Water Simulation for Maya using Smoothed Particle Hydrodynamics

Caroline Gard, carga635
Kristin Bäck, kriba265

December 20, 2017

Abstract

The paper demonstrates a research about the simulation method Smoothed Particle Hydrodynamics, used for fluid simulation with properties such as pouring and splashing. The paper also includes description and results of a simulation project where the SPH method was implemented as a script for the 3D software Maya to simulate three dimensional fluid volumes. The project was developed during a three week period in the course SFX – Tricks of the Trade at Linköping University.

1 Introduction

Simulations of water has for long been a problem in the field of animation and simulations, and is still a huge challenge for research in the topic. Depending on the desired behavior of the fluid, different simulation methods is possibly applied to obtain the desired motion of the fluid. *Smoothed Particle Hydrodynamics* (SPH) is a possible implementation method to simulate interactive and realistic fluid dynamics, which generates accurate results when used properly. SPH is not only limited to simulations of water, it can be used for fluids in many different forms, such as rain, mud, as well as fog, smoke and foam.

Various methods can be implemented to generate fluid simulations, methods each suitable for fluids with different behaviors and requirements of special properties. In general, there are two basic methods for fluid simulation; grid-based simulations and particle-based simulations. The grid-based fluid implementations, has for long been a standard basic method, and sets all particles to be aligned on a continuous grid, to simulate fluid surfaces [1]. The particle-based methods, are in some cases considered as a better approach, compared to the grid-based methods. Particle-based methods uses particle simulation with interpolation to combine the particles as a constant volume. The main difference between these two general methods, is that grid-based methods only consider the particles on the water surface, but the particle simulations considers the whole fluid volume in the calculations. The SPH method is particle based and is discussed more in detail in the next section in this paper.

2 Smoothed Particle Hydrodynamics

To get a wider understanding of the SPH simulation method, the properties of the algorithms is necessary to be described in detail. This section contains a general description of the SPH method, including a description of associated forces in the algorithm. Additional, the section describes the SPH equation and the kernel function associated with the equations.

2.1 Method

Particle-based simulation methods have become important in the field of computer graphics, were the most popular methods nowadays are based on SPH. To simulate water splashes or breaking waves, the appropriate method to implement is particles-based approaches instead of using grid-based simulation methods. SPH is a *Lagrangian* based interpolation method to simulate fluids by approximating the values of the corresponding properties. The method can be used both for a two-dimensional and three-dimensional environment and uses particles with properties such as mass, velocity and position. The method allows additional entities based on the desired result, such entities could for example be temperature and applied physics.

To compute the physical properties of each particle for a time step, the method focuses on the neighbouring particles to the particle of interest. The properties of the neighboring particles are individually weighted using a weighting function, which depends on the distance between the particle and the neighbouring particle. This is described more in detail in the next section, section 2.2.

2.2 Smoothing Kernel

SPH computes forces from weighted contributions of neighbouring particles and the choice of weighting function is crucial for the stability, the accuracy and the speed for the particles. The weighting functions are called smoothing kernels and one commonly used is the Cubic Spline kernel [2]. Cubic Spline weights the closest neighbouring particles in contrary to other kernels which often have a small contribution from all particles regardless of distance. However, the Cubic Spline kernel is known for its low cost of computational time and is therefore to prior over the Gaussian which is computationally heavy. An alternative for Cubic Spline and Gaussian is to use a combination of three kernels for different force calculations. The combination is used in the project presented in this report and each and one of them is presented in section 2.3.

2.3 Forces

Equations for fluid simulations often includes advanced physical equations, where *Navier-Stokes Equation* is the basic equation for most fluid simulations. Thereafter, forces and new criteria are applied for specify the equation for suited purpose. Forces often considered in *Navier-Stokes Equation* for fluid simulation, is gravity, viscosity, pressure and surface tension. For the SPH method, density is the only required force to solve the SPH equation, while the other forces can be chosen depending on desired result. Gravity is a typically used external force, and is constant in downward direction. The gravity force of each particle, is based on the density of the mass multiplied by gravity. The volume of fluids is considered as consistent, which means that it is not possible to change the volume, which brings new criteria for the pressure unit. The pressure limits extension and compression of the fluid, and differences in pressure, will affect the velocity and the force of the pressure is the gradient of pressure.

