Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Editor grafov s pridanými funkciami

Bakalárska práca

2023
Nikola Radošinská

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

Editor grafov s pridanými funkciami

Bakalárska práca

Študijný program:	Informatika
Študijný odbor:	Informatika
Školiace pracovisko:	Katedra informatiky
Školiteľ:	doc. RNDr. Edita Mačajová, PhD.

Bratislava, 2023
Nikola Radošinská

54185180

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Nikola Radošinská
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: slovenský
Sekundárny jazyk: anglický

Názov: Editor grafov s pridanými funkciami
Graph editor with additional functions

Anotácia: Úlohou študenta bude zostrojiť grafický editor, na zadávanie grafov a multipólov, ich kopírovnie, duplikovanie, rotovanie, spájanie do nových objektov. Graf spolu s jeho nakreslením bude možné uložiť pre ďalšie použitie. Na takto zadanom grafe program umožní nájsť farbenie bodovo-blokovými konfiguráciami so špecifikovanými obmedzeniami. Program bude umožňovať generovanie štandardne používaných grafov. Výstup bude možné uložiť do textového súboru v niektorom z preddefinovaných formátov.

Vedúci: doc. RNDr. Edita Mačajová, PhD.
Katedra: FMFI.KI - Katedra informatiky
Vedúci katedry: prof. RNDr. Martin Škoviera, PhD.

Dátum zadania: 08.11.2022

Dátum schválenia: 10.11.2022

doc. RNDr. Dana Pardubská, CSc.
garant študijného programu

.....
študent

.....
vedúci práce

Poďakovanie: Ďakujem školiteľke za pomoc a rady, ktoré počas celej tvorby poskytovala.

Abstrakt

Obsahom práce je doplnenie funkcionalít do grafického editora grafov, pre vyučujúcich na katedre informatiky, určeného na vedecké účely. Dopĺňali sme do editora, ktorý sme predtým vytvárali vrámci inšieho predmetu. Pôvodný editor vedel vytvárať grafy, hýbať s nimi, mal lokálnu databázu, vedel kopírovať grafy, rotovať s nimi a mal implementovanú Undo funkcionalitu. Okrem refaktorizácie starého kódu, sa dopĺňali funkcionality na hranové farbenie grafu, generovanie štandardných typov grafov, zachovanie pracovnej plochy pri vypnutí aplikácie, Redo funkcionalita, kontrakcia hrany, spojenie dvoch visiacich hrán do jednej a nastavenia aplikácie. Text práce sa venuje úvodu do teórie grafov, opisu technológií, ktoré sme pri tvorbe využili, vymenovanie funkcionalít a zdôvodnenie prečo boli potrebné pridať, návrh aplikácie a implementácia jednotlivých častí.

Kľúčové slová: teória grafov, editor, hranové farbenie grafu

Abstract

The content of the thesis is to add functionalities to the visual graph editor, meant for teachers in the department of informatics, created for scientific purposes. We added them to the editor, which we created for other subject. That editor was able to create graphs, manage them, copy them, and rotate them. It had a local database and implemented Undo functionality. Besides refactoring old parts of editor, we added functionalities for edge coloring, generation of standard types of graphs, retention of work place when shutting off aplication, redo functionality, edge contraction, merge of two hanging edges, and settings of aplication. The text of the work describes an introduction to graph theory, technologies we used during development, a list of functionalities and the need to create them, design of the application, and implementation of different parts.

Keywords: graph theory, editor, edge coloring of a graph

Obsah

Zoznam obrázkov

Úvod

Teória grafov je rozšírená vedná disciplína, ktorá má široké využitie v bežnom živote a je rozšírene skúmaná na akademickej pôde. Od začiatku štúdia sa stretávame s úlohami z oblasti teórie grafov aj mimo predmetov venujúcim sa matematike, a to napríklad v programátorských úlohách. Taktiež sa dá mnoho iných praktických problémov preniesť, do obdobných problémov v teórii grafov.

Keďže je to obšírna a praktická časť matematiky, venuje sa jej aj niekoľko vyučujúcich na našej fakulte. A práve pre nich bola tvorená táto aplikácia, na základe ich požiadaviek. Aplikácia je editor grafov a ponúka práve tie funkcionality, ktoré boli pre vyučujúcich dôležité a praktické. Potreba vytvorenia tohto editora spočíva v tom, že neexistuje dostupný editor, ktorý by spĺňal, čo i len malé množstvo podmienok kladených na ten náš.

Cieľom našej bakalárskej práce bolo zdokonaliť editor[2], ktorý sme vyrobili na ročníkovej práci pridaním funkcií, ktoré jednak uľahčia ovládanie a prácu s grafmi ako aj dodaním funkcií, ktoré umožnia rôzne druhy farbenia. Tento editor ponúka široké možnosti, ako vytvoriť graf a vo vizuálnom prostredí s ním narábať. Jeho hlavnou úlohou je uľahčiť skúmanie a vedeckú prácu s grafmi.

Táto bakalárska práca sa bude venovať z počiatku stručnému úvodu do teórie grafov a potom detailnému opisu procesu tvorby a implementácie editora. Popíšeme si ostatné editory, ukážeme si potrebu vytvorenia toho nášho a oboznámime sa s vývojovým prostredím. Ďalej sa budeme venovať samotnému procesu tvorby aplikácie. Vymenujeme si nutné funkcionality, ktoré robia náš editor jedinečným, jeho návrh a implementáciu.

Kapitola 1

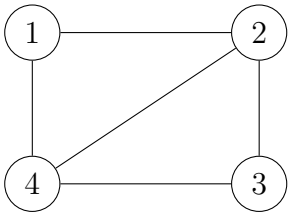
Súčasný stav problematiky

V tejto kapitole si zadefinujeme základné pojmy z teórie grafov, oboznámime sa s už existujúcimi editormi a s prostredím, v ktorom sme editor vyvíjali.

1.1 Základné pojmy z teórie grafov

Graf je matematický objekt.

Neorientovaný graf G je usporiadaná dvojica $G = (V, E)$, kde V je neprázdna konečná množina vrcholov grafu a E je konečná množina dvojprvkových podmnožín množiny V. Prvky množiny E budeme nazývať hrany. Hranu zapisujeme $\{u, v\}$, skrátene uv. Každý netriviálny graf má nekonečne veľa zobrazení v rovine. Príklad neorientovaného grafu: $G = (V, E)$, $V = \{1, 2, 3, 4\}$, $E = \{\{1, 2\}, \{2, 3\}, \{3, 4\}, \{4, 1\}, \{2, 4\}\}$ a jeho zobrazenia máme na obrázku 1.1.



Obr. 1.1: Príklad grafu G

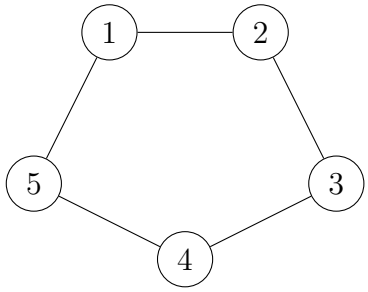
Ak dve hrany zdieľajú jeden vrchol nazývame ich susedné. Vrchol v je incidentný s hranou e ak $v \in e$. Vrcholy nazývame susedné ak sú incidentné s rovnakou hranou. Nech $G = (V, E)$. Graf $G' = (V', E')$ je podgraf grafu G ak $V' \subseteq V$ a $E' \subseteq E$. Komponent je maximálny súvislý podgraf grafu. Stupeň vrchola $deg(v)$ je počet hrán incidentných s vrcholom v. Ak majú všetky vrcholy v grafe G stupeň k, nazývame graf $k-$regulárny. Bipartitný graf je taký graf, pre ktorý platí, že množinu vrcholov vieme rozdeliť na dve množiny tak, že žiadne dva vrcholy z rovnakej množiny nebudú susedné. Visiaca hrana je taká hrana, ktorá má jeden koniec voľný, čiže nemá pripojený jeden vrchol. Značíme ju $\{v, -1\}$. Násobné hrany, sú také hrany, ktoré spájajú tie isté vrcholy. Slučka je taká hrana, ktorá má na svojich koncoch rovnaký vrchol.

V teoretickej časti textu v definíciách neuvažujeme násobné hrany aj slučky, no náš editor ich povoľuje.

1.1.1 Štandardné typy grafov

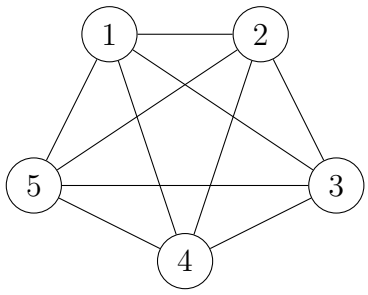
Teraz si predstavíme niektoré štandardné typy grafov, ktorých generovanie sme implementovali do editora.

Graf G nazývame kružnicou ak $V = \{v_1, v_2, ..., v_n\}$ a $E = \{v_1v_2, v_2v_3, ..., v_{n-1}v_n, v_nv_1\}$, kde $n \geq 3$. Označujeme ho C_n a ukážku jeho zobrazenia máme na obrázku 1.2.



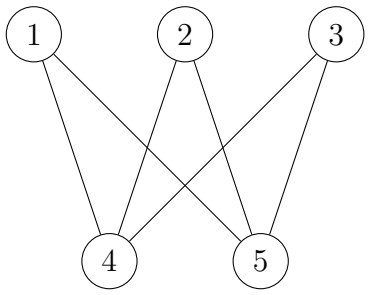
Obr. 1.2: Príklad C_5 grafu

Graf G nazývame kompletný ak $|V| = n$ a je $n-1$-regulárny, $n \geq 1$. To znamená, že každá dvojica vrcholov je susedná. Označujeme ho K_n a príklad jeho zobrazenia máme na obrázku 1.3.



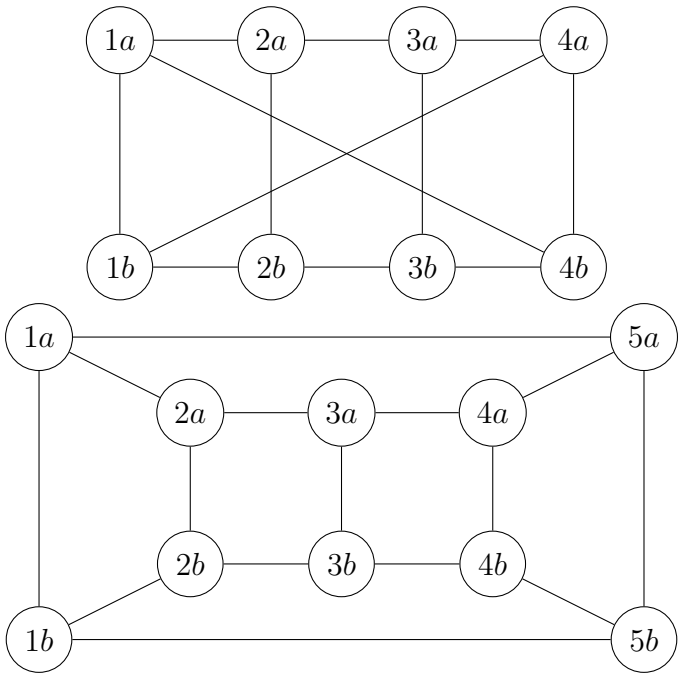
Obr. 1.3: Príklad K_5 grafu

Graf G nazývame kompletný bipartitný ak je bipartitný, $V = V_n \cup V_m$ a $E = \{v_nv_m | v_n \in V_n, v_m \in V_m\}$, $|V_n| = n, |V_m| = m, n \geq 1, m \geq 1$. To znamená, že každý vrchol z jednej množiny susedí s každým v druhej. Označujeme ho $K_{n,m}$ a príklad jeho zobrazenia máme na obrázku 1.4.



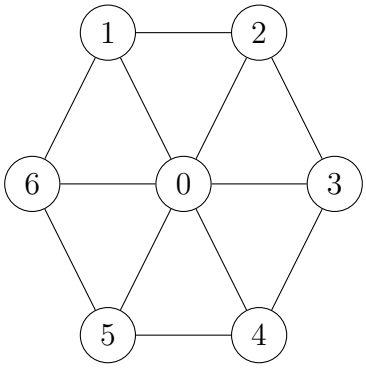
Obr. 1.4: Príklad $K_{3,2}$ grafu

Graf G nazývame rebrík ak $V = \{v_{1a}, v_{1b}, v_{2a}, v_{2b}, ..., v_{na}, v_{nb}\}, n \geq 2$, $E = \{v_{ia}v_{ib} | 1 \leq i \leq n\} \cup \{v_{ic}v_{i+1c} | c \in \{a, b\}, 1 \leq i < n\}$. Prekrížený rebrík ešte obsahuje hrany $v_{1a}v_{nb}$ a $v_{1b}v_{na}$ a neprekrížený rebrík ešte obsahuje hrany $v_{1a}v_{na}$ a $v_{1b}v_{nb}$. Zobrazenie prekríženého rebríka s ôsmimi vrcholmi a neprekríženého rebríka s desiatimi vrcholmi máme na obrázku 1.5



Obr. 1.5: Príklad prekríženého a neprekríženého rebríka

Graf G nazývame kolesom ak $V = \{v_0, v_1, v_2, ..., v_n\}$ a $E = \{v_1v_2, v_2v_3, ..., v_{n-1}v_n, v_nv_1\} \cup \{\forall i, 1 \leq i \leq n : v_0v_i\}$, kde $n \geq 3$. Zobrazenie kolesa s $n = 6$ máme na obrázku 1.6.



Obr. 1.6: Príklad kolesa

1.1.2 Farbenie grafov

Hranové farbenie grafu je zobrazenie $c : E \to S$, kde E je množina hrán grafu a S je množina farieb. Regulárne farbenie je také farbenie, ktoré priradí susedným hranám

inú farbu. Hranové $k-$farbenie je také farbenie, kde $S = \{1, 2, ..., k\}$. Chromatický index $\chi'(G)$ je najmenšie k také, že graf G má hranové regulárne $k-$farbenie.
Analogicky definujeme vrcholové farbenie, kde chromatické číslo $\chi(G)$ je najmenšie k také, že graf G má vrcholové regulárne $k-$farbenie. Blokové farbenie je také farbenie, kedy definujeme bloky, množiny, farieb, ktoré môžu byť priradené hranám susedné s jedným vrcholom. Blokové hranové farbenie sa dá aplikovať iba na grafy, ktoré sú regulárne. Napríklad ak máme bloky definované: $\{\{1, 2, 3\}, \{2, 3, 4\}\}$, nemôžu z jedného vrchola vychádzať farby jeden, dva a štyri.

1.2 Požiadavky na editor

Cieľom ročníkového projektu a bakalárskej práce bolo vytvoriť grafický editor pre vyučujúcich ako náhradu za papier a pero. Vytvoriť digitálne prostredie, kde si budú vedieť efektívne vytvoriť niekoľko grafov, rôznych či rovnakých a pozorovať ich vlastnosti. Bolo nutné, aby sa dalo efektívne vytvárať a pracovať aj s väčšími grafmi, skladať menšie do väčších a získavať výstupy, ktoré prijímajú aj inšie grafické editory. Tieto výstupy sú často dlhé a náročné na to, aby sa človek nepomýlil v indexoch, preto to náš editor robí za nás.

