Gameplay programming
Lezione 2

De Nadai Mattia - denadaimattia@gmail.com

NELLA LEZIONE PRECEDENTE

Abbiamo visto:

- Obiettivo
- Struttura progetto
- Managers
  - Grafica
    - RenderFactory
    - RenderManager
  - Fisica
    - PhysicsFactory
    - PhysicsWorld
  - FSM
    - FSMManager
    - FSM
    - State
    - Transition

INPUT

Per semplicità e velocità di implementazione abbiamo diviso l'input in base al framework che utilizziamo per il render.

Questa scelta è data dal fatto che i framework utilizzati fornivano già una gestione dell'input:

- OIS per Ogre3D
- Sistema ad eventi per SDL

INPUT

E’ stata utilizzata la stessa struttura dei manager visti in precedenza anche per l’istanziazione dell’ input manager.

```
class InputManagerFactory {
public:
	static void  CreateInstance(Ogre::RenderWindow* i_pWindow);
	virtual		~InputManagerFactory();

	static void CreateInputManager(bool i_bIs3D, Ogre::RenderWindow* i_pWindow){
		if(i_bIs3D) {
			m_pInputManager = InputManager3D::CreateInstance(i_pWindow);
		}
		else {
			m_pInputManager = InputManager2D::CreateInstance(NULL);
		}
	}
	static IInputManager* GetInputManagerPtr() {
		return m_pInputManager;
	}
private:
	static IInputManager* m_pInputManager;
};
```

INPUT

```
class IInputManager
{
public:
        IInputManager() {}
        virtual ~IInputManager() {}

        virtual void Update(real i_fFrametime, real i_fTimestep) = 0;

        MGDVector<int32>                        m_KeyButtonDown;
...
};
```

Interfaccia per l'inputManager

INPUT 3D -> OIS

```
class InputManager3D :      public IInputManager,
                            public OIS::MouseListener,
                            public OIS::KeyListener,
                            public Ogre::WindowEventListener
{
public:
    static IInputManager* CreateInstance(Ogre::RenderWindow* i_pWindow);

    virtual                 ~InputManager3D();

    virtual void Update(real i_fFrametime, real i_fTimestep);
...
}
```

INPUT 2D -> SDL

```
class InputManager2D : public IInputManager
{
public:
        static IInputManager*  CreateInstance(Ogre::RenderWindow* i_pWindow);

        virtual                ~InputManager2D();

        virtual void OnEvent(void* i_pEvent);
        virtual void Update(real i_fFrametime, real i_fTimestep);
...
}
```

SCRIPT MANAGER

Cosa è un linguaggio di scripting?

Un linguaggio di scripting è un linguaggio di alto livello interpretato da un'altro programma a runtime.

Pregi:

- Non ha bisogno di essere compilato
- Facilita la prototipazione
- E' un linguaggio ad alto livello

Difetti:

- Difficile da debuggare
- Pesante computazionalmente
- Pesante in memoria

SCRIPT MANAGER

Linguaggi di scripting

- *Python*

```
# This is a comment in python
def process_actors(actors):
    for actor in actors:
    if actor.is_alive():
    actor.process()
```

- *LUA*

```
-- This is a comment in Lua
function process_actors(actors)
        for index, actor in ipairs(actors) do
                if actor:is_alive() then
                        actor.process()
                end
        end
end
```

In questo engine utilizzeremo:

LUA

SCRIPT MANAGER

```
class LuaManager: public Singleton<LuaManager>, public IScriptManager {
public:
        static void CreateInstance();
        ~LuaManager();

        virtual void Init();

        //It's used to run a lua file
        virtual void ExecuteFile(const char* i_pFilename);

        //It's used to run a script directly
        virtual void ExecuteString(const char* i_pString);

        //Get the global evironment variables
        LuaPlus::LuaObject GetGlobalVars(void);

        //Get the LuaState
        LuaPlus::LuaState* GetLuaState(void) const;

private:
        LuaPlus::LuaState* m_pLuaState;
};
```

Esecuzione Script

Tabella globale di LUA

Lua State

SCRIPT MANAGER

Come possiamo debuggare un linguaggio di scripting?

LOG

Implementando un sistema che mi possa fornire tutte le informazioni necessarie durante l'esecuzione del programma

Questo sistema è comunemente chiamato

LOG

LOG MANAGER

Il LOG in fase di debug è uno strumento indispensabile!

