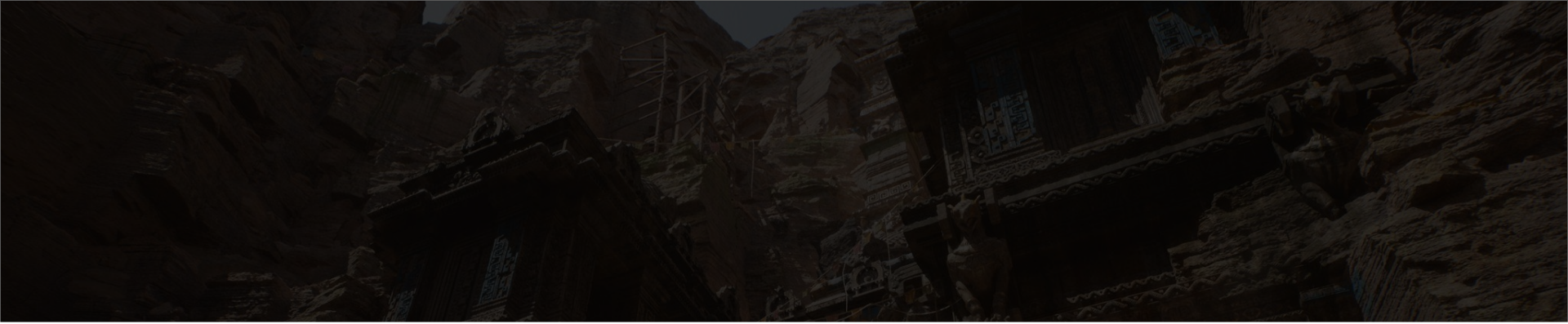
Lesson 3

Scripting & Data

Nicolas Serf
serf.nicolas@gmail.com



Script


Scripting & Data
Context

Scripting & Data

1. Context
2. Encapsulation
3. Component
4. Lifecycle
5. Languages
6. Data structures

Features

Type

Definition

When you write code, you do it in order to develop a new **feature** for your game. It can be as **simple** as developing a new quest, or as complex as creating the quest system underneath

Scripting vs Programming

The distinction is sometimes quite difficult, but generally speaking, we say **programming** when we are **coding** in a **low level language**, while **scripting** is using the things **made** in that **low level**

Properties

When we develop a feature, we'll 100% of the time have some **properties** that will **alter** the features, bring to it different **behavior** based on some values, etc... **Exposing** properly the properties is essential


Scripting & Data
Encapsulation

Scripting & Data

1. Context
2. Encapsulation
3. Component
4. Lifecycle
5. Languages
6. Data structures

Namespace

Platform

Editor

Testing

Referencing

Definition

A **namespace system** is used to **regroup** an amount of **classes** in a **self contained environment**. It allows to bring a layer of **encapsulation** in order to have as little coupling as possible.

Visibility

A namespace comes with a concept of **visibility**. Base on some visibility **properties** both on **classes** and **namespace** level, you'll allows some code to have **visibility** to different **namespace** or not.

Accessibility

Directly coming from the **visibility**, you **cannot access codes** which are from another **namespace**. It is important to notice that because from the accessibility comes the using / **import** of **namespaces**

Scripting & Data

Platform

Namespace

Editor

Testing

Referencing

Definition

The **encapsulation** is important when it comes to **platform**. It allows to **not have some code running** on a **platform** on which it will not be **used** for example.

Exclusion

You can explicitly **excludes** in most engines some module when you are **building** an **executable**. You'll have an interface where you can **specify** the module that needs to be added or not.

Modules

Module is like a **wrapper** around **namespaces**. A module is basically a **source folder code**. You'll find a lot of modules both from engines and game. It allows to have another level of encapsulation.

Scripting & Data

Editor

Namespace

Platform

Testing

Referencing

Definition

It is important to **encapsulate** some code when it comes to **editor**. Most engine provide some code to **interact** with editor, but this code **must not** be in the final **executable**, because editor doesn't exists.

