encompass Documentation

Release 1.0

Evan Hemsley

Jan 15, 2019

Contents

encompass is a powerful engine-agnostic Hyper ECS framework ideal for game development.

CHAPTER 1

Getting Started

CHAPTER 2

API Reference

2.1 World

```
local World = require('encompass').World
local world = World:new()
```

A World is the glue that holds all the Engines, Entities, and Components together.

The World's `update` function drives the simulation and can be controlled from your engine's update loop.

It is a mistake to instantiate elements of **encompass** directly except for World. These elements should be created either through appropriate World functions, or by Engines.

It is not a good idea to add Engines to the world at runtime. Set Engines up at initialization time.

2.1.1 List of Functions

2.1.2 Function Reference

World:new()

Returns An instance of World.

World:create_entity()

Returns An instantiated Entity.

World:create_message(*message_type*, ...)

Arguments

- **message_type** (*prototype*) – A Message prototype.
- **...** – Alternating key and value arguments. The values should have types appropriate to their matching keys as defined by the Message prototype. If these do not match the Message prototype, an error will be thrown.

Returns An instantiated Message of the given prototype.

Example:

```
world:create_message(MyShipSpawnMessage,
    'x_position', 640,
    'y_position', 360,
    'x_velocity', 0,
    'y_velocity', 0
)
```

World:add_detector(*detector_type*)

Arguments

- **detector_type** (*prototype*) – A Detecter prototype.

Returns An instantiated Detecter of the given prototype.

World:add_spawner(*spawner_type*, *args*)

Arguments

- **spawner_type** (*prototype*) – A Spawner prototype
- **args** (*table*) – Arguments that will be passed to the Spawner `initialize` callback.

Returns An instantiated Spawner of the given prototype.

World:add_modifier(*modifier_type*, *args*)

Arguments

- **modifier_type** (*prototype*) – a Modifier prototype.
- **args** (*table*) – Arguments that will be passed to the Modifier `initailize` callback.

Returns An instantiated Modifier of the given prototype.

World:add_renderer(*renderer_type*)

Arguments

- **renderer_type** (*prototype*) – a Renderer prototype.

Returns An instantiated Renderer of the given prototype.

World:destroy_all_entities()

Destroys all entities that currently exist in the world. Useful for situations where you may want to clear the world to reset it without needing to re-initialize engines.

World:update(*dt*)

Arguments

- **dt** (*number*) – Delta time.

Updates the simulation based on given delta time, advancing the simulation by one frame.

The actions performed in a frame update are as follows:

- Activates/Deactivates marked Entities
- Updates Detectors
- Registers Messages with appropriate Spawners
- Updates Spawners
- Updates Modifiers
- Destroys marked Entities
- Recalculates draw order of components if any changed

Example:

```
function love.update(dt)
    world:update(dt)
end
```

World:draw(*canvas*)

Arguments

- **canvas** (*Canvas*) – A reference to a Canvas that all Renderers will be given.

Draws to the given Canvas based on the composition of all Renderers in the simulation.

If you're not using LOVE, this could be your engine's concept of a Framebuffer or anything you can send draw calls to.

Example:

```
function love.draw()
    love.graphics.setCanvas(world_canvas)
    love.graphics.clear()
    love.graphics.setCanvas()
    love.graphics.clear()

    world:draw(world_canvas)

    love.graphics.setCanvas()
    love.graphics.setBlendMode('alpha', 'premultiplied')
    love.graphics.setColor(1, 1, 1, 1)
    love.graphics.draw(world_canvas)
end
```

2.2 Entities

2.2.1 Entity

```
local World = require("encompass").World
local world = World:new()
local Entity = world:create_entity()
```

An Entity is composed of a unique internal ID and a collection of *Components*.

Entities do not have any implicit properties or behavior, but are granted these by their collection of Components.

There is no limit to the amount of Components an Entity can have, and Entities can have any number of Components of a given type.

Entities are *active* by default and can be *deactivated*. While deactivated, Entities are still tracked by Engines, but the Engines temporarily ignore them. Note that the activation status of an Entity does not affect the activation status of its Components.

Defining logic on an Entity is an anti-pattern. *Do not do this.*

List of Functions

- *Entity:add_component(component_type, fields)*
- *Entity:get_component(component_type)*
- *Entity:get_components(component_type)*
- *Entity:has_component(component_type)*
- *Entity:remove_component(component)*
- *Entity:destroy()*
- *Entity:activate()*
- *Entity:deactivate()*

Function Reference

Entity:add_component(*component_type*, ...)

Arguments

- **component_type** (*Component*) – A Component prototype.
- `...` – Alternating key and value arguments. The values should have types appropriate to their matching keys as defined by the Component prototype. If these do not match the Message prototype, an error will be thrown.

Returns An instantiated Component of the given prototype.

Example:

```
local World = require('encompass').World
local Component = require('encompass').Component
local PositionComponent = Component.define('PositionComponent', {
    x = 'number',
```

(continues on next page)

(continued from previous page)

```
    y = 'number'
})

local world = World:new()
local entity = world:create_entity()
entity:add_component(PositionComponent,
    'x', 0,
    'y', 0
)
```

Entity:get_components(*component_type*)

Arguments

- **component_type** (*prototype*) – A Component prototype.

