PHY 350 Process Document

By: Tommy A. Brosman IV
tbrosman@digipen.edu

Fall 2009

Major Revision Summary

December 4, 2009 – Tommy Brosman
Created initial document.

December 8, 2009 – Tommy Brosman
Improved a lot of the information on constraints/etc.

Concept and Overview

High Level Concept

A constraint-based physics engine that has sets of limbs joined by constraints.

Feature List

Constraint-based physics
Contacts, friction
Joints
Interactive picking and dragging

Overview

A set of rigid bodies connected by joints is resting on the ground plane. Any of the bodies can be picked and dragged. All bodies have pre-calculated inertial tensors.

Unimplemented Features

Motors implemented using constraints with time-dependencies

Listed in Technical Design Document. Rheonomic constraints were not added, most of the development time after Code Review II was spent making the contacts more stable and creating a generalized scleronomic constraint class.

Dynamically breakable kinematic chains

Listed in Preliminary Design Document. Dropped mainly due to the current architecture being unable to support non-contact constraints being removed in an intuitive manner.

Stress calculation for joints (for finding when they break)

Listed in Preliminary Design Document. Not really dropped, but the joints do not break. The "stress calculation" is essentially just the magnitude of the resulting Lagrangian multiplier required to correct the relative velocity of the two bodies. This is not actual stress, since the joints themselves are not rigid bodies and are massless.

Extra Features

Mostly stable contact constraints

Although the simulation does not support stacking (and was never intended to), contact constraints were an early step in getting the constraint solver to work. This simplified debugging joints when those were added later.

Multi-point contact and collision

This was added after single-contact constraints were implemented. Single-point collision is very unstable and looks bad, even with a good solver.

Friction constraints

Friction was added to make resting bodies slide around less. Originally a drag was applied to all bodies when integrating, but this caused falling to appear unnaturally slow. Friction constraints were thrown in at the last minute, and have caused very few issues.

Multi-threaded physics/collision

The engine can run in two modes, depending on whether or not USE_THREADS is defined in main.cpp. With USE_THREADS defined, the GameLogic, InputManager, Graphics, and other modules run on the main thread, while the Physics and CollisionManager modules each run on their own thread. The results depend on the hardware. On a Pentium 4 (Prescott architecture), the difference is unnoticeable. On a Pentium D (hyperthreaded) or a Core 2 Duo processor, the speed increase is roughly 10x.

Collision Detection

Broad Phase Using Sweep and Sort

Each rigid body has a bounding sphere. All bounding spheres are projected onto the X axis first, then sorted along that axis. For each object, a bucket of overlapping objects is formed. In the cases where this bucket has only one object in it, it is discarded. For each bucket in the set, the objects in the bucket are projected onto the Z axis (Y is up). The bucket is "pruned" by taking any objects out that are not overlapping the first object. The pruned buckets are then passed to the narrow phase, which generates pairwise collisions.

Axis/Vertex Set Reduction For SAT

Typically, the Seperating Axis Theorem in 3D during detection uses the unique edges out of the set generated from the normals of object A, the normals of object B, and the cross products between all the edges of A and all the edges of B. Uniqueness in this implementation is determined using a structure called a VectorSet. The VectorSet takes the norm of the difference of the inserted vector with each of the existing vectors using an L1 norm and comparing it to some epsilon:

$$|x2-x1| + |y2-y1| + |z2-z1| \leq \epsilon$$

This takes advantage of norm equivalence for Lp norms to avoid using a Euclidean norm and calculating a square root. The second part of the optimization is the epsilon value. Since seperating axis theorem only requires that all axis have overlapping spans, throwing out similar axis poses no risk in terms of failing to detect a collision. The one major drawback is that the collision norm/depth the algorithm chooses may not be the smallest. However, if epsilon is significantly small and the geometry of the object is relatively uniform, this effect will be unnoticeable.

During load time, the geometry of the objects are simplified using the same technique, this time the epsilon being written in terms of the radius of the bounding sphere of the mesh. Since the collision geometry does not require any retriangulation and can be left as a point set, this is relatively simple to implement.

As part of the comparison, any vectors that are significantly close to the zero vector are removed. This avoids complications that arise when taking cross products between parallel vectors.

