Eighth Hungarian Conference on Computer Graphics and Geometry, Budapest, 2016

Simulation methods for elastic and fluid materials

Tamás Umenhoffer, Artúr M. Marschal, Péter Suti

Department of Control Engineering and Informatics, Budapest University of Technology and Economics, Budapest, Hungary

Abstract

In this paper we present techniques for physical simulation of deformable objects. We investigate three problems namely: incompressible fluid flow, elastic body simulation, and elasto-plastic material simulation. For high detail fluid simulation we discuss the particle in cell method. For an intuitive elastic material simulation we present the position based dynamics method. Finally we show how to use the material point method for the simulation of the most complex, elastic plastic materials.

1. Introduction

A wide range of real world materials are not rigid but deformable. These materials include fluids and materials with fluid like motion (e.g. smoke or clouds), elastic materials like rubber or plastic materials like plasticine. Some materials have both plastic and elastic properties like snow.

Physical simulation of such materials is a hard problem, but necessary for realistic behaviour. Specially this is true for medical simulations. The human body is not rigid, it has elastic and plastic properties, and even handling the motion of fluids inside our body is crucial. What makes the simulation even harder is that materials with different properties are in close interaction.

In our recent project we investigate the blood flow around the aortic and pulmonary valves in a beating heart. Such a simulation requires the simulation of the blood flow as the motion of an incompressible fluid, but taking into account that the valves are elastic-plastic materials which are both moved by the flow, and they also hinder the blood flow.

In this paper we summarize our experiences with the physical simulation techniques we tried for fluid, elastic and plastic material simulation. As modern GPU-s can be used to spped up general purpose calculations[19], we used GPU acceleration where we could to enhance performance. In the following section we list the most important papers about deformable material simulation. Section 3 describes the particle in cell method for fluid simulation, Section 4 shows how to use the position based dynamics to simulate elastic materials, and Section 5 describes the material point method for simulating elastic-plastic materials like snow.

2. Previous Work

What makes deformable material modeling hard is that we have to model the whole volume of the objects, taking into account the internal friction of material elements. From this aspect fluid simulation and elastic material simulation is similar: the shape of the object deforms, material particles can roll away from each other, but their internal friction tries to keep the volume together.

Fluid motion can be modelled with particle systems[15], however it is not physically driven and demand a huge animating time. Solving the equations of motion has a much natural result. Foster and Metaxas used the full three dimensional Navier-Stokes equations to simulate smoke motion on a coarse grid[4]. Because of the explicit integration scheme, their algorithm was only stable for small time steps. This problem was solved by Stam[16], who introduced a semi-Lagrangian advection method and implicit solvers in his stable fluid simulation. Harris[7] built his simulation algorithm on Stam's stable fluid and redesigned the algorithm to run entirely on GPU. He also gave an excellent survey on fluid simulation on graphics hardware[6]. The smoothed particle hydrodinamics[9] method works on particles instead of a grid, Müller[12] introduced an implementation for interactive applications. Hybrid methods like particle in cell[5] and fluid implicit particles[3] combine the strength of both representations.

Elastic materials can be easily modelled with mass spring

systems, however setting spring parameters to achieve realistic result is not a trivial problem. The most widely used techniques in computational sciences are finite element methods. However these solvers are rathed slow. Position based integration[13] is an intuitive stable method, but (like mass spring systems) also hard make physically correct.

Some graphics papers use simplified particle or rigid body systems [10, 2] to model granular material, whose behaviour is mostly directed by inter-granular friction. However it is hard to keep efficiency when increasing the detail, thus increasing the number of grains. This lead researchers to apply continuum models [20, 8, 14, 1]. Incompressible fluid based methods model granular friction with viscosity. For some materials compressibility should be modelled. The material point method (MPM)[18], was designed to extend FLIP to solid mechanics problems that require compressibility.

3. Fluid simulation with the particle in cell method

A fluid with constant density and temperature can be described by its velocity $\vec{u}$ and pressure p fields. These values both vary in space and in time:

$$\vec{u} = \vec{u}(\vec{x},t), \quad p = p(\vec{x},t).$$

The motion of a fluid is described by the Navier–Stokes equations:

$$\frac{\partial \vec{u}}{\partial t} = -(\vec{u}\cdot\nabla)\vec{u} - \frac{1}{\rho}\nabla p + \nu\nabla^2\vec{u} + \vec{F}, \tag{1}$$

$$\nabla\cdot\vec{u} = 0, \tag{2}$$

where ρ is the density, ν is the viscosity of the fluid, and in a Cartesian coordinate system $\nabla = (\partial/\partial x, \partial/\partial y, \partial/\partial z)$. Equation 1 describes the conservation of momentum while equation 2 states the conservation of mass (i.e. that the velocity field is divergence free). These equations should also be associated with the definition of the boundary conditions. The first term on the right side of equation 1 expresses the advection of the velocity field itself. This term makes the Navier–Stokes equation non-linear. The second term shows the acceleration caused by the pressure gradient. The third term describes diffusion that is scaled by the viscosity, which a measure of how resistive the fluid is to flow. Finally $\vec{F}$ denotes the influence of external forces.

