

# Effects and animation.

## Part 2.



Particle systems.





Crouch



Put Away



080









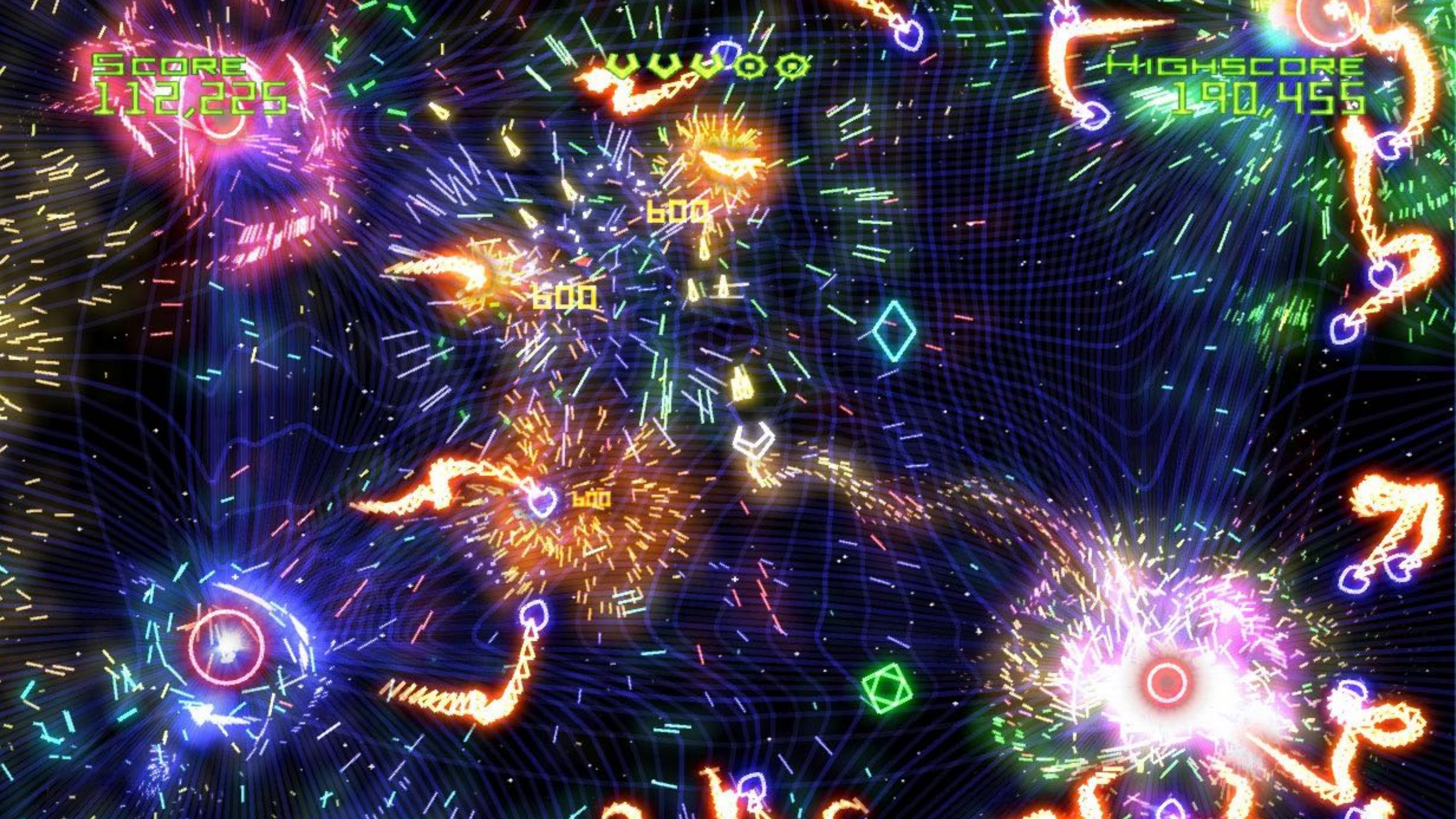






SCORE  
112,225

HIGHSCORE  
190,455









*Anatomy of a basic particle system.*

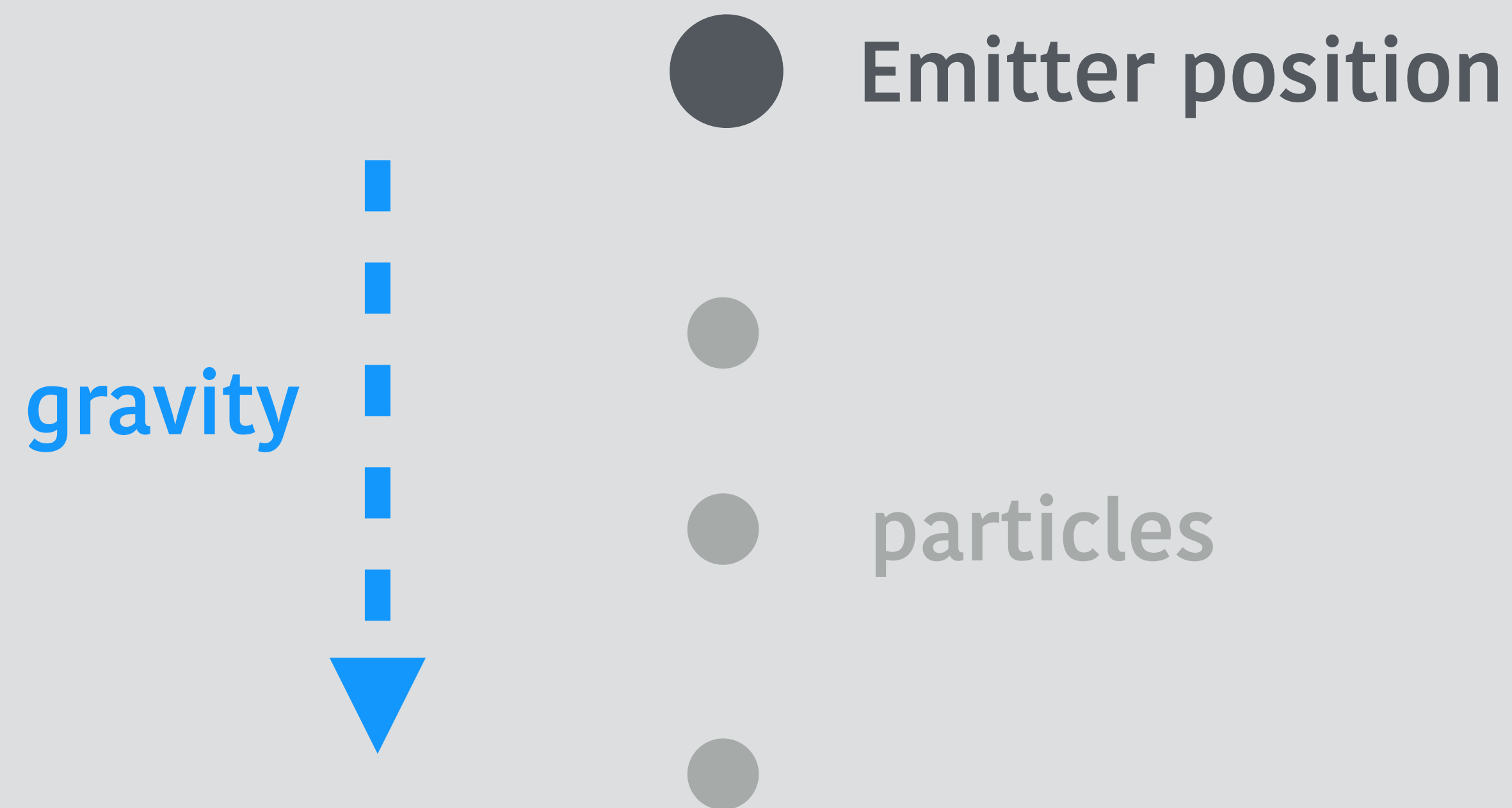


A dripping faucet.





