QUADOTS TUTORIAL

Souren Papazian (ssp2155)
Surashree Kulkarni (ssk2197)
Srilakshmi Chintala (sc3772)

INTRODUCTION

Quadots is a C++ library of for making 2D simulations on a set of points. More specifically, the movement of these points can be manipulated based on some logic the user defines.

The library provides various functions that help construct behaviors, such as the nearest neighbors of a point or the average direction of all points.

This document will teach the user to use the Quadots library. By the end of this tutorial, the user should know how to:

- Create points in 2D space.
- Create functions to manipulate their behavior.
- Generate complex (pretty!) patterns in space.

Difficulty level: Easy

1. Create a Rule

Quadots lets a user have two types of points: 1. Point 2. Dot
The distinction between the two is:

Point:
- *is static.*
- *has only position coordinates (x,y) associated with it.*
- *can be manipulated only through its position.*

Dot:
- *can move in space.*
- *has position (x,y), velocity, and direction associated with it.*
- *can be moved, accelerated or rotated.*

The first thing to do is decide on a rule for the points to control their behavior. For this, you have to write a *Rule*. A rule is a function in the Simulation library that takes as parameters a pointer to a Point/Dot and a Control. We will discuss Control later.

Syntax:

```
Simulation<Point>::rule rule_name = &rule_function_name;
```

As an example, we will try to get a Point to rotate in a circle.

Example:

```
Simulation<Point>::rule rot = &rotate;
```

Our rule is the `rot` object. We will write the `rotate` function towards the end.

2. Create a Behavior

Once you have created your rule(s), you can create Behavior, which is simply the set of rules the points should follow.

Syntax:

```
Simulation<Point>::behavior behavior_name = {rule_name};
```

Example:

```
Simulation<Point>::behavior pattern = {rot};
```

3. Initialise Simulation

To create and points to a simulation, you need to initialize a simulation first.

Syntax:

```
Simulation<Point> *sim_name = new Simulation<Point>(width, height);
```

The Simulation constructor takes as parameters the size of your simulation in terms of its width and height. This is basically a bound on the location of your points.

Example:

```
Simulation<Point> *s = new Simulation<Point>(800, 800);
```

will create a Simulation of size 800x800.

4. Get a Behavior index

Once we have initialized our simulation, we are free to access all the functions in our simulation object. The Quadots library creates an index value for each behavior. This is just an integer value that we need later to assign behavior to our Point(s).

The index value is returned by the `CreateBehavior()` function in the Simulation library, that takes in as parameter the Behavior we generated earlier.

Syntax:

```
int b_index = sim_name->CreateBehavior(behavior_name);
```

Example:

```
int b = s->CreateBehavior(pattern);
```

5. Create a Point/Dot

We are now ready to create a Point/Dot.

The `CreateElement()` is a `void` function takes in a Point/Dot type. The Point constructor takes x and y coordinates of the point, while the Dot constructor takes x, y, velocity, and direction of the point. Both Point and Dot also take the Behavior Index. (Part 4.)

Syntax:

```
sim_name->CreateElement(Point(x, y, b_index));
```

```
sim_name->CreateElement(Dot(x, y, direction_angle, velocity,
b_index));
```

Example:

```
s->CreateElement(Point(100, 100, b));
```

This creates a Point with coordinates (100,100) and assigns behavior `b` to it.

6. Define a Renderer

The Renderer object creates a window for the points to be displayed.

Syntax:

```
Renderer<Point> renderer_name = Renderer<Point>(window_width,
window_height, steps_per_second);
```

Example:

```
Renderer<Point> twodee = Renderer<Point>(800, 800, 1);
```

7. Write our behavior function

The behavior function determines how the point is going to act at every frame.

Syntax:

```
void rule_function_name(Point::Point_p p, Control<Point>& c) {
      //do something
}

void rule_function_name(Dot::Dot_p p, Control<Point>& c) {
      //do something
}
```

Example:

```
void rotate(Point::Point_p p, Control<Point>& c) {
      p->set_ang(p->get_ang() + 2);
}
```

The Point/Dot class have functions to get/set their position, velocity, and direction.

8. Run the simulation

We can now run our simulation. To do this, we use the Run() function in the Simulation library. The function takes the number of frames for which the simulation should run, and the renderer object we created in 6. Passing in a 0 for the first argument makes the simulation run infinitely.