Returns An array-style table containing references to each component of the given Component prototype on the Entity.

Entity:get_component(*component_type*)

Arguments

- **component_type** (*prototype*) – A Component prototype.

Returns A reference to an arbitrary first instantiated Component belonging to the object.

This is a convenience method for the common case of having only one component of a certain type on an Entity.

Entity:has_component(*component_type*)

Arguments

- **component_type** (*prototype*) – A Component prototype.

Returns `boolean` Whether or not the Entity has a component of the given Component prototype.

Entity:remove_component(*component*)

Note An error is thrown if the Entity does not contain the component.

Arguments

- **component** (*Component*) – An instantiated Component.

Removes the given component from the entity.

Runs the *destroy* callback on the component.

Entity:destroy()

Note The entity is not destroyed until the end of the current frame.

Destroys the Entity and all its components.

Entity:activate()

Note The entity is not activated until the end of the current frame.

Activates the Entity if it was previously deactivated. Runs `activate` callback on all of its components.

Entity:deactivate()

Note The entity is not deactivated until the end of the current frame.

Deactivates the Entity if it was previously activated. Runs `deactivate` callback on all of its components.

2.2.2 PooledEntity

A PooledEntity is a special kind of *Entity* which is created by a *PooledSpawner*.

Instead of creating and destroying Entities at runtime, a PooledSpawner generates a fixed amount of PooledEntities at initialization time, and then handles them appropriately when they are deactivated. This pattern is useful for entities with components that are expensive to generate at runtime.

PooledEntities are automatically activated by PooledSpawners when their `spawn` callback is called.

PooledEntities should be deactivated instead of destroyed. It is an error to destroy a PooledEntity.

2.3 Components

2.3.1 Component

```
local Component = require("encompass").Component
```

A Component can be thought of as a collection of data which can be attached to an entity.

Component types are defined by their required field types and optional field types, which are used to instantiate the Components themselves. See the *Field Type Reference* for a list of valid field types.

Components are *active* by default and can be deactivated. While inactive, Entities containing the Component will not be tracked by Detectors that track the Component prototype. This is useful for situations where you may want to temporarily disable a component without destroying information. Note that the activation status of a Component does not affect the activation status of its Entity.

Components do not contain any logic, but they do have optional callbacks to handle any side-effects, for example, creating or destroying bodies in a physics world.

Defining non-side-effect logic on a Component is an anti-pattern. *Do not do this.*

Function Reference

`Component.`**`define`**(*name*, *required_field_types*, *optional_field_types*)

Arguments

- **`name`** (*`string`*) – The name of the Component prototype.
- **`required_field_types`** (*`table`*) – A table where the keys are field names and the values are types. If one of these fields is missing when the Component is instantiated, or a field is passed to the component with an incorrect type, an error is thrown.
- **`optional_field_types`** (*`table`*) – A table where the keys are field names and the values are types. If a field is passed to the component with an incorrect type, an error is thrown.

Returns A new Component prototype.

Example:

```
local Component = require("encompass").Component
local PositionComponent = Component.define('PositionComponent', {
    x = 'number',
    y = 'number'
```

(continues on next page)

(continued from previous page)

```
})

return PositionComponent
```

Component:activate()

Activates the Component. Registers attached Entity with appropriate Detectors. Does nothing if the Component is already active.

Component:deactivate()

Deactivates the Component. Unregisters attached Entity with appropriate Detectors. Does nothing if the Component is already inactive.

Callback Reference

Component:initialize() Runs when the Component is added to an Entity.

Component:on_destroy() Runs when the Component is removed from an Entity, or the attached Entity is destroyed.

Component:on_activate() Runs when the attached Entity is activated.

Component:on_deactivate() Runs when the attached Entity is deactivated.

2.3.2 DrawComponent

```
local DrawComponent = require('encompass').DrawComponent
```

A DrawComponent is a special kind of *Component* intended for use by Renderers.

The only difference is that it implicitly contains a `layer` property so it can be ordered properly by the draw system.

It is very expensive to modify the `layer` property at runtime. Do so sparingly.

Example

```
local DrawComponent = require('encompass').DrawComponent

local DrawCanvasComponent = DrawComponent.define('DrawCanvasComponent', {
    canvas = 'userdata',
    w = 'number',
    h = 'number'
})

entity:add_component(DrawCanvasComponent, {
    canvas = love.graphics.newCanvas(1280, 720)
    w = 1280,
    h = 720,
    layer = -5
})
```

2.4 Messages

Similar to *Components*, Messages are collections of data. Message prototypes are defined by their inherited prototype and their required and optional fields, which are used to instantiate the Messages themselves. See the *Field Type Reference* for a list of valid field types.

Messages are used to send data to Engines so they can manipulate the game state accordingly.

Unlike Components, Messages are temporary and destroyed at the end of each frame.

For performance reasons, it is discouraged to create new tables to pass to messages, as this will cause garbage collection issues.

Defining any logic on a Message is an anti-pattern. Messages should not need to directly trigger any side-effects. *Do not do this.*

2.4.1 ComponentMessage

```
local ComponentMessage = require("encompass").ComponentMessage
```

A ComponentMessage is a kind of message that is consumed by *ComponentModifiers* to modify a particular component.

