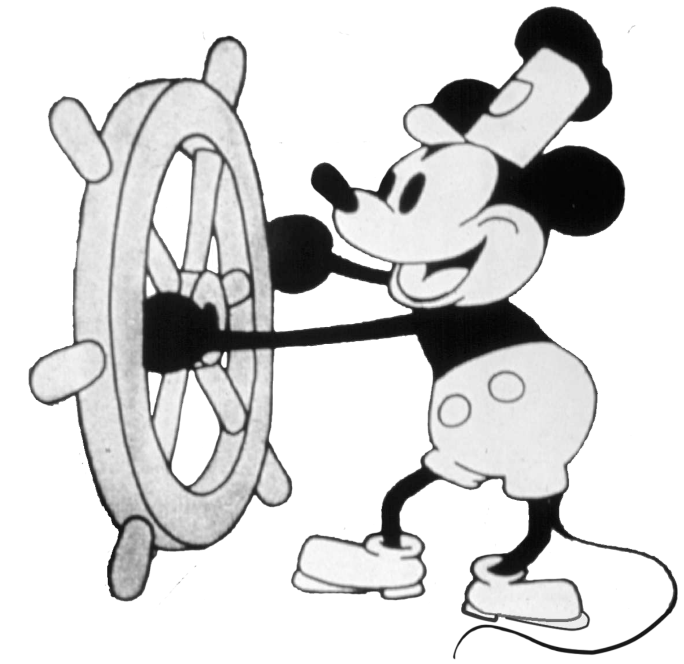


CSCI 461: Computer Graphics

Middlebury College, Spring 2025

Lecture 11: Particle Animation

In Lab 1, we prescribed motion using curves.

how to model this?

fluid motion is complex

today, our artist is the laws of physics

By the end of today's lecture, you will be able to:

- implement **Euler's method** for updating the **position** and **velocity** of particles,
- **draw** particles as either squares or using a sprite,
- use **transform feedback** to **ping-pong** updated position and velocity data during an animation.

Revisiting the Pixar ball from the first lab.

$$\vec{F}_d = -\frac{1}{2}\rho A C_d \frac{\vec{V}}{\|\vec{V}\|}\|\vec{V}\|^2$$

velocity

density of air

area

drag coefficient

$\vec{F}_d$ (drag force)

opposes motion of ball (velocity)

why does the ball fall

+y

(gravity force)

$$\vec{F}_g = m\vec{g}$$

mass

$$\vec{g} = \begin{bmatrix} 0 \\ -9.81 \\ 0 \end{bmatrix} \; m/s^2$$

Let's just use a computer...

$$\vec{v}^k = \frac{\Delta \vec{p}}{\Delta t} = \frac{\vec{p}^{k+1} - \vec{p}^k}{\Delta t}$$

$$\vec{a}^k = \frac{\Delta \vec{v}}{\Delta t} = \frac{\vec{v}^{k+1} - \vec{v}^k}{\Delta t}$$

$$\sum \vec{f} = m\vec{a} \qquad (2)\ \vec{a} = \frac{\sum \vec{f}}{m}$$

$$(1)\ \vec{p}^{k+1} = \vec{p}^k + \Delta t\, \vec{v}^k$$

$$(3)\ \vec{v}^{k+1} = \vec{v}^k + \Delta t \underbrace{\left(\frac{\sum \vec{f}}{m}\right)}_{\vec{a}}$$

For ball:

$$\sum \vec{f} = \vec{f}_g + \vec{f}_d \quad \text{maybe} + \vec{f}_w \ (\text{wind})$$

