RICE UNIVERSITY

Combining Particles and Waves For Fluid Animation

by

Mark Hall

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Doctor of Philosophy

Approved, Thesis Committee:

Joe Warren, Director
Assistant Professor of Computer Science

Keith Cooper
Associate Professor of Computer Science

Ed Akin
Professor of Mechanical Engineering

Houston, Texas

April, 1992

Combining Particles and Waves For Fluid Animation

Mark Hall

Abstract

Modeling fluid motion is a problem largely unsolved by traditional modeling techniques. Animation of fluid motion has been possible only in special cases, falling into one of two general categories. Upper surface representations model wave phenomena for fluid in placid situations, such as calm ocean waves. Particle systems define the chaotic motion of fluid in highly volatile states, like waterfalls. Each technique can mimic physical motion of liquid only in limited situations.

We propose a system unifying previous techniques. We describe a system containing fluid of both categories. Two representations for fluid, corresponding to previous methods, allow modeling a wide range of situations. Automatic transitions between representations allow using the most appropriate technique for a given physical situation. Fluid is represented by two types of primitives: drops and pools. The drops constitute a particle system describing small, independent components. Pools model large bodies of fluid in more placid situations.

The notion of support differentiates the situations that each representation models best. In general terms, fluid is supported when there is a solid underneath the fluid. Previous techniques generally either assume that support is omnipresent or that support is nonexistent. In our system, support determines which representation should be used and when transitions between the two should occur. Supported drops flatten and become pools. Unsupported pools spawn drops. Combining the two techniques into a single system allows mimicking fluid in a broader range of physical situations than previous methods.

The resulting system models fluid motion based on physical properties of the environment. Gravity causes fluid to fall or flow downward. Solids restrict fluid motion, changing the course of flowing fluid and defining the shape of contained fluid.

Acknowledgments

I would like to thank my committee, Joe Warren, Keith Cooper, and Ed Akin, for their help in this research.

I am especially grateful to my wife, Mary, and my parents for their support during my graduate studies.

Contents

Illustrations

Chapter 1

Introduction

Much current research in computer graphics emphasizes the study of natural phenomena. Topics in display and design of machine-like parts have been studied for years. Only recently have the more complex and less straightforward modeling problems become popular. Models of waves, plants, muscles, and natural motion are gaining increasing attention. They have in common a less structured composition within a basic model. The complexity of each model, combined with the differences between instances make human design unwieldy at best. One oak tree looks much like another, with so much detail that a human designer would probably not attempt to model a forest of different trees. Likewise each wave is similar but different from the others. The motion is too complex for a human designer to explicitly control, and simplistic models do not have the desired visual effect. Fluid motion fits into this category of natural phenomena.

Fluids have been sparsely studied in computer graphics. Research has concentrated on the design, representation, and rendering of solids. The greater difficulty in modeling fluids arises because fluids, unlike most solids, are not rigid. Fluids deform continuously whenever unequal forces act on the fluid. Fluids do not have an easily specified shape, because the shape depends on the surroundings. Fluids in general are harder to represent than their solid counterparts.

Further, determining fluid motion is more complicated than predicting the motion of a solid object. Fluids under the influence of gravity flow downhill. The task is sufficiently difficult that an animator cannot hope to specify a fluid's position by hand for any reasonably sized body. Fluid motion needs to be generated algorithmically. For the motion to be recognizable, much less believable, some sort of physical simulation must be used to guide the animation. Animators have searched for a computationally affordable means of including fluids in their animation sequences.

There have been two major thrusts in solving the problem, surface techniques and particle systems. The two categories are differentiated by the environments they model well. Surface techniques are designed for situations where fluid is primarily contained by solids. There is typically little fluid movement. Particle systems model fluid where solids are primarily obstacles for chaotic, often rapidly moving fluid. In these situations solids, when present at all, are barriers with which fluids collide. Contact between fluid and solids is usually transitory. We will present a system that unifies many of the strengths of previous work.

Previous surface techniques involve situations where the fluid can be represented by its upper surface only. Generally these techniques produce ripples or waves. The surface is modeled as a height field [Max81, MWM87, Sch80], a beta-spline surface [TB87], or simply a bump map textured on a plane [Per85]. Varying the height field or texture map over time allows animating the waves. Using the underlying terrain to modify the time-varying nature of the wave motion creates realistic wave motion near shorelines[Pea86]. By using a parametric surface representation rather than a height field, waves can be made to break as they approach the shore[FR86]. Surface techniques generally do not allow for movement of fluid from one location to another. They work best in representing large bodies of stationary fluid where the only motion can be represented by minor changes in the surface.

Particle systems[Ree83] are used to model chaotic, unconstrained fluid, such as fluid in a waterfall or other nearly free-falling situations. A particle is a small entity. Clouds or swarms of particles combine to collectively represent a larger object. Particles used to represent fluid are governed by gravity and treated as projectiles. Peachey [Pea86] and Fournier and Reeves [FR86] use particles to model the spray formed by breaking waves. The film *Red's Dream* [Pix87] contains particle systems for raindrops and splashes. Miller and Pearce [Mil87, MP89] use particles to simulate viscous fluid. Sims produced realistic fluid sequences in his film *Particle Dreams*[Sim88] by animating approximately 60,000 particles. Particle systems work adequately for fluid with high kinetic energy, but do not model fluid at rest very well.

A third technique, Computational Fluid Dynamics, is not generally used by animators because the volume of calculations required is prohibitive for most animators. A small 100x100x100 grid contains a million elements. Any non-trivial calculation on that grid quickly becomes expensive. Moreover, the goal of most CFD simulations is different than the goal of an animator. CFD is used to predict or explain a physical situation, for instance to design a more aerodynamic car body. Animators want a simulation of fluid realistic enough to foster a willing suspension of disbelief. A simple form of CFD is used by Kass and Miller[KM90a]. Their approach models the fluid as a height field over a two dimensional grid. It assumes that there is always a solid support under the fluid. By limiting the dimension of their simulation, and making some simplifying stipulations about the situations they model, they are able to do remarkable simulations of flowing fluid. Their technique does not work in representing more complicated physical situations, or free-falling fluid.

Our method combines two representations in a single system to simulate fluid motion in a wider range of physical situations. There are two forms used to represent fluid: pools for larger, more contained bodies of fluid, and drops for fluid moving under fewer constraints. The two representations allow modeling fluid in a wide range of kinetic energies, a feature not present in other modeling systems. Transitions between the representations are automatic, allowing the appropriate representation to be used as situations change.

We use a particle system of drops to model fluid in an active state. Drops move as projectiles, affected by gravity and bouncing off solids. Normally, drop surfaces are calculated as a level set of a three dimensional field function. The function is the sum of fields centered at each particle. Varying the level set value controls the amount of enclosed volume. We maintain the correct drop volume, a feature not present in previous methods, by choosing the appropriate level set.

In our system, pools are data structures for modeling large, nearly stationary bodies of fluid. They are based on a voxel structure, with volume and movement calculated on a voxel by voxel basis. Solids limit the movement of fluid from voxel to voxel. Fluid surfaces are calculated for voxels on the periphery of the pool. Standard surface effects are used to enhance the appearance of the pool.

Physical aspects of the environment control the fluid motion. Gravity affects the motion of both fluid representations, and contributes to the change from one representation to another. Solids comprise the containers for enclosed fluid and obstacles for moving fluid. The presence of solids affects the shape of nearby fluid. The combination of gravity and solids causes fluid to act in expected ways, flowing downhill, filling containers, pouring out of spouts, and so on.

The concept of "support" determines transitions between the two fluid representations. A portion of fluid is supported when there is a solid preventing motion in the direction of gravity. A drop that is resting on a surface will flatten and transform into a pool. Unsupported portions of pools convert into drops. Automated representation switching allows the system to use the representation most suited for each physical situation.

Control parameters are provided to aid the animator. A key drawback to physically-based animation is that “physics by itself does what it wants to do. It doesn't want to do what you want to do.”[Bar89] Our system has variables to enable the user to “encourage” the fluid into the desired action. Sources of fluid allow addition of fluid in the system. Fluid may also be removed under user control.

A related topic that is not covered in this work concerns displaying fluid. Recent work [Wat90, NKON90] has produced realistic rendering of pooled fluids. Our work has concentrated on modeling fluid motion and surfaces. Display techniques from others' work will be applicable since they use polygonal fluid surfaces.

Chapter 2

Preliminaries

This chapter introduces some terms and concepts used throughout this document.

2.1 Voxel Structures

Our implementation divides space into evenly sized rectangular volume elements called *voxels*. Voxels are usually cubes, although the length, width, and height could be different sizes. Voxel edges are aligned with the coordinate axes.

The simulation region is a rectangular solid as suggested in Figure 2.1B. The number of voxels along each dimension can vary but each voxel has the same size, shape, and orientation. The individual voxels exactly partition the region.

Data is kept within an array where each element of the array corresponds to a single voxel. Information about a voxel includes details about the types and amounts of fluid within the voxel and the solids that intersect the volume. Voxels that share a common face, edge, or vertex do not always contain the same data about their shared regions.

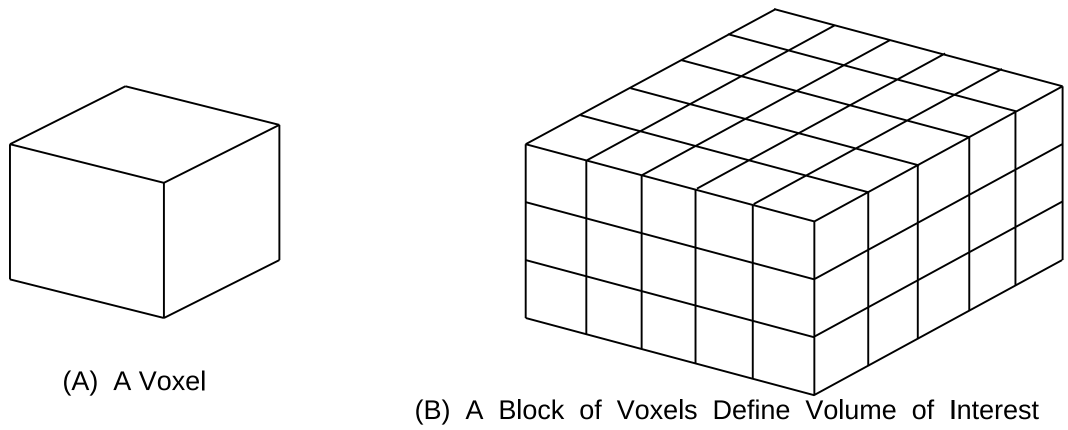


Figure 2.1 Examples of Voxel Structures

Many operations use the voxel structure to break global calculations into local ones that are then aggregated. For instance, fluid volume is calculated on a per-voxel basis, and the volumes summed to determine the overall fluid volume.

Both global and local structures use voxels. The pool structure, for example, stores fluid location information in a voxel structure. All voxels in the system, global and local, have the same size, shape, orientation, and are aligned in space. That is, the region within a local voxel corresponds exactly to the same region in some global voxel. The word *voxel* will be used to denote the global structure elements. Phrases like "pool voxel", or "pvox" for short, will be used for the local voxel information.

An animator must make a tradeoff when specifying the voxel size. Much of the information stored and calculated by the simulation concerns knowledge about the voxel corners. The voxel corners constitute sampling points in three dimensions. Details smaller than twice the distance between samples are not guaranteed to be detected at the sample points. Small voxel sizes allow more closely monitored conditions and more accurate calculations. Figure 2.2 shows how details can be missed if the voxel size is too large. In 2.2A, the distance between sample points, the size of a voxel, is too large for the hole in the solid to be noticed. A solid is detected at every sample point, so there is no way to detect the hole. In 2.2B the voxel size is small enough that the hole is detected. The voxel size must be chosen to ensure that important features are not overlooked. On the other hand, smaller voxels are undesirable because more of them are required to enclose the same region. The computational complexity rises when the number of voxels increases.

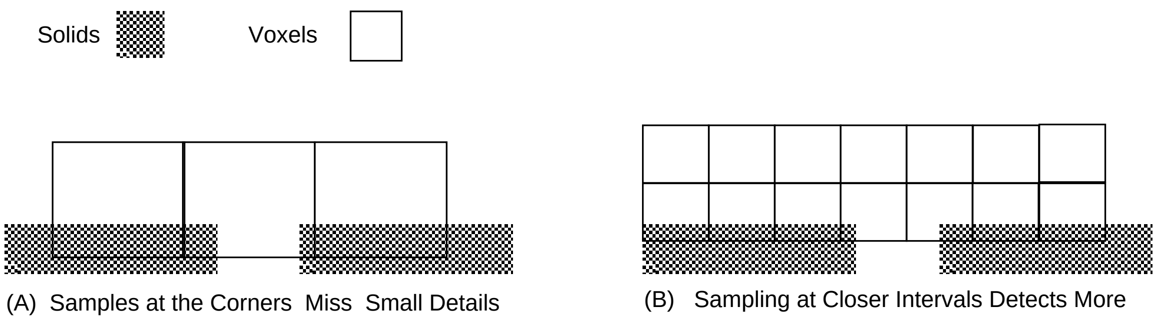


Figure 2.2 Voxel Size Affects Level of Detail

2.2 Representation of Solids

In our system, solids are bounded by convex planar polygons. These polygons are specified as an ordered list of coplanar vertices. Complicated objects are specified as multiple polygons, normally by decomposing them into triangles.

Polygons are further clipped to each voxel they cross. Each voxel contains a list of the polygons that cross it and a copy of the polygonal portion within the voxel. Figure 2.3A shows a polygon and a voxel it crosses. Figure 2.3B shows the same polygon clipped to the voxel. The voxel also records the positions where the polygon crosses the voxel edges. Each voxel face has a list of the lines where the polygon crosses that face.

Polygons are by definition infinitely thin, which can be a problem. Solids have volume, so they should be defined by a closed set of polygons. Many computer-graphics objects are specified as polygons, without any true solid volume. We allow solids defined as infinitely thin polygonal shells, but the results may be poor. Since our technique involves sampling the region of interest, inifinitely thin solid 'volumes' are not always treated correctly. This is an inherent problem with sampling methods. It is best to define objects by defining surfaces around a volume.

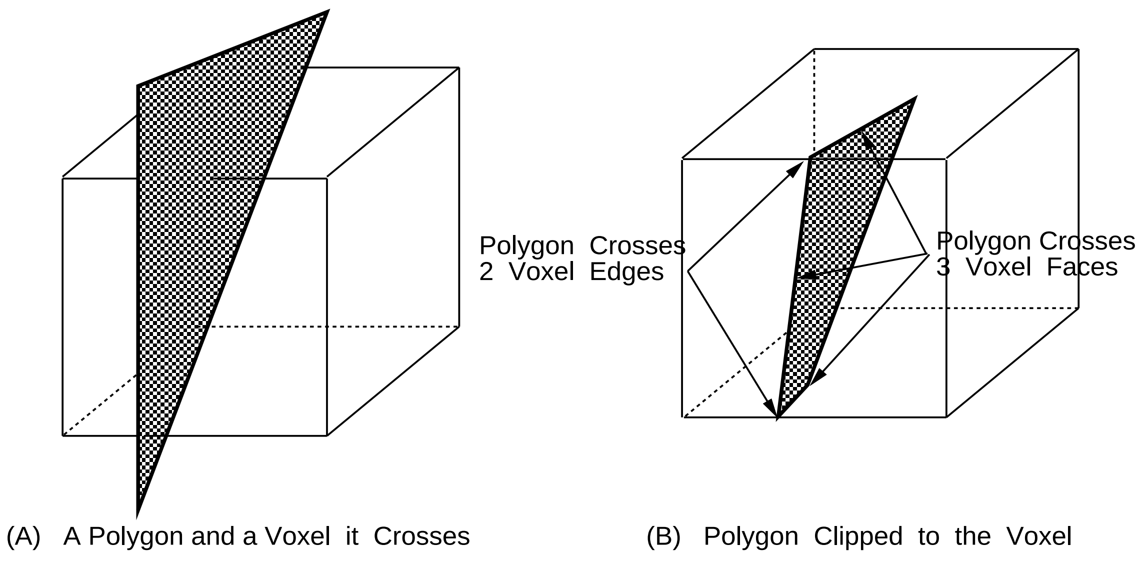


Figure 2.3 Solids are Clipped to Voxels

Chapter 3

Drops: Fluid in an Active State

A drop is a small amount of fluid moving as a ballistic object. It is an atomic, undividable fluid portion. Since drops are small, there is no flow within the drop and it can be treated as a single nearly-solid entity. A drop's motion is determined by gravity, other external forces such as wind, and the presence of solids. Many drops may exist in a small region, and their combined motions define the complete fluid motion.

Reeves introduced *particle systems* to the graphics community [Ree83] as a technique for modeling fuzzy objects. Particle systems are clouds or swarms of primitive objects, the particles. Particles can be thought of as objects so small that each occupies a single point in space. Each particle has associated data. The exact data structure depends on the application but contains information like position, velocity, size, color, and age. In general, particles can be created, spawn other particles or particle systems, change characteristics, and die. Usually a particle moves and changes independently of others.

The combined characteristics of the individual particles determine the appearance and motion of the fuzzy object. Consider a swarm of bees in flight. The swarm has a nebulous, changing shape. Each bee, though, has its own appearance and flight path. The shape and path of the swarm is determined by the combination of all the bees. Simple rules can determine each bee's actions yet define a complicated swarm.

Stochastic processes, not the user, specify the characteristics of individual particles. A programmer specifies valid ranges for each characteristic, usually as an average value and an acceptable variance. Specific values for a particle are determined by the computer. Once created, motion and appearance of a particle are calculated automatically. Differences in the particles can create complicated systems with little human effort. The programmer controls the general behavior but not the details.

Chaotic fluid motion is modeled well by particle systems. Reeves [Pix87], Miller and Pearce[Mil87], and Sims[Sim88] have all used particle systems to depict fluid falling under the influence of gravity. The situations they modeled are characterized by fluid in a state of flux, where there is constant change. We use a particle system to model similar situations.

The actual use of the particles system varies. Reeves and Sims draw the particles. They draw a motion-blurred anti-aliased line for the path of the particle during a single frame. Thus part of the particle data structure for these two films is a color and transparency value. Each particle can be drawn independently from all the others because the particles are drawn as points of light on a black background.

Miller and Pearce define each particle to be a sphere and draw each of them. The shading of each sphere is dependent on a function determined by all the particles. Since the particle system as a whole is not used to determine the fluid shape, finding the fluid surface is fast. Using the particle system as a whole for the shading calculation gives a smoother look to the fluid, but at a higher computational cost.

Wyvill's Graphicsland project used a grid of point sources to define a bubbling fluid in the video *Soft II*[oC87]. The technique used in that video can be considered a particle system too. The particles have a limited lifetime, have motion characteristics initialized in a user-specified range, and a single composite object is defined by the combination of all the particles. The system varies from the others because the point sources could only move up and down, and gravitational forces were not used to control the motion. No physically-based simulation was part of the system. Another difference is that the particle themselves are not displayed. Instead, they are used to define a surface.

We also use a particle system to represent complicated fluid motion. The individual particles are a structure called a *drop*. A drop is modeled as a point in space, and some surrounding volume. Each drop has an associated structure containing information on that drop. A drop has a position, a mass, and a velocity. Gravity and nearby surfaces exert forces on the drop. The drop structure also records some history of the drop, such as the drop's age, and a length of time that a drop has been supported or stationary.

3.1 Drop Movement and Collision Detection

Drops move according to Newton's Law of Motion. The drop velocities change as a function of the drop masses and the forces acting on each drop. Their motion is that of point masses. Each drop has an associated velocity vector that is assumed to be constant for each time step or until a collision with a solid occurs. If a drop has velocity

$$\vec{V}_t = (X_t \; Y_t \; Z_t)$$

at the beginning of a time step of length Δt, and the forces on it cause an acceleration $\vec{A}$, then the drop will have velocity

$$\vec{V}_{t+1} = V_t + (\Delta t)\vec{A}$$

at the end of the time step if the drop does not collide with a solid. The acceleration modifies the velocity at the end of each time step, or when a collision forces a velocity change.

Collisions with solids are detected by traversing the linear drop path during a single time step. This technique is also used in many other places to check visibility between two points. Figure 3.1 depicts the process in two dimensions. A drop's initial velocity would cause it to move in a straight line during one timestep, as depicted by the arrow. An initial position and a direction vector define a ray. The drop path through the voxels is calculated using a method developed for ray-tracing [FTI86]. Starting at the voxel containing the drop at the beginning of a time step, the ray is tested to see if it intersects solids in the voxel. A sequence of voxels is determined by calculating which voxel side the ray intersects first, then moving along the ray to the appropriate neighbor voxel and repeating the process. In the example, the algorithm traverses voxels 1 through 4.

