Bound drag and drop

Lesson №18

Restrictions

Dragging with restrictions

Previous lesson

- Free (unbound) dragging
- Position depends only on mouse position

In reality

- Object motion is restricted
- Different reasons for restrictions
- Leads to “disagreement” between mouse and object

Reasons of restrictions

Conceptual (caused by design)

- Motion on a circle
- Motion on the surface of a sphere
- Motion on the edges of a cube

Physical (caused by external factors)

- The code does not work for all input values
- Difficult to support smooth motions
- Limited screen or window size

Idea

Parametric motion

- There are always parameters – at least mouse coordinates are such parameters

Techniques

- Binding parameters
- Binding calculated values
- Finding most useful results

Expectations

- No object dragging, only object relocation
- Mouse cursor used only for initial grabbing
- During motion the cursor may be away from the object

Goals

- Intuitive object motions
- Easy implementation

Dragging along a line

Dragging along a line and a segment

With linear combination

- Parameter is controlled by the mouse
- Linear combination uses this parameter to find position

Closest point

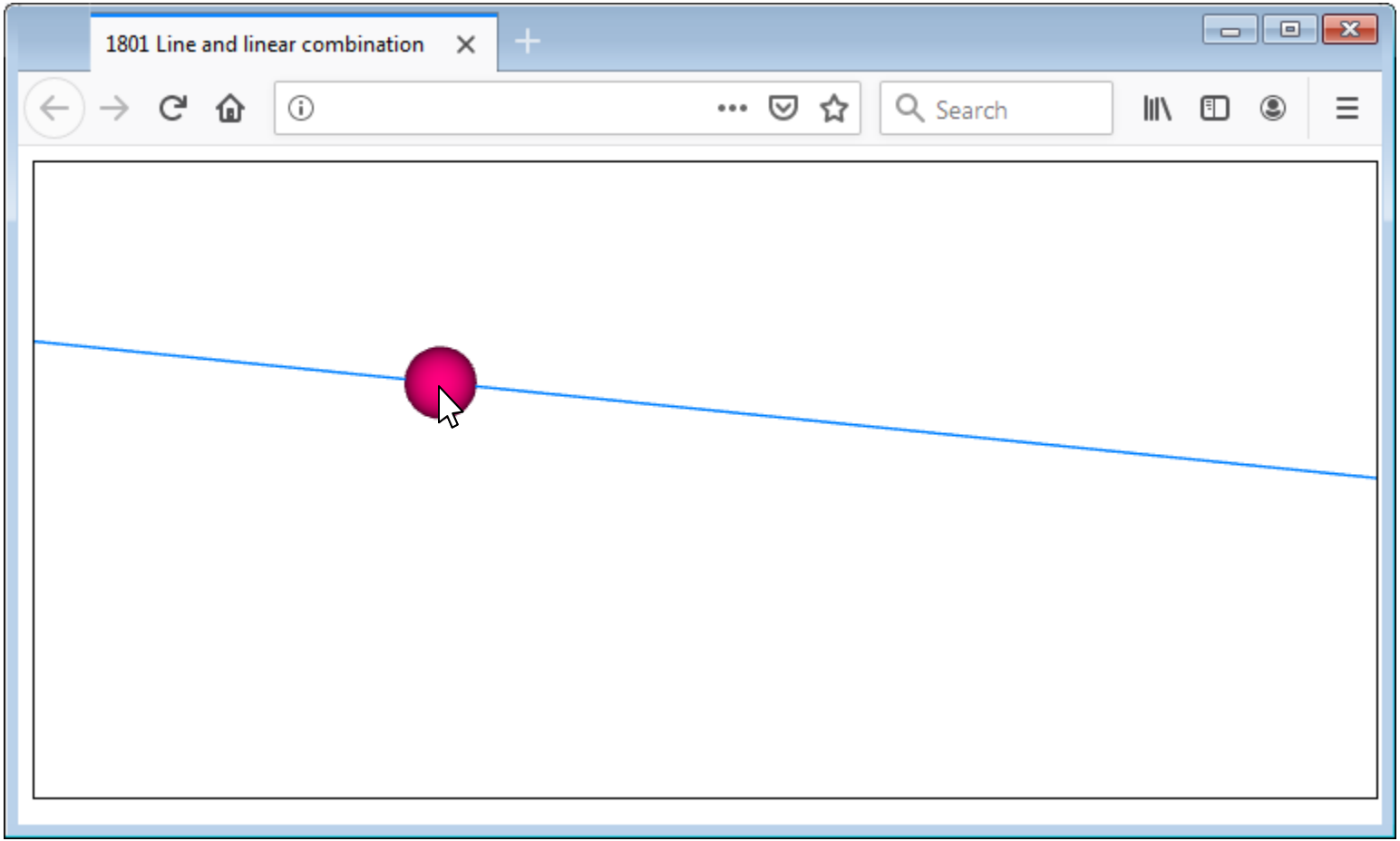
- More complex calculations
- Dragging appears more natural

Dragging with linear combination

Dragging along a line

- Object's center is on the line through points p1 and p2
- Captured motion changes linear combination's parameter k for finding the new center

```
if (obj)
{
   k -= (event.clientX-x)/500;
   s.center[0] = p1[0]*k+(1-k)*p2[0];
   s.center[1] = p1[1]*k+(1-k)*p2[1];
}
x = event.clientX;
```

