Real Time Rigid Body Dynamics in 3D[*]

†

Sayantan Datta
McGill University
sayantan.datta@mail.mcgill.ca

ABSTRACT

In this project we investigate Jacobi and Gasuss-Siedel for solving Rigid Body LCP in real time and compare their performance, stability, and penetration errors. We attempt to eliminate major performance bottlenecks and also investigate the feasibility of using GPUs for doing physics computation.

KEYWORDS

Rigid Bodies, Contacts, GPU

ACM Reference format:
Sayantan Datta. 2017. Real Time Rigid Body Dynamics in 3D. In *Proceedings of ACM Conference, Washington, DC, USA, July 2017 (Conference'17),* 4 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Rigid Body Dynamics is useful for simulating the behavior of rigid bodies and their interactions. Rigid Body collisions are constrained optimization problem and can be formualted as Linear Complementarity Problem(LCP).

2 RELATED WORK

Velocity Based Shock Propagation[3] provides the mathematical foundation for modeling contact dynamics and derives an LCP formulation, solved using Projected Gauss Siedel. *Mass Splitting for Jitter-Free Parallel Rigid Body Simulation[5]*, is a good reference for understanding various parallelization and stabilization techniques for LCP solvers.

3 TECHNICAL DETAILS

3.1 Software

We use *Ogre3D v1.9* as our real time render engine using OpenGL 4.5 backbone, *Bullet 3.2* as the collision detection library, *OpenCL 1.2* as GPU programming API and C++ to glue the pieces together. We also make use of C++11 threading and STL libraries. We make use of the starter code form *Ogre3d* and added various components on top of it. Ogre3D has built-in screen capture tool with various output file formats including *Jpeg*. The application setup uses *Ubuntu 14.04.03, AMD Fglrx Driver v15.302, AMD APP SDK 2.9* for OpenCL, *GNU GCC 4.8* and *Eclipse C++ IDE*

[*]Produces the permission block, and copyright information
[†]The full version of the author's guide is available as `acmart.pdf` document

3.2 Hardware

We use *Intel Core i5 2500@3.3Ghz* CPU and *AMD 7970* GPU.

3.3 Application Structure

We divide the application into three main threads - rendering, physics, and screen capture. The screen capture is modeled as a producer consumer problem where the rendering thread produces the images and puts them to a queue while the screen capture thread pops the queue and saves the images to disk.

Rendering and physics threads run out-of-sync; meaning there is no bound on the speed with which rendering and physics are processed. Additional complexity is introduced due to the fact that all Ogre3D function calls must be made from the thread that initialized Ogre system. There are two critical sections where the rendering and physics threads interacts. The physics thread continuously updates location and orientation of rigid bodies. When the rendering thread needs the location and orientation, it may have to wait in case the same variables are being updated by physics thread. So the code altering and reading the location and orientation variables are kept inside critical section. Second, when new bodies are added, they must be registered with the rendering and physics systems. We pause the physics thread when new objects are being added by the rendering thread and resume physics once new bodies are registered.

For Rigid Body Mechanics we refer to *Pixar Course Notes[6]*. We detect collisions using Bullet library, which provides the contact point, contact normal and penetration distance. A maximum of four contact points per contact manifold, sorted according to their distance from other contact points in the manifold[4] is provided by the Bullet library. While we do not use penetration distance for anything other than error reporting, it is worthwhile to note that it can be used for constraint stabilization. We form the Contact Jacobian [3] for each contact without assembling the actual matrices. From this point onward we diverge into two paths - Gauss Siedel on CPU and Jacobi on GPU(OpenCL).

3.4 Notations

For brevity, we have not included explanation for all mathematical symbols. The symbols are consistent with *Velocity-based Shock Propagation for Multibody Dynamics Animation[3]* and any new symbols and notations are explained in the text.