Implementors of previous systems have solved the collision problem in several ways. Miller and Pearce used a similar collision detection scheme for their "globules".

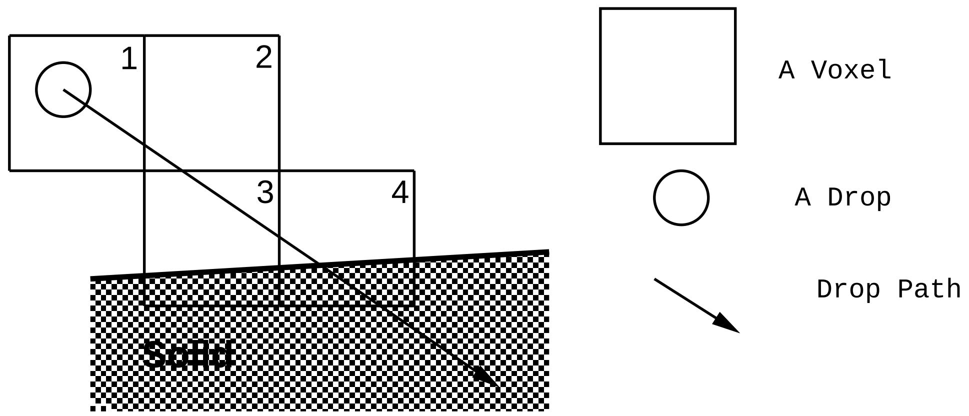


Figure 3.1 2D Example of Drop Motion

Each globule has an unchanging radius. In their system, collisions are sometimes calculated for the center point of the globule, which can allow fluid to penetrate solids. In these situations, containers must have a thickness equal to the largest globule radius or fluid will show through the side of the container. Another solution, displayed in the animation short *Natural Phenomena*, is to calculate an offset surface for the solids. This offset surface is used in the collision calculations instead of the original solids. This works well for simple scenes, but creating an offset surface for complicated polygonal objects is computationally expensive. For globules of varied sizes, multiple offset surfaces would have to be created for each solid.

Sims' collision detection was simplified by using two types of solids: infinite planes and spheres. The exact method of collision detection is not specified [Sim90]. When a collision is detected in his system, a drop is moved to the point of impact, and its velocity is modified. The drop is not moved again during the frame where a bounce occurs.

Reeves sidestepped the collision detection problem for raindrops in *Red's Dream*. The drops fall to the ground and are deleted when they leave the air, perhaps just after falling below ground level. In that film, drops did not interact with anything else in the scene.

The Graphicsland project makes no mention of collision detection. Their field sources are moved by the animators or under program control without concern for collisions. They either ignore collisions or ensure that collisions do not occur.

In each voxel, the ray is tested against any solids in that voxel. We use an intersection calculation method by Haines [Gla89] that is similar to polygon filling. Each planar polygon is in a plane defined by

$$A * x + B * y + C * z + D = 0$$

where

$$A^2 + B^2 + C^2 = 1.0$$

A ray through points $Start$ and End is defined as points satisfying

$$ray = Start + t * Dir$$

where

$$Start \equiv (X_0 \; Y_0 \; Z_0) \quad and$$

$$Dir \equiv (X_d \; Y_d \; Z_d) = End - Start$$

and t is a positive number. We use the convention that values of t between 0 and 1 are on the path a drop follows during a single time step.

The intersection of the ray and the plane is calculated by substituting the expanded form of the ray equation into the plane equation:

$$A * (X_0 + X_d * t) + B * (Y_0 + Y_d * t) + C * (Z_0 + Z_d * t) + D = 0$$

and solving for t:

$$t = \frac{-(A * X_0 + B * Y_0 + C * Z_0 + D}{A * X_d + B * Y_d + C * Z_d}$$

If the denominator is zero, the ray is parallel to the plane and there is no intersection. If the calculated intersection value of t is between zero and one, the ray intersects the plane of the polygon in the time interval. The point of collision is calculated by substituting t back into the ray equation.

Now we have an intersection point, and a polygon, all in 3-space. To determine whether the point is inside the polygon, we turn this into a two dimensional problem. We pick a plane defined by two coordinate axes and project the polygon and the intersection point onto that plane. The plane chosen is the one nearest the plane of the polygon. We find the largest magnitude A, B, or C component in the polygon plane equation and discard the corresponding x, y, or z component of each polygon corner vertex and the intersection point. This creates points in a two-dimensional UV coordinate system. This two dimensional problem is less complicated than the original three dimensional one.

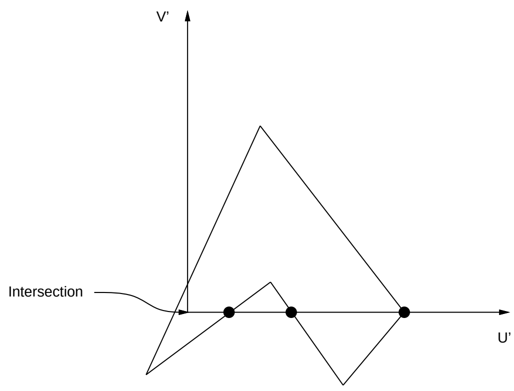


Figure 3.2 Determination of Whether a Point is in a Polygon

We now have a problem familiar to graphics programmers when filling a polygon in two dimensions. The Jordan Curve Theorem says that a point is in a polygon if an infinite ray from that point crosses the polygon edges an odd number of times. We define a new coordinate system (U', V') where the intersection point is at the origin. We test each polygon edge for intersection with the positive U' axis as show in Figure 3.2. If the number of intersections is odd, the point is in the polygon, and therefore the drop collides with the solid. The special cases of what to do when one or both vertices on a side lie on the ray are handled by defining them away. Points on the line are considered to be above the line.

When a collision is detected, as in voxel 4 of Figure 3.1, the drop will change direction. The time elapsed is calculated from the distance traveled and the drop speed. The drop velocity is modified by acceleration during the elapsed time to determine the drop velocity just before collision. The drop velocity is then changed by the collision, as described below. The drop motion calculation is resumed with the new velocity for the remaining time.

Figure 3.3 shows that drops rebound off solids inelastically. Drops can lose speed on impact, and the rebound is generally not at the perfect reflecting angle. For a drop with velocity $\vec{I}$ hitting a solid with unit normal $\vec{n}$, $\vec{I}$ can be broken into

$$\vec{I}_n = \left(\vec{I} \cdot \vec{n}\right)\vec{n}$$

and

$$\vec{I}_p = \vec{I} - \vec{I}_n$$

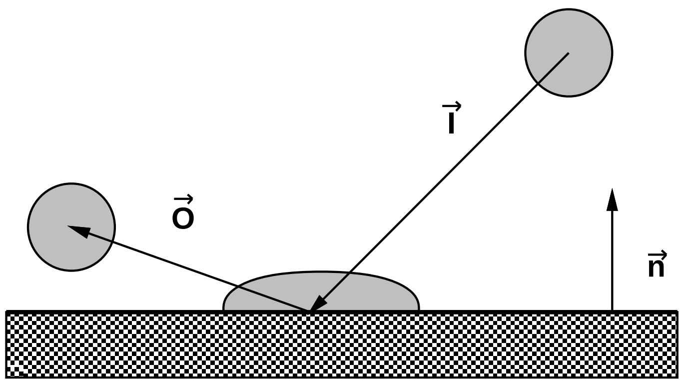


Figure 3.3 Drops bounce Off Solids Inelastically

where $\vec{I}_n$ is the component if $\vec{I}$ in the normal direction and $\vec{I}_p$ is the component tangent to the solid. Another vector in the plane of the solid is perpendicular to both $\vec{I}_n$ and $\vec{I}_P$ and is defined by

$$\vec{I}_{perp} = \frac{\vec{I}_n \times \vec{I}_p}{\left\| \vec{I}_n \times \vec{I}_p \right\|}$$

The amount of variation in the bounce direction in the plane of the solid is characterized by the *SPLAY*. The variation for a collision is a random variable, denoted θ, in the range -SPLAY to +SPLAY. The rebound direction component in the plane of the solid is

$$\vec{I}_{plane} = \|\vec{I}_p\| \; sin(\theta) \; \vec{I}_{perp} + cos(\theta) \; \vec{I}_p$$

The direction of rebound $\vec{O}_{dir}$ is calculated as

$$\vec{O}_{dir} = \frac{\vec{I}_{plane} + (1 - BF)\vec{I}_n}{\left\| \vec{I}_{plane} + (1 - BF)\vec{I}_n \right\|}$$

and the velocity after collision, $\vec{O}$, is given by

$$\vec{O} = KEF \;\; \|\vec{I}\|\vec{O}_{dir}$$

Kinetic Energy Factor (KEF) is a number between 0.0 and 1.0 that determines how much of the kinetic energy is preserved in the bounce. Setting KEF to 1.0 makes

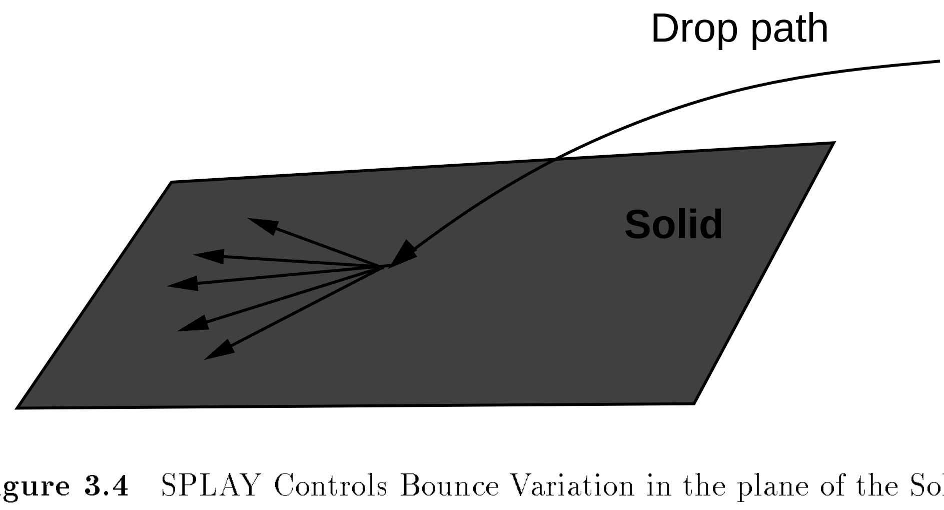


Figure 3.4 SPLAY Controls Bounce Variation in the plane of the Solid

for frictionless motion. Drops will never slow down. A KEF of 0.0 causes totally inelastic collisions. Drops that collide with solids immediately stop moving.

Bounce Factor (BF) is a number between 1.0 and 2.0 that determines how close the bounce is to a perfect reflection off the surface. A BF value of 1.0 causes drops to skim along the surface of solids they encounter. A BF of 2.0 causes drops to bounce elasticly, like rubber balls.

SPLAY is the possible amount of variation from a true bounce in the plane of the solid. Splay values are specified in degrees and generally have values from 0.0 to 90.0. SPLAY determines how much a stream of fluid will spread out when splashing against a solid.

Chapter 5 describes the usage of the parameters KEF, BF, SPLAY to control the liquid motion.

3.2 Displaying the Drop Volume

Drop surfaces are defined by level sets of field functions. Each drop is considered a source of a field that is strongest at the source, and weakens as the distance from the source is increased. A drop **d** in space has a spherical field function defined by

$$f_d(x,y,z) = \frac{mass_d}{distance^2} = \frac{mass_d}{(P_x - x)^2 + (P_y - y)^2 + (P_z - z)^2}$$

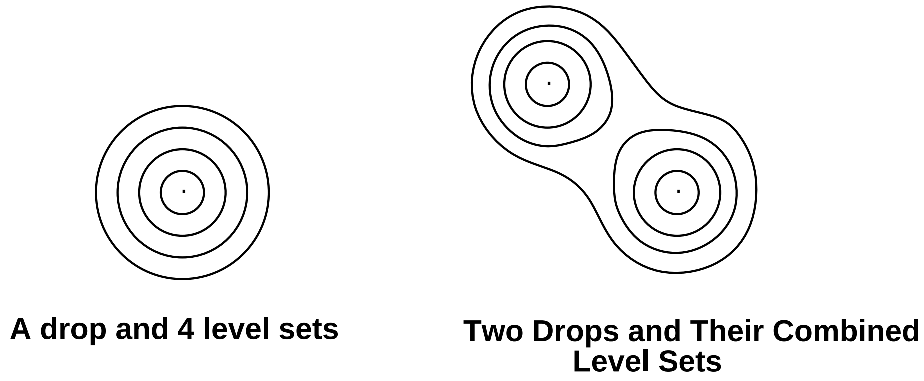


Figure 3.5 Level Sets of Drop Fields Define Various Volumes

where $P_x, P_y, P_z$are the coordinates of drop **d** and $mass_d$ is the mass of drop **d**. A surface is defined as points in space satisfying

$$f(x, y, z) = c$$

where **c** is the surface threshold, the level. The surface defined in this manner is often called a *level set*. Figure 3.5 shows that different volumes can be enclosed by varying the threshold. For a level set where several drops contribute significantly to the field value at a single point, their field functions add to create a single level set enclosing all the drops. The fluid shape is determined by the influences of nearby drops.

We choose a level set to make the drop volume correct. We guess a level set value and see what volume is enclosed. This process is repeated until the volume is within a user-specified tolerance of the desired volume or until a maximum number of guesses have been made.

The process of determining the enclosed volume proceeds on a voxel by voxel basis. For each voxel edge, we calculate what fraction of that edge would be submerged. If both edge endpoints have a value greater than the level, the entire edge is submerged. If only one endpoint has value above the level, we assume the function is linear over the voxel edge. We calculate the percentage of the edge that is submerged by simple division. The estimated fluid volume in each voxel is proportional to the percentage of "wet" edges for that voxel. The summed volume for each voxel gives a volume for the entire particle system.

The system may use other methods to display drops. The simplest way to display moving particles is to draw a line representing the path of each particle over time. Miller and Pearce use a sphere for each drop, which gives fluid a real volume. Sims draws motion-blurred, possibly radius-changing, spheres. We use volumes enclosed

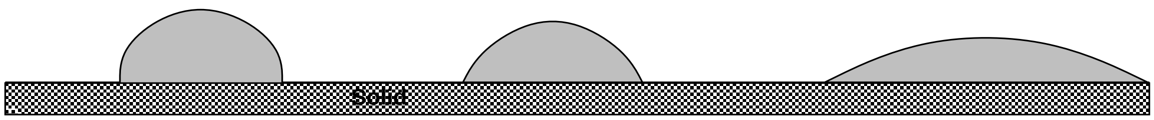


Figure 3.6 Drop Shape Changes Over Time From Changing Field Function

by level sets because it gives a polygonal surface description and a natural volume measure. Both features fit well with the representation for pools. The other methods are less computationally expensive and might be suitable for use in some situations in our system.

Using field functions for modeling fluid surfaces is not a new idea. Blinn[Bli82] used fields to display covalent bonds for “blobby” molecules. The University of Calgary Graphicsland Project [WMW86b] uses field functions extensively to model a range of *soft* objects, including fluid. In their animation *Soft II*[oC87], a boiling fluid was modeled using spherical fields. Upson included tiling of field functions in his methods for displaying amorphous phenomena [Ups86]. Miller and Pearce [MP89] use field functions to find surface normals for displaying spherical drops.

In our simulation, a drop’s field function varies over time because of its environment. Specifically, a drop that is sitting on a surface will tend to flatten out over time as in Figure 3.6. The notion of “on a surface” is defined as follows. A radius **r** is calculated for a drop, by knowing the drop volume and assuming the drop is spherical. If a line segment of length **r** drawn from the drop position downwards intersects a solid, the drop is on the surface. Each drop contains a record of the latest length of time that it has been supported by a surface. The field function for a supported drop is

$$f_d(x, y, z) = \frac{mass_d}{distance^2} = \frac{mass_d}{(P_z - z)^2 + C * (P_x - x)^2 + (P_y - y)^2}$$

where **C** is a fraction that starts at 1.0 and decreases over time for supported drops. The time for **C** to reach a minimum value is the Time To Flatten (TTF) and can be specified for each drop. Over time, the drop’s field function is altered to maximize the component in the direction of the surface normal. Over time the field becomes more and more like a planar field, causing the fluid volume to flatten.

Flattening of drops adds realism missing in previous systems. Miller and Pearce’s spherical particles look realistic as a fountain or when pouring. Fluid piling up or rigidly standing on a floor looks less realistic and demonstrates the underlying data structure. Wyvill mentions no change in the field function. In our system flattening drops provide a more normal appearance and also allow easier transitions to the pool representation. Pools tend to have flat upper surfaces. Flattened drops look much like a pool, making transition visually simple.

In our system, solids act as barriers for field functions, shaping the fluid boundaries. The solids limit the contributions of drops to their environment. Figure 3.7

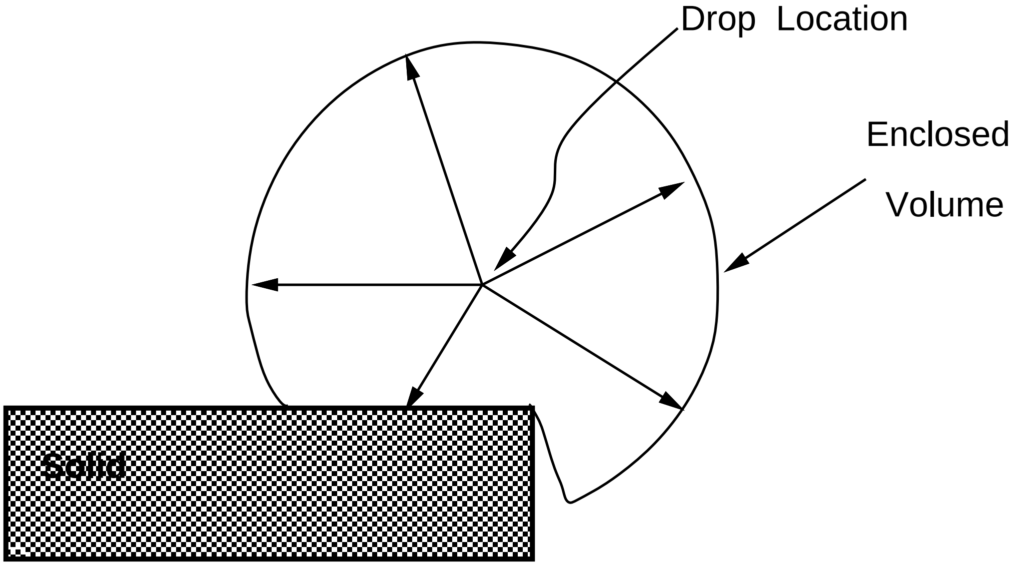


Figure 3.7 Solids Block Field Functions Along Line-of-Sight

shows a drop near a solid. The drop's field function is blocked, that is, has a value of zero, for those locations where the solid is in the path from the drop. When the drop is near the solid, its natural spherical shape would cause part of its volume to exist inside the solid. Since the solid blocks the drop's field, a surface threshold that encloses the proper volume will give the drop a flattened shape as shown.

The field function blocking calculation uses the same routine as detecting drop-solid collisions. A line segment from the drop location to an arbitrary point in space is tested for intersection with solids. Voxels are traversed from the drop location to the opposite endpoint and the line intersection calculation is performed for all solids in the voxel. When an intersection is detected, the field contribution of the drop is zero. If there is no intersection, the field is evaluated as described above.

The Graphicsland project modeled infinite solids, specifically the ground, as a surface of constant potential[WMW86a]. This works for infinite surfaces, but is not generally applicable for non-infinite solids. Some solids were defined as sources of a negative field function. The level set would be "repelled" from the solid, deforming the fluid. Such a solid with a hole in it causes moving objects to squeeze through the hole, as in the film *The Great Train Rubbery*. It is not clear how to keep fluid in containers using this technique. A strong drop function could overcome the negative function of a solid, which would allow fluid to exist outside its container.

3.3 Polygonalizing the Drop Surface

This section details the method for defining polygons to represent the drops' fluid surface. The technique is derived from Lorenson and Cline's work [LC87] with a few minor changes. Other similar methods [WMW86b, Blo88] could have been used.