Another property common in fluid simulations is viscosity, which is a physical property of fluids which can be described as a measure of friction in liquids. Thin liquids, as water for example, have a smaller value of viscosity, and thicker liquids as syrup have a greater value of viscosity. Surface tension is required for describing the changes in the forces closer to the water surface, due to smaller quantity of neighboring particles closer to the surface. The tension force is not yet considered in the implemented project though.

3 Equations

SPH is an interpolation method based on an integral using a fluid volume divided into particles for approximation [3]. The entire fluid is computed with the SPH equation for each particle one by one, by looking at the properties of the neighbouring function to the current particle, see equation1.

$$A_s(r) = \sum_j m_j \frac{A_j}{\rho_j} W(r - r_j, h) \tag{1}$$

where A is the chosen physical quantity, A_j describes the physical quantity for neighbouring particle j, m_j is the mass for particle j, p_j is the density for the neighbouring particle j, W is the weighting function where $r - r_j$ describes the distance between the current particle and the neighbouring particle and h is a constant for the kernel function. The equation is defined over the whole volume of the fluid and the force quantities are computed separately for every particle in the volume, where the physical properties are summed from all neighboring particles.

The properties of each particle are smoothed by a kernel function and is obtained by summing the neighboring particles placed within the smooth kernel radius. Each neighboring particle contribute to the sum of the current particle, by weighting the properties to the distance and density. The kernel function Cubic Spline is given by equation 2 below

$$W(r,h) = \frac{1}{\pi h^3} \begin{cases} 1 - \frac{3}{2}x^2 + \frac{3}{4}x^3 & 0 \leq x \leq 1 \\ \frac{1}{4}(2-x)^3 & 1 \leq x \leq 2 \\ 0 & 2 \leq x \end{cases} \tag{2}$$

The q in the equation is defined by $q(x) = \frac{\|x\|}{h}$, where x is the distance between the current particle and the neighbouring particle and h is the smooth kernel radius in which the neighbouring particle lies.

The basic functions for both SPH and the kernel function, used in the implemented project in this report, are further improved to add more stability and accuracy to the simulation [2]. The smoothing kernels vary with the different force quantities used in the SPH equation. The kernels are even and normalised with second order interpolation errors. Another feature of the improved kernel functions, is that they obtain the value of zero at the boundaries, which contributes to stability of the system. For calculating density, the kernel function called Poly6 is used, see equation 3. The most important feature of the Poly6 smoothing kernel is that r , which is the particle's position, only appears squared, which means that the distance to neighbouring particles can be computed without the requirement of square root computations.

$$W_{poly6}(r,h) = \frac{315}{64\pi h^9} \begin{cases} (h^2 - r^2)^3 & 0 \leq r \leq h \\ 0 & otherwise \end{cases} \tag{3}$$

When particles are approaching each other, the repulsive force for pressure vanish. This is because, when the distance between the particles is getting close to zero, the weighting function returns zero, which will set the total pressure force to zero. To solve the problem, the kernel function is improved for the pressure force and the new function is called Spiky, see equation 4. The Spiky kernel function has the feature of a non vanishing gradient when two particles intersects, which generates the necessary repulsion forces.

$$W_{spiky}(r,h) = \frac{15}{\pi h^6} \begin{cases} (h - r)^3 & 0 \leq r \leq h \\ 0 & otherwise \end{cases} \tag{4}$$

The viscosity force has a smoothing effect on the velocity field since the viscosity phenomena is caused by friction. Using the kernel functions Spiky or Poly6 for the viscosity calculations, would lead to negative values when two particles are close, which could lead to increasing forces for the velocity in an undesirable manner. A third kernel is therefore introduced and can be seen in equation 5. The Laplacian of the kernel is positive everywhere which improves the stability when the number of particles are low.

