Hacking a minimum ECS in Scala 3

張瑋修 Walter Chang

@weihsiu / weihsiu@gmail.com

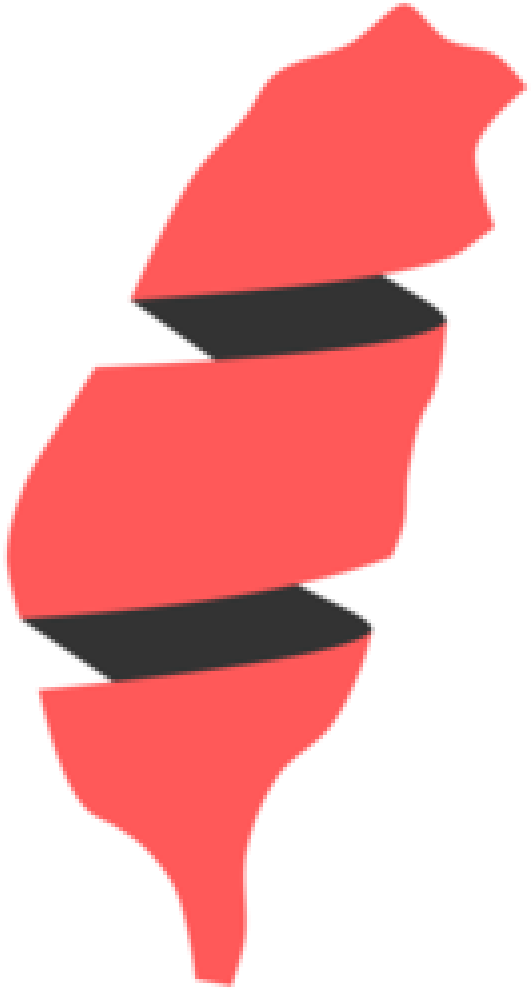
https://github.com/weihsiu/secs

Scala Taiwan

Scala Taiwan gitter channel

Scala Taiwan FB group

Scala Taiwan meetup



