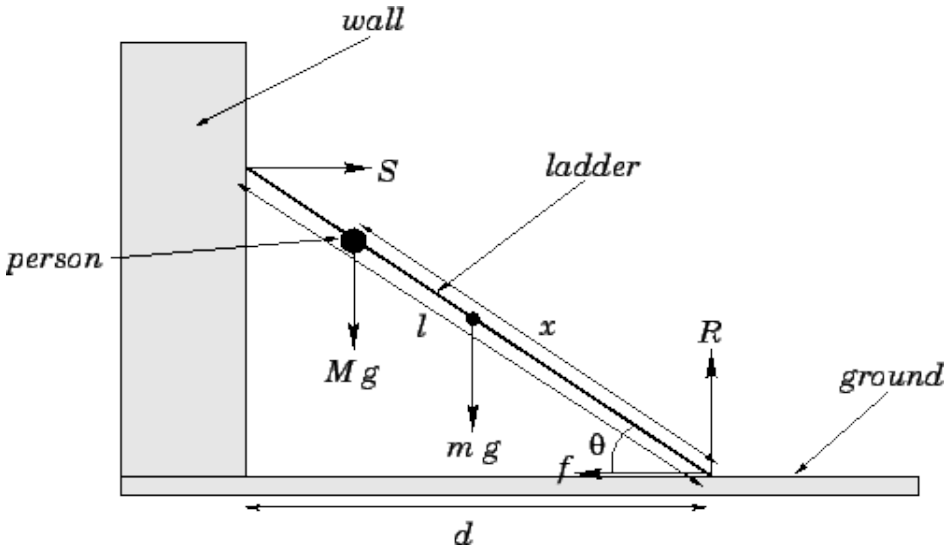
Physics Tutorial 3: Constraints



Summary

In this lecture we explore the concepts of Constraints. First, we discuss some simple physical interaction examples, to frame the concept of a Constraint. We introduce the Constraint as a general basis for physical interactions. We explore the possibility of driven Constraints such as motors and finally we discuss the calculation of the value for λ.

New Concepts

Sphere-Sphere Interfaces, Axis-Aligned Bounding Box interfaces, Sphere-Plane Interfaces, Application of Constraints in 1 Dimension, Application of Constraints in 3 Dimensions, Adding Energy to the System, Calculation of Lambda

Introduction

In the previous lecture we introduced the idea of numerical integration as a means of governing movement within our system. In this lecture, we explore how the variables which inform our calculus can be derived through a concept known as a *constraint*.

To start with, we'll discuss some simplistic physical interactions. This helps get the idea into our heads of how a physical system can be constrained. We'll then undertake a mathematical exploration of the 1D case of a constraint, which is the simplest case, before moving on to a more detailed consideration of the 3D case (which is what our physics engine is intended to solve).

In the real world, it is fair to assume that constraints cannot inject energy into a system - energy can be neither created nor destroyed. But this isn't the real world, and we do have the option of designing a constraint which adds or removes energy from the system. In fact, due to cumulative errors (discussed in lecture 2), we don't really have an option at all - we have to do it. We'll discuss this today, also.

After that, we'll consider the mathematics behind the calculation of λ. Understanding this helps greatly with the understanding of a constraint-based physics engine. After discussing the mathematics, we'll go through a worked example, making reference to the functions provided in the downloadable framework.

Simple Interactions

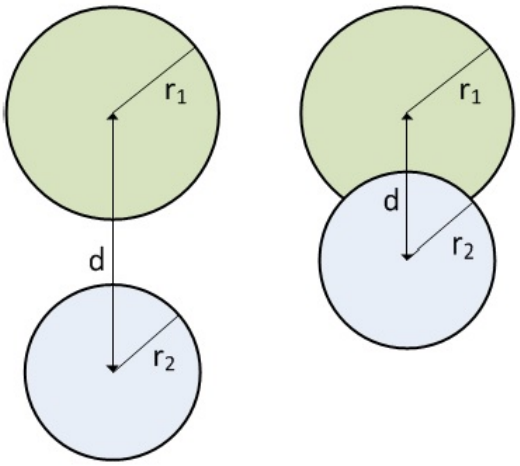
Previously, we asserted that one of the core functions of a physics engine was to detect collisions - more appropriately, interfaces - of items within our environment. Axiomatically, the simpler an item's shape, the fewer computations are required to determine if it has collided with another object.