Nell’engine è stata integrata la libreria fornita da Google chiamata:

GLOG

LOG MANAGER

Livelli di severity:

- INFO
- WARNING
- ERROR
- FATAL

Ci permette di selezionare il livello di criticità dell’informazione

LOG MANAGER

I problemi:

- per grosse applicazioni potrebbe diventare troppo verboso
- si potrebbe perdere il significato delle informazioni

Per ridurre questi problemi:

Il LOG è stato suddiviso in contesti per poter scegliere di quale porzione di codice vogliamo avere informazioni.

LOG MANAGER

Un file di configurazione definisce cosa loggare:

```
<LogConfiguration>
        <Context value="ACTOR_FACTORY_CONTEXT"/>
        <Context value="FSM_FACTORY_CONTEXT"/>
        <Context value="COMPONENTS_CONTEXT"/>
        <!--<Context value="SYSTEMS_CONTEXT"/>-->
        <Context value="RENDERING_CONTEXT"/>
        <Context value="FSM_CONTEXT"/>
        <Context value="INPUT_CONTEXT"/>
        <Context value="MANAGER_CONTEXT"/>
        <Context value="ASSERT_CONTEXT"/>
        <Context value="APPLICATION_CONTEXT"/>
</LogConfiguration>
```

LOG MANAGER

IMPLEMENTAZIONE

```
class LOGManager : public Singleton<LOGManager> {
	public:
		static void CreateInstance();
		~LOGManager();

		void AddLogContext(const ObjectId& i_oContext)
		{
			m_LogContextAvailable.push_back(i_oContext);
		}

		void WriteLog(LogSeverity i_oLogSeverity, const ObjectId& i_oContext, const char*
i_szMassage, ...);

	private:
		LOGManager();
		MGDVector<ObjectId> m_LogContextAvailable;

		static const char* m_szFileName;
	};
```

LOG MANAGER

Proviamo ad implementare un semplice parser per il file XML usato per la configurazione dei contesti da utilizzare.

Cosa fare:

- Check se il path è valido
- Load file
- Iterazione sui nodi identificati con context
- Aggiungere il nome del contesto

Funzioni utili di Tinyxml2:

- *tinyxml2::XMLDocument oDocument;*
  oDocument.LoadFIle(const char);*
  Utilizzata per aprire il file
- *oDocument.RootElement();*
  Utilizzata per farsi ritornare il nodo root dell'xml, Ritorna un *const tintxml2::XMLElement**
- *pRoot->FirstChildElement(const char*)*
  Ritorna il primo nodo identificato con il parametro che gli abbiamo passato. Il tipo di ritorno è XMLElement*
- *pContext->NextSiblingElement(const char*)*
  Ritorna il nodo successivo identificato con il parametro che gli abbiamo passato. Il tipo di ritorno è *XMLElement**
- Il tipo *XMLElement* ha la fuzione *Attribute(const char*)* che permette di farsi ritornare il valore dell'atributo identificato con il nome che gli passiamo alla funzione

SYSTEM MANAGER

Questa classe è il cuore di tutta l’architettura.

Gestisce:

- lista dei sistemi creati all’avvio
- lista dei componenti creati

SYSTEM MANAGER

- I sistemi creati all'avvio vengono inseriti in un array
- L'update usa un sistema di priorità per aggiornare i sistemi
- L'update viene eseguita in fasi ed ogni fase contiene i sistemi che devono essere aggiornati in quel momento
- Questo permette di gestire le priorità/dipendenze tra i sistemi
  - Es. permette di aggiornare la posizione prima di renderizzare

```
typedef MGDMap<ObjectId, System*>        SystemTable;
MGDVector<SystemTable>                   m_vPerPhaseSystemArray;
```

SYSTEM MANAGER

- Gestisce l’inserimento e la rimozione dei componenti
- E’ l’unico oggetto che conosce tutti i componenti presenti nel gioco
- Tutti i sistemi interrogano il SystemManager per ottenere i componenti di loro competenza da aggiornare
- I componenti sono registrati in una mappa

SYSTEM MANAGER

La mappa che contiene i componenti è:

```
typedef MGDMap<ObjectId, MGDVector<Component*> >        EntityComponentTable;
typedef MGDMap<ObjectId, EntityComponentTable>           ComponentTable;
ComponentTable                                           m_vComponentTable;
```