$$W_{viscosity}(r,h) = \frac{15}{2\pi h^3} \begin{cases} \frac{r^3}{2h^3} + \frac{r^2}{h^2} + \frac{h}{2r} - 1 & 0 \leq r \leq h \\ 0 & otherwise \end{cases} \tag{5}$$

The improved kernel functions replaces the Cubic Spline kernel for respective force in the SPH equation. The SPH equation is further improved for the physic quantities of pressure and viscosity, implemented in the project. After implemented the Poly6 kernel function to the density equation, its resulting SPH equation is shown in equation 6 below [3].

$$\rho(i) = \rho(r_i) = \sum_j m_j W_{poly6}(r_i - r_j, h) \tag{6}$$

where r is the position of the current particle i, m is the mass of the particle and j is the neighbouring particle. $h$is as described above, the smooth kernel radius. The SPH equation for pressure is improved as well, since the pressure force is not symmetric for two particles interacting. A simple solution to improve the stability and maintain the computation time is to sum the particles pressure forces, see equation 7 below, which also includes the Spiky kernel function

$$f_i^{pressure} = -\triangle p(r_i) = -\sum_j m_j \frac{p_i + p_j}{2\rho_j} \triangle W_{spiky}(r_i - r_j, h) \tag{7}$$

where p is computed by $p = k(\rho - \rho_0)$ and ρ is the density of the neighbouring particle j. A similar issue is related to the SPH equation for the viscosity force, with asymmetric forces due to changing velocities of interacting particles. The problem is solved by using the difference of the velocity between the particle of interest and the neighbouring particle , instead of using the absolute value of the velocity of the neighbouring particle, see equation 8, which includes the improved kernel function for velocity as well

$$f_i^{viscosity} = -\mu \triangle u(r_i) = \mu \sum_j m_j \frac{u_j + u_i}{\rho_j} \triangle^2 W_{viscosity}(r_i - r_j, h) \tag{8}$$

where u is the velocity of each particle and μ is the viscosity constant.
After computing the forces separately, all forces are summarized into one total force for the current particle, see equation 9 below

$$F_i = f_i^{pressure} + f_i^{viscosity} + f_i^{external} \tag{9}$$

The total force for the particle, is further used in combination with the density of the particle, calculated in equation 10, to compute the acceleration for the particle for the particular time step, to obtain the new position for the particle, see equation 10 below

$$a_i = \frac{F_i}{\rho_i} \tag{10}$$

4 Implementation and Result

The SPH equations are implemented in Maya to create water simulations. The implementation is presented in this section along with the result with different amount of particles.

4.1 Setting up the scene

The particles in the scene was build of sphere objects within Maya with the properties mass, initial velocity and initial position. An open box of glass material was added to the scene for the particles to interact with but also contain the visibility of the particles.

4.2 Maya Script

The implementation was done using the programming language Python and Maya.cmds which is a Python wrapper for MEL commands. MEL is the scripting language for maya used to access properties and functions within Maya for creating and handle objects and functions within the software.

The script includes lists to save the particles and their neighbours separately along with their positions. A particle is defined as neighbour to another particle if the distance between them is less or equal to h.

The position of the particles was used to calculate the density, pressure and the viscosity as described in section 3. The acceleration and new position of the particles were updated using the result of the force calculations. For every update of acceleration and position a new keyframe was created and the calculated forces was applied in order to create an animation of the simulation. The full implementation is presented in appendix A.

4.3 Result

The result of the implementation of SPH in Maya is particles falling from one specific height and landing into a glass box. Different amount of particles is shown in Figure 1 - 3 to demonstrate the various result at this stage.

