TU BRAUNSCHWEIG
Prof Dr. Marcus Magnor
Institut für Computergraphik
Stephan Wenger (wenger@cg.tu-bs.de)

October 30, 2013

Physikbasierte Modellierung und Simulation
Assignment 1

Present your solutions for this sheet in the exercise on Wednesday, November 6, 2013. The exercises are going to take place in the CIP pool, room G40 in Mühlenpfordstraße 23. Please make sure your solutions compile and run on the CIP pool computers. Note that you need a y-number, which can be obtained at the Gauß-IT-Zentrum, to use the computers. If for some reason you are not able to attend the exercise, you may send your solution to wenger@cg.tu-bs.de instead.
This exercise is about the simulation of particle systems. You will

- get to know a code framework for working with physical quantities,
- use this framework to formulate Newton's law of universal gravitation,
- implement a solver for the equations of motion of a system of particles, and finally
- simulate our solar system and experiment with your algorithm.

1.1 Physical quantities (15 points)

For our simulation, we will need a number of physical quantities, namely *mass*, *length*, and *time*, as well as some derived quantities, such as *velocity* (length over time) and *acceleration* (length over time squared). Some of these quantities are scalar (such as mass), some are vector-valued (such as the position of a planet in the solar system, which is a three-dimensional length). Physical quantities can only be combined in a meaningful way when their dimensions match; for example, it makes no sense to add a length to a time. You may however divide a length by a time to obtain a velocity. Checking the dimensionality of a formula – that is, making sure that all the quantities are combined in a meaningful way – is an important step in assuring the physical plausibility of your algorithm. This concept is called *dimensional analysis* (`http://en.wikipedia.org/wiki/Dimensional_analysis`).
On the course website, you will find a code framework that implements a data type for physical quantities. Please download the code framework and familiarize yourself with the `Quantity` class. Several commonly used quantities, such as `Mass`, `Length`, `Time`, `Velocity` and `Acceleration`, are already predefined, so that you can simply write

```
Mass m;
```

to create a new quantity with the dimension of a mass. For some quantities, such as length and velocity, two- and three-dimensional vectorial variants exist. These are appropriately called `Length2D`, `Length3D` and so on.
When you create a new quantity, you will have to specify the units in which you provide the initial value. In `Quantity.h`, several units are predefined, for example `kg` and `g` for mass, `m` and `km` for length, or `hours` and `days` for time. You can instantiate a new time, for example, by writing:

```
Time t = 5.0 * days + 12.0 * hours + 3.0 * minutes + 21.4 * seconds;
```

You can also provide dimensionless values (e.g. `doubles` or `floats`) as constructor parameters, which will be interpreted as the corresponding SI base units, that is, kilograms, meters and seconds, respectively. To avoid confusion, try to explicitly specify the units whenever possible.
In order to get familiar with the code framework, implement the following toy problem: An object is located at position $\vec{p}_0 = (0\,\mathrm{m}, 50\,\mathrm{m}, 0\,\mathrm{m})$ and moves with an initial velocity of $\vec{v}_0 = (10\,\mathrm{m/s}, 0\,\mathrm{m/s}, 0\,\mathrm{m/s})$ in the x direction. It is accelerated by the gravitational acceleration of $\vec{g} = (0\,\mathrm{m/s^2}, -9.81\,\mathrm{m/s^2}, 0\,\mathrm{m/s^2})$ in the negative y direction. At any given time t, the position of the object is given by

$$p = p_0 + v_0 t + \frac{1}{2} g t^2 \quad . \tag{1}$$

Create a short C++ program that computes the location of the object at $t = 3.0\,\mathrm{s}$. You will have to `#include "Quantity.h"` and you should consider `using namespace quantity;` to be able to use `Length3D`, `Velocity3D`, `Acceleration3D` and `Time` as well as the units `m` and `s`.

1.2 Gravitational forces (30 points)

The movements of the bodies of the solar system are defined by Newton's law of universal gravitation. The law states that for two particles with masses m_1 and m_2 which are located at the positions $\vec{p}_1$ and $\vec{p}_2$, the force F which the second particle exerts on the first one is given by

$$\vec{F} = G \frac{m_1 m_2}{||\vec{p}_2 - \vec{p}_1||^2} \frac{\vec{p}_2 - \vec{p}_1}{||\vec{p}_2 - \vec{p}_1||} \quad . \tag{2}$$

The acceleration $\vec{a}$ of the first particle due to the presence of the second one is then given as $\vec{a} = \vec{F}/m_1$. When multiple particles are present, the force on one particle is simply the sum of all the forces caused by all other particles.
In `GravitationalSystem.cpp`, implement the function `computeAccelerations()` which computes the acceleration for every particle in a system of particles. You may want to have a look at `Particle.h` to see which data members are available. The gravitational constant $G = 6.67428 \cdot 10^{-11}\,\mathrm{kg^{-1}m^3s^{-2}}$ is defined in `Quantity.h`.

1.3 Explicit Euler method (40 points)

The motion of the particles in the system is described by a system of linear differential equations. With particle positions $\vec{x}$, velocities $\vec{v}$ and accelerations $\vec{a}$, the time derivatives are $\dot{\vec{x}} = \vec{v}$ and $\dot{\vec{v}} = \vec{a}$.
In `EulerSolver.cpp`, implement the function `step(const Time timestep)` which computes the position and velocity of all particles after a small `timestep` based on the current position, velocity and acceleration. Use the explicit Euler method (`http://en.wikipedia.org/wiki/Euler_method`) which linearly extrapolates the current position and velocity.
The file `main.cpp` reads the sun and planets of our solar system from `solarsystem.dat`, runs a simulation of this system and, depending on the command line parameters, dumps the results into a data file or shows a simple live OpenGL visualization.

1.4 Create interesting showcases (15 points)

As soon you are able to simulate the motion of our solar system, you can move on to experiment with the algorithm. Try at least two of the following experiments:

- Attach a moon to one of the planets and try to find initial values that keep it in a stable orbit.
- Create a solar system with two suns that revolve around each other.
- Let an object with high mass and high initial velocity traverse the solar system.
- Check how small the step size has to be in order to keep the simulation stable. You will later get to know other solvers which remain stable even with larger step sizes.
- Enhance the OpenGL visualization by realistic lighting, textured planets, and a simulation time step that adapts to the actual time between two screen updates.

`http://graphics.tu-bs.de/teaching/lectures/ws1011/PBM/`