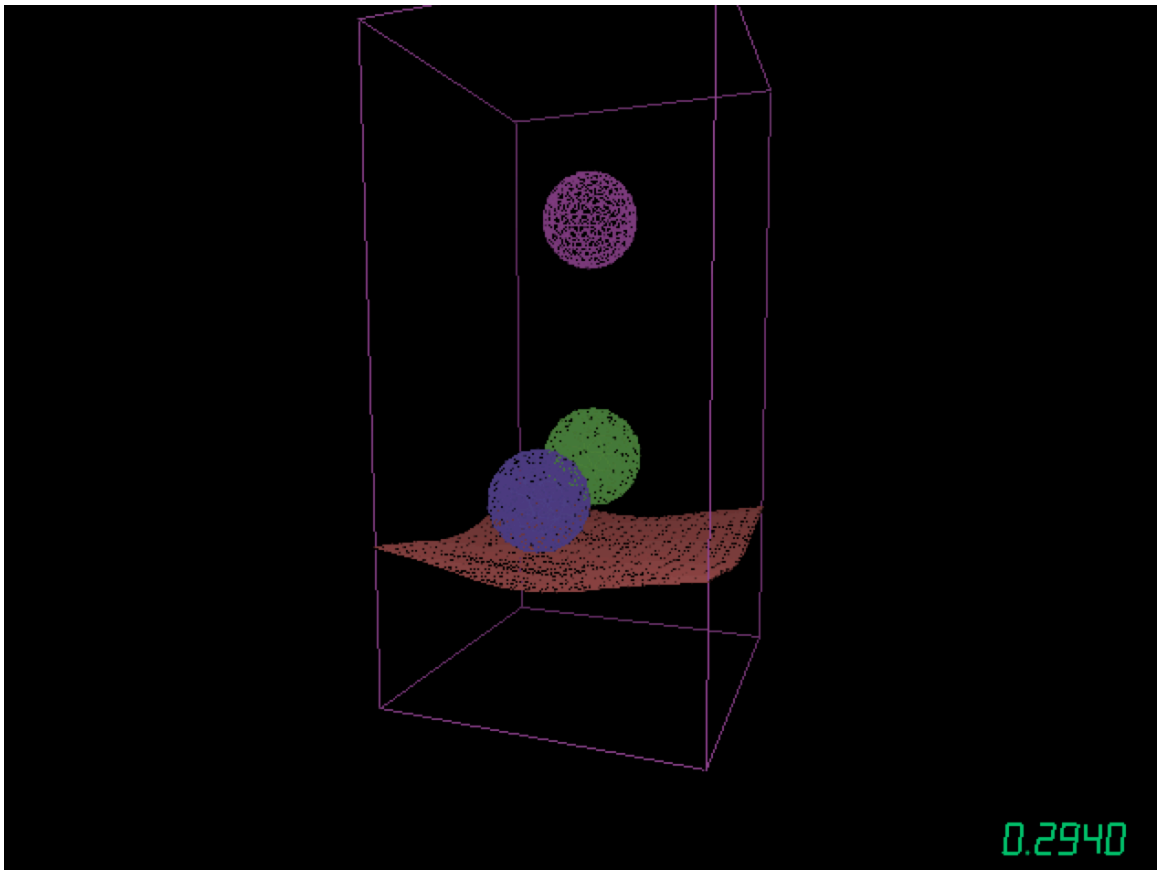
By Yingzhuo Zhang and Ruitao Du

Visualizer Extensions

Overview

For our final project, our team implemented some visualizer extensions, in which we added many interactive features to our SLDViewer using keyboard events. Some of these interactive events control the viewer such as automatically tracing and centering a moving object in the viewer, and some of the events make our simulations interactive such as start/pause/resume wind, add/reduce strength of wind, pause/resume and speed up or slow down the simulation, and moving any constraints during the simulation. Using these interactive features, along with the collaboration with some other teams that developed collision and tetrahedral mesh, our team developed our final project into a simulation of room (a rectangular box) with concrete walls, ceiling and floor in which two tetrahedral balls are bouncing on a trampoline, which is tied to the four corner of the walls, and there is a fan below the trampoline that could blow wind on the trampoline upon demand (from the keyboard). During the simulation the balls could potentially bounce off any of the four walls, the ceiling and floor of the room, or the trampoline, or each other. We have also put a fixed ball below the ceiling, so that the two tetrahedral balls could also bound off the fixed ball. Below is a screen shot of our final simulation.