Customization

Customization is one of the most **obvious** aspect about encapsulation. When you are developing some code to **enhanced** the editor, you **don't want** that to be **shipped** with your game.

Scripting & Data

Testing

Namespace

Platform

Editor

Referencing

Definition

Unit testing is another important aspect to development, and not specifically on **game development**. Again, you'll have **code to run unit tests**, and this one should be **encapsulated** to not be in **executable**.

Mandatory

Some people would disagree, but I think **unit testing pure logical code** is mandatory even in game development. It ensure that your core code is running smoothly when you do some modification.

Runtime vs editor

When it comes to **testing**, you'll have 2 kind of test. **Editor test** that **do not imply** to enter in game. Runtime which imply to **enter in game**, and having a **game context** to be running. This is great because it ensure development wise that you develops **feature encapsulated without big dependencies**.

Scripting & Data

Referencing

Namespace

Platform

Editor

Testing

Definition

When you properly **divide** your code base into **modules** and namespace, you'll have referencing between modules. This is **mandatory** but must be done cleverly to avoid **spider web dependencies**.

Core layer

When it comes to referencing, you must develop at least 2 layers. Core layer is containing all modules that are mandatory to the game, and heavy reference in various places.

Feature layer

On the other hand, **features layer** contains additional stuff, that do **not have big dependencies**, or not at all in best cases. When a feature layer **start** to have big dependencies, consider **moving** it to **core**.


Scripting & Data
Components

Scripting & Data

Encapsulation

Linked

Transform

Properties

Definition

Component are also **tied** to an **encapsulation concept**. Code you'll be **writing** will be **located** in a component that is a reusable piece.

Feature

A component comes with a feature. If you want to ensure **SOLID** principles, you need to be sure that **1 component = 1 feature**,

Reusable

By nature, a **component** is something you can attach to any **object living** in the world. It makes it a **reusable piece of code** to bring any functionalities to an object.

Scripting & Data

Linked

Encapsulation

Transform

Properties

Definition

By the nature of component, there is a link between an actor and the component. This link **allows** to easily **retrieve** a **component** from the object, and **vice versa**.

Dependency

You must be **careful** about **creating dependency** in that situation. Because **component** can **require** other **component**, or **object** not **working anymore** if a component is missing. It can leads to **issues**

Scripting & Data

Transform

Encapsulation

Linked

Properties

Definition

Based on engine, some component have **transform**, while other doesn't and a just **linked** the **object's transform** on which they are applied.

Mandatory

Even if a **component don't have a personal transform**, it is still **mandatory**, and by **essence**, every object **has a transform** component, on which other component can be linked to

Scripting & Data

Definition

If you remember, I've talked about **properties** for features. **Component** are the way to expose the properties of your features. You'll see properties from detail panel linked to a component

Designers

This is the only interface you'll have with **designers** in order to let them modify the properties. Comes with that, you'll need to write a **documentation** about how the **properties** are **affecting** the feature.


Scripting & Data
Lifecycle

Scripting & Data

Begin

End

Update

Game loop

Definition

begin is the starting phase of any actors and components. This is basically where initialization and creation are made in order to have everything needed during runtime.

Multiple levels

It depends on **engines**, but there is various **begin level**, going from a **pre-init**, **component initialization**, **awaking of object**, **proper start**, etc...

Access dependency

Because there is **multiple levels** of **initialization**, you must be careful when you try to **access** some **components**, because it is possible that they are not **already created** or **initialized**.

Scripting & Data

End

Begin

Update

Game loop

Definition

End is obviously the **opposite** of **begin**. It is something that is **called** in **various situation** like when the **game exit**, when an **object** is **destroyed**, etc...

Multiple levels

Just like **begin**, there is **multiple** level of end. You can be in **pre-destroy**, **destroy**, **post-destroy state**, etc... It again depends on engine.

Memory

Something to be considered is **memory**. Given most game engine comes with a **garbage collector**, some reference will become **invalid** + you must be sure to **keep** a **reference** on **objects**.

