Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Riadenie projektu
Tímový projekt VizReal

Predmet: Tímový projekt I.
Členovia tímu: Bc. Nikodém Adler
Bc. Ivana Frankovičová
Bc. Andrej Hoferica
Bc. Michal Jozefek
Bc. Michael Kročka
Bc. Samuel Šouc
Vedúci práce: Ing. Peter Kapec, PhD.
Akademický rok: 2020/2021

Obsah

1 Úvod

Predmet Tímový projekt má za cieľ v praxi naučiť jednotlivé skupiny študentov pracovať ako jeden tím a riadiť tím agilne. Práca v tíme zahŕňa komunikáciu medzi členmi tímu, komunikáciu s product ownerom, organizované stretnutia, plánovanie a rozdeľovanie úloh v projekte.

Náš tímový projekt VizReal sa venuje vizualizácii softvéru vo virtuálnej a rozšírenej realite. Vedúci tímového projektu a product owner, Ing. Peter Kapec, Phd., má za úlohu nás usmerňovať, aby náš tím správne fungoval a zadávať nám úlohy a požiadavky na softvér.

V tomto dokumente sa pokúsime opísať riadenie nášho tímu. Dokument je rozdelený na kapitoly - kapitola 2 Roly členov tímu predstaví náš tím, kapitola 3 Aplikácie manažmentov opíše jednotlivé manažmenty, ktorými sa riadime, kapitola 4 Sumarizácia šprintov zhrnie prvé tri naše absolvované šprinty, kapitola 5 Motivačný dokument je dokument, ktorý sme odovzdávali na začiatku semestra pri výbere tímového projektu a poslednou samostatnou kapitolou je kapitola Metodiky, ktorá pozostáva z jednotlivých našich metodík, ktorými je potrebné sa riadiť.

2 Roly členov tímu

Ing. Peter Kapec, PhD.
- Vedúci tímu
- Vlastník produktu
- Definuje víziu projektu

Bc. Nikodém Adler
- Team leader
- Manažér vývoja
- Unity developer

Bc. Ivana Frankovičová
- Manažér dokumentácie
- Dokumentaristka

Bc. Andrej Hoferica
- Manažér testov
- Tester

Bc. Michal Jozefek
- Scrum master
- Backend developer

Bc. Michael Kročka
- Manažér webového sídla
- Unity developer

Bc. Samuel Šouc
- Dokumentarista
- Backend developer

Podiel práce na dokumentácii k riadeniu projektu	
Názov sekcie	Autor
Úvod	Ivana Frankovičová
Roly členov tímu	Michael Kročka, Ivana Frankovičová
Aplikácie manažmentov	Ivana Frankovičová
Sumarizácia šprintov	Ivana Frankovičová, Michael Kročka, Michal Jozefek
Motivačný dokument	Všetci
Metodiky	Ivana Frankovičová, Michal Jozefek, Andrej Hoferica, minuloročný tím
Webové sídlo	Michael Kročka

Podiel práce na dokumentácii k inžinierskemu dielu	
Názov sekcie	Autor
Úvod	Ivana Frankovičová, tímy z minulých rokov
Globálne ciele projektu	Ivana Frankovičová
Architektúra systému	Ivana Frankovičová, tímy z minulých rokov

Infraštruktúra	Ivana Frankovičová, tímy z minulých rokov
Funkcionalita systému	Ivana Frankovičová, tímy z minulých rokov
Príručky	Ivana Frankovičová, tímy z minulých rokov
Dokumentácia k zdrojovému kódu	Ivana Frankovičová, tímy z minulých rokov
Časté problémy	Ivana Frankovičová, tímy z minulých rokov

Väčšinu metodík sme prebrali od minuloročného tímu. Upravili sme si metodiky Definitions of done, Definitions of ready, Ako komunikovať v tíme, Metodiku iterácií a pridali sme si novú metodiku Metodika záznamov zo stretnutí.

Dokumentáciu k produktu sme od minulého roku menili minimálne, pretože sme zatiaľ nezačali s implementáciou a nijako sme nezasahovali do kódu projektu. Upravili sme iba inštalačnú príručku pre Windows, kde sme pridali posledný bod k častým problémom.

3 Aplikácie manažmentov

Táto kapitola obsahuje opis jednotlivých manažmentov, ktoré využívame pri práci na projekte. Pre každý z nich je taktiež uvedený zoznam príslušných metodík, ktorými sa v rámci daného manažmentu riadime.

3.1 Manažment iterácií

Na projekte pracujeme podľa agilnej metodiky Scrum. Scrum je spôsob riadenia vývoja softvéru, ktorý je založený na báze šprintov (iterácií). Jeden šprint trvá v našom projekte dva týždne. Každý šprint začíname jeho plánovaním, kedy si zadefinujeme a rozdelíme úlohy na nasledujúce dva týždne a odhadneme ich zložitosť. Počas šprintu sa konajú tímové stretnutia (pondelky a piatky) a každú stredu stretnutia s vedúcim projektu (product ownerom). Každý šprint končíme retrospektívou, kde diskutujeme, čo je možné zlepšiť v nasledujúcich šprintoch.

Metodiky:

- Metodika iterácií
- Metodika záznamov zo stretnutí

3.2 Manažment riadenia úloh

Na riadenie úloh sme sa rozhodli použiť nástroj Azure DevOps, do ktorého si zapisujeme úlohy, ktoré sme si definovali počas plánovania šprintu. Pre každú úlohu sa v tomto nástroji nastaví člen tímu, ktorý má úlohu splniť, vieme úlohám nastaviť ich zložitosť, opísať ich a priebežne si značiť stav, v ktorom sa nachádzajú. Úlohy delíme podľa ich abstraktnosti, zložitosti a účelu na user story, epic, feature, backlog a task.

Metodiky:

- Metodika riadenia úloh
- Ako popisovať položky v Azure DevOps

3.3 Manažment prehliadok kódu

Nová funkcionalita vytvorená členom tímu musí prejsť kontrolou predtým, než sa naozaj pridá do projektu. Tieto kontroly prebiehajú na GitLabe a vykonáva ich iný člen tímu. Pri prehliadke kódu sa kontroluje správnosť a funkčnosť kódu, jeho kvalita, príslušné testy a spísaná dokumentácia.

Metodiky:

- Ako vykonať code review

3.4 Manažment verziovania kódu

Projekt VizReal funguje už niekoľko rokov na verziovacom systéme Git, využíva cloudovú službu GitLab a používa sa spôsob verziovania Git Workflow. Bližšie informácie o verziovaní kódu nájdete v príslušných metodikách.

Metodiky:

- Ako verziovať kód a pracovať s gitom

3.5 Manažment dokumentovania

Dokumentácia k projektu, ktorá sa generuje na stránku nášho tímu, je upravovaná na GitLabe. Je rozdelená na časti Architektúra systému, Infraštruktúra, Funkcionalita systému, Príručky, Dokumentácia k zdrojovému kódu, Časté problémy. Dokumentáciu sa snažíme udržiavať aktuálnu a pri jej upravovaní sa každý člen tímu riadi metodikou.

Metodiky:

- Ako písať dokumentáciu

4 Sumarizácia šprintov

4.1 Šprint 1

Trvanie šprintu: 07.10.-21.10. 2020
Doručené story pointy: **Nehodnotený**

Prvý šprint slúžil najmä na oboznámenie sa s projektom. Rozdelili sme roly v tíme, učili sme sa pracovať ako tím a vytvárať si úlohy v Azure DevOps (TFS). Čítali sme si dokumentáciu k projektu a zároveň sme si všetci rozbehávali projekt na svojom zariadení podľa inštalačnej príručky. Pri problémoch sme kontaktovali členov minuloročného tímu, ktorí na tomto projekte pracovali pred nami. V tomto šprinte sme sa ešte rozhodli vynechať estimáciu taskov.

ID	Type	Title	State	Assigned To
760	Task	1. spustenie projektu - Jožo	Done	Bc. Michal Jozefek
766	Task	1. spustenie projektu - Samo	Done	Bc. Samuel Šouc
849	Task	1. spustenie projektu - Mišo	Done	Bc. Michael Kročka
892	Task	1. spustenie projektu - Niko	Done	Bc. Nikodém Adler
893	Task	1. spustenie projektu - Iva	Done	Bc. Ivana Frankovičová
719	Product Backlog Item	Vytvorenie stromovej štruktúry v TFS	Done	Bc. Michal Jozefek
742	Task	Vytvorenie prvotnej stromovej štruktúry (1. šprint)	Done	Bc. Michal Jozefek
720	Product Backlog Item	Metodiky - časť 2	Done	Bc. Michal Jozefek
740	Task	Metodika iterácií	Done	Bc. Michal Jozefek
741	Task	Metodika riadenia úloh	Done	Bc. Michal Jozefek
751	Product Backlog Item	Inštalácia podľa príručky	Done	Bc. Andrej Hoferica
755	Task	Vyriešiť problém s inštaláciou	Done	Bc. Andrej Hoferica
756	Task	Dokončiť inštaláciu	Done	Bc. Andrej Hoferica
752	Product Backlog Item	Kontaktovať člena predchádzajúceho tímu pracujúceho na macOS	Done	Bc. Andrej Hoferica
753	Task	Napísať správu členovi minuloročného tímu, ktorý pracoval na macOS	Done	Bc. Andrej Hoferica
748	Product Backlog Item	Kontaktovať predchádzajúceho manažéra webového sídla	Done	Bc. Michael Kročka
749	Task	Napísať správu predchádzajúcemu manažérovi web sídla	Done	Bc. Michael Kročka

Tabuľka. 1: Export úloh z TFS pre šprint 1

Začať	Pokračovať	Prestať
Retrospektívy vyriešiť posledné stretnutie pred koncom šprintu	3 stretnutia za týždeň	Nedávať všetky úlohy do šprintu len vybrané
Lepšie spravený backlog	Komunikácia v MS teams	Dlhé pauzy medzi monológmi
Vytvárať tasky poriadne (AC, presun stavov)	Forma stretnutí a dodržiavanie agendy	
Lepšie si špecifikovať úlohy s product ownerom		
Začať ohodnocovať tasky		
Rovnomerne rozdeľovať úlohy členom tímu		
Lepšie mastrovať tasky v TFS		

Tabuľka. 2: Retrospektíva pre šprint 1

	Počet naplánovaných taskov	Počet dodaných taskov	Počet dodaných story pointov (tento šprint sme nehodnotili tasky)	Percentuálny podiel dokončenej práce	Hodiny
Nikodém Adler	2	2	X	12,5%	6
Ivana Frankovičová	2	2	X	12,5%	4
Michael Kročka	3	3	X	18,75%	5
Samuel Šouc	2	2	X	12,5%	4
Andrej Hoferica	3	3	X	18,75%	12
Michal Jozefek	6	4	X	25%	3

Obrázok 1 Percentuálny podiel práce za prvý šprint

4.2 Šprint 2

Trvanie šprintu: 21.10.-04.11. 2020
Doručené story pointy: **54**

V druhom šprinte sme analyzovali architektúru projektu, vytvárala sa webová stránka. Ďalej sme analyzovali repozitáre na GitLabe, kde sme sledovali nemergnuté vetvy v nich, ich stav CI a pokrytie dokumentácie. Začali sme pracovať aj na metodikách k projektu.

ID	Type	Title	State	Assigned To
802	Product Backlog Item	Štúdium CI	Done	Bc. Michal Jozefek
803	Task	Príručka CI na gitlabe	Done	Bc. Michal Jozefek
799	Product Backlog Item	Analýza repozitára denev	Done	Bc. Michal Jozefek
800	Task	Analýza repozitára denev	Done	Bc. Michal Jozefek
765	Product Backlog Item	Analýza CI na GitLabe	Done	Bc. Samuel Šouc
767	Task	Prečítať príručku k CI/CD na GitLabe	Done	Bc. Samuel Šouc
768	Task	Zistiť informácie o CI projektu od Jakuba Blaženja	Done	Bc. Samuel Šouc
769	Task	Naštudovať kód CI na GitLabe	Done	Bc. Samuel Šouc
785	Task	Tutoriál na GitLab runner	Done	Bc. Samuel Šouc
788	Task	Napísať report	Done	Bc. Samuel Šouc
798	Task	Preštudovať dokumentáciu k CI	Done	Bc. Samuel Šouc
787	Product Backlog Item	Analýza architektúry [Niko]	Done	Bc. Nikodém Adler
827	Task	Analyzovať architektúru	Done	Bc. Nikodém Adler
789	Product Backlog Item	Analýza architektúry [Ivka]	Done	Bc. Ivana Frankovičová
826	Task	Analyzovať architektúru	Done	Bc. Ivana Frankovičová
790	Product Backlog Item	Luadb	Done	Bc. Michael Kročka
791	Task	Luadb - Analyzovať pokrytie dokumentácie	Done	Bc. Michael Kročka
792	Task	Luadb - Analyzovať stav CI	Done	Bc. Michael Kročka
793	Task	Luadb - Nemergnuté branche	Done	Bc. Michael Kročka
794	Product Backlog Item	Luametrics	Done	Bc. Michael Kročka
796	Task	Luametrics - Analyzovať stav CI	Done	Bc. Michael Kročka
795	Task	Luametrics - Analyzovať pokrytie dokumentácie	Done	Bc. Michael Kročka
797	Task	Luametrics - Nemergnuté branche	Done	Bc. Michael Kročka
720	Product Backlog Item	Metodiky - časť 2	Done	Bc. Michal Jozefek
730	Task	Definitions of done	Done	Bc. Michal Jozefek
731	Task	Definitions of ready	Done	Bc. Michal Jozefek
915	Task	Metodika záznamov zo stretnutí	In Progress	Bc. Andrej Hoferica
783	Product Backlog Item	Naklonovanie dokumentácie	Done	Bc. Michal Jozefek
784	Task	Naklonovať	Done	Bc. Michal Jozefek
829	Product Backlog Item	Luameg	Done	Bc. Andrej Hoferica

834	Task	Luameg - Analyzovať dokumentáciu	Done	Bc. Andrej Hoferica
835	Task	Luameg - Analyzovať CI	Done	Bc. Andrej Hoferica
836	Task	Luameg - Analyzovať nemergnuté branche	Done	Bc. Andrej Hoferica
831	Product Backlog Item	Luagit	Done	Bc. Andrej Hoferica
838	Task	Luagit - Analyzovať dokumentáciu	Done	Bc. Andrej Hoferica
841	Task	Luagit - Analyzovať CI	Done	Bc. Andrej Hoferica
844	Task	Luagit - Analyzovať nemergnuté branche	Done	Bc. Andrej Hoferica
715	Product Backlog Item	Vytvorenie webstránky	Done	Bc. Michael Kročka
761	Task	Analýza Documentation z Gitlabu	Done	Bc. Michael Kročka
762	Task	Update obsahu stránky	Done	Bc. Michael Kročka
764	Task	Naklonovanie repozitára Documentation	Done	Bc. Michael Kročka
777	Task	Vytvorenie stromovej štruktúry a vloženie obsahu dokumentu	Done	Bc. Michael Kročka
763	Product Backlog Item	Deployment webstránky	Done	Bc. Michael Kročka
781	Task	Deployment stránky	Done	Bc. Michael Kročka
782	Task	Nastavenie správnej domény	Done	Bc. Michael Kročka
786	Task	Upratanie github-pages	Done	Bc. Michael Kročka
713	Product Backlog Item	Aktualizácia pri zaheslovanom SSH kľúči	Done	Bc. Michael Kročka
775	Task	Vytvorenie postupu riešenia problému pri zaheslovanom SSH kľúči	Done	Bc. Michael Kročka
832	Product Backlog Item	Luatree	Done	Bc. Andrej Hoferica
842	Task	Luatree - Analyzovať CI	Done	Bc. Andrej Hoferica
839	Task	Luatree – Analyzovať dokumentáciu	Done	Bc. Andrej Hoferica
845	Task	Luatree - Analyzovať nemergnuté branche	Done	Bc. Andrej Hoferica
778	Product Backlog Item	Analýza Lua zdrojových kódov	Done	Bc. Andrej Hoferica
779	Task	Analyzovať generovanie grafov	Done	Bc. Andrej Hoferica
780	Task	Analyzovať účele Lua repozitárov	Done	Bc. Andrej Hoferica
804	Product Backlog Item	CI Images	Done	Bc. Samuel Šouc
806	Task	CI Images - Analyzovať stav CI	Done	Bc. Samuel Šouc
807	Task	CI Images - Analyzovať pokrytie dokumentácie	Done	Bc. Samuel Šouc
808	Task	CI Images - Nemergnuté branche	Done	Bc. Samuel Šouc
812	Task	CI Images - Správa o stave	Done	Bc. Samuel Šouc
805	Product Backlog Item	Documentation	Done	Bc. Samuel Šouc
809	Task	Documentation - Analyzovať stav CI	Done	Bc. Samuel Šouc

810	Task	Documentation - Analyzovať pokrytie dokumentácie	Done	Bc. Samuel Šouc
811	Task	Documentation - Nemergnuté branche	Done	Bc. Samuel Šouc
813	Task	Documentation - správa o stave	Done	Bc. Samuel Šouc
814	Product Backlog Item	LuaInterface	Done	Bc. Ivana Frankovičová
820	Task	LuaInterface - Analyzovať pokrytie dokumentácie	Done	Bc. Ivana Frankovičová
821	Task	LuaInterface - Analyzovať stav CI	Done	Bc. Ivana Frankovičová
822	Task	LuaInterface - Nemergnuté branche	Done	Bc. Ivana Frankovičová
815	Product Backlog Item	LuaGraph	Done	Bc. Ivana Frankovičová
817	Task	LuaGraph - Analyzovať pokrytie dokumentácie	Done	Bc. Ivana Frankovičová
818	Task	LuaGraph - Analyzovať stav CI	Done	Bc. Ivana Frankovičová
819	Task	LuaGraph - Nemergnuté branche	Done	Bc. Ivana Frankovičová
816	Product Backlog Item	3DSoftviz_remake	Done	Bc. Nikodém Adler
823	Task	3DSoftviz_remake - Analyzovať pokrytie dokumentácie	Done	Bc. Nikodém Adler
824	Task	3DSoftviz_remake - Analyzovať stav CI	Done	Bc. Nikodém Adler
825	Task	3DSoftviz_remake - Nemergnuté branche	Done	Bc. Nikodém Adler
830	Task	Terra	Done	Bc. Andrej Hoferica
837	Task	Terra - Analyzovať dokumentáciu	Done	Bc. Andrej Hoferica
840	Task	Terra - Analyzovať CI	Done	Bc. Andrej Hoferica
843	Task	Terra - Analyzovať nemergnuté branche	Done	Bc. Andrej Hoferica

Tabuľka. 3: Export úloh z TFS pre šprint 2

Začať	Pokračovať	Skončiť
Viac špecifikovať úlohy	Forma stretnutí	Meškať na stretnutia
Posúvať si v TFS aj User Stories do iných stavov, nielen Tasky	Poker počas prvého tímového stretnutia v šprinte	Dorábať veci na poslednú chvílu! (TFS, agendy, ...)
Posunúť čas stretnutí v piatok na 7:30		
Na začiatku a na konci stretnutí sumarizácia		
Upravovať zápisy zo stretnutí – ideálne na po konci stretnutia		
Pripraviť šablónu na zápisy zo stretnutí		

Častejšie čítať / odpovedať na MS Teams		
Produktívnejší počas prvej polovice šprintu		

Tabuľka. 4: Retrospektíva pre šprint 2

	Počet naplánovaných taskov	Počet dodaných taskov	Počet dodaných story pointov	Percentuálny podiel dokončenej práce (vzhľadom na story points)	Hodiny
Nikodém Adler	4	4	9	16,66%	6
Ivana Frankovičová	7	7	10	18,51%	6
Michael Kročka	14	14	11	20,37%	8
Samuel Šouc	14	14	7	12,96%	7
Andrej Hoferica	13	13	9	16,66%	4,5
Michal Jozefek	5	5	8	14,81%	4

Obrázok 2 Percentuálny podiel práce za druhý šprint

4.3 Šprint 3

Trvanie šprintu: 04.11.-18.11. 2020
Doručené story pointy: **29**

V treťom šprinte sme sa venovali mergovaniu vetiev na GitLabe, riešili sa problémy so serverom a s GitLab runnerom na CI. Okrem toho sa začal vytvárať tento dokument, riešili sa problémy s testami a dockerom a vytvoril sa skript na generovanie LuaDB grafov. Analyzovali sa dll knižnice, kde sa zistilo, kde sa aké knižnice používajú a ako často.

ID	Type	Title	State	Assigned To
868	Product Backlog Item	Prvotné spustenie testov	Done	Bc. Andrej Hoferica
869	Task	Spustiť test 'dir_files_spec.lua' z repozitáru luadb	Done	Bc. Andrej Hoferica
870	Task	Modifikovať test tak, aby poskytoval viac DEBUG výpisov	Done	Bc. Andrej Hoferica
871	Task	Vyriešiť problém s metódou describe()	Done	Bc. Andrej Hoferica
859	Product Backlog Item	Asistencia pri instalicii CI	Done	Bc. Michal Jozefek

860	Task	Instalacia CI na servery so Samom	Done	Bc. Michal Jozefek
874	Product Backlog Item	Doplnenie informacií o CI do tabuľky repozitárov	Done	Bc. Samuel Šouc
895	Task	Luadb	Done	Bc. Samuel Šouc
896	Task	3dsv_remake	Done	Bc. Samuel Šouc
897	Task	Luametrics	Done	Bc. Samuel Šouc
898	Task	documentation	Done	Bc. Samuel Šouc
899	Task	luameg	Done	Bc. Samuel Šouc
900	Task	luatree	Done	Bc. Samuel Šouc
901	Task	luagit	Done	Bc. Samuel Šouc
902	Task	luagraph	Done	Bc. Samuel Šouc
903	Task	luainterface	Done	Bc. Samuel Šouc
884	Product Backlog Item	Výmena servera	Done	Bc. Michal Jozefek
885	Task	Informovanie K.Kostala	Done	Bc. Michal Jozefek
886	Task	Ziskat pristup od minuleho teamu	Done	Bc. Samuel Šouc
889	Task	Zmena DNS	Done	Bc. Michal Jozefek
890	Task	Napisat spravcom ohladom Virtualky minulorcneho teamu	Done	Bc. Michal Jozefek
863	Product Backlog Item	Oprava Metodik z minuleho roka	Done	Bc. Michal Jozefek
865	Task	Definition of Done	Done	Bc. Michal Jozefek
866	Task	Definition of ready	Done	Bc. Michal Jozefek
864	Task	Poslanie Misovi na stranku	Done	Bc. Michal Jozefek
916	Product Backlog Item	Asistencia pri dokumentácia na odovzdanie	Done	Bc. Michael Kročka
917	Task	Formálna úprava časti - Inžinierske dielo	Done	Bc. Michael Kročka
918	Task	Formálna úprava časti - Riadenie projektu	Done	Bc. Michael Kročka
919	Task	Doplnenie časti - Riadenie projektu - Motivačný dokument	Done	Bc. Michael Kročka
910	Product Backlog Item	Identifikácia miest v C# kde su využité C++ importy	Done	Bc. Nikodém Adler
905	Task	LuaObject	Done	Bc. Nikodém Adler
906	Task	GraphObject	Done	Bc. Nikodém Adler
907	Task	LuaEdge	Done	Bc. Nikodém Adler
908	Task	LuaIncidence	Done	Bc. Nikodém Adler
909	Task	LuaNode	Done	Bc. Nikodém Adler
911	Task	LuaGraph	Done	Bc. Nikodém Adler
912	Task	LuaInterface	Done	Bc. Nikodém Adler
847	Product Backlog Item	3dsoftvis_remake	Done	Bc. Nikodém Adler
875	Task	danis	Done	Bc. Nikodém Adler
876	Task	borecky	Done	Bc. Nikodém Adler
877	Task	stevlik	To Do	Bc. Nikodém Adler
878	Task	karas	To Do	

879	Task	kulbak	To Do	
862	Product Backlog Item	Dokumentácia na odovzdanie	Done	Bc. Michal Jozefek
883	Task	Zoznam na odovzdanie	Done	Bc. Michal Jozefek
888	Task	Sformovanie dokumentu na odovzdanie	Done	Bc. Ivana Frankovičová
848	Product Backlog Item	luadb	Done	Bc. Ivana Frankovičová
880	Task	stevlik	To Do	
881	Task	kratky	Done	Bc. Ivana Frankovičová
851	Product Backlog Item	Rozbehanie devenv	Done	Bc. Michael Kročka
852	Task	Rozbehať devenv	Done	Bc. Michael Kročka
854	Task	$'\r': command not found	Done	Bc. Michael Kročka
857	Task	module 'lrdb_server' not found	Done	Bc. Michael Kročka
872	Product Backlog Item	Skript - generovanie luadb grafov zo "spec"	Done	Bc. Michael Kročka
858	Task	Vytvorenie skriptu	Done	Bc. Michael Kročka
873	Task	Nezaradené hrany a uzly	Done	Bc. Michael Kročka
891	Task	Branch, Commit, Merge Gitlab	Done	Bc. Michael Kročka
894	Task	Zlyhanie CI testov	Done	Bc. Michael Kročka
850	Product Backlog Item	luametrics	Done	Bc. Ivana Frankovičová
882	Task	kratky	Done	Bc. Ivana Frankovičová
914	Product Backlog Item	Zápisy zo stretnutí	Done	Bc. Ivana Frankovičová
920	Task	Tímové stretnutia	Done	Bc. Ivana Frankovičová
921	Task	Stretnutia s vedúcim	Done	Bc. Ivana Frankovičová
922	Task	Doplnenie chýbajúcich častí zápisov	Done	Bc. Samuel Šouc

Tabuľka. 5: Export úloh z TFS pre šprint 3

Začať	Pokračovať	Skončiť
Rozdeľovať konverzácie podľa témy	Na začiatku a na konci stretnutí sumarizácia	Prestať písať všetko do jedného kanála
Označovať ľudí v konverzáciách ak sa ich to týka		

Nahadzovať tasky do TFS predtým, než na ňom začne me pracovať		
Upravovať zápisy zo stretnutí		
Začať sa riadiť retrospektívou minulého šprintu		

Tabuľka. 6: Retrospektíva pre šprint 3

	Počet naplánovaných taskov	Počet dodaných taskov	Počet dodaných story pointov	Percentuálny podiel dokončenej práce	Hodiny
Nikodém Adler	10	7	2	6,98%	6
Ivana Frankovičová	5	5	6	20,68%	7
Michael Kročka	10	10	9	31,03%	14
Samuel Šouc	11	10	2	6,98%	3
Andrej Hoferica	3	3	3	10,34%	5
Michal Jozefek	8	8	7	24,13%	6

Obrázok 3 Percentuálny podiel práce za tretí šprint

5 Motivačný dokument

5.1 Predstavenie tímu

Náš tím sa skladá zo šiestich členov. Nikodém Adler, Ivana Frankovičová, Samuel Šouc, Michal Jozefek, Michael Kročka a Andrej Hoferica. Piati z nás študujeme inteligentné softvérové systémy a jeden študuje internetové technológie. Náš tím nie je zameraný len na jednu oblasť informatiky, ale pokrývame široké spektrum predmetov, ktoré sa vyučujú na inžinierskom štúdiu od oblasti počítačového videnia, cez spracovanie veľkých dát až po vnorené systémy.

Väčšina členov tímu už má predchádzajúce skúsenosti s prácou na tímových projektoch, teda plánovanie projektu, stand-upy a práca s viacerými ľuďmi pre nás nie sú novinka a určite nebudeme mať problém pomôcť nabehnúť do tohto kolotoča členom, ktorí s tímovým projektom skúsenosť nemajú.

Rovnako ako zameranie našich predmetov v škole, aj naše predchádzajúce skúsenosti mimo školy sú rôznorodé:

Niko sa mimo školy venuje tvorbe mobilných aplikácií pre rozšírenú realitu na platformy Android a iOS v prostredí Unity. Práci s Unity sa venuje aj vo svojom voľnom čase. Zapísal si predmet Návrh a vývoj počítačových hier, ktorý mu v Unity ešte viac rozšíri jeho znalosti.

Ivka ma skúsenosti s tvorbou webových aplikácií v ASP.NET Core a vo voľnom čase sa venuje UX dizajnu a navrhovaniu používateľských rozhraní. Taktiež má zapísaný predmet Návrh a vývoj počítačových hier, kde rozšíri svoje znalosti aj v tomto smere.

Aďo sa venuje automatizovanému testovaniu - hlavne regresnému, ale sčasti aj jednotkovému testovaniu. Robí v TCL a popritom skripty v Bashi. Vo voľnom čase sa venuje Pythonu (knižnice ako pandas, numpy, selenium).

Mišo má skúsenosti s umelou inteligenciou a strojovým učením, tvorbou webových stránok a viacvrstvových aplikácii (JAVA a JEE). Tiež sa hrabe v Pythone, konkrétne knižnice pandas, numpy, scikit-learn, keras a tensorflow. Zaujíma ho spracovanie dát, čo je dôvodom, prečo si vybral aj predmet Vyhľadávanie informácií.

Jožo (Michal Jozefek) má skúsenosti s tvorbou mobilných aplikácií pre Android, tvorbou databáz, programovaním backendu a prácou so serverom pomocou XAMMPu. Na svoju prácu používa najmä jazyky Java a PHP.

Samo má skúsenosti najmä v oblasti práce s databázami, konkrétne PostgreSQL a MySQL. Taktiež v rámci náplne práce v zamestnaní vykonáva úpravu tabuľkových dát v Pythone s knižnicami pandas a numpy. V rámci súčasného zamestnania naprogramoval v Node.js middleware, ktorý pomohol pri prechode na nový systém.

5.2 Motivácia - Téma č. 18 – FIFé International Cat Show [MIAOW]

Téma nás oslovila svojou všestrannosťou, vytvoriť taký komplexný inteligentný informačný systém znie ako skvelá výzva, v ktorej by sa náš tím rád angažoval pomocou vlastných prínosov. Zároveň spolupráca s priemyselným partnerom by bola užitočnou skúsenosťou, ktorá by nám mohla pomôcť v budúcom kariérnom živote. Na takúto spoluprácu sme sa všetci pripravovali do určitej miery na

niektorých predmetoch bakalárskeho štúdia, ale najmä v praxi, keďže väčšina členov tímu už pracuje a stretli sme sa s vytváraním softvéru na mieru podľa požiadaviek zákazníka, resp. zadávateľa. Náš tím má pokročilé znalosti v spracovaní údajov a práci s databázami, taktiež máme skúsenosti s vývojom mobilných aplikácií a práve to je jedna z oblastí, ktoré preferujeme pri výbere tímového projektu. Viacerí z nás si tiež osvojili hlbšie poznatky o návrhu užívateľského rozhrania vo voliteľných predmetoch, pričom všetci sme absolvovali predmet zameraný na fundamentálny návrh užívateľského rozhrania. Rovnako sme mali možnosť pracovať na vývoji elementárneho informačného systému na predmete bakalárskeho štúdia, ktorý využíval architektúru rozhrania REST. Myslíme si, že naše poznatky a skúsenosti získané či už v škole alebo mimo nej sú dostačujúce na realizáciu tohto projektu. Veľmi zaujímavo znie aj požiadavka na komplexný projektový manažment a hoci náš tím doposiaľ nemá skúsenosti s manažmentom projektov, takúto úlohu by sme si radi vyskúšali. Pozitívne hodnotíme fakt, že zadávateľ má presnú predstavu o tom, aké funkcie by mal systém zabezpečiť. Vďaka tomu bude náš tím schopný tieto požiadavky naplniť tak, aby bol zadávateľ čo najviac spokojný.

5.3 Motivácia - Téma č. 17 – VR laboratórium pre dištančné vzdelávanie [VRLab]

Pre niektorých členov tímu sa tato téma nachádzala na prvom mieste, no po dlhšej konzultácii sme sa ju rozhodli zaradiť na druhé miesto. Na našom liste priorít sa nachádza tak vysoko z dvoch dôvodov:
1. Téma je veľmi zaujímavá. (Ako často môžete vyvíjať aplikácie pre rozšírenú realitu?)
2. Schopnosti, skúsenosti a preferencie nášho tímu by sa skvelo hodili na riešenie tohto projektu.

Prečo je tato téma podľa nás zaujímavá? Pretože oblasť vývoja aplikácii a hier pre virtuálnu realitu je nové a rýchlo sa rozvíjajúce odvetvie, v ktorom by sme radi nabrali nové, niektorí prvé skúsenosti. Ďalej vývoj hier je tiež zaujímavé odvetvie, v ktorom sa niektorí členovia tímu (Ivana a Niko) plánujú pohybovať aj ďalej, ale ani pre zvyšok tímu to nie je oblasť, v ktorej by sa báli ako Danko diktátu, práve naopak, radi by sa v nej začali pohybovať. Ale teraz k bodu dva. Najväčšie skúsenosti s touto témou má z nášho tímu Niko. Vývoju aplikácii do rozšírenej reality sa venuje už cez rok aj na profesionálnej úrovni, v startupe Cviker, vyvíjajú aplikáciu Akular v Unity3D. Iste rozšírená realita ≠ virtuálna realita, no minimálne má bohaté skúsenosti s prostredím Unity, ktorému sa venuje aj vo voľnom čase a ktoré bude predstavovať hlavný nastroj na vývoj tejto aplikácie (je spomenuté v zadaní a za náš tím je to skvelé prostredie). Tiež má zapísaný predmet Návrh a vývoj počítačových hier, rovnako ako ďalší člen tímu Ivana, ktorá má už tiež skúsenosti s Unity, aj keď nie na úrovni Nika.

Ostatní členovia tímu nemajú predchádzajúce skúsenosti s Unity, alebo vývojom hier, ale nemali by problém sa to naučiť a vyjadrili nadšenie z tejto témy. Všetci už ale majú skúsenosti s vývojom aplikácii alebo majú iné skúsenosti a predispozície, ktoré by sme vedeli využiť niekde inde vo vývoji.

Naša predstava riešenia projektu, aj keď je ešte otvorená na diskusiu a prípadné úpravy, vyzerá zhruba nejako takto. Aplikácia by sa vyvíjala pre virtuálnu realitu podľa headsetov, ktoré by boli k dispozícii, no tiež by sme chceli spraviť jednoduchšiu verziu spustiteľnú na mobilných zariadeniach pre možnosť zachytenia väčšieho publika. Do aplikácie, laboratória, by sa vedelo pripojiť viacero žiakov naraz, prípadne aj učiteľ/učitelia. Aplikácia by však mala obsahovať aj automatického učiteľa, pomocníka, ktorý by zadával experimenty, vysvetľoval, ktoré zariadenie slúži na čo a podobne. Úlohou žiakov by potom bolo vykonávať tieto experimenty vo virtuálnom laboratóriu. Samotne laboratórium by bolo realizované ako sandbox, všetky potrebné veci by sa nachádzali v laboratóriu a bolo by úlohou žiakov ich nájsť a správne dať dokopy, žiaci by mali možnosť spraviť pri experimentoch chyby s ich prípadnými následkami, nebezpečie by im hroziť nemalo a vedelo by to lepšie nasimulovať reálne

prostredie, kde asi nebudú mať k dispozícii iba úzku časť pomôcok, ktoré iste súvisia s experimentom. Naše riešenie ale ešte nie je vyryté do kameňa a je otvorené na diskusiu.

5.4 Motivácia - Téma č. 4 – Educational Content Engineering Hub – Databáza otázok, odpovedí, úloh a riešení [ECEH-DU]

Téma nás zaujala najmä svojou hlavnou myšlienkou. S členmi tímu sme sa zhodli, že takýto typ databanky otázok a úloh by bol veľmi nápomocný študentom počas ich štúdia. Všímame si najmä potenciál projektu. Zjednodušením procesu učenia by študenti ušetrili pre nich drahocenný čas, čo by mohlo viesť ku skvalitneniu výsledkov a spokojnosti študentov. Sami by sme existenciu databanky s konečnou množinou otázok na skúšky uvítali aj na našej fakulte. Ďalším faktorom, ktorý prispel k vysokej preferencii danej témy je jej realizácia vo forme online prostredia. Pri prvom prečítaní zadania si členovia nášho tímu vedeli predstaviť pozície, ktoré by pri práci na projekte zastávali a zoznam úloh, ktoré by počas práce museli plniť. Náš tím disponuje skúsenosťami s prácou s databázami a so všetkými uvedenými povinnými technológiami. S návrhom architektúry systému sa stretli všetci členovia tímu počas bakalárskeho štúdia. Myslíme si, že sme dostatočne schopní a skúsení na realizáciu tohto projektu. Za pozitívum a zároveň výzvu vnímame fakt, že pôjde o pokračovanie projektu, čo môže priniesť veľmi dobrú a praktickú skúsenosť do pracovného života po ukončení štúdia. Zároveň predstava o tom, že by nami pridaná funkcionalita napomohla k uvedeniu databanky do prevádzky na niektorej škole alebo fakulte je dostatočnou motiváciou.

5.5 Príloha A - Preferencie tém

1. Téma 18 - FIFé International Cat Show [MIAOW]
2. Téma 17 - VR laboratórium pre dištančné vzdelávanie [VRLab]
3. Téma 4 - Educational Content Engineering Hub - Databáza otázok, odpovedí, úloh a riešení [ECEH-DU]
4. Téma 8 - Automatické rozpoznávanie spektier [ARS]
5. Téma 3 - Vizualizácia softvéru vo virtuálnej a rozšírenej realite [VizReal]
6. Téma 5 - Educational and coworking driven orchestration portal [EDUCO]
7. Téma 14 - Platforma pre sledovanie dodávateľského reťazca s využitím technológie blockchain [S-Chain]
8. Téma 19 - Podporný informačný systém pre študijné oddelenie [CROSS-CHECK]
9. Téma 16 - Inteligentný informačný systém zameraný na projektový manažment a automatizáciu procesu verejného obstarávania [iPP]
10. Téma 1 - Blockchain platobné brány [BlockPay]
11. Téma 7 - Vnorený systém pre zabezpečený zber dát [DSC]
12. Téma 10 - Game-chain: Ako bezpečne vymieňať herné účty
13. Téma 9 - Vzdialené monitorovanie zdravotného stavu človeka pomocou E-Health
14. Téma 6 - Transformácia priestorov na bezpečné a inteligentné miesta na prácu [SmartSpace]
15. Téma 12 - Safety panel a spätná analýza údajov pre vývoj autonómneho vozidla [avPANEL]
16. Téma 11 - Cyber Range: Simulačné prostredie pre testovanie kybernetickej ochrany [CYRAN]
17. Téma 13 - Korekcia dynamických vlastností virtuálnych modelov komponentov vozidiel [CarComponents]
18. Téma 2 - Webové IDE pre ASIC [ASICDE]
19. Téma 15 - Webový vyhľadávač podobnosti [AntiPlag]

5.5 *Príloha B – Rozvrh tímu*

Obrázok 4 Rozvrh tímu

Metodiky

1. Definície

1.1 Definitions of done

User story

- Je splnená definition of ready
- Je priradená členovi teamu, ktorý ma na starosti jej dodanie/vyriešenie
- Všetky tasky sú v stave Done
- AC user story sú splnené
- Výsledok story je odprezentovaný teamu
- Product owner schválil na stretnutí schválenie user story

Šprint

- Všetky úlohy v šprinte spĺňajú definition of done
- Projekt je nasadený
- Backlog je aktualizovaný
- Šprint je zdokumentovaný
- Na webovej stránke tímového projektu je nahrané sprint review, retrospektíva, export úloh a percentuálny podiel práce

Web

- Na webe sú všetky potrebné informácie o projekte a tíme
- Na webe sú všetky potrebné súbory (retrospektívy, šprint reporty)
- Sú splnené akceptačné kritéria webu
- Web je prístupný cez internet

Analytická úloha

- Je vytvorená dokumentácia k analyzovanému problému alebo report o nájdených riešeniach

Implementačná úloha

- Napísaný a funkčný kód pre určené funkcionality
- V kóde je logovanie
- Vytvorené testy
- Kód prejde jednotkovými/funkcionálnymi testami
- Splnené akceptačné kritéria
- Revízia kódu prebehla úspešne
- Kód je mergnutý do develop branche a schválený product ownerom
- Kód je zdokumentovaný
- Ak je to vhodné/potrebné používateľská príručka je aktualizovaná

Úloha týkajúca sa infraštruktúry

- Je funkčná celá pipeline
- Je spísaná/upravená dokumentácia
- V prípade, že pipeline produkuje artefakt/artefakty, je potrebné ich niekde uložiť

Testy

- Testy sú napísané podľa metodiky na písanie testov

Dokumentačná úloha

- Dokumentácia prešla review
- Je mergnutá v develop vetve
- Je vygenerovaná celá dokumentácia
- Dokumentácia je uplodnutá na webovú stránku tímu

1.2 Definitions of ready

Epic

- Má zapísaný opis, ktorý v širšom kontexte vysvetľuje prínos
- Je akceptovaný product ownerom
- Má vytvorenú aspoň jednu feature

Feature

- Má zapísaný opis, ktorý užšie špecifikuje pridávanú funkcionalitu, čo presne má funkcionalita robiť
- Je akceptovaná product ownerom
- Je zaradená do epicu
- Má vytvorenú aspoň jednu user story

User story

- Má zapísané akceptačné kritériá, ktoré sú merateľné
- Má zapísaný odhad zložitosti
- Sú zapísané jednotlivé tasky pre túto user story
- Tasky majú zapísané description
- Je zaradená do feature a epicu

2. Ako popisovať položky v Azure DevOps

BUG

1. TITLE

- Názov bug reportu musí byť stručný a jasný (pointa problému musí byť obsiahnutá v čo najmenej slovách a nesmie obsahovať iné než kľúčové informácie pre uvedenie čitateľa do kontextu)

2. REPRO STEPS
    - Celý názov - steps to reproduce
    - Musí obsahovať kroky, ktoré je nutné vykonať, aby bolo možné bug zreprodukovať
    - Kroky musia byť napísané v poradí, v ktorom musia byť vykonané
    - Kroky musia obsahovať do detailu úplne všetky informácie, ktoré sú potrebné k tomu, aby bolo možné problém zreprodukovať (ak sa vám javí nejaká informácia ako samozrejmosť, napriek tomu ju uveďte - nikdy neviete, aké sú vedomosti osoby, ktorá bude s vaším bug reportom pracovať)
3. SYSTEM INFO
    - Informácie o softvéri a konfigurácii systému, ktoré sú relevantné pre uskutočnenie testu.
4. ACCEPTANCE CRITERIA
    - Poskytnite kritériá, ktoré je potrebné splniť skôr, ako bude možné chybu alebo príbeh používateľa uzavrieť.
    - Pred začiatkom prác opíšte čo najjasnejšie akceptačné kritéria zákazníka.
    - Akceptačné kritériá sa môžu použiť ako základ pre akceptačné testy, aby ste mohli efektívnejšie vyhodnotiť, či bola položka uspokojivo dokončená.
5. FOUND IN BUILD
    - Keď správca testov vytvorí chyby, automaticky vyplní *System info* a *found in build* informáciami o softvérovom prostredí a o tom, kde sa chyba vyskytla. Pre viac informácií o definovaní softvérových prostredí, navštívte stránku: https://docs.microsoft.com/en-us/azure/devops/test/test-different-configurations?view=azure-devops
6. INTEGRATED IN BUILD
    - Keď vyriešite chybu, použite Integrated in Build na označenie názvu buildu, ktorý obsahuje kód, ktorý opravuje chybu.
7. PRIORITY
    - Subjektívne hodnotenie chyby, pretože sa týka podnikania alebo požiadaviek zákazníka. Priorita označuje poradie, v ktorom by sa mali opraviť chyby kódu.
    - **Akceptované hodnoty**
        - **1**: Produkt sa nedá odoslať bez úspešného vyriešenia pracovného predmetu a mal by sa riešiť čo najskôr.
        - **2**: Produkt sa nedá odoslať bez úspešného vyriešenia predmetu práce, nemusí sa však okamžite riešiť.

- **3**: Vyriešenie pracovnej položky je voliteľné na základe zdrojov, času a rizika.

8. SEVERITY
    - Subjektívne hodnotenie vplyvu chyby na projekt alebo softvérový systém. Napríklad: Ak kliknutie na vzdialený odkaz (zriedkavá udalosť) spôsobí zlyhanie aplikácie alebo webovej stránky (závažná skúsenosť so zákazníkom), môžete určiť Závažnosť = 2 (vysoká) a Priorita = 3.
    - **Akceptované hodnoty**
        - **1 - Kritická**: Musí sa opraviť. Porucha, ktorá spôsobuje ukončenie jedného alebo viacerých komponentov systému alebo celého systému alebo spôsobuje rozsiahle poškodenie údajov. Neexistujú žiadne prijateľné alternatívne metódy na dosiahnutie požadovaných výsledkov.
        - **2 - Vysoká**: Zvážte opravu. Porucha, ktorá spôsobuje ukončenie jedného alebo viacerých komponentov systému alebo celého systému alebo spôsobuje rozsiahle poškodenie údajov. Existuje však prijateľná alternatívna metóda na dosiahnutie požadovaných výsledkov.
        - **3 - Stredná**: Porucha, ktorá spôsobuje, že systém poskytuje nesprávne, neúplné alebo nekonzistentné výsledky.
        - **4 - Nízka**: Menší alebo kozmetický defekt, ktorý má prijateľné riešenie na dosiahnutie požadovaných výsledkov.

