Rigid Body Joints in Real-Time Unified Particle Physics

B. Lovrovic[1,2] and Z. Mihajlovic[1]

[1]University of Zagreb, Faculty of Electrical Engineering and Computing, Croatia
[2]The Future Group AS, Norway

Abstract

In this paper, we propose a physically-based method for a rigid body joint simulation. The proposed solution is based on the unified particle physics engine, a simulator that uses only particles for all the dynamic bodies. Such engines are implemented on the GPU and they simulate fluids, rigid bodies or deform-able materials like cloth or ropes. To support more complex systems like skeletal simulation, we show a joint implementation that is intuitive and unique to this environment. Four types of joints will be shown, as well as the necessary details about the rigid body data structure. This will enable the construction of a popular method called ragdoll. Lastly, a performance measurement and a comparison with alternatives will be given.

CCS Concepts
•***Computing methodologies*** $\rightarrow$ *Massively parallel and high-performance simulations; Real-time simulation; Interactive simulation; Physical simulation; Collision detection;*

1. Introduction

Offloading the work from the central processing unit (CPU) to the graphics processing unit (GPU) can often be desirable in modern, real-time computer graphics applications. From cellular automata [OLG*07] to fluids, majority of those result in particles-based (Lagrangian), or grid-based (Eulerian) solutions that can run in parallel on many processors, and therefore, are suitable to be executed on the GPU. An attractive property observed in Lagrangian methods is unification of usually-separated physical simulators (e.g. fluid, cloth or rigid body simulators).

The idea presented in this paper is based on such a unification and extends it by adding rigid body joints, a movement constraint between two or more rigid bodies. There are many different types of joints and combining those can help to model complex physical structures like the human body. This method is referred to as 'ragdoll' and is commonly used in games and animated films. Our approach enables easier integration with the particle system, alleviates the work from the CPU and removes the need to send rigid body transformation matrices from the CPU to the GPU when rendering.

2. Related work

Processing physically-based simulations on the CPU is a well established field and authors like [Mil10], [Ebe10] and [Eri04] provide sufficient knowledge to implement most of all the well known simulation methods.

Running simulations on the GPU in a newer concept which is one of the reasons it is still being heavily researched. It is mostly used in fluid simulations such as Eulerian [Har05] or Lagrangian methods. Lagrangian methods come in traditional force-based dynamics (FBD) and newer, position-based dynamics (PBD) forms. Example of a PBD fluid is shown in [MM13], while a more general approach is given in [MHHR07]. As all matter consists of a collection of particles, it is intuitive that this approach can be extended to deformable and rigid bodies. Harada [Har07] and Macklin et al. [MMCK14] do this in FBD and PBD forms, respectively.

When it comes to the collision handling in a particle simulation, either event-driven (ED) or molecular dynamics (MD) algorithms are used. With MD it is easier to simulate the elastic deformation of the material elements, as in [BYM05]. MD is achieved here by using relaxation in the collision solver and by other methods that do not guarantee complete separation of particles at all times.

This paper uses methods presented in [MMCK14] as a foundation, which makes it a Lagrangian PBD solver. We choose PBD over a force-based solution for it's ability to interact with the simulation in real time without limitations, a feature valuable in interactive environments. Other than that, it is easier to understand and implement, it is more stable and allows for uniform constraint handling. For spatial hashing, hash function and methods from [THM*03] are used. With all particles being the same size, grid cell sizes are the same as those of the particle as in [Gre08]. Fluids, which are based on [MM13], and granular materials are included for the purpose of showing interaction with the joint-constrained rigid bodies.

DOI: 10.2312/egs.20181031

3. System overview

Rigid body simulation presented here is a modified version of that in [MMCK14]. An overview of the methods will be presented along with our modifications in the subsection 3.3.

