable objects, such as water, clouds or clay, very well. Further it will be obvious, these representations do not inherently allow objects to change state and few allow them to change at all. Thus, to model such objects and the dynamic changeability of the objects, modification of the basic object representation is necessary.

A.1.1 Basic Objects

While there are a multitude of ways to represent 3D objects in a computer, in general, they can be placed into three categories: Raw Data, Surfaces and Solids. These three categories can then be used to classify various representations as shown in table A1.

Raw Data	Surfaces	Solids
Point Cloud	Mesh	Voxels
Range Image	Subdivision	BSP Tree
Polygon Soup	Parametric	CSG
	Implicit	Sweep

A1 - Basic object types.

Theoretically each of these representations is equivalent to any other. Specifically it is possible to model any geometric shape using any of the above representations. It is also possible to perform any geometric operation on an object represented in any of the above forms. However, each representation has advantages and disadvantages in terms of computational complexity, memory requirements and numerical accuracy. Likewise each may be compared in terms of how easily it is to acquire or create an object in a given form, how much hardware acceleration is possible when using a representation and how much existing software supports a given representation. Beyond this each may also be compared in terms of its usability in object design versus its usability in a physically based modeling and animation.

A.1.1.1 Raw Data

Raw data object representations include point clouds, range images and polygon soup representations. Each representation is similar in there is no explicit surface or connectivity defined within the representation itself.

Point clouds are unstructured sets of 3D point samples. They may be created by range finders, random samplings, particle systems or other methods. For such representations to be useful surface reconstruction techniques must be applied to obtain a well defined object surface. The complexity and shear data size of these representations has kept them from being popular.

Range images are created from structured range scanners. Instead of storing points in a local coordinate object space, they store distances of surface points from the scanner. Thus the object is represented as a set of points based on a specific, unchanging viewpoint. Multiple range images can be used to create point clouds.

Polygon soup models are sets of unstructured polygons. These are usually created by interactive modeling systems. They may also be created from various processes performed on range images, or point clouds, in

an effort to create a surface of polygons by joining points based on proximity to one another. These representations are usually temporary or transitory in creating more structured representations.

A.1.1.2 Surface Representations

In most scenarios the only part of an object of interest is its surface. Surfaces of objects may be represented using meshes, subdivision schemes, parametric and implicit equations. Each representation has some advantages over the others, but the most prevalent is a simple mesh composition of vertices, edges and faces. Such a representation is often referred to as a boundary surface representation or B-rep.

In mesh representations objects are described as a connected set of polygons, usually triangles. These representations are often created in modeling software under the guidance and control of the user. Almost any graphical based software will support some form of mesh representation.

Subdivision schemes are based on mesh representations. In most cases this representation begins with a coarse mesh approximation of the object which is refined to a smoother representation by subdividing the polygons into smaller and smaller polygons. On occasion this method is reversed, particularly in level of detail applications, where the object begins as a very fine representation at near viewing distances and then becomes coarser and coarser as distance from the object increases. Various object and image compression techniques using wavelet or Fourier methods may also qualify as subdivision representations.

Parametric representations of objects are usually defined as tensor products of spline patches where care is given to ensure continuity across the patches. These representations include B-spline and Bezier forms of objects. They are useful in the design of smooth objects and it is possible to create mesh representations from parametric representations.

Implicit representations of objects describe the surface of the object as a function such that $f(x, y, z) = 0$. This representation is most useful in determining object collisions as it is easy to calculate whether a given point is inside, outside, or on the surface of the object. It is difficult to design complex objects using such a representation, however it is possible to convert parametric representations into implicit form.

A.1.1.3 Solid Representations

There are four common ways to represent a solid object, where solid implies the interior of the object is also represented in some fashion. The degree and usefulness of how the interior of the object is represented varies between these methods. Specifically it will be seen the constructive solid geometry (CSG) and sweep forms superficially seem no better than surface representations, yet such representations do offer information about the interior as well as the surface of the object and thus are solid representations.

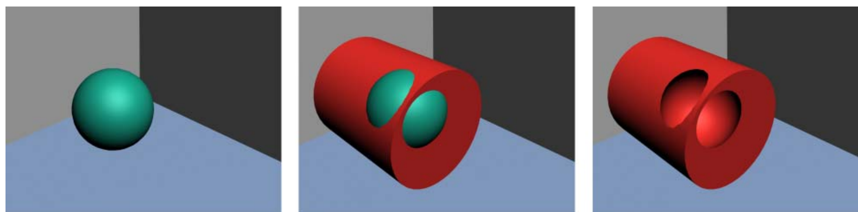
A voxel representation is a uniform grid of volumetric samples. The samples are often indicative of the density of the object at a given point within it. These samples are usually obtained from MRI or CAT scans. In this representation the object (and its surrounding space) is viewed as a set of small cubes of possibly varying densities. The display of the object is done using transparency and coloring variation to represent different densities within the object. This form of representation is common in medical imaging. To ultimately be useful the grid resolution must be exceptionally fine and thus the data storage required for its use is immense. Within the representation itself there is no connectivity or material properties, though in most cases they can be deduced or calculated, given proper knowledge of the object scanned. There is also no explicit description of the object surface.

Binary space partitioning (BSP) trees offer another possible way to represent solid objects. In these representations space again is partitioned into cells. However the cells need not be uniform in shape or size. Each cell is then associated with a value to indicate if it is a solid or empty cell. There are many variations of this technique. In most cases the BSP trees are created using mesh (polygonal) models of

objects. This method tends to be used to store the description of space occupancy rather than individual objects.

Another option to represent solid objects is in a sweep form. In this type of representation a 2D shape is described, usually by equation, and then rotated around a defined axis. Thus 2D shapes are swept through 3-space to generate a volume. For example half of the unit circle ($\theta \in [0, \pi]$) may be rotated around the x-axis to produce a solid sphere. In some ways, this is backwards in design to lathing of wood. In this process it is assumed that all points within the swept region are part of the object's interior. This is more an implied solid than the previous two methods, and offers no way to represent any material properties within the object, such as varying density or composition.

The most commonly used method of representing solids is found in constructive solid geometry (CSG). In this methodology an object is represented as a set theoretic Boolean expression of primitive solid objects. These objects are often stored in a tree format, which describes the primitives and how they are joined together using Boolean expressions. The primitives are typically cones, cubes, cylinders, spheres and tori. An example of a sphere being subtracted from a cylinder is shown in figure A2. This representation is used in various computer aided design software. However, like the sweep forms, the interior of the object and all of the material properties are assumed rather than explicitly represented.



A2 - Boolean CSG subtraction.
The volume of the sphere is subtracted from the cylinder.

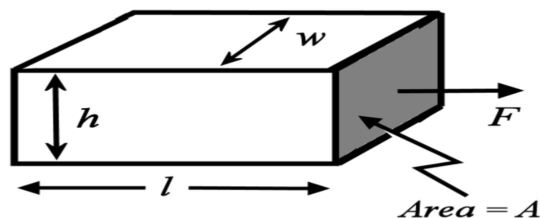
A.1.2 Deformable Objects

While many objects used in graphics and animation are rigid, there has always been an interest in deformation. This is clearly necessary to model such things as rope, hair, or cloth, to name just a few. So, naturally, a variety of methods have been developed to model deformable objects. An excellent survey of such methods was given in 1997 by Gibson and Mirtich [119]. It is noted there are several ways to categorize the approaches taken. One such distinction is in the underlying object representation used. In this distinction there are two predominant techniques: those based on adaptable meshes [120, 121] and those which use some form of underlying particle system [32, 122, 123]. Another method of classification is in the design. Here also there are two techniques: those designed for deformations created by a user and those designed to represent physically based deformations. The non-physically based methods would include techniques using splines and patches, free-form deformations and subdivision surfaces [120]. However, in the context of this dissertation we are interested in physically based deformations and will focus on those techniques.

To accurately model physically based deformations a continuum model of the deformable object must be created. However, as this model will be implemented on a computer, it will eventually be approximated. Before we discuss the approximations we will begin with a presentation of what is meant by a continuum model with respect to deformable objects. The reader may find further information on this topic in the papers by Terzopooulos et. al. [124], Gibson and Mirtich [119], or Müller et. al. [125]. The book by Heinbockel [126] may also be useful.

Continuum mechanics is defined as the study of how external influences affect the behavior of materials. While these influencing factors include such things as gravitational forces, temperature, chemical reactions and electric phenomena, we are interested in mechanical forces.

To be considered a continuous media an object must be viewed as a collection of small material parts interconnected by internal forces. Effectively this is the representation of atoms making up the object material. When dealing with continuous media, we observe there are two types of deformable material. There are elastic materials, which will return to their original form once the deforming forces are removed and there are plastic materials which will remain at least partially deformed when the deforming forces are removed. For simplicity we will only consider elastic materials, though most everything applies to plastic materials as well. When the relation between applied forces and material displacements is linear, then the material is called a linear elastic material. Most deformable objects implemented on computers are assumed to be linear in nature.



A3 - Rubber Band Segment.
Pulling on the end of this rubber band produces a stress of F/A.

There are two basic concepts which may cause deformation, stress and strain. Stress is defined as force divided by the area to which the force is applied. Strain is defined as change in length divided by original length. So if we consider a small segment of a rubber band experiencing a force, as shown in figure A3, we see we have stress defined by:

$$\frac{F}{A}$$

and strains defined by:

$$\frac{\Delta l}{l} \qquad \frac{\Delta h}{h} \qquad \frac{\Delta w}{w}$$

It is worth noting in the case of linear elastic materials, where all forces are one dimensional, Hooke's law offers a relationship between the stress and strain. Specifically, it states the stress is proportional to the strain in the stretch direction. Specifically:

$\frac{F}{A} = k\left(\frac{\Delta l}{l}\right)$, where k is the Young's modulus (spring constant) for a given material.

Notice this is slightly different than what is usually presented in simple spring descriptions as we are considering a volume of material. Thus the area acted on by the force is being included. Another part of Hooke's law, usually not mentioned in simple spring analysis, deals with the strain contraction perpendicular to the stretch direction. In this it is known the strain contraction is identical in both the width and height directions and is proportional to the strain in the stretch direction. Explicitly:

$\frac{\Delta h}{h} = \frac{\Delta w}{w} = -v\frac{\Delta l}{l}$, where $0 < v < ½$ and v is called Poison's ratio, which varies by material.

The significance of this will become apparent shortly as it is used in our first method of modeling deformable objects. However, before proceeding to the methods there remain a few small points to make about using continuum models to represent deformable objects. First, there are three axioms of continuum mechanics to be applied to deformable objects [127]:

1. A material continuum remains continuum under the action of forces.
2. Stress and strain can be defined everywhere in the body.
3. Stress at a point is related to the strain and rate of change of strain with respect to time at the same point.

Together these imply we may consider stress at a given point to be locally defined by the deformation near that point and the relationship between stress and strain may be viewed separately.

With the above axioms in mind and a general understanding of continuum mechanics it is now possible to describe a continuum model of deformable objects in terms of forces and energy. Specifically, a deformable object may be viewed as a continuum model based on the equilibrium of external forces acting on the object. Such an equilibrium is achieved when the object's potential energy is at a minimum (e.g. a spring is at its rest length).

To represent this description the potential energy of a deformable object can be denoted:

$$\Pi = \Lambda - W$$

where Λ is the total strain energy of the object and W is the work done on the object by external loads.

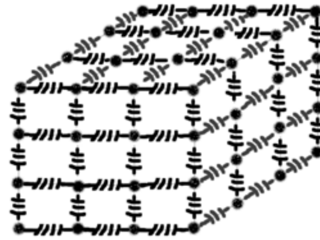
To apply the above model, deformation of the object is expressed as a function, D, of the material displacement over the object. Thus both Λ and W can be expressed in terms of the deformation of the object. Once this is done the potential energy of the object is at a minimum when $d\Pi / dD$ is zero.

Of course, to implement the above, the deformable object must be discretized in some fashion. There are three common approaches to approximate continuum models of deformable objects which can be implemented on a computer. The first is a spring-mass model. The second is based on finite differencing methods, and the third is based on finite element methods.

A.1.2.1 Point-Mass Spring Systems

The first method to model deformable objects uses point masses connected together by a set of springs. The objects within this system usually take the form of small cubes, where the vertices of the cubes are point masses and the edges are springs. An example of such a system is pictured in figure A4. Additional diagonal and cross diagonal springs may be used for extra support. Torsional springs may also be used to restrict the change in angle between the edge springs.

Using this method it is difficult to properly model continuum properties and computational issues, such as stability, often arise. However, these systems are easy to understand and have straightforward implementations. As they qualify as a particle based implementation of deformable objects we will give a brief presentation of them here. This is done for contrast in later sections.



A4 - Spring, point-mass cube.

In most cases the springs are created according to a system based on Hooke's law. Specifically, for a given spring and a given particle, the following definitions are used:

x = the position of a particle,
$v = x'$ = the velocity of the particle,
$a = v' = x''$ = the acceleration of the particle,
m = the mass of the particle,
R = rest length of the spring,
k = spring constant (stiffness of the spring),
b = damping constant (friction of the spring),
s = stretch (displacement) of the spring = $x - R$, notice $s' = x'$ and $s'' = x''$.