1.3 Existujúce editory

Existuje niekoľko programov na tvorbu a editovanie grafov. Väčšina z nich je však zameraná na UML grafy alebo na tvorbu rôznych typov diagramov, zobrazujúcich ľubovoľné štruktúry a dáta. Programy, zamerané na prácu s matematickými grafmi, sú väčšinou jednoduchšie a neobsahujú implementáciu algoritmov. Jediné grafické editory s algoritmami sa mi podarilo nájsť Graph Online[1], ktorý je vo forme webovej aplikácie a je dostupný zadarmo alebo Graph Engine Service[2] od Huawei, ktorý má iba platenú verziu.

Za spomenutie taktiež stojí knižnica NetworkX[3] pre Python. Možnosti, ktoré NetworkX ponúka sú rozsiahle, avšak sám o sebe neobsahuje možnosť vizualizácie grafov. Na grafické zobrazenie sa používa knižnica Matplotlib[4].
Ďalšie ľahko dostupné editory, skôr diagramov ako grafov, sú yEd - graph editor[5] a zoomcharts - graph editor[6]. Tieto editory ponúkajú rozsiahlu radu funkcií, ktoré pomáhajú presnejšej vizualizácii dát a štruktúr v diagramoch, čo je pre prácu s matematickými grafmi zbytočné, respektíve nepoužiteľné.

Náš editor sa sústredí najmä na efektivitu manipulácie, tak, aby užívateľ, čo najrýchlejšie a najjednoduchšie vytvoril ním žiadúci graf. O to, ako vyzerá vizuálne rozhoduje len malé množstvo nastavení.

1.3.1 Graph Online

Graph Online je jednoduchá webová 2D aplikácia, ktorá funguje v štyroch modoch. "Default", v ktorom sa užívateľ dokáže pohybovať v priestore a hýbať prvkami grafu. "Add vertex", v ktorom sa dajú pridávať vrcholy, ktoré sú indexovateľné podľa rôznych základných enumerátorov alebo sa dajú nazvať ľubovoľne. "Connect vertices", v ktorom sa dajú vytvárať hrany, buď orientované, alebo neorientované, taktiež sa im dá dať váha a pomenovanie. "Remove object", v ktorom sa dá odstrániť hrana, alebo vrchol s všetkými hranami z neho vychádzajúcich. Taktiež sa dá nastaviť farba pozadia, vrcholu a hrany. Algoritmy sú tam základné aj pokročilé, no z farbenia tam je iba algoritmus na minimálne vrcholové farbenie, bez parametrov. Taktiež jeho veľkým nedostatkom je, že sa nedá pracovať s viacerými prvkami naraz a ani kopírovať celé časti grafu.

Narozdiel od Graph Online, náš editor ponúka hranové farbenie grafov, kde si človek bude môcť zvoliť navyše možnosti, ako napríklad použitie konkrétnej farby na danú hranu alebo voľbu, že chceme konkrétne dve hrany rovnakej alebo rôznej farby. Graph Online síce ponúka väčšie množstvo algoritmov, no tvorba samotného grafu je zdĺhavá, najmä ak chce užívateľ pracovať s viacerým, alebo väčšími grafmi súčasne.

1.3.2 NetworkX, Matplotlib

NetworkX obsahuje veľké množstvo štruktúr pre rôzne typy grafov. Má implementované veľa základných a pokročilých algoritmov na prácu a analýzu grafov ako aj na prácu s maticami. Má taktiež implementované veľké množstvo generátorov. No stále je to iba knižnica pre Python a nie editor.

Matplotlib ponúka možnosť vizualizácie grafov, no je skôr orientovaný na grafy, ako reprezentáciu dát, ako na matematické grafy. My ponúkame čisto grafický editor, to znamená, že okrem vloženia grafu sa s grafom manipuluje čisto grafickým prostredím, a teda, aj algoritmy sa zobrazujú aj vizuálne. Takýto editor má tú výhodu, že je oveľa jednoduchší na naučenie a prácu, na veľa funkcií sa dá prísť intuitívne bez toho, aby užívateľ hľadal v dokumentácii či daná štruktúra ponúka ním žiadanú funkcionalitu.

1.4 Json formát

Json [10] formát sa využíva na rôzne účely. Je využívaný najmä pri tvorbe internetových aplikácií, no my ho budeme tiež využívať, a to práve na pamätanie si informácií o grafe.

Json alebo JavaScript Object Notation je štandardný formát využívaný na zápis dát. Json súbor reprezentuje buď hodnotu, pole alebo objekt. Hodnota môže byť buď slovo, číslo, objekt, pole, `true`, `false` alebo `null`. Pole je zoznam hodnôt a objekt je neusporiadaná množina dvojíc. Dvojicu tvorí kľúč a hodnota. Kľúč je vždy slovo.

Pomocou takejto definície dokážeme ukladať stromy dát. Vnorenie dokáže byť ľubovoľne veľké. Na obrázku 1.7 uvádzame príklad možného json súboru.

```
{
  "firstName": "John",
  "lastName": "Smith",
  "isAlive": true,
  "age": 27,
  "address": {
    "streetAddress": "21 2nd Street",
    "city": "New York",
    "state": "NY",
    "postalCode": "10021-3100"
  },
  "phoneNumbers": [
    {
      "type": "home",
      "number": "212 555-1234"
    },
    {
      "type": "office",
      "number": "646 555-4567"
    }
  ],
  "children": [
    "Catherine",
    "Thomas",
    "Trevor"
  ],
  "spouse": null
}
```

Obr. 1.7: Príklad obsahu Json súbora

1.5 Unity - Game Engine

Editor vyvíjame v Unity[8]. Unity je vývojové prostredie, teda editor, primárne určené na tvorbu 2D a 3D hier. Ponúka možnosti na vizuálnu tvorbu aplikácií, zabezpečuje vykresľovanie grafiky, má implementovaný fyzikálny engine, simuluje pôsobenie síl, ako

napríklad pohyb a gravitáciu, zvukový engine a podporuje skriptovanie. Skriptovanie je možnosť vytvoriť si vlastné triedy na pokročilé programovanie. Programovacím jazykom je C# a využíva sa editor zdrojového kódu Visual Studio Community.

Pre vývoj v prostredí Unity sme sa rozhodli, pretože za nás vďaka jeho funkciám, vyrieši niekoľko vecí, ako napríklad zobrazovanie objektov, ponúka nám základné možnosti na interakciu s užívateľom, pokročilejšie nástroje na prácu s užívateľským rozhraním (UI), a mnoho ďalšieho. Vďaka týmto možnostiam, môžeme viac práce venovať implementácii ďalších funkcií editora. Naším hlavným cieľom je zhotoviť editor grafov, ktorý sa bude používať pohodlne a efektívne.

1.5.1 Triedy a objekty v Unity

Aby sme vysvetlili niektoré implementačné prvky, vysvetlíme si ešte, ako pracuje samotné Unity. Unity pri spustení nami vytvorenej aplikácie, najprv inicializuje scénu a následne objekty v nej. To aká scéna sa má inicializovať prvá sa nastavuje pri kompilácii. Teda dá sa povedať, že táto scéna a všetky objekty v nej, sú akým si vstupným bodom Unity aplikácie. To aké objekty sa nachádzajú v scéne určuje programátor pomocou grafického rozhrania Unity Engine.

Inštancie tried sú buď triedy, ktoré sú priradené k objektom v scéne a teda dedia z triedy `MonoBehaviour`, alebo existujú ako štandardné premenné v iných triedach, alebo sú statické. Triedy priradené objektu nazývame komponent objektu. Tieto komponenty sú buď nami vytvorené skripty alebo Unity ponúka veľké množstvo komponentov. `MonoBehaviour` je základná trieda Unity, a ak chceme priradiť skript objektu, musí nevyhnutne dediť z `MonoBehaviour`. Táto trieda ponúka základné funkcie ako `Awake`, `Start` a `Update`. `Awake` funkcia sa volá pri inicializácii objektu. `Start` funkcia sa zavolá pred prvým volaním `Update` funkcie a iba vtedy ak je skript povolený na objekte a funkcia `Update` sa volá pri každom zobrazení snímku, teda za sekundu je to presne fps (frames per second) krát. `Update` sa tak ako `Start` zavolá iba tie snímky kedy je skript povolený na objekte. To, že či je skript povolený alebo zakázaný sa dá nastaviť pomocou kódu aj pomocou editora. `MonoBehaviour` má referenciu na objekt, na ktorom sa skript nachádza. Objekty sú základnou stavebnou jednotkou Unity aplikácií. Zobrazujú sa v scéne, musia mať komponent `Transform`, ktorý udržuje informáciu o ich umiestnení, a môžu mať priradenú grafiku, a teda sa dajú snímať kamerou. Kamera je špeciálny komponent objektu. Pomocou tried, ktoré im priradíme definujeme ich správanie. Objekty sa vytvárajú ak sú v scéne pri spustení aplikácie alebo pomocou kódu za behu.

V Unity sa často využíva možnosť vkladania jednotlivých objektov do seba. Takto sa dajú nastavovať relatívne pozície oproti rodičovi. Táto vlastnosť Unity je hlavne potrebná a využívaná pri tvorbe hier alebo pri vytváraní používateľského rozhrania.

1.5.2 Označenia a vrstvy

Každý objekt má v základe tieto dve premenné, a to sú `tag` a `layer` . Možné hodnoty definujeme v rozhraní Unity a k premenným máme prístup vďaka `MonoBehaviour`. `Tag` je slovo, ktoré môžeme priradiť objektu a tým definovať čo to je. Týmto spôsobom označíme, že objekt patrí do tej skupiny. `Layer`, po slovensky vrstva, sa využíva na rozdelenie fyzického priestoru, podľa jeho vlastností. Môžeme definovať, ktoré vrstvy nemajú medzi sebou interagovať alebo ktoré majú byť viditeľné kamerou. Vrstvy v Unity sa využívajú rôznymi spôsobmi, primárne na oddelenia objektov používateľského rozhrania a ostatných.

1.5.3 Prefab

Pri tvorbe z kódu sa štandardne využívajú prefaby. Prefab je špeciálny typ objektu, ktorý je uložený v pamäti aplikácii, a teda neexistuje v scéne a môže byť priradený ako premenná. Pomocou tejto premennej môžeme potom v kóde vytvoriť klony tohto prefabu ako objekty v scéne.

1.5.4 Detekcia kolízií a Raycasting

Na to, aby sme si vysvetlili, čo je to raycasting, potrebujeme si objasniť, čo sú to `Collider`-y v Unity. Zatiaľ, čo existujú špeciálne Unity komponenty, ktoré nám slúžia na vykresľovanie grafiky, tie ešte nedefinujú fyzické telo objektu. Na to, aby sme definovali fyzické telo objektu, musíme mu priradiť komponent `Collider 2D`. Ten môže mať buď jednoduché tvary ako štvorec alebo kruh, alebo môže do určitej miery opisovať grafiku objektu.

Komponent, ktorý zisťuje či do `Collider 2D` vošiel iný `Collider 2D` a teda vznikla kolízia, sa volá `Rigidbody 2D`. Ten pri detekcii kolízie pošle správu (funkcia `SendMessage`), ktorú následne dostanú všetky komponenty. Komponent môže implementovať funkciu s daným názvom, a teda pri prijatí správy sa táto funkcia zavolá. Takto môžeme definovať správanie objektu pri kolízii s iným objektom.

Pri raycastingu sa volá špeciálna Unity funkcia `Physics2D.Raycast`, ktorá vytvorí lúč z miesta pôvodu (origin) v danom smere (direction). Lúču môžeme priradiť ďalšie vlastnosti, ako napríklad vzdialenosť, do ktorej má siahať, alebo masku1.5.2, pomocou ktorej môžeme vylúčiť nežiadúce vrstvy `Collider`-ov. `Physics2D.Raycast` vracia struct typu `RaycastHit2D`, ktorý okrem iného obsahuje referenciu na `Collider2D`, ktorý lúč trafil, a ten obsahuje referenciu na trafený objekt. Raycasting sa využíva aj na interakciu s UI.

1.5.5 UnityEvent

`UnityEvent` je implementácia návrhového vzoru pozorovateľ v Unity. Vytvoríme si novú triedu, ktorá derivuje od `UnityEvent`. Následne premennej tejto triedy môžeme pridávať a odoberať počúvajúcich a taktiež môžeme zavolať, aby sa všetci počúvajúci vykonali.

1.5.6 Unity knižnice, ktoré budeme požívať

Vrámci Unity budeme používať knižnicu OR-Tools[7], ktorá implementuje triedy ktoré hľadajú riešenia pre rôzne optimalizačné problémy, napríklad lineárne programovanie, SATsolver alebo CPsolver. Táto knižnica je implementovaná v C++, no dá sa používať aj s .NET, Java a Python. My používame .NET s Visual Studio. Túto knižnicu budeme využívať pri hľadaní farbenia pre graf.

Ďalšou knižnicou, ktorú budeme využívať je DOTween. Tá nám ponúka možnosť animovať objekty priamo z kódu.

Kapitola 2

Funkcionality editora

Na začiatku je potrebné vysvetliť, aké rôzne funkcionality náš editor ponúka. Funkcionality rozdelíme na tie, ktoré sú predmetom ročníkového projektu a na tie, ktoré sú predmetom bakalárskej práce. Predmetom ročníkového projektu bolo postaviť aj samotný základ editora.

2.1 Ročníkový projekt - funkcionality

Základ aplikácie sú samotné funkcie editora grafov. To sú vytvorenie ľubovoľného grafu a modifikácia grafu, ako odpojenie a pripojenie hrany k vrcholu. Tieto funkcionality nám stačia na vytvorenie ľubovolného grafu z logického hľadiska, ale keďže je to grafický editor potrebujeme aby sa aj vizuálne dal vytvoriť ľubovolný graf. To docielime tým, že sa budú dať prvky grafu posúvať a hrany nebudú len rovné čiary, ale budú sa dať nakresliť ako krivky.

Editor sme založili na tom, aby sa pohodlne pracovalo s viacerými prvkami naraz. Preto zámerom viacero funkcionalít je práve to.

Od editora taktiež vyžadujeme Undo/Redo funkcionalitu a ďalšie, ktorým sa budeme venovať v samostatných kapitolách.

2.1.1 Kopírovanie

Pri tvorbe grafov sa nám často stáva, že graf obsahuje časti, ktoré sa nám opakujú alebo, že potrebujeme jeden graf viac krát vedľa seba. Preto jednou prvých implementovaných častí, bolo možnosť kopírovania celých častí grafu.

2.1.2 Lokálna databáza

Editor ponúka možnosť uloženia si vytvorených grafov do súkromnej databázy. Z tejto databázy sa potom dajú získať, ako vytvorené grafy alebo vieme dostať ich textové

výstupy, ktoré použijeme buď ako vstup pre inšie aplikácie alebo pre náš editor. Vďaka tejto databáze si vieme udržovať pracovnú plochu čistú a zároveň, mať prístup ku grafom, ktoré sme vytvorili dávnejšie.

2.1.3 Rotácia grafu

Posúvaním častí grafu dokážeme síce získať vizuálne ľubovolný graf, ale je praktické keď sa dajú, hlavne väčšie časti grafu posúvať aj rotovaním, tak dokážeme tvoriť symetrické grafy oveľa rýchlejšie a pritom si udržať pekné rozostavanie prvkov.

2.2 Bakalárska práca - funkcionality

Súčasťou bakalárskej práce boli funkcionality spomínané nižšie a taktiež refaktorizácia niektorých častí editora, konkrétne základu editora a Undo/Redo funkcionality..

