Karelia University of Applied Sciences
Bachelor of Business Administration
Game programming

Applying Flux Architecture to an Unreal Engine Game

Riku Jako

Thesis, December 2022

www.karelia.fi

Karelia
UNIVERSITY OF APPLIED SCIENCES

THESIS
December 2022
Business Information Technology

Tikkarinne 9
80200 JOENSUU
FINLAND
+ 358 13 260 600 (switchboard)

Author (s)
Riku Jako

Title
Applying Flux Architecture to an Unreal Engine Game

Abstract

The goal of the thesis was to find out if Flux programming architecture can be used in a game created using the visual scripting language of Unreal Engine and if it helps people with little experience of using Unreal Engine to develop scalable games.

In this study a game was first implemented using a language where the Flux architecture is commonly used. Secondly, the game was divided into different logical segments, and features with corresponding functionalities were searched from Unreal Engine. Lastly, a game with identical functionality and comparable structure was implemented in Unreal Engine.

The Flux architecture is fairly well suited for smaller-scale games implemented in Unreal Engine. Due to the limited requirements of the game chosen for the implementation the suitability of the Flux architecture for Unreal Engine games could not be fully explored. Certain logical segments of the game were easier to transfer to Unreal Engine, while others require further studies.

Language
English

Pages 51

Keywords
Unreal Engine, Flux architecture, game development

Karelia AMMATTIKORKEAKOULU

OPINNÄYTETYÖ
Joulukuu 2022
Tietojenkäsittelyn koulutusohjelma

Tikkarinne 9
80200 JOENSUU
+358 13 260 600 (vaihde)

Tekijä(t)
Riku Jako

Nimeke
Flux-arkkitehtuurin hyödyntäminen Unreal Enginellä toteutetussa pelissä

Tiivistelmä

Opinnäytetyön tavoitteena oli selvittää, voiko Flux-ohjelmistoarkkitehtuuria hyödyntää Unreal Enginen visuaalisella ohjelmoinnilla toteutetussa pelissä sekä auttaako se ihmisiä, joilla on vähäinen kokemus Unreal Enginestä kehittämään skaalautuvia pelejä.

Ensiksi tutkimuksessa toteutettiin peli käyttäen ohjelmointikieltä, jossa Flux-arkkitehtuuria perinteisesti on käytetty. Seuraavaksi toteutettu peli pilkottiin loogisiin lohkoihin, joille etsittiin vastineita Unreal Enginestä. Lopuksi Unreal Enginessä toteutettiin toiminnallisuudeltaan identtinen ja rakenteeltaan vastaava peli.

Flux-arkkitehtuuria voi hyödyntää ainakin pienemmän mittakaavan Unreal Engine peleissä. Toteutettavaksi valitun pelin alkeellisten vaatimusten johdosta Flux-arkkitehtuurin soveltuvuus Unreal Engine -peleihin jäi paikoin kartoittamatta. Pelin tietyt loogiset lohkot olivat helposti siirrettävissä Unreal Engineen, kun taas jotkin vaativat lisätutkimuksia.

Kieli
Englanti

Sivuja 51

Asiasanat
Unreal Engine, Flux-arkkitehtuuri, pelikehitys

Contents

1 Introduction

The studies at the Karelia University of Applied Sciences taught game programming using Unreal Engine. The games created as course work during the studies were not particularly large or complex, but nevertheless the code and relations tended to steer towards a chaos. This was understandable, as the focus in the course was to learn specific principles and functionality, not clarity and sustainability.

Later, the studies included on-the-job training. During the practical training period the author was employed into a company where React (Meta Platforms 2022a) and Redux (Abramov 2022a) were used to build web applications. The code was clean and well organised according to the principles of React and Redux, making it easy to comprehend multiple large applications developed by the company.

The clarity and systematic approach in these large applications impressed the author. The two courses in school were not centered around managing and structuring scalable code in Unreal Engine. From this, several questions came up. Could code structuring from React and Redux be applied to games built using Unreal Engine? Will the conventions Unreal Engine has been designed for have to be circumvented? Can some of the tools of Unreal Engine be utilized for purposes for which they were not designed? This thesis is an experiment to see if the tools and principles the author has learned during their time in the company can be used to create games. The desired result is to gain an understanding of how the structures and data flow from React and Redux can be implemented in Unreal Engine, and to see if this makes developing Unreal Engine games easier for people with strong understanding of React and Redux.

2 Background

2.1 Flux architecture

Flux is a software architecture by Meta (formerly known as Facebook). It is a pattern which dictates where data is stored and how it flows in the program in a unidirectional manner. The Flux architecture primarily consists of four parts. The first part is the store, where the data is stored. The second part is the view, which is most often a user interface (UI). The third part are actions, which are generated by the view, when the user interacts with it. The fourth part is the dispatcher, which transmits these actions to the store and, optionally, other modules that need to react to the actions. Depending on the action, the store updates its state, and the views receive this updated state. Through all these parts the unidirectional data flow is complete (figure 1).

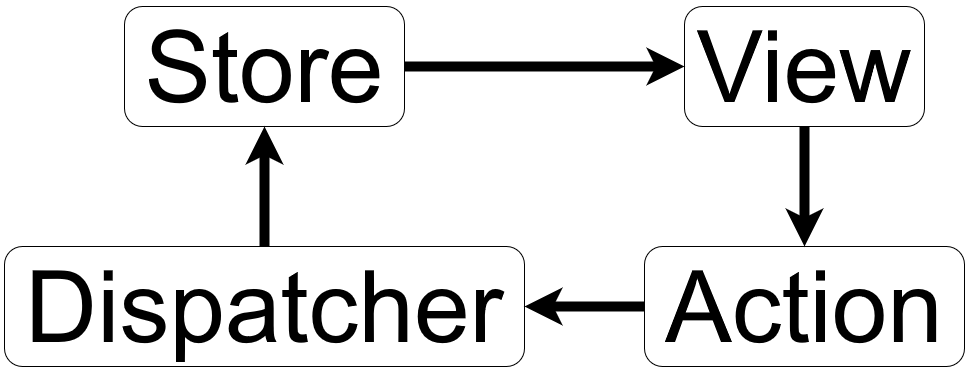

Figure 1. Unidirectional data flow in Flux architecture.

2.2 Flux in practice

Since Flux is merely a pattern and nothing tangible, programming tools to implement it in every project would require substantial boilerplate code. To streamline the process, multiple libraries implementing the Flux model have been written. The one used in the company the author worked at is called Redux.

Redux has a few differences from the Flux model. Mainly, in Flux the stores modify themselves, while in Redux store manipulation is handled by reducers. Secondly, Flux can have multiple mutable stores, while Redux only has an immutable one, which can be divided into sections. (Asiri 2018)

Sometimes actions can trigger complex logic, which can conditionally modify different store sections asynchronously. Writing this logic into the store reducers is not feasible, as reducers are supposed to be pure functions which update the store state based on the actions and values it receives. To make writing this complex logic and managing the side effects easier, the company uses a library called Redux-Saga (Redux-Saga 2022a). Sagas are asynchronous functions which can listen to and dispatch actions. This makes it possible to manage potential complex logic flow between the user actions and store state, enable communication between different contexts such as browser local storage or an application programming interfaces (API), and propagate changes throughout the application.

2.3 Unreal Engine

Unreal Engine is one of the leading game engines (Doucet & Pecorella 2021) developed by the Epic Games incorporated. It is suitable for developers small and large (Perforce 2020a), as it offers extensive documentation and tutorials (Epic games 2022a), an extensive development and tool set which the customer can modify to their needs (Epic Games 2022b), and easy to understand royalty-based pricing (Epic Games 2022b) compared to Unity (Unity Technologies 2022), a notable competitor of Unreal Engine (Doucet & Pecorella 2021). Epic Games takes five percent royalties after the lifetime gross revenue of a product exceeds one million dollars, with the first million being royalty free (Epic Games 2022b). The free plan offered by Unity requires the revenue or funding during the last twelve moths being less than 100000 dollars, with product plans with more advanced features ranging from 399 dollars to 2040 dollars per user per year (Unity Technologies 2022).

By default, the structures offered by Unreal Engine, such as Blueprints and Actors, steer development towards mainly object-oriented programming (Romero & Sewell 2019), where entities hold and manage their own state, and define their interactions with each other. By contrast, structures like Game State (Epic Games 2022c), are comparable to the stores in Flux architecture. Unlike in Flux architecture, Unreal Engine enables storing a lot of data and functionality in object-oriented structures (image 1), so converting a whole game implemented using traditional Unreal Engine data storage paradigms to follow a Flux like structure can turn out to be a challenge.

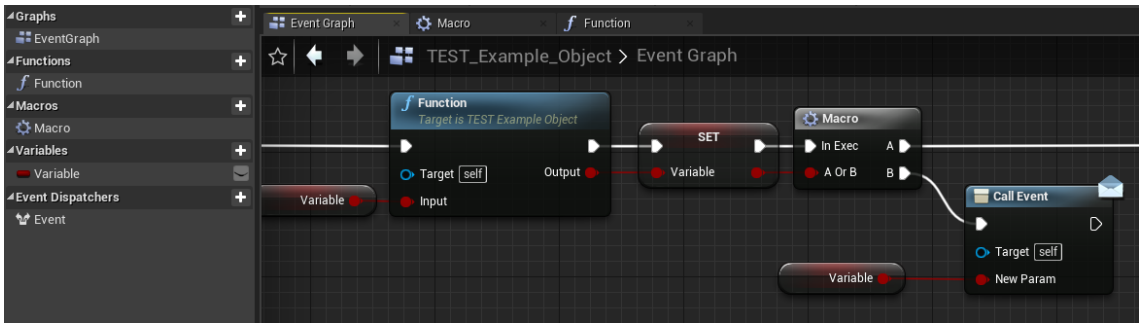

Image 1. An Unreal Engine game object containing data, functionality, and communications.

2.4 Prior Flux implementations in game development

Scouring the internet resulted in a negligible number of search results relating to the use of Flux or unidirectional data flow in Unreal Engine, or games in general.

A couple of the earliest records of these concepts converging are a GitHub repository for a game of tic-tac-toe (Hack Hat 2015) implemented using flux-library (Facebook 2022) and React, and a blog post by Lev Perlman explaining about implementing tic-tac-toe using the Flux architecture (Perlman 2017). The third mention the author found about these is in an article by Thiago Negri, in which they report about their experiences creating a simple mobile game in Unity (Negri 2015). A fourth mention is when Alvin Ourrad demonstrates unidirectional data flow usage in WebGL utilizing REGL-library during DevConf

2017 (Ourrad 2017). A fifth union of these is in a paper by B. Penz and J. Mäkiö, where they consider using Reflux, a Flux inspired JavaScript library, to be used in teaching software development to children (Penz & Mäkiö 2019). A sixth close search result is a paper by Sampsa Kaskela, where they implement a real-time user interface and consider how their findings could be used in games (Kaskela 2019).

3 Methods

3.1 Steps of the architectural migration

This thesis is a practical study, mostly comprised of regular software development. Potential approaches are explored, some are implemented, and the suitability of the chosen approaches are evaluated. The platform specific tools will be mentioned in later chapters.

The first step is to make a simple game for the browser. The second step is to split the browser game into parts, such as store or view, and see if Unreal Engine already has components with matching functionality. The third step is to implement the same game in Unreal Engine while trying to utilize the Flux architecture. It will likely take some trial and error to see which Unreal Engine components are the best suited to implement Flux.

The game itself should be as minimal as possible to speed up development and reduce possible noise from findings. Increasing the amount of similar logic should not greatly increase the accuracy of the findings, as this similar logic would likely use comparable or the same structures used in the implementation of the more minimalistic game. A more complex game would test the scalability of the Flux architecture in Unreal Engine, but the primary goal of this study is to see how well the Flux architecture can be implemented in Unreal Engine.

3.2 Minimal viable project

To reduce the time and resources required for development the game should be state based, have a limited rule set, preferably require little logical processing from the game itself, and have plain graphics. If the game has subjects updating in real time, this could cause the game state to update thousands of times a second, possibly leading to cascading updates in other places. The frequent updates could cause, for example, logical, physical, data transmission and memory limitations. Separate research should be done into each of these if one wants to apply Flux based architecture to games with real-time elements. A limited rule set and little logical processing from the game speed up programming the game both for the browser and Unreal Engine, as well as speed up iteration between implementations of Flux architecture on different platforms. Lastly, graphical fine tuning is not included in the scope of the thesis, as it does not contribute to the logic of the game.

One well known game to fit all these requirements is tic-tac-toe, also known as Noughts & Crosses, or OXO (Marcus 2021). In it players take turns claiming squares from a game board. The rules can be explained in five sentences (Cyber Octopus web page 2000), and the game logic mostly manages the moves the players can make as well as the win-state. The graphics consist of a three-by-three grid with X and O characters marking the moves the players have taken (Vectoreezy 2022).

Tic-tac-toe meets the requirements set for the game to be chosen, and the rules of it go as follows:

- The game board is a three-by-three grid.
- The first player is X, and the second player is O.
- The players alternate placing their respective characters in one of the empty squares on the game board.
- If a player manages to place three of their characters in a row, either horizontally, vertically, or diagonally, they win.
- If all 9 spaces of the board have been filled, but neither player has managed to place 3 of their characters in a row, the game ends in a draw.

3.3 Specifications for the game

The game starts immediately, and the X player can make their move. A restart button is always displayed, and either player can press it to reset the game to the beginning state. The players whose turn it is currently is indicated with a text displaying “X’s turn” or “O’s turn”. When a player wins, “the X wins” or “the O wins” message is displayed in the text. In the case of a draw, “Draw” is displayed in the text. The game is played locally on a single machine, requiring no networking to communicate between game clients. It also requires only one input device, the mouse. The game board squares have three possible states, empty, X or O. The text instantaneously switches between the messages it displays. The graphics require no effort to create, consisting of black and white colours, with no animations included.

