6.837 Intro to Computer Graphics
Assignment 9: Particle Systems

In this assignment you will play around with Particle Systems. In the new code base, we provide the `GLCanvas` and `System` classes that manage the user interface and a dynamically-resizeable set of particles (`Particle` and `ParticleSet`). The `System::Update` method is called at a regular time interval (provided on the command line as the option `-refresh <seconds>`). The `Update` function advances the simulation forward in time (by the amount specified with the command line option `-dt <seconds>`).

Once the particle system has been launched you can interact with it through a simple gui. Use the mouse to rotate or zoom the scene. Hit 'p' to pause the simulation, and 'p' again to unpause it. When paused you can hit 's' to perform a single step of integration. You can also restart the simulation by hitting 'r'. To restart the system, all existing particles are deleted and any state stored within the system (including all random number generators) should be reset. We provide a simple random number generator class (see random.h) which generates a reproducible stream of pseudo-random numbers. By using the `Random` class instead of `drand48()` you can use multiple identical random number streams to directly compare various particle system options (the velocity or position randomness, the integration scheme, etc.)

You can also specify various rendering options at the command line. By default each particle is drawn as a large dot of the particle's current color. The `-integrator_color` option colors the particles based on integration scheme. The `-draw_vectors <acceleration_scale>` option draws the velocity vector in the same color as the particle, and a scaled version of the acceleration vector in white. This should be helpful when debugging your force fields and your integration scheme. Finally the `-motion_blur` option draws a thin line between the particle's current position and its last position.

Tasks

- Implement a base class `ForceField` and derive from this class the subclasses: `GravityForceField`, `ConstantForceField`, `RadialForceField` and `VerticalForceField`. The primary method for a force field is:

  ```
  virtual Vec3f ForceField::getAcceleration(const Vec3f &position, float mass, float t) const = 0;
  ```

  Remember from physics that $f = ma$, so if your field is specified with forces, you'll need to divide by the mass of the particle to get the acceleration. On the other hand, if your field is specified with acceleration ("local" earth gravity is a field of constant acceleration) you don't want to divide by the mass.

  Acceleration can vary based on position. To test your integrators you'll implement a `RadialForceField` that always pulls particles towards the origin. The strength of the force is proportional to the distance from the origin. Also implement a `VerticalForceField` that pulls particles towards the y = 0 plane (again the strength of the force is proportional to the distance from the plane). The constructors for the subclasses look like this:

  ```
  GravityForceField(Vec3f gravity);
  ConstantForceField(Vec3f force);
  RadialForceField(float magnitude);
  VerticalForceField(float magnitude);
  ```

  The force field acceleration can also vary based on time. For extra credit you can implement a time-varying field to imitate wind or other phenomena. Perlin Noise (from assignment 6) can be used to make nice random looking force fields (for extra credit).

- Implement a base class `Integrator` and derive from it the `EulerIntegrator` and `MidpointIntegrator` subclasses. (Implementing Trapezoid or Runge Kutta integration is worth extra credit). The prototype for the main method of the integrator class looks like this:

  ```
  virtual void Update(Particle *particle, ForceField *forcefield, float t, float dt) = 0;
  ```

  This method advances (with stepsize `dt`) the particle's position & velocity through the force field. If it's not a time-varying forcefield, `t` (the current time) will be ignored. The Integrator::Update method should also adjust the particle's age (`call Particle::increaseAge()`). Remember from class that the Euler method updates the particle as follows:
  $p_{n+1} = p_n + v_n * dt$
  $v_{n+1} = v_n + a(p_n, t) * dt$

  And the Midpoint method takes a half Euler step and uses that velocity to update the particle:
  $p_m = p_n + v_n * dt/2$
  $v_m = v_n + a(p_n, t) * dt/2$

  $p_{n+1} = p_n + v_m * dt$
  $v_{n+1} = v_n + a(p_m, t+dt/2) * dt$

  a(p,t) is the acceleration at point p, at time t. Each integrator also has a method `Vec3f getColor()` which returns a color for visualization when comparing different integration schemes.