Thus the force generated by the spring on the particle is:

$$F_{spring} = -k * s.$$

In addition there is a damping force which is proportional to velocity and resists motion. This force is:

$$F_{damping} = -b*v.$$

Thus the total force is:

$$F = F_{spring} + F_{damping} = -ks - bv$$

These forces are then combined with Netwon's law of motion, $F = ma$ to give us $ma = -ks - bv$, or rather:

$$x'' = (-ks - bv) / m$$

However, to solve this using numerical methods the above second order differential equation must be expressed as a set of first order differential equations. This is a straightforward process and the set of equations used is:

$$x' = v$$
$$v' = (-ks - bv) / m$$

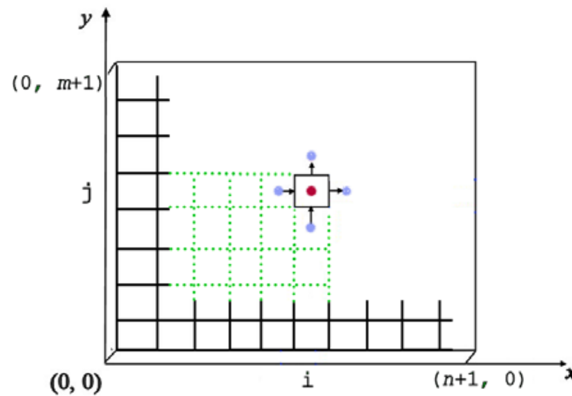
Where the initial values of x and v are assumed known and the motion of the spring system is simulated using discrete time steps.

This approach has been used successfully to model deformable objects implemented using particle based systems [12, 49, 122, 128, 129] and using finite differencing methods [124]. However it suffers numerical instability when there are large number of particles or when the stiffness of the springs is too great and the time step is not decreased to accommodate the greater stiffness and numbers. This is further complicated by computational errors within the computer when dealing with such small time steps [130, 131]. A number of solutions have been proposed to overcome this difficulty such as constraining the motion [132]

or using adaptive integration methods. Yet, this method remains difficult to implement on a large scale due to the nature of the spring systems.

A.1.2.2 Deformable Finite Methods

Another approach to modeling deformable objects as continuum models is to use finite differencing methods (FDMs) and finite element methods (FEMs). A detailed discussion of finite methods will be presented in a section A.5, so here we will be brief. Understand while this presentation focuses on the limitations of these methods, they are well designed and useful.



A5 - Typical FDM grid of cells.

In applying FDMs to model deformable objects, the objects are usually broken into cubic cells. Thus the object is represented as a regular, structured grid of cells, as shown in figure A5. The structure may limit the use of an FDM by restricting the accuracy with which it can represent objects. Yet, once the grid is created, the properties of each cell are then modeled by propagation through each cell by examination of its neighboring cells. This is a point-wise approximation.

Finite element methods (FEMs) are more versatile than FDMs and are often more accurate. However, they are computationally expensive and require sophisticated mathematical operations to achieve the necessary equations to be used in the computer implementation. In theory FEMs should best reflect the continuous nature of a material.

The goal of an FEM is to find an approximation of a continuous function which satisfies some equilibrium expression based on deformation, e.g. $d\Pi/dD = 0$. In general they will divide an object into elements of some primitive shape, e.g. tetrahedron. An equation which would satisfy the equilibrium expression is then derived for that shape. The solution to the expression is then carried out for each element, however special care is given to maintain continuity between elements along their boundaries and at their shared nodes. This is usually done using interpolation based ideas.

In modeling deformable objects, the above is carried out by deriving an equilibrium equation from the potential energy equation in terms of material displacement. An appropriate finite element and corresponding interpolation function is then selected (e.g. tetrahedrons and linear interpolation between nodes). It should be understood this derivation and selection is done by hand, and not by the computer. Then for each element the components of the equilibrium equation are restated in terms of the selected interpolation functions. The derived set of equations for each element is then placed in a single system of equations. Adjustments are made for any other forces and effects and the system is solved by the computer to determine the displacement of the nodes of each element. From this solution the displacement of any point within the deformable object may be determined as needed. Examples of modeling deformable

objects using FEMs may be found in the work done by Terzopoulos et. al. [124] or O'Brien and Hodgins [50].

A.1.3 Amorphous Objects

Another type of object to be modeled is one of amorphous shape. These objects would include such things as water or smoke. The distinction between these objects and deformable objects will be stated simply: amorphous objects have significantly less structure.

Both liquid and gas behavior can be modeled by the physical interaction of particles or by using mathematical methods such as those based on Navier-Stokes equations or smoothed particle hydrodynamics [133, 134]. In general terms, the behavior of the substance is modeled as the motion of density or mass of the substance.

While the underlying model of motion is complex, the issue of displaying the substance is also nontrivial as there are differences in the visible characteristics of gases and liquids. Liquids have a defined, often reflective, surface. Gases, however, have no distinct surface. Both have unique transparency effects. This being the case, smoke is often displayed using some form of particle system, where the particles have varying degrees of transparency. Water can be displayed in a similar fashion, however is more often displayed as a highly dynamic surface mesh. The difficulty in modeling water in such a way is apparent when the water splashes into itself. Meshes do not readily allow changes in topology.

This continual change of topology with no definable structure is a major challenge of modeling amorphous substances. Such issues can be addressed by using blob based methods [135]. They can also be alleviated by using marching cube, iso-surface, and level set methods [89-91]. However these methods are complex to implement within a modeling environment which may have a multitude of objects in various phases. This complexity is mostly in maintaining unique display methods for each type of object, while also maintaining code to handle the interaction of each type of substance. In particular if a substance has no well defined surface, how does it collide with a substance that does? This question has yet to be definitively answered, though various solutions have been proposed [136-138].

A.2 Display and Rendering of Objects

A great deal can be said about displaying and rendering objects in terms of computer graphics. However, this section is meant only to offer a glimpse into the ideas behind such rendering. The topics presented here are those most relevant to the object display methods used in the context of this dissertation. In section A.2.1 we establish a surface is required to display an object. While this may be accomplished in a variety of ways, surface triangulation is the most prevalent. In section A.2.2 we note we have a special interest in the methods of splatting and surfel techniques, as these adapt nicely to our particle systems.

A.2.1 General Display Concepts

No matter how the object is modeled it must also be displayed, on a 2D screen. To accomplish this some form of surface representation must be derived or approximated from the given object representation. For objects stored as raw data, they may be displayed as points, circles, spheres, or may have an interpolated surface derived for them, which is then triangulated (see also 2.2.2). For surface representations a triangulation of the surface is often used. In the case of volume representations, an equivalent surface representation must be calculated. This representation is then triangulated and the object displayed. In the end, the objects are always displayed as their surface, which is defined to be a set of polygons (usually triangles), with vertices defined in a 3D coordinate system.

The nature of the surface set of polygons will vary. If lighting is to be accurately modeled each surface polygon must be oriented. If the surface is to appear smooth, the polygons must meet continuity specifications at their boundaries, or be blended across their boundaries to smooth the edges. If the objects are transparent, special blending may be used.

Regardless of object representation, the basic method of display is described by the Painter's Algorithm. In this technique the objects farthest from the viewing plane (canvas location) are drawn first. Then, in sequential order, objects which are closer are drawn. Thus objects which are far away from the viewer will be covered up by objects that are more near. While this is easy to implement it is rather slow and may draw a large number of objects which, in the end, are not at all visible. It also may encounter cases where it is unable to determine which surface polygon is in front of another.

To overcome the problem of determining which polygon is in front of any other a technique known as Z-buffering may be used. In this method, as each polygon is drawn to the screen, every pixel of that rendering is assigned a depth value. Thus each (colored) pixel of the view plane has a depth value. As depth may be thought of as in the "z-direction" these depth values are stored in the "z-buffer" array. Thus when another polygon is drawn to the screen it too generates a depth for each of its pixels. If the new polygon would be rendered onto a pixel of the screen which is already colored, the depth of the new polygon's pixel is compared to the depth of the current pixel of the screen. Whichever pixel has the smaller (nearer) value becomes the current pixel. Thus rather than compare each polygon against another, a simple comparison is made pixel by pixel.

Another issue in rendering and displaying objects is how to process the light sources. As object's cannot be seen without light this is a rather critical function. In all cases this requires some form of orientation to be specified for each surface polygon. This orientation and the position and type of the light source is used to determine if, and how much, light falls on the polygon. This in turn allows the surface to be shaded, possibly with specular highlights and shadows. The exact details of how this is accomplished are beyond the scope of this dissertation. For further information the reader is directed to the numerous books available on rendering such as the one by Foley et al. [139].

In sum, the Painter's Algorithm along with the Z-buffering Algorithm form the basis of most rendering techniques used today. There are many variations and most are now implemented on the graphics cards. Also, the programming details of most of the rendering methods have been migrated into libraries such as OpenGL and DirectX.

For more photo-realistic scenes a technique referred to as ray-tracing may be used. In this method, for every pixel of an image, a ray is sent out from the camera. Each ray moves through the scene, bouncing off objects encountered. In this fashion the color of each pixel is determined. It is important to note, even this method of rendering requires at least the surface of each object to be identified. For more details on this method, we again refer to the numerous books on the subject, such as the one by Foley et al. [139].

The majority of the work done in this dissertation uses the built-in capabilities of the OpenGL programming library. This simplifies the programming effort and makes use of hardware capabilities when possible. In most cases we use only a global, ambient, light source and at most one focused point light source. In very few cases do we implement shadows. For more photo-realistic renderings we export our scene data to a series of text files, which are then rendered using the free raytracing program POV-Ray. Further details of our rendering methods may be found in section 5, where finding the surface of an object to render and how to display it becomes a topic of interest.

A.3 Modeling and Simulation

In modeling for animation and gaming purposes there are two ways to conceptualize objects within the modeled world. The first is to view every object as a distinct object in the world space. In this approach every object has its own properties and its own representation. This allows every object to be unique. The second approach is to recognize many of the objects within the world are basically the same. For example a tree in the background more or less appears the same as any other tree in the background. Likewise various enemies will physically appear to be identical. This allows various objects to be displayed using the same model. This in turn leads to a distinction between object space and world space.

Both approaches have advantages. The first approach allows a very detailed experience of the modeled world as every object may be examined and seen as different from any other. Unfortunately, this requires a large amount of memory and may require overspecialization of code to handle the unique characteristics of each object. The second approach simplifies a majority of the objects at the cost of losing such unique appearances. However, this simplification conserves memory and may allow the modeling engine to be streamlined to handle only a few exceptional cases. It should also be noted in this second method the object representations never actually move. Their location and orientation is simply stored as a transformation matrix from object space into world space. This is significantly different than in the first approach where each object would have its position and orientation described directly in world space.

For the majority of this dissertation we will approach our modeling space as necessarily containing unique objects, as each may change in appearance at any time. Thus conceptually we will be using the first approach described in the above paragraphs. Also of significance is that every object will be viewed as a composition of particles. Thus the only real object is a particle and each particle will be unique.

Within the next few subsections we expand upon some of the concepts and methods necessary to develop a modeling paradigm. This includes the necessary background in rigid body and particle motion concepts needed in later sections. Among the topics discussed are physically based modeling, integration methods, collision detection and various space partitioning techniques.

A.3.1 Physically Based Modeling

The goal of physically based modeling can be stated as: the design and implementation of methods which allow the creation of models capable of automatically behaving in a physically realistic manner. A reason for creating such models is to simplify the animation process. Unfortunately it is difficult to model materials exactly and the computational requirements to solve the systems of equations developed in such modeling efforts can be quite demanding.

An enormous amount of work has been spent in physically based modeling. With a small amount of investigation, it is easy observed almost every implementation follows the same basic design where the fundamental key is time. Specifically every modeling system has a function which is called every *n* milliseconds. A trivial example of such a function is shown in table A6.

Description	Every *n* milliseconds OnTimer() is called by the modeling system. This function is expected to update the state of the objects within the system and display the objects to some pre-specified window.

Pseudo-code	OnTimer()
	{
	    world.Update(time);
	    world.Display();
	}

A6 - General form of a timing function for modeling.

Thus the topmost level of a modeling engine is just two functions. However within these two functions a great deal of work must be performed. In general the update function will be passed a time parameter, whether this is the current time or a requested simulation time step varies. The function itself will then calculate the new positions and velocities of every object within world space. In these calculations is where the automatic behavior of the objects is found. For within them, gravity, collisions, and other forces act on the objects as determined by the rules of the engine. Of course, determining the cumulative effects of such forces on every object requires some skill in model design as well as programming skill. In the next few sections we will offer a glimpse into the basic ideas behind these tasks. In table A7 is a pseudo-

code example to demonstrate a layout of a basic update routine, there are other possible implementations. This is offered to give the reader an idea of the conceptual layout being used to design the engine of the modeling paradigm.

Description	The Update() function of a modeling engine is designed to move the objects of the system, across a specified discrete time step. Within this process it is assumed the motion of the objects is ‘valid.’ So the motion should appear continuous and collisions should be realistic in results.

Pseudo-code	void Update(double max_time_step)
	{
	    Set the time step, dt, to max_time_step.
	    Set time_passed equal zero.
	    While (time_passed < max_time_step)
	    {
	        Copy the current state (pos., vel.) of each object to an array, s1.
	        Calculate the forces (accelerations) acting on each object.
	        Store the derivative state (vel., accel.) of each object to an array, d1.
	        Call an ODE solver to advance the states in s1 by dt.
	        Check for collisions.
	        If there is at least one collision then
	        {
	            Set dt equal to the time of the first collision.
	            Flag object(s) in collision for collision force processing.
	        }
	        Else
	        {
	            Increment time_passed by dt.
	            Set dt equal to max_time_step – time_passed.
	            Copy the new states of s1 back to the objects.
	        }
	    }
	} *// end Update*

A7 - Basic update routine.