The surface is evaluated over a three dimensional array of values that are considered samples of a continuous function $f(x, y, z)$. The array represents a rectangular region of space aligned with the x, y, and z coordinate axes in world space. In this system, the values are the summed field functions of all the drops. A surface is defined as all points where $f(x, y, z) = C$ in the volume the array covers. C is the surface value, or level set.

Figure 3.8 shows that a rectangular volume is defined as the region between pairs of consecutive indices in each direction. In the following sections, we will call this region a "cube" regardless of the actual dimensions of the rectangular prism.

For efficiency, we limit the extent over which a drop's field is evaluated. A drop will only affect the values near it. We calculate a radius for each drop assuming that the drop volume is a sphere by

$$r = \sqrt[3]{\frac{3 * volume}{4\pi}}$$

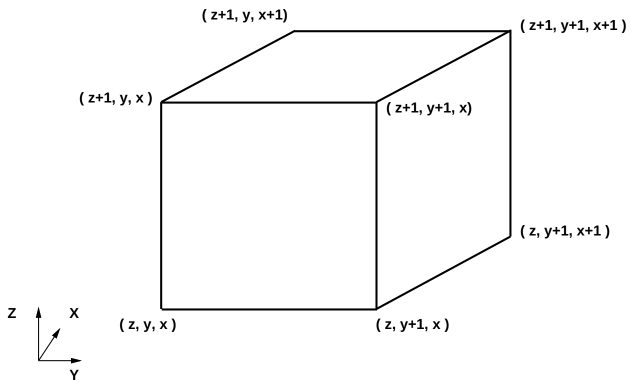


Figure 3.8 "Cube" Corner Values Defined by Array Values

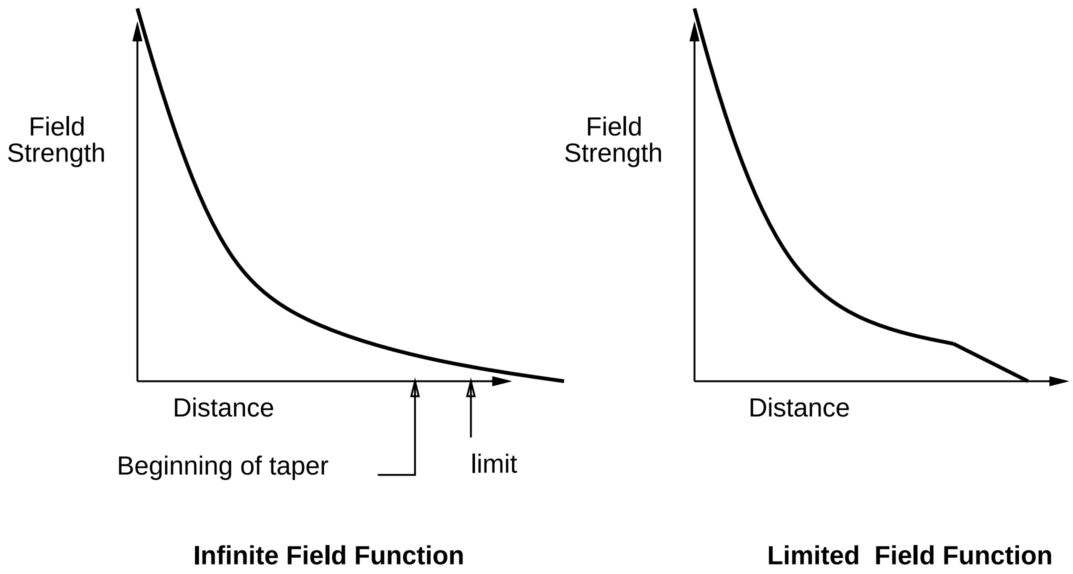


Figure 3.9 Infinite Drop Field Function versus Limited Field

The distance r is divided by the length of each cube side to find the spherical drop size in terms of array indices. Currently we increase the range by a factor of two in each direction for a total volume of eight times normal, and use that as a limit for a drop's contribution.

Although a drop's field decreases rapidly with increased distance as shown in Figure 3.9, to ensure that the field function is continuous we need to make an indi-

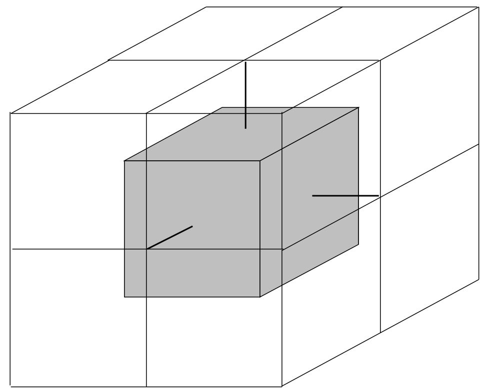


Figure 3.10 Simple Assumption Allows Quick Level Set Calculation

vidual drop's contribution drop to zero at the limits of its range. For each drop, we define a point at which the function begins to taper toward zero in each direction. The function is modified so that from the beginning of the taper to the limit is a linear function. Currently, the function is tapered over the last tenth of the extent of a drop's contribution.

To determine the array values, we walk through the list of drops in the system. For each drop, its contribution is added to those voxel vertices that are within its range. For each voxel, a list is kept of those drops contributing to the vertex values.

We choose a level set to enclose the correct volume. A simplification allows estimating the enclosed volume quickly. Each array value that is higher than the level denotes a point inside the fluid volume. We assume that the enclosed region is half the distance along each voxel corner in each of six directions. Figure 3.10 depicts the surrounding voxels for a single voxel vertex with array value greater than the level

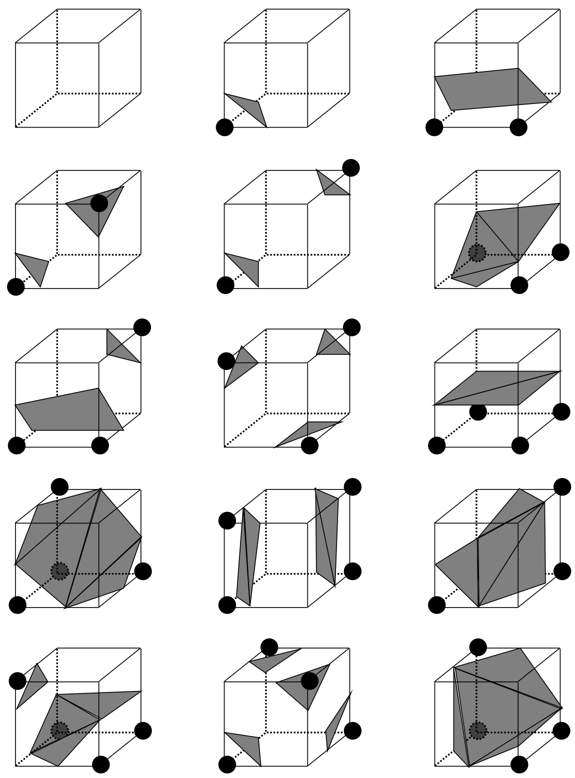


Figure 3.11 Fifteen Canonical Cases for Surface Detection

set. This vertex contributes one eighth of a voxel volume in eight voxels, or a single voxel volume total.

A binary search is done to determine the correct level set for the combined drop volume. Initially, the level set and upper bound chosen is π. This would be the correct level for isolated drops in space. The lower bound is zero, where the entire voxel array would be considered inside the drop volume. The enclosed volume for a level set is calculated as described above. If the enclosed volume is too low, the higher bound is set to the current level and a new value between the upper and lower bounds is tried. If the enclosed volume is too high, the lower bound is raised to the current level. When volumes for both a lower and upper bound are known, linear interpolation determines the next level set to try. A user-defined maximum of guesses will be tried in the search for the correct level set. The search is halted early when the enclosed volume is within a user-defined percentage tolerance of the desired volume.

Once the level set is determined, the array values are used to determine voxels through which the fluid surface passes. Whenever a cube has corners with values above and below the level set, the surface must pass through that cube. An 8-bit value can encode which cube vertices are "wet", that is, have a function value above the level set. A 256-entry lookup table defines the fluid polygon structure for each

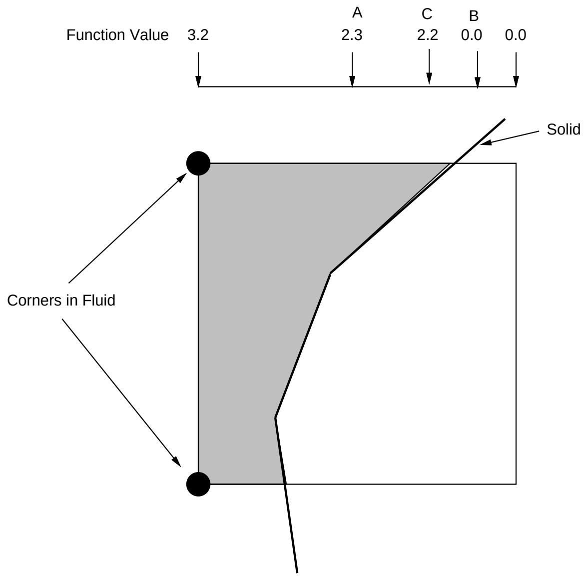


Figure 3.12 Finding Polygon Vertex Positions by Binary Search

case. Figure 3.11 shows the fifteen canonical cases. All 256 combinations are derived from these fifteen by rotations, inversions, and mirror imaging. The exact locations of the polygon vertices are calculated as described below.

In practice, to get a nicer polygonal approximation, each cube containing the fluid surface is subdivided into 8 or 27 smaller subcubes. Subdividing voxels near the fluid surface allows a finer polygonal mesh without the added overhead of additional voxels everywhere in the environment.

There is a minimum drop size to prevent problems in detecting the drops' surface. Small drops emit a weak field. When a small drop is located near a voxel corner, the corner is noticed as inside the fluid. If a small drop is centered inside a voxel, there may be no corners with field values high enough to be considered inside the fluid, so the drop temporarily disappears. Small moving drops thus tend to strobe in and out of existence, as Figure 3.13 shows. To circumvent this problem, there is a minimum drop volume, currently one-eighth of a pool voxel volume.

A more complicated surface-finding routine would be required to ensure that small drops were always detected. One way would be to treat small drops differently than large ones. Small drops' surface could be found using a different method from other drops'. Each small drop could be rendered using a local, finer grid. We have used the minimum drop size as the simpler solution.

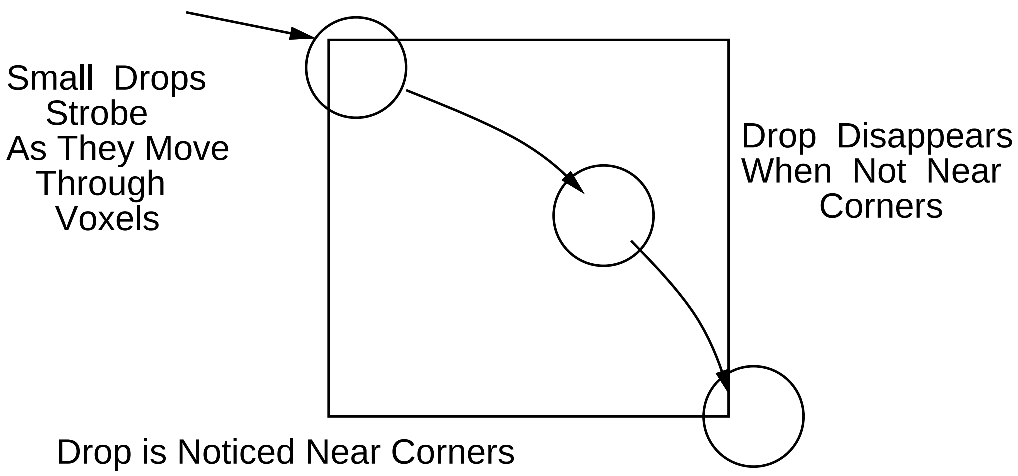


Figure 3.13 A Small Moving Drop's Surface is Detected Only Near Voxel Corners

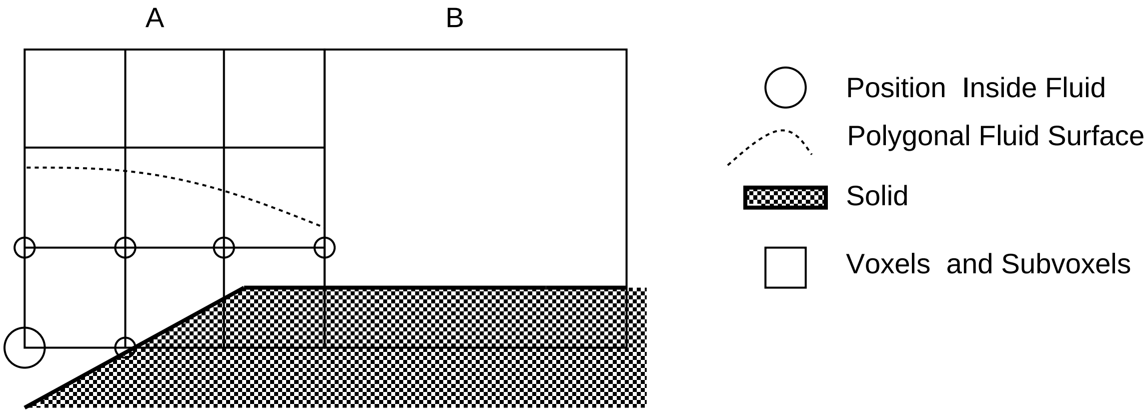


Figure 3.14 Hole caused by Unequal Sampling

We find the exact location for polygon edges by calculating the field function at additional points within the cube. The original method proposed by Lorenson and Cline suggests using a linear approximation of the field function within a cube. While that method is quick, it is not appropriate for our system because of the complications created by solids. Solids create field functions that are not continuous, invalidating a key assumption. We use a binary search to get a better approximation of the correct contour. We choose a position with the closest field value to the desired level set, except that we require the position have a field value above the level set. This ensures that the fluid surface we depict is slightly inside the ideal polygonal surface.

Consider the two-dimensional example depicted in Figure 3.12. The upper edge has one corner in the fluid and one dry corner. The fluid surface must cross that edge. In this example, the desired field value is 2.1 and the edge has endpoints with values 3.2 and 0. To find the fluid surface, first the field function is evaluated at point A, halfway between the endpoints of the edge. The voxel has a list of drops that contribute to its field, so only these drops are used in the evaluation. The function value at point A is 2.3, too high. Next the field is evaluated at point B, midway between two points known to bracket the desired value. Since the drops are all to the left of the solids, the field value at point B is 0, too low. Next the field is evaluated as 2.2 at point C. This value is above the desired level, but within the user-specified tolerance.

When a fluid surface is found in a voxel, all its neighboring voxels are examined for fluid surfaces also. Figure 3.14 depicts why this technique is used. Voxel A has a corner field value above the level set threshold. Voxel A is subdivided and the field function is evaluated on the subcubes, creating the fluid surface as shown. If Voxel B is not likewise subdivided and the field function evaluated, there will be a hole in the fluid surface. This hole is caused by the unequal sampling of space on adjacent voxels A and B. Forcing evaluation of subcubes near the detected surface corrects this source of error.

Chapter 4

Pools: Constrained Fluid

4.1 Pool Definitions and Goals

A pool, in our scheme, is a data structure to model fluid in a state of low kinetic energy. As the name implies, these are usually large bodies of liquid, in a nearly stable state.

As we have seen, fluids are defined by their ability to flow. Containers create fluid stability. For our purposes, a container might be an enclosing box, or simply an impenetrable floor. The containers prevent gravity from adding kinetic energy to the fluid. Friction between the container and the fluid slows the fluid, for a net decrease in fluid motion.

Under the influence of gravity, fluid seeks a state of lowest potential energy. Gravity and the ability of fluid molecules to move over one another cause fluid to flow downhill wherever possible. Stationary fluid has stopped moving because it is in a stable configuration. This flow causes stationary fluid to take the shape of its container, with a flat upper surface if the container is not full.

Pools correspond to traditional “upper surface” models. Since pools are usually moving slightly or not at all, and gravity has flattened the fluid, usually their only free surfaces are on the top. Lower fluid boundaries are usually defined by the container. Often only the upper surface needs to be rendered to produce acceptable results. The ocean, a water tank, and a cup of tea are all examples modeled well by pools.

While there are many possible implementations of pooled fluid, there are several qualities that an implementation should have regardless of the actual method used. These qualities provide a means for judging representations.

First, pools should be represented by a volume structure. One goal for the entire fluid modeling system is to represent the fluid volume correctly. For example a liter of water should maintain a constant volume, regardless of the environment. Adding fluid to a pool should increase its volume. Removing fluid should lessen the volume. Many previous simple models have no notion of the fluid volume.

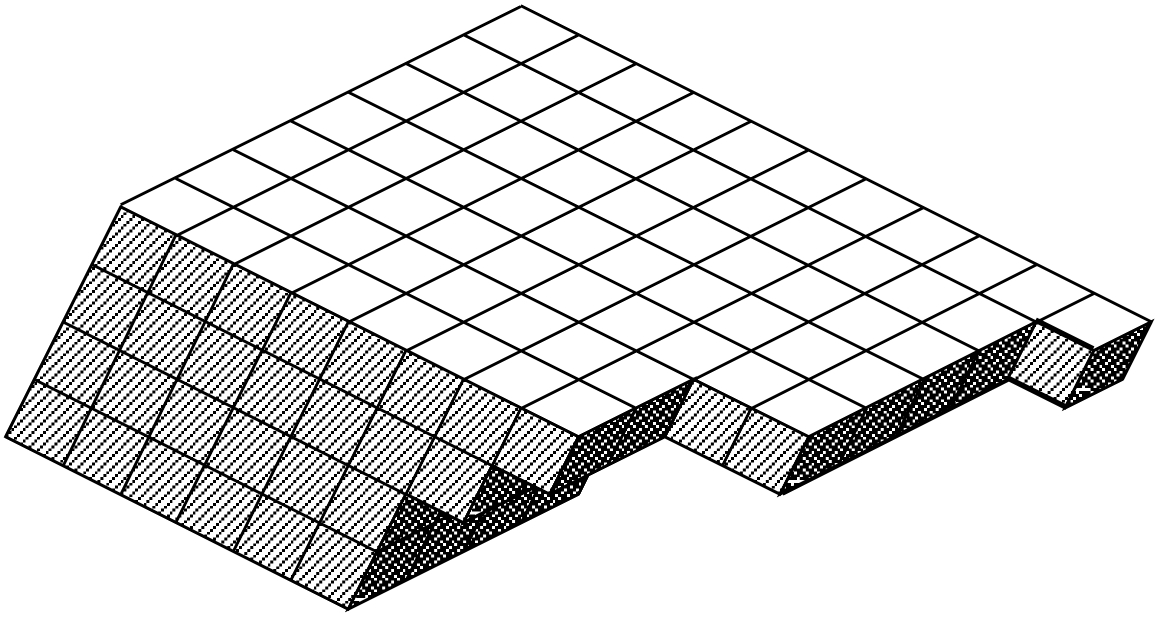


Figure 4.1 A Pool's Voxels Form a Contiguous Volume

Pools in a container should take the shape of their container. If the pool does not fill the container, any free upper surface should be flat to mimic the effect of gravity. The fluid-container boundary is often not modeled by previous systems. The container is often not explicitly defined at all. For instance, flight simulators generally model the ocean as a flat plane, and there is no representation for the ocean floor at all.

Pools, being fluid, should have the ability to move. By their nature, since we differentiate pools from drops by their stability, the amount of movement is usually limited. But without even the possibility of movement, the applicability of the model would be too restricted. For believability, pool movement should be gravity driven. Solids should guide the flow by preventing motion through them. Pools should seek a stable energy state by preferring downward motion to all other directions.

Finally, a good representation should be able to incorporate many of the previous representations' good features. A pool should be able to handle micro-features like ripples and waves. Lighting effects, such as reflections and refractions, should also be available. Any new technique should incorporate as many good features from the past as possible.

4.2 Representation

Pools are represented by a voxel structure of rectangular elements. The size, shape, orientation, and location of these pool voxels, or *pvoxes*, corresponds to the voxels used in drop surface calculations. Each pool maintains its own voxel structure. Each pool represents a single contiguous portion of fluid, so its pvoxes are contiguous also. Figure 4.1 shows a sample of a pool's voxels. The pvoxes delimit a region that contains the pool.

The top-level pool structure has data about the pool as a whole. The pool has a volume that, with the voxel information, determines the surface of the pool. The pool *height* is the world-coordinate of the highest portion of a pool. A list holds data on positions and times where drops have struck the pool. The list is used in creating wave patterns on the pool surface. Spatial extents are stored for each dimension for trivial overlap tests. The modeling space is divided into equal-sized voxels, which correspond to parts of the pool.

