PHYSICAL MODELING OF NON-RIGID OBJECTS AND INTERACTIVE DISPLAY USING BSP TREES

APPROVED:

Supervisor: ______________________________

Donald S. Fussell

____

Mohamed G. Gouda

To my father.

PHYSICAL MODELING OF NON-RIGID OBJECTS AND INTERACTIVE DISPLAY USING BSP TREES

by

ANDREAS SPIEGL

THESIS

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

MASTER OF ARTS

THE UNIVERSITY OF TEXAS AT AUSTIN

August, 1993

Acknowledgments

I would like to thank my supervisor, Donald S. Fussell, for his guidance and editorial help during this work. Gratitude is extended to Jihun Park for introducing me to the topic of physical modeling and for many helpful academic discussions.

I am deeply indebted to Brian Zoller for numerous hours spent together in the lab, for always patiently listening to my problems, for providing many invaluable hints and for all the good time we had outside of school. Many thanks to all my friends in Austin who made this year an unforgettable exerience for me.

Patrick Wunsch deserves my undying gratitude for many years of friendship. His encouragements, clarifications and constructive critique improved this work by a great deal.

The love, support and faith my parents have shown in me over the years is fundamental to my success. I could never thank them enough.

My greatest thanks go to my fiancée Heidi Kaufmann. This thesis would not be possible without her love and emotional support.

ANDREAS SPIEGL

The University of Texas at Austin

August, 1993

Abstract

PHYSICAL MODELING OF NON-RIGID OBJECTS AND INTERACTIVE DISPLAY USING BSP TREES

by

ANDREAS SPIEGL

SUPERVISING PROFESSOR: Donald S. Fussell

In recent years realistic modeling of 3D-objects has become an active area of computer graphics research. Early approaches modeled objects as rigid bodies, a simplification that yields satisfactory results only in a few special cases. In order to achieve a realistic simulation, physical properties allowing deformations of the modeled object have to be taken into account.

This thesis presents a modeling system and an animation tool designed to simulate and analyze physical behavior of elastic, 3-dimensional objects in a realistic fashion. The modeling system represents objects in terms of point masses, springs and hinges. Unlike previous approaches to this problem, our model imposes no restrictions on an object's deformation.

The animation tool represents objects as binary space partitioning trees, a data structure that allows us to efficienty determine surface visiblity.

Along with a flat-shading color model, every frame of the animation is rendered in real time. Each scene can be inspected from any desired viewpoint, thus providing a tool for exact analysis of motion and deformation of modeled objects. An intuitive graphical user interface supports easy handling.

Table of Contents

List of Figures

Chapter 1

Introduction

Law of Selective Gravity:

"An object will fall so as to do the most damage."

1.1 Background

In the past few years, there has been increasing interest in the use of physical simulation as a means of producing animation. Creating realistic animations raises many problems related to modeling objects that change in time due to influences from the environment. A very natural and basic approach for research in this area is to employ a finite element method with mass-spring lattices [ZT89]. This is based on the attempt to approximate the molecular structure of objects. Unfortunately, such systems tend to be computationally very expensive, primarily since they often lead to numerically stiff equations. Haumann describes this approach in [H87]; Thingvold and Cohen apply a modified version in [TC90].

A number of authors have investigated models based on global and local deformation functions, which proved to be a more efficient method [B84] [TPBF87] [BW92]. Although this technique promises much performance improvement, it also imposes loss of generality (e.g. restrictions to linear deformations in [WW90]).

The work for this thesis is based on the first approach. This approach is fundamentally more "natural" than others which is the determining factor for choosing this method. The advantages of a very realistic simulation and the wealth of impressive results outweigh the complexity of the method and the computational expense. In addition, a strong effort has been made to improve the performance (see section 1.3).

A second goal of this thesis is realistic rendering of an animated sequence. The binary space-partitioning (BSP) tree algorithm is employed to achieve this goal. It was first developed by Fuchs, Kedem, and Naylor [FKN80] [FAG83] and has since been used for rendering purposes [NT86] [J87], as well as for shadow generation [CF92] [C91] and solid geometry [N90] [NAT90].

1.2 Previous Work

The physical modeling system developed in this thesis is based on an implementation which was done for a term project [ZS92]. A mesh of masses and springs was implemented in order to approximate soft, cubic objects of arbitrary size. With the aid of an Fourth-Order Runge-Kutta method, the resulting differential equations were solved in order to animate the model. The granularity of the mesh as well as the choice of an appropriate step size for the numerical integrator had great influence on the realism of the simulation. Unfortunately, the stiff differential equations yielded great computational expense.

So that the animation could be rendered for examination, a separate program was developed to display the precomputed movie with flat shaded surfaces and hidden lines removed. BSP trees (see section 4.1) were used to allow for fast hidden surface removal.

In addition, with the aid of VOGL[1], a powerful user interface was written. A 3 button mouse as input device allowed the user to move through space with 6 degrees of freedom. Full control over the playback of the movie was provided.

1.3 Improvements

Research for this thesis is founded on the above mentioned term project. Some basic ideas have been reused, but generalizations and major enhancements have been added. What follows is a brief description of the improvements of this work over the term project.

The point mass - spring model has been enhanced to a point mass - spring - hinge model (*PSH-model*). It allows for hinges whose purpose it is to constrain angles between 3 distinct point masses. The advantage of hinges is twofold: First, the notion of hinges reflects the natural phenomenon of preserving angles between finite elements. Secondly, and more importantly, the PSH-model allows considerable reduction in the number of point masses. The original model calls for distributing point masses throughout the whole region encompassed by the object. That is, the number of point masses was proportional to the volume of the object. The *PSH-model*, however is capable of modeling objects by sampling point masses only on their surfaces. Springs keep the model from breaking into pieces, while hinges prevent it from collapsing and losing its shape. Thus, the number of points in the *PSH-model* is only proportional to the size of the object's surface as opposed to its volume. This fact has a major influence on the amount of computation necessary for the

[1]VOGL is a library of C routines which try to allow a programmer to write programs that can easily be ported to machines with the Silicon Graphics GL library [E91].

simulation, and this is the point where a notable speedup over the previous implementation has been accomplished.

The most considerable speedup was achieved by changing the numerical treatment of differential equations. Previously, the system of n differential equations (where n is the number of point masses in the model) was broken down into n separate equations, which were then solved independently. When solving a differential equation, the numerical integrator requires multiple function evaluations. One function evaluation corresponds to examining all the constraints in the system which influence the net force upon the respective point mass. Since some constraints have to be examined several times when evaluating the function for neighboring point masses, this method led to many redundant computations. Therefore the n differential equations are now treated as an n-dimensional system and are solved concurrently. Here, a function evaluation is equivalent to examining *all* the constraints in the system and determining the net force upon *each* point mass. Although this is definitely a more complex task, the change constitutes a notable optimization, since only a constant[2] number of function evaluations is required for every timestep.

Other changes include:

- Generalization to allow for more complex objects than cubes.
- Automated generation of input data from various data formats. (with optional refinement to achieve higher accuracy)
- Adaption of the BSP tree algorithms to handle the more general models.

[2]The constant depends on the order of the method. For the Fourth-Order Runge-Kutta method applied in this thesis, the constant is 4.

- Support for special effects like wind simulation, fixed position of points, motion in viscous medium, variable direction of gravity.

- Frames can be appended to the end of the movie, i.e. simulations can be continued without the need to recompute from the beginning.

Chapter 2

Elements of the Model

The goal of this thesis is to provide means to simulate the physical behavior of any 3D object approximately represented by a combination of polyhedra and polylines in 3-space. To achieve this goal in a realistic fashion, a computational model for objects must be found. This model has to consist of only a finite number of elements. It must also be able to provide all necessary information for numerical treatment of the resulting differential equations.

This chapter describes a point mass - spring - hinge model, referred to as *PSH-model*, which is well-suited to determine the motion of objects and their interaction with each other. This model was developed as an enhancement of the point mass - spring model which was employed in the term project mentioned in section 1.2. It has many similarities with commonly used physical models, such as in [H87] and [TC90].

