

Patrones de Diseño

Gestión de Memoria Dinámica (para Entity-Component-System)

TPV2

Samir Genaim



Lecture Notes
Samir Genaim
TPV2
Facultad de Informática
Universidad Complutense de Madrid

UNIVERSIDAD
COMPLUTENSE
MADRID

En este tema está basado en el ECS que usa Template meta-programming para calcular, entre otras cosas, los índices de los componentes, etc. Se puede adaptar a cualquier otro ECS que hemos visto.

El Contexto

```
template<typename T = _grp_GENERAL>
inline Entity* addEntity() {
```

```
...
```

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

```
...
```

```
}
```

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

```
...
```

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

```
...
```

```
}
```

Cada vez que necesitamos una instancia de la clase **Entity** o de un componente **T**, la pedimos al sistema operativo usando **new**. En otros métodos, cuando dejamos de usar una instancia la devolvemos usando **delete**.

El Problema

- ♦ Llamar a **new** (e después **delete**) para cada entidad o componente, puede afectar negativamente al juego en términos de tiempo de ejecución:
- ✓ Muchas peticiones al sistema operativo para pedir y liberar memoria dinámica — puede ser muy costoso en algunas plataformas.
- ✓ Fragmentación de la memoria: los objetos viven en distintas partes de la memoria algo que incrementa el número de "**cache miss**".
- ✓ Siempre hay que construir los objetos (ejecutar la constructora).

Lo que queremos conseguir ...

- ★ Crear todas las entidades y los componentes que vamos a necesitar al principio del programa.
- ★ Además, queremos almacenar los objetos (o al menos los que tienen el mismo tipo) en una **memoria contigua** para minimizar "**cache misses**".
- ★ Cuando necesitamos una entidad usamos una que no está actualmente en uso, y cuando muere la marcamos como no usada para poder usarla de nuevo.
- ★ Cada entidad va a tener todos los componentes posibles ya creadas y sólo indicamos que componentes están activos (que componentes tiene).
- ★ Vamos a ver varios posibles soluciones.

Manager.h: Pool de Entidades

```
class Manager {  
...  
private:
```

Todas las entidades se crean (usando la constructora por defecto) cuando creamos el Manager. maxNumEnts está definido en ecs.h

```
...  
std::array<Entity, maxNumEnts> entsPool_;  
std::bitset<maxNumEnts> usedEnts_;  
std::size_t lastUsed_;
```

bitset para marcar las entidades usadas

Indice de la ultima entidad usada

```
inline Entity* allocEntity() {  
    auto i = lastUsed_ + 1 % maxNumEnts;  
    while (usedEnts_[aux] && aux != lastUsed_)  
        i = aux + 1 % maxNumEnts;  
    assert(!usedEnts_[i]);  
    usedEnts_[i] = true;  
    lastUsed_ = i;  
    return &entsPool_[i];  
}
```

Búsqueda circular de una entidad no usada, a partir del de lastUsed_

```
inline void freeEntity(Entity* e) {  
    usedEnts_[e-entsPool_] = false;  
}
```

Para liberar una entidad, la marcamos como no usada (el indice se calcula usando pointer arithmetic)

Entity.h

```
#include "../components/all_components.h"
```

Necesitamos los tipos completos, no es suficiente un forward declaration.

```
struct Entity {
```

```
    Entity() : cmps_(), activeCmps_(), alive_(), gId_(0)
```

```
{  
}
```

```
virtual ~Entity() {  
}
```

La clase Entity lleva todos los componentes como `std::tuple`. Las instancias se crean cuando se crea una entidad. Es decir, cuando se crea el manager. Ver la definición de `tuple_of_cmps_t` en la siguiente pagina.

```
    tuple_cmps_t cmps_;
```

```
    std::bitset<maxComponentId> activeCmps_;
```

```
    bool alive_;
```

```
    ecs::grpId_type gId_;
```

bitset para marcar los componentes activos

```
};
```

La definición de tuple_cmps_t

```
template<typename>  
struct PoolHelper;
```

```
template<typename ...Ts>  
struct PoolHelper<mpl::TypeList<Ts...>> {  
    using tuple_t = std::tuple<Ts...>;  
};
```

```
using tuple_cmps_t = PoolHelper<CmpsList>::tuple_t;
```



std::tuple de los distintos componentes. Todos los componentes tienen que tener constructora por defecto.

