



**Vyšší odborná škola
a Střední průmyslová škola elektrotechnická
Plzeň, Koterovská 85**

DLOUHODOBÁ MATURITNÍ PRÁCE S OBHAJOBOU

Téma: DnD Interaktivní Příběh

**Autor práce: Patrik Holub
Třída: 4. M
Vedoucí práce: Jiří Švihla
Dne: 31. 3. 2023**

Hodnocení:



Vyšší odborná škola
a Střední průmyslová škola elektrotechnická
Plzeň, Koterovská 85

Zadání dlouhodobé maturitní práce

Žák: Patrik Holub
Třída: 4.M
Studijní obor: 78-42-M/01 Technické lyceum
Zaměření: Kybernetika

Školní rok: 2022 - 2023

Téma práce: DnD Interaktivní Příběh

Pokyny k obsahu a rozsahu práce:

Vytvoření funkční komunikace (Říjen 2022)

Vytvoření příběhu (Prosinec 2022)

Kompletace fyzické části (Únor 2023)

Vytvoření dokumentace (Březen 2023)

Termín odevzdání: 31. března 2023

Čas obhajoby: 15 minut

Vedoucí práce: Jiří Švihla

Projednáno v **katedře ODP** a schváleno ředitelem školy.

V Plzni dne: 30. září 2022

Mgr. Jan Syřínek, v.r.
ředitel školy

Anotace

V této maturitní práci se snažím o vytvoření programu který je schopný přednést příběh a předložit hráči možnosti na které příběh bude reagovat. Tímto se docílí interaktivita příběhu. Zároveň vše je děláno systémem skriptů, tudíž je možno hru, při dodržení pravidel pro psaní daných skriptů, měnit pomocí uživatelsky napsaných programů. Projekt také obsahuje metodu se kterou se dá zjednodušeně příběh napsat a použít jednotlivé skripty pro interaktivitu. Poslední část práce je uložení hráče a jeho odpovídajících skriptů na raspberry pi pico a čtení předem zmíněného obsahu ze zařízení. Celá práce je dělaná v programovacím jazyku Micropython.

„Prohlašuji, že jsem tuto práci vypracoval samostatně a použil(a) literárních pramenů a informací, které cituji a uvádím v seznamu použité literatury a zdrojů informací.“ „Souhlasím s využitím mé práce učiteli VOŠ a SPŠE Plzeň k výuce.“

V Plzni dne: Podpis:

Obsah

1	Fyzický klíč	5
1.1	Komunikace	5
1.2	Datová struktura	6
2	Struktura adresářů a souborů	6
3	Hlavní program	7
3.1	Handler	7
3.1.1	Paměť programu	8
3.1.2	Struktura Scriptu	9
3.2	Okno Hry	9
3.2.1	Logické provedení	10
3.2.2	Formátování	11
3.2.3	Vytváření a práce s oknem	11
3.2.4	Input	12
3.3	Eventy	12
4	Skriptovací nástroj	13
4.1	Handler	13

1 Fyzický klíč

Pro komunikaci jsem vytvořil design vlastní desky, založené na čipu RP2040. Tato deska je schopná zobrazit informace ohledně hráče, pokud daná informace má min a max.

Zobrazování je provedeno pomocí 8 LED na každé straně, které jsou ovládaný přes 8 bitový posuvný registr v časovém multiplexu pro ovládání dvou stran zároveň. Tento časový multiplex běží rychlostí 2kHz aby každá strana běžela na rychlosti 1kHz. Zvolená rychlost byla ze dvou důvodů. První byl, že při takové rychlosti by oko nemělo být schopné zaznamenat jak strany blikají. Druhý důvod souvisí s frekvencí na které často běží PWM. Díky těmto vlastnostem jsem schopný ovládat LED buď aby jenom blikali, nebo abych ovládal jejich jas pomocí simulovaného PWM.

1.1 Komunikace

Máme dvě komunikační zařízení a potřebujeme vytvořit komunikaci mezi nimi. Komunikace je řešena přes comports. Hostitel, v tomto případě počítač, na kterém běží hlavní program, si nechá pomocí nástrojů knihovny serial vypsát všechny rozpoznané comports. Následně se pokusí otevřít comport a poslat mu příkaz. Pokud se mu nepovede otevřít comport, víme že je to nechtěné zařízení. Když se otevření povede, pošle mu příkaz "type". Pokud je na druhé straně správné zařízení neboli pico se správným nahráním programem, tak zpátky odešle zprávu "player". Jenom tehdy je zařízení dáno do listů zařízení, se kterým potom program pracuje dál.

