Joshua Newth
CS297
Professor Pollett
5/14/2012

Rigid Body Physics Simulation

My project investigation began with the rough plan of implementing the graphics and physics systems required for a modern videogame. I took as my inspiration **The Neverhood**, a videogame from the 1990s that used stop-motion photography to create a plasticine world with deformable characters and terrain. It was my goal to mimic in realtime those carefully and laboriously prepared animations.

At the same time, I was applying to the videogame middleware provider Havok whose flagship product *Havok Physics* is the industry-leading physics engine behind many modern AAA videogames. Getting that job has been one of the sweetest triumphs of my professional career. Through this work I have had the opportunity to immerse myself in the study of the physics engines used in modern videogames, known as rigid body physics engines. My study and my work have (delightfully) merged in to a unified intellectual exploration.

This paper is my attempt to document the work I have done this semester and to outline a research question appropriate for a Masters in Computer Science at San Jose State University.

Introduction

A modern realtime physics simulation is a technological marvel. Accurate simulation of the interplay of forces and geometry in the motion of bodies in space is incredibly computationally intense, far too expensive for the 30 frames-per-second minimum requirement of modern day videogames. To plausibly simulate this behavior at the high framerate required of most videogames is marvelous, indeed. Some background information is valuable in describing the work I did over the semester.

Broadly stated, physics simulations operate in a series of discrete timesteps. At each timestep, forces are applied, velocities are updated, potential collisions between rigid bodies are detected, constraint forces are computed and applied, and finally the rigid bodies are integrated to their new positions, orientations, and velocities. This computation must take place many times each second to insure that objects move in a physically believable way according to the requirements of the physics system.

In reality, this simple description glosses over a great deal of complexity, complexity I will return to later. For now, though, it is enough to describe the basic challenges of implementing even a simple physics engine.

Deliverable 1: Collision Detection with GJK

One of the major components of a physics system is detecting where a collision will occur between two rigid bodies. Arbitrary collision detection is quite hard, so most rigid body physics system make several assumptions to simplify this problem. The first and most important is that all shapes will be convex hulls. A convex hull around a set of points can be thought of as the resulting shape if a rubber band was wrapped around the set of points. The requirement that shapes be at least convex hulls provides the flexibility required to represent a large number of shapes as well as allowing for some clever mathematical optimizations.

Once we make the simplifying assumption that two objects are convex hulls, there exists a wonderful algorithm that can detect collisions. This algorithm is called GJK, after its inventors Gilbert, Johnson, and Keerthi.

Minkowski Difference

The basic tool of GJK is the Minkowski Difference. It can be thought of as an object where each point is the result of a point in object A - a point in object B.

Minkowski (x, y) = (Ax - Bx, Ay - By)

For the purposes of finding collisions one can simply consider the vertices of each object, but it is important to remember that we are talking about the infinite number of points that make up the 2D area (or in 3D, the volume) of each object. It is also worth drawing out a few to get the concept fixed in your brain. Heres a screenshot of Paul Firth's applet:

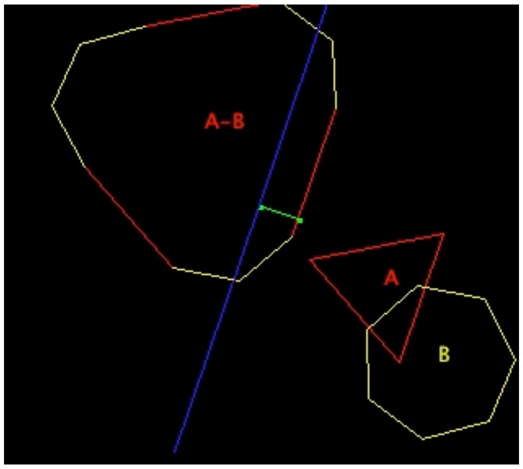


Figure 1. Minkowski Difference

The weird object is the Minkowski Difference of the triangle and heptagon. We also note that if two objects are overlapping then there exists at least one point on each object A and B are the same, so the equation becomes:

Minkowski point (x, y) = (Ax - Bx, Ay - By) = (0, 0)

This means that if two convex hulls collide, then the Minkowski Difference of the two objects contains the origin. GJK is how to implement the question: Can I draw a simplex using only points on the surface of the Minkowski Difference object that contain the origin? If so, I have detected a collision. If not, then there is no collision. In 2D, a simplex means a triangle and in 3D simplex means a tetrahedron. Both are the smallest set of points that can contain another point in 2D and 3D, respectively.

