Tech Talk

Dynamic Text Localization

Introduction

About me!

Mollie, Software Engineer

- Survival game at Blizzard
- My route into engineering
- Why UI?

Formerly at Blizzard, on game engine for an unannounced survival game

- You may know as Odyssey
- Canceled, but most rewarding 5 years of my career

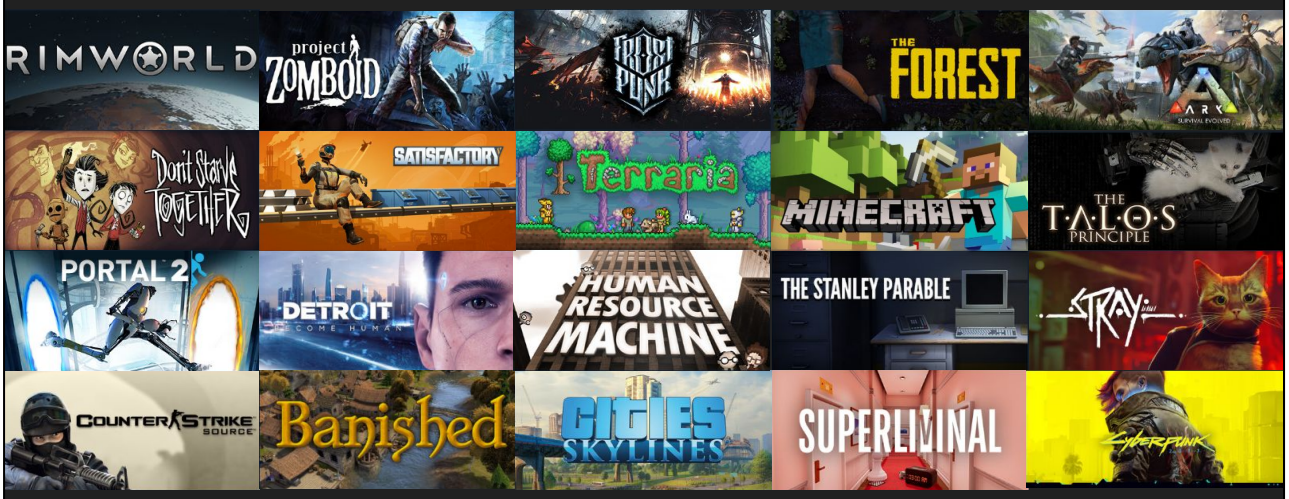
Unconventional route into engineering

- First hired as a tech writer to document the game engine architecture
- I'd always been into both programming and games, hobby projects. Gravitated toward things I was interested in, how I arrived at Blizzard
- Blizzard is good about letting people try new things. Learned a ton from the people around me

Why UI? - Art and a science to communicating

- Complex information in a simple, accessible way (something I enjoyed about tech writing, taking a complex topic and distilling it to be as accessible as possible)
    - Different form of the same challenge
- There's something very satisfying about taking functional system and refine it into something that feels smooth, responsive, and visually appealing.
    - Feels like solving a challenging puzzle
- Bridge between the player and game

Favorite Games



All time favorite games (top Steam games)

- Wide variety of genres - hook me and I spend a lot of time thinking and strategizing
- Scratch the same itch as writing code, designing and optimizing things
- Minecraft college days - public server with friends
- Learned how to mod Minecraft - programming!
- Youtube channel teaching kids how to enter commands on the server, walkthroughs and reviews of indie games

Some Background

For context!

Game Engine Dev

Part of a team developing a game engine

- Design for the future of Blizzard games
- Rapid iteration and prototyping

At Blizzard, I was part of a team developing a game engine

Idea was that this engine will be used to develop future Blizzard games

A key tenet is to enable rapid iteration and prototyping

- Big part of that is getting functionality into the hands of designers

Game Engine Dev

Part of a team developing a game engine, alongside a game

- Survival-crafting genre game to run on this engine
- Game requirements inform engine direction

At the same time, we were also developing a survival crafting genre game (that is Odyssey), which would run on the engine

Game requirements were a jumping off point for engine requirements

- We need the engine to handle this game's requirements, plus anticipate future game

Game Engine Dev

Part of a team developing a game engine, alongside a game, on UI

- Game engine UI systems
- Game UI

And specifically, on that game engine I was working on the UI system

That is the equivalent of UMG in Unreal

And then I was using that system to implement game UI or player-facing features such as inventory and character screens

