Vx Developer Guide

Contents | Introduction | Classes Overview | First Program | Universes and Assembly | Parts and Sensors | Collision Geometries | Constraints | Materials | Contacts | Fluid Interaction | Vortex Utilities | Graphics Integrations | Tuning for Stable Simulations | Advanced Topics | Sample Graphics Integration | Vortex Q&A

Constraints

A constraint imposes a restriction on the motion of one or several parts. This is expressed using equality or inequality algebraic expressions which are functions of the coordinates and velocities of the parts involved, and which must be satisfied during the motion of the system. Vortex computes the forces which must be applied on the rigid bodies so they satisfy all the algebraic conditions simultaneously, at least approximately.

Constraints offer a way to model physical phenomena without having to model the details of how they work but concentrating on the net result. Instead of computing the complicated atomic forces which are responsible for non-penetration of solids and dry friction forces for instance, we impose the kinematic constraint that solids do not interpenetrate and the extra constitutive laws, or effort constraints, which describe dry friction, on the relative motion of contacting bodies within the contact plane.

The Vortex library contains several types of constraints already and this is sufficient for most users to understand how to configure these to achieve a desired effect, e.g., put a hinge joint between two bodies, attach steering wheels to a car model, etc. Note that contact constraints are generated directly by the geometric collision detection module and, therefore, the user never has to create them explicitly, though the physical properties of contact constraints are configurable by the user via the material identifiers and corresponding VxMaterial objects.

The constraints covered in Vortex include pure kinematics constraints such as a ball and socket joint, a hinge joint, or a fixed distance constraint for instance. Inequality constraints include contact constraints and joint limits. Velocity constraints allow the user to model drivers and motors.

Vortex also provides some effort constraints which impose relationships between the kinematic variables and constraint force applied on the parts. For instance, one can specify a driver which attempts to produce a given target velocity, but never delivers a constraint force larger than the preset value. Another example of an effort constraint is the Vortex approximation of the Coulomb friction force law which is covered in detail below.

Constraint Configuration

For all constraint classes defined in Vx, one of the constraint constructors allows complete configuration in one function call. For most constraints, the full configuration includes a pair of VxPart objects, a constraint attachment position given in world coordinates, as well as one or two extra axes, also in world coordinates. For the case of a ball joint for instance, the following code will create and configure the constraint:

```
#include "Vx/VxPart.h"
#include "Vx/VxBallAndSocket.h"

// ...

VxUniverse *u;
VxPart *p1;
VxPart *p2;
VxBallAndSocket *bs;
VxReal3 x = {1, 2, 3};   // the center of rotation of the ball

// create set the position and orientation of the two VxPart objects
// ...

// create and configure the ball joint
bs = new VxBallAndSocket(p1, p2, x);  // create and configure the
                                      // ball joint
u->disablePairIntersect(p1, p2); // disable contacts between p1, p2
u->addConstraint(bs);            // register constraint in universe
                                 // to enable simulation
```

The configuration details differ for each constraint type but there are common elements and corresponding common methods between all different constraint objects which are all derived from a common abstract base class. Each constraint contains a list of constrained VxPart objects. Each of these can be in any of the allowed states, namely, frozen, kinematic, or dynamic, indifferently. The number of parts involved in a given constraint can vary but is often one or two. For instance, if a dynamic part is in contact with the ground, there is just one part in the contact constraint. A hinge joint can involve one or two parts, and a differential constraint can involve up to six parts. For each part, the user must then specify an attachment frame of reference that is fixed on the given part. The function of these attachment frames differs for each constraint type though.

Other than specifying the list of constrained parts, the user must also specify the geometric configuration of the constraint itself. This can be done either through the specific constructor or with a set of methods common to all constraints. The essential components of geometric constraint configuration include the attachment frames of each of the involved parts, as well as one or two axes, depending on the constraint type. This can be done in a number of different ways as follows.

A constraint can be disabled at any time by calling

```
constrain->enable(false);
```

A disabled constraint has no effect on the simulation. For a constraint to be active, it must be enabled and it is necessary that the constraint and all the parts added to it are added to the universe. As long as those conditions are not met, the constraint will remain internally disabled. This can be verified using

```
bool b = constraint->isEnabledInternal();
```

Constraint Attachments

One convenient way to specify the geometric configuration of a constraint is to set the constraint geometric center and axes using world coordinates. Assuming that user is setting a VxHinge between two VxPart objects which have already been moved to their initial position and orientation, the following code example configures the VxHinge via the generic methods instead of through the constructor:

```
#include "Vx/VxPart.h"
#include "Vx/VxHinge.h"

// ...

VxUniverse *u;
VxPart *p1;
VxPart *p2;
VxHinge *hinge;
VxReal3 x = {1, 2, 3};      // the center of rotation of the hinge
VxReal3 n  = {0, 0, 1};     // the axis of rotation of the hinge

// set the position and orientation of the two VxPart objects
// ...

hinge = new VxHinge();           // create and configure the hinge
hinge->setParts(p1, p2);         // set parts
hinge->setAttachmentPosition(x); // set current center of rotation
hinge->setAxis(n);               // set current axis of rotation
u->disablePairIntersect(p1, p2); // disable contacts between p1, p2
u->addConstraint(hinge);         // register constraint in universe
                                 // to enable simulation
```

In this case, the VxHinge object will convert the given position and axis into each of the body coordinate frames of VxPart `p1` and `p2`. The fixed body position and orientation of the attachments will be used dynamically to compute forces required to keep the hinge condition satisfied.

The *attachment position* is a coordinate system, which includes a point and two axes (the third is implicit) that is rigidly attached to a part with a fixed offset and orientation, and is typically specified in world coordinates for the current (initial) part position. It is used to determine the relationship of that part to the other part(s) in the constraint.

Advanced: Setting Relative Constraint Attachments

Alternately, it is possible to setup the attachment frame on each of the VxPart objects in world coordinates. When doing this, it is not necessary that the constraint condition be satisfied with the given data: constraint stabilization will resolve this over a short amount of simulation time. It is not recommended however for the distances to be too large as large velocities will be induced. Assuming the same setup as in the previous code example, this is done as follows:

```
hinge->setPartAttachmentPosition(0, x);
hinge->setPartAttachmentPosition(1, x);
```

If the user knows exactly the rigid body coordinates of the attachment position and orientation, she may set that directly using the relevant methods as follows:

```
VxReal x0 = {0, 0, 2};
VxReal n0 = {0, 0, 1};
hinge->setPartAttachmentPositionRel(0, x0);
hinge->setPartAttachmentAxesRel(0, x0);
hinge->setPartAttachmentPositionRel(1, n0);
hinge->setPartAttachmentAxesRel(1, n0);
```

or similarly, if the position is known relative to the center of mass of the part it is possible to call

