Deformable Body Simulation

CIS 563 HW2
Outline

● Workflow
● Time Integration Method
  ○ Force Based Methods
    ■ Explicit & Implicit
  ○ Position Based Dynamics (PBD)
    ■ Constraints Projection
    ■ Pros & Cons
● Base code
  ○ Data Structure
  ○ Framework
  ○ Extra Credits
Space Discretization

Particle

- mass: \( m_i \)
- position: \( x_i \)
- velocity: \( v_i \)
Springs

- rest length: $l_0$
- stiffness: $k$

Structural/Stretch Spring

Bending Spring
Workflow

- Calculate external force
- Time integration
  - Force based methods
  - Position based dynamics (PBD)
- Detect Collisions
- Resolve Collisions
- Damp velocity
Workflow

- Calculate external force
- **Time integration**
  - Force based methods
  - PBD
- Detect Collisions
- Resolve Collisions
- Damp velocity
Time Integration

Predict the simulation state at the next timestep

\[ v_{n+1} = v_n + \int_{t_n}^{t_{n+1}} a(t) \, dt \]
\[ x_{n+1} = x_n + \int_{t_n}^{t_{n+1}} v(t) \, dt \]

- Force based method
- PBD (constraints)
Workflow
● Update external force
● Time integration
  ○ Force based methods
  ○ PBD
● Detect Collisions
● Resolve Collisions
● Damp velocity
Force Based Integration

\[ m_i a_i = F_{ext} + F_{int} \]

For all vertices, calculate \( F_{int} \)

\[ F_{int} = -\frac{\partial E}{\partial x_i} = -\frac{\partial}{\partial x_i} \left( \frac{1}{2} k_s (\|x_i - x_j\| - l_0)^2 \right) \]

\[ F_{int} = -k_s (\|x_i - x_j\| - l_0) \frac{x_i - x_j}{\|x_i - x_j\|} \]
Force Based Integration

- **Explicit**
  - Euler (Runge-Kutta)
  - Heun’s Method (RK2)
  - RK4
  - Symplectic

- **Implicit**
  - Baraff-Witkin’s Method
Explicit Euler (RK1)

\[ y(t_0) = y_0 \]

\[ y(t_0 + h) \]

\[ y_1 = y_0 + h \cdot f(t_0, y_0) \]
Heun’s Method (RK2)
RK4

\[ y_{n+1} = y_n + \frac{1}{6} h \cdot (k_1 + 2k_2 + 2k_3 + k_4) \]
RK4 in Practice

- Calculate $k_1$, $k_2$, $k_3$, $k_4$

```plaintext
// step 1: compute k1
VectorX f1;
computeForces(x, f1);
// get the slope for x and v separately
VectorX k1_x = v;
VectorX k1_v = m_mesh->m_inv_mass_matrix * f1;
// setup a temporary point for next iteration
temp_x = x + half_dt * k1_x;
temp_v = v + half_dt * k1_v;
```

```plaintext
// step 2: compute k2
VectorX f2;
computeForces(temp_x, f2);
// get the slope for x and v separately
VectorX k2_x = temp_v;
VectorX k2_v = m_mesh->m_inv_mass_matrix * f2;
// setup a temporary point for next iteration
temp_x = x + half_dt * k2_x;
temp_v = v + half_dt * k2_v;
```

$f(t + 0.5h, y_n + 0.5h \cdot k_1)$  \hspace{1cm}  $f(t + 0.5h, y_n + 0.5h \cdot k_2)$
RK4 in Practice

- Combine them together

\[ y_{n+1} = y_n + \frac{1}{6} h \cdot (k_1 + 2k_2 + 2k_3 + k_4) \]

```c
// Put it all together
x = x + 1/6.0 * dt * (k1_x + 2*k2_x + 2*k3_x + k4_x);
v = v + 1/6.0 * dt * (k1_v + 2*k2_v + 2*k3_v + k4_v);
```
Explicit Symplectic

Euler:

\[ v_{n+1} \leftarrow a_n \]
\[ x_{n+1} \leftarrow v_n \]

Explicit Symplectic: update position using \textit{updated} velocity

\[ v_{n+1} \leftarrow a_n \]
\[ x_{n+1} \leftarrow v_{n+1} \]
Demo

Explicit Euler
Heun’s Method
RK4
Explicit Symplectic

Implicit method
Explicit vs Implicit

- Explicit: \[ y_{n+1} = y_n + h f(t_n, y_n) \]
  
  all quantities are known

- Implicit: \[ y_{n+1} = y_n + h f(t_{n+1}, y_{n+1}) \]

Implicit method is more stable
Baraff-Witkin Method

\[ v_{n+1} = v_n + hM^{-1}f(x_{n+1}) = v_n + hM^{-1}\nabla E(x_{n+1}) \]

\[ x_{n+1} = x_n + hv_{n+1} \]

\[ \nabla E(x_{n+1}) \approx \nabla E(x_n) + \nabla^2 E(x_n)(x_{n+1} - x_n) \]

\[ \nabla E(x_{n+1}) \approx \nabla E(x_n) + \nabla^2 E(x_n)hv_{n+1} \]

\[ v_{n+1} = v_n + hM^{-1}(\nabla E(x_n) + \nabla^2 E(x_n)hv_{n+1}) \]

\[ Av_{n+1} = b \]
Workflow

- Calculate external force
- Time integration
  o Force based methods
  o PBD
- Detect Collisions
- Resolve Collisions
- Damp velocity
PBD