The user interface elements do not contain any logic. For example, the game board squares do not determine if a square can be pressed, or who can or does press them. Instead, the UI elements follow the game state and pass the player made events to the game logic.

The game state is stored in two sections, also known as stores in the Flux architecture. The first store stores the general game state, such as whose turn

is it currently, and if the game has ended. The second store stores the state of the game board. The amount of data each of these stores contains is so little that it would be feasible to combine them into one. More complex games and applications can contain much more data, requiring segmentation to improve maintainability and expandability. The aim is to simulate this need and possible problems that come with the architecture.

One possible problem with this proposed data segmentation is handling the end state. The second store contains information about a three-in-a-row situation, while the first is responsible for propagating the end state. The UI components could deduce the end state of each turn from the game board state, which would be a sensible implementation in a game of tic-tac-toe, but not necessarily in a heavier application. Because the end state is stored in the store it needs to be calculated only once regardless of the number of entities dependent on the information.

A more sensible segmentation of data could be to store the turn, end state, and game board data in one store, and store player-defined player names and match history in another store. However, this would expand the specifications to contain more logic and UI elements, so the idea is foregone to speed up development.

4 Implementations

4.1 Implementation in a web browser

4.1.1 Prerequisites

React and Redux are JavaScript libraries designed to run in browser environments; thus the browser implementation will be a JavaScript project. JavaScript projects most often utilize libraries, also known as packages, written by third parties, which can contain logic or resources. Package managers are

used for fetching and managing these dependencies. NPM is one commonly used package manager (npm inc 2022) and comes pre-installed with Node.js, which is a runtime for JavaScript (OpenJS foundation 2022). Both at the company the author was employed and in this project Yarn (Yarn 2022) was used instead of NPM for package management, which is an alternative to NPM. The basic functionality between the two is mostly the same, with the largest difference being Yarn having a few speed improvements, such as parallel package installation (Krishna 2022).

4.1.2 Boilerplate

Setting up a project with TypeScript, React and Redux requires configuration. To initialize the project folder, type the command *yarn* to the command line in the project folder, which generates *.yarn-integrity* file in *node_modules* folder and *yarn.lock* file. Secondly, run the *yarn create react-app my-app --template redux-typescript* command, which creates a folder called *my-app*, and a pre-configured and functioning example application which uses TypeScript, React and Redux. Finally, to de-abstract the template project, run the *yarn eject* command, which exposes the installed dependencies, build tools and configurations. Using these provided commands and templates removes the need to write 40 files spread across ten folders, not counting *node_modules*, according to the Windows file explorer (image 2).

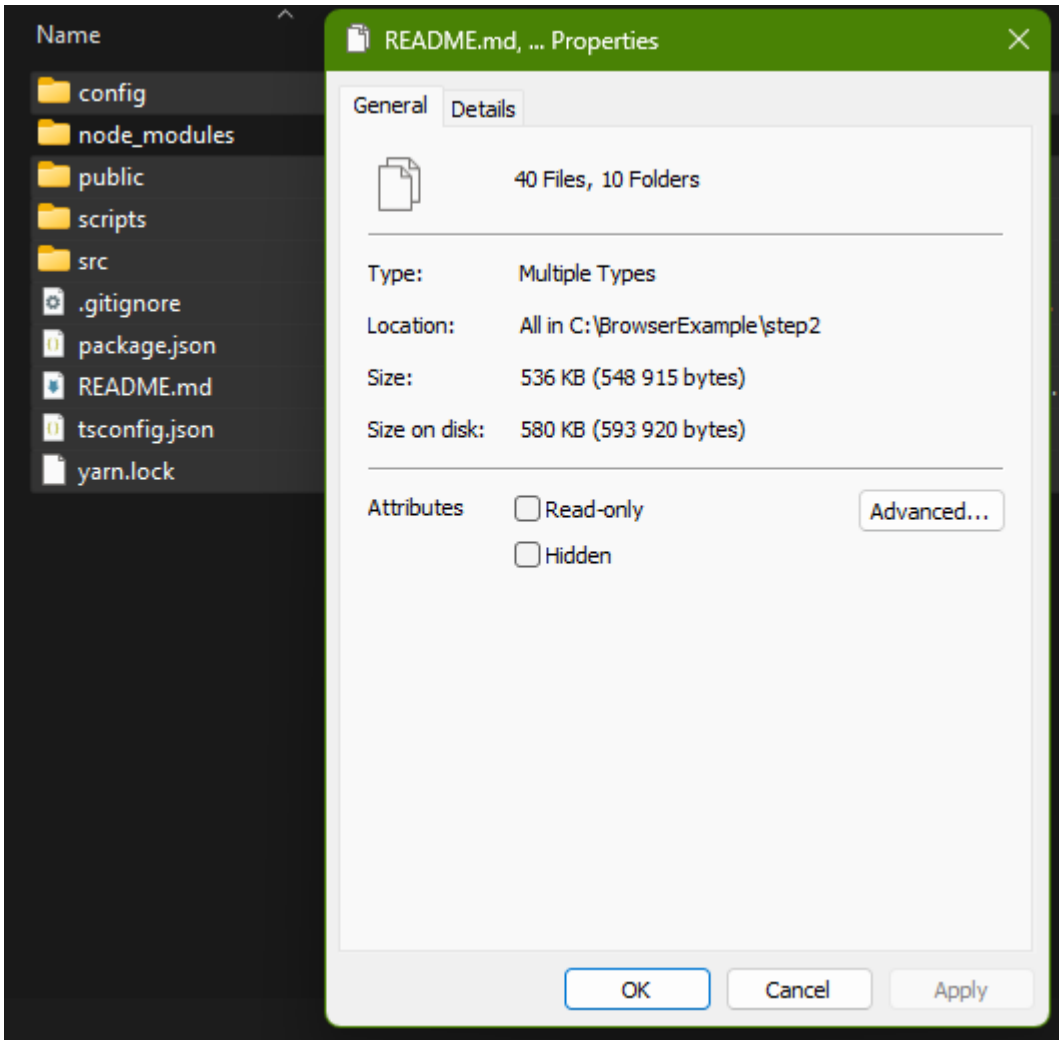

Image 2. File and folder count for automatically generated project content.

4.1.3 Browser implementation structure

4.1.3.1 User interface

The user sees 11 elements in the user interface; the current game status, nine board cells, and a restart button (image 3). The user interface at the end of the game can be seen in image 7.

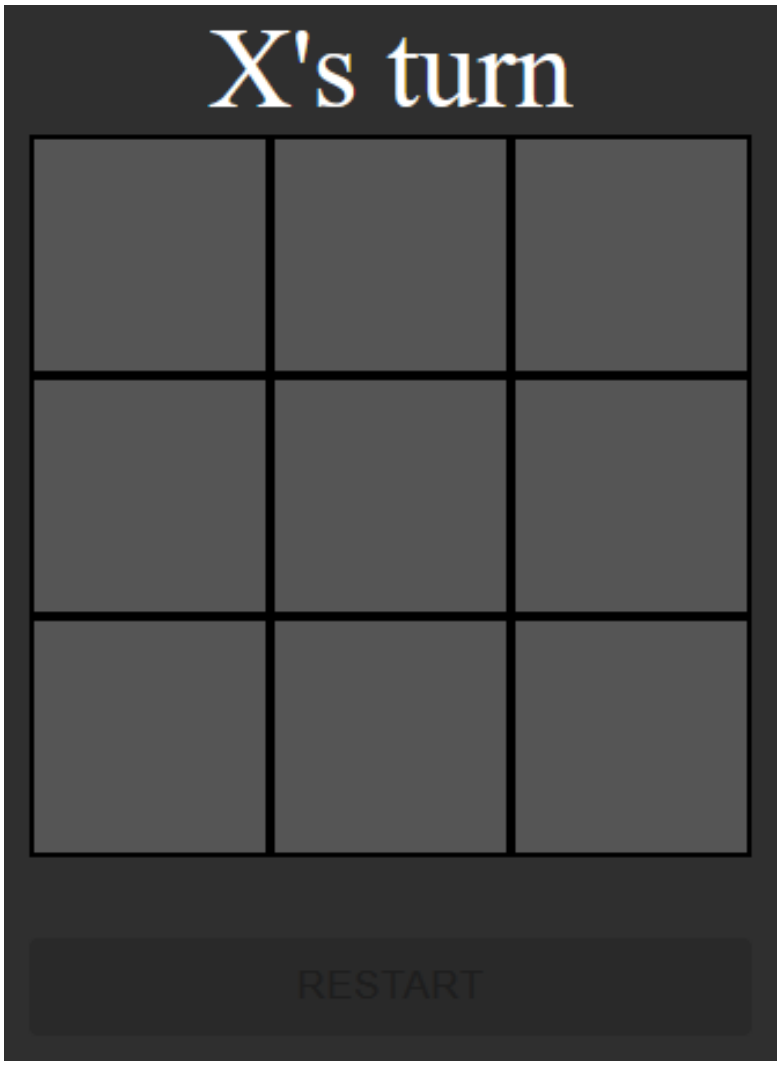

Image 3. Initial game state.

The game status element contains a message displaying which player is currently making their move, the winning player, or draw depending on the state of the game. The message component uses selectors to receive localizations and game status and selects the correct location to display depending on the game status. Additionally, it colours the message in a different colour if a player wins. (Image 4)

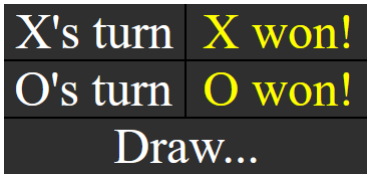

Image 4. Possible displayed messages by the message element.

The nine visible cells are contained within a board component. The board tells the nine cells which cell each of them is, as well as positions them in a three-by-three grid. Each cell then reads its status from the store using a selector, displays *nothing*, *X* or *O* depending on its status (image 5), and dispatches actions containing the clicked cell when clicked.

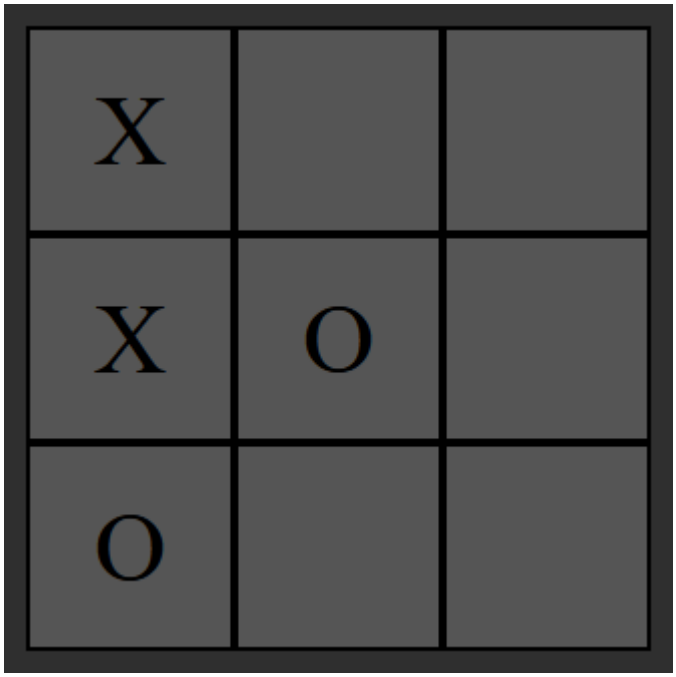

Image 5. The game board mid-game.

The restart button uses selectors to receive localizations and information whether a player has made a move. If no player has made any moves, the button is disabled. There is no point in setting the game state to the initial state, as it already is in the initial state. When a player has made a move, the restart button becomes enabled and clicking it will dispatch a *restart* action. (Image 6)

Image 6. Disabled and enabled restart button.

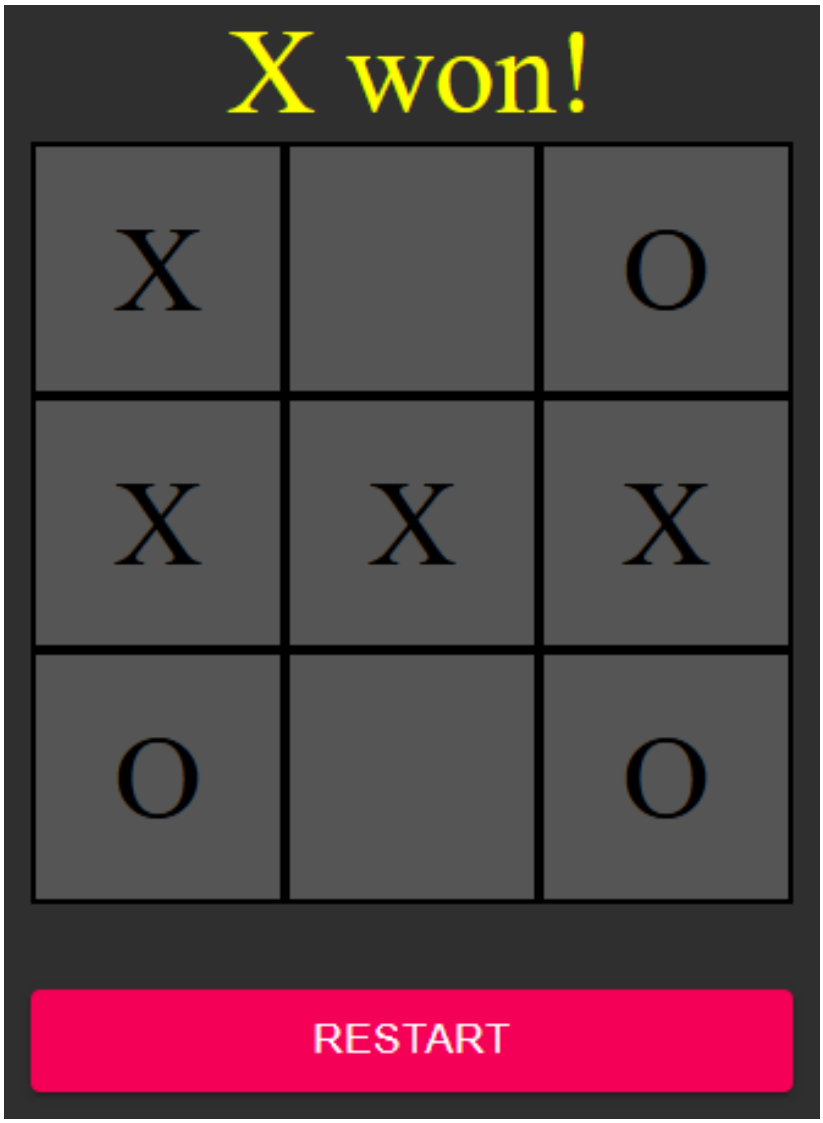

Image 7. Player X has won the game.

