Vehicle Simulation With Bullet

Date: 16/8/2010
Author: Kester Maddock

Introduction

This document records my experience implementing a driving model. It should provide some useful tips for starting a driving simulation, and provide some ideas for future improvements.

Raycast Vehicle

The ray cast vehicle consists works by casting a ray for each wheel. Using the ray's intersection point, we can calculate the suspension length, and hence the suspension force. The suspension force is applied to the chassis, keeping it from hitting the ground. In effect, the vehicle chassis 'floats' along on the rays. The friction force is calculated for each wheel where the ray contacts the ground. This is applied as a sideways and forwards force.

The rays should originate inside the vehicle chassis's btCollisionShape. If the start point of the suspension ray is outside the world, then the ray may never find a ground contact point, and the vehicle will get stuck.

Roll influence effectively lowers the vehicles centre of mass, reducing the chance of the vehicle rolling over.

Parameters

```
/// btWheelInfo is the main struct for defining suspension & wheel
parameters
struct btWheelInfo
{
	/// RaycastInfo contains info for raycasting the wheel.
	/// These are updated by Bullet
	struct RaycastInfo
	{
		/// The normal at the ray contact point (world space)
		btVector3 m_contactNormalWS;
```

```
            /// The point of contact of the ray (world space)
            btVector3 m_contactPointWS;
            /// The current length of the suspension (metres)
            btScalar m_suspensionLength;
            /// The starting point of the raycast (world space) (=
chassisTransform * m_chassisConnectionPointCS)
            btVector3 m_hardPointWS;
            /// The direction of ray cast (in world space) (=
chassisTransform * m_wheelDirectionCS)
            /// The wheel moves relative to the chassis in this
direction, and the suspension force acts along this direction.
            btVector3 m_wheelDirectionWS;
            /// The direction of the wheel's axle (world space) (=
chassisTransform * m_wheelAxleCS)
            /// The wheel rotates around this axis
            btVector3 m_wheelAxleWS;
            /// This is redundant (= m_groundObject != NULL)
            bool m_isInContact;
            /// The other object the wheel is in contact with
(btCollisionObject*)
            void* m_groundObject;
        };

        RaycastInfo m_raycastInfo;

        /// The wheel's world transform
        btTransform m_worldTransform;

        /// The starting point of the ray, where the suspension connects to
the chassis (chassis space) (see also: m_raycastInfo.m_hardPointWS)
        btVector3 m_chassisConnectionPointCS;
        /// The direction of ray cast (chassis space) (see also:
m_raycastInfo.m_wheelDirectionWS)
        btVector3 m_wheelDirectionCS;
        /// The axis the wheel rotates around (chassis space) (see also:
m_raycastInfo.m_wheelAxleWS)
        btVector3 m_wheelAxleCS;
        /// The maximum length of the suspension (metres)
        btScalar m_suspensionRestLength1;
        /// The maximum distance the suspension can be compressed
(centimetres)
        btScalar m_maxSuspensionTravelCm;
        /// The radius of the wheel
        btScalar m_wheelsRadius;
```

```
	/// The stiffness constant for the suspension.  10.0 - Offroad
buggy, 50.0 - Sports car, 200.0 - F1 Car
	btScalar m_suspensionStiffness;
	/// The damping coefficient for when the suspension is compressed.
Set to k * 2.0 * btSqrt(m_suspensionStiffness) so k is proportional to
critical damping.
	/// k = 0.0 undamped & bouncy, k = 1.0 critical damping
	/// 0.1 to 0.3 are good values
	btScalar m_wheelsDampingCompression;
	/// The damping coefficient for when the suspension is expanding.
See the comments for m_wheelsDampingCompression for how to set k.
	/// m_wheelsDampingRelaxation should be slightly larger than
m_wheelsDampingCompression, eg 0.2 to 0.5
	btScalar m_wheelsDampingRelaxation;
	/// The coefficient of friction between the tyre and the ground.
	/// Should be about 0.8 for realistic cars, but can increased for
better handling.
	/// Set large (10000.0) for kart racers
	btScalar m_frictionSlip;
	/// Set the angle of the wheels relative to the vehicle. (radians)
	btScalar m_steering;
	/// The rotation of the wheel around it's axle (output radians.)
	btScalar m_rotation;
	/// The amount of rotation around the wheel's axle this frame.
(output radians)
	btScalar m_deltaRotation;
	/// Reduces the rolling torque applied from the wheels that cause
the vehicle to roll over.
	/// This is a bit of a hack, but it's quite effective. 0.0 = no
roll, 1.0 = physical behaviour.
	/// If m_frictionSlip is too high, you'll need to reduce this to
stop the vehicle rolling over.
	/// You should also try lowering the vehicle's centre of mass
	btScalar m_rollInfluence;
	/// Amount of torque applied to the wheel.
	/// This provides the vehicle's acceleration
	btScalar m_engineForce;
	/// Amount of braking torque applied to the wheel.
	btScalar m_brake;
	/// Set to true if the wheel is a front wheel.
	/// You can use this to select to apply either engine force or
steering.
	bool m_bIsFrontWheel;
	/// A handy place to stash a pointer to your own structures.
	void* m_clientInfo;
```

```
        /// An internal suspension used to modify the suspension forces by
the contact normal
        btScalar m_clippedInvContactDotSuspension;
        /// Output: the velocity of the wheel relative to the chassis.
        btScalar m_suspensionRelativeVelocity;
        /// Output: the force applied to the chassis by the suspension.
        btScalar m_wheelsSuspensionForce;
        /// Output: the amount of grip the wheels have with the driving
surface.
        /// 0.0 = wheels are sliding, 1.0 = wheels have traction.
        /// Use to trigger sliding sounds, dust trails, skid marks etc.
        btScalar m_skidInfo;

};
```

