README.md

Introduction

The third assignment will introduce you to the most common numerical method for simulating, well, almost anything in Computer Graphics, the mighty Finite Element Method.

Prerequisite installation

On all the platforms, we will assume you have installed cmake and a modern c++ compiler on Mac OS X[1], Linux[2], or Windows[3].

We also assume that you have cloned this repository using the `--recursive` flag (if not then issue `git submodule update --init --recursive`).

Note: We only officially support these assignments on Ubuntu Linux 18.04 (the OS the teaching labs are running) and OSX 10.13 (the OS I use on my personal laptop). While they *should* work on other operating systems, we make no guarantees.

All grading of assignments is done on Linux 18.04

Layout

All assignments will have a similar directory and file layout:

```
README.md
CMakeLists.txt
main.cpp
assignment_setup.h
include/
  function1.h
  function2.h
  ...
src/
  function1.cpp
  function2.cpp
  ...
data/
  ...
...
```

The `README.md` file will describe the background, contents and tasks of the assignment.

The `CMakeLists.txt` file setups up the cmake build routine for this assignment.

The `main.cpp` file will include the headers in the `include/` directory and link to the functions compiled in the `src/` directory. This file contains the `main` function that is executed when the program is run from the command line.

The `include/` directory contains one file for each function that you will implement as part of the assignment.

The `src/` directory contains *empty implementations* of the functions specified in the `include/` directory. This is where you will implement the parts of the assignment.

The `data/` directory contains *sample* input data for your program. Keep in mind you should create your own test data to verify your program as you write it. It is not necessarily sufficient that your program *only* works on the given sample data.

Compilation for Debugging

This and all following assignments will follow a typical cmake/make build routine. Starting in this directory, issue:

```
mkdir build
cd build
cmake ..
```

If you are using Mac or Linux, then issue:

```
make
```

Compilation for Testing

Compiling the code in the above manner will yield working, but very slow executables. To run the code at full speed, you should compile it in release mode. Starting in the **build directory**, do the following:

```
cmake .. -DCMAKE_BUILD_TYPE=Release
```

Followed by:

```
make
```

Your code should now run significantly (sometimes as much as ten times) faster.

If you are using Windows, then running `cmake ..` should have created a Visual Studio solution file called `a3-finite-elements-3d.sln` that you can open and build from there. Building the project will generate an .exe file.

Why don't you try this right now?

Execution

Once built, you can execute the assignment from inside the `build/` using

```
./a3-finite-elements-3d
```