A ComponentMessage implicitly contains a `component` key, but it can and should be overwritten with a more specific Component subtype.

Function Reference

`ComponentMessage`**`.define`**(*name*, *required_field_types*, *optional_field_types*)

Arguments

- **`name`** (*`string`*) – The name of the ComponentMessage prototype.
- **`required_field_types`** (*`table`*) – A table where the keys are field names and the values are types. If one of these fields is missing when the ComponentMessage is instantiated, or a field is passed to the message with an incorrect type, an error is thrown.
- **`optional_field_types`** (*`table`*) – A table where the keys are field names and the values are types. If a field is passed to the message with an incorrect type, an error is thrown.

Returns A new ComponentMessage prototype.

Example:

```
local ComponentMessage = require('encompass').ComponentMessage
local TransformComponent = require('game.components.transform')

return ComponentMessage.define('MotionMessage', {
    component = TransformComponent,
    x_velocity = 'number',
    y_velocity = 'number',
    angular_velocity = 'number',
    instant_linear = 'boolean',
    instant_angular = 'boolean'
})
```

2.4.2 EntityMessage

```
local EntityMessage = require("encompass").EntityMessage
```

An EntityMessage is a kind of message that is consumed by *EntityModifiers* to modify a particular entity.

An EntityMessage implicitly contains an `entity` key.

Function Reference

`EntityMessage.`**`define`**(*name*, *required_field_types*, *optional_field_types*)

Arguments

- **`name`** (*`string`*) – The name of the EntityMessage prototype.
- **`required_field_types`** (*`table`*) – A table where the keys are field names and the values are types. If one of these fields is missing when the EntityMessage is instantiated, or a field is passed to the message with an incorrect type, an error is thrown.
- **`optional_field_types`** (*`table`*) – A table where the keys are field names and the values are types. If a field is passed to the message with an incorrect type, an error is thrown.

Returns A new EntityMessage prototype.

Example:

```
local EntityMessage = require("encompass").EntityMessage
return EntityMessage.define('DeactivateEntityMessage')
```

2.4.3 SpawnMessage

```
local SpawnMessage = require("encompass").SpawnMessage
```

A SpawnMessage is a kind of message that is consumed by *Spawners* to spawn new entities.

Function Reference

`SpawnMessage.`**`define`**(*name*, *required_field_types*, *optional_field_types*)

Arguments

- **`name`** (*`string`*) – The name of the SpawnMessage prototype.
- **`required_field_types`** (*`table`*) – A table where the keys are field names and the values are types. If one of these fields is missing when the SpawnMessage is instantiated, or a field is passed to the message with an incorrect type, an error is thrown.
- **`optional_field_types`** (*`table`*) – A table where the keys are field names and the values are types. If a field is passed to the message with an incorrect type, an error is thrown.

Returns A new SpawnMessage prototype.

2.4.4 StateMessage

```
local StateMessage = require("encompass").StateMessage
```

In certain cases, you may want to be able to broadcast information about the state of the game to multiple Modifiers without it being consumed by any one Modifiers in particular. StateMessage exists for this reason.

When a StateMessage is created, it is visible to all Modifiers that track its prototype.

Defining any logic on a StateMessage is an anti-pattern. *Do not do this.*

Function Reference

StateMessage.**define**(*name*, *required_field_types*, *optional_field_types*)

Arguments

- **name** (*string*) – The name of the StateMessage prototype.
- **required_field_types** (*table*) – A table where the keys are field names and the values are types. If one of these fields is missing when the StateMessage is instantiated, or a field is passed to the message with an incorrect type, an error is thrown.
- **optional_field_types** (*table*) – A table where the keys are field names and the values are types. If a field is passed to the message with an incorrect type, an error is thrown.

Returns A new StateMessage prototype.

Example

time_dilation_state_message.lua

```
local StateMessage = require('lib.encompass').StateMessage

return StateMessage.define(
    'TimeDilationStateMessage',
    {
        factor = 'number'
    }
)
```

music_modifier.lua

```
local MusicMessage = require('hyperspace.messages.music')
local TimeDilationStateMessage = require('hyperspace.messages.state.time_dilation')

local ComponentModifier = require('lib.encompass').ComponentModifier
local MusicModifier = ComponentModifier.define(
    'MusicModifier',
    MusicMessage,
    { TimeDilationStateMessage }
)

function MusicModifier:modify(music_component, frozen_fields, message, dt)
    local time_dilation_state_message = self:get_state_
↪message(TimeDilationStateMessage)

    local factor = 1
```

(continues on next page)

(continued from previous page)

```
    if time_dilation_state_message ~= nil then
        factor = time_dilation_state_message.factor
    end

    dt = dt * factor

    local source = frozen_fields.source
    source:setPitch(factor)
    music_component.time = frozen_fields.time + dt
end

return MusicModifier
```

2.4.5 SideEffectMessage

```
local SideEffectMessage = require("encompass").SideEffectMessage
```

A SideEffectMessage is a kind of message that is consumed by *SideEffecters* to modify game state that isn't captured by any particular Entities or Components. For example, if your game has multiple top-level states with separate Worlds, changing the top-level state would be a side effect.