Game Engine Dev

The engine needs a system responsible for localizing dynamic text for UI

- Our game needs robust dynamic localized text
- Stop reinventing the wheel

We needed a way to localize dynamic text that would be shown all over the UI of the game

- Survival crafting games tend to have complex UI
- Story driven aspects, needs to be capable of being dynamic (how the world will come alive)
    - Flexible, work correctly, manageable by designers
- 100,000 discrete pieces of text to localize

During my research, found that every game team at Blizzard had built a bespoke localization system

We want to break the cycle of creating a bespoke localization systems for each game

- Anticipate needs of future games

Game Engine Dev

The engine needs a system responsible for localizing dynamic text for UI

- My case studies are within the context of developing this system!

Key takeaway here:

My case studies are within the context of developing this localization system

Game Engine Dev

Working within Entity-Component-System architecture

- Asset-oriented/data-driven architecture design
- UI elements are entities which may have components
- Components hold data, systems manipulate the data

Lastly, a note on the game engine framework

We are working within the constraints of custom game engine with Entity-Component-System architecture

The gist of ECS architecture, in the context of UI:

- Each UI element is entities (buttons, icons, etc.)
- Entities may have components, components hold same data that informs behavior or functionality
- Systems manipulate the data

behavior of an entity can be changed at runtime by simply adding or removing components.

Game Engine Dev

Working within Entity-Component-System architecture

- Informs localization system design
- Decouple data and functionality

In keeping with the overall engine architecture, the localization system is also built around an asset-oriented and data-driven design

And we maintain a degree of separation between data and functionality

Case Studies

Overview

Here’s a super high level overview of my case studies

Case Studies Overview

1. Data
2. Functionality
3. Workflow

We’re covering a slice out of each of those ECS architecture concerns - data and functionality

And lastly, something related to the workflow within that framework

Case Studies Overview: Data

- Designing a scalable, flexible data model for localized dynamic text
- Make it accessible to designers

For the data aspect of this,

We'll walk through designing a flexible data model that we'll use for localized dynamic text,

And how we'll make it accessible to designers

Case Studies Overview: Functionality

- Designing a system to handle language rules and dynamic text (logic layer)
- Make it accessible to designers

As far as functionality,

We walk through what that system looks like, which will take in that data, and use it to handle localization concerns, like…

- Applying language rules based on language
- Resolving dynamic text
- Formatting things for different locales

So this is the logic layer, which also needs to be made accessible to designers

Case Studies Overview: Workflow

- Improving the localization workflow in the editor
- Designers want inline asset property editing

Last, I'll walk through a pretty big workflow problem that came up as a result of the data format, and how I figured out a way to solve that problem

Case Study 1

Let’s go.

A scalable, flexible model for localized text

Case Study 1

Problem

- Many, maaany lines of dynamic localized text
- Must be maintainable without engineer intervention

Many lines of localized dynamic text

- Survival crafting genre tend to have complex UI
- Story drives this game, needs dynamic rules

We want to designers to prototype fast and iterate fast, so we want to expose to designers whatever they need

Requirements: Designers

- Define a line of text to be localized and shown in game
- Define variables to be used to populate tokens in the string
- Use tokens in the string to indicate where to use variable data
- Use markup in the string to indicate where to apply specific language rules
- Provide context for external translators
- Define locale codes (enUS, esMX, arAE, frFR, frCA)
- Associate data for use translating for each locale

We want to give designers a lot of control..

- Define a line of text to be localized and shown in game
  - That's the most central piece here
- Define variables to be used to populate tokens in the string
  - A way for them to point to a piece of data that will be used by the system to resolve some other text
- Use tokens in the string to indicate where to use variable data
  - That's where to put those variables within the string
- Use markup in the string to indicate where to apply specific language rules
  - That looks similar to format string
- Provide context for external translators
  - So translators understand the meaning and usage of each string
- Define locale codes (enUS, esMX, arAE, frFR, frCA)
  - English (US), Spanish (Mexico), Arabic (UAE)
- Associate data for use translating for each locale
  - This could tell the system how to format, for example, date, time, or currency for a locale

Requirements: System

- Apply language rules based on the locale
- Format numbers, currency, dates, times based on the locale
- Expose ways for designers to get localized text in visual scripting
- Fallback to a default value when a translation is not available
- Efficiently retrieve localized text for a given locale at runtime

Some additional callouts for requirements that the system will need to fulfill

- Functionality that knows how to apply language rules
- As well as format numbers, currency, dates, time, etc.
- We have a visual script system similar to Unreal Blueprint, so we need to expose the system's functionality in script
- Handle missing data
    - Fallback to default
- Efficiently retrieve localized text for a given locale at runtime

Solution: Data representing..

