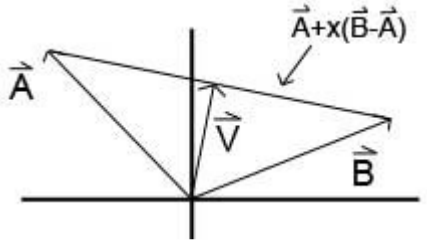
Mark Walker
Senior Project Progress Report
April 3, 2006

Thus far in the project, the goal has been to develop a flexible physics engine that can simulate multiple simultaneous rigid body collisions between circles and non-rotational squares. Attempts have been fairly successful with a few minor glitches, but the level of sophistication of the engine limits the scope of its applicability. The code has not yet been optimized or thoroughly reviewed to check for small errors, as it has become apparent, given my own goals for this project, that a more effective system needs to be implemented.

What has been done:

The first steps in this project were made on paper by deriving formulas for basic primitive tests that would be used in collision detection.

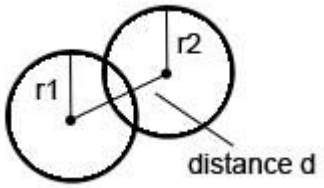
Distance from a Point to Line:
Given a point C and two vectors A and B that point to the ends of the line segment, find the exact point between A and B that is closest to C, and the distance from C to that closest point. Let x be a decimal between 0 and 1, and V be the vector from the origin to the closest point.



$$x = -A \cdot \frac{B-A}{(B-A)\cdot(B-A)}$$

This computation is used when performing a collision test between an AABB and circle.

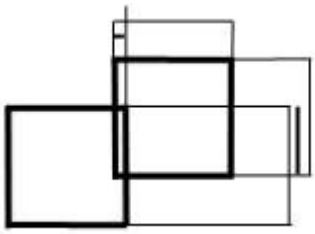
Circle/Circle collision detection:
This is the easiest test to perform due to their symmetry. Two circles overlap if the sum of their radii is less than the distance between them.



$$d < r_1 + r_2$$

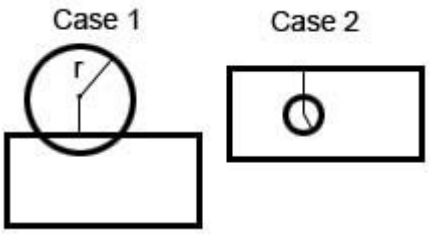
Circle collision takes place if and only if the above is true.

AABB/AABB collision detection:
Collision between two non-rotational boxes is also rather simple. According to the Separating Axis Theorem (SAT), any two squares intersect if and only if they overlap along both axes parallel to their sides (for AABB's, these are the x and y axes). The SAT will also be useful for OBB's (Oriented Bounding Boxes).



Two boxes and their projected axes are shown to overlap by the medium-thick lines. This theorem also works for rotational convex polygons.

Circle/AABB collision detection:
For this test, there are two cases. The first is if the circle's center is outside the box, but its perimeter is intersecting with a side. The point/line distance algorithm is used to find the closest distance between the center of the circle and each edge of the box. If one of those distances is less than the radius of the circle, then there is a collision. The second case is if the circle is completely enclosed by the box so that it is not hitting any sides, and the circle is simply tested to see if it is contained within.



The two cases considered in a circle/AABB collision.

Collision Response:

These collision tests are relatively easy compared to the physical responses they invoke, which is where the overall design of the engine becomes very important.

The first design issue to consider is exactly *when* the collisions are going to be tested for. The original method I used tested for collisions after they already occurred, which forced the object back. However, I found that this became extremely difficult because it involved going back a time step to find *where* each object had come from. In fact, this

was virtually impossible without having to save the previous environment state from the last time step, because there are so many factors that must be accounted for. Take, for instance, a scenario in which a stationary box is hit by another box coming at a very fast speed from the top left corner. In order to find whether the box was hit on the left side, or on the top, its position had to be tracked back by one time step. This could be approximated by simply moving it back along a trajectory determined by its velocity and constant acceleration. But there may have been numerous other factors that could have influenced its path such as other collisions or forces. Thus I found it impractical to wait for collisions to happen before responding, so I now *predict* when a collision is about to occur, and respond the instant before it hits. This ensures that I know where each object is coming from, and avoids having to physically move each object back so that they do not overlap each other.

