15-462 Project 4: Simple Physics Engine

Release Date: Thursday, March 26, 2015

Checkpoint: Thursday, April 9, 2015, 23:59:59

Due Date: Tuesday, April 28, 2015, 23:59:59

Starter Code:
http://www.cs.cmu.edu/afs/cs/academic/class/15462-s15/www/project/p4physics.tar.gz

Useful references:
OpenGL Reference Pages: http://www.opengl.org/sdk/docs/man/
SDL Documentation/Tutorials: http://www.libsdl.org/

1 Overview

In this lab, you will learn about physical interactions such as collisions and the animation of simple objects. The project consists of writing a physics engine that simulates both spheres, triangles, planes, and static models. Rendering will be done with OpenGL.

This project is designed to teach you about physical interactions and animation. Also, as a more general concept, you will learn about detecting intersections between 3D volumes. The project requires that you be able to simulate spheres, static triangles, and meshes; the creation of additional physical primitives is open to you, but not required. By the end of the project, you will be able to understand and program concepts dealing with physics engines.

2 Description

The main part of this project is collision detection. You will have to program the logic for elastic collisions between two spheres, a sphere and a triangle, a sphere and a model, and a sphere and a plane. You will not have to program collisions between triangles or models since all of the triangles and models in the scenes we will give you are static bodies.

For the simulation of physical bodies, you will have to apply both direct force and torque to spheres. This means that not only will the position of spheres change, but also the orientation.

You will also program springs, which are relatively simple to implement, but add much functionality to your physics engine.

The last physics concept you have to implement is two kinds of damping, which is a nice way to simulate friction.

3 Submission Process and Handin Instructions

Failure to follow submission instructions will negatively impact your grade.

1. Your handin directory may be found at one of the two directories below, depending on which class you are enrolled in. Graduate students should be enrolled in 15-662 and undergraduates in

15-462.

`/afs/cs.cmu.edu/academic/class/15462-s15-users/`*andrewid*`/p4physics/`.

`/afs/cs.cmu.edu/academic/class/15662-s15-users/`*andrewid*`/p4physics/`.

All your files should be placed here. Please make sure you have a directory and are able to write to it well before the deadline; we are not responsible if you wait until 10 minutes before the deadline and run into trouble. Also, remember that you must run `aklog cs.cmu.edu` every time you login in order to read from/write to your submission directory.

2. You should submit all files needed to build your project, as well as any textures, models, shaders, or screenshots that you used or created. Your deliverables include:
   - `src/` folder with all `.cpp` and `.hpp` files.
   - CMake build system.
   - `writeup.txt`
   - Any models/textures/shaders needed to run your code.
3. Please **do not** include:
   - Anything in the `build/` folder
   - Executable files
   - Any other binary or intermediate files generated in the build process.
4. Do not add levels of indirection when submitting. For example, your source directory should be at `.../`*andrewid*`/p4physics/src`, **not** `.../`*andrewid*`/p4physics/myproj/src` or `.../`*andrewid*`/p4physics/p4physics.tar.gz`.
   Please use the same arrangement as the handout.
5. We will enter your handin directory, and run `cd build && rm -rf * && cmake ../src && make`, and it should build correctly. **The code must compile and run on the GHC 5xxx cluster machines**. Be sure to check to make sure you submit all files and that it builds correctly.
6. The submission folder will be locked at the deadline. There are separate folders for late handins, one for each day. For example, if using one late day, submit to `.../`*andrewid*`/p4physics-late1/`. These will be locked in turn on each subsequent late day.