- A single line of localized text
- A particular locale (language)
- Locale-specific translation data
- All translations for a particular locale

High level look at data, we know we'll need a way to represent

- A single line of localized text
  - Any associated data
- A particular locale (language)
  - enUS
- Locale-specific translation data
  - How to format dates, numbers, etc.
- All translations for a particular locale
  - We're going to send all these lines of text out for translation, and when we get that back we have to store it somewhere

Solution: Data representing..

- A single line of localized text
- A particular locale (language)
- Locale-specific translation data
- All translations for a particular locale

We'll start with the single line of localized text

LocalizedText

A new asset type

Recall this is all happening within an asset-driven design, so we'll create a new asset type, called LocalizedText, to hold all the data about a single line of text

Data representing a single line of localized text

Asset structure for `LocalizedText`

- `String Text`
- `List<Name/ScriptType> Variables`
- `String Context`
- `List<LocaleId/String> Translations`

Here's what that looks like.

On that asset, we have 4 main fields for data. We've got..

- Text, the string
- Variables, a list of key/values mapping a name to a special type called Script Type
- Context, another string
- Translations, a list of key/values mapping a localeId, to the translated string for this text

This is basically the "component" data in the ECS framework

We'll go into a bit of detail on each

Asset: LocalizedText

`Text` defines the line of text

- Placeholders for variables
- Markup to apply language rules

Text, defines the line of text, in the origin language, in our case that’s English, and we are the locale enUS

Let’s look at what we expect that might look like.

Asset: LocalizedText

`Text` defines the line of text

- Placeholders for variables
- Markup to apply language rules

"Hello `{ playerName }`, welcome to the game!"

We plan to allow using tokens in the string, so that's a placeholder for a variable, like PlayerName you see here

Asset: LocalizedText

`Text` defines the line of text

- Placeholders for variables
- Markup to apply language rules

"You have `{ amount } %pl(point|points)`."

We also plan to support markup showing where to apply language rules, like you see here indicating where we'll show "point" if the value of amount is, 1, otherwise, it'll be points.

Asset: LocalizedText

`Text` defines the line of text

- Placeholders for variables
- Markup to apply language rules

"Masz `{ amount } %dcl(punkt|punkty|punktów).`"

And just kind of a peak at what this look like translated.

Some languages have more complex rules than english, for example this same line in Polish looks like this, and in Polish we'd apply a declension rule.

This is just to emphasise there are other language rules, this string wouldn't go in the text field since that is there we're storing the original version, which is English for us.

The translator would be the one who enters this string, and we store this in Translations field, in an entry keyed by the locale for Polish in Poland (PL-PL)

Asset: LocalizedText

`Variables` is a list of things we'll use to resolve dynamic text

- Designers only need to worry about `ScriptType`
- Basically a wrapper for a bunch of different possible types

Next field on the LocalizedText asset, Variables

This is a list of things we'll use to resolve dynamic text, these are keyed by the token name, and at this stage we don't usually know their value

Asset: LocalizedText

`Variables` is a list of things we'll use to resolve dynamic text

- Designers only need to worry about `ScriptType`
- Basically a wrapper for a bunch of different possible types

"Hello `{ playerName }`, welcome to the game!"

The values are all a special type ScriptType, which is basically a wrapper for a bunch of different possible types (e.g., assetId, int).

Designers don't have to worry about types too much, they just pass something in that can be cast to a script type

So here we know we have a variable called playerName

The script might pass in some asset type that knows about player data…

Asset: LocalizedText

`Variables` is a list of things we'll use to resolve dynamic text

- Designers only need to worry about `ScriptType`
- Basically a wrapper for a bunch of different possible types

"Hello Mollie, welcome to the game!"

and we could resolve that to the players name.

Variables could also point to another localized text asset, which might itself have a variable that points to another localized text asset.

