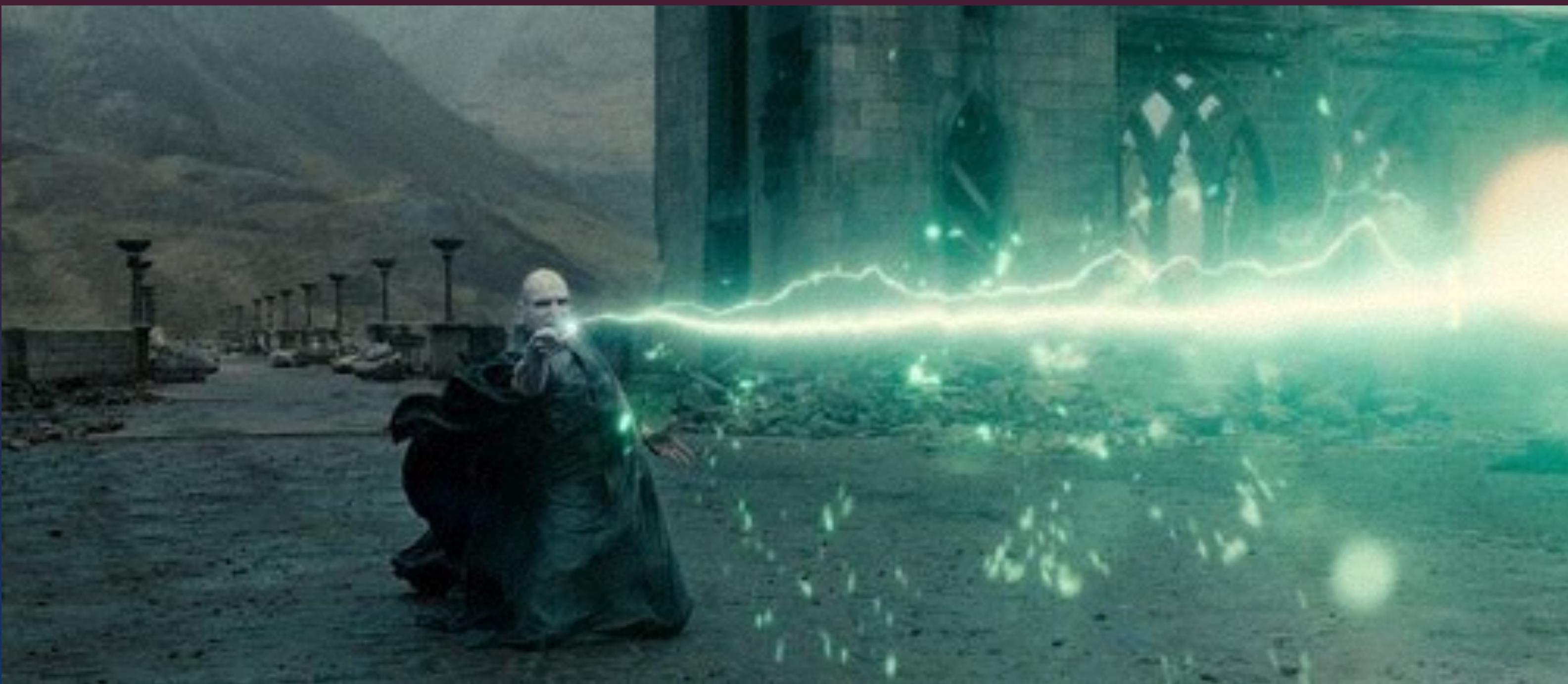


Effects and animation.

Part 2.