- The last thing to do is is create the particles. Implement the base class `Generator`, with subclasses `HoseGenerator` and `RingGenerator`. The code we provide expects a number of methods for each generator (some may be virtual, some may be pure virtual depending on your implementation):

```
// initialization
void Generator::SetColors(Vec3f color, Vec3f dead_color, float color_randomness);
void Generator::SetLifespan(float lifespan, float lifespan_randomness, int desired_num_particles);
void Generator::SetMass(float mass, float mass_randomness);

// on each timestep, create some particles
int Generator::numNewParticles(float current_time, float dt) const;
Particle* Generator::Generate(float current_time, int i);

// for the gui
void Generator::Paint() const;
void Generator::Restart();
```

The first 3 methods set some common particle parameters. Each of these has an optional randomness component. Not all of the possibilities for randomness are exercised in the examples below.

The next two methods are helper functions for the `System::Update` function. On each timestep the generator is asked how many new particles should be created. To maintain a roughly constant number of particles throughout the simulation (desired_num_particles), *dt * desired_num_particles / lifespan* particles should be created on each timestep. If the number of particles varies throughout the simulation (e.g., the fire ring), you'll need to do something else. The `Generate` method takes 2 arguments (the current time and an integer between 0 and n-1, where n is the number of new particles for this timestep).

Finally, the `Paint()` method draws any geometry related to the generator. In the fire example below a polygon is drawn to represent the ground plane. The `Restart` method reseeds the random number generator for the class to ensure that the particle simulation is exactly reproducible.

The generator subclass constructors look like this:

```
HoseGenerator(Vec3f position, float position_randomness, Vec3f velocity, float velocity_randomness);
RingGenerator(float position_randomness, Vec3f velocity, float velocity_randomness);
```

The hose generates particles from a point in space (`position`) in a particular direction (`velocity`). Both position & velocity can be tweaked with randomness. The ring generator has similar arguments, except the particles are placed (with randomness) on an expanding ring from the origin. The radius of the ring depends on time. To keep the density of the ring constant, the number of particles created per time step increases. Caution: we provide you with a dynamically-resizeable array for particles, but don't get carried away generating particles -- your machine does have memory limits! In the fire ring example below the whole scene is shifted down by -4 units in y.

- NOTE: Particle systems (like solid textures) are an art form. Your results for test cases 4-7 should match very closely to the sample images below. But the other test cases depend on exactly how you incorporate randomness, or on how you tweak your fire density, etc. There is no single correct answer for these test cases, just make sure you understand how your implementation corresponds to your results, and have fun! Feel free to modify the base implementation we've provided or make new test cases. Just submit everything and document your changes/additions in your `README.txt` file.

References

- SIGGRAPH education materials on Particle Systems (includes Star Trek video)
- SIGGRAPH course notes on Physically Based Modeling (from Pixar)

Ideas for Extra Credit

- Implement Trapezoid or Runge Kutta integration.
- Design and implement a test scene for a fun, new particle generator.
- Implement time-varying force fields and/or use Perlin Noise to generate force fields.
- Implement particle collisions with external objects, like a sphere or plane (no particle-particle collisions).
- Add damping forces.
- Add particle-particle interactions (e.g., n-body problem, Lennard Jones forces, etc.)
- Burning planets, water fountains, fireworks, etc., etc., etc.

New Code Base for this Assignment

You may modify these files as you wish, but you should only need to create the new classes described above.

- vectors.h
- matrix.h
- matrix.C
- particle.h
- particle.C
- particleSet.h
- random.h
- system.h
- system.C
- parser.h
- parser.C
- glCanvas.h

