UNIVERSITY OF ZAGREB
FACULTY OF ELECTRICAL ENGINEERING AND COMPUTING

MASTER THESIS No. 1373

Skeletal Animation in Unified Particle Physics Engine

Bojan Lovrovic

Zagreb, July 2017.

Umjesto ove stranice umetnite izvornik Vašeg rada.
Da bi ste uklonili ovu stranicu obrišite naredbu `\izvornik`.

Thanks to my friends and colleagues Mihajlo Arbanas, Jure Ratković, Tomislav Maršić, Igor Dudjak, Ivan Glova and Matija Varga for providing advice, technical and mental support. Also, a special thanks to my bosses at Little Green Men for allowing me to use the company equipment during the development of this thesis and its implementation.

CONTENTS

1. Introduction

Particles have been used in computer simulations for decades to render all sorts of physical phenomena. The use is found in both offline and online (real-time) rendering and while the graphics hardware gets more and more powerful, a lot of methods used in offline rendering are finding their way into real-time applications, such as particle physics. Most known uses are fluid simulations which include liquid and gas rendering. Works like Macklin and Müller (2013) describe a real time application of particle hydrodynamic. Particles can also be used to simulate cloth and bodies with similar elastic properties. In addition to this Macklin et al. (2014) introduced rigid bodies to a particle framework with the goal of handling interactions between different types of objects in a uniform manner, hence the name Unified Particle Physics. Their work resulted in an real-time three-dimensional rendering framework called Flex[1]. Implementation that served as a basis for this thesis is mostly made out of methods presented in said paper and therefore is quite similar to the Flex framework.

The idea of this implementation was to extend the use of particles in Flex and to add support for skeletal animation (both procedural and data driven) while not excluding the features of the existing solutions. Thanks to the PBD, the data driven animation does not require any special handling besides uploading the transformations from the CPU. Procedural animation of skeletal mesh, namely rag-doll[2] will require some specially-defined particles (joints) which is the highlighted part of this thesis.

One of the biggest concerns when designing particle systems is spatial partitioning so this will be the first area to cover. From that, this thesis will move on to building the base of the simulation. It will explain the main structures used to represent the data in the context of Position-based dynamics and go through the main simulation loop. After the foundation, some concrete physical models will be introduced, like water,

[1] A demo of NVIDIA Flex is available (in the time of writing) on this address: https://developer.nvidia.com/flex

[2] Popular name for a simulation where character's bones are being treated as rigid bodies connected with joints.

rigid bodies and joints, which is the highlight of thesis. Lastly, the implementation details will be explained as well as some of the possible extensions and improvements.

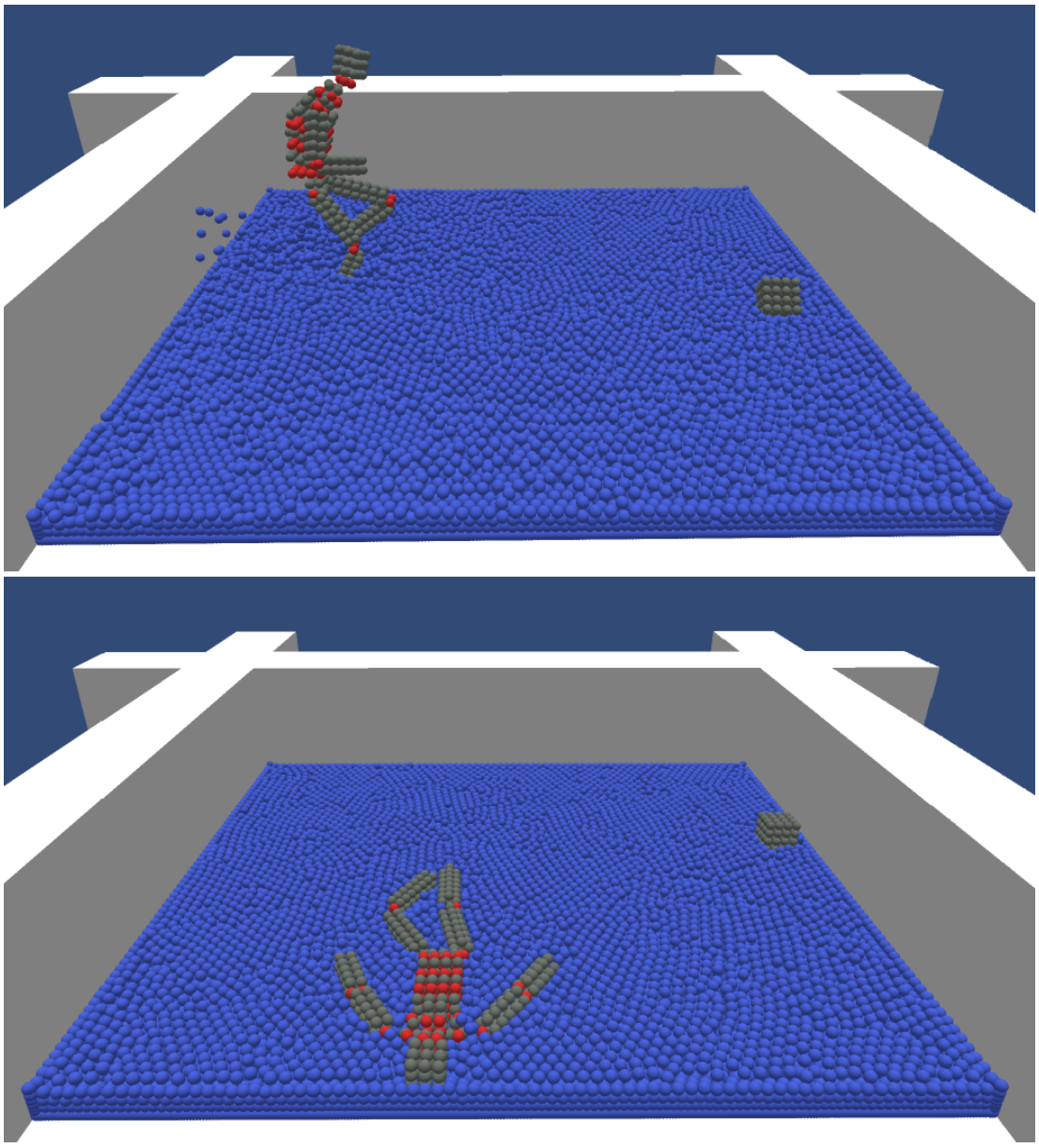

Figure 1.1: Ragdoll physics using uniform particle solver. On the upper image the skeletal system is being animated as it splashes the water in the pool it is in. Lower image displays the same skeletal system but later in time where the physical simulation has taken control over the bones and the body is just floating on the surface.

