COMPSCI 220 Fall 2017
Assignment 9: Real-Time Computing

Out: 28 November, 2017
Due: 5 December, 2017

General Instructions For All Assignments

Assignments in COMPSCI 220 are to be programmed in C++11, using the virtual machine image provided for the class. All assignments must follow the instructions on the handouts exactly, including instructions on input and output formatting, use of language features, libraries, file names, and function declarations. Failure to do so will automatically result in docked points. All assignments are individual: you must not discuss assignment contents, solutions, or techniques with fellow students. Please review the course policies on the course website, `https://people.cs.umass.edu/~joydeepb/220R`.

Instructions For This Assignment

In this assignment, you will be writing optimized code to simulate and render a fountain of particles in real time. Your code must be written in the file `assignment9.cpp`, and it must be accompanied by a header file `assignment9.h`. You may have helper functions in your submission, as long as there is no `main` function. You are not allowed to use any library other than the Google Testing library, CImg, and iostream. Your submission must use the simulation and visualization constants provided in the handout - you may vary them to experiment on your own, but we will be verifying correctness of your submission using the constants provided.

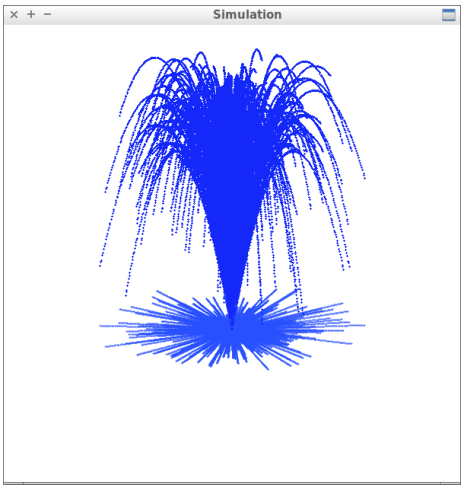

Figure 1: Visualizing the simulation of a fountain of particles.

1 File Structure and Integration

Simulation and animation of a physical system will require several software components working together. Those components are: data storage, forward simulation, and animation. We strongly recommend that you write and test these components individually, before integrating them. This practice will drastically reduce the amount of time spent locating bugs.

The following is an outline of the included files and their purpose:

1. `assignment9.cpp`: This file contains the definitions for the Particle struct, as well as some helper functions. You will will need to modify this file.
2. `assignment9.h`: This file contains the declarations for the particle struct, as well as some helper functions. You may modify this file.
3. `simulation.cpp`: This file contains a main function for running the simulation. DO NOT MODIFY THIS FILE.
4. `vector3.h`: This file contains the definition of the Vector3 class. DO NOT MODIFY THIS FILE.
5. profile: This is the profiling script demonstrated in lecture 20 - you may use it to identify computational bottlenecks in your program.
6. `reference_simulation`: This is a pre-compiled reference implementation executable that you can use to compare the performance of your code to.

The `Vector3` class (defined in `vector3.h`) is used to store data about positions and velocities of particles. It also supports some arithmetic operations such as addition, and multiplication by a scalar. The get and set methods will be necessary to read and write any updates which occur to the state of a particle.

`GetMonotonicTime()` and `Rand()` (defined in `assignment9.cpp`) are functions which will help you time the execution of your code and randomly initialize particle states, respectively.

2 Representing And Simulating Particles

Declare a `Particle` struct in `assignment9.h`, and the associated function definitions in `assignment9.cpp` with the following members. Particle states must be represented using only the Vector3 class.

1. Member variables to store the state of the particle:

   ```
   Vector3 p0;
   Vector3 v0;
   Vector3 p;
   double t0;
   ```

   The time of creation of the particle is stored in `t0`, the initial 3D location of the particle in the variable `p0`, and the initial 3D velocity vector in the variable `v0`. The current 3D location of the particle will be updated to the variable `p`.

2. A default constructor,

```
Particle();
```

which will set the time of creation of the particle, and randomly initialize the initial location and initial velocity of the particle according to the following equations:

$$p_0 = (0, 0, 0) \tag{1}$$

$$v_0 = \alpha[\sin(\phi)\cos(\theta), \sin(\phi)\sin(\theta), \cos(\phi)]. \tag{2}$$

Here, α is the speed of the particle, and the direction of travel is given by ϕ (the angle from the vertical axis) and θ (the angle from the horizontal, x-axis). α, ϕ, and θ are randomly initialized using the following equations:

$$\alpha = 9.0 + \mathcal{U}(0.0, 2.0) \tag{3}$$

$$\phi = \mathcal{U}(0.0, \frac{\pi}{18}) \tag{4}$$

$$\alpha = \mathcal{U}(0.0, 2\pi) \tag{5}$$

$\mathcal{U}(a, b)$ denotes a uniform distribution between a and b.

3. The update function,

```
void Update();
```

which will update the particle's location `p` according to the following equations:

$$\Delta t = t - t_0 \tag{6}$$

$$p = p_0 + v_0\Delta t + \frac{1}{2}g\Delta t^2 \tag{7}$$

Here, `t` is the current time, and `g` is the constant gravity vector $[0, 0, -9.8]$.

