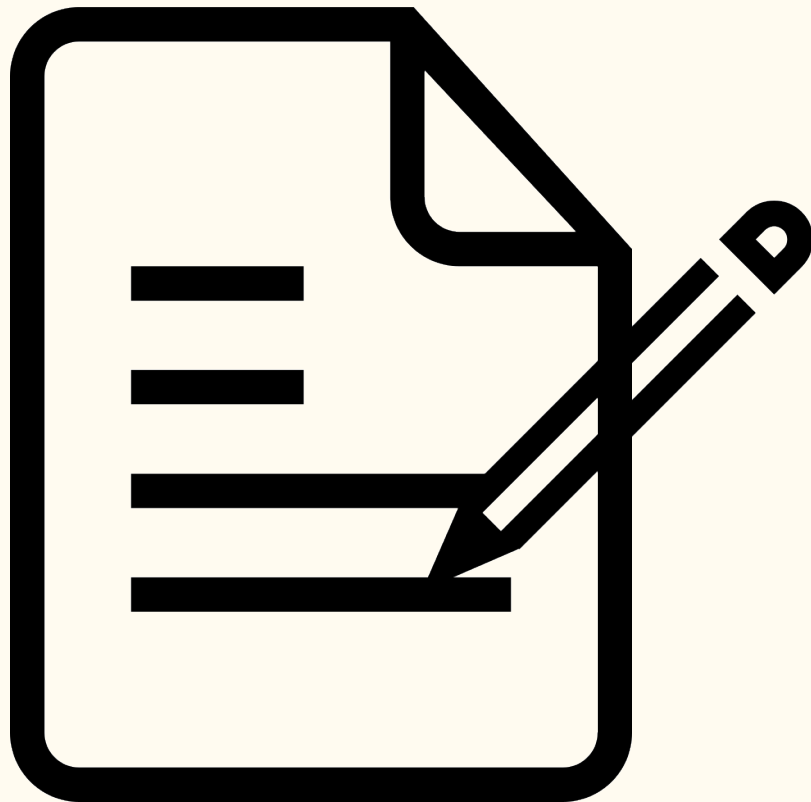


Unreal Engine 5 - Lesson 7 - UI & UMG

Nicolas Serf - Gameplay Programmer - Wolcen Studio
serf.nicolas@gmail.com

Summary

- **HUD**
- **UMG**
 - **Designer View**
 - **Nesting and Slots**
 - **Anchors**
 - **Clipping**
- **User Widget**
- **C++ for UI**
 - **UMG and Slate**
 - **UserWidget in C++**
 - **UWidget in C++**
 - **Slate in C++**
- **Optimization**
 - **Event Driven Update**
 - **Loading & Construction**
 - **Layout & Positioning**



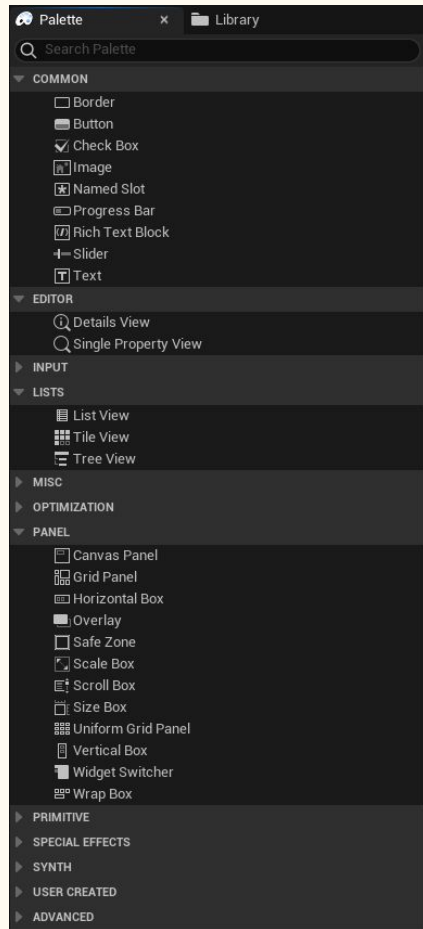
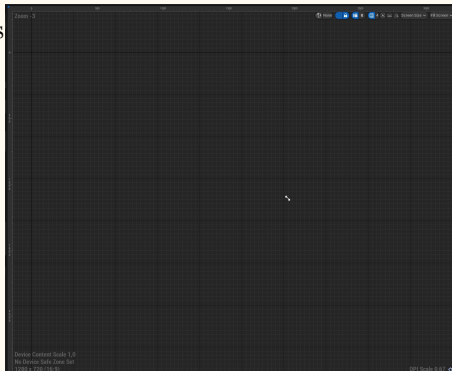
HUD

- First things first, we'll talk about the **HUD** standing for **Head Up Display**
- HUD is basically the **interface** that **allow** your game to **transfer information** from the **gameplay** to the **player** in front of the screen
 - Health bar
 - Experience bar
 - Skills
 - Etc...
- Most **likely**, **HUD** is a **cosmetic only** thing that is always display on screen and is **free of user interaction**
 - **Inventory** is for example **not an HUD thing** as it requires **interaction** with player and is not always displayed
- It is important to understand that **AHUD** is a **legacy class** from **UE3** and **UDK**
 - It was **created before UMG** get introduced and facilitate by a lot interface development on the engine
 - It is kind of **replace** by **UMG** but it is still **part of the engine** and **gameplay framework**
 - It has some **uses**
- **HUD** can be accessed from anywhere as it is part of the **gamemode** for the **CDO**, and part of the **player controller** as the **instantiation**



