Comportamiento

- Forma en que se especifica las acciones que realizan los NPCs
- ¿Cuáles?
  - Que acción llevar a cabo
- ¿Cuándo?
  - En que momento empezar a hacerla
- ¿Cómo?
  - Qué pasos seguir para realizarla

Implementación Acción

- Animación
  - Una pieza clave para comunicar acciones al jugador
  - Casi cualquier acción llevará una o varias animaciones asociadas
  - No confundir comportamiento con animación
- Movimiento
  - Coordinación entre steerings y animaciones de movimiento
  - Evitar moverse resbalando
  - Evitar caminar en el sitio
  - Pathfinding

Elección Acción

- Dos ramas principales
- Escriptado
  - Desde diseño
  - Siempre igual
- Emergente
  - Capacidad de decisión
  - Autonomía
  - Varía según distintos factores: jugador, entorno, etc.
- No es blanco o negro, muchos niveles de gris

Escriptado

- Igual que en el cine o teatro
- El NPC sigue un guión preestablecido
- No se adaptan a la situación
  - Sólo funciona si el comportamiento no depende de otros factores
  - O si se dan las condiciones para poder aplicarse
- Tiene su aplicación en los juegos modernos
  - Cinemáticas in-game
  - Misiones de escolta
  - Personajes de ambiente con los que no se interactúa

Escriptado

- Diseñador tiene control 100%
- No se van a producir comportamientos no previstos
  - Aunque sí comportamientos no adecuados para la situación

Escriptado

- Muy díficil de contemplar todas las posibilidades
- Sólo funciona en los casos que hemos previsto
- Para comportamientos sencillos es buena opción
- Se complica muy rápidamente
- Díficil visualizar qué está haciendo el NPC
  - Depuración de problemas
- Fácil que se convierta en algo muy técnico

Emergente

- Da cierta libertad al NPC
- Teatro de improvisación
  - Directrices de comportamiento
  - Libertad para adaptarse a situaciones no previstas
- Se definen ciertas acciones o comportamientos
- Se usan o combinan según la situación

Emergente

- Es muy dificil o imposible hacer algo totalmente emergente
- El diseñador pierde control
- El NPC no tiene porqué hacer exactamente lo que el diseñador pensaba que era lo más adecuado
- Complejo ajustarlo a las necesidades del diseño
  - Me gustaría que si ocurre esta situación haga esta acción
  - Pero si ocurre esta otra no
  - …

Emergente

- No es necesario contemplar todos los casos
- Comportamiento sólido
  - En respuesta a una acción del jugador reacciona el NPC
  - Tal vez no sea la acción más dramática
  - Pero es una acción razonable
- Facilidad de reuso en distintos niveles
  - Incluso juegos

Escriptado - Emergente

- Mucho terreno gris
- Es un continuo
- No son dos opciones
- Un NPC quedará en una situación intermedia
  - Ni 100% escriptado
  - Ni 100% emergente

Escriptado vs Scripts

- Un script es sólo una opción para comportamientos escriptados
- Aunque la forma más sencilla es mediante scripts

Scripts

- El comportamiento se define como unas líneas de lua
- El personaje se crea en el mundo
- Comienza a ejecutar su script
- Continúa su script hasta que termina o muere

Scripts

- Un código de lua no tiene que ser super rígido
  - Puede adaptarse
  - Condiciones para realizar unas acciones u otras
- No tiene porque ser una secuencia de animaciones
  - Como sería hacer una cinemática offline
- No tiene porque ser una lista de acciones seguidas
  - Primero ve a este punto
  - Luego abre el cofre
  - Luego coge una espada
  - Luego ve a la mesa
  - Luego deja la espada
  - ...

