Introduction to Cloth

Christopher Twigg
15-864 Spring 2004
Outline

• Review of physically based modeling
• A simple cloth system
• Implementation details
Review of PBM basics
The rest is just details

\[ F = ma \]
Recall: Particles
Midpoint method

\[ x_{t+\Delta t=2} = x_t + \frac{\Delta t}{2} f(x_t; t) \]

\[ x_{t+\Delta t} = x_t + (\Delta t)f(x_{t+\Delta t=2}; t + \Delta t=2) \]
Midpoint vs. Euler

- **Midpoint method**

- **One Euler step**

- **Two Euler steps**
Adaptive stepping
Idea 1: step doubling

Estimated error
Idea 2: Steps of different order

Euler step (1\textsuperscript{st} order)

Estimated error

Midpoint step (2\textsuperscript{nd} order)
Idea 3: Embedded pairs

• Compare methods of different orders, but:
• Reuse derivatives
Idea 4: Use somebody else’s code

• Because who wants to write code like this?

```c
static float a2=0.2,a3=0.3,a4=0.6,a5=1.0,a6=0.875,b21=0.2,
b31=0.0/40.0,b32=0.0/40.0,b41=0.3,b42 = -0.9,b43=1.2,
b51 = -11.0/54.0, b52=2.5,b53 = -70.0/27.0,b54=35.0/27.0,
b61=1831.0/5526.0,b62=175.0/512.0,b63=575.0/13824.0,
b64=44275.0/110592.0,b65=253.0/4096.0,c1=37.0/378.0,
c2=259.0/621.0,c4=125.0/594.0,c6=612.0/1771.0,
dc5 = -277.0/14356.0;
float c1=2825.0/27648.0,dc3=c3=18875.0/48884.0,
dc4=c4=13525.0/55296.0,dc6=c6=6.0/25;
float *ak2,*ak3,*ak4,*ak5,*ak6,*ytmp;

ak2=vector(1,n); ak3=vector(1,n); ak4=vector(1,n); ak5=vector(1,n); ak6=vector(1,n);
ytmp=vector(1,n); for (i=1; i<=n; i++)
    ytmp[i]=y[i]+b21*h*dydx[i]; /*derivs(x+a2*h,ytemp,ak2); Second step.
for (i=1; i<=n; i++)
    ytemp[i]=i=b31*dydx[i]+b32*ak2[i]; /*derivs(x+a3*h,ytemp,ak3); Third step.
for (i=1; i<=n; i++)
    ytmp[i]=i=b41*dydx[i]+b42*ak2[i]+b43*ak3[i]; /*derivs(x+a4*h,ytemp,ak4); Fourth step.
for (i=1; i<=n; i++)
    ytemp[i]=i=b51*dydx[i]+b52*ak2[i]+b53*ak3[i]+b54*ak4[i]; /*derivs(x+a5*h,ytemp,ak5); Fifth step.
for (i=1; i<=n; i++)
    ytmp[i]=i=b61*dydx[i]+b62*ak2[i]+b63*ak3[i]+b64*ak4[i]+b65*ak5[i]; /*derivs(x+a6*h,ytemp,ak6); Sixth step.
for (i=1; i<=n; i++)
    yout[i]=i+c1*dydx[i]+c3*ak3[i]+c4*ak4[i]+c6*ak6[i]; for (i=1; i<=n; i++)
    yerr[i]=i=dc1*dydx[i]+dc3*ak3[i]+dc4*ak4[i]+dc5*ak5[i]+dc6*ak6[i];
Estimate error as difference between fourth and fifth order methods.
```
Review: Second-order systems

Remember this guy?

\[ F = ma \]
Review: Second-order systems

To be more precise:

\[ F = m \frac{d^2x}{dt^2} \]
Review: Second-order systems

• Fortunately, we can rewrite this as

\[
\begin{align*}
\frac{dx}{dt} &= v \\
\frac{dv}{dt} &= \frac{F}{m}
\end{align*}
\]
A simple cloth system
Recall: spring-mass systems

\[ f_{\text{spring}} = -k_s (s - s_0) \frac{d}{jjdjj} \]

where

\[ d = p \cdot q \]

s is the rest length of the spring

\( k_s \) is the spring constant
Damped spring

• Add a simple damping force for stability
• Properties:
  – Depends on *relative velocity* in the *direction of force*
  – Acts in direction of spring force

\[ f_{\text{spring}} = i \left( k_s (jjdjj \ i \ s) + k_d \frac{\dot{d} \circ \dot{d}}{jjdjj} \right) \frac{d}{jjdjj} \]

where

\[ \dot{d} = \frac{dp}{dt} \ i \ \frac{dq}{dt} \]
Simple spring-based cloth

Now, we just start connecting things up…
Step 1: Stretch springs
Step 1: Stretch springs

Oops…

Source: [Lander 1999]
Step 2: Shear springs

Source: [Lander 1999]
Step 3: Bend springs
Are we done?

