Physics Tutorial 6: Collision Response

Summary

Collision response, the final step of our physics engine, is introduced. Solutions are discussed and implemented.

New Concepts

Collision response, Penalty method, Impulse method, Projection method, Impulses, contact constraint, constraint-based collision resolution, constraint drift, Baumgarte offset.

Introduction

Our physics engine is almost complete. So far we can move objects around the environment in a believable manner, and we can detect when two objects intersect. In this tutorial we discuss how to proceed when we have identified that two objects have collided in order to move them apart. As you might expect, the solution is to give each intersecting object a nudge in the direction away from the collision.

In the previous tutorials we have discussed how to identify when an intersection has occurred, and how to calculate the data required to resolve an intersection. The data which we have calculated consists of:

- The contact manifold - the point (or points) of the objects which were in contact at the instant of collision.
- The contact normal - i.e. the direction vector from the object's position to the point of collision.
- The penetration distance - i.e. the depth to which two objects have interfaced.

Remember that, at this point, the physics engine is still dealing with all the simulated objects - the new physical state of the simulated objects is not made available to the render loop until after the physics update is complete, so the player will not see any of the intersections between objects, so long as they are successfully resolved.

The question which is addressed in the next pair of tutorials is: how do we use this collision data to move the intersecting objects apart?

A simple solution would be to simply move the objects along the collision normal by a distance equal to the penetration depth, by directly changing the position vectors. This is known as the *projection method*, and while it will suffice for a simple simulation, it has some fairly obvious drawbacks related to the objects' velocities. If the objects are moved apart without changing their velocities then they will just continue along the same path during the next physics update and are likely to intersect again; alternatively if the objects are moved apart and the velocity then set to zero then the simulation feels very unrealistic, as objects tend to bounce off one another rather than stop dead on first contact. We clearly need a solution which affects the velocities and/or accelerations of the objects rather than directly affecting the positions.

Algorithms which directly affect the velocities of the intersecting objects are known as *Impulse Methods*, whereas algorithms which directly affect the acceleration of the bodies are known as *Penalty Methods*. Penalty methods use spring forces to, in effect, pull the objects away from each other by affecting the acceleration through Newton's second law ($F = ma$). Impulse methods use instantaneous nudges, or impulses, to push the objects apart by directly controlling their velocities.

In summary:

- **Projection methods** - control the position of the intersecting objects directly.
- **Impulse methods** - control the velocity of the objects, i.e. the first derivative.
- **Penalty methods** - control the acceleration of the objects, i.e. the second derivative.

In this tutorial, we consider the theoretical aspects of the impulse and penalty methods (the underpinnings of projection method being trivial). We discuss the nature of impulse and the manner in which objects guided by conservation of momentum interact. We then go on to discuss springs and their role in the penalty method, and include code samples illustrating the manner in which a spring might be introduced as a new form of constraint.

Impulses

The impulse method allows us to directly affect the velocities of the simulated objects which have intersected. This is achieved through the application of an impulse, which can be thought of as an immediate transfer of momentum between the two bodies. Impulse is a term defined by classical physics as the accumulated force applied to a body over a specific amount of time (it is therefore measured in Newton seconds Ns). The impulse J is defined in terms of force F and time period Δt as

$$J = F\Delta t$$

We know from Newton's second law, that $F = ma$, and we can also write the acceleration a as the rate of change of velocity v. Substituting these values into the equation for impulse gives us:

$$J = F\Delta t = ma\Delta t = m\frac{\Delta v}{\Delta t}\Delta t = m\Delta v$$

We also know that momentum is equal to the product of mass and velocity, so an impulse is equivalent to the change in momentum.

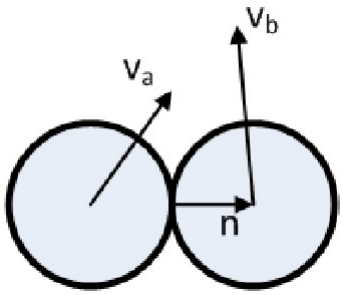
Our plan then is to give colliding objects a nudge, by changing their velocity by an amount equal to

$$\Delta v = \frac{J}{m}$$

The question then, is how to calculate the impulse J generated when two bodies collide.

Calculation of Linear Impulse

First, we will discuss what we would like to happen to the velocities of two colliding objects - namely, we want the bodies to bounce off one another. We will consider the simple case of two spheres colliding, as shown in the figure – sphere a is moving with velocity v_a, while sphere b has velocity v_b; the collision normal is n. We want to calculate the impulse J.