Funkcionality pre pohodlnejšiu prácu sme pridali:

- udržanie pracovnej plochy pri vypnutí aplikácie.
- označenie celých komponentov po kliknutí
- pridanie rotácie pri kopírovaní
- kontrakcia hrany
- spojenie dvoch visiacich hrán do jednej
- nastavenia aplikácie

2.2.1 Hranové farbenie

Jednou z funkcionalít na získanie informácie o grafe je farbenie. Zafarbenie grafu si vedieme definovať rôzne, to znamená, že aplikácia ponúka možnosti pre farbenie, čo nám umožňuje dostať rôzne farbenia jedného grafu. Konkrétne tieto možnosti sú zadefinovanie koľkými farbami chceme nafarbiť graf. Čiže aplikácia nezískava chromatický index grafu, ale hľadá či existuje nami požadované $k-$farbenie. Máme možnosť výberu konkrétnych hrán, ktoré budeme nazývať špeciálne. Týmto špeciálnym hranám vieme následne určiť, že či majú mať konkrétnu farbu alebo, že nech majú ľubovolné dve inšiu farbu, alebo rovnakú. Poslednou možnosťou je voľba blokového farbenia. Táto možnosť sa týka iba kubických grafov.

Rôzne farbenia vieme uložiť tiež do lokálnej databázy a získať výstup aj so zafarbením, čo je opäť užitočné ako vstup pre inšie grafové aplikácie.

2.2.2 Generovanie

V aplikácii chceme generovať štandardné typy grafov a to konkrétne kružnice, kompletné grafy, kompletné bipartitné grafy, ktoré majú oba parametre rovnaké, prekrížené a neprekrížené rebríky a kolesá.

Aplikácia nám taktiež ponúka možnosť vygenerovania grafu z JSON súbora a to buď pomocou cesty k textovému súboru s JSON obsahom alebo priamo zadefinovania JSON-u. Tento formát dokážeme získať aj ako jeden z formátov výstupu z grafov v súkromnej databáze, a je unikátny pre našu aplikáciu.

Generovanie má zmysel v našej aplikácii ako ďalšia možnosť pohodlnejšieho tvorenia cieľového grafu. Napríklad ak vieme, že si chceme vytvoriť graf, ktorý sa nelíši priveľa od K_5 grafu, tak namiesto toho, aby sme ho tvorili vrchol po vrchole a hranu po hrane, máme možnosť vytvorenia K_5 grafu, ktorý následne už len upravíme.

Kapitola 3

Návrh

Návrh aplikácie vyplýva z funkcionalít, ktoré chceme, aby boli v editore implementované. Teraz si popíšeme, čo všetko bude treba implementovať a pozrieme sa zvysoka na architektúru aplikácie.

Kód editora sa dá rozdeliť do štyroch veľkých častí:

- práca s grafom
- interakcia s užívateľom
  - pomocou UI
  - pomocou klávesnice a myši
- manažment aplikácie.

Aplikáciu som navrhla tak, aby stála na troch základných triedach:

- `Settings` (manažment aplikácie a interakcia s používateľom pomocou klávesnice a myši)
- `GraphManager` (práca s grafom)
- `UIManager` (interakcia s užívateľom pomocou UI)

Tieto triedy sú dostupné ako premenné v abstraktnej triede `AccesBehaviour`, a teda ak potrebujeme v nejakej triede prístup k niektorej zo základných tried, tak určíme, aby dedila od `AccesBehaviour`. Celá `AccesBehaviour`, sa v podstate správa podobne ako návrhový vzor singleton pre viacero tried. Tieto tri základné triedy sa musia, teda nachádzať na objekte v scéne, aby sa do `AccesBehaviour` priradili pri spustení scény.

Pod jednotlivé základné triedy sa budú dať rozdeliť všetky ostatné triedy tak, že budú priamo pod ne spadať alebo veľmi úzko s nimi súvisieť.

Editor funguje tak, že sa pracuje s množinou prvkov, ktoré sú graficky zvýraznené, používame termín označené. Hrana je reprezentovaná jej koncami, stredom a krivkou,

ktorá spája tieto tri body. Označenie hrany je reprezentované zvýraznením jej stredu. Označenie vrchola spôsobí označenie všetkých k nemu pripojených koncov hrán.

Jednotlivé prvky grafu, teda vrchol, stred hrany a koniec hrany, budú implementované pomocou návrhového vzoru kompozit a budú spadať pod `GraphManager`. Ten bude vedieť pracovať s nimi tak, aby z nich získal potrebné informácie pre ostatné triedy, alebo ich bude upravovať podľa toho ako bude treba.

Najčastejšie budeme využívať samotné funkcie editora, ako sú tvorba a hýbanie s časťami grafu, a preto je najdôležitejšie, aby to, čo užívateľ, robí väčšinu času v aplikácii bolo pohodlné, jednoduché a intuitívne.

3.1 Štandardný výstup

Definíciu štandardného formátu sme získali od školiteľky. Skladá sa z dvoch častí, zadefinovanie susedov a farbenie. Zadefinovanie susedov vyzerá tak, že jeden riadok obsahuje index vrchola a ďalej indexy vrcholov, s ktorými susedí (napríklad: i: x y z). Indexy sú reprezentované prirodzenými číslami. Ak z nejakého vrchola vychádza visiaca hrana, tak tá je reprezentovaná -1. Farbenie sa nachádza v druhej časti a jeho formátovanie je také, že každej hrane vypísanej v prvej časti, zodpovedá v druhej časti jedno číslo, ktoré reprezentuje index farby, ktorým je táto hrana zafarbená. Teda počet riadkov a čísel v riadku za dvojbodkou je v oboch častiach rovnaký.

3.2 Json výstup

Tento výstup/vstup je zadefinovaný jedinečne pre náš editor a nedáva zmysel inším programom a ani človeku, ktorý nepozná implementačné pozadie.

Využívame ho pri ukladaní do lokálnej databázy alebo ako jeden z výstupov grafu. Keďže je náš editor grafický tak informáciu o grafe potrebujeme ukladať ako relatívne pozície vzhľadom na pomyselný stred. Ďalej v ňom uchovávame zoznam susedov, pre tvorbu štandardného výstupu. Taktiež sa v ňom môže nachádzať informácia o farbení, ale to nieje nutnosť, keďže nie každý uložený graf musí mať zadefinované farbenie.

3.3 Používateľské rozhranie (UI)

Pri návrhu UI to nebolo také jednoduché. Niektoré časti boli plne ovplyvnené špecifikáciou, ako napríklad panel farbenia, a niektoré neboli špecifikované vôbec, takže som sa opäť snažila použiť štandardný formát s hornou lištou.

UI je veľkou časťou každej aplikácie, a preto jeho manažér je jednou zo základných tried. On má prístup ku všetkým ostatným panelom. Paneli, ktoré bolo potrebné imple-

mentovať sú panel databázy, generovania, výstupový panel, panel farbenia a nastavení. Je potrebné, aby bolo jednoduché pridať ďalšie paneli pre ďalšiu potrebu, prípadne pri paneloch generovaných kódom, aby sa pri pridávaní ďalších častí, ostatné prispôsobili. Zároveň však treba aby trieda nemala príliš veľa kódu. Preto má architektúra UI silnú stromovú štruktúru. Každý objekt rieši iba seba a generuje svoje deti. Všetky podstatné informácie sa dostávajú na potrebné miesto skrz všetkých predkov. Týmto spôsobom generujeme zložito na seba nadväzujúce časti UI s tým, že sa ovplyvňujú, velice tolerantným spôsobom. Strom skriptov, ktoré spadajú pod UI, zobrazujúci to, ktorý skript má na starosti aké ďalšie, čo sa generovania objektov týka, máme popísané v neskoršej kapitole 4.5.1.

3.4 Interakcia s používateľom pomocou klávesnice a myši

Keďže je to editor, tak na to, aby boli implementované všetky funkcionality je nutná interakcia s používateľom ako pomocou UI, tak aj pomocou klávesnice a myši. Klávesnica a myš je primárne využívaná na modifikovanie grafu a použitie UI. Zatiaľ čo UI je využívané na prácu okolo už vytvorených grafov alebo na generické vytváranie grafov.

Pri implementácii interakcie pomocou klávesnice a myši bolo nutné, aby sa dalo intuitívne narábať s grafom, preto som sa nechala inšpirovať inšími grafickými editormi, ako napríklad samotným Unity Game Engine či editormi na tvorbu grafiky. Pri tvorbe aplikácií je dôležité aby sa programátor nesnažil znova vynájsť koleso, to znamená, že keďže užívatelia bežne používajú, v iných editoroch, skratku Ctrl+S na uloženie, tak dáva zmysel to použiť ako skratku na uloženie grafu do databázy.

3.4.1 Módy aplikácie

Keďže sme kládli veľký dôraz na to, že chceme pracovať s viacerými prvkami naraz, tak sme prácu v editore rozdelili do niekoľkých modov. Tri hlavné sa nazývajú “SINGLE“, “ADD“ a “PLURAL“ mód. Postupne si predstavíme všetky a začneme práve “ADD“ modom.

V “ADD“ móde sa dajú pridávať samostatné prvky grafu, teda vrcholy a hrany.Označený prvok je vždy jeden a to práve ten, ktorý sme bezprostredne predtým vytvorili a preto pri odstraňovaní dokážeme odstrániť iba ten.

“SINGLE“ mód nám umožňuje základnú prácu s jedným prvkom grafu, presúvať ho, rotovať okolo neho komponent grafu a odpájať a pripájať existujúce hrany. Hrany sú buď rovné a udržujú si polohu v priestore medzi dvoma vrcholmi, ktoré spájajú a stred sa udržuje v strede medzi nimi, alebo sa dajú deformovať potiahnutím ich stredu,

kedy nadobudnú tvar Bézierovej krivky[1]. Vtedy sa stred hrany zakotví na mieste a jeho polohu neovplyvňuje poloha koncových bodov hrany. Hrana musí mať vždy pripojený aspoň jeden vrchol, a teda nedokáže existovať samostatne. Ak nám nejakým spôsobom takáto hrana vznikne, odstráni sa automaticky z pracovnej plochy. Dajú sa odstraňovať označené prvky grafu. Pri označení hrany sa nám zobrazí inak skryté tlačidlo, ktoré slúži na kontrakciu hrany.

"PLURAL" mód nám slúži na prácu s viacerými prvkami grafu. V tomto móde sa dajú označovať viaceré prvky grafu buď osobitným klikaním na požadované prvky alebo pomocou "selectora", ktorý má tvar obdĺžnika[2]. Rotácia sa uskutočňuje tak, že sa zistí pomyselný stred všetkých označených prvkov a až potom sa rotuje graf, tvorený komponentami s aspoň jedným označeným prvkom, okolo daného pomyselného stredu. Dokážeme odstraňovať označené prvky grafu.

Na kopírovanie celých častí grafu nám slúži "HOLD" mód. Doň sa môžeme dostať rôznymi spôsobmi, keď kopírujeme z plochy, z databázy alebo keď generujeme graf. V "HOLD" móde dokážeme kopírovaný graf dostať na ľubovoľné miesto na ploche, teda potrebujeme hýbať kamerou, v akejkoľvek rotácii.

Potom máme ešte niekoľko vedľajších modov, a to napríklad, keď sa užívateľ nachádza v nejakom z User Interface paneli alebo keď vyberá možnosti farbenia.

Tieto módy implementujeme pomocou návrhového vzoru stratégia a bude ich spravovať trieda `Settings`.

[1] Bézierová krivka[9] troch bodov, kvadratická Bézierova krivka, je množina bodov $C = \{(1-t)^2 P_0 + 2t(1-t)P_1 + t^2 P_2, t \in < 0, 1 >\}$, kde P_0 a P_2 sú koncové body a P_1 je bod, vzdialený dvojnásobne od stredu spojnice koncov hrany ako stred hrany.

[2] Obdĺžnik má rovnaký charakter ako hromadný výber na pracovnej ploche Windowsu.

Kapitola 4

Implementácia

V tejto kapitole si vysvetlíme implementáciu jednotlivých funkcionalít aplikácie.

4.1 Reprezentácia grafu

Graf a jeho prvky máme implementované pomocou upraveného návrhového vzoru kompozit. Diagram tried grafových častí, bez funkcií jedinečné pre danú triedu máme na obrázku 4.1.

Uzol nám reprezentuje `SubGraph` a listy sú triedy `Vertex`, `EdgeMid` a `EdgeEnd`.

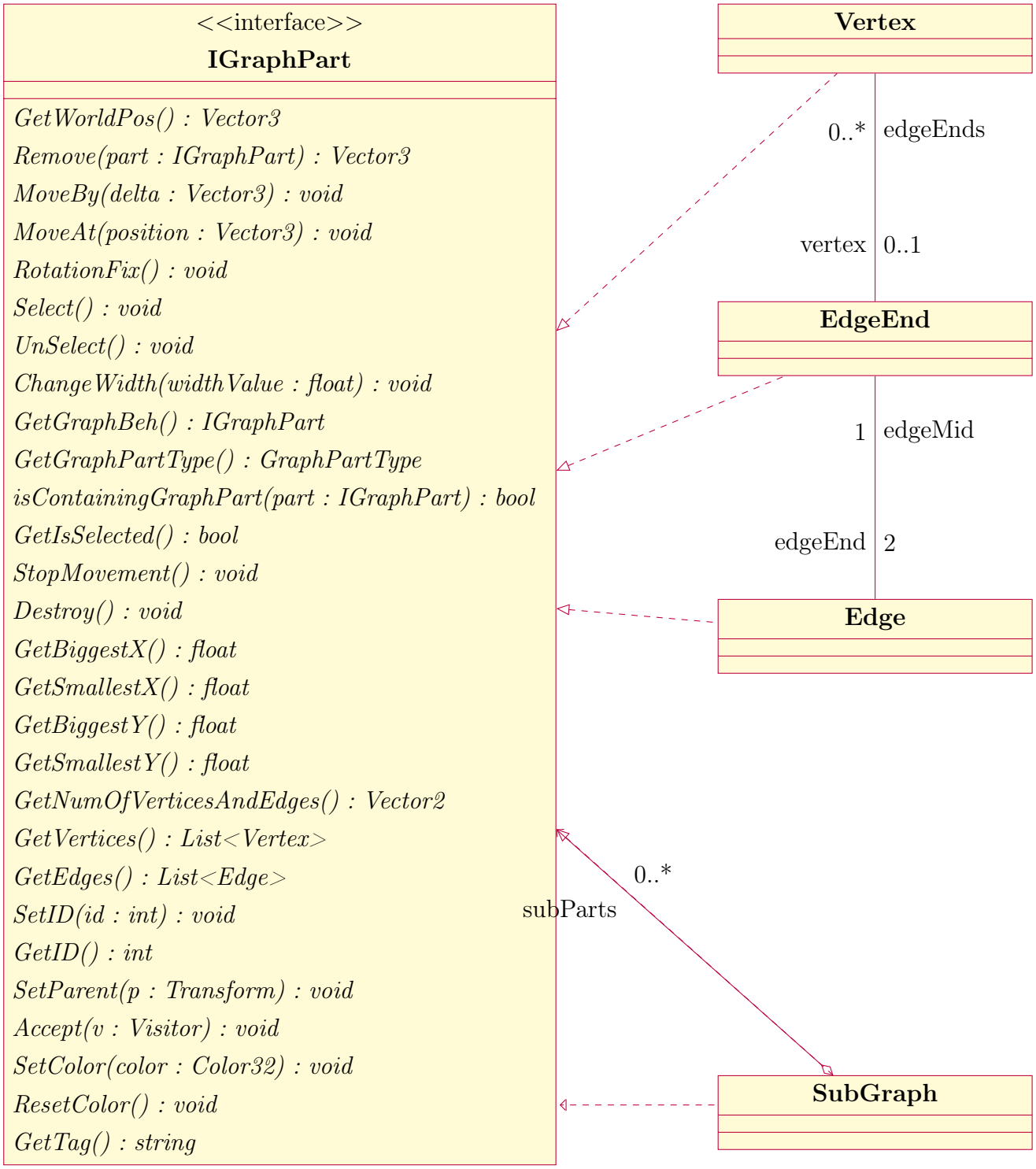
Upraveného znamená, že v `SubGraph` subParts nemáme nikdy ďalší `SubGraph`. Taktiež `EdgeMid` sa nám premenlivo správa ako uzol s dvoma prvkami, čo sú konce hrany a `Vertex` si taktiež uchováva informáciu o `EdgeEnd`-och, ktoré sú k nemu pripojené. Dané triedy obsahujú ďalšie unikátne metódy. Keďže existuje `EdgeEnd` aj ako samostatný prvok a, aj ako prvok `Vertex`-u, aj `EdgeMid`-u, treba ošetriť, aby nedostával násobne tú istú informáciu. Tento problém nám vzniká, kvôli tomu, že potrebujeme, aby sme mohli pracovať aj s visiacimi hranami, kedy `EdgeEnd` je samostatný a nezávislý, ale keď má pripojený vrchol túto samostatnosť mať nesmie.

