Practical 1: Rigid-Body Simulation

Handout date: Thursday 9 February.

Deadline: Thursday 2 March at 09:00AM.

The first practical is about simulating the motion of rigid bodies, and their interactions through collision resolution. The objectives of the practical are:

1. Implement rigid velocity, position, and orientation integration in a discrete-time iteration.
1. Implement interpenetration resolution.
2. Implement impulse-based collision resolution.
3. Extend the framework with some chosen effects.

This is the repository for the skeleton on which you will build your first exercise. Using CMake allows you to work and submit your code in all platforms. The entire environment is in C++, but most of the "nastier" coding parts have been simplified; for the most part, you only code the mathematical-physical parts.

Scope

The basic scenario is limited in scope to convex objects, and includes no air resistance or friction. The environment loads scenes that describe a set of objects, and their positions in space. The world contains a big, immobile (fixed) plate on which the objects fall. There are no ambient forces in the basic setting other than earth's gravity, pulling towards the negative y axis.

The environment is already configured to run in a time loop, where it detects collisions in each step. Collisions are assumed to be with a single point of contact per object per time-frame, and in case of more, one point is chosen (this might lead to mildly non-physical behavior).

The practical includes the following basic mandatory requirements:

1. For every time step, integrate the accelerations (linear and angular) into velocities, and the velocities into positions and orientations. Use a *Semi-implicit Euler scheme*, as learned in class (Topic 4)—first integrate velocity, then position with the new velocity. This requires changing both the position of the COM by the linear velocity, and the orientation by the angular velocity (Topic 2). The time step difference Δt is given by the GUI, and controllable by the menu.
1. For every time step, resolve interpenetration (Topic 3) *linearly*. The means, given the penetration point, depth, and normal, move the objects apart linearly so they are only tangent (touching exactly without overlapping). You may assume there is a

single point of contact; you are not required to solve multiple interpenetrations in the iterative manner.

2. For every time step, resolve the collision by assigning opposite impulses to two colliding objects, and correcting their velocities instantaneously (Topic 3). This will change both the linear and the angular velocities. The environment computes the inverse inertia tensor for you, given in the original orientation of the object (as appears in the file), and around its COM. You are responsible to transform the inverse inertia tensor to what is needed upon the moment of collision.

See below for details on where to do all that in the code.

Extensions

The above will earn you 70% of the grade. To get a full 100, you must choose 2 of these 6 extension options, and augment the practical. Some will require minor adaptations to the GUI or the function structure which are easy to do. Each extension will earn you 15%, and the exact grading will commensurate with the difficulty. Note that this means that all extensions are equal in grade; if you take on a hard extension, it's your own challenge to complete it well.

1. Add low-velocity drag forces in the air (Topic 1). You should use the *total velocity* (the entire velocity of a point from linear and angular velocity), and a drag coefficient which is controllable by the user. **Level: easy**.
1. Add friction to the collision impulses (Topic 3), again with a user-controllable coefficient. **Level: easy**.
2. Change the time-integration system to one of the more sophisticated methods learned in class. The improvement should be exemplified with a scene file that can demonstrate that your implementation is superior to the basic method. **Level: intermediate**.
3. Resolve interpenetration with an additional rotational movement. For that you will have to decide on a good heuristic. I suggest to read chapter 14 in Ian Millington's Game Physics Engine Development. **Level: intermediate**.
4. Add a possibility in both the interface, and the program, for objects to start with some initial velocity (like throwing stuff around) **Level: easy**. Possible further extension: allow to "push" an object with an artificial force/impulse. **Extension level: easy-intermediate**.
5. Resolve multiple interpenetrations and collisions. It is not easy to demonstrate, and you will have to build a scene file that proves you did the job correctly. This should involve many objects tightly packed. **Level: hard**.

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

Installation

The skeleton uses the following dependencies: libigl, and consequently Eigen, for the representation and viewing of geometry, and libccd for collision detection. libigl viewer is

using dear imGui for the menu. Everything is bundled as either submodules, or just incorporated code within the environment, so that installation should be straightforward.

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-Practical1-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 twice `configure` 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 `practical1_bin`, or the project will not run. *Note*: it only seems to work in 64-bit mode. 32-bit mode might give alignment errors.

MacOS / Linux

To compile the environment, go into the `INFOMGP-Practical1-master` folder and enter in a terminal:

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

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 parts of code that are background, or collision detection code, which is not a direct part of the practical. The most significant exception is Eigen for the representation and manipulation of vectors and matrices. However, it is a 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).

Working with the repository

All the code you need to update is in the `practical1` folder. **WARNING: Do not make your solutions public online!** Solutions to the practical which are made publicly accessible will disqualify the students.