2. Spatial partitioning

The implementation of this paper can simulate up to 10^5 particles at once in real time. This number is empirically chosen as it meets the "standards" of the current particle simulators and does not limit the creativity of the user too much. Also, it prevents overly complex setups[1] where, due to many cache misses on GPU memory in particle-to-particle processes, drop in frame rate is highly probable. Note that since particles are in global memory, every look up is a 'cache miss' because global memory does not have cache (except textures).

Each of those particles can interact with any other at any given moment. This means that without any optimization the collision detection has time complexity of $O(n^2)$. This is where choosing the spatial partitioning structure comes into play. The only acceptable structure is the type that has construction time complexity of $O(n)$. With that requirement methods like BSP[2] and octree[3] are unsuitable. Grid based methods, on the other hand, do have a fast-enough construction time since position for an element in the structure can be calculated directly from the location in the simulation space.

The grid based method used here is called spatial hashing, but before it is described I'll go through sort of a predecessor of this method called uniform grid. The uniform grid is made out of cells which have the size equal to the double radius of the particle. This way, in a simulation where interpenetration is absent, there should be room for only 4 particles in 3 dimensions, as in Green (2007). Since particles can in fact interpenetrate, each cell needs to have a capacity to hold more than 4 particles. The address of the cell is calculated from particle coordinates $(i, j, k) : i = \lfloor x/l \rfloor, j = \lfloor y/l \rfloor, k = \lfloor z/l \rfloor$ where (x, y, z) are particle's coordinates, l is the size of a cell edge and (i, j, k)

[1] This statement is relative to the today's hardware limitations. One can hope that this limitations will decrease dramatically in the future.

[2] Binary space partitioning. Space is divided recursively so that each division leaves equal number of elements on both sides of a separating hyperplane.

[3] Octal tree partitioning. Three dimensional space is divided recursively so that each subdivision creates eight cuboids from one.

is address of a cell as described in Matthias Teschner (2003).

The main disadvantage of the presented method is the limited space in which particles can exist, which is defined by the size of the grid. Spatial hashing solves this problem by making the available space infinite and it just adds one additional step in the algorithm for the uniform grid which is a hash function. This function takes three numbers for input, namely (i, j, k) and calculates a hash value (one integer) from it. The one used in my implementation is copied from Green (2008). This method has its disadvantages since multiple cells with different (i, j, k) indices now can have the same resulting hash value and therefore write in the same memory allocated for just one cell. This is partially handled by increasing the available memory (maximum number of particles in each cell) and by making sure the grid is not constructed around the origin of the simulation coordinate system (such grids have poor performance while using hash from Green (2008)).

Each cell has an array of some predefined size (*maxParticlesPerCell*) where it contains indices of particles that are located in this cell. There is one additional value that counts the number of particles that try to store their index inside a cell they calculate is the one that contains them. Since each particle is processed in parallel on GPU, some mutual exclusion will be required, namely atomic operations ($atomicAdd()$). Full spatial hashing is presented in algorithm 1.

Algorithm 1 Spatial hashing

1: **for all** p in $particles$ **do** :
2: $(x, y, z) \leftarrow particlePositions[p]$
3: $i \leftarrow \lfloor x/l \rfloor, j \leftarrow \lfloor y/l \rfloor, k \leftarrow \lfloor z/l \rfloor$
4: $index \leftarrow hash(i, j, k)$
5: $count \leftarrow atomicAdd(cellParticleCounts[index], 1)$ ▷ Get the position to store the particle ID and increment the count
6: **if** $count < maxParticlesPerCell$ **then**
7: $cellParticleCounts[index][count - 1] \leftarrow particleIDs[i]$
8: **end if**
9: **end for**

3. The main simulation loop

In this Chapter, the thesis will go through data representation of particles and forces that is designed to be suitable for position based dynamics (PBD).

3.1. Particle data structure

Each particle is represented by listing 3.1 and there is a buffer that contains an array of this structure. The length of this array is chosen to be one hundred thousand, which is the maximum number of particles as explained in chapter 2.

```
struct Particle
{
    float3 position;
    float3 position_predicted;
    float3 velocity;
    float mass;
    uint code;
    float specialParameters[SPECK_SPECIAL_PARAM_N];
};
```

Listing 3.1: Particle structure

The most important information of every particle is its *position*. Everything else can be omitted for a sufficiently simple system. This information is stored in a *float3*[1] type, which holds three coordinates for a single particle (x, y, z) in floating point precision (32-bits). Next members are *position_predicted*, *velocity* and *mass* which use will be explained later. Particles can behave in different ways, for example rigid body particles will have some properties fluid particles dont. This information is coded into *code* variable. Dependent on particle's type, the *specialParameters* have different uses (e.g. cohesion, adhesion and viscosity).

[1]This is a type of data for a three dimensional vector in High Level Shader Language (HLSL). This language is used alongside Microsoft's DirectX API.

3.2. Force data structure

Forces are given in listing 3.2. This structure is also contained in an array since there could be multiple forces. The buffer that holds this array is different from the one for the particles since this one does not change on the GPU side. The host[2] constructs the forces array and sends it to the device[3] that treats it like a constant.

```
struct ExternalForce
{
        float3 vec;
        int type;
};
```

Listing 3.2: Forces structure

Vector *vec* is used based on *type*. The only force type needed for this implementation was gravity where *vec* represents the direction of it. Other force types could be implemented with ease (e.g. repulsive or attractive point forces).

3.3. Position based dynamics

As in Müller et al. (2006), there are certain advantages of PBD over the traditional approach in simulating dynamic objects:

· Removes typical instability problems.

· Enables direct interaction with objects in simulation.

· Easy to understand and implement.

· Uniform constraints handling.

The simulation loop in algorithm 2 is taken from Macklin et al. (2014). They also included something they call *mass scaling* as a solution to the interpenetration resolution inefficiencies when there are bigger number of particles stacked one on top of another. This step is not included in this demo due to certain inconveniences, but the reader is welcome to try adding it.