Syntax:

```
s->Run(run_steps, renderer_name);
```

Example:

```
s->Run(1000, twodee);
```

9. Putting it all together

```
#include <iostream>
#include "Point.h"
#include "Simulation.h"
#include <cstdlib>
#include <time.h>
using namespace std;

void random_move(Point::Point_p p, Control<Point>& c) {
     p->set_ang(p->get_ang() + 2);
}

int main()
{

    // Brains
    Simulation<Point>::rule move = &random_move;
    Simulation<Point>::behavior pattern = {move};

    // Initialize Simulation
    Simulation<Point> *s = new Simulation<Point>(800, 800);

    // Create behavior circle
    int b = s->CreateBehavior(pattern);

    // Create a Point
    s->CreateElement(Point(100, 300, b));

    // Run simulation
    Renderer<Point> twodee = Renderer<Point>(800, 800, 1);
    s->Run(1000, twodee);

    delete s;
    return 0;
}
```

A MORE COMPLEX PATTERN: BOIDS

Boids is an artificial life program, developed by Craig Reynolds in 1986, which simulates the flocking behaviour of birds. the complexity of Boids arises from the interaction of individual agents (the boids, in this case) adhering to a set of simple rules. The rules applied in the simplest Boids world are as follows:

- *separation: steer to avoid crowding local flockmates*
- *alignment: steer towards the average heading of local flockmates*
- *cohesion: steer to move toward the average position (center of mass) of local flockmates*

More complex rules can be added, such as obstacle avoidance and goal seeking.

The code below generates a basic version of boids.

```
#include <stdlib.h>
#include <time.h>
#include <iostream>
#include <cmath>
#include "Dot.h"
#include "Simulation.h"
using namespace std;

// Helper functions

/*
    Returns a random float in the range [min, max)
*/
float random_pos(int min, int max) {
    return(float) (rand() % max + min);
}

/*
    Steer a point towards a goal direction
*/
float delta_dir(float curr_dir, float goal_dir, float steps) {
    float delta1 = goal_dir - curr_dir;
    float delta2 = (goal_dir > curr_dir) ? (delta1 - 360) : (delta1 +
360);
    if(abs(delta1) < abs(delta2))
        return delta1/steps;
    else
        return delta2/steps;
}

// Boid rules

/*
    Steer to avoid crowding local flock mates
```

```
*/
void separation(Dot::Dot_p p, Control<Dot>& c) {
    vector<Dot::Dot_p> neighbors = c.qneighbors(p, 50);
    float steps = 15;

    if(neighbors.size()) {
        float avg_x = c.avg_x(neighbors);
        float avg_y = c.avg_y(neighbors);
        float goal_dir = c.dir_towards(p, avg_x, avg_y) - 180;
        float step_ang = delta_dir(p->get_ang(), goal_dir, steps);
        p->add_ang(step_ang);
    }
}

/*
    Steer towards the average heading of local flock mates
*/
void alignment(Dot::Dot_p p, Control<Dot>& c) {
    vector<Dot::Dot_p> neighbors = c.qneighbors(p, 100);
    float steps = 10;

    if(neighbors.size()) {
        float avg_dir = c.avg_dir(neighbors);
        float step_ang = delta_dir(p->get_ang(), avg_dir, steps);
        p->add_ang(step_ang);
    }
}

/*
    Steer a boid towards average position of flock mates
*/
void cohesion(Dot::Dot_p p, Control<Dot>& c) {
    vector<Dot::Dot_p> neighbors = c.qneighbors(p, 150);
    float steps = 15;

    if(neighbors.size()) {
        float avg_x = c.avg_x(neighbors);
        float avg_y = c.avg_y(neighbors);
        float goal_dir = c.dir_towards(p, avg_x, avg_y);
        float step_ang = delta_dir(p->get_ang(), goal_dir, steps);
        p->add_ang(step_ang);
    }
}

// Loop Dots that go out of bounds

void bounds(Dot::Dot_p p, Control<Dot>& c) {
    if(p->get_x() <= 0)
        p->set_x(799);
    else if(p->get_x() >= 800)
        p->set_x(1);
    if(p->get_y() <= 0)
        p->set_y(799);
    else if(p->get_y() >= 800)
```

```
        p->set_y(1);
}

int main()
{
    // Brains
    Simulation<Dot>::rule r1 = &cohesion;
    Simulation<Dot>::rule r2 = &alignment;
    Simulation<Dot>::rule r3 = &separation;
    Simulation<Dot>::rule r4 = &bounds;

    Simulation<Dot>::behavior boid = {r1, r2, r3, r4};

    // Initialize Simulation
    Simulation<Dot> *s = new Simulation<Dot>(800, 800);

    // Create behavior circle
    int b1 = s->CreateBehavior(boid);

    // Define random seed
    srand(time(NULL));
    // Create two Points in the middle of the screen facing opposite
directions
    for(int i=0; i < 200; i++)
        s->CreateElement(Dot(random_pos(0,800), random_pos(0,800),
random_pos(0,360), 1, b1));

    // Initialize Renderer
    Renderer<Dot> twodee = Renderer<Dot>(800, 800, 200);

    // Run Simulation for 5000 steps
    s->Run(5000, twodee);

    delete s;
    return 0;
}
```