4 Required Tasks

A very general overview of the implementation requirements is as follows. Refer to subsequent sections of the handout for more details.
Input: We provide you with an application, empty functions to implement your program, and different scenes to simulate. The scenes are a direct extension of the raytracer scenes.
Output: You must fill in the functions provided with your implementation of sphere and triangle collisions. Using your implementation of an Ordinary Differential Equation, you must simulate the position and orientation of a sphere. You need to program the function to apply force and implement springs which can be anchored to spheres, triangles, or models.
You must be able to correctly simulate all of the scenes we provide.
Requirements:

- Implement collisions between spheres and other spheres, triangles, planes, and models.
- Implement simulation of both velocity and angular velocity for spheres along with force and torque.
- Use Runge-Kutta 4th Order to integrate bodies.

- Program the spring interactions between bodies.
- Add gravity (the value is given as a member in the physics class).
- Implement collision damping and spring damping. The collision damping value is given as a member of the physics class. Spring damping is a member of the Spring class.
- Implement bounding volumes for your objects and a bounding volume hierarchy such as an octree to allow for quicker intersection tests.

`writeup.txt` should contain a description of your implementation, along with any information about your submission of which the graders should be aware. Provide details on which methods and algorithms you used for the various portions of the lab. Essentially, if you think the grader needs to know about it to understand your code, you should put it in this file. You should also note which source files you edited and any additional ones you have added.

Examples of things to put in `writeup.txt`:

- Mention parts of the requirements that you did not implement and why.
- Describe any complicated algorithms used or algorithms that are not described in the book/handout.
- Justify any major design decisions you made, such as why you chose a particular algorithm or method.
- List any extra work you did on top of basic requirements of which the grader should be aware.

There is also opportunity for extra credit by implementing things above the minimum requirements. See section 9 for more details.

5 Starter Code

We recommend that you start by reading the given starter code. Most of the files should be familiar to you by now. The `README` gives a breakdown of each source file, the most important of which are `physics/physics.cpp`, `physics/spherebody.cpp/hpp`, `physics/spring.cpp/hpp`, `physics/collisions.cpp/hpp`.

5.1 What You Need to Implement

`physics/spherebody.cpp` contains three functions, step_position, step_orientation, and apply_force that you need to fill out to complete the simulation of spheres. It is assumed that the spheres have uniform density (which will affect how torque is applied). For more information on the equations needed for these functions, see the Useful Equations sheet in the back of the writeup. The step functions are intended to point you in the right direction for setting up RK4. Feel free to add additional functions to help you out.

`physics/collisions.cpp` contains the four versions of the collides function that you need to implement. These will be for sphere to sphere collisions, sphere to triangle collisions, sphere to model collisions, and sphere to plane collisions. These functions need to detect if there is a collision between the two bodies passed in as arguments and perform elastic collisions when there is. Use of helper functions in this file is encouraged. Failure to do so will reflect in your grade.

`physics/spring.cpp` contains the step function which should apply forces (based on Hooke's Law) to the two bodies which are attached to the spring. Must also implement damping for the spring.

`physics/physics.cpp` contains the step function for the world which should use all of the functions you implement in the previous files. Need to detect all valid collisions (for example, triangle to triangle never happens unless dynamic triangles have been implemented).

5.1.1 What We Give You

We provide you with the code which will run your physics engine and load the scenes. Please check with the course staff before you modify any of this code since it is critical for testing your physics implementation. There are also scenes for you to simulate with your physics engine. As always, you are encouraged to create your own.

- Scenes with both dynamic and static bodies for you to simulate (we take care of loading these scenes for you)
- Event code to move the camera

6 Grading: Visual Output and Code Style

Your project will be graded both on the visual output (both screenshots and running the program) and on the code itself. We will read the code.

In this assignment, part of your grade is on the quality of the visuals, in addition to correctness of the math. So make it look nice. Extra credit may be awarded for particularly good-looking projects.

Part of your grade is dependent on your code style, both how you structure your code and how readable it is. You should think carefully about how to implement the solution in a clean and complete manner. A correct, well-organized, and well-thought-out solution is better than a correct one which is not.

We will be looking for correct and clean usage of the C language, such as making sure memory is freed and many other common pitfalls. These can impact your grade. Additionally, we will comment on your C++-specific usage, though we will generally be more lenient with points (at least earlier in the semester). More general style and C-specific style (i.e., rules that apply in both C and C++) will, however, affect your grade.

