

★Gameplay programming

De Nadai Mattia - denadaimattia@gmail.com



★ CHI SONO

Nome: Mattia

Cognome: De Nadai

E-Mail: denadaimattia@gmail.com

Attuale: Lead programmer presso Milestone

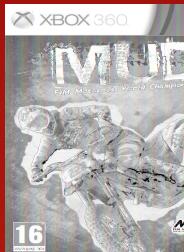
Precedente: Master in Game Development
Università degli studi di Padova



CHI SONO

Giochi pubblicati:

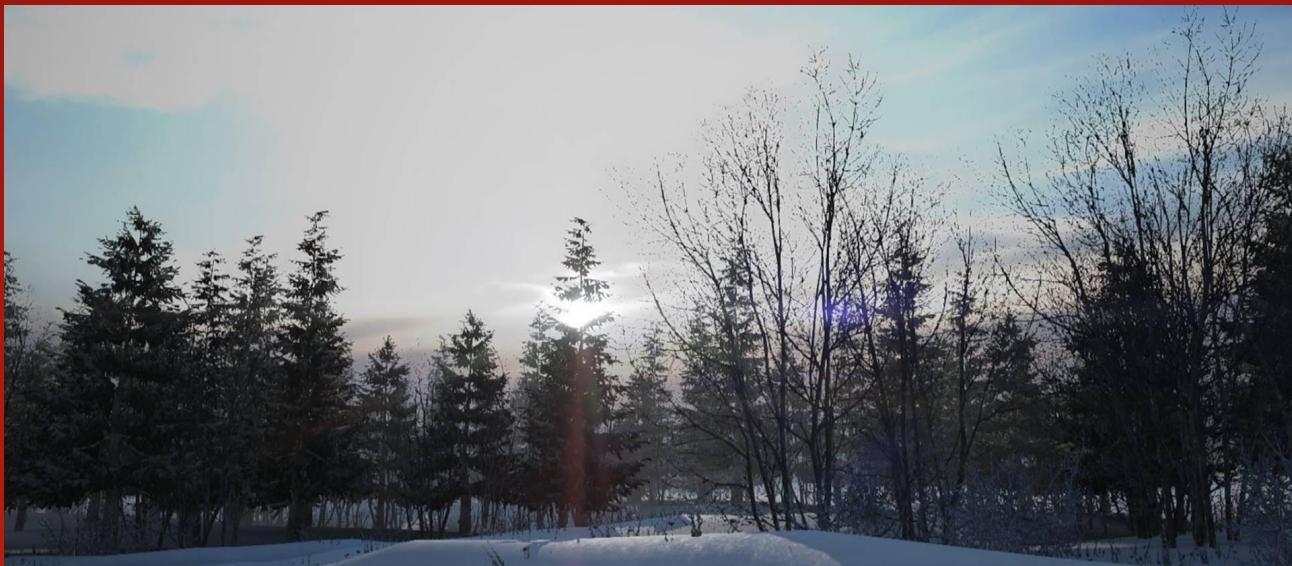
- WRC3
- WRC4
- MUD
- MotoGP13
- WRC Powerslide
- MXGP
- MotoGP14
- MXGP HD
- Ride