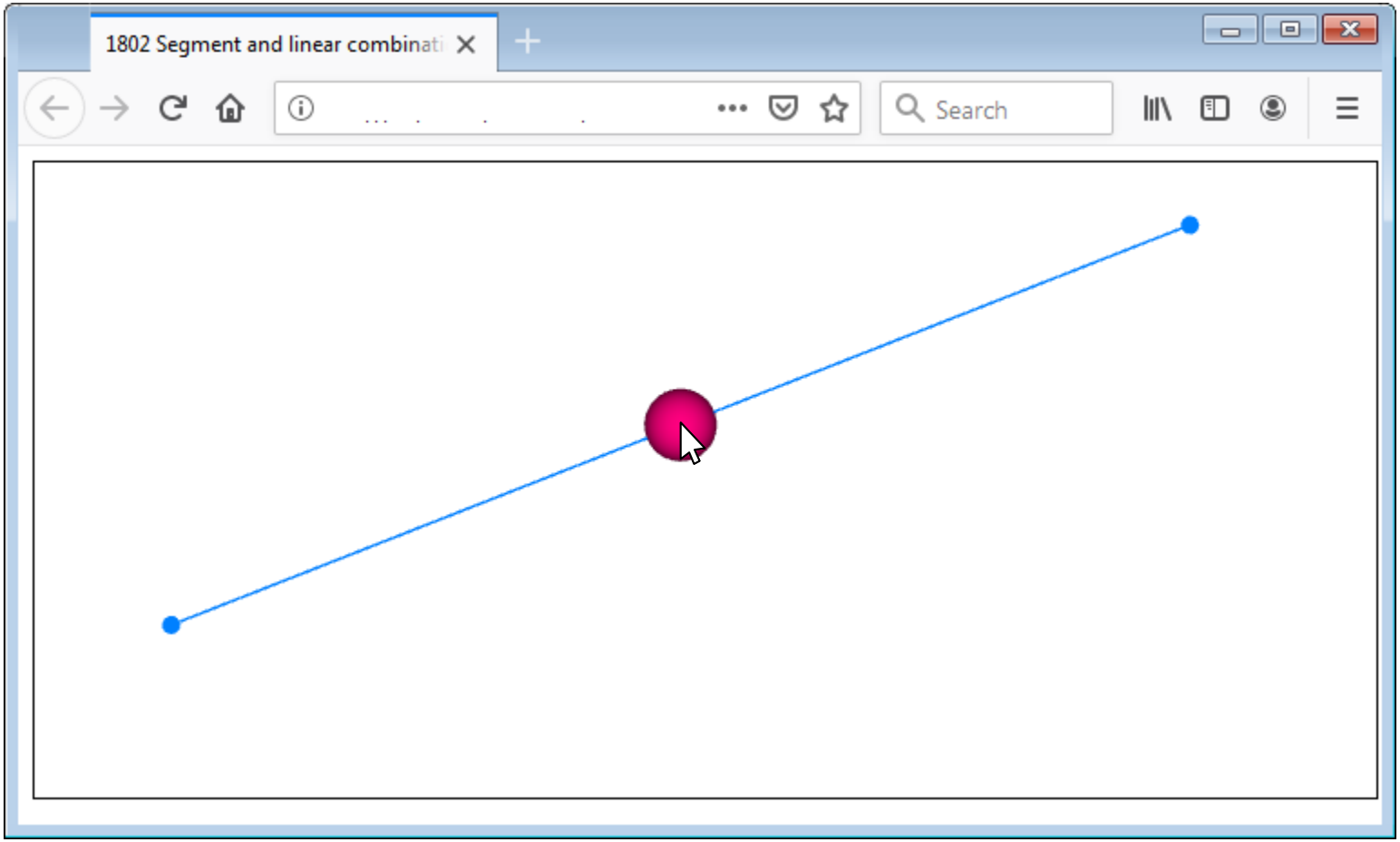


TRY IT

Dragging along a segment

- Similar to dragging along a line
- Additional restriction $k \in [0,1]$
- Link between mouse and object is broken – i.e. the cursor could go away from the segment (and the object)

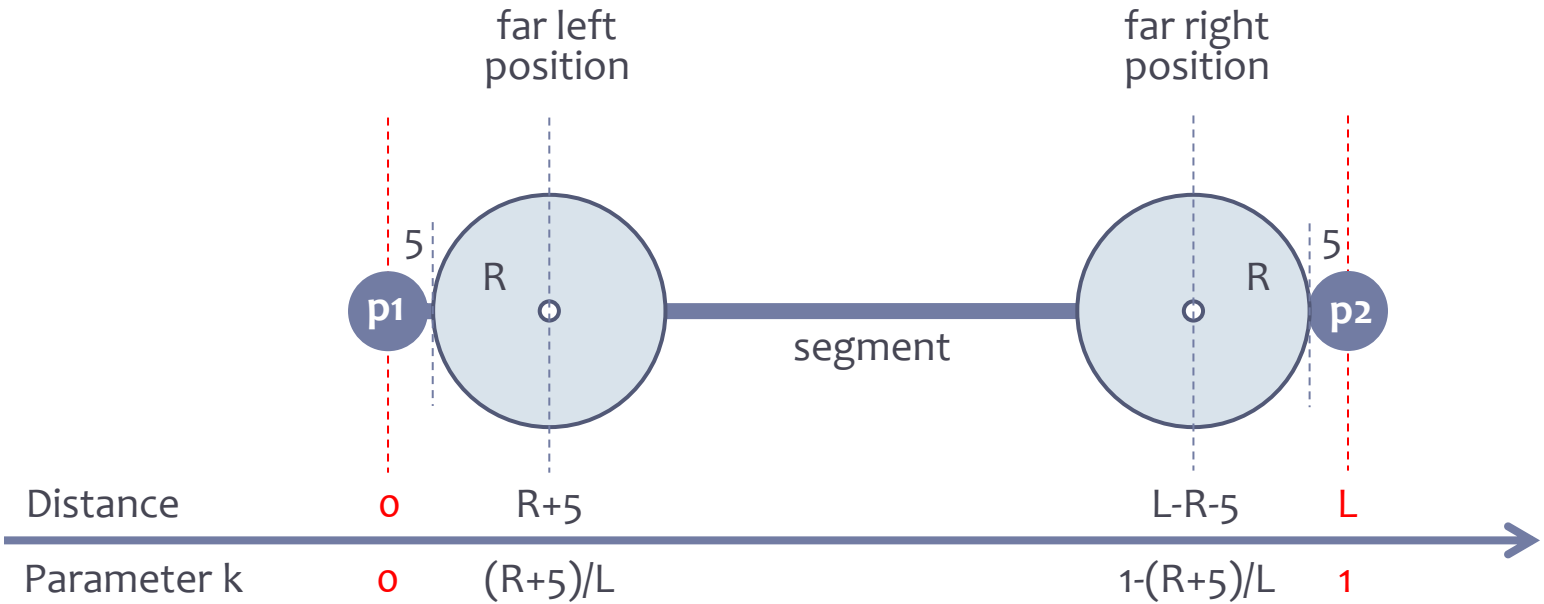
```
if (obj)
{
   k -= (event.clientX-x)/500;
   if (k<0) k=0;
   if (k>1) k=1;
   s.center[0] = p1[0]*k+(1-k)*p2[0];
   s.center[1] = p1[1]*k+(1-k)*p2[1];
}
```