As such, we will often simplify the shape of our items when we consider interface checks. This is especially true if we are using an interface check for a purpose other than physics. An aggro-range check in an MMOG, for example, is simply a check to see if the player avatar's location has interfaced with an NPC's region of awareness (ususally a sphere or circle).

Some fairly simple examples of interface detection are discussed in this tutorial, before we tie them into the idea of constraints.

Sphere-Sphere Interactions

The simplest approach to interface detection is to represent each object as a sphere centred on the object's position vector, and to calculate whether the two spheres intersect.



The algorithm to detect whether two spheres intersect is very straightforward. If the distance between the centres of the two spheres is less than the sum of the radii of the two spheres, then an intersection has occurred: As the simulation knows the location of the centre of the spheres, we use Pythagoras' theorem to calculate the distance between them, and compare the results to the sum of the radii. So an intersection has occurred if

$$d < r_1 + r_2$$

where

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

However, a square root is an expensive thing to compute, so usually the comparison will be between the square of d and the square of the sum of the radii. So an intersection has occurred if:

$$d^2 < (r_1 + r_2)^2$$

where

$$d^2 = (x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2$$

The collision response data is also straightforward to calculate; indeed most of the work has already been done in the detection algorithm. The contact point P is on the vector which connects the centre of the two spheres, which is also the normal vector N, while the penetration distance p is simply the difference between the sum of the radii and the distance between the sphere centres (S_1 and S_2).

$$p = r_1 + r_2 - d$$

$$N = |S_1 - S_2|$$

$$P = S_1 - N(r_1 - p)$$

In C++, the sphere-sphere collision test might be written:

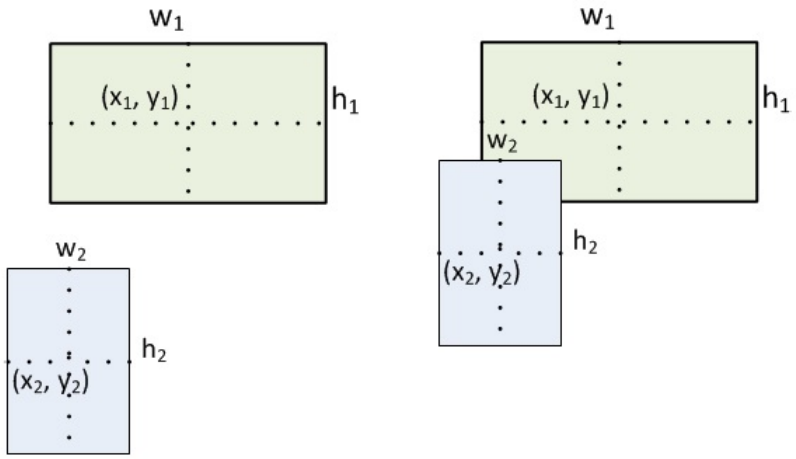
```
bool SphereSphereInterface (PhysicsObject* obj1,
   PhysicsObject* obj2, CollisionShape* shape1,
   CollisionShape* shape2)
   {
      // Check that both shapes are Spheres

      SphereCollisionShape* sphere1 =
         dynamic_cast <const SphereCollisionShape*>(shape1);
      SphereCollisionShape* sphere2 =
         dynamic_cast <const SphereCollisionShape*>(shape2);

      if (sphere1 == NULL || sphere2 == NULL)
      {
         return false ; //At least one object isn't a sphere!
      }

      // Sphere-Sphere Check
      float sum_radius = sphere1->GetRadius() + sphere2->GetRadius();
      float sum_radius_squared = sum_radius * sum_radius ;

      Vector3 ab = obj2 -> GetPosition () - obj1 -> GetPosition ();
      float distance_squared = Vector3::Dot(ab,ab);

      return (distance_squared <= sum_radius_squared);
      // True if distance between centre points is less or equal
      // to the sum of the two radii
   }
```

Sphere–Sphere Interface

The collision detection algorithm does not use the square root; it compares the squares of the two distances which is much faster. It is only after a collision has been detected that, depending on our approach to collision resolution (discussed in a future tutorial) we may need to call on the more computationally expensive square root. Most calls to the interface detection routine will result in a false result (i.e. no intersection has occurred); it is only the rare case of a positive result (i.e. an intersection) which then triggers the more expensive collision data calculation.