- glCanvas.C
- main.C

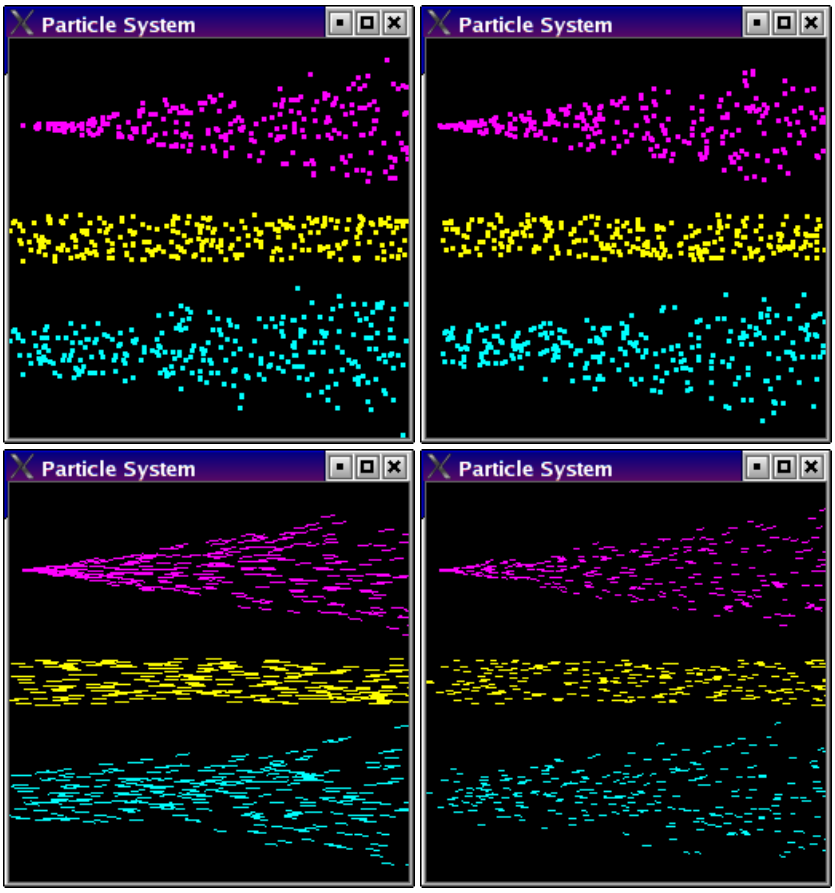
- Makefile (for Athena Linux) **FIXED 11/18**

Input Files

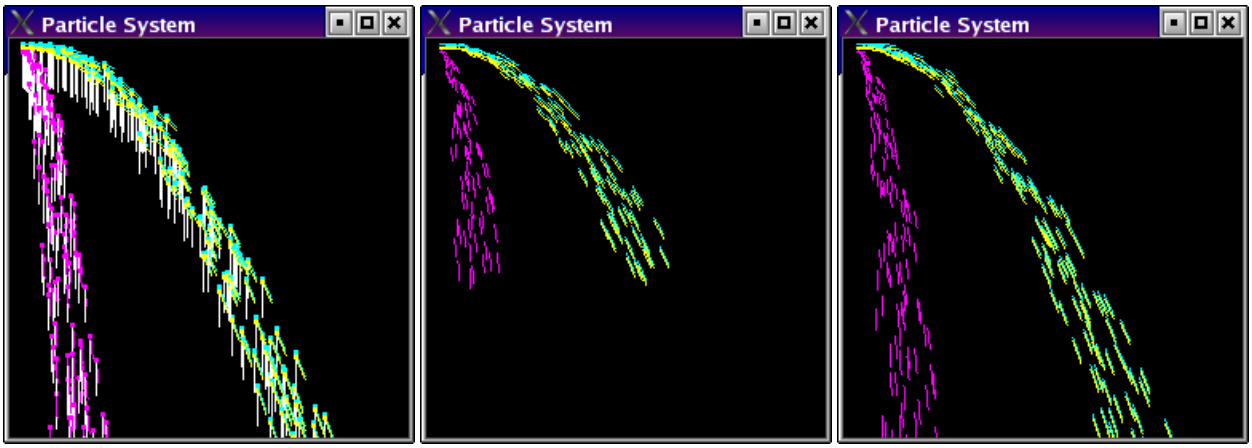
- system9_01_hose.txt
- system9_02_hose_gravity.txt
- system9_03_hose_force.txt
- system9_04_circle_euler.txt
- system9_05_circle_midpoint.txt
- system9_06_circle_rungekutta.txt
- system9_07_wave.txt
- system9_08_fire.txt
- system9_09_wind.txt