- ComponentTable
  - ID del tipo componente
  - Lista di Entity (Oggetto composto - Actor)
- EntityComponentTable
  - ID dell'entity (Nome dell'oggetto composto)
  - Componenti che definiscono l'oggetto

SYSTEM MANAGER

Il *SystemManager* gestisce anche le entità create come template -> Prefab di Unity3D

Anche i template sono inseriti in una mappa di tipo:

```
EntityComponentTable        m_vTemplateTable;
```

Quando voglio istanziare un template lo inserisco nella lista dei componenti principali così viene aggiornato dai sistemi

CREAZIONE DEI MANAGERS

All’inizio del gioco vengono istanziati tutti i manager:

```
void InitInstance()
{
…
        SystemManager::CreateInstance();
        LuaManager::CreateInstance();
        RendererFactory::CreateRenderer(is3D);
        InputManagerFactory::CreateInputManager(is3D, RendererFactory::GetRendererPtr()->GetWindow());
        ResourceManager::CreateInstance(is3D);
        EventRegistry::CreateInstance();
        ActorFactory::CreateInstance();
        FSMManager::CreateInstance();
        FSMFactory::CreateInstance();
…
}
```

SISTEMI

Nella creazione di un sistema viene passato un valore che ne determina la fase in cui viene fatta la sua update.

I sistemi creati all’avvio del gioco sono:

- RenderingSystem
- PhysicsSystem
- ScriptSystem
- AnimatorSystem
- GuiSystem

SISTEMA AD EVENTI

Utilizziamo un sistema ad eventi per permettere che i sistemi siano sempre aggiornati su quello che a loro compete.

I sistemi si registrano a degli eventi che il “SystemManager” lancia in determinati momenti

SISTEMA AD EVENTI

Il sistema ad eventi è formato da:

- EventRegistry
- RegistryEventHandler
- RegistryEventPublisher

EVENTREGISTRY

E’ colui che gestisce tutti gli eventi lanciati/ascoltati durante il gioco.

Ha una mappa:

ObjectId -> Lista di Handler

EVENTREGISTRY

```
class EventRegistry : public Singleton<EventRegistry> {
public:
        class EventHandler      { … }
private:
        typedef MGDMap<ObjectId,EventHandler*>      EventsMap;
        typedef EventsMap::iterator                 EventsMapIt;

public:
        ~EventRegistry();
        static void CreateInstance();

        …

private:
        EventRegistry();
        void RegisterGlobalLuaFunction();

        EventsMap m_Registry;
        EventsMap m_LuaRegistry;
};
```

EVENTREGISTRY

```
class EventHandler {
public:
        EventHandler();
        virtual ~EventHandler();
        const ObjectId& GetPath() const;

        void SetNextHandler(EventHandler* i_pNextNode);
        EventHandler* GetNextHandler() const;

        void SubscribeEventHandler(const ObjectId& i_oEventPath);
        void UnsubscribeEventHandler();

        void AddHandler( EventHandler* i_pEventHandler );
        void RemoveHandler(EventHandler* i_pEventHandler);

        virtual void Invoke( const ObjectId& pParamType, void* i_pParam ) = 0;

private:
        EventHandler*   m_pNextHandler;
        ObjectId        m_oEventPath;
};
```

Registrazione dell'evento

Deregistrazione dell'evento

Esecuzione della funzione

REGISTRYEVENTHANDLER

Questo oggetto viene utilizzato quando si vuole mettersi in ascolto di un evento.

E’ composto da 2 membri:

- Puntatore alla classe che deve ascoltare l’evento
- Funzione da invocare quando l’evento viene lanciato

REGISTRYEVENTHANDLER

Il funzionamento consiste in:

- Creare l'oggetto
- Registrarlo all'EventRegistry specificando:
  - Puntatore alla classe
  - Puntatore alla funzione da invocare
  - Path dell'evento da ascoltare
- Una volta che viene lanciato l'evento, l'EventRegistry si preoccuperà di chiamare la funzione Invoke() la quale chiama il metodo precedentemente registrato

REGISTRYEVENTHANDLER

```
template < typename Class, typename ParamType>
class RegistryEventHandler : public EventRegistry::EventHandler
{
public:
	typedef void (Class::*CallbackFunc)(const ParamType&);

	RegistryEventHandler();

	~RegistryEventHandler()
	{
		Unsubscribe();
	}

	void Subscribe( Class* i_pTargetObject, CallbackFunc i_pMethod, const ObjectId i_oEventPath )
	{
		m_pTargetObject = i_pTargetObject;
		m_pMethod = i_pMethod;

		SubscribeEventHandler(i_oEventPath);
	}

	…
```