Background

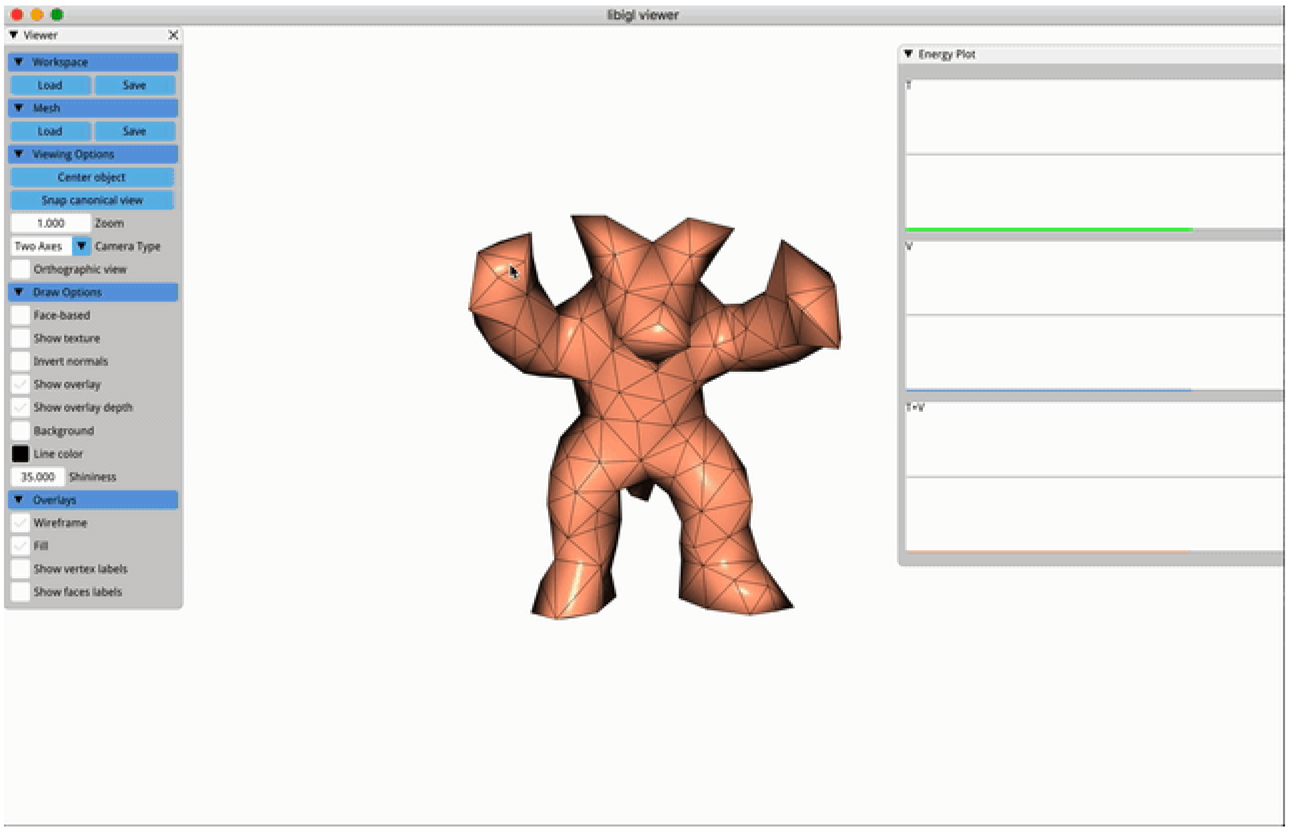
In this assignment you will get a chance to implement one of the gold-standard methods for simulating elastic objects -- the finite element method (FEM). Unlike the particles in the previous assignment, the finite-element method allows us compute the motion of continuous volumes of material. This is enabled by assuming that the motion of a small region can be well approximated by a simple function space. Using this assumption we will see how to generate and solve the equations of motion.

FEM has wonderfully practical origins, it was created by engineers to study complex aerodynamical and elastic problems in the 1940s. My MSc supervisor used to regale me with stories of solving finite element equations by hand on a chalkboard. With the advent of modern computers, its use as skyrocketed.

FEM has two main advantages over mass-spring systems. First, the behaviour of the simulated object is less dependent on the topology of the simulation mesh. Second, unlike the single stiffness parameter afforded by mass spring systems, FEM allows us to use a richer class of material models that better represent real-world materials.

Resources

Part I of this SIGGRAPH Course, by Eftychios Sifakis and Jernej Barbic, is an excellent source of additional information and insight, beyond what you will find below.