Axis-Aligned Bounding Box

The axis-aligned bounding box (AABB) method is also straightforward. It is typically used as a high level collision test to decide whether it is worthwhile continuing with a more complex test, or to trigger a piece of game logic. Each simulated object is represented as a bounding box aligned with the axes of the world, so each collision object has a position, as well as a height, width and length.



The axes are considered in turn, and if there is an overlap of all three axes then an intersection has occurred. An overlap along a particular axis has happened if the distance between the centres of the two boxes on that axis is less than half the sum of the boxes' lengths along that axis. So an intersection has occurred if all three of these conditions is met:

$$|x_2 - x_1| < \frac{1}{2}(w_1 + w_2)$$

$$|y_2 - y_1| < \frac{1}{2}(h_1 + h_2)$$

$$|z_2 - z_1| < \frac{1}{2}(l_1 + l_2)$$

Importantly, as soon as one of these checks fails, the algorithm can bail out as there can't possibly be an intersection unless there is overlap in all three axes.

This is a very cheap collision detection algorithm, as the mathematics is very straightforward (only additions and subtractions). However it is very limited – in particular the bounding boxes need to be axis-aligned, so they can not rotate as the object they represent moves around the world. Also the collision response data is not generated by the algorithm. Consequently this algorithm tends to be used only when we need a quick binary decision on whether a collision is likely, before moving on to a more complex collision detection algorithm, or making a high-level game logic decision, such as detecting when the player has entered a new region of the world, or some game logic needs to be triggered by an invisible bounding box.

Sphere-Plane Collision

Surfaces within the environment are most efficiently simulated as planes in the physics engine. Hence a simple physics simulation of a game would entail representing the game objects as spheres, and the surfaces of the environment (floors, walls, etc) as planes. We therefore require a method for detecting when a sphere has intersected a plane. You will recall from the tutorial on frustum culling, that the plane equation can be used to calculate how far a point is from an infinite plane. Obviously if this distance is less than the radius of a sphere, then the sphere intersects the plane.

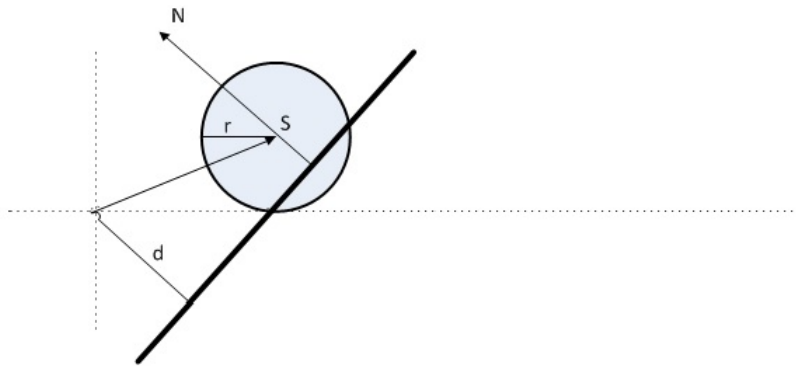
The plane equation is:

$$Ax + By + Cz + D = 0$$

where (A, B, C) is the normal to the plane, D is the distance of the plane from the origin, and (x, y, z) is the position of the test point.

Consequently, a sphere at position S of radius r, intersects a plane with normal N at distance d from the origin if

$$N.S - d < r$$



where $N.S$ is the dot product of N and S. Note that N is a normal and therefore must be of length 1 (i.e. a unit vector), whereas S is simply the position of the centre of the sphere (ie the vector from the origin to the sphere's centre) and therefore not a unit vector.

The penetration p is simply the difference between the radius and the distance between the sphere centre and the plane. The collision normal is the normal of the plane. The contact point P is calculated by taking the sphere position, and adding a vector along the direction of the normal equal to the distance between the sphere centre and the plane. Mathematically:

$$p = r - (N.S - d)$$

$$P = S - N(r - p)$$

The plane equation is based on testing for intersection with an infinite plane. Of course, even a simple game environment can't be represented exclusively by infinite planes - but every object in a game environment can be represented by a collection of finite planes.