Interpolate Normals

Interpolating the normals from the raycast is an important part of improving the simulation. It tends to smooth out the edges between triangles, and provide a smoother ride. Fortunately, Bullet provides us with enough information to calculate the normals, especially if we have them lying around for the graphics. If you've written a btMeshInterface class to interface with your renderer, this shouldn't be too hard.

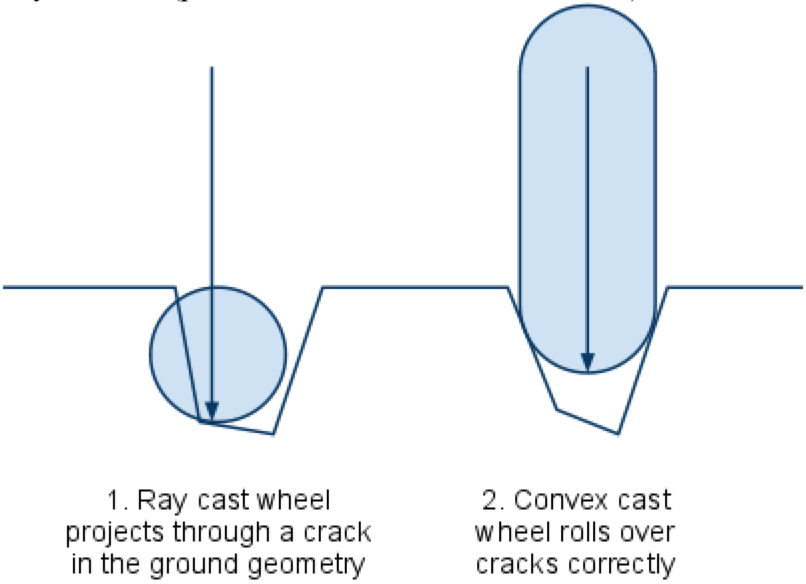
You can compute the Barycentric coordinates of the ray hit point `m_contactPointWS` in a triangle from the triangles' vertex positions. You can then use the Barycentric coordinates to interpolate the normal.

```
/// Compute the Barycentric coordinates of position inside triangle p1, p2,
p3
btVector3 BarycentricCoordinates(const btVector3& position, const
btVector3& p1, const btVector3& p2, const btVector3& p3)
{
                btVector3 edge1 = p2 - p1;
                btVector3 edge2 = p3 - p1;

                // Area of triangle ABC
                btScalar p1p2p3 = edge1.cross(edge2).length2();
                // Area of BCP
                btScalar p2p3p = (p3 - p2).cross(position - p2).length2();
                // Area of CAP
                btScalar p3p1p = edge2.cross(position - p3).length2();
```

```
                btScalar s = btSqrt(p2p3p / p1p2p3);
                btScalar t = btSqrt(p3p1p / p1p2p3);
                btScalar w = 1.0f - s - t;

//#ifdef BUILD_DEBUG
//                  // Unit test...
//                  btVector3 regen_position = s * p1 + t * p2 + w * p3;
//                  btAssert((regen_position - position).length2() <
0.0001f);
//#endif

                return btVector3(s, t, w);
}

// shape, subpart and triangle come from the ray callback.
// transform is the mesh shape's world transform
// position is the world space hit point of the ray
btVector3 InterpolateMeshNormal( const btTransform& transform,
btCollisionShape* shape, int subpart, int triangle, const btVector3&
position )
{
                // Get the geometry from somewhere...
                const btVector3* normals = GetNormals(shape, subpart);
                const btVector3* positions = GetPositions(shape, subpart);
                const unsigned short* indices = GetIndices(shape,
triangle);

                unsigned int i = indices[0], j = indices[1], k =
indices[2];

                btVector3 barry =
BarycentricCoordinates(transform.invXform(position), positions[ i ],
positions[ j ], positions[ k ]);

                // Interpolate from barycentric coordinates
                btVector3 result = barry.x() * normal[i] + barry.y() *
normal[j] + barry.z() * normal[k];

                // Transform back into world space
                result = transform.getBasis() * result;
                result.normalize();
                return result;
}
```

