Simulating Rube Goldberg Machines Using Rigid Body Collisions

Aaron Hill Austen Ross

1 Motivation

There are many impressive video demonstrations of domino chains or Rube Goldberg-esque machines in computer graphics. They capture the potential of rigid body simulation by showing the actions of multiple simple objects chaining together to produce large or entertaining effects. For example, the Fiat Lux video that was shown in lecture used a chain of dominoes falling over to enhance the already impressive ray tracing effects on display. Our goal was to use rigid body collision detection to create our own version of a computer-modeled Rube Goldberg machine that could produce similarly awe-inspiring results.

2 Related Work

Much of our implementation was informed by the works of David Baraff, who has done a lot of work in the field of rigid body simulations (and co-developed the engine used in the Fiat Lux video). Baraff laid out the foundation for both the object representation of rigid bodies as well as how to simulate contacts between them [1].

Others have made significant contributions in this area as well; Gottschalk et al. described a data structure which allows for significantly faster interference detection in many scenarios [4]. Kaufman et al. outlined a method of calculating friction between bodies in an approach that can handle over one thousand nonconvex rigid bodies simultaneously [5].

3 Implementation

3.1 Representation of Rigid Bodies

Using the rigid body structure described in [1] as a guide, we represent each object (referred to in our code as a *Body*) as a collection of point masses (vertices) that share “state variables”. Each body stores a vector of vertices, allowing it to render itself to the screen in its current position. Additionally, each body stores four “state variables”: position (usually the position of the center of mass), orientation, and both linear and angular momentum.

At each time step, the net force and torque being currently applied to the body produce the rate of change of each state variable. From these state variables, the “auxiliary variables,” the linear and angular velocity as well as the inertia tensor, are

calculated at each time step using equations from physics described in [1].

There are also a number of constant variables associated with each rigid body object. The object's mass and coefficient of friction (currently unused) can be specified at object creation. There's also a constant component of the body's inertia tensor that is calculated at object creation, called I_body.

I_body is determined by the average distance of a point mass from the center of mass of the body. In our current implementation, we inaccurately use only the vertices to represent all point masses, as an approximation; while potentially not an issue for shapes like cubes or spheres, this would certainly cause issues for the vast majority of possible meshes. In the future, I_body should be properly calculated by sampling all of the point masses of the body. Additionally, if one wanted to model a body with non-uniform density, the calculation would need to be modified even further.

Bodies store their orientation in the form of a quaternion; a pair consisting of a real value and a three-vector describing an imaginary axis. The quaternion class was given overloaded operators to support addition and multiplication by other quaternions, 3-vectors, and scalars, as well as addition with other quaternions. This quaternion can also be converted to a three-by-three orientation matrix to be used in calculating the angular velocity of the body at each timestep.

Compared to a three-by-three matrix, quaternions need to store less redundant data, which makes them both more storage efficient and less prone to numerical drift than matrices. While interpreting the meaning behind the values of a quaternion is somewhat unintuitive, it can always be converted back into a standard orientation matrix.

Since motion is applied equally to all point masses on a rigid body, most information was stored only by the body - the individual vertices do not need to store their momentum or orientation, nor do they need to know about the forces acting on them. However, each vertex does store its position in "body space" - its position in relation to the body's center of mass. Since we are only performing translations and rotations in our simulations, each vertex's body space position should remain constant, and only needs to be computed at object creation.

Using the body's linear and angular velocities, as well as a vertex's current position relative to the body's center of mass, we are able to

determine the rate of change of that vertex's position in world space at each time step.

3.2 Handling Physics

We modeled the physics of our system using Newton's laws of motion. At each step of our simulation, we computed the forces on each object (and therefore the acceleration via **F = ma**), applied these forces as an update to our velocity, and applied the velocity as an update to our position.

We used the simplest possible numerical solution to this system of ordinary differential equations - Euler's method. This had the advantage of integrating nicely with the rest of our simulation - since we only needed to know the values of our derivatives (acceleration and velocity) at the current time step to compute the update for the next time step, there was no need to store additional information with each body.

However, Euler's method is only a first-order method - in a more complicated simulation, this lack of accuracy might become apparent. One area for enhancement is to apply a more accurate, higher-order ODE solver, such as Runge-Kutta. However, we do not have an explicit function for the values of the derivatives (e.g. acceleration) in our system - we must compute them at each time step based on the current positions of bodies within our simulation.