$$= m\vec{g} - \frac{1}{2}\rho A C_d \vec{v} \|\vec{v}\| = m\vec{a}$$

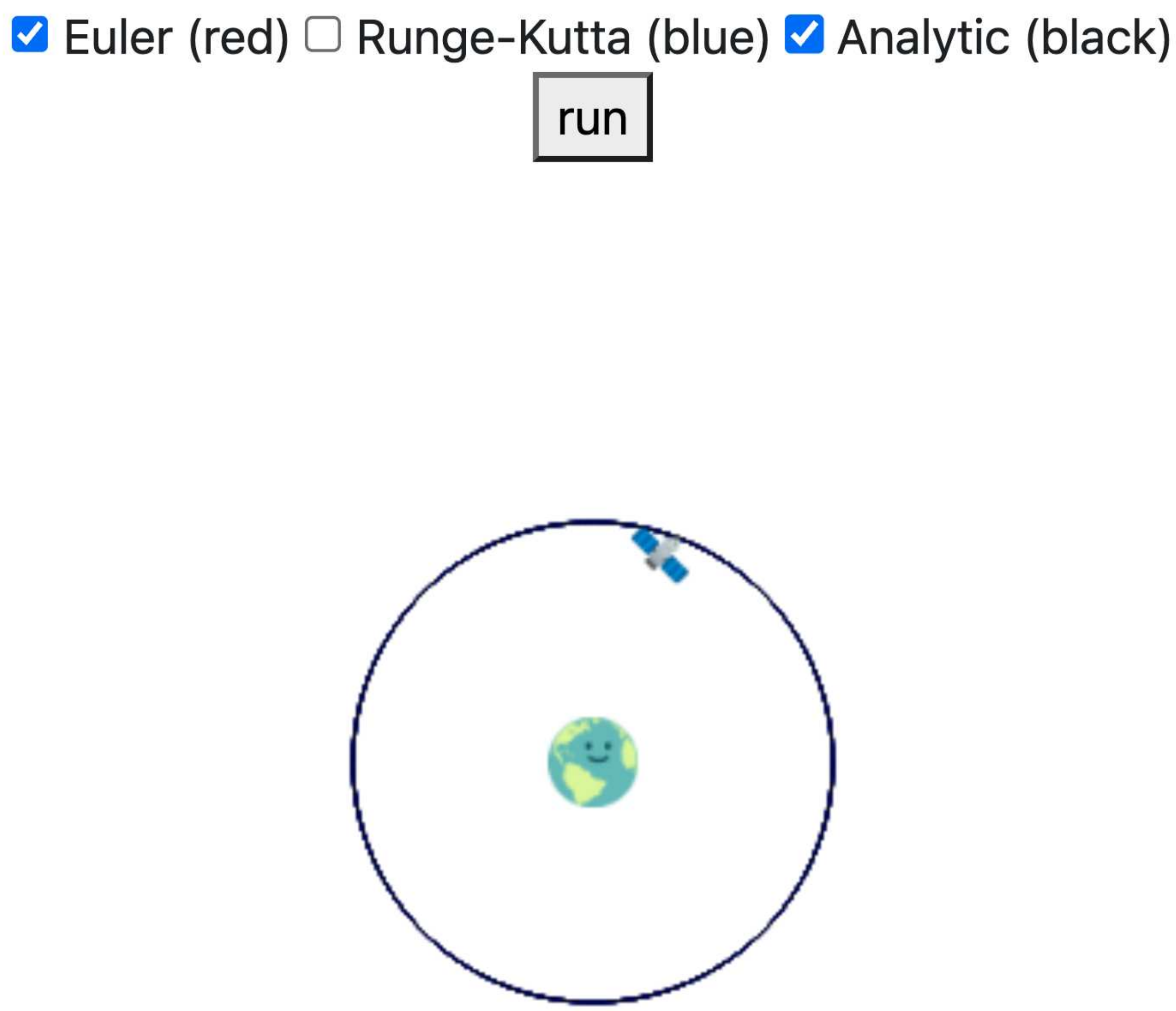
$$\vec{a} = \vec{g} - \frac{1}{2}\rho \frac{A C_d \vec{v}}{m} \|\vec{v}\|$$

Euler's method was used in *Hidden Figures*.