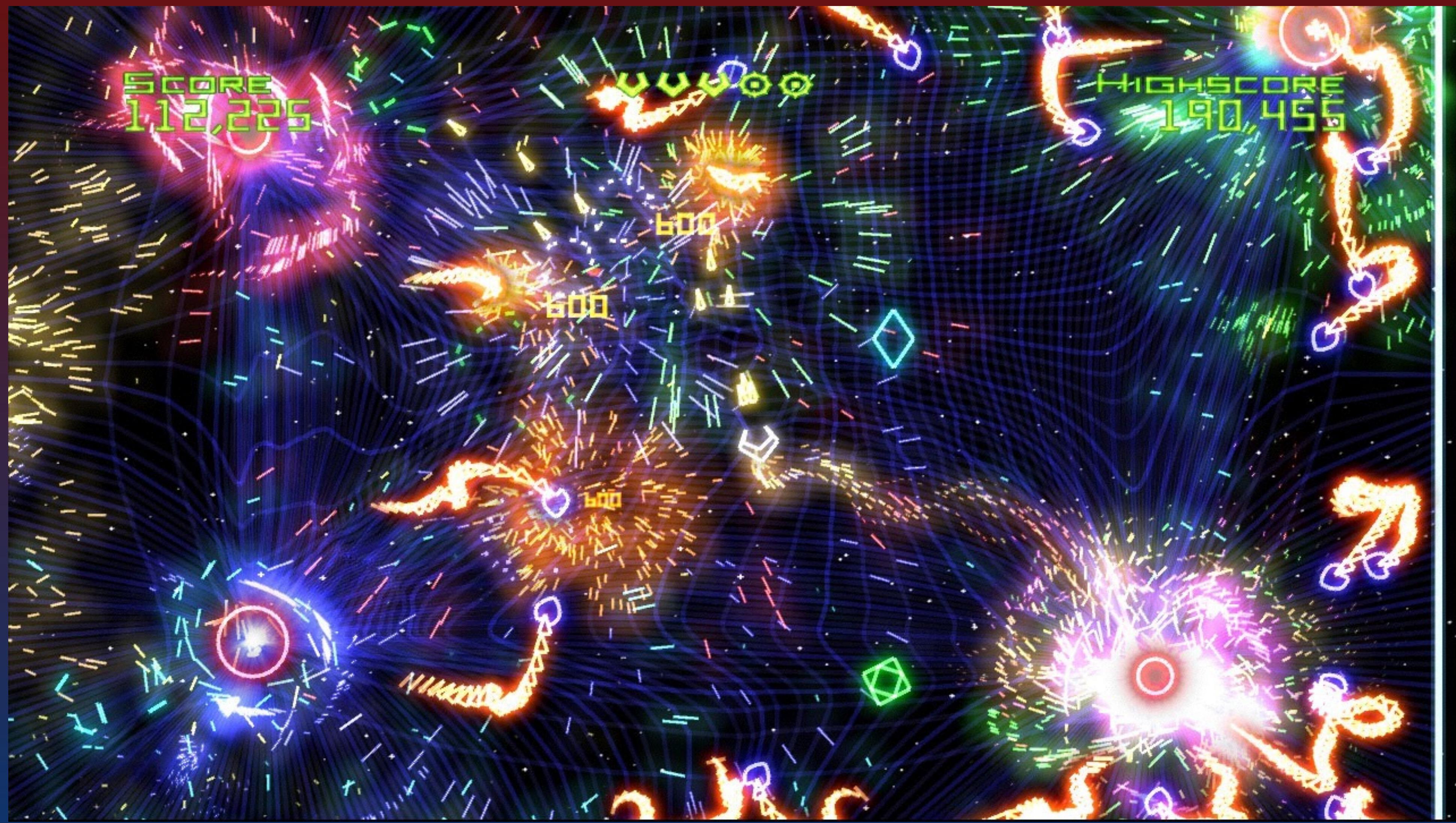


Particle systems.











Game
reactor

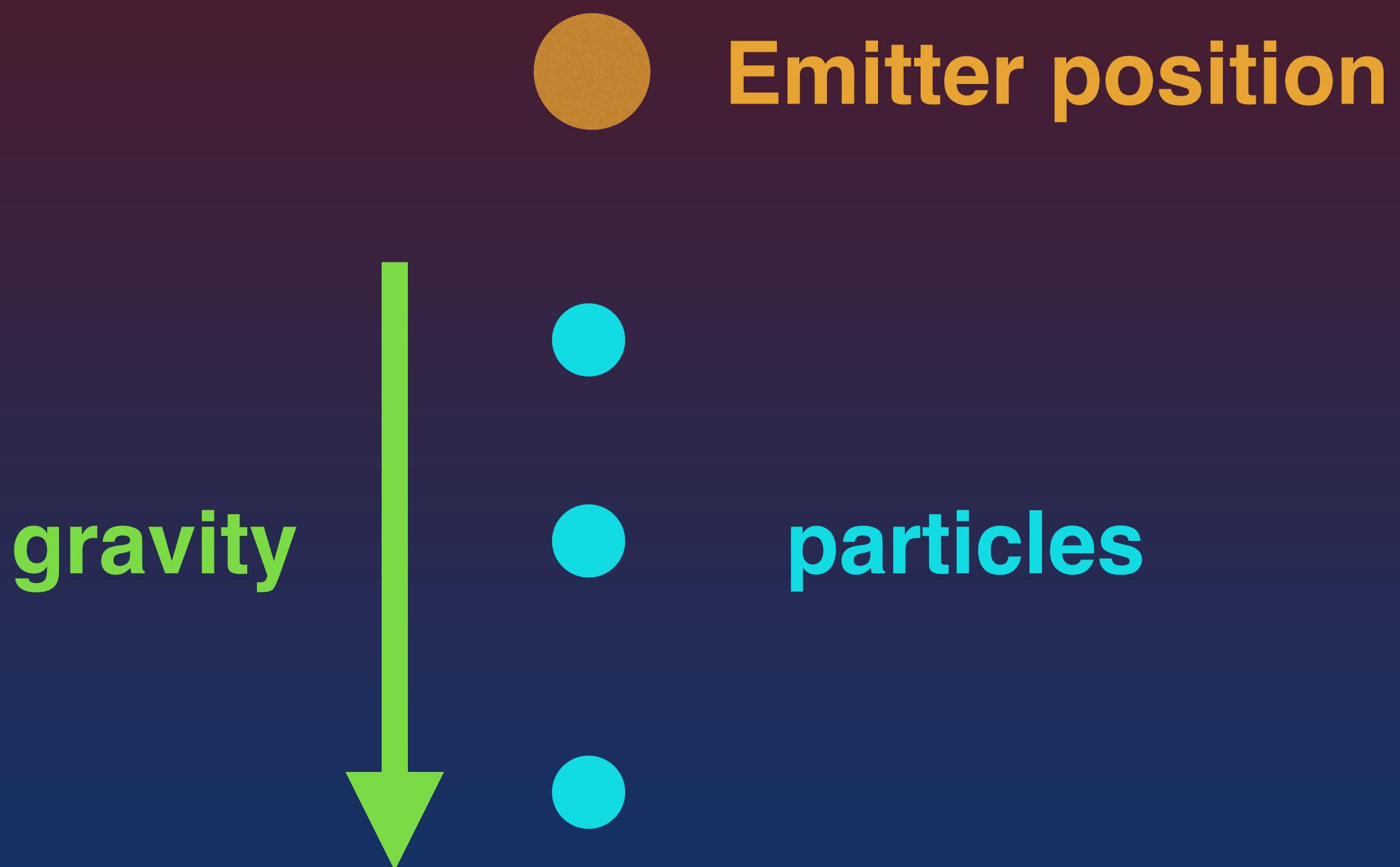
www.gamereactor.se/dk/no

Anatomy of a basic particle system.