4.1.3.2State management

The state is managed by two reducers, the first managing the general game state, and the second managing the state of the board cells.

The general state reducer in *turn.ts* file contains the state of the game and current player. The game status has four possible values; *ongoing*, *X won*, *O won*, and *draw*. The initial state has game status as *ongoing*, and current player as *X*. When the general state reducer receives a *switch current player* -action, it flip-flops to the other player according to the current state. When it receives a

game end -action, it sets the game status to the value specified in the action, which can be *X won*, *O won*, or *draw*. When the reducer receives a *restart* action, it sets the general game state back to the initial state.

The board state contains nine values, one for each board cell. The state of the cell can be *empty*, *X* or *O*. When the board reducer in *board.ts* file receives a *set cell state* -action, it sets the cell specified in the action to the state specified in the action. When the board reducer receives a *restart* action, it sets the whole board state to the initial state, in which the state of each cell is empty.

The *on cell select* -saga is called whenever a *cell select* -action is dispatched. The saga then reads the game status from the store and stops handling the event if the game is no longer ongoing. Then the saga reads the current player and value of the selected cell from the store. If the cell is empty, the saga dispatches *set cell* -action to give the selected cell to the current player.

The *set cell* -action is then read by the board state reducer, which saves the owning player of the cell. The *set cell* -action also triggers the *on cell update* -saga. The saga reads the board state from the store and calls a function with it to receive an up-to-date game status based on it. If the game is still ongoing, the saga dispatches a *switch current player* -action, which is read by the general game state reducer. If the cell update instead results in an end state, the saga dispatches an *end game* -action with the end state, which also is read by the general game state reducer.

4.1.3.3Final browser implementation thoughts

In this implementation the board cells always dispatch actions on click and the saga decides whether the click should be reacted to, while the restart button element prevents interaction depending on the game state. In case of the cell elements, all the game logic has been removed from the UI element, and in the case of the restart button, a little bit of game logic, in the form of whether the

player can restart the game, is handled by the UI element. Following the Flux architecture more closely the logic from the restart button ought to be moved into the state and sagas, while conventions of Unreal engine would place the logic into the UI component. This is a balancing act between principles and practicality. The conventions defined by Flux and Unreal Engine are meant to make development easier, so for the sake of convenience it is acceptable to blur the line between the two practices, or UI elements and game logic.

In the end the browser implementation consists primarily of eight parts: interface elements, actions, utility functions, sagas, reducers, store, selectors, and localizations (figure 2). Correspondence for these needs to be found or created in Unreal Engine.

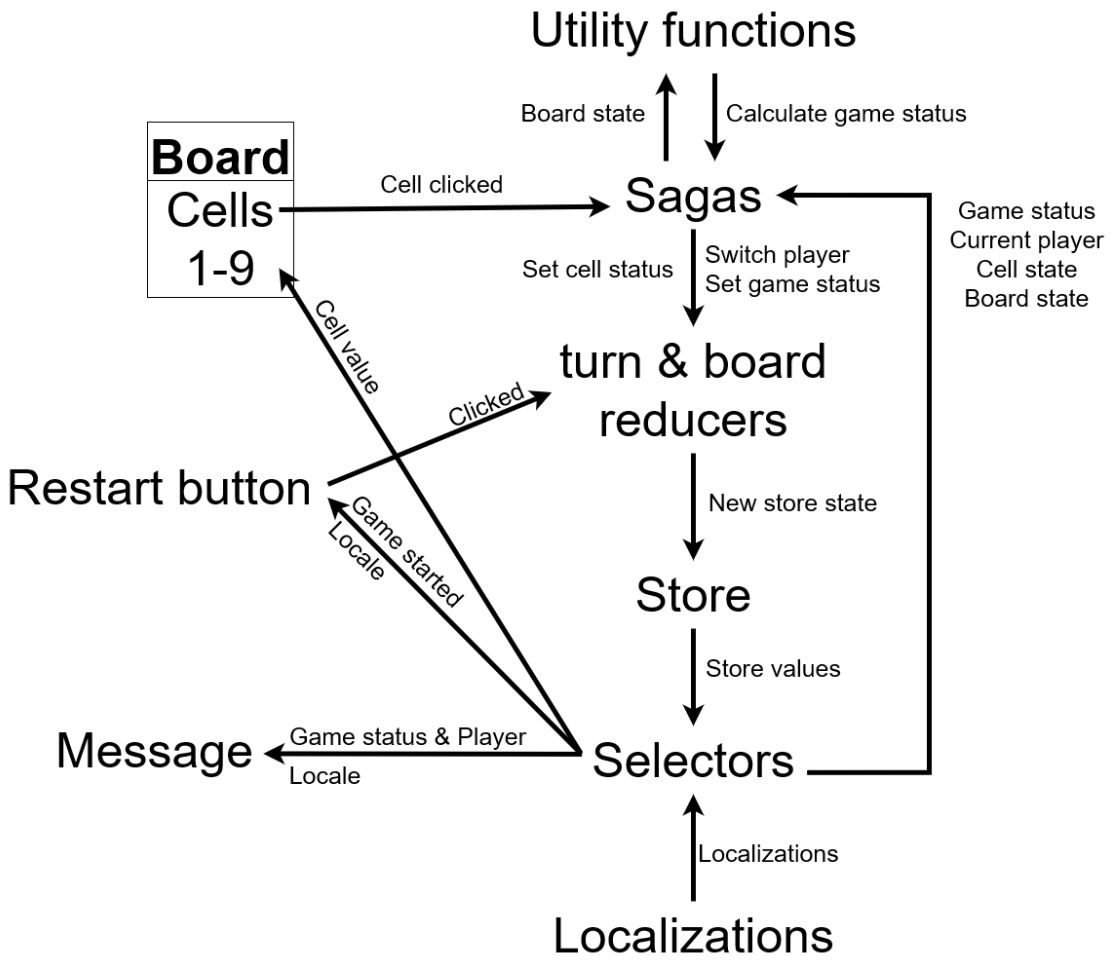

Figure 2. Module relations and communications.

Overall, with the templates and conventions provided by React and Redux and with the tree years of work experience the author was able to get the game to a working order in less than a day. It is likely easy to add new features, such as a score board, player name selection, or increased board size. For example, to add a score board, the game would have to merely require a new store containing the number of games won for each player, a reducer which increments these values whenever a game ends, and an UI element displaying the scores. The ease at which other people would be able to join in on the programming should mostly depend on whether they are familiar with the conventions of TypeScript, React, Redux and sagas, but this remains as no more than an assertion without conducting further studies, as the focus of this study is to transfer the implemented structure to Unreal Engine instead of the ease of project code onboarding.

4.2 Correspondent components between the browser implementation and the tools provided by Unreal Engine

4.2.1 Unreal Engine overview

In addition to traditional C++ programming Unreal Engine enables developers to implement their logic using visual scripting through interfaces provided by the Unreal Editor (Epic Games 2022d). This way of interfacing with game development helps visualize logic flow and other aspects of the game, as well as increases accessibility for those who are not familiar with C++ or programming languages in general. Unreal Engine offers an object-oriented paradigm for structuring logic flow, data storage and how different parts of the game interface with each other through, for example, the project templates (Epic Games 2022e) and blueprint classes (Epic Games 2022f). Learning to use them in the intended way can take a while, but what if existing knowledge of React and Redux could be applied?

4.2.2 User interface

In Unreal Engine, interface elements can be implemented using widgets (Epic Games 2022g). Widgets can contain multiple interface elements. For example, a menu containing multiple buttons and information displays can be implemented using a single widget blueprint. Elements within can have their properties, for example, text or colour, bound to either a variable within the widget, or a function, which returns the property (image 8).

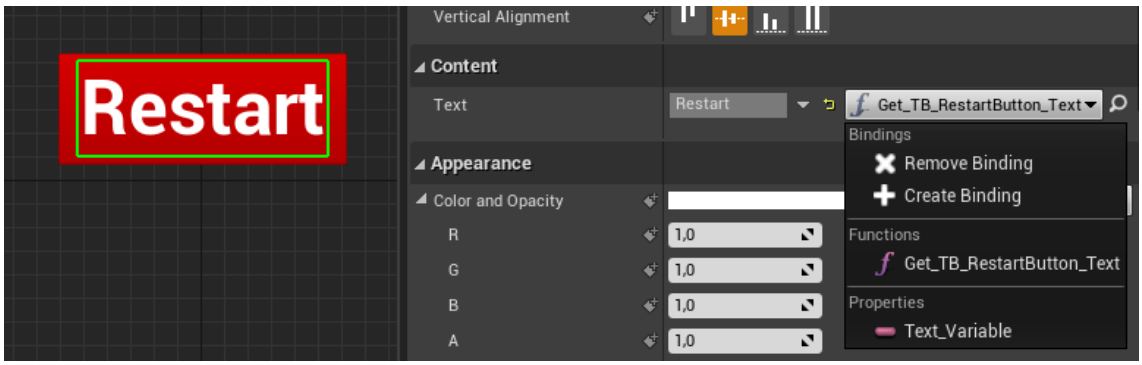

Image 8. A widget element property can be bound to functions or variables.

If a property is bound to a function, the function could use selectors to read the store state, and return a desired property based on that. If a property is bound to a variable, the variable needs to be updated using some other method

whenever the store state updates. Elements can also trigger events (image 9). The events can then trigger the dispatching of actions to the store (image 10).

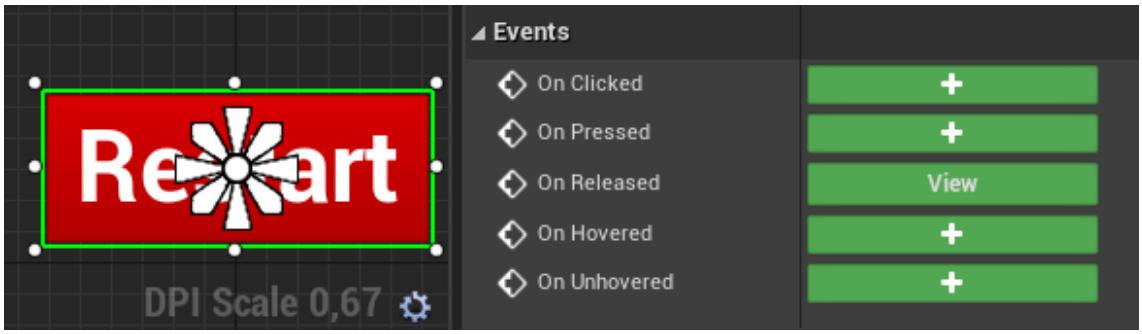

Image 9. Button elements have five different events available.

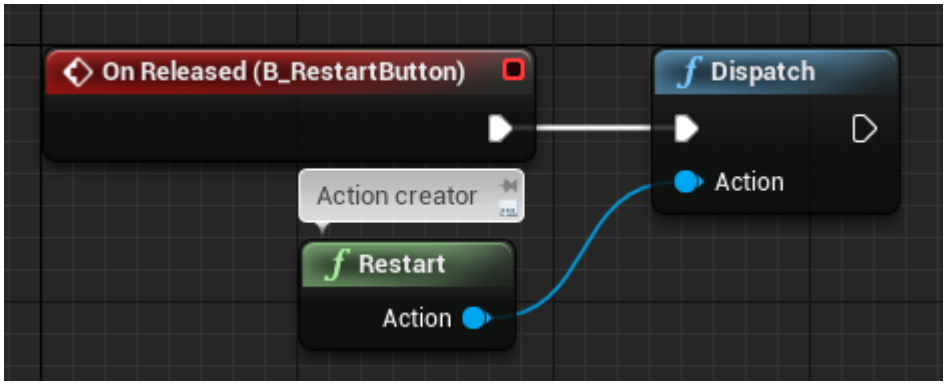

Image 10. On Released event dispatching a *restart*-action.

4.2.3 Actions

Actions are objects which include two values: *type* and optionally *payload* (Meta Platforms 2022b). Unreal Engine's equivalent in blueprints of a JavaScript object is a structure (Epic Games 2022h). The action *type* indicates what the message is. This implementation of tic-tac-toe includes five action types: *cell select*, *set cell state*, *switch current player*, *end game*, and *restart*. The action *type* can be stored as an enumerator (Wadstein 2015a), which is a collection of strings, and allows defining valid types for the action *type*. The action type could also be stored as a string in the action structure. This would allow creating actions with any arbitrary types. A string type action type could be a valid option in larger projects, as the amount of action types in a larger application can reach hundreds and managing a single enumerator with hundreds of values can

be cumbersome. On the other hand, having a single source of truth for available action types does lessen the probability of programming errors. The optional *payload* value in an action can contain information related to the action. This implementation of tic-tac-toe has three different payloads which contain *selected cell*, *cell and cell status*, and *end status*. The reducers receive every action which gets dispatched in the application, even if the actions do not cause them to update the state. This means that the action payload can be any of the payloads defined in the application, which in TypeScript terms is an union type (Microsoft 2022). Union types define a value to either be of type A or type B. Unreal Engine does not natively support union types, which means it is not possible to create a structure with all potential payloads defined using a union type.

