

Arquitectura de videojuegos

Máster en Programación de videojuegos



Javier Arévalo

2º Trimestre
Curso 2013-2014

Arquitectura de videojuegos

Game Objects



Game Objects / Entidades

- Elementos de juego con comportamiento complejo y/o autónomo
- Jugador principal, enemigos
- Props, balas, explosiones, plataformas
- Elementos de lógica ocultos
 - Zonas y puertas lógicas
 - 'Comandantes' y jefes de grupo de otras entidades

Game Objects / Entidades

- Necesidades comunes
 - Creación y destrucción
 - Ejecución de lógica, input y render
 - Referencias mutuas: apuntar al objetivo, etc.
 - Interacción: colisiones, ordenes
 - Jerarquía: orden de ejecución, objetos enganchados a otros

Game Objects / Entidades

- Clases y objetos que representan esas entidades
- Un gestor de entidades que las contiene, gestiona y les da las ordenes adecuadas
- Manejar mecanismos para optimizar la ejecución del juego
 - No calcular colisiones para objetos que no colisionan
 - No ejecutar lógica de objetos 'dormidos'

Evolución

- Objetos ad-hoc
- Objetos generalizados
- Jerarquía de clases
- Composición de comportamientos
- Paralelización y orientación a datos

Objetos ad-hoc

```
class Enemy
{
    Vector2 m_position;
    Vector2 m_speed;
    float    m_shootDelay;
public:
    void Update(float elapsed);
    void Render(float elapsed);
};
```

```
class EnemyBullet
{
    Vector2 m_position;
    Vector2 m_speed;
    float    m_damage;
public:
    void Update(float elapsed);
    void Render(float elapsed);
};
```

```
class Player
{
    Vector2 m_position;
    float    m_life;
    float    m_speed;
public:
    void Input(float elapsed);
    void Update(float elapsed);
    void Render(float elapsed);
};
```


Objetos ad-hoc

```

void Game::Input(elapsed) {
    m_Player->Input(elapsed);
}

void Game::Update(elapsed) {
    m_Player->Update(elapsed);
    for (auto enemyIt = m_Enemies.begin();
        enemyIt != m_Enemies.end();
        ++enemyIt) {
        (*enemyIt)->Update(elapsed);
    }
    for (auto bulletIt = m_EnemyBullets.begin();
        bulletIt != m_EnemyBullets.end();
        ++bulletIt) {
        (*bulletIt)->Update(elapsed);
    }
    CheckCollisions(m_Player, m_EnemyBullets);
}

```

```


class Game
{
    Player* m_Player;
    vector<Enemy*> m_Enemies;
    vector<EnemyBullet*> m_EnemyBullets;
public:
    void Input(float elapsed);
    void Update(float elapsed);
    void Render(float elapsed);
};

```

Objetos ad-hoc

- Sencillo de hacer y de entender
- Manejable para un conjunto pequeño de tipos de objetos
- Multitud de cosas repetidas en cuanto incluimos mas tipos de objetos:
 - Impactos, disparos propios y ajenos, tipos de enemigos diferentes, bosses, powerups, etc.
 - Explosión combinatoria de interacciones

Objetos generalizados

- Ya en tiempos de los 8-bits era un problema
- Idea: 
 - Muchos objetos comparten bastantes datos
 - Posición, velocidad, vida, representación gráfica, etc.
 - Crear una clase genérica con todos esos datos
 - Usar algún tipo de sistema para seleccionar comportamientos
 - Y marcas de tipo para segregar objetos durante las interacciones

Objetos generalizados

```
class Entity
{
    int      m_type;
    Vector2  m_position;
    Vector2  m_speed;
    float    m_life;
    float    m_damage;
    float    m_shootDelay;
    Vector2  m_size;
public:
    void Input(float elapsed);
    void Update(float elapsed);
    void Render(float elapsed);
};
```

```
class Game
{
    Player* m_Player;
    vector<Entity*> m_Entities;
public:
    void Input(float elapsed);
    void Update(float elapsed);
    void Render(float elapsed);
};
```

```
void Entity::Update(elapsed)
{
    switch (m_type)
    {
        case ENTITY_PLAYER:
            UpdatePlayer(elapsed);
            break;
        case ENTITY_ENEMY:
            UpdateEnemy(elapsed);
            break;
        case ENTITY_PLAYERBULLET:
        case ENTITY_ENEMYBULLET:
            UpdateBullet(elapsed);
            break;
        // .. etc
    }
}
```

```
void Game::Update(elapsed) {
    for (auto entityIt = m_Entities.begin();
         entityIt != m_Entities.end();
         ++entityIt) {
        (*entityIt)->Update(elapsed);
    }
}
```