EPIC

1. DESCRIPTION
    - Jasne a podrobne popíšte cieľ epicu.
2. ACCEPTANCE CRITERIA
    - Jasne definovať, kedy je epic hotový a všetko funguje presne tak, ako to bolo na začiatku definované.
3. STATUS
    - **start date** : Dátum začiatku práce na epicu
    - **target date** : Dátum, do ktorého sa má epic implementovať.
4. PRIORITY
    - Subjektívne hodnotenie
    - Výber zo štyroch hodnôt podľa priority - **1 (najvyššia)** až **4 (najnižšia)**
5. EFFORT
    - Poskytnite relatívny odhad množstva práce potrebnej na dokončenie epicu. Použite akúkoľvek číselnú jednotku merania, na ktorej ste sa s tímom dohodli.

6. BUSINESS VALUE
    - Zadajte prioritu, ktorá zachytáva relatívnu hodnotu epicu v porovnaní s ostatnými položkami rovnakého typu. Čím vyššie číslo, tým vyššia je hodnota podniku.
    - Toto pole použite, ak chcete zachytiť prioritu osobitne od meniteľného poradia nevybavených položiek backlogu.
7. TIME CRITICALITY
    - Subjektívna merná jednotka, ktorá zachytáva, ako sa biznis hodnota epicu v priebehu času znižuje. Vyššie hodnoty znamenajú, že epic je zo svojej podstaty časovokritickejší ako položky s nižšími hodnotami.
8. VALUE AREA
    - Oblasť hodnoty pre zákazníka určená epicom.
    - **Zahrnuté hodnoty**
        - **Architectural** : technické služby na implementáciu obchodných funkcií, ktoré poskytujú riešenie
        - **Business** : služby, ktoré napĺňajú potreby zákazníkov alebo zainteresovaných strán a ktoré priamo prinášajú hodnotu pre zákazníka na podporu podnikania

FEATURE

- Rovnaká štruktúra a pravidlá ako pri epicu (Viď popis epicu).

IMPEDIMENT

- Impedimenty (prekážky) predstavujú neplánované činnosti. Ich riešenie si vyžaduje viac práce nad rámec toho, čo sa sleduje podľa skutočných požiadaviek.
- Použitie typu pracovnej položky impediment vám pomáha sledovať a spravovať tieto problémy, až kým ich nevyriešite a nezatvoríte.
- Impedimenty sa nezobrazujú v backlogu. Namiesto toho je možné ich sledovať použitím dopytov.
    a. DESCRIPTION
        - Jasne a podrobne popíšte prekážku.
    b. RESOLUTION
        - Jasne definovať okolnosti, za akých môže byť daná prekážka považovaná za plne vyriešenú.
    c. PRIORITY
        - Subjektívne hodnotenie
        - Výber zo štyroch hodnôt podľa priority - **1 (najvyššia)** až **4 (najnižšia)**

PRODUCT BACKLOG ITEM

- Rovnaká štruktúra a pravidlá ako pri epicu (Viď popis epicu).

TASK

- Do backlogu pridávate tasky (úlohy), keď chcete sledovať prácu potrebnú na ich implementáciu a odhadnúť prácu priradenú jednotlivým členom tímu a tímu ako celku.
- Ak chcete porovnať kapacitu so skutočne plánovanou prácou, musíte pre každú nevybavenú položku definovať a odhadnúť tasky.
    a. DESCRIPTION
        - Jasne a podrobne popíšte úlohu.
    b. PRIORITY
        - Subjektívne hodnotenie
        - Výber zo štyroch hodnôt podľa priority - **1 (najvyššia)** až **4 (najnižšia)**
    c. REMAINING WORK
        - Množstvo zostávajúcej práce na dokončenie úlohy. Postupom práce aktualizujte toto pole. Používa sa na výpočet kapacitných diagramov a grafu burzového šprintu. Môžete určiť prácu v akejkoľvek meracej jednotke, ktorú si tím vyberie.
    d. BLOCKED
        - V nastavení scrum alebo agile je „blokovaný“ stav pracovnej položky, ktorú nemôžete dokončiť z dôvodu vonkajších faktorov. Napríklad nemôžete pridať funkciu, pretože čakáte, až váš spolupracovník dokončí pull request, na ktorý nadväzuje vaša práca.
        - položku môžete označiť ako blokovanú výberom možnosti **Yes**

TEST CASE

- Vytvorte manuálne testovacie prípady a skontrolujte, či každý z výstupov vyhovuje potrebám vašich používateľov.
- Usporiadajte svoje testovacie prípady tak, že ich pridáte do testovacích plánov a testovacích súborov.
- Vyberte pre danú úlohu testerov.
- Nový krok môžete pridať zvolením možnosti **Click or type here to add a step**
- Pre každý krok zadáte:
    a. ACTION
        - Presný popis akcie, ktorú má tester vykonať
    b. EXPECTED RESULT

- Presný popis výsledku, ktorý by sa mal udiať za predpokladu, že všetko funguje, ako má

Pre všetky položky, pri ktorých je táto možnosť dostupná, je potrebné priradiť nadradenú položku (parent).

3. Ako komunikovať v tíme

Na internú komunikáciu v tíme a s vedúcim tímu používame nástroj Microsoft Teams. Nástroj umožňuje rozdeliť témy komunikácie na príslušné kanály. Nástroj poskytuje dobré vyhľadávanie v histórii a taktiež integritu iných nástrojov. Nástroj nám zároveň slúži ako úložisko pre zápisky zo stretnutí a iné podporné dokumenty pre našu internú organizáciu.

Kanály

- General
- Team Chat
- Unity
- CI
- Webstránka
- Testy

Okrem vyššie definovaných kanálov využívame aj chat, ktorý slúži na neformálnu komunikáciu na rôzne témy.

E-Mailová komunikácia

Ako tím máme vytvorený e-mailový alias, ktorého správy sú presmerované do našich osobných e-mailových schránok. Prostredníctvom e-mailu komunikujeme s tretími stranami, ktoré sa zapoja do procesu vývoja projektu. Za pravidelné sledovanie a odpovedanie na prijaté správy je zodpovedný manažér komunikácie.

E-Mailová adresa: `fiit.tp.team3@gmail.com`

Pravidlá komunikácie v kanáloch

- Kanál musí mať výstižný názov
- Člen príslušného kanála v ňom komunikuje výhradne o téme, na ktorú je kanál určený

Pravidlá komunikácie na stretnutiach

- Na začiatku stretnutia si spravíme StandUp
- Stretnutie vždy niekto vedie/určuje smer komunikácie (zvyčajne toto robí scrum master)
- Nepoužívame notebooky, pokiaľ to nie je potrebné v súvislosti so stretnutím
- Neprekrikujeme sa
- Neriešime veci, ktoré nesúvisia s prácou na projekte

4. Konvencie

Riadky by mali byť maximálne 80-120 znakov dlhé, aby sa ľahšie hľadali zmeny v Gite.

4.1. Konvencie písania kódu pre C#

Konvencie pomenovania

- Konštanty – konštanty by mali byť pomenované na základe PascalCasing bez ohľadu na modifikátor prístupu.

```
private conts int TheUniversalAnswer = 42;
public conts double Pi = 3.14;
```

- Private premenné – mali by začínať podčiarkovníkom a malým písmenom

```
private int x;
public static readonly myLock = new Object();
```

- Metódy a Triedy – používať PascalCasing

```
public class ClientActivity
{
    public void ClearStatistic()
    {
        //...
    }
    public void CalculateStatistic()
    {
        //...
    }
}
```

- Argumenty metód a lokálne premenné – používať camelCasing

```
public class UserLog
{
    public void Add(LogEvent logEvent)
    {
        int itemCount = logEvent.Items.Count;
        //...
    }
}
```

- Triedy – používať podstatné mená na pomenovanie tried

```
public class Employee
{
}
public class BussinesLocation
```

```
{
}
public class DocumentCollection
{
}
```

- Rozhrania – používať prefix I

```
public interface IShape
{
}
public interface IShapeCollection
{
}
public interface IGroupable
{
}
```

- Nepoužívať bulharské konštanty
- Jednoriadkové príkazy – môžu mať zátvorky, ktoré začínajú a končia na rovnakom riadku

```
public class Foo
{
    int bar;
    public int Bar()
    {
    get { return bar; }
    set { bar = value; }
    }
}
```

- Curly braces (kučeravé zátvorky)
    - Allman style - vertikálne zarovnanie

```
if(condition)
{ //this brace should never be omitted
    DoSomething();
}
DoSomethingElse();
```

- Deklarácie
    - Vždy špecifikovať viditeľnosť (public, private, protected)
    - Viditeľnosť by mala byť prvým modifikátorom (public abstract)
    - Členy enum, by mali byť zoradené podľa hodnoty
    - Pri statických poliach kľúčové slovo readonly po static
- Referencie

        - Používať this. na rozoznanie medzi lokálnou a členskou premennou
        - Ak je možné, vždy používať var namiesto špecifických typov
        - Používať kľúčové slová (int, string, float...) namiesto BLC typove (String, Int32, Single,...), rovnako pre volania metód (int.Parse nie Int32.Parse)
    - Menné priestory
        - Import by mal byť na začiatku súboru, mimo menného priestoru, deklarácie by mali byť zoradené abecedne, ale System by mal byť pred všetkými ostatnými
    - Organizácia tried – zoradiť v nasledujúcom poradí
        - Konštanty
        - Polia
        - Vlastnosti
        - Udalosti
        - Metódy
        - Vnútorné typy
        - Rovnaké typy by mali byť ďalej zoradené podľa viditeľnosti - public, protected, private
    - Komentovanie
        - XML komentáre na dokumentovanie metód, tried a rozhraní (`///<sumary>...</summary>`)
        - Používať single line komentovanie na všeobecné komentovanie (`// ....`)
        - Komentáre umiestnovať na zvlášť riadok, nie na koniec riadku s kódom
        - Necomitovať mŕtvy kód (bez komentárov)
        - Nepoužívať bloky komentárov (`/* ... */`)
    - Layout konvencie
        - Jeden príkaz na riadok
        - Jedna deklarácia na riadok
        - Aspoň jeden voľný riadok medzi definíciami metód a definíciami vlastností

4.2 Konvencie písania kódu pre C++

Formátovanie

- Zátvorky sú stále na novom riadku

```
//správne
class MyClass
{
   //...
};
```

```
//nesprávne
class MyClass{
   //...
};
```

Pomenovanie

- Na pomenovanie používame PascalCase
- Typy - názvy začínajú veľkým písmenom

```
class MyClass: public MyParent
{
    //...
};
```

- Funkcie - názvy začínajú malým písmenom

```
void myFuncion(int a);
```

- Konštanty - celé veľkými písmenami, slová sú oddelené podčiarnikom _

```
const int ANSWER_TO_EVERYTHING = 42;
```

- Premenné - názvy začínajú malým písmenom, nepoužívame prefix _, m_ pre private/protected premenné

```
int myVariable = 42;
```

Správne programovanie

- Uprednostňujeme referencie pred ukazovateľmi
- Snažíme sa zabrániť zbytočnému kopírovaniu dátových štruktúr

```
//správne
//Poznámka: Pre návratovú hodnotu bude spravená RVO (Return Value
Optiomalization)
std::string toLowercase(const std::string& str)
{
    std::string result;
    //...
    return result;
}
//nesprávne
std::string toLowercase(const std::string str)
{
    std::string result;
    //...
    return result;
}
```

POUŽÍVAME

- `nullptr` namiesto `NULL`
- `#include guards` namiesto `#pragma once`
- forward deklarácie všade, kde je to možné
- `const` všade, kde je to možné (funkcie, premenné, ukazovatele)
- lambda funkcie namiesto function objects všade, kde je to možné

NEPOUŽÍVAME

- `using namespace` v header súboroch
- using namespace std;
- makrá, ak to nie je úplne nevyhnutné

5. Ako logovať z kódu

Aplikácie, ktoré boli nasadené do výroby, sa musia monitorovať. Jedným z najlepších spôsobov, ako monitorovať správanie aplikácií, je emitovanie, ukladanie a indexovanie logovacích dát. Logy je možné posielať do rôznych aplikácií na indexovanie, kde ich možno vyhľadať v prípade problémov.

Je potrebné, aby každý log obsahoval informácie o tom, kde v kóde chyba nastala a kedy k nej došlo.

5.1 Lua

Zdrojové kódy napísané v jazyku `Lua` využívajú na logovanie modul lualogging. Všetky moduly, v ktorých je logovanie použité umožňuje buď použitie jednej, globálnej, inštance logovacieho modulu, alebo vytvorenie vlastnej inštancie. Použitie globálnej inštancie uľahčí následne logovanie v rámci celého projektu, nakoľko je možné zjednotiť konfiguráciu.

Deklarácia

Deklaráciu je možné vidieť na ukážke zdrojového kódu.

```
local logger = _G.logger or logging.new(function(self, level, message)
                            print(level, message)
                            return true
                          end)
```

Umiestnenie

Inštancia logovacieho modulu sa spravidla nachádza v submoduloch s názvom `utils` v repozitároch jednotlivých Lua modulov.

Severity

Logovací modul umožňuje nastavenie viacerých úrovní :

- DEBUG
- INFO
- WARN
- ERROR
- FATAL

Použitie

Zalogovať udalosť je potom možné volaním funkcie v tvare `logger:severity("message")`, teda napr. `logger:error("server unreachable")`.

Moduly pokryté logovaním

Logovanie je implementované v nasledujúcich moduloch :

- `Luadb`
- `Luametrics`
- `Luameg`
- `Luagit`
- `Luatree`

6. Ako verziovať kód a pracovať s gitom

Git repozitár pozostáva z 5 typov vetiev:

- master vetva - produkčná verzia, na 100% funkčná
- develop vetva - funkčná verzia najnovšie pridávanej funkcionality, práve testovaná, je vytvorená z `master` vetvy, neskôr bude spojená s `master` vetvou
- feature - obsahuje rôznu novú funkcionalitu, na ktorej sa pracuje vrámci šprintu (šprintov), je vytvorená z `develop` vetvy, po dokončení funkcionality je spojená s `develop` vetvou
- bug - obsahuje opravu funkcionality, ktorá ešte nie je v produkčnej verzii, je vytvorená z `develop` vetvy, po dokončení opravy je spojená s `develop` vetvou
- hotfix - obsahuje opravu funkcionality, ktorá je v produkčnej verzii a je potrebné, aby bola čo najskôr opravená, je vytvorená z `master` vetvy, po dokončení opravy je spojená s `develop` aj `master` vetvou

Pre prácu s gitom postupujte podľa nasledujúcich pokynov:

1. Zosynchronizovanie všetkých vetiev s repozitárom pomocou príkazu `git fetch --prune`.

2. Vytvorenie novej vetvy z develop vetvy. Pre jednu user story sa vytvára len jedna vetva, do ktorej všetci, ktorí na danej user story participujú pridávajú zmeny. Meno developera v názve vetvy bude podľa toho, komu je daná user story priradená.
3. Keď je úloha hotová, vytvorí sa merge request podľa code review metodiky.
4. Po prehliadke kódu, zapracovaní pripomienok a následnom schválení, autor merge requestu vykoná spojenie vetvy s develop vetvou.

Pomenovanie vetiev

Pre riešenie novej funkcionality programu je potrebné vytvoriť vetvu podľa nasledovných pravidiel:

- Názov vetvy nech obsahuje typ vetvy (feature, bug, hotfix), priezvisko developera, výstižný popis toho, čo sa na vetve rieši a id položky z Azure DevOps
- Celý názov branche je napísaný v angličtine a malým písmom
- Oddeľovanie slov pomocou pomlčky (znak -)

Príklad: `git checkout -b feature/hanakova-created-repository-us-12356`

Commit správy

Pri písaní commit správ je potrebné dodržiavať nasledujúce pravidlá:

- Správy prisluchajúce commit-om sú písané v anglickom jazyku
- Správa sa začína názvom modulu alebo sekcie (v prípade dokumentácie), ktorý je upravovaný a za názvom dvojbodka
- Nasleduje krátky a výstižný popis toho čo bolo v commite spravené, tento popis môže mať maximálne 70 znakov
- Správa nie je ukončená bodkou
- Slovesá sú v správach uvádzané v tvare imperatívu
- Za krátkym popisom sa nachádza v hranatých zátvorkách T (pre task) alebo B (pre bug), pomlčka a identifikačné číslo z Azure DevOps
- V prípade, že to je potrebné nasleduje prázdny riadok a za ním obšírnejší popis a objasnenie commitu

Príklad: `git commit -m "methodology: add base structure of document [T-12578]"`

7. Ako písať testy

Metodika testovania sa týka všetkých členov tímu, pričom definuje spôsob písania a spúšťania testov. Nakoľko sa vyžaduje otestovanie každej logickej časti modulu, dôraz kladieme na testovanie hraničných prípadov. Testy sa vyvárajú programátorské, jednotkové, ale aj integračné, pričom musia pokývať aj určitú časť funkcionality.

Pravidlá písania testov

- Dodržiavať
    - dĺžku vykonania testu - aby bola čo najkratšia
    - hraničné prípady
    - otestovanie čo najviac možných scenárov danej funkcionality

Písanie testov by nemalo zabrať viac ako 40% času vývoja produktu.

Programátoské testy vykonáva iný programátor, ako ten ktorý bol autorom zdrojového kódu. Program je v tomto stupni kontrolovaný na úrovni zdrojového kódu.

Jednotkové testy by mali byť písané tak, aby overovali práve jednu funkcionalitu, akonáhle sa jednotkový test skladá z viacerých častí, ktoré sa dajú dekomponovať, je potrebné rozdeliť tieto časti do viacerých jednotkových testov.

Integračné testy budú vykonávané vo väčšine prípadov manuálne.

Údržba testov

Pri zmene logiky kódu sa môže stať, že dané testy už viac nie sú aktuálne a je potrebné ich upraviť. V takom prípade treba opäť zvážiť, či daná úprava overuje dostatočne veľa hraničných prípadov, či jeho dĺžka vykonania po zmene nie je neprimerane veľká a či testuje dostatok možných scenárov danej funkcionality.

Code coverage

Code coverage je pokrytie kódu automatizovanými testami. Je vyjadrený v percentách a je možné merať ho viacerými spôsobmi. Využívame prístup merať code coverage prostredníctvom line coverage - pomer riadkov kódu, ktoré sa vykonajú počas testovania ku počtu riadkov kódu, ktoré sa počas testovania nevykonajú. Každý modul produktu by mal mať minimálne 50% code coverage a celkové pokrytie projektu by malo mať aspoň 75% code coverage.

Zmazanie testu

Pokiaľ čas vykonania testu je neprimerane veľký, alebo čas strávený údržbou testu je príliš veľký, je potrebné tento test zmazať z dôvodu, aby nezdržoval vývoj. Písanie testov a ich údržba nesmie byť nadradená samotnému vývoju. Ak čas vykonania testu je príliš veľký, tak vývojári nemajú záujem spúšťať automatizované testy dostatočne často počas vývoja a tento prístup je nežiaduci z dôvodu neskorého odhalenia chýb.

8. Ako písať dokumentáciu

Dokumentáciu píšeme v markdowne na gitlabe v repozitári Dokumentácia

https://gitlab.com/FIIT/3DSoftVis_Remake/documentation.

Celá dokumentácia sa delí na dve časti:

1. **Dokumentácia k produktu** - zachytáva architektúru celého systému, opisuje jeho funkcionalitu a obsahuje tiež dokumentáciu zo zdrojových kódov
2. **Dokumentácia k riadeniu** - dokumentuje procesy riadenia v projekte a jednotlivé metodiky

Vygenerovaná dokumentácia sa nachádza na našej stránke

https://team03-20.studenti.fiit.stuba.sk/home.html.

Dokumentácia k produktu

1. Architektúra systému

   Táto časť dokumentácie obsahuje ucelený pohľad na systém zhora. Zachytáva jednotlivé moduly systému ich prepojenie. Opisuje tiež funkcionalitu modulov. Pre túto časť je zvlásť vhodné použitie diagramov na lepšie zachytenie architektúry systému.

   Táto časť dokumentácie by tiež mala zachytávať komunikáciu medzi jednotlivými modulmi (najlepšie formou diagramu). Túto časť je potrebné ešte len doplniť.

2. Infraštruktúra

   Časť o infraštruktúre slúži na opis všetkých podporných nástrojov pre správu a nasadenie systému. Obsahuje tri základné časti: build systému, CI/CD a testy.

   - Build systém popisuje spôsob buildovania pre moduly v projekte.
   - Časť Gitlab CI/CD obsahuje použité pipelines. Pre každý modul je potrebné popísať, aké etapy pipeline obsahuje, na čo slúžia a aké artefakty produkujú.
   - Dokumentácia k testom obsahuje základné informácie pre písanie a používanie testov. Má opisovať aké typy testov sú v projekte použité, ako ich spustiť a kedy je vhodné ich použiť. Každý test case by mal obsahovať krátky popis, na čo slúži. Tento popis sa píše priamo do zdrojových kódov.

3. Funkcionalita systému

   Táto časť sa nepozerá na systém z pohľadu architektúry ale z pohľadu funkcionality. Táto časť pritom neslúži ako príručka pre používateľa, je to skôr návod pre vývojára. Jeho úlohou je poskytnúť informáciu o nejakej črte alebo funkcionalite z rôznych pohľadov. Ako prvé je potrebné vysvetliť účel funkcionality. Je potrebné ju tiež zachytiť z pohľadu architektúry, ktoré časti sú pre jej vykonanie potrebné a ako spolu komunikujú. Je vhodné popísať aj niektoré podstatné funkcie alebo pridať odkaz na testy. Na záver by mal nasledovať popis funkcionality z pohľadu používateľa, môže obsahovať aj odkaz na príslušnú časť v používateľskej príručke.

   Predloha pre písanie dokumentácie k funkcionalite systému:

   - predloha sa nachádza v priečinku `Sablony.exclude`

```
# Názov modulu
```

```
Popis modulu (high level pohľad, na čo modul slúži, aká je jeho funkcionalita, a
pod.)

## Architektúra

Zachytáva časti systému a triedy, v ktorých je daná funkcionalita implementovaná.
Je potrebné opísať aké moduly a časti meddzi sebou komunikujú.
Môžu byť použité tiež diagramy komunikácie ak je to vhodné. Pri niektorých
funkcionalitách
je možné pridať aj stavový diagram a popísať jednotlivé stavy.

## Technická dokumentácia

Vysvetľuje použitú implementáciu. Môže obsahovať zoznam použitých tried a
funkcií.
Podstatné funkcie alebo metódy je vhodné bližšie popísať.

## Testy

Aké máme pre tento modul testy, aké je pokrytie kódu testami.
Je dobré pridať odkaz na testy.

## Používateľská príručka

Stručný popis použitia funkcionality z pohľadu používateľa.
Je vhodné použiť odkaz na príslušnú časť používateľskej príručky.
```

4. Príručky

   Pod časť dokumentácie príručky spadajú všetky návody a postupy, s ktorými sa môžeme pri vývoji alebo alebo používaní systému stretnúť. Príručky sú rozdelené na tri časti a inštalačnú, vývojársku a používateľskú.

   Každá príručka by mala obsahovať jasný a podrobný postup ako sa dostať k cieľu. Mala by tiež vysvetľovať menej zrejmé kroky, prečo ich treba vykonať. Pri písaní príručiek je dôležité pamätať na to, že čo sa teraz môže zdať jasné a samozrejmé, nemusí také byť aj v budúcnosti alebo pre niekoho menej skúseného.

5. Dokumentácia k zdrojovému kódu

   Táto časť obsahuje vygenerovanú dokumentáciu zo zdrojových kódov. Generuje sa pomocou Gitlab CI, pre každý modul a vetvu zvlášť.

   Pri komentovaní kódu je potrebné dodržať nasledujúce pravidlá.

   modul/package/namespace - popis na čo slúži - API, zoznam verejných funkcií, ktoré poskytuje aj so stručným popisom - zoznam interných funkcií aj so stručným popisom - zoznam tried, ktoré obsahuje aj so stručným popisom *(Poznámka: v jazyku Lua triedy neexistujú, my sa však v projekte tvárime, že áno. Využívame na to funcie vracajúce objekt.)*

   trieda

- popis čo robí a na čo slúži
- zoznam atribútov
- zoznam metód
- popis jednotlivých atribútov, ich účel
- popis jednotlivých metód, čo robia
    - parametre metódy + stručný opis
    - návratová hodnota metódy + stručný opis
    - vo vnútri metódy je vhodné pridať komentár ku každej ucelenej časti kódu, aby bolo možné rýchlo zistiť, čo funkcia alebo metóda interne robí

Dobre zdokumentovaná ukážka package-u v jazyku Lua.

Vhodná na inšpiráciu je tiež dokumentácia Qt, ktorú najdete tu.

Pomocné dokumenty pri písaní kódu: 1. Konvencie písania kódu 2. Ako logovať z kódu

6. Časté problémy

   Časť časté problémy obsahuje často sa vyskytujúce problémy pri vývoji systému. Tieto problémy sa môžu týkať rôznych oblastí. Podstatné je zapísať ku každému problému spôsob alebo postup jeho odstránenia. Táto časť slúži na predchádzanie riešenia tých istých problémov zas a znova.

Pri vkladaní ukážok kódu do dokumentácie je nutné, pokiaľ je to možné, vkladať ich dynamicky a to pomocou makra code_snippets, ktoré je popísané v časti MkDocs.

Pre každú novú časť dokumentácie je vždy potrebné napísať úvod, ktorý podá čitateľovi základné vysvetlenie konceptu alebo funkcionality, ktorej opis bude nasledovať.

Riadky by mali byť maximálne 80-120 znakov dlhé, aby sa ľahšie hľadali zmeny v Gite.

Užitočné tipy a triky môžete nájsť na https://www.mkdocs.org.

9. Ako vykonať code review

Pokyny pre autora

1. Na GitLab repozitári choď do *Merge Requests* v ľavom menu
2. Stlač *New merge request*
3. Nájdi svoju branchu v kolónke *Source Branch*
4. Ako `Target Branch` nastav príslušnú branchu
5. (pre `feature/` a `bugfix/` - `develop`, pre `hotfix/` - `master`)
6. Vyplň *Title* & *Description*, taktiež *Assignee* a *Approvers*
7. Do *Description* pridaj link na dokumentáciu, ktorú si vytvoril ako súčasť práce na US

8. Skontroluj, či zbehol build v karte *Build & Release*. Na schválenie merge requestu je potrebné, aby build prešiel
9. Skontroluj ostatné artefakty z CI (cpplint, doxygen,...)
10. Ak ti recenzent nájde chybu, je ju potrebné opraviť a po opravení napísať ako odpoveď ku komentáru, že chyba je opravená

Pokyny pre recenzenta

1. Ak si pridelený na merge request, choď do *Merge Requests* a nájdi názov merge requestu, na ktorý si bol pridelený
2. V karte *Changes* môžeš vidieť zmeny v zdrojovom kóde autora
3. Na chyby sa snaž upozorniť a napísať do komentárov alebo ku riadku kódu. Ak sa ti čokoľvek nezdá, tiež to napíš do komentárov alebo ku riadku kódu
4. Po opravení chyby označ komentár ako *Resolved*
5. Po vyriešení všetkých komenárov mergni kód tlačidlom *Merge*
6. Pred tým, ako definitívne mergneš kód, zvoľ `Modify merge commit` a uprav merge commit správu. Táto správa bude stručne obsahovať, čo pridáva obsah merge requestu do projektu.

Čo je potrebné kontrolovať

1. Statická analýza
    - Vymazal autor niečo, čo nemal?
    - Je kód v súlade s metodikou pre coding conventions?
    - Je kód jasne okomentovaný?
2. Dynamická analýza
    - Je funkcionalita správna?
3. Testy
    - Prešli všetky testy úspešne?
    - Je kód dostatočne pokrytý testami?

Link na rady pre autora aj recenzenta ohľadom code review

10. Metodika iterácií

Počas každého šprintu máme šesť stretnutí, dve s vedúcim tímu a štyri bez neho.

Každé stretnutie začína StandUpom. Aj keď by sa mal uskutočňovať každých 24 hod., tento princíp uskutočňujeme 3x za týždeň vzhľadom na podmienky a rozvrh predmetu. V tejto diskusii scrum master dohliada na to, aby každý člen tímu (vrátane scrum mastra) zodpovedali na otázky:

1. Na čom som pracoval za dobu od posledného stretnutia
2. Na čom sa chystám pracovať
3. Aké problémy mám v súvislosti s mojou úlohou

1. stretnutie (s vedúcim)

V prvej časti tohto stretnutia ukončujeme predchádzajúci šprint. Vyhodnotíme celkovú iteráciu, zhodnotíme burndown chart (graf zobrazujúci množstvo práce, ktoré je nutné ešte dokončiť), v ideálnom prípade je jeho hodnota na nule. Prejdeme si s vedúcim tímu jednotlivé user stories, ktoré nám ohodnotí story pointami (vyhodnocuje, či sme splnili definition of done pre jednotlivé user stories). Po akceptovaní všetkých user stories vedúcim tímu môžeme iteráciu uzavrieť. V druhej časti stretnutia plánujeme nasledujúci šprint. Spoločne špecifikujeme user stories, ktoré sú súčasťou šprint backlogu a počas šprintu tieto user stories zásadne nemeníme. Pred zaraďovaním user stories do backlogu vyhodnocujeme jednotlivo každú user story príslušným bodovým ohodnotením (story points), ktoré určujú zložitosť a relatívnu časovú náročnosť konkrétnej úlohy. Každú user story si rozdelíme na tasky. V prípade, že toto nestihneme na stretnutí, **každý je povinný si rozdeliť svoje user stories, na ktorých bude pracovať na tasky, ešte v ten deň**.

2. stretnutie (bez vedúceho)

Počas tohto stretnutia rozdeľujeme úlohy medzi členov tímu, pracujeme na priradených taskoch a konzultujeme prípadné problémy.

3. stretnutie (bez vedúceho)

Počas tohto stretnutia ohodnocujeme tasky, pracujeme na priradených taskoch a konzultujeme prípadné problémy.

4. stretnutie (s vedúcim)

Na tomto stretnutí sa zameriavame na riešenie vzniknutých problémov počas prvej časti šprintu. V tejto fáze šprintu by už mali byť všetky user stories aspoň začaté. Venujeme sa tiež úprave backlogu. Vymýšľame nové user stories, ohodnocujeme ich a prioritizujeme.

5. stretnutie (bez vedúceho)

Počas tohto stretnutia pracujeme na priradených taskoch a konzultujeme prípadné problémy.

6. stretnutie (bez vedúceho)

Na poslednom stretnutí v iterácii vytvárame pod dohľadom scrum mastera retrospektívu, na ktorej zapíše každý člen tímu všetky pozitíva, negatíva a nápady na vylepšenie za uplynulý šprint. Po zapísaní týchto informácii do dokumentu je uverejnený na tímovej web stránke. Počas tohto stretnutia si tiež vyhodnotíme percentuálny podiel práce na jednotlivých user stories za uplynulý šprint. Ďalej pokračujeme samostatnou prácou na priradených taskoch.

11. Metodika riadenia úloh

Na riadenie úloh používame nástroj Azure DevOps.

ŽIVOTNÝ PROCES USER STORY

1. Backlog

   Po identifikovaní sa user story pridá do backlogu a nachádza sa v stave *New*. User stories, ktoré sa nachádzajú v backlogu nemusia mať kompletný alebo vôbec nejaký opis, nie sú rozdelené na tasky ani nie je ohodnotená ich zložitosť. User stories, ktoré sa nachádzajú v backlogu nám slúžia hlavne ako zdroj nápadov a funkcionality, ktorú by sme chceli niekedy v projekte mať.

2. Ready to plan

   User stories, ktoré majú pridaný opis, sa presúvajú do stavu *Ready to plan*. Nad user stories, ktoré sa nachádzajú v tomto stave, sa môžeme hlbšie zamýšľať, rozdeľovať ich na tasky a ohodnocovať ich zložitosť.

3. Plan done

   V prípade, že už má user story pridaný opis, je rozdelená na jednotlivé tasky a je odhadnutá jej zložitosť, môžeme ju presunúť do stavu *Planned*. Znamená to, že user story je pripravená na to, aby mohla byť vybraná do šprintu.

4. Ready to develop

   User stories, ktoré boli vybrané do šprintu, sa presúvajú do stavu *Ready to develop*. Ako tím sa zaväzujeme, že tieto stories budú do konca šprintu v stave *To accept*. Každá user story v tomto stave musí byť niekomu priradená. Ten, komu je user story priradená, je potom zodpovedný za jej dokončenie. To znamená, že dbá na to, aby ľudia participujúci na tejto user story, začali na nej včas pracovať, prípadne pomáha riešiť vzniknuté problémy.

5. Develop doing

   Akonáhle sa na user story začne pracovať (aspoň jeden z jej taskov je v stave *In Progress*), presúva sa do stavu *In development*.

6. Ready to review

   Po dokončení všetkých taskov prislúchajúcich k danej user story sa táto story presúva do stavu *Ready to review*. Akonáhle je user story v tomto stave, je potrebné, aby niekto skontroloval prácu, ktorá sa na tejto story vykonala. To sa môže udiať dvomi spôsobmi. Prvý spôsob je že ten, kto je zodpovedný za danú user story vytvorí task na review a priradí ho niekomu na kontrolu. Druhá možnosť je, že user story nie je nikomu priradená na kontrolu. V takomto prípade bude robiť review ten, kto má aktuálne čas alebo schopnosti na review. Reviewer si sám vytvorí task na review a priradí si ho.

7. In review

   Akonáhle začne reviewer pracovať na kontrole user story, presúva sa do stavu *In review*. V tomto stave zostáva do momentu, kým reviewer nedokončí kontrolu. To znamená, že všetky tasky sú dokončené a spĺňajú definition of done. Zároveň sú splené akceptačné

kritériá danej user story a všetky zapracované zmeny sú mergnuté v develop vetve príslušného repozitára. V prípade, že toto všetko je splnené, presúva sa user story do stavu *To accept*. V opačnom prípade sa user story presúva naspäť do stavu *In development* a je potrebné opraviť nájdené nedostatky.

8. To accept

   User stories, ktoré prešli review sa nachádzajú v stave *To accept*. V tomto stave zostávajú až do momentu ich akceptovania alebo neakceptovania product ownerom. V prípade neakceptovania sa user story presúva do stavu *In development* a je potrebné zapracovať všetky nedostatky.

9. Done

   V prípade akceptovania product ownerom sa user story presúva do stavu *Done*. Tento stav je konečný.

Životný proces taskov v šprinte

1. To Do

   Všetky novo vytvorené tasky pre nejakú user story sa najprv nachádzajú v stave *To Do*. Každý task v tomto stave je niekomu priradený, má napísaný opis a odhadovanú časovú zložitosť v hodinách.

2. In Progress

   Po začatí práce na nejakom tasku sa tento task presúva do stavu *In progress*. To, že sa task nachádza v tomto stave značí, že rozpracovaný. Na takýchto taskoch môžeme a mali by sme upravovať zostávajúci čas dokončenia tasku. Tento čas sa môže zvyšovať alebo znižovať podľa toho, či práca na tasku ubúda, alebo sme narazili na nejaký problém a práca sa teda predĺži.

3. Done

   Po dokončení tasku sa presúva do stavu *Done*.

12. Metodika záznamov zo stretnutí

Pred samotným stretnutím si do časti Agenda zapíšeme všetko, čo by sme chceli na danom stretnutí prekonzultovať, či už sú to rôzne problémy, otázky alebo iné záležitosti súvisiace s našim projektom. Účasť je doplnená na začiatku stretnutia v závislosti od prítomnosti jednotlivých členov. Stand-up zachytáva naše monológy, kedy sa snažíme interpretovať progres v našej práci, či prípadné problémy, na ktoré sme počas práce narazili. Po kole stand-upov sa venujeme agende, teda záležitostiam, ktoré je potrebné vyriešiť na stretnutí. Následne si definujeme nadväzujúce úlohy v našom projekte a tie sa snažíme vyriešiť do ďalšieho stretnutia. V závere stretnutia v krátkosti zhrnieme, na čom sa každý z nás chystá pracovať.

Na dokumentovanie priebehu našich stretnutí sa nám najviac osvedčilo používanie jednotnej šablóny, do ktorej si zapisujeme potrebné informácie. Táto šablóna pozostáva z nasledujúcich častí:

- **Účasť**
  je doplnená na začiatku stretnutia v závislosti od prítomnosti jednotlivých členov
- **Agenda**
  do časti Agenda zapisujeme všetko, čo by sme chceli na danom stretnutí prekonzultovať, či už sú to rôzne problémy, otázky alebo iné záležitosti súvisiace s našim projektom
- **Stand-up**
  zachytáva naše monológy, kedy sa snažíme interpretovať progres v našej práci, či prípadné problémy, na ktoré sme počas práce narazili
- **Úlohy do ďalšieho stretnutia**
  po vyriešení agendy si definujeme nadväzujúce úlohy v našom projekte a tie sa snažíme vyriešiť do ďalšieho stretnutia
- **Zhrnutie priebehu stretnutia**
  v závere stretnutia v krátkosti zhrnieme, na čom sa každý z nás chystá pracovať

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

Inžinierske dielo
Tímový projekt VizReal

Predmet:	Tímový projekt I.
Členovia tímu:	Bc. Nikodém Adler
	Bc. Ivana Frankovičová
	Bc. Andrej Hoferica
	Bc. Michal Jozefek
	Bc. Michael Kročka
	Bc. Samuel Šouc
Vedúci práce:	Ing. Peter Kapec, PhD.
Akademický rok:	2020/2021

Obsah

Úvod

Keď si predstavíme projekt, ktorý je napísaný v programovacom jazyku, vybavia sa nám funkcie, premenné a závislosti týchto funkcií. Práve tieto dve veci, teda funkcie a závislosti sa dajú zobrazovať graficky vo forme grafov, kde funkcie predstavujú vrcholy a ich volania zase hrany. Náš tím sa venujeme zobrazovaniu práve takýchto grafov pomocou virtuálnej alebo rozšírenej reality. Vizualizovanie týchto grafov nám dokáže pomáhať pri analýze kódu, chápaní závislosti jednotlivých modulov alebo na prezentačné účely.

Cieľom nášho projektu je teda poskytnúť používateľovi väčší prehľad v napísanom kóde, či už je to vo virtuálnej realite kde kód predstavuje graf, alebo v rozšírenej realite kde je kód zobrazený ako metafora mesta.

Globálne ciele projektu

Zimný semester

Projekt VizReal je veľký, komplexný projekt, ktorý sa vyvíja už niekoľko rokov, preto bolo počas zimného semestra zo začiatku naším cieľom najprv sa s týmto projektom oboznámiť a analyzovať jeho časti.

V súčasnom stave je projekt postavený na vrstvovej architektúre. Hlavná logická vrstva je implementovaná v prostredi Lua a hlavná zobrazovacia vrstva je implementovaná pomocou prostredia Unity engine. Tieto vrstvy komunikujú medzi sebou prostredníctvom viacerých vrstiev, ktoré slúžia len na ich komunikáciu. Tie sú implementované v C++ a v C#.

Chceli by sme odstrániť C++ vrstvu a navrhnúť a implementovať nové komunikačné rozhranie, pomocou ktorého budú tieto vrstvy komunikovať. Pripraviť sa na takéto prerábanie architektúry systému a samotné odstránenie vrstvy je ale väčším cieľom na zimný semester, ktoré pravdepodobne nestihneme kompletne splniť a bude sa musieť časť z toho presunúť do cieľov letného semestra.

Architektúra systému

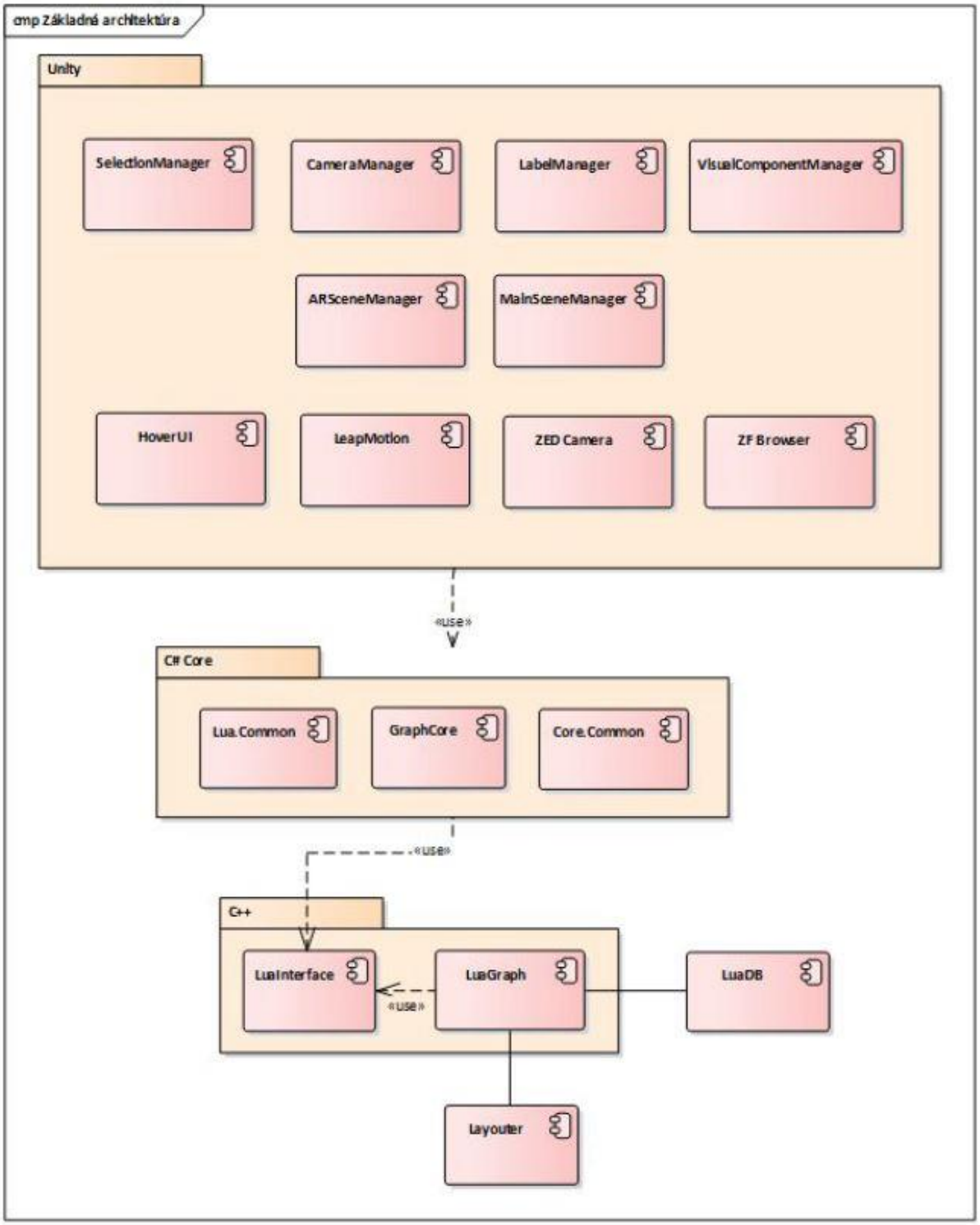
Celkový pohľad na systém

Kapitola obsahuje opis základnej architektúry projektu, jednotlivé vrstvy a ich moduly. Všetky moduly majú vybudovanú infraštruktúru, ktorá zahŕňa kompiláciu, testovanie, coverage, generovanie dokumentácie zo zdrojového kódu a jej export na vzdialený server.

Základná architektúra

Architektúra projektu je zložená zo 4 vrstiev.
Na najnižšej vrstve sa nachádzajú moduly jazyka Lua. Tieto moduly priamo narábajú s reprezentáciou grafových údajov a vykonávajú nad nimi operácie.
Druhá vrstva je reprezentovaná programovacím jazykom C++. Na tejto vrstve sú implementované 2 moduly, ktoré slúžia ako rozhranie pre moduly jazyka C# pre prácu s najnižšou vrstvou, t.j. modulmi jazyka Lua.
Tretia vrstva je tvorená modulmi jazyka C#, ktoré poskytujú rozhranie pre Unity vrstvu, prostredníctvom ktorej je možné pracovať s nižšími vrstvami.
Najvyššia vrstva je reprezentovaná Unity, ktoré poskytuje grafickú reprezentáciu nižších vrstiev.

Architektúru projektu je možné vidieť na obrázku nižšie.



Lua

Na tejto vrstve sa nachádza reprezentácia grafových údajov. Jednotlivé operácie nad grafom sú teda vykonávané na tejto vrstve. Vo všeobecnosti teda dochádza k prepočítavaniu a zmenám súradníc uzlov, analýze zdrojového kódu a počítaniu metrík. Bližšie informácie o generovaní grafu je možné nájsť v dokumentácii generovania grafu.
S touto vrstvou úzko súvisí tiež jazyk Terra, v ktorom je implementovaný layoutovací algoritmus Fruchterman-Reingold. Na tejto vrstve je tiež možné analyzovať Moonscript projekty. Vrstva ďalej poskytuje funkcionalitu pre generovanie UML a sekvenčných

diagramov.
Vrstva je celkovo tvorená 5 modulmi.