Hostitelský program je o dost složitější než ten na picu. Hostitel musí zprávu nejdřív převést do kódování utf-8 neboli bytes. To je posláno na sériovou linku comportu, který je zrovna otevřený. Pico má práci velmi jednodušší, neboť micropython je schopný číst sériovou linku pomocí příkazu `input()`. Následně pico kontroluje jakou zprávu dostal vůči svému slovníku. Pokud zpráva odpovídá nějakému příkazu, tak pico navrátí odpovídající hodnoty, jinak jenom pošle zpět že zprávu zaregistroval. Hostitel nyní musí projít zpětným procesem převádění zpět z bytes na string. Pokud pico pošle víceřádkovou zprávu, tak samozřejmě musí komunikační program převést celé pole, které přečte z comportu.

1.2 Datová struktura

Díky způsobu práce s daty na klíči (nepracuje s nimi přímo uživatel, vše je ovládáno pomocí programu) tak zde může existovat velmi neflexibilní struktura souborů. V rootu musí existovat `main.py`, který se sám spustí, když se pico zapojí. Zároveň musí být v rootu samotný soubor hráče a jakékoli knihovny, které `main.py` potřebuje. Následně všechny skripty, které jsou specifické vůči hráči, musí být v adresáři `player_scripts`.

2 Struktura adresářů a souborů

Během tvorby programu byl kladen důraz na co nejméně vynucené struktury ze strany uživatele. Hodně problémů jsem tudíž vyřešil pomocí tvorby souborů se specifickými názvy, pro které se program dívá a vytvoří si strukturu cest, se kterou pracuje a podává do dalších funkcí. Je zde ale určitá kultura, kterou bych doporučil při tvoření souborů.

Hlavní adresář příběhu

- Soubor s názvem `_story.txt`
- Adresář s inicializačními skripty
- Adresář se skripty eventů
- Adresář s hráči (může být prázdný, pokud je použit klíč)
- Adresář se skripty hráčů
- `player_scripts`
- další adresáře, které už tvoří vlastní strukturu příběhu

Všechny adresáře, které obsahují skripty, musí obsahovat soubor `_scripts.txt` a `info.json`. První soubor je hledán aby program věděl kde má skripty uložené a druhý soubor obsahuje informace pro program jak se jednotlivé skripty chovají.

Jednotlivé části příběhu, se kterými chceme aby program pracoval, jsou určeny soubory `storypart.json`. Tento soubor obsahuje jednu informaci pro program, "include" určující ob-

sáhnutí při prvotním načtení, a dále jakékoli další informace, se kterými chce následně uživatel pracovat ve svých skriptech.

Jedno místo kde program nehledá pro *storypart.json* jsou soubory, které začínají tečkou. Jedno fixní pravidlo s mou verzí programu je, že u každého souboru *storypart.json* musí být *.orig* adresář. V tomto adresáři musí být původní verze mateřského adresáře. Logické skripty přepisují soubory destruktivně (bez možnosti vrácení změn), proto zde adresáře *.orig* musí být. I kdyby uživatel vytvořil příběh bez logických skriptů, tak po splnění všech eventů v rámci jedné příběhové části přepíše klíč "include" v *storypart.json* na "false". Toto je moje metoda jak vytvořit pseudo uložení hry. Má to ale stejný problém jako logické skripty, jedná se o destruktivní změny.

3 Hlavní program

Vše je psáno pomocí programovacího jazyka Python, přičemž velmi využívám datového souboru typu json pro ukládání rozmanitých dat.

Hlavní program nastaví všechny iniciální hodnoty, provede inicializační sekvenci, najde místo příběhu a najde všechny eventy, které se budou muset stát. Všechny eventy mají své odpovídající skripty a hlavní program samozřejmě provede jejich import. Zároveň vytvoří list importovaných funkcí aby s nimi metody a funkce mohli pracovat.

Jako poslední, hlavní program samozřejmě obsahuje smyčky pro průběh programu.

3.1 Handler

Nejdůležitější součást programu je handler. Tato funkce je schopná vzít script a odpovídající parametry, dosadit hodnoty z paměti, funkci zavolat a uložit její návratovou hodnotu.

Handler jako funkce má dva svoje parametry. První parametr musí být název skriptu bez koncovky *.py* (takto jsou uloženy v naimportovaných skriptech). Druhý parametr je pole argumentů, které chceme do funkce poslat. Handler používá vybalovací znaménko *** aby první dimenze pole byla předána funkci jako jednotlivé parametry. Zároveň se v této první

dimenzi hledá & znak, který jsem zvolil jako identifikátor že je třeba vzít hodnoty z paměti. Pokud nějakou najde hodnotu v paměti, nahradí & znak odpovídající hodnotou.