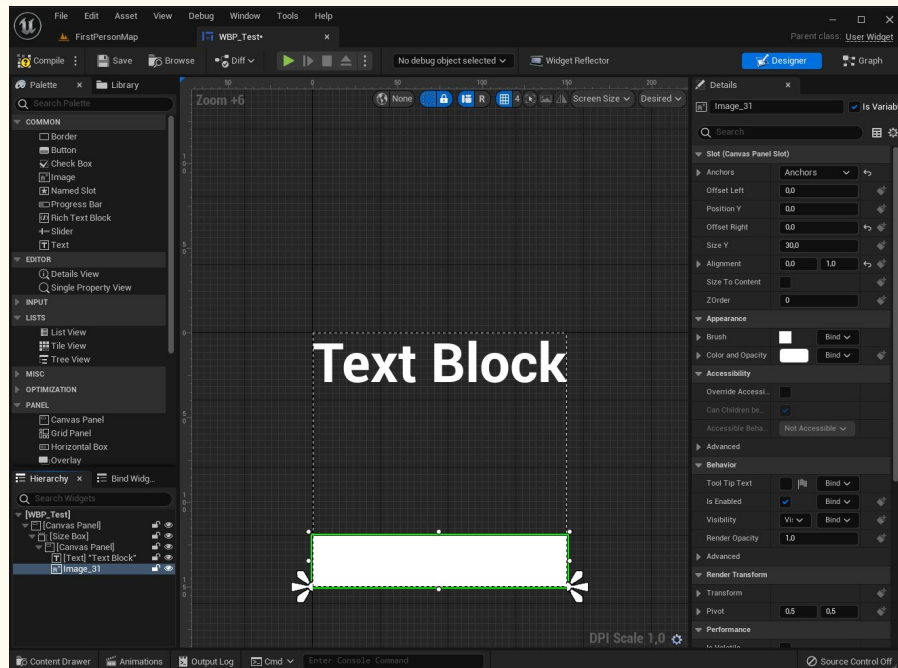
UMG

- **UMG** stands for **Unreal Motion Graphics**
- UMG is the name given by Unreal to there **latest UI system**
- **Before UMG**, UI create was more **cumbersome** and **less visual**
- It is **built on top** of the already existing **Slate system**
 - Slate being the **old system** doesn't mean it's **obsolete**. It is needed for **more complex functionality** for UIs
 - **Unreal editor** is also **built** on **slate**
- UMG system can feel quite **daunting** at first with all options
- It **offers** a proper **interface** in order to create your user interface :
The Designer View
- It offers a **Graph View** like **standard blueprint**, which is directly **linked** to the **panel** you create in the **designer view**
 - You can set your **logic** in it
 - You can **control** how the widget **behave**
 - Etc...



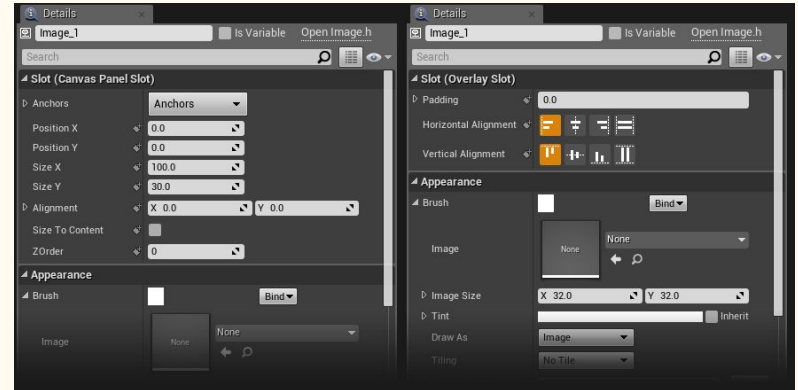
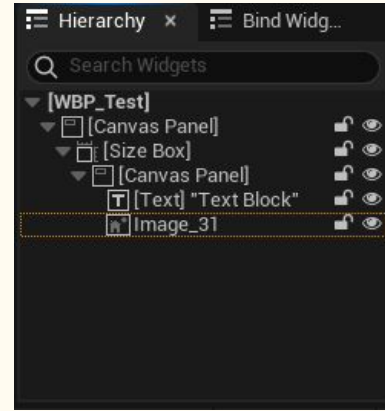
Designer View

- Let's see how the designer view is composed
- **Palette**
 - It lists all **widgets** that are available to you
 - Keep in mind that you can **create custom widget** which will also appears here
 - From Engine's widget are already **stored** in section
- **Hierarchy**
 - It is the treeview of your **UserWidget**
 - **UserWidget's root** is at the very **top** and can't be **alter**
 - Widget in bold are "IsVariable" widget
 - It means that this **widget will appears** in the **Graph view** variable and be accessible to **interact** from your **blueprint code**
- **Viewport**
 - It is where you **design** the layout of your UI
 - A **selected widget** will be shown with **green border**
- **Details**
 - It shows **property according** to the **selected widget**
 - It will **differs** from **widget to widget**
 - It is much like a standard **detail panel**
- **Animations**
 - It list animations within the current **UserWidget** and **Timeline** allows to show the **currently selected anim**



