Game Physics – Programming Exercise

Exercise 1 – *Mass Spring System*

Task Overview

The exercise consists of four tasks that should be completed: First, take a paper and calculate a single time step for the mass-spring system below (Table 1.1) by hand with the Euler and the midpoint method, understand the difference between these two integration methods.

Then, based on the template project, implement a simulator for 3D mass-spring systems. The implementation should contain different time integration methods.

Set up the mass-spring system in Table 1.1 in your simulator. As a validation test, run this setup for one time step (with Euler and midpoint) and compare the result with the previous (manually calculated) answer.

Afterward, set up a complex demo scene to show how your simulator works.

Table 1.1 - A basic 3D Mass-Spring System

Two mass points are connected with a single spring.
The points have the following initial positions and velocities:
$p_0 = (0, 0, 0)^T$, $v_0 = (-1, 0, 0)^T$
$p_1 = (0, 2, 0)^T$, $v_1 = (1, 0, 0)^T$
The points have masses $m_0 = m_1 = 10$, and spring length $L = 1$, with stiffness $k=40$.
Assume that there are no velocity damping or gravity forces.

Demo requirements:

Your submission should contain the following demos:

- Demo 1, a simple one-step test:
  - Manually calculate the solution to the 2-point mass-spring setup above, with the parameters given there for **Euler** and **midpoint** (no need to submit this) for a time step $h = 0.1$.
  - According to the class definition in “MassSpringSystemSimulator.h” file, implement your “MassSpringSystemSimulator” class. Your class should have the two integration methods.
  - In Visual Studio, set the “simulationsRunner” as the startup project. In addition, in main.cpp, replace “#define TEMPLATE_DEMO” with “#define MASS_SPRING_SYSTEM” at line 23. Then you can run and test your MassSpringSystemSimulator class.

  - Build and run the basic test case, and print the solution (i.e., new position and velocity for both points) after one time step to the command line. (Hint: don't be surprised if the differences between both methods are quite small – the important thing is to note how they're different.)

- Demo 2: a simple Euler simulation:

  Simulate the 2-point setup from Demo 1 with the **Euler** method and display the results for a time step h = 0.005.

- Demo 3: a simple Midpoint simulation:

  Simulate and display the 2-point setup from Demo 1 with the **midpoint** method (also set the time step h to 0.005).

- Demo 4: a complex simulation, compare the stability of Euler and Midpoint method:
  - Set up a simulation with at least 10 mass points and 10 springs.
  - Simulate it. Allow users to change the method (Euler or Midpoint) and the time step interactively in the UI.
  - Provide methods for interaction, include gravity, and add collisions with the ground floor (or walls).
- Optional Demo 5: additionally implement the **Leap-Frog** method.

Submission

Register your group via the Moodle activity until 5 November 23:59.

Please pack all your source files (.h and .cpp) and project files (.vcxproj) under the "Simulations" directory into a zip file, and name it "Group??_Ex1_VS20??.zip". Then, upload this file in the corresponding Moodle submission activity. Ensure not to include the compiler temporary files (they will be under the Simulations/Win32/ directory). **Your package should be smaller than 100kb.**

The deadline for this exercise is on **19 November at 23:59**. Please submit all your files only via Moodle. **Late submissions and submissions via email are not allowed.**

Recommendations & Tips:

1. **Ensure you can run the template project.** Download/git clone the template project from https://github.com/tum-pbs/gamephysicstemplate
2. **Mode switch.** Take a look at the TemplateSimulator. It is an example of implementing a simple simulator in the given framework. It has a drop-down list to switch among different scenes. Use this switch to toggle between the four different demos below.
3. **Boundary conditions.** To realize simple obstacles like planes or spheres, you must enforce that all points of your mass-spring system lie on the outside of all obstacles at all times. i.e., after each position update, correct invalid point positions: If a point penetrates an obstacle, move it along the surface normal back to the outside/surface of this obstacle.
   Other possibilities to influence your simulation are, e.g., to fix specific points (do not move them at all) or to artificially modify the direction and intensity of the gravity force.
4. **Interaction.** For the last demo task, you can provide methods for user interaction, e.g., through interactive parameter changes or mouse/keyboard input. Feel free to experiment…
5. **Stability.** The main difference between the different integration methods is stability. In the last demo task, experiment with different settings and spring setups to explore when and how your simulations remain stable.
6. **Rendering.** There are some tips below about how to use the DrawingUtilitiesClass class. The TemplateSimulator is also a good example of your own implementation.

Tips for Rendering

To make you focused only on the simulation programming, we offer a set of drawing functions in the `DrawingUtilitiesClass`. In the mass-spring system, you can use the DUC to draw spheres and lines to represent your setup.

To draw a **sphere**, please first set up the **lighting** by calling:
`DUC->setUpLighting(Vec3 EmissiveColor, Vec3 SpecularColor, float SpecularPower, Vec3 DiffuseColor);`
where `EmissiveColor` defines the color emitted (use `Vec3()` for non-emissive material), `SpecularColor`, `SpecularPower` defines the specular material, and `DiffuseColor` defines the diffuse material. All the colors are represented in RGB format (`Vec3`). Each color channel ranges from 0 to 1, e.g., the color green can be represented as "`Vec3(0, 1, 0) // Green`".

Then, draw the actual sphere by calling `DUC->drawSphere (Vec3 pos, Vec3 scale);`
where `pos` defines the sphere's center, and `scale` defines the size of it.

To draw a **line**, you need to first notify the system that you will start drawing a line by calling
`DUC->beginLine ();`
Then, draw the actual lines by calling: `drawLine (Vec3 pos1,Vec3 color1, Vec3 pos2,Vec3 color2);`
where `pos1`, `color1` define the position and color of the starting point of the line, and `pos2`, `color2` define those of the ending point.

After finishing all the lines, finally notify the system by calling:
`DUC->endLine ();`

The template project has more examples. You can run and test it to get a better understanding of the DUC class.