The coding environment for the tasks

Most of the action happens in `scene.h`. The main function is:

```
void updateScene(double timeStep, double CRCoeff){

    //integrating velocity, position and orientation from forces and previous
```

```
states
    for (int i=0;i<meshes.size();i++)
      meshes[i].integrate(timeStep);

    //detecting and handling collisions when found
    //This is done exhaustively: checking every two objects in the scene.
    double depth;
    RowVector3d contactNormal, penPosition;
    for (int i=0;i<meshes.size();i++)
      for (int j=i+1;j<meshes.size();j++)
        if (meshes[i].isCollide(meshes[j],depth, contactNormal, penPosition))
          handleCollision(meshes[i], meshes[j],depth, contactNormal,
penPosition,CRCoeff);

    currTime+=timeStep;
  }
```

The two most important functions are `integrate()` and `handleCollision()`. They all contain a mixture of written code, and code you have to complete. The code you have to complete is always marked as:

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

The description of the function will tell you what exactly you need to put in.

Input

The program is loaded by providing a TXT file that describes the scene as a commandline argument to the executable. Note that in Visual Studio you can add command line arguments under `Project > Properties > Configuration Properties > Debugging > Command Arguments`

The TXT file should be in the `data` subfolder, which is automatically discovered by CMake. The format of the file is:

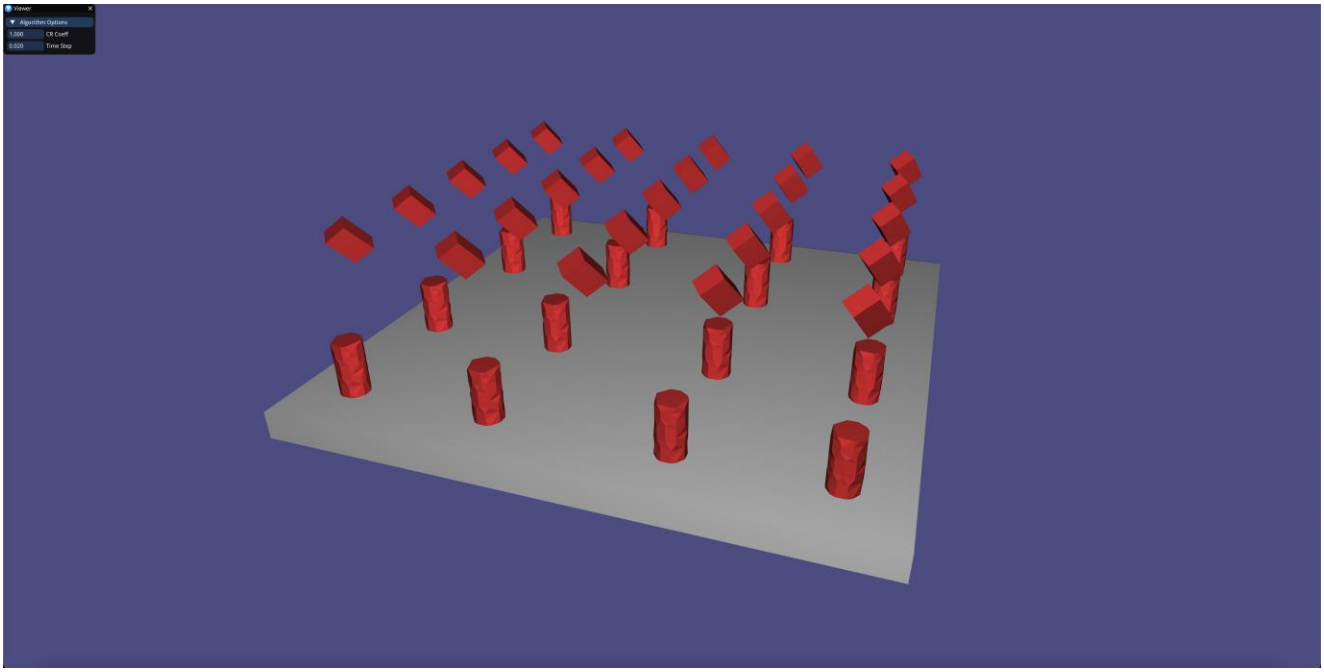
```
#num_objects
object1.mesh  density1  youngModulus1 PoissonRatio1 is_fixed1    COM1     q1
object2.mesh  density2  youngModulus2 PoissonRatio2 is_fixed2    COM2     q2
.....
```

Where:

1. `objectX.mesh` - an MESH file (automatically assumed in the `data` subfolder) describing the geometry of a tetrahedral mesh. The original coordinates are translated automatically to have $(0,0,0)$ as their COM.
1. `density` - the uniform density of the object. The program will automatically compute the total mass by the volume.
2. `is_fixed` - if the object should be immobile (fixed in space) or not.

