VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA STROJNÍHO INŽENÝRSTVÍ

FACULTY OF MECHANICAL ENGINEERING

ÚSTAV AUTOMATIZACE A INFORMATIKY

INSTITUTE OF AUTOMATION AND COMPUTER SCIENCE

APLIKACE HLEDÁNÍ CESTY V POČÍTAČOVÉ HŘE

PATHFINDING APPLICATION IN COMPUTER GAME

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

AUTOR PRÁCE — Miroslav Tihlařík

AUTHOR

VEDOUCÍ PRÁCE — Ing. Ladislav Dobrovský

SUPERVISOR

BRNO 2020

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ FAKULTA STROJNÍHO INŽENÝRSTVÍ

Zadání bakalářské práce

Ústav:	Ústav automatizace a informatiky
Student:	**Miroslav Tihlařík**
Studijní program:	Strojírenství
Studijní obor:	Aplikovaná informatika a řízení
Vedoucí práce:	**Ing. Ladislav Dobrovský**
Akademický rok:	2019/20

Ředitel ústavu Vám v souladu se zákonem č.111/1998 o vysokých školách a se Studijním a zkušebním řádem VUT v Brně určuje následující téma bakalářské práce:

Aplikace hledání cesty v počítačové hře

Stručná charakteristika problematiky úkolu:

Hledání cesty v prostředí je velice důležitým problémem v reálných i virtuálních prostředích. Počítačové hry představují vhodné virtuální prostředí pro vývoj a testování algoritmů pro řešení tohoto problému.

Cíle bakalářské práce:

Porovnání stávajících algoritmů pro hledání cesty a možnosti jejich kombinace v závislosti na rozsáhlosti prostředí.
Volba a implementace vhodného algoritmu pro volená prostředí.
Integrace algoritmu do virtuálního prostředí počítačové hry.

Seznam doporučené literatury:

CUI, Xiao; SHI, Hao. A*-based Pathfinding in Modern Computer Games, 2010, 11.

BOTEA, Adi; MÜLLER, Martin; SCHAEFFER, Jonathan. Near optimal hierarchical path-finding. Journal of game development, 2004, 1.1: 7-28.

Termín odevzdání bakalářské práce je stanoven časovým plánem akademického roku 2019/20

V Brně, dne

L. S.

doc. Ing. Radomil Matoušek, Ph.D. ředitel ústavu	doc. Ing. Jaroslav Katolický, Ph.D. děkan fakulty

ABSTRAKT

Tato práce se zabývá popisem a porovnáním různých mapových sítí a algoritmů vyhledávání cest používaných v prostředí videoher.

ABSTRACT

This thesis deals with the description and comparison of various map grids and pathfindng algorithms used in the video game environment.

KLÍČOVÁ SLOVA

Mapová síť, plánování cesty, video hra

KEYWORDS

Map grid, pathfinding, video game

BIBLIOGRAFICKÁ CITACE

TIHLAŘÍK, Miroslav. *Aplikace hledání cesty v počítačové hře.* Brno, 2020. Dostupné také z: https://www.vutbr.cz/studenti/zav-prace/detail/125435. Bakalářská práce. Vysoké učení technické v Brně, Fakulta strojního inženýrství, Ústav automatizace a informatiky. Vedoucí práce Ladislav Dobrovský.

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu práce Ing. Ladislavu Dobrovskému za trpělivost, a cenné rady a připomínky ke tvorbě této bakalářské práce.

ČESTNÉ PROHLÁŠENÍ

Prohlašuji, že tato práce je mým původním dílem, zpracoval jsem ji samostatně pod vedením Ing. Ladislava Dobrovského a s použitím literatury uvedené v seznamu literatury.

V Brně dne 22. 5. 2020 ………………………………………………

Miroslav Tihlařík

OBSAH

1 Úvod ... 15

2 Cíl práce ... 17

3 Mapové sítě ... 18

3.1 Pravidelné mřížky ... 18
3.1.1 Čtvercové sítě ... 18
3.1.2 Trojúhelníkové sítě ... 20
3.1.3 Hexagonální sítě ... 20

3.2 Nepravidelné mřížky ... 21
3.2.1 Grafy viditelnosti ... 21
3.2.2 Navigační síť (Navigation mesh) ... 22

4 Algoritmy ... 23

4.1 Prohledávání do hloubky (Depth-first search) ... 23

4.2 Prohledávání do šířky (Breadth-first search) ... 24

4.3 Rychle prozkoumávající náhodný strom (RRT) ... 24

4.4 Dijkstrův algoritmus ... 25

4.5 A* algoritmus ... 27
4.5.1 Optimalizace algoritmu A* ... 28
4.5.1.1 Heuristické funkce ... 28
4.5.1.2 Optimalizace vyhledávacího prostoru ... 30
4.5.1.2.1 Navigační síť (NavMesh) ... 31
4.5.1.2.2 Hierarchické hledání cesty (HPA*) ... 32

5 Vlastní řešení ... 33

5.1 Vytvoření hexagonální mřížky ... 33

5.2 Úpravy a sousedé ... 39

5.3 Aplikace algoritmu ... 45

6 Závěr ... 50

7 Seznam použité literatury ... 51

8 Seznam obrázků ... 53

9 Přílohy ... 55

1 ÚVOD

Plánování cest za použití nejrůznějších algoritmů je problematika probíraná v mnoha oblastech od reálných map, přes robotiku, umělou inteligenci až po video hry. Za posledních několik desetiletí se technika vyhledávání zlepšila v přesnosti a efektivnosti, avšak stále přitahuje další výzkumníky, kteří se snaží tento navigační systém vylepšit. Díky nejrůznějším metodám, které dokážou reálné či virtuální mapy rozdělit na průchodné a neprůchodné plochy, je potřeba správně vybrat a integrovat algoritmy hledání cest, které se dále ještě dají optimalizovat a upravovat pro získání lepších výsledků.

Tato práce se soustřeďuje na videoherní prostředí, použitelné mapové sítě ve 2D, které jsou za použití technik skeletonizace a rozkládání upraveny k testování algoritmů pro vyhledávání cest a samotné grafové algoritmy, které mají vlastní výhody a nevýhody uplatnění. V herním prostředí je pohyb z jednoho místa na druhé základním a zásadním komponentem, který by se bez algoritmů, které tuto dráhu vypočítají, neobešel. Ve velkém množství případů se jedná o vyhledání cesty ze startovní pozice, na které se herní charakter vyskytuje, do cílového bodu, který uživatel vybere pro ať už taktické nebo pouze pro rychlý pohyb mezi body. Uživatel tudíž vyžaduje vyhledání cesty rychle a efektivně s tím, že samotná dráha mezi body musí být nejrychlejší a nejbezpečnější, a tak se musí počítat s dalšími proměnnými parametry, které jsou různé u většiny herních titulů.

2 CÍL PRÁCE

Cílem této práce je zhodnotit a porovnat nejrozšířenější mapové sítě a základní používané algoritmy vyhledávání cest, používané ve videoherním prostředí ať už pro vlastní implementaci do her nebo pro výzkumné a testovací úmysly. Proto se tato práce věnuje několika bodům, mezi které patří rešeršní část zabývající se mapovými sítěmi a technikami pro jejich vytvoření. Dále se obrací na grafové algoritmy pro vyhledávání cest se zaměřením na algoritmus A*, optimalizaci jeho funkce a také vyhledávacího prostoru. Praktická část se zaměřuje na vytvoření herního prostředí, přesněji hexagonální mřížky, která je velice rozšířená mezi herními vývojáři a integraci algoritmu, který dokáže najít nejkratší cestu mezi startovním a cílovým bodem s tím, že nebude prohledávat části mapy, které jsou jasně nepoužitelné pro zadaný problém.

3 MAPOVÉ SÍTĚ

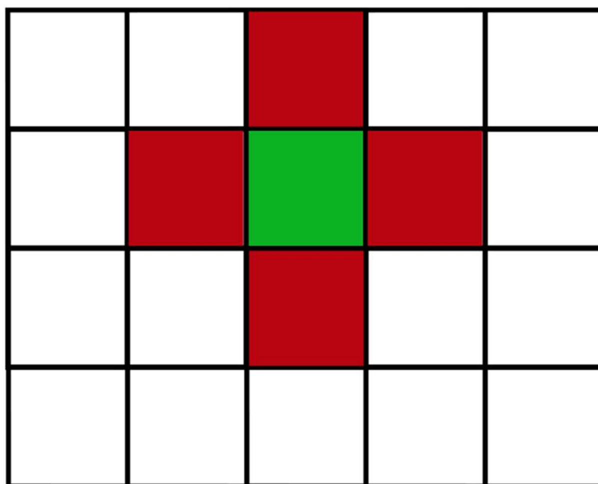
Mapové sítě, nebo také mřížky, jsou grafické reprezentace prostředí, sloužící pro aplikaci algoritmů vyhledávání cest. Účinnost a výkon algoritmů závisí na atributech mřížek. Dva nejpopulárnější koncepty mřížek obsahují mřížky pravidelné a nepravidelné. [8]

3.1 Pravidelné mřížky

Pravidelné mřížky jsou jedny z nejpoužívanějších a nejznámějších mřížek v herním prostředí, a také robotice. Pro pravidelná 2D prostředí se využívá čtverců, trojúhelníků nebo hexagonů k pravidelnému vyplnění prostředí. V následujících oddílech budou tyto hlavní metody popsány.

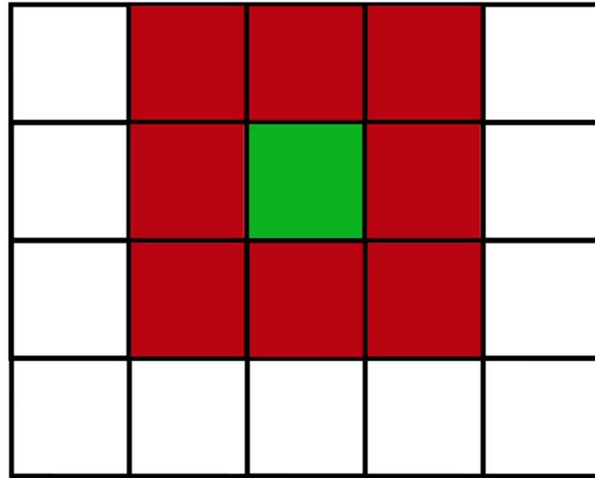
3.1.1 Čtvercové sítě

Velké množství herních prostředí umožňuje aplikaci čtvercové sítě k vyhledání optimální cesty k cíli, která umožňuje pohyb ve čtyřech základních směrech, znázorněno v obr. 1. Algoritmus musí najít optimální cestu ze startovní pozice a musí tedy brát v potaz čtyři sousední čtverce. Jelikož se při hledání optimální cesty algoritmus nikdy nevrací, v dalších krocích se k prohledání využijí pouze tři další, neprohledané čtverce.



Obr. 1: Čtvercová síť čtyři směry

U tohoto typu se dá také využít pohyb diagonální, který přidá další čtyři směry znázorněn v obr. 2.