Vrchol má jednoduché správanie oproti hrane. Ovplyvňuje ho iba sám on a má jednoduché správanie i vykresľovanie. Keďže je hrana ako objekt a aj jej kód dokopy, mnohonásobne komplikovanejší, budeme sa venovať najbližších pár podkapitol aké problémy sme museli vyriešiť pri jej implementácii.

4.1.1 Priraďenie hrany vrcholu

Najprv sa budeme venovať tomu ako implementujeme priradenie hrany vrcholu.

Koniec hrany môžeme odobrať vrcholu iba v "SINGLE" móde. To, že či koniec hrany patrí alebo nepatrí nejakému vrcholu sa vždy zisťuje keď dokončíme pohyb daného konca. Presnejšie sa to zisťuje o dve snímky potom, a to, pretože na zisťovanie toho či je pri danom konci hrany vrchol, ku ktorému sa má pripojiť, využívame raycasting1.5.4.



Obr. 4.1: Diagram tried grafových častí

Raycasting zisťuje či v danom momente lúč koliduje s nejakým `Collider`-om. Ale keďže niekedy vytvárame objekty, v tej istej snímke, ako ich chceme aj umiestniť, potrebujeme, aby sa najprv vygenerovali, čo sa nedeje hneď, ale až medzi snímkami. Preto potrebujeme počkať s raycastingom, aby sa nám objekty stihli vygenerovať.

Aby sme zabezpečili, že sa to deje o dve snímky potom využívame číselnú premennú `toStopMovement`, ktorú nastavíme na jedna vo funkcii `StopMovement()`. V `Update` kontrolujeme hodnotu `toStopMovement`. Ak je jej hodnota jedna tak ju nastavíme na dva, a ak je jej hodnota dva vykonáme všetky funkcie, ktoré chceme, aby sa po zastavení konca hrany vykonali.

Výsledkom raycastingu môžu byť tieto možnosti. Buď sa zistí, že je hrana stále pri tom istom vrchole a len sa nastaví nech sa presunie presne na jej pozíciu alebo sa zistí, že je pri inom vrchole, kedy sa potrebuje odpojiť z predchádzajúceho a pripojiť k novému, alebo sa zistí, že nie je pri žiadnom vrchole, a teda ak bol predtým pripojený, tak sa musí odpojiť. Jednou z vecí, čo sa deje pri odpojení hrany je, že sa povie stredu hrany, aby skontrolovala či sa nemá zničiť. Hrana sa má zničiť vtedy, keď nemá ani na jednom konci vrchol.

Taktiež sa nám farba krivky hrany mení v závislosti od toho či je hrana visiaca alebo nie. To nám pomáha jednak jednoznačne vidieť, kde máme visiace hrany, ale taktiež pri tom, keby sme omylom hranu nepripojili.

4.1.2 Spájanie voľných hrán do jednej

Potom, čo sa pri zastavení skontroluje vrchol treba na to mimo toho náležite zareagovať. Treba, aby si stred hrany skontroloval či má alebo nemá byť zakotvený a potom ak má koniec hrany vrchol, treba aby sa odstránil z označených prvkov. To nám slúži na to, aby sme s koncom hrany, keď je pripnutý k vrcholu, dokázali pracovať iba vtedy, keď pracujeme s vrcholom alebo, keď ho chceme odopnúť z vrchola. Ak však koniec hrany vrchol nemá, treba zistiť či sa nenachádza v okolí iný koniec hrany, s ktorým ho chceme spojiť. Týmto spôsobom sme implementovali spájanie dvoch voľných hrán do jednej.

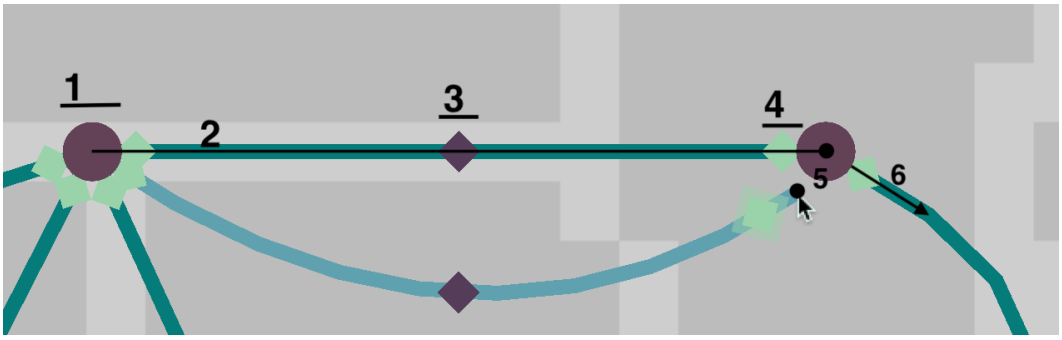
Každý `EdgeEnd` má implementované dve funkcie, `OnTriggerEnter2D(Collider2D collision)` a `OnTriggerExit2D(Collider2D collision)`. Toto sú funkcie, ktoré reagujú na príslušnú správu1.5.4 od `Rigidbody 2D`-a. Vždy, keď do ich `Collider`-a vojde iný `Collider`, ktorý má na svojom objekte aj `EdgeEnd`, uložia si ho v množine. Následne pri zastavení pohybu sa táto množina prechádza a keď sa nájde koniec hrany, ktorý tiež nemá vrchol, tak sa tá druhá hrana zničí a kontrolovaný koniec sa pripojí k tomu vrcholu, z ktorého visela zničená hrana.

4.1.3 Kotvenie hrany a vykresľovanie krivky

Ako sme si už spomínali v návrhu, chceme dovoliť, aby sa hrana mohla deformovať do Bézierovej krivky. To znamená, že existuje v dvoch rôznych stavoch. Buď je hrana zakotvená, udržuje si stred medzi svojimi koncami a má tvar úsečky alebo nie je zakotvená, stred je na pevnom mieste a hýbe sa iba keď hýbeme priamo s ním. Niekedy chceme, aby pohyb konca hrany ovplyvňoval pohyb stredu a niekedy nie. Toto sme implementovali pomocou `UnityEvent`. V triede `EdgeEnd` máme implementovanú triedu `EdgeEndMoveEvent`, ktorá derivuje z `UnityEvent`. Na jej premennej voláme funkciu `Invoke()` pri oboch pohyboch (`MoveBy` a `MoveAt`). `EdgeMid` pri každom skončení pohybu či svojom alebo svojich koncov, kontroluje, či sa nachádza, s nejakou odchýlkou niekde medzi svojimi koncami. Ak áno pridá do počúvajúcich metódu, aby pri pohybe oboch svojich koncov, si udržoval pozíciu v strede. Táto metóda sa volá `LockedMove` a je v triede `EdgeMid`. Ak nie, tak odstráni z počúvajúcich, aby sa volala funkcia `LockedMove` a pridá funkciu `DrawCurve`. Takto keď má byť hrana zakotvená a bude sa hýbať iba jej koncom, bude to ovplyvňovať aj jej stred a, keď nebude zakotvená tak pohyb konca hrany bude ovplyvňovať iba vykresľovanie krivky.

Funkcia `DrawCurve` slúži na vykreslenie krivky. Krivka sa vykresľuje pomocou Unity komponentu `LineRenderer`. `LineRenderer` je požadovaný skript pre `EdgeMid` a slúži na to, aby vykreslil čiaru pomocou zoznamu bodov. `LineRenderer` má mnohé nastavenia, ktoré ponúkajú široké možnosti pre tvorbu liniek, no my využívame len základnú množinu nastavení, ktorú potrebujeme na to, aby sme vykreslili hranu. Základom je, že sa vytvorí desať bodov na Bézierovej krivke a tie sa potom pospájajú rovnými čiarami.

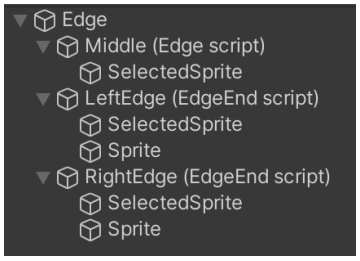
4.1.4 Štruktúra hrany



Obr. 4.2: Štruktúra hrany: 1 vrchol, 2 krivka hrany, 3 stred hrany, 4 grafika konca hrany, 5 pozícia konca hrany, 6 otočenie konca hrany

Na to, aby sa vyššie uvedené vlastnosti hrany aj pekne zobrazovali, má hrana špeciálnu štruktúru, ktorú máme zobrazenú na obrázku 4.2 a hierarchiu, ktorú máme zobrazenú na obrázku 4.3.

Ako sa dá vidieť na obrázku 4.2 pozícia konca hrany je posunutá oproti jeho grafike. To kvôli tomu aby sme mohli priradiť za pozíciu konca hrany priamo pozíciu vrchola a nemuseli vypočítavať posunutie od stredu podľa toho, ktorým smerom ide hrana.



Obr. 4.3: Hierarchia hrany, ako unity objektu

Rotáciu konca hrany nastavujeme tak aby smeroval k prvému bodu lomu hrany, čiže tam kam nám ide krivka hrany od vrchola. `EdgeEnd` má funkciu `TurnToVector(vector: Vector3)`, ktorá nám natočí koniec hrany smerom k danému bodu v priestore. Keďže je to 2D aplikácia, ale reálne ju vytvárame v 3D prostredí, meníme rotáciu iba v osi z, teda v tej, ktorá nám ide akoby do obrazovky. Os x a y nastavujeme na nulu. Výpočet rotácie sa dá vidieť naprogramovaný na obrázku 4.4.

```
public void TurnToVector(Vector3 vector){
    if (vector.y - transform.position.y < 0){
        transform.rotation = Quaternion.Euler(new Vector3(0, 0,
            180 + Mathf.Rad2Deg * Mathf.Asin(
                (vector.x - transform.position.x) /
                Vector3.Distance(vector, transform.position))));
    }else{
        transform.rotation = Quaternion.Euler(new Vector3(0,0,
            - Mathf.Rad2Deg * Mathf.Asin(
                (vector.x - transform.position.x) /
                Vector3.Distance(vector, transform.position))));
    }
}
```

Obr. 4.4: `TurnToVector(vector: Vector3)` z triedy `EdgeEnd`

Takúto hierarchiu objektu sme zvolili, pretože chceme, aby sme mali hranu pokope. Ale reálne sa nám skladá z troch oddelených častí, ktoré sa navzájom ovplyvňujú špeciálnym spôsobom, ktorý sa mení od závislosti k rôznym premenným. Takáto hierarchia nám umožňuje mať objekt pokope a zároveň logicky oddelene.

4.1.5 Dátové triedy

Pri implementácii využívame rôzne triedy, ktoré udržujú informáciu o grafe. Tieto triedy sú serializovateľné, to znamená, že majú pred definovaním triedy pridaný atribút `[Serializable]`. Vďaka tomu sa dajú jednoducho konvertovať na Json súbory. To

využívame pri implementácii databázy.

Tieto triedy majú využitie najmä pri kopírovaní a generovaní. Pre vrchol si pamätáme iba pozíciu a pre hranu si pamätáme tri pozície a či je zakotvená. Na zapamätanie informácií o hrane využívame ďalšiu dátovú triedu s názvom `EdgeInfo`. Mimo toho si pamätáme zoznam susedov a či máme aj informácie o farbení. Vo farbení si pamätáme indexy farieb podľa zoznamu susedov. V triede `SubGrafInfo` máme niekoľko konštruktorov, ktoré berú grafy a podľa nich uložia do premenných informácie. Zoznam susedov a farbenie využívame iba pre vytvorenie štandardného výstupu.

4.1.6 Tag objektov

Všetky objekty, ktoré majú na sebe skripty `Vertex` a `EdgeMid` majú `tag`1.5.2 nastavený na `Collectorable`. Objektom so skriptom `EdgeEnd` sa tento tag mení, a to konkrétne pri pridávaní a odoberaní vrchola koncu hrany. Tento tag potom slúži na to, že ostatné objekty ich berú ako samostatný prvok. Keď má objekt `EdgeEnd`-u nastavený tag na `Untagged`, čo znamená, že je pripojený k vrcholu, berú ho ako niečo čo patrí k vrcholu a teda ho neberú ako samostatný prvok. Tieto objekty si predstavíme postupne neskôr.

4.2 Práca s grafom

`GraphManager` má na starosti celú prácu s grafom, všetky úpravy a informácie robíme a získavame skrz neho. `GraphManager` si udržuje informáciu o všetkých vrcholoch, všetkých hranách a všetkých označených prvkoch, ktoré môžu byť ľubovoľného typu. Jeho súčasťou je aj príkazový vzor ako implementácia Undo/Redo funkcionality. Všetky jeho premenné a funkcie môžeme vidieť na obrázkoch 4.5 a 4.6.

Metódy `GraphManager`-a sa delia na tie, ktoré obsluhujú príkazový vzor tie, ktorých obsahom je len príkaz z príkazového vzoru a na tie, ktoré síce pracujú s grafom, ale príkaz nemajú, a teda využívajú iné príkazy alebo pre ne príkaz neexistuje a ich zmeny zachytí inší príkaz.

4.2.1 Undo a Redo

Veľkú časť Undo/Redo funkcionality som dorábala a refaktorovala vrámci prác na implementáciách funkcionalít spadajúc do bakalárskej práce. Táto funkcionalita je štandardne dostupná a očakáva sa, že bude k dispozícii v každom lepšom editore. Jej implementácia je však netriviálna a je úzko spätá s tým o aký editor sa jedná. Všetky triedy a rozhranie spadajúce pod príkazový návrhový vzor sa dajú vidieť v diagrame tried na obrázku 4.7.