★ NUOVI SVILUPPI



★ NUOVI SVILUPPI



MILESTONE 2015 - Sweden - Work in progress

SÉBASTIEN **LOEB**
RALLY EVO

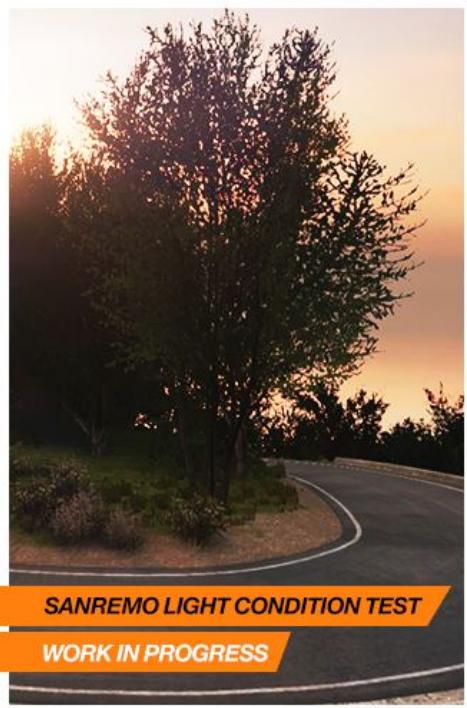
★ NUOVI SVILUPPI



MILESTONE 2015 - Wales-Ground Test - Work in progress

SÉBASTIEN LOEB
RALLY EVO

★ NUOVI SVILUPPI



★ NUOVI SVILUPPI



★ FPS

- Efficiente sistema di rendering per gestire ampie mappe
- Accurata gestione della camera per il movimento e la mira
- Animazioni ad alta fedeltà
- AI intelligenti



★ Platformers

- Puzzle-like environmental elements;
- Camera in terza persona che segue sempre il player
- Complesso sistema di collision detection tra camera e ambiente



★ FIGHTING

- Ricco sistema di animazioni
- Sistema di “Hit detection” molto accurato
- Sistema di input complesso per permettere complesse combinazioni di tasti
- Fisica basata sulla simulazione di vestiti e capelli dei giocatori



★ RACING

- Vari trucchi sono usati per renderizzare elementi di sfondo, come l'utilizzo di immagini bidimensionali per alberi, colline o montagne.
- La camera tipicamente segue il veicolo utilizzando una prospettiva in terza persona. In altri casi viene utilizzata una camera cockpit.
- Sistema di collision detection tra la camera e l'ambiente.



★ REAL-TIME STRATEGIC

- Ogni oggetto sullo schermo ha una risoluzione bassa in modo da permettere di averne molti sullo schermo nello stesso momento.
- L'interazione dell'utente avviene tipicamente con singoli click con l'aggiunta di interfaccia UI.



★ MMOG

- Server molto capienti e performanti
- La grafica è quasi sempre di più bassa qualità rispetto a quelli non-massively.



★ ALTRI GIOCHI

- Giochi sportivi
- RPG
- God games come Black&White
- Social
- Puzzle game



★ NUOVE TENDENZE

- Il ruolo del programmatore sta cambiando
- Il modo di sviluppare un gioco è cambiato
- Oggi chiunque può sviluppare un gioco

★ NUOVE TENDENZE

Oggi non serve più saper programmare per creare un gioco

Serve trovare lo strumento giusto.

★ NUOVE TENDENZE

Il programmatore si deve preoccupare di fornire gli strumenti giusti in modo da poter creare efficientemente un gioco il quale abbia una ottima qualità visiva e interattiva.

★ NUOVE TENDENZE

Gli strumenti oggi più utilizzati sono:

- Unity 3D
- Unreal Engine

★ OBIETTIVO DEL CORSO

Sviluppare un'architettura in grado di creare giochi
in maniera data-driven con logiche scritte
utilizzando linguaggi di scripting



★ STRUMENTI

Gli strumenti principali che utilizzeremo in questo corso sono i seguenti:

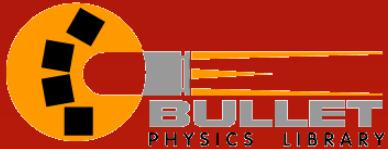
- Ambiente di sviluppo (IDE - Integrated Development Environment)
- Debugger
- Sistema di controllo versione (SVN - Subversion)
- Tool per creare la documentazione del nostro software



★ Strumenti

Framework che utilizzeremo:

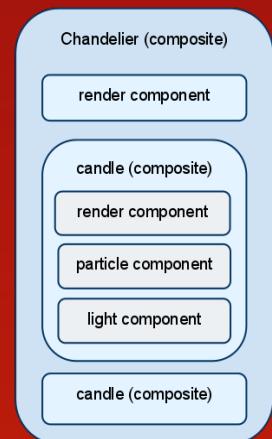
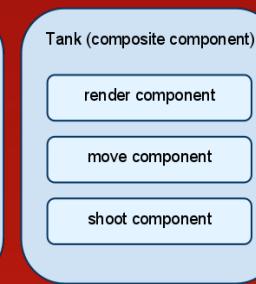
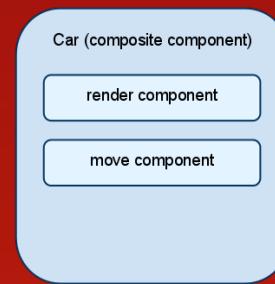
- Engine 3D: Ogre3D
- Engine 2D: SDL 2.0
- Physics 3D: Bullet
- Physics 2D: Box2D
- User Interface: CEGUI
- Scripting language: LUA



L'IDEA

L'idea è quella di avere una serie di componenti i quali definiscono dei comportamenti che si possono combinare per creare degli oggetti complessi.

Invece di scrivere grandi classi in codice per definire un oggetto, scriviamo dei componenti che definiscono un preciso e generico comportamento e li aggreghiamo insieme per creare l'oggetto desiderato.



L'IDEA

Questo ci permette di avere:

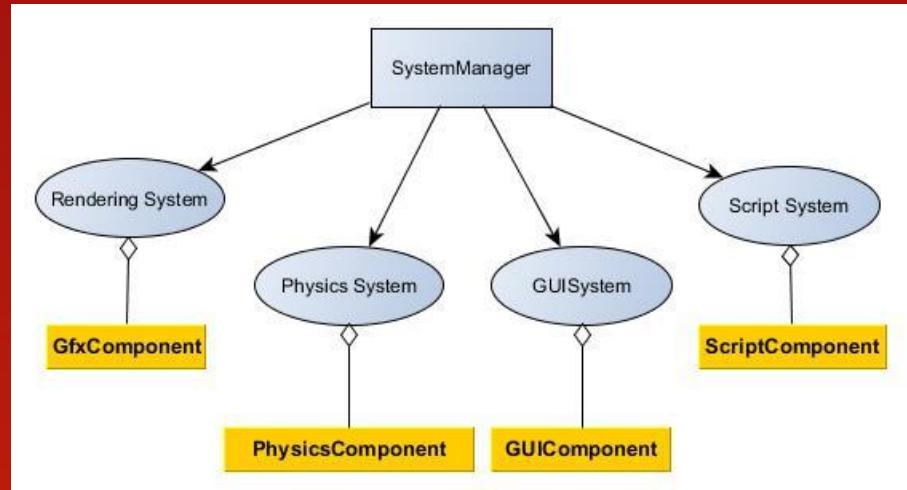
- una migliore separazione del codice
- un miglior riuso del codice
- una facilità nel correggiere eventuali bug
- una maggiore facilità di comprensione del codice
- permette di avere una ben definita struttura di come sono creati i componenti.

L'IDEA

I componenti non contengono alcuna logica ma sono soltanto una collezione di informazioni.

Tutti i componenti sono gestiti da opportuni sistemi i quali si occupano del loro aggiornamento.

I sistemi hanno una funzione di “Update” la quale viene chiamata dal “SystemManager”.