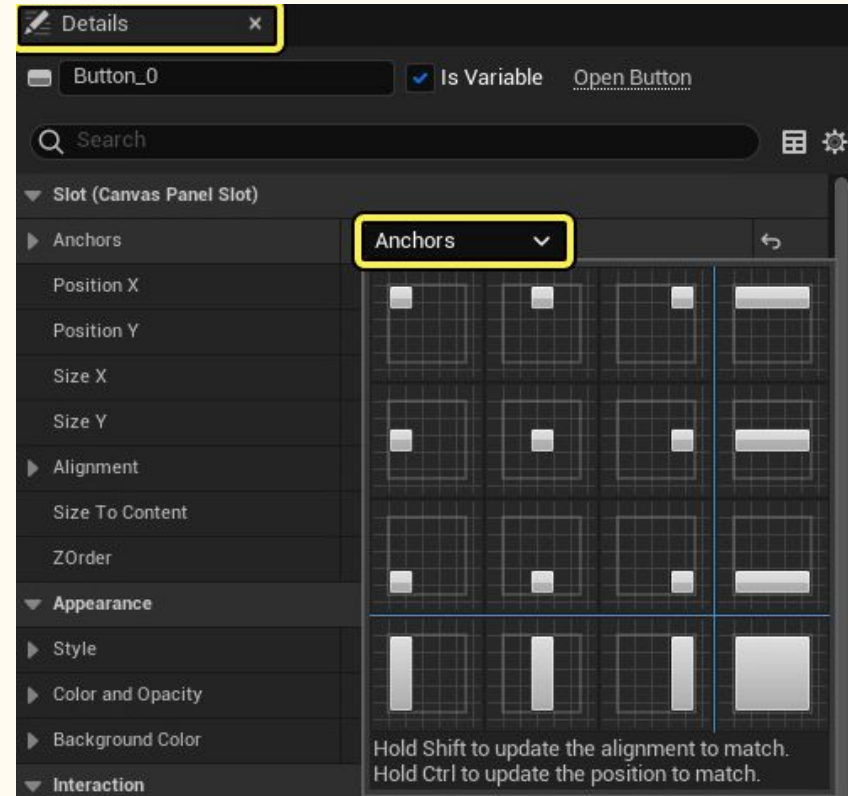
Nesting and Slots

- Nesting is the **core system** to understand how **widget** behave all **together**
- UMG is a **tree of nested widgets**. When you put a widget inside another widget, we control how they are **arranged** on the screen
 - There is a **parent - child** relation just like actors that get **created**
- This nesting relation count **vary depending** on the **widget**
 - 0 : The parent **cannot** contain children (Image, Text Block, etc...)
 - 1 : The parent can contain **at most one** child (Root, Border, NamedSlot, etc...)
 - Many : The parent can contain **many children** (CanvasPanel, Overlay, WidgetSwitcher, HorizontalBox, etc...)
- When a **widget** is inside another, we can **custom** how it **behave inside** its parent through **Slot property**. There is 2 possible slot type
 - **Overlay** : We can set how **stretched** or **aligned** it is according to parent
 - **Canvas** : We can set at **pixel level** how **offsetted** and **anchored** it is
- Slotting system is fundamental for organisation but also when it comes to responsivity of UIs



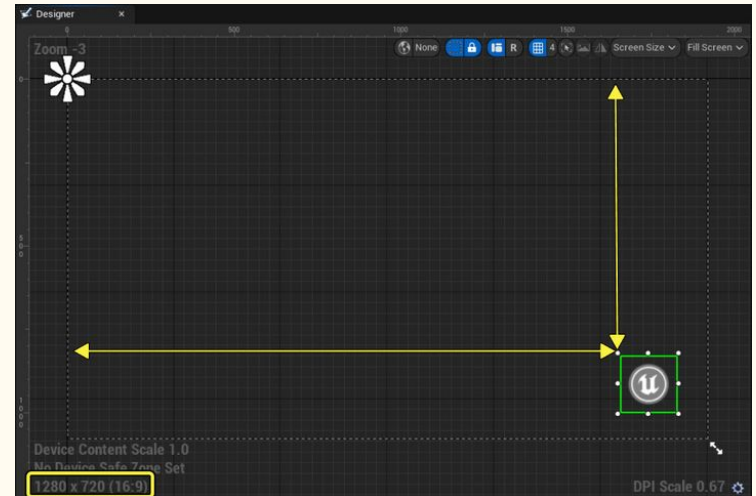
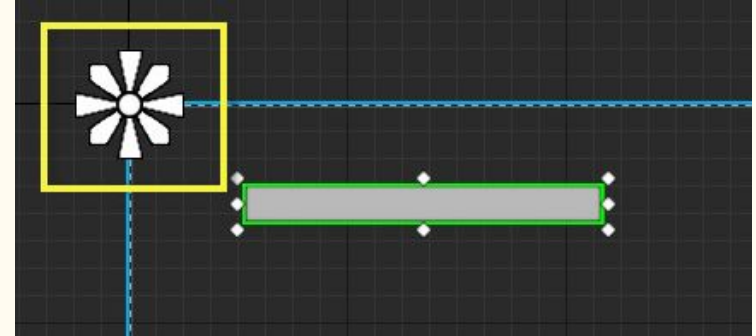
Anchors

- Use **Anchors** to set **location** of the UI widgets on a **Canvas Panel**. The Anchors supports **settings** with **different screen sizes** and **aspect ratios**.
- **Minimum (X,Y)** and **Maximum (X,Y)** Anchors parameters and **offset** parameters determine **location** of each **widget**.
- You are able to select from a number of **Anchors presets** or set up it manually by the **Minimum (X,Y)** and **Maximum (X,Y)** parameters (where **Min (0,0)** and **Max (0,0)** determine the **upper left corner** of the Canvas Panel; **Min(1,1)** and **Max(1,1)** determine the **bottom right corner** of the Canvas Panel). Anchors presets differ in the set of offset parameters.



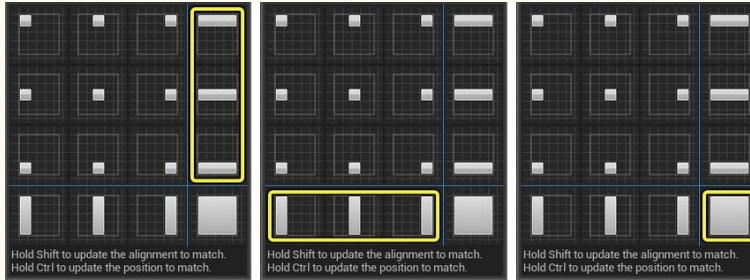
Anchors Medallion

- **Anchor Medallion** shows the **Anchor position** in the Canvas Panel.
- By **default**, when you place an **element** into a **canvas**, it uses default settings of Anchors (**top left corner** location).
- The **horizontal yellow line** is **X-axis Button offset**. It determines **distance** in **Slate Units** from **Anchor Medallion** to the **Image** in the **X-direction**.
- The **vertical yellow line** is **Y-axis Button offset**. It determines **distance** in **Slate Units** from **Anchor Medallion** to the **Image** in the **Y-direction**.
- The **offset parameters** based on **Canvas Panel size** and **adapt to size** changes.
- Click the **Screen Size Button** in the graph to **change** the **currently used size**. It is very useful to test the **UI widget layout** with **different screen sizes** or **aspect ratios** and adjust accordingly.
- It is important to take into **account differences** in **device screen sizes** and **aspect ratios**, when you set Anchors and offset parameters of the widgets. You should **avoid shifting widget** out of the **viewport**. It could happen in case **inappropriate Anchors settings** for some **screen sizes**.