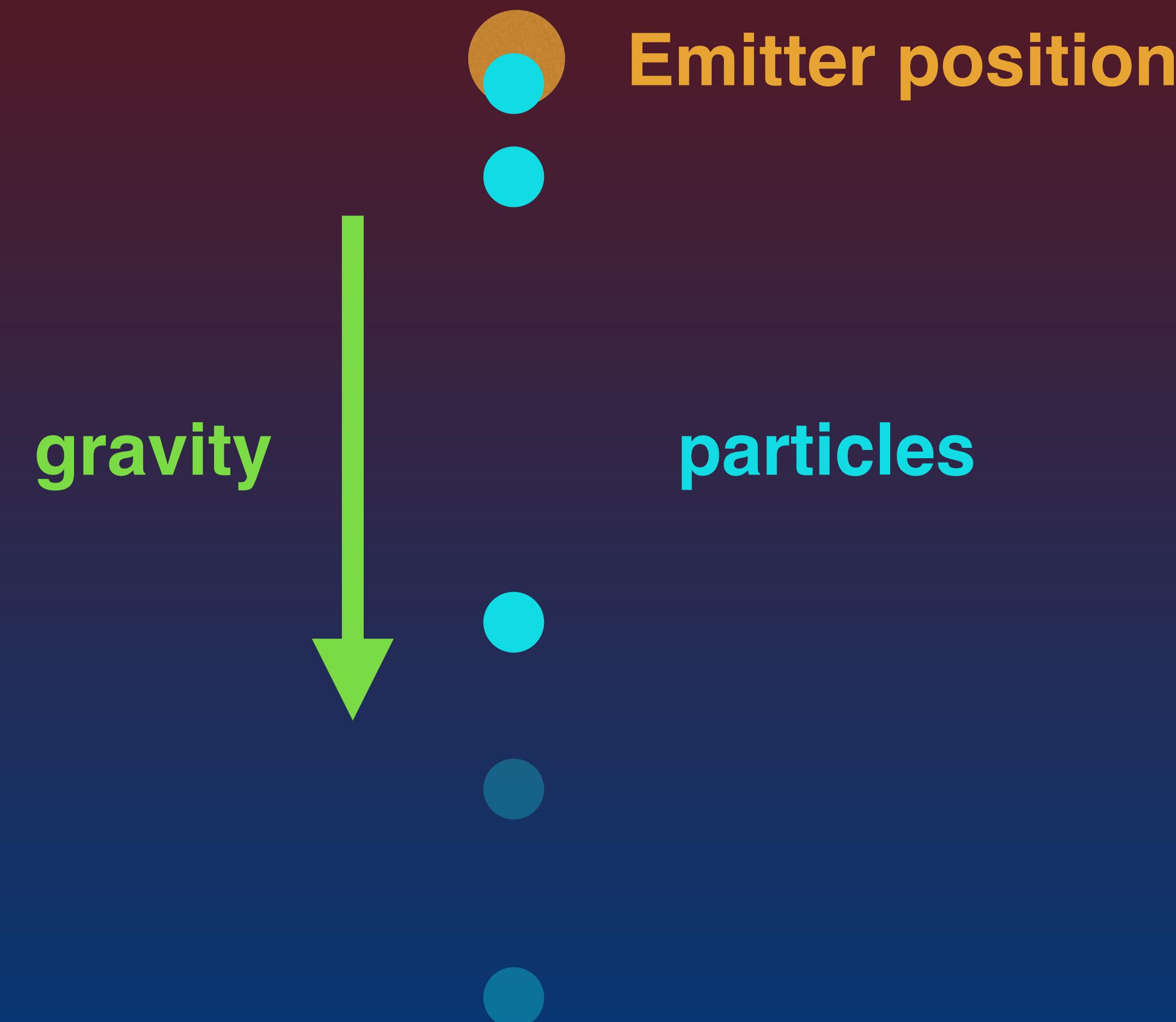
A dripping faucet.



An emitter 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. When a particle's lifetime exceeds the particle system's max lifetime, it is reset to the emitter's position.



Particle Emitter

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

```
class Particle {  
public:  
    Vector position;  
    Vector 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.

We want to draw all of our particles together as a single array so it's nice and fast. Let's use GL_POINTS for simplicity.

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

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

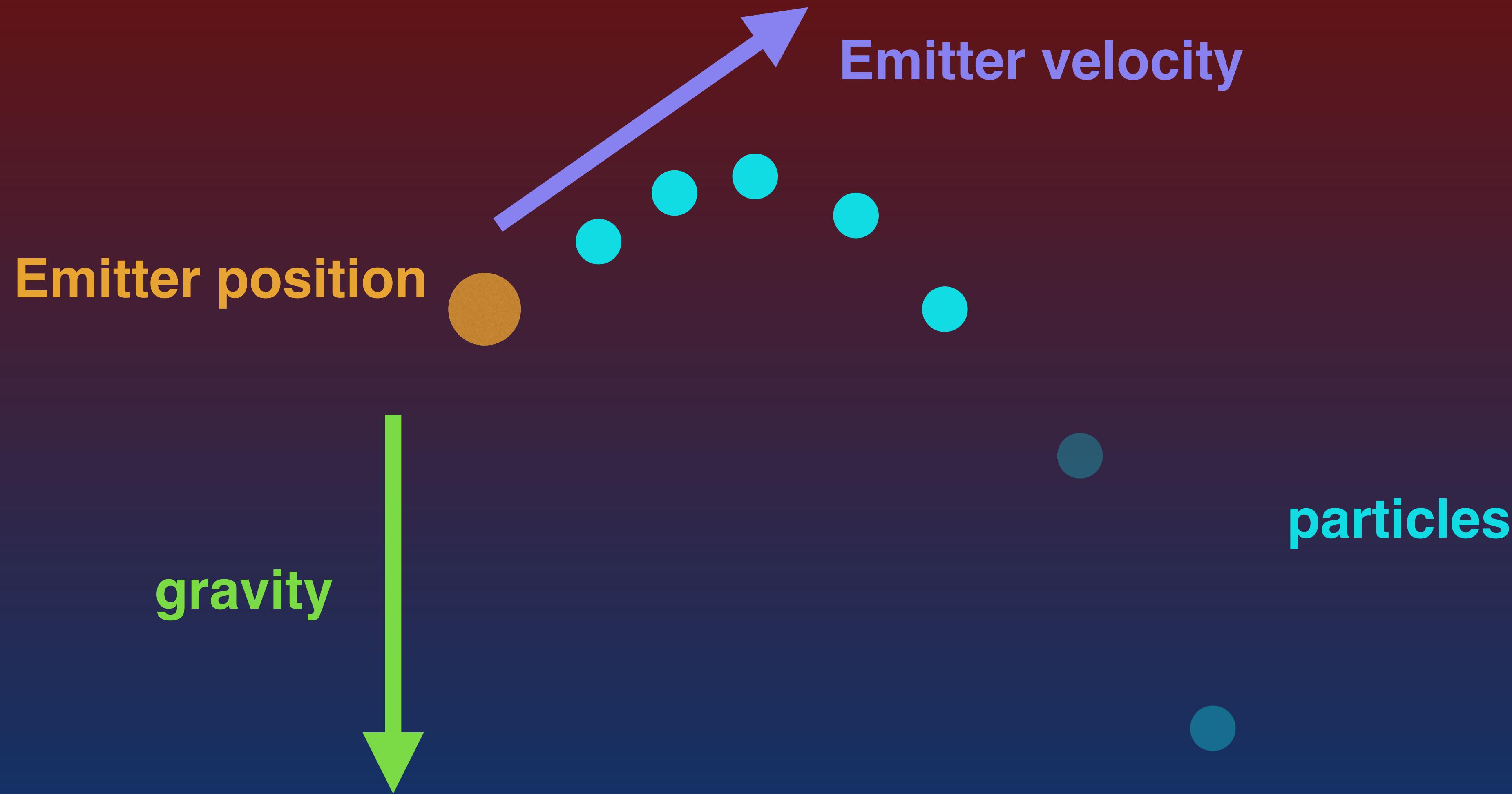
glVertexPointer(2, GL_FLOAT, 0, particleVertices.data());
 glEnableClientState(GL_VERTEX_ARRAY);

glDrawArrays(GL_POINTS, 0, particleVertices.size()/2);
```

