Assignment 5: Cloth Animation

NAME: BINGNAN LI
STUDENT NUMBER: 2020533092
EMAIL: LIBN@SHANGHAITECH.EDU.CN

ACM Reference Format:
Name: Bingnan Li, student number: 2020533092, email: libn@shanghaitech.edu.cn. 2022. Assignment 5: Cloth Animation. 1, 1 (December 2022), **??** pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

- Force computation with Hooke's law.
- Structural, shear, and bending springs.
- Fix the location of two mesh points to stop the cloth falling down.
- Real-time and stable animation.
- Apply external forces to the cloth to simulate the behavior of wind.
- Add a sphere or cube obstacle to simulate a piece of cloth falling on a sphere or a cube with collision handling.
- Drag a mesh point to move the cloth with mouse in real-time.

2 IMPLEMENTATION DETAILS

(1) Force computation with Hooke's law.

By Hooke's law, we have

$$f = k\Delta x$$

where k is stiffness.

Hence, the programming code is

```
Vec3 ComputeHookeForce(int iw_this, int
    ih_this, int iw_that, int ih_that,
    Float dx_world) const {

size_t this_idx, that_idx;

if (!Get1DIndex(iw_this, ih_this,
    this_idx) || !Get1DIndex(iw_that,
    ih_that, that_idx)) {
return {0, 0, 0};
}
Vec3 p = local_or_world_positions.at(
    this_idx);
Vec3 q = local_or_world_positions.at(
    that_idx);
Vec3 dis = p - q;

return stiffness * (dx_world - glm::
    length(dis)) * glm::normalize(dis);
}
```

(2) Structural, shear, and bending springs.

By this illustration,

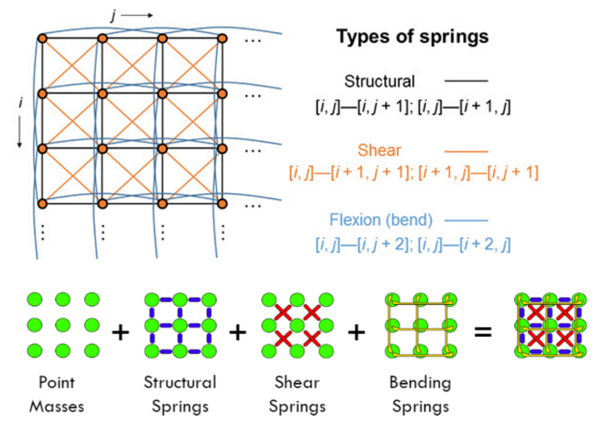
j
i
Types of springs
Structural
[i, j]—[i, j + 1]; [i, j]—[i + 1, j]
Shear
[i, j]—[i + 1, j + 1]; [i + 1, j]—[i, j + 1]
Flexion (bend)
[i, j]—[i, j + 2]; [i, j]—[i + 2, j]
Point Masses
Structural Springs
Shear Springs
Bending Springs

Fig. 1. simple box

we know that the spring force contains three types of forces, which are structural, shear and bending forces. Thus, we have the following code

```
Vec3 RectCloth::ComputeSpringForce(int
    iw, int ih) const {

  const Vec3 scale = object->transform->
     scale;

  Vec3 springForce(0);
  size_t idx;
  if (!Get1DIndex(iw, ih, idx) ||
     is_fixed_masses.at(idx)) {
    return springForce;
  }
```