By Yingzhuo Zhang and Ruitao Du

Functions and Features

1. **Auto tracing and centering**

We made some modifications in GLCamera.hpp, which enabled the camera to automatically trace moving objects. When an object is about to move out of bound, the viewer will detect that the object close to its boarder, so it will call camera to move its center so that the object will be in the center of the viewer again. Auto tracing and centering can be enabled and disabled at any time during the simulation using the keyboard. Auto tracing and centering function will have some impact on the original zoom in function on the mouse that came with our SDLViewer. If one zooms in small enough that the object(s) are still quite far from the boarders of the viewer, the zoom in feature will work as it is. However if one tries to zoom in too much that the object(s) are moving out of bounds, auto tracing and centering will take in effect and overwrite the zoom in to move the object(s) back to the center. The original zoom out function on the mouse will not affected since zooming out will not cause an already existing object to fall out of bounds. The original rotate feature of the viewer is still working and will not be affected by any of our added functions.

Keyboard instruction for this feature is:

- <T> key for pausing and resuming the wind

2. **Interaction with the wind**

As described above, we simulated a huge fan under the trampoline that can blow wind from below onto the trampoline. We defined our own wind force, and this interactive function could pause and resume the wind at any time. When the wind force is active, one could increase or decrease the strength of the wind.

Keyboard buttons for these features are:

- <Space key> for pausing and resuming the wind
- <UP arrow key> for increasing the wind
- <DOWN arrow key> for decreasing the wind

3. **Interaction with the simulation**

One can pause and resume the entire simulation at any time, and when the simulation is running, one can increase or decrease the speed of the simulation (i.e. increasing and decreasing dt). Note that since dt can't not be too large or the simulation would not work as expected, so one should be cautious when using the speed up and slow down feature because they will incur potential breakdown of the simulation.

Keyboard instructions for these features are:

- <P> key for pausing and resuming the simulation
- <O> for speeding up the simulation
- <L> for slowing down the simulation

4. **Interaction with constraints**

When we specify constraints such as a ball, a sphere, etc, one could activate and deactivate a constraint. When a constraint is activated, one can move the position of

these constraints at any time. For our final simulation, we created a ball shaped constraint on the trampoline that will press down the trampoline upon activation. When this ball constraint is deactivated, the trampoline will bounce back up

Keyboard instructions for these features are:

- <W> key for moving the constraint on the z-axis in the positive direction
- <S> key for moving the constraint on the z-axis in the negative direction
- <D> key for moving the constraint on the y-axis in the positive direction
- <A> key for moving the constraint on the y-axis in the negative direction
- <F> key for moving the constraint on the x-axis in the positive direction
- <R> key for moving the constraint on the x-axis in the negative direction
- <E> key for activating/deactivatin a constraint

Implementation

There are two types of visualizer extensions in our project. One is the auto tracing and centering feature; the other is various interactions feactures using the keyboard. For auto tracing and centering, we added two functions in GLCamera.hpp that returns the center position of where the camera is currently focusing on and the distance between that center of focus and the camera. Below is the code snippet for the functions we added into GLCamera.hpp.

```
/** Return Center of Camera
 */
inline const Point center() const{
  return point;
}

/** Return Distance of Camera
 */
inline float distance() const{
  return dist;
}
```