[2]Common name for system (CPU and RAM) in GPGPU.

[3]Common name for a graphics card in GPGPU.

Algorithm 2 Simulation loop

for all p in $particles$ **do** :
    for all f in $forces$ **do** :
        $\mathbf{v}_p \leftarrow \mathbf{v}_p + \Delta t \mathbf{f}_f$
    end for
    $\mathbf{x}_p^* \leftarrow \mathbf{x}_p + \Delta t \mathbf{v}_p$
end for
for all p in $particles$ **do** :
    findNeighbouringParticles
    findSolidContact
end for
for $stabilizationIterations$ **do** :
    for all p in $particles$ **do** :
        $\Delta \mathbf{x} \leftarrow \mathbf{0}, n \leftarrow 0$
        $\Delta \mathbf{x}, n \leftarrow$ solveContactConstraints
        $\mathbf{x}_p^* \leftarrow \mathbf{x}_p^* + \Delta \mathbf{x}/n$
        $\mathbf{x}_p \leftarrow \mathbf{x}_p + \Delta \mathbf{x}/n$
    end for
end for
for $solverIterations$ **do** :
    for all p in $particles$ **do** :
        $\Delta \mathbf{x} \leftarrow \mathbf{0}, n \leftarrow 0$
        $\Delta \mathbf{x}, n \leftarrow$ solveAllConstraints
        $\mathbf{x}_p^* \leftarrow \mathbf{x}_p^* + \Delta \mathbf{x}/n$
    end for
end for
for all p in $particles$ **do** :
    $\mathbf{v}_p \leftarrow \frac{1}{\Delta t}(\mathbf{x}_p^* - \mathbf{x}_p)$
    $\mathbf{x}_p \leftarrow \mathbf{x}_p^*$ or apply sleeping
end for

The simulation step starts with PBD integration, where forces update particle velocities and then based on new velocities, predicted positions ($\mathbf{x}_p^*$) are calculated. Next, neighbouring particles are found. This process comes down to calculating the containing cell of a particle and then fetching all the particles in that cell and the neighbouring 26 cells. This area is visualized in figure 3.1. Since all cells require hash calculation

for index, their indices are usually cached per particle.

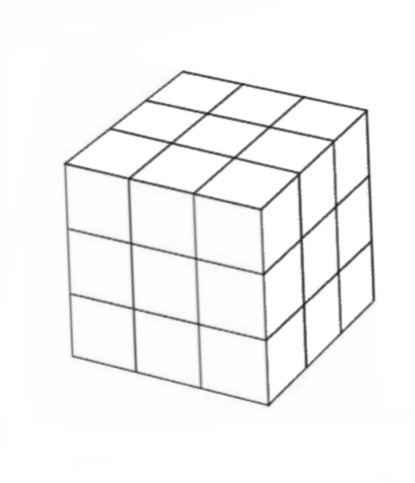

Figure 3.1: Neighborhood of a cell that needs to be checked for containing particles.

Constraints in PDB are given by equations 3.1 and 3.2 where $\mathbf{x} = [x_1, x_2, ..., x_n,]^T$ is a vector of positions from all particles contained in a constraint. Inequality constraints are used typically for penetration where update is required only on one side of the collision surface. Equality, for example, could be used in modeling springs in a cloth simulation.

$$C_i(\mathbf{x} + \Delta\mathbf{x}) = 0, i = 1, ..., n \tag{3.1}$$

$$C_j(\mathbf{x} + \Delta\mathbf{x}) \geq 0, j = 1, ..., n \tag{3.2}$$

Solution to this problem is usually solved in Gauss-Seidel-type iteration, but considering GPU is processing particles in parallel, Gauss-Jacobi-type is more suited to the problem. The difference between the two is in that the first one calculates new solutions to each of its equations (constraints) in sequence and immediately uses the newly calculated solutions. Second iteration uses the newly calculated solutions after they have all been calculated (from the old ones). This is more efficient to implement on a parallel system since data is read from one address and written to another in a single iteration, after which a synchronization barrier is used. Note that both of those methods are used in solving linear systems. Although majority of constraints are not such systems, most authors still use these names.

As in Müller et al. (2006) constraint equations give us equation 3.3 for updating particle position.

$$\Delta \mathbf{x}_i = -s \nabla_{\mathbf{x}_i} C(\mathbf{x}_1, ..., \mathbf{x}_n) \tag{3.3}$$

$$s = \frac{C_i(\mathbf{x}_1, ..., \mathbf{x}_n)}{\sum w_j \mid \nabla_{\mathbf{x}_j} C(\mathbf{x}_1, ..., \mathbf{x}_n) \mid^2} \tag{3.4}$$

Where equation 3.4 is the same for all particles in the constraint and $w_i = 1/m_i$ is inverse mass of a particle. Example of this general formula in use is distance constraint $C(\mathbf{x}_1, \mathbf{x}_2) = \mid \mathbf{x}_1 - \mathbf{x}_2 \mid -d \geq 0$ between two particles. Using 3.3 yields:

$$\Delta \mathbf{x}_1 = -\frac{\mathbf{w}_1}{\mathbf{w}_1 + \mathbf{w}_2}(\mid \mathbf{x}_1 - \mathbf{x}_2 \mid -d)\frac{\mathbf{x}_1 - \mathbf{x}_2}{\mid \mathbf{x}_1 - \mathbf{x}_2 \mid} \tag{3.5}$$

$$\Delta \mathbf{x}_2 = +\frac{\mathbf{w}_2}{\mathbf{w}_1 + \mathbf{w}_2}(\mid \mathbf{x}_1 - \mathbf{x}_2 \mid -d)\frac{\mathbf{x}_1 - \mathbf{x}_2}{\mid \mathbf{x}_1 - \mathbf{x}_2 \mid} \tag{3.6}$$

Where d is the maximum distance between particles. Both 3.5 and 3.6 are inequality constraints, so the $\Delta \mathbf{x}$ is not zero only if $\mid \mathbf{x}_1 - \mathbf{x}_2 \mid$ is lesser than d. Usually while generating contacts, tolerance is a bit higher for something to be added as a contact since constraint resolution can make something previously regarded as non violated constraint into violated one. In this implementation a factor of 1.2 is used as a multiplier of the particle double radius which is the area of contact creation. There is one more distance constraint needed and that is the one between a particle and a plane. Its given by $C(\mathbf{x}_1) = (\mathbf{x}_1 - \mathbf{p})\hat{\mathbf{n}} - d \geq 0$. The particle position update is:

$$\Delta \mathbf{x} = [(\mathbf{x} - \mathbf{p})\hat{\mathbf{n}} - d]\,\hat{\mathbf{n}} \tag{3.7}$$

Where $\mathbf{p}$ is some arbitrary position on a plane, $\hat{\mathbf{n}}$ is plane normal vector and d is the maximum allowed distance from the plane.