Basically two approaches exist for solving these equations according how they represent the continuum. Euler based approaches use a grid mesh for storage. Sample points thus does not move in time, neighbourhood information is obvious, and sample space does not deform. On the other hand material quantities should be transfered between gridpoints as the medium evolves, and because of interpolation a significant amount of smoothing can occur. Lagrangian methods sample the medium at individual moving fluid parcells. In the Lagrangian approach boundary sample points remain on boundary, collisions are easier to handle, and as material properties are fixed to the particles, time dependent properties are easier to manage.

In case of fluid dynamics Eulerian approaches are better in preserving volume, but small scale movements are smoothed out. Lagrangian approaches are better in handling collisons and particle advection. The Eulerian grids also have high storage needs. As volume preserving is an important feature for us, we should consider the grid based approach.

Particle in cell (PIC) methods combine the strength of both techniques. It samples the medium on two meshes: an Eulerian and a Lagrangian mesh. Using both representations we can choose the appropriate coordinate system for each simulation substep. Thus particle advection and collision handling should be performed on the Lagrangian particle representation, while computing pressure and making the flow field divergence free should be done on an Eulerian grid. Between these steps quantities should be transfered between the two meshes.

Our system is a classic grid based fluid simulation algorithm, but the advection part is replaced with a particle based advection. Before advection, particle velocities are transfered to the particles: for each particle we visit nearby voxels and collect velocities with a B-spline based interpolation function. Advection with an explicit Euler integration is trivial on particles. After advection we transfer velocities back to the Eulerian grid using the same interpolation function. All other simulation steps work the same as in a classical grid based approach.

On initialization we fill in our volume evenly with particles, placing four particles randomly in each voxel. During simulation, paricles move and can accumulate in certain voxels, while other voxels will be emptied. This behaviour is unwanted as it causes instability. Empty areas should be filled with new particles, while in dense areas particles should be removed. In practice we keep particle count between 2 and 6 in each cell.

3.1. Implementation

We implemented our algorithm on the OpenGL framework. From the first time our goal was to use GPU acceleration for simulation, so our simulation steps were implemented in GLSL shaders. There are numerous shader based Eulerian fluid simulation implementations that are publicly available. Our implementation basically does not differ from them, so now we focus only on the changes that should be made for the PIC method.

The classic semi-implicit advection step was removed and replaced with a particle based advection. To transfer velocities from the grid to the particles we send point primitives to the graphics pipeline, each point representing a particle. For each particle we sample the neighbourhood from the velocity texture and interpolate velocity data using our interpolation function. This interpolated velocity can be stored for