In SLDViewer.hpp, we added a private variable center_ that stores the position of the object, then we keep track of this center_ in the add_node(…) function. Every time a node is added, we check whether this center_ is within the range of the SDLViewer, if it is not, then we call the function viewpoint(Point point) in GLCamera.hpp to move the focus position of the camera. We also added private variables to keep track of the dimensions of the object so that if the object itself is already larger than the SDLViewer, the camera will not only move its focus but also zoom out so that we can see the entire object. Below is a code snippet of what we added into add_nodes(…) function in SDLViewer.

```
void add_nodes( ... ) {
  // Lock for data update
  { safe_lock mutex(this);

    for (; first != last; ++first) {
      // Get node and record the index mapping
      auto n = *first;
      auto r = node_map.insert(typename Map::value_type(n,coords_.size()));
      Point pos = position_function(n);
      if (r.second) {   // new node was inserted
        coords_.push_back(pos);
        colors_.push_back(color_function(n));
      }
      else {          // node already exists and not updated
        unsigned index = r.first->second;
        center_ -= coords_[index];
        coords_[index] = pos;
        colors_[index] = color_function(n);
      }
      if(pos.x>maximum_[0]) maximum_[0] = pos.x;
      if(pos.x<maximum_[1]) maximum_[1] = pos.x;
      if(pos.y>maximum_[2]) maximum_[2] = pos.y;
      if(pos.y<maximum_[3]) maximum_[3] = pos.y;
      if(pos.z>maximum_[4]) maximum_[4] = pos.z;
      if(pos.z<maximum_[5]) maximum_[5] = pos.z;
      center_ +=pos;
    }

    // Zoom out if graph is large
    if(auto_track_){
      if((maximum_[0]-maximum_[1])>camera_.distance()*0.75) camera_.zoom(1.2);
      else if((maximum_[2]-maximum_[3])>camera_.distance()*0.75) camera_.zoom(1.2);
      else if((maximum_[4]-maximum_[5])>camera_.distance()*0.75) camera_.zoom(1.2);
    }

    Point dis=center_/coords_.size()-camera_.center();
    if(auto_track_&& (norm(dis)>camera_.distance()/2) ){
      camera_.view_point(center_/coords_.size());
      }
    }
  request_render();
}
```

For the various interaction features, we added the listener objects that are notified whenever a keyboard event is received. First we need to add a listener object in SDLViewer that enables the viewer to register listeners. Then we wrote three listeners for feature 2, 3, and 4 described in the above section in mess_mass_sping.cpp. Below we put two code snippets, the first snippet is what we added in SLDViewer to register listener objects, and the second on is one example of the actual listener object.

CS207 Final Project Report

By Yingzhuo Zhang and Ruitao Du

Snippet 1:

```
//listener
struct Listener{
  virtual void handle(SDL_Event e)=0;
};

void add_listener(std::shared_ptr<Listener> l){
  listeners_.push_back(l);
}
```

Snippet 2:

```
//customize my listener to change the wind
struct Listener_Wind: public CS207::SDLViewer::Listener{
  WindForce& wind;
  double pre_level;
  bool is_pause=false;
  Listener_Wind(WindForce& w): wind(w), pre_level(wind.level){}
  void handle(SDL_Event e){
    switch (e.type) {

      case SDL_KEYDOWN: {
        // Keyboard 'arrow up' to increase wind
        if (e.key.keysym.sym == SDLK_UP){
          wind.level+=0.01;
          pre_level=wind.level;
          std::cout<<"Increase wind"<<std::endl;
        }
        // Keyboard 'arrow down' to decrease wind
        if (e.key.keysym.sym == SDLK_DOWN){
          wind.level-=0.01;
          pre_level=wind.level;
          std::cout<<"Decrease wind"<<std::endl;
        }
        // Keyboard 'space' to pause and resume the wind
        if (e.key.keysym.sym == SDLK_SPACE){
          is_pause ^= true;
          if(is_pause){
            wind.level = 0;
            std::cout<<"Pause wind."<<std::endl;
          }
          else{
            wind.level = pre_level;
            std::cout<<"Resume wind."<<std::endl;
          }
        }
      } break;

    }
  }
};
```

CS207 Final Project Report

By Yingzhuo Zhang and Ruitao Du

The trampoline in our final simulation is implemented with the Graph class we developed during the semester. All three balls in the final simulation are collaboration work with other teams

Collaboration with other teams

We attempted to collaborate with four other teams to develop our final simulation. First we attempted to use the bouncing ball developed by Tian Lan and Xide Xia in their meshed mass spring, and we used their bouncing ball as the fixed ball, which the other two tetrahedral balls bounce upon in our final simulation. Next, we incorporated the tetrahedral balls developed by Jeff Shen and Yung-jen. We used their mesh class and tetrahedral mesh class in our meshed_mass_spring to produce the three balls. Then, we used the collision detection algorithm by Erik and Lukas; however, their collision detection works only with triangle class, whereas we are using tetrahedral balls, so we followed their algorithm and wrote our own collision algorithm for tetrahedral mesh to detect collisions. We also used part of Brian Zhang's algorithm in the interaction between the trampoline and the tetrahedral balls, so that the balls could bounce on and off the trampoline.