Convexcast Vehicle

Because rays are infinitely thin, it is possible for the wheel to fall through cracks in the geometry. You can either cover these cracks up in the physics geometry, or convex cast the wheels. If you convex cast the wheels, you need to deal with the case where the wheel hits geometry front on. This will compress the springs, and the vehicle will be able to drive over any obstacle (provided the chassis does not hit it.)



Torus Shape

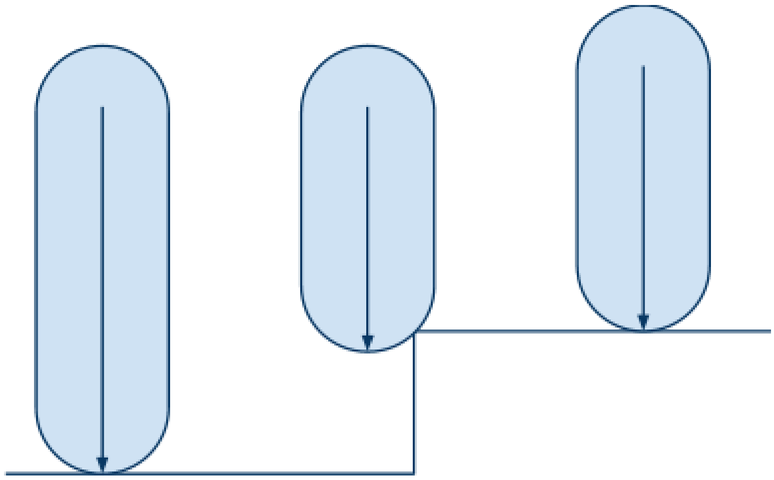
Creating a torus shape should be easy. Convex shapes are all defined using the localGetSupportingVertexWithoutMargin. This returns the point furthest along the given vector. To implement a wheel shape, project the vector onto a circle, and add a large amount of margin.

Convex Cast

The convex cast is pretty easy. Derive from btVehicleRaycaster and override the castRay method to convex cast instead of ray cast. Because of the extra fatness of the ray, you'll need to fix up m_hitPointInWorld, m_distFraction and m_hitNormalInWorld to get correct results.

Problems

The main problem with convex cast wheels is if the wheel hits a curb, or gutter. The wheel will simply reduce the suspension, and the vehicle will be able to drive over most obstacles. In the end, I did not use convex casting, so this problem remains unsolved.