We'll get into handling that bit later

Asset: LocalizedText

`Context` is for use by external translation team

- Helps translators understand the meaning and usage of each string

For this presentation I shortened Context to a single field, but in practice this would likely need to be more extensive, because it contains a lot of important data that translators will use to make sure we're preserving the original meaning and intent behind each line of text.

You might include context about the tone being formal or informal, details about who its addressing, gendered language considerations, cultural considerations

At the end of the day, this is design-time only data, it's not included in any of the binary resources the game uses at runtime.

Asset: LocalizedText

`Translations` maps a locale to corresponding translations of this string

- Not editable
- Set by importing translated strings from loctool

Asset: LocalizedText

`Translations` maps a locale to corresponding translations of this string

- Export tool packages up the data for external translation teams
- Import tool writes the results back to the assets

Translations is a list of key/values mapping a localeId, to the translated string for this text

These are not writable by designers working with in the asset.

- Export tool packages up the data from the assets to send to external translation teams,
    a. flags assets as out for translation
- Import tool writes to the assets Translations list and unflags the asset

Asset: LocalizedText

Any other asset type can now include a field for localized text

- A weapon asset has a Name field, which is type `LocalizedText`

So now we have a localized Text asset

The end result is that any other asset type can now have a field that takes a type of localized text

- We no longer allow game UI text to be set using a raw string, all game text must go through the localization system
- This means we can audit and account for all of it

As an example, a weapon asset could have a field called Name, which takes a localized text asset. And a “GetName()” functions, knows how to extract that data

Solution: Data representing..

- A single line of localized text
- A particular locale (language)
- Locale-specific translation data
- All translations for a particular locale

The LocalizedText asset was the big one, but there are a couple other pieces of data we'll need to represent

Next up, we need a way to identify a particular locale, so we know what language to use

LocaleId

A new asset type

That’s going to be a new asset type called localeId

Asset: LocaleId

- String identifying a locale (esSP, esMX, etc.)
  - In C++, packed int for faster compare
- Reference to a LocaleData asset (up next)

This one's pretty simple

We just have a string field, which expects a standardized locale code

- Consists of language and region, spanish-Spain, spanish-Mexico
- I'll note that in C++, we don't compare this string, instead we'll use a packed integer for much faster comparisons

And a reference to another asset type we'll define in a minute, which holds some locale-specific translation details

Solution: Data representing..

- A single line of localized text
- A particular locale (language)
- Locale-specific translation data
- All translations for a particular locale

And so about that locale-specific translation data..

LocaleData

A new asset type

It’s a new asset we’ll call locale data

Asset: LocaleData

Data about how to translate for a particular locale

- Translations for days and months
- How to format common things (numbers, currency, date, etc.)
  - Thousands separator: `1,000` (English) → `1.000` (Spanish)
  - Currency symbol (`€`, `$`, etc.) and placement: $10` (English) → `10 $` (Spanish)

LocaleData has a bunch of fields, the gist of it is that it's all data that describes how to translate and format for one particular locale.

- Has translations for some common time and date related things, like days of the week and months
- It defines how to format dates and times, 24 clock
- Has the different symbols and placement for things like currency, temperature
- Measurement units, metric vs. imperial
- Some info that might be important when we're rendering, like text direction

Because LocaleId and LocaleData are also both assets, designers can set up a new locale and define a lot about how we'll handle it without the need to wait for an engineer to do anything behind the scenes. So designers can get in there and start prototyping right away

Solution: Data representing..

- A single line of localized text
- A particular locale (language)
- Locale-specific translation data
- All translations for a particular locale

And the last important piece of data, all translations for a particular locale

So that means like, the complete collection of all localized text as translated to spanish

Resource

Compiled from LocalizedText

That will be a resource, which we will compile from all localized text assets

- Recall each translation is stored on the asset
    a. On a field called Translations

Data representing all translations for a particular locale

Some context about resources

- Each asset gets compiled into a binary resource
- Resources are identified by a resource key
  - Hash of assetId + other metadata

Some context about resources

Each asset gets compiled into a binary resource

Resources are identified by a resource key, which is a hash of assetId

Data representing all translations for a particular locale

Some context about resources

- Many resources of the same type can be compiled into a resource catalog
- Load that catalog once, avoid waiting for async resource loading

Many resources of the same type can be compiled into a resource catalog
We like this because it’s effectively one resource with all the resources (of that type) in it

Data representing all translations for a particular locale

- Recall each translation is stored on the asset
- We compile one resource catalog, containing all strings, for each locale
- Change locale at runtime? Swap the resource catalog

For each locale, we compile a resource catalog containing one resource for each translated string.