Side effects should be used as an absolute last resort. Use them only if there is no way to produce the behavior with other types of Engines.

Function Reference

SideEffectMessage.**define**(*name*, *required_field_types*, *optional_field_types*)

Arguments

- **name** (*string*) – The name of the SideEffectMessage prototype.
- **required_field_types** (*table*) – A table where the keys are field names and the values are types. If one of these fields is missing when the SideEffectMessage is instantiated, or a field is passed to the message with an incorrect type, an error is thrown.
- **optional_field_types** (*table*) – A table where the keys are field names and the values are types. If a field is passed to the message with an incorrect type, an error is thrown.

Returns A new SideEffectMessage prototype.

2.5 Engines

An Engine is the **encompass** notion of a System.

There are two major types of Engines.

Detectors and Renderers are responsible for iterating over *Entities* that contain particular components and producing Messages in response. They do not modify Entities or Components directly.

Spawners and Modifiers are responsible for consuming Messages and modifying the game state in response. They do not read from Entities or Components.

2.5.1 Detecter

```
local Detecter = require('encompass').Detecter
```

A Detecter is an Engine which is responsible for reading the game world and producing *Messages*.

Detectors are defined by the *Component* prototype(s) they track, and the `detect` function they implement.

It is an anti-pattern to modify Entities or Components from inside a Detecter. *Do not do this.*

Function Reference

`Detecter.`**`define`**(*name*, *component_types*)

Arguments

- **`name`** (*`string`*) – The name of the Detecter.
- **`component_types`** (*`table`*) – An array-style table containing Component types.

Defines a Detecter that will track Entities which contain _all_ of the given Component types.

`Detecter:create_message`(*message_type*, ...)

Arguments

- **`message_type`** (*`prototype`*) – A Message prototype.
- **`...`** – Alternating key and value arguments. The values should have types appropriate to their matching keys as defined by the Message prototype. If these do not match the Message prototype, an error will be thrown.

Returns An instantiated Message of the given prototype.

`Detecter:detect`(*entity*)

Arguments

- **`entity`** (*`Entity`*) – A reference to an entity which is tracked by the Detecter.

Every frame, this callback runs for each entity tracked by the Detecter. The programmer must override this callback or an error will be thrown.

Example

```
local TransformComponent = require("game.components.transform")
local VelocityComponent = require("game.components.velocity")
local MotionMessage = require("game.messages.motion")

local Detecter = require("encompass").Detecter
local MovementDetector = Detecter.define('MovementDetector', {
    TransformComponent, VelocityComponent
})

function MovementDetector:detect(entity)
    local transform_component = entity:get_component(TransformComponent)
    local velocity_component = entity:get_component(VelocityComponent)

    self:create_message(MotionMessage,
        'component',        transform_component,
```

(continues on next page)

(continued from previous page)

```
        'x_velocity',        velocity_component.linear.x,
        'y_velocity',        velocity_component.linear.y,
        'angular_velocity', velocity_component.angular,
        'instant_linear',    false,
        'instant_angular',   false
    )
end

return MovementDetector
```

2.5.2 Spawner

```
local Spawner = require("encompass").Spawner
```

A Spawner is an Engine which is responsible for reading a *SpawnMessage* to create an *Entity*.

Spawners are defined by the Message prototype they track, and the `spawn` function they implement.

It is an anti-pattern to read Entities or Components from inside a Spawner. *Do not do this.*

Function Reference

`Spawner.`**`define`**(*name*, *message_type*)

Arguments

- **`name`** (*`string`*) – The name of the Spawner.
- **`message_type`** (*`prototype`*) – A Message prototype that should be tracked by the Spawner.

Defines a Spawner that will track the given Message prototype.

`Spawner:initialize`(...)

Arguments

- `...` (*`args`*) – An arbitrary list of arguments.

This callback is triggered when the Spawner is added to the World. Useful for Spawners that need to deal with side effects, such as physics worlds or particle systems.

`Spawner:create_entity`()

Returns `Entity` An instantiated entity.

`Spawner:spawn`(*message*)

Arguments

- **`message`** (*`SpawnMessage`*) – A reference to a message that has been tracked by the Spawner.

This callback is triggered when a Message of the specified prototype is produced by a Detecter. The programmer must override this function or an error will be thrown.

Example

```
local SoundComponent = require('game.components.sound')
local SoundSpawnMessage = require('game.messages.spawners.sound')

local Spawner = require('encompass').Spawner
local SoundSpawner = Spawner.define('SoundSpawner', SoundSpawnMessage)

function SoundSpawner:spawn(sound_spawn_message)
    local source = sound_spawn_message.source

    local sound_entity = self:create_entity()
    sound_entity:add_component(SoundComponent, {
        source = source
    })

    source:play()
end

return SoundSpawner
```

2.5.3 PooledSpawner

```
local PooledSpawner = require("encompass").PooledSpawner
```

A PooledSpawner is a special kind of *Spawner* which utilizes object pooling. Instead of creating and destroying Entities at runtime, the PooledSpawner generates a fixed amount of a special kind of Entity called a *PooledEntity* at initialization time, and then handles them appropriately when they are deactivated. This pattern is useful for entities with components that are expensive to generate at runtime.