Dalším krokem je zavoláním funkce. Hlavní program obsahuje list importovaných funkcí, a protože Handler žije uvnitř hlavního programu, tak může sáhnout do tohoto listu a funkci zavolat. Volání funkce je provedeno přes funkci `getattr()`.

Zpracování návratových hodnot vyžaduje správné zapsání informací v *info.json* souborech, které si program načte do paměti. Tyto soubory mají strukturu slovníku a obsahují dvě sekce. První sekcí, jménem `id`, je soupis názvů skriptů a jejich `id`. Druhá sekce, jménem `info`, je určená jak pro důležité programové informace tak pro uživatelské. Dva programové listy které se zde nacházejí jsou pod klíči `"func"` a `"obj"`. Každá funkce se musí nacházet v jedné z nich a ne v obou zároveň. Handler zjišťuje jestli funkce vrátila hodnotu, pokud ano tak ji ukládá do paměti. Pokud ale skript neobsahuje funkci, obsahuje objekt, tak handler zavolal konstruktor a musí se tudíž celý objekt uložit do paměti.

Mezi další, již nepovinné, informace patří například listy pod klíči `"lock"` a `"logic"`. První pole říká handlerovi aby hodnotu, kterou načtl z paměti nezahazoval a ponechal ji v paměti. Jsou to tudíž `"zamčené"` hodnoty. Tato možnost by měla být použita pouze pro skripty u kterých víme že nebudou volány více než jednou. Nebo pokud víme že následně v eventech budou hodnoty násilně požrány.

Druhý list určuje takzvané logické skripty. Pokud je jejich návratová hodnota typu slovník, mají schopnost přidávat eventy a tudíž jsou logické skripty schopné vytvářet smyčky. Kromě návratových hodnot typu slovník handler také u rozpoznává stringy s koncovkou `.json`. Pokud je takový string nalezen, tak Handler z odpovídajícího souboru vezme eventy a přidá je. Handler je ještě schopný nalézt jeden identifikátor, `"self"`. Pokud je `"self"` nalezen a jeho hodnota je `true`, tak handler přidá, celý původní script, který byl volán. Toto umožňuje snadno vytvořit nekonečnou smyčku.

3.1.1 Paměť programu

Struktura paměti je takzvaný stack. Pro tento účel jsem vytvořil třídu, abych si mohl dodat manipulační metody, které bych mohl potřebovat. Samotný stack má strukturu listu (toto

jsem zvolil abych mohl použít indexované hledání, mazání, či přidávání hodnot), kdy hodnotou listu je vždy list o dvou hodnotách. Na první pozici stojí vlastní hodnota a na druhé stojí id, neboli adresa, funkce či objektu, co tuhle hodnotu vytvořila. Pokud funkce chce pracovat s hodnotami jiných funkcí, tak je použita první nalezená hodnota "shora"(Pokud bychom se dívali na hodnoty od indexu 0 až do X, tak by to byla první hodnota nalezená od nuly).

Pro manipulaci hodnot jsem vytvořil také dva skripty, které může uživatel používat a to je putVal, pushVal a eatVal. PutVal jednoduše přidá hodnotu do stacku pod svojí vlastní ID. PushVal je schopný posunout hodnotu na další místo ve stacku za nejbližší hodnotu se stejnou adresou. EatVal požere hodnotu ze stacku i za možnosti že je to "uzamčená" hodnota.

Jak jsem již zmiňoval v sekci Handler, pro čtená z paměti jsem vybral znak &. Inspiroval jsem se z programovacího jazyka C kde & operátor vrací adresu proměnné. V mém případě je za znakem & adresa kde se hodnota má nacházet v paměti.

3.1.2 Struktura Scriptu

Kvůli použití funkce getattr() ve funkci Handler, a zjednodušení programu, má script jedno nutné pravidlo, které se musí dodržovat. Script vždy musí obsahovat stejnojmennou funkci jako soubor, ve kterém je obsáhlá. Na první pohled se toto zdá jako divné pravidlo a vytvoří to velmi elegantní zápis:

```
volana_funkce = getattr(importy[funkce], funkce)
```

3.2 Okno Hry

Zobrazování hry jsem se rozhodl použít příkazový řádek. Avšak oproti normálnímu příkazovému řádku, je tento schopný dodržet šířku a výšku okna, udanou v počtu vykreslených znaků. V rámci těchto limitů respektuje slova, tudíž se nikdy nestane, že by bylo slovo odříznuto, nebo vykreslena neúplná věta.