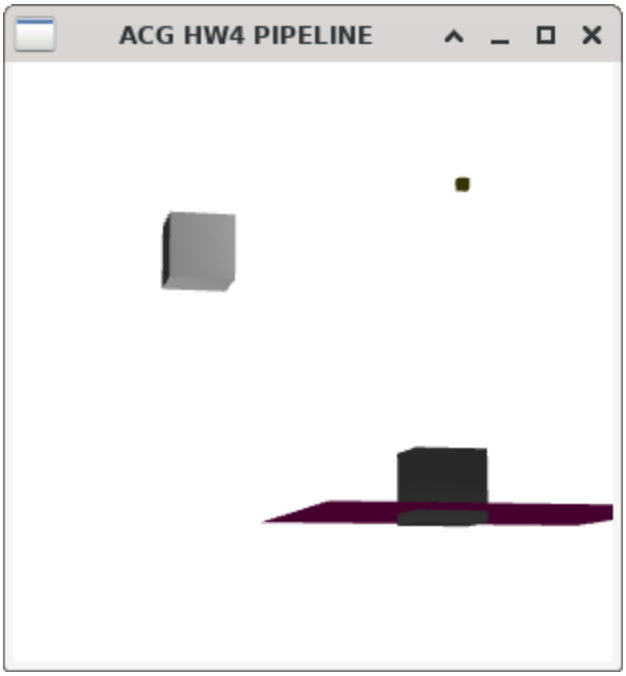
Since higher-order methods like Runge-Kutta require evaluating derivatives at multiple values of the independent variable to compute a single time step, implementing this in our simulation would require some moderately significant refactorings to our code. Specifically, we will need to extend our program with the ability to 'speculatively' determine the forces at a future point in time, and then 'rewind' back to a previous point in time. For simple scenes involving only two or three objects, the extra precision might not be noticeable. However, it could reduce the need for non-physical tricks such as velocity dampening in situations where the simulation might not otherwise converge.

3.3 Collision Detection and Response

Our implementation of collision handling was based heavily off David Baraff's work in [1]. Specifically, we implemented the 'contact collisions' described in the referenced paper. Contact collisions occur when two objects are observed to interpenetrate at a particular time step. To handle contact collisions, the simulation must 'rewind' time to the point where the objects first collided, and calculate the necessary forces to apply to prevent interpenetration.

One significant limitation of our current approach is that all models must have their vertices specified by hand in a single input file. We implemented some basic functionality to duplicate a model with a translation applied - however, this approach will not scale to a large number of finely-positioned objects. To make it easier to design complex scenes involving many objects, it would be useful to be able to position the objects within the simulation, instead of needing to edit individual vertex positions manually.

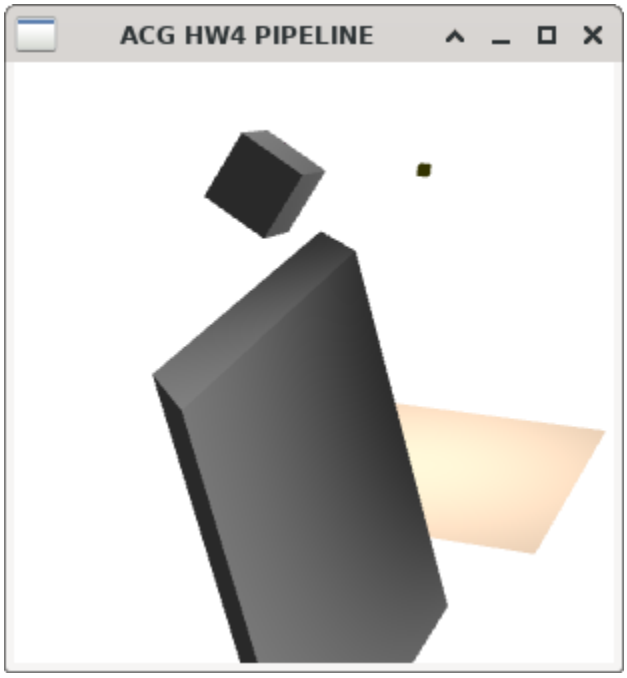
While working on the project, we initially had difficulty with the correct computation of contact forces between objects and the ground plane. At the start, an object passing through the ground plane caused a force to be applied in the opposite direction, which was intended to bring the velocity to zero instantaneously. However, this did not result in the desired effect - instead, the applied force resulted in an overcorrection to the velocity. As a result, the cube appeared to oscillate in and out of the ground plane:



Since the oscillating effect occurs over several frames, it cannot easily be demonstrated via a still image. However, the above screenshot gives an idea of the nature of the problem - the grey cube has penetrated through the ground plane to the other side. When contact forces are being simulated correctly - this should never happen - the interpenetration should be detected and resolved by the simulation through a contact force, with the result that every frame rendered to the screen shows objects in physically correct positions (i.e. no interpenetration).

