ITN, Norrköping September 7, 2012

Linköpings universitet

Real time multi threaded Rigid body simulation

REAL TIME MULTI THREADED RIGID BODY SIMULATION

TNCG14

Christopher Birger chrbi049@student.liu.se
Erik Englesson erien139@student.liu.se
Anders Hedblom andhe893@student.liu.se

Contents

1 Introduction

It's getting more and more common for game developer to use physics engines in their games. Sometimes, a physics engine can be a necessity, but even when simulations could be replaced with pre-rendered animations, people tend to take advantage of a physics anyway, and there are of course good reasons for this. First of all, it is very difficult to animate objects manually to make realistic and natural motions. Secondly, computers are getting so powerful that adding some real physics doesn't really slow them down too much. One of the secrets lies in the ability to do multi threaded calculations, which speeds up the process immensely. The problem lies in how to write efficient code that can run in parallel. This report describes two of the most fundamental components in a physics engine; collision handling and rigid body calculations, as well as how to speed up the calculations by using multiple threads.

2 Problem

What should be achieved by creating a physics engine? That of course depends on its purpose. Mostly, it is about handling moving objects that collide with other objects and to calculate the result from the collisions. The laws of physics says that no object with mass can have its kinetic or potential energy changed without having any forces applied to it. So what really happens when two objects collide is that during a small time period, which varies depending on material etc., forces are applied to all involved objects so that the sum of the forces are zero. However, in the computer world where time is discrete, we will have to cheat a little bit.

3 Method

There are many ways to do physics simulations, all with their pros and cons. This project handles colliding convex polyhedra objects and includes:

- The structure for handling rigid bodies.
- A broad phase algorithm to quickly find out whether any pair of objects might be colliding.
- The GJK algorithm to find out exactly, in a narrow space, if a pair is colliding or not.
- The EPA algorithm to calculate the colliding points. Using the results of GJK.
- The LCP method to create impulses as a collision response.
- A simple OpenGL viewer, with shaders, to visualize the simulation.
- OpenMP for multithreaded usage.

3.1 Rigid bodies

Rigid bodies are used to resemble solid objects without the capability of deformation, and can be defined by several different attributes. These can be split into two types; linear and angular entities.

Linear entities: mass (m), momentum (p), position (x,y,z) and velocity (v). Angular entities: inertia (i), angular momentum (l), rotation (r) and angular velocity (w).

One can think of a rigid body as a point, which corresponds to the body's center of mass. Vertices surround this point to create the shape of the object and are used both during collision detection and for the rendering part. However, movement of objects are always splitted into two distinct parts; linear and angular movement. Linear forces affect the center of mass, whereas angular forces (torque) affect the rotation of objects. When an object rotates, vertices adapt their positions with different transformation matrices.

Linear entities are affected by forces (F) and the angular equivalents are affected by torque (T) (though mass and inertia stay constant for the whole simulation) and, as said, the two act separately on the rigid body.

The basic idea is to update the colliding objects with impulses. An impulse can be seen as an infinitely big force that is applied during an infinitesimal time. In reality, this is impossible, but can be approximated with a very large force during a very short time, for instance a discrete simulation time step dt. The forces will then affect the velocities of the objects so that they move in a realistic way. But first, the forces have to be derived, which in turn depends on how the objects collide.

3.2 Collision detection

The collision detection system tells us whether or not any objects are colliding in the scene and how they collide. Normally, colliding objects do not penetrate each other. There are obviously cases where this happens; a bullet penetrating glass, for instance. But no object in the world is infinitely stiff. Even a stone to stone collision involves some form of distress in the material, so that forces can be applied over time, even though a very short time in this case. However, in a rigid body simulation, the object are considered to be completely stiff. No collision can change their form. This means that a collision is defined when two objects intersect one another. Though, there are methods that utilize the ability to predict an object's position by its trajectory, and prevent any penetrations by making a collision response before the actual collision has occurred, this report does not cover any of them. Our system is divided into two parts. First we do a broad phase test to find pairs of objects that might collide. These pair are stored in a list. The second phase, the narrow phase, takes each pair and runs a special algorithm to check if the the pair really collide. If they collide, this phase also calculates the points on the bodies that collide. Broad phase collision testing