2.1 Points

A "Point" represents the abstraction often used in physics known as a "point mass." It has no size but does have a finite mass. In the *PSH-model* all points have the same mass since it is assumed that the modeled objects are homogeneous.

A point is also associated with a position and a velocity vector. However, the orientation of a point mass does not have any meaning in the *PSH-model*; all forces are broken down into linear components so that a point will never have any rotation.

2.2 Springs

The model of a spring defines a distance relationship between two points. This relationship is the force that tries to prevent the object from falling apart. The springs in the *PSH-model* follow a modified version of the formulation first presented by Hooke [HR60]. The formulas are often referred to as the "ideal spring equations," "Hookean equations" or "Hooke's laws." First, the basic formula:

$$F = -kd \tag{2.1}$$

where F is the force applied to the springs, d is the distance the spring is displaced from its equilibrium distance, and k is the spring constant, the factor that indicates the strength or stiffness of the spring. Note that F, the force, is a scalar in the above equation. It is turned into a force vector by finding the norm of the spring's axis vector and multiplying it by F. This resulting force vector is applied to both point masses P_0 and P_1 connected to the spring, but in opposite directions. This equation conserves linear momentum, since equal but opposite forces are applied to each of the endpoints. No torque is involved, so angular momentum is not a concern.

The equilibrium length of a spring is the length at which it exerts no forces to its endpoints P_0 and P_1. The displacement is the difference between the equilibrium length and the actual distance between P_0 and P_1. A spring will always attempt to move towards its equilibrium state.

The modified formula of the basic spring equation as defined in the

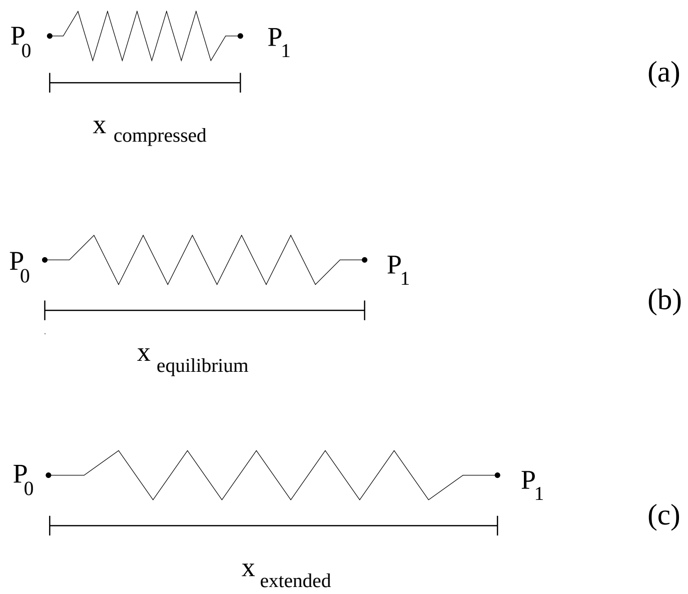

Figure 2.1: (a) compressed (b) equilibrium state (c) extended

PSH-model uses a percentage of the change in length of the spring rather than the absolute displacement:

$$F = -k\frac{x - x_0}{x_0} \tag{2.2}$$

where x_0 is the equilibrium length of the spring and x is the current length of the spring. Here, the force a spring exerts when stretched is relative to the original length of the spring. This modification is a more accurate description of physical springs: a small spring of a given material will resist a unit change in length more than a large spring of the same material.

2.3 Hinges

Hinges are analogous to springs. However, they define an orientation relationship between three points as opposed to springs which maintain a distance relationship between two points. Since hinges maintain angular relationships, torques are involved, but a careful definition of the hinge constraint keeps angular momentum considerations at a minimum. Haumann's [H87] triangularized models leads to a definition of the hinge using the common edge between two triangles and maintaining an equilibrium angle using the normals of the two triangles. However, to conserve angular momentum, Haumann's definition requires solving six equations in six unknowns. Therefore he assumes that all triangles will generally be equilateral. This assumption simplifies the problem to only one equation. However, since this assumption is very inaccurate for the *PSH-model*, approximation to realism would be lost. For this reason the *PSH-model* uses a different approach.

The three points A, B, C of a hinge (see figure 2.2) define a plane, in which all hinge forces are applied. The angle formed by the hinge edges AB and AC is compared with the equilibrium angle to determine the displacement of the angle. A is the hinge's apex, while B and C are its endpoints. The forces are applied to the endpoints such that they rotate about the apex (in the hinge plane). Since the endpoints always rotate about the same point, angular momentum is conserved.

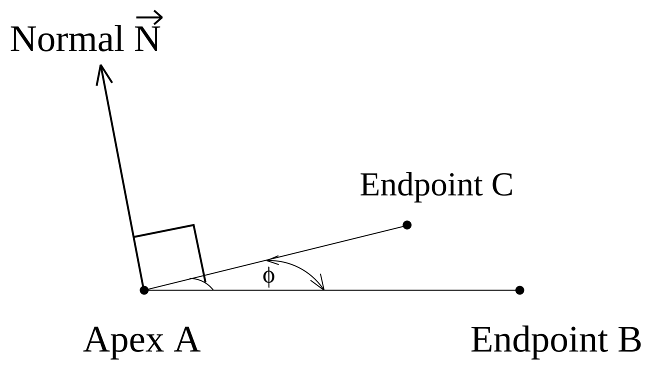

Figure 2.2: A hinge

Computationally, it is not possible to allow the hinge to cover 360 degrees of motion unless an "orientation" of the hinge is determined. Differentiating between three points that form an 80 degree angle and three points that form an 280 degree angle is otherwise impossible. To provide such an orientation in the hinge definition, the normal $\vec{N}$ of the hinge plane is considered, calculated as the cross product of the two edges of the hinge. Since the *PSH-model* is applied within a time-discrete numerical integration technique (see section 3.2), $\vec{N}$ can be compared to the hinge normal from the previous timestep. If $\vec{N}$'s orientation is opposite to the direction of the previous normal, we know that the hinge changed from an angle less than 180 degrees to one greater than 180 degrees (or reverse). Thus, now the larger (or smaller) angle has to be chosen. This is achieved by subtracting the value of the angle from 360 degrees. The current normal has to be stored for the computation of the forces at the next timestep of the numerical integration, along with a flag which indicates the current state (less than or greater than 180 degrees)

When A, B, and C are collinear, they do not determine a plane. In this case the *PSH-model* again takes advantage of the hinge normal from the previous timestep to determine the direction of the forces that are to be applied. So that a previous normal is always available, it has to be initialized before the first timestep of the numerical integration. The initial positions of the points are used for this purpose, and all angles are assumed to be acute or obtuse angles.

The force calculations in the hinge model are similar to the force calculations in the spring model. The angle between the three points is compared against the equilibrium angle and a torque is calculated by the following equation:

$$T = -k\frac{\phi - \phi_0}{\phi_0} \tag{2.3}$$

where T is the torque, k is the hinge constant, ϕ is the current angle of the

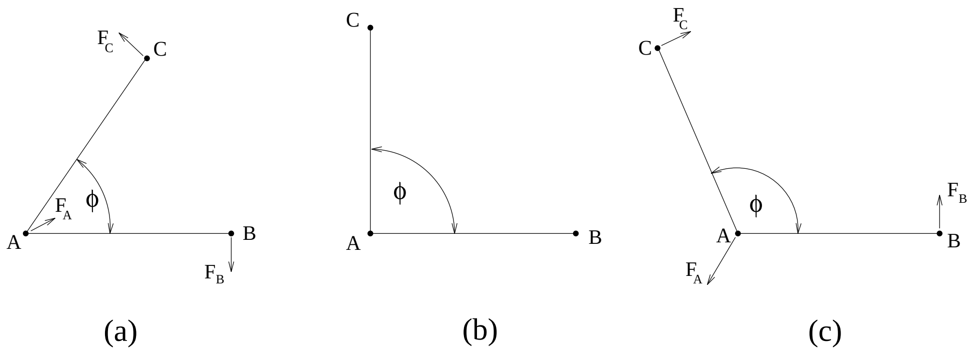

Figure 2.3: (a) compressed (b) equilibrium state (c) extended

hinge, and ϕ_0 is the equilibrium angle of the hinge. In the above equation T is a scalar. Forces are calculated from T as follows: The values of T is divided by the distance between apex A and endpoint B. The resulting value is the magnitude of the force. It is applied to B in a direction orthogonal to the vector from A to B and in the plane of the hinge. This process is then repeated for C since each endpoint may be at a different distance from the apex. In order to conserve momentum, the forces F_B and F_C that were applied to the endpoints are added, and the negation is then applied to the apex of the hinge (see equation 2.4 and figure 2.3). Since all the forces are applied in the same plane, every angular momentum is accounted for in this process.

