QUADOTS
Design Document

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

Feature set:

1. General design.

The Quadots library provides Point and Dot objects and helper functions to manipulate the behavior of these objects in 2D space. The library makes use of the Quad tree data structure on which the helper functions are built.

The user can define the behavior of these objects to produce interesting simulations such as:

- Boid's implementation (or other physical simulations)
- Collision detection in two dimensions
- Spatial Indexing
- Conway's Game of Life simulation program
- Image analysis
- State Estimation

A user on the more granular level can make use of the library to:

- Add elements to the simulator (default elements are Point and Dot). Points are static and have an x and y coordinate. Dots inherit Point but add velocity and direction.
- Update the state/position of objects by defining rule functions that will be applied (every step) to whichever elements they are assigned to.
- Run simulations for n steps either by printing out the step each state or passing in a renderer that will render the state each step. The library currently has a simple renderer that draws elements as small black squares.

2. Rules and Behaviors

The Rule type is defined by the simulation class.
It is basically a function pointer that takes a shared_ptr to an element and a reference to control. In the rule function the element is manipulated using it's get-er/set-er functions. The control object is used to retrieve information from the current state. This way the behavior defined can depend on the rest of the state.
This is the main tool the user will use to make simulations.

A behavior is a vector of rules. When the user passes a behavior to the simulation object, it will receive back an unsigned integer. This integer is the behavior index which is assigned to an element.

Every step, the simulation will iterate over all the elements in the current state and using their behavior indexes, retrieve what functions need to be applied to the element to manipulate it.

3. Quad tree

The Quadot library inherently makes use of a Quad tree structure on which the helper functions are built. Since quad trees are special structures, there are many advantage over just using a brute force methods to access the point objects based on location.

Quad tree is a tree data structure in which each internal node has exactly four children. Quad trees are most often used to partition a two-dimensional space by recursively subdividing it into four quadrants or regions. The regions may be square or rectangular, or may have arbitrary shapes.
Essential features of a Quad tree are:

- They decompose space into adaptable cells.
- Each cell (or bucket) has a maximum capacity. When maximum capacity is reached, the bucket splits.

The Quad tree structure currently provides functionality for:

- Inserting objects into the quad tree
- Deleting objects from the tree
- Getting the nearest neighbours of an object (logic for collision detection)

4. Abstracted classes.

Since everything is templated, the user is also free to define his/her own element type, render and additional control functions.
This way you can simulate a variety of applications using the library.

Interface design:

There are four main classes: Simulation, State, Control.
All of them are templated to element. This way you can run a simulation on any type of element as long as it has these three functions: get_x(), get_y(), update().
It is highly recommended to the user to inherit the Point class to create new element types.

Simulation<elem>

This is the class that actually runs the simulation.
A Quadot simulation is a essentially a set of elements being modified every step. The elements are stored in a State object. Every step of the simulation, an UpdateState function is called that creates a new State based on some logic defined by the user and deletes the old State.
At this point you can only have one type of element per simulation.

State<elem>

A state owns the container of the elements. For quadots this container is a quad tree. The reason we hold all the elements is because our library is for spatial simulations. So functions that are related to space such as getNeareastNeighbors are much more efficient with a quad tree.
State has an add function that is used to add new elements and a get_elements function that is used to retrieve elements.
All elements are stored using shared_ptrs, so there is no need to worry about who owns the actual element. The user, Simulation, Quad tree, State and Control create new pointers often, so a shared_ptr was the best thing to use.

Control<elem>

This is the class that contains all the useful functions for the user to get information about the current state. The user can use these functions to manipulate it's points in the rule functions (rules will be explained soon).
For example:

```
float avg_x() const;
float get_distance(const Elem_p a, const Elem_p b) const;
vector<Elem_p> qneighbors(Elem_p a, float radius);
```

where Elem_p is a shared pointer to an element.

Optimizations

Our main optimization was making the storage a quad tree. This way (according to our intense measurements), nearest neighbors performs much better than brute force.

Features for the convenience of users

The whole design of the library is made for the user to simulate points easily.
Everything is abstracted: the user just defines what behavior each element should have by declaring rule functions and then passes those elements and behaviors into the simulation. The simulation class will handle the rest.

Error handling resource management

Error handling is not done by throwing exceptions.
We simply take care of any way the user might break the library by controlling the logic with if statements and printing to cerr when needed.

All elements are stored by the user and are passed around using shared_ptrs, so their management is very easy and foolproof.
Large objects such as Simulation, State, Control and Quad tree are stored on the heap with regular pointers and are taken care of by the destructors.

Only temporary objects/data and pointers are stored on the stack.

Ideas for release 1.2

The Quadots library version 1.0 lets the user perform only basic manipulations on a Point/Dot's coordinates, velocity and angle. Version 1.2 should have the following features:

- More default element types that can be used.
- Many more methods in Control that get information based on the current state. Especially more that take advantage of the quad tree class.
- Concurrency- manipulate clusters of points in parallel to improve execution speed.
- Run simulations with several types of elements. Currently all class objects need to be initialized to one type.
- Store all states for user
- Add feature to remove elements from simulation/state/quad tree.
- Many optimizations for quad tree.

References

- http://en.wikipedia.org/wiki/Quadtree
- https://isocpp.org/std/library-design-guidelines