The GJK algorithm for generic convex shapes is quite expensive, and the convex cast sub divides the cast ray and iterates to find the intersection. It can be improved somewhat by implementing the Chung-Wang separating axis test.

```
// The separating axis test returns true if there is an axis that separates
the two objects.
// It checks the vector between their origins, and the preferred directions
of each shape.
bool ChungWangSeparatingAxisTest(const btConvexShape* shape1, const
btConvexShape* shape2, const btTransform& transform1, const btTransform&
transform2, btVector3& cachedSeparatingAxis)
{
	const btMatrix3x3& basis1 = transform1.getBasis();
	const btMatrix3x3& basis2 = transform2.getBasis();

	const btVector3 separatingVector = (transform2.getOrigin() -
transform1.getOrigin());

	btVector3 separatingAxis = cachedSeparatingAxis;
	// The paper suggests an algorithm to calculate when the SAT should
terminate... I'll just
	// do a few iterations and then give up.  The separating axis is
cached between steps, so
	// we can use it & refine it over a few frames.
	int max_iterations = 5;
	while(max_iterations--)
	{
		btVector3 p0 = basis1 * shape1-
>localGetSupportingVertex(separatingAxis * basis1);
		btVector3 p1 = basis2 * shape2-
>localGetSupportingVertex(-separatingAxis * basis2);
```

```
            btScalar radius1 = p0.dot(separatingAxis) -
p1.dot(separatingAxis);

            if (radius1 + SIMD_EPSILON <
separatingVector.dot(separatingAxis))
            {
                  cachedSeparatingAxis = separatingAxis;
                  return true;
            }

            btVector3 r = (p1 - p0).normalized();
            separatingAxis = separatingAxis + 2.0f * r.dot(separatingAxis)
* r;
      }

      cachedSeparatingAxis = separatingAxis;
      return false;
}
```

Suspension

Suspension is provided by the spring force plus a damping force. The damping force stops the car from bouncing forever. There are two coefficients for damping: one for spring compression, and one for spring relaxation. In a real vehicle, the compression damping is set much lower than the relaxation damping. This means, when the vehicle hits a bump, it won't be transmitted to the chassis, resulting in a smooth ride.
Set the suspension damping as a fraction of critical damping: = k * 2.0 * btSqrt(m_suspensionStiffness), where k is the proportion of critical damping. Now you can tweak k to control the bounciness of the suspension. k = 0.0 is very bouncy, k = 1.0 is no bounce, and k > 1.0 is over damped. Values around 0.5 work quite well. For more information, see http://en.wikipedia.org/wiki/Damping

Why are my wheels sinking through the ground?

The wheels sink through the ground when the suspension cannot support the weight of the vehicle. You need to increase the suspension stiffness, max travel or suspension length. Increasing the suspension too much will make the simulation unstable. The max travel is the maximum amount the springs can be compressed: the suspension will provide the maximum force at the point.

Centre of Mass

Lowering the centre of mass improves handling. In a real car, most of the mass is in the chassis & engine, at the bottom of the car. Lowering the centre of mass is a bit tricky in Bullet. You need to create a collision shape class that can handle a transform, and a motion state that undoes the transform. Note that the CCD motion clamping assumes no centre of mass transform, so you need to disable it.

Friction Model

The friction model in Bullet is implemented as separate impulses applied to each wheel. This means that it's possible for a single wheel to counteract all horizontal motion of the chassis, and for multiple wheels to add additional velocity, resulting in oscillation or jitter of the vehicle. The solution to this is to create a friction constraint model. Expressing the friction as constraints allows Bullets constraint solver to perfectly balance the friction on each wheel, to counteract any sideways velocity.

