Particles2D

quasilyte @ GoFunc 2024

What? Why?

Reasons to care about this talk:

- You’re curious about game development in Go

What? Why?

Reasons to care about this talk:

- You’re curious about game development in Go
- You’re into weird optimizations

What? Why?

Reasons to care about this talk:

- You’re curious about game development in Go
- You’re into weird optimizations
- You’re interested in VFX generated via code

What? Why?

Reasons to care about this talk:

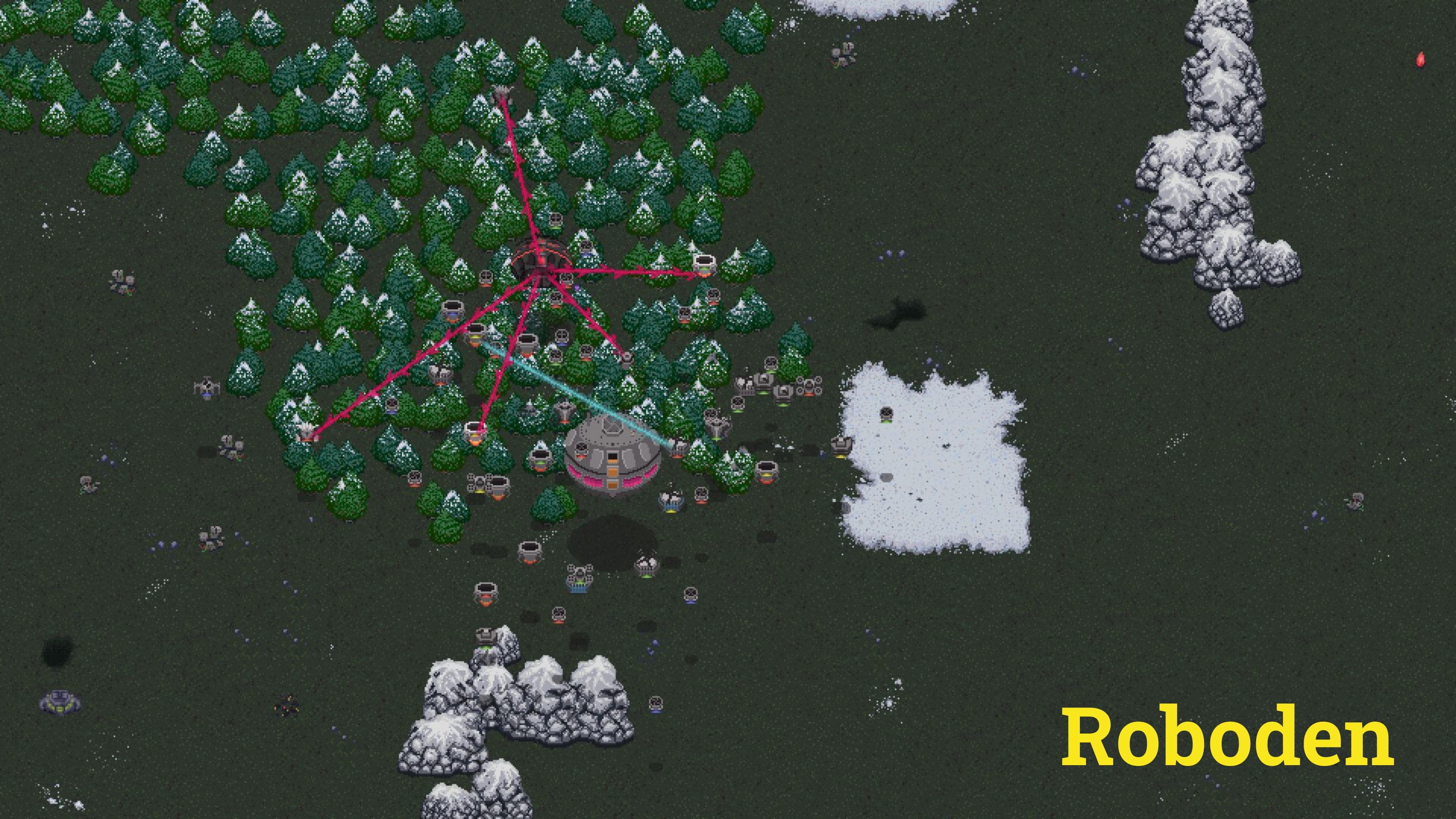
- You’re curious about game development in Go
- You’re into weird optimizations
- You’re interested in VFX generated via code
- You’re working on a game called NebuLeet

Agenda

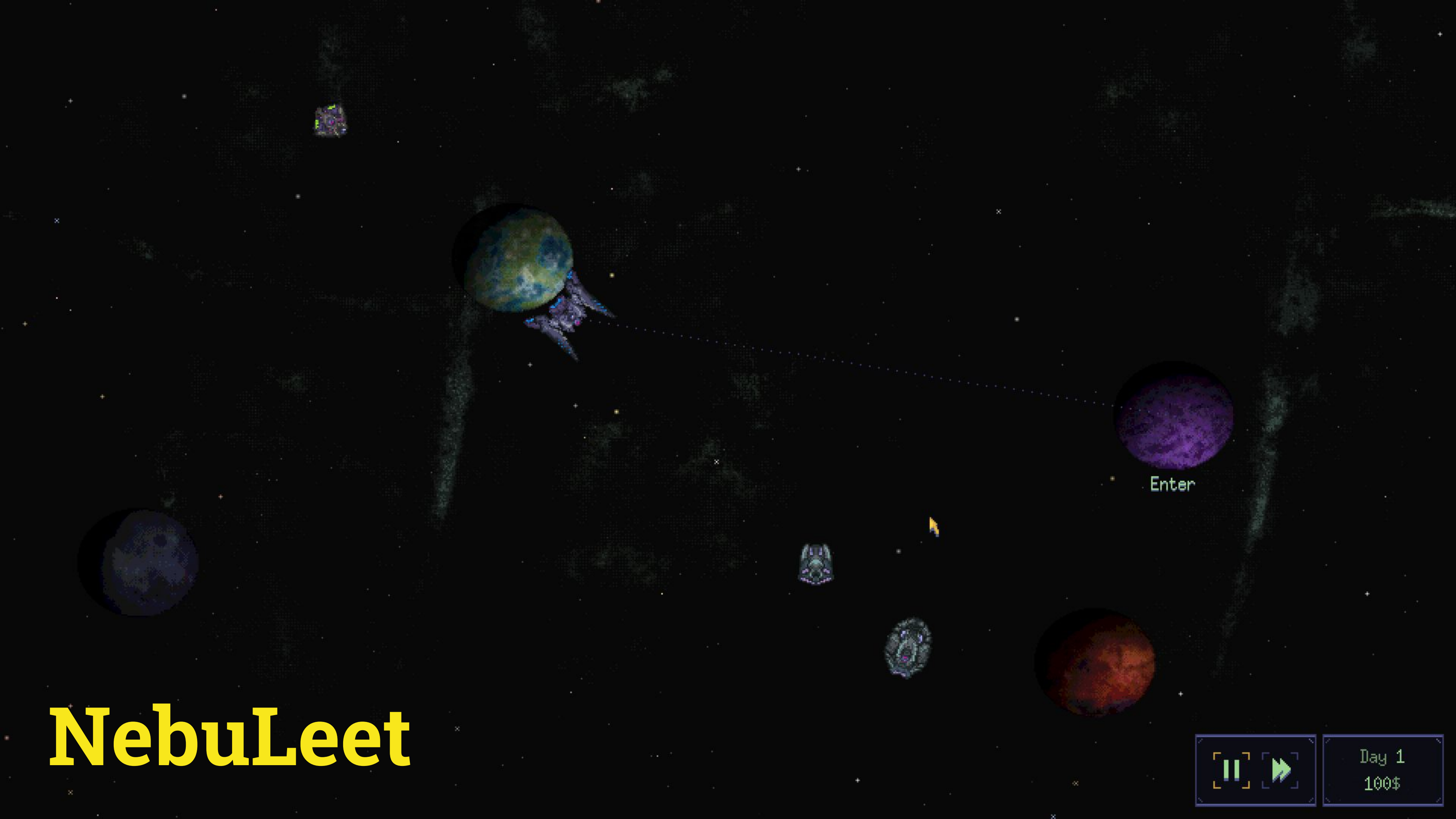
- **Intro**
- VFX methods
- Particle system overview
- Particles layout
- Batch rendering
- GPU particles

quasilyte tech

- I’m making games using Go (Ebitengine),
- maintaining gamedev libraries for Go,
- creating related learning materials,
- organizing Russian-speaking Go gamedev community



Roboden



NebuLeet

CaveBots

DECIPHERISM
IN
A-Z
?
OUT
A-Z

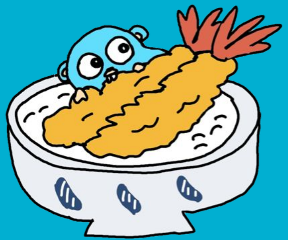
Tune Fire

ASSEMBLOX

SINECORD

RETROWAVE CITY

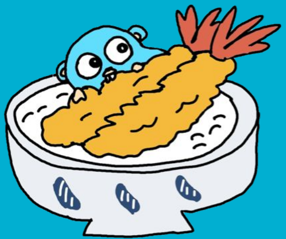
Making games with Go



Ebitengine™

- 2D game engine
- Engine is written in Go
- Games are written in Go
- Covers many platforms

Making games with Go



Ebitengine™

Game engine

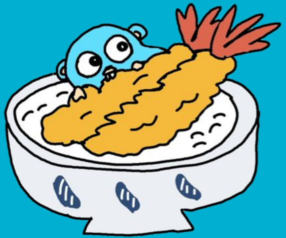
Code

Visuals

Sound

… and more

Making games with Go



Ebitengine™

Game engine

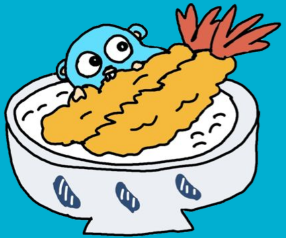
Code

Visuals

Sound

... and more

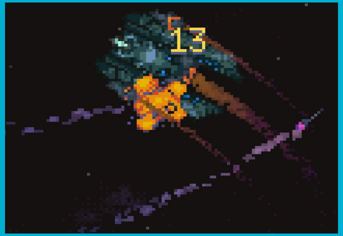
More specifically...



Ebitengine™

Game engine

Visuals



VFX

Agenda

- Intro
- **VFX methods**
- Particle system overview
- Particles layout
- Batch rendering
- GPU particles

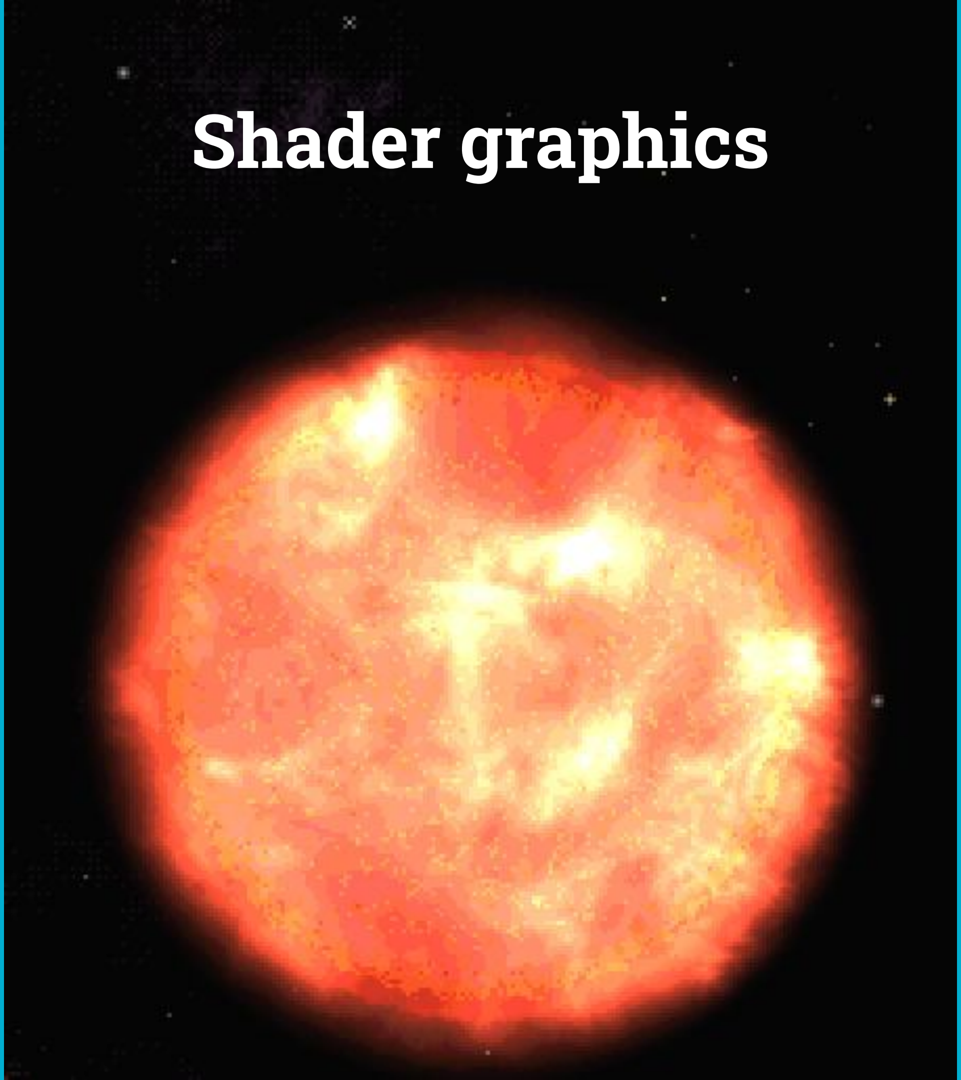
Creating visuals effects in games

Main tools:

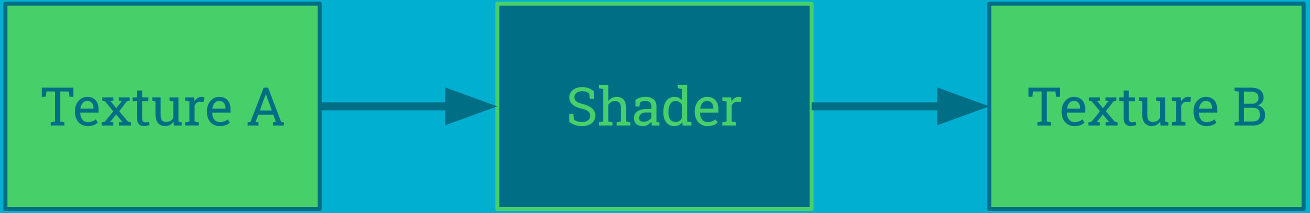
- Shaders
- Using a bunch of Sprite and/or Animation objects
- Particle systems

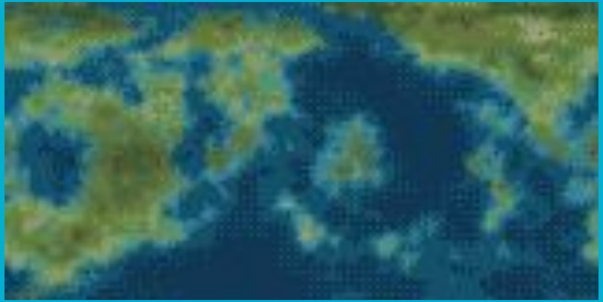
+ combinations of these

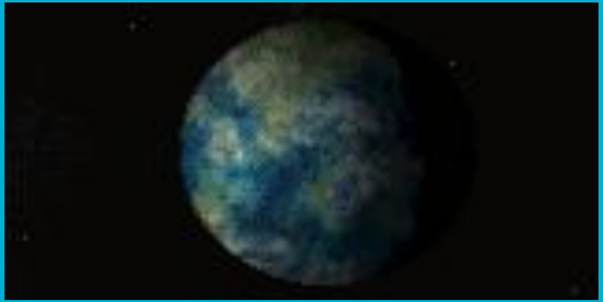
Shader graphics



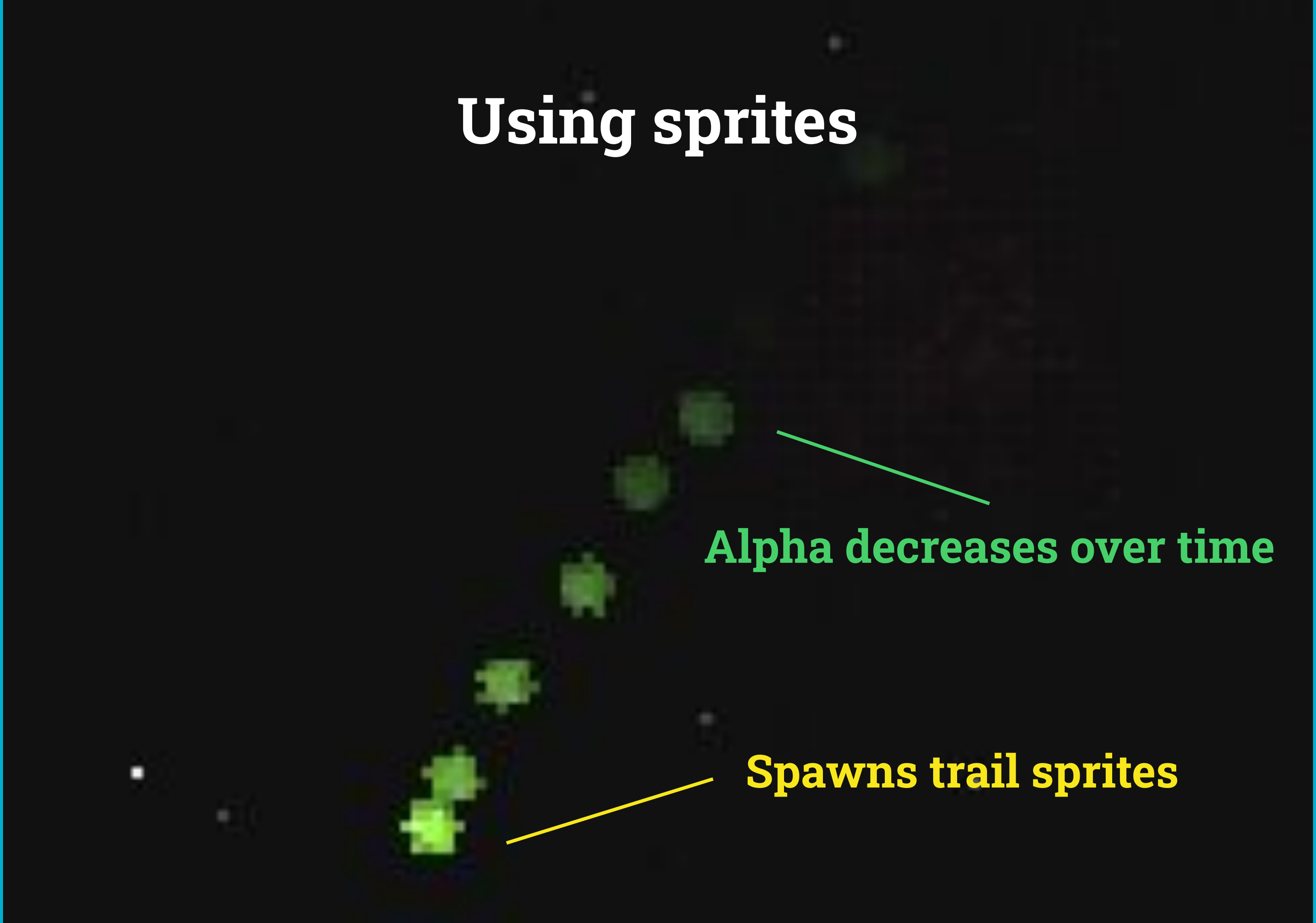
What is a shader







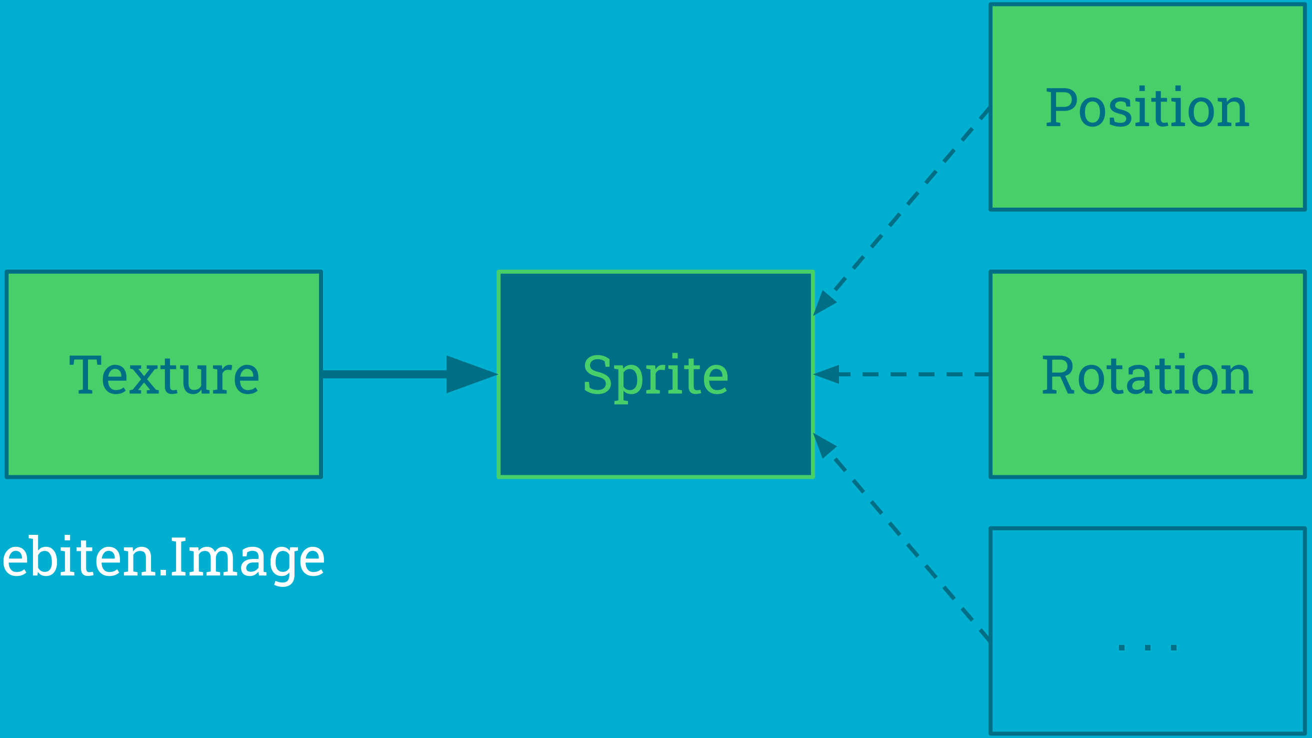
Using sprites



Alpha decreases over time

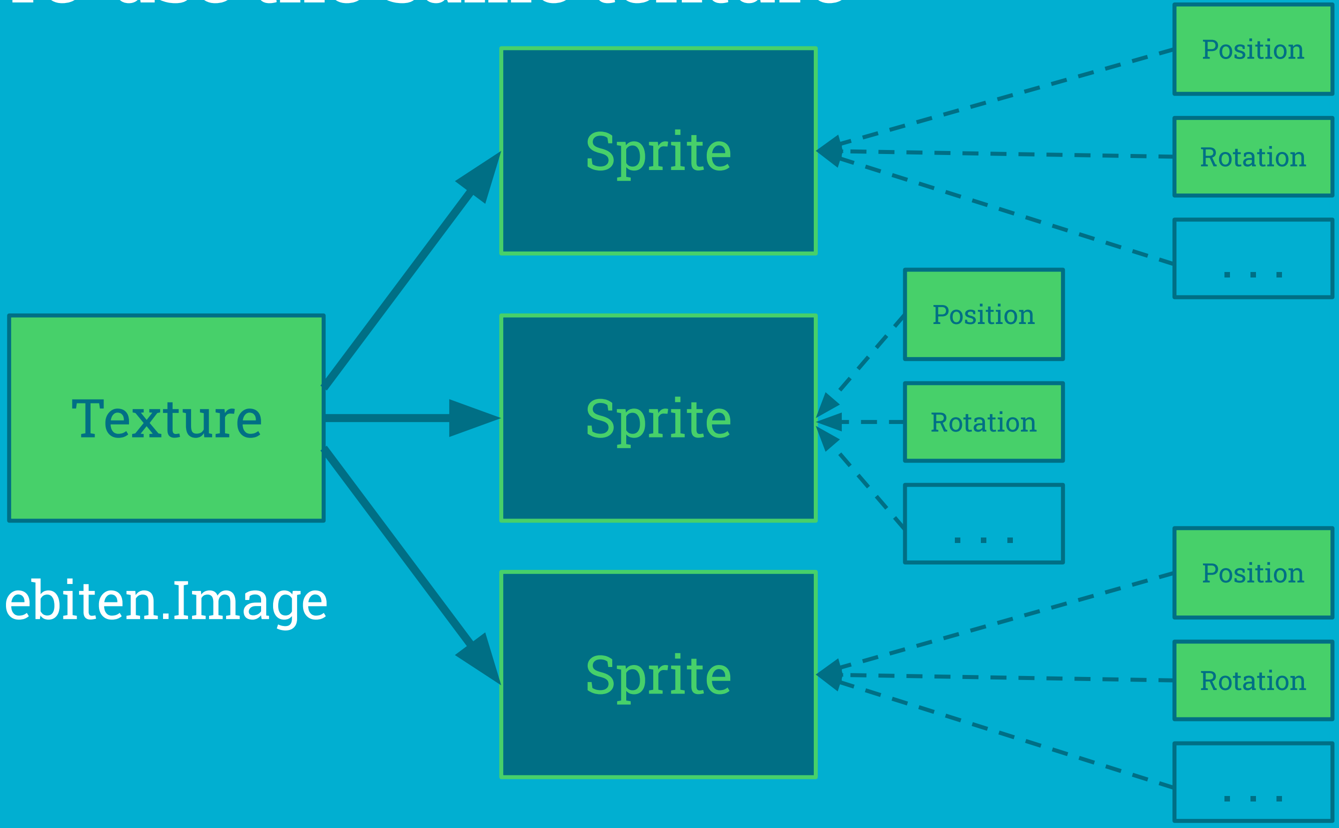
Spawns trail sprites

What is a sprite

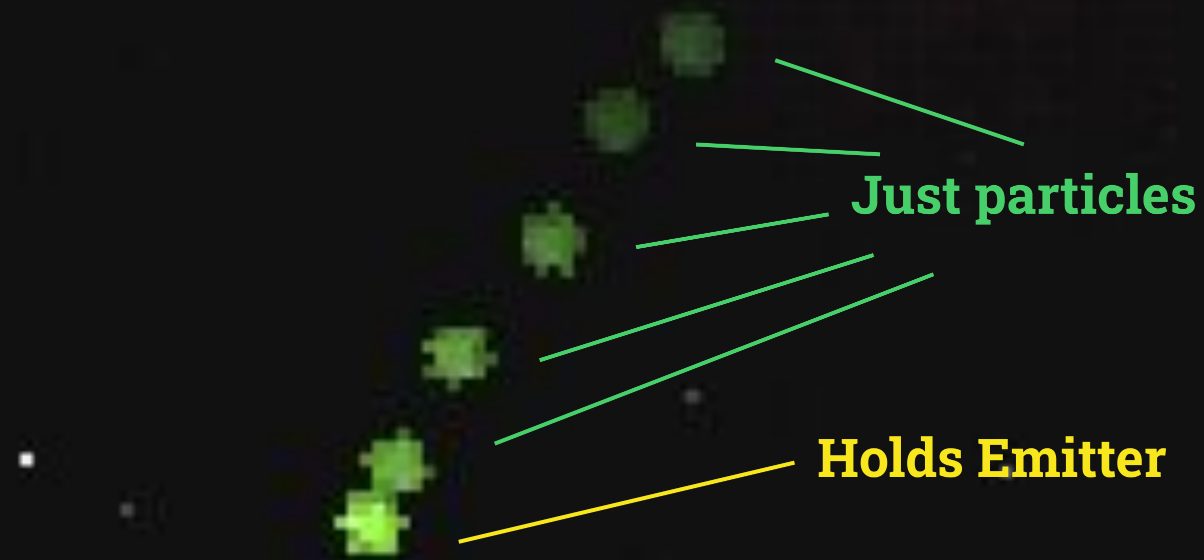