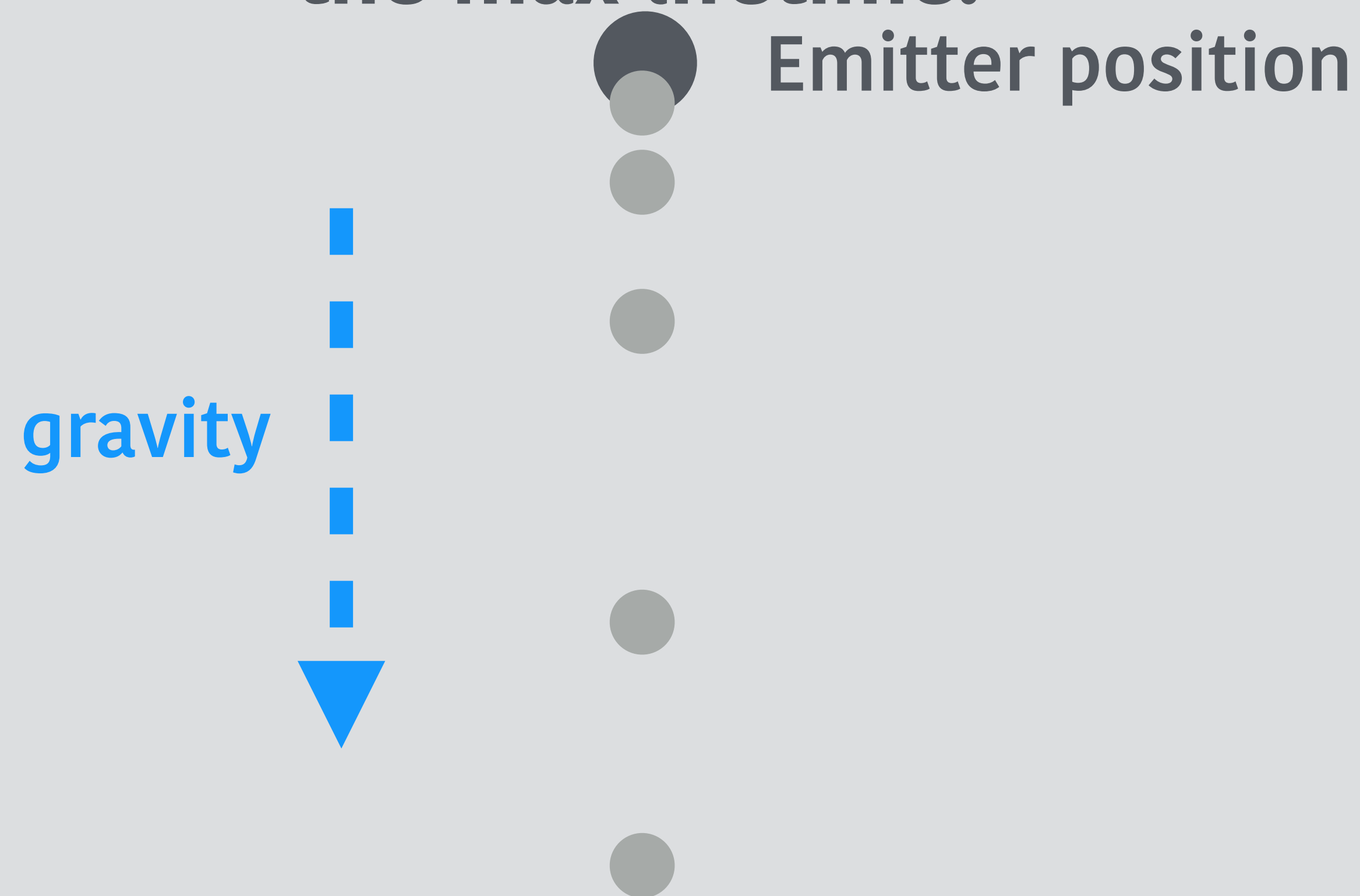
An **emitter** has an **emitter position** and contains an **array of particles**, which have a **position**, **velocity** and a **lifetime**. Their only **acceleration** is the particle emitter's **gravity** and we **don't care about friction**.





For **each particle**, adjust **velocity** and **position** based on **elapsed time** and add **elapsed time** to the **particle's lifetime**.

When a particle's **lifetime exceeds** the particle system's **max lifetime**, it is **reset to the emitter's position** (and reset its **velocity**) and its **lifetime is set to the remainder amount over the max lifetime**.





# Particle Emitter

```
class ParticleEmitter {  
    public:  
        ParticleEmitter(unsigned int particleCount);  
  
        ParticleEmitter();  
        ~ParticleEmitter();  
  
        void Update(float elapsed);  
        void Render();  
  
        glm::vec3 position;  
        glm::vec3 gravity;  
        float maxLifetime;  
  
        std::vector<Particle> particles;  
};
```

```
class Particle {  
    public:  
        glm::vec3 position;  
        glm::vec3 velocity;  
        float lifetime;  
};
```



**Make sure the particles' lifetime is set to a random value up to the maximum lifetime, so they don't reset all together!**



Drawing the particles.



To draw vertices as single points we can pass **GL\_POINTS** as the first argument to `glDrawArrays`.

(You can set how many pixels each point is using the `glPointSize` function).



```
std::vector<float> vertices;
```

```
for(int i=0; i < particles.size(); i++) {  
    vertices.push_back(particles[i].position.x);  
    vertices.push_back(particles[i].position.y);  
}
```

```
glVertexAttribPointer(program.positionAttribute, 2, GL_FLOAT, false, 0, vertices.data());  
glEnableVertexAttribArray(program.positionAttribute);
```