Author's address: Name: Bingnan Li
student number: 2020533092
email: libn@shanghaitech.edu.cn.

```
  springForce += ComputeHookeForce(iw,
      ih, iw, ih - 1, dx_local);
  springForce += ComputeHookeForce(iw,
      ih, iw, ih + 1, dx_local);
  springForce += ComputeHookeForce(iw,
      ih, iw - 1, ih, dx_local);
  springForce += ComputeHookeForce(iw,
      ih, iw + 1, ih, dx_local);

  springForce += ComputeHookeForce(iw,
      ih, iw - 1, ih - 1, (float) std::
      sqrt(2) * dx_local);
  springForce += ComputeHookeForce(iw,
      ih, iw - 1, ih + 1, (float) std::
      sqrt(2) * dx_local);
  springForce += ComputeHookeForce(iw,
      ih, iw + 1, ih - 1, (float) std::
      sqrt(2) * dx_local);
  springForce += ComputeHookeForce(iw,
      ih, iw + 1, ih + 1, (float) std::
      sqrt(2) * dx_local);

  springForce += ComputeHookeForce(iw,
      ih, iw, ih - 2, 2 * dx_local);
  springForce += ComputeHookeForce(iw,
      ih, iw, ih + 2, 2 * dx_local);
  springForce += ComputeHookeForce(iw,
      ih, iw - 2, ih, 2 * dx_local);
  springForce += ComputeHookeForce(iw,
      ih, iw + 2, ih, 2 * dx_local);

  return springForce;
}
```

(3) Damping.

Once we get new spring force and gravity, we need to add damping so that the cloth would not keep fluctuating.

By the physical inference, we have

$$a = TotalForce - damping\_ration \times v$$

Thus, we have the following code:

```
void RectCloth::ComputeAccelerations() {
  std::normal_distribution<float> nd(ud(
      gen), 2 * std::max(ud(gen), 5.0f))
      ;
  for (int w = 0; w < mass_dim.x; ++w) {
    for (int h = 0; h < mass_dim.y; ++h)
        {
      size_t idx;
      if (Get1DIndex(w, h, idx) && !
          is_fixed_masses.at(idx)) {
        world_accelerations.at(idx) =
            (ComputeSpringForce(w, h) -
                damping_ratio *
                world_velocities.at(idx)
                ) / mass_weight;
      }
    }
  }
}
```

(4) External force

In order to add external force, we need to add a new random acceleration to the total acceleration.

(5) Collision with sphere.

Once a mass was fell into the sphere, we firstly set its position to the surface of sphere along the normal direction. Then, we need to set the component along the normal direction of velocity to zero. Therefore, we have the following code:

```
if (glm::distance(
    local_or_world_positions.at(i),
    SphereCenter) < R) {
  Vec3 normal = glm::normalize(
      local_or_world_positions.at(i) -
      SphereCenter);
  local_or_world_positions.at(i) =
      SphereCenter + R * normal;
  Vec3 v_n_component = glm::dot(
      world_velocities.at(i), normal) *
      normal;
  world_velocities.at(i) -=
      v_n_component;
}
```

(6) Interactive points

In order to achieve interaction, we first generate a ray based on the mouse position. We can use $glm :: unproject$ function to do that. After that, we need to check all masses about whether the ray interacts with a sphere whose sphere center is the location of mass and a radius 0.05. If it did, then we set its velocity to zero and a drag index to its index.

After that, once the drag index in not -1, we set the dragged point to the new position based on the new mouse position.

Hence, we have the following code:

```
if (drag_idx != -1 && Input::
    GetMouseButton(0)) {
  Vec3 p = local_or_world_positions.at(
      drag_idx);
  world_velocities.at(drag_idx) = Vec3
      (0);
  local_or_world_positions.at(drag_idx)
      = GetDragPos(p);
  return;
}

if (Input::GetMouseButton(0)) {
  Vec3 dir = glm::normalize(
      GetWorldPosFromCursor() - cam->
      transform.position);
  for (size_t i = 0; i < length; ++i) {
    if (Interact(cam->transform.position
        , dir, i)) {
      world_velocities.at(i) = Vec3(0);
      drag_idx = (int) i;
      break;
    }
  }
} else {
  drag_idx = -1;
}
```