A three dimensional array of voxels holds more detailed information about the pool and its environment. Figure 4.2 shows a container holding some fluid and a

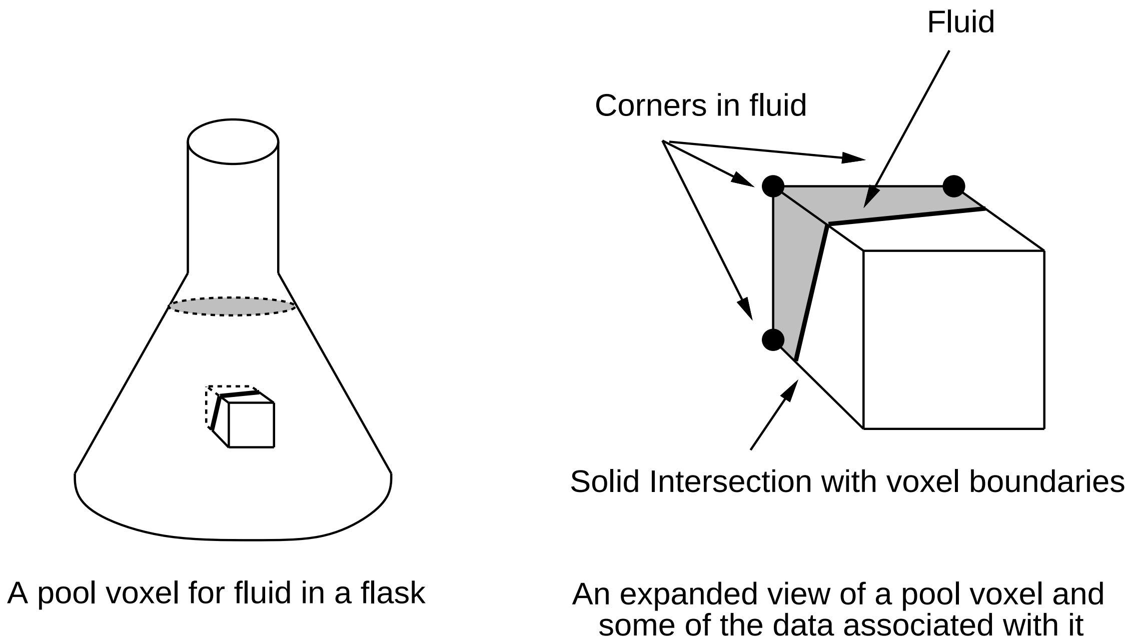


Figure 4.2 A pvox and some of its associated data

```
struct   pvox {               /* structure for a single pool voxel */
         int corners;         /* bitvector denoting fluid presence */
         float volume;        /* current volume in this pvox */
         float maxvolume;     /* maximum volume for current setup */
         CONS *surflist;      /* list of pool surface polygons */
         int supbelow;        /* bitvector of supported corners */
         int flags;
};
```

Figure 4.3 The pvox data structure

single pvox from the pool structure. Each voxel contains detailed information about the pool and the environment in a small region.

Figure 4.3 shows the C language structure representing a single pvox. The field *corners* is a bitvector recording the pvox corners inside the fluid, hereafter called "wet". *Maxvolume* is the maximum amount of fluid the wet corners can contribute to this voxel when the solids are taken into account. *Volume* is the amount of fluid in this voxel. The pool surfaces within the pvox are stored in a linked list in *surflist*. Support information for each pvox corner is stored in the *supbelow* bitvector, which is used in the movement calculations.

A global array holds information on the solids intersecting each voxel, including which voxel edges are intersected. Figure 4.4 shows the structure of this global array. The array is used to conserve space, since multiple entities need the same information about a voxel in space.

```
struct   box {                  /* a rectangular solid region in space */
         CONS *polylist;        /* polygons intersecting this box */
         CONS *faceedges[6];    /* for each face, the polygon edges on that face */
         CONS *fluid;           /* "drop" fluid surfaces in this box */
         CONS *pools;           /* pools affecting this box */
         CONS *drops;           /* drops affecting this box */
         int cuts;              /* 12 low bits are 0/1 for each box edge */
         float cutval[12];      /*for each edge, a point where a solid crosses the edge*/
         };
```

Figure 4.4 The global voxel data structure

A list of polygons intersecting a voxel is stored in the *polylist*. The intersection of the polygons with the sides of the voxel are stored as line segments in the *faceedges* list for each voxel face. These edges are used in the pool motion calculations described below. The *cuts* element is a bitvector recording the voxel edges crossed by a solid. The *cutval* array records a single value denoting the position of the intersection of an edge and a solid. The geometry of the model is assumed to be simple enough that a single value for the cut is sufficient. Voxels are supposed to be small enough that geometry is simple inside a single voxel.

4.2.1 Individual Pvox Calculations

Volume and support calculations proceed in a voxel by voxel basis. Summing pvox volume and capacity for a pool's voxels provide the total pool volume and capacity. The following sections describe the algorithms for determining capacity, actual volume, and support information for individual pvoxes.

4.2.1.1 Volumes

A capacity, or maximum volume, for each pvox is calculated using the *corner* bitvector, and the solid intersection data for a voxel. This quantity is the greatest volume a voxel with the given corner values and solids could hold. Volume calculations are estimates, for simplicity. For each pvox, the maximum fluid volume is proportional to

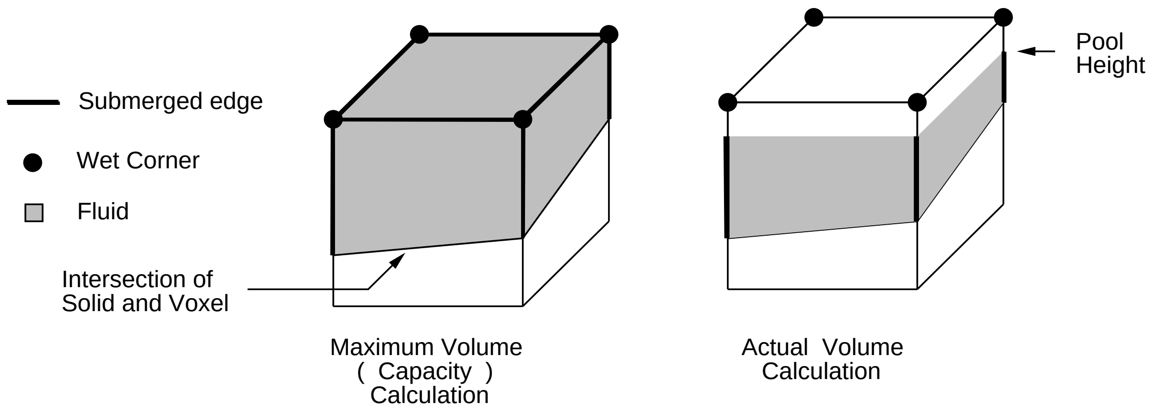


Figure 4.5 Voxel Volume Calculations

the percentage of its edges that are accessible from submerged corners. An example is pictured in Figure 4.5.

A few simple cases are handled quickly. A global variable *voxelvolume* is calculated once by multiplying the length, width, and height of a voxel. Voxels with all or no wet corners have easy maximum volumes: *voxelvolume* or zero. Voxels with no cuts have a maximum volume of *voxelvolume*. All other cases calculate maximum volume by looking at each voxel edge. If both corners are wet, the whole edge is wet. If a single edge is wet, the distance from the wet corner to the cut is wet. If neither corner is wet, the edge is dry. The maximum volume for a given *corner* vector and *cutval*s is calculated as the percentage of wet edges times *voxelvolume*.

Actual volume is calculated in a similar way, but additionally uses the pool height. The pool height further limits the fluid extent. The right side of Figure 4.5 shows pool height can reduce the fluid volume in a voxel. These volume calculations determine the correct height for the pool. The summed voxel volumes are the pool volume. As described below, the height is adjusted to maintain the correct total pool volume.

4.2.1.2 Support

The "support" concept in our system influences pool motion and the choice of representation. Supported pools flow in reaction to gravity. Unsupported portions of pools transform to the drop representation to begin free fall. Support is calculated using information local to each voxel and its neighbors.

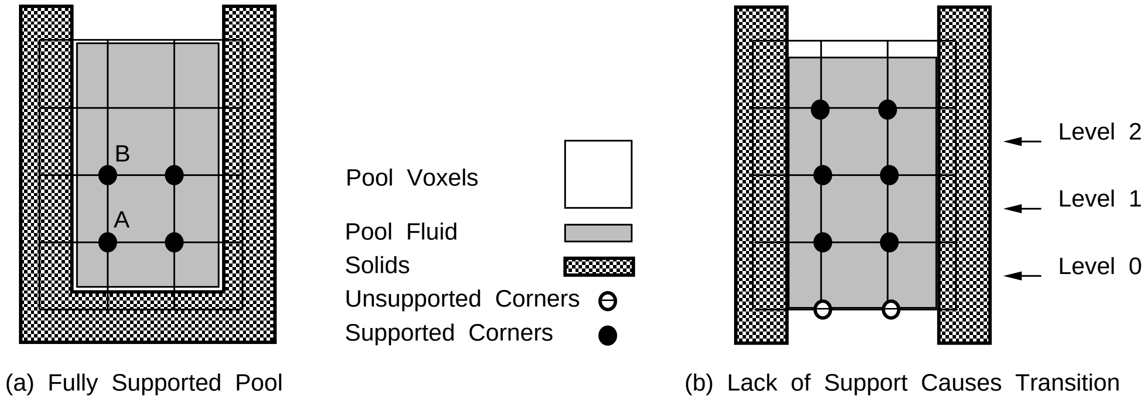


Figure 4.6 Support of Pvox Corners

Support for pvox corners means that there is something underneath the corner preventing free fall. The support may be a solid in the pvox itself or fluid in the pvox immediately underneath the corner For a single pvox, an upper wet corner is supported if there is a solid cutting the edge underneath the corner. Figure 4.6 shows a small pool's pvoxes and some wet corners. Corners **A** is supported because there is a solid crossing the voxel edge below **A**. Corner B is supported because there is fluid underneath it, namely at corner **A**.

When a wet pvox corner is unsupported, the fluid will begin to fall under the influence of gravity. As discussed in the next chapter, fluid in unsupported pvoxes changes to the drop representation. Letting fluid support other fluid causes a controlled transition, as Figure 4.6b shows. The lowest pvoxes in Figure 4.6b, marked Level 0, are unsupported because there is nothing below them to prevent downward movement. The fluid in the other pvoxes are temporarily supported by the fluid in the Level 0 pvoxes. The fluid in Level 0 pvoxes will change representation and begin falling. In successive time steps, the fluid in the higher level pvoxes will be unsupported and begin falling.

4.3 Voxel Traversal Algorithms

The pool voxel edges form a connected graph in three dimensions. Various calculations on pools involve traversal of this graph.,

4.3.1 Pool Creation

Because a pool is a complicated entity, user specification of a pool would be a time-consuming, error-prone process. The **protopool** construct allows the user to specify pool creation simply. The user-supplied information is simply a position and a volume.

Initially, a single pvox enclosing the point is created. The initial *corner* info is determined by the visibility of the pvox corners from the initial point. If there is no solid between the initial point and a corner, the corner is considered wet.

Given a collection of pvoxes, the pool creation algorithm proceeds recursively. For each pvox, flow within the pvox is handled as described below under pool movement. For each wet corner, the neighboring pvoxes to the sides or below are created and marked with their corresponding wet corners. The algorithm recurses for newly-created neighboring pvoxes, using depth-first traversal. This recursion stops when no

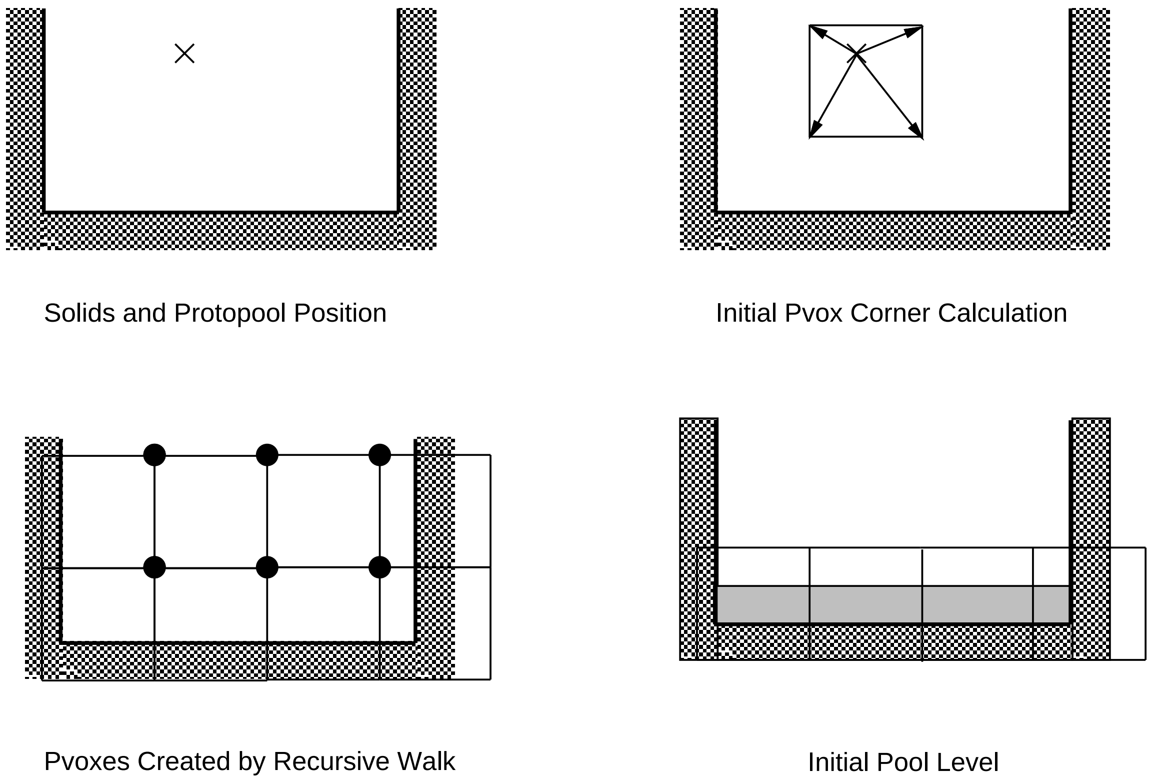


Figure 4.7 Protopool Pool Initialization Procedure

more sideways or downwards expansion is possible, either because solids enclose the pvoxes or pvoxes reach the limits of the simulation region.

After the recursive section creates pvoxes for the pool, the initial volume is compared to the volume the pvoxes can hold. A maximum volume for the pool is calculated by summing the pvox maximum volumes calculated as described above. If the maximum volume is larger than the user-specified volume, the pool height is calculated in the normal manner. If the pool maximum volume is too small, the pool is expanded as described below.

The complicated pool structure is added to the simulation with minimal effort of the user. The user merely has to be careful about the initial position and size. The initial position must be within the proposed container. The initial size should be small enough to fit within the container. If these two criteria are met, the pool location should be easy to predict.

4.3.2 Pool Movement

There have been only a few previous systems with large moving bodies of fluid. As we noted before, large bodies of fluid in computer graphics animations tend to be stationary. The fluid is usually only identifiable as fluid by context. Most of the work in this area has centered on how to realistically model waves on the upper surfaces of these fluids.

Oceans waves have been an area of research interest since the early days of flight simulators. Waves are generally formed by displacing a flat face. Portions of the plane are moved up or down, creating an uneven surface that is still single-valued in the Z direction. This is called a *height field*. An early video using this technique was "Carla's Island" [Max81], which superimposed time-varying sine waves to produce moving waves. Ts'o and Barsky used a similar technique but used splines to model the surface [TB87].

Additional wave realism has been accomplished by using physical characteristics to guide the wave motion. Wave speed and wavelength depend on the water depth. These speed changes make waves nearing the shore take the shape of the shoreline and determine where waves break. Peachey [Pea86] used depth to guide his height field models. Fournier and Reeves [FR86] used similar equations with a parametrically defined surface. Their model allows the surface to fold over itself, producing breaking waves. Both systems added a particle system to model foam. Using physical characteristics to guide the wave motion allows realistic wave deformations, greatly enhancing realism.

A drawback of the above ocean wave systems is that they do not allow any net fluid motion. Although waves give the impression of endless motion, the water actually moves in an elliptical path parallel to the apparent wave motion. The simpler systems above merely move the fluid at the upper surface up and down. The subject of the simulations, the ocean, does not really move towards the shore with each passing wave. Systems that take advantage of this are not applicable to situations where fluid really moves, like water cresting a dam and flowing into a new area.

Kass and Miller [KM90a] have the most capable upper-surface fluid system yet implemented. They base their technique on equations describing shallow water. Their implementation allows a net motion of fluid, a major improvement over previous systems. Their system also handles wave reflections and changing fluid topology.

Kass and Miller model fluid as a two-dimensional height field on a rectangular grid. They use a height field model based on three related assumptions: a height field is adequate, the vertical water velocity can be ignored, and the horizontal velocity is constant for a column of fluid. They acknowledge that the height field cannot model splashes and won't allow waves to break. Implicit in their writing is another assumption, the presence of an underlying support for the fluid.

Kass and Miller's approach models the fluid as a connected array of water columns. Fluid in a column has a height and a relative depth. The height is absolute, and the depth is calculated as the difference between the fluid height and the supporting solid underneath. These determine the flow of fluid from one area to another. To simplify their equations, Kass and Miller ignore flow information. There is no notion of fluid velocity within a column.

The fluid height is calculated from previous values of nearby columns, tending to remove height differences. Their system solves the differential equations

$$\frac{\partial^2}{\partial t^2} = gd\frac{\partial^2 h}{\partial x^2}$$

and

$$\frac{\partial^2}{\partial t^2} = gd\frac{\partial^2 h}{\partial y^2}$$

where g is the force of gravity, d is depth, and h is height. The two-dimensional problem is solved as two passes of a one-dimensional problem. The partial differential equations are solved using finite differences. Since the equations can be represented as a tridiagonal matrix multiplication, the system can be solved quickly.

Kass and Miller's system does several things better than previous systems. Their system is good at representing non-chaotic flow over nearly flat terrain. It allows for real motion of fluid from one location to another in contrast to mere perceived motion. Fluid can split and join naturally. Waves reflect off walls, an effect not normally found.

Their system still has several drawbacks. Because of the simplifying assumptions made, their work is only applicable for supported fluid. There must be an underlying terrain at all points. In addition, using a height field representation limits the type of terrain. Fluid in a pipe that curves under itself cannot be represented, because the fluid surface must be single-valued in the Z direction. The speed of wave propagation is determined by the grid size and the timestep. Since there is no velocity information stored for the fluid, the wave speed is constant for all waves. Flows likewise move at a constant speed, regardless of the steepness of the terrain.

Our implementation of pooled fluid uses a three-dimensional structure. Motion is determined in a manner similar to Kass and Miller's. The pool is divided into discrete regions. Flow occurs when adjoining regions have an unstable difference in fluid level. Solids can prevent fluid motion from one region to another.

4.3.3 Expansion

Pool movement begins by expanding the collection of pvoxes. The expansion process can add new voxels to a pool. After the expansion, the pool surface is computed as described below. At each time step, the system attempts to add voxels adjacent to previously occupied voxels. The surface composed of points a given distance, the *offset distance*, from the previous pool is called an offset surface. An offset surface from the old pool encompasses the potential new pool volume.

The offset surface prevents infinitely quick motion of the pool. Pools are conceptually formed by fluid building up at a location. Figure 4.8A shows various heights for a pool at various volumes. Figure 4.8B shows the problem with this simple model. If the pool volume changes slightly, causing a height difference, the fluid location can change over an arbitrary distance. Our model only allows fluid to occupy space near previous locations. An offset surface from previous fluid locations limits the region fluid can move to, and thus limits the speed of pool motion.