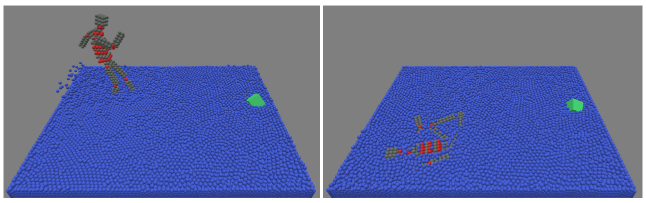

Figure 1: *An actor made up of particles is dancing and interacting with the environment (left image) and initiating simulation of its skeletal system (right image).*

3.1. The particle simulation update

Simulation loop from [MMCK14] performs the Euler integration and collision handling on every particle. Particles that reach sufficiently small sequential position change will be frozen. While working on the implementation, the best minimum change value for freezing emerged as half of the radius of the particle.

Usually while generating particle-particle 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's diameter which is the area of contact creation. Big factors can hinder performance.

3.2. PBD constraints

Constraints in PBD are given by equations (1) and (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 typically used for penetration in cases 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.

$$C_i(\mathbf{x}+\Delta\mathbf{x}) = 0, i = 1, ..., n \tag{1}$$
$$C_j(\mathbf{x}+\Delta\mathbf{x}) \geq 0, j = 1, ..., n \tag{2}$$

As in [MHHR07], the constraint equations give us eq. (3) for updating the position of a single particle $\mathbf{x}_i$. The Gauss-Jacobi method is used here to exploit the parallelism of the GPU.

$$\Delta\mathbf{x}_i = -s\nabla_{\mathbf{x}_i} C(\mathbf{x}_1, ..., \mathbf{x}_n) \tag{3}$$
$$s = \frac{C_i(\mathbf{x}_1, ..., \mathbf{x}_n)}{\sum w_j \left\|\nabla_{\mathbf{x}_j} C(\mathbf{x}_1, ..., \mathbf{x}_n)\right\|^2} \tag{4}$$

Where eq. (4) is the same for all particles in the constraint and $w_i = 1/m_i$ is the inverse mass of a particle. An example of this general formula in use is the distance constraint $C(\mathbf{x}_1, \mathbf{x}_2) = \|\mathbf{x}_1 - \mathbf{x}_2\| - d \geq 0$ between two particles. Using (3) yields:

$$\Delta\mathbf{x}_1 = -\frac{w_1}{w_1+w_2}(\|\mathbf{x}_1 - \mathbf{x}_2\| - d)\frac{\mathbf{x}_1 - \mathbf{x}_2}{\|\mathbf{x}_1 - \mathbf{x}_2\|} \tag{5}$$
$$\Delta\mathbf{x}_2 = \frac{w_2}{w_1+w_2}(\|\mathbf{x}_1 - \mathbf{x}_2\| - d)\frac{\mathbf{x}_1 - \mathbf{x}_2}{\|\mathbf{x}_1 - \mathbf{x}_2\|} \tag{6}$$

Where d is the maximum distance between particles. Both equations (5) and (6) represent inequality constraints, so $\Delta\mathbf{x}$ is not zero only if $\|\mathbf{x}_1 - \mathbf{x}_2\|$ is less than d. An additional distance constraint is the one between a particle and a plane. It is given by $C(\mathbf{x}_1) = (\mathbf{x}_1 - \mathbf{p}) \cdot \hat{\mathbf{n}} - d \geq 0$. The particle position update is:

$$\Delta\mathbf{x} = [(\mathbf{x}-\mathbf{p}) \cdot \hat{\mathbf{n}} - d]\,\hat{\mathbf{n}} \tag{7}$$

Where $\mathbf{p}$ is a point on the plane, $\hat{\mathbf{n}}$ is the plane's normal vector and d is the minimum allowed distance from the plane.