Další funkčnost, kterou toto provedení okna obsahuje je pomalý a rychlý výpis. Celé okno je po každém kliknutí enter překresleno. Informace, které již na obrazovce byly, jsou vy-

kresleny rychle (po slovech) a informace, které jsou nové, jsou vykresleny pomalu (znak po znaku). Konečný efekt, pokud je počítač dostatečně rychlý, vypadá jako kdyby se pouze poslední řádka překreslovala.

Zde bych rád zmínil, pokud bych další pokračování mé práce zaměřil čistě na grafickou složku, tak bych byl schopný odstranit momentální problém s cenou na výkonu. V této práci jsem toho nebyl schopný, protože s mými znalostmi, v rámci této práce, bych při vytvoření takového okna byl nucen použít velký počet proměnných, které by mi dávali informace o momentálním stavu okna. Řádná optimalizace by vyžadovala mnohem složitější funkci a nebo bych přešel na systém vytvoření okna, které by simulovalo chování příkazového řádku. V případě nového okna by optimalizace byla zajištěna grafickým enginem, který bych používal pouze k vykreslování textu.

3.2.1 Logické provedení

První co se musí provést, z kterékoli části programu, je metoda `add_text`. Ta je schopná přijmout dva možné vstupy. Buď přes list, pole, nebo jiné iterovatelné objekty, nebo přes string. Pokud je přijat iterovatelný objekt, tak je každá hodnota braná jako string a přidána do paměti okna, pokud je přijat string tak je přidán do paměti rovnou. Než ale se cokoliv přidá, tak projde kontrolou, jestli je možno takovou větu vykreslit. Pokud řádka projde skrz kontroly, tak je přičtena jednička k proměnné, která řídí postupné vykreslování.

Ve skriptech jsem vytvořil ještě jeden, rozšiřující, skript. Je stejnojmenný jako metoda okna pro přidání textu, ale je schopný vzít txt soubory. Skripty mají přístup k adrese, pokud si ji vyžádají, kde momentálně mají pracovat. Při použití tohoto skriptu je důležité brát na vědomí, že skript pro textové soubory hledá v adresáři *texts*, v pracovní cestě.

Vykreslování samotné je tvořené přes dvě metody. Jedna metoda jenom připravuje pole, které vykreslovací metoda dá do okna. Vykreslovací metoda neobsahuje žádnou logiku jestli může nebo nemůže vykreslit něco na obrazovku, vše zařizuje formátovací metoda. Formátovací metoda pomalu postupuje skrz paměťový list a prochází skrz něj odshora dolů.

3.2.2 Formátování

Jak jsem již zmínil tak formátování je schopné dodržovat velikost okna. Kromě toho ale umí pracovat i se speciálními znaky. Momentálně implementované znaky jsou `<h>`, `<un>` a `<r>`. Syntax těchto znaků je převzatý z tagů v HTML. `<h>` a `<un>` jsou znaky, které jsou "hokrovací", což zde znamená že jsou schopné připnout řádku k vrchu formátovaného okna. Připnuté řádky budou připnuté tak dlouho dokud se přímo pod nimi neobjeví řádka s `<un>` znakem. Takto se dá připnout kolik řádek je potřeba, ale program nepustí text do paměti, pokud by připnuté řádky zaplnili okno úplně. Poslední znak `<r>` je určený pro započetí čtení z příkazového řádku.

Řádky v paměti nejsou rozřezány podle šířky okna takže se musí se zalomením textu počítat při formátování. Jako správné chování programu jsem uznal, že pokud je textu enter, tak ho to vykreslí. Samozřejmě při přidávání textu do paměti se kontroluje aby vůbec řádka bylo možno ji celou dostat na obrazovku. Formátování slov na kraji obrazovky jsem vyřešil tak že jsou posunuty na další řádek.

3.2.3 Vytváření a práce s oknem

Okno se vytvoří přes event, kdy parametry eventu jsou šířka a výška okna. Jak jsem již zmiňoval v Handlerovi tak okno je uloženo do paměti, jako objekt, protože Handler zavolá konstruktor. U okna silně doporučuji hodnotu dát mezi "uzamčené" hodnoty, aby po každém podání okna eventu, nebylo zničeno.

Jak už jsem zmínil v sekci logické provedení, tak pro přidání textu do okna se dá použít buď přímo metoda objektu, nebo stejnojmenný skript. Pokud je skriptu povolen přístup k objektu, tak je teoreticky schopný ovládat rendrování do okna přímo. Jedna z hlavních věcí co ale okno umí je zapnout automatický render. Opět se dá spustit přes metodu nebo přes skript. Objekt má metodu jak pro zapnutí tak pro vypnutí, ale skript přepíná mezi zapnutým auto renderem a vypnutým.