It is important to understand plane intersection, as it lays the foundations for the advanced, narrow phase checks we will explore in the next couple of tutorials. The algorithms discussed there will give you some insight into the complexities involved in handling physics for irregular objects, and hopefully encourage you to explore the domain further.

```
1 bool SpherePlaneInterface (Vector3 &position, float radius)
2 {
3    if(Vector3::Dot(position,normal) + distance <= -radius)
4    {
5       return false ;
6    }
7    return true ;
8 }
```

Sphere–Plane Interface

Relevance

These interface detection algoriths are all related to a constraint upon our system, if we assume that a positive result (i.e., an intersection) should be impossible. If two balls should not be able to occupy the same space at the same time within our physical simulation, a constraint upon the system is that distance d between their centres should always be equal to or greater than the sum of their radii.

As such, when we are resolving motion within our system, rather than constraints telling us what must happen, they tell us what mustn't happen, and everything else is assumed to come out in the wash so long as our calculus is accurate. We should remember, we don't need our simulation to be accurate, it just needs to be accurate *enough* to be believable.

Constraints

So having discussed some very specific examples of constraints, let us consider the constraint as a more general concept.

Simple 1D problem

Consider a simple 1D example. Suppose we restrict ourselves to a single axis and we define the constraint:

$$\mathrm{C}(x, y) = \tfrac{1}{2}((x - y)^2 - L^2)$$

This constraint will maintain a constant distance between the variables x and y of length L. To resolve our constraint, we need to find the Jacobian: this is the key to maintaining the distance constraint between x and y. We find the Jacobian of this constraint by applying the multivariate chain rule, which in this case is of the form:

$$\frac{\mathrm{dC}}{\mathrm{d}t} = \frac{\partial \mathrm{C}}{\partial x}\frac{\mathrm{d}x}{\mathrm{d}t} + \frac{\partial \mathrm{C}}{\partial y}\frac{\mathrm{d}y}{\mathrm{d}t}$$

Two factors of this expression are elements of the vector $\mathbf{V}$, namely ${}^{\mathrm{d}x}/_{\mathrm{d}t} = \dot{x}$ and ${}^{\mathrm{d}y}/_{\mathrm{d}t} = \dot{y}$. The other two factors can be calculated as:

$$\frac{\partial \mathrm{C}}{\partial x} = x - y$$
$$\frac{\partial \mathrm{C}}{\partial y} = (-1)(x - y)$$

Replacing these factors in the original expression we get:

$$\frac{\mathrm{dC}}{\mathrm{d}t} = (x - y)\frac{\mathrm{d}x}{\mathrm{d}t} + (-1)(x - y)\frac{\mathrm{d}y}{\mathrm{d}t}$$

or in dot notation this is written:

$$\dot{\mathrm{C}} = (x - y)\dot{x} + (-1)(x - y)\dot{y}$$

Now by comparing coefficients of $\dot{x}$ and $\dot{y}$ we can determine the Jacobian. In this example we only consider two variables and the equation we are comparing against is of the form:

$$\dot{\mathrm{C}} = \mathbf{j}_1\dot{x} + \mathbf{j}_2\dot{y} = \begin{pmatrix}\mathbf{j}_1 & \mathbf{j}_2\end{pmatrix}\begin{pmatrix}\dot{x}\\ \dot{y}\end{pmatrix}$$

This means that the Jacobian in this example is of the form:

$$\mathbf{J} = \begin{pmatrix}x - y & y - x\end{pmatrix}$$

The constraint force will be of the form:

$$\mathbf{F} = \begin{pmatrix}x - y\\ y - x\end{pmatrix}\lambda$$

for some value of λ. Notice that this force is proportional to the distance between the two points x and y but equal and opposite for each object. This means that Newtons laws of motion are satisfied. The justification for this will be explored in a future lecture. Later we will see that the constant λ is dependent on velocity and therefore a constant force will not be applied to the two objects.

Full 3D problem

Now that we have gone through the 1D example we can generalize this to 3D. Figure 1 shows a diagram of two boxes connected at points $\mathbf{x}_1$ attached to box 1 and $\mathbf{x}_2$ attached to box 2. We also highlight the relative vectors $\mathbf{r}_1$ and $\mathbf{r}_2$. These two vectors are the relative positions of $\mathbf{x}_1$ and $\mathbf{x}_2$ with respect to the centre's of gravity $\mathbf{p}_1, \mathbf{p}_2$ of their own boxes.