The Support Function

The key to using GJK is to be able to compute the point on each object A and B that lies farthest along a search direction. Although the Minkowski object is composed of all possible combination of points, we can solve GJK using only the vertices of of our polygons. In addition, although it is mathematically perfectly reasonable to pick a point not one of the vertices of the polygon, programmatically it is more sensible to only consider the points that make up the polygon. The way we do this is with the Support function, which given a direction to search along returns the vertex on an object that is farthest in that direction.

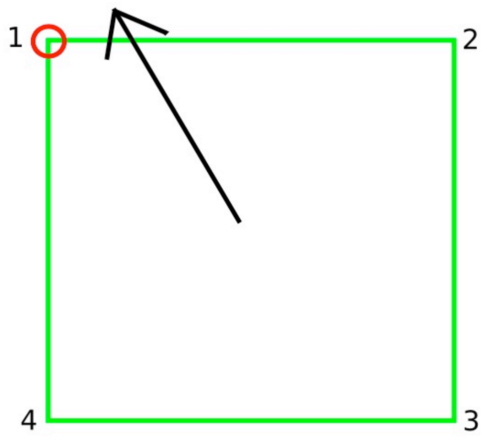


Figure 2. The Support Function

Point one is the farthest point on the object along the search direction. Also, rather than explicitly compute all the points of the Minkowski object and then calling our support function on that, we can simply call the support function on each of our two simple polygonal objects and compute the appropriate Minkowski point using the relationship below. Here's the pseudocode for the support function:

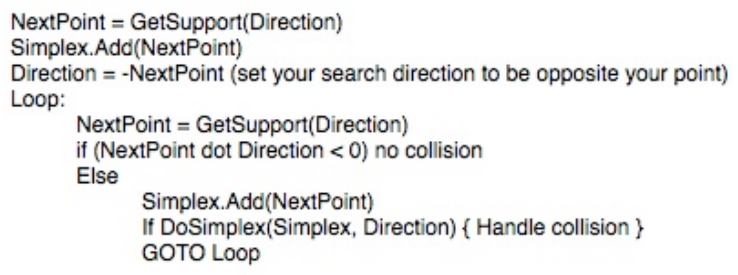
Function DoSupport()

Point on Minkowski along D = (Farthest point on A in direction D) - (Farthest point on B in direction -D)

Observe that we use "-D" in the support function for object B.

Evolving the Simplex

The core of the GJK algorithm is the concept of the simplex, the smallest set of points on the surface of the Minkowski object that contain the origin. The algorithm proceeds by evolving this simplex until the origin is contained. If it is contained, there is a collision. If the simplex is fully evolved and does not contain the origin, no collision has occured. In pseudocode:



```
NextPoint = GetSupport(Direction)
Simplex.Add(NextPoint)
Direction = -NextPoint (set your search direction to be opposite your point)
Loop:
    NextPoint = GetSupport(Direction)
    if (NextPoint dot Direction < 0) no collision
    Else
        Simplex.Add(NextPoint)
        If DoSimplex(Simplex, Direction) { Handle collision }
        GOTO Loop
```

To evolve the simplex we repeatedly choose a new direction to search for the next candidate point to add to the simplex and repeatedly throw away points unecessary to our simplex. Thus our DoSimplex function examines the existing simplex and then sometimes updates the simplex (by removing a point) and always picks the next direction for us to search in. In our formulation we never call DoSimplex on a simplex of size 1, so let's consider the 2-point simplex.