ebiten.Image

Sprites re-use the same texture



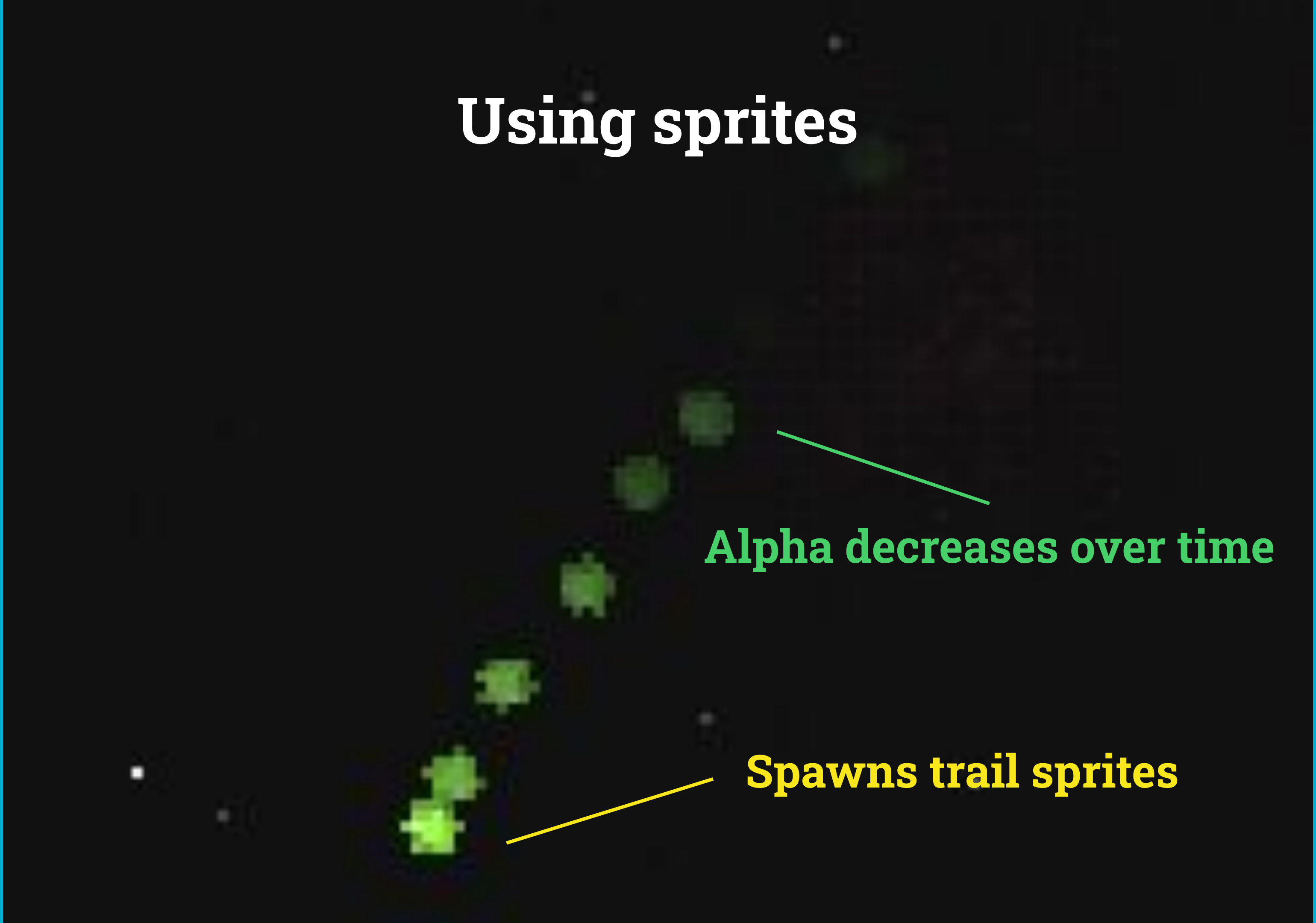
Particle system



Just particles

Holds Emitter

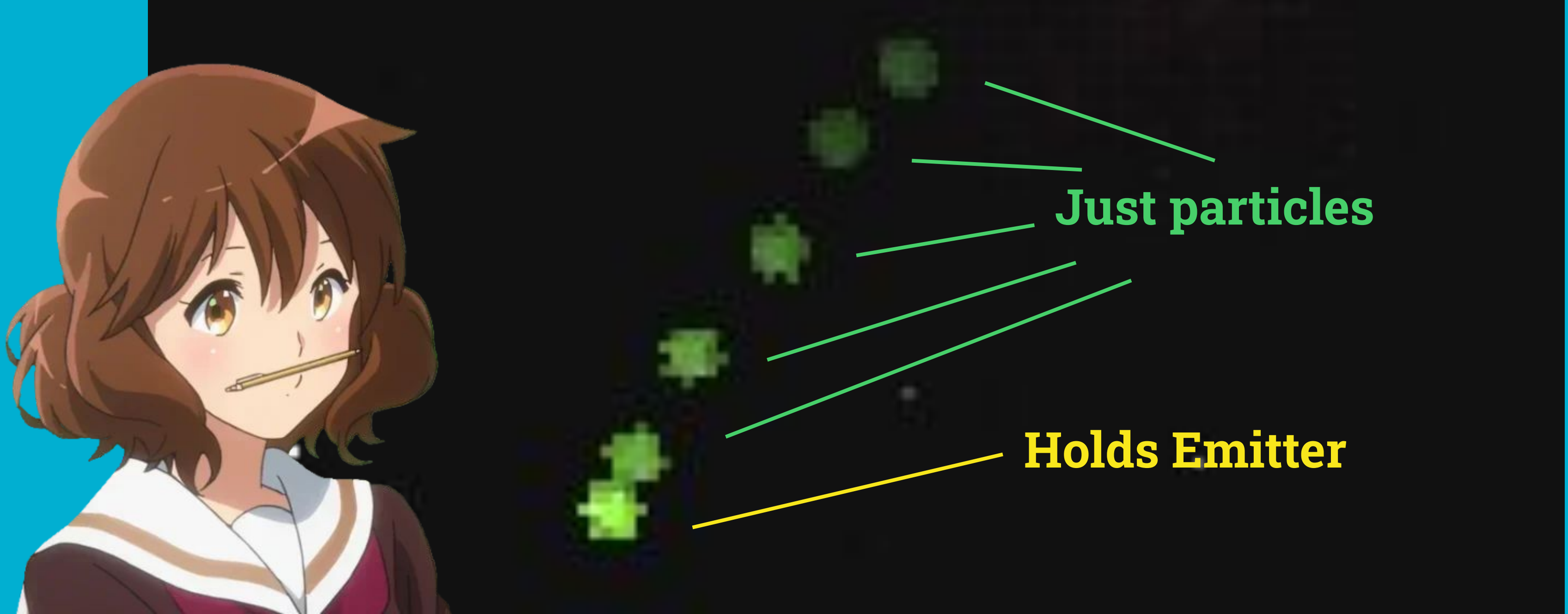
Using sprites



Alpha decreases over time

Spawns trail sprites

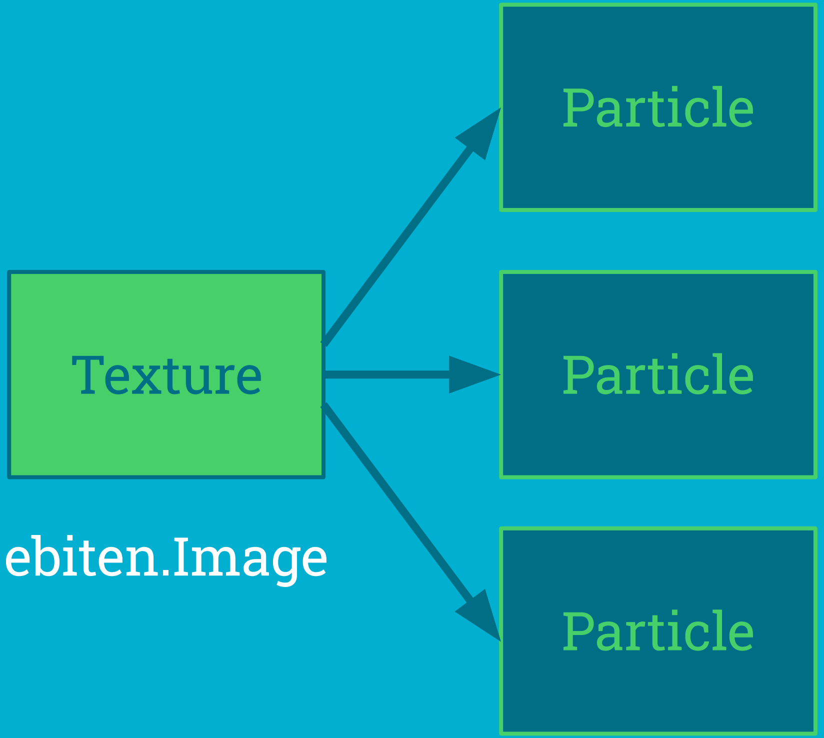
Particle system



Just particles

Holds Emitter

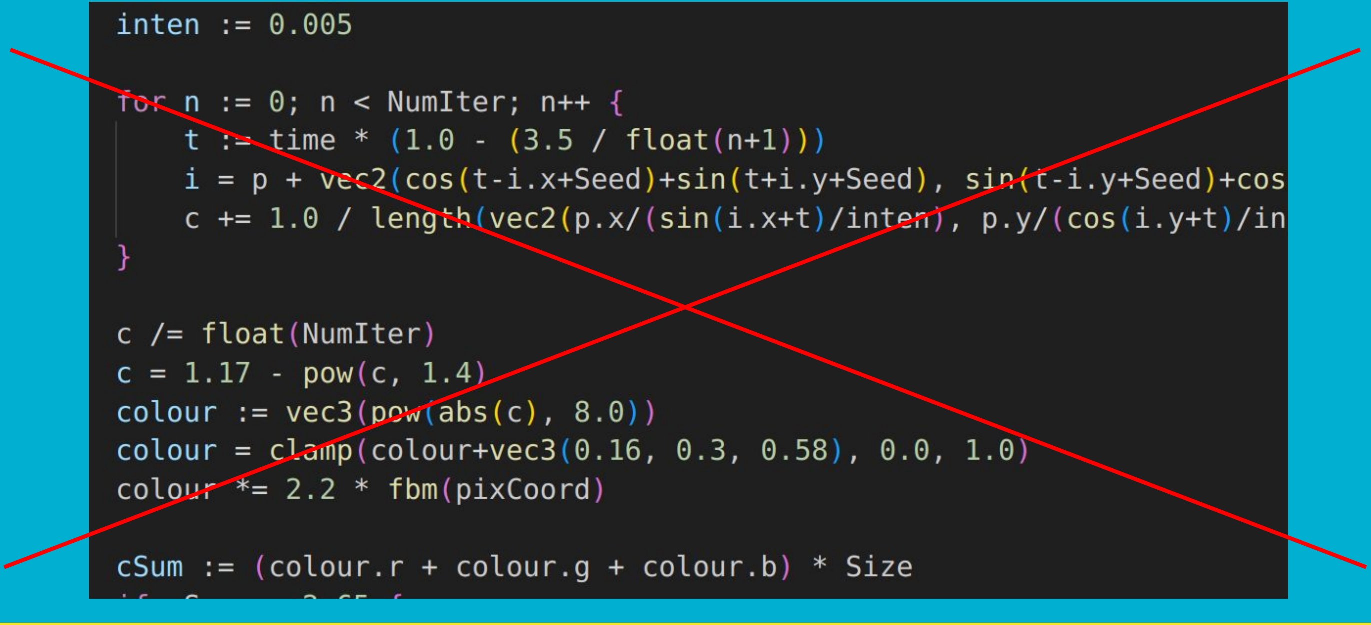
Particles are (very) lightweight sprites



- Less memory
- Better batching
- Ephemeral

Why do we need particle systems?