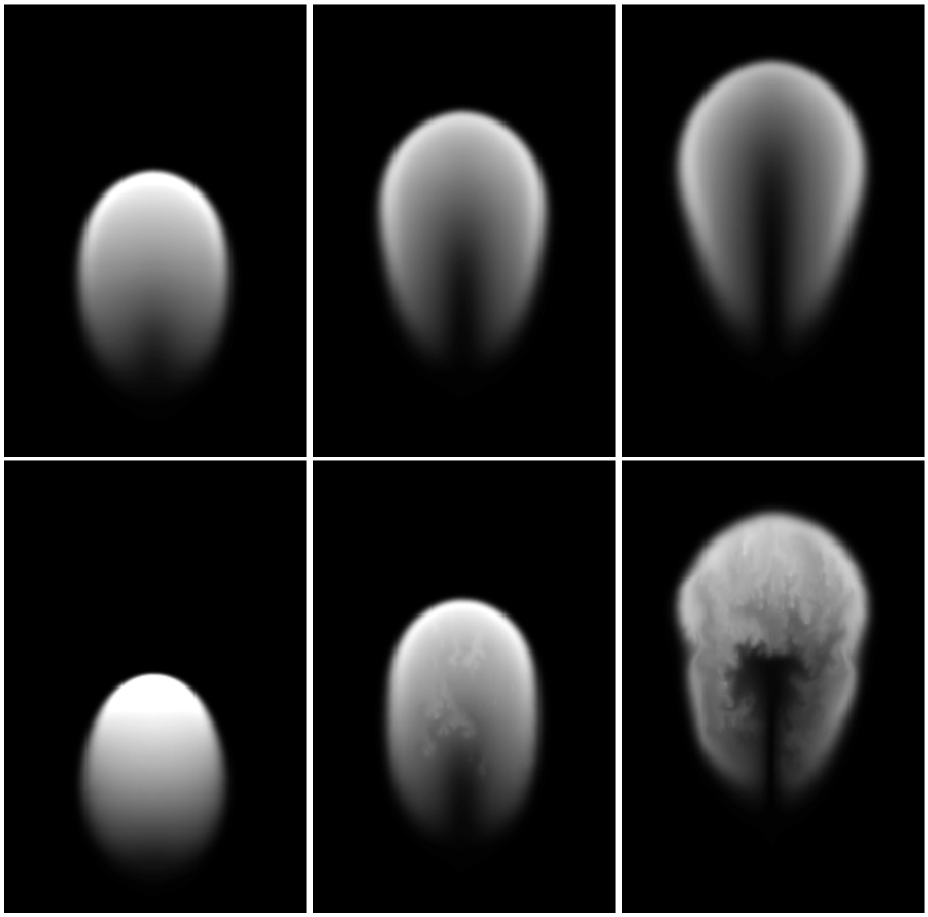

Figure 1: *Grid based (upper row) and PIC based fluid simulations. Notice the fine features present in the PIC result.*

the particle using vertex transform feedback. We should note that advection was also computed in this shader and new position values were also feed back to vertexbuffers. Using these vertex buffers we again send all particle point down the pipeline as point primitives, and use a geometry shader to extend them to quads covering the neccessary voxel area. The pixel shader calculates velocity value for each underlying voxel using the same interpolation function, and use additive blending to accumulate data from different particles. This last rendering is directed to a render texture, which can be used by forthcoming simulation steps.

Using alpha blending for particle data accumulation is effective as it does not require atomic operations, as it would if we would use a GPGPU framework. Handling particle insertion and deletion can also be implemented effectively with the 3D pipeline. We rasterize each particle to the underlying voxel of a special buffer, and write a color value of one. Again using additive blending we get the number of particles for a given cell. Using this texture we send all particles down the pipeline and a geometry shader removes some particles from a cell if the cell contains too much particles. The particles discarded by the geometry shader won't be fed back to the vertexbuffer. Similarly we can send new particles down the pipeline (we used the same buffer as for initializing particles), and the geometry shader keeps some of them if their cell contains too much particles. Here the feedback buffer is directed to the end of our previous vertex buffer. Thus our particle count changes continuosly during simulation.

One question still remains, namely: how does a geometry shader instance know which particle to remove or to add without communicating with other geometry shader instances. We need a strategy to know exactly for a given particle count in a cell which particles should be removed, and which new particles should be added. We use rejection sampling for this purpose. For each particle a priority value is assigned, which is a random number between 0 and 1. When removing particles only those particles will be removed whose priority is above the ratio of desired particle count and current particle count in the cell. After removal the distribution of priorities will change, higher priorities will

be removed, thus priorities should be scaled back to the unit interval. When adding new particles we use a similar technique: we add new particles with priorities above the ratio of current particle count and desired particle count. Again priorities should be scaled back to the unit interval.

Figure 1 compares the classic Eulerian simulation with the particle in cell method. Note that the two technique differs only in the advection step. The particle in cell method has fine details that were smoothed out by the semi implicit intergration of the purely grid based approach.

4. Elastic material simulation with position based dynamics

4.1. The position based dynamics method

The most popular approaches for the simulation of dynamic systems in computer graphics are force based. Internal and external forces are accumulated from which accelerations are computed based on Newton's second law of motion. A time integration method is then used to update the velocities and finally the positions of the object. One typical force based solution for elastic materials is mass a spring systems.

The position based dynamics (PBD) is an approach which omits the velocity layer and immediately works on the positions. The main advantage of a position based approach is its controllability. Overshooting problems of explicit integration schemes in force based systems can be avoided. In addition, collision constraints can be handled easily and penetrations can be resolved completely by projecting points to valid locations.

The objects to be simulated are represented by a set of N particles and a set of M constraints. Each particle i has three attributes, namely mass (m_i), position (x_i) and velocity (v_i). Constraints $C_j : \mathbb{R}^{3n_j} \rightarrow \mathbb{R}$, that limit particle motion are defined on a set of particles with indices $\{i_{1_n}, ..., i_{n_j}\}$, $i_k \in [1, ..., N]$. Constraints are satisfied if $C_j(x_{i_1}, ..., x_{i_{n_j}}) > 0$ or $C_j(x_{i_1}, ..., x_{i_{n_j}}) \geq 0$ depending on the type of the given constraint. Constraints also have a stiffness parameter $k_j : [0, ...1]$ that defines their strength.