TRY IT

Improvement

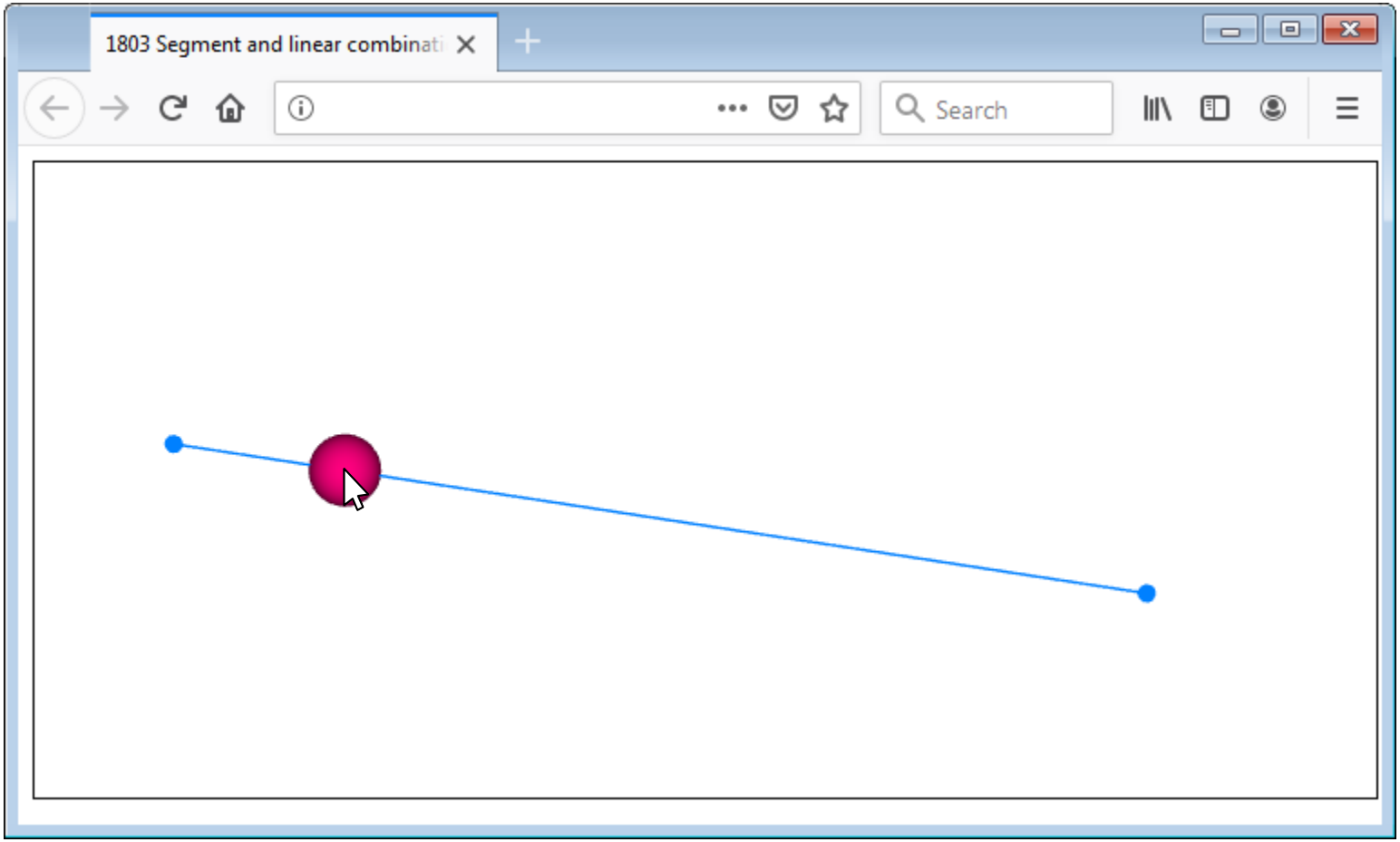
- Sphere is dragged up to the limiters



Implementation

- The domain of k is simetrically reduced with kLimit, i.e. $k \in [0+kLimit, 1-kLimit]$
- Value of kLimit is the sum of both radii (limiter's and object's) relative to the segment length

```
kLimit = (s.radius+5)/
        Math.sqrt( (p1[0]-p2[0])*(p1[0]-p2[0]) +
                   (p1[1]-p2[1])*(p1[1]-p2[1]) );
...
if (k<kLimit) k=kLimit;
if (k>1-kLimit) k=1-kLimit;
```

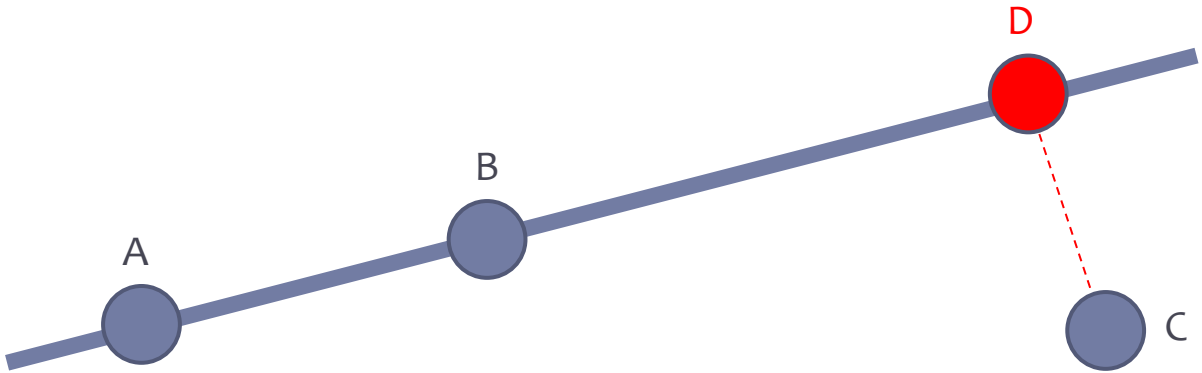


TRY IT

Closest point

The closest point problem

- Given a line through two points A and B
- Given a random point C
- Find point D on the line that is closest to C



Решение

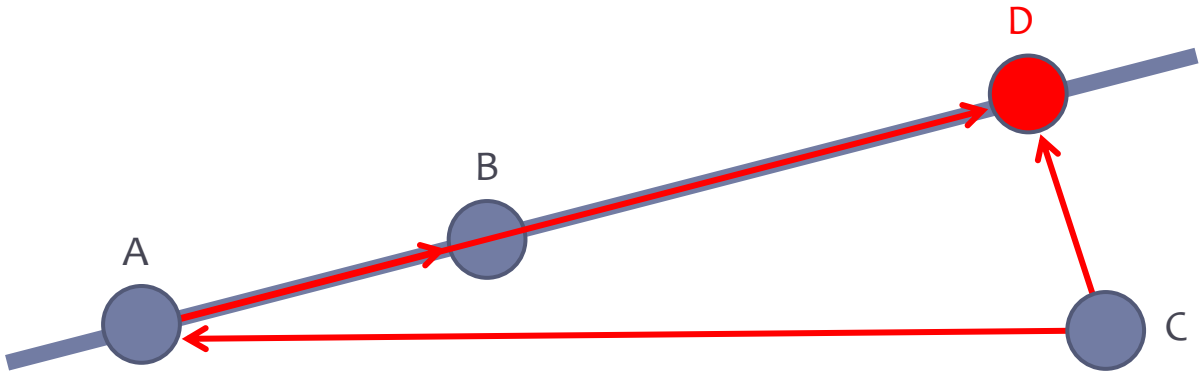
- Working with vectors:

$$\overrightarrow{AD} = \overrightarrow{AB}.k$$

$$\overrightarrow{CD} = \overrightarrow{CA} + \overrightarrow{AD} = \overrightarrow{CA} + \overrightarrow{AB}.k$$

$$\overrightarrow{CD} \perp \overrightarrow{AD} \Rightarrow \overrightarrow{CD} \perp \overrightarrow{AB} \Rightarrow \overrightarrow{CD}.\overrightarrow{AB} = 0 \Rightarrow \left(\overrightarrow{CA} + \overrightarrow{AB}.k\right).\overrightarrow{AB} = 0$$

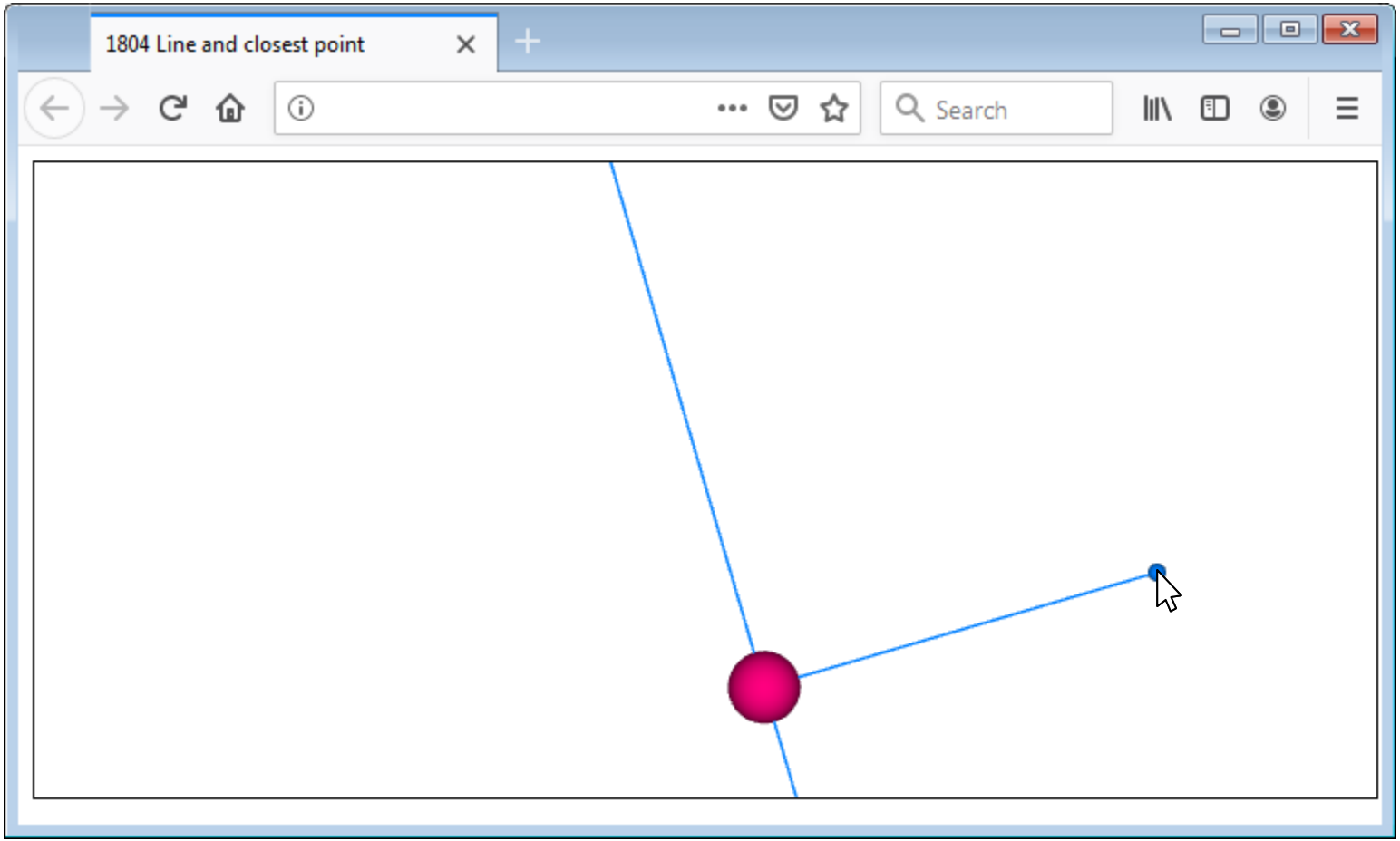
- Solving for k and $\overrightarrow{AD} = \overrightarrow{AB}.k$ leads to solution for D