The impulse is generated by the velocity at which the two spheres have collided so we are interested in the relative velocity of the two objects, which we will label v_{ab}. The component of the relative velocity which caused the collision is along the normal vector, so we calculate the dot product of the relative velocity and the normal:

$$v_{ab} = v_a - v_b$$

$$v_n = v_{ab} \cdot n$$

The velocity along the normal after collision is dependent on the *coefficient of elasticity* ε. A coefficient of 1 means the collision will be purely elastic, so all the velocity is transferred, whereas a coefficient of zero is purely non-elastic, so no velocity is transferred. A purely non-elastic collision will result in the two bodies staying together (i.e. no bounce); a purely elastic collision is a perfect bounce so no damping or slowing down occurs. Your simulation is likely to require a figure somewhere in between. Quite often different object types in a game will have different coefficients of elasticity which are stored as a member of the object class, in the same way as the mass is.

The coefficient of elasticity is the factor by which the velocity before the collision is multiplied to calculate the velocity after the collision. Hence:

$$v_n^+ = -\varepsilon v_n^-$$

or, substituting for v_n:

$$(v_a^+ - v_b^+) \cdot n = -\varepsilon(v_a^- - v_b^-) \cdot n$$

Note the introduction of $^-$ and $^+$ nomenclature to denote the state of the bodies before and after the collision respectively. Also note the negation of the velocity - remember we want to push the two bodies back apart in the opposite direction to their colliding velocity.

We also need to think about the momentum of the two bodies. You will remember from the first tutorial in this series on Newtonian mechanics that the total momentum must remain constant in any collision. However our plan to resolve the collision is to "inject" some momentum into the system. Hence we need to ensure that the overall additional momentum is equal to zero, which is achieved by making the momentum used to nudge the second body, the exact opposite of that used to nudge the first. This is shown in the equations below, showing the relationship between momentum before and after the collision for each body, where J is the injected impulse along the normal vector n.

$$m_a v_a^+ = m_a v_a^- + Jn$$

$$m_b v_b^+ = m_b v_b^- - Jn$$

Note that it is the injected *momentum* which is equal and opposite, not the *velocity* as that will be affected by the mass of the object and therefore unequal for differently sized objects. Combining the

last three equations, allows us to solve for the impulse J

$$J = \frac{-(1+\varepsilon)v_{ab} \cdot n}{n \cdot n(\frac{1}{m_a} + \frac{1}{m_b})}$$

which in turn allows us to calculate the velocities of the two bodies after the collision:

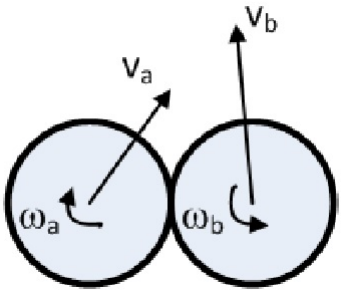
$$v_a^+ = v_a^- + \frac{J}{m_a}n$$

$$v_b^+ = v_b^- - \frac{J}{m_b}n$$

So we can now calculate the velocity at which two colliding bodies should move away from one another in a manner which feels believable, as it is based on Newtonian mechanics, and which incorporates a "bounciness" factor for different types of object in the form of the coefficient of elasticity. Remember, as ever, that the velocities we are discussing are three dimensional vectors – the bodies can move along all three axes of the simulated world. They can also *rotate* around all three axes, so next we need to consider how to account for rotational elements in collision response.

Calculation of Angular Impulse

Let's add some spin to our two colliding spheres. The angular velocity of each is ω_a, ω_b and the radius is r_a, r_b respectively



In order to realistically add angular motion to a collision response, we need some more information about the actual contact point between the two objects – specifically the velocity of that point on each of the objects. This is the sum of the object's linear velocity, and the extra tangential velocity v_t created by the fact that the point is rotating around the object's centre. The velocity at a point which is distance r from the centre on an object turning with angular velocity ω is

$$v_t = \omega r$$

Remember that rotational velocity is measured in radians, and there are 2π radians in a full revolution; similarly the distance around that revolution is $2\pi r$.

