TypeScript game devlog

Introduction

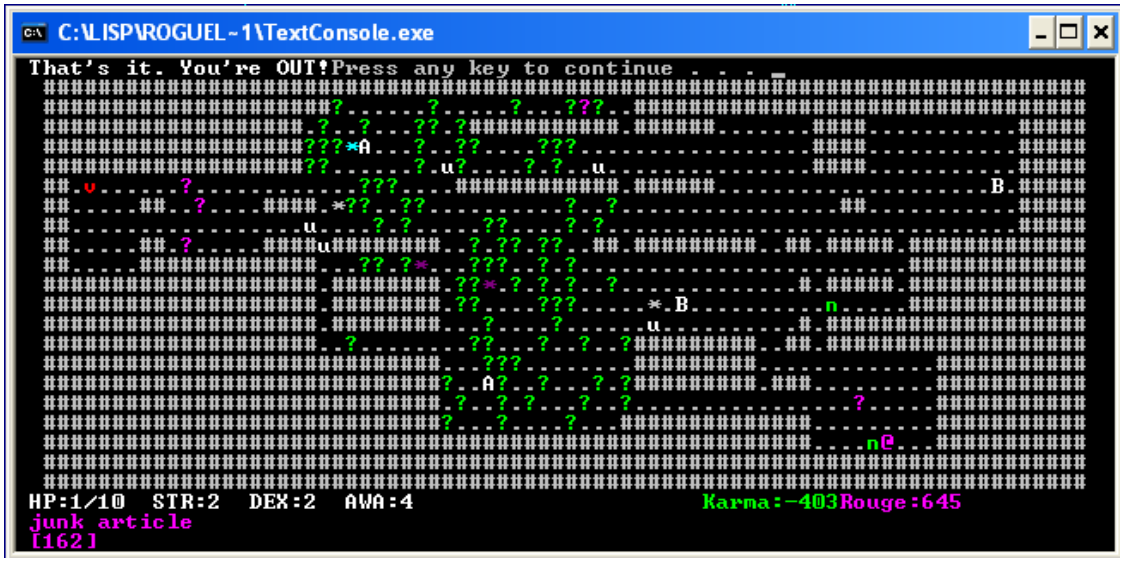
I have some personal experience in game development and in a few languages, such as Python, C++, Haskell, GDscript in the Godot game engine. But not only had I never fully completed a game before, I have never touched TypeScript before, and I have only barely used JavaScript, the language it is heavily based on. My only experience with JS is learning the basics of the language through some tutorials 4 years ago and I have little to no experience with HTML and CSS. So, for this game jam, I decided to write an entire game in TypeScript, which would then get compiled into JavaScript and run in HTML with some light CSS to help with the styling of the game.

TypeScript vs JavaScript

TypeScript and JavaScript are, syntactically speaking, the exact same language but with one very key difference. While JavaScript variables can be any data type and can change between to any other data type, TypeScript has a static type system, which means that a variable can only contain one assigned data type. While this can seem more restrictive, it minimises errors that are caused by a variable being the wrong type since the compiler knows what data type a variables will always be, and I personally prefer this system. In addition, the TypeScript compiler is better at sniffing out certain errors such as when a method name is incorrect (source)

The game itself

The general plan for the game is to create a puzzle game where the player must explore a top down 2D environment in order to track down something that has escaped into the area. The area is non-Euclidean, and part of the puzzles is trying to figure out what combination of doors will open the next area. The game will have an art style similar to Rogue's (pictured below), where the walls are built from hashtags or Unicode box drawing characters and the character is represented by something like a @ symbol.



The player is controlled by the keyboard and then can move using either WASD or the arrow keys (both will work), interact with the enter key or spacebar, and cancel with the escape key. This is the basic plan as of writing this section at the beginning of development. However, as development continues, the different systems and aspects of the game will become clearer, such as who the player is hunting and why, and maybe other systems will be implemented along the way, such as an inventory and item using system.

