FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Editor behaviorálních stromů
Student:	Michael Moyal
Vedoucí:	Ing. Jan Matoušek
Studijní program:	Informatika
Obor / specializace:	Počítačová grafika 2021
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Behaviorální stromy se stále více využívají k popisu chování počítačem ovládaných postav. Cílem této práce je prozkoumat nové možnosti a vytvořit alternativu k nástrojům používaným v herním průmyslu.

Pokyny k vypracování:

1) Proveďte rešerši problematiky behaviorálních stromů a existujících nástrojů k jejich editaci a ladění.
2) Analyzujte požadavky kladené uživateli na tyto nástroje.
3) Na základě analýzy navrhněte architekturu aplikace a její uživatelské rozhraní.
4) Implementujte aplikaci dle návrhu.
5) Aplikaci podrobte vhodnému testování.
6) Získané poznatky shrňte a navrhněte další možnosti vývoje.

Elektronicky schválil/a Ing. Radek Richtr, Ph.D. dne 17. února 2024 v Praze.

Bakalářská práce

EDITOR BEHAVIORÁLNÍCH STROMŮ

Michael Moyal

Fakulta informačních technologií
Katedra softwarového inženýrství
Vedoucí: Ing. Jan Matoušek
16. května 2024

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Na prvním místě bych chtěl poděkovat především svému vedoucímu práce, Ing. Janovi Matouškovi, za trpělivost a vedení této práce. Bez jeho pokynů a rad by tato práce vznikala jen těžko. Dále bych chtěl poděkovat rodině, která mě při studiu i tvorbě tohoto díla podporovala při každém kroku a umožnila mi plně se soustředit na studium po většinu své akademické kariéry.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 16. května 2024

Abstrakt

Tato bakalářská práce řeší potřebu moderního uživatelského rozhraní pro nástroje určené k vyvíjení behaviorálních stromů. Nejprve je provedena analýza existujících řešení spolu s kratkým úvodem do problematiky behaviorálních stromů. V rámci této práce je vyvíjena nová desktopová aplikace, která je navržená dle požadavků vývojářů z průmyslu, kteří s těmito nástroji denně pracují. Výsledkem práce je prototyp nového uživatelského rozhraní implementující některé vyžádané vlastnosti. Tento prototyp pak byl na těchto uživatelích otestován a poznatky z testování shrnuty. Další možnosti vývoje jsou pak také navrženy.

Klíčová slova Uživatelské rozhraní, Umělá inteligence, Behaviorální strom, Editor, Electron, TypeScript, Vue.js

Abstract

This bachelor thesis addresses the need for a behavior tree development tool with a modern user interface. Firstly there is an analysis of existing solutions paired with a short introduction to the field of behavior trees. The main focus of this thesis is the design and development of a new desktop app designed according to the requirements of the developers than use these tools everyday. The result is a prototype of the potential new user interface implementing some of the developers' requirements. This prototype is then user tested with the previously mentioned developers and the findings are summarized. Further development options are then also proposed.

Keywords User interface, Artificial Inteligence, Behavior Tree, Editor, Electron, TypeScript, Vue.js

Seznam zkratek

BISim	Bohemia Interactive Simulations (firma, kde jsem zaměstnán)
NPC	Počítačem ovládaná postava (Non-player character)
HFSM	Hierarchický konečný stavový automat (Hierarchical finite state machine)
DAG	Orientovaný acyklický graf (Directed acyclic graph)
BT	Behaviorální strom (Behavior Tree)
GUI	Grafické uživatelské rozhraní (Graphical user interface)
UI	Uživatelské rozhraní (User interface)
BB	Blackboard
OS	Operační systém
API	Programovatelné rozhraní aplikace (Application programming interface)
SDK	Sada pro vývoj softwaru (Software development kit)
XML	Extensible markup language
JSON	JavaScript object notation
UML	Unified modeling language
DOM	Document object model

Úvod

V posledních letech se herní průmysl čím dál více přiklání k multiplayerovým zážitkům. Nejsou to už ale hry, které si s kamarádem zahrajete u jednoho displeje. Většina počítačových her dnešní doby se soutředí výhradně na hry společně s, nebo proti jiným hráčům, kteří mohou být tisíce kilometrů daleko. Ať už jde o hry žánru battle-royale, jako je například Fortnite nebo Call of Duty: Warzone 2, hry na styl MOBA – Multiplayer online battle arena – jako jsou League of Legends nebo Dota 2, až po oddechové hry jako například Minecraft, Roblox a mnoho dalších, může se zdát, že chování většiny charakterů v těchto herních světech stojí na jejich hráčích. Opak je ale pravdou. Většina zdánlivého života, především v single-playerových hrách, je ironicky ovládána počítačem. Postavami, se kterými hráč po dobu hraní interaguje a jež hráče vedou skrz dějovou linii – někdy nazývanými zkráceně jako NPC nebo-li Non-player character. Od nepřátel, se kterými musí hlavní postava bojovat a kteří dost často bývají zdrojem surovin a „lootu“ (vybavení pro vaši postavu), po faunu, která dodává hernímu světu pohyb a nádech života. Všechny tyto bytosti a jejich chování ovládá sofistikovaný systém instrukcí, které zajišťují větší ponoření do virtuálního světa.

Za počátek umělé inteligence v počítačových hrách se dá považovat počátek her jako takových. Jedna z úplně nejprvnějších her – Pong – měla možnost hrát v módu singleplayer, kde hráč ovládal jedno padlo a druhé ovládal počítač. Zda-li se tomuto dalo říkat umělá inteligence je možná k diskuzi, co se ale nemění je fakt, že počítač vykonával nějakou sadu vyhodnocovacích instrukcí, aby předešel proletění míčku za okraj obrazovky a ztrátě bodu. Dalším krokem, který určitě stojí za zmínku, jsou duchové ve hře Pac-Man. V této hře byl úkol hráče posbírat všechny kuličky na herní mapě. Tento úkol mu ztěžovali počítačem ovládaní duchové, kteří hledali cestu směrem k hráči a pronásledovali jej, někdy mu dokonce odstřihovali cestu.

Dnešní umělá inteligence v počítačových hrách je mnohem sofistikovanější. NPC postavy umí reagovat na akce hráče, předpovídat jeho chování a dynamicky se podle těchto vjemů a okolního světa rozhodovat. Se zvýšenou fidelitou a komplexitou rozhodovacích algoritmů stoupá realističnost postav ve světě, ale také zároveň imerze do herního světa jako takového, což je jednou z hlavních cílů herních vývojářů. Nejpopulárnějším nástrojem, po kterém herní vývojáři v dnešní době sahají, jsou behavior trees nebo-li behaviorální stromy. Vyvíjení pomocí behaviorálních stromů je druh vizuálního programování a s tím spojený je nástroj, ve kterém se tyto stromy vyvíjí. Tyto stromy používáme také u nás v Bohemia Interactive Simulations (dále BISim). Jelikož se na vývoji těchto stromů v práci podílím, nastřádal jsem si nějaké nápady a požadavky, které bych jako uživatel rád v takovýchto nástrojích uvítal. Když se v práci naskytla potřeba právě tento nástroj modernizovat a přenést do 3. desetiletí 21. století, neváhal jsem a příležitosti jsem se chytil. To je koneckonců jednou z motivací této bakalářské práce.

Cíle

Cílem práce bude vytvoření nové, moderní aplikace pro vývoj a ladění behaviorálních stromů. Aplikace vznikne na základě analýzy stávajících nástrojů a požadavků na aplikaci kladených uživateli. Pomocí aplikace bude prozkoumáno a otestováno několik nových prvků uživatelského rozhraní, které mají vývojářům ulehčit práci na stromech. Důraz je kladen na funkcionality, které v jiných stávajících řešeních neexistují nebo existují ve stavu, který by se dal vylepšit. Aplikace bude otestována na uživatelích, kteří behaviorální stromy vyvíjí denně jako součástí náplně své práce a z poznatků budou navrženy další možné kroky k rozšíření stávajícího prototypu.

Kapitola 1

Analýza

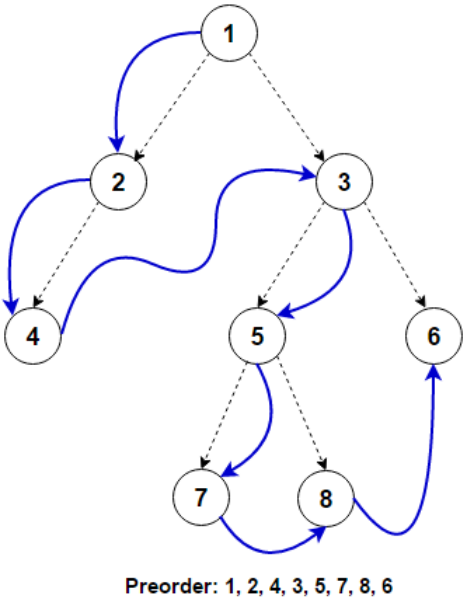
K prozkoumání možností nových prvků GUI je V první řadě potřeba prototyp řešení. V této kapitole si definujeme behaviorální stromy. Vysvětlíme si, jak fungují, z čeho se skládají a podíváme na nějaké již existující nástroje na trhu.

1.1 Behaviorální stromy

K pochopení problematiky vývoje behaviorálních stromů je první potřeba, aby se čtenář seznámil s pojmem jako takovým. Pojďme si tedy definovat behaviorální stromy a rozebrat si je více do hloubky.

1.1.1 Definice

Při vývoji „umělé inteligence“ pro účely ovládání herních postav potřebujeme nějakou kontrolní strukturu, pomocí které bychom mohli jednoduše a škálovatelně popsat její chování. **Behaviorální stromy** byly prvně definované V [1] jako HFSM (hierarchické konečné stavové automaty), nebo přesněji, behaviorální DAGy (orientované acyklické grafy). V těchto stromech dále popisují role jednotlivých typů uzlů. V typickém DAG schématu je rolí vnitřních uzlů dělat *rozhodnutí*, buďto podle nějaké logiky předdefinované pro každý typ vnitřního uzlu zvlášť, anebo mohou jeho potomci soutěžit a potomek s největší relevancí je vybrán. V tomto textu budeme pracovat s předpokladem, že každý typ vnitřního uzlu má svou vlastní rozhodovací logiku a budeme jim říkat **kontrolní uzly**. Listy potom reprezentují nějakou již specifickou *akci*, kterou postava vykonává. Listům budeme dále říkat **akční uzly**. Posledním typem „uzlu“ jsou takzvané **dekorátory**. Dekorátory jsou uzly, které se připojují k existujícím kontrolním nebo akčním uzlům a rozhodují o tom, jestli se daná akce či celý podstrom vyhodnotí. V některých behaviorálních systémech se jim proto říká „conditionals“. Stromy se vyhodnocují téměř vždy technikou *Preorder Traversal*, neboli od kořene směrem dolů se vyhodnotí právě vyhodnocovaný uzel, poté jeho levý podstrom a až poté pravý podstrom, jak můžeme vidět na obrázku 1.1. Tento průchod se děje každý simulační krok naší hry.



Obrázek 1.1 Preorder průchod stromem [2]

Dalším důležitým konceptem, se kterým bychom se měli seznámit je návratová hodnota uzlu. V behaviorálních stromech připouštíme 3 možné návratové hodnoty:

Succeeded Vyhodnocování uzlu se zdařilo a končí V tomto simulačním kroku

Running Vyhodnocování uzlu V tomto simulačním kroku neskončilo a bude V dalším kroku pokračovat (například uzel, který říká postavě kam ve světě se má pohnout vrátí **Running** během cesty k cíli)

Failed Vyhodnocování uzlu se nezdařilo a končí V tomto simulačním kroku

Zde je dobré ještě zmínit, že kontrolní uzly vždy vrátí tu stejnou hodnotu jako jejich právě vyhodnocený potomek, protože samy žádnou akci nevykonávají [3].

1.1.2 Kontrolní uzly

Typů kontrolních uzlů je V dnešní době mnoho. Odvíjí se také od toho, jaký zvolíte nástroj a jakou funkcionalitu jako uživatel potřebujete. Co se ale s jistotou říci dá je, že dva typy které nebudou chybět V žádném nástroji na tvoření behaviorálních stromů jsou **sekvence** a **selektory**.

Sekvence vyhodnocuje své potomky jeden po druhém. V případě, že potomek vrátí **Succeeded**, vyhodnotí následujícího potomka. Pokud potomek vrátí **Running**, pak vrátí sekvence také **Running** a V příštím iteračním běhu pokračuje vyhodnocováním tohoto potomka znova. Vrátí-li potomek **Failed**, pak vyhodnocování podstromu pod sekvencí končí se stejnou návratovou hodnotu.

Selektor by se dal popsat jako negace sekvence. Také vyhodnocuje své potomky jeden po druhém, ale skončí u prvního akčního uzlu, který vrátí **Succeeded**. V takovém případě vrátí selektor také **Succeeded**. Také může nastat situace, že ani jeden potomek selektoru neuspěje. V takovém případě vrátí i selektor **Failed**.

1.1.3 Akční uzly

Akční uzly mohou reprezentovat prakticky jakoukoliv činnost, kterou může naše postava vykonávat. Vzhledem k tomu, že jsou to listy našeho DAGu, jim většinou přiřazujeme jednu funkcionalitu k vykonání. Tady bych rád zmínil pár speciálních případů:

Reference Uzel odkazující na jiný BT. Umožňuje modularitu a znovupoužitelnost

Wait Pozastaví vyhodnocování tím, že buď určitý čas nebo dokud je nějaká podmínka nesplněna vrací návratovou hodnotu **Running**.

Script Provede nějakou naskriptovanou sadu instrukcí. V některých systémech je možnost vytvářet celé vlastní akční uzly.

Tyto typy listů jsou zpravidla nejgeneričtější a dají se přizpůsobit mnoha případům užití. Mohou ale plnit jakoukoliv činnost, jakou si může člověk vymyslet. Od přehrání animací, zaútočení na nějakou jinou postavu, schováním před nebezpečím nebo i komunikaci s jinými postavami.

1.1.4 Dekorátory

Jak jsme již popsali V sekci 1.1.1, dekorátory jsou uzly, které se připojují na jiné, již existující uzly. Jejich funkcí zpravidla je posuzování podmínky, jestli se má daný akční uzel či podstrom provést V závislosti na podmínce. z praxe ale můžu uvést ještě jiné užitečné příklady:

Restart If Vyhodnotí celý podstrom od začátku při splněné podmínce

Loop Vyhodnotí celý podstrom vícekrát po sobě, podle zadaného parametru

Succeeder Připojený uzel vrátí **Succeeded** nehledě na návratovou hodnotu vrácenou potomkem.

1.1.5 Blackboard proměnné

Častokrát během návrhu a vývoje narazíme na problematiku sdílení informací mezi různými uzly nebo dokonce mezi celými BT. K tomuto účelu můžeme právě použít proměnné uložené V takzvaném **Blackboardu**. Jedná se o datovou strukturu, která je většinou implementovaná buď na mozku, na kterém běží daný BT, nebo V některých systémech (jako například Unreal Engine) na kořenovém uzlu stromu. **Blackboard** většinou bývá hešovací tabulka, datová struktura která různým klíčům přiřazuje hodnoty a hodí se tak k uchovávání a sdílení dat mezi jednotlivými logickými celky [4]. Do této datové struktury můžeme z jednoho místa během vyhodnocování zapisovat nějaké data a V jiné části (při implementaci na mozku i V jiném BT) je zase číst. To nám umožňuje mít jakousi „paměť“ a sofistikovanější systémy chování.

1.2 Existující řešení

Jelikož behaviorální stromy jsou V dnešní době velice populární pro vývoj systémů umělé inteligence nejen ve hrách, je zcela přirozené, že nástroje určené k jejich tvoření již existují. Ať už zabudované přímo do herních enginů nebo jako pluginy, pojďme si některé z nich prozkoumat.

1.2.1 Unreal Engine

Tento herní engine vlastněný firmou Epic Games je V dnešní době jeden z nejpopulárnějších enginů na trhu. Mezi jeden z mnoha příčin jeho popularity patří krásná, až neuvěřitelně realistická kvalitou renderovacích schopností. Pyšní se titulama, jako jsou například The Callisto Protocol, Batman: City Arkham nebo i masově populární série her Borderlands. Pochopitelně V sobě má zabudovaný nástroj pro vytváření behaviorálních stromů [5].

První věc, které si můžeme všimnout při otevření editoru behaviorálních stormů, je že největší část plochy zabírá samotné plátno, do kterého sázíme uzly a hrany. Samotné uzly se dají přidávat více způsoby. Nejpraktičtější z nich je kliknutí pravým tlačítkem myši kamkoliv do plátna, přičemž se zobrazí seznam možných typů uzlů. Potvrzení volby se dá provést buďto kliknutím na zvolený typ uzlu, nebo použitím šipek pro výběr a stisknutím klávesy enter pro volbu. Toto menu je taky možné otevřít stisknutím klávesy „tab“. Co je pohodlnější už bude celkem subjektivní. Další bod, který bych chtěl vytknout je, že strom je orientován klasicky zvrchu dolů. Také se z UI prvků zdá, že právě na kořenovém uzlu jsou uloženy blackboard data.