Of note in table 4’s pseudo-code is the copying of the state of every object. In some cases this may be wasteful. However it is beneficial in two ways. The first is that it allows the state of the objects to always be viewed as an array of values regardless of what data structure is used to represent the objects as a whole. The second benefit it offers applies only if there is a fast way to perform array (vector) arithmetic. This benefit may be seen during the call to the ordinary differential equation solver, in the updating of the state vector after the ODE solver call, or in the collision detection routine. This design feature is worth mentioning as it can be applied to the graphics processing units (GPUs) of current video cards [5, 88].

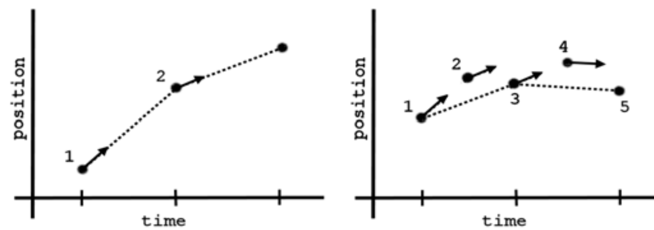
The display function is easier to write, particularly when dealing with just rigid, polygonal bodies. However for particle systems it may require the particles to be sorted by distance from the user, with careful blending measures and sprite mappings. If isosurfaces, splatting or level set methods are being used more coding might be required [40, 123]. Should there be a combination of systems or objects requiring special display techniques, then the complexity of this function will further increase.

The true difficulty of these two functions is they must complete with exceptional speed, as they will likely be called again in a few milliseconds. Should they not complete, then the simulation may be seen as

running too slow. Two major parts of the update function are the integration methods (ODE solver) and the collision detection routines, which will be discussed below.

A.3.2 Integration Methods

In modeling multiple objects an array of state vectors must be maintained. In most cases this state array represents the position, orientation and velocities of all the objects within the system. To update the state of the objects it is assumed a function to calculate the forces acting on the objects exists. Thus the accelerations of the objects may also be calculated from $F = ma$ (or equivalent interpretation). While this is useful these calculations are technically instantaneous values. Thus to actually move the objects these calculations would need to be performed an infinite number of times over infinitely small time steps.



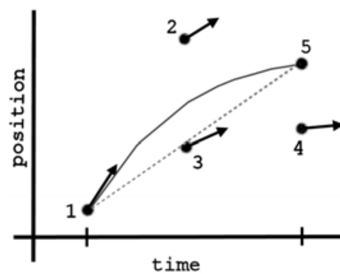
A8 - Visual depiction of Euler (left) and Midpoint (right) integration methods.

Stating this mathematically, the change in velocity over a given time frame can be calculated by integrating the acceleration. Likewise the change in position can be calculated by integrating the velocity. Clearly such integration cannot be done precisely on a computer, so numerical integration methods are used to approximate the functions of motion. Specifically, in physically based systems, modeling the motion of the objects usually leads to a system of ordinary differential equations (ODEs) or partial differential equations (PDEs). Calculating the solution of these systems often dominates the computation time of the simulation. Fortunately, there have been methods devised to solve ODE and PDE systems on the computer so libraries and source code to do so are readily available. In most cases the ODE solver of choice is an explicit Euler, Midpoint, or Runge Kutta 4 (RK4) based solver. A visual depiction of the Euler and Midpoint methods is shown in figure A8, a visual depiction of RK4 is found in figure A10, and the general methodology of RK4 is summarized in table A9. Another popular choice is the leapfrog or Verlet method. Yet, in particularly unstable situations, more sophisticated techniques such as implicit or adaptive methods may be necessary.

Problem statement:	Solve the differential equation given by: $s(t)$ = position of object at time t and $s'(t) = f(s, t)$ with initial position, $s(t_0) = s_0$, given.
RK4 Method:	$h = t_{i+1} - t_i$ $k_1 = f(s_i, t_i)$ $k_2 = f(s_i + 0.5k_1h, t_i + 0.5h)$ $k_3 = f(s_i + 0.5k_2h, t_i + 0.5h)$ $k_4 = f(s_i + k_3h, t_i + h)$ $s_{i+1} = s_i + \left(\frac{1}{6}\right) h(k_1 + 2k_2 + 2k_3 + k_4)$

A9 - Runge Kutta 4 update step.

A more detailed discussion of numerical methods, ODEs and PDEs can be found in section A.4 which also addresses finite methods.



**A10 - Visual depiction of RK4 integration method.
The numbers are indicating point subscripts.**

At this point it should be noted there is an inherent assumption that the motion of the object is continuous and differentiable. This assumption is broken by collisions which cause instantaneous changes in velocity. To handle this, special processing must be done when objects collide. A common method to deal with such situations is to apply impulse based methods [140]. But first the collisions must be detected.

A.3.3 Collision Detection

Another time consuming component of a physically based modeling system is the collision detection method used. Almost all collision detection techniques can be divided into two phases: the broad phase and the narrow phase. The broad phase determines which objects *might* be in collision. The narrow phase determines if two objects are indeed in collision and the details of that collision. These details include which two objects are involved, the point of contact, the normal of the separating plane (or equivalent), the type of contact, and the relative time of contact.

In the broad phase some routines partition space and report objects that are in the same space as potentially colliding [141]. Other routines give every object in the world some form of bounding box or sphere. These techniques include axis aligned bounding boxes (AABB) and object oriented bounding boxes (OOBB) [142]. With these boxes, various hierarchal tree structures, such as a BSP, are created which allow for quickly determining if two objects might be in a collision state. In other collision detection methods, nearest features of any two objects are maintained throughout the simulation and are used to detect when objects might collide [143, 144]. Most of these routines are efficient and effective in implementation. However when the actual collision is narrowed down and the collision point is determined they still must determine a separating plane (or equivalent) to perform the collision response. A more complete discussion of separating planes and how they relate to this dissertation may be found in the technical report by Dingle [78]. A general survey of collision detection methods can be found in the work of Lin and Gottschalk [145] or Jiménez et. al. [146].

When dealing with a large number of objects, collision detection is one of the slowest operations of a modeling engine. This is mostly caused by the number of comparisons between objects. In a brute force method this number is $O(n^2)$, where n is the number of objects. However, as stated above, it is the goal of the broad phase of collision detection to reduce this number. The techniques we will implement in the work of this dissertation are based on methods which maintain neighbor lists and space partitioning schemes to reduce the number of possible objects colliding with a given object. For example if it is known object A has only seven other objects near enough to be in collision, then only seven collision checks need to be performed. If there are over one hundred objects in space, this is a significant reduction. As these partitioning methods can prove useful in collision detection, and as will be shown later are also useful in display methods, some examples of existing methods are presented in the next section.

A.4 Space Partitioning

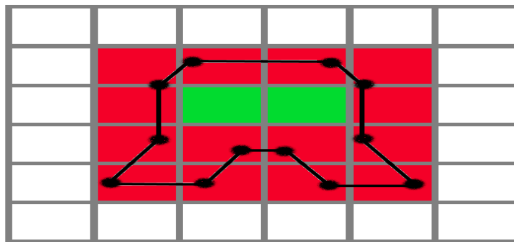
Space partitioning methods come in many forms. They are significant to this dissertation as they are used both to improve the performance of the collision detection routines as well as the display routines.

Fundamentally the goal of every such method is to organize objects in such a way as to quickly determine which objects are “near” a given object. In several cases these partitioning methods also present ways to approximate values, such as density or attractive forces. These latter such methods have proven useful in various particle simulations [38].

In most scenarios space partitioning methods are used in conjunction with object bounding methods to improve the performance of collision detection. The most common object bounding methods are hierarchies of spheres or spherical shells [147, 148], oriented bounding boxes (OBBs) and axis aligned bounding boxes (AABBs) [142, 149, 150]. The work of this dissertation only calculates collisions between the basic particle elements, which are spheres. Thus no object bounding methods are used. An understanding of such methods may be useful so the reader is encouraged to investigate the cited works as no detailed presentation of such methods will be found here. Instead we will focus on the space partitioning methods.

A.4.1 Simple Grid Cells

A basic division method of space is into grid cells [151]. In most cases the size of the cells are based on the size of the objects and how far the objects can move within one time step. To use this method boundaries are set in every direction. These boundaries allow a specific number to be associated with each grid cell. Thus an array of cells is created and every object is assigned a cell number by modular division of the object’s coordinates. If the cell size is not sufficiently large to completely contain every object then more complex computations are necessary to determine which cells contain a given object. Further, each cell is assigned a marker of empty, mixed, or full with regard to each object (as opposed to just containing or not). Empty indicates the cell contains none of a given object, mixed indicates part of the object resides in the cell, and full indicates the cell is completely contained within the object. Cells marked as mixed or full with regard to two or more objects suggest the contained objects may be in collision. An example of mixed cell containment is shown in figure A11.



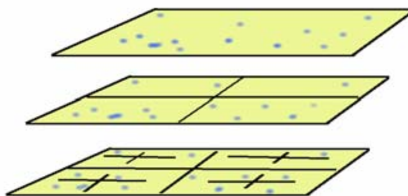
A11 - Illustration of mixed cell containment.
In the above, a space invader is outlined in black.
The green cells are full, red cells are mixed, and white cells are empty.

If not carefully implemented, for modeling environments where the objects are of extremely different sizes and shapes, this method is not very efficient. This is illustrated in a scenario where there are many small objects and one or two excessively large objects. The large objects may force the cell sizes to be large (as world space is larger). Thus cells may be so large that all the small objects can fit into one cell. From this it is possible no real reduction in the number of comparisons for collision is gained. Of course, this is an extreme example, but similar, more complex examples, could be devised. Stating this in another manner, if the average number of objects in a given cell varies greatly across time steps, this space partitioning will perform poorly. However, in the case of particles, where every particle is nearly the same size, this partitioning is effective and can be used to assist in efficient collision detection.

A.4.2 Hierarchal Tree Divisions

Octrees [152, 153] and binary space partitioning (BSP) trees [154, 155] have proven useful to divide space and locate objects for collision detection. Both methods offer hierarchical representations of space.

Octrees partition space recursively into cubical octants. Thus world space is seen as a large cube which is divided into eight smaller cubes, which in turn are divided into sixty-four cubes, and this continues until a predefined minimal sized cube is reached. Once the space is so divided each octant is marked as empty, mixed or full with respect to each object. Empty octants have none of a the specified object in them. Mixed octants contain part of the object, and full octants are entirely contained in the object. Collision detection is then performed by examining octants which are marked as full or mixed with respect to two or more objects. Notice this partitioning of space is faster than a simple grid-cell division as large empty areas may be detected at the higher levels of the tree. Thus a greater volume of space is examined in less time as each small cell need not be examined. This speed advantage is lost if the space is densely packed with objects. Notice, however, the placement of objects in the tree and the maintenance of the tree is more complex than using a simple grid. A 2D version of an octree is a quadtree. An example of a quadtree is depicted in figure A12.



A12 - Three levels of a quadtree division of 2D space.

A variation of octrees are kd-trees. Unlike octrees, kd-trees allow the splitting planes to be at variable positions (i.e. not all cubes). KD-trees are also designed in creation to balance the number of objects in any given node. Thus the number of objects in a given node stays nearly the same for any node, but the size of the nodes may vary greatly. This algorithm works well for visibility testing. Yet, in general, this is not the best arrangement for collision detection, as objects in the same node are not necessarily near each other. However, this division might be useful to evenly balance the workload across multiple processors.

Another variation of octrees are BSP trees. These trees work in a similar fashion to octrees but recursively cut the space into hyperplanes. These divisions are based on object locations, and usually use the (assumed) planar faces of the objects. They have an advantage over octrees and kd-trees as they are not required to partition space in an axis-aligned fashion. KD-trees are often considered axis-aligned BSP trees. BSP-trees are popular in gaming environments to sort rooms, however, the data structure can be complex.

A.4.3 Fast Multipole Methods

Fast Multipole Methods (FMMs) are very similar in basic design to octrees. They were introduced in the paper by Greengard and Rokhlin [156]. As far as the physical partitioning of space is concerned they are the same as an octree. However, within these methods more information on the contained objects is maintained. This extra information often represents long range forces of some kind. In maintaining this information at the various levels of the tree, larger and larger approximation for the effects of objects “far away” is maintained. This means the division is useful not only in collision detection and the forces produced, but also in approximating long range effects. Further, it is possible to estimate the error of these approximations as the error for each level may be calculated. Another difference in these methods from octrees is they usually include an influence list for each octant. Thus if an object is known to be in a specific node, the influence list for that node will indicate what other tree nodes to examine for effects that may influence the given object. To a degree this circumvents the necessity of looking for neighboring octants of the containing node. However it also increases the complexity of the data structure and the code to maintain it.

A.5 Differential Equations and Finite Methods
As presented in section A.2, in modeling, or simulating, physically based processes it is often necessary to solve large systems of equations, particularly those derived from differential equations. There are a variety of methods to perform this task. In this section an overview of some of the methods currently used to solve such systems will be presented. Particular attention will be given to systems of partial differential equations (PDEs) and how they may be solved using finite difference methods (FDMs) and finite element methods (FEMs). We will begin with a brief discussion of basic ideas and techniques. In this presentation is a demonstration of advancing simple ideas into much more complex mathematics. In a direct sense the information provided here is for background purposes, however, the presentation method is also demonstrative of the approach taken to achieve the results of this dissertation. The fundamental point is ***conceptually the methodologies should be easy to understand, and it is in the details of applying the methods where complexity, creativity and skill are required***. In such application deeper understanding of the methodologies will be achieved.