Since we read the code, please remember that we must be able to understand what your code is doing. So you should write clearly and document well. If the grader cannot tell what you are doing, then it is difficult to provide feedback on your mistakes or assign partial credit. Good documentation is a requirement.

7 Implementation Details

7.1 Object Simulation

7.1.1 Forces

You must be able to apply both linear forces and torque to spheres in your physics engine. Every time we apply a force, there is an offset from the center of mass upon which it is applied. If the offset is zero, or the direction and the offset from the center of mass are parallel, then just linear force is applied.

This can be modeled with Newton's 2nd Law of Motion, which states:

$$F = ma \tag{1}$$

When the offset from the center of mass is not zero or parallel to the direction of force, we split the force into two components, the linear and angular. The linear force is the component of the force which is parallel to the offset from the center of mass. The angular component, or torque, is the perpendicular component. This can easily be found by taking the cross product of the offset and the force. The formulas for finding the angular acceleration (for spheres) are

$$I = \frac{2}{5}mr^2 \tag{2}$$

$$\alpha = \tau / I \tag{3}$$

Where α is the angular acceleration, I is the moment of inertia, and τ is the torque. You only have to calculate angular acceleration for spheres since no other geometry is dynamic.

For modifying the orientation of the physics objects (stored as a quaternion) by a set of angles, we suggest looking at the yaw, pitch, and roll functions located in math/camera.cpp.

7.1.2 Integrators

This simulation falls into a category of problems known as initial value problems. Initial value problems are classified by equations of the form:

$$\dot{x} = f(x, t) \tag{4}$$

Here f is some function of x, some state of our system, and t, time. $\dot{x}$ is the change in x with respect to time. Basically, in this system, we are given some initial state $x_0 = x(t_0)$ and wish to follow x in time, hence the name 'initial value problem'.

7.1.3 Runge-Kutta 4

One method of solving initial value problems is to find numerical solutions using differential equation solvers, or integrators. For this project, you must use Runge-Kutta 4 (or a better method) to integrate the position and orientation of your physics objects.

$$k_1 = hf(x_0, t_0) \tag{5}$$

$$k_2 = hf(x_0 + \frac{k_1}{2}, t_0 + \frac{h}{2}) \tag{6}$$

$$k_3 = hf(x_0 + \frac{k_2}{2}, t_0 + \frac{h}{2}) \tag{7}$$

$$k_4 = hf(x_0 + k_3, t_0 + h) \tag{8}$$

$$x(t_0 + h) = x_0 + \frac{1}{6}k_1 + \frac{1}{3}k_2 + \frac{1}{3}k_3 + \frac{1}{6}k_4 + O(h^5) \tag{9}$$

Here we have some initial state $x_0 = x(t_0)$. We can estimate x at some later time, which we denote as $t_0 + h$. The parameter h is our time step. Thus, we can evaluate $x(t_0 + h)$ by starting at our initial value and adding fractions of each k_i

If you are curious or ambitious and want to try writing a more practical integrator, we discussed better integrators in class, and there are plenty of better integrators described online.

7.1.4 Collision Detection

First of all, it is a good idea to note that physical simulation is imperfect. There are certain pitfalls that are not necessarily intuitive to a physics model which are required to prevent odd behavior. Collision Detection is one of these areas.

You should only register a collision if the relative velocity of one object to the other is positive. That is, the object is traveling towards the other. If this is not true, ignore the collision.

It should also be noted that velocities are computed directly from the collisions. Forces do not come into play for this part of your code.