3.5 Gauss Siedel

Gauss Siedel iterations are implemented as follows:

$$T = M^{-1}J^T \quad (1)$$

For each iteration k,

$$\lambda_i^{k+1} = \lambda_i^k - b_i' - J'_{row_i}\Delta V^k \quad (2)$$

$$\Delta\lambda_i = \lambda_i^{k+1} - \lambda_i^k \quad (3)$$

$$\Delta V^{k+1} = \Delta V^k + T_{col_i}\Delta\lambda_i \quad (4)$$

3.6 Jacobi

There are two approaches for Jacobi solve:

```
Data: b', J', T, $\Delta V = 0, \Delta\lambda = 0$, n = #Contacts
Compute in parallel over i, $i \in 1...n$
for each iteration, k do
    $\lambda_i^{k+1} = \lambda_i^k - b_i' - J'_{row_i}\Delta V^k$
    $\Delta\lambda_i = \lambda_i^{k+1} - \lambda_i^k$
    $deltaV_i = T_{col_i}\Delta\lambda_i$
    Barrier synchronize
    Do Atomic Add on variable $\Delta V$
    $\Delta V^{k+1} = \Delta V^k + deltaV_i$
end
```

Algorithm 1: Jacobi- Approach 1

```
Data: b', J, M, T, $\Delta V = 0, \lambda = 0$, n = #Contacts
Compute in parallel over i, $i \in 1...n$
/* Compute $A = JM^{-1}J^T$ */
for $j \in 0...n$ do
    $A_{ij} = J_{row_i}M^{-1}_{sub_{ij}}J^T_{col_j}$
end
Barrier synchronize
for each iteration, k do
    $a_i = 0$
    /* Compute $A_{row_i}\lambda^k$ */
    for $j \in 0...n$ do
        $a_i = a_i + A_{ij}\lambda_j^k$
    end
    $\lambda_i^{k+1} = \lambda_i^k - b_i' - a_i$
    Barrier synchronize
end
$deltaV_i = T_{col_i}\lambda_i$
Do Atomic Add on variable $\Delta V$
$\Delta V = \Delta V + deltaV_i$
```

Algorithm 2: Jacobi- Approach 2

Approach 1 requires lesser computations per thread but require atomic operations inside for-loop. Atomic operations are slower since writes to same memory location is serialized. ΔV is a vector of size O(*#Bodies*). So, number of writes serialized per thread is of order O($\frac{\#Contacts}{\#Bodies}$), which is practically O(constant).

Approach 2 requires memory lookup of order O(*#Contacts*) per thread. Optimized memory lookups are much less costly than atomic operations but compared to Approach 1, the order is higher. So, Approach 2 does not scale well with increasing number of contacts.

We have tested both approaches and approach 1 is much faster.

3.7 Stabilizing Jacobi

It is known that Jacobi iterations do not converge under certain condition[5]. We test two approaches for constraint stabilization.

In our first approach, we divide the mass(linear and angular) of each body with number of contacts[1] affecting the body.

In our second approach, we try to estimate spectral radii ρ of matrix $A = JM^{-1}J^T$ and multiply D_{ii}^{-1} with $\frac{2}{p}$ [5] such that

$$D^{-1}_{ii_{eff}} \leq \frac{2}{p}D^{-1}_{ii} \quad (5)$$

Computing spectral radii,

$$||A||_2 = \sqrt{\rho(A^TA)} \quad (6)$$

Since $A = JM^{-1}J^T$ is symmetric, equation 6 reduces to

$$||A||_2 = \rho(A) \quad (7)$$

Finding 2-norm is again an optimization problem, however we can find the upper bound for 2-norm as follows:

$$||A||_2 \leq \sqrt{n}||A||_1, \;\; n = \#contacts \quad (8)$$

1-norm of a matrix is the maximum of absolute column sums. Since, A is symmetric it is equivalent to finding maximum of absolute row sums.

Our first approach is computationally cheaper, converge faster and reasonably stable. We could simulate a random pile of 450 regular objects. However a mixture of regular and irregular shape objects is sometime unstable beyond a few hundred, in which case we further reduce D_{ii}. Second approach is stable for any object shape and count but is slow to converge and does not scale well with increasing number of contacts. Computing the 1-norm has a time complexity of order O(n^2).