Pvoxes are added wherever solid constraints allow. The system examines the neighborhood of each previously submerged pvox. This is a breadth-first traversal from previous pvoxes. The offset distance is the number of steps in the traversal. Movement from old pvox corners is along pvox edges. Solids crossing a voxel edge

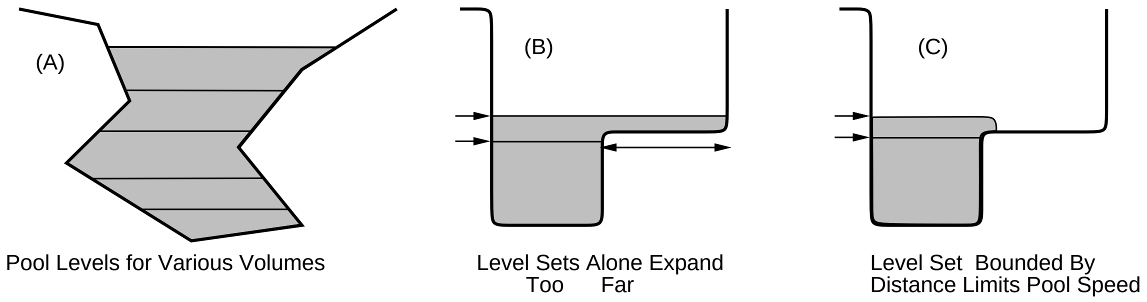


Figure 4.8 Bounded Pool Motion Limits Speed

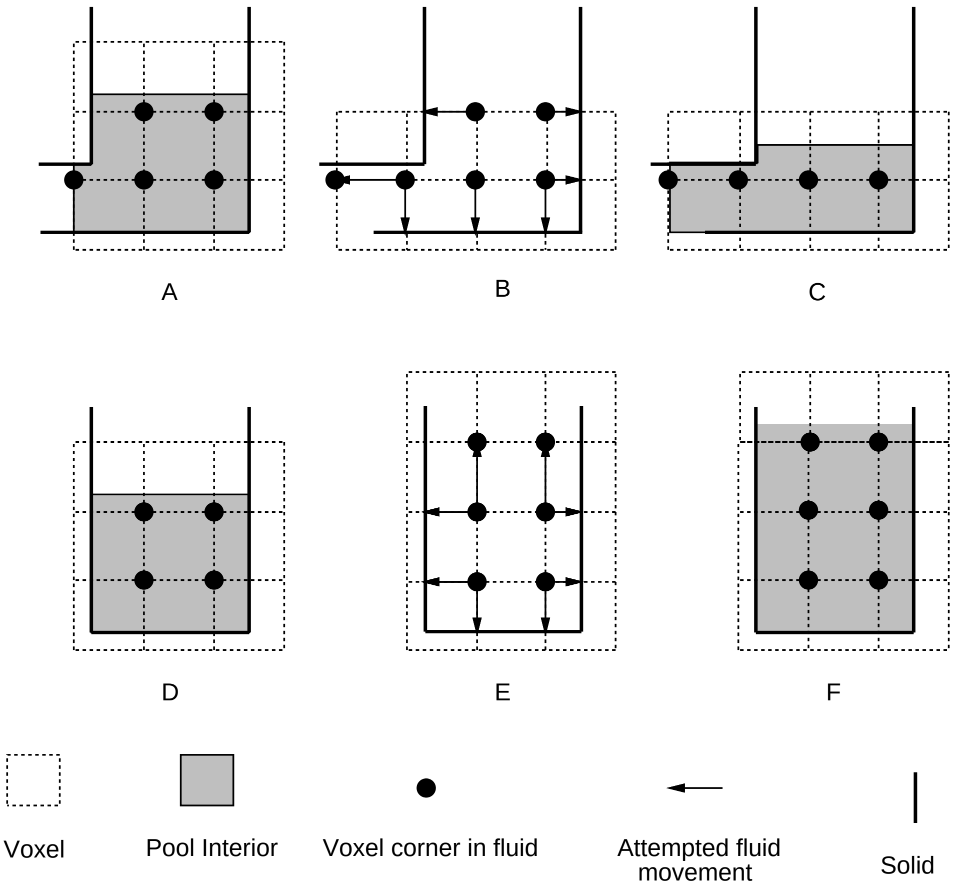


Figure 4.9 2D examples of pool motion

prevent fluid movement. Newly created voxels store information about from which direction their fluid arrived, the only flow information kept in the system.

Figure 4.9 depicts a two dimensional version of the process. Figure 4.9A shows some fluid in a tank, and the pvoxes. Submerged voxel corners are shown as black circles. In Figure 4.9B, arrows show the attempted expansion of the pool. Expansion occurs along voxel edges. In this example, solids block most of the pool movement, but two new voxels are added. Picking a surface height to maintain the pool volume results in Figure 4.9C.

When the total capacity of a pool's voxels cannot contain the pool's volume, the pool will expand upwards. Although gravity pulls fluid down, pools can rise when necessary. When a pool's volume increases, if the gravity-induced flow is not great enough, the pool height will rise. In Figure 4.9ABC, if the pool volume has increased by more than the capacity of the newly created pvoxes, the height will rise. In Figure 4.9DEF, increased pool volume will cause the pool height to rise because there is no outflow.

Each voxel corner is marked as "confined" if it is supported and there are solids to the sides preventing sideways motion. A pool will attempt to expand upwards only from "confined" voxel corners. Figure 4.9D shows a situation with confined voxel corners. The pool moves upwards to encompass two new corners. If the pool volume is unchanged, the surface will not change. If the pool has become larger, for example by absorbing more fluid, the level of the pool will rise as in Figure 4.9F.

Pools are by definition contiguous portions of fluid. As pools move, volumes change, and levels rise and fall, single pools may split or disjoint pools may merge. Manipulation of the changing topology is handled at each time step. To prevent volume flow between disjoint fluids, a split pool must become multiple pools. To maintain correct volume, overlapping pools must merge.

4.3.4 Pool Splitting

Splitting becomes necessary when a pool becomes discontinuous. Figure 4.10 shows a pool in a non-convex container. If the pool height lowers to the level in Figure 4.10B, the pool must be split. The pool height is consistent among the voxels in Figure 4.10A because flow tends to maintain equal heights. If Figure 4.10B is maintained as a single pool, there will be implicit fluid flow between the disjoint parts.

A breadth-first walk is used to ensure continuity. The *flags* element in the pvox structure (Figure 4.3) determines the connected components. The lowest 8 bits are used as a bitvector of connected corners. An initial wet corner is chosen as a starting

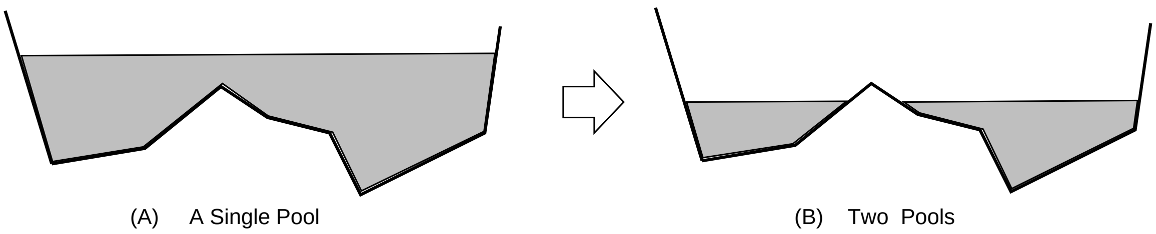


Figure 4.10 A Pool Splits into Two Pools

point. Fluid motion within the pvox is calculated as described above. For each wet corner, neighboring pvoxes with the corresponding wet corner are placed on a list. The process repeats on the first element of the list. When the list is empty, the *flags* elements of these pvoxes represent wet corners reachable from our initially chosen corner. If these correspond to the wet pvox corners, the pool is contiguous. If some wet corners are not reachable, the pool must be split. The process repeats on the unreached corners, in case the pool needs to be split into more than two pieces.

4.3.5 Pool Joining

Two pools that share a common wet pvox corner will be combined into a single pool. For pools with heights within a user-settable tolerance, the process is simple. Pools with widely varying heights take time to merge. The extra time allows simulating the flow necessary for the change to propagate.

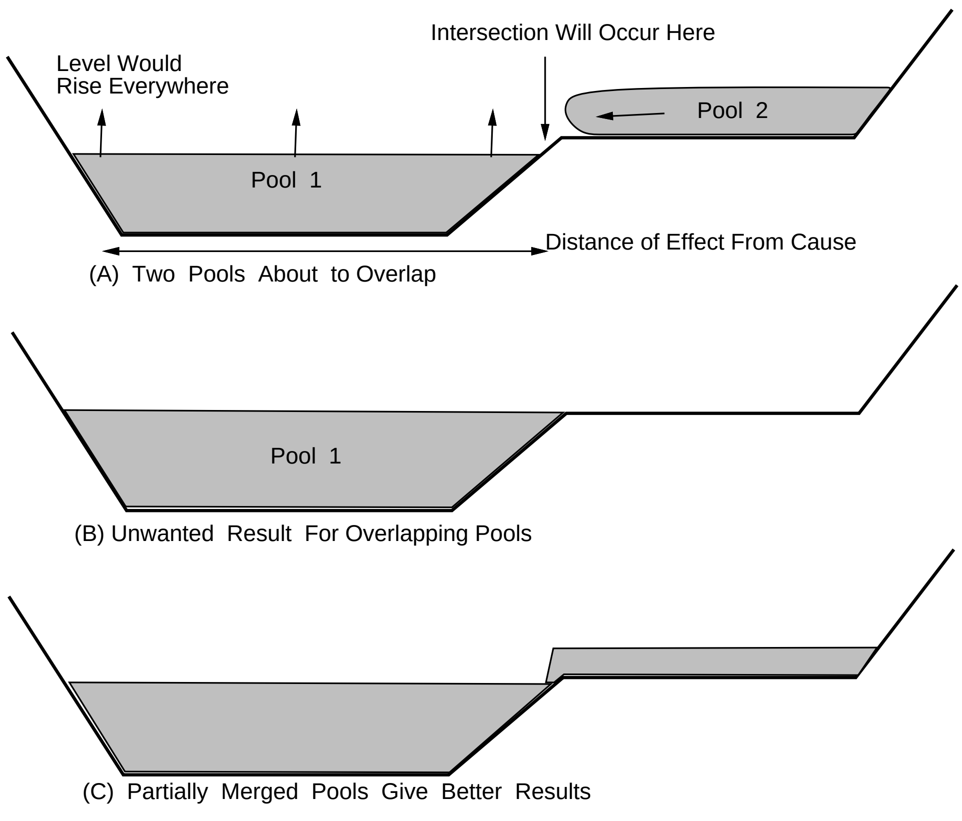


Figure 4.11 Merging of Pools of Different Heights

For pools with similar heights, merging is a matter of forming a single monolithic pool enclosing the same volume as the two component pools. The combined pool's pvoxes are a union of the two pools' pvoxes. Wet corners exist wherever either component had a wet corner. The pool volumes are simply added to produce the new pool's volume.

Combining pools of disparate height is not instantaneous. It takes time for fluid to flow from one structure to another. Consider the situation of Figure 4.11. Pool 2 moves to intersect Pool 1. If we merge the voxels using the simple routine described above, Pool 1 will rise immediately, perhaps leaving no fluid at all in the volume previously occupied by Pool 2. The problem comes from treating the pool as a whole when finding its height. For quiescent pools, this makes sense. For unstable states, it fails to account for the time necessary for changes to propagate. The intersection of the two pools affects distant pool regions equally as much as local regions. A better model, as described in Section 4.4.4, would correct this fault. To minimize this problem in the current implementation, we slow the merging process for pools of different heights.

For pools with different heights, volume is transferred from the higher pool to the lower. The amount transferred is the pvox volumes of the overlapping pvoxes. The pvoxes are deleted from the upper pool and their volumes are removed from the upper pool's volume. The lower pool's volume increases, which will normally raise its height during the next timestep. The overlapping voxels are deleted from the upper pool. The effects of overlap still occur over the entire lower pool, but are lessened. The additional time required makes the fluid movement more realistic.

4.4 Finding the Pool Surfaces

The first step in finding the surface of a pool is ensuring that the pool fits into its voxels. If the pool has more volume than its voxels can hold, the pool expands as described above, until the voxels are able to contain the pool.

If the voxels can hold the pool, the next step is to find the fluid within each voxel. The pvoxes and their enclosed solids comprise a container for the pool fluid. Conceptually, the fluid is poured into the container, filling it from the bottom, until the entire pool volume is present. Starting from the bottom level of voxels, the pvox maximum volumes are summed to find the maximum volume each level can hold. The pool fluid will be contained in the physically lowest pvoxes capable of holding the correct amount of fluid. Voxels below the top level are assumed to have their maximum volume of fluid. Voxels in the top level may be partially filled.

Polygonal surfaces are generated for voxels on the periphery of the pool. We have implemented several algorithms for producing these polygons. For the most part, the techniques are similar to the polygonalization algorithms in Lorenson and Cline[LC87] and Wyvill, McPheeters and Wyvill[WMW86b]. Corner and edge information for each voxel is used to create a polygonal representation of the fluid surface in the voxel. The upper fluid surface is found by intersecting a plane of constant height with the voxel and its corners.

4.4.1 Finding the Pool Upper Surface

A two dimensional algorithm finds a flat upper surface for the pool. Each voxel is processed separately. The data used are: the pool height, the list of solids in the pvox, and the fluid corner locations. If there are no solids in the pvox, a simple rectangular surface piece is created. The remainder of this section details how the pool surface is calculated in the more complicated situations.

Upper pool surfaces are created by a two dimensional graph traversal algorithm. The pool height determines the plane of the traversal. Graph edges are formed by intersecting this plane with the voxel boundaries and the solids in the voxel, as Figure 4.12A shows.

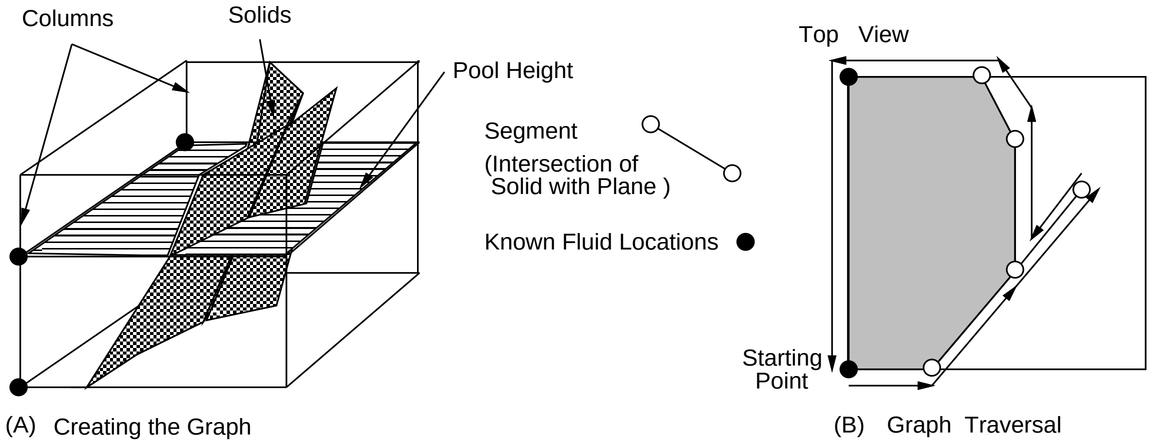


Figure 4.12 Two Dimensional Graph Traversal Determines Upper Pool Surfaces

We find which vertical pvox edges, called *columns*, have fluid at the pool height by examining the pvox corners and solids. A column will have fluid at the pool height if there is a wet corner at its endpoint with no solid between the corner and the plane of the pool. Figure 4.12A demonstrates a typical pvox. There are no solids between the wet pvox corners and the plane of the pool. Two columns have fluid at the pool height. The wet columns are recorded. Upper surface polygons will be created to account for all wet columns.

The solids in each voxel are also intersected with the plane of the pool to form segments in the graph. Since each solid is a simple convex polygon, the intersection with a plane cannot produce disjoint segments. If the solids do not intersect the plane, there will be no line segments and the pool surface will cover the entire pvox.

The segments are organized in two ways, by vertex and by segment. Each segment records the locations of its endpoints, to allow traversing the graph. Vertices point to all segments containing them, which gives the potential next segment at each point in the traversal.

Other edges in the graph are created during the traversal. The intersection of the pool's plane and the sides of the voxel create edges in the graph too. These edges are created as needed when the traversal has reached a side of the voxel.

A depth-first traversal begins at known fluid locations, along the edges, until the starting point is reached again. Each circuit defines a polygon on the pool surface. This process is repeated until all known fluid locations are on a circuit. These paths are the edges of the upper surface polygons for the pvox. Figure 4.12B shows the traversal for this example. The following section walks through the example step by step.

4.4.2 An Example of the Algorithm

The initial graph is generated from the situation of Figure 4.12A. The plane defined by the pool height intersects the voxel as shown. First the two columns with wet corners are examined. The voxel "cut" information described in section 4.2 is used to determine whether a solid intersects the column between the wet corners and the pool height. Since there are no solids intersecting the columns in this example, there are two initial known fluid locations at the pool height.

The plane of the pool intersects the solids in the voxel to form segments in the graph. Figure 4.13 shows the initial graph. The segment information includes pointers

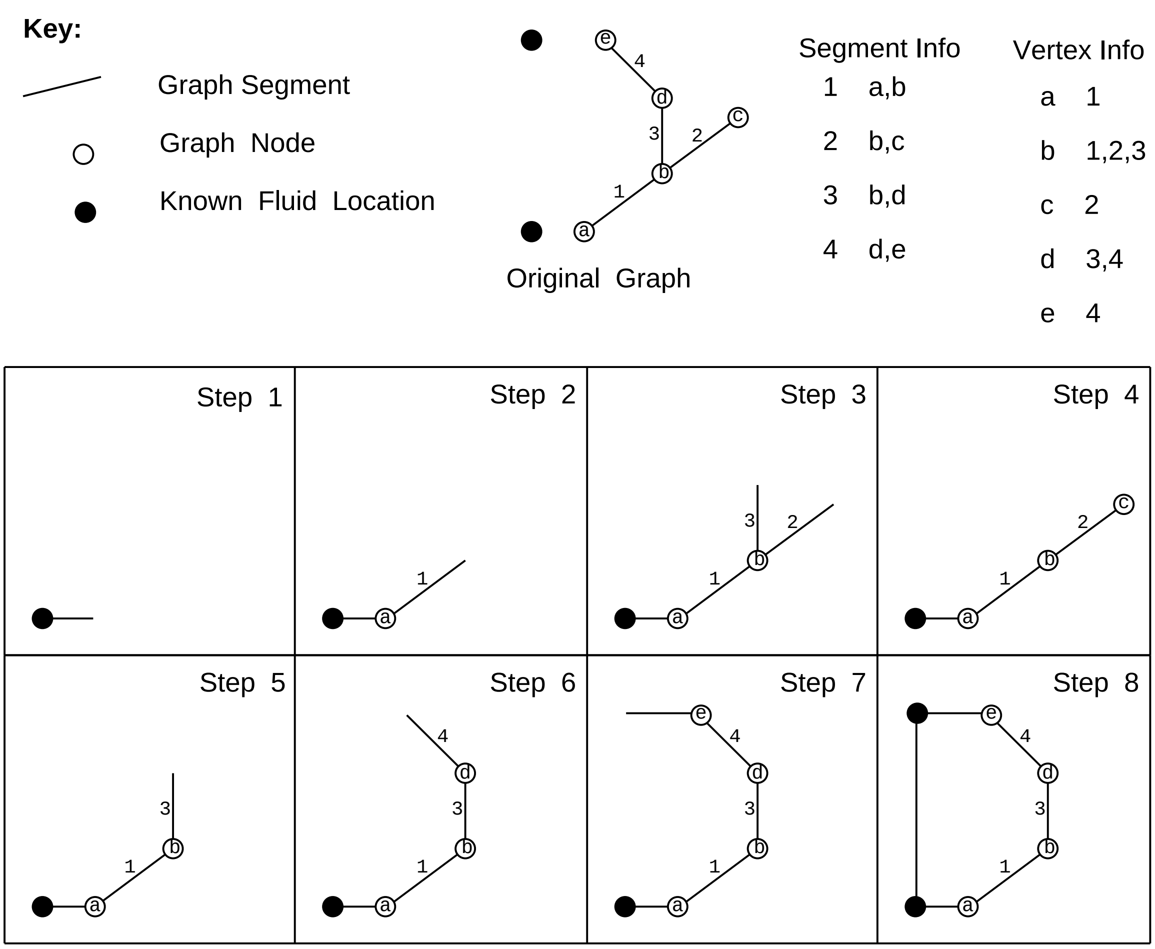


Figure 4.13 Example of Traversal Algorithm for Pool Surfaces

to the endpoints for each segment. The vertex graph information contains a list of segments containing this endpoint. These tables are summarized in Figure 4.13.

Initially, the walk begins at a known fluid location. Since this location is on a voxel side, with no edges associated with the vertex, an edge is generated along the voxel side. The walk proceeds in a counter-clockwise direction. The edge reaches to the first intersection with a solid, at vertex **a**. Vertices are pushed onto a stack as they are encountered. When a circuit is made, (the walk has progressed to its original point), the stack holds the polygon vertices in order.