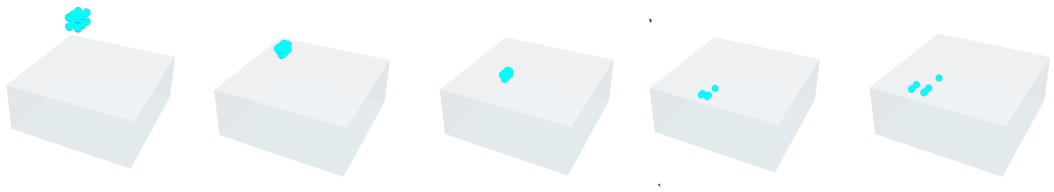

Figure 1: Fluid simulation using 12 particles.

Figure 2: Fluid simulation using 27 particles.

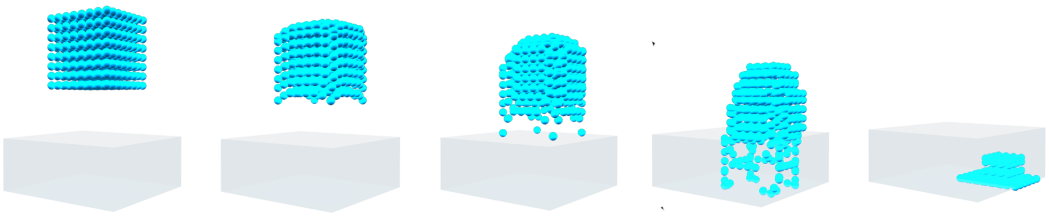

Figure 3: Fluid simulation using 512 particles.

5 Discussion and Future Work

The results seen in section 4.3, shows three figures each containing a picture sequence of five with different amount of particles, moving towards a box for interaction. The computation time increased sharply with an increasing amount of particles. For image 1, with 12 particles, the computation time was about 13 seconds, including setting up the scene and setting out 200 keyframes for every new position. For image 2, including 27 particles, the computation time had the average of 2 minutes and 53 seconds. For image 3 the computation time was about 5 hours.

One disadvantage with particle-based algorithms for fluid simulations is that the entire volume is necessary to be computed, including the particles which never is rendered. This problem is not easily solved and not possible to work around, since one particle's movement depends consistent on all particles nearby. To maintain a reasonable computation time, other parts in the method needs to be focused on for improvements, such as the neighbour search algorithm. The searching method for finding the neighboring particles in the implemented project is simply done by going through the whole fluid volume, and compare each particle to the smooth kernel radius, to determine if the current particle is classified as a neighbouring particle. This method has a slow computation time, but is today enough fast to test the project simulation. For more advanced simulations, using more particles, another search algorithm is required. The computation time would be improved by using the search algorithm Plane Sweep algorithm for example. The algorithm uses a line to sweep over the particles along a specified direction, to then do a similar sweep in a new direction. The method is mesh-free which offers a great flexibility of smooth kernel radius h. The line stops at every particle to obtain information to apply the forces and calculations [4]. Another possible method to use is to implement a cell list over the particles, where all particles are located inside a voxel with the size as the smooth kernel radius h in all directions. The neighbour search is then simplified by only looking at the particles inside the same voxel as the particle of interest, to classify those particles as neighbouring particles.

In early stages of the simulation, the result varied for different amount of particles. For eight particles, the simulation was smooth from initial position to reaching the floor of the box. Problem aroused when the amount of particles increased, which lead to the particles moving in the wrong direction, upwards. The reason for this could possibly have been functions in the code calculating the forces, returning a value with a very small error. When the amount of particles increased, more computations were made for every particle and the error changed faster, which was visibly seen. Tweaking the constant values in the equations, such as the smooth kernel radius h, lead to a more realistic simulation with a bigger amount of particles.

Today, the particles within the simulation only handles collisions with the boundary box, but not between the particles. Correct force calculations including pressure should handle collisions between particles. Using the improved kernel function Spiky kernel should in addition prevent the repulse force from being vanished when two particles are close to each other, but for some reason this is not the case in the resulting simulation. The exact problem with the implementation is not successfully investigated.