Keďže príkazy pre zmenu pracovnej plochy sú jemnejšie ako niektoré reakcie na

GraphManager
+ vertexPrefab : Vertex + edgePrefab : GameObject - selected : SubGraph - allEdges : SubGraph - allVertices : SubGraph + verticesParent : Transform + edgessParent : Transform - rotator : Transform - commands : Stack<ICommand> - redoCommands : Stack<ICommand> - isUnduing : bool - isReduing : bool - prevPositions : List<Vector3> - hangingEdgesCorrection : bool
- Start() : void - Update() : void + Undo() : void + Redo() : void + DestroyAll(isSimple : bool) : void + Checkpoint() : void + DeleteLastCheckpointCommand() : ICommand + AddCheckpointCommand(command : ICommand) : void - NewCommand(command : ICommand) : void + AddToSelected(IGraphPart part) : void + RemoveFromSelected(IGraphPart part) : void + RemoveAllFromSelected() : void + MoveAllInSelectedByVector(Vector2 delta) : void + MoveAllInSelectedOnVector(Vector2 vector) : void + FillRotator() : void + EmptyRotator() : void + RotateAllInSelectedByValue(camScreenDeltaX : float) : void + StartMovement() : void + StopMovenent(movement : bool, edgeCreation : Edge) : void + DestroyAllSelected() : void + GetSelectedNumOfVerticesAndEdges() : Vector2

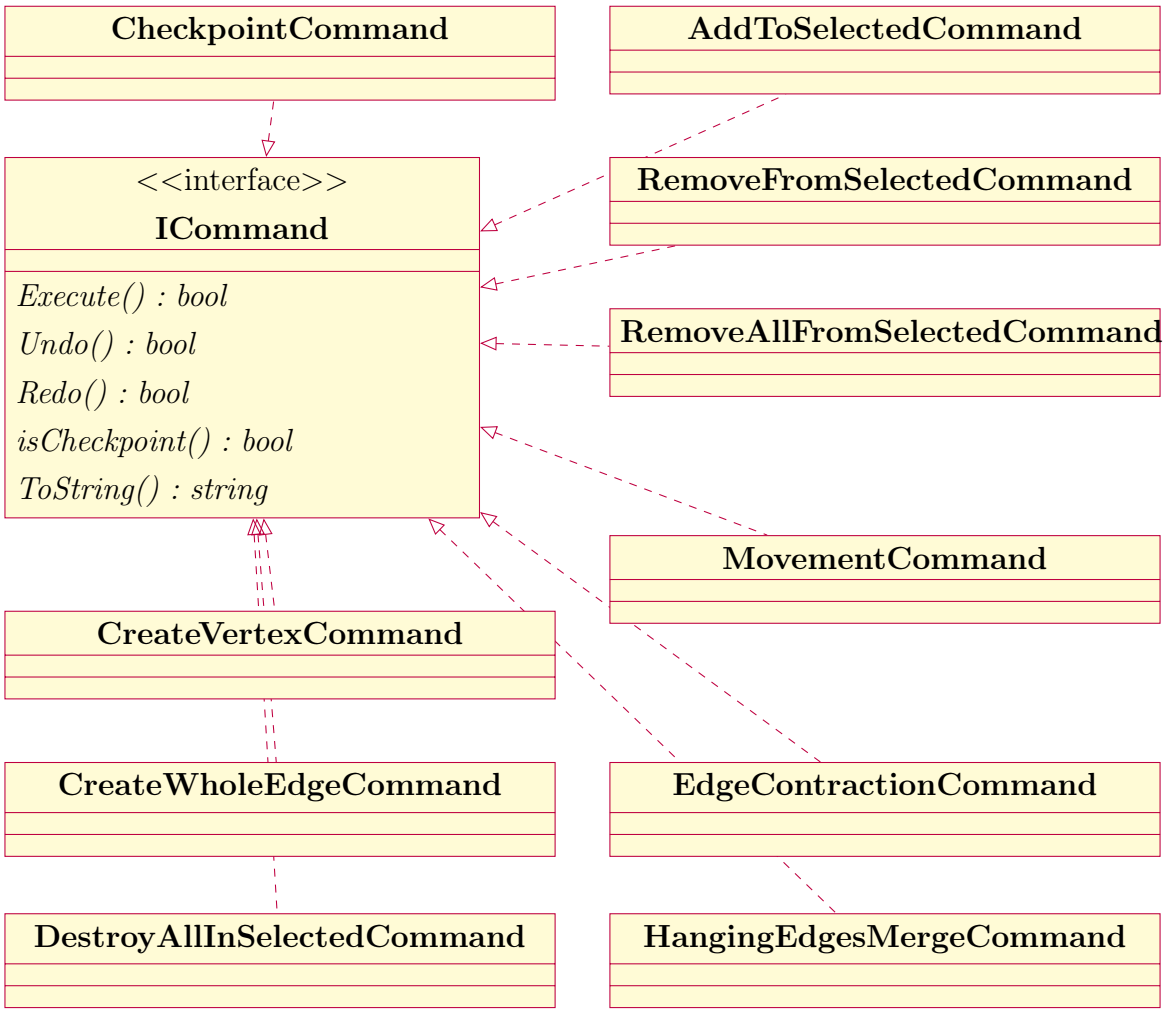
Obr. 4.5: `GraphManager` prvá časť

+ GetSelectedSubGraphInfo() : SubGraphInfo
+ GetSelectedSubGraphInfoWithIds() : SubGraphInfoWithIds
+ GetAllPartsSubGraphInfo() : SubGraphInfo
+ GetAllPartsSubGraphInfoWithIds() : SubGraphInfoWithIds
+ GetAllGraphStringInfo() : string
+ CreateVertex(mousePos : Vector3) : Vertex
+ CreateVertex(mousePos : Vector3, ID : int) : Vertex
+ CreateEdge(edgeInfo : EdgeInfo, pivot : Vector3) : Vertex
+ CreateEdge(edgeInfo : EdgeInfoWithIds, pivot : Vector3) : Vertex
+ EdgeContraction() : void
+ HangingEdgesMerge(e1 : EdgeEnd, e2 : EdgeEnd) : void
+ CreateEdgeBeginning(vertex : Vertex) : Edge
+ EcpandSelectedOnWholeComponents() : void
+ StartEdgeColoring() : SubGraph
- SetSelectedParent(Transform parent) : void
+ CleanPartsBackToOriginParent() : void
+ SetActiveParts(bool val) : void
+ CreateSubGraphFromSubGraphInfo(subGraphInfo
: SubGraphInfo, middlePoint : Vector3) : SubGraph
+ ResetSelectedColors() : void
+ ChangeColorAllVertices(color : Color32) : void
+ ChangeColorAllEdgeMids(color : Color32) : void
+ ChangeColorAllEdgeEnds(color : Color32) : void
+ ChangeColorAllLineTwoConnected(color : Color32) : void
+ ChangeColorAllLineOneConnected(color : Color32) : void

Obr. 4.6: `GraphManager` druhá časť

vstupy od používateľa, využívame špeciálnu triedu `Checkpoint`, ktorá implementuje `ICommand`. To znamená, že napríklad, keď užívateľ označí viacero prvkov naraz kedy sa použije `AddToSelected(IGraphPart part)` na každú časť samostatne, chceme, aby keď použil Undo alebo Redo tak aby sa označili prípadne odznačili všetky naraz a nie postupne. Preto `ICommand` funkcie `Undo` a `Redo` vracajú hodnotu `bool`. Všetky triedy implementujúce `ICommand` vracajú hodnotu `true`, čo znamená, že po vykonaní funkcie majú pokračovať s ďalším príkazom. Jediná trieda, ktorá vracia `false` je `Checkpoint`.

Funguje to tak, že po zavolaní funkcie `Undo` v `GraphManager`, sa vyhodí zo zásobníka s príkazmi posledný `Checkpoint` a volá sa `Undo` na všetkých príkazoch až dokým niektorý nevráti `false`, čo znamená že sme narazili na ďalší `Checkpoint` a máme skončiť s vracaním zmien. Ten `Checkpoint` následne vrátime späť do zásobníka. Zásobník či už ten s Undo príkazmi alebo Redo príkazmi, teda vždy vyzerá tak, že sú v ňom uložené inštancie `Checkpoint` triedy a medzi nimi sú bloky iných príkazov, ktoré sa vykonali na jeden povel od používateľa. Zásobníky môžu byť aj prázdne.



Obr. 4.7: Diagram tried príkazového návrhového vzoru

Každá trieda, ktorá implementuje `ICommand`, že dedí z triedy `CommandCommonInfo`.

V tejto triede máme niekoľko `protected` statických premenných, do ktorých pri spustení priradíme premenné potrebné pre vykonanie funkcií tried, napríklad `vertexPrefab` a podobne. Ostatné informácie získavajú triedy v konštruktoroch.

V triede `Checkpoint` si pamätáme či sú zmeny, ktoré sme pred ním robili jednoduché alebo nie. To, že sú jednoduché znamená, že majú implementované `Execute`, `Undo` aj `Redo`, tak, že robia to, čo sa od nich očakáva. Ak vykonáme príkaz, ktorý nie je jednoduchý, teda v tele funkcií `Undo` a `Redo` má iba `return;`, tak nastavíme pri zadávaní `Checkpoint`-u, že nieje jednoduchý. `Checkpoint` má v `Undo` funkcii, podmienku, v ktorej kontrolujeme či je jednoduchý alebo nie. Ak je jednoduchý, vykonáme iba to, že správne označíme prvky grafu, ktoré sú nastavené, že sú vybrané. Ak však `Checkpoint` nie je jednoduchý, spraví to, že pri jeho inicializácii si získa informáciu o celom grafe na ploche a znova ho vygeneruje, s tým, že zničí všetko, čo sa na ploche nachádzalo a následne označí vybrané prvky. Toto som implementovala týmto spôsobom, pretože niektoré zmeny, bolo komplikovanejšie naprogramovať aby sa vrátili, ako povedať, že sa má celý graf vygenerovať znova. Taktiež nám to umožňuje pridávať nové príkazy, kedy implementujeme iba `Ecexute` funkciu, a `Undo` a `Redo` môžeme definovať neskôr, prípadne vôbec.

Niektoré zmeny na grafe sa vykonávajú až potom, čo sa už zapíše `Checkpoint`. Napríklad, že sa zistí, že sa má zničiť hrana, lebo nie je pripojená k žiadnemu vrcholu. S tým sme sa vysporiadali tak, že sa dá vybrať posledný `Checkpoint` zo zásobníka, vykonať dodatočné zmeny k minulému bloku a následne znova vložiť daný `Checkpoint`. Pri opätovnom vkladaní sa kontroluje či je jednoduchý, ak nie tak sa vloží namiesto neho nový `Checkpoint` s novou informáciou o grafe.

4.2.2 ID systém

Funkcionalita Undo a Redo nám priniesla ďalší problém, a to so zapamätaním referencií na objekty. Počas práce s grafom sa nám stáva, že sa objekty často ničia a vytvárajú. Preto pre pamätanie si informácií v príkazoch nemôžeme použiť referencie na objekty, preto sme vymysleli systém, ktorý si bude všade pamätať iba `integer` hodnotu, teda ID a máme triedu `IDManager`, ktorá si udržuje informácie o objektoch podľa ich `ID`. Preto, keď budeme potrebovať objekt a nie len referenciu naň, vypýtame si ho od `IDManager`-a, ak nám stačí iba referencia, tak pracujeme iba s `integer` hodnotou.

Pre tento ID systém sme vytvorili ekvivalentné dátové triedy, v ktorých si pamätáme objekty aj s ich ID. Táto trieda by sa neskôr dala rozšíriť na návrhový vzor fond. Ten zabezpečuje vytváranie a ničenie objektov, pričom za celý beh programu vytvorí iba toľko objektov, koľko sa využívalo naraz v jeden moment. To znamená, že zničené objekty recykluje a dáva ich do programu znova, namiesto toho, aby vytváral nové objekty.

Pre jednoduchosť, v bakalárskej práci uvažujeme nad objektami akoby tento ID systém nebol, no v skutočnosti si výnimočne pamätáme referencie priamo na objekty.

4.2.3 Rotovanie grafu

Pri implementácii rotovania sme využili hierarchiu objektov v Unity. Totižto pri rotácii alebo pohybu rodiča to má taký vplyv na deti, že tie si udržujú relatívnu pozíciu oproti rodičovi, lokálna pozícia a tým sa mení ich umiestnenie v priestore, globálna pozícia. To znamená, že ak má dieťa lokálnu pozíciu $< 5, 0 >$, čiže je napravo od rodiča o päť jednotiek, keď zrotujeme rodiča o deväťdesiat stupňov v proti smere hodinových ručičiek, pozícia dieťaťa bude $< 0, 5 >$ a teda bude sa nachádzať nad ním, pričom opíše štvrt kružnice. Vďaka tomu, že si všetky deti zachovávajú takto relatívne pozície, keď chceme rotovať graf, tak nám stačí umiestniť objekt na pozíciu, okolo ktorej chceme rotovať a priradiť mu za deti tie objekty, ktoré chceme rotovať. Tento objekt voláme rotator a je uložený v premennej s rovnakým názvom v triede `GraphManager`.

4.2.4 Označenie celých komponentov

Označenie celého komponentu nám bolo potrebné naprogramovať pri implementácii rotovania. Následne sme to potom pridali ako samostatnú funkcionalitu, užitočnú pre používateľa napríklad, keď chce kopírovať celý komponent a tak podobne. Je to implementované pomocou návrhového vzoru vizitor. V `IGraphPart` máme funkciu na prijatie vizitora a máme implementovanú triedu `ExpandVizitor`, ktorá má funkciu pre každú triedu implementujúcu `IGraphPart`. Do `SubGraph`-u by sa nikdy nemala dostať a pri ostatných daný prvok označí, a pošle vizitor do pripojených častí. Koniec hrany označuje iba v tom prípade, že nemá vrchol.

Implementácia pomocou vizitora bola zvolená čisto z akademických dôvodov, a neuvažovala som nad inými spôsobmi.

4.3 Settings

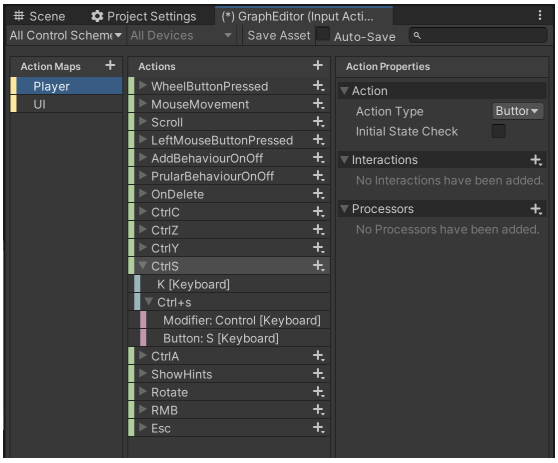
V aplikácii si želáme, aby sa niekoľko informácií ukladalo aj po vypnutí aplikácie. Potrebujeme na to spôsob a úložisko ako, a kde budeme tieto informácie ukladať. Taktiež potrebujeme triedu, ktorá bude obsluhovať a meniť módy aplikácie a prijímať vstupy na klávesnici a myši od používateľa. Toto všetko zaobstaráva trieda `Settings`.

Jej práca sa dá rozdeliť teda na dve časti, manažment aplikácie a prijímanie vstupov. Najprv sa budeme venovať prijímaniu vstupov.

4.3.1 Prijímanie vstupov od používateľa

Unity nám ponúka možnosť zadefinovania vlastných `InputActions`. Túto triedu sme nazvali `GraphEditorInputAction`. Táto trieda sa automaticky generuje podľa toho ako ju zadefinujeme v Unity editore 4.8. Následne môžeme túto triedu inicializovať v skripte a na jednotlivé jej akcie vytvoriť referencie pomocou premenných `InputAction` (bez s na konci). Táto `InputAction` má niekoľko akcií, ktorým môžeme priradiť nech sa vykoná nejaká nami zadefinovaná metóda. Tieto akcie sú rôzne, no my využívame iba tie, ktoré predstavujú stlačenie tlačidla a zdvihnutie tlačidla.