3.8 Simplifying Frictional Constraint

In this section we describe a process for simplifying frictional constraint using only one tangential direction. Since one tangent does not span the entire tangent plane, there is a direction orthogonal to the only tangent which experience no component of friction force. So, if we carefully push the object in a direction orthogonal to the only tangent, the object will move forever as there is no opposing force in that direction. To address this issue, we randomize the direction of the tangent at each time step, which over several time steps have an effect similar to having two tangents. This is because, there is no unique direction which experience zero force over several time steps. With this simplification, we reduce a third of computation and memory required for solving LCP.

Component of friction force along the tangent is $Fcos(\theta)$ where θ is a random variable between 0 and 2π. Magnitude of effective force of friction over several time steps is the expectation of $F|cos(\theta)|$. Evaluating the expectation, we get $\frac{2F}{\pi}$. We multiply the coefficient of friction μ with $\frac{\pi}{2}$ so that the expectation is F. The calculation however assumes infinite number of time steps, which is not the case. Practically, objects moves along a zig-zag path before coming to rest. A good random number generator and smaller time step can minimize artifacts to visual convergence.

3.9 Testing and Results

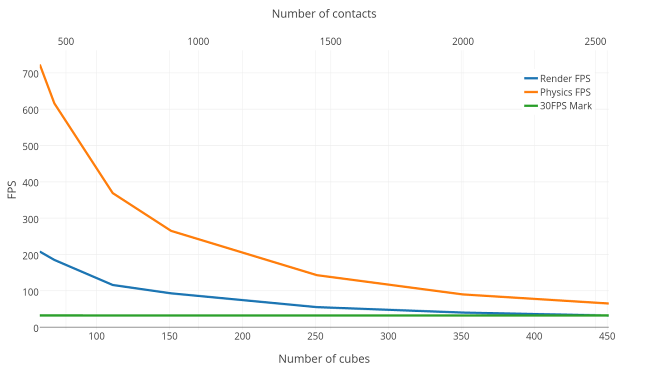
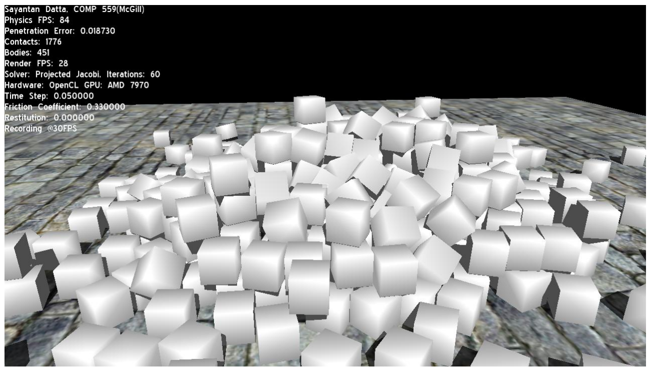


Figure 1: PGS with 5 iterations

From Figure 1, it is evident that simulating hundreds of object is more graphics bound than physics bound. Since all of the GPU performance is utilized for rendering alone, the blue plot represents the maximal rendering performance. For measuring physics performance we add objects to into the pile until Render FPS reaches 30 FPS mark and we record the Physics FPS with this setting.



```
Sayantan Datta, COMP 559(McGill)
Physics FPS: 84
Penetration Error: 0.018730
Contacts: 1776
Bodies: 451
Render FPS: 28
Solver: Projected Jacobi, Iterations: 60
Hardware: OpenCL GPU: AMD 7970
Time Step: 0.050000
Friction Coefficient: 0.330000
Restitution: 0.000000
Recording @30FPS
```

Figure 2: Pile of cube objects

We measure inter penetration error by dropping objects into a random pile till the 30FPS mark is reached. We then allow the objects to settle down for 5 minutes and record the average inter penetration. Average inter penetration is computed as total inter penetration among all objects divided by number of contacts.

