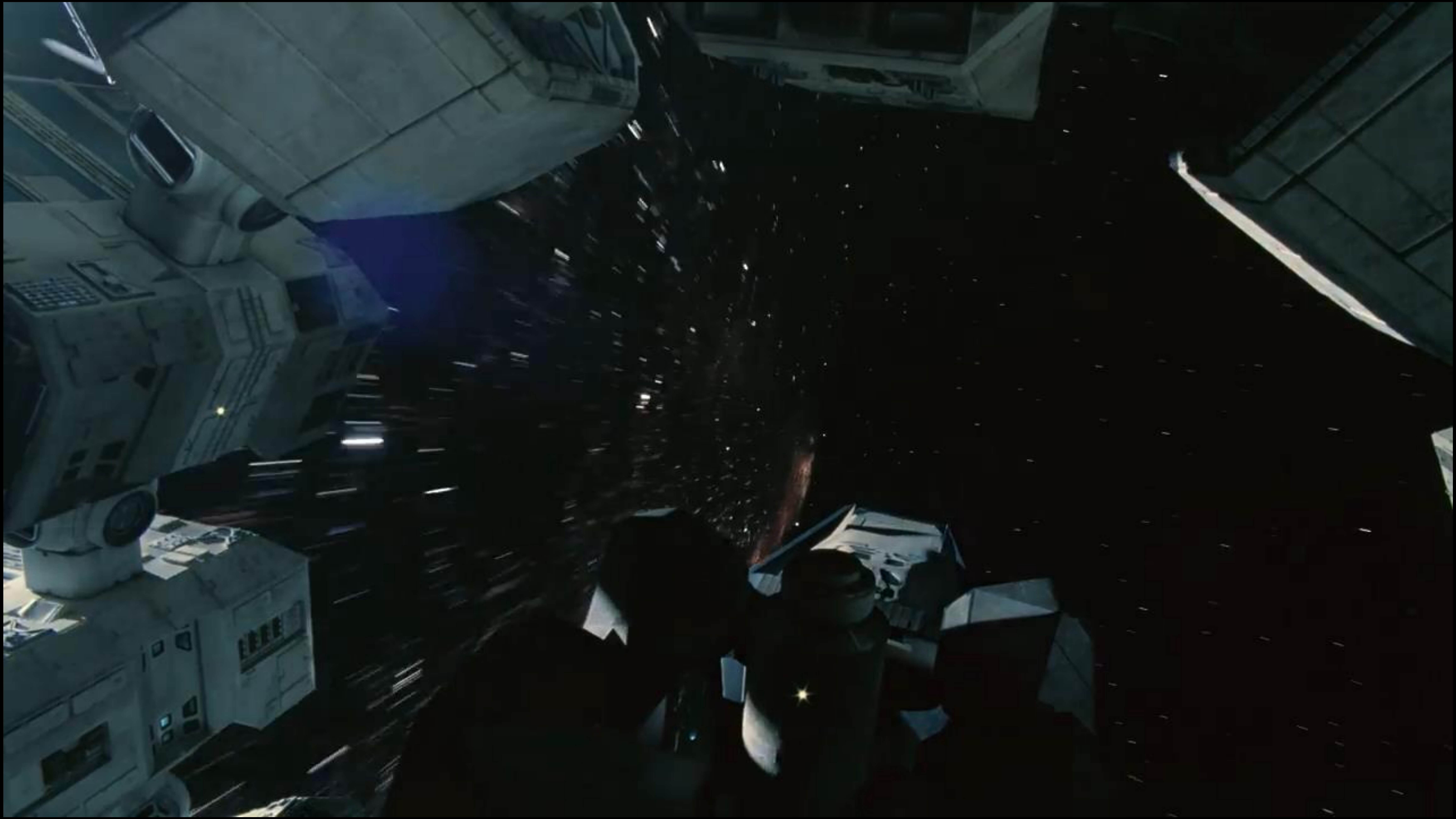
CS3451:
Particles

Bo Zhu
School of Interactive Computing
Georgia Institute of Technology

Image Credit: Ben Newhall

Motivational Video: Ben Newhall's Gravitational Lensing of Black Holes



Motivational Video: Interstellar

Motivational Video: Frozen

https://www.youtube.com/watch?v=YVVTZgwYwVo

Motivational Video: Blender Fireworks

Motivational Video:

Unity Game VFX Demoreel

https://www.youtube.com/watch?v=SuSdcq54mho

Study Plan

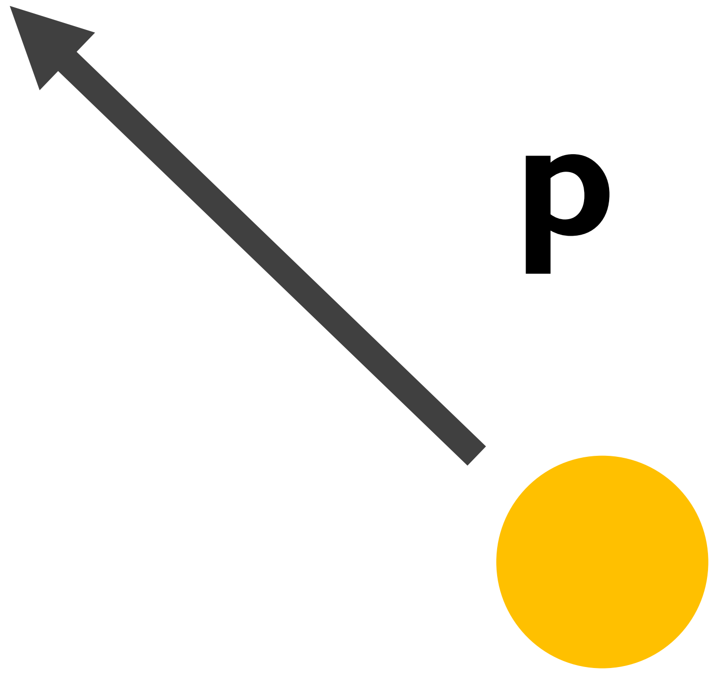
- Particle Motion
- Particle Force
- Particle System on CPU
- Particle System on GPU
- Firework Implementation

Particle Motion

Particle

Attributes

Position: **x**
Velocity: **v**
Force: **f**
Mass: m
Color: **c**
...



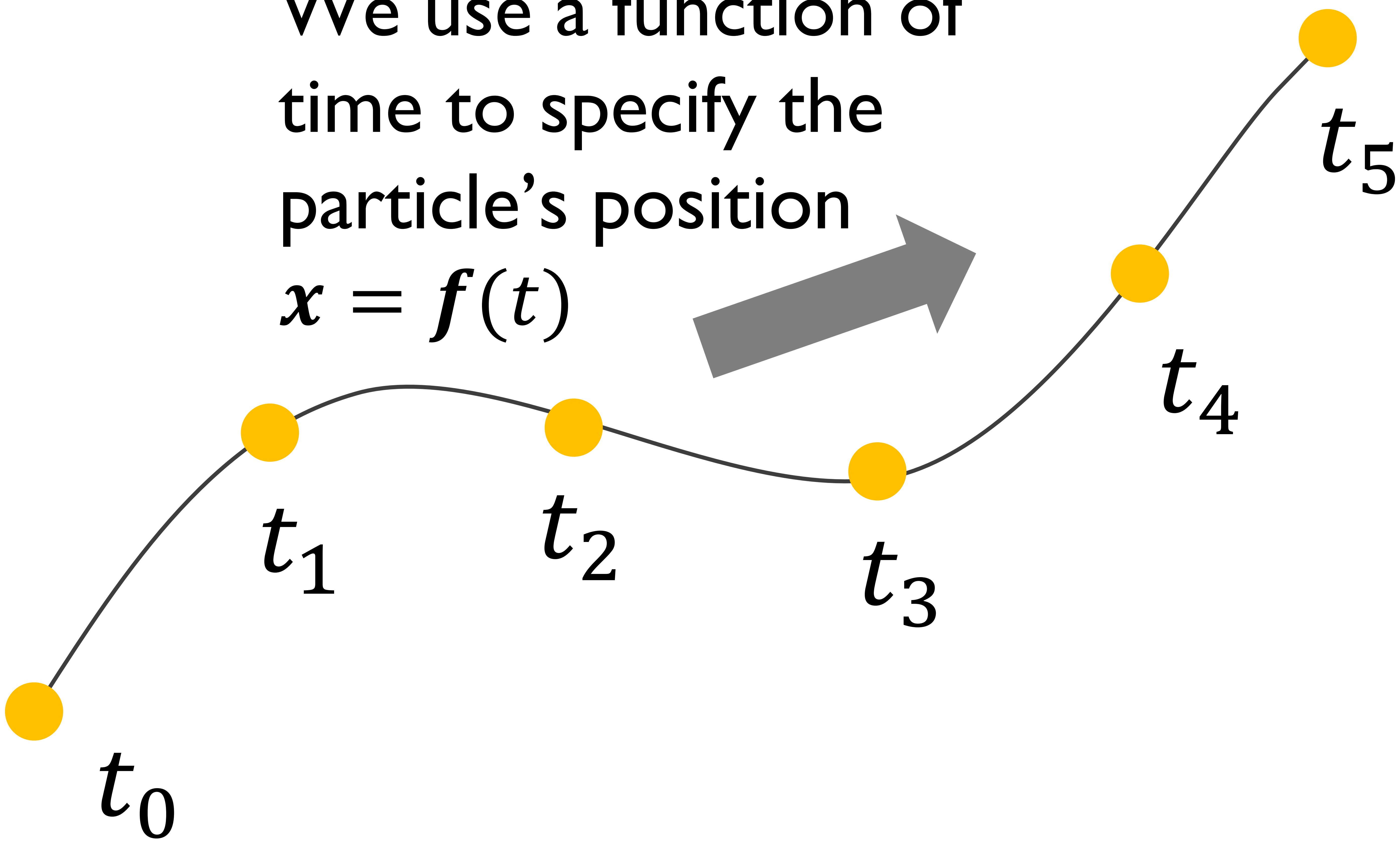
Particle

A point that can move in space according to the external forces

Each particle carries a set of attributes that controls its appearance and dynamics

Trajectory

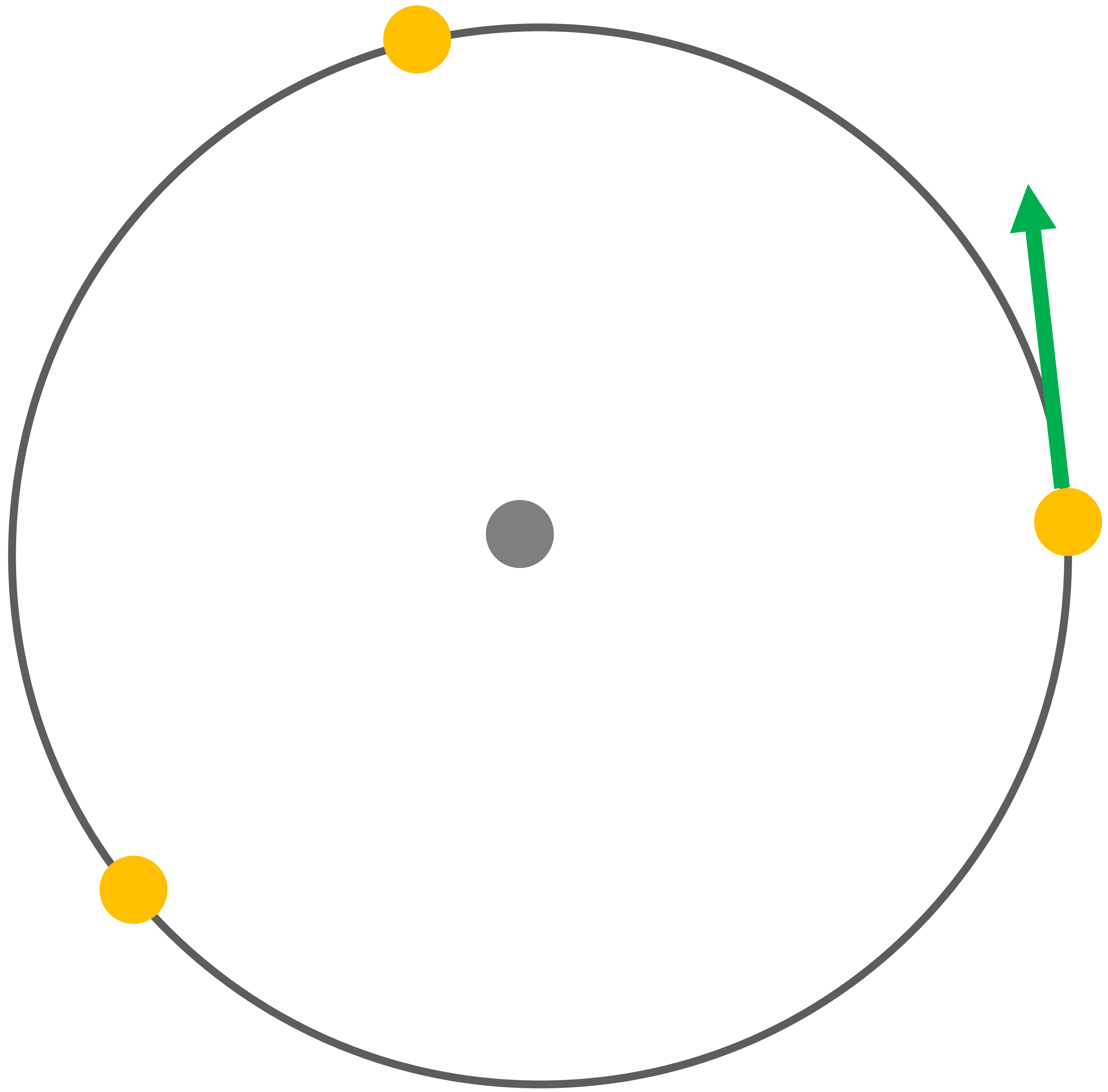
We use a function of time to specify the particle's position $\boldsymbol{x} = \boldsymbol{f}(t)$



A particle **trajectory** is the path that a simulated particle follows through three-dimensional space over time, typically due to various forces and influences applied to it

How do we specify a particle's trajectory in program?

First idea: define an analytic curve for each particle

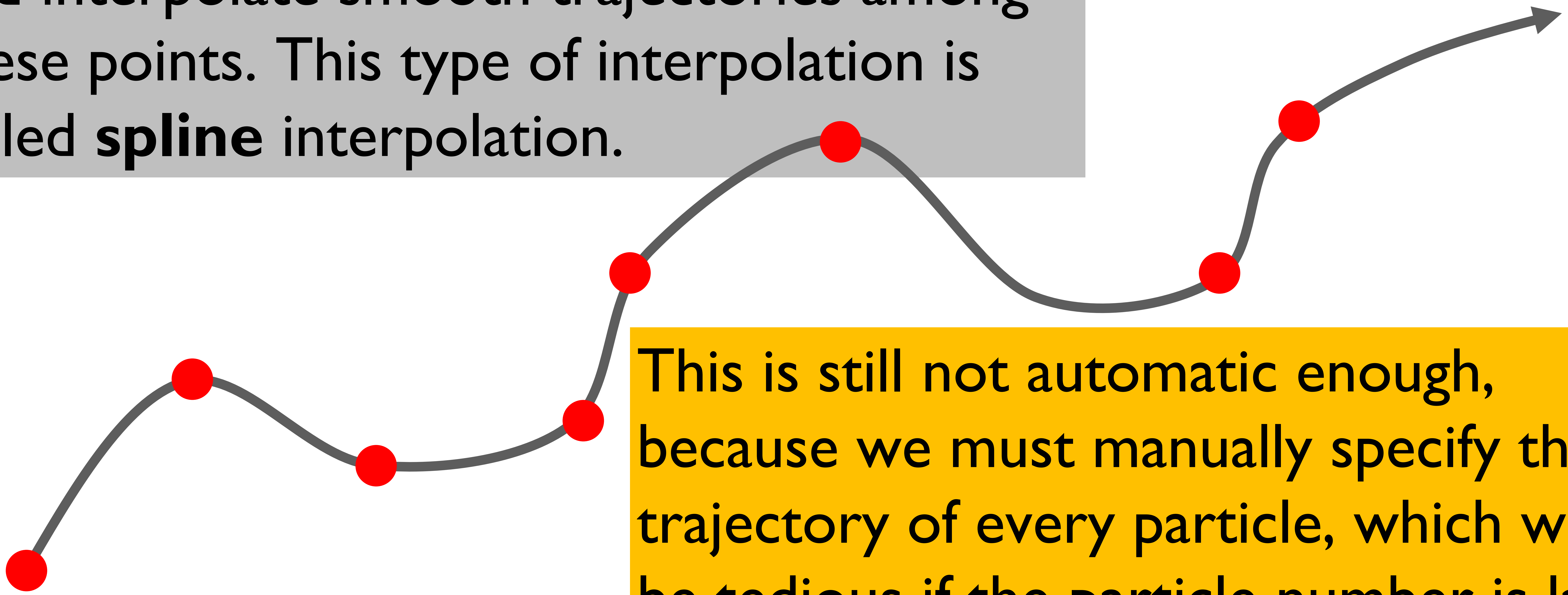


$$x = rcos(\omega t)$$
$$y = rsin(\omega t)$$

This woks for simple trajectories with analytic expressions. But how about more complicated cases?

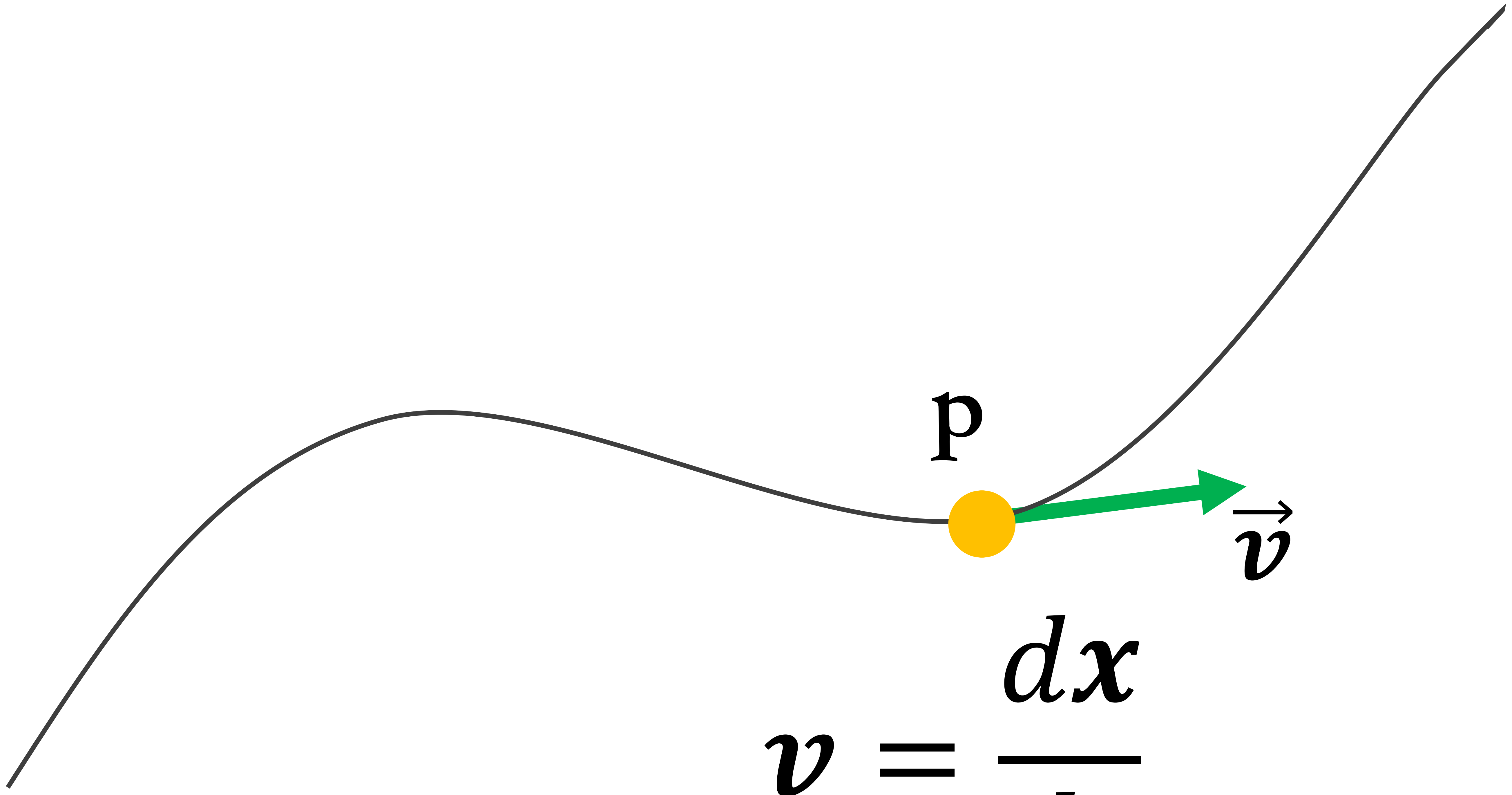
E.g., how about an arbitrary trajectory like this?