★ INIZIAMO

Abbiamo strutturato il nostro engine in 2 progetti:

- *Main*: rappresenta il progetto utilizzatore dell'engine
- *Engine*: implementa tutte le funzionalità dell'engine

★ MAIN PROJECT

Di cosa si occupa:

- Istanzia l'engine (*Singleton*)
- Inizializza l'engine
- Esegue l'update dell'engine

SINGLETON

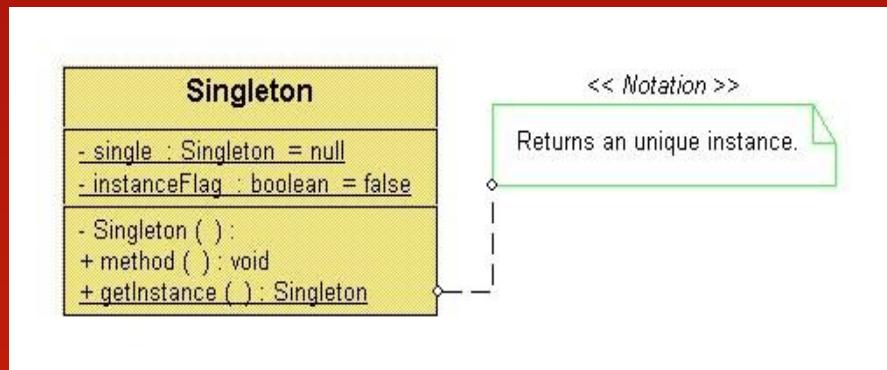
Il singleton è un design pattern creazionale che ha lo scopo di garantire che di una determinata classe venga creata *una ed una sola* istanza, e di fornire un punto di accesso globale a tale istanza.

Pregi:

- Chiarezza

Difetti:

- E' globalmente accessibile
- Non si conosce l'ordine di creazione e distruzione



★ ENGINE PROJECT

Di cosa si occupa:

- Gestisce tutti i sistemi ed i componenti che compongono il nostro gioco

★ ENGINE PROJECT

Ci sono 2 aspetti importanti per quanto riguarda lo sviluppo di un engine:

- Performance
- Memoria

★ ENGINE PROJECT

Come identifichiamo le entità che compongono il nostro gioco?

- Stringa (string)
- Intero (int, float, long int)
- Classe (oggetto complesso)

★ ENGINE PROJECT

Analizziamo le varie alternative:

- Classe: inutile utilizzare un'oggetto complesso per identificare un'entità. Spreco di memoria e performance
- Stringa: sicuramente da molta più chiarezza in fase di debug ma a livello di memoria è troppo dispendiosa
- Intero: è sicuramente troppo poco leggibile in fase di debug ma a livello di memoria è ideale

★ ENGINE PROJECT

La soluzione migliore è:

In fase di debug avere una stringa che identifica il nome dell'entità, mentre in fase di release avere solo un intero come identificatore.

★ ObjectId

Questo oggetto verrà utilizzato per identificare ogni entità.

- Useremo ObjectId anzichè String
- Genera un hash data una stringa
- Molto meno pesante della stringa
- In debug abbiamo la possibilità di leggere la stringa che genera l'hash

★ ENGINE PROJECT

Il nostro engine sarà formato da:

- *Managers*: gestori a basso livello
- *Systems*: aggiornano i componenti
- *Components*: definiscono un comportamento

★ MANAGERS

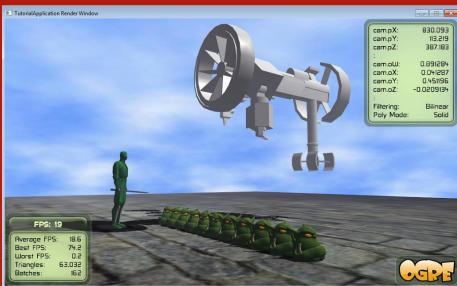
Vediamo in dettaglio i manager presenti nell'engine i quali riguardano la gestione di:

- Grafica
- FSM
- Fisica
- Input
- Script
- Log
- Risorse
- Sistemi

★ GRAFICA

L'engine integra 2 framework:

- OGRE3D: Per il rendering 3D
- SDL (Standard Direct Library): Per il rendering 2D



★ GRAFICA

Per poter gestire 2 o più modi di rendering la soluzione migliore:

Utilizzare un'interfaccia comune che poi verrà implementata da ogni renderer

★ GRAFICA

La creazione viene eseguita da un oggetto “*RenderFactory*” (*Singleton*).