Je zde i pár UI prvků, které méně zkušeným vývojářům určitě pomůžou a které bych tady rád zmínil. Jedním z nich je očíslování uzlů podle pořadí, V jakém se budou při vyhodnocování stromu vykonávat. Toto se může zdát jako trivialita, neboť to lze vyvodit z toho jak je strom poskládán, ale během toho, co jsem se já sám učit vyvíjet behaviorální stromy bych tento UI prvek určitě ocenil. Dalším prvkem je určitě popis typu uzlu při najetí na něj myší. U kurzoru se objeví menší okénko, kde je stručně ale výstižně popsané, jakou funkci uzel ve stromě vykonává.

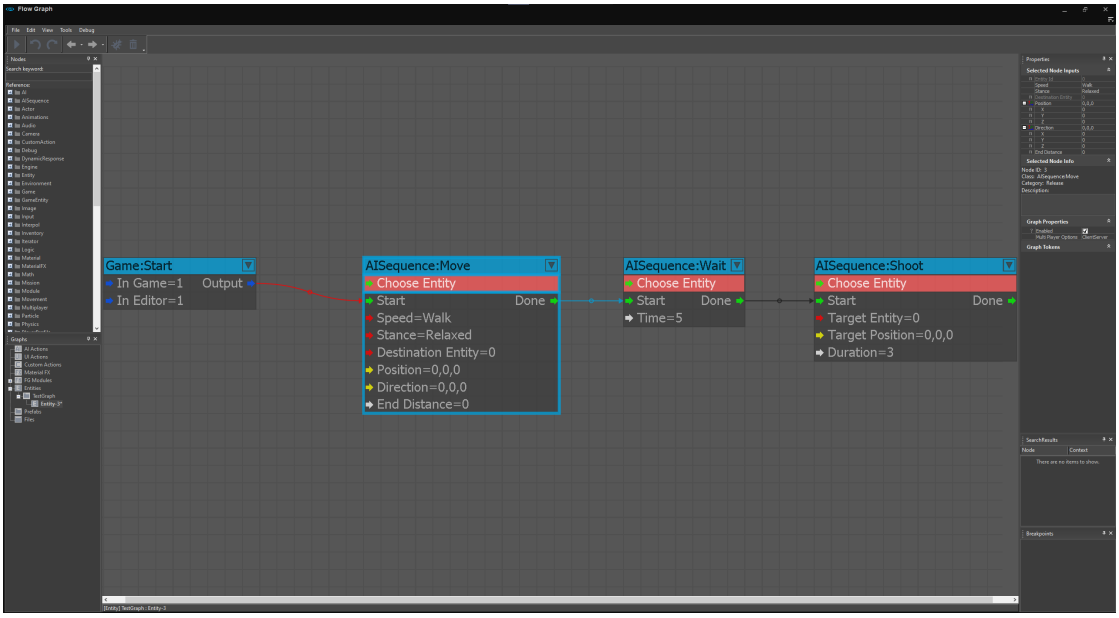
Poslední, na co bych chtěl poukázat, je že editor má V sobě zabudovanou funkcionalitu pro vytváření nových, uživatelem definovaných uzlů. Tyto uzly mohou od jiných dědit, mohou to být kombinace jiných uzlů nebo dokonce mít svoji vlastní implementaci. Tyto skripty se potom píší V jazyce C++.

1.2.2 Cryengine

Cryengine je herní engine od německé firmy Crytek. V nedaleké minulosti se tento engine spolu se sérií Crysis, která V něm byla vytvořená, zapsaly do síně slávy herního průmyslu. Po vydání první z dlouhé série her Far Cry V roce 2004 vycházela druhá verze Cryenginu, Cryengine 2 a Crytek potřebovali novou hru, se kterou by ukázaly jeho dovednosti. Nedlouho na to V roce 2007 vyšla hra Crysis, která předčila očekávání tehdejších hráčů [6]. Dokonce i dnes, 17 let později, vypadá hra pořád dobře. Pozdější iterace Crysis 2 a 3 na tom jsou obdobně. Dnes již trošku méně populární než za svých zlatých let, možná kvůli menšímu množství podpůrného materiálu, ale pořád je adekvátní volbou se zabudovaným editorem behaviorálních stromů [7].

Uživatelské rozhraní tohoto editoru je oproti Unreal Enginu znatelně starší a je zřejmé, že to nebyla hlavní věc, na kterou se vývojáři soustředili. Taky si můžeme povšimnout toho, že stromy jsou orientované zleva doprava jak je vidno na obrázku 1.2. Každý uzel V sobě má nějaké vstupní a výstupní atributy. Tento návrh uzlů a obecně editoru by mohl být povědomý uživatelům Blenderu, kde například Shader Editor vypadá velmi podobně, až na o poznání více moderní GUI.

Uzly se také přidávají způsobem „drag and drop“, neboli kliknutím a táhnutím z menu připnutého na levé straně GUI. Jednotlivé atributy uzlu lze editovat jak zvolením uzlu a pak následným upravování hodnot V menu připnutém k pravé straně, tak dvojitým kliknutím na atribut V samotném uzlu. Líbí se mi způsob vyhledávání, jaký má Cryengine implementován pro uzly. Uživatel vyhledá například typ uzlu a V malém menu se zobrazí všechny uzly, které s vyhledaným výrazem souvisí. Po kliknutí na jeden z výsledků se vám přiblíží a posune plátno tak, aby hledaný uzel byl ve středu obrazovky.

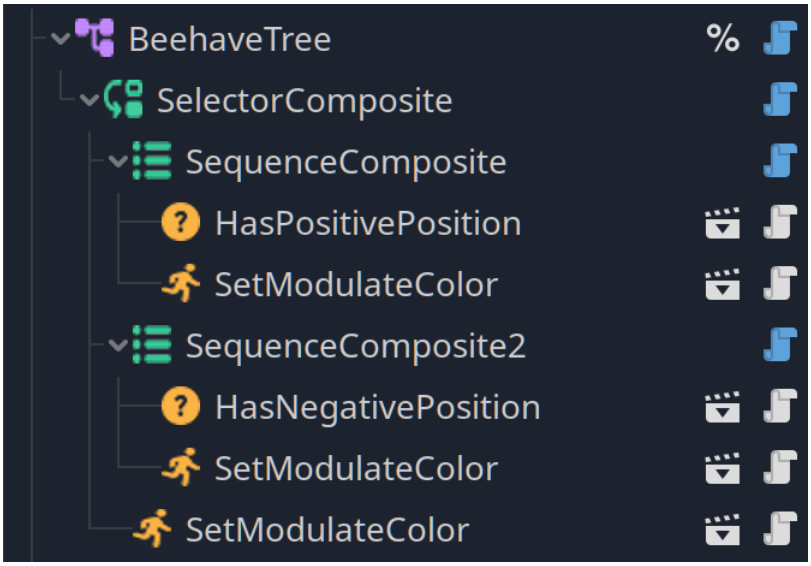


Obrázek 1.2 Cryengine editor behaviorálních stromů – FlowGraph

1.2.3 Godot – Beehave plugin

Dalším existujícím řešením, který bych zde chtěl zmínit je plugin Beehave do open-sourceového herního enginu Godot. Godot nemá svůj vlastní integrovaný BT editor, ale existuje nejeden plugin přidávající tuto funkcionalitu. Beehave přistupuje k vytváření stromů trošku jiným způsobem. Místo klasického plátna, do kterého uživatel vkládá krabičky reprezentující uzly a propojuje je hranami, řeší Beehave vytváření BT pomocí takzvaného rozložení **Tree view**. Tento styl vykreslení hierarchického seznamu se většinou používá k zobrazení adresářových struktur, můžeme si ale rozmyslet že V podstatě se také jedná o klasický strom s kořenem a listy. Ukázku tohoto vykreslení můžeme vidět na obrázku 1.3.

Beehave narozdíl od předchozích editorů obsahuje velmi omezený počet druhů uzlů. V základu obsahuje selektor, sekvenci, invertor a succeeder (a pár dalších, které zde nestojí za zmínku). Většinu uzlů si ale vývojář musí naimplementovat sám. Beehave dává vývojářům přístup k API V podobě abstraktních tříd „ActionLeaf“ a „ConditionLeaf“, které může vývojář naimplementovat a napsat si tak svůj vlastní akční uzel. Jako kontrolní uzly má pouze sekvenci a selektor a dekorátory obsahují několik užitečných ale jednoduchých uzlů [8]. Toto umožňuje mimo jiné obrovské množství na míru „ušitých“ uzlů, ale dle mého názoru za cenu vyšší obtížnosti naučení.



Obrázek 1.3 Strom vytvořený V pluginu Beehave pro herní engine Godot [8]

1.2.4 Control Editor

Control Editor je název aplikace, kterou momentálně ve společnosti BISim používáme k vývoji behaviorálních stromů pro vojenský simulátor, VBS4. Také je součástí produktu, který si zákazníci mohou koupit, tudíž to není jen interní nástroj. Je založený na Visual Studio SDK 2012, což už je poměrně starý development kit, a na vzhledu Control Editoru je to znát. Vzhledově se dost podobá vzhledu Visual Studia 2012, nástroje na vývoj desktopových aplikací od Microsoftu.

Control Editor se rozložením aplikace podobá editoru stromů V Unreal Enginu. Stromy jsou orientovány shora dolů. Vyhodnocují se preorder průchodem, vidno na obrázku 1.1. Jednotlivé části UI jsou modulární, jak je ve Visual Studiu zvykem. Typů uzlů má Control Editor mnoho, včetně těch nejzákladnějších. Tyto uzly jsme jako developeři této aplikace schopni přidávat, ale V aplikaci jako takové na to UI není.

Některé prvky UI by ale potřebovaly modernizovat. Přidávání uzlů je možné pouze přes „Toolbox“, boční panel odkud lze jednotlivé uzly přetáhnout. Tato akce učtitě lze zjednodušit a zrychlit. Dalším nepohodlným prvkem UI je přemisťování a spojování uzlů. Když chce uživatel uzel přemístit, musí na něj první kliknout a až poté jej může přetáhnout. Bez předchozího kliknutí se z uzlu začne vést hrana, což může být pro uživatele velmi neintuitivní. Aplikace vůbec nevyužívá kontextového menu pro interakci s plátnem a uzly. Samotný vzhled uzlů a obecně aplikace taky není nic k pohledání. Jako poslední věc, kterou bych vytkl, je že aplikace nemá tmavý mód, což může během delšího sezení unavovat oči.

Velká většina funkčních požadavků na tuto novou aplikaci vznikla během konverzací s kolegy, kteří Control Editor používají denně. Také proběhla jedna výuková hodina se studenty z Univerzity Karlovy, kteří s Control Editorem pracovali poprvé bez zkušeností s BT a já jsem měl příležitost u toho být a nejen si dělat poznámky, ale ptát se co bylo neintuitivní, nejasné atp. Více k těmto funkčním požadavkům V sekci 1.4

1.3 Formát ukládání BT

Je zřejmé, že budeme potřebovat stromy nejen vyvíjet, ale i nějakým způsobem ukládat a reprezentovat je ve vhodném formátu. Nejdůležitější vlastností takového formátu je čitelnost pro člověka. V rámci vývoje je přirozeným krokem procesu takzvaný „code review“, kde se nově

napsaný kód podrobuje kontrole jiného vývojáře, aby se předešlo chybám nebo nedodržení standardů daného kódu. Abychom tento krok procesu co nejvíce ulehčili, je třeba, aby formát zápisu stromů byl co nejjednodušší a nejčitelnější. V práci používáme formát JSON, textový formát, který facilituje přenos strukturovaných dat mezi velkou škálou programovacích jazyků [9]. Například ve výše zmiňovaném Cryengine používají formát XML, další z textových formátů, které jsou pro člověka celkem lehce čitelné. Má podobnou syntaxi jazyku HTML a také je V něm možné vytvářet objekty s různými atributy, stejně jako V JSON [10].

1.3.1 Výchozí formát pro tuto práci

Jedním z cílů této práce je zjednodušení formátu, který aktuálně V BISim používáme pro ukládání BT. Pojďme si V této sekci analyzovat, jak aktuální verze tohoto formátu vypadá a V pozdější části této práce se jej pokusíme zjednodušit. Jak jsem již zmínil, soubory jsou ve formátu JSON, tudíž si popíšeme jednotlivé položky, které aktuální řešení Control Editoru V BISim očekává V každém souboru reprezentující BT.

Atribut	Typ	Význam
name	string	jméno uzlu
id	string	unikátní identifikátor uzlu
type	string	typ uzlu
subtrees	array	seznam potomků uzlu
meta	array	seznam extra informací ohledně uzlů a stromu
parameters	array	seznam parametrů pro daný BT
locals	array	seznam lokálních proměnných

Tabulka 1.1 Atributy formátu očekávané Control Editorem

Popišme si pro lepší pochopení každý atribut trošku více do hloubky:

name Jméno, které se zobrazí na uzlu jako pojmenování viditelné V editoru

id Unikátní identifikátor, podle kterého jsme schopni rozlišit každý uzel ve stromě. Obvykle se generuje pomocí algoritmů, které garantují, že každý nový identifikátor je opravdu unikátní.

type Typ uzlu. Může to být cokoliv od selektoru, sekvence, pohybu, výstřelu, reference a mnoho dalších.

subtrees Seznam potomků. Tito potomci obsahují stejné atributy jako dosud vyjmenované. Zbytek atributů se týká jen kořenového uzlu.

meta Seznam extra parametrů. Obsahuje pozice uzlů na plátně, celkovou velikost plátna, koeficient přiblížení a jiné.

parameters Seznam parametrů předané stromu k vyhodnocování. Může to být například pozice ve světě, kam se má entita pohnout nebo cíl, po kterém má entita vystřelit, následovat jej atd.

locals Seznam lokálních proměnných. V klasických BT systémech se toto řeší přes blackboard, ale V naší implementaci jsou lokální proměnné pro sdílení dat V rámci jednoho stromu implementované na kořenovém uzlu. Blackboard je pak na samotném mozku, který vyhodnocuje tyto stromy.

1.4 Katalog požadavků

V této sekci si shrneme požadavky na nový editor. Vytyčíme si, co by měla aplikace umět, jaké by měla mít vlastnosti. Stejně důležité je ale konkrétně definovat, co aplikace umět *nebude*. Tímto předejdeme situaci, kdy si bude muset vývojář domýšlet nějaké informace.

1.4.1 Funkční požadavky

Funkční požadavek nám říká, co by aplikace měla dělat. Využívá se k identifikaci a specifikaci nějaké funkcionality naší aplikace. Shrňme si pár žádoucích vlastností funkčních požadavků. Funkční požadavky by dle [11] měly:

- Specifikovat co má aplikace umět, ne jakým způsobem toho docílí.
- Definovat, jak má aplikace reagovat na platné i neplatné vstupy, tj. jaké má pro tyto vstupy mít výstupy.
- Být stručné a výstižné
- Být formulované V pozitivním smyslu. Ověřit, že se něco neděje je těžší než ověřit, že se něco děje.

Teď když už víme, co to funkční požadavky jsou, pojďme si tyto požadavky definovat. Začneme funkčními požadavky kladené uživateli na aplikaci:

F1 Vytvoření nového BT
Uživatel může pomocí aplikace vytvořit nový strom. Může do něj přidat uzly, tyto uzly spojovat hranami a upravovat jejich atributy.

F2 Editování již existujícího BT
Uživatel může pomocí aplikace otevřít předem vytvořený soubor. Aplikace načte všechny informace a zobrazí strom. Tento strom je dále možno upravovat stejně, jako nově vytvořený.

F3 Uložení vytvořeného BT
Uživatel může uložit strom do souboru.

F4 Zobrazení projektu (soubor .btset)
Uživatel může otevřít projektový soubor, zobrazit si jednotlivé stromy V projektu a případně je upravovat dle libosti. Uživatel nemůže upravovat projekt jako takový, tj. přidávat do projektu nové soubory nebo soubory z projektu mazat.

F5 Více módů zobrazení stromů
Uživatel může přepínat mezi horizontálním a vertikálním směrem zobrazování stromů.

F6 Zobrazení stromu V referenčním uzlu
Uživatel může zobrazit V referenčním uzlu graf stromu, na který se uzel referuje.

F7 Jednoduché přidávání uzlů do stromu
Uživatel bude moci přidávat uzly do stromu jednoduchými gesty. Při kliknutí pravého tlačítka myši do plátna se zobrazí menu k přidání uzlu. Při kliknutí pravého tlačítka myši na uzel se nově vytvořený uzel přidá jako potomek zvoleného.

F8 Vyhledávání ve stromech
Uživatel bude moct ve stromě vyhledávat pomocí textu různé uzly podle jména. Nalezené uzly budou zvýrazněny.

F9 Zobrazení rozdílu mezi různými verzemi souboru
Uživatel bude moct pomocí gitu vytvořit .patch soubor, který po načtení do aplikace zobrazí vizualizace rozdílů.

F10 Zobrazení průchodu stromu
Uživatel si může zapnout zobrazení průchodu stromem. Jednotlivé uzly budou mít u sebe číslo značící pořadí, V jakém se budou vyhodnocovat.