A friction constraint is on of the simplest constraints you can implement. You provide an axis to act along, and set the target velocity to 0.0. You set the maximum impulse according to Coulomb's friction law.

$$F = \mu N$$

F = maximum friction force
mu = friction coefficient `m_frictionSlip`
N = normal force

To create a constraint, you derive from btTypedConstraint and implement the interface getInfo1 and getInfo2:

```
void WheelFrictionConstraint::getInfo1(btConstraintInfo1* info)
{
	// Add two constraint rows for each wheel on the ground
	info->m_numConstraintRows = 0;
	for (int i = 0; i < vehicle->getNumWheels(); ++i)
	{
		const btWheelInfo& wheel_info = vehicle->getWheelInfo(i);
		info->m_numConstraintRows += 2 * (wheel_info.m_groundObject !=
NULL);
	}
}

void WheelFrictionConstraint::getInfo2(btConstraintInfo2* info)
{
	const btRigidBody* chassis = vehicle->getChassis();

	int row = 0;
	// Setup sideways friction.
	for (int i = 0; i < vehicle->getNumWheels(); ++i)
```

```
    {
        const btWheelInfo& wheel_info = vehicle->getWheelInfo(i);

        // Only if the wheel is on the ground:
        if (!wheel_info.m_groundObject)
            continue;

        int row_index = row++ * info->rowskip;

        // Set axis to be the direction of motion:
        const btVector3& axis = wheel_info.m_raycastInfo.m_wheelAxleWS;
        info->m_J1linearAxis[row_index] = axis;

        // Set angular axis.
        btVector3 rel_pos = wheel_info.m_raycastInfo.m_contactPointWS -
chassis->getCentreOfMassPosition();
        info->m_J1angularAxis[row_index] = rel_pos.cross(axis);

        // Set constraint error (target relative velocity = 0.0)
        info->m_constraintError[row_index] = 0.0f;

        info->m_cfm[row_index] = WHEEL_FRICTION_CFM; // Set constraint
force mixing

        // Set maximum friction force according to Coulomb's law
         // Substitute Pacejka here
        btScalar max_friction = wheel_info.m_suspensionForce *
wheel_info.m_frictionSlip / info->fps;
        // Set friction limits.
        info->m_lowerLimit[row_index] = -max_friction
        info->m_upperLimit[row_index] =  max_friction
    }

    // Setup forward friction.
    for (int i = 0; i < vehicle->getNumWheels(); ++i)
    {
        const btWheelInfo& wheel_info = vehicle->getWheelInfo(i);

        // Only if the wheel is on the ground:
        if (!wheel_info.m_groundObject)
            continue;

        int row_index = row++ * info->rowskip;

        // Set axis to be the direction of motion:
        btVector3 axis =
wheel_info.m_raycastInfo.m_wheelAxleWS.cross(wheel_info.m_raycastInfo.m_wheelDirectionW
        info->m_J1linearAxis[row_index] = axis;
```

```
        // Set angular axis.
        btVector3 rel_pos = wheel_info.m_raycastInfo.m_contactPointWS -
chassis->getCentreOfMassPosition();
        info->m_J1angularAxis[row_index] = rel_pos.cross(axis);

        // FIXME: Calculate the speed of the contact point on the wheel
spinning.
        //        Estimate the wheel's angular velocity = m_deltaRotation
        btScalar wheel_velocity = wheel_info.m_deltaRotation *
wheel_info.m_wheelsRadius;
        // Set constraint error (target relative velocity = 0.0)
        info->m_constraintError[row_index] = wheel_velocity;

        info->m_cfm[row_index] = WHEEL_FRICTION_CFM; // Set constraint
force mixing

        // Set maximum friction force
        btScalar max_friction = wheel_info.m_suspensionForce *
wheel_info.m_frictionSlip / info->fps;
        // Set friction limits.
        info->m_lowerLimit[row_index] = -max_friction
        info->m_upperLimit[row_index] =  max_friction
    }

}
```

This constraint acts in a pyramid friction model. A conical friction model is a bit harder to implement.

Problems

The pyramid friction model isn't quite correct, and gives too much friction at the corners. However, a conical friction model (J1linearAxis = relative_velocity.normalized()) suffered from numerical instability.

Calculating the wheel velocity is also quite tricky; it should be added as a rigid body and solved by the constraint solver. This was solved by calculating the forward friction force outside of the constraint solver, and only using the constraint solver for sideways constraints.

