

Computer Science 236 Final Project
Catchwater the Game
Interactive Particle-Based Fluid Simulation
Modeling Water Flow Through Pipes

Ken Brooks and Brian Clipp

May 9, 2006

1 Introduction

In this project we created an environment where the user can place pipe sections in a three dimensional grid and visualize rain water flowing through the pipes. This involved developing models of the various pipe sections, organizing the placement of pipes in the world, determining the interconnectivity of the pipe network and modeling the motion of water interacting with the pipe network.

2 The Game

The game of Catchwater takes place in a world where water is scarce, and rain is precious. The inhabitants are willing to go to whatever lengths it takes to get more than their share, including building fantastic edifices of pipes and water-catchers. To position your catcher above someone else's catcher, thus stealing some of his water supply, is a particular feat. He whose structure can collect enough water, in excess of his fair share by a set margin, wins the game. The tools provided in the game are catchers and pipe segments of a wide variety of shapes. The players are free to connect them in any combination and orientation that will do the job.

3 Modeling a Pipe Network

3.1 Graphics Challenges

The first graphics challenges were these:

1. The design of the pipe components themselves. Tees and other branched components could not simply be compositions of cylinders, because they need to look right when viewed through the interior. Interpenetrating cylinders would not serve.
2. The design of a user interface that gives the user full freedom to orient pipe sections.
3. Development of a data structure that keeps track of sections as placed and oriented
4. Development of a data structure that keeps track of the connectivity of the pipe structure, so that we can calculate where the water is flowing.

3.2 Modeling the Pipes

Pipe segments consist, fundamentally, of cylinders and pieces of cylinders. In the case of tee joints and other more complicated branched parts, those pieces must be cut carefully so as to leave an unobstructed interior. Now all of these could have been modelled easily enough with a constructive solid geometry modeller. Except for one detail: what is the

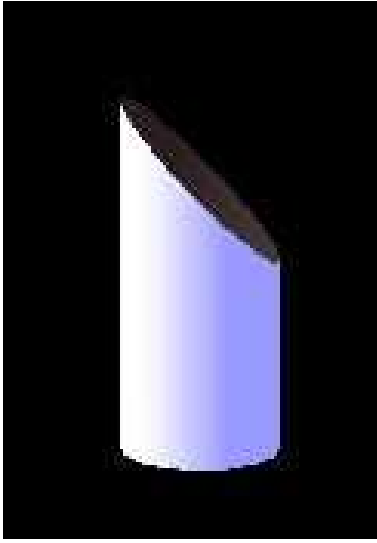


Figure 1: Bevel 1

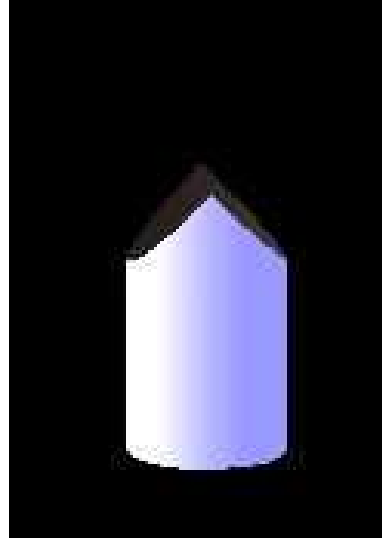


Figure 2: Bevel 2

shape of the water in a partially full pipe? It is not simple; it varies in a complex way involving trigonometric functions, according to the fraction of the pipe that is filled. For this reason we decided to model all parts algorithmically, rather than as stored data.

The fundamental unit of construction is the sliced cylinder. The elbow fig. 4, for example, is made up of two pieces (suitably rotated) that are sliced on the plane $X=Y$, fig. 1. The Tee joint fig. 5 uses these, and also a piece that is sliced on $Y=X$ and $Y=-X$, Fig. 2. A more complex part, the five-way connector fig. 6, uses four of a piece that is sliced on $Y=X$, $Y=-X$, and $Y=Z$, fig. 3.

These cylinder fragments are composed into whole pipe parts using OpenGL rotations. Upon consideration one can see that such pieces join exactly, leaving no gap and no overlap.