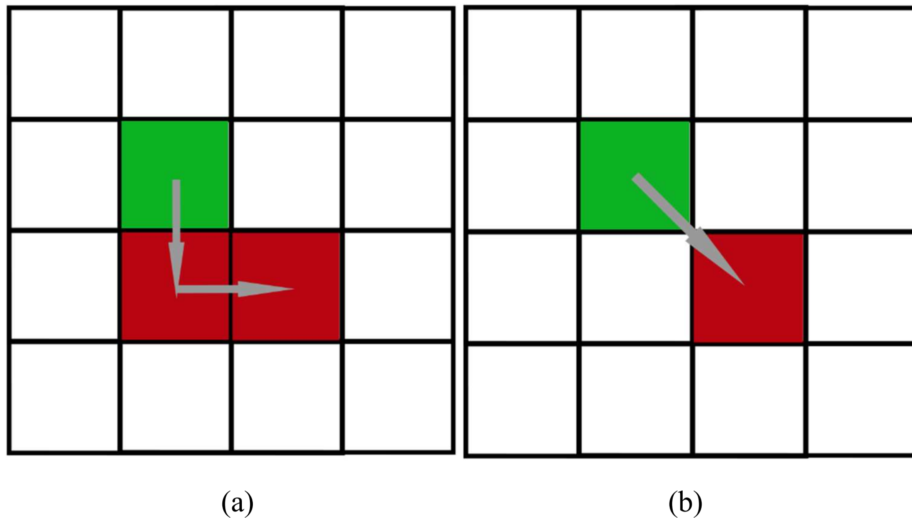


Obr. 2: Čtvercová síť osm směrů

Pokud uvažujeme cenu pohybu mezi středy čtverců, je zřejmé, že diagonální pohyb bude ve výsledku stát méně. Při dané ceně 1 by výsledná cesta v obr. 3a) při pravoúhlém pohybu stála 2. Při aplikaci základního vzorce pro výpočet diagonály:

$$|u| = a * \sqrt{2},$$

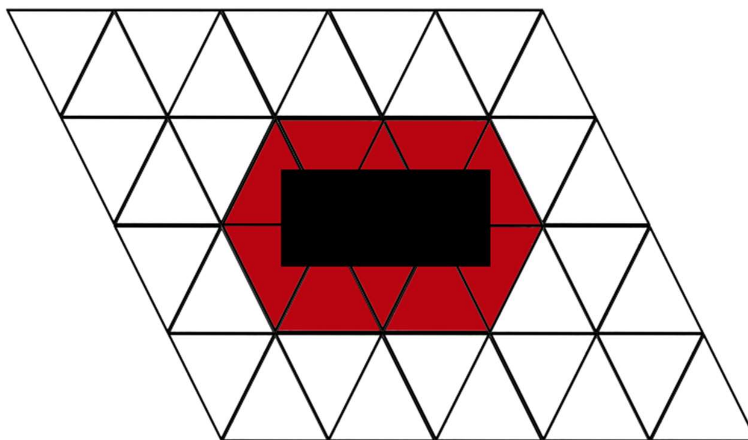
je tato cesta, znázorněná v obr. 3b), přibližně 1,414, čímž je zřejmé, že diagonální pohyb bude pro aplikaci vyhledávacích algoritmů výhodnější.



Obr. 3: Pravoúhlý pohyb (a), diagonální pohyb (b)

3.1.2 Trojúhelníkové sítě

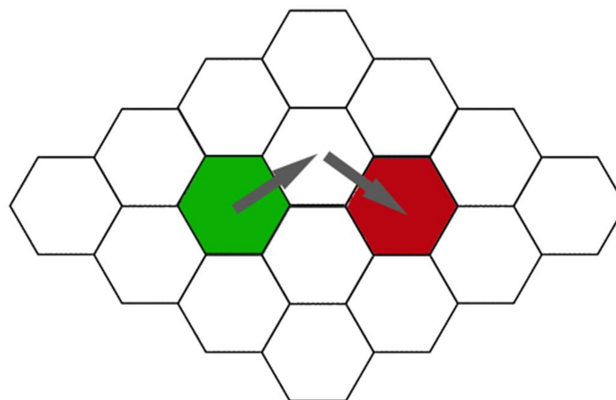
Trojúhelníkové mřížky se využívají jen v některých případech, kde je potřebné využití jejich vlastností. Damyen a Buro představili metodu [8], která snížila vyhledávací úsilí při použití omezené Delaunayovy triangulace. Změnou objektů zkoumali vliv na pohyb a bylo zjištěno, že jejich algoritmy TA* a TRA* fungují nejlépe na velkých mapách. [8] Na obr. 4 je zakreslena trojúhelníková síť s překážkou, které zamezuje využití překrytých, červených trojphelníků.



Obr. 4: Trojúhelníková síť s překážkou

3.1.3 Hexagonální sítě

Hexagonální sítě, které se využívají v herním prostředí hlavně ve strategických hrách, jsou využívané více než čtvercové. Výhodou oproti čtvercovým mřížkám je vzdálenost od středů polygonů, kde z hexagonu, který sousedí s dalšími šesti hexagony, je každá vzdálenost mezi sousedy stejná. Další výhodou je skutečnost, že sousedící hexagony sdílejí vždy své hrany, takže není mezi dvěma hexagony styk pouze v jednom bodě, jako je to například u diagonálního pohybu ve čtvercové síti. Nevýhodou oproti čtvercovým sítím je to, že pohyb se může uskutečnit pouze v šesti směrech. Při dané orientaci v obr. 5 je vidět, že směry na východ a západ jsou zcela nepřístupné, a proto charakter musí na daný hexagon přejít delší cestou.



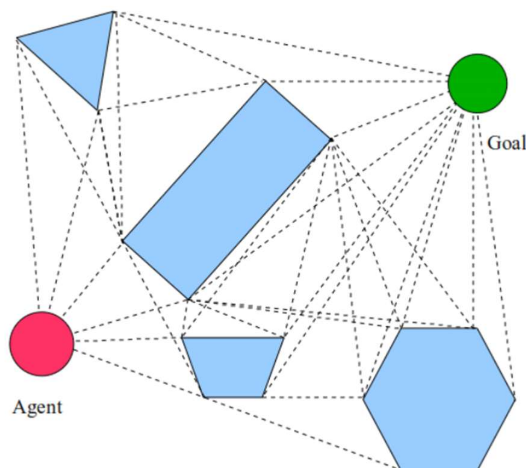
Obr. 5: Hexagonální mřížka

3.2 Nepravidelné mřížky

U nepravidelných mřížek se rovina nebo prostor nepravidelně vyplňují jednoduchými tvary. Tyto sítě na rozdíl od pravidelných vyžadují seznam konektivit, které specifikují, jak určité vektory vytvářejí jednotlivé elementy. Jako jednoduché tvary se využívají trojúhelníky, čtyřúhelníky i šestiúhelníky.

3.2.1 Grafy viditelnosti

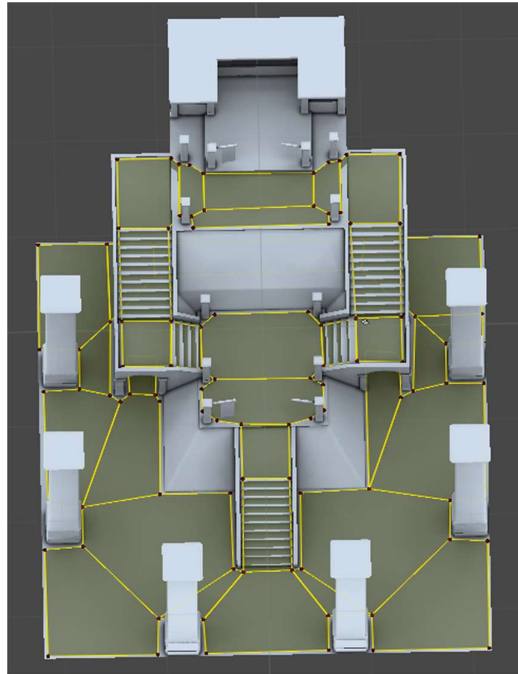
Graf viditelnosti tvoří graf viditelných míst pro body a překážky. Základním principem vytvoření takového grafu je spojení dvou uzlů úsečkou, které pokud není přerušena překážkou, je zakreslena jako hrana grafu, jako například na obr.6. Pokud zde překážka úsečku přeruší, spojení mezi body se může kolem překážky stočit. Grafy viditelnosti se využívají například při hledání nejkratší cesty v Euklidovském prostoru, kde dochází ke kombinaci vytvoření grafu a užití algoritmu hledání cesty.



Obr.6: Graf viditelnosti [10]

3.2.2 Navigační síť (Navigation mesh)

Navigační síť (Navigation mesh) je složena z konvexních polygonů, definujících oblast, která je volně průchozí, bez jakýchkoliv překážek. Sousední polygony jsou propojeny v grafu. Aplikace sítě je znázorněna na obr. 7



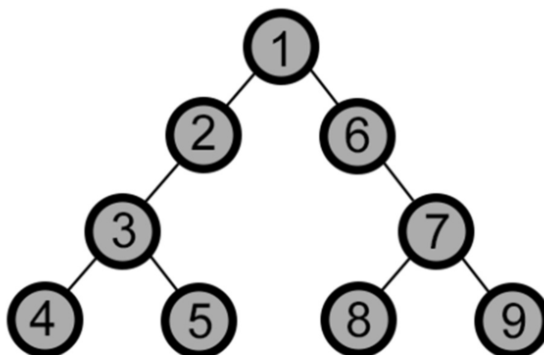
Obr. 7: Navigační síť v herním prostředí [11]

4 ALGORITMY

Algoritmy pro hledání cesty se nejběžněji využívají pro řešení problému nejkratší, nejlevnější nebo nejbezpečnější cesty jednoho bodu do druhého za využití teorie grafů. Jsou velice důležité, protože se hojně používají v reálném i digitálním světě. K nalezení cesty slouží předdefinovaná kritéria. Graf se skládá z uzlů nebo buněk, které jsou s dalšími propojeny hranami nebo linkami. Cesta mezi uzly se musí vyhodnotit a porovnat s dalšími tak, aby se našla preferovaná cesta, například nejkratší. Existuje několik typů algoritmů, které startovní bod s cílovým propojí. Jsou to ty, které prohledají všechny možné cesty a celou mapu před tím, než vyberou tu nejkratší a ty, které eliminují cesty nepoužitelné, nebo které budou jednoznačně delší než ostatní. [8]

4.1 Prohledávání do hloubky (Depth-first search)

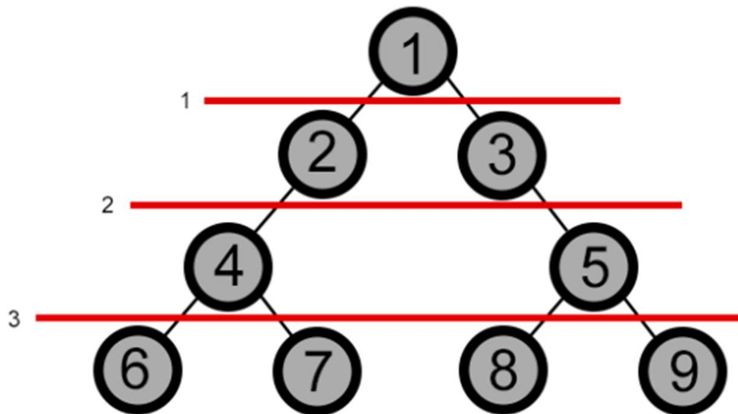
Prohledávání do hloubky je algoritmus, který prochází grafy s použitím metody backtracking, neboli zpětného vyhledávání. [6]



Obr. 8: Schéma prohledávání do hloubky

Obr. 8 naznačuje postup algoritmu, kde začíná na kořenovém uzlu a pokračuje po větvi až na konec. Po dosažení koncového bodu se vrací a vybírá novou cestu. Algoritmus takto pokračuje, dokud neprohledá všechny možnosti. [6] Tento algoritmus úplný, tudíž je schopen najít všechny vrcholy spojené se startovním bodem. Není ale optimální, což znamená, že pokud graf není strom, nemusí najít nejkratší cestu k cíli a pokud prochází nekonečnou větev, nikdy nemusí projít větev konečnou.