3.3. Rigid bodies implementation

By applying the shape matching method from [MHTG05], rigid bodies are handled as a constraint as well, with a solution given in eq. (8):

$$\Delta\mathbf{x}_i = (\mathbf{Q}\mathbf{r}_i + \mathbf{c}) - \mathbf{x}_i^* \tag{8}$$

Where $\mathbf{Q}$ is a rigid body rotation matrix and $\mathbf{c}$ is its center of mass. The vector $\mathbf{r}_i$ is the particle position in the local coordinate space of the rigid body. The trickiest part in this equation is to calculate the rotation matrix $\mathbf{Q}$, which can be done by polar decomposition of the matrix $\mathbf{A}$ given by:

$$\mathbf{A} = \sum_i (\mathbf{x}_i^* - \mathbf{c})\mathbf{r}_i^T \tag{9}$$

In [MMCK14] the authors evaluated eq. (9) efficiently by assigning one thread per particle, calculating all outer products separately and then using parallel reduction to sum all of them. However, the need for joints yields a slightly different approach. In listing 1 we introduce our structure. This structure allows linking 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.

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

Listing 1: *Particle rigid body link structure*

Note that *posInRigidBody* from listing 1 is $\mathbf{r}_i$ from eq. (8). Variables *particleLinksBlockStart* and *particleLinksBlockCount* define a block of links in the array for the same rigid body. This means that links are grouped by the rigid body they are assigned to. In table 1 the first four elements are for the rigid body with an index of 2 and the remaining three are for the one with an index of 3. Note that the particle with an index of 3 belongs to both rigid bodies. This is a format ready for a parallel reduction algorithm.

	particleIndex	rbIndex	blockInfo (start, count)
0	3	2	0, 4
1	1	2	0, 4
2	0	2	0, 4
3	4	2	0, 4
4	3	3	4, 3
5	8	3	4, 3
6	7	3	4, 3

Table 1: *Example of a valid array of ParticleRigidBodyLink elements.*

Some storage is required to calculate **c**, **A** and mass (a cache value). Those are stored in a separated buffer with one-to-one relation to the buffer from listing 1.

The parallel reduction algorithm 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 rigid body that has the most particles. This number of steps is enough for all blocks of links (rigid bodies).

$$numberOfSteps = \lceil log_2(p) \rceil + 1 \tag{10}$$

Where p is the number of particles contained in the biggest rigid body. This type of reduction is performed two times, once for **c** calculation and once for **A**, which requires **c** to be precomputed. Also, note the $+1$ in eq. (10). This is the first step to calculate the initial value per link before the actual reduction begins.

Once **A** has been calculated, construction of **Q** can begin. This process requires polar decomposition. Although there have been recent publications that describe more efficient ways of implementing such a decomposition for a 3x3 matrix (see [HN16]), the method shown here is efficient enough and simpler to implement. In polar decomposition, **Q** is an orthogonal and **S** is a symmetrical matrix.

$$\mathbf{A} = \mathbf{QS} \tag{11}$$

$$\mathbf{A}^T\mathbf{A} = \mathbf{S}^T\mathbf{Q}^T\mathbf{QS} = \mathbf{S}^2 = \mathbf{M} \tag{12}$$

$$\mathbf{S} = \sqrt{\mathbf{M}} \tag{13}$$

$$\mathbf{S}^{-1} = \mathbf{M}^{-\frac{1}{2}} \tag{14}$$

$$\mathbf{Q} = \mathbf{AS}^{-1} = \mathbf{AM}^{-\frac{1}{2}} \tag{15}$$

So the problem of polar decomposition comes down to finding $\mathbf{M}^{-\frac{1}{2}}$. For this, eigendecomposition of **M** is required. The algorithm used here is the QR algorithm which decomposes a matrix in a series of QR decompositions. Householder transformation is used for each decomposition. Equation (12) shows that **M** is symmetric and because of that, the QR algorithm yields, not only eigenvalues (non-zero elements of the diagonal matrix λ), but also an orthogonal eigenvector basis **E**.

$$\mathbf{M} = \mathbf{E}\lambda\mathbf{E}^{-1} = \mathbf{E}\lambda\mathbf{E}^T \tag{16}$$

$$\mathbf{M}^{-\frac{1}{2}} = \mathbf{E}\lambda^{-\frac{1}{2}}\mathbf{E}^T \tag{17}$$

Since λ is diagonal (non-zero elements are only in the diagonal running from the upper left to the lower right), $\lambda^{-\frac{1}{2}}$ is easy to calculate as it performs the operation on every non-zero element separately.