- Ci permette di avere accesso al renderer anche da altre classi dell’engine

(es. *PhysicsDebugDrawer*)

```
class RendererFactory {  
public:  
    virtual ~RendererFactory() {}  
    static void CreateRenderer(bool i_bIs3D)  
    {  
        if(i_bIs3D) {  
            m_pRenderer = Renderer3D::CreateInstance();  
        }  
        else {  
            m_pRenderer = Renderer2D::CreateInstance();  
        }  
    }  
    static IRenderer* GetRendererPtr() {  
        return m_pRenderer;  
    }  
private:  
    static IRenderer* m_pRenderer;  
};
```

★ GRAFICA

INTERFACCIA COMUNE PER IL RENDERING

```
class IRenderer
{
public:
    IRenderer()
        : m_bFullscreen(false)
        , m_fPosX(0)
        , m_fPosY(0)
        , m_fWidth(640)
        , m_fHeight(480)
    {
        ...
    }
    ...
}
```

★ GRAFICA

OGRE 3D

```
class Renderer3D : public IRenderer {  
  
public:  
    static IRenderer* CreateInstance();  
    virtual ~Renderer3D();  
  
...  
    virtual void Update(real i_fFrametime,  
real i_fTimestep);  
...  
}
```

SDL

```
class Renderer2D : public IRenderer {  
  
public:  
    static IRenderer* CreateInstance();  
    virtual ~Renderer2D();  
  
...  
    virtual void Update(real  
i_fFrametime, real i_fTimestep);  
...  
}
```

★ GRAFICA

OGRE 3D

```
void Renderer3D::Update(real i_fDelta, real  
i_fTimestep) {  
    if (m_pMainWindow->isActive()) {  
        m_pRoot->renderOneFrame(i_fTimestep);  
    }  
    else if (m_pMainWindow->isVisible()) {  
        m_pMainWindow->update();  
    }  
}
```

SDL

```
void Renderer2D::Update(real  
/*i_fFrametime*/, real /*i_fTimestep*/) {  
    CEGUI::System::getSingleton().  
    renderGUI();  
  
    SDL_GL_SwapWindow(m_pWindow);  
}
```

★ GRAFICA

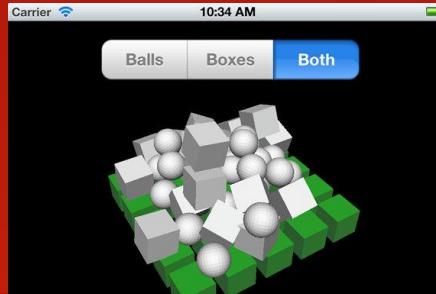
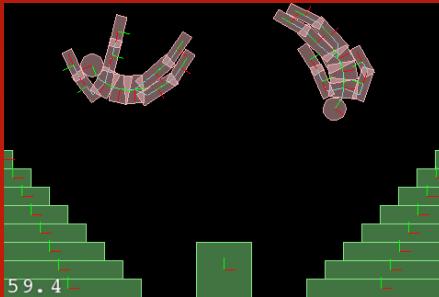
In un file di configurazione viene definito quale tipo di rendering utilizzare.

```
<Root>
  <Is3D enable="false"/>
  <AmbientColor r="0.0" g="1.0" b="1.0" a="1.0"/>
  <ShadowTechnique value="Stencil-Additive"/>
  <FullScreen enable="false"/>
  <ScreenSize x="100" y="100" width="800" height="600"/>
  <ColorBkg r="0" g="1" b="1" a="1"/>
  <Viewport left="-100" right="100" bottom="-100" top="100" near="-10" far="10"/>
</Root>
```

★ PHYSICS

Per quanto riguarda la gestione della fisica nel gioco, ci siamo basati su:

- *Bullet*: per quando utilizziamo una rappresentazione 3D
- *Box2D*: per quando utilizziamo una rappresentazione 2D