REGISTRYEVENTHANDLER

```
        …

        void Unsubscribe()
        {
                UnsubscribeEventHandler();
        }

        void Invoke( const ObjectId& pParamType, void* i_pParam )
        {
                if (m_pTargetObject)
                {
                        const ParamType& oParam = *static_cast<const ParamType*>(i_pParam);

                        (m_pTargetObject->*m_pMethod)(oParam);
                }
        }

private: // data members
        Class* m_pTargetObject;
        CallbackFunc m_pMethod;
};
```

REGISTRYEVENTHANDLER

Nell’engine sono stati implementati 3 tipi di handler:

- Handler senza parametri
- Handler con parametro const char*
- Handler con parametro generico

REGISTRYEVENTPUBLISHER

Per poter usufruire degli Handler abbiamo bisogno di un oggetto che sollevi l'evento.

L'oggetto che si occupa di ciò è il

"RegistryEventPublisher"

REGISTRYEVENTPUBLISHER

E’ formato solo da:

- Un ObjectID che definisce il path dell’evento che si vuole sollevare
- Una funzione Raise la quale si occupa di sollevare l’evento

REGISTRYEVENTPUBLISHER

Il funzionamento consiste in:

- Creare l’oggetto
- Settare il path dell’evento
- Chiamare la Raise passando eventualmente l’ oggetto di tipo desiderato

REGISTRYEVENTPUBLISHER

```
template < typename T >
class RegistryEventPublisher {
public:
	RegistryEventPublisher();
	virtual ~RegistryEventPublisher();

	void SetPath(const ObjectId& i_oEventPath);
	const ObjectId& GetPath() const;

	void Raise(const T& i_pParam)  {
		MGDMap<ObjectId, EventRegistry::EventHandler*>::iterator it = EventRegistry::GetSingleton().EditRegistry().find(GetPath(
		if (it != EventRegistry::GetSingleton().GetRegistry().end()) {
			EventRegistry::EventHandler* pNode = (*it).second;
			while (pNode)  {
				pNode->Invoke(T::ID, static_cast<void*>(&const_cast<T&>(i_pParam)));
				pNode = pNode->GetNextHandler();
			}
		}
	}
private:
	ObjectId	m_oEventPath;
};
```

REGISTRYEVENTPUBLISHER

```
template < typename T >
class RegistryEventPublisher {
public:
        RegistryEventPublisher();
        virtual ~RegistryEventPublisher();

        void SetPath(const ObjectId& i_oEventPath);
        const ObjectId& GetPath() const;


        void Raise(const T& i_pParam)     {
                MGDMap<ObjectId, EventRegistry::EventHandler*>::iterator it =
                                        EventRegistry::GetSingleton().EditRegistry().find(GetPath());
                if (it != EventRegistry::GetSingleton().GetRegistry().end()) {
                        EventRegistry::EventHandler* pNode = (*it).second;
                        while (pNode)     {
                                pNode->Invoke(T::ID, static_cast<void*>(&const_cast<T&>(i_pParam)));
                                pNode = pNode->GetNextHandler();
                        }
                }
        }
private:
        ObjectId      m_oEventPath;
};
```

REGISTRYEVENTPUBLISHER

Ne abbiamo di 3 tipi:

- Senza parametri
- Con parametro const char*
- Con parametro generico

SISTEMA AD EVENTI

Quali sono i benefici ed i malefici di questo sistema?

BENEFICI

MALEFICI

SISTEMA AD EVENTI

Quali sono i benefici ed i malefici di questo sistema?

BENEFICI

- Possibilità di comunicare tra componenti in maniera semplice e centralizzata
- Disaccoppiamento
- Comunicazione con 1 o molti listener

MALEFICI

SISTEMA AD EVENTI

Quali sono i benefici ed i malefici di questo sistema?

BENEFICI

- Possibilità di comunicare tra componenti in maniera semplice e centralizzata
- Disaccoppiamento
- Comunicazione con 1 o molti listener

MALEFICI

- Complessità nel debug
- Comunicazione non performante
- Non controllo del flusso di eventi che vengono lanciati ed eseguiti

ESERCITAZIONE

Proviamo a far comunicare due entità utilizzando il sistema appena descritto.