We can specify a number of "control points" and interpolate smooth trajectories among these points. This type of interpolation is called **spline** interpolation.



This is still not automatic enough, because we must manually specify the trajectory of every particle, which will be tedious if the particle number is large (and animation time is long)

A better idea: use the particle's velocity



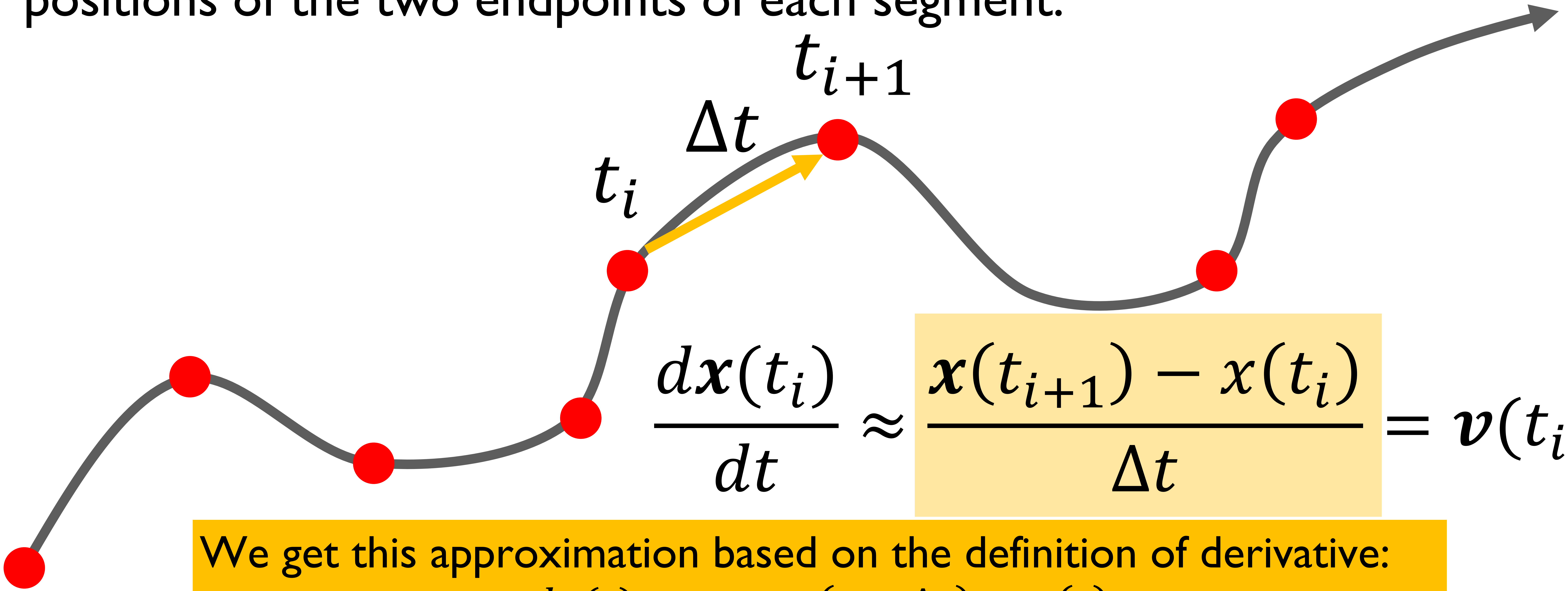
$$\boldsymbol{v} = \frac{d\boldsymbol{x}}{dt}$$

For a particle, if we know its velocity for time t as $\boldsymbol{v}(t)$, we can calculate its position by solving the equation

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{v}(t)$$

Numerical Differentiation

- **Key Idea:** Discretize the trajectory into many small segments and approximate the velocity by calculating the difference between the positions of the two endpoints of each segment.



$$\frac{d\boldsymbol{x}(t_i)}{dt} \approx \boxed{\frac{\boldsymbol{x}(t_{i+1}) - x(t_i)}{\Delta t}} = \boldsymbol{v}(t_i)$$

We get this approximation based on the definition of derivative:

$$\frac{dx(t)}{dt} = \lim_{\Delta t \to 0} \frac{x(t+\Delta t) - x(t)}{\Delta t}$$

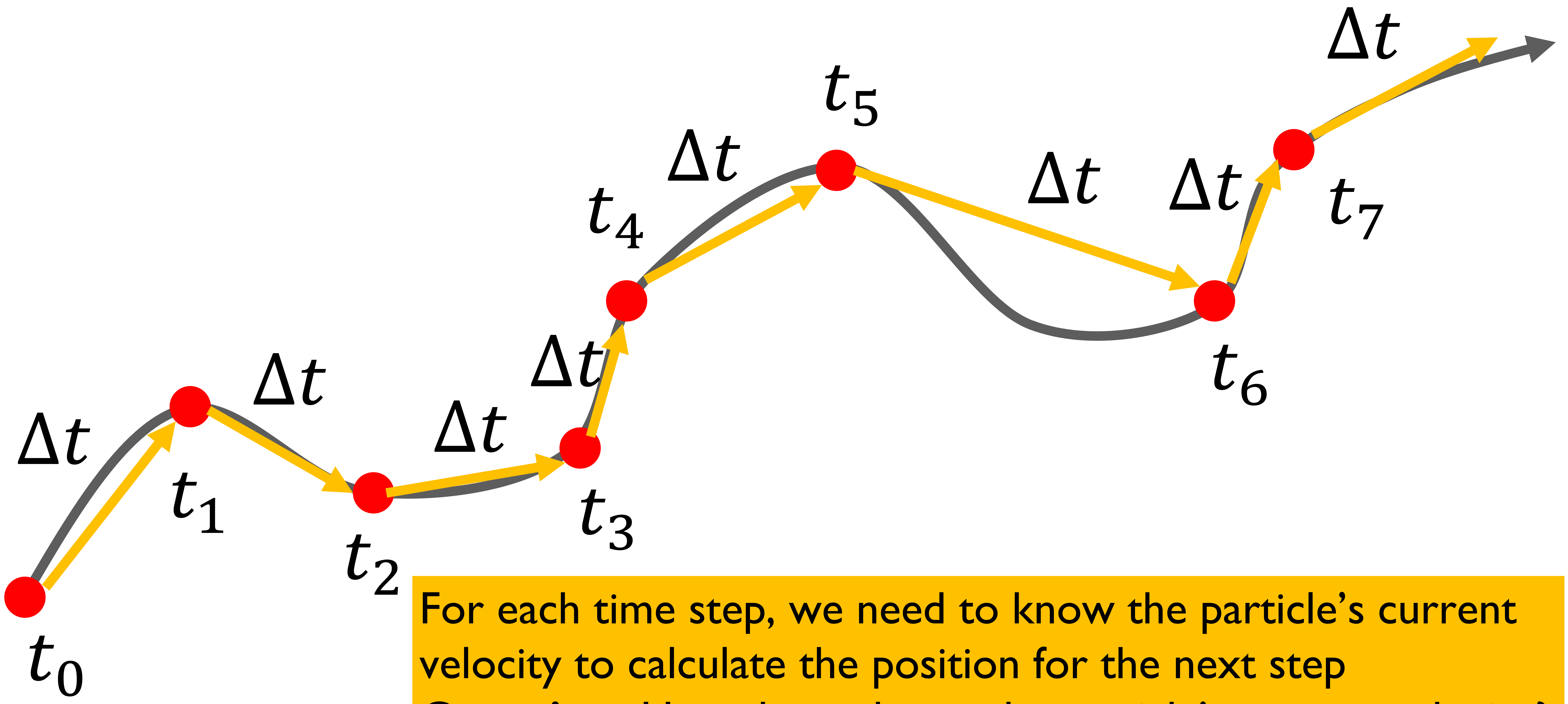
Numerical Integration

Based on $\frac{d\boldsymbol{x}(t_i)}{dt} \approx \frac{\boldsymbol{x}(t_{i+1})-\boldsymbol{x}(t_i)}{\Delta t} = \boldsymbol{v}(t_i)$

we can calculate the particle's position for the next time step as

$$\underbrace{\boldsymbol{x}(t_{i+1})}_{\text{Position for the next timestep}} = \underbrace{\boldsymbol{x}(t_i)}_{\text{Position of the current timestep}} + \underbrace{\boldsymbol{v}(t_i)}_{\text{Velocity of the current timestep}}\Delta \mathrm{t}$$

Calculate the particle's trajectory by conducting time integration step by step



For each time step, we need to know the particle's current velocity to calculate the position for the next step
Question: How do we know the particle's current velocity?

We have two options to update velocity

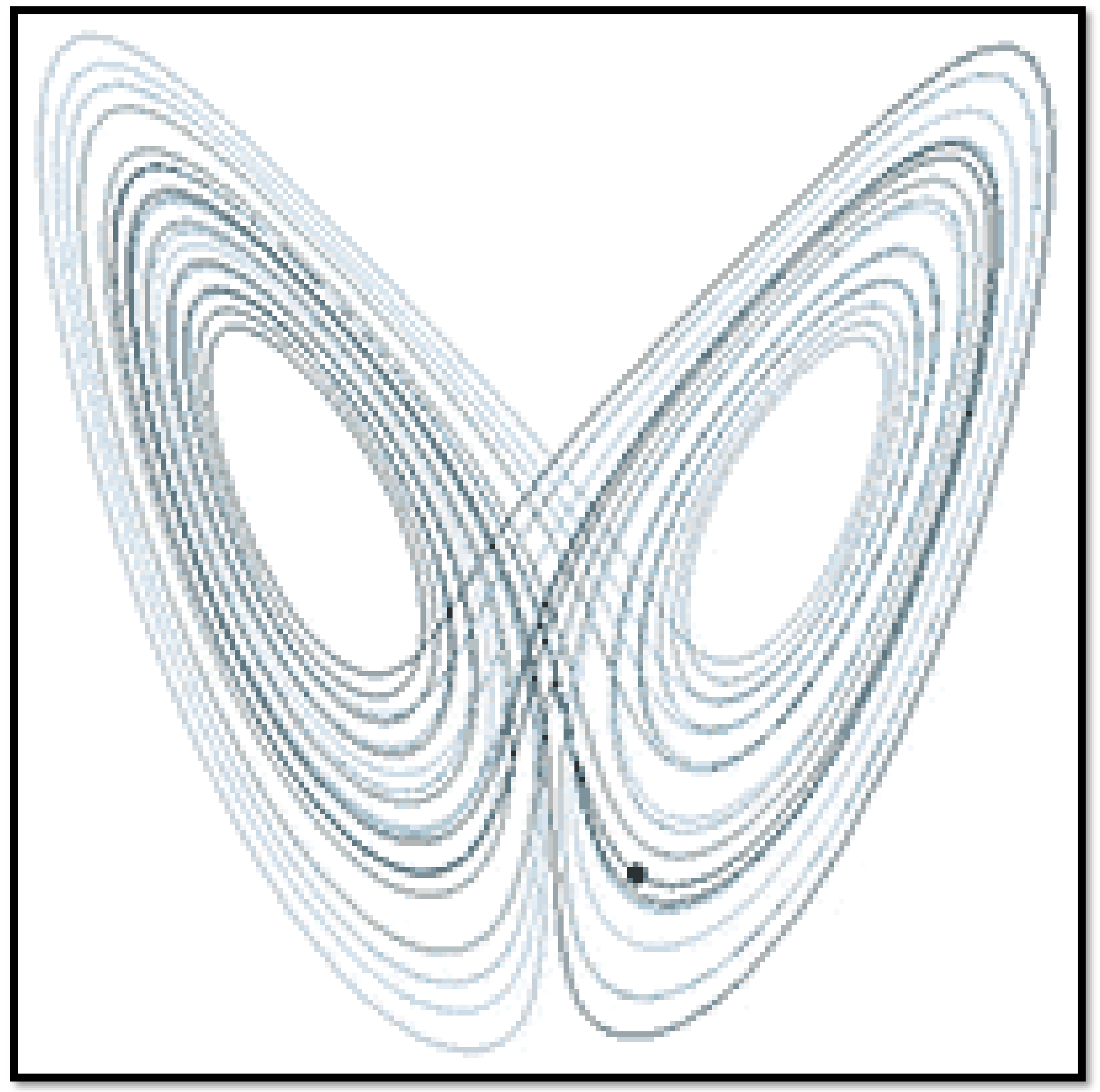
- Option I: directly update the particle's velocity based on its position

Example:
Chaotic system

$$\frac{\mathrm{d}x}{\mathrm{d}t} = \sigma(y - x),$$

$$\frac{\mathrm{d}y}{\mathrm{d}t} = x(\rho - z) - y,$$

$$\frac{\mathrm{d}z}{\mathrm{d}t} = xy - \beta z.$$

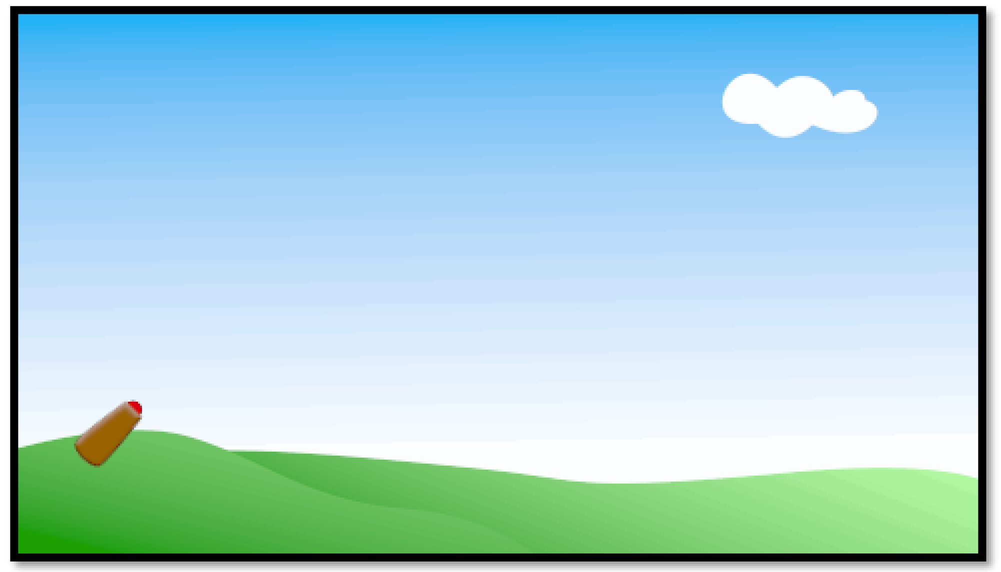


- Option II: update the particle's force first, and then update its velocity based on Newton's law

Example:
Parabolic shot

$$\frac{d\boldsymbol{u}(t)}{dt} = \boldsymbol{g}$$

$$\frac{d\boldsymbol{x}(t)}{dt} = \boldsymbol{u}(t)$$



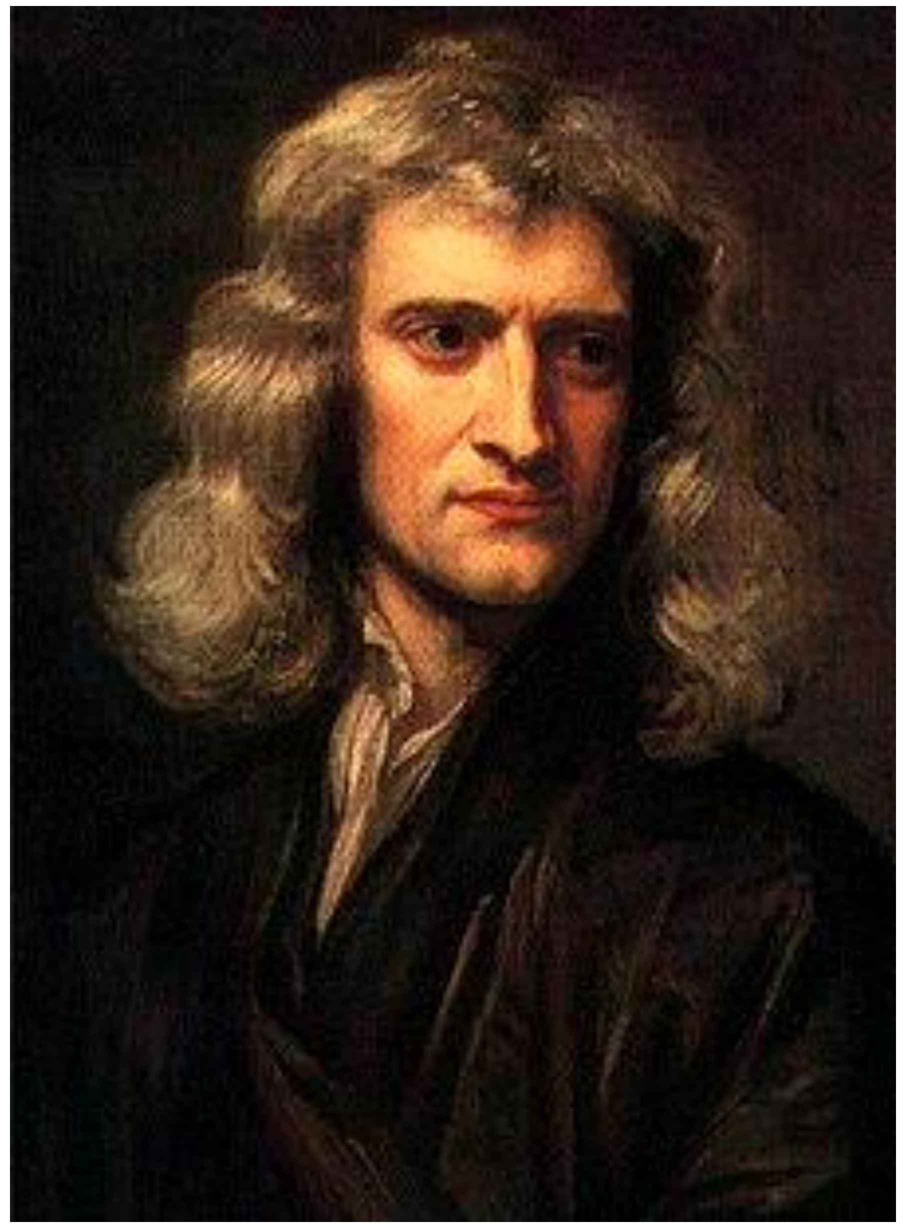
Particle Force

Newton's Second Law

$$\boldsymbol{f} = m\boldsymbol{a}$$

$$\boldsymbol{a} = \frac{d\boldsymbol{u}}{dt}$$

$$\frac{d\boldsymbol{u}(t)}{dt} = \boldsymbol{f}(t)/m$$



$$\frac{d\boldsymbol{u}(t_i)}{dt} \approx \frac{u(t_{i+1}) - u(t_i)}{\Delta t} = \boldsymbol{f}(t_i)/m$$

$$\boldsymbol{u}(t_{i+1}) = \frac{\boldsymbol{f}(t_i)}{m}\Delta t + \boldsymbol{u}(t_i)$$

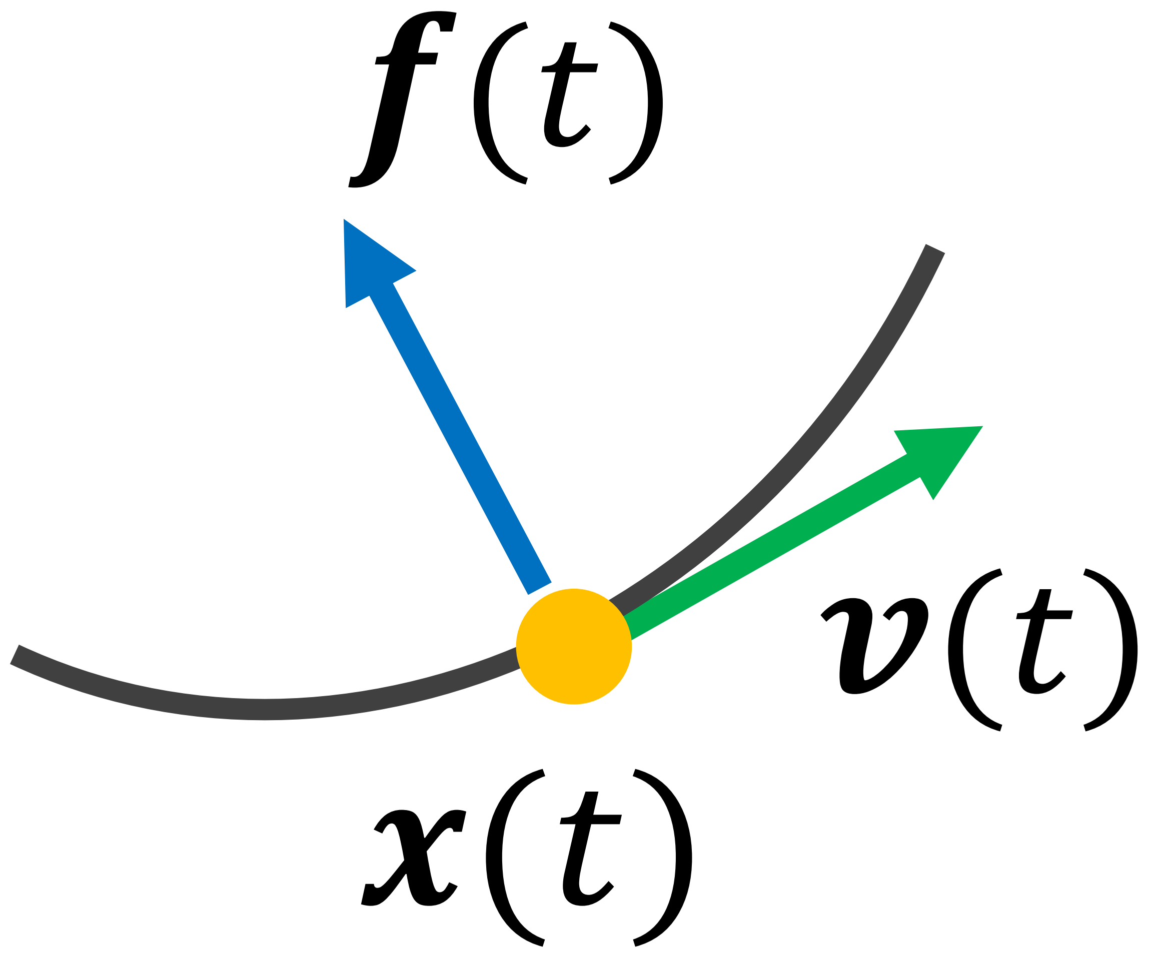
The problem becomes specifying the force applied on the particle for each time step

We can apply the same idea of time integration here to calculate velocity of the next time step based on the velocity of the current time step!