The basic steps of the algorithm are listed in Algorithm 1. Before the main simulation loop begins, initial positions and velocities of the particles should be set. The first step of the simulation loop is an explicit Euler integration step, where particle velocities and positions are updated. These temporal positions will be projected by the constraints. Non permanent constraints, like collision constraints should be recalculated in each simulation loop. In the iteration loop all constraints are evaulated one after an other, and predicted positions will be corrected. Corrected positions will be stored as final positions and can also be used to compute the actual particle velocities.

The position based dynamics approach uses a non-linear Gauss-Seidel solver, which means that each constraint will

Algorithm 1 Position Based Dynamics Method

```
1: for all particles i do
2:     initialize x_i = x_i^0; v_i = v_i^0; w_i = 1/m_i
3: end for
4: loop:
5: for all particles i do v_i ← v_i + △t w_i f_ext(x_i)
6: for all particles i do p_i ← x_i + △t v_i
7: for all particles i do createCollisionConstraints(x_i → p_i)
8: for all iterations do
9:     projectConstraints(C_1, ..., C_{M+M_coll}, p_1, ..., p_N)
10: end for
11: for all particles i do
12:     v_i ← (p_i − x_i)/△t
13:     x_i ← p_i
14: end for
15: endloop
```

be solved separately, but each constraint works on the result of the previously solved constraint. The PBD approach linearizes the constraints, and searches the correction value $\triangle p_i$ in the direction of the constraint gradient $\nabla_{p_i} C(p)$. The general formula for solving a constraint is: $\triangle p_i = -s w_i \nabla_{p_i} C(p)$ where,

$$s = \frac{C(p)}{\sum_j w_j |\nabla_{p_j} C(p)|^2}, w_i = \frac{1}{m_i}$$

The following subsections cover the constraints we have implemented in our simulation system.

4.2. Distance constraint

The goal of this constraint is to maintain a fixed distance between two particles. The constraint can be given with

$$C(p_1, p_2) = |p_1 - p_2| - d$$

. The derivatives are $\nabla_{p_1} C(p_1, p_2) = n, \nabla_{p_2} C(p_1, p_2) = -n$ where $n = \frac{p_1 - p_2}{|p1_p 2|}$. The final correction formulas are:

$$\triangle p_1 = -\frac{w_1}{w_1 + w_2}(|p_1 - p_2| - d)\frac{p_1 - p_2}{|p_1 - p_2|}$$

$$\triangle p_2 = +\frac{w_1}{w_1 + w_2}(|p_1 - p_2| - d)\frac{p_1 - p_2}{|p_1 - p_2|}$$

This states that, based on the difference between the desired distance d and current distace, particles will move towards or away each other, and their movement ratio are defined by their mass ratio. This constraint keeps the material together, preventing the simulated object from streching and tearing.

4.3. Bending constraint

Bending constraint defines the bending resistance of the material, giving it an extra stiffness. Its goal is to keep an initial angle between adjacent triangles formed by particles. The

constraint is given by $C(p_1,p_2,p_3,p_4) = arccos(n_1 \cdot n_2) - \varphi_0$, where n_1 and n_2 are the normal vectors of the triangles, which can be calculated from particle positions using triangle edge cross product. The derivation of the constraint formulas are given in[11].

4.4. Collision constraint

Collision constraints can appear and dissapear in each simulation step. For a moving particle q and particles p_1,p_2,p_3 forming a triangle with normal n, the collision constraint is

$$C(q,p_1,p_2,p_3) = (q-p_1)\cdot n \geq 0$$

. The final formula for solving the triangle collision constraint is:

$$\triangle p = -((q-p_1)\cdot n)n$$

What the above formula does is moving the particle along the triangle normal until it gets in front of the triangle surface.

4.5. Implementation

We implemented the position based dynamics algorithm on the OpenCL framework with OpenGL visualization. We reserve particle data arrays for particle positions, predicted positions, velocities and inverse mass. We also reserve arrays for each constraint types. The constraint arrays store constraint specific infromation, for example for distance constraints we should store the two particle id this constraint works on and an initial distance. For each type of constraint a proper OpenCL kernel was written, that can handle the given constraint. These kernels read and write data from the particle data arrays, and one kernel thread works on one specific constraint instance. Thus the constraints are evaluated parallel and independently. Because no synchronization is used between working threads, it can happen that two parallelly ran thread work on joint particles.

This is unfortunatelly agains the principles of a Gauss-Seidel solver, where constraints use the result of previously solved constraints. Particulary, in our system constraints can override the result of other constraints without any concern. We overcome this problem by rearranging the order constraints are evaluated. We suffle constraint orders to decrease the chance of buffer read-write collisions. This ordering can even be changed in each iteration step to make the simulation more stable.