Direct control over positions
- Update velocity and position after constraint projection

Delay in force propagation
- Project Constraints one by one
- Next linearization works with the updated state

[position based dynamics by m müller, b heidelberger, m hennix, j ratcliff]
Main idea:

- Explicit Integration
  \[ v_i \leftarrow v_i + \Delta t \frac{f_{ext}(x_i)}{m_i} \]
  \[ p_i \leftarrow x_i + \Delta t v_i \]
- Generate Constraints (collision constraints)
- Project Constraints \((C_1, \ldots, C_M, p_1, \ldots, p_N)\)
- Update \(v\) & \(x\)
  \[ v_i \leftarrow \frac{(p_i - x_i)}{\Delta t} \]
  \[ x_i \leftarrow p_i \]
Pros & Cons

Pros:

● Provide control over positions directly.
● Fast and stable.
● Generality of all kinds of constraints.

Cons:

● Parameters are less intuitive.
Constraints

- Stretch Constraints
- Attachment Constraints
- Collision Constraints
- Bending Constraints
- ...

Stretch Constraint

Spring/Distance constraint:

\[ \Delta p_1 = -\frac{w_1}{w_1 + w_2}(|p_1 - p_2| - d) \frac{p_1 - p_2}{|p_1 - p_2|} \]

\[ \Delta p_2 = +\frac{w_2}{w_1 + w_2}(|p_1 - p_2| - d) \frac{p_1 - p_2}{|p_1 - p_2|} \]

Update predicted position:

\[ p_i += k\Delta p_i \]
Attachment Constraint

Spring Constraint with rest length 0
Constraints

- Stretch Constraints
- Attachment Constraints
- Collision Constraints
- Real Bending Constraints
- ...

Solver iteration:
Reverse the order in each iteration
Workflow

- Calculate external force
- Time integration
  - Force based methods
  - PBD
- Detect Collisions
- Resolve Collisions
- Damp velocity
Detect Collision

- Point-Static Object (plane, sphere, cube and objMesh)
- Self-Collision [chapter 4.3 in PBD paper]
Workflow

- Update external force
- Time integration
  - Force based methods
  - PBD
- Detect Collisions
- Resolve Collisions
- Damp velocity
Resolve Collision

Velocity (normal direction): Reflect

Position: Move to the collider surface
Workflow

● Calculate external force

● Time integration
  ○ Force based methods
  ○ PBD

● Detect Collisions

● Resolve Collisions

● Damp velocity
Damp Velocity

Real-world mechanical systems dissipate energy over time.

- Simplest: $v_i = k_d v_i$
- Momentum Conservation

[chapter 3.5 in PBD paper]

1. $x_{cm} = (\sum_i s_i m_i) / (\sum_i m_i)$
2. $v_{cm} = (\sum_i v_i m_i) / (\sum_i m_i)$
3. $L = \sum_i r_i \times (m_i v_i)$
4. $I = \sum_i \ddot{r}_i \ddot{r}_i^T m_i$
5. $\omega = I^{-1} L$
6. Forall vertices $i$
7. $\Delta v_i = v_{cm} + \omega \times r_i - v_i$
8. $v_i \leftarrow v_i + k_{damping} \Delta v_i$
9. endfor
Code Framework

- framework
  ...
- object
  - mesh
  - primitive
  - scene
  ...
- shader_headers
  ...
- simulation
  - constraints
  - simulation
enum MeshType { MESH_TYPE_CLOTH, MESH_TYPE_TET};

- key properties:

```cpp
MeshType m_mesh_type;

unsigned int m_vertices_number; // m
unsigned int m_system_dimension; // 3m

// vertices positions/previous positions/mass
VectorX m_current_positions; // 1x3m
VectorX m_current_velocities; // 1x3m
SparseMatrix m_mass_matrix; // 3mx3m
SparseMatrix m_inv_mass_matrix; // 3mx3m
SparseMatrix m_identity_matrix; // 3mx3m

// for generating constraints.
std::vector<Edge> m_edge_list;
```
- **enum** PrimitiveType {PLANE, SPHERE, CUBE, OBJMESH};

- Intersection detection:
  – collider normal (outVector)
  – the distance of a particle penetrating into a collider, \( \text{dist} < 0 \)

```cpp
bool StaticIntersectionTest(const EigenVector3& p,
                           EigenVector3& normal, ScalarType& dist)
```
Scene

- Load Scene

- Check intersections among all primitives

- global_headers.h

```c
#define DEFAULT_SCENE_FILE "./scenes/test_scene.xml"
#define DEFAULT_CONFIG_FILE "./config/config.txt"
```
Constraint

