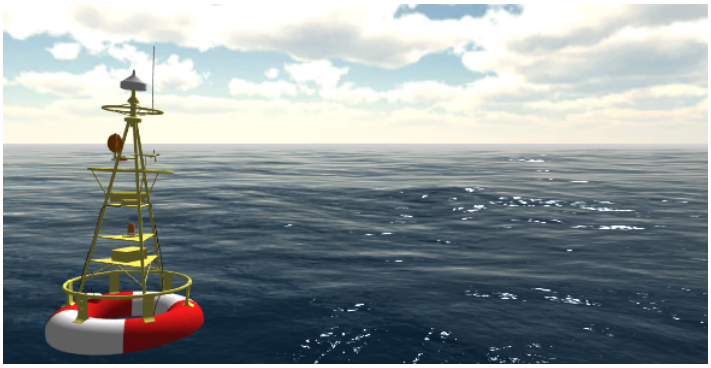
Lab 2 Spring-Mass Systems
Models and Simulation (DD1354)

January 15, 2019

Abstract

This lab is about mass-spring systems - a mathematical concept utilised in many different areas. In the lab, you will look at two different applications: Ocean Waves and Rope Simulation. The first part, ocean waves, will be quite theoretical and you will go through and understand two quite substantial implementations. The second part is lighter, and will focus on implementing parts of a rope simulation!

1 Water Simulation and Rendering



In this section we will use two different water simulations in Unity. The mathematical concepts required to understand why the simulations work are out of the scope of this course. For example, waves are tightly related to the concept of Fourier Transform. Intuitively, such tool allows us to generate waves in Spatial domain from their frequency in Fourier domain. In a way, it allows our program to generate complete waves from a discrete set of frequencies that we choose to be represented. Other methods simply use Cosines and Sines to generate realistic waves directly in the update method. You will familiarize yourself with these two concepts in the following parts.

The simulations can be found in the associated lab archive as a Unity project:

- The first of these simulations is tilted **OceanSurfaceEffects** and uses a radial grid to simulate and render the waves of the ocean. It contains a couple of objects that float on the water and a mouse controlled camera.
- The second of these simulations is tilted **PhillipsOcean** and uses a square shape grid with Fourier generated waves. It does not contain any objects, but allows more detailed control of the wave shapes and dynamics.

In each section a link to a git repository is available. To get the project, simply click the link and refer to the setup section (section 4)

In the following 2 subsections you will be asked to modify the code in order to achieve different results. This requires that (to some extent) you have read and understood the code. Remember that formulating a hypothesis about how the code works and then checking if it was indeed accurate, is a good way to get through this exercise.

1.1 OceanSurfaceEffects

The project files for this section is available at DD1354 lab materials link *Lab 2 download (part 1)*: `https://www.kth.se/social/course/DD1354/page/lab-materials/`.

Open Unity and then click in File, Open Project. Choose the folder called OceanSurfaceEffects from your directory and notice the Scripts from the Assets folder: Open *AddBouyancy* and *Ocean* in the text editor that comes with Unity.

Suppose there was an oil spill in the ocean that we are simulating. This means that a black fluid with a density[1] different than the water's is now mixed with it. Assume the water has mixed entirely with the oil, then proceed to answer the following questions and update your simulation accordingly.

Note: you can enable or disable the grid of the ocean's mesh by pressing F2.

1. Which of the fluids should be on the top and why? Please, enunciate the physical law that is acting on this scenario with mathematical notation. You can choose the type of oil of your spill[2].

2. Assume that the layer of oil has a certain depth of your choice d. Should the buoys and the box sink more inside the new fluid than before due to the change of density? If so, why? and by how much more will they sink?

3. Update your simulation so that now that color of the fluid corresponds to the oil color. To access properties of the ocean you can run the simulation and then click on the mesh to see the Inspector with the Ocean's shader.

4. Update your simulation so that now the buoys and the box sink more (on less) than before, according to the calculations that you made with the density of the new fluid and the assumed weight of the objects[3].

5. The viscosity[4] of water is also different from that of the oil. Without going into deeper mathematical modeling issues, describe what the viscosity of the fluid would be resulting from the aforementioned mixture.

6. One of the effects that the viscosity change will produce on the simulation is in the speed of propagation of the waves and their height. Modify your simulation to achieve realistic results in these 2 aspects (wave propagation speed and wave height).

7. The capability of reflecting light of water is different than that of the oil. Being more opaque, the oil tends to have lower reflectivity. Update your simulation so that the specular reflections[5] are reduced to produce more realistic results.

8. (optional) To add more realism to your simulation you can proceed to import a 3D model of an oil rig[6], and then import it into Unity[7].