Objetos generalizados

- Algunas partes de la lógica quedan unificadas
 - Creación y destrucción
 - Lanzar todos los Update() y Render()
 - Interacción entre objetos
- Hemos dividido el problema y tenemos una forma sencilla de implementar diferentes partes
 - Algo de la explosión combinatoria se mantiene

Objetos generalizados

- Donde hay un `switch/case`, hay una oportunidad para *dynamic dispatch*
- Antes de C++, usábamos punteros a funciones directamente
 - Configurados en la creación del objeto
 - Dinámicos: podemos cambiarlos sobre la marcha para construir una pequeña maquina de estados

Objetos generalizados

- Muestra: la 'thing' de Speed Haste

```
typedef struct THN_SThing {
    uint32 type;                // Type of thing as it was created.
    uint32 magic;               // Magic to avoid repeating calculus.
    uint32 x, y, z;             // Map position.
    word angle;                 // Angle it's looking at.
    byte flags;                 // ....
    FS3_PSprite spr;            // Graphics data.
    THN_PBounds bounds;         // Collision tracking.
    void (*routine)(void *data); // Action routine.
    void *data;                 // Pointer passed to routine();
    THN_TLink block;            // Linked list for each map tile.
    THN_TLink global;           // Global linked list for all things.
    THN_TLink sector;           // Linked list for the sector.
    struct SEC_SSector *sec;     // Sector this thing is in.
    word tirerot;                // Angle of tires (if a car).
    word tiredir;                // Direction of tires (if a car).
    sint32 fill[4];
} THN_TThing, *THN_PThing;
```

Jerarquía de clases

- Hasta ahora todo esta orientado a lenguajes imperativos clasicos: asm, C, Pascal.
- Hemos usado técnicas para tratar homogéneamente objetos distintos en naturaleza
- Con C++ podemos usar mecanismos propios del lenguaje para gestionar polimorfismo
 - Y todos nos tiramos a la piscina de cabeza

Jerarquía de clases

```
class Entity; // Abstracta

// Entities
class E_Mobile: public Entity;
class E_Prop: public Entity;

// Mobiles
class E_Player: public E_Mobile;
class E_Enemy: public E_Mobile;
class E_Ally: public E_Mobile;
class E_Enemy: public E_Mobile;

// Props
class E_Breakable: public E_Prop;

// Enemies
class E_Soldier: public E_Enemy;
class E_Tank: public E_Enemy;
class E_PanzerTank: public E_Tank;
```

Jerarquía de clases

```
class Entity
{
    Vector3 m_position;
public:
    virtual void Update(elapsed);
    virtual void Render(elapsed);
};

class E_Mobile: public Entity
{
    Vector3 m_speed;
public:
    virtual void Update(elapsed);
};

class E_Player: public E_Mobile
{
    float m_life;
public:
    void Input(elapsed);
    virtual void Update(elapsed);
    virtual void Render(elapsed);
};
```

Jerarquía de clases

- Hemos ganado en:
 - Expresividad: las clases se llaman como los tipos de objeto que representan
 - Seguridad: las clases no tienen métodos que no tienen sentido para ellas, p.ej. Input() pertenece al Player.
 - Optimización:
 - los objetos no tienen apenas miembros que no les sean relevantes
 - Objetos del mismo tipo no tienen copias de los punteros a funciones – los métodos virtuales se ocupan de todo

Jerarquía de clases

```
// Mobiles
class E_Player: public E_Mobile;
class E_Enemy: public E_Mobile;
class E_Ally: public E_Mobile;
class E_Enemy: public E_Mobile;

// Props
class E_Breakable: public E_Prop;

// Enemies
class E_Soldier: public E_Enemy;
class E_Tank: public E_Enemy;
class E_PanzerTank: public E_Tank;

// Hm, what?
class E_ShermanTank: public E_Tank, public E_Ally;
```

- El horror de las jerarquías de clases

Jerarquías de clases

- Los problemas ya lo habéis visto en Ingeniería del Software:
 - Diamantes de la muerte
 - Inflexibilidad
 - Definidas en el código
 - Crece proporcionalmente a la cantidad de tipos de objetos en el diseño
- Deriva de pensar en el problema en vez de en la solución – CBruce!

