Fakulta informatiky a informačných technológií STU v Bratislave

Ilkovičova 2, 842 16 Bratislava 4

Princípy počítačovej grafiky a spracovania obrazu

Téma projektu: **Hrad**

Projekt CastleCall

Marek Klanica & Ondrej Špánik  
ID: 96914 & 103151

Meno cvičiaceho: Ing. Lukáš Hudec, PhD.

Časy cvičení: Utorok 16:00-17:50

Akademický rok: 2021/22 ZS

Obsah

[1. Dátové štruktúry 3](#_Toc90500155)

[Objekty a konverzia scény 4](#_Toc90500156)

[2. Algoritmy 6](#_Toc90500157)

[Procedurálne generovanie scény 6](#_Toc90500158)

[Vhodný spôsob osvetlenia 6](#_Toc90500159)

[Tiene 7](#_Toc90500160)

[Post-process bloom efekt 7](#_Toc90500161)

[3. Scény, priestorové vzťahy 9](#_Toc90500162)

[Zmena scén 9](#_Toc90500163)

[*Poznámka o UV* 10](#_Toc90500164)

[*Prechod z prvej scény do druhej: Brána* 10](#_Toc90500165)

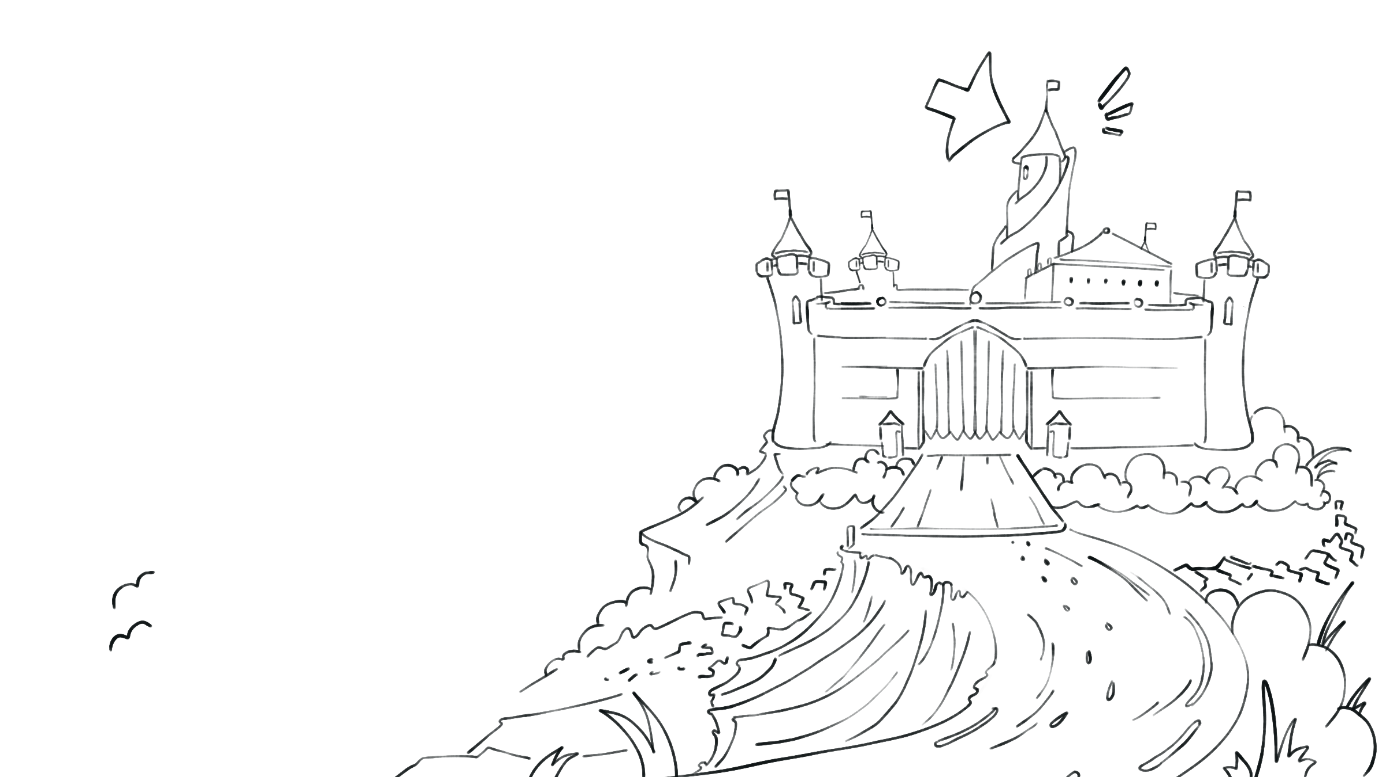
[*Prechod z druhej scény do tretej: Hala* 11](#_Toc90500166)