F11 Vytvoření souboru z podstromu
Uživatel bude mít možnost z podstromu vytvořit zcela nový strom a uložit jej do samostatného souboru.

F12 Náhled kde nový uzel vznikne
Při přidávání uzlu bude aplikace zobrazovat náhled, kde se nový uzel objeví ještě před přidáním.

F13 Automatické zarovnávání do mřížky
Během pohybování s uzlem bude automaticky poskakovat mezi pozicemi V zarovnané mřížce.

1.4.2 Nefunkční požadavky

Nefunkční požadavek [11] nám říká, jaké vlastnosti by měla naše aplikace splňovat. Souvisí s nimi časování, vývojářské procesy a standardy. Nefunkční požadavky se vztahují k celé aplikaci namísto k jednotlivým funkcionalitám.

Nefunkční požadavky kladené na náš nový editor uživateli jsou následující:

N1 Tmavé GUI
Uživatelné rozhraní bude V „tmavém režimu“.

N2 Kompatibilita s formátem Control Editoru
Aplikace bude schopná pracovat se stromy vytvořenými V programu Control Editor.

N3 Jednodušší formát ukládání souborů
Formát, ve kterých budou soubory ukládány, bude založen na formátu z Control Editoru, ale bude zjednodušen a nepotřebné data budou vynechány.

N4 Podpora pro Windows
Aplikace bude primárně určena jako desktopová aplikace pro operační systém Windows.

N5 Moderní vzhled GUI
Uživatelské rozhraní bude používat zásady „material designu“, standardem pro vzhled aplikací vytvořených Googlem.

1.4.3 Priority požadavků

V této sekci si rozdělíme funkční i nefunkční požadavky do kategorií podle priorit. Bude se jednat o kategorie **Must have**, **Should have** a **Could have**. Požadavky V kategorii **Must have** považujeme za funkcionalitu, bez které se aplikace neobejde. **Should have** požadavky pokrývají takovou funkcionalitu, bez které se obejdeme, ale pořád aplikaci přináší značnou hodnotu. Požadavky z kategorie **Could have** bychom V aplikaci rádi uviděli, ale jsou jakýmsi bonusem nad rámec základních funkcí [12]. Pojďme si naše funkční a nefunkční požadavky rozdělit do těchto kategorií V následující tabulce:

Must have	Should have	Could have
F1	F8	F9
F2	F10	F11
F3	N1	F12
F4	N5	F13
F5		
F6		
N2		
N3		
N4		

Tabulka 1.2 Tabulka znázorňující priority funkčních a nefunkčních požadavků

1.4.4 Uživatelské persony

Uživatelská persona je jakási fiktivní postava, kterou používáme k návrhu a designování počítačových aplikací [13]. Pomáhají nám reprezentovat různé skupiny uživatelů. V této práci budeme pracovat se třemi různými uživatelskými personami:

Zkušený BT developer Vyvíjí BT jako součástí náplně své práce. Ví, jak behaviorální stromy fungují, jaký uzel provádí jakou činnost a pravděpodobně by byl rád, kdyby si mohl své vlastní uzly vytvářet sám.

Herní vývojář Jednotlivec, který vyvíjí nějakou herní funkcionalitu a potřeboval by si vytvořit nějaký jednoduchý BT pro otestování své funkcionality, ale nemá s vývojem BT přímo žádnou zkušenost.

Modovací nadšenec Nikdy nevyvíjel žádnou hru, ani nemá zkušenosti s programováním, ale rád by si udělal mod do své oblíbené hry.

1.5 Případy užití

Případy užití [11] jsou technika ke zjišťování požadavků na vyvíjený software. V dnešní době se jedná o jeden ze základních pilířů UML. Případ užití lze zjednodušeně popsat jako interakci mezi uživatelskou personou a softwarem a umožňuje této interakci dát název. Při analýze a návrhu složitějších systémů je vhodné tyto vztahy mezi akcemi a uživateli vizualizovat pomocí takzvaných „use case“ diagramů, ale V našem případě si vystačíme s textovou podobou, jelikož jediný typ uživatele pro naší aplikaci bude vývojář stromů.

Pojďme si vypsat několik častých případů užití, které se naší aplikace týkají:

UC1 Vytvoření nového stromu.

UC2 Otevření a zobrazení existujícího stromu.

UC3 Uložení aktuálního stromu do souboru.

UC4 Otevření a zobrazení projektu behaviorálních stromů.

UC5 Přidání nového uzlu do stromu.

UC6 Změna atributu nějakého uzlu.

UC7 Zobrazení diagramu stromu, na který se referenční uzel odkazuje.

UC8 Zobrazení rozdílu mezi verzemi stromu.

UC9 Extrakce funkcionality podstromu do samostatného souboru.

UC10 Vyhledání specifického uzlu.

UC11 Přepnutí módu zobrazení stromů.

UC12 Pochopení chování implementované stromem.

UC13 Úprava rozložení stromu.

1.5.1 Pokrytí funkčních požadavků případy užití

Pro přehled si zde ještě uvedeme tabulku, kde lze názorně vidět jaké funkční požadavky jsme pomocí našich případů užití pokryli.

Funkční požadavek	Případ užití
F1	UC1
F2	UC2, UC5, UC13
F3	UC3
F4	UC4
F5	UC11
F6	UC7
F7	UC5
F8	UC10
F9	UC8
F10	UC12
F11	UC9
F12	UC5
F13	UC13

Tabulka 1.3 Tabulka znázorňující pokrytí funkčních požadavků pomocí případů užití

Kapitola 2
Návrh

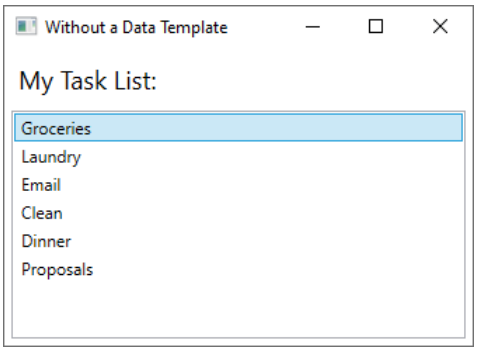
Další součástí procesu vývoje aplikace je volba technologie, kterou použijeme k jejímu vytvoření. Nechybí v tomto kroku ani návrh architektury aplikace včetně diagramů pro znázornění. V neposlední řadě v této kapitole navrhneme GUI naší aplikace. Ukážeme nějaké jednoduché wireframy, ze kterých konečná aplikace čerpala inspiraci.

2.1 Výběr technologie

V první sekci této kapitoly se podíváme na to, jaké technologie jsou k dispozici pro tvorbu aplikace převážně soustředěné na GUI. Vytyčíme si různé pro a proti několika možných „tech stacků“ a nakonec vybereme jednu možnost, kterou použijeme pro realizaci naší aplikace.

2.1.1 Windows Presentation Foundation (WPF) a .NET

WPF popisuje oficiální dokumentace jako „architekturu uživatelského rozhraní, která je nezávislá na rozlišení a používá vektorový vykreslovací modul navržený tak, aby využíval moderní grafický hardware“ [14]. Dále zde vysvětlují, že WPF je součástí rozhraní .NET, tudíž je možno používat prvky rozhraní .NET API. WPF se dá považovat za následníka „Windows Forms“, jež byla knihovna k vytváření jednoduchých, ač ne moc pohledných GUI aplikací v minulosti. Ačkoliv se knihovna prezentuje jako moderním nástupníkem Windows Forms, dle mého názoru už na dnešní dobu taky vypadá celkem zastarale, jak je možno vidět na obrázku 2.1.



Obrázek 2.1 Příklad GUI vytvořeného pomocí WPF [14]

Nyní si zde shrneme nějaké pro a proti k této technologii:

Pro:

- Výkon aplikace
- Velmi dobrá integrace s operačním systémem Windows
- Aplikace bude díky vzhledu podobná ostatním aplikacím na platformě Windows

Proti:

- Bez zkušeností může být obtížnější
- Není možnost rozvést řešení na více platforem než Windows
- V dnešní době už vypadá zastarale

2.1.2 Qt Framework a C++

Qt Framework obsahuje řadu vysoce intuitivních a modularizovaných knihoven pro C++ třídy spolu s mnoha API, které ulehčují vývoj aplikace. Produkuje čitelný, lehce udržitelný a znovupoužitelný kód s velmi dobrou výkonnostní stopou. Tato knihovna také podporuje jak vývoj desktopových aplikací, tak mobilních i vestavěných aplikací. Pomocí knihovny „Qt Essentials“ implementuje společnou funkcionalitu na všech podporovaných operačních systémech [15]. Příklad vzhledu i multiplatformnosti můžeme vidět na obrázku 2.2.

Obrázek 2.2 Příklad GUI vytvořeného pomocí Qt Frameworku na mnoha různých zařízeních [15]

Shrňme si pro tento „tech stack“ nějaké klady a zápory:

Pro:

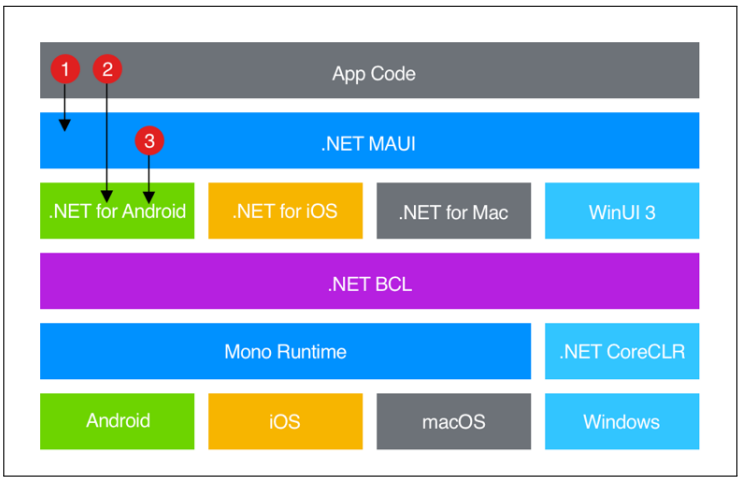
- Nesmírný výkon aplikace
- Jedna kódová základna se použitelná na celé škále zařízení a platforem
- Nespočet předdefinovaných a přizpůsobitelných widgetů
- Podporuje multi-threading
- Obsahuje lokalizační knihovny

Proti:

- Závislost na „Qt Runtime“ – uživatelé musí mít nainstalovaný Qt Runtime pro spuštění aplikace, což může kompikovat distribuci
- Vzhledem k tomu, že se Qt Framework váže na C++, bude čas strávený vývojem vysoký, neboť vývoj v tomto jazyce je komplikovaný
- Škála knihovny může být pro menší aplikace zbytečná. Hodí se více na rozsáhlejší projekty

2.1.3 .NET MAUI

.NET Multi-platform App UI (.NET MAUI) je multiplatformní rozhraní pro vytváření nativních mobilních a desktopových aplikací pomocí C# a XAML. Stejně jako WPF umožňuje rozhraní .NET MAUI vyvíjet multiplatformní aplikace, které mohou běžet jak na mobilních operačních systémech jako jsou Android nebo iOS, tak na desktopových systémech macOS či Windows. Umožňuje distribuci na všechny tyto systémy z jednoho kódového základu [16]. Lze jej považovat za nástupce WPF a také je rozšířením knihovny „Xamarin.Forms“. Pro ilustraci architektury vizte obrázek 2.3.



Obrázek 2.3 Diagram architektury knihovny .NET MAUI [16]

Opět si shrneme klady a zápory tohoto technologického řešení:

Pro:

- Multiplatformní přístup
- Podporuje nové technologie, jako .NET 6.
- Na Windows využívá nativní rozhraní WinUI 3

Proti:

- Je to relativně nová knihovna, takže ještě není úplně dodělaná a někdy může řešení obsahovat chyby
- Vzhledem k novotě chybí podpůrné materiály a komunita

2.1.4 Flutter

Flutter je open-source UI SDK vyvíjené Googlem. Jako většina možností zmíněných výše je také multiplatformní, umožňující vývoj a distrubuci jak pro mobilní operační systémy jako Android a iOS, tak pro desktopové systémy macOS, Windows a systémy založené na Linuxu [17]. Umožňuje ale vývoj i pro web, je tedy ze všech možností zatím nejflexibilnější. Flutter aplikace se nepřekládají do nativního kódu pro platformu, na které běží. Běží na „Flutter rendering engine“, napsaný v C++ a Flutter frameworku, který je ostatně jako Flutter aplikace napsán v programovacím jazyce Dart, taky vyvíjen Googlem [18]. Tato architektura ale trochu přidává na velikosti aplikací, jelikož je potřeba tento engine i framework zabalit do distribuce.

Zde se můžeme nejspíše zdržet obrázku, neboť mnoha nám známých aplikací od googlu využívá Flutter a tak si pro ilustraci toho, jak vypadá UI vytvořené ve Flutter můžeme otevřít například Google Pay nebo Google Earth. Dokonce i světově známý čínský e-shop Alibaba využívá Flutter pro své webové stránky [19].

Nesmí chybět porovnání kladů a záporů této technologie:

Pro:

- Možnost vyvíjet jak pro mobilní platformy, tak desktopové a dokonce i pro web.
- Velmi populární a vyvíjený Googlem, což znamená obrovské množství podpůrných materiálů a kvalitní dokumentaci
- Obsahuje velkou škálu widgetů, pomocí kterých můžeme aplikaci postavit
- Drží se zásad „material design“, což je jakási směrnice jak by měly vypadat uživatelsky přívětivé aplikace

Proti:

- Nutnost balení i renderovacího enginu trošku nafukuje velikost aplikací
- Dart je celkem nový a ne úplně rozvinutý jazyk, takže učit se jej je práce navíc
- Výkon může být v porovnání s předchozími možnostmi horší, neboť musí renderovací engine komunikovat s nativním rozhraním a to může způsobovat zpomalení

2.1.5 Electron, Typescript a Vue.js

Tento „tech stack“ se skládá z více samostatných částí, které dohromady umožňují vytváření GUI desktopových aplikací. Pojďme si každou z nich prozkoumat trošku více do hloubky.

Prvně se podíváme na **Electron**. Electron je framework, který nám umožňuje vytváření desktopových aplikací pomocí JavaScriptu, HTML (Hypertext markup language) a CSS (Cascading style sheets). Tyto technologie se standardně používají k vyvíjení webových stránek, ale díky Electronu není potřeba znát nativní technologie a je možnost využít zkušenosti s vyvíjením pro web také pro vyvíjení desktopových aplikací. Umožňuje multiplatformní řešení jak pro Windows, tak pro macOS a Linux [20].

Dále si popíšeme **TypeScript**. TypeScript je silně typovaný programovací jazyk, který je postavený na JavaScriptu. Přidává k JavaScriptu syntaxi navíc k popisu typů proměnných a pomáhá zlepšit integraci s kódovým editorem. TypeScriptový kód se poté kompiluje do JavaScriptu, tudíž ho lze použít kdekoliv, kde lze použít JavaScript. Také pomáhá zachytit chyby v kódu před spuštěním programu, což šetří čas při vývoji [21].

Jako poslední přichází na řadu **Vue.js**. Vue.js je JavaScriptový framework určený k vývoji grafických uživatelských rozhraní. Staví na HTML, CSS a JavaScriptu a poskytuje deklarativní programovací model založený na komponentách. Vue.js je navržen tak, aby pokrýval velkou většinu funkcí, kterou na webu potřebujeme [22]. V kontextu této práce bychom tento framework využili pro vytvoření takzvané „Single-Page Application“, což ve zkratce znamená, že budeme měnit obsah zobrazované stránky místo načítání zcela nové stránky.

Kombinace těchto technologií nám umožní tvořit „webové stránky“, které jsou schopny komunikovat s operačním systémem pomocí Electronu. Vyjmenujme si nějaké kladné a záporné stránky tohoto možného řešení:

Pro:

- Multiplatformní možnosti z jednoho kódového základu
- Můžeme použít moderní technologie k vyvíjení pohledných webových stránek (knihovny podporující material design)
- Vývoj GUI aplikace stylem webové aplikace bývá o mnoho rychlejší než používaní nativních technologií
- Podpůrných materiálů pro tyto knihovny, dokonce i v kombinaci s Electronem, je nespočet
- Možnost jednoduše zpřístupnit aplikaci i z webového prohlížeče.

Proti:

- Electronové aplikace jsou notoricky známé využíváním velkého množství paměti
- Nutnost distribuovat celý balík Chromia i Node.js způsobuje větší velikost instalačních souborů
- Jelikož se aplikace řešená tímto způsobem skládá ze dvou procesů, výkon aplikace může být pomalejší než nativní řešení

Pro řešení zadání této bakalářské práce využijeme výše zmiňovanou kombinaci technologií Electron, TypeScript a Vue.js. K těmto technologiím přibudou ještě další podpůrné knihovny, které si popíšeme v následující sekci.

2.1.6 Další použité knihovny

V této sekci si popíšeme další podpůrné knihovny, které jsou využity v aplikaci. Jedná se čistě o Vue.js knihovny, které nám ulehčí práci s vývojem GUI. Pojďme si je vyjmenovat a podívat se na ně trošku blíže:

PrimeVue PrimeVue je vývojová sada UI komponent vyvinutá právě pro Vue.js. Obsahuje bohatou sadu UI komponent, bloků a šablon, které do své aplikace můžeme vložit a fungují takzvaně „out of the box“ [23]. Tyto komponenty také dodržují zásady „material design“.

