Week 10 — *Ping-pong implementation and overall optimization of program*

The goal of the present exercise is to extend the particle code to handle contact between spheres, to profile the code and subsequently optimize its critical parts.

Exercise 1: *Implementation of contact forces*

We want to extend the code to handle contact forces between balls themselves, but also between balls and a boundary box. The contact forces between any two ping pong balls should follow the formula:

$$\boldsymbol{f}_{ij} = \frac{\boldsymbol{r}_{ij}}{r_{ij}} \beta p(i,j) \tag{1}$$

where β is a penalty parameter of the contact interpenetration and $p(i,j)$ is the interpenetration/gap function defined as:

$$p(i,j) = \begin{cases} (R_i + R_j) - r_{ij} & \text{if } r_{ij} < R_i + R_j \\ 0 & \text{otherwise} \end{cases} \tag{2}$$

where R_i is the radius of particle i. The contact with the boundary is handle in a simpler manner: if the particle goes past a domain bound (say $x_{\max}$), then the sign of the corresponding velocity component is flipped ($v_x := -v_x$, similar to light reflection in optics).

1. Study the new class diagram (use *doxygen* to generate it)
2. Implement the factory for `PingPongBalls`
3. Implement the method `ComputeContact::computePairInteraction()` (hint: you should use the `static_cast` instruction)
4. We introduce a new class called `ComputeBoundary` in charge of computing the contact interaction with borders. What are the advantages of using another class and not putting the code directly in `ComputeContact`?
5. What set of units would you like to use?
6. Modify the python script provided to generate N particles with a radius of your choice.
7. Launch the code over this input file and observe the behavior of the ping-pong balls.

Exercise 2: *Code profiling*

1. Install the *perf* tool:

```
sudo apt install linux-perf
```

2. The *perf* tool needs special permissions from the kernel:

```
sudo bash -c 'echo -1 > /proc/sys/kernel/perf_event_paranoid'
```

3. Add the option `-fno-omit-frame-pointer` to the compilation options (CMake variable `CMAKE_CXX_FLAGS`)
4. Generate 1000 particles with the provided python script.

5. Launch a planets simulation with the following command:

```
./particles 500 1 <your_input.csv> planet 2e-9
```

6. (Optional) Observe the result of the simulation with Paraview. You should see the collapse of a cluster of stars.

7. Launch the code with:

```
perf record -g ./particles 500 1 <your_input.csv> planet 2e-9
```

   See question 2 if you get an error message. You can permanently deactivate the protection to kernel events by following the instructions of the error message.

8. Browse the call graph:

```
perf report -G
```

   What class do you think is the most important for performance?

9. Compile the code in `Release` mode (CMake variable `CMAKE_BUILD_TYPE`)

10. Browse the call graph: why does the main take up only about 60% of the cpu time? Change the program arguments to fix this.

11. Look at the assembly code for `computePairInteraction` (press `a` when selecting the function in *perf*) and take note of the `callq` instructions.

Exercise 3: ***Code optimization***

We now want to improve the performance of the code.

1. Change the input parameters to get a runtime of approximately 30s (you can measure the time with `perf stat ...`)
2. Apply what you learned in class to optimize the code (inline functions, etc.), using the information you gathered in the previous exercise

Exercise 4: ***(Optional) Adding gravity and dissipative forces***

1. Make a class named `ComputeVerticalGravity` that adds the vertical gravitational acceleration to all particles ($F = mg$).
2. Make a class named `ComputeDissipativeForces` which adds a frictional force to all particles.
3. Run the code again. Vizualize the output with Paraview. Generate a video.