Homework 4

Computer Graphics

Francesco Palandra 1849712

January 9, 2022

Contents

1 Wind
(easy) 3

2 Vortex
(easy) 4

3 Tornado
(medium) 5

4 Cloths
(hard) 6

1 Wind
(easy)

For the wind it has been added the wind contribution for the predict of the positions. For each position velocity a vector, that has been calculated as the power of the wind on the direction of the wind, has been added for the new position.

Algorithm 1 Wind algorithm

Input: $vec3f\ d,\ float\ k$

$wind \leftarrow d \cdot k$

for $(eachshape\ s)$ **do**

  for $(eachposition\ p)$ **do**

    $s.velocities[p] \leftarrow += (wind - gravity) \cdot deltat$

    $s.positions[p] \leftarrow += s.velocities[p] \cdot deltat$

  end for

end for

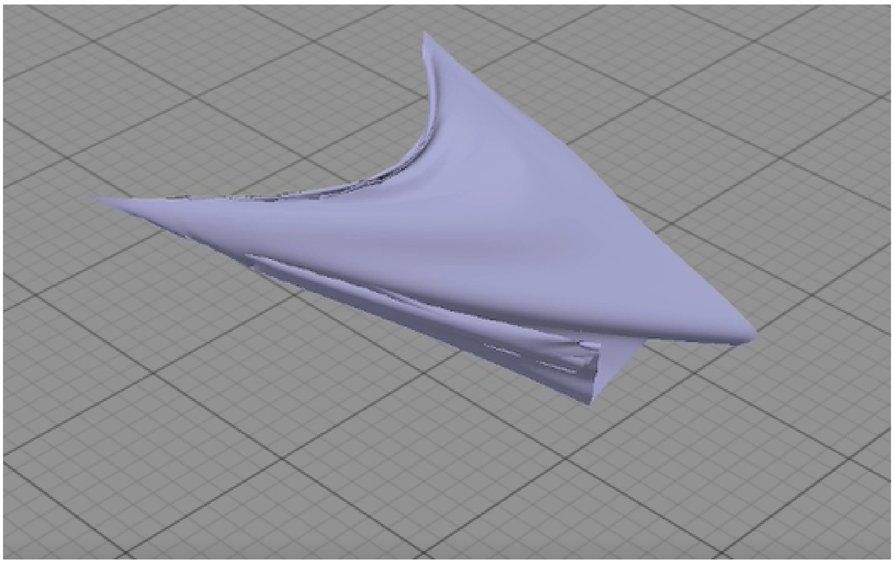

Figure 1: Wind

2 Vortex
(easy)

For the vortex case the strategy adopted is similar to the previous one, the only different is that the wind now is not fixed but change according to the position of the particles. The wind will have a circular trajectory around one direction, hence for each position of each shape a vector ,tangent to that circular trajectory, multiplied by the wind force over the distance from the center of the vortex squared, will be added to the predicted position.

Algorithm 2 Vortex algorithm

Input: $vec3f\ d,\ float\ k$

for $(eachshape\ s)$ **do**
    for $(eachposition\ p)$ **do**
        $o \leftarrow \{\ d.x,\ p.y,\ d.z\ \}$
        $distance \leftarrow distance(p, o)$
        $direction \leftarrow normalize(p - o)$
        $vortex \leftarrow normalize(cross(o, d)) \cdot \frac{k}{d^2}$
        $s.velocities[p] \leftarrow += (vortex - gravity) \cdot deltat$
        $s.positions[p] \leftarrow += s.velocities[p] \cdot deltat$
    end for
end for

