Advanced Character Physics

Thomas Jakobsen
IO Interactive, Farvergade 2
DK-1463 Copenhagen K
Denmark
Email: tj@ioi.dk, www: www.ioi.dk/~tj

Abstract
This paper explains the basic elements of an approach to physically-based modeling which is well suited for interactive use. It is simple, fast, and quite stable, and in its basic version the method does not require knowledge of advanced mathematical subjects (although it is based on a solid mathematical foundation). It allows for simulation of both cloth; soft and rigid bodies; and even articulated or constrained bodies using both forward and inverse kinematics.

The algorithms were developed for IO Interactive's game Hitman: Codename 47. There, among other things, the physics system was responsible for the movement of cloth, plants, rigid bodies, and for making dead human bodies fall in unique ways depending on where they were hit, fully interacting with the environment (resulting in the press oxymoron "lifelike death animations").

The article also deals with subtleties like penetration test optimization and friction handling.

1 Introduction

The use of physically-based modeling to produce nice-looking animation has been considered for some time and many of the existing techniques are fairly sophisticated. Different approaches have been proposed in the literature [Baraff, Mirtich, Witkin, and others] and much effort has been put into the construction of algorithms that are accurate and reliable. Actually, precise simulation methods for physics and dynamics have been known for quite some time from engineering. However, for games and interactive use, accuracy is really not the primary concern (although it's certainly nice to have) – rather, here the important goals are *believability* (the programmer can cheat as much as he wants if the player still feels immersed) and *speed of execution* (only a certain time per frame will be allocated to the physics engine). In the case of physics simulation, the word believability also covers stability; a method is no good if objects seem to drift through obstacles or vibrate when they should be lying still, or if cloth particles tend to "blow up".

The methods demonstrated in this paper were created in an attempt to reach these goals. The algorithms were developed and implemented by the author for use in IO Interactive's computer game Hitman: Codename 47, and have all been integrated in IO's in-house game engine Glacier. The methods proved to be quite simple to implement (compared to other schemes at least) and have high performance.

The algorithm is iterative such that, from a certain point, it can be stopped at any time. This gives us a very useful time/accuracy trade-off: If a small source of inaccuracy is accepted, the code can be allowed to run faster; this error margin can even be adjusted adaptively at run-time. In some cases, the method is as much as an order of magnitude faster than other existing methods. It also handles both collision and resting contact in the same framework and nicely copes with stacked boxes and other situations that stress a physics engine.

In overview, the success of the method comes from the right combination of several techniques that all benefit from each other:

- A so-called Verlet integration scheme.
- Handling collisions and penetrations by projection.
- A simple constraint solver using relaxation.
- A nice square root approximation that gives a solid speed-up.
- Modeling rigid bodies as particles with constraints.
- An optimized collision engine with the ability to calculate penetration depths.

Each of the above subjects will be explained shortly. In writing this document, the author has tried to make it accessible to the widest possible audience without losing vital information necessary for implementation. This means that technical mathematical explanations and notions are kept to a minimum if not crucial to understanding the subject. The goal is demonstrating the possibility of implementing quite advanced and stable physics simulations without dealing with loads of mathematical intricacies.

The content is organized as follows. First, in Section 2, a "velocity-less" representation of a particle system will be described. It has several advantages, stability most notably and the fact that constraints are simple to implement. Section 3 describes how collision handling takes place. Then, in Section 4, the particle system is extended with constraints allowing us to model cloth. Section 5 explains how to set up a suitably constrained particle system in order to emulate a rigid body. Next, in Section 6, it is demonstrated how to further extend the system to allow articulated bodies (that is, systems of interconnected rigid bodies with angular and other constraints). Section 7 contains various notes and shares some experience on implementing friction etc. Finally, in Section 8 a brief conclusion.

In the following, **bold** typeface indicates vectors. Vector components are indexed by using subscript, i.e., $\mathbf{x}=(x_1, x_2, x_3)$.

2 Verlet integration

The heart of the simulation is a particle system. Typically, in implementations of particle systems, each particle has two main variables: Its position **x** and its velocity **v**. Then in the time-stepping loop, the new position **x**' and velocity **v**' are often computed by applying the rules

$$\mathbf{x}' = \mathbf{x} + \mathbf{v} \cdot \Delta t$$
$$\mathbf{v}' = \mathbf{v} + \mathbf{a} \cdot \Delta t,$$

where Δt is the time step, and **a** is the acceleration computed using Newton's law **f**=m**a** (where **f** is the accumulated force acting on the particle). This is simple Euler integration.

Here, however, we choose a velocity-less representation and another integration scheme: Instead of storing each particle's position and velocity, we store its current position **x** *and* its previous position **x***. Keeping the time step fixed, the update rule (or integration step) is then

$$\mathbf{x}' = 2\mathbf{x} - \mathbf{x}^* + \mathbf{a} \cdot \Delta t^2$$
$$\mathbf{x}^* = \mathbf{x}.$$

This is called Verlet integration (see [Verlet]) and is used intensely when simulating molecular dynamics. It is quite stable since the velocity is implicitly given and consequently it is harder for velocity and position to come out of sync. (As a side note, the well-known demo effect for creating ripples in water uses a similar approach.) It works due to the fact that 2**x**-**x***=**x**+(**x**-**x***) and **x**-**x*** is an approximation of the current velocity (actually, it's the distance traveled last time step). It is not always very accurate (energy might leave the system, i.e., dissipate) but it's fast and stable. By lowering the value 2 to something like 1.99 a small amount of drag can also be introduced to the system.