The catchers use a different method entirely. At its top rim, a catcher must be square, to entirely catch any rain falling in its column. At the bottom, it must be round, to join properly with a pipe. We model this by interpolation: in four sections per side of the square, 16 all together, a line is taken from a point on the square rim to the corresponding point on the circular base. Each adjacent pair of such lines defines

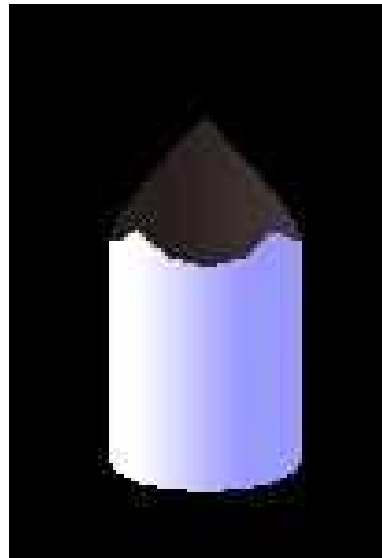


Figure 3: Bevel 3

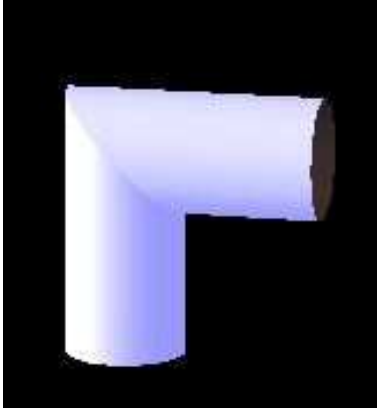


Figure 4: Elbow Joint

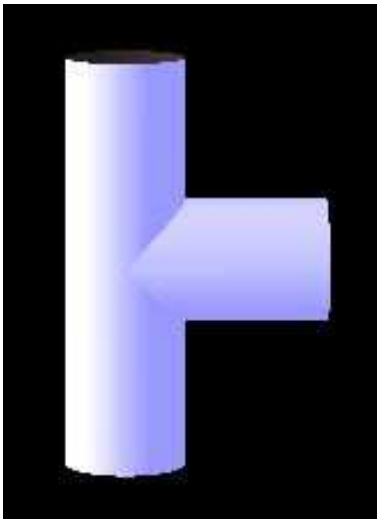


Figure 5: Tee Joint

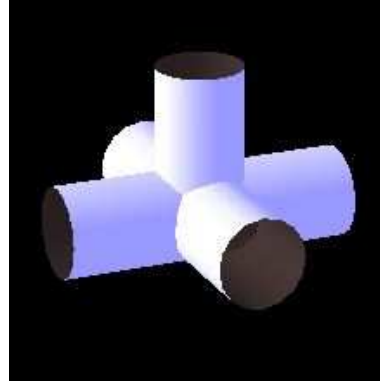


Figure 6: Five Way Connector

a sloping strip. In the case of the horizontal catcher `picture`, instead of a circle, the round end of the interpolation is the ellipse forming the cut edge where the pipe has been diagonally sliced.

The data points thus calculated, for sliced cylinders and for catchers, are passed to the particle simulator and define the water-stopping surfaces. For smoother and more accurate rendering the sloping strips in the catchers are further subdivided into four tiers.

3.3 User Interface

We began with the notion that old-style computer gamers made very efficient use of key groups IJKL or ESDF to move in a 2D world, and decided to try to extend it to 3D. So: when the user is placing a pipe, ESDF move the pipe in the plane parallel to the screen. In the interest of keeping a tight, efficient finger position, R moves the pipe away from the user, and W brings it out toward the user.

Pressing a "mode" key changes from moving to rotation mode. In this mode, the four core keys work in an intuitive way: F turns a front facing connector to the right, S turns it to the left; E turns a front facing connector to the top, D turns it to the bottom. R and W serve the remaining axis: R turns a top facing connector to the right, W to the left.

In either mode, pressing Enter commits the pipe to its current location and orientation, and ends a turn. The mouse is constantly available to adjust the user's

viewpoint, which proves vital for seeing clearly what one is doing.

