

Patrones de Diseño

Entity-Component-System
(Parte I)

TPV2
Samir Genaim



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

Objetivos

- ◆ Ya hemos dado el primer paso hacia el diseño basado en componentes
- ◆ Hemos sacado los comportamientos de un GameObject fuera usando la interfaz Component
- ◆ La clase Container es una colección de componentes que definen la semántica de la entidad del juego correspondiente
- ◆ Ahora los componentes definen sólo comportamientos, los datos van en la clase GameObject o en clases concretas que heredan de Container.

Objetivos

- ◆ Containers representan Entidades del juego (paddle, ball, etc)
- ◆ Sacar los datos fuera y ponerlos en componentes.
- ◆ Simplificar la comunicación entre componentes: un componente puede pedir a su Entidad una referencia a otro componente para acceder a sus datos, etc.
- ◆ Preguntar si una entidad tiene un componente
- ◆ Quitar un componente de una entidad
- ◆ Agrupar Entidades “similares” (balas, asteroids, etc)
- ◆ Poder añadir entidades al juego desde cualquier parte fácilmente ...
- ◆ ...

EC (sin la S de momento)

- ◆ EC = Entity-Component
- ◆ **Component**: representa datos (atributos de una clase) y/o comportamientos de una entidad – en nuestro caso **muy particular** vamos a definir un componente como una clase que tiene métodos update y render.
- ◆ **Entity**: mantiene un **array de componentes de tamaño fijo**, en la posición “i” se puede poner un sólo componente de tipo C_i , se puede consultar el componente en una posición. La entidad puede ser viva o muerta (las muertas salen de juego).
- ◆ **Manager**: una clase que nos ayuda a manejar las entidades. Crear entidades, agrupar entidades similares, borrar las entidades muertas, etc.
- ◆ Los componentes se comunican a través de su entidad: un componente puede pedir a su entidad una referencia a otro componente y acceder a sus métodos/atributos directamente.

Component

```
class Component {  
public:  
    Component() : ent_(), mngr_() {}  
    virtual ~Component() {}  
    inline void setContext(Entity* ent, Entity* mngr) {  
        ent_ = ent;  
        mngr_ = mngr;  
    }  
}
```

```
...  
virtual void initComponent() {}  
virtual void update() {}  
virtual void render() {}  
protected:  
    Entity* ent_;  
    Manager* mngr_;  
};
```

Para pasarle un puntero a su entidad y al manager

se invoca al añadir un componente a una entidad. Para inicializar el componente si es necesario

para actualizar el estado, manejar entrada, etc,

para renderizar el estado

Se pueden añadir métodos, p.ej., handleInput, o definir varios tipos de componentes (entrada, física, gráfica, datos, etc.), depende de las necesidades del juego

Entity

```
class Entity {  
public:  
    Entity();  
    virtual ~Entity();  
    inline void setContext(Manager *mngr) ...  
    inline bool isAlive() ...  
    inline void setAlive(bool alive) ...  
    ...  
private:  
    bool alive_;  
    Manager *mngr_;  
    std::vector<Component*> currCmps_;  
    std::array<Component*, ecs::maxComponentId> cmps_;  
};
```

Para pasarle el Manager, lo necesita para pasarle a sus componentes

Consultar y cambiar el estado (alive o dead).

Los componentes de la entidad - next slide

Entity

```
class Entity {  
    ...  
    std::vector<Component*> currCmps_;  
    std::array<Component*, ecs::maxComponentId> cmps_;  
};
```

- ◆ `cmps_` se usa para tener un acceso rápido a los componentes por identificador (posición).
- ◆ `currCmps_` se usa para recorrer a todos los componentes actuales de la entidad.

Header File: ecs.h

```
using cmpId_type = uint8_t;
```

```
enum cmpId : cmpId_type {
    _TRANSFORM = 0,
    _IMAGE,
    _CTRL,
    // do not remove this
    _LAST_CMP_ID
};
```

Identificadores para los components, en las entidades podemos añadir un componente para cada identificador, en principio en la posición "i" siempre ponemos un componente de tipo Ci ... volvemos a este tema más adelante

```
constexpr cmpId_type maxComponentId = _LAST_CMP_ID;
```

Entity - addComponent

```
template<typename T, typename ...Ts>
inline T* addComponent(cmpId_type cId, Ts&&...args) {
```

```
    T *c = new T(std::forward<Ts>(args)...);
```

```
removeComponent(cId)
```

```
currCmps_.push_back(c);
```

```
cmps_[cId] = c;
```

```
c->setContext(this, mngr_);
```

```
c->initComponent();
```

```
return c;
```

```
}
```

Crea el componente

Borra el componente actual en la posición cId - next slide

Añadir al array y a la lista de componentes

Inicializar el componente

Entity - ejemplo de addComponent

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

Ejecutando

```
e->addComponent<Transform>(ecs::_TRANSFORM, e1,e2,...)
```

T = Transform

Ts = los tipos de e1,e2, ...

args = parámetros correspondientes a1,a2,...