At the end of each step, for each particle the current position **x** gets stored in the corresponding variable **x***. Note that when manipulating many particles, a useful optimization is possible by simply swapping array pointers.

The resulting code would look something like this (the Vector3 class should contain the appropriate member functions and overloaded operators for manipulation of vectors):

```
// Sample code for physics simulation
class ParticleSystem {
      Vector3     m_x[NUM_PARTICLES]; // Current positions
      Vector3     m_oldx[NUM_PARTICLES]; // Previous positions
      Vector3     m_a[NUM_PARTICLES]; // Force accumulators
      Vector3     m_vGravity; // Gravity
      float       m_fTimeStep;
public:
      void        TimeStep();
private:
      void        Verlet();
      void        SatisfyConstraints();
      void        AccumulateForces();
//    (constructors, initialization etc. omitted)
};
// Verlet integration step
void ParticleSystem::Verlet() {
      for(int i=0; i<NUM_PARTICLES; i++) {
            Vector3& x = m_x[i];
            Vector3 temp = x;
```

```
            Vector3& oldx = m_oldx[i];
            Vector3& a = m_a[i];
            x += x-oldx+a*fTimeStep*fTimeStep;
            oldx = temp;
      }
}
// This function should accumulate forces for each particle
void ParticleSystem::AccumulateForces()
{
// All particles are influenced by gravity
      for(int i=0; i<NUM_PARTICLES; i++)  m_a[i] = m_vGravity;
}
// Here constraints should be satisfied
void ParticleSystem::SatisfyConstraints() {
// Ignore this function for now
}
void ParticleSystem::TimeStep() {
      AccumulateForces();
      Verlet();
      SatisfyConstraints();
}
```

The above code has been written for clarity, not speed. One optimization would be using arrays of float instead of Vector3 for the state representation. This might also make it easier to implement the system on a vector processor.

This probably doesn't sound very groundbreaking yet. However, the advantages should become clear soon when we begin to use constraints and switch to rigid bodies. It will then be demonstrated how the above integration scheme leads to increased stability and a decreased amount of computation when compared to other approaches.

Try setting **a**=(0,0,1), for example, and use the start condition **x**=(1,0,0), **x***=(0,0,0), then do a couple of iterations by hand and see what happens.

3 Collision and contact handling by projection

So-called penalty-based schemes handle contact by inserting springs at the penetration points. While this is very simple to implement, it has a number of serious drawbacks. For instance, it is hard to choose suitable spring constants such that, on one hand, objects don't penetrate too much and, on the other hand, the resulting system doesn't get unstable. In other schemes for simulating physics, collisions are handled by rewinding time (by binary search for instance) to the exact point of collision, handling the collision analytically from there and then restarting the simulation – this is not very practical from a real-time point of view since the code could potentially run very slowly when there are a lot of collisions.

Here, we use yet another strategy. Offending points are simply projected out of the obstacle. By projection, loosely speaking, we mean moving the point as little as possible until it is free of the obstacle. Normally, this means moving the point perpendicularly out towards the collision surface.

Let's examine an example. Assume that our world is the inside of the cube (0,0,0)-(1000,1000,1000) and assume also that the particles' restitution

coefficient is zero (that is, particles do not bounce off surfaces when colliding). To keep all positions inside the valid interval, the corresponding projection code would be:

```
// Implements particles in a box
void ParticleSystem::SatisfyConstraints() {
      for(int i=0; i<NUM_PARTICLES; i++) { // For all particles
            Vector3& x = m_x[i];
            x = vmin(vmax(x, Vector3(0,0,0)),
                              Vector3(1000,1000,1000));
      }
}
```

(vmax operates on vectors taking the component-wise maximum whereas vmin takes the component-wise minimum.) This keeps all particle positions inside the cube and handles both collisions and resting contact. The beauty of the Verlet integration scheme is that the corresponding changes in velocity will be handled automatically. In the following calls to TimeStep(), the velocity is automatically regulated to contain no component in the normal direction of the surface (corresponding to a restitution coefficient of zero). See Figure 1.

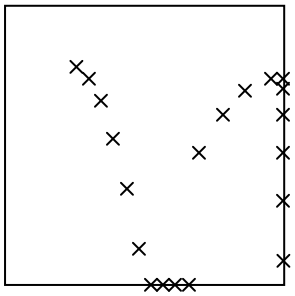


Figure 1. Ten timesteps and two particles.

Try it out – there is no need to directly cancel the velocity in the normal direction. While the above might seem somewhat trivial when looking at particles, the strength of the Verlet integration scheme is now beginning to shine through and should really become apparent when introducing constraints and coupled rigid bodies in a moment.

4 Solving several concurrent constraints by relaxation

A common model for cloth consists of a simple system of interconnected springs and particles. However, it is not always trivial to solve the corresponding system of differential equations. It suffers from some of the same problems as the penalty-based systems: Strong springs leads to stiff systems of equations that lead to instability if only simple integration techniques are used, or at least bad performance – which leads to pain. Conversely, weak springs lead to elastically looking cloth.