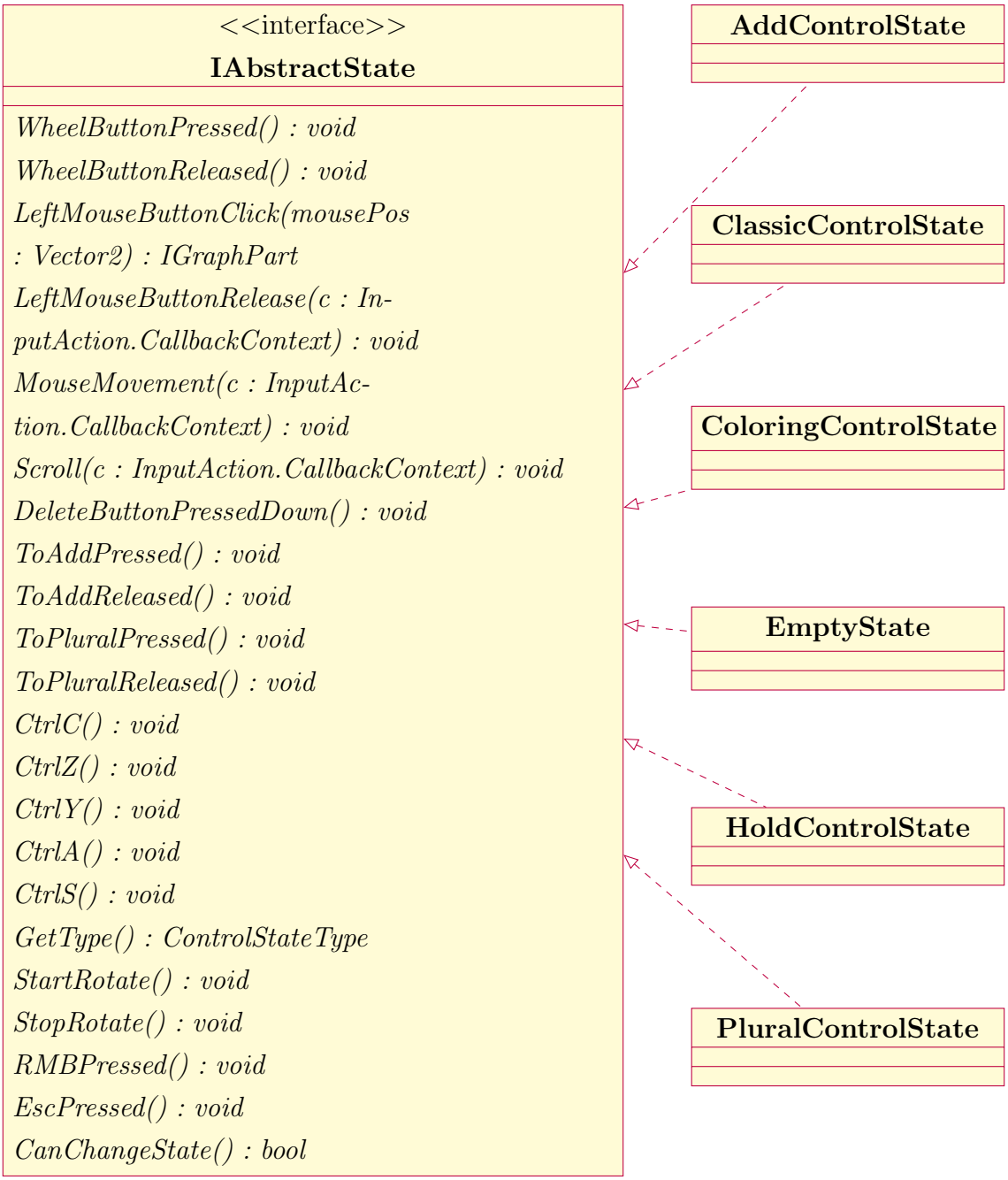
Takže v `Settings` máme premennú `GraphEditorInputAction`, ktorá implementuje triedu `InputActions` a ktorú inicializujeme vo funkcii `Awake`1.5.1. Potom pre každú `InputAction` máme premennú, napríklad `IALeftMouseButtonPressed`, ktorá nám predstavuje ľavé tlačítko na myši. Tú priradíme nech sa rovná ekvivalentej `InputAction` z inicializovanej `GraphEditorInputAction`. A keďže chceme, aby sa nám niečo dialo aj pri stlačení ľavého tlačítka na myši, aj pri jeho pustení priradíme premennej `IALeftMouseButtonPressed`, aj pre `performed`, aj pre `canceled` náležité funkcie. Takto to potrebujeme pre každú `InputAction`. Výhodou Unity Input systému je to, že môžeme mať akciu nazvanú `IALeftMouseButtonPressed`, ale reálne môžeme nastaviť, aby bola zviazaná s rôznymi tlačidlami, ako napríklad akciu `CtrlS` máme zviazanú aj s Ctrl modifikátorom a klávesnicou S, ale aj s obyčajnou klávesnicou K.



Obr. 4.8: Editovanie `InputActions` v Unity

Avšak nie vždy chceme, aby sa nám dialo to isté pri rovnakej akcii. Máme módy, podľa ktorých aplikácia mení svoje správanie. To sme docielili pomocou pomocou návrhového vzoru stratégia. Kde máme interface `IAbstractState` a premennú `controlState` typu `IAbstractState` v triede `Settings`. `IAbstractState` má funkciu na každé potrebné stlačenie a zdvihnutie tlačidla. V triede `Settings` akurát voláme potrebné funkcie v premennej `controlState`. Do `controlState` priraďujeme zrovna takú inštanciu triedy, podľa toho aký máme mód. Triedy implementujúce `IAbstractState` máme popísane v obrázku 4.9.

Všetky tieto triedy derivujú z triedy `AbstractState`, ktorá derivuje z `AccessBehaviour`. V triede `AbstractState` máme niekoľko všeobecných premenných pre všetky kontrolné triedy, vrátane statickej premennej `state`. Táto premenná je typu `ControlStateState`, čo je enum, ktorý má hodnoty podľa obrázka 4.10.



Obr. 4.9: Diagram kontrolných tried

Hodnotu premennej `state` testujeme na začiatku každej funkcie, či má požadovanú hodnotu. Požadovaná hodnota je väčšinou `ControlStateState.WAITING`, pretože chceme, aby sa nič inšie už predtým nedialo. To znamená, že chceme striktne oddeliť, aby sa nám nekrižovali vstupy. Ako napríklad, aby sme nemohli zmeniť mód aplikácie, keď práve rotujeme graf. Pri každom stlačení či zdvihnutí tlačidla kontrolujeme, či má požadovaný stav a taktiež ho meníme aby nemohlo dojsť k prekrytiu funkcionalít. Taktiež sa nám vďaka tomu môžu diať rôzne veci pri rovnakej akcii v rovnakom móde. Ako napríklad v "ADD" móde keď zdvihneme ľavé tlačidlo myši tak sa nám buď vytvorí

```
public enum ControlStateState
{
    WAITING = 0,
    TO_MOVE = 1,
    MOVING = 2,
    TO_ROTATE = 3,
    ROTATING = 4,
    CAMERING = 5,
    TO_ADD = 6,
    TO_PLURAL = 13,
    RMB_PRESSED = 7,
    TO_COLLECTOR = 8,
    COLLECTOR = 9,
    TO_CREATE_VERTEX = 10,
    TO_CREATE_EDGE = 11,
    CREATING_EDGE = 12,
    TO_DROP = 14,
}
```

Obr. 4.10: `enum ControlStateState`

vrchol, ak sme od stlačenia ľavého tlačidla myši nepohli myšou, alebo ak sme vytvárali hranu tak sa nám položí do plochy koniec hrany, ktorý ťaháme myšou. Na obe akcie máme hodnotu v `ControlStateState`.

Celú logiku kontrolných tried, pomocou stavu, som refaktorovala vrámci bakalárskej práce.

V triede `Settings` máme funkciu, ktorá nastavuje `controlState`. Pre potrebu `HoldControlState`, máme aj špeciálnu funkciu, ktorá berie `SubGraphInfo`, podľa ktorého vytvorí `HoldControlState` graf, ktorý udržuje dokým ho používateľ neumiestni na plochu.

4.3.2 Množinový výber

Ako sme spomínali v návrhu, chceme, aby sme vedeli vyberať objekty obdĺžnikovým výberom, podobným, ako napríklad na ploche Windowsu. Na to nám slúži špeciálny objekt `Collector` so skriptom s rovnakým menom. Tento objekt sa nachádza, ako premenná v `Settings` a posiela sa ako jedna zo všeobecných premenných pre `AbstractState`. Následne `PluralControlState` s ním pracujeme. Pri kliknutí myšou na prázdne miesto do plochy mu nastavíme pozíciu na túto súradnicu a šírku a výšku

objektu na nulu. Následne nastavujeme iba šírku a výšku objektu podľa vzdialenosti myši od miesta, kde sme stlačili tlačidlo. Tým sa nám vytvára obdĺžnik, ktorý má rovnobežné strany s hranami obrazovky a jedna uhlopriečka vždy spája pozíciu myši s miestom, kde sme stlačili tlačidlo.

Pri výbere prvkov využívame opäť `Collider`, ktorý je na objekte `Collector`. Keď doň vojde nejaký objekt ktorý má tag nastavený na `Collectorable` pridá ho k vybraným objektom. Týmto spôsobom odfiltrujeme konce hrán, ktoré sú zavesené na hrane.

4.3.3 Manažment aplikácie

Trieda `Settings` má na starosti aj správu informácií, ktoré sa ukladajú do pamäti. Máme triedu `PlayerSettings`, ktorú sme definovali ako serializovateľnú. Vďaka tomu vieme jednoducho konvertovať túto triedu do Json súboru. Takýto Json súbor následne uložíme do pamäte počítača. V tejto triede si ukladáme premenné z nastavení, grafy do databázy a graf, ktorý sa ukladá na ploche. Pri spustení aplikácie sa podľa týchto uložených dát nastaví uložený graf a databáza. Taktiež sa nastavujú niektoré premenné v `AbstractState`. Ďalšie funkcie triedy `Settings` obsluhujú ukladanie grafu do databázy a zakázanie a povolenie prijímania vstupov. To sa uskutočňuje tak, že sa do `controlState` nastaví inštancia triedy `EmptyState`. Tá má vo všetkých funkciách akurát `return`. Táto trieda slúži na to aby sa nič nedialo napríklad, keď používateľ zadáva meno nového grafu na uloženie.

4.4 Kopírovanie a Generovanie

Dátové triedy sa jednoducho využívajú pri kopírovaní, v `GraphManager`-ovi vieme získať `SubGrafInfo` o označených prvkoch. To následne pošleme do funkcie v `Settings`, ktorá priradí `controlState HoldControlState` pričom mu pošle vytvorené `SubGrafInfo`. `HoldControlState` následne pomocou `GraphManagera` vytvorí daný duplikát označeného grafu.

Ten istý princíp sa využíva tiež pri generovaní až na to, že sa potrebné `SubGrafInfo` získava zo statickej funkcie v `SubGrafInfo` triede. Pre každý typ grafu máme vlastnú funkciu, ktorá prijíma parameter k. Vo funkcii sa vypočítajú pozície vrcholov a uložia sa. Následne tie vrcholy, ktoré majú byť spojené sa pre ne vytvorí hrana, kde konce sú pozície hrán a stred dávame buď do stredu alebo tak, aby to bolo vizuálne pekné. Ako napríklad pre neprekrížený rebrík aby šla hrana, spájajúca vrchol na začiatku a konci, ponad alebo popod všetky hrany a nie skrz ne.

4.5 Interakcia s používateľom pomocou UI

Používateľské rozhranie alebo UI je ďalším spôsobom, ako môže používateľ komunikovať s aplikáciou. Teraz si popíšeme akým spôsobom je to implementované.

UI sa v Unity do určitej miery dá spraviť iba v editore a netreba k nemu žiaden kód. Avšak v našej aplikácii je značná časť panelov generovaná pomocou kódu. Pri generovaní UI sa využívajú prefaby1.5.3. Taktiež sa využívajú Unity komponenty na UI objektoch. Všetky UI objekty sú zavesené na špeciálnom objekte s komponentom `Canvas`. `Canvas` nám slúži na to, aby sa nám zobrazilo UI na obrazovke. My používame také, ktoré sa rozpína podľa okna, to však ešte neznamená, že sa budú elementy držať v obrazovke.

Objekty, ktoré sú v hierarchii pod objektom s `Canvas`-om majú namiesto `Transform` komponentu jeho rozšírenú verziu `RectTransform`. V nej sa dá definovať či si má vertikálne alebo horizontálne udržovať veľkosť podľa čísla, alebo podľa vzdialenosti oproti hraniciam objektu rodiča. Takto sa UI správa responzívne podľa toho ako sa zväčšuje a zmenšuje okno aplikácie. Dá sa to popísať tak, že sú to obdĺžniky, ktoré sa pohybujú v závislosti od svojho rodiča.

Pre pokročilejšie UI správanie nám slúžia ďalšie Unity UI komponenty, ako napríklad `ContentSizeFitter`, ktorý mení svoju veľkosť v závislosti od svojho obsahu alebo `LayoutGroup`, ktorý delí spravodlivo svoj rozsah medzi deti podľa rôznych kritérií. Spraviť pekné a responzívne UI v Unity je časovo náročná a komplikovaná práca. Pri implementácii farbenia mi urobenie UI a jeho kódu, trvalo dlhšie ako zvyšok celej funkcionality.

4.5.1 Hierarchia

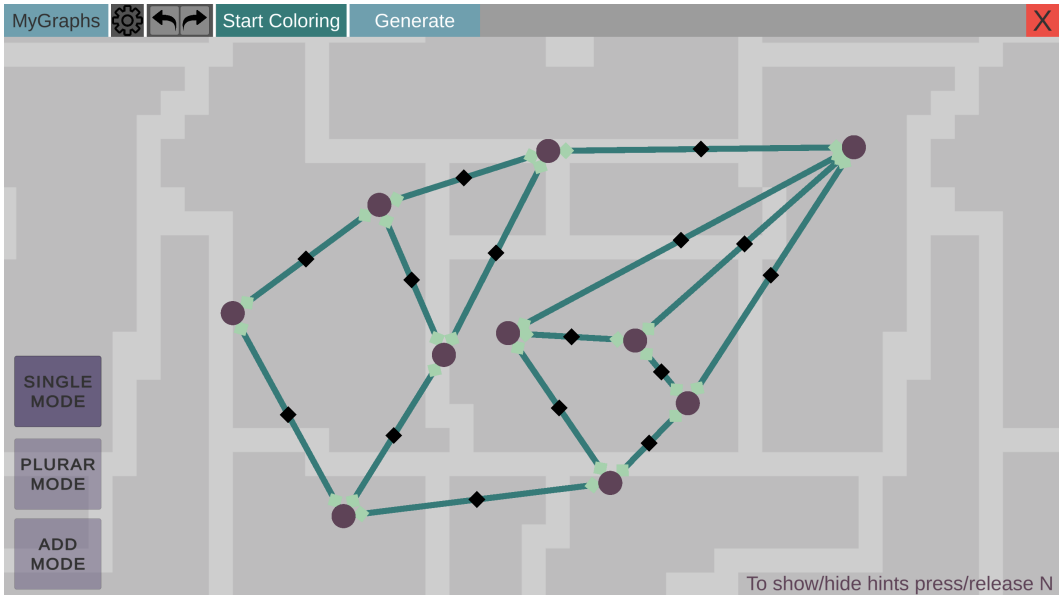
V objekte, kde máme komponent `Canvas`, máme aj skript `UIManager`. Ten má v sebe premennú pre každé tlačidlo a každý panel, ktorý je dostupný z hlavnej obrazovky. Následne pre každý panel máme vytvorenú triedu a premennú v `UIManager`-ovi. Keďže väčšina ostatných skriptov má prístup iba k `UIManager`-ovi a nie k jednotlivým panelom, `UIManager` slúži predovšetkým, ako distribútor pre paneli a reálne spravuje iba nevyhnutné množstvo elementov, ako sú tlačidlá pre zobrazenie panelov, tlačidlá modov, výstražný panel a tlačidlo pre ukončenie aplikácie. Každému tlačidlu priraďujeme počúvajúce funkcie v kóde, i keď sa to dá nastaviť v editore. Zvyšuje to prehľadnosť a ľahšie sa hľadajú chyby v implementácii.