In most high schools in the United States we are "intuitively" taught how to apply numerical integration methods to real world problems. Specifically we are given a framework which uses derivatives and Euler integration. From this simple beginning we may advance to the more complex topics of Taylor expansions, finite differences, interpolation, solving systems of ordinary differential equations (ODEs), solving systems of partial differential equations (PDEs), finite difference methods (FDMs), and finite element methods (FEMs). As this is how we have been taught, so shall it be presented here. ***This presentation will allow the more advanced reader the option to skip to the later sections of this section while still allowing the more novice to obtain some of the necessary background to follow the eventual presentation of the finite methods.***

In the beginning, probably in a physics class, we start with a simple description of motion where:

$$\text{acceleration} = a,$$
$$\text{velocity} = v(t) = at + v_0$$

and

$$\text{position} = s(t) = (½)\, at^2 + vt + s_0.$$

We might also be presented with the notion of force is the product of mass times acceleration:

$$F = ma$$

Which in turn tells us:

$$a = F/m.$$

From these simple equations we learn how to approximate the position of an object by applying the equations:

$$s_{new} = s_{old} + v_{old} * \Delta t$$

and

$$v_{new} = v_{old} + at.$$

Notice this, effectively, this is applying an Euler integration.

As we advance through the educational system, eventually we come to a presentation of Taylor Expansions. In basic form this resolves to the equation:

$$f(x+h) = f(x) + f'(x)h + f''(x)\frac{h^2}{2!} + \cdots$$

From this expansion we are taught a method to approximate derivatives. Applying this method we discover the forward, backward and central difference approximations of the first derivative.

$$f'(x) \cong \frac{f(x+h)+f(x)}{h} \quad \text{(forward difference, first derivative)}$$

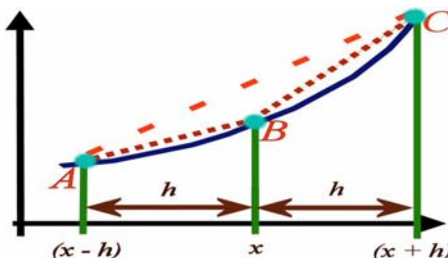
$$f'(x) \cong \frac{f(x)-f(x-h)}{h} \quad \text{(backward difference, first derivative)}$$

$$f'(x) \cong \frac{1}{2h}\big(f(x+h)-f(x-h)\big) \quad \text{(central difference, first derivative)}$$

Further application of the method shows similar approximations for the second and higher derivatives. For example the central difference approximation of the second derivative is:

$$f''(x) \cong \frac{1}{h^2}\big(f(x+h)-2f(x)+f(x-h)\big)$$

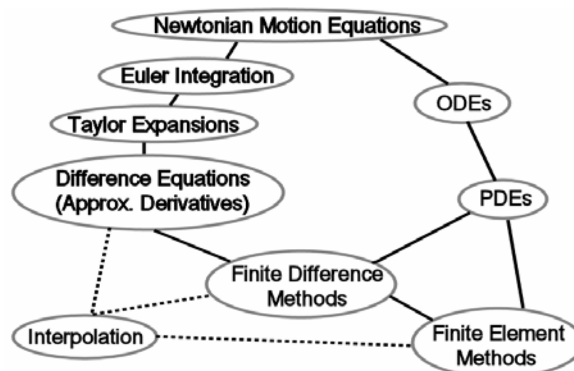
Pictorially we may think of these differences as the slope of various lines drawn through the points (*x*, *f*(*x*)), (*x*-*h*, *f*(*x*-*h*)) and (*x*-*h*, *f*(*x*+*h*)). This is illustrated in figure 000, where the slope of BC would be the forward difference, the slope of BA would be the backward difference and the slope of AC would be the central difference.



A13 – Slope of lines and differencing methods.

Notice, from the graph in figure A13, it is easily seen the difference methods have conceptual ties to linear interpolation. Such ties to interpolation methods also will be noticeable in later discussions of finite methods.

Having shown that we may arrive at differencing methods to approximate derivatives by starting from simple Newtonian equations of motion, we will return to those equations. While we can solve such problems knowing nothing of calculus, it is in such a course where we discover the equations of motion represent a basic ordinary differential equation (ODE) problem.



A14 – Relation of mathematical methods.

If we are fortunate enough to continue in mathematics, we learn of similar problems involving partial differential equations, or rather PDE problems. So from a problem we learned to solve in high school we can arrive at very advanced mathematical topics. In being able to apply the topics back to solve similar problems we gain an understanding and appreciation of the mathematics. Again, the reason for this basic presentation is to create a frame of reference for the following presentations on PDEs and methods for solving problems related to them. So from this point we will move to a more detailed discussion of some definitions and descriptions of several PDE based problems. This in turn will lead to a general presentation of current methods used to solve such problems. Focus will be given to finite difference methods (FDMs) and finite element methods (FEMs). A visual representation of how these various methodologies relate is presented in figure A14.

A.5.1 Partial Differential Equations

A PDE is defined to be an equation involving partial derivatives of an unknown function of two or more independent variables. We also note, in terms of modeling, it is the state of the object being modeled that is likely represented by the PDE.

The order of a PDE is the highest order partial derivative appearing in the equation. A PDE is linear if it is linear in the unknown function and all its derivatives, with coefficients depending only on the independent variables. Examples of these include:

A second order PDE: $$\frac{\partial^2 u}{\partial x^2} + x\frac{\partial^2 u}{\partial y^2} = 5$$

A third order PDE: $$\frac{\partial^3 u}{\partial x^2 \partial y} + 4y\frac{\partial^3 u}{\partial y^2} + 3u = 2x$$

A linear PDE: $$7xy\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} = 12$$

A non-linear PDE: $$xu\frac{\partial^2 u}{\partial x^2} + 3\frac{\partial^2 u}{\partial y} = 2x\,.$$

Many engineering problems appear as linear second order PDEs. In two variables these problems take the general form of:

$$A\frac{\partial^2 u}{\partial x^2} + B\frac{\partial^2 u}{\partial x \partial y} + C\frac{\partial^2 u}{\partial y^2} + D = 0$$

These linear second order PDE's may be classified by the value of $d = B^2 - 4AC$. If d is less than zero the PDE is said to be elliptic. If d is equal to zero it is parabolic and if d is greater than zero the PDE is hyperbolic.

Classical examples in engineering of elliptic linear second order PDEs are problems involving steady state systems, or systems that are not dependent on time. An example of this might be a thin heated plate, insulated on its faces, which has been determined to behave according to Laplace's Equation:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0,$$

where T is the temperature of the plate, and x and y are spatial measurements on the plate. Note that T in this case is the unknown function of the PDE.

Parabolic linear second order PDE problems are often used to determine how an unknown varies in both space and time. For example in the heat conduction equation there are both spatial and temporal derivatives:

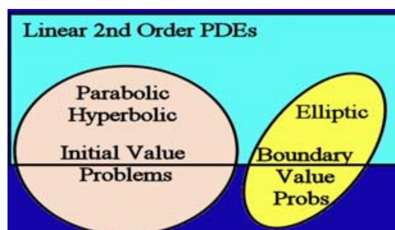
$$k\frac{\partial^2 T}{\partial x^2} = \frac{\partial T}{\partial t}.$$

These types of problems are referred to as *propagation problems*. A 1D-spatial example, employing the heat conduction equation as stated, might be a heated insulated rod.

Hyperbolic linear second order PDE problems are also propagation problems. Unlike the parabolic cases, hyperbolic PDEs contain a term which involves the second derivative with respect to time of the unknown function. This usually leads to oscillation in the solution. The classic example of such a PDE is the wave equation:

$$\frac{\partial^2 y}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 y}{\partial t^2}.$$

The three types of linear second order PDEs may further by categorized as Initial Value Problems or Boundary Value problems. Some prefer this classification when discussing what criteria is needed to apply a solution method. Specifically, parabolic and hyperbolic PDEs are considered to be Initial Value Problems and elliptic PDEs are considered Boundary Value Problems. Thus, it can be remembered, to solve parabolic and hyperbolic PDE problems requires at least some form of initial value. Likewise, to solve elliptic PDE problems requires a boundary condition to be known. This is illustrated in figure A15.



A15 - Classification of PDEs.

A.5.2 Finite Difference Methods

A popular technique to solve PDE based problems is that of Finite Difference Methods (FDMs). In these methods the solution domain is divided into a grid of discrete points. The PDE is then written for each node. The derivatives in the PDE are then replaced by a difference equation. These methods result in a point-wise approximation of the solution. In general terms, FDMs follow some basic steps:

1. Place an evenly spaced, regular, grid on the object being modeled.
   - In 2D this means a grid of squares.
   - The solution to be derived will be at the nodes of the grid (i.e. at the intersection points of the grid lines).
2. Apply/Derive a solution expression for the entire model.
   - This must be a PDE, otherwise another method is required.
3. Replace the derivatives of the PDE using difference equations.
4. Apply this new equation to each node.
   - Thus a system of equations is created.
5. Rewrite the system of equations in matrix form and solve.
   - The use of difference equations should force the matrix to be banded and diagonally dominant, which guarantees a solution exists and it can be obtained quickly in terms of computational expense.

The implementation of this method will vary in exact details depending on the type and details of the problem. Examples of using this method may be found in Appendix B.

FDMs have proven to be useful in many cases and are often the quickest and easiest solution method. However, as shown in the examples (Appendix B), the explicit methods suffer severe stability problems, requiring small step sizes. The implicit methods correct for the stability problems, but demand the solution of potentially large systems of equations. Further, it is noted that in most cases involving higher dimensions these systems will not be able to be to form systems which can be easily solved (i.e. no longer tridiagonal). Further these systems are often extremely large, requiring large amounts of storage space and time to solve.

Application-wise FDMs also fail to address many real world scenarios. The most common difficulty is found in applying them to objects with irregular shaped boundaries. In these cases to achieve accurate results requires a very fine grid, which increases the system size. Or very special treatment of nodes near the boundaries is necessary.

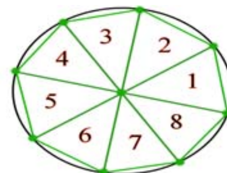
FDMs are also difficult to apply to scenarios which do not have a “global solution” for the entire object. Specifically, if an object is heterogeneous or has unusual boundary conditions applying an FDM based technique may prove excessively challenging. Fortunately other methods, referred to as finite element methods (FEMs), have been developed. These methods are designed to more easily account for irregular boundaries and unusual circumstances.

A.5.3 Finite Element Methods

In contrast to the FDMs, the finite element methods (FEMs) divide the solution domain into simple elements, such as triangles or tetrahedrons. For each element a solution to the PDE is then derived, in terms of equations. The solution for the entire domain is then created by *assembling* the element-wise solutions. This assembling must be done in such a way to ensure continuity between the element boundaries. In this process a piecewise solution to the underlying PDE is developed.

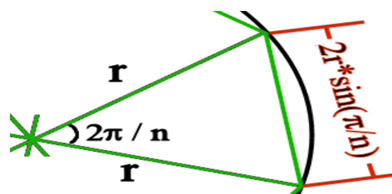
To illustrate the conceptual idea behind FEMs we will begin with an oversimplified example. So, consider the problem of approximating the circumference of a circle and pretend for the moment we do not know

the explicit equation, but miraculously know about sin() and π. We first divide our solution domain, the circle, into simple elements as pictured in figure A16.



A16 - FEM subdivision of a circle.

We then develop equations useful to solving the problem by examining a single element. In this process we discover an equation for the outside line of the triangle, as shown in figure A17.



A17 - FEM element equation for a circle.

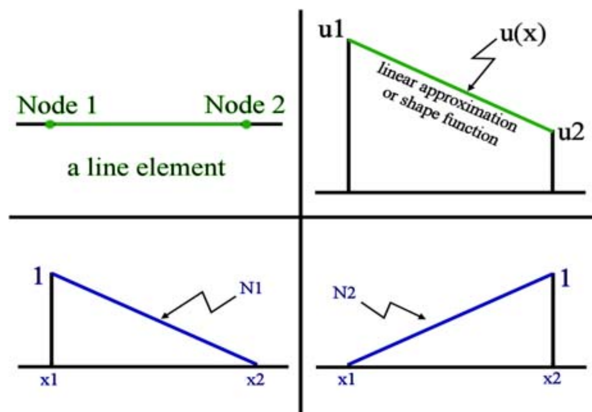
Upon examining this equation we see that if we align all our triangles correctly the sum of these outer edge lengths will approximate the circumference of the circle. Thus the final step for this example is calculating those lengths and summing them.

While trivial, this example illustrates some of the major concepts involved in Finite Element Methods. One such concept is the derivation of the solution by placing equations on each element in a way that is consistent with a global solution. Another important concept to be realized from this example is the "removability" of each element from the global object. Thus the solution equations for each element may be derived in a coordinate system local to the element. While this solution may be identical for each element, there is nothing forcing each to have the same solution. Though the solutions must be capable of maintaining continuity between elements. In both of these concepts is the underlying point that, while deriving the solution for an individual element, care must be taken that the solution may be applied back to the global object in such a way as to maintain a continuity between each element's solution. It is in this last point that the implementation of FEMs often becomes difficult.