However, an interesting thing happens if we let the stiffness of the springs go to infinity: The system suddenly becomes solvable in a stable way with a very simple and fast approach. But before we continue talking about cloth, let's revisit the previous example. The cube considered above can be thought of as a collection of unilateral (inequality) constraints (one for each side of the cube) on the particle positions that should be satisfied at all times:

$$x_i \geq 0 \text{ and } x_i \leq 1000 \text{ for } i = 1,2,3. \quad \text{(C1)}$$

In the example, constraints were satisfied (that is, particles are kept inside the cube) by simply modifying offending positions by projecting the particles onto the cube surface. To satisfy (C1), we use the following pseudo-code

```
// Pseudo-code to satisfy (C1)
for i=1,2,3
      set x_i=min{max{x_i, 0}, 1000}
```

One may think of this process as inserting infinitely stiff springs between the particle and the penetration surface – springs that are exactly so strong and suitably damped that instantly they will attain their rest length zero.

We now extend the experiment to model a stick of length 100. We do this by setting up two individual particles (with positions $\mathbf{x1}$ and $\mathbf{x2}$) and then require them to be a distance of 100 apart. Expressed mathematically, we get the following bilateral (equality) constraint:

$$|\mathbf{x2}-\mathbf{x1}|=100. \quad \text{(C2)}$$

Although the particles might be correctly placed initially, after one integration step the separation distance between them might have become invalid. In order to obtain the correct distance once again, we move the particles by projecting them onto the set of solutions described by (C2). This is done by pushing the particles directly away from each other or by pulling them closer together (depending on whether the erroneous distance is too small or too large). See Figure 2.

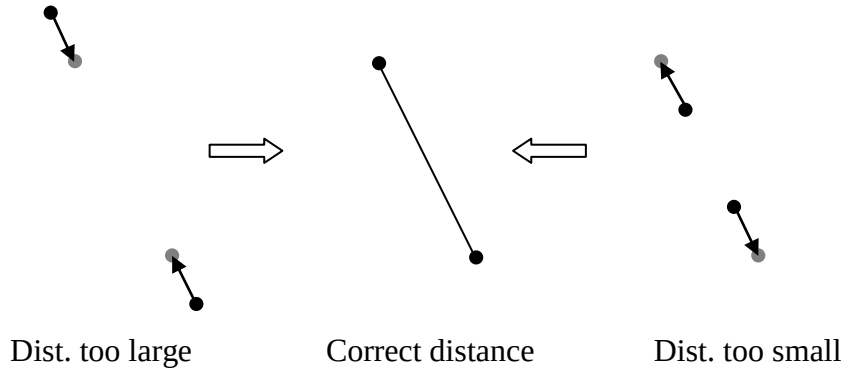


Figure 2. Fixing an invalid distance by moving the particles.

The pseudo-code for satisfying the constraint (C2) is

```
// Pseudo-code to satisfy (C2)
delta = x2-x1;
deltalength = sqrt(delta*delta);
diff = (deltalength-restlength)/deltalength;
x1 -= delta*0.5*diff;
x2 += delta*0.5*diff;
```

Note that delta is a vector so delta*delta is actually a dot product. With restlength=100 the above pseudo-code will push apart or pull together the particles such that they once more attain the correct distance of 100 between

them. Again we may think of the situation as if a very stiff spring with rest length 100 has been inserted between the particles such that they are instantly placed correctly.

Now assume that we still want the particles to satisfy the cube constraints. By satisfying the stick constraint, however, we may have invalidated one or more of the cube constraints by pushing a particle out of the cube. This situation can be remedied by immediately projecting the offending particle position back onto the cube surface once more – but then we end up invalidating the stick constraint again.

Really, what we should do is solve for all constraints at once, both (C1) and (C2). This would be a matter of solving a system of equations. However, we choose to proceed indirectly by local iteration. We simply repeat the two pieces of pseudo-code a number of times after each other in the hope that the result is useful. This yields the following code:

```
// Implements simulation of a stick in a box
void ParticleSystem::SatisfyConstraints() {
      for(int j=0; j<NUM_ITERATIONS; j++) {
            // First satisfy (C1)
            for(int i=0; i<NUM_PARTICLES; i++) { // For all particles
                  Vector3& x = m_x[i];
                  x = vmin(vmax(x, Vector3(0,0,0)),
                              Vector3(1000,1000,1000));
            }

            // Then satisfy (C2)
            Vector3& x1 = m_x[0];
            Vector3& x2 = m_x[1];
            Vector3 delta = x2-x1;
            float deltalength = sqrt(delta*delta);
            float diff = (deltalength-restlength)/deltalength;
            x1 -= delta*0.5*diff;
            x2 += delta*0.5*diff;
      }
}
```

(Initialization of the two particles has been omitted.) While this approach of pure repetition might appear somewhat naïve, it turns out that it actually converges to the solution that we are looking for! The method is called relaxation (or Jacobi or Gauss-Seidel iteration depending on how you do it exactly, see [Press]). It works by consecutively satisfying various local constraints and then repeating; if the conditions are right, this will converge to a global configuration that satisfies all constraints at the same time. It is useful in many other situations where several interdependent constraints have to be satisfied at the same time.

The number of necessary iterations varies depending on the physical system simulated and the amount of motion. It can be made adaptive by measuring the change from last iteration. If we stop the iterations early, the result might not end up being quite valid but because of the Verlet scheme, in next frame it will probably be better, next frame even more so etc. This means that

stopping early will not ruin everything although the resulting animation might appear somewhat sloppier.

Cloth simulation

The fact that a stick constraint can be thought of as a really hard spring should make apparent its usefulness for cloth simulation as sketched in the beginning of this section. Assume, for example, that a hexagonal mesh of triangles describing the cloth has been constructed. For each vertex a particle is initialized and for each edge a stick constraint between the two corresponding particles is initialized (with the constraint's "rest length" simply being the initial distance between the two vertices).

