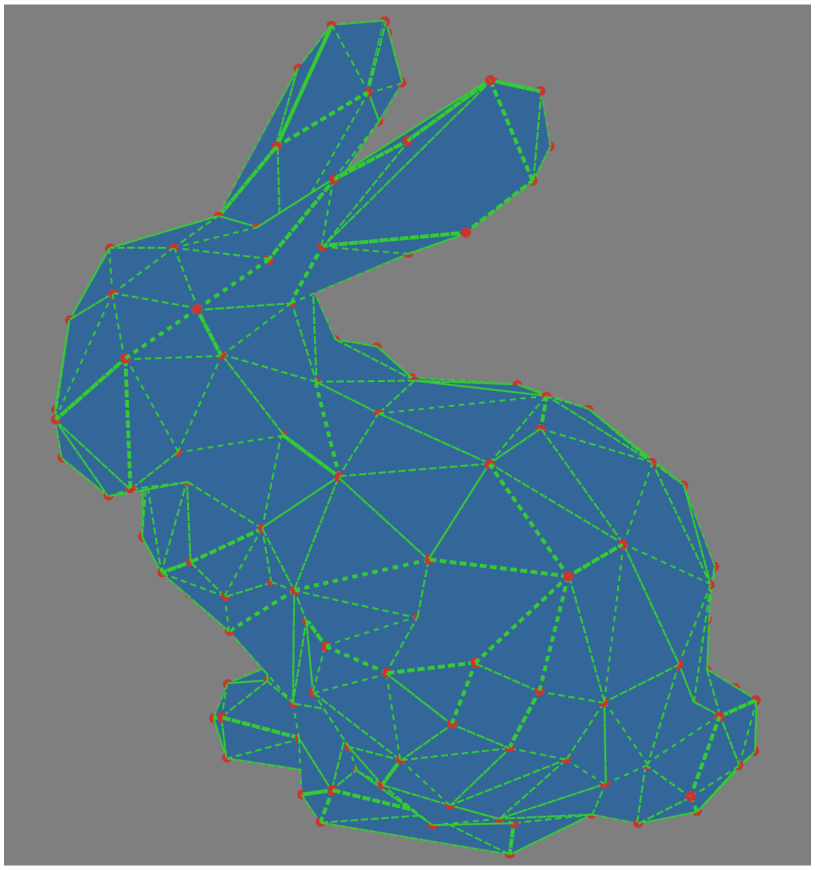
Assignment 1: Mass-Spring System

Due February 15 at 11:59pm



Introduction

Mass-spring systems are a fundamental tool in physical simulations, and a wide range of natural phenomena can be described or approximated by these systems. In this assignment, you will implement a simple mass-spring system operating according to the equations described in class. You will also implement a few numerical integrators in order to simulate the system. Lastly, you will extend the base mass-spring system in order to model some physical phenomenon of your choosing. The best simulations (as determined by the instructors) will receive prizes!

Getting Started

The codebase has been successfully tested on Windows and Linux. We have not tested the code on Mac, but if you are able to get it running, please feel free to e-mail the TA about how you were able to do it.

On Windows

Some version of Visual Studio is required; the assignment was developed with Visual Studio 2015 Community, which is available for free. Download the code from the site and unzip it. In the `code` directory, you will find a Visual Studio solution file, `SCP.sln`. The code will not compile out of the box – you will need to add the `libs` directory to your system path environment variable. Instructions on how to do this are in the included `README.txt`.

On Linux

First, obtain the dependencies listed in `README.txt`. You will also need `make` and `g++`. After this, run `sh make_makefile.sh` in order to generate the Makefile, and then run `make` to compile the project. The executable will be in `out/cs467`. Note that you will have to `cd out` before running `./cs467`, as the relative paths in the project require the working directory to be `out`.

Code overview

Within the project, you will see several subdirectories. **You should not edit any files outside of the `Playground` directory.** However, it may be helpful to familiarize yourself with some of the classes in these libraries. In particular, you will be using the classes in `MathLib` to perform vector calculations. Here are the classes that correspond to the dimensions of vectors and matrices you will need:

- 3×1 point: `P3D`
- 3×1 vector: `V3D`
- $d \times 1$ vector: `dVector`
- 3×3 matrix: `Matrix3x3`
- $d \times d$ matrix: `MS_SparseSquare`

The `ParticleSystem` class is where time integration takes place. You should familiarize yourself with the way the system state is stored. In a system with n particles, the positions and velocities are stored in `dVector`s of length $3n$, where the values of the vector for particle i are stored in indices $3i$, $3i+1$, and $3i+2$. (Particles are zero-indexed.) The masses are stored in a `dVector` of length n, where the mass of particle i is in index i. The force and acceleration vectors you will compute should have identical structure to the position and velocity vectors. Places where you will need to fill in code are marked with a `TODO`.