One alternative is to define structures for each payload and add each of them under a different key in a generic action structure. This approach could work if there are not many places reading the same data from different actions. For example, in this implementation the *selected cell* data is included in *select cell* and *set cell status* actions. If there was logic which should react to all actions containing *selected cell*, said logic would have to know which action payload key to inspect.

Another approach would be to define a singular payload with all possible data from different actions. This would retain the basic *type* and *payload* key structure of the actions but would require making sure the keys inside the payload do not conflict with the data types from different actions. In a larger game with more data moving around the payload structure would quickly become too large and unwieldy to manage effectively.

A third option could be to create a key-value maps, and parse them wherever the action is read, but again due to Unreal Engine not supporting union types, the values would have to be converted into strings, assuming they even are a data type which can be converted to a string.

A fourth possible method of implementing actions could be to use blueprints instead of structures. Actions with payloads can inherit the *type* enumerator variable form a *payload*-less parent blueprint class. Blueprints can be cast to and from this parent action blueprint class, allowing us to pass different actions through the same channel. The downside to this approach is that it is required to know which child action blueprint class to cast the action when reading its contents, but this can be deduced from the action type common to all actions.

4.2.4 Utility functions

Utility functions are mostly pure functions, meaning they produce output values based on the input values they receive when called. For this purpose, blueprint function libraries (Wadstein 2015b) or blueprint macro libraries (Epic Games 2022i) can be used, as they can group multiple functions or macros together. Impure function nodes always have one input and one output execution pin, meaning there is only one code execution path available before and after a function node, while pure function nodes have no execution pins, and will always be executed when data from the function is used (Epic Games 2022j). The limitation of only having a single output execution pin can be circumvented, for example, with a branch node controlled by a return value of the function.

Macros on the other hand can have multiple execution inputs and outputs, or none of one or the other (Epic Games 2022k). Macros with multiple outputs can serve as an analogue to callback functions (Mozilla 2022a) in JavaScript, meaning different code execution paths can be taken based on the inputs. A downside to macros is that they cannot contain local variables, but this limitation can be somewhat worked around by passing along the data pins instead of saving and reading variables.

4.2.5 Sagas

From practical perspective sagas are functions triggered by actions, which can in turn read the store, call utility functions, and dispatch new actions. Saga functions are not called directly, so they do not have return values. Due to no logic execution taking place directly after a saga ends, blueprint functions can be used for branching logic without the need for the ability of to have multiple output execution pins.

4.2.6 Reducers

From a practical perspective reducers are pure functions, which receive the old store state and action, and return a new store state. The store can be divided into multiple sections. Each section is updated by their own reducer, each reducer having visibility only to their own section of the store, also referred to as state, or store state. Due to reducers having one input and output path means they can be implemented using blueprint functions.

4.2.7 Store

The store holds all the data which is of interest to multiple actors in the application. For example, interface elements and sagas can depend on the game status. The store also receives all the actions passing through the application and passes them along the store state to all reducers and root sagas. There is always only one store. While implementing a store would be possible with a basic blueprint, a game state blueprint (Epic Games 2022l) can serve us better here. There is only one game state object in play at a time, it has a construction script (Epic Games 2022m) where the state can be initialized, if need be, and in an online multiplayer game it gets mirrored to all

connected players. The store also needs to be able to receive actions from sagas and interface elements.

4.2.8 Selectors

Selectors are pure functions, which receive the store state, and return a value from it. For example, a *get current player* selector would read the player from the store state, and return either *X* or *O*. A blueprint function library will work well to group these together. In the browser implementation the selectors are passed to an *useSelector* hook (Abramov 2022b) as a parameter, which then calls the selector with the store state and returns the return value of the selector. Functions are difficult to implement as parameters using only the visual scripting of Unreal Engine. One possible workaround could be to use blueprint interfaces (Epic Games 2022n) and casting, but selectors can have different return types, and a new blueprint would have to be added for each selector, of which there can be tens. Considering the complexities of one-to-one implementation of selectors from the browser it is a fine compromise to have the selector functions themselves read the store state from the game state, even if this means they are no longer pure functions. Another partial implementation would be to read the whole store state where the selector is called and pass the state to the selector.

4.2.9 Localizations

In the browser implementation the localizations are typed objects, each language having its own object with a matching typing and structure. Translations can be grouped and nested to move translations relating to the same subject together. For example, main menu button translations could be grouped together. The translations can then be saved into their own files, one for each language.

One way to implement nested translations in Unreal Engine is to define the locale structure using multiple structures, creating a data table (Wadstein 2017) based on it, and adding a row for each language. The locale selector can then fetch the localization structure for the correct language based on the data table row name. A large downside to this approach is that each group and level of nesting requires creating a new structure for it, but some upsides include all localizations having the same structure, ability to edit localizations inside the editor, as well as the ability to export and import *.csv* and *.json* file formats to work on the localisations outside of the editor. Unreal Engine also has composite data tables (Wadstein 2018), which are collections of data tables based on the same structure. Spreading localizations into separate data tables and combining them into a composite data table enables multiple people to work on translations for different languages at the same time without encountering problems with file versioning.

Unreal Engine also has string tables, which at first glance sounds like fitting asset type for storing localization strings but accessing and using string tables directly from blueprints is limited (image 11). However, string tables are closely tied to the robust localization pipeline of Unreal Engine (Epic Games 2022o). String table values can be directly referenced in code, making them a single source for truth (image 12). Progress for different languages can then be tracked and done through the localization dashboard and translation editor. Localizations can also be exported and imported in *.po* file format. The localization pipeline of Unreal Engine also offers many other features to enable a workflow for professional production environments, but for a beginner working on a small project getting it to work can be challenging (image 13).

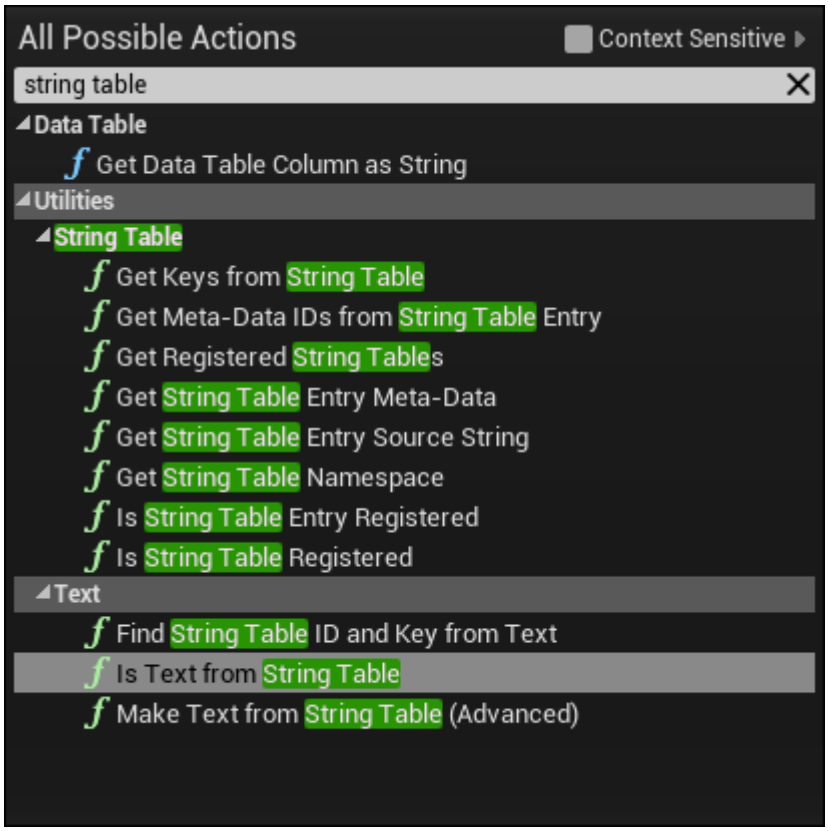

Image 11. Available string table actions.

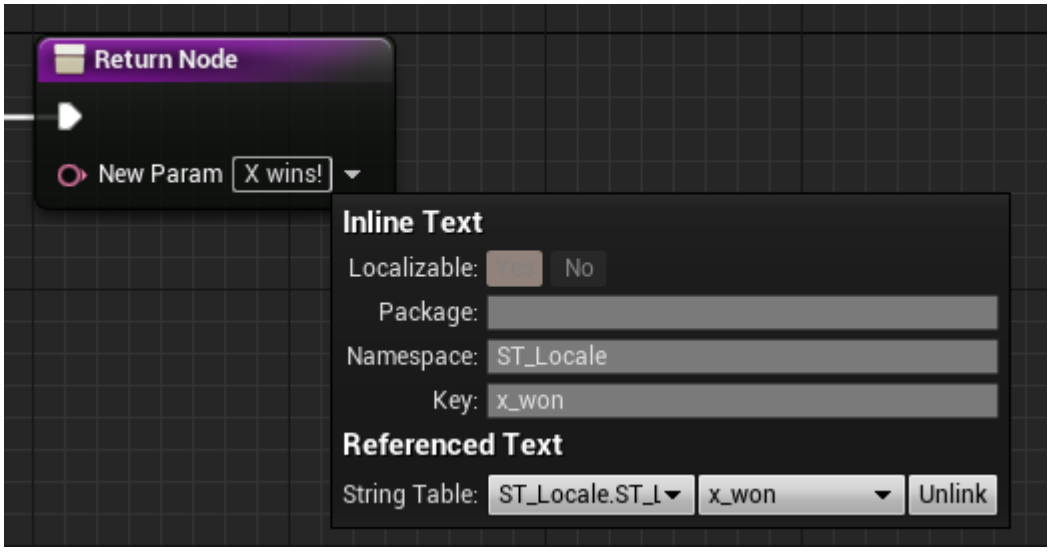

Image 12. String table value reference in a blueprint.

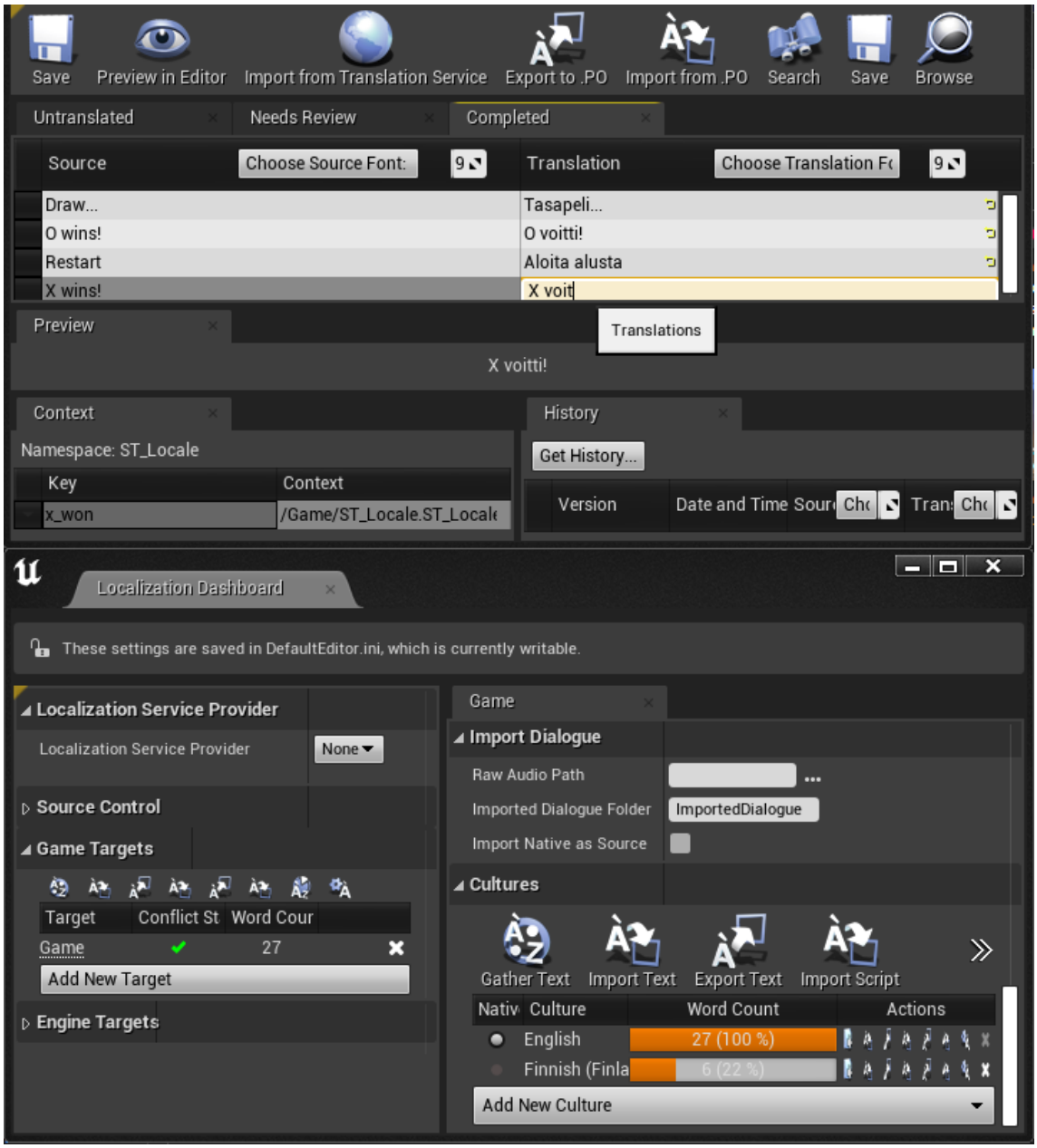

Image 13. The translation editor and localization dashboard of Unreal Engine.