$$F_A = -(F_B + F_C) \tag{2.4}$$

2.4 Damping

The assumption that springs are ideal, i.e. without internal friction, is not very realistic. Using the slightly changed Hookean spring as defined in section 2.2 does not yield a "natural" behavior of the model. Therefore a damping factor is added to the spring equation. A damping force is analogous to a friction force. It is the internal resistance to motion of a spring and is proportional to the relative velocity of the endpoints of the spring. The spring equation is modified as follows:

$$F = -k\frac{x - x_0}{x_0} - b\frac{dx}{dt} \tag{2.5}$$

This force is calculated using the velocity values from the previous timestep of the numerical integration. The change in length of the spring is computed by subtracting the velocities of the endpoints along the spring axis. It is then multiplied by b which represents the damping constant.

2.5 Faces

The *PSH-model*, as defined so far, is sufficient for analyzing the behavior of objects. However, the model also requires the definition of polygons as the object's faces. In order to apply shading and hidden line algorithms for the rendering of an animation (see chapter 4) it is necessary to know which sets of points constitute the faces of the object. It would require global analysis algorithms to determine the shape of the object automatically, and in this thesis no attempt has been made to do so.

Therefore the user defines polygons, referred to as "faces". These polygons can consist of an arbitrary number of points, its vertices. It is not required that the polygons be planar.

Chapter 3

Modeling Motion

The essential elements of the *PSH-model* have now been defined. Once the points of an object have been specified, along with the connecting springs and hinges, enough information exists to compute the motion of the object. What follows is a description of how this model is employed in the program *jello* to determine the physical behavior of the object.

3.1 Physical Background

The most basic formula in physics used to describe motion is the simple Newtonian Formula:

$$F = ma \tag{3.1}$$

where F is the force that corresponds to the acceleration a of a mass m. The definition of the springs and hinges in the *PSH-model* allows one to compute the forces that they create and which are to be applied to the points. Springs and hinges have one addititional property that make them very desirable for the simulation of motion: at any given time the force that a spring or hinge exerts upon a set of points depends solely on the position of the points. Hence, it is possible to examine each spring in the system and to determine the force each one applies to the point masses it is connected to, and it does not matter in what order the springs are examined. The forces in *jello* are expressed in Newtons per meter squared, masses are in kilograms, and distance is expressed in meters. Any other standard units of measurement could be used, and this choice would not have a direct effect on the framework of the physical calculations. The only case in which it is necessary to know the measurement system is when declaring certain system properties, such as gravity or object mass.

In order to compute the new position of a point after a certain amount of time has elapsed, the Newtonian Formula is solved for the acceleration a and then integrated twice with respect to time. The following equations illustrate these relationships:

$$a = \frac{F}{m} \quad \Longleftrightarrow \quad \frac{\delta v}{\delta t} = \frac{F}{m} \quad \Longleftrightarrow \quad \frac{\delta^2 x}{\delta t^2} = \frac{F}{m} \tag{3.2}$$

where a is the acceleration of a point, v its velocity, and x its position. F can be broken down into:

$$F = F_{springs} + F_{dampers} + F_{hinges} + F_{external} \tag{3.3}$$

where $F_{external}$ are all other forces exerted upon the point masses. These are forces which result from gravity, from collision with the floor and walls, from friction when sliding along the floor and walls, and from others which are listed in section 3.3.

The mass of the object M has to be defined by the user of *jello* and indirectly determines the mass m of each point. Homogeneity is assumed, thus m is obtained by dividing M by the number of points comprising the object. Since formula 3.2 calls for the division of the force by the mass of the point, m should not be a value near zero. This would yield a very large value for the acceleration, which could cause instabilities to occur during the numerical integration.

3.2 Solving the Differential Equation

3.2.1 Numerical Methods

Having specified the mass and computed the net force upon a point, this information can now be used to determine the movement by solving the differential equation for x. Any number of numerical integration methods could be used, including Euler's Method, Taylor Series approximations, or even one of the various multistep methods. For the implementation of *jello*, a Fourth-Order Runge-Kutta method was chosen. For, although the evaluation of the derivative function is computationally rather expensive, high accuracy is required for the integration of these stiff differential equations. Small errors are sufficient to introduce instabilities in the numerical solution, which cause the model to literally "explode".

3.2.2 Fourth-Order Runge-Kutta Method

With $v()$ and $a()$ being the acceleration and velocity values calculated for a given timestep and state of the system, the Fourth-Order Runge-Kutta method has the following form:

$$x_{i+1} = x_i + \frac{1}{6}(k_1^x + 2 \cdot k_2^x + 2 \cdot k_3^x + k_4^x) \tag{3.4}$$

where

$$\begin{aligned} k_1^x &= \Delta t \cdot v(t_i, x_i) \\ k_2^x &= \Delta t \cdot v(t_i + \tfrac{\Delta t}{2}, x_i + \tfrac{k_1^x}{2}) \\ k_3^x &= \Delta t \cdot v(t_i + \tfrac{\Delta t}{2}, x_i + \tfrac{k_2^x}{2}) \\ k_4^x &= \Delta t \cdot v(t_i + \Delta t, x_i + k_3^x) \end{aligned}$$

and analogously:

$$v_{i+1} = v_i + \frac{1}{6}(k_1^v + 2 \cdot k_2^v + 2 \cdot k_3^v + k_4^v) \tag{3.5}$$

where

$$\begin{aligned} k_1^v &= \Delta t \cdot a(t_i, v_i) \\ k_2^v &= \Delta t \cdot a(t_i + \tfrac{\Delta t}{2}, v_i + \tfrac{k_1^v}{2}) \\ k_3^v &= \Delta t \cdot a(t_i + \tfrac{\Delta t}{2}, v_i + \tfrac{k_2^v}{2}) \\ k_4^v &= \Delta t \cdot a(t_i + \Delta t, v_i + k_3^v) \end{aligned}$$

t_i is the value for the current time

Δt is the step size of the integration

x_i is the position of the point mass at time t_i

v_i is the velocity of the point mass at time t_i

Any single step method requires as many evaluations of the derivative function per iteration step as the order of the method. Since this Runge-Kutta method is of order four, four function evaluations are required. One function evaluation is equivalent to examining all the constraints in the system and determining the net force upon each point mass.

In the term project mentioned in section 1.2, this method was used in a rather inefficient fashion. The numerical integrator was employed for each point mass by itself. This led to redundant computations: forces had to be recomputed since the net force exerted upon one point depends on position the and speed of all its neighboring points.

3.3 Special Effects

To make the motion of objects look somewhat interesting, and even to get it moving in the first place, some additional (external) forces have to be considered. There are many possibilities, but only some basic ones have been chosen for the implementation of *jello*.