Contact Formulation

After a collision is detected in the narrow phase, the vertices of the two collision geometries are projected onto the collision normal. The extreme points of each mesh are taken (extreme in the direction of the other mesh) and then projected onto the plane at the position where they should end up after collision resolution. To project the vertices, the collision depth is multiplied by the collision normal and added to each vertex's position. This can cause artifacts when coarse meshes are collided with finer meshes, since there is no actual plane projection and the "projected" vertices may end up nowhere near the actual collision. Eventually this method will be replaced with a face clipping method.

Each contact point is stored as a seperate collision event, along with the time, depth, velocities of the bodies, and any other relevant information. These events can be polled by any module, but are usually used by Physics to formulate the contact constraints used to resolve the collision.

Planned Improvements

In the future, a contact manifold will likely replace the contact point projection. This will reduce the number of constraints needed to keep bodies from interpenetrating, as well as give better contact points. One way to implement this would be to find the convex hull intersection of the projected surfaces, and use that to generate contact points. This is similar to the face clipping scheme Catto mentions, but doesn't describe in detail. Currently stacking objects directly on top of eachother breaks the simulation, as there is no clipping and all vertices are projected directly.

Another improvement that may or may not be implemented is nonlinear projection. Linear projection is done with the assumption that all points travel at roughly the same velocity during a collision. While this doesn't cause many visual anomalies with constraint-based simulations, it can cause excess rotation in impulse-based simulations.

A feature that was originally planned and didn't make it into the final (class) version is contact caching. Currently the constraint solver starts from zero for every undetermined multiplier when solving. With contact caching, the last value would be used, giving better convergence rates for resting bodies. One of the reasons this was not implemented is the fact that the Physics module does not deal with multi-point contact as two bodies with multiple contact points, but treats each contact point as an individual constraint.

In the preprocessing step, the collision mesh is currently generated using the epsilon scheme earlier described. This works well for uniform meshes, but doesn't take the actual geometry into account. A nicer way to generate collision geometry would be to use QuickHull to find the minimal convex hull, with an epsilon tolerance scheme used for collapsing faces with similar norms, much like the one used for removing similar axes.

Constraint Solver

Contact Constraints and Collision Resolution

For each collision, the collision normal and collision depth are used for form a Jacobian pair. Each Jacobian pair is a 12x1 row vector consisting of the non-zero portions of a row in the Jacobian. This is essentially a row in Jsp as described by [1].

```
JacobianRowPair GenerateNormalConstraint(CollisionEvent ev, PhysicsBody A,
PhysicsBody B)
{
      Calculate the Jsp entry for the constraint (see [1]) and return it
}

void Physics::ConstrainedPhysicsUpdate(Time t, Time h)
{
      Array<JacobianPair> Jsp, Bsp
      Array<IndexPair> indices
      Array<float> velocityBias, Lmin, Lmax
      float beta = 0.7/(h*2)
      float slop = 0.01

      // ...

      // formulate contact constraints
      for each CollisionEvent ev in the current collisionEventList
      {
            bodyA, bodyB = PhysicsBody objects that ev refers to
            jspEntry =  GenerateNormalConstraint(ev, bodyA, bodyB)

            if either of the bodies is the ground plane
              set its portion of jspEntry to 0

            Jsp.push_back(jspEntry)
            indexEntry = IndexPair(index of bodyA, index of bodyB)
            indices.push_back(indexEntry)

            // Baumgarte stabilization incorporated into velocity bias
            velocityBias.push_back( beta*(ev.depth - slop) )

            Lmin.push_back(0)
            Lmax.push_back(+infinity)
      }

      // ... evaluate other constraints, evaluate forces ...
      SolveConstraints(Jsp, Bsp, Lmin, Lmax, indices, velocityBias, h)
      // ...
}
```

This version has no contact caching, and has some sliding (which is bad for stacking). 0.7 was found to be a relatively stable factor for Baumgarte stabilization (this is sometimes referred to as the "error reduction parameter", see [2], which also uses a value between 0.1 and 0.8).

Solver Algorithm

The constraint solver itself solves the following equation iteratively each frame [1b]:

$$J M^{-1} J^T \lambda = \frac{1}{\Delta t}\beta(d_{err} - d_{slop}) - J(\frac{1}{\Delta t} V_0 + M^{-1} F_{ext})$$

$$\lambda_{min} \leq \lambda \leq \lambda_{max}$$

The algorithm is the same as [1b], run for 4 iterations per frame. Forward Euler integration is used to turn the working variable "a" (from [1b]) into an impulse and apply it to each body:

```
for(i = 0; i < numBodies; i++)
{
  PhysicsBody body = BodyFromIndex(i)
  body.V += h*a[i]
}
```

Note that each Jpair is broken up into an angular portion and a linear portion [3] as is the velocity on each rigid body.