Composición

- Se populariza desde el 2000 mas o menos
- Vuelve al concepto de una clase 'Entity' genérica pero concreta
- Contiene mecanismos generales de gestión
 - Creación, destrucción, etc
- Y contiene componentes creados, asignados y configurados en tiempo de ejecución

Composición

```
class Component
{
    Entity* m_Owner;
public:
    virtual void Update(elapsed) = 0;
};

class Entity
{
    vector<Component*> m_Components;
public:
    virtual void Update(elapsed);
    void AddComponent(Component*);
};

void Entity::Update(elapsed)
{
    for (auto compIt = m_Components.begin();
         compIt != m_Components.end();
         ++compIt) {
        (*compIt)->Update(elapsed);
    }
}
```

Composición

```

class C_Renderable: public Component;
class C_Life: public Component;
class C_InertialMovement: public Component;
class C_RigidBody: public Component;

class C_Damage: public Component;
class C_WeaponHolder: public Component;

class C_Perception: public Component;

class C_PlayerControl: public Component;
class C_EnemyAI: public Component;
class C_PartnerAI: public Component;

```

Player:

Renderable, Life, Damage, InertialMovement,
WeaponHolder, RigidBody, PlayerControl

Enemy:

Renderable, Life, Damage, InertialMovement,
WeaponHolder, RigidBody, EnemyAI, Perception

Composición

```
class C_Renderable: public Component;
class C_Life: public Component;
class C_InertialMovement: public Component;
class C_RigidBody: public Component;

class C_Damage: public Component;
class C_WeaponHolder: public Component;

class C_Perception: public Component;

class C_PlayerControl: public Component;
class C_EnemyAI: public Component;
class C_PartnerAI: public Component;
```

Turret:

Life, Damage, WeaponHolder, EnemyAI,
Perception

Prop:

Renderable, Life, RigidBody

Explosion:

Renderable, Life, Damage

Trigger:

Perception, EnemyAI

Composición

- Es muy fácil construir las entidades en base a descripciones en ficheros de datos
 - Definiciones, Arquetipos, MetaObjetos, ...
- Esos archivos pueden ser editables por diseñadores
 - Nuevos tipos de entidades sin recompilar y sin molestar a los programadores :)
- Ya que lo hacemos, que sea parametrizable

Composición

- Problema: comunicación entre componentes dentro de una entidad
 - El Movement genera impulsos al Rigidbody
 - El Damage produce una perdida de Life
 - La EnemyAI recibe objetivos de la Perception
 - Todos quieren tocar al Renderable
- Los componentes ya no tienen un interfaz que exponga funcionalidades propias de su tipo

Composición

- Soluciones:
- Búsqueda de componentes de tipo concreto
 - QueryInterface / FindComponent
- Comunicación por paso de mensajes
 - SendMessage()

Composición

- FindComponent

```
template<class T>
T* Entity::FindComponent()
{
    for (auto compIt = m_Components.begin();
         compIt != m_Components.end();
         ++compIt)
    {
        T* comp = dynamic_cast<T*>(*compIt);
        if (comp)
            return comp;
    }
    return NULL;
}

C_Life *compLife = m_Owner->FindComponent<C_Life>();
if (compLife)
{
    compLife->ReduceLife(damage);
    ...
}
```

Composición

- Ventajas:
 - Llamadas directas – eficiente
 - Llamadas a funciones declaradas – seguro
- Desventajas:
 - Mayor acoplamiento: necesario conocer que componentes necesitan ser llamados
 - Mas farragoso hacer multicast a varios componentes
 - Comprobación antes de cada llamada

Composición

- SendMessage

```
struct Message
{
};

struct ReduceLifeMessage: public Message
{
    float amount;
};

void Entity::ReceiveMessage(Message *msg)
{
    for (auto compIt = m_Components.begin();
         compIt != m_Components.end();
         ++compIt)
        (*compIt)->ReceiveMessage(msg);
}

void LifeComponent::ReceiveMessage(msg)
{
    ReduceLifeMessage *rlmsg = dynamic_cast<ReduceLifeMessage*>(msg);
    if (rlmsg)
        m_life -= rlmsg->amount;
}
```

Composición

- Retorno de valores

```
struct ReduceLifeMessage: public Message
{
    float amount;
    bool dead; // out
};

void LifeComponent::ReceiveMessage(msg)
{
    ReduceLifeMessage *rlmsg = dynamic_cast<ReduceLifeMessage*>(msg);
    if (rlmsg)
    {
        m_life -= rlmsg->amount;
        rlmsg->dead = (m_life <= 0);
    }
}
```


Composición

- Ventajas:
 - Acoplamiento mínimo: cada componente sabe a que mensajes debe responder
 - Multicast: varios componentes pueden responder al mismo mensaje
 - Buena base para soportar paralelismo (PostMessage).
- Desventajas:
 - Poco eficiente: todos los mensajes procesados por todos los componentes
 - Poco seguro: RTTI normalmente gestionado 'a mano' por eficiencia
 - Crear una struct por cada tipo de llamada
 - Valor de retorno de múltiples componentes?