The Finite Element method

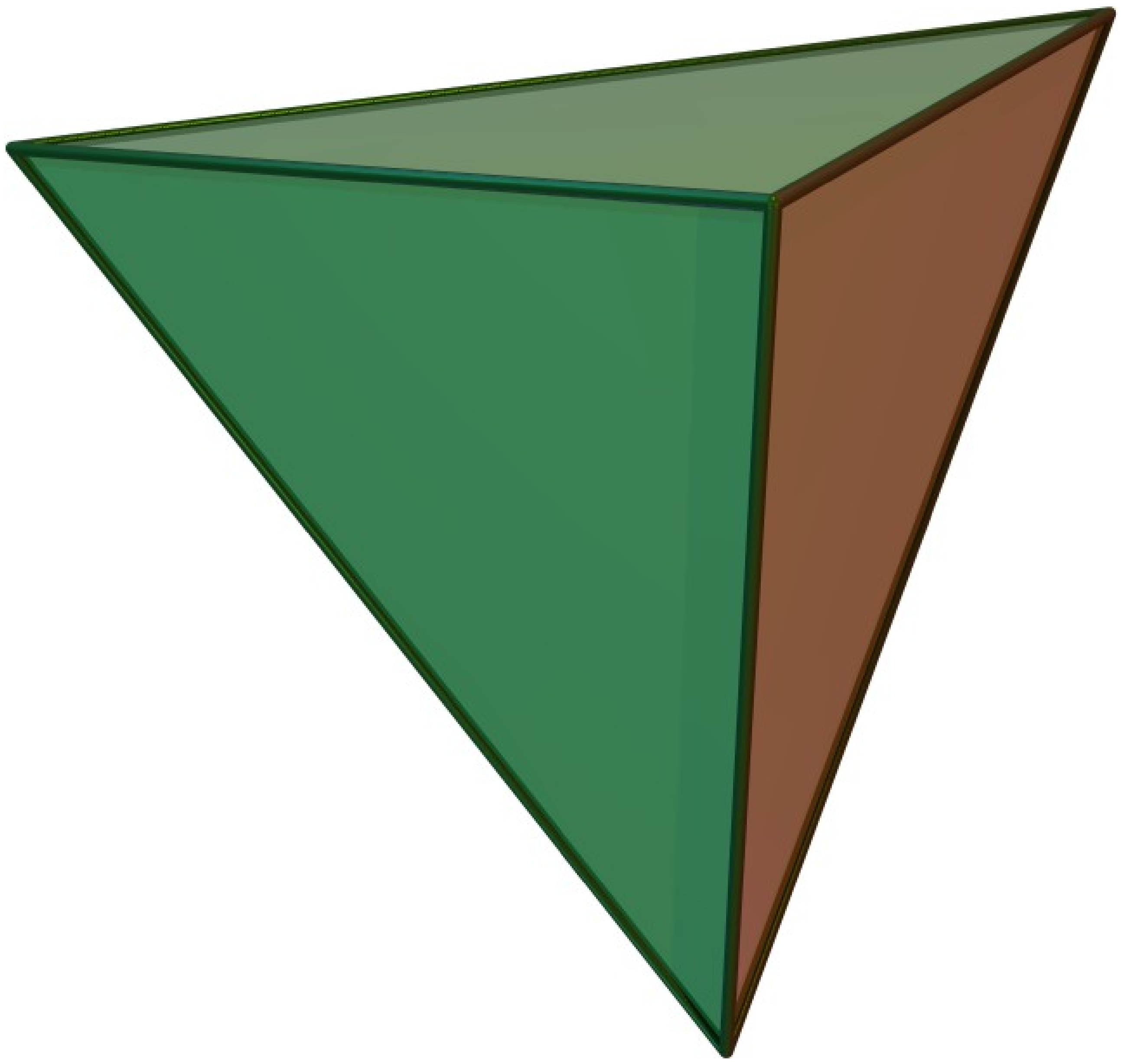
The idea of the finite element method is to represent quantities inside a volume of space using a set of scalar *basis* or *shape* functions $\phi_i(\mathbf{x})$ where $\mathbf{x} \in \mathcal{R}^3$ is a point inside the space volume. We then represent any quantity inside the volume as a linear combination of these basis functions:

$$f(\mathbf{x}) = \sum_{i=0}^{b-1} w_i \phi_i(\mathbf{x})$$

where w_i are weighting coefficients. Designing a finite element method involves making a judicious choice of basis functions such that we can compute the w_i's efficiently. Spoiler Alert: in the case of elastodynamics, these w_i's will become our generalized coordinates and will be computed via time integration.

Our Geometric Primitive: The Tetrahedron

For this assignment we will use a tetrahedron as the basic space volume. The reason we work with tetrahedra is two-fold. First, as you will see very soon, they allow us to easily define a simple function space over the volume. Second, there is available software to convert arbitrary triangle meshes into tetrahedral meshes.



A Piecewise-Linear Function Space

Now we are getting down to the nitty-gritty -- we are going to define our basis functions. The simplest, useful basis functions we can choose are linear basis functions, so our goal is to define linear functions inside of a tetrahedron. Fortunately such nice basis functions already exist! They are the barycentric coordinates. For a tetrahedron there are four (4) barycentric coordinates, one associated with each vertex. We will choose ϕ_i to be the i^{th} barycentric coordinate.

Aside from being linear, barycentric coordinates have another desireable property, called the *Kronecker delta property* (or fancy Identity matrix as I like to think of it). This is a fancy-pants way of saying that the i^th barycentric coordinate is zero (0) when evaluated at any vertex, j, of the tetrahedron, $j \neq i$, and one (1) when evaluated at $j = i$. What's the practical implication of this? Well it means that if I knew my function $f\left(x\right)$, then the best values for my w_i's would be $f\left(\mathbf{x}_i\right)$, or the value of f evaluated at each vertex of my tetrahedron.

