CS550 Physics Simulation Process Paper

Evan Collier

Overview:

The purpose of this project is to be a robust real-time simulation of 3D convex rigid bodies in a multi-threaded fashion.

Libraries:

The glm math library is used for the basic vector, matrix and quaternion arithmetic. Various glm objects such as glm::mat3, glm::quat, and glm::vec3 are used quite extensively.

The imgui GUI library is being used for all GUIs in engine.

The graphics engine was initially written by Nicholas Ammann and is being used with modifications with his permission. The graphics engine is written using OpenGL 3.3. Additionally, it is dependent on stb for image loading, assimp for model loading, glfw 3.2.1 for window management, and glad for OpenGL function bindings.

Engine:

The engine uses a component based architecture. Each object is comprised of a collider, a rigid body, and a mesh. Graphics is updated with a dynamic time step while physics is updated with a fixed time step. If the dynamic time step is larger than the fixed time step, I will update physics multiple times during the given frame until they take the same amount of time.

Since all objects are treated as convex hulls, any mesh file can be loaded from disk and used as the shape of the object. An inertia tensor will be generated based on the mesh data as well as a bounding box.

Additionally, my engine has a work-stealing Job System in which jobs can be queued up and done on different worker threads. Different workers will attempt to steal work from their coworkers if they have nothing to do. Otherwise, they will take an item from their own queue and work on that. This Job System has been used to significantly speed up my narrow phase to the point of being able to maintain high performance (60 fps) while supporting high numbers of objects and collisions (~1000).

Object Classes:

The class representing a 3D rigid body contains the following state information:

```
glm::vec3 x;                // position
glm::quat q;                // orientation
glm::vec3 P;                // Linear Momentum
glm::vec3 L;                // Angular Momentum
float mass;                 // Mass
glm::mat3 Ibody;            // Inertia tensor of the body
glm::vec3 force;            // Net force acting on the body
glm::vec3 torque;           // Net torque acting on the body
```

This information is sufficient for applying the effects of the forces and torques on the body and integrating the position and orientation of the object. The linear and angular momentum state values can be used to derive our linear and angular velocities. Similarly, the force and torque can be used to derive our linear and angular accelerations. Additionally, the base object contains the following information for debugging and validating state:

```
glm::vec3 v;                // linear velocity
glm::vec3 w;                // angular velocity
glm::vec3 a;                // linear acceleration
glm::vec3 wp;               // angular acceleration
glm::vec3 cm;               // center of mass
```

The Rigid Body also keeps a pointer to the model of the object as this is used for the initial inertia calculation.

Additionally, there is a Collider object which contains the AABB of the shape and the convex hull model of the shape. This object is relatively simple and is updated after the Rigid Body updates to fix the collider positions and update the broad phase.

Broad Phase:

For the broad phase, I am using a dynamic AABB tree. Each object in the world has a fat bounding box that surrounds it. This acts as a node in the dynamic AABB tree. This tree can quickly generate a list of potential collision results in O(n * log(n)) time due to checking the tree against itself and being able to early out when parent nodes don't overlap.

To maximize efficiency, my tree stores nodes inside of a vector as opposed to dynamically allocating when removing/inserting. This means I only need to dynamically allocate whenever my vector grows as I can reuse nodes that are no longer in use after removing objects. Since new objects being removed is not the normal case (only happens at initialization and can happen at runtime but is unlikely) the vector rarely grows.

As this stage of my physics simulation is very fast, it is done on a single thread. I have had no performance hits when testing with 1000+ objects in this stage of my simulation.

Narrow Phase:

Using the potential collision list generated from the broad phase, a GJK/EPA scheme is used to generate contact information. As the contact information is a new structure, this stage is easily parallelized as I can queue a job per collision pair in which GJK and EPA are run. After they run, it will lock the manifold and add the new contact information to the manifold. If GJK does not detect collision, the manifold is not locked and EPA is not run.

For my GJK implementation, I created a simplex object that does all of the heavy lifting of the algorithm. The simplex is initialized with the furthest point in the direction from A's center to B's center. Each iteration, we attempt to reduce the simplex to the Voronoi region closest to the search point. If simplex contains the search point, the objects collide. Otherwise, we add a new point and continue. In general, each iteration of the algorithm is O(n) time and I've found that, on average, it converges on a solution in about 6 iterations.

For my EPA implementation, I created a polytope object that, similar to the simplex, does all of the heavy lifting. This polytope object is created from the resulting simplex of GJK and is iteratively updated with new points and reduced to a convex hull after every added point. Unfortunately, EPA is a significantly slower algorithm and tends towards O(n^2) time in the worst case. It also takes significantly longer to converge, usually averaging around 15 iterations.

With the resulting polytope from EPA, contact information (normal, collision point on each shape, penetration depth) is generated. This is done in constant time as we simply get the closest triangle of the polytope (determined during EPA) and generate info using that.

Collision Resolution:

For collision resolution, sequential impulses is being used. I start by creating the Jacobian from the contact points and collision normal. Then, the effective inverse mass is calculated by applying the Jacobian to each body's inverse mass and inverse inertia tensors. Then, the bias is calculated using a Baumgarte factor and restitution. External Acceleration is computed by applying the Jacobian to the acceleration caused from forces and torques acting on the body this frame. After this, the impulse is warm started using the previous frame's impulse as this leads to better, faster converging results.

To solve the sequential impulses I use an eta value calculated from the external acceleration and the bias. I subtract the current impulse from the eta value and multiply by the effective mass to get the delta impulse. This delta is added to the current impulse and is clamped to [0, inf]. I then subtract the current lambda from the previous lambda to get the delta lambda post clamping. If the delta lambda is non-zero, I apply the delta lambda to the body.

Numerical Integration:

For my rigid body update, I am using a simple forward Euler integrator. Since all of my forces are relatively simple (no spring forces, for example), a more complex integrator is not necessary. Additionally, since this stage of the simulation is easily parallelized via simple batching, I queue a job for each object and the updates are done in parallel.

Additionally, I update my Linear and Angular momentums using the impulses and force/torques that have been applied to my objects. From that, I derive my velocities and update my positions. Since my angular position is stored in a quaternion, I must transform my angular velocity into a quaternion before applying it to the angular position.

References:

[1] Baraff, David. (1997). *An Introduction to Physically Based Modeling: Rigid Body Simulation I—Unconstrained Rigid Body Dynamics*. Carnegie Mellon University.

[2] Coutsias, Evangelos A. and Romero, Louis. (1999). *The Quaternions with an application to Rigid Body Dynamics*. University of New Mexico.

[3] Mandre, Indrek. (2008). *Rigid body drnamics using Euler's equations, Runge-Kutta and quaternions*.

[4] (2018, February 8). *Quaternions*. https://en.wikipedia.org

[5] (2017, September 20). *Exterior Algebra*. Retrieved from https://en.wikipedia.org

[6] Mamou, Khaled. (2011, October 2). *HACD: Hierarchical Approximate Convex Decomposition*. Retrieved from http://kmamou.blogspot.com

[7] (2017, December 8). *Gilbert–Johnson–Keerthi distance algorithm*. https://en.wikipedia.org

[8] (2013, December 31). *Game Physics: Resolution – Contact Contraints*. http://allenchou.net

[9] Catto, Erin. (2005). *Iterative Dynamics with Temporal Coherence*. California, Menlo Park

[10] Bergen, Gino. (unknown). *Proximity Queries and Penetration Depth Computation on 3D Game Objects*.