Another challenge came during the implementation of rotation. Our rotation implementation caused the cube to grow in size as it rotated, instead of keeping the vertices at a fixed distance relative to each other. Unfortunately, we were not able to solve this problem - this is certainly

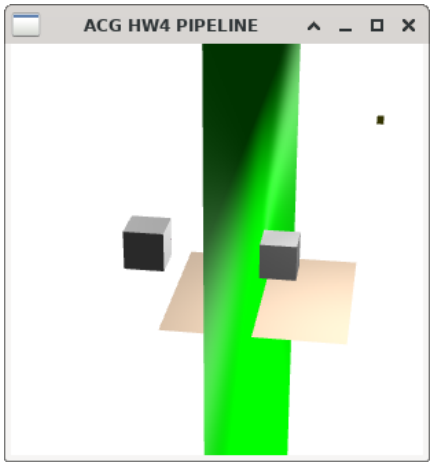
an area for additional improvement. This problem may be the result of using Euler's method rather than a more precise method like Runge-Kutta when updating vertex positions at each time step.



In the above image, one of the cubes has become significantly larger than the other (and has stretched and sheared along one axis) after rotating for several seconds. The smaller cube has not rotated much, as and a result is not yet experiencing the same enlargement and shearing. Given enough time, the smaller cube will also display the same effects as the larger cube.

4 Results

One early result was the rendering of the 'separating plane' between two objects:



In this image, the grey cubes are both 'bodies' in our simulation. The green plane is the 'separating plane' between them. The separating plane is not unique - there are an infinite number of planes that can be constructed between the two cubes. However, since our simulation is limited to convex polyhedra, the separating plane must be parallel to one of the faces of one of the two bodies. We use this property to construct a separating plane by simply testing all faces on each cube. The final separating plane that we compute is offset slightly by a small distance *epsilon* from the chosen face - this allows us to render the plane without Z-fighting.

5 Conclusions

One area that we would like to improve on is the types of allowed interactions between objects. Currently, each object is modeled as a rigid body, with a single acceleration vector for the entire object. As a result, interactions between objects are

limited to instantaneous collisions and resting contacts. It is not possible for objects to be 'coupled' or 'linked' to each other in a way that our simulation can understand. Allowing a persistent connection between objects would allow us to simulate more complex scenes, such as a lever attached to a block.

Another area of improvement is improving the ease of debugging our program. Debugging issues with the simulation, such as incorrect velocities or rotations, currently requires logging information about a body at each step. If the problem only being visible through many time steps of the simulation, this will produce a large amount of output for the developer to sort through. Adding in some debugging visualizations could make these kinds of issues much easier to diagnose and solve. For example, rendering the force and acceleration vectors as arrows on each object could help make it obvious to developers when a problem is occuring, when it might not be clear simply from looking at a table of acceleration and velocity values.

A final area for extension is implementing additional types of collision handling. In addition to 'colliding contacts', the paper [1] describes 'resting contacts' between two objects. These collisions allow two objects to remain stationary while in contact with each other, while still being free to separate under the influence of other forces. However, implementing this type of collision is significantly more complicated - using the approach described in [1], a quadratic programming solver is required.

Aaron worked on the implementation of the separating plane calculation and rendering, extensions to the object file format used to render collisions, and the initial implementation of collisions with the ground plane. Austen worked on the initial framework setup, implementation of the physics simulation (mass, forces, and quaternion-based rotation), as well as improvements to ground-plane and object collision handling.

6 References

[1] Baraff D. An Introduction to Physically Based Modeling. In *SIGGRAPH 1997 Course Notes*, 1997.

[2] Baraff D. Analytical methods for dynamic simulation of non-penetrating rigid bodies. In *Computer Graphics (Proc. SIGGRAPH)*, volume 23, pages 223–232. ACM, July 1989.

[3] Baraff D. Fast contact force computation for nonpenetrating rigid bodies. In *Computer Graphics (Proc. SIGGRAPH)*, 28:23–34, 1994.

[4] Gottschalk S., Lin M. C., Manocha D. OBBTree: A Hierarchical Structure for Rapid Interference Detection. In *Computer Graphics (Proc. SIGGRAPH),* pages 171-180, August 1996.

[5] Kaufman D. M., Edmunds T., Pai D. K. Fast Frictional Dynamics for Rigid Bodies. In *ACM Transactions on Graphics* 24(3):946-956*,* 2005

[6] Moore M. and Wilhelms J. Collision Detection and Response for Computer Animation. In *Computer Graphics,* Volume 22, Number 4, 1988.