There are four types of collision detections you must program. The first is the sphere to sphere collisions detection. This is one of the simplest collision detections there is. Both spheres have a center point. If the distance between these two points is less than the sum of the radii, then there is a collision. The difficulty from this problem comes from performing elastic collisions. We have to calculate the velocities for each sphere after the collision. It turns out that this is much easier if one body is at rest. We can use the same equations if we zero out one of the velocities and subtract the velocity from the other.

$$v_1' = v_1 - v_2 \tag{10}$$

$$v_2' = (0, 0, 0) \tag{11}$$

Next we need to calculate the new velocities. We can do this with the following equations.

$$d = \frac{p_2 - p_1}{|p_2 - p_1|} \tag{12}$$

$$v_2'' = 2d\frac{m_1}{m_1 + m_2}(v_1' \cdot d) \tag{13}$$

$$u_2 = v_2 + v_2'' \tag{14}$$

$$u_1 = \frac{m_1 v_1 + m_2 v_2 - m_2 u_2}{m_1} \tag{15}$$

Where u_1 and u_2 are the new velocities, p_1 and p_2 are the centers of the respective spheres, m_1 and m_2 are the mass for each sphere, and d is the direction from the first sphere to the second.

Next we need to do collision detection with a plane. We do this by projecting the center of the sphere onto the plane. We give you a point on the plane and the normal of the plane. Using the following equations will detect a collision with a plane

p_1 is the center of the sphere
p_2 is the point on the plane
n is the normal of the plane

$$a = p_1 - p_2 \tag{16}$$

$$d = a \cdot n \tag{17}$$

If $abs(d)$ is less than the radius of the sphere, there is a collision. For the new velocity of the sphere, since we are doing elastic collisions, we use the following equation

$$u = v - 2(v \cdot n)n \tag{18}$$

The third collision you have to detect is between a sphere and a triangle, and this solution can be extended to encounters of the fourth kind, between a sphere and a model. This is a problem that is best broken up into many parts. We can use some of the things we have already done such as sphere-plane collision. If the point projected onto the plane is within the triangle, we can just check the distance to that point. To find the point on the plane define by the triangle, just take d as defined from sphere to plane collision and use it to find the new point.

$$p' = p - dn \tag{19}$$

This just uses barycentric coordinates like in the raytracer project. If it is not within the barycentric coordinates, the easiest way to handle this is to project the point onto each of the edges of the triangles and check the distance to that point. You can use the following equation to project a point onto a line.

$$p'' = \frac{((p' - a) \cdot (b - a))(b - a)}{|b - a|^2} + a \tag{20}$$

7.1.5 Springs

If you already have forces, this is straightforward to implement. It should be noted that springs are the only part of your physics engine which causes torque. The equations you need for springs is Hooke's Law, and the damping force from a spring.

$$F = -kx - c\frac{dx}{dt} \tag{21}$$

Where x is the displacement from the equilibrium position, k is the spring constant, and F is the resulting force. c is the damping constant for the spring which is multiplied by the derivative of the displacement with respect to time. Note, $\frac{dx}{dt}$ is not the velocity, it is the velocity component parallel to the direction to the other physics body.

7.2 Damping

The last thing you have to implement is damping. You already went through damping for springs, but you will also have to implement damping for collisions. For collision damping (as discussed in class), we have the equation

$$v' = v - cv \tag{22}$$

When there is a collision, we apply this to the new velocities. v' is the damped velocity, v is the velocity after the collision, and c is the damping constant.

To make things reasonable (so that we don't keep on making minor changes), when the magnitude of the velocity is less than some ε, just set the velocity to 0. This is part of a physics engine optomization where physics objects are disabled until something interacts with them. Objects that are disabled do not have to be updated.

7.3 Bounding Volume Hierarchy

Modern real-time physics engines implement collision detection in a two-pass system: In the first, "broad-phase" pass, simple bounding volumes are tested to quickly eliminate pairs of objects that are too far apart to be intersecting. A second, "narrow-phase" pass then performs full collision tests on the remaining pairs.

The intersection tests you've written so far would be used in "narrow-phase" collision detection - they take a pair of objects and perform a full intersection test. You could apply these methods to every possible pair of objects, but this is wasteful and can be very slow for scenes with many objects.