A pool overflow callback is called when there are no more inactive entities available in the pool. There are four types of pool overflow behaviors included by **encompass**, but you are free to define your own as well.

Unlike other types of engines, it is OK to read from PooledEntities in a PooledSpawner in the `spawn` method.

Note: it is an error to call `destroy()` on a PooledEntity.

Overflow Behavior Reference

`PooledSpawner.OverflowBehaviors.`**`fallible`**
: Does nothing. The default behavior.

`PooledSpawner.OverflowBehaviors.`**`cycle`**
: Takes an arbitrary active element and calls it to be spawned.

`PooledSpawner.OverflowBehaviors.`**`expand`**
: Expands the pool count by 1 and generates a new entity.

`PooledSpawner.OverflowBehaviors.`**`throw`**
: Calls a custom `throw(spawner)` callback on the PooledSpawner.

Function Reference

`PooledSpawner.`**`define`**(*name*, *message_type*, *pool_count*, *overflow_behavior*)

Arguments

- **name** (*string*) – The name of the PooledSpawner prototype.
- **message** (*prototype*) – A Message prototype that should be tracked by the PooledSpawner.
- **pool_count** (*number*) – The number of entities that should be generated and placed in the pool at initialization time.
- **overflow_behavior** (*function*) – *Optional.* A function that is called when there are no more inactive entities available in the pool, and an entity is requested to be activated. The function should take the PooledSpawner as an argument and return an entity, or nil if a new entity should not be activated. If no argument is passed, defaults to `PooledSpawner.OverflowBehavior.fallible`

Defines a PooledSpawner that will track the given Message prototype.

Example:

```
local PooledSpawner = require('encompass').PooledSpawner
local PhysicsParticleSpawner = PooledSpawner.define(
    'PhysicsParticleSpawner',
    PhysicsParticleSpawnMessage,
    PARTICLE_MAX,
    PooledSpawner.OverflowBehaviors.cycle
)
```

PooledSpawner:generate(*entity*)

Arguments

- **entity** (*Entity*) – A new instance of PooledEntity.

This callback function runs when the PooledSpawner is added to the World at initialization time. It runs *n* times, where *n* is the given pool count.

The programmer must override this function or an error will be thrown.

Example:

```
function ParticleSpawner:generate(particle_entity)
    particle_entity:add_component(VelocityComponent, { x = 10, y = 10 })
end
```

If the pool count were 50, the ParticleSpawner would create 50 of these entities and add them to the inactive pool at initialization time.

PooledSpawner:spawn(*entity*, *message*)

Arguments

- **entity** (*Entity*) – A reference to the entity that is to be activated.
- **message** (*Message*) – A reference to a message that has been tracked by the PooledSpawner.

This callback is triggered when a Message of the specified prototype is produced by a Detecter. The programmer must override this function or an error will be thrown.

Example:

```
function PhysicsParticleSpawner:spawn(particle_entity, physics_particle_spawn_message)
    local position = physics_particle_spawn_message.position
    local velocity = physics_particle_spawn_message.velocity
```

(continues on next page)

(continued from previous page)

```
    local transform_component = particle_entity:get_component(TransformComponent)
    local velocity_component = particle_entity:get_component(VelocityComponent)
    local collision_component = particle_entity:get_component(PhysicsComponent)
    local body = collision_component.body

    transform_component.position = position
    velocity_component.linear = velocity
    collision_component.x_position = position.x
    collision_component.y_position = position.y
    collision_component.linear_velocity = velocity:clone()
    body:setX(position.x)
    body:setY(position.y)
    body:setLinearVelocity(velocity:unpack())

    if particle_entity:has_component(CreateMessageTimerComponent) then
        particle_entity:get_component(CreateMessageTimerComponent).time = 10
    else
        particle_entity:add_component(CreateMessageTimerComponent, {
            time = 10,
            message_to_create = DeactivateMessage,
            message_args = { entity = particle_entity }
        })
    end
end
```

2.5.4 ComponentModifier

```
local ComponentModifier = require('encompass').ComponentModifier
```

A ComponentModifier is an Engine which is responsible for consuming *ComponentMessages* and modifying the referenced *Component* in response.

ComponentModifiers are defined by the Message prototype they track and the `modify` function they implement.

If a ComponentModifier needs to receive multiple messages that modify the same component in a single frame, it should be a *MultiMessageComponentModifier* instead.

It is an anti-pattern to read Entities or Components inside a ComponentModifier. *Do not do this.*

Function Reference

`ComponentModifier.`**`define`**(*name*, *message_type*, *state_message_types*)

Arguments

- **`name`** (*`string`*) – The name of the ComponentModifier.
- **`message_type`** (*`prototype`*) – A Message prototype that should be tracked by the ComponentModifier.
- **`state_message_types`** (*`table`*) – An array-style table of *StateMessage* types that should be tracked by the ComponentModifier. *Optional*

Defines a ComponentModifier that will track the given Message prototype, and optionally track the given StateMessage types.

ComponentModifier:get_state_message(*state_message_type*)

Arguments

- **state_message_type** (*StateMessage*) – A *StateMessage* prototype that has been tracked by the ComponentModifier.

Returns An instantiated StateMessage of the given prototype, or nil if none has been produced this frame.