4.2 Prohledávání do šířky (Breadth-first search)

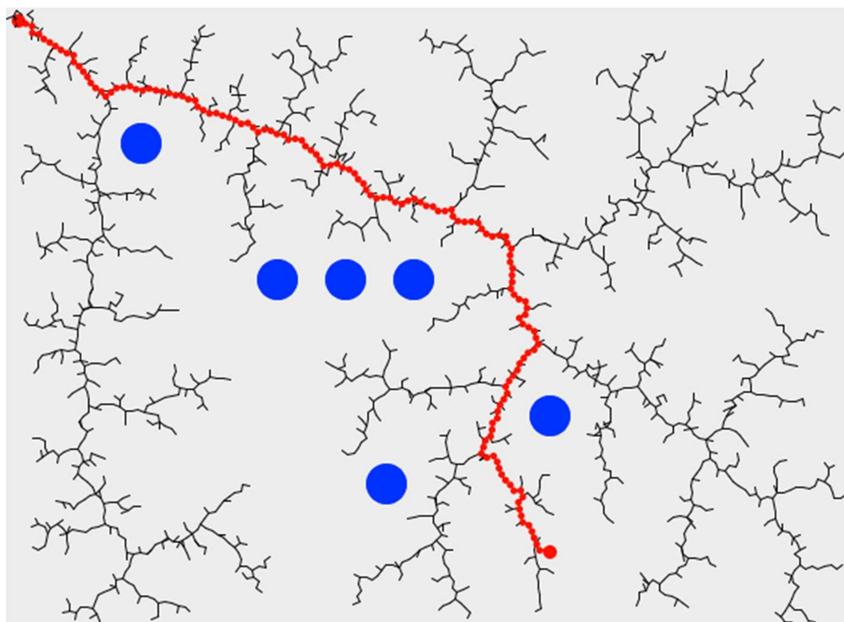


Obr. 9: Schéma prohledávání do šířky

Prohledávání do šířky je podobně jako prohledávání do hloubky grafový algoritmus, který prochází všechny možné sousedy startovního bodu a dále prochází sousedy sousedů, dokud neprojde všechny možné body. V obr. 9 je zakreslen postup algoritmu, kdy začíná na kořenovém uzlu a dále pokračuje po jednotlivých úrovních až na konec grafu. Je to systematické prohledávání grafu, při které se nepoužívá žádná heuristická analýza. Při procházení si zaznamenává předchůdce jednotlivých bodů, a tak je schopen vytvořit strom nejkratších cest k vrcholům od kořenového bodu. Využitelný je zejména pro nalezení dosažitelných vrcholů z daného počátečního vrcholu, zda je připojen další, nepřímý graf nebo nalezení nejkratší cesty z vrcholu počátečního ke všem ostatním dosažitelným. [7]

4.3 Rychle prozkoumávající náhodný strom (RRT)

Tento algoritmus byl navržen k prozkoumávání nekonvexních, velkých prostorů náhodným vytvářením stromu, který celý prostor postupně vyplňuje. Tento strom roste směrem k velkým, nevyhledaným oblastem, takže se primárně nezabývá malými prostory, které mohou být bezprostředně omezeny překážkami. Výhodou algoritmu RRT je využití nejen přímočarých pohybů, ale také pohybů s poloměrem zatáčky se závislostí na rychlosti. Jak již bylo zmíněno, prostor může obsahovat různé překážky a algoritmus RRT s nimi nemá problém, proto se hojně využívá v plánování pohybu robotů.

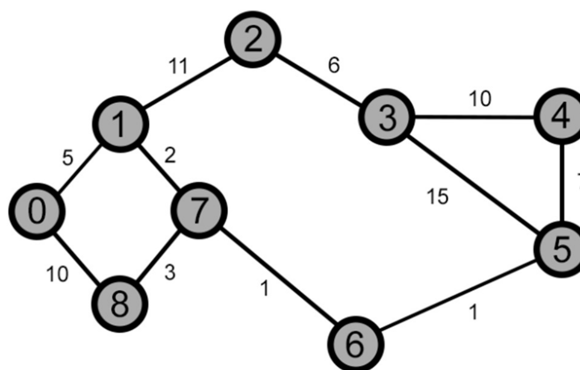


Obr. 10: Výsledek algoritmu RRT [12]

Na obr. 10 je vidět, že algoritmus RRT začíná vytvořením kořene stromu a roste pomocí náhodných vzorků vybraných z prostoru. Po nakreslení každého vzorku se pokusí o spojení s nejbližším stavem stromu, kdy spojení musí procházet volným prostorem při dodržování daných omezení. Takové omezení může být například délka spojení. Vzorek, který je od nejbližšího stavu stromu dále, než je dovoleno, nemůže být využit pro vytvoření spojení. Namísto tohoto bodu se využije stav v maximální vzdálenosti od stromu podél linie k náhodnému vzorku. [9]

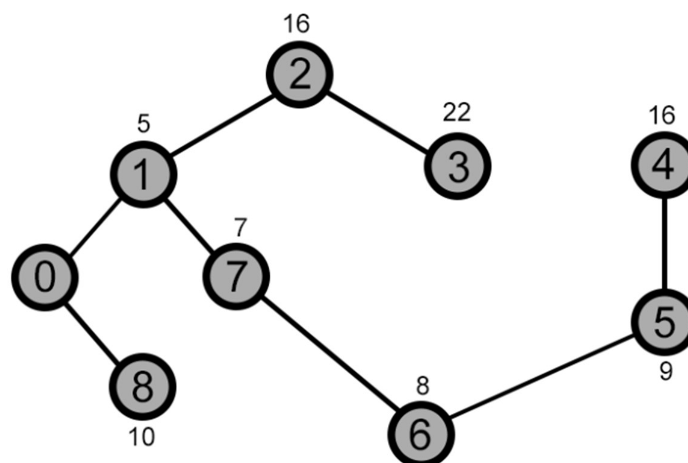
4.4 Dijkstrův algoritmus

Edsger Dijkstra, nizozemský informatik, představil v roce 1959 metodu pro nalezení nejkratší cesty mezi dvěma body v grafu, kde se přechází mezi uzly přes hrany. [3;4]



Obr. 11: Graf uzlů s cenami přechodů

Na obr. 11 je zaznačeno osm uzlů a spojnice mezi nimi, jež drží určitou přechodovou hodnotu. Algoritmus prohledává celý graf a zaznamenává ceny přechodů mezi body jako celkovou hodnotu vzdálenosti od startovního bodu. Při pohybu z jednoho uzlu do druhého se cena přechodu zapíše do celkové hodnoty. Jelikož může existovat i několik různých cest do cílového uzlu, musí se zvážit možnost kratší cesty. To znamená, že se spočítá hodnota cesty ze startovního bodu do současného a vybere se ta cesta, která drží hodnotu nejkratší. To nám zaručí, že cesta bude jednoznačně nejkratší.



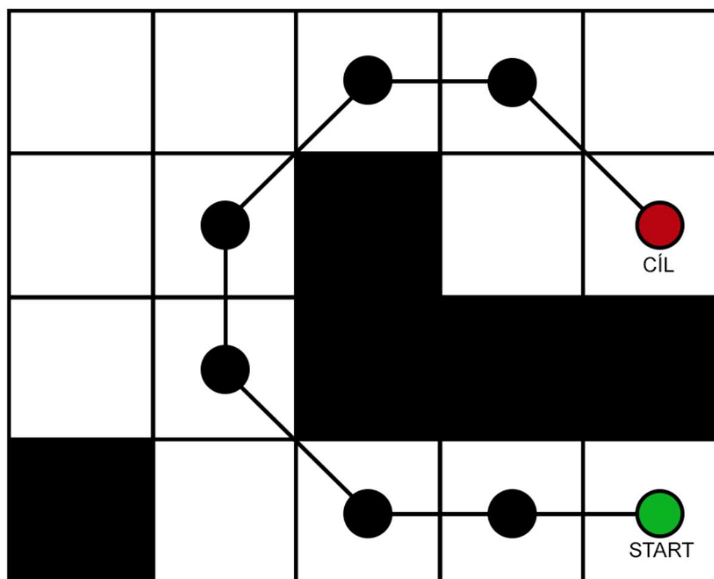
Obr. 12: Výsledné nejkratší cesty k uzlům

Výsledek příkladu z obr.10 můžeme vidět na obr. 12, kde z uzlu 0 vedou ke všem ostatním uzlům cesty díky Dijkstrově algoritmu nejkratší.

Při prohledávání Dijkstrův algoritmus narazí na body, které ještě nebyly nelezeny, které nebyly navštíveny a ty, které navštíveny byly. [4] Po objevení nového bodu se musí vypočítat cena pohybu ze startu a poté musí být osobně objeven. Tento bod je vložen do skupiny s dalšími body, které musí být objeveny. Tato skupina se nazývá otevřený list. Otevřený list tvoří seznam bodů, každý s vlastní cenovou hodnotou a vybere se ten, který má hodnotu nejmenší, to znamená, že je startu nejblíže. Vybraný bod je poté vložen do skupiny zvané zavřený list, ve kterém se nachází pouze body, které byly už jednou navštíveny. Pokud v otevřeném listu již nejsou žádné další body k objevení, algoritmus vypíše nejkratší možnou cestu ze startovního bodu do cílového. Tento algoritmus je kompletní i optimální. [5]

4.5 A* algoritmus

Peter Hart, Nils Nilson a Bertram Raphael představili v roce 1986 algoritmus A* jako variaci Dijkstrova algoritmu. [3] Na rozdíl od Dijkstrova algoritmu, A* zaručuje při použití přípustné a zároveň monotónní heuristiky, které směrují vyhledávání k cíli, namísto prohledávání celého grafu, nalezení nejkratší cesty.



Obr. 13: Nejkratší cesta při použití algoritmu A*

Obr. 13 vyznačuje 2D mřížku s překážkami a cestu z červeného zdroje do zeleného cíle, vypočítanou pomocí algoritmu A*

Tento algoritmus je populární nejen v herním prostředí ale také u práce s umělou inteligencí. A* pracuje s uspořádaným vyhledáváním, což znamená, že objevované území směřuje k cílovému bodu, dokud není nalezena první přijatelná, nejkratší cesta. Při objevování si také algoritmus pamatuje již prohledané uzly, a tak se vyvaruje opětovnému průchodu těmito uzly. A* používá cenu mezi uzly, stejně jako Dijkstra, a navíc používá další heuristickou hodnotu H, což je vzdálenost startovního bodu od cílového. Tato hodnota se sečte s cenou přechodu a vytvoří se tak hodnota nová, unikátní pro A* algoritmus

$$F(n) = C(n) + H(n), \tag{1}$$

kde F(n) představuje celkovou odhadovanou cenu, C(n) je cena přechodu mezi uzly a H(n) značí heuristický odhad. [5]

Dalším důležitým rozdílem od Dijkstrova algoritmu je vybírání dalšího bodu z otevřeného listu. A* totiž vybere bod, který má nejmenší hodnotu F, neboli bod který

je nejblíže předem určenému cíli, a také který je ve smyslu přechodu nejlevnější. V tomto případě se může stát, že dva uzly budou mít stejnou hodnotu F. Proto je dobré zakomponovat výběr lepšího uzlu například porovnáním hodnot ceny přechodu C. [5]

4.5.1 Optimalizace algoritmu A*

Následující pododdíly jsou zaměřeny na nejznámější optimalizační techniky algoritmu A*. Optimalizace vyhledávacích algoritmů se obecně využívají pro zrychlení nebo například na zvýšení efektivnosti algoritmů. V některých případech optimalizace také odebírá na výhodách, proto se musí technika optimalizace pečlivě vybrat, aby algoritmus dále fungoval lépe při určité funkci.

4.5.1.1 Heuristické funkce

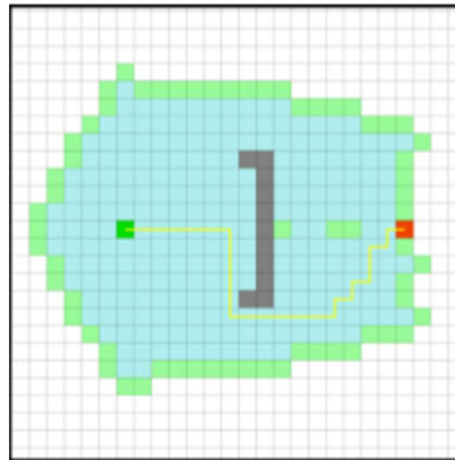
Heuristické funkce odhadují vzniklé náklady při cestování mezi dvěma uzly. Heuristika může změnit výkonnost vyhledávacího algoritmu, což jej zrychlí nebo zpomalí, použije více či méně paměti, bude více či méně přesný nebo optimální. [5]

V A* algoritmu se vybírá uzel podle hodnoty F(n) a tudíž záleží na důležitosti heuristického odhadu přes cenu přechodu. Vyskytne se nám hned několik možností v rozdílu těchto dvou hodnot. Pokud heuristický odhad H(n) je 0, hledání bude ovlivněno pouze cenou přechodu mezi uzly C(n). Když je H(n) menší nebo rovna C(n), algoritmus A* zaručeně najde nejkratší cestu. Když se H(n) rovná C(n), A* prochází pouze nejlepší cestu a nevěnuje se dalším uzlům, což zaručí rychlost vyhledávání. Pokud je H(n) v některých případech vyšší, vyhledávání bude rychlejší, ale nezaručí se optimální cesta. V posledním případě, kdy H(n) dosáhne hodnot o hodně vyšších, pouze hodnota heuristiky ovlivní vyhledávání, a to až do takové míry, že vyhledá cestu stejně jako by to udělal algoritmus uspořádaného prohledávání, který si vybere cestu nejslibnější. [5]

Heuristika je přípustná v takovém případě, že nikdy nepřecení náklady na dosažení cíle, což znamená, že odhadovaná cena musí být nižší nebo rovna skutečným nákladům pro dosažení cíle.