The function HandleConstraints() then uses relaxation over all constraints. The relaxation loop could be iterated several times. However, to obtain nicely looking animation, actually for most pieces of cloth only one iteration is necessary! This means that the time usage in the cloth simulation depends mostly on the N square root operations and the N divisions performed (where N denotes the number of edges in the cloth mesh). As we shall see, a clever trick makes it possible to reduce this to N divisions per frame update – this is really fast and one might argue that it probably can't get much faster.

```
// Implements cloth simulation
struct Constraint {
      int   particleA, particleB;
      float restlength;
};
// Assume that an array of constraints, m_constraints, exists
void ParticleSystem::SatisfyConstraints() {
      for(int j=0; j<NUM_ITERATIONS; j++) {
            for(int i=0; i<NUM_CONSTRAINTS; i++) {
                  Constraint& c = m_constraints[i];
                  Vector3& x1 = m_x[c.particleA];
                  Vector3& x2 = m_x[c.particleB];
                  Vector3 delta = x2-x1;
                  float deltalength = sqrt(delta*delta);
                  float diff=(deltalength-c.restlength)/deltalength;
                  x1 -= delta*0.5*diff;
                  x2 += delta*0.5*diff;
            }

            // Constrain one particle of the cloth to origo
            m_x[0] = Vector3(0,0,0);
      }
}
```

We now discuss how to get rid of the square root operation. If the constraints are all satisfied (which they should be at least almost), we already know what the result of the square root operation in a particular constraint expression ought to be, namely the rest length r of the corresponding stick. We can use this fact to approximate the square root function. Mathematically, what we do is approximate the square root function by its 1^{st} order Taylor-expansion at a neighborhood of the rest length r (this is equivalent to one Newton-Raphson iteration with initial guess r). After some rewriting, we obtain the following pseudo-code:

```
// Pseudo-code for satisfying (C2) using sqrt approximation
delta = x2-x1;
delta*=restlength*restlength/(delta*delta+restlength*restlength)-0.5;
x1 -= delta;
x2 += delta;
```

Notice that if the distance is already correct (that is, if |delta|=restlength), then one gets delta=(0,0,0) and no change is going to happen.

Per constraint we now use *zero* square roots, one division only, and the squared value restlength*restlength can even be precalculated! The usage of time consuming operations is now down to N divisions per frame (and the corresponding memory accesses) – it can't be done much faster than that and the result even looks quite nice. Actually, in Hitman, the overall speed of the cloth simulation was limited mostly by how many triangles it was possible to push through the rendering system.

The constraints are not guaranteed to be satisfied after one iteration only, but because of the Verlet integration scheme, the system will quickly converge to the correct state over some frames. In fact, using only one iteration and approximating the square root removes the stiffness that appears otherwise when the sticks are perfectly stiff.

By placing support sticks between strategically chosen couples of vertices sharing a neighbor, the cloth algorithm can be extended to simulate plants. Again, in Hitman only one pass through the relaxation loop was enough (in fact, the low number gave the plants exactly the right amount of bending behavior).

The code and the equations covered in this section assume that all particles have identical mass. Of course, it is possible to model particles with different masses, the equations only get a little more complex.

To satisfy (C2) while respecting particle masses, use the following code:

```
// Pseudo-code to satisfy (C2)
delta = x2-x1;
deltalength = sqrt(delta*delta);
diff = (deltalength-restlength)
      /(deltalength*(invmass1+invmass2));
x1 -= invmass1*delta*diff;
x2 += invmass2*delta*diff;
```

Here invmass1 and invmass2 are the numerical inverses of the two masses. If we want a particle to be immovable, simply set invmass=0 for that particle (corresponding to an infinite mass). Of course in the above case, the square root can also be approximated for a speed-up.

5 Rigid bodies

The equations governing motion of rigid bodies were discovered long before the invention of modern computers. To be able to say anything useful at that time, mathematicians needed the ability to manipulate expressions

symbolically. In the theory of rigid bodies, this lead to useful notions and tools such as inertia tensors, angular momentum, torque, quaternions for representing orientations etc. However, with the current ability to process huge amounts of data numerically, it has become feasible and in some cases even advantageous to break down calculations to simpler elements when running a simulation. In the case of 3D rigid bodies, this could mean modeling a rigid body by four particles and six constraints (giving the correct amount of degrees of freedom, 4x3-6 = 6). This simplifies a lot of aspects and it's exactly what we will do in the following.

Consider a tetrahedron and place a particle at each of the four vertices. In addition, for each of the six edges on the tetrahedron create a distance constraint like the stick constraint discussed in the previous section. This is actually enough to simulate a rigid body. The tetrahedron can be let loose inside the cube world from earlier and the Verlet integrator will let it move correctly. The function SatisfyConstraints() should take care of two things: 1) That particles are kept inside the cube (like previously), and 2) That the six distance constraints are satisfied. Again, this can be done using the relaxation approach; 3 or 4 iterations should be enough with optional square root approximation.

Now clearly, in general rigid bodies do not behave like tetrahedrons collision-wise (although they might do so kinetically). There is also another problem: Presently, collision detection between the rigid body and the world exterior is on a vertex-only basis, that is, if a vertex is found to be outside the world it is projected inside again. This works fine as long as the inside of the world is convex. If the world were non-convex then the tetrahedron and the world exterior could actually penetrate each other without any of the tetrahedron vertices being in an illegal region (see Figure 3 where the triangle represents the 2D analogue of the tetrahedron). This problem is handled in the following.