Implementation

- Changing points A and B for a line in motion
- Calculating k from points A, B and C
- Finding point D on the line as $D = A + \overrightarrow{AB}.k$

```
A = [...];
B = [...];

AB = vectorPoints(B,A);
k =((C.center[0]-A[0])*AB[0]+(C.center[1]-A[1])*AB[1])
                              /(AB[0]*AB[0]+AB[1]*AB[1]);
D.center[0] = A[0]+k*AB[0];
D.center[1] = A[1]+k*AB[1];
```



TRY IT

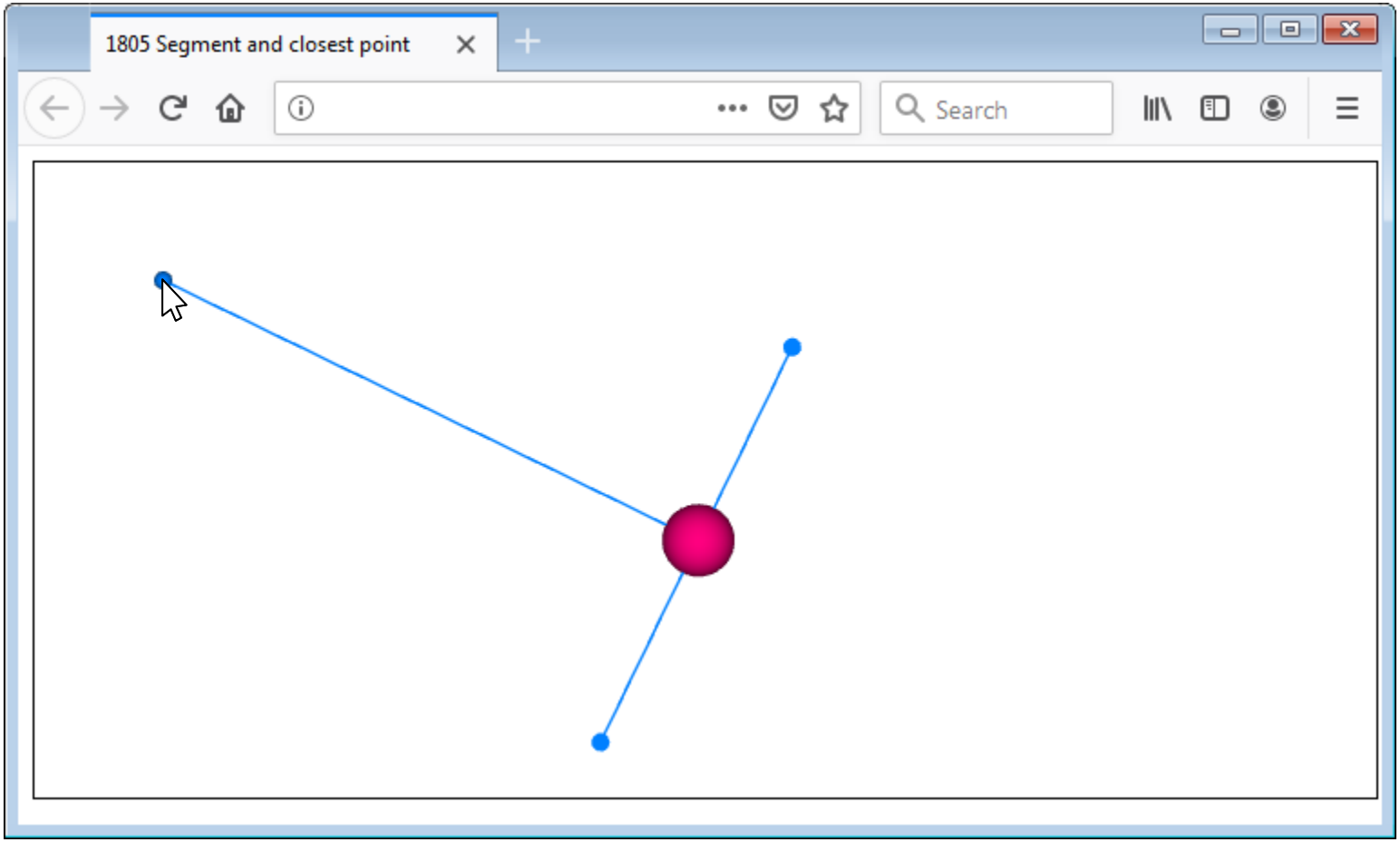
Closest point to a segment

- Same algorithm, same calculations
- For D to be between A and B, it needs $k \in [0,1]$
- This comes from the linear combination

Is it a coincidence?

- Checking: $A+\overrightarrow{AB}.k = A+(B-A)k = A(1-k)+kB$

```
k = ...;
if (k<0) k=0;
if (k>1) k=1;
```