Pro algoritmus A* existuje několik známých heuristik. Euklidovská vzdálenost, obr. 14, přímá vzdálenost mezi body A a B, definována jako:

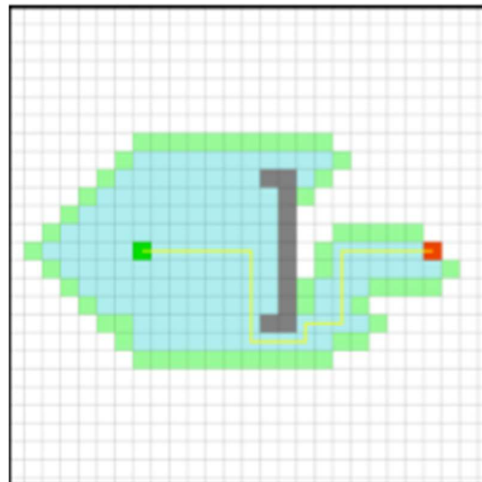
$$D_e(A,B) = \sqrt{\sum_{i=1}^{k}(A_i - B_i)^2}. \quad (2)$$



Obr. 14: Výsledek při zavedení Euklidovské heuristiky [5]

Manhattanská vzdálenost, obr.15, vzdálenost mezi dvěma body měřená podél os v pravém úhlu, definována jako:

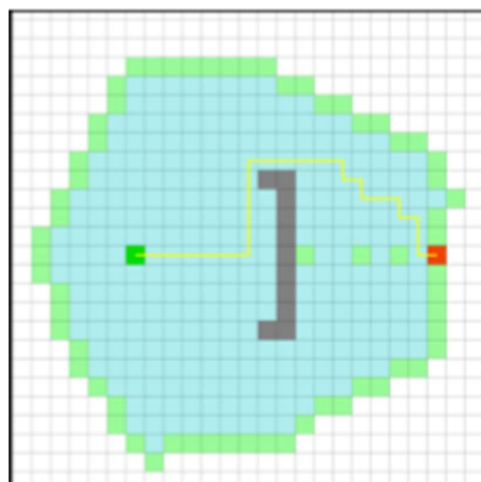
$$D_m(A,B) = \sum_{i=1}^{k} |A_i - B_i|. \tag{3}$$



Obr. 15: Výsledek při zavedení Manhattanské vzdálenosti [5]

Chebyshevova vzdálenost, obr. 16, největší rozdíl vzdálenosti mezi dvěma body nebo vektory v jedné dimenzi, definována jako:

$$D_{ch}(A,B) = max\left(|A_x - B_x|, \left|A_y - B_y\right|\right). \tag{4}$$



Obr. 16: Výsledek při zavedení Chebyschevovy vzdálenosti [5]

Obr. 14 nám ukazuje výslednou cestu algoritmu A* za použití Euklidovské vzdálenosti. Tato vzdálenost používá operaci odmocniny, která snižuje výkon a je lepší se jí vyhýbat, pokud je to možné. Heuristická hodnota bude v tomto případě menší než u Manhattanské nebo Chebyshevově vzdálenosti, což způsobí, že hodnota C(n) neodpovídá heuristice. [5]

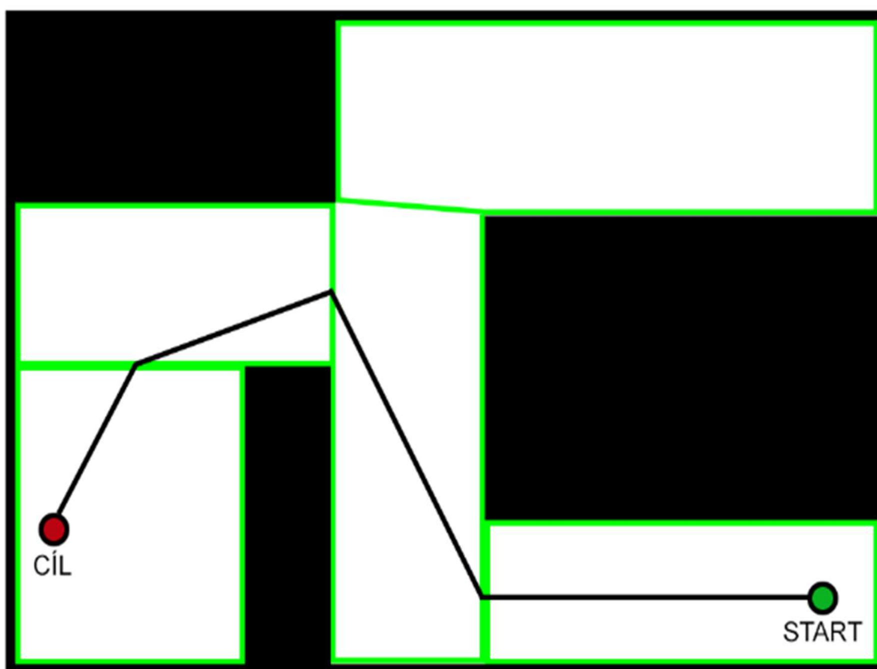
Výběr heuristické funkce závisí částečně také na typu mřížky, na které cestu vyhledáváme. Proto se pro čtvercovou mřížku, která dovolí pohyb čtyřmi směry, využívá Manhattanská vzdálenost, jelikož se jedná pouze o jednoduchý, přímý pohyb na sever, jih, východ a západ. Pohyb po čtvercové mřížce s osmi povolenými směry pohybu již vyžaduje využití Chebyschevovy vzdálenosti. Jedná se o výpočet vzdálenosti také nazýván diagonální, a proto je pro tuto situaci nejvýhodnější. Hexagonální mřížka, která dovolí pohyb v šesti směrech, vyžaduje využití Manhattanské vzdálenosti, která je samozřejmě upravená pro šestistranné mřížky, kde se již nepohybujeme pouze ve čtyřech směrech, jako tomu bylo u čtvercových mřížek.

4.5.1.2 Optimalizace vyhledávacího prostoru

V herním prostředí musí pohyblivé charaktery využívat podkladovou datovou strukturu, reprezentující vyhledávací prostor, pro prohledávání cesty k danému cíli. Nalezení nejvhodnější struktury je zásadní pro dosažení realisticky vypadajícího pohybu a přijatelného výkonu při hledání cesty. Jednodušší vyhledávací prostor znamená, že algoritmus A* nebude mít tolik práce s prohledáváním prostoru a tím se celý proces urychlí. V následující podkapitole jsou zmíněny dvě populární optimalizace založené na algoritmu A*. [1]

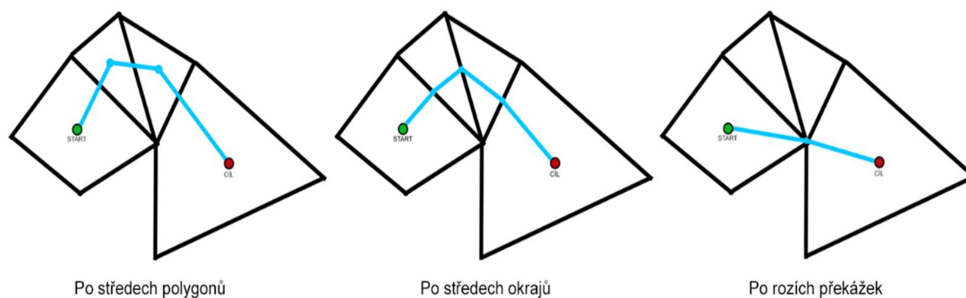
4.5.1.2.1 Navigační síť (NavMesh)

Navigační síť je metoda, sloužící k vytvoření herního prostředí za použití konvexních mnohoúhelníků. Vlastnosti takovýchto polygonů by mohly zaručit volný pohyb herní postavy za předpokladu, že zůstane v určitém mnohoúhelníku. [1] Metoda navigační sítě generuje graf trasových bodů minimalizující počet navgraph uzlů, které jsou nezbytné pro reprezentaci daného herního prostředí a tím se také zaručí dokonalé pokrytí průchodného prostředí. [4]



Obr. 17: Výsledná cesta s optimalizací NavMesh

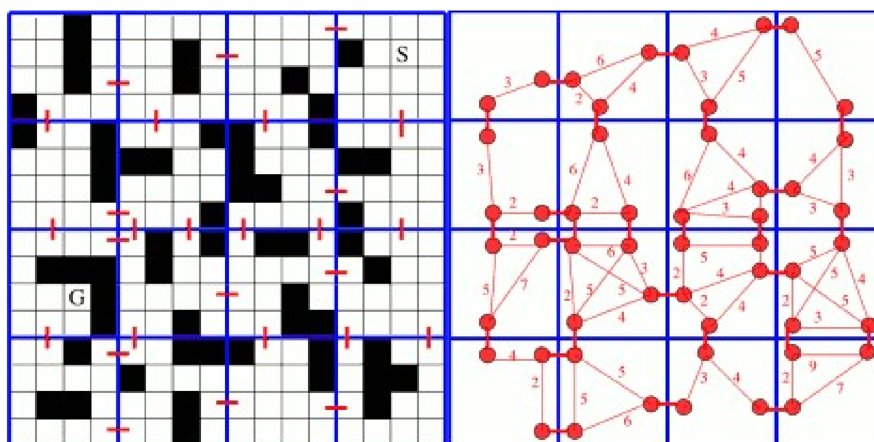
V prostředí vytvořeném metodou navigační sítě algoritmus A* funguje následovně. Mějme polygony průchodné, na obr. 17 ohraničené zeleně a neprůchodné, které jsou černobílé. Nejprve musíme nalézt polygony, kterými by optimální cesta mohla vést. Do této množiny jednoznačně patří polygony startovní a cílové příslušně řazeny. Pro nalezení dalších polygonů musíme všechny polygony přiřadit k uzlům. Například polygon 1 bude mapován k uzlu 1 a polygon 2 k uzlu 2. Polygony, které se dotýkají hranami, vytvoří spojnice mezi uzly, což bude reprezentovat možnou cestu. Poté algoritmus A* najde cestu ze startovního uzlu ke koncovému. Existuje několik možností cest, po středech polygonů, podél středového okraje a podél rohů překážky, vykreslené na obr. 18.



Obr. 18: V pořadí využití středů polynomů, okrajů a rohů překážek

4.5.1.2.2 Hierarchické hledání cesty (HPA*)

Hierarchické hledání cesty je výkonná technika, urychlující celý proces. Pokud by měl algoritmus A* hledat nejkratší cestu ve velmi rozsáhlém prostředí, lze prostředí rozdělit na několik menších částí. Tím se vytvoří hierarchický systém, kde se nejprve vyřeší cesta ve velkém měřítku, a poté se algoritmus může soustředit detailněji na jednotlivé sektory. Využití této techniky je zvýrazněno na obr. 19. Levá strana obsahuje mapu s překážkami, startovním místem S a cílem G. Pravá strana rozdělila celou mapu na šestnáct stejně velkých čtverců a A* algoritmus vyřešil cesty mezi sousedními čtverci. Tím se vybraly nejefektivnější přechody, na které se algoritmus poté soustředil detailněji, aby našel nejkratší cestu. [2]



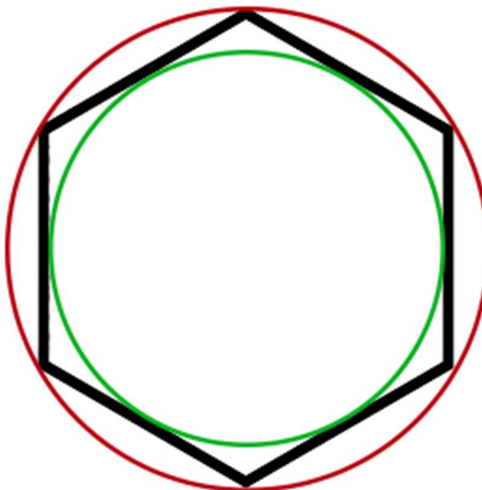
Obr. 19: Vlevo obecná mapa, vpravo mapa rozdělena pro využití hierarchie [19]

5 VLASTNÍ ŘEŠENÍ

Praktickou částí práce byla integrace algoritmu do virtuálního prostředí počítačové hry. Pro vyhledávání cesty byl zvolen algoritmus A* kvůli již rozšířenému využití v počítačových hrách a také kvůli přizpůsobivé optimalizaci. Jako tvar virtuálního prostředí byla zvolena hexagonální mřížka, která, jak již bylo zmíněno v předešlé kapitole 3.1, je v počítačových hrách používaná často díky jednoduchosti přechodu mezi jednotlivými hexagony. Pro vytvoření a testování kódu mřížky, a také implementaci algoritmu A*, byl vybrán multiplatformní engine Unity vyvinutý společností Unity Technologies, ve kterém je možno vytvářet 2D i 3D hry a nabízí primární rozhraní pro programování aplikací (API) v jazyce C#.

