

Patrones de Diseño

Entity-Component-System (Parte II)

TPV2
Samir Genaim



Lecture Notes
Samir Genaim
TPV2 2021/22
Facultad de Informática
Universidad Complutense de Madrid

De GameObject a Entity-Component

- ◆ Empezamos usando GameObject(s)
 - Los objetos de juego tenían todo: handleInput, update, render, datos, ...
 - Cambios de comportamientos mediante herencia.
 - La jerarquía de clases se puede complicar mucho si tenemos muchos cambios.
- ◆ Pasamos al primer diseño de componentes
 - Sacamos lo comportamientos fuera (encapsulación de comportamientos)
 - Introducimos la clase Container (hereda de la clase GameObject) que es una colección de componentes que definen la semántica del objeto de juego.
 - No era claro donde van los datos ...

De GameObject a Entity-Component

- ◆ Pasamos a un diseño basado en Entity-Component
 - Los componentes pueden llevar tanto datos como comportamientos (como render, update, etc)
 - Una entidad es una colección de componentes, no lleva nada de datos (salvo información útil para todas las entidades – p.ej., el boolean que indica si está viva, referencia al Manager, etc.)
 - La clase Entity permite a un componente acceder a otros componentes de la misma entidad.
 - El diseño de la clase Manager permite acceder a las entidades, definir grupos de entidades, definir entidades de interés, etc.

La lógica del juego ...

El bucle principal en nuestro diseño de Component-Entity parece al siguiente ...

```
while (!exit){  
    ...  
    manager_->update();  
    manager_->render();  
    manager_->refresh();  
}
```

Es un diseño muy flexible, que nos permite cambiar el comportamiento de entidades fácilmente, podemos añadir información sin herencia, etc.

La lógica del juego ...

Además los grupos nos permiten tener más control sobre el orden en el que manejamos las entidades ...

```
while (!exit) {  
    ...  
    manager_->update();  
    manager_->refresh();  
  
    // dibujar la estrellas  
    for (auto e : mngr_->getEntities(ecs::__grp_STARS))  
        e->render();  
  
    // y después dibujar los fantasmas  
    for (auto e : mngr_->getEntities(ecs::__grp_GHOSTS))  
        e->render();  
  
    ...  
}
```

Problemas con el diseño actual

- ◆ Mucho del control/lógica está en los componentes, para entender la lógica del juego necesitamos entender los comportamientos de muchos componentes y saber en que entidades están.
- ◆ Muchas veces no sabemos si poner un comportamiento como componente o no (p.ej., el código para comprobar colisiones).
- ◆ En juegos con muchos “actores”, necesitamos mas control sobre sobre la lógica (o el fuljo de control) del juego. Necesitamos hacer operaciones sobre conjuntos de entidades y el orden puede ser esencial.
- ◆ En algunos juegos, hay muchas entidades similares, y sólo parte de la información es distinta. Hay copias innecesarias de los componentes y necesitamos alguna arquitectura que nos permite eliminar esa duplicación.

Más control en el bucle principal

Podemos tener más control organizando el bucle principal de otra manera

```
while (!exit) {  
    ...  
    // mover el pacman  
    // mover los fantasmas  
    // comprobar colisiones y modificar marcador  
    // dibujar las frutas  
    // dibujar fantasmas  
    ...  
}
```

Estas partes de esta lógica las vamos a llamar sistemas

El orden es importante, p.ej., hay que mover todos los fantasmas antes de comprobar colisiones o antes de mover otro tipo de objetos de juego, etc.

Más Control Usando Grupos, etc

En la arquitectura actual, podemos hacer operaciones sobre conjuntos de objetos usando por ejemplo grupos, en lugar de llamar al update del manager (todas las entidades), llamamos **directamente** al update de los miembros de un grupo

```
// mover los fantasmas
for (Entity *e : mngr_->getEntities(ecs::__grp_GHOST)) {
    if ( e->hasGroup<Ghost>() ) {
        e->update();
    }
}
```

Para cada entidad de este grupo, estamos llamando a update para actualizar el estado ...