Luadb

Logika modulu Luadb priamo súvisí s tvorbou grafu. Obsahuje reprezentáciu hrán, uzlov a metauzlov. V rámci modulu sú tiež implementované operácie, ktoré je možné nad grafom vykonávať. Bližší popis modulu a layoutovacieho algoritmu je možné nájsť v dokumentácii Lua modulov.
Modul má vybudovanú infraštruktúru a vlastný build systém.

Luametrics

Modul Luametrics vykonáva analýzu zdrojového kódu a produkuje AST, spolu s metrikami.
Modul má vybudovanú infraštruktúru a vlastný build systém.

Luameg

Úlohou modulu je vytváranie UML a sekvenčných diagramov zo zdrojových kódov. Modul podporuje jazyky Lua a Moonscript. Bližšie informácie k jednotlivým diagramom je možné nájsť v dokumentácii UML diagramov.
Modul má vybudovanú infraštruktúru a vlastný build systém.

Luatree

Luatree je balík nástrojov na analýzu a inšpekciu AST, ktoré produkuje modul Luametrics, a grafov volaní funkcií vytváraných modulom Luadb.
Modul má vybudovanú infraštruktúru a vlastný build systém.

Luagit

Modul Luagit slúži na analýzu Git repozitárov, pričom k svojej činnosti využíva moduly Luametrics a Luadb.
Modul má vybudovanú infraštruktúru a vlastný build systém.

C++

Druhá vrstva, reprezentovaná programovacím jazykom C++, slúži ako rozhranie pre C# moduly, aby mohli komunikovať s Lua modulmi a získavať tak grafové údaje, prípadne nad týmito údajmi vykonávať operácie. Vrstva zahŕňa 2 moduly. Bližšie informácie o týchto moduloch je možné nájsť v dokumentácii C++ modulov.

LuaInterface

Modul slúži ako rozhranie medzi C# a Lua modulmi. Poskytuje metódy pre volanie Lua funkcií a získavanie informácií.
Modul má vybudovanú infraštruktúru a vlastný build systém.

LuaGraph

Úlohou modulu je načítať grafové údaje modulu Luadb a poskytovať ich vyšším vrstvám.
Modul má vybudovanú infraštruktúru a vlastný build systém.

C#

Programovací jazyk C# predstavuje pomyselnú tretiu vrstvu. C# moduly sú priamo využívané vyššou vrstvou. Tieto moduly zabezpečujú prístup k metódam C++ modulov a tiež umožňujú pracovať s grafovými údajmi. Na tejto vrstve obsahuje projekt 2 moduly. Bližší popis modulov je možné nájsť v dokumentácii C# modulov.
Infraštruktúra a build systém sú združené pre všetky C# moduly.

Lua.Common

Modul poskytuje prístup k C++ metódam, ktoré sú exportované vo forme DLL `LuaGraph` a `LuaInterface`. Modul obsahuje viaceré pomocné metódy, ktoré zjednodušujú reprezentáciu údajov, prípadne vykonávajú konverziu. Modul ďalej obsahuje viacero atribútov, ktoré zjednodušujú načítavanie polí, štruktúr a reťazcov.

GraphCore

V module je obsiahnutá logika súvisiaca s grafovými údajmi. Modul umožňuje prácu s grafom, načítavanie jeho aktuálneho stavu, polohy, prípadne iných špecifických vlastností.

Unity

Unity predstavuje najvyššiu vrstvu, ktorá slúži na zobrazovanie a manipuláciu grafových údajov prostredníctvom rozhraní nižších vrstiev. Údaje nižších vrstiev graficky reprezentuje. Logika práce Unity s grafovými údajmi je popísaná v dokumentácii generovania grafu.
Vrstva zahŕňa viacero modulov. Bližší popis jednotlivých modulov je možné nájsť v dokumentácii Unity modulov.

Možnosti komunikácie medzi procesmi

Kvôli problémom s kompabilitou medzi platformami je vhodné, ak ďalší vývoj projektu bude smerovať ku komunikácii medzi procesmi prostredníctvom IPC alebo RPC.
Cieľom je tak úplne izolovať časť, ktorá pre svoju činnosť vyžaduje programovacie jazyky Lua a Terra. Táto časť bude kontajnerizovaná. Kontajner bude obsahovať interpreter LuaJIT z distribúcie Luapower. Kontajner bude ďalej obsahovať všetky závislosti a Lua moduly potrebné k činnosti. Potrebné je tak zabezpečiť komunikáciu medzi kontajnerom a hostiteľským systémom. Taktiež je nevyhnutné navrhnúť spôsob serializácie údajov.

Do budúcna je možné túto komunikáciu riešiť buď na úrovni komunikácie v rámci lokálneho stroja, alebo je možné upraviť projekt tak, aby využíval vzdialený server. Z hľadiska povahy tejto komunikácie je potrebné zabezpečiť vysokú priepustnosť.

Kontajner by bol založený na Lua CI Image, ktorý je použitý taktiež vo vývojovom prostredí Devenv. V kontajneri by teda bežal Layouter, ktorý by prepočítaval súradnice. Tie by potom poskytoval C# modulu GraphCore. Týmto spôsobom je možné zároveň projekt odľahčiť, nakoľko by nebolo potrebné využívať C++ moduly, ktoré sprostredkúvajú komunikáciu medzi C# a Lua vrstvou.

Komunikácia

Komunikáciu medzi kontajnerom a hostiteľským systémom je v princípe možné vykonať buď medziprocesovou komunikáciou (IPC), alebo vzdialeným volaním procedúr (RPC). Zároveň je potrebné brať do úvahy kompatibilitu medzi platformami. Nakoľko kontajner využíva operačný systém Linux Ubuntu, nie je možné zabezpečiť kompatibilitu so všetkými paradigmami.

Z hľadiska rýchlosti sú jednotlivé paradigmy zoradené zostupne od najrýchlejšej po najpomalšiu :

- Zdieľaná pamäť (shared memory/shm)
- Message Queue
- Unix domain sockets
- Pipe
- FIFO (named pipe)
- TCP socket

Dôležité je poznamenať, že rozdiel medzi FIFO a TCP socketmi predstavuje približne 16%.

Ako bolo vyššie uvedené, kvôli kompatibilite nie je možné použiť všetky paradigmy. Z hľadiska kompatibility je najvýhodnejšie použitie TCP socketu. Prostredníctvom TCP socketu je následne možné prejsť na použitie vzdialeného servera bez potreby modifikácie klientskej časti.

Na zabezpečenie komunikácie existuje viacero knižníc, ktoré sa líšia rýchlosťou, podporovanými vzormi komunikácie, prípadne použitím brokerov (centrálnych uzlov). Pre potreby projektu je nevyhnutné, aby bola dostupná knižnica pre programovacie jazyky Lua a C#.

ZeroMQ

ZeroMQ (tiež známe ako ØMQ, 0MQ, alebo zmq) predstavuje rámec, ktorý umožňuje prenášať správy medzi viacerými uzlami. Podporuje viaceré spôsoby prenosu - v rámci procesu, medzi procesmi, TCP a multicast.

ZeroMQ ďalej implementuje mechanizmus High Water Mark, ktorý predstavuje pevný limit maximálneho počtu správ v rade, po naplnení ktorého dôjde k blokovaniu alebo zahadzovaniu ďalších prijatých správ.

Výhodou riešenia je kvalitná dokumentácia. K dispozícii je ako verzia pre jazyk C#, tak aj pre jazyk Lua. Lua-zmq predstavuje binding nad ZeroMQ verzie 2. Knižnica Lzmq je binding nad verziami 3 a 4.
Obe knižnice je možné inštalovať prostredníctvom správcu Luarocks.

V spojení s interpreterom LuaJIT by mal rámec poskytovať dostatočnú rýchlosť aj pri komunikácii prostredníctvom TCP socketu. Podľa dostupných informácii je možné dosiahnuť rýchlosť až 1 478.619 Mb/s.

```
Throughput benchmark using the tcp transport over localhost:
message size: 30 [B]
message count: 100000000

Using send/recv functions running under Lua 5.1.4:
mean throughput: 1577407 [msg/s]
mean throughput: 378.578 [Mb/s]

Using send/recv functions running under LuaJIT2 (git HEAD):
mean throughput: 5112158 [msg/s]
mean throughput: 1226.918 [Mb/s]

Using send_msg/recv_msg functions running under LuaJIT2 (git HEAD):
mean throughput: 6160911 [msg/s]
mean throughput: 1478.619 [Mb/s]

C++ code:
mean throughput: 6241452 [msg/s]
mean throughput: 1497.948 [Mb/s]
```

V repozitároch Lzmq-guide a Zguide je možné nájsť viacero ukážkových implementácií pre rôzne programovacie jazyky.

Nižšie je možné vidieť ukážkové riešenie jednoduchej klient-server komunikácie prostredníctvom Unix domain socketov. Server bol vytvorený v jazyku C a klient v jazyku Lua.

```
//  C Hello World server

#include <zmq.h>
#include <stdio.h>
#include <unistd.h>
#include <string.h>
#include <assert.h>

int main (void)
{
    //  Socket to talk to clients
    void *context = zmq_ctx_new ();
    void *responder = zmq_socket (context, ZMQ_REP);
    int rc = zmq_bind(responder, "ipc:///src/test");
    assert(rc == 0);

    while (1) {
        char buffer [10];
        zmq_recv (responder, buffer, 10, 0);
        printf ("Received Hello\n");
        sleep (1);          //  Do some 'work'
        zmq_send (responder, "World", 5, 0);
    }
    return 0;
}
--
--  Lua Hello World client
--  Sends "Hello" to server, expects "World" back
--
```

```
--  Author: Robert G. Jakabosky <bobby@sharedrealm.com>
--
local zmq = require "lzmq"

local context = zmq.init(1)

--  Socket to talk to server
print("Connecting to hello world server…")
local socket = context:socket(zmq.REQ)
socket:connect("ipc:///src/test")

for n=1,10 do
   print("Sending Hello " .. n .. " …")
   socket:send("Hello")

   local reply = socket:recv()
   print("Received World " ..  n .. " [" .. reply .. "]")
end
socket:close()
context:term()
```

ZeroMQ poskytuje tiež viacero vzorov komunikácie :

Request-reply

Prepojenie množiny klientov s množinou služieb. V tomto prípade sa jedná o RPC a model distribúcie úloh.

Publish-subscribe

Vzor prepája množinu producentov s množinou konzumentov. Ide o model distribúcie údajov.

Pipeline

Potrubie, ktoré spája viacero uzlov. Môže mať viacero krokov a slučiek. Jedná sa o model paralelnej distribúcie a zberu.

Exclusive pair

Vzor prepája práve 2 sockety. Ide o model prepojenia dvoch vláken v procese.

Serializácia

Pred samotným odoslaním je potrebná serializácia údajov. Čím efektívnejšia bude serializácia, tým efektívnejšie môže prebiehať celá komunikácia, obzvlášť pokiaľ by výsledné riešenie komunikovalo so vzdialeným serverom.
Medzi najbežnejšie formáty údajov patrí JSON a XML. Existuje viacero knižníc pre jazyky C# aj Lua.

JSON

JSON predstavuje jednoduchý formát na výmenu údajov. Patrí medzi pomerne ľahko čitateľné a zapisovateľné notácie. Má taktiež širokú podporu. Je založený na 2 základných štruktúrach, a to kolekcia párov kľúč-hodnota (objekt, záznam, štruktúra, slovník a pod.) a usporiadaný zoznam hodnôt (pole, vektor, zoznam a pod.).
Pre programovací jazyk Lua je možné použiť knižnicu priamo z distribúcie Luapower, konkrétne CJSON.

MessagePack

MessagePack je rámec na serializáciu údajov. Podobne ako JSON, umožňuje výmenu údajov medzi viacerými uzlami a programovacími jazykmi. Oproti formátu JSON by mal byť MessagePack efektívnejší a rýchlejší, nakoľko serializované údaje majú menšiu veľkosť. Menšia veľkosť so sebou prináša aj niekoľko obmedzení, ktoré sa týkajú možnej veľkosti údajových typov. Jednotlivé obmedzenia, rovnako ako aj podporované údajové typy je možné nájsť v špecifikácii.
Pre programovací jazyk Lua je k dispozícii knižnica lua-MessagePack, ktorá by mala byť podľa autora rýchla v kombinácii s interpreterom LuaJIT. Rovnako aj túto knižnicu je možné inštalovať prostredníctvom Luarocks.
Pre C# je potom k dispozícii knižnica msgpack-cli.

Ukážka RPC

Repozitár 3DSoftviz obsahuje adresár examples, v ktorom sa nachádza ukážka možností implementácie mechanizmov RPC s použitím ZeroMQ a MessagePack.
Za týmto účelom bol modifikovaný Lua CI Image, ktorý obsahuje všetky potrebné závislosti. Ukážka obsahuje jednoduchú implementáciu serverovej časti v jazyku Lua a klientskej časti v jazyku C#.
Serverová časť počúva na porte TCP/49155. Prijatá správa je následne rozbalená prostredníctvom knižnice MessagePack. Rozbalená správa má potom formu tabuľky, ku ktorej obsahu je možné pristupovať prostredníctvom číselných indexov. V ukážke sa očakáva správa v nasledujúcom tvare.

```
local message = {}
message[1] = "command"
message[2] = "params"
message[3] = "result"
```

Všetky atribúty tabuľky sú reprezentované vo forme reťazcov.
Jednotlivé lokálne funkcie sú uložené v tabuľke s názvom _L, čím je možné zabezpečiť volanie funkcie na základe jej názvu vo forme reťazca. Rovnakým spôsobom je možné uchovávať tiež lokálne premenné.

```
local _L = {}
_L["extract"] = extract
```

Pre účely ukážky bola vytvorená funkcia s názvom extract, ktorá spracováva parametre a následne volá funkciu extractor.extract, ktorej výstup je odovzdaný vo forme návratovej hodnoty.

```
local function extract(params)
  local path = params[1]
  return extractor.extract(path, astMan)
end
```

Funkcia je potom volaná bezpečným spôsobom prostredníctvom funkcie pcall. Funkcií sú tiež odovzdané požadované parametre.

```
local command = message[1]
local params = message[2]
local status, result = pcall(_L[command], params)
```

Pokiaľ atribút command obsahuje reťazec "end", dôjde k nastaveniu premennej cyklu na hodnotu false, server upovedomí klienta a program skončí.

```
if command == "end" then
   loop = false
   request[3] = "Server stopping"
   socket:send(mp.pack(request))
   break
end
```

Do atribútu result môže byť vložený výsledok. Pre účely ukážky je atribút reprezentovaný vo forme reťazca. Do budúcna by však bolo vhodnejšie tento atribút reprezentovať na klientskej strane vnoreným objektom, takže na strane servera by mal atribút formu vnorenej tabuľky. Odpoveď je následne odoslaná klientovi.

```
message[3] = json.encode(result)
socket:send(mp.pack(message))
```

Klientska časť je implementovaná v programovacom jazyku C#. Na úspešnú komunikáciu medzi klientskou a serverovou časťou s využitím serializácie prostredníctvom knižnice MessagePack, je potrebné vytvoriť objektové štruktúry, ktoré budú dodržané ako na klientskej, tak aj serverovej strane. Dodržanie štruktúry sa týka hlavne použitých údajových typov. Štruktúra objektu, ktorý je mapovaný na tabuľku uvedenú vyššie je možné vidieť na nasledujúcej ukážke.

```
public class RemoteCall
{
   public string functionName { get; set; }
   public string[] functionParams { get; set; }
   public string result { get; set; }
}
```

Atribút functionName zodpovedá atribútu označenému
ako command, functionParams zodpovedá params a result je mapovaný na result.
Takýmto spôsobom je možné zabezpečiť bezproblémové mapovanie C# objektov na tabuľku v jazyku Lua. Rovnako je bezproblémové spätné mapovanie tabuľky na C# objekt, ku ktorého atribútom je možné pristupovať štandardnou bodkovou notáciou.

```
MessagePackSerializer serializer = MessagePackSerializer.Get(rpc.GetType());
MemoryStream toUnpack = null;

byte[] responseFromServer = client.ReceiveFrameBytes();
toUnpack = new MemoryStream(responseFromServer);
RemoteCall unpacked = (RemoteCall)serializer.Unpack(toUnpack);
Console.WriteLine(unpacked.result);
```

Spustenie

Na spustenie ukážky je potrebné prostredie Docker a IDE Visual Studio.
Pre spustenie serverovej časti ukážky je potrebná inštalácia vývojového prostredia Devenv. Je teda potrebné postupovať podľa krokov uvedených v dokumentácii.
Repozitár obsahujúci súbory potrebné pre vývojové prostredie devenv by mali byť naklonované do rovnomenného adresára, t.j. devenv. Tento adresár po úspešnej inštalácii obsahuje ďalej adresár luadev. Do tohto adresára je potrebné naklonovať repozitár 3DSoftVis_Remake. Pre spustenie serverovej časti je potrebné prejsť do adresára examples/rpc a následne príkazom lua server.lua program spustiť.

- **Tip**

- Na spustenie nie je potrebné klonovanie celého repozitára. Pokiaľ už repozitár bol naklonovaný, stačí nakopírovať adresár examples do adresára luadev.

Na spustenie klientskej časti je potrebné otvoriť súbor examples/rpc/Lua_rpc/Lua_rpc.sln. Solution je potrebné následne v IDE Visual Studio zostaviť a spustiť.

Poznámka

Ako testovací vstup je v súčasnosti pre účely ukážky použitý Moonscript projekt. Použitie Lua projektu momentálne nie je možné, pravdepodobne kvôli vzniku cyklov vo výslednom grafe, v dôsledku čoho nie je možné nad týmito údajmi vykonať serializáciu ani prostredníctvom knižnice ZeroMQ, ani CJSON. Po odstránení cyklov by so serializáciou nemal byť problém.
Tento problém sa vyskytuje pri priamom volaní funkcie extractor.extract, ktorá je súčasťou submodulu luadb.extraction.extractor.

Poznámky

K dipozícii je viacero vývojových rámcov umožňujúcich komunikáciu medzi viacerými uzlami. Jedným z riešení môže byť gRPC, ktoré však nemá k dispozícii oficiálnu knižnicu pre jazyk Lua.

Ďalšou alternatívou môže byť komunikácia prostredníctvom JSON-RPC. Údaje by sa prenášali vo formáte JSON. Nižšie je možné vidieť ukážkové volania.

```
{"method":"doString","params":["asd = {ma=12,mk=45}"], "id":12}
{"method":"doString","params":["asd = \"asd\\nxncbv\""], "id":13}
{"method":"getObject","params":["asd"], "id":14}


{"method":"doString","params":["aasd()"]}
{"method":"doString","params":["gr = getGraph(0)"]}
{"method":"getObject","params":["gr"]}


{"method":"doString","params":["aasd()"]}
{"method":"doString","params":["UMLVALUE = getSequenceDiagram(0, 68)"]}
{"method":"getString","params":["UMLVALUE"]}

{"method":"doString","params":["print(\"asd\\nxncbv\")"], "id":13}
```

Viacero užitočných informácií, vrátane implementácie
funkcií doString, getString a getObject, rovnako aj použitie exportovaných funkcií, je možné nájsť v poznámkach v OneNote.

Užitočné linky

- https://github.com/luapower
- https://luarocks.org/modules/neopallium/lua-zmq
- https://luarocks.org/modules/moteus/lzmq
- http://zguide.zeromq.org/lua:all
- https://github.com/zeromq/libzmq

- https://github.com/goldsborough/ipc-bench
- https://github.com/grpc
- https://docs.docker.com/engine/reference/run/#ipc-settings---ipc
- https://github.com/msgpack
- https://luarocks.org/modules/fperrad/lua-messagepack

Lua

Layouter

Tento modul zodpovedá za výpočet rozloženia grafu a správu layoutov grafu.

Layout Manager

Zodpovedá za spravovanie layoutov grafu. Každý graf má svoj vlastný layout manager. Obsahuje základne funkcie pre kontrolu layoutovania ako sú štart, stop, pauza. Zabezpečuje to, ktorý algoritmus je v súčasnosti aktívny a zabezpečuje prenos dát z a do algoritmu.

Vo funkcii `updateNodes` dochádza k aktualizovaniu uzlov LuaDB grafu z práve aktívneho algoritmu. Z algoritmu sa načíta pozícia každého uzla, a prípadne ďalšie atribúty uzla, ktoré algoritmus môže obsahovať, ako napríklad veľkosť uzla, a táto pozícia sa nastaví zodpovedajúcemu uzlu v LuaDB grafe.

Algoritmus Fruchterman-Reingold

Jeho implementácia je v jazyku terra, ktorého rýchlosť je porovnateľná s jazykom C. Algoritmus si na začiatku vytvorí kópiu LuaDB grafov. Z lua tabuliek obsahujúcich uzly a hrany si vytvorí pole terra štruktúr `TNode` a `TEdge`, ktoré reprezentujú graf a obsahujú všetky atribúty, ktoré sú potrebné pre výpočet rozloženia grafu. Vo funkcii `initialize`, sa vypočíta počiatočné rozloženie grafu. Po zavolaní funkcie `runLayouting` sa vytvorí nové vlákno, kde iteratívne prebieha výpočet nového rozloženia grafu.

LuaDB

Modul je napísaný v programovacom jazyku Lua. Jeho hlavnou úlohou je analýza zdrojového kódu, ktorý je modulu poskytnutý zatiaľ zadaním priamej cesty, analyzuje volania funkcií, komplexitu, a rôzne metriky, na základe ktorých sa neskôr ostatné moduly rozhodujú o vlastnostiach vykreslenia uzlu, napríklad veľkosť uzlu.

C++ Core

LuaInterface

Tento modul zabezpečuje komunikáciu C# časti s Lua časťou využitím `Sol2` knižnice. Je naprogramovaný v jazyku C++. Poskytuje základné funkcie pre vykonávanie Lua kódu, volanie Lua funkcii, a získavanie informácií z modulov v jazyku Lua. Každá funkcia, ktorá má byť prístupná v `Lua.Common` module by mala mať svoj ekvivalent v príslušnom `ExportC` súbore. Napríklad pre `LuaInterface.cpp` je to `LuaInterfaceExportC.cpp`. Tieto funkcie sa musia nachádzať v extern "c" bloku a musia mať definíciu kompatibilnú s jazykom C.

LuaGraph

Úlohou tohto modulu je s využitím modulu `LuaInterface` načítať LuaDB graf z jazyka Lua do jazyka C++. Jeho hlavná logika sa nachádza vo funkcii `LuaGraph::loadGraph(int id)`. Podobne ako `LuaInterface`, každá pridaná funkcia, ktorá má byť prístupná v `Lua.Common` by mala byť definovaná aj v príslušnom `ExportC` súbore.

C# Core

Lua.Common

Tento modul využíva `PInvoke` pre prístup k funkciám exportovaným v C++ DLL `LuaGraph` a `LuaInterface`. Jeho jedinou úlohou je zjednodušiť prístup k týmto funkciám. Hlavnými triedami sú `LuaInterface` a `LuaGraph`, ktoré volajú príslušné C++ funkcie a v prípade potreby vykonávajú konverziu dát na príslušné dátové typy v jazyku C#. Pre zjednodušenie načítania poľa, štruktúr a stringov bola vytvorená trieda `MarshallingUtils`, ktorá obsahuje pomocné metódy na prevod týchto dátových typov z jazyka C do C#.

Pre reprezentáciu akéhokoľvek objektu v jazyku Lua slúži trieda `LuaObject`, ktorá implementuje aj konverziu lua typu na C# typ. Keďže väčšina tried v tomto module pracuje priamo s pamäťou, je potrebné myslieť na to, že túto pamäť treba manuálne dealokovať pretože nebude odstránená automaticky garbage collectorom. Pre zjednodušenie je každá takáto trieda implementovaná návrhovým vzorom `Dispose`. Je na to potrebné myslieť najmä pri volaní metód `LuaObject.GetObject` a `LuaInterface.GetObject` (a každej inej, ktorá priamo vracia `LuaObject`), pretože dealokácia pamäti, ktorú si alokuje `LuaObject`, je na tom, kto túto funkciu zavolal.

Pre zjednodušenie načítania dát pre uzly a hrany z tabuľky data v LuaDb grafe bol vytvorený atribút `LuaParameterAttribute`. Každý údaj v tabuľke `data`, ktorý chceme načítať by mal mať svoj ekvivalent v triede `LuaNodeData/LuaNodeEdge` alebo `GraphObjectData`, ak ide o spoločnú vlastnosť ako je napríklad farba. Každá premenná tejto triedy by mala byť označená atribútom `LuaParameter`, kde v parametri `LuaName` by mal byť názov tejto premennej v `data` tabuľke. Všetky premenné označené týmto atribútom sú automaticky pri vytvorení objektu typu `GraphObject` načítané z LuaDB grafu vo funkcii `GraphObject.LoadData`. Pri

odstránení objektu typu `GraphObject` je automatický dealokovaná všetka pamäť, ktorú tieto dáta používajú, takže nie je potrebné volať metódu `Dispose` na žiadnej premennej označenej týmto atribútom.

GraphCore

V tomto module je obsiahnutá celá logika týkajúca sa grafu. V metóde `Graph.InitializeGraph` dochádza k vytvoreniu grafu z `Lua.Common.Graph`. K ďalšiemu aktualizovaniu tohto grafu z LuaDb grafu dochádza v metóde `Graph.UpdateNodes`, ktorá sa prostredníctom `GraphManager`-a volá v pravidelných intervaloch z unity herného objektu `GraphLoader`. Súčasťou modulu je trieda `LayoutManager`, ktora je reprezentáciou layout managera, ktorý sa nachádza v module `layouter`.

Vlastnosti objektov typu `GraphCore.Node` a `GraphCore.Edge`, ktoré sa pravidelne menia, ako napríklad farba, pozícia, tvar a podobne, by mali obsahovať aj príslušnú udalosť, ktorá signalizuje zmenu tohto atribútu, aby bolo možné podľa neho následne aktualizovať vizuálnu reprezentáciu grafu v Unity.

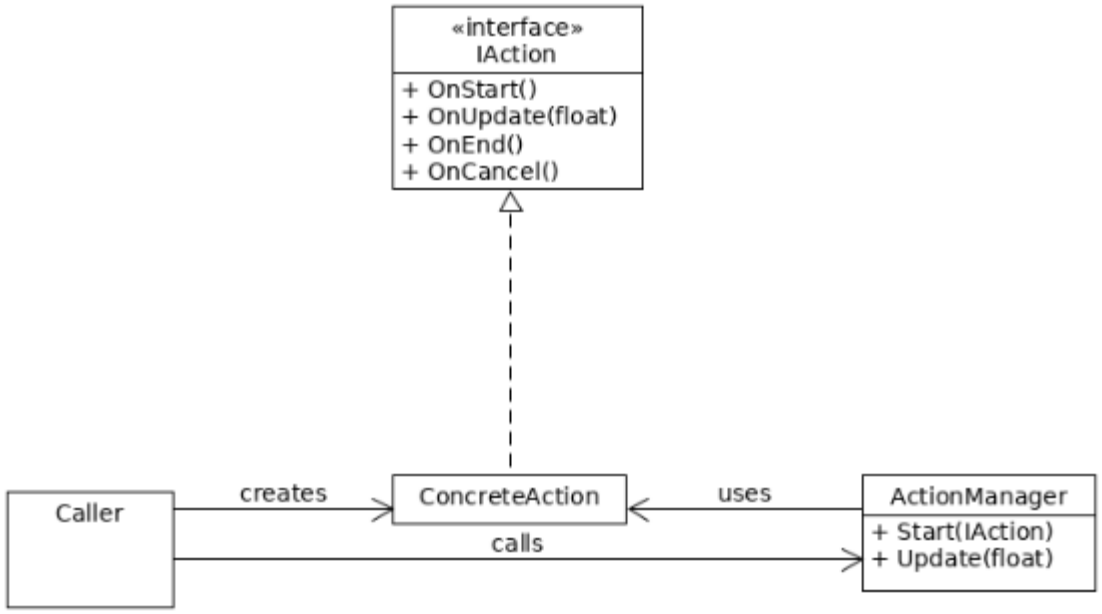
Unity

Modul tvorí prezentačnú vrstvu celého projektu. Vizualizuje graf analyzovaného zdrojového kódu a umožňuje manipuláciu grafu ako celku aj s jeho jednotlivými vrcholmi a prepojeniami. Graf zobrazuje komplexitu analyzovaného zdrojového kódu a prepojenia medzi modulmi. Rámec Unity si vyžaduje implementáciu skriptov v C#.

Súčasťou tohto modulu sú dodatočné moduly ako podpora vykresľovania vo virtuálnej realite na Oculus Rift alebo HTC Vive a interakciu s grafom pomocou senzoru rúk Leap Motion, vykresľovanie v rozšírenej realite a manipuláciu s grafom sledovaním špeciálnych symbolov pomocou bežnej kamery.

Actions

Tento modul vytvára jednotný systém pre spúšťanie znovupoužiteľných kusov kódu spúšťané z grafického rozhrania. Je to implementácia Command patternu. Diagram Command patternu je možné vidieť na obrázku nižšie.



Akcie sa spúšťajú pomocou metódy `ActionManager.Start`. `IAction` má svoj životný cyklus `OnStart`, `OnUpdate`, `OnEnd`, `OnCancel`. Taktiež určuje svoju `IActionGroup`. V jeden moment nemôžu prebiehať viaceré akcie z rovnakej `IActionGroup`. Ak sa začne iná akcia, tak sa predošlá zruší a zavolá sa jej `OnCancel`. Akcie majú predvolenú skupinu `ActionGroup.None`, pre ktorú tieto vzájomné vylučovanie neplatí.

Taktiež je zadefinovaná `OneTimeAction`, ktorá je špeciálny prípad `IAction`, ktorá nemá svoj životný cyklus a vykoná sa iba jedenkrát.

AR Technológie

Na spustenie AR scény sú nutné okrem VR/AR okuliarov 3 technológie. Tie sú:

- LEAP Motion Senzor
- ZED Kamera
- HoverUI Kit

ZED Kamera poskytuje zobrazenie priestoru okolo používateľa. V projekte je použité SDK v2.8. Pred používaním je nutné SDK najprv nainštalovať. To je dostupné na stránke: `https://www.stereolabs.com/developers/`. Kamera dokáže okrem snímania aj vnímať okolie ako 3D priestor vďaka hĺbkovému snímaniu. V projekte sa preto používa primárne na dve veci. Prvou je že dokáže vykonávať "spatial mapping", čo je vytvorenie digitálneho meshu z prostredia a následne ho aj ukladať. Druhou vlastnosťou je "depth perception", pri ktorej ZED kamere zadáme dvojrozmerné súradnice a ona nám dokáže vrátiť vzdialenosť kamery od body, na ktorý sa pozeráme. Táto funkcia sa používa na umiestňovanie grafu na vybraný bod.

Do scény sa po nainštalovaní pridá modul, ktorý sa nazýva Zed_Rig_Stereo. Ten slúži namiesto kamery v scéne a taktiež obsahuje komponent ZED Manager, čo je knižnica funkcií

ZED kamery. Pokiaľ potrebujeme využiť funkciu tejto knižnice, v skripte si vytiahneme práve tento komponent.

LEAP Motion zabezpečuje sledovanie pohybu rúk. Použite je SDK v2.3.1. Pre inštaláciu je nutné stiahnuť vyššie spomínaný SDK zo stránky: `https://developer.leapmotion.com/get-started/` a nainštalovať. Parametre sú v Unity už prednastavené. LEAP je v tejto scéne primárne kvôli interaktívnemu menu HoverUI.

Na sprístupnenie LEAP senzoru v AR scéne je potrebné na kameru pripnúť LEAP XR Service Priovider a na sprístupnenie rúk treba pridať objekt LEAP Rig do scény.

HoverUI Kit slúži ako interaktívne menu, ktoré je pripnuté na rukách pomocou LEAP senzoru. Inštalácia nie je potrebná, všetky potrebné prostriedky sú už v projekte. Aktivuje sa obrátením ľavej ruky smerom nahor a naviguje sa ním pomocou ukazováka pravej ruky. Pomocou tohto menu sa interaguje so samotným grafom, teda funkcie ako zapnúť layoutovač, zmeniť typy uzlov/hrán alebo sa pomocou neho umiestni graf na miesto, na ktoré sa momentálne používateľ pozerá.

Na pripnutie Hover UI menu na LEAP ruky je potrebné na Leap Rig pripnúť Hover Input Leap Motion.

Camera Manager

Skript zahŕňa funkcionalitu používanú pri interakcii s kamerou. Zabezpečuje efektívne manipulovanie a pohyb vo vizualizovanom grafe. Medzi hlavnú funkcionalu tejto časti systému patrí približovanie sa ku konkrétnym uzlom (FocusObject), orbitovanie okolo jedného alebo skupiny uzlov (OrbitalMode) a automatické oddialenie kamery do vzdialenosti potrebnej na zobrazenie kompletného grafu (ZoomToFit).

- FocusObject - funkcionalita zabezpečujúca priblíženie kamery ku konkrétnemu uzlu po vykonaní akcie používateľa. Vyvolanie samotnej akcie zabezpečuje metóda ZoomInTo(clickedObjectTransform), ktorej parameter clickedObjectTransform nesie používateľov zvolený objekt. V procese približovania má kamera prednastavené dva stavy - isZoomed a isZooming v závislosti od toho, či je kamera k uzlu priblížená, alebo je v režime približovania (UpdateZoomInTo). Pri tomto režime sú použité štandardné operácie pre interpoláciu a rotáciu, ktorými Unity disponuje.
- OrbitalMode - orbitálny mód, ktorého úlohou je centrovanie kamery na žiadaný bod a následné orbitovanie. Nemusí ísť nutne o uzol v grafe, funkcionalita ponúka aj orbitovanie okolo ťažiska viacerých vybraných uzlov používateľom.
    - Selekcia uzla/uzlov: Poskytuje SelectionManager, ktorého metóda GetSelectedObjects() vracia zoznam označených uzlov.
    - Výpočet ťažiska: Zo zoznamu označených uzlov vykonáva výpočet pozície ťažiska.
    - Interakcia: Prvým krokom je vycentrovanie kamery na žiadaný bod. Držaním zvoleného tlačidla je možné myšou orbitovať po X-ovej alebo Y-ovej osi.

- ZoomToFit - funkcionalita, ktorá po jej zavolaní oddiali, alebo priblíži kameru od grafu tak, aby sa zmestil na obrazovku. Vyvolanie samotnej akcie zabezpečuje metóda ZoomToFit(). Táto metóda ráta cieľovú pozíciu a natočenie kamery, ktoré musí kamera dosiahnuť, aby sa dosiahol výsledný efekt. V procese približovania / oddiaľovania má kamera prednastavené dva stavy - isZoomed a isZooming v závislosti od toho, či je dosiahnutý cieľový stav kamery, alebo je v režime pohybu k tomuto stavu (UpdateZoomToFit).

Všetky akcie sú aplikované za pomoci Action Patternu a ich vstupy sú spracovávané Input Handlerom.

HoverUI Kit

K modulu HoverUI Kit patria, v rámci AR scény, herné objekty:

HoverCast

Herný objekt obsahuje všetky vizuálne vlastnosti, komponenty a definície animácií, ktoré sa dejú pri používaní menu. Vyskytujú sa v ňom 4 objekty: * OpenItem - definícia tlačidla na otváranie/zatváranie menu. * TitleItem - obsahuje nastavenie textu pre názov menu. * BackItem - definícia tlačidla pre krok späť, ktoré sa nachádza sa v strede dlane. * Rows - aktuálne rozloženie hlavného menu, ktoré môžeme upravovať (prípadne pridávať nové položky).

HoverMock

Dcérsky herný objekt Mock obsahuje makety všetkých nami používaných typov komponentov v hlavnom menu. Pri vytváraní nových komponentov je možné tieto herné objekty kopírovať, čím sa môžeme vyhnúť zdĺhavému a komplikovanému manuálnemu vytváraniu komponentov prostredníctvom externých HoverBuilderov. Pri nakopírovaní netreba zabudnúť na zmenu názvu herného objektu a jeho vlastností ID a Label (prípadne Navigate To Row pre typ RowSelector, Radio Group ID pre typ Radio, Range Min a Range Max pre Slider).

Tento objekt obsahuje makety komponentov:

- Row - riadok vnoreného menu po kliknutí na RowSelector, ktorý obsahuje konkrétne komponenty.
- RowSelector - komponent typu Selector, ktorý sa stará o navigáciu do vnoreného menu (Row).
- Back - komponent typu Selector, ktorý má za úlohu o navigáciu do rodičovského menu (krok späť).
- Selector - komponent, ktorý sa správa ako klasické tlačidlo - neuchováva si stav.
- Radio - komponent, ktorý sa spojí s ostatnými komponentami tohto typu prostredníctvom Group ID - v jednej skupine môže byť označené maximálne jedno Radio tlačidlo.
- CheckBox - tlačidlo, ktoré si uchováva stav (vieme určiť, či bol stlačený alebo nie).

- Slider - posuvník, ktorý sa je možné nastaviť na jednu z hodnôt z intervalu .
- TextItem - komponent, ktorý zobrazuje text.

HoverUIManager

Tento herný objekt je jedným z manažérov v AR scéne, ktoré sú uložené v objekte ManagersAR. Obsahuje referencie na všetky komponenty HoverUI menu a skript HoverUIManager, ktorý na začiatku životného cyklu priradí komponentom event handlery, ktoré budú pri interakcii spúšťať jednotlivé akcie z Action Pattern-u.

```
public class HoverUIManager : MonoBehaviour {

    private GraphLoader graphLoader;
    // ...
    public HoverItemDataCheckbox pauseLayouting;

    public void Start()
    {
        graphLoader = GameObject.FindGameObjectWithTag(GameObjectTags.GraphLoader).GetComponent<GraphLoader>();

        // ...

        pauseLayouting.OnSelected += (sender) =>
        {
            if (pauseLayouting.Value)
            {
                ActionManager.Instance.Start(new PauseLayoutingAction(graphLoader.GraphManager.Graphs));
            }
            else
            {
                ActionManager.Instance.Start(new ResumeLayoutingAction(graphLoader.GraphManager.Graphs));
            }
        };
    }

    // ...
}
```

Ukážku priradenia eventu v kóde zobrazuje obrázok vyššie a sumár komponentov s ich udalosťami tabuľka nižšie.

Komponent	**Udalosť**
HoverItemDataSelector	*OnSelected*
HoverItemDataCheckbox	*OnSelected, OnDeselected*
HoverItemDataRadio	*OnSelected, OnDeselected*
HoverItemDataSlider	*OnValueChanged*

Input Handler

Tento modul vytvára jednotný systém pre spracovanie vstupov. Desktopová časť zaobaľuje Unity input systém a AR časť zaobaľuje Leap input systém. Modul pracuje na základe Observer patternu, kde skript, ktorý chce používať Input sa prihlási na vstup metódou `InputHandler.Subscribe`. Akcie pre vstupy sú zadefinované v `IBaseInputAction`. Konkrétne definície vstupu si definujú jednotlivé spôsoby vstupov.

(napr. `DesktopInputAction`, `LeapInputAction`). `InputHandler.Subscribe` zároveň vracia `InputSubscribtion`, s ktorou je možné input zrušiť cez metódu `InputHandler.Unsubscribe`.

Callbacky pre inputy zavolá `InputHandler`, pričom dozerá na to, aby zároveň neboli zavolané callbacky pre rovnaké vstupy (napr. Ctrl+Mouse0 a zároveň Mouse0). Jedna vstupová akcia resp. vstupová udalosť je definovaná ako `InputEvent`, pričom sa event skladá z jedného alebo viacerých `InputElement`. `InputElement` sa dá predstaviť ako jedna klávesa alebo axis (napr. stlačenie Mouse0) a `InputEvent` ako kombinácia týchto elementov (napr. Ctrl+Mouse0).

Desktopová časť inputov zaobaľuje Unity konštanty a metódy:

- `Input.Desktop.Key` - zaobaľuje `UnityEngine.KeyCode`
- `Input.Desktop.Axis` - zaobaľuje `UnityEngine.Axis`, resp. string

Label Manager

Tento skript obsahuje konštanty a funkcie, ktoré sú potrebné pre zobrazovanie jednotlivých popiskov pre uzly alebo hrany. Skript konkrétne obsahuje:

- konštanty:
- LabelCharSet - hodnota typu float, ktorá ovplyvňuje všeobecnú veľkosť popiskov.
- showAllLabels - indikuje, či sa majú popisky zobraziť ihneď po načítaní grafu.
- funkcie:
    - ToggleAllLabels - prepína všetky popisky medzi stavmi viditeľné a neviditeľné.
    - ShowLabel - zobrazí popisok pre určitý herný objekt.
    - HideLabel - skryje popisok pre určitý herný objekt.
    - ShowLabels - zobrazí popisky pre poskytnuté herné objekty.
    - HideLabels - skryje popisky pre poskytnuté herné objekty.

Selection Manager

Tento skript slúži na označovanie uzlov a hrán v grafe. Ponúka tieto funkcie:

- Select - po kliknutí na uzol/hranu sa daný objekt označi, a pridá sa do zoznamu označených objektov
- MultiSelect - pri držaní CTRL a označovaní uzlov/hrán sa pridávajú označené objekty do ISet
- Unselect - ak je uzol/hrana už označený a opätovne naň klikneme, objekt sa odznačí
- UnselectAll - pokiaľ klikneme mimo grafu alebo máme označný objekt/-y a klikneme na iný objekt bez držania CTRL, ostatné označené objekty sa odznačia

Všetky grafové objekty, ktoré sú označené sa udržujú v ISet selectedObjects, ktorý si dokážeme getnúť ak by sme k nim potrebovali pristúpiť zvonku. Po vizuálnej stránke v projekte vidíme označené objekty pomocou oranžového obrysu.

Infraštruktúra

Build system

Build systému je vykonávaný prostredníctvom nástroja `make`, ktorý je známy z prostredia unixových operačných systémov. Ďalšou možnou alternatívou je použitie vývojového prostredia `Visual Studio` pre platformu MS Windows.

Generovanie súborov potrebných pre build je vykonané nástrojom `CMake`, v ktorom je možné špecifikovať, pre ktorý build systém majú byť súbory určené.

Build systém `3DSoftviz_Remake` je zložený z build súborov pre jednotlivé moduly, na vrchole ktorých je build samotnej aplikácie. Jednotlivé repozitáre obsahujú súbor s názvom `CMakeLists.txt`, ktorý obsahuje postupnosť krokov, ktorá má byť vykonaná. Ide prevažne o vytvorenie `Makefile` súborov, nastavenie premenných prostredia a taktiež nastavenie príznakov pre kompilátor. V nástroji CMake je možné definovať viaceré závilosti. Najskôr teda dôjde k buildu jednotlivých modulov a až následne C# modulov a Unity. Všetky potrebné moduly sú nástrojom stiahnuté z GitLab a GitHub repozitárov. V prípade, ak zlyha build niektorého z modulov, zlyhá celý build proces.

CMake

V rámci celého projektu je použitý nástroj CMake, ktorý výrazne zjednodušuje zostavovanie, testovanie a balíkovanie projektu. Každý modul, ktorý využíva CMake obsahuje konfiguračný súbor `CMakeLists.txt`. Na základe konfiguračného súboru je potom vytvorený jeden alebo viacero `Makefiles` na Unix platformách, prípadne `projekt/workspace` na platforme Windows. Výhod tohto riešenia je viacero, medzi hlavné patria : - podpora viacnásobnej závislosti, - spolupráca s nástrojom Git, - možnosť definície makier, príkazov a funkcií, - možnosť definovať poradie vykonávania - závislosť pri kompilácii a linkovaní. - možnosť parametrizovať správanie - podpora multiplatformových riešení

Prehľad základných príkazov

- **set**
    - slúži na nastavenie hodnoty premennej,
- **option**
    - konfigurovateľný prepínač - umožňuje interaktívne meniť parametre,
- **add_library**
    - pridá build target pre knižnicu, ktorá sa zostaví zo zdrojových súborov zadaných ako parameter,
- **add_dependencies**
    - pridá závislosť pre target, čo tiež znamená, že závislosť musí byť zostavená skôr ako target, ktorý je od nej závislý,
- **ExternalProject_Add**

        - umožňuje pridať externý projekt, čím dôjde k vytvoreniu nového target-u,
    - **set_target_properties**
        - príkazom je možné meniť vlastnosti target-u,
    - **message**
        - umožňuje vypísať správu do konzoly,
    - **add_custom_command**
        - príkazom je možné vytvoriť vlastný príkaz, v ktorom je možné špecifikovať závislosti, command-line-príkaz a správu, ktorá sa má vypísať do konzoly,
    - **add_custom_target**
        - pridá target so zadaným menom, ktorý vykoná požadované príkazy,
    - **install**
        - príkaz generuje inštalačné pravidlá pre projekt,
        - na zdrojový adresár je možné aplikovať viaceré príkazy,
        - základné použitie v projekte zahŕňa kopírovanie súborov závislostí do adresárovej štruktúry projektu.