#Iterations	Contacts/sec	Inter-penetration
5	165,751	1.23*
10	128,276	0.45*
20	84,864	0.023

Table 1: Projected Gauss Siedel

#Iterations	Contacts/sec	Inter-penetration
60	157,557	0.022*
120	135,214	0.017*

Table 2: Projected Jacobi

**Increasing inter-pentration error with time.*

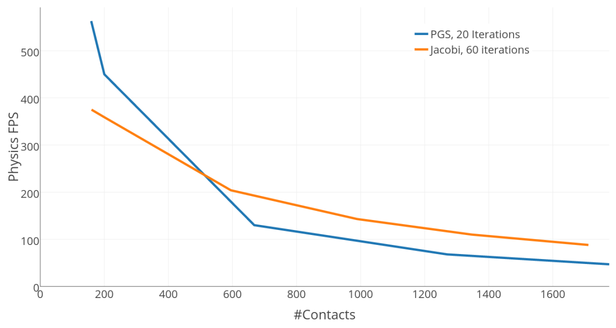


Figure 3: Performance in FPS

4 CONCLUSION

There is no clear winner between Jacobi and Gauss-Siedel. PGS wins hands down when inter-penetration error and stability are important while Jacobi shines when scalability is a concern. There is a trade off between stability and inter-penetration error for Jacobi. While it is possible stabilize Jacobi by multiplying D_{ii} with a fraction(≤ 1) but too small value of the fraction results in visible inter-penetrations. A typical use case for Jacobi could be in fast paced video games simulating thousands of object where inter-penetration is not a concern. In some cases, like simulating few hundreds of objects, do not have enough parallelism to warrant a GPU implementation but is expensive enough to overwhelm a single core CPU. In this regard, Jacobi on multi-core CPU makes more sense. PGS should be used when there is only a handful of objects or when absolute visual convergence is required.

5 DISCUSSION

The most challenging part of the project is optimizing and testing various kernels for doing Jacobi iterations. It is also difficult to separate stability issues due to mathematical reasons from coding bugs.

After running both physics and rendering on GPU, we were still not able to saturate GPU utilization. Graphics FPS is bound by the number of draw calls which limits the number of objects we can display on screen. It is more efficient to render few objects with large polygon count than to render large number of objects with small polygon count, the latter being increasingly CPU bound. Limit on number of objects limits the number of contacts which limits GPU utilization due to physics. Second, we are doing some physics computation on sequentially on CPU, like building the contact constraints and updating object properties. This lowers GPU utilization because GPU is not being fed with enough data to keep it busy, a phenomenon commonly known as pipeline bubble. Solution to the first problem is to use new APIs like Vulkan or DirectX12, which reduces CPU overhead and allows multiple threads to issue draw calls. Second, problem can be solved by moving as much physics processing as possible to GPU.

ACKNOWLEDGMENTS

The author would like to thank *Streamcomputing.eu* for sharing the OpenCL 1.2 floating point atomic-add code. The author would also like to thank *Stackoverflow*, *Ogre3d forum[2]*, *Bullet Physics Wiki* for sharing valuable informations and tutorials.

REFERENCES

[1] Robert Bridson, Ronald Fedkiw, and John Anderson. 2002. Robust Treatment of Collisions, Contact and Friction for Cloth Animation. *ACM Trans. Graph.* 21, 3 (July 2002), 594–603. https://doi.org/10.1145/566654.566623

[2] Duke and others. 2005–. Basic Tutorial 1, Your First Scene. (2005–). http://www.ogre3d.org/tikiwiki/tiki-index.php?page=Basic+Tutorial+1 [Online; accessed <today>].

[3] Kenny Erleben. 2007. Velocity-based Shock Propagation for Multibody Dynamics Animation. *ACM Trans. Graph.* 26, 2, Article 12 (June 2007). https://doi.org/10.1145/1243980.1243986

[4] Bullet Physics. 2005–. Bullet Persistent Manifold. (2005–). http://bulletphysics.org/Bullet/BulletFull/classbtPersistentManifold.html [Online; accessed <today>].

[5] Richard Tonge, Feodor Benevolenski, and Andrey Voroshilov. 2012. Mass Splitting for Jitter-free Parallel Rigid Body Simulation. *ACM Trans. Graph.* 31, 4, Article 105 (July 2012), 8 pages. https://doi.org/10.1145/2185520.2185601

[6] Andrew Witkin and David Baraff. 2001. Pixar Physically Based Modeling Course Notes. (2001). https://www.pixar.com/assets/pbm2001/index.html [Online; accessed <today>].