Paralelización

- Aprovechar múltiples núcleos / hilos de ejecución

	Comp A	Comp B	Comp C	Comp D
Entity 1				
Entity 2				
Entity 3				
Entity 4				

Paralelizacion

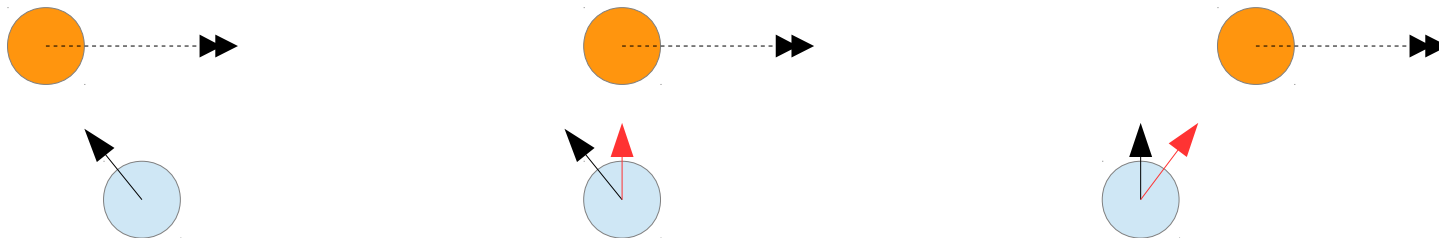
- En cada hilo de ejecución podemos ejecutar:
- Primer plan: horizontal
 - Update() de todos los Componentes de una Entity
 - Problema: interacción entre Entities
- Segundo plan: vertical
 - Update() de cada tipo de Componente para todas las Entities que tienen uno
 - Problema: rompemos la ejecución atómica de cada Entity

Orientación a Datos

- El segundo plan es mucho mas complejo
 - Ya no pensamos en una Entidad como compuesta de varios Componentes
 - Pensamos en una Entidad como 'cliente' de varios sistemas de Componentes
 - Diseño de cada tipo de Componente pensando en sus relaciones con los demás
 - Orden estricto de ejecución de cada tipo de Componente
 - Definición estricta de que Componentes pueden hablar con que otros

Otros asuntos generales

- Soportar entidades asociadas a otras
 - Define el orden en que se ejecutan
 - Si un arma esta en un brazo, debe moverse después de que el brazo lo haga
 - Esta necesidad aparece en otros casos, p.ej. Una entidad apunta a otra



Otros asuntos generales

- Referencias entre entidades:
 - Capacidad de consultar el mundo para encontrar entidades: enemigos, objetivos, puntos estratégicos, etc
 - Que es esa referencia? Un puntero, un 'handle'?
 - Cuando una entidad va a desaparecer, todas las referencias a ella deben eliminarse
 - Los comportamientos que mantienen referencias a entidades tienen que comprobar su validez

Otros asuntos generales

- Ciclo de vida de una entidad:
 - Creacion: es construida e inicializada
 - Activacion: insertada en el 'mundo'
 - Ejecución normal
 - Desactivacion: eliminada del 'mundo'
 - Destrucción
- La creación de una entidad es costosa
 - Podemos reciclar Componentes o Entidades completas?

Otros asuntos generales

- Problema de 'el frame siguiente':
 - Al crear una entidad, no debería activarse e insertarse en el mundo hasta haber terminado la ejecución de todas las actuales
 - O creamos condiciones de carrera
 - Pero el creador si puede necesitar una referencia, p.ej. crear una cámara voladora para investigar

Otros asuntos generales

- Fases de ejecución:
 - Muchos de estos problemas se pueden abordar dividiendo el Update() en múltiples fases
 - Definiendo que cosas se pueden realizar en cada fase
 - Creación de entidades, búsqueda de entidades, alteración de otra entidades, etc
 - Introduce un lugar mas de burocracia en el código, con la consiguiente posibilidad de bugs y perdidas de eficiencia

Vuestro turno!

Preguntas?

