Practical 3: Finite-Element Soft-Body Deformation

Handout date: 17/Mar/2023
Deadline: 06/Apr/2023 08:59

1 Introduction

The third practical is about simulating soft bodies with the finite-element method learned in class, where the material is covered in Topic 9 and its associated lecture notes (about soft-body simulation). The system will handle basic collision constraints for you, and you will not be tested on them in the basic version.

The objectives of the practical are:

1. Implement the full linear finite-element dynamic equation integration, to create soft-body internal forces on a tetrahedral mesh.
2. Extend the framework with some chosen effects.

This is the repository for the skeleton on which you will build your practical. Using CMake allows you to work and submit your code in all platforms. This is essentially the same system as in the previous practical, in terms of compilation and setup.

2 Background

The practical runs in the same time loop (that can also be run step-by-step) as the previous practical. The objects are not limited to convex ones; any triangle mesh can be given as input. The software automatically converts a surface OFF mesh into a tetrahedral mesh using the library Tetgen. A tetrahedral mesh can be provided by the scene files with the .MESH format. There are no ambient forces but gravity in the basic version.

The first considerable difference in the representation of the mesh from the previous practicals is that `origPositions` replaces `origV` as a big vector in the ordering of $xyzxyz$ (3 number per vertex, so the total size of `origPositions` is $3\,|V|$), instead of a matrix where each vertex' coordinates is placed in a separate row. This is because you will need this form for the FEM formulation. The rest of the code, including constraints resolution, is adapted to this. Moreover, the scene keeps a huge vector of all stacked coordinate vectors of all meshes, to make the constraint resolution a flat process that does not distinguish meshes. You do not necessarily have to touch these big vectors in the practical; this information is provided only to help you understand how the template works.

The second considerable difference is that every vertex now has its own volume and mass, and reacts in the world as its own independent object, for the purpose of collisions and constraints. The relationship between the different vertices of the same mesh is generally (unless constrained) done by the forces enacted on them by the FEM system. Thus, there is no more handling of center-of-mass nor inertia tensor. As an example, in a collision only the colliding vertices move by impulses; the deformation this causes will in turn generate internal forces (computed by the FEM system) that will move the entire object. Since the linear finite element method is not entirely accurate, even a correct implementation will have considerable visual scaling artifacts. This is normal; you can see it in the demo.

The objects move in time steps by altering the position of each vertex of the tet mesh, where the deformation is computed by solving the finite-element equation for movement as learnt in class (Topic 9). For this, you will set up the mass M, damping D, and stiffness K matrices at the beginning of time, and only again whenever the time step-size changes. Moreover, you will set up the Cholesky solver that factorizes the left-hand side A of the entire system **once** in every **change** of time step (so it might be only once in the beginning of time unless you are keen to play with Δt).

Note that every vertex has one velocity; this practical will not explicitly model rigid-body behavior with angular velocity in the basic setting.

3 The Time Loop

The algorithm will perform the following within each time loop:

1. Integrate velocities and positions from external and internal forces (the finite-element stage).
2. Collect collision and other constraints if active.
3. Resolve velocities and then positions to valid ones according to the constraints.

4 Finite-Element Integration

Finite-element integration consists of two main steps that you will implement:

4.1 Matrix construction

At the beginning of time, or any change to the time step, you have to construct the basic matrices M, K and D, and consequently the left-hand side matrix $A = M + \Delta t D + \Delta t^2 K$. Those all have to be sparse matrices (matrices with very few non-zero elements, represented in Eigen by the datatype `Eigen::SparseMatrix<double>`). Then, you can see the Cholesky decomposition code preparing the solver. This is done in the function `createGlobalMatrices()`. Quite a few steps are required for constructing these matrices. The lecture notes (chapter 7) provide detailed descriptions of these steps. We advise you use these to your advantage.

4.2 Working with sparse matrices

When adding values to sparse matrices, you should **not** try to change the matrix values at specific indices directly. This is for performance reasons. Instead, use Triplets. A triplet is an Eigen datatype that holds the indices i, j of the element in the matrix to which you want to write and the value you want to write. You can create a triplet like this:

```
Triplet<double> trip = Triplet<double>(i, j, value);
```

You can make a `std::vector` of multiple triplets, one for each of the values you want to write to some index of the matrix. If we call this vector `tripletVector`, and the matrix you want to write to `M`, then you can use `tripletVector` to apply the changes to `M` in bulk like so:

```
M.setFromTriplets(tripletVector.begin(), tripletVector.end());
```

4.3 Integration

At each time step, you have to create the right-hand side from current positions and velocities, and solve the system to get the new velocities (and consequently positions). This is in the usual function `integrateVelocities()`, calling the `solve()` function of the Cholesky solver. As this function only changes right-hand side, it should be quite cheap in time. Note: this is only true if you don't rebuild your M, D, K matrices each time step, so be sure that you only do it during the first timestep (or when the timestep length has been changed).