Shooting particles.

A fountain.





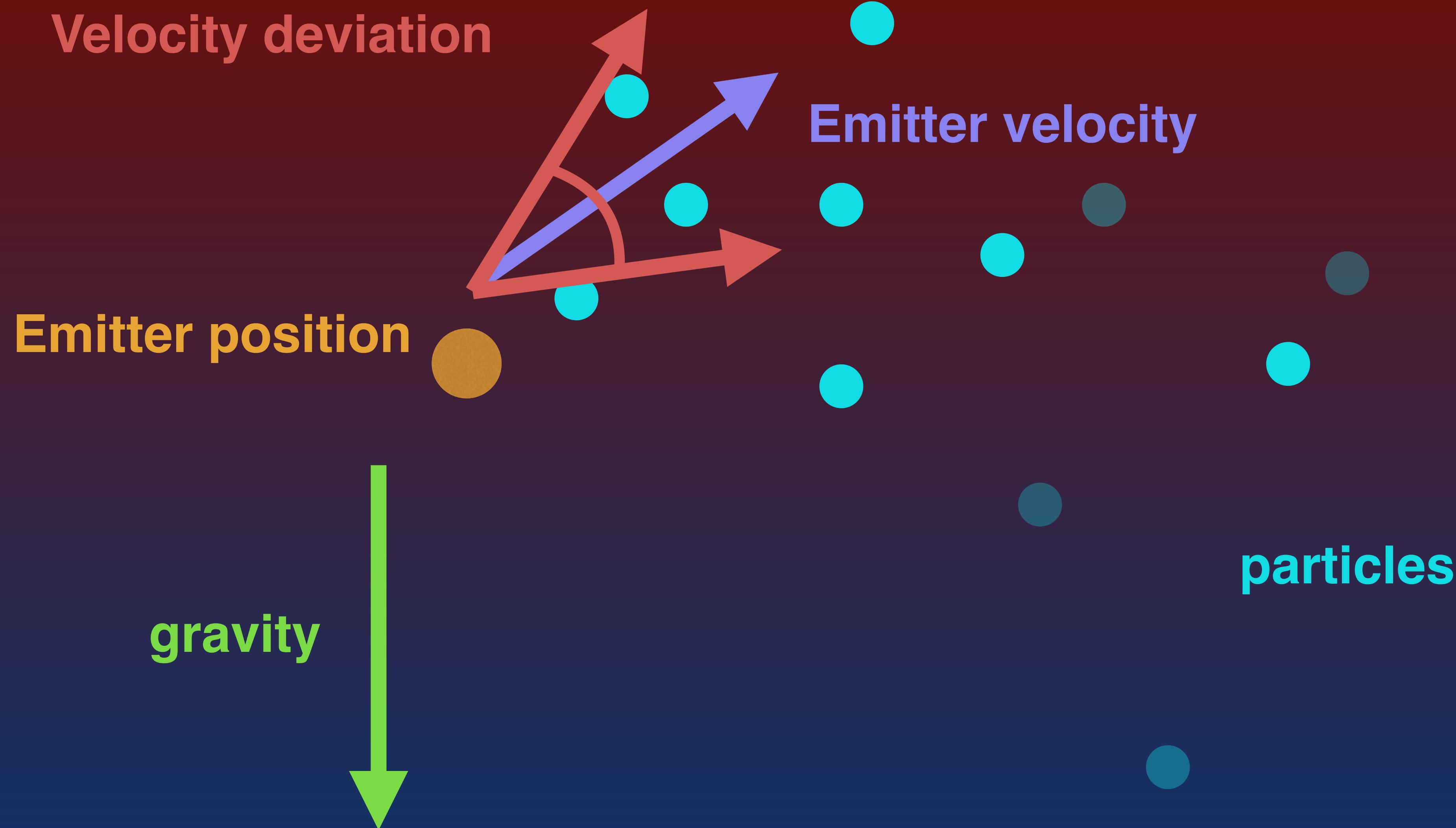
Let's add a velocity to the emitter.

```
class ParticleEmitter {  
public:  
  
    // ...  
    Vector 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.

Velocity deviation



Emitter velocity

Emitter position

gravity

particles

Let's add a velocity deviation.

```
class ParticleEmitter {  
public:  
  
    // . . .  
    Vector velocity;  
    Vector 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:  
  
    // ...  
    Color startColor;  
    Color 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 m = (particles[i].lifetime/maxLifetime);  
    particleColors.push_back(lerp(startColor.r, endColor.r, m));  
    particleColors.push_back(lerp(startColor.g, endColor.g, m));  
    particleColors.push_back(lerp(startColor.b, endColor.b, m));  
    particleColors.push_back(lerp(startColor.a, endColor.a, m));  
}  
  