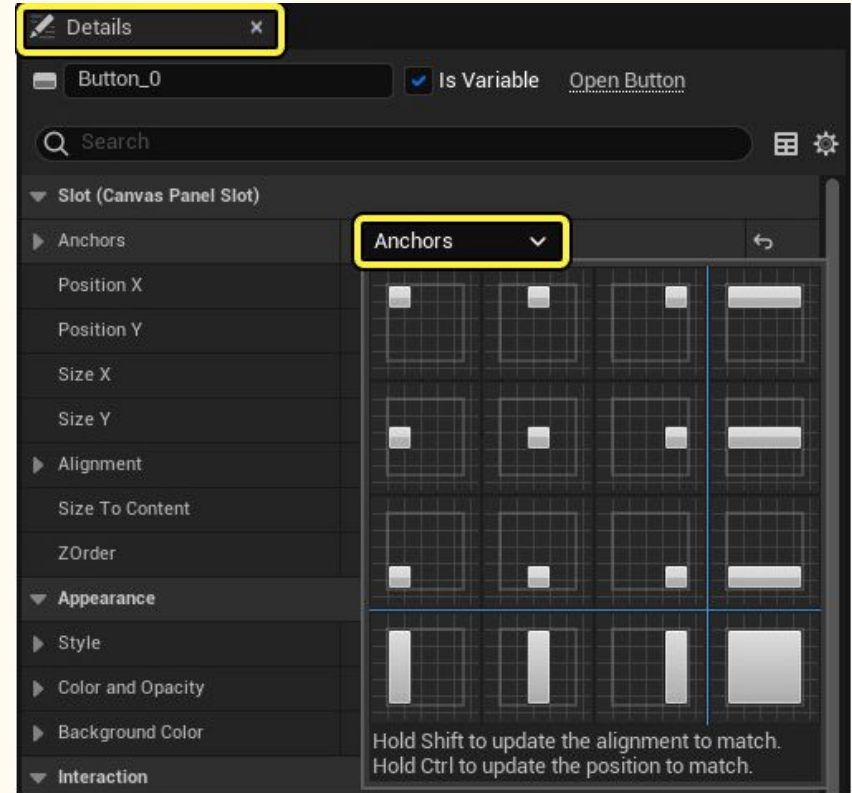


Anchors Preset

- **Preset Anchors** is the **most common** method of Anchor point setting for widgets. With the help of this, you will be able to **cover most of your needs** in setting position of UI widget.
- Select **preset** from **Anchors drop-down** window at the **details panel**. Each preset determines the **Anchor point location**. The silver box marks this **location**.
- There are also **preset stretching** methods of the Anchor Medallion, when it is **split into several components**.

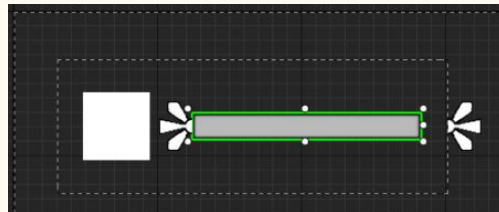


- With the help of this, you will be able set up **widget to stretch** along with the viewport, based on **screen size**.



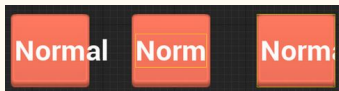
Anchors Manual

- In some cases, it is **necessary** to set **Anchors manually**. For example, this is useful, when you need to **anchor widgets to each other**.
- **Anchors Manual** are **case to case** dependant and therefore there is **no general explanation** about them
- Keep in mind that it comes with **medaillon** placement and **splitting it** according to your needs

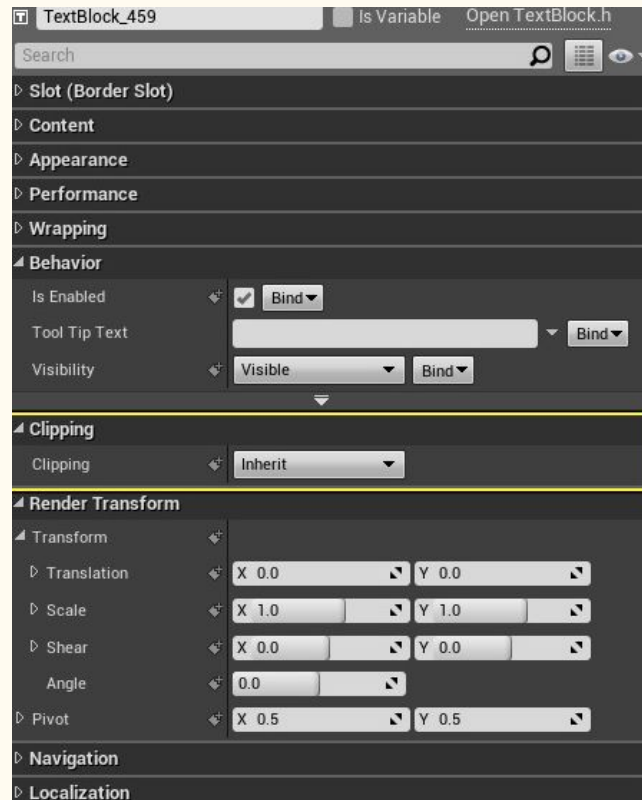


Clipping

- The **clipping system** in UMG uses **Slate's Clipping System** as a framework to control how text, images, or content is shown for **Widgets** (as well as the rest of the Editor). Clipping works by **restricting rendered objects** (graphics and text) to a region using a **bounding box** so that anything outside of it is not shown. The clipping system is now **axis-aligned meaning** that it can clip any rotation, which was not possible before because of the way transforms were handled.

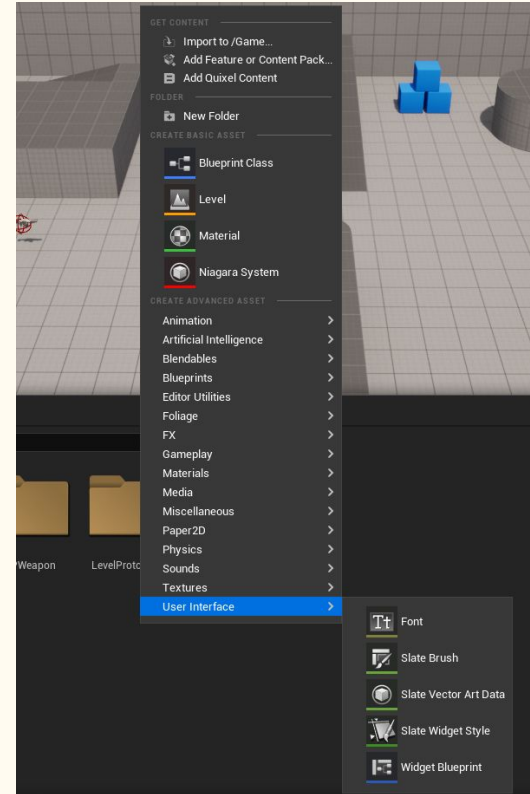


- In this example, each of the buttons is a parent to the **displayed** text. These examples demonstrate **whether** the button or the text is responsible for **clipping**.
 - Left - No clipping is enabled on the button or the text.
 - Center - Clipping is enabled on the text
 - Right - Clipping is enabled on the button.
- You can configure how you set your **clipping** to **behave** from the **details panel**