As the resource's key is based on the original asset, it stays the same, regardless of which locale's resource catalog we are using.

When we want to change the locale for the game, we just swap out the resource catalog, and some callback functions will cause all the text elements to be refreshed with the new string from the new resource catalog.

Case Study 1

Fin.

That’s the end of case study 1. So that wraps up the data model.

Case Study 2

Let’s go.

Case study 2 is all about the logic layer.

A system to handle language rules and dynamic text

Case Study 2

That is essentially, we need to design a system to handle language rules and dynamic text. So that's the "system" in the ECS framework. It's going to do something with the component data.

Problem

- Complex language rules
- Formatting rules
- Dynamic text
- Accessible to designers

The highlights of this problem are..

- We'll need to handle complex language rules to make sure our strings are grammatically correct in whatever language we display them in
- There is locale-specific formatting for things like date, time, currency, as defined in LocaleData
- There's also dynamic text that we need to resolve and replace tokens in the string with the resolved values

And, we need designers to be able to tap into it through the visual scripting system

Solution

C++ systems to handle parsing strings, applying language rules, formatting numbers/currency/date/time, and exposing those to designers in script.

- Initialize a default locale to fall back on
- Initialize a user-selected locale
- Load the resource catalog(s)

The solution here was building out a C++ system that handles all that efficiently in C++ code.

Some other noteworthy responsibilities of that system..

- We need a default locale to fall back on if data is missing
- Initialize locale data for locales we might display, so we can load up the appropriate resources

There are several key functions for this system..

Key Functions

`FormatNumber()` `FormatDate()` `FormatTime()` `FormatCurrency()`

Parse format string and return result

- `"h:mm a"` formats to "12:08 PM"
- `"EEE, MMM d, yyyy"` formats to "Mon, Mar 3, 2025"

We have functions to format numbers, date, time, and currency

These work by parsing a format string and returning the result

For example,

The format **`EEE, MMM d, yyyy`** translates to:

- **EEE** – Abbreviated day of the week (e.g., Mon, Tue)
- **MMM** – Abbreviated month name (e.g., Mar, Jan)
- **d** – Day of the month without leading zeros (e.g., 3, 15)
- **yyyy** – Four-digit year (e.g., 2025)

Key Functions

`FormatNumber()` `FormatDate()` `FormatTime()` `FormatCurrency()`

Expose in visual scripting system

- Designers can display things like cost, date, time
- The system knows which locale to format for

Exposed access to these functions in the visual scripting system

So designers can use them to display things like cost, date, or time in the game UI

The localization system knows the locale we need to format for, so it chooses the appropriate format string to pass into the function

Key Functions

`ResolveDynamicText()`

Parse string looking for tokens to replace with provided data for variables

- Variables can be any of several types at runtime (ScriptType)
  - Token { weapon }, script passes *something* in at runtime
  - We determine it’s a weapon asset

Another key function, ResolveDynamicText()

Responsible for parsing the string looking for tokens to replace with data from variables

Variables are a special type that is basically a wrapper for many different possible types..

For example

- we might have a token like { weapon } but we don't know what type that is until a script passes something in at runtime
  a. When it's time to resolve that string, we can find out what type "weapon" is and use it to resolve the string

The system might determine it’s a weapon asset and know how to handle that, by say getting its Name field

- Alternately…

Key Functions

`ResolveDynamicText()`

Parse string looking for tokens to replace with provided data for variables

- Localized text can nest other localized text
  - Resolve only once all variables are ready
  - Minimize allocations for performance

- Alternately, maybe we determine the variable refers to another localized text asset, which contains the name of the item, as well as another variable for its level.

So, Localized text can have other localized text nested within it recursively.

Script system has mechanisms to ensure we only resolve the text string once we know we have the data available to fully resolve the entire string.
So we only allocate strings when we really need to. Not intermediate ones.

Key Functions

`ApplyLanguageRules()`

Parse string looking for markup identifying where to apply language rules

- Rules ensure grammatical correctness of the string for given locale
- Modify string based on pluralization, declension, gender, capitalization, etc.
  - Passes string around as c-style, pointer to first char (not copying)

Another key function, `ApplyLanguageRules()`, parses the string looking for markup where we need to apply language rules

Modify string based on rules for..