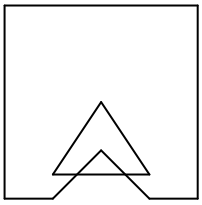


Figure 3. A tetrahedron penetrating the world.

We'll first consider a simpler version of the problem. Consider the stick example from earlier and assume that the world exterior has a small bump on it. The stick can now penetrate the world exterior without any of the two stick particles leaving the world (see Figure 4). We won't go into the intricacies of constructing a collision detection engine since this is a science in itself. Instead we assume that there is a subsystem available which allows us to detect the collision. Furthermore we assume that the subsystem can reveal to us the penetration depth and identify the penetration points on each of the two colliding objects. (One definition of penetration points and penetration depth goes like this: The penetration distance d_p is the shortest distance that would prevent the two objects from penetrating if one were to translate one of the objects by the distance d_p in a suitable direction. The penetration points are the points on each object that just exactly touch the other object after the aforementioned translation has taken place.)

Take a look again at Figure 4. Here the stick has moved through the bump after the Verlet step. The collision engine has identified the two points of penetration, **p** and **q**. In Figure 4a, **p** is actually identical to the position of particle 1, i.e., **p**=**x1**. In Figure 4b, **p** lies between **x1** and **x2** at a position ¼ of the stick length from **x1**. In both cases, the point **p** lies on the stick and consequently it can be expressed as a linear combination of **x1** and **x2**, **p**=c1 **x1**+c2 **x2** such that c1+c2=1. In the first case, c1=1 and c2=0, in the second case, c1=0.75 and c2=0.25. These values tell us how much we should move the corresponding particles.

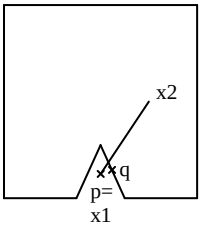


Figure 4a. Colliding stick I.

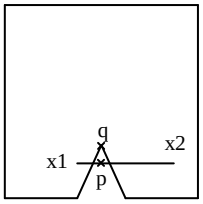


Figure 4b. Colliding stick II.

To fix the invalid configuration of the stick, it should be moved upwards somehow. Our goal is to avoid penetration by moving **p** to the same position as **q**. We do this by adjusting the positions of the two particles **x1** and **x2** in the direction of the vector between **p** and **q**, Δ=**q**-**p**.

In the first case, we simply project **x1** out of the invalid region like earlier (in the direction of **q**) and that's it (**x2** is not touched). In the second case, **p** is still nearest to **x1** and one might reason that consequently **x1** should be moved more than **x2**. Actually, since **p**=0.75 **x1** + 0.25 **x2**, we will choose to move **x1** by an amount of 0.75 each time we move **x2** by an amount of 0.25. In other words, the new particle positions **x1**' and **x2**' are given by the expressions

$$\begin{aligned} \mathbf{x1}' &= \mathbf{x1} + 0.75\lambda \cdot \boldsymbol{\Delta} \\ \mathbf{x2}' &= \mathbf{x2} + 0.25\lambda \cdot \boldsymbol{\Delta} \end{aligned} \qquad (*)$$

where λ is some unknown value. The new position of **p** after moving both particles is **p**'=c1 **x1**'+ c2 **x2**'.

Recall that we want **p**'=**q**, i.e., we should choose λ exactly such that **p**' ends up coinciding with **q**. Since we move the particles only in the direction of Δ, also **p** moves in the direction of Δ and consequently the solution to the equation **p**'=**q** can be found by solving

$$\mathbf{p}' \cdot \boldsymbol{\Delta} = \mathbf{q} \cdot \boldsymbol{\Delta} \quad (**)$$

for λ. Expanding the left-hand side yields:

$$
\begin{aligned}
\mathbf{p}'\cdot\boldsymbol{\Delta} &= (0.75\cdot\mathbf{x1}'+0.25\cdot\mathbf{x2}')\cdot\boldsymbol{\Delta} \\
&= (0.75\cdot(\mathbf{x1}+0.75\lambda\cdot\boldsymbol{\Delta})+0.25\cdot(\mathbf{x2}+0.25\lambda\cdot\boldsymbol{\Delta}))\cdot\boldsymbol{\Delta} \\
&= (0.75\cdot\mathbf{x1}+0.25\cdot\mathbf{x2})\cdot\boldsymbol{\Delta}+\lambda(0.75^2+0.25^2)\cdot\boldsymbol{\Delta}^2 \\
&= \mathbf{p}\cdot\boldsymbol{\Delta}+\lambda(0.75^2+0.25^2)\cdot\boldsymbol{\Delta}^2
\end{aligned}
$$

which together with the right-hand side of (**) gives

$$\lambda = \frac{(\mathbf{q}-\mathbf{p})\cdot\boldsymbol{\Delta}}{(0.75^2+0.25^2)\cdot\boldsymbol{\Delta}^2}.$$

Plugging λ into (*) gives us the new positions of the particles for which **p**' coincide with **q**.

Figure 5 shows the situation after moving the particles. We have no object penetration but now the stick length constraint has been violated. To fix this, we do yet another iteration of the relaxation loop (or several) and we're finished.

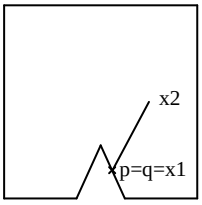


Figure 5a. Collision I resolved.

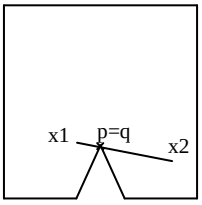


Figure 5b. Collision II resolved.