4.3 Implementation in Unreal Engine

4.3.1 User interface

The user interfaces were implemented using widget blueprints. There is little reason to develop something new to replace the existing and extensive tools and resources in Unreal Engine for interface creation (Epic Games 2022p). The widget blueprint elements get relevant information from the store using

selectors, and communicate actions taken by the user to the application by constructing action blueprints and dispatching them (image 10).

4.3.2 Actions

Different actions need to be passed through the same channels while also possibly having contents with differing structures. Blueprint inheritance was used due to the lack of native support for union types in Unreal Engine. The base action is a blueprint with a single variable for storing the action type. For specific actions, child blueprints can be created from this base parent blueprints (image 14).

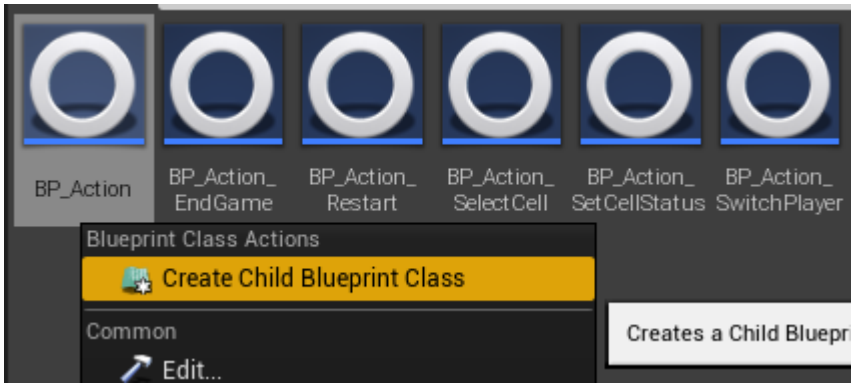

Image 14. Create action child class from parent.

The created child action blueprint can then, for example, define the action type to be of type *SET_CELL_STATUS* and have a payload structure (image 15) variable containing the specified cell and its status (image 16).

Image 15. *set cell status* -action payload structure.

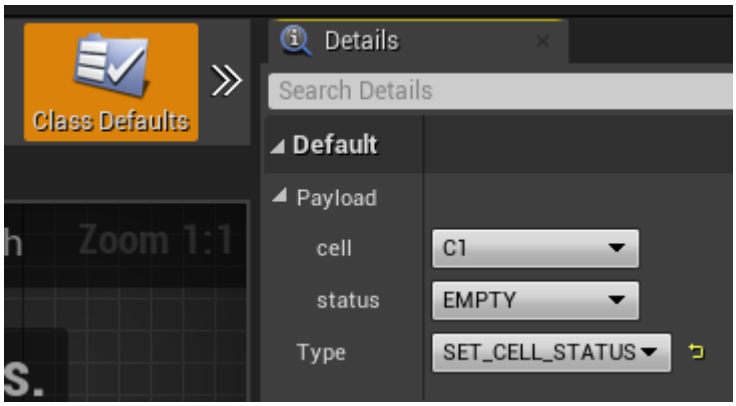

Image 16. *set cell status* -action structure and default values.

Exposing the payload variable on spawn (image 17) makes creating instances of these actions easier (image 18).

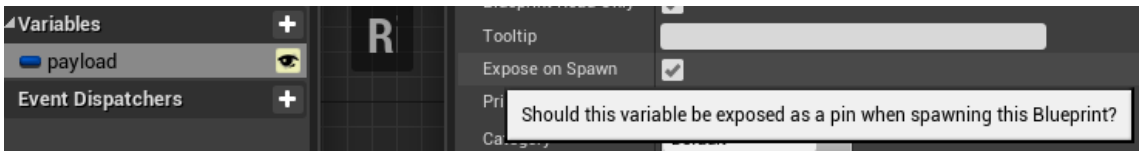

Image 17. Expose action payload on spawn.

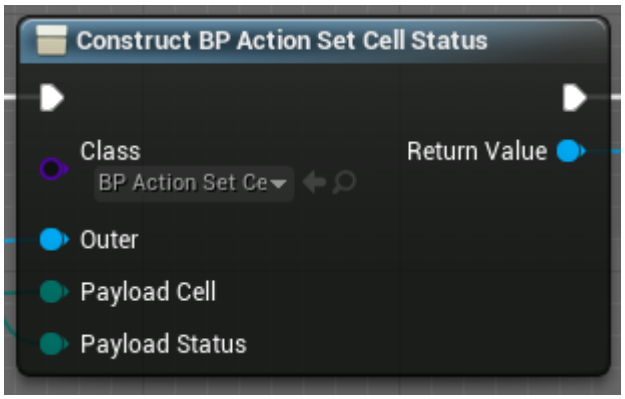

Image 18. Set exposed action payload values during construction.

Action creators are pure functions for constructing instances of the action blueprint classes (image 19, image 20), and a blueprint function library was created for storing them.

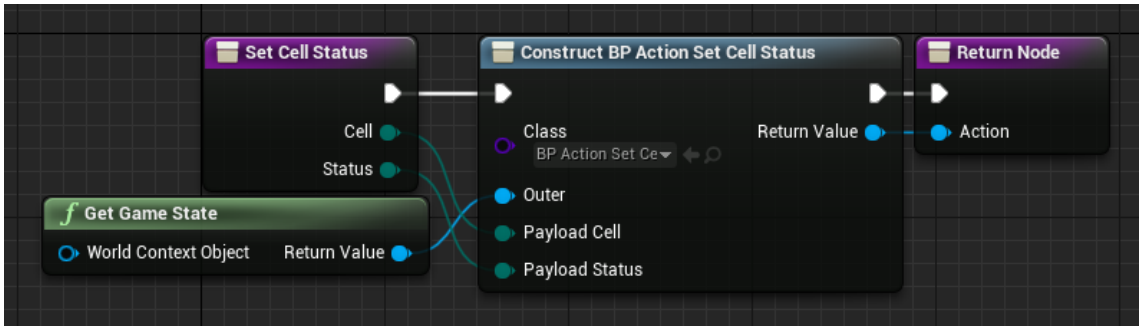

Image 19. Structure of *set cell status* -action creator.

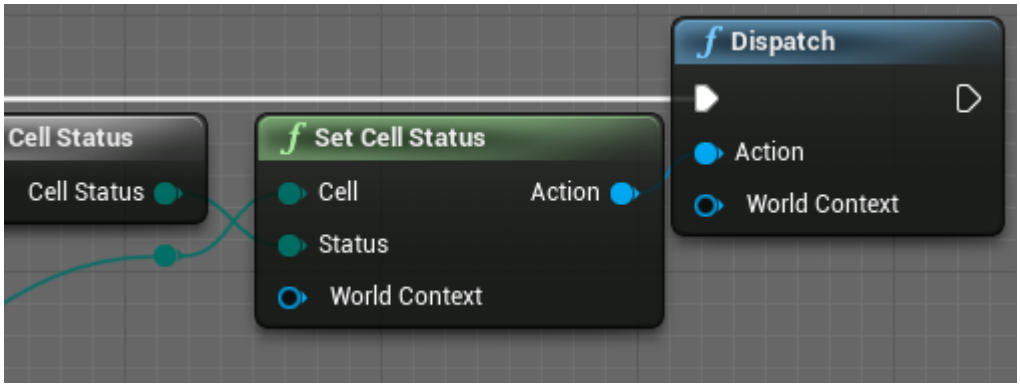

Image 20. *set cell status* -action creator called.

4.3.3 Utility functions

Blueprint function libraries and macro libraries both have their strengths and weaknesses, so both were used for utility functions. For example, blueprint functions can get the game state (image 21), while macros can control logic flow (image 22).

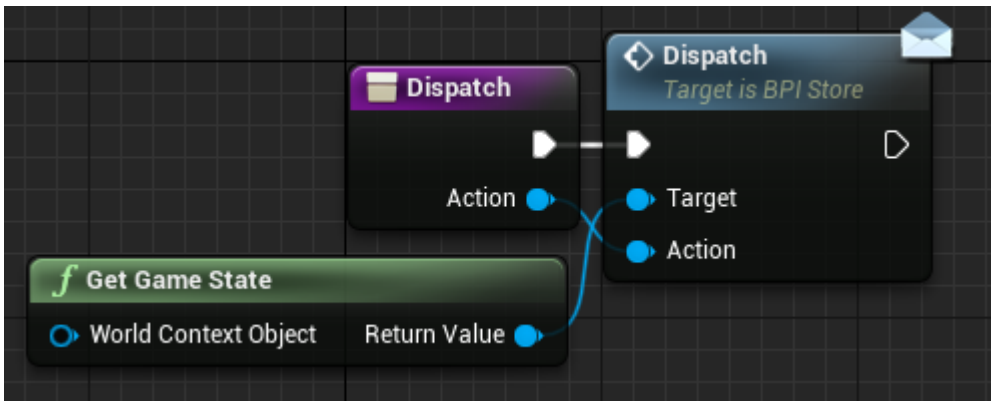

Image 21. Blueprint function accessing game state.

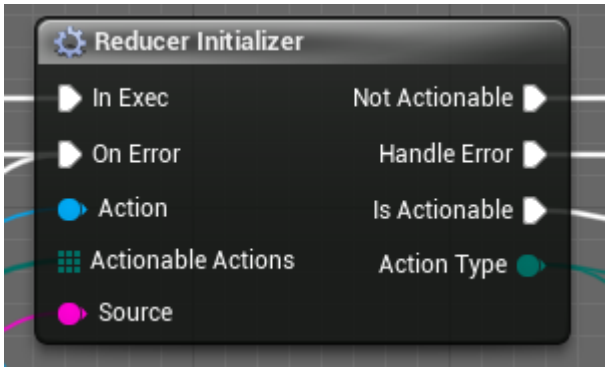

Image 22. A macro controlling logic flow.

During the development the author encountered situations where certain blueprint functions could not be added to the event graph, and certain macros caused errors. With a deeper understanding of the functionality and correct practices in Unreal Engine these problems could likely be avoided or fixed, but with the limited resources available it was faster to move the problematic logic from a blueprint function to a macro or vice versa.

4.3.4 Sagas

For sagas a blueprint function library was once again used. In the browser implementation sagas are generator functions (Mozilla 2022b), but Unreal Engine blueprint scripting does not have analogous functionality available. For more complex sagas which wait for some other logic to complete execution to continue this could become a problem, but in this implementation of tic-tac-toe the sagas are functions which receive an action, read the store state, and immediately dispatch new actions, so regular blueprint functions fit the job. There is also a root saga (Redux-Saga 2022b), which calls the appropriate sagas depending on the action type (image 23).

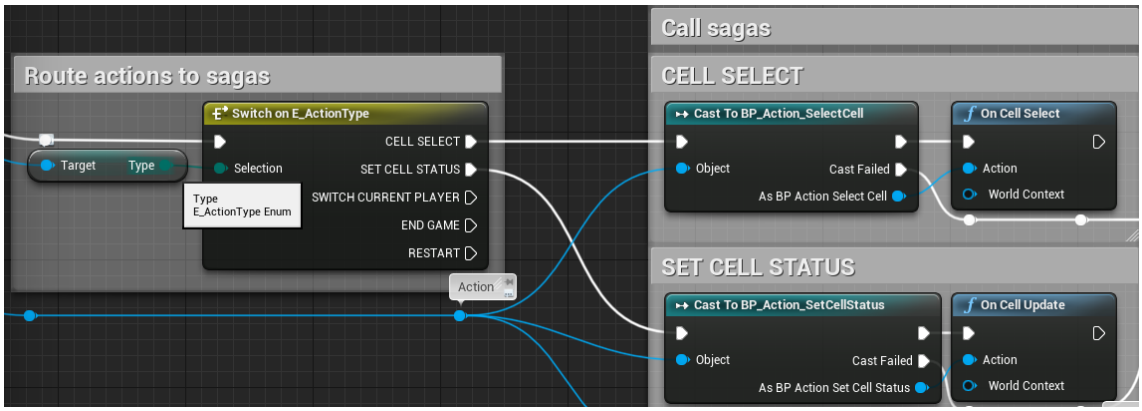

Image 23. Root saga calling sagas.

4.3.5 Reducers

For reducers a blueprint interface and child blueprints inherited from a common base blueprint were used. The interface defines a function, which receives the old state as a generic state blueprint object and a generic action blueprint object and returns the updated state as a generic state blueprint object. The base blueprint also has a reducer type variable for identifying the reducers from each other. The child reducers then specify the reducer type and contain logic for manipulating the state based on received actions. For example, the board reducer receives the part of the store state containing the board, and if the action is of type *set cell status*, the reducer creates a new board state blueprint

instance with modified values and returns it. It is more scalable to use blueprint interfaces and inheritance, as it makes it possible to call all reducers with all actions without resorting to ever-expanding switch cases or calling all the reducers in a sequence (image 24).

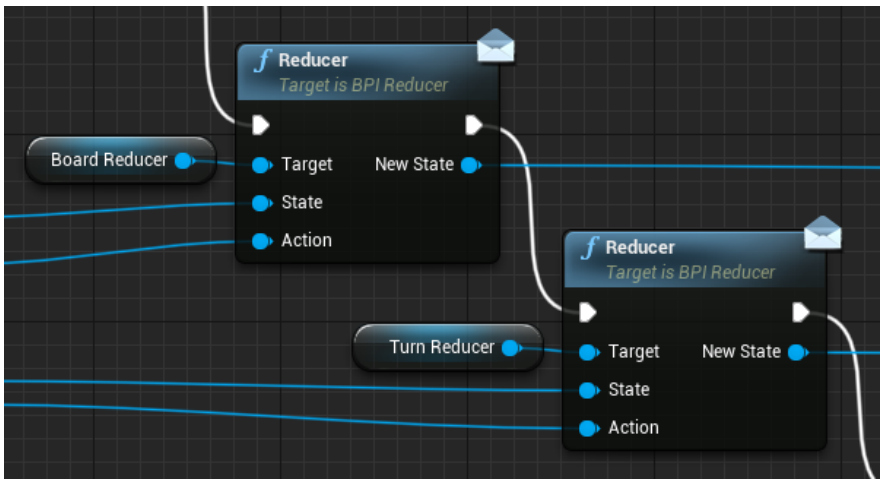

Image 24. A theoretical implementation where all reducers are called sequentially.