The surface tension force was not considered within the algorithm which is a commonly used force in particle-based fluid simulations. Surface tension is required for describing the changes in the forces closer to the water surface, due to smaller quantity of neighboring particles near the surface. The tension force was not added to the project due to the already described issues with the implemented forces, but would be a necessary complement to the algorithm to improve the accuracy in the simulation.

6 Conclusion

The SPH method is useful for fluid simulations since it results in accurate and realistic fluid simulations. However, the method is not suitable for all types of simulations and is often used in combination with other methods. The project presented in this report simulates particles falling from a specific height into a glass box, and is today not a completed water simulation, but a good result due to the time spended on the project. Implementations of fluid simulations are time consuming and future improvements is needed for the simulation to imitate realistic water, but the result of this project is a proper start for further development. Improvements such as development of the neighbour search method and a substituted time integration method is desirable, as suggested in section 5. Also correction of the physics equations is required for future work.

Even though SPH is a good method for achieving accurate fluid simulations, there is today not one enough good method that contains all the characteristics of a fluid, delivered in a realistic time of computation. The topic of fluid simulations still requires more research and development for achieving realistic results for real time renders. But, the SPH method is nowadays one of the best methods for simulating fluids, and in combination with other implementation methods, such as grid-based methods, would generate an accurate fluid simulation with lower cost of computation time.

References

[1] Mickey Kelager, *Lagrangian Fluid Dynamics Using Smoothed Particle Hydrodynamics*, DIKU 2006

[2] Matthias Müller, David Charypar and Markus Gross *Particle-Based Fluid Simulation for Interactive Applications*, Eurographics/SIGGRAPH Symposium on Computer Animation, 2003

[3] Aditya Hendra, *Accelerating Fluids Simulation Using SPH and Implementation on GPU*, Uppsala Universitet, Department of Information Technology, 2015

[4] Dong Wang, Yisong Zhou and Sihong Shao, *Efficient Implementation of Smoothed Particle Hydrodynamics (SPH) with Plane Sweep Algorithm*, Global-Science Press, 2016

