

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. Nuestro objetivo es sacar también los datos fuera en componentes.
- ◆ Usamos terminología estándar: Entity, Component, System. Muy parecido a lo que ya hemos visto ...

EC (sin la S de momento)

- ◆ EC = Entity-Component
- ◆ Component: representa datos y/o comportamientos de una entidad – en nuestro caso **muy particular** vamos a definir un componente como clase que tiene métodos update y render puede .
- ◆ Entity: un container de componentes
- ◆ Manager: 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 (p.ej., Engine, Game, etc.)

Component

```
class Component {  
public:  
    virtual ~Component() { }  
    inline void setEntity(Entity* e) { entity_ = e; }  
    ...  
    virtual void init() { } ←  
    virtual void update() { } ←  
    virtual void render() { } ←  
protected:  
    Entity* entity_;  
};
```

The diagram illustrates the structure of the Component class with several annotations:

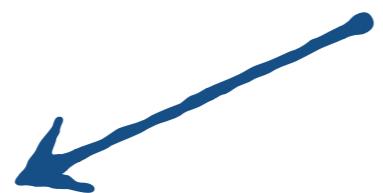
- An annotation for the `setEntity` method points to its declaration with the text "Para pasarle un puntero a su entidad".
- Annotations for the `init`, `update`, and `render` methods are grouped together, pointing to the `init` method with the text "se invoca al añadir un componente a una entidad. Para inicializar el componente si es necesario".
- Annotations for `update` and `render` are grouped together, pointing to `update` with the text "para actualizar el estado, manejar entrada, etc," and to `render` with the text "para renderizar el estado".
- An annotation for the protected member `entity_` points to its declaration with the text "Referencia a su entidad, accesible desde las sub-clases".
- A final annotation at the bottom right points to the entire class definition with the text "Se pueden añadir métodos, p.ej., handleInput, o definir varios tipos de componentes (entrada, física, gráfica, datos, etc.), depende de las necesidades del juego".

Referencia a su entidad,
accesible desde las sub-
clases

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

Header File: ecs.h

namespace para evitar conflictos de nombres, identificadores, etc.



```
namespace ecs {  
  
constexpr std::size_t maxComponents = 10;  
  
}
```



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

Entity

```
class Entity {  
    using uptr_cmp = std::unique_ptr<Component>;  
public:  
    Entity(Manager* mngr);  
    virtual ~Entity();  
    inline Manager* getMngr() ... ← Tipo de unique_ptr de Component  
    inline void setMngr(Manager*) ... ← Para pasarle el Manager,  
y consultarla (por sus  
componentes)  
    inline bool isActive() ... ← Consultar o/y cambiar el  
estado (active o no).  
    inline void setActive(bool active) ...  
    ...  
private:  
    bool active_;  
    Manager *mngr_;  
    std::vector<uptr_cmp> components_;  
    std::array<Component*, ecs::maxComponents> cmpArray_ = { };  
};
```

Entity

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

- ◆ `cmpArray_` se usa para tener un acceso rápido a los componentes por identificador – Se supone que en componente de tipo X siempre va en la misma posición. El tamaño es como el máximo numero de componentes `ecs::maxComponents`.
- ◆ `components_` 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 ...Ts> Variadic Template Function
inline T* addComponent(Ts&&...args) {
    T *c = new T(std::forward<Ts>(args)...); Crea el componente
    auto id = ... Calcula la posición en el array a partir
    if ( cmpArray_[id] != nullptr ) { Si existe un componente con el
        removeComponent<T>() mismo identificador lo borramos
    } antes. Se puede remplazar para
        mantener el orden actual en el vector
    cmponents_.emplace_back(c);
    cmpArray_[id] = c;
    c->setEntity(this);
    c->init();
    return c; Inicializar el componente
}
```

Entity - ejemplo de addComponent

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

    ...
}
```

Ejecutando

```
e->addComponent<Transform>(exp1,exp2,...)
```

T = Transform

Ts = los tipos de exp1,exp2, ...

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(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 - getComponent, hasComp...