Cose da fare:

- Creare 2 oggetti i quali devono comunicare
- Creare un handler
- Creare un publisher
- Creare il link di comunicazione (Lanciare l’evento)

RIEPILOGO

Abbiamo visto:

- Managers
- Sistema di LOG
- Sistema ad eventi utilizzato per la comunicazione

SISTEMI

Tutti i sistemi derivano da un’interfaccia “System” la quale fornisce due principali funzioni:

- Init
- Update

Ogni sistema implementerà queste funzioni in base al proprio scopo

SISTEMI

```
class System
{
public:
	ID_DECLARATION;

	System(uint8 i_uiUpdatePhase = 0);
	virtual ~System();

	virtual void	Init() = 0;
	virtual void	Update(real i_fFrametime, real i_fTimestep) = 0;

	//restituisce la fase in cui verrà invocata la update
	uint8			GetUpdatePhase() const;

private:
	uint8		m_uiUpdatePhase;
	ObjectId	m_oSystemId;
};
```

RENDERING SYSTEM

Il sistema di rendering si occupa di aggiornare tutti i componenti che si occupano della rappresentazione grafica dell’oggetto.

RENDERING SYSTEM

```
class RenderingSystem : public System
{
public:
	ID_DECLARATION;

	RenderingSystem(Ogre::RenderWindow* i_pMainWindow,
	                Ogre::SceneManager* i_pSceneManager);
	virtual ~RenderingSystem();

	virtual void   Init();
	virtual void   Update(real i_fFrametime, real i_fTimestep);
…
}
```

RENDERING SYSTEM

Oltre alle funzioni dell’interfaccia da cui deriva, implementa anche delle funzioni legate alla creazione e distruzione dei componenti di cui si occupa.

Questo ci permette di essere sempre aggiornati sulla situazione dei componenti

RENDERING SYSTEM

```
class RenderingSystem : public System {
public:
…
        void CreateCameras();
        void CreateLights();
        void CreateMesh();
        void CreateBaseGfx();

        void DeleteCameras(const char* i_szName);
        void DeleteLights(const char* i_szName);
        void DeleteMesh(const char* i_szName);
        void DeleteBaseGfx(const char* i_szName);

…
```

RENDERING SYSTEM

```
...

    RegistryEventHandler<RenderingSystem> m_oMeshGfxEvent;
    RegistryEventHandler<RenderingSystem> m_oBaseGfxEvent;
    RegistryEventHandler<RenderingSystem> m_oCameraEvent;
    RegistryEventHandler<RenderingSystem> m_oLightEvent;

    RegistryEventHandler<RenderingSystem, const char*> m_oDeleteMeshGfxEvent;
    RegistryEventHandler<RenderingSystem, const char*> m_oDeleteBaseGfxEvent;
    RegistryEventHandler<RenderingSystem, const char*> m_oDeleteCameraEvent;
    RegistryEventHandler<RenderingSystem, const char*> m_oDeleteLightEvent;

...
}
```

RENDERING SYSTEM

Per questioni di performance il sistema si mantiene un puntatore ai componenti che deve aggiornare senza doverli richiedere a frame-time al “SystemManager”

RENDERING SYSTEM

Nella creazione delle varie entità viene salvato in determinate strutture il componente utilizzato nelle funzioni chiamate a frametime

```
MGDMap<std::string, const TransformationComponent*> m_vMeshGfx;
MGDMap<std::string, const TransformationComponent*> m_vBaseGfx;
MGDMap<std::string, CameraInfo> m_vCameras;
MGDMap<std::string, LightInfo> m_vLights;
```

RENDERING SYSTEM

```
void RenderingSystem::UpdateMeshGfx() {
	if(m_pSceneManager) {
		for (	MGDMap<string,const TransformationComponent*>::iterator ecIt = m_vMeshGfx.begin();
			ecIt != m_vMeshGfx.end();
			++ecIt) {

			Ogre::SceneNode* pNode = m_pSceneManager->getSceneNode(ecIt->first);
			if (pNode) {
				const TransformationComponent* pTransformationComponent = ecIt->second;
				if (pTransformationComponent) {
					pNode->setPosition(pTransformationComponent->GetPosition());
					pNode->setOrientation(pTransformationComponent->GetRotation());
					pNode->setScale(pTransformationComponent->GetScale());
				}
			}
		}
	}
}
```