A Implemented Code

```
# calculateForcesSpheres.py

import maya.cmds as cmds
import maya.mel as mel
import math

# ***************************
# ---------- SETUP ----------
# ***************************

# ----- Create Light -----

# Create ambient light

light = cmds.ambientLight(intensity=1.0)
cmds.ambientLight( light, e=True, ss=True, intensity=0.2, n='lightAmb')
cmds.move( 0, 8, 0 )

# Create directional light
light = cmds.directionalLight(rotation=(45, 30, 15), n='lightDir')
cmds.directionalLight( light, e=True, ss=True, intensity=0.0 )

# Query it
cmds.ambientLight( light, q=True, intensity=True )
cmds.directionalLight( light, q=True, intensity=True )


# ----- Create Transparent Box -----

# Create the groundPlane
cmds.polyCube(w=5, h=2, d=5, sx=1, sy=1, sz=1, ax=(0, 1, 0), name='transpCube')
cmds.polyNormal( nm=0 ) #change normals

# Delete top of cube
cmds.select( 'transpCube.f[1]' )
cmds.delete()

# Create transparent material for transparent box
cmds.select( 'transpCube' )
cmds.setAttr( 'lambert1.transparency', 0.922581, 0.922581, 0.922581,
type = 'double3')
cmds.setAttr( 'lambert1.refractions', 1 )
cmds.setAttr( 'lambert1.refractiveIndex', 1.52 )
cmds.setAttr( 'lambert1.color', 0.417, 0.775769, 1, type = 'double3' )

# ----- Create Particles -----

count=0
for i in range( 0, 8 ):
    for j in range( 0, 8 ):
        for k in range( 0, 8 ):
            count=count+1
            result = cmds.polySphere( r=0.15, sx=1, sy=1, name='particle#' )
            cmds.select('particle' + str(count))
            cmds.move(-i*0.35+0.75, 3+j*0.35, k*0.35-0.75,'particle' + str(count))
```

```
            cmds.sets( name='redMaterialGroup', renderable=True, empty=True )
            cmds.shadingNode( 'lambertian', name='redShader', asShader=True )
            cmds.setAttr( 'redShader.color', 0.0, 0.5333, 0.8, type='double3' )
            cmds.surfaceShaderList( 'redShader', add='redMaterialGroup' )
            cmds.sets( 'particle' + str(count), e=True,
            forceElement='redMaterialGroup' )


# ***************************
# ------- FOR ANIMATION -------
# ***************************


# ------- FUNCTIONS -------


# Set Keyframes
def setNextKeyParticle( pName, pKeyStart, pTargetAttribute, pValue ):

    # keyNext = pKeyStart + pDt
    keyNext = pKeyStart

    # clear selection list and select all particles
    cmds.select( clear=True )

    # create animation, set startkeyframe, startvalue=0 at first key frame. Make
    linear keyframes
    cmds.setKeyframe( pName, time=keyNext, attribute=pTargetAttribute, value=pValue )

    # cmds.setKeyframe( pName, time=pEndKey, attribute=pTargetAttribute,
    value=pValue )
    cmds.selectKey( pName, time=( pKeyStart, keyNext ), attribute=pTargetAttribute,
    keyframe=True )
    cmds.keyTangent( inTangentType='linear', outTangentType='linear' )



# Function returnig array of neighbouring particles with smoothing length
def findNeighbours( pParticle, pSmoothLength, pNr ):

    neighbourList = []

    for i in range( 1, pNr ):

        pos = [ cmds.getAttr( 'particle'+str(i)+'.translateX' ),
                cmds.getAttr( 'particle'+str(i)+'.translateY' ),
                cmds.getAttr( 'particle'+str(i)+'.translateZ' ) ]

        deltadist = [0, 0, 0]

        # Calculate distance
        deltadist[0] = pos[0] - pParticle[0]
        deltadist[1] = pos[1] - pParticle[1]
        deltadist[2] = pos[2] - pParticle[2]

        distance = math.sqrt( math.pow(deltadist[0], 2) +
        math.pow(deltadist[1], 2) +
        math.pow(deltadist[2], 2) )
```

```
        # if distance <= psmoothLength, put in array
        if distance <= pSmoothLength and distance > 0:
            neighbourList.append( 'particle'+str(i) )

    return neighbourList

# wfPoly6
def wfPoly6( pParticle, pH, pNr ):

    a = 315/( 64*3.14*math.pow( pH, pNr ) )
    w = a * math.pow( ( math.pow( pH, 2 ) - math.pow( pParticle, 2 ) ), 3 )

    return w

# wfGradientSpiky
def wfGradientSpiky( pParticle, pH ):

    a = -45/(64*3.14*math.pow( pH, 6 ) )