VueFlow VueFlow je knihovna implementující interaktivní editor blokových diagramů a grafů. Obsahuje komponenty pro plátna, do kterých se grafy dají modelovat. Také obsahuje přizpůsobitelné šablony jak pro uzly, tak pro hrany. V neposlední řadě také implementuje nějaké užitečné utility, jako je například miniaturní mapa, malý panel se základním ovládáním nebo okno s uzly, které se dají přetáhnutím přidat do plátna [24].

VueUse VueUse je kolekce utility funkcí založených na Vue.js. Obsahuje například funkce pro zpracovávání vstupu od uživatele přes klávesnici a myš, práci s časem, práci s asynchronním kódem, animace a jiné [25].

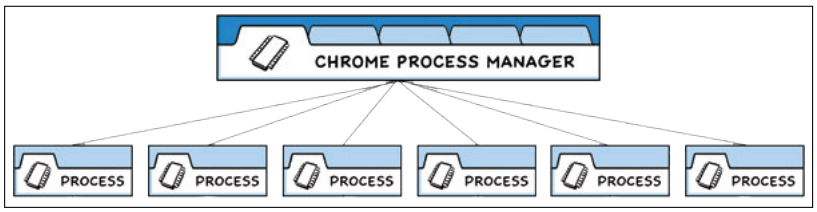
Pinia Pinia je knihovna pro správu stavu aplikace. Umožňuje přístup ke stavu z jakékoliv stránky či komponenty. Také se hodí během vyvíjení jelikož podporuje funkcionalitu „hot reload“, což znamená, že při změně zdrojového kódu se stránka jen obnoví ale stav zůstane stejný [26].

Dagre.js Dagre je JavaScriptová knihovna, která automaticky počítá umístění uzlů pro orientované acyklické grafy, nebo z našem případě stromy. Stačí jí předat uzly s definovanými vztahy rodičů a potomků a Dagre nám vrátí uzly se správnými pozicemi.

2.2 Architektura aplikace

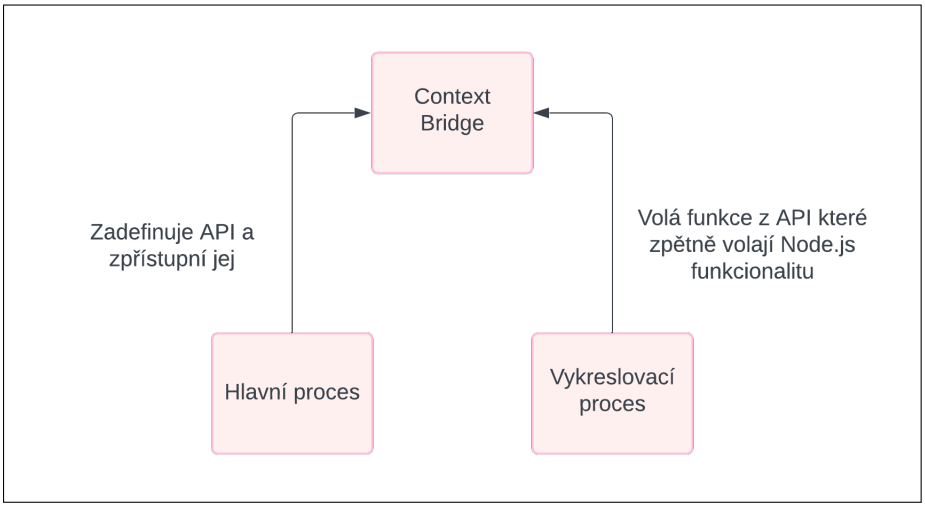
Důležitou součástí vývoje softwaru je typ architektury, který na dané řešení aplikujeme. Jak jsme již zmínili v 2.1.5, zadání této bakalářské práce bude implementováno jako „desktopová webová stránka“. Abychom tohoto dokázali, použijeme framework s názvem Electron.

Electron v sobě obsahuje balíček s názvem Chromium, aby mohl emulovat chování klasického webového prohlížeče. Co si také převzal od prohlížečů založených na Chromiu je dvouprocesová architektura znázorněna na obrázku 2.4. Vývojář má k dispozici dva procesy, **hlavní** (main) a **vykreslovací** (renderer). Hlavní proces je v každé aplikaci jen jeden a považujeme jej za vstupní bod našeho programu. Tento proces běží v Node.js prostředí, může importovat Node.js moduly a využívat všechny API poskytované Nodem. Tento proces má za primární úkol vytváření oken, ve kterých zobrazujeme GUI pro naši aplikaci. Každé toto okno se dá považovat jako samostatná webová stránka. Vykreslovací proces má poté za úkol, jak jistě z názvu lze předpovědět, vykreslování obsahu stránky (v našem případě uživatelské rozhraní aplikace) [27]. Těchto procesů může být více, my ale v této práci budeme pracovat pouze s jedním.



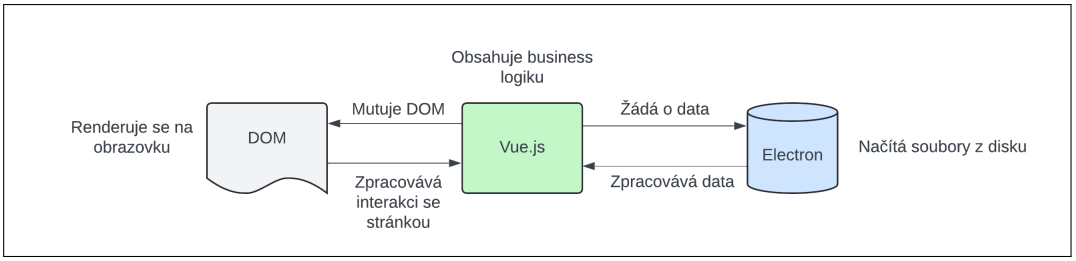
Obrázek 2.4 Diagram architektury webových prohlížečů založených na Chromiu [27]

Zde je dobré vysvětlit jeden důležitý detail. Prostředí Node.js a rozhraní pro funkcionalitu specifickou k desktopovým aplikacím je přístupné pouze z hlavního procesu. Častokrát ale budeme chtít reagovat na interakci uživatele s GUI nějakou touto funkcionalitou (například zobrazení dialogu pro volbu a otevření souboru). Electron toto řeší pomocí takzvaných „preload skriptů“. Tyto skripty obsahují kód, který běží ve vykreslovacím procesu ještě před tím, než se začne načítat obsah stránky. Na rozdíl od zbytku kódu běžící na tomto procesu ale mají tyto skripty přístup k Node.js API. Vývojář je tedy schopen nadefinovat své vlastní API, které může volat z vykreslovacího procesu a ve kterém se nechá specifikovat zpětné volání funkcí na straně hlavního procesu. K tomuto se využívá takzvaný „Context Bridge“ [27]. Pro ilustraci vizte obrázek 2.5.



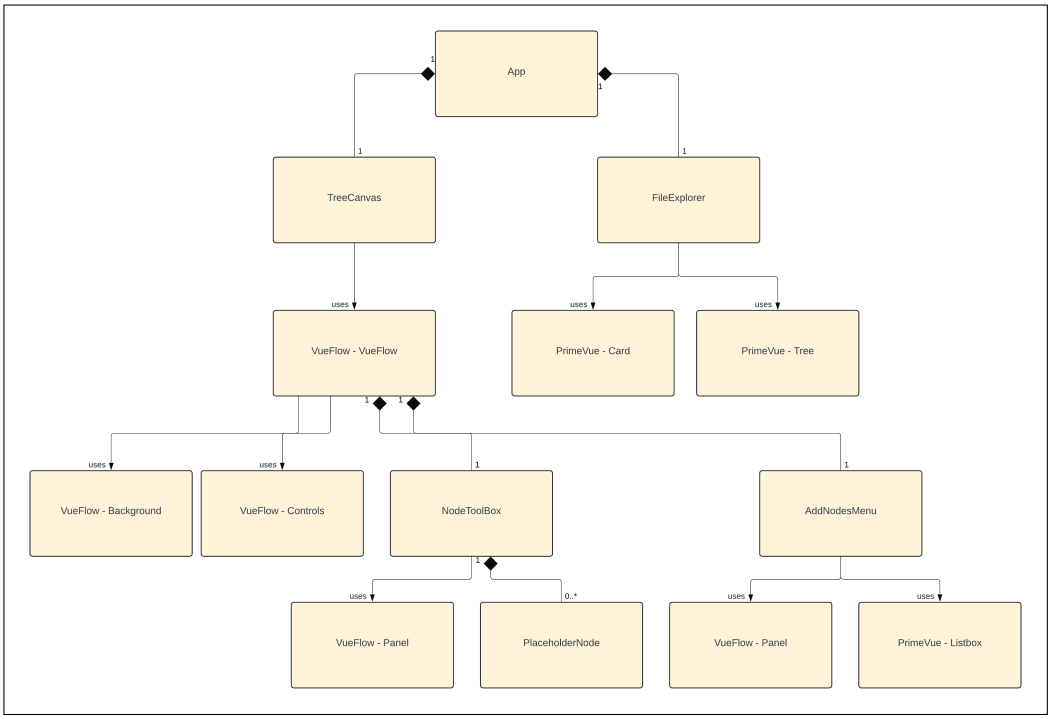
Obrázek 2.5 Diagram vytvoření API mezi hlavním a vykreslovacím procesem pomocí Context Bridge

Nyní si popíšeme, jak je navržena architektura samotné webové stránky. Vue implementuje ViewModel část MVVM (Model-View-ViewModel) architektury. Propojuje View a Model části pomocí dvousměrné datové vazby. Modelovou část naší aplikace bude v podstatě provádět Electron, přes který budeme načítat soubory reprezentující stromy. Vue poté reaktivně na měnící se data mutuje DOM (Document Object Model) neboli obsah, ze kterého se webová stránka skládá. Pro ilustraci vizte obrázek 2.6



Obrázek 2.6 Diagram architektury aplikace využívající knihovnu Vue.js

Jak jsme již zmiňovali, naše aplikace bude implementovaná jako „Single-page application“. To znamená, že všechny části uživatelského rozhraní budou načítány dynamicky a stránka se nikdy nebude obnovovat. Pro tvorbu GUI jsem se rozhodl použít JavaScriptovou knihovnu Vue.js, což je knihovna s architekturou založenou na komponentách. Tato architektura je založena na znovupoužitelných „součástkách“, které pak můžeme dávat dohromady a vytvořit z nich interaktivní aplikaci. Každá komponenta pak obsahuje nějakou svoji zapouzdřenou funkcionalitu, která se týká jen jí samotné [28]. Struktura aplikace je ilustrována v následujícím diagramu vztahů komponent 2.7.

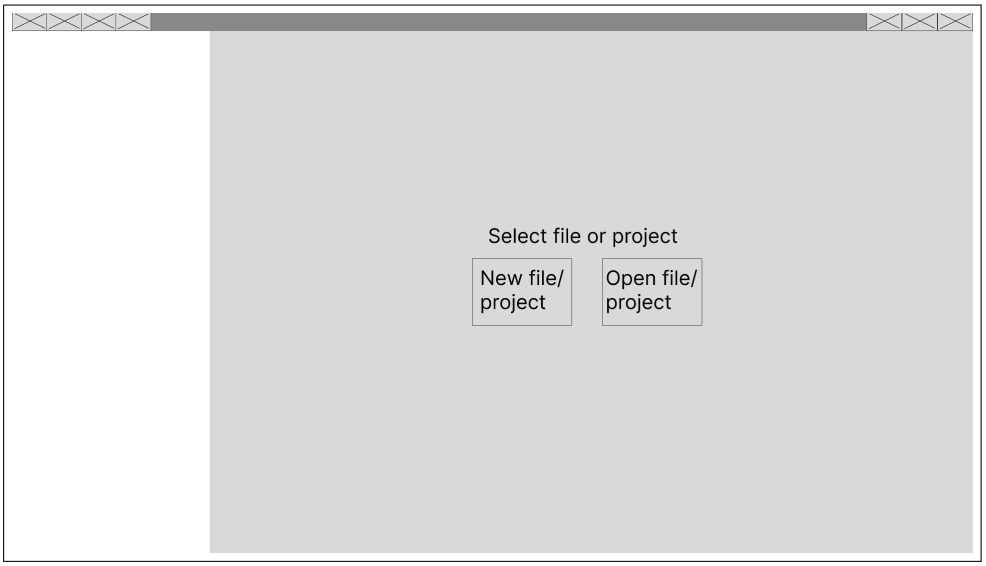


Obrázek 2.7 Diagram architektury uživatelského rozhraní této práce

2.3 Návrh uživatelského rozhraní

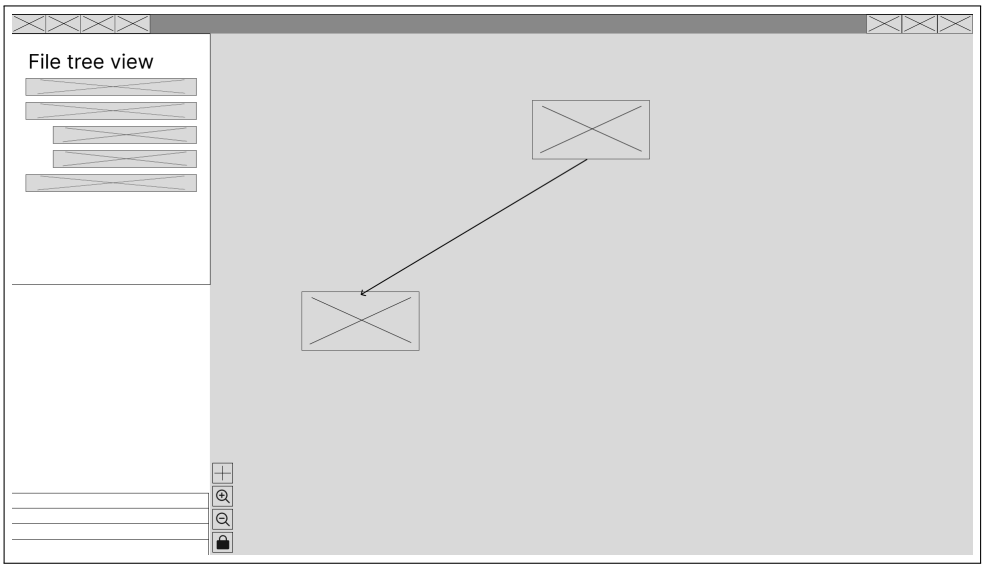
V této sekci si popíšeme návrh GUI pro naši aplikaci. Tento popis podpoříme ukázkami takzvaných „wireframů“, což jsou jednoduché prototypy návrhu uživatelského rozhraní, které se soustředí na rozložení obrazovky a nijak neřeší styl a vzhled. Tyto wireframy se vyplatí vytvářet jako jakýsi hrubý základ, jak by mohla naše stránka být rozvržena a umožňuje nám iterovat rychleji, protože není třeba stránku stylizovat.

Pro naši aplikaci budeme potřebovat nějakou část uživatelského rozhraní, která se bude starat o načítání a správu aktuálně otevřených souborů či projektů. Této komponentě, která bude připnutá k levé straně obrazovky říkáme trošku nepřekvapivě „File Explorer“. Dále potřebujeme samotné plátno, do kterého budeme vkládat uzly a celkově vyvíjet náš BT. Toto plátno bude připnuté k pravé straně obrazovky a budeme mu říkat „Tree Canvas“. Jednoduchá vizualizace rozložení aplikace při prvotním spuštění lze vidět na obrázku 2.8.



Obrázek 2.8 Wireframe prázdné aplikace

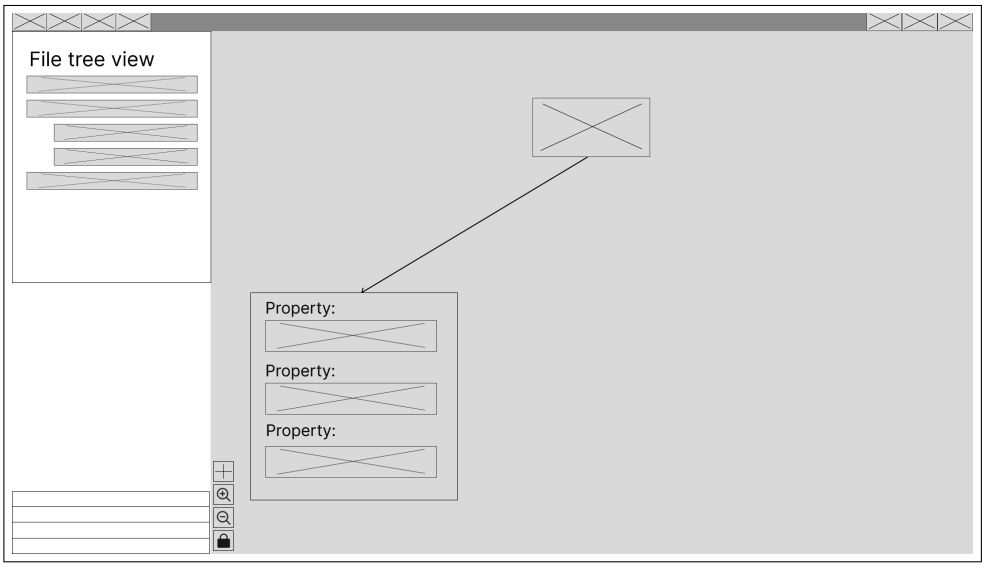
Dále si popíšeme trošku více dopodrobna, jak bude každá z těchto částí rozložená a co bude obsahovat. Začněme u File Exploreru. Ten bude v sobě obsahovat rozepsanou strukturu projektu nebo souboru, který je momentálně otevřen. V případě souboru bude rozhraní vypadat jako projekt s jedním souborem. Tato struktura bude reprezentována pomocí reprezentace „Tree View“, které je popsané v 1.2.3 a lze vidět v 1.3. Samotné plátno zobrazuje graf stromu, který je aktuálně otevřen. Obsahuje uzly propojené hranami. V levém spodním rohu je skupina tlačítek, které nám umožňují základní manipulaci s plátnem. Mezi těmito tlačítky je i tlačítko k otevření panelu k přidávání nových uzlů. Pro ilustraci vizte obrázek 2.9.