Entity - addComponent

Porqué creamos el componente en addComponent? porqué no usar algo más sencillo como el siguiente método y dejar al usuario que crea sus componentes?

```
void addComponent(cmpId_type cId, Component *c) {  
    ...  
}
```

- ◆ Así sabemos de quien es la responsabilidad de borrar el componente (y liberar la memoria dinámica)
- ◆ Tenemos más control sobre la gestión de la memoria, p.ej., en lugar de usar `new` podemos usar allocators, object pool, etc., el usuario no tiene que cambiar su código (ni saberlo!) – lo vemos mas adelante

Entity - removeComponent

```
inline void removeComponent(cmpId_type cId) {  
    if (cmps_[cId] != nullptr) {  
        auto iter = std::find(currCmps_.begin(),  
                             currCmps_.end(),  
                             cmps_[cId]);  
        currCmps_.erase(iter);  
        delete cmps_[cId];  
        cmps_[cId] = nullptr;  
    }  
}
```

Si hay un com
en la posición

Borralo de la lista
de componente

Borra el objeto correspondiente

Quítalo del array de componentes

Entity - getComponent, hasComp...

Devuelve el componente haciendo casting a T* para simplificar el uso (en lugar de hacer casting para cada llamada)

Transform *c = e->getComponent<Transform>(ecs::_TRANSFORM);

```
template<typename T>
inline T* getComponent(cmpId_type cId) {
    return static_cast<T*>(cmps_[cId]);
}

inline bool hasComponent(cmpId_type cId) {
    return cmps_[cId] != nullptr;
}
```

Simplemente comprueba si tiene un componente en la posición cId

Entity - update y render

```
inline void update() {  
    auto n = currCmps_.size();  
    for (auto i = 0u; i < n; i++)  
        currCmps_[i]->update();  
}
```

```
inline void render() {  
    auto n = currCmps_.size();  
    for (auto i = 0u; i < n; i++)  
        currCmps_[i]->render();  
}
```

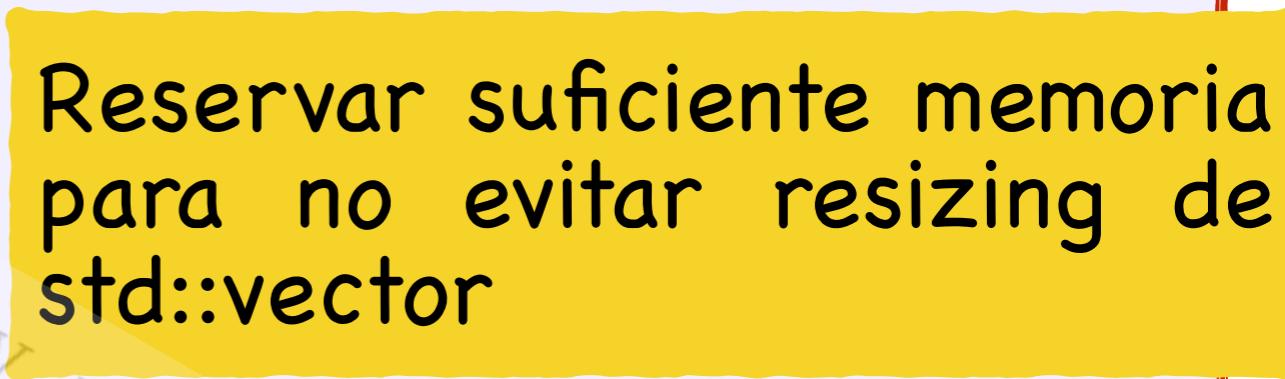
ejecuta update() y render() de todos los componentes ...

Si añadimos un componente mientras ejecutando los bucles, no se va llamar a su update/render (en realidad puede llamar a su render, pero lo ignoramos de momento!). No se puede quitar un componente mientras ejecutando los bucles, aunque técnicamente no está prohibido ...

Entity - Constructor y Destructor

```
Entity() :  
    mngr_(nullptr), cmps_(), currCmps_(), alive_() {  
    currCmps_.reserve(ecs::maxComponentId);  
}  
  
virtual ~Entity() {  
    for (auto c : currCmps_) {  
        delete c;  
    }  
}
```

 A watermark of the University of Madrid logo is visible in the background, featuring a circular emblem with a coat of arms and Latin text.

 Reservar suficiente memoria para no evitar resizing de std::vector

 Borrar todos los componentes actuales

Manager

```
class Manager {  
public:  
    Manager();  
    virtual ~Manager();  
  
    Entity* addEntity();  
    void refresh();  
    void update();  
    void render();  
  
private:  
    std::vector<Entity*> ents_;  
};
```

Añadir una entidad

Borrar entidades no vivas

Llamar a update/render de todas las entidades

La lista de entidades

Manager - addEntity

```
Entity* Manager::addEntity() {
```

```
    Entity *e = new Entity();
```

crear

```
    e->setAlive(true);
```

Inicializar

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

```
    ents_.push_back(e);
```

Añadir a la lista de
entidades

```
    return e;
```

```
}
```

Como en el caso de addComponent, creamos la entidad dentro de addEntity para tener más control sobre la gestión de la memoria ...