Collision constraints need special handling as their number change continuosly. We store scene triangle data in buffers. Before starting solving constraint iterations, for each particle we check collision with each scene triangle using ray-triangle intersection. As collision constraint count is not known in advance we reserve a fixed size array for them, preparing for worst case (thus with size $particleCount \cdot triangleCount$). We plan to use dynamically growing buffers in the future.

We tested our implementation on a thin layer draperie object and on solid objects too. Figure 2 shows frames from an animation where a cloth is dropped onto a cube. For cloth like materials we used distance, bending and collision constraints too. For solid objects we fill in the interiour of the objects with randomly placed particles and create distance constraints between closely placed particles.

We found that position based dynamics is a fast, stable and intuitive algorithm that can be well parallelized. However reconciling it with real physical quantities like elastic modulus is hard.

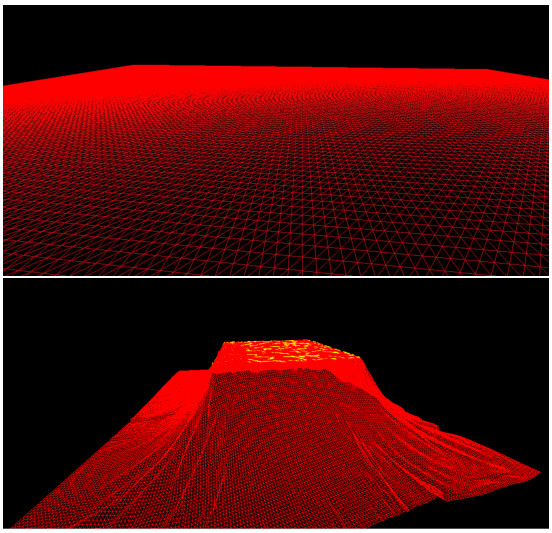

Figure 2: *Frames of an animation of a dropped draperie onto a cube.*

5. Elastic-plastic material simulation with the material point method

We have seen that elastic materials can be simulatied with the position based dynamics method, but that method is not based on real physical quantities. Material point method uses a different strategy. The method is based on the conservation equations for mass and momentum:

$$\frac{d\rho}{dt} + \rho\nabla\cdot v = 0, \rho\frac{dv}{dt} = \nabla\cdot\sigma + \rho g$$

Figure 3 shows the main components of the simulation. The algorithm can be described briefly as follows. We track material properties at particle positions. However some quantities are easier to compute on a grid, so our first step is to rasterize the particles onto a 3D grid. Note that this is the same strategy as was used in the particle in cell method. More precisely the material point method is an extension of the particle in cell method to computational solid dynamics.

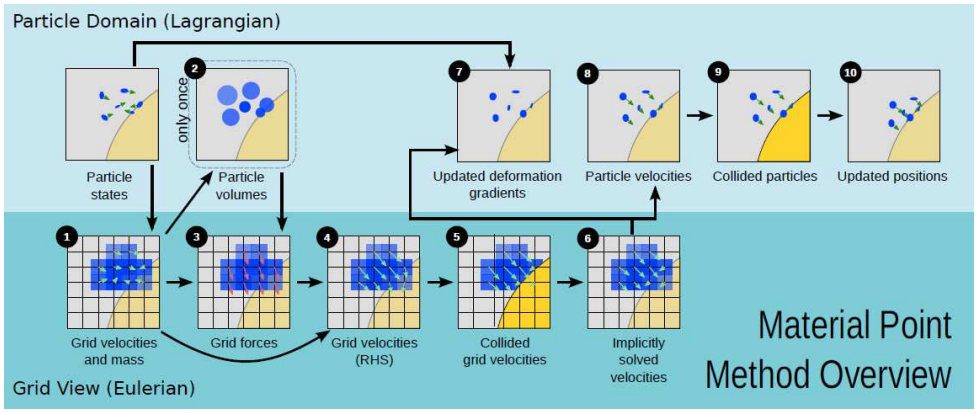

Figure 3: *Overview of the material point method (figure from [17]).*

The next step is to compute forces based on deformation gradients at each grid voxel, and update grid velocities. Using these velocities we perform collision detection on the grid. The final velocities are transferred back to the particles. Then calculate a new deformation gradient for each particle using refreshed voxel velocities. Table 1 list the notations of parameters used in our expressions and gives their typical values where available. This table also serves as a useful guide for implementing the MPM method, as all important particle and grid properties are listed. The next subsections describe the steps of the MPM simulation in more detail.

5.1. Rasterize particles

Each simulation frame starts with transferring particle velocities and mass to grid voxels, and ends with transferring updated velocities back to the particles. The grid can be thrown away at the end of each simulation step, so it can be redefined at the beginning of the simulation step, and can be fit to the actual particle positions. However for implementation reasons we used fixed grid position and resolution.