Anche per la fisica abbiamo un file di configurazione.

```
<Root>
    <Is3D enable="false"/>
    <AmbientColor r="0.0" g="1.0" b="1.0" a="1.0"/>
    <ShadowTechnique value="Stencil-Additive"/>
    <FullScreen enable="false"/>
    <ScreenSize x="100" y="100" width="800" height="600"/>
    <ColorBkg r="0" g="1" b="1" a="1"/>
    <Viewport left="-100" right="100" bottom="-100" top="100" near="-10" far="10"/>
    <PhysicsEngine type="Box2D"/>
    <PhysicsDebugDraw enable="false"/>
    <Gravity value="9.8"/>
</Root>
```

★ PHYSICS

```
class PhysicsWorldFactory
{
public:
    virtual ~PhysicsWorldFactory() { }
    virtual PhysicsWorld* CreateWorld() const = 0;
};
```

BULLET

```
class BulletWorldFactory : public
PhysicsWorldFactory
{
public:
    PhysicsWorld* CreateWorld() const {
        return new BulletWorld();
    }
};
```

Interfaccia
creatore del
mondo fisico

BOX2D

```
class Box2DWorldFactory : public PhysicsWorldFactory
{
public:
    PhysicsWorld* CreateWorld() const {
        return new Box2DWorld();
    }
};
```

★ PHYSICS

```
class PhysicsWorld
{
public:
    PhysicsWorld();
    virtual ~PhysicsWorld();
    virtual void Configure() = 0;
    virtual void Cleanup() = 0;

    virtual void AddObject(const PhysicsObject* i_object) = 0;
    virtual void RemoveObject(const PhysicsObject* i_object) = 0;
    virtual void RemoveAllObjects() = 0;

    virtual void Step(const real i_dt) = 0;
    virtual void PostStep() = 0;

    ...
}
```

INTERFACCIA
DEL MONDO
FISICO

★ PHYSICS

BULLET

```
class BulletWorld : public PhysicsWorld  
{  
    ...  
}
```

BOX2D

```
class Box2DWorld : public PhysicsWorld  
{  
    ...  
}
```

```
PhysicsWorldFactory* fact = NULL;  
if(is3D)  
{  
    fact = new BulletWorldFactory();  
}  
else  
{  
    fact = new Box2DWorldFactory();  
}  
pPhysicsSystem = new PhysicsSystem(fact);  
delete fact;
```



BULLET

```
void BulletWorld::Step(const real i_dt)
{
    m_dynamicsWorld->stepSimulation(1.f / 60.f);
    PostStep();
}
```

BOX2D

```
void Box2DWorld::Step(const real i_dt)
{
    m_World->Step(1.f / 60.f, 6, 2);
    PostStep();
}
```

Nei 2 mondi fisici andiamo a specializzare le funzioni dichiarate nell'interfaccia.

★ PHYSICS

Utilizzando 2 mondi fisici diversi dobbiamo differenziare l'implementazione dei corpi rigidi che il sistema dovrà gestire.

Tutto questo sarà inserito nel PhysicsComponent che vedremo più avanti.