Joints

Currently, only ball-and-socket joints are implemented. These joints constrain 3 degrees of freedom and have 3 scalar-valued constraint equations [3b]. As a result, the resulting Jsp section is a 3x12 matrix. When creating a base class for storing constraints, a structure was created for storing the results of the evaluation function:

```
class ConstraintResult
{
      Array<JacobianPair> rows
      Array<float> velocityBias
      Array<float> Lmin, Lmax
}
```

Each constraint class is associated with two rigid bodies and takes the following form:

```
class Constraint
{
      ConstraintResult Eval(PhysicsBody A, PhysicsBody B)

      // for looking up the physics bodies
      string nameA
      string nameB
}
```

When constraints are evaluated each frame, the results are added to the Jsp, velocityBias, Lmin, and Lmax arrays the physics engine is currently working with. Note that there is no time-dependency in the Eval() function. This limits constraints created with this base class to scleronomic constraints only.

Planned Improvements

In Catto's paper, he mentions the use of SIMD architecture to accelerate constraint solving. His paper was writtein in 2005, and since then many other more powerful architectures have become available for general purpose calculation. Eventually, the solver (and/or collision detection phase) may be accelerated using an API like OpenCL or CUDA.

An improvement that hasn't been mentioned in many papers is the use of Projected Conjugate Gradient as a "priming" function to help the Projected Gauss Seidel portion converge faster. This is a little-used method, but there is some documentation on it [5]. PCG has a much faster convergence rate than PGS for most systems.

Since the Constraint class is very general, other types of joints will definitely be implemented later. Hinge joints will probably be the most interesting, since they constrain all but one degree of freedom.

As well as adding more joints, my original proposal specified breaking constraints. This is something that didn't make it into the current project, but would be an excellent addition once the contacts are more stable (for example, towers of blocks with breakable rods connecting them).

Currently the joints have no angular limits. It would be a good addition to add limits, allowing for more realistic (and useful) results. One possible way of implementing this would be to correct the joint position using the velocity bias term.

The fact that the Constraint class has no time dependency (or time-step dependency) makes rheonomic constraints impossible for now. Adding support for time-dependent constraints would make motors possible, which could lead to much more interesting simulations (a car with motors turning the wheels, robot arms with servos, etc).

Rigid Body Representation and Equations of Motion

The Physics Update

Since the Physics module runs in parallel with the rest of the application, all game objects containing physics information are used to generate PhysicsBody objects before the update is calculated.

```
void Physics::Update()
{
	get Time t and Time h from the System module
	get all physics bodies and store them internally

	ConstrainedPhysicsUpdate(t, h)

	take all physics bodies and add the change to the game objects
}
```

Rigid Bodies

The PhysicsBody object itself is independent of how the game objects are stored:

```
class PhysicsBody
{
	float mass
	float inertialTensor // in body-space

	Quaternion rot
	Vector trans

	Vector angularVel
	Vector linearVel

	Vector forceAccum
	Vector torqueAccum

	// stuff for delta updates
	Quaternion oldRot
	Vector oldTrans

	Vector oldAngularVel
	Vector oldLinearVel
}
```

The inertial tensor is pre-calculated when the object is created. Currently, the objects are all cuboids (rectangular prisms), spheres, or cones. The diagonalized inertial tensors for all of these are well defined.

Integration

The integration technique used is Forward Euler. Initially velocity verlet was used, but the constraints required evaluation for every integration, which required the constraint engine to be run twice per update. Note that the coriolis torque term is ignored completely.

```
void DampedEulerIntegrate(PhysicsBody body, float h)
```

```
{
	body.linearVel = body.linearVel  + h*body.invMass*body.forceAccum;
	body.angularVel = body.angularVel + h*body.invInertia*body.torqueAccum;

	body.trans	= body.trans	+ h*body.linearVel;
	// angular velocity in worldspace: q' = 0.5*omega*q
	body.rot	= body.rot	+ 0.5f*h*body.angularVel*body.rot;
}
```

Note that the quaternions must be renormalized every frame. This is much faster than other methods of quaternion integration used for non-realtime applications, and produces similar results for small timesteps [4].