Experience: using the keys for rotation as described seems fairly natural. For placement, ESDF are fine but R and W are lastingly non-intuitive. There does not seem to be any easy way to satisfy the conflicting requirements of quick finger access and intuitive placement. Room for further study.

3.4 Modeling the Structure

All pipe sections are aligned and oriented on a three-dimensional grid. But the problem of finding a pipe's neighbors is still non-trivial. There are 24 possible grid-aligned orientations of a cube: any of six faces could be facing you, and having chosen one of them, any of four remaining faces could be facing up. How to find the neighbors of an oriented pipe section?

This is our solution: for each class of pipe part (straight, elbow, tee, etc.) we store a canonical "link table", having one entry for each connector on the part. Each table entry includes a unit vector indicating the direction of this connector, when in the canonical orientation. As the user rotates a pipe while placing it, a quaternion describing its orientation is maintained. When the user commits the pipe to its position, we transform the unit vector for each connector by this quaternion to get its direction as oriented, and store it in a "link entry" belonging to this pipe. Adding the transformed unit vector to our grid location gets the neighboring location.

Upon discovering a neighbor with a connector suitably facing our new connector, we establish a link. Each link entry includes a pointer to the linked neighbor, and a link index referring to the neighbor's backfacing link. So the pipe structure as a whole forms a doubly linked graph, with a node for each pipe.

3.5 Flow

At the user's request, the rain begins. The particle simulator emits particles falling from the sky. They gather in collectors and are funnelled into the pipe structure. But not quite: the particle simulator proves very inefficient at modelling horizontal flow

driven by water pressure. So instead we provide a transducer to an alternative flow simulation. At the base of each catcher is a transducer surface. Particles falling through such a surface are counted, and vanish. The count at each surface is averaged over time to provide a flow rate.

Flows are processed as follows: we consider a pipe segment that has an incoming flow. For any connector that does not have an incoming flow: if there is a connector pointing down, all flow goes to that connector. Otherwise if there are connectors pointing horizontally, the flow is divided evenly among them. Any flow reaching the base of the world terminates, and based on its location, is counted to the benefit of one player or another to determine the current score.

Any flow reaching an open, unattached connector is passed back to the particle simulator, providing a rate at which particles should be generated at that connector.

3.6 Seeing Water Inside the Pipes

To make the game visually appealing, we have intended from the beginning to provide a mode in which the pipes are transparent, with the flowing water visible within. Here lies another challenge: what is the shape of the water in a partially full pipe? And especially, what is the shape of the water in a partially full cylinder sliced on the diagonal in multiple planes?

This is the sliced cylinder algorithm, for a cylinder of radius r : subdivide into a number of sections n around the cylinder. Each section subtends angle $\phi = \frac{2\pi}{n}$.

For $i=0$ to n :

$$\theta = -\pi + phi$$

$$x = \cos(\theta)$$

$$z = \sin(\theta)$$

$$y = \text{cut}(x, z)$$

generate one vertex at $(x, -0.5, z)$, the clean end and one at (x, y, z) ,

the cut end

The function cut varies depending on the type of cut being modelled; some examples are:

- x for a single cutting plane $y=x$
- $\min(x, -x)$ for two cutting planes at $y=x$ and $y=-x$
- $\min(x, z)$ for two perpendicular cutting planes at $y=x$ and $y=z$

All sliced cylinders are modelled in one orientation: clean end at $-Y$, cut end at $+Y$, "down" at $+Z$. We have mentioned the link entries containing a unit vector for each connector in a situated pipe. Each link entry also contains a quaternion q . This quaternion expresses the orientation of this sliced cylinder, this arm of the pipe, relative the canonical orientation of the pipe itself. Each pipe stores a quaternion o representing its orientation relative to the world. We calculate $r = o * q$ representing the orientation of this sliced cylinder relative to the world. We use r to transform the three axes X , Y , and Z . And that determines how water will lie in the cylinder.