Instead, we would like you to implement a simple broad-phase pass in your simulator by constructing a hierarchy of bounding volumes around objects. You can perform fast, conservative collision tests on pairs of objects using only bounding volumes, and more importantly you can ignore collisions between objects which lie in different regions of the scene.

The design and implementation are up to you, but we suggest a loose octree as a good starting point. By dividing space recursively, you separate objects into the disjoint regions of space that they occupy. Thus, for an object whose bounding volume is entirely contained within the bounding volume of an octree node, you do not need to test collisions against any objects not inside the same node.

At a minimum, you should create bounding volumes for triangle meshes, and organize your models into some hierarchy that allows for less than $O(n^2)$ full collision tests except in extremely bad cases.

Your implementation will also need to be fast enough to allow modifying the data structure (e.g. moving objects from one node to another as they move through the scene), and ideally should not require reconstructing the entire data structure each frame.

7.4 Suggested Sequence

We suggest you implement the assignment in the following order:

1. Have a look at the provided code.
2. Implement sphere to plane collisions. This is the simplest case.
3. Implement sphere to triangle, sphere to model, and sphere to sphere collisions.
4. Get spheres to simulate falling down by adding gravity (gravity is defined in the scenes, so do not set the gravity of the world in your code).
5. Implement the Runge-Kutta 4 integrator.
6. Add functionality for springs.
7. Make sure that your simulations do not lose or gain energy (outside of the bounds of error and damping). Note that scenes without forces should have constant energy.
8. Add bounding volumes and a bounding volume hierarchy to your implementation.
9. Consider expanding your physics engine with additional functionality.

8 Checkpoint

For this assignment, you will be required to submit your progress once before the final due date. This checkpoint will be on Thursday, April 9th. By this time, you should have the following aspects of your project working:

- All necessary collision detection functions
- An integrator which will allow motion and collisions to be displayed visually (it does not yet have to be Runge-Kutta 4, so you may use the simple Euler method)
- Acceleration due to gravity (again, you may use the simple Euler method)

Please include in your checkpoint submission a writeup that lists what you did or did not complete. You should feel free to complete more of the project than just these three things, but these are what we will be looking for from everyone in this checkpoint.

Failing to submit anything for the checkpoint will negatively impact your grade, so please remember this deadline, and submit even if you have not yet completed everything in the above list.

9 Extra Credit

Any improvements, optimizations, or extra features for the project above the minimum requirements can be cause for extra credit. Extra credit is generally awarded for impressive achievements beyond the project requirements, at the discretion of the graders.

Ideas may include but are not limited to:

- Make triangles and models dynamic. This could include creating triangle to triangle collisions, adding mass, approximating moment of inertia for meshes, etc.
- Implement an interesting game on top of your physics engine. The pool table scene is there for this purpose (otherwise the scene should just seem to be static).

- Write a different integrator for your program. Runge-Kutta 4 is extremely popular in simulations, but you may also wish to investigate one of the following integration schemes: Verlet, Leapfrog, sympletic, and implicit (in order of increasing difficulty to implement).
- Add some interesting shaders. **Warning:** Copying your shaders over from project 3 is fine, but you will not receive any additional points for shaders that have already received extra credit. As in project 3, this does not mean copying some shader that you find online and fiddling with it a little.
- Make some interesting scenes. For example, demonstrate your simulator's ability to simulate cloth, using small masses and springs.

10 Words of Advice

10.1 General Advice

- As always, start early.
- Be conscious about your code. Performance is a large issue for this project.
- Be careful with memory allocation, as too many or too frequent heap allocations will severely degrade performance.
- Make sure you have a submission directory that you can write to as soon as possible. Notify course staff if this is not the case.
- While C has many pitfalls, C++ introduces even more wonderful ways to shoot yourself in the foot. It is generally wise to stay away from as many features as possible, and make sure you fully understand the features you do use.