In Step 2, the walk has traversed to vertex **a**, where there is only one choice of segment, segment **1**. Segment 1 connects vertices **a** and **b**, so the next location is vertex **b**. In Step 3 there are two choices for segments, since three segments meet at vertex **b**. Segment **1** is marked as used, and so will not be walked again. The algorithm proceeds along segment **2** to vertex **c**.

Since there are no untravelled segments for vertex **c**, the algorithm backs up by popping the stack, and returning to vertex **b**. Now there is only one untried segment from vertex **b**, namely segment **3**. In Step 6, the algorithm is at vertex **d**, with only segment **4** as a choice. The next vertex is vertex **e**.

Vertex **e** has no other segments, and is on the voxel side, so an edge is generated starting at vertex **e** in the counterclockwise direction. This new segment reaches a wet corner. From the wet corner, a segment is generated again in the counterclockwise direction. This last segment reaches the starting vertex, making a circuit.

An upper surface polygon for this voxel has been defined. The polygon has six sides and accounts for two wet corners in the plane. Since there are no unreached known fluid locations from the original graph that are not accounted for by this polygon, the algorithm ceases for this voxel.

4.4.3 Finding Other Pool Surfaces

Previous systems have ignored all pool surfaces but the uppermost. Usually, this simplification causes no problems. Pools are stable, taking the shape of their containers except for upper surface phenomena. There are a few cases where all pool surfaces are required. Pools in an unstable state may have lower surfaces that are changing. Rendering these moving lower surfaces can be important. The moving wall of water created by a bursting dam would not be sufficiently modeled by its upper

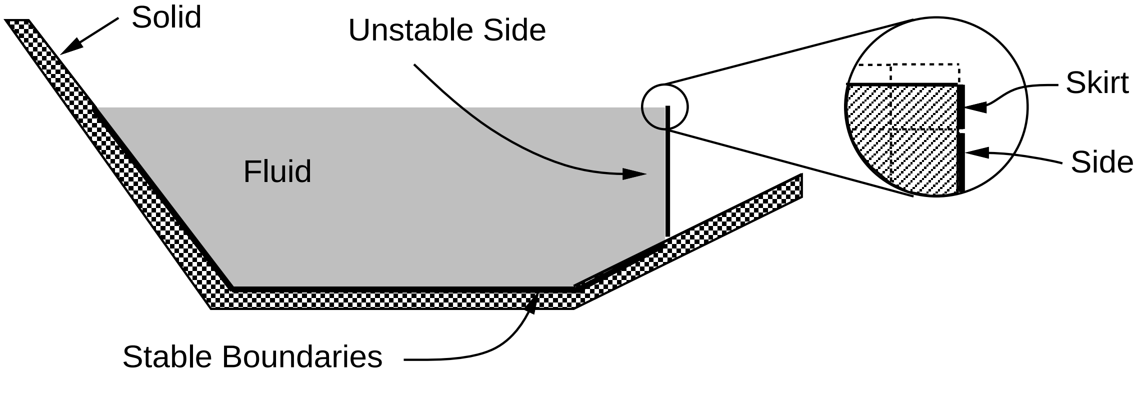


Figure 4.14 Unstable Pool Sections Need Side Polygons

surface. Lighting effects can require generating lower pool surfaces when the container is transparent or translucent. Correctly rendering a filled fish tank, for example, may require explicit glass-water interface data for refraction calculations. We provide a user-selectable switch to control whether pool side data is generated.

Unstable pool regions often have an air/fluid boundary on the sides as well as the top. Figure 4.14 displays an unstable pool. Although sometimes the sides could be ignored, some choices of camera position make explicit side polygon information necessary. We implement two degrees of side information: unstable sides and all boundaries.

Unstable sides are present in pvoxes with wet corners on a pvox side with no neighboring pvox on that side. Figure 4.15 shows two pvox sides in dashed lines. Each of these pvox sides has at least one wet corner, and does not adjoin another pvox. Polygons are created for these sides by the same two dimensional graph traversal algorithm as described for the pool upper surface. The difference is that the plane is a vertical one, instead of horizontal.

The lower traversal is just like the upper surface algorithm. Corner information and line segments provide the graph to traverse. For the skirt, in the upper traversal, a slight modification is needed. The pool upper surface polygon is treated as a solid, so the intersection of the pool top and the pvox side creates a segment in the graph. The traversal algorithm creates a polygonal side for the pool.

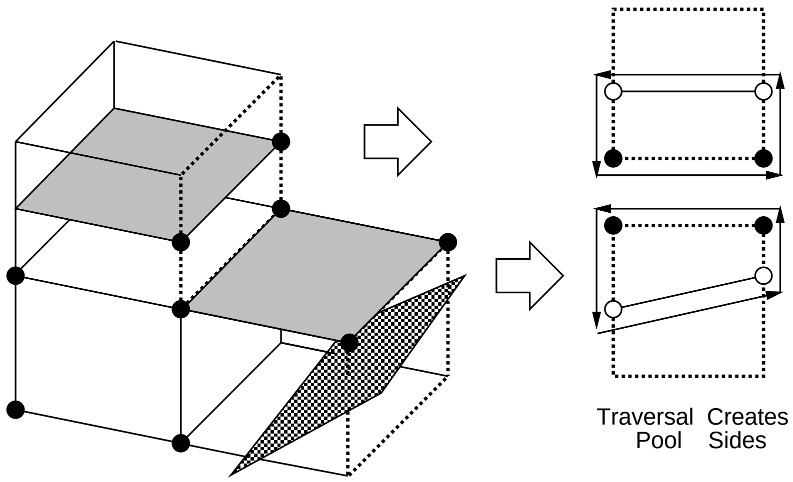


Figure 4.15 Pool Sides Generated for Peripheral Pvoxes

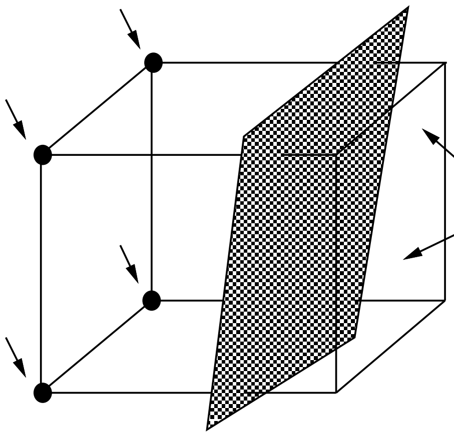


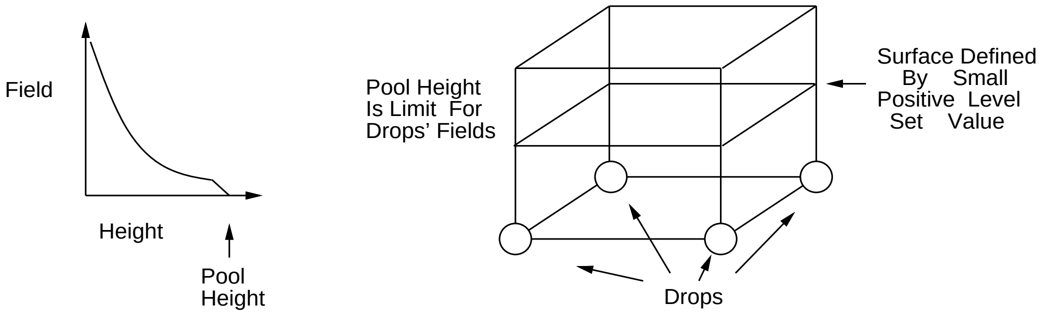
Figure 4.16 Stable Pool Boundaries Created with Field Functions

Stable boundaries, ones created by solids, are created with ideas used to find drop surfaces. Figure 4.16 shows a pvox where a solid creates a stable fluid boundary. An array of drops, used only in the pool surface calculations, are placed in known fluid locations within a single pvox. We have used the wet pvox corners for these drop locations for simplicity. The drops are considered as radiation sources. A surface defined by a level set with small positive value is found using the same procedure described in the previous chapter. Since the drops' summed field function is zero outside the container, the algorithm produces a surface just inside the container. Conceptually, these drops are surrounded by a bubble. Solids limit the bubble volume, producing a bubble surface just inside the solids nearest the drops.

4.4.4 Other Pool Surface Methods

Other methods for finding pool surfaces have been considered, but not implemented. After further experimentation, one of these may prove a better method than the ones described above.

The same drop-based method described above for creating stable pool side polygons can be used to create the upper surface too. Each drop within the pvox has its upper limit set to the pool height. The drops' field functions are zero above the pool height. The limited drop field technique is explained in Chapter 3. A surface is found for a small positive level set, which will be close to the pool height. The

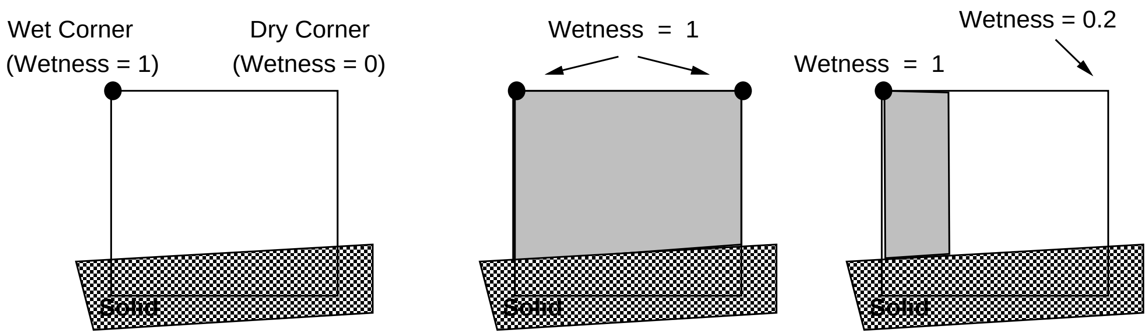


(A) Field Value Nonzero Over Limited Region (B) Pool Upper Suface Defined Using Drop Surface Method

Figure 4.17 Alternate Pool Upper Surface Method

advantage of this approach is that the top and side polygon calculations use the same algorithm. The disadvantage is that the calculations are more complicated than the two-dimensional approach described above.

Another method considered but not implemented involves keeping more corner information at each pvox. Currently, the wetness structure is a bitvector: a pvox corner is wet(1) or dry(0). Figure 4.18 shows the consequences of a binary wetness indicator. Fluid motion within a pvox is done in a single timestep, as the transition from Figure 4.18 A to B shows. In simulations, the grid size becomes apparent when



(A) Beginning of Timestep (B) End of Timestep (C) Proposed Non-binary Wetness Value

Figure 4.18 Non-binary Wetness Values for Pvox Corners Would Allow Partially Filled Pvoxes pools move in discrete voxel-sized steps, unless the gridsize is small. A finer grid adds computational complexity to the simulation.

If the corner wetness values could range from zero to one, instead of the current zero or one, wet/dry, method, perhaps pool motion could be a more controlled process. More detailed wetness information would allow description of partially-filled voxels, as Figure 4.18C shows.

4.4.5 Surface Effects

Standard surface techniques add realism to the pool surfaces. Figure 4.19 shows a couple of traditional techniques. At the top is a surface created by taking a plane and displacing portions varying amounts to create a curved surface. Traditional graphics display algorithms use the surface normals, light position, and viewer orientation to shade the surface as in the middle surface. Mapping the surface normals onto a flat surface, as the bottom example shows, and using the same shading techniques

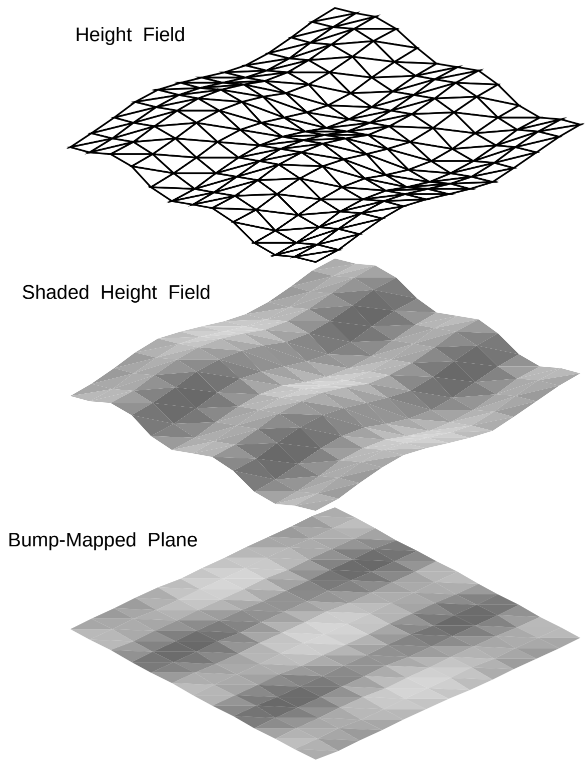


Figure 4.19 Wave Representations

gives the illusion of a curved surface even though the underlying surface is flat. This technique, called bump mapping, allows adding perceptual detail with a small amount of work. Both of these techniques, surface displacement and bump mapping, have been used to create the impression of waves on planar surfaces.

Variations between implementations arise from the method for creating the displacements or surface normal patterns. Max[Max81] adds sin waves. Perlin[Per85] uses three dimensional procedural texture functions. Mastin, Waterberg, and Mareda [MWM87] use filtered white noise as a texture source. Ts'o and Barsky[TB87] trace rays representing a user-defined wavefront. Peachey[Pea86] uses a linear source of sine waves. Fournier and Reeves[FR86] move trains of waves along the ocean floor. Each of these techniques has its strong and weak points.

We use a similar method to add waves to the upper surface of pools. We can use either bump mapping or surface displacement. Surface displacement is harder because care must be taken to avoid moving the surface through a solid. Surface displacement is more realistic, especially for large displacements. Silhouettes are modeled correctly, and shadows and refractions can be created correctly. Bump mapping is easier, because there are fewer calculations involved. Bump mapping is preferable for distant views since it gives a similar result with less work.

The source of waves is drops that interact with the pool. Each pool keeps a list of drops that have impacted it. Normally the drop is absorbed into the pool when the impact occurs, as is discussed in Chapter 5. The drop position, size, and speed are recorded along with the time when collision is detected.

Spherical waves, centered at the point of impact, modify the appearance of the surface. Figure 4.20 shows how each wave source creates a single peak and trough.

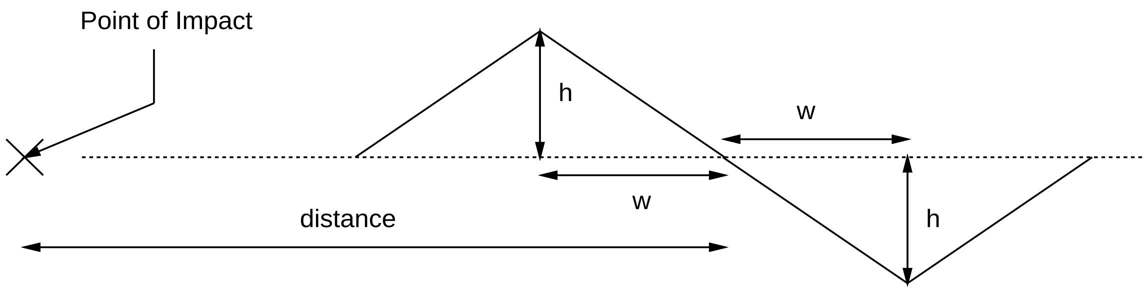


Figure 4.20 Wave Created by a Single Source

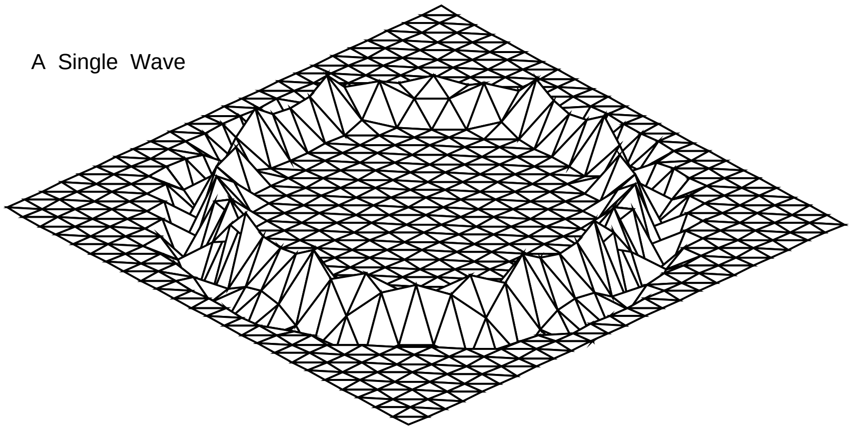


Figure 4.21 A single Impact Creates a Circular Wave

The wave trough and peak are symmetric, so the wave causes only a small loss of volume. This is important so that adding many waves to a pool does not visibly change its volume. Waves have a limited lifetime, stored in a global value. The wave has a height **h** that is initially the radius of the drop creating the wave and that linearly shrinks to zero during the lifetime of the wave. The wave center moves at a constant speed taken from the creating drop. The width of the wave, **w**, is a constant. All these details could be modified if needed. They were chosen for simplicity.

If surface displacement is used, as in the current implementation, care must be taken not to displace the surface through solids in the scene. For each surface corner, the effects of the pool's wave sources are added to create an offset for the point. Each source will not contribute to an offset if the offset would then be larger than its maximum wave height. This limit prevents catastrophic upward or downward movement that could be caused by the chance superposition of waves. Once an offset is defined, the path from the surface initial position to its offset position is tested for intersecting solids in the same way as drop collision detection. If there is no solid in the path, the surface vertex is moved by the offset. If a solid intervenes, the vertex is moved to the point of intersection.

Chapter 5

Transitions Between Representations

Previous systems for modeling fluid generally contain a single representation for the fluid. Their results are generally good, but are limited in scope. The characteristics of fluid are too varied to be captured in a single representation. For instance, the ocean's surface far from shore is well modeled by a simple flat surface model. There is no fluid transport, only slight changes in the appearance of the surface. Waterfalls present a different challenge: representation of chaotic, surging, energetic fluid where motion dominates. Representing the exact waterfall surface is less important than correct mimicking of the overall motion.

Our system uses two representations, making it useful in more varied situations than previous schemes. Although single-representation systems often produce good results for still photos, it can be difficult to use them in animations, which have the extra dimension of time. Situations change. Sometimes fluid changes from a chaotic state to a placid one, or vice-versa. There are several possibilities for dealing with this problem: use a single representation that works poorly in some situations, find a single representation that does all things well, never allow a changing environment, use multiple representations while managing transitions manually, or use multiple representations and have transitions automatically managed by the system.

Single-representation systems use various methods for dealing with fluid for which the representation is ill-suited. A single representation can be used for all situations, despite poor results. Most animators do not willingly use this approach. An animator can ensure that only good results are recorded. Sometimes bad situations can simply be disallowed: a quiet pool of fluid is never disturbed. At other times, the ill-suited situations are only allowed to occur off-camera. Attention is drawn to regions where the representation works well. Fluid in inappropriate regions is recycled or ignored.

So far, computer graphics researchers have discovered no single representation suitable for all physical situations. Even in the mechanical engineering world of fluid analysis, there are two general methods in use. Lagrangian methods follow a small portion of fluid as it moves. These methods are closely related to particle systems as used by computer graphics artists. They are used primarily in sharply bounded fluid flows, such as the motion of isolated fluid droplets. The bulk of fluid mechanics algorithms use Eulerian methods instead of Lagrangian. Eulerian methods compute the changing characteristics at fixed positions, for instance the velocity and pressure at points within a pipe. Eulerian methods work best where fluid is omnipresent so that fluid always exists at the fixed points. For fluid flowing through a pipe, the fluid constantly moves, but the pipe is always full. If the desired information is the rate of flow for a cross-section of the pipe, it is important that there really be fluid in the pipe over the whole section. A single representation is unlikely to model diverse situations well.

It is possible to use multiple fluid representations within an animation, having the animator control each part. This gives total control to an artist. For instance, in the film *Red's Dream*[Pix87], there are two fluid systems: the rain, and the puddles on the street. The rain is a particle system, and the puddles are a non-moving upper-surface model. There are splashes on the puddles, created by a separate particle system. Reeves simplified the design by having the fluid systems unrelated to each other. There is no correspondence between raindrops and the splashes. In addition, the raindrops do not interact with the rest of the scene at all.