Evolving the Simplex: Two Point Simplex and Voronoi Regions

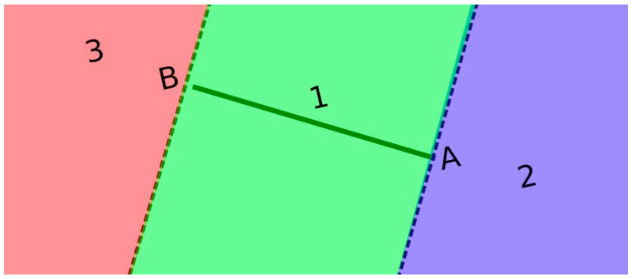


Figure 3. Two point simplex

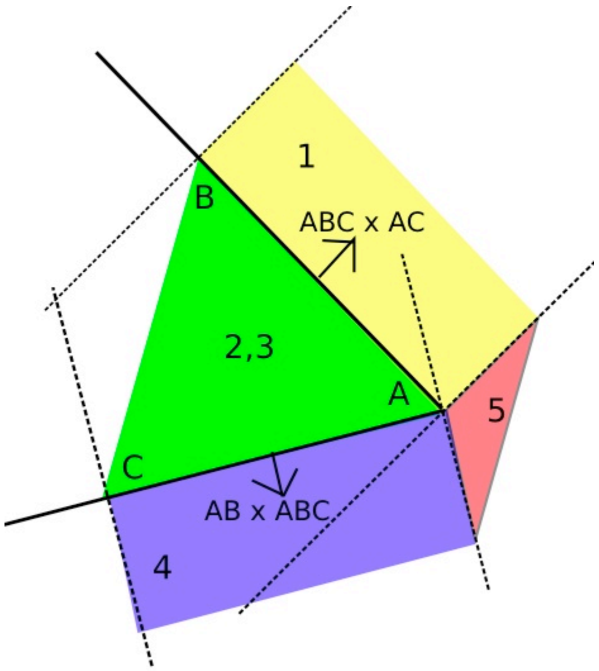


Figure 4. Three point simplex

Without going in to detail already covered in other documents, these two images show how the simplex is evolved using the concept of Voronoi regions to select where the origin might be. In the first image, our two point simplex forms a line, and we select the next search region and direction based on some careful reasoning about our simplex. The three point simplex works in the same way. The exit case is when we discover that region 2,3 contains the origin. At this point we have discovered the Minkowski object contains the origin and a collision has occured.

The output of the GJK algorithm is merely true or false but there are two useful tools, EPA and GSK, that can extend or build upon GJK to provide more useful collision detection information. In collision resolution, the next step of a rigid body physics simulation, we require not only the location where the collision occurs but also the separating normal (the line of collision) and the penetration depth (how much the bodies must be separated to prevent overlap). I implemented both EPA and GSK in the course of my research but as such details are mere extensions to GJK I will not cover them in any detail here.

Deliverable 2: Collision Resolution

The second component of physics simulation is collision resolution, where the contact points discovered in our previous work is used to adjust the motion of rigid bodies to maintain the various constraints our simulation places on them. As before, some background is necessary.

An accurate and complete mathematical description of the motion of a physical object under the influences of various forces is sometimes called “the equations of motion”. There are various ways to determine the equations of motions, from the original formulation by Newton, where all forces are solved for explicitly, to Lagrange’s equations, with its concept of zero work constraint forces, to the more modern Cain’s method, which describes the motion of a body using “motion variables”. Although these methods are all describing the same physical phenomenon and must therefore be equivalent, it turns out the Lagrangian form of the equations of motion lends itself to computation and simulation.

A complete explanation of the Lagrange formulation of the equations of motion is beyond the scope of this paper (or its author) but I shall attempt summarize the relevant information, largely gleaned from the GDC2008 presentation of Erin Catto, the author of Box2D and one of the leading lights in game physics.