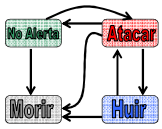
Scripts: Ejemplo

```
local maxChaseDistance = 20
local origPos = {}
local wasSpawned = NOSPAWN
local lastAttack = 0
local followTimeout = 10

local function isSkilled(cid)
    local skillType = getPlayerSkillType(cid)
    if skillType >= 0 then
        return true
    end

    return false
end

local function goToOrigPos()
    target = 0
    lastAttack = 0
    selfFollow(0)
    doTargetCombatHealth(getNpcCid(), origPos)
end

local function updateTarget()
    if(isPlayer(target) == false) then
        goToOrigPos()
    elseif(not isSkilled(target)) then
        target = 0
        selfSay("Now, beware in the future.")
    end

    if(target == 0) then
        local list = getSpectators(getNpcPos(), 8, 8, false)
        for i=1, table.getn(list) do
            if(isPlayer(list[i])) then
                if(isSkilled(list[i]) and isSkilled(_target)) then
                    if(selfFollow(list[i])) then
                        target = list[i]
                        if(target ~= preyTarget) then
                            selfSay("Hi is no suicide people like you here!")
                        end

                        preyTarget = target
                        break
                    end
                end
            end
        end
    end
end
```

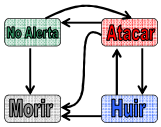
```
function onCreatureDisappear(cid)
    if(target == cid) then
        goToOrigPos()
    end
end

function onCreatureMove(creature, oldPos, newPos)

end

function onThink()
    updateTarget()

    if(target == 0) then
        return
    end

    local playerPos = getCreaturePosition(target)
    local myPos = getMyPos()

    if(myPos.z ~= playerPos.z) then
        goToOrigPos()
        return
    end

    if(math.abs(myPos.x - origPos.x) > maxChaseDistance or math.abs(myPos.y - origPos.y) > maxChaseDistance) then
        selfSay("I'll catch you next time.")
        goToOrigPos()
        return
    end

    if(lastAttack == 0) then
        lastAttack = os.clock()
    end

    if(os.clock() - lastAttack > followTimeout) then
        --To prevent hopping the npc by going to a place where he can't reach
        selfSay("See you on this side, but just here.")
        goToOrigPos()
        return
    end

    if(math.abs(playerPos.x - myPos.x) <= 1) and (math.abs(playerPos.y - myPos.y) <= 1) then
        doTargetCombatHealth(getNpcCid(), target, COMBAT_LIFEDRAIN, -200, -300, CONST_ME_BLOODHIT)
        lastAttack = os.clock()
    end
end
```

Máquinas de Estado (SM)

- Se basan en:
- Definir estados en los que puede estar el NPC
- Definir cómo se pasa de un estado a otro
  - Transiciones
- Un NPC esta siempre en un estado
  - Y sólo un estado



SM: Ejemplo



SM: Estados

- Situación en la que se encuentra el NPC en un momento dado
- Se suele empezar una acción al entrar en el estado
- Se continúa con esa acción hasta salir del estado
- Por ejemplo:
  - Al entrar al estado de no alerta → Poner animación de "mirar alrededor"
  - Continuar con esta animación (o sus variantes)
  - Parar la animación al salir del estado

SM: Transiciones

- Parte de un estado y acaba en otro
- Constan de una serie de condiciones
- Examinan:
  - El entorno: mundo, otros NPCs, personajes
  - El estado interno: Variables del NPC
- Se cumplen condiciones de la transición → Cambio de estado

SM: Proceso

- Incializar máquina de estados con estado inicial
- Ejecutar acciones del estado inicial
  - Ejemplo: Reproducir una animación
- Evaluar sus transiciones en orden
- Si se cumple una
  - Actualizar estado actual al destino de la transicion
  - Ejecutar acción del nuevo estado

SM: Extensiones

- Es posible añadir acciones para cada evento de la SM
- Entrada en un estado
  - Queremos reproducir un sonido cuando se entra al estado
  - No queremos hacerlo en cada iteración dentro del estado
- Salida de un estado
  - Queremos eliminar un efecto
- Usar una transición
  - Poner un diálogo u otro dependiendo de la condición que ha disparado
  - "¡Se está cubriendo detrás de las cajas!"
  - "¡Está huyendo!"
  - Tal vez llevan al estado "Persecución", pero con condiciones distintas