TRY IT

Dragging with the closest point

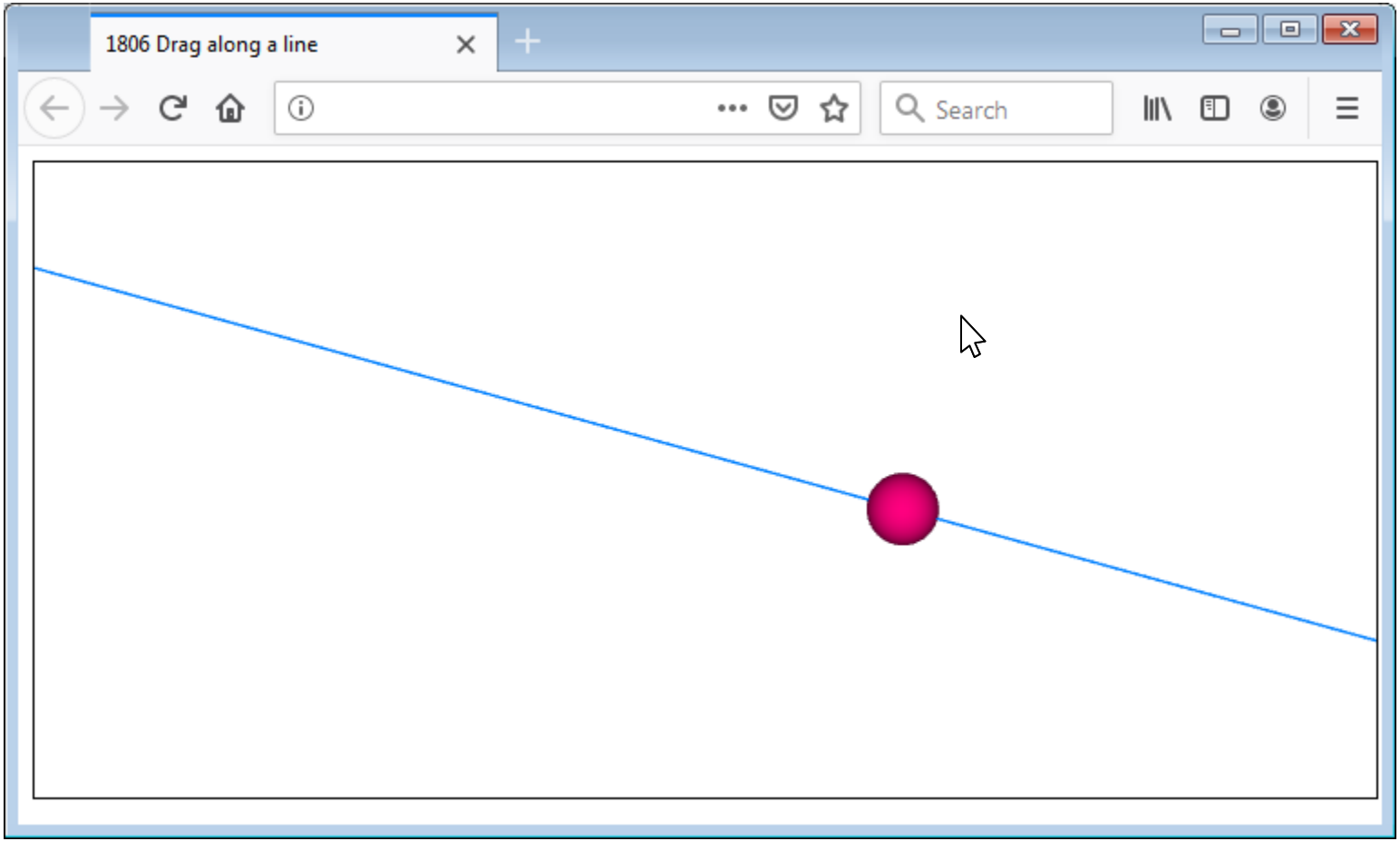
Dragging along a line

- Event processing remembers the last graphical coordinates and whether there is grabbed object

```
function mouseDown(event)
{
   obj = p.objectAtPoint(...);
   mouseMove(event);
}
function mouseUp(event) {obj = null;}
function mouseMove(event) {x = ...; y = -(...);}
```

- Vector AB and the square of its length L are calculated once (the line in the example is not moving)

```
AB = vectorPoints(B,A);
L = AB[0]*AB[0]+AB[1]*AB[1];
...
function animate()
{
   if (obj)
   {
      k = ((x-A[0])*AB[0]+(y-A[1])*AB[1])/L;
      D.center[0] = A[0]+k*AB[0];
      D.center[1] = A[1]+k*AB[1];
   }
}
```

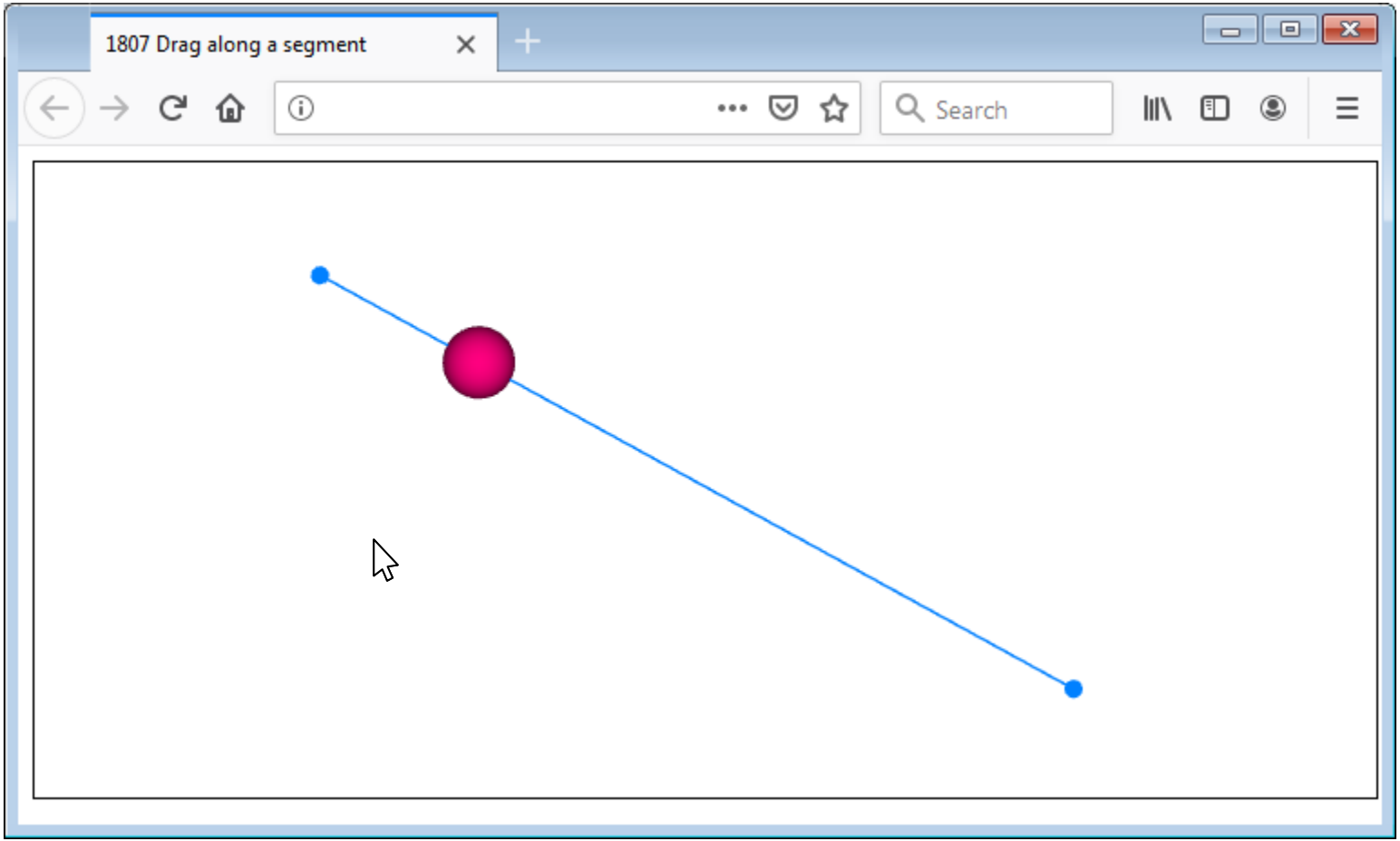


TRY IT

Dragging along a segment

- Adding restriction on k

```
function animate()
{
   if (obj)
   {
      k =((x-A[0])*AB[0]+(y-A[1])*AB[1])/L;
      if (k<0) k=0;
      if (k>1) k=1;

      D.center[0] = A[0]+k*AB[0];
      D.center[1] = A[1]+k*AB[1];
   }
}
```



TRY IT

Dragging along a circle

Dragging along a circle

With angle

- Controlling a parameter with the mouse
- Using the parameter as an angle in polar coordinates
- Using for dragging along a circle or an arc