We consider a single automated system, composed of multiple fluid representations, a better alternative than the above strategies. The appropriate representation is used in each part of the system. Automated transitions allow modeling a wide range of physical situations while relieving the animator of the overwhelming burden that manual control requires.

The idea of *support* drives when and where transitions occur. Loosely, fluid is supported when there are solids underneath the fluid. Implicit in the situations previous systems model are assumptions about support. Particle systems work well where there is no support. Upper surface models require support to be present. We use support to decide which representation to use. When inappropriate situations arise, the representation will change.

5.1 Drops Become Pools

As discussed above, we use a particle systems of drops to represent high-energy fluid. In the real world, high energy doesn't last. Internal friction(viscosity) and external forces absorb energy from moving fluid. Energy input to a system mainly comes from gravity, so unless fluid is endlessly falling, energy input is of short duration.

Animations based on particle systems alone are limited by not handling stationary fluid. In the video *Particle Dreams*[Sim88], Sims depicts a waterfall composed of approximately 60,000 particles. The particles are created at the top of a series of planar and spherical solid obstacles. The particles fall and rebound off the solids until reaching a lower plane. At the bottom, while they are still moving, the particles are recycled. Particles are removed before they can reach an inappropriate situation.

In the rain sequence of *Red's Dream*[Pix87], Reeves uses trickery to avoid dealing with stationary particles. He achieves the correct visual effect without the expense of simulating physics. Raindrops fall, but do not interact with anything else in the scene. They are conceptually falling in a featureless void. The rest of the scene, the puddles and the solid objects are not considered at all. The drops are merely overlaid onto the scene. A separate particle system creates the ripples in the puddles, creating a realistic appearance. The ripples do not occur where drops strike puddles, but that is virtually impossible to detect.

In their video *Upwardly Mobile*[Mil87], Miller and Pearce show a fountain of fluid made up of interpenetrating spherical particles. The fountain itself looks good, but the illusion of fluid weakens as the spheres fall to the floor. The spherical shapes are apparent, especially for those spheres that are isolated from the others.

5.1.1 Criterion for Transforming a Drop into a Pool

In our system, drops in a quiet state convert to pool representation. The transition occurs when a drop is non-moving (idle) for a user-specified length of time. This allows using a straightforward method of transformation. This is not a perfect choice of criteria. A slowly moving stream is probably more like a pool than a collection of drops. It is possible that our system would represent the stream as drops. Simple rules governing transitions are undoubtedly inadequate to ensure the best representation in all cases. A tradeoff between complexity and speed forced the use of a simple criterion.

While the actual transformation is instantaneous, appearance changes take place before a drop becomes a pool. At transition, the drop's appearance should mimic the pool it will become. As discussed above, supported drops flatten out over time. The flattened drop volume becomes more and more like the volume for a pool in the same location. Drops on flat surfaces will spread out. This motion is similar to what the pool structure does in the same location.

Drops supported by an underlying solid lose speed from friction, tending to slow the drop. Eventually a supported drop will stop altogether, at least if the solid is flat enough that gravity doesn't continually move the drop along. Once a drop is stopped, a timer begins. A non-moving drop's structure records how long it has been stopped. If a drop is not moving for some user-specified time, its representation is changed into a pool. Support causes a drop to slow and eventually stop, indirectly causing a transition.

An exception is made to mimic the effects of surface tension. Small drops, defined as having volume less than a user-defined minimum, are allowed to remain drops even though they are stationary. Physically, the surface tension of these drops is strong enough to counteract the sheer forces on the drop. By specifying an appropriate minimum drop size, the animator can successfully represent surface tension in this instance. This minimum drop size would not prevent absorption of drops into existing pools.

5.1.2 Method for Drop Transformation

Creating a pool from a drop involves creating a voxel-oriented pool structure that encloses the volume of the drop. To convert a drop to pool form, first the drop's surface is determined. The method is as described in Chapter 3, except that only a single drop is involved. The drop's field is evaluated in a volume surrounding the

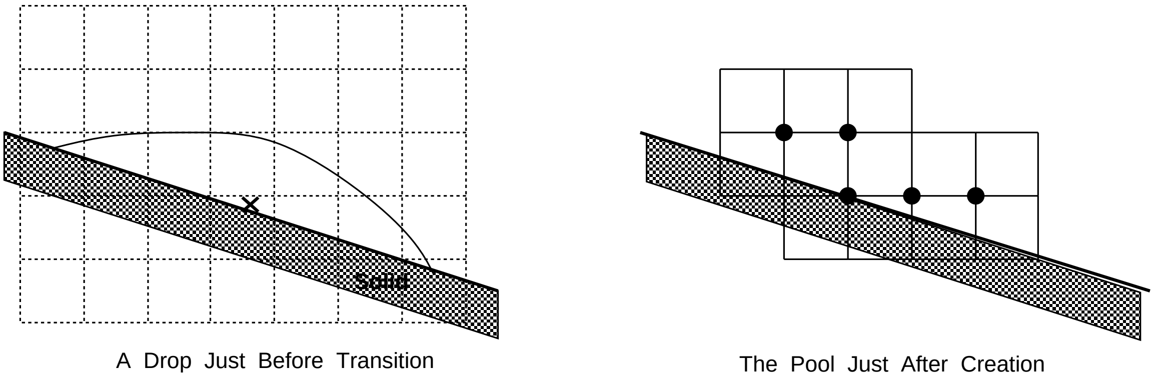


Figure 5.1 Example: Transformation from Drop to Pool

drop. A level set is chosen that encloses the correct volume for the single drop, as described previously. A pool voxel is created for each voxel having a corner inside the drop volume. Thus the newly created pool encloses nearly the same volume as the drop does by itself.

The process is diagramed in Figure 5.1. The left side shows a drop just before transformation. The drop field is evaluated over the grid as shown. A field level is chosen to enclose the correct drop volume, which would generate the fluid surface drawn. The right side depicts the pools structure immediately after transition. Voxels containing fluid on the left are mimicked by pool voxels on the right. Corner data, used by the Marching Cubes algorithm to generate a polygonal fluid surface for the drop, marks "wet" corners on the pool voxels.

The transition of an idle drop into a pool occurs at the end of a time step. The fluid surfaces for that time step have already been formed by the time transition happens. The idle drop is deleted but replaced by a pool of equal volume so that the transition preserves fluid volume. During the next time step, the pool will move as described in Chapter 4.

5.1.3 Absorption

There is another situation where a drop becomes part of a pool. Many animations have included the situation where drops of water fall into a larger body of fluid, creating ripples. Demonstration programs depict the surface waves in real time. Usually the addition of drops to the pooled liquid only affects its surface, not the volume.

In our system, drops that fall into a pool are absorbed. This is an obvious solution that coincidentally makes volume preservation easy. The volume calculations for drops and pools are done independently. Volume common to both introduces error, causing loss of system volume. Some error of this type is acceptable, but we minimize it by transferring fluid volume from the drop to the pool in this situation. This transformation is simpler than the transition described above.

Drops that are inside a pool are absorbed by the pool. At each time step, drops are checked to see if they are inside a pool by checking each drop's voxel location. If a pool is in the same voxel, and a voxel corner is both in the pool and visible from the drop, the drop is considered to be in the pool.

If a drop is in a pool, the drop location and the time are stored in the pool's impact list. This list contains the information used to create the ripples on the pool surface. A description of the impact list and its use is found in Chapter 4.

When a drop is enclosed in a pool, the drop volume is added to the volume of the pool, and the drop is deleted. During the next time step, the pool will expand as the pool surface is determined in the normal manner. Usually, the pool volume is much greater than a drop's, so the additional volume will change the pool's position only slightly.

5.2 Pools Become Drops

Upper surface models work well in many situations. The common characteristic of these situations is an implied base of support. The fluid is always on top of a solid. Many upper surface models do not allow movement of fluid at all, but if they do, as in the video *Splash Dance* [KM90b], there is no provision for what we call unsupported fluid.

Consider a lake formed from a dammed stream. It is easily modeled by its upper surface. Often, a static upper surface is sufficient to display the lake. There is assumed to be no fluid motion, or that it is unimportant. If the dam is opened, or the lake overflows, upper surface methods do not realistically represent the correct fluid motion.

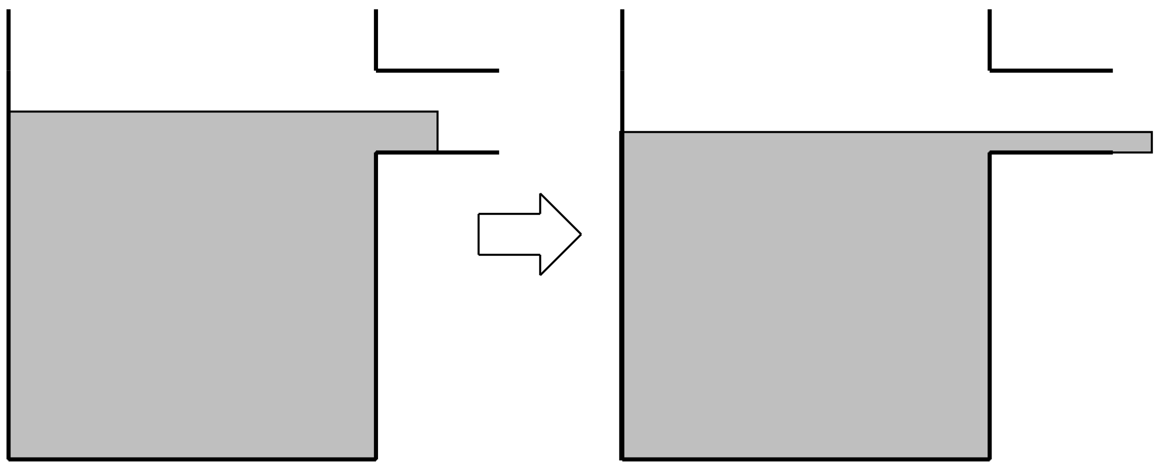


Figure 5.2 Pool Motion Causes Transition Situation

Since we allow large bodies of fluid to move, pools can easily encounter situations where an upper surface model is inappropriate. Figure 5.2 shows such a sequence. On the left, a pool is in an unstable situation. Its motion causes the situation on the right, where some of the pool has no support. A stationary pool would not have this problem.

5.2.1 Criterion for Transforming Part of a Pool to a Drop

When pool voxels are unsupported, they are transformed into drops. As stated before, upper surface models are dependent on support to be valid. We therefore can use a simple rule to determine when to abandon an upper surface representation. A pool voxel corner is unsupported if there is no solid directly underneath the corner.

In our current system, there are two way in which pool voxels can become unsupported. A solid can disappear from the database, as in a valve opening, or pool motion can propel fluid into an unsupported region.

Unlike the transformation from drop to pool, the pool-to-drop transformation is an instantaneous one. As soon as a pool portion is discovered to be in an inappropriate situation, that portion is transformed. A further difference between the two transformations is that usually only a portion of the pool will transform into a drop. Pools are normally large entities. Presumably only a small portion of a pool will be in an inappropriate situation. The decision whether to transform to drop form is therefore done on a voxel-by-voxel basis.

If a pool voxel corner is unsupported, the entire fluid volume in the pool voxel will be turned into a drop. The pool voxel is removed from the pool structure. Figure

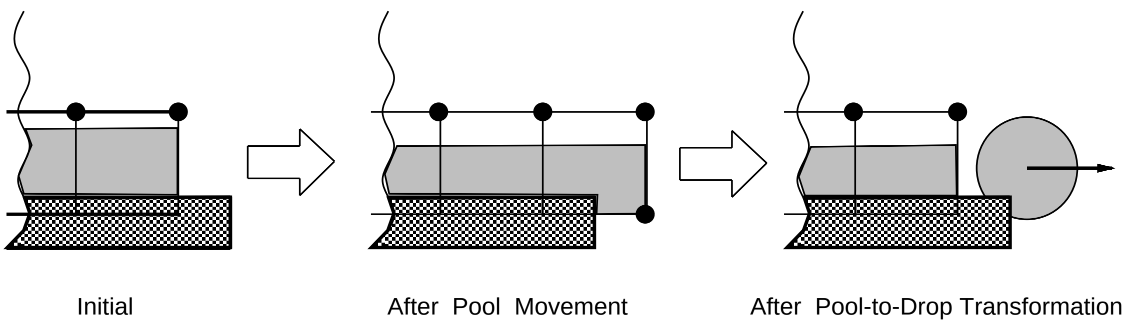


Figure 5.3 Pool Voxel Transformation Causes Cyclic Pattern

5.3 shows a sequence where a spreading pool flows over an edge. The pool voxels are depicted as squares. A solid is drawn as a hatched region. Initially, the pool voxels are as the leftmost picture shows. Pool motion creates the situation in the middle. The pool surface is generated based on the middle situation, as drawn in this figure. Since the rightmost part of the newly-created voxel is unsupported, a drop would be formed. The position and velocity of the drop are determined as described below. The rightmost pool voxel is destroyed when its fluid is converted to a drop, recreating the initial situation. The pool volume and height will probably change, but the voxels that compose the pool are the same at the beginning and end of the time step.

5.2.2 Mechanics of Pool-to-Drop Transformation

The transformation from pool to drop proceeds through several steps. Once transformation is determined to be necessary, a drop is created. The drop volume, position, and velocity must be determined.

Normally, the entire volume in a pool voxel will be converted into a drop. As mentioned in Chapter 3, the minimum drop size is one-eighth of a voxel volume. If the voxel holds less than that amount of fluid, the minimum sized drop is created. If the pool volume is less than the minimum drop volume, the pool volume will be set to zero. Volume is not strictly preserved in that case, but the effect should be negligible.

It is also advantageous to specify a minimum drop size when reducing a pool's volume. Consider a shallow pool on a flat tabletop. As drops are formed and the pool height lowers, the volume in each pool voxel gets smaller and smaller. Without a minimum drop size, the pool would become infinitely thin, but never disappear.

Several techniques are used to prevent drop creation from being too regular. For instance, if an overflowing pool creates drops with the same position, size, and speed at each time step, the effect looks unnatural. The fluid surface created in that situation is too regular.

Drop position is determined primarily by the pattern of "wet" corners in the pool voxel. A simple geometric average of the wet corners centers the drop within the convex hull of the voxel fluid. Weighting the coordinates, as is done in computing spline curves, positions the drop within the convex hull of the wet corners. We first compute a random location within the convex hull as described here and test that the location is visible from each of the wet corners. If it is not, we try again using the

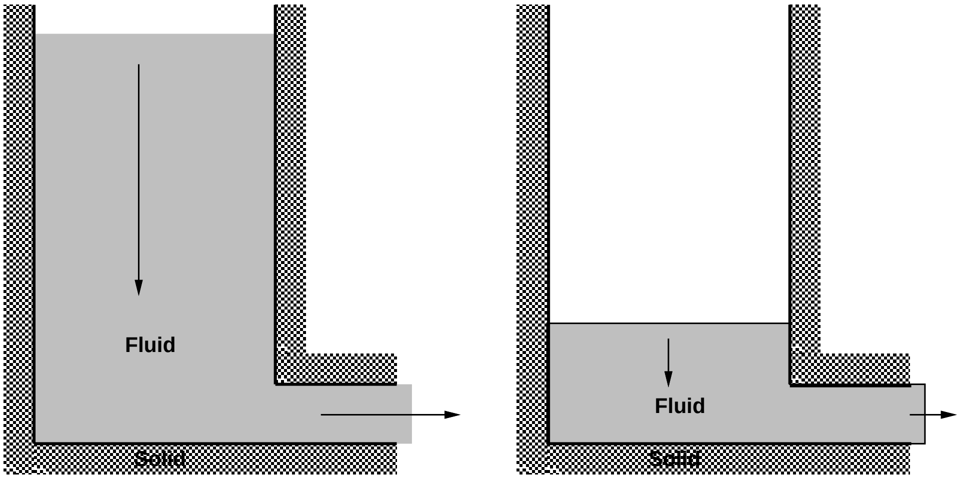


Figure 5.4 Pressure Determines Initial Drop Velocity

central point determined by equally weighting the wet corner locations. If that point is not visible from all wet corners, we use a wet corner location, picked randomly.

Initial drop velocity depends on the pool geometry. The direction depends on the location of unsupported corners. The sum of the vectors from the center of the voxel to the unsupported corners defines the initial drop velocity direction. The magnitude of the drop velocity depends on the difference between pool height and the drop position. In an open container, pressure at a point in the fluid is determined by the weight of fluid above it. Figure 5.4 shows a tank of fluid draining through a valve. On the left, the tank is nearly full, generating the velocity vector shown. On the right, as the tank is nearly empty, the pressure is less, pushing the fluid out at a slower rate. We use a linear relationship between the initial drop speed and the difference in pool height and drop position.

Drop creation is modeled as a continuous process, with creation times evenly distributed within a single timestep. Each new drop is therefore moved for a portion of a timestep after creation. This technique prevents the regular creation of drops, which causes a regular pulsing motion for overflowing pools.

To maintain system fluid volume, the drop volume is subtracted from the pool volume. If the pool volume becomes zero, the pool is deleted.

Chapter 6

Control Parameters

We agree heartily with the words of John Platt: "(You) need both simulation and control to create animation that does what you want." "...you want to guide models. You don't want just a pure simulation. You want to be able to specify some amount of control and then let the rest be automated."[Pla89] One reason physical simulations were not popular until recently is that a purely physical simulation would not let the animator create the effect he wanted[Bar89]. A useful system for fluid animation needs a physical basis, but should also allow the animator to control motion when necessary. Our system has numerous user-defined control values that affect fluid movement and appearance.

The twin goals of control and automated fluid motion are in conflict. Sometimes, an animator will want stringent control of some aspect of the scene. At other times, a straightforward hands-off simulation is desired. Consider an animation where a character is supposed to catch some falling fluid in a bucket. Even if such a sequence is physically realizable, it is unlikely an animator could discover the correct bucket motion needed for the task. In this instance, the animator would need much control to get the desired effect. We provide a set of controls that can allow detailed control, but we assume that normal usage will be to specify most parameters once per animation.

The key difference between solids and liquids is that solids are usually rigid but fluids are not. This is the major reason that solids are much more prevalent in computer graphics works. Their shapes are static and easy to define. The atoms in fluids are not connected to each other in static patterns as are atoms in solids. Since the atoms in fluids can move about each other, fluids can flow and change shape. A net force on fluids causes continuous deformation of the fluid.

The relative ease with which various portions of fluid can slide over others is embodied in the fluid's *viscosity*. Consider Figure 6.1. The right hand side shows fluid between a pair of parallel plates. The top plate is moving to the right. This causes stress τ on the adjacent fluid, tangent to the fluid surface, a *shear stress*. The stress causes the upper layer of fluid to move right at the same velocity as the

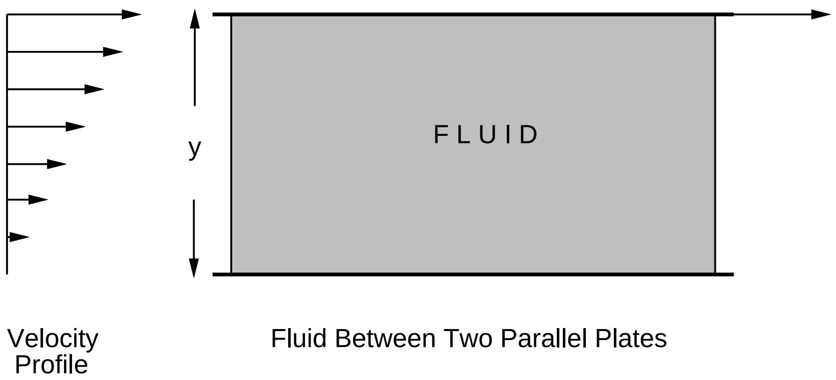


Figure 6.1 Viscosity defined by Velocity Profile

upper plate. For many fluids, the fluid velocity varies linearly in the **y** direction. The viscosity μ is defined by the equation

$$\tau = -\mu \frac{dv}{dy}$$

For small values of μ, the shearing force required to create a given velocity profile is small. It is easy to make layers of fluid move over each other. For thicker, more viscous, fluids the required force is larger. Many of the control parameters change the apparent viscosity of fluids by changing the shape or motion characteristics.

The gravity vector affects drop acceleration. Gravity is the primary source of kinetic energy for fluid in the system. The usual gravity vector is $(0, 0, -9.8)$ meters per second per second. A weaker gravity vector will make drops appear buoyant because they will fall more slowly. A smaller gravity vector will also add less energy to the system, causing the fluid to move more slowly. A stronger gravity vector will create faster-moving fluid. Changing the direction of the gravity vector is equivalent to tilting the whole scene or the addition of a space-invariant wind vector.