Crear una Entidad

```
template<typename T = _grp_GENERAL>
```

```
inline Entity* addEntity() {
```

```
    constexpr auto gId = grpId<T>;
```

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

Pedir una entidad al Pool

```
    e->cmps_.reset();
```

```
    e->gId_ = gId;
```

```
    e->alive_ = true;
```

Inicializar

```
    entsByGroup_[gId].push_back(eIdx);
```

```
    return e;
```

Añadir a la lista correspondiente

```
}
```

Liberar una Entidad

```
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(),  
                [this](Entity *e) {  
                    if (isAlive(e)) {  
                        return false;  
                    } else {  
                        freeEntity(e);  
                        return true;  
                    }  
                }  
            ),  
            grpEnts.end());  
    }  
}
```

Devolver la entidad al Pool

add/remove Component

```
template<typename T>
inline T* addComponent(Entity *e) {
    constexpr cmpId_type cId = ecs::cmpId<T>;
    T *c = std::get<cId>(e->activeCmps_);
    c->initComponent();
    e->activeCmps_[cId] = true;

    return c;
}
```

No recibe argumentos como antes, porque los componentes ya están creados. Hay que hacer la inicialización fuera.

Ya tenemos el componente creado, sólo accedemos al elemento correspondiente del std::tuple

```
template<typename T>
inline void removeComponent(Entity *e) {
    constexpr cmpId_type cId = ecs::cmpId<T>;
    e->cmps_[cId] = false;
}
```

Sólo marcamos el componente como no activo

get/has Component

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

Devuelve el bit que corresponde a T del bitset correspondiente

```
template<typename T>
inline T* GetComponent(Entity *e) {
    constexpr cmpId_type cId = ecs::cmpId<T>;
    return std::get<cId>(e->activeCmps_);
}
```

Devuelve el componente en la posición correspondiente del std::tuple

Resumen Version 1


- ★ Los componentes tienen que tener constructora por defecto.
- ★ Las entidades y los componentes están en **memoria contigua**
- ★ Entity.h tiene que hacer **#include** de todos los componentes, esto puede afectar al tiempo de compilación (todos los que usan Entity.h van a incluir todos los componentes). Resolvemos este problema en las siguientes versiones.
- ★ Hay que asegurarse que `std::get` hace acceso directo a los elementos correspondientes, en algunas implementaciones hay que compilar con la opción **-O2** o **-O3** para conseguir esto.

std::tuple de arrays de components

```
template<typename>
struct PoolHelper;

template<typename ...Ts>
struct PoolHelper<mpl::TypeList<Ts...>> {
    using tuple_t = std::tuple<Ts...>;
    using tuple_arr_t = std::tuple<Ts[maxNumEnts]...>;
};

using tuple_cmps_t = PoolHelper<CmpsList>::tuple_t;
using tuple_arr_cmps_t = PoolHelper<CmpsList>::tuple_arr_t;
```



std::tuple de arrays de componentes. La idea es que la entidad con el índice 'i' usa los componentes en la posición 'i' de cada array. Todos los componentes tienen que tener constructora por defecto.

Entity.h

```
struct Entity {  
    Entity() : cmps_(), activeCmps_(), alive_(), gId_(0)  
    {  
    }  
  
    virtual ~Entity() {  
    }  
  
    std::bitset<maxComponentId> activeCmps_;  
    bool alive_;  
    ecs::grpId_type gId_;  
};
```

Lleva solo el bitset para marcar los componentes activos. No hace falta hacer `#include` de todos los componentes.

addEntity es igual al anterior.

Manager.h: Pool de Componentes

```
#include "../components/all_components.h"
```

```
class Manager {
```

```
...  
private:
```

```
...  
tuple_arr_cmps_t cmpsPool_;
```

```
template<typename T>
```

```
inline bool GetComponentFromPool(Entity *e) {
```

```
    constexpr cmpId_type cId = ecs::cmpId<T>;
```

```
    auto poolT = std::get<cId>(cmpsPool_);
```

```
    return poolT[e-entsPool_];
```

```
}
```

```
...
```

```
}
```

Necesitamos los tipos completos, no es suficiente un forward declaration.