3.3.1 Floor and Walls

Three boundaries are implemented in the current version of *jello*. These are simply the three planes defined by

$$x = 0 \qquad y = 0 \qquad z = 0 \tag{3.6}$$

Although more complex constraints could easily be implemented, the collision detection routines (located in the most time critical parts of the program) would have a very negative effect on the execution time. However, slanted surfaces can be simulated easily by changing the direction of the gravity force. (see section 3.3.3)

3.3.2 Friction

When the object collides with a surface, it experiences not only forces that make it bounce off of the surface, but there are also friction forces which try to resist its sliding along the surface. This force is applied to a point mass as a force vector parallel to the surface, and its value depends on the speed of the point mass parallel to the surface. Thus, if the object hits for example the $(y = 0)$ plane, the resulting friction force is computed as follows:

$$\begin{aligned} F_x &= -f \cdot v_x \\ F_y &= 0 \\ F_z &= -f \cdot v_z \end{aligned} \tag{3.7}$$

3.3.3 Gravity

In physically based modeling it is common practice to represent gravity as a constant force in one direction rather than to use the inverse square law of gravitational attraction. Since objects which are modeled in *jello* are tiny compared to "the earth" which is the source for the gravitation, this simplification is sufficient here as well. As mentioned above, there are cases where it might be desirable to change the direction of gravity. Therefore, gravity is specified as a user-definable 3D vector (see section 5.2). A small value for gravity has the effect that objects appear to be at a greater distance, thus moving slower. This adjustment can be helpful when the system is numerically unstable, as reducing the forces that act upon the object reduces the acceleration and therefore improves the stability.

3.3.4 Initial Speed

It is useful to extend the definition of point masses to allow for initial velocity values. As an example, the simulation of a free fall before the object hits a surface can be skipped. Positioning the object close to the surface and setting the velocity to the intermediate speed at that height, gives the same result.

3.3.5 Fixed Points

By constraining the motion of a subset of points in the object, one can study the effect on the rest of the points, which will follow the motion according to the constraints acting upon them. A special case of this has been implemented in *jello*: point masses can be constrained to be "fixed". This provides a simple way to anchor a set of points so the other points may react to gravity without the whole system going into a free fall. As examples will show later in this thesis, this allows to simulate some realistic effects.

3.3.6 Wind

So that gravity is not the only driving force of motion, the possibility has been added to define a constant wind force. It is important to note, however that specifying too large of an additional force reduces the numerical stability of the modeled system.

3.3.7 Viscosity

Another form of damping used to reduce the velocity of point masses is to cut the velocity by some percentage at each timestep. This method causes the object to move as if it were in a viscous medium. Although this is not physically correct, it is a useful way to damp out most of the vibrations that occur in a modeling session and reach a minimum force configuration. This viscosity constant can be specified by the user through the input data file. (see section 5.2)

Chapter 4

Enhancing the Animation

After many iterations of the numerical differential equation solver, the motion of the object has been computed and an animation of it can be displayed on the screen. So far, however, it is only possible to draw the outline of the object as a wireframe. To be able to enhance the viewing, it is necessary to apply a hidden surface algorithm. Many approaches could be chosen to tackle this task. For the implementation of *shade* — the logical next step after the execution of *jello* — a BSP tree approach has been chosen.

4.1 BSP Trees

Foley and Van Dam give a very concise description of the notion of BSP trees in [FD82]:

> "The binary space-partitioning (BSP) tree algorithm, developed by Fuchs, Kedem, and Naylor [FKN80] [FAG83], is an extremely efficient method for calculating the visibility relationships among a static group of 3D polygons as seen from an arbitrary viewpoint. It trades off an initial time- and space-intensive preprocessing step against a linear display algorithm that is executed whenever a new viewing specification is desired. Thus, the algorithm is well suited for applications where the viewpoint changes, but objects do not."

Since *shade* was designed to provide a convenient user interface so that it is possible to view the motion of the object from any viewpoint, BSP trees were chosen as the means to provide the characteristics of a hidden surface algorithm. Since the object to be displayed is dynamic, however, a separate BSP tree has to be computed for every frame of the movie[1]. Although this may seem very costly (and it definitively is), the reward is certainly worth it. What follows is an outline of how binary space-partitioning trees are created and used. A more complete description can be found in e.g. [FD82].

4.1.1 How to create a BSP tree

Probably the most intuitive way to explain what a BSP tree is, is by describing how to construct one. Given a region R in 3-space[2], one chooses some hyperplane H that partitions R into 2 subregions R^+ and R^-. These are referred to as the positive and negative subregion (with respect to H). Each of these can in turn be partitioned by choosing an appropriate hyperplane. Then, arranging the resulting partitionings in a parent-child hierarchy produces a binary tree of 3-space regions. The internal nodes represent the partitioning planes; its leaves are convex volumes of space determined by the partitioning.

Example Figure 4.1 shows a 2-dimensional example, in which line segments represent polygons. Every polygon is supplied with a normal vector. The chosen orientation of the normal vector is typically arbitrary. The subregion which the normal points to is called the positive subregion, the other one is the negative. The figure illustrates the relationship between the partitioning of

[1]The (theoretical) attempt has been made to use the same BSP tree for two or more frames which show little difference, but the amount of computation necessary would outgrow the amount necessary to create and store a new tree.

[2]Initially the complete space is chosen as region R. Therefore, its partitionings are open halfspaces.

space and the corresponding BSP tree. In (a), we see a recursive partitioning of the plane. Note how partitioning first by u produces two new subspaces whose subsequent partitionings proceed independently of each other. In (b), we see that in the corresponding BSP tree each line is associated with an internal node of the tree. The right subtree of each internal node represents the positive subregion of the plane with respect to the line. The left subtree represents the negative side. The resulting partitioning produces a set of unpartitioned subspaces that correspond to leaves of the tree (labeled with capital letters).

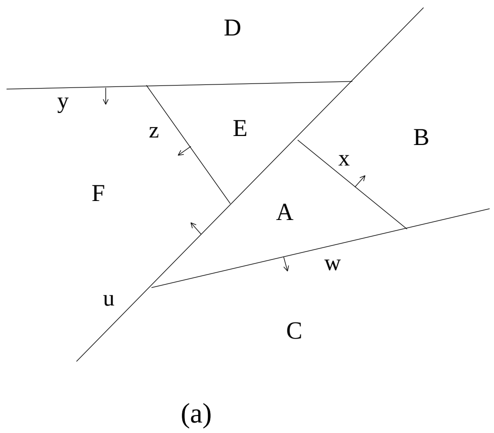

Figure 4.1: (a) 2D partitioning and (b) its BSP tree

One of the secrets of the efficiency of BSP trees is to choose suitable hyperplanes. A bad choice will result in excessive splitting of faces, thus creating a larger tree than necessary. Better choices will cause less splitting, and posssibly none. It is hard to find the *best* tree, nevertheless, since there is an infinite number of hyperplanes to choose from[3]. But it is possible to approximate this goal by trying a certain number of planes to see how many splittings they

[3]Determining an optimal set and order of hyperplanes out of a finite set is known to be NP-complete [N81]

induce, and then pick the one that requires the least. To reduce the number of planes to be tried, it is common practice to choose the set defined by the object's faces[4]. This leads to the following algorithm.

The BSP tree's root is an arbitrary polygon from the set of faces. It is used to partition the environment into two half-spaces. The positive half-space contains all polygons in front of the root polygon, relative to its surface normal; the negative half-space contains all of the remaining polygons located in the back of the root polygon. Any polygon lying on both sides of the root polygon's plane is split by the plane. Its front and back pieces are assigned to their appropriate half-space. One polygon each from the root polygon's front and back half-space become its front and back children, and each child is recursively used to divide the remaining polygons in its half-space in the same fashion. The algorithm terminates when each node contains only a single polygon.

Using this approach, convex objects never need partitioning, because none of the planes defined by the faces intersect. In this case, the smallest possible tree results.

Example Figures 4.2 and 4.3 illustrate how ordering the faces in the wrong way can result in unnecessarily larger BSP trees. The order in figure 4.2 causes a splitting of line b which leads to a more complex BSP tree.

[4]All non-planar faces have to be decomposed into several planar polygons—typically triangles—so that they can define a hyperplane.