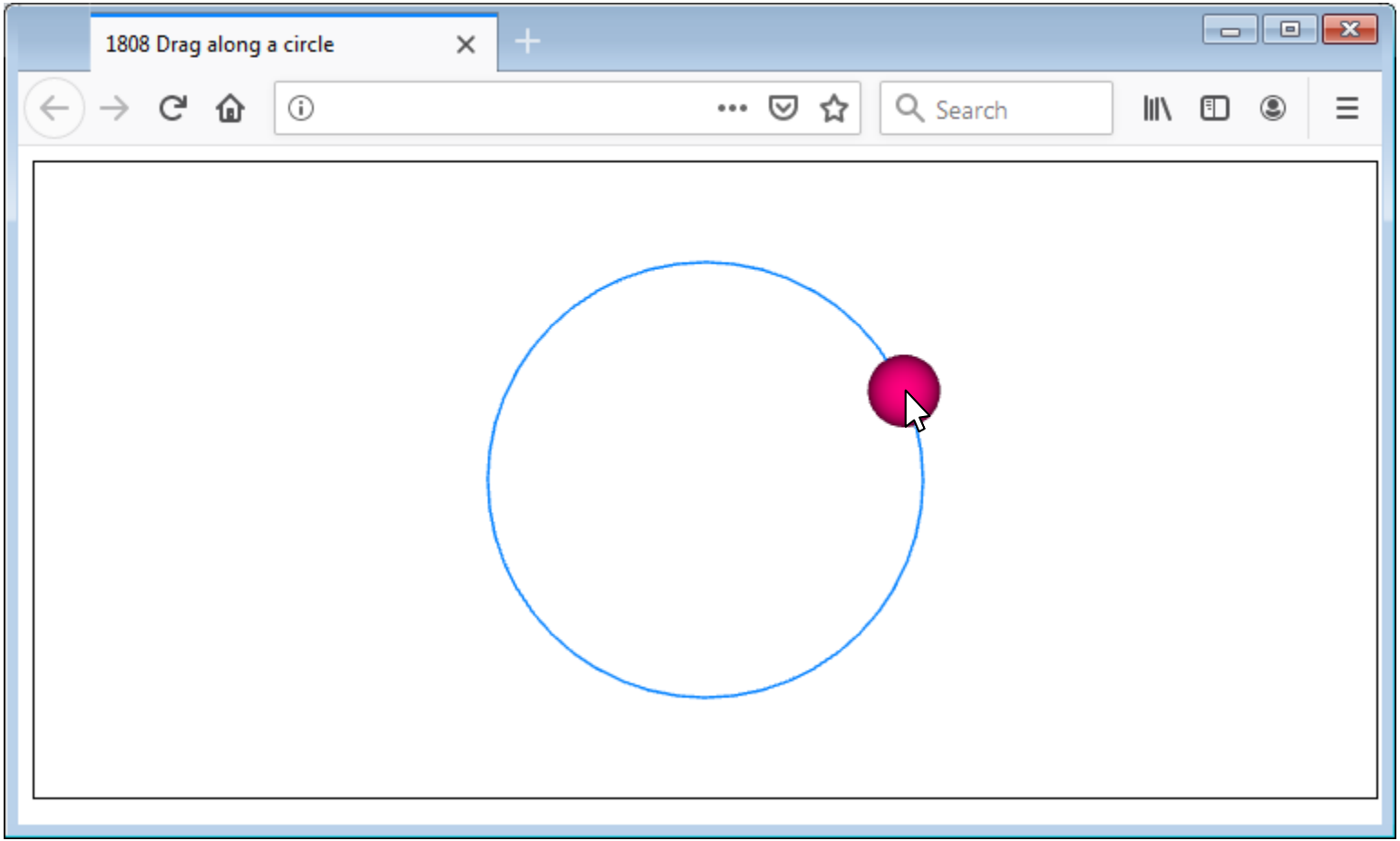
With closest point

- Projecting the point on the circle
- Harder for an arc

Dragging with an angle

- Local property alpha for the angle along the circle
- Offset divided by 100 – i.e. 100 pixels is 1 radian

```
function mouseMove(event)
{
   if (obj)
   {
      obj.alpha -= (event.clientX-x)/100;
      obj.center = [120*Math.cos(obj.alpha),
                    120*Math.sin(obj.alpha),0];
   }
   x = event.clientX;
}
```

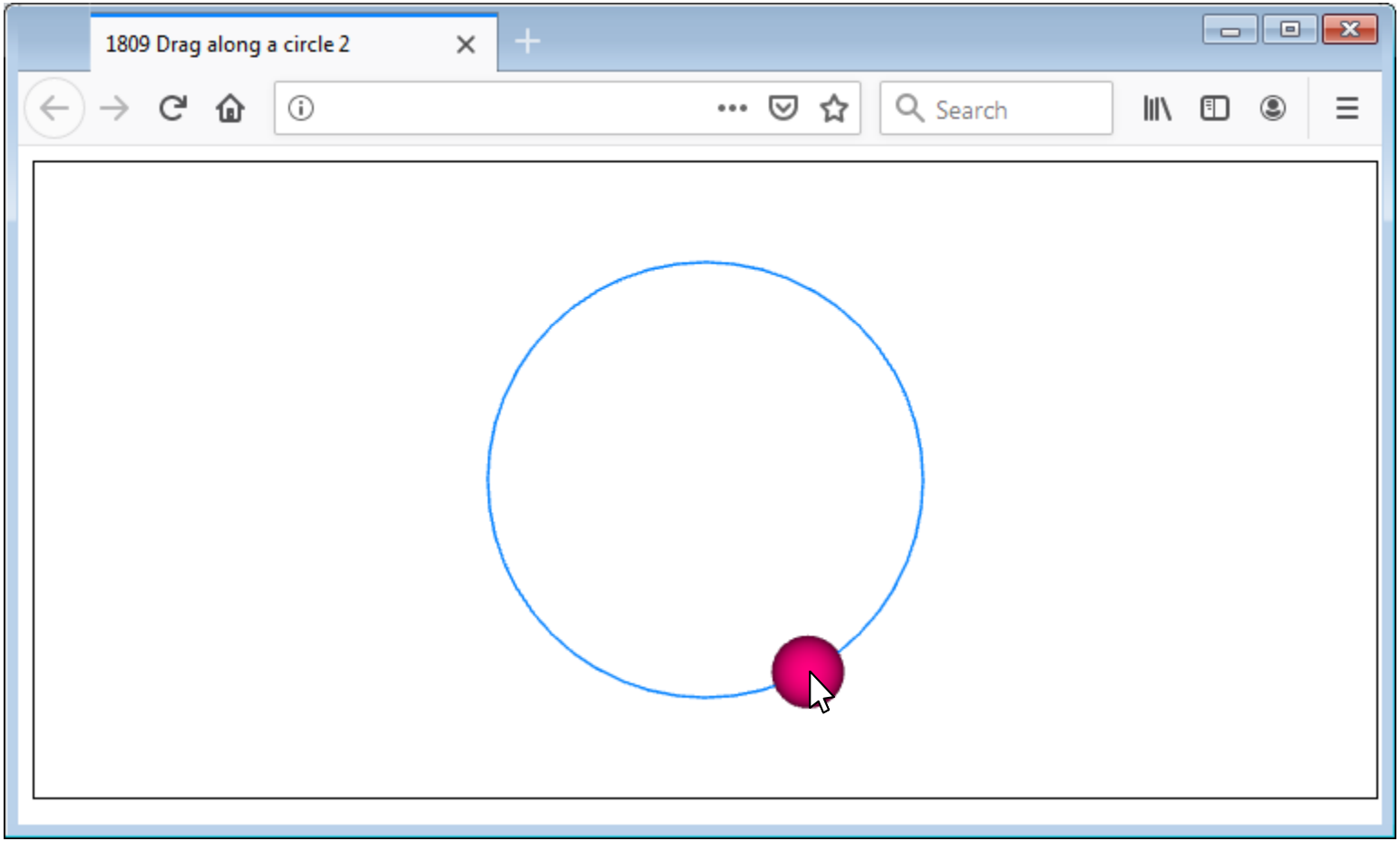


TRY IT

Dragging with the closest point

- Mouse position is scaled so that the distance is 120 (as is the radius of the circle)
- This is the dragged object

```
function mouseMove(event)
{  if (obj)
   {
      var x = ...;
      var y = -(...);
      var d = Math.sqrt(x*x+y*y);
      obj.center[0] = 120*x/d;
      obj.center[1] = 120*y/d;
   }
}
```



TRY IT

Comparison of dragging along a circle

With parameter angle	With closest point
Lost of intuitiveness in some parts of the dragging	Intuitiveness for the whole duration of dragging
Easy adaptation to dragging along an arc	Harder to implement to dragging along an arc
Projection and view point are not important	Prefers orthographic projection and vertical view point
No special points, dragging is the same	Mouse near the circle center makes sharp jumps of dragged object