All of this means that a reasonable way to approximate any function in our tetrahedron is to use

$$f\left(\mathbf{x}\right) = \sum_{i=0}^{3} f_i \phi_i\left(\mathbf{x}\right)$$

where $\phi\left(\mathbf{x}\right)$ are now the tetrahedron barycentric coordinates and f_i are the values of f at the nodes of the tetrahedron. Because our basis functions are linear, and the weighted sum of linear functions is still linear, this means that we are representing our function using a linear function space.

The Extension to 3D Movement

To apply this idea to physics-based animation of wiggly bunnies we need to more clearly define some of the terms above. First, we need to be specific about what our function f will be. As with the particles in the previous assignments, what we care about tracking is the position of each mesh vertex, in the world, over time. For the i^th vertex we can denote this as $\mathbf{x}^t_i \in \mathcal{R}^3$. We are going to think of this value as a mapping from some undeformed space $\mathbf{X} \in \mathcal{R}^3$ into the real-world. So the function we want to approximate is $\mathbf{x}^t\left(\mathbf{X}\right)$ which, using the above, is given by

$$\mathbf{x}^t\left(\mathbf{X}\right) = \sum_{i=0}^{3}\mathbf{x}^t_i\phi_i\left(\mathbf{X}\right)$$

The take home message is that, because we evaluate ϕ_i's in the undeformed space, we need our tetrahedron to be embedded in this space.

The Generalized Coordinates

Now that we have our discrete structure setup, we can start "turning the crank" to produce our physics simulator. A single tetrahedron has four (4) vertices. Each vertex has a single $\mathbf{x}^t_i$ associated with it. As was done in assignment 2, we can store these *nodal positions* as a stacked vector and use them as generalized coordinates, so we have

$$\mathbf{q}^t = \begin{bmatrix}\mathbf{x^t_0}\\ \mathbf{x^t_1}\\ \mathbf{x^t_2}\\ \mathbf{x^t_3}\end{bmatrix}$$

Now let's consider the velocity of a point in our tetrahedron. Given some specific $\mathbf{X}$, the velocity at that point is

$$\frac{d\mathbf{x}^t\left(\mathbf{X}\right)}{dt} = \frac{d}{dt}\sum_{i=0}^{3}\mathbf{x}^t_i\phi_i\left(\mathbf{X}\right)$$

However, only the nodal variables actually move in time so we end up with

$$\frac{d\mathbf{x}^t\left(\mathbf{X}\right)}{dt} = \sum_{i=0}^{3}\frac{d\mathbf{x}^t_i}{dt}\phi_i\left(\mathbf{X}\right)$$

Now we can rewrite this whole thing as a matrix vector product

$$\frac{d\mathbf{x}^t\left(\mathbf{X}\right)}{dt} = \underbrace{\begin{bmatrix}\phi_0\left(\mathbf{X}\right)I & \phi_1\left(\mathbf{X}\right)I & \phi_2\left(\mathbf{X}\right)I & \phi_3\left(\mathbf{X}\right)I\end{bmatrix}}_{N}\underbrace{\begin{bmatrix}\dot{\mathbf{x}}^t_0\\ \dot{\mathbf{x}}^t_1\\ \dot{\mathbf{x}}^t_2\\ \dot{\mathbf{x}}^t_3\end{bmatrix}}_{\dot{\mathbf{q}}}$$

where I is the 3×3 Identity matrix.

The Kinetic Energy of a Single Tetrahedron

Now that we have generalized coordinates and velocities we can start evaluating the energies required to perform physics simulation. The first and, and simplest energy to compute is the kinetic energy. The main difference between the kinetic energy of a mass-spring system and the kinetic energy of an FEM system, is that the FEM system must consider the kinetic energy of every infinitesimal piece of mass inside the tetrahedron.

Let's call an infinitesimal chunk of volume dV. If we know the density ρ of whatever our object is made out of, then the mass of that chunk is ρdV and the kinetic energy, T is $\frac{1}{2}\rho dV \dot{\mathbf{x}^t}\left(\mathbf{X}\right)^T \dot{\mathbf{x}^t}\left(\mathbf{X}\right)$. To compute the kinetic energy for the entire tetrahedron, we need to integrate over it's volume so we have

$$T = \frac{1}{2}\int_{\text{tetrahedron}} \rho \dot{\mathbf{q}}^T N\left(\mathbf{X}\right)^T N\left(\mathbf{X}\right)\dot{\mathbf{q}} dV$$

BUT~ $\dot{\mathbf{q}}$ is constant over the tetrahedron so we can pull that outside the integration leaving

$$T = \frac{1}{2}\dot{\mathbf{q}}^T \underbrace{\left(\int_{\text{tetrahedron}} \rho N\left(\mathbf{X}\right)^T N\left(\mathbf{X}\right) dV\right)}_{M_e}\dot{\mathbf{q}}$$

in which the *per-element* mass matrix, M_e, makes an appearance.. In the olden days, people did this integral by hand but now you can use symbolic math packages like *Mathematica*, *Maple* or even *Matlab* to compute its exact value.