Manager - refresh

Su objetivo es borrar todas las entidades muertas, es decir las que han salido del juego en la última iteración

```
void Manager::refresh() {  
    ents_.erase(  
        std::remove_if(ents_.begin(), ents_.end(), [](Entity *e) {  
            if (e->isAlive()) {  
                return false;  
            } else {  
                delete e;  
                return true;  
            }  
        }), //  
    ents_.end());  
}
```

Para cada elemento de `ents_`, `remove_if` llama a la función (lambda expression) para decidir si borrarlo o no, aprovechamos para liberar la memoria también...

Manager - update y render

```
void Manager::update() {  
    auto n = ents_.size();  
    for (auto i = 0u; i < n; i++)  
        ents_[i]->update();  
}
```

```
void Manager::render() {  
    auto n = ents_.size();  
    for (auto i = 0u; i < n; i++)  
        ents_[i]->render();  
}
```

ejecuta update() y render()
de todas las entidades ...

Si añadimos una entidad mientras ejecutando los bucles,
no se va a llamar a su update/render (en realidad puede
llamar a su render, pero lo ignoramos de momento ...)

Manager - Constructor y Destructor

```
Manager::Manager():  
    ents_() {  
ents_.reserve(100);  
}
```

```
virtual ~Manager::Manager(){  
for (auto e : ents_){  
    delete e;  
}  
}
```

Reservar suficiente memoria para no evitar resizing de std::vector

Borrar todas las entidades

Usar Namespace

Para evitar conflictos de nombres, suponemos que Component, Entity, Manager y todas la declaración en ecs.h están dentro de un namespace ecs

```
namespace ecs {
```

```
// ...
```

```
}
```



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

Para usar todo lo que hay dentro (classes, variables, tipo, constantes, etc) tenemos que poner ecs:: como prefijo

p.ej., ecs::Component

Ejemplo del bucle principal del juego

```
while (!exit_){
```

```
...
```

```
    manager_->update();
```

```
...
```

```
    sdlutils().clearRenderer();
```

```
    manager_->render();
```

```
    sdlutils().presentRenderer();
```

```
...
```

```
    manager_->refresh();
```

```
...
```

```
}
```

Actualizar entidades

Renderizar entidades

Borrar entidades no vivas

- ◆ La clase Game tendrá un Manager, el método init del Game crea las entidades, etc.
- ◆ El sitio de llamada a refresh() depende mucho de la lógica del juego ...

Ejemplos de Componentes y Entidades



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

Transform

```
class Transform: public ecs::Component {  
public:  
    Transform(...) : ... { ... }  
    virtual ~Transform() ...  
    inline Vector2D& getPos() ...  
    inline Vector2D& getVel() ...  
    ...  
private:  
    Vector2D position_;  
    Vector2D velocity_;  
    float width_;  
    float height_;  
    float rotation_;  
};
```

Componente de sólo datos, tiene las características físicas de una entidad y getters/setters correspondientes. Su render y update son los de Component.

A veces, el componente Transform tiene update con movimiento básico como **pos_ = pos_ + vel_**

BallPhysics

Componente para mover la pelota en el juego Ping Pong

```
class BallPhysics: public ecs::Component {  
public:  
    BallPhysics() :  
        tr_(nullptr) {}  
    virtual ~BallPhysics();  
    void initComponent() override;  
    void update() override;  
private:  
    Transform *tr_; ←  
};
```

El componente Transform de
la entidad para modificar las
características físicas

BallPhysics

Obtener el componente Transform de la entidad.
Se supone que ya se ha añadido – se puede consultar cada vez en update también

```
void BallPhysics::initComponent() { ↗  
    tr_ = ent_->getComponent<Transform>(ecs::_TRANSFORM);  
    assert( tr_ != nullptr );  
}
```

Se puede mover esta línea
al Transform...

```
void BallPhysics::update() { ↗  
    tr_->getPos.set(tr_->getPos() + tr_->getVel());  
    float y = tr_->getPos().getY();  
    if (y <= 0 ) {  
        tr_->getVel().setY(-tr_->getVel().getY());  
        tr_->getPos.setY(0) ↗  
        ...  
    } else ...  
}
```

Actualizar la posición. Tiene en cuenta el
choque con la parte inferior y superior, etc.

Image

```
class Image: public ecs::Component {  
public:  
    Image(Texture* tex) : //  
        tr_(nullptr), //  
        tex_(tex) {  
    }  
    virtual ~Image() ...  
    void initComponent() override;  
    void render() override;  
private:  
    Transform* tr_;  
    Texture* tex_;  
};
```

Un componente para renderizar una imagen. No redefine el update() ...