CMake ďalej podporuje štandardnú syntax pre vetvenie a cykly.

Poznámky

Pri vytváraní alebo modifikácii `CMakeLists.txt` súborov treba dodržiavať Unix adresárovú štruktúru.

- knižnice uchovávať v adresári s názvom `lib`,
- hlavičkové súbory v adresári s názvom `include`,
- spustiteľné súbory v adresári s názvom `bin`,
- ostatné súbory modulov (licence, readme) v adresári s názvom `share`.

Adresár s názvom `cmake` v koreňovom adresári repozitára obsahuje súbory určené na hľadanie závislostí, akými sú hlavičkové súbory, knižnice, spustiteľné súbory a pod. Spravidla je potrebné vytvoriť takýto súbor na úrovni jednotlivých modulov. CMake umožňuje ďalej definovať množinu viacerých možných názvov hľadaných súborov a ciest. Vyhľadanie závislostí je potom potrebné explicitne zavolať v súbore `CMakeLists.txt` príkazom `find_package`.

Help

3DSoftvis_Remake

Build systém projektu 3DSoftvis_Remake je zložený z veľkého množstva modulov, ktoré do projektu vnášajú značnú mieru závislosti. Tým je komplexnosť celého riešenia zvýšená. Zároveň však umožňuje modifikovať projekt na úrovni jednotlivých modulov. Zložitosť celého riešenia spočíva v prepojení 3 programovacích jazykov (pokiaľ nepočítame syntax CMake). V jazyku Lua sú reprezentované jednotlivé grafové štruktúry. Unity, v ktorom je

vytvorená prezenčná časť aplikácie na svoju činnosť využíva moduly, ktoré sú vytvorené v jazyku C#. Aby však bolo možné zabezpečiť komunikáciu medzi jazykmi Lua a C#, je nevyhnutné použiť jazyk C++, v ktorom sú implementované moduly `LuaInterface` a `LuaGraph`.

Build systém projektu musí teda zabezpečiť build jednotlivých C++ modulov a aj C# modulov a následne ich skopírovať do príslušnej adresárovej štruktúry.

V projekte Unity používa C# moduly. C# moduly k svojej činnosti potrebujú dynamické knižnice vytvorené v C++ - `LuaInterface` a `LuaGraph`. C++ prostredníctvom knižníc, bližšie špecifikovaných pri obidvoch moduloch, dokáže komunikovať s jazykom Lua, prípadne jeho nadstavbou - Terra.

Testovanie C# modulov je realizované prostredníctvom frameworku NUnit.

Pokyny pre build projektu pre platformy MS Windows a macOS sú uvedené v inštalačnej príručke.

Prerekvizity

Prerekvizity uvádzame v inštalačnej príručke pre Windows a macOS.

Prepínače

Pri zostavovaní projektu je možné použiť viacero prepínačov.

BUILD_UNITY

Použitím prepínača bude zbuildovaná aj Unity časť projektu - je potrebné mať Unity nainštalované.

BUILD_SOFTVIZ_MODULES_TESTS

Použitie prepínača spôsobí, že budú zbuildované jednotkové testy C# modulov.

RUN_IMPORTER_TEST

S použitím prepínača dôjde k spusteniu testu importovania grafu prostredníctvom modulu `Busted`

USE_TERRA_BINARIES

Prepínač umožňuje použiť nadstavbu nad jazykom Lua - `Terra`.

Závislosti

Pre build modulu sú potrebné nasledujúce závislosti :

- `Terra` (v prípade implementácie pre macOS je zahrnutá v implementácii Luapower)
- `Lua` (v prípade implementácie pre macOS je použitá implementácia LuaJIT, ktorá je súčasťou balíka Luapower)
- `LuaGraph` - modul na prácu s grafovými štruktúrami (modul automaticky stiahne a zbuilduje aj modul `LuaInterface` - rozhranie medzi jazykmi C#, C++ a Lua)
- `doxygen` - nástroj na automatické generovanie dokumentácie zo zdrojového kódu
- `lpeg` - knižnica pre porovnanie textových dát v jazyku Lua
- `leg` - knižnica exportuje kompletnú gramatiku Lua 5.1 a API pre manipuláciu
- `luametrics` - knižnica pre analýzu metrík zdrojového kódu
- `luafilesystem` - súborový systém pre jazyk Lua
- `lualogging` - API na logovanie v jazyku Lua
- `StackTracePlus` - modul poskytuje vylepšený StackTrace pre debug v jazyku Lua
- `luasocket` - sieťová podpora pre jazyk Lua
- `mobdebug` - vzdialený debugger pre jazyk Lua
- `luacov` - analyzátor pokrytia zdrojového kódu
- `luacheck` - statický analyzátor jazyka Lua
- `lua_cliargs` - knižnica na spracovanie command-line argumentov
- `luasystem` - platformovo nezávislá knižnica na vykonávanie systémových volaní
- `dkjson` - JSON modul pre jazyk Lua
- `say` - modul pre ukladanie mapovaní kľúč-hodnota
- `luassert` - rozšírený testovací modul
- `lua-term` - knižnica pre prácu s terminálom
- `penlight` - knižnica poskytuje input data handling, funkcionálne programovanie a správu ciest OS
- `mediator_lua` - knižnica na správu udalostí (events)
- `busted` - framework pre testovanie
- `luacomments` - parser Lua komentárov
- `luameg` - Moonscript parser
- `luadb` - modul na analýzu hypergraph štruktúr
- `luagit` - modul pre jazyk Lua umožňujúci prístup ku Git repozitárom

LuaInterface

Generovanie build súborov pre modul LuaInterface je realizované prostredníctvom nástroja CMake. Jednotlivé kroky, závislosti a voliteľné súčasti sa nachádzajú v súbore `CMakeLists.txt` v koreňovom adresári modulu.

Prepínače

Pri zostavovaní modulu je možné použiť viacero prepínačov.

USE_COTIRE

Použitím prepínača je možné použiť modul `Cotire` na urýchlenie build procesu.

DOWNLOAD_AND_BUILD_LUA

Pri použití prepínača je stiahnutá a zbuildovaná zdieľaná knižnica pre prácu s programovacím jazykom `Lua`.

USE_TERRA_BINARIES

Prepínač umožňuje použiť nadstavbu nad jazykom Lua - `Terra`.

DOWNLOAD_AND_BUILD_SOL2

Použitím prepínača bude stiahnutá a zbuildovaná knižnica `Sol2`, ktorá slúži ako wrapper nad jazykom Lua pre programovací jazyk `C++`.

BUILD_LUAINTERFACE_TESTS

Použitie prepínača spôsobí, že spolu so zdieľanou knižnicou `luainterface` bude zbuildovaný aj spustiteľný súbor vykonávajúci jednotkové testy modulu.

USE_STRICT_COMPILE_WARNINGS

Pri `DEBUG` režime budú použité striktné varovania kompilátora.

USE_CODE_COVERAGE

Prostredníctvom nástroja `gcovr` bude vygenerovaný report pokrytia zdrojového kódu.

USE_INCLUDE_WHAT_YOU_USE

Pri použití prepínača bude vykonaná analýza zdrojových súborov v jazyku C++ `include-what-you-use` - IWYU.

USE_CLANG_TIDY

Použitím prepínača bude vykonaná analýza zdrojových kódov za účelom odhalenia programovacích chýb, nesprávneho použitia rozhraní, prípadne ďalších chýb.

Závislosti

Pre build modulu sú potrebné nasledujúce závislosti :

- `Terra` (v prípade implementácie pre macOS je zahrnutá v implementácii Luapower)
- `Lua` (v prípade implementácie pre macOS je použitá implementácia LuaJIT, ktorá je súčasťou balíka Luapower)
- `Sol2` (v prípade použitia prepínača)
- `Easyloggingpp` - logovací modul pre jazyk C++
- `Catch2` (v prípade použitia prepínača BUILD_LUAINTERFACE_TESTS) - framework pre jednotkové testovanie v jazyku C++
- `include-what-you-use` (v prípade použitia prepínača USE_INCLUDE_WHAT_YOU_USE) - nástroj na analýzu zdrojových súborov v jazyku C a C++
- `clang-tidy` (v prípade použitia prepínača USE_CLANG_TIDY) - nástroj na analýzu zdrojového kódu za účelom odhalenia chýb
- `cpplint` - nástroj slúži na statickú analýzu kódu a overuje, či zdrojový kód v jazyku C++ spĺňa konvencie spoločnosti Google
- `cppcheck` - nástroj slúži na statickú analýzu zdrojového kódu písaného v jazyku C++
- `cmakelint` - nástroj vykonáva statickú analýzu CMake súborov
- `astyle` - nástroj na automatické formátovanie zdrojového kódu
- `doxygen` - nástroj na automatické generovanie dokumentácie zo zdrojového kódu

Build modulu

Prerekvizity

- `CMake 3.15.4`
- CMake je nástroj na správu procesu zostavovania softvéru, podporuje adresárovú hierarchiu a viacnásobné závislosti.
- `Doxygen 1.8.16`
- Doxygen je nástroj na generovanie dokumentácie z anotovaného zdrojového kódu aplikácie.
- `Git`
- potrebné je nahrať svoj verejný kľúč na **GitLab**, ale aj **GitHub**

Naklonovanie repozitára

- pokiaľ sme nastavili SSH kľúč, repozitár naklonujeme prostredníctvom SSH `git clone git@gitlab.com:FIIT/Common/Cpp/LuaInterface.git` a prejdeme do adresára `cd luainterface/`,

Výber vetvy

- príkazom `git checkout remake` vyberieme požadovanú vetvu - v našom prípade označenú remake

Kofigurácia build systému

- otvoríme program CMake,
- ako `source code` zvolíme priečinok `luainterface`,
- ako `build binaries` zvolíme priečinok `luainterface/_build`,
- stlačíme tlačidlo configure,
- ako generátor použijeme `Unix Makefiles` v prípade macOS a Linux, inak `Visual Studio 15 2017 Win64` (alebo inú verziu Visual Studio) a vyberieme možnosť `Use default native compilers`,
- zvolíme prepínače `DOWNLOAD_AND_BUILD_SOL2`, `BUILD_LUAINTERFACE_TESTS` a `DOWNLOAD_AND_BUILD_LUA`
- ako `CMAKE_INSTALL_PREFIX` zvolíme priečinok `luainterface/_install`,
- stlačíme tlačidlo configure,
- stlačíme tlačidlo generate

Build modulu

V prípade MS Windows je možné build vykonať v nástroji Visual Studio :

- Otvoríme projekt
- Pravým tlačidlom stlačíme Solution LuaInterface
- Vyberieme možnosť Build Solution

Pri platforme macOS postupujeme nasledujúcim spôsobom :

- prejdeme do adresára so súbormi vygenerovanými nástrojom CMake `cd luainterface/_build`
- spustíme build modulu `make install`
- overíme, či boli súbory zbuildované a nakopírované do adresára luainterface/_install

LuaGraph

Generovanie build súborov pre modul LuaGraph je realizované prostredníctvom nástroja CMake. Jednotlivé kroky, závislosti a voliteľné súčasti sa nachádzajú v súbore `CMakeLists.txt` v koreňovom adresári modulu.

Prepínače

Pri zostavovaní modulu je možné použiť viacero prepínačov.

USE_COTIRE

Použitím prepínača je možné použiť modul `Cotire` na urýchlenie build procesu.

DOWNLOAD_AND_BUILD_LUAINTERFACE

Pri použití prepínača je stiahnutý a zbuildovaný modul `LuaInterface`, ktorý slúži ako rozhranie medzi jazykmi C#, C++ a Lua.

USE_TERRA_BINARIES

Prepínač umožňuje použiť nadstavbu nad jazykom Lua - `Terra` v module `LuaInterface` - nie je použitá priamo v module LuaGraph.

BUILD_LUAGRAPH_TESTS

Použitie prepínača spôsobí, že spolu so zdieľanou knižnicou `luagraph` bude zbuildovaný aj spustiteľný súbor vykonávajúci jednotkové testy modulu.

USE_STRICT_COMPILE_WARNINGS

Pri `DEBUG` režime budú použité striktné varovania kompilátora.

USE_CODE_COVERAGE

Prostredníctvom nástroja `gcovr` bude vygenerovaný report pokrytia zdrojového kódu.

USE_INCLUDE_WHAT_YOU_USE

Pri použití prepínača bude vykonaná analýza zdrojových súborov v jazyku C++ `include-what-you-use` - IWYU.

USE_CLANG_TIDY

Použitím prepínača bude vykonaná analýza zdrojových kódov za účelom odhalenia programovacích chýb, nesprávneho použitia rozhraní, prípadne ďalších chýb.

Závislosti

Pre build modulu sú potrebné nasledujúce závislosti :

- `LuaInterface` - rozhranie medzi jazykmi C#, C++ a Lua
- `Easyloggingpp` - logovací modul pre jazyk C++
- `Catch2` (v prípade použitia prepínača BUILD_LUAINTERFACE_TESTS) - framework pre jednotkové testovanie v jazyku C++
- `include-what-you-use` (v prípade použitia prepínača USE_INCLUDE_WHAT_YOU_USE) - nástroj na analýzu zdrojových súborov v jazyku C a C++
- `clang-tidy` (v prípade použitia prepínača USE_CLANG_TIDY) - nástroj na analýzu zdrojového kódu za účelom odhalenia chýb
- `cpplint` - nástroj slúži na statickú analýzu kódu a overuje, či zdrojový kód v jazyku C++ spĺňa konvencie spoločnosti Google
- `cppcheck` - nástroj slúži na statickú analýzu zdrojového kódu písaného v jazyku C++
- `cmakelint` - nástroj vykonáva statickú analýzu CMake súborov
- `astyle` - nástroj na automatické formátovanie zdrojového kódu
- `doxygen` - nástroj na automatické generovanie dokumentácie zo zdrojového kódu
- `lpeg` - knižnica pre porovnanie textových dát v jazyku Lua
- `leg` - knižnica exportuje kompletnú gramatiku Lua 5.1 a API pre manipuláciu
- `luametrics` - knižnica pre analýzu metrík zdrojového kódu
- `luafilesystem` - súborový systém pre jazyk Lua
- `lualogging` - API na logovanie v jazyku Lua
- `luasocket` - sieťová podpora pre jazyk Lua
- `mobdebug` - vzdialený debugger pre jazyk Lua
- `luacov` - analyzátor pokrytia zdrojového kódu
- `luacheck` - statický analyzátor jazyka Lua
- `lua_cliargs` - knižnica na spracovanie command-line argumentov
- `luasystem` - platformovo nezávislá knižnica na vykonávanie systémových volaní
- `dkjson` - JSON modul pre jazyk Lua
- `say` - modul pre ukladanie mapovaní kľúč-hodnota
- `luassert` - rozšírený testovací modul
- `lua-term` - knižnica pre prácu s terminálom
- `penlight` - knižnica poskytuje input data handling, funkcionálne programovanie a správu ciest OS
- `mediator_lua` - knižnica na správu udalostí (events)

- `busted` - framework pre testovanie
- `luacomments` - parser Lua komentárov
- `luameg` - Moonscript parser
- `luadb` - modul na analýzu hypergraph štruktúr

Build modulu

Prerekvizity

- `CMake 3.15.4`
- CMake je nástroj na správu procesu zostavovania softvéru, podporuje adresárovú hierarchiu a viacnásobné závislosti.
- `Doxygen 1.8.16`
- Doxygen je nástroj na generovanie dokumentácie z anotovaného zdrojového kódu aplikácie.
- `Git`
- Potrebné je nahrať svoj verejný kľúč na **GitLab**, ale aj **GitHub**

Naklonovanie repozitára

- pokiaľ sme nastavili SSH kľúč, repozitár naklonujeme prostredníctvom SSH `git clone git@gitlab.com:FIIT/Common/Cpp/LuaGraph.git` a prejdeme do adresára `cd luagraph/`,

Výber vetvy

- príkazom `git checkout remake` vyberieme požadovanú vetvu - v našom prípade označenú remake

Inicializácia submodulov

Inicializáciu potrebných submodulov vykonáme príkazom `git submodule update --init --recursive`

Kofigurácia build systému

- otvoríme program CMake,
- ako `source code` zvolíme priečinok `luagraph`,
- ako `build binaries` zvolíme priečinok `luagraph/_build`,
- stlačíme tlačidlo configure,
- ako generátor použijeme `Unix Makefiles` v prípade macOS a Linux, inak `Visual Studio 15 2017 Win64` (alebo inú verziu Visual Studio) a vyberieme možnosť `Use default native compilers`,

- zvolíme
  prepínače `DOWNLOAD_AND_BUILD_LUAINTERFACE` a `BUILD_LUAGRAPH_TESTS`
- ako `CMAKE_INSTALL_PREFIX` zvolíme priečinok `luagraph/_install`,
- stlačíme tlačidlo configure,
- stlačíme tlačidlo generate

Build modulu

V prípade MS Windows je možné build vykonať v nástroji Visual Studio :

- Otvoríme projekt
- Pravým tlačidlom stlačíme Solution LuaGraph
- Vyberieme možnosť Build Solution

Pri platforme macOS postupujeme nasledujúcim spôsobom :

- prejdeme do adresára so súbormi vygenerovanými nástrojom CMake `cd luagraph/_build`
- spustíme build modulu `make install`
- overíme, či boli súbory zbuildované a nakopírované do adresára luagraph/_install

LuaDB

Generovanie build súborov pre modul LuaDB je realizované prostredníctvom nástroja CMake. Jednotlivé kroky, závislosti a voliteľné súčasti sa nachádzajú v súbore `CMakeLists.txt` v koreňovom adresári modulu.

Prepínače

BUILD_STANDALONE

Použitie prepínača umožní zostaviť modul LuaDB so všetkými závislosťami za účelom testovania.

Závislosti

Pre build modulu sú potrebné nasledujúce závislosti :

- `luametrics` - knižnica pre analýzu metrík zdrojového kódu
- `luacomments` - parser Lua komentárov
- `lualogging` - logovací modul

Build modulu

Prerekvizity

- `CMake`- CMake je nástroj na správu procesu zostavovania softvéru, podporuje adresárovú hierarchiu a viacnásobné závislosti.
- `Git` - Potrebné je nahrať svoj verejný kľúč na **GitLab**, ale aj **GitHub**

Naklonovanie repozitára

- pokiaľ sme nastavili SSH kľúč, repozitár naklonujeme prostredníctvom SSH `git clone git@gitlab.com:FIIT/Common/Lua/luadb.git` a prejdeme do adresára `cd luadb/`,

Výber vetvy

- príkazom `git checkout` vyberieme požadovanú vetvu

Inicializácia submodulov

Inicializáciu potrebných submodulov vykonáme príkazom `git submodule update --init`

Kofigurácia build systému

- vytvoríme adresár `_build` príkazom `mkdir _build`,
- príkazom `cd _build` prejdeme do adresára `_build`,
- spustíme príkaz `cmake -D BUILD_STANDALONE=ON ..`,

Build modulu

- prejdeme do adresára so súbormi vygenerovanými nástrojom CMake `cd luadb/_build`
- spustíme build modulu a jeho závislostí `make install`.
- nainštalovaný modul a jeho závislosti je možné nájsť v adresári `../_install`.

LuaMetrics

Generovanie build súborov pre modul LuaMetrics je realizované prostredníctvom nástroja CMake. Jednotlivé kroky, závislosti a voliteľné súčasti sa nachádzajú v súbore `CMakeLists.txt` v koreňovom adresári modulu.

Prepínače

BUILD_STANDALONE

Použitie prepínača umožní zostaviť modul LuaMetrics so všetkými závislosťami za účelom testovania.

Závislosti

Pre build modulu sú potrebné nasledujúce závislosti :

- `lpeg` - knižnica pre porovnanie textových dát v jazyku Lua
- `leg` - knižnica exportuje kompletnú gramatiku Lua 5.1 a API pre manipuláciu
- `luacomments` - parser Lua komentárov

Build modulu

Prerekvizity

- `CMake`- CMake je nástroj na správu procesu zostavovania softvéru, podporuje adresárovú hierarchiu a viacnásobné závislosti.
- `Git`- Potrebné je nahrať svoj verejný kľúč na **GitLab**, ale aj **GitHub**

Naklonovanie repozitára

- pokiaľ sme nastavili SSH kľúč, repozitár naklonujeme prostredníctvom SSH `git clone git@gitlab.com:FIIT/Common/Lua/luametrics.git` a prejdeme do adresára `cd luametrics/`,

Výber vetvy

- príkazom `git checkout` vyberieme požadovanú vetvu

Inicializácia submodulov

Inicializáciu potrebných submodulov vykonáme príkazom `git submodule update --init`

Kofigurácia build systému

- vytvoríme adresár `_build` príkazom `mkdir _build`,
- príkazom `cd _build` prejdeme do adresára `_build`,
- spustíme príkaz `cmake -D BUILD_STANDALONE=ON ..`,

Build modulu

- prejdeme do adresára so súbormi vygenerovanými nástrojom CMake `cd luametrics/_build`
- spustíme build modulu a jeho závislostí `make install`
- nainštalovaný modul a jeho závislosti je možné nájsť v adresári `../_install`.

LuaMeg

Generovanie build súborov pre modul LuaMeg je realizované prostredníctvom nástroja CMake. Jednotlivé kroky, závislosti a voliteľné súčasti sa nachádzajú v súbore `CMakeLists.txt` v koreňovom adresári modulu.

Prepínače

BUILD_STANDALONE

Použitie prepínača umožní zostaviť modul LuaMeg so všetkými závislosťami za účelom testovania.

Závislosti

Pre build modulu sú potrebné nasledujúce závislosti :

- `luadb` - modul na analýzu zdrojového kódu, komplexity, a rôzne metrík, na základe ktorých sa neskôr ostatné moduly rozhodujú o vlastnostiach vykreslenia uzlov

Build modulu

Prerekvizity

- `CMake`- CMake je nástroj na správu procesu zostavovania softvéru, podporuje adresárovú hierarchiu a viacnásobné závislosti.
- `Git`- Potrebné je nahrať svoj verejný kľúč na **GitLab**, ale aj **GitHub**

Naklonovanie repozitára

- pokiaľ sme nastavili SSH kľúč, repozitár naklonujeme prostredníctvom SSH `git clone git@gitlab.com:FIIT/Common/Lua/luameg.git` a prejdeme do adresára `cd luameg/`,

Výber vetvy

- príkazom `git checkout` vyberieme požadovanú vetvu

Inicializácia submodulov

Inicializáciu potrebných submodulov vykonáme príkazom `git submodule update --init`

Kofigurácia build systému

- vytvoríme adresár `_build` príkazom `mkdir _build`,
- príkazom `cd _build` prejdeme do adresára `_build`,
- spustíme príkaz `cmake -D BUILD_STANDALONE=ON ..`,

Build modulu

- prejdeme do adresára so súbormi vygenerovanými nástrojom CMake `cd luameg/_build`
- spustíme build modulu a jeho závislostí `make install`
- nainštalovaný modul a jeho závislosti je možné nájsť v adresári `../_install`.

LuaGit

Generovanie build súborov pre modul LuaGit je realizované prostredníctvom nástroja CMake. Jednotlivé kroky, závislosti a voliteľné súčasti sa nachádzajú v súbore `CMakeLists.txt` v koreňovom adresári modulu.

Prepínače

BUILD_STANDALONE

Použitie prepínača umožní zostaviť modul LuaGit so všetkými závislosťami za účelom testovania.

Závislosti

Pre build modulu sú potrebné nasledujúce závislosti :

- `lualogging` - logovací modul

Build modulu

Prerekvizity

- `CMake`- CMake je nástroj na správu procesu zostavovania softvéru, podporuje adresárovú hierarchiu a viacnásobné závislosti.
- `Git`- Potrebné je nahrať svoj verejný kľúč na **GitLab**, ale aj **GitHub**

Naklonovanie repozitára

- pokiaľ sme nastavili SSH kľúč, repozitár naklonujeme prostredníctvom SSH `git clone git@gitlab.com:FIIT/Common/Lua/luagit.git` a prejdeme do adresára `cd luagit/`,

Výber vetvy

- príkazom `git checkout` vyberieme požadovanú vetvu

Inicializácia submodulov

Inicializáciu potrebných submodulov vykonáme príkazom `git submodule update --init`

Kofigurácia build systému

- vytvoríme adresár `_build` príkazom `mkdir _build`,
- príkazom `cd _build` prejdeme do adresára `_build`,
- spustíme príkaz `cmake -D BUILD_STANDALONE=ON ..`,

Build modulu

- prejdeme do adresára so súbormi vygenerovanými nástrojom CMake `cd luagit/_build`
- spustíme build modulu a jeho závislostí `make install`
- nainštalovaný modul a jeho závislosti je možné nájsť v adresári `../_install`.

LuaTree

Generovanie build súborov pre modul LuaTree je realizované prostredníctvom nástroja CMake. Jednotlivé kroky, závislosti a voliteľné súčasti sa nachádzajú v súbore `CMakeLists.txt` v koreňovom adresári modulu.

Prepínače

BUILD_STANDALONE

Použitie prepínača umožní zostaviť modul LuaTree so všetkými závislosťami za účelom testovania.

Závislosti

Pre build modulu sú potrebné nasledujúce závislosti :

- `lualogging` - logovací modul

Build modulu

Prerekvizity

- `CMake`- CMake je nástroj na správu procesu zostavovania softvéru, podporuje adresárovú hierarchiu a viacnásobné závislosti.
- `Git`- Potrebné je nahrať svoj verejný kľúč na **GitLab**, ale aj **GitHub**

Naklonovanie repozitára

- pokiaľ sme nastavili SSH kľúč, repozitár naklonujeme prostredníctvom SSH `git clone git@gitlab.com:FIIT/Common/Lua/luatree.git` a prejdeme do adresára `cd luatree/`,

Výber vetvy

- príkazom `git checkout` vyberieme požadovanú vetvu

Inicializácia submodulov

Inicializáciu potrebných submodulov vykonáme príkazom `git submodule update --init`

Kofigurácia build systému

- vytvoríme adresár `_build` príkazom `mkdir _build`,
- príkazom `cd _build` prejdeme do adresára `_build`,
- spustíme príkaz `cmake -D BUILD_STANDALONE=ON ..`,

Build modulu

- prejdeme do adresára so súbormi vygenerovanými nástrojom CMake `cd luatree/_build`
- spustíme build modulu a jeho závislostí `make install`
- nainštalovaný modul a jeho závislosti je možné nájsť v adresári `../_install`.

Gitlab CI/CD

Repozitáre 3dsoftvis_remake, LuaGraph, LuaInterface, LuaDB, LuaMeg, LuaMetrics a LuaGit majú plne vybudovanú CI infraštruktúru. Táto infraštruktúra slúži na kontrolu, či je vyvíjaný projekt vo funkčnom stave a pripravený na dodanie zákazníkovi. Infraštruktúra sa stará o build projektu a všetkých jeho variantov spúšťaním automatizovaných jednotkových testov, inštrumentálnych testov, analyzovaním pokryia kódu testami a zverejňovaním ich

výsledkov na GitLab-e. Taktiež analyzuje zdrojový kód a jeho kvalitu pomocou nástroja Lint a ohlasuje možné sémantické problémy. Ďalej sa stará o generovanie dokumentácie. Na generovanie dokumentácie z Markdown súborov je vytvorená samostatná pipeline - MkDocs. Jednotlivé pipeline sú umiestnené v príslušných repozitároch a jedná sa o súbory `.gitlab-ci.yml`.

Ako fungujú pipelines

Pipelines sú najvyššou súčasťou CI, dodania a zavádzania produktu. Medzi základné časti pipeline patria:

- Etapy, ktoré určujú, ktoré úlohy sa majú kedy spustiť. Napríklad testy sa spustia až po úspešnej kompilácii kódu.
- Úlohy, ktoré definujú postupnosť príkazov. Napríklad kompilácia kódu alebo testovanie.

Viac informácií je možné nájsť na stránke GitLab Docs

Premenné (variables)

Premenná je dynamicky pomenovaná hodnota, ktorá môže ovplyvniť spôsob správania jednotlivých procesov. Sú súčasťou prostredia, v ktorom prebieha daný proces. Premenné sú užitočné na prispôsobenie úloh v pipeline GitLab CI / CD.

Viac informácií je možné nájsť na stránke GitLab Docs

Etapy (stages)

Pipeline je zbierka úloh rozdelených do rôznych etáp. Všetky úlohy v tej istej etape prebiehajú súčasne a ďalšia etapa sa začína iba vtedy, ak všetky úlohy z predchádzajúcej etapy boli úspešne ukončené. Akonáhle úloha zlyhá, celá pipeline zlyhá. Ak je úloha označená ako manuálna, zlyhanie nespôsobí zlyhanie pipeline. Etapy sú iba logickým rozdelením medzi dávkami úloh, pri ktorých nemá zmysel vykonávať ďalšiu úlohu, pokiaľ predchádzajúca zlyhala. Žiadna práca by nemala mať závislosť od inej práce v rovnakej etape. Naproti tomu ďalšie etapy môžu používať výsledky predchádzajúcich.

before_script

before_script sa používa na definovanie postupnosti príkazov, ktoré by sa mali spustiť pred každou úlohou vrátane nasadenia, ale až po obnove akýchkoľvek artefaktov z predchádzajúcej etapy.

Skripty špecifikované v skripte `before_script` sú spojené s akýmikoľvek skriptami špecifikovanými v hlavnom skripte a vykonávajú sa spoločne v jednom prostredí.

Artefakty

Artefakty slúžia na export vygenerovaných súborov z etapy, ako napríklad kompiláty, reporty z testov, HTML stránky, dokumentácie, a pod.

Aktuálne pokryté repozitáre

- 3DSoftVis_Remake
- LuaInterface
- LuaGraph
- LuaDB
- LuaMeg
- LuaMetrics
- LuaGit

3DSoftVis_Remake

Pipeline v repozitári 3dsoftvis_remake sa skladá z viacerých úrovní, pričom každá úroveň má na starosti inú časť procesu nasadenia systému. Pipeline sa z dôvodu šetrenia prostriedkov spúšta len pre vetvy s vytvoreným merge requestom.

Premenné

Súbor `gitlab-ci.yml` obsahuje viacero preddefinovaných premenných (príklad: CI_PROJECT_DIR).

`before_script`

- nastavenie ciest k jednotlivým priečinkom projektu podľa obsahu do premenných. Premenné
  - `SOFTVIZ_ROOT`, `SOFTVIZ_MODULES`, `SOFTVIZ_UNITY`, `BUILD_DIR`, `INSTALL_DIR`
- vytvorenie súboru, ktorý obsahuje licenčný kľúč pre Unity.
- nastavenie SSH - kontrola, či je SSH nainštalované. Ak sa želaný súbor nenájde, prebehne inštalácia s automatickou odpoveďou "áno" na interakcie, ktoré počas inštalácie vyskočia. Nasleduje spustenie ssh-agent session, pridanie SSH kľúča agentovi, vytvorenie SSH priečinku so správnymi povoleniami (read, write aj execute) a zozbieranie verejných SSH kľúčov.
- konfigurácia cache pamäte pre kompilátor (kvôli urýchleniu rekompilácií)
- naklonovanie projektu

Etapy

`build` stage

Táto úroveň vykonáva základný build samostatných modulov, ktoré nie sú závislé na prezentačnej vrstve Unity.

`build_modules` stage

Stará sa o buildovanie C# modulov.

`build_unity_player` Táto úroveň vykonáva build Unity playeru (spustiteľnú binárku), vytvorené Unity moduly a použité knižnice Leap Motion, Zed Camera, HoverUI a ZF Browser.

`build_unity_windows` Builduje unity player pre platformu Windows. Táto úloha je povinná.

`build_unity_linux` Builduje unity player pre platformu Linux. Táto úloha je volitelná.

`build_unity_mac` Builduje unity player pre platformu Mac OS. Táto úloha je volitelná.

`QA` stage

Táto úroveň vykonáva testovanie a udržiavanie kvality a zabránenie regresii.

`run_tests` Táto úloha testuje C# moduly.

`run_lua_tests` Táto úloha testuje Lua moduly.

`docs` stage

Táto úroveň vykonáva generovanie dokumentácie

`make_doxygen` Úloha generuje dokumentáciu pomocou nástroju Doxygen pre C# moduly a Unity moduly.

`extract code snippets` Úloha generuje code snippety pomocou doc-extra/snippet-extract.py

Artefakty

`_install` - tento priečinok obsahuje všetky súbory, ktoré vznikli počas buildu pre platformy Windows, Mac Linux.

`TestResult.xml` - priečinok obsahujúci súbory potrebné pre reprezentáciu výsledkov testov vo formáte jednoduchej webstránky.

`3dsoftviz-techdoc` - priečinok obsahujúci dokumentáciu vygenerovanú vo formáte html.

`deploy_docs` stage

Úlohou úrovne je export automatickej dokumentácie zdrojového kódu na vzdialený server zhromažďujúci všetku dokumentáciu - high-level aj low-level.

`deploy_doxygen` Úloha exportuje prostredníctvom programu `rsync` vygenerovanú Doxygen dokumentáciu úrovne `docs` - priečinok `3dsoftviz-techdoc` - v HTML formáte na vzdialený server.

LuaInterface

Pipeline sa skladá z `before_script` a 4 stage. Celá pipeline využíva image `registry.gitlab.com/fiit/common/ci-images/cpp`, ktorý obsahuje potrebné závislosti pre zostavovanie, testovanie a dokumentáciu modulov vytvorených v jazyku C++.

Premenné

V súbore `gitlab-ci.yml` je definovaných viacero premenných : - CMAKE_MANDATORY - obsahuje parametre pre CMake - CCACHE_BASEDIR - cesta k pracovnému adresáru pre `ccache` - CCACHE_DIR - adresár pre `ccache`

`before_script`

- nastavenie SSH klúčov
- vytvorenie adresáru pre `ccache`
- prechod do koreňového adresáru modulu
- klonovanie submodulov

`build` *stage*

- vytvorenie `build` adresáru
- generovanie build systému modulu s parametrami uloženými v premennej `CMAKE_MANDATORY`
- build modulu

`QA` *stage*

Táto úroveň vykonáva testovanie a zabezpečuje udržiavanie kvality zdrojového kódu.

`make_tests` Úloha vykoná build testov prostredníctvom nástrojov CMake a Make. Následne sú vykonané jednotkové testy. Taktiež sa v rámci tejto úlohy vykoná analýza pokrytia zdrojového kódu prostredníctvom nástroja `gcovr`.

`make_cppcheck` V rámci úlohy je vykonaná statická analýza zdrojového kódu písaného v jazyku C++ nástrojom `cppcheck`.

`make_cpplint` Úloha prostredníctvom nástroja `cpplint` vykoná ďalšiu statickú analýzu kódu a overí, či zdrojový kód v jazyku C++ spĺňa konvencie spoločnosti Google.

`make_cmakelint` Poslednou úlohou úrovne je statická analýza CMake súborov nástrojom `cmakelint`.

`docs` stage

- generovanie dokumentácie zo zdrojového kódu nástrojom `doxygen`

`extract code snippets` Úloha generuje code snippety pomocou doc-extra/snippet-extract.py

Artefakty

`_build` - priečinok obsahuje všetky súbory, ktoré vznikli počas vykonávania úlohy `make_tests`

`_build/coverage` - priečinok obsahuje výsledky analýzy pokrytia zdrojového kódu testami vo forme `html`

`_build/tests` - priečinok obsahuje výsledky jednotkových testov vo formáte `txt`

`cpplint-report.txt` - výstup statickej analýzy C++ zdrojového kódu nástrojom `cpplint`

`_build/cppcheck` - adresár obsahuje výstup statickej analýzy zdrojového kódu nástrojom `cppcheck`

`cmakelint-report.txt` - výstup statickej analýzy CMake súborov nástrojom `cmakelint`

`_build/luainterface-techdoc` - priečinok obsahuje dokumentáciu vygenerovanú zo zdrojového kódu nástrojom `doxygen`

`deploy_docs` stage

Úlohou úrovne je export automatickej dokumentácie zdrojového kódu na vzdialený server zhromažďujúci všetku dokumentáciu - high-level aj low-level.

`deploy_doxygen` Úloha exportuje prostredníctvom programu `rsync` vygenerovanú Doxygen dokumentáciu úrovne `docs` - priečinok `luainterface-techdoc` - v HTML formáte na vzdialený server.

LuaGraph

Pipeline sa skladá z `before_script` a 4 stage. Celá pipeline využíva image `registry.gitlab.com/fiit/common/ci-images/cpp`, ktorý obsahuje potrebné závislosti pre zostavovanie, testovanie a dokumentáciu modulov vytvorených v jazyku C++.

Premenné

V súbore `gitlab-ci.yml` je definovaných viacero premenných : - CMAKE_MANDATORY - obsahuje parametre pre CMake - CCACHE_BASEDIR - cesta k pracovnému adresáru pre `ccache` - CCACHE_DIR - adresár pre `ccache`

`before_script`

- nastavenie SSH klúčov
- vytvorenie adresáru pre `ccache`
- prechod do koreňového adresáru modulu
- klonovanie submodulov

`build` stage

- vytvorenie `build` adresáru
- generovanie build systému modulu s parametrami uloženými v premennej `CMAKE_MANDATORY`
- build modulu

`QA` stage

Táto úroveň vykonáva testovanie a zabezpečuje udržiavanie kvality zdrojového kódu.

`make_tests` Úloha vykoná build testov prostredníctvom nástrojov CMake a Make. V rámci úlohy je vykonaná aj dynamická analýza prostredníctvom nástroja `valgrind`. Úloha ďalej zahŕňa aj jednotkové testovanie. Taktiež sa v rámci tejto úlohy vykoná analýza pokrytia zdrojového kódu prostredníctvom nástroja `gcovr`.

`make_cppcheck` V rámci úlohy je vykonaná statická analýza zdrojového kódu písaného v jazyku C++ nástrojom `cppcheck`.

`make_cpplint` Úloha prostredníctvom nástroja `cpplint` vykoná ďalšiu statickú analýzu kódu a overí, či zdrojový kód v jazyku C++ spĺňa konvencie spoločnosti Google.

`make_cmakelint` Poslednou úlohou úrovne je statická analýza CMake súborov nástrojom `cmakelint`.

`docs` stage

- generovanie dokumentácie zo zdrojového kódu nástrojom `doxygen`

`extract code snippets` Úloha generuje code snippety pomocou [doc-extra/snippet-extract.py]

Artefakty

`_build/tests` - priečinok obsahuje výsledky jednotkových testov vo formáte `txt`

`cpplint-report.txt` - výstup statickej analýzy C++ zdrojového kódu nástrojom `cpplint`

`_build/cppcheck` - adresár obsahuje výstup statickej analýzy zdrojového kódu nástrojom `cppcheck`

`cmakelint-report.txt` - výstup statickej analýzy CMake súborov nástrojom `cmakelint`

`_build/luagraph-techdoc` - priečinok obsahuje dokumentáciu vygenerovanú zo zdrojového kódu nástrojom `doxygen`

`deploy_docs` stage

Úlohou úrovne je export automatickej dokumentácie zdrojového kódu na vzdialený server zhromažďujúci všetku dokumentáciu - high-level aj low-level.

`deploy_doxygen` Úloha exportuje prostredníctvom programu `rsync` vygenerovanú Doxygen dokumentáciu úrovne `docs` - priečinok `luagraph-techdoc` - v HTML formáte na vzdialený server.

LuaDB

Pipeline sa skladá z `before_script` a 2 stage. Celá pipeline využíva image `registry.gitlab.com/fiit/common/ci-images/lua`, ktorý obsahuje potrebné závislosti pre zostavovanie, testovanie a dokumentáciu modulov vytvorených v jazyku Lua.

`before_script`

- import SSH klúčov

`build` stage

- sťahovanie závislosti
- cmake a následne make
- vygenerovanie dokumentácie zo zdrojového kódu nástrojom `ldoc` do adresára `doc` v HTML formáte
- vygenerovanie code snippetov pomocou doc-extra/snippet-extract.py

`test` stage

- spustenie testov vo frameworku busted
- kontrola pokrytia zdrojového kódu nástrojom `luacov`

- statická analýza zdrojového kódu v jazyku Lua

`deploy` stage

- export vygenerovanej dokumentácie na vzdialený server prostredníctvom programu `rsync`

LuaMetrics

Pipeline sa skladá z `before_script` a 2 stage. Celá pipeline využíva image `registry.gitlab.com/fiit/common/ci-images/lua`, ktorý obsahuje potrebné závislosti pre zostavovanie, testovanie a dokumentáciu modulov vytvorených v jazyku Lua.

`before_script`

- import SSH klúčov

`build` stage

- sťahovanie závislosti
- cmake a následne make
- vygenerovanie dokumentácie zo zdrojového kódu nástrojom `ldoc` do adresára `doc` v HTML formáte
- vygenerovanie code snippetov pomocou doc-extra/snippet-extract.py

`test` stage

- spustenie testov vo frameworku busted
- kontrola pokrytia zdrojového kódu nástrojom `luacov`
- statická analýza zdrojového kódu v jazyku Lua

`deploy` stage

- export vygenerovanej dokumentácie na vzdialený server prostredníctvom programu `rsync`

LuaMeg

Pipeline sa skladá z `before_script` a 2 stage. Celá pipeline využíva image `registry.gitlab.com/fiit/common/ci-images/lua`, ktorý obsahuje potrebné závislosti pre zostavovanie, testovanie a dokumentáciu modulov vytvorených v jazyku Lua.

`before_script`

- import SSH klúčov

`build` stage

- sťahovanie závislosti
- cmake a následne make
- vygenerovanie dokumentácie zo zdrojového kódu nástrojom `ldoc` do adresára `doc` v HTML formáte
- vygenerovanie code snippetov pomocou doc-extra/snippet-extract.py

`test` stage

- spustenie testov vo frameworku busted
- kontrola pokrytia zdrojového kódu nástrojom `luacov`
- statická analýza zdrojového kódu v jazyku Lua

`deploy` stage¶

- export vygenerovanej dokumentácie na vzdialený server prostredníctvom programu `rsync`

LuaGit

Pipeline sa skladá z `before_script` a 2 stage. Celá pipeline využíva image `registry.gitlab.com/fiit/common/ci-images/lua`, ktorý obsahuje potrebné závislosti pre zostavovanie, testovanie a dokumentáciu modulov vytvorených v jazyku Lua.

`before_script`

- import SSH klúčov

`build` stage

- sťahovanie závislosti
- cmake a následne make
- vygenerovanie dokumentácie zo zdrojového kódu nástrojom `ldoc` do adresára `doc` v HTML formáte
- vygenerovanie code snippetov pomocou doc-extra/snippet-extract.py

`test` stage

- spustenie testov vo frameworku busted
- kontrola pokrytia zdrojového kódu nástrojom `luacov`
- statická analýza zdrojového kódu v jazyku Lua

`deploy` stage

- export vygenerovanej dokumentácie na vzdialený server prostredníctvom programu `rsync`

LuaTree

Pipeline sa skladá z `before_script` a 2 stage. Celá pipeline využíva image `registry.gitlab.com/fiit/common/ci-images/lua`, ktorý obsahuje potrebné závislosti pre zostavovanie, testovanie a dokumentáciu modulov vytvorených v jazyku Lua.

`before_script`

- import SSH klúčov

`generate_doc` stage

- vygenerovanie dokumentácie zo zdrojového kódu nástrojom `ldoc` do adresára `doc` v HTML formáte

`deploy_docs` stage

- export vygenerovanej dokumentácie na vzdialený server prostredníctvom programu `rsync`

Dokumentácia

Pipeline sa skladá z `before_script` a troch stage, `prepare`, `build` a `deploy`, a je spúštaná na vlastnom docker image. Po úspešnom builde a uploade sa dokumentácia nachádza na stránke https://team05-19.studenti.fiit.stuba.sk/docs/projekt v priečinku príslušnej git branche.

Docker image

Obraz sa builduje pomocou `docker build` zo súboru `CI/image-docs/Dockerfile`. Obraz obsahuje predinštalovaný `python3`, `openssh-client` a `rsync` pre upload cez ssh, `mkdocs`, mkdocs tému a niekoľko mkdocs pluginov.

`before_script`

- import SSH klúčov
- nastavenie kódovania

`prepare` stage

- sťahovanie code snippetov z ostatných repozitárov pomocou doc-extra/artifact-download.py
- kontrola platnosti linkov v rámci dokumentácie, teda aby všetky linky smerujúce na iný dokument boli relatívne a smerovali na .md súbor, nie priečinok. Existenciu samotného .md súboru kontroluje mkdocs, pokiaľ má link príponu .md

`build` stage

- odstránenie šablón z dokumentácie
- vygenerovanie HTML dokumentácie do priečinka `site`
- job `build pdf`
    - nastaví premennú prostredia `ENABLE_PDF_EXPORT`
    - vykoná všetko ako v `build` a tým vygeneruje aj PDF
    - tento job sa spúšťa iba manuálne

`deploy` stage

- obsah priečinka `site` sa uploaduje na vzdialený server

CI GitLab Images
C++ moduly

Image slúži na zreťazené spracovanie modulov naprogramovaných v jazyku C++. Image je založený na Ubuntu 18.04 a obsahuje nainštalované závislosti potrebné na zostavenie, testovanie a dokumentáciu jednotlivých modulov, vrátane exportu dokumentácie na vzdialený server.