Other draggings

Dragging of a direction

Dragging of a direction

- Given are n dials, each with a hand
- Random positions and sizes

Goal

- Interactive rotation of selected dial hand

Implementation of dials

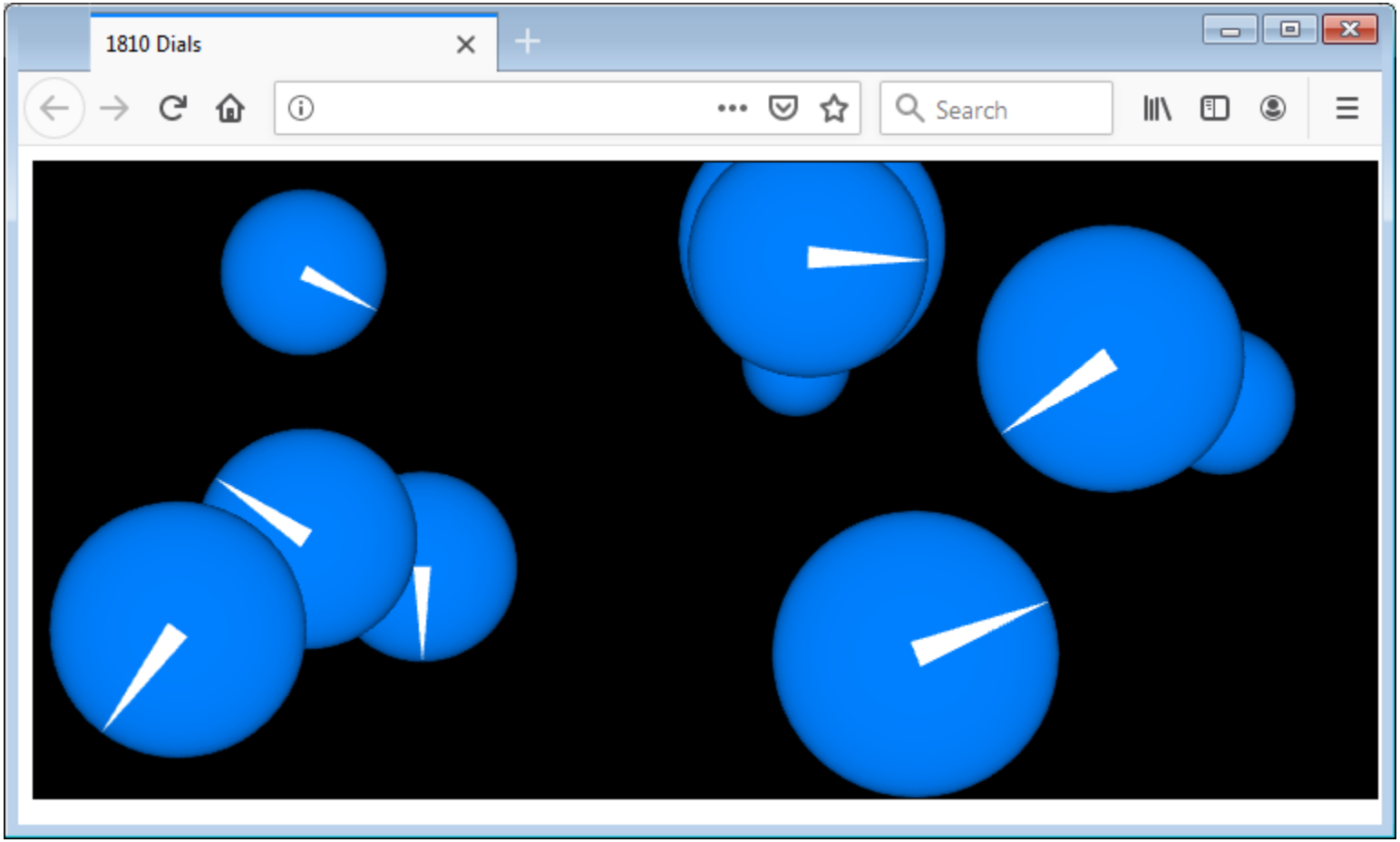
- Flat spheres (Z radius is 20)
- Interactive cone 20 units above each sphere
- Cone not oriented along Z axis, but along Y axis

```
var c = [random(-300,300),random(-150,150),100*i];
var r = random(30,80);
spheroid(c,[r,r,20]).custom({color:[0,0.5,1]});
cone([c[0],c[1],c[2]+20],r/10,r).custom({
   focus:[0,1,0],
   light:false,
   color:[1,1,1],
   interactive:true});
```

Interactivity

- Vector focus of the "dragged" objects is set to the mouse position (x,y)
- This turns the cone towards the mouse

```
function mouseMove(event)
{
   if (obj)
   {
      var x =...;
      var y = -(...);
      obj.focus = [x-obj.center[0],y-obj.center[1],0];
   }
}
```



TRY IT

Dragging of a height

Example

- Regular pyramid
- Horizontal mouse motion rotates the pyramid (i.e. dragging base vertex along a circle)
- Vertical mouse motion changes its height (i.e. dragging top vertex along a vertical segment)

Implementation

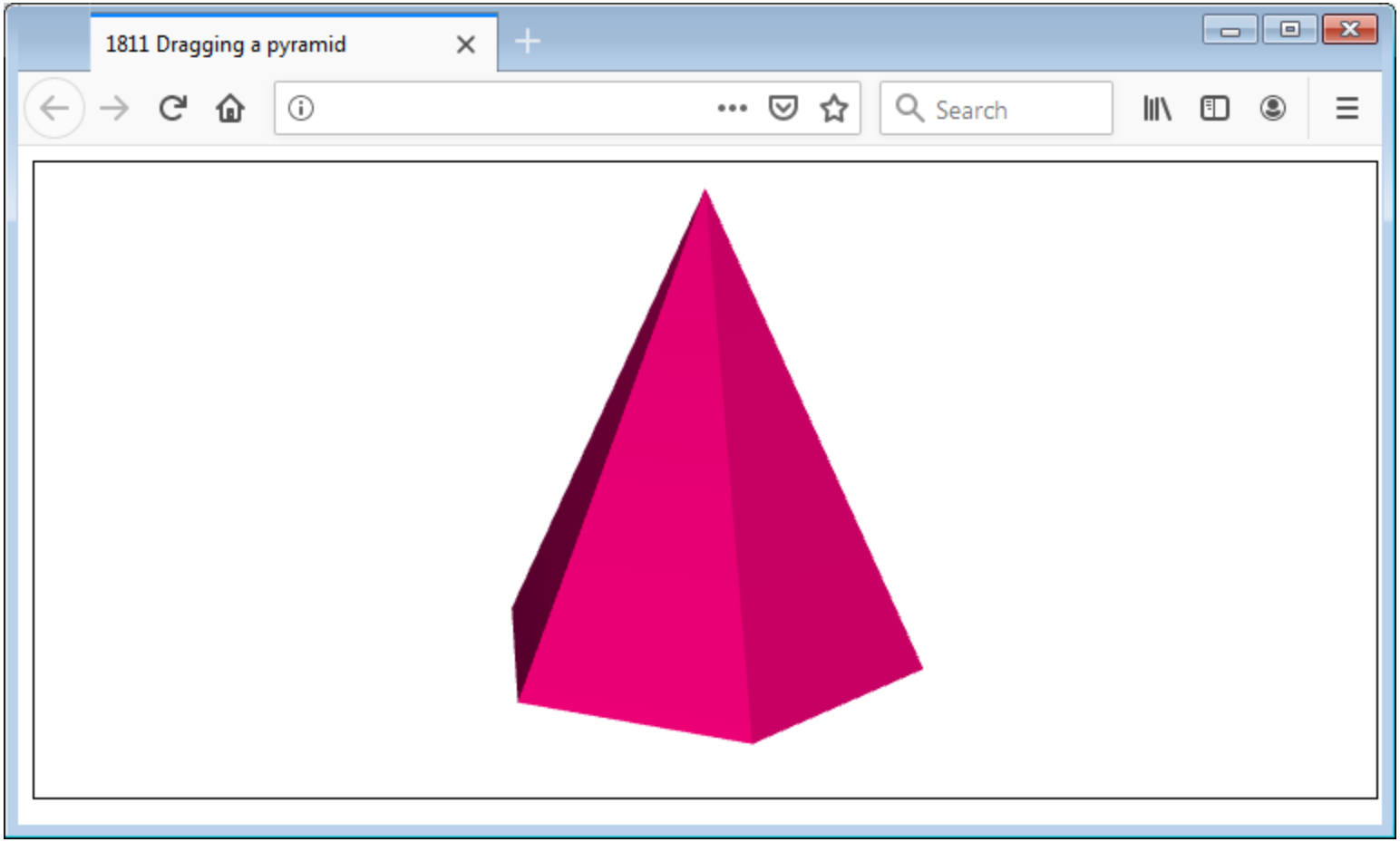
- Mouse offset in dx and dy
- They are used to calculate pyramid spin and height

```
if (obj)
{
   var dx = event.clientX-x;
   var dy = event.clientY-y;

   obj.spin -= dx/100;
   obj.height -= dy/5;
}
```