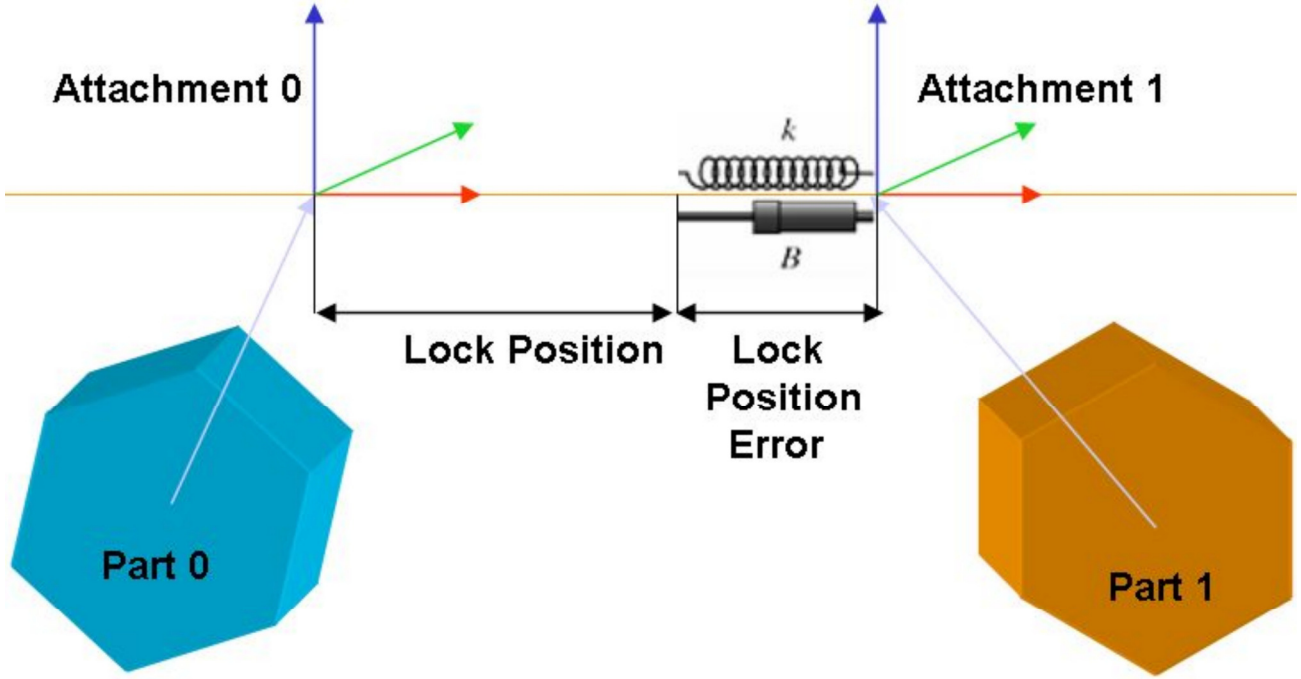
```
hinge->setPartAttachmentPositionCOMRel(0, x0);
hinge->setPartAttachmentAxesRel(0, x0);
```

As there are no offset orientation between the part center of mass and the part, the method setPartAttachmentAxesRel() still applies.

In this case, the center of rotation is not specified explicitly, neither is the rotation axis. However, the constraint stabilization mechanism will eventually make the world coordinates of the attachment point and axis on each body coincide. If you know the body coordinates of your attachment points explicitly, this is the mechanism to use. For the example shown above, the two VxPart objects are labelled 0 and 1 respectively. Note that these calls override any attachments set using the setAxes() or setAttachmentPosition() calls so these two sets of calls are not typically used at the same time.

Note also that it is important that relevant attachment points and axes for *all* parts involved in a constraint need to be set for a constraint to work properly. Note however that for certain constraints, axes or positions are not relevant and don't need to be set at all: for example the ball and socket axes need not be set.

The picture below illustrates the attachment coordinate systems



Degrees of Freedom and Constraint Coordinates

The number of *degrees of freedom* of a system indicates the minimum number of independent parameters needed to describe the motion of a system. A point particle constrained to move on a line has one degree of freedom, a free rigid body has six. The relative orientation of two bodies constrained by a hinge has one degree of freedom, and that of two bodies constrained by a ball joint has three degrees of freedom.

Degrees of freedom can be mapped to coordinate systems locally but it is important to realise that it is not always possible to have a global Cartesian coordinate system in all cases. The most important counter example for this discussion is a ball joint, which has three rotational degrees of freedom. In this case, one can use three Euler angles (in any of the 24 possible conventions) to parameterise the orientation. However, the three angles are not independent of each other for all configurations. Indeed, each of the 24 conventions breaks down for some values of the three angles, something known as gimbal lock.

Nevertheless, for some constraints, there is a set of coordinates which are globally independent and orthogonal. A hinge joint has one degree of freedom which is an angle, a prismatic joint has one degree of freedom which is linear, a cylindrical joint has two independent degrees of freedom, one which is an angle, the other is a linear displacement, etc.

Vortex offers a unified interface to add control (i.e., additional constraints) on the degrees of freedom, at least in cases where there exists an orthogonal coordinate system to parameterize the degrees of freedom.

First off, the user can find out how many degrees of freedom a given constraint has by invoking the following method:

```
int VxConstraint::getCoordinateCount() const;
```

A degree of freedom specified by a single parameter will either be linear or rotational. In either case, it is possible to specify a point within or rotational. In either case, it is possible to specify a point within the coordinate system from which to measure movement. The following functions set or get an offset used to transform the measured relative position coordinate into the user's coordinate system:

```
void VxConstraint::setCoordinateCurrentPosition(CoordinateID coordinate,
                                                VxReal newPos)
VxReal VxConstraint::getCoordinateCurrentPosition(CoordinateID coordinate)
```

Then, the user can set control on this coordinate by adding range limits, coordinate lock, or velocity motor. From this point, we will refer to degrees of freedom as *constraint coordinates*, or sometimes *joint coordinates*, since we will only be concerned with degrees of freedom that are accessible through specific parameterisations.