The connection between particles and the grid is achieved with an interpolation function. We used the same function as in [17], which use dyadic products of one-dimensional cubic B-splines. When transferring particle data, we compute the weights in a 5x5x5 voxel neighbourhood of each particle, and add the scaled particle quantity to the voxel value:

$$m_i^n = \sum_p m_p w_{ip}^n$$

$$v_i^n = \sum_p v_p^n m_p w_{ip}^n / m_i^n$$

When transferring voxel data back to particles, we again sample the 5x5x5 voxel neighbourhood of each particle, and sum their weighted average:

$$v_p^n = \sum_i v_i^n w_{ip}^n$$

Name	Notation	Typical values
Global parameters		
Young modulus	E_0	1.4×10^5
Poisson ratio	ν	0.2
Critical compression	θ_c	2.5×10^-2
Critical stretch	θ_s	7.5×10^-3
Hardening coefficient	ξ	10
Particle data		
Initial density	ρ_{p_0}	400
Initial volume	V_{p_0}	
Elastic force	F_{E_p}	
Plastic force	F_{P_p}	
Rotational force	R_{E_p}	
Elastic determinant	J_{E_p}	
Cell data		
Mass	m_i	
Velocity	v_i	
Force	f_i	
Collider flag	co_i	
Collider velocity	v_{co_i}	
Collider normal	n_{co_i}	

Table 1: *Notation of parameters used in our simulation framework. Default values are also listed where possible.*

Because of the additivity, if we are interested in the derivative of one quantity we can simply transfer with the derivative of the weight function.

5.2. Particle volume and density

As an initial step particle density and volume should also be calculated. After rasterizing particle mass, density is given with the following expression:

$$\rho_p^0 = \sum_i m_i^0 w_{ip}^0 / h^3$$

Particle volume can be calculated from particle mass and density:

$$V_p^0 = m_p / \rho_p^0$$

These values are initial values, they are not going to change during simulation.

5.3. Compute forces

Calculating stress-based forces needs derivatives, which are easier to evaluate on the grid rather on the particles. Stress-based forces are defined by the deformation gradient, the final expression of these forces is:

$$f_i(x) = -\sum_p V_p^0 \cdot (2\mu T_{co-rot} + \lambda T_{contour}) \cdot (F_{E_p}^n)^T \cdot \nabla w_{ip}^n,$$

$$T_{co-rot} = F_{E_p}^n - R_{E_p}^n, T_{contour} = (J_{E_p}^n - 1) J_{E_p}^n F_{E_p}^{n\,-T},$$

where $\mu = \mu(F_P) = \mu_0 \cdot e^{\xi(1-J_{P_p})}$,

and $\lambda = \lambda(F_P) = \lambda_0 e^{\xi(1-J_{P_p})}$

Here P_p and E_p are the plastic and elastic deformations of a particle, J_{P_p} and J_{E_p} are their determinants. λ and μ are the Lamè coefficients and can be computed from the Poisson ratio and Young modulus in an initial step:

$\lambda = \frac{E_0}{(1+\gamma)(1-2\gamma)}$ and $\mu = \frac{E_0}{2(1+\gamma)}$

5.4. Update velocities

If the stress based forces are calculated, voxel velocities can be updated:

$$v_i^* = v_i^n + \triangle t / m_i \cdot f_i^n$$

Here we can also add the effect of any additional external forces like gravity.

5.5. Grid based collision

Collision handling is performed on the voxelized scene geometry after adding forces. An inelastic sliding collision is used. If the voxel is a collider cell and its velocity is not zero, the relative velocity is calculated: $v_{rel_i} = v_i - v_{co_i}$. If the relative velocity has opposing direction with the collider normal (thus the particle and the collider are not separating), only the tangential component of the velocity vector is kept. After this collision handling step we can finalize our voxel velocities v_i^{n+1}.

Here we should note that [17] used a semi-implicit integration scheme here, which had much better accuracy, so smaller time steps could be used. Due to its high implementation complexity and additional memory needs we did not implement it. This results about five times longer simulation times in our system.

5.6. Update deform gradient

From the updated velocities the deformation gradient of each particle should be calculated. This gradient is divided into a plastic and an elastic part F_{E_p} and F_{P_p}. First we assume that all changes are attributed to the elastic part:

$$\hat{F}_{E_P}^{n+1} = (I + \triangle t \nabla v_p^{n+1}) F_{E_P}^n,$$

$$\hat{F}_{P_p}^{n+1} = F_{P_p}^n,$$

where $\nabla v_p^{n+1} = \sum_i v_i^{n+1} (\nabla w_{ip}^n)^T$.