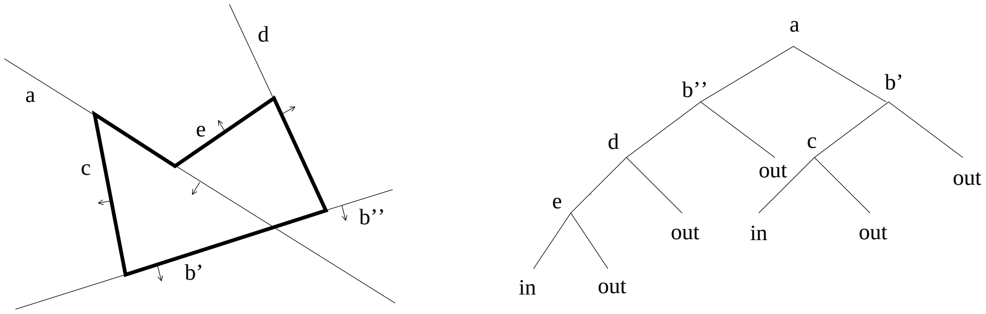

Figure 4.2: Bad choice of hyperplanes

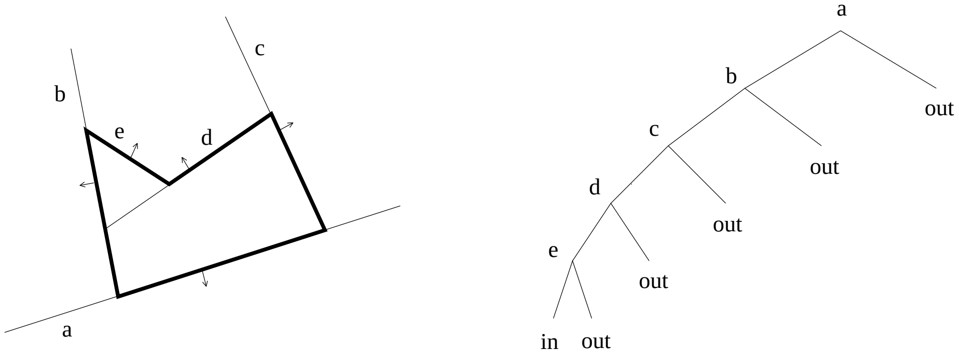

Figure 4.3: Good choice of hyperplanes

4.1.2 How to use a BSP tree

Each region of the tree is associated with a unique order in which clusters can obscure one another if the viewpoint is located in the region. Thus, determining the region in which the eyepoint lies involves descending the tree from the root and choosing the left or right child of an internal node by comparing the viewpoint with the plane at that node. Given a plane H

$$H : Ax + By + Cz + D = 0 \tag{4.1}$$

and its normal

$$\vec{n}_H = (A, B, C) \tag{4.2}$$

the following function determines the (oriented) distance of a point from this plane:

$$f_H(x, y, z) = Ax + By + Cz + D \tag{4.3}$$

Therefore, for an arbitrary point $P(x_0, y_0, z_0)$, we know:

$$\begin{array}{lcl} f(x_0, y_0, z_0) > 0 & \rightarrow & P \ above \ H \\ f(x_0, y_0, z_0) < 0 & \rightarrow & P \ below \ H \\ f(x_0, y_0, z_0) = 0 & \rightarrow & P \ coplanar \ with \ H \end{array} \tag{4.4}$$

Now, one can make use of the unique ordering of the tree: By traversing it such that the polygons that have the greatest distance from the viewpoint are visited first, the backfaces of the object will be drawn first, and the front faces last. This is easily accomplished by recursively traversing the tree in this fashion: Starting at the root of the BSP tree, the above formulas are used at each internal node to determine whether the viewpoint is in the front or rear subspace of the node's partitioning plane—represented by the left or right child. If the front subspace is the one that contains the viewpoint, then the branch for the rear subtree is traversed first, since it might include polygons which are obscured by the other branch. Once all polygons in this subtree have been visited, the node's polygon is displayed, and then the second branch—representing the front subspace—is traversed. However, if the rear subspace contains the viewpoint, the branches are traversed in the opposite order.

Finally, after all nodes in the BSP tree have been visited, the object has been displayed so that all hidden faces are covered up by the visible faces. This procedure can be repeated for *any* position of the viewer. There is no need to rebuild the BSP tree. Drawing the scene as seen from a different viewpoint simply involves traversing the tree again.

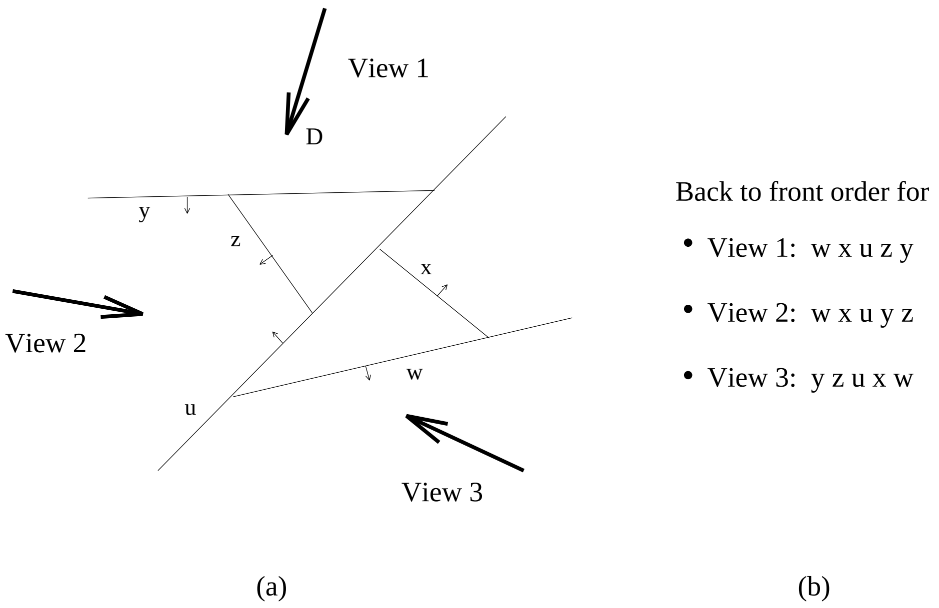

Figure 4.4: Tree traversal

Example Figure 4.4(b) illustrates the output order using the above algorithm for several different viewing positions of the scene in figure 4.4(a). Recall the corresponding BSP tree from figure 4.1(b).

4.2 Color Shading

The user of the program *shade* can specify the source and color of a light source. This allows to apply a shading algorithm when displaying the object. The current implementation of *shade* uses a flat shading model for this purpose. Given the normal $\vec{n}$ of the polygon that is to be displayed, the angle enclosed by $\vec{n}$ and the light source vector is used to determine the color of this specific polygon. For cases where the normal is close to perpendicular to the light vector, the darkest color is used; if the normal is collinear with the light vector, the brightest color is applied. Since the orientation of the polygon is not considered here, no difference is made between faces on the back and on the front of the object. This yields the illusion that there are in fact two light sources in opposite corners of the room. The advantage of this simplification is that faces on the opposite side of the light source are not completely hidden in the shadow. This results in a more interesting display, due to higher contrast between neighboring faces.

Chapter 5

Program Flow

5.1 Overview

See figure 5.1 for a visualization of the flow of data and the use of each program. Note that the generation of an animation can begin at any of three starting points. For example, if no object data exists, one would want to start with the construction tool *sculpt*.

5.2 Input File Formats

Jello has to be provided with information about the object whose behavior is to be simulated. For this reason the user supplies an ASCII file, referred to as *datafile*. Various parameters controlling the simulation can be adjusted here. However, only the following four sections are obligatory.

5.2.1 Basic Definitions

1. “Points:”

   Coordinates of all points which comprise the object. If “fixed” is stated after a point’s coordinates, this point will be constrained to these coordinates throughout the entire simulation.