3.2.1 Broad phase collision testing

Collisions between rigid bodies are decided pair by pair. So it would be possible to do a nested loop that, for every objects, do a collision detection with every other object. However, the time for this is O(n2), which means that if the number of objects is doubled, the time it takes to check collisions is multiplied by four. This is not desirable. It is not really necessary to loop over all objects twice. For instance, when the first object has been checked against every other object, the first one can be excluded when stepping to the next outer iteration. But it still takes a lot of time. It is possible to bypass this somewhat by doing a much more crude broad phase collision detection first. This only involves to check if objects are are intersecting on each separate world coordinate axis. So for each pair of rigid bodies, this is done:

1. Loop over all vertices and decide max and min positions, separate for all axes, for both body A and body B.
2. Check, for each axis, if the two intervals A[min, max] and B[min, max] intersect.
3. If any of two intervals do not intersect, there is no collision. If all intervals intersect, there might be a collision and a more narrow detection has to be calculated.

There are also ways to speed up the process even more by removing unnecessary collision check. This can be done by using a grid where the whole simulation takes place. If objects can be associated with different grid cells, it is only essential to check collisions with objects in the adjacent grid cells. Using a grid requires a nice data structure that can tell which grid cell an object belongs to, and all the objects contained by a specific grid cell. Using this broad phase method makes the axis intersection check less important but can still be used as a sort of mid phase collision detection.

3.2.2 Narrow phase

The narrow phase collision detection is far more complicated than the broad phase. This is where the collision detection actually decides if and where two rigid bodies collide. One problem that follows is the discrete time nature of computer calculations. The different times are considered as states. At a state right before collision, the time can be seen as t0. In the next state, t0+dt, the bodies will have penetrated each other with an unknown distance. The smaller time step that is used, the more shallow the penetration will be. Any how, this penetration has to be corrected, as no two objects should be allowed to intersect one another. One way of solving this is to use the bi-section method. When two objects have collided, time is stepped back to t0+0.5*dt. If the objects do not intersect here, the time step is once again halved and added in a forward motion. The state will be t0+0.75*dt. This back-and-forth method is repeated until a small intersection threshold is reached. However, it is also possible to cheat and just move one of the objects outwards the same distance as the penetration depth. The direction of the movement depends on the collision type and the collision normal.

Collision types There are several ways that convex objects can collide:

- Vertex-to-face
- Edge-to-edge
- Edge-to-face
- Face-to-face

Collisions with vertex-to-vertex and vertex-to-edge are extremely rare and will never happen in practice.

GJK To know exactly if two bodies intersect, the famous Gilbert-Johnson-Keerthi distance algorithm is used [1],[2]. It uses the principles of a Minkowski difference (MD) between the two bodies. The Minkowski Difference is a volume that consists of vertices created by the difference between every pair of vertices from the two bodies. If the origin is enclosed in the Minkowski difference, the two bodies are indeed intersecting. So the real problem here is to find out if the origin is inside the MD. The goal of the algorithm is to try to find a simplex (tetrahedron in 3D), built from four of the MD vertices, that encloses the origin. If such a tetrahedron can be found, the two bodies intersects. It starts off by picking any point in the MD. From this point, in direction to the origin, it looks for the MD-point that is farthest away. This point is added to the simplex to create a line. From this line, in the direction to the origin, orthogonal to the line, once again the farthest MD-point is looked for. If found, the simplex is now a triangle. The process is repeated to get a tetrahedron. If no tetrahedron is ever created, there is no collision.