Exceso de llamadas virtuales

```
// mover los fantasmas
for (Entity *e : mngr_->getEnteties(ecs::_grp_GHOST)) {
    if ( e->hasGroup<Ghost>() ) {
        e->update();
    }
}
```



Llama al update de todos los componentes (llamada virtual)

- ◆ Llamadas a métodos virtuales cuestan más (en términos de tiempo de ejecución) porque es un salto indirecto.
- ◆ Evitan que el compilador haga optimizaciones (como inlining) porque no se sabe a qué método está llamando durante la compilación.
- ◆ Si el objetivo es mejorar el tiempo de ejecución, es mejor minimizar el uso de métodos virtuales.

Evitar Exceso de llamadas virtuales

Hay muchas técnicas que nos permiten evitar el exceso de llamadas virtual, por ejemplo usando templates, pero una forma muy sencilla es cambiar el diseño para modificar las entidades directamente, sin usar 'update'

```
// mover los fantasmas
for (Entity *e : mngr_->getEnteties(ecs::__grp_GHOSTS)) {
    if ( e->hasGroup<Ghost>() ) {
        auto t = e->getComponent<Transform>();
        // - modificar el transform t directamente (o mediante
        //   llamada a un método suyo)
        // - se puede distinguir entre varios tipos de
        //   fantasmas según sus componentes, grupos, etc.
    }
}
```

Salvo autorización expresa, los materiales
entregados a estudiantes por cualquier medio
durante la carrera sólo se podrán utilizar
para el estudio de la asignatura correspondiente
en la Universidad o su distribución posterior
(incluidos en la publicación o divulgación en redes sociales
o similares). La persona que lo haga responderá a la normativa de protección
de datos y/o la legislación sobre propiedad intelectual y
avísanos en denuncia@contenidos.ucm.es o reportcontent@ucm.es
Si encuentras esta obra en la extensión ucm.es o en otro sitio web.

Exceso de componentes ...

Suponemos que cada fantasma lleva un componente `Image` que almacena la textura que hay que usar para dibujarlo

```
// dibujar los fantasmas
for (Entity *e : mngr_->getEnteties(ecs::_grp_GHOSTS)) {
    if ( e->hasGroup<Ghost>() ) {
        auto t = e->getComponent<Transform>();
        auto img = e->getComponent<Image>();
        SDL_Rect dest = ...
        img->tex_->render(dest)
    }
}
```

Si todos los fantasmas se dibujan igual, para que tener tantas copias de `Image`? Se puede almacenar la textura una vez en un atributo `tex_` (p.ej., de `Game`) y usarla para todos

```
{ SDL_Rect dest = ...
    tex_->render(dest)
```

ECS: S for System

- ◆ La lógica (el flojo de control en general) del juego se puede dividir en varias partes donde cada parte es responsable de alguna funcionalidad del juego.
- ◆ Estas partes las vamos a llamar sistemas (**Systems**)
- ◆ Los sistemas son responsables de los comportamientos.
- ◆ Los componentes en principio no incluyen comportamientos como 'update' y 'render', incluyen datos y al mejor otros métodos (no virtuales) para hacer operaciones específicas. Sin embargo, en nuestra implementación vamos a dejar los métodos render y update para poder usar las 2 formas .
- ◆ Las entidades siguen siendo conjuntos de componentes, los componentes añaden más datos a los entidades.
- ◆ Los sistemas pueden tratar las entidades de una manera o otra dependiendo de la información que llevan (i.e., de sus componentes, grupos, etc), pueden llamar a sus métodos, etc.