ComponentModifier:modify(*component*, *frozen_fields*, *message*, *dt*)

Arguments

- **component** (*Component*) – A reference to an instantiated component referenced by the tracked Message.
- **frozen_fields** (*table*) – A copied table of fields on the referenced component, so that the Modifier can read component data without it being affected by other Modifiers.
- **message** (*Message*) – A reference to a message that has been tracked by the Component-Modifier.
- **dt** – The delta time given by the World's current frame update.

This callback is triggered when a Message of the specified prototype is produced by a Detecter. The programmer must override this callback or an error will be thrown.

Example

```
local SoundMessage = require('game.messages.sound')
local TimeDilationStateMessage = require('game.messages.state.time_dilation')

local ComponentModifier = require('encompass').ComponentModifier
local SoundModifier = ComponentModifier.define(
    'SoundModifier',
    SoundMessage,
    { TimeDilationStateMessage }
)

function SoundModifier:modify(_, frozen_fields, message, dt)
    local time_dilation_state_message = self:get_state_
↪message(TimeDilationStateMessage)

    if time_dilation_state_message ~= nil then
        local source = frozen_fields.source
        source:setPitch(time_dilation_state_message.factor)
    end
end

return SoundModifier
```

2.5.5 MultiMessageComponentModifier

```
local MultiMessageComponentModifier = require('encompass').
↪MultiMessageComponentModifier
```

A MultiMessageComponentModifier is a special kind of *ComponentModifier*. It conveniently aggregates multiple ComponentMessages that are intended to modify the same component.

It is fundamentally the same as a ComponentModifier, with a slightly different `modify` callback.

Function Reference

MultiMessageComponentModifier:modify(*component*, *frozen_fields*, *messages*, *dt*)

Arguments

- **component** (*Component*) – A reference to an instantiated component referenced by the tracked Message.
- **frozen_fields** (*table*) – A copied table of fields on the referenced component, so that the Modifier can read component data without it being affected by other Modifiers.
- **messages** (*table*) – An array-style table containing references to multiple messages that have been tracked by the ComponentModifier and all reference the same component.
- **dt** – The delta time given by the World's current frame update.

This callback is triggered when one or more ComponentMessages of the specified prototype are produced. The programmer must override this callback or an error will be thrown.

Example

```
local Vector = require('hump.vector')
local forward_identity_vector = Vector(0, 1)

local MotionMessage = require('game.messages.motion')
local TimeDilationStateMessage = require('game.messages.state.time_dilation')

local MultiMessageComponentModifier = require('compass').MultiMessageComponentModifier
local MotionModifier = MultiMessageComponentModifier.define(
    'MotionModifier',
    MotionMessage,
    { TimeDilationStateMessage }
)

function MotionModifier:modify(transform_component, frozen_fields, messages, dt)
    local new_x = frozen_fields.position.x
    local new_y = frozen_fields.position.y
    local new_r = frozen_fields.rotation

    local time_dilation_state_message = self:get_state_
↪message(TimeDilationStateMessage)
    dt = dt * (time_dilation_state_message ~= nil and time_dilation_state_message.
↪factor or 1)

    for _, message in pairs(messages) do
        local instant_linear_or_dt = message.instant_linear and 1 or dt
        local instant_angular_or_dt = message.instant_angular and 1 or dt
        new_x = new_x + message.x_velocity * instant_linear_or_dt
        new_y = new_y + message.y_velocity * instant_linear_or_dt
        new_r = new_r + message.angular_velocity * instant_angular_or_dt
    end
```

(continues on next page)

(continued from previous page)

```
    if frozen_fields.screen_wrap then
        new_x = new_x % love.graphics.getWidth()
        new_y = new_y % love.graphics.getHeight()
    end

    transform_component.position.x = new_x
    transform_component.position.y = new_y
    transform_component.rotation = new_r
    transform_component.forward = forward_identity_vector:rotated(new_r)
end

return MotionModifier
```

2.5.6 EntityModifier

```
local EntityModifier = require('encompass').EntityModifier
```

An EntityModifier is an Engine which is responsible for consuming *EntityMessages* and manipulating *Entities* in response, for example, adding a component to an entity or deactivating an entity.

EntityModifiers are defined by the EntityMessage prototype they track and the `modify` function they implement.

It is an anti-pattern to read Entities or Components inside an EntityModifier. *Do not do this.*

Function Reference

`EntityModifier.`**`define`**(*name*, *message_type*, *state_message_types*)

Arguments

- **`name`** (*`string`*) – The name of the EntityModifier.
- **`message_type`** (*`prototype`*) – An EntityMessage prototype that should be tracked by the EntityModifier.
- **`state_message_types`** (*`table`*) – An array-style table of StateMessage types that should be tracked by the EntityModifier. *Optional*

Defines an EntityModifier that will track the given EntityMessage prototype, and optionally track the given StateMessage types.

`EntityModifier:get_state_message`(*state_message_type*)

Arguments

- **`state_message_type`** (*`StateMessage`*) – A StateMessage prototype that has been tracked by the EntityModifier.

Returns `StateMessage` or `nil` An instantiated StateMessage of the given typrototypepe, or nil if none has been produced this frame.

`EntityModifier:modify`(*entity*, *messages*, *dt*)

Arguments

- **`entity`** (*`Entity`*) – A reference to an entity referenced by the tracked Message.