2. “Connections:”

   Pairs of points which are to be connected by a spring, followed by a

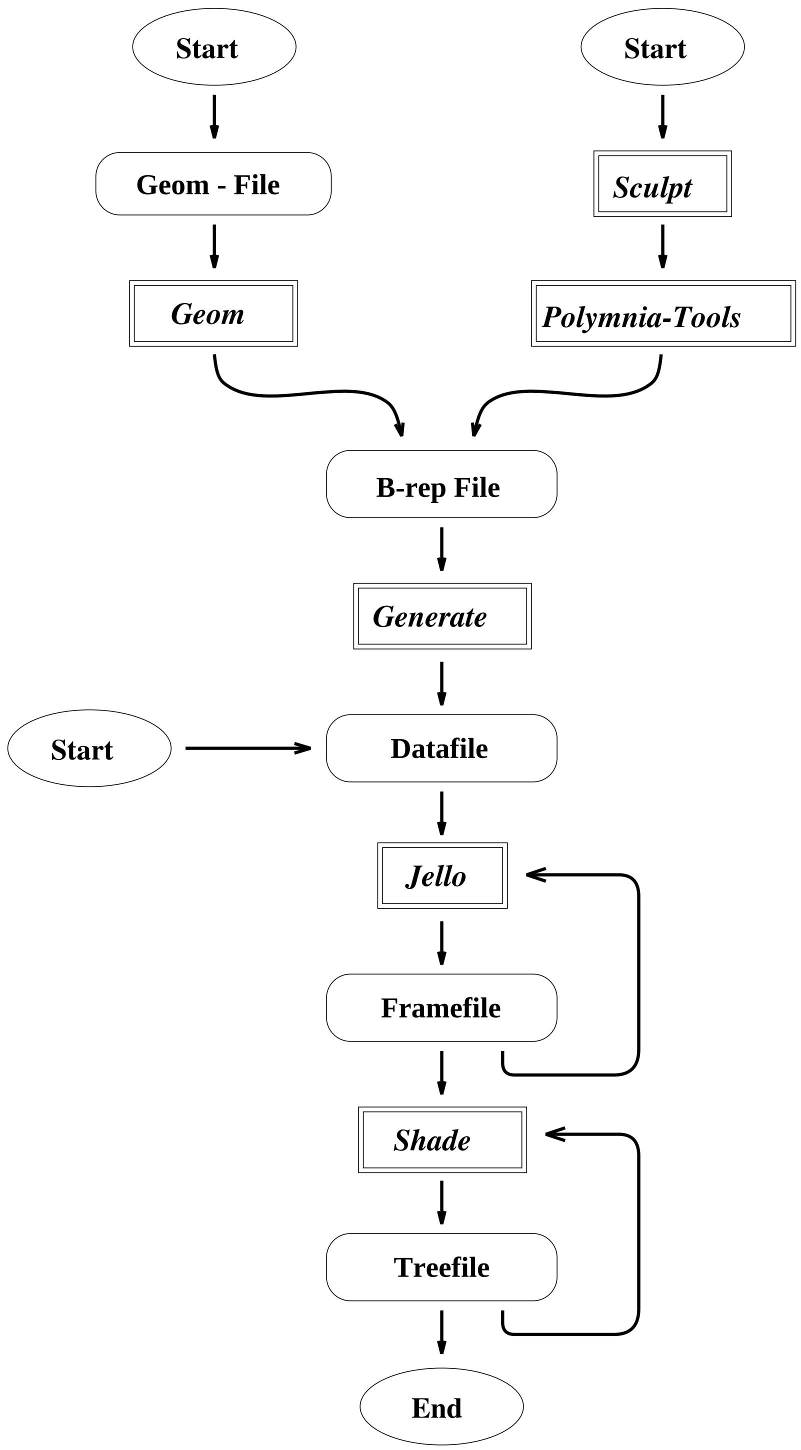

Figure 5.1: Overview of data flow

material constant for each spring. Points which are not connected to any other point will not be considered in the simulation.

3. "Hinges:"
   Triples (apex and two endpoints) of points which specify a hinge, followed by a material constant for each hinge.

4. "Faces:"
   Lists of points defining the object's faces. The points in each list have to be in order such that the normal of the face points outward (with respect to the object).

See figure 5.3 for a simple example[1]. It lists the contents of a minimal *datafile* to define the pyramid as seen in figure 5.2. Note that section "Hinges:" does not define the following two hinges:

```
4, 0, 2:   40.00000
4, 1, 3:   40.00000
```

To obtain more stability those hinges could be added to the model. But if a goal is to minimize the computational expense, it is often sufficient to include only hinges which are part of faces. Unfortunately, no general rule can be given, since this decision depends totally on the specific object.

Note that there is no restriction as to how many separate objects can be defined in the same datafile. A scene can consist of multiple objects, but since no intersections or collisions are detected, they will not interfere with each other.

[1]Note that the y-coordinate is the *up*-direction.

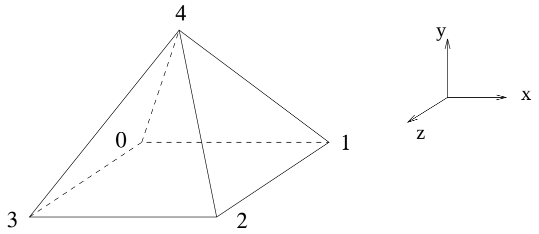

Figure 5.2: Pyramid as defined in figure 5.3

5.2.2 Further Options

Optionally, many more parameters can be included in the *datafile* to override the default values. Among them are the length (time) of the simulation (implied by the number of frames and the desired timestep between two consecutive frames) and various material constants concerning the object and the walls. To prevent unnecessary computations, the coordinates of a sample point are examined after every iteration. If its values are greater than `TooBig`, the simulation is interrupted with a warning that instabilities in the model might have caused it to "explode". In addition there are numerous controls which have effects on the playback of the animation: position and view direction of the viewer at the beginning, source of light and the color of all objects. See figure 5.4 for a complete listing of all options.

5.3 Geom

A popular format to describe complex 3D objects is the *geom* format. This format describes objects as a combination of polyhedra. Each polyhedron consists of a list of polygons which in turn are described by listing the indices of their vertices. A simple utility program called *geom* is supplied to convert *geom* objects into the format which can then be processed by *generate*.

```
Points:
  0:  0.0, 0.0, 0.0
  1:  5.0, 0.0, 0.0
  2:  5.0, 0.0, 5.0
  3:  0.0, 0.0, 5.0
  4:  2.5, 4.0, 2.5       // This is the tip of the pyramid.
 -1:  -1, -1, -1

Connections:
  0,  1:   50.00000       // Base of the pyramid
  1,  2:   50.00000
  2,  3:   50.00000
  3,  0:   50.00000

  0,  4:   50.00000       // Edges connecting the tip
  1,  4:   50.00000
  2,  4:   50.00000
  3,  4:   50.00000
 -1, -1:  -1

Hinges:
  0, 3, 1:   40.00000     // Hinges around the base
  1, 0, 2:   40.00000
  2, 1, 3:   40.00000
  3, 2, 0:   40.00000

  4, 0, 1:   40.00000     // Hinges where tip is apex
  4, 1, 2:   40.00000
  4, 2, 3:   40.00000
  4, 3, 0:   40.00000
 -1,-1,-1:  -1

Faces:
  0 1 2 3 -1              // Base
  4 1 0   -1              // Faces on the side
  4 2 1   -1
  4 3 2   -1
  4 0 3   -1
 -1
```

Figure 5.3: Datafile for a pyramid

```
NumFrames:       2500                    (how many iterations)
SaveEvery:       5                    (only save every 5th frame)
Timestep:        0.0025                                  (in sec)

Scaling:         1.0, 1.0, 1.0          (modify object before
Translation:     0.0, 0.0, 0.0           simulation starts    )
Rotation:        0.0, 0.0, 0.0

Speed:           0.0, 0.0, 0.0                   (initial speed)

Viscosity:       0.0                           (from 0.0 to 1.0)
Wind:            0.0, 0.0, 0.0        (direction and magnitude)

Ground_K:        10000.0                 (spring, damping and
GroundDamp:      5.0                      friction constants
GroundFriction:  10.0                     for the walls      )

SpringDamp:      0.1                      (global damp factor)
MeshMass:        1.0                      (object mass, in kg)

Gravity:         0.0, 9.80665, 0.0                  (in m/s^2)
TooBig:          1000000.0           (stop if values greater)

Viewer:          50.0,  10.0,  100.0
#either "Viewpoint" or "Viewdirection":
Viewpoint:        0.0,   0.0,    0.0
#Viewdirection: -50.0, -10.0, -100.0      (length irrelevant)

Color:           255, 255,   0              (Red, Green, Blue)
LightSource:    -1.0, -2.0, 3.0
Ambience:        0.1
Brightness:      0.9
FloorColor:      0, 0, 0            (zero-vector -> transparent)
XY_WallColor:    0, 0, 0              (Defaults to transparent)
YZ_WallColor:    0, 0, 0                                (ditto)
```

Figure 5.4: Default values of addititional parameters

5.4 Generate

As seen in section 5.2, the amount of data which is needed for the animation can grow quite large. It is cumbersome to manually create *jello* input data for the simulation of complex objects. *Generate* automates this process.