Sample Results

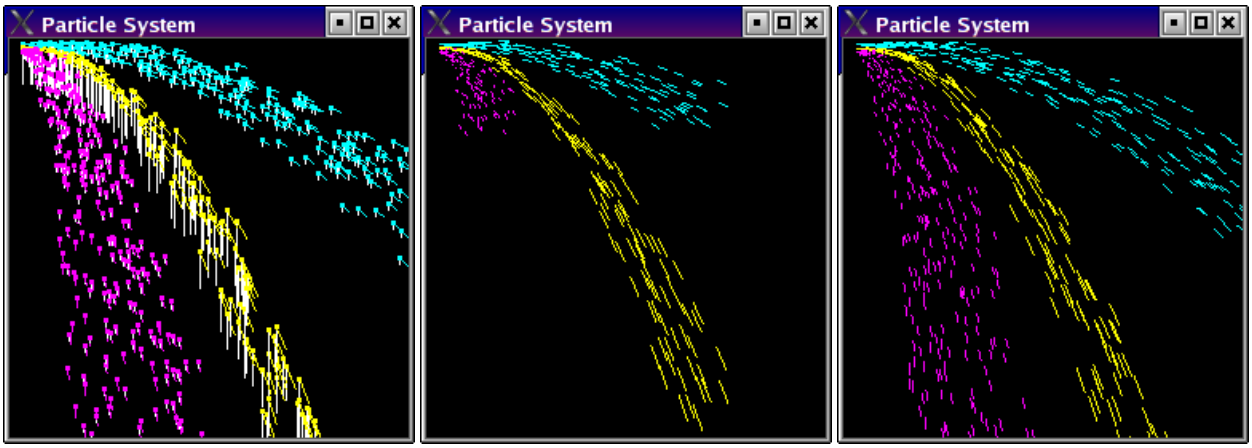
```
particle_system -input system9_01_hose.txt -refresh 0.1  -dt 0.1
particle_system -input system9_01_hose.txt -refresh 0.05 -dt 0.05
particle_system -input system9_01_hose.txt -refresh 0.1  -dt 0.1  -motion_blur
particle_system -input system9_01_hose.txt -refresh 0.05 -dt 0.05 -motion_blur
```



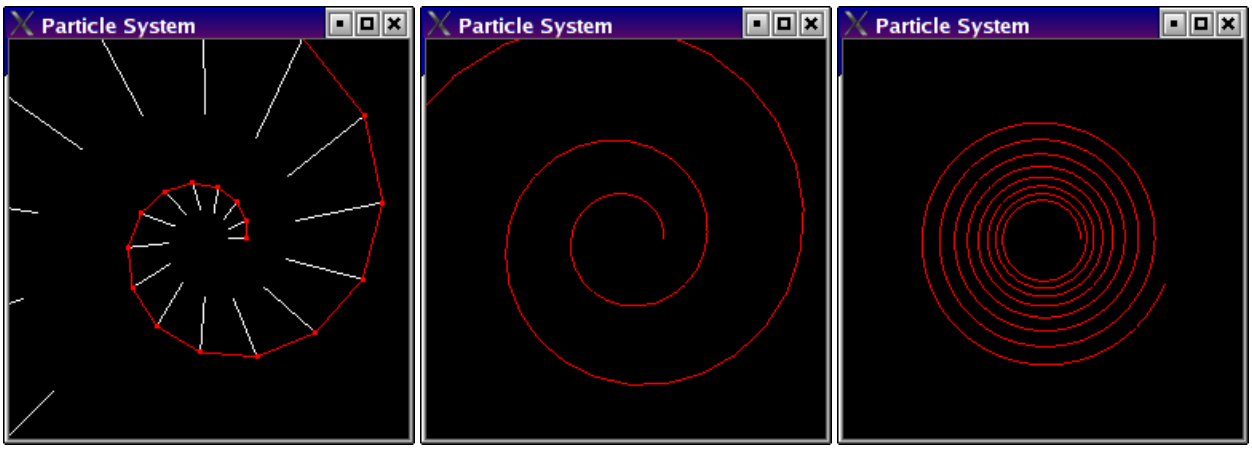

```
particle_system -input system9_02_hose_gravity.txt -refresh 0.05 -dt 0.05 -draw_vectors 0.1
particle_system -input system9_02_hose_gravity.txt -refresh 0.05 -dt 0.05 -motion_blur
```