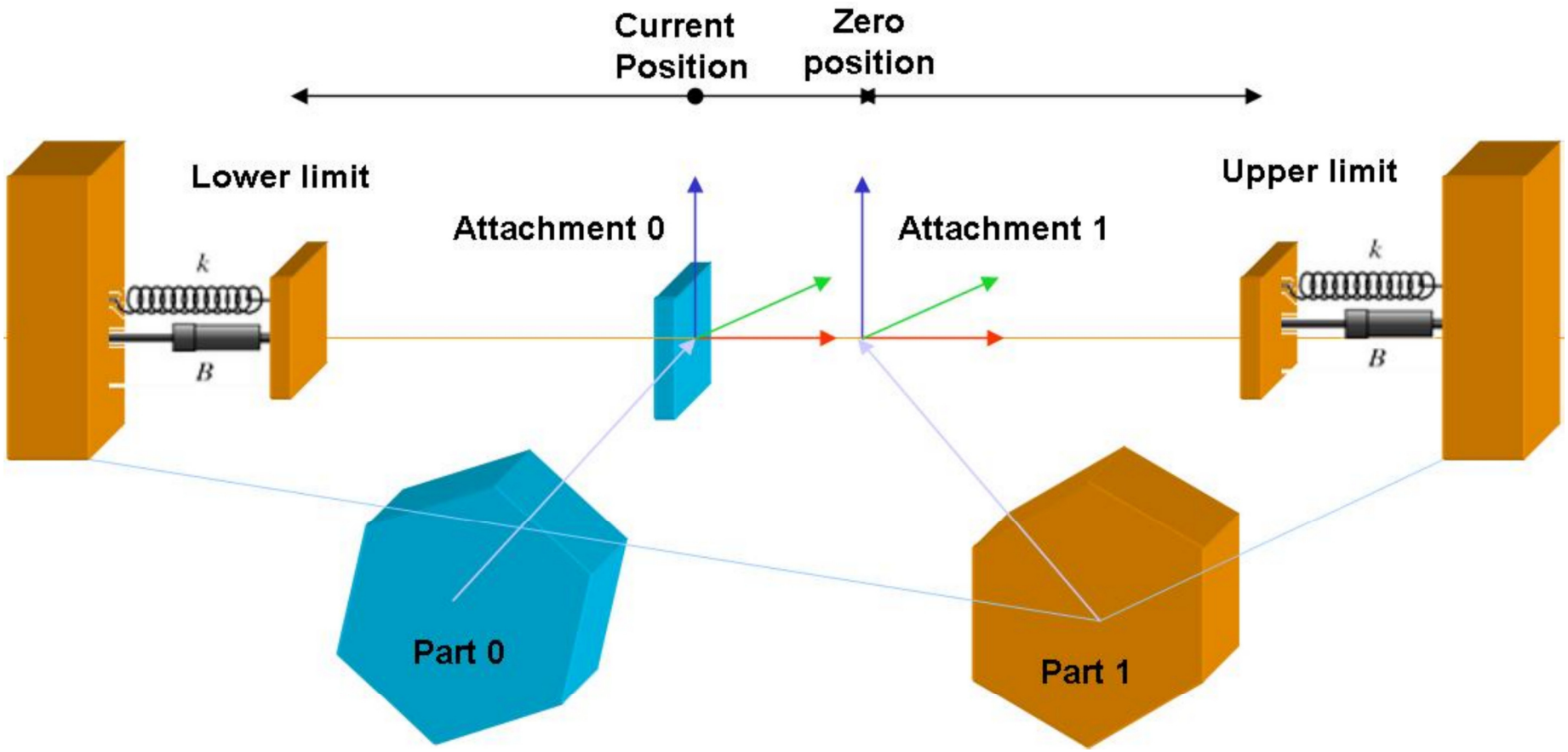
Limiting Joint Coordinates

The parameterised movement of some joint constraints can be limited by specifying bounds for the joint coordinates. These limits let a developer create simple models of real-world behavior: a hinge can be limited, for example, so that it only opens 270 degrees. A joint can have one or more limits, or none, depending upon its geometry.

Each limit consists of an upper and a lower bound. You can specify these limits independently to appear either hard or soft. A hard bounce reverses the velocities of both bodies in a single time step, while a soft bounce may take many time steps to complete. If the limits are soft, you can also set their damping, so that, beyond the limits, the joint behaves like a damped spring. If the limits are hard, you can set the limit restitution between zero and one to govern the loss of momentum as the bodies rebound.

As an example, consider the prismatic joint, which allows only linear motion along an axis between parts. Here the upper limit will be full extension, and the lower limit the fully retracted position. It is always possible to set the values for these joint limits, but some joints do not use the values, and do not respond to changes in these values. For example, the ball and socket will not respond to changes in the limits.

The picture below illustrates the attachment coordinate systems and the limits with compliance and damping, represented by a spring and damper for the prismatic joint.



It is also possible to use the contacts between two objects connected by a joint to prevent movement. For example, when a door turns on its hinges, its motion is blocked by the door frame. However, the more contacts generated in the system, the slower the simulation will run. Wherever possible, use joint limits rather than contacts to control the movement of joints. In fact collision between constrained parts is disabled as in the first hinge example above.

To set or get the upper and lower limits for a particular joint coordinate, use the following functions:

```
void VxConstraint::setLowerLimit(CoordinateID coordinate, VxReal limitPos,
                                 VxReal limitVel, VxReal restitution,
                                 VxReal stiffness, VxReal damping);
void VxConstraint::setLowerLimit(CoordinateID coordinate, VxReal limitPos,
                                 VxReal limitVel, VxReal restitution,
                                 VxReal stiffness, VxReal damping);
```

Angular limit positions are specified in radians. The default value for the bounds is VX_INFINITY, which means unlimited.

Critical damping for a given amount of stiffness can be computed using the method

```
VxReal VxUniverse::getCriticalDamping(VxReal stiffness,
                                      VxReal halfLife) const;
```

where halfLife is roughly the number of frames required by the solver to recover from a constraint violation; it is specified in multiples of the current time step.

To activate a limit, or check if it is activated:

```
void VxConstraint::setLimitsActive(CoordinateID coordinate,
                                   bool activate);
bool VxConstraint::getLimitsActive(CoordinateID coordinate);
```

The limit restitution, stiffness and damping specify whether the constraint should exhibit spring-like behavior when a limit is exceeded, or an immediate bounce. The coefficient of restitution is the ratio of rebound velocity to impact velocity when the joint reaches the lower or upper limit. Restitution must be in the range zero to one inclusive: the default value is zero, which results in inelastic collision. Note that if the stiffness is not VX_INFINITY and the damping is not critical, there will be some rebound even if the restitution is 0.

Actuating a Joint Coordinate (Motors)

Some joints can be motorized, allowing you to control the relative movement of the bodies along the unconstrained degrees of freedom. This simulates a motor driving motion along or about the axis. As the motor acts, the joint constraints are maintained. Additionally, the power applied to move a joint can be limited, allowing for a very realistic simulation of machinery. The joint limits discussed in the previous section can be enabled, and the motor will respect the limits. Motorized joints with power limits allow very stable modeling of things like engines, brakes, motors, and stiff springs.

For example, to characterize a hinge motor, you set a desired angular speed and the motor's maximum torque. A torque no greater than this is applied to the hinged bodies to change their relative angular velocity, until either the desired velocity is achieved, or the hinge angle hits an upper or lower limit, if one has been set. The loss parameter will cause slipping between the motorized parts.

```
void VxConstraint::setMotorParameters(CoordinateID coordinate,
                                      VxReal desiredVelocity,
                                      VxReal maxForce, VxReal loss);
void VxConstraint::setControl(CoordinateID coordinate,
                              VxConstraint::kControlMotorized);
```

Attention:

To achieve physically realistic simulation results, realistic values must be chosen for motor maximum force. By default, `maxForce` is set to VX_INFINITY, which will give unrealistic results.