After allowing particles to be affected by all the constraints and then updating them with the eq. (8) to regain their configuration as a last step, the resulting behaviour will mimic that of a rigid body.

Rigid bodies that have all of their particles lying on a plane, or on a line will produce **A** that is rank deficient. This will make **M** rank deficient as well and the algorithm won't produce the required rotation matrix. We solve this problem by introducing three virtual particles per rigid body. With them being basis vectors of the identity matrix in local coordinate space and basis vectors of the previous frame's rotation matrix in global coordinate space, using the eq. (9) yields:

$$\mathbf{A}'_n = c\mathbf{Q}_{(n-1)} \tag{18}$$

Where $\mathbf{A}'_n$ is the matrix that needs to be added to the summation given in eq. (9) and c is a constant that prevents the previous frame's rotation matrix $\mathbf{Q}'_{(n-1)}$ from influencing the current one more than required. For our tests, we found that the value of $c = 0.01$ fixes the dimension problem while making changes in the behaviour of other rigid bodies unnoticeable. Another solution could be to use oriented particles (see [CL18]).

It must be noted that the system presented so far is still susceptible to some other frequent problems. For example, adding unnecessary energy when resolving initial invalid states must be solved with stabilization. There is also an issue with tunneling, e.g. fluid particle getting stuck inside a grid-like rigid body particle structure. Both of those are solved in [MMCK14].

4. Joints implementation and types

The way we implement joints is by sharing one or more particles between two or more rigid bodies (like in table 1). As the system tries to keep the initial configurations of all the bodies, it will position the joint-particles in the mean of the their separated positions from all the rigid bodies they are a part of. This will in turn affect the position and rotation of the rigid bodies and the process will converge towards an acceptable configuration, should it exist. The selection and position of the shared particles, as well as their 'normal' neighbours, will determine how the joint behaves. This is, unlike traditional joints, the only way of specifying behaviour.

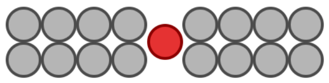

Figure 2: *Particles belonging to just one rigid body are colored gray, while the one belonging to two rigid bodies is colored red.*

The ball and socket joint has only one common particle between the rigid body pair which gives it three degrees of freedom (swinging in two directions and twisting). Swinging is limited by the contacts between the rigid body particles, while twisting has no limitations (figure 3).

The hinge joint implementation shown in figure 3 removes one swing direction, removes the twisting and limits the only remaining swing direction by adding two rigid body particles next to the joint ones. This can be used to model knees and elbows whereas a regular hinge would allow equal rotation on the both sides of the swing direction.

With four joint-particles, the stiff joint has no degrees of freedom. However, the inability of the system to converge fast enough results in an interesting behaviour where the object can slightly bend to it's sides but still preserve it's general shape. This is ideal for modeling spinal cord joints.

The universal joint removes the twist direction. Adding support for sub-dimensional rigid bodies from section 3.3 is important for this joint to work, since it contains such bodies. It consists of three rigid bodies connected by two hinges.

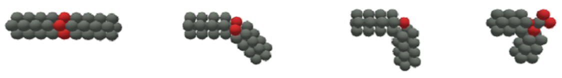
Figure 3: *The described joint implementations. From left to right: stiff joint, hinge joint, ball and socket joint and universal joint.*

5. Results and possible improvements

The skeletal actor from figure 1 is made up of joints presented so far. The system was compared with PhysX in Unreal Engine 4 (UE4) which gave results shown in table 2. Each object is a pair of two rigid bodies and a joint between them. All presented joints are equally used in all demos. As complexity was increased, both systems became more similar in frames per second (FPS). Without knowing the details about the inner workings of PhysX it is hard to make a fair comparison, but it gives an idea about the expected performance. Also, our method used the GPU considerably more than UE4 did. It would be interesting to see a similar implementation, but in a force-based system like the one from [Har07]. This would remove the need to perform expensive matrix calculations from section 3.3. Another improvement would be to exploit temporal coherence similar to [BYM05]. Storing particle data in a texture should also improve performance, because that way, two-dimensional cache of the modern GPU architectures would make cache misses less frequent.