Scripting & Data

Update

Begin

End

Game loop

Definition

Finally, another important part of the **lifecycle** is the **update**. It is a function that will be called **inside** the **game loop**, and based on engine, there is **multiple level** of **update** (**Physic**, **rendering**, **late**, etc...)

Delta time

You'll find in **every engine**, with the update method, a parameter which is called **Delta time**. basically, it is a **float value** that indicate **how many time has elapsed** from the **last update**.

Framerate dependency

Just like explained above, update is **framerate dependant**, because if you have **low fps**, it will be called **less often**. This is why you need to use the **delta time** as a scaling **standard**.

Scripting & Data

Game loop

Begin

End

Update

Physic update

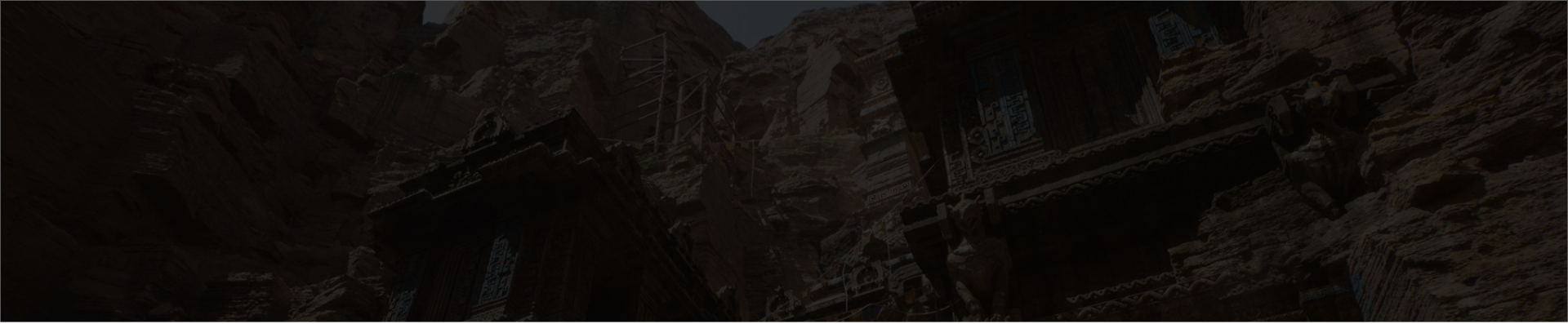
Coming first : physics update. It obviously drives the physics engine and ensure that every object are moved in that frame, according to physics. **Animations**, **bodies**, **collisions**, etc.. are in it

Gameplay update

The **gameplay update** can be also composed of multiple update functions, but this is the most common one on which you'll be **doing most stuff** that should be computed on **tick**.

Rendering update

Last but not least, the **rendering update** will make sure that everything that has been **computed** during this frame is properly **displayed** on the screen. This is where you can **modify** the **rendering** of some things



Scripting & Data

Languages

Scripting & Data

Usage

Accessibility

High-level

Restriction

Definition

This is most likely the most **important** aspect when it comes to **choosing** the **language**. Defines the **usage** and how **important** will be scripting / programming part in the game..

Goals

Usage comes with **goals**, you must know how your **languages** will be used, only by **devs** ? By **designers** ? For **quests** ? For **AIs**, etc... it will **lead** the **direction** of what needs to be used

Performances

It is quite obvious, but a **scripting language** will most likely be **interpreted**, so it will be **less optimized**. You should not do any **big treatment** on scripting side. **Programming** on the other side is **fine**

Scripting & Data

Accessibility

Usage

High-level

Restriction

Definition

There is always a **trade-off** when it comes to choosing what belongs to **programming** and what belongs to **scripting**. This is all tied to **accessibility** of the language and the features.

Complexity

Programming is more complex, and low level features will be develops with it, scripting must use features made in the **programming** side. You must defines this **layers** of **complexity** in your **architecture**.

Visual

A lot of game engine now offers a **visual solution** for **scripting**. It is important for **designers** because it **abstract** completely the **code complexity** and increase **productivity**.