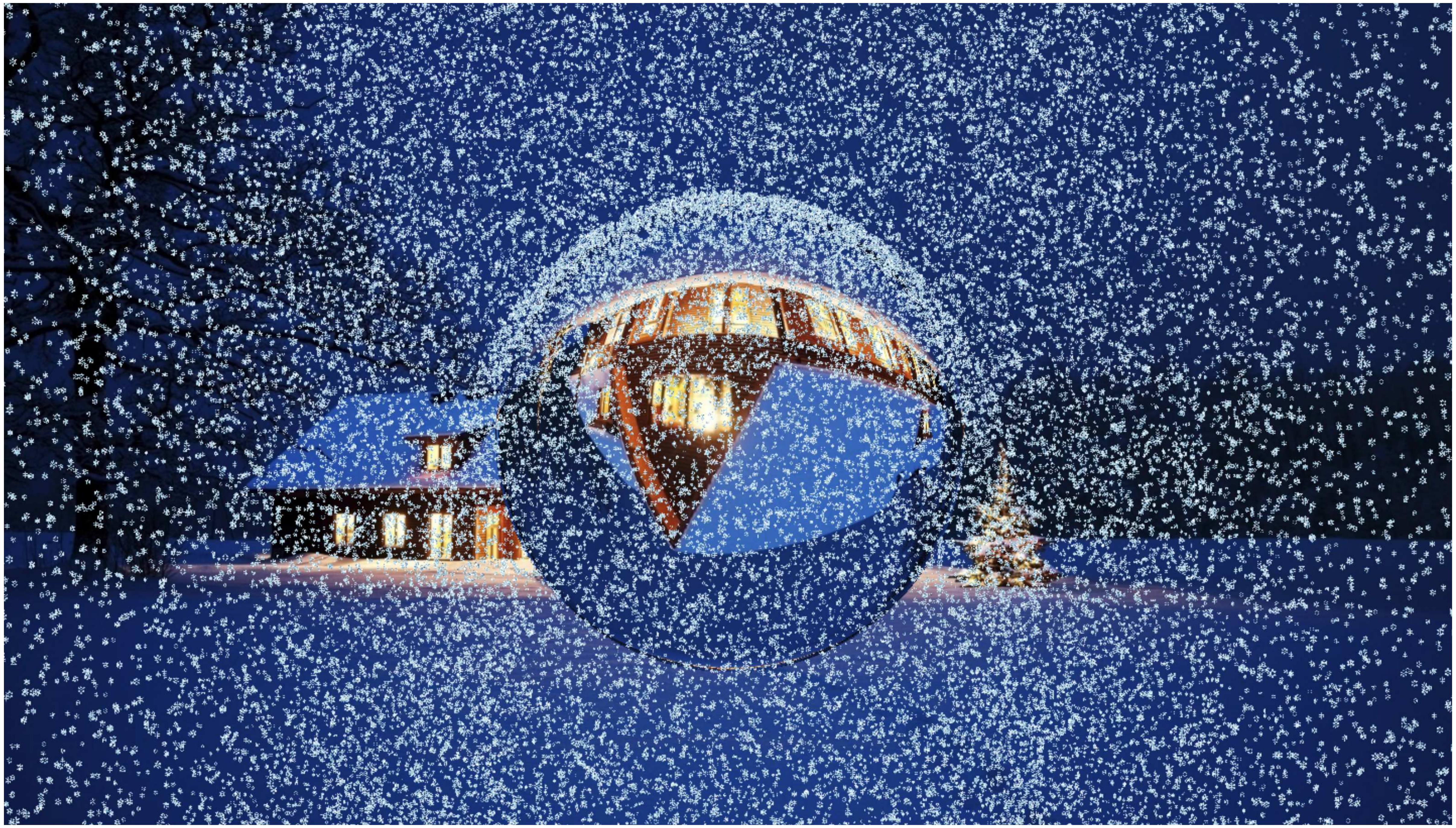
Euler's Method scene in Hidden Figures

Accuracy of Euler's method depends on time step Δt.

demo: https://philipclaude.github.io/csci461s25/chapter11



Our goal: animate thousands or millions of particles!



Draw particles with `gl.drawArrays` and `gl.POINTS`.

JS

```
gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
gl.vertexAttribPointer(a_Position, 3, gl.FLOAT, false, 0, 0);
gl.drawArrays(gl.POINTS, 0, nParticles);
```

new

VS

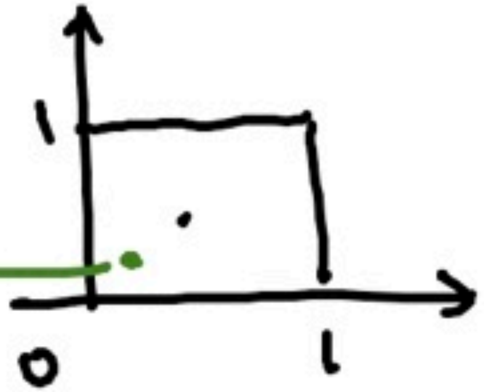
```
attribute vec4 a_Position;

uniform mat4 u_ProjectionMatrix;
uniform mat4 u_ViewMatrix;

void main() {
  gl_Position = u_ProjectionMatrix * u_ViewMatrix * vec4(a_Position, 1);
  gl_PointSize = 50.0 / gl_Position.w;
}
```

new

new

FS

```
uniform sampler2D tex_Sprite;

void main() {
  gl_FragColor = texture2D(tex_Sprite, gl_PointCoord);
  //gl_FragColor = vec4(1, 1, 1, 1);
}
```