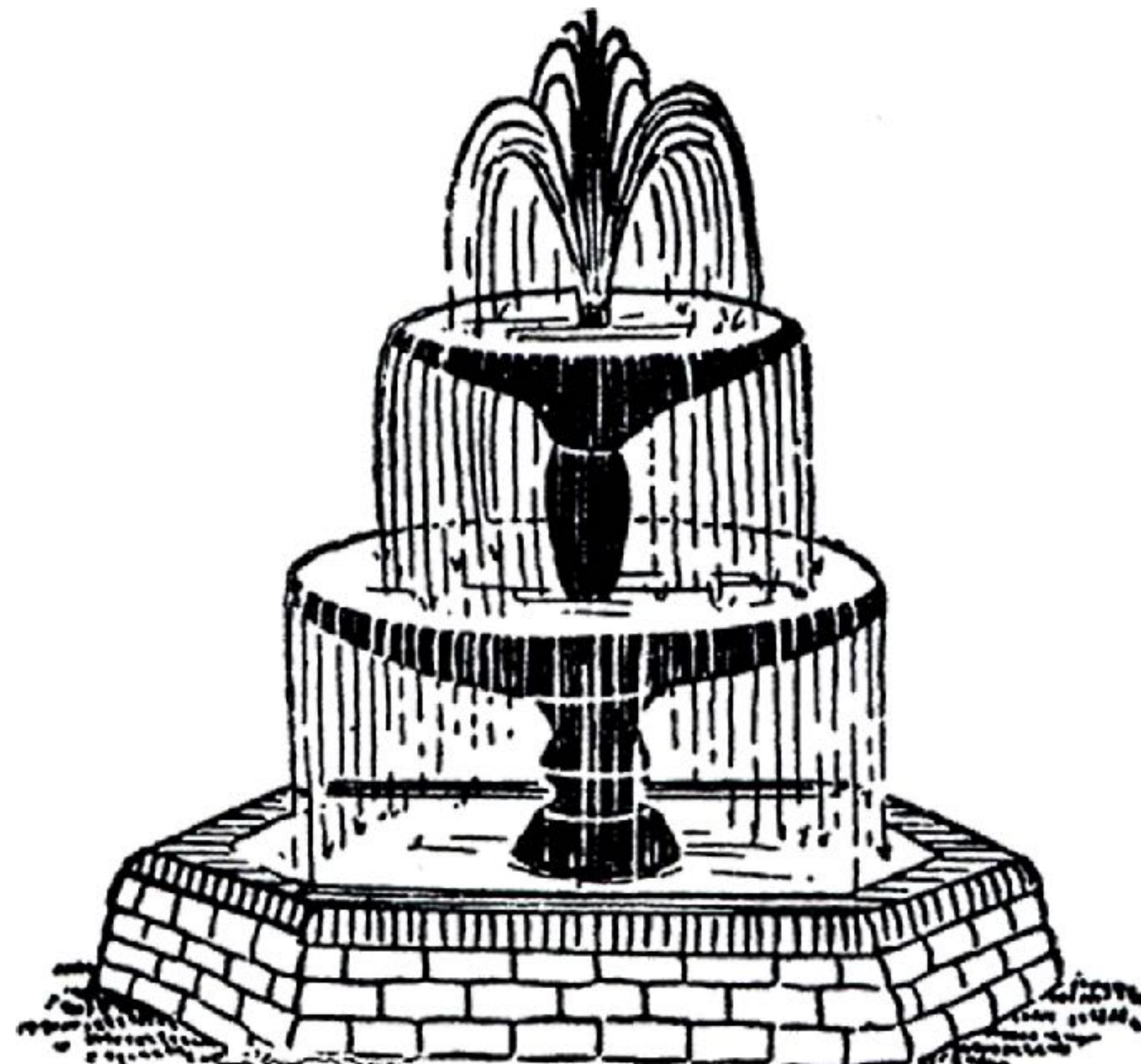
```
glDrawArrays(GL_POINTS, 0, particles.size());
```



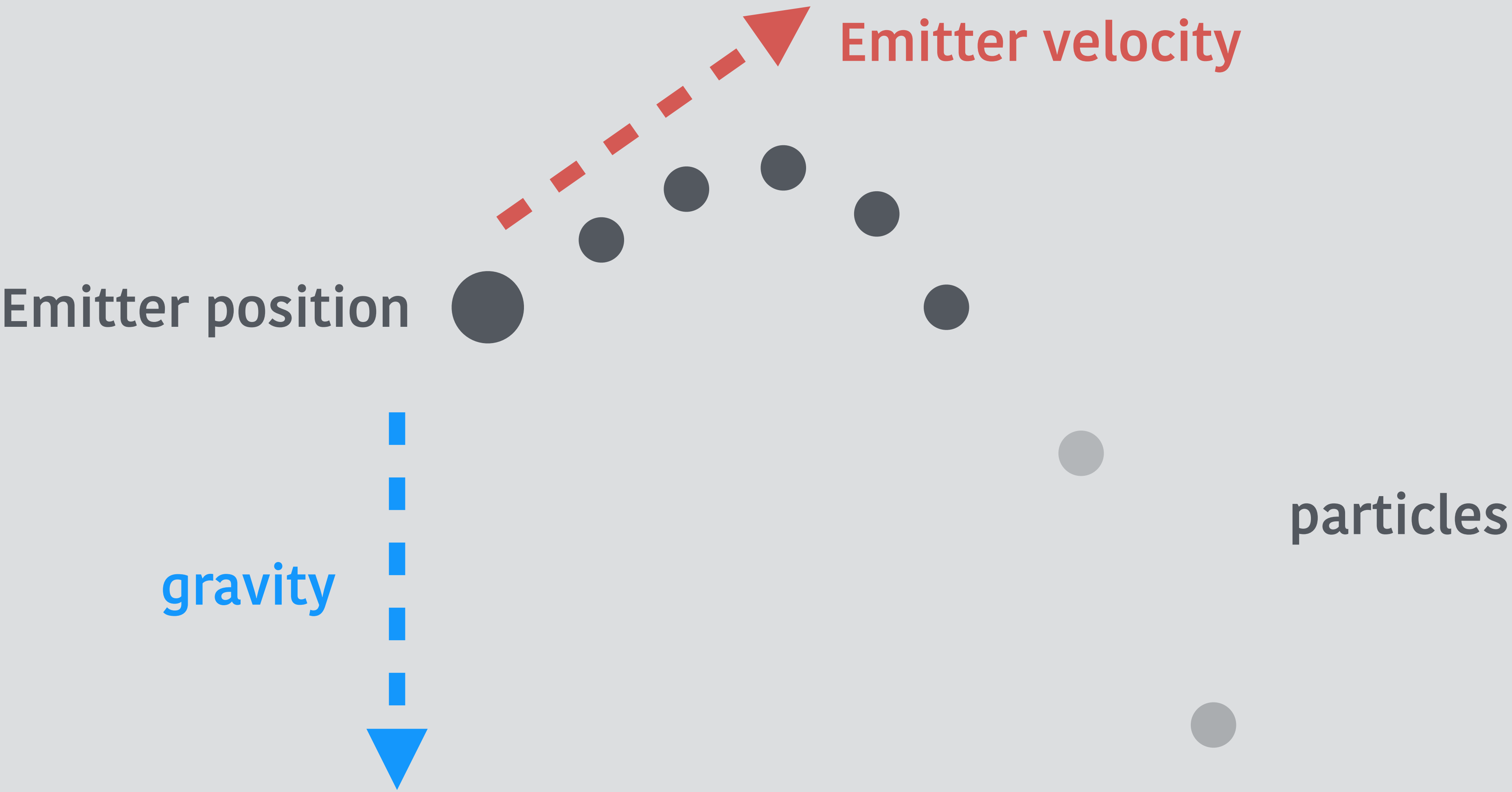
Launching particles.



**A fountain.**









Let's add a **velocity** vector to the emitter.