5.1 Vytvoření hexagonální mřížky

V první části této kapitoly byla popsána tvorba hexagonální mřížky. Jedním z problémů byly jednotlivé hexagony, které jsou na rozdíl od čtverců konstrukčně náročnější, ale řeší některé dlouhodobé problémy, jako jsou například jednotliví sousedé a přechody mezi nimi.



Obr. 20: Hexagon s opsanou (červená) a vepsanou (zelená) kružnicí

Na obr. 20 je zakreslen hexagon orientovaný rohem vzhůru s délkou hrany 10 jednotek. V počítačových hrách se ještě využívá druhá orientace stranou vzhůru, ale pro tuto praktickou část byla vybrána orientace první. Hexagon se skládá z šesti rovnostranných trojúhelníků, vzdálenost od středu k rohu bude v našem případě 10 jednotek. Pro vytváření cesty mezi hexagony je také důležitá vepsaná kružnice, jelikož nám určí vzdálenost od středu hexagonu do středu hrany. Dvojnásobek této vzdálenosti

se bude rovnat přechodu ze středu jednoho hexagonu do středu druhého. Poloměr této kružnice je zadán rovnicí:

$$r_v = a * \left(\frac{\sqrt{3}}{2}\right), \tag{5}$$

kde r_v je poloměr vepsané kružnice, a je poloměr opsané kružnice, v tomto případě 10 jednotek. Tyto hodnoty byly zapsány do veřejné statické třídy (public static class), která poskytuje plány svých zavedených tříd, ve zdrojovém kódu HexMetrics uvedené níže a budou využity pro vytvoření jednotlivých hexagonů:

```
public static class HexMetrics {

   public const float outerRadius = 10f;

   public const float innerRadius = outerRadius * 0.866025404f;
}
```

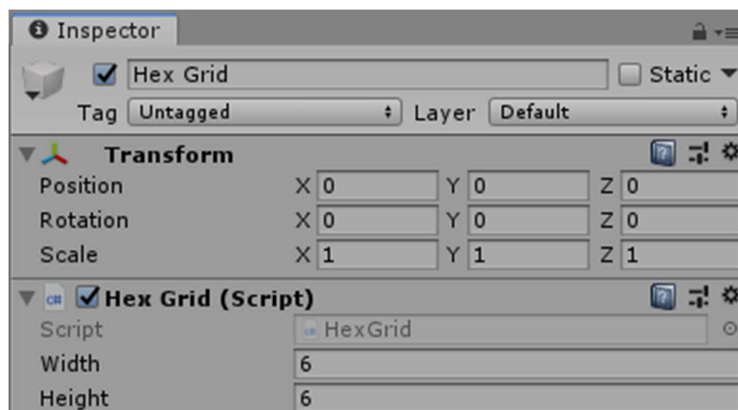
Jak již bylo zmíněno výše, je potřeba vytvořit jednotlivé hexagony, které poté rozložíme a tím vytvoříme mřížku. Jako první se musely definovat rohy hexagonu se začátkem v horním rohu a umístěním v rovině XZ, jak je vidět níže:

```
public static Vector3[] corners = {
   new Vector3(0f, 0f, outerRadius),
   new Vector3(innerRadius, 0f, 0.5f * outerRadius),
   new Vector3(innerRadius, 0f, -0.5f * outerRadius),
   new Vector3(0f, 0f, -outerRadius),
   new Vector3(-innerRadius, 0f, -0.5f * outerRadius),
   new Vector3(-innerRadius, 0f, 0.5f * outerRadius),
   new Vector3(0f, 0f, outerRadius)
}
```

Samotná mřížka je herní objekt (GameObject) s vlastním zdrojovým kódem HexGrid, se kterým je možné komunikovat a který obsahuje veřejné proměnné (public int) pro úpravu výšky a šířky mřížky:

```
public class HexGrid : MonoBehaviour {
   public int width = 6;
   public int height = 6;
}
```

Veřejná proměnná je v tomto případě číslo, které je uloženo v paměti počítače a přístup k němu není omezen. Unity obsahuje okno pro inspekci herních objektů (Inspector), kde je možné výšku a šířku mřížky měnit, zobrazeno na obr. 21.



Obr. 21: Inspekce objektu v Unity

Jako další byla potřeba komponenta buňky šestiúhelníku HexCell, pro kterou se vytvořil nový zdrojový kód a následně se do zdrojového kódu mřížky vložila nová veřejná proměnná pro buňku, která bude kód šestiúhelníkové buňky využívat:

```
public class HexCell : MonoBehaviour {
}
```

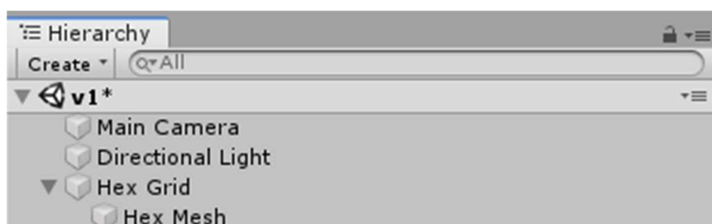
Vykreslování šestiúhelníků bude probíhat rozložením jednotlivých hexagonů na trojúhelníky v síti HexMesh, která pokrývá celou mřížku a využívá síťový filtr a vykreslovač, síť a seznam pro vrcholy a trojúhelníky:

```
using UnityEngine;
using System.Collections.Generic;

[RequireComponent(typeof(MeshFilter), typeof(MeshRenderer))]
public class HexMesh : MonoBehaviour {
   Mesh hexMesh;
   List<Vector3> vertices;
   List<int> triangles;
   void Awake () {
      GetComponent<MeshFilter>().mesh = hexMesh = new Mesh();
      hexMesh.name = "Hex Mesh";
      vertices = new List<Vector3>();
      triangles = new List<int>();
   }
```

Po vložení sítě pod mřížku v záložce hierarchie, naznačeno v obr. 22, je mřížka schopna načíst svou hexagonální síť a po probuzení sítě mřížka nařídí rozdělení buněk na trojúhelníky:

```
public class HexGrid : MonoBehaviour {
   …
   HexMesh hexMesh;
   void Awake () {
      hexMesh = GetComponentInChildren<HexMesh>();
```

```
}
void Start () {
hexMesh.Triangulate(cells);
}
```



Obr. 22: Založení sítě pod mřížku jako jejího potomka

Jelikož mřížka nařizuje síti rozdělení buněk na trojúhelníky, je potřeba tuto metodu také vytvořit v samotném zdrojovém kódu sítě. Metodu, což je kódový blok obsahující příkazy, které budou provedeny voláním této metody, lze vyvolat kdykoliv, proto je potřeba vymazat stará data, a poté projít jednotlivě každou buňku a rozdělit ji na trojúhelníky:

```
public void Triangulate (HexCell[] cells) {
   hexMesh.Clear();
   vertices.Clear();
   colors.Clear();
   triangles.Clear();
   for (int i = 0; i < cells.Length; i++) {
      Triangulate(cells[i]);
   }
   hexMesh.vertices = vertices.ToArray();
   hexMesh.colors = colors.ToArray();
   hexMesh.triangles = triangles.ToArray();
   hexMesh.RecalculateNormals();
}

void Triangulate (HexCell cell) {
   Vector3 center = cell.transform.localPosition;
   for (int i = 0; i < 6; i++) {
      AddTriangle(
      center,
      center + HexMetrics.corners[i],
      center + HexMetrics.corners[i + 1]
   }
}
```

Jelikož se šestiúhelníky skládají z rovnostranných trojúhelníků, byla vytvořena metoda, která tyto trojúhelníky přidá přidáváním jednotlivých vrcholů v pořadí:

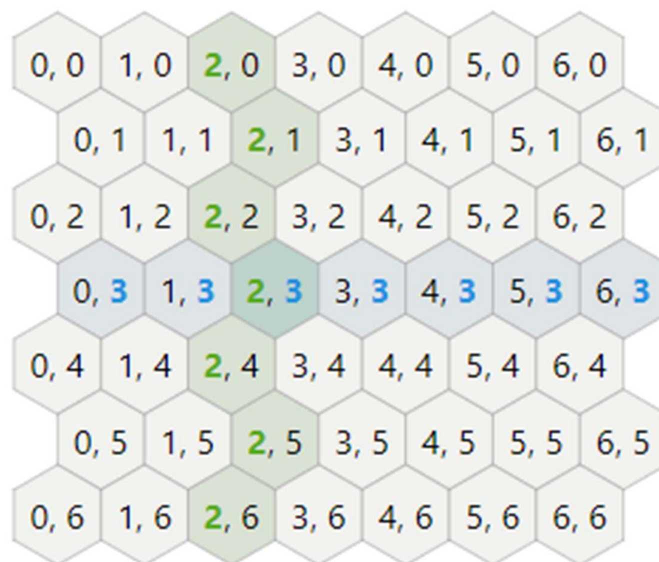
```
void AddTriangle (Vector3 v1, Vector3 v2, Vector3 v3) {
   int vertexIndex = vertices.Count;
   vertices.Add(v1);
```

```
    vertices.Add(v2);
    vertices.Add(v3);
    triangles.Add(vertexIndex);
    triangles.Add(vertexIndex + 1);
    triangles.Add(vertexIndex + 2);
}
```

V tento moment je vyřešena síť, která vytváří trojúhelníky, ze kterých se složí šestiúhelníky, ale mřížka ještě neumí vytvářet buňky, které by mohly být rozděleny. Musela být sepsána metoda pro vytvoření buněk, která se postupně posunuje po mřížce tak, aby se šestiúhelníky nepřekrývaly a zároveň celou mřížku zaplnily. Toto řešení je vidět na řádcích dva až pět metody CreateCell. Souřadnice x potřebuje odsazení o polovinu souřadnice z, kdy každý druhý řádek vyžaduje posunutí zpět, aby se zachoval tvar obdelníku. Souřadnice z nevyžaduje složité výpočty posunutí, pouze umožňuje vytvoření další buňky v nové řadě a souřadnice y v tomto případě nehraje žádnou roli.

```
void Awake () {
    …
    cells = new HexCell[height * width];
    for (int z = 0, i = 0; z < height; z++) {
        for (int x = 0; x < width; x++) {
            CreateCell(x, z, i++);
        }
    }
}

void CreateCell (int x, int z, int i) {
    Vector3 position;
    position.x = (x + z * 0.5f - z / 2) * (HexMetrics.innerRadius * 2f);
    position.y = 0f;
    position.z = z * (HexMetrics.outerRadius * 1.5f);
    HexCell cell = cells[i] = Instantiate<HexCell>(cellPrefab);
    cell.transform.SetParent(transform, false);
    cell.transform.localPosition = position;
}
```

Pro lepší orientaci a vyhledávání jednotlivých buněk je výhodné zobrazovat souřadnice. Klasický souřadnicový systém by ale v tomto případě nezobrazoval nejlepší výsledky, jelikož se nejedná o čtvercovou mapu, ale o šestiúhelníkovou, takže zobrazování souřadnic z postupuje klikatě, naznačeno v obr. 22.