Locking a Joint Coordinate

For some joints, it is possible to lock a joint in place. This lock can be maintained up to a specified force limit, so that applied forces larger than this limit will break the lock and cause movement. In the absence of such large forces, the joint will behave as if one joint coordinate has been lost. A joint lock can be used for example to simulate a motor with position control, always bringing the joint to the desired position.

```
void VxConstraint::setLockParameters(CoordinateID coordinate,
                                     VxReal lockPosition,
                                     VxReal maxForce, VxReal stiffness,
                                     VxReal damping, VxReal velocity );
```

To lock the constraint and activate it in a single call:

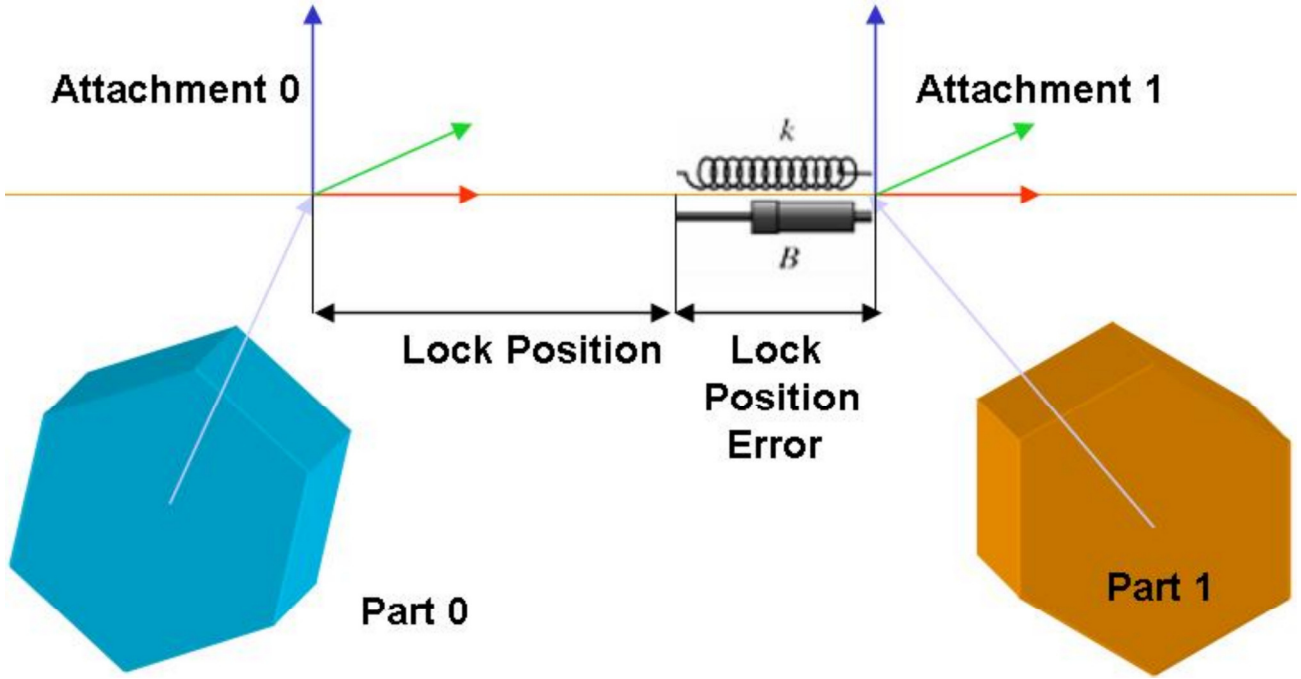
```
void VxConstraint::setControl(CoordinateID coordinate,
                              VxConstraint::kControlLocked );
```

Attention:

To achieve physically realistic simulation results, realistic values must be chosen for lock maximum force, stiffness and damping. To determine what realistic values are, you'll need to know the masses of objects involved and/or the forces that will be applied externally to the constraint.

The picture below illustrates the attachment coordinate systems and the position lock with compliance and damping, represented by a spring and damper for a linear 1 degree of

freedom constraints such as the prismatic constraint.



Constraint Relaxation

Vortex computes constraint forces explicitly for each removed degree of freedom. By default, the constraint equations are satisfied almost exactly, within limits allowed by numerical precision and system stability. In some cases, the user will want to relax some of the constraint equations.

Relaxation can be useful for introducing some compliance in the constraints and contact properties to stabilize the simulation and model the fact that real-world objects and contacts between them are not perfectly rigid, etc. For example, a prismatic constraint can be allowed to go slightly out of alignment, given a sufficient lateral force. Constraint relaxation allows the user to simulate slightly compliant objects, such as long beams and cables. The advanced section in the Vx Programming Guide describes its use for stabilizing a simulation.

First off, the user can find out how many individual constraint equations there are on a given constraint by invoking the method:

```
int VxConstraint::getConstraintEquationCount() const;
```

Next, the meaning of each constraint equation is described in an enum member of each of the constraints:

```
VxHinge::kConstraintP0  //Relative displacement along primary axis.
VxHinge::kConstraintP1  //Relative displacement along secondary axis.
VxHinge::kConstraintP2  //Relative displacement along third axis.
VxHinge::kConstraintA1  //Relative rotation about secondary axis.
VxHinge::kConstraintA2  //Relative rotation about third axis.
```

For each of the fields in this enumeration, the user can set **compliance** and **damping** parameters which apply to position constraints, and **loss**, which apply to velocity constraints, through the method:

```
void
VxConstraint::setRelaxationParameters(ConstraintEquationID c,
                                      VxReal stiffness, VxReal damping,
                                      VxReal loss, bool enabled);
```

This interface enables the user to set the same set of parameters which are described in Section **Setting Global Compliance, Damping and Kinetic Loss**, but for an individual constraint equation. This is in fact the preferred method of stabilizing a simulation which exhibits overly large forces, see that section for more detail.

Some constraints allow relative part orientation to change. In this case the reference axes of the parts are usually not matching. For example in the hinge constraint, the part0 and part1 primary axes always match but the hinge angle is defines as the relative orientation of the parts' secondary axes. If one were to relax the constraint along VxHinge::kConstraintP1, it is necessary to specify along which par's secondary axis the relaxation should happen. By default no part is chosen. This can be changed using

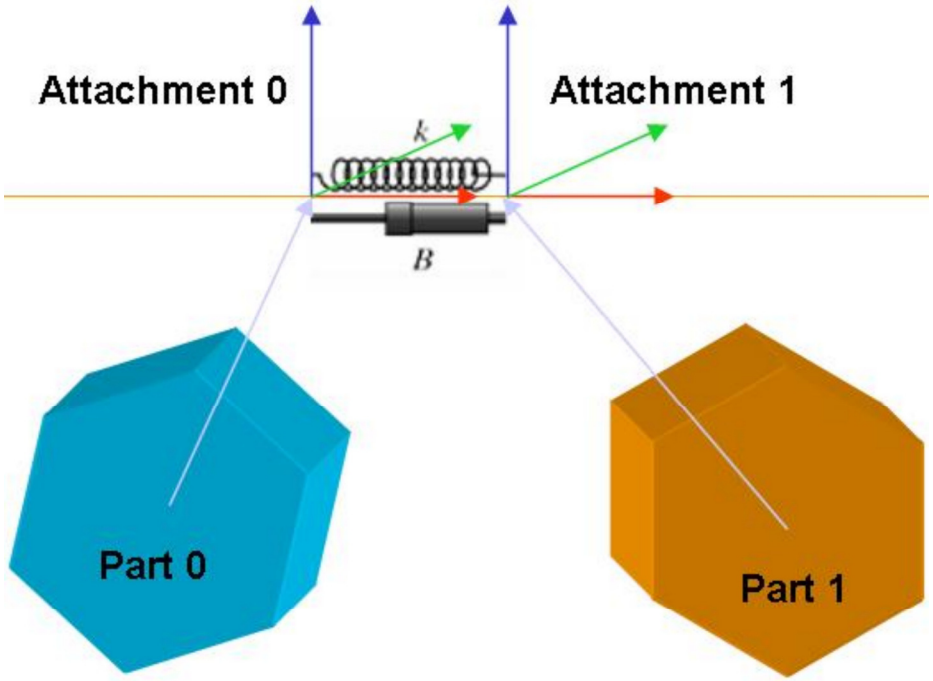
```
VxHinge::setLinearAxisReferencePartIndex(LinearAxisReferencePartIndex ref);
```

where *ref* is either *kLinearAxisReferencePartIndex0* for part0, *kLinearAxisReferencePartIndex1* for part1 or *kLinearAxisReferencePartIndexNone*. This api is available for VxHinge, VxBallAndSocket, VxUniversal and VxHomokinetic.