EPA The GJK algorithm only tells us id two objects intersects. In order to calculate the correct forces in the proceeding step one has to know exactly the point on the bodies where the contact was made. When two objects touch the origin will be on the border of the MD. This means that the depth of penetration of two objects can always be found by calculating the shortest distance from the origin out of the MD. With convex objects in 3D, this will always be a vector parallel with a MD face normal. The purpose of the EPA (Expanding polytope algorithm) is to find the penetration depth and the points of intersection, depending on the collision type. For instance, we can have a vertex from body A that penetrates a face from body B. In that case, both the vertex position and the vector parallel with the face normal, from the vertex to the face, are needed in world coordinates. The EPA is done right after the GJK, and uses the simplex that the GJK derived. We know that the origin is enclosed by the simplex generated from GJK. It turns out that the point in the MD space that corresponds to the true collision points lies on the surface of the convex hull of the MD. Thus, EPA essentially expands the simplex generated from GJK until it finds the plane which is on the convex hull of the MD and is closest to the origin. This plane can then be used to determine the type of the collision and also the world coordinate of the collision point. The only extra thing we need to keep track of is that each MD point must know which world space coordinate points was used to create it.

Contact pairs For every intersection point, a contact pair is created. It is a simple struct that contains information about which two bodies have collided, what type of collision it was and the position of the collision. Again, this depends on the type of collision, but can for instance be a vertex position and a vector parallel with the face normal, which norm tells the penetration depth. All collision pairs are stored in a vector that is sent to the LCP solver. In this way, many collisions can be resolved during the same time step, which is important in order to allow object to lie still.

3.2.3 Parallelizing the collision detection

As said we implemented a grid structure to decrease the number of narrow phase checks. We define a uniform grid structure that covers the entire scene. Then we loop over all the bodies, calculate the bounding box of each object and add the index of the body in each voxel corresponding to the corners of the bounding box. Instead of defining a grid data structure we use a hashmap, therefore we only need to store the voxels containing a body. Since we don't know in advance how many objects will end up in a voxel we add the bodies dynamically in a list. In order to run this on multiple processor

we use a vector of multimaps. In runtime we know the number of processors so we can give each of the processor its own multimap. After this stage we basically have a list of $< hashindex, bodyindex >$ pairs. But we have not accounted for the fact that the list contains duplicates, e.g. the pair $< a, b >$ might also be added as $< b, a >$. Therefore, we apply a reduction scheme to remove the duplicates, to ensure that only unique pairs are sent to the broad phase. The broad phase checks each unique pair more closely as described in section 3.2.1, and adds each possible colliding pair into another list. Then we apply the narrow phase scheme which produces contacts. Again, since we do not know the number of contacts beforehand, they are added dynamically. As described earlier each processor uses its own list which is then reduced into a single list, which is sent to the physics response system.

3.3 Collision response

There are at least two different methods when handling collision response, sequential and simultaneous processing of the collision points. The simultaneous processing is more robust when handling challenging tasks like stacking objects, at the cost of a more complex implementation.

3.3.1 Simultaneous processing

One of the ways of doing simultaneous processing of collisions is by applying impulsive forces at the collision points. Consider the case of a stack of blocks. The block on the bottom is colliding with the floor and the second block. The second block is colliding with the bottom block and a block above etc. The forces keeping them from penetrating each other are all in some sense connected. The idea of simultaneous processing is to form an equation system based on this connection to compute the impulsive forces at the collision points all at once. This can be done by looking at the relative velocity between colliding objects and adding the unknown impulsive forces in the conservation equations, as described in [3]. What you end up with is an equation like this:

$$newRelVel = oldRelVel + \sum_i constant(i) * f(i) \quad (1)$$

where newRelVel is the relative velocity after collision and oldRelVel is the relative velocity at collision and the summation is over collision points and f(i) is the impulsive force for that collision point. This can be written in matrix form instead:

$$\mathbf{newRelVel} = \mathbf{oldRelVel} + A\mathbf{f} \quad (2)$$

where A is a symmetric matrix. The equation above is the actual equation we want to solve and this can be done by minimization, if the equation is modified. The modification is changing the components of $\mathbf{oldRelVel}$ vector to 0 if the component is positive and $2\mathbf{oldRelVel}$ if it is negative. $\mathbf{oldRelVel}$ is negative if they are trying to penetrate each other and positive otherwise. If we do this modification, a minimization of the length of the $\mathbf{newRelVel}$ vector will yield a collision response without loss of energy. This is easy to see, since when they are not penetrating we set $\mathbf{oldRelVel}$ to zero so the minimization will force f to zero,i.e no force if they are not penetrating. In the case of penetration $\mathbf{oldRelVel}$ is set to $2\mathbf{oldRelVel}$ and the minimization will make $A\mathbf{f} = -2\mathbf{oldRelVel}$, which will make the right hand side of equation 2 equal to $-\mathbf{oldRelVel}$, i.e. just a reflection of the velocity. The minimization is done by minimizing $|A\mathbf{f} + \mathbf{oldRelVel}|^2$ with some constraints. This is a linear complementarity problem(LCP) which was solved with Lemke's algorithm.