3. `COM` - the initial position in the world where the object would be translated to. That means, where the COM is at time $t = 0$.
4. `q` - the initial orientation of the object, expressed as a quaternion that rotates the geometry to $q * object * q^{-1}$ at time $t = 0$.
5. `youngModulus1` and `PoissonRatio1` should be ignored for now; we will use them in the 3^{rd} practical.

User interface



The viewer presents the loaded scene, and you may interact with the viewing with the mouse: rotate with the left button pressed and moving around (the "[" and "]" buttons change the behaviour of the trackball), zoom with the mousewheel, and translate with the right button pressed and dragging. Some other options are printed to the output when the program starts.

The menu also controls the visual features, and the setting of the coefficient of restitution and the time step. They can be updated at any point in the simulation. You might add more parameters with some extensions. Everything is set up in `main()`.

The simulation can be run in two modes: continuously, toggled with the `space` key (to stop/run), and step by step, with the `S` key. This behavior is already encoded. The visual update of the scene from the objects is also already encoded.

Data structure

There are two main classes, `scene`, and `Mesh`. Both are in `scene.h`, and will be updated by you. They are commented throughout, so their individual properties are understood. Each mesh, and the platform, are rigid bodies of their own. The geometry is encoded as follows:

```
MatrixXd origV;   //original vertex positions, where COM=(0.0,0.0,0.0) -
never change this!
MatrixXd currV;   //current vertex position
```

`origV` and `currV` are $|V| \times 3$ matrices encoding all the vertices of the mesh, row by row. You should **never update `origV`**. `currV` should be updated to reflect the result of every time step, and this is what you see on screen.

Quaternions represent orientations and rotations, where if the neutral (initial) orientation of a vector is v, and the orientation quaternion is q, then the final orientation is qvq^{-1}. The `QRot` function in the `auxfunctions.h` file implements that (and several other functions for quaternions are available in that file). Note the function `Q2RotMatrix` that produces the rotation matrix corresponding to that orientation, which should be used for the transformation of the inertia tensor.

Existing software components

You do not have to compute the entire algorithmic environment from scratch. The things that you are given are:

1. Collision detection, as explained above.
1. A function `initStaticProperties` computes the original COM and the inverse inertia tensor for each original MESH files, and is called by the `Mesh` constructor. You do not need the COM it computes; the constructor translates the object (`origV` coordinates) to the origin, so it always has $COM=(0,0,0)$. The constructor also initializes `currV` as a translation and rotation of `origV` to fit the prescribed values from the scene file.

The inverse inertia tensor you get from `initStaticProperties` is **not after applying the orientation, not even that in the scene file**. That is, what you get is the inverse inertia tensor of `origV` around its COM. You will have to compute the inverse inertia tensor for a given `currV` according to its current orientation, and it is always then around the COM of the moving object. See Topic 2 for how to do that efficiently, and be careful to apply the correct rotation!

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 2 March at 09:00AM**. 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 **Tuesday 7 March**. 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.

Frequently Asked Questions

Here are detailed answers to common questions. Please read through whenever ou have a problem, since in most cases someone else would have had it as well.

Q: I am getting "alignment" errors when compiling in Windows. A: Delete everything, and re-install using 64-bit configuration in `cmake-gui` from a fresh copy. If you find it doesn't work from the box, contact the Lecturer. Do not install other non-related things, or try to alter the cmake.

Q: Why is the demo not working out of the box? A:: with the same parameters as your input program: infomgp_practical1 "folder_name_without_slash" "name of txt scene files".

Q: How do I do inverse mass weighting for linear interpenetration? A:: Given two objects with masses m_1, m_2 interpenetrating in mutual distance d, their individual corrections need to be $d_1 = \frac{d \cdot m_2}{m_1 + m_2}$ and respectively for d_2. Note that: 1) the lighter object needs to move more (that's why it's *inverse* mass weighting) 2) if one object is fixed, it's like it has infinite mass, and the other object moves the entire of d back alone.

Q: cmake fails to clone external dependencies (like glad) although they exist. What is the problem? A: This is a rare bug that would suggest some SSL issues with the specific computer. To counter this, clone instead the `dev` branch of `libigl` independently into the libigl folder, download all the external libraries libigl needs (eigen, glad, glfw, imgui and libigl-imgui) manually, and add them to the external folder. Finally, set the LIBIGL_SKIP_DOWNLOAD variable to ON (`cmake -DLIBIGL_SKIP_DOWNLOAD=ON ..`), then proceeded to run CMake normally from there.

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.