6. Conclusions

The presented system enables great alleviation of 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 harder if not impossible to implement on the GPU. Making this a uniform particle simulation allows for interaction with other types of bodies (e.g. fluids) to emerge without any additional computation. The biggest utilization would be in real-time interactive applications. Entertainment software is usually built in such a way and could benefit from methods presented here. The measurements show that the presented system works well in comparison with today's commercial solutions such as Nvidia's PhysX.

Environment	Object count	FPS (GPU1)	FPS (GPU2)
UE4 (PhysX)	300	195	195
	600	95	95
	1200	39	41
Our method	300	273	233
	600	133	95
	1200	40	36

Table 2: *Measured performance of a simple demo scene where objects are being piled up. (GPU1: Quatro P6000, GPU1: GTX 1080, CPU: i7-6800k)*

References

[BYM05] BELL N., YU Y., MUCHA P. J.: Particle-based simulation of granular materials. In *Proceedings of the 2005 ACM SIGGRAPH/Eurographics symposium on Computer animation* (2005), ACM, pp. 77–86. 1, 4

[CL18] CHOI M. G., LEE J.: As-rigid-as-possible solid simulation with oriented particles. *Computers & Graphics 70* (2018), 1–7. 3

[Ebe10] EBERLY D. H.: *Game physics*. CRC Press, 2010. 1

[Eri04] ERICSON C.: *Real-time collision detection*. CRC Press, 2004. 1

[Gre08] GREEN S.: Cuda particles. *NVIDIA whitepaper 2*, 3.2 (2008), 1. 1

[Har05] HARRIS M. J.: Fast fluid dynamics simulation on the gpu. In *SIGGRAPH Courses* (2005), p. 220. 1

[Har07] HARADA T.: Real-time rigid body simulation on gpus. *GPU gems 3* (2007), 123–148. 1, 4

[HN16] HIGHAM N. J., NOFERINI V.: An algorithm to compute the polar decomposition of a 3× 3 matrix. *Numerical Algorithms 73*, 2 (2016), 349–369. 3

[MHHR07] MÜLLER M., HEIDELBERGER B., HENNIX M., RATCLIFF J.: Position based dynamics. *Journal of Visual Communication and Image Representation 18*, 2 (2007), 109–118. 1, 2

[MHTG05] MÜLLER M., HEIDELBERGER B., TESCHNER M., GROSS M.: Meshless deformations based on shape matching. In *ACM transactions on graphics (TOG)* (2005), vol. 24, ACM, pp. 471–478. 2

[Mil10] MILLINGTON I.: *Game physics engine development: how to build a robust commercial-grade physics engine for your game*. CRC Press, 2010. 1

[MM13] MACKLIN M., MÜLLER M.: Position based fluids. *ACM Transactions on Graphics (TOG) 32*, 4 (2013), 104. 1

[MMCK14] MACKLIN M., MÜLLER M., CHENTANEZ N., KIM T.-Y.: Unified particle physics for real-time applications. *ACM Trans. Graph. 33*, 4 (July 2014), 153:1–153:12. URL: `http://doi.acm.org/10.1145/2601097.2601152`, `doi:10.1145/2601097.2601152`. 1, 2, 3

[OLG*07] OWENS J. D., LUEBKE D., GOVINDARAJU N., HARRIS M., KRÜGER J., LEFOHN A. E., PURCELL T. J.: A survey of general-purpose computation on graphics hardware. In *Computer graphics forum* (2007), vol. 26, Wiley Online Library, pp. 80–113. 1

[THM*03] TESCHNER M., HEIDELBERGER B., MÜLLER M., POMERANTES D., GROSS M. H.: Optimized spatial hashing for collision detection of deformable objects. In *Vmv* (2003), vol. 3, pp. 47–54. 1