Note:

the constraint may become unstable if the reference part's angular velocity becomes very large. If *kLinearAxisReferencePartIndexNone* is select, the default, both part may spin as fast as desired.

The picture below illustrates the attachment coordinate systems with a spring and damper representing the relaxation along one constraint equation (removed degree of freedom), though again, relaxation can occur in any removed degree of freedom.



Constraint Friction

It is possible to specify internal friction in the constraint. The Friction is available in any constraint coordinate or constraint equations using one of

```
VxConstraintFriction* VxConstraint::getCoordinateFriction(CoordinateID coordinate) const;
VxConstraintFriction* VxConstraint::getConstraintEquationFriction(ConstraintEquationID c) const;
```

The class VxConstraintFriction allows specifying the friction parameters. The friction force can be specified manually or, in proportional mode, can be computed from the constraint's force as applied on the connected part and a friction coefficient. In this mode, the friction increase with the constraint's tension which may add realism to the simulation. The internal friction is very similar to the friction models VxMaterialBase::kFrictionModelBox and VxMaterialBase::kFrictionModelScaledBoxFast and uses similar parameters

```
VxConstraintFriction::setProportional(bool proportional);
VxConstraintFriction::setCoefficient(VxReal coeff);
VxConstraintFriction::setMaxForce(VxReal force);
VxConstraintFriction::setLoss(VxReal loss);
```

Static and dynamic friction can be simulated using the static friction scale specifying the force ratio between static and dynamic friction:

```
VxConstraintFriction::setStaticFrictionScale(VxReal scale);
```

The amount of force or torque added by the friction can be accessed using

```
VxReal VxConstraint::getCoordinateFrictionForceLastFrame(CoordinateID coordinate) const;
VxReal VxConstraint::getConstraintEquationFrictionForceLastFrame(ConstraintEquationID c) const;
```

Internal friction is only available on two part's constraints. Internal friction availability can be obtained using

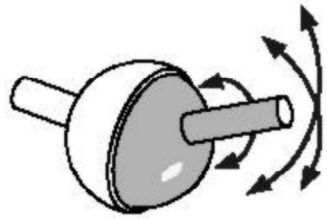
```
bool VxConstraint::getCoordinateFrictionAvailable(CoordinateID coordinate) const;
bool VxConstraint::getConstraintEquationFrictionAvailable(ConstraintEquationID c) const;
```

Note:

Adding proportional internal friction helps improvincreasing simulation robustness while simulating chain of objects with large mass ratios.

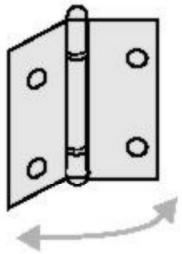
Available Constraint Classes

VxBallAndSocket



A ball and socket joint (VxBallAndSocket) forces a point fixed in one body to be at the same location as that of a point fixed in another body, removing three degrees of freedom. In the diagram shown above, the center of the sphere on one body always coincides with the center of the socket on the other body. This ideal joint allows all rotations about the common point. This joint is sometimes referred to as a spherical, or ball joint.

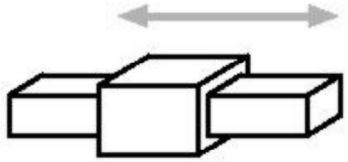
VxHinge



A hinge joint (VxHinge) leaves a pair of bodies free to rotate about a single axis (the hinge axis), but otherwise completely fixed with respect to each other. The axis has a fixed position and orientation in each rigid body, and the hinge constraint forces those axes to coincide at all times. In doing so, it removes five degrees of freedom, and is therefore more computationally costly than, for example, a ball and socket joint.

A hinge joint can be used to attach a gate to a gatepost, a lever to its fulcrum, or a rotating part such as a wheel to a chassis, a propeller shaft to a ship, or a turntable to a deck.

VxPrismatic



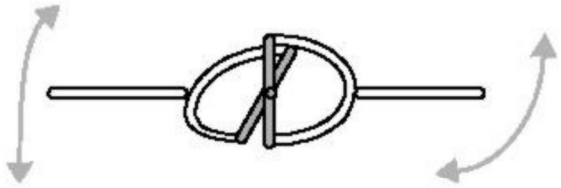
The prismatic joint (VxPrismatic) is imagined as a bar with a block that slides along it without rotating. The name ‘prismatic’ refers to the imagined shape of the bar whose non-circular cross-section restricts rotation by keying into the matching hole in the block.

A prismatic joint leaves a single, linear, degree of freedom which can be controlled by a displacement coordinate. It constrains the two remaining linear DoFs and all three angular DoFs, as it acts to maintain the relative orientation of the parts.

The prismatic joint is not symmetric as its two attachments have different geometric roles:

1. The ‘bar’ is a line defined by an axis fixed in the first attachment.
   If one attachment is absolute it should usually be the first one as this fixes a line in the world.
2. The ‘block’ is a point position corresponding to the origin of the second attachment.
   Making the second attachment absolute fixes a point in the world. Combined with relaxation of the angular constraint this point acts as a springy pivot.

VxUniversal



In the universal joint (VxUniversal), two axes, one fixed in each of the two constrained bodies, are forced to have one point in common and to be perpendicular at all times. This is a lot like the ball-and-socket joint. But here the ball is not allowed to twist in its socket.

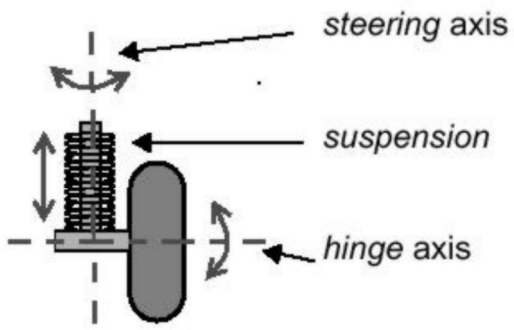
A universal joint removes four degrees of freedom from the attached bodies: it fixes their relative positions and also prevents them from twisting about a third axis, perpendicular to the two given axes. This joint can also be pictured as a joystick mechanism in which two hinges, with perpendicular axes, are joined one on top of the other, allowing an attached stick to move first in the x-direction then in the y-direction.

This mechanism is also known as a gimbal, and suffers from the phenomenon of gimbal lock. Gimbal lock relates to the common use of the universal joint in transmitting torque from one body to another around a small bend. The transmission becomes increasingly unsmooth as the angle of bend approaches ninety degrees, and finally cannot transmit torque at all.