What kind of forces can we apply on a particle?

$$\boldsymbol{f}(\boldsymbol{x}, \boldsymbol{v}, t)$$

Position Velocity Time



<u>Examples of different forces</u>

- Constant forces (e.g. gravity)
- Time dependent forces (e.g. wind)
- Position dependent forces (e.g. magnetic force)
- Velocity dependent forces (e.g. drag, friction)
- Position & velocity dependent forces (e.g. springs)

Force Example: Gravity

$$f = mg$$

- The force is constant over space and time
- The trajectory is simple ballistic motion

Force Example: Wind

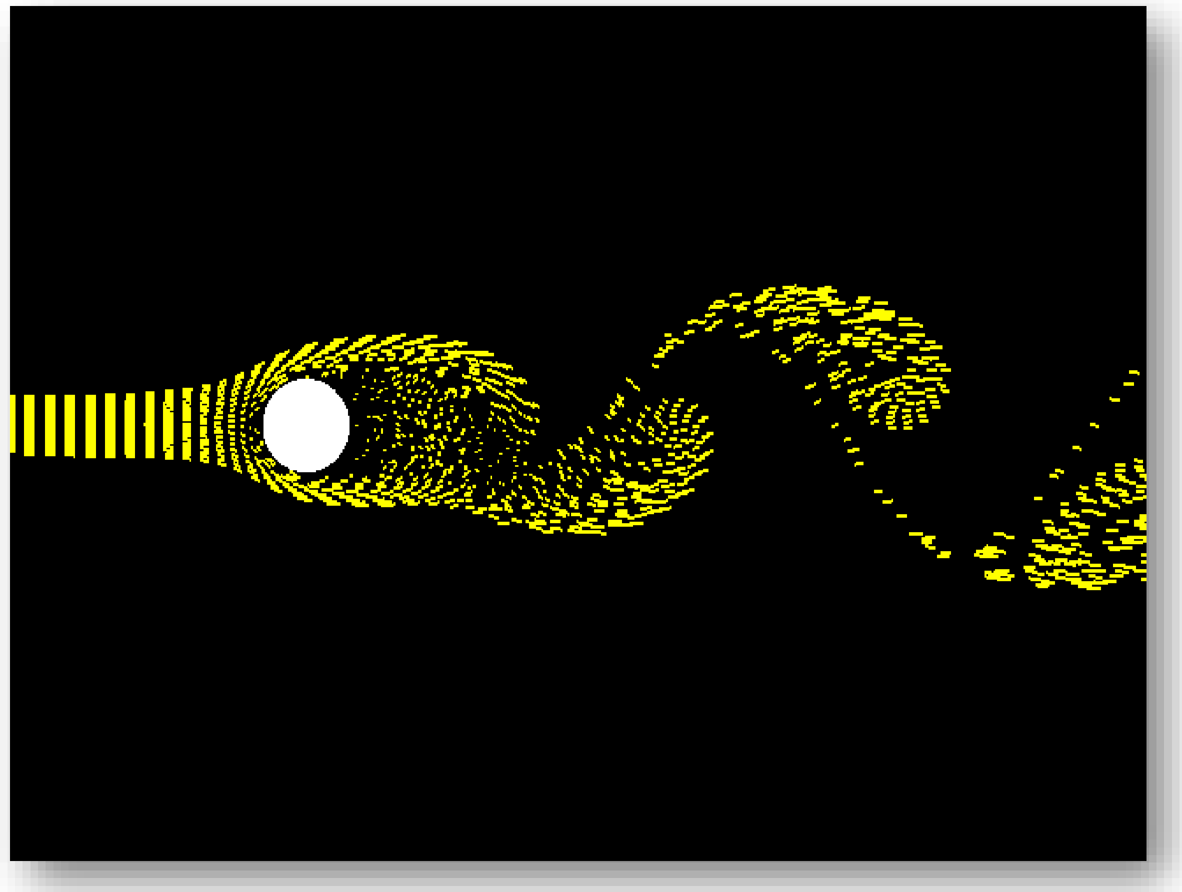
$$\boldsymbol{f}_{wind} = \boldsymbol{f}(\boldsymbol{x}, t)$$

- Wind force depends on position and time
- The particle trajectory is a curve controlled by the force field

Force Example: Drag

$$f_{drag} = f(v)$$

- Velocity dependent force (linear in velocity)
- The faster the velocity, the larger the drag
  - Think molasses or honey

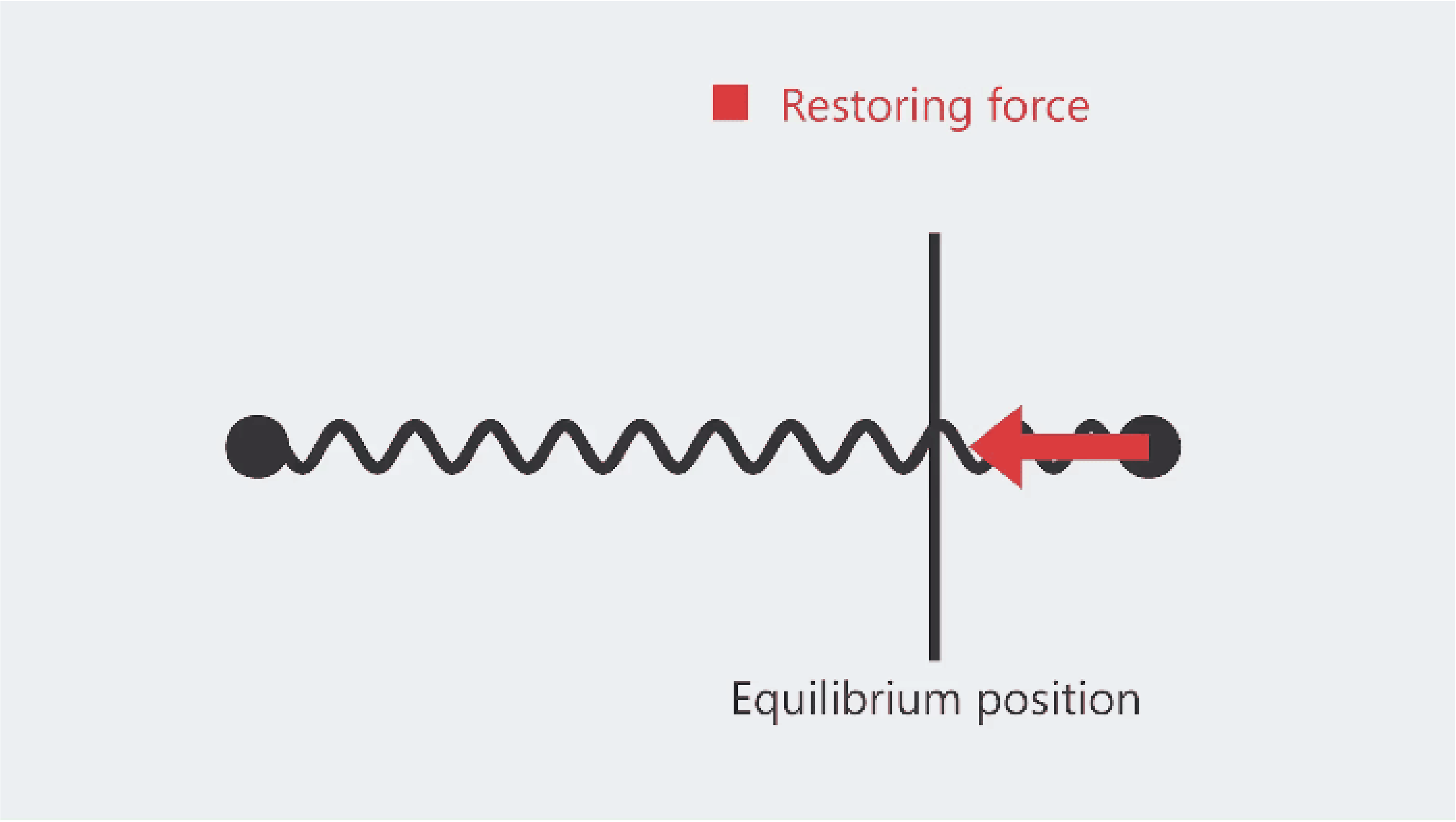


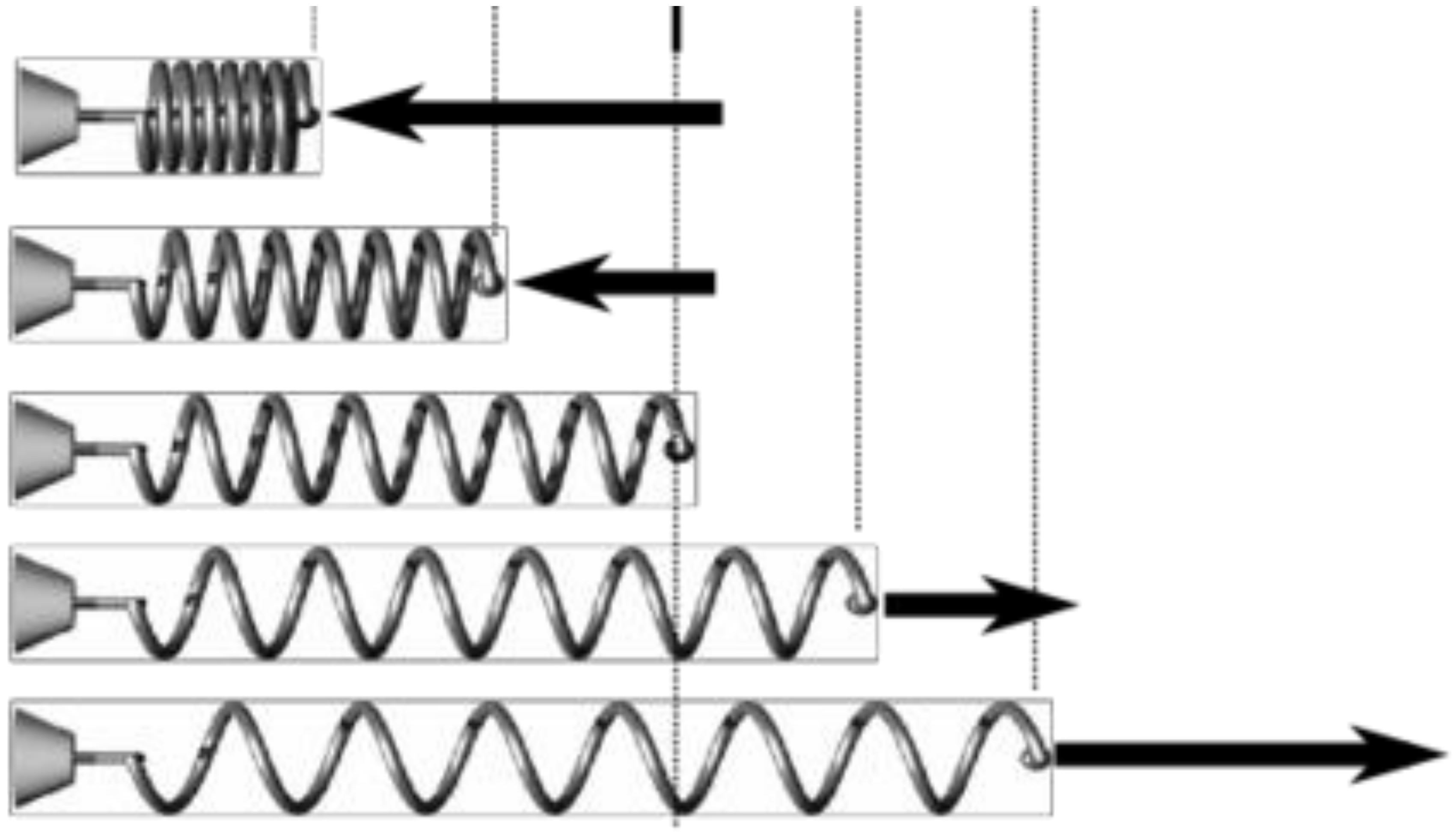
$$\text{e.g., } f_{drag} = -|v|^2 v$$

Force Example: Spring

$$f_{spring} = -kx$$

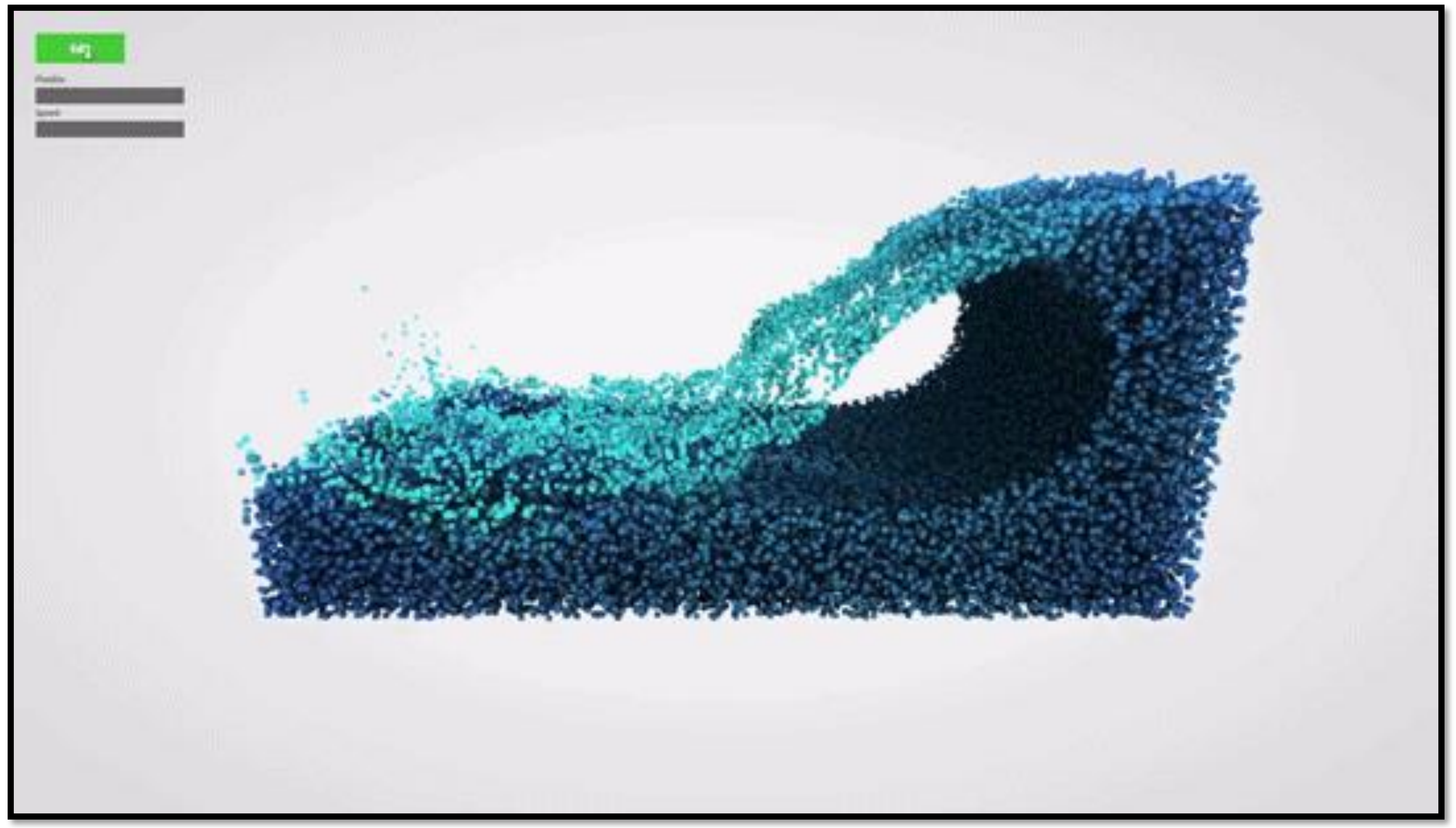
- Spring force depends on the particle's position only
  - The further the particle is away from the origin, the larger the force (Hooke's Law)





More Particle Force Examples

- A particle system that simulates **water** requires gravity as an external force as well as internal forces among particles to model fluid flow
- A particle system that simulates **fish school** or **bird flock** needs to model attractive forces for collective motion and repulsive forces to avoid collisions



Fluid particles

Bird particles

Let's put everything together

- For a single particle, if we know how to calculate the force applied on it for each time step, then we can update its position as:

(I) Update the particle's **velocity**:

$$\boldsymbol{u}(t_{i+1}) = \frac{\boldsymbol{f}(t_i)}{m}\Delta t + \boldsymbol{u}(t_i)$$

(II) Update the particle's **position**:

$$\boldsymbol{x}(t_{i+1}) = \boldsymbol{x}(t_i) + \boldsymbol{v}(t_i)\Delta \mathrm{t}$$