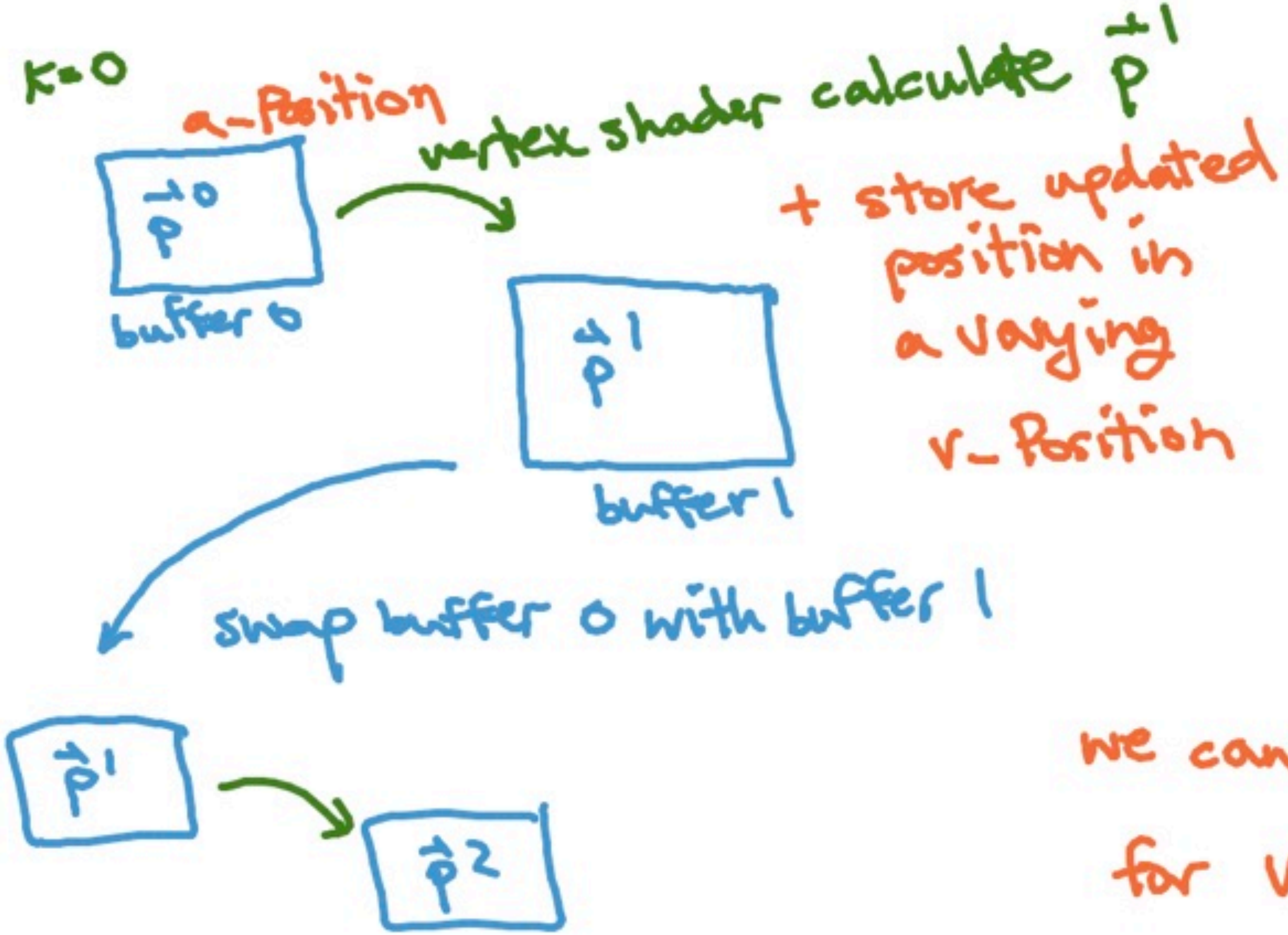


Complete particle animation might look like this:

```
// initialize particle positions and velocity
let nParticles = 1000;
let position = new Array(nParticles * 3);
// initialize position and velocity with random or known data...

// assume gl is some WebGLRenderingContext
const draw = (position) {
  let positionBuffer = gl.createBuffer();
  gl.bindBuffer(gl.ARRAY_BUFFER, positionBuffer);
  gl.bufferData(gl.ARRAY_BUFFER, new Float32Array(position), gl.STATIC_DRAW);
  gl.drawArrays(gl.POINTS, 0, nParticles);
}

const mass = 1; // some mass in kg
const nSteps = 1000;
const tFinal = 10;
const deltaT = tFinal / nSteps;

for (let k = 0; k < nSteps; k++) {
  draw(position);

  // for each particle, calculate the update
  for (let i = 0; i < nParticles; i++) {
    const ak = vec3.fromValue(0, -9.81, 0); // only gravity in this example
    for (let d = 0; d < 3; d++) {
      const vNext = velocity[3 * i + d] + deltaT * ak[d];
      const pNext = position[3 * i + d] + deltaT * velocity[3 * i + d];

      velocity[3 * i + d] = vNext;
      position[3 * i + d] = pNext;
    }
  }
}
```

calculate $\vec{p}^{k+1}$ $\vec{v}^{k+1}$

We'll use *transform feedback* to capture updated position & velocity to a buffer and swap buffers at every iteration.



① parallel updates of particle positions + velocities

② data is already on GPU for rendering.

we can use the same idea for velocity updates

Example: use transform feedback to animate particles.

- Part 1: declare varyings to capture.
- Part 2: create buffers for transform feedback.
- Part 3: calculate updated position and velocity in vertex shader.