Efectos en estado

Efectos en estado

SM: Extensiones

- Es posible añadir acciones para cada evento de la SM
- Entrada en un estado
  - Queremos reproducir un sonido cuando se entra al estado
  - No queremos hacerlo en cada iteración dentro del estado
- Salida de un estado
  - Queremos eliminar un efecto
- Usar una transición
  - Poner un diálogo u otro dependiendo de la condición que ha disparado
  - "¡Se está cubriendo detrás de las cajas!"
  - "¡Está huyendo!"
  - Tal vez llevan al estado "Persecución", pero con condiciones distintas

SM: Implementación

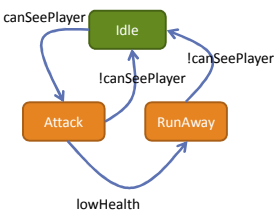
- Multitud de formas de implementación
- No existe **la forma** de implementar máquinas de estados
- Dependiendo de las necesidades
  - Más flexible
  - Más sencillo
- Dependiendo de los usuarios
  - ¿Diseñadores van a desarrollar máquinas de estados?
  - ¿Tarea del programador de C++?

SM: Simple

- Estados: enums + procedimientos
- Transiciones: funciones
- Máquina de estados: switch + condiciones if

```
class SM
{
    enum States
    {
        eIdle,
        eAttack,
        eRunAway
    };
...
```


SM: Simple




```
class SM
{
...
    // States
    void idle();
    void attack();
    void runaway();

    // transiciones
    bool canSeePlayer() const;
    bool lowHealth() const;
...
```

SM: Simple

```
class SM
{
    enum States
    {
        eIdle,
        eAttack,
        eRunAway
    };
...
```

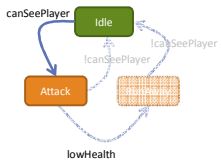
```
// States
void idle();
void attack();
void runaway();

// transiciones
bool canSeePlayer() const;
bool lowHealth() const;
...
```

```
...
public:
    SM();
    void update();
private:
    States m_currentState;
};
```


SM: Simple

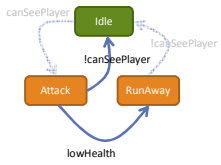
```
void SM::update()
{
    switch(m_currentState)
    {
        case eIdle:
        {
            break;
        }
        case eAttack:
        {
            break;
        }
        case eRunAway:
        {
            break;
        }
    }
}
```


SM:Simple



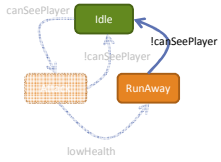

```
case eIdle:
{
    PlayAnimation("idle");
    if (canSeePlayer())
    {
        m_currentState = eAttack;
    }
    break;
}
```

SM:Simple




```
case eAttack:
{
   PlayAnimation("shoot");
   if (lowHealth())
   {
      m_currentState = eRunAway;
   }
   else if (!canSeePlayer())
   {
      m_currentState = eIdle;
   }
   break;
}
```


SM:Simple




```
case eRunAway:
{
   PlayAnimation("run");
   if (!canSeePlayer())
   {
      m_currentState = eIdle;
   }
   break;
}
```


SM: Flexible

- Encapsular en clases
- La lógica no está en el código sino en los datos
- Posible cambiar lógica sin recompilar
- Posible cambiar lógica mediante un editor
- Posible cambiar lógica por diseñadores
- Más complejo que versión simple programada

SM: Flexible

- Clase estado
  - Lista de acciones
  - Lista de transiciones
- Clase transición
  - Estado destino
  - Condición de disparo
- Clase acción
  - Clase abstacta
  - Reproducción de animaciones
  - Reproducción de sonido
  - Creación de efectos
  - etc.