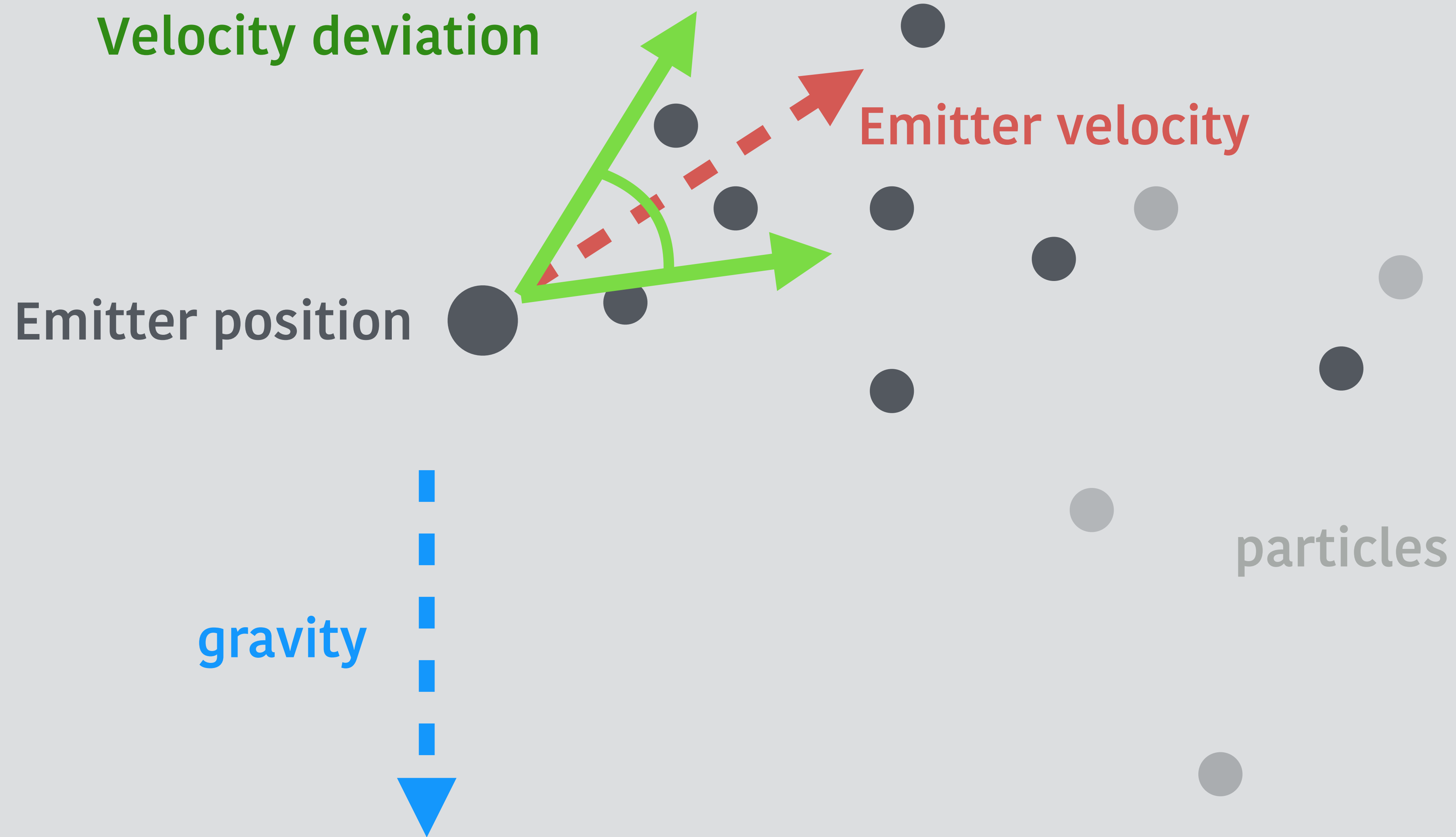
```
class ParticleEmitter {  
    public:  
        // . . .  
        glm::vec3 velocity;  
};
```

When resetting particles, let's now **set their velocity** to the **emitter's velocity**.



**We don't always want to shoot all  
particles in a straight line.**







# Let's add a velocity deviation.

```
class ParticleEmitter {  
    public:  
  
    // . . .  
    glm::vec3 velocity;  
    glm::vec3 velocityDeviation;
```

When setting the particles' velocity to the emitter's velocity, let's **add a random value within the velocityDeviation** to the velocity.

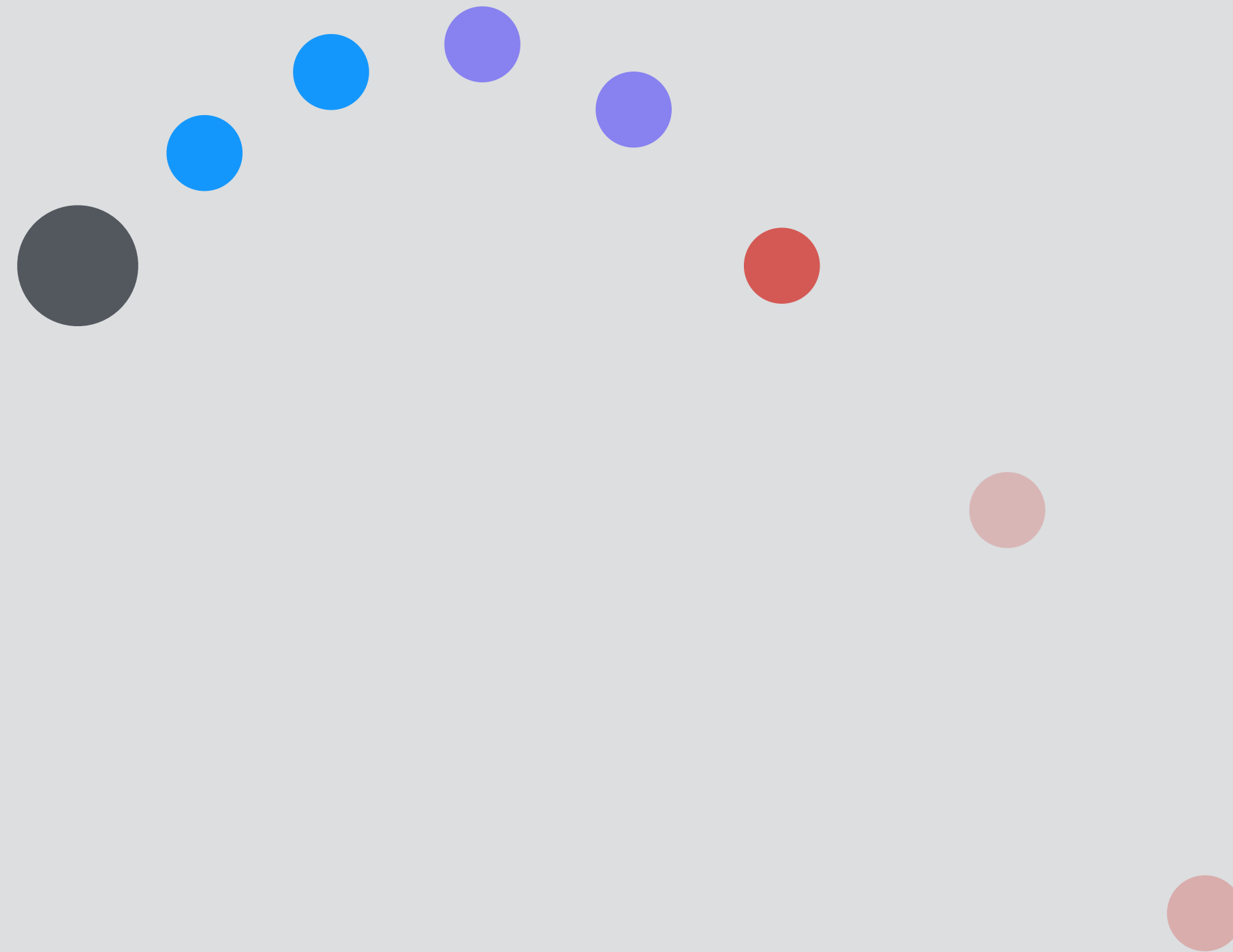


# Adding color





Let's add a **starting** and **ending colors** to the emitter,  
so the particles can **change color as they travel**.





```
class ParticleEmitter {  
    public:  
  
        // . . .  
        glm::vec4 startColor;  
        glm::vec4 endColor;  
};
```