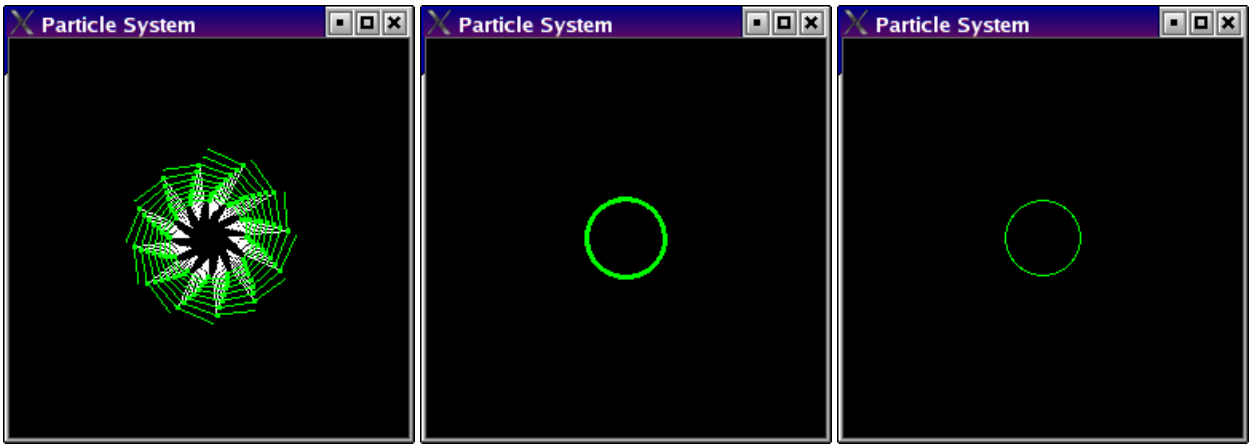

```
particle_system -input system9_03_hose_force.txt -refresh 0.05 -dt 0.05 -draw_vectors 0.1
particle_system -input system9_03_hose_force.txt -refresh 0.05 -dt 0.05 -motion_blur
```




```
particle_system -input system9_04_circle_euler.txt -refresh 0.1  -dt 0.1  -integrator_color -draw_vectors 0.02
particle_system -input system9_04_circle_euler.txt -refresh 0.05 -dt 0.05 -integrator_color -motion_blur
particle_system -input system9_04_circle_euler.txt -refresh 0.01 -dt 0.01 -integrator_color -motion_blur
```

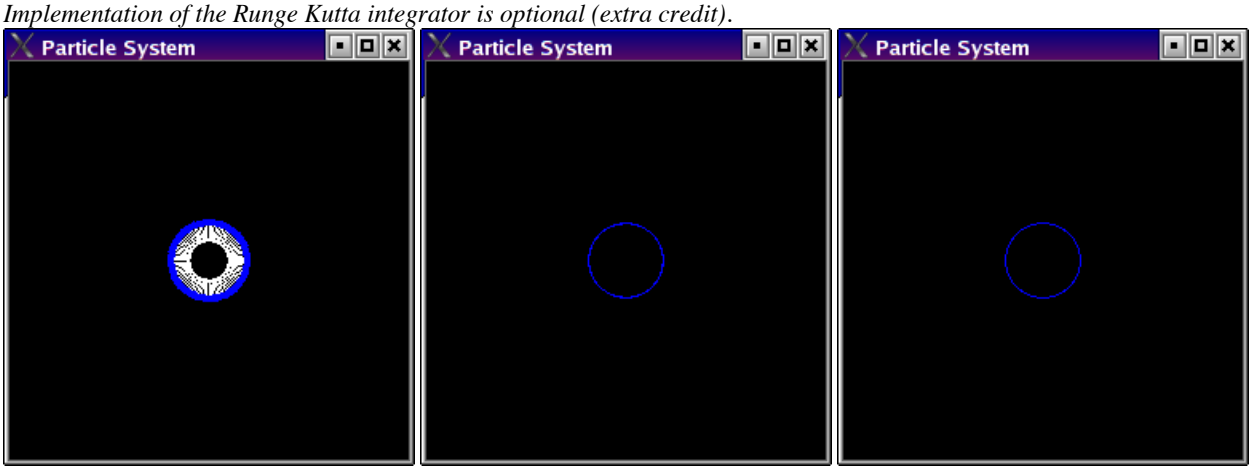



```
particle_system -input system9_05_circle_midpoint.txt -refresh 0.1  -dt 0.1  -integrator_color -draw_vectors 0.02
particle_system -input system9_05_circle_midpoint.txt -refresh 0.05 -dt 0.05 -integrator_color -motion_blur
particle_system -input system9_05_circle_midpoint.txt -refresh 0.01 -dt 0.01 -integrator_color -motion_blur
```