- **messages** (*table*) – An array-style table containing references to multiple messages that have been tracked by the EntityModifier and all reference the same component.
- **dt** – The delta time given by the World's current frame update.

This callback is triggered when one or more EntityMessages of the specified prototype is produced by a Detecter. The programmer must override this callback or an error will be thrown.

Example

```
local AddComponentMessage = require('game.messages.add_component')

local EntityModifier = require('encompass').EntityModifier
local AddComponentModifier = EntityModifier.define('AddComponentModifier',␣
↪AddComponentMessage)

function AddComponentModifier:modify(entity, messages, dt)
    for _, message in pairs(messages) do
        entity:add_component(message.component_to_add, message.component_args)
    end
end

return AddComponentModifier
```

2.5.7 Renderer

```
local Renderer = require('encompass').Renderer
```

A Renderer is an Engine which is responsible for reading components to draw elements to the screen.

Renderers are defined by the *Component* prototype(s) they track, of which *exactly one* must be a *DrawComponent*, and the `render` function they implement.

It is an anti-pattern to modify Entities or Components from inside a Renderer. *Do not do this.*

Function Reference

`Renderer.`**`define`**(*name*, *component_types*)

Arguments

- **name** (*string*) – The name of the Renderer
- **component_types** (*table*) – An array-style table containing Components, *exactly one* of which must be a DrawComponent. If this is not the case, an error will be thrown.

Defines a Renderer that will track Entities which contain *all* of the given component types.

`Renderer:initialize`(...)

Arguments

- **...** (*args*) – An arbitrary list of arguments.

This callback is triggered when the Renderer is added to the World.

Useful for dealing with side effects, like initializing framebuffers.

`Renderer:render`(*entity*, *canvas*)

Arguments

- **entity** (*Entity*) – A reference to an entity which is tracked by the Renderer
- **canvas** (*Canvas*) – A reference to the current canvas (framebuffer)

Every frame, this callback runs for each entity tracked by the Renderer. The programmer must override this callback or an error will be thrown.

2.5.8 SideEffecter

```
local SideEffecter = require('encompass').SideEffecter
```

A SideEffecter is an Engine which is responsible for consuming *SideEffectMessages* and producing side effects in response. For example, if your game has multiple top-level states with separate Worlds, changing the top-level state would be a side effect.

Side effects should be used as an absolute last resort. Use them only if there is no way to produce the behavior with other types of Engines.

Using a SideEffecter to modify Entities or Components is an anti-pattern. *Do not do this.*

Function Reference

SideEffecter.**define**(*name*, *message_type*)

Arguments

- **name** (*string*) – The name of the SideEffecter.
- **message_type** (*prototype*) – An SideEffecter prototype that should be tracked by the SideEffecter.

SideEffecter:effect(*message*)

Arguments

- **message** (*SideEffectMessage*) – A reference to a SideEffectMessage that has been tracked by the SideEffecter.

This callback is triggered when a SideEffectMessage of the specified prototype is produced. The programmer must override this function or an error will be thrown.

2.6 Field Type Reference

The following is a list of valid types which can be specified for *Components* and *Messages*.

- 'boolean'
- 'number'
- 'string'
- 'userdata'
- 'table'
- Component
- Entity

- any user-defined `Component` prototype

CHAPTER 3

About Hyper ECS

What is ECS?

ECS stands for Entity-Component-System. It is a programming architectural pattern that is very useful for game development.

Components are the most basic element. They can be thought of as containers of related data. In a 2D game, we could have a Position Component with an x and y value, for example. Components typically do not contain any logic, though they may have callbacks to deal with side effects.

An **Entity** is simply a generic object, which has a unique ID and a collection of Components. Entities have no inherent behavior or properties of their own, but are granted these by Components. We can add or remove Components from Entities as necessary during the simulation.

Finally, we have **Systems**. Systems iterate over Entities that have one or more designated Components, and perform transformations on them. For example, a Motion System could look at Entities that have a Position Component and a Velocity Component, and update the Position Component based on the Velocity Component.

Why not just use object-oriented programming?

Object orientation is an intuitive idea when it comes to building simulation-oriented applications, like video games. We can think of each "thing" in the game as a self-contained object that can be acted upon externally via access to methods. For example, in an Asteroids game, the ship is an object, the bullets the ship fires are objects, the asteroids are objects, and so on.

However, problems quickly emerge when we use object-oriented programming in practice.

As programmers we want to reuse code as much as possible. Every bit of duplicated code is an opportunity for bugs to be introduced. Object-oriented code accomplishes reuse through inheritance. But this immediately runs into problems. Suppose I have an object where it would make sense to inherit behavior from two different classes. This is common in games, because it is natural to mix-and-match behaviors. But if those classes both implement a method with the same name, now our new object has no idea what to do. Most object-oriented systems, in fact, forbid multiple inheritance, so we have to share code with helper functions or other awkward constructions. **ECS** accomplishes code reuse via *composition*, not inheritance, which avoids these problems.

Another pattern that often occurs with object-oriented code is tight coupling. Objects that reference each other directly become a problem when we change those objects in any way. If we modify the behavior of object B, and object A

references object B, then we end up also having to modify object A. In a particularly poorly designed system, we might end up having to modify a dozen objects just to slightly change the behavior of one simple object.