We call these updates a **particle simulator**

Particle Simulation Loop

Time **n**

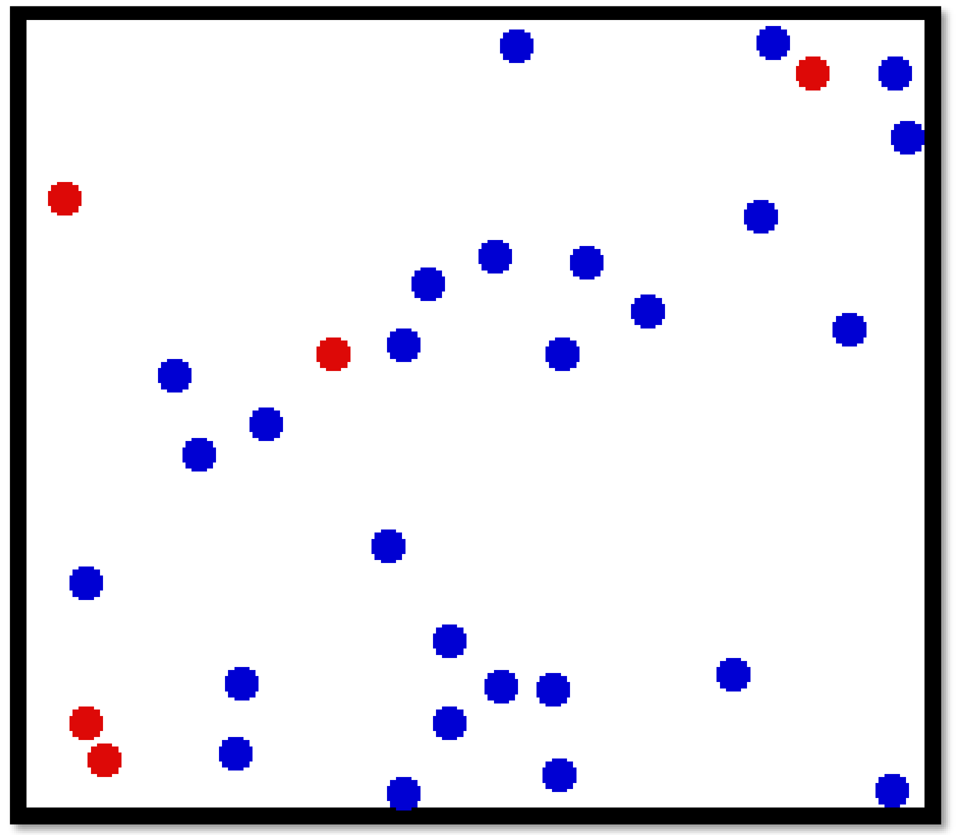
Update Particle Appearance

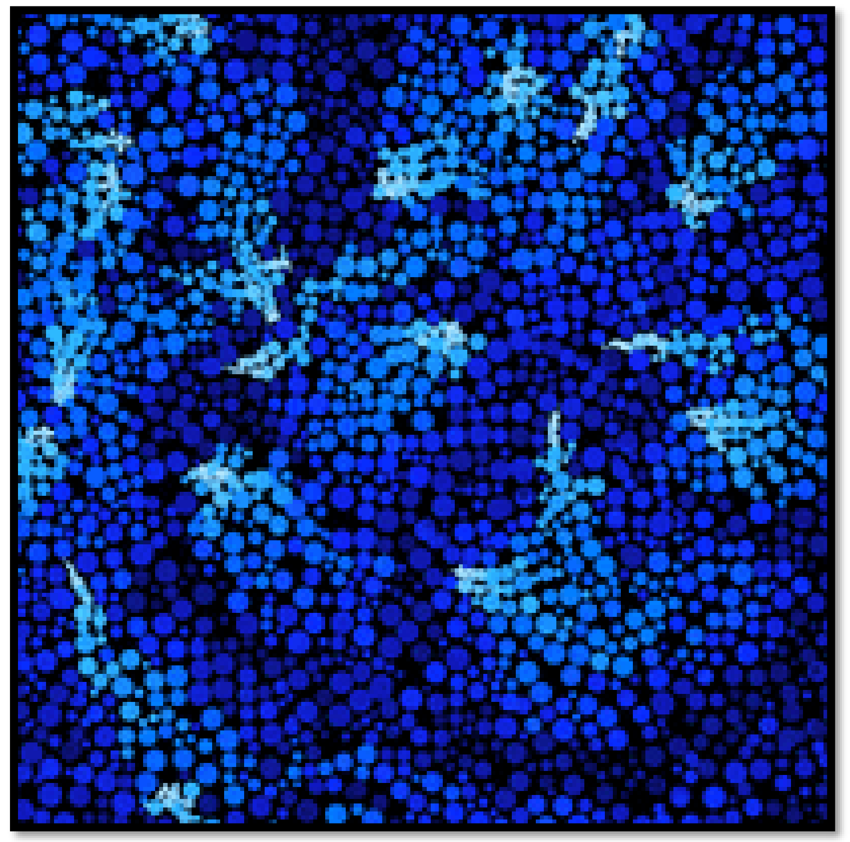
Update Particle Velocity

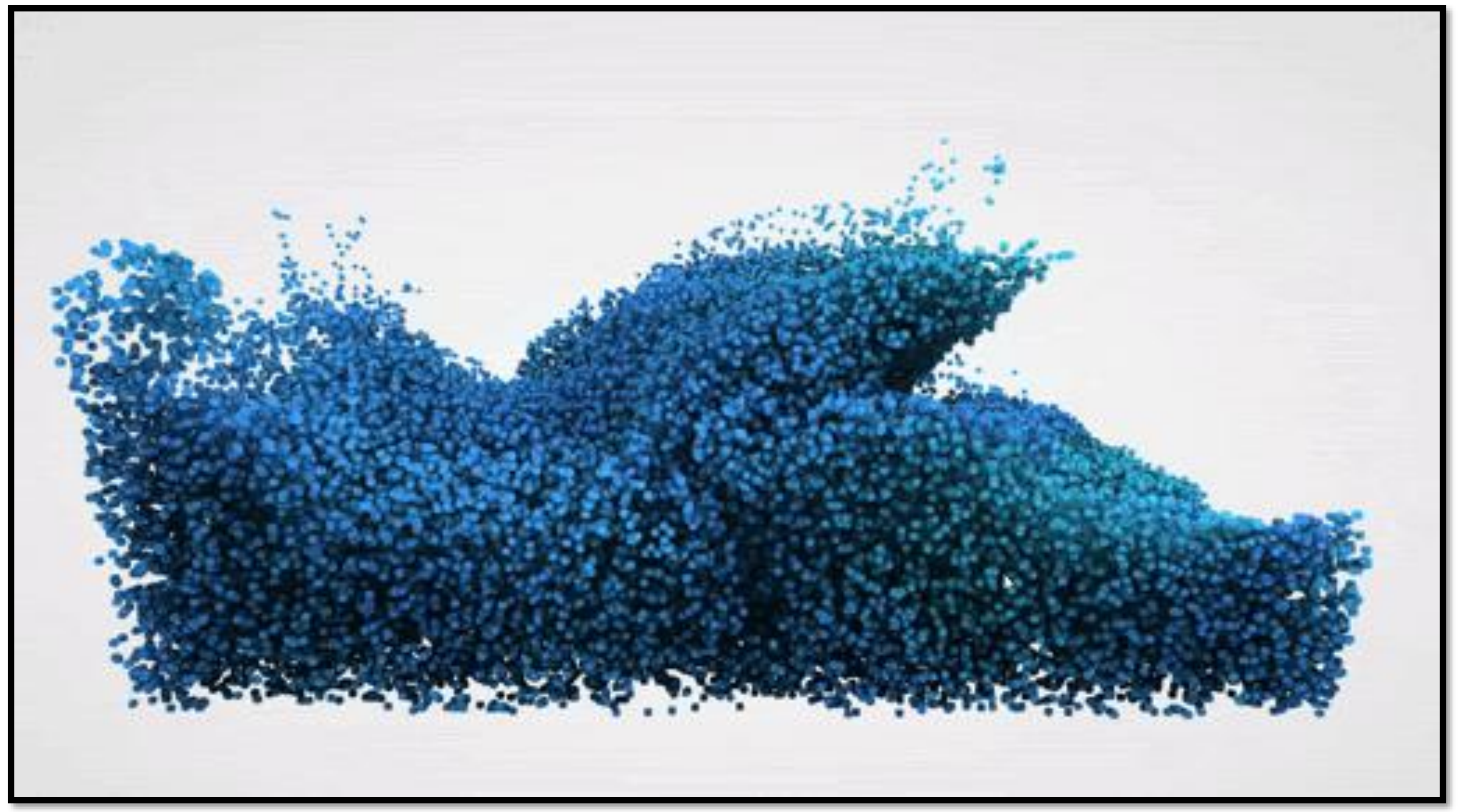
Update Particle Position

Time **n+1**

n++







Key Takeaway for Particle System Simulation

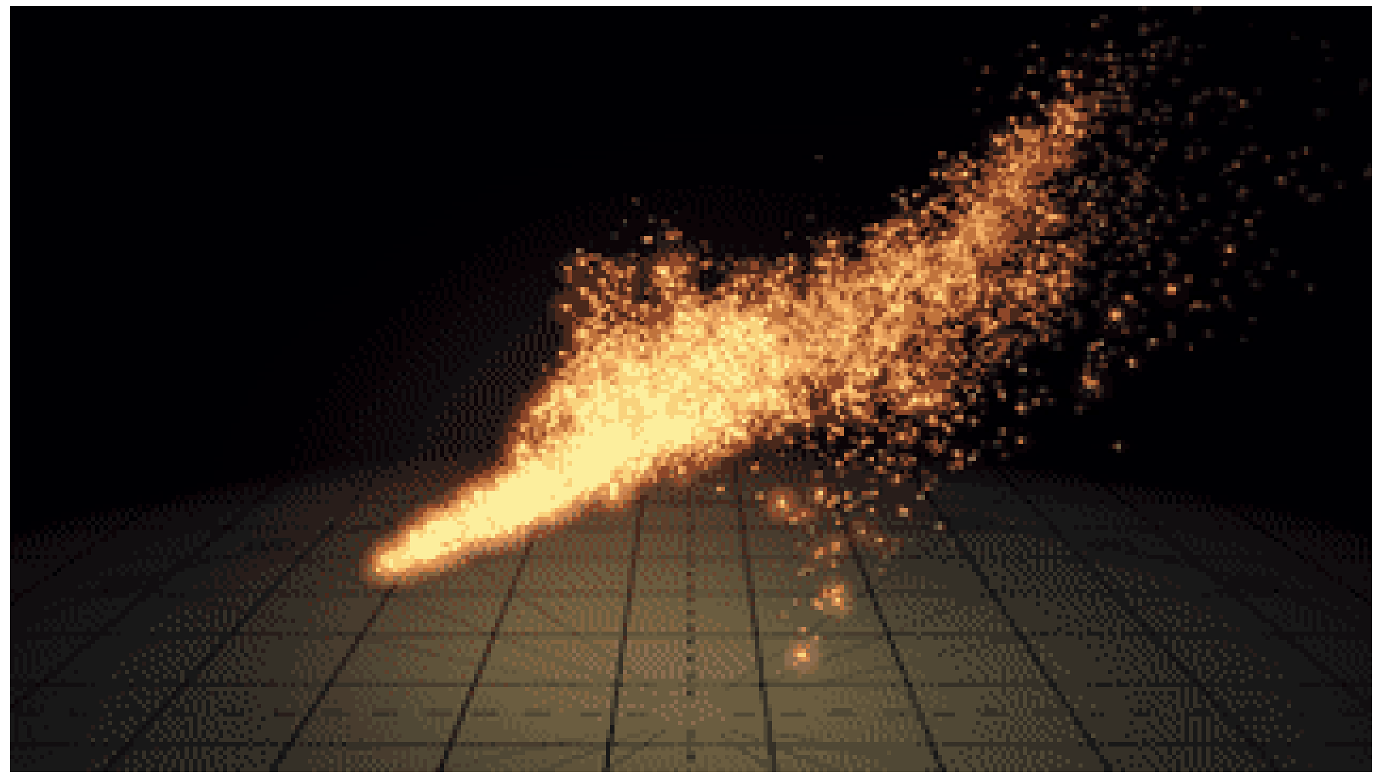
- Three ways to simulate particle trajectory
  - Calculate each particle's **position** with a function of time t
  - Calculate each particle's **velocity** and then update its position
  - Calculate each particle's **force**, then velocity, then position

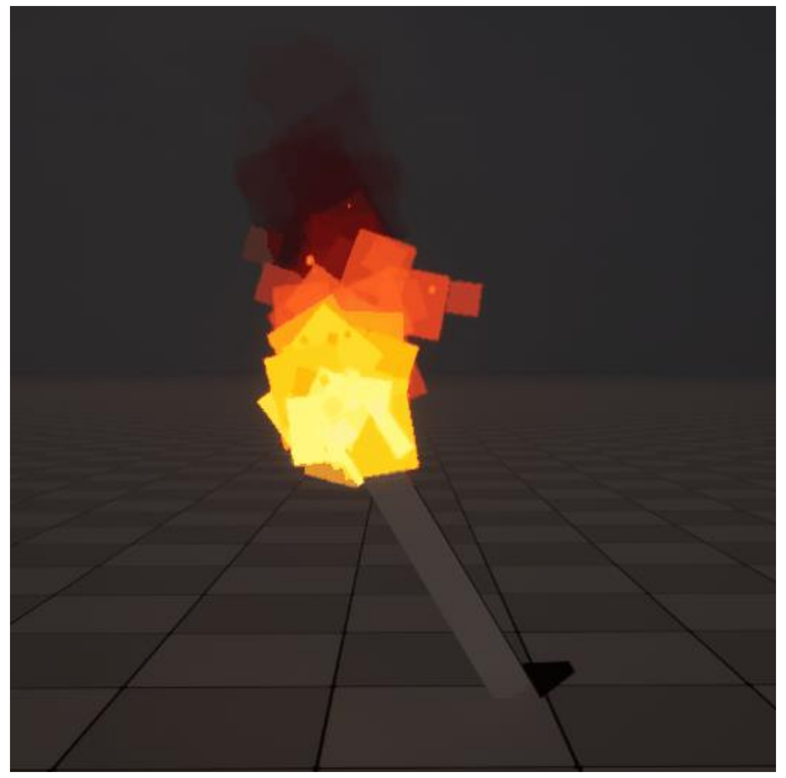
Notice: The analytical function method is usually used for simulating particles directly on GPU with GLSL shaders (in which it is difficult to access memory and change variable values), while the velocity and force methods are used to simulate particles on CPU with complex dynamics

In A8, we will practice the analytical function method with GLSL shader

Particle Emitter

- A particle usually comes with a 'life' number that ticks down after it pops into existence.
- If its life drops too low, we zap the particle out of existence and queue up a new one to take its place when it's time for another to show up.
- There's a particle emitter that's the boss of all the particles it spits out (let's call it the **boss particle**) controlling the initial state of the emission.

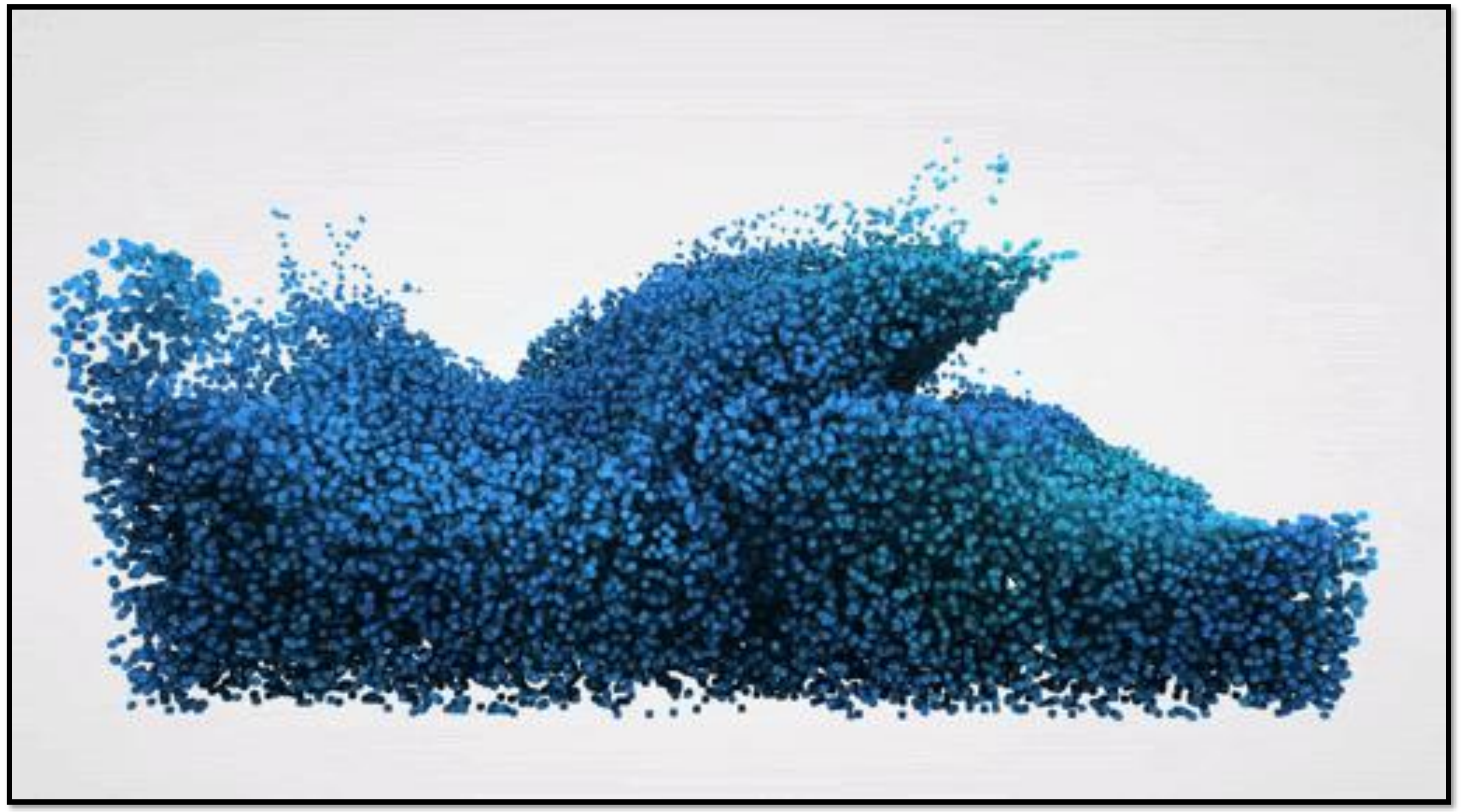




How do we implement a particle system?

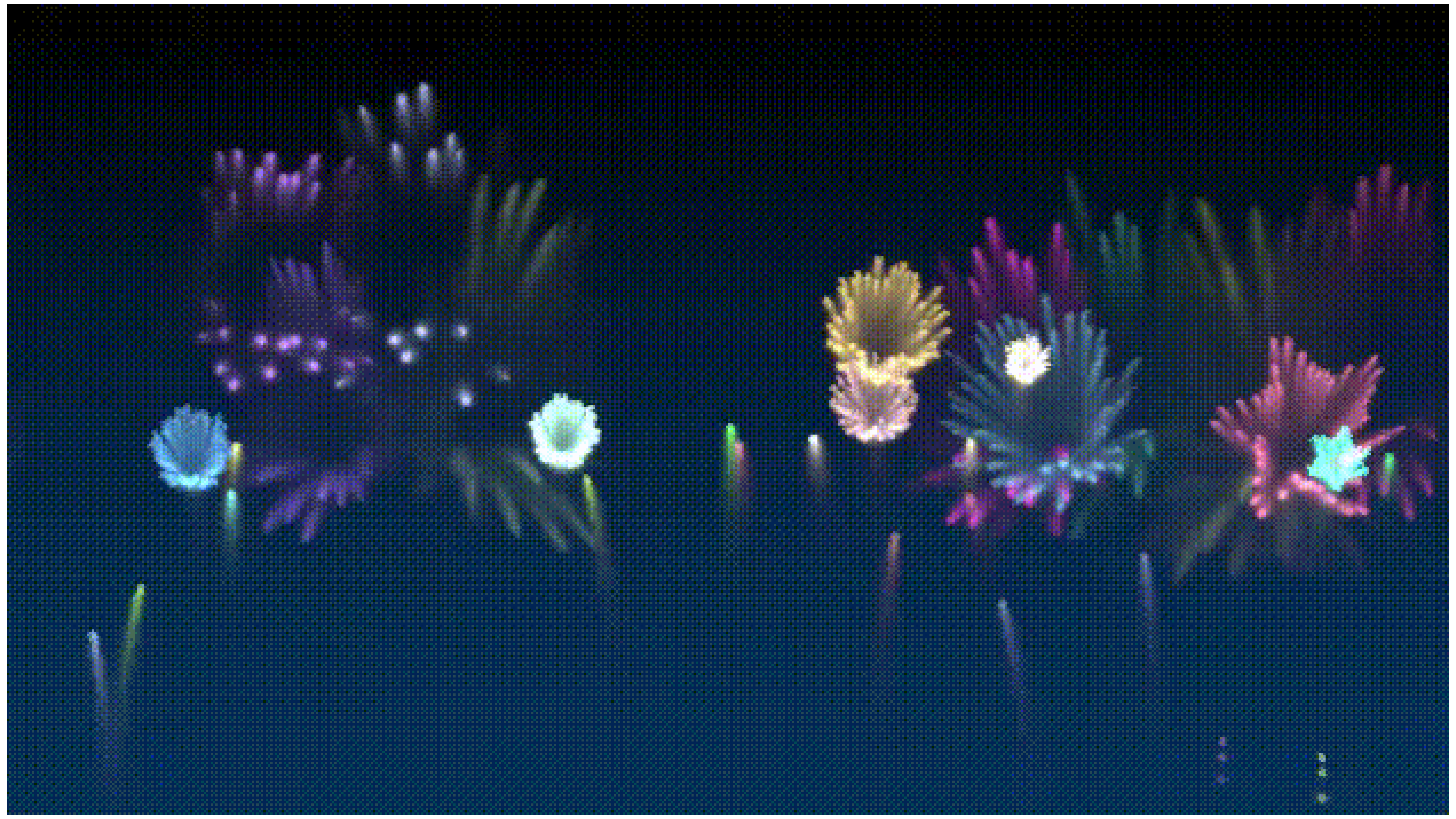
We will practice this in A8

- Program a particle system on **CPU (C++)**



Supports complicated logic but runs slow (if not parallelized)

- Program a particle system on **GPU (GLSL)**



Runs fast but does not support complicated particle dynamics (if not implemented with buffers)

Particle System on CPU

Implementing a Particle System on CPU

- Maintain the data structure of a particle system on CPU
- Update particle status (e.g., color, velocity, position, etc.) on CPU
- Synchronize particle data (e.g., position) from CPU to GPU
- Each particle is rendered as a billboard
- **Pros:** we can implement complex logic of particle dynamics (e.g., different forces, particle interactions, etc.)
- **Cons:** CPU-to-GPU data transfer is slow

A Particle System Data Structure on CPU:

$\boldsymbol{x}_1$	$\boldsymbol{u}_1$	$\boldsymbol{f}_1$	$\boldsymbol{x}_2$	$\boldsymbol{u}_2$	$\boldsymbol{f}_2$	$\boldsymbol{x}_3$	$\boldsymbol{u}_3$	$\boldsymbol{f}_3$	$\boldsymbol{x}_4$	$\boldsymbol{u}_4$	$\boldsymbol{f}_4$