As a front end tool for constructing objects, we chose *sculpt*, an interactive 3D object design program [N90]. *Sculpt* is based on *polymnia* [NAT90], a C-library providing operations on geometric sets exploiting the advantages of the BSP tree format. *Generate* can read *sculpt*'s B-rep (boundary representation) format, which represents the boundary faces of objects as arbitrary polygons. Optionally, those polygons are refined into smaller polygons, mostly quadrilaterals and triangles. The granularity depends on a user-supplied parameter `precision`, which relates to the maximally allowed length of an edge. The smaller this length the finer will be the resulting mesh of polygons. This allows to achieve more accurate and realistic results during the simulation. Note however that finer meshes consist of more primitives and thus more time is required to compute a simulation.

An effort has been made to improve the performance of *jello* by converting objects into a mesh of points which represent the respective surfaces only. A previous version (see section 1.2) required that points be distributed evenly across the whole volume of the object. The inclusion of hinges in the improved model eliminate that need.

5.5 Jello

Jello is the main tool to simulate the physical behavior. It reads a *datafile* (see section 5.2), where the objects to be simulated are defined. It then performs at least one numerical integration for each requested movie frame, as described in section 3.

Upon completion of all iterations, the frames are stored in a binary *framefile*, along with all information which has been gathered so far. This information can later be used to continue the simulation without the need to redo all calculations and starting with the first frame.

Jello then displays the movie in a preview mode, in which the object is rendered as a wireframe model. The user interface, however, provides all the controls as defined below. Typically, a user will want to preview the movie to examine the simulation. Then he will go back and adjust some parameters in the *datafile*.

5.6 Shade

Once all parameters have been set so that the simulation shows the desired result, the second stage—producing the final movie—can be initiated. *Shade* reads the respective *framefile* and builds a BSP tree for every frame of the movie (as described in section 4.1). The resulting list of trees is stored in a binary *treefile*.

Shade then displays the final product. Information about light source and colors is read from the *datafile*. The fact that those parameters are not read from the *framefile* allows easy modification after the BSP trees have been computed. They are totally independent from the information which is stored in the BSP trees. The scene for a specific frame of the movie is rendered by traversing the corresponding BSP tree and applying a flat-shading model. The current position of the viewer and all light and color information are taken into account at this point in time.

BSP trees are built in such a fashion that they are completely independent from the position of the viewer. Therefore the user interface (see below) enables the viewer to easily change his position and viewing angle *while* viewing the movie. No expensive routines have to be executed to account for those changes before the scene is rendered again.

5.7 Displaying the Animation and User Interaction

The user interfaces for the preview mode in *jello* and for the fully rendered movie in *shade* are equivalent. While the series of movie frames is being displayed on the screen, the user can move freely through the 3D-space to examine the scene from any viewpoint. Intuitive navigation with a mouse enables the user to move with 6 degrees of freedom:

- move forward/backward, left/right and up/down
- change the view direction to the left/right and up/down
- twist clockwise/counter-clockwise

To permit convenient examination of the object, there is an additional, specially designed viewing mode. Instead of rotating the view direction around the current position of the viewer, the viewer rotates around the center of the object.

Full control over the playback is provided through keyboard input. Frames can be skipped to increase the apparent speed of the motion. The capability of freezing a frame and of single-stepping through the series of frames is implemented. The direction can be put in forward or reverse, and it is possible to skip to the first or last frame at any time.

5.8 Software Tools

All programs were developed and written in the object-oriented language C++. The GNU tools were used for efficient compilation and debugging. Suitable 3-dimensional objects were created with *Sculpt*[2]. The capabilities of *Polymnia* were employed to convert these objects from BSP tree format to a boundary representation format. *VOGL*[3] provided the means to create the user interface.

[2]see reference in section 5.4
[3]See [E91] and footnote in section 1.2.

Chapter 6

Results

Various examples are shown in this chapter in order to demonstrate some of the features and capabilities of the modeling system and rendering tool. Comparisons to the results of the term project (see section 1.2) are drawn in section 6.2.1.

6.1 1D and 2D Objects

The modeling system is not restricted to three-dimensional objects. Its design also allows for modeling polylines, surfaces or any combinations. Figure 6.1 outlines the motion of a relatively stiff rope. Figure 6.2 illustrates how hinges prevent a circle from collapsing. It was also attempted to model a flag waving in the wind, but the current wind model (see section 7) does not yield satisfactory results.

6.2 3D Objects

6.2.1 Cube

As mentioned in section 1.2, the term project's modeling system only allows simulation of cubes of various sizes and granularities. Figure 6.3 shows such a cube. The computational expense required for this animation corresponds approximately to the animation shown in figure 6.5 which was computed with the improved *PSH-model*. It is a refined version of the cube in figure 6.4. Note that the latter two reflect the motion in a much more realistic fashion than figure 6.3.

The initial conditions in all three examples were the same. The cube was initialized with velocity in negative y- and z-direction so that interference with both a wall and the floor could be illustrated.

Finally, figure 6.6 shows the same animation as figure 6.5, but from a different viewpoint, and with shaded faces and hidden lines removed.

6.2.2 Sphere

The simulation of a sphere in figures 6.7 and 6.8 makes use of the feature that an arbitrary orientation of the gravity force can be defined. So that the object is still falling in a vertical direction the scene was twisted by 45 degrees. What was designed to be the floor is now a slanted surface, as is the former vertical wall on the left side of the scene. Figure 6.9 shows more detail of the sphere's behavior when hitting a surface.

6.2.3 Swing

The modeling is not limited to purely geometric objects. As illustrated in figure 6.10, the flexibility of the *PSH-model* allows the simulation of a variety of scenes.

Figure 6.1: Bouncing rope

Figure 6.2: Circle

Figure 6.3: Term project cube

Figure 6.4: Cube

Figure 6.5: Refined cube

Figure 6.6: Shaded cube

Figure 6.7: Sphere

Figure 6.8: Sphere (continued)

Figure 6.9: Detailed view of sphere

Figure 6.10: Swing

Chapter 7

Conclusion and Future Work

This thesis presented a modeling system to simulate, view and examine the physical behavior of 3-dimensional objects. It consists of *jello*, a modeler based on physical constraints; *shade*, a rendering tool for the animation sequence based on BSP trees; and various utilities to create and refine objects. Unlike other authors conducting recent research in this field, we chose a point - spring - hinge model for representation of the modeled object. An advantage of this approach is that the simulation yields very realistic results. There are no restrictions which have to be imposed on the deformations of the model. Its properties can easily be adjusted by varying material constants for the springs and hinges. A disadvantage however is the amount of computation necessary for the simulation of a sophisticated object. Although a strong effort has been made to improve the performance of the numerical integrator, the entire animation sequence has to be precomputed in order to show a smooth movie of the resulting motions. Another weakness is that physically based modeling systems often produce unexpected results. Simulations have to be run many times before the desired result is achieved. Minor alterations of a single parameter can dramatically affect the outcome of a simulation.

The second part of this work covers a priori construction of BSP trees, which allow us to perform fast visibility determination of 3D-objects. Along with a flat-shading color model, every frame of the animation can be rendered in real time. The scene can be inspected from any desired viewpoint, thus providing a tool for exact analysis of the motion and behavior of the modeled objects. An intuitive user interface supplies additional support to achieve this goal.

