

# Patrones de Diseño

## Entity-Component-System (Parte I)

TPV2  
Samir Genaim

# Objetivos

- ✦ Ya hemos dado el primer paso hacia el diseño basado en componentes
- ✦ Hemos sacado los comportamientos de los 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. **El objetivo es sacar también los datos fuera en componentes**
- ✦ Usamos terminología estándar: Entity, Component, System. Muy parecido a lo que ya hemos visto ...

# ECS (sin la S de momento)

- ♦ ECS = Entity-Component-System
- ♦ **Component**: representa datos o comportamientos de una entidad — en nuestro caso **muy particular** vamos a definir un componente cómo clase que tiene métodos update y draw.
- ♦ **Entity**: un container de componentes
- ♦ **EntityManager**: una clase para agrupar las entidades y hacer operaciones sobre entidades, a veces no hace falta y la lista de entidades puede ser parte de la clase principal (e.g., Engine)

# Component

```
class Component {  
public:
```

```
    Component(ecs::CmpId id) { id_ = id; }
```

```
    virtual ~Component() { }
```

```
    inline void setEntity(Entity* e) { entity_ = e; }
```

```
    inline ecs::CmpId getId() const { return id_; }
```

```
    ...
```

```
    virtual void init() { }
```

```
    virtual void update() { }
```

```
    virtual void draw() { }
```

```
protected:
```

```
    Entity* entity_;
```

```
    ecs::CmpId id_;
```

```
};
```

Métodos para cambiar y consultar valores de atributos

- ♦ `init()`: se invoca cuando se añade el componente a una entidad.
- ♦ `update()`: para actualizar el estado
- ♦ `draw()`: para renderizar el estado
- ♦ Se pueden añadir más métodos, p.ej., `handleInput`, depende de las necesidades del juego

Referencia a su entidad, su identificador, etc. Se pueden añadir mas atributos, p.ej., referencia al `SDLGame`, etc.

# Header File: ecs.h

```
namespace ecs {  
using CmpIdType = std::size_t;
```

```
enum CmpId : CmpIdType {  
    Transform = 0,  
    SimpleMoveBehavior,  
    BallMoveBehaviour,  
    PaddleMoveBehaviour,  
    Rectangle,
```

```
    ...  
    // number of components
```

```
    _LastCmptId_  
};
```

```
constexpr std::size_t maxComponents = _LastCmptId_;  
}
```

namespace para que los ids de componentes no tengan conflictos con nombres de clases (si queremos usar el mismo nombre)

identificadores de componentes

El máximo número de componentes. Se usa en Entity para el tamaño del array de componentes

# EntityManager

```
class Entity {  
    using uptr_cmp = std::unique_ptr<Component>;  
public:  
    Entity();  
    virtual ~Entity();  
    inline EntityManager* getEntityMngr() ...  
    inline void setEntityMngr(EntityManager*) ...  
    inline bool isActive() ...  
    inline void setActive(bool active) ...  
    ...  
private:  
    bool active_;  
    EntityManager *mngr_  
    std::vector<uptr_cmp> cmp_  
    std::array<Component*, ecs::maxComponents> cmpArray_ = { };  
};
```

Tipo de unique\_ptr de Component (for brevity)

cambiar/consultar atributos

El estado, el EntityManager lo usa para quitar la entidad de la lista cuando sea false

Referencia a su EntityManager

Los componentes - next slide



# Entity

```
class Entity {  
    using uptr_cmp = std::unique_ptr<Component>;  
    ...  
    std::vector<uptr_cmp> cmp_  
    std::array<Component*, ecs::maxComponents> cmpArray_ = { };  
};
```

- ♦ `cmpArray_` se usa para tener un acceso rápido a los componentes por identificador — su tamaño es como el máximo número de componentes `ecs::maxComponents`
- ♦ `cmp_` se usa para recorrer a todos los componentes de la entidad. Usamos `unique_ptr` para que el puntero se borre automáticamente al borrar el elemento del vector o al final cuando se borra el vector

# Entity - addComponent

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

Variadic Template  
Function

```
    T *c(new T(std::forward<Targs>(args)...));  
    uptr_cmp uPtr(c);
```

Crea el componente y  
lo pone en unique\_ptr

```
    if ( cmpArray_[c->getId()] != nullptr ) {  
        auto old_uPtr = std::find_if(...);  
        *old_uPtr = std::move(uPtr);
```

```
    } else {  
        cmp_.push_back(std::move(uPtr));  
    }
```

```
    cmpArray_[c->getId()] = c;
```

Si existe un componente  
con el mismo identificador  
cámbialo (el anterior se  
borra). Si no, añádelo al  
final.

```
    c->setEntity(this);  
    c->init();  
    return c;
```

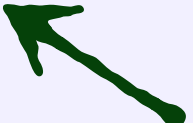
Inicializar el componente

```
}
```



# Entity - ejemplo de addComponent

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



SomeClass \*c(new SomeClass(a1,a2,...))