Component

```
struct Component {  
    Component() : ent_(), mngr_() {}  
    virtual ~Component() {}  
    inline void setContext(Entity* ent, Manager* mngr) {  
        ent_ = ent;  
        mngr_ = mngr;  
    }  
    ...  
    virtual void initComponent() {}  
    virtual void update() {}  
    virtual void render() {}  
protected:  
    Entity* ent_;  
    Manager* mngr_;  
};
```

La clase Component la dejamos de momento como antes, aunque en una arquitectura más “ limpia” será sin los update y render

```
struct Component {  
    virtual ~Component {}  
};
```

Usamos struct para enfatizar que estamos representando datos y no comportamientos, pero podemos usar class.

Entity

```
class Entity {  
public:  
    Entity(ecs::grpId_type gId) ...  
    Entity(const Entity&) = delete;  
    Entity& operator=(const Entity&) = delete;  
    virtual ~Entity() ...  
  
    void update() ...  
    void render() ...  
  
private:  
    friend Manager;  
  
    std::vector<Component*> currCmps_;  
    std::array<Component*, maxComponentId> cmps_;  
    bool alive_;  
    ecs::grpId_type gId_;  
};
```

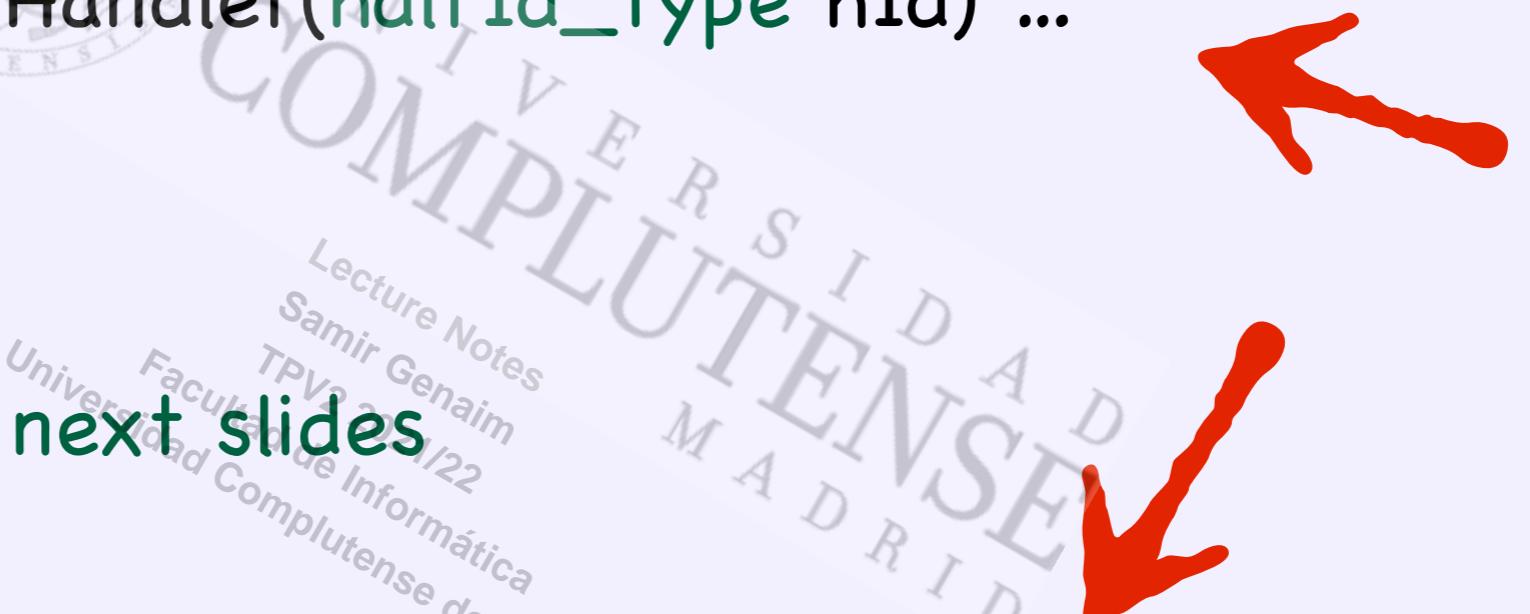
Casi como antes, pero el manejo de componentes y grupos lo vamos a mover al manager (pensando ya en el siguiente paso: gestión de memoria automático).

En una arquitectura más “limpia” quitamos update/render y la lista de componentes currCmps_

Manager

```
class Manager {  
public:  
    Manager() ...  
    virtual ~Manager() ...  
    inline Entity* addEntity(grpId_type gId); ...  
    inline const auto& getEntitiesByGroup(grpId_type gId) ...  
    inline void setHandler(hdlrId_type hId, Entity *e) ...  
    inline Entity* getHandler(hdlrId_type hId) ...  
    void update() ...  
    void render() ...  
    void refresh() ...  
  
    //new methods in next slides  
private:  
    ...  
    std::array<Entity*, maxHandlerId> hdlrs_;  
    std::array<std::vector<Entity*>, maxGroupId> entsByGroup_;  
    std::array<System*, maxSystemId> sys_;  
};
```

Muchos métodos y atributos quedan como antes



Manager - addComponent

```
template<typename T, typename ...Ts>
inline T* addComponent(Entity *e, Ts&&...args) {
    T *c = new T(std::forward<Ts>(args)...);
```

Recibe la entidad como parámetro

```
constexpr cmpId_type cId = T::id;
```

```
removeComponent<T>()
```

```
e->currCmps_.push_back(c);
```

```
e->cmps_[cId] = c;
```

```
e->setContext(e, this);
```

```
e->initComponent();
```

```
return c;
```

```
}
```

Como antes, pero ahora recibe la entidad como parámetro. Para añadir un componente necesitamos pedirlo al manager pasándole una entidad.

```
auto ball = mngr_->addEntity();
```

```
auto tr = mngr_->addComponent<Transform>(ball);
```

Manager - removeComponent

```
template<typename T>
inline void removeComponent(Entity *e) {
    constexpr cmpId_type cId = T::id;

    if (e->cmps_[cId] != nullptr) {
        auto iter = std::find(e->currCmps_.begin(),
                              e->currCmps_.end(),
                              e->cmps_[cId]);
        e->currCmps_.erase(iter);
        delete e->cmps_[cId];
        e->cmps_[cId] = nullptr;
    }
}

auto tr = mngr_->removeComponet<Transform>(e);
```

Como antes, pero ahora recibe la entidad como parámetro.

Manager - otros métodos

```
template<typename T>
inline bool hasComponent(Entity *e) {
    constexpr cmpId_type cId = T::id;
    return e->cmps_[cId] != nullptr;
}
```

```
template<typename T>
inline T* getComponent(Entity *e) {
    constexpr cmpId_type cId = T::id;
    return static_cast<T*>(e->cmps_[cId]);
}
```

```
inline void setAlive(Entity *e, bool alive) {
    e->alive_ = alive;
}
```

```
inline bool isAlive(Entity *e) {
    return e->alive_;
}
```

```
inline grpId_type groupId(Entity *e) {
    return e->gId_;
}
```

Como antes, pero ahora recibe la entidad como parámetro.

La clase System

```
class System {  
public:  
    virtual ~System() {}  
    void setContext(Manager *mngr) {  
        mngr_= mngr;  
    }  
  
    virtual void initSystem() {}  
    virtual void update() {}  
  
protected:  
    Manager *mngr_;  
};
```

Para pasarle la referencia al manager, se puede pasar en la constructora.

Para inicializar si es necesario.

A update llamamos desde el bucle principal para que el sistema haga su tarea,

Ejemplo de Sistema

Un sistema para mover (y rotar) todos los asteroids. Suponemos que los atributos de `Transform` son públicos ...

```
class AsteroidsSystem : public ecs::System {  
public:  
    virtual ~AsteroidsSystem() {}  
    virtual void update() {  
        for (auto e : mngr_->getEntities(ecs::__grp_ASTEROIDS)) {  
            auto tr = mngr_->getComponent<Transform>(e);  
            tr->pos_ = tr->pos_ + tr->vel_;  
            tr->rot_ += 5.0f;  
        }  
        ...  
    };
```

Salvo autorización expresa, los materiales entregados a los estudiantes por cualquier medio durante la carrera sólo se podrán utilizar para el estudio de la asignatura correspondiente en la Universidad Complutense de Madrid. La publicación o distribución en Internet y sus servicios de comunicación social, incluyendo la divulgación en las páginas web que la persona infractora, o sucesivamente, genere, o que no tenga la extensión ucm.es, puede vulnerar la legislación de protección de datos y de propiedad intelectual y puede ser responsabilidad de la persona infractora. Si encuentra este material en otro sitio web o portada de la Universidad Complutense de Madrid, o denuncia contenido@ucm.es

Crear, inicializar, usar ...

```
void Game::init() {
```

```
...
```

```
asteroidsSys_ = new AsteroidsSystem(...);
```

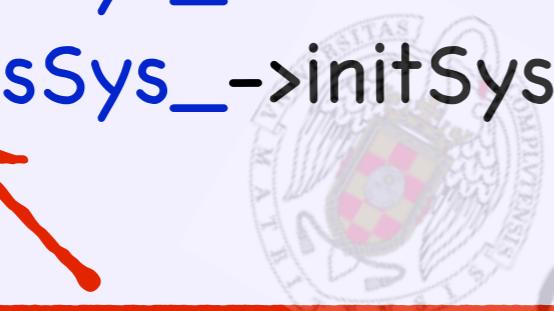
```
asteroidsSys_->setManger(mngr_);
```

```
asteroidsSys_->initSystem();
```

```
...
```

```
}
```

Atributo en la clase
Game



Crear y inicializar
un sistema

```
while (!exit) {
```

```
...
```

```
asteroidsSys_->update();
```

```
...
```

```
}
```

Usar en el bucle principal,
por ejemplo ...

Lecture Notes
Samir Genaim
TPY2 2021/22
Facultad de Informática
Universidad Complutense de Madrid

Communication entre Sistemas

Normalmente un sistema necesita comunicar con otros sistemas. La solución más sencilla es pasar un sistema a otro (en la constructora por ejemplo) y usar sus métodos directamente ...

```
void Game::init() {  
    ...  
    asteroidsSys_ = new AsteroidsSystem(...);  
    asteroidsSys_->setContext(mngr_);  
    asteroidsSys_->initSystem();  
    ...  
    collisionsSys_ = new CollisionSystem(asteroidsSys_);  
    collisionsSys_->setContext(mngr_);   
    collisionsSys_->initSystem();  
    ...  
}
```

Necesita avisar a AsteroidsSystem
cuando haya una colisión

Accesso directo usando el Manager

Otra posibilidad es modificar el manager para tener acceso directo a todos los sistemas. La comunicación sigue siendo a través de llamadas a métodos correspondientes

```
void Game::init() {  
    ...  
    mngr_ = new Manager();  
    ...  
    asteroidsSys_ = mngr_->addSystem<AstroidsSystem>(...);  
    ...  
}
```

Añadir un sistema al manager ...

```
auto asteroidsSys = mngr_->getSystem<AstroidsSystem>();  
asteroidsSys->onCollision(e)  
...
```



Acceder a un sistema desde otro (cualquiera que tenga una referencia al manager puede acceder).

Cada Sistema Lleva su Identificador

```
class AsteroidsSystem : public ecs::System {  
public:  
    constexpr static ecs:: sysId_type id = ecs::_sys_ASTEROIDS;  
    virtual ~AsteroidsSystem() {}  
    virtual void update() {  
        for (auto e : mngr_->getEntities(ecs::_grp_ASTEROIDS)) {  
            auto tr = e->getComponent<Transform>();  
            tr->pos_ = tr->pos_ + tr->vel_;  
            tr->rot_ += 5.0f;  
        }  
    }  
    ...  
};  
  
using sysId_type = uint8_t;  
enum sysId : sysId_type {  
    _sys_ASTEROIDS = 0,  
    _sys_COLLISIONS,  
    // do not remove this  
    _LAST_SYS_ID  
};  
constexpr sysId_type maxSystemId = _LAST_SYS_ID;
```

En ecs.h

Manager - addSystem

```
class Manager {  
public:
```

```
...
```

```
template<typename T, typename ...Ts>  
inline T* addSystem(Ts &&... args) {  
    constexpr sysId_type sId = T::id;  
    removeSystem<T>();  
    System *s = new T(std::forward<Ts>(args)...);  
    s->setContext(this);  
    s->initSystem();  
    sys_[sId] = s;  
    return static_cast<T*>(s);
```

```
}
```

```
private:
```

```
...
```

```
std::array<System*, ecs::maxSystemId> sys_;
```

```
}
```

El identificador
del sistema

Borrar el actual
si lo hay

Crear, inicializar y añadir
al array de sistemas

El array de
sistemas

... = mngr_->addSystem<AsteroidsSystem>(...);

Manager - getSystem, removeSystem

```
class Manager {  
public:
```

```
...  
template<typename T>  
inline T* getSystem() {  
    constexpr sysId_type sId = T::id;  
    return static_cast<T*>(sys_[sId]);  
}
```

Consultar un sistema

```
template<typename T>  
inline void removeSystem() {  
    constexpr sysId_type sId = T::id;  
    if (sys_[sId] != nullptr) {  
        delete sys_[sId];  
        sys_[sId] = nullptr;  
    }  
}  
...  
}
```

Quitar un sistema

Salvo autorización expresa, los materiales entregados a estudiantes por cualquier medio durante la carrera se podrán utilizar para el estudio de asignaturas correspondientes. La publicación o distribución posterior (incluida la divulgación en redes sociales o servicios de comunicación en Internet) puede vulnerar la normativa de protección de datos y/o la de propiedad intelectual y general que responsabiliza a la persona infractora. avisando en la extensión ucm.es, reportcontent@ucm.es

```
[mngr_->getSystem<AsteroidsSystem>()->onCollision(e);  
[mngr_->removeSystem<CollisionsSystem>();
```

El Problema con la Comunicación Directa

```
auto pm = mngr_->getHandler(ecs::_hdlr_PACMAN);
auto pTR = mngr_->getComponent<Transform>(pm);
auto &stars = mngr_->getEntities(ecs::_grp_STARS);
auto n = stars.size();

for (auto i = 0u; i < n; i++) {
    auto e = stars[i];
    if (mngr_->isAlive(e)) {
        auto eTR = mngr_->getComponent<Transform>(e);
        if (Collisions::collides(pTR->pos_, ..., eTR->pos_, ...)) {
            mngr_->getSystem<AsteroidsSystem>()->onCollision(e);
        }
    }
}
```

Un sistema tiene que saber mucho del los otros para poder comunicar. P.ej., que tiene método `onCollision` que gestiona la `collision`

Si a otros sistemas les interesa saber cuando haya una `collision`, tenemos que avisar (añadir llamadas) a todos

```
mngr_->getSystem<GameCtrlSystem>()->onCollision(e);
```

Comunicación Usando Mensajes

```
for (auto i = 0; i < n; i++) {  
    auto e = stars[i];  
    if (mngr_->isAlive(e)) {  
        auto eTR = mngr_->getComponent<Transform>(e);  
        if (Collisions::collides(pTR->pos_, ..., eTR->pos_, ...)) {  
            Message m;  
            m.id = _m_STAR_EATEN;  
            m.star_eaten_data.e = e;  
            mngr_->send(m);  
        }  
    }  
}
```

Salvo autorización expresa, los materiales entregados a estudiantes por cualquier medio para el estudio de la asignatura correspondiente en la Universidad Complutense de Madrid, en su publicación o distribución posterior (incluida la divulgación en redes sociales o servicios de comunicación en Internet) o servicios de comunicación en Internet, puede vulnerar la propiedad intelectual o derechos de autor de la persona infractora. Se responsabiliza de la divulgación de datos que se encuentren en este material en otro sitio web (así como de la divulgación de información de protección de datos). La denuncia de contenido ucm.es, generalmente se hace a través de la dirección de correo electrónico reportcontent@ucm.es.

Cuando ocurre algún evento, como la collision, creamos un Mensaje (estructura de datos) y lo enviamos a todos los sistemas (p.ej., a través del manager). Todos lo reciben, y los interesados reaccionan y los otros lo ignoran.

Una Estructura de Datos para Mensajes

```
using msgId_type = uint8_t;  
enum msgId : msgId_type {  
    _m_STAR_EATEN, //  
    _m_ADD_STARS  
};
```

Identificador que indica lo que ha ocurrido, representa el tipo de mensaje, depende de este valor un sistema pueden reaccionar de una manera o otra.

```
struct Message {  
    msgId_type id; // _m_STAR_EATEN  
    struct {  
        ecs::Entity *e;  
    } star_eaten_data;  
  
    struct {  
        unsigned int n;  
    } add_stars_data;  
};
```

Para cada tipo de mensaje, si hace falta pasar información adicional, añadimos otro atributo (como struct) que lleva toda esa información ...

De momento el mensaje lleva la información de todos los tipos de mensajes aunque no se usan. Vamos a ver cómo reducir el uso de memoria en este caso usando union types

Recibir un Mensaje

```
class System {  
public:  
    virtual ~System() { }  
    void setContext(Manager *mngr) {  
        mngr_= mngr;  
    }  
  
    virtual void initSystem() { }  
    virtual void update() { }  
    virtual void receive(const Message& m) { }  
  
protected:  
    Manager *mngr_;  
};
```

Añadimos un método a la clase System para poder recibir un mensaje ...

Enviar un Mensaje

```
class Manager {  
public:
```

Un métodos del manager que recibe un mensaje y lo envía a todos los sistemas

...

```
inline void send(const Message &m) {
```

```
    for (System *s : sys_) {
```

```
        if (s != nullptr)
```

```
            s->recieve(m);
```

```
}
```

...

```
};  
...  
if (Collisions::collides(pTR->pos_, ..., eTR->pos_, ...)) {
```

```
    Message m;
```

```
    m.id = _m_STAR_EATEN;
```

```
    m.star_eaten_data.e = e;
```

```
    mngr_->send(m);
```

```
}
```

...



El envío del mensaje que avisa que ha habido un collision

Reaccionar a Mensajes

```
void StarsSystem::receive(const Message &m) {  
    switch (m.id) {  
        case _m_STAR_EATEN:  
            onStarEaten(m.star_eaten_data.e); ←  
            break; ←  
        case _m_CREATE_STARS:  
            addStar(m.create_stars_data.n); ←  
            break; ←  
        default:  
            break;  
    } ←  
}
```

Depende del tipo de mensaje, el sistema reacciona de una manera o otra.

```
void GameCtrlSystem::receive(const Message &m) {  
    switch (m.id) {  
        case _m_STAR_EATEN:  
            score_ += 1; ←  
            break; ←  
        default:  
            break;  
    } ←  
}
```

Accede a los datos adicionales depende del tipo de mensaje

Cada sistema reacciona sólo a los mensajes que le interesan

Enviar a un solo Sistema

```
if (Collisions::collides(pTR->pos_, ..., eTR->pos_, ...)) {  
    Message m;  
    m.id = _m_STAR_EATEN;  
    m.star_eaten_data.e = e;  
    mngr_->getSystem<StarsSystem>()->receive(m);  
}
```

Enviar sólo a StarsSystem

- ◆ A veces no hace falta enviar el mensaje a todos los sistemas, si sabemos que le interesa sólo a uno.
- ◆ Enviar un mensaje a un sólo sistema usando el método receive, en lugar de llamar a otro método específico como onStarEaten, tiene la ventaja que no tenemos que saber mucho sobre esa clase y sus métodos

El problema del orden de mensajes

```
class Manager {  
public:  
    ...  
    inline void send(const Message &m) {  
        for (System *s : sys_) {  
            if (s != nullptr)  
                s->recieve(m);  
        }  
    ...  
};
```

- ◆ Suponemos que hay 3 sistemas: A, B, y C (en ese orden en el array de sistemas).
- ◆ El sistema A envía un mensaje m1
- ◆ Cuando el sistema B recibe el mensaje m1, como reacción envía un mensaje m2
- ◆ En este caso, C recibe m2 y después m1, si el orden es importante esto es un problema ...

No enviar inmediatamente ...

```
class Manager {  
...  
    inline void send(const Message &m, bool delay = false) {  
        if (!delay) {  
            for (System *s : sys_) {  
                if (s != nullptr)  
                    s->receive(m);  
            }  
        } else {  
            msgs_.emplace_back(m);  
        }  
    }  
...  
private:  
...  
    std::vector<Message> msgs_;  
};
```

Al enviar un mensaje, se puede elegir entre enviarlo inmediatamente o ponerlo en la cola para enviarlo más tarde (next slide ...)

Un vector que representa una cola de mensajes

Enviar los mensajes pendientes

```
class Manager {  
...  
    inline void flushMessages() {  
        auto size = msgs_.size();  
        for (auto i = 0u; i < size; i++) {  
            auto &m = msgs_[i];  
            for (System *s : sys_) {  
                if (s != nullptr)  
                    s->receive(m);  
            }  
        }  
        msgs_.erase(msgs_.begin(), msgs_.begin() + size);  
    }  
...  
}
```

El el bucle principal, hay que llamar a flushMessages para enviar los mensajes pendientes

Recorremos al vector con un indice para no enviar mensaje que se añaden como reacción

Borra los mensajes que se han enviado, esto también desplaza los nuevos mensajes, tiene coste lineal ...

Enviar los mensajes pendientes (II)

```
class Manager {
```

```
...
```

```
    inline void flushMessages() {  
        std::swap(msgs_, aux_msgs_);  
        for (auto &m : aux_msgs_) {  
            for (System *s : sys_) {  
                if (s != nullptr)  
                    s->receive(m);  
            }  
        }  
        aux_msgs_.clear();  
    }
```

```
private:
```

```
...
```

```
    std::vector<Message> msgs_;  
    std::vector<Message> aux_msgs_;
```

```
}
```

Intercambia las 2 las colas

envía todos los mensajes de `aux_msgs_`. Los nuevos mensajes que entran como reacción van a `msgs_`

Borrar los mensajes enviados, sin la necesidad de desplazar los nuevos.

Mantener 2 colas de mensajes. El método send usa `msgs_` como antes

flushMessages usando 2 colas. Mas eficiente del anterior porque al borrar los mensajes enviados no se desplaza nada

Resumen

- ◆ Los sistemas nos permiten organizar el juego en varias partes tal que cada parte es responsable de alguna funcionalidad.
- ◆ Los comportamientos **van en los sistemas**, y los componentes llevan **sólo datos** (y al mejor métodos para gestionar **estos datos**).
- ◆ Los sistemas pueden **comunicarse** directamente o a través de mensajes.
- ◆ El uso de mensajes **es muy cómodo**, para añadir nueva funcionalidad no introducimos dependencias directas entre sistemas.
- ◆ Usar los mensajes con moderación porque tiene costa extra.

Resumen

- ◆ Hemos visto varias arquitecturas hasta el momento. Es muy importante entender que cada arquitectura tiene sus ventajas y que puede ser más adecuada que otras depende del contexto (el juego)
- ◆ Todas las arquitecturas que hemos visto son sólo ideas generales que se pueden combinar/adaptar para un juego específico ...