Una FSM o macchina a stati finiti è utilizzata per descrivere flussi logici



Le FSM (Finite State Machine) nel campo dei videogiochi sono molto utilizzate

- AI
- Flusso di gioco
- Animazioni

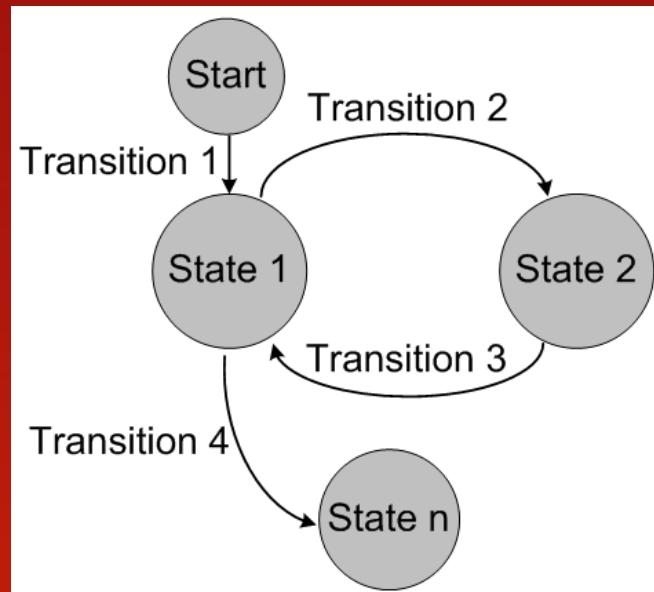
Nell'engine è stata implementata una semplice FSM che ci permetterà di creare un flusso di gioco.

★ FSM

Una FSM è formata da:

- Stati
- Transizioni

Ogni FSM ha uno Stato iniziale



★ FSM - Manager

```
class FSManager : public Singleton<FSManager> {  
public:  
    static void CreateInstance();  
    ~FSManager();  
    void AddFSM(FSM* i_pFSM);  
    void RemoveFSM(const ObjectId& i_oName);  
    void Update(real i_fFrametime, real i_fTimestep);  
    static void StartFSM(const char* i_szFSMName);  
  
private:  
    FSManager();  
    void RegisterLuaFunctions();  
    typedef ObjectId FSMID;  
    MGDMMap<FSMID, FSM*> m_vFSMMap;  
};
```

FSM Manager

★ FSM

```
class FSM {  
public:  
    FSM(const std::string& i_szName);  
    ~FSM();  
    void Update(real i_fFrametime, real i_fTimestep);  
    void AddState(State* i_pState);  
    void SetInitialState(const StateID& i_oInitialState)  
    void RegisterTransitions(const MGDMAP<TransitionID, Transition*>& m_TransitionMap);  
    void DoTransition(const char* i_szTransition);  
    void Start();  
private:  
    MGDMAP<StateID, State*> m_States;  
    StateID m_oInitialState;  
    State* m_pCurrentState;  
    RegistryEventHandler<FSM, const char*> m_TransitionEvent;  
    bool m_bIsStarted;  
};
```

FSM

★ FSM - State

Uno stato definisce il comportamento del sistema in un dato momento.

Uno stato è formato da:

- OnEnter: Invocata all'entrata dello stato
- OnUpdate: Invocata quando lo stato è attivo
- OnExit: Invocata all'uscita dello stato

★ FSM - State

```
class State {  
public:  
    ID_DECLARATION;  
    State(const std::string& i_szName);  
    ~State();  
    void OnEnter();  
    void OnUpdate();  
    void OnExit();  
    const StateID& DoTransition(const char* i_szTransition);  
    void AddTransition(Transition* i_pTransition);  
    void SetScriptOnEnter(const char* i_szFilename);  
    void SetScriptOnUpdate(const char* i_szFilename);  
    void SetScriptOnExit(const char* i_szFilename);  
private:  
    MGMap<TransitionID, Transition*> m_TransitionMap;  
    std::string m_szScriptOnEnter;  
    std::string m_szScriptOnUpdate;  
    std::string m_szScriptOnExit;  
};
```

★ FSM - Transition

Una transizione definisce un possibile cambiamento di stato.



Lo stato da cui parte la transizione si chiama “Source State” e quello in cui arriva “Target State”

★ FSM - Transition

```
class Transition {  
public:  
    ID_DECLARATION;  
    Transition(const std::string& i_szName, const StateID& i_oStateTargetID);  
    ~Transition();  
    const StateID& GetStateTargetID() const {  
        return m_oStateTargetID;  
    }  
private:  
    StateID           m_oStateTargetID;  
    std::string        m_szName;  
    TransitionID      m_old;  
};
```

★ FSM - Transition

Una transizione avviene quando si verifica un evento.