La clase Manager lleva los arrays de los componentes.

Devuelve un puntero al componente T de la entidad 'e' (el índice de la entidad se calcula usando pointer arithmetic)

add/remove Component

```
template<typename T>
inline T* addComponent(Entity *e) {
    constexpr cmpId_type cId = ecs::cmpId<T>;
    T *c = GetComponentFromPool<T>(e);
    c->initComponent();
    e->activeCmps_[cId] = true;

    return c;
}
```

Ya tenemos el componente creado, se lo pedimos al pool de componentes.

```
template<typename T>
inline void removeComponent(Entity *e) {
    constexpr cmpId_type cId = ecs::cmpId<T>;
    e->cmps_[cId] = false;
}
```

Sólo marcamos el componente como no activo

get/has Component

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

Devuelve el bit que corresponde a T del bitset correspondiente

```
template<typename T>
inline bool GetComponent(Entity *e) {
    constexpr cmpId_type cId = ecs::cmpId<T>;
    return GetComponentFromPool<T>(e);
}
```

Pide el componente correspondiente al pool de componentes

Resumen Version 2

- ★ Los componentes tienen que tener constructora por defecto.
- ★ La clase Entity no lleva los componentes.
- ★ Todas las entidades están en **memoria contigua**
- ★ Todos los componentes están en **memoria contigua**
- ★ Manager.h tiene que hacer **#include** de todos los componentes, esto puede afectar al tiempo de compilación (todos los que usan Manager.h van a incluir todos los componentes).
- ★ En la siguiente versión, en lugar de usar tuple de arrays, vamos a usar tuple de arrays dinámicos para evitar la inclusion de todos los componentes.

std::tuple de punteros a arrays

```
template<typename ...Ts>
struct PoolHelper<mpl::TypeList<Ts...>> {
    ...
    using tuple_parr_t = std::tuple<Ts*...>;

    void alloc(tuple_parr_t &t) {
        ((std::get<cmpId<Ts>>(t) = new Ts[maxNumEnts]),...);
    }
    void free(tuple_parr_t &t) {
        ((delete [] std::get<cmpId<Ts>>(t)),...);
    }
};
...
using tuple_parr_cmps_t = PoolHelper<CmpsList>::tuple_parr_t;
```

std::tuple de punteros a arrays de componentes

Crear todos los array de un tuple_parr_t

Liberar la memoria de todos los arrays de un tuple_parr_t

Manager.h

```
class Manager {
```

```
...  
private:
```

Como antes, sólo usamos el nuevo tipo. No hace falta incluir hacer #include de todos los componentes en Manager.h

```
...  
tuple_parr_cmps_t cmpsPool_;
```

```
template<typename T>
```

```
inline bool GetComponentFromPool(Entity *e) {
```

```
    constexpr cmpId_type cId = ecs::cmpId<T>;
```

```
    auto eIdx = e->entsPool_;
```

```
    auto poolT = std::get<cId>(cmpsPool_);
```

```
    return poolT[eIdx];
```

```
}
```

```
...
```

```
}
```

En **Manager.cpp**, incluimos todos los componentes y:

1. Llamamos a `PoolHelper<CmpsList>::init(cmpsPool_)` en la constructora
2. Llamamos a `PoolHelper<CmpsList>::free(cmpsPool_)` en la destructora.

Resumen Version 3

- ★ Los componentes tienen que tener constructora por defecto.
- ★ La clase Entity no lleva los componentes.
- ★ Todas las entidades están en **memoria contigua**
- ★ Componentes del mismo tipo están en **memoria contigua**
- ★ Recordar que hay que asegurarse que `std::get` hace acceso directo a los elementos correspondientes, en algunas implementaciones hay que compilar con la opción **-O2** o **-O3** para conseguir esto.

Ejercicio

- ★ Los usuarios de la clase Manager nunca acceden directamente a la clase Entity. Tienen métodos para todo lo que necesitan en la clase Manager.
- ★ Cada entidad se puede identificar con una instancia de Entity o con su índice (un número) en el pool correspondiente.
- ★ Cambia el diseño para que los métodos del Manager devuelvan al usuario el índice de la entidad en lugar de un puntero a la instancia correspondiente.