Constraint forces can be thought of as forces that operate on the rigid bodies. These forces are special in that they do no work, that is, they inject no new energy in the system. A useful mathematical tool for computing these forces is the Jacobian, which can be thought of a matrix transform between “Cartesian space”-- that is, free motion, and “constraint space”--the space of possible configurations of the system that preserve the constraints (such as no interpenetration). This is expressed as

$$\dot{C} = \mathbf{J}\mathbf{v}$$

Where C is the first derivative of our constraint expression (which in turn depends solely on configuration, that is, position and orientation), J is our Jacobian matrix and V is our velocity matrix. Figuring out how to describe a constraint is something of an art, so we confine our observations to no-penetration. By analogy, the force on a body can be expressed as

$$\mathbf{F}_c = \mathbf{J}^T \lambda$$

where Fc is the “Cartesian space” force vector, Jt is the transpose of our Jacobian, and lambda is our “constraint space” force vector. Without going in to too much detail, our non-penetration constraint is written:

$$\begin{bmatrix} -\mathbf{n} \\ -(\mathbf{p} - \mathbf{x}_1) \times \mathbf{n} \\ \mathbf{n} \\ (\mathbf{p} - \mathbf{x}_2) \times \mathbf{n} \end{bmatrix}^T$$

Where n is the vector representing our contact normal (from our previous collision detection routines). p is our contact point (from the collision detection) and the x's are the displacement between the body's center of mass and the contact point.

Constraints for friction, motors, string, rigid links, ragdolls, etc can be described in this form using different Jacobian matrices to map between "Cartesian space" and "Constraint space". Once we are able to express the Jacobian, we can compute lambda, the impulsive force that operates on the body to preserve the constraints.

It is important to remember that the equations of motion describe a body, but these bodies may interact on each other simultaneously at each time step. Therefore it is necessary to consider all of the impulses simultaneously to arrive at a correct overall solution. This leads to two different types of solvers: global solvers, which attempt to solve all constraints simultaneously, and iterative solvers, which solve each constraint in turn for multiple cycles, in the hopes that the system will converge. This convergence is generally measured by the magnitude of the largest corrective impulse applied to the simulation in a given iteration. For simplicity's sake iterative solvers are typically used.

Last, we must note that our constraints are turned in to impulses, that is, forces that are so large but act for so short a time that it is easier to consider them making instantaneous changes in a body's velocity. Rather than compute Cartesian forces which produce accelerations, and integrate these to find new velocities, we apply our impulses to adjust the bodies' velocity directly.

The procedure for computing lambdas is as follows:

Compute an "uncorrected" velocity based on the external forces

$$\overline{\mathbf{v}}_2 = \mathbf{v}_1 + h\mathbf{M}^{-1}\mathbf{F}_a$$

Where V1 is the velocity of the body from the previous timestep, deltaT is the timestep (typically a single physics frame of 16ms, Fa is the vector sum of forces, and M-1 is the inverse of the mass.

We then compute the corrected velocity by adding the impulses to the uncorrected velocity:

$$\mathbf{v}_2 = \overline{\mathbf{v}}_2 + \mathbf{M}^{-1}\mathbf{P}_c$$

Where P is

$$\mathbf{P}_c = \mathbf{J}^T\lambda$$

We can compute lambda, our impulse, as:

$$\lambda = -m_C\left(\mathbf{J}\overline{\mathbf{v}}_2 + b\right)$$

where mc is the effective mass (which takes in to account the inertia tensor) and is computed as:

$$m_C = \frac{1}{\mathbf{J}\mathbf{M}^{-1}\mathbf{J}^T}$$

Although the math is quite dense, and care must be taken when formulating this algebra in a manner solveable by a computer, it is sufficient to create a constraint based physics system. I implemented a number of extensions (such as Baumgart Stabilization and lambda clamping) to improve numerical stability, convergence, and performance. These are well documented improvements to cure well-known ills of iterative rigid body solvers.

The final result is Havok2D, a very simple 2D physics system that performs convex hull collision detection and resolution, with an iterative constraint solver.