Quando una transizione parte viene chiamata la funzione di uscita del “Source State” e successivamente la funzione di entrata del “Target State”.

★ FSM - Transition

```
void FSM::DoTransition(const char* i_szTransition) {  
    if(m_pcurrentState) {  
        const StateID& oStateTarget(m_pcurrentState->DoTransition(i_szTransition));  
        MGDMMap<StateID, State*>::iterator it = m_States.find(oStateTarget);  
        if(it != m_States.end()) {  
            State* pNewState = (*it).second;  
            if(pNewState) {  
                m_pcurrentState->OnExit();  
                m_pcurrentState = pNewState;  
                m_pcurrentState->OnEnter();  
            }  
        }  
    }  
    ...  
}
```

★ FSM - Transition

```
...
else {
    MGDMMap<StateID, State*>::iterator it = m_States.find(m_oLnitialState);
    if(it != m_States.end()) {
        m_pCurrentState = (*it).second;
        if(m_pCurrentState) {
            m_pCurrentState->OnEnter();
        }
    }
}
```

★ FSM - Transition

```
const StateID& State::DoTransition( const char* i_szTransition ) {  
    MGDMMap<TransitionID,Transition*>::iterator it = m_TransitionMap.  
find(ObjectId(i_szTransition));  
  
    if(it != m_TransitionMap.end()) {  
        return (*it).second->GetStateTargetID();  
    }  
    return INVALID_HASH;  
}
```

★ FSM - Transition

Come viene lanciata la transizione?

- Viene lanciato un evento da script con il path “DoTransition”
- L’FSM è in ascolto sull’evento e lancia la funzione DoTransition();

```
m_TransitionEvent.Subscribe(this, &FSM::DoTransition, ObjectId("DoTransition"));
```

★ FSM - Factory

Come abbiamo gestito la creazione degli stati?

- Da script chiamiamo la funzione che carica il descrittore dell'FSM
 - static void LoadFile(const char* i_szFilename)
- Creiamo la FSM leggendo il suo descrittore
- Aggiungiamo la FSM creata all'FSMManager
 - FSMManger::GetSingleton().AddFSM(pFSM);

★ FSM - Factory

```
class FSMFactory : public Singleton<FSMFactory> {  
public:  
    static void CreateInstance();  
    ~FSMFactory();  
private:  
    FSMFactory();  
    typedef ObjectID StateID;  
    static void LoadFile(const char* i_szFilename);  
    static State* CreateState(const tinyxml2::XMLElement* i_pState);  
    static Transition* CreateTransition(const tinyxml2::XMLElement* i_pNode);  
    static void RemoveFile(const char* i_szFilename);  
    void RegisterComponents();  
    void RegisterGlobalLuaFunction();  
};
```

★ FSM - Data-Driven

La FSM sarà definita da dati in un file XML.

Il file XML ha la seguente struttura...

★ FSM - Data-Driven

```
<FSM Name="2D_GAME_FSM">
    <InitialState Name="MainMenu"/>

    <State Name="MainMenu"> <!-- Create GUI -->
        <ScriptOnEnter Filename="Resources/2D/FSM/MainMenuScript.lua"/>
        <Transition Name="START_GAME" Target="Game"/>
    </State>

    <State Name="Game"> <!-- Create players and ball -->
        <ScriptOnEnter Filename="Resources/2D/FSM/GameLogicScript.lua"/>
        <Transition Name="END_GAME" Target="EndGame"/>
    </State>

    <State Name="EndGame">
        <!-- remove players and ball, create GUI to back to main menu or restart the game -->
        <ScriptOnEnter Filename="Resources/2D/FSM/EndGameScript.lua"/>
        <Transition Name="RESTART" Target="Game"/>
        <Transition Name="MAIN_MENU" Target="MainMenu"/>
    </State>
</FSM>
```



Proviamo ad implementare la nostra FSM!

Le classi da implementare sono:

- `FSMManager`
- `FSM`
- `State`