[Hierarchické transformácie 11](#_Toc90500167)

[4. Diagram tried objektov scény 12](#_Toc90500168)

[5. Doplnky 12](#_Toc90500169)

[Mapa návrhu vs implementácie 12](#_Toc90500170)

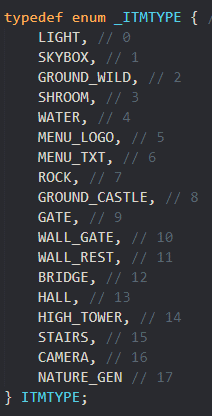
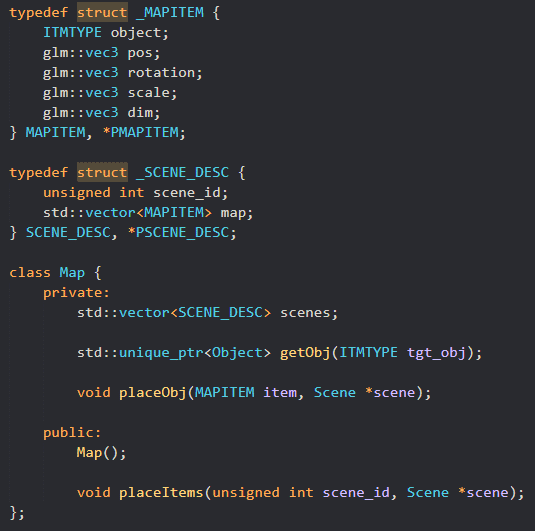
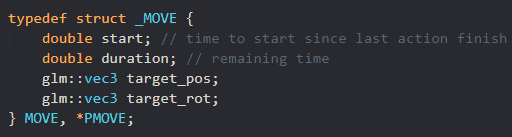


# Dátové štruktúry

Počas tvorby nášho projektu sme zvolili prístup aplikovania vlastných dátových štruktúr podľa jednotlivých bodov hodnotenia projektu, podľa potreby bližšie popísať objekty a podľa potrieb vlastnej implementácie.

Väčšina dát aplikovaných v našom projekte je uložená za použitia existujúcich dátových typov v glm knižnici ako sú vec3. Keďže je projekt implementovaný v C++, ktoré je od C rozšírené o podporu tried, tak ich bohato využíva vo viacerých aplikovaniach.

Mapa sa načítava z externého súboru map.txt. V súbore s mapou sú zadefinované typy objektov, ktoré sa musia zhodovať s enum ITMTYPE. Každý platný riadok mapy sa uskladní do štruktúry MAPITEM, ktorá sa priradí k svojej scéne v štruktúre SCENE\_DESC.