glColorPointer(4, GL_FLOAT, 0, particleColors.data());  
 glEnableClientState(GL_COLOR_ARRAY);
```

Make sure you enable blending!

Color deviation.

We can add a color deviation to the emitter. 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.

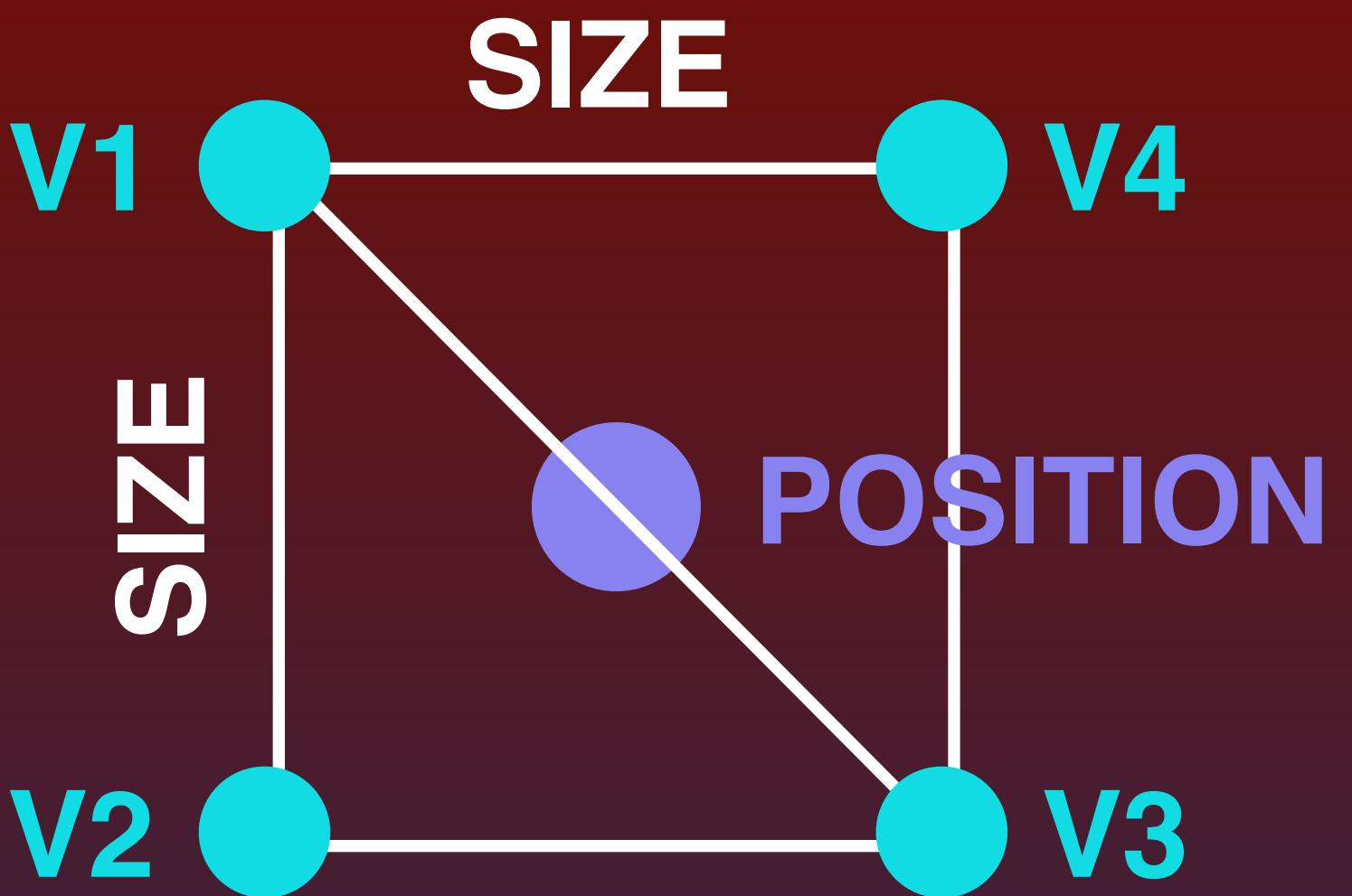
```
particleColors.push_back(lerp(startColor.r, endColor.r, relativeLifetime) +  
particles[i].colorDeviation.r);
```

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.

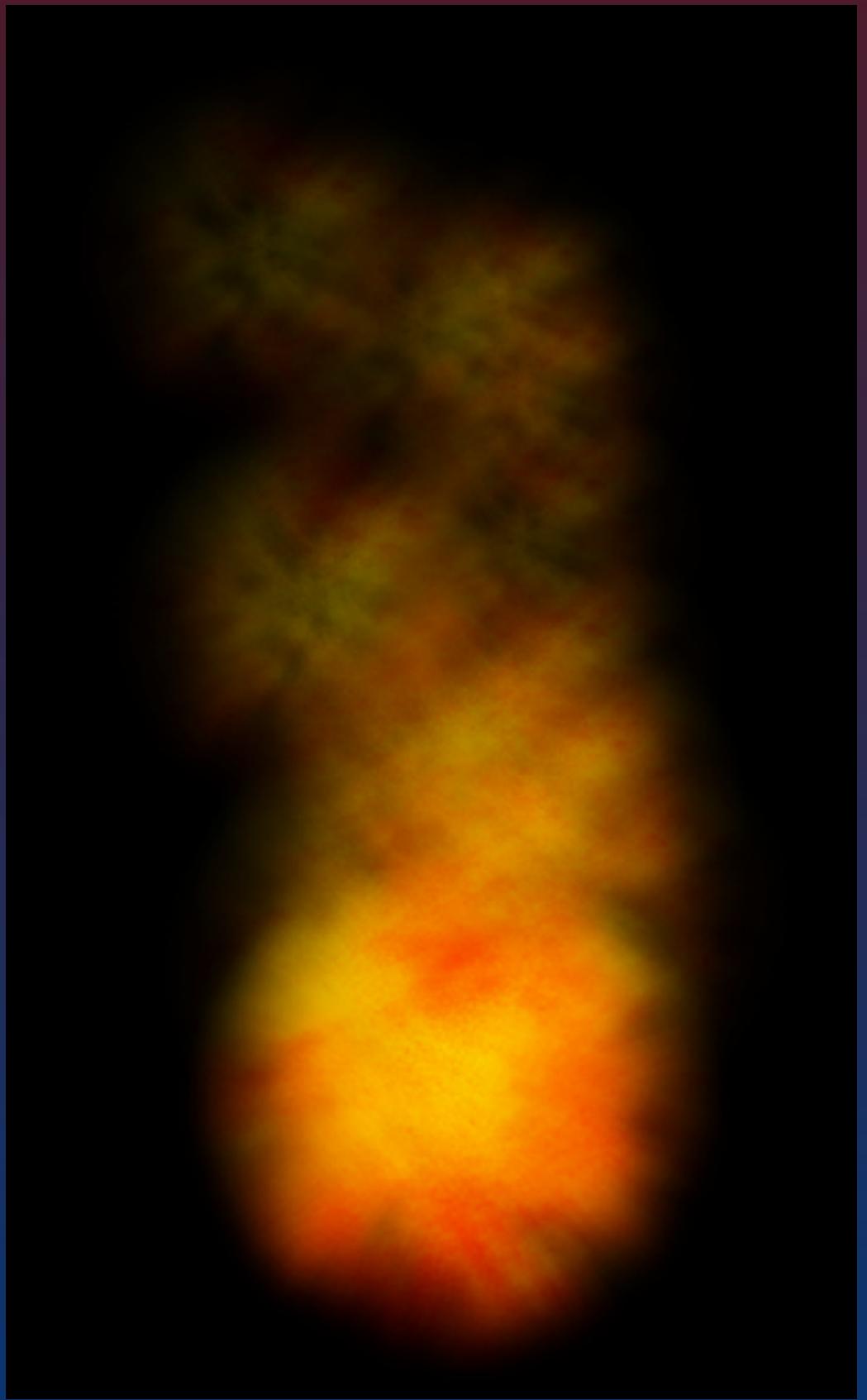
```
class ParticleEmitter {  
public:  
  