With this math out of the way, steps from lines 11 to 18 in algorithm 2 can be explained. These steps work only with the constraints presented so far (distance constraints). This is not part of the main solver, as this loop only serves to validate certain states without adding unwanted energy to the system. It is called stabilization pass and the problem it solves is presented in figure 3.2. Namely when, for example, two particles end up interpenetrating (distance constraints being violated) either by initial conditions, user interaction or if convergence was not met in the beginning of the simulation step, resolving the interpenetration will result in additional velocity. This is because changing only the predicted position of a particle results in a bigger difference between the predicted and old positions which increases velocity. Therefore in

stabilization pass, both positions are being updated so the difference is zero.

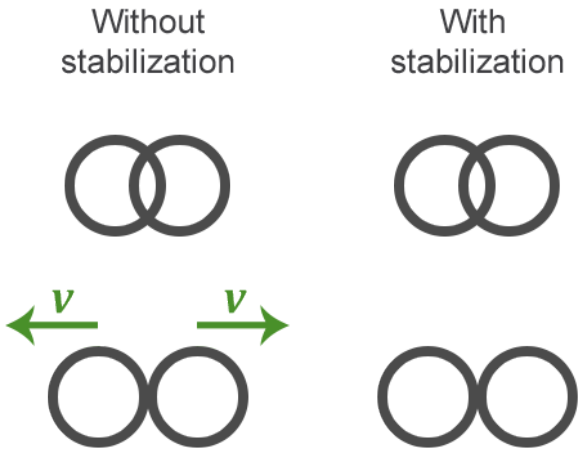

Figure 3.2: Resolving invalid state without adding any unnecessary energy to the system (right). Both pairs have zero initial velocities.

In stabilization iterations, total position change $\Delta \mathbf{x}$ is not applied directly, but divided by n (number of constraints violated). This is a common practice in physics simulation called relaxation. It sacrifices convergence speed for stability and sometimes even solves situations where convergence is impossible otherwise as is described in Macklin et al. (2014).

Lines from 19 to 25 in algorithm 2 are the main loop of the simulation where all constraints are processed. Some forces like cohesion and adhesion for fluids are also processed in this loop for simplicity. Friction can also be found here and it is implemented as in Macklin et al. (2014).

Lines from 26 to 29 update particle positions and are a part of the final step. Notice also that here, velocity is being calculated as a difference in position and at the beginning of the simulation loop it is used as a prediction for a new particle position. So it only exists to compensate for any delta time changes, everything else is dependent on positions, hence the name position based dynamics. Sleeping is also included in this final part. This is popular method for reaching convergence (or at least what seems like convergence). Once a particle has velocity below a certain threshold, it is simply frozen (no positional update is applied) as in equation 3.8.

$$\mathbf{x}(t+\Delta t) = \begin{cases} \mathbf{x}^*, & \text{if } |\mathbf{x}^* - \mathbf{x_0}| > \varepsilon \\ \mathbf{x}_0, & \text{otherwise} \end{cases} \tag{3.8}$$

One additional and easy to implement extension to the algorithm 2 is to divide delta time by n and then run the algorithm n times. This makes the simulation more precise as multiple steps are calculated before each frame presentation. It is recommended to utilize this, especially if the frame time of the application is bigger than the refreshing rate of the monitor.

4. Applications of the uniform particle simulator

So far this thesis has presented the base of the implementation. Next steps will show how this way of handling particles can be used to simulate behaviour of simple heaps like in figure 4.1, fluids, rigid bodies and ultimately joints between rigid bodies.

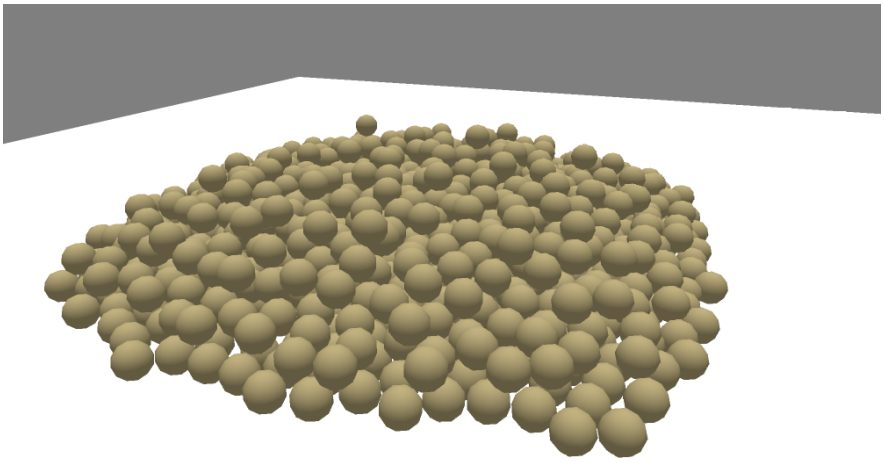

Figure 4.1: "Sand" particles in a heap using the system presented so far.

4.1. Fluids

Fluid simulation has long been a field of interest for many graphical researches. Its use in film and interactive environments, over the years, was the main driving force behind the development of newer, more optimized and visually rich methods. Approach used here, which is also very suitable for the PBD system described so far, is called smoothed particle hydrodynamic SPH (Monaghan (1992)).

Fluids are usually simulated in one of two ways:

· **The Lagrangian method** which calculates properties of each particle based on its

position in space and neighbours.

· **The Eulerian method** is a grid based method which calculates properties over a set of fixed positions in space.

The first one is used in this implementation since it is more suitable for PBD and has some additional advantages. For example the computation is only performed where necessary (this also positively affects the required storage) and particles are not restricted to move inside a finite box in space. The only disadvantage of Lagrangian method is that it requires a relatively large amount of particles to obtain realistic results, but this is feasible due to the possibility of parallel execution on today's GPU hardware.