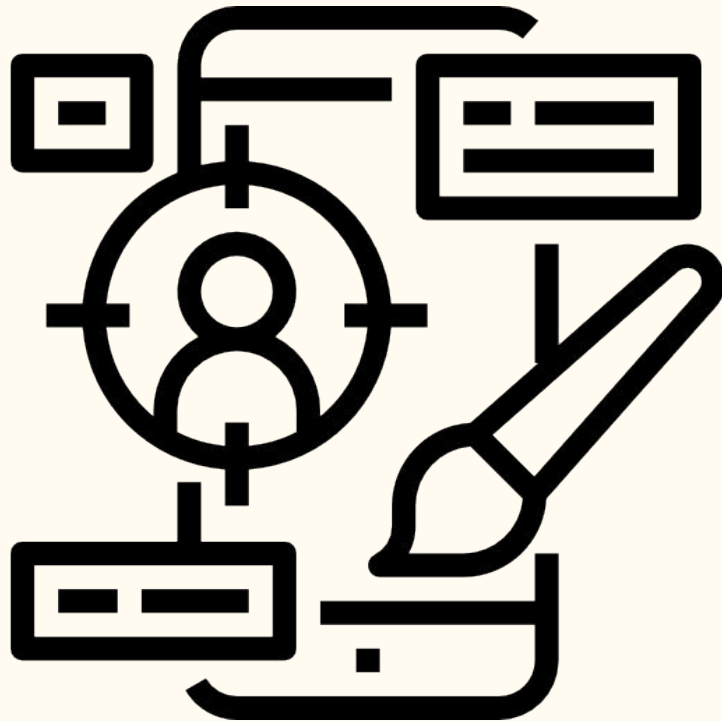
User Widget

- When you are dealing with **UMG**, you are dealing with **UserWidget** asset. It is the **base asset type** that is use to have a **UMG representation into the game**
- In order to **create** a **User Widget**, you simply need to **right click** in the content browser go into **User Interface** and select **Widget Blueprint**
- The naming convention is generally to make every widget blueprint **prefix** with **WBP_**
- **Similar** to the way **Blueprint classes** are used to **create reusable object**, **UserWidgets** are used to **create reusable objects** with custom logic about user interfaces.
- Some examples of **UserWidgets**
 - A **custom button** that includes a **Button** instance, a **TextBlock** for the label, and a **general behavior** that needs to be **shared by all button from your game**
 - An **health bar**
 - A **tooltip** when you are **hovering** an item in the **inventory**
- It is your role as an **UI programmer** to decide if you need to **create** a new User Widget or **implement** the UI directly into the **parent widget**
- It is also your choice to decide if you need **to** implement the base class as a **blueprint** or **c++ class**



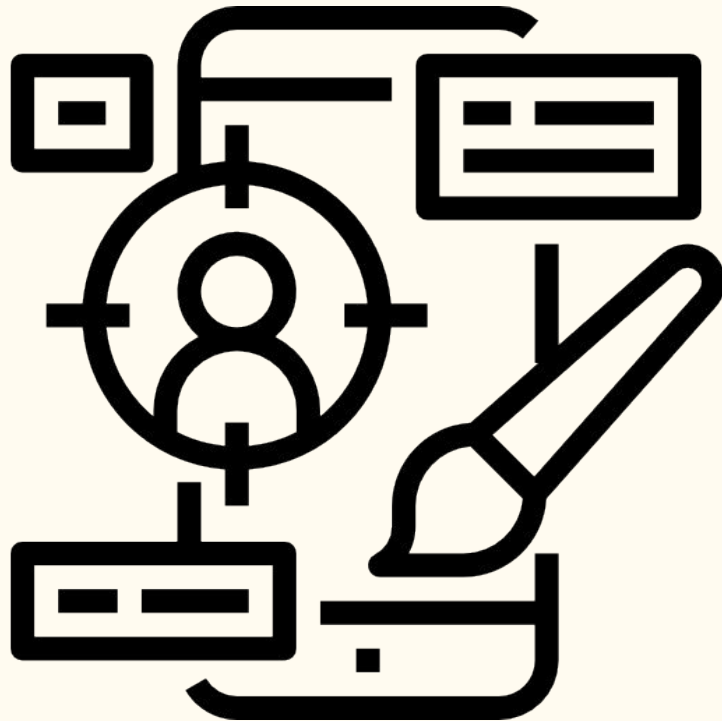
C++ for UI

- It is possible to **entirely** create a **UI** for a game in **blueprint**
- However, **creating a professional-level UIs** will require to use **C++**
 - May be **needed to communicate on c++ level** even if it is **generally a bad idea to couple UI and gameplay** in two direction
 - You are likely to **hit performance problems** in large Blueprint-based UI. Especially for **blueprint called every frame with complex code**
 - Blueprint is quite **verbose**, and **complicated logic** is a **nightmare to maintain**, leading to **spaghetti graph**
 - Blueprint will more **easily leads to no separation of acquisition, processing and data presentation**. Blueprint having access to **everything**, it is more likely that you'll **end-up mixing** in the same graph **data acquisition, formatting and displaying**
 - Blueprint are **binary assets**, meaning they're **impossible to merge** so only **1 person** can work at the same time. It means that if the **UI developer** want to update the **blueprint logic** and **UI artist** change some **appearance**, it is **not possible at the same time**
- There is hundred ways to do the same thing but are **3 ways** of using **C++ in UIs**
 - **Subclassing UUserWidget**
 - **Subclassing** or creating a new **UMG widget**
 - **Subclassing** or creating a new **Slate widget**
- **Generally speaking**, it is a **common approach** to declare the class in **C++** to handle **logic** in it, and implement **details** in the **BP inheriting** from **that c++ class**, just like gameplay