Drag Hack

Originally the Forward Euler integration was further modified to add a damping term. This kept things from sliding around (there was no friction). This is roughly equivalent to putting a drag force on everything, or adding friction to all contacts/constraints. The damping factor can be relatively high without visual anomaly. The reason for this is the constraint equation itself:

$$J M^{-1} J^T \lambda = \frac{1}{\Delta t}\beta(d_{err} - d_{slop}) - J(\frac{1}{\Delta t} V_0 + M^{-1} F_{ext})$$

Which carried over the accumulated damped velocity on the right hand side of the equation.

Friction Constraint

The drag hack was removed before the final code review, and friction constraints are now added with each contact. The friction constraint is different from the one described by Catto, but conceptually the same. Instead of two tangent vectors, only one is used, in the direction of the velocity projected onto the contact surface.

$$u = v_{rel} - (\hat{n} \cdot v)\hat{n}$$

$$J_{friction} V = \begin{bmatrix} -\hat{u}^T & -(r_A \times \hat{u})^T & \hat{u}^T & (r_B \times \hat{u})^T \end{bmatrix} \begin{bmatrix} v_A \\ \omega_A \\ v_B \\ \omega_B \end{bmatrix}$$

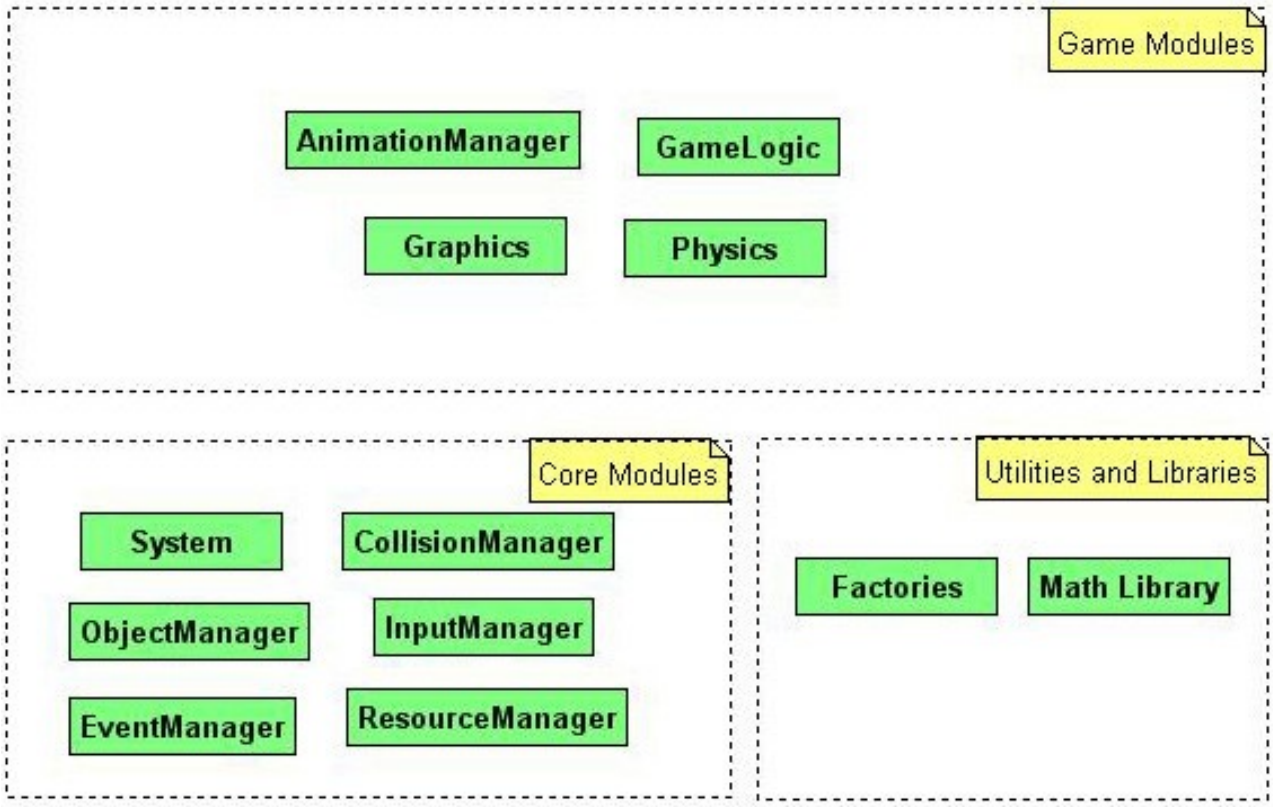
$$m_c = \frac{m_A + m_B}{2}$$

$$-\mu m_c g \leq \lambda \leq \mu m_c g$$

This gives much better results than damping, but since there is one friction constraint for each contact constraint, the actual friction between two surfaces is not accurately modelled. Instead, the simulation behaves simililarily to an infinite friction model, since the amount of friction between two surfaces scales with the number of contact points.

Application Architecture

Overview



Game Modules cannot be directly accessed, with the exception of their Init(), Update(), and Kill() functions. They can directly access Core Modules as well as Utilities and libraries.

Core Modules can be directly accessed, and are highly synchronized. Usually they contain shared data used by other modules.

Utilities and Libraries are considered stateless entities, and require no synchronization.

Modules

Core Modules

System
- Handles any and all interaction with the window manager.
- Has high-resolution timing functions and stores the timestep.
- Has logging/debugging functionality.

InputManager
- Holds keystates, mouse states, and other input information.
- Information in the InputManager is updated by callbacks

EventManager
- Contains all events, and also acts as a timing mechanism for expiring/timed events
- Allows for publishing/polling events synchronously from multiple threads

ResourceManager
- Has functions for generating hardware buffers for models.

CollisionManager