To enforce incomprehensibility each particle is given a density constraint which is defined as

$$C_i(\mathbf{x}_1, ..., \mathbf{x}_n) = \frac{\rho_i}{\rho_0} - 1 \leq 0 \tag{4.1}$$

Where ρ_0 is the rest density

$$\rho_0 = \frac{m}{V} = \frac{3m}{4r^3\pi} \tag{4.2}$$

This is a density formula of a full sphere with equally distributed mass where r is the radius of the sphere and m is its mass. This is calculated per fluid since inside a fluid all particles have equal mass and volume. The value of ρ_i from equation 4.1 is given by

$$\rho_i = \sum_j m_j W(\mathbf{x}_i - \mathbf{x}_j, h) \tag{4.3}$$

W is a kernel function where h is called core radius. In this implementation it is chosen to be a little bit bigger than the double radius of a particle $h = 1.2(2r)$, same as contact constraints in 3.3. Poly6 kernel is used for density estimation and Spiky kernel for gradient calculation. Both of those kernels are from Müller et al. (2003). Following the steps from Müller et al. (2006) position update is given as

$$\Delta \mathbf{x}_i = \frac{1}{\rho_0} \sum_j (\lambda_i + \lambda_j) m_j \nabla W(\mathbf{x}_i - \mathbf{x}_j, h) \tag{4.4}$$

$$\lambda_i = -\frac{C_i(\mathbf{x_1}, ..., \mathbf{x_n})}{\sum_k \mid \nabla_{\mathbf{x}_k} C_i \mid^2 + \epsilon} \tag{4.5}$$

$$\nabla_{\mathbf{x}_k} C_i = \frac{1}{\rho_0} \begin{cases} \sum_j m_j \nabla_{\mathbf{x}_k} W(\mathbf{x}_i - \mathbf{x}_j, h), & \text{if } k = i \\ -m_j \nabla_{\mathbf{x}_k} W(\mathbf{x}_i - \mathbf{x}_j, h), & \text{if } k = j \end{cases} \tag{4.6}$$

This is similar with the Macklin and Müller (2013) with the difference of mass being included in the equations. In their work, all particles had the same mass since they were simulating water, so they simply removed particle's mass from their equations. When the system had bigger particle radius, bigger ϵ[1] had to be chosen.

Viscosity is important for coherent motion, so a simplified version of XSPH from Schechter and Bridson (2012) is applied and integrated $\Delta \mathbf{x}_i^{new} = \mathbf{x}_i + \Delta t \mathbf{v}_i^{new}$, where c is viscosity constant and $\mathbf{v}_i^{new}$ is:

$$\mathbf{v}_i^{new} = \mathbf{v}_i^* + c \sum_j \frac{w_i}{w_i + w_j} (\mathbf{v}_j^* - \mathbf{v}_i^*) W(\mathbf{x}_i - \mathbf{x}_j, h) \tag{4.7}$$

This is all the math necessary to enforce incomprehensibility and stable behaviour of the simulation, but there is a lot more that can be added to increase realism and visual fidelity. Cohesion is one of predominant effects in fluid simulation and the method used here is from Akinci et al. (2013). Note that like in Macklin et al. (2014), only pairs of particles that are of the same type should have cohesion and surface tension applied to them. All these additional constants for viscosity and cohesion are stored in $specialParameter$ from listing 3.1

4.2. Rigid Bodies

Previous section, although helping with demonstration of particle interaction, did not have any integral importance to the problem of skeletal animation. Rigid bodies, on the other hand, represent bones in a skeletal model, so their implementation is crucial. Method from Macklin et al. (2014) is based on transformation calculation which is applied to a set of particles, after each simulation step (see figure 4.2).

[1] User specified relaxation parameter that is constant over the simulation.

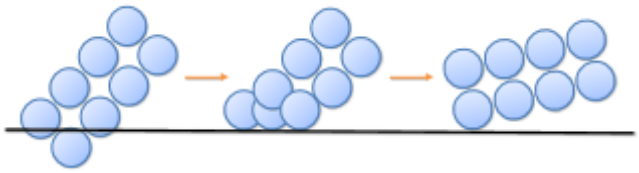

Figure 4.2: Taken from Macklin et al. (2014) this figure shows rigid body shape matching constraint.

$$\Delta \mathbf{x}_i = (\mathbf{Q}\mathbf{r}_i + \mathbf{c}) - \mathbf{x}_i^* \tag{4.8}$$

Where $\mathbf{Q}$ is rigid body rotation matrix and c is its center. Vector $\mathbf{r}_i$ is particle position in local coordinate space of the rigid body (where the center of mass is in the origin of the coordinate system). The trickiest part in this equation is calculating the rotation matrix $\mathbf{Q}$. It is a product of polar decomposition of matrix $\mathbf{A}$ given by:

$$\mathbf{A} = \sum_i (\mathbf{x}_i^* - \mathbf{c})\mathbf{r}_i^T \tag{4.9}$$

In Macklin et al. (2014) they evaluate eq. 4.9 efficiently by assigning a thread per particle, calculating all outer product separately and then using parallel reduction to sum all the products. This is similar to what is being used here. The later need for joints yields a slightly different approach. In listing 4.1 a new structure is introduced. This structure allows the linking of particles to the rigid bodies they make, so multiple particles can be linked to the same rigid body, as is expected. In addition to that, with this structure, a single particle can be linked to multiple rigid bodies, which will be utilized later with joints.

```
struct ParticleRigidBodyLink
{
        uint particleIndex;
        uint rbIndex;
        float3 posInRigidBody;
        uint particleLinksBlockStart;
        uint particleLinksBlockCount;
};
```

Listing 4.1: Particle rigid body link structure

Notice that *posInRigidBody* from listing 4.1 is $\mathbf{r}_i$ from eq. 4.8. The two values

that haven't been discussed before are *particleLinksBlockStart* and *particleLinksBlockCount*. These two define a block of links in the array (buffer) for the same rigid body. In other words, links are organized so that they are grouped by the rigid bodies they are assigned to (see figure 4.3).

Figure 4.3: Example of the buffer that holds particle - rigid body links. First 6 elements are for rigid body with index 2 and the remaining 4 are for the one with index 3.