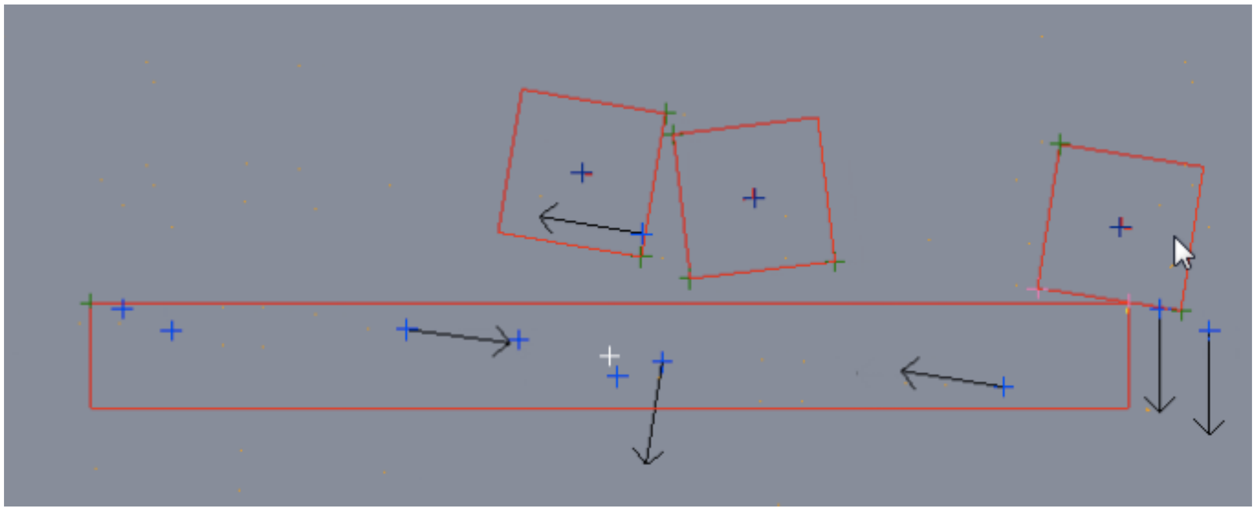


Figure 5. Rigid Body Collision Resolution

The arrows and blue dots show an overlay of the Minkowski object that is computed for each pair of objects. In this simulation there are

n*(n-1)= 4*3 = 12 pairs

that each must be checked for collision. In a professional physics engine this n^2 growth would be managed using a “broadphase”, where objects are checked for potential overlap using axial aligned bounding boxes (AABBs) using a “3-axis sweep” algorithm before being passed to the GJK solver. This is a very important optimisation but unecessary for a toy simulation like this.

Deliverable 3: Experiments with Box2D

After developing a collision detection and collision resolution system, the next research step was to explore the capabilities of existing physics simulation packages. There are four packages commonly used for physics simulation:

Package	Source	License	Limitations
Havok Physics	Headers only	Intel free license for independent developers	
Bullet Physics	Yes	zlib	Unreadable code
Box2D	Yes	zlib	2D simulation only
PhysX	No	PhysX licensing	Restrictive Licensing

Table 1. Physics Engine Comparison

Box2D is a smaller readable codebase and seems appropriate for further experimentation. Havok and Bullet are extremely capable, but are intended primarily for 3D applications and as such have a lot of additional functionality that would only complicate the intended experimentation. PhysX is primarily intended to drive adoption of NVidia graphics cards and is also intended for 3D applications. Thus Box2D was selected as the appropriate platform for further experimentation.

Experiment 1: Configuring and Developing with Box2D

Having examined three commercial physics systems in the course of this research it is clear that there is a remarkably consistent design philosophy. Essentially, physics simulations are constructed in the following manner:

1. Create shapes, which are purely geometric and carry no physics information
2. Create rigid bodies which have no geometric representation but have mass, friction, restitution, and inertia, as well as linear and rotational velocity.
3. Combine shapes with rigid bodies
4. Add constraints between rigid bodies and between rigid bodies and Cartesian space. For example, two objects may have a constraint that allows one to rotate around the other with 1 degree of freedom. These constraints may be thought of as hinges, springs, rope, chain, and motors (which actively drive the system).
5. Add these rigid bodies to a physics world, which contains actions (like mouse inputs) and forces (like gravity, or wind, or buoyancy)

To learn the particulars of Box2D, I decided to simulate rope. Box2D ships with a great testbed demonstrating many typical use cases (colliding boxes, springs, rope, bridges, etc) but I wanted to go through the process of configuring my own simulation.