    w = [ a * math.pow( pH - pParticle[0], 2 ),
          a * math.pow( pH - pParticle[1], 2 ),
          a * math.pow( pH - pParticle[2], 2 ) ]

    return w

# wfLaplacianviscosity
def wfLaplacianviscosity( pParticle, pH ):

    a = 45/( 3.14*math.pow( pH, 6 ) )
    w = [ a * ( pH-pParticle[0] ),
          a * ( pH-pParticle[1] ),
          a * ( pH-pParticle[2] ) ]

    return w


# Compute Density
def calculateDensity( pPosition, pList, pH, pMass, pNr ):

    density = [0.1, 0.1, 0.1]

    for i in range( 0, len( pList ) ):

        nPos = [ cmds.getAttr( pList[i]+'.translateX' ),
                 cmds.getAttr( pList[i]+'.translateY' ),
                 cmds.getAttr( pList[i]+'.translateZ' ) ]

        deltaPos = [ pPosition[0] - nPos[0],
                     pPosition[1] - nPos[1],
                     pPosition[2] - nPos[2] ]

        density[0] += pMass * wfPoly6( deltaPos[0], pH, pNr )
        density[1] += pMass * wfPoly6( deltaPos[1], pH, pNr )
        density[2] += pMass * wfPoly6( deltaPos[2], pH, pNr )

    return density
```

```
# Compute pressures from density
def calculatePressure(  pDensity ):

    k = 1
    p0 = 0

    pressure = [ k * (pDensity[0] - p0),
                 k * (pDensity[1] - p0),
                 k * (pDensity[2] - p0) ]

    return pressure

# Compute Pressure Force from pressure interaction between neighbouring particles
def calculatePressureForce( pPosition, pDensity, pDensityList, pList, pMass, pNr ):

    pressureF = [0, 0, 0]
    pH = 1

    for i in range( 0, len( pList ) ):

        nPos = [ cmds.getAttr( pList[i]+'.translateX' ),
                 cmds.getAttr( pList[i]+'.translateY' ),
                 cmds.getAttr( pList[i]+'.translateZ' ) ]

        deltaPos = [ pPosition[0] - nPos[0],
                     pPosition[1] - nPos[1],
                     pPosition[2] - nPos[2] ]


        nDensity = calculateDensity( nPos, pList, 1, pMass, pNr )

        nPressure = calculatePressure( pDensityList[i] )
        particlePressure = calculatePressure( pDensity )

        gradient = wfGradientSpiky( deltaPos, pH )

        pressureF[0] += (-1) * pMass * ( ( particlePressure[0] + nPressure[0] )
        / ( 2 * nPressure[0] ) ) * gradient[0]
        pressureF[1] += (-1) * pMass * ( ( particlePressure[1] + nPressure[1] )
        / ( 2 * nPressure[1] ) ) * gradient[1]
        pressureF[2] += (-1) * pMass * ( ( particlePressure[2] + nPressure[2] )
        / ( 2 * nPressure[2] ) ) * gradient[2]

    return pressureF

# Compute Viscosity
def calculateViscosity( pPosition, pID, pList, pVelocity, pDensity, pH ):

    viscosity = [0, 0, 0]
    uConstant = 1

    pVel = pVelocity[ pID ]

    for i in range( 0, len( pList ) ):

        nPos = [ cmds.getAttr( pList[i]+'.translateX' ),
                 cmds.getAttr( pList[i]+'.translateY' ),
                 cmds.getAttr( pList[i]+'.translateZ' ) ]
```

```
            deltaPos = [ pPosition[0] - nPos[0],
                         pPosition[1] - nPos[1],
                         pPosition[2] - nPos[2] ]


            nVel = pVelocity[ i ]

            smoothVisc = wfLaplacianviscosity( deltaPos, pH )

            viscosity[0] += mass * ( ( nVel[0] - pVel[0] ) / pDensity[0] )
            viscosity[1] += mass * ( ( nVel[1] - pVel[1] ) / pDensity[1] )
            viscosity[2] += mass * ( ( nVel[2] - pVel[2] ) / pDensity[2] )

        return ( uConstant * viscosity )


    def calculateNewPosition( pParticlePos, pVelocityList, pDt ):

        newPosition = [ pDt * pVelocityList[0] + pParticlePos[0],
                        pDt * pVelocityList[1] + pParticlePos[1],
                        pDt * pVelocityList[2] + pParticlePos[2] ]

        #Boundary conditions
        Xmin = -2.5
        Xmax = 2.5
        Ymin = -0.5
        Zmin = -2.5
        Zmax = 2.5

        # X
        if ( newPosition[0] < Xmin or newPosition[0] > Xmax ):

            pVelocityList[0] = 0.0

            if ( newPosition[0] < Xmin ):
                newPosition[0] = Xmin

            if ( newPosition[0] > Xmax ):