**LERP** the **color** based on the **percentage of particle lifetime to the maximum lifetime**.

```
std::vector<float> particleColors;  
  
for(int i=0; i < particles.size(); i++) {  
    float relativeLifetime = (particles[i].lifetime/maxLifetime);  
    particleColors.push_back(lerp(startColor.r, endColor.r, relativeLifetime));  
    particleColors.push_back(lerp(startColor.g, endColor.g, relativeLifetime));  
    particleColors.push_back(lerp(startColor.b, endColor.b, relativeLifetime));  
    particleColors.push_back(lerp(startColor.a, endColor.a, relativeLifetime));  
}
```



Using vertex colors.



```
attribute vec4 position;  
attribute vec4 color;  <img alt="blue dashed arrow pointing left" data-bbox="265 90 437 125"/>
```

```
uniform mat4 modelviewMatrix;  
uniform mat4 projectionMatrix;
```

```
varying vec4 vertexColor;  <img alt="blue dashed arrow pointing left" data-bbox="298 240 469 277"/>
```

```
void main()  
{  
    vec4 p = modelviewMatrix * position;  
    vertexColor = color;  <img alt="blue dashed arrow pointing left" data-bbox="265 395 437 430"/>  
    gl_Position = projectionMatrix * p;  
}
```

```
varying vec4 vertexColor;  <img alt="blue dashed arrow pointing left" data-bbox="288 690 460 725"/>
```

```
void main()  
{  
    gl_FragColor = vertexColor;  <img alt="blue dashed arrow pointing left" data-bbox="350 810 520 845"/>  
}
```



# Get the color attribute location.

```
GLuint colorAttribute = glGetAttribLocation(program->programID, "color");
```

# Set the attribute pointer to the color data vector.

```
glVertexAttribPointer(colorAttribute, 4, GL_FLOAT, false, 0, colors.data());  
glEnableVertexAttribArray(colorAttribute);
```

Color deviation.



We can add a color deviation to the emitter and particle class.  
Every time we reset a particle, we set its color deviation to a random value within the color deviation range.

```
particles[i].colorDeviation.r = (-colorDeviation.r * 0.5) + (colorDeviation.r *  
((float)rand() / (float)RAND_MAX));
```

Then, when we LERP our color values, we can add the deviation for each color component.

```
particleColors.push_back(lerp(startColor.r, endColor.r, relativeLifetime) +  
particles[i].colorDeviation.r);
```

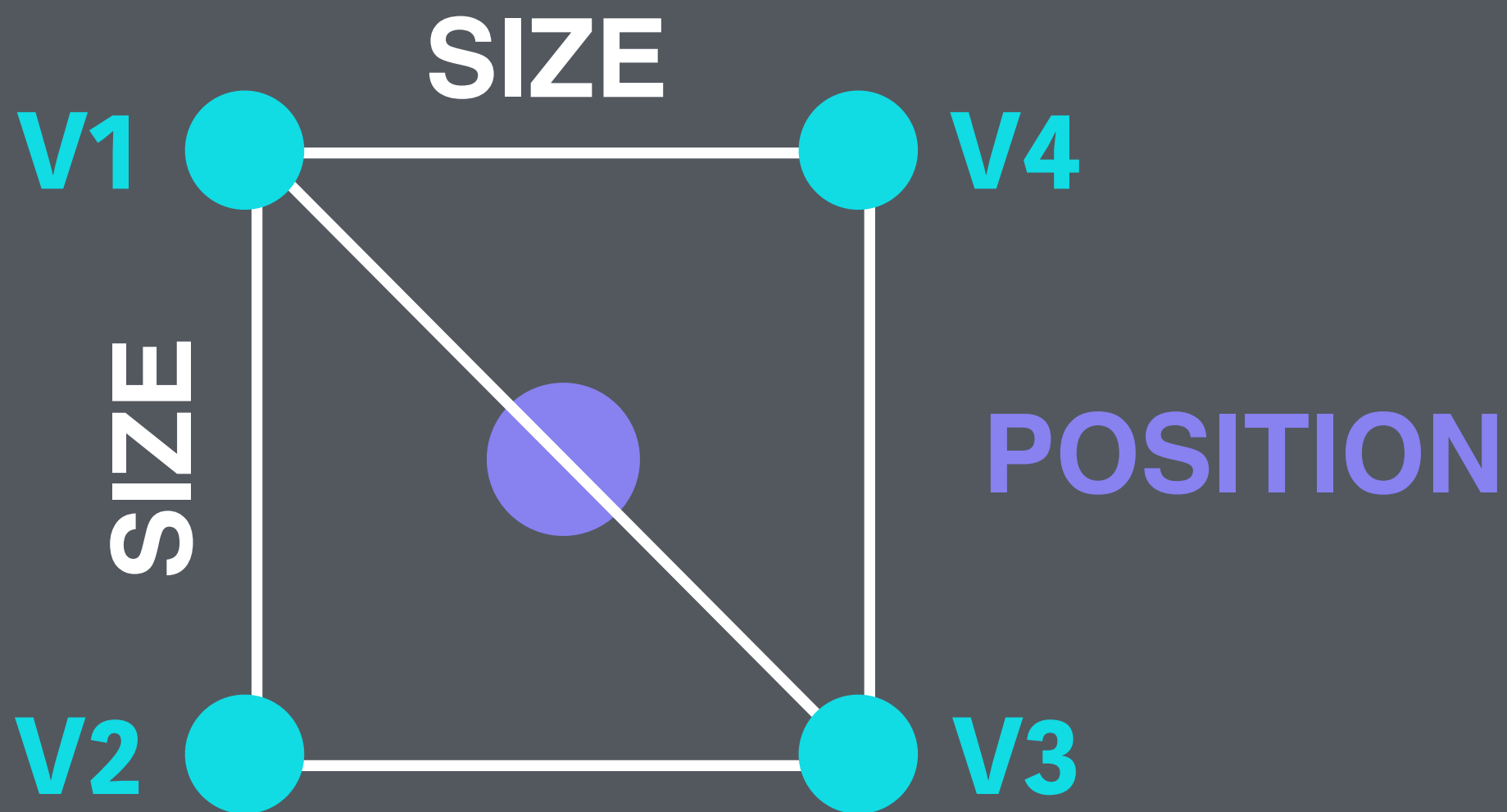
Textured particles.





Instead of points, let's draw textured **GL\_TRIANGLES**.





Particles now need a **size**.

Like color, we can define **starting** and **ending sizes** and lerp between them when we draw our triangles.