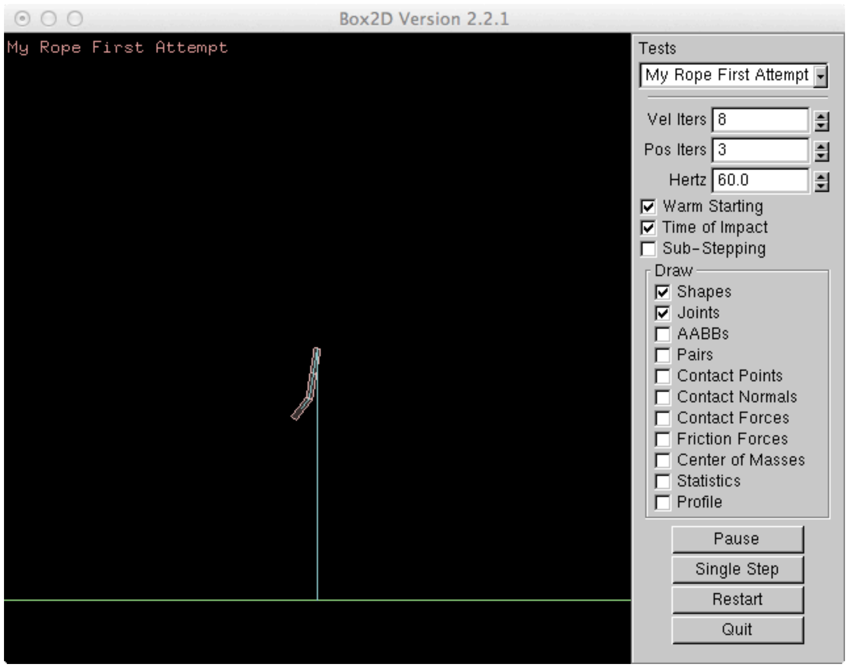
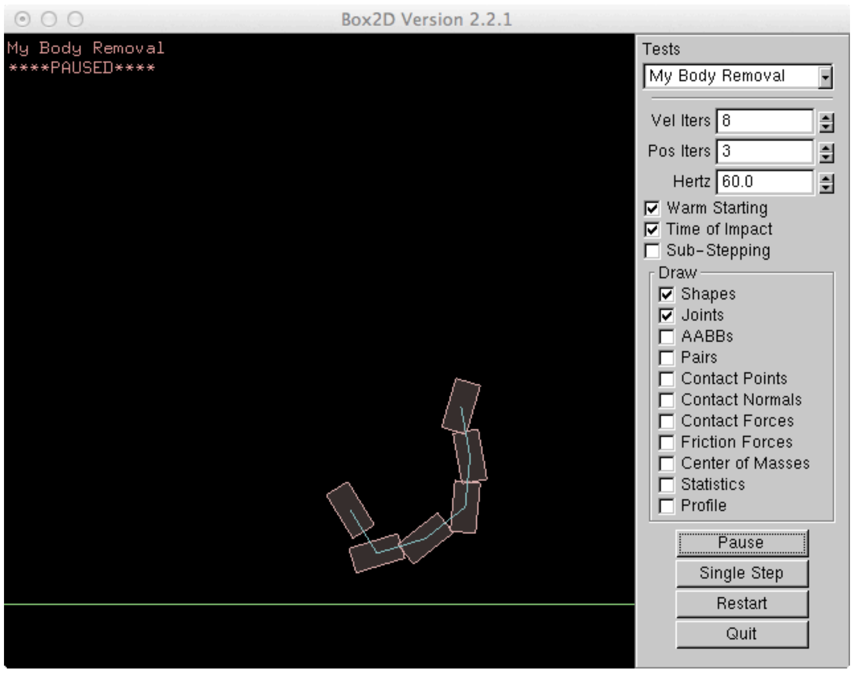


Figure 6. Simulating Rope with Box2D

The green line represents the earth plane (a fixed rigid body). The blue line represents the constraint between the earth and the top of the rope. This “rope” is actually modeled as a series of discrete chain links. These links are represented as pink rectangles. The simulation also demonstrates the use of collision filtering to prevent rigid bodies that are connected by a constraint and in close physical proximity are prevented from colliding. These bodies can overlap, which allows some numerical and physical inaccuracy but produces a much more stable simulation.