Obr. 23: Souřadnice hexagonální mřížky [14]

Před řešením tohoto problému je ale potřeba vytvořit nové uživatelské rozhraní, na kterém se souřadnice buněk ukážou. Přidáním plátna (Canvas), což je herní objekt uživatelského rozhraní, se může vytvořit textový typ pro souřadnice, který se ukáže na plátně pro jednotlivé buňky. Plátno se zařadí pod mřížku, stejně jako tomu bylo u sítě:

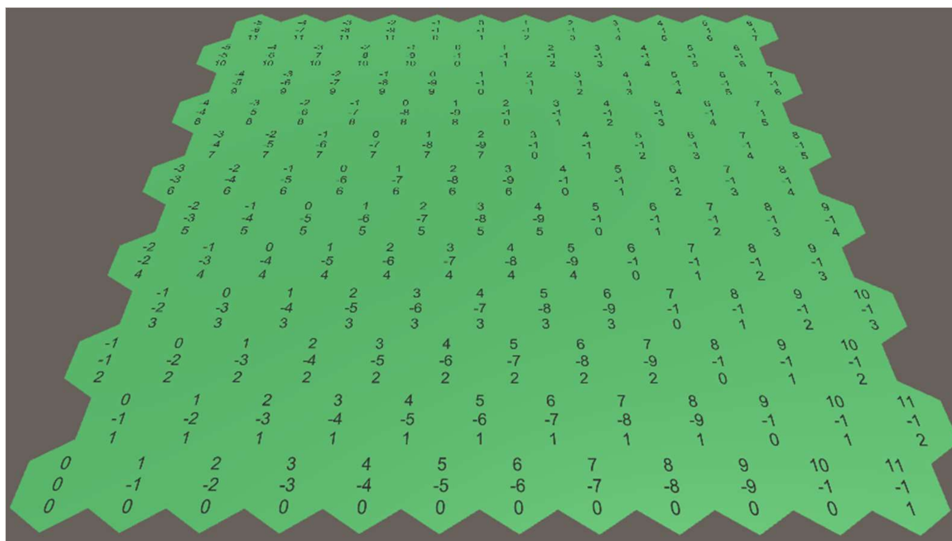
```
public class HexGrid : MonoBehaviour {
   …
   public Text cellLabelPrefab;
   Canvas gridCanvas;
   …
   void Awake () {
      …
      gridCanvas = GetComponentInChildren<Canvas>();
      …
   }
   Text label = Instantiate<Text>(cellLabelPrefab);
   label.rectTransform.SetParent(gridCanvas.transform, false);
   label.rectTransform.anchoredPosition = new Vector2(position.x, position.z);
   label.text = cell.coordinates.ToStringOnSeparateLines();
   cell.uiRect = label.rectTransform;
```

Následné vyřešení problému posunutí celého souřadnicového systému tak, aby žádné souřadnice neprocházely skrz mřížku klikatě vyžaduje vytvoření nové struktury HexCoordinates a následné nastavení nového, použitelného souřadnicového systému, který zároveň využívá dohromady tři souřadnice, kde y je převrácená hodnota x:

```
using UnityEngine;

[System.Serializable]
public struct HexCoordinates {

   public int X { get; private set; }
```

```
    public int Z { get; private set; }

    public HexCoordinates(int x, int z) {
        X = x;
        Z = z;
    }
}

public static HexCoordinates FromOffsetCoordinates (int x, int z) {
    return new HexCoordinates(x - z / 2, z);
}

public override string ToString() {
    return "(" + X.ToString() + ", " + Y.ToString() + ", " + Z.ToString() + ")";
}

public string ToStringOnSeparateLines() {
    return X.ToString() + "\n" + Y.ToString() + "\n" + Z.ToString();
}
```

Souřadnicový systém zobrazený na plátně, společně s rozkládáním buněk na trojúhelníky a vytváření šestiúhelníků přes síť vytvoří hexagonální mřížku, se kterou je možno dále pracovat ať už pro účely testování algoritmů hledání cesty nebo dalšímu tvoření a vývoji počítačové hry. Na obr. 24 je vidět, jak celá mapa prozatím vypadá v Unity.



Obr. 24: Realizace kódů v Unity

5.2 Úpravy a sousedé

Herní mapa, která pouze ukazuje souřadnice jednotlivých polí, je v tento moment prakticky nepoužitelná. Interakce s mapou je základní podmínka funkční počítačové hry a pro účely testování algoritmu hledání cesty je také důležitá pro vytváření překážek, se kterými si algoritmus bude muset poradit. Další důležitou složkou je určení sousedních buněk pro zapsání do otevřeného a uzavřeného listu.

Pro interakci s buňkami je nejlepší použít počítačovou myš, z jejíž pozice vystřelí paprsek, který se dotkne dané buňky. Základem programování je přehlednost, a proto je vytvoření nového zdrojového kódu HexMapEditor, který se bude zabývat zásadně úpravou buněk, výhodou. Také se musí do sítě HexMesh přidat komponent kolize (mesh collider), který bude na myš patřičně podle potřeb reagovat:

```
public class HexMapEditor : MonoBehaviour {
   public HexGrid hexGrid;
   private Color activeColor;
   void Awake () {
      SelectColor(0);
   }

   void Update () {
      if (Input.GetMouseButton(0) &&
                                  !EventSystem.current.IsPointerOverGameObject(){
         HandleInput();
      }
      else {
         previousCell = null;
      }
   }
   void HandleInput() {
      Ray inputRay = Camera.main.ScreenPointToRay(Input.mousePosition);
      RaycastHit hit;
      if (Physics.Raycast(inputRay, out hit)) {
         TouchCell(hit.point);
      }
   }
   void TouchCell(Vector3 position) {
      position = transform.InverseTransformPoint(position);
      Debug.Log("touched at " + position);
   }

public class HexMesh : MonoBehaviour {
   MeshCollider meshCollider;
   …
   public void Triangulate (HexCell[] cells) {
   …
   meshCollider.sharedMesh = hexMesh;
}
```

Na buňky je již možné působit, ale neví se přesně, na kterou z nich je toto působení mířeno. Dotyková poloha musí být převedena na hexagonální souřadnice, které jsou zapsány ve zdrojovém kódu HexCoordinates. Problémem je, že metoda zabývající se tímto problémem neví, které souřadnice patří k určité pozici. Začít se může určením středů šestiúhelníků. Souřadnice X se musí vydělit šířkou šestiúhelníku a její záporná hodnota je rovna souřadnici Y, jelikož se zrcadlí. Pro získání souřadnice Z se musí každé dvě řady počítat s posunutím o celou jednotku doleva. Zaokrouhlením souřadnic na celá čísla by se měly získat výsledné souřadnice. Zaokrouhlování souřadnic bohužel vede

k problémům blízko hranic jednotlivých šestiúhelníků, proto je jasné, že souřadnice, které se zaokrouhlují nejvíce jsou problémové. Souřadnice, která se zaokrouhluje nejvíce, je tudíž nepotřebná a může se zrekonstruovat zbývajícími dvěma souřadnicemi. Toto je důležité jen u souřadnic X a Z, protože Y je pro řešení toho problému nepodstatná.

```
public struct HexCoordinates {
   …
   public static HexCoordinates FromPosition (Vector3 position) {
      float x = position.x / (HexMetrics.innerRadius * 2f);
      float y = -x;
      float offset = position.z / (HexMetrics.outerRadius * 3f);
      x -= offset;
      y -= offset;
      int iX = Mathf.RoundToInt(x);
      int iY = Mathf.RoundToInt(y);
      int iZ = Mathf.RoundToInt(-x -y);
      if (iX + iY + iZ != 0) {
         float dX = Mathf.Abs(x - iX);
         float dY = Mathf.Abs(y - iY);
         float dZ = Mathf.Abs(-x -y - iZ);
         if (dX > dY && dX > dZ) {
            iX = -iY - iZ;
         }
         else if (dZ > dY) {
            iZ = -iX - iY;
         }
      }
      return new HexCoordinates(iX, iZ);
}
```

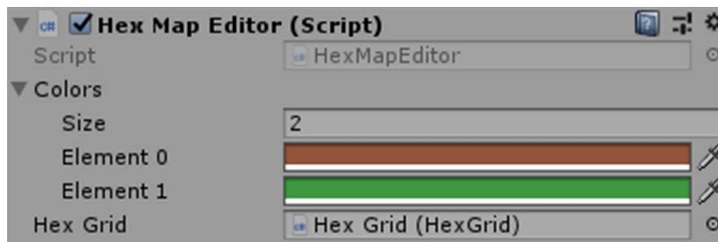
Jak již bylo zmíněno v úvodu této podkapitoly, testování algoritmů hledání cesty vyžaduje překážky. Proto je dobré tyto překážky vytvářet a mazat jednoduchou změnou parametru buňky, který je binární proměnná (bool) a může nabývat dvou možných hodnot 0 (false) a 1 (true). Další důležitou součástí této práce je samozřejmě integrace algoritmu, kde je potřeba vybrat buňku startovní a buňku cílovou. Za takových okolností je dobré umožnit změnu mezi režimem úprav a režimem testování algoritmu. Novou metodu, která bude sloužit pouze ke změně určitých parametrů, vytvoříme ve zdrojovém kódu HexMapEditor:

```
public class HexMapEditor : MonoBehaviour {
   bool applyWalkable;
   bool applyObstacle;
   …
   void EditCell(HexCell cell){
   if (applyObstacle) {
      cell.IsObstacle = true;
      cell.color = colors[0];
      hexGrid.Refresh();
   }
   if (applyWalkable) {
      cell.IsObstacle = false;
      cell.color = colors[1];
      hexGrid.Refresh();
   }
```

Pro zajištění změny barvy je nutné ve zdrojovém kódu HexCell vytvoření veřejného pole barvy, což je proměnná jakéhokoli typu, deklarovaná přímo ve třídě nebo struktuře, přidat k vytváření buněk ve zdrojovém kódu HexGrid výchozí barvu a přidání informace o barvě v kódu HexMesh, kde se ke každému trojúhelníku, ze kterého se poté složí šestiúhelník, přidá daná barva:

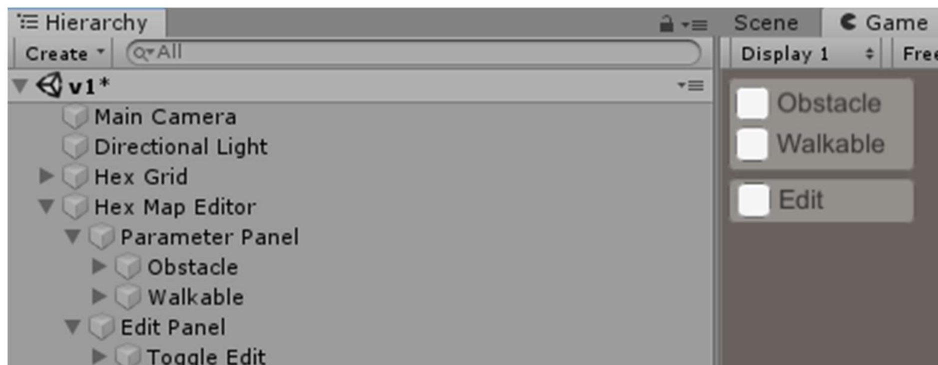
```
public class HexCell : MonoBehaviour {
   …
   public Color color;
   …
}

public class HexGrid : MonoBehaviour {
   …
   public Color defaultColor = Color.white;
   …
   void CreateCell (int x, int z, int i) {
      …
      cell.color = defaultColor;
      …
   }
   …
}

public class HexMesh : MonoBehaviour {
   List<Color> colors;
   void Awake () {
      …
      colors = new List<Color>();
   }
   public void Triangulate (HexCell[] cells) {
      …
      colors.Clear();
      …
      hexMesh.colors = colors.ToArray();
   }
   void Triangulate (HexCell cell) {
      …
      for (int i = 0; i < 6; i++) {
         …
         AddTriangleColor(cell.color);
      }
   }
   void AddTriangleColor (Color color) {
      colors.Add(color);
      colors.Add(color);
      colors.Add(color);
   }
}
```

Inspekce herního objektu Hex Map Editor ukazuje nový komponent na obr. 25, kde je vidět počet barevných prvků, které je možné měnit.