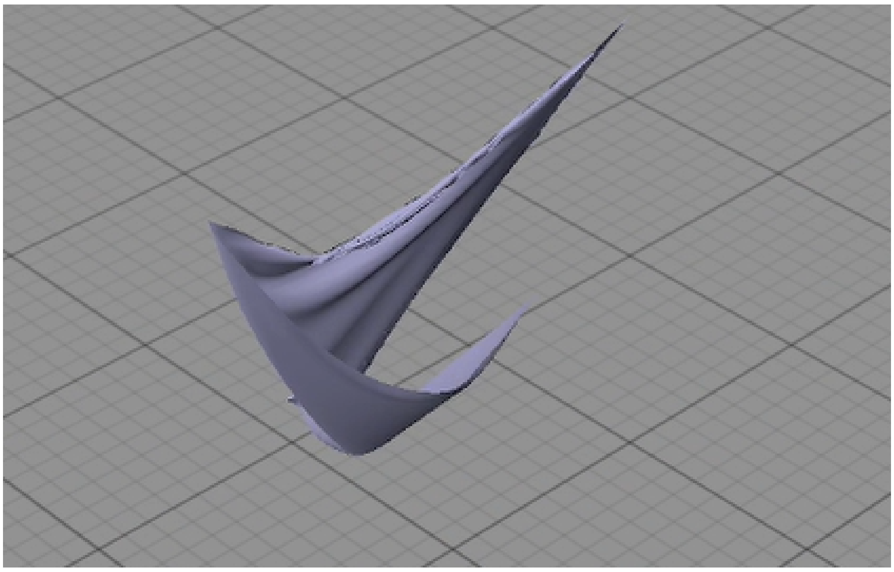

Figure 2: Vortex

3 Tornado
(medium)

In this section a tornado, or something that seems a tornado, has been created, the idea is that a particle starts at a bottom and rise following a spiral trajectory the expand according to the high from the ground.
The problem has been divided into 3 parts, one for each axis.
On the y axis according to the power of the wind a particle can go higher or lower.
On the z axis each particle spin around the center of the tornado with a velocity proportional to the power of the wind.
On the x axis the behavior is a bit particular, in order to get the shape of a tornado, each particle is attracted to the surface with a velocity proportional to the distance between them. A particle outside tends to go towards the center of the tornado until it reaches the surface, instead if the distance between the center of the tornado and the particle is less than the distance between the center of the tornado and the surface, the particles tends to move outwards.

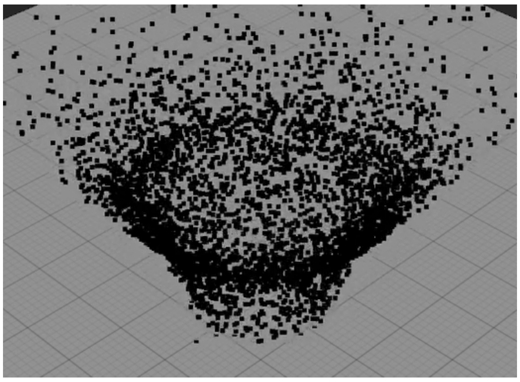

Figure 3: Tornado

Algorithm 3 Tornado algorithm

Input: $vec3f\ d,\ float\ k$

for $(eachshape\ s)$ **do**
    for $(eachposition\ p)$ **do**
        $o \leftarrow \{\ d.x,\ p.y,\ d.z\ \}$
        $direction \leftarrow normalize(p - o)$
        $surface \leftarrow o + direction \cdot log(h)$
        $distance \leftarrow distance(p, surface)$
        $y \leftarrow \{\ 0,\ 1,\ 0\ \} \cdot k$
        $z \leftarrow normalize(cross(o, distance))$
        if $distance(p, o) < distance(surface, o)$ **then**
            $x \leftarrow normalize(surface - p)/distance$
        else
            $x \leftarrow normalize(p - surface)/distance$
        end if

        $vortex \leftarrow x + y + z$
        $s.velocities[p] \leftarrow += (vortex - gravity) \cdot deltat$
        $s.positions[p] \leftarrow += s.velocities[p] \cdot deltat$
    end for
end for

4 Cloths
(hard)

In this section will be explained a failed attempt to solve the collision detection problem due to cloths. Many attempts with different strategies have been developments, many of which failed for the same reason: the signature distance function, or SDF.
In fact the algorithm proposed [paper] generate contacts between non-convex rigid bodies by point sampling triangle mesh features within each overlapping shape's SDF. At each particle' location we store the SDF and his gradient and the we detect the collision, if present. A model of the cloth is create using networks of distance constraints along triangle edges to model stretching, and across edges to model bending.
An other computation is for the error estimation, initially for each cell we estimate the error contributed by the cell, accumulate the error in the total error variable and insert the cell index based on its error contribution into a priority queue. Then in a loop we refine the approximation as long as the error exceeds a certain threshold θ.
All the problems occurs when a SDF has been generated, more specifically on an integral over the surface of the object, in fact a multi-dimensional Gauss quadrature of order 4p algorithm is needed to e heuristically approximate this integral.
The SDF algorithm is commented on the end of the code due to the missing functions for the integral and the gradient. Here is a paper with the following algorithm that explain the method used, and above their amazing results.

Figure 4: Results of their algorithm

Algorithm 1: *hp*-adaptive SDF construction.

```
Data: n_x, n_y, n_z, τ, Ω, p_max, l_max
ε ← 0
n ← n_x n_y n_z
pending ← priority_queue{}
for e ← 0 to n do
    fit_polynomial(e, 2)              // Fit polynomial of lowest order 2 to each base cell e. Equation (5)
    ε_e ← estimate_error(e)           // Equation (6)
    ε ← ε + ε_e
    pending.push({e, ε_e})
end
while not pending.empty() and ε > τ do
    {e, ε_e} ← pending.pop()
    {p, l} ← {degree(e), level(e)}
    μ_e ← estimate_improvement_p(e)   // Equation (8)
    ν_e ← estimate_improvement_h(e)   // Equation (9)
    refine_p ← p < p_max and ( 1 == l_max or μ_e > ν_e )
    refine_h ← l < l_max and not refine_p
    if refine_p then
        fit_polynomial(e, p + 1)      // Equation (5)
        ε ← ε − ε_e
        ε_e ← estimate_error(e)       // Equation (6)
        ε ← ε + ε_e
        pending.push({e, ε_e})
    end
    if refine_h then
        children ← subdivide(e)       // Octree subdivision.
        ε ← ε − ε_e
        for j ∈ children do
            fit_polynomial(j, p)      // Equation (5)
            ε_j ← estimate_error(j)   // Equation (6)
            ε ← ε + ε_j
            pending.push({j, ε_j})
        end
    end
end
```

Figure 5: hp-adaptive algorithm

Thanks for this course.