c3particles:
Modeling a Particle System in C++

Rosalie Kletzander

Practical Course "Advanced Software Development with Modern C++"
Summer Term 2018
Institute for Computer Science
Ludwig-Maximilians-Universität München

1 Introduction

Particle systems are used in many different areas: most prominently in the entertainment industry in games and movies and for simulations and visualizations scientific research. No matter the area of application, the basic rules governing these systems are the same: the laws of physics. c3particles (cpp particles) implements a model of a particle system in C++ that separates the physical concepts and laws from the underlying graphics library. This enables a mathematical formulation of the forces influencing the particles.

2 A Short Recap of Mechanical Physics

In order to model a particle system that simulates natural phenomena, it is necessary to first understand the basic rules of motion.

Newton's First Law of Motion states:

> "Every object in a state of uniform motion tends to remain in that state of motion unless an external force is applied to it." [1]

This means that an object will not move unless it is accelerated by a force, which brings us to Newton's Second Law of Motion:

> "The relationship between an object's mass m, its acceleration a, and the applied force F is $F = m * a$." [1]

With this information it is possible to calculate the acceleration of an object by dividing the applied force by the object's mass. The next step is deriving the velocity and location of the object per time step by integration [5].

The velocity of an object can be calculated by integrating the acceleration over time t.

$$\overrightarrow{v}(t) = \int (\overrightarrow{a})\mathrm{d}t = \overrightarrow{a} * t + C_v \tag{1}$$

C is the integration constant, in this case it is equal to the velocity of t-1 for discrete time steps. Integrating the velocity over t yields the location.

$$\vec{s}(t) = \int (\vec{v})\mathrm{d}t = \int (\vec{a} * t + C_v)\mathrm{d}t = \frac{\vec{a} * t^2}{2} + C_v + C_s \tag{2}$$

Analogous to C_v, C_s is equal to the location at t-1.

With these formulas, it is possible to calculate an object's change in location over time.

A further basic law of physical mechanics is the superposition principle, which states that applying the sum of two forces to an object has the same effect as if they were applied individually. Or, more formally:

$$f(a * x) = a * f(x) \tag{3}$$

$$f(x + y) = f(x) + f(y) \tag{4}$$

[3]

These laws serve as the foundations of the physical model of the particle system.

3 Modeling the Particle System

The basic concepts needed in order to model the particle system are already given by the physics described in the previous section. In fact, a particle system is a fairly simple construct. It contains objects, its "particles", which behave according to newtonian physics, i.e. "newtonian objects". They have a location, velocity, acceleration and mass. As Newton's First Law states, a force needs to be exerted in order for a particle to move. Since movement is merely a change of location over time, time also needs to be considered.

3.1 Concepts

In fact, these two concepts, "newtonian object" and "force" are sufficient for modeling a particle system. The basic .. expressions ..:

Particle ≪ Force applies a force to a particle using the formulas 1 and 2

Force calc_force(Particle, Particle, Function) calculates a force between two particles using the supplied function, when given the same particle twice, it returns the additive identity

Force gravity(Particle, Particle, List params) a specialization of calc_force for calculating the gravitational force between two particles (is not actually a concept, but shows what calc_force is capable of)

Force accumulate(Particle, ParticleContainer, Function) calc_force for p with each other p in the container and reduce (addition) to one force

Force accumulate(Particle, ParticleContainer, List params, Function) use a pre-defined function to calculate the forces, pass initializer list for parameters
paragraphForce accumulate(List forces) sum up a set of forces

3.2 Time

As the particle system is made to be rendered on a screen in (semi) real time, the frame per second count gives discrete time steps that are used for integrating over acceleration. For each frame, one time step is calculated, which simplifies the result of formula 2:

$$\frac{\overrightarrow{a} * t^2}{2} + C_v + C_s = \frac{\overrightarrow{a}}{2} + C_v + C_s \tag{5}$$

4 Implementation

c3particles contains several separate modules (Figure 1. The Particle System module contains the physical model and uses input given by the user to select forces. It updates the particles that are then read by the Particle Renderer, which uses the location to calculate the vertices and faces that need to be drawn. It passes the vertex buffers of all the particles to the OpenGL Rendering Pipeline, which processes them accordingly. It then writes the framebuffer to the screen and triggers a new calculation.