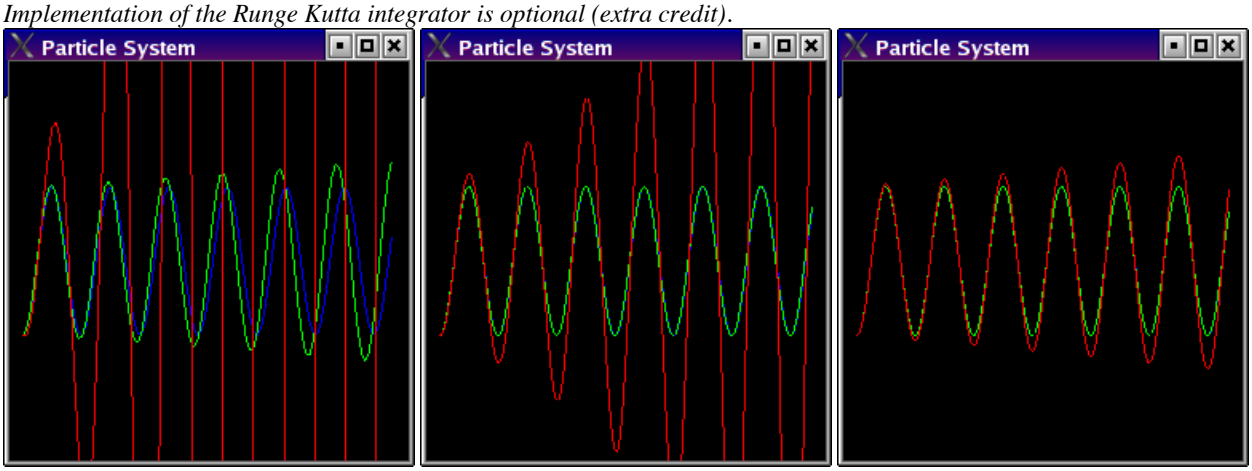

```
particle_system -input system9_06_circle_rungekutta.txt -refresh 0.1  -dt 0.1  -integrator_color -draw_vectors 0.02
particle_system -input system9_06_circle_rungekutta.txt -refresh 0.05 -dt 0.05 -integrator_color -motion_blur
particle_system -input system9_06_circle_rungekutta.txt -refresh 0.01 -dt 0.01 -integrator_color -motion_blur
```

Implementation of the Runge Kutta integrator is optional (extra credit).

