3D Rigid Body Simulation with Contacts using the LCP Formulation

Vincent Petrella

Abstract

We present our implementation of a Rigid Body simulation implemented using 3D meshes. The simulation takes into account any types of forces, including gravity and resolve collisions in a graceful manner using the Linear Complementary Problem formulation of contact forces.

1 Introduction

The original goal of this project, in content, was to implement efficient 3D rigid bodies collisions with an LCP contact solver to culminate with the ability to fracture 3D objects in real-time. However, due to unforeseen time constraints and severely blocking issues in the implementation of earlier parts, this ambitious proposal could not be carried to completion. We were able to successfully implement collision detection, Physically based Rigid Body motion and contact resolution using the Linear Complementary Problem (LCP) formulation described in [Erleben 2007]. In this paper, I will aim at describing the approach we took to implement some of the concepts we saw in class as well as describe the shortcomings of some of our attempts.

2 Related Work

There are many commercial and open-source Rigid Body simulation engines. Some of them approach the problem in different fashions: Engines like Bullet, ODE and Havok appear to each use some of the same tools in different flavours with different optimizations. the LCP formulations such as the one we implement is indeed popular among them [Bul 2006]. Collision response algorithms mostly come in three flavours: Projection methods, Penalty based, and Impulse based methods. The first of these -the projection- will, after detecting a collision, control objects positions directly so as to satisfy non-penetration properties. The second, the penalty method, will apply forces to rigid bodies in an effort to keep them separate. Some video games and earlier engines used spring like forces to keep object separated from their contact points. However, such techniques suffer from stiffness issues that arise in the integration scheme, and have to be fine tuned to provide good realism. Finally, Impulse based methods work by applying impulses (changes in velocity) on objects to cancel the relative velocity of two colliding bodies in the direction of their collision. Typical and simple methods will treat each collision separately as they occur, applying the required impulses on each body. However simple these methods are, they suffer from poor realism and stability issues while simulating resting objects (such as piles of boxes), as they do not consider the entire system before applying said impulses. For this project, we implemented the LCP formulation, which we will describe further down.

3 Implementation

The implementation was comprised of three major components: The integration scheme for the equations of motions of rigid bodies, detecting collisions between bodies, and finally solving the LCP with a projected Gauss-Seidel method.

3.1 Rigid Body Motion

The "starter code" used in the project allowed us to load an display objects from .obj files, but there was no proper rigid body motion implemented. We implemented a RigidBodySystem class, holding information about RigidBody objects, and integrating their position using a Forward Euler integration scheme. Rigid body accumulate forces between each integration step, which are then integrated into the linear and angular velocity. Linear forces can be applied on a body at any position and the proper linear acceleration and torque are computed. Only simple objects were used in the demonstration videos, so as to constraint the use to simple (diagonal) moments of Inertia tensors. The state of a Rigid Body position is held in a 4×4 matrix (in homogeneous coordinates) made of a 3×1 position vector, and a 3×3 rotation matrix. The resulting transform matrix is then used in the rendering pipeline for object rasterization. Consequently, we simplify the code by considering the center of mass to be the $\vec{0}$ in model coordinates. In order to help this approximation, we ensure that 3D data is normalized upon loading. We used the exponential map formulation with Rodrigues Formula to compute the ΔR rotation from state t to $t + 1$. After multiplying the rotation by ΔR, we normalize each column of the new rotation matrix to avoid scale drift along each axes. While seemingly straightforward and easy to implement, this integration step needs to work flawlessly and as expected if we are to debug subsequent implementations. We encountered some issues, for example: a negative time-stepping bug was left unnoticed due to double integration. Furthermore, when something did not move in the expected direction, we at first simply changed the quantities sign. These kinds of "fixes" complicated debugging down the line and proper investigation about why the system did not react as expected would have been more appropriate.

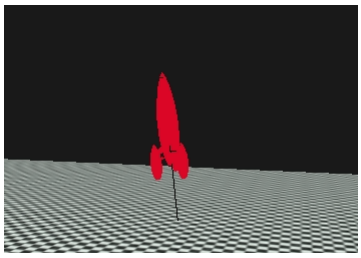

Figure 1: *A rocket is propelled through the air by applying a constant thrust on the bottom.*