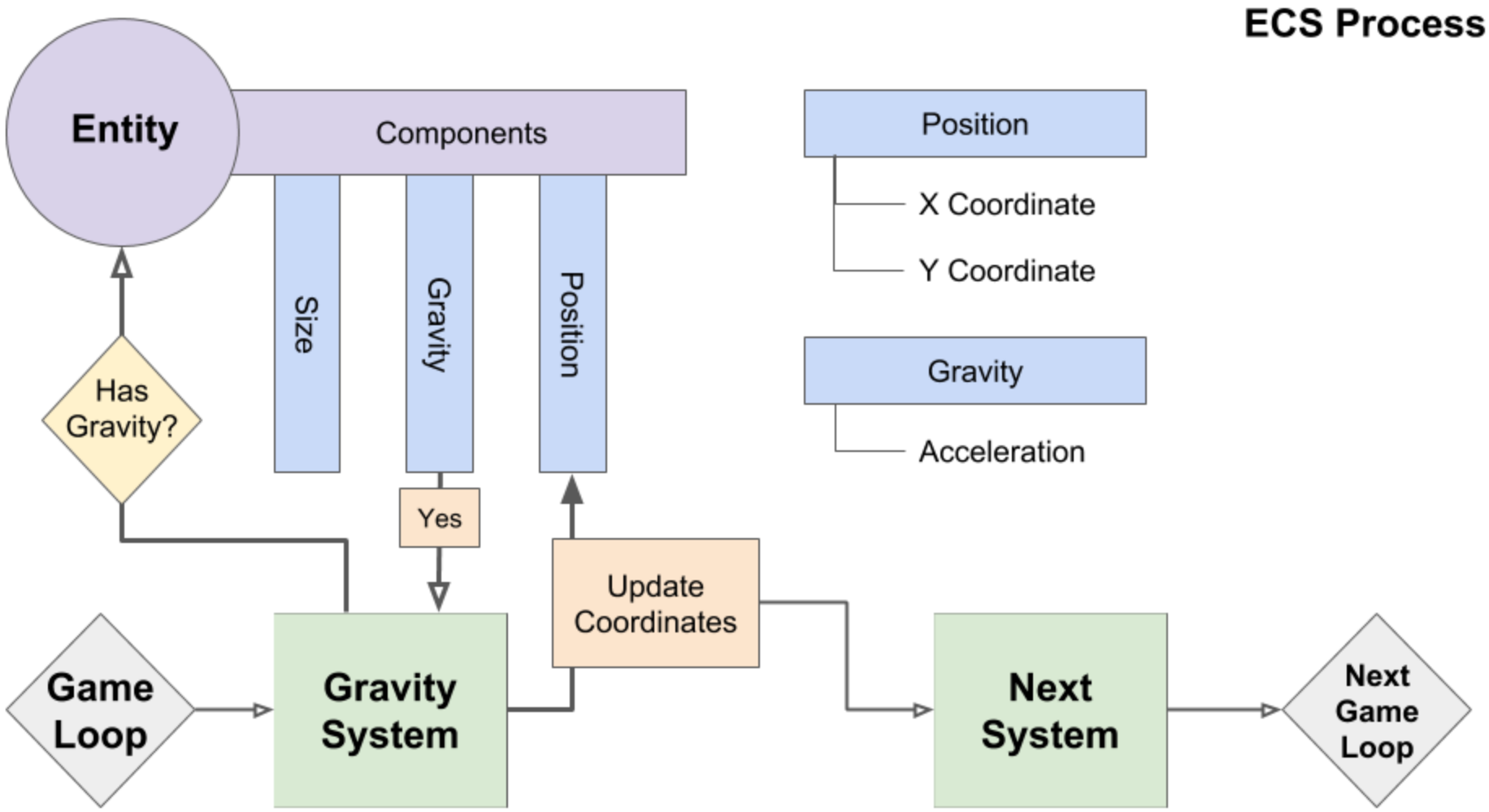
Agenda

- What is ECS?
- Overview
- Example #1
- Entity
- Component
- System
- Command
- Query
- Filter
- Event
- Secs
- Example #2
- Q&A

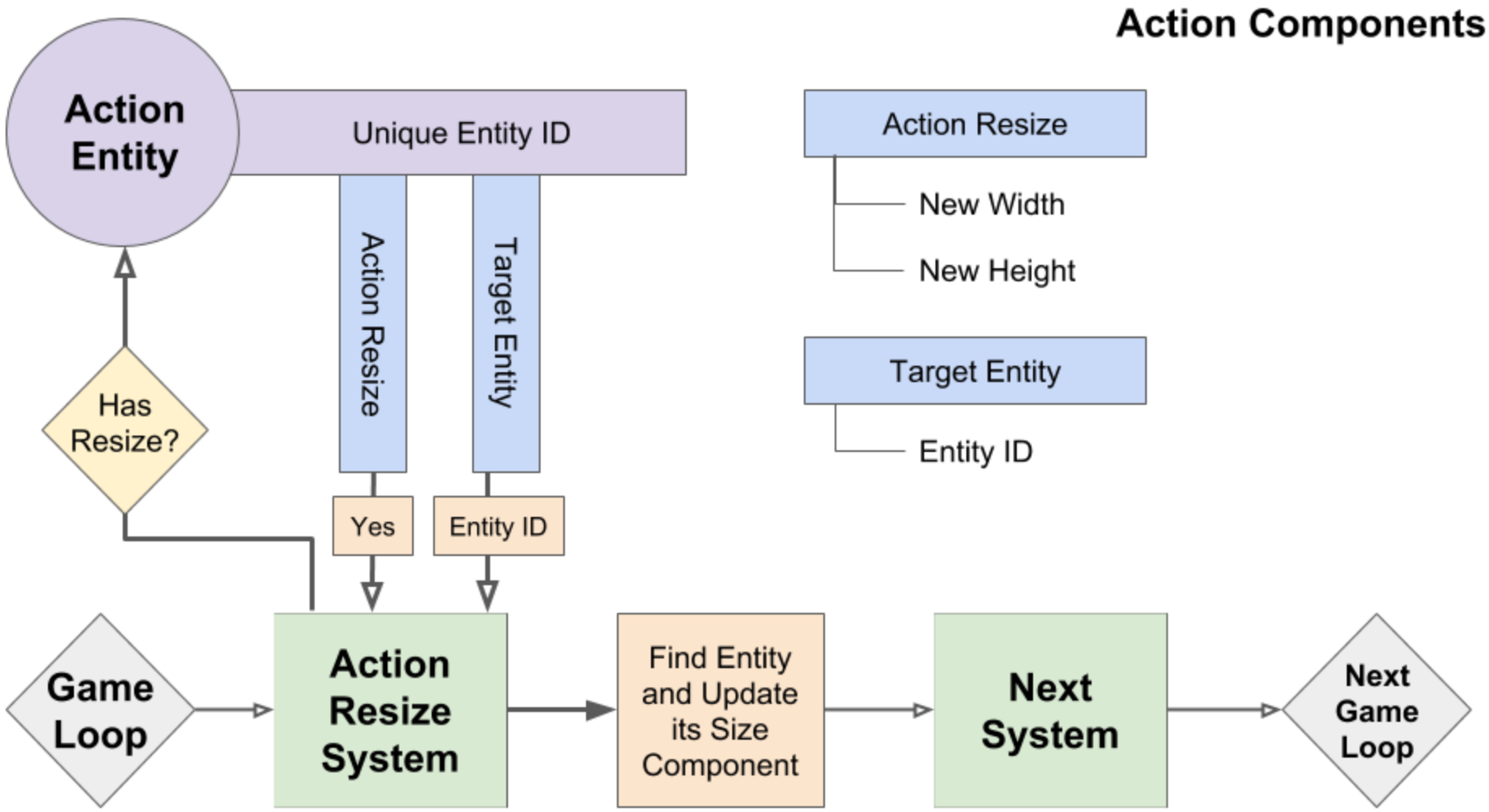
What is ECS?