- Pluralization
- declension,
- grammatical gender rules,
- capitalization and other case transforms
- Lots of other rules for ensuring we the grammatical correctness of the text for the given locale

Noteworthy in the C++, we're aiming for performance, so the code passes string around as c-style, pointer to first char (not copying)

Key Functions

`ApplyLanguageRules()`

Parse string looking for markup identifying where to apply language rules

- Rules ensure grammatical correctness of the string for given locale
- Modify string based on pluralization, declension, gender, capitalization, etc.
  - Passes string around as c-style, pointer to first char (not copying)

"You have `{ amount } %pl(point|points).`"
"Masz `{ amount } %dcl(punkt|punkty|punktów).`"

Here's an example.

"You have { amount } %pl(point|points)."

Amount is a variable,

When we see that "%pl" we know we are going to apply a language rule to determine the correct form of the word "point" to use. There's a function like "Apply Pluralization Rule" that will take the value of amount, see we are in enUS locale, and if the value is 1, use "point" (you have 1 point) and otherwise it'll be "points." (You have 0 points, you have 10 points)

In Polish, the language rules for pluralization words are a bit more complex and there are 3 possible options.
That same text needs to use a declension rule, so the Apply Declension Rule function will see we are in the plPL (polish) locale, it will do some other logic to determine which declined version of this word to use. The amounts here a 1, 3, 7

Key Functions

`GetLocalizedText()`

Returns fully resolved string based on localeId + variables

- Calls `ResolveDynamicText()` until variables are fully resolved
- Calls `ApplyLanguageRules()` for given locale

The last key function, GetLocalizedText, puts everything together

It takes the values for variables to be used when resolving the dynamic text, which of course might refer to other dynamic text

It calls ResolveDynamicText until we've fully resolved all the variables, which might have a bit of recursion if we are nesting other variables within them

Then it applies the language rules before returning the result

Key Functions

`GetLocalizedText()`

Expose as a node in visual script

- Designers configure script to provide a localized text asset + variables
- Flexibility, within the safety of the localization system

The script node is calling GetLocalizedText and providing all the variables to be used creating that dynamic text

This gives designers lots of flexibility and power, while also funneling everything through the localization system, which also validates it and flags any problems like missing translations or bad references

Putting it all together

A UI element wants to display some localized dynamic text when you pick up an item worth some amount of points.

And putting it all together, we'll step through what happens when we get a piece of localized text to display in the UI.

A UI element wants to display some localized dynamic text when you pick up an item worth some amount of points.

Putting it all together

In visual script, a Get Localized Text node has the following

- Asset LocalizedText
  - `"Found a { itemName } worth { amount } %pl(point|points)."`
- Variables itemName, amount
  - Values passed in

In the visual script, a designer can add a Get Localized Text node, and give it a reference to a localized text asset, as well as some data to be used to fill out the variables, for itemName and amount in the string, which is “Found a itemName worth Amount points/points.

Putting it all together

The node calls the C++ function `GetLocalizedText()`

- System knows the current locale
- The appropriate resource catalog has already been loaded

Found a { itemName } worth { amount } %pl(point|points).

Once that node gets called, we're in the c++ function GetLocalizedText. The system knows the locale, and the appropriate corresponding resource catalog has already been loaded.

Putting it all together

The node calls the C++ function `GetLocalizedText()`

- `ResolveDynamicText()`
  - itemName is type LocalizedText
    - Resolves to "star" by looking up the value in the resource catalog
  - amount is type integer, resolves to "4"

Found a star worth 4 %pl(point|points).

GetLocalizedText calls ResolveDynamic text, and because itemName is another localized text, we can look that up in the resource catalog

Putting it all together

The node calls the C++ function `GetLocalizedText()`

- `ApplyLanguageRules()`
  - Finds markup indicating we should apply pluralization rule
  - Does logic on the amount variable

Found a star worth 4 points.

Then apply language rules does some logic using the value of Amount and determines which usage of the word “point” to go with.

Putting it all together

In visual script, the Get Localized Text node gets the result

- The resolved string is displayed on the UI element.

Found a star worth 4 points.

And that resolved string is returned, and the node can assigns its value to an entity (a UI element)

And that’s it.