Before the jam

Before the jam had started, I wanted to make sure I had installed one of the JavaScript game engine frameworks on my computer and that it would actually run. I decided that this would be allowed because I wouldn't really be learning anything by doing this and I wouldn't be actually working on the game. The 3 that I tried to use were JSRL, MelonJS, and PixiJS, and no matter what I did I could not understand how to use them at all. I didn't know how to import them into the JS code file I was working with, what files I had to include in the project folder, or what functions I had to use in my JS code in order to actually use the framework.

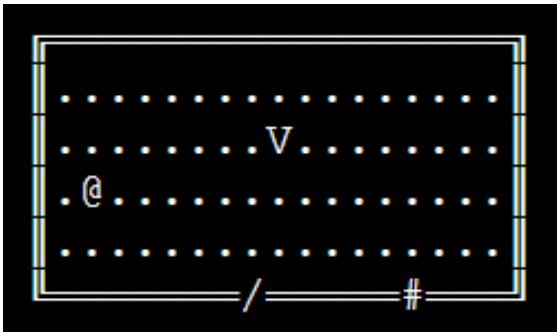
After a while of looking for any documentation of the frameworks, any examples of the frameworks, or any help in general with TypeScript on how to use frameworks in general, I eventually concluded that I just didn't know enough about JavaScript yet, and I felt that trying to learn that much before the jam was cheating so I decided to drop the rendering engines and make the game without a previously developed framework.

I did however want to make sure that I was able to write TypeScript in VS code, compile it to JavaScript code, and that I could run that code in an HTML file with a CSS file that I had written myself to handle the font and the colours of the

game. All of that worked perfectly and the files that I had used during this time are placed in the folder labelled "scrap" in the GitHub repo for this project. The files all have the word "test" at the beginning of their file names.

Beginning of development

The first few days, I started to figure out the fine details of how the game would work and some of the things that I can do in TypeScript. I found out that in HTML you can't create a new line with a simple "\n" and, after some research, I found that the TS code needs to instead use "
". I decided that the game should use the Unicode box drawing characters for the walls. From my experience in C++, I know that sometimes Unicode has some issues displaying in some contexts, like in the terminal for instance. However, this shouldn't be a problem for JavaScript code running in HTML since both HTML and Unicode use UTF-8 encoding.



Pictured: A mock-up of how the game will generally look.

The game's visuals will loosely be made up of:

- The walls of the game will be displayed using Unicode box drawing characters.
- Empty spaces showing where the player can go in a level are full stops,
- The player, who is represented by a V. In the introduction I said that the character used would be something like a @, but I decided to go with a V and to make @ an interactable.
- Interactables, which can be either #, @ or other characters if there need to be more types of interactables later in development.
- Doors, that are represented by slashes.

The basic design of the game's systems

- The player uses the keyboard to interact with the game, so the code will need to sense key presses. In addition, once a single key press is noticed, the game should not recognise another key press until that key is no longer pressed.

- ~ The game will generate the map itself based on . Then there are a set of other arrays that detail what is in that room and where it is, like doors and interactables.
- ~ The doors in the game should be able to send people to rooms and then send them to a different room when they walk back into the door (part of the non-Euclidean design).
- ~ The interactables in the room should provide the player with some kind of information about how to escape to the next area of the game.
- ~ Each area that the player finished will lock behind them when they move onto the next area to avoid the plater getting lost. The game should show a message saying "Need to go further" when the player tries to backtrack.

The HTML and CSS files

The game will run inside an HTML file, which imports JavaScript code that was compiled from the TypeScript file and also uses a CSS file for the styling of the game window. The CSS file dictates that the entire game window will have a black background, the text will be white, and the font will be Courier New (just like this devlog document). Inside the HTML file, the CSS file is linked in the "head" section, and the JavaScript code is called inside the "body" section. This is very basic code for both files, but what it means is that the functionality of the game entirely relies on the JavaScript code, and which simplifies the development process.

```html
<!DOCTYPE html>
<html>
    <head>
        <title> Facility hunt </title>
        <link rel="stylesheet" type="text/css" media="screen" href="gameCSS.css">
    </head>
    <body>
        <script type="text/JavaScript" src = "./gameCode.js"> </script>
    </body>
</html>
```

```css
html {
    background-color: black;
    color: white;
    font-family: 'Courier New', Courier, monospace;
}
```

Pictured: the HTML code (top) and the CSS code (bottom).