Obr. 25: Prvky barvy v inspekci herního objektu

Aby bylo možné upravovat parametry jednotlivých buněk, je potřeba využít nový herní objekt patřící pod uživatelské rozhraní s názvem panel (Panel) a pod něj vložit další objekty uživatelského rozhraní, které umožní přepínat mezi různými možnostmi, což je vidět na obr. 26. V tomto případě bude potřeba zajistit přepínání režimu úprav a výběr prvků překážky a cesty, které jsou použitelné pouze tehdy, když je režim úprav povolený.



Obr.26: Hierarchie objektů a viditelnost ve hře

Algoritmus potřebuje znát sousední buňky, aby je mohl zapsat do otevřeného a zavřeného listu. Každá buňka má šest sousedů, které lze jednoduše identifikovat podle směrů kompasu severovýchod, východ, jihovýchod, jihozápad, západ a severozápad. Pomocí výčtu (enum) lze definovat typ výčtu, který je uspořádaným seznamem jmen. Každé z těchto jmen odpovídá číslu, ve výchozím nastavení počínaje nulou. Výčet je užitečný, pokud je potřeba omezený seznam pojmenovaných voleb. Výčet by měl být vytvořen ve zdrojovém kódu HexDirections pro přehlednost a dále je nutné v kódu HexCell vytvořit pole (array), kde se tito sousedé uloží. Pole se ukáže v inspekci herního objektu, ale zatím bez hodnot:

```
public enum HexDirections {
	NE, E, SE, SW, W, NW
}
```

```
public class HexCell : MonoBehaviour {
   [SerializeField]
   HexCell[] neighbors;
   …
   public HexCell GetNeighbor(HexDirections direction) {
      return neighbors[(int)direction];
   }
   public void SetNeighbor(HexDirections direction, HexCell cell) {
      neighbors[(int)direction] = cell;
      cell.neighbors[(int)direction.Opposite()] = this;
   }
   …
}
```

Vztahy mezi soused jsou obousměrné, proto po nastavení jednoho směru je rozumné okamžitě nastavit i směr opačný. Toho lze docílit metodou rozšíření HexDirections, což je statická metoda uvnitř statické třídy, která se chová jako instanční metoda. V tomto případě je potřeba přičíst tři v původním směru a tři odečíst v ostatních.

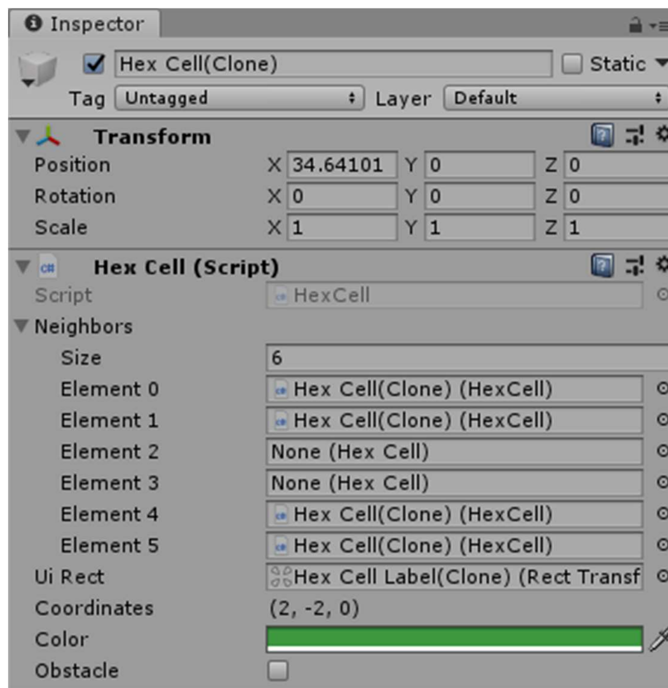
```
public static class HexDirectionExtensions {
   public static HexDirections Opposite(this HexDirections direction) {
      return (int)direction < 3 ? (direction + 3) : (direction - 3);
   }
}
```

Vytvoření sousedského spojení mezi buňkami je rozumné dělat postupně po řádcích, zleva doprava, protože je jasné, které buňky již byly vytvořeny a ke kterým je možné se připojit. Nejjednodušší je propojení východu a západu, jelikož první buňka v každé řadě nemá západního souseda, ale všechny ostatní ano. Jednotlivé buňky jsou schopny ukázat, které s nimi sousedí, proto všechny tyto operace patří do HexGrid k vytváření buněk:

```
public class HexGrid : MonoBehaviour {
   …
   if (x > 0) {
      cell.SetNeighbor(HexDirections.W, cells[i - 1]);
   }
```

Další obousměrná spojení vyžadují přeskočení prvního řádku, jelikož se zde nevyskytují. Jedná se o dvě obousměrná spojení. Jihovýchodní a k ní opačná severozápadní a jihozápadní a k ní severovýchodní. Nejlepší je soustředit se prvně na například sudé řádky od prvního. Všechny buňky v tomto řádku mají jihovýchodního souseda a každá buňka až na první vlevo má jihozápadního souseda. Liché řádky od prvního se řídí stejnou logikou, ale jsou přetočeny zrcadlově. Po dokončení těchto operací jsou všichni možní sousedé propojeni a jsou vidět při inspekci herního objektu buňky na obr. 26:

```
if (z > 0) {
      if ((z & 1) == 0) {
         cell.SetNeighbor(HexDirections.SE, cells[i - width]);
         if (x > 0) {
            cell.SetNeighbor(HexDirections.SW, cells[i - width - 1]);
         }
      }
      else {
         cell.SetNeighbor(HexDirections.SW, cells[i - width]);
         if (x < width - 1) {
            cell.SetNeighbor(HexDirections.SE, cells[i - width + 1]);
         }
      }
   }
```



Obr. 27: Unity inspektor buňky

5.3 Aplikace algoritmu

Poslední podkapitola vlastního řešení praktické části, která se zabývá integrací vyhledávacího algoritmu do prostředí počítačové hry, se zabývá především samotným algoritmem. Před tím se ale musí vyřešit vzdálenosti mezi jednotlivými buňkami a vyhýbání se překážkám.

Nejpřehlednější je zajistit vizualizaci hodnot vzdálenosti od startovní buňky. Ve zdrojovém kódu HexCell se proto musí vytvořit nová celočíselná proměnná (int), metoda, která bude aktualizovat vzdálenost buňky a veřejnou vlastnost, která získá, nastaví a aktualizuje vzdálenost buňky:

```
public class HexCell : MonoBehaviour {
   …
   int distance;
   …
   void UpdateDistanceLabel() {
      Text label = uiRect.GetComponent<Text>();
      label.text = distance == int.MaxValue ? "" : distance.ToString();
   }
   public int Distance {
      get {
         return distance;
      }
      set {
         distance = value;
         UpdateDistanceLabel();
      }
   }
```

Pro zaručení přehledné vizualizace výsledné cesty, kterou algoritmus vyřeší, je vhodné přidat jednoduché zvýraznění. Jelikož se pracuje s hexagonální mřížkou, bílý šestiúhelník postačí. Po vytvoření nového herního objektu, pod který se vloží komponent obrázku zaručí, že každá buňka bude mít svoje vlastní zvýraznění, které se bude ovládat aktivováním a deaktivováním komponentu obrázku:

```
public class HexCell : MonoBehaviour {
   …
   public void DisableHighlight() {
      Image highlight = uiRect.GetChild(0).GetComponent<Image>();
      highlight.enabled = false;
   }
   public void EnableHighlight(Color color) {
      Image highlight = uiRect.GetChild(0).GetComponent<Image>();
      highlight.color = color;
      highlight.enabled = true;
   }
```

Toto zvýraznění, které je bílé, je možné využít i pro určení startovní a cílové pozice, kdy uživatel musí vybrat obě tyto místa, pokud se zrovna nevyskytuje v režimu úprav. Startovní pozici je buňka, ve které vyhledávaná cesta začne, proto HexMapEditor potřebuje vědět v metodě HandleInput, která buňka je první, ke které se cesta vyhledává a která předcházela, aby se algoritmus nevracel:

```
public class HexMapEditor : MonoBehaviour {
   HexCell previousCell,searchFromCell, searchToCell;
   …
   void HandleInput () {
      Ray inputRay = Camera.main.ScreenPointToRay(Input.mousePosition);
      RaycastHit hit;
```

```
      if (Physics.Raycast(inputRay, out hit)) {
         HexCell currentCell = hexGrid.GetCell(hit.point);
         if (editMode) {
            EditCell(currentCell);
        }
        else if (Input.GetKey(KeyCode.LeftShift) && searchToCell != currentCell){
           if (searchFromCell) {
              searchFromCell.DisableHighlight();
           }
           searchFromCell = currentCell;
           searchFromCell.EnableHighlight(Color.blue);
        }
```

Algoritmus pracuje s velkým množství buněk, proto se cílová pozice a hledání cesty řeší ve zdrojovém kódu HexGrid, kde je nutná metoda pro hledání cesty a zavedení koprogramu (coroutine), což umožní postupné sledování práce algoritmu. Rozhraní IEnumerator umožňuje iterovat seznamem ovládacích prvků a vyžaduje návrat na začátek seznamu a posuv vpřed:

```
public class HexGrid : MonoBehaviour {
   …
   public void FindPath(HexCell fromCell, HexCell toCell) {
      StopAllCoroutines();
      StartCoroutine(Search(fromCell, toCell));
   }
   IEnumerator Search(HexCell fromCell, HexCell toCell) {
      for (int i = 0; i < cells.Length; i++) {
         cells[i].Distance = int.MaxValue;
         cells[i].DisableHighlight();
      }
      fromCell.EnableHighlight(Color.blue);
      toCell.EnableHighlight(Color.red);
      WaitForSeconds delay = new WaitForSeconds(1 / 60f);
      List<HexCell> frontier = new List<HexCell>();
      fromCell.Distance = 0;
      frontier.Add(fromCell);
      while (frontier.Count > 0) {
         yield return delay;
         HexCell current = frontier[0];
         frontier.RemoveAt(0);
         if (current == toCell) {
            current = current.PathFrom;
            while (current != fromCell) {
               current.EnableHighlight(Color.white);
               current = current.PathFrom;
            }
            break;
         }
         for (HexDirections d = HexDirections.NE; d <= HexDirections.NW; d++) {
            int distance = current.Distance;
            HexCell neighbor = current.GetNeighbor(d);
            if (neighbor == null || neighbor.Distance != int.MaxValue) {
               continue;
            }
            if (neighbor.IsObstacle) {
               continue;
            }
            neighbor.Distance = current.Distance + 1;
            neighbor.PathFrom = current;
```

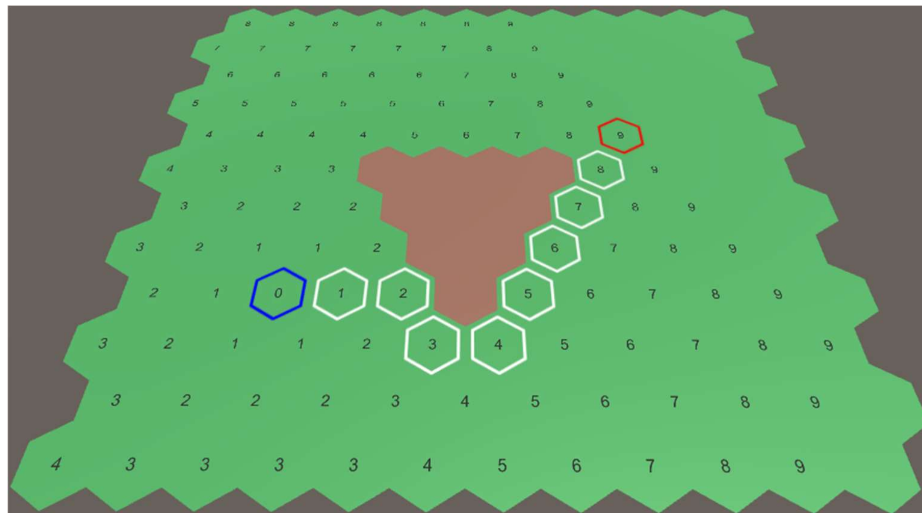
```
            neighbor.SearchHeuristic =
            neighbor.coordinates.DistanceTo(toCell.coordinates);
            frontier.Add(neighbor);
          frontier.Sort((x, y) => x.SearchPriority.CompareTo(y.SearchPriority));
        }
    }
}
```

Do tohoto kódu již bylo přidáno několik dalších příkazů. Postupně bylo přidáno zvýraznění cílové buňky červeným šestiúhelníkem a následně seznam (List), což je objekt obsahující proměnné v určitém pořadí, v tomto případě jednotlivé buňky. Ke sledování buněk, které je potřeba prohledat, poslouží kolekce otevřená sada (frontier). Algoritmus pracuje, dokud jsou v seznamu buňky určené k prohledání. Aby se zabránilo prohledávání celé mřížky, algoritmus se zastaví, jakmile najde výslednou nejkratší cestu, kterou zvýrazní bílými šestiúhelníky.

