Verlet Integration

CS-116B: Computer Graphics Algorithms
Spring 2018

Verlet Integration

Verlet integration: a numerical integration scheme based on Euler integration. Recall in previous discussions of numerical integration:

$$x_{new} = x_{current} + v_{current} * \Delta t$$
$$v_{new} = v_{current} + a * \Delta t$$

Verlet integration uses previous displacement and current displacement and a constant timestep:

$$x_{new} = 2x_{current} - x_{previous} + a * \Delta t^2$$
$$x_{previous} = x_{current}$$

Source: *Advanced Character Physics*, p. 3.

Verlet Integration

```
// 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)
};
```

Source: *Advanced Character Physics*, p. 3.

Verlet Integration

```
// 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;
}
```

Source: *Advanced Character Physics*, p. 4.

Verlet Integration

```
// Here constraints should be satisfied
void ParticleSystem::SatisfyConstraints()
{
  // Ignore this function for now
}


void ParticleSystem::TimeStep()
{
  AccumulateForces();
  Verlet();
  SatisfyConstraints();
}
```

Source: *Advanced Character Physics*, p. 4.

Verlet Integration

```
// Here constraints should be satisfied
void ParticleSystem::SatisfyConstraints()
{
  // Ignore this function for now
}


void ParticleSystem::TimeStep()
{
  AccumulateForces();
  Verlet();
  SatisfyConstraints();
}
```

Source: *Advanced Character Physics*, p. 4.

Solving concurrency through relaxation

- “Strong springs leads to stiff systems of equations that lead to instability if only simple integration techniques are used....” (p. 5).
- “Conversely, weak springs lead to elastically looking cloth” (p. 5).
- “...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” (p. 5).

Source: *Advanced Character Physics*.

Solving concurrency through relaxation

- **PROBLEM:** “Although the particles might be correctly placed initially, after one integration step the separation distance between them might have become invalid” (p. 6). This introduces error that acculumates over time and creates instability.

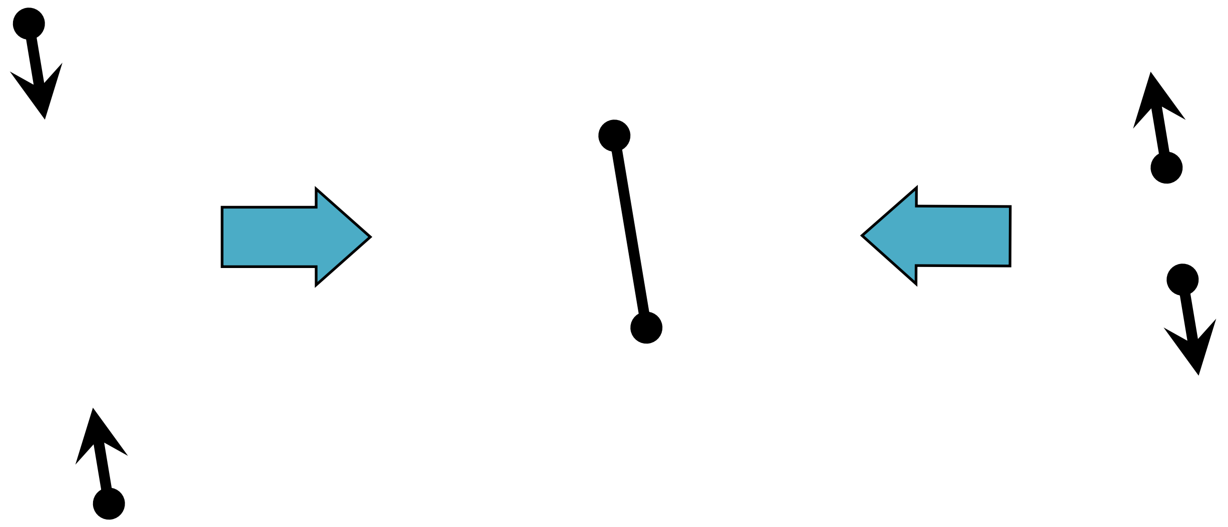
- **SOLUTION:** After each iteration, reposition the particles to correct the incorrect distance (p. 6). This brings stability and accuracy to cloth simulation by ellimating the cumulative error from each iteration.

- “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” (p. 6).

- This technique for solving concurrency through relaxation is called **Jacobi or Gauss-Seidel iteration**.

Source: *Advanced Character Physics*, p. 7.

Solving concurrency through relaxation: constraint C2



Distance too large

Correct distance

Distance too small

Source: *Advanced Character Physics*, p. 6.

Jacobi or Gauss-Seidel iteration

```
// 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;
  }
}
```

Source: *Advanced Character Physics*, p. 7

Pseudo code to satisfy C2

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

Source: *Advanced Character Physics*, p. 7