## Objekty a konverzia scény

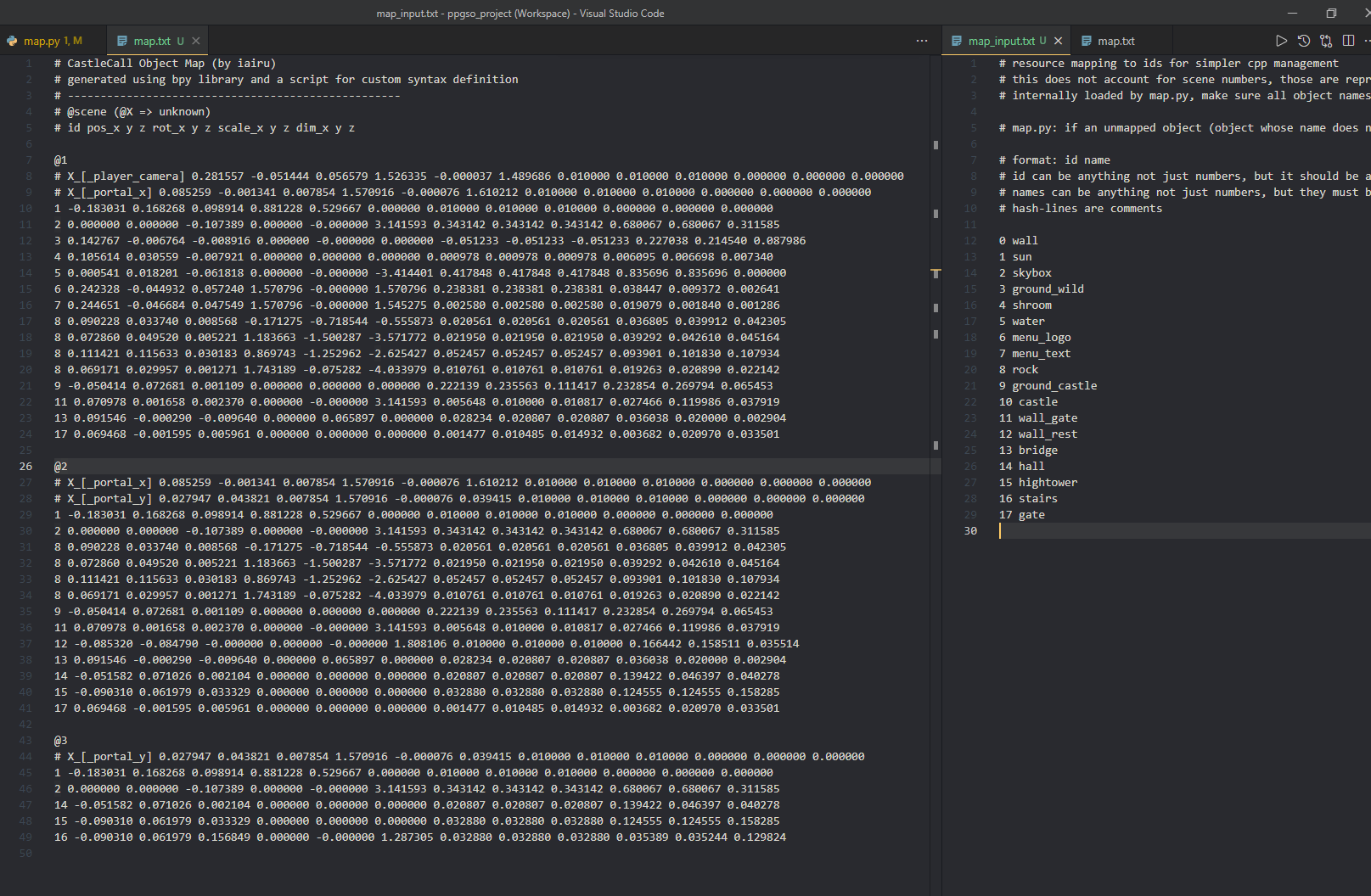
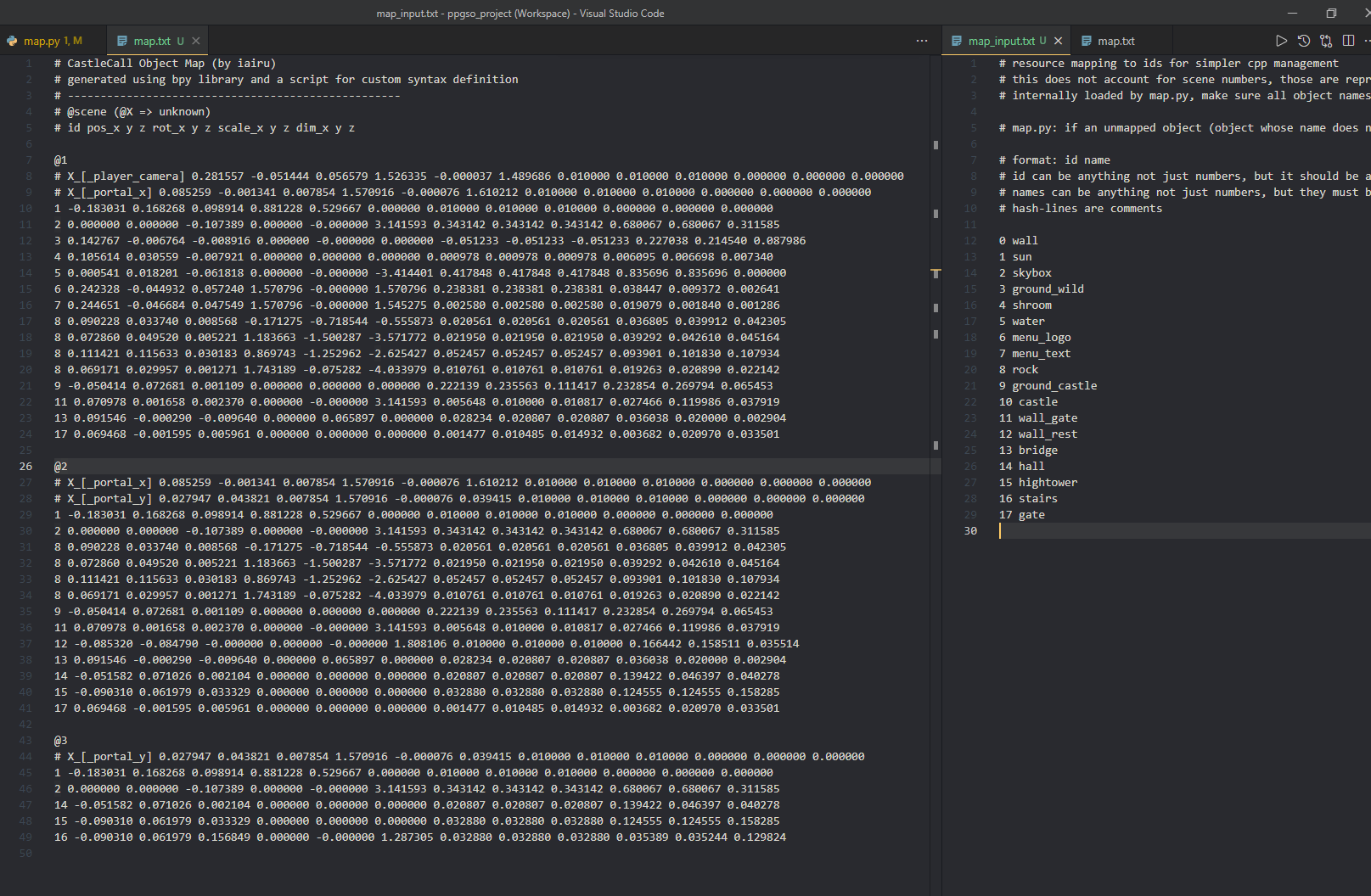
Predtým, ako priblížim rozloženie tried spomeniem **spôsob, ktorým je v projekte implementované načítavanie objektov a dátové štruktúry, ktoré museli byť aplikované za týmto cieľom:**