Obrázek 2.9 Wireframe běžného vzhledu aplikace při vývoji stromů

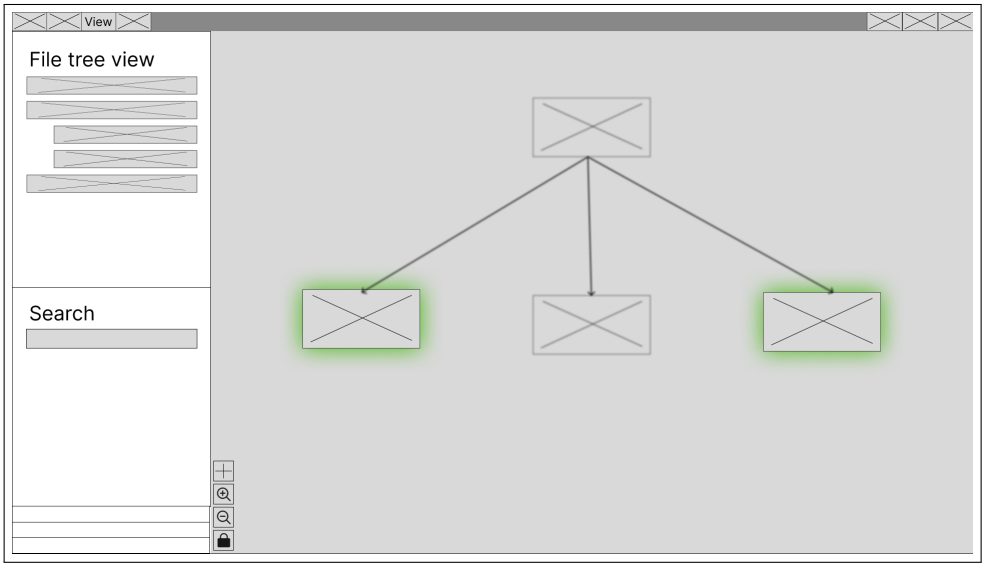
Uzly budou mít nepřekvapivě nějaké atributy. Tyto atributy budeme chtít měnit. V mnoha editorech je toto řešeno dedikovaným bočním panelem, který při nezvoleném uzlu neobsahuje nic a jen zabírá místo na ploše, které by bylo lépe využito zvětšením plátna. Nově navržený vzhled

by místo tohoto nešikovného řešení editovaný uzel rozbalil a UI na změnu atributů zobrazil v „nafouklé“ verzi tohoto uzlu. Tímto se ušetří místo na obrazovce a během doby, kdy se žádný uzel needituje nebude zbytečně prázdná nezanedbatelná část obrazovky. Wireframe znázorňující tento návrh je na obrázku



Obrázek 2.10 Wireframe editování atributů uzlu

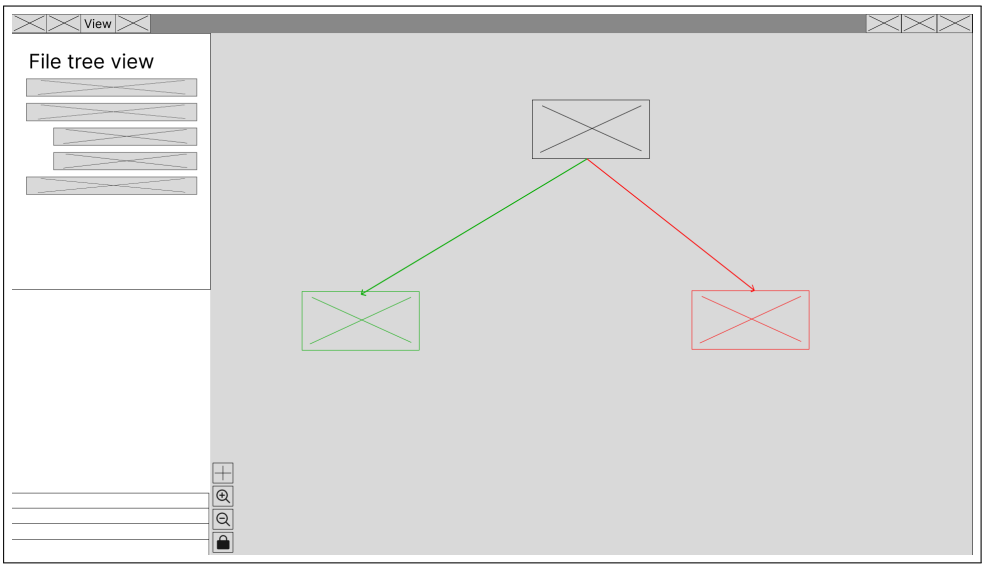
Jednou z „could have“ funkcionalit je vyhledávání ve stromě. Budeme například chtít najít uzel, který se nějak jmenuje. Také můžeme chtít najít uzel, ve kterém se používá nějaká proměnná, pokud vůbec. Tyto nalezené uzly bychom chtěli nějak elegantně zvýraznit. Následující obrázek 2.11 je jedním možným návrhem této funkcionality.



Obrázek 2.11 Wireframe vyhledávání uzlu ve stromě

Poslední wireframe, který bych zde ještě rád ukázal, je návrh funkcionality zobrazování rozdílů mezi jednotlivými soubory. Tato funkcionalita by značně pomohla s revidováním změn během

vývoje. Porovnávat změny mezi verzemi stromu v textové podobě v gitovém uložišti je v nejlepším případě nepraktické. Častokrát v práci otevíráme v našem stávajícím editoru novou verzi a tu porovnáváme s textem zobrazeným v gitu. Tato vizualizace změn by mnohonásobně ulehčila proces „code review“ a ušetřila vývojářům čas i nervy. Možný návrh takového funkcionality je ilustrován na obrázku 2.12.



Obrázek 2.12 Návrh vizualizace rozdílů mezi verzemi stromu. Přidané změny jsou zvýrazněny zeleně, odebrané červeně

Kapitola 3
Implementace

Tato kapitola slouží jako dokumentace procesu, kterým si aplikace prošla během vývoje. Trošku blíže si popíšeme strukturu zdrojového kódu aplikace. Vysvětlíme si zde některé problémy, na které jsem během vývoje aplikace narazil. Také si ukážeme a do detailu popíšeme něčím zajímavé části zdrojového kódu.

3.1 Vývojový proces

Začátek vývoje nové aplikace není nikdy lehký. Pro mě byl začátek práce na tomto projektu ale obzvlášť složitý, jelikož jsem se zvolenými technologiemi měl z praktického hlediska nulové zkušenosti. Taky to byl ale jedním z důvodů, proč jsem si právě tyto technologie vybral. Myslím si, že tento projekt byla dobrá příležitost naučit se nové technologie, i když to bylo místy celkem obtížné.

Celý proces jsem započal zjišťováním, jak vůbec nový projekt nakonfigurovat a založit. Dozvěděl jsem se, že preferovaný způsob vývoje Electronových aplikací, aspoň pro začátečníky, je použití nějaké šablony, která za vás domovský adresář s potřebnými soubory a nastaveními vytvoří. Osobně jsem k tomuto účelu použil šablonu od vývojáře Yukun Guo na vytvoření šablony Electronového projektu založeném na TypeScriptu a Vue.js [29].

3.2 Vue.js do hloubky

Dalším krokem v procesu vyvíjení této aplikace bylo delší pročítání oficiální dokumentace na webových stránkách knihovny Vue.js. Musel jsem se seznámit s tím, jak webové stránky fungují. Jak se strukturuje webová stránka v základním rozložení pomocí HTML. Také jsem se musel naučit používat jazyk CSS ke stylování komponent. Pak už přišlo na řadu se seznámit se všemi různými funkcemi knihovny Vue.js. A že jich není málo. Aplikace vytvořená pomocí Vue.js začíná v kořenové komponentě, která se většinou nazývá „App“. Tato, jako každá jiná komponenta ve Vue se implementuje jako soubor s příponou **.vue**. Každý tento soubor se skládá ze tří částí, které jsou obalené klasickými HTML tagy `<template>`, `<script>` a `<style>`. Do template části se zasazují jednotlivé komponenty či základní HTML tagy. Je to efektivně kostra komponenty. Script část obsahuje funkcionalitu komponenty. Jsou zde definované proměnné, metody, eventy a další. Ve style sekci se pak pomocí CSS definuje vzhled komponenty.

Další důležitou vlastností komponent jsou takzvané **props**. Props jsou efektivně proměnné, které může rodičovská komponenta nastavit a na které může daná komponenta reagovat či číst jejich data. Jednou z důležitých funkcionalit je to, že tyto proměnné mohou být i reference a tudíž může komponenta dynamicky měnit svůj obsah v závislosti na datech poslaných rodičem. Pokud

bychom chtěli, aby naše komponenta kromě změny obsahu definovaným těmito daty prováděla při změně těchto dat i jiné věci, můžeme použít takzvaný **watch**. Watch je funkce, které předáme nějakou referenci a funkci, která se má zavolat při změně dat, na které reference odkazuje. Tato funkcionalita je velmi užitečná pro reagování na změny stavu a umožňuje mnoho kódu specifického k dané komponentě umístit přímo do ní.

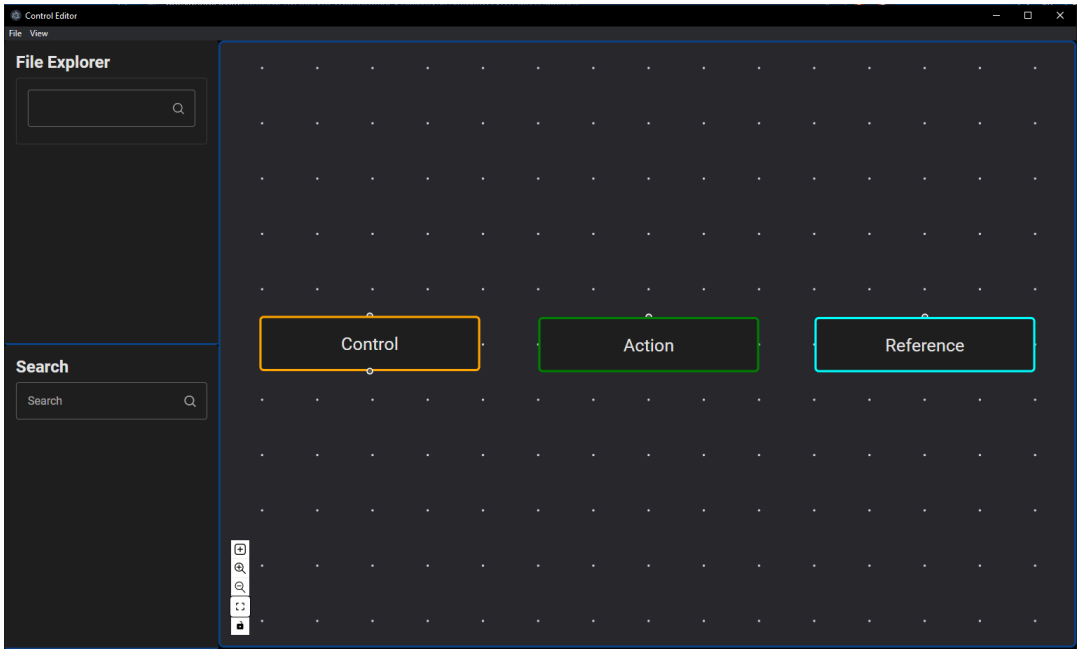
Vue.js také podporuje systém eventů. Každá komponenta si může definovat libovolný počet eventů, které může během vyhodnocování některé ze svých funkcí vyvolat. Na tyto eventy potom může rodičovská komponenta čekat a když je některý z nich vyvolán, reagovat na tuto událost. Ve svém kódu jsem tyto vlastní eventy, až na pár výjimek, moc nevyužil. Komponenty z knihoven, které jsem použil, většinou obsahovaly všechny potřebné eventy a tak nebylo potřeba přidávat vlastní.

Poslední důležitou funkcionalitou, kterou Vue poskytuje, jsou **sloty**. Sloty nám umožňují předat komponentě z rodiče nějaký obsah do `<template>` tagu. Zjednodušeně řečeno může rodič předat potomkovi kostru, kterou má vyrenderovat. Tato funkcionalita se nesmírně hodí, když chceme vytvořit nějakou univerzální komponentu s vlastní sadou funkcí, ale chceme, aby pořád byla přizpůsobitelná zvenku. Můžeme jí totiž pomocí slotů předat její „vnitřnosti“. Hlavním případem využití slotů v této práci je definice vlastních ulzů v knihovně VueFlow. Příklad využití této funkcionality můžeme vidět v 3.1.

```
<VueFlow>
    <template #node-reference="referenceNodeProps">
        <ReferenceNode v-bind="referenceNodeProps"/>
    </template>
    <template #node-action="actionNodeProps">
        <ActionNode v-bind="actionNodeProps" />
    </template>
    <template #node-control="controlNodeProps">
        <ControlNode v-bind="controlNodeProps" />
    </template>
<VueFlow/>
```

Výpis kódu 3.1 Příklad použití slotů k definování vlastních uzlů v komponentě VueFlow

V úryvku kódu výše jsou **ReferenceNode**, **ActionNode** a **ControlNode** naše vlastní komponenty, které jsme v rámci tohoto projektu vytvořili. Stačí si pomocí klíčového slova **import** importovat komponentu a Vue pro ni automaticky vytvoří nový tag. Pro lepší představu vzhledu aplikace vizte obrázek 3.1 níže.



Obrázek 3.1 Vzhled každého typu uzlu v plátně

3.3 Problémy během vývoje

Vývoj této aplikace se samozřejmě neobešel bez komplikací. Jakožto nováček ve vývoji právě s těmito technologiemi jsem během vývoje aplikace dělal mnoho chyb. Díky popularity těchto frameworků bylo ale více než dost podpůrného materiálu, ze kterého se daly čerpat informace a ladit tak ne úplně perfektní přístup k programování efektivně webové aplikace.

3.3.1 Volání Electronových funkcí z renderovacího procesu

Prvním problémem, se kterým jsem se potkal, byla potřeba volat nativní funkce electronu. V tomto konkrétním případě to bylo API pro otevírání souborů přes nativní Windows API. Je to takové to klasické okno, které se zobrazí v jakékoliv aplikaci, která vás vyzve k vybrání souboru na operačních systémech Windows. Prvně jsem se snažil rozhraní **dialog** z electronu naimportovat přímo ve Vue komponentě. To se ale mimo hlavní proces nepodaří, jelikož renderovací proces nemá k funkcionalitě electronu přístup. Po neúspěchu v tomto přístupu jsem se dozvěděl o různých procesech a jakým způsobem tohoto docílit. Prvně je potřeba vytvořit skript s názvem **preload.ts**. Tento skript, jak je již popsáno v 2.2 definuje API mezi hlavním a render procesem. Ve zdrojovém kódu této aplikace tohle API vytváříme nějak takto:

```
contextBridge.exposeInMainWorld('electronAPI', {
  foo: (callback: any) =>
    ipcRenderer.on('channel-name', (_event, value) => callback(value)),
})
```

Výpis kódu 3.2 Příklad definování funkce na rozhraní mezi hlavním a renderovacím procesem

Tímto kódem jsme efektivně provedli následující: Na rozhraní mezi hlavím a renderovacím procesem jsme definovali funkci se jménem **foo**. Tato funkce přijímá jinou funkci jako parametr.