                newPosition[0] = Xmax
        # Y
        if ( newPosition[1] < Ymin ):

            pVelocityList[1] = 0.0
            newPosition[1] = -0.5

        # Z
        if ( newPosition[2] < Zmin or newPosition[2] > Zmax ):
            pVelocityList[2] = 0.0

            if ( newPosition[2] < Zmin ):
                newPosition[2] = Zmin

            if ( newPosition[2] > Zmax ):
                newPosition[2] = Zmax


        posAndVel = [ newPosition[0], newPosition[1], newPosition[2], pVelocityList[0],
```

```
    pVelocityList[1], pVelocityList[2] ]
    return posAndVel



# ************************************************#

# ------------------------ MAIN ------------------------

# ************************************************#


h = 1
mass = 5
G = 9.82
nr = 1001
time = startTime

dt = 1 # 24 f/s

# Playback options
cmds.playbackOptions( playbackSpeed=0, maxPlaybackSpeed=1 )
cmds.playbackOptions( min=1, max=300 )
startTime = cmds.playbackOptions( query=True, minTime=True )
endTime = cmds.playbackOptions( query=True, maxTime=True )


# Set first Keyframe for all partices
for i in range ( 1, nr ):
    pos = [ cmds.getAttr( 'particle'+str(i)+'.translateX' ),
            cmds.getAttr( 'particle'+str(i)+'.translateY' ),
            cmds.getAttr( 'particle'+str(i)+'.translateZ' ) ]

    setNextKeyParticle( 'particle'+str(i), time, 'translateX', pos[0] )
    setNextKeyParticle( 'particle'+str(i), time, 'translateY', pos[1] )
    setNextKeyParticle( 'particle'+str(i), time, 'translateZ', pos[2] )


# vel [col][row]
velocityList = [ [0.0, 0.0, 0.0] ] * 13
densityList = [ [0.1, 0.1, 0.1] ] * 13

for j in range ( 1, 300 ):
    print 'frame: ' + str(j)
    time += dt
    for i in range ( 1, nr ):

        particlePos = [ cmds.getAttr( 'particle'+str(i)+'.translateX' ),
                        cmds.getAttr( 'particle'+str(i)+'.translateY' ),
                        cmds.getAttr( 'particle'+str(i)+'.translateZ' ) ]

        # ---------- CALCULATE FORCES ----------

        listNeighbours = findNeighbours( particlePos, h, nr )

        # 1. Compute Density
        density = calculateDensity( particlePos, listNeighbours, h, mass, nr )
        densityList[i] = density
```

```
# 2. + 3. Compute pressure force from pressure
interaction between neighbouring particles
pressure = calculatePressureForce( particlePos, density, densityList,
listNeighbours, mass, nr )


# 4. Compute viscosity force between neighbouring particles
viscosity = calculateViscosity( particlePos, i, listNeighbours, velocityList,
density, h )

# 5. Sum the pressure force, viscosity force and external force: gravity
gravityForce = -9.82 * mass * time

totalForce = [ pressure[0] + viscosity[0],
               pressure[1] + viscosity[1] + gravityForce,
               pressure[2] + viscosity[2]  ]

# 6. Compute the acceleration

acceleration = [ totalForce[0] / density[0],
                 totalForce[1] / density[1],
                 totalForce[2] / density[2] ]

# velocity
velocityList[i] = [ acceleration[0] * (time/576),
                    acceleration[1] * (time/576),
                    acceleration[2] * (time/576)]


'''
velocityList[i] = [ velocityList[i][0] + acceleration[0] * (time/576),
                    velocityList[i][1] + acceleration[1] * (time/576),
                    velocityList[i][2] + acceleration[2] * (time/576) ]
'''

# 7. new position

positionAndVelocity = calculateNewPosition( particlePos,
velocityList[i], dt )
newParticlePosition = [ positionAndVelocity[0], positionAndVelocity[1],
positionAndVelocity[2] ]

velocityList[i][0] = positionAndVelocity[3]
velocityList[i][1] = positionAndVelocity[4]
velocityList[i][2] = positionAndVelocity[5]


# Set keyframes
cmds.select( 'particle'+str(i) )
setNextKeyParticle( 'particle'+str(i), time, 'translateX',
newParticlePosition[0] )
setNextKeyParticle( 'particle'+str(i), time, 'translateY',
newParticlePosition[1] )
setNextKeyParticle( 'particle'+str(i), time, 'translateZ',
newParticlePosition[2] )
```