Načítavanie objektov prebieha v map.cpp za použitia vlastného enumerátora a switcha. Prvý priraďuje všetkým druhom statických objektov ľudsky čitateľné pomenovanie a je využitý najmä z hľadiska zjednodušenia programovacích nárokov. Druhý na základe prvého vytvorí unikátne inštancie jednotlivých objektov počas volania z cyklu nižšie, ktorý nájde svoje využitie pri spustení aplikácie.

Za dátovú štruktúru sa ďalej dá považovať spôsob uloženia objektov a informácií o ich precíznych transformáciách, k čomu okrem map.cpp slúži aj utilita **map.py**, špecificky pre **automatizovaný export týchto detailov z prostredia Blender scény do syntakticky-vlastného map.txt** súboru, ktorého uložené dáta sú na objekty aplikované počas ich tvorby a priraďovania scéne v map.cpp.

Spomenuté uloženie objektov je vykonané do Wavefront .obj súborov priamo z Blenderu, pričom každý súbor reprezentuje unikátny druh objektu s vlastným UV. Každý súbor je uložený bez textúr, tie sú pridané separátne až po jeho načítaní v aplikácií – obsahuje avšak údaje potrebné k ich presnému UV mappingu.

Dáta objektov vrátane UV sú uložené v zložke data, pri builde CMake cache sú ďalej kopírované do zložky res a odtiaľ patrične aplikované v projekte. Zdrojové súbory pre generovanie sa nachádzajú v zložke blender.



# Algoritmy

Počas tvorby projektu sme pristúpili aj k bodu procedurálneho generovania scény, výberu vhodného spôsobu osvetlenia, spôsobu tvorby tieňov a k aplikovaniu post-process bloom efektu.

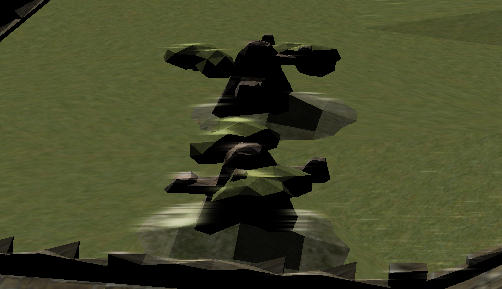
Sumárny obrázok

## Procedurálne generovanie scény

Pre procedurálne generovanie sme zvolili **časť existujúcej scény**, konkrétne nádvoria hradu, kde sme sa rozhodli procedurálnym generovaním automaticky **pridať stromy na vhodné lokácie** tak, aby pripomínal čistinky a husté časti reálneho lesa. Pozície stromov sú náhodne určené pri každom spustení aplikácie.

**Stromy**, ktoré sú v lese použité sú všetky **skladané pomocou hierarchických transformácií** jednotlivých častí, pôvodne uložených externe, ako sú koruna stromu, listové časti stromu a kopček (podlaha) pod stromom.

Každé spustenie funkcie na generovanie stromov vygeneruje 1 alebo 2 stromy na určenej oblasti. Na to, aby sa vygeneroval aj druhý strom musí byť oblasť dostatočne veľká a prvý vygenerovaný strom sa musí nachádzať dostatočne ďaleko od stredu náhodne generovanej oblasti.