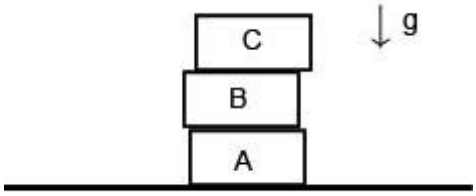
The next design issue to consider is the order in which objects respond to collisions. There are two ways to do this: sequentially, and simultaneously. Sequential ordering has each object respond as soon as a collision is detected. In other words, each object is checked one at a time for collision as usual, and a response is triggered when the object is tested for collisions. Simultaneous ordering is when all collisions found first before responding, then it calculates reactions at the end depending on where each object is, not where it is going. Currently, I have mixed these two options so that each object tested sequentially, and reacts to collisions only by changing its physical state (velocity, acceleration, etc.) and moving so that it is not intersecting with the object it is being tested against, but it "acts" using a simultaneous ordering. "Acting" refers to when it is moved due to its velocity and acceleration. A rough outline of the code is:

1. Get an object, A, and test it against every other object, B.
2. If A and B are overlapping, move A back so that they are no longer intersecting, or until a maximum number of back-steps have been performed.
3. Change the velocities of both A and B as needed
4. Once all collision tests have been completed, add each object's acceleration to their respective velocities, and move it according to their velocity vectors.

This has many issues and is still being experimented with.

Simulation Problems and Corrections:

Often when one object on top of another object non-stationary should have come to rest, it would jitter. This was due to the fact that I had not taken into consideration the normal force that objects exert on each other at rest in an environment subject to gravity (or any force). And while the normal force of one object on the ground on another object is easy to calculate, the issue becomes much more complicated to program once many objects stack on top of each other. Consider the diagram of a stack of boxes:



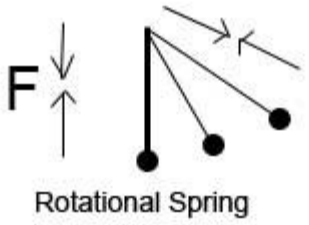
A stack of boxes, subject to a gravitational force, at rest on the ground.

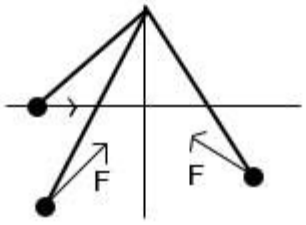
The problem arises with stacks of object such as the one above. If at equilibrium (at rest), the force that block B exerts on block C is equal to the weight of C. Similarly, the force that A exerts on B is equal to the sum of the weights of B *and* C. Then, the force that the ground exerts on A is equal to the sum of the weights of all the blocks. I have not been able to come up with a practical solution to this problem other than a recursive function that would attempt to calculate the sum of the forces on a given object. In other words, if I wanted the sum of the forces on A, I would have to have the force exerted on A by B and the force exerted on A by the ground, so, in the exact same way, I would ask B what the sum of the forces on it without A is, which would involve asking C what the sum of the forces (except for from B) is. Since C is only affected by gravity (except for the force from B), the recursive function could "unwind." I could then return the call from B to get the forces on C, which would return the call for the forces on B to A. In theory it sounds reasonable, but certain scenarios could result in infinite loops or strange results. It just isn't a very safe way to go about a simulation.

The solution is to put objects that have a kinetic energy below a certain threshold into a "sleeping" state, in which they are treated like stationary objects, until awoken by some other object with enough kinetic energy. Objects can only go to sleep if they are below that threshold, and are colliding with a stationary (static) or sleeping object. This is where I have had to draw the line between realism and simulation. Parameters can be tweaked to give the best results, but the engine has become merely an approximated simulation.