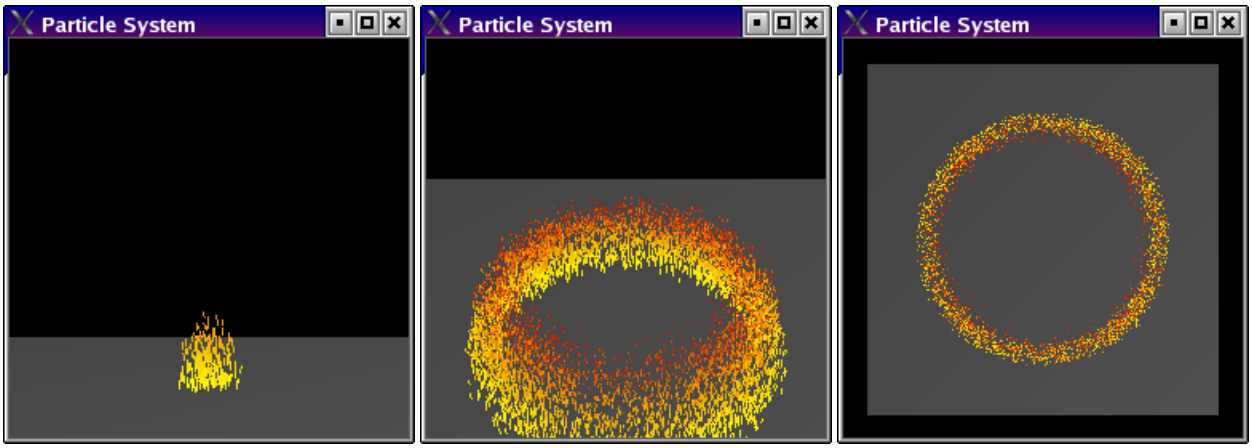



```
particle_system -input system9_07_wave.txt -refresh 0.01 -dt 0.2  -integrator_color -motion_blur
particle_system -input system9_07_wave.txt -refresh 0.01 -dt 0.05 -integrator_color -motion_blur
particle_system -input system9_07_wave.txt -refresh 0.01 -dt 0.01 -integrator_color -motion_blur
```

Implementation of the Runge Kutta integrator is optional (extra credit).

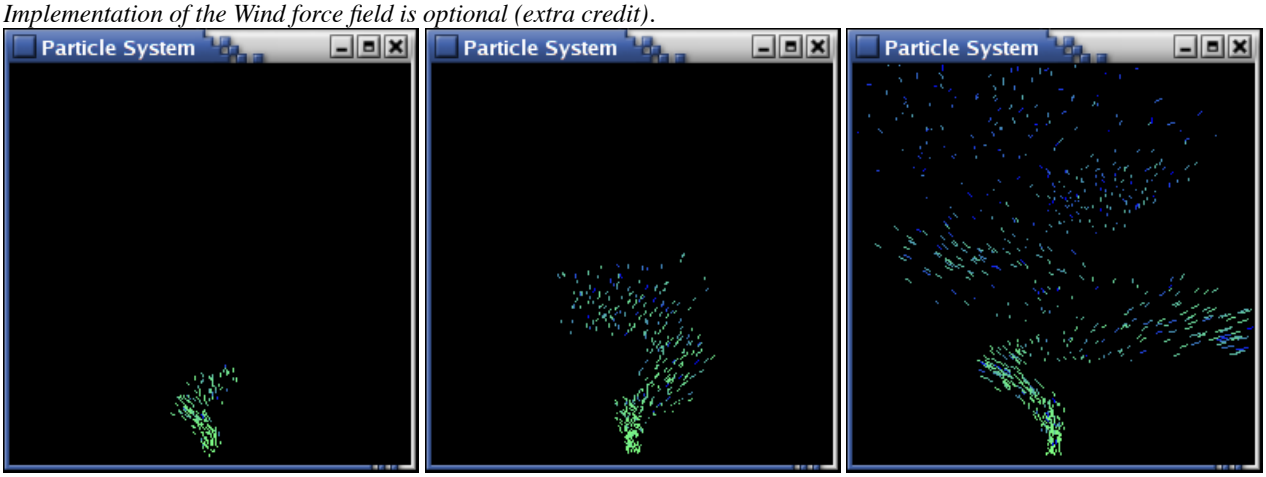



```
particle_system -input system9_08_fire.txt -refresh 0.05 -dt 0.05  -motion_blur
```




```
particle_system -input system9_09_wind.txt -motion_blur -dt 0.05 -refresh 0.05
```

Implementation of the Wind force field is optional (extra credit).



See the main Assignments Page for submission information.