El componente Transform de la entidad para consultar las características físicas

← Textura (imagen) para renderizar

Image

Obtener el Transform.

```
void Image::initComponent() {  
    tr_ = ent_->getComponent<Transform>(ecs::__TRANSFORM);  
    assert( tr_ != nullptr );  
}
```

```
void Image::render() {  
    SDL_Rect dest = build_sdlrect(tr_->getPos(),  
                                  tr_->getW(),  
                                  tr_->getH());  
  
    tex_->render(dest, tr_->getRot());  
}
```

Renderizar la textura usando las características físicas de la entidad

Entidad para la Pelota (en PingPong)

```
void Game::init() {  
    ...  
    manager_ = new Manager();  
    ...  
    auto ball = manager_->addEntity();  
    auto ballTR = ball->addComponent<Transform>(...);  
    ball->addComponent<BallPhysics>(...);  
    ball->addComponent<Image>(...);  
    ballTR->getPos().set( ... );  
    ...  
}
```

Añadir una entidad

Añadir componentes

Pasa una textura para la pelota

Posición inicial de la pelota, etc.

Lecture Notes
Santh Genaim
TPV2 2021/22
Universidad Complutense de Madrid

¡Cuidado! Varios Comp., mismo Id.

El diseño actual permite asignar el mismo identificador a varios components, y así poder poner uno de ellos en la posición correspondiente — esto puede ser útil cuando queremos tener varias **versiones** del mismo componente, pero también permite hacer errores que el compilador no puede detectar ... usar **con cuidado!**

```
e->addComponent<Transform>(ecs::__TRANSFORM,...)
```

```
// p.ej., SuperTransform es otra forma de especificar la  
// características físicas (al mejor hereda de Transform)  
//
```

```
e->addComponent<SuperTransform>(ecs::__TRANSFORM,...)
```

Identificadores vs. Implementaciones

En nuestro diseño usamos identificadores de componentes y implementaciones de ecs::Componente

```
enum cmpId : cmpId_type {  
    _TRANSFORM = 0,  
    _IMAGE,  
    ...  
};
```

```
class Transform: public ecs::Component { ... }  
class Image: public ecs::Component { ... }  
...
```

Nosotros tenemos que mantener la relación entre los dos y usarlos correctamente

```
e->addComponent<Transform>(ecs::_TRANSFORM,...)  
...  
e->getComponent<Transform>(ecs::_TRANSFORM);
```

Identificadores vs. Implementaciones

Esto es un problema porque permite hacer errores que el compilador no puede detectar ...

```
e->addComponent<Image>(ecs::__TRANSFORM,...)
```

...

```
auto c = e->getComponent<Transform>(ecs::__IMAGE)
```

...

Necesitamos quitar esa forma de tener que pasar el tipo y su identificador, queremos pasar sólo el tipo ...

```
e->addComponent<Transform>(e1,e2,...)
```

```
e->removeComponent<Transform>()
```

```
e->getComponent<Transform>()
```

```
e->hasComponent<Transform>()
```

Sin Pasar los Identificadores!

- ◆ Una forma de conseguirlo es pedir que cada componente tenga un atributo estático que lleva su identificador (un valor del enum cmpId).
- ◆ Lo declaramos usando `constexpr` para indicar al compilador que es una constante que se puede calcular durante la compilación (esto permite hacer más optimizaciones durante la compilación)

```
class Transform: public ecs::Component {  
public:  
    constexpr static cmpId_type id = ecs::_TRANSFORM;  
    ...  
};  
  
class Image: public ecs::Component {  
public:  
    constexpr static cmpId_type id = ecs::_IMAGE;  
    ...  
};
```

addComponent sin el Identificador

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

constexpr cmpId_type cId = T::id;

removeComponent<T>()

```
currCmps_.push_back(c);
cmps_[cId] = c;
```

...

}

e->addComponent<Transform>(e1,e2,...)

cId es el valor del atributo
estático 'id' de la clase T

Otros métodos sin el Identificador

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

```
}
```

```
template<typename T>
inline T* getComponent(cmpId_type cId) {
    constexpr cmpId_type cId = T::id;
```

```
}
```

```
template<typename T>
inline bool hasComponent(cmpId_type cId) {
    constexpr cmpId_type cId = T::id;
```

```
...
```

```
}
```