Závislosti

`gcc` - súbor kompilátorov projektu GNU (C/C++)

`cmake` - nástroj pre multiplatformové zostavenie projektu - vytvára adresárovú štruktúru a pripravuje zdrojové súbory pre zostavenie kompilátormi príslušného operačného systému

`make` - nástroj pre automatizované zostavenie vykonateľných programov alebo knižníc zo zdrojových kódov - vstupom je tzv. Makefile súbor, ktorý obsahuje pokyny pre vytvorenie výsledného programu

`clang-tidy` - nástroj na analýzu zdrojového kódu za účelom odhalenia chýb

`iwyu (include-what-you-use)` - nástroj na analýzu `#include` direktív - každý symbol použitý v zdrojovom kóde by mal obsahovať .h súbor obsahujúci export príslušného symbolu - nástroj teda analyzuje, kedy dochádza k porušeniu a navrhuje opravy

`cppcheck` - nástroj slúži na statickú analýzu zdrojového kódu písaného v jazyku C++

`python` - interpreter jazyka Python

`python-pip` - nástroj pre inštaláciu Python balíkov

`doxygen` - nástroj na automatické generovanie dokumentácie zo zdrojového kódu

`graphviz` - nástroj pre reprezentáciu štrukturálnych informácií vo forme diagramov

`openssh-client` - secure-shell klient pre zabezpečený prístup k vzdialeným zariadeniam

`git` - distribuovaný verziovací systém

`astyle` - nástroj na automatické formátovanie zdrojového kódu

`gcovr` - nástroj na vytváranie reportu pokrytia zdrojového kódu

`rsync` - nástroj na syschronizáciu a prenos súborov medzi vzdialenými systémami

`valgrind` - nástroj na memory debugging, detekciu únikov pamäti (memory leak) a profilovanie

`setuptools` - balíkovací nástroj pre Python projekty

`cmakelint` - nástroj vykonáva statickú analýzu CMake súborov

`cpplint` - nástroj slúži na statickú analýzu kódu a overuje, či zdrojový kód v jazyku C++ spĺňa konvencie spoločnosti Google

`pygments` - zvýrazňovač syntaxe pre jazyk Python

Lua moduly

Image slúži na zreťazené spracovanie modulov naprogramovaných v jazyku Lua a taktiež je ho možné použiť na lokálny vývoj, testovanie a ladenie.
Image je založený na Ubuntu 18.04 a obsahuje nainštalované závislosti potrebné na zostavenie, testovanie, coverage a dokumentáciu jednotlivých modulov vrátane exportu dokumentácie na vzdialený server. Základom pre image je implementácia `Luapower`, obsahujúca interpreter `LuaJIT` a ďalšie nástroje potrebné pre prácu s programovacím jazykom `Lua`.

Argumenty a premenné prostredia

Aby image fungoval správne, je potrebné nastaviť premenné LUA_PATH a LUA_CPATH. Aby bola zabezpečená maximálna možná kompatibilita medzi verziou pre CI a verziou pre lokálny vývoj, premenné majú formu argumentov, ktoré je možné prekonať. Až pri zostavovaní dôjde k vytvoreniu výsledných ciest vo forme premenných prostredia, ktoré sú zložené z dvoch argumentov - zo systémových (predvolených) ciest a dodatočných ciest (špecifických pre konkrétne prostredie). Predvolená hodnota dodatočných ciest zodpovedá použitiu v prostredí CI. Pre lokálny vývoj je možné argumenty prekonať v `docker-compose.yml`.
Hodnoty obidvoch premenných je možné zmeniť v `Dockerfile`, pričom je možné zmeniť samostatne ako systémovú, tak aj dodatočnú časť. Systémové časti, t.j. argumenty LUA_PATH a LUA_CPATH by sa modifikovať nemali ! Na prispôsobenie podľa prostredia slúžia dodatočné cesty, t.j. argumenty LUA_PATH_ADD a LUA_CPATH_ADD, ktoré je možné modifikovať - prekonať - v súbore `docker-compose.yml`.

Pre použitie na lokálny vývoj existuje samostatný repozitár, devenv, ktorý obsahuje `docker-compose.yml` a všetky potrebné závislosti.

Pre správne nastavenie je odporúčané postupovať podľa príručky pre vývojára.

Premennou LUA_PATH je možné definovať cesty, na ktorých kompilátor hľadá Lua knižnice.
Premenná LUA_CPATH definuje, na akých cestách má kompilátor hľadať knižnice v jazyku C.

```
ARG LUA_PATH="./?.lua;/usr/local/share/lua/5.1/?.lua;/usr/local/share/lua/5.1/?/init.lua; \
/usr/local/lib/lua/?.lua;/usr/local/lib/lua/?/init.lua;/usr/local/lib/lua/5.1/?.lua; \
/usr/local/lib/lua/5.1/?/init.lua; /usr/share/lua/5.1/?.lua;/usr/share/lua/5.1/?/init.lua"
ARG LUA_CPATH="./?.so;/usr/local/lib/?.so;/usr/local/lib/lua/?.so;/usr/local/lib/lua/5.1/?.so; \
/usr/lib/x86_64-linux-gnu/lua/5.1/?.so;/usr/lib/lua/5.1/?.so;/usr/local/lib/lua/5.1/loadall.so"

# Setting additional LUA_PATH and LUA_CPATH for consistent usage in CI and docker-compose
ARG LUA_PATH_ADD="src/?.lua;src/?/init.lua;_install/lib/lua/?.lua;_install/lib/lua/?/init.lua"
ARG LUA_CPATH_ADD="_install/lib/?.so;_install/lib/lua/?.so"

# Setting ENV variables LUA_PATH and LUA_CPATH
# Values can be modified in docker-compose.yml
ENV LUA_PATH="${LUA_PATH};${LUA_PATH_ADD}"
ENV LUA_CPATH="${LUA_CPATH};${LUA_CPATH_ADD}"
```

Závislosti

`cmake` - nástroj pre multiplatformové zostavenie projektu - vytvára adresárovú štruktúru a pripravuje zdrojové súbory pre zostavenie kompilátormi príslušného operačného systému

`wget` - nástroj na prenos údajov prostredníctvom HTTP, HTTPS, FTP a FTPS

`unzip` - nástroj na rozbaľovanie .zip súborov

`git` - distribuovaný verziovací systém

`libz-dev` - komprimovacia knižnica

`luarocks` - správca balíkov pre programovací jazyk Lua

`make` - nástroj pre automatizované zostavenie vykonateľných programov alebo knižníc zo zdrojových kódov - vstupom je tzv. Makefile súbor, ktorý obsahuje pokyny pre vytvorenie výsledného programu

`build-essential` - informatívny zoznam balíkov potrebných na zostavovanie

`openssh-client` - secure-shell klient pre zabezpečený prístup k vzdialeným zariadeniam

`rsync` - nátroj na syschronizáciu a prenos súborov medzi vzdialenými systémami

`libzmq3-dev` - knižnica ZeroMQ pre IPC/RPC

Lua závislosti

Image obsahuje viaceré závislosti, ktoré sú potrebné pri vývoji, testovaní a nasadení projektu 3DSoftviz. Tieto závislosti sú vo fáze zostavovania image skompilované a nainštalované. Inštalované môžu byť vo všeobecnosti 3 spôsobmi :

Multigit

Luapower využíva na inštaláciu závislostí z luapower repozitárov wrapper multigit. Jedná sa o wrapper nad verziovacím systémom Git.
Jednotlivé moduly je teda možné inštalovať z adresára `/luapower/` príkazom `./mgit clone <url-prislusneho-luapower-repozitara>`.
Tieto závislosti sa sťahujú vopred zostavené. Tieto moduly sú následne skopírované do systémových adresárov. Lua moduly je potrebné kopírovať do adresára `/usr/local/lib/lua/` - v prípade ak adresár `lua` neexistuje, je potrebné vytvoriť ho pomocou príkazu `mkdir` - a moduly s príponou `.so` je potom potrebné kopírovať do adresára `/usr/local/lib/`.

Závislosti nainštalované cez Multigit

`luajit` - interpreter a kompilátor programovacieho jazyka Lua

`terra` - knižnica pre prácu s nízkoúrovňovým systémovým programovacím jazykom

`lua-headers` - Lua hlavičkové súbory

`cjson` - knižnica na serializáciu údajov vo formáte JSON

Manuálna kompilácia

Jednotlivé moduly sú naklonované z GitHub repozitárov do adresára `/_install`. Následne sú skompilované a nainštalované do systému. Klonovanie, kompilácia a inštalácia je vykonávaná nasledujúcimi príkazmi.

```
cd /_install; \
git clone https://github.com/LuaDist/lpeg.git && cd lpeg; \
git checkout 5019d49ce847a87662983acba1e35c9b252bb936; \
mkdir _build && cd _build; \
cmake .. && make install; \
```

Luarocks

Na inštaláciu Lua modulov je taktiež použitý správca balíkov Luarocks. Inštalácia je jednoduchá, pričom je možné explicitne špecifikovať požadovanú verziu modulu, napríklad príkazom `luarocks install luafilesystem 1.6.2`.

Nainštalované Lua závislosti

`busted` - nástroj pre testovanie Lua zdrojových súborov

`lpeg` - lua parser gramatických vzorov

`leg` - lua knižnica exportujúca kompletnú Lua 5.1 gramatiku

`luafilesystem` - knižnica na prácu so súborovým systémom

`luasocket` - knižnica poskytujúca podporu pre TCP a UDP

`lualogging` - knižnica pre logovanie

`luacomments` - gramatiky a parsery na parsovanie komentárov v Lua zdrojových kódoch

`ldoc` - nástroj na automatické generovanie dokumentácie z Lua zdrojového kódu

`luacov` - code coverage analyzátor

`luacheck` - statický analyzátor pre zdrojové kódy v jazyku Lua

`lrdb` - debugger pre jazyk Lua

`lzmq` - knižnica pre IPC/RPC, binding nad knižnicou ZeroMQ

`lua-messagepack` - knižnica na serializáciu údajov

Poznámky

Pre správnu funkciu Lua modulov je potrebné cez Multigit správne nainštalovať závislosti `luajit`, `terra`, `asdl`, `strict`, `terralib`, `terralib_luapower`, `terralist` a `lua-headers`. Tieto artefakty je následne potrebné kopírovať do systémových adresárov. Kopírovanie závislostí sa vykonáva pri zostavovaní image.

LRDB

Lua Remote Debugger umožňuje umožňuje vzdialené ladenie programov vytvorených v programovacom jazyku Lua. Debugger je potrebné použiť spolu s pluginom pre IDE Visual Studio Code. Tento modul je v projekte použitý kvôli lokálnemu vývoju. Je teda potrebné aby v Docker kontajneri bežal image so správne nastavenými cestami LUA_PATH a LUA_CPATH. Ďalej je potrebné podľa konfigurácie namapovať port kontajnera na port hostiteľského stroja. Lokálny vývoj je momentálne možný pomocou IDE Visual Studio Code, do ktorého je potrebné nainštalovať plugin vscode-lrdb.

Pri inštalácii pluginu je odporúčané postupovať podľa príručky pre vývojára. Projekt využíva vlastný fork, ktorý umožňuje použitie absolútnych ciest.

Doc extra

Tento image slúži pre dodatočné skripty pre generovanie dokumentácie. Obsahuje dva skripty, `snippet-extract.py` a `artifact-download.py`

snippet-extract.py

Skript slúži na generovanie/extrahovanie ukážok zo zdrojového kódu. Má dva argumenty, vstupný a výstupný priečinok.

Skript rekurzívne prehľadáva vstupný priečinok. Následne v každom nájdenom súbore hľadá komentáre `code_block start` a `code_block end` a obsah medzi nimi uloží do jednotlivých súborov. Novo vytvorené súbory zachovávajú pôvodnú adresárovú štruktúru a nachádzajú sa v priečinku `snippets`.

Za komentárom `code_block start` a `code_block end` sa vždy musí nachádzať tag. Tag musí byť rovnaký pre začiatočný aj ukončujúci komentár. Obsahovať môže len alfanumerické znaky, - a _.

Skript podporuje spracovanie súborov pre jazyky C++, C#, Lua, Python a Bash/Shell. Jazyky sú rozlíšené pomocou prípony súboru.

artifact-download.py

Týmto skriptom sa sťahujú artefakty z ostatných repozitárov s podporou záložných vetiev pre prípad, že artefakt na primárnej vetve neexistuje. Primárne sa používa na sťahovanie ukážok zdrojového kódu. Pre fungovanie potrebuje personal access token s oprávnením `read_api`.

Skript má viacero argumentov:

- `repository`
  názov/cesta k repozitáru (napr. `FIIT/Common/Lua/luameg`)
- `job`
  názov jobu generújúceho požadovaný artefakt
- `--branch/-b`
  názov vetvy z ktorej požadujeme artefakt
- `--fallback-branch`
  názov jednej alebo viacerých záložných vetiev
- `--extract-dir`
  názov cieľového adresára pre rozbalený obsah. Bez adresára sa uloží celý zip
- `--token`
  prístupový token

- `--soft-fail`
  príznak aby program skončil bez chybového kódu, ak sa nepodarí artefakt stiahnuť

3DSoftVis_Remake

Basic

Image slúži na zreťazené spracovanie, konkrétne generovanie a export dokumentácie zo zdrojového kódu prostredníctvom nástroja `Doxygen`. Image je založený na Ubuntu 18.04 a obsahuje nainštalované závislosti potrebné na vytvorenie dokumentácie a jej export na vzdialený server.

Závislosti

`build-essential` - informatívny zoznam balíkov potrebných na zostavovanie

`gnupg` - implementácia štandardu PGP na šifrovanie a podpisovanie údajov a komunikácií

`ca-certificates` - balík obsahujúci certifikačné autority, dodávané s prehliadačmi Mozilla - umožňuje kontrolovať autenticitu SSL spojení

`wget` - program na sťahovanie súborov prostredníctvom protokolov HTTP, HTTPS, FTP, FTPS

`gcc` - súbor kompilátorov projektu GNU (C/C++)

`cmake` - nástroj pre multiplatformové zostavenie projektu - vytvára adresárovú štruktúru a pripravuje zdrojové súbory pre zostavenie kompilátormi príslušného operačného systému

`make` - nástroj pre automatizované zostavenie vykonateľných programov alebo knižníc zo zdrojových kódov - vstupom je tzv. Makefile súbor, ktorý obsahuje pokyny pre vytvorenie výsledného programu

`openssh-client` - secure-shell klient pre zabezpečený prístup k vzdialeným zariadeniam

`git` - distribuovaný verziovací systém

`graphviz` - nástroj pre reprezentáciu štrukturálnych informácií vo forme diagramov

`doxygen` - nástroj na automatické generovanie dokumentácie zo zdrojového kódu

`rsync` - nástroj na syschronizáciu a prenos súborov medzi vzdialenými systémami

Mono

Image slúži na zreťazené spracovanie - zostavenie a následne testovanie projektu. Image je založený na Ubuntu 18.04 a obsahuje nainštalované závislosti potrebné na zostavenie a testovanie.

Závislosti

`mono-complete` - balík obsahuje kompilátor a runtime pre vývoj a spúšťanie aplikácií založených na technológiách Mono alebo Microsoft .NET

`mono-devel` - vývojové nástroje mono - kompilátor a dynamické knižnice, ktoré dokážu produkovať a vykonávať CIL (Commmon Intermediate Language) bajtkód a knižnice tried

`nuget` - správca balíkov pre NuGet repozitáre

Unity

Image slúži na zostavenie Unity časti projektu. Image je založený na Ubuntu 18.04 a obsahuje nainštalované závislosti potrebné na zostavenie pre platformy MS Windows, Linux a macOS.

Závislosti

`libgtk-3-0` - knižnica GUI GTK+

`libglu1-mesa` - pomocná knižnica OpenGL

`libnss3` - knižnice Network Security Systems - obsahuje implementácie kryptografických knižníc pre TLS/SSL a S/MIME

`libasound2` - zdieľaná knižnica pre aplikácie ALSA (Advanced Linux Sound Architecture) - zvuková architektúra pre Linux

`libgconf-2-4` - zdieľané knižnice - systém databázy konfigurácií na ukladanie nastavení aplikácií

`libcap2` - knižnica, ktorá implementuje rozhrania `capabilities` podľa štandardu POSIX 1003.1e, ktoré sú dostupné v linuxovom jadre - umožňujú rozčleniť oprávnenia `root-a` do množín jednotlivých oprávnení

`xvfb` - virtuálny framebuffer - falošný X server

`gconf-service` - podpora služby D-Bus, ktorý používa GConf na prístup ku konfiguračným údajom

`lib32gcc1` - knižnica interných podprogramov, ktoré využíva GCC na prekonanie nedostatkov alebo špeciálnych potrieb konkrétnych strojov

`lib32stdc++6` - GNU C++ knižnica (32-bit verzia)

`libasound2` - knižnica ALSA, jej štandardné zásuvné moduly a konfiguračné súbory

`libc6` - štandardné knižnice - napr. štandardná C knižnica, matematické kižnica a pod.

`libc6-i386` - 32-bitová verzia libc6 pre systémy AMD64

`libcairo2` - 2D vektorová grafická knižnica

`libcups2` - tlačový systém, všeobecná náhrada lpd; podpora Internet Printing Protocol (IPP)

`libdbus-1-3` - knižnica pre systém posielania správ medzi procesmi

`libexpat1` - knižnica jazyka C na syntaktickú analýzu XML

`libfontconfig1` - všeobecná knižnica na konfiguráciu písiem, ktorá nezávisí na X Window System

`libfreetype6` - systém na správu fontov

`libgcc1` - zdieľaná verzia podpornej knižnice vnútorných funkcií, ktoré GCC používa na odstránenie nedostatkov niektorých strojov alebo pre špeciálne potreby niektorých jazykov

`libgdk-pixbuf2.0-0` - knižnica umožňuje načítanie a ukladanie obrázkov, zmenu veľkosti a ukladanie pixbuf, načítanie jednoduchých animácií (GIF)

`libgl1-mesa-glx` - implementácia OpenGL API - GLX Runtime

`libglib2.0-0` - knižnica obsahujúca funkcie jazyka C pre stromy, hashovanie, zoznamy a reťazce

`libgtk2.0-0` - knižnica pre tvorbu GUI

`libnspr4` - knižnica poskytuje platformovo nezávislé možnosti OS (negrafické) - vlákna, synchronizácia vlákien, V/V bežných súborov a siete, časovanie intervalov a čas kalendára, základnú správu pamäte (malloc a free), linkovanie zdieľaných knižníc

`libpango1.0-0` - knižnica pre layoutovanie a renderovanie textu

`libstdc++6` - dodatočné dynamické knižnice pre programy v C++ zostavené kompilátorom GNU

`libx11-6` - balík poskytuje klientské rozhranie k X Window System inak známe ako „Xlib“
- poskytuje kompletné API základných funkcií systému okien

`libxcomposite1` - klientské rozhranie X Window System k rozšíreniu protoloku X Composite
- rozšírenie Composite umožňuje klientom zvaným kompozitní správcovia riadiť konečné vykreslenie obrazovky, pričom sa vykonáva do pamäte - mimo obrazovky

`libxcursor1` - knižnica určená na pomoc s nájdením a načítaním kurzorov pre X Window System
- kurzory možno načítať zo súborov alebo z pamäte a môžu existovať v niekoľkých veľkostiach
- knižnica automaticky vyberie najvhodnejšiu veľkosť
- pri použití obrázkov načítaných zo súborov preferuje Xcursor na vykreslenie použitie volania CreateCursor rozšírenia Render

`libxdamage1` - knižnica rozšírenia poškodených oblastí X11
- poskytuje klienta X Window System protokolu DAMAGE rozšírenia protokolu X
- upozorňuje, keď sa oblasti na obrazovke zmenia („poškodia“)

`libxext6` - knižnica rozličných rozšírení X11 - poskytuje klientské rozhranie X Window System k niekoľkým rozšíreniam protokolu X

`libxfixes3` - rozširujúca knižnica rozličných „opráv“ X11
- poskytuje klientské rozhranie X Window System k rozšíreniu protokolu X „XFIXES“
- podpora typov regiónov a niektorých funkcií kurzora

`libxi6` - knižnica rozšírenia X11 Input
- poskytuje klientské rozhranie X Window System k rozšíreniu XINPUT protokolu X
- rozšírenie Input umožňuje nastavenie viacerých vstupných zariadení a hotplugging (pripájanie a odstraňovanie zariadení počas behu)

`libxrandr2` - knižnica rozšírenia X11 RandR
- poskytuje klientské rozhranie X Window System k rozšíreniu RandR protokolu X
- rozšírenie RandR umožňuje konfiguráciu atribútov obrazovky ako rozlíšenie, otočenie a odraz za behu

`libxrender1` - klientská knižnica X Rendering Extension
- X Rendering Extension (Render) používa skladanie digitálneho obrazu ako základ nového vykresľovacieho modelu v rámci X Window System
- vykresľovanie geometrických obrazcov sa robí teseláciou na strane klienta na buď trojuholníky, alebo lichobežníky
- text sa vykresľuje načítaním grafém na server a vykreslením ich množín
- knižnica Xrender sprístupňuje toto rozšírenie X klientom

`libxtst6` - testovanie X11 - knižnica rozšírenia Record
- poskytuje klientské rozhranie X Window System rozšírenia X protokolu Record
- rozšírenie Record umožňuje X klientom syntetizovať vstupné udalosti, čo je užitočné na automatické testovanie

`zlib1g` - zlib je knižnica implementujúca komprimačnú metódu deflate, používanú v gzip a PKZIP

`debconf` - systém na správu konfigurácie

`npm` - správca balíkov pre javascriptovú platformu Node.js

`libsoup2.4-1` - implementácia knižnice HTTP v C
- možňuje aplikáciám GNOME prístup k HTTP serverom na sieti asynchrónnym spôsobom veľmi podobným programovaciemu modelu GTK+ (tiež je podporovaný aj synchrónny režim)

Testy

Testy sú napísané v C# a C++. Repozitáre 3dsoftvis_remake, LuaGraph a LuaInterface majú plne vybudovanú CI infraštruktúru. Táto infraštruktúra sa stará o to, aby sme vyvíjaný projekt mali vždy vo funkčnom stave a vždy bol pripravený na dodanie zákazníkovi. Infraštruktúra sa stará o build projektu a všetkých jeho variant, púšťaním automatizovaných jednotkových testov, inštrumentálnych testov, analyzovaním pokrytia kódu testami a zverejňovaním ich výsledkov na GitLabe a na stránke tímového projektu vo forme artefaktov. Taktiež analyzuje zdrojový kód a jeho kvalitu pomocou Lintu a ohlasuje možné sémantické problémy. Tiež sa stará o generovanie dokumentácie zo zdrojových kódov pomocou Doxygenu. Všetky vygenerované artefakty sú prístupné na webovej stránke tímu http://team07-18.studenti.fiit.stuba.sk/ci-artifacts/.

QA úroveň

Táto úroveň vykonáva testovanie a udržiavanie kvality a zabránenie regresii, teda aby po zmene kódu naďalej správne fungoval.

`run_tests` - Táto úloha testuje C# moduly

`run_lua_tests` - Táto úloha testuje Lua moduly

CMAKE

BUILD_SOFTVIZ_MODULES_TESTS

- Ak je hodnota `TRUE`, do buildu sa zahrnie build testov pre softviz moduly
- Predvolená hodnota: `FALSE`
- Výstupný adresár: `./_install/SoftvizModules.Tests/` Testy pre Softviz moduly

3DSoftviz/

- `CSProjects/` - Softviz moduly (C#) spolu so svojimi testami, tie sú automaticky pribalené do unity projektu a obsahujú celú logiku
- `./Assets/`
- `UnityTests` - Testy pre Unity projekt

Všeobecná adresarová štruktúra repozitára

- `Project/`
  - `./_install/` - obsahuje buildnuté artefakty
    - `Softviz/` - Plne funkčný softviz projekt
    - `SoftvizModules/` - Moduly pre Softviz projekt
    - `SoftvizModules.Tests/` - Testy pre Softviz moduly
  - `./cmake/` - Obsahuje cmake moduly

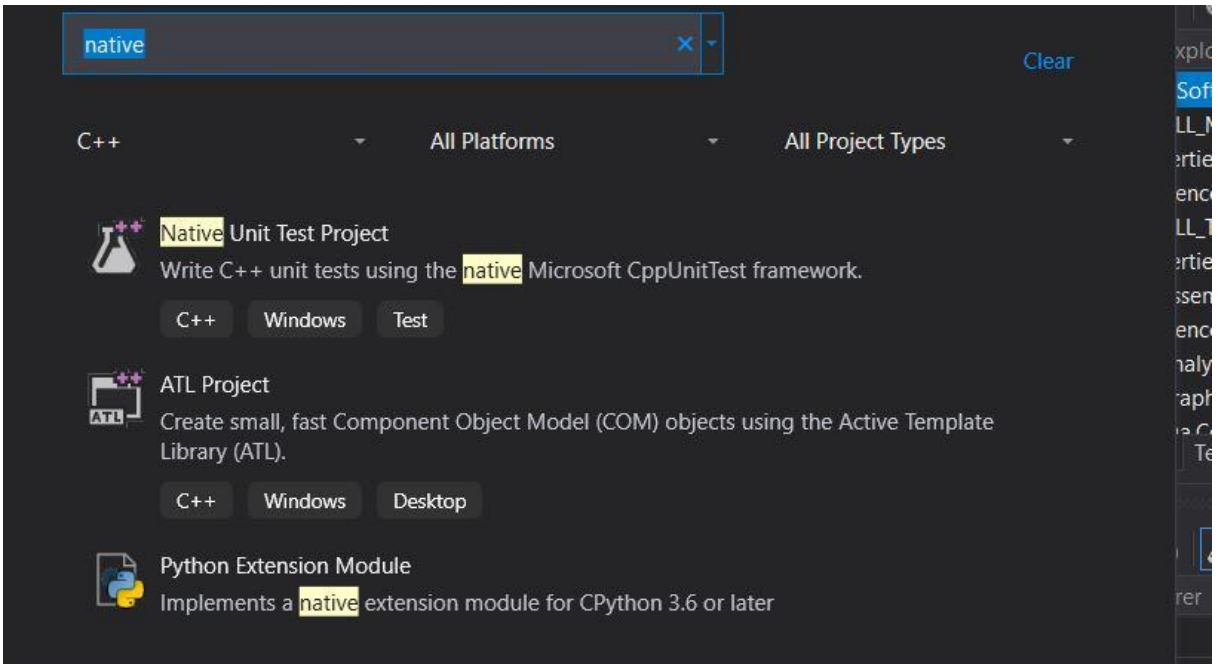
- `./Documentation/` - Dokumentácia k projektu
- `./Projects/` - Obsahuje jednotlivé projekty Softvizu
    - `LuaDependencies/` - Lua knižnice, ktoré sú pridané do repozitára ako git submoduly
    - `3DSoftviz/`
        - `CSProjects/` - Softviz moduly (C#) spolu so svojimi testami, tie sú automaticky pribalené do unity projektu a obsahujú celú logiku
        - `UnityProject/` - Unity projekt pre Softviz, slúži iba ako vizualizačná vrstva. Všetka ostatná logika patrí do Softviz modulov
- `./resources/` - Dodatočné súbory k projektom (spoločné scripty, obrázky, ...)

Adresarová štruktúra pre Unity projekt

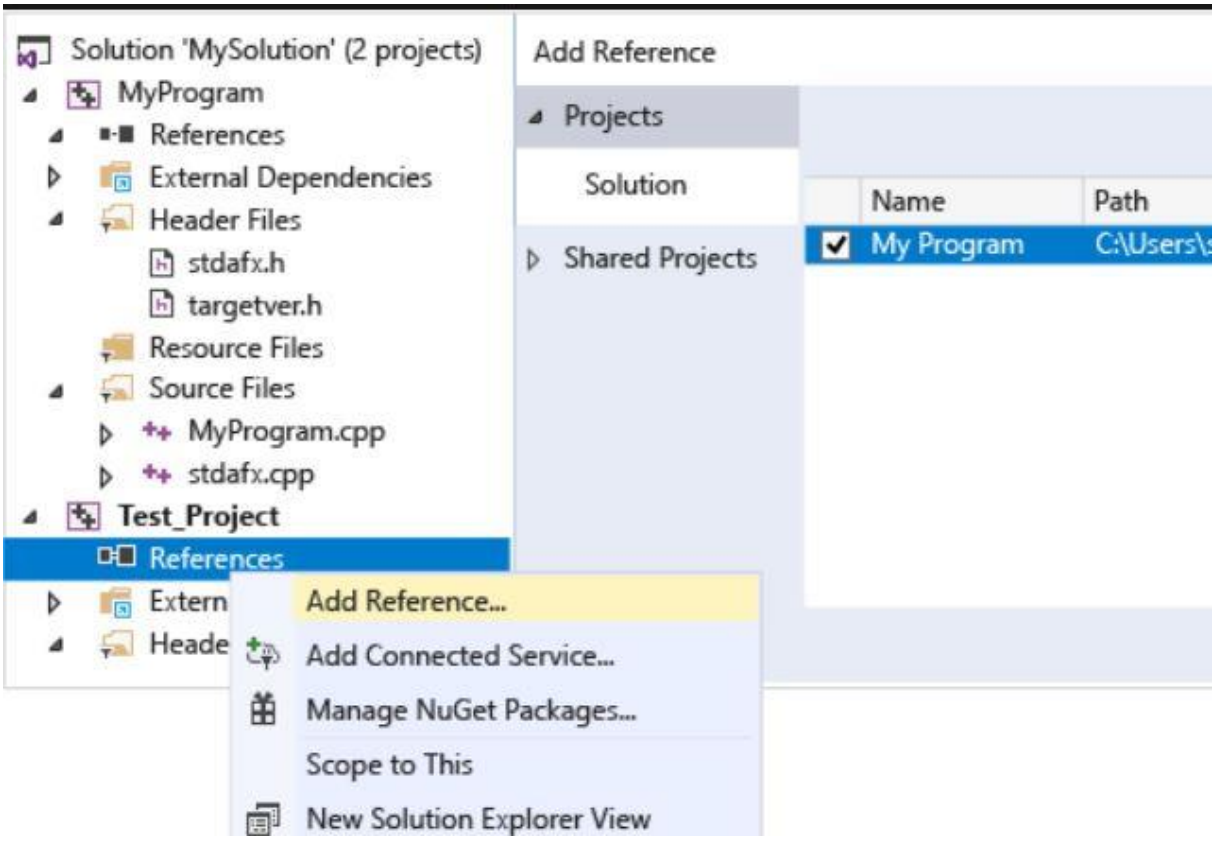
- `./Assets/`
    - `Animations` - Skripty pre animácie
    - `Editor` - Skripty pre Unity Editor
    - `External` - Externé assety z Assets Store
    - `Plugins` - Externé knižnice
        - `ExternalModule` - Externé C# moduly, napr. Softviz moduly
        - `Lua` - Natívne (C, C++) knižnice pre prácu s Lua
    - `Resources` - Zdroje pre Unity
      - `Prefabs`, `Textures`, `Materials`, `Fonts`, ...
    - `Scenes` - Unity scény
    - `Scripts` - Skripty pre Unity objekty,
    - `StreamingAssets` - Assety, ktoré nie sú pri builde pribalené do projektu, ale iba prekopírované k projektu
        - `LuaScripts` - Všetky lua scripty, ktoré by malo byt možné dynamicky meniť aj po skompilovaní unity projektu
    - `UnityTests` - Testy pre Unity projekt
- `./ProjectSetting`

C++ testy

Pre písanie nového testu je potrebné vytvoriť nový projekt v Solution. Ako nový projekt je potrebné Native Unit Test Project.



Pre povolenie prístupu testovacieho kódu vo funkciách v testovanom projekte je nutné pridať odkaz do projektu v testovacom projekte.



Je potrebné pridať `#include` pre všetky hlavičkové súbory, ktoré deklarujú typy a funkcie, ktoré chceme testovať.

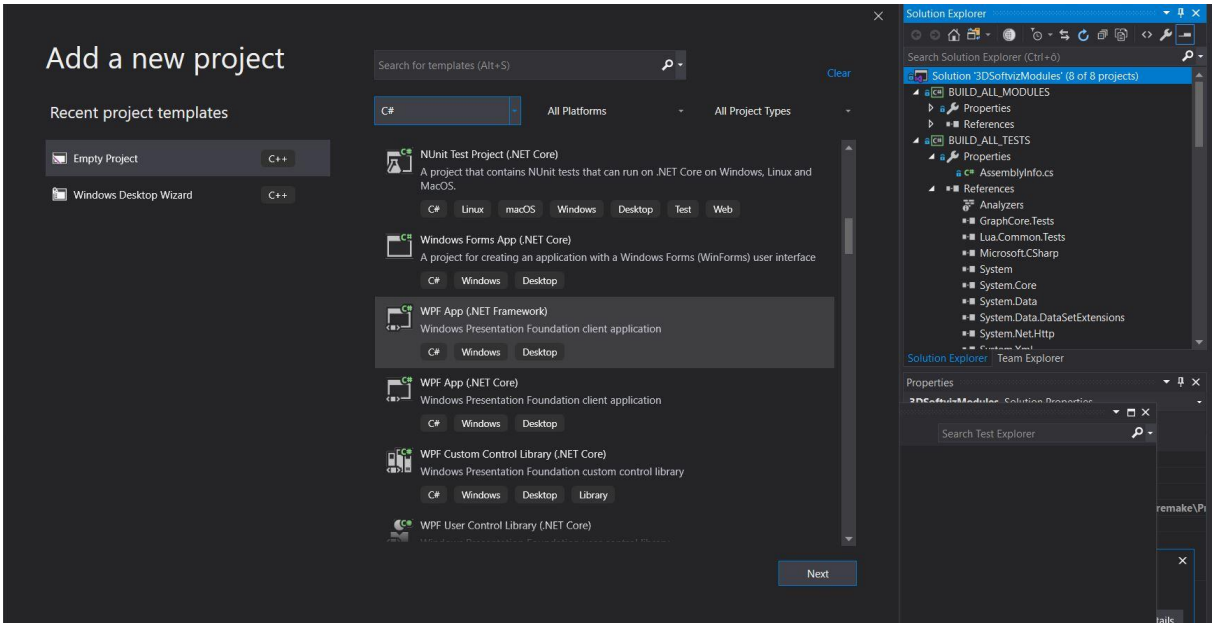
```cpp
#define CATCH_CONFIG_MAIN  // This tells Catch to provide a main() - only do this in one cpp file
#include "catch.hpp"
#include "LuaGraph/LuaGraph.h"
#include "LuaInterface/LuaInterface.h"
#include <string>
#include <fstream>

TEST_CASE( "Get nodes", "[getNodes]" )
{
    Lua::LuaInterface luaInterface;
    Lua::LuaGraph luaGraph( &luaInterface );

    //INSERT NODE
    auto nodes = luaGraph.getNodes();

    Lua::LuaNode* node = new Lua::LuaNode();
    node->setLabel( "myLabel" );
    nodes->insert( { 0, node } );

    //GET NODES
    nodes = luaGraph.getNodes();

    REQUIRE( nodes->at( 0 )->getLabel() == "myLabel" );
    REQUIRE( nodes->size() == ( std::size_t ) 1 );
}
```

Spustenie testov

Testy sa nachádzajú v repozitároch LuaInterface/tree/remake/tests a LuaGraph/tree/remake/tests v branchi remake, ktoré sú najaktuálnejšie. Testy sa púšťajú pomocou CI v stage QA.

C# testy

Pre písanie nového testu, je potrebné vytvoriť nový projekt v Solution. Ako nový projekt je potrebné NUnit Test Project (.NET Framework)



Následne je potrebné pridať potrebné namespaces.

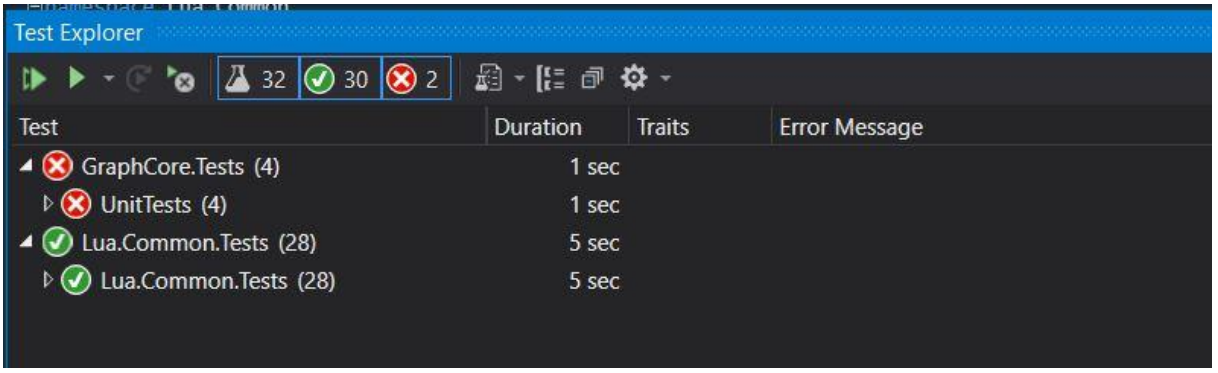
Príklad:

```
using Lua.Common.Graph;
using NUnit.Framework;
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
```

Pre testovanie triedy je potrebné používať atribút `[TestClass]` a pre testovanie metódy triedy je potrebné používať `[TestMethod]`.

```
01. using System;
02. using Microsoft.VisualStudio.TestTools.UnitTesting;
03. namespace BasicMathTest {
04.     [TestClass]
05.     public class UnitTest1 {
06.         [TestMethod]
07.         public void TestMethod1() {
08.             // TODO : Write test code
09.         }
10.     }
11. }
```

Metóda `Assert.AreEqual(x,y)` porovnáva či tieto dva vstupy sú rovnaké. Napríklad `x` je hodnota metódy triedy, ktorú testujeme a `y` správna hodnota, ktorú očakávame.



Spustenie testov

Pre spustenie testov je potrebné nainštalovať do Visual Studia komponent NUnit 3 Test Adapter. Následne je potrebné do priečinkov (`CSProjects\Lua.Common.Tests\bin\Debug`, `CSProjects\GraphCore.Tests\bin\Debug`) prekopírovať knižnice `lua51.dll`, `luagraph.dll`, `luainterface.dll`.

`AllNodesHasPosition` a `Graph_LoadGraph` aktuálne nezbehnú, ale vo výsledkoch predchádzajúcich testov, ktoré vykonávali predchadzajúce tímy zbehli úspešne.

Lua testy

V projekte pre Unit testing je použitý framework Busted.

Pre nastavenie testov sa používajú bloky `describe` a `it`. `before_each` definuje funkciu, ktorá sa má vykonať pred každým testovacím prípadom a `after_each` definuje funkciu, ktorá sa má vykonať po každom testovacom prípade.

Ukážka testu:

Nastavenie testovacieho modulu:

```
local utils, mapper, db, filter
local object1, object2, object3, object4
local adapter, testMapper, nameFilter


setup(function()
  -- specify connector to be used for testing
  db     = require "luadb.connection.redis.adapter"
  utils  = require "luadb.utils"
  utils.logger:setLevel(logging.FATAL)
end)
```

Preddefinovanie objektov:

```
before_each(function()

  object1 = {
    id    = "1",
    type  = "test object",
    label = "object 1 for test purpose",
    data  = { name = "Frank", age = '22' },
    tbl1  = { points = '10', structure = "nested" },
    tbl2  = { points = '20', structure = "nested", label = "test" }
  }

  object2 = {
    id    = "2",
    type  = "test object",
    label = "object 2 for test purpose",
    data  = { name = "George", age = '20' }
  }

  object3 = {
    id    = "3",
    type  = "test object",
    label = "object 3 for test purpose",
    data  = { name = "George", age = '18' }
  }

  object4 = {
    id    = "4",
    type  = "test object",
    label = "object 4 for test purpose",
    data  = { name = "Cavin", age = '12', email = "cavin@example.com" }
  }

  adapter:flush()
end)
```

Test Lua:

```
describe("Adapter", function()

  setup(function()
    adapter = db.new()
    adapter:connect()
  end)


  it("adapter functions presence", function ()
    assert.is.not_falsy(adapter.connect)
    assert.is.not_falsy(adapter.disconnect)
    assert.is.not_falsy(adapter.flush)
    assert.is.not_falsy(adapter.count)
    assert.is.not_falsy(adapter.add)
    assert.is.not_falsy(adapter.addMultiple)
    assert.is.not_falsy(adapter.set)
    assert.is.not_falsy(adapter.setMultiple)
    assert.is.not_falsy(adapter.delete)
    assert.is.not_falsy(adapter.deleteAll)
    assert.is.not_falsy(adapter.remove)
    assert.is.not_falsy(adapter.removeAll)
    assert.is.not_falsy(adapter.get)
    assert.is.not_falsy(adapter.getAll)
    assert.is.not_falsy(adapter.isMember)
    assert.is.not_falsy(adapter.members)
  end)


  it("adapter HASH functionality (set, get, delete, exists)", function ()
    assert.is_false(adapter:exists("key"))
    adapter:set("hash", "key", "value")
    assert.are.same("value", adapter:get("hash", "key"))
    assert.is_true(adapter:exists("hash", "key"))
    adapter:delete("hash", "key")
    assert.is_false(adapter:exists("hash", "key"))
  end)
```

Terra testy

Do funkcie sa zadeklaruje `local test = require("test")` Následne tento test testuje spravnosť pomocou eq. viď ukážku:

```
local test = require("test")

terra foo(a : double, b : double, c : double) : bool
    return a < b and b < c
end

test.eq(foo(1,2,3),true)
test.eq(foo(1,2,1),false)
test.eq(foo(2,1,2),false)
```

Podporné nástroje

Vývojárske prostredie

Vývojárske prostredie - devenv - umožňuje lokálny vývoj a ladenie aplikácií. Základom je použitie CI image. Aby bolo možné čo najviac sa priblížiť prostrediu CI, prostredie bolo prispôsobené potrebám lokálneho vývoja prostredníctvom prostriedkov `docker-compose`, pričom image zostal nezmenený oproti CI.

Riešenie je implementované vo forme kontajnerov, čo umožňuje ďalšie budúce rozšírenie. Súčasťou riešenia sú aj všetky potrebné závislosti, ktoré sú buď priamo obsiahnuté v repozitári, alebo vo forme Git submodulov. Momentálne je v riešení podporovaný lokálny vývoj modulov v programovacom jazyku Lua prostredníctvom kontajnera `luadev`.

Luadev

Luadev je kontajner, ktorý umožňuje lokálny vývoj, vrátane ladenia, testovania a code coverage modulov v programovacom jazyku Lua. Prostredie vychádza z Lua CI Image. Najzásadnejší rozdiel oproti prostrediu CI je mierna modifikácia ciest LUA_PATH a LUA_CPATH. Systémové cesty zostali zachované, rovnako ako aj celá štruktúra použitého CI image. K zmene došlo v cestách LUA_PATH_ADD a LUA_CPATH_ADD.

Image bol navrhnutý tak, aby tieto premenné bolo možné prekonať v závislosti od prostredia, v ktorom bude nasadený - CI alebo lokálny vývoj. Rovnako ako aj pri samotnom image platí, že systémové cesty, t.j. LUA_PATH a LUA_CPATH by sa nemali meniť ! Je ich možné a podľa prostredia vhodné mierne prispôsobiť argumentami LUA_PATH_ADD a LUA_CPATH_ADD.

```
args:
  LUA_PATH_ADD: "\
        src/?/init.lua;src/?.lua;\
        /luadev/luadb/src/?/init.lua;/luadev/luadb/src/?.lua;\
        /luadev/luameg/src/?/init.lua;/luadev/luameg/src/?.lua;\
        /luadev/luametrics/src/?/init.lua;/luadev/luametrics/src/?.lua;\
        /luadev/luagit/src/?/init.lua;/luadev/luagit/src/?.lua;\
        /luadev/luatree/src/?/init.lua;/luadev/luatree/src/?.lua"
  LUA_CPATH_ADD: ""
```

V tomto prípade nie je potrebné použiť špecifickú cestu LUA_CPATH, a tak má premenná LUA_CPATH_ADD hodnotu prázdneho reťazca.

Závislosti

Prostredie luadev má niekoľko závislostí, ktoré sú buď umiestnené priamo v repozitári, alebo majú formu Git submodulov.