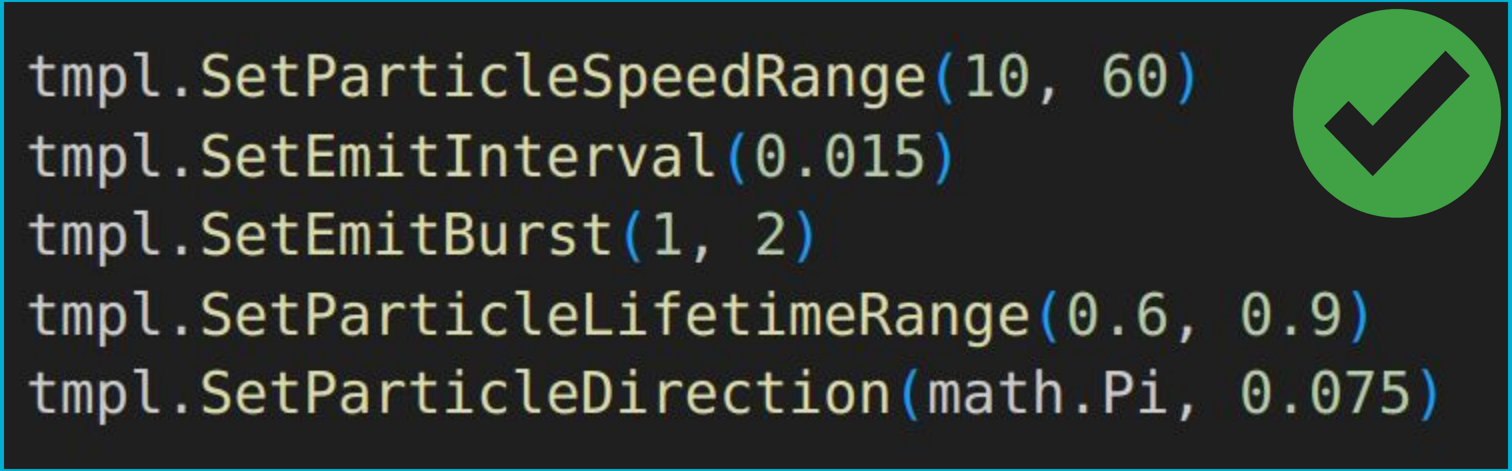
- Easy to learn and use (in comp. with shaders)



```
inten := 0.005

for n := 0; n < NumIter; n++ {
    t := time * (1.0 - (3.5 / float(n+1)))
    i = p + vec2(cos(t-i.x+Seed)+sin(t+i.y+Seed), sin(t-i.y+Seed)+cos
    c += 1.0 / length(vec2(p.x/(sin(i.x+t)/inten), p.y/(cos(i.y+t)/in
}

c /= float(NumIter)
c = 1.17 - pow(c, 1.4)
colour := vec3(pow(abs(c), 8.0))
colour = clamp(colour+vec3(0.16, 0.3, 0.58), 0.0, 1.0)
colour *= 2.2 * fbm(pixCoord)

cSum := (colour.r + colour.g + colour.b) * Size
```

Why do we need particle systems?

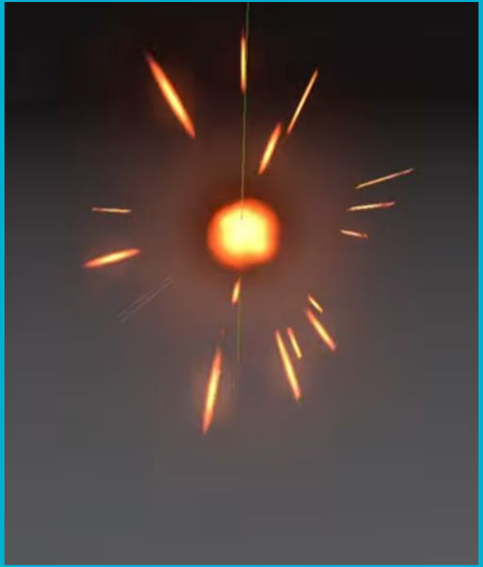
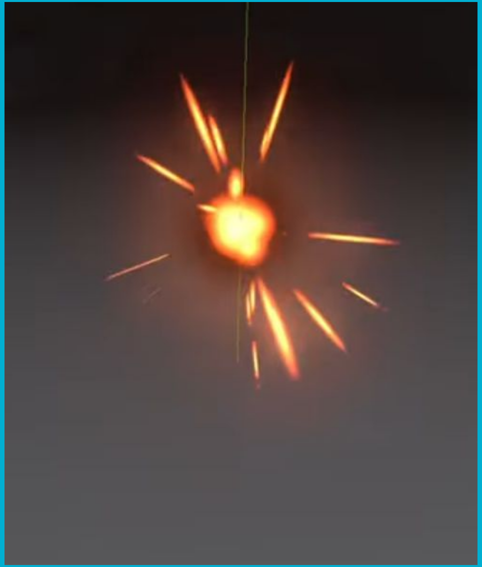
- Highly customizable look via numeric parameters



```
tmpl.SetParticleSpeedRange(10, 60)
tmpl.SetEmitInterval(0.015)
tmpl.SetEmitBurst(1, 2)
tmpl.SetParticleLifetimeRange(0.6, 0.9)
tmpl.SetParticleDirection(math.Pi, 0.075)
```

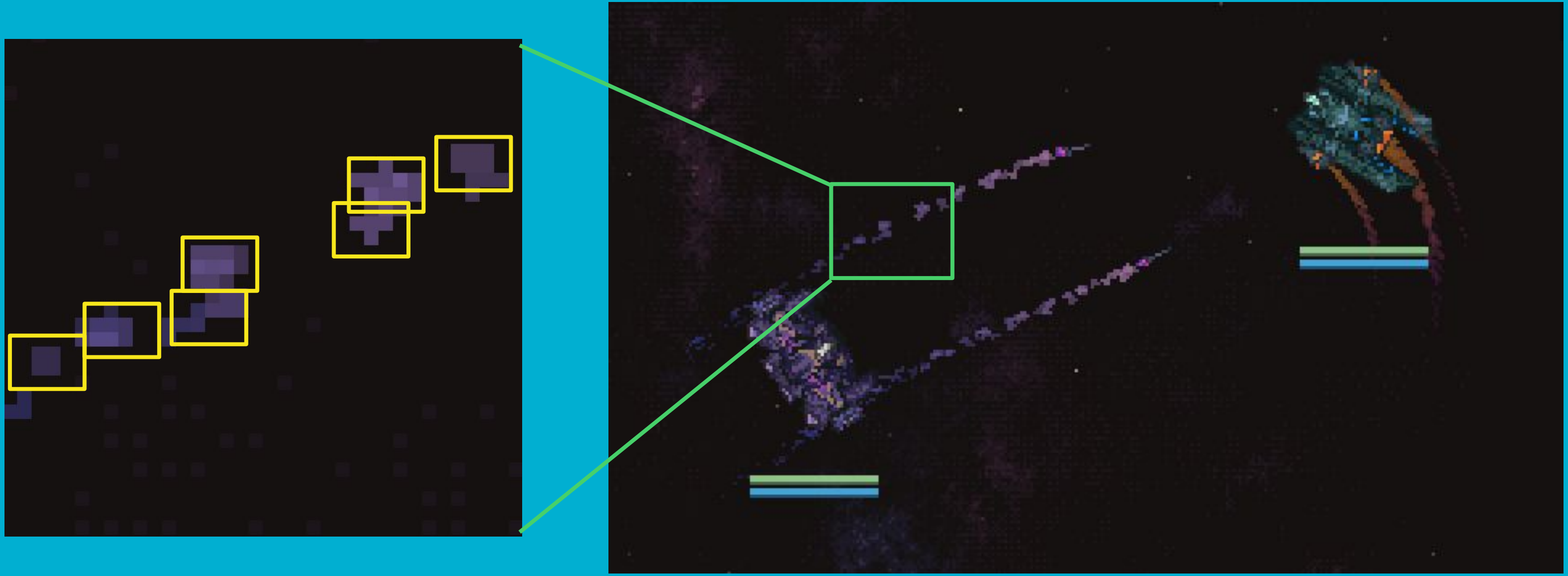
Why do we need particle systems?

- High visual variation (procedural graphics)

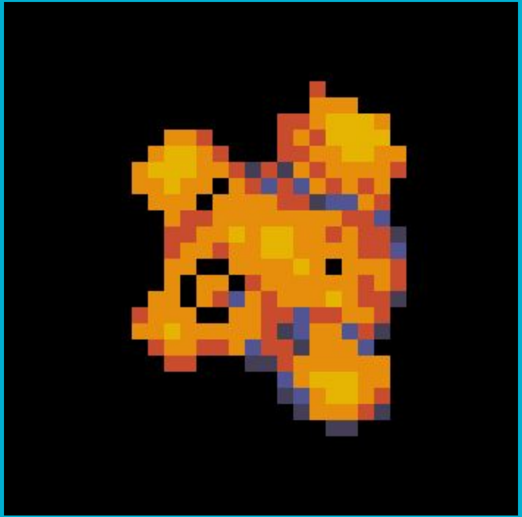
Why do we need particle systems?

- Batch rendering and processing

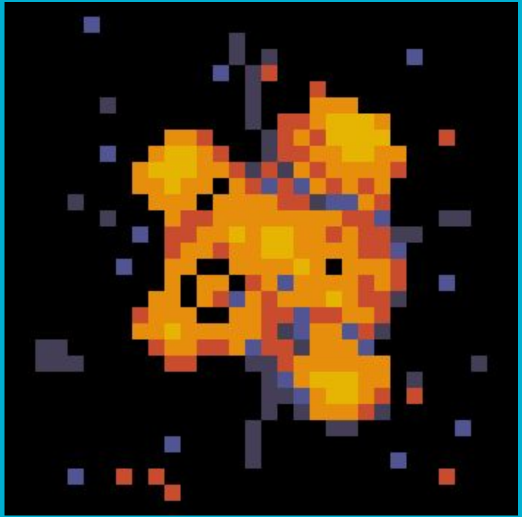


Animation/Sprites + particles

Adds extra randomness and juiciness to your effects



w/o particles



with particles

Agenda

- Intro
- VFX methods
- **Particle system overview**
- Particles layout
- Batch rendering
- GPU particles

Which particle system do I use?

- I was looking for a particle system for NebuLeet game

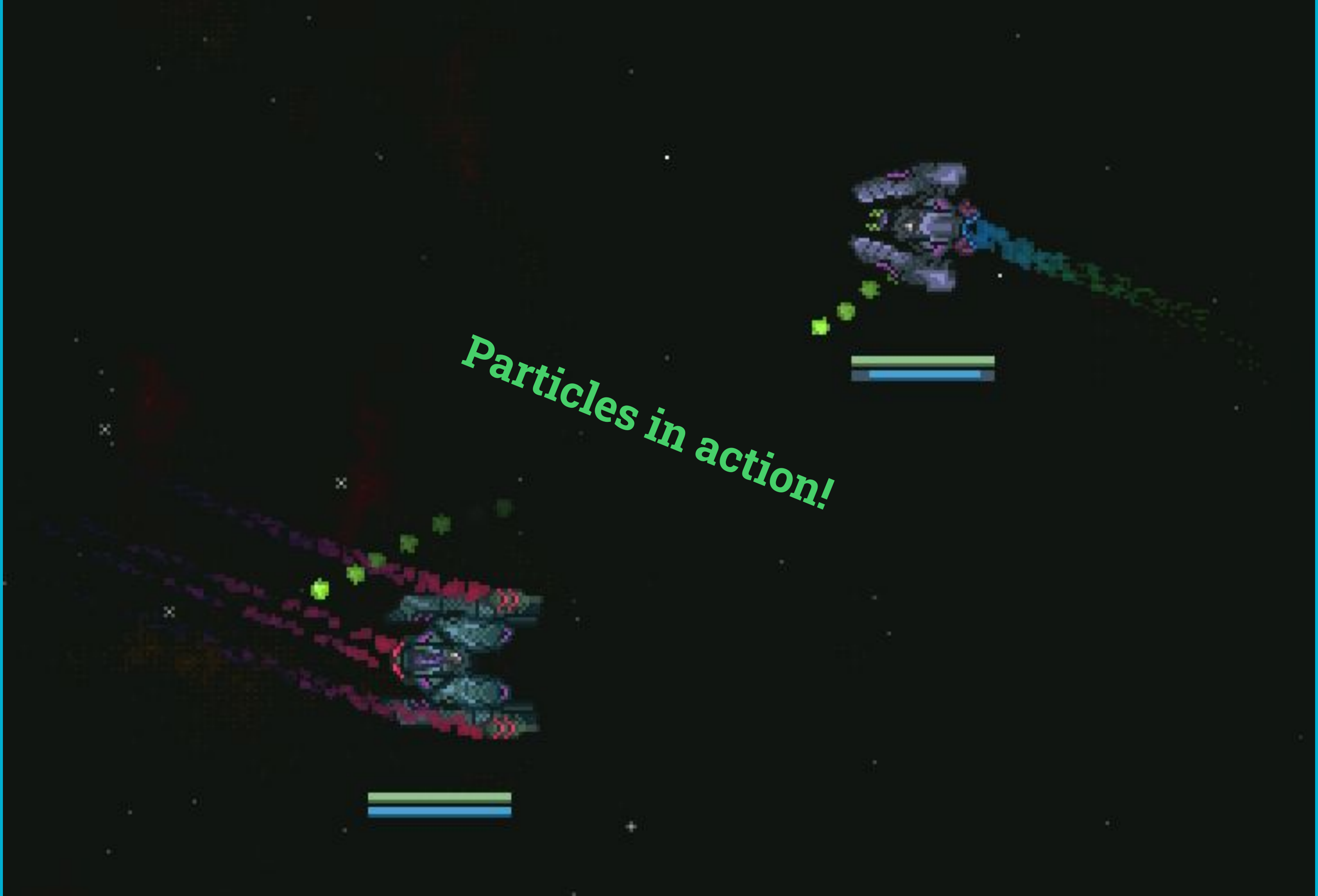
Which particle system do I use?