- `class` Constraint

```cpp
virtual void EvaluateGradient(const VectorX& x, VectorX& gradient) = 0;
virtual void EvaluateHessian(const VectorX& x, std::vector<SparseMatrixTriplet>& hessian_triplets) = 0;
virtual void PBDProject(VectorX& x, const SparseMatrix& inv_mass, unsigned int ns) = 0;
```

- AttachmentConstraint: `bool m_selected;`

- SpringConstraint: `unsigned int m_p1, m_p2;
  // rest length
  ScalarType m_rest_length;`
Simulation

CollisionInstance
- normal
- distance

```c
struct CollisionInstance
{
    unsigned int point_index; // the id of the colliding particle
    EigenVector3 normal; // normal direction of the collider
    ScalarType dist; // dist is the distance of a particle penetrating into a collider, dist < 0

    CollisionInstance(const unsigned int id, const EigenVector3& n, const ScalarType d)
};
```
Simulation

- key properties

```c++
// integration method
IntegrationMethod m_integration_method;

// key simulation components: constraints
std::vector<Constraint*> m_constraints;
AttachmentConstraint* m_selected_attachment_constraint;

// external force (gravity, wind, etc...)
VectorX m_external_force;
```
Initialization:

- Add AttachmentConstraints and SpringConstraints

    void setupConstraints(); // initialize constraints
Update

Workflow

- Calculate external force

// update external force
calculateExternalForce();
Update

Workflow

- Calculate external force
- Time integration
  - Force based methods

```cpp
// get a reference of current position and velocity
VectorXf x = m_mesh->m_current_positions;
VectorXf v = m_mesh->m_current_velocities;
// substepping timestep for explicit/implicit euler
ScalarType dt = m_h / m_iterations_per_frame;
for (unsigned int it = 0; it != m_iterations_per_frame; ++it) {
    // update cloth
    switch (m_integration_method) {
    case INTEGRATION_EXPLICIT_EULER:
        integrateExplicitEuler(x, v, dt);
        break;
    case INTEGRATION_EXPLICIT_RK2:
        integrateExplicitRK2(x, v, dt);
        break;
    case INTEGRATION_EXPLICIT_RK4:
        integrateExplicitRK4(x, v, dt);
        break;
    case INTEGRATION_EXPLICIT_SYMPLECTIC:
        integrateExplicitSymplectic(x, v, dt);
        break;
    case INTEGRATION_IMPLICIT_EULER_BARAFF_WITKIN:
        integrateImplicitBW(x, v, dt);
        break;
    }
}
```
Update

Workflow
● Calculate external force
● Time integration
  ○ Force based methods
  ○ PBD

integratePBD:
  // update v and x explicitly
  // loop: PBD projections
  // update velocity and position

// pbd integration method
if (m_integration_method == INTEGRATION_POSITION_BASED_DYNAMICS)
{
    integratePBD(x, v, m_iterations_per_frame);
}
Update

Workflow

- Calculate external force
- Time integration
  - Force based methods
  - PBD
- Detect Collisions
- Resolve Collisions
- Damp velocity

```cpp
// collision detection and resolution
std::vector<CollisionInstance> collisions;
detectCollision(x, collisions);
resolveCollision(x, v, collisions);

// damping
dampVelocity();
```
TO-DOs

Initialization (10 points):
void Simulation::setupConstraints()

Compute force (15 points):
// spring gradient: k*(current_length-rest_length)*current_direction;
void SpringConstraint::EvaluateGradient(const VectorX& x, VectorX& gradient)

Intersection detection (10 points):
bool Cube::StaticIntersectionTest(const EigenVector3& p, EigenVector3& normal, ScalarType& dist)
TO-DOs

Integration (10+10+10+15 points):
void Simulation::integrateExplicitEuler(VectorX& x, VectorX& v, ScalarType dt)
void Simulation::integrateExplicitRK2(VectorX& x, VectorX& v, ScalarType dt)
void Simulation::integrateExplicitSymplectic(VectorX& x, VectorX& v, ScalarType dt)

void Simulation::integratePBD(VectorX& x, VectorX& v, unsigned int ns)

Constraint Projection (20 points):
void SpringConstraint::PBDProject(VectorX& x, const SparseMatrix& inv_mass, unsigned int ns)
Extra Credits

● **OBJ mesh intersection detection (15 points):**
  - obj file loading and initialization function is given
  - Complete ObjMesh: StaticIntersectionTest

● **Implicit Integration method (20 points):**
  - Complete EvaluateHessian for all constraints
  - Complete integrateImplicitBW
Extra Credits

Add other constraints:

● **Real bending constraint** (20 points):
  - Construct a new constraint class
  - Add real bending constraints in initialization
  - Projection formulas are in appendix in PBD paper

● **Pressure/Inflating constraint** (30 points):
  - Closed mesh required (ObjMesh)

● **Cloth self-collision constraint** (50 points):
Requirement

- Complete all the TODOs
- Make a video record for your best scene
- Describe the extra credits you have implemented in README.txt
Tips

- Explodes, try:
  - Tune your parameters (in the config file)
  - Add more springs
  - Make your timestep smaller
  - Implicit Euler / PBD is stable

- Cloth disappears, check
  - Divided by zero
Due: Mar. 16

Good Luck with HW2!