The two relative vectors are useful as substitutions. By using the substitution $\mathbf{x}_1 = \mathbf{p}_1 + \mathbf{r}_1$ and $\mathbf{x}_2 = \mathbf{p}_2 + \mathbf{r}_2$ we can simplify the portions of the maths which depend on rotating vectors with their quaternion representing orientation. We can calculate the derivatives of $\mathbf{x}_1, \mathbf{x}_2$ to be:

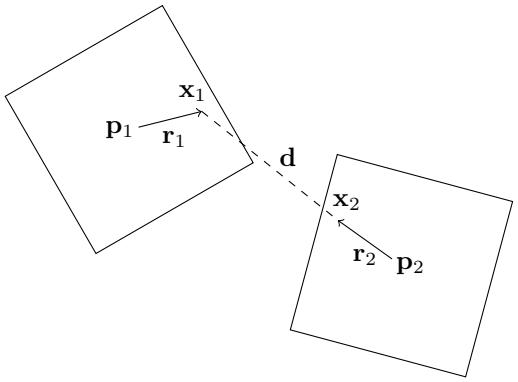


Figure 1: Two boxes connected by a distance constraint at $\mathbf{x}_1$ and $\mathbf{x}_2$.

$$\frac{\mathrm{d}\mathbf{x}_1}{\mathrm{d}t} = \mathbf{v}_1 + \omega_1 \times \mathbf{r}_1$$
$$\frac{\mathrm{d}\mathbf{x}_2}{\mathrm{d}t} = \mathbf{v}_2 + \omega_2 \times \mathbf{r}_2$$

We define the distance constraint C as:

$$\mathrm{C}(\mathbf{x}_1, \mathbf{q}_1, \mathbf{x}_2, \mathbf{q}_2) = \tfrac{1}{2}((\mathbf{x}_2 - \mathbf{x}_1)^2 - L^2)$$

With this expression for the constraint, as was in the last example, the expression is zero when the two points $\mathbf{x}_1$ and $\mathbf{x}_2$ are a distance L apart. We could apply a square root to each term of the equation to give the constraint the true euclidean distance between the two points, however the current form is much easier to calculate the derivative of. When we differentiate with respect to time and use the derivatives we discussed earlier we get:

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

The entire expression is dotted with the vector $\mathbf{d} = \mathbf{x}_2 - \mathbf{x}_1$. We can use the vector $\mathbf{d}$ to simplify the expression. We can also use a vector identity $A \cdot B \times C = C \cdot A \times B$ to rearrange the equation such that:

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

We now have the constraint equation in a form where we can read off the coefficients of $\mathbf{v}_1$, ω_1, $\mathbf{v}_2$ and ω_2 by comparing with:

$$\dot{\mathrm{C}} = \mathbf{j}_1 \cdot \mathbf{v}_1 + \mathbf{j}_2 \cdot \omega_1 + \mathbf{j}_3 \cdot \mathbf{v}_2 + \mathbf{j}_4 \cdot \omega_2$$

We now see that we can rewrite this as:

$$\dot{\mathrm{C}} = \mathbf{JV} = \begin{pmatrix} -\mathbf{d}^T & -(\mathbf{r}_1 \times \mathbf{d})^T & \mathbf{d}^T & (\mathbf{r}_2 \times \mathbf{d})^T \end{pmatrix} \begin{pmatrix} \mathbf{v}_1 \\ \omega_1 \\ \mathbf{v}_2 \\ \omega_2 \end{pmatrix}$$

and therefore our value for the Jacobian $\mathbf{J}$ is:

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

Adding Energy & Motors

So far we have restricted Constraints such that they cannot add energy to the system. A simple modification to the constraint formulation allows us to introduce simple or repetitive motion to the system. We introduce a vector function ζ and set it equal to $\dot{\mathrm{C}}$ giving:

$$\dot{\mathrm{C}} = \mathbf{JV} = \zeta$$

Using this formulation we can simulate a simple motor set up. The vector ζ is called a bias vector and can depend on position, angle and time. We can also use the bias vector to introduce and remove energy from the system allowing us to correct errors which enter the system over time.