Voláni `ipcRenderer.on()` zaregistruje event-listener na kanálu se jménem „channel-name“. Když na kanál přijde nová zpráva, zavolá námi předanou funkci, neboli funkci z renderovacího vlákna. Toto nám umožňuje reagovat v renderovacím procesu na události, které se stanou v hlavním procesu. V komponentě aplikace poté můžeme následujícím voláním zaregistrovat takovou funkci, která by se měla při nějaké události v hlavním procesu zavolat:

```
onBeforeMount(() => {
    window.electronAPI.foo(bar);
  })
```

Výpis kódu 3.3 Příklad zaregistrování funkce na event v hlavním procesu

Někdy ale potřebujeme přesně opačnou funkcionalitu. Tudíž bychom potřebovali reagovat v hlavním procesu na události, které se stanou v procesu renderovacím. Tohoto docílíme efektivně opačným přístupem. V API definujeme funkci v souboru **preload.ts**, ale tentokrát se bude jednat o volání `ipcRenderer.send()`. To nám pošle zprávu do hlavního procesu, spolu s jakýmikoliv daty, které předáme volané funkci. Tato registrace je ilustrována na příkladu 3.4:

```
contextBridge.exposeInMainWorld('electronAPI', {
  foo: (data: any) => ipcRenderer.send('channel-name', data),
})
```

Výpis kódu 3.4 Příklad zaregistrování funkce na event v renderovacím procesu

Pak už nám stačí zavolat funkci `window.electronAPI.foo(data)` v renderovacím procesu s nějakými daty a do hlavního procesu se pošle zpráva přes kanál s názvem „channel-name“ a s našimi daty. Hlavní proces už na tuto zprávu pak může reagovat pomocí zaregistrování zpětného volání nějaké funkce **func** zavoláním `ipcMain.on('channel-name', func)`.

3.3.2 Stav aplikace přístupný odkudkoliv

K zamyšlení je také způsob sdílení dat mezi různými komponentami. Ze začátku jsem všechny data propagoval od rodičovské komponenty reprezentující celou aplikaci směrem k potomkům. Narazil jsem ale na problém, když jsem v rámci referenčního uzlu potřeboval zjistit, jaké soubory aktuálně otevřený projekt obsahuje a jakou orientaci strom má. V tento moment jsem si řekl, že by se mi hodil nějaký „centrální sklad“ popisující stav aplikace v daný moment. Nejen na webu se hodí mít nějaké globální uložiště dat, na které se dá dotazovat z jakékoliv části kódu. Naše aplikace na tom není jinak. Způsobů, jak docílit nějakého globálního stavu přístupného v celé kódové základně je mnoho. Nejpopulárnější volbou mezi Vue vývojáři je knihovna Pinia. Pinia umožňuje zapisování a čtení dat odkudkoliv z renderovacího procesu. Tímto se nám nesmírně zjednodušil kód pro referenční uzel. Stačí se totiž dotázat stavu spravovaného Piniou na potřebné informace.

Globální stav inicializujeme pomocí funkce `defineStore()` importované z knihovny Pinia. Tomuto stavu můžeme dát jméno, tudíž můžeme stavů zadefinovat kolik potřebujeme. Dále ve stavu můžeme definovat proměnné a metody, které mohou při zavolání mutovat stavové proměnné. Tento stav se poté exportuje jako „composable“, neboli funkce která si udržuje nějaký svůj vlastní stav. Tato specifická vlastnost Vue funguje pak následovně:

```
// appState.ts je soubor, kde definujeme stavové uložiště a exportujeme jej
// jako composable funkci
import { useAppStateStore } from 'appState';
// stav aplikace si uložíme do proměnné
const state = useAppStateStore();
// s tímto stavem pak pracujeme jako s normálním objektem v JavaScriptu
const orientation = state.treeOrientation;
```

Výpis kódu 3.5 Příklad dotazování se globálního stavu na orientaci stromů v aplikaci

3.3.3 Eventy v komponentě, která není „focused“

S tímto problémem jsem se potýkal více než jednou. Eventuálně jsem si řekl, že musí existovat lepší cesta. Prvky DOMu totiž nejsou schopny reagovat na eventy, pokud není prvek během jeho vyvolání „focused“ neboli aktivně zvolen. To způsobovalo problémy při implementaci nějakých klávesových zkratek při najetí na uzel. Najetí totiž nestačí ke zvolení prvku DOMu. Je potřeba na něj kliknout. Tento problém jsem vyřešil použitím knihovny VueUse. Ta má totiž k dispozici senzor `onKeyStroke()`. Využití této funkce můžeme vidět na příkladu 3.6. Funkce předaná jako druhý argument se pak provede při každém stisknutí klávesy korespondující znaku v prvním argumentu, nehledě na ostatní faktory. Tohohle jsem využil k implementaci mazání uzlů pomocí klávesy **Delete**, aniž bych na uzly musel předem klikat, čímž jsem uživateli ušetřil při mazání každého uzlu jedno kliknutí.

```
onKeyStroke('A', (e) => { console.log('Key A pressed') });
```

Výpis kódu 3.6 Příklad zpracovávání vstupu z klávesnice

3.3.4 Stylování v rámci komponenty

Jak jsme již v sekci 3.2 zmiňovali, komponenty se dají stylovat vložením CSS kódu mezi `<style>` tagy. Tento CSS kód píšeme jako v klasickém **.css** souboru když vyvíjíme webovou aplikaci klasickým způsobem. Definujeme třídy či identifikátory a pak pro ně nastavujeme styly, tak jak jsme zvyklí. Zde se ale setkáváme s lehce neintuitivním návrhem stylů ve Vue.js. Jakékoliv styly, které nastavíme v dané komponentě, jsou potom efektivní globálně skrz celou aplikaci. Když ale definujeme styly v rámci nějaké komponenty, dá se očekávat že budeme chtít, aby se tyto styly vztahovaly jen k té dané komponentě. Tento problém má dvě řešení. Prvním řešením je do tagu přidat klíčové slovo **scoped**. Naše komponenta by pak obsahovala následující ukázku:

```
<style scoped>
.node {
    background-color: red;
}
</style>
```

Výpis kódu 3.7 Příklad lokálních stylů v komponentě pomocí „scoped“

Z [22] ale můžeme zjistit, že tento přístup je mnohonásobně pomalejší, než naše druhá možnost, což je využívání tříd a identifikátorů. Tuto možnost jsem se rozhodl použít pro vyřešení stylování v této aplikaci. Každá komponenta, která potřebuje mít nějaké specifické stylování, obsahuje jeden `<div>` tag, který dostane nějakou unikátní třídu (například jméno komponenty).

Poté můžeme upravovat styly tím, že před každý selektor v CSS přidáme selektor na třídu s tímto pojmenováním a máme jistotu, že všechny naše styly budou aplikované jen na aktuální třídu. Pro ilustraci vizte výpis kódu 3.8.

```
<template>
  <div class="myComponent">
    <p>Toto je odstavec</p>
  </div>
</template>

...

<style scoped>
.myComponent p {
    font-weight: 'bold';
}
</style>
```

Výpis kódu 3.8 Příklad tučných odstavců v komponentě pomocí CSS tříd

3.3.5 Reprezentace projektu pomocí rozložení TreeView

Jako poslední bych v této části rád vypíchl, jak jsem vyřešil reprezentaci BT projektu v bočním panelu aplikace pomocí rozložení **TreeView**. Formát BT projektu, neboli souboru s příponou **.btset**, je JSON obsahující na každém řádku řetězec s cestou k souboru relativně k projektu. Rozdělením na třídy ekvivalence podle prefixů těchto řetězců dostaneme strukturu podobnou adresářové struktuře. V následující ukázce kódu vizte funkci, která na vstupu bere pole řetězců obsahující cesty ke souborům a jako výstup vrací pole uzlů reprezentující adresářovou strukturu tohoto projektu. Kód je pro pohodlí čtenáře ještě blíže okomentován.

```
const createTree = (fileNames) =>
{
  // Seznam stringů rozdělíme do tříd ekvivalence podle prefixů
  const equivalenceClasses = fileNames.reduce((classes, fileName) => {
    // Vezmeme si prefix
    const prefix = fileName.split('\\')[0];
    // Pokud už v seznamu prefix neexistuje, vytvoříme pro něj prázdné pole
    if(!classes[prefix])
      classes[prefix] = [];
    // Odžízneme od názvu souboru prefix
    fileName = fileName.substring(fileName.indexOf('\\') + 1)
    // Uložíme si zbývající string do třídy jeho prefixu
    classes[prefix].push(fileName);
    // Vrátíme seznam tříd abychom v další iteraci mohli pokračovat
    return classes;
  }, {});

  let nodes = <TreeNode>[];

  // Pro každou třídu ekvivalence vytvoříme podstrom
  for (let name in equivalenceClasses)
  {
    if(Object.prototype.hasOwnProperty.call(equivalenceClasses, name))
    {
      // Všechny položky už jsou jen soubory, vytvoř pro ně uzly
      if(equivalenceClasses[name].length <= 1 && equivalenceClasses[name] == name)
        nodes.push({key: counter++, label: name, selectable: true});
      else
        nodes.push({
        key: counter++,
        label: name,
        selectable: true,
        children: createTree(equivalenceClasses[name])
        });
    }
  }

  return nodes;
}
```

Výpis kódu 3.9 Rozložení projektového souboru do stromové struktury

Pro ilustraci jak finální prohlížeč projektů vypadá v aplikaci vizte obrázek 3.2 níže.

Obrázek 3.2 Vzhled aplikace s otevřeným projektem

Kapitola 4
Testování

Uživatelské testování je nedílnou součástí vývoje softwaru. Dává nám důležitou zpětnou vazbu od lidí, kteří o aplikaci zatím nic neví a tak nám dávají vhled do myšlení reálných uživatelů. V této kapitole si sepíšeme nějaké testovací scénáře založené na případech užití. Provedeme testování na uživatelích, kteří zastupují námi definované uživatelské persony. Tyto výsledky poté shrneme a vyvodíme z nich závěry o námi provedené implementaci.

Uživatelské testování budeme provádět vždy s jedním uživatelem najednou. Uživateli budeme postupně dávat jednoduché úkoly, které v aplikaci bude muset provést. Nebudeme mu ale říkat, přesně kde a s čím interagovat. Našim cílem testování je zjistit, zdali je námi vyvinuté GUI intuitivní a jednoduché na používání. Proto chceme nějakou kontrolu nechat na uživateli. Po průchodu všemi testovacími scénáři ještě uživateli poskytneme krátký dotazník, kde nám bude moci poskytnout další zpětnou vazbu v sérii krátkých otázek.

4.1 Testovací scénáře

Abychom mohli s uživateli testovat funkcionalitu naší aplikace, potřebujeme si nejprve připravit nějaké úkoly, které necháme uživatele provádět. Budeme je během provádění těchto úkolů sledovat, ptát se je na jejich pocity a názory a zaznamenávat si, jak se jim plnění úkolů bude dařit. Každá podsekce bude reprezentovat jeden testovací scénář. Bude popisovat sekvenci kroků, kterou bude uživatel muset provést. Také bude obsahovat nějakou metriku, pomocí které budeme schopni ověřit, zda uživatel scénář splnil, prerekvizity pro scénář a jaké případy užití scénář pokrývá. Tyto testy budeme provádět z většiny na testovacím projektu, který je přiložený v elektronické příloze.

4.1.1 Vytvoření nového stromu

Prerekvizity

Tento scénář nemá žádné prerekvizity

Testovací kroky

1. Spusťte aplikaci
2. Otevřete nový soubor

3. Přidejte uzel typu **Control** do stromu
4. Přidejte uzel typu **Action** nebo **Reference** do stromu
5. Připojte uzel **Control** jako předka druhého uzlu
6. Klikněte pravým na přidaný **Control** uzel a zvolte z menu další uzel

Kritéria splnění

- Uživatel má otevřený editor s novým souborem
- V plátně je několik uzlů, všechny jsou přidány uživatelem

4.1.2 Úprava stromu v rámci projektu

Prerekvizity

Uživatel má k dispozici testovací projekt

Testovací kroky

1. Spusťte aplikaci
2. Otevřete testovací projekt
3. Z projektu otevřete strom s názvem **edit.bt**
4. Z otevřeného souboru smažte libovolný list
5. Změňte orientaci stromu z vertikálního na horizontální
6. Uložte změny v souboru

Kritéria splnění

- Upravený soubor je uložen a změny v něm jsou viditelné

4.1.3 Vyhledávání ve stromě

Prerekvizity

Uživatel má k dispozici testovací projekt

Testovací kroky

1. Spusťte aplikaci
2. Otevřete testovací projekt
3. V testovacím projektu si otevřete strom s názvem **search.bt**
4. V tomto stromě vyhledejte a identifikujte uzel s názvem „Find me“
5. Tomuto uzlu změňte hodnotu v položce **Position**

Kritéria splnění

- Výše zmíněný uzel má změněnou hodnotu v políčku **Position**

4.1.4 Úprava a zobrazení reference

Prerekvizity

Uživatel má k dispozici testovací projekt

Testovací kroky

1. Spusťte aplikaci
2. Otevřete testovací projekt
3. V testovacím projektu si otevřete strom s názvem **reference.bt**
4. V tomto stromě zvolte a editujte libovolný **Reference** uzel
5. Vyberte libovolný jiný strom z projektu jako cíl reference uzlu
6. Zobrazte si strom, na který reference ukazuje

Kritéria splnění

- Referenční uzel je nastaven na nějaký jiný strom
- Zobrazený náhled stromu v referenci odpovídá realitě

4.2 Pokrytí případů užití testovacími scénáři

Zde bych ještě pro přehlednost rád uvedl tabulku, která znázorňuje které případy užití jsou pokryty kterými testovacími scénáři. To nám pomůže jednodušeji poznat, jestli jsme na žádné případy užití při testování nezapomněli.

Případ užití	Testovací scénář
UC1	Scénář 1
UC2	Scénář 2, 3, 4
UC3	Scénář 2
UC4	Scénář 2, 3, 4
UC5	Scénář 1
UC6	Scénář 3, 4
UC7	Scénář 4
UC8	
UC9	
UC10	Scénář 3
UC11	Scénář 2
UC12	
UC13	Scénář 1

Tabulka 4.1 Tabulka znázorňující pokrytí případů užití testovacími scénáři

Jak můžeme z tabulky vidět, některé případy užití nejsou pokryty žádným testovacím scénářem. Důvodem je, že tyto případy užití jsou propojeny s funkčními požadavky, které byly

buď **Could have** anebo **Should have** a tím pádem nebyly doimplementovány. Více o těchto požadavcích a případech užití doplníme v následující kapitole, specificky v sekci 4.5.

4.3 Testovací dotazník

V této sekci si uvedeme otázky, které jsou součástí dotazníku poskytovaném uživatelům po ukončení uživatelského testování. Odpovědi zaznamenané v rámci dotazníku budou k dispozici jako tabulka v příloze. Dotazník bude napsaný v angličtině, jelikož budou součástí testování i lidé, kteří neumí česky. Pro pohodlnost jsou zde otázky z dotazníku uvedené v českém překladu. Dotazník obsahuje následující otázky:

- Prosím zvolte možnost, která pro vás dává největší smysl: "Preferuji ovládání tohoto prototypu více než aktuální verze Control Editoru."
  - Souhlasím
  - Částečně souhlasím
  - Ani jedno
  - Částečně nesouhlasím
  - Nesouhlasím
- Prosím zvolte možnost, která pro vás dává největší smysl: "Preferuji návrh/vzhled tohoto prototypu více než aktuální verze Control Editoru."
  - Souhlasím
  - Částečně souhlasím
  - Ani jedno
  - Částečně nesouhlasím
  - Nesouhlasím
- Co se vám líbí na návrhu a ovládání tohoto prototypu?
- Co se vám nelíbí na návrhu a ovládání tohoto prototypu?
- Stalo se vám že jste nebyli schopni najít způsob jak provést úkol, který vám byl zadán? Pokud ano, stručně prosím popište tuto situaci.
- Setkal(a) jste se s nějakými chybami či bugy v prototypu? Pokud ano, stručně prosím popište tuto situaci.
- Máte nějaké jiné připomínky?

4.4 Výsledky testování

Pro účely testování jsem využil ochoty kolegů z praxe. Tři ze čtyř vývojářů, které jsem požádal, jsou přímo vývojáři behaviorů pro NPC. Poslední je také vývojářem v BISim, ale nezaměřuje se na vývoj behaviorů. Požádal jsem také kamaráda, aby zastupoval uživatelskou personu modovacího nadšence, který je zručný v práci s počítačem, ale nikdy nevyvíjel hru ani behaviorální stromy pro ni. Tímto máme pokryté všechny uživatelské persony definované v 1.4.4.

Následující tabulka nám ilustruje, jak se testerům dařilo scénáře plnit pouze s přístupem k testovacímu scénáři, který jim říkal co mají dělat, ale ne jak to udělat. Když nebyli schopni na něco přijít, nechal jsem je se pokoušet dokud si sami neřekli o radu.

Uživatelé	Scénář 1	Scénář 2	Scénář 3	Scénář 4
Uživatel 1	Bez pomoci	Bez pomoci	S pomocí	Bez pomoci
Uživatel 2	Bez pomoci	S pomocí	S pomocí	Bez pomoci
Uživatel 3	Bez pomoci	Bez pomoci	S pomocí	S pomocí
Uživatel 4	Bez pomoci	Bez pomoci	Bez pomoci	Bez pomoci
Uživatel 5	Bez pomoci	Bez pomoci	S pomocí	Bez pomoci

Tabulka 4.2 Tabulka popisující splnění scénářů uživateli

Třetí scénář ve většině případů vyžadoval mou pomoc, protože upravování uzlů je schované za klávesovou zkratkou „E“. Od většiny uživatelů jsem během testování dostal zpětnou vazbu že by dávalo větší smysl tuto interakci mít v nějakém menu při pravém kliknutí na uzel. V tomto menu by pak mohl být náznak, že klávesová zkratka existuje a jaká to je. Také bylo během uživatelského testování nalezeno několik bugů, které budou detailněji popsány v příloze obsahující odpovědi na testovací dotazník.