[1] `http://en.wikipedia.org/wiki/Density`
[2] `http://www.engineeringtoolbox.com/liquids-densities-d_743.html`
[3] `http://en.wikipedia.org/wiki/Archimedes'_principle`
[4] `http://en.wikipedia.org/wiki/Viscosity`
[5] `http://en.wikipedia.org/wiki/Specularity`
[6] `http://www.thepixellab.net/free-c4d-3d-model-oil-rig-platform`
[7] `http://docs.unity3d.com/Manual/HOWTO-ImportObjectCinema4D.html`

9. (optional) you may use the Unity particle system to simulate fire in the oil rig, and even the oil coming out of it filling the ocean. You can use the properties of the particle system studied in the previous lab to simulate collisions between the oil stream and the surface of the contaminated ocean. How much realism you want to achieve is up to you.

1.2 PhillipsOcean

The project files for this section is available at DD1354 lab materials link *Lab 2 download (part 2)*: `https://www.kth.se/social/course/DD1354/page/lab-materials/`.

Open Unity and then click in File, Open Project. Choose the folder called PhillipsOcean from your directory and notice the Scripts from the Assets folder: Open *FourierCPU* and *Ocean* in the text editor that comes with Unity.

Suppose that we are simulating how water would behave in planets different than the earth where the simulation is based. In such planets there are different weather conditions, and the planets themselves have different masses and sizes. Assume these two planets have masses which are 100 times smaller and 100 times bigger than that of the earth, then proceed to answer the following questions and update your simulation accordingly.

Note: you can enable or disable the grid of the ocean's mesh by clicking on it in the Scene window.

1. How would the water waves look like in each of these planets, make your predictions and then modify the code accordingly to see if your hypothesis was true. Remember how to calculate the gravity of a planet based on their mass from lab1[8].

2. Suppose a storm is coming and it is starting to rain heavily. Use a particle system (or several) to simulate rain and enable the collision between the particles and the mesh to add realism to the simulation.

3. The storm has finally arrived, modify the code so that the waves reflect the effect caused by the increased wind. You should expect higher waves which also move faster.

4. Assume that the viscosity of water is different in each of these planets due to certain materials that are mixed with it. Modify the dampening parameter of the simulation and explain how it relates to the viscosity of the fluid.

5. Suppose the storm is over and now the water is calm. Modify the dispersion of the simulation to replicate such effect. Notice that now the waves are smaller and they do not seem to be displacing large amounts of water.

6. What differences do you notice between the grid of this simulation and the grid of the previous simulation. Which one do you think would be better (more realistic), which one would be more scalable, and why?

For further details on the implementation of water simulations and rendering through grid based approaches, refer to the related papers in the lab 3 folder.

PART 2 - Rope Simulation

2 Introduction

2.1 Springs

Springs can be simulated by using Hooke's law[9]:

$$F_{spring} = kx \tag{1}$$

Where F is the force applied to the object attached to the spring. k is the spring stiffness and x is the distance from the springs resting place.

To make the object not bounce forever dampening can be added:

$$F_{dampening} = -bv_{rel} \tag{2}$$

Where b is the dampening factor and v_{rel} is the relative velocity between the two objects connected by the spring.

Air resitance can be added with:

$$F_{airresistance} = -cv_{abs} \tag{3}$$

v_{abs} is the absolute velocity and c is the air resistance factor.

2.2 The Rope

A rope can be simulated using multiple particles tied together with springs. Also a special point can be used which the rope is tied to. This point's position is static and no forces affects it. Each other particle (except the last) is affected by two springs. Then there are some additional forces from the environment such as gravity, air resistance and collision. If two points $p1$ and $p2$ are bound together with a spring, r is the vector between $p1$ and $p2$, and d is the resting distance, the spring forces are:

$$\begin{aligned} F_{p1} &= k(|r| - d)(r/|r|) - b(v_{p1} - v_{p2}) \\ F_{p2} &= -F_{p1} \end{aligned} \tag{4}$$

2.3 Integration methods

There are many different integration methods, all with different properties that can affect the stability of the simulation. Stability in the sense that the simulation works as intended when it fails it could be an object's orbit around another

[8] http://en.wikipedia.org/wiki/Newton's_law_of_universal_gravitation
[9] http://en.wikipedia.org/wiki/Hooke's_law

object changes in each iteration until it flies away into space. In a physics system it could be that the energy is not preserved.

x is the time y the position, we then have a function which gives us the derivative $f(x, y) = y'(x)$ at the current time. h is the time step taken in each iteration.