Some storage is required to calculate $\mathbf{c}$ and $\mathbf{A}$, so an additional buffer is supplied. This buffer has an element for each link element in lising 4.1.

```
struct ParticleRigidBodyLinkCache
{
        float3 cm;
        float m;
        float3x3 A;
};
```

Listing 4.2: Particle rigid body link structure

The parallel reduce is performed over the array of *ParticleRigidBodyLink* elements in several steps. The first step calculates the value in question (e.g. $\mathbf{c}_i$). Then in each subsequent step, the algorithm adds the value with the value of the neighbour that is $2^{iteration-1}$ elements from the element being processed. In the last step, the total sum is stored in the first element of the block. The number of steps in this algorithm is calculated from the most complex rigid body (the one that has most particles). This number of steps is enough for all blocks of links (rigid bodies).

$$numberOfSteps = \lceil log_2(biggestRigidBodyParticleCount) \rceil + 1 \qquad (4.10)$$

This type of reduction is performed two times, once for $\mathbf{c}$ calculation and once for $\mathbf{A}$ (which requires precomputed $\mathbf{c}$). Also, note the $+1$ in eq. 4.10. This is the first step to calculate the initial value per link before the actual reduction begins.

Once $\mathbf{A}$ has been calculated, construction of $\mathbf{Q}$ can begin. This process requires polar decomposition. Although there have been recent publications that describe more

efficient ways of implementing this (see Higham and Noferini (2016)), method implemented here is efficient enough and quite simpler. In polar decomposition $\mathbf{Q}$ is orthogonal and $\mathbf{S}$ is symmetrical matrix. This also extends to complex numbers, but that is out of our concern.

$$\mathbf{A} = \mathbf{QS} \tag{4.11}$$

$$\mathbf{A}^T\mathbf{A} = \mathbf{S}^T\mathbf{Q}^T\mathbf{QS} = \mathbf{S}^2 = \mathbf{M} \tag{4.12}$$

$$\mathbf{S} = \sqrt{\mathbf{M}} \tag{4.13}$$

$$\mathbf{S}^{-1} = \mathbf{M}^{-\frac{1}{2}} \tag{4.14}$$

$$\mathbf{Q} = \mathbf{AS}^{-1} = \mathbf{AM}^{-\frac{1}{2}} \tag{4.15}$$

So the problem of polar decomposition comes down to finding $\mathbf{M}^{-\frac{1}{2}}$. For this, eigendecomposition of $\mathbf{M}$ is required. The algorithm used here is called QR algorithm which decomposes a matrix in a series of QR decompositions[2]. Equation 4.12 tells us that $\mathbf{M}$ is symmetric and because of that, the QR algorithm yields not only eigenvalues[3] $\mathbf{\Lambda}$, but also an orthogonal eigenvector basis $\mathbf{E}$.

$$\mathbf{M} = \mathbf{E\Lambda E}^{-1} = \mathbf{E\Lambda E}^T \tag{4.16}$$

$$\mathbf{M}^{-\frac{1}{2}} = \mathbf{E\Lambda}^{-\frac{1}{2}}\mathbf{E}^T \tag{4.17}$$

Since $\mathbf{\Lambda}$ is diagonal, $\mathbf{\Lambda}^{-\frac{1}{2}}$ is easy to calculate. Now finally everything is ready to get $\mathbf{Q}$, and with it transform the particles in the rigid body.

There is an issue with tunneling that comes up using this system. An example is visualized in figure 4.4 where a particle of water has a velocity high enough to get stuck between the cells of a rigid body. Each cell is pushing the particle away from itself and as a result, a water particle cannot escape. To solve this, a signed distance field (SDF) is constructed, the same approach as in Macklin et al. (2014). Method for SDF gradient used in this implementation is given by 4.18

$$\nabla\phi_i = \sum_j \frac{\mathbf{x}_i - \mathbf{x}_j}{\mid \mathbf{x}_i - \mathbf{x}_j \mid^k} \tag{4.18}$$

Also, Macklin et al. (2014) stores both, magnitude ϕ and gradient $\nabla\phi$ with the

[2]For this purpose, Householder method is used.

[3]$\mathbf{\Lambda}$ is diagonal matrix whose elements are eigenvalues.

particle. In this implementation storing just the gradient is enough. Its length, however, has a coded value. It can be zero for isolated particles of a rigid body (like joints) or one and two for internal or boundary particles.

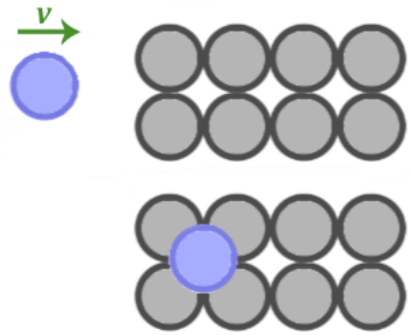

Figure 4.4: Tunneling of a water particle inside a rigid body.

The engine so far can support simulations of sand, water and rigid bodies as can be seen in figure 4.5. What is interesting, buoyancy occurs as different masses are assigned to different particles. This phenomenon does not need any special handling.

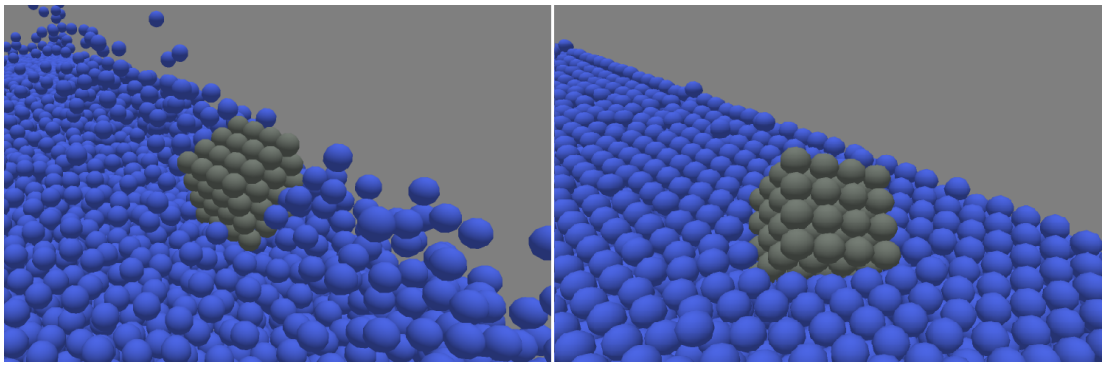

Figure 4.5: Fluid and rigid body interaction in the system built so far. Notice how buoyancy is keeping the rigid body partially submerged.