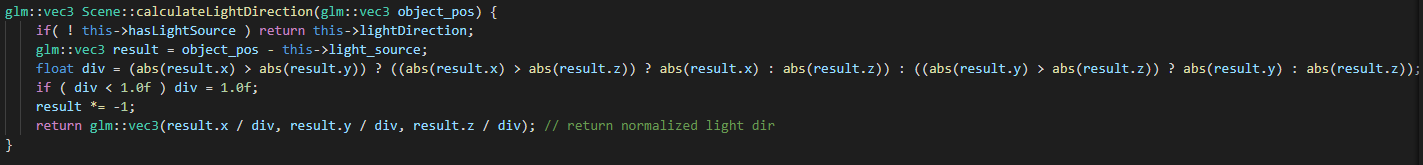
Časť kódu // no to neviem či chceš, reálne je tam len sprasené random device a pár náhodných variables kde sa mi nechcelo vymýšľať názvy

## Vhodný spôsob osvetlenia

Diffuse lighting // add description later || paste wiki

Smer osvetlenia pre každý objekt v scéne sa počíta v reálnom čase a je závislý na umiestnení zdroja svetla. Scéna aktuálne podporuje len jeden zdroj svetla, ktorý je možné meniť. V prípade, že zdroj svetla neexistuj, funkcia pre výpočet smeru svetla vráti predvolenú hodnotu.

Obrázok



## Tiene

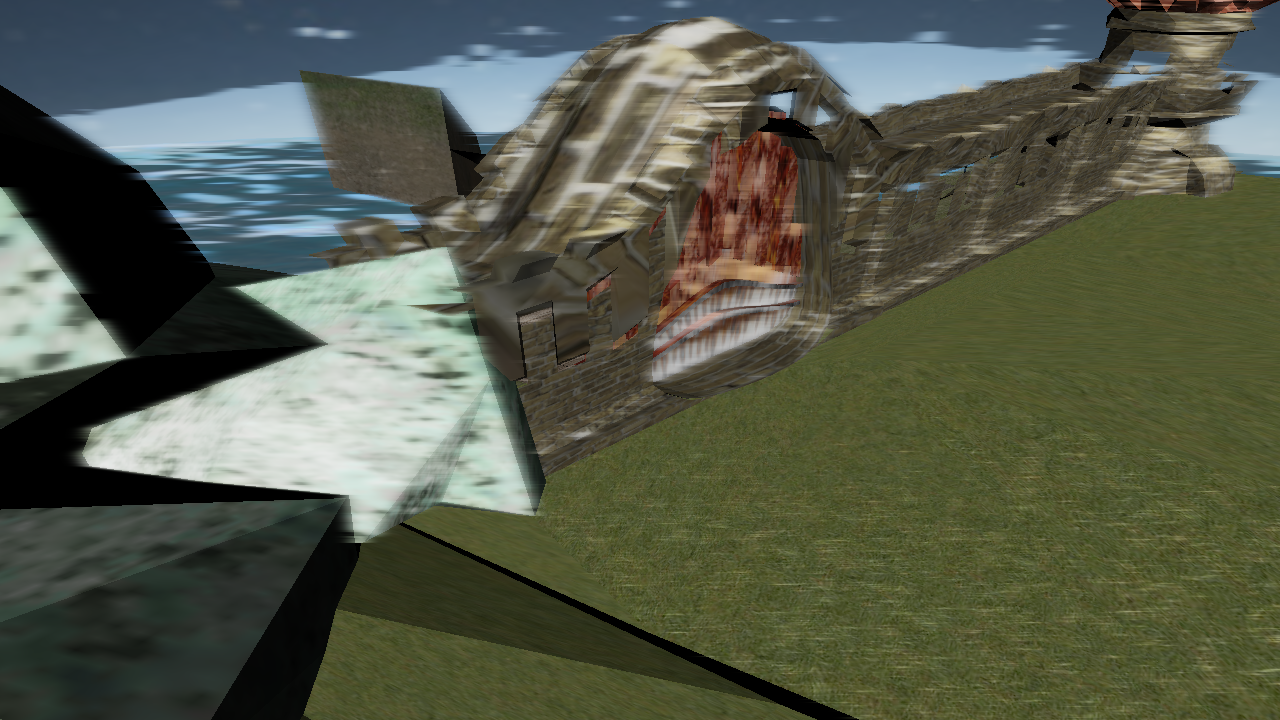
Názov a opis algoritmu

Obrázok