We can introduce a correction factor with the equation:

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

the corrective factor of $-\beta$C will push objects back towards zero as time passes in the simulation as long as the constant $\beta > 0$. We will discuss this corrective factor in more detail in lecture 10.

Forces & λ

In this lecture, we have introduced the force vector $\mathbf{F}$ in terms of the Jacobian $\mathbf{J}$ and a constant λ:

$$\mathbf{F} = \mathbf{J}^T \lambda$$

We wish to calculate a value of λ which will correctly balance forces needed to fulfil the constraint C. We start by considering Newtons second law of motion $f = ma$. When applied to this system we get a vector equivalent where:

$$\mathbf{F} = \mathrm{M}\dot{\mathbf{V}}$$

In this equation $\dot{\mathbf{V}}$ is the acceleration equivalent of $\mathbf{V}$ and M is a matrix which defines the distribution of mass throughout the entire system:

$$\mathrm{M} = \begin{pmatrix} m_1 & 0 & 0 & 0 \\ 0 & \mathrm{I}_1 & 0 & 0 \\ 0 & 0 & m_2 & 0 \\ 0 & 0 & 0 & \mathrm{I}_2 \end{pmatrix}$$

where m_1 and m_2 are the masses for objects 1 and 2, and I_1 and I_2 are the inertial tensors for objects 1 and 2. Note that the inertial tensors occupy several rows and columns of the matrix.

Now consider a small change in the velocity vector $\mathbf{V}$. We can numerically approximate the acceleration vector $\dot{\mathbf{V}}$ by the change in $\mathbf{V}$ over a time step Δt:

$$\dot{\mathbf{V}} \approx \frac{\mathbf{V}_2 - \mathbf{V}_1}{\Delta t}$$

Multiplying by the mass matrix M we can re-write this as:

$$\mathrm{M}\dot{\mathbf{V}} = \mathbf{F} = \mathbf{J}^T\lambda = \tfrac{1}{\Delta t}\mathrm{M}(\mathbf{V}_2 - \mathbf{V}_1)$$

Since the matrix M is made up of invertible parts (m_1, m_2, I_1 and I_2) that means that M is invertible and we can calculate M^{-1}. If we multiply by $\mathbf{J}\mathrm{M}^{-1}$ and use the equation $\mathbf{JV} = \zeta$ to replace the $\mathbf{JV}_2$ term we get:

$$\mathbf{J}\mathrm{M}^{-1}\mathbf{J}^T\lambda = \frac{\zeta - \mathbf{JV}}{\Delta t}$$

Which can be re-arranged to get an equation for λ:

$$\lambda = \frac{\zeta - \mathbf{JV}}{\mathbf{J}\mathrm{M}^{-1}\mathbf{J}^T\Delta t}$$

It should be noted that this is actually slightly simplified. There are more exhaustive explorations of the mathematics which underpin the calculation of λ available online, but this satisfies our purposes. We now move on to a worked example, outlining how we go about solving this in the physics framework.

Practical Computation of Velocity Update

Consider the example in Figure 2. Two objects, A and B, are connected by a distance constraint (the dashed line) along direction x. pA and pB are their respective centres, and rA and rB are the vectors connecting those centres to the point on their surfaces where the constraint exists, indicated by grey triangles.

Figure 2: Example Constraint

We declare a class DistanceConstraint, which inherits from constraint. We'll work our way through aspects of this class, at each stage linking back to the diagram, so as to illustrate the computations involved. Naturally, the class is illustrative, and you are free to implement your own interpretation of the distance constraint; the complete code for DistanceConstraint.h is included at the end of the tutorial.

The Jacobian

```
1  Vector3 ab = globalOnB - globalOnA;
2  Vector3 abn = ab;
3  abn.Normalise();
4
5  Vector3 r1 = (globalOnA - objA->GetPosition());
6  Vector3 r2 = (globalOnB - objB->GetPosition());
7
8  this->j1 = -abn;
9  this->j2 = Vector3::Cross(-r1, abn);
10 this->j3 = abn;
11 this->j4 = Vector3::Cross(r2, abn);
12 this->b = 0.0f;
13
14 this->distance = ab.Length();
```

Jacobian

This portion of the code is computing the Jacobian. Linking the variables back to our diagram, pA and pB are the position properties of the objects. The grey triangles are defined as `globalOnA` and `globalOnB` for objects A and B respectively, and used to compute `r1` and `r2`.