If Y transforms into $+Y$: water will sit at the clean end, forming a (possibly sliced) shorter cylinder.

modify the above algorithm by generating the cut-end vertex at $(x, \min(y, \text{level}), z)$

If Y transforms into $-Y$: water will sit at the cut end

modify the above algorithm by generating the clean-end vertex at $(x, \min(0.5, \text{level}), z)$

Otherwise the modified algorithm is more interesting:

Assuming that level ranges between $-r$ and r :

$$a = \arccos\left(\frac{\text{level}}{r}\right)$$

$$\phi = \frac{2a}{n}$$

For $i=0$ to n :

$$\theta = -a + \phi$$

$$x = \cos(\theta)$$

$$z = \sin(\theta)$$

$$y = \text{cut}(x, z)$$

generate one vertex at $(x, -0.5, z)$, the clean end and one at (x, y, z) , the cut end

And then regenerate the original pair of vertices one more time, to complete the upper surface of the water.

This can compute the cylinder with water lying against one side of it. But a horizontal cylinder could have any of four sides oriented down. Here is a solution for this:

Based on the orientation of the transformed X and Z axes, we compute the orientation angle b , which will be some multiple of $\pi/2$. Modify the computation of θ to:

$$\theta = -a + b + \phi$$

Then the orientation of the water surface will change relative to the orientation of the cutting planes, and the water is correctly shaped.

4 Fluid Simulation

Our approach to fluid simulation followed the direction of [1]. Their work is based upon modeling a fluid using a Lagrangian method called smooth particle hydrodynamics (SPH)[2]. Smoothed particle hydrodynamics was originally developed to model the interaction of multiple astronomical bodies such as planets, meteorites and stars. SPH assumes linearity in the interaction of the multiple particles and generates the total force on a particle as the sum of the forces due to interactions with all the other particles, each calculated separately.

Two methods are used primarily to model fluids, Euclidean simulations and Lagrangian simulations. In Euclidean simulations, the velocity field of the fluid is evaluated at points on a regular grid. This is the method that is used in most engineering computational fluid dynamics programs. For the Euclidean method to be effective it must be performed on an extremely fine grid which makes it computationally intense and too slow for interactive applications. Lagrangian simulations model a fluid as a set of discrete points moving through space, referred to here as particles. These points has attributes of mass, density and velocity. Each particle represents an certain, fixed mass of water distributed over a varying volume, which is represented by a changing density. Because the Lagrangian method only models the water where the water is and not over an entire grid, most of which might be empty, the Lagrangian method is more computationally attractive.

4.1 Smoothing Kernels

Smoothed particle dynamics uses three dimensional smoothing kernels to scale the influence of each particle depending on the distance the particle from the point where the influence is being calculated. Each kernel has a finite distance of support h , outside of which the kernel is zero. The kernels are normalized so that their volume integral over their volume of support is one. Smoothing kernels $W(r, h)$ are applied to interpolate a quantity A at location r using equation 1. In this equation, ρ_j is the density of particle j .

$$A(\mathbf{r}) = \sum_j \frac{A_j}{\rho_j} W(\mathbf{r} - \mathbf{r}_j, h) \quad (1)$$

One of the major advantages of SPH is that to interpolate the gradient or laplacian of a quantity one need only replace the smoothing kernel with the gradient or laplacian of the smoothing kernel as in equation 2.

$$A(\mathbf{r}) = \sum_j \frac{A_j}{\rho_j} \nabla W(\mathbf{r} - \mathbf{r}_j, h) \quad (2)$$

Kernels used in this project were developed by [1] to model the magnitude of the various effects in the simulation according to the type of effect. The reader is referred to [1] for the smoothing kernel equations and a discussion of their properties. Muller et al. developed three types of kernels, a general kernel for density and surface tension calculations which is approximately gaussian, a spiky kernel used in pressure calculations to model the effect of close proximity on increasing pressure rapidly and a third kernel which is applied for viscosity effects.

4.2 Fluid Mechanics

In Eulerian computational fluid dynamics, two equations are required to model the change in the fluid. The first, 3, ensures the conservation of mass in the simulation. The second, 4, is the familiar Navier-Stokes Equation, which ensures conservation of momentum. In this equation p is a pressure field, \mathbf{g} is an external force density field and μ is the viscosity of the fluid.