Lua CI Image

- CI Image pre Lua moduly obsahujúci všetky závislosti na spustenie, ladenie, testovanie a code coverage,
  má formu Git submodulu
- Pokiaľ nie je image dostupný lokálne, bude zostavený pomocou `Dockerfile`, ktorý bude po inicializácii
  submodulov dostupný v koreňovom adresári `devenv`

vscode-lrdb

- Plugin pre IDE Visual Studio Code, ktorý umožňuje ladenie aplikácií v prostredí Docker kontajneru, vo vývojovom prostredí používame vlastný fork vscode-lrdb, ktorý umožňuje prácu s absolútnymi cestami, plugin je umiestnený priamo v repozitári devenv vo formáte vsix

Prerekvizity

Medzi prerekvizity patria :
- Visual Studio Code
- Docker
- Docker-compose

Moduly

Primárnym účelom prostredia luadev je lokálny vývoj Lua modulov, z tohto dôvodu sú jednotlivé moduly, ktoré sú vyvíjané v rámci projektu 3DSoftviz umiestnené v repozitári vo forme Git submodulov.
Príkazom `git submodule update --init --remote` dôjde k inicializácii a aktualizácii všetkých Git submodulov. Moduly sú následne umiestnené do adresára `luadev`, ktorý slúži ako pracovný adresár pri lokálnom vývoji.

Zväzky

Na prezistenciu údajov v prostredí luadev je použitý adresár `luadev`, ktorý je súčasťou repozitára. Do tohto adresára sú inicializované všetky vyvíjané moduly, v ktorých je následne možné vykonávať požadované zmeny, testovať ich alebo ladiť.
Adresár `devenv/luadev/` hostiteľského stroja je mapovaný na adresár `/luadev/` kontajnera. Tento adresár je zároveň nastavený ako pracovný adresár.

```
volumes:
  - ./luadev:/luadev
```

working_dir: /luadev

Porty

Kvôli správnej funkčnosti LRDB debuggera je potrebné správne mapovanie portu kontajnera na port hostiteľského stroja, cez ktorý bude prebiehať komunikácia medzi IDE Visual Studio Code a kontajnerom.
Port musí korešpondovať s konfiguráciou LRDB na hostiteľskom stroji, rovnako ako s portom použitým v main súbore určenom pre ladenie, ktorý by sa mal nachádzať v príslušnom repozitári. O konfigurácii je možné dozvedieť sa viac v príručke pre vývojárov alebo dokumentácii k Lua CI Image. Predvolený je port 21110.
Mapovanie portu 49155 je uvedené kvôli ukážke RPC medzi serverovou časťou vo forme kontajnera a klientskou časťou. Viac informácií o RPC je možné nájsť v dokumentácii možností komunikácie.

ports:
  - "21110:21110"
  - "49155:49155"

Spustenie

Pred prvým spustením je potrebná inštalácia pluginu vscode-lrdb do IDE Visual Studio Code. Taktiež môže byť potrebná jednoduchá konfigurácia pre konkrétny repozitár. Odporúčané je postupovať podľa príručky pre vývojára.

Poznámky

Rozdielom medzi prostredím CI a prostredím pre lokálny vývoj, ktorý súvisí s modifikovanými cestami LUA_PATH a LUA_CPATH, je nepoužitie nástoja `CMake`. Nakoľko vo vývojovom prostredí máme k dispozícii všetky potrebné moduly, ktoré môžu predstavovať navzájom závislosti, nie je potrebná ich inštalácia ako v prípade prostredia CI, kedy bol pre každý modul vytvorený adresár `_install` obsahujúci samotný modul a jeho závislosti. V prípade vývojového prostredia je toto správanie modifikované prostredíctvom modifikovanej cesty LUA_PATH_ADD. Jednotlivé moduly hľadá interpreter v adresári `src` príslušného modulu.
V prípade pridania ďalšieho modulu je potrebné doplniť cestu do argumentu LUA_PATH_ADD v súbore `docker-compose.yml`.

V súbore `docker-compose.yml` je definovaný ako preferovaný režim interaktívny, t.j. konzola operačného systému Linux Ubuntu , v ktorej je možné vykonávať požadované operácie.

stdin_open: true
tty: true

Mkdocs

MkDocs je rýchly, jednoduchý statický generátor stránok, ktorý je zameraný na generovanie projektovej dokumentácie. Zdrojové súbory dokumentácie sú zapísané v Markdown a konfigurované pomocou jedného konfiguračného súboru YAML.

MkDocs vytvára statické HTML stránky, ktoré môžu byť hostované na stránkach GitHub, Amazon S3 alebo kdekoľvek inde.

Pluginy

Do MkDocs je možné pridať ďalšie pluginy, pre rozšírenie funkcionality.

Macros

Plugin makrá nám umožňuje vytvoriť si vlastné funkcie a tie využívať pri písaní dokumentácie. Makrá sú zapísané v súbore `main.py` vo funkcii `define_env`, kde sú označené `@env.macro` tagom.

Aktuálne je definované jedno makro `code_snippet`. Toto slúži na dosadenie častí kódu do dokumentácie. Funkcia berie dva vstupné argumenty:

- `path` cesta k súboru, v ktorom sa nachádza kód, ktorý chcem vložiť. Cesta sa zapisuje od src/ adresára. Príklad z luagit repozitára: `src/luagit/repository.lua` zapíšem ako `luagit/repository.lua`.
- `tag` názov tagu. Ten musí byť unikátny vrámci jedného súboru.

Použitie makra

```
code_snippet("luagit/repository.lua", "executeOsCommand")
```

Makro sa vkladá do dvoch vnorených {}.

Pre zistenie presnej cesty k súbory je možné sa pozrieť do artefaktu pipeline. Príklad pre develop.

Pred vložením časti kódu pomocou makra do dokumentácie, je potrebné si najprv zvolenú časť kódu označiť v príslušnom súbore so zdrojovým kódom. Začiatok ukážky kódu je označený komentárom `code_block start <<názov tagu>>`, koniec ukážky je označený komentárom `code_block end <<názov tagu>>`. Začiatočný aj konečný názov tagu musí byť pre jednu ukážku kódu rovnaký. Vždy musí byť zapísaný začiatočný aj ukončujúci tag. Ukážky kódu môžu byť do seba vnorené alebo sa prekrývať.

Aby boli makrá funkčné aj pri lokálnom spustení mkdocs-u, je potrebné si stiahnuť artefakt s vygenerovanými snippetmi a umiestniť ho do priečinka s dokumentáciou.

Funkcionalita systému

Generovanie grafu
Lua v 3DSoftViz

Hlavným Lua súborom na interakciu s vyššími vrstvami a prácu s grafom je súbor `resources/scripts/app/main.lua`. V tomto súbore sa nachádza funkcia `loadGraph()`, ktorá je volaná z vyšších vrstiev, a slúži na načítanie grafu. Parametre funkcie môžeme vidieť nižšie. Ide prevažne o špecifikáciu grafu a konfiguráciu, v akej sa má graf načítať.

Vo funkcii `loadGraph()` si môžeme všimnúť vytvorenie dvoch objektov - `layoutFactory` a `graphHandler`, ktoré si popíšeme v ďalšej časti.

Funkcia `loadGraph()` pridá do poľa `loadedGraphs graphHandler` s príslušnou konfiguráciou.

```
function loadGraph(path, extractor, algorithm, buildingLayouter, functionMode, variableMode)
   local algorithm = algorithm
   local optionalSetup =
   {
      buildingLayouterType = buildingLayouter,
      functionMode = functionMode,
      variableMode = variableMode
   }
   local factory = layoutFactory.new(algorithm, optionalSetup)

   local graphHandler = graph_handler.new()
   if extractor == 'EvoGraph' then
      graphHandler:extractEvoGraph(graphManager:findFirstGraph(), graphManager:findLastGraph(), factory)
   else
      graphHandler:extractGraph(path, extractor, factory)
   end

   loadedGraphs[inc()] = graphHandler
end
```

`resources/scripts/module/softviz/graph_handler.lua` je modul, ktorý slúži na prácu s grafom pri načítavaní. Funkcia `extranctGraph()`, ktorú môžeme vidieť nižšie je volaná pri pridávaní `graphHandleru` do vyššie spomínaného poľa grafov. Táto funkcia na vstupe prijme cestu ku grafu, extraktor grafu a tzv. `layoutFactory`, ktorú si popíšeme nižšie. Toto sú všetko parametre konfigurácie, ktoré sa nastavujú v Unity časti a sem sa prešíria cez jednotlivé vrstvy pomocou volaní funkcií.

V tejto funkcii sa volá funkcia z modulu `Projects/LuaDependencies/luadb/src/luadb/extraction/extractor.lua`, v ktorej prebieha samotná extrakcia a načítavanie grafu. Nachádzajú sa tu funkcie, ktoré zistia názvy funkcií v načítavanom grafe, ich volania, volania modulov, globálne volania a rôzne iné parametre zdrojového kódu. Tieto následne poprepájajú do grafu.

Nakoniec je zavolaná funkcia `loadFromLuadbGraph()`, ktorú si popíšeme v ďalšej časti.

```
function pGraphHandler:extractGraph(absolutePath, graphPicker, layoutFactory)
  self.layoutFactory = layoutFactory
  local luadbGraph = {}

  utils.logger:setLevel(utils.logging.INFO)
```

```
 utils.logger:info("Started extration")
 if(graphPicker == "FunctionCallGraph") then
  luadbGraph = artifactsExtractor.extract(absolutePath, self.astMan)
 elseif(graphPicker == "ModuleGraph") then
  luadbGraph = moduleExtractor.extract(absolutePath, self.astMan)
 elseif(graphPicker == "MooscriptGraph") then
  luadbGraph = moonscriptExtractor.getGraphProject(absolutePath, self.astMan)
 else
  utils.logger:info("graphPicker set to unknown value")
 end

 self:loadFromLuadbGraph(luadbGraph)
end
```

Vo funkcii `loadFromLuadbGraph()` je volaná funkcia `addGraph()` z modulu `Projects/LuaDependencies/luadb/src/luadb/manager/graph/manager.lua`, ktorá slúži na jednoduchšie manažovanie grafu. Táto funkcia vytvorí z načítaného grafu dvojičku `graphId - graph`, ktorá je reprezentovaná objektom `GraphNode`.

Následne prebehne vo funkcii `initializeGraph()` inicializácia grafu, kde sa jednotlivým vrcholom a hranám nastavia názvy a iné parametre.

Ako posledná sa zavolá funkcia `layoutManager:initialize()` modulu `resources/scripts/module/layouter/layout_manager.lua`, ktorú si opíšeme nižšie.

```
function pGraphHandler:loadFromLuadbGraph(luadbGraph)
 graphManager:addGraph(luadbGraph)
 self.graph = luadbGraph
 self:initializeGraph()

 self.layoutManager = layoutManager.new(self.graph, self.layoutFactory)
 self.layoutManager:initialize()

 utils.logger:info("Extraction successfully finished")

 self.layoutManager:updateVisualMapping(luadbGraph.modified_nodes)
end
```

Poslednou dôležitou funkciou je spomínaná funkcia `initialize()` z modulu `resources/scripts/module/layout_manager.lua`, ktorá grafu nastaví parametre potrebné na správny beh programu. Nastavuje v prvom rade obmedzenia, ďalej nastavuje layoutovač na základe požadovaného layoutovacieho algoritmu a aktualizuje uzly grafu.

```
function pLayoutManager:initialize()
   utils.logger:info("LayoutManager::Initialize")

   if self.currentAlgorithm ~= nil then
      self.currentAlgorithm:terminate()
      self.currentAlgorithm = nil
   end

   if(self.restrictionManager == nil) then
      self.restrictionManager = RestrictionManager.new()
   end


   self.currentAlgorithm = self.layoutFactory:getLayouter(self.graph, self.restrictionManager)
   self.currentAlgorithm:initialize()

   self:updateNodes()

   self.restrictionManager:createRestriction(DefaultRestriction, restrictions.Default())
   self:setRestrictionToAllNodes(DefaultRestriction)
end
```

Po vykonaní týchto funkcií sú v poli `loadedGraphs` v súbore `main.lua` načítané grafy, ku ktorým sa dá pristupovať cez `id` grafu. Po tejto prvotnej inicializácii vedia vyššie vrstvy projektu pristupovať ku poľu a vykonávať nad ním operácie, ako napríklad `getGraph()`, `getLayoutManager()`, `setNodeColor()`, a iné.

Unity

Celá logika načítavania grafu v Unity sa začína v triede `GraphLoader`. Tento objekt sa nachádza v scéne v Unity a pri načítaní scény, sa podobne ako pri ostatných objektoch, ktoré dedia od triedy MonoBehaviour, zavolá funkcia `Awake()`. V tejto metóde sa najprv do listu konfigurácií, zapíše jedna alebo viacero konfigurácií, ktoré zabezpečujú aký graf sa načíta, kde je uložený, aký algoritmus sa použije a ďalšie iné nastaviteľné parametre.

```
Configurations.Add(new GraphConfiguration
{
   Algorithm = LayoutAlgorithmsEnum.FruchtermanReingold,
   Path = $"{Application.streamingAssetsPath}/LuaScripts/Modules/say",
   GraphExtractor = GraphExtractor.ModuleGraph,
   OptionalParameters = new OptionalParameters
   {
      BuildingLayout = BuildingLayoutAlgorithmEnum.RowAlgorithm,
      FunctionType = FunctionTypeEnum.Cyclomatic,
      VariableType = VariableTypeEnum.None
   }
});
```

Ďalej sa tu zavolá funkcia `InvokeRepeating()`, ktorá zabezpečuje aktualizovanie grafu každých X sekúnd. Tu sa pošlú tri parametre. Prvým je názov funkcie, ktorá sa má opakovať, druhým je čas, kedy sa funkcia prvý krát zavolá a tretím je časový interval medzi jednotlivými volaniami. V našom prípade funkcia, ktorá sa bude volať opakovane, nesie názov UpdateGraph a taktiež sa nachádza v triede `GraphLoader`.

```
InvokeRepeating(nameof(GraphLoader.UpdateGraph),
            GraphLoader.UpdateTimeInitialDelayTime,
            GraphLoader.UpdateTimeRepeatingTime);
```

Samotné načítanie grafu sa uskutoční zavolaním funckie LoadAllGraphs na objekte `GraphManager`. `GraphManager` je trieda, ktorá sa nachádza v module `GraphCore` a ako parameter sa tu pošle zoznam s konfiguráciami.

```
GraphManager.LoadAllGraphs(Configurations);
```

V tejto funkcii sa postupne prejdú jednotlivé konfigurácie a prostredníctvom `LuaInterface.Instance.DoString()` sa zavolá metóda `loadGraph()` s konkrétnou konfiguráciou. Následne sa na novovytvorenom objekte `LuaGraph` zavolá funkcia `LoadGraph()`, ktorá na vstupe prijme index grafu.

```
public void LoadAllGraphs(List<GraphConfiguration> configurations)
{
   for (var i = 0; i < configurations.Count; ++i)
   {
      var configuration = configurations[i];
      LuaInterface.Instance.DoString($"loadGraph({configuration.ToLuaTable()})");

      var luaGraph = new LuaGraph();
      luaGraph.LoadGraph(i);
      Graphs.Add(new Graph(luaGraph, configuration));
   }
}
```

V tejto funkcii sa zavolá metóda `LuaGraphLoadGraph()`, ktorá vyvolá metódu `loadGraph()` v module `LuaGraph`. Následne sa tu volajú funkcie `LoadNodes()`, `LoadEdges()` a `LoadIncidences()`, ktoré do príslušných listov načítajú objekty z Lua a namapujú ich na tie, s ktorými sa pracuje v rámci C#. V týchto funkciách sa každému objektu priradí iba `Id` a `Label`.

```
public void LoadGraph(int graphId)
{
   GraphId = graphId;
   LuaGraphLoadGraph(NativeObject, graphId);

   Nodes = LoadNodes();
   Edges = LoadEdges();
   Incidences = LoadIncidences();
}
protected IList<LuaNode> LoadNodes()
{
   return MarshallingUtils.ReadArray<LuaNodeDTO, LuaNode>(
            LuaGraphGetNodesSize(NativeObject),
            a => LuaGraphGetNodes(NativeObject, a),
            a => MapNode(a))
            .ToList();
}
private unsafe LuaNode MapNode(IntPtr node)
{
   var nodeDTO = *(LuaNodeDTO*)node;
   return new LuaNode(nodeDTO.NativeLuaNode)
   {
      Id = nodeDTO.Id,
      Label = MarshallingUtils.ReadString(nodeDTO.Label) ?? UnknownLabel
   };
}
```

Následne sa načítaný objekt `LuaGraph` pretransformuje na objekt `Graph` a ten sa pridá do listu grafov. V konštruktore objektu `Graph` sa nainicializujú potrebné hodnoty a následne sa zavolá funkcia `InitializeGraph()`.

```
public Graph(LuaGraph luaGraph, GraphConfiguration configuration)
{
   LuaGraph = luaGraph;
   LayoutManager = new LayoutManager(LuaGraph.GraphId);
   LayoutFactory = new LayoutFactory(LuaGraph.GraphId);
   Nodes = new ObservableCollection<Node>();
   Edges = new ObservableCollection<Edge>();
   luaNodesMapping = new List<Tuple<LuaNode, Node>>();
   luaEdgesMapping = new List<Tuple<LuaEdge, Edge>>();
   CurrentConfiguration = configuration;

   InitializeGraph();
}
```

V tejto funkcii sa opäť pretransformujú jednotlivé LuaObjekty na tie, potrebné pre ďalšiu prácu. `LuaNode` -> `Node`, `LuaEdge` -> `Edge`. List s LuaIndices sa nevyužíva.

```
protected void InitializeGraph()
{
   var luaNodes = LuaGraph.Nodes;
   var luaEdges = LuaGraph.Edges;
   var luaIncidences = LuaGraph.Incidences;

   foreach (var luaNode in luaNodes)
   {
      var node = GraphObjectFactory.CreateNodeFromLuaNode(luaNode);

      if (!IsFiltered(luaNode))
      {
         Nodes.Add(node);
```

```
        }

        luaNodesMapping.Add((luaNode, node).ToTuple());
    }

    foreach (var luaEdge in luaEdges)
    {
        var sourceNodeId = luaEdge.Data.SourceId;
        var destinationNodeId = luaEdge.Data.DestinationId;

        var edge = GraphObjectFactory.CreateEdgeFromLuaEdge(
            luaEdge,
            AllCreatedNodes.First(node => node.Id == sourceNodeId),
            AllCreatedNodes.First(node => node.Id == destinationNodeId));

        AllCreatedEdges.Add(edge);

        if (!edge.IsFiltered && IsVisible(luaEdge))
        {
            Edges.Add(edge);
        }

        luaEdgesMapping.Add((luaEdge, edge).ToTuple());
    }

    RecreateHierarchy();
}
```

V nasledujúcich funkciách je možné vidieť, aké dáta si jednotlivé objekty uchovávajú.

```
public static Node CreateNodeFromLuaNode(LuaNode luaNode)
{
    return new Node
    {
        LuaNode = luaNode,
        Id = luaNode.Id,
        Label = luaNode.Label,
        Type = luaNode.RawData.GetString("type"),
        Position = luaNode.Data.Position,
        Size = luaNode.Data.Size,
        IsFiltered = (int)luaNode.RawData.GetFloat("filtered.value", true) == 1, // Fix this
        Color = luaNode.Data.Color,
        Shape = luaNode.Data.Shape,
        Flag = luaNode.Data.Flag,
        IsFixed = luaNode.Data.Layouter.GetBool("isFixed"),
        EvolutionData = luaNode.Data.Evolution,
    };
}

public static Edge CreateEdgeFromLuaEdge(LuaEdge luaEdge, Node sourceNode, Node destinationNode)
{
    return new Edge
    {
        LuaEdge = luaEdge,
        Id = luaEdge.Id,
        SourceNode = sourceNode,
        DestinationNode = destinationNode,
        Label = luaEdge.Label,
        Flag = luaEdge.Data.Flag,
        Color = luaEdge.Data.Color,
        IsFiltered = sourceNode.IsFiltered || destinationNode.IsFiltered,
    };
}
```

Poslednou metódou, ktorá sa volá vo funkcii `InitializeGraph()` je metóda `RecreateHierarchy()`, ktorá zabezpečuje to, aby sa podľa jednotlivých hrán vytvorili dvojice uzlov, ktoré tieto hranu spájajú.

```
private void RecreateHierarchy()
{
```

```
    Hierarchy = new List<Tuple<Node, Node>>();
    foreach (var edge in LuaGraph.Edges)
    {
        if (IsParent(edge) && !IsFiltered(edge))
        {
            var sourceNode = Nodes.FirstOrDefault(x => x.Id == edge.Data.SourceId);
            var destinatioNode = Nodes.FirstOrDefault(x => x.Id == edge.Data.DestinationId);
            Hierarchy.Add(new Tuple<Node, Node>(sourceNode, destinatioNode));
        }
    }
}
```

Týmto sme prebehli celú `GraphCore` vrstvu a môžeme sa vrátiť do funkcie `Awake()` v triede `GraphLoader`. Tu nám ostala ešte jedna funkcia a to `CreateGraphs()`. Tá zabezpečuje vytvorenie jednotlivých objektov z príslušného prefabu pre graf. Každý takýto objekt musí obsahovať komponent `GraphUnity`. Nad týmto komponentom sa zavolá metóda `InitializeGraph()`, ktorá inštancuje všetky viditeľné uzly a hrany podľa príslušných prefabov prostredníctvom metód `CreateNode()` a `CreateEdge()`.

```
public void InitializeGraph(CoreGraph graph)
{
    BaseGraph = graph;
    nodes = graph.Nodes;
    edges = graph.Edges;

    if (nodes != null)
    {
        nodes.CollectionChanged += OnNodesChanged;
        foreach (var node in nodes)
        {
            if (!node.IsFiltered)
                CreateNode(node);
        }
    }

    if (edges != null)
    {
        edges.CollectionChanged += OnEdgesChanged;
        foreach (var edge in edges)
        {
            if (!edge.IsFiltered)
                CreateEdge(edge);
        }
    }

    if (graph.Hierarchy != null)
    {
        foreach (var link in graph.Hierarchy)
        {
            var source = nodeScripts.FirstOrDefault(a => a.Id == link.Item1.Id);
            var destination = nodeScripts.FirstOrDefault(a => a.Id == link.Item2.Id);

            destination.gameObject.transform.SetParent(source.gameObject.transform, true);
            destination.gameObject.transform.localPosition = new Vector3(destination.BaseObject.Position.X,
destination.BaseObject.Position.Y, destination.BaseObject.Position.Z);
            destination.gameObject.transform.localScale = new Vector3(1, 1, 1);
        }
    }
}

private GameObject CreateNode(Node node)
{
    GameObject nodePrefab = (GameObject)Instantiate(Resources.Load(GraphUnity.NodePrefabPath), gameObject.transform);
    var nodeScript = nodePrefab.GetComponent<NodeUnity>();
    nodeScript.BaseObject = node;
    nodeScript.Mesh = NodeMesh;
    nodeScript.tag = GameObjectTags.Node;
    nodeScripts.Add(nodeScript);
    nodePrefab.transform.localPosition = new Vector3(node.Position.X, node.Position.Y, node.Position.Z);
    node.PropertyChanged += nodeScript.OnNodeChanged;
```

```
    return nodePrefab;
  }

  private void CreateEdge(Edge edge)
  {
    var source = nodeScripts.FirstOrDefault(a => a.Id == edge.SourceNode.Id);
    var destination = nodeScripts.FirstOrDefault(a => a.Id == edge.DestinationNode.Id);

    GameObject edgePrefab = (GameObject)Instantiate(Resources.Load(GraphUnity.EdgePrefabPath), gameObject.transform);
    var edgeScript = edgePrefab.GetComponent<EdgeUnity>();
    edgeScript.BaseObject = edge;
    edgeScript.Mesh = EdgeMesh;
    edgeScript.SourceNode = source;
    edgeScript.DestinationNode = destination;

    edgeScripts.Add(edgeScript);
  }
```

LuaDB

Logika priamo súvisejúca s tvorbou grafu je rozdelená v súboroch nachádzajúcich sa v dvoch priečinkoch a to *src/luadb/extraction/** a *src/luadb/hypergraph/**. Samotné hrany, uzly, ich výskyty a graf sú definované v jednlotlivých .lua súboroch v priečinku *src/luadb/hypergraph/* , ktoré nesú rovnaké názvy ako ako to, čo definujú. (tj. edge, node, incidence a graph) Sú v nich definované nielen samotné objekty a ich konštruktory, ale i základné operácie priamo súvisejúce s nimi, ako napríklad funkcie na pridanie hrán, uzlov a podobne. Napríklad kód nižšie je možné zavolať nad hranou a funkcia vracia informáciu o tom, či ide o orientovanú hranu.

```
function pEdge:isOriented()
  return self.orientation and self.orientation == "Oriented" and true or false;
end
```

Okrem toho obsahujú funkcie aj na vyhľadanie daných objektov podľa zadaného argumentu, ktorým sú napríklad meno alebo typ. Väčšina takýchto funkcií sa nachádza v súbore *graph.lua*. Napríklad funkcia nižšie zavolaná nad určitým grafom s argumentom *typ* vráti všetky uzly v danom grafe daného typu.

```
 -- get all nodes with selected type
function pGraph:findNodesByType(type)
  local occurrences = {}
  for i,node in pairs(self.nodes) do
if node.meta.type and (node.meta.type == type) then
    table.insert(occurrences, node)
  end
  end
return occurrences
end
```

V priečinku *src/luadb/extraction* sa extrahuje graf z lua súborov a to následovne.

1. Najskôr je nutné získať lua súbory zo zadanej cesty. Táto funkcionalita sa nachádza v súbore *luadb/extraction/filestree/extractor.lua*.
    a. Najskôr sa načíta cesta, skontroluje sa, či je daná cesta string a z nej sa následne vystrihne posledná časť - meno priečinku.
    b. Ak ide o úplne prvé volanie funkcie, vytvorý sa root uzol.
    c. V ďalšej funkcii sa cyklom rekurzívne prehľadávajú súbory daného priečinka, pričom ak súbor v priečinku je priečinok, tak sa do

množiny lua-uzlov v grafe pridá uzol typu priečinok (directory) a prehľadávanie sa posúva hlbšie

d. Ak súbor nebol priečinok tak sa ešte overí, či ide o *.lua* súbor a ak áno pridá sa do množiny lua-uzlov v grafe s typom súbor (file).

e. Ako posledné sa vytvorí hrana medzi súborom a jeho rodičom. Pri prvoúrovňovom volaní je to novovytvorený root, čiže rodičovký repozitár. V prípade, že už bola zavolaná funkcia rekurzívne, ako argument jej bol poslaný aktuálne iterovaný uzol (súbor), tak je hrane priradený ako rodič repozitár s ktorým bola funkcia volaná.

f. V poslednom kroku sa do grafu pridá uzol a jeho hrana.

```
    local function extractFilesTree(graph, path, parent)
      -- root node
if not parent then
  local newNode = hypergraph.node.new()
       newNode.data.name = getNameFromPath(path, "/")
       newNode.data.path = path
       newNode.meta = newNode.meta or {}
       newNode.meta.type = "directory"
       graph:addNode(newNode)
       parent = newNode
      end

for file in lfs.dir(path) do
       if file ~= "." and file ~= ".." and not utils.isHidden(file) then
         local fullPath = path..'/'..file
         local attr = lfs.attributes (fullPath)
         assert (type(attr) == "table")

         -- new node
         local newNode = hypergraph.node.new()
         newNode.meta = newNode.meta or {}
   if attr.mode == "directory" then
           newNode.data.name = file
           newNode.data.path = fullPath
           newNode.meta.type = "directory"
           -- recursive call
     extractFilesTree(graph, fullPath, newNode)
         else
           newNode.data.name = file
           newNode.data.path = fullPath
           newNode.meta.type = "file"
           graph.luaFileNodes = graph.luaFileNodes or {}
     if utils.isLuaFile(file) then table.insert(graph.luaFileNodes,newNode) end
         end

         -- new edge
   local newEdge = hypergraph.edge.new()
   newEdge.label = "Subfile"
   newEdge:addSource(parent)
   newEdge:addTarget(newNode)
   newEdge:setAsOriented()

   graph:addNode(newNode)
   graph:addEdge(newEdge)
      end
    end
   end
```

Súbor *luadb/extraction/functioncalls/extractor.lua* obsahuje funkcie, ktoré extrahujú funkcie a volania, pričom ich delia na globálne a modulové.

1. Ako prvé sa získajú všetky funkcie z daného stromu, ktoré sa potom v cykle prejdú a pre každú funkciu sa vytvorí uzol s typom funkcia a uložia sa takisto informácie ako meno, či cesta.

2. Následne je nutné vytvoriť hrany, ktoré budú tieto uzly spájať.
    a. Tu sa sa získajú všetky volania funkcií z abstraktného syntaktického stromu
    b. Listuje sa medzi volaniami, pričom každá funkcia má (môže mať) viacero volaní, ktoré je tiež nutné prelistovať
    c. Pre každné volanie funkcie sa vytvorí hrana, pričom do zdroja a cieľu sa priradia uzly, ktoré sa sa zistia pomocou metódy uzlu *(hypergraph/node.lua) findByName*, ktorá vracia tabuľku uzlov. (v tomto prípdade uzlov funkcii, kedže tie sa do nej posielajú)
    d. Ak hrana (volanie) nevolá nijakú funkciu, tak je roztriedená podľa toho, či je globálna alebo modulová do príslušných polí v grafe. (graph.moduleCalls alebo graph.globalCalls)
    e. Nová hrana sa nastaví ako orientovaná a je pridaná do grafu.
    f. Funkcia extract, ktorá volá zvyšné funkcie, vracia hrany(volania funkcií) a uzly(funkcie).

```
    local function extractFunctionCalls(AST, graph, nodes)
      local edges = {}
      local functions = ast.getFunctions(AST)
local functionsCalls = ast.getFunctionsCalls(AST)

for calledFunction,functionCalls in pairs(functionsCalls) do
  for index,call in pairs(functionCalls) do
        local newEdge = hypergraph.edge.new()
        newEdge.label = "FunctionCall"
        newEdge.meta  = newEdge.meta or {}
        newEdge.meta.calleeFunction = getCalleeFunctionName(call)
        newEdge.meta.calledFunction = calledFunction
        newEdge.data.text = call.text
        newEdge.data.position = call.position
        newEdge.data.tag = call.tag
   newEdge:addSource(hypergraph.node.findByName(nodes, newEdge.meta.calleeFunction))
   newEdge:addTarget(hypergraph.node.findByName(nodes, calledFunction))

        -- function's declaration not found
   if utils.isEmpty(newEdge.to) then
         logger:debug('found undeclared function '..calledFunction)
    addGlobalCall(graph, newEdge, call, calledFunction)
        end

   newEdge:setAsOriented()
   graph:addEdge(newEdge)
        table.insert(edges, newEdge)
      end
    end
    return edges
   end
```

Tretím súborom, ktorý rieši extrakciu uzlov a hrán v luaDB module je súbor priamo v priečinku *luadb/extraction* a to *extraction.lua*.

1. Ako prvá sa volá funkcia getFilesTree(graph, sourceDirectory) zo súboru *luadb/extraction/filestree/extractor.lua* , ktorá je popísaná v prvej časti dokumentu
2. V tejto chvíli máme uzly typu file, ktoré potrebujeme prehľadať a získať z nich uzly typu function a to za pomoci funkcie getFunctionCalls(graph, luaFileNodes).
    a. V tejto funkcii sa listujú všetky súborové uzly

b. Pre každý uzol sa zavolá funkcionalita zo súboru *src/luadb/extraction/functioncalls/extractor.lua* popísanom vyššie, takže v tomto bode už máme aj funkcie a volania funkcií.

c. Pre každý uzol priradíme do functionNodes uzly funkcií a do functionCalls hrany volaní funkcií získané v kroku vyššie.

d. Následne každý uzol funkcie napojím novovytvorenou hranou na príslušný uzol súboru (takto viem, ktoré funkcie patrie ktorému súboru)

```
e.  for j,functionNode in pairs(luaFileNode.functionNodes) do
f.    local connection = hypergraph.edge.new()
g.    connection.label = "FunctionDeclaration"
h.    connection:addSource(luaFileNode)
i.    connection:addTarget(functionNode)
j.    graph:addEdge(connection)
    end
```

k. Takisto preiterujem aj všetky volania funkcií a ak majú nastavený zdroj tak túto hodnotu nahradíme uzlom aktuálneho súboru. Predtým bol nastavený pomocou metódy *findByName* a teda nabral hodnotu tabuľky uzlov funkcií, teda funkcia, ktorá tento uzol (funkciu) volala pochádzala z tohto súboru.

```
l.  for k,functionCallEdge in pairs(luaFileNode.functionCalls) do
m.    if utils.isEmpty(functionCallEdge.from) then
n.    functionCallEdge.label = "FunctionCall"
o.    functionCallEdge:addSource(luaFileNode)
p.    end
    end
```

Ako bolo spomínané pri opise súboru *src/luadb/extraction/functioncalls/extractor.lua* volania funkcií, ktoré nemali cieľovú funkciu, (nemali target/to), čiže nevolali inú funkciu, boli roztriedené do dvoch skupín na globálne a modulové. V tomto súbore sa napoja na graf aj tieto hrany a to vo funkciách connectModuleCalls(graph) a assignGlobalCalls(graph).

connectModuleCalls(graph)

V tejto funkcií sa napoja na graf modulové funkcie.

1. Prelistujú sa všetky modulové volania pričom pre všetky volania sa :
    a. získa názov modulu a cesta k nemu
    b. prelistujú všetky súborové uzly a hľadá sa taký, ktorý má rovnakú cestu ako je cesta k modulu a zároveň či vo funkciách daného súborového uzlu existuje funkcia, ktorá má rovnaký názov ako dané modulové volania (teda sa pozrie, či a, existuje súbor z ktorého bola funkcia volaná, b, či v tom súbore existuje funkcia, ktorá bola volaná)
    c. ak áno sú tieto uzly funkcii pridané do pomocného poľa

```
d.  for i,luaFileNode in pairs(luaFileNodes) do
e.    if luaFileNode.data.path:find(modulePath) and
    getFunctionWithName(luaFileNode.functionNodes, moduleFunctionCall) ~= nil then
f.      functionNodes = luaFileNode.functionNodes
g.    end
    end
```

    h. Ak takáto funkcia (uzol) neexistuje ani jedna, tak sa zaregistruje globálne volanie, funkcionalita bude popísaná nižšie.

```
i.  if utils.isEmpty(functionNodes) then
j.    registerGlobalModule(graph, moduleName, moduleFunctionCall)
k.    functionNodes = getGlobalModuleFunctions(graph, moduleName)
    end
```

l. Nakoniec sa preiterujú všetky hrany volania a priradia sa k cieľovému uzlu.

```
m.  for j,moduleFunctionCallEdge in pairs(edges) do
n.      -- add target connection for each module function call
o.      moduleFunctionCallEdge:addTarget(functionNode)
    end
```

Tu je možno na mieste si pripomenúť, ako vyzerá jedna položka v tejto iterácii, teda ako vyzerá "key => value" a to tak, že znovu zobrazíme riadok, kde sa do teraz iterovaných položiek pridával nový prvok. `table.insert(graph.moduleCalls[calledFunction], newEdge)` *calledFunction* je v tomto prípade meno funkcie, ktoré je kľúčom v poli *moduleCalls*.

assignGlobalCalls(graph)

V tejto funkcií sa napoja na graf globálne funkcie.

1. Preiterujú sa všetky globálne volania(hrany), pričom pre každé sa:
    a. skontroluje, či po rozdelení `utils.split(globalFunctionCall, "%.")` je vzniknuté pole veľkosti 2 `if table.getn(parts) == 2 then`. Toto indikuje, že ide o zložené volanie (napr. `table.insert()`)
        i. Registruje sa daný modul (*table*) vo funkcii *registerGlobalModule(graph, moduleName, moduleFunctionCall)*. Vytvoria sa uzly pre modul a pre funkciu (*insert*) z daného modulu, takisto sa vytvorí hrana medzi nimi, ktorá má label *FunctionDeclaration*. Oba uzly aj hrana medzi nimi sú pridané do grafu.
    b. ak podmienka vyššie nebola vyhodnotená ako *true* ide o globálnu funkciu, napríklad `pair()` alebo `print()`.
        i. Vytvorí sa nový uzol typu *globalFunction* a pridá sa do grafu.
2. Následne sa prelistujú všetky hrany pre danú funkciu (table.insert(graph.globalCalls[calledFunction], newEdge) hrany) a aktuálna funkcia (node) sa pridá ako ich Target, teda atribút *to* .

Luametrics

V priečinku `luametrics/src/hypergraph` sa nachádzajú súbory zodpovedné za vyhľadávanie a uchová- vanie vzťahov v súbore do hypergrafovej dátovej štruktúry. Tu je taktiež využitá lua knižnica `hypergraph`. Podobne ako v LuaDB, objekty pre hrany, uzly a výskyty sú definované v rovnomenných súboroch, pričom v súbore `graph.lua` je definovaný hypergraf a funkcie nad ním, ako pridanie alebo vymazanie hrán/uzlov.

Metriky

Najvýznamenjšie úlohu v module LuaMetrics sú súbory obsiahnuté v priečinku `luametrics/src/metrics/capture`, kde sa počítajú jednotlivé metriky. V tomto priečinku sa nachádza 11 súborov, komplexnejšie metriky sa počítajú v 5-och, ostatné slúžia na analýzu, rozdelenie kódu alebo na čiastkové metriky. (napr. na bloky, čo je používané pri zvýrazňovaní častí kódu alebo pri počítaní metriky toku informácii pre funkciu).

LOC.lua

V tomto súbore sa počíta základná metrika, ktorou je počet riadkov kódu. Keďže funkcia doMetrics() nie je volaná rekurzívne, metriky sú počítané len pre priamych potomkov uzlu `for key, value in pairs(children) do`. Konkrétne sa pre jednotlivé uzly počítajú riadky podľa podmienok, ktoré na základe tagu kontrolujú o aký typ riadku ide. Čiže je pre jeden uzol počítaný nielen základný počet riadkov (bez podmienok, ak nový riadok +1), ale aj počt riadkov komentárov, úplne prádznych riakov, počet neprázdnych riadkov a počet riadkov obsaujúcich vykonateľný kód. Každé počtídalo je vnorené do inej podmienky. Tieto výsledky sú zapísané v tabuľke pre `node.metrics.moduleDefinitions[exec].LOC.`

```
addCount(LOC, 'lines', value.metrics.LOC.lines)
addCount(LOC, 'lines_comment', value.metrics.LOC.lines_comment)
addCount(LOC, 'lines_blank', value.metrics.LOC.lines_blank)
addCount(LOC, 'lines_nonempty', value.metrics.LOC.lines_nonempty)
addCount(LOC, 'lines_code', value.metrics.LOC.lines_code)
```

infoflow.lua

V tomto súbore je počítaná metrika toku informácií pre funkciu. Pri výpočte tohto druhu metriky je potrebné analyzovať vzťahy danej funkcie k dátam na jej vstupe, pri jej výstupe a vzťah k dátam s ktorými funkcia pracuje. V tomto súbore sú využívané výstupy zo súboru block.lua Na ukážke nižšie sa do uzlu zapisuje, či ide o premennú, z ktorej sa číta alebo do ktorej sa zapisuje, táto informácia je dôležitá pri výpočte metriky toku informácií.

```
local function newVariable(node, text, secondary_nodes, isRead)

   if (locals_stack[text]==nil) then                    -- if variable was not defined before - it belongs to the 'remotes' variables
      addItemToArray(remotes_stack, text, node)
      addItemToHighlightArray(highlight_remote, node, secondary_nodes)

 if (isRead) then -- variable is read from
    node.isRead = true
 else -- write
    node.isWritten = true
 end
   else
      table.insert(locals_stack[text], node)            -- the variable is local - table was defined before
                                          -- insert it into the table (table holds froup of nodes corresponding to the variable with the
same text)
      addItemToHighlightArray(highlight_local, node, secondary_nodes)
   end

   if (moduleMetrics) then
      table.insert(moduleMetrics.variables, node)
   end

end
```

Na riadku 2 je zvýraznené priradenia konkrétnej metriky na príslušné miesto *infoflow.information_flow* v abstraktnom syntaktickom strome funkcii. Je vypočítaná podľa vzťahu pre vyvážený tok informácii *infoflow = počet premmených do kt sa zapisuje ∗ (počet_premmených_z_kt_sa_číta + pocet_výstupov_z_funkcie)*^{2} V *metric*s sa takisto uchovávajú aj všetky počítadlá využité na výpočet metrík. Komplexita rozhrania je vypočítaná ako ako súčet argumentov a return statementov.

```
    funcAST.metrics.infoflow = {}
funcAST.metrics.infoflow.information_flow = (#v_in * (#v_out + return_counter))^2
    funcAST.metrics.infoflow.arguments_in = in_counter
    funcAST.metrics.infoflow.arguments_out = return_counter
funcAST.metrics.infoflow.interface_complexity = in_counter + return_counter
    funcAST.metrics.infoflow.used_nodes = usedNodes
```

cyclomatic.lua

V tomto module je využitý výstup z modulu `statements.lua`, ktorý do metrík zapíše zoznam príkazov spolu s jeho početnosťou. Metriky sú postupne spočítavané pridávaním čiastočných zložitostí volaním pomocnej funkcie.

```
local function add(node, name, count)
    count = count or 0

    if (not node.metrics) then node.metrics = {} end
    if (not node.metrics.cyclomatic) then node.metrics.cyclomatic = {} end

    if (node.metrics.cyclomatic[name]) then
        node.metrics.cyclomatic[name] = node.metrics.cyclomatic[name] + count
    else
        node.metrics.cyclomatic[name] = count
    end

end
```

Argumentami sú uzol (funkcia), meno ('decisions','decisions_all','conditions','conditions_all') a samotná "zložitosť" (väčšinou 1, niekedy počet výskytov if_else v prípade výrazu začínajúcim if. Horné a dolné ohraničenie pre uzol je zapísané vo funkciách *setUpperBound* a *setLowerBound*.

halstead.lua

V tomto súbore sa do metrík zapísajú tzv. Halsteadove metriky.

```
    if (node.metrics == nil) then node.metrics = {} end
    if (node.metrics.halstead == nil) then node.metrics.halstead = {} end

calculateHalstead(node.metrics.halstead, operators, operands)

    ...

    metricsHalstead.operators = operators
    metricsHalstead.operands = operands

    metricsHalstead.number_of_operators = number_of_operators
    metricsHalstead.number_of_operands = number_of_operands
    metricsHalstead.unique_operands = unique_operands
    metricsHalstead.unique_operators = unique_operators

    metricsHalstead.LTH = number_of_operators + number_of_operands
    metricsHalstead.VOC = unique_operands + unique_operators
    metricsHalstead.DIF = (unique_operators / 2) * (number_of_operands/unique_operands)
```

```
    metricsHalstead.VOL = metricsHalstead.LTH * (math.log(metricsHalstead.VOC) / math.log(2) )
    metricsHalstead.EFF = metricsHalstead.DIF * metricsHalstead.VOL
    metricsHalstead.BUG = metricsHalstead.VOL/3000
    metricsHalstead.time = metricsHalstead.EFF / 18
```

Táto metrika ukladá operátory, operandy a zapisuje do metrík nielen tieto zoznamy, ale i ich početnosti, početnosti unikátnych operandov a operátorov. Tieto údaje ďalej využíva na počítanie halsteadových metrík dosadením do príslušných vzťahov. Podobne ako pri LOC aj tieto metriky sa počítajú nielen pre funkcie, ale aj pre moduly.

document_metrics.lua

V tomto súbore sa počítajú metriky dokumentácie, teda komentárov v kóde. Do metrík sa zapisujú metriky ako počet zdokumentovaných a počet nezdokumentovaných funkcií/tabuliek a pomer týchto množín. Ďalšou metrikou sú počty informatívnych komentárov rozdelených podľa druhu.

```
COMMENT = function(data)
    data.parsed=comments.Parse(data.text)
    if(data.parsed and data.parsed.style=="custom")then
        if(data.parsed.type == "todo")then
            table.insert(TODOs,data)
        end
        if(data.parsed.type == "bug")then
            table.insert(BUGs,data)
        end
        if(data.parsed.type == "question")then
            table.insert(QUESTIONs,data)
        end
        if(data.parsed.type == "fixme")then
            table.insert(FIXMEs,data)
        end
        if(data.parsed.type == "how")then
            table.insert(HOWs,data)
        end
        if(data.parsed.type == "info")then
            table.insert(INFOs,data)
        end
    end

return data
```

Počty zdokumentovaných rsp. nezdokumentovaných funkcii sa počíta pomocou jednoduchých cyklov ako je vidieť nižšie.

```
if(data.metrics.functionDefinitions ~=nil) then
        for k,v in pairs(data.metrics.functionDefinitions) do
      if(v.documented==1)then
          data.metrics.documentMetrics.documentedFunctionsCounter = data.metrics.documentMetrics.documentedFunctionsCounter +1
      else
          data.metrics.documentMetrics.nondocumentedFunctionsCounter =
data.metrics.documentMetrics.nondocumentedFunctionsCounter +1
          end
        end
    end
```

Magnety

Metauzly sú uzly, ktoré nie sú priamo súčasťou grafu, ale môžu ovplyvňovať výsledné rozloženie grafu. Tieto uzly majú fixné pozície a môžu byť pridané do scény napríklad

používateľom. Môže byť viacero druhov metauzlov, z ktorých jedným druhom sú práve magnety.