  - Today, we'll use $\vec{p}^{k+1} = \vec{p}^k + \vec{v}^0 \Delta t$ with a *constant* velocity $\vec{v}^0$.
  - You will actually use Euler's method to update velocity in the lab on Thursday.
  - Only really working with position in this example (to practice with transform feedback).
- Part 4: draw and capture updated position in feedback buffer, then swap buffers.

Everything you need to include velocity updates is in the reading.

Recap of particle animation (with transform feedback).

$$\vec{v}^k = \frac{\Delta p}{\Delta t} = \frac{\vec{p}^{k+1} - \vec{p}^k}{\Delta t} \quad \rightarrow \quad \vec{p}^{k+1} = \vec{p}^k + \Delta t\, \vec{v}^k.$$

$$\vec{a}^k = \frac{\Delta v}{\Delta t} = \frac{\vec{v}^{k+1} - \vec{v}^k}{\Delta t} \quad \rightarrow \quad \vec{v}^{k+1} = \vec{v}^k + \Delta t\, \vec{a}^k.$$

$\vec{p}^{k+1}$: v_Position; $\vec{p}^k$: a_Position; $\vec{v}^k$: a_Velocity

$\vec{v}^{k+1}$: v_Velocity; $\vec{v}^k$: a_Velocity; $\vec{a}^k$: compute within shader based on physics.

y (up)

$\vec{f}_d \uparrow$, $\vec{v} \downarrow$

$\vec{f}_g = m\vec{g}$ $\quad \vec{g} = \begin{bmatrix} 0 \\ -9.81 \\ 0 \end{bmatrix}$

$\vec{f}_d \sim \|\vec{v}\|^2$

$$\vec{f}_d = -\frac{1}{2}\rho A C_d \|\vec{v}\|^2 \frac{\vec{v}}{\|\vec{v}\|} = -\frac{1}{2}\rho A C_d \|\vec{v}\|\,\vec{v}$$

$$\vec{a} = \frac{\sum \vec{f}}{m} = \frac{m\vec{g} - \frac{1}{2}\rho A C_d \|\vec{v}\|\,\vec{v}}{m} = \vec{g} - \frac{\frac{1}{2}\rho A C_d \|\vec{v}\|\,\vec{v}}{m}$$

Summary

- Use Euler's method to update motion,
- Be careful with time step Δt,
- Use transform feedback to keep everything in the GPU,
- Can even use transform feedback for GPGPU programming,
- More complicated when there are interparticle forces (e.g. springs) - next week!