Hint : You may recognize equation (7); it is one of the basic kinematics equations for projectile motion, and describes the position of a projectile (in this case, a particle) given it's initial position, velocity, and the acceleration. Changing p_0 will result in your particles starting from a different place. Changing g will result in particles falling either faster (higher g) or slower (lower g). An incorrect application of equations 2-5 will result in an incorrect starting velocity, which could result in particles going the wrong speed or direction. Miscalculating Δt may lead to particles which seem to jump around, or which do not move fast enough or at all.

4. A reflection function,

```
bool Reflect();
```

that will simulate the particle reflecting off the ground plane when it collides with it. The reflection function will be called externally after the update function is called. This function returns `true` if the particle is reflected, and `false` otherwise. The particle gets reflected only if its z-coordinate is negative. When reflected, the particle's state is modified as follows:

(a) The time of creation `t0` is reset to the current time;

(b) The particle's current z-coordinate will be multiplied by -1 and the current location `p` will be saved to its initial location `p0`;

(c) The z-coordinate of the particle's initial velocity `v_0` will be multiplied by 0.25.

This procedure simulates the particle bouncing off the ground plane. Part c) simulates the loss of energy from the inelastic collision of the particle with the ground.

5. A reset function

```
bool CheckAndReset();
```

which will reset the particle after reflection if the magnitude of the z-component of the initial velocity of the particle is less than 0.05. Note that this function will only be called externally immediately after the `Reflect()` function if the `Reflect()` function returns true. Once reset, the particle will be assigned new values as in the constructor. The return value of the function is true iff the particle was reset.

3 Visualization

To see your particle fountain in action, you will need to write some functions which animate the particles by first converting the 3D position of the particle into a 2D location in an image, and then drawing a feature at the given 2D location.

1. The function

```
void TransformPoint(const Vector3f& p, Vector3f* p_img);
```

takes a constant reference to a 3D point, `p`, and then writes the equivalent 2D point to the vector `p_img`. The equations for computing the image coordinates of a given particle are:

$$p_x^{img} = \gamma p_x + 300 \tag{8}$$

$$p_y^{img} = 400 - \gamma(\cos(\rho)p_z - \sin(\rho)p_y) \tag{9}$$

Here, γ is related to the focal length of the camera. ρ is related to angle of the camera. 300 and 400 are related to the camera's position. Use $\gamma = 60$, and $\rho = \frac{\pi}{9}$.

Keep in mind that `p_img` is a `Vector3f`, and that these equations only specify the x and y components. The z component is irrelevant for drawing, but be sure not to depend on its value for some reason later on, since it may be initialized with an arbitrary value!

2. In order to visualize the particles, you need to implement

```
void DrawParticles(const std::vector<Particle>& particles,
                   cimg_library::CImg<unsigned char>* img_ptr);
```

This function takes in a constant reference to a vector of particles, and for every particle it computes the transformed location (image location) and then calls the CImg function `draw_circle` at that location with a color and size of your choice. The pointer to a CImg, `img_ptr`, should store the results, ie the newly drawn circles.

4 Grading

Note: You must use the get and set methods from the vector class. Any other method for modifying particle locations will not be allowed.

4.1 Grading

Submissions will be graded on correctness of each of the parts, as well as the efficiency of the implemented code. Correctness will contribute to 70 points, and efficiency will contribute to 30 points. Code efficiency will be evaluated in terms of the following criteria:

1. Minimizing expensive memory copies, in particular the vector of particles.
2. Mathematical factorization to minimize the mathematical operators used for the simulation update.
3. Factorizing out shared computation for simulation updates, as well as visualization.
4. Total run time compared to the reference implementation.

4.2 Bonus Grading

You may earn up to a total of 10 bonus points for further optimization of your code, and for implementing an additional visualization feature:

1. Up to 8 bonus points will be awarded for submissions which are able to outperform the reference solution in terms of number of particles processed per second. You can use the reference executable `reference_simulation` in the handout to compare against your solution.

2. Two bonus points will also be awarded for drawing shadows under the particles. For determining the shadow positions, assume the light source is located at $s = [0, 0, \infty]$.

   You will need two functions to do this, the signatures for these functions are:

```
void PointShadow(const Vector3& point, Vector3* p_img);
```

and

```
void DrawShadows(const std::vector<Particle>& particles,
                 cimg_library::CImg<unsigned char>* img_ptr);
```

`PointShadow` determines the image location of a shadow for a given particle, and `DrawShadows` draws the shadows of the provided particles on the given `CImg` passed in as a pointer.

5 What To Turn In

Using the provided submission script `submit.sh`, the files `assignment9.h` and `assignment9.cpp` will be checked to see if they compile, and if successful, will be added to an archive, `assignment9.tar.gz`. You must upload this archive to Moodle.

You can compile your code with the provided main file using the following command:

```
1 clang++ -std=c++11 -O3 simulation.cpp assignment9.cpp -lX11 -lpthread -o
  ↪ run_simulation
```