    // ...  
    float startSize;  
    float endSize;  
    float sizeDeviation;
```

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

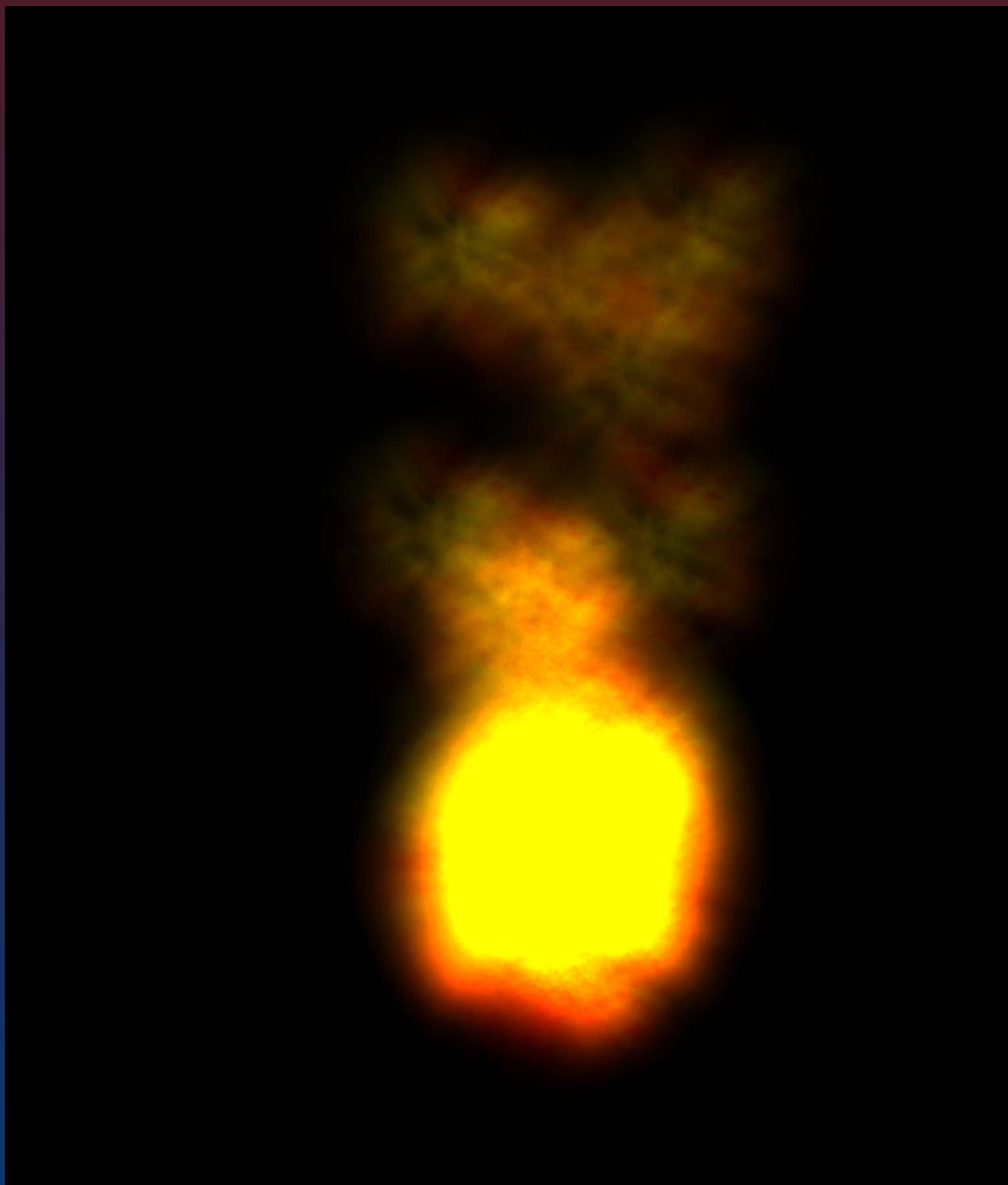
```
for(int i=0; i < particles.size(); i++) {  
  
    float relativeLifetime = (particles[i].lifetime/maxLifetime);  
    float size = lerp(startSize, endSize, relativeLifetime) + particles[i].sizeDeviation;  
  
    unsigned int vertexOffset = particleVertices.size()/2;  
  
    particleVertices.insert(particleVertices.end(), {  
        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  
    });  
  
    particleUVs.insert(particleUVs.end(), {  
        0.0f, 0.0f,  
        0.0f, 1.0f,  
        1.0f, 1.0f,  
        1.0f, 0.0f  
    });  
  
    indices.insert(indices.end(), {vertexOffset+0,vertexOffset+1, vertexOffset+2, vertexOffset+0, vertexOffset+2, vertexOffset+3});  
  