4 Result

The result of this project is a program that can handle multiple boxes, colliding with one another and a floor. Position, velocity, size and mass of the boxes can be initiated as desired before execution time. The calculations are programmed to be able to utilize multiple threads, which speeds up the execution time. OpenGl is used to render the scene with Phong shading and no shadows. Figure 1 and 2, in appendix A, are two frames from a simulation

where we let several hundreds of blocks collide with each other and the floor.

5 Conclusion and discussion

Implementing a stable rigid body simulation system is very difficult and there are lots of implementation "tricks" there are left out in the literature. The collision detection system we implemented is known for numerical issues which are difficult to resolve. When the penetration depth is very small the simplex GJK creates gets very thin which creates problems both for the GJK and the EPA. This is because both algorithm depends greatly on distance to plane comparisons and sign checking through dot products which are sensitive when simplex's are thin. The collision detection system is a very important part of any rigid body physics simulation because if collision normals or points are wrong the response system will produce incorrent responses and ultimately the simualtion can become unstable. Another issue with our implementation is that we dont use predictive collision detection. Therefore, pairs of bodies can collide and intersect which must be resolved by moving the objects out of each other. This is a trivial problem when there are only two objects and the time step is kept quite small. But, assuming a stack of blocks, all pairs of objects must be considered simultaneously. The intention was to use a predictive collision detection so the objects never actually penetrated each other and run the LCP solver at the real collision time without the need to move the objects out of each other manually. This would have increased the stability of the system in the case of stacking.

6 Further work

Friction Without friction, objects can not really change direction perpendicular to the normal of a surface they bounce against. I.e., if an object with arbitrary shape is dropped on a perfectly straight floor, without friction, it will bounce straight up, if there is any bounce at all. But we all know that if a stone is dropped on asphalt it will most likely bounce in a random direction. This is due to friction. Adding friction to the simulation would improve the realism vastly, as we almost never see environments without friction. Friction would also make objects come to a stop when sliding over a surface.

References

[1] C. Ericson, "The gilbert-johnson-keerthi (gjk) algorithm," *Siggraph 2004*, 2004.

[2] C. Zealot, "Gjk (gilbert–johnson–keerthi) @ONLINE," Sept. 2012.

[3] D. H. Eberly, "Game physics," pp. 295–538, 2010.