Hierarchia tried UI elementov, zobrazujúca závislosti, ktorá trieda má, ktoré na starosti. Triedy pod `UIManager`-om:

- `MyGraphsPanel` => panel databázy
  - `MyGraphOption` => možnosť pre jeden uložený graf

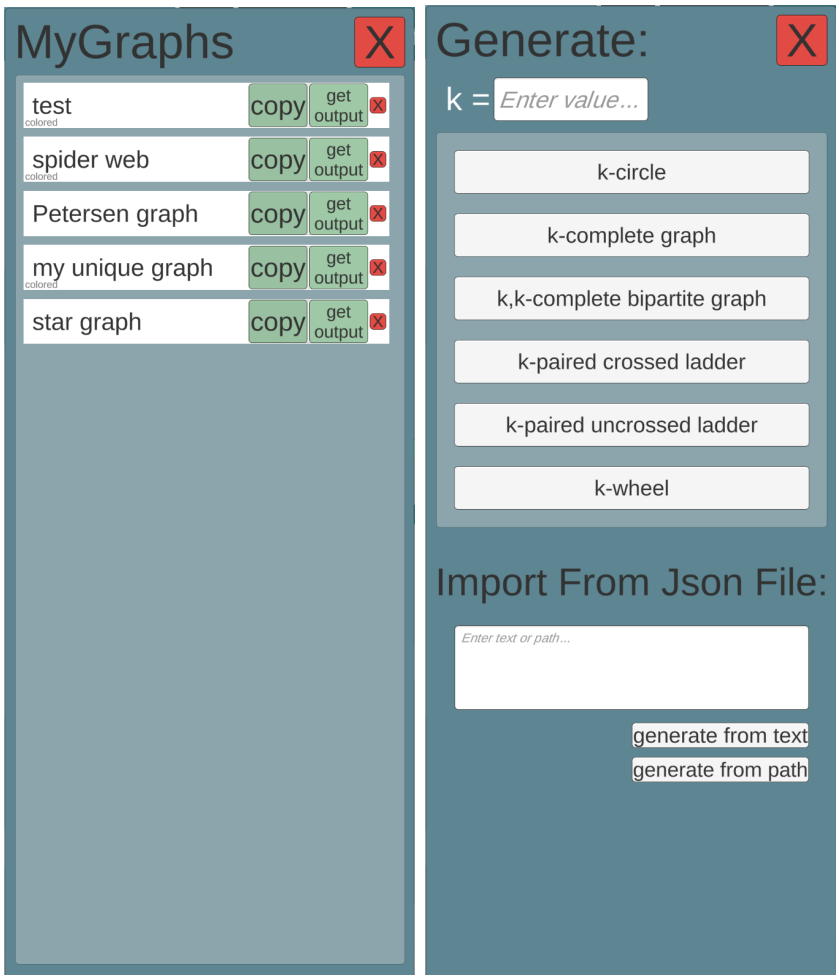
- `ChooseNamePanel` => panel pre zadanie mena pre uloženie grafu
- `OutputPanel` => výstupový panel
- `SettingsPanel` => panel nastavení
- `ColoringPanel` => panel farbenia
  - `EdgeIdentificator` => element s informáciami o špeciálnej hrane
  - `LocalEdgeIdetificator` => zobrazenie mena v grafe
  - `UpperIndex` => hlavička stĺpca s pomenovaním špeciálnej hrany v mriežke
  - `GridRow` => riadok s indexom a s tlačidlami pre rovnosť
    - `EqualityButton` => tlačidlo pre rovnosť dvoch hrán
  - `BlockDefinition` => element so zadefinovaním bloku
- `UIWarning` => panel upozornenia
- `GeneratePanel` => panel generovania

4.5.2 Pracovná plocha



Obr. 4.11: Ukážka pracovnej plochy

Pre užívateľa je potrebné, aby mal stále možnosť sa medzi módmi prepínať ako aj pomocou klávesnice tak, aj pomocou tlačidiel v UI. Preto sme sa rozhodli spraviť rozostavenie UI aplikácie tak, že tlačidlá modov budú zobrazené stále a potom bude v hornej časti lišta, ktorá bude sprístupňovať ostatné paneli, a to panel databázy, farbenia, nastavení a generovania. To sa dá vidieť na obrázku 4.11. Paneli sa zobrazujú v ľavej časti obrazovky.



Obr. 4.12: Panel databázy a generovania

4.5.3 Panel databázy a generovania

V panel databázy zobrazujeme uložené grafy a potrebné minimum o uloženom grafe, a to je názov, pod akým sme si ho uložili a či bol uložený aj s farbením alebo bez neho. Ďalej máme možnosť graf skopírovať, získať z neho výstup, čo nám zobrazí inší panel alebo uložený záznam odstrániť.

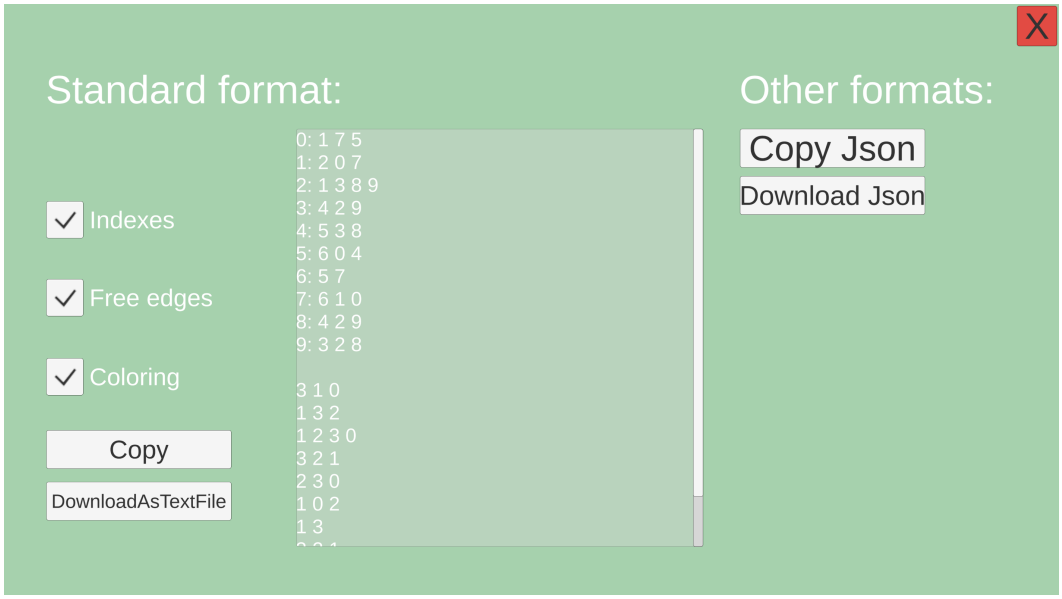
Panel generovania svoj obsah nemodifikuje, vždy zobrazuje výber typov, ktoré môžeme vygenerovať a jeden vstupný parameter k. Ďalej ponúka možnosť vygenerovania grafu z json[3.2] súboru, a to buď definovaním jeho cesty alebo vložením jeho obsahu.

Oba tieto paneli sú zobrazené na obrázku 4.12.

4.5.4 Výstupový panel

Výstupový panel sa dá vidieť na obrázku 4.13. Vo výstupovom paneli máme možnosť získania dvoch formátov, a to json[3.2] alebo štadarný formát[3.1]. Tento formát dokážeme modifikovať podľa toho či si želáme, aby tam boli indexi, vysiace hrany, a ak sme graf uložili aj s farbením, či chceme aby sa aj to pridalo na koniec. Oba formáty

sa nám dajú uložiť do globálneho buffera, ako keď napríklad kopírujeme text z nejakej stránky na internete alebo sa dá uložiť do textového súboru v počítači do Downloads priečinku. Názov súboru sa automaticky odvodzuje od názvu grafu.



Obr. 4.13: Výstupový panel

4.6 Farbenie

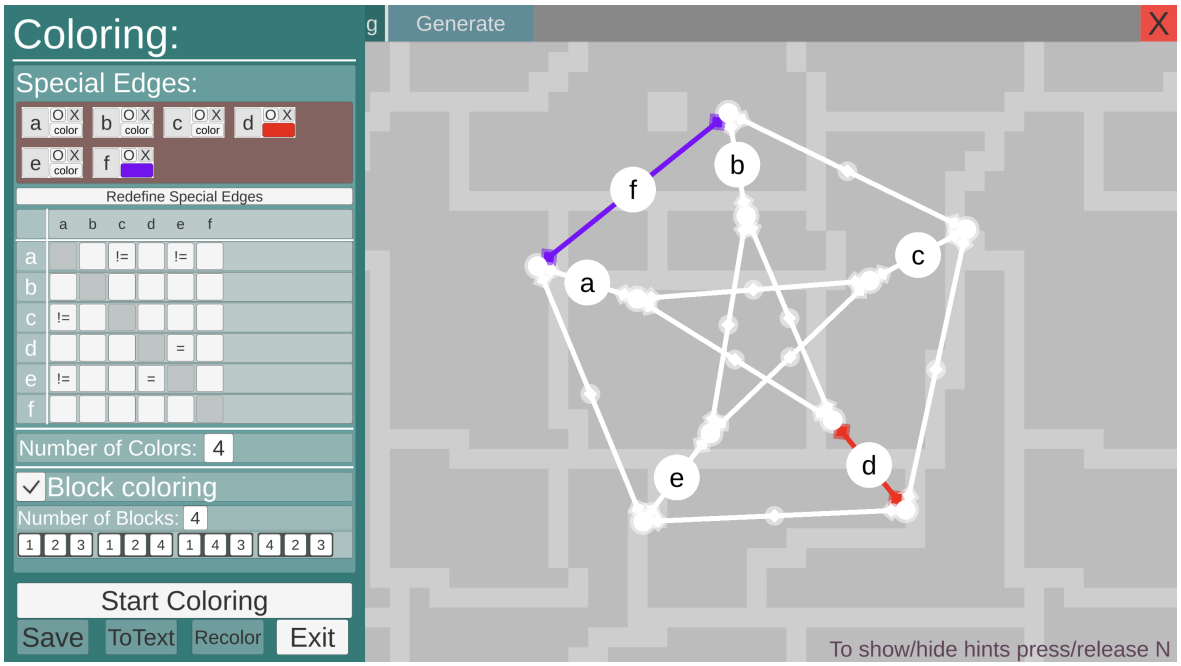
Farbenie si rozdelíme do dvoch častí, a to implementácia UI a implementácia samotnej logiky zafarbenia grafu.

4.6.1 UI panel

Panel farbenia je generovaný z drvivej väčšiny v kóde, jeho obsah je premenlivý a jeho implementácia mi trvala rovnako dlho ako implementácia logiky farbenia. Ukážka celej obrazovky pri farbení sa dá vidieť na obrázku 4.14. Panel farbenia sa modifikuje podľa požiadaviek užívateľa. Počet farieb je jednoduché vstupné pole. Špeciálne hrany sa vyberajú mimo panelu, priamo kliknutím na danú hranu. Po vybraní špeciálnych hrán v ploche, sa dá vytvoriť matica, do ktorej sa dá zadefinovať či musia mať hrany rovnakú alebo inšiu farbu. Taktiež sa dá špeciálnym hranám zadefinovať konkrétna farba alebo zmeniť jej názov. Názvy hrán však nemajú žiaden dopad na farbenie, ide len o uľahčenie práce pri väčších grafoch. Prípadne ak by sa aplikácia rozširovala do budúcna o posielanie aj samotných nastavení farbenia, tak tam pomenovanie hrán by malo väčší zmysel.

Blokové farbenie sa dá zapnúť iba pri kubických grafoch, mimo nich je daný prepínač neaktívny a nedá sa zapnúť. To, že či je graf kubický sa pýta na začiatku farbenia

`GraphManager`-a. Blokové farbenie sa definuje ako množina množín, ak teda sú dva bloky zadefinované s rovnakými indexmi farieb, nedá sa graf dať zafarbiť a daný blok je vyznačený červenou.



Obr. 4.14: Ukážka aplikácie počas farbenia

Celý tento panel je prispôsobený na to, aby aj tí užívatelia bez programátorského či matematického vzdelania boli schopný ho pochopiť a zafarbiť daný graf. Je to kvôli tomu, aby nebolo nutné používať pri definovaní farbenia manuál alebo tutoriál, ale aby na to bol schopný užívateľ prísť intuitívne.

Náročné bolo zadefinovanie všetkých vzťahov, ako sa jednotlivé elementy UI ovplyvňujú. Keďže sa obsahy a veľkosti elementov menia v reálnom čase, treba manuálne obnovovať prepočítavanie zobrazenia UI.

4.6.2 Logická časť implementácie farbenia

Táto časť je implementovaná v triede `ColoringManager`. Keďže nespadá priamo pod jednu zo základných tried, dali sme ju na rovnakú úroveň, ako sú oni, i keď spravuje oproti nim veľmi málo. Táto úroveň je, že je premenná tejto triedy dostupná v `AccesBehaviour`.

Kód buď obsluhuje UI farbenia alebo sa priamo pokúša zafarbiť graf. Obsluhou UI je najmä inicializácia panelu a vrátenie sa do normálneho behu aplikácie. Na to, aby zafarbil graf používame knižnicu OR-Tools 1.5.6. OR-Tools ponúka rôzne triedy, my používame triedu `CpSolver`. CP je skrátene od Constraint Programming. Táto trieda je v podstate SAT-solver s nejakou nadstavbou, aby sa s ním pracovalo jednoduchšie. Na

to, aby sme zistili, aké farbenie môže mať graf potrebujeme vytvoriť model, ktorému priradíme premenné a podmienky na tých premenných. Model vytvoríme pomocou triedy `CpModel` a tento model dáme následne vyriešiť `CpSolver`-u.

Teraz si vymenujeme funkcie `CpModel`-a, ktoré budeme využívať. `NewIntVar(long lb, long ub, string name)` slúži na to, že v modeli vytvoríme premennú typu integer, ktorá môže nadobúdať hodnoty od `lb` po `ub` vrátane. `NewBoolVar(string name)` nám v modeli vytvorí premennú typu bool. Tieto funkcie vracajú premennú typu `IntVar` a `BoolVar`. Tie sa potom využívajú v ďalších funkciách `CpModel`-a.