From this trivial example we might also determine the general steps of using a FEM. We will first list the steps and then present a brief discussion of each. The steps are:

1. Discretization of the solution space into elements.
2. Develop a solution equation, or set of, for each element.
3. Assemble the elements and their solution equations.
4. Check boundary conditions (*this point was not illustrated in the circle example*).
5. Calculate the solution for the entire object.

In the discretization step the solution domain, or object, is broken into a finite number of elements. These elements are usually triangles or tetrahedrons, but other shapes may be used. The intersection points of the lines which make up the sides of the elements are called *nodes* and the sides are called *nodal lines* or *nodal planes*.



A18 - General graphs of approximating and interpolating functions for FEMs.

In the development of solution equations for each element, the equations often are taken to be polynomials and the choice is most frequently a linear polynomial. While any type of solution may be sought, and in some cases required, linear polynomials are the easiest to implement in most circumstances. In this process the solution usually begins in the form of $u(x) = a_0 + a_1x$, where a_0 and a_1 are treated as constants and x can for this discussion be thought of as a 1D scalar. To picture this process, the graphs shown in figure A18, may be useful.

We note we will know the spatial coordinates of our nodal points, or rather the "endpoints" of the element we are currently examining. Thus if we denote the endpoints as x_1 and x_2, then we will be seeking the values $u(x_1)$ and $u(x_2)$, or simple u_1 and u_2 respectively. However for the purpose of deriving our equations we assume we know what u_1 and u_2 are, or specifically we want the equations in term of what we know (the x's) and what we are looking for (the u's). Thus we create a system of two equations and solve for a_0 and a_1:

$$u_1 = a_0 + a_1x_1$$

$$u_2 = a_0 + a_1x_2$$

$$a_0 = \frac{u_1x_2 - u_2x_1}{x_2 - x_1}$$

$$a_1 = \frac{u_2 - u_1}{x_2 - x_1}$$

This allows us to express the function $u(x)$ in terms of x values and the unknown u's. To generalize this we rewrite our starting function as:

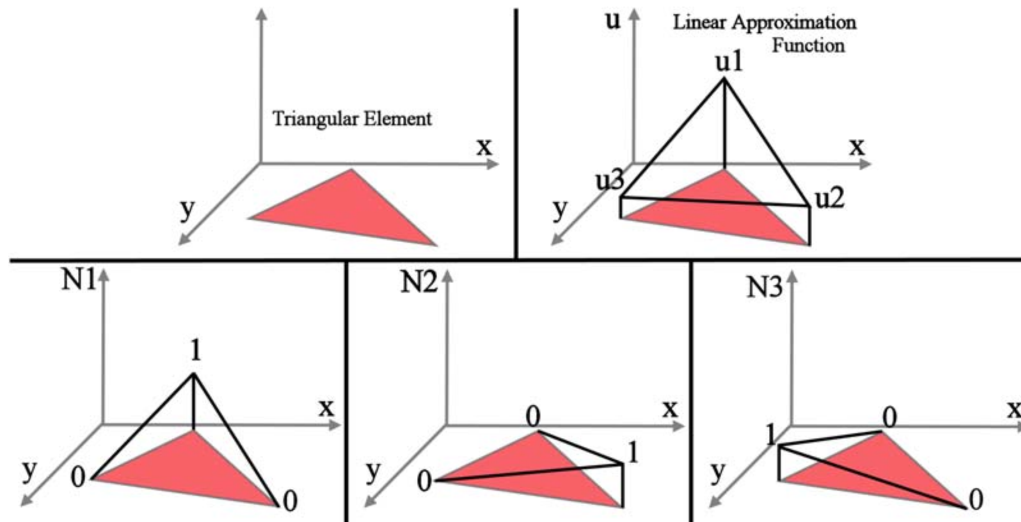
$$u(x) = N_1u_1 + N_2u_2$$

Where for the case just described,

$$N_1 = \frac{x_2 - x}{x_2 - x_1} \text{ and } N_2 = \frac{x - x_1}{x_2 - x_1}.$$

We then refer to $u(x)$ as the approximating or shape function and the N's as the interpolating functions. This allows for general terminology to be used to refer to a variety of possible solutions. However, this is just the beginning of deriving the solution. Once we have the basic shape function determined we must then develop equations that describe the behavior of the element itself. These equations are based on the relationships between the unknown u's in the approximating function and the underlying PDE of the given problem. Figure A19 may assist in visualizing this.

In simpler terms the shape function describes how the values within the element will vary from one node to the next. The equations describing the values at the nodes themselves must be determined based on the underlying PDE of the entire problem. However, a balance must be maintained within the element as each node value affects the others. Further it must be possible to assemble the elements in a manner that maintains continuity of the values at the features shared between nodes during the assembly step.



A19 – Visual presentation of an approximating function for a triangular FEM element. Notice the approximation function of formed from the interpolating functions.

The exact process of how the relation to the PDE is accomplished varies from problem to problem and often depends on what is viewed as important and what assumptions about the problem are made. The desired result, however, is always the same, which is a system of equations.

In most, but not all, cases the resulting system is a set of linear algebraic equations. This system deals only with a given element and can be expressed in matrix form as:

$$[k]\boldsymbol{u} = \boldsymbol{F},$$

where $[k]$ is the *element property* or *stiffness matrix*, $\boldsymbol{u}$ is a column vector of unknowns at the element's nodes and $\boldsymbol{F}$ is a column vector denoting any external forces applied at the nodes.

Once the equations for each element have been derived the assembly step begins. The focus of this step is to maintain continuity between the shared features of each element to obtain a continuous solution across the entire object. Thus great care must be made to ensure the solution values at nodes shared between elements are equivalent. The assembly of all the equations for each element will result in a system of equations. Following the notation for the system derived for each element we then have:

$$[K]\boldsymbol{u'} = \boldsymbol{F'},$$

where $[K]$ is the *assemblage property matrix*, $\boldsymbol{u'}$ is a column vector of unknowns, and $\boldsymbol{F'}$ denotes the forces applied to the elements.

The last “setup” step is only necessary if there are boundary conditions to consider. Such conditions may influence the equations of elements on the object’s boundaries and may introduce new forces acting on the object. To reflect these changes the matrix system is denoted:

$$[\boldsymbol{K}]\boldsymbol{u}' = \boldsymbol{\Phi}'.$$

The final step, once everything is setup, is to solve the resulting system. This is most often accomplished with a matrix solving routine. In some scenarios it is possible during the assembly step to arrange the elements (simplistically via numbering) to force the matrix into a banded form which allows for faster, more specialized routines to be used.

Having outlined the general steps of FEMs, it is noted a true understanding can only be gained by using them. An example of applying these general FEM steps to a spring system may be found in Appendix C.

A.5.4 Comparison of FDMs and FEMs

So we have now presented a detailed discussion of two methods to solve PDE problems, namely finite difference methods (FDMs) and finite element methods (FEMs). Both of these methods have been used to solve PDE problems for some time in various fields. Both have strengths and weaknesses.

FDMs are conceptually easy and relatively simple to program. They approximate a solution using a uniform grid. Unfortunately they are difficult to apply to complicated geometries and do not work well under heterogeneous or unusual conditions. Another problem is an issue of grid sizes required. The more accurate the solution desired the finer the grid required and thus the more memory and computer time required. FDMs may also suffer from numerical instability if the matrices to solve become too stiff.

FEMs are based on easily understood ideas and are designed to handle unusual geometries. To an extent they may also be applied to heterogeneous conditions if extreme care is taken in the assembly step. The real difficulty in FEMs is two-fold. The first difficulty is in properly setting up the solution. The second difficulty is in implementing the solution. Properly writing the computer code to solve such problems can be challenging. So in both the setup and implementation the complexity of the problem can magnify the complexity of deriving an accurate solution. FEMs, like FDMs, suffer from a tendency to require large amounts of memory and computer time to solve the complex systems of equations that may be developed in the quest for accuracy. Also in similar fashion to FDMs, care must be taken when dealing with solving the equation systems as instabilities may occur, particularly when applying generic or non-robust techniques.

As a final note, both FDMs and FEMs are magnificent tools in solving PDE based problems. Both serve a variety of communities in solving difficult problems. However, it takes skill in the problem domain to apply the methods correctly as well as skill in programming a computer to obtain such solutions. Of course, a not so small understanding of mathematics also proves useful.

APPENDIX B

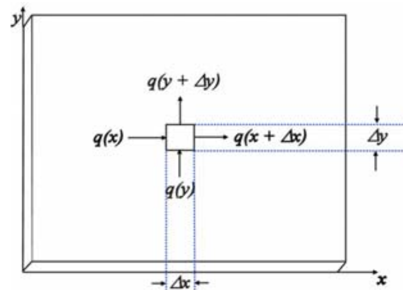
FINITE DIFFERENCING METHOD EXAMPLES

The implementation of this finite differencing methods will vary in exact details depending on the type and details of the problem. To illustrate this a solution example for an elliptic linear second order PDE will be given, followed by an explicit solution of a parabolic PDE, which in turn will be followed by an implicit solution of the same parabolic PDE.

FDM Example 1 – Elliptic Linear Second Order PDE:
An example involving an elliptic linear second order PDE can be illustrated by the following problem: Consider a thin rectangular plate of thickness Δz insulated everywhere but its edges. Assume at the edges the temperature may be set to a constant value. Notice the insulation and thinness of the plate will allow us to model the heat transfer in just the *x* and *y* directions. Further note that we are only interested in the final steady state of the plate. So under these conditions it can be shown that the Laplace equation,

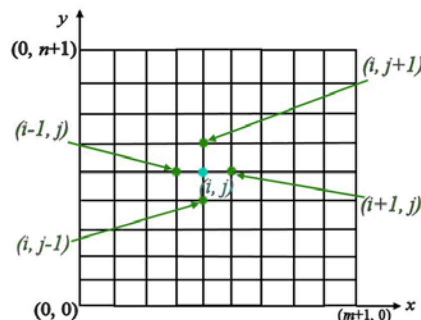
$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0,$$

may be used to model the temperature, *T*, of the plate. Effectively this means the rate of change of temperature throughout the plate is constant, or simply the temperature at any point on the plate remains constant, in a steady state. Note this steady state is dependent on the steady state of the boundary conditions. The picture shown in figure B1 may prove useful in visualizing this.



B1 – Thin rectangular plate. q(x) is a heat function.

The first step, to solve this problem, is to apply a grid over the solution space. Thus we arrive at a picture similar to the one in figure B2:



B2 – A grid is placed over the plate, with elements indexed as show.

Looking at the Laplace equation we see we must approximate the second derivative of our unknown function T with respect to x as well as its second derivative with respect to y. For this approximation we will use central divided difference approximations. Specifically,

$$\frac{\partial^2 T}{\partial x^2} = \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2}$$

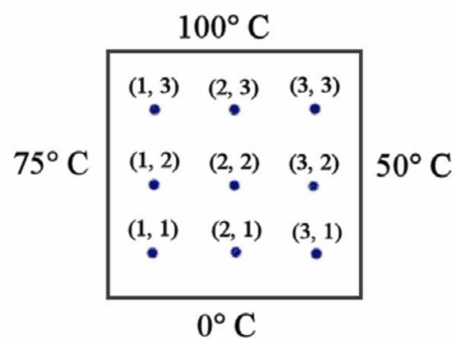
and

$$\frac{\partial^2 T}{\partial y^2} = \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2}$$

Substituting these into the Laplace equation gives us:

$$\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = \frac{T_{i+1,j} - 2T_{i,j} + T_{i-1,j}}{\Delta x^2} + \frac{T_{i,j+1} - 2T_{i,j} + T_{i,j-1}}{\Delta y^2} = 0$$

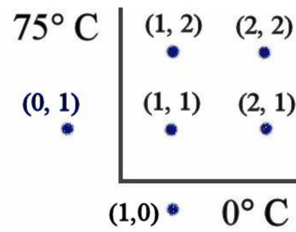
So this equation will be applied to each node of our regular grid. However, we now recall that to solve elliptic PDE problems, we need an explicit boundary condition. While the problem statement merely said the boundary temperature could be set, and thus we could use constant variables, for clarity we will arbitrarily select numbers. Specifically we assume the top of the plate is set to 100°C, the bottom to 0°C, the left edge to 75°C and the right edge to 50°C. We also use a 3x3 interior grid, or a 5x5 grid if we include points on the edges. This is visualized in figure B3.



B3 – Problem setup for a heat plate.

Applying these numbers to node (1, 1) we begin with our approximated PDE:

$$T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1} - 4T_{i,j} = 0$$



B4 – Neighbor nodes of node (1, 1).

Which, looking at figure B4, for node (1, 1) is:

$$T_{2,1} + T_{0,1} + T_{1,2} + T_{1,0} - 4T_{1,1} = 0$$

We then incorporate the boundary conditions, where $T_{0,\,1} = 75$ and $T_{1,\,0} = 0$:

$$-4T_{1,1} + T_{1,2} + T_{2,1} = -75$$

In a similar fashion eight more equations for the other eight interior nodes may be obtained:

$$\begin{array}{cccccccccl}
-4T_{1,1} & +T_{2,1} & & +T_{1,2} & & & & & & = -75 \\
T_{1,1} & -4T_{2,1} & +T_{1,3} & & +T_{2,2} & & & & & = \ \ 0 \\
 & T_{2,1} & -4T_{1,3} & & & +T_{3,2} & & & & = -50 \\
T_{1,1} & & & -4T_{1,2} & T_{2,2} & & +T_{1,3} & & & = -75 \\
 & T_{2,1} & & +T_{1,2} & -4T_{2,2} & +T_{3,2} & & +T_{2,3} & & = \ \ 0 \\
 & & T_{3,1} & & +T_{2,2} & -4T_{3,2} & & & +T_{3,3} & = -50 \\
 & & & T_{1,2} & & & -4T_{1,3} & +T_{2,3} & & = -175 \\
 & & & & T_{2,2} & & +T_{1,3} & -4T_{2,3} & +T_{3,3} & = -100 \\
 & & & & & T_{3,2} & & +T_{2,3} & -4T_{3,3} & = -150
\end{array}$$

This system of equations is then written in matrix form and solved for each $T_{i,j}$. It is important to observe the created matrix is banded and diagonally dominant. In this it is recalled that *diagonally dominant* means for any given row, i, the absolute value of the element, $e_{i,i}$, is greater than or equal to the sum of the absolute values of the other elements in row i. It should be understood these features are a direct result of choosing the difference equations to approximate the derivatives.