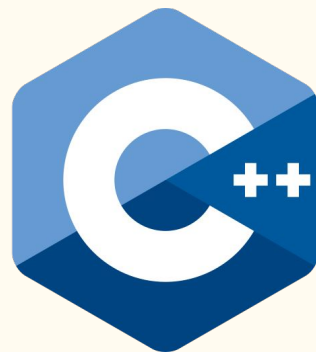
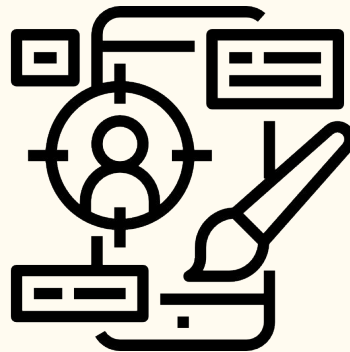
Important point about UMG and slate

- I'll emphasize again that **UMG is built on top of Slate**.
- **Before** version **4.0** of the engine, the system called **Slate** was the **standard**.
- It was **designed** to make **UI creation in C++** as simple as possible
- UMG is a simple **wrapper** around **Slate**, to make it **easier** to use from **blueprint**, and let **UI designers create custom UIs** from the **editor** with a direct visual feedback
- This direct **implication** means that **everything** you'll find in UMG will exists in Slate as it is a simple wrapper
 - For example, a **UImage** which is the **UMG** class wrapper contains a **SImage** instance inside it
 - The **Slate** class handles **most of the logic** while the **UMG** allow to display it in the editor and play with it
- Generally speaking the **lower** you goes into the **UI hierarchy**, meaning that you get **closer** to **Slate**, the **more customization** you'll be able to do but the **more work** it may involve



Create a UserWidget in C++

- We need to **declare** a **class** that inherit from **UUserWidget**
- **NativeConstruct** function is **similar** to **C++ constructor** but **should be used** instead because it is **integrated** into the **UMG**
 - You'll typically **bind delegates**
 - Set up **default appearance**
 - Etc...
- You'll then create a **UserWidget** in **blueprint**, **inheriting** from the **C++ class** we just created
- Keep in mind that if you have **already created a widget**, you can **reparent** the blueprint



Create a UserWidget in C++ - Bind widget



- A common issue when we are dealing with C++-based UIs is how to control blueprint created widget from C++
- **BindWidget** meta property is the answer

```
UCLASS(Abstract)
class UBindExample : public UUserWidget
{
    GENERATED_BODY()

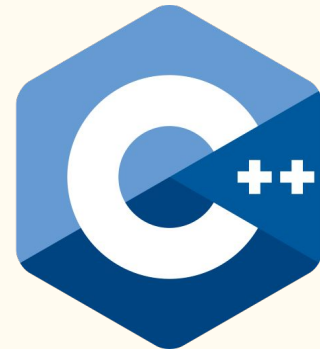
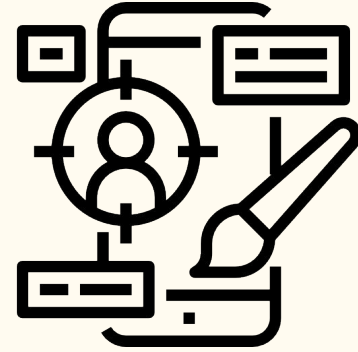
protected:
    virtual void NativeConstruct() override;

    UPROPERTY(BlueprintReadOnly, meta=(BindWidget))
    class UTextBlock* ItemTitle;
};
```

- This meta property will create an **identically-named widget** in a **blueprint subclass** of that c++ class
- As it is **declared in the class**, the blueprint at **runtime** will **access** it from the c++
- You'll see that **after compiling** and **inheriting** from the class, you may have **an error** like **so**

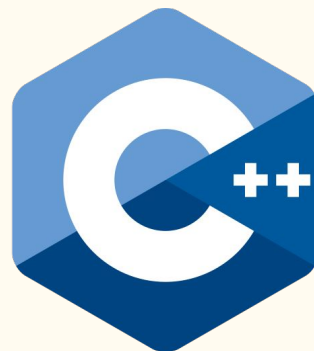
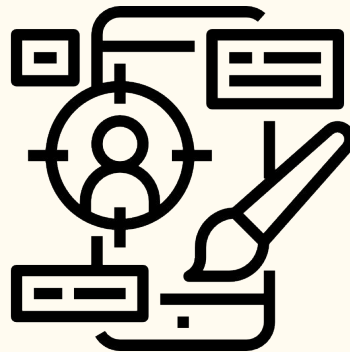
 A required widget binding "ItemTitle" of type  Text was not found. w a widget named ItemTitle

- **BindWidget** is a strong tool for making it **easier to main complex logic** in C++ and **ease collaboration**
- **BindWidgetOptional** is the **same logic** without the compiling error



Create a UserWidget in C++ - Updating UserWidgets

- When you'll be **developing more and more complex UIs**, that may be **very different** to their **final appearance** in-game based on data
- It is possible that you may want to **develop** a **User Widget** that will be used **for multiple visual purpose**, for example in an **inventory**
- You may give a **widget class** in order to know which **representation** needs to be used, but still the visual in the **blueprint viewport editor** will **not** **change**
- In order to do so, you need to use **SynchronizeProperties** function
 - This function comes from **UUserWidget**
 - It is called in **editor every time a property is modified** or blueprint is **compiled**
 - You can **override** it and **initialize** your widget the same way it will be **set up** **in-game**.
- It is worth mentioning that after Unreal 4.16, in Blueprint, the node **PreConstruct** can be used a bit like **SynchronizeProperties**.