VxHomokinetic

A Homokinetic joint is similar to the universal joint, but exhibits no gimbal lock or rotational velocity variations. It constraints the rotation of the first part around its primary axis to the rotation of the second part around its primary axis and it also maintains relative positional constraint.

VxCarWheel



The car wheel joint (VxCarWheel) models the behavior of a car wheel, with steering and suspension. It is like a combination of two hinge joints, one for the steering and one for wheel rotation, along with a prismatic joint for the suspension.

The first part is the chassis and the second part is the wheel. The connection point for the wheel part is its center of mass. The default steering axis is the Z-axis of the first part, and the default hinge axis is the Y-axis of the second part.

It is useful to think of this constraint as having two main components, the steering column and driving axle. In the API, these are referred to as the steering and hinge components respectively. The steering column is attached at point p0 which is fixed on the chassis and at point p1 which is the center of mass of the wheel. The steering column has a rest length, upper and lower limits, and a spring constant for oscillations about the rest length.

The axle can be made to rotate about the steering column in a plane perpendicular to it. This angle is controlled with either a position control or a limited-force, constant-velocity motor, i.e., a motor which applies up to the specified maximum torque to achieve the specified velocity. The motion of the wheel about the axle is either free or controlled with another limited-force, constant-velocity motor. There are many parameters controlling this constraint: for details see the Vx Reference Manual.

VxCylindrical

The VxCylindrical joint simulates a tubular block sliding on a cylindrical rod allowing both linear and rotational movement about the axis direction. It acts like a prismatic and a hinge together. The linear and angular coordinates can be controlled independently.

The cylindrical joint is not symmetric as its two attachments have different geometric roles:

1. The 'rod' is a line defined by an axis fixed in the first attachment.
   If one attachment is absolute it should usually be the first one as this fixes a line in the world.
2. The 'block' is a point position corresponding to the origin of the second attachment.
   Making the second attachment absolute fixes a point in the world. Combined with relaxation of the angular constraint this point acts as a springy pivot.

VxDifferential

The VxDifferential constraint is a velocity-based constraint that can take up to 6 parts and simulates a car's differential joint, the 3 first parts being the shaft and the two driven wheels, the 3 last parts being the reference for the 3 first ones respectively. The user can set the linear relationship between the angular velocities of the constrained parts. For instance, in a standard differential, the angular velocity of the drive shaft is equal to the average angular velocity of the output shafts. This constraint is licensed with the VxVehicle license.

The part's velocities are taken relative to the constraint attachment's primary axis relative to its reference part. In the case of a car's differential, as the wheels attached to the differential may be steered, the relative position of the wheels relative to their reference part, the chassis, will change. By default, the differential will recompute the reference part's attachment axis automatically so that the steered wheels won't cause problem. This default behaviour can be disabled using

```
VxDifferential::useIndependentReferenceAxes(true);
```

It is possible to convert the constraint so that linear velocity may be considered instead of angular velocity for a given part.

```
VxDifferential::setMotionAngular(partIndex, false);
```

By default the constraint is velocity-based. It is possible to convert it to be position-based by calling

```
VxDifferential::setPositional(true);
VxDifferential::resetPositions();
```

where the method resetPositions() will set the part's and the differential reference positions such that the constraint is initially at rest. This can be used to prevent slip which happens with velocity-based constraints with a loss parameter > 0 (and some loss will always be present due to the minimum loss specified in the Universe solver parameters). The reference positions are available using

```
VxDifferential::setPartReferencePosition(partIndex, pos);
VxDifferential::getPartReferencePosition(partIndex);
VxDifferential::setDifferentialReferencePosition(VxReal pos);
VxDifferential::getDifferentialReferencePosition();
```

Finally the speed at which constraint violation is recovered in position mode is modified using

```
VxDifferential::setPositionOffsetMeanLifeTime(lifeTime);
```

VxDistanceJoint

The VxDistanceJoint is a position-based constraint that constrains two parts' attachment points to be within a given distance.

VxDoubleHinge

The VxDoubleHinge can be thought of as an assembly consisting of a hinge attachment fixed on one VxPart, a distance constraint to a second part, and a hinge fixed on the second VxPart. This can be used to simulate a torsion bar rod arm and wheel assembly on a tracked vehicle, for instance. The user controls the suspension of the torsion bar, the stiffness of the road arm, as well as the drive or brake on the rolling wheel.

VxGearRatio

The VxGearRatio constraint is a velocity-based constraint that constrains the velocity of a body relative to its primary attachment axis to the velocity of another body relative to

its primary attachment axis. The gear supports up to 4 bodies where the third and the fourth bodies would be used as reference frame for the first and the second bodies respectively. This means that the gear will only act on the two first bodies' velocity relative to their reference bodies' velocity.

The part's velocities are taken relative to the constraint attachment's primary axis relative to its reference part. Normally the offset transform between a part and its reference part should be fixed. In case it is not the reference part axis is automatically recomputed. This default behaviour can be disabled using

```
VxGearRatio::useIndependentReferenceAxes(true);
```

By default, the constraint uses the part's angular velocity but it is possible to specify for it to consider linear velocity instead:

```
VxGearRatio::setMotionAngular(partIndex, false);
```

This would allow for example to use the gear ratio constraint to simulate a screw joint constraint.

By default the constraint is velocity-based. It is possible to convert it to be position-based by calling

```
VxGearRatio::setPositional(true);
VxGearRatio::resetPositions();
```

where the method resetPositions() will set the part's and the differential reference positions such that the constraint is initially at rest. This can be used to prevent slip which happen with velocity-based constraint with a loss parameter > 0. The reference positions are available using

```
VxGearRatio::setPartReferencePosition(partIndex, pos);
VxGearRatio::getPartReferencePosition(partIndex);
VxGearRatio::setDifferentialReferencePosition(VxReal pos);
VxGearRatio::getDifferentialReferencePosition();
```

It is possible while in position mode to specify the gear back lash using

```
VxGearRatio::setBacklash(backLash);
```

Then the current position within the backlash is adjusted to fit a graphical representation of the gear using the setDifferentialReferencePosition method.

Finally the speed at which constraint violation is recovered in position mode can be modified using

```
VxGearRatio::setPositionOffsetMeanLifeTime(lifeTime);
```

VxContactGear

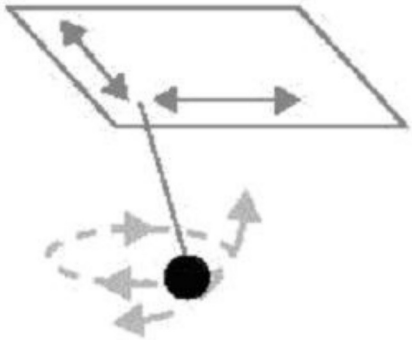
The VxContactGear is a position-based constraint simulating the contact between two gears teeth. The contraints takes two parts. The two gears radius are provided as well as an optional backlash distance. Radius = 0 may be used to simulate a linear gear. In this case the primary axis of the part is along the gear otherwise, for circular gear, the primary axis is the rotation axis. The max force may be tuned using the setMaxForceForward and setMaxForceReverse methods to simulate soft teeth. In this case the gear teeth count must also be provided.