Traditionally this system of equations is solved using a Gauss-Seidel based method, which is referred to by some as Liebmann's Method. Other methods to solve sparse or banded matrix systems may also be used. However, as such numerical methods are inherently used throughout this dissertation a presentation of this simple method seems in order.

In Liebmann's Method, the equation $T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1} - 4T_{i,j} = 0$ is restated, solving for $T_{i,\,j}$, to arrive at: $T_{i,j} = \dfrac{T_{i+1,j} + T_{i-1,j} + T_{i,j+1} + T_{i,j-1}}{4}$. This will allow a solution to be derived iteratively, where each $T_{i,\,j}$ is initially assigned a reasonable arbitrary value and for each iteration we define,

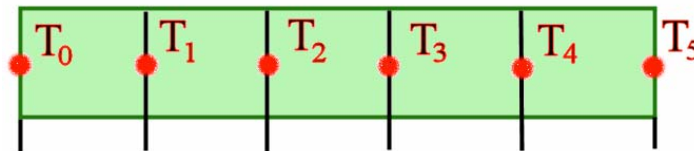
$T_{i,j}^{new} = \frac{T_{i+1,j}^{old} + T_{i-1,j}^{old} + T_{i,j+1}^{old} + T_{i,j-1}^{old}}{4}$. This leads to an absolute relative error of $\left|\varepsilon_{i,j}\right| = \left|\frac{T_{i,j}^{new} - T_{i,j}^{old}}{T_{i,j}^{new}}\right|$. Thus we would iterate until $\left|\varepsilon_{i,j}\right| \le \varepsilon_F$, where ε_F is a pre-decided tolerance value.

FDM Example 2 – Parabolic Linear Second Order PDE:
As we have now finished a solution to an elliptic linear second order PDE we will present two ways to solve a parabolic second order PDE. The motivation for showing such explicit examples is to illustrate the variability of the details of the finite difference methods as well as to show some of the strengths and weaknesses of such methods.

Consider a rod insulated everywhere but its ends. Assume we wish to model the heat propagation through the rod and wish to apply Fourier's law of heat conduction, or simply the heat conduction equation, $k\frac{\partial^2 T}{\partial x^2} = \frac{\partial T}{\partial t}$, where k is a constant reflecting the material properties, T is the temperature of the rod at position x along its length at time t. In this description we are inherently assuming the radius of the rod is sufficiently small compared to the length of the rod. This with the insulation, allows us to only be concerned with the heat propagation in the x direction.

We will solve this problem first using an explicit method and then using an implicit method. This will illustrate some of the strengths and weaknesses of the Finite Difference Methods, as well as the diversity of the actual implementation of the methods.

The first step of solving this problem using a Finite Difference Method is to apply a grid to the solution space. As we only have one spatial dimension we cut the rod into equal segments. The nodes of our grid are at the x values where we perform the cuts, as shown in figure B5.



B5 – Nodes of a segmented rod.

The next step is to approximate the derivatives involved in the problem's associated PDE. In this case we will use the central difference approximation for the second derivative of temperature, T, with respect to x and the forward difference approximation for the first derivative of T with respect to time, t. These choices are made to allow an explicit method to be applied. Specifically we will say:

$$\frac{\partial^2 T}{\partial x^2} = \frac{T_{i+1}^{l} - 2T_i^{l} + T_{i-1}^{l}}{\Delta x^2}$$

and

$$\frac{\partial T}{\partial t} = \frac{T_i^{l+1} - T_i^{l}}{\Delta t}$$

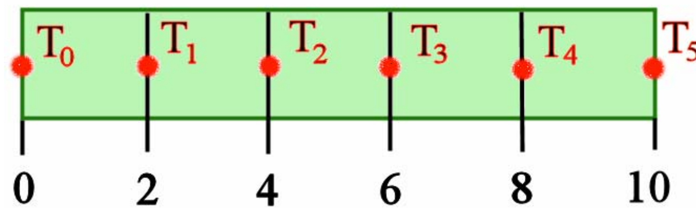
Notationally the superscript l denotes the time step, or location in the time dimension, whereas the subscript, i, denotes the spatial step in the x dimension.

Thus the heat conduction equation, $k\frac{\partial^2 T}{\partial x^2} = \frac{\partial T}{\partial t}$, becomes $k\frac{T_{i+1}^l - 2T_i^l + T_{i-1}^l}{\Delta x^2} = \frac{T_i^{l+1} - T_i^l}{\Delta t}$, where solving for T_i^{l+1} yields:

$$T_i^{l+1} = T_i^l + \frac{k\Delta t}{\Delta x^2}\left(T_{i+1}^l - 2T_i^l + T_{i-1}^l\right)$$

Thus if we know the initial temperature values of the rod at our nodal points we may *explicitly* determine the temperature of the rod at the nodes at any future time by applying the above equation.

To better illustrate this we introduce some arbitrary numbers. Assume the length of the rod is 10 cm, Δx is 2 cm, Δt is 0.1 seconds, k is 0.835 and the temperature of the rod everywhere at $t = 0$ is 0°C. We will further assume the left end of the rod is then held at a constant temperature of 100°C and the right end of the rod is held at 50°C. So $T(0,0) = 100$ and $T(10,0) = 50$.



B6 – Length measures of 0 to 10, showing 2 cm between each node.

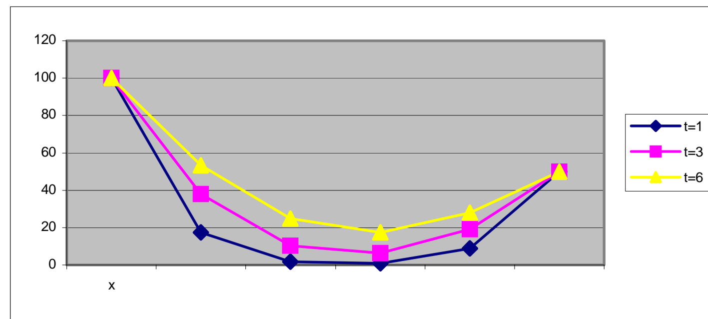
Thus, as shown in figure B6,we begin with the initial conditions:

$$T_0^0 = 100,\ T_1^0 = 0,\ T_2^0 = 0,\ T_3^0 = 0,\ T_4^0 = 0,\ T_5^0 = 50$$

So if we apply the formula $T_i^{l+1} = T_i^l + \frac{k\Delta t}{\Delta x^2}\left(T_{i+1}^l - 2T_i^l + T_{i-1}^l\right)$, for time = 0.1 second (i.e. l is 1) for the node at $x = 2$ (i.e. i is 1), we would arrive at the following:

$$T_1^1 = T_1^0 + \frac{0.835 * 0.1}{2^2}\left(T_2^0 - 2T_1^0 + T_0^0\right)$$
$$T_1^1 = 0 + 0.02875\left(0 - 2*0 + 100\right)$$
$$T_1^1 = 2.0875$$

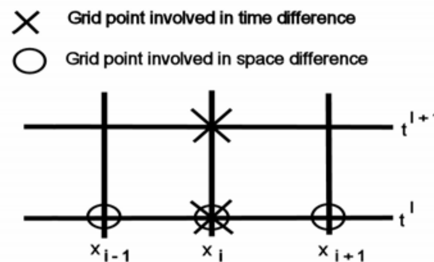
In a similar manner, the temperature of the other nodes may be determined for any time, t. By using a simple spreadsheet implementation we can derive a graph of the results for various times. This graph is illustrated in figure B7 for $t = 1$, 3 and 6 seconds, the x-axis is the value of x along the rod's length and the y-axis is the temperature of the rod in degrees Celsius.



B7 – Temperature of the nodes at times, 1, 3, and 6 seconds.

While perhaps not immediately obvious the temperature at each node as time increases will converge to a specific steady temperature as long as the end point temperatures remain fixed. However in this process we are able to answer any time dependent question about the rod. For example we know where the coolest points in the rod are at time, $t = 0.5$ seconds. This is subtly different than had we solved the problem as a steady state problem.

As a side note, the explicit method just described is often associated with a computational molecule as described in the figure B8. Computational molecules are useful in comparing various solution methods, and serve as pictorial reminders of the process involved in each method.

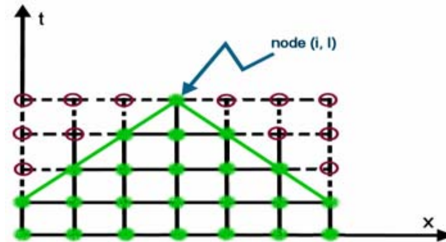


B8 – Computational model of explicit method.

While explicit solution techniques are probably the easiest to implement there exist some problems when implementing them. The first issue, and most obvious, is not all PDEs will have an explicit solution, or rather, some cases will prevent the algebraic solution for the unknown. The second, and perhaps more important, issue is one of convergence and stability. Specifically, explicit solution techniques do not always converge to the correct solution (they may oscillate or diverge from the true solution) and they may not always be stable (numerically they simply "blow up" as the errors become larger and larger). In the above example it has been shown that if we let $\lambda = \frac{k\Delta t}{(\Delta x)^2}$, then the explicit solution, as described, is stable but may oscillate for $\lambda \leq ½$. To prevent the oscillation, in this problem, λ must be less then or equal to ¼ [157].

A further difficulty in using explicit methods results from some influencing nodes not being taken into account. This is mostly an issue related to how time is being treated. This is best explained in an illustrated manner. Consider figure B9, where the solid nodes are those nodes which affect node (i, l) according to the explicit method. However in reality the dashed nodes also influence node (i, l). How significant this

oversight is depends on the nature of the problem. In some situations it may prove insignificant and in cases where extreme accuracy is not important it may be overlooked. But it is an issue to consider when selecting a solution method.



B9 – Node (*i*, *l*) should be influenced by nodes on dashed lines.

To overcome some of the convergence and stability issues of explicit methods, implicit methods have been developed. In general this means no explicit formula for T_i^{l+1} will be used and in many cases the spatial derivative will be approximated at an advanced time step. In the case of the above problem instead of using

$$\frac{\partial^2 T}{\partial x^2} = \frac{T_{i+1}^l - 2T_i^l + T_{i-1}^l}{\Delta x^2},$$

we use

$$\frac{\partial^2 T}{\partial x^2} = \frac{T_{i+1}^{l+1} - 2T_i^{l+1} + T_{i-1}^{l+1}}{\Delta x^2}.$$

This, of course, makes it impossible to algebraically obtain an explicit solution. However, by using the initial and boundary conditions of the rod it is possible to develop a system of equations and solve the system simultaneously for a given time step.

For the interior nodes the heat conduction equation,

$$k\frac{\partial^2 T}{\partial x^2} = \frac{\partial T}{\partial t},$$

becomes

$$k\frac{T_{i+1}^{l+1} - 2T_i^{l+1} + T_{i-1}^{l+1}}{\Delta x^2} = \frac{T_i^{l+1} - T_i^l}{\Delta t}.$$

Solving for T_i^l (as opposed to T_i^{l+1}) yields:

$$T_i^l = -\left(\frac{k\Delta t}{(\Delta x)^2}\right)T_{i-1}^{l+1} + \left(1 + \frac{k\Delta t}{(\Delta x)^2}\right)T_i^{l+1} - \left(\frac{k\Delta t}{(\Delta x)^2}\right)T_{i+1}^{l+1}$$

However, for the nodes at the ends of the rod, the external temperatures must be taken into consideration. In this we are applying the given boundary conditions. For the left end of the rod, where spatial index i = 0, we will say $T_0^{l+1} = f_0(t^{l+1})$, where f_0 is a function describing how the boundary temperature changes with time. So applying this to the above equation for i = 1 we see:

$$T_1^l + \left(\frac{k\Delta t}{(\Delta x)^2}\right) f_0(t^{l+1}) = \left(1 + \frac{k\Delta t}{(\Delta x)^2}\right) T_1^{l+1} - \left(\frac{k\Delta t}{(\Delta x)^2}\right) T_2^{l+1}$$

Similarly, if we let m denote the last spatial index and f_{m+1} be the function describing the temperature at the right end of the rod we see the temperature for the last interior node, T_m satisfies the equation:

$$T_m^l + \left(\frac{k\Delta t}{(\Delta x)^2}\right) f_{m+1}(t^{l+1}) = -\left(\frac{k\Delta t}{(\Delta x)^2}\right) T_{m-1}^{l+1} + \left(1 + \frac{k\Delta t}{(\Delta x)^2}\right) T_m^{l+1}$$