Algoritmus použitý v tomto riešení podporuje tri typy metauzlov:

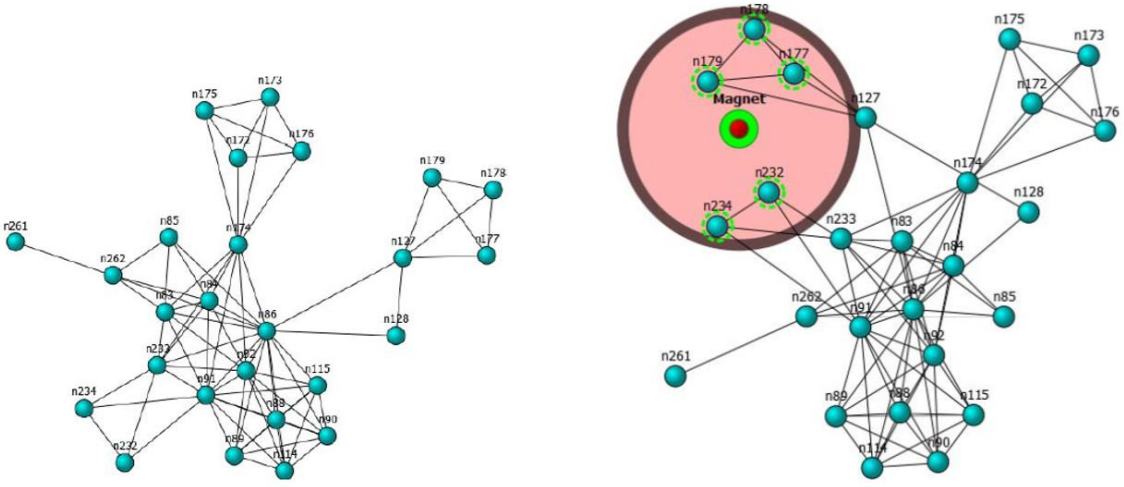
1. Priťahuje k sebe uzly, ktoré sú s ním spojené.
2. Priťahuje k sebe uzly, ktoré sa nachádzajú v určenom rozsahu vzdialenosti od metauzla.
3. To, či je uzol k metauzlu priťahovaný, je určené na základe funkcie, ktorej vstupom je konkrétny uzol. Takto môžeme zadefinovať aj komplikovanejšiu podmienku.

Každý metauzol má určený aj násobič sily, ktorou má uzol priťahovať.

Ako je vyššie spomenuté, magnety k sebe priťahujú uzly v grafe, ktoré spĺňajú určité kritéria. Môže ísť napríklad o stupeň uzla, vzdialenosti od iného uzla alebo o nejakú vlastnosť, ktorú tento uzol v sebe nesie. Takéto kritérium však môže závisieť aj na iných magnetoch v scéne. Môžeme vytvoriť magnet, ktorý bude priťahovať všetky uzly, ktoré nie sú priťahované inými magnetmi. Magnety pôsobia príťažlivou silou na uzly, ktoré spĺňajú zadané kritérium a prípadne odpudivou silou na všetky zvyšné uzly v grafe.

Mnohokrát môže byť potrebné umiestniť uzly so spoločnými znakmi do určitého geometrického tvaru. V takomto prípade môžeme geometrický útvar umiestniť okolo magnetu. Magnet následne zaručí, že všetky uzly, ktoré spĺňajú kritérium, sú priťahované do určenej oblasti a zvyšné uzly sú od nej odpudzované.

Upravovanie rozloženia grafov silovo-riadených algoritmov pomocou magnetov bolo implementované vo vizualizačnom nástroji MagnetViz. Okrem vyššie popísaných použití magnetov, MagnetViz umožňuje používať aj hierarchie magnetov, kde dcérske magnety môžu priťahovať len určitú podmnožinu uzlov svojho rodiča. Na obrázku 1 vidíme graf rozložený nástrojom MagnetViz pred použitím magnetu a po použití magnetu spolu s geometrickým ohraničením.



Naľavo je graf bez použitia magnetu. Napravo vidíme ten istý graf spolu s magnetom a kruhom do ktorého magnet priťahuje uzly.

Technická dokumentácia

Implementácia v aktuálnom stave zahŕňa podporu v layoutovači (Terra časť), obslužné Lua funkcie v layout manageri a GraphCore. Chýba podpora úprav metauzlov tretieho druhum pre ktoré je potrebné navrhnúť spôsob definovania a priraďovania podmienkových funkcií. V súbore `3dsoftvis_remake\resources\scripts\module\layouter\algorithms\terra\fruchterman_reingold.t`, na riadkoch 273 až 305 je časť k magnetom. Táto časť slúži na vypočítanie príťažlivých a odpudivých síl medzi metauzlami a jednotlivými uzlami grafu a na preskupenie grafu. Na riadkoch 418 až 584 sa nachádzajú funkcie pre manažment metauzlov.

Layoutovač obsahuje pole `metaNodes` v ktorom sa ukladajú metauzly a premennú `metaNodeCount`. Pri pridávaní a odoberaní metauzlov sa pole realokuje na menšie alebo väčšie a príslušne sa zmení aj hodnota premennej. Metauzly sa kedysi nachádzali priamo v grafe, teraz sú už oddelené v tomto poli.

Implementácia je robená podľa vzoru obmedzovačov. Implementáciu pridávania obmedzovačov môžeme vidieť v súbore `3dsoftvis_remake\resources\scripts\module\layouter\layout_manager.lua`. Z tohto súboru sa následne volá funkcia s implementovaným algoritmom na layoutovanie - v našom prípade už spomínaný `fruchterman_reingold.t`, keďže v tomto je spravená predpríprava na magnety. V súbore `layout_manager.lua` sa zavolá funkcia `runLayouting()`, ktorá na viacero krokov nakoniec v súbore `fruchterman_reingold.t` spustí funkciu `calculateLayout()`, ktorá už zabezpečuje samotné layoutovanie. Funkcie na pridávanie a upravovanie obmedzovačov sú volané zo súboru `3dsoftvis_remake\Projects\3DSoftviz\CSProjects\GraphCore\LayoutManager.cs` od riadku 139 a pre magnety od riadku 322. Zo C# súboru sa Lua funkcie volajú cez C++ wrapper `LuaInterface` pomocou funkcie `DoString()`.

Zhrnutie

1. Čo bolo doposiaľ implementované
    - layoutovanie metauzlov (magnetov) - Terra
    - pridávanie a upravovanie magnetov - Lua
    - pridávanie a upravovanie magnetov - C#
2. Čo bude treba implementovať v budúcnosti
    - metauzly tretieho druhu
        - pridávanie a upravovanie magnetov - Terra, Lua, C#
    - interakcia s používateľom - Unity
    - testy

Testy

Testy pre túto funkcionalitu zatiaľ neexistujú, preto sa zatiaľ snažíme spraviť aspoň high-level návrh testov.

High-level návrh testov

1. **Test na počet metauzlov** - či sedí premenná `metaNodesCount` s reálnym počtom metauzlov v poli
2. **Test na fixnosť metauzlov** - keďže metauzly majú mať fixné pozície, pridáme do scény zopár magnetov, prejdeme zopár iteráciami layoutovania a skontrolujeme, či je každý z magnetov naozaj na tej pozícii, kam sme ho umiestnili
3. **Test na vzdialenosť od prvého druhu magnetu** - vytvoríme magnet, ktorý bude spojený s jedným uzlom grafu, vypočítame ich vzdialenosť vzdušnou čiarou, takisto si náhodne vyberieme zopár ďalších uzlov v grafe a vypočítame ich vzdialenosť - po prelayoutovaní by sa mala vzdialenosť uzla spojeného s magnetom od magnetu zmenšiť a vzdialenosť ostatných náhodne vybraných uzlov od magnetu by sa nemala zmenšiť (to znamená, že ostane rovnaká, alebo sa zväčší)
4. **Test na vzdialenosť od druhého druhu magnetu** - analogicky k testu na prvý druh magnetu spravíme test aj na tento druh magnetu
5. **Test na vzdialenosť od tretieho druhu magnetu** - analogicky k testu na prvý druh magnetu spravíme test aj na tento druh magnetu

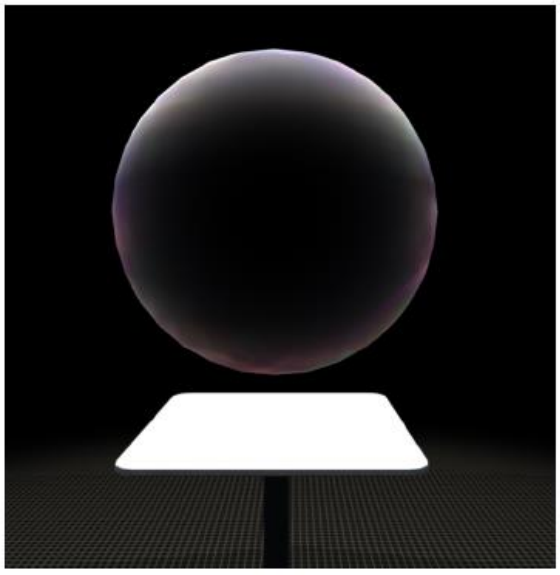
Obmedzovače a gizmá

Všeobecný opis (motivácia)

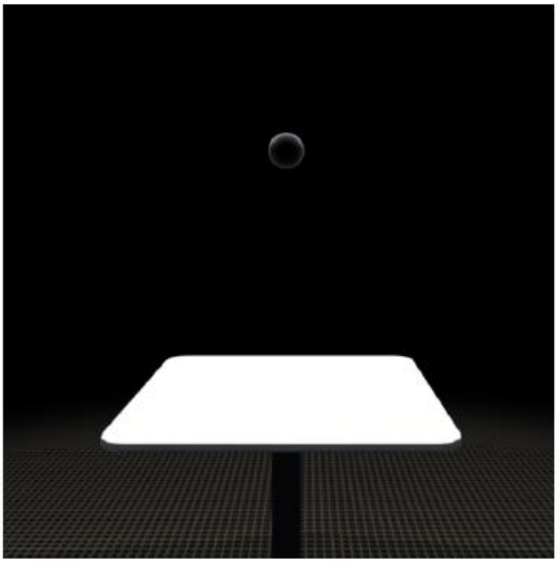
Natívne funkcie, ktoré sa nachádzajú v Leap core moduloch pre Unity, poskytujú jednoduchú interakciu rukami, ako napríklad uchopenie alebo ťahanie objektov, avšak táto funkcionalita je pomerne obmedzená. Taktiež tieto moduly často nestačia čo sa týka presnosti, hlavne v prípade takýchto rozsiahlych grafov, s akými v tomto projekte pracujeme. Chceme docieliť aby používateľ mohol pohodlne a intuitívne pracovať s grafom, prípadne v budúcnosti s grafmi, pričom celá interakcia bude založená na gestách. Používateľ taktiež potrebuje v každom momente poznať s akým uzlom, časťou grafu alebo celým grafom aktuálne pracuje. Práve tu prichádzajú do úvahy gizmá a obmedzovače, ktoré sú v projekte implementované.

Gizmo je jednoduchý kváder, ktorý sa zobrazí okolo objektu v grafe a poskytuje používateľovi možnosť vykonávať nad daným objektom (alebo skupinou objektov) rôzne transformácie ako napríklad presúvanie, rotáciu, škálovanie.

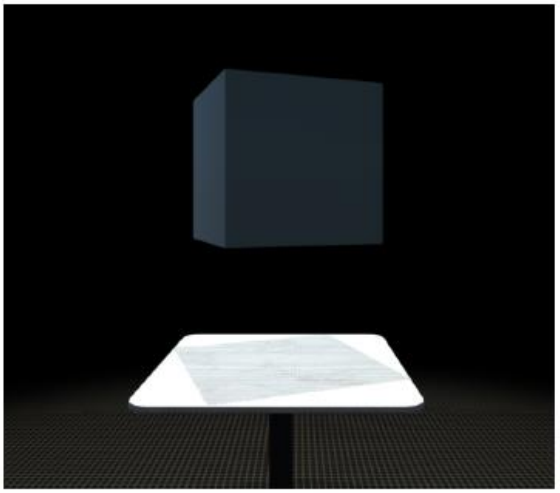
Obmedzovač je objekt, ktorý pomáha vybrať skupinu objektov a re-layoutovať (reorganizovať) ich. K dispozícii sú aktuálne 4 typy a to guľa, guľový bod, kocka a doska.



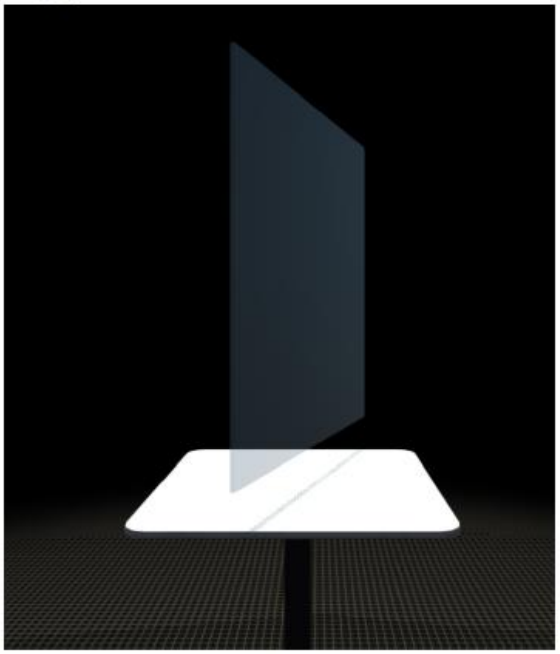
(a) Sphere restriction



(b) Sphere-point restriction



(c) Box restriction



(d) Plane restriction

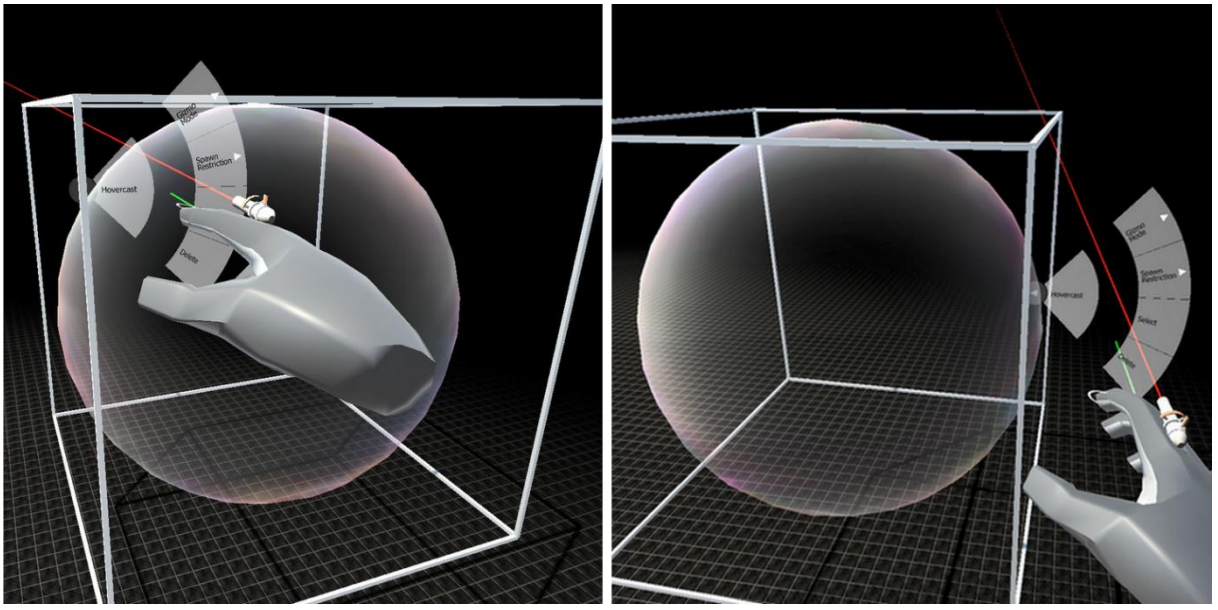
Gizmá - bližšia špecifikácia

Aktuálne je v assetoch pre Unity možné nájsť také, ktoré podobnú funkcionalitu poskytujú pre väčšinu typov objektov, tieto riešenia sa väčšinou spoliehajú na shadery, ktoré tieto gizmá vykreslujú počas post-processingu, teda potom, čo bola celá scéna už dávno vykreslená. Toto je dobré v prípade ovládania myšou, nakoľko myš bude vždy vykresľovaná nad všetkým ostatným. V našom prípade je to však nepoužiteľné, nakoľko ruky sú fyzicky prítomné v samotnej scéne a môžu interagovať iba s inými objektami v scéne. V jednoduchosti to znamená to, že tieto gizmá by boli vždy renderované nad rukami, čo by jednoznačne pokazilo akúkoľvek simuláciu hĺbky nášho priestoru pre používateľa.

Z pohľadu funkcionality sú aktuálne implementované dva typy giziem. Gizmo, ktoré dokáže objektom pohybovať a rotovať a gizmo, ktoré dokáže daný objekt škálovať.

Gizmo na hýbanie a rotáciu objektu

Tento typ gizma je zobrazený v prípade priblíženia sa rukou k interagovateľnému objektu. Vizuálne sa zobrazí ako skupina bielych hrán v tvare kocky okolo daného objektu. Po zobrazení vieme jeho uchopením objekt rotovať alebo presúvať. V prípade, že je v blízkosti ruky viacero objektov, gizmo sa zobrazí okolo najbližšieho z nich. Vedľa aktívneho gizma sa zobrazuje UI element, ktorý je zobrazený na obrázku nižšie.



Počas toho ako používateľ prechádza cez menu, algoritmus na zobrazovanie gizma je pozastavený, aby nedošlo k tomu, že keď používateľ prekročí minimálnu vzdialenosť od elementu, potrebnú pre zobrazenie gizma, samotné gizmo a jeho UI zmiznú.

Samotná kocka gizma je predstavovaná skupinou úzkych blokov, ktoré sú zoskupené pod jedným prázdnym elementom, aby bolo možné ich automaticky škálovať naraz ako jeden objekt. Rovnako je tomu tak aj v prípade gizma pre škálovanie.

Layoutovanie grafu je počas transformácie objektu prostredníctvom gizma pozastavené, aby sme predišli zbytočne náročným operáciám.

Gizmo na škálovanie objektu

Tento typ gizma umožňuje škálovať objekt po rôznych kombináciách osí. Pri používaní tohto gizma sa okolo objektu zobrazí 26 ovládacích kociek, ktoré spolu formujú celú kocku. Uchopením a ťahaním týchto kociek je možné daný objekt škálovať.

UI

Keď je gizmo aktívne, zobrazí sa pri ňom samostatné UI, ktoré ponúka používateľovi akcie, súvisiace s daným gizmom. Tie sú selekcia, deselekcia a odstránenie. Taktiež je možné prostredníctvom tohto rozhrania zmeniť typ gizma. Toto rozhranie sa zobrazí vždy na bode na

hrane gizma, ktorá ja k používateľovej ruke najbližšie. Keď je používateľova ruka v blízkosti tohto rozhrania, jeho presúvanie sa pozastaví, aby mohol používateľ s UI pracovať a neuskakovalo pred tým podľa pohybu jeho ruky. (viď Mura s. 44)

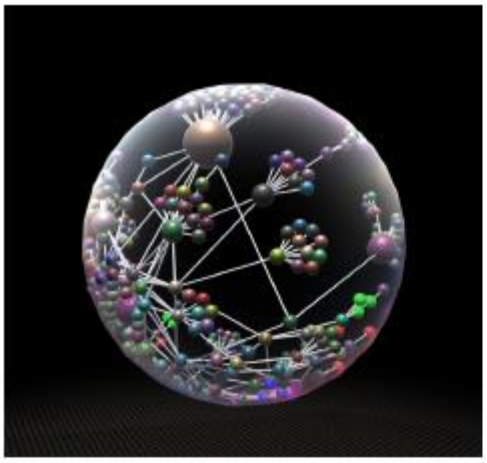
Obmedzovače – bližšia špecifikácia

Aktuálne sú teda implementované 4 typy obmedzovačov a to:

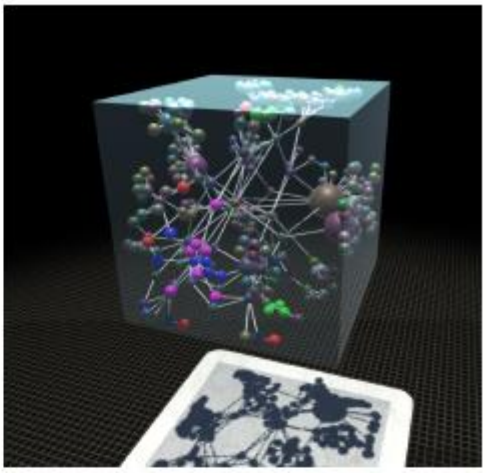
- guľa (sphere)
- guľový bod (sphere point)
- kocka (box/cube)
- doska (plane)

Potrebné je poznamenať, že guľový bod je v skutočnosti guľa v malej veľkosti, čo znamená, že v nej nie je miesto pre rozloženie objektov, takže pri použití budú cez seba. To je dobré v prípade, že potrebujeme zoskupiť nejaké množstvo objektov bez toho, aby sme ich neskôr chceli skúmať.

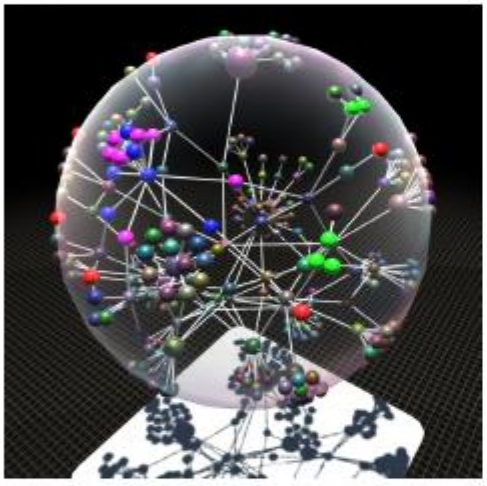
Obmedzovač môže mať 3 módy. Prvým je negatívny mód, kedy sa objekty vnútri obmedzovača, dotýkajúce sa okraja prilepia na okraj zvnútra. Druhý mód je taký, že sú tieto objekty v polovici preseknuté hranou a teda polovica je vo vnútri a polovica vonku. Tretí je pozitívny mód, kedy sú tieto objekty prilepené na hrane zvonka. Najlepšie je to možné pozorovať na obrázku nižšie.



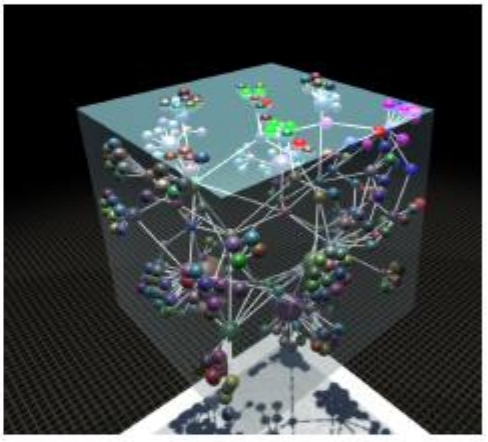
(a) Sphere restriction Negative mode



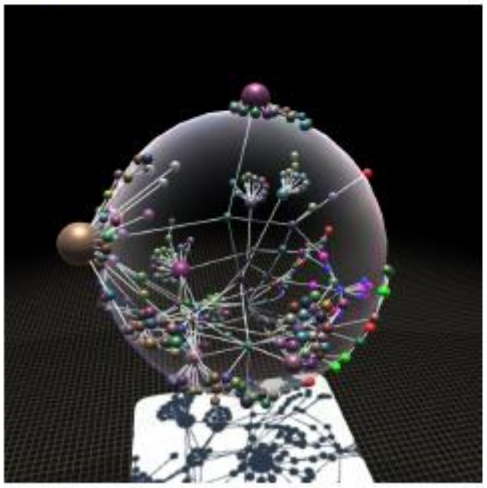
(b) Box restriction Negative mode



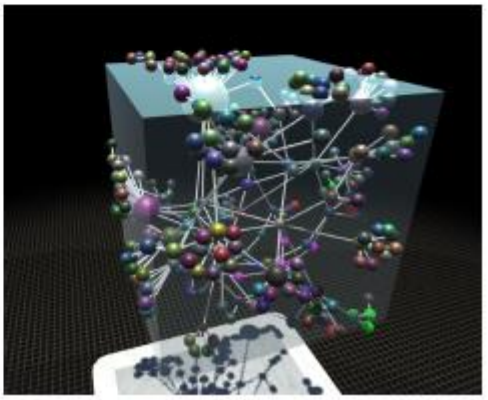
(c) Sphere restriction Center mode



(d) Box restriction Center mode



(e) Sphere restriction Positive mode

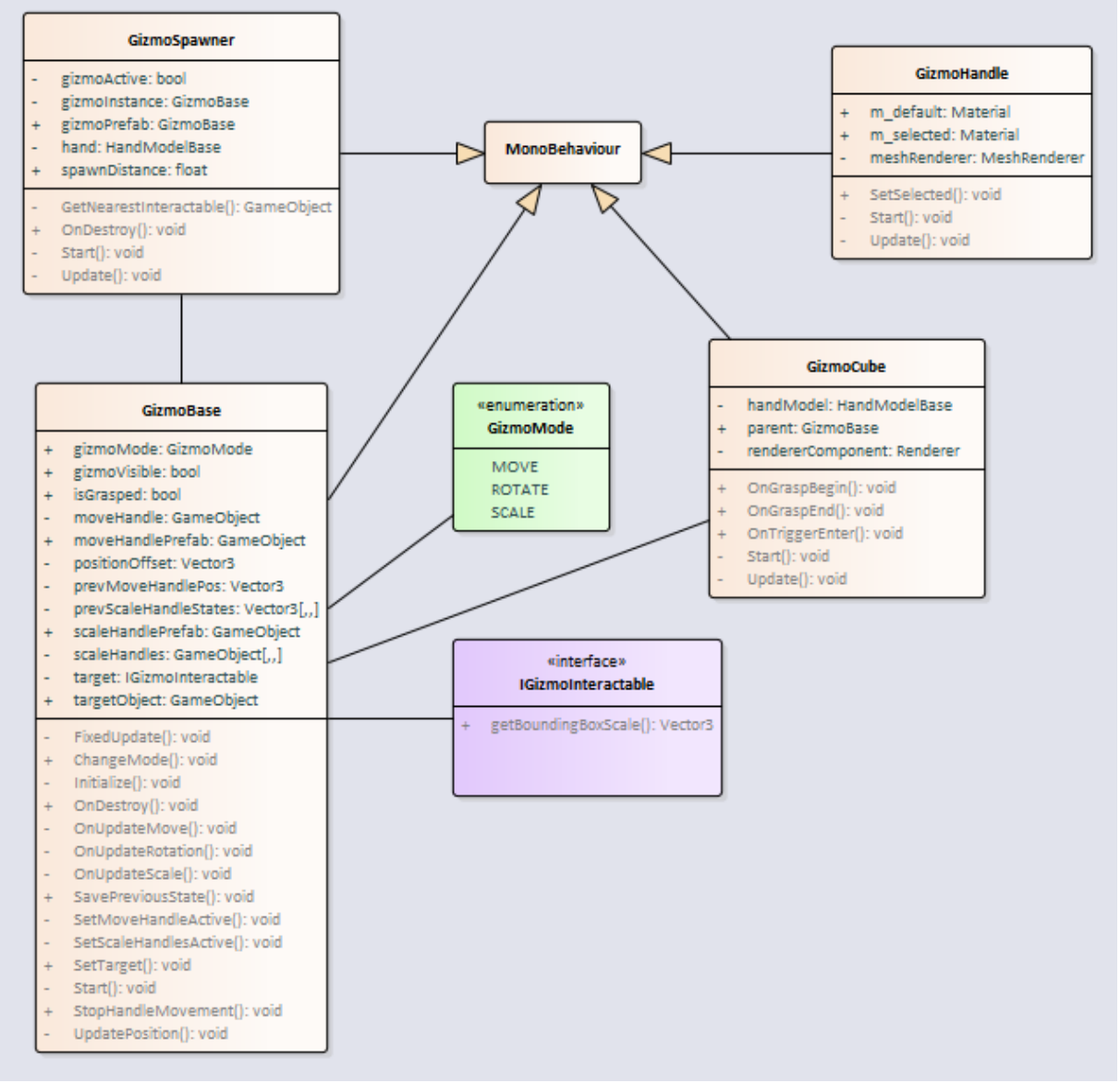


(f) Box restriction Positive mode

Technická dokumentácia

Gizmá

Na nasledujúcom diagrame tried sa nachádza štruktúra priečinka, v ktorom sú implementované gizmá (3DSoftviz\UnityProject\Assets\Scripts\Gizmo)



V prípade, že chceme aby bolo možné s objektom interagovať prostredníctvom gizma, musí tento objekt implementovať interface *IGizmoInteractable.* Ten obsahuje dve metódy:

getBoundingBoxScale() – na základe Vector3 určí veľkosť požadovaného gizma, aby bol cieľový objekt obalený presne

getSpawnDistance() – minimálna vzdialenosť ruky od daného objektu na to, aby sa gizmo zobrazilo

GizmoSpawner skript je vložený v Leap hand modeli. V tomto skripte sa prechádzajú všetky interagovateľné objekty v scéne a určí najbližší k aktuálnej pozícii ruky. Týmto spôsobom sa docieli, že v jednom čase sa môže gizmo nachádzať iba nad jedným objektom.

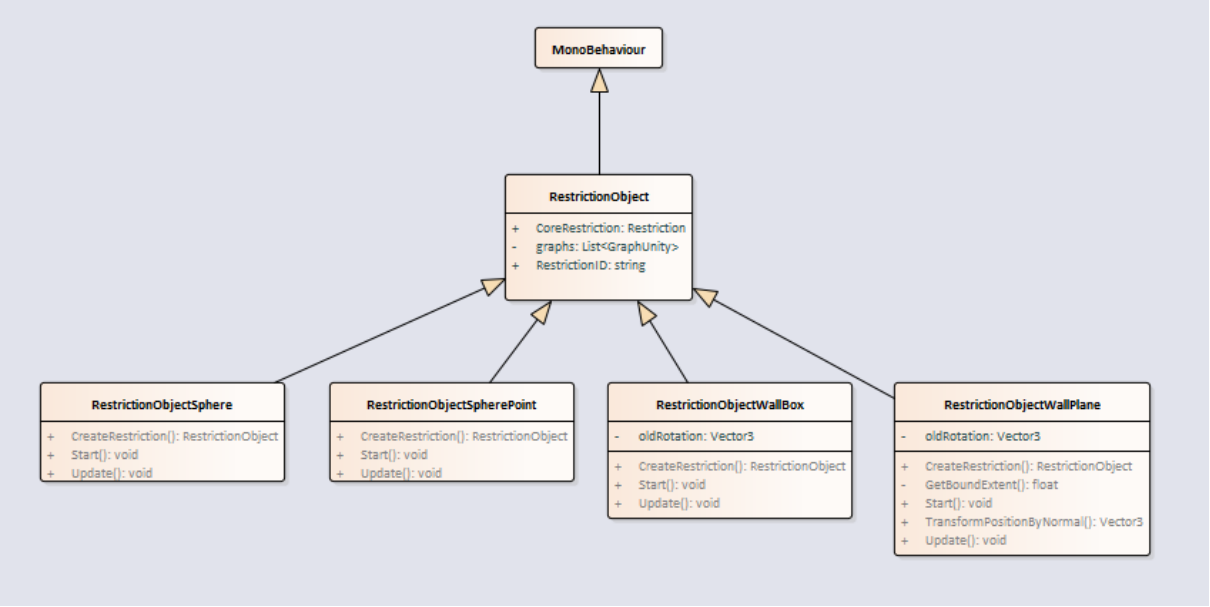
GizmoBase skript je vložený do prázdneho rodičovského objektu, v ktorom sa nachádzajú bloky pre vykreslenie kocky. Tento skript obsluhuje väčšinu logiky týkajúcej sa giziem. Zároveň si pamätá cieľový objekt (napríklad uzol) a aktualizuje jeho pozíciu podľa pozície gizma.

InteractionBehavior je komponent pripojený na kocku gizma a zabezpečuje uchopenie a pohyb pomocou Leap ruky.

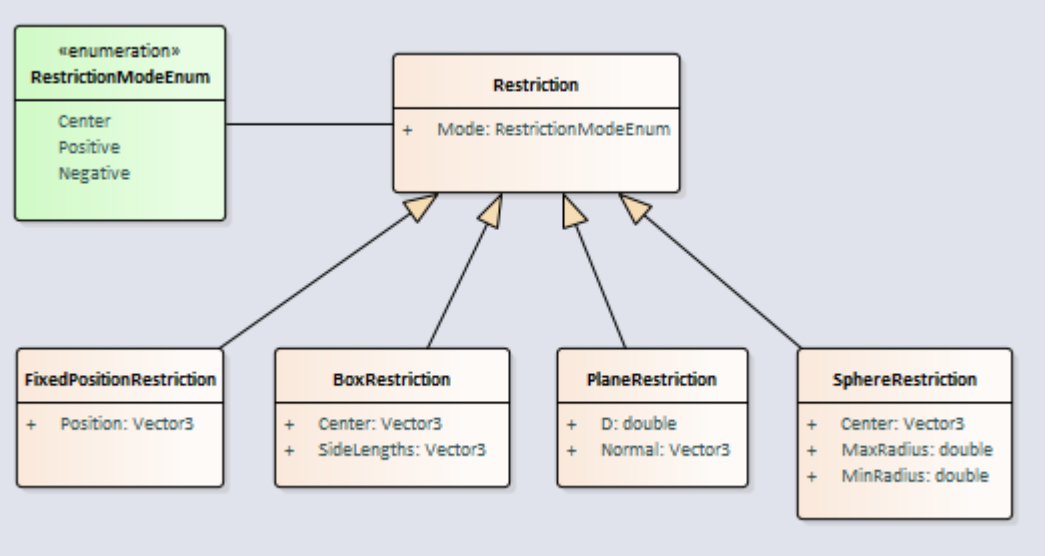
setNodePosition() je funkcia na vrstve LuaInterface, ktorú je potrebné zavolať po skončení posúvania objektu pomocou gizma.

Obmedzovače

Na nasledujúcom diagrame je možné vidieť štruktúru priečinka RestrictionObjects v Unity projekte (3DSoftviz\UnityProject\Assets\Scripts\RestrictionObjects)



Na druhom diagrame je možné vidieť štruktúru priečinka Restrictions v GraphCore projekte (3DSoftviz\CSProjects\GraphCore\Restrictions).



Testy

Testy sa k tejto časti funkcionality projektu aktuálne nenachádzajú. Je teda potrebné nejaké navrhnúť a doimplementovať a následne budú vložené do tejto dokumentácie.

UML diagramy

Motivácia

Používateľ programu by mal mať možnosť zobraziť si rôzne metriky a UML diagramy analyzovaného kódu. Dva základné diagramy, ktoré chceme generovať sú Class diagram a Sequence diagram.

Návod na použitie

1. načítame graf moonscript projektu (návod)
2. myšou označíme triedu (červená) alebo metódu (zelená)
3. stlačíme klávesu 5

Technická dokumentácia

V aktuálnom stave je možné generovať Class diagram a Sequence diagram pre Moonscript kódy. Implementácia je realizovaná v module `luameg` a v lua časti 3D Softvizu (`App/main.lua` a `softviz/graph_handler`).

Luameg

Modul `luameg.plantuml` exportuje príslušné funkcie pre generovanie Class a Sequence diagramov buď v plantuml txt alebo priamo svg. Pri generovaní rekurzívne prehľadáva graf od zvoleného uzla a generuje plantuml textový súbor. Následne je spustený `plantuml.jar` s týmto textovým súborom na vstupe a vygeneruje svg súbor. Obsah svg súboru je načítaný do premennej, súbory odstránené a hodota navrátená.

Pre zobrazenie grafu sú generované uzly `method_sequenceDiagram` a `class_ClassDiagram`. Tieto uzly slúžia pre layoutovanie, aby boli vhodne umiestnené v priestore.

Lua časť 3DSoftVizu

V `App/main.lua` sú dve funkcie pre generovanie Class a Sequence diagramov. Ako argument majú id grafu a id uzla. Ak je tento uzol
typu `method_sequenceDiagram` alebo `class_ClassDiagram`, vyhľadá sa k nim prislúchajúci `class/method` uzol. Nasleduje volanie rovnomennej funkcie v `graph_handler`-i, v ktorom je referencia na graf.

Unity

Unity obsahuje webový prehliadač `ZFBrowser`, v ktorom sa UML diagramy zobrazujú. Hlavná trieda, ktorá obsluhuje tento prehliadač je `BrowsersManager`, ktorá vyhľadáva vyššie spomínané layoutovacie uzly a pre ňe inštanciuje príslušné prehliadače. Prehliadače sú spočiatku skryté, zobrazia sa po označení príslušneho uzla a stlačení číselnej klávesy 1 až 5, pre UML diagramy to je práve klávesa 5.

Každý typ prehliadača má svoj 'skript', ktorý obsluhuje svoju inštanciu `ZFBrowser`-a a prefabu. `UmlBrowserScript` pri aktivácii (zobrazení) prehliadača spustí funkciu generovania diagramu v svg formáte a string posunie do prehliadača.

Súvisiace súbory

- `UnityProject/Assets/Scripts/Browser/UmlBrowserScript.cs`
- `UnityProject/Assets/Scripts/Browser/BrowsersManager.cs`
- `UnityProject/BrowserAssets/`
- `resources/scripts/app/main.lua`
- `resources/scripts/module/softviz/graph_handler.lua`

Príručky

Inštalačná príručka

Windows

Prerekvizity

- Visual Studio 2019 s nasledovnými komponentmi
    - .net framework v4.7.1
    - Desktop C++ development
    - Visual Studio Tools for Unity
- Unity editor 2018.3.7f2
- Cmake
- Doxygen
- Git for Windows
- vytvorený nezaheslovaný SSH kľúč a nahraný na GitLabe aj GitHube

Postup

Stiahnutie zdrojákov

- vo vhodnom priečinku vykonáme `git clone git@gitlab.com:FIIT/3DSoftVis_Remake/3dsoftvis_remake.git`
- prepneme na develop vetvu `git checkout develop`
- inicializujeme závislosti `git submodule update --init`

Kofigurácia build systému

- otvoríme CMake
    - ako `source code` zvolíme priečinok `3dsoftvis_remake`
    - ako `build binaries` vytvoríme a zvolíme `3dsoftvis_remake_build`
    - klik na configure
    - v prvotnej konfiguracii vyberieme
        - generátor Visual Studio 2019
        - architektúru x64
        - zaškrtneme `build_unity` a `use_terra`
    - v niektorých prípadoch je potrebné nakonfigurovať `unity_executable` ako cestu k `unity.exe`

- klik na generate

Vytvorenie premenej prostredia `INCLUDE_PATH`

- otvoríme vlastnosti počítača > rozšírené nastavenia > premenné prostredia > nová
- názov `INCLUDE_PATH`
- v hodnote skontrolujeme a prispôsobíme verzie v cestách
- pre prehladnosť uvádzame hodnotu rozdelenú na riadky, ale pred použitím ju je potrebné spojiť do jedného riadka bez medzier

```
C:\Program Files (x86)\Microsoft Visual Studio\2019\Community\VC\Tools\MSVC\14.24.28314\include;
C:\Program Files (x86)\Windows Kits\10\Include\10.0.17763.0\ucrt;
C:\Program Files (x86)\Windows Kits\10\Include\10.0.17763.0\um;
C:\Program Files (x86)\Windows Kits\10\Include\10.0.17763.0\shared;
C:\Program Files (x86)\Windows Kits\10\Include\10.0.17763.0\winrt;
C:\Program Files (x86)\Windows Kits\10\Include\10.0.17763.0\cppwinrt
```

Build závislostí

- otvoríme solution `3dsoftvis_remake_build\3DSoftviz.sln` vo Visual Studio
- klik pravým na CopyExternalDataToUnity > build

Spustenie Unity

- otvoríme `3dsoftvis_remake\Projects\3DSoftviz\UnityProject` v Unity
- zmeníme nasledovné a reštartujeme Unity
  Unity menu > edit > preferences > external tools > editor > VS2019

Časté problémy

- `error CS1704: An assembly with the same simple name 'SyntaxTree.VisualStudio.Unity.Bridge' has already been imported.`
  táto chyba by sa mala vyskytovať len pri použití VS2019
  pre jej odstránenie vymažte celý
  priečinok `Projects\3DSoftviz\UnityProject\Assets\UnityVS`
- pri použití VS2017 býva opačný problém, "SyntaxTree" asset je potrebné importovať
  v Unity importujeme Asset `Visual Studio 2017 Tools.unitypackage` z `Program Files (x86)/Microsoft Visual Studio Tools for Unity`
- cmake nevie vygenerovať projekt, ak je na počítači viacero verzií Visual Studia a zvolí sa taká, ktorá nemá komponent Desktop C++ development
- ak pri kroku Build závislostí neprebehnú všetky buildy úspešne (28 suceeded) a buildy, ktoré zlyhali hlásia zablokovaný prístup k repozitáru, je pravdepodobné že máte vytvorený zaheslovaný SSH kľúč alebo ho máte nesprávne nahodený na gitlabe/githube. Treba si vytvoriť buď nový nezaheslovaný SSH kľúč a spustiť príkazy `ssh git@gitlab.com` a `ssh git@github.com`, ktoré by Vám mali dať nasledujúci výstup:

```
Welcome to GitLab, @username!
Hi username! You've successfully authenticated, but GitHub does
not provide shell access.
```

Problémy častejšie evidujeme aj v skupinovom OneNote, prístupný po pridaní do MS Teams

MacOS

Prerekvizity

- Mono 6.4.0
    - Mono predstavuje open source implementáciu Microsoft .NET Framework.
- CMake 3.15.4
    - CMake je nástroj na správu procesu zostavovania softvéru, podporuje adresárovú hierarchiu a viacnásobné závislosti.
- Doxygen 1.8.16
    - Doxygen je nástroj na generovanie dokumentácie z anotovaného zdrojového kódu aplikácie.
- Unity 2018.1.9f2
    - Unity je vývojová platforma pre tvorbu real-time 3D aplikácií,
    - Potrebné je stiahnuť a nainštalovať aplikáciu Unity Hub, do ktorého je potom možné pridať požadovanú verziu Unity.
- Git
    - potrebné je nahrať svoj verejný kľúč na **GitLab**, ale aj **GitHub**

Postup

Vygenerovanie ssh kľúča

- Vygenerujeme ssh kľúč `ssh-keygen -t rsa -b 4096 -C "your_email@example.com"`,
- Spustíme ssh-agent na pozadí `eval "$(ssh-agent -s)"`,
- Prostredníctvom ľubovoľného textového editora modifikujeme súbor ~/.ssh/config

```
Host *
  AddKeysToAgent yes
  UseKeychain yes
  IdentityFile ~/.ssh/id_rsa
```

- Pridáme privátny SSH kľúč do ssh-agenta, pokiaľ sa vygenerovaný privátny kľúč nachádza v súbore s iným menom, názov *id_rsa* týmto názvom nahradíme `ssh-add -K ~/.ssh/id_rsa`,
- príkazom *pbcopy* skopírujeme verejný kľúč do schránky (clipboard) `pbcopy < ~/.ssh/id_rsa.pub`,
- kľúč potom vložíme do svojho konta na portáloch **GitHub** a **GitLab**.

Naklonovanie repozitára 3DSoftVis_Remake

- pokiaľ sme nastavili SSH kľúč, repozitár naklonujeme prostredníctvom SSH

`git clone git@gitlab.com:FIIT/3DSoftVis_Remake/3dsoftvis_remake.git`
a prejdeme do adresára `cd 3dsoftvis_remake/`,

Výber vetvy

- príkazom `git checkout zemko-us-42` vyberieme požadovanú vetvu - v našom prípade pracovnú vetvu pre platformu macOS

Inicializácia sub-modulov

- inicializáciu vykonáme príkazom `git submodule update --init --recursive`,

Konfigurácia CMake

- otvoríme program CMake,
- ako `source code` zvolíme priečinok `3dsoftvis_remake`,
- ako `build binaries` zvolíme priečinok `3dsoftvis_remake/_build`,
- stlačíme tlačidlo configure,
- ako generátor použijeme `Unix Makefiles` a vyberieme možnosť `Use default native compilers`,
- vyberieme možnosti `BUILD_DOXYGEN_DOCUMENTATION`, `BUILD_SOFTVIZ_MODULES_TESTS`, `BUILD_UNITY`, `USE_TERRA_BINARIES`,
- ako `CMAKE_INSTALL_PREFIX` zvolíme priečinok `3dsoftvis_remake/_install`,
- stlačíme tlačidlo configure,
- ako `UNITY EXECUTABLE` nastavíme cestu k `Unity.app`,
- ako `UNITY_TARGET_PLATFORM` zvolíme MacOS,
- stlačíme tlačidlo generate

Build

- v príkazovom riadku prejdeme do adresára `3dsoftvis remake/ build`,
- build vykonáme príkazom `make CopyExternalDataToUnity`,

Import do Unity

- spustíme aplikáciu Unity Hub,
- vyberieme možnosť Add,
- vyhľadáme podadresár `3dsoftvis_remake/Projects/3DSoftviz/UnityProject`
- vyberieme možnosť Open.