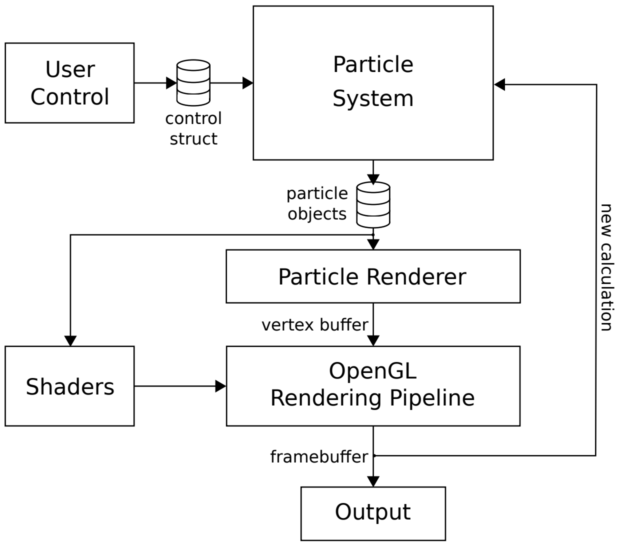

Fig. 1: c3particles system diagram

4.1 Particle System

The Particle System module contains the physical model of the particle system and the particle objects (Figure 2) The physical model includes algorithms for calculating the forces on the particles, and the particles themselves. For each frame, the old values of the particles are read and used to update to the new values. The functions used to update the values are taken directly from formulas 1 and 5.

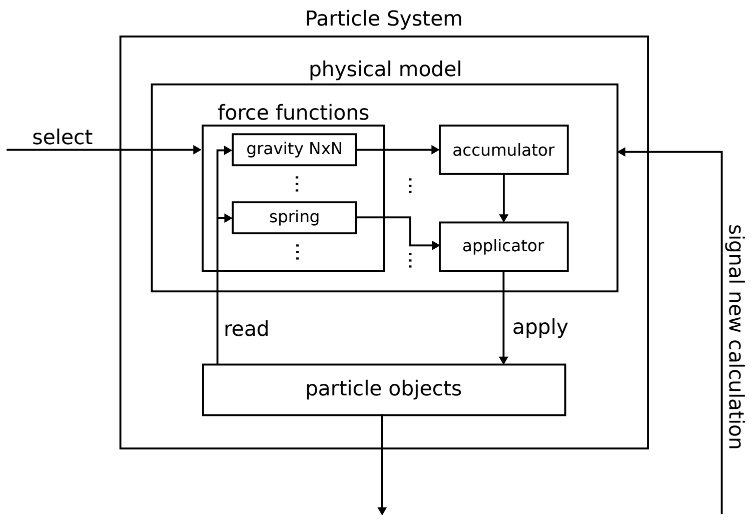

Fig. 2: detailed diagram of the particle system

Algorithms

- concepts and expressions come to fruition here
- differentiation between ”external” forces and inter-particle forces
- apply force ” << ”
- calc_force
- accumulate
- specializations of calc_force, e.g. gravity

The concepts and expressions defined above are incorporated into the system's algorithms. The core algorithm is calc_force (Listing 4), which takes a lamda defining the calculation of a force between two particles.

This enables very straightforward definition of any force between two particles. It can be used to describe forces on the fly, or to create pre-defined force functions, such as gravitational force (Listing 5).

```
void ParticleSystem::update()
{
  // deltaT will always be 1.0 because calculation is based on frames
  for (Particle &p : _particles)
    {
      //v(t) = a*t + v(t-1)
      p.velocity = p.acceleration * 1.0f + p.velocity; //deltaT = 1.0

      //s(t) = (a*t^2)/2 + v(t) + s(t-1)
      p.location = (p.acceleration * 1.0f) / 2.0f
                 + p.velocity + p.location;

      //acceleration is not accumulative, but recalculated at each time
          step
      p.acceleration = {0, 0, 0};
    }
}
```

Fig. 3: ParticleSystem::update()

For the calculation of the forces it is helpful to split them logically into "inter-particle forces" and "external forces". The former refers to the forces that exist between all pairs of particles (e.g. gravitational forces) and the latter refers to forces that are applied to each particle independently of all others (e.g. wind).

c3p::accumulate (Listing 6, line 6) calculates the forces between each pair of particles (i.e. inter-particle forces) and reduces them. The result is a force that can then be applied normally.