Position Velocity Force

A Typical Particle Data Structure in C++

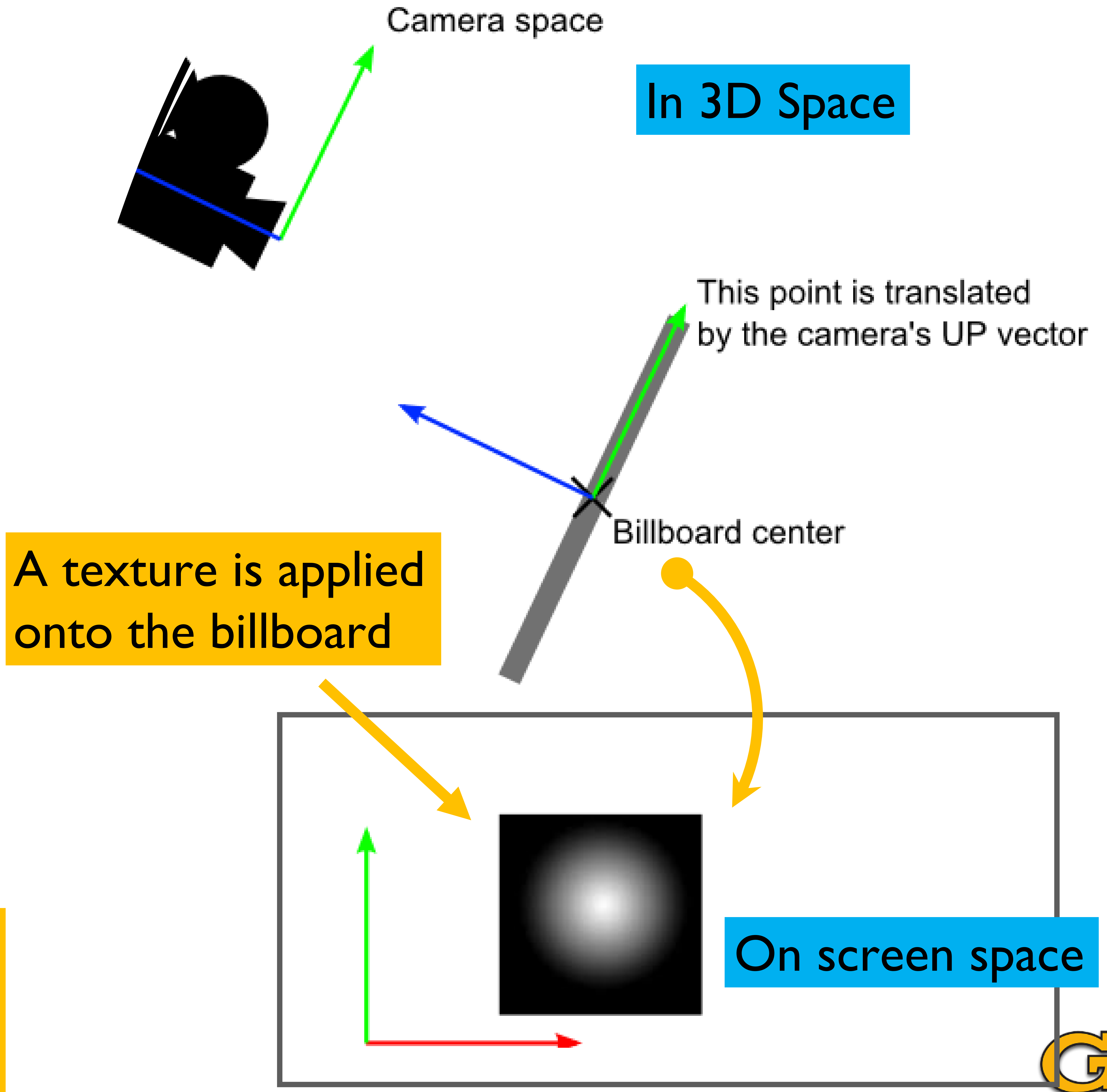
```
class Particle
{
	Vector3 position;
	Vector3 velocity;
	Vector3 force;
};

std::vector<Particle> particles;
```

Particle Rendering with a **Billboard**

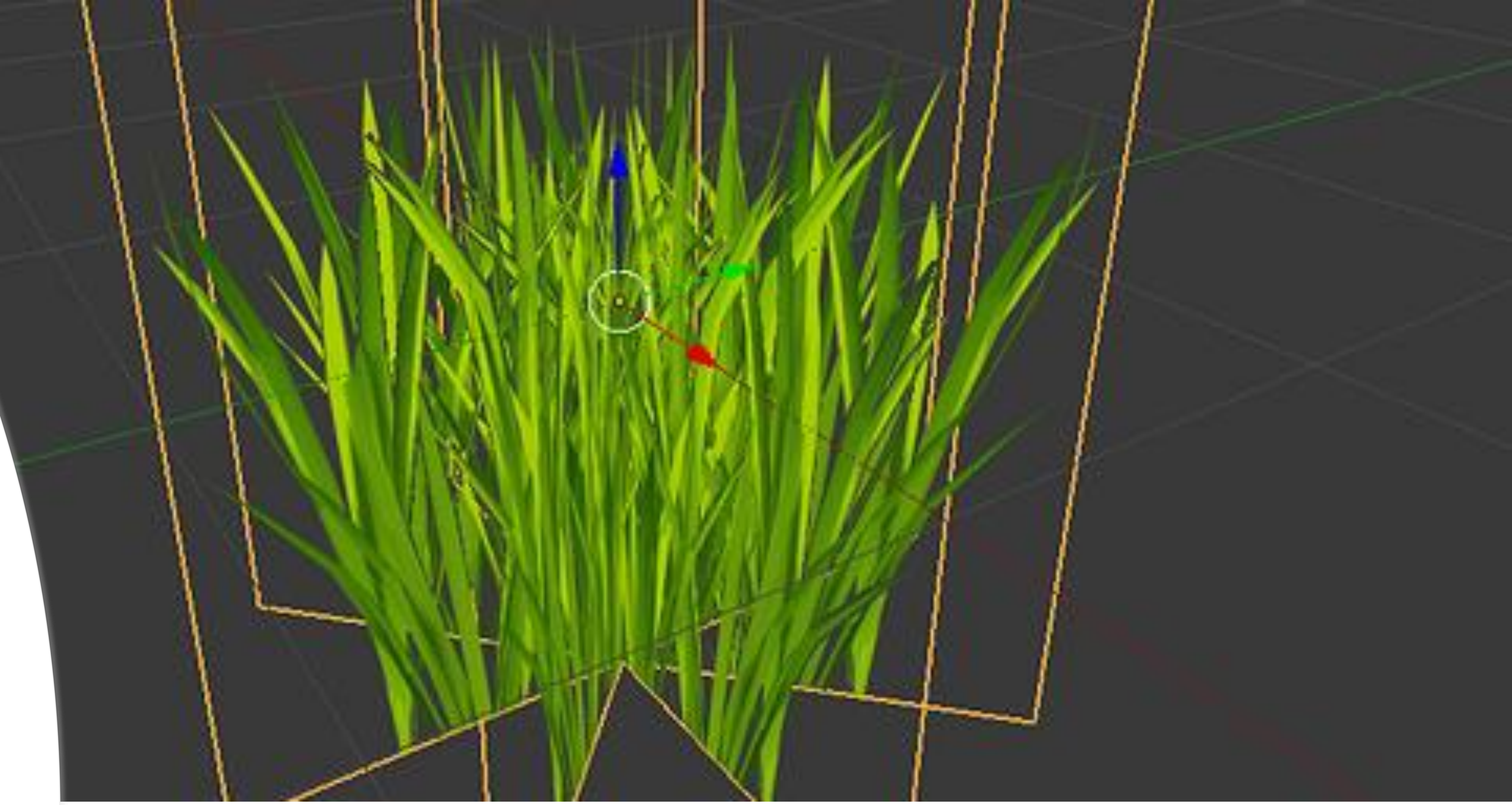
- **Key idea**: Render each particle as a billboard that can automatically face the camera.
- We then put texture on each billboard with transparency
- The texture can be implemented by either reading a texture or by calculating an analytical function

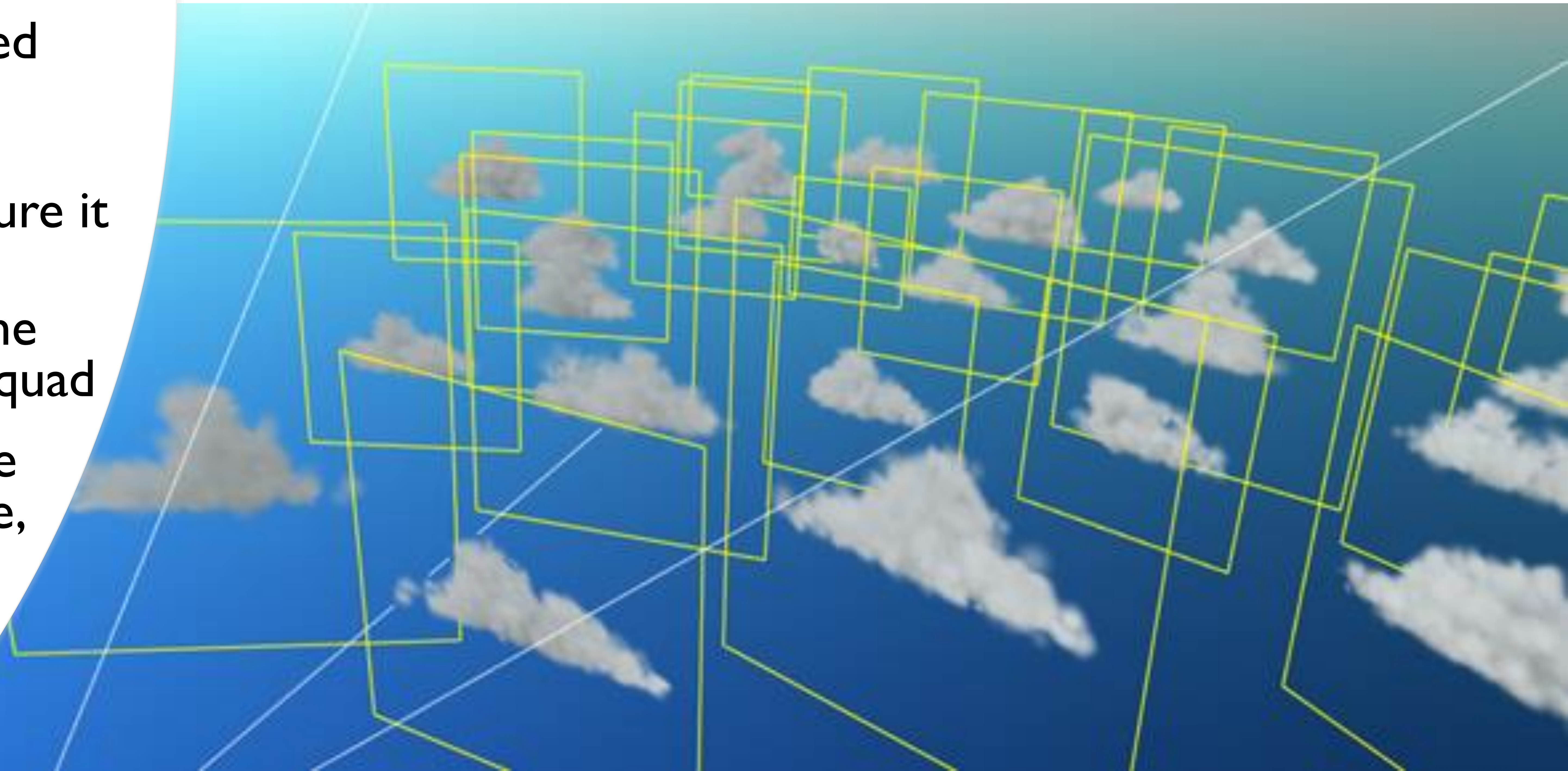
This model is particularly suitable for rendering particles thar are passed from CPU to GPU



Implementing Billboard particles with GLSL shaders

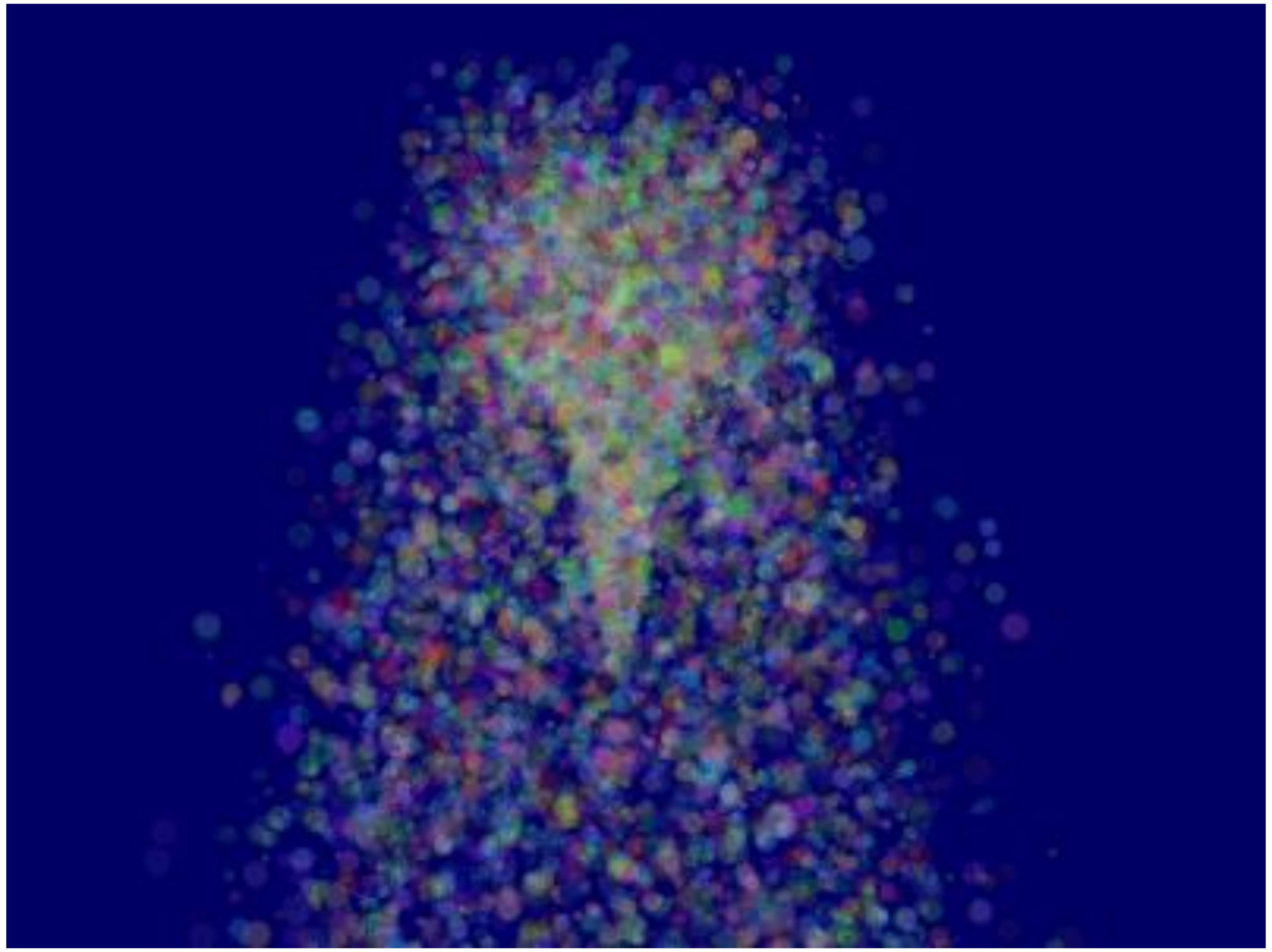
- **Pipeline:** Updating the particle positions on CPU, synchronize these positions from CPU to GPU, and render each particle as a small quad with transparency (billboard) that faces the camera on GPU
- **GLSL Shaders:** This step needs to be implemented with both vertex shader and fragment shader:
  - In the vertex shader, we need to process the geometry information of the quad and make sure it faces the camera
  - In the fragment shader, we need to calculate the color of each fragment that is covered by the quad
- **Applications:** This idea applies not only to particle rendering but also other objects such as smoke, fire, cloud, trees





Example: Render Particle Billboards with Transparency

Rendered without transparency



Rendered with transparency

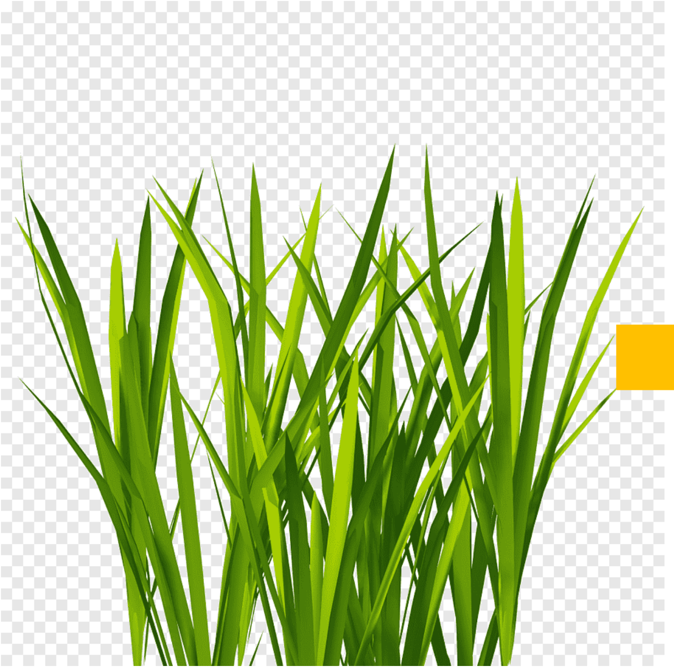
https://www.opengl-tutorial.org/intermediate-tutorials/billboards-particles/particles-instancing/

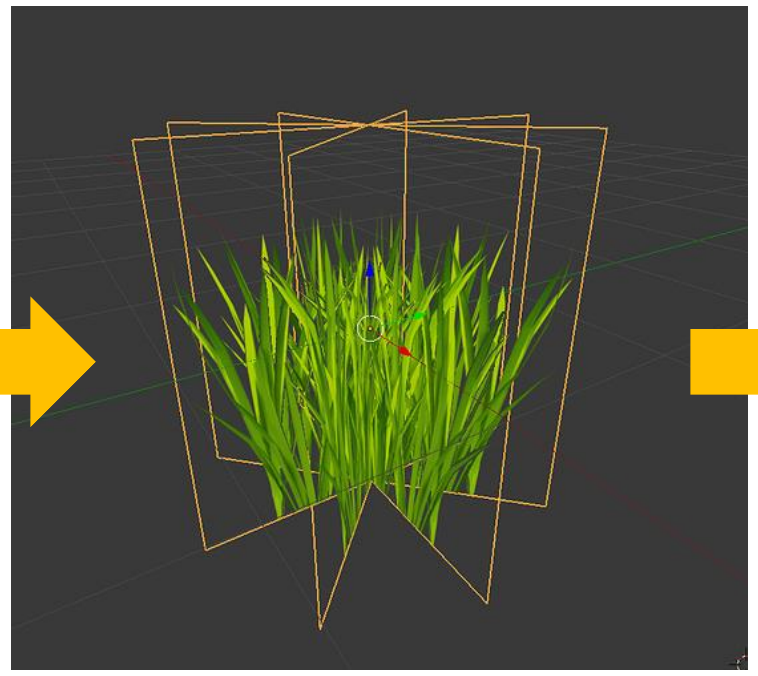
Example: **Cloud** Rendering in Video Games