}  
  
glTexCoordPointer(2, GL_FLOAT, 0, particleUVs.data());  
 glEnableClientState(GL_TEXTURE_COORD_ARRAY);  
  
 glDrawElements(GL_TRIANGLES, indices.size(), GL_UNSIGNED_INT, indices.data());
```

Additive blending!

Regular



Additive



Alpha blending

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

Additive blending

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

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 modelview 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;

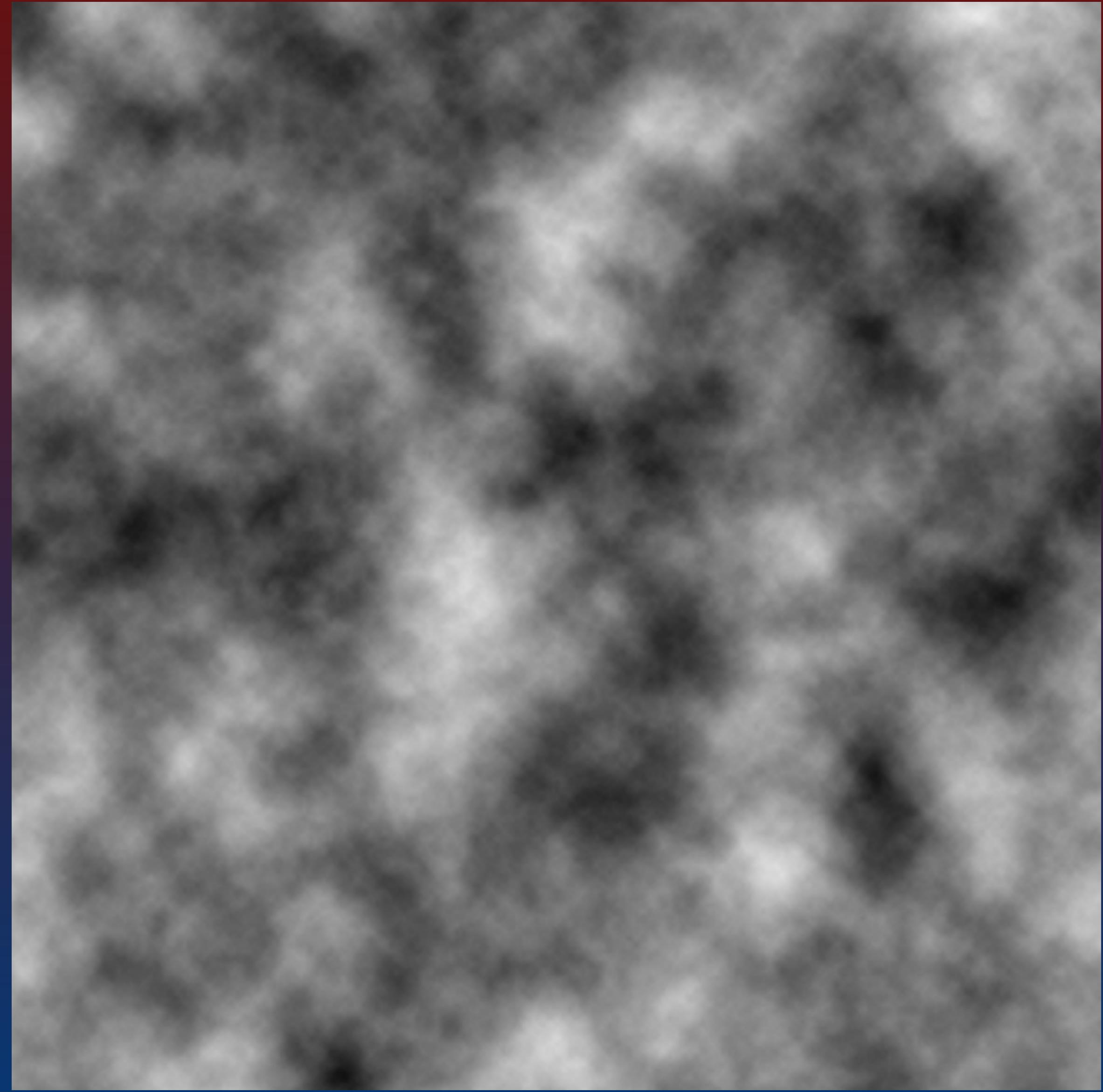
particleVertices.insert(particleVertices.end(), {
    particles[i].position.x + TL_x,
    particles[i].position.y + TL_y,

    particles[i].position.x + BL_x,
    particles[i].position.y + BL_y,

    particles[i].position.x + BR_x,
    particles[i].position.y + BR_y,

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

Natural particle motion using perlin noise.



Use PerlinNoise.h and PerlinNoise.cpp
in class repository.

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;
```

Changing emitter size.