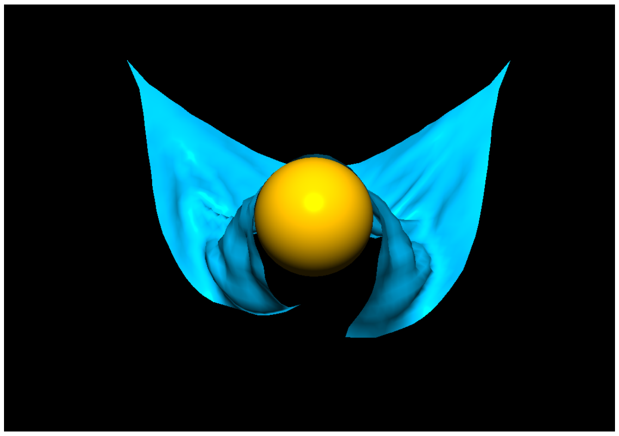

Fig. 3. Sphere Collision

3 RESULTS

pictures should be in

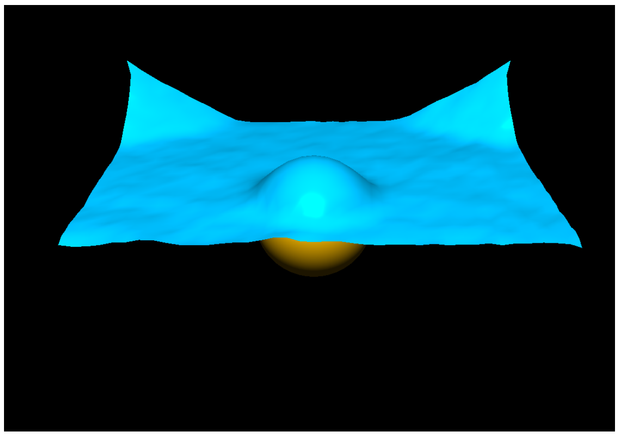

Fig. 2. Sphere Collision

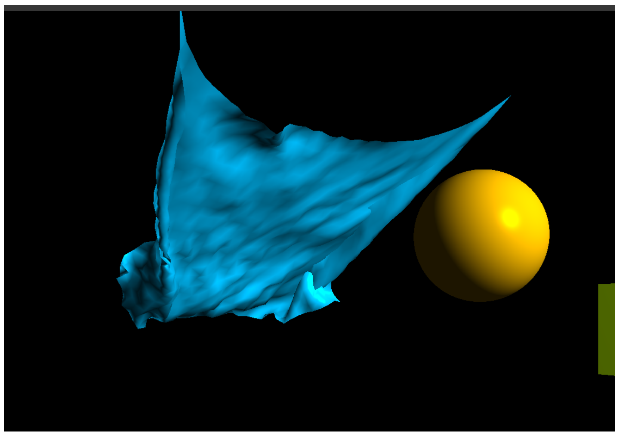

Fig. 4. Sphere Collision

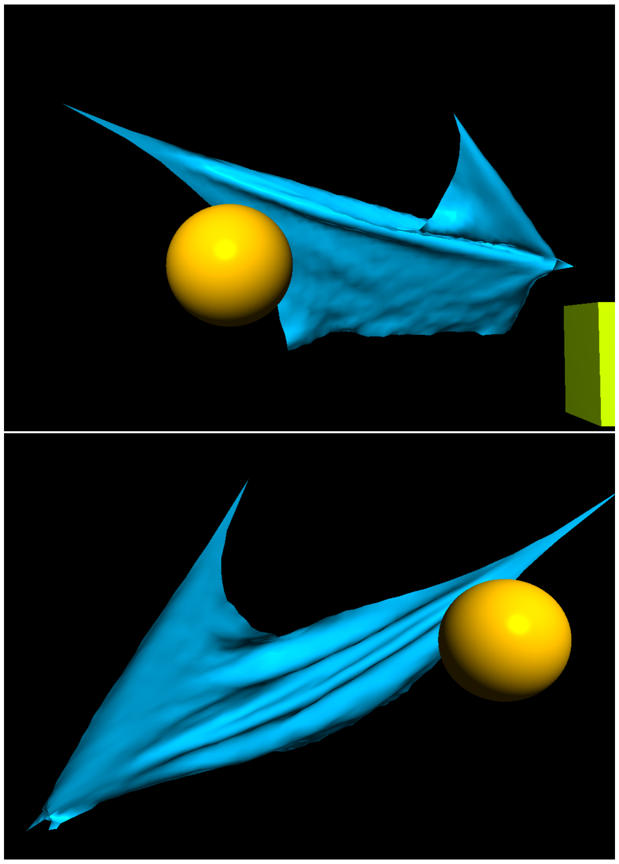

Fig. 5. Interactive points