The steps to follow for setting up this constraint are:

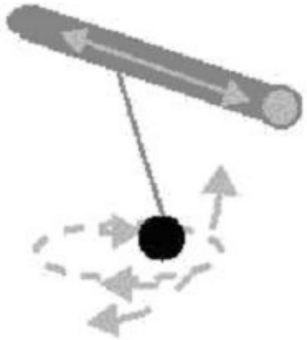
- Set the gear parts, axes and radii
- Set the gear offset position so that one gear's teeth is positionned exactly between two of the other gear's teeth while simulating.
- Set the backlash distance so that the relative gears displacement is realistic.
- If the gear is allowed to jump teeth, set the gear max forces and teeth count for the circular gear so that accurate teeth thickness can be extracted.

The gear constraint won't add any force if the distance between the gear parts exceeds the sum of the gear's radius.

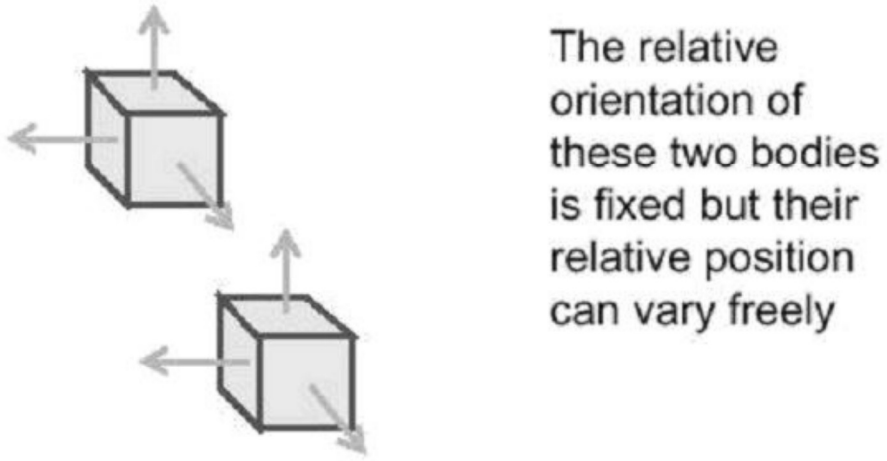
Linear and Angular Constraints



- VxLinear1 is a planar constraint which keeps a point fixed on body 0 onto a plane defined to be perpendicular to the first axis of body 1, with an origin at the attachment position of body 1. There are no limits available on this constraint.



- The VxLinear2 constraint removes two degrees of freedom by confining a point fixed on part 0 to move on a line fixed to part 1. Unlike the case for a cylindrical or prismatic joint, there is no constraint on the bodies' orientations.
- VxLinear3 extends on the VxLinear2 constraint, adding one perpendicularity constraint so that the axis of body 0 is kept perpendicular to the axis of the line that is fixed in body 1.



- The VxAngular3, or coplanar, joint removes three rotational degrees of freedom by constraining a body to have a fixed orientation with respect to another. In other words, while one body can move freely in space (irrespective of the other body's location) its orientation is fixed relative to the other body's orientation. It is possible to add one rotational degree of freedom about a specified axis, enabling a rotation of a body with respect to the other, effectively converting the VxAngular3 joint to an Angular2 joint. Angular3/2 joints are useful for keeping things upright, particularly when relaxation is used as well. Note that the VxAngular3 can be replaced with a VxRPRO for which the linear force is set to 0. This would provide the same constraint type but with additional robustness and at increased cpu cost.

VxScrewJoint

The VxScrewJoint constraint is a velocity-based constraint that constrains the angular velocity of a body along an axis to the linear velocity of another body around another axis. The screw joint supports up to 4 bodies where the third and the fourth bodies would be used as reference frame for the first and the second bodies respectively. This means that the screw joint will only act on the two first bodies' velocity relative to their reference bodies' velocity. As this is stricckly a velocity constraint, in general, the rotative part and the translative part constrained by other constraint. For instance, they could be a VxHinge and a VxPrismatic. The screw joint constraint allow also to set a linear velocity offset.

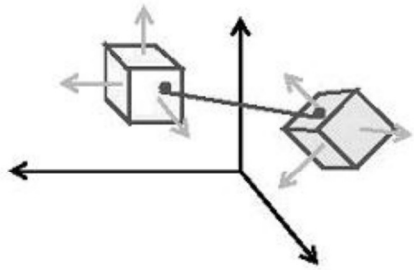
VxSpring

The VxSpring constraint creates a spring-like constraint between two parts' attachment points.

This joint attaches one part to another, or to the inertial reference frame, at a given separation. The VxSpring constraint tends to restore itself to its resting or "natural" length, the length at which no spring force is generated, by opposing any stretching or compression. The mathematical relation between the force **F** exerted by a spring, its resting length **l**, its stiffness **k**, and the distance **d** between its two endpoints is called Hooke's Law and is written as: **F = -k (d-l)**

The upper limit can be given a velocity to implement a winch like behaviour with a small spring stiffness or control set to free.

VxRPRO



VxRPRO, which stands for Relative Position Relative Orientation, is a robust quaternion-based multi-purpose constraint used for controlling the relative transform between two VxPart objects.

A relative position, relative orientation joint can be used to join two simulated objects with a relative distance between them, as well as with a specified relative orientation. Parts joined by a relative position, relative orientation constraint that has not been relaxed have no freedom to move with respect to each other, in other words, the joint removes all six degrees of freedom from the attached parts.

However, the relative orientation can be animated by supplying a sequence of relative quaternions and relative angular velocities. You can set one of the joint attachments to a specific location of a part and specify the relative orientation of that attachment with respect to the part's reference frame. An actual angular movement between the two bodies is done by updating the orientation between the two bodies using setRelativeQuaternion() and the relative angular velocity using setRelativeAngularVelocity().

The relative transform between the two parts can be computed as follows. If A_0 and A_1 are the matrices corresponding to the attachements of part 0 and part 1 respectively, with columns formed by the primary, secondary and orthogonal axes, and if R_0 and R_1 are the part transform rotations, then the relative orientation between the two part coordinate systems is: $R = (R_1 A_1)^{-1}(R_0 A_0)$.

If you are changing the relative orientation dynamically, you must compute an angular velocity to help the constraint stabilize to the new value more quickly.

One of the most useful features of the RPRO joint is to be able to set its attachments and orientation relative to the current part relative positions, using the method

```
rpro->setRelativeQuaternionFromPart();
```

which would freeze the two constrained parts' relative transforms.

Note:

This constraint currently breaks the regular attachment and relaxation constraint API. Attachment axes need to be set using specific VxRPRO member functions. This will be corrected in the future.

VxRPRO Relaxation

For relaxation, the standard VxConstraint functions such as VxConstraint::setRelaxationParameters() are still available. Refer to the VxRPRO::ConstraintEquation enum. It is possible to set the constraint maximum force independently on the 3 linear and the 3 angular degrees of freedom. If the constraint maximum force for the 3 linear degrees of freedom is set to 0, it would act as an angular 3 with the difference of additional robustness due to its quaternion based implementation.