The scene file contains the necessary material parameters: Young's Modulus, Poisson's ratio, and mass density. You need to produce the proper masses and stiffness tensors from them. Damping parameters α and β are given as inputs to the function, and hardcoded as in `main.cpp`.

4.4 Extensions

The above will earn you 70% of the grade. To get a full 100, you must choose 1 of these 3 extension options, and augment the practical. Some will require minor adaptations to the GUI or the function structure which are easy to do.

1. Other than FEM, constrain the boundary edges of the mesh (the edges on the faces) to have flexiblity bounds as constraints, so they do not stretch or compress more than these bounds. **Level: easy**.
2. Add functionality to apply random user-prescribed deformations of objects. For instance, instantaenous "squeezing" of an object to see it deform back after a time. **Level: easy-intermediate**.
3. Implement the corotational elements method. This requires using a solving method that does not rely on matrix factorization, such as conjugate gradients. This is available through Eigen, but you'll have to figure out the details and proper initialization. **Level: intermediate-hard**.
4. Tweak the visual artifacts caused by the fact that only the touching vertices collide directly. Do this by sending impulses to the entire body, but weaker than the ones the colliding vertex gets. For instance, use some inverse distance weighting to send velocity impulses to the other vertices. **Level: easy**.

You may invent your own extension as substitute to **one** in the list above, but it needs approval on the Lecturer's behalf **beforehand**.

5 Installation

This installation is exactly like that of Practical 1 and 2, repeated here for completeness

The skeleton uses the following dependencies: libigl, and consequently Eigen, for the representation and viewing of geometry, and libccd for collision detection. The libigl viewer uses or the menu. Everything is bundled as submodules that automatically install with cmake, so that installation should be straightforward.

5.1 Windows

On a Windows machine you can use cmake-gui to compile the skeleton. Create a folder `build` inside the practical root directory, i.e. the INFOMGP-Practical3-master folder in which this readme file is stored. After downloading cmake-gui, enter the path to the root directory in the field labelled `Where is the source code`. In the field `Where to build the binaries` paste the path to the build folder you created. Pressing `Configure` twice and then `Generate` will generate a Visual Studio solution in which you can work. After opening the solution, remember to set the startup project to `Practical3_bin`, or the project will not run. Note: it only seems to work in 64-bit mode. 32-bit mode might give alignment errors.

5.2 MacOS / Linux

To compile the environment, go into the 'practical3' folder and enter in a terminal:

```
bash
mkdir build
cd build
cmake -DCMAKE_BUILD_TYPE=Release ../
make
```

6 Using the dependencies

You do not need to acquaint yourself much with any dependency, nor install anything auxiliary not mentioned above. For the most part, the dependencies are used in components of the code that are not a direct part of the practical. The most significant exception is Eigen for the representation and manipulation of vectors and matrices. However, this package has quite a shallow learning curve. It is generally possible to learn most necessary aspects (multiplication of matrices and vectors, intialization, etc.) just by looking at the existing code. However, it is advised to go through the "getting started" section on the Eigen website (reading up to and including "Dense matrix and array manipulation" should be enough).

7 The coding environment for the tasks

The GUI is very similar to the one you used in the previous practicals.
Most of the action, and where you will have to write code, happens in `scene.h`. The functions in `scene.h` implement the time loop and the algorithmic steps, while `constraint.h` implements the `Constraint` class that evaluates constraints and resolves them.
The code you have to complete is always marked as:

```
/***************
TODO
***************/
```

For the most part, the description of the functions will tell you what exactly you need to put in, according to what you need to implement as described above.

7.1 Input

The input is the same as in previous practicals; you will notice that there are no user constraints in the given data, since we focus on the FEM part. You can therefore use `emptyconstraints.txt` as the constraints file.

The program is loaded by giving two TXT files as arguments to the executable (three arguments in total; the first being the data folder again): one describing the scene, and another the user-prescribed distance constraints. It is similar to the format in the first practical, but with extensions. That means that there is no backward compatibility; if you made your own data files, you will have to modify them slightly to make them work in this practical.