The Deformation of a Single Tetrahedron

Now we need to define the potential energy of our tetrahedron. Like with the spring, we will need a way to measure the deformation of our tetrahedron. Since the definition of length isn't easy to apply for a volumetric object, we will try something else -- we will define a way to characterize the deformation of a small volume of space. Remember that all this work is done to approximate the function $\mathbf{x}^t\left(\mathbf{X}\right)$ which maps a point in the undeformed object space, $\mathbf{X}$, to the world, or deformed space. Rather than consider what happens to a point under this mapping, let's consider what happens to a vector.

To do that we pick two arbitary points in the undeformed that are infinitesimally close. We can call them $\mathbf{X}_1$ and $\mathbf{X}_2$ (boring names I know). The vector between them is $dX = \mathbf{X}_2 - \mathbf{X}_1$. Similarly the vector between their deformed counterparts is $dx = \mathbf{x}\left(\mathbf{X}_2\right) - \mathbf{x}\left(\mathbf{X}_1\right)$. Because we chose the undeformed points to be infinitesimally close and $\mathbf{x}\left(\mathbf{X}_2\right) = \mathbf{x}\left(\mathbf{X}_1 + dX\right)$, we can use Taylor expansion to arrive at

$$dx = \underbrace{\frac{\partial \mathbf{x}\left(\mathbf{X}\right)}{\partial \mathbf{X}}}_{F} dX$$

where F is called the deformation gradient. Remember, F results from differentiating a 3-vector by another 3-vector so it is a 3×3 matrix.

Because dX is pointing in an arbitrary direction, F, captures information about how any dX changes locally, it encodes volumetric deformation.

The FEM discretization provides us with a concrete formula for $\mathbf{x}\left(\mathbf{X}\right)$ which can be differentiated to compute F. *An important thing to keep in mind* -- because our particular FEM uses linear basis functions inside of a tetrahedron, the deformation gradient is a constant. Physically this means that all dX's are deformed in exactly the same way inside a tetrahedron.

Given F we can consider the squared length of any dx

$$l^2 = dx^T dx = dX^T \underbrace{\left(F^T F\right)}_{\text{right Cauchy-Green Strain tensor}} dX$$

Like the spring strain, $F^T F$ is invariant to rigid motion so it's a pretty good strain measure.

The Potential Energy of a Single Tetrahedron

The potential energy function of a tetrahedron is a function that associates a single number to each value of the deformation gradient. Sadly, for the FEM case, things are a little more complicated than just squaring F (but thankfully not much).

The Strain Energy density

Like the kinetic energy, we will begin by defining the potential energy on an infinitesimal chunk of the simulated object as $\psi\left(F\left(\mathbf{X}\right)\right) dV$ where ψ is called the *strain energy density function. Mostly, we look up strain energy density functions in a book. Material scientists have been developing them for many years so that they mimic the behaviour of realistic materials. For this assignment you will use the well established, Neo-Hookean (its better than Hooke's Law because its new) strain energy density for compressible materials. This model approximates the behaviour of rubber-like materials. There are many formulations on that page, and we'll use the following:

$$\psi(F) = C(J^{-2/3}\text{tr}(F^T F) - 3) + D(J-1)^2$$

where $J = \det(F)$.