Restrictions

- Avoiding mixing of horizontal and vertical mouse motion
- Recognizing only the largest motion
- Height limited to 5 at the bottom (physical considerations) and 40 at the top (aesthetical considerations)

```
if (Math.abs(dx)>Math.abs(dy))
   obj.spin -= dx/100;
else
   obj.height -= dy/5;

if (obj.height<5) obj.height=5;
if (obj.height>40) obj.height=40;
```



TRY IT

Minigame

Flying 3D crosses

Rules of the game

- 3D crosses are flying
- All are big and red
- Clicking on a cross makes it white and gradually smaller

Goal

- Clicking on all crosses for minimal time

Implementation of a 3D cross

- Three mutually orthogonal cuboids
- Organized in a group, all groups are in an array
- Every group is interactive and red
- Initially every group size is scaled by 2

```
cross[i] = group( [ cuboid([0,0,0],[10,2,2]),
                    cuboid([0,0,0],[2,10,2]),
                    cuboid([0,0,0],[2,2,10])])
   .custom({
        interactive: true,
        color: [1,0,0.2],
        sizes: [2,2,2]});
```

Additional setting

- Using offset and speed to randomize the objects positions
- Using shrink as a flag whether an object is shrinking
- Property mergeColor makes the whole group unicoloured
- With merge the whole group is made a single object in respect to objectAtPoint – i.e. the whole group will be selected instead of individual cuboids

```
cross[i] = group(...).custom({...,
        offset: random(0,10*Math.PI),
        speed: random(1,2),
        shrink: false});
cross[i].merge();
cross[i].mergeColor();
```

Mouse events

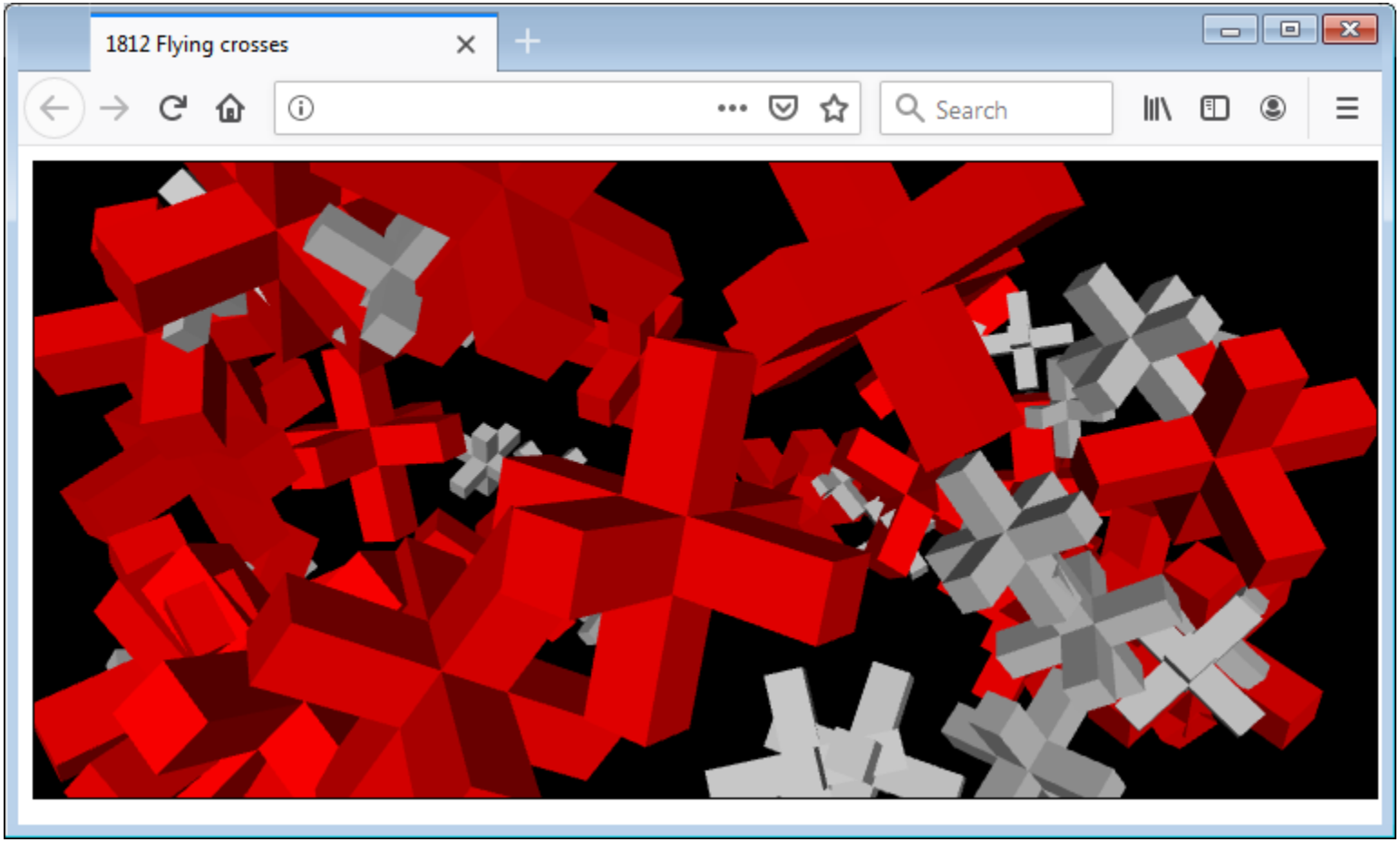
- Using only pressing a button
- If pressing is over a cross, make it white and set shrink flag

```
function mouseDown(event)
{
   obj = p.objectAtPoint(...);
   if (obj)
   {
      obj.color = [0.8,0.8,0.8];
      obj.shrink = true;
   }
}
```

Main animation loop

- Every cross has own time t
- Using time to calculate center, orientation and spin
- When shrinking, the size decreases with a linear combination between the current size and size 1

```
t = Suica.time/3*cross[i].speed+cross[i].offset;
cross[i].center = [40*cos(t),40*sin(t),15*cos(1.5*t+i)];
cross[i].focus = [cos(2*t),sin(1.3*t),sin(-1.5*t)];
cross[i].spin = (t-i)*cross[i].speed;
if (cross[i].shrink)
{
   var s = cross[i].sizes[0]*0.9+0.1*1;
   cross[i].sizes = [s,s,s];
}
```



TRY IT

Summary

Bound drag and drop

Restrictions

- Objective – e.g. shape of the trajectory defines the available drag positions
- Subjective – e.g. some drag positions are avoided due to aesthetical reasons

Implementation

- Restricting input parameters
- Restricting result coordinates

Dragging along a line

- With a linear combination of two points
- With the closest point
- A line could be straight or circular

Other types

- Dragging non-coordinate values (orientation, spin, height, etc.)

The end

Comments, questions