Map generation code

The map generation code takes a 2-element array which represents how large the map is, as well as some other arrays here the player and the interactables had to be and what symbol represented the interactables, and then generates a 2D array containing the characters that will be displayed in the game window.

This would require the code to be able to utilize arrays and for loops which I did not previously know how they worked in TypeScript. The first basic iteration of the code took the size of the map and generated the characters, making sure to draw the walls with the appropriate Unicode characters. The code shown below is both the code to create the room and the code to display the room using the created room string array.

```
function create_room(room_size:number[]):string[][] { //This is how TypeScript handles function definitions
    //Takes string array input, outputs 2D string array.
    let output:string[][] = [];

    for (let i=0; i<room_size[1]; i++) { //Loop through every row
        output.push([]); //Add new array for the current line to be added to

        for (let j=0; j<room_size[0]; j++) { //Loop through every unit of that row
            if (i==0) { //If on the top row
                if (j==0)                  { output[i].push('╔'); } //If top left
                else if (j==room_size[0]-1) { output[i].push('╗'); } //If top right
                else                       { output[i].push('='); } //If just on the top row
            }
            else if (i==room_size[1]-1) { //If on the bottom row
                if (j==0)                  { output[i].push('╚'); } //If bottom left
                else if (j==room_size[0]-1) { output[i].push('╝'); } //If bottom right
                else                       { output[i].push('='); } //If just on the bottom row
            }
            else if (j==0 || j==room_size[0]-1) { output[i].push('‖'); } //If on the left or right walls (corners already taken care of)
            else { output[i].push('.'); } //Else, empty space
        }
    }

    return output;
    //return [ ["place","holder"],["out","put"] ]
}

function display_room(room:string[][]):void {
    let writing:string[] = [];
    for (let row of room) { //Iterate over every row using for of
        writing.push(row.join(""));
    }
    document.write(writing.join("<br>"));
}
```

Pictured: the original source code for the create_room function

The next stage was to allow the code to spawn the player and any objects that may exist in the room like doors and interactables. The first step was to edit the input parameters of the create_room function in order for it to accept the player's spawn point and the objects of the room. The player's spawn position should be represented by a 2-element array for the X and Y coordinates, so that when the player can spawn in from the default position or a door entrance if that's how they entered the room.

However, since the room objects had many types of data associated with them such as display character, position, and object type, I needed something similar to a struct to house

all of that data and anymore that the object might need. JavaScript's equivalent of the struct, just without the methods, is the interface, which was perfect for this situation. The interface I created to contain the room objects I appropriately decided to call room_object. The images below show the new input parameters of the create_room function, the room_object interface, and the new code added to the create_room function.

```
function create_room(room_size:number[], objects:room_object[], player_spawn:number[]):string[][] { //This is how TypeScript handles
function definitions
    //Takes string array input, outputs 2D string array.
    let output:string[][] = [];
```

```
interface room_object { //Interfaces work very similarly to structs but without the methods
    chr:string;         //The character that is used
    x:number;           //X coordinate
    y:number;           //Y coordinate
    object_type?:string;
    /* Object types
    ~Door
    ~Text box
    ~Wall
    ~Empty space
    */
    interaction?:string; //If it can be interacted with, this is where the text box lives
}
```

```
    }
    output[player_spawn[1]][player_spawn[0]] = 'V' //Place the player
    for (let thing of objects) { output[thing.y][thing.x] = thing.chr; } //Place the room's objects

    return output;
    //return [ ["place","holder"],["out","put"] ]
}
```

Registering key inputs

[The next step is to register the 4 directions and the interact key. Do not allow for holding the button. Do a test to figure out how the key presses work by showing the code of the key and the time that it was pressed. Testing showed that detecting "keydown" was the best idea because it worked like holding down a key in a text editor. Holding down the movement keys would still allow the character to essentially run, but there was no worry of the player accidently interacting with something twice by tapping the interact key. For some reason, document.write did not use the style as dictated in the CSS file, so the display_room function was refactored to make document.body.innerHTML equal what would be displayed. The code for the process_input function below will have the string outputting replaced with other function calls that are explained in the next section.]