4.3.6 Store

The job of the store is to maintain the state, receive actions and pass them to sagas and reducers, and update the state with values received from reducers. For this a game state blueprint was used, which gets spawned in the beginning of the game by the game mode blueprint. An additional benefit to storing the state in the game state instead of some generic blueprint is that it is easy to get a reference to the store with the *Get Game State* node from most places (image 25).

Image 25. Reading the store state in a blueprint.

The game state blueprint also calls the reducers and sagas (image 26). As shown in images 26 and 27, calling of the reducers and constructing the store states were not turned into a truly generic and scalable implementation in the blueprints, as *select* and *switch* statements are still in use.

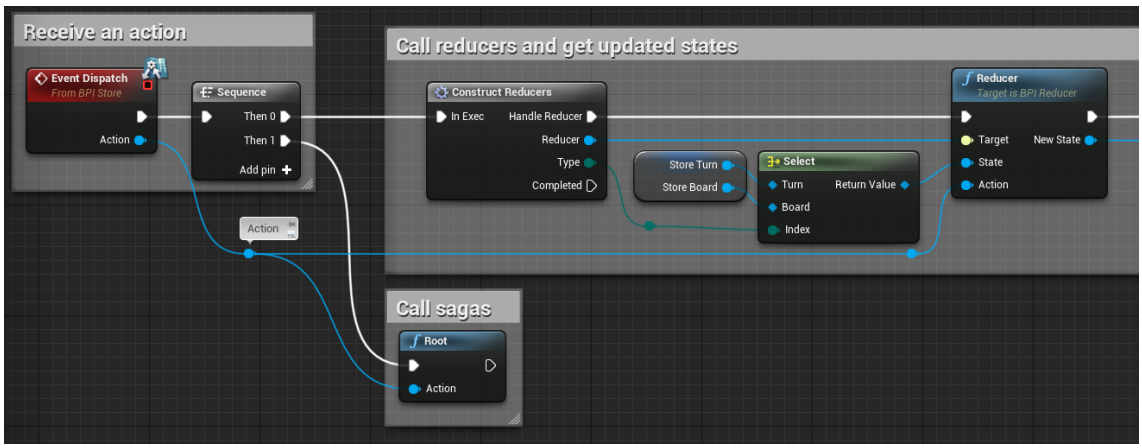

Image 26. Store distributing actions to reducers and sagas.

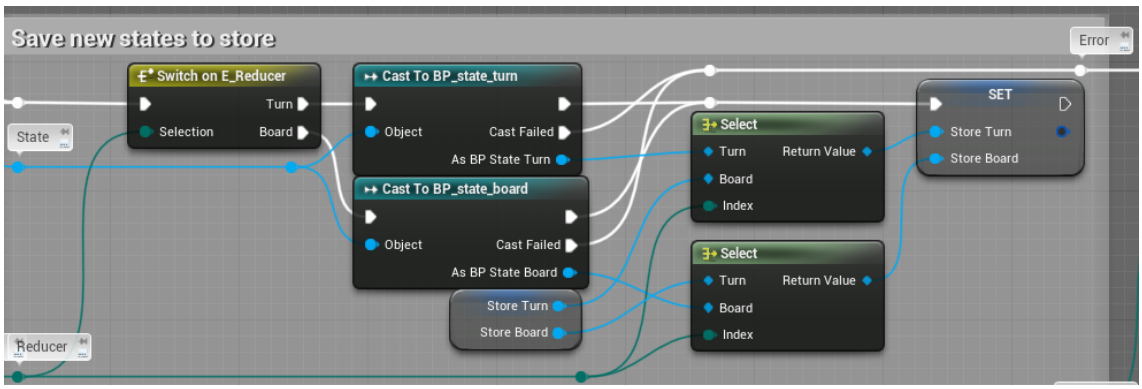

Image 27. Store updating state with new states.

4.3.7 Selectors

Selectors were implemented using blueprint functions stored in a blueprint function library. Selectors receive the store state as a structure and return the desired value from within that structure. Because they return atomic parts of the store state, as the size of the store state increases, so does often the number of selectors. Even this basic implementation of tic-tac-toe has seven selectors. A blueprint function library is a single asset capable of storing a large number of

functions. These functions can also be categorized to improve maintainability and discovery for development purposes.

In the browser implementation selectors are passed to a *useSelector* hook which then calls the selector using the store state as a parameter, but there is no straightforward method to implement hooks with functions with differing return values as parameters in Unreal Engine blueprints. Thus, in place of the useSelector hook, a function which finds the game state from the world (image 25) and returns the store state stored in it as a structure, was created. Then said store state structure is passed to the selectors whenever they are called (image 28).

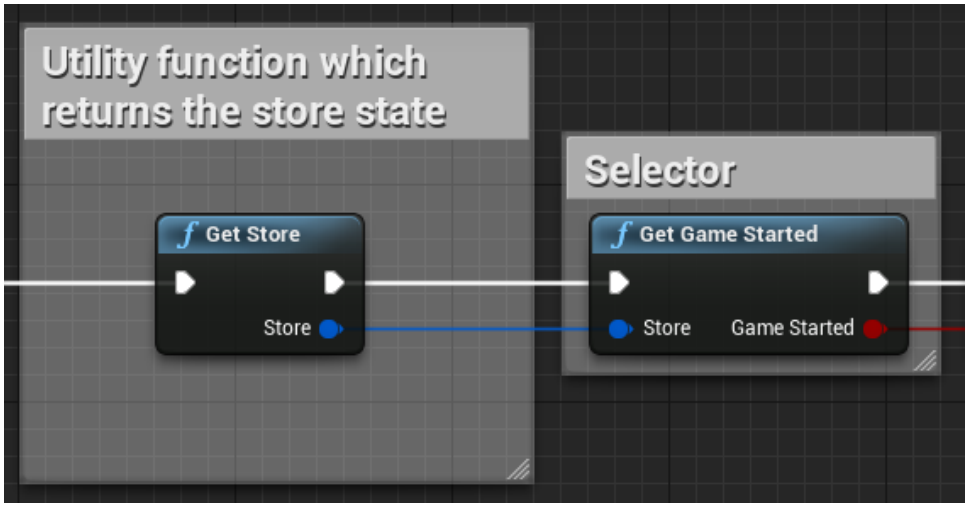

Image 28. Store state structure received from a utility function is passed to a selector.

4.3.8 Localizations

Nested structures were used for localizations. This allows grouping and organizing localizations related to similar topics. Structure values can be given default values (image 29), and by creating multiple structures with the same shape, multiple localizations can be created.

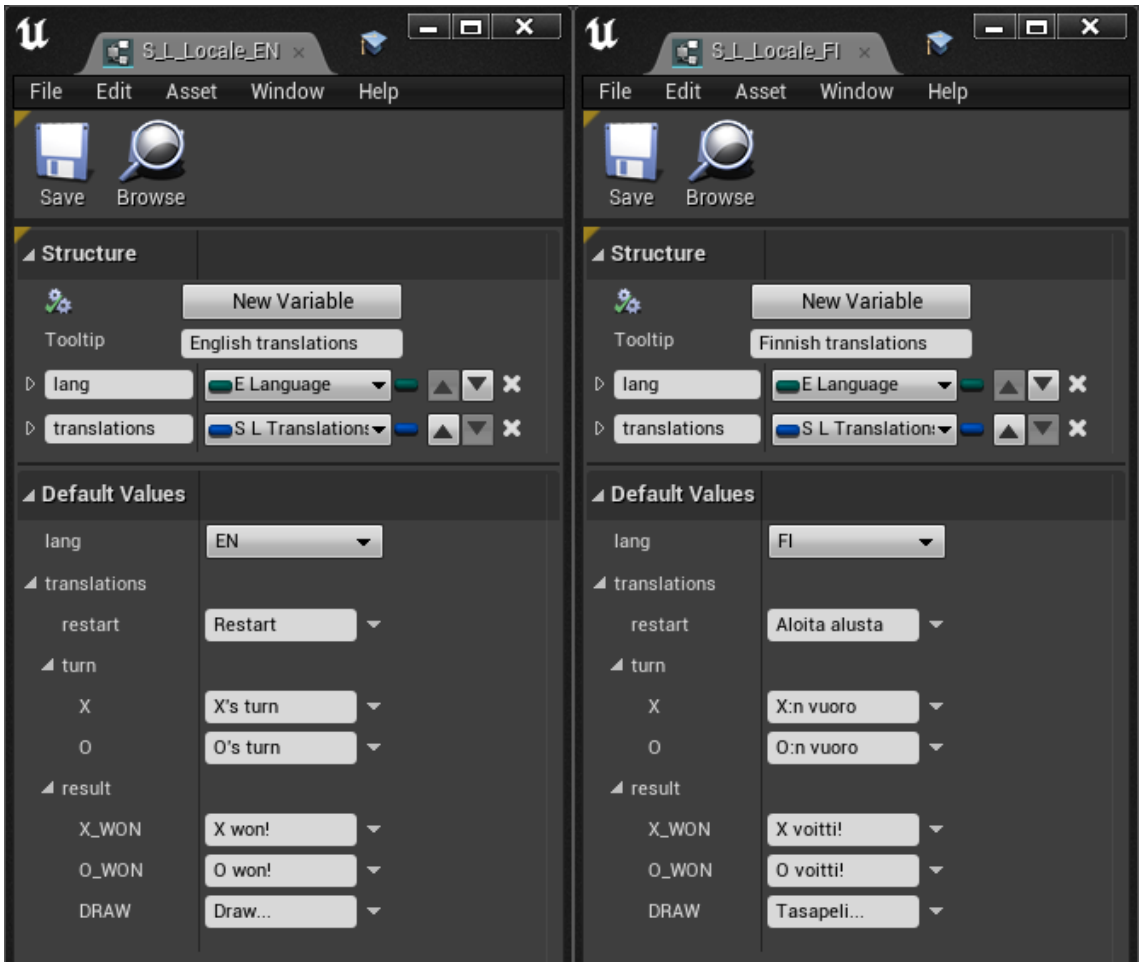

Image 29. Structures containing English and Finnish translations as default values.

The language selector can then construct these language specific structures, and because they have identical shapes, a select node can be used to pick the desired localizations (image 30). This implementation of tic-tac-toe does not have language selection inside the game, but by editing the blueprint the selector can be seen working (image 31).

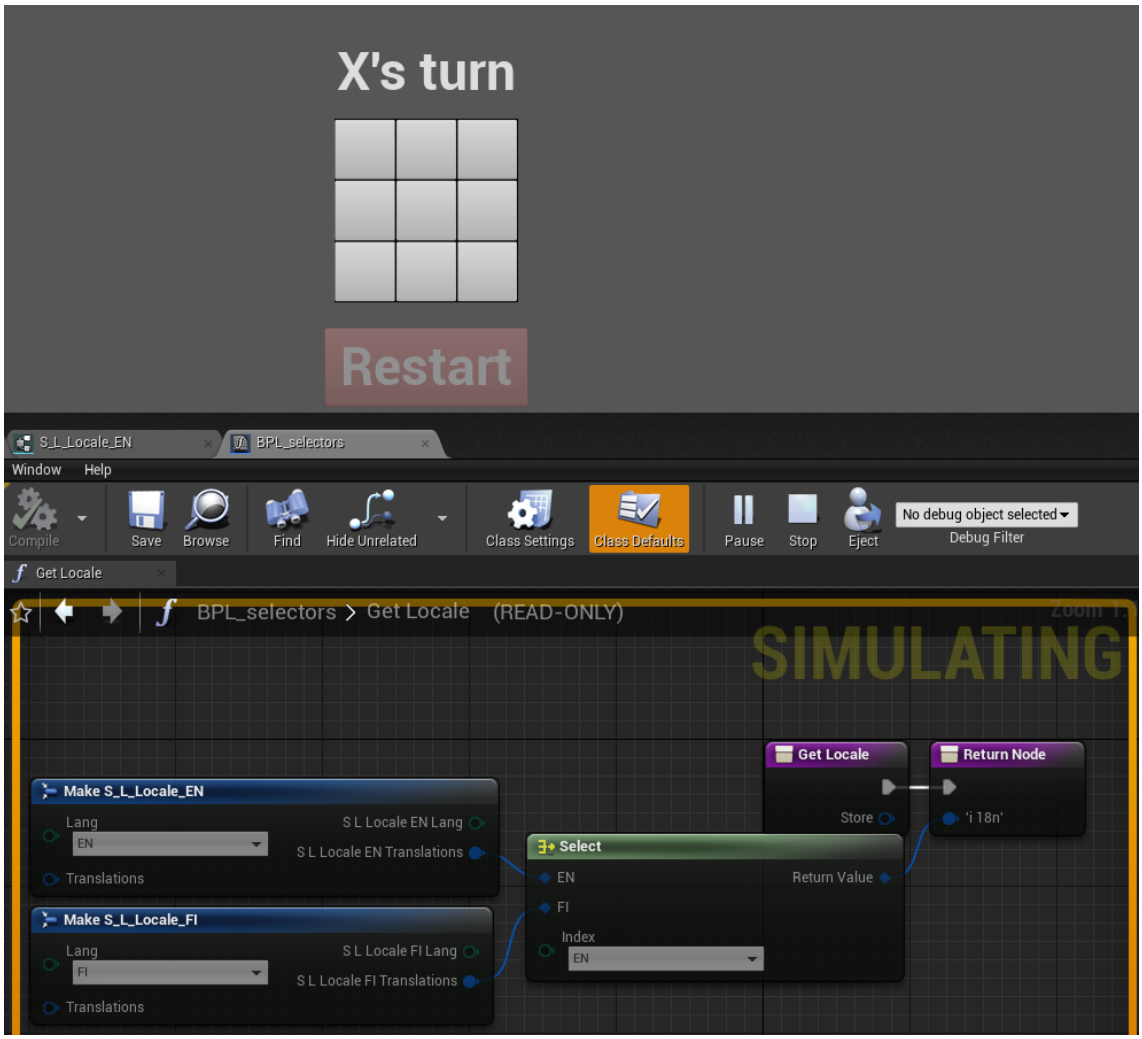

Image 30. Localization selector returning English translations.