The next step is to extract the stretching part of this gradient, and identify the amount of deformation the material could not hold, thus it breaks. This is done with a singular value decomposition and clamping the singular values:

$$SVD(F_{E_p}^{n+1}) = U_p \hat{\Sigma}_p V_p^T,$$

$$\Sigma_p = clamp(\hat{\Sigma}_p, [1-\theta_c, 1+\theta_c])$$

From the clamped singular values we can recalculate the elastic and plastic deformation gradients:

$$F_{E_p}^{n+1} = U_p \Sigma_p V_p^T,$$

$$F_{P_p}^{n+1} = F_{E_p}^{n+1\,-1} \hat{F}_{E_p}^{n+1} \hat{F}_{P_p}^{n+1} = V_p \Sigma_p^{-1} U_p^T \hat{F}_{E_p}^{n+1} F_{P_p}^n$$

These gradients will be used in the next simulation step to calculate stress-based forces.

5.7. Update particle velocities

Now that each voxel stores an updated velocity, these velocities should be written back to the particles. We use the same interpolation functions as for voxelizing particle data. Basically two methods can be used to update velocities: interpolate new velocities or interpolate the velocity change. The former is the classical particle in cell (PIC) method, the later is used in the fluid implicit particles (FLIP) method. For best results these two solutions should be mixed:

$$V_p^{n+1} = (1-\alpha)(\sum_i v_i^{n+1} w_{ip}^n) + \alpha(v_p^n + \sum_i (v_i^{n+1} - v_i^n) w_{ip}^n)$$

We used $\alpha = 0.9$.

5.8. Particle based collision

An additional collision handling step is needed as interpolation can bring back collision errors. We do the same calculations as in grid collision handling, but use particle velocities instead of voxel velocities, and address the grid cell the particle is in for collider information.

5.9. Update particle positions

Particles can be advected using their new velocities:

$$x_p^{n+1} = x_p^n + \triangle t v_p^{n+1}$$

Figure 4: *Frames of an animation of a dropped snowball simulated with the material point method.*

5.10. Implementation

We created a parallel CPU implementation of the material point metod. For statisfying results we have to keep particle count and grid resolution relatively high, which has notable memory needs. This can make GPU implementation difficult, as GPU memory size is much more limited.

We used OpenMP for parallelization. Most of the steps are easy to make parallel, but particle rasterization and elastic force calculation is a scattering operation, which is hard to rewrite as a gathering operation efficiently. Thus these steps were run on a single thread. In our tests we took into account rigid body interaction with an arbitrary shaped moving rigid body. Thus we created a framework to store animated geometry in a general format and voxelize it storing normal and velocity values in voxels. We used the Alembic API for animated scene storage and the OpenVDB API for hierechical voxelized scene storage.

Figure 4 shows frames from a simulation where a snowball was dropped to the ground. We used 600000 particles and 128x128x128 grid resolution. One frame of the animation took 10-15 minutes to calculate. Figure 5 demonstrates animated scene geometry, where a flat layer of snow is pushed. This animation was simulated with 250000 particles in a 80x36x60 grid.

6. Conclusions

We presented three methods for deformable body simulation: particle in cell for incompressible fluid flow, position based dynamics for elastic materials, and material point method for elastic-plastic materials. The results of the particle in cell method was pleasing, and introducing particles in the grid based method allows us to handle fluid and rigid body interaction much easier in the future. In our specific

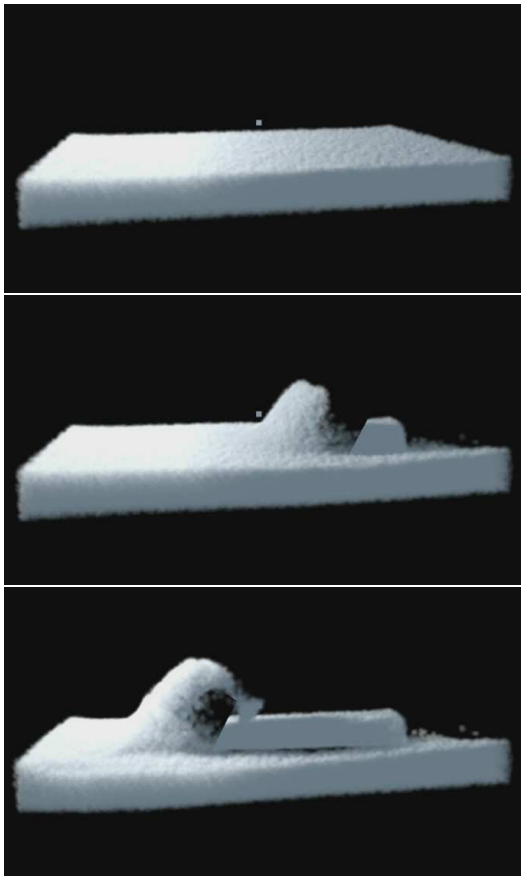

Figure 5: *A layer of snow pushed.*

heart valve simulation project the wall of the atrium will behaves as an infinitely strong rigid boundary, that pushes the fluid particles. On the other hand the valves are thin elastic layer that are deformed by the fluid flow. Using fluid particles this interaction is easier to implement as particle-triangle collisions. We could also provide an efficient GPU implementation of the PIC method.

