Advanced Computer Graphics
IMA904/IG3DA Exercise - Position Based Dynamics

Kiwon Um, Telecom Paris

In this exercise, you will learn how to implement a real-time simulation method, called position based dynamics (PBD), in three-dimensional (3D) space. This exercise starts with a provided codebase where basic modules such as a simple rendering and simulation routines are already implemented. The goal of this exercise is to achieve a simple soft body PBD simulator with a cloth example. Once you finish all the tasks described here, you are strongly encouraged to extend your codes further as you want. The potential directions are described in the end.

1 Codebase

Your first mission is to understand the overall structure of the given codebase. Please read through the `main.cpp` file and the others.

The main solver (`PbdSolver`) is implemented in the `PbdSolver.hpp` file, and the main routine is implemented in the `main.cpp` file. As shown in Code 1, the given codes perform an update-and-render routine, which iteratively calls the solver's `step` function that updates the state (such as vertex position) of the soft body and the global `render` function that redraws the new state using the OpenGL APIs.

Code 1. PBD solver routine

```
// ... PbdSolver.hpp ...
class PbdSolver {
public:
  // ...
  void step(const tReal dt) { /* ... */ }
};

// ... main.cpp ...
void render() { g_scene.render(); }
void update(const float currentTime)
{
  // ...
  g_scene.solver.step(std::min(dt, 0.017f));
  g_scene.solver.updateMesh(*(g_scene.cloth));
}
// ...
```

Code 2. Build and run

```
cmake -B build
# or mkdir build; cd build; cmake ..
make -C build  # or make
./tpPbd
```

1.1 Build and run

Important! This guideline is written for Linux systems. If you use other operating system, you should adapt it accordingly.

The given codebase uses *cmake* as a build system. You can easily build an executable via general cmake commands. (See Code 2.)

If everything works well with your machine, you should be able to see a simple initial simulation setup as on the left of Fig. 1; the middle and right of Fig. 1 are example screenshots of a simulation result you may achieve if you implement important functions properly. You can use Esc to quit, P to toggle pause of your simulation, R to reset/restart your simulation, and S to save a screenshot of the current frame into a file. Try H to see the help.

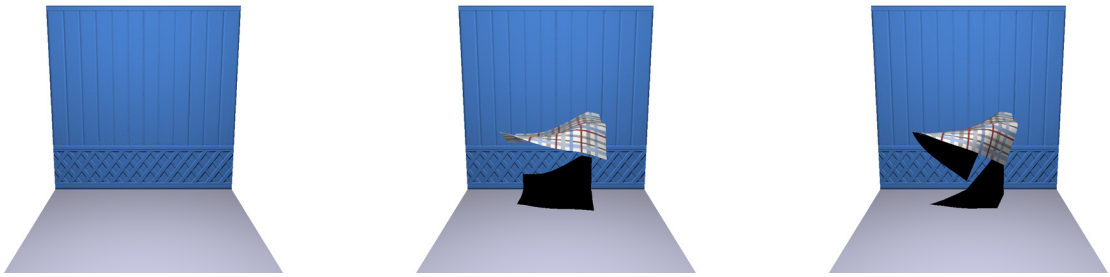

Fig. 1. Screen captures of (left) the first frame and (middle and right) two frames after certain numbers of simulation steps.

2 Cloth mesh generation

Your first mission is to create a simple rectangular cloth mesh. You can see the member function of the `Mesh` class in `Mesh.h`; there exists a function declaration `addCloth`, which takes a resolution `rx` × `ry` and a size `w` × `h` of the mesh. You should create a triangular mesh that consists of `rx` × `ry` vertices and is laid on the xz-plane.

3 Time integration

To make your cloth mesh move, you have to implement the main time integration scheme. This should be written in the `PbdSolver::step()` function. The algorithm is shown in **ALGORITHM 1**. You should refer to the lecture slides or original PBD paper for details.

Before going into the `PbdSolver::step()` function, you should initialize essential variables at the `PbdSolver::initSim()` function. For now, you can ignore constraints, which will come soon.

ALGORITHM 1: Position-based dynamics

for *all vertices i* **do**
  initialize $\mathbf{x}_i = \mathbf{x}_i^0$, $\mathbf{v}_i = \mathbf{v}_i^0$, $w_i = 1/m_i$;
end
for *simulation steps* **do**
  for *all vertices i* **do**
    $\mathbf{v}_i \leftarrow \mathbf{v}_i + \Delta t w_i \mathbf{f}_{\text{ext}}(\mathbf{x}_i)$;
  end
  `dampVelocities`$(\mathbf{v}_1, ..., \mathbf{v}_N)$;
  for *all vertices i* **do**
    $\mathbf{p}_i \leftarrow \mathbf{x}_i + \Delta t \mathbf{v}_i$;
  end
  for *all vertices i* **do**
    `generateCollisionConstraints`$(\mathbf{x}_i \rightarrow \mathbf{p}_i)$;
  end
  for *solver iterations* **do**
    `projectConstraints`$(C_1, ..., C_{M+M_{coll}}, \mathbf{p}_1, ..., \mathbf{p}_N)$;
  end
  for *all vertices i* **do**
    $\mathbf{v}_i = (\mathbf{p}_i - \mathbf{x}_i)/\Delta t$;
    $\mathbf{x}_i = \mathbf{p}_i$;
  end
  `velocityUpdate`$(\mathbf{v}_1, ..., \mathbf{v}_N)$;
end

4 Constraints projection

If your time integration code works as expected, you should implement the heart of PBD, i.e., constraint projection. Here, you will try three important types of constraint: Attachment, stretch, and bending.

The different constraints are structured in each *struct* that inherits the `Constraint` struct. Please see the `structs`, `Constraint`, `ConstraintAttach`, `ConstraintStretch`, and `ConstraintBend`. As a straightforward example, the project function of the attachment constraint is already implemented, and you should implement the others.

4.1 Attachment

The projection of this attachment has been already implemented. However, you have to initialize a set of attachments in the `initSim()` function. You can attach either one row of vertices of your mesh or randomly selected vertices.

4.2 Stretch

Similar to the attachment constraint, you should implement the `project()` function that takes the arrays of vertices positions `x` and inverse masses `w` as inputs and projects/updates two corresponding vertices' positions according to the given stretch constraint.

4.3 Bending

There is nothing special for bending. You will project/update four corresponding vertices' positions.

5 Extensions

There is no limitation of this exercise. You are strongly encouraged to further investigate any extensions you are interested in. You can find a list of potential directions in the following:

- Handling other meshes; it may need loading an obj file.
- Handling other types of constraints such as collisions
- Handling multiple objects
- ...