Pokud pomineme tento jeden případ s editací uzlů, můžeme z výsledků uživatelského testování vyvodit, že námi navržené a implementované GUI je uživatelsky přívětivé a intuitivní, neboť většina uživatelů neměla větší problém splnit většinu úkolů. Uživatelé také podle odpovědí v dotazníku preferovali vzhled a návrh naší aplikace oproti aktuální verzi Control Editoru. Také jsem v dotazníku obdržel odpovědi chválící způsob přidávání uzlů, možnost měnit orientaci stromu z vertikálního na horizontální a ušetření místa na obrazovce pro samotný strom schováním některých prvků uživatelského rozhraní. V rámci tohoto uživatelského testování bylo nalezeno menší množství bugů, které jsou detailněji popsané v odpovědích na uživatelský dotazník dostupný v C. Tyto bugy byly spíše vizuálního typu a nejedná se o chyby, které by totálně znemožnily fungování aplikace. Díky tomu by tyto chyby by mohly být vysoko na seznamu možností dalšího vývoje.

Během tohoto testovacího sezení také přirozeně vzniklo nemalé množství nových nápadů na požadavky. Velký podíl těchto požadavků bylo nepřekvapivě o funkcionalitu, na kterou je každý ve všech moderních editorech jakéhokoliv stylu v dnešní době zvyklý. Jedná se o funkcionalitu typu kopírování a vkládání, možnost vrátit poslední provedenou akci nebo možnost si okna přeskládat podle sebe. Také bylo vytknuto, že chyběly známé zkratky pro běžné akce, jako je například „Ctrl + S“ pro uložení. Cílem tohoto prototypu ale bylo prozkoumat nové možnosti a zdržovat se implementací známých funkcionalit mi přišlo jako odbíhání od cíle této práce. Chtěl bych zde ale vypsat několik nápadů na požadavky, které smysl dávají a nejsou úplně zažité:

- Způsob jak zobrazit všechny parametry všech uzlů pro snazší orientaci ve funkcionalitě stromu.
- Možnost přepnout hrany z plynulých křivek na rovné čáry s pravými úhly.
- Nějaký ještě jednodušší formát, který by byl lehčí na čtení v textové podobě.

Pro detailnější vhled do odpovědí z uživatelského dotazníku vizte přílohu C.

4.5 Další možnosti vývoje

Aplikace vyvinutá v rámci této bakalářské práce má v porovnání s existujícími řešeními pouze velmi základní funkcionalitu. Účel této práce bylo prozkoumání nových prvků UI, které v těchto řešeních doposud neexistují, nebo existující funkcionality vylepšit či zpříjemnit. Očividným prvním krokem k rozšiřování této aplikace jsou funkční požadavky uvedené v 1.4, které jsou označené jako **Could have** nebo **Should have** a nebyly v rámci této bakalářské práce vyvinuté. Pro příklad uvádím:

- Zobrazování rozdílů ve stromech
- Zobrazování průchodu stromu
- Vytvoření souboru z podstromu
- Náhled kde nový uzel vznikne během přidávání
- Automatické zarovnávání do mřížky

Další poznatky, které jsem dostal od kolegů mezi řečí, ale nestály za převedení do funkčních požadavků pro tuto bakalářskou práci:

- Světlý režim
- Možnost mít více stromů otevřených najednou
- Možnost vytvořit nový projekt
- API pro připojení k běžící instanci hry a možnost živého debugování

Posledním zdrojem inspirace mohou být odpovědi testerů v dotazníku k uživatelskému testování, které je stručně shrnuté v 4.4 a detailněji k dispozici v příloze C. Věřím že i jen některé z těchto funkcionalit by ze stávajícího prototypu udělaly plnohodnotnou aplikaci, která bude stát za to zvážit při vývoji her či vlastních herních enginů.

Kapitola 5
Závěr

Tato bakalářská práce popisovala vývoj nového grafického nástroje pro vyvíjení behaviorálních stromů. Poznatky z této bakalářské práce budou použity pro vývoj nového nástroje u nás v práci. Takto byla práce i zamýšlena.

Nejprve jsme si vysvětlili co to behaviorální stromy jsou a k čemu se používají. Podívali jsme se na nějaké již existující řešení, většinou úzce spjatých s herním enginem, pro který byly vyvinuty. Podívali jsme se na to, jaký formát ukládání stromů používáme u nás v Bohemia Interactive Simulations. Sepsali jsme si funkční požadavky, které byly z většiny kladeny buďto mnou jakožto vývojářem stromů, nebo mými kolegy v práci. Pro tyto funkční požadavky jsme pak vytvořili případy užití, které jsme později použili pro vytváření testovacích scénářů.

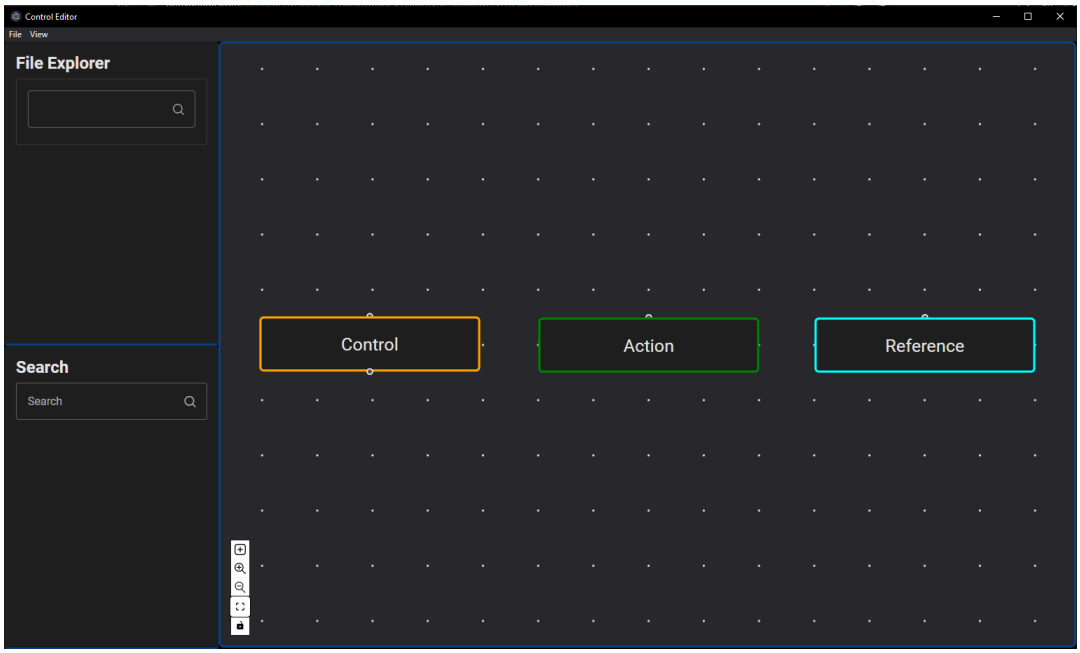
Následně jsme si v kapitole o návrhu popsali několik možností technologického stacku. Vypsali jsme si pro každou možnost její pro a proti a na základě tohoto jsme vybrali vyhovující technologie. Na těchto technologiích jsme navrhli architekturu aplikace, kde komunikaci s operačním systémem řeší Electron a GUI aplikace je postavené na TypeScriptu a Vue.js. Pro komponenty, ze kterých se skládá uživatelské rozhraní jsme použili knihovnu komponent PrimeVue, jež obsahuje nespočet komponent dodržujících zásady „material designu“. Pro vykreslování samotných stromů jsme použili knihovnu VueFlow, která se stará o většinu základních funkcionalit souvisejících se zobrazováním a upravováním stromů. Na základě funkčních požadavků a mých zkušeností s vývojem stromů a softwaru jako takového jsem pak navrhl jednoduché wireframy ilustrující, jak by rozložení aplikace mohlo vypadat.

V kapitole o implementaci jsme trochu detailněji popsali, jak vypadal proces vývoje nového protoypu. Dále jsme si vysvětlili některé základní koncepty knihovny Vue.js, abychom si mohli později vysvětlit některé problémy a zajímavosti, na které jsem v průběhu vývoje narazil. Tyto zajímavosti a problémy jsme si pak popsali do hloubky, ukázali jsme si příslušné kusy kódu pro ilustraci a zakončili jsme funkcí na rozdělení adresářové struktury do formátu TreeView.

V neposlední řadě jsme se v rámci testování zamysleli nad tím, jakým způsobem chceme naši aplikaci otestovat. Vymysleli jsme testovací scénáře, které jsme nechali uživatele projít abychom zjistili, jestli je naše aplikaci intuitivní a pohodlná na používání. Uživatelům jsme poté dali dotazník, aby nám mohli sdělit zpětnou vazbu. Výsledky testování jsme poté shrnuli a navrhli další možnosti vývoje.

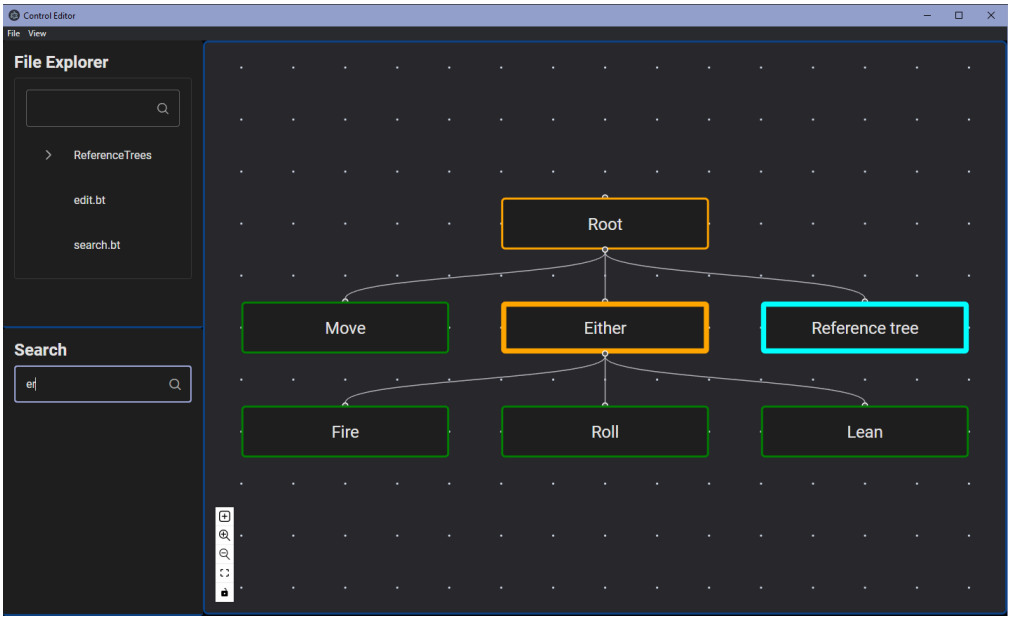
Příloha A

Přehled ukázek aplikace

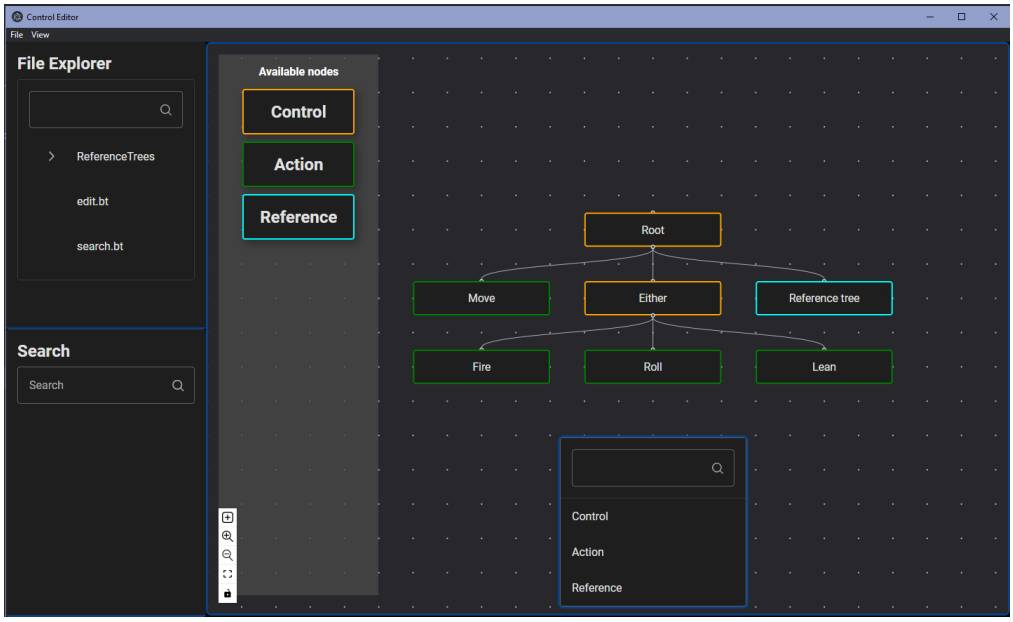


Obrázek A.1 Ukázka základního vzhledu aplikace

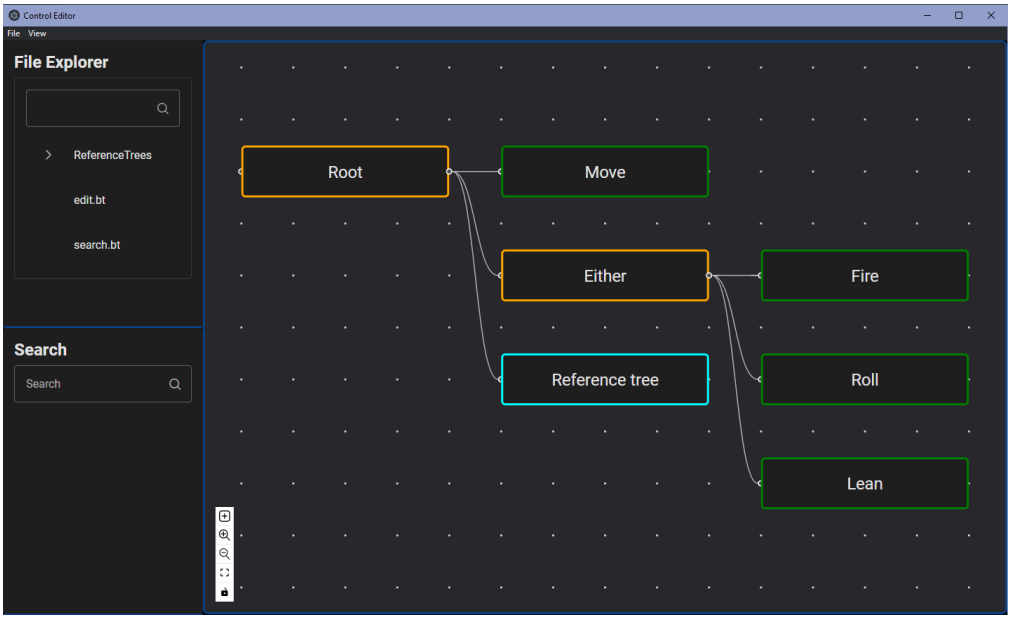
Obrázek A.2 Ukázka rozložení adresářové struktury v komponentě File Explorer



Obrázek A.3 Ukázka vzhledu aplikace při vyhledání uzlu



Obrázek A.4 Ukázka obou možností, jak přidat uzel do stromu



Obrázek A.5 Ukázka horizontální orientace stromu

Příloha B

Uživatelský manuál

Tato příloha obsahuje uživatelský manuál popisující jak aplikaci spustit, její rozhraní a ovládací prvky včetně klávesových zkratek. Také popisuje všechny funkcionality, které aplikace obsahuje a jak je využívat. Tento manuál byl vygenerován z Google Docs souboru.

Control Editor
User Manual

Michael Moyal
moyalmic@cvut.cz

Installation and startup

The application will come in a folder unpacked and built for the Windows operating system. All you need to do is open the folder and launch the **behavior-tree-editor.exe** file.

Application layout

On the left sidebar there are two separate windows. The **File Explorer** will display the hierarchical structure of a project once opened. The **Search** section can be used to search for nodes in the currently opened behavior tree. These nodes will be highlighted in the tree view itself.

Then there is the large empty panel on the right side. This is where the behavior tree will be displayed once a file is opened or a new one is created.

Finally at the top left of the screen there are two menu options, just like you are used to in most Windows applications. The **File** menu contains 4 options:

- New File, which creates an empty behavior tree
- Open File, which allows the user to select an existing file to be opened
- Save File, which saves the current state of the behavior tree being edited into a **.bt** file
- Open Project, which allows the user to select a behavior tree project to be opened

The **View** menu currently only contains one function which is to toggle the orientation of the trees from horizontal to vertical and vice versa.