Case Study 2

Fin.

Case Study 3

Let’s go.

Final case study is about overcoming a workflow issue that came up.

Improving the localization workflow in the editor

Case Study 3

Problem

Designers report

- The workflow sucks for embedding dynamic text within other dynamic text

I got the localization system into the hands of game designers as quickly as possible so that I could start getting feedback on workflows.

Designers reported this issue over and over: the workflow sucks for embedding dynamic text within other dynamic text.

Problem

- Asset editor is an Unreal-style property grid
  - Handles one asset at a time
  - References to other assets open in a new document
- Quickly end up with a lot of open documents
  - Hard to conceptualize end result
  - Easy to introduce circular references

Asset editor is an Unreal-style property grid, where you open a single asset and you see all the fields on that one asset at a time.

When a field holds a reference to another asset, you can click on it to open that asset in a separate document, where you can then edit fields in that asset

Key: Property grid is set up to handle one asset at a time, each in a separate document

Quickly end up with a lot of open documents, it's hard to conceptualize the end result of nested strings, easy to introduce circular references

Problem

- The property grid is owned by the tools team
  - They have an enormous backlog
- We need this to better now

The property grid is owned by the tools team, so I went and asked them about it. They had a multi-year backlog, so it is NOT a near-term option.

How do we get inline editing of assets in the property grid without waiting for the tools team to do the work?

Requirements

Figure out the best workflow within the constraints of the system

- Property grid shows all the data from any referenced asset
  - Nested where it was referenced in the parent asset
    - Recursively
- Fields must be editable
  - Save data on the correct asset(s)
- Kick off recompiles
  - Updated resources to reflect the changes

Approach: Figure out what the best workflow looks like within the constraints of the systems we're working in

Worked with designers who would use this feature; we want:

- The property grid to show all the data from any referenced asset, nested where it was referenced in the parent asset, recursively
- Fields must be editable and data must be saved on the correct asset(s)
- Kick off recompiles for any referencing assets that need to have their resource updated to reflect the changes

Solution

Frankenstein approach.

- Track down places in the code that do these things
- Stitch them together such that the property grid handles inline editing of nested assets

The solution was a frankenstein approach.

I knew that there did exist bits of code, scattered all over the place, that did all the little pieces I wanted

I knew this from using the editor and seeing functionality that we needed

It was just a matter of finding it

Solution

All the functionality exists. It’s just a matter of finding and adapting it.

- Open assets, display fields and make them editable, find dependencies
- Modify multiple assets without actually opening them in a document
- Fix up the presentation in the property grid (recursively nest)

There was existing code to open an asset in a document, display the asset's fields, make them editable, save changes to the asset, and then kick off a re-compile of the corresponding resource as well as any others for which it was a dependency. Those are all pieces I can pull out for my purposes.

I knew we had some asset migration functionality that would modify a bunch of assets at once to keep them valid if the asset data structure changed, so I could adapt that to edit many assets at once without opening them in documents

The presentation layer was ImGui, so it was pretty easy to figure out how to indent and nest ImGui elements once I had the data to populate them with

Now we just pull the fields out of the referenced asset and display them in the same property grid.

And recursively, nest any assets that asset references.. so you're now seeing all of this in one view. Don’t have to open any other documents.

Solution

Write up a proposal for the tools team.

- Get feedback
- Do the work
- Ask questions

Pretty significant code change. Largely made up of re-using existing bits of code.

So I tracked those items down, looked at their implementation, and wrote up a summary of how I thought I could Frankenstein all these pieces together myself.

The tools team lead was able to quickly review it and provide some feedback, much faster than if he were to do/assign the work himself.

While doing the work, I was able to ask specific questions to the tools team without taking up much time.

Pretty significant code change, but largely made up of re-using existing bits of code.

Result

I delivered this feature the designers really needed ASAP,

- I didn't go do a solution specific to text, so it works on all asset types
- Now all asset types support inline property grid editing referenced assets

I delivered this feature the designers really needed ASAP.

Because I didn't go do a bespoke solution specific to text, it worked on all asset types.

Now all asset types support inline property grid editing of fields on referenced assets, so when an asset references another asset, we can see and edit that in one view.

So doing it the right way has instant payoff

Case Study 3

Fin.

And that wraps up case study 3.

Thank you!

Questions?