Funkcia `Add` je základná funkcia na pridávanie podmienok. Prijíma premennú typu `BoundedLinearExpression`. Tá sa dá popísať rovnicou alebo nerovnicou so znamienkami plus, mínus a krát. V rovnici sa môžu nachádzať premenné typu `IntVar` alebo konštanty. Príklad správnej `BoundedLinearExpression` je napríklad `2 + x = y - 3 * z`, ak `x`, `y` a `z` sú C# premenné typu `IntVar`. `Add` vracia premennú typu `Constraint` (obmedzenie). Na tejto premmenej potom môžeme zavolať metódu `OnlyEnforceIf`, ktorá ako argument berie premennú typu `BoolVar`. Táto funkcia slúži na to, že dané obmedzenie má platiť iba v tom prípade, že hodnota `BoolVar` bude true. Poslednou funkciou `CpModel`-a, ktorú využívame, je `AddExactlyOne`. Tá berie ako argument iterovateľný objekt, obsahujúci `BoolVar`. Tá zapríčiní, že z daných `BoolVar` premenných bude vyžadovať, aby mala `true` hodnotu práve jedna. Z týchto funkcií sa nám podarilo vytvoriť celú podmienku pre farbenie.

Funkcia `ColoringManager`-a, ktorá sa pokúša zafarbiť graf sa nazýva `TryColor`. Ako argumenty berie počet farieb či sú definované špeciálne hrany, informácie k špeciálnym hranám, informácie o rovnostiach a nerovnostiach špeciálnych hrán, či je definované blokové farbenie a definíciu blokov. Informácie potrebné pre jednu špeciálnu hranu sú, ktorá hrana to je, či jej špecifikoval používateľ index konkrétnej farby, ktorú má hrana mať, index tejto farby a ako bola pomenovaná v editore. Informácie o rovnostiach a nerovnostiach sú trojice: hrana, druhá hrana a aký vzťah majú mať medzi sebou. Tento vzťah môže byť rovnosť, nerovnosť alebo nezáleží na tom. Bloky sú definované pomocou množiny trojprvkových množín. Ako prvé sa vytvorí premenná `model` a priradí sa jej inštancia `CpModel`-u. Ďalej sa vytvorí `Dictionary` (slovník). Ekvivalentom `Dictionary` v C++ je `unordered_map`. Slovníku nastavíme ako typ kľúča `Edge` a ako typ hodnoty `IntVar`. Tento slovník využijeme na to aby sme si v ňom pamätali model premenné na základe hrán. Najprv pridáme špeciálne hrany a pridáme pridáme do modelu obmedzenia týkajúce sa iba špeciálnych hrán. Každej špeciálnej hrane sa vytvorí model premenná a uloží sa do slovníka. Ak má hrana priradenú konkrétnu farbu tak sa pri vytváraní model premennej nastaví, že má mať práve tú hodnotu, ak ju priradenú nemá, tak nastavíme, že môže mať hodnotu od nuly po počet farieb mínus jedna. Ten istý rozsah sa nastaví aj pri vytváraní model premenných pre ostatné hrany.

Ďalej pomocou funkcie `Add` nastavíme, aby sa rovnali alebo nerovnali hodnoty tých

model premenných, ktorým sme nastavili rovnosť, alebo nerovnosť v mriežke počas definovania premenných. V kóde rovnosť vyzerá takto:

```
model.Add(specialEdgesVars[var.e1] = specialEdgesVars[var.e2]);.
```

Keď pridáme všetky hrany do slovníka, pridáme obmedzenie pre každú dvojicu hrán, ktorá zdieľa vrchol, že sa ich model premenné, teda ich farba, nemôžu rovnať.

Ak máme blokové farbenie musíme zaistiť to, že každá trojica hrán, okolo vrchola, má nejaké blokové farbenie. To sme docielili tak, že pre každý blok a každú jeho permutáciu sme vytvorili `BoolVar` premennú. Na poradí nám záleží, pretože máme tri hrany a tri farby v jednom bloku, a to aké farby priradíme hranám, sa dá spraviť šiestimi spôsobmi. Čiže pre jednu trojicu vrcholov vytvoríme šesť krát počet blokov `BoolVar` premenných. A jednu premennú použijeme tri krát, keď každej hrane z trojice priradíme práve jednu farbu z bloku.

Následne nám už len stačí povedať, aby sa `CpSolver` snažil nájsť také priradenie hodnôt premenným, aby to spĺňalo model, a ak sa mu také podarí nájsť, tak tým našiel aj jedno farbenie.

Záver

Cieľom bakalárskej práce bolo pridanie ďalších funkcionalít do aplikácie na tvorbu a editovanie grafov, ktorá bude slúžiť ako vedecký nástroj pre niektorých vyučujúcich na fakulte matematiky, fyziky a informatiky, Univerzity Kamenského.

Práca na tejto práci pozostávala z tvorby nových funkcionalít, refaktorizácie starých implementácií a testovania. Podarilo sa vytvoriť komplexnú aplikáciu, ponúkajúcu široké možnosti, pre prácu s grafmi.

Najväčší dôraz bol na to aby sa dalo pracovať s viacerými prvkami naraz. Aby bolo jednoduché kopírovať a pracovať s viacerými grafmi. Taktiež tvorba grafu, aby bola pohodlná a rýchla. Preto sme implementovali niekoľko spôsobov získavania celých vytvorených grafov, či už nami, ktoré sme si uložili v databáze, alebo získali json s dátami o grafe, alebo vygenerovaných podľa štandardných typov grafov. Taktiež sme prácu s grafom rozdelili na prácu s jedným prvkom, viacerými prvkami alebo pridávanie prvkov.

Tento editor bude slúžiť najmä ako pomôcka pri skúmaní grafov a jeho praktickosť a využiteľnosť sa ukáže až pri samotnom využívaní v budúcnosti. Samozrejme, že tam je ešte priestor pre ďalšie funkcionality, ktoré by mohli potenciálne rozšíriť využitie tohto editora, aj na študijné účeli alebo pre laikov, ktorý majú záujem o teóriu grafov.

Literatúra

[1] Graph Online. Graph Online, Find shortest path. Retrieved December 12, 2022, from https://graphonline.ru/en/

[2] Huawei Cloud. Graph Engine Service. Retrieved December 12, 2022, from https://support.huaweicloud.com/intl/en-us/ges/index.html

[3] NetworkX. NetworkX documentation. Retrieved December 12, 2022, from https://networkx.org/

[4] Matplotlib. Matplotlib - Visualization with Python. Retrieved December 12, 2022, from https://matplotlib.org/

[5] yEd Live. yEd Live - organic2. Retrieved December 12, 2022, from https://www.yworks.com/yed-live/

[6] ZoomCharts. Graph Editor from ZoomCharts. Retrieved December 12, 2022, from https://apps.zoomcharts.com/graph-editor/

[7] Google OR-Tools. About OR-Tools. Retrieved February 21, 2023, from https://developers.google.com/optimization/introduction

[8] Unity. Unity Introduction. Retrieved February 23, 2023, from https://unity.com/

[9] Wikipédia. Bézierova křivka. Retrieved February 23, 2023, from https://cs.wikipedia.org/wiki/B

[10] Json. Introducing JSON. Retrieved May 19, 2023, from https://www.json.org/json-en.html

Príloha A: obsah elektronickej prílohy

V elektronickej prílohe priloženej k práci sa nachádzajú súbory so zdrojovým kódom a súbory pre spustenie aplikácie. Zdrojový kód nie je zverejnený na žiadnej stránke, ponúkajúcu systém riadenia verzií.

Zdrojový kód sám o sebe nie je sebestačný na vytvorenie aplikácie, pretože neobsahuje žiadne Unity skripty a informácie pre Unity editor. Zdrojový kód sú len skripty, ktoré som ja napísala a majú slúžiť na to, keby si chcel niekto pozrieť priamo implementáciu niektorých tried.

Druhá časť sú súbory pre spustenie aplikácie. Súbor so samotnou aplikáciou musí ostať medzi ostatnými súbormi. Čiže treba vytvoriť odkaz na aplikáciu, ktorý si dáte do požadovaného priečinka a odtiaľ ju budete spúšťať. Alebo celý priečinok so všetkými súbormi umiestite kde to chcete mať a aplikáciu budete spúšťať priamo odtiaľ.

Aplikácia je určená iba na Windows operačný systém. Funguje plnohodnotne vo všetkých Windowsoch lepších ako Windows 7 a vo Windows 7 nefunguje len zafarbenie grafu.

Príloha B: Používateľská príručka

V tejto prílohe uvádzame používateľskú príručku k nášmu softvéru. Popíšeme si, čo presne a ako sa dá spraviť s naším editorom. Túto kapitolu je vhodné čítať pri prvom spustení aplikácie, kedy sa budete snažiť naučiť s aplikáciou pracovať.

V "ADD" móde po kliknutí na prázdnu plochu sa vytvorí vrchol. Kliknutím na vrchol a potiahnutím vytvoríme hranu, ktorá bude visiaca alebo sa pripne k vrcholu podľa toho či zdvihneme ľavé tlačidlo myši nad vrcholom alebo nad prázdnou plochou. Pri stlačení Delete alebo Backspace tlačidla sa nám vymaže označený prvok. Označený prvok je vždy posledný vytvorený a nedokážeme vybrať iný. Kamerou hýbeme potiahnutím ľavého tlačidla alebo kolečka myšou z prázdneho miesta.

V "SINGLE" móde ak stlačíme na ľubovoľný prvok vždy sa nám všetky ostatné odznačia a kliknutý sa nám označí, alebo ak bol označený tak sa nám odznačí. Ak potiahneme či už označený alebo neoznačený prvok, vždy sa označí a presunieme ho. Po kliknutí na hranu sa nám zobrazí EdgeContraction tlačidlo, ktoré spôsobí kontrakciu hrany. Kamerou hýbeme potiahnutím ľavého tlačidla alebo kolečka myšou z prázdneho miesta. Jediný mód, v ktorom vieme odpájať hrany vrcholom.

V "PLURAL" móde ak stlačíme na prvok tak sa nám zmení jeho označenie. To znamená, že keď je označený tak sa odznačí alebo ak je neoznačený tak sa označí. Keď potiahneme označeným prvkom, hýbu sa nám všetky označené. Keď potiahneme ľavým tlačidlom myši z prázdneho miesta, vytvorí sa hromadný výber podobný ako na ploche Windowsu. Kamerou vieme hýbať potiahnutím koliečkom myši z prázdneho miesta.

Rotujeme potiahnutím pravého tlačidla myši alebo pri stlačení klávesy R. Hýbať myšou treba vodorovne, pretože sa zisťuje zmena v osi x a podľa toho sa rotuje graf.

Keď dáme dva visiace konce hrany na seba, tak sa nám spoja do jednej hrany. Ak je hrana visiaca má inšiu farbu ako keď nie je.

V "SINGLE" a "PLURAL" móde máme ešte možnosť označiť celé komponenty. Ak nemáme vybraný žiaden prvok označia sa nám všetky a keď máme vybrané prvky, tak sa označia iba ich komponenty.

Kopírovať označený graf sa dá v "SINGLE" a "PLURAL" móde, po stlačení Ctrl+C. Po skopírovaní sa nám graf drží okolo myši, dokým neklikneme na prázdne miesto na ploche, kedy sa umiesti tam, kde práve je. Počas kopírovania ho dokážeme rotovať, a

to pomocou pravého tlačidla myši alebo stlačením klávesy R. Počas kopírovania vieme hýbať kamerou pri stlačení ľavého tlačidla myši a potiahnutím. Pri kopírovaní nám nereaguje UI dokým graf neumiestnime alebo dokým nestlačíme Esc, kedy sa nám kopírovanie zruší.

Ukladať graf dokážeme stlačením Ctrl+S v móde "SINGLE" alebo "PLURAL". Následne sa nám zobrazí panel s vstupom pre meno grafu. Z tohto panela vieme odísť stlačením Esc.

Z panela databázy dokážeme kopírovať uložené grafy alebo sa dostať do výstupového panela. Vo výstupovom paneli dokážeme modifikovať štandardný výstup, ktorého ukážka sa nám zobrazuje v strede. Následne vieme vybrať pomocou UI, aby sa nám to, čo máme v strede nakopírovalo do globálneho buffera alebo, aby sa nám do priečinka Downloads na našom počítači uložil súbor s tým obsahom. Taktiež vieme skopírovať Json výstup alebo si ho uložiť tiež do Downloads priečinka.

Ďalej máme v hornej lište tlačidlo nastavení, Undo a Redo tlačidlo. Undo a Redo sa taktiež spúšťajú pri stlačení Ctrl+Z a Ctrl+Y.

Keď máme označení aspoň jeden prvok a stlačíme Start Coloring tlačidlo, tak sa nám spustí nastavovanie farbenia pre celé komponenty prvkov, ktoré máme označené. Keď sme v nastaveniach farbenia, tak vieme hýbať kamerou a klikať na stredy hrán. Po kliknutí na stred hrany sa nám pridá do špeciálnych hrán a zobrazí sa nám nad ňou označenie s jej názvom. V paneli jej vieme meniť označenie a odstrániť ju zo zoznamu špeciálnych hrán. Keď potvrdíme špeciálne hrany, vytvorí sa nám diagonálne symetrická matica, v ktorej máme tlačidlá. Po stlačení na tlačidlo vieme cyklicky prepínať medzi rovnosťou, nerovnosťou alebo, že nám nezáleží aký vzťah majú dané dve hrany. Po potvrdení hrán už nevieme meniť ich označenie alebo ich odstrániť zo špeciálnych hrán. Vieme im však určiť konkrétnu farbu. Pre každú špeciálnu hranu máme ešte tlačidlo, ktoré dá kameru priamo nad ňu a zvýrazní nám jej označenie v ploche.

V paneli farbenia máme ešte jednoduché vstupné pole na počet farieb. Povoliť blokové farbenie vieme iba v tedy, keď je graf kubický. Keď povolíme blokové farbenie tak sa nám zobrazí vstupné pole na počet blokov a definície blokov. Definície blokov vyzerajú tak, že sú to trojice vstupných polí, do ktorých máme napísať indexy farieb. Pozadie bude červené ak sa nám nejaká trojica zadaných indexov zhoduje, ako množina s inou trojicou zadaných indexov alebo ak definujeme index, ktorý je väčší ako počet farieb.

Ďalej máme tlačidlo na pokus o zafarbenie grafu. Tento pokus sa spustí iba ak nemáme červenú definíciu blokového farbenia a buď nemáme špeciálne hrany alebo ich máme potvrdené. Keď sa nám graf zafarbí máme tlačidlá na uloženie alebo na znova definovanie nastavení farbenia. Ak nám neexistuje farbenie aplikácia nás na to upozorní.

Ďalším panelom je panel generovania. Tam máme vstupné pole pre parameter k a

ďalšie tlačidlá pre jednotlivé typy grafu. Stačí zadať k parameter, akú má mať hodnotu a po stlačení tlačidla sa nám vytvorí taký graf, ako pri kopírovaní. Ďalej tam máme možnosť vytvoriť graf z json súboru. Do vstupného pola vložíme buď samotný json a klikneme na tlačidlo "generate from text" alebo vložíme cestu k súboru, ktorý obsahuje json, a klikneme na tlačidlo "generate from path". Ak to nebude korektný vstup a výber aplikácia nás na to upozorní, ináč vytvorí graf.

Z väčšiny UI panelov sa dá odísť pomocou Esc.