The total potential of the tetrahedron can be defined via integration as

$$V = \int_{\text{tetrahedron}} \psi\left(F\left(\mathbf{X}\right)\right) dV$$

Numerical quadrature

Typically we don't evaluate potential energy integrals by hand. They get quite impossible, especially as the FEM basis becomes more complex. To avoid this we typically rely on numerical quadrature. In numerical quadrature we replace an integral with a weighted sum over the domain. We pick some quadrature points $\mathbf{X}_i$ (specified in barycentric coordinates for tetrahedral meshes) and weights w_i and evaluate

$$V \approx \sum_{i=0}^{p-1} \psi\left(F\left(\mathbf{X}_i\right)\right) \cdot w_i$$

However, for linear FEM, the quadrature rule is exceedingly simple. Recall that linear basis functions imply constant deformation per tetrahedron. That means the strain energy density function is constant over the tetrahedron. Thus the perfect quadrature rule is to choose $\mathbf{X}_i$ as any point inside the tetrahedron (I typically use the centroid) and w_i as the volume of the tetrahedron. This is called *single point* quadrature because it estimates the value of an integral by evaluating the integrated function at a single point.

Forces and stiffness

The per-element generalized forces acting on a single tetrahedron are given by

$$\mathbf{f}_e = -\frac{\partial V}{\partial \mathbf{q}}$$

and the stiffness is given by

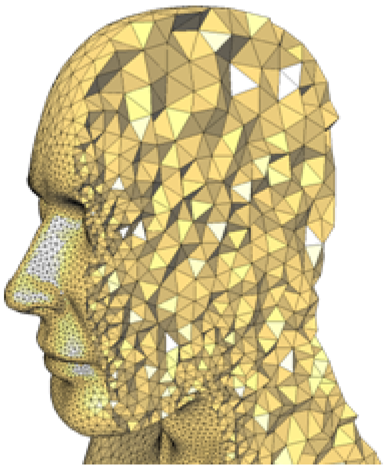
$$K_e = -\frac{\partial^2 V}{\partial \mathbf{q}^2}$$

These can be directly computed from the quadrature formula above. Again, typically one uses symbolic computer packages to take these derivatives and you are allows (and encouraged) to do that for this assignment.

For a tetrahedron the per-element forces are a 12×1 vector while the per-element stiffness matrix is a dense, 12×12 matrix.

From a Single Tetrahedron to a Mesh

Extending all of the above to objects more complicated than a single tetrahedron is analogous to our previous jump from a single spring to a mass-spring system.



The initial step is to divide the object to be simulated into a collection of tetrahedra. Neighboring tetrahedra share vertices. We now specify the generalized coordinates of this entire mesh as

$$\mathbf{q} = \begin{bmatrix} \mathbf{x^t}_0 \\ \mathbf{x^t}_1 \\ \mathbf{x^t}_2 \\ \vdots \\ \mathbf{x^t}_n \end{bmatrix}$$

where n is the number of vertices in the mesh. We use selection matrices (as we did in assignment 2) which yield identical assembly operations for the global forces, stiffness and mass matrix.

Time integration

Because you can compute all the necessasry algebraic operators (M, K, $\mathbf{f}$) you can use your linearly-implict Euler code from assignment 2 to integrate your FEM system. To see the limitations of this approach, run `a3-finite-elements-3d arma` and press `N` . Interacting with the armadillo will almost immediately cause your simulation to explode. This is because our bunny was secretly gigantic! So its effective stiffness was very low. This armadillo is much smaller (less than 1 meter tall). Even though it uses the same material parameters, its effective stiffness is much higher. Linearly-implicit Euler just cannot handle it, and so ... Kaboom!

Backward (Implicit) Euler

To fix this you will implement the full backward Euler integration scheme which will solve the discrete time stepping equations

$$M\dot{\mathbf{q}}^{t+1} = M\dot{\mathbf{q}}^{t} + \Delta t \mathbf{f}\left(\mathbf{q}^{t} + \Delta t \dot{\mathbf{q}}^{t+1})\right)$$

The position of the mesh is updated using $\mathbf{q}^{t} + \Delta t \dot{\mathbf{q}}^{t+1}$.

To solve for $\dot{\mathbf{q}}^{t+1}$ it is useful to notice that solving the update equation above is equivalent to the optimization problem

$$\dot{\mathbf{q}}^{t+1} = \arg\min_{\dot{\mathbf{q}}} \frac{1}{2}\left(\dot{\mathbf{q}} - \dot{\mathbf{q}}^{t}\right)^{T} M \left(\dot{\mathbf{q}} - \dot{\mathbf{q}}^{t}\right) + V\left(\mathbf{q}^{t} + \Delta t \dot{\mathbf{q}}\right)$$

We are going to solve this minimization problem using Newton's method.