The system is now ready for the implementation of the final feature.

4.3. Joints implementation

Recall from listing 4.1 that each particle can be linked to multiple rigid bodies. This enables us to set up a configuration like in figure 4.6. Here, the red particle is of type joint and as such is a member of both rigid bodies, the left and the right one.

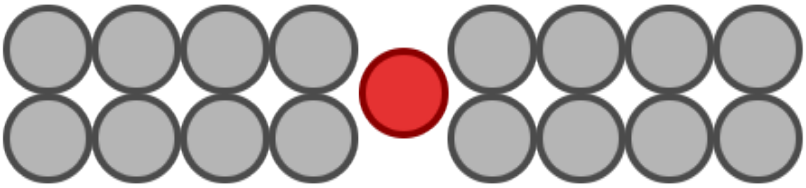

Figure 4.6: A simple 2D joint where movement in two directions is allowed.

When the rigid body constraint system tries to keep the initial configuration of a particle that has, for example, two rigid bodies, the particle ends up in a position that is an arithmetic mean of the two positions for that particles given by the two transformation matrices of the respective rigid bodies. If there are any differences between those positions, over the course of the simulation the rigid bodies will align themselves so that the configuration is in an acceptable state.

Two dimensional representation can serve well since a three-dimensional case follows the same principle. If we imagine that there is another layer of particles behind the one on the figure 4.6 in three dimensions then we would end up with something like a hinge joint (4.7). This joint has only one degree of freedom and the angle of the allowed bend is defined by the shape of the included rigid bodies.

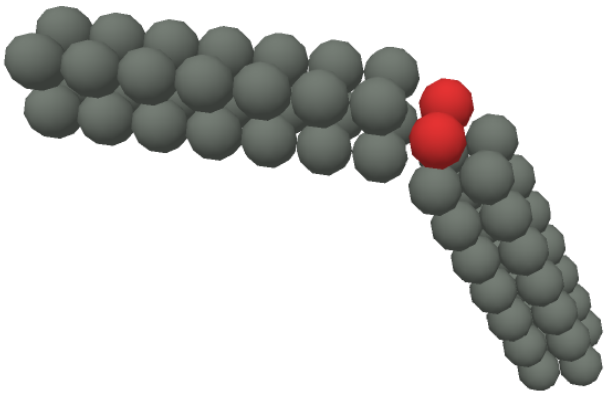

Figure 4.7: Extension of the simple 2D joint to three-dimensional hinge joint. Left hand sided rigid body has a locked transformation (it cannot rotate or translate) and the right hand sided one is being affected by gravity and therefore pulled down.

Hinge joint can be modeled further by adding an angular movement constraint to achieve a behaviour one would expect in a knee. This is done by simply putting two particles as part of one rigid body in between that rigid body and the one connected by

joint, as seen in figure 4.8. Now, rigid bodies will be able to rotate only on one side of the joint.

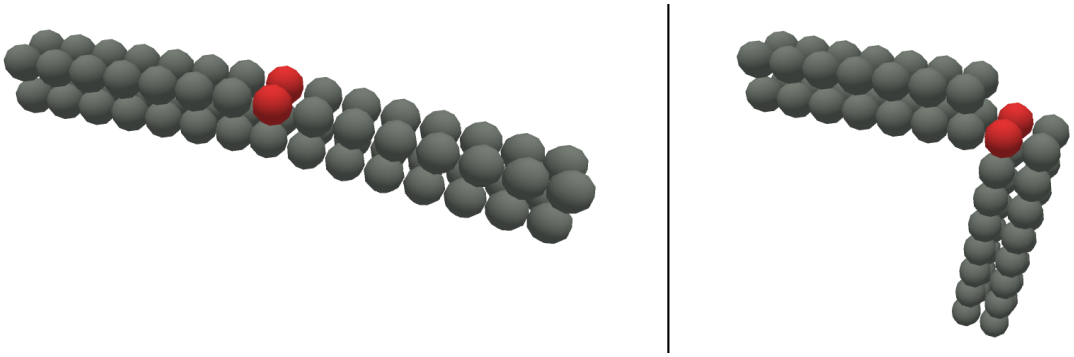

Figure 4.8: Modeling a knee-like joint. Like in figure 4.7, left-hand-sided rigid body is locked and rotated by 180 degrees from left to right images. The other rigid body is being affected by gravity in both pictures.

But those are just examples of what can be achieved with this system. The user is welcome to try and implement her or his own configurations. Combining these can result in a more complex system such as a 'ragdoll' in figure 1.1

5. Implementation details and possible improvements

The whole engine is built on top of DirectX 12 API, which enables it to run on Nvidia and, unlike Flex, AMD hardware. For all the work on the GPU side, compute shader with shader model 5.1 was used. Apart from STL[1], a library from Autodesk was integrated into the solution. This is FBX SDK and it includes a parser for FBX file format that was required to load the skeletal model and animations for it.

As stated in chapter 2, GPU cache misses are the main concern in the implementation. This could be alleviated by keeping the particle data in a texture. That way, two-dimensional cache of modern GPU architectures would decrease the amount of cache misses. Performance increase of 15% was observed in a similar attempt with CUDA[2] by Green (2007).

There are many features left in Flex that weren't implemented here. Deformable bodies, cloth and smoke for example. Existing features could also be improved. Rigid bodies could be made destructible and water could use vorticity confinement. Perhaps one of the more radical changes would be to add varying particle sizes. The rule of largest difference between object sizes $1 : 10$ from Flex would then be irrelevant and new possibilities would arise.

Rendering in this implementation is rudimentary and serves the purpose of demonstrating the physics simulation. In a commercial project, this would be replaced with more appealing graphics. Rendering skinned meshes and rigid bodies should be straightforward since all the transformation matrices are already in buffers on the GPU. Fluids and effects like foam could also be added.

[1] Standard template library. A C++ library that is part of the standard and comes with every modern compiler.

[2] A parallel computing platform and application programming interface (API) model created by Nvidia.