The two methods for handling elastic and plastic materials differ in principles. Position based dynamics use intuitive constraint definitions, even plastic properties can be described with proper constraint setup. The resulting algorithm is easier to handle and it can be kept more stable. One drawback of this method is that it is not based on real world quantities, so there is no guaranty that the resulting simulation will behave physically correctly though we can tweak it to look correct.

Material pont method on the other hand is based on exact physical equations, physically correct results can be achieved giving real world quantities. On the other hand this technique has very high computational and memory costs. It

requires a lot of parameter tweaking to make the simulation stable and behave real.

Acknowledgements

This work has been supported by OTKA K–104476 and SCOPIA projects.

References

1. Iván Alduán and Miguel A. Otaduy. Sph granular flow with friction and cohesion. In Adam W. Bargteil and Michiel van de Panne, editors, *Symposium on Computer Animation*, pages 25–32. Eurographics Association, 2011. 2
2. Nathan Bell, Yizhou Yu, and Peter J. Mucha. Particle-based simulation of granular materials. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '05, pages 77–86, New York, NY, USA, 2005. ACM. 2
3. J. U. Brackbill and H. M. Ruppel. FLIP - A method for adaptively zoned, particle-in-cell calculations of fluid flows in two dimensions. *Journal of Computational Physics*, 65:314–343, August 1986. 1
4. Nick Foster and Dimitris Metaxas. Modeling the motion of a hot, turbulent gas. In *Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '97, pages 181–188, New York, NY, USA, 1997. ACM Press/Addison-Wesley Publishing Co. 1
5. F. H. Harlow. The particle-in-cell method for numerical solution of problems in fluid dynamics. In *Experimental Arithmetic, High-Speed Computations and Mathematics*, pages 319–343. RI: American Math. Society, 1964. 1
6. Mark Harris. Fast fluid dynamics simulation on the gpu. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM. 1
7. Mark J. Harris, William V. Baxter, Thorsten Scheuermann, and Anselmo Lastra. Simulation of cloud dynamics on graphics hardware. In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware*, HWWS '03, pages 92–101, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. 1
8. Toon Lenaerts and Philip Dutré. Mixing fluids and granular materials. *Comput. Graph. Forum*, 28(2):213–218, 2009. 2
9. L. B. Lucy. A numerical approach to the testing of the fission hypothesis. *The Astronomical Journal*, 82:1013–1024, December 1977. 1
10. Victor J. Milenkovic. Position-based physics: Simulating the motion of many highly interacting spheres and polyhedra. In *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '96, pages 129–136, New York, NY, USA, 1996. ACM. 2
11. Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. In César Mendoza and Isabel Navazo, editors, *VRIPHYS*, pages 71–80. Eurographics Association, 2006. 5
12. Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, SCA '03, pages 154–159, Aire-la-Ville, Switzerland, Switzerland, 2003. Eurographics Association. 1
13. Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, 18(2):109–118, 2007. 2
14. Rahul Narain, Abhinav Golas, and Ming C. Lin. Free-flowing granular materials with two-way solid coupling. *ACM Trans. Graph.*, 29(6):173, 2010. 2
15. W. T. Reeves. Particle systems—a technique for modeling a class of fuzzy objects. *ACM Trans. Graph.*, 2(2):91–108, April 1983. 1
16. Jos Stam. Stable fluids. In *Proceedings of the 26th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 121–128, New York, NY, USA, 1999. ACM Press/Addison-Wesley Publishing Co. 1
17. Alexey Stomakhin, Craig Schroeder, Lawrence Chai, Joseph Teran, and Andrew Selle. A material point method for snow simulation. *ACM Trans. Graph.*, 32(4):102:1–102:10, July 2013. 6, 7
18. D. Sulsky, S.-J. Zhou, and H. L. Schreyer. Application of particle-in-cell method to solid mechanics. *Comp. Phys. Comm.*, 87:236–252, 1995. 2
19. L. Szirmay-Kalos and L. Szécsi. General purpose computing on graphics processing units. In A. Iványi, editor, *Algorithms of Informatics*, pages 1451–1495. MondArt Kiadó, Budapest, 2010. http://sirkan.iit.bme.hu/šzirmay/gpgpu.pdf. 1
20. Yongning Zhu and Robert Bridson. Animating sand as a fluid. *ACM Trans. Graph.*, 24(3):965–972, 2005. 2