VxRPRO Plastic Deformation

Plastic deformation can be implemented by telling the constraint to reset itself to the current part relative position automatically at every frame, which, after a displacement between the attachements points, causes the constraint to remain in the displaced state. The function

```
rpro->setPlasticDeformationEnable();
```

enables this automatic behaviour.

VxWinch

VxWinch is a useful 4 part constraint that simulates the winch-pulley-hook-winchReference apparatus. This constraint is licensed with the VxCable license.

The constraint's part0 is the driving part whose movement, relative to part4, will be used, via a given ratio, to update the desired distance between part1 and part2 attachment positions. The current desired winch distance is available using:

```
VxWinch::setDistance(distance);
VxWinch::getDistance();
```

Distance limits can be set using

```
VxWinch::setMinDistance(VxReal value);
VxWinch::getMinDistance() const;
VxWinch::setMaxDistance(VxReal value);
VxWinch::getMaxDistance() const;
```

The max forces for maintaining the distance is set using

```
VxWinch::setMaxTorque(VxReal value);
VxWinch::getMaxTorque() const;
VxWinch::setMinTorque(VxReal value);
VxWinch::getMinTorque() const;
```

By default, the min torque is 0: no force is added if the distance between part1 and part2 attachment positions is within the desired distance.

By default, the angular velocity around part0's primary axis is used. Linear velocity can be used instead by calling

```
VxWinch::setMotionAngular(false);
```

By default, the desired distance is updated from the driver part's velocity. It is possible to call

```
VxWinch::setPositional(true);
VxWinch::resetPositions();
```

so that the driver position will be used instead. This can be used to prevent slip which happen with velocity-based constraint with a loss parameter > 0. The method resetPositions() sets internal reference positions to match the current winch distance.

VxWheelConstraint

The VxWheelConstraint has 4 controlled coordinates: one for vertical, one for lateral, one for steering and one for the wheel rotation along its axis. Each coordinate can be locked, motorized or limited. The longitudinal, lateral, yaw, roll and pitch displacements of the wheel can be constrained as functions of vertical displacement of the wheel described by tables. Using this constraint to attach a wheel to a chassis allows the user to simulate numerous types of chassis-wheel constraints.

VxConeConstraint

The cone constraint is a limit constraint which can be combined with a VxBallAndSocket for example which has no limits. It will prevent the angle between the connected parts' primary axes to exceed the cone constraint angle. The cone constrain attachment position should match the ball and socket ones.

VxAngular1Position3

The VxAngular1Position3 is a general purpose constraint which allows full control of the rotation around the part's primary axis as well as translation along the 3 axes. It is similar to a hinge with full position control.

As the relative rotation between the part change, the reference axis used for translation can be taken from the first or second part. The reference part can be specified using:

```
VxConstraint::setCoordinateReferencePartIndex(CoordinateID coordinate,
                                              int partReferenceIndex);
```

VxAngular2Position3

The VxAngular2Position3 is a general purpose constraint which allows full control of the rotation around the first part's primary axis and second part's secondary axis as well as translation along the 3 axes. It is similar to a universal constraint with full position control

As the relative rotation between the part change, the reference axis used for translation can be taken from the first or second part. The reference part can be specified using:

```
VxConstraint::setCoordinateReferencePartIndex(CoordinateID coordinate,
                                              int partReferenceIndex);
```

VxMotorConstraint

The VxMotorConstraint constraint is a simple coordinate constraint. It is like a VxHinge or a VxPrismatic without the coordinate equations. The coordinate is angular by default but can be made linear

```
    void VxMotorConstraint::setCoordinateAngular(CoordinateID coordinate, bool b);
```

Note:

The coordinate position evaluation is related to the parts primary axes and a minimum of correlation between these axes is required for computing it accurately. This constraint is normally combined with other constraints.

VxDoubleWinch

This is a 2 parts constraint simulating the dynamics between 2 pulleys.

Given two pulleys of radius r0 and r1, rotating around the part attachment primary axis at the part attachment position. The constraint computes the two line attachment positions and addd a distance constraint along this position. As the line attachment may be going above of below the pulley, the secondary axis should be used as a hint on the which side of the pulley to choose from.

The distance constraint rest length corresponds to the initial distance between the line attachment and is updated according to the pulley's rotation as well as pulley's rotation around each other. Slack may accumulate.

It is possible to specify minimum and maximum line length as well as a line elongation velocity.

```
void VxDoubleWinch::setMinLineLength(VxReal value);
void VxDoubleWinch::setMaxLineLength(VxReal value);
void VxDoubleWinch::setLineLengthVelocity(VxReal value);
```

If a pulley's radius is set to 0, the line attachment is identical to the part attachment. If the two radii are 0 the constraint acts as a distance constraint.

The two pulleys rotation axes don't have to be parallel but they cannot be on the same line (as in real world). If the line is pulled along the pulley's toration axis, the line length will increase as if it unroll from the pulley.

The computed contact position on the pulley during last step can be retreived using

```
VxVector3 VxDoubleWinch::getLineAttachment(int index);
```

VxSumDistance

VxSumDistance is a 2 to 6 parts velocity-based constraint constraint that maintains the sum of the attachment distance to a constant value. The first and last part are rigidly connected while all parts in between are allowed to slide on the line as a pearl on a string.

For the moment, a given part can only appears once in the part list. With only two parts, the constraint acts as a velocity-based distance constraint. When computing the segment count, count will stop when encountering one of the following:

- if two consecutive non-dynamics parts are encoutered
- if GetLineCount(segmentIndex) == 0
- max part count was reached

As a NULL part is allowed to pin a segment to a 3d position in the world, use

```
VxSumDistance::setLineCount(eSegmentIndex segment, VxReal count);
```

with count = 0 to stop the segments.

The total current distance between the segments is set using

```
void VxSumDistance::setDistance(distance);
```

It is possible to pin a point on a segment to a 3d position in the world using

```
void VxSumDistance::addSegmentMidPoint(int segment, const VxVector3& pos);
```

The segment length is then forced to pass through the mid point position. The segment length becomes distance(left_Attachment, mid_point) + distance(right_Attachment, mid_point).

The constraint is currently limited to have its attachment at the center of mass of the parts it is connected to.

VxContactGear

The VxContactGear is a position-based constraint simulating the contact between two gears teeth.

The contraints takes two parts. The two gears radius are provided as well as a backlash distance. Radius = 0 may be used to simulate a linear gear. In this case the primary axis of the part is along the gear otherwise, for circular gear, the primary axis is the rotation axis.

The constraint adds no force if the distance between the parts exceeds the sum of the radius (gears are disengaged). The max force may be tuned using the setMaxForceForward and setMaxForceReverse methods to simulate soft teeth.

```
void VxContactGear::setMaxForceForward(VxReal value);
void VxContactGear::setMaxForceReverse(VxReal value);
```

Initial teeth position within the backlash is specified using

```
void VxContactGear::setOffsetPosition(VxReal value);
```

Finally, the last position of contact between the two gears teeth may me read using

```
const VxVector3& VxContactGear::getLastContactPosition() const;
```

All Classes Namespaces Files Functions Variables Typedefs Enumerations Friends

---