- Performs broad phase collision detection, generating a set of isolated buckets.
- Tests collision between all objects within the buckets using SAT collision.
- Stores any collisions found as CollisionEvents which can be polled by other modules.

ObjectManager
- Holds all GameObjects.
- Holds the CameraObject.
- Can be used to synchronize access to GameObjects using a mutex.

Game Modules

GameLogic
- Handles any camera movement and interaction.

AnimationManager
- Unused for this project.

Graphics
- Renders all models and effects using OpenGL.
- Has debug drawing capabilities which can be used by passing messages.

Physics
- Converts all physical objects from ObjectManager into PhysicsBody objects.
- Generates contact constraints from CollisionEvents polled from the Collision Manager.
- Applies constraints and external forces to the PhysicsBody objects and integrates.
- Applies updated physics information to objects in the ObjectManager.

Gameloop and Concurrency Model

The gameloop runs in parallel on three threads that are created at the beginning of the program. After each thread has finished one iteration of the loop, it waits at a barrier. Once all three threads have hit the barrier, the gameloop continues. This allows for synchronization to be done across all threads without relaunching or over-synchronizing any of the child threads.

Main Thread	**Physics Thread**	**Collision Thread**

`while(running)`
`{`
`  EventManager Update`
`  System Update`
`  InputManager Update`
`  GameLogic Update`
`  AnimationManager Update`
`  Graphics Update`
 	
`  Barrier Wait`
`}`
	`while(running)`
	`{`
	`  Physics Update`
	
	`  Barrier Wait`
	`}`
		`while(running)`
		`{`
		`  CollisionManager Update`
		
		`  Barrier Wait`
		`}`

Bibliography

1. Catto, Erin. "Iterative Dynamics With Temporal Coherence." June 5, 2005. http://www.gphysics.com/files/IterativeDynamics.pdf p.6-7

1b. Ibid, http://www.gphysics.com/files/IterativeDynamics.pdf p.14-17

2. Smith, Russell. "Open Dynamics Engine v0.5 User Guide." February 23, 2006. http://www.ode.org/ode-latest-userguide.html Section 3.7

3. Erleben, Kenny. "Stable, Robust, and Versatile Multibody Dynamics Animation." April 2005. ftp://ftp.diku.dk/diku/image/publications/erleben.050401.pdf. p.67

3b. Ibid, ftp://ftp.diku.dk/diku/image/publications/erleben.050401.pdf. p.71

4. Omelyan, Igor P. "On the numerical integration of motion for rigid polyatomics: The modified quaternion approach." January 18, 1999. http://arxiv.org/abs/physics/9901028v1. p.5

5. Barenbrug, Bart. "Designing a Class Library for Interactive Simulation of Rigid Body Dynamics." 2000. http://users.bart.nl/users/starcat/dynamo/publications/bartbthesis.pdf. p.51