Source: [Provot 1995]
Option 1: increase $k_s$

- Problem: stiffness (on board)
Stiffness (2)

\[ x = -kx \quad \Delta x = -hkx \]

\[ h = 0.5(1/k) \quad h = 1(1/k) \quad h = 1.5(1/k) \quad h = 2(1/k) \quad h = 3(1/k) \]

\[ h > 1/k: \text{ oscillate.} \quad h > 2/k: \text{ explode!} \]

Source: Baraff course notes (2001)
Stiffness (3)

• We’ll deal with this specifically on Thursday, but for now...
Other solutions?

• Simple heuristic due to Provot:
  – After every timestep, adjust spring lengths

(a) Adjustment of a “super-elongated” spring linking two loose masses.

(b) Adjustment of a “super-elongated” spring linking a fixed mass and a loose mass.
Note: must iterate until convergence
Results:

(a) Our method applied to "structural" springs
Heuristic approach…

• Also used by [Bridson et al., 2002] (covered in more depth on Thursday)
Implementation details

• Use an external solver!
• GSL (Gnu Scientific Library) contains:
  – Embedded 2nd order Runge-Kutta with 3rd order error estimate
  – Embedded 4th order Runge-Kutta-Fehlberg method with 5th order error estimate
  – Embedded 8th order Runge-Kutta Prince-Dormand method with 9th order error estimate (!!)
  ...

ODE solver (1): storing state

double y[6*N]

3*N positions

3*N velocities
ODE solver (2): the derivs function

```c
int derivs (double t, const double y[],
           double dydt[], void * params)
```

Evaluates the state derivatives at \((y, t)\)

\[
\begin{align*}
    \text{(input)} & \\
    y & \\
    x & \\
    v & \\
\end{align*}
\]

\[
\begin{align*}
    \text{(output)} & \\
    dy/dt & \\
    dx/dt &= v \\
    dv/dt &= f/m
\end{align*}
\]
ODE solver (3): the derivs function

```c
int derivs (double t, const double y[],
    double dydt[], void * params)
```

Pass in anything that is useful; e.g., a pointer to the particle system.
int derivs (double t, const double y[], double dydt[], void * params)
{
    ClothModel* model =
        reinterpret_cast< ClothModel *>(params);

    // Copy velocities into dx/dt spot
    std::copy( y + 3*model->N,
                y + 6*model->N, f );

    // Evaluate forces at the current position and timestep
    model->f( t, y, y + 3*model->N, f + 3*model->N );

    // Scale forces by 1/m
    // Note that invMass is 0 for constrained particles
    for( unsigned int i = 0; i < 3*model->N; ++i )
        f[ 3*model->N + i ] *= model->invMass( i/3 );
}
ODE setup for GSL

• First, pick a stepping function from this list:
  – gsl_odeiv_step_rk2
  – gsl_odeiv_step_rk4
  – gsl_odeiv_step_rkf45
  – gsl_odeiv_step_rkck
  – gsl_odeiv_step_rk8pd

• (implicit methods are off-limits for now)
Now, do some boilerplate allocation:

```c
// At start
gsl_odeiv_step* s_;
gsl_odeiv_control* c_;
gsl_odeiv_evolve* e_; 

s_ = gsl_odeiv_step_alloc(stepType, 6*N_);
c_ = gsl_odeiv_control_y_new(error, 0.0);
e_ = gsl_odeiv_evolve_alloc(6*N);

// At exit
gsl_odeiv_evolve_free(e_);
gsl_odeiv_control_free(c_);
gsl_odeiv_step_free(s_);
```
• Or, if we’re clever...

class GSLWrapper
{
public:
    GSLWrapper( const gsl_odeiv_step_type* stepType,
                unsigned int systemSize,
                double *timestep,
                Real error)
    : timestep_( timestep ),
      systemSize_(systemSize)
    {
        s_ = gsl_odeiv_step_alloc (stepType, systemSize_);
        c_ = gsl_odeiv_control_y_new (error, 0.0);
        e_ = gsl_odeiv_evolve_alloc (systemSize);
    }

    ~GSLWrapper()
    {
        gsl_odeiv_evolve_free(e_);
        gsl_odeiv_control_free(c_);
        gsl_odeiv_step_free(s_);
    }

    // ...
};
Now, for the solver loop

```cpp
std::vector<double> state( 6*N );
// copy state from system to state vector

double currentTime = 0.0;
double endTime = 1.0/30.0; // one animation frame
double timestep = 0.01; // initial guess
while( currentTime < endTime )
{
    gsl_odeiv_system sys =
        { derivs, 0, 6*N, (*this) };
    int status = gsl_odeiv_evolve_apply(
        e_, c_, s_,
        &sys,
        &currentTime, endTime,
        &timestep, &y[0]);
    if( status != GSL_SUCCESS )
        // do something!
        // should correct particle positions here.
}
// now copy state back into the particle system
```
And that’s it!

• You should have (almost) everything you need to implement your cloth solver
• Collisions on Thursday