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \quad (3)$$

$$\rho \left(\frac{\partial \mathbf{v}}{\partial t} + \mathbf{v} \cdot \nabla \mathbf{v} \right) = -\nabla p + \rho \mathbf{g} + \mu \nabla^2 \mathbf{v} \quad (4)$$

In the lagrangian formulation conservation of mass is ensured because each particle is conserved and has a fixed mass, so equation 3 is unnecessary. In particle based simulations the left hand side of 4 can be replaced by the derivative dv/dt because the particles move with the velocity field and so the convective term $\mathbf{v} \cdot \nabla \mathbf{v}$ is unnecessary as well. We are left with 5.

$$\left(\rho \frac{d\mathbf{v}}{dt} \right) = -\nabla p + \rho \mathbf{g} + \mu \nabla^2 \mathbf{v} \quad (5)$$

Each of the addends on the right of 5 represents a force density field. The term $-\nabla p$ is a pressure force density field, $\rho \mathbf{g}$ is an external force density field and $\mu \nabla^2 \mathbf{v}$ is a viscosity force density field. The additional term $f_{surface}$ adds surface tension, which is not modeled in the Navier-Stokes formulation. Summing these force density fields calculated at each particle center and dividing by the particle density yields the acceleration of the particle.

$$\mathbf{a}_j = \frac{d\mathbf{v}_j}{dt} = \frac{-\nabla p_j + \rho_j \mathbf{g} + \mu \nabla^2 \mathbf{v}_j + f_{surface}}{\rho_j} \quad (6)$$

The following sections describe the calculation of the particle densities as well as pressure, external and viscosity force density fields. For a more extensive discussion of fluid mechanics the reader is referred to [3]

4.2.1 Particle Densities

All of the other calculated forces depend on the distribution of mass throughout the simulated space. The calculation of particle densities determines this distribution of mass. Densities are calculated using equation 7. In this equation m_j is the mass of particle j and $\rho(r)$ is the density at location r . Location r

is the center of each of the particles in each of the simulation update equations.

$$\rho(r) = \sum_j m_j W(\mathbf{r} - \mathbf{r}_j, h) \quad (7)$$

The various forces in the simulation are determined using the calculated densities.

4.3 Pressure Forces

As muller et. al. suggest, we use a symmetric form of the pressure equation 8. Symmetry is necessary for the application of SPH. In this equation p is the pressure at a given particle. It is calculated as $p = k\rho$ where $k = \frac{k_b T}{m}$ and k_b is Boltzman's constant, T is the temperature in Kelvin and m is the molecular weight of water. In this simulation temperature was held constant at 295K, approximate room temperature.

$$-\nabla p(\mathbf{r}_i) = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \nabla W(\mathbf{r}_i - \mathbf{r}_j, h) \quad (8)$$

It should be noted that none of the resulting terms on the right of 8 is a vector, but the pressure force is. To overcome this the pressure force was directed along a vector from \mathbf{r}_j to \mathbf{r}_i , which matches the repulsive nature of pressure forces.

4.4 External Forces

External force density fields were not used in the simulation. However, they could be added easily and are mentioned here only for completeness. In order to model gravity the acceleration due to gravity, g , is added to the acceleration calculated by from other forces.

4.5 Viscosity Forces

Viscosity forces are also symmetrized as was suggested in [1] yielding equation 9.

$$\mu \nabla^2 \mathbf{v}(\mathbf{r}_i) = \mu \sum_j m_j \frac{\mathbf{v}_j - \mathbf{v}_i}{\rho_j} \nabla^2 W(\mathbf{r}_i - \mathbf{r}_j, h) \quad (9)$$

4.6 Surface Tension Forces

The Navier-Stokes formulation leaves off an important aspect of fluid dynamics, namely surface tension. To model surface tension, a color field, c , is generated which is one at particle locations and zero elsewhere. This allows the calculation of the number of particles in a given direction relative to a particular particle.

$$c(\mathbf{r}_i) = \sum_j \frac{m_j}{\rho_j} W(\mathbf{r}_i - \mathbf{r}_j, h) \quad (10)$$

The gradient of the color field gives the surface normal and laplacian of the color field yields the curvature. Since surface tension increases near the edge of a fluid and is higher at areas of high curvature of the fluid surface the following equation 11 works well in practice. In this equation σ is a fluid viscosity term.

$$f_{surface} = -\sigma \nabla^2 c(\mathbf{r}_i) \frac{\nabla c(\mathbf{r}_i)}{|\nabla c(\mathbf{r}_i)|} \quad (11)$$

4.7 Simulation Cycle

In each step of the fluid simulation the following occur.