So the velocity of the contact point v_C on each object is:

$$v_{Ca} = v_a + \omega_a r_a$$

$$v_{Cb} = v_b + \omega_b r_b$$

Again we need to ensure that angular momentum is conserved, so:

$$I_a\omega_a^+ = I_a\omega_a^- + r_a \times Jn$$

$$I_b\omega_b^+ = I_b\omega_b^- - r_b \times Jn$$

Remember that the angular momentum of a body is the product of the angular velocity ω and the inertia tensor I. J is the impulse which we are calculating and n is the collision normal. Solving

our equations for J, taking into account the angular momentum, leads to a much more complicated looking calculation for the impulse:

$$J = \frac{-(1+\varepsilon)v_{ab} \cdot n}{n \cdot n(\frac{1}{m_a} + \frac{1}{m_b}) + [(I_a^{-1}(r_a \times n)) \times r_a + (I_b^{-1}(r_b \times n)) \times r_b] \cdot n}$$

which again allows us to calculate the velocities of the two bodies after the collision, as well as the angular velocities:

$$v_a^+ = v_a^- + \frac{J}{m_a}n$$

$$v_b^+ = v_b^- - \frac{J}{m_b}n$$

$$\omega_a^+ = \omega_a^- + \frac{r_a \times Jn}{I_a}$$

$$\omega_b^+ = \omega_b^- + \frac{r_b \times Jn}{I_b}$$

These equations allow us to write an algorithm in C++ to believably simulate two objects colliding and bouncing off one another using Newtonian mechanics. Both linear and angular movement are accounted for, and the elasticity of the collision is also incorporated.

Springs

In the real world a spring tends to be a coiled up strand of metal. We are interested in simulating the properties of a spring when it is used to connect two objects; these properties can be summarised as

- When the spring is compressed it forces the two objects apart.
- When the spring is elongated it forces the two objects toward one another.

Basically the spring continually tries to return to its rest length. This is expressed mathematically as:

$$F = -kx$$

where k is the spring constant (a measure of how difficult the spring is to stretch), x is the difference between the rest length of the spring and its current length, and F is the ensuing force at each end of the spring. The equation is known as *Hooke's Law of Elasticity*, which you probably studied during Physics classes in school. Note the negative sign, which shows that this is a restorative force – i.e. it is trying to pull the spring back to its equilibrium length. A higher value of the spring constant k means the spring requires a higher force to stretch or compress it; more to the point for our purposes, a higher value of k will result in a greater force pushing our intersecting objects apart. As ever, for the purposes of our physics engine, the distance and force in the equation are three dimensional vectors, but we will discuss them in simpler terms for clarity.

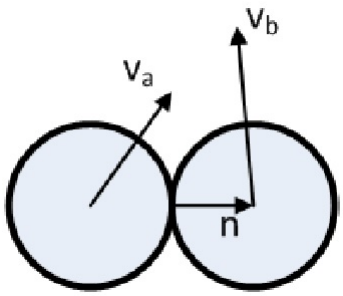
If this were the sum total of our spring simulation, then the objects which are connected by the spring would continue to bounce back and forth for ever. This is of course unrealistic and not the intention of the simulation. In the real world, an object on the end of a spring will gradually slow down and come to a rest. Hence we need to introduce a *damping factor* into our algorithm.

The damping factor is higher at larger velocities of the object, and a proportional force is introduced via the *damping coefficient* c. The equation for our spring force now becomes

$$F = -kx - cv$$

where v is the velocity of the object along the axis of the spring.

So we have a simple equation which describes how a spring can force two objects apart, or draw them together, while taking into account a realistic damping factor to bring the objects to rest.



Springs for Collision Response

We will again consider the simple case of two colliding spheres. Our intention here is to address any intersection identified in the collision response routines by attaching the intersecting objects to either end of a virtual spring, and then rely on the resulting spring forces to push the objects back apart.

We again have the three pieces of intersection information from the collision detection routines; namely the contact point, the contact normal n and the penetration depth d.

The relative velocity is again simple to calculate ($v_{ab} = v_a - v_b$), and we are interested in the component of the relative velocity in the direction of the collision normal (i.e. the dot product of v_{ab} and n. The coefficient of elasticity is k and the damping factor is c, giving the equation:

$$F = -kd - c(n \cdot v_{ab})$$

An equal and opposite force is applied to each object. However, often one of the objects attached to the spring is actually a fixed point (for example, part of the environment), in which case the force on that end of the spring is ignored, and only the free object is affected.

Remember that this method results in a force, which is then applied in the same way as other game forces – i.e. the resulting acceleration is calculated from $F = ma$. The impulse method resulted in a direct change to velocity, so the mass of the objects was taken into account in calculating the impulse; with the penalty method the mass is taken into account when calculating the acceleration from the force.