- I was looking for a particle system for NebuLeet game
- I didn’t find one (I wasn’t searching that well)

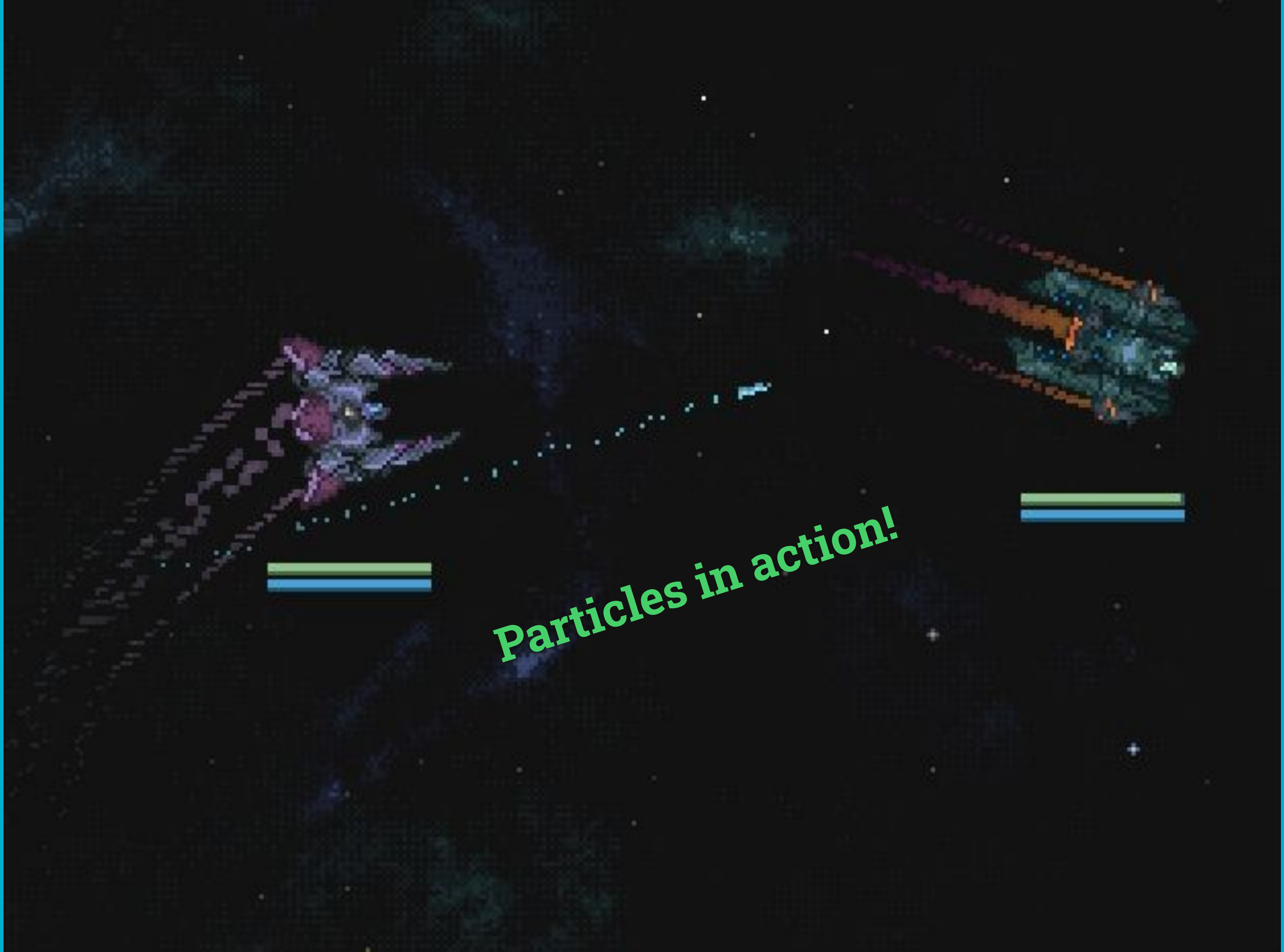
Which particle system do I use?

- I was looking for a particle system for NebuLeet game
- I didn’t find one (I wasn’t searching that well)
- I created my own (as a part of existing gfx package)

See github.com/quasilyte/ebitengine-graphics



Particles in action!

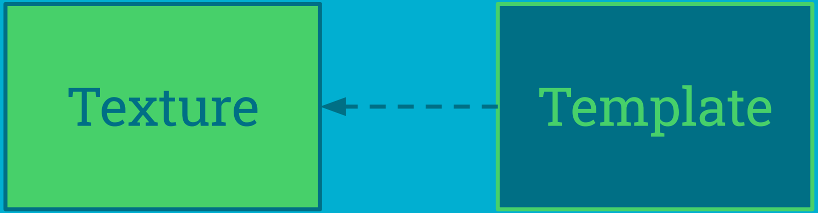


Particles in action!

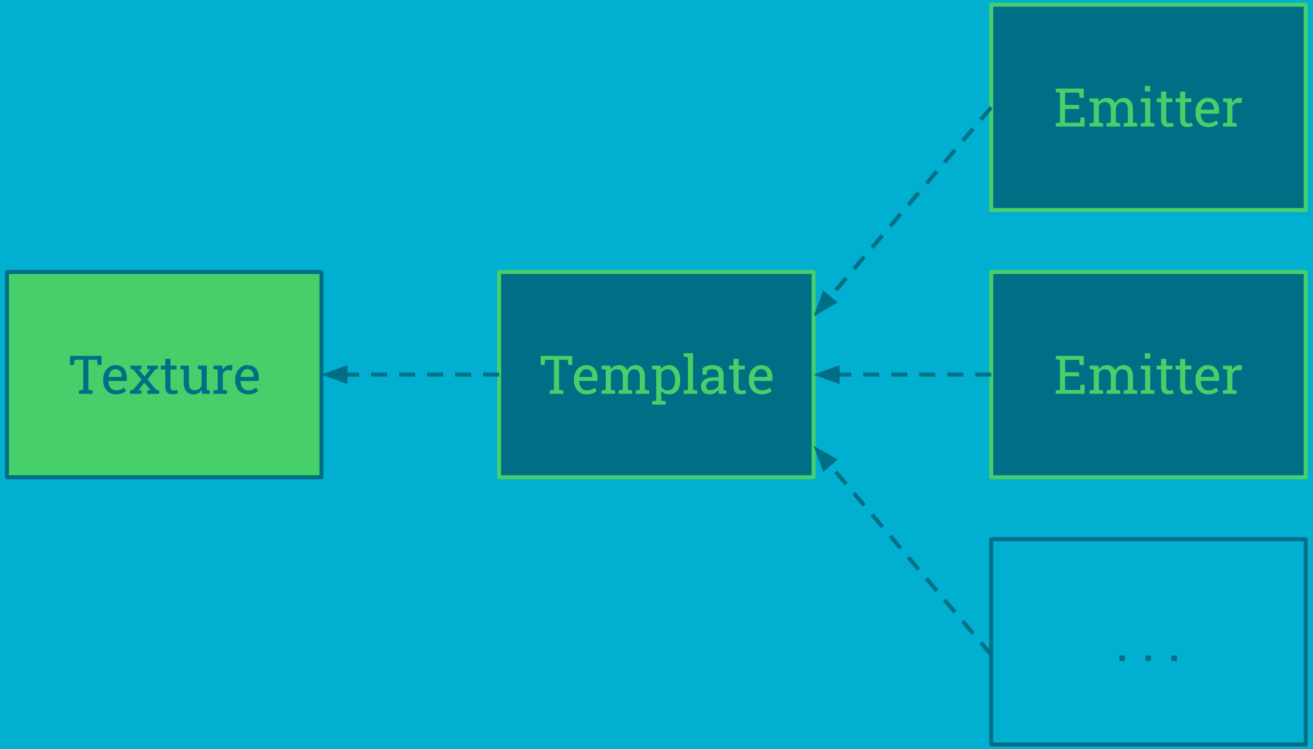
Particle system overview

Texture

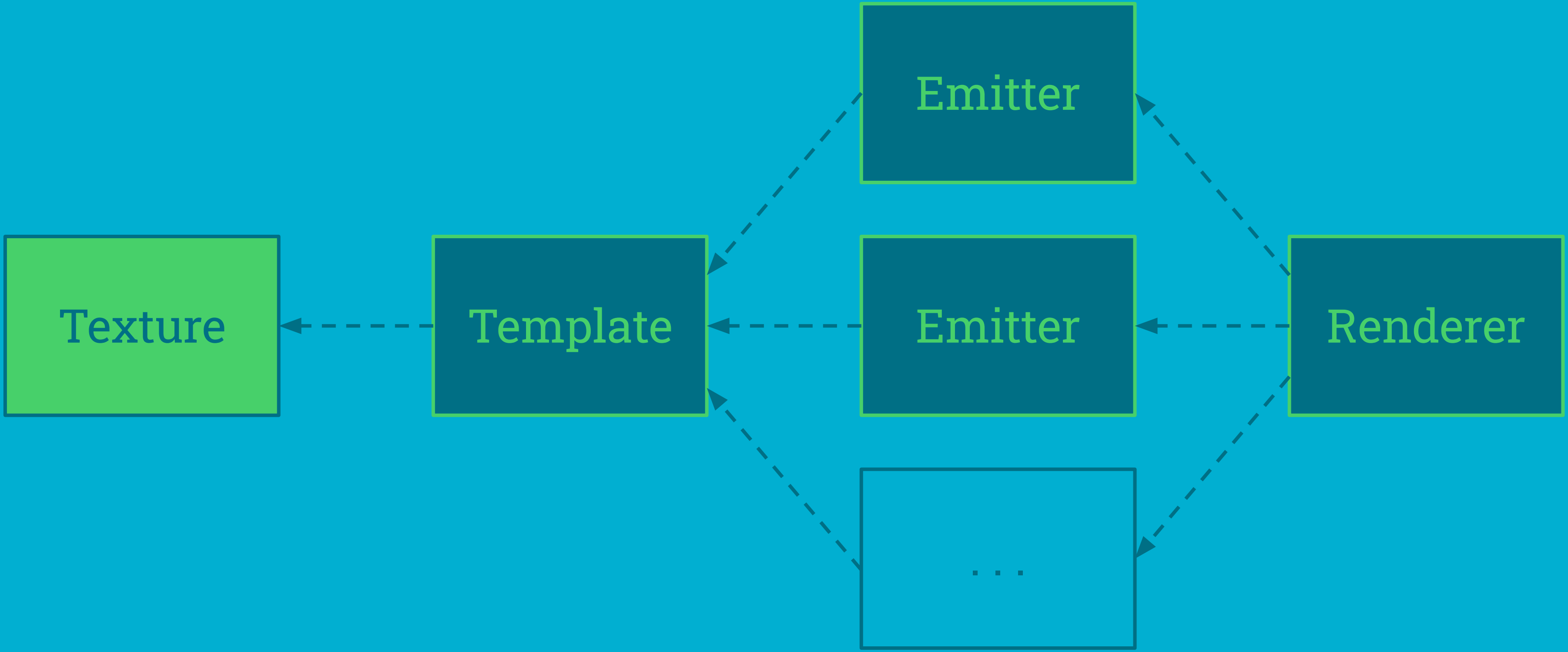
Particle system overview



Particle system overview



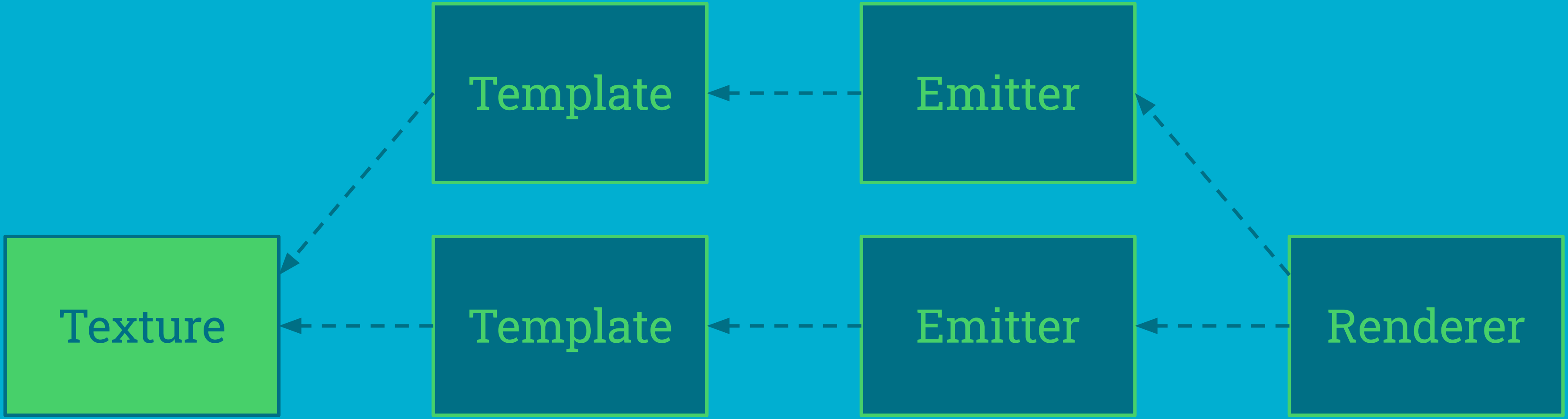
Particle system overview



Particle Templates

- Texture
- All parameters
- Precomputed values
- Bound funcs
- No logic, just data