The above strategy also works for the tetrahedron in a completely analogous fashion. First the penetration points **p** and **q** are found (they may also be points interior to a triangle), and **p** is expressed as a linear combination of the four particles **p**=c1 **x1**+c2 **x2**+c3 **x3**+c4 **x4** such that c1+c2+c3+c4=1 (this calls for solving a small system of linear equations). After finding Δ=**q**-**p**, one computes the value

$$\lambda = \frac{(\mathbf{q}-\mathbf{p})\cdot\boldsymbol{\Delta}}{(c1^2+c2^2+c3^2+c4^2)\cdot\boldsymbol{\Delta}^2}$$

and the new positions are then given by

$$
\begin{aligned}
\mathbf{x1}' &= \mathbf{x1}+c1\cdot\lambda\cdot\boldsymbol{\Delta} \\
\mathbf{x2}' &= \mathbf{x2}+c2\cdot\lambda\cdot\boldsymbol{\Delta} \\
\mathbf{x3}' &= \mathbf{x3}+c3\cdot\lambda\cdot\boldsymbol{\Delta} \\
\mathbf{x4}' &= \mathbf{x4}+c4\cdot\lambda\cdot\boldsymbol{\Delta}.
\end{aligned}
$$

Here, we have collided a single rigid body with an immovable world. The above method generalizes to handle collisions of several rigid bodies. The collisions are processed for one pair of bodies at a time. Instead of moving only **p**, in this case both **p** and **q** are moved towards each other.

Again, after adjusting the particle positions such that they satisfy the non-penetration constraints, the six distance constraints that make up the rigid body should be taken care of and so on. With this method, the tetrahedron can even be imbedded inside another object that can be used instead of the tetrahedron itself to handle collisions. In Figure 6, the tetrahedron is embedded inside a cube.

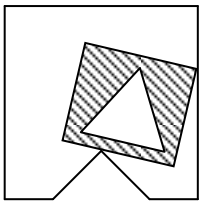


Figure 6. Embedding the tetrahedron inside another object.

First, the cube needs to be 'fastened' to the tetrahedron in some way. One approach would be choosing the system mass midpoint 0.25*(**x1**+**x2**+**x3**+**x4**) as the cube's position and then derive an orientation matrix by examining the current positions of the particles. When a collision/penetration is found, the collision point **p** (which in this case will be placed on the cube) is then treated exactly as above and the positions of the particles are updated accordingly. As an optimization, it is possible to precompute the values of c1-c4 for all vertices of the cube. If the penetration point **p** is a vertex, the values for c1-c4 can be looked up and used directly. Otherwise, **p** lies on the interior of a surface triangle or one of its edges and the values of c1-c4 can then be interpolated from the precomputed values of the corresponding triangle vertices.

Usually, 3 to 4 relaxation iterations are enough. The bodies will not behave as if they were completely rigid since the relaxation iterations are stopped prematurely. This is mostly a nice feature, actually, as there is no such thing as perfectly rigid bodies – especially not human bodies. It also makes the system more stable.

By rearranging the positions of the particles that make up the tetrahedron, the physical properties can be changed accordingly (mathematically, the inertia tensor changes as the positions and masses of the particles are changed).

Other arrangements of particles and constraints than a tetrahedron are possible such as placing the particles in the pattern of a coordinate system basis, i.e. at (0,0,0), (1,0,0), (0,1,0), (0,0,1). Let **a**, **b**, and **c** be the vectors from particle 1 to particles 2, 3, and 4, respectively. Constrain the particles' positions by requiring vectors **a**, **b**, and **c** to have length 1 and the angle between each of the three pairs of vectors to be 90 degrees (the corresponding dot products should be zero). (Notice, that this again gives four particles and six constraints.)

6 Articulated bodies

It is possible to connect multiple rigid bodies by hinges, pin joints, and so on. Simply let two rigid bodies share a particle, and they will be connected by a

pin joint. Share two particles, and they are connected by a hinge. See Figure 7.

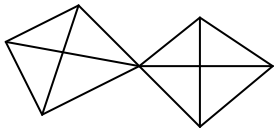


Figure 7a. A pin joint.

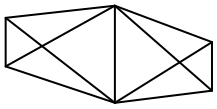


Figure 7b. A hinge.

It is also possible to connect two rigid bodies by a stick constraint or any other kind of constraint – to do this, one simply adds the corresponding 'fix-up' code to the relaxation loop.

This approach makes it possible to construct a complete model of an articulated human body. For additional realism, various angular constraints will have to be implemented as well. There are different ways to accomplish this. A simple way is using stick constraints that are only enforced if the distance between two particles falls below some threshold (mathematically, we have a unilateral (inequality) distance constraint, |**x2**-**x1**|>100). As a direct result, the two particles will never come too close to each other. See Figure 8.

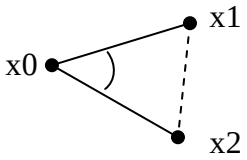


Figure 8. Two stick constraints and an inequality constraint (dotted).

Another method for restraining angles is to satisfy a dot product constraint:

$$(\mathbf{x2}-\mathbf{x0})\cdot(\mathbf{x1}-\mathbf{x0})<\alpha.$$

Particles can also be restricted to move, for example, in certain planes only. Once again, particles with positions not satisfying the above-mentioned constraints should be moved – deciding exactly how is slightly more complicated that with the stick constraints.

Actually, in Hitman corpses aren't composed of rigid bodies modeled by tetrahedrons. They are simpler yet, as they consist of particles connected by stick constraints in effect forming stick figures. See Figure 9. The position and orientation for each limb (a vector and a matrix) are then derived for rendering purposes from the particle positions using various cross products and vector normalizations (making certain that knees and elbows bend naturally).