Constraints:

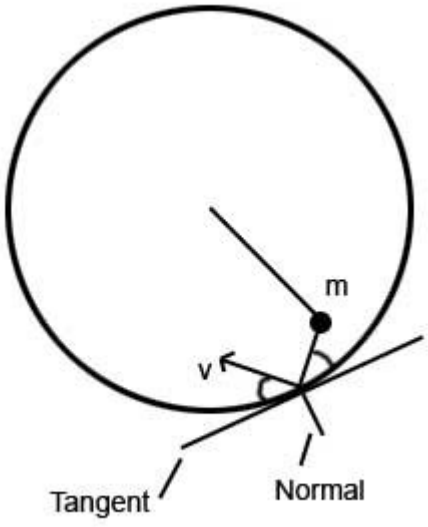
Several constraint classes have been added to the project. There are two different kinds of springs: "bendy" and "rotational" springs. "Bendy" springs are, in effect, systems of two springs, one in the x direction, and one in the y direction. Rotational springs maintain a straight, rigid shape, can freely rotate about the pivot point, but will only force an object away from or towards the center. This would be like a metal pole, attached to a pivot point, with a spring wrapped around it. Note that Hooke's Law is used to calculate the force on the object ($F = -kx$), where k is passed in as a parameter in the spring classes.





Rotational springs apply a force parallel to the direction from the center to the object. Bendy springs will apply a force toward a point of equilibrium.

The other kind of constraint is an ideal, massless rope, which effectively constrains an object to a circular area of a given radius. It works by conserving momentum parallel to the tangent of the circle at the point of contact. In other words, an object reaching the end of the rope will be pushed back a certain amount into the circle, depending on the angle is makes with edge of the circle on contact.



The momentum of mass m is conserved in the direction parallel to the tangent line, and reversed in the direction parallel to the normal line, resulting in equal angles of incidence and refraction with respect to the edge of the circle.

Ray Tracing:

Ray tracing is often used in 3D graphics engines to render shadows, mirrors, water reflections, and other special effects, by literally extending a line in some direction and finding where it first intersects with an object. I am using ray tracing as an added feature that would be useful for fast-moving objects, usually bullets and other projectiles. Its use can also be extended to improve collision detection. For each kind of bounding area, a different algorithm is used to find the points of intersection (if there are any). Note that there are almost always two solutions if it does intersect: one going in, and one coming out (if the ray is tangent to the circle then the radical in the equation must be 0). The program then determines which of all the points found is the closest, and returns that location.

For bounding circles, the following is true:

$$x\hat{r} = Point\ of\ Intersection$$
$$x = (\hat{r} \cdot \vec{c}) \pm \sqrt{(\hat{r} \cdot \vec{c})^2 - |\vec{c}|^2 + R^2}$$
$$(\hat{r} \cdot \vec{c})^2 - |\vec{c}| + R^2 \geq 0$$
$$x \geq 0$$

If the directional ray r intersects with a circle at c with radius R, then the above equations are always true. The normalized ray will intersect at xr.

For AABB's, the algorithm tests to see if the ray intersects with both the vertical and horizontal "slabs" of the box. For a normalized ray r, and a box at c with width w and height h, and for:

$$x_1 = \frac{c_y \pm h}{r_y} \qquad x_2 = \frac{c_x \pm w}{r_x}$$

either value of x1 is greater than 0, and between the values of x2, or vice versa, then the ray intersects at xr (where x is the single value between the other two values). If it does intersect then there will always be two solutions for the point of intersection: one going in, and one going out (unless it passes through a corner).

Demonstrations:

Thus far I have created a number of demonstration applets to show off the engine's capabilities. Currently I'm working on a rope demonstration applet that also uses ray

tracing. The player can aim and attach a rope to the borders of the area. These are posted on the internet at:

http://wwwalker.net/mark/

Note that you will need to make sure you have the Java runtime environment installed on your computer to run the applets (usually it prompts you automatically to install it).