The choice of values for the two coefficients k and c is vital, and can result in a range of desired collision response types. The higher the value of the damping factor c, the less bouncy the collision, while the higher the value of the elasticity coefficient, the more solid the objects will feel. For example, a low value of k and a low value of c will feel like a trampoline (a big bounce on a soft surface), while a low value of k with a high value of c will feel more like a swamp (sinking into a soft surface). You will need to experiment with values to tune the effect that you want.

Implementation of Springs for Collision Response

You can consider the concept of the spring as a new form of constraint. See if you can implement a spring constraint in C++. As our entire system is already constraint-based in its update loop, you should be able to neatly interface this with your existing framework to explore the behaviours this type of constraint between objects.

The penalty method for collision response in the physics simulation is more straightforward to implement than the Impulse method, and it has the advantage of directly utilising the force-based movement implementation. However great care must be taken so that the results don't feel as though there are actual springs connecting things together; without that care objects may bounce around in an unrealistic manner.

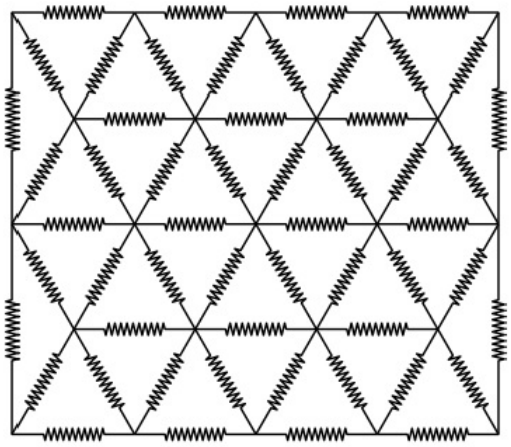
Soft Bodies

So far we have simulated each object as a single item in the simulation (either a particle or a rigid body). We may have extended some objects into hierarchical skeletons of items, or rigid, conjoined

items with a shared inverse inertia matrix (as is the case if we decompose concave objects into multiple convex objects). Importantly, in both cases, these objects are treated by the physics engine as distinct entities - if they weren't, there'd be no point in us doing it in the first place *up to now*.

The approach we've employed thus far is perfect for most entities which are to be simulated in a game, as we do not anticipate that many objects will be required to change shape as the simulation progresses. Remember that a rigid body can change it's position and orientation, but not its shape. In the cases when we do require a body to change its shape, we need a new approach known as soft body simulation – springs can be utilised to provide soft-bodies in the physics simulation.

A soft body can be modelled as a network of nodes interconnected by simulated springs. The diagram shows a piece of cloth simulated in this way.



The ideal rest position of the cloth has no tension in the springs, so the mesh is evenly spaced. Applying a force to one part of the mesh will cause the surrounding springs to stretch, and to pull the connected nodes which in turn stretch the next layer of springs attached to them. As the force is removed the springs contract and bring the overall mesh back to its rest position. A three dimensional deformable object (such as a jelly) can be simulated similarly, with a three dimensional array of nodes. It is also common to use a one-dimensional set of noes to simulate a rope, a chain or even a strand of hair.

A common approach in simulating deformable objects using this spring-based soft body solution is to match the graphical model to the simulated model. That is to say that each vertex of the graphical model is represented by a node in the physical model, and each edge in the graphical model coincides with a spring.

It should be stressed that soft bodies are extremely expensive items to simulate both in terms of computation and memory. Each spring connecting each node must be simulated, which is obviously considerably more computationally expensive than treating the object as a single entity as happens in rigid body simulation. Consider that two rigid balls bouncing off one another only employ a single constraint between them - while two deformable balls, each defined by a hundred nodes in a spring-constraint mesh, might have hundreds of constraints to be resolved when they impact one another.

Similarly, as the mesh of springs and nodes must be stored in a data structure, there is also a greater cost to memory. For these reasons, soft body simulation tends to be used for specific instances in games, which give maximum impact to the player. It is also worth pointing out that this approach is not really based on any kind of real-world physics; it is a technique for providing some behaviour which looks and feels good in the game simulation (i.e. in reality a cloth isn't a network of connected springs).

Deformation of Semi-Rigid Bodies

There are instances where a body needs to deform, but not return to its original shape (for example a crumple zone on a car) – in those instances the spring approach is not suitable as the basic nature of a spring is that it is constantly trying to return to its rest length. In the instance of the car crumple zone, the bodywork is still modelled as a network of interconnected nodes, and the simulation knows that they have freedom of movement along a particular axis in one direction only. Consequently applying a sufficiently high force will cause the nodes to squash up together, but removing the force does not allow the nodes to revert to the uncompressed state.

