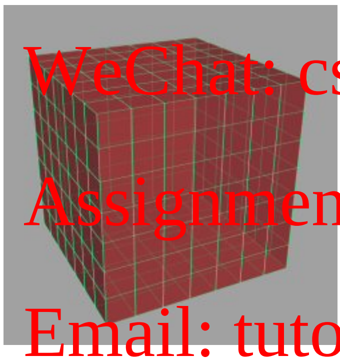
CSCI 520 Assignment 1: Simulating a Jello Cube

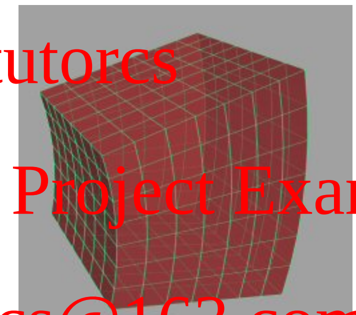
Due Wed Feb 14 2024, by 11:59pm

An Overview

For this assignment you will program [illegible] tion of a jello cube. The jello cube is made of elastic material, much like the jello sold in grocery stores. When the [illegible] will try to contract. When squeezed together, it will tend to inflate back to the original shape. In order to simulate th [illegible] nass-spring network, which is a very common tool in physically based modeling, especially with interactive [illegible]

Note: Please don't get discouraged b [illegible] ent. Starter code for this assignment already provides a lot of the code, including all of the OpenGL rendering [illegible] ration code. Once you have read this webpage carefully and understood the starter code, you will see that this as [illegible] ult – but it does take its time, so please start early.


Undeformed jello cube

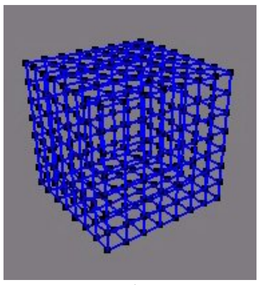

Deformed jello cube

The mass-spring system

You will model a cube of jello, which, when undeformed, has the shape of a cube of dimensions 1 meter x 1 meter x 1 meter. The cube will be constrained to stay **in the interior of a bounding box** of dimensions 4 meters x 4 meters x 4 meters. The center of the bounding box is located at the origin of the world-coordinate system. The axes of the bounding box are aligned with the world-coordinate system axes. The cube will stretch, contract, oscillate, change velocity, bounce off the walls of the bounding box, based on the physical laws for a **mass-spring system**. The cube will initially be positioned inside the box. When the cube reaches one of the walls of the bounding box, it should bounce off that wall back towards the interior of the bounding box.

A real-world jello cube is a continuous object, and if we were to simulate every small infinitesimal part of the elastic material, infinite memory would be required. For this reason, we will discretize the jello cube. This approach (*discretization*) is very common in phyisically-based modeling, and also applied mathematics and engineering in general. The cube will be modeled by 8 * 8 * 8 = 512 discrete mass points, all of equal mass. In the undeformed configuration, the mass points will form a uniform grid covering the volume of the cube. They will be positioned at (i / 7, j / 7, k / 7), where i, j, k are in { 0, 1, 2, 3, 4, 5, 6, 7 }. The rest length of all the springs will be determined based on the positions of the mass points in the undeformed configuration. As the cube deforms, the points will change position, giving us the shape of the cube at any given moment of time. Note that as the cube deforms, the points will no longer be the same distance apart from their neighbors. At places where the jello cube is squeezed, the points will be closer together, and where it is stretched, they will be farther apart than in the undeformed position.

The points with i=0 or i=7 or j=0 or j=7 or k=0 or k=7 are the points on the boundary of the cube. We will call these point *surface points,* since they define the surface of the jello cube. There are a total of 296 surface points. The surface of the cube can be rendered by drawing triangles connecting the surface points. The complete OpenGL code to do the cube rendering is provided in showCube.cpp. Interior points are not necessary for rendering, as they are invisible. They are, however, necessary for simulating physics.


The 296 surface control points, connected with structural springs (only structural springs connecting two surface points are shown).

Dynamics

You will model the movement of the cube by numerically solving a system of ordinary differential equations, which sounds perhaps a bit scary, but is actually not that difficult. The equations to be solved incorporate Newton's second law (F=ma), Hook's linear model of elasticity (F=kx), and linear damping (F=-kv). To solve the differential equations, you will use two methods: Euler integration and Runge-Kutta 4th order (RK4) integration. The starter code already provides code which implements both the Euler method and the RK4 method - look for it in physics.cpp. All you have to do is appropriately call either of these two methods during the simulation. For a particular simulation, you will typically just use one of the two methods - but you have to make sure that both methods work correctly with your code.