V poslední části kódu je řešeno určení vzdáleností mezi sousedy, kdy se všichni možní sousedé dané buňky vloží do pořadí vyhledávání, pouze tehdy když ještě nemají určenou hodnotu vzdálenosti a nejsou označené jako překážka. V této části je také vidět zavedení heuristiky. Algoritmus sice najde nejkratší cestu, ale prohledává také buňky, které zaručeně nebudou součástí nejkratší cesty. Heuristika proto slouží k odhadnutí nejkratší vzdálenosti k cílové buňce a algoritmus se nebude zabývat těmi, které jdou v opačném směru. Tato odhadovaná vzdálenost se počítá u každé řešené buňky zvlášť, proto se kód heuristiky zapíše v HexCell jako vlastnost priority vyhledávání, kterou HexGrid využije k seřazení otevřené sady:

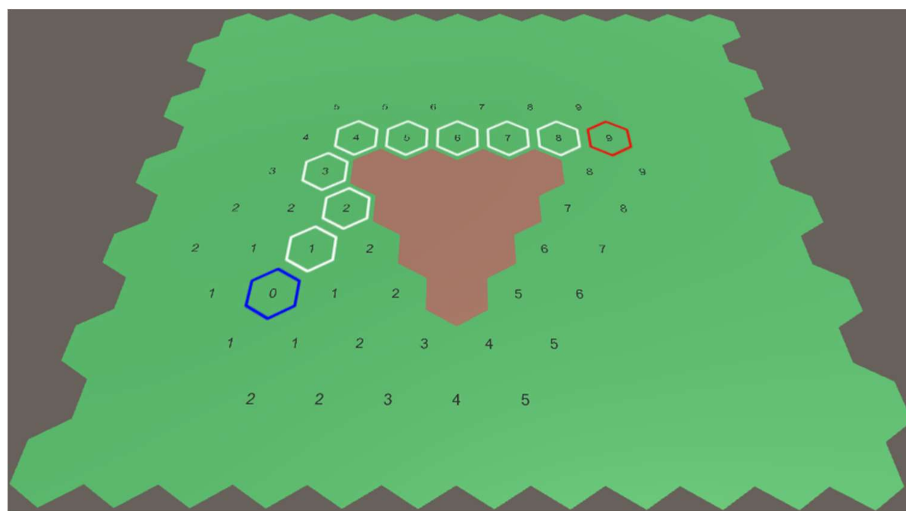
```
public class HexCell : MonoBehaviour {
   public int SearchHeuristic { get; set; }
   …
   public int SearchPriority {
      get {
         return distance + SearchHeuristic;
      }
   }
```

Výsledek aplikování těchto kódů lze vidět na obr. 27 a obr. 28. Obr. 27 zobrazuje integraci vyhledávacího algoritmu, kdy není zavedena žádná heuristika, pouze je vyhledávání omezeno, takže nebude prohledávat všechny přístupné buňky. Na obrázcích je vidět, že heuristika opravdu urychluje vyhledávání, což je žádoucí efekt a cesta sama o sobě zůstává vzdáleností pořád stejná, jako při nevyužití optimalizace pomocí heuristiky.



Obr. 28: Výsledná cesta z modré do červené pozice bez heuristiky

Následující obrázek naopak počítá se zavedením heuristiky a počet prohledaných buněk je tím omezen.



Obr. 29: Výsledná cesta z modré do červené pozice se zavedením heuristiky

6 ZÁVĚR

Jednotlivé body cíle této práce, zabývající se zkoumáním a porovnáním mapových sítí, algoritmů a integrací algoritmu pro vyhledání cesty ve virtuálním herním prostředí byly vypracovány ve vlastních kapitolách.

Mapové sítě, hlavně mřížky pravidelné a nepravidelné využívané ve 2D prostoru byly popsány v kapitole 3, kde jednotlivé příklady vyznívaly vlastními výhodami a nevýhodami. Hexagonální sítě, které sice využívají pouze pohyb v šesti směrech a jejich tvorba je složitější, jsou nadále nejefektivnější, jelikož pohyb mezi jednotlivými hexagony je vždy stejný, na rozdíl od čtvercových, které jsou buď omezenější, nebo počítají s různými hodnotami přechodů kvůli diagonálám.

Grafové algoritmy, od základních až po propracovanější, používané v herním prostředí, byly probrány v kapitole 4, kdy z porovnaných algoritmů je nejlepší algoritmus A*, který je ze základu podobný Dijkstrově algoritmu, ale dokáže rychle a efektivně najít cestu k cílovému bodu bez prohledávání celé mapy, za použití optimalizací funkce i vyhledávacího prostoru.

V kapitole 5 byla podrobně popsána práce na integraci vyhledávacího algoritmu v herním prostředí, kdy se musela vytvořit hexagonální mřížka, rozdělená na jednotlivé šestiúhelníky, které byly využity k nalezení nejkratší cesty ze startovního bodu do cílového, za využití předčasného ukončení procesu po vyhledání této cesty a heuristiky, která umožnila algoritmu soustředit se nejprve na prohledávání buněk, které byly cíli nejblíže v daný moment. Tyto dvě funkce ukončení procesu a heuristiky urychlily vyhledání nejkratší cesty bez nežádoucích vedlejších efektů.

Tato práce by se mohla v budoucnu rozšířit o další mapové sítě používané například ve 3D prostředí, algoritmy a optimalizace efektivnější než výše zmíněné a integrace do herního prostředí by se mohla rozšířit o další parametry, které by musel algoritmus zvážit. Takové parametry by mohly být rychlejší či pomalejší cesty, parametry popisující vlastnosti některých buněk, pokud chce uživatel dosáhnout cíle za určitých podmínek.

7 SEZNAM POUŽITÉ LITERATURY

[1] CUI, Xiao; SHI, Hao. *A*-based Pathfinding in Modern Computer Games*, 2010, 11.

[2] BOTEA, Adi; MÜLLER, Martin; SCHAEFFER, Jonathan. *Near optimal hierarchical path-finding*. Journal of game development, 2004, 1.1: 7-28.

[3] REDDY, Harika. *PATH FINDING - Dijkstra's and A* Algorithm's*, 2013 [cit. 2020-06-04]. Dostupné z: http://cs.indstate.edu/hgopireddy/algor.pdf

[4] ANGUELOV, Bobby. *Video Game Pathfinding and Improvements to Discrete Search on Grid-based Maps*, 2011 [cit. 2020-06-04]. Dostupné z: https://repository.up.ac.za/handle/2263/22940

[5] ADAIXO, Michaël. *Influence Map-Based Pathfinding Algorithms in Video Games,* 2014 [cit. 2020-06-04]. Dostupné z: https://pdfs.semanticscholar.org/5dce/47f2f4571e75c6e1945e1fbacd50380a92f2.pdf

[6] *Depth-First Search (DFS)* [online]. [cit. 2020-06-04]. Dostupné z: https://brilliant.org/wiki/depth-first-search-dfs/

[7] *Breadth-First Search (BFS)* [online]. [cit. 2020-06-04]. Dostupné z: https://brilliant.org/wiki/breadth-first-search-bfs/

[8] ALGFOOR, Zeyad Abd; SUNAR, Mohd Shahrizal; KOLIVAND, Hoshang. *A Comprehensive Study on Pathfinding Techniques for Robotics and Video Games*, 2015 [cit. 2020-06-04]. Dostupné z: https://www.hindawi.com/journals/ijcgt/2015/736138/

[9] *Rapidly-exploring Random Trees (RRTs).* [online]. [cit. 2019-05-19]. Dostupné z: http://lavalle.pl/rrt/

[10] NIU, Hanlin; LU, Yu; SAVVARIS, Al; TSOURDOS, Antonios. *An energy-efficient path planning algorithm for unmanned surface vehicles,* 2018. Dostupné z: https://www.researchgate.net/publication/325371124_An_energy-efficient_path_planning_algorithm_for_unmanned_surface_vehicles

[11] *Pathfinding I.* [online]. Dostupné z: https://yushutong.wordpress.com/2013/06/08/pathfinding-i/

[12] *Path finding using Rapidly-Exploring Random Tree.* [online]. Dostupné z: https://nakkaya.com/2011/10/27/path-finding-using-rapidly-exploring-random-tree/

[13] *How does Hierarchical Pathfinding deal with obstructions in the same chunk?*. [online] Dostupné z: https://gamedev.stackexchange.com/questions/130399/how-does-hierarchical-pathfinding-deal-with-obstructions-in-the-same-chunk

[14] *Hexagon grid coordinate systém.* [online]. Dostupné z: https://math.stackexchange.com/questions/2254655/hexagon-grid-coordinate-system

8 SEZNAM OBRÁZKŮ

Obr. 1 - Čtvercová síť čtyři směry……………………………………………………18
Obr. 2 - Čtvercová síť osm směrů………………………………………………….19
Obr. 3 – Pravoúhlý pohyb (a), diagonální pohyb (b)…………………………...19
Obr. 4 - Trojúhelníková síť s překážkou……………………………………...20
Obr. 5 - Hexagonální mřížka……………………………………………………21
Obr. 6 - Graf viditelnosti………………………………………………………..21
Obr. 7 - Navigační síť v herním prostředí………………………………………22
Obr. 8 - Schéma prohledávání do hloubky………………………………………23
Obr. 9 - Schéma prohledávání do šířky…………………………………………24
Obr. 10 - Výsledek algoritmu RRT……………………………………………...25
Obr. 11 - Graf uzlů s cenami přechodů…………………………………………25
Obr. 12 - Výsledné nejkratší cesty k uzlům……………………………………..26
Obr. 13 - Nejkratší cesta při použití algoritmu A*………………………………27
Obr. 14 - Výsledek při zavedení Euklidovské heuristiky………………………29
Obr. 15 - Výsledek při zavedení Manhattanské vzdálenosti……………………29
Obr. 16 - Výsledek při zavedení Chebyschevovy vzdálenosti…………………30
Obr. 17 - Výsledná cesta s optimalizací NavMesh……………………………...31
Obr. 18 - V pořadí využití středů polynomů, okrajů a rohů překážek …………...32
Obr. 19 - Vlevo obecná mapa, vpravo mapa rozdělena pro využití hierarchie….32
Obr. 20 - Hexagon s opsanou (červená) a vepsanou (zelená) kružnicí………….33
Obr. 21 - Inspekce objektu v Unity……………………………………………...35
Obr. 22 - Založení sítě pod mřížku jako jejího potomka………………………...36
Obr. 23 - Souřadnice hexagonální mřížky………………………………………38
Obr. 24 - Realizace kódů v Unity………………………………………………..39
Obr. 25 - Prvky barvy v inspekci herního objektu………………………………43
Obr. 26 - Hierarchie objektů a viditelnost ve hře………………………………..43
Obr. 27 - Unity inspektor buňky………………………………………………...45
Obr. 28 - Výsledná cesta z modré do červené pozice bez heuristiky ……………49
Obr. 29 - Výsledná cesta z modré do červené pozice se zavedením heuristiky …49

9 PŘÍLOHY

Soubor .rar, který obsahuje:

- Zdrojové kódy programu
- Unity projekt