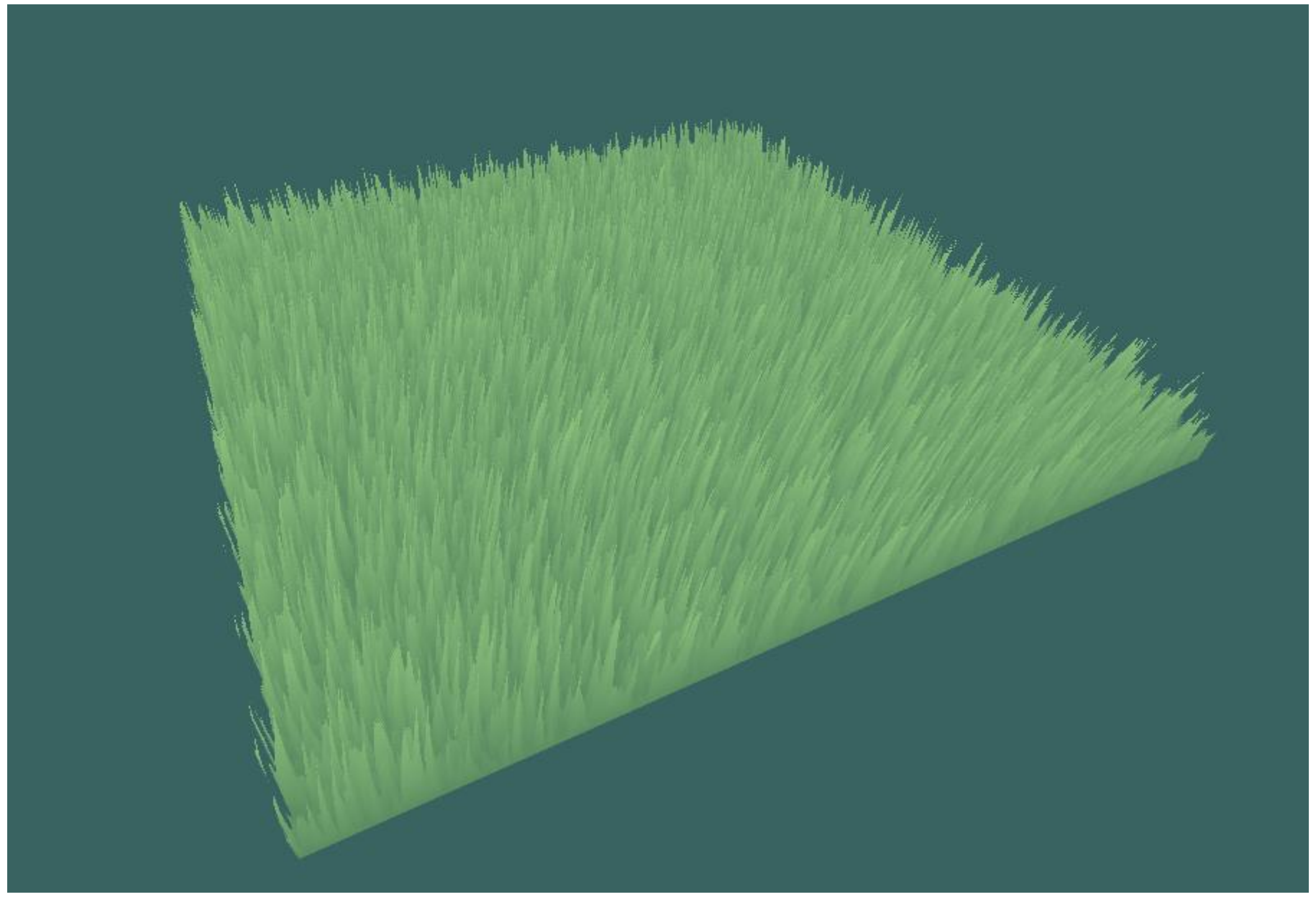
- Render each cloud as a billboard with cloud textures
- Move particles to animate the motion of cloud
- Render a static skybox as the background

Example:
Grass Rendering in Video Games

- Render each piece of grass as a billboard with textures
- Put multiple billboards with rotated angles to mimic 3D
- Animate a large number of billboards with wind

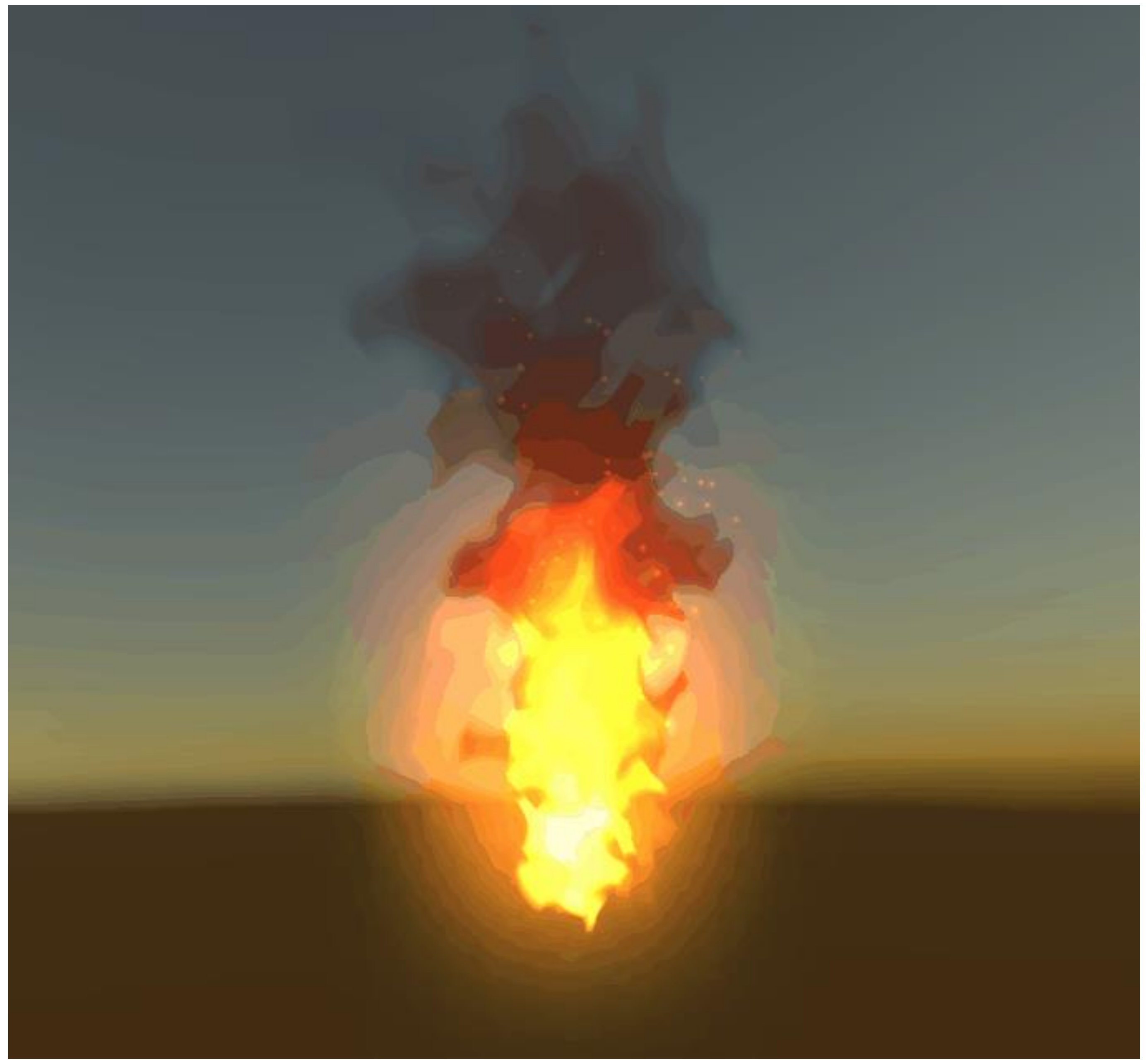


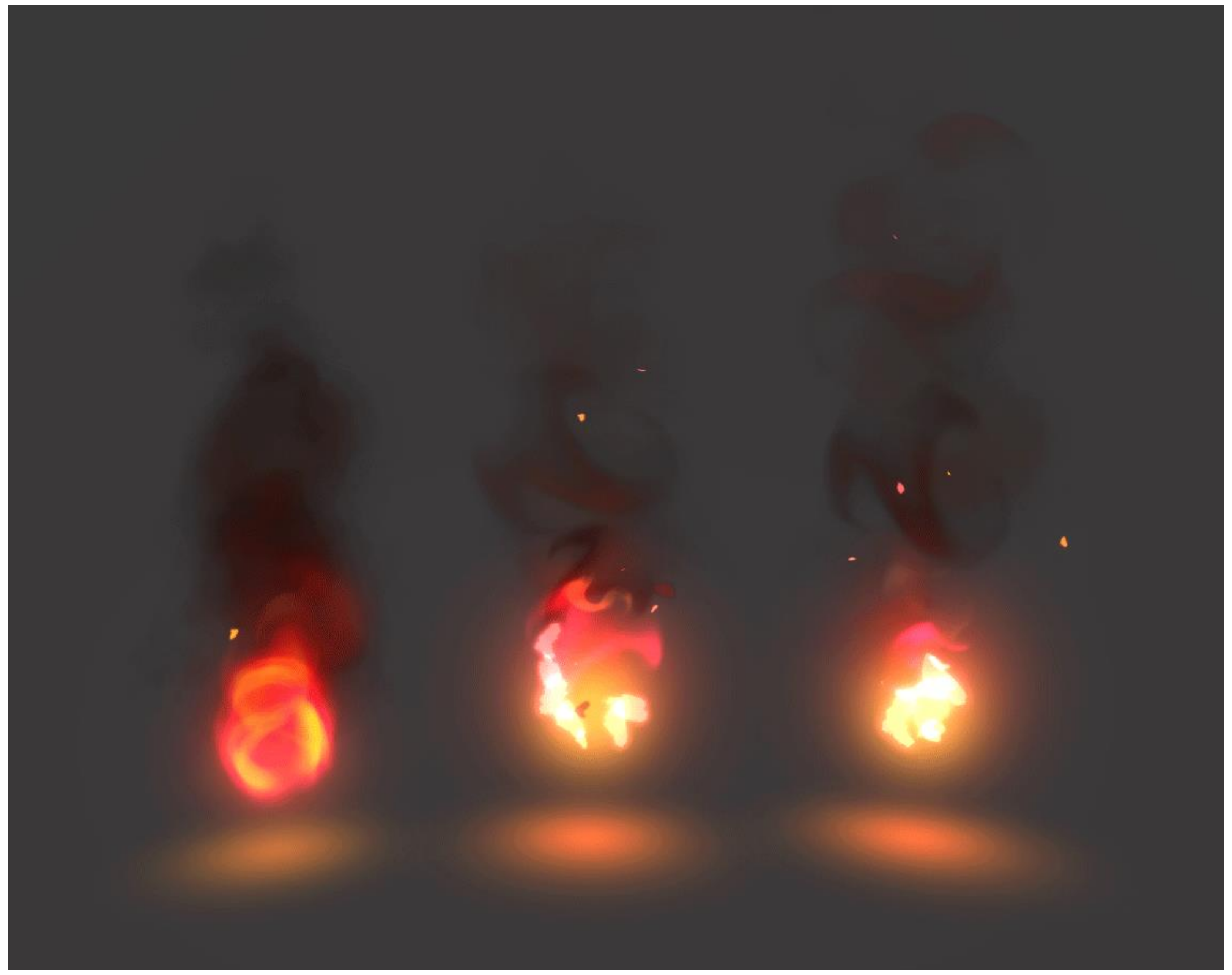


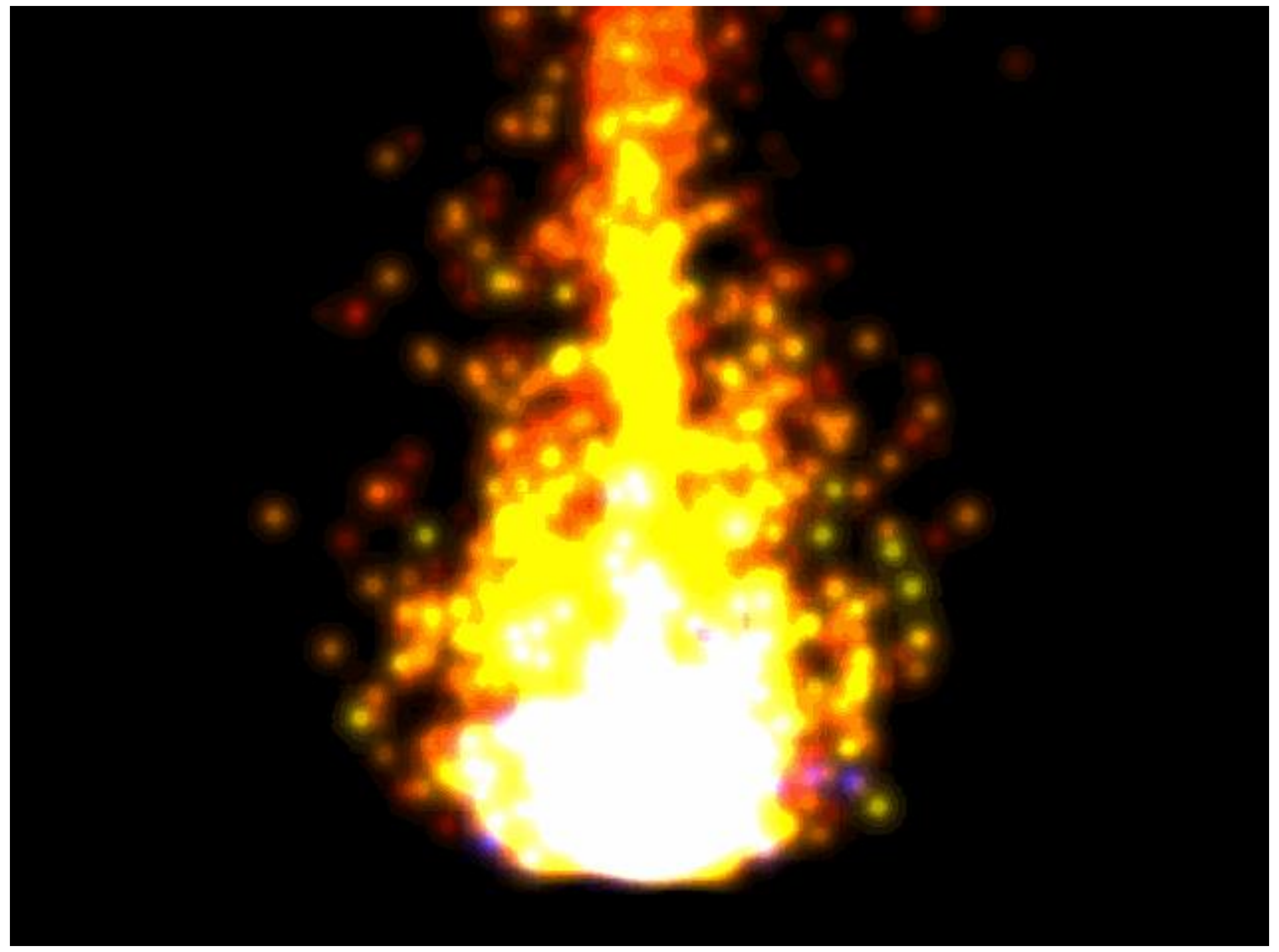


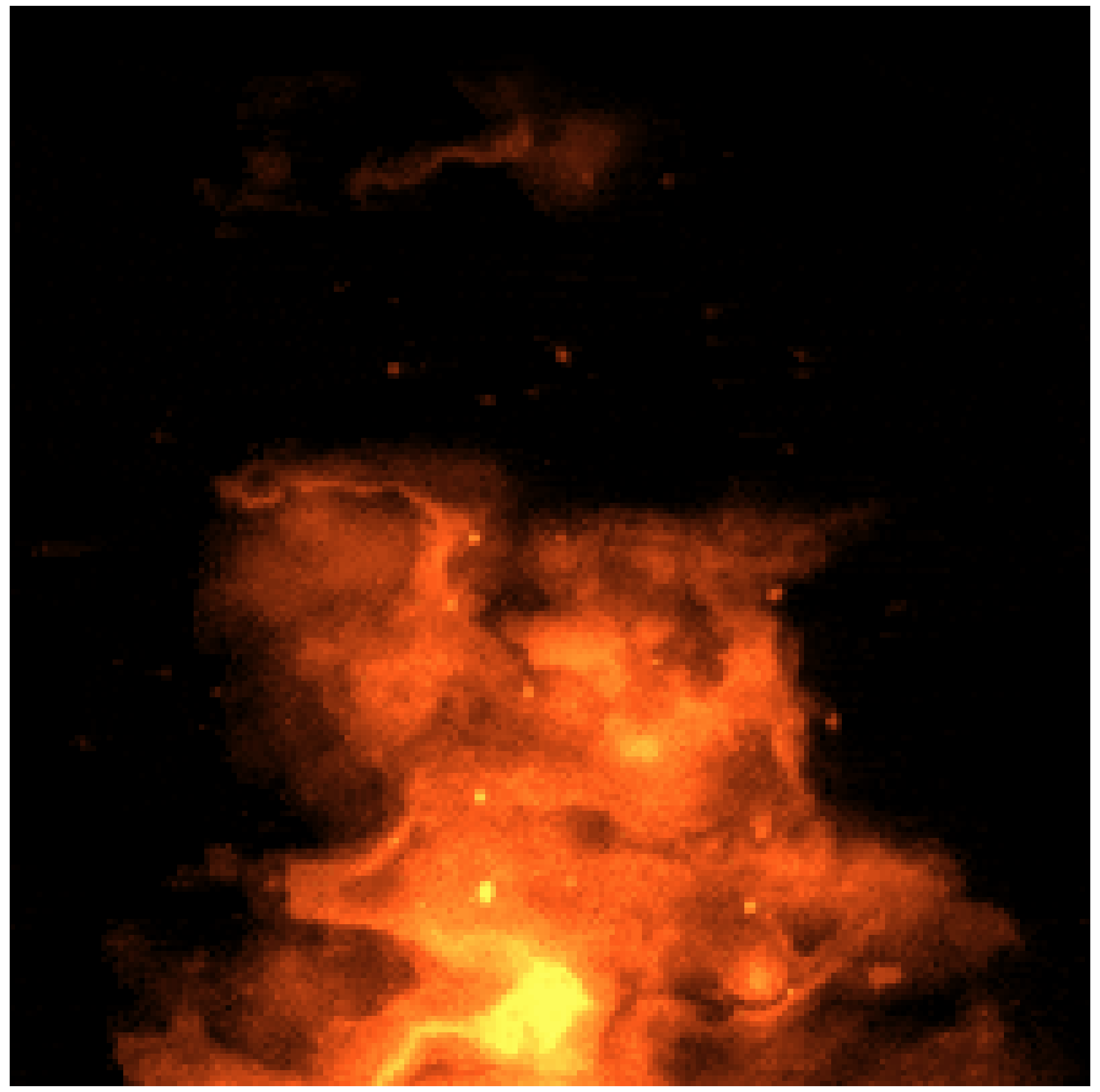
Example:
Fire Rendering in Video Games

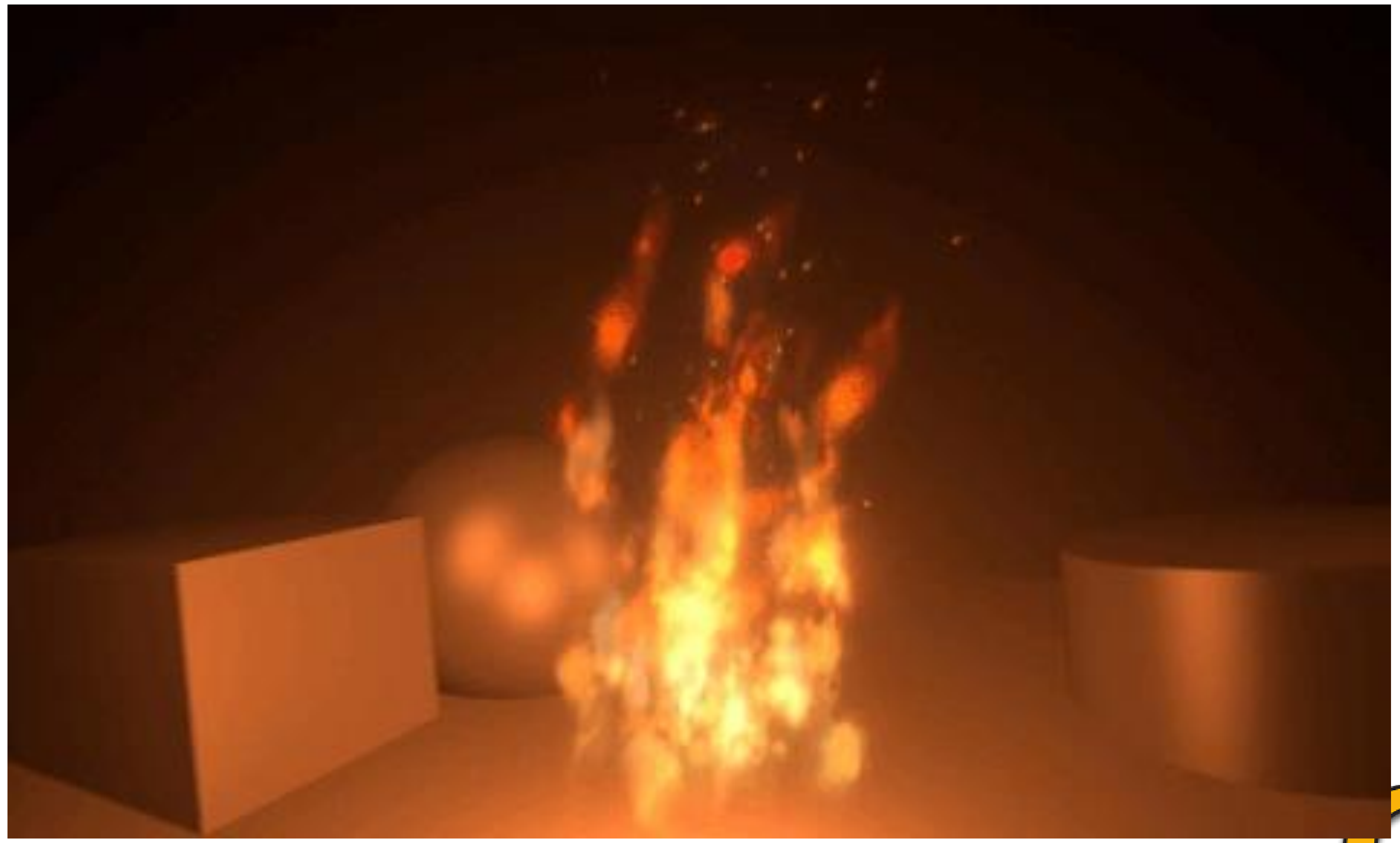
- Render the rising flame as a set of particles
- Each particle carries a billboard with an emissive color
- Animate a large number of fire billboards with wind