Like colors, we can also add a **size deviation** to the particles and set it to a **random value** within the emitter size deviation range.

```
class ParticleEmitter {
    public:

        // . . .
        float startSize;
        float endSize;
        float sizeDeviation;

class Particle {
    public:
        // . . .
        float sizeDeviation;
};
```

```

for(int i=0; i < particles.size(); i++) {
    float m = (particles[i].lifetime/maxLifetime);
    float size = lerp(startSize, endSize, m) + particles[i].sizeDeviation;

    vertices.insert(vertices.end(), {
        particles[i].position.x - size, particles[i].position.y + size,
        particles[i].position.x - size, particles[i].position.y - size,
        particles[i].position.x + size, particles[i].position.y + size,

        particles[i].position.x + size, particles[i].position.y + size,
        particles[i].position.x - size, particles[i].position.y - size,
        particles[i].position.x + size, particles[i].position.y - size
    });

    texCoords.insert(texCoords.end(), {
        0.0f, 0.0f,
        0.0f, 1.0f,
        1.0f, 0.0f,

        1.0f, 0.0f,
        0.0f, 1.0f,
        1.0f, 1.0f
    });

    for(int j=0; j < 6; j++) {
        colors.push_back(lerp(startColor.r, endColor.r, m));
        colors.push_back(lerp(startColor.g, endColor.g, m));
        colors.push_back(lerp(startColor.b, endColor.b, m));
        colors.push_back(lerp(startColor.a, endColor.a, m));
    }
}

```



```
glVertexAttribPointer(program->positionAttribute, 2, GL_FLOAT, false, 0, vertices.data());
glEnableVertexAttribArray(program->positionAttribute);

glVertexAttribPointer(colorAttribute, 4, GL_FLOAT, false, 0, colors.data());
glEnableVertexAttribArray(colorAttribute);

glVertexAttribPointer(program->texCoordAttribute, 2, GL_FLOAT, false, 0, texCoords.data());
glEnableVertexAttribArray(program->texCoordAttribute);

glDrawArrays(GL_TRIANGLES, 0, vertices.size()/2);
```

```
attribute vec4 position;
attribute vec4 color;
attribute vec2 texCoord;

uniform mat4 modelviewMatrix;
uniform mat4 projectionMatrix;

varying vec4 vertexColor;
varying vec2 varTexCoord;

void main()
{
    vec4 p = modelviewMatrix * position;
    vertexColor = color;
    varTexCoord = texCoord;
    gl_Position = projectionMatrix * p;
}

uniform sampler2D diffuse;

varying vec4 vertexColor;
varying vec2 varTexCoord;

void main()
{
    gl_FragColor = texture2D(diffuse, varTexCoord) * vertexColor;
}
```



Particle **rotation**.

Particles now have a **rotation property**, that is increased based on time elapsed for each particle.

```
class Particle {  
    public:  
        // . . .  
        float rotation;  
};
```



Since we are drawing as a single array,  
we cannot use the model matrix to rotate and need to  
rotate the vertices manually.

$$\begin{bmatrix} \cos\theta & -\sin\theta & 0 & 0 \\ \sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$\begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} x\cos\theta - y\sin\theta \\ x\sin\theta + y\cos\theta \end{bmatrix}$$



```
float cosTheta = cosf(particles[i].rotation);
float sinTheta = sinf(particles[i].rotation);

float TL_x = cosTheta * -size - sinTheta * size;
float TL_y = sinTheta * -size + cosTheta * size;

float BL_x = cosTheta * -size - sinTheta * -size;
float BL_y = sinTheta * -size + cosTheta * -size;

float BR_x = cosTheta * size - sinTheta * -size;
float BR_y = sinTheta * size + cosTheta * -size;

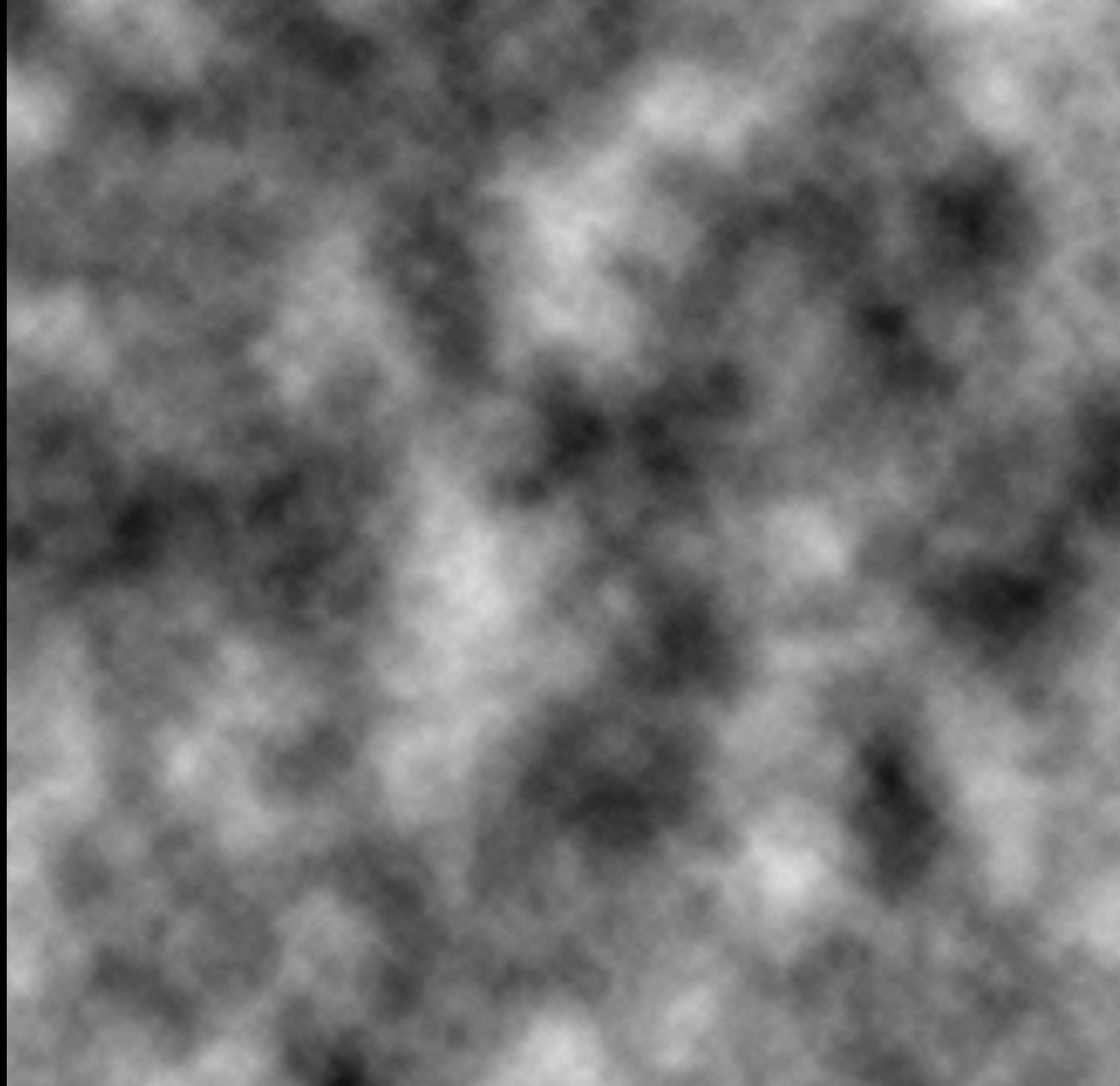
float TR_x = cosTheta * size - sinTheta * size;
float TR_y = sinTheta * size + cosTheta * size;

vertices.insert(vertices.end(), {
    particles[i].position.x + TL_x, particles[i].position.y + TL_y,
    particles[i].position.x + BL_x, particles[i].position.y + BL_y,
    particles[i].position.x + TR_x, particles[i].position.y + TR_y,

    particles[i].position.x + TR_x, particles[i].position.y + TR_y,
    particles[i].position.x + BL_x, particles[i].position.y + BL_y,
    particles[i].position.x + BR_x, particles[i].position.y + BR_y
});
```

Natural particle motion using perlin noise.