Particle Container The particles need to be stored in some data structure. At the moment, this is a std::vector which provides all the needed operations.

4.2 User Control Window

The user controls are implemented with GTK+[4]. The control window runs in a different thread that fills a C struct with the values set by the user. These values are then read by the system in order to calculate the desired forces.

4.3 Particle Renderer

The particle renderer iterates over the particles in the particle system and fills the vertex buffers for each one. How the vertex buffers are filled depends on the function called. At the moment, it is possible to render the particles as pixels with no depth perception and uniform size (*c3p::ParticleRenderer::renderPoints* or as cubes (*c3p::ParticleRenderer::renderCubes*). The size of the cubes depends on the size of the particle (which, at the moment is dependent on its mass). The vertex buffers are then passed to the OpenGL Rendering Pipeline.

```
// calculates a force between two particles using the function ff
// when given the same particle twice, it returns the identity
Force calc_force(
    const Particle &p1,
    const Particle &p2,
    std::function<Force(const Particle &p1, const Particle &p2)> ff)
{
  if (&p1 == &p2)
    { return glm::vec3(0, 0, 0); }
  return ff(p1, p2);
}
```

Fig. 4: calc_force function

4.4 Shaders

4.5 Graphics Engine

The graphics engine is only secondary for this project. OpenGL[2] was chosen because of its prominence, however, with an appropriate particle renderer, any one could be used.

5 Assets of c3particles

- mathematical functions can be easily applied
- clean separation allows for easy access to velocity etc (for reflection, reverse)

6 Complexity and Possible Optimizations

- complexity of naive implementation of inter-particle forces is $O(n^2)$
- complexity of application of external forces and also updating is O(n)
- force matrix for inter-particle forces
- parallelize particle calculation (works very well, because applyforce only updates acceleration, so location and velocity are unchanged until all forces have been calculated)
- well suited for offloading to the GPU
- binary space partitioning for forces that are inversely proportional to distance
- no pruning because particles that are outside of the viewport still exert forces

References

1. Newton's Three Laws of Motion. `http://www.pas.rochester.edu/~blackman/ast104/newton3laws16.html`. Accessed: 2018-08-5.

```cpp
// uses calc_force to calculate gravitational force between two particles
Force gravity(const Particle &p1,
          const Particle &p2,
             std::initializer_list<float> params)
{
  Force result = calc_force(p1,
                     p2,
                      [p1, p2](const Particle &, const Particle &) {
        glm::vec3 direction = p2.location - p1.location;
        float gforce = (p1.mass * p2.mass) / pow(glm::length(direction),
            2);
        glm::normalize(direction);

        return (gforce * direction);
      });
  // multiply by gravity constant
  for (auto c : params)
    {
      result *= c;
    }
  return result;
}
```

Fig. 5: gravity function

2. OpenGL: The Industry's Foundation for High Performance Graphics. https://www.opengl.org/. Accessed: 2018-08-5.
3. Superposition principle wikipedia. https://en.wikipedia.org/wiki/Superposition_principle. Accessed: 2018-08-5.
4. The GTK+ Project. https://www.gtk.org/. Accessed: 2018-08-5.
5. Zusammenhang Ruck, Beschleunigung, Geschwindigkeit und Weg. https://www.johannes-strommer.com/rechner/basics-mathe-mechanik/ruck-beschleunigung-geschwindigkeit-weg. Accessed: 2018-08-5.

```cpp
//iterate over all particles in the particle system
std::for_each(ps.begin(), ps.end(), [&ps](c3p::Particle& p)
{

   //gravitational forces between particles
   p << c3p::accumulate(p,
                 ps.particles(),
                 {ps.g_constant()},
                 c3p::gravity);

   // spring force from virtual particle at (0,0,0) to each particle
   p << spring(p, Particle(0,0,0), {spring_constant, spring_length);

   // simple user-defined attraction force pulling towards (0,0,0)
   p << calc_force(p, Particle(), [p](const Particle &, const Particle &)
   {
      glm::vec3 direction = glm::normalize(glm::vec3(0,0,0) -
          p.location());
      return direction * 0.1;
    });
}

//calculate new location from acceleration
ps.update();
```

Fig. 6: example of how forces are applied