Newton's method

Newton's method computes the local minimum of an objective function by iteratively solving a sequence of quadratic minimizations. Let's look at the process for a single iteration (say the i^{th} iteration). At this iteration we already have the i^{th} guess for our new velocity, denoted $\dot{\mathbf{q}}^{t+1}_{(i)}$. The goal of the $i+1$ iteration is to compute a small change in the current velocity estimate,$\Delta \dot{\mathbf{q}}$, that reduces the objective function.

This is done by solving a local, quadratic optimization of the formula

$$\Delta \dot{\mathbf{q}}^{*} = \arg\min_{\Delta \dot{\mathbf{q}}} \frac{1}{2} \Delta \dot{\mathbf{q}}^{T} H\left(\dot{\mathbf{q}}^{t+1}_{(i)}\right) \Delta \dot{\mathbf{q}} + \mathbf{g}\left(\dot{\mathbf{q}}^{t+1}_{(i)}\right)^{T} \Delta \dot{\mathbf{q}}$$

where $\mathbf{g}$ is the Gradient of the backward Euler integration cost function, and H is the Hessian. In this case we have $\mathbf{g} = M\dot{\mathbf{q}}^{t+1}_{(i)} - M\dot{\mathbf{q}}^{t} + \Delta t \frac{\partial V\left(\mathbf{q}^{t} + \Delta t \dot{\mathbf{q}}^{t+1}_{(i)}\right)}{\partial \mathbf{q}}$ and

$H = M + \Delta t^{2} \frac{\partial^{2} V\left(\mathbf{q}^{t} + \Delta t \dot{\mathbf{q}}^{t+1}_{(i)}\right)}{\partial \mathbf{q}^{2}}$

The solution to this problem is given by $\Delta \dot{\mathbf{q}}^{*} = -H^{-1}\mathbf{g}$. You then compute $\dot{\mathbf{q}}^{t+1}_{(i+1)} = \dot{\mathbf{q}}^{t+1}_{(i)} + \Delta \dot{\mathbf{q}}^{*}$.

Repeating the process of constructing and solving this quadratic optimization is at the heart of Newton's method.

Line Search

Unfortunately the $\Delta\dot{\mathbf{q}}$ computed by Newton's method can be overly ambitious causing the algorithm to never find a local minimum. To avoid this problem, robust optimization schemes give themselves the option of taking a fractional newton's step. One simple way to find a good step size is to use backtracking line search. Line search is so named because it searches in the *direction* computed by Newton's method (along the line) but looks for a value of the full energy function that guarantees, for instance, sufficient decrease. Backtracking line search gets its name because it initially tries to take a full Newton Step, and if that step is flawed, divides the step by some ratio and then checks the result of this new step. This procedure is repeated until either a suitable step is found, or the step length being checked goes to zero.

High Resolution Display Mesh via Skinning

The final component of this assignment involves making our FEM simulation look a bit more appealing. FEM calculations can be slow, especially if we want real-time performance. This often limits us to the use of relatively low resolution simulation meshes. To compensate for this we can adopt the concept of skinning from computer animation.

Let us define two different meshes. The first is our simulation tetrahedral mesh with vertex positions given by $\mathbf{q}^t$. The second is going to be a higher resolution triangle mesh, for display purposes. We will assume that our display mesh is completely enclosed by the simulation mesh when then simulation mesh is undeformed.

Our goal is to transfer the motion of the simulation mesh to the display mesh. Remember that the FEM discretization defines the motion of our object, not just at the vertices, but everywhere inside the mesh (via the basis functions). Specifically, for any point in the undeformed space $\mathbf{X}$ I can reconstruct the deformed position $\mathbf{x}^t\left(\mathbf{X}\right)$ as long as I know which tetrahedron contains $\mathbf{X}$.

This gives us a simply algorithm to deform our display mesh. For each vertex in the display mesh ($\mathbf{X}_j$), find the tetrahedron, e, that contains $\mathbf{X}_j$ then move that vertex to position $N_e\mathbf{q}_e$. Here N_e and $\mathbf{q}_e$ are the basis function and generalized coordinates for the containing tetrahedron.

This can be expresses as a linear operation

$$\mathbf{x}^t_{\text{display}=W\mathbf{q}^t}$$

where $\mathbf{x}_{\text{display}}$ are the deformed vertex positions of the display mesh and W is the *skinning matrix* which contains the appropriate basis function values.