After mastering the basics of physics simulation in Box2D, I wanted to investigate the possibility of non-rigid body simulation. All of these physics simulations depend on the concept of rigid, impenetrable bodies. Bodies can not deform under load and as such do not absorb energy as deflection or deformation. It may be possible, or even desirable, to have a system that can approximate deformable bodies. For example, we consider a simple cantilevered (ie fixed at one end beam) under a point load at its distally unsupported end. In the real world, this beam would deflect. If the stress remained in the beams elastic region (defined in terms of its material properties) then when the load was removed the beam would return to its undeflected shape. This restoration would be accomplished because the beam has stored the work that went in to deflecting it and would use this stored energy to restore its original shape. If the load exceeded its elastic stress, the beam would undergo permanent deflection and would no longer return elastically after the load was removed.

An approximation of beam deflection would need to subdivide the rigid beam in to smaller bodies to show the geometric deflection of the real world beam. For example, if a large beam was deflected to 20 degrees, it could be modeled as four beams placed end to end, with 5 degrees of deflection between each beam. This might approximate the curve of the original deflection. The preliminary investigation was to accomplish the geometric approximation of body subdivision at runtime. The results of this experiment are shown below.



My Body Removal
****PAUSED****

Tests: My Body Removal
Vel Iters 8
Pos Iters 3
Hertz 60.0
[✓] Warm Starting
[✓] Time of Impact
[ ] Sub-Stepping
Draw:
[✓] Shapes
[✓] Joints
[ ] AABBs
[ ] Pairs
[ ] Contact Points
[ ] Contact Normals
[ ] Contact Forces
[ ] Friction Forces
[ ] Center of Masses
[ ] Statistics
[ ] Profile

Figure 7. Body Subdivision

It's trivial to replace a rigid body with a set of bodies in Box2D. This is simply a matter of keeping track of beam locations and attachment points for the rotational joint contraint between successive blocks.

The bigger problem is to capture the concept of the stored strain energy, and to drive the constraints so that the beam relaxes back to a normal shape after the beam deflection has occurred. This restoration must have some concept of the elastic and inelastic deformation which is a function of the material, geometric configuration, and total deformation. No physics engine currently contains the concept of stored energy as beam deflection.

CS299 Research Outline

The work this semester familiarized me with an interesting and important aspect of videogame technology. I also discovered some of the performance challenges and limitations of rigid body solvers, namely, collision detection and deformable bodies.

The collision detection outlined here assumes discrete simulation timesteps. This is a reasonable assumption for lots of simulation situations, but it leads to what is known as the "bullet through paper" problem. This occurs when a high speed object is in frame 0 on one side of an object. In frame 1 it has "passed through" the other object. Because collision detection (and resolution) takes place at discrete intervals these two objects never interact. This leads to physically unbelievable situations.

The current solution to this problem is called "continuous physics", where certain objects (decided typically by the user) are shape-cast through the physics world at each timestep. Because the shapecast captures all possible positions for the object at any point in a continuous time interval, it is able to detect collision occuring at intermediate timesteps. These intermediate events are called "time of impact" events, or TOIs. The resolution of this TOI may lead to further interactions (imagine an object A colliding with object B and bouncing off to hit object C), so they are computationally expensive and care is taken in physics systems to only use continuous physics when necessary for accurate simulation.

A new approach being developed is called “speculative contacts” which attempts to provide continuous collision detection without resorting to TOIs. My thesis project for CS299 will be to extend Box2D to include speculative contacts. This approach will then be compared against Box2D’s current continuous collision detection approach (TOIs) for performance and stability. It is also hoped that this research will be a meaningful contribution to this excellent and widely used open source physics engine.