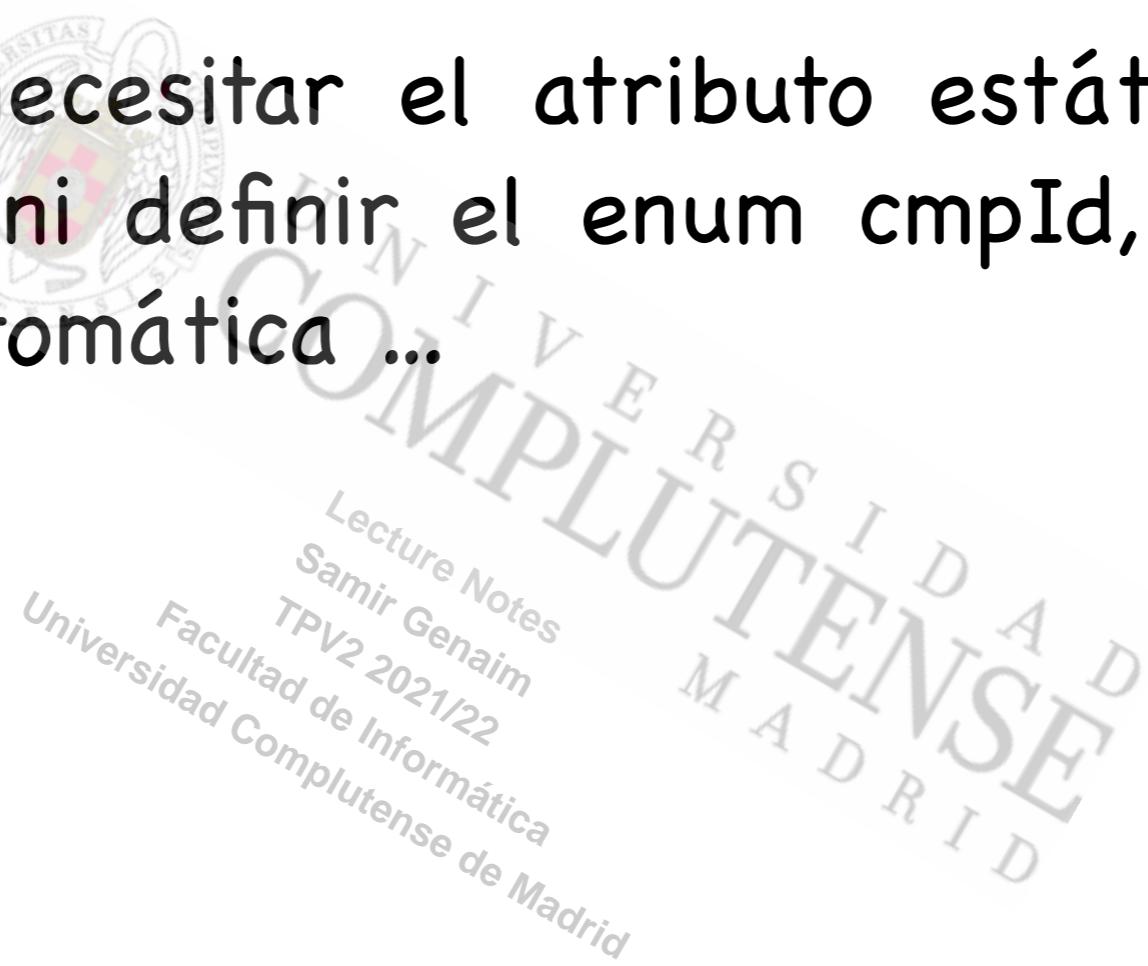
```
e->removeComponent<Transform>()
e->getComponent<Transform>()
e->hasComponent<Transform>()
```

¡Cuidado!

- ◆ Usando los atributos estáticos es más cómodo de tener que pasar el tipo y el identificador, pero todavía podemos hacer errores que el compilador no puede detectar.
- ◆ Ej. 1: Si usamos herencia entre componentes (algo que no se debe hacer normalmente), un componente hereda el atributo 'id' de otro componente. Esto puede ser útil (diferente componentes mismo identificador), pero también peligroso porque si se olvida definir 'id' lo hereda de la clase base.
- ◆ Ej. 2: Se puede asignar el mismo identificador a varios componentes completamente distintos, p.ej., asignar `ecs::__TRANSFROM` a `Image` y `Transform`.

Otras formas de conseguirlo ...

- ◆ Mas adelante vamos a ver otra forma de quitar la necesidad del pasar los identificadores, usando **template meta-programming**.
- ◆ No vamos a necesitar el atributo estático 'id' en los componentes, ni definir el enum cmpId, todo se hace de manera automática ...



Groups y Handlers

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 Complutense de Madrid, para la publicación o distribución posterior en las Redes sociales (incluida la divulgación en Internet) o servicios de comunicación en línea, o servicios de protección de la propiedad intelectual y de datos y/o la normativa de protección de datos y/o la responsabilidad de la persona infractora. Si encuentras este material en otro sitio web que no tenga la extensión ucm.es, puedes vulnerar la normativa de protección de datos y/o la responsabilidad de la persona infractora. Si encuentras este material en otro sitio web que no tenga la extensión ucm.es, avísanos en denunciacontenido@ucm.es o reportcontent@ucm.es

Handlers (Objetivo)

- ◆ Normalmente en una parte del juego, necesitamos acceso a una entidad específica (p.ej. la pelota en Ping Pong) para acceder a sus componentes, etc.
- ◆ Se puede pasar a esas partes las referencias a todas las entidades que necesitan ...
- ◆ Sería más sencillo proporcionar un acceso global a entidades de interés, a través del manager, para evitar la modificación del código si necesitamos acceso a más entidades ...

Handlers (Implementación)

```
namespace ecs {  
...  
using hdldrId_type = uint8_t;  
enum hdldrId : hdldrId_type { BALL, ... ,_LAST_HDLR_ID };  
constexpr hdldrId_type maxHdldrId = _LAST_HDLR_ID;  
...  
}  
  
class Manager {  
public:  
...  
    inline void setHandler(hdldrId_type hId, Entity *e) {  
        hdlrs_[hId] = e;  
    }  
    inline Entity* getHandler(hdldrId_type hId) {  
        return hdlrs_[hId];  
    }  
...  
private:  
...  
    std::array<Entity*, ecs::maxHdldrId> hdlrs_;  
}
```

Este código implementa la clase Manager que gestiona los handlers. La clase tiene un array privado de tipo Entity* llamado hdlrs_ que almacena los handlers para cada ID. Los IDs están definidos en el espacio de nombres ecs con un tipo de dato hdldrId_type y un enum que incluye BALL y otros valores hasta LAST_HDLR_ID. La constexpr maxHdldrId establece el límite superior para los IDs. La clase Manager proporciona métodos para establecer y obtener un handler dado su ID.