When you first run the project, you will see a collection of particles and springs on screen. The camera is controllable by clicking and dragging the mouse buttons. You can also use the left mouse button to pick and drag particles around, and you can use the right mouse button to create a “pin,” which is a zero-length spring binding the clicked particle to its current location. Along the left sidebar, you will see several options, some of which affect the display, and some of which affect the simulation. The bottom area is the console. In your code, if you include `Utils/Logger.h`, you can write to this console by calling `Logger::consolePrint`, which may be useful for debugging.

You can press G to turn gravity on or off. You can use F, S, and B to switch between forward, symplectic, and backward Euler integrators, respectively. You can press the spacebar to begin the simulation. However, because none of these things have been implemented yet, none of the particles will move. This makes for a very dull system, so we will begin by implementing some basic forces and integrators.

Part 1: Gravity and Forward Euler (1 point)

In `ParticleSystem.cpp`, you will find most of the functions for time integration. We will begin by implementing a simple gravity force here, and by writing forward Euler to integrate this force over time. Your tasks in this section are:

1. In `ParticleSystem::computeForceVector`, add gravity to the force experienced by each particle. Note that the positive y direction points upward in the project's coordinate frame, so gravity should act in the negative y direction.
2. Complete the function `ParticleSystem::computeAccelerations`, which computes the acceleration vector from the force vector.
3. Implement forward Euler in `ParticleSystem::integrate_FE`. The acceleration to be integrated is the acceleration returned by `computeAccelerations`.

Once you have completed these tasks, you should be able to start the simulation, check "Enable Gravity" on the left sidebar (or alternatively press G), and see the particles fall down.

Part 2: Spring Forces and Symplectic Euler (3 points)

While the gravity force we have implemented is physically accurate, it is not particularly compelling. Recall also that forward Euler is unconditionally unstable on oscillatory systems. While this does not pose a problem for the constant gravitational force, mass-spring systems are inherently oscillatory, so forward Euler will not be able to simulate these systems.

In this section, we will add spring forces to our system, and implement symplectic Euler to integrate these forces.

In our system, we have two types of springs: zero-length springs, and non-zero length springs. Zero-length springs are used as "pins", attaching particles to a constant point in world space, while non-zero length springs are used to attach pairs of particles together.

Your tasks in this section are:

1. Complete the function `ZeroLengthSpring::computeForces`, which computes the force experienced by an attached particle to a zero-length spring.
2. Complete the function `Spring::computeForces`, which computes the forces experienced by both attached particles to the spring.
3. In `ParticleSystem::computeForceVector`, accumulate all forces from all springs into the system's force vector. Both `Spring` and `ZeroLengthSpring` have a function called `addForcesToVector` which you can use for this purpose, provided that you have implemented `computeForces`.
4. Implement symplectic Euler in `ParticleSystem::integrate_SE`.

After you have completed these tasks, you should be able to start the simulation and see the motions induced by springs pulling on the particles. You can manually move particles by dragging with the left mouse button, and you can pin particles to their current positions by right-clicking them.

You should also be able to see that forward Euler is unstable on typical mass-spring systems. If you switch to using symplectic Euler, you should see a more accurate and stable simulation. (The two `.mss` examples are particularly illustrative; note that `triangle.mss` has some initial velocity on the particles.)

Part 3: Backward Euler (3 points)

Symplectic Euler is a fairly good integrator for many systems, but backward Euler has the advantage of unconditional stability (for linear ODEs). In this section, we will implement the spring force Jacobians required for implicit integration. We will use these to implement backward Euler.

Your tasks for this section are:

1. Complete the functions `ZeroLengthSpring::compute_dFdx` and `Spring::compute_dFdx`, which compute the force Jacobians respectively for zero-length springs and non-zero-length springs.
2. Complete the function `ParticleSystem::computeForceJacobian`, which accumulates the Jacobian matrix by adding all the blocks contributed by all springs. Both spring classes have functions `addBlocksToMatrix` which will add the blocks to their proper locations in a `MS_SparseSquare` matrix.
3. Complete the function `ParticleSystem::newtonStep`, which takes one step of Newton's method. The argument `v_guess` is the current estimate of v_{n+1}, the future velocity; the function should return the refined estimate of v_{n+1}. To solve the linear system, use the function `MS_SparseSquare::solve`, where `A.solve(b)` solves the system $Ax = b$ for x.
4. Implement backward Euler in `ParticleSystem::integrate_BE`.