Templates are not always 1-to-N



Can’t re-use a template if different params are needed

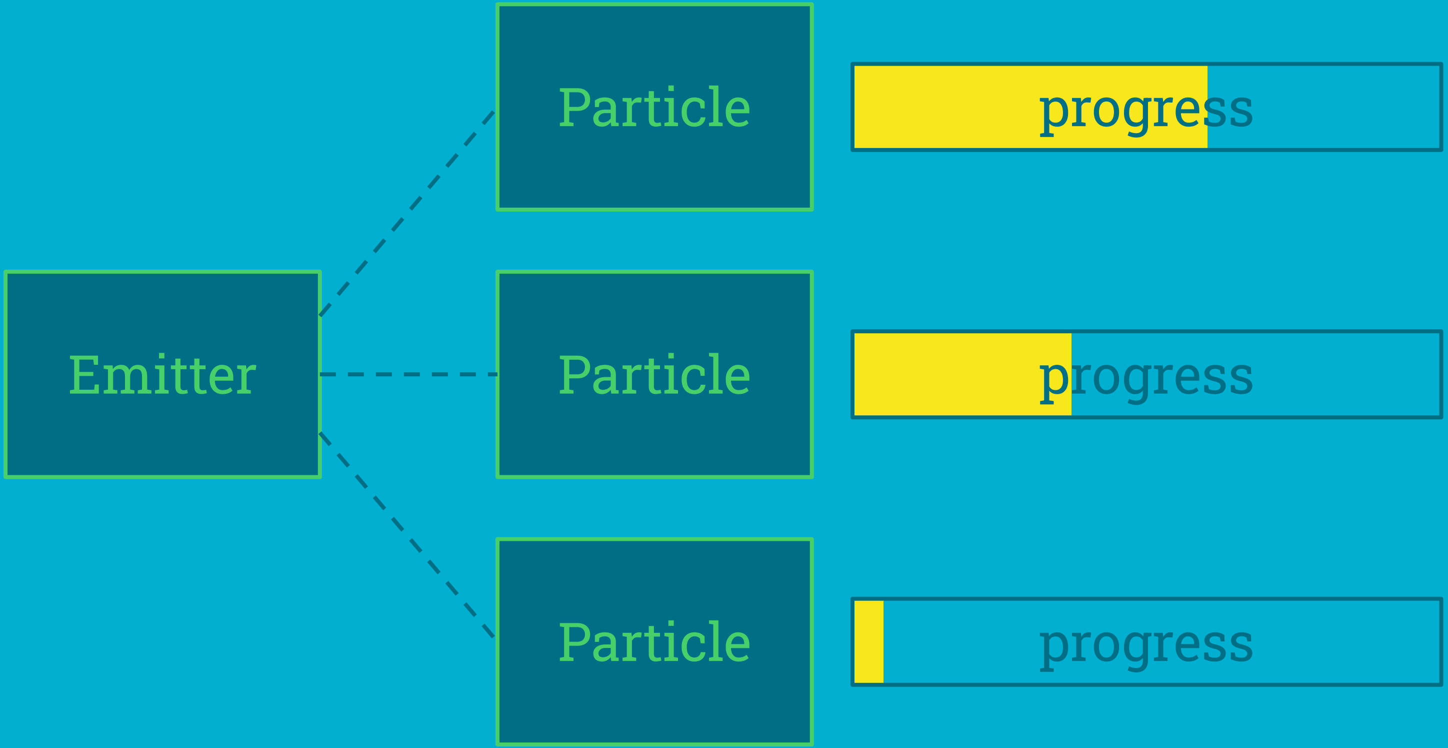
Particle Emitter

- Has a Template
- Has a world position
- Manages own Particles
- Advances Particle t
- Part of Update() tree

Particle Renderer

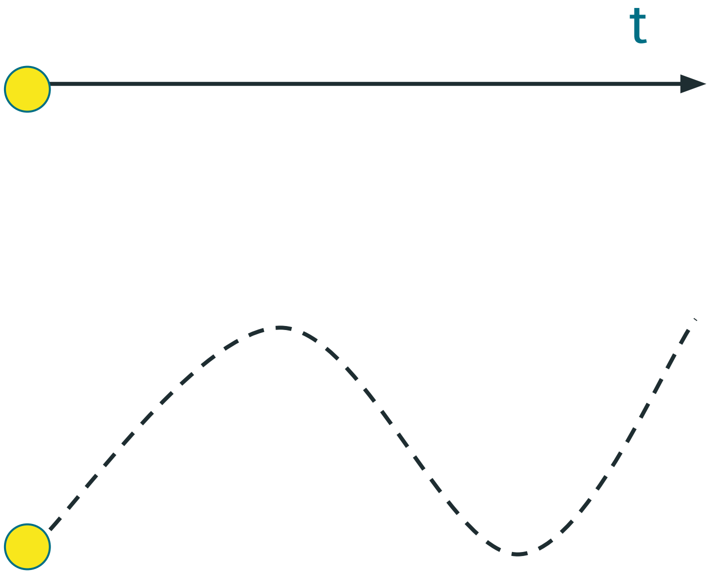
- Stores Emitters
- Batch-renders Particles
- Computes simulate(t)
- Part of Draw() tree

Particles

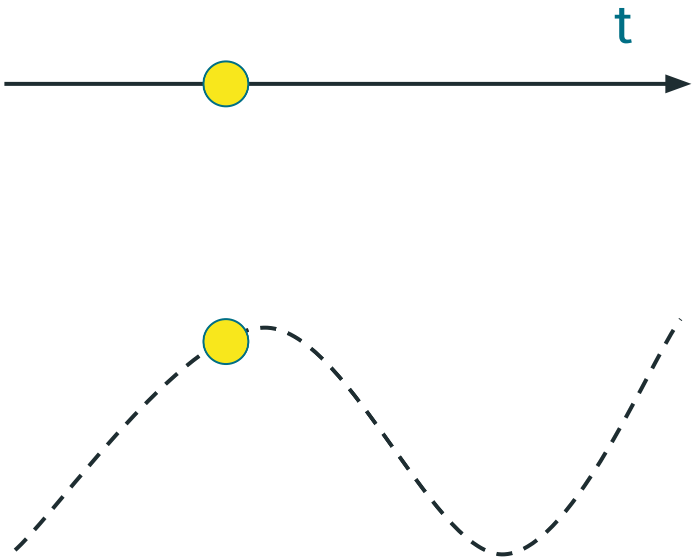


Emitter
Particle
Particle
Particle
progress
progress
progress

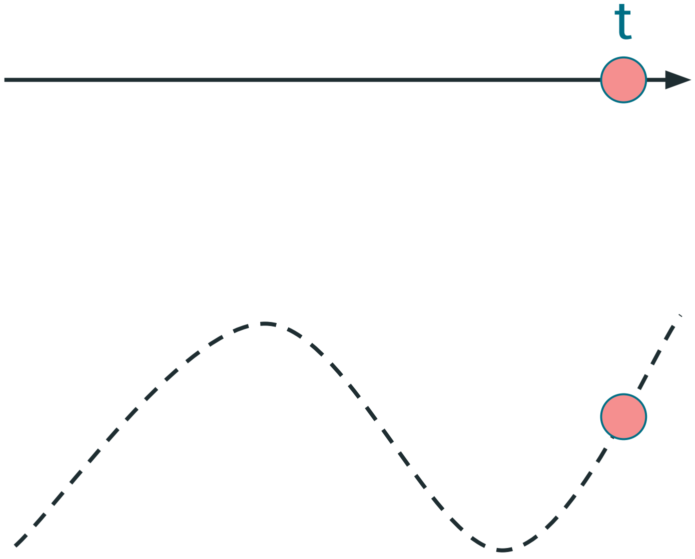
Particle Simulation



Particle Simulation



Particle Simulation



Agenda

- Intro
- VFX methods
- Particle system overview
- **Particles layout**
- Batch rendering
- GPU particles

Particle struct (the first draft)

```
type particle struct {
    progress float64 // t, [0, 1]
}
```

Particle struct (the first draft)

```
type particle struct {
    progress float64 // t, [0, 1]
}
```

Randomized lifetime?

Randomized speed?

...

Randomized direction?

Randomized color?

Randomized scaling?

Particle struct (naive version)

```go
type particle struct {
    progress float64
    lifetime time.Duration

    scaling float64
    speed   float64
    angle   float64
    color   color.RGBA

    pos [2]float64
}
```

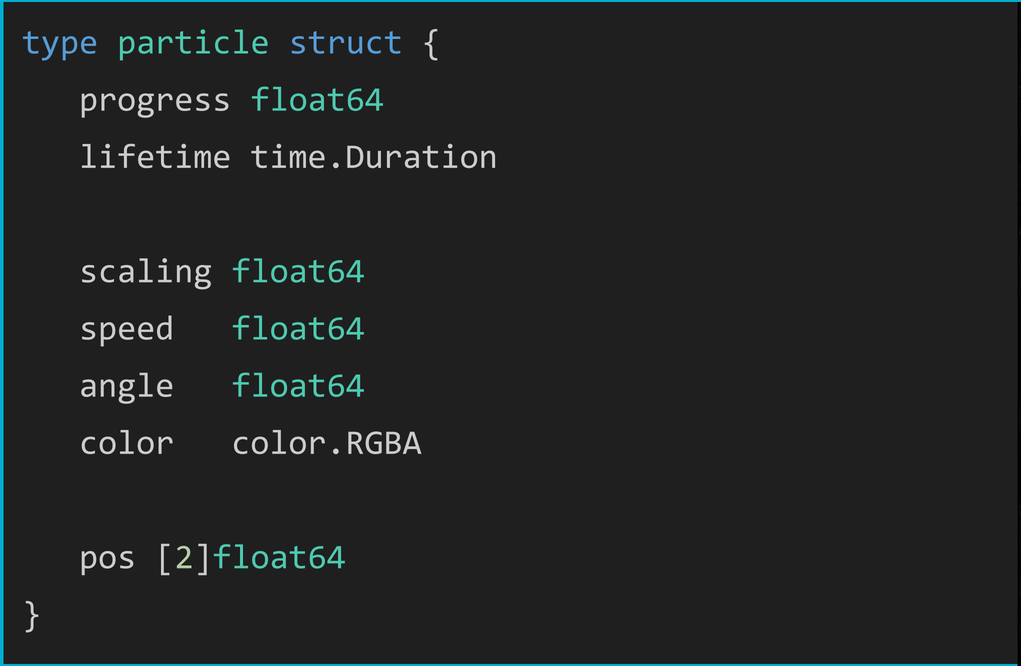
Particle struct (naive version)

```
type particle struct {
   progress float64
   lifetime time.Duration

   scaling float64
   speed   float64
   angle   float64
   color   color.RGBA

   pos [2]float64
}
```

64 bytes per particle

10000 particles = 640 kb

Particle struct (naive version)



```
type particle struct {
    progress float64
    lifetime time.Duration

    scaling float64
    speed   float64
    angle   float64
    color   color.RGBA

    pos [2]float64
}
```

Particle struct (improved version)

```
type particle struct {
    progress float32
    lifetime time.Duration

    scaling float32
    speed   float32
    angle   float32
    color   color.RGBA

    pos [2]float32
}
```

Reducing the precision,
float64 -> float32

40 bytes per particle
10000 particles = 400 kb
66% of original size

Particle struct

```
type particle struct {
   progress uint16
   lifetime uint16

   scaling float32
   speed   float32
   angle   float32
   color   color.RGBA

   pos [2]float32
}
```