Handlers (Ejemplo de uso)

```
void Game::init() {  
    ...  
    manager_ = new Manager();  
    ...  
    Entity *ball = manager_->addEntity();  
    ...  
    manager_->setHandler(ecs::BALL, ball);  
    ...  
}
```

Declarar a que entidad corresponde ecs::BALL

Acceder a la entidad que corresponde a ecs::BALL
desde un componente por ejemplo

```
auto ball = mngr_->getHandler(ecs::BALL);
```

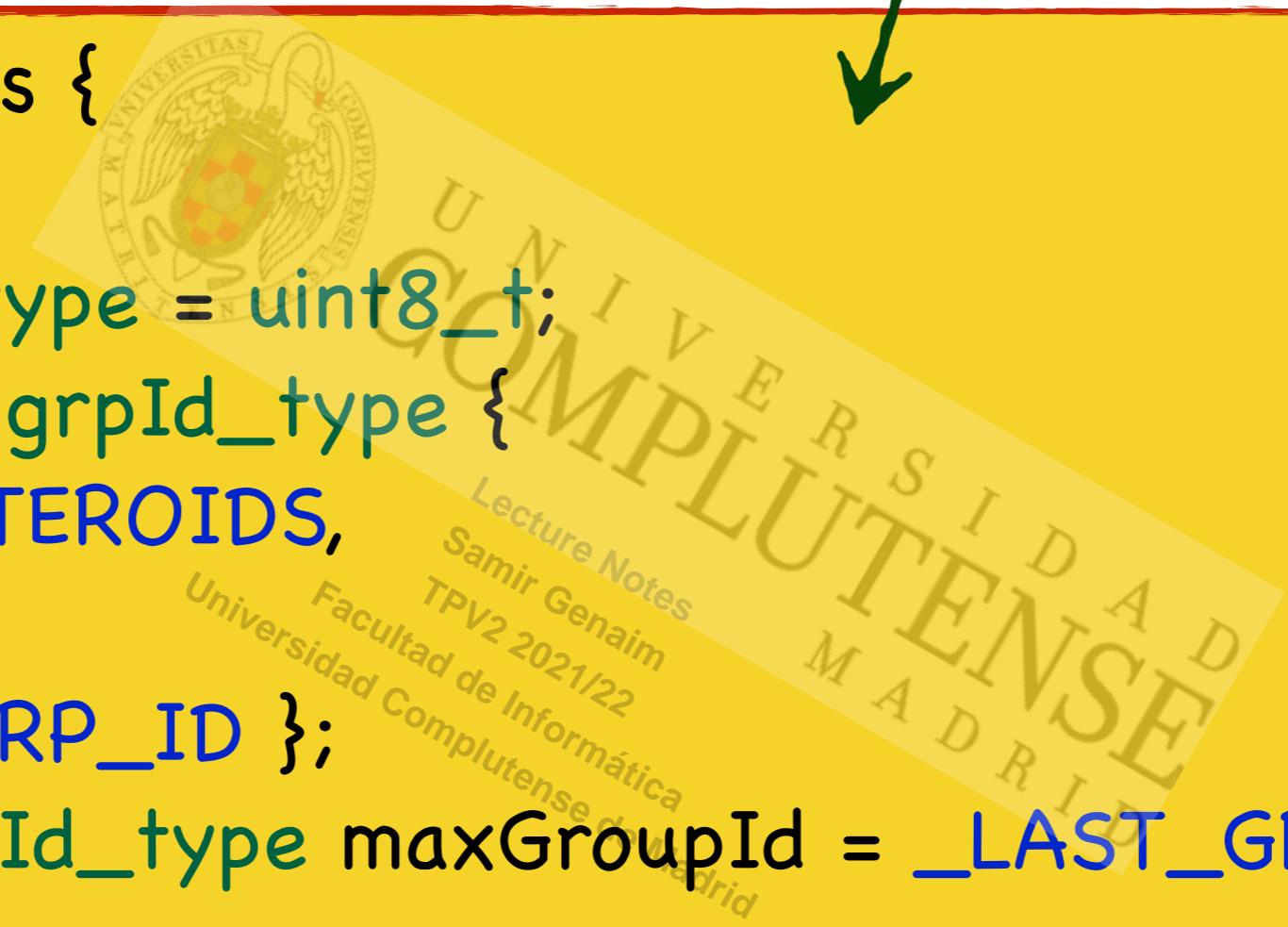
Groups (Objetivo)

- ◆ Muchas veces necesitamos identificar entidades similares, p.ej., los asteroides, las fantasmas, las balas, etc.
- ◆ En principio para hacer **operaciones** sobre esas entidades, p.ej., comprobar choques de todos los asteroides con el caza, dibujar todas las **fantasmas** y sólo después dibujar todas la cerezas, etc.
- ◆ Una posibilidad de **conseguirlo es asignar etiquetas (grupos)** a las entidades.
- ◆ La clase Entity tiene que permitir al programador asignar una etiqueta a la entidad, quitar una etiqueta de la entidad, consultar si la entidad tiene una etiqueta, etc.