In other words, seen isolated each limb is not a rigid body with the usual 6 degrees of freedom. This means that *physically* the rotation around the length axis of a limb is not simulated. Instead, the skeletal animation system used to setup the polygonal mesh of the character is forced to orientate the leg, for instance, such that the knee appears to bend naturally. Since rotation of legs and arms around the length axis does not comprise the essential motion of a falling human body, this works out okay and actually optimizes speed by a great deal.

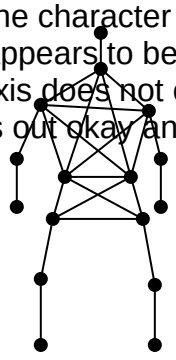


Figure 9. The particle/stick configuration used in Hitman for representing the human anatomy.

Angular constraints are implemented to enforce limitations of the human anatomy. Simple self collision is taken care of by strategically introducing inequality distance constraints as discussed above, for example between the two knees – making sure that the legs never cross.

For collision with the environment, which consists of triangles, each stick is modeled as a capped cylinder. Somewhere in the collision system, a subroutine handles collisions between capped cylinders and triangles. When a collision is found, the penetration depth and points are extracted, and the collision is then handled for the offending stick in question exactly as described in the beginning of Section 5.

Naturally, a lot of additional tweaking was necessary to get the result just right.

7 Comments

This section contains various remarks that didn't fit anywhere else.

Motion control

To influence the motion of a simulated object, one simply moves the particles correspondingly. If a person is hit at the shoulder, move the shoulder particle backwards over a distance proportional to the strength of the blow. The Verlet integrator will then automatically set the shoulder in motion.

This also makes it easy for the simulation to 'inherit' velocities from an underlying traditional animation system. Simply record the positions of the particles for two frames and then give them to the Verlet integrator, which then automatically continues the motion. Bombs can be implemented by pushing each particle in the system away from the explosion over a distance inversely proportional to the square distance between the particle and the bomb center.

It is possible to constrain a specific limb, say the hand, to a fixed position in space. In this way, one can implement inverse kinematics (IK): Inside the relaxation loop, keep setting the position of a specific particle (or several particles) to the position(s) wanted. Giving the particle infinite mass (invmass=0) helps making it immovable to the physics system. In Hitman, this strategy is used when dragging corpses; the hand (or neck or foot) of the corpse is constrained to follow the hand of the player.

Handling friction

Friction has not been taken care of yet. This means that unless we do something more, particles will slide along the floor as if it were made of ice. According to the Coulomb friction model, friction force depends on the size of the normal force between the objects in contact. To implement this, we measure the penetration depth d_p when a penetration has occurred (before projecting the penetration point out of the obstacle). After projecting the particle onto the surface, the tangential velocity $\mathbf{v}_t$ is then reduced by an amount proportional to d_p (the proportion factor being the friction constant).

This is done by appropriately modifying **x***. See the Figure 10. Care should be taken that the tangential velocity does not reverse its direction – in this case one should simply be set it to zero since this indicates that the penetration point has seized to move tangentially. Other and better friction models than this could and should be implemented.

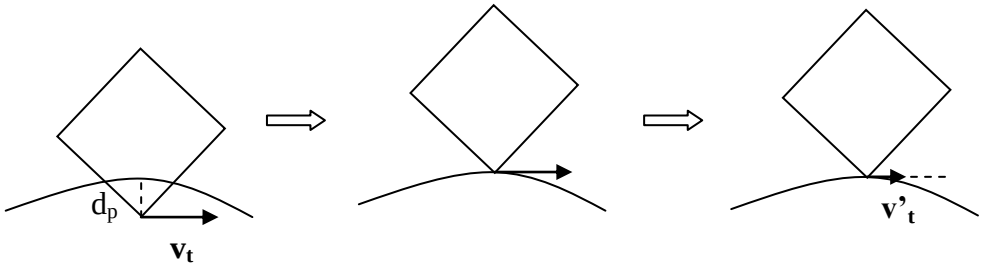


Figure 10. Collision handling with friction (projection and modification of tangential velocity).

Collision detection

One of the bottlenecks in physics simulation as presented here lies in the collision detection, which is potentially performed several times inside the relaxation loop. It is possible, however, to iterate a different number of times over the various constraints and still obtain good results.

In Hitman, the collision system works by culling all triangles inside the bounding box of the object simulated (this is done using a octtree approach). For each (static, background) triangle, a structure for fast collision queries against capped cylinders is then constructed and cached. This strategy gave quite a speed boost.

To prevent objects that are moving really fast from passing through other obstacles (because of too large time steps), a simple test if performed. Imagine the line (or a capped cylinder of proper radius) beginning at the position of the object's midpoint last frame and ending at the position of the object's midpoint at the current frame. If this line hits anything, then the object position is set to the point of collision. Though this can theoretically give problems, in practice it works fine.

Another collision 'cheat' is used for dead bodies. If the unusual thing happens that a fast moving limb ends up being placed with the ends of the capped cylinder on each side of a wall, the cylinder is projected to the side of the wall where the cylinder is connected to the torso.