Ejecutando

e->addComponent<ClassName>(a1,a2,...)

T = ClassName

Targs = los tipos de a1,a2, ...

args = 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?

```
inline T* addComponent(Component *c) {  
    ...  
}
```



- ♦ Así sabemos de quien es la responsabilidad de borrar el componente (y liberar la memoria dinámica)
- ♦ En general, tenemos más control sobre la gestión de la memoria, p.ej., en lugar de usar **new** podemos usar object pool o factory y el usuario no tiene que cambiar su código (ni saberlo!)

# Entity - GetComponent, hasComp...

Devuelve el componente haciendo casting a T\*

```
SomeClass *c = e->GetComponent<SomeClass>(ecs::SomeId);
```

```
template<typename T>  
inline T* GetComponent(ecs::CmpId id) {  
    return static_cast<T*>(cmpArray_[id]);  
}  
  
inline bool hasComponent(ecs::CmpId id) {  
    return cmpArray_[id] != nullptr;  
}
```



Simplemente comprueba si tiene un componente en la posición correspondiente de `cmpArray_`

# Entity - update y draw

Recorremos la lista de componentes sin usar iterator y hasta el tamaño actual porque se pueden añadir componentes mientras recorriendo — puede crear problemas depende en que tipo de container guardamos los componentes

```
inline void update() {  
    std::size_t n = cmp_.size();  
    for(int i=0; i<n; i++){  
        cmp_[i]->update();  
    }  
}
```

ejecuta update()  
y draw() de todos  
lo componentes ...

```
inline void draw() {  
    std::size_t n = cmp_.size();  
    for(int i=0; i<n; i++){  
        cmp_[i]->draw();  
    }  
}
```

¡Cuidado! remplazar un componente  
mientras recorriendo es peligroso.  
Piensa que puede pasar ...

# EntityManager

```
class EntityManager {  
    using uptr_ent = std::unique_ptr<Entity>;  
public:  
    EntityManager();  
    virtual ~EntityManager();  
  
    Entity* addEntity();  
    void refresh();  
    void update();  
    void draw();  
  
private:  
    std::vector<uptr_ent> ents_;  
};
```

Tipo de unique\_ptr de Entity (for brevity)

Añadir una entidad

Borrar entidades no activas

llamar a update/draw de las entidades

Las entidades, se usa unique\_ptr para que se borren automaticamente

Se puede convertir en Singleton si es necesario

# EntityManager - addEntity

```
Entity* EntityManager::addEntity() {  
    Entity *e = new Entity();  
    e->setEntityMngr(this);  
    ents_.emplace_back(e);  
    return e;  
}
```

The diagram illustrates the steps of the `addEntity` method:

- crear** (red box): Points to the `new Entity()` line.
- inicializar** (blue box): Points to the `e->setEntityMngr(this);` line.
- Añadir a la lista de entidades** (green box): Points to the `ents_.emplace_back(e);` line.

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



# EntityManager - refresh

Su objetivo es borrar todas las entidades no activas, es decir las que han salido del juego en la última iteración

```
void EntityManager::refresh() {
```

```
    ents_.erase(
```

```
        std::remove_if( ←
```

```
            ents_.begin(),
```

```
            ents_.end(),
```

```
            [](const uptr_ent &e) {
```

```
                return !e->isActive();
```

```
            }),
```

```
            ents_.end()); ←
```

```
}
```

remove\_if desplaza a todos los elementos que cumplen la condición al final del vector y devuelve un iterator a la primera posición de dichos elementos ...

... después erase borra a partir de ese iterator hasta el final (el uso de remove\_if es para evitar desplazamientos innecesarios)

# EntityManager - update y draw

Recorremos la lista de entidades sin usar iterator y hasta el tamaño actual porque se pueden añadir entidades mientras recorriendo — puede crear problemas depende en que tipo de container guardamos las entidades

```
void EntityManager::update() {  
    std::size_t n = ents_.size();  
    for (int i = 0; i < n; i++)  
        ents_[i]->update();  
}
```

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

```
void EntityManager::draw() {  
    std::size_t n = ents_.size();  
    for (int i = 0; i < n; i++)  
        ents_[i]->draw();  
}
```

# El bucle principal del juego

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

Actualizar el gestor de eventos de entrada

```
...  
inputHandler_>update();  
entityManager_>update();  
entityManager_>refresh();  
entityManager_>draw();  
...  
}
```

Actualizar entidades

Borrar entidades no activas

Renderizar entidades

- ♦ El sitio de llamada a refresh depende del juego, incluso si no mueren entidades durante el juego se puede la llamada.