In the given example $f_0 = 100$ for all t and $f_{m+1} = 50$ for all t. Note also with Δx as 2 cm, Δt as 0.1 seconds, k as 0.835 that

$$\frac{k\Delta t}{\Delta x^2} = 0.020875 \cdot$$

Thus for the first interior node ($i = 1$) at time $t = 0$ we have:

$$0 + 0.020875*(100) = 1.04175*T_1^1 - 0.020875*T_2^1$$

or

$$1.04175*T_1^1 - 0.020875*T_2^1 = 2.0875$$

Applying the above equations to the other three interior nodes gives the following system:

$$\begin{bmatrix} 1.04175 & -0.020875 & & \\ -0.020875 & 1.04175 & -0.020875 & \\ & -0.020875 & 1.04175 & -0.020875 \\ & & -0.020875 & 1.04175 \end{bmatrix} \begin{bmatrix} T_1^1 \\ T_2^1 \\ T_3^1 \\ T_4^1 \end{bmatrix} = \begin{bmatrix} 2.08075 \\ 0 \\ 0 \\ 1.04375 \end{bmatrix}$$

This system can be solved for $t = 0.1$ to discover:

$$\begin{bmatrix} T_1^1 \\ T_2^1 \\ T_3^1 \\ T_4^1 \end{bmatrix} = \begin{bmatrix} 2.0047 \\ 0.0406 \\ 0.0209 \\ 1.0023 \end{bmatrix}$$

It is important to remember when applying these methods the system may change at each time step. Thus the newly derived temperature must be used to derive a new right hand side of the above system. For example, for the leftmost node, $i = 1$ at time step $l = 1$, the derived node equation:

$$T_1^l + \left(\frac{k\Delta t}{(\Delta x)^2}\right) f_0(t^{l+1}) = \left(1 + \frac{k\Delta t}{(\Delta x)^2}\right) T_1^{l+1} - \left(\frac{k\Delta t}{(\Delta x)^2}\right) T_2^{l+1}$$

would become:

$$2.0047 + 0.020875*(100) = 1.04175*T_1^2 - 0.020875*T_2^2$$

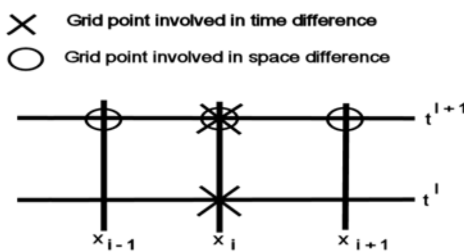
or

$$1.04175*T_1^1 - 0.020875*T_2^1 = 4.0922$$

Thus for $t = 0.2$ the system of equations would be:

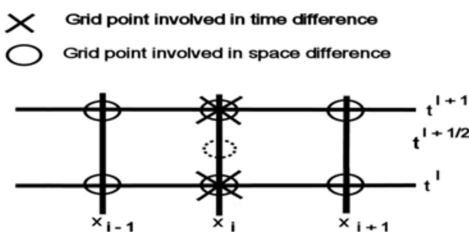
$$\begin{bmatrix} 1.04175 & -0.020875 & & \\ -0.020875 & 1.04175 & -0.020875 & \\ & -0.020875 & 1.04175 & -0.020875 \\ & & -0.020875 & 1.04175 \end{bmatrix} \begin{bmatrix} T_1^1 \\ T_2^1 \\ T_3^1 \\ T_4^1 \end{bmatrix} = \begin{bmatrix} 4.0922 \\ 0.0406 \\ 0.0209 \\ 2.0461 \end{bmatrix}$$

The computation molecule associated with this implicit method is described figure B10. Notice this corrects for some of the time-oversights of the explicit method.



B10 – Computational molecule for implicit method.

While this basic implicit method is stable and convergent it has problems of its own. Specifically it should seem rather odd that the temporal difference approximation is only first order accurate, while the spatial approximation is second order. Of course, depending on the problem such issues may or may not matter and there are ways to correct for them. One such correction would be to employ a Crank-Nicolson implicit method, which uses a half-step in the time dimension. A detailed description of this method would only add excessive length to this paper and will not be presented here. However for comparison purposes its computational molecule is described in figure B11.

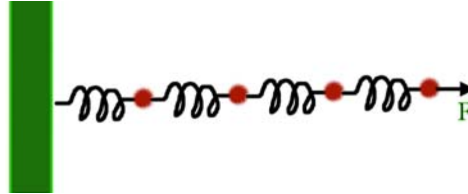


B11 – Computational model for Crank-Nelson implicit method.

APPENDIX C

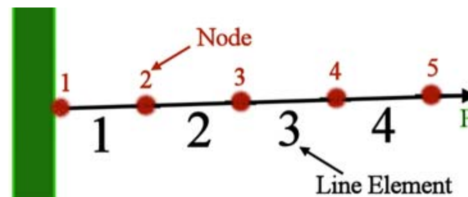
EXAMPLE OF FINITE ELEMENT METHOD

To demonstrate the steps of a FEM, consider a system of springs where one end is attached to a wall and the other end is subject to a constant force, ***F***. This is illustrated in the figure C1.



C1 – System of springs attached to a wall.

For the discretization step we will treat each spring as a line element. Thus we will have four elements and five nodes. This is pictured in figure C2.



C2 – Spring systems as four elements and five nodes.

Having broken our object into finite elements, we now must derive the solution system of equations for a single element. For this problem this system is found using a straight forward application of $F = kx$. Explicitly we will examine element 1. We will use F_1 to denote the force applied at node 1 and denote the force applied at node 2 as F_2. For the element we know the force is a constant, k, times the elongation or compression of the spring for the given element. Thus we have $F = k(x_1 - x_2)$, where x_1 is the displacement of node 1 from its rest position and x_2 is the displacement of node 2 from its rest position. Because the system is stationary we may infer that $F_2 = -F_1$. Hence we arrive at $F_1 = k(x_1 - x_2)$ and $F_2 = k(x_2 - x_1)$. So for line element 1 we have the system of equations in matrix form as:

$$\begin{bmatrix} k^{(1)} & -k^{(1)} \\ -k^{(1)} & k^{(1)} \end{bmatrix} \begin{bmatrix} x_1^{(1)} \\ x_2^{(1)} \end{bmatrix} = \begin{bmatrix} F_2^{(1)} \\ F_2^{(1)} \end{bmatrix}$$

Notice the superscript of (1) indicates each of these variables is associated and derived for element 1. This notation proves useful in the next step of assembly.

To assemble our system we must number each element to specify the topology of the system. This numbering will indicate how the elements are connected and allows us to transition from the local coordinates of an element to the global coordinates of the entire system. It should be noted we have already numbered our elements in such a fashion. We also have already adopted a notational convention in superscripting to indicate which variables come from which elements. In this notation the subscripts will

designate the row and column of the local element matrix in which the variable appeared. For example $k_{21}^{(3)}$ is the constant derived in element 3 that was in row 2 and column 1 of the element's local matrix. Thus our assembling step results in the following matrix:

$$\begin{bmatrix} k_{11}^{(1)} & -k_{12}^{(1)} & & & \\ -k_{21}^{(1)} & k_{22}^{(1)}+k_{11}^{(2)} & -k_{12}^{(2)} & & \\ & -k_{21}^{(2)} & k_{22}^{(2)}+k_{11}^{(3)} & -k_{12}^{(3)} & \\ & & -k_{21}^{(3)} & k_{22}^{(3)}+k_{11}^{(4)} & -k_{12}^{(4)} \\ & & & -k_{21}^{(4)} & k_{22}^{(4)} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} F_1^{(1)} \\ 0 \\ 0 \\ 0 \\ F_2^{(4)} \end{bmatrix}$$

We now must apply the boundary conditions. In this case the only significant condition is the attachment of one end of the spring system to a wall. We have assumed this attachment to be at node 1, thus $x_1 = 0$. We will also assume all the spring constants equal one. So our matrix system becomes:

$$\begin{bmatrix} 2 & -1 & & \\ -1 & 2 & -1 & \\ & -1 & 2 & -1 \\ & & -1 & 1 \end{bmatrix} \begin{bmatrix} x_2 \\ x_3 \\ x_4 \\ x_5 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ F \end{bmatrix}$$

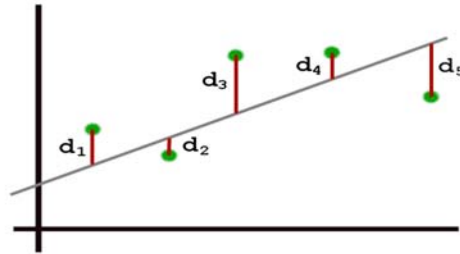
We may now complete the method by solving the above matrix system. This can be accomplished using any matrix solver, though one specialized for solving tridiagonal systems would be the best option.

APPENDIX D

LEAST SQUARES

To understand moving least squares (MLS) algorithms it is best to start with a review of the method of least squares. This is not an in depth coverage of the topic, it is designed to merely present a reminder to the reader of what the method of least squares does. We will present the method as it is used to find a line to approximate the relation between a set of 2D points. It should be understood this is for simplicity and the method is useful in solving a multitude of more complex problems.

So assume we are given five 2D points P_1 to P_5. We then plot the points and attempt to draw a line through them as shown in figure D1.



D1 – Line through a set of points.

Notice we assume we hit the *x* values precisely as we are assuming *y* is dependent on *x*. Thus we can measure the error of the line by measuring the distance the line is from the known *y* values. As in the picture above, we denote each error as d_i, where $i \in [1, 5]$. As we are looking for a line to approximate the data we then have an estimate in the algebraic form of $y = mx + b$. Thus, as the error distances are measured perpendicular to the x-axis, they may be expressed as $y_i - y = y_i - (mx_i + b)$. In the method of least squares the error this results in is measured in terms of the square of the Euclidean, or L^2, norm. Specifically the total error of the approximating line is calculated as:

$$E = \left\| y_i - (mx_i + b) \right\|^2$$
$$E = \sum_i \left(y_i - (mx_i + b) \right)^2$$

This of course is minimal when the partial derivatives are both equal to zero. So we solve:

$$\frac{\partial E}{\partial b} = 2\sum_i \left(y_i - (mx_i + b) \right) = 0$$
$$\frac{\partial E}{\partial m} = 2\sum_i x_i \left(y_i - (mx_i + b) \right) = 0$$

Which gives us a system of two equations and two unknowns (*m* and *b*):

$$\sum_i y_i = m\sum_i x_i + b\sum_i 1$$
$$\sum_i x_i y_i = m\sum_i x_i + b\sum_i x_i^2$$

Which reduce to a simpler form of:

$$bn + m\left(\sum_i x_i\right) = \sum_i y_i$$

$$b\left(\sum_i x_i\right) + m\left(\sum_i x_i^2\right) = \sum_i x_i y_i$$

Thus we simply calculate the sums:

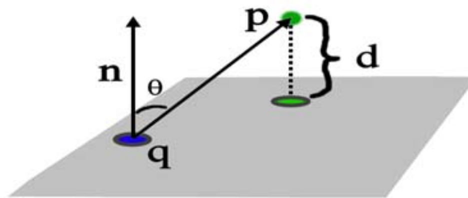
$$\sum_i x_i\ ,\quad \sum_i y_i\ ,\quad \sum_i x_i y_i\ ,\quad \sum_i x_i^2$$

And solve the resulting 2x2 system of equations.

Moving Least Squares

The concept of moving least squares (MLS) is similar to the above method of least squares in the error measurement being applied and the derivative techniques used to determine the final system of equations. In both methods the derivation of the solution system is unnecessary to actually apply the results. However understanding the ideas used to derive the solution is often useful in understanding how the technique can be applied.

In the above method of least squares we were trying to find the best fitting line to a set of 2D data points. For creating a surface we need to find a best fitting plane to a set of 3D points. To accomplish this we begin, as in the previous case, by defining a distance measure. The distance we will be measuring is the distance a point is from a plane. But first we need to define some variables, so consider the figure D2:



D2 - Point projected onto a plane.

Let the plane be defined by $ax + by + cz + d = 0$. Let **n** be the normal for the plane. Let **p** = (x_0, y_0, z_0) be a point in space. Let **q** be a point in the plane. Then the shortest distance from **p** to the plane is given by the dot product:

$$\frac{\mathbf{n}\cdot(\mathbf{p}-\mathbf{q})}{|\mathbf{n}|} \quad = \quad \frac{ax_0 + by_0 + cz_0 + d}{\sqrt{a^2 + b^2 + c^2}}$$

For our desired plane we will use the square of this distance and minimize it across every point in a given neighborhood. Thus we must find a, b, c, and d which minimize the function denoted:

$$f(a,b,c,d) = \sum_{i=1}^{n} \frac{\left(ax_i + by_i + cz_i + d\right)^2}{(a^2 + b^2 + c^2)}$$, where n = number of points being considered.