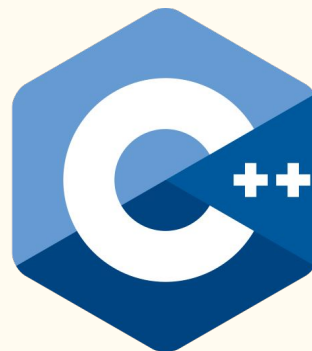
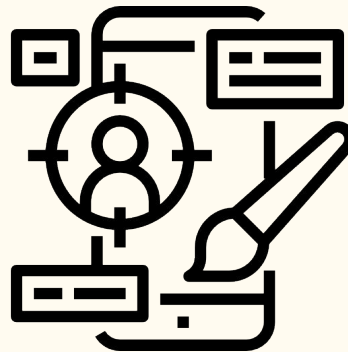
Create a UWidget in C++

- As kind of everything is extendable in the engine, it is also possible to **extend** or **create** a new **UWidget**
- It is the way to go if you want to create **new compound widget** that needs to be re-use **everywhere**
 - For example, a button with an icon and a text
- The function **RebuildWidget()** is the place where the **stuff needs to be done**
- It is directly **requiring** to **write Slate** code in it, as you know, **UUserWidget** are just **wrappers around Slate**.
- This is an example of a slate code for buttons

```
USharedRef<SWidget> UButton::RebuildWidget()
{
    MyButton = SNew(SButton)
        .OnClicked(BIND_UOBJECT_DELEGATE(FOnClicked,
            SlateHandleClicked))
        .OnPressed(BIND_UOBJECT_DELEGATE(FSimpleDelegate,
            SlateHandlePressed))
        .OnReleased(BIND_UOBJECT_DELEGATE(FSimpleDelegate,
            SlateHandleReleased))
        .OnHovered_UObject(this, &ThisClass::SlateHandleHovered)
        .OnUnhovered_UObject(this, &ThisClass::SlateHandleUnhovered)
        .ButtonStyle(&WidgetStyle)
        .ClickMethod(ClickMethod)
        .TouchMethod(TouchMethod)
        .IsFocusable(IsFocusable)
        ;

    if ( GetChildrenCount() > 0 )
    {
        Cast<UButtonSlot>(GetContentsSlot())
            ->BuildSlot(MyButton.ToSharedRef());
    }

    return MyButton.ToSharedRef();
}
```

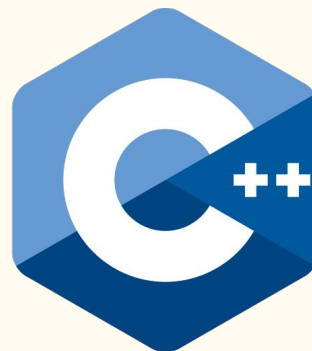
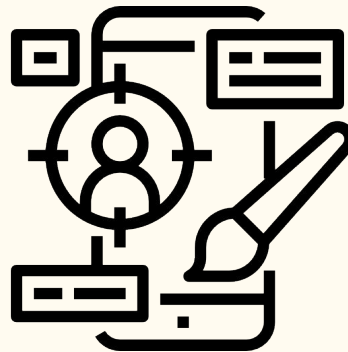


Manipulating Slate from C++

- Because using slate, let's try to pin down some reasons to use it
 - **Creating Editor UI** : Custom **asset inspectors**, **windows**, or other visual tool to be used inside the engine have to be implemented using Slate.
 - In 4.22, **EditorUtilityWidgets** was introduced, which can be written with **UMG + Blueprint**
 - It eases the process but still **asset inspectors** must be written in **Slate**
 - **Implementing low level functionality** : Some things are not **supported** from **UMG**, for example complex **graph-drawing** which may be better written in **Slate** than **UMG**
- A slate code will always look like this

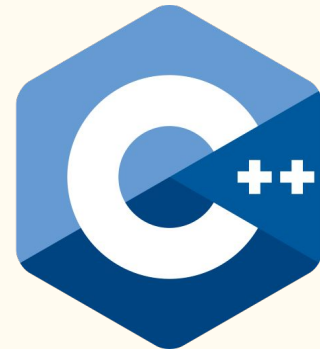
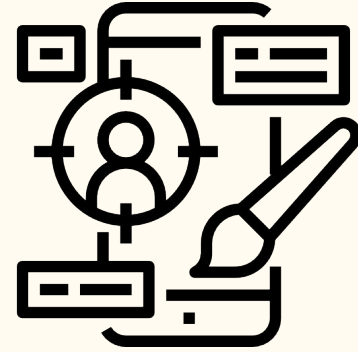
```
SNew( SButton )
+ SButton::Slot()
[
    SNew( SHorizontalBox )
    + SHorizontalBox::Slot()
      .VAlign( VAlign_Center )
      .HAlign( HAlign_Center )
    [
        SNew( SImage )
        .Image( MyIconBrush )
    ]
    + SHorizontalBox::Slot()
      .VAlign( VAlign_Center )
      .HAlign( HAlign_Fill )
    [
        SNew( STextBlock )
        .Text( FText::FromString( "Click me!" ) )
    ]
]
```

- We are declaring here a **button** with an **icon** and **text**, **side by side**
- Last important note, **Slate** is a **module**, as such, it **needs** to be added in **dependencies**



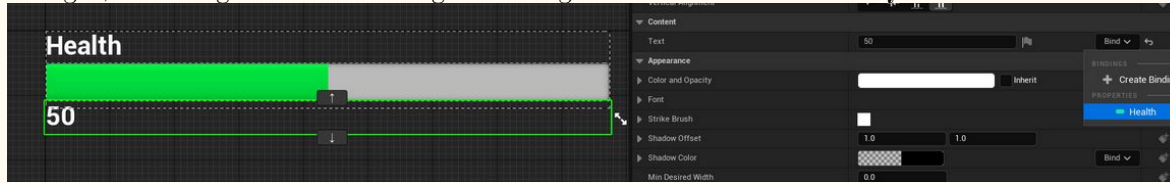
Manipulating Slate from C++

- Let's see some **key functions** when you want to duplicate and **extend** an **existing widget**
 - **Construct** : Is how the slate widget is setup
 - **SLATE_BEGIN_ARGS** : Argument **allowed** by the widget are defined
 - **OnPaint** : It is where the widget defines how it is rendered
- It is **more likely** that if you want to **add** some **specific behavior** to existing **widget**, **extending** one is the way to go
- In this case, do not **hesitate** to check how the **original** was **implemented**, pick up the code and then make your **modifications**
- In some other cases, you may want to create a new Slate widget from scratch, not based on anything, in this case you have a few choices for subclass
 - **SCompoundWidget** : [CompoundWidget](#) is the **base** from which **non-primitive widget** should be built. It has a protected member named **ChildSlot**.
 - **SLeafWidget** : [LeafWidget](#) is a Widget that has **no slots** for children. It is usually **intended** as **building blocks** for aggregate widget. The names speaks for itself, it should be used for **leaf widget** like **SImage**
 - **SPanel** : [SPanel](#) is a Widget that allows to arranges its child widget on the screen. It offers the **slotting system**