Tight coupling is a nightmare when it comes to programming games, because games are by nature very complex simulations. The more coupling we have between objects, the more of the entire environment of the game we have to understand before we can pick apart the behavior of the game. It is also possible that we could surprise ourselves by unexpectedly changing the behavior of separate objects by changing the behavior of one particular object.

We want our architecture to encourage us to have as little coupling between different elements of the game logic as possible, so that we can look at any individual element of the game and easily understand how it behaves without needing to know anything about the other elements of the game, and modify the behavior without worrying about introducing strange behavior in other supposedly unrelated objects. ECS helps us accomplish this.

Object-orientation also makes it difficult for us to automatically test our behavior, because it encourages us to encapsulate lots of different behavior in single objects that vary based on their internal state. ECS makes it very simple to to do automated testing, because the responsibilities of each element are cleanly separated.

Issues with standard ECS

Imagine a situation where multiple systems need to manipulate the position of objects. In normal ECS, all kinds of problems could be introduced by the execution order of Systems that manipulate position. For example, let's say we have a special ability that lets the player character teleport forward over a short distance.

In regular ECS, we might have a Teleport System that sets the x and y value of the Position Component so that it is in front of the player. But if the Teleport System runs before our regular Motion System, the Position Component will be overwritten by the regular Motion System and the teleportation behavior won't occur.

This is known as a race condition, and it can be responsible for some very tricky bugs. This illustrates another issue with ECS, which is that in Systems that have similar behavior, we don't really have a convenient way to share that behavior.

A new approach

Hyper ECS is a new pattern that eliminates these common problems with ECS. The core of the architecture is a 2-pass pattern for Systems. We now have multiple kinds of Systems, and a new construct called a **Message**. Entities and Components remain the same. I will now elaborate on the different kinds of Systems.

Detectors are responsible for reading the game world and producing Messages. A **Message** is fundamentally a kind of Component, but it is designed to be temporary and is discarded at the end of each frame.

As an example, let's say we have Transform Components, which contains position and orientation data, and Velocity Components, which have an x and y component for linear motion. The **Motion Detecter** would simply read each Entity that has a Transform Component and a Velocity Component, and create a **Motion Message**, which contains a reference to the specific Transform Component, and the x and y velocity given by the Velocity Component.

In the first pass of each frame update, each Detecter runs in an arbitrary order. It does not matter which order the Detectors run in, because none of them are directly manipulating the state of the game, which would affect the other Detectors.

In the second pass, we have other types of Systems which read Messages and mutate the game world in response.

Modifiers consume a single Message prototype and manipulate game state in response. **Component Modifiers**, are, unsurprisingly, responsible for modifying Components. Continuing our above example, a **Motion Modifier** would see the Motion Message generated by the Motion Detecter, and update the data in the given Transform Component accordingly.

We also have **Entity Modifiers**. These are responsible for adding or removing components, or destroying Entities. In a similar vein, we have **Spawners**, which are responsible for creating new Entities from a given Message.

Finally there are **Renderers**. Renderers function similarly to Detectors, in that they iterate over Entities that contain a particular collection of components and then respond by drawing things.

At the end of the frame, all Messages are destroyed.

Components have strictly zero logic. Only Systems do, and each System should have precisely one responsibility. There should only be one Component Modifier per Component prototype. If we have two types of Component Modifiers for the same Component prototype, we introduce the possibility of race conditions. But we can easily avoid this because of the 2-pass structure.

More decoupling

Take our above teleportation example. We had a problem where two Systems were manipulating the same data, and the result could change based on the order of that manipulation.

In Hyper ECS, this is not a problem, since we can simply send multiple messages to the same Modifier from different Detectors. Instead of a generic Teleport System that manipulates a Position Component alongside the regular Motion System, in Hyper ECS we would know that we do not want multiple systems manipulating the same kinds of components, so we would have a Teleport Detecter create a Motion Message with the relative motion, and the Motion Modifier would process this message in the same way as it processes Motion Messages from the Motion System. This pattern allows us to easily reuse code for similar actions and avoid bugs.

encompass, a Hyper ECS implementation

You're probably here because you are curious about using Hyper ECS in practice. These ideas would work in any language or game engine, but as a reference I have implemented a fully-featured Hyper ECS framework in Lua that I call **encompass**. It has many convenient features, like a built-in object pooling system and layer-based rendering.

encompass will work with any engine that has a scripting layer in Lua, but I have built a small reference game that runs on the LÖVE framework. I believe it should illustrate the kinds of patterns that you can use to build games using encompass.

https://gitlab.com/ehemsley/hyperspace-redux

If you are used to programming games in an object-oriented way, you will likely find the ECS pattern confusing at first. But once you learn to think in an ECS way, you will be shocked at how flexible and simple the pattern really is, and how quickly it lets you introduce new features. Give it a try!

CHAPTER 4

Acknowledgements

Special thanks to Mark Kollasch <https://mastodon.technology/@fool> for his insight into the 2-pass pattern and for excellent architecture feedback.

Thanks also to the creators and maintainers of LÖVE, a wonderful tool which awakened me to the joys of game programming.

Index