The `Vector3` which defines the constraint (the dashed line on our diagram) is denoted `ab`, and the normalised vector defining its direction, x, is stored as `abn`. The next few lines define the elements of our Jacobian.

We recall that the Jacobian takes the form $(\mathbf{v_1}, \omega_1, \mathbf{v_2}, \omega_2)$. The first and third elements contain the linear components of our Jacobian. As the direction of the constraint is from A to B, the linear

component of the Jacobian with respect to A is the negative of that direction (`-abn`, or $-x$); the linear component with respect to B, conversely, is `abn`, or x. Directionality of the constraint also determines the sign of the cross product which defines the second and fourth (angular) elements of the Jacobian.

We can overlook `b` for now - this relates to the Baumgarte constant, which you'll experiment with as part of your practical tasks for today. The `distance` between the points of connection is merely the length of the vector (the dashed line).

Constraint Mass

From this point on, we discuss all physical parameters relating to the constraint (mass, velocity, etc.) in terms of their magnitude along the Jacobian directions (`j1`, `j2`, `j3`, `j4`). The next step in the process is to compute the Constraint Mass of our constraint. We define this as a `float` (a scalar), and compute it using the function below:

```
1 float constraint_mass = objA->GetInverseMass() * Vector3::Dot(j1, j1)
2    + Vector3::Dot(j2, (objA->GetInverseInertia() * j2))
3    + objB->GetInverseMass() * Vector3::Dot(j3, j3)
4    + Vector3::Dot(j4, (objB->GetInverseInertia() * j4));
```

Constraint Mass

This function employs the inverse mass and inverse inertia of each object in turn. You can see how each maps to the elements of the Jacobian (the inverse mass of Object A to the linear component of the Jacobian with respect to Object A, etc.).

Differentiated Constraint ($\dot{C}$)

We're reminded at $\dot{C}$ is the product of the Jacobian and the Velocity Vector $\mathbf{V}$. We obtain this by summing the dot products of the vector components (remembering that each component is a vector itself!). This value is held in `float jv`.

```
1 float jv = Vector3::Dot(j1, objA->GetLinearVelocity())
2 + Vector3::Dot(j2, objA->GetAngularVelocity())
3 + Vector3::Dot(j3, objB->GetLinearVelocity())
4 + Vector3::Dot(j4, objB->GetAngularVelocity());
```

Cee–Dot

Lambda (λ)

Having gone through the above steps, the computation to obtain λ is relatively simple: we divide `-jv` by the constraint mass. Again, `b` is present, but in this example it is set to `0.0f` and won't impact the simulation. The code is included below:

```
1 float denom = -(jv + b);
2 float lambda = denom / constraint_mass;
```

Lambda

Updating Velocity

And, having gone through all of that, we are now in a position to update the velocities (both linear and angular) of our objects. Again, we employ the elements of the Jacobian, as these elements define the relationship between our constraint and the physical properties of our objects. The result looks something like this:

```
objA->SetLinearVelocity(objA->GetLinearVelocity()
   + (j1 * lambda) * objA->GetInverseMass());
objA->SetAngularVelocity(objA->GetAngularVelocity()
   + objA->GetInverseInertia() * (j2 * lambda));
objB->SetLinearVelocity(objB->GetLinearVelocity()
   + (j3 * lambda) * objB->GetInverseMass());
objB->SetAngularVelocity(objB->GetAngularVelocity()
   + objB->GetInverseInertia() * (j4 * lambda));
```

Update Velocity Pseudocode

And there we have it - a step by step walkthrough of our constraint-based solver.

Implementation

Below you have all the code needed to implement a distance constraint in your engine. Note that this code sample is more simplified than the step-by-step walkthrough above, with the formation of the Jacobian kept abstract. Review and implement the code snippet below, and look at the remaining practical exercises for Day 2 in the Practical Tasks handout.