Compressing time

1 unit = 1ms (delta*1000)
rounding error accumulation

28 bytes per particle
10000 particles = 280 kb
46% of original size

Particle struct

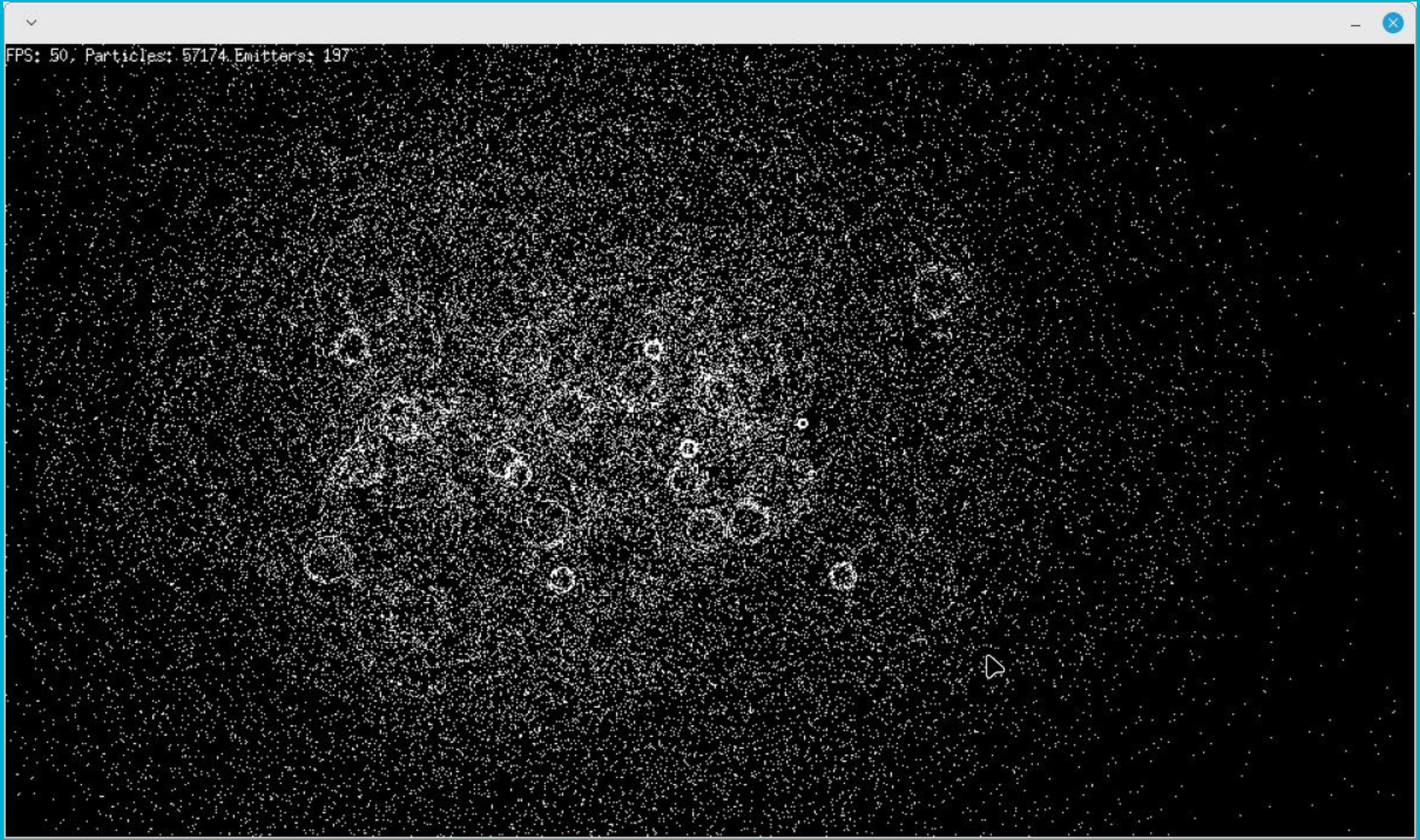
```
type particle struct {
    progress uint16
    lifetime uint16

    scaling uint8
    speed   uint8
    angle   uint8
    color   uint8

    pos [2]float32
}
```

Storing “palette indices”

16 bytes per particle

10000 particles = 160 kb

26% of original size



```
FPS: 50, Particles: 57174 Emitters: 197
```

Fitting scaling/speed/etc in uint8

Example: a speed in range of [100, 200]

- Store the min value of parameter => 100

Fitting scaling/speed/etc in uint8

Example: a speed in range of [100, 200]

- Store the min value of parameter => 100
- Calculate the “value step”: max-min/255 => 0.39

Fitting scaling/speed/etc in uint8

Example: a speed in range of [100, 200]

- Store the min value of parameter => 100
- Calculate the “value step”: max-min/255 => 0.39
- Generate a random “seed” value [0-255] => 60

Fitting scaling/speed/etc in uint8

Example: a speed in range of [100, 200]

- Store the min value of parameter => 100
- Calculate the “value step”: max-min/255 => 0.39
- Generate a random “seed” value [0-255] => 60
- Store that “seed” inside the uint8 field

Fitting scaling/speed/etc in uint8

Example: a speed in range of [100, 200]

- Store the min value of parameter => 100
- Calculate the "value step": max-min/255 => 0.39
- Generate a random "seed" value [0-255] => 60
- Store that "seed" inside the uint8 field
- The real value is computed as: min+(seed*step) => 123

Extra ideas

- Bucket-based particles
- Tiny particles (~8 bytes) with per-frame full re-calc
- Mapping user funcs into N precomputed points

Comparing with sprites (memory)

10000 particles ~ 160 kb

10000 sprite objects ~ 1360 kb

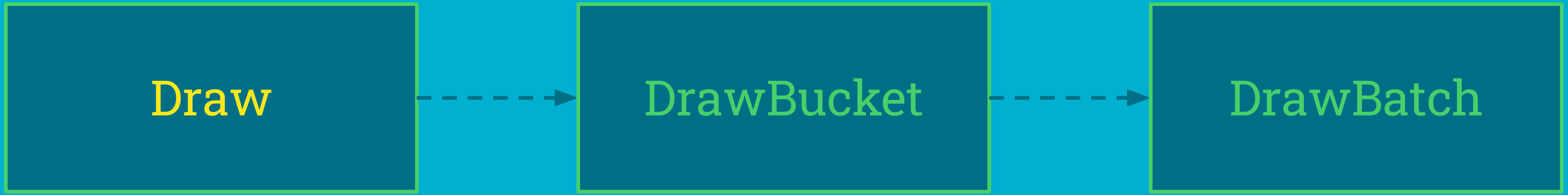
Sprites would also need to store extra state somewhere, like animation progress

Agenda

- Intro
- VFX methods
- Particle system overview
- Particles layout
- **Batch rendering**
- GPU particles

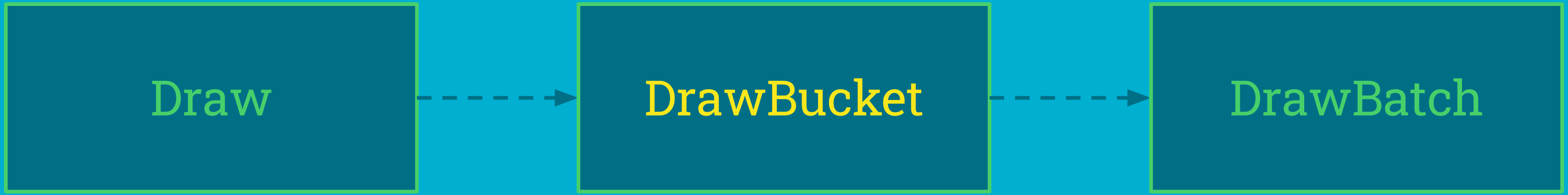
Drawing process

Draw walks buckets and calls DrawBucket on them



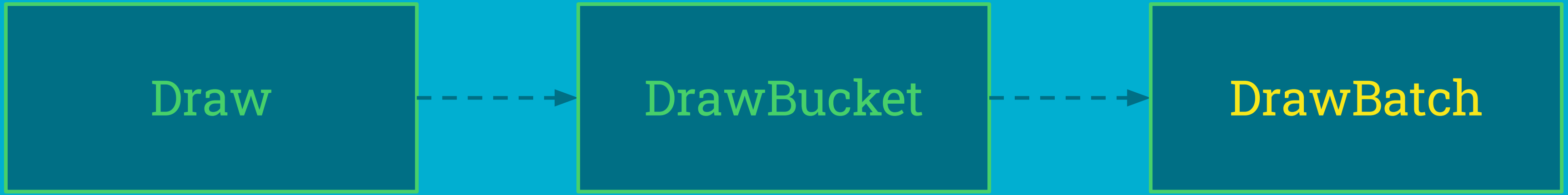
Drawing process

DrawBucket groups Emitters in batches, and then passes them to DrawBatch



Drawing process

DrawBatch generates vertices for every particle in the batch, and then calls Ebitengine's DrawTriangles



Renderer bucket

```
type rendererBucket struct {
   texture *ebiten.Image
   emitters []*Emitter
}
```

Comparing with sprites (rendering)

Particles: explicit & guaranteed batch rendering

Sprites: batch rendering is up to Ebitengine*

(*) Depends on various factors, like sprite draw order

Rendering method comparison (Ebitengine API)

DrawImage/particle: ~44000 particles at ~60 FPS*

DrawTriangles/batch: ~58000 particles at ~60 FPS*

(*) On my crappy laptop

Rendering method comparison (Ebitengine API)

DrawImage/particle: ~44000 particles at ~60 FPS*

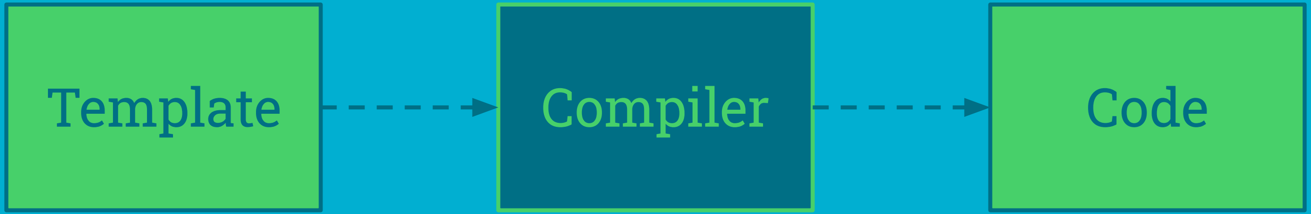
DrawTriangles/batch: ~58000 particles at ~60 FPS*

Godot (for comparison): ~65000 particles at ~60 FPS*

(*) On my crappy laptop

Code generation in particle systems

Code generation can generate a specialized particle renderer based on the template



Template

Compiler

Code

Specialized (generated) renderer example

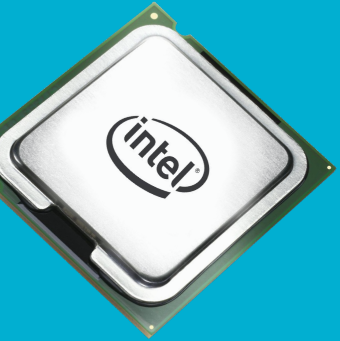
Given: template doesn’t use dynamic particle scaling

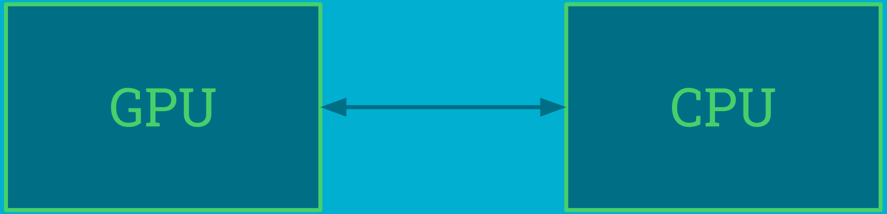
Result: the generated particle type has no “scaling” property, the generated renderer has no code managing the possibility of dynamic particle scaling.