A Images

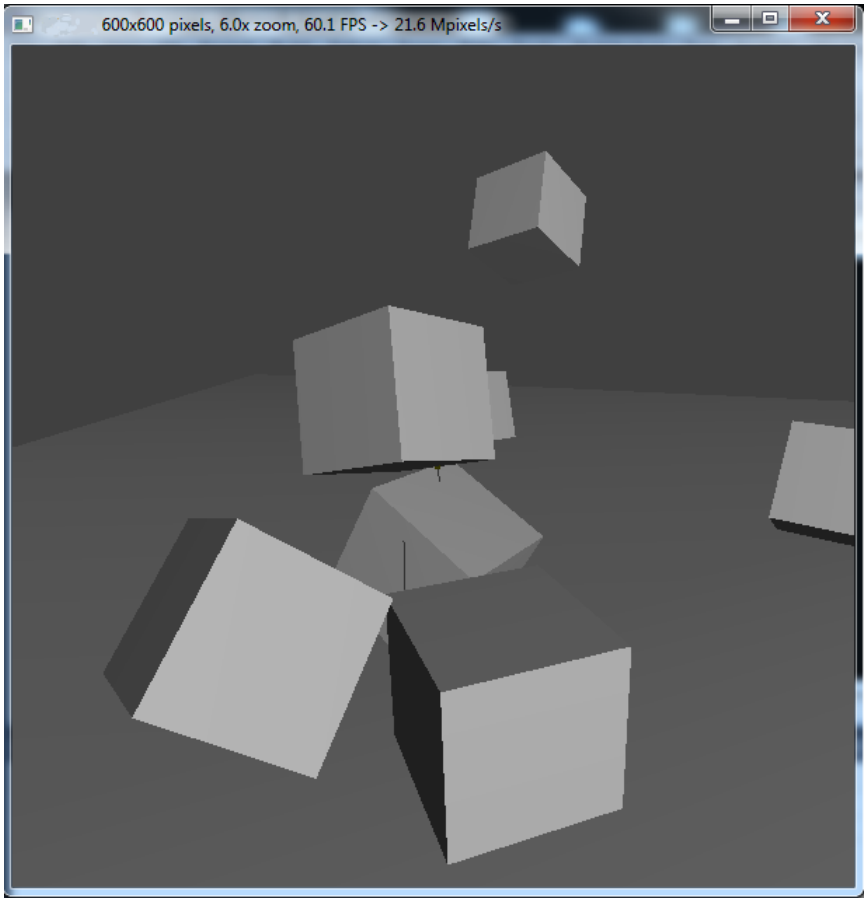

Figure 1: *Cubes falling onto plane.*

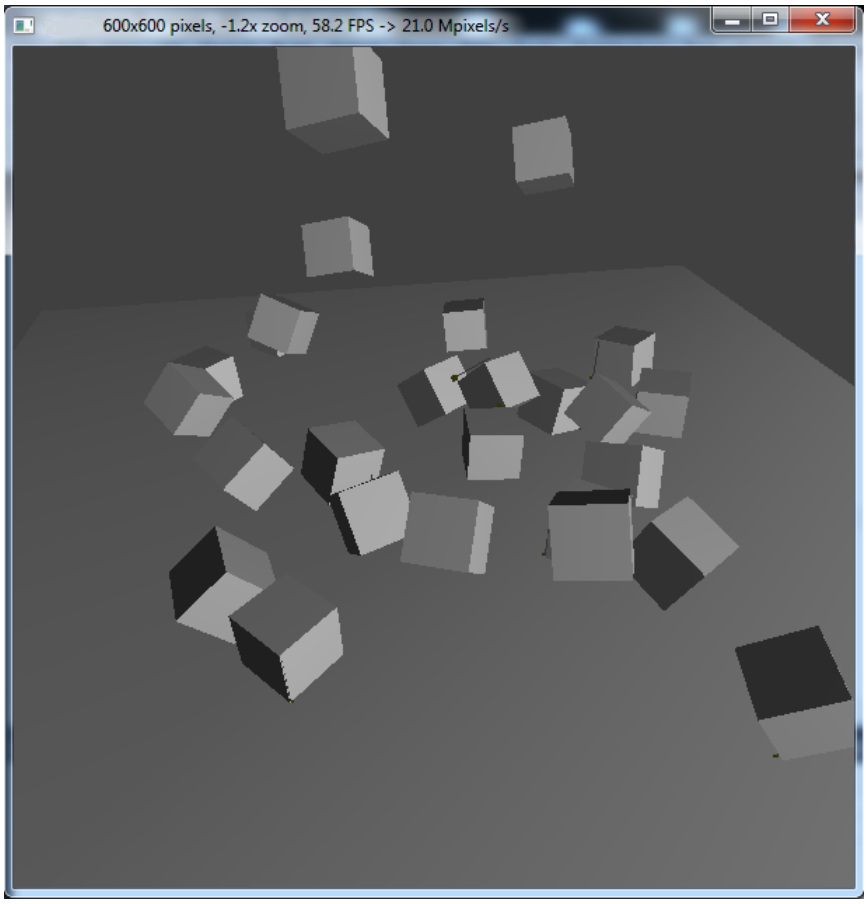

Figure 2: *Cubes falling on to plane.*