Taking the partial derivative of this function with respect to d and setting it equal to zero, allows us to solve for d in the following manner:

$$\begin{aligned}
\frac{\partial f}{\partial d} = \quad & \left(\frac{2}{(a^2+b^2+c^2)}\right)\sum_{i=1}^{n}\left(ax_i+by_i+cz_i+d\right) &= 0\\
& \sum_{i=1}^{n}\left(ax_i+by_i+cz_i+d\right) &= 0\\
& \sum_{i=1}^{n}\left(ax_i\right)+\sum_{i=1}^{n}\left(by_i\right)+\sum_{i=1}^{n}\left(cz_i\right)+\sum_{i=1}^{n}\left(d\right) &= 0\\
& nd = -\left(\sum_{i=1}^{n}\left(ax_i\right)+\sum_{i=1}^{n}\left(by_i\right)+\sum_{i=1}^{n}\left(cz_i\right)\right) &\\
& d = -\left(\frac{a\sum_{i=1}^{n}\left(x_i\right)}{n}+\frac{b\sum_{i=1}^{n}\left(y_i\right)}{n}+\frac{c\sum_{i=1}^{n}\left(z_i\right)}{n}\right) &
\end{aligned}$$

Thus if the mean position, or centroid, of the points being considered is $[\mu_x, \mu_y, \mu_z]^T$ we have:

$$d = -(a\mu_x + b\mu_y + c\mu_z)$$

and f becomes:

$$f(a,b,c,d) = \sum_{i=1}^{n}\frac{\left(a(x_i-\mu_x)+b(y_i-\mu_y)+c(z_i-\mu_z)+d\right)^2}{(a^2+b^2+c^2)}$$

We will now switch to a matrix form of notation and let **v** and **M** be defined as follows:

$$\mathbf{v} = \begin{bmatrix} a \\ b \\ c \end{bmatrix} \quad \text{and} \quad \mathbf{M} = \begin{bmatrix} x_1-\mu_x & y_1-\mu_y & z_1-\mu_z \\ x_2-\mu_x & y_2-\mu_y & z_2-\mu_z \\ \vdots & \vdots & \vdots \\ x_n-\mu_x & y_n-\mu_y & z_n-\mu_z \end{bmatrix}$$

Thus

$$\begin{aligned}
f(\mathbf{v}) &= \frac{\left(\mathbf{v}^T\mathbf{M}^T\right)\left(\mathbf{M}\mathbf{v}\right)}{\mathbf{v}^T\mathbf{v}}\\
&= \frac{\mathbf{v}^T\left(\mathbf{M}^T\mathbf{M}\right)\mathbf{v}}{\mathbf{v}^T\mathbf{v}}
\end{aligned}$$

We now define $\mathbf{A} = \mathbf{M}^T\mathbf{M}$ and note $\mathbf{A}/n$ is the covariance matrix of the data. We also note $f(\mathbf{v})$ is the Rayleigh quotient, which is known to be minimized by the eigenvector of **A** which corresponds to the smallest eigenvalue. Thus this minimizing vector must be the normal to the plane we are seeking and that plane must contain the mean position, or centroid, of the points being considered.

APPENDIX E

LENNARD JONES POTENTIALS

In Molecular Dynamics a major task is to solve Newton's equations of motion on a potential energy surface. One of the more commonly used representations of this potential in computer simulations is the Lennard-Jones (LJ) potential. This potential is defined as:

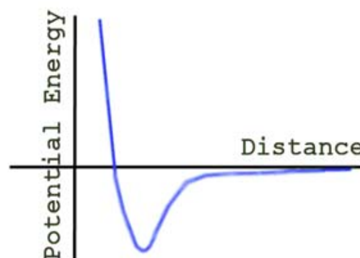
$$u(r) = 4\varepsilon\left(\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right)$$

In most simulations everything will be processed in a pair-wise fashion. Thus, developing the equations in terms of pairs of particles is necessary. From the above we note, for a pair of particles, the potential energy between particle *j* and *i* is defined as:

$$u(r_{ij}) = \begin{cases} 4\varepsilon\left(\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right) & r_{ij} < r_c \\ 0 & r_{ij} \geq r_c \end{cases}$$

Where $\mathbf{r}_i$ is the location in space of particle *i* and $r_{ij} = |\mathbf{r}_i - \mathbf{r}_j|$. In this relationship the particles will repel each other when close together, attract each other when slightly apart and eventually have no effect on one another at a distance of r_c. In this relation, the value of ε will control the strength of the relationship between the particles and σ will define a length scale. Another way to say this is: there is a negative well depth of ε and a steeply rising repulsive wall at $r_{ij} = \sigma$ [38].

In general the curve for this relationship has the form shown in figure E1:



E1 - General form of repulsive force between particles as a function of distance.

For simplification purposes r_c is usually chosen to be $2^{1/6}\sigma$, and the attractive tail is ignored. The choice of r_c is such that $u(r_c) = 0$. Thus the effective simulation equation becomes:

$$u(r_{ij}) = \begin{cases} 4\varepsilon\left(\left(\frac{\sigma}{r_{ij}}\right)^{12} - \left(\frac{\sigma}{r_{ij}}\right)^{6}\right) + \varepsilon & r_{ij} < 2^{1/6}\sigma \\ 0 & r_{ij} \geq 2^{1/6}\sigma \end{cases}$$

Given $r_{ij} < r_c$ then the vector force particle j exerts on particle i is:

$$\mathbf{f}_{ij} = -\nabla \mathrm{u}(r_{ij}) = \left(\frac{48\varepsilon}{\sigma^2}\right)\left[\left(\frac{\sigma}{r_{ij}}\right)^{14} - \left(\frac{1}{2}\right)\left(\frac{\sigma}{r_{ij}}\right)^{8}\right]\mathbf{r}_{ij}$$

If $r_{ij} \geq r_c$ then the force is zero.

From the above equations the motion of the particles is derived from F = $m\mathbf{a}$, or more specifically:

$$m\mathbf{a_i} = \mathbf{f_i} = \sum_{\substack{j=1 \\ j \neq i}}^{n} \mathbf{f_{ij}}$$

where n is the total number of particles in the system, m_i is the mass of particle i and $\boldsymbol{a}_i$ is the acceleration of particle i. We use a form of this equation to calculate $\boldsymbol{a}_i$ and apply physically based modeling techniques to move our particles around. Note that while our immediate description involves 2D entities, none of the above equations are so limited.

A variety of sources provide more detail on Lennard-Jones forces. An excellent one is [38] which discusses Lennard-Jones potentials and molecular dynamics.

The reason we have chosen to use an LJ based method for controlling our particles is to assist in the dense packing [63]. LJ forces are effective at packing molecules in the real world, so we would expect them to be effective packing spheres into objects, as well. The forces also provide a way to keep the neighbors of particles from changing too frequently. These properties assist in the overall performance of the method. Without LJ forces, and with only collisions providing forces, the particles tend to bounce around and do not easily form themselves into anything. Simply stated, the attractive forces are literally the glue that maintains a semblance of coherence. While the object boundary will eventually force this behavior, the particles do not tend to be so nicely arranged without such a glue.

APPENDIX F

SMOOTHED PARTICLE HYDRODYNAMICS (SPH)

Particle systems have been used to model fluids. Such models have been adapted to model gasses and visco-elastic solids, though not all at the same time. In this section we present a brief history of particle modeling techniques which incorporate Smooth Particle Hydrodynamics (SPH). We also offer a simplified overview of the processes involved. For a more detailed presentation we suggest a thorough reading of the associated papers [7, 36, 128, 134, 158-161] and their references.

F.1 SPH History

One of the recent adaptations using particles to model fluids, or more accurately gooey liquids, is based on Smooth Particles Hydrodynamics (SPH). However, SPH methods are not localized to computer science and have a long history dating back to the early 19th century. The history begins with fluid equations developed by Navier and Stokes and progresses through a chain of adaptations and applications. The fluid equations, in conjunction with the conservation of energy and Newton's Third Law of Motion, may be used to model an incompressible fluid. Putting these properties together was first done by astrophysicists to model cosmological fluids in 1977 [158, 162, 163].

The development of SPH in computer graphics took a little while longer. However it, like many other particle-based approaches, began its course when Reeves introduced particle systems controlled by stochastic processes [7]. As described in 2.6 this led to the use of particles in many different directions. One of those was to model clay-like, deformable, inelastic objects [128, 159]. This work was later improved by applying SPH methods [160]. Various others have improved upon this technique and in 2003, Muller et al. demonstrated SPH methods could model liquids at reasonably interactive rates [134]. While these methods all use volumetric particle systems, they limit themselves to fluids. The achievement of clay-like behavior is accomplished by employing a large value for the viscosity of the fluid. The melting of objects is accomplished by reducing the viscosity. Reversing the melting is not so easily accomplished as increasing the viscosity results in odd visual behavior, and frozen objects are deformable rather than rigid.

F.2 SPH Limitations

While SPH particle methods are similar to the work of this dissertation, they have some striking differences and limitations. One such distinction is the SPH methods offer no means for initial object creation, most deal with simple cubes, rings or spheres. Another difference occurs in surface representation and collision detection. Instead of the actual particles, the SPH methods use a derived iso-surface for the display of the objects, and in some cases for collision detection. They further are limited to modeling fluid-based objects. When used to model deformable solid objects they are actually modeling very sticky pastes rather than solid objects. Rigid objects and effects such as fracture and breaking are not supported. Further, modeling gasses and liquids using the same engine is not directly supported, mostly as the surface is not easily defined for gasses and the viscosity and compressibility behavior changes. SPH methods also do not freeze objects. This limitation is in two forms. The first is the best they could freeze an object into is a deformable solid (which substances such as ice are not). The second is the behavior of a liquid as it becomes more viscous produces odd visual behaviors as the object becomes "rubbery."

These limitations are not meant to diminish the strength of SPH methods, but are intended to distinguish them from the work of this dissertation. It is possible to modify the SPH methods to model gasses, and hybridization may allow more rigid representations of solids. The SPH models, while mostly Lagrangian (non-grid based) in nature, may also be used in Semi-Lagrangian particle systems which might further enhance their modeling capabilities. Yet in all of these everything, in the end, is being modeled as a fluid using the equations describing fluid-like motion. In our work everything is being modeled as a

composition of particles, where the interaction of the individual particles determines the overall motion of a substance. This distinction is similar to that of describing the motion of a herd of animals versus describing how each animal moves within a herd. One is a visualization of a pre-described (variable) outcome. The other is a visualization of emergent behavior. Both produce viable results.

F.3 SPH Method Overview

To better understand SPH methods we include here a brief overview of the details involved. This is not a complete description, but should give a general understanding of the technique as used in computer graphics.100

SPH methods typically qualify as a Lagrangian simulation of fluids. In such simulations the fluid is represented by a set of particles, where SPH methods are used to calculate each value of a fluid's variables, such as density. These values may be computed at arbitrary positions in the fluid by averaging the values over a set of particles near the point of interest. This is similar to FDM or FEM methods, but with a much less structured set of elements and less guarantee of continuity, as the elements themselves move and reorganize. This technique is also in contrast to Eulerian based methods, which use a fixed grid and determine values within each grid cell as the liquid moves through them.

The primary equation used in SPH methods, as stated previously, is derived from the Navier-Stokes equation of fluid motion. As presented by Muller et al. [134] for incompressible fluids this equation is:

$$\rho\left(\frac{\partial v}{\partial t} + v \cdot \nabla v\right) = -\nabla P + \rho g + \mu \nabla^2 v$$

where ρ is the density field of the fluid, v is the velocity field, P is the pressure field, t is time and μ is a viscosity constant. The ρg is considered an external force density field such as gravity. The convective term is $v \nabla v$. The viscosity term is $\mu \nabla^2 v$. The force per unit volume caused by pressure is represented by - ∇P.

In Eulerian methods the convective term $v \nabla v$ is required as the grid is fixed in space, while the fluid passes through it. In SPH methods this term is not necessary as the particles are moving with the fluid. It should be noted viscosity term, $\mu \nabla^2 v$, smoothes the velocity field of the fluid.

The force per unit volume due to pressure, - ∇P, is ideally related to the density by the gas state equation:

$$p = k\rho$$

where k is a constant dependent on the gas's temperature. This relationship may be adapted to simulate an incompressible fluid [160] by altering the relation to be:

$$p = k(\rho - \rho_0)$$

This causes the fluid's density to tend toward the value of ρ_0. As such, this value is referred to as the rest density.

The second important equation in SPH is the conservation of mass. This can be stated as:

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho v) = 0$$

However, when using a particle based approach with the particles representing the mass of the substance, this equation is usually viewed as not necessary and may be omitted [134]. However, if this is done

without care the interpolatory evaluation of some values may be misleading when the particles separate in significant fashion. More specifically if the volume becomes large enough that the particles are not offering a significantly dense sample therein, the accuracy of the interpolation of values from nearby points declines.

The last relation of importance to SPH is defined by Newton's Third Law of Motion, which states, "for every action there is an equal and opposite reaction." This means forces between particles must be dealt with correctly. More specifically the net force between any particle pair must be zero. This is accomplished by setting the pressure and viscosity forces between two given particles equal and opposite to one another based on their relative velocities [161].

To complete the SPH method a kernel must be selected for the interpolation of values to be performed. The selected kernel must assign a weight to a particle based on the vector between the kernel's center and the given particle, where the kernel is centered at the point of interest. In this method a support radius, h, is also provided. Thus the kernels may be designed to return zero values for vectors of length greater than h. This is referred to as a kernel with compact support. This property is desirable in particle based simulations as it allows only particles within a distance of h to be considered for evaluating a value at the point in space specified. The selection of a kernel is varied and will greatly influence the performance and results of SPH methods. Often multiple kernels will be used within a given program, depending on the task to be performed. Examples of such kernels can be found in the papers by Monaghan, Desbrun and Gascuel and Müller et. al [134, 158, 160].

VITA

Name: Brent Michael Dingle

Address: Department of Computer Science, Texas A&M University, TAMU 3112
College Station, TX 77843-3112

Email Address: elganid.A.T.gmail.com

Education: B.S., Mathematics and Computer Science, Bradley University, Peoria, 1995.
M.S., Mathematics, Texas A&M University, College Station, 2000.
Ph.D., Computer Science, Texas A&M University, College Station, 2007.