External forces such as wind can be added to influence drop motion. In combination with the gravity vector, these forces modify the motion of drops. In general, the external forces will be a function of position and time. These forces might be the result of a Computational Fluid Dynamics (CFD) simulation, a static definition, or a simple function of space or time. For example, wind vectors might be computed from a computational model, declared as a constant vector, or a time- and space-varying procedurally defined value.

6.1 Drop Controls

Several parameters are available to change the characteristics of moving drops. The parameters all relate in some way to the apparent viscosity. A default value for each parameter can be specified for the whole system. Currently, drops created by a user-specified source all have parameters set on a drop by drop basis. Drops formed from pools have the default parameter values.

The Kinetic Energy Factor (KEF) determines how elastic a collision of a drop and a solid is. A KEF value of 0.0 is totally inelastic, while a KEF of 1.0 creates a perfectly elastic collision. KEF determines how fast a drop will lose its kinetic energy, and so greatly affects the liveliness of the fluid motion. The transformation of a drop into a pool occurs after the drop is stationary. KEF determines the speed at which moving drops will slow to a stop, which is a prerequisite for transforming into a pool. Simulations like the waterfall done by Sims [Sim88] can be done by setting KEF to 1.0. Drops will never slow down, and so will never change representations.

Two parameters influence the amount of chaos caused by drops colliding with solids. The Bouncing Factor (BF) parameter affects the angle at which drops rebound off solids. BF is the drop change in velocity in the direction of the surface normal. A BF of 2.0 exactly reverses the drop velocity in the normal direction, as in a perfectly elastic collision. A BF of 1.0 would exactly cancel the velocity in the normal direction, and so would make the drop skim along the surface. SPLAY affects the drop motion in the plane of the solid. It is the angle over which the rebound path can vary from the ideal ballistic motion path. Larger values of SPLAY cause moving drops to spread out more quickly. Smaller SPLAY values make drops look more viscous.

The "Time To Flatten" (TTF) parameter controls the apparent viscosity of supported fluid. The drop structure maintains a record of how long a drop has been sitting or moving on top of some supporting solid. This time modifies the drop field function by maximizing the field function in the direction of the solid's surface normal. The effect is to flatten a drop out as time progresses (Figure 3.6). The TTF parameter controls the speed of flattening. TTF is the time required for a drop to flatten completely. Currently a drop will flatten until its height is one fifth of its radius. A short TTF (for example 0.3 seconds) causes drops to flatten quickly, as water or oil would. A longer TTF makes fluid look thick.

The user specifies how long a non-moving drop will remain a drop. The drop-to-pool transformation time (D2P) is the duration that a drop will sit idle on a surface. A long D2P would mimic drops where surface tension kept the drop as a bubble for extended periods. A long D2P would work best for modeling water beading on a car. Since pools will spread as soon as they form, the transformation from a drop to a pool models the point where surface tension can no longer hold a drop together. A short D2P makes supported drops appear to have little surface tension.

6.2 Sources and Sinks

Fluid sources allow explicit addition of fluid to the system. Sources provide a controlled entrance mechanism for bringing new fluid into the system. Since sources create new drops, they must specify all the characteristics of the drops.

Each source has a user-specified probability to create fluid during each time step. A low probability value might be used for a dripping faucet, for example. A high probability in the same situation could simulate a running faucet.

Sources specify a location and velocity for each newly-created drop. The location is specified as a central location and a possibly non-zero distance in each axial direction, providing a rectangular region of space. The exact starting position of a new drop is determined by three random numbers chosen at drop creation time. The initial velocity of a new drop is similarly specified as a central vector and a maximal variation in each axis direction. When the drop is created, three random numbers between -1.0 and 1.0 are multiplied with the variation vectors. The initial drop velocity is the vector sum of the central vector and these scaled variation vectors. The exact time of drop creation is randomly specified within a single timestep. Newly created drops therefore are moved some fraction of a single timestep immediately after creation, using the same procedures as normal drop motion.

Drop mass is specified as a minimum value and a range. The size of individual drops directly affects the bumpiness of the fluid surface. A collection of one hundred small drops in a random pattern will give a much more complex surface than a single large drop with the same volume. Multiple sources at the same location will produce more complex surfaces than a single source. Surface texturing techniques can add much of the same apparent complexity to the smoother surface.

Fluid sources have user-specified starting and ending times. The user controls the duration of each source. By staggering the start and end times, a gradual change in flow volume can be achieved.

Fluid sinks, destroyers of fluid, provide a controlled method for removing fluid from the system. They are specified similarly to sources, but are simpler because they do not specify parameters for the fluid they consume. Sinks have a user-specified chance of being invoked during each time step. The amount of fluid destroyed is specified as a minimum amount and a variation. The exact amount that a sink can destroy in a single timestep is recalculated each time the sink is active. Sinks have a position but are considered to occupy an entire voxel. Even if active, a sink's operation depends on the presence of fluid in its voxel. If the sink is active, drops in the voxel are examined one by one, and deleted while their summed masses are less than the mass to be deleted. One drop may have its mass merely reduced to make the mass loss correct. If necessary, a pool in the voxel is reduced by any remaining mass to be deleted.

Sinks have a user-specified lifetime, specified as a starting and ending time. As with sources, sinks starting and end times can be staggered to create smoothly varying outflows. If sources and sink volumes are matched, the amount of fluid in a system can be stabilized.

6.3 Pool Motion Control

The speed of pool motion is determined by the pool height, much as discussed in Chapter 5 for created drops. The weight of fluid creates a shear force that causes it to flow. The higher the uppermost pool level above a point, the more pressure causes motion. Prior discussion of pool motion neglected this effect. In reality, we use the pool height to determine the chance that motion will occur at each potential pvox corner. The animator has two variables to control pool motion speed.

First, the user specifies a minimum height for pool motion. Surface tension counteracts the shear forces when fluid is thin. Otherwise, fluid would flatten to a single molecule thickness. The chance of pool movement is zero for points within this distance of the pool height.

Secondly, the animator can control the height difference needed to ensure motion from one pvox corner to another at each time step. This height difference will cause the maximum speed. The pool will move at one voxel length during each timestep.

Chapter 7

Conclusions

This research originally started with dissatisfaction with the limited use of fluid in computer graphics animations. Although several impressive works existed, most animations make no use of fluid, despite it being ubiquitous in everyday life.

We began by researching previous fluid animation systems and the effects they portrayed. Categorizing previous work, we discovered two distinct implementation styles, each with its own strengths. Fluid was characterized as motionless with wave phenomena, or in constant motion. The amount of motion seemed to be the key differentiating quality. Each animator picked a limited subset of fluid characteristics to portray, then used a representation geared towards those characteristics.

Further examination revealed a more general differentiation between the categories. Surface effects were used where a supporting container could be assumed. Particle systems assumed the lack of such support. The realization of these differences provided a guiding principle in developing our system.

We originally proposed a dual representation system for modeling fluid of both types. Each representation used field functions to define fluid surfaces. A drop representation consisted of a particle system of field generators. A pool composed of a field linear in the Z direction. It was assumed that a combination of the two field types would produce a single fluid surface for the entire system.

During implementation of this system, it became obvious that unlimited pool field functions would cause undesirable motion in simple situations, as described in section 4.3.3. We switched to a limited volume field function, and used a simpler surface technique.

We use previous techniques where applicable. Several techniques from previous work are incorporated into our system. A particle system models chaotic fluid. Gravity and solids guide the particle motion. Time-varying waves are superimposed on planar surfaces.

Our initial implementation proves soundness of the thesis. We have implemented a physically based simulation of flowing fluid that allows animating previously impossible scenes. An animator can now include moving fluids into computer graphics animations, using machines of modest cost. The source is about 15,000 lines of *C* code. Sequences containing 40,000 voxels, 2,000 polygons, and up to 10,000 drops take 1 to 15 minutes per frame to compute. Several test animations have been produced.

7.1 Contributions of the Dissertation

7.1.1 Key Ideas From Previous Work

Previous research in the area of fluid animation provided several key ideas to build upon. The primary idea is to use physics to control the fluid motion. Fournier and Reeves[FR86], Peachey[Pea86], Sims[Sim90], Miller and Pearce[MP89], and Kass and Miller[KM90a] all use the physical environment to modify fluid motion. Their work shows that using physical models, even crude models, produces better results than animator-controlled fluid motion.

Other research has also shown that physical models alone are insufficient for animation for several reasons. First, simple models cannot create correct simulations of complex phenomena. The same assumptions used to simplify the model make it inadequate in some situations. Secondly, animators sometimes want unrealistic fluid motion. A correct physical model could not handle this task. Thirdly, even if the fluid motion is physically possible, the animator may be unable to specify the correct starting conditions to cause the desired result. For these reasons, extra controls are necessary to make a system usable.

Kass and Miller's system preserves correct fluid volume. Most of the other previous work does not model fluid volume at all. Volume is necessary when modeling filling and overflowing, and adds to realism for all fluid representations. We extended volume preservation to the entire system.

Wave phenomena have been extensively studied [Max81, Per85, MWM87, TB87, FR86, Pea86, KM90a]. The visual effect of surface waves provides a strong impression of fluidity, and therefore must be included in any representation. We include wave effects based on previous results.

7.1.2 New Ideas

The key contribution of this work is that it unites the two major fluid animation "camps". Previous work can be divided into one of two disjoint methodologies. This work provides a method encompassing both representation classes The resulting system models fluid in a wider domain than previously possible.

In *Red's Dream*[Pix87], Reeves used a particle system for falling raindrops, and non-moving planar puddles. These systems were unconnected. The puddles had their own waves that were generated without using the drops. The drops do not interact with either the solids or the puddles in the scene. In *Splash Dance*[KM90b], Kass and Miller have a particle system of raindrops that fall on a beach. Where the drops collide with the beach, fluid height is added to their grid. Drops are not described in the paper describing the system[KM90a], which concerns their pool movement code. Drops are used as a source of fluid, but are otherwise not central to their work.

Transitions between the representations allow using the most appropriate representation at each position at each time. Automating the transitions frees the animator from the need to choose the best representation by hand, an overwhelming task. Unsupported pools portions are converted to drops. Non-moving drops transform to pool form. Drops below a user-specified size will simply bead, modeling surface tension.

Our pool implementation itself, while part of a larger system, is a new contribution to fluid animation. The three-dimensional pool data structure creates a more capable representation than the previous height-fields. The pool has a true volume because it is an enclosed region. Fluid transport is automatic, including splitting and merging of pools.

We introduce several additions to the idea of using a particle system of drops to model fluid in motion. We are the first to maintain correct volume of a particle system by varying the level set that defines the fluid surface. Previous systems do not calculate a volume, or have no volume. We also use solids as a barrier for the field function, a new idea that makes containers practical. Previous systems required calculating an offset surface for all containers, or required containers to be thicker than the largest fluid droplet. The concept of barriers used with field functions is applicable to other surface design systems, not just fluid animation.

Modifying the field function to flatten drops on surfaces is a new technique that adds realism to fluid animation. Previous systems where drops had volume maintained spherical drops throughout. The best approximation came when drops were half enclosed by solids, producing hemispherical drops, although the volume was incorrect. Our system allows correct volume as well as more realistic flattening. Apparent viscosity is lessened by flattening the drops more, making them less spherical.

7.1.3 Other Observations

Physically-based modeling is not a panacea. Sometimes animators do not want correct motion. Reality is only a starting point for many animators. Controls must be provided to allow non-realistic motion when desired. Even when correct motion is the goal, it may not be obvious to an animator exactly what initial conditions will produce the desired effect. Controls can be provided in at least two forms: the ability to modify physical parameters and the ability to add elements that are free from the physical system. The animator might want fluid to fall more quickly that gravity would allow. He could either change the gravity acceleration vector, or manually add some faster-moving fluid to the scene description. In either case, additional controls would be necessary for the system to provide the desired effect. This desire for control is one reason physical models have taken so long to become popular.

We discovered blobby molecule techniques make fluids appear thick. This is partially explained by the simplicity of the underlying data. Isosurfaces created with our technique have a complexity that corresponds to the number of drops. The number of convexities can be no more than the number of drops. For speed, the animator wants to use as few drops as possible. Using many drops improves surface complexity, so these goals oppose each other. Traditional methods for increasing surface complexity, texture and bump mapping, should alleviate the problem.

Careful consideration of control parameters is needed to prevent blobby molecule motion from appearing sluggish. The drop bounce parameters, which determine drop speed and direction, control the perceived thickness of moving drops. Fluid that spreads out immediately on collision with solids looks like water. Fluid that bounces high looks thick and rubbery. The apparent viscosity of motionless drops depends on how quickly and how far a drop flattens out, both user-defined in our system. Drops that remain spherical have apparent high surface tension, which looks unrealistic unless the drops are small.

During the experimentation process, it soon became apparent that the ability to quickly run through a simulation was valuable. The process of creating several hundred frames of animation can be time consuming. A problem occurs when the simulation doesn't produce the desired results. A perfectly running simulation can surprise the animator with correct, but unexpected results. For instance, a stream of fluid is incorrectly specified and doesn't fill the desired container. We found running the simulation generating minimal surface data to be a prudent idea. Simple errors were caught before wasting a day of computing time.

Storing the system state was invaluable for both debugging and production purposes. We wrote the system state to disk for each timestep, originally as a debugging measure. As the implementation grew more complete, the written data became useful for restarting animations at key points.

Previewing requires viewing animations at their intended speed. We found that previewing a sequence a single frame at a time was insufficient for animations. Even previewing scenes at a factor of two slower than the final output speed was not good enough. Although this discovery is probably a truism for all animations, it is especially true for fluid studies because the perceived fluid characteristics are highly dependent on speed. Mere playing an animation at half speed makes fluid look viscous.

7.2 Future Work

Although this project proved the concept of a dual representation, much more work should be done. The implementation could be improved in a number of ways. Many of these might require more calculations, so experimentation would be necessary to decide whether each of the following would be a good idea.

One area of interest would be to look at pool height as a localized characteristic. The current single height for a pool has several drawbacks, as discussed in Chapter 4. Perhaps a modification of the techniques of Kass and Miller would prevent instantaneous height changes happening over arbitrary distances. Their technique could be modified to allow less regular connections of fluid columns. Unfortunately, changing the applicable situations would destroy a key feature of their implementation: the ability to encode their equations in a tri-diagonal matrix. The resulting system would be slower that Kass and Miller's but more powerful. Experimentation alone would determine whether the tradeoff was worthwhile.

Encoding a partial wetness value for pvox corners might allow for partially filled pvoxes. Our current implementation only has a single bit to represent whether fluid is present at each corner, with a corresponding wet/dry state for the pvox. Using a more complex specification for wetness would allow finer specification of fluid location within a pvox. It might be possible to use these pvox corners as weighted control points for parametrically defined-surfaces. The added benefit would be smoother control of pool fluid, which currently depends on the voxel grid dimensions.

The current implementation attempts no drop-to-drop interaction. Intersecting streams do not affect each other. The naive $O(N^2)$ approach would be prohibitively complex. Luckily the influence of a particle is limited by the particle size, so not every particle must interact with all others. Miller grouped globules (particles) by location to lessen the complexity to a claimed $O(N)$. Spatially grouping particles should help the complexity, but could still be prohibitively expensive. We feel a limited interaction between drops could be added without serious degradation, but further experimentation is needed to answer decisively.

Currently no solids move in our system. Obviously solid motion is desirable for animation. There are several problems to be solved before moving solids could be incorporated. A method for motion description would have to be added to the file format, or a controlling program could handle solid motion between steps in the animation. Solids would have to be grouped, so that collections of polygons could move together. Some notion of the containing entity would have to be added to pools, so that they could be affected by their containers' motion. The motion might modify the gravity vector for the contained pool.

Pool translation involves taking the known pvox corners, moving them by translation matrix to get new locations. Line-of-sight calculations are performed from the rotated old wet pvox corners to the new pvox corners.

Drops in the path of swept solids would have to be detected, and moved and accelerated accordingly. This is the dual of the line-of-sight calculation, where normally a drop moves and the question is whether the path it takes intersects a solid. Moving the drops in the reverse movement should allow the same test to be used in this case.

In a practical vein, our system should produce input for other rendering systems. We have largely ignored lighting effects since our focus is on producing motion. The fluid animation will not be complete until well-known rendering techniques are combined with our motion studies. Lighting effects like transparency, refraction, reflection, and caustics would add much to the realism of our technique.

Bibliography

[Bar89] Alan Barr. Physically-based modeling: Past, present, and future. *Computer Graphics*, 23(5):193, December 1989.

[Bli82] Jim Blinn. A generalization of algebraic surface drawing. *ACM Transactions on Graphics*, 1(3):235–256, July 1982.

[Blo88] Jules Bloomenthal. Polygonalization of implicit surfaces. *Computer Aided Geometric Design*, 5:341–355, 1988.

[FR86] Alain Fournier and Bill Reeves. A simple model of ocean waves. *IEEE Computer Graphics and Applications*, 20(4):75–84, August 1986.

[FTI86] Akira Fujimoto, Takayuki Tanaka, and Kansel Iwata. Arts: Accelerated ray-tracing system. *IEEE Computer Graphics and Applications*, 20(2):16–26, April 1986.

[Gla89] Andrew Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, 1989.

[KM90a] Michael Kass and Gavin Miller. Rapid, stable fluid dynamics for computer graphics. *Computer Graphics*, 24(4):49–57, August 1990.

[KM90b] Michael Kass and Gavin Miller. Splash dance [videotape]. *SIGGRAPH Video Review*, 61, 1990. segment 2.

[LC87] W. Lorenson and H. Cline. Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics*, 21(4):163–169, August 1987.

[Max81] Nelson Max. Vectorized procedural models for natural terrain: Waves and islands in the sunset. *Computer Graphics*, 15(3):317–324, August 1981.

[Mil87] Gavin Miller. Upwardly mobile [videotape]. *SIGGRAPH Video Review*, 38/39, 1987. segment 43.

[MP89] Gavin Miller and Andrew Pearce. Globular dynamics: A connected particle system for animating viscous fluids. *Computers and Graphics*, 13(3):305–309, Mid-Summer 1989.

[MWM87] Gary Mastin, Peter Waterberg, and John Mareda. Fourier synthesis of ocean scenes. *IEEE Computer Graphics and Applications*, 7(3):16–23, March 1987.

[NKON90] Eihachiro Nakamae, Kazufumi Kaneda, Takashi Okamoto, and Tomoyuki Nishita. A lighting model aiming at drive simulators. *Computer Graphics*, 24(4):395–404, August 1990.

[oC87] University of Calgary. Soft II [videotape]. *SIGGRAPH Video Review*, 24, 1987. segment 20.

[Pea86] Darwyn Peachey. Modeling waves and surf. *Computer Graphics*, 20(4):65–74, August 1986.

[Per85] Ken Perlin. An image synthesizer. *Computer Graphics*, 19(3):287–296, July 1985.

[Pix87] Pixar. Red's dream [videotape]. *SIGGRAPH Video Review*, 36, 1987. segment 3.

[Pla89] John Platt. Physically-based modeling: Past, present, and future. *Computer Graphics*, 23(5):192, December 1989.

[Ree83] Bill Reeves. Particle systems- a technique for modeling a class of fuzzy objects. *ACM Transactions on Graphics*, 2(2):91–108, April 1983.

[Sch80] Bruce Schacter. Long crested wave models. *Computer Graphics and Image Processing*, 12(2):187–201, February 1980.

[Sim88] Karl Sims. Particle dreams [videotape]. *SIGGRAPH Video Review*, 38/39, 1988. segment 44.

[Sim90] Karl Sims. Particle animation and rendering using data parallel computation. *Computer Graphics*, 24(4):405–413, August 1990.

[TB87] Pauline Ts'o and Brian Barsky. Modeling and rendering waves: Wave-tracing using beta-splines and reflective and refractive texture mapping. *ACM Transactions on Graphics*, 6(3):191–214, July 1987.

[Ups86] Craig Upson. The visual simulation of amorphous phenomena. *The Visual Computer*, 2(5):321–326, September 1986.

[Wat90] Mark Watt. Light-water interaction using backward beam tracing. *Computer Graphics*, 24(4):377–385, August 1990.

[WMW86a] Brian Wyvill, Craig McPheeters, and Geoff Wyvill. Animating *soft* objects. *The Visual Computer*, 2(4):235–242, August 1986.

[WMW86b] Geoff Wyvill, Craig McPheeters, and Brian Wyvill. Data structure for *soft* objects. *The Visual Computer*, 2(4):227–234, August 1986.