Constraint-Based Collision Response

Our Framework is built upon constraint-based solutions to the mathematical concepts outlined in the first part of this tutorial. The method by which we relate the two, put simply, is to generate a distance constraint between two colliding objects that prevents any overlap between them. This is then coupled with the addition of an elasticity force to generate an appropriate reaction to the collision within our simulation.

Inequality Constraints

When constructing constraints we sometimes need a constraint which only acts in certain regions of space or in certain directions. For example the constraints we will be dealing with next, which allow use to simulate contact points, only act when two objects are colliding. Once the two objects are separated again the constraint should not act upon the two objects.

We can achieve effects like this by applying an inequality restriction on λ like so:

$$\lambda_- \leq \lambda \leq \lambda_+$$

An unrestricted constraint would have a range equivalent to $(\lambda_-, \lambda_+) = (-\infty, \infty)$. A constraint like the contact constraint which will be discussed shortly would have a range of the form $(\lambda_-, \lambda_+) = (0, \infty)$.

Contact Constraint

We define the constraint which will be employed to model collision as:

$$\mathrm{C} = (\mathbf{x}_2 + \mathbf{r}_2 - \mathbf{x}_1 - \mathbf{r}_1) \cdot \mathbf{n}$$

Where $\mathbf{x}_1$ and $\mathbf{x}_2$ are the centres of masses for the two object and $\mathbf{r}_1$ and $\mathbf{r}_2$ are the relative positions of the points of contact from the centres of the two objects. Figure 1 illustrates the set-up for this constraint, for objects 1 and 2, with (v_1, ω_1) and (v_2, ω_2) respectively, colliding at the blue point.

The normal $\mathbf{n}_1$ is considered to point away from object 1 and towards object 2. Given this constraint equation the quantity is negative when the two bodies penetrate and positive when they are separate.

Before we calculate the Jacobian consider the quantity $\mathbf{p} = \mathbf{x} + \mathbf{r}$ which represents the point of contact. If we differentiate this vector with respect to time we get the result:

$$\frac{\mathrm{d}\mathbf{p}}{\mathrm{d}t} = \mathbf{v} + \omega \times \mathbf{r}$$

Using this result we can differentiate the constraint to obtain $\dot{C}$ of the form:

$$\dot{\mathrm{C}} = (\mathbf{v}_2 + \omega_2 \times \mathbf{r}_2 - \mathbf{v}_1 - \omega_1 \times \mathbf{r}_1) \cdot \mathbf{n}_1 + (\mathbf{x}_2 + \mathbf{r}_2 - \mathbf{x}_1 - \mathbf{r}_1) \cdot \omega_1 \times \mathbf{n}_1$$

The second term in this expression is dependent on the penetration $(\mathbf{x}_2 + \mathbf{r}_2 - \mathbf{x}_1 - \mathbf{r}_1)$. We consider the penetration of this collision to be a small value - which it ought to be if our time steps have been chosen appropriately to the scale of our environment, and vice versa. As such, the second term can be ignored leaving us with the simplified expression:

$$\dot{\mathrm{C}} \approx (\mathbf{v}_2 + \omega_2 \times \mathbf{r}_2 - \mathbf{v}_1 - \omega_1 \times \mathbf{r}_1) \cdot \mathbf{n}_1$$

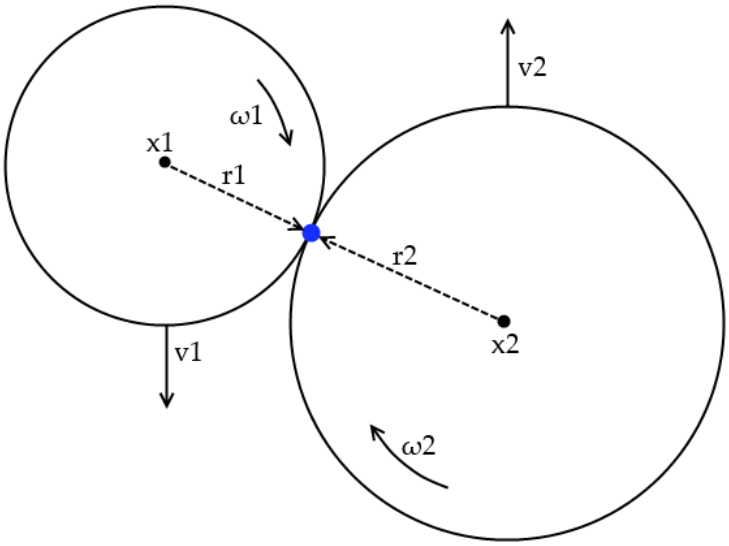


Figure 1: Illustration of the contact constraint.