Scripting & Data

High-level

Usage

Accessibility

Restriction

Definition

We've been **speaking** about **low level stuff** or **high level**. It is important to understand that it differs from **feature** to **feature**. Some may require only a **tiny layer of high level**, while other will require **a lot**.

Behaviors & Interactions

As a general rule of thumb, behaviors and interactions will be on the scripting side. For example, **listening for a collision**, changing the **color on interact**, etc...

Scripting & Data

Restriction

Usage

Accessibility

High-level

Definition

We talked about **tradeoffs**, with it come obviously **restriction**. Generally speaking, programming will be **restriction-less**, except the one **implied** by the **engine**. **Scripting** on the other side is more **restrictive**.

APIs

For example, most APIs are coded only in **languages**, like the **console's ones**. You cannot access thus from **scripting**, or you'll need to create **wrappers** around it.

Wrappers

We've seen **wrappers** concept, and on the scripting side, it will **often** be **needed**. You'll expose some **functionalities** that are only on **programming** side, to be **used** on **scripting** side.

Scripting & Data

Data structures

Scripting & Data

Container

Configuration

Externalize

Import

Definition

Data structures are mostly **containers**, they are a **set** of **properties** that will be used by systems. It is **important** to ensure that most of datas are **externalize**.

Types

There is a lot of **different type** of **data structure**, and it depends on **engine**, so we'll not dive into details here, but obviously, it'll be based on some well known type like **struct**, **classes**, etc...

Encapsulation

Encapsulating datas into a **data structure** ensure that a system is **independent** on **implementation**, we can change the **code**, the datas still **exists** in an **external storage**.

Scripting & Data

Configuration

Container

Externalize

Import

Definition

Data structures are often used for **configuration** both for **engine** and **game**. If you take **Unity** and the new **rendering pipeline**, they heavy use data structure for **rendering configuration**, **localization**, etc..

Scripting & Data

Externalize

Container

Configuration

Import

Definition

An important point about a data structure is that it is an **asset** inside the engine, so that can be **cooked** easily, and easily **modified** and **observable** from the engine, but still a collection of values.

Export

By the simple nature of being a **collection** of **value**, it is quite easy to **externalize** a data structure in order to export it as a readable format from outside the engine like **JSON** or **XML**

Working outside

The **exportable** nature makes it possible for **designers** or even **programmers** to **not have to open** the **editor** in order to **modify** some datas about the game.

Scripting & Data

Import

Container

Configuration

Externalize

Definition

If we can externalize, we can obviously **import**. This is also a **processus** heavily used. We can see two **type** of **import** when we are **treating** datas.

Runtime process

If you decide for a **runtime** process, you'll need to code a **parser** that will be reading a **text file** written in **JSON** for example, and import the **values** into the **component** for example to feed the **properties**.

Editor process

The other way is to have an **editor piece** of **code** that will allows to **import** an external format like **JSON**, and convert it into an **asset data structure** understandable from the engine.

Unity

Unity

Monobehavior

Awake
OnEnable
Reset (Editor)
Start

FixedUpdate
OnTriggerEnter / OnCollisionEnter
OnTriggerStay / OnCollisionStay
OnTriggerExit / OnCollisionExit
Update
LateUpdate

OnApplicationQuit
OnDisable
OnDestroy

DontDestroyOnLoad

enabled
gameObject
transform
name

GetComponents<XXX>()
GetComponentsInChildren<XXX>()
GetComponentsInParent<XXX>()
AddComponent<XXX>()

[SerializeField]
[System.NonSerialized]
[HideInInspector]
[RequireComponent(typeof(RigidBody))]

https://github.com/teebarjunk/Unity-Built-In-Attributes

Unity

Scriptable Object

Awake
OnEnable
Reset (Editor)

OnDisable
OnDestroy

CreateInstance

[CreateAssetMenu(menuName = "My ScriptableObject", order = 100)]

Live demonstration

? Questions ?