Physically based modeling is an area that touches many different problems in graphics, modeling, simulation, and rendering. Some of these problems, by themselves, are topics deserving special attention. Other problems have known solutions but implementing them can nevertheless be difficult. Thus, it is not surprising that various portions of this modeling system are incomplete. What follows is a brief description of some unsolved problems and extensions. They are beyond the scope of this thesis, but would be considered useful capabilities within this modeling system.

Real Time Simulation: Further improvement of the efficiency of the numerical integration may yield the benefit of simulations in real time. Since the computations performed to evaluate the net forces upon the single point masses are highly independent from each other, the use of multiple processors promises impressive results. It would also certainly be worthwhile to port this modeling system to a parallel architecture.

Plasticity: A common property of springs and hinges is plasticity. If the strain reaches a certain threshold, a permanent deformation takes places. A possible result can be tearing of connections which may lead to a breaking apart of the model. The implementation of this property would further improve the realism of the simulation.

Intersection Detection: Currently there is no restriction on the number of objects which can be modeled at the same time. However, since no collision detection routines are implemented yet there is no interference between them. Adding this feature to the modeling system promises convincing results.

Wind and Viscosity: The current realization of wind and movement in a viscious medium are rather crude. A more sophisticated approach involves the mapping of a vector field onto the region of interest. Permitting this field to change with time would provide the means to build a realistic and interesting testing environment. Influences of currents and fluids could be studied.

Varied Gravitation Force: As Thingvold pointed out with his work [TC90] different kinds of gravitation could prove useful. E.g. attraction toward a point, line or even plane along with efficient intersection detection could be applied to simulate a shrink wrapping effect.

Non-linear Constraints: Physically based modeling alone is not suitable to animate arbitrary sequences. External controls are required if it is desired to exert more influence on the object's path of motion. Contraints need to be provided, allowing model points to be moved accurately along specified trajectories. So far only a special case of external control has been realized. Point masses can be constrained to a fixed position throughout the entire simulation. Witkin among others has conducted much research in this area [WW90] [WFB87].

Improved Shading model: The applied flat-shading model does not produce convincing results. A more sophisticated shading algorithm, like Gouraud [G71] or Phong [P73] shading or could be substituted without great effort. Since VOGL is compatible with the SGI GL library, it would be a worthwhile task to port the modeling system to an SGI machine.

Combined Interface: Lastly, the various tools presented in this thesis should be combined into one system. A menu-driven interface could simplify the use and provide the user with helpful support when experimenting with modified sets of parameters.

BIBLIOGRAPHY

[B84] Barr, A., *Global and Local Deformations of Solid Primitives*, Computer Graphics, 18(3): 21-30, SIGGRAPH '84 Proceedings

[BW92] Baraff, D., Witkin, A., *Dynamic Simulation of non-penetrating Flexible Bodies*, Computer Graphics, 26:303-308, SIGGRAPH '92 Proceedings

[C91] Campbell, A.T., *Modeling Global Diffuse Illumination for Image Synthesis*, PhD Dissertation, Department of Computer Sciences, University of Texas at Austin, 1991

[CF92] Chin, N., Feiner, S., *Fast object-precision Shadow Generation for areal Light Sources using BSP Trees*, Computer Graphics, 25:21-30, Symposium on Interactive 3D Graphics, 1992

[E91] Echidna, E.H., *VOGL - Very Ordinary Graphics Library, compatible with Silicon Graphics GL*, Department of Engineering Computer Resources, University of Melbourne, Australia, 1991

[FD82] Foley, J., Van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison Wesley Publishing Company, 1982

[FKN80] Fuchs, H., Kedem, Z.M., Naylor, B.F., *On Visible Surface Generation by A Priori Tree Structures*, Computer Graphics, 14(3):124-133, SIGGRAPH '80 Proceedings

[FAG83] Fuchs, H., Abram, G.D., Grant, E.D., *Near Real-Time Shaded Display of Rigid Objects*, Computer Graphics, 17:65-72, SIGGRAPH '83 Proceedings

[G71] Gouraud, H., *Computer Display of Curved Surfaces*, PhD Dissertation, University of Utah, June, 1971

[H87] Heckbert, P.S., *Ray Tracing JELL-O Brand Gelatin*, Computer Graphics, 21(4):73-75, SIGGRAPH '87 Proceedings

[HR60] Halliday, D., Resnick, R., *Physics for Students of Science and Engineering*, John Wiley and Sons, Inc., 1960

[H87] Haumann, D., *Modeling the Physical Behavior of Flexible Objects*, Siggraph Course Notes #17 (1987), 1-13

[HW87] Haumann, D., Wejchert, E., Arya, K., *et al.*, *An Application of Motion Design and Control For Physically-Based Animation*, Proceedings of Graphics Interface '91, 279-286

[J87] Lin, J.J., *Fast Ray Tracing on BSP-Trees*, Master's Thesis, 1987, Department of Computer Science, University of Central Florida

[N81] Naylor, B., *A priori Based Techniques for Determining Visibility Priority for 3-D scenes*, PhD Dissertation, University of Texas at Dallas, May, 1981

[N90] Naylor, B., *SCULPT, An Interactive Solid Modeling Tool*, Proceedings of Graphics Interface '90, 138-148

[NAT90] Naylor, B., Amanatides, J., Thibault, W., *Merging BSP Trees Yields Polyhedral Set Operations*, Computer Graphics, 24(4):115-124, SIGGRAPH '90 Proceedings

[NT86] Naylor, B., Thibault, W., *Application of BSP Trees to Ray-Tracing and CGS Evaluation*, Technical Report, Georgia Institute of Technology, School of Information and Computer Science, 1986

[P73] Phong, B.T., *Illumination for Computer Generated Images*, PhD Dissertation, University of Utah, 1973

[TPBF87] Terzopoulos, D., Platt, J., Barr, A., Fleischer, K., *Elastically Deformable Models*, Computer Graphics, 21(4):205-214, SIGGRAPH '87 Proceedings

[TC90] Thingvold, J.A., Cohen, E., *Physical Modeling with B-spline Surfaces for Interactive Design and Animation*, Computer Graphics, 24(4):129-137, 1990 Symposium on Interactive 3D Graphics

[WFB87] Witkin, A., Fleischer, K., Barr, A., *Energy Constraints On Parameterized Models*, Computer Graphics, 21(4):225-232, SIGGRAPH '87 Proceedings

[WK88] Witkin, A., Kass, M., *Spacetime Constraints*, Computer Graphics, 22:159-168, SIGGRAPH '88 Proceedings

[WW90] Witkin, A., Welch, W., *Fast Animation and control of Nonrigid Structures*, Computer Graphics, 24(4):243-252, SIGGRAPH '90 Proceedings

[ZT89] Zienkiewicz, O.C., Taylor, R.L., *The Finite Element Method*, McGraw Hill 1989

[ZS92] Zoller, B., Spiegl, A., *Physical Modeling of Deformable Structures*, Report on term project, Univ. of Texas at Austin, Fall 1992

VITA

Andreas Spiegl was born in Munich, Germany on May 23, 1968, the son of Hans and Inge Spiegl. While attending the *Dante Gymnasium* in Munich, he spent one year as a foreign exchange student at the *Memorial High School* in Eau Claire, Wisconsin in 1985-86. After graduating from the *Gymnasium* in 1988 (*Abitur*), he was commissioned for a fifteen month military service.

He entered the *Technische Universität München* in 1989 majoring in computer sciences. In 1990 he received a scholarship from the *Cusanuswerk*. After earning his intermediate diploma (*Vordiplom*) in 1991, he spent two more semesters of graduate study at the *TU München*. He then was awarded a special financial grant from the *Cusanuswerk* enabling him to transfer to the University of Texas at Austin in the fall of 1992. He enrolled in the graduate program of the department of Computer Sciences, where he concentrated on computer graphics.

Permanent address: Kirchweg 11a
81379 München
Germany

This thesis was typeset[1] with LaTeX by the author.

[1] LaTeX document preparation system was developed by Leslie Lamport as a special version of Donald Knuth's TeX program for computer typesetting. TeX is a trademark of the American Mathematical Society. The LaTeX macro package for The University of Texas at Austin thesis format was written by Khe-Sing The.