- "Entity–component–system (ECS) is a software architectural pattern that is mostly used in video game development." -- Wikipeida
- It makes manipulating entities with the same types of components (attributes) very simple

From https://medium.com/@clevyr/entity-component-system-for-react-js-e3ab6e9be776



From https://medium.com/@clevyr/entity-component-system-for-react-js-e3ab6e9be776



Overview

- Scala 3 meta-programming is utilized to make ECS programming simpler and less error-prone
- Meta-programming in Scala 3 is made out of the following features:
  - Typelevel programming
    - Match types
    - scala.compiletime package
  - Inline functions
  - Macros (not used in this project)

Example #1

- Astroids
  - As an Java app
  - As a browser page

Entity

- Just an identifier
- Each entity can have multiple components associated with it

```
opaque type Entity = Int
```

Component

- Just Scala case classes that extend `Component` (marker trait) and derives `ComponentMeta`
- Immutable
- Builtin components:
  - EntityC
  - Label

```
case class Dimension(width: Double, height: Double) extends Component derives ComponentMeta
```

System

- Just Scala inline functions
- Used to insert/modify/delete components each entity is associated with
- The order of invoking them is important

```
inline def updateDimensions(using
      command: Command,
      query: Query1[(EntityC, Dimension)]
  ): Unit = ???
```

Command

- Used to insert/modify/delete components
  - Modify is done by replacing old components with new ones
- In order to manipulate components of an entity, you have to obtain it's `EntityCommand` first
- obtained via `using` with system functions

```
trait Command:
  def spawnEntity(): EntityCommand
  def entity(entity: Entity): EntityCommand
  def despawnEntity(entity: Entity): Command

trait EntityCommand:
  def entity: Entity
  def insertComponent[C <: Component](component: C)(using CM: ComponentMeta[C]): EntityCommand
  def updateComponent[C <: Component](update: C => C)(using CM: ComponentMeta[C]): EntityCommand
  def removeComponent[C <: Component]()(using CM: ComponentMeta[C]): EntityCommand
```

Query

- Used to select entities with required components
  - Select entities with required and/or optional components
  - Refine selection further with `Filter`
- All done in compile-time, thanks to type-level programming
- obtained via `using` with system functions

```
trait Query[CS <: Tuple, OS <: BoolOps]:
  inline def result: List[CS]
```

```
inline def updateSpaceship(time: Double)(using
      C: Command,
      Q: Query1[(EntityC, Direction, Movement, Option[CoolOff])]
  ): Unit =
    Q.result.foreach((e, d, m, cO) =>
      // e: EntityC, d: Direction, m: Movement, cO: Option[CoolOff]
      ???
    )
```

Filter

- Used to select entities that meet the filtering criteria
- Component alone is used to signify `contains` relationship
- `Added[]` and `Changed[]` is used to decorate `Component` to signify `added` and `changed` relationships
- Boolean operators can be used to form more complicated relationships
  - `¬` is equivalent to `not`
  - `∧` is equivalent to `and`
  - `∨` is equivalent to `or`
- All done in compile-time, thanks to type-level programming
- Specified in the second generic parameter of Query

```
inline def system(using Q: Query[(EntityC, Movement), Dimension ∧ Added[Rotation]]): Unit = ???
```

Event

- Builtin components with functions to send and receive events
- Useful for communication between system functions `
- Events are produced and consumed all in a single frame
    - Thus the order of system function invocation matters

```
case class EventSender[E]() extends Component:
  def send(event: E)(using CM: ComponentMeta[EventSender[E]], W: World): Unit = W.sendEvent(event)

case class EventReceiver[E]() extends Component:
  def receive(using CM: ComponentMeta[EventSender[E]], W: World): Iterable[E] = W.receiveEvents
```

Secs

- To tie everything together, implement `Secs`
- In init(), call system functions that will only be called in the very beginning
- In tick(), call system functions that will be invoke on every rendering frame
- in renderEntity(), every entity that satisfy `EntityStatus` will have a chance to render itself

```
enum EntityStatus:
  case Spawned, SpawnedAndAlive, Alive, AliveAndChanged, Despawned
trait Secs[SS <: Tuple]:
  type Worldly = World ?=> Unit
  def init(): Worldly
  def tick(time: Double): Worldly
  def beforeRender(): Unit
  def renderEntity(
      entity: Entity, status: EntityStatus,
      components: Components, previousComponents: => Components
  ): Unit
  def afterRender(): Unit
```

Example #2

- Retained
  - retained mode operations

Q&A

That's all and thank you for your attention

https://github.com/weihsiu/secs