SM: Flexible

```
class State
{
   Action* m_enterAction;
   Action* m_exitAction;
   Action* m_stateAction;
   Transitions m_transitions;
public:
   void onEnter();
   void update();
   void onExit();
   const Transitions& getTransitions();
};
```


SM: Flexible

```
class Transition
{
   Condition* m_condition;
   State* m_targetState;
   Action* m_triggerAction;
public:
   bool canTrigger() const;
   State* trigger();
};
```

SM: Flexible

```
class Condition
{
public:
    bool check() const = 0;
};
```

```
class CanSeePlayer: public
Condition
{
public:
    bool check() const
    { /* raycast to player pos*/ }
```

```
class AndCondition: public Condition
{
public:
    AndCondition(Condition* c1, Condition* c2);
    bool check() const { return m_c1->check() && m_c2->check();}
private:
    Condition* m_c1;
    Condition* m_c2;
};
```


SM: Flexible

```
class Action
{
public:
    void start() {};
    void update() {};
    void end() {};
};
```

```
class GotoAction: public Action
{
public:
    void start() { /* pathfind to destination */
}
    void update() { /* move through path */ }
    GotoAction(const Pos3D& pos);
private:
    Pos3D m_destination;
};
```


SM: Flexible

```
class SM
{
    std::vector<State*> m_States;
    State* m_currentState;
public:
    void load();
    void start();
    void update();
};
```

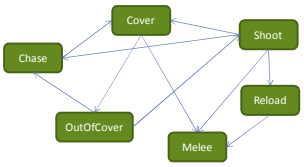
SM: Flexible

```
void SM::update()
{
    m_currentState->update();
    const Transitions& trans = m_currentState->getTransitions();
    for tran in trans
    {
        if (tran.canTrigger())
        {
            m_currentState->onExit();
            State* nextState = tran.trigger();
            nextState->onEnter();
            m_currentState = nextState;
            return;
        }
    }
}
```

```
class SM
{
    std::vector<State*> m_States;
    State* m_currentState;
public:
    void load();
    void start();
    void update();
};
```


SM: Tamaño

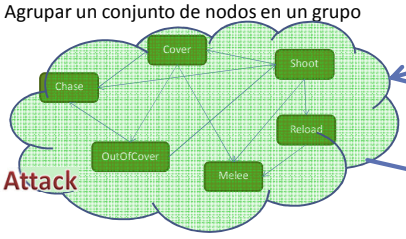
- Estado Attack generalmente mucho más complicado



RunAway

SM Jerárquicas (HSM)

- Agrupar un conjunto de nodos en un grupo



Idle

RunAway

HSM

- Agrupar un conjunto de nodos en un grupo
- El grupo funciona como un estado normal
- En su update actualiza su máquina de estados interna

```
class SM
{
    std::vector<State*> m_States;
    State* m_currentState;
public:
    void load();
    void start();
    void update();
};
```

```
class State
{
    Action* m_enterAction;
    Action* m_exitAction;
    Action* m_stateAction;
    Transitions m_transitions;
public:
    void onEnter();
    void update();
    void onExit();
    const Transitions& getTransitions();
};
```


HSM

- Al entrar en el estado reinicio la máquina de estados
- En el update del estado gestiono la máquina de estados

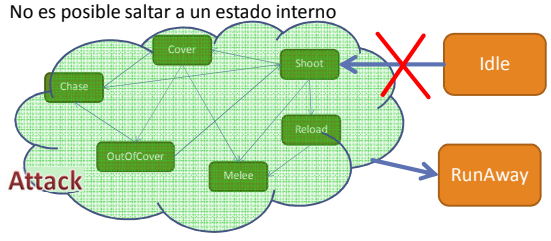
```
class State
{
    Action* m_enterAction;
    Action* m_exitAction;
    Action* m_stateAction;
    Transitions m_transitions;
public:
    void onEnter();
    void update();
    void onExit();
    const Transitions& getTransitions();
};
```

SM::start

SM::update

HSM

- No es posible saltar a un estado interno



HSM

- Implementación más compleja
- La complejidad ya no se puede encapsular fácilmente
- En la bibliografía hay soluciones para ello