Auto render má ještě jednu hlavní vlastnost a to že spustí vykreslování okna na jiném vlákně než hlavní smyčka. Umožňuje to přidávat text a procházet neblokující eventy i když

uživatel nic nedělá, ale vytvořilo to lehké potíže při tvoření čtení dat.

3.2.4 Input

Jak již bylo zmíněno, tak okno je schopno rozpoznat tag `<r>` a započne čtení. Teoreticky by toto čtení neměl vůbec být problém, ale kvůli použití vláknění to nebylo triviální. Na první pohled by se zdálo že jsem měl použít knihovnu *Asyncio* ale našel jsem, podle mě, lepší řešení. Knihovna *Threading* je schopná tvořit asynchroní input a output, a to díky tomu že sdílí primitivní typ *Future* (V *Threading* se jmenuje *Event*). Tento typ funguje tak že po jeho nastavení jsem schopný z různých míst programů "sepnout" tento typ nebo ho "vypnout". Následně je ve funkci možno nastavit aby funkce asynchroně čekala na "sepnutí".

Tuto teorii jsem zkonstruoval na `got_input`, což je *Future*, a metoda `get_return_value()`, která získává hodnotu z příkazového řádku. Pokud kterýkoliv skript zavolá metodu pro získání hodnoty, tak v ten moment začne čekat dokud uživatel něco nezadá. Okno ale bude čekat než se dostane na řádek s tagem `<r>`.

Kvůli mému provedení nedoporučuji dávat čtecí tag přímo do textu. Vytvořil jsem skript *ask*, který je schopný nejen získat input, ale je zároveň schopný poslat otázku do okna, přidat k ní možnosti, pokud je uživatel zadá, a dodat nutný čtecí tag. Možnosti, které může uživatel poskytnout eventu *ask*, jsou typu slovník (dict). Klíče jsou nahrány do okna. Hodnoty, potom co si uživatel vybere možnost, jsou vráceny do paměti programu. Tudíž může uživatel pracovat s daty, které se nebudou uživateli zobrazovat. Pokud uživatel zadal hodnotu, na kterou se nikdo neptal, tak *ask* vypíše do okna, že uživatel zadal neplatnou možnost, a zopakuje otázku.

3.3 Eventy

Během předchozích textů jsem zaměňoval slova event a skript. Event je přepis pro hlavní program aby byl schopný pustit skript. Skript je podprogram, který hlavní program používá pro své akce. Přepis jednotlivých eventů je v souboru *event.json*, který se musí nacházet na úrovni *storypart.json*.

Samotný zápis je syntax json slovníku. Klíčem je název skriptu, který chce uživatel použít, plus `_X`, kde `X` je pořadí eventu v momentálním souboru. Část `_X` je zde jenom kvůli tomu, že klíče nemohou být stejné. Hodnotou ke klíči je vždy pole argumentů, které Handler podá skriptu. Zde se píší již zmiňované znaky `&` aby byla umožněna komunikace mezi skripty.

4 Skriptovací nástroj

Když si shrneme co by vše uživatel musel umět aby si příběh napsal, tak by to znamenalo, že je nejspíš programátor a může si tudíž klidně naprogramovat vlastní hru. Tudíž jsem se rozhodl udělat nástroj, který by byl schopný pomoci uživateli a dát mu inteligentní předpovídání při psaní eventů a jiných souborů. Stejně jako hlavní program, tak skriptovací nástroj obsahuje vlastní verzi Handlera, díky které se opět stává kompletně rozšiřitelný. Bohužel ale není naprosto kompletní, nepatří mezi hlavní části mé práce, ale i tak mi přišlo ho do určité míry připravit.

Nástroj je opět v příkazovém řádku ale oproti Hlavnímu programu je jeho výpisová schopnost vyšší. Má sice jeden zásadní problém ale k němu se vyjádřím později.

4.1 Handler

Podobně jako Handler v hlavním programu, je tento Handler schopný volat skripty a podávat jim argumenty. Má ale mnohé změny. První nastanme rovnou přímo při importování programu. Skriptovací nástroj má fixní strukturu souborů. Což v tomto případě znamená, že všechny skripty musí existovat na úrovni *main.py*.

První změna je celkem regresivní, další byly podle mě vylepšení oproti Handlerovi hlavního programu. V každém skriptu musí existovat funkce `get_attr()`, která vrací informace pro Handler. Návrátová hodnota je slovník, který v klíčích obsahuje názvy funkcí, které skript obsahuje. Jako hodnoty jsou zde nápisy, které by měl nástroj předložit uživateli jako možnosti.