Sample Design Spec. : 3D Viewer of Simulation Data

Overview

The objective of the display program is to receive an input file (containing the simulation data) and display it graphically. The number of the particles, and the number of time steps is known in advance.

To make things more interesting, we assume that the viewer is also required to display the velocity of each particle and an optional label for it. (this is *only hypothetical*, our viewer - and your simulation - will not behave this way)

Data Structures

The design of data structures should be influenced by (among others):

- Clarity. The organization of the data should be logical and meaningful, so that it'd be easy to understand and return to in the future.
- Usage. The way the data is used in the program should be taken into account in the design, e.g. if we usually need a particle's position and velocity, then we should keep them together.

The basic building block of the viewer's data is the *particle*. Displaying a particle, requires detailed information about it. Therefore, we shall define a basic data structure to hold this data.

```
struct particle {
  float mass;
  float charge;
  float radius;
  float position[3];            /* position - {x,y,z} */
  float velocity[3];            /* velocity - {v_x,v_y,v_z} */
  char label[20];               /* particle's label */
};
```

The other main "object" in the program is a *time step*:

```
struct timestep {
  float time;                   /* the actual time of this step */
  int num_ptcls;                /* the number of particles in this step
                                   (this changes due to collisions)*/
  struct particle particles[MAX_NUM_PTCLS]; /* the particles' data */
};
```

Finally, the whole input will be stored in an array of `timestep`s:

```
struct timestep sim_data[MAX_NUM_STEPS];
```

(Obviously, `MAX_NUM_PTCLS` and `MAX_NUM_STEPS` should be also specified and `#define`d in the program.)

Also, we should store all viewer parameters in a seperate structure:

```
struct viewer_params{
  float viewport[6];            /* viewport's co-ordiantes, stored as
                                   {min_x,max_x,min_y,max_y,min_z,max_z} */
  char neg_color[10];           /* color of negative charged particle */
  char pos_color[10];           /* color for positive charged particle */
  /* etc. */
};
```

Program Modules

The program can be divided into logical parts ("modules"), each responsible for a specific task and, usually, has little to do with the others. We can consider the following modules:

- **Input** (processing the input files and converting it to internal data structures)
- **Graphics** (initializing the graphics display, performing basic graphics operations)
- **"Controller"** (the part which co-ordinates the operation of the entire program)

In the following section we shall describe the functions inside each of the modules.

Input

- `parse_viewer_params()` parses the viewer parameters from the input file and validates them.
- `parse_prtcl_input()` parses the particle data in the input files and creates the appropriate internal data structures to hold the simulation's data. This will fill the `sim_data` array.
- `validate_prtcl_input()` checks that the input is valid.

Graphics

- `init_graph()` initializes the graphics library(e.g., set the screen size, boundaries, etc.)
- `draw_sphere()` draws a sphere at a given point, with given parameters
- `draw_arrow()` draws an arrow at a given point, with a given direction (for the velocity plots)
- `draw_label()` draws a string at a given point (for the particle's label)
- `close_graphs()` perform graphics library clean up (when the program finishes)
- `draw_particle()` draws a particle (i.e., a sphere in the correct color, with its velocity and label)

Controller

- `main()` program entry point, responsible for coordinating everything