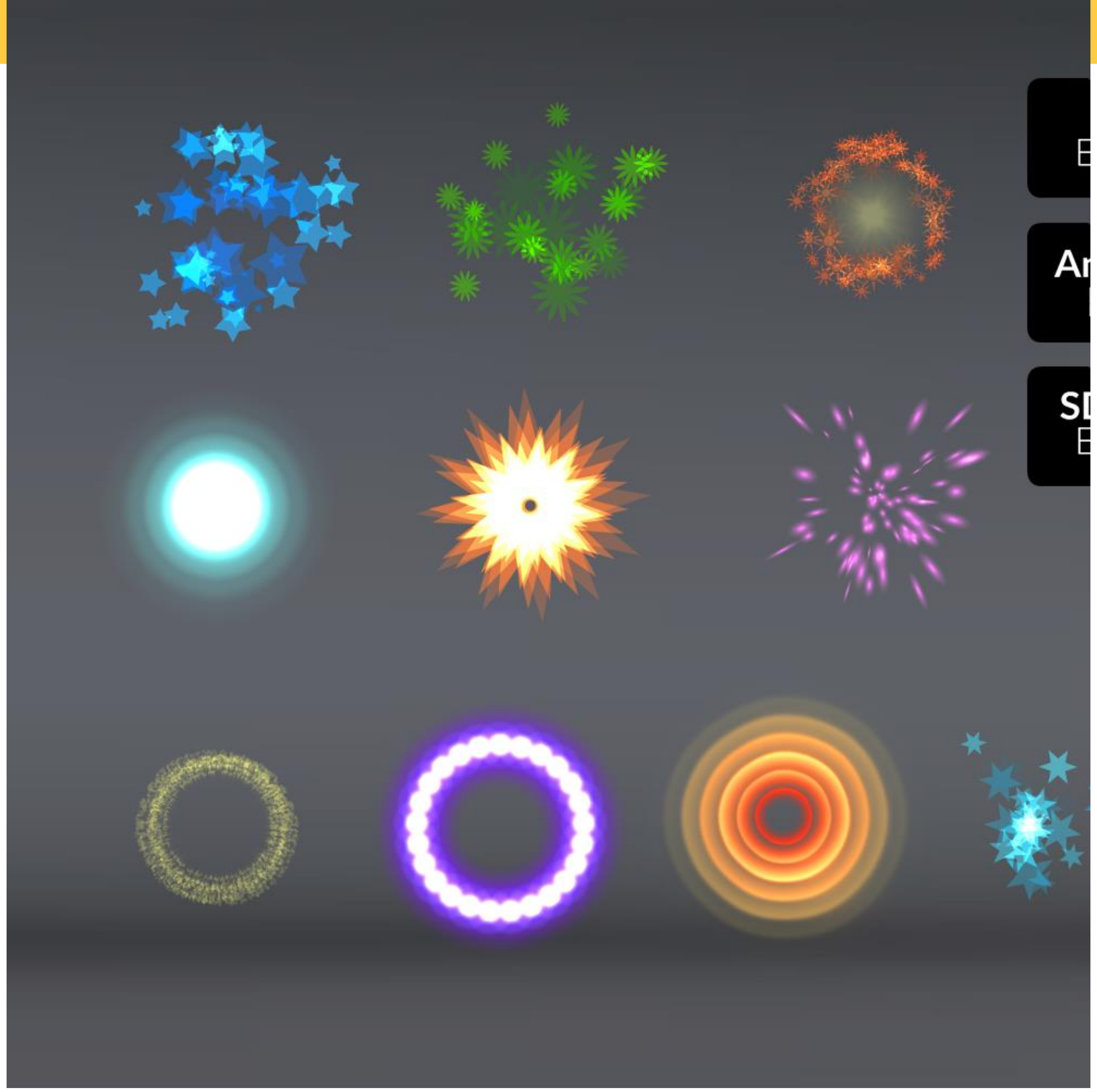


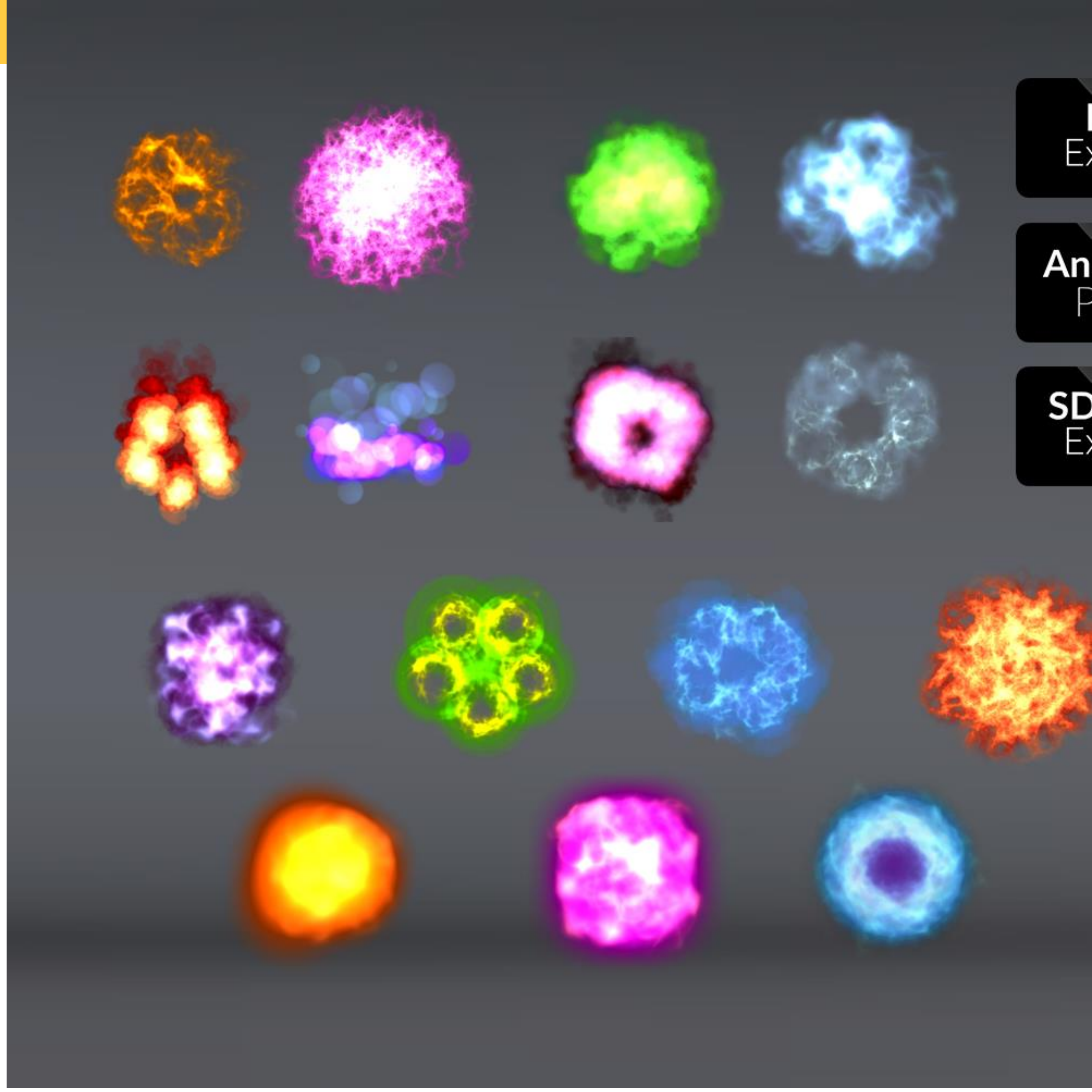






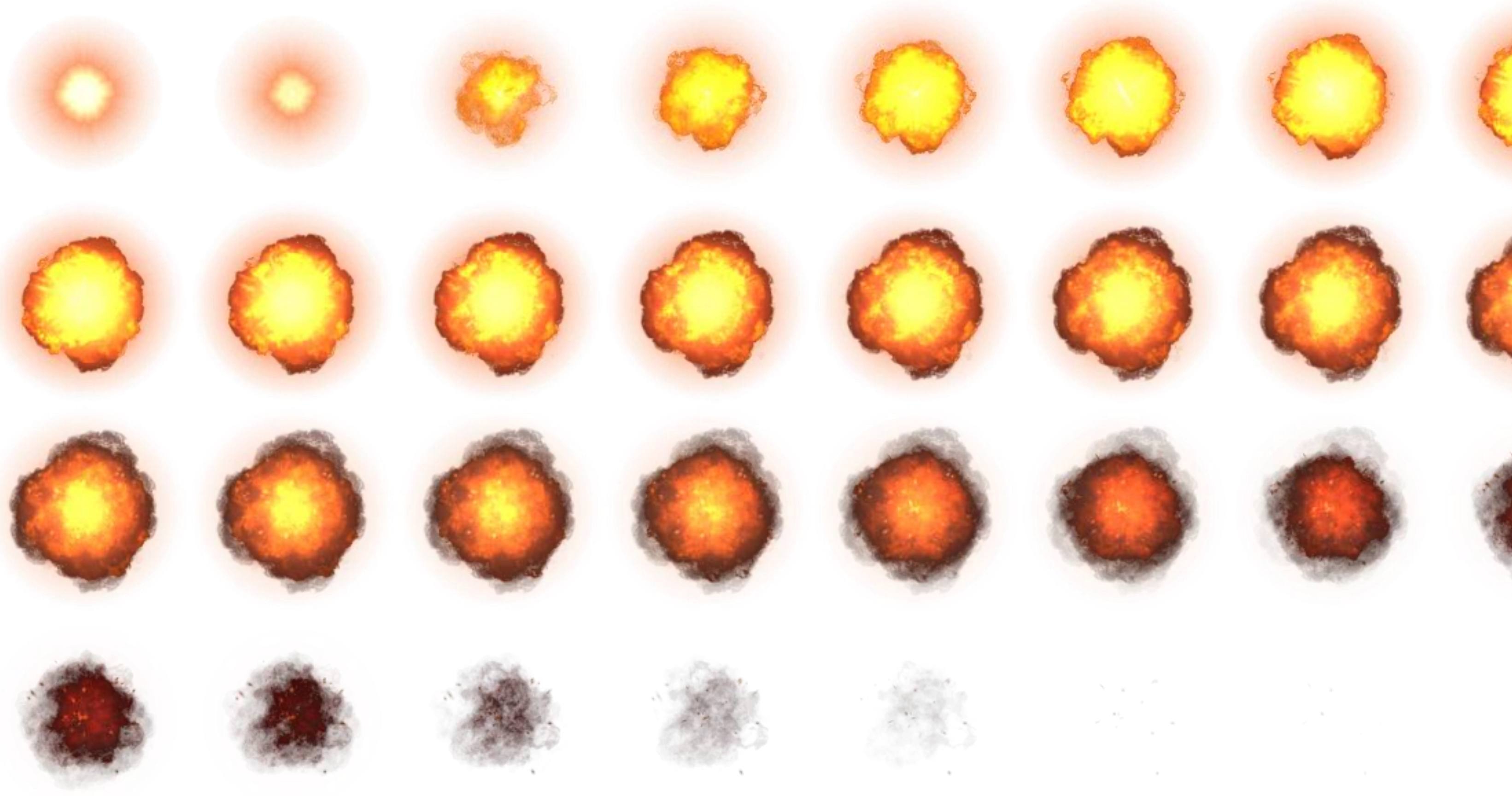






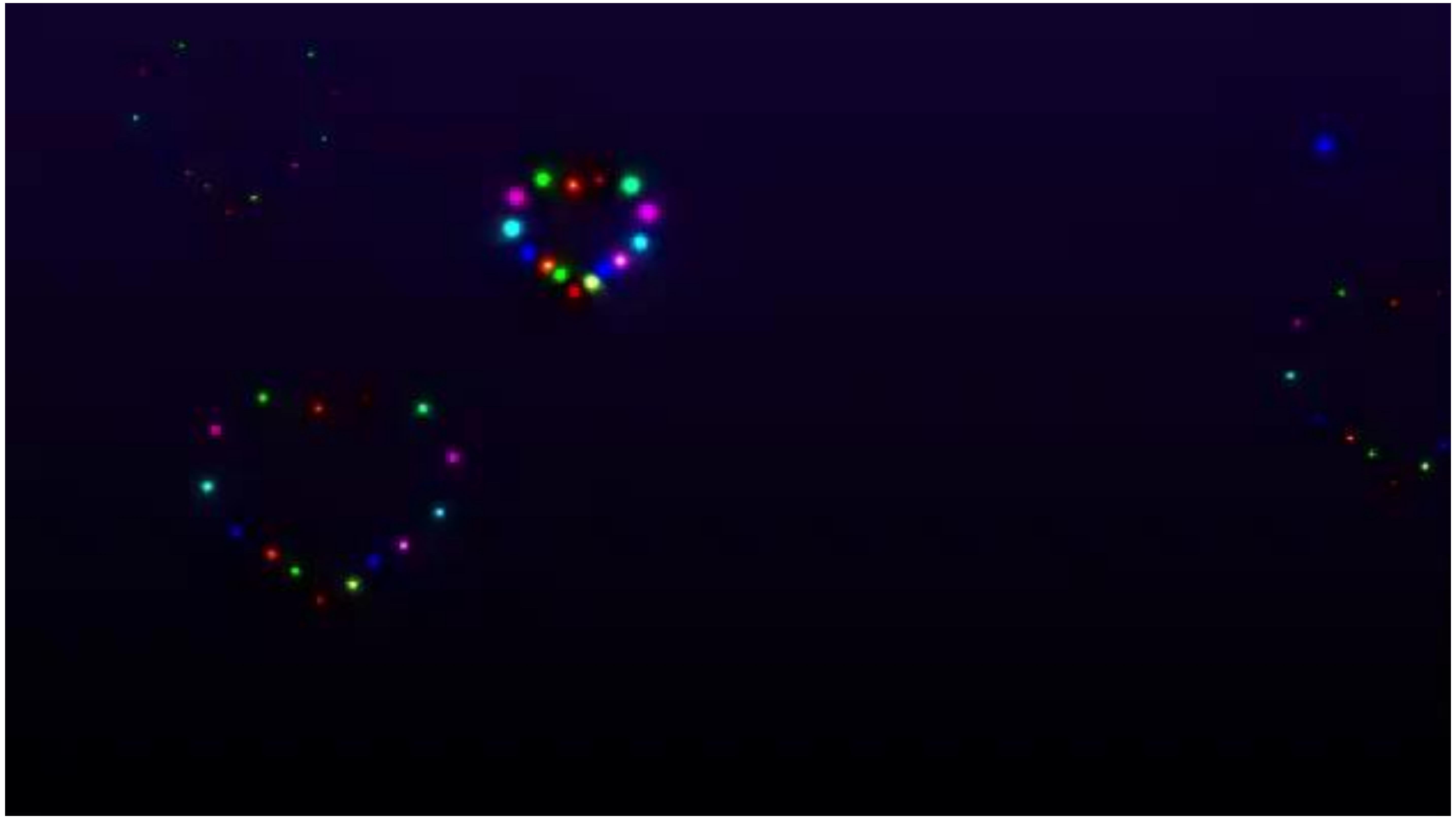
A lot of Particle Textures Available Online

- We can read frames from different textures to produce different types of animations, e.g., smoke, cloud, explosion, magic effects, etc.



Particle System on GPU

Case Study: Firework Animation



Particle Systems on ShaderToy

Search key words "particles", "firework", etc.

https://www.shadertoy.com/results?query=fireworks

A8 Live Demo: **Fireworks under the Starry Sky**

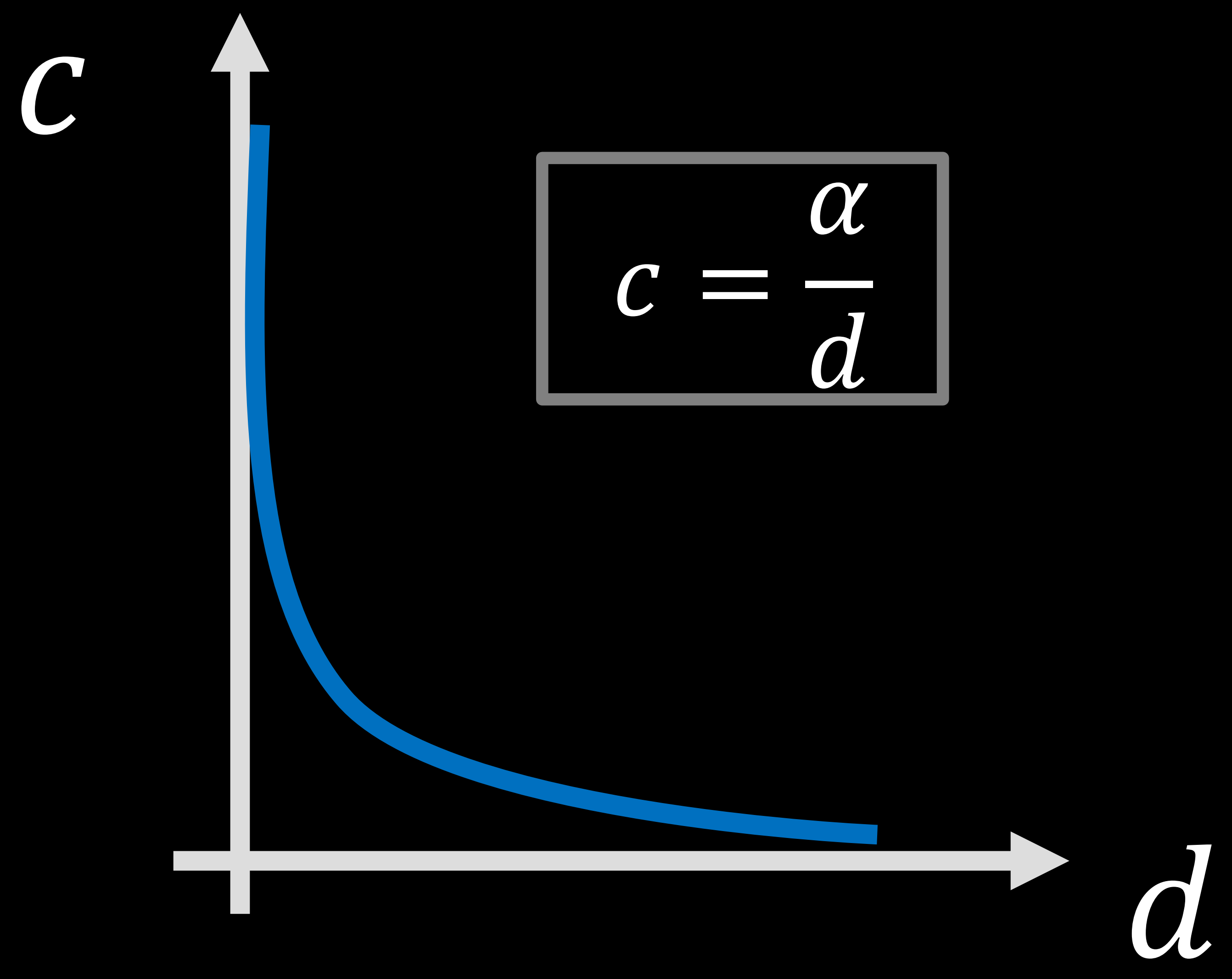
to celebrate our achievements over the semester ☺

Simulating a Particle System on GPU (GLSL)

- **Key Idea:** Simulate a 2D particle system in fragment shader and render each particle's shape by checking its distance to the fragment
- Each particle carries a position only
- Each particle's position is updated by an analytical function of time t
- Each particle is rendered by checking the distance between its current position and the fragment's position and then draw a blur function
- Each fragment checks the status of all particles on the screen by using a for loop
- New particles can be generated on the fly by modifying the for loop