Testy

- testy pre C# moduly - SoftvisModules - je možné zostaviť príkazom `make SoftvizModulesTests` z adresára `3dsoftvis remake/ build`,
- testy sa potom nachádzajú v adresári `3dsoftvis_remake/_install/SoftvizModules.Tests`,
- prejdeme do adresára `3dsoftvis_remake/Projects/3DSoftviz/CSProjects/`,
- spustíme príkaz `nuget restore . `, ktorý nainštaluje závislosti potrebné pre spúšťanie testov,
- prejdeme do adresára `packages/NUnit.ConsoleRunner.3.9.0/tools/`,
- následne je možné testy spustiť príkazom `mono nunit3-console.exe ../../../../../../_install/SoftvizModules.Tests/*.dll`

Problémy

- **Terra**
    - v projekte sa využíva nadstavba nad jazykom Lua - Terra (pre layouter v Unity vrstve),
    - použitá implementácia Terra v sebe zahŕňa Lua kompilátor, ktorý má na platforme macOS problém s alokáciou pamäti,

        - namiesto Terra je momentálne použitý nástroj Luapower, resp. LuaJIT, ktorý problém s alokáciou pamäti nemá, problémom je však implementácia Terra, nakoľko obsahuje bug, ktorý momentálne neumožňuje možnosti nadstavby Terra využiť.
- **Unity**
    - pri snahe o spustenie projektu aplikácia vypíše chybové hlásenie `DllNotFoundException: luainterface`,
    - problém môže spôsobovať Terra, nakoľko ju využíva layouter,
    - ďalším problémom môže byť nastavenie ciest
- **Testy**
    - po zmenách CMakeLists.txt súborov a použití implementácie Luapower sú uspešne vykonané všetky testy zamerané na modul LuaInterface
    - pri testoch Lua.Common.Tests je úspešne vykonaná väčšina testov, neúspešne sú vykonané 2 testy zamerané na modul LuaGraph - GraphObjectRawDataAreLoadedCorrectly a LuaGraphLoadsMockedLuaGraphCorrectly,
    - testy GraphCore.Tests sú neúspešné všetky,
    - chyba vzniká pri volaní konštruktora modulu LuaGraph.

Vývojárska príručka

IDE pre Lua

Pre vývoj Lua modulov je možné použiť Visual Studio Code, ZeroBrane Studio a vyvíjať lokálne alebo v Docker kontajneri. Odporúčame použiť Visual Studio Code, najmä kvôli použitiu docker-u a prostredia zhodného s CI.

Visual Studio Code

Príprava repozitárov a kontajnera

Na disku na ľubovoľné miesto naklonujeme a repozitár devenv a inicializujeme submoduly. Submoduly jednotlivých submodulov (teda závislosti lua modulov) nie je potrebne sťahovať, pretože niektoré sú nainštalované v image a zvyšné ako submoduly devenv repozitára. Následne vytvoríme image a spustíme kontajner.

```
git clone git@gitlab.com:FIIT/Common/devenv.git
cd devenv
git submodule update --init --remote
docker-compose run --service-ports --name luadev luadev
```

Kontajer stačí vytvoriť len raz, neskôr je možné ho jednoducho spustiť docker start -i luadev. Ak potrebujeme image a kontajner vytvoriť nanovo, spustíme:

```
docker-compose down
docker-compose build --no-cache
docker-compose run --service-ports --name luadev luadev
```

V každom repozitári by mal byť priečinok dev so súbormi lrdb_debug.lua a start_debug.sh. Ak ho niektorý neobsahuje, skopírujeme ho z develop vetvy, prípadne iného repozitára. Táto ukážka je pre použitie luadb, podstatné sú prvé dva riadky, kde sa akivuje debugger.

```
lrdb = require("lrdb_server")
lrdb.activate(21110)

    astManager = require "luadb.manager.AST"
    extractor  = require "luadb.extraction.extractor"
    path = "src"

    astMan = astManager.new()
    extractedGraph = extractor.extract(path, astMan)

    lrdb.deactivate()
```

Nastavenie IDE

Do IDE nainštalujeme rozšírenie LRDB debugger nachádzajúce sa v repozitári devenv. Je možné ho nainštalovať príkazom code --install-extension lrdb-0.3.5.vsix alebo manuálne vo VS Code cez Extensions -> More Actions... -> Install from VSIX. Nakoniec v IDE otvoríme priečinok luadev osahujúci všetky repozitáre.

Spustenie

Súbor lrdb_debug.lua slúži ako 'main', kde je ukážkové použitie danej knižnice. Pri vývoji používame tento súbor ako pre vyvolanie vyvíjanej funkcionality, ale zmeny v ňom necommitujeme. Pre jeho spustenie vykonáme príkaz bash dev/start_debug.sh. V tomto bash skripte sa cyklicky vykonáva lua lrdb_debug.lua, teda nie je potrebné ho vždy manuálne spúšťať. Ukončenie skriptu sa realizuje štandardne pomocou Ctrl+C.

- **Tip**
    - Pri debugovaní používame ako pracovný adresár samotný repozitár, teda napríklad cd /luadev/luadb

Po spustení lrdb_debug.lua je vykonávanie pozastavené a čaká na pripojenie IDE. Pre pripojenie vo VS Code stlačíme klávesu F5.

ZeroBrane Studio

ZeroBrane Studio je oveľa jednoduchšie na nastavenie ako VSCode, avšak doposiaľ sa nám v ňom nepodarilo spojazdniť remote-debugging. Taktiež je VSCode uživateľsky oveľa prívetivejšie prostredie na prácu. Z týchto dôvodov odporúčame použiť skôr VSCode a teda postup popísaný vyššie. Takisto je treba poznamenať, že v tomto IDE je možné debugovať len jazyk LUA, nie TERRA.

Nastavenie IDE

Štandardne je IDE nastavené tak, že pri debugovaní je potrebné mať otvorené všetky súbory, do ktorých môže program počas behu vojsť. Preto musíme pred začatím nastaviť IDE tak, aby volané moduly otváralo automaticky. Toto spravíme v Menu => Edit => Preferences => Settings: User. Otvorí sa nám konfiguračný súbor user.lua, do ktorého na nový riadok pridáme príkaz

```
editor.autoactivate = true
```

a následne zmeny uložíme a môžme súbor zavrieť. Týmto je IDE pripravené na debuggovanie.

Príprava

V IDE si otvoríme ako projekt adresár s LUA skriptami. Ten je v projekte 3dsoftviz uložený na tomto mieste: <cesta-k-projektu>\3dsoftvis_remake\Projects\3DSoftviz\UnityProject\Assets\StreamingAssets\LuaScripts. V podadresári App si vytvoríme súbor debug.lua, do ktorého vložíme nasledovný kus kódu.

```
asset_directory = "<cesta-k-projektu>/3dsoftvis_remake/Projects/3DSoftviz/UnityProject/Assets/StreamingAssets/LuaScripts"

local paths = {
   "/Modules/?.lua",
   "/Modules/?/init.lua"
}

local cpaths = {
   "/Modules/?.dll",
   "/Modules/loadall.dll",
   "/Modules/?.so",
   "/Modules/loadall.so",
   "/Modules/?.dylib",
   "/Modules/loadall.dylib"
}

local function appendPath(path, prefix, pathsToAppend)
   path = path or "."
   for _, pathToAppend in ipairs(pathsToAppend) do
      path = path .. ";" .. prefix .. pathToAppend
   end
   return path
end

package.path       = appendPath(package.path, asset_directory, paths)
package.cpath      = appendPath(package.cpath, asset_directory, cpaths)
```

V prvom riadku zameníme placeholder <cesta-k-projektu> za cestu k 3dsoftviz projektu na našom filesystéme. Následne je IDE pripravené na debugovanie. Kód ktorý chceme debugovať píšeme za vložený kus kódu do súboru debug.lua. Debugovanie funguje rovnako ako v iných IDE.

Lokálny mkdocs

Táto príručka slúži pre vývojárov, ktorí píšu dokumentáciu. Mkdocs je možné nainštalovať na svojom počítači a okamžite vidieť ako vyzerá písaná dokumentácia.

Viac o MkDocs tu.

Inštalácia

Mkdocs je napísaný v jazyku python, preto je potrebné ho mať nainštalovaný. Odporúčame verziu 3.6 alebo 3.7, stiahnutý zo stránky python.org. V čase písania (marec 2020) pod verziou 3.8 nefunguje z dôvodu bugu.

Následne nainštalujeme samotný mkdocs a závislosti.

pip3 install mkdocs mkdocs-material mkdocs-awesome-pages-plugin pygments mkdocs-pdf-export-plugin
pip3 install weasyprint

Balík `weasyprint` je obvykle nutné inštalovať zvlášť prípadne na dva krát, pretože sa nenainštaluje korektne.

Verzie

Je potrebné inštalovať verziu mkdocs 1.0.4 kvôli nepodpore slovenského vyhľadávania v novej verzii

```
pip3 install mkdocs==1.0.4 mkdocs-material==4.5.0 mkdocs-awesome-pages-plugin==2.1.0 pygments==2.4.2 mkdocs-pdf-export-plugin==0.5.5
```

Nastavenie

Do vhodného priečinka naklonujeme repozitár `git clone git@gitlab.com:FIIT/3DSoftVis_Remake/documentation.git`. Následne je možné v `documentation` priečinku spustiť príkaz `mkdocs serve` a otvoriť v prehliadači adresu localhost:8000.

Pred commitom a pushnutím zmien spustíme príkaz `mkdocs build --strict`, ktorý upozorní na nefunkčné linky a podobné problémy, ktoré by zabránili generovaniu dokumentácie v pipeline.

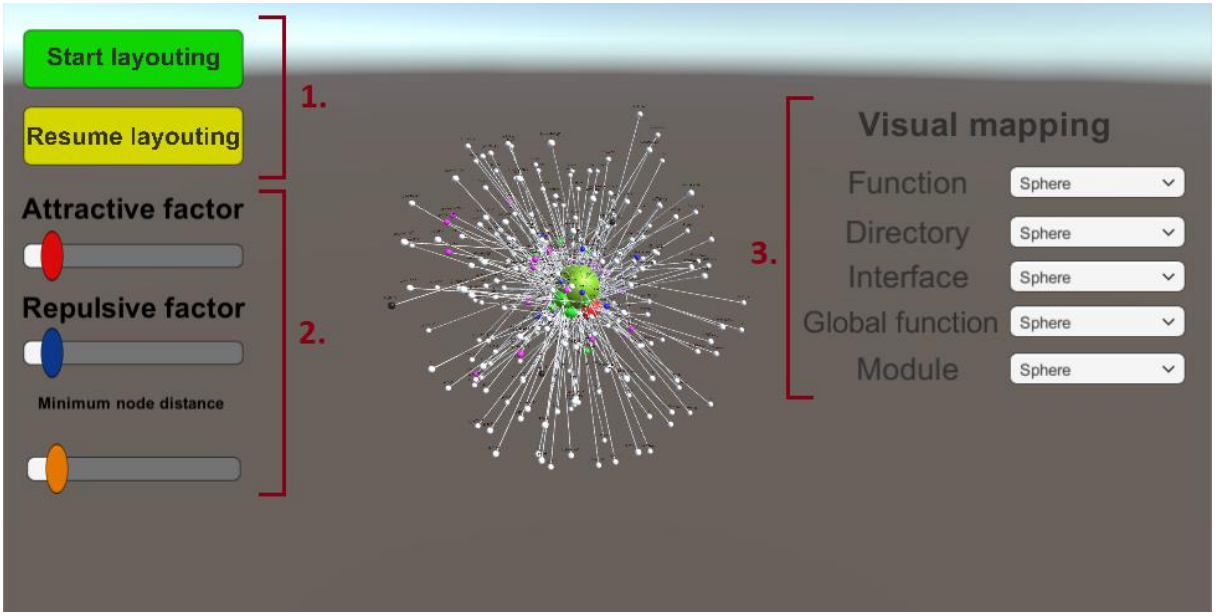
Používateľská príručka

Desktop

Na interakciu so scénou je možné použiť klávesnicu a myš. Pohyb po scéne je riešený klávesami W (dopredu), A (doľava), S (dozadu) a D (doprava). Otáčanie kamery je umožnené pohybom myši v smere želaného otočenia.

Táto scéna disponuje grafickým používateľským rozhraním. Jeho zobrazenie a skrytie sa vykonáva stlačením klávesy M. Toto používateľské rozhranie poskytuje viacero funkcií.

Ovládanie



1. **Layout control** - tlačidlá pre spustenie a zastavenie layoutovania grafu.
2. **Layout settings** - posuvníky nastavení pre Fruchterman-Reinfoldový layoutovací algoritmus (červený - nastavenie veľkosti príťažlivej sily, modrý - nastavenie odpudivej sily, oranžový - nastavenie minimálnej dĺžky hrán grafu).
3. **Visual mapping** - nastavenie vizuálneho mapovania jednotlivých typov uzlov. Pre každý je možné zvoliť rôzne geometrické útvary.

Pri zobrazení menu otáčanie kamery zablokované a myšou je možné:

- **LMB** - označit objekt.
- **CTRL + LMB** - označit viacero objektov.
- **SRCOLL** - zoomovať k / od onačeného(ných) objekt(ov).
- **MMB + pohyb myši** - krúženie okolo označeného(ných) objekt(ov).

Ďalšie klávesy akcií:

- **P** - zapnutie a vypnutie gravitácie.
- **G** - zapnutie a vypnutie hýbania grafu za uzol.
- **F** - zapnutie a vypnutie fyziky.
- **V** - zmena layoutovania grafu.
- **B** - layoutovanie grafu na plochu.
- **K** - prepínanie medzi vizualizovanim uzlov kockou a guľou.
- **J** - prepínanie medzi vizualizovanim hrán kužeľom a hranolom.
- **I** - skrytie hrán grafu.
- **+** - zväčšenie grafu.
- **-** - zmenšenie grafu.

- **O** - označenie všektých popiskov.
- **PgUp** - priblíženie sa k zdrojovému uzlu.
- **PgDn** - priblíženie sa k cieľovému uzlu.
- **Home** - priblíženie sa k označenému uzlu.
- **Z** - zoom-to-fit (priblíženie / oddialenie kamery tak, aby bol viditeľný celý graf).

AR + VR

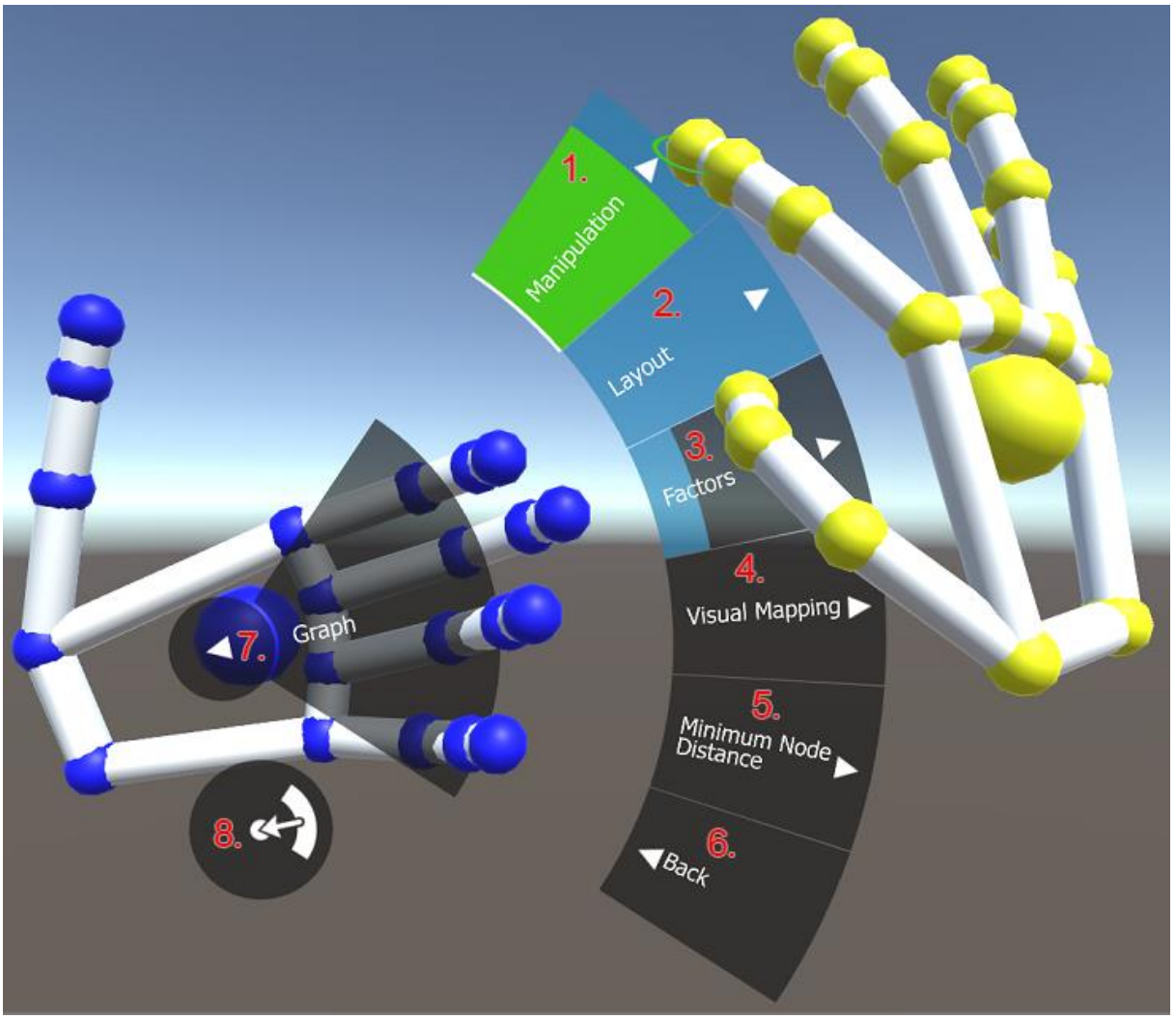
V AR a VR prostredí je možné sa pohybovať prirodzeným spôsobom, tak ako v reálnom prostredí. Headset sa stará o mapovanie pozície v reálnom prostredí do virtuálneho prostredia. Interakcia s prostredím je riešená sledovaním rúk používateľa a ich mapovaním do virtuálneho prostredia. Ovládacie menu je možné otvoriť natočením ľavej dlane smerom k tvári.

Interakcia

V scéne je zapnutá fyzická interakcia. To znamená, že rukou môžeme udierať, uchopiť a hádzať uzlami. Uchopenie je nutné vykonávať troma prstami - palec, ukazovák a prostredník.

HoverUI

HoverUI menu sa zobrazí len pri natočení ľavej dlane smerom na kameru. Stlačenie jednotlivých tlačidiel sa realizuje prostredníctvom priblíženia pravého ukazováka na tlačidlo. Ak tlačidlo zasvieti na svetlo-modro, znamená to, že kurzor na pravom ukazováku sa dostal k jeho blízkosti. Zelená farba znamená, že sa spustila udalosť “kliknutia”. Pomocou tlačidiel je možné manipulovať s celým grafom, kontrolovať layoutovanie, zmeniť jednotlivé faktory grafu alebo zmeniť vizuálne mapovanie.



Vysvetlenie legendy:

1. Manipulation - obsahuje tlačidlá, ktoré sa týkajú manipulácie s grafom (napr. umiestnenie grafu na určité miesto alebo škálovanie grafu).
2. Layout - obsahuje tlačidlá týkajúce sa layoutovania grafu (napr. štart/stop layoutu, pauza layoutu, zmena layouterov, zmena builderov layoutu, zmena layout funkcií a zmena premenných a funkcií).
3. Factors - nastavenie faktorov (attractive a repulsive faktory).
4. Visual Mapping - zmena vizuálneho mapovania pre jednotlivé Lua objekty - funkcie, priečinky, rozhrania, globálne funkcie a moduly. Pre každý typ objektu je na výber zo šiestich tvarov.
5. Minimum Node Distance - nastavenie minimálnej vzdialenosti medzi uzlami.
6. Back - tlačidlo pre návrat do rodičovského menu.
7. ◄ - tlačidlo pre návrat do rodičovského menu. Toto tlačidlo sa taktiež dá stlačiť zovretím ľavej dlane do päste.
8. ·□) - tlačídlo pre skrytie/zobrazenie menu.

Selekcia

Selekciu uzlov a hrán je možné vykonať gestom pištole. Teda natiahnutím ukazováku a prostredníku a stiahnutím prstenníka a malíčka. Palec pri tomto geste slúži na selekciu a deselekciu objektu, na ktorý týmto gestom ukazujeme. Zdvihnutím palca sa objekt označí, zložením sa odznačí.

Obmedzovače

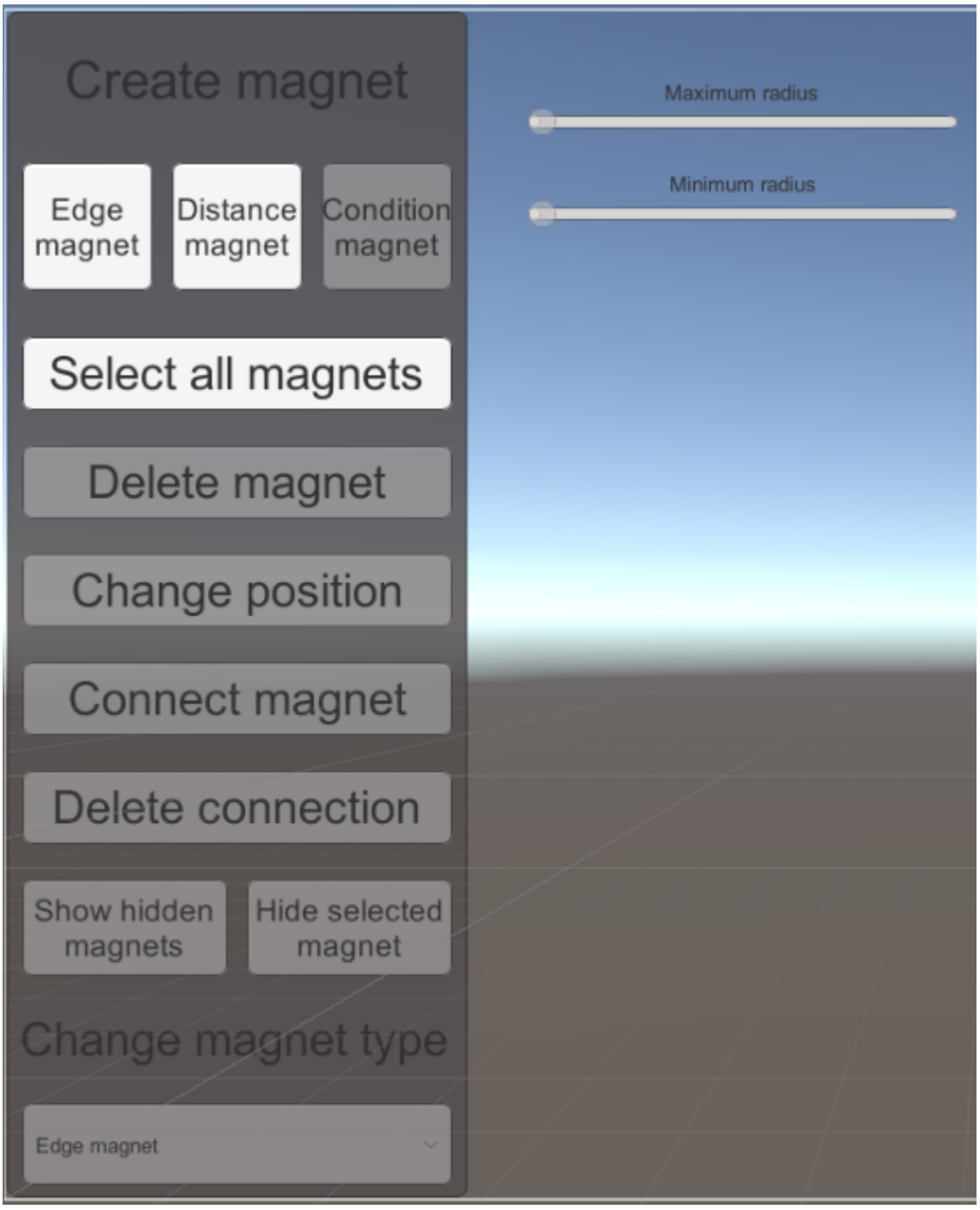
Pre zlepšenie čitateľnosti je možné použiť obmedzovače, ktoré vedia držať a hýbať uzly vo svojom priestore a na ktorých povrch sa uzly viažu. Je možné ich vytvoriť pomocou položky “Spawn restriction” v HoverUI menu. Je možné pridať guľu, kocku, alebo plochu, ktorá sa pred nami vytvorí. Tento obmedzovač je potom možné presúvať a meniť jeho veľkosť. Pre pridanie uzlov do tohto objektu stačí jednoducho uzol chytiť a presunúť ho dovnútra. Pre pridanie viacerých uzlov naraz je potrebné si ich označiť. Pre označenie všetkých uzlov je možné tiež použiť funkciu poskytnutú v HoverUI menu.

Magnety

Magnety slúžia na interakciu s grafom. Do scény je počas behu programu možné pridávať rôzne typy magnetov, ktoré k sebe priťahujú alebo odpudzujú uzly grafu, každý však iným spôsobom. V tejto príručke je opísané, ako pracovať s magnetmi celkovo a zároveň ako pracovať s jednotlivými druhmi magnetov samostatne.

Ovládanie magnetov všeobecne

Do magnetového menu je možné dosať sa z hlavného menu pomocou stlačenia klávesy "M". To znamená, že keď počas behu programu stlačíme klávesu "M", dostaneme sa do hlavného menu. Ak následne opätovne stlačíme klávesu "M", dostaneme sa do magnetového menu. Tretie stlačenie klávesy spôsobí úplné zavretie všetkých menu a návrat do scény.



Pridanie magnetu

Na pridanie magnetu do scény slúži v magnetovom menu horná rada tlačidiel, kde je možné zvoliť si príslušný typ magnetu. Po stlačení tlačidla s názvom príslušného magnetu, ktorý chceme pridať do scény sa magnet zobrazí v scéne v takzvanom "pridávacom móde". Magnet je vtedy priehľadný a vieme s ním hýbať (rovnako, ako sa pohybujeme v scéne - pomocou tlačidiel a myši). Keď magnet dostaneme do pozície, kde ho chceme umiestniť, pre potvrdenie pozície stlačíme Enter. V prípade, že pridávanie magnetu chceme ukončiť, je možné použiť buď tlačidlo Esc alebo tlačidlo "M".

Zmena pozície magnetu

Magnet, ktorý je pridaný v scéne je nehybný. Zmenu jeho pozície však vieme vynútiť tlačidlom v menu "Change position". Po označení magnetu (klik myšou), ktorý chceme presunúť a stlačení tlačidla "Change position" (toto tlačidlo je možné stlačiť len v prípade, že je označený práve jeden magnet) sa dostane magnet do podobného módu ako v prípade pridávania magnetu. S magnetom je vtedy možné hýbať rovnako ako pri pridávaní a taktiež novú pozíciu magnetu potvrdíme klávesou Enter.

Vymazanie magnetu

Magnet (alebo viaceré magnety) je možné vymazať tlačidlom "Delete magnet". Najprv treba označiť magnet/magnety, ktoré chceme vymazať a následne stlačiť tlačidlo "Delete magnet".

Označenie magnetov

Označovanie magnetov funguje rovnako ako označovanie iných objektov v scéne. Po kliknutí na objekt (magnet) sa tento označí. V prípade, že ich chceme označiť viac, je možné ich označovať so stlačenou klávesou Ctrl.

V prípade, že máme v scéne vložených veľa magnetov a chceme nad všetkými naraz vykonať nejakú operáciu (napríklad ich všetky vymazať, alebo schovať), môžeme použiť tlačidlo "Select all magnets", ktorým označíme všetky magnety, ktoré sú aktuálne vložené v scéne.

Skrytie a zobrazenie magnetov

Nie vždy chceme, aby sme mali magnety viditeľné v scéne. Niekedy potrebujeme, aby plnili svoju funkcionalitu, ale nechceme, aby boli viditeľné, aby sme mali graf prehľadnejší. Na ovládanie tejto funkcionality nám slúžia dve tlačidlá - "Hide selected magnets" a "Show hidden magnets". Tlačidlo "Hide selected magnets" funguje podobne ako pri vymazaní magnetov, najprv musia byť označené magnety, ktoré chceme skryť a po stlačení tlačidla sa funkcionalita schovania na tieto označené magnety aplikuje. Tlačidlo "Show hidden magnets" len zobrazí všetky skryté magnety (v prípade, že nejaké také sú). To znamená, že po stlačení tohto tlačidla sú všetky magnety, ktoré sú v scéne, viditeľné.

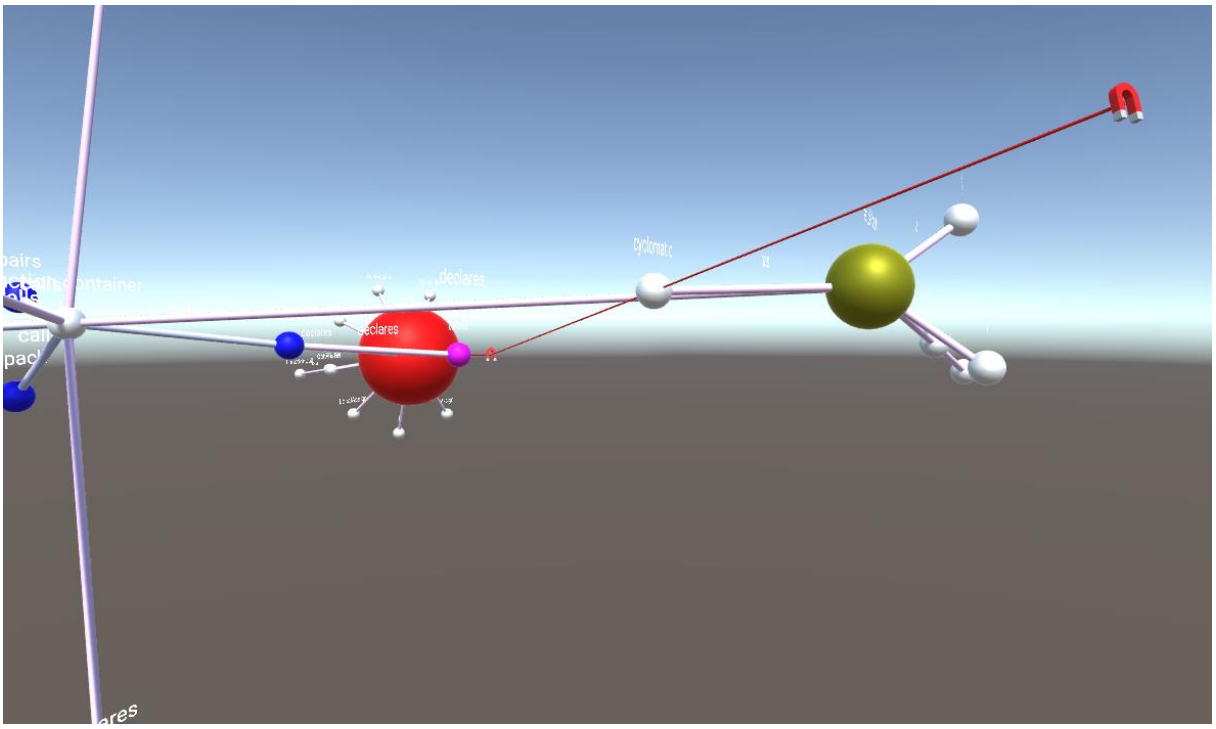
Zmena typu magnetu

Ako bolo spomenuté, magnet môže byť rôzneho druhu. Existujú dokopy 3 druhy magnetov: také, ktoré interagujú s uzlami grafu na základe spojenia pomocou meta-hrany, také, ktoré interagujú s uzlami na základe definovanej vzdialenosti od magnetu a také, ktoré interagujú na základe nejakej definovanej funkcie (podmienky).

V magnetovom menu je predpripravený ovládací prvok na zmenu typu magnetu, avšak táto funkcionalita zatiaľ nie je implementovaná, takže s týmto ovládacím prvkom nie je možné interagovať.

Magnet interagujúci na základe spojenia s uzlami

Tento typ magnetu priťahuje uzly, s ktorými je spojený meta-hranou. Na pridanie tohto magnetu do scény použijeme tlačidlo "Edge magnet". Samotné pridávanie je opísané vyššie vo všeobecnej časti "Pridanie magnetu".



Spojenie magnetov s uzlami

Na to, aby tento typ magnetu priťahoval uzly, ho po pridaní do scény musíme spojiť s uzlami, ktoré chceme, aby priťahoval. Na to slúži tlačidlo "Connect magnet", ktoré môžme nájsť v magnetovom menu. Toto tlačidlo je možné použiť len vtedy, keď je označený práve jeden magnet a zároveň je to magnettypu "Edge magnet". Po stlačení tlačidla sa dostaneme do "spájacieho módu", kedy si pomocou myši môžme vybrať jeden alebo viacero uzlov grafu, s ktorými chceme magnet spojiť. Po označení uzlov tlačidlom Enter potvrdíme voľbu a magnet sa s uzlami spojí hranami. Zároveň magnet hneď začne uzly priťahovať. Spájací mód môžme kedykoľvek ukončiť pomocou klávesy Esc alebo "M". Magnet je možné spojiť s ľubovolným počtom uzlov. Takisto je možné k magnetu, ktorý už je spojený s nejakými uzlami kedykoľvek pridať nové spojenia.

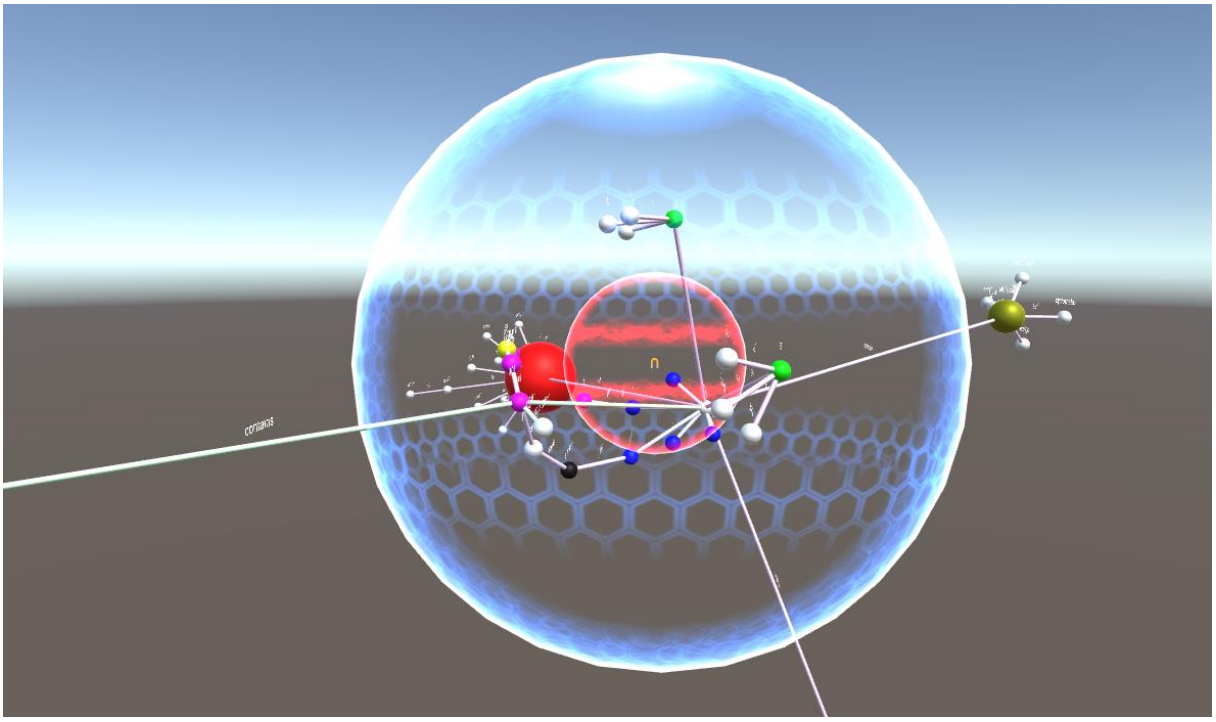
Zmazanie existujúcich spojení

Nepotrebné alebo nevyhovujúce spojenia je možné kedykoľvek odstrániť. Jednak sa všetky spojenia magnetu odstránia, pokiaľ odstránime samotný magnet. Taktiež je možné spojenia odstraňovať aj jednotlivo, bez toho, že by sme museli odstrániť celý magnet. Na toto nám slúži tlačidlo "Delete connection". Tlačidlo je možné použiť, keď je označená jedna alebo viac meta-hrán. Meta hrany označujeme rovnako ako ostatné objekty v scéne kliknutím myšou (prípadne klikaním myšou so stlačenou klávesou Ctrl, ak chceme označiť viacero hrán). Naraz môžme odstrániť meta-hrany patriace aj viacerým magnetom. Po označení

všetkých meta-hrán, ktoré chceme vymazať stačí stlačiť tlačidlo "Delete connection" a týmto sa spojenia odstránia.

Magnet interagujúci na základe vzdialenosti

Ďalším typom magnetu je magnet, ktorý priťahuje alebo odpudzuje uzly na základe vzdialenosti od magnetu. Na pridanie tohto typu magnetu do scény použijeme tlačidlo "Distance magnet". Samotné pridávanie je opísané vyššie vo všeobecnej časti "Pridanie magnetu".



Nastavenie vzdialenosti

Na to, aby tento magnet fungoval mu musíme nastaviť vzdialenosť, v akej má okolité uzly priťahovať. Po pridaní tohto magnetu do scény môžme vidieť, že je okolo magnetu zobrazené modré (maximálna vzdialenosť) a červené (minimálna vzdialenosť) magnetické pole. Tieto vzdialenosti sú na začiatku nastavené na preddefinované malé hodnoty. Tieto hodnoty je možné zmeniť pomocou sliderov, ktoré nájdeme v ľavom hornom rohu obrazovky vedľa magnetového menu. Slidery je možné použiť po označení magnetu, ktorému chceme meniť vzdialenosti. Rovnako, ako pri vyššie spomínanom type magnetu, opäť musí byť označný práve jeden magnet, na to aby sme mu mohli meniť vzdialenosť a zároveň tento magnet musí byť typu "Distance magnet". Pomocou sliderov vieme magnetu nastaviť dve vzdialenosti: modré pole pôsobnosti (maximum radius) - to je také, v ktorom keď sa ocitnú uzly grafu, magnet ich začne k sebe priťahovať, a červené pole pôsobnosti (minimum radius) - to je také, za ktoré sa priťahované uzly grafu nikdy nedostanú, aby sa nepritiahli úplne k magnetu. V konečnom dôsledku to vyzerá tak, že priťahované uzly grafu sa zastavia na povrchu červeného pola pôsobnosti, na ktorom ostanú "nalepené". Z tohto vyplýva, že minimálna vzdialenosť a teda červené pole pôsobnosti nikdy nemôže byť väčšie, ako maximálna vzdialenosť a teda modré pole pôsobnosti. Sliderom "Maximum radius" sa nastavuje veľkosť polí celkovo, t.j. hýbu sa obe polia pôsobnosti, pričom je zachovaný pomer ich veľkostí, a sliderom "Minimum radius" sa nastavuje veľkosť menšieho poľa voči tomu väčšiemu. Po

nastavení veľkostí nie je toto nastavenie nijako potrebné potvrdiť, nastavenie má účinnosť hneď ako je hýbané so slidermi.

Magnet interagujúci na základe funkcie

Tento typ magnetu zatiaľ nie je implementovaný.

Obmedzovače

Obmedzovač je do scény možné pridať prostredníctvom menu. V ňom sa nachádza sekcia Spawn restriction a v rámci nej už konkrétne typy obmedzovačov. Po zvolení sa daný typ obmedzovača zjaví pri našich rukách.

Čo sa týka gizma, tak v menu je možné nastaviť či sa zobrazuje posúvacie a rotovacie gizmo, škálovacie gizmo alebo sú gizmá vypnuté. Gizmo sa automaticky zobrazí okolo objektu, keď sa k nemu dostatočne blízko priblížime rukou.

Pohyb alebo rotácia objektom

V prípade, že máme zvolený typ gizma na pohyb alebo rotáciu, po priblížení sa k objektu sa zobrazí okolo neho biely obrys kocky. Keď tento objekt uchopíme, obrys kocky zmení farbu na žltú a vtedy vieme pohybom ruky objektom hýbať alebo ho rotovať.

Škálovanie objektu

Keď máme zobrazené gizmo pre škálovanie, vidíme okolo neho kocku, na ktorej sú rozmiestnené malé zelené ovládacie kocky. Keď uchopíme rohovú kocku, objekt môžeme zväčšovať po všetkých troch osiach. Vtedy sa zároveň farba konkrétnej zelenej kocky zmení na žltú.

Uchopením prostrednej zelenej kocky na hrane, môžeme škálovať objekt po dvoch osiach. Vtedy zmeníme tvar celkového gizma z kocky na kváder.

Posledným spôsobom škálovania je uchopenie zelenej kocky v prostriedky strany kocky, kedy vieme objekt škálovať po jednej osi. Vtedy kocku taktiež zmeníme na kváder, ktorý akoby naťahujeme.

Pohyb alebo rotácia celým grafom

Keď sa naše ruky nachádzajú v určitej vzdialenosti od celého grafu, gizmo sa zobrazí okolo. Vtedy môžeme uchopením začať hýbať alebo rotovať celým grafom.

Dôležité je poznamenať, že obmedzovače sú na gizmách pomerne závislé. Keď pridáme do scény obmedzovač, tak jeho veľkosť a polohu meníme práve pomocou gizma.

Analýza Moonscript projektov

Bežne sa softvér 3DSoftViz používa na analyzovanie Lua projektov. Jeho implementácia však umožňuje analyzovanie aj Moonscript projektov.

Potrebné kroky pre analyzovanie Moonscript projektov sú:

1. V časti Unity otvoriť súbor GraphLoader.cs
2. V časti, kde sa pridáva konfigurácia - teda v Configuration.Add():
    a. Nastaviť cestu k Moonscript projektu
    b. Nastaviť atribút GraphExtractor na "GraphExtractor.MoonscriptGraph", čo zabezpečí používanie modulu LuaMeg namiesto LuaDb

Dokumentácia k zdrojovému kódu

Pre každý modul existuje zoznam všetkých vetiev, pre ktoré bola vygenerované dokumentácia. Medzi týmito dokumentáciami sa dá jednoducho prepínať podľa potreby. Za hlavnú dokumentáciu je považovaná dokumentácia na vetve `develop` alebo `master`, v prípade, že pre daný modulul nie je vetva `develop` vytvorená.

Pre niektoré moduly nebola ešte na vetvách `master` alebo `develop` vygenerovaná dokumentácia, preto na ne nie je pridaný odkaz.

3D SoftViz

Zoznam všetkých dokumentácií pre 3dsoftviz

Lua Graph

Zoznam všetkých dokumentácií pre LuaGraph

Lua Interface

Zoznam všetkých dokumentácií pre LuaInterface

Luadb

Develop

Zoznam všetkých dokumentácií pre luadb

Luagit

Master

Zoznam všetkých dokumentácií pre luagit

Luameg

Master

Zoznam všetkých dokumentácií pre luameg

Luametrics

Develop

Zoznam všetkých dokumentácií pre luametrics

Luatree

Develop

Zoznam všetkých dokumentácií pre luatree

Časté problémy

Vytváranie objektov starou syntaxou

Niektoré Lua moduly využívajú starú syntax vytvárania objektov v tvare `module(_nazov_modulu_)` namiesto novej syntaxe, ktorá vracia vybrané funkcie a premenné modulu prostredníctvom návratovej hodnoty vo forme tabuľky v nižšie uvedenom tvare.

Vytváranie objektov novou syntaxou

```
local logger
local mojaPremenna
local moja Funkcia

return {
   logger = logger,
   mojaPremenna = mojaPremenna,
   mojaFunkcia = mojaFunkcia
}
```

Výskyt starej syntaxe

Stará syntax je použitá v nasledujúcich moduloch :

- Luametrics
    - metrics/init.lua
    - metrics/rules.lua
    - metrics/captures/ast.lua
    - metrics/captures/block.lua
    - metrics/captures/cyclomatic.lua
    - metrics/captures/document_metrics.lua
    - metrics/captures/functiontree.lua
    - metrics/captures/halstead.lua
    - metrics/captures/hypergraph.lua
    - metrics/captures/infoflow.lua
    - metrics/captures/LOC.lua
    - metrics/captures/statements.lua
    - metrics/luadoc/captures.lua
    - metrics/luadoc/commentParser.lua
- Luameg
    - meg.lua