The next step is to register the 4 directions and the interact keys. From this site, I found out that JavaScript supports keyboard detection. When using the "keydown" event listener, the program will detect the initial keyboard press, but only

once for around 0.4 seconds even if the key is being held. After 0.4 seconds, the function will react to the key being pressed as often as possible, making this a great function for detecting key presses in the game. From there I could add my own code to the function that would display what ever key value was being used when I pressed the keys so that I could find out the values for WASD, the arrow keys, enter, and space.

However, when I went to use document.write to display the key value, the style I'd set up in the CSS file was ignored and the background returned to white. I'm not sure why this happened but I found that making document.body.innerHTML equal to what I wanted to display did not have the same problem, so the display room function was refactored to include this.

The next step was to take the key values and read them correctly. I chose to do this with a switch statement since TypeScript has them and switch case is a bit faster than an if statement because switch case will exit when it has found the correct branch were as if else has to go through the other branches. In the code below, you see a

```
let validKeys:string[] = ['w','a','s','d','ArrowUp','ArrowLeft','ArrowDown','ArrowRight','Enter',' ','Escape'];
//This is the function where everything goes from when something happens.
//This runs everytime a key is pressed and will run continously after the initial press if the player holds a key down
//Got code from https://www.codeinwp.com/snippets/detect-what-key-was-pressed-by-the-user/
window.addEventListener('keydown', function (e: KeyboardEvent):void {
    if (validKeys.includes(e.key)) { //If a valid key is pressed
        process_input(e.key);
    } else {} //Else ignore
  }, false);
```

```
function process_input(key:string):void {
    document.body.innerHTML = key;
    switch(key) {
        case 'w':
        case 'ArrowUp':
            document.body.innerHTML = "Going up"; break;
        case 'a':
        case 'ArrowLeft':
            document.body.innerHTML = "Going left"; break;
        case 's':
        case 'ArrowDown':
            document.body.innerHTML = "Going down"; break;
        case 'd':
        case 'ArrowRight':
            document.body.innerHTML = "Going right"; break;
        case 'Enter':
        case ' ': //Represents the spacebar
            document.body.innerHTML = "Interacting"; break;
        case 'Escape':
            document.body.innerHTML = "Cancelling";
    }
}
```

Pictured: the key detection function code (top) and the prototype for the process_input function (bottom)

Something to remember is that later in development, I found no use for the escape key in the game's design, so I remembered

that branch of the switch statement and replaced it with a default branch.

Movement, collision, and interaction

For the movement code, the game looks into the 2D room array and finds what character is in the position that the player is trying to go to. If it's empty then the player will change to that position, but if there's something there then the game checks what exactly is there. If it's a door, then the door is interacted with, and if not, then the player doesn't move because they are blocked.

For the interaction code the idea was the same except having to check through 4 places instead of just one. Then checking through the different things that can be interacted with and acting accordingly. This was where I found that, that the buttons must be searched through the room_objects array and not found using room since the color part was caused by an HTML command. In addition, changing the color of the button wasn't going to work unless the color was only applied in the display_room function, so the function was refactored.

A few extra things

The game needed an end screen and a title screen, so I created the functions for the 2 of them and then refactored the event listener code so that events would be read different depending on which one was active. The title screen could only be exited by pressing enter, and the end screen cannot be exited out to add to the spookiness.

Conclusion

I found using TypeScript to be much more understandable than I thought it would be since a lot of the syntax is very similar to languages I already understand like C++. In addition, I also understood a little more about how JavaScript interacts with HTML and how the TypeScript compiler works since I had to use a tsconfig.json file for the compiled code to include the array.includes function. As well as that, creating my own primitive game engine was changed the way I think about how games are displayed. For things like Pygame in Python for example, the game window updates every single frame, or longer if the code in the loop takes longer than expected. But in this game, the window will only update on a key press, meaning that the game's mechanics, features, and even the code that I write had to be different.