Summary

We have explored the application of constraints to actual physical problems. We have investigated the addition of energy to our system via constraints, before stepping through a worked example in code of a constraint-based solver relating to distance.

```
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 (Constraint* c : constraints) c->PreSolverStep(updateTimestep);

   // Optional pre-computation step, needed for some constraints

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

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

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

PhysicsEngine.cpp

```
1
2  // DistanceConstraint::ApplyImpulse
3
4  virtual void ApplyImpulse() override
5  {
6     // Compute current constraint vars based on object A/B's
7     // position/rotation
8
9     Vector3 r1 = pnodeA->GetOrientation().ToMatrix3() * relPosA;
10    Vector3 r2 = pnodeB->GetOrientation().ToMatrix3() * relPosB;
11
12    //Get the global contact points in world space
13
14    Vector3 globalOnA = r1 + pnodeA->GetPosition();
15    Vector3 globalOnB = r2 + pnodeB->GetPosition();
16
17    //Get the vector between the two contact points
18
19    Vector3 ab = globalOnB - globalOnA;
20    Vector3 abn = ab;
21
22    abn.Normalise();
23
24    // Compute the velocity of objects A and B at the point of
25    // contact
26
27    Vector3 v0 = pnodeA->GetLinearVelocity()
28       + Vector3::Cross(pnodeA->GetAngularVelocity(), r1);
29
30    Vector3 v1 = pnodeB->GetLinearVelocity()
31       + Vector3::Cross(pnodeB->GetAngularVelocity(), r2);
32
33    //Relative velocity in constraint direction
34    float abnVel = Vector3::Dot(v0 - v1, abn);
35
36    // Compute the 'mass' of the constraint
37    // e.g. How difficult it is to move the two objects in
38    // the direction of the constraint
39
40    float invConstraintMassLin = pnodeA->GetInverseMass()
41       + pnodeB->GetInverseMass();
42
43    float invConstraintMassRot = Vector3::Dot(abn,
44       Vector3::Cross(pnodeA->GetInverseInertia()
45       * Vector3::Cross(r1, abn), r1)
46       + Vector3::Cross(pnodeB->GetInverseInertia()
47       * Vector3::Cross(r2, abn), r2));
48
49    float constraintMass = invConstraintMassLin + invConstraintMassRot;
50
51    if (constraintMass > 0.0f)
52    {
53       // Baumgarte Offset (Adds energy to the system to counter
54       // slight solving errors that accumulate over time - known
55       // as 'constraint drift')
56
57       // Experiment by commenting this out and see how it
58       // affects the constraints over time and when you manually
```

```
59        // move the objects apart.
60
61        // The key is to find a nice value that is small enough
62        // not to cause objects to explode but also enough to make
63        // sure all constraints /will/ be satisfied. This value
64        // (0.1) will change based on your physics objects,
65        // timestep etc., and also how many constraints you are
66        // chaining together.
67
68        float b = 0.0f;
69
70        //-Optional-
71        float distance_offset = ab.Length() - targetLength;
72        float baumgarte_scalar = 0.1f;
73        b = -(baumgarte_scalar
74           / PhysicsEngine::Instance()->GetDeltaTime())
75           * distance_offset;
76
77        //-Eof Optional-
78
79        //Compute velocity impulse (jn)
80        // In order to satisfy the distance constraint we need
81        // to apply forces to ensure the relative velocity
82        // (abnVel) in the direction of the constraint is zero.
83        // So we take inverse of the current rel velocity and
84        // multiply it by how hard it will be to move the objects.
85
86        // Note: We also add in any extra energy to the system
87        // here, e.g. baumgarte (and later elasticity)
88
89        float jn = -(abnVel + b) / constraintMass;
90
91        // Apply linear velocity impulse
92
93        pnodeA->SetLinearVelocity(pnodeA->GetLinearVelocity()
94           + abn * (pnodeA->GetInverseMass() * jn));
95
96        pnodeB->SetLinearVelocity(pnodeB->GetLinearVelocity()
97           - abn * (pnodeB->GetInverseMass() * jn));
98
99        //Apply rotational velocity impulse
100
101       pnodeA->SetAngularVelocity(pnodeA->GetAngularVelocity()
102          + pnodeA->GetInverseInertia()
103          * Vector3::Cross(r1, abn * jn));
104
105       pnodeB->SetAngularVelocity(pnodeB->GetAngularVelocity()
106          - pnodeB->GetInverseInertia()
107          * Vector3::Cross(r2, abn * jn));
108    }
109 }
```

DistanceConstraint.h