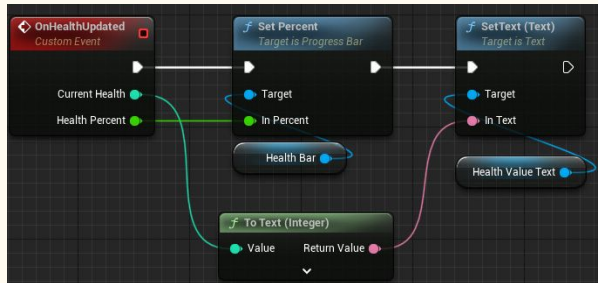


Optimization - Event Driven Update

- Whenever **possible**, you should **avoid** using **On Tick** or **On Paint** to run **logic** in your UI. **Event Dispatchers** and **Delegates** can help you create logic that responds to specific events **without** needing to rely on **Tick**.
- When you **bind attributes to fields** in your UI, they **poll** the **attribute** every frame. For example, if you bind the **Text value** of a **Text Field** to an **integer**, that Progress Bar will assign the **integer's value** to that field **every tick**. This is **inefficient**, so you should avoid using bound attributes.



- Instead, you should set up your project to call **functions and events in your UI** to update these fields. For instance, instead of **binding a health bar** to a **Health attribute**, you should call an **OnHealthChanged** event in your UI's **Blueprint script** to change the necessary fields in your UI.



- While setting these up is more **time consuming**, this ensures that the **UI only changes** on the frame when the value changes instead of **each frame**.

Optimization - Loading & Construction

- **Reduce Unused Widgets As Much as Possible**

- All children **inside a widget** are **loaded** and **constructed** regardless of whether they are **visible**. Even if your UI does **not render them**, they will use **memory** and require **loading** and **construction time**.
- At a basic level, your **UI designers** should regularly check for **any unused widgets** and remove them before **committing** their work. Regular cleanup will help with **organization** as well as **performance**.

- **Break Complex Widgets Into Pieces that can Load at Runtime**

- An especially complex widget for a major system could have well over **1000 children** to **handle** all of its **functionality**, but only need to **display** a few **hundred widgets** at a time. In these situations, if you were to load the entire widget as one big piece, **hundreds of inactive children** would take up **space** without being used. Depending on how specialized these child widgets are, **users** may spend hours **without seeing** them.
- In these cases, you should break up your widgets into categories:
 - Widgets that are **always visible**.
 - Widgets that **must be shown as quickly as possible**.
 - Widgets that **can afford to have a small amount** of latency when shown.
- Any widgets that require **fast response times** should be **loaded** in the **background** even when not **displayed**. For example, an **inventory screen** in a competitive shooter is used very frequently and should be highly responsive due to its **crucial function for the player**, so **keeping it loaded** but not **visible** is good practice.
- **Any widgets** that are not present for **long periods** of time and do not **require fast response** should be **loaded asynchronously** at **runtime** and destroyed when **dismissed**. For complex widgets with diverse functions, this may entail keeping a **base widget loaded** at **all times**, but loading different sets of child widgets asynchronously depending on what mode or function is needed. This can save a **great deal of memory** and reduce the **CPU impact** on initialization.

Optimization - Layout & Positioning

- **Use Canvas Panels Sparingly**
 - The Canvas Panel is a **powerful container widget** that can position other widgets using a **coordinate plane** and **per-widget anchors**. This makes it easy to both position widgets exactly where you want them, and also to maintain widgets' positions relative to the corners, edges, or center of the screen.
 - However, Canvas Panels also have **high performance demands**. **Draw calls in Slate** are grouped by **widgets' Layer IDs**. Other container widgets, such as **Vertical** or **Horizontal Boxes**, **consolidate their child widgets' Layer IDs**, thus reducing the **number of draw calls**. However, Canvas Panels **increment** their child widgets' IDs so they can render on top of one another if need be. This results in Canvas Panels using multiple draw calls, thus making them **highly CPU-intensive** compared with alternatives.
 - As a **rule of thumb**, if your widget consists of a **single element**, you definitely do **not need** a Canvas Panel. Even with full menus and HUDs, you can **often avoid** using Canvas Panels altogether by using **Overlays** and **Size Boxes** together with **Horizontal**, **Vertical**, and **Grid Boxes** to handle layouts.
- **Use Spacers Instead of Size Boxes When Possible**
 - **Size Boxes** use **multiple passes** to calculate their **size** and **render** themselves. If you need content to take up a certain **size** in **width** and **height**, **Spacers** are significantly cheaper
- **Use Rich Text Widgets Sparingly**
 - **Rich Text** widgets provide robust **formatting** for text, but are **very expensive** compared with standard text boxes due to the **wide range** of extra **capabilities** they add. If you want text to be stylized or expressive, but do not need the full functionality of rich text, you should **choose** or **create** a font that **reflects** the sense of stylization you want by default and **fall back** on **standard Text** widgets.

Time to.... highlight a concept

Hierarchy and professions

Practice

- **General**

- Create a UMG UI representing a main menu, you do not need to make it interactable but just as an introduction to UMG placement
 - 3 buttons : Play, Options, Exit
 - Buttons with 20 spacing in vertical
 - Buttons of the exact same size
 - Make sure that if you add a button, it will not break your UI
- Understand how BindWidget works by creating a base c++ UserWidget and making sure that the BP inheriting from it has an image and a text
- Create a 3 slot inventory-like system purely in blueprint without any logic involve but which contains
 - Drag & Drop feature (Optional, but worth taking a look at how drag & drop is working on UMG)
 - Regardless of UI size, the 3 slots have always the same size

- **Follow-through project**

- You'll create your whole interface for your game, think the best way to do it, drive value change through events, etc...
- Display the number of ingot the player has already collected
- Display a skill icon for both skill you created last week with
 - An icon representing the skill
 - A text representing the cooldown of the skill
- Display a message when you collected an ingot notifying the player like "Ingot collected"
- Display a kind of alert on your HUD which will be displayed when a guard is tracking you