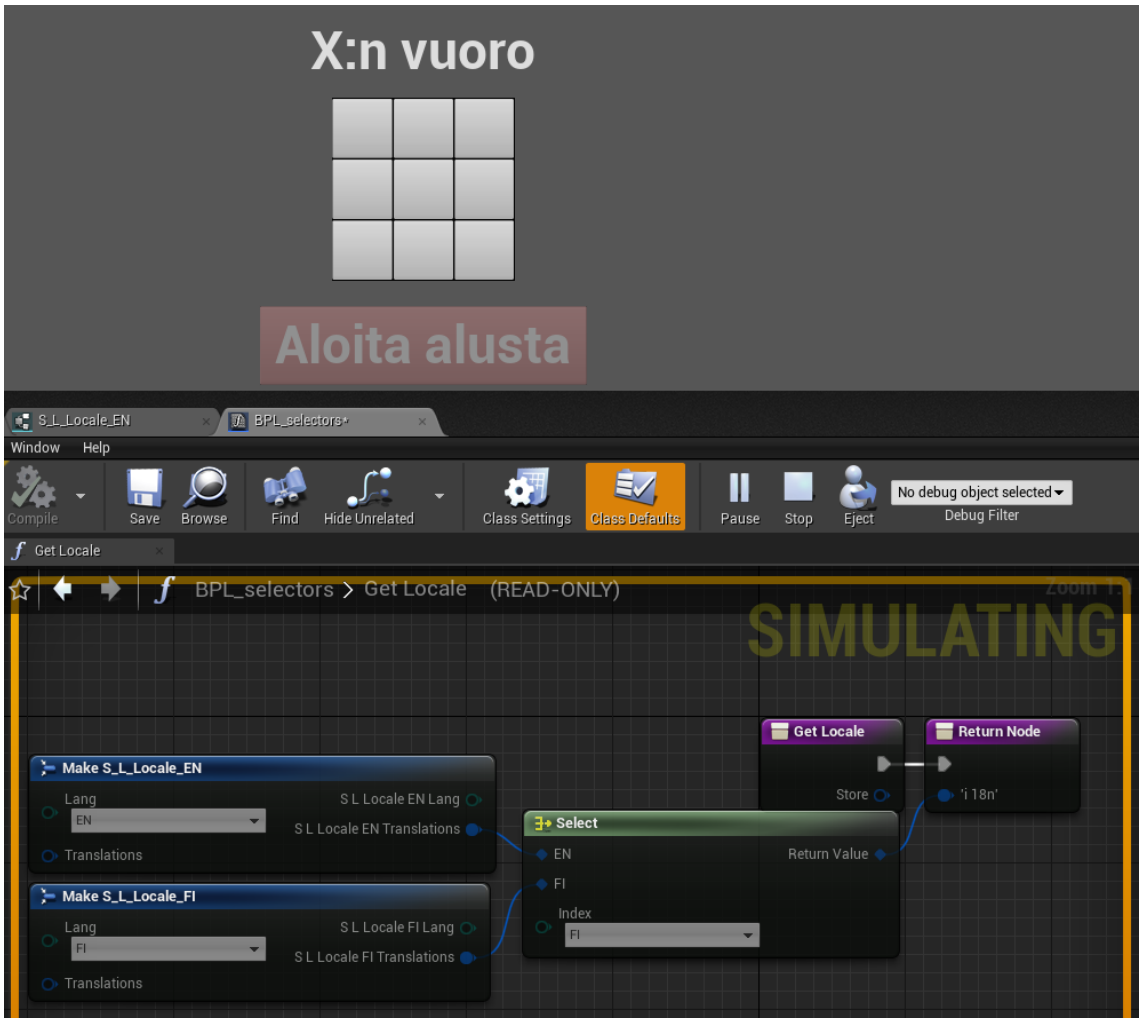

Image 31. Localizations selector returning Finnish translations.

5 Results and discussions

5.1 User interfaces and localizations

User interfaces and localizations are common elements to most games, so there is no need to reinvent the wheel for these. A downside to the chosen localization approach utilizing nested structures is that each group requires creating a new structure asset. In a medium sized project this can easily mean dozens, possibly hundreds of structure assets for defining the shape for accessing specific localizations. If a developer wants to move a localization or a group of localizations to a different location in the structure hierarchy, they must

move every single affected localization for every single language to the new location, and if they remove the old structure before this, they lose the translations. If the localization structures store the translations as *Text* instead of *String*, the built-in localization pipeline of Unreal Engine can be applied on top of the existing implementation. The author heavily recommends using the built-in localization pipeline of Unreal Engine from the beginning (image 32). The chosen implementation unnecessarily adds a layer of complexity and slows down development.

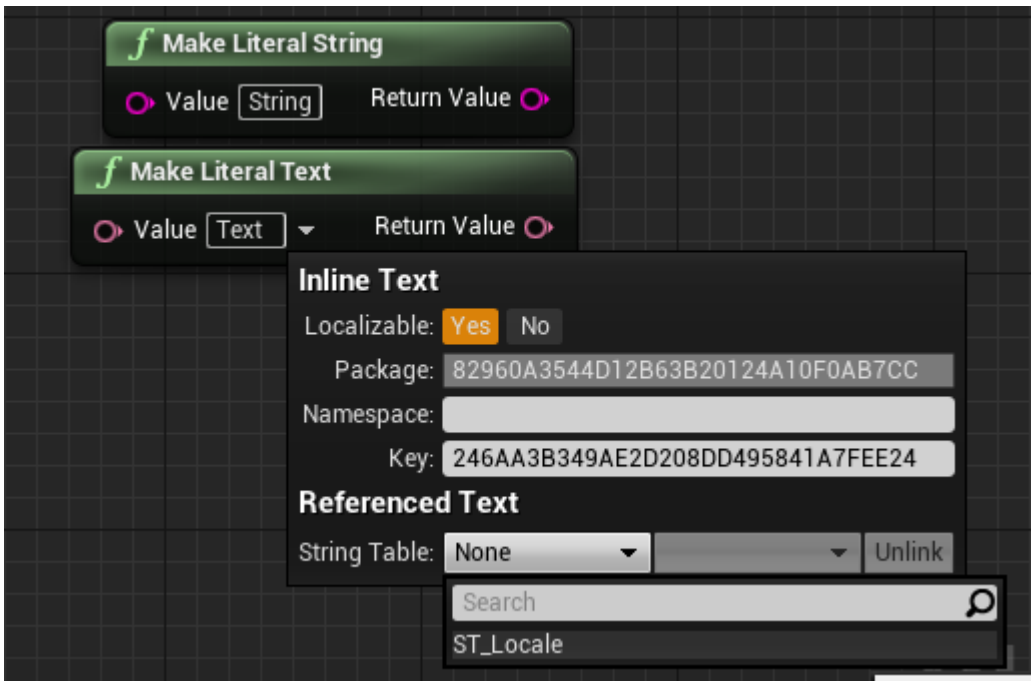

Image 32. An example of string and text types in blueprints. Text can be localized without the developer having to write logic for localizations.

5.2 Utility functions and state management

Utility functions are normal functions, a concept found in most programming languages. Selectors are functions with inputs and outputs, requiring nothing new on a technical level. Actions, states, and reducers all can utilize blueprint inheritance to a great effect, but reducers could use streamlining (image 33) in the form of common utility functions for de- and reconstructing the state to make

adding and removing new reducers and action cases (image 34) to them less burdensome.

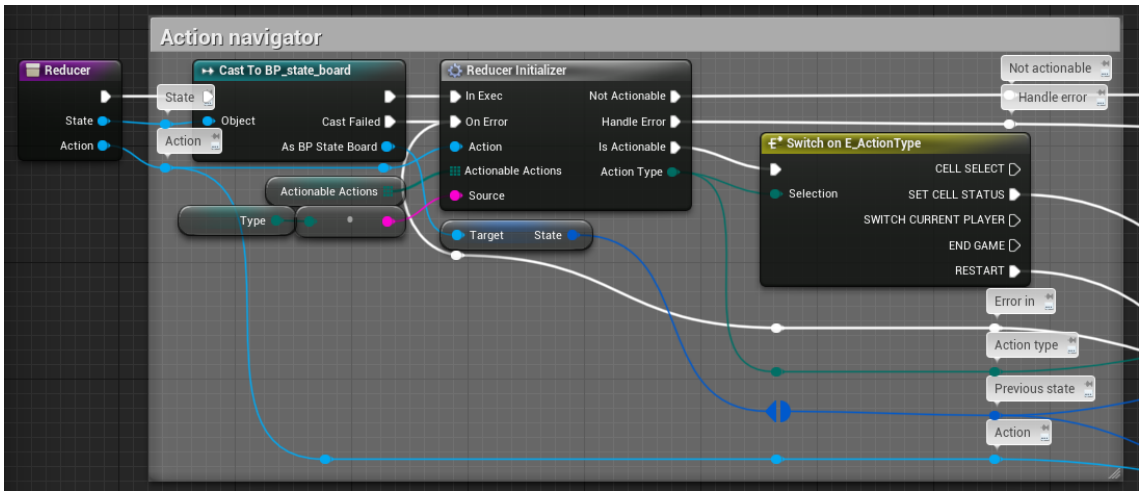

Image 33. Reducer logic for directing different actions.

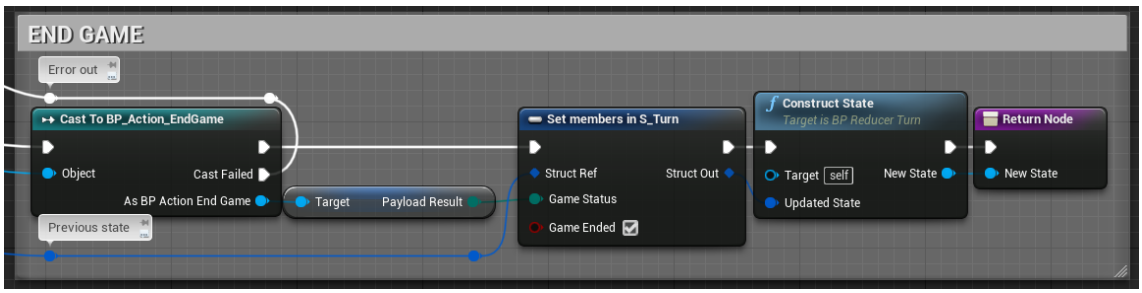

Image 34. Turn reducer logic for handling *end game* -action type.

5.3 Store

The store also requires refining to make it truly scalable. One way to make the logic more scalable would be to pass the whole store state to all of the reducers (image 35). However, this would diverge from the original Redux model where each reducer only receives the part of the store state they are responsible for managing.

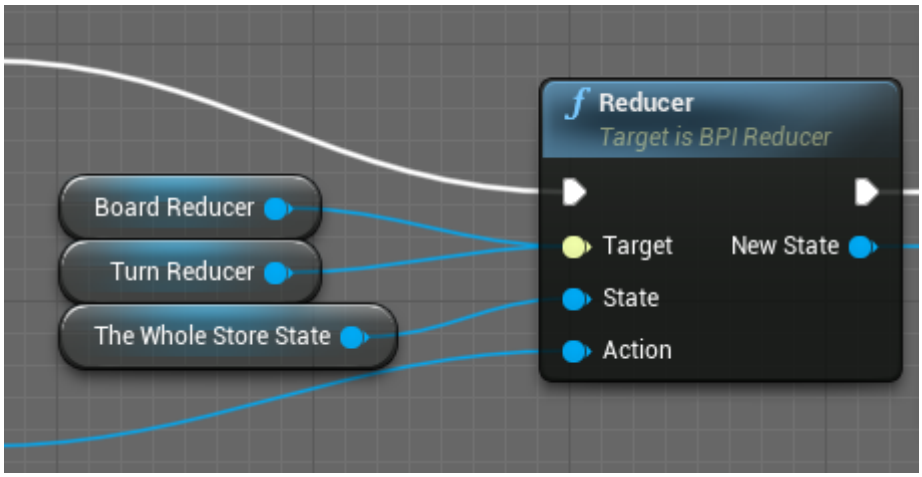

Image 35. Theoretical implementation for reducers receiving the whole store state.

Another way to make the update logic more scalable could be to store the reducers and states in a map instead of a struct. It is possible to get and set map values using keys, which removes the need for select or switch statements (image 36, image 37). A downside to this approach is that the values stored in the map have the generic base type instead of store section specific typing, which means that each value of the map needs to be cast to the correct type when reading the state from it. While this would require some additional logic, it could be isolated to a single utility function. The prospects of using maps instead of structures for storing the store state may seem promising, but unfortunately the author did not have the time to research the viability of this, as it would require rewriting large sections of the core logic for the game.