2.3.1 Euler

Euler[10] is one of the simplest integration methods and can in many scenarios give good enough results while easy to implement. The algorithm calculates the next position by taking the current and moves at the current speed for one time step.

$$y_{n+1} = y_n + hf(x_i, y_i) \tag{5}$$

2.3.2 Runge-Kutta 4

Runge-Kutta(RK4)[11] is a more advanced method which takes four weighted samples during the time step to give a better approximation.

$$y_{i+1} = u_i + \frac{h}{6}(f_1 + 2f_2 + 2f_3 + f_4) \tag{6}$$

$$\begin{aligned} f_1 &= f(x_i,\, y_i) \\ f_2 &= f(x_i + \frac{h}{2},\, y_i + \frac{h}{2} f_1) \\ f_3 &= f(x_i + \frac{h}{2},\, y_i + \frac{h}{2} f_2) \\ f_4 &= f(x_i + h,\, y_i + h f_3) \end{aligned} \tag{7}$$

2.3.3 Leapfrog

The Leapfrog[12] integration method updates the velocity every odd time step and the position at every even time step. So the position and the velocity is only updated every second time step. The procedure is quite similar to Euler's method but gives better results. It has two other useful properties, it conserves the energy of the system and can give stable orbits.

$$\begin{aligned} x_i &= x_{i-1} + hv_{i-1/2} \\ v_{i+1/2} &= v_{i-1/2} + a_i h \end{aligned} \tag{8}$$

3 The Lab

3.1 Download

The project files for this section is available at DD1354 lab materials link *Lab 2 download (part 2)*: `https://www.kth.se/social/course/DD1354/page/lab-materials/`.

[10] `http://en.wikipedia.org/wiki/Euler_method`
[11] `http://en.wikipedia.org/wiki/Runge-Kutta_methods`
[12] `http://en.wikipedia.org/wiki/Leapfrog_integration`

3.2 Tasks

The first thing you will want to do is to setup the lab environment. This is explained in the Setup section (4)

These are the main tasks:

1. Finish the rope implementation by adding spring forces and dampening, see section 2.1, in the class `Rope` in the method `ApplySpringForces()`. You can access the position of a point with `p.State.Position`, the velocity with: `p.State.Velocity`, the dampening factor with `m_ropeDampening` and the stiffness with `m_ropeStiffness`.
2. When using Euler as integration method, try changing the stiffness of the rope. What happens at higher values?
3. Now switch to RK4 as the integration method and try the same values as when Euler method was used, also test some higher values. What happens?
4. What's the difference when running Euler and RK4 as a integration method?

3.3 Optional extra challenges

1. Look at the code which handles the floor collision and implement collision with the sphere located in the scene (initially the sphere is hidden).
2. Choosing the LeapFrog integrator will not do anything since it is not implemented. Implement the LeapFrog integrator! How does it compare to the other integration methods with regard to stability and features?
3. The lab gives you a really good foundation to explore cloth simulation. Try to extend the lab to simulate cloth!

3.4 The Unity setup

You can find the main scene "RopeScene.unity" in `Assets/Scenes/`. The main object in the scene is the `Rope` which has all the parameters for the rope. The `RopePointRoot` is a fixed point which the rope is attached to in one end.

3.5 The Rope Class

This class is the container for the rope simulation. It has a list of all the points and some settings. It also methods which calculates the spring force, gravity, dampening and other forces. These methods are used in the simulation to update the acceleration, velocity and position of the rope points.

3.6 The Rope Point Class

The rope point class represents one joint of the rope and could be seen as a particle with some special properties. It has a state with velocity and position which are used in the simulation. Use the method `ApplyForce(Vector3 force)` to use a force on a rope point.

3.7 The InputControl Class

This class handles basic drag interaction, which lets you drag objects around the scene in game view. The drag mouse button is the left button by default.

3.8 The MaxCamera Class

A simple camera class attached to the main camera enables **rotation** with the right mouse button, **panning** with the middle mouse button and **scrolling** with the mouse-wheel.

3.9 The Integrators

The integrators are classes implements different integration strategies. Each strategy has different properties that may or may not be suitable for this kind of simulation. You have tested the simplest, Euler integration, before but there other more advanced like Runge-Kutta and LeapFrog which have a lower truncation error and/or stability properties.

```
public interface Integrator
{
    void Advance(List<RopePoint> points, Action<float>
        updateForcesFunc, float timeStep);
}
```

Listing 1: The Integrator interface

All the integrator classes implements the interface `Integrator` which comes with a single method as can be seen in listing 1. The `Action<float>` type is a function reference, also called a delegate in C#, which is a reference to a function that takes a single argument of the type float. This function, `updateForcesFunc`, is used in the integrator to update all the forces.

4 Setup

The Unity projects are available on bitbucket via git, and we ask you to download the code this way. Each lab assignment has a link to a git repository. Once browsing a repository follow one of the instructions below:

- Download via clicking *Downloads* and *Download Repository*
- *clone* the repo with a git client (examples are TortoiseGit or Cygwin. This option is preferable but needs a bit setting up. We urge you to learn this eventually, but not necessarily for this lab.

5 Resources

In the following section you will find links to useful resources that will help you to get through this lab successfully.

`http://docs.unity3d.com/Manual/index.html`
`http://gafferongames.com/game-physics/integration-basics/`
`http://gafferongames.com/game-physics/spring-physics/`

6 Troubleshooting

This lab was tested on a Windows installation of Unity 3D. The free version of the software is enough to complete the tutorial and requires only that you register the software with your email prior to running it. Unity is also available for Mac, but not for Linux.