Add a variable to particles to vary their Perlin noise Y-position and set it to some random value.

```
class Particle {  
    public:  
        // . . .  
        float perlinY;  
};
```

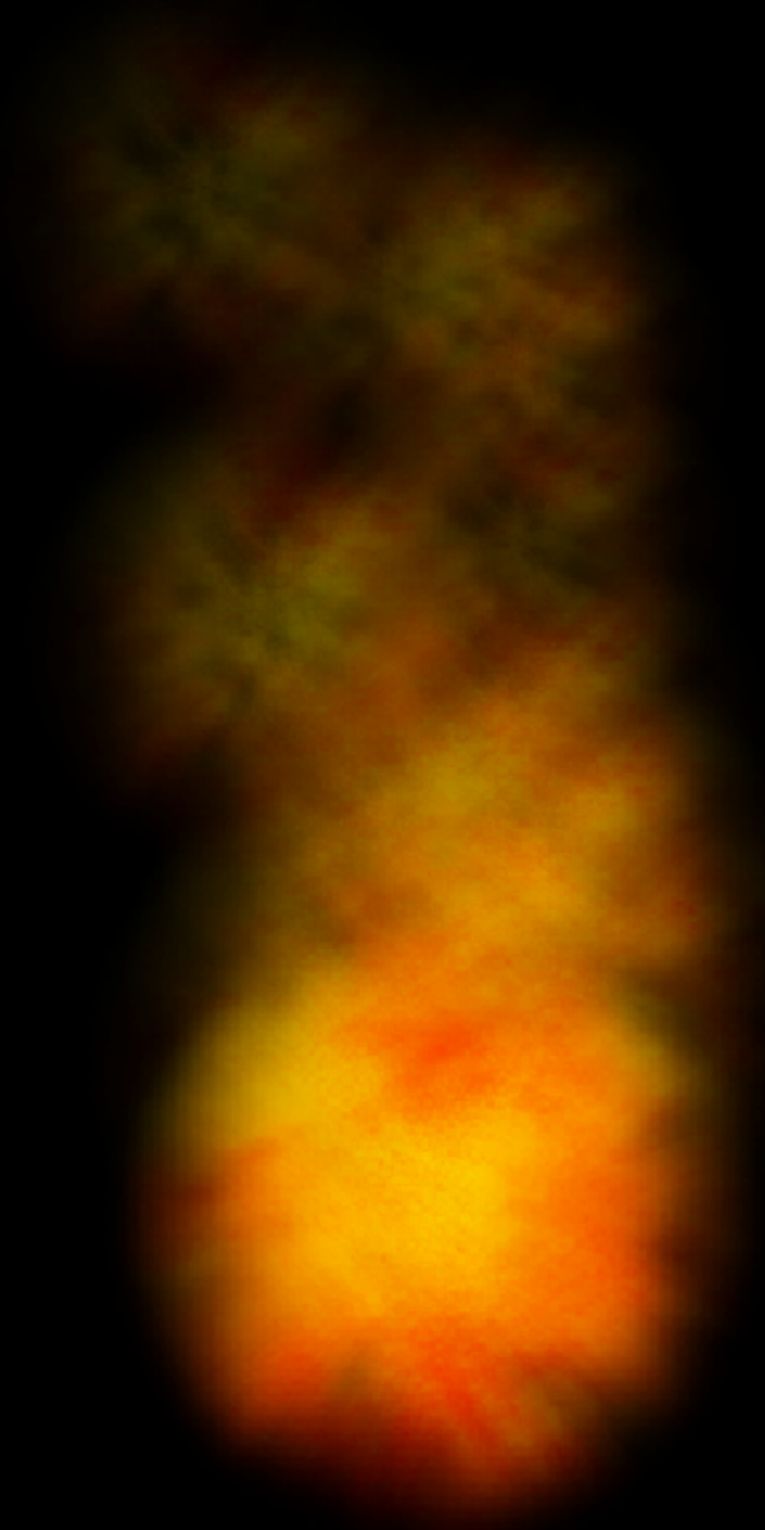
Use the Perlin noise value from their Y-position and current X-position to modify the particles' movement.

```
perlinValue += elapsed;  
// ...  
  
float coord[2] = {perlinValue, particles[i].perlinY};  
particles[i].position.x += noise2(coord) * perlinSize;  
coord[0] = perlinValue * 0.5f;  
particles[i].position.y += noise2(coord) * perlinSize;
```

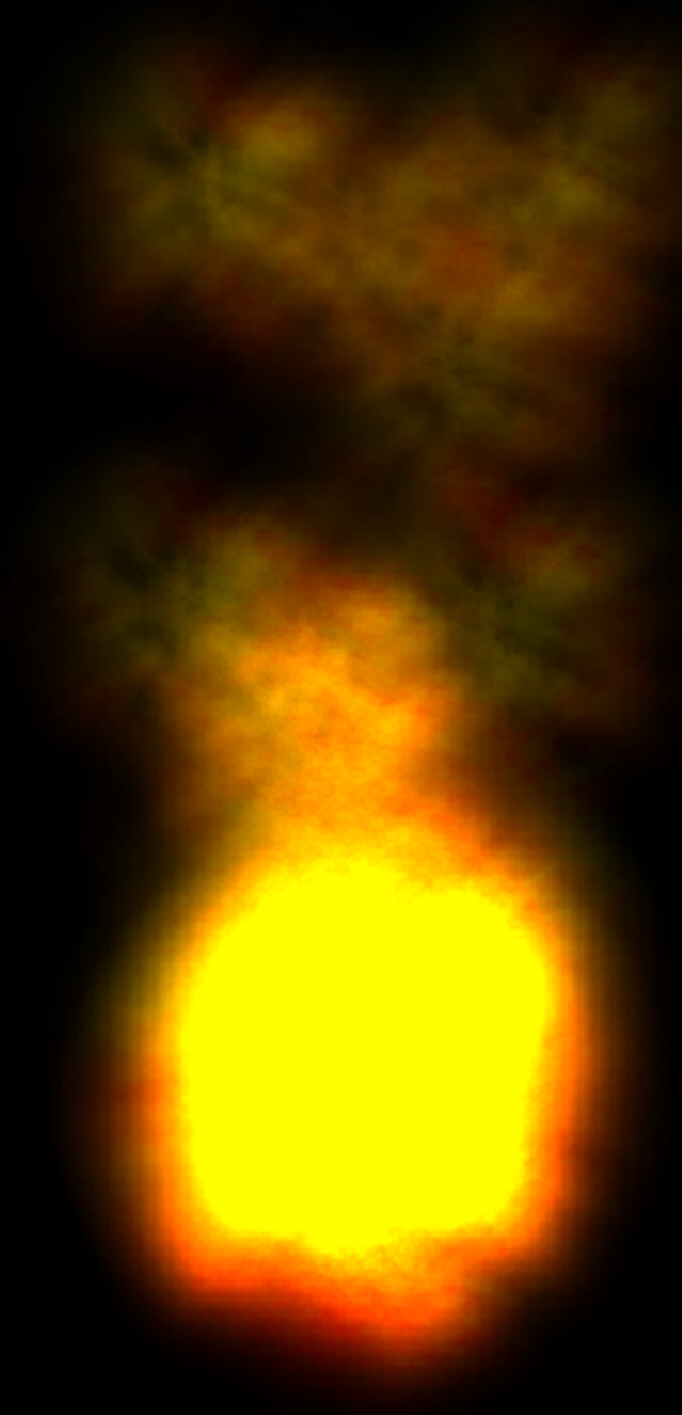


**Additive blending.**

Regular



Additive





Alpha blending

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

Additive blending

```
glBlendFunc (GL_SRC_ALPHA, GL_ONE);
```

Changing emitter size.