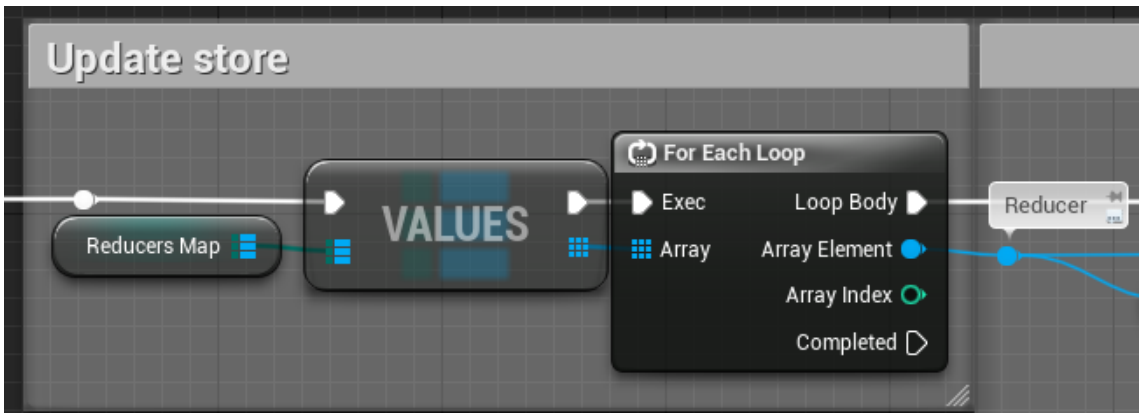

Image 36. Store update using maps, looping over the reducers.

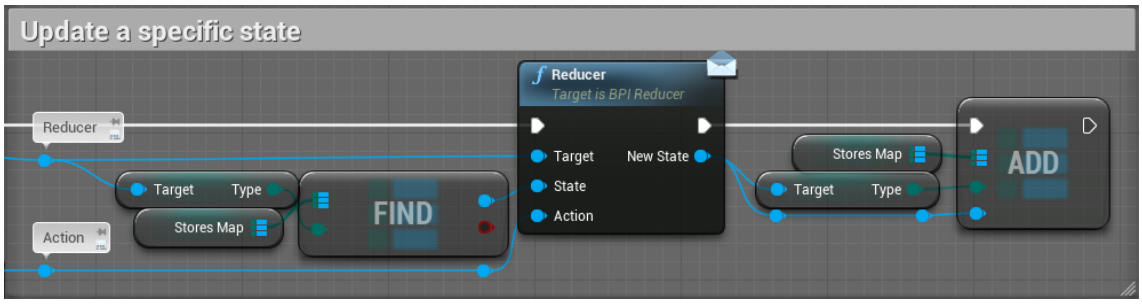

Image 37. Store update using maps, getting new store states, and saving them.

5.4 Sagas

Sagas also require much more refining. Tic-tac-toe is a game where logic is only executed upon user interaction and the execution of said logic is instantaneous, games with real-time elements which possibly interact with each other instead of just the player could require different effects (Redux-Saga 2022c) present in libraries such as Redux-Saga (Redux-Saga 2022a) to be implemented. Using more traditional Unreal Engine programming paradigms, such as action binding (Epic Games 2022q), might be more productive than imitating sagas and effects.

6 Conclusions

Having implemented the same logic in two different environments it was observed how all the different elements are mostly just different shapes built from the same building blocks, such as functions and objects, which can be found in most programming languages. The author looks forward to the opportunity to use the Flux architecture in their future projects, even outside of projects based on JavaScript and Unreal Engine. However, in the end, the Flux

architecture is simply a tool for the job, and better architectures may be available for different situations or environments.

Situations where the Flux architecture is an excellent choice, could be games with checkpoints or save states. Both of them are snapshots of the game, and the Flux architecture already has these snapshots ready for storing in the store state. All that needs to be done is to replace the store state currently in use with a previously saved store state, and the feature is complete.

Another more refined implementation of the same feature could be time travel, as store states could be saved into an array with a timestamp or a turn number, and the player can choose how far back in time they want to travel.

A third variation could be asynchronous multiplayer (Chilli Connect 2022), a concept mostly forgotten in modern games. Modern games prefer synchronous multiplayer thanks to the development of network infrastructure and more powerful hardware enabling real-time interactions. In an asynchronous game, player A sends their Flux state to Player B, and player B interacts with the state and sends it back to player A.

An undo/redo history is a variation of save states and time travel. Because reducers are pure functions with predictable outcomes actions have a pre-defined effect on the state. It could be possible to reverse changes made further out in history without deleting all the changes made after it by sending a reverse action to the reducer (figure 3). Of course, partial or mutable undo history is a feature with its own set of problems, which no doubt has been explored

thoroughly in the past, but the Flux architecture could offer a frame for implementing it.

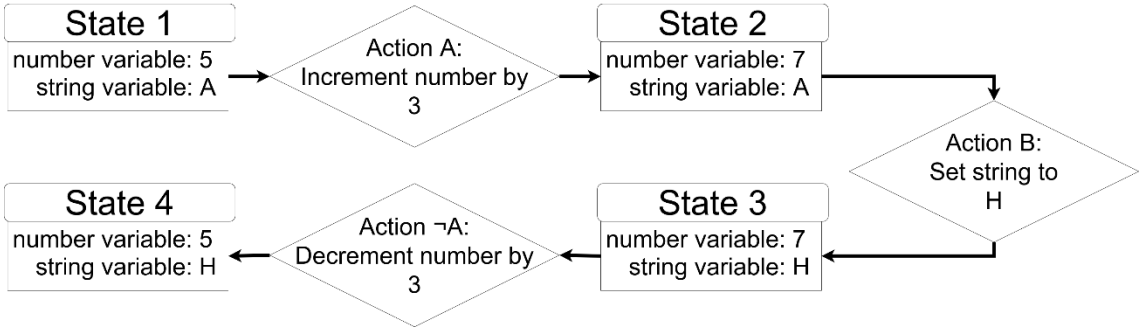

Figure 3. String variable maintains its state after the effect of action A is undone.

Overall, the results of the thesis can be seen as a success. Applying the Flux architecture to Unreal Engine made state management straightforward for the author despite requiring further refinement if implemented in larger and more complex projects. On the other hand, translating the logic of the game into structures mimicking sagas is still lacking, and once again requires a more demanding scenario for encountering and solving a wider array of challenges present in game programming. While Redux sagas are a useful tool for extending the capabilities of an implementation using both Flux architecture and Redux sagas, the sagas were not a central focus for this study. A separate study could be conducted for imitating Redux sagas in Unreal Engine, but it is recommend exploring options native for Unreal Engine (chapter 5.4).

Sources

Asiri, S. 2018. Flux and Redux. https://medium.com/@sidathasiri/flux-and-redux-f6c9560997d7. 2.12.2022.

Abramov, D. 2022a. Redux. https://redux.js.org/. 2.12.2022.

Abramov, D. 2022b. Hooks. https://react-redux.js.org/api/hooks. 28.10.2022.

Chilli Connect. 2022. Asynchronous Multiplayer. https://docs.chilliconnect.com/guide/multiplayer-async/. 29.11.2022.

Cyber Octopus web page. 2000. Tic-Tac-Toe Rules. http://web.cecs.pdx.edu/~bart/cs541-fall2001/homework/tictactoe-rules.html. 2.12.2022.

Doucet, L & Pecorella, A. 2021. Game engines on Steam: The definitive breakdown. https://www.gamedeveloper.com/business/game-engines-on-steam-the-definitive-breakdown. 2.12.2022.

Epic Games. 2022a. Unreal Engine 4 Documentation. https://docs.unrealengine.com/4.27/en-US/. 2.12.2022.

Epic Games. 2022b. Frequently Asked Questions (FAQs). https://www.unrealengine.com/en-US/faq. 15.9.2022.

Epic Games. 2022c. Game Mode and Game State. https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Framework/GameMode/. 2.12.2022.

Epic Games. 2022d. Tools and Editors. https://docs.unrealengine.com/4.27/en-US/Basics/ToolsAndEditors/. 30.10.2022.

Epic Games. 2022e. Templates. https://docs.unrealengine.com/4.27/en-US/Resources/Templates/. 30.10.2022.

Epic Games. 2022f. Blueprint Class. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Types/ClassBlueprint/. 30.10.2022.

Epic Games. 2022g. Widget Components. https://docs.unrealengine.com/4.27/en-US/Basics/Components/Widget/. 30.11.2022.

Epic Games. 2022h. Struct Variables in Blueprints. https://docs.unrealengine.com/4.27/en-

US/ProgrammingAndScripting/Blueprints/UserGuide/Variables/Structs/. 30.10.2022.

Epic Games. 2022i. Blueprint Macro Library. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Types/MacroLibrary/. 30.10.2022.

Epic Games. 2022j. Functions. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Functions/. 2.12.2022.

Epic Games. 2022k. Macros. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Macros/. 2.12.2022.

Epic Games. 2022l. Game Mode and Game State. https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/Framework/GameMode/. 2.12.2022.

Epic Games. 2022m. Construction Script. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/UserConstructionScript/. 30.10.2022.

Epic Games. 2022n. Implementing Blueprint Interfaces. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/Types/Interface/UsingInterfaces/. 30.10.2022.

Epic Games. 2022o. Localization. https://docs.unrealengine.com/4.27/en-US/ProductionPipelines/Localization/. 30.10.2022.

Epic Games. 2022p. UMG UI Designer. https://docs.unrealengine.com/4.27/en-US/InteractiveExperiences/UMG/. 18.11.2022.

Epic Games. 2022q. Binding and Unbinding Events. https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/UserGuide/EventDispatcher/BindingAndUnbinding/. 29.11.2022.

Facebook. 2022. flux. https://github.com/facebook/flux. 2.12.2022.

Hack Hat. 2015. tic-tac-toe-flux. https://github.com/hackhat/tic-tac-toe-flux. 2.12.2022.

Kaskela, S. 2019. Applying real-time user interface practices to games. Turku university of applied sciences. Information and communications technology. Thesis. https://urn.fi/URN:NBN:fi:amk-2019052010619. 2.12.2022.

Krishna, A. 2022. Yarn vs NPM: Which Package Manager is Best to Choose? https://www.knowledgehut.com/blog/web-development/yarn-vs-npm. 2.12.2022.

Marcus, M. 2021. Tic Tac Toe History: Three-in-a-Row Through The Ages. https://www.coolmathgames.com/blog/tic-tac-toe-history-three-in-a-row-thru-the-ages. 2.12.2022.

Meta Platforms. 2022a. React. https://reactjs.org/. 2.12.2022.

Meta Platforms. 2022b. In-Depth Overview. https://facebook.github.io/flux/docs/in-depth-overview/. 2.12.2022.

Microsoft. 2022. Everyday Types. https://www.typescriptlang.org/docs/handbook/2/everyday-types.html#union-types. 26.10.2022.

Mozilla. 2022a. Callback function. https://developer.mozilla.org/en-US/docs/Glossary/Callback_function. 30.10.2022.

Mozilla. 2022b. function*. https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/function*. 8.11.2022.

Negri, T. 2015. Using Facebook's Flux architectural style for game development in Unity 3D. https://tnegri.wordpress.com/2015/03/20/using-facebooks-flux-architectural-style-for-game-development-in-unity-3d/. 2.12.2022.

npm inc. 2022. Build amazing things. https://www.npmjs.com/. 28.10.2022.

OpenJS Foundation. 2022. About Node.js®. https://nodejs.org/en/about/. 28.10.2022.

Ourrad, A. 2017. "How I Fell in Love with Unidirectional Data Flow @ DevConf 2017." DevConf PL. YouTube. https://www.youtube.com/watch?v=Zq1Iz2SjIR0. 2.12.2022.

Penz, B & Mäkiö, J. 2019. Concept And Implementation of a Web Based Tool for The Development of Online Games for Early Childhood Educators. https://www.researchgate.net/publication/334674473_CONCEPT_AND_IMPLEMENTATION_OF_A_WEB_BASED_TOOL_FOR_THE_DEVELOPMENT_OF_ONLINE_GAMES_FOR_EARLY_CHILDHOOD_EDUCATORS. 2.12.2022.

Perforce. 2020. What Are the Best Game Engines? https://www.perforce.com/blog/vcs/most-popular-game-engines. 2.12.2022.

Perlman, L. 2017. Flux architecture in games — Porting the web design pattern to game development. https://medium.com/front-end-weekly/flux-architecture-in-games-porting-the-web-design-pattern-to-game-development-f9d0959346ec. 2.12.2022.

Redux-Saga. 2022a. Redux-Saga. https://redux-saga.js.org/. 29.11.2022.

Redux-Saga. 2022b. Root Saga Patterns. https://redux-saga.js.org/docs/advanced/RootSaga/. 2.12.2022.

Redux-Saga. 2022c. API Reference. https://redux-saga.js.org/docs/api/#effect-creators. 29.11.2022.

Romero, M & Sewell, B. 2019. Blueprints Visual Scripting for Unreal Engine - Second Edition. Packt Publishing. O'Reilly. 2.12.2022.

Unity Technologies. 2022. Choose the plan that is right for you. https://store.unity.com/compare-plans. 2.12.2022.

Vectoreezy. 2022. Tic Tac Toe Vector Art, Icons, and Graphics for Free Download. https://www.vecteezy.com/free-vector/tic-tac-toe. 2.12.2022.

Wadstein, M. 2015a. "WTF Is? The Switch Node". YouTube. https://www.youtube.com/watch?v=2Kcecffh2IY. 2.12.2022

Wadstein, M. 2015b. "WTF Is? Blueprint Function Library in Unreal Engine 4." YouTube. https://www.youtube.com/watch?v=WZsAq05fN_o. 2.12.2022.

Wadstein, M. 2017. "WTF Is? Data Table in Unreal Engine 4 (UE4)." YouTube. https://www.youtube.com/watch?v=nt1hlJO-DPo. 2.12.2022.

Wadstein, M. 2018. "Unreal Engine 4.21 - Composite Tables." YouTube. https://www.youtube.com/watch?v=2tdMAYF4Tu8. 2.12.2022.

Yarn. 2022. Safe, stable, reproducible projects. https://yarnpkg.com/. 28.10.2020.