We can now use the vector identity $A \cdot B \times C = C \cdot A \times B$ (which you'll recall from Tutorial 4 in this series) to re-arrange this expression and extract the Jacobian:

$$\dot{\mathrm{C}} \approx (-\mathbf{n}) \cdot \mathbf{v}_1 + (-(\mathbf{r}_1 \times \mathbf{n})) \cdot \omega_1 + \mathbf{n} \cdot \mathbf{v}_2 + (\mathbf{r}_2 \times \mathbf{n}) \cdot \omega_2$$

This gives us a Jacobian of the form:

$$\mathbf{J} = \begin{bmatrix} -\mathbf{n}^T \\ -(\mathbf{r}_1 \times \mathbf{n})^T \\ \mathbf{n}^T \\ (\mathbf{r}_2 \times \mathbf{n})^T \end{bmatrix}$$

You may well recognise the form of this from our earlier consideration of the distance constraint as worked through in a previous tutorial, addressing the linear and angular properties of each object in turn, with signs dependent on the direction of the contact normal (in this case, object 1 to object 2).

Since the purpose was to push the two objects apart we restrict λ to the range:

$$0 \leq \lambda < \infty$$

Friction

We can extend the approach in the previous section in order to implement friction. We construct two perpendicular unit vectors $\mathbf{u}_1$ and $\mathbf{u}_2$ which when multiplied by the cross product give us the normal $\mathbf{n}$:

$$\mathbf{n} = \mathbf{u}_1 \times \mathbf{u}_2$$

The vectors $\mathbf{u}_1$ and $\mathbf{u}_2$ represent two tangential vectors along the surface of the objects which are in contact with each other. We then apply the constraint that was constructed in the previous section too each of $\mathbf{u}_1$ and $\mathbf{u}_2$ giving two more constraints:

$$\mathrm{C}_1 = (\mathbf{x}_2 + \mathbf{r}_2 - \mathbf{x}_1 - \mathbf{r}_1) \cdot \mathbf{u}_1$$
$$\mathrm{C}_2 = (\mathbf{x}_2 + \mathbf{r}_2 - \mathbf{x}_1 - \mathbf{r}_1) \cdot \mathbf{u}_2$$

These two new constraints taken together allow us to restrict the movement of the contact point along the surface in any direction along the surface. As the equation is the same as the previous constraint we can calculate the Jacobians of the two constraints as:

$$\mathbf{J}_1 = \begin{pmatrix} -\mathbf{u}_1^T & -(\mathbf{r}_1 \times \mathbf{u}_1)^T & \mathbf{u}_1^T & (\mathbf{r}_2 \times \mathbf{u}_1)^T \end{pmatrix}$$
$$\mathbf{J}_2 = \begin{pmatrix} -\mathbf{u}_2^T & -(\mathbf{r}_1 \times \mathbf{u}_2)^T & \mathbf{u}_2^T & (\mathbf{r}_2 \times \mathbf{u}_2)^T \end{pmatrix}$$

However since we are not using these constraints to resolve the penetration therefore the restriction on the coefficients proportionality λ_1 and λ_2 relate to the maximum force friction can apply before the objects begin to slip past one another.

In realistic models of friction this maximum force is proportional to the reactive force pushing the two objects apart. However this complicates the calculation within our constraint based system. To simplify the model we hold the bounds on λ_1 and λ_2 constant and base the restrictions on a reasonable approximation for what will be the common case. That case is when one object is resting under the force of gravity. Therefore we use the approximation:

$$-\mu mg \leq \lambda_1 \leq \mu mg$$
$$-\mu mg \leq \lambda_2 \leq \mu mg$$

Where μ is a constant of friction g is the acceleration of gravity and m is a portion of the mass of the object associated with the contact point. We divide the mass among the contact points so that friction does not change with the number of contact points.

Constraint Drift & Baumgarte

We have introduced a correction factor for constraints which drift from numerical errors using the equation:

$$\mathbf{JV} = -\beta \mathrm{C}$$

This is called the Baumgarte scheme and allows us to compensate for constraint drift which takes place over the course of our simulation. As we mentioned in our tutorial on numerical integration, our physics system is built on the idea that a series of discrete time steps is analogous to continual motion through space and time.