Devuelve el componente haciendo casting a T*

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

```
template<typename T>
inline T* getComponent() {
    auto id = ...
    return static_cast<T*>(cmpArray_[id]);
}
```

```
template<typename T>
inline bool hasComponent() {
    auto id = ...
    return cmpArray_[id] != nullptr;
}
```

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

Entity - removeComponent

```
template<typename T>
void removeComponent() {
    auto id = ...
    if (cmpArray_[id] != nullptr) {
        Component *old_cmp = cmpArray_[id];
        cmpArray_[id] = nullptr; ← Borralo de array de componentes
        components_.erase(
            std::find_if(
                components_.begin(),
                components_.end(),
                [old_cmp](const uptr_cmp &c) {
                    return c.get() == old_cmp;
                }));
    }
}
```

Si hay componente de este tipo

Borralo de array de componentes

Busca el unique_ptr correspondiente, usando `find_if`, en el vector de componentes y bórralo con `erase`. Recuerda que al borrar el unique_ptr se borra `old_cmp`

Entity - update y render

Recorremos el vector 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 = components_.size();  
    for(int i=0; i<n; i++) {  
        components_[i]->update();  
    }  
}
```

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

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

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

Calcular el identificador: intento 1

Necesitamos convertir T en un número. Lo primero que podemos pensar es resolverlo usando enum. Definimos un enum con valores exactamente como los nombres de las clases de componentes y lo ponemos en el namespace ecs para evitar conflictos con las clases

```
namespace ecs {  
    enum CmpId : std::size_t { Transform = 0, Rectangle, Image, ... }  
}
```

```
template<typename T, typename ...Ts>  
inline T* addComponent(Ts&&...args) {  
    auto id = ecs::CmpId::T;  
    ...  
}
```

Convertimos T en número refiriéndose al valor correspondiente del enum

El compilador lo rechaza! No está permitido usar un parámetro de tipo como un valor!

Calcular el identificador: intento 2

```
namespace ecs {  
enum CmpId : std::size_t { Transform = 0, Rectangle, Image, ... }  
}
```

```
template<typename T, typename ...Ts>  
inline T* addComponent(ecs::CmpId id, Ts&&...args) {  
...  
}
```

Se puede añadir como parámetro de
addComponent/getComponent/etc.

Es correcto, pero no es cómodo, siempre tenemos que pasar
el tipo y su valor correspondiente del enum, el compilador
no puede verificar que lo estamos usando correctamente ...

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

```
e->getComponent<Transform>(ecs::Transform);
```

Calcular el identificador: intento 3

```
// Compile-time list of types.  
template<typename ... Ts>  
struct TypeList {  
    // Size of the list.  
    static constexpr std::size_t size { sizeof...(Ts) };  
};  
  
template<typename, typename >  
struct IndexOf;  
  
// IndexOf base case: found the type we're looking for.  
template<typename T, typename ... Ts>  
struct IndexOf<T, TypeList<T, Ts...>> : std::integral_constant<std::size_t, 0> {  
};  
  
// IndexOf recursive case: 1 + IndexOf the rest of the types.  
template<typename T, typename TOther, typename ... Ts>  
struct IndexOf<T, TypeList<TOther, Ts...>> : std::integral_constant<std::size_t,  
    1 + IndexOf<T, TypeList<Ts...>> { }> {  
};  
  
using ComponentsList = TypeList<Transform, Rectangle, Image, ...>;
```

Es un meta-programa usando templates – ver ecs.h

Lista de componentes

Se usamos `IndexOf<T, ComponentsList>()` como número en el programa, el compilador lo remplaza por el indice del tipo `T` en la lista `ComponentsList`. `IndexOf` es un meta-programa que se “ejecuta” durante la compilación. De momento lo usamos, en unas semanas, después de ver el tema de template meta-programming lo vais a entender

Calcular el identificador: intento 3

En lugar de escribir siempre `IndexOf<T, ComponentsList>()` podemos definir algo breve de la siguiente manera

```
namespace ecs {  
...  
  
template<typename T>  
using cmpIdx = IndexOf<T, ComponentsList>;  
...  
}
```

```
namespace ecs {  
...  
  
template<typename T>  
constexpr std::size_t cmpIdx =  
    IndexOf<T, ComponentsList>();  
...  
}
```