You will have to implement the *accel*[illegible]Acceleration, see physics.cpp). This function takes as input the positions of the 512 nodes, the velocities of the 512 r[illegible]arameters of the model. It returns the acceleration for each of the 512 points. It also adds any effects of an external [illegible] present of course. In general, the acceleration will of course be different for each of the 512 simulation points. To[illegible], the function must take into account the forces due to

- structural, shear and bend sprir[illegible]
- external forces (force field, if a[illegible]
- bouncing off the walls.

Initial conditions

Initially, the cube will be in some prescribed **deformed** position, with some prescribed initial velocities. Your program will read the information about the initial position of the cube, initial velocity, and various other simulation parameters from a file on disk. The movement of the jello cube will depend on the information given in this file. We shall call these files 'world' files, because they describe the initial world (state) of the jello cube. The default extension for these files is .w (example: jello1.w) . Several world files are provided with the assignment, so that you can test your code. Included in the starter code is the routine **loadWorld** which loads all the information from a world file into processor memory. There is also a routine **writeWorld** which works in the opposite direction: it saves the current point positions, velocities, and other parameters to a valid world file on disk. See the comments in the input.cpp file for parameters to these two routines. Also, the starter code provides a separate application called **createWorld**. It can be found in the starting package (createWorld.cpp) . This application allows you to create your own world files without having to manually edit world files. For more info about createWorld and for a description of the structure of a world file, click here.

The cube will deform as a result of different velocities at different locations inside the cube. The velocity differences will happen for three reasons:

- Initial velocities (specified by the world file) might not be the same everywhere inside the cube.
- There might be an external force field (time-independent, as described below).
- Cube bounces off the wall of the bounding box or off some other collision object.

Contact with the bounding box

You have to implement collision detection and response for the jello cube hitting any of the six walls of the bounding box.

External Force field

The world file might also specify an external non-homogeneous time-**in**dependent (i.e. possibly varying with location, but not with time) force field. If the force field is present, you must model its effect on the cube. The force field is given as an array of vectors, each specifying the 3D force vector (in Newtons) at a node of a 3D discrete grid. Note that this is a different grid than the jello cube grid; this force field grid covers the entire scene bounding box. Use interpolation to obtain values of the force field at an arbitrary position inside the bounding box. The grid is uniform, covers precisely the volume of the bounding box, and consists of n x n x n equally spaced points, where the parameter n is the *resolution* of the grid. It is specified in the world file. For this assignment, you need to support values of 2 <= n <= 30.

Helper slides

The provided slides (PDF | PDF, black and white, good for printing) contain information on:

- how to organize the springs (structural, shear, bend springs),
- how to compute forces due to springs,
- how to implement collision detection and bouncing response,
- and many other tips.

Rendering

To obtain a good view of the scene, the camera has to be positioned appropriately. The starter code already provides a complete code to do this (see jello.cpp). It allows the user to move the camera around in real-time, and observe the jello-cube from different angles. The description of the method to position the camera is here.

Your program will allow two rendering modes: wireframe and triangle. In the wireframe mode, only the surface points (no filled triangles) are shown, together with the *structural* springs connecting them. Structural springs are one of the three kinds of springs that are used with the mass-spring system. You can learn more about this from the provided slides (see the Helper slides section). In the triangle mode, surface of the cube is rendered as triangles (and the surface points are vertices of these triangles). Start-up code provides supports both for the wireframe and triangle rendering mode. It is possible for the user to switch among the two rendering methods at run-time, by pressing 'v' (see also showCube.cpp). Also, in wireframe mode, user can independently turn on/off the display of structural springs, shear springs and bend springs. Thus, any of the 8 on/off combinations can be selected. Pressing 's','h','b' toggles the display of structural, shear and bend springs, respectively. Note that to avoid screen clutter, only the springs connecting two surface points are shown, but the simulation of course has to simulate the structural, shear and bend springs appropriately connecting all the 512 simulation points.

In the triangle mode, the jello is illuminated using proper OpenGL lighting. Starter code provides a complete lighting environment in jello.cpp . You are free to change lighting parameters, or make other rendering improvements/modifications. This way, you can personalize your assignment – otherwise all the cubes for the entire class will look the same (except for any differences in implementation of physics of course), which would make for a quite boring final assignment movie.

Extra credit

The following are some suggestions:

- Implement collision detection w[illegible] object, such as an inclined plane, sphere, a cone, a spiral, etc.
- Make the animation interactive. [illegible]vely (i.e. while the program is running) push the cube in a certain direction by dragging the mouse. The cube s[illegible]dingly. Apply the user force equally to all the simulation points of the cube, regardless of where on the scre[illegible]ed.
- Extra extra credit: Allow the user to push only a certain (surface) simulation point, by selecting that simulation point on the screen and then pulling on it by dragging the mouse. For this, you will need to make use of *selection and feedback* capabilities of OpenGL (see OpenGL Red Book, Chapter 13). Or, you can distribute the force, making it larger closer to the cube location where the user pulled with the mouse. You can make the force falloff radius into a parameter, potentially modifiable at runtime. This will allow you to control how localized deformations your system will produce (extremely localized if force only applies to one simulation point, or not localized at all if all simulation points always receive the same user force; and anything in between).