- Calculation of Particle Densities
- Calculation of Pressure Forces
- Calculation of Viscosity Forces
- Calculation of Surface Tension Forces
- Application of External Forces Including Gravity
- Calculation of New Particle Velocities and Positions
- Collision of Particles with Surfaces

4.7.1 New Particle Velocities and Positions

After the sum of forces on the particles is calculated using SPH, the new particle accelerations are found by dividing the force vector by the particle mass. These accelerations are then integrated over a

fixed time period dt and added to the current particle velocity vector, yielding the new particle velocities. These velocities are then integrated again over dt to determine the new particle positions before any surface interactions have taken place. Numerical instability was observed for values of dt of greater than 0.005ms. More sophisticated numerical integration techniques might be used to lessen this instability and allow larger time steps. In practice this time step did not limit the simulation's progress.

4.7.2 Surface Interactions

Surface interactions are handled by determining which surface, if any, was the closest surface to be passed through in the last time step. The intersection with the closest surface is calculated and the new velocity is calculated by reflecting the current velocity about the surface normal. The distance between the previous position of the particle and the new position is interpolated to find the intersection point with α being the fraction of the travel distance that the collision occurred. The remaining distance $(1 - \alpha)$ is multiplied by dt and the new velocity and added to the collision point to determine the point the particle would have bounced to off of the surface. During each collision a fraction of the particle's velocity is removed, modeling the energy loss to heat during a collision. In order to prevent particles from passing through surfaces due to numerical inaccuracy a small additional position offset in the normal direction is added to the final position of the particle. Finally, after each collision, a new collision must be calculated to allow that a particle which bounces off of one surface may hit another surface.

5 Implementation

Catchwater was implemented using OpenGL for rendering and using the CPU to calculate fluid flows. Further work has been done by [4] to implement SPH on the GPU. For our purposes of developing a game, the CPU implementation appears to work adequately. A first implementation of SPH was done calculating the interaction of all particles with each

other and testing collisions with all surfaces in the simulation. This proved to be woefully inefficient and did not take advantage of the spatial distribution of particles and surfaces and the fact that the smoothing kernels are zero magnitude outside of radius h . To take advantage of these facts, a grid structure with cells of size h on a side was imposed on the simulation environment. Doing this, each particle need only be interacted with the particles and surfaces within its own grid cell and the twenty-six surrounding grid cells. Another optimization was suggested in [1], which was to copy the particles into memory locations in the grid rather than using particle pointers. This at first seems inefficient, performing copies when particles move from one cell to another. However, the performance gain due to increased cache hits greatly outweighed the cost of the copies.

Further implementation work could be done to render the fluid as a surface, rather than a collection of droplets as was done in Catchwater. This might be done using surface splatting or marching cubes on the particles closest to the fluid surface as found by the color field magnitude and direction.

6 Results

Two screen shots are included of the operation of Catchwater, figures 7 and 8. In its current instantiation, the fluid simulation and interaction of particles with pipes is running. Further work needs to be done to create the scoring system and improve the user interface for ease of game play.

7 Conclusion

Particle based fluid simulation has proved to be an effective, if computationally intensive, method for simulating water flow in three dimensions. Future work on this project could include implementing the fluid simulation on the graphics processing unit to offload processing there and speed up the fluid simulation. This would allow for many more particles than are possible on the CPU. Additionally, an algorithm that recognizes when a group of particles are close enough