This, as we point out, isn't really the case. Errors are inherent to every time step. Moreover, since the results our system produces for time step $n + 1$ are based upon the already slightly erroneous results obtained for time step n, the cumulative error over time becomes more pronounced.

In many ways, that doesn't really affect us. Our real-time physics engine is meant to make objects behave believably, not 'accurately'. It isn't meant to be accurate to the level of, for example, a turbine simulation tool used in the construction of real-life jet engines; it's meant, instead, to be able to represent a jet plane flying through the air, and tell us when it hits something, in real-time.

Where it affects believability, however, is problematic for us. More problematic is when it affects *stability*, and that's what the Baumgarte scheme helps us mitigate.

When we first introduced constraints, we said that a constraint should not introduce energy to the system. For this to be true, we stated that $\dot{C} = 0$. The *whole* truth is a little more complex. You should by now have played around with the Baumgarte offset in the framework, and seen the consequences of changing it; the rope bridge begins to sag, or vibrates under its own power.

Essentially, the errors which creep into our iterative updates actually make the equation a little more like $\dot{C} \approx 0$. We need some means of offsetting this, to ensure that our constraint-based system remains stable, and we introduce the Baumgarte offset as a multiple of our constraint to address the inequality:

$$\dot{\mathrm{C}}(t) + \beta \mathrm{C}(t) = 0$$

We clamp this value to a specific region, as a result of the differential equations which underpin its accuracy (these equations are omitted here as they are non-examinable):

$$0 < \beta < \tfrac{1}{\Delta t}$$

Framework Note

As with our other discussion of Jacobians, the Framework abstracts some of this mathematics for you, presenting what is on the face of it a simpler solution.

Implementation

Look into the first couple of tasks from day 5 on the Practical Tasks hand-out. Most of the tasks will need to wait until after this afternoon's lecture, where we pull everything regarding collision response together.

Summary

In this tutorial we have introduced the three forms of collision response, paying attention to the complexities of theories which support them. We've considered the impulse method in depth, and explored the spring equation as a basis for penalty-based collision response. We have explained the nature of collision response in a constraint based system, and defined the Contact Constraint. We have worked through an example of it in practise, and provided code samples to assist in a better understanding of engineering a solution to the problem. We have introduced the idea of friction as a constraint, and provided a more in-depth explanation of the Baumgarte offset. Our next lecture will conclude the physics tutorial series, connecting the dots back to the software implementation in the context of linear solvers.

```
1
2  // After c.sumImpulseFriction = (This is used in the next tutorial)
3
4  // Baumgarte Offset (Adds energy to the system to counter slight
5  // solving errors that accumulate over time - known as 'constraint
6  // drift')
7
8  // Very slightly different to the one we used to prevent the distance
9  // constraints breaking, as we now allow a minimum allowed error.
10
11 // In this case, we allow the objects to overlap by 1mm before adding
12 // in correctional energy to counteract the error, this is a little
13 // dirty trick that results in us always getting a manifold for
14 // constantly colliding objects which normally we would get collide
15 // one frame and not the next, endlessly jittering. If you're
16 // interested, the _slop part (and this issue in general) is usually
17 // handled in physics engines by pretending the collision volume is
18 // larger than it is during the narrowphase, so manifolds are
19 // generated for colliding, and almost colliding pairs.
20
21 const float baumgarte_scalar = 0.1f;
22 const float baumgarte_slop = 0.001f;
23 const float penetration_slop =
24    min(c.colPenetration + baumgarte_slop, 0.0f);
25
26 c.b_term +=
27    -(baumgarte_scalar / PhysicsEngine::Instance()->GetDeltaTime())
28    * penetration_slop;
29
30 //Compute Elasticity Term
31
32 // This is the total velocity going into the collision relative to the
33 // collision normal, as elasticity is 'adding' energy back into the
34 // system we can attach it to our 'b' term which we already add to
35 // 'jt' when solving the contact constraint.
36
37 const float elasticity =
38    pnodeA->GetElasticity() * pnodeB->GetElasticity();
39 const float elatisity_term = Vector3::Dot(c.colNormal,
40
```

```
pnodeA->GetLinearVelocity()
    + Vector3::Cross(c.relPosA, pnodeA->GetAngularVelocity())
    - pnodeB->GetLinearVelocity()
    - Vector3::Cross(c.relPosB, pnodeB->GetAngularVelocity()));

c.b_term += (elasticity * elatisity_term) / contactPoints.size();
```