Also, you might be awarded credit for any other creative or interesting solution to an issue in your implementation, as long as it is well-documented in your readme file. So, if you can think of any interesting feature, feel free to implement it.

Please note that the amount of extra credit awarded will not exceed 25% of this assignment's total value.

Starter code

This assignment consists of two applications: jello and createWorld. Starter code provides a complete implementation for createWorld, which is a tool to design your own physical environments. createWorld is a single-file application, and independent from the rest of the assignment. Its implementation is createWorld.cpp .

You will be developing the application called 'jello'. The main files for this application are jello.h and jello.cpp, but several other files are used as well.

In the starter package, all of these files already contain code for pre-implemented functionality such as lighting, camera positions, integrators, etc. You can modify any of these files in any way you want, and you can add your own files, if you need to. Also in the starter package, there are some examples of world files. You can download the complete starter package here.

Installing OpenGL and GLUT: Windows, Linux and Mac OS X come with OpenGL preinstalled. On Linux and Mac, GLUT will also already be installed. If you are using Mac OS X Mojave, make sure you have updated the OS to the latest version; otherwise, you may get erroneous behavior such as a black screen. For Windows, we provide GLUT with the starter code. You should be able to build the starter code in MS Visual Studio 2017 by opening hw1.sln and clicking "Build". There are two projects: "jello" and "createWorld"; "jello" is the main driver. Note: Because there are many versions of Windows 10 (different build versions), you may get an error:

```
2>C:\Program Files (x86)\Microsoft Visual Studio\2017\Community\Common7\IDE\VC\VCTargets\Microsoft.Cpp.WindowsSDK.targets(46,5): error MS
The Windows SDK version 10.0.XXXXX was not found.
Install the required version of Windows SDK or change the SDK version in the
project property pages or by right-clicking the solution and
selecting "Retarget solution".
```

This can be solved, as the message says, by right-clicking the solution and selecting "Retarget solution".

An example workflow to do the assignment

Here is one suggested workflow for solving the assignment. You can of course take alternative routes.

1. Make sure starter and createWorld code compiles.
2. Run the application, and make sure OpenGL works (you can open a GLUT window, etc.)
3. Understand the createWorld/loadWorld/saveWorld functionality.
4. Familiarize yourself with the provided routines which, given a world structure, render the surface of the cube. Be able to use both the wireframe and the triangle mode. See how the camera and lighting work. Experiment with different world files. Personalize your assignment by changing the lighting parameters, or change/improve the lighting environment in some other interesting way.
5. Do a test: move the points of the cube according to a prescribed made-up (non-physical) sequence of deformations, and observe if the cube on the screen changes accordingly. To do this, hard-code the modifications of the world structure as time evolves (i.e. don't use the physical model, but just model some simple movement, e.g. by manipulating positions and velocities in some simple way as the time evolves). For example, increment the z-value of all positions. See if the cube is moving appropriately on the screen.
6. Implement the acceleration function (computeAcceleration, see physics.cpp). For now, however, don't implement any collisions or the force field.

7. Test the acceleration function:
   Make up a world structure, which you will use for the test: Position the points at some position, assign certain velocities, use the Euler or RK4 integrator, assign spring constants, ... As a starting point, you should definitely use one of the provided world structures. Selecting good values for elasticity and damping factors can be tricky. See the tips below.
8. Implement collision detection and response for bouncing off the wall. Test it.
9. Implement the force field. Test it. Make up some force fields which create really cool effects.
10. Extra credit
11. Make the animation jpegs.
12. Submit assignment. Congratulations!!

Grading Criteria

Your program must:

- Animate the movement of a jell[illegible] physical model.
- Must incorporate a bounding b[illegible]ction and response.
- Must implement an arbitrary no[illegible]ependent force field, with appropriate force-field interpolation.
- Use the provided code to intera[illegible], use proper lighting, and render the cube in wireframe and triangle mode.
- Read the description of the world from a world file and simulate the cube according to this information. Your program should work with arbitrary world files. Your program, of course, need not work correctly for world files containing invalid data, bad integrator parameters (blow up effects and similar), bad elasticity, damping parameters and other physical parameters, or bad position or velocity parameters.
- Run at interactive frame rates (>15fps at 640x480)
- Be reasonably commented and written in an understandable manner
- Be submitted along with JPEG frames for the required animation.
- Be submitted along with a **README file** documenting your program's features and describing the approaches you took to each of the open-ended problems. This is especially important if you have done something spectacular for which you wish to receive extra credit!