# **Ejemplos de Componentes y Entidades**

# Transform

```
class Transform: public Component {  
public:  
    Transform(...) : Component(ecs::Transform), ... { ... }  
    virtual ~Transform() ...  
  
    inline const Vector2D& getPos() ...  
    inline void setPos(const Vector2D &pos) ...  
    inline const Vector2D& getVel() ...  
    inline void setVel(const Vector2D &vel) ...  
  
    ...  
private:  
    Vector2D position_;  
    Vector2D velocity_;  
    double width_;  
    double height_;  
    double rotation_;  
};
```


El identificador del componente. Usamos el mismo nombre como la clase pero puede ser otro ...

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

# BallMoveBehaviour

Componente de física para el movimiento de la pelota en el juego Ping Pong

```
class BallMoveBehaviour: public Component {
public:
    BallMoveBehaviour() :
        Component(ecs::BallMoveBehaviour), tr_(nullptr) {
    }
    virtual ~BallMoveBehaviour();
    void init() override;
    void update() override;
private:
    Transform *tr_;
```



El componente Transform de la entidad para modificar las características físicas



# BallMoveBehaviour

```
void BallMoveBehaviour::init() {  
    tr_ = entity_->GetComponent<Transform>(ecs::Transform);  
}  
  
void BallMoveBehaviour::update() {  
    tr_->setPos(tr_->getPos() + tr_->getVel());  
    double y = tr_->getPos().getY();  
    if (y <= 0 || y + tr_->getH() >= game_->getWindowHeight()) {  
        tr_->setVelY(-tr_->getVel().getY());  
        game_->getAudioMgr()->playChannel( ... );  
    }  
}
```

Obtener el componente Transform de la entidad.

Actualizar la posición. Tiene en cuenta el choque con los lados inferior y superior (efecto rebote con sonido), etc.

# ImageViewer

```
class ImageViewer: public Component {
```

```
public:
```

```
    ImageViewer(Texture* tex) : //
```

```
        Component(ecs::ImageViewer),
```

```
        tr_(nullptr), //
```

```
        tex_(tex) {
```

```
}
```

```
    virtual ~ImageViewer() ...
```

```
    void init() override;
```

```
    void draw() 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

# ImageViewer

```
void ImageViewer::init() {  
    // tr_ = GETCMP1_(Transform);  
    tr_ = entity_->getComponent<Transform>(ecs::Transform);  
}
```

Obtener el Transform. Se puede usar el macro GETCMP1\_ de ecs.h (ver slide mas adelante)

```
void ImageViewer::draw() {  
    SDL_Rect dest = RECT(tr_->getPos().getX(),  
                          tr_->getPos().getY(),  
                          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 PingPong::initGame() {
```

```
...
```

```
entityManager_ = new EntityManager(game_);
```

Añadir una entidad

```
...
```

```
Entity *ball = entityManager_>addEntity();
```

```
Transform *ballTR = ball->addComponent<Transform>();
```

```
ball->addComponent<BallMoveBehaviour>();
```

```
ball->addComponent<ImageViewer>(...);
```

```
ballTR->setPos( ... );
```

```
...
```

```
}
```

Añadir componentes

Pasa una textura para la pelota

Posición inicial de la pelota, etc.

# Los Macros GETCMP en ecs.h

ecs.h incluye algunos macros para llamar a addComponent de manera más corta ...

Usar sólo dentro de un componente (porque refiere a entity\_) cuando el identificador es como el nombre de la clase

```
GETCMP1_(A) → entity_->getComponent<A>(ecs::A)
```

Usar sólo dentro de un componente (porque refiere entity\_) cuando el identificador es distinto del nombre de la clase

```
GETCMP2_(A,ecs::AID) → entity_->getComponent<A>(ecs::AID)
```

Usar cuando el identificador es como el nombre de la clase

```
GETCMP2(e,A) → e->getComponent<A>(ecs::A)
```

Usar cuando el identificador es distinto del nombre de la clase

```
GETCMP2_(A,ecs::AID) → e->getComponent<A>(ecs::AID)
```

# 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 quitar el draw de Component y modificar la clase Entity para distinguir entre componentes de entrada, física, gráfica, o datos (sería más eficiente porque ahorramos alguna llamada a métodos)
- ✦ 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 memoria -- todo se crea en addComponent y addEntity