The format of the scene file is:

```
#num_objects
object1.off/mesh     density1   young1  poisson1     is_fixed1    COM1     o1
object2.off/mesh     density2   young2  poisson2     is_fixed2    COM2     o2
...
```

Where:

1. `objectX.off/mesh` - either an OFF file in the same manner as the first practical, or a tetahedral MESH file. OFF files will be automatically tetrahedralized upon loading, so you don't have to spend the effort creating tetrahedral meshes offline.
2. `density` - the uniform density of the object.
3. `Young` - Young's modulus, used to create the stiffness tensor. Usually contains big values; look up values for general materials, which are usually expressed in Mega or Giga Pascals. The value you give here will be interpreted as Pascal.
4. `poisson` - Poisson's ratio, which describes the incompressibility of the object. Values should be between $[0, 0.4999...]$—note that using exactly 0.5 will produce infinite stiffness that will kill your simulation by dividing by zero, so avoid it.
5. `is_fixed` - Specifies whether the object should be immobile (fixed in space) or not.
6. `COM` - The initial position in space where the object would be translated to. That means, where the COM is at time $t = 0$.
7. `o` - The initial orientation of the object, expressed as a quaternion that rotates the geometry to $o{*}object{*}inv(o)$ at time $t = 0$.

The constraints file has the following format:

```
#num_constraints

m1 v1 m2 v2
```

This results in distance constraints between vertex `v1` on mesh `m1` to vertex `v2` on mesh `m2` in their initial configuration (after COM+orientation transformations). As was said above, you don't need to use constraints. Loading the file `emptyconstraints.txt` will keep your scene void of constraints.

7.2 Indexing

We use two indexing methods for vertices and coordinates:

1. `Local indexing`: the vertices in each mesh are kept in one big column vector of $3\,|V|$ coordinates in $x_1y_1z_1x_2y_2z_2x_3y_3z_3$... format (dominant order by vertices and subdominant order by coordinate). In general, corresponding quantities like velocities, forces, and impulses will be encoded the same way. Quantities that are the same for all xyz coordinates of a single vertex (like mass, Voronoi volumes, etc.) are encoded in a vector of $|V|$ with one quantity per vertex. The comments in the code will tell you which is which.

2. `Global indexing`: this is the indexing used by the scene class, which aggregates all vertex-coordinates in the scene. The index vector contains **all** coordinates in the scene in the same format as the local indexing (so $x_1y_1z_1x_2y_2z_2x_3y_3z_3$...). As such, the individual coordinates of every mesh together form a continuous segment within this global vector. The `globalOffset` member variable in each mesh indicates on which index this segment begins. The vectors containing impulses, velocities, and inverse masses have a similar format. The scene class contains a variable `globalInvMasses` which stores the inverse mass of each vertex three times in a row (so $m_{inv1}m_{inv1}m_{inv1}m_{inv2}m_{inv2}m_{inv2}$...). The mesh class, on the other hand, only stores one instance of the inverse mass per vertex. That is because the scene class handles the constraints resolution, for which the format of 3-times repeating vertex masses is much more comfortable (for every vertex, each coordinate of the vertex then has its own copy of the vertex' inverse mass). At any rate, check the comments next to the variable definitions.

Two functions: `global2mesh()` and `mesh2global()` update the values back and forth between the scene and the individual meshes.

8 Submission

All the files of the practical that you have modified to achieve your solution are to be submitted in a single zip file on blackboard. Note that this does not include any visual studio solution- or project files; only the c++ header files are to be submitted. The deadline is Thursday 06 April at 09:00 AM. Students who have not submitted the practical by that time will not be checked in the session.

You are highly encouraged to do the practical in pairs. Doing it alone requires permission beforehand by the lecturer, and should come with the warning that the work load for a one person team may be quite significant.

The practical will be checked during a checking session on Thursday 6 April. There will be no lecture that day. Every pair will have 10 minutes to shortly present their practical, and be tested by the lecturer with some fresh scene files. In addition, the lecturer will ask each team member a short question that should be easy to answer if this person was fully involved in the practical.

For the demonstration please bring a computer with an operating executable of your practical solution, compiled beforehand in release mode, and working on all given scene files.

Note: In order to be able to give everyone sufficient time to demonstrate their code, the checking times will be strict. If you cannot come with your own computer, tell the teaching assistant in advance, and your code will be compiled on the lecturer's computer beforehand.

The registration for time slots is to be done in a spreadsheet that will be posted on Teams well before the deadline. You are not obligated to write your own explicit names—if you do not wish to do so, just write "occupied" and tell the teaching assistant via e-mail or Teams PM who you are and in which slot you want to present. Please do not change other people's time choices without their consent.

Good luck!

If you have questions about the practical you can post these in the Practicals channel on Teams. The lecturer and teaching assistant will try to respond as quickly as possible, but you are also encouraged to respond to each other's posts if you think you can help out.