Extending UpdateConstraint() in Manifold.cpp

```
void Manifold::ApplyImpulse()
{
    for (ContactPoint& contact : contactPoints)
    {
        SolveContactPoint(contact);
    }
}

void Manifold::SolveContactPoint(ContactPoint& c)
{
    Vector3 r1 = c.relPosA;
    Vector3 r2 = c.relPosB;

    Vector3 v0 = pnodeA->GetLinearVelocity()
        + Vector3::Cross(pnodeA->GetAngularVelocity(), r1);
    Vector3 v1 = pnodeB->GetLinearVelocity()
        + Vector3::Cross(pnodeB->GetAngularVelocity(), r2);

    Vector3 dv = v1 - v0;

    //Collision Resolution

    float constraintMass =
        (pnodeA->GetInverseMass() + pnodeB->GetInverseMass()) +
        Vector3::Dot(c.colNormal,
        Vector3::Cross(pnodeA->GetInverseInertia()
        *Vector3::Cross(r1, c.colNormal), r1) +
        Vector3::Cross(pnodeB->GetInverseInertia()
        *Vector3::Cross(r2, c.colNormal), r2));

    if (constraintMass > 0.0f)
    {
        float jn = max(-Vector3::Dot(dv, c.colNormal) + c.b_term, 0.0f);
        jn = jn / constraintMass;

        pnodeA->SetLinearVelocity(pnodeA->GetLinearVelocity()
            - c.colNormal*(jn * pnodeA->GetInverseMass()));
        pnodeB->SetLinearVelocity(pnodeB->GetLinearVelocity()
            + c.colNormal*(jn * pnodeB->GetInverseMass()));

        pnodeA->SetAngularVelocity(pnodeA->GetAngularVelocity()
            - pnodeA->GetInverseInertia()
            * Vector3::Cross(r1, c.colNormal * jn));
        pnodeB->SetAngularVelocity(pnodeB->GetAngularVelocity()
            + pnodeB->GetInverseInertia()
            * Vector3::Cross(r2, c.colNormal * jn));
    }

    // Friction
    Vector3 tangent = dv - c.colNormal * Vector3::Dot(dv, c.colNormal);
```

```
    float tangent_len = tangent.Length();

    if (tangent_len > 1e-6f)
    {
        tangent = tangent / tangent_len;
        float frictionalMass = (pnodeA->GetInverseMass()
        + pnodeB->GetInverseMass()) + Vector3::Dot(tangent,
        Vector3::Cross(pnodeA->GetInverseInertia()
        * Vector3::Cross(r1, tangent), r1) +
        Vector3::Cross(pnodeB->GetInverseInertia()
        * Vector3::Cross(r2, tangent), r2));

        if (frictionalMass > 0.0f)
        {
            float frictionCoef =
                (pnodeA->GetFriction() * pnodeB->GetFriction());
            float jt = -Vector3::Dot(dv, tangent) * frictionCoef;

            jt = jt / frictionalMass;

            pnodeA->SetLinearVelocity(pnodeA->GetLinearVelocity()
                - tangent*(jt*pnodeA->GetInverseMass()));
            pnodeB->SetLinearVelocity(pnodeB->GetLinearVelocity()
                + tangent*(jt*pnodeB->GetInverseMass()));

            pnodeA->SetAngularVelocity(pnodeA->GetAngularVelocity()
                - pnodeA->GetInverseInertia()
                * Vector3::Cross(r1, tangent*jt));
            pnodeB->SetAngularVelocity(pnodeB->GetAngularVelocity()
                + pnodeB->GetInverseInertia()
                * Vector3::Cross(r2, tangent*jt));
        }
    }
}
```

Adding new functions to Manifold.cpp

```
PhysicsEngine::UpdatePhysics()
{
    //A whole physics engine in 6 simple steps =D
    //1. Broadphase Collision Detection (Fast and dirty)
    //2. Narrowphase Collision Detection (Accurate but slow)
    //3. Initialize Constraint Params (precompute elasticity/baumgarte
    //  factor etc)

    for (Manifold* m : manifolds) m->PreSolverStep(updateTimestep);
    for (Constraint* c : constraints) c->PreSolverStep(updateTimestep);

    //4. Update Velocities
    //5. Constraint Solver     Solve for velocity based on external
    //  constraints

    for (Manifold* m : manifolds) m->ApplyImpulse();
    for (Constraint* c : constraints) c->ApplyImpulse();

    //6. Update Positions (with final 'real' velocities)
}
```

PhysicsEngine.cpp