If you switch to integrating using backward Euler, you should now be able to see the effects we discussed in class. Backward Euler exhibits numerical damping, meaning the system loses energy over time; this is reflected by the particles gradually coming to a halt. On larger systems, you should also notice that the implicit backward Euler method runs slower than the explicit forward and symplectic Euler methods, due to the need to solve a linear system for each step of Newton's method. This can be improved by limiting the number of Newton steps or by implementing a line search for Newton's method, but these are not required for this assignment.

Part 4: Extension (3 points)

Now that you have created a basic mass-spring system, you will use it to simulate some physical phenomenon of your choosing. We discussed several possibilities in class (cloth, shells, hair, contact and friction, rigid and elastic objects, plasticity, viscosity and fracture, tensegrity structures, etc) which you are free to build upon, or you may pursue an idea of your own. Be creative! Feel free to talk to the instructors about your ideas for this part.

Creating examples

In order to show off your extension, you may want to create more examples beyond the ones included with the release. There are two ways to import particle systems from external files. One is to create an `.obj` file using some external tool and read it in. The vertices of the `.obj` model will become particles, and the edges will become springs joining the two endpoints. The masses of the particles and stiffnesses of the springs will be the default values defined in `Constants.h`. Additionally, the faces of the `.obj` model will continue to be rendered even as the vertices deform.

The other way to import particle systems is to specify them in an `.mss` file, which is just a listing of particles, springs, and zero-length springs. You can generate this file manually or procedurally through code. The structure of the `.mss` file is as follows. First, the file should contain a list of particles, one on each line, where the particle is formatted as:

```
v x1 x2 x3 v1 v2 v3 m
```

Here, `v` is just the character `v`, while `x1`, `x2`, and `x3` are the coordinates of the particle's initial position, `v1`, `v2`, and `v3` are the coordinates of the particle's initial velocity, and `m` is the particle's mass.

Next, the file contains a list of springs, formatted as:

```
s p1 p2 k
```

`s` is the character `s`, while `p1` and `p2` are the indices of the endpoints of this spring, and `k` is the stiffness. The two indices are zero-indexed with respect to the order in which the particles appear in the file.

Finally, the file contains a list of zero-length springs, formatted as:

```
z p x1 x2 x3 k
```

`z` is the character `z`, `p` is the index of the particle attached to the spring, `x1`, `x2`, and `x3` are the world coordinates of the fixed endpoint of the spring, and `k` is the stiffness.

To load a new particle system without recompiling the project, click in the console window at the bottom of the project interface and type `load <file>`, where `<file>` is the path to the `.obj` or `.mss` file (e.g. load `../meshes/bunny200.obj`).

Also feel free to tinker with the values in `Constants.h` in order to achieve any desired effects.

Submission instructions

You will need to submit three parts: a write up, your code, and a submission video. When submitting your code, please delete the `Playground/Release` and `Playground/Debug` directories if they are present. Then, zip **only** the `Playground` directory.

If you have created additional examples for your extension, you may also zip those and send them to us.

To export a submission video from your project, you can use the "Record Screenshots" checkbox at the top-left of the project interface. Once checked, the program will write each frame of the simulation to an image in the `screenShots` folder in the project directory. Note that it will be writing uncompressed `.bmp` images for each frame, so the size of this directory can grow quite large – make sure you have sufficient disk space before doing this. Additionally, this will cause the simulation to become quite slow as it writes the images, though this will not affect the speed of the video.

Once you have exported the series of frames, you should be able to combine them into a video using most video editing software. VirtualDub is a free tool that also has this functionality. Alternatively, if it is more convenient, you can use a screen recording tool to capture the window directly.

In order to submit, please e-mail the instructor (scoros@cmu.edu) and TA the (christoy@cs.cmu.edu) following items:

- A brief writeup (2 pages max) that describes your effort in implementing tasks 1 through 4, including the extension you have implemented, how you modeled the system, what worked and what did not work, and any interesting insights you may have gained.
- A `.zip` of the contents of your `Playground` directory, after having deleted the `Debug` and `Release` directories inside (please be careful not to delete your code!).
- A `.zip` of any additional data (e.g. .mss) file you have created for your extension, if applicable.

- Either your submission video, or if it is too large, a link to somewhere we can download your submission video (e.g. Youtube, Dropbox, etc.).

Please also include the string "CS15467" in your subject line so we know to look for it.

Notes on Academic Integrity

You are allowed to collaborate on the assignment in terms of formulating ideas, developing physical models and mathematical equations. However, you must implement the code and do the write up completely on your own, and understand what you are writing. Please also list the names of everyone that you discussed the assignment with.