Suspension Constraints

Bullet's suspension has one major drawback: when the spring is fully compressed, it cannot provide enough force to keep the vehicle's chassis off the ground. In a real vehicle, the spring will hit a bumper, keeping the wheel away from the chassis. To simulate this, we use a constraint.

A suspension constraint has two parts: the suspension limits, and the suspension force. The suspension force is responsible for the spring force applied by suspension, and the suspension limits stops the wheels penetrating the chassis, or flying off the vehicle. Although the suspension spring forces could be applied outside the constraint system, it is convenient to implement them as a constraint row, since we need the force applied by the suspension to calculate the wheel friction forces.

Suspension Constraints + Rigid Body Wheels

Adding wheels as rigid bodies adds inertia to the suspension system, improving it's realism. It also lowers the centre of mass of the vehicle, improving the handling.

To implement this, we create a btRigidBody per wheel. Then we create pin constraints between the wheel & chassis rigid bodies along the axle and wheel forward vector to keep the wheels in the correct position & orientation. We also need a constraint for suspension force and suspension limits. Lastly, we create wheel friction constraints along the axle and wheel forward vector, and a contact constraint to keep the wheel from penetrating the ground.

Now, we apply engine power as torque to the wheels to get the car moving, and increase the angular damping to provide braking. The wheels should spin out properly under high torque.

The major problem with the rigid body wheel setup, is the constraint stiffness. The wheel pin constraint & orientation pin constraint need to be infinitely stiff to prevent wheel wobble. High angular velocity can also become a problem.

I haven't implemented the rigid body wheels system completely yet, even though it should provide an increase in simulation quality. Featherstone's articulated body method should be used instead of the sequential impulse constraint solver.

Pacejka Tyre Friction

Pacejka's Magic formula is an empirical formula for modeling tyre friction. It calculates the maximum friction force a tyre can provide at a given slip angle and load. I haven't implemented this properly, it always seems to underestimate the amount of friction required to stop the car sliding sideways.

Calculating the Slip Angle

Pacejka uses the slip angle to calculate the maximum friction force.

```
const btRigidBody* const chassis = vehicle->getRigidBody();
```

```
        const btWheelInfo& wheel_info = vehicle->getWheelInfo(i);
        // Calculate velocity of wheel wrt ground
        btVector3 rel_pos1 = wheel_info.m_raycastInfo.m_contactPointWS -
chassis->getCenterOfMassPosition();
        btVector3 vel = chassis->getVelocityInLocalPoint(rel_pos1);

        // Calculate velocity in x & y
        btScalar vz = -vehicle->getForward(i).dot(vel);
        btScalar vx = vehicle->getAxle(i).dot(vel);

        // Calculate slip angle of wheel.
        btScalar slip_angle = btAtan2(vx, btFabs(vz));
```

Counteracting a slide

To counteract a slide, you turn your steering wheels in the direction of the slide.

Calculate the slip angle of all wheels (see above) and compute the weighted average, weighted by slip. Then, add this onto wheel_info.m_steering. That way, the vehicle goes into a controlled slide, and the driver can adjust the amount of slide. (I'm pretty sure Motorstorm does this - you can see the front wheels of the car auto adjusting in the replays.)

Handbrake Turn

Pulling the handbrake locks up the rear wheels, causing a large friction force. Implement a friction constraint (set target relative velocity = 0.0 in the direction of motion) when the wheels are locked up.

Performance

Simulating vehicles can be quite taxing on a physics system. This is because vehicle simulation relies on the ray casting features, whereas physics engines are usually optimised for box-stacking benchmarks. Therefore, most engine performance work goes into optimising the constraint solver. This also works to our advantage, since the more advanced simulation models make use of the constraint solver. It's still likely you will want to target the ray casting for optimisation.

Some things to try are:
- Batch raycasting. Create a ray cast handle, and cast all rays at once, multithreaded
- Multi-raycasting. All the vehicle rays are coherent, so cast them all at once

· Simulation LODs. You can switch from a forces model to constraint model to constraint + rigid body model depending on the distance to the player.