**When resetting a particle, set its position to emitter position + random value within emitter size range.**



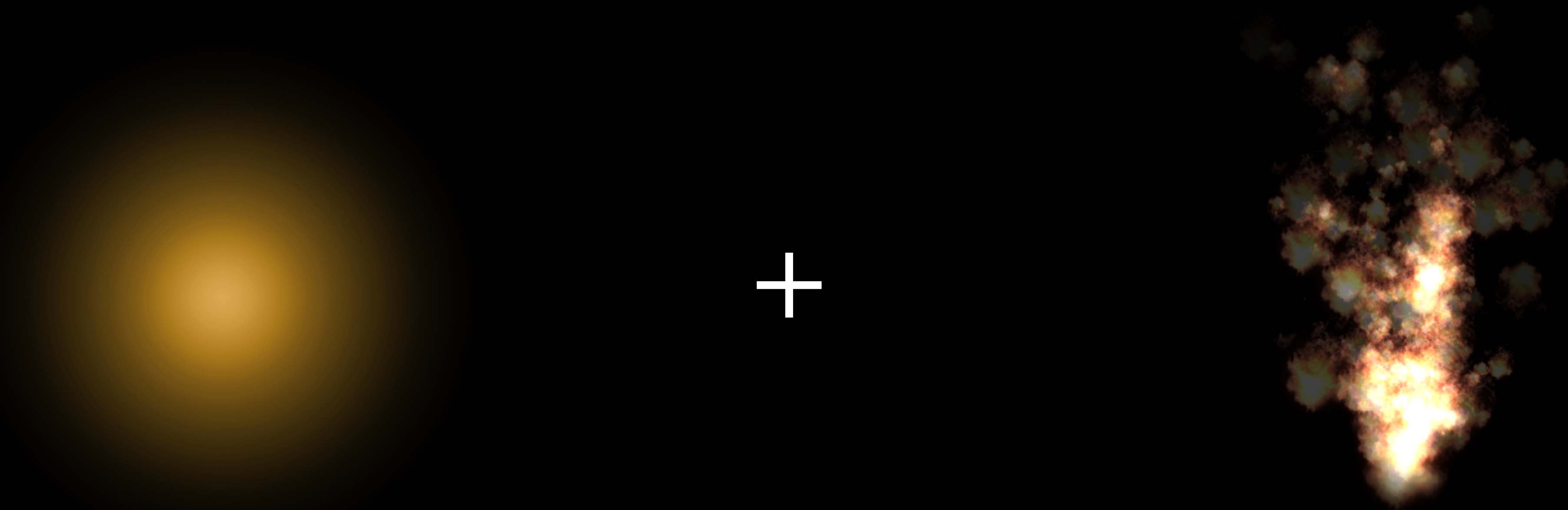
Triggered particle emitters.

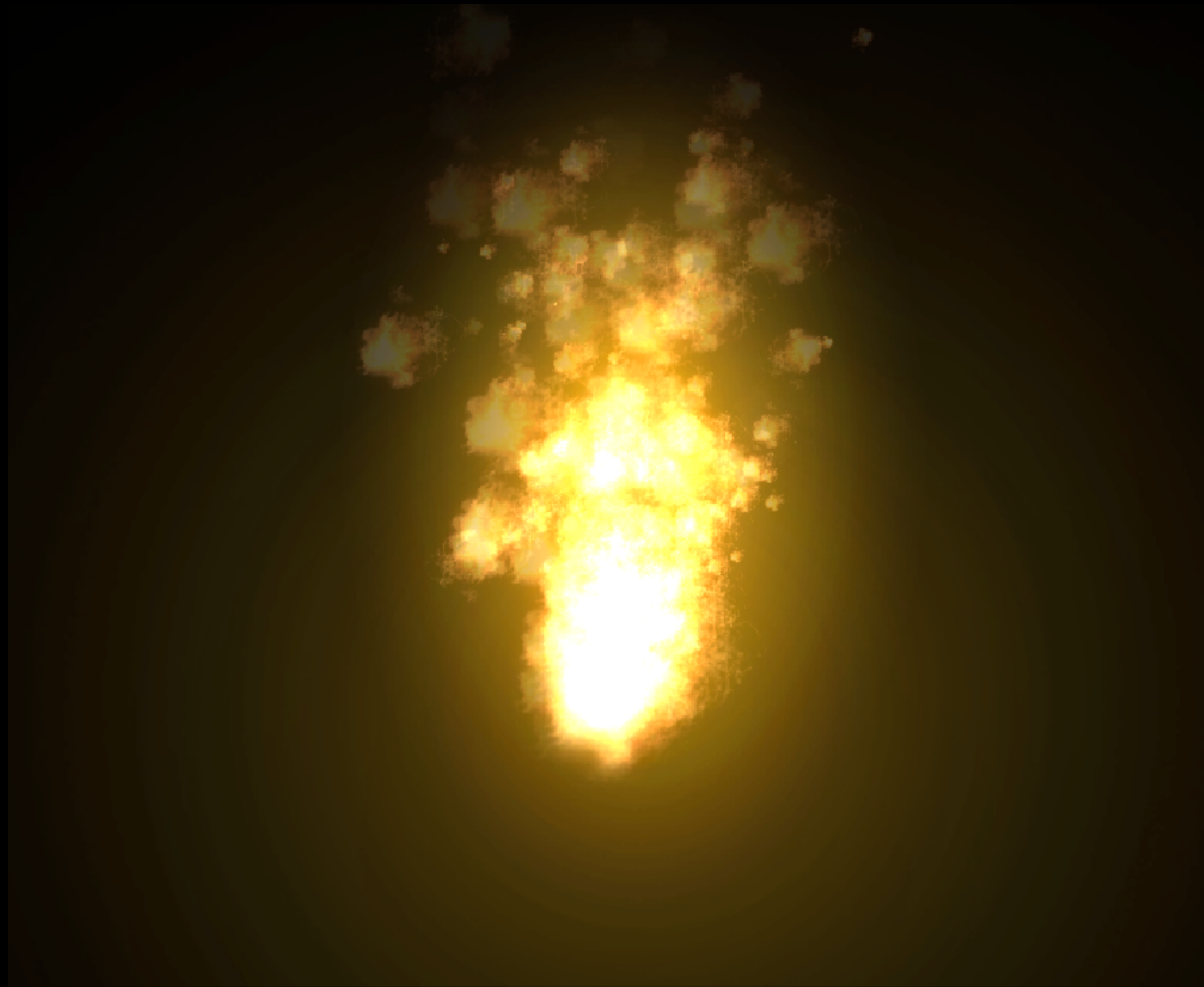
For a triggered emitter, simply do not reset particles after their lifetime is maxed out. Add a Trigger method to the emitter, to reset all of the particles instead.

Making things glow.



Drawing a glow texture using additive blending on top of other graphics.





# Final project requirements.

- Must have a title screen and proper mode for game over, etc.
- Must have a way to quit the game.
- Must have music and sound effects.
- Must have at least 3 different levels or be procedurally generated.
- Must be either local multiplayer or have AI (or both!).
- Must have at least some animation or particle effects.



# Bonus points for...

- Getting it running on your phone.
- Having 3D elements.
- Having shader effects.

(we haven't covered any of this yet!)