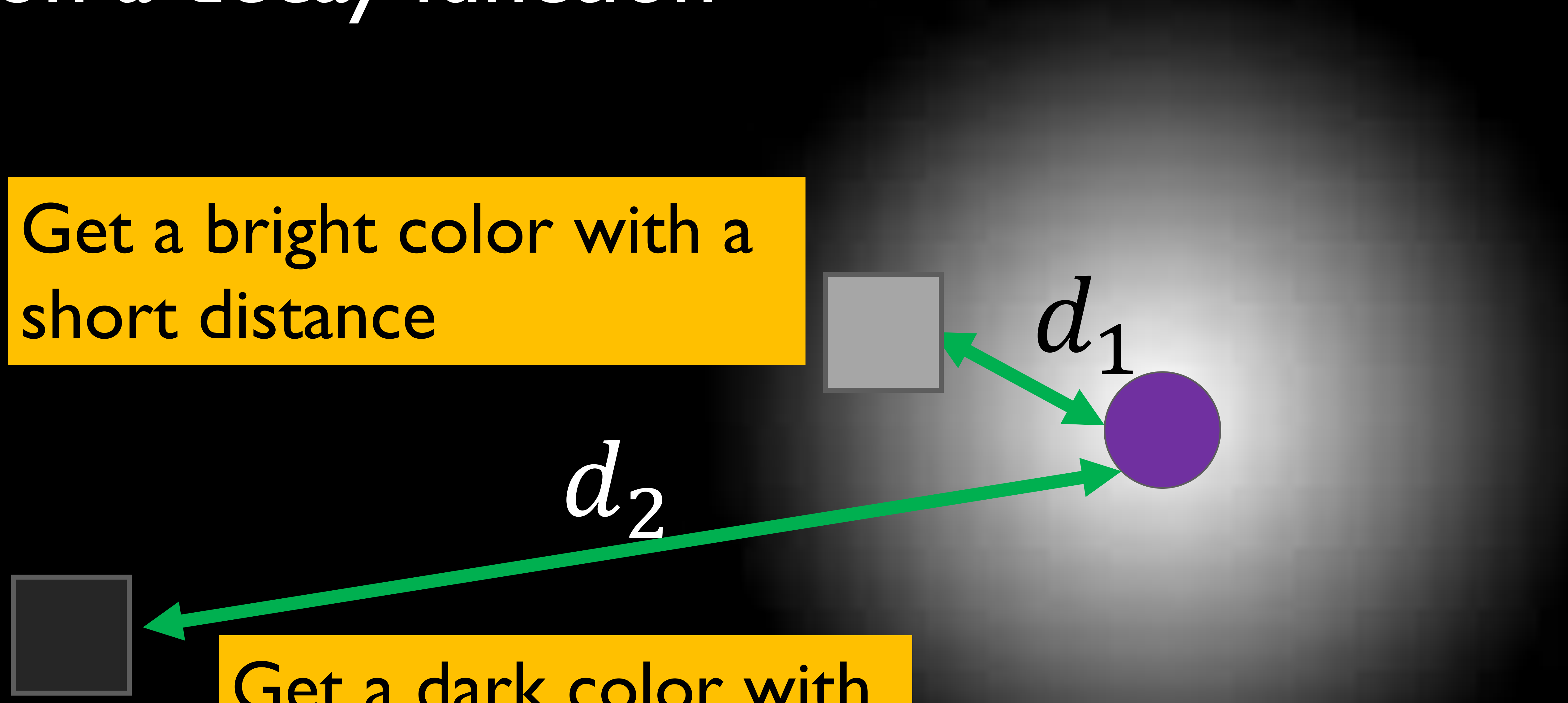
Render a Single Particle in Fragment Shader

- Let's start with a simple case: render a **single particle** on screen
- We know the particle's position, and we want to render it like a Gaussian blob in the fragment shader
- For each pixel, we check its distance to the center of the particle, and then calculate a color based on a decay function



Decay function example

Get a bright color with a short distance



Get a dark color with a long distance

Pseudocode: **Render Particle**

- Input: /*fragment position*/ frag_pos, /*particle position*/ particle_pos, /*particle's default brightness*/ brightness, /*particle's default color*/ color
- Output: /*fragment color*/ frag_color
- Algorithm:

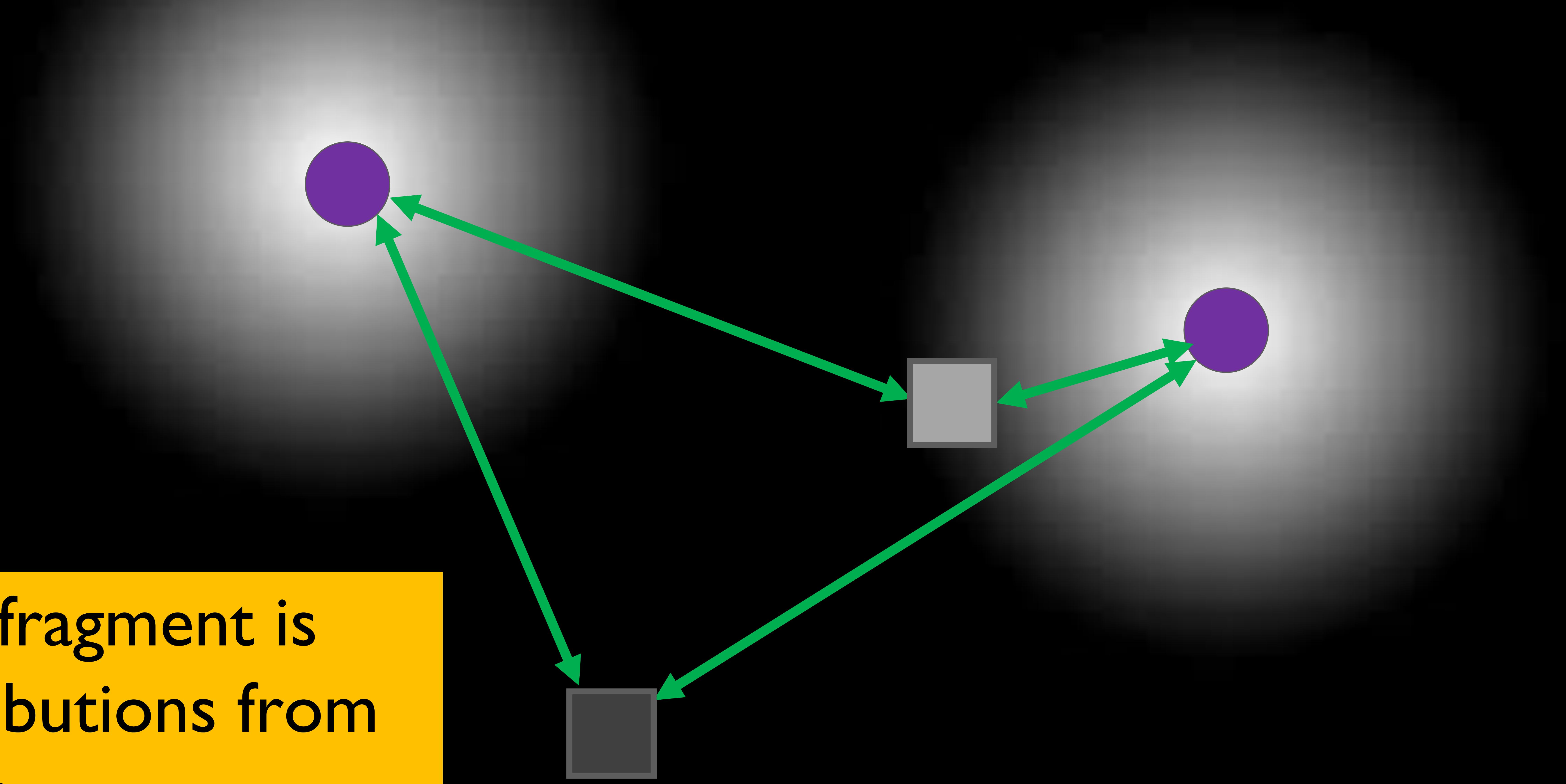
```
distance = length(frag_pos – particle_pos)
decay = 1 / distance
frag_color = color * brightness * decay
return frag_color
```

We can play with the distance, direction, as well as the decay function to produce different particle shapes

Render a single particle in GLSL

Render Multiple Particles in Fragment Shader

- Easy! Write a **for loop** to go over all particles for each pixel



The color of the fragment is the sum of contributions from **all** particles on the screen

Pseudocode: **Render Multiple Particles**

- Input: /*fragment coordinate*/ frag_pos
- Output: /*fragment color*/ frag_color
- Algorithm:

```
frag_color = vec3(0,0,0)
for each particle p:
    get its position, brightness, and color
    frag_color += SingleParticleColor(frag_pos, particle_pos, brightness, color)
return frag_color
```

We can create functions of time to change each particle's appearance (e.g., brightness and color) to produce effects like flickering, fading, color transition, etc.

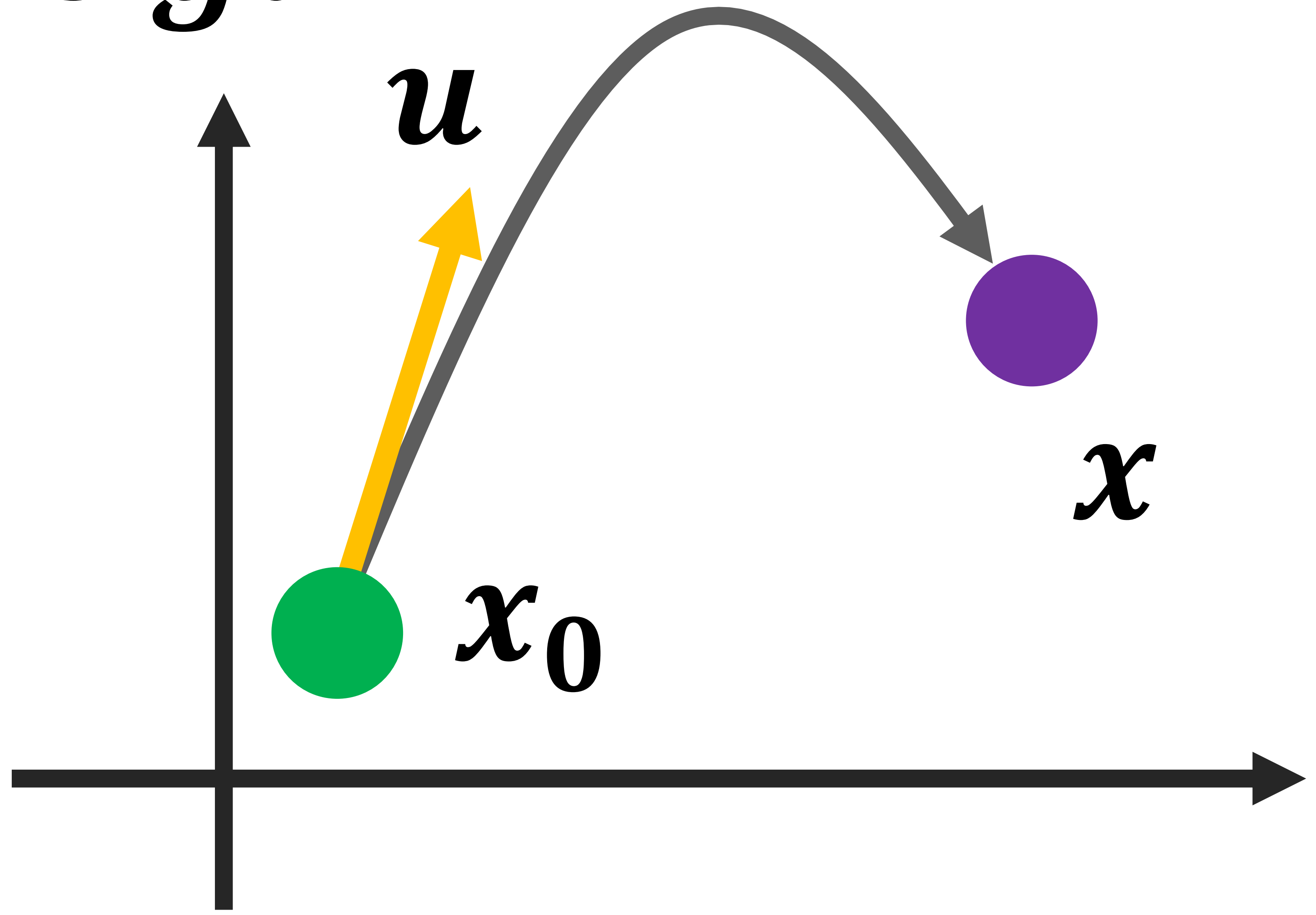
Render a starry sky in GLSL

Move a Particle in a Fragment Shader

- **Key idea:** implement a function of time t to calculate the particle's position in the current time
- For instance, we can implement the ballistic motion of a firework particle by calculating the particle's trajectory as a function of t:

$$\boldsymbol{x} = \boldsymbol{f}(t) = \boldsymbol{x}_0 + \boldsymbol{u}t + 0.5\,\boldsymbol{g}t^2$$

Here we take the ballistic motion as an example, but you may extend the idea to other particle trajectories



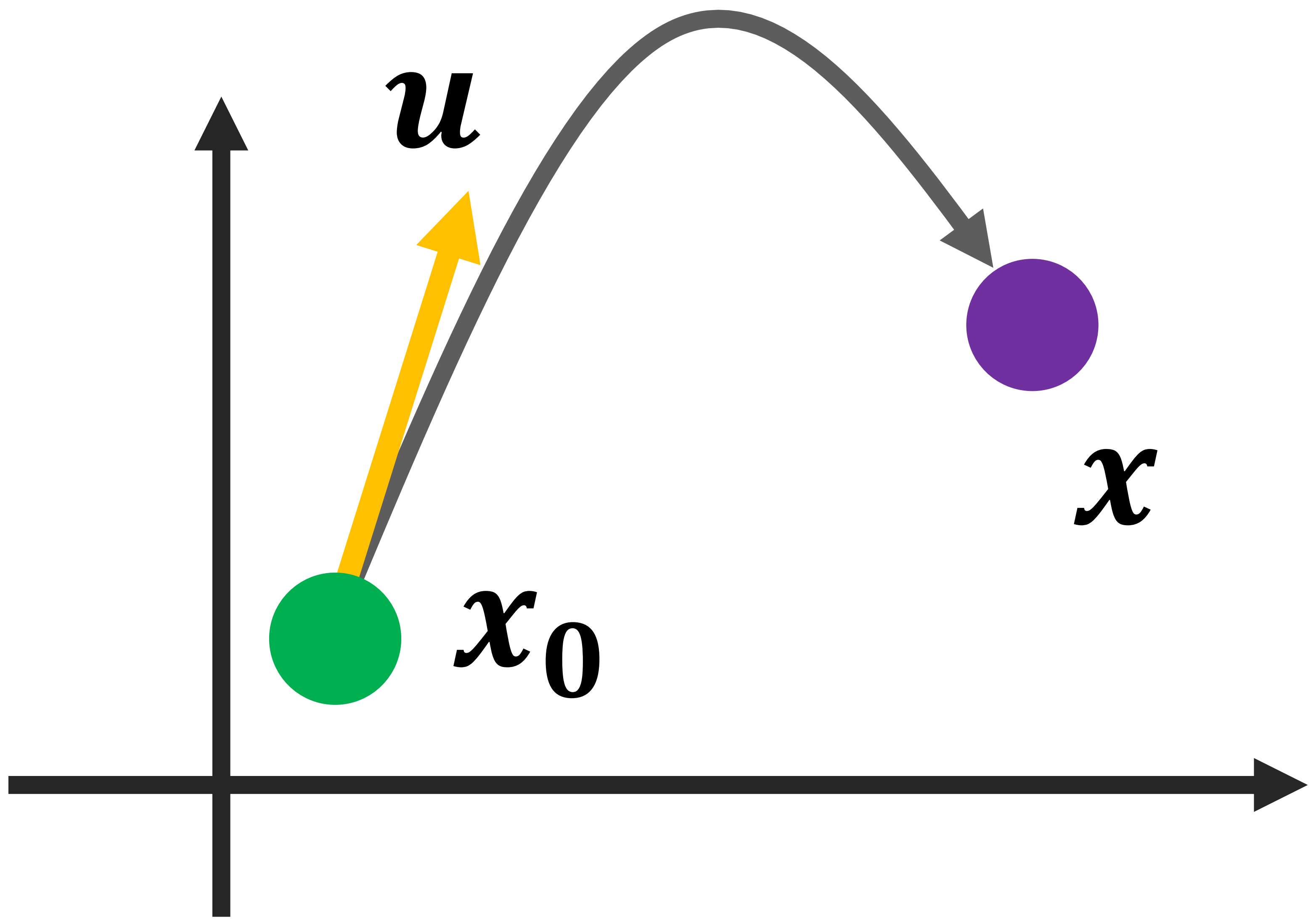
Pseudocode: **Move Particle**

- Input: /*particle's initial position*/ init_pos, /*particle's initial velocity*/ init_vel, /*time*/ t
- Output: /*particle's current position*/ pos
- Algorithm:

  calculate the particle's current position with its initial position, velocity, and time t (typically using an analytical function, e.g., ballistic motion)

  return current position

$$\boldsymbol{x} = \boldsymbol{f}(t) = \boldsymbol{x}_0 + \boldsymbol{u}t + 0.5\,\boldsymbol{g}t^2$$



Simulate a Single Particle System

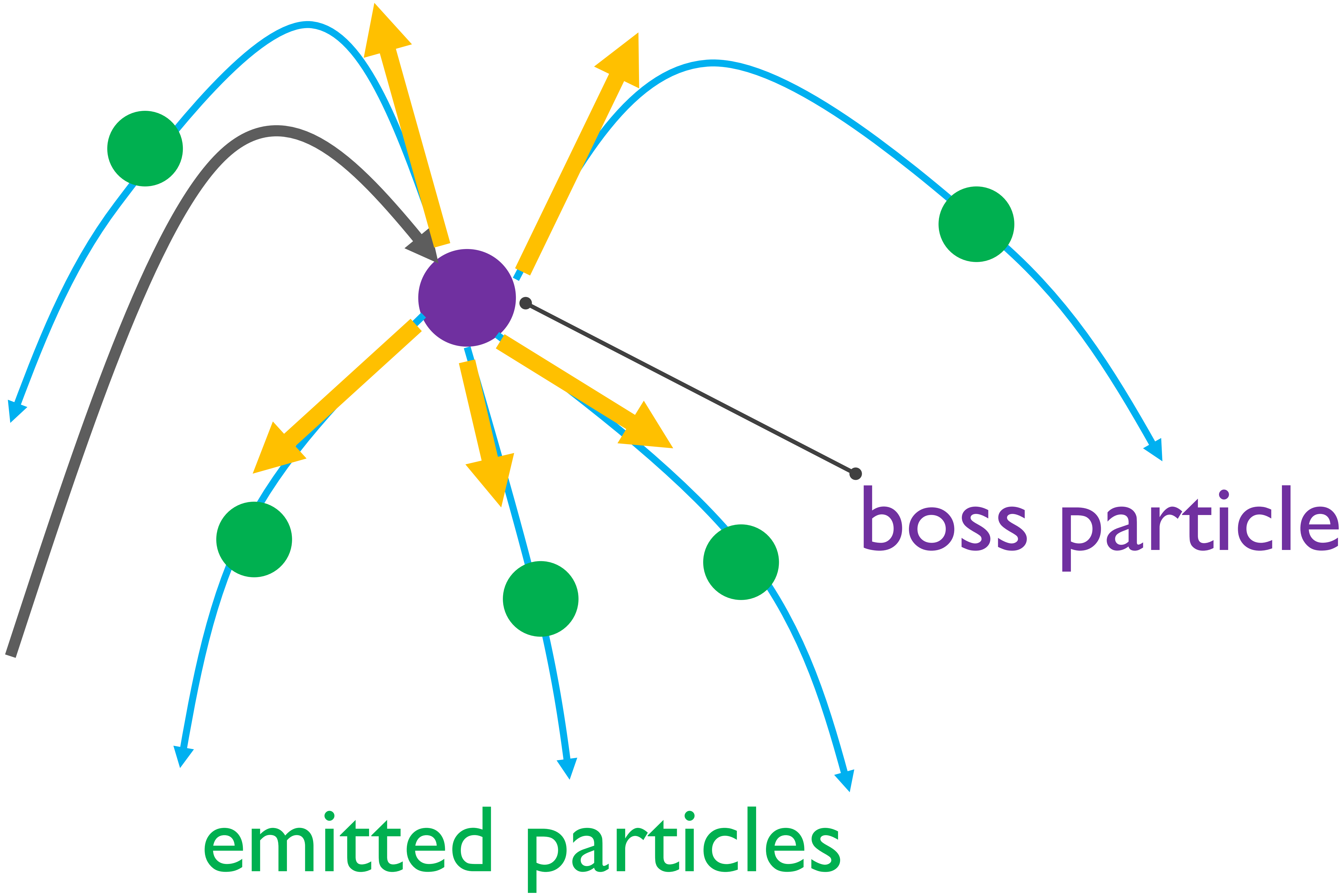
- We can simulate the motion of a single particle by repeating the two steps to update the particle's position and appearance to the current time t:

```
particle_pos = MoveParticle( init_pos, init_vel, t)
frag_color = RenderParticle( frag_pos, particle_pos, brightness, color)
```

Simulate a single particle

Firework in a Fragment Shader

- **Key Idea:** Emit a group of new particles around a **boss particle** with a **for loop**, and update the position of each emitted particle



We can divide the simulation into two phases: for **Phase I**, we only simulate and render the boss particle; for **Phase II**, we remove the boss particle, seed a group of emitted particles, and then simulate and render these particles for the rest period of time

Pseudocode: Firework

- Input: /*time*/ t
- Output: /*fragment color*/ frag_color
- Algorithm:

```
if t < emitTime:
        simulate the boss particle to time t
        frag_color += render boss particle
otherwise:
        emitPos = boss particle's position at emitTime
        for each emission particle
                simulate its position from emitTime to t2
                frag_color += render emit particle
return frag_color
```

Simulate a firework

Simulate many fireworks

Reading Materials

- [CPU particle system] SIGGRAPH course notes on particle dynamics: https://www.cs.cmu.edu/~baraff/sigcourse/notesc.pdf
- [Particle system in GLSL] https://learnopengl.com/In-Practice/2D-Game/Particles