Animation Requirement

In addition to your code and the README file, you are required to submit an animation, represented by a series of JPEG images which are screenshots from your program. Do not exceed 300 frames. The frame rate for the animation should be 15 fps, which corresponds to 20 seconds of animation running time. Please name your JPEG frames 000.jpg, 001.jpg, and so on. Note that 15 fps here refers to the frame rate at which the animation is to be played. It is NOT the frame rate of your program when you are creating the screenshots. Your program will of course slow down when creating the screenshots because it has to save the files to disk. This is normal.

In the animation, you should try to demonstrate your results in some interesting way. Show your OpenGL lighting skills, apply texture, and above all, show interesting deformations and dynamics. In order to create the animation, you can create your own world files, as you see fit.

How to create the animation: Starter code includes the functionality to output the current screen to a "Portable Pixel Format" (PPM) file. You can always take the screenshot of the current screen by pressing spacebar. In order to capture an animation, comment the "saveScreenToFile=0" line in the "doIdle" routine, and press spacebar to launch the continuous frame capture. You can then convert the PPM files to JPEG using any image manipulation program that supports batch file conversion. For example, you can use the freely available IrfanView program.

Submission

Please submit:

- Your complete source code
- A compiled executable (Windows or Mac OS X)
- Animation JPEGs

Please submit the assignment using the USC Blackboard. In order to do so, create a single zip file containing your entire submission. Submit this file to the blackboard using the "Attach File" field. After submission, please verify that your zip file has been successfully uploaded. You may submit as many times as you like. If you submit the assignment multiple times, we will grade your LAST submission only. Your submission time is the time of your LAST submission; if this time is after the deadline, late policy will apply to it.

Note: The instructor will read your code. Quality of your code and comments might affect your grade. This is an individual assignment. Your code might be compared to code of other students using a code comparison tool.

Tips

- Use double precision for all internal calculations. That is to say, declare all the floating-point variables as 'double'.
- The provided code uses some elements of C++, such as declarations in the middle of the code, and parameter passing by reference (&). It doesn't use object-oriented programming. Under Linux/Mac OS X, you have to compile the code with 'g++', not with 'gcc'.
- You can measure your program's frame rate as follows.
- Default key and mouse setup as provided by input.cpp:
  - ESC: exit application
  - v: switch wireframe/triangle mode
  - s: display structural springs on/off
  - h: display shear springs on/off

- b: display bend springs on/off
- space: save the current screen to a file, filename index increments automatically
- p: pause on/off
- z: camera zoom in
- x: camera zoom out
- right mouse button + move mouse: camera control
- e: reset camera to default position

- Don't over-stretch the z-buffer. It has only finite precision. A good call to setup perspective is:

```
gluPerspective(90.0,1.0,0.01,1000.0);
```

A bad call would be:

```
gluPerspective(90.0,1.0,0.[illegible]
```

or even worse:

```
gluPerspective(90.0,1.0,0.[illegible]
```

In the last two examples, the problem is that the ratio between the distance of the far clipping plane (=last parameter to gluPerspective), and the distance of the near clipping plane (=third parameter to gluPerspective) is way too large. Since the z-buffer has only finite precision (only a finite number of bits to store the z-value), it cannot represent such a large range. OpenGL will not warn you of this. Instead, you will get all sorts of strange artifacts on the screen and your scene will look nothing like what you intended it to be.

- The starter code uses the Gouraud shading model to do the lighting. Since there are only 64 points on every face, you might experience some aliasing effects along the edges of the triangles.
- Tips for choosing elasticity and damping constants:
  - If you set any of these constants (elasticity or damping) to a too large value, your simulation will "blow up" (if it's a collision parameter, the simulation will blow up when the cube reaches a wall). Blowing up means that the mass points of your cube will fly towards infinity. It is a good idea to add some code which will make the simulation terminate when it detects that point coordinates have become unreasonably large. To avoid the blow-up effect, decrease the value of timestep, or decrease the value of the elasticity/damping parameter. Euler tends to blow up much faster than RK4, so you have to decrease the parameter more for Euler than for RK4.
  - Euler is faster than RK4, but less accurate and much more unstable. It requires smaller values of the timestep to avoid blowing up.
  - Without damping, the simulation will be unstable.
  - A general rule of thumb is that the time step should be inversely proportional to the square root of the spring elasticity [Courant condition]
  - A time-step of 0.0005 is a good start.

- Start this assignment *as soon as you can*. It is a significant endeavor, with many intermediate steps. If you wait until a few days before the assignment is due, you probably will not finish. This project is a lot of fun if you're not rushed, and if enough time is put in the end product is something fun that you can show off to your friends.
- Experiment with your own ideas and **have fun**.

written by Jernej Barbic, USC