```
template<typename T, typename ...Ts>  
inline T* addComponent(Ts&&...args) {  
  
    auto id = ecs::cmpIdx<T>();  
    ...  
}
```

```
template<typename T, typename ...Ts>  
inline T* addComponent(Ts&&...args) {  
  
    auto id = ecs::cmpIdx<T>;  
    ...  
}
```

Variable template, valido a partir de C++14

Manager

```
class Manager {  
    using uptr_ent = std::unique_ptr<Entity>;  
public:  
    Manager();  
    virtual ~Manager();  
  
    Entity* addEntity();  
    void refresh();  
    void update();  
    void render();  
  
private:  
    std::vector<uptr_ent> enteties_;
```

The diagram illustrates the Manager class with several annotations:

- An orange callout box with a black border contains the text "Tipo de unique_ptr de Entity". An orange arrow points from the word "unique_ptr" in the code's `using` declaration to this box.
- A dark red callout box contains the text "Añadir una entidad". A red arrow points from the `addEntity()` method signature to this box.
- A blue callout box contains the text "Borrar entidades no activas". A blue arrow points from the `refresh()` method signature to this box.
- A red callout box contains the text "Llamar a update/render de las entidades". Two red arrows point from the `update()` and `render()` method signatures to this box.
- A green callout box contains the text "Las entidades, se usa unique_ptr para que se borren automaticamente". A green arrow points from the `enteties_` member variable to this box.

EntityManager - addEntity

```
Entity* Manager::addEntity() {  
    Entity *e = new Entity(this); ← crear  
    if (e != nullptr)  
        enteties_.emplace_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
- ◆ Entity no recibe parámetros, más adelante los vamos a permitir

EntityManager - refresh

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

```
void Manager::refresh() {  
    enteties_.erase(  
        std::remove_if( ←  
            enteties_.begin(),  
            enteties_.end(),  
            [](const uptr_ent &e) {  
                return !e->isActive();  
            }),  
        enteties_.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 ...

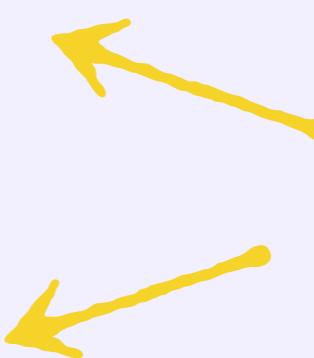
... después erase borra a partir de ese iterator hasta el final (el uso de remove_if es para evitar desplazamientos innecesarios)

EntityManager - update y render

Recorremos el vector 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 Manager::update() {  
    std::size_t n = entities_.size();  
    for (int i = 0; i < n; i++)  
        entities_[i]->update();  
}
```

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



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

Ejemplo bucle principal del juego

```
...  
while (!exit_) {  
    ...  
    manager_->update();  
    manager_->refresh();  
    manager_->render();  
    ...  
}  
...  
  
Actualizar entidades  
Borrar entidades no activas  
Renderizar entidades
```

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

Ejemplos de Componentes y Entidades

Transform

```
class Transform: public 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

BallPhysics

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

```
class BallPhysics: public Component {  
public:  
    BallPhysics() :  
        tr_(nullptr) {}  
    virtual ~BallPhysics();  
    void init() 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::init() {  
    tr_ = entity_->getComponent<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 ...  
}
```

Se puede mover esta línea al transform p.ej.

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

Image

```
class Image: public Component {  
public:  
    Image(Texture* tex) : //  
        tr_(nullptr), //  
        tex_(tex) {}  
    virtual ~Image() ...  
    void init() 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



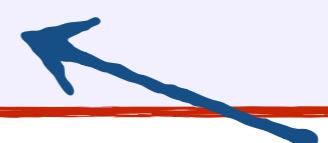
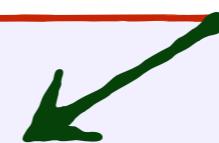
Textura (imagen) para renderizar

Image

```
void Image::init() {  
    tr_ = entity_->getComponent<Transform>();  
}
```

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

Obtener el Transform.



Renderizar la textura usando las características físicas de la entidad

Entidad para la Pelota (en PingPong)

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

Posición inicial de la pelota, etc.

