Author: Joel Gonzales | LowpolyCount
(Author's Note: this should be considered a "rough draft" that will be updated during the next few weeks. See the UIPT page for more information on the subject)

Technical Companion for How to build UI At Scale

We're going to talk about how we can build UI better when we are dealing with a large scale. It will be a high level view of how to approach the problem. More about architecture and philosophy than the actual code.

How have UI Responsibilities grown:

The UI needs in products over time have grown in two ways:

- Depth
- Breadth

The customer for Depth has always been game HUDs. The Minimaps we make for them are more complicated.. Health Bars, Weapon Displays, Players, it's all more complicated now.

Breadth growth comes from the need for… **More**. More Screens. Screens that handle Social, Payments, Account Information, the list goes on. This need for **More** has caused Breadth to grow faster than depth. What I'm proposing here will help with dept, but it shines when dealing with Breadth.

The Approach

Using Inheritance? As your project grows, so does the inheritance tree. Changes in the tree causes problems, you can read all about it.

Let's look at the adage of "composition over inheritance". If we can define functionality for our screens through components instead of classes, What does that look like?

Composition

This isn't the first time "composition over inheritance" has shown up in the Game Industry. Entity-Component-System was made to handle similar problems.

Let’s call this approach UICS - User Interface Component System.

Concepts:

Screen - the base unit of our UI. Any UI that is displayed, has to be attached to a screen. A Screen will also own Components

Widget - a piece of UI that is visual. Can contain visuals, can be interactive to the user.

Screen Components - Functionality that can be added to a Screen

Entry - A Piece of data that needs to be displayed and may be interacted with by the user.
As an example of an Entry, Here in midnight suns, we have a list of items you can buy. Each item is an entry. We can refer to the widgets that display that data as entry widgets.

Screen Components

I know people are used to Model/View. Throw that out. A Screen Component defines pure functionality for a screen.

System Setup

All screens, widgets, and Screen Components get access to the functions:

Type* GetScreenComponent<Type>()
TArray<Type*> GetAllScreenComponents<Type>()
Type* GetScreenComponentByName<Type>(FName)
Type* GetScreenComponentBySelector<Type>(FSelector)

This allows them access to Screen Components on the screen they are attached to.

Screens have an array of Screen Components. We can leverage an editor like Unreal to make it easier to add.
<Show Image>

Screen Component Types

What functionality should our Screen Components have? In UI, the 3 things we do the most are:

- Get Data
- Display Data
- Handle User Actions

Examples of these steps are:

We're in an inventory screen. We retrieve the player's inventory entries and display them on a grid. When the player clicks on an inventory entry, we will make a change to the system where that entry is discarded.

Another is character select. Here we retrieve all the hero entries that the player has unlocked and display them. When the user clicks on an entry, we open a screen.

What does this functionality look like when broken up into separate components?

Data Component

Responsible for retrieving a list of entries. This data is stored as a TArray<UObject*>

After Retrieval, it can optionally filter the entries and then optionally apply a transform

View Component

- Creating and Caching Widgets
- Sending Entry Data to those Widgets
- Listens to Widgets for user actions (Click, Hover, etc)

The nice side effect about this is that it breaks apart the creation and management of widgets from their container. The container the widgets are attached to, like an Horizontal or Vertical Box, are responsible for organizing how the list of widgets is displayed and organized.

Some containers in Unreal like X, do both of these things. This way allows you to make or use whatever container you want and not worry about Creation, Sending Data, etc.

Action Component

Handles an action the user wants to perform
Usually a change to the system. When the User clicks an icon in their inventory, it would handle dropping the item.

But can also handle things like Opening a screen in response to a button click.

These 3 components are our bread and butter of what we do in UI right. Get Data, Display that Data, and respond to a user action on that data.

Component Communication

Breaking functionality apart means that we have to think about how these screen components will work together.

UI Traditionally operates through events, so we can use Delegates to communicate between screen components.

Data Component has a OnDataRetrieved Delegate

Component Selectors

One way to make it easier to communicate, especially when dealing with multiple components of the same type is to make use of the Unreal Editor. Component Selectors allow us to pick specific components that exist on a screen

Example a case where we have two data and two view components. These let us choose which data component a view will listen to.

This allows us to hook up components without requiring any code.

Chart of Component Interactions

Picture of them working together using delegates

Examples

We're through the dry stuff. So let's go through examples of what we just learn, and put it into examples

Inventory Screen

Select Character Screen

Tooltips

Interaction Simplifications

Mission Select

Composing Widgets

Composable Widget

Widget Layers

Configurations

Stacking

Tooltips

Note for Artists

This may take some getting used to for artists. When they look through your content directory, they'll see assets appear in multiple places. To prevent them from having a heart attack, let

them know that each asset will only need to be changed in one place, but it may show up in different configurations

Takeaways

Should you switch?

UICS isn't the only system in games to propose using components: Entity-Component-System (ECS for short) predates us by a number of years. I'm going to drastically simplify its timeline, but the first game to use Entity-Component-System was Thief: The Dark Project in 1998. In 2015, an implementation of ECS appeared in Apple's GameplayKit with Unity demonstrating their own implementation in 2018. That is about 15 to 20 years between first use and a decent availability in game engines and libraries.

Things have changed a lot since then and I don't think 20 years is need to make a transition for UI, whether it's UICS or another technique

But there are things you should take into consideration when you want to make this change.

- Cost of Implementing UICS in your systems (pin)
    - Cost of training
        - I've found that senior level developers are comfortable with UICS in about 3 months.
- Is your project a good fit?
    - Do you have a lot of screens?
    - Do you have the wiggleroom to do something different than before?
- Existing codebase

I bring up these considerations because I believe technology exists for people and I want you to make good choices about what is best for your team and your users.