Groups VI (Implementación)

Definir los identificados de grupos (etiquetas) disponibles como enum

```
namespace ecs {  
...  
using grpId_type = uint8_t;  
enum grpId : grpId_type {  
    _grp_ASTEROIDS,  
    ...  
    _LAST_GRP_ID };  
constexpr grpId_type maxGroupId = _LAST_GRP_ID;  
...  
}
```



A large watermark of the University of Complutense Madrid logo and text is visible across the slide. The text includes 'UNIVERSIDAD COMPLUTENSE MADRID', 'Lecture Notes', 'Samir Genaim', 'TPV2 2021/22', 'Facultad de Informática', and 'Universidad Complutense de Madrid'.

Groups VI (Implementación)

```
class Entity {  
    ...  
    inline void addToGroup(grpId_type gId) {  
        if (!groups_[gId]) groups_[gId] = true;  
    }  
    inline void removeFromGroup(grpId_type gId) {  
        if (groups_[gId]) groups_[gId] = false;  
    }  
    inline bool hasGroup(grpId_type gId) {  
        return groups_[gId];  
    }  
private:  
    ...  
    std::bitset<ecs::maxGroupId> groups_;
```

Usamos bitset en lugar de array
de boolean, más eficiente

```
class Manager {  
    ...  
    inline const auto& getEntities() { return ents_; }  
    ...
```

Groups VI (Ejemplo de uso)

Cuando creamos las entidades de los asteroids les asignamos el grupo ecs::ASTEROIDS

```
for(...) {  
    Entity *a = manager_->addEntity();  
    ...  
    a->addToGroup(ecs::_grp_ASTEROIDS)  
    ...  
}
```

```
for (auto &e : mngr_->getEnteties()) {  
    if (e->hasGroup(ecs::_grp_ASTEROIDS)) {  
        // ...  
    }  
}
```

Procesar las entidades que pertenecen al grupo ecs::ASTEROIDS

Groups V2: Manager

Se puede mejorar el manejo de grupos y mantener listas de entidades de cada grupo. Así no tenemos que recorrer la lista de todas las entidades si nos interesa sólo un grupo ...

```
class Manager {  
    ...  
    inline const auto& getEntitiesByGroup(grpId_type gId) {  
        return entsByGroup_[gId];  
    }  
    ...  
private:  
    ...  
    std::array<std::vector<Entity*>, maxGroupId> entsByGroup_;  
}
```

Un array de lista de entidades por grupo y un método para pedir una lista del grupo gId

```
for (auto &e : mngr_->getEntitiesByGroup(ecs::__grp_ASTEROIDS)) {  
    // ...  
}
```

Groups V2: Entity

```
class Entity {  
...  
    inline void addToGroup(grpId_type gId) {  
        if (!groups_[gId]) {  
            groups_[gId] = true;  
            mngr_->addToGroupList(gId, this);  
        }  
    }  
}
```

Además de cambiar el estado, pide al manager que le añade a la lista del grupo gId

```
    inline void removeFromGroupList(grpId_type gId) {  
        if (groups_[gId]) {  
            groups_[gId] = false;  
        }  
    }  
...  
}
```

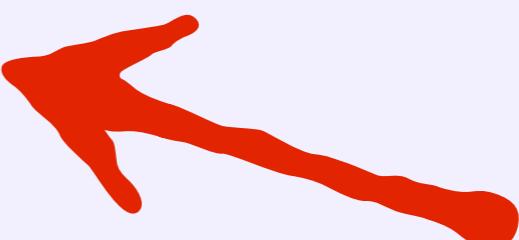
No vamos a quitarlo de la lista del grupo gId, lo vamos a hacer en refresh() para no borrar un elemento de la lista mientras la estamos recorriendo ...

```
void Manager::addToGroupList(grpId_type gId, Entity *e) {  
    entsByGroup_[gId].push_back(e);  
}
```

Groups V2: refresh

En Manager::refresh(), aparte de borrar las entidades muertas de la lista de entidades, tenemos que quitarlas de las listas de grupos también ... y también los que no pertenecen a cada grupo ...

```
void Manager::refresh() {  
    for (ecs::grpId_type gId = 0; gId < ecs::maxGroupId; gId++) {  
        auto &grpEnts = entsByGroup_[gId];  
        grpEnts.erase(  
            std::remove_if(grpEnts.begin(), grpEnts.end(),  
                           [gId](Entity *e) {  
                               return !e->isAlive() || !e->hasGroup(gId);  
                           }),  
            grpEnts.end());  
    }  
    ...  
}
```



