

# Patrones de Diseño

## Entity-Component-System (Parte II)

TPV2  
Samir Genaim

# 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 (que hereda de GameObject) — 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 activo, referencia al Manager, etc.)
  - ➡ El diseño de la clase Entity permite a componentes acceder a otros componentes de la misma entidad
  - ➡ El diseño de la clase Manager permite acceder a otras entidades, definir grupos de entidades, 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., pero ...

# Problemas con el diseño actual

- ✦ Todo el control 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 (cómo comprobar colisiones) ...
- ✦ En juegos con muchos "actores", necesitamos mas control sobre sobre la lógica (o el flujo 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

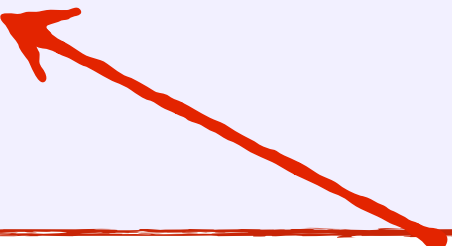
```
while (!exit) {  
    ...  
    // mover el packman  
    // mover los fantasmas  
    // comprobar colisiones y modificar marcador  
    // dibujar las frutas  
    // dibujar fantasmas  
    ...  
}
```

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 la entidades), llamamos directamente al update de los miembros de un grupo

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



Llama al update de todos los componentes (llamada virtual)

- ♦ Llamadas a métodos virtuales cuestan más (en tiempo de ejecución) porque es un salto indirecto ...
- ♦ ... más importante, también evitan que el compilador haga optimizaciones (como inlining) porque no se sabe a que 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

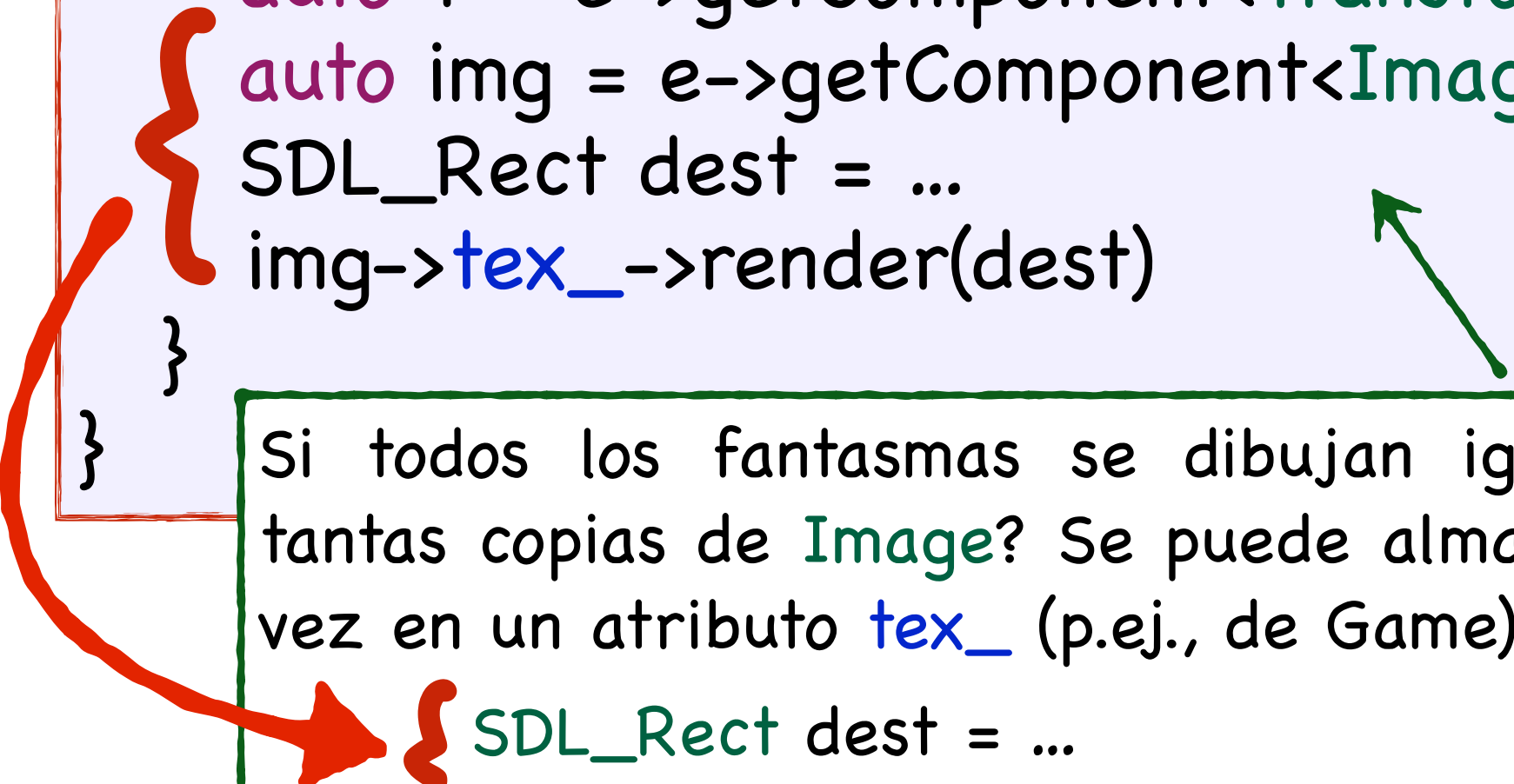
Hay muchas técnicas que nos permiten evitar el exceso de llamadas virtual, por ejemplo usando templates, pero una forma muy sencilla es cambiar el diseño para modificar las entidades directamente, sin usar 'update'

```
// mover los fantasmas
for (Entity *e : mngr_>getEntities()) {
    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.
    }
}
```

# 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_>getEntities()) {
    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 (y el flujo 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, así que los componentes no incluyen comportamientos como 'update' y 'render', incluyen sólo datos
- ♦ Las entidades siguen siendo conjuntos de componentes, los componentes añaden más datos a las entidades ...
- ♦ ... y los sistemas pueden tratar las entidades de una manera o otra dependiendo de la información que llevan (i.e., de sus componentes, grupos, etc)

# Component

```
struct Component {  
};
```

A red arrow points from the closing brace of the struct definition to the text 'Los componentes heredan del struct Component.' A green arrow points from the same text to the 'struct Component' part of the code.

Los componentes heredan del struct Component.

Usamos struct para enfatizar que estamos representando datos y no comportamientos (podemos usar class). En C++ la diferencia entre struct y class es sólo en la visibilidad por defecto (struct todo public, class todo private).

Los componentes pueden tener métodos, normalmente para hacer operaciones sobre sus atributos. Podemos también dejar una referencia a la entidad o incluir método init como antes si es necesario, todo depende del contexto

# Entity

```
struct Entity {  
    friend Manager;
```

Para que el Manager pueda acceder a los atributos privados

```
private:
```

```
    bool active_;
```

Se puede dejar public

```
    std::array<std::unique_ptr<Component>, ecs::maxComponents> cmps_ = { };
```

```
    std::bitset<ecs::maxGroup> groups_;
```

```
};
```


Es una colección de componentes. No tiene métodos update y render porque ya no hacen falta.

No tiene addComponent, GetComponent, etc., los vamos a poner en el Manager para tener más control sobre la gestión de la memoria – se pueden dejar aquí de momento, pero queremos estar preparados para la gestión de memoria que vamos a ver más adelante

# Manager

```
class Manager {  
public:  
    Manager();  
    virtual ~Manager();
```

Necesitamos acceso a las entidades porque no hay render/update



```
    inline const std::vector<Entity*>& getEntities() {  
        return entities_;  
    }
```

No hacen falta update y render, dejamos refresh de momento



```
    void refresh();  
    inline Entity* addEntity();  
    // more methods in next slides  
private:  
    std::vector<Entity*> entities_;  
    std::array<Entity*, ecs::maxHdlr> hdlrs_;  
};
```



# Métodos para Entidades

```
Entity* addEntity() {  
    Entity *e = new Entity();  
    e->active_ = true;  
    entities_.emplace_back(e);  
    return e;  
}
```

Crea la entidad y la añade a la lista de entidades

```
template<typename T>  
inline void setActive(Entity *e, bool state) {  
    e->active_ = state;  
}
```

```
template<typename T>  
inline bool isActive(Entity *e) {  
    return e->active_;  
}
```

Consultar y cambiar el estado de una entidad

# Métodos para Handlers

Los métodos de handlers son como antes ...


```
template<typename T>
inline void setHandler(Entity *e) {
    hdlrs_[ecs::hdlrIdx<T>] = e;
}
```

```
template<typename T>
inline Entity* getHandler() {
    return hdlrs_[ecs::hdlrIdx<T>];
}
```




# Métodos para Grupos

```
void resetGroups(Entity *e) {  
    e->groups_.reset();  
}
```




Son como el diseño anterior  
pero reciben la entidad como  
parámetro

```
template<typename GT>  
inline void setGroup(Entity *e, bool state) {  
    assert(e != nullptr);  
    e->groups_[ecs::grpIdx<GT>] = state;  
}
```



```
template<typename GT>  
inline bool hasGroup(Entity *e) {  
    assert(e != nullptr);  
    return e->groups_[ecs::grpIdx<GT>];  
}
```



# Método para Componentes

```
template<typename T, typename ...Ts>
inline T* addComponent(Entity *e, Ts &&...args) {
    T *c = new T(std::forward<Ts>(args)...);
    e->_cmps[ecs::cmpIdx<T>] = std::unique_ptr<Component>(c);
    return c;
}

template<typename T>
inline T* GetComponent(Entity *e) {
    return static_cast<T*>(e->_cmps[ecs::cmpIdx<T>].get());
}

template<typename T>
inline bool hasComponent(Entity *e) {
    return e->_cmps[ecs::cmpIdx<T>].get() != nullptr;
}

template<typename T>
inline void removeComponent(Entity *e) {
    e->_cmps[ecs::cmpIdx<T>] = nullptr;
}
```

Como antes, pero reciben la entidad como parámetro

# La calse System

```
class System {  
public:  
    virtual ~System() { }  
    void setMngr(Manager *manager) {  
        manager_ = manager  
    }  
    virtual void init() { }  
    virtual void update() { }  
protected:  
    Manager *manager_;  
};
```

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...



Los sistemas heredan de System sólo para tener interfaz común, se puede también definirlos sin interfaz común ...

# Crear, inicializar, usar ...

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

```
...
```

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

```
asteroidsSys_>setManger(manager_);
```

```
asteroidsSys_>init();
```

```
...
```

```
}
```

Crear y inicializar  
un sistema

Atributo en la clase  
Game

Usar en el bucle principal,  
por ejemplo ...

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

```
...
```

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

```
...
```

```
}
```

# 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_>setManger(manager_);  
    asteroidsSys_>init();  
    ...  
    collisionsSys_ = new CollisionSystem(asteroidsSys_);  
    collisionsSys_>setManger(manager_);  
    collisionsSys_>init();  
    ...  
}
```

# Acceso directo usando el Manager

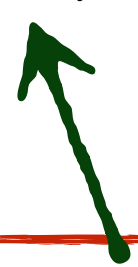
Otra posibilidad es modificar el manager para tener acceso directo a todos los sistemas. La comunicación sigue siendo a través de llamada a métodos correspondientes

```
void Game::init() {  
    ...  
    manager_ = new Manager();  
    ...  
    asteroidsSys_ = manager_>addSystem<AsteroidsSystem>(...);  
    ...  
}
```



Añadir un sistema al manager ...

```
auto asteroidsSys = manager_>getSystem<AsteroidsSystem>();  
asteroidsSys->onCollision(e)  
...
```



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



# Acceso directo usando el Manager

```
class Manager {  
public:
```

```
...  
template<typename T, typename ...Ts>  
inline T* addSystem(Ts &&...args) {  
    T *s = new T(std::forward<Ts>(args)...);  
    sys_[ecs::sysIdx<T>] = std::unique_ptr<System>(s);  
    s->setMngr(this);  
    s->init();  
    return s;  
}
```

Crear el sistema y almacenarlo en el array de sistemas, usamos unique\_ptr para que se destruya al destruir el manager

Inicializar el sistema

```
template<typename T>  
inline T* getSystem() {  
    return static_cast<T*>(sys_[ecs::sysIdx<T>].get());  
}
```

Pedir un sistema

El array de sistemas

```
private:
```

```
...  
std::array<std::unique_ptr<System>, ecs::maxSystem> sys_;  
}
```

# Acceso directo usando el Manager

En `ecs.h` añadimos la definición de `sysIdx` y `maxSystem` usando template meta-programming, como en el caso de componentes, grupos, handlers, etc.

```
class CollisionSystem;
class AsteroidSystem;
...
using SysList = TypeList<CollisionSystem, AsteroidSystem, ...>;

namespace ecs {
...
template<typename T>
constexpr std::size_t sysIdx = mpl::IndexOf<T, SysList>();
constexpr std::size_t maxSystem = SysList::size;
...
}
```



# Resumen

- ✦ Los sistemas nos permiten organizar la funcionalidad de un videojuego en varias partes, 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)
- ✦ 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 (para que sea más eficiente).