Añadir una entidad

Añadir componentes

Pasa una textura para la pelota

```
graph TD; A[Añadir una entidad] --> B[addEntity()]; C[Añadir componentes] --> D[addComponent<Transform>(...)]; C --> E[addComponent<BallPhysics>()]; C --> F[addComponent<Image>(...)]; G[Pasa una textura para la pelota] --> H[Image]
```

Grupos y Handlers

Handlers

Necesitamos acceso desde un componente a los componentes de otras entidades ... lo más sencillo es pasar la referencia a la entidad/componente, pero se puede también proporcionar acceso global a través del manager ...

```
namespace ecs {  
    ...  
    enum hdlrId : std::size_t { BALL, ... ,_LAST_HDLRID_ };  
    constexpr std::size_t maxHdlr = _LAST_HDLRID_;  
    ...  
}  
class Manager {  
public:  
    ...  
    inline void setHandler(std::size_t id, Entity *e) { hdlrs_[id] = e; }  
    inline Entity* getHandler(std::size_t id) { return hdlrs_[id]; }  
  
private:  
    ...  
    std::array<Entity*, ecs::maxHdlr> hdlrs_  
}
```

Handlers

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



Declara a que entidad corresponde ecs::BALL

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



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

Handlers

Otra posibilidad es usar meta-programming y tipos de datos “ficticios”

```
struct BALL;
struct LEFT_PADDLE;
...
using HdIrls = TypeList<BALL, LEFT_PADDLE, ...>;
namespace ecs {
...
template<typename T>
constexpr std::size_t hdIrlsIdx = mpl::IndexOf<T, HdIrls>();
constexpr std::size_t maxHdIrls = HdIrls::size;
}
class Manager {
public:
...
template<typename T>
inline void setHandler(Entity *e) { hdlrs_[ecs::hdIrlsIdx<T>] = e; }
template<typename T>
inline void getHandler() { return hdlrs_[ecs::hdIrlsIdx<T>] }
}
...
```

Handlers

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

Declarar a que entidad corresponde BALL

Acceder a la entidad que corresponde a BALL desde un componente

```
auto ball = entity_->getMngr()->getHandler<BALL>();
```

Groups

Necesitamos hacer operaciones sobre conjuntos de entidades – p.ej., comprobar choques de todos los asteroides con el caza, dibujar todas la fantasmas y después todas la cerezas, ...

```
namespace ecs {
...
enum groupId : std::size_t { ASTEROID, ... ,_LAST_GROUPID_ };
constexpr std::size_t maxGroup = _LAST_GROUPID_;
}
class Entity {
public:
...
    inline bool hasGroup(std::size_t id) { return groups_[id]; }
    inline void setGroup(std::size_t id, bool state) { groups_[id] = state; }
    inline void resetGroups() { groups_.reset(); }

...
    std::bitset<Entity*, ecs::maxGroup> groups_;
}
class Manager {
...
    inline const auto& getEntities() { return entities_; }
...
}
```

Groups

```
for(...) {  
    Entity *a = manager_->addEntity();  
    ...  
    asteroid->addGroup(ecs::ASTEROID,a)  
    ...  
}
```

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

Groups

Se puede usar usar meta-programming y tipos de datos “ficticios”

```
struct ASTEROID;
struct FIGHTER;
...
using GroupList = TypeList<ASTEROID, FIGHTER, ...>;
namespace ecs {
...
template<typename T>
constexpr std::size_t grpIdx = mpl::IndexOf<T, GroupList >();
constexpr std::size_t maxGroup = GroupList::size;
...
class Entity {
public:
...
template<typename T>
inline bool hasGroup() { return groups_[ecs::grpIdx<T>]; }
...
}
```

Groups

```
for(...) {  
    Entity *a = manager_->addEntity();  
    ...  
    asteroid->addGroup<ASTEROID>(a)  
    ...  
}
```

```
for (auto &e : mngr_->getEnteties()) {  
    if (e->hasGroup<ASTEROID>()) {  
        ...  
    }  
}
```

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 render de Component y modificar la clase Entity para distinguir entre componentes de entrada, física, gráfica, o datos (sería más eficiente porque ahorraremos 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 la memoria -- todo se crea en addComponent y addEntity