6. Conclusion

The system built throughout this thesis enables us to greatly alleviate the work on the CPU by migrating the rigid body and ragdoll physics processing to the GPU. This way the CPU can spend its time to do other work that is much harder if not impossible to implement on the graphics hardware. To do this as a uniform particle simulation also allows for interaction with other types of bodies (e.g. fluids). The biggest utilization of this implementation would be in real time interactive applications. Entertainment software is usually built in such a way and would benefit from implementations presented here.

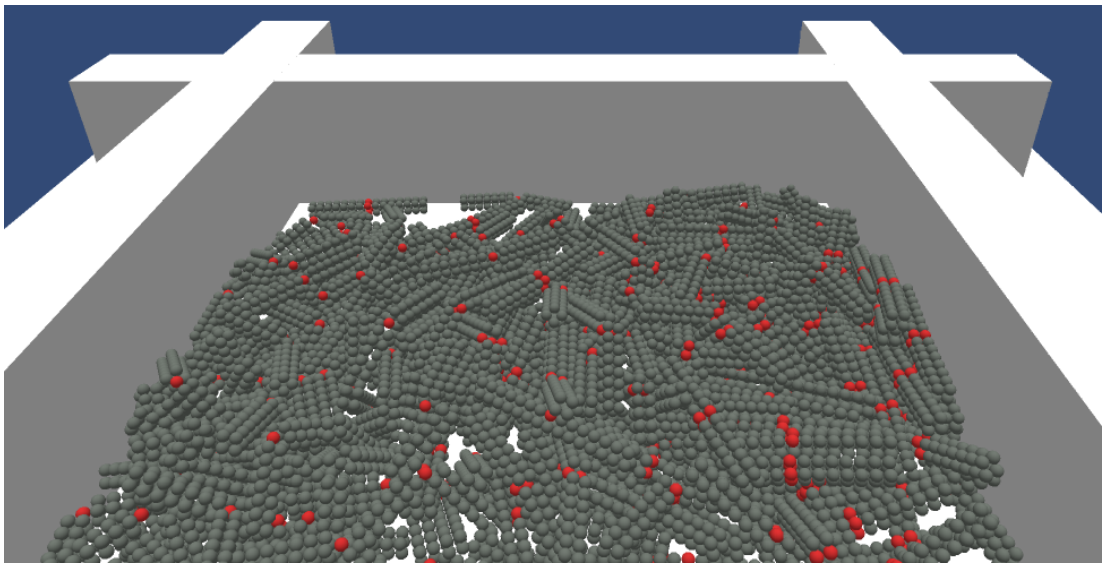

Figure 6.1: There are 270 rigid body pairs connected with joints (540 rigid bodies) on this figure. The application runs on GTX 970 at over 60 frames per second

BIBLIOGRAPHY

Nadir Akinci, Gizem Akinci, and Matthias Teschner. Versatile surface tension and adhesion for sph fluids. *ACM Trans. Graph.*, 32(6):182:1–182:8, November 2013. ISSN 0730-0301. doi: 10.1145/2508363.2508395. URL http://doi.acm.org/10.1145/2508363.2508395.

Simon Green. Cuda particles. In *CUDA Particles*, November 2007.

Simon Green. Particle-based fluid simulation. In *Particle-based Fluid Simulation*, February 2008.

Nicholas J Higham and Vanni Noferini. An algorithm to compute the polar decomposition of a 3× 3 matrix. *Numerical Algorithms*, 73(2):349–369, 2016.

Miles Macklin and Matthias Müller. Position based fluids. *ACM Trans. Graph.*, 32(4): 104:1–104:12, July 2013. ISSN 0730-0301. doi: 10.1145/2461912.2461984. URL http://doi.acm.org/10.1145/2461912.2461984.

Miles Macklin, Matthias Müller, Nuttapong Chentanez, and Tae-Yong Kim. Unified particle physics for real-time applications. *ACM Trans. Graph.*, 33(4):153:1–153:12, July 2014. ISSN 0730-0301. doi: 10.1145/2601097.2601152. URL http://doi.acm.org/10.1145/2601097.2601152.

Matthias Müller Markus Gross Danat Pomeranets Matthias Teschner, Bruno Heidelberger. Optimized spatial hashing for collision detection of deformable objects. *Proceedings of Vision, Modeling, Visualization VMV'03*, 2003.

Joe J Monaghan. Smoothed particle hydrodynamics. *Annual review of astronomy and astrophysics*, 30(1):543–574, 1992.

Matthias Müller, David Charypar, and Markus Gross. Particle-based fluid simulation for interactive applications. In *Proceedings of the 2003 ACM SIGGRAPH/Euro-*

graphics symposium on Computer animation, pages 154–159. Eurographics Association, 2003.

Matthias Müller, Bruno Heidelberger, Marcus Hennix, and John Ratcliff. Position based dynamics. *Journal of Visual Communication and Image Representation*, 2006.

Hagit Schechter and Robert Bridson. Ghost sph for animating water. *ACM Trans. Graph.*, 31(4):61:1–61:8, July 2012. ISSN 0730-0301. doi: 10.1145/2185520.2185557. URL `http://doi.acm.org/10.1145/2185520.2185557`.

Abstract

This thesis presents an implementation of a system similar to Nvidia Flex. It improves upon Flex, with its implementation of joints. This feature can be utilized to construct a 'ragdoll' (a popular physical representation of a skeletal system) on the GPU, therefore alleviating the work on the CPU side. Also, unlike Flex, which requires a Nvidia GPU, this thesis and its implementation have been built with graphics hardware agnosticism. Although not every feature is implemented, features that are, do have more elaboration.

Keywords: particle, physics, GPGPU, simulation, fluid simulation, unified solver, position-based dynamics

Sažetak

Ova teza prezentira implementaciju sustava sličnog Nvidia Flex-u. Nadograđuje se na taj sustav sa implementacijom zglobova. Ovo svojstvo se moze iskoristiti za konstrukciju "krpene lutke" (popularnog sustava za prikazivanje fizicki pogonjene skeletalne animacije) na grafičkoj kartici, te s time ublažiti posao dodijeljen CPU. Također, za razliku od Flexa, kojemu treba Nvidijin GPU, ova teza i njezina implementacija je građena za grafičke kartice proizvoljnog proizvođača. Iako svako svojstvo Flexa nije uključeno, ona koja jesu su opsežnije objašnjena.

Ključne riječi: čestice; fizika; GPGPU; simulacija; simulacija fluida; unificirani rijesavac; dinamika temeljena na poziciji