Miscellaneous

The number of relaxation iterations used in Hitman vary between 1 and 10 with the kind of object simulated. Although this is not enough to accurately solve the global system of constraints, it is sufficient to make motion seem natural. The nice thing about this scheme is that inaccuracies do not accumulate or persist visually in the system causing object drift or the like – in some sense the combination of projection and the Verlet scheme manages to distribute complex calculations over several frames (other schemes have to use further stabilization techniques, like Baumgarte stabilization). Fortunately,

the inaccuracies are smallest or even nonexistent when there is little motion and greatest when there is heavy motion – this is nice since fast or complex motion somewhat masks small inaccuracies for the human eye.

A kind of soft bodies can also be implemented by using ‘soft’ constraints, i.e., constraints that are allowed to have only a certain percentage of the deviation ‘repaired’ each frame (i.e., if the rest length of a stick between two particles is 100 but the actual distance is 60, the relaxation code could first set the distance to 80 instead of 100, next frame 90, 95, 97.5 etc.).

As mentioned, we have purposefully refrained from using heavy mathematical notation in order to reach an audience with a broader background. This means that even though the methods presented are firmly based mathematically, their origins may appear somewhat vague or even magical.

For the mathematically inclined, however, what we are doing is actually a sort of time-stepping approach to solving differential inclusions (a variant of differential equations) using a simple sort of interior-point algorithm (see [Stewart] where a similar approach is discussed). When trying to satisfy the constraints, we are actually projecting the system state onto the manifold described by the constraints. This, in turn, is done by solving a system of linear equations. The linear equations or code to solve the constraints can be obtained by deriving the Jacobian of the constraint functions. In this article, relaxation has been discussed as an implicit way of solving the system. Although we haven’t touched the subject here, it is sometimes useful to change the relaxation coefficient or even to use over-relaxation (see [Press] for an explanation). Since relaxation solvers sometimes converge slowly, one might also choose to explicitly construct the equation system and use other methods to solve it (for example a sparse matrix conjugate gradient descent solver with preconditioning using the results from the previous frame (thereby utilizing coherence)).

Note that the Verlet integrator scheme exists in a number of variants, e.g., the Leapfrog integrator and the velocity Verlet integrator. Accuracy might be improved by using these.

Singularities (divisions by zero usually brought about by coinciding particles) can be handled by slightly dislocating particles at random.

As an optimization, bodies should time out when they have fallen to rest.

To toy with the animation system for dead characters in Hitman: Codename 47, open the Hitman.ini file and add the two lines “enableconsole 1” and “consolecmd ip_debug 1” at the bottom. Pointing the cursor at an enemy and pressing shift+F9 will cause a small bomb to explode in his vicinity sending him flying. Press K to toggle free-cam mode (camera is controlled by cursor keys, shift, and ctrl).

Note that since all operations basically take place on the particle level, the algorithms should be very suitable for vector processing (Playstation 2 for example).

8 Conclusion

This paper has described how a physics system was implemented in Hitman. The underlying philosophy of combining iterative methods with a stable integrator has proven to be successful and useful for implementation in computer games. Most notably, the unified particle-based framework, which handles both collisions and contact, and the ability to trade off speed vs. accuracy without accumulating visually obvious errors are powerful features. Naturally, there are still many specifics that can be improved upon. In particular, the tetrahedron model for rigid bodies needs some work. This is in the works.

At IO Interactive, we have recently done some experiments with interactive water and gas simulation using the full Navier-Stokes equations. We are currently looking into applying techniques similar to the ones demonstrated in this paper in the hope to produce faster and more stable water simulation.

9 Acknowledgements

The author wishes to thank Jeroen Wagenaar for fruitful discussions and the entire crew at IO Interactive for cooperation and for producing such a great working environment.

Feedback and comments are very welcome at tj@ioi.dk.

References

[Baraff] Baraff, David, *Dynamic Simulation of Non-Penetrating Rigid Bodies*, Ph.D. thesis, Dept. of Computer Science, Cornell University, 1992. http://www.cs.cmu.edu/~baraff/papers/index.html

[Mirtich] Mirtich, Brian V., *Impulse-base Dynamic Simulation of Rigid Body Systems*, Ph.D. thesis, University of California at Berkeley, 1996. http://www.merl.com/people/mirtich/papers/thesis/thesis.html

[Press] Press, William H. et al, *Numerical Recipes,* Cambridge University Press, 1993. http://www.nr.com/nronline_switcher.html

[Stewart] Stewart, D. E., and J. C. Trinkle, “An Implicit Time-Stepping Scheme for Rigid Body Dynamics with Inelastic Collisions and Coulomb Friction”, *International Journal of Numerical Methods in Engineering*, to appear. http://www.cs.tamu.edu/faculty/trink/Papers/ijnmeStewTrink.ps

[Verlet] Verlet, L. "Computer experiments on classical fluids. I. Thermodynamical properties of Lennard-Jones molecules", *Phys. Rev.*, **159**, 98-103 (1967).

[Witkin] Witkin, Andrew and David Baraff, *Physically Based Modeling: Principles and Practice*, Siggraph ’97 course notes, 1997. http://www.cs.cmu.edu/~baraff/sigcourse/index.html

The author

Thomas Jakobsen is head of research and development at IO Interactive, where he is developing new technology for the in-house engine, Glacier. He created the physics simulation

and the 3D pathfinder for IO's Hitman: Codename 47. Jakobsen earned his M.Sc. in engineering from The Technical University of Denmark, where he also obtained his Ph.D. in mathematics. He spent a number of years in academia and has published several scientific articles on cryptanalysis. He began programming at age nine with a Sinclair ZX80. In 1986, he co-designed Vikings for the Commodore 64 and Amstrad CPC. In the late 1980s, he developed tools for the Amiga classics Hybris and Battle Squadron.