Editing a behavior tree

Once a tree is opened, the first thing you can notice is a small white panel. The panel contains 5 buttons, going from top to bottom:

- The **Add Node** button - Pulls up a toolbox containing nodes that you can drag and drop into the canvas. Close this menu by hitting the Escape key.
- The **Zoom in** button - Makes the view of the tree zoom in. This can also be achieved using the mouse scroll wheel.
- The **Zoom out** button - Makes the view of the tree zoom out. Similarly to zooming in, this can also be done using the mouse wheel.
- The **Fit view** button - Adjusts the view to fix all of the nodes in the canvas. This function has a limit because of the minimum zoom limitation.
- The **Lock** button - Locks the nodes and edges on the canvas so that they cannot be moved without unlocking.

Adding nodes is also possible by pressing tab or the right mouse button, which pulls up a quick add menu.

Deleting nodes is possible by pressing delete when hovering the mouse over the selected node.

Editing nodes is possible by left clicking a node to select it, then pressing E. The different node types have different attributes you can edit:

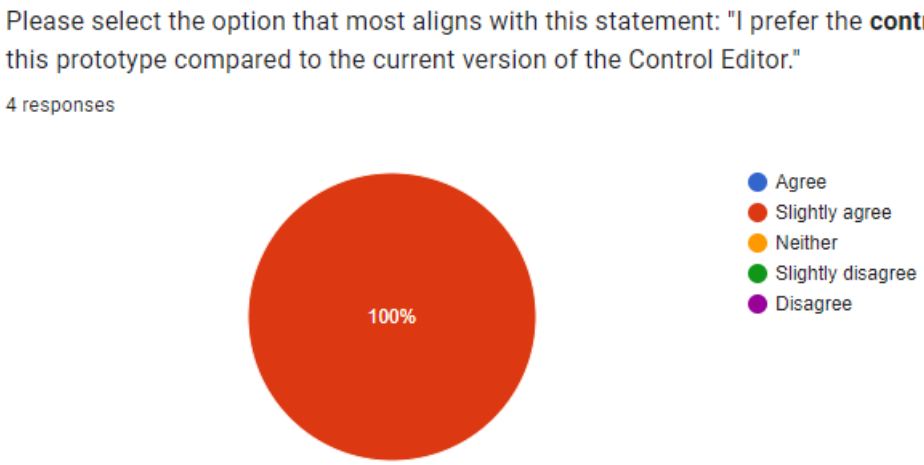
- Reference nodes have a dropdown for you to select when a project is actively open
- Action nodes have two text fields, position and heading

Edges can be **removed** by clicking on them.

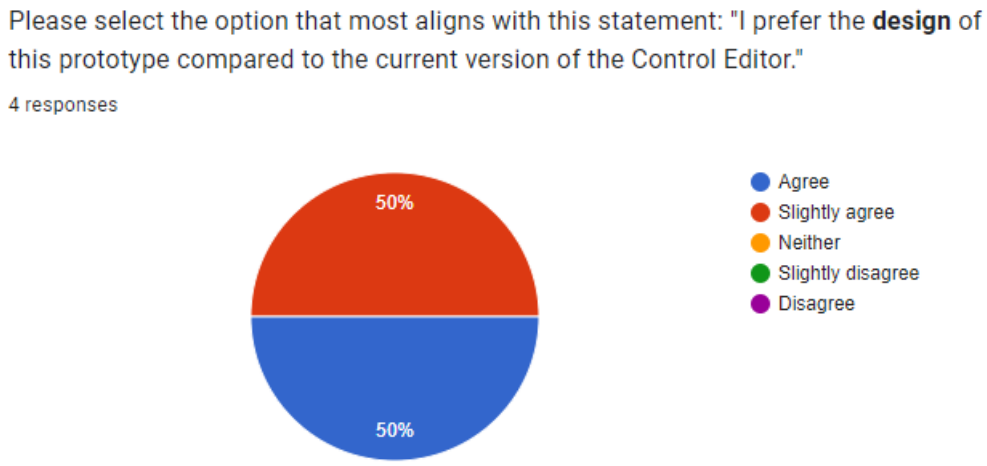
When adding a node by right-clicking another control node, the added node will be automatically placed as the child of the right-clicked node.

Příloha C

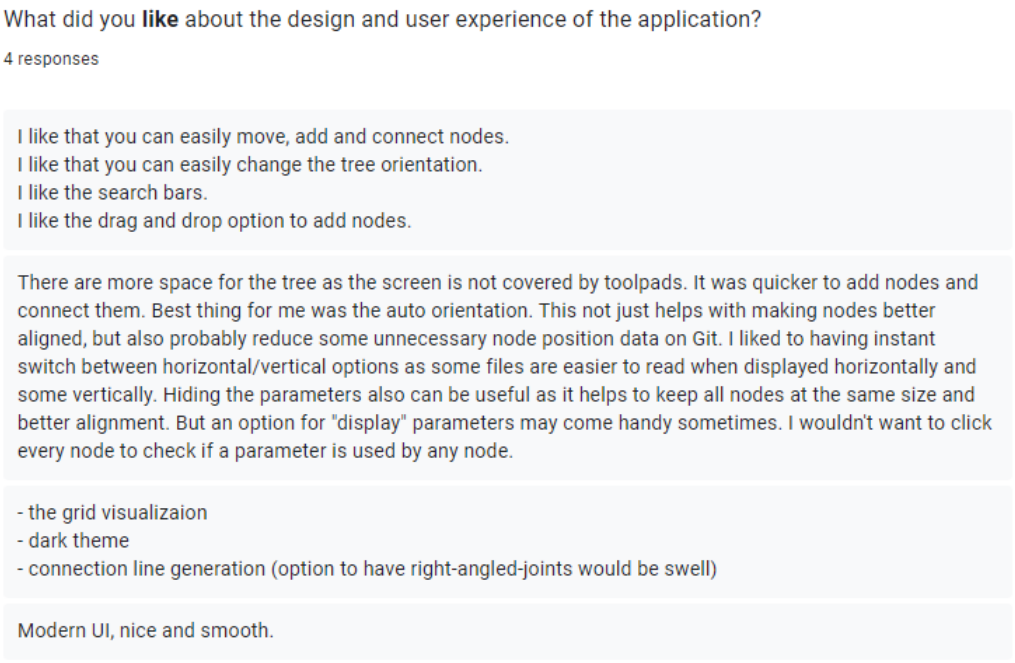
Odpovědi z testovacího dotazníku



Obrázek C.1 Přehled odpovědí na otázku č. 1 v dotazníku



Obrázek C.2 Přehled odpovědí na otázku č. 2 v dotazníku



Obrázek C.3 Přehled odpovědí na otázku č. 3 v dotazníku

What did you **dislike** about the design and user experience of the application?

4 responses

Found nodes are not easy to see. Having a thicker border might not be easily spotted by everybody. Especially when there are multiple hits.

- using mouse-over to change deletion (+ maybe other) behavior is terrible, it should be operating on selection somehow
- the search highlighting is not visible enough - maybe the fill color?
- even new unsaved files should appear in the file list
- if I want to save a file I'm editing I would expect it to overwrite itself automatically, not by me having to do it manually
- more options in context (right-click) menus
-- the context menu disappears inconsistently? sometimes it disappears, sometimes it doesnt, idk
-- E to edit vs. context menu
- when I unraveld a referenced tree, I would like to be able to move the whole panel to the side
- let me close the node toolbox by toggling with the same button with which I opened it
- undo would be nice

When clicking a cell to focus it, and the click happened in the area of the cell's name, it automatically started editing the name. Also deleting by right clicking and selecting delete was missing for me.

Obrázek C.4 Přehled odpovědí na otázku č. 4 v dotazníku

Was there ever a time where you weren't able to find what you needed to do? If so, please describe that situation shortly.

4 responses

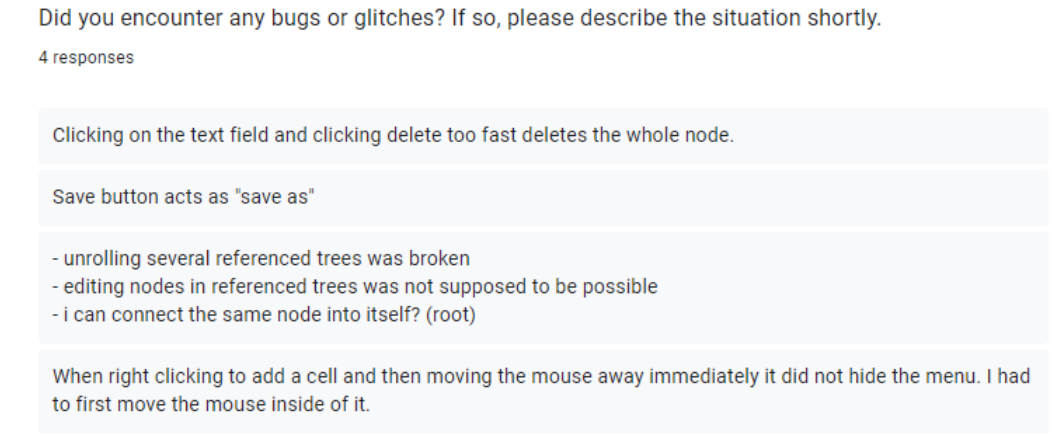
I did not know that you need to press 'e' to edit a node's properties

"E" button for Edit.

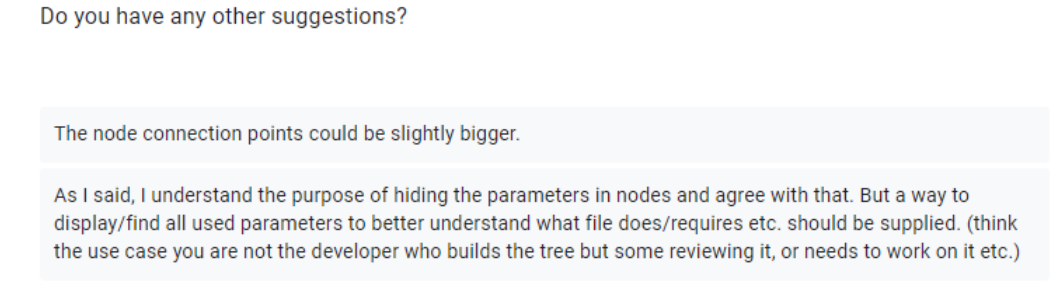
- the E to edit - I would have found it, but I wouldve disliked finding it

I had trouble with figuring out where to change the direction of the tree

Obrázek C.5 Přehled odpovědí na otázku č. 5 v dotazníku



Obrázek C.6 Přehled odpovědí na otázku č. 6 v dotazníku



Obrázek C.7 Přehled odpovědí na otázku č. 7 v dotazníku

Bibliografie

1. ISLA, Damian. Handling Complexity in the Halo 2 AI. *GDC 2005 Proceeding* [online]. 2005. Dostupné také z: https://www.gamedeveloper.com/programming/gdc-2005-proceeding-handling-complexity-in-the-i-halo-2-i-ai.
2. DELIGHT, Techie. *Preorder Tree Traversal – Iterative and Recursive* [online]. 2023. [cit. 2024-05-09]. Dostupné z: https://www.techiedelight.com/preorder-tree-traversal-iterative-recursive/.
3. COLLEDANCHISE, Michele; ÖGREN, Petter. Behavior Trees in Robotics and AI: An Introduction. *CoRR*. 2017, roč. abs/1709.00084. Dostupné z arXiv: 1709.00084.
4. CORMEN, Thoman H.; LEISERSON, Charles E.; RIVEST, Ronald L.; STEIN, Clifford. *Introduction to Algorithms, Fourth Edition*. MIT Press, 2022. ISBN 9780262046305.
5. GAMES, Epic. *Behavior Trees* [online]. 2024. [cit. 2024-05-13]. Dostupné z: https://docs.unrealengine.com/4.26/en-US/InteractiveExperiences/ArtificialIntelligence/BehaviorTrees/.
6. GAMERANX. *Why Was Crysis A Big Deal?* [online]. 2017. [cit. 2024-04-07]. Dostupné z: https://www.youtube.com/watch?v=y0Xpk2a9BZQ.
7. CRYTEK. *Legacy (GameSDK) AI* [online]. 2022. [cit. 2024-05-13]. Dostupné z: https://www.cryengine.com/tutorials/view/ai/legacy-gamesdk-ai.
8. WAWRZKO, Michal. *Beehave documentation* [online]. 2023. [cit. 2024-05-13]. Dostupné z: https://bitbra.in/beehave/#/.
9. CROCKFORD, Douglas; MORNINGSTAR, Chip. *Standard ECMA-404 The JSON Data Interchange Syntax*. 2017. Dostupné z DOI: 10.13140/RG.2.2.28181.14560.
10. BRAY, Tim; PAOLI, Jean; SPERBERG-MCQUEEN, C. M.; MALER, Eve; YERGEAU, François. *Extensible Markup Language (XML) 1.0 (Fifth Edition)* [W3C Recommendation]. 2008. Available at http://www.w3.org/TR/REC-xml/.
11. SOMMERVILLE, Ian. *Software Engineering*. 9. vyd. Harlow, England: Addison-Wesley, 2010. ISBN 978-0-13-703515-1.
12. MADSEN, Susanne. *How to Prioritize With the MoSCoW Method* [online]. 2023. Dostupné také z: https://www.projectmanager.com/training/prioritize-moscow-technique.
13. VEAL, Ravel. How to Define a User Persona. *CareerFoundry* [online]. 2023 [cit. 2024-05-03]. Dostupné z: https://careerfoundry.com/en/blog/ux-design/how-to-define-a-user-persona/#what-is-a-user-persona.
14. GEORGE, Andy De. *Průvodce pro desktop (WPF .NET)* [online]. 2023. [cit. 2024-04-16]. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/desktop/wpf/overview/?view=netdesktop-8.0.

15. COMPANY, The Qt. *Qt | Development Framework for Cross-Platform applications* [online]. 2024. [cit. 2024-04-16]. Dostupné z: https://www.qt.io/product/framework.
16. BRITCH, David. *Co je .NET MAUI?* [Online]. 2024. [cit. 2024-04-16]. Dostupné z: https://learn.microsoft.com/cs-cz/dotnet/maui/what-is-maui?view=net-maui-8.0.
17. PRICE, Greg. *Flutter Documentation* [online]. 2024. [cit. 2024-04-16]. Dostupné z: https://docs.flutter.dev/.
18. AMADEO, Ron. *Google starts a push for cross-platform app development with Flutter SDK* [online]. 2018. [cit. 2024-04-16]. Dostupné z: https://arstechnica.com/gadgets/2018/02/google-starts-a-push-for-cross-platform-app-development-with-flutter-sdk/.
19. GOOGLE. *Showcase - Flutter apps in production* [online]. 2024. [cit. 2024-04-16]. Dostupné z: https://flutter.dev/showcase.
20. FOUNDATION, OpenJS. *What is Electron?* [Online]. 2024. [cit. 2024-04-20]. Dostupné z: https://www.electronjs.org/docs/latest/.
21. MICROSOFT. *TypeScript is JavaScript with syntax for types.* [Online]. 2024. [cit. 2024-04-20]. Dostupné z: https://www.typescriptlang.org/.
22. YOU, Evan. *What is Vue?* [Online]. 2024. [cit. 2024-04-20]. Dostupné z: https://vuejs.org/guide/introduction.html.
23. PRIMETEK. *Introduction - PrimeVue* [online]. 2024. [cit. 2024-04-21]. Dostupné z: https://primevue.org/introduction/.
24. CAKMAKOGLU, Burak. *Introduction | Vue Flow* [online]. 2024. [cit. 2024-04-21]. Dostupné z: https://vueflow.dev/guide/.
25. FU, Anthony. *Get Started | VueUse* [online]. 2024. [cit. 2024-04-21]. Dostupné z: https://vueuse.org/guide/.
26. MOROTE, Eduardo San Martin. *Introduction | Pinia* [online]. 2024. [cit. 2024-04-21]. Dostupné z: https://pinia.vuejs.org/introduction.html#Introduction.
27. FOUNDATION, OpenJS. *Process Model | Electron* [online]. 2024. [cit. 2024-04-20]. Dostupné z: https://www.electronjs.org/docs/latest/tutorial/process-model.
28. GILLIN, Paul. What is Component-Based Architecture? *Mendix* [online]. 2024 [cit. 2024-04-25]. Dostupné z: https://www.mendix.com/blog/what-is-component-based-architecture/.
29. GUO, Yukun. *vite-vue3-electron-ts-template* [online]. GitHub, 2023 [cit. 2024-04-25]. Dostupné z: https://github.com/Yukun-Guo/vite-vue3-electron-ts-template.

Obsah příloh

- `attach` — adresář obsahující ostatní přílohy
  - `examples` — adresář obsahující screenshoty z aplikace
  - `test-project` — adresář obsahující testovací projekt
  - `testing-questionnaire-answers.csv` — odpovědi v dotazníku k testování
  - `user-manual.pdf` — uživatelská příručka
- `exe` — adresář se spustitelnou formou implementace
- `src`
  - `impl` — zdrojové kódy implementace
  - `thesis` — zdrojová forma práce ve formátu LaTeX
- `text` — text práce
  - `thesis.pdf` — text práce ve formátu PDF
- `readme.txt` — stručný popis obsahu média