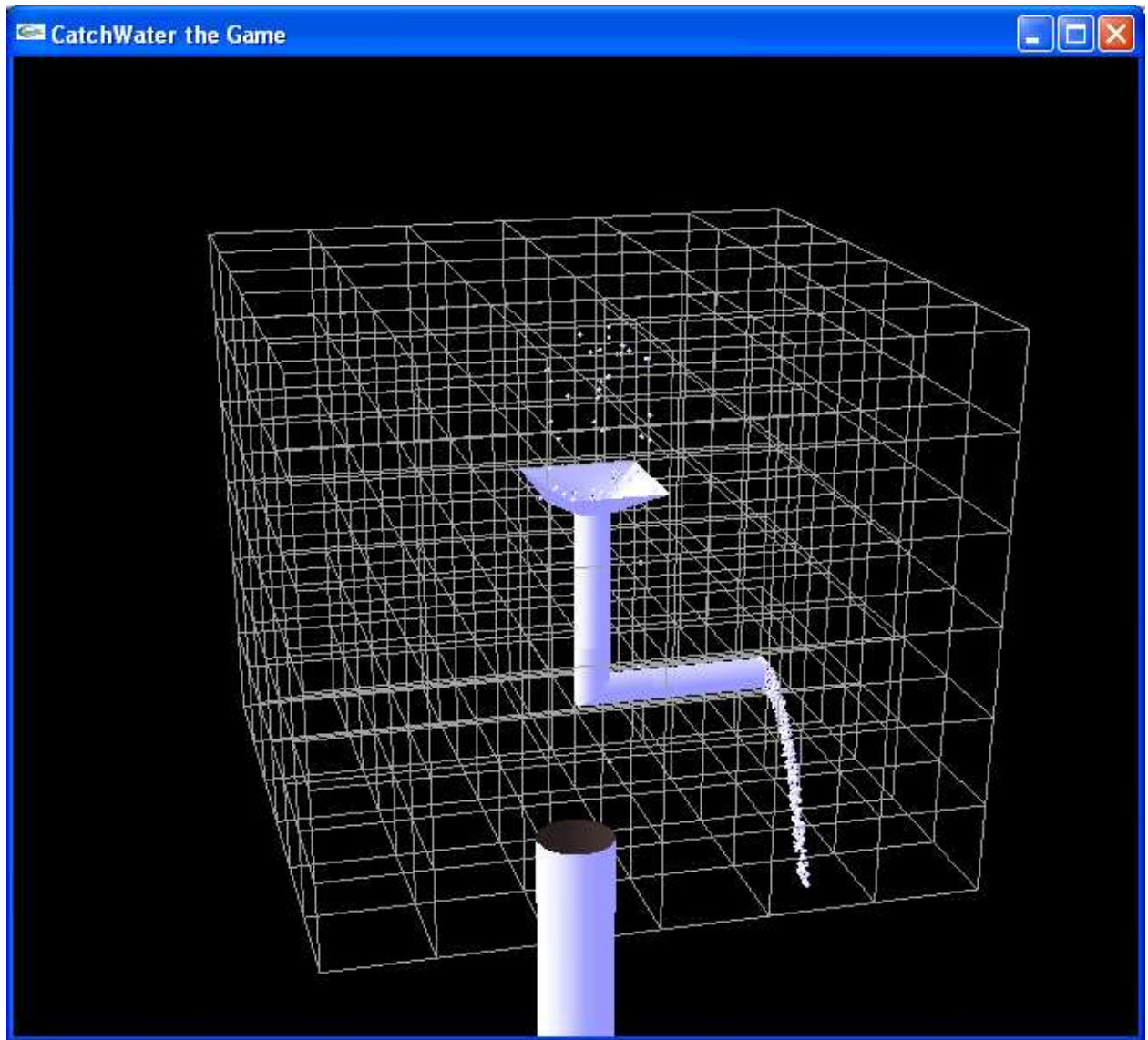


Figure 7: Configuration with input catcher and right angle pipe

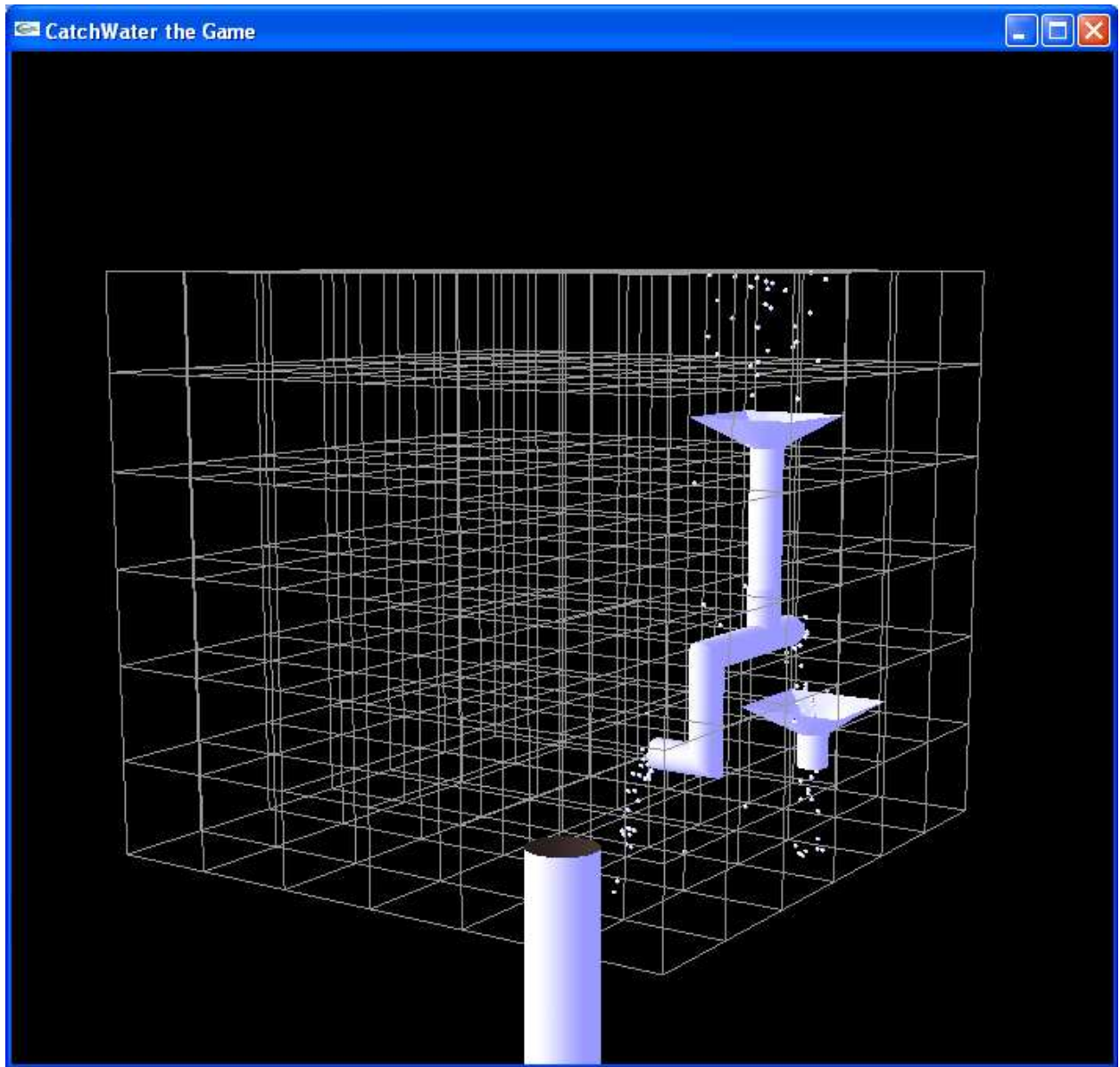


Figure 8: Configuration with one input catcher and two output pipes

together to form a united surface and renders that surface would greatly improve the appearance of the water.

References

- [1] C. D. Muller, Matthias and M. Gross, “Particle-based fluid simulation for interactive applications,” *Eurographics/SIGGRAPH Symposium on Computer Animation*, 2003.
- [2] R. A. Gingold and J. J. Monaghan, “Smoothed particle hydrodynamics: theory and application to non-spherical stars,” *Monthly Notices of the Royal Astronomical Society*, pp. 181:375,398, 1977.
- [3] D. Pnueli and C. Gutfinger, *Fluid Mechanics*. NY: Cambridge University Press.
- [4] A. Kolb and N. Cuntz, “Dynamic particle coupling for gpu-based fluid simulation,” *Proc. 18th Symposium on Simulation Technique*, 2005.