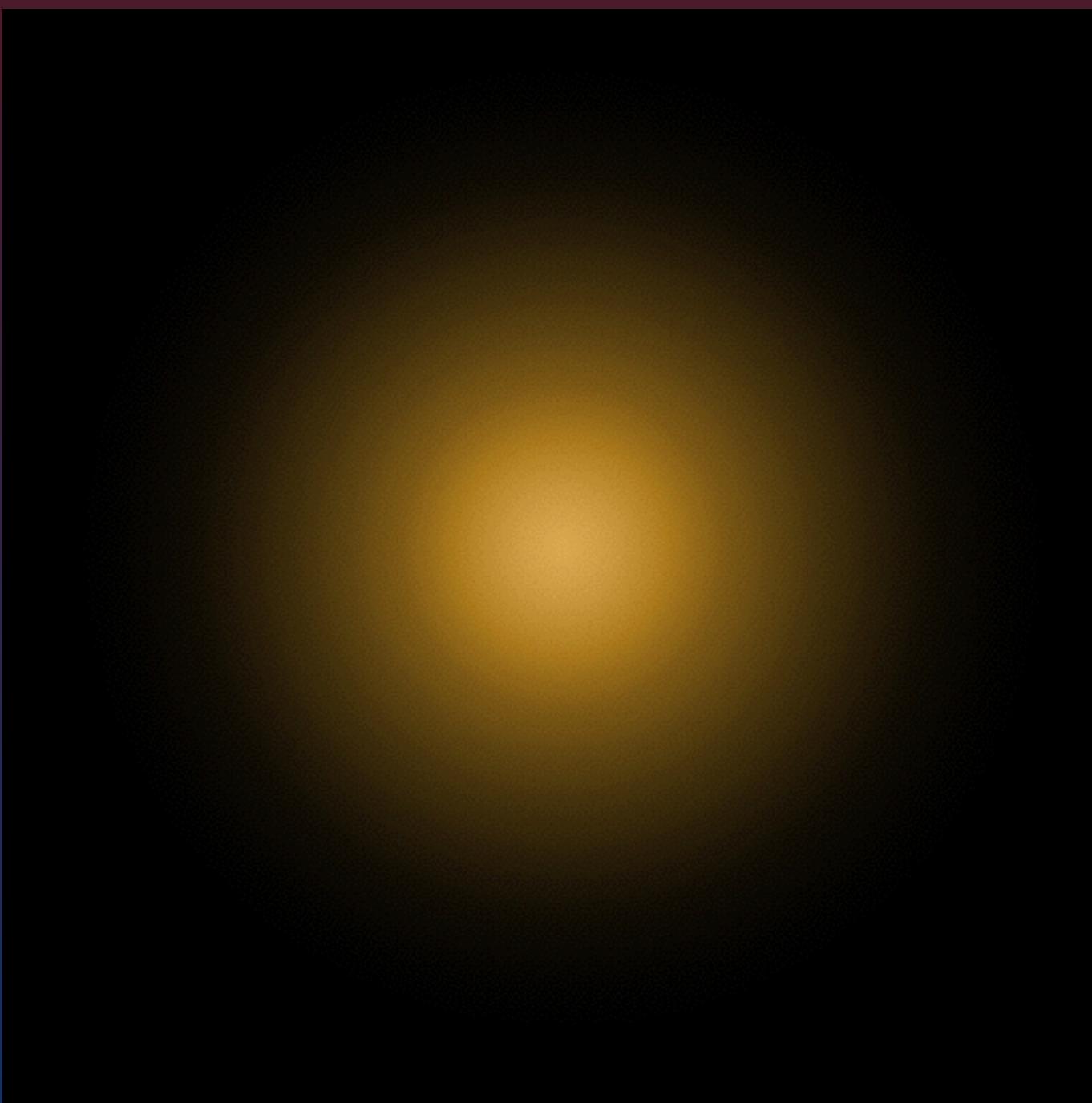
When resetting a particle, set its position to emitter position + random value within emitter size range.

Triggered particle emitters.

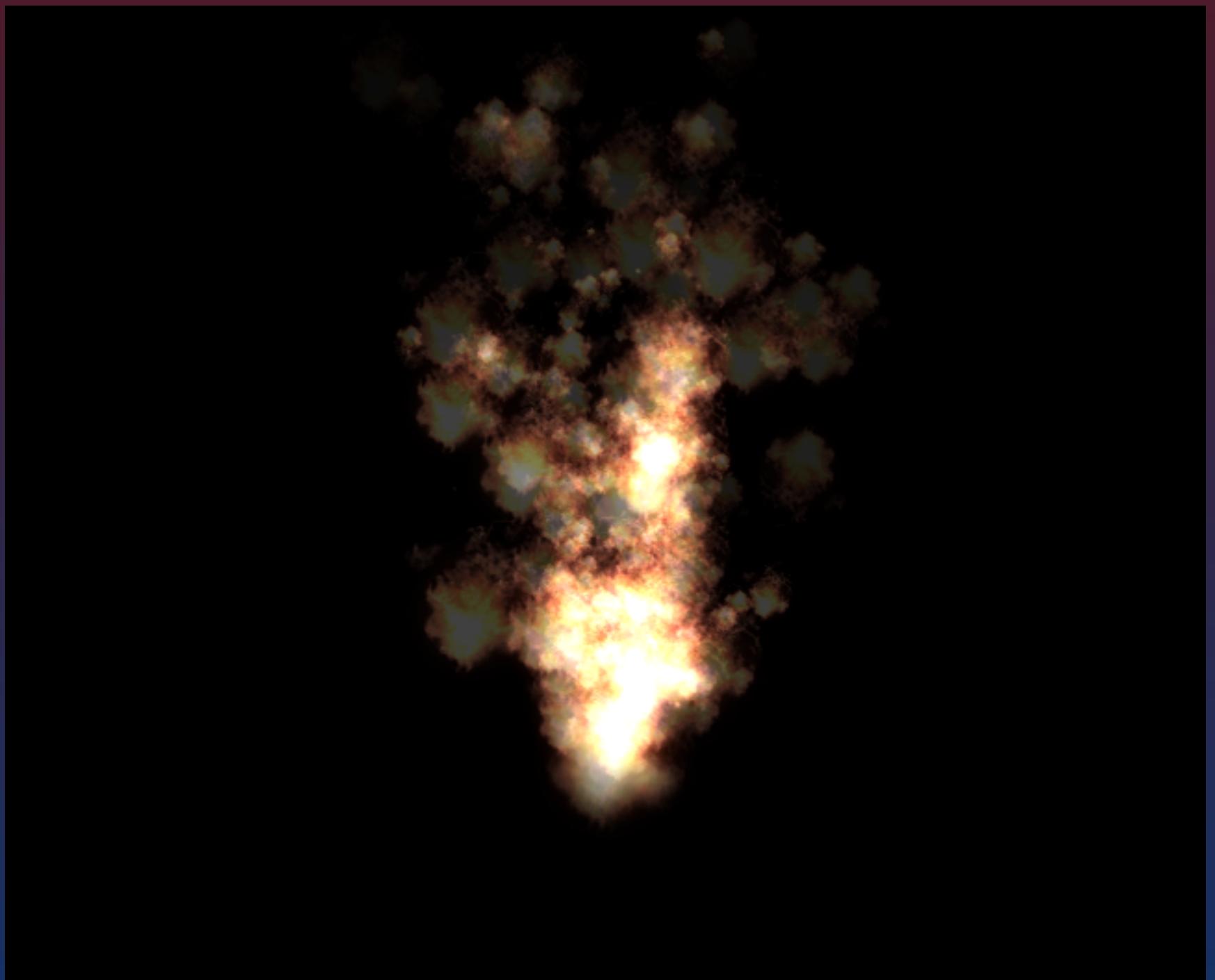
For a triggered emitter, simply do not reset particles after their lifetime is maxed out. Add a “Trigger” method to the emitter, to reset all of the particles instead.

Making things glow.

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



+





Assignment.

Add some effects to an existing game.

- At least 3 different effects.
- Animation or particles/glow.