Assignment Implementation

Implementation Notes

For this course most functions will be implemented in **.cpp** files. In this assignment the only exception is that time integrators are implemented in **.h** files. This is due to the use of lambda functions to pass force data to the time integration algorithms. Finite element derivatives are both tricky and tedious to do correctly. Because of this you **DO NOT** have to take these derivatives by hand (unless you want to show off your mad skills). You can use a symbolic math package such as *Maple*, *Mathematica* or *Matlab*. **Note:** You can not use automatic differentiation, only symbolic math packages.

Running `a3-finite-elements-3d` will show a coarse bunny mesh integrated with linearly-implicit Euler. Running `a3-finite-elements-3d arma` will show a coarse armadillo mesh integrated using backward Euler. For each mesh you can toggle between the simulation and skinned meshes by pressing `S`. You can toggle integrators by pressing `N`. **Note:** the linearly-implicit integrator **WILL** explode when used with the armadillo mesh.

Important

Some of the functions from assignment 2 are reused here. If you correct implementation errors in those functions, your grades for the previous assignment will be updated to reflect that.

phi_linear_tetrahedron.cpp

Evaluate the linear shape functions for a tetrahedron. This function returns a 4D vector which contains the values of the shape functions for each vertex at the world space point x (assumed to be inside the element).

dphi_linear_tetrahedron_dX.cpp

Piecewise constant gradient matrix for linear shape functions. Row i of the returned matrix contains the gradient of the i^{th} shape function.

psi_neo_hookean.cpp

Compute the Neo-Hookean strain energy density.

dpsi_neo_hookean_dF.cpp

Compute the first Piola-Kirchoff (the gradient of the strain energy density with respect to the deformation gradient). You can use a symbolic math package to do this (but don't just look it up on the internet please).

d2psi_neo_hookean_dF2.cpp

Compute the hessian of the strain energy density with respect to the deformation gradient. You can use a symbolic math package to do this (but don't just look it up on the internet please).

T_linear_tetrahedron.cpp

Compute the kinetic energy of a single tetrahedron.

quadrature_single_point.h

Single point quadrature for a constant strain tetrahedron (CST).

V_linear_tetrahedron.cpp

Compute the potential energy of a single tetrahedron. **Note:** you will need both *psi_neo_hookean.cpp* and *quadrature_single_point.h* to do this.

dV_linear_tetrahedron_dq.cpp

Compute the gradient of the potential energy of a single tetrahedron. **Note:** you will need both *dpsi_neo_hookean_dq.cpp* and *quadrature_single_point.h* to do this.

d2V_linear_tetrahedron_dq2.cpp

Compute the hessian of the potential energy of a single tetrahedron. **Note:** you will need both *d2psi_neo_hookean_dq2.cpp* and *quadrature_single_point.h* to do this.

V_spring_particle_particle.cpp

The potential energy of a non-zero rest length spring attached to two vertices of the mesh. **Use your code from the last assignment**.

dV_spring_particle_particle_dq.cpp

The gradient of the spring potential energy. **Use your code from the last assignment**.

mass_matrix_linear_tetrahedron.cpp

Compute the dense mass matrix for a single tetrahedron.

mass_matrix_mesh.cpp

Assemble the full mass matrix for the entire tetrahedral mesh.

assemble_forces.cpp

Assemble the global force vector for the finite element mesh.

assemble_stiffness.cpp

Assemble the global stiffness matrix for the finite element mesh.

build_skinning_matrix.cpp

Build the skinning matrix that maps position from the coarse simulation mesh to the high resolution rendering mesh.

fixed_point_constraints.cpp

Use your code from the last assignment

pick_nearest_vertices.cpp

Use your code from the last assignment

linearly_implicit_euler.h

Use your code from the last assignment

newtons_method.h

Implement Newton's method with backtracking line search. Use the following parameter values: *alpha (initial step length) = 1*, *p (scaling factor) = 0.5*, *c (ensure sufficient decrease) = 1e-8*.

implicit_euler.h

Using your Newton's method, implement a fully implicit solver. **To ensure reasonable performance, use a maximum of five (5) iterations**.

Releases

No releases published

Packages

No packages published

Contributors 4

dilevin David I.W. Levin

abhimadan Abhishek Madan

SteveZhao13 Qi Zhao

Languages

● **C++** 76.4% ● **CSS** 21.0% ● **CMake** 2.6%