Reduces memory and CPU requirements of particles

Agenda

- Intro
- VFX methods
- Particle system overview
- Particles layout
- Batch rendering
- **GPU particles**





GPU

CPU

GPU particle systems

- Usually more efficient than CPU systems

GPU particle systems

- Usually more efficient than CPU systems
- Usually less feature-rich than CPU systems

GPU particle systems

- Usually more efficient than CPU systems
- Usually less feature-rich than CPU systems
- Usually require a shader compilation at run-time

GPU particle systems

- Usually more efficient than CPU systems
- Usually less feature-rich than CPU systems
- Usually require a shader compilation at run-time
- Usually have a different (more complicated) API

GPU particle systems

- Usually more efficient than CPU systems
- Usually less feature-rich than CPU systems
- Usually require a shader compilation at run-time
- Usually have a different (more complicated) API
- Dynamic parameter are harder (or impossible)

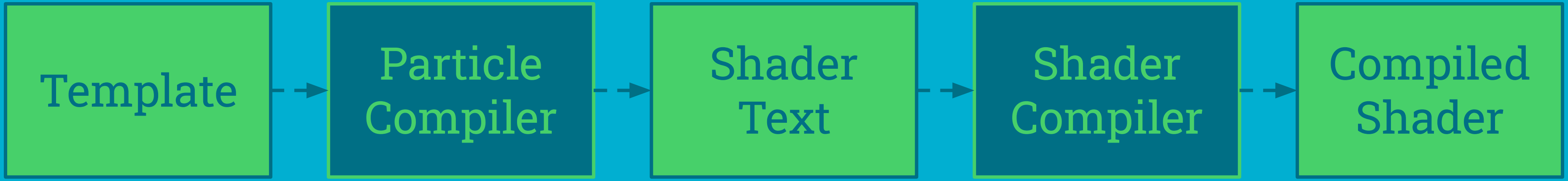
GPU particle systems

- Usually more efficient than CPU systems
- Usually less feature-rich than CPU systems
- Usually require a shader compilation at run-time
- Usually have a different (more complicated) API
- Dynamic parameter are harder (or impossible)

Can simulate much-much more particles at a lower cost

Shader generation at run-time

A Template is converted into a specialized Shader at run-time (which will be compiled further by GPU)



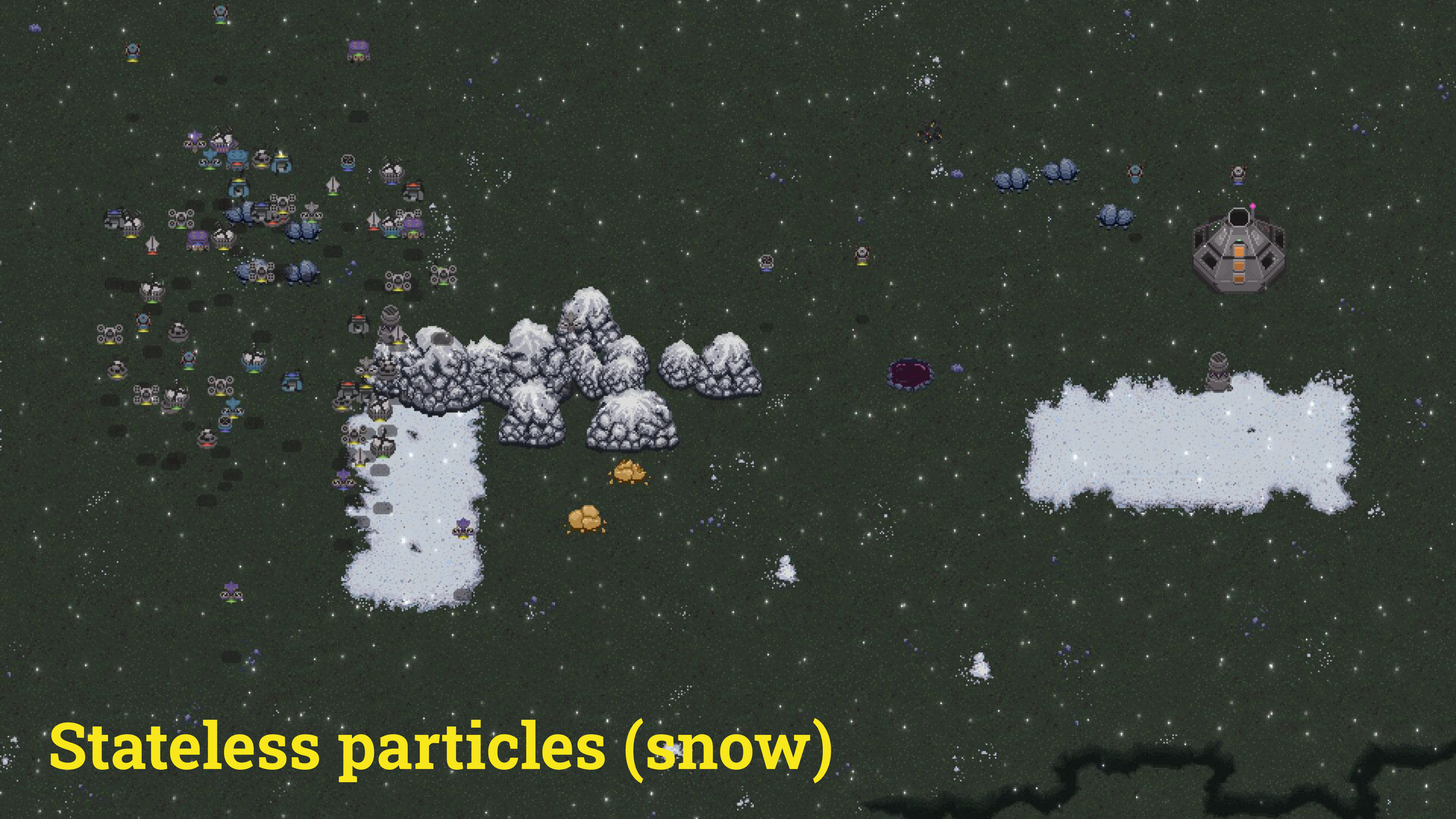
Kage and GPU stateless particles

- Like with a normal shader, particles depend on “noise”
- Particles don’t have individual state

Pros: can work with millions of particles for ~free

Cons: less features

Stateful particles are less efficient, but offer more feature



Stateless particles (snow)

Kage* and GPU stateful particles

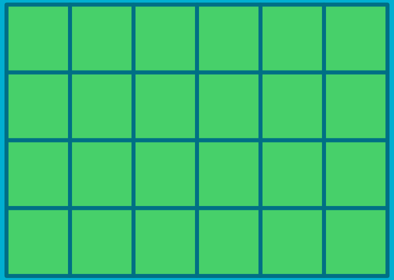
- No vertex shader support
- No efficient data buffers support

We can still try to create something, but it may be sub-optimal

(*) Kage is Ebitengine's shader language

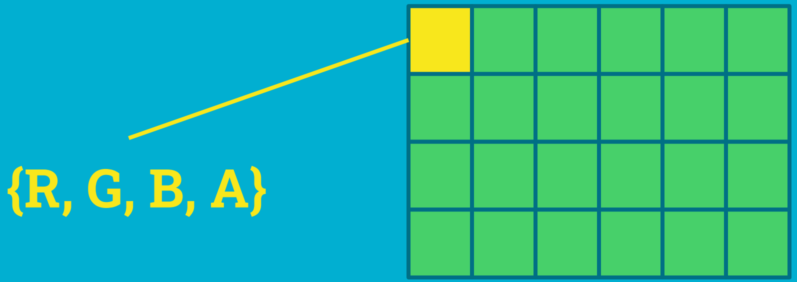
Textures as storage

A NxM texture can store information about N*M/K particles, where K is number of “pixels” per particle



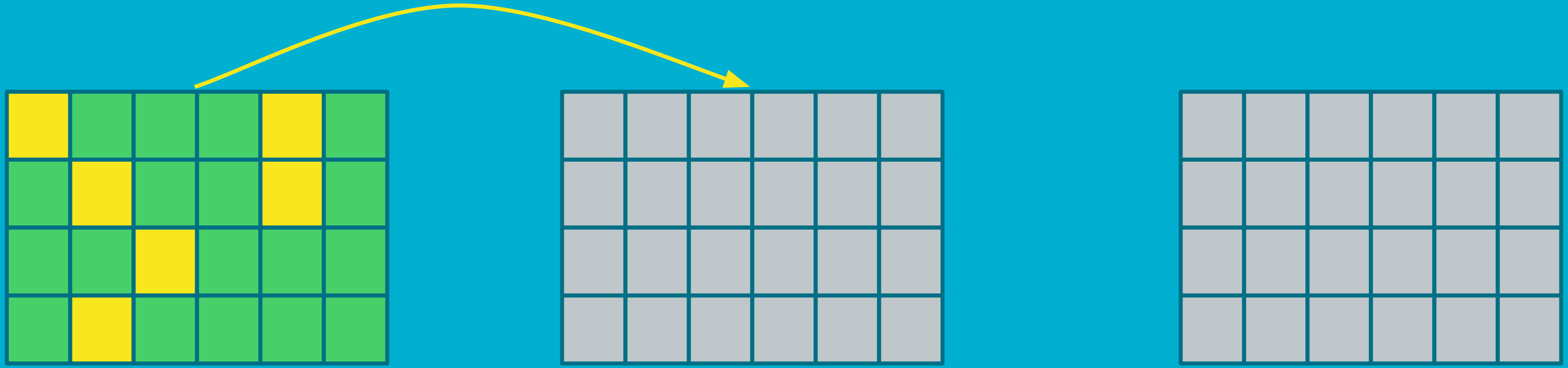
Textures as storage

Every pixel is vec4 - 4 float values of unspecified precision (usually 16 or 32 bits)



Rendering process

Calculate the new state by rendering a current state into a new state image



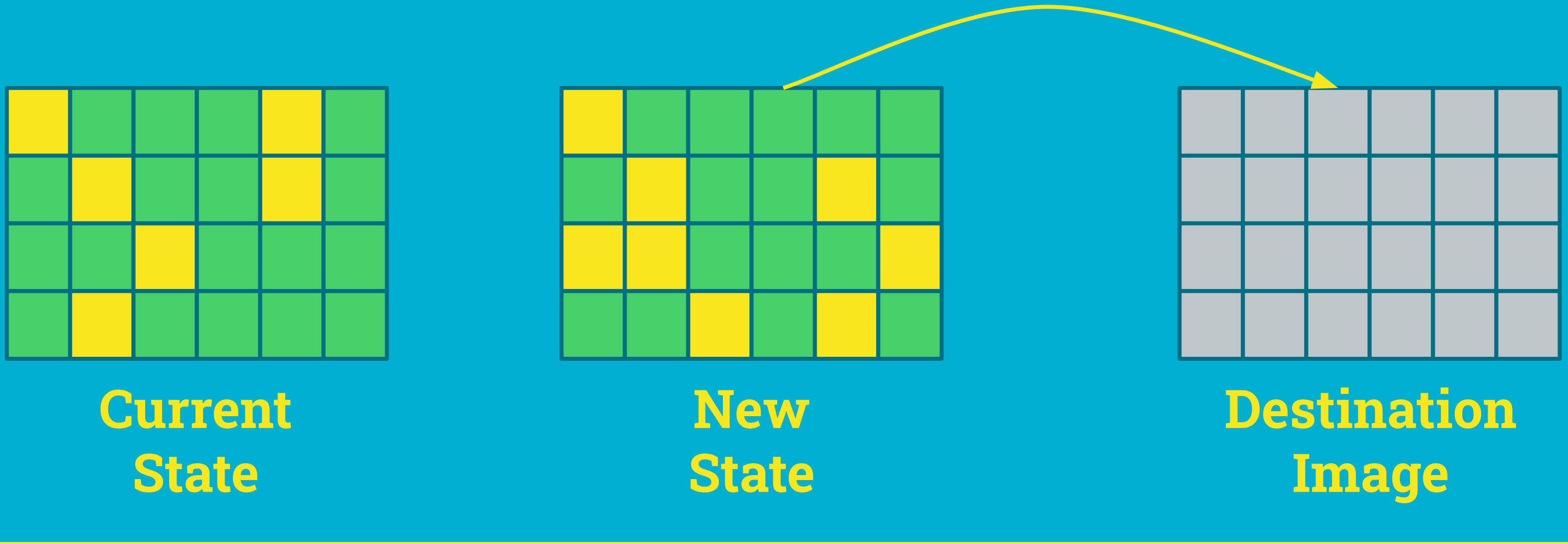
Current State

New State

Destination Image

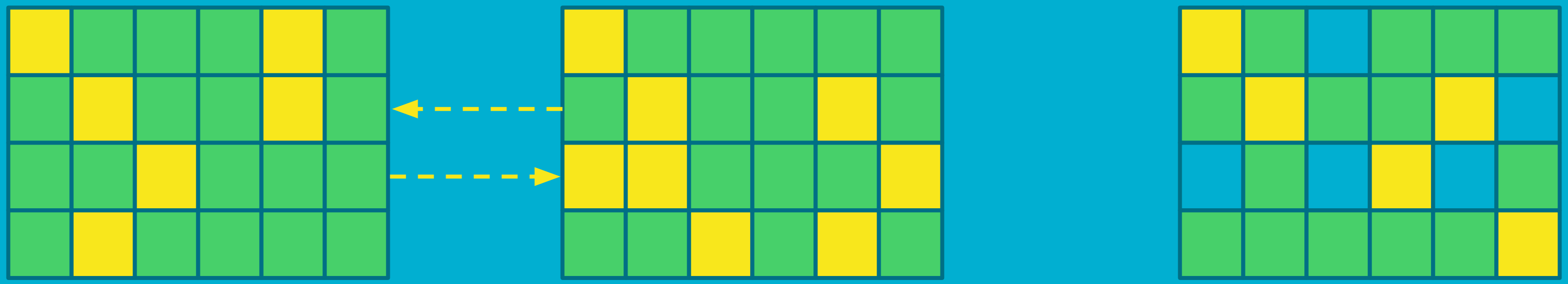
Rendering process

Render the particles using the new state texture onto the destination image



Rendering process

Swap the current and new state buffers (without copying)



Current State

New State

Destination Image

CPU vs GPU particles - which to use?

CPU vs GPU particles - which to use?

Which games benefit from particles?

Almost any game as they complement everything else.

Particles2D

Wishlist my game on Steam :)

quasilyte @ GoFunc 2024