3.2 Collision Detection

Before tackling the issue of collision response, we need to detect when two bodies collide. As the main application of our method is real-time rigid body simulation, the main object representation is indeed the polygonal mesh. Standard techniques notably used in video games simplify the detection task by using simpler colliding primitives enclosing the mesh. Then very fast detection techniques can be applied to find collision points, such as sphere-sphere. We elected to do mesh-mesh collision detection in order to easily test collisions of different types of objects. Collision types can be reduced to two well supported kinds: Vertex-Face and Edge-Edge. To facilitate and accelerate collision detection, we looked for libraries implementing the OBB acceleration data-structure (Ori-

ented Bounding Boxes). OBB's facilitate pruning triangle-triangle tests and is one of the many bounding volume hierarchy structure available. It is both efficient and relatively easy to implement. After struggling to integrate "big" implementations such as Z-Collide and OpCode, we turned to a simpler library, ColDet [col 2013], with readable code and straightforward integration. However, multiple modifications had to be made in order to retrieve all contact points per mesh-mesh queries, as well as the appropriate contact normals for the collision cases. The contact normal was approximated to be the triangle's normal for vertex-triangle test and the direction of the cross-product of the two edges for edge-edge cases. This acceleration, however proved insufficient when dealing with hundreds of bodies in a scene. To help increase the interactivity of the simulation, we introduced a pre-check during broad-phase collision detection. We test for Bounding Sphere (computed from mesh data) intersection for each pairs of body to avoid entering the expensive narrow-phase setup for each body. This help significantly increase performance on large systems.

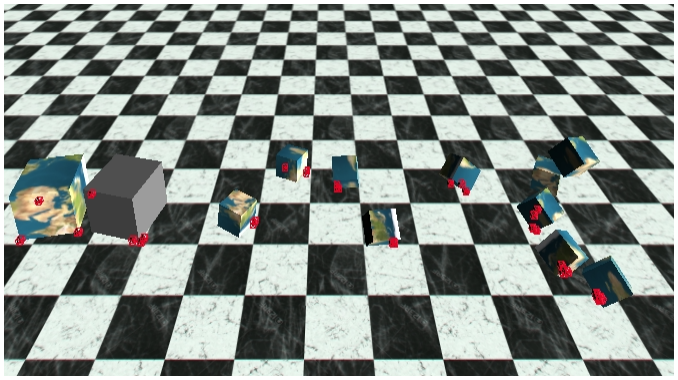

Figure 2: *Cubes falling down due to gravity collide with one another in a realistic fashion*

3.3 Linear Complementary Problem solver

Once we obtain the contact normal and position for each collision, we construct the contact Jacobian as described in [Erleben 2007]. The iterative LCP solver requires knowledge on the upcoming velocity of each object. We therefore precompute acceleration and velocity update before resolving contacts. Using a sparse representation of the matrix operations required to solve for each Lagrange Multiplier λ_i in the Gauss-Seidel LCP iterative algorithm, we obtain in linear-time the appropriate correction velocity (impulse) to apply on our objects. However, this ended up much more difficult than expected. Even-though we had already implemented the algorithm in 2D, the intricacies of the method, along with bad programming practices made the task of debugging the behaviour of the collision response very time consuming. Among others, notable mistakes were bad indexing in the T matrix (used in the ΔV update to push objects in the allowed directions). Another bug lied in the way linear and angular velocities are stored in our Rigid Body object. Since we are considering Rigid Bodies at their centers of mass, we were keeping the linear velocity in world coordinate while the angular velocity, to evaluate the exponential map, was expressed in the body coordinate system. However, the Jacobian formulation indeed specifically writes the angular velocity constraints in World coordinate. In the end, a bulk of this implementation was in programming all the tools required to debug, visualize and explore the behavior of the system.

Figure 3: *Friction allows this red cube to remain in steady contact while tilted at an angle. A slightly more tilted angle forces the cube to tumble down the slope to satisfy non penetration constraints.*

Table 1: *Average execution times for blocks resting on floor*

# of Bodies	Col. Detection (ms)	Col. Processing (ms)
5	0.6	1.5
10	0.9	1.8
100	20	60
200	44	140

4 Results

Qualitative Results can be observed from figure 1 to 4, as well as in the provided video. We can see that for non stacking objects, performance is relatively good, with some spikes in frame-rate here and there (due to convergence issues). As expected objects react accordingly to contact. Bounciness and friction handle well, as demonstrated in the video (and figure 3.). In terms of performance, we indeed, have much lower collision detection performance for high-poly models. We however decided to use solely simple models, which limits the impact of this phenomenon. While multiple objects at rest on still surfaces with occasional collisions handle very well, systems with stacking contacts converge fairly slowly. However, the most prominent issue is that the solver frequently fails to converge (even after 500 iterations). There may be some bugs remaining in the contact Jacobian setup, in which some collisions properties (like the normal direction) are inconsistent with the system, preventing convergence. We were unable to stack more than 25 cubes without the tower crumbling. Multiple towers (such as figure 4.) handle fairly well due to the error not propagating as much as in single towers. Further results for collision detection time and average collision processing time (LCP solver) can be found on table 1. The average number of iterations for stacks of cubes can be found in table 2.

5 Future Work and Conclusion

Many stages of this process act as bottleneck for efficient computations, it should be further optimized to enable real-time applications. Firstly, parts of this pipeline can run in parallel to try and leverage modern multi-threaded systems. In an efficient parallel programming setup, acceleration and velocity update, as well as Body-Body collision detection tasks can be bundled in mutually exclusive subsets and evaluated in parallel independently with no risk to data consistency. The LCP solver, however, cannot be easily parallelized due to the sequential nature of each iteration. Indeed, for each iteration, the latest correction velocity (ΔV in [Erleben 2007], section 3) computed in the first $(i-1)$ contacts is required to compute λ_i and guarantee the Gauss-Seidel convergence. Due to this constraint, a safe and efficient parallelization of this algorithm would be challenging. However, it should be noted that only

Table 2: *Average number of iterations for stacked cubes*

# of Bodies	Avg. # of Iterations
5	1.65
10	36
25	47

objects in serially incident contacts need to be solved together. Indeed, computing the new λs for a stack of cubes requires no information about another stack, as long as the stacks do not collide with one another. Therefore, solving a simple graph search problem (where objects are connected if they collide) to find islands of contact stacks would make a bundling such as described above possible. In such cases, multiple instances of LCP could be solved in parallel, running on each of these islands. Note that contacts with pinned bodies should not appear in the graph as the λs computed for a pinned body are discarded, and thus stacks of objects resting on the same immovable object are not considered connected.

There are others and perhaps more crucial "obvious" optimizations to apply to the method such as contact pruning: Collision detection routines may report many different, potentially redundant collisions. Extremely close collisions can be processed as one, while multiple co-planar collisions involving two same bodies may be replaced with a single one, by interpolating their positions (a resting cube will have four contact points, but only one in the barycenter is required).

Another essential optimization would be to not discard information about the system once solved. Indeed, many contact from frame to frame will remain the same due to space-time consistency in most application. "Warm start" techniques can be usedwith the Gauss-Seidel algorithm, where the results of a previous solution to a similar contact can be applied on start-up, to help start faster to the solution point and accelerate convergence.

In conclusion, we see that the LCP formulation for contact resolution is a viable option. Resting contacts behave very well with no "jiggling" issue and stacking does exhibits a good behaviour up to around 25 boxes stacked. Along with the optimization described above, the LCP formulation for contact resolution should be well suited in real-time environments. Another advantage of this method, is that several other types of constraints can be implemented with different types of Jacobian matrix entries, allowing the method to be versatile once properly optimized.

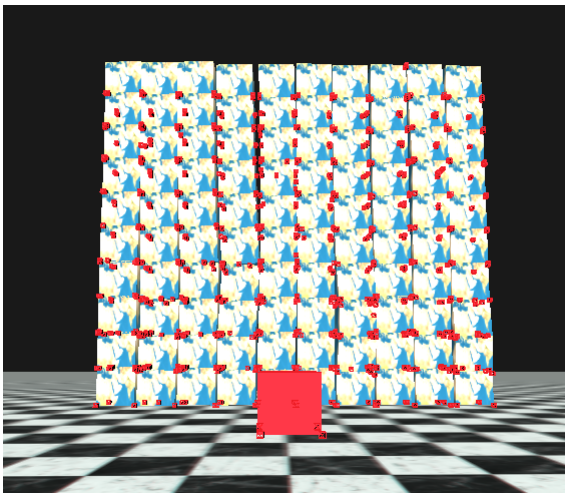

Figure 4: *A wall of blocks falls on the floor and remains at rest. Performance drops significantly.*

6 Code

You can find the Github repository where all revisions of the code are archived at [git 2013].

References

2006. Which lcp's solver have been used by popular physics engine?

2013. Coldet 3d, collision detection library. sourceforge.net.

ERLEBEN, K. 2007. Velocity-based shock propagation for multibody dynamics animation. *ACM Trans. Graph. 26*, 2 (June).

2013. Code of the program used to implement the rigid body system.