Groups V2: Constr. del Manager

```
Manager::Manager() :  
    ents_() {  
    ents_.reserve(100);  
    for (auto &grpEnts : entsByGroup_) {  
        grpEnts.reserve(100);  
    }  
}  
}
```

Reservar suficiente memoria
para no evitar resizing de
std::vector

Groups V3: Mecanismo más Sencillo

- ◆ Normalmente no es necesario usar un mecanismo tan complicado para los grupos.
- ◆ Es suficiente asignar cada entidad a un sólo grupo, en el momento de la creación, y sin poder cambiarlo.
- ◆ Tener acceso a las entidades por grupo
- ◆ Es un mecanismo más sencillo y más eficiente

```
namespace ecs {  
enum grpId : grpId_type {  
    _grp_GENERAL, ← Un grupos especial para usarlo  
    _grp_ASTEROIDS,  
    ...  
    _LAST_GRP_ID };  
}
```

por defecto si la entidad no pertenece a un grupo definido por el usuario ...

Groups V3: addEntity

```
Entity* Manager::addEntity(ecs::grpId_type gId =  
                           ecs::_grp_GENERAL) {
```

```
Entity *e = new Entity();  
e->setAlive(true);  
e->setContext(this);  
entsByGroup_[gId].push_back(e);  
return e;
```

```
}
```

```
const auto& Manager::getEntities(grpId_type gId =  
                           ecs::_grp_GENERAL) {
```

```
return entsByGroup_[gId];
```

```
}
```



Indicamos a que grupo queremos añadiré a la entidad

Se puede consultar la lista de entidades de un grupo

La clase Entity no tiene información sobre los grupos, no tiene el std::bitset. Si es necesario, se puede almacenar el identificador del grupo en la entidad también ...

Groups V3 (Ejemplo de uso)

Pasamos el grupos a addEntity

```
for(...) {  
    Entity *a = manager_->addEntity(ecs::__grp_ASTEROIDS);  
    ...  
}
```

```
for (auto &e : mngr_->getEntities(ecs::__grp_ASTEROIDS)) {  
    // ...  
}
```

Procesar las entidades que pertenecen
al grupo ecs::__grp_ASTEROIDS

Groups V3: update y render

```
void Manager::update() {  
    for (auto &ents : entsByGroup_) {  
        auto n = ents.size();  
        for (auto i = 0u; i < n; i++)  
            ents[i]->update();  
    }  
}
```

ejecuta update() y render()
de todas las entidades de
todos los grupos ...

```
void Manager::render() {  
    for (auto &ents : entsByGroup_) {  
        auto n = ents.size();  
        for (auto i = 0u; i < n; i++)  
            ents[i]->render();  
    }  
}
```

Groups V3: refresh

```
void Manager::refresh() {
    for (ecs::grpId_type gId = 0; gId < ecs::maxGroupId; gId++) {
        auto &grpEnts = entsByGroup_[gId];
        grpEnts.erase(
            std::remove_if(grpEnts.begin(), grpEnts.end(),
                [](Entity *e) {
                    if (e->isAlive()) {
                        return false;
                    } else {
                        delete e;
                        return true;
                    }
                }),
            grpEnts.end()));
    }
}
```

Salvo autorización expresa, los materiales entregados a estudiantes por cualquier medio durante la cursada solo se podrán utilizar para el estudio de la asignatura correspondiente en la Universidad Complutense de Madrid. La publicación o distribución posterior incluida la divulgación en redes sociales y/o la de propiedad intelectual y/o la de normalidad de la persona infractora. Se encuentra en la normativa de protección de datos y/o la de propiedad intelectual en otro sitio web que no sea la extensión ucm.es, avales o denuncia contenido@ucm.es. Siencuentras más información en la dirección reportcontent@ucm.es

Groups V3: Con. y Des. Del Manager

```
Manager::Manager() : hdlrs_(), entsByGroup_() {  
    for (auto &groupEntities : entsByGroup_) {  
        groupEntities.reserve(100);  
    }  
}
```

Reservar suficiente memoria como antes.

```
virtual ~Manager::Manager() {  
    for (auto &ents : entsByGroup_) {  
        for (auto e : ents)  
            delete e;  
    }  
}
```

Borrar todas las entidades de todos los grupos

La clase Manager ya no tiene la lista ents_ de las entidades, sólo listas por grupo

Resumen

- ◆ El diseño actual de entidades y componentes es sólo una posible manera de hacerlo
 - ✓ Se pueden añadir más métodos a la clase Component
 - ✓ Se puede modificar la clase Entity para distinguir entre componentes de entrada, física, gráfica, o datos, etc.
- ◆ Todos los diseños que vamos a ver durante la asignatura son idea que se pueden combinar/modificar para los objetivos de un juego específico ...
- ◆ Hay que tener cuidado si cambias los componentes de una entidad durante el juego (hay que asegurarse de que la implementación de Entity lo hace de manera segura)
- ◆ Es muy importante tener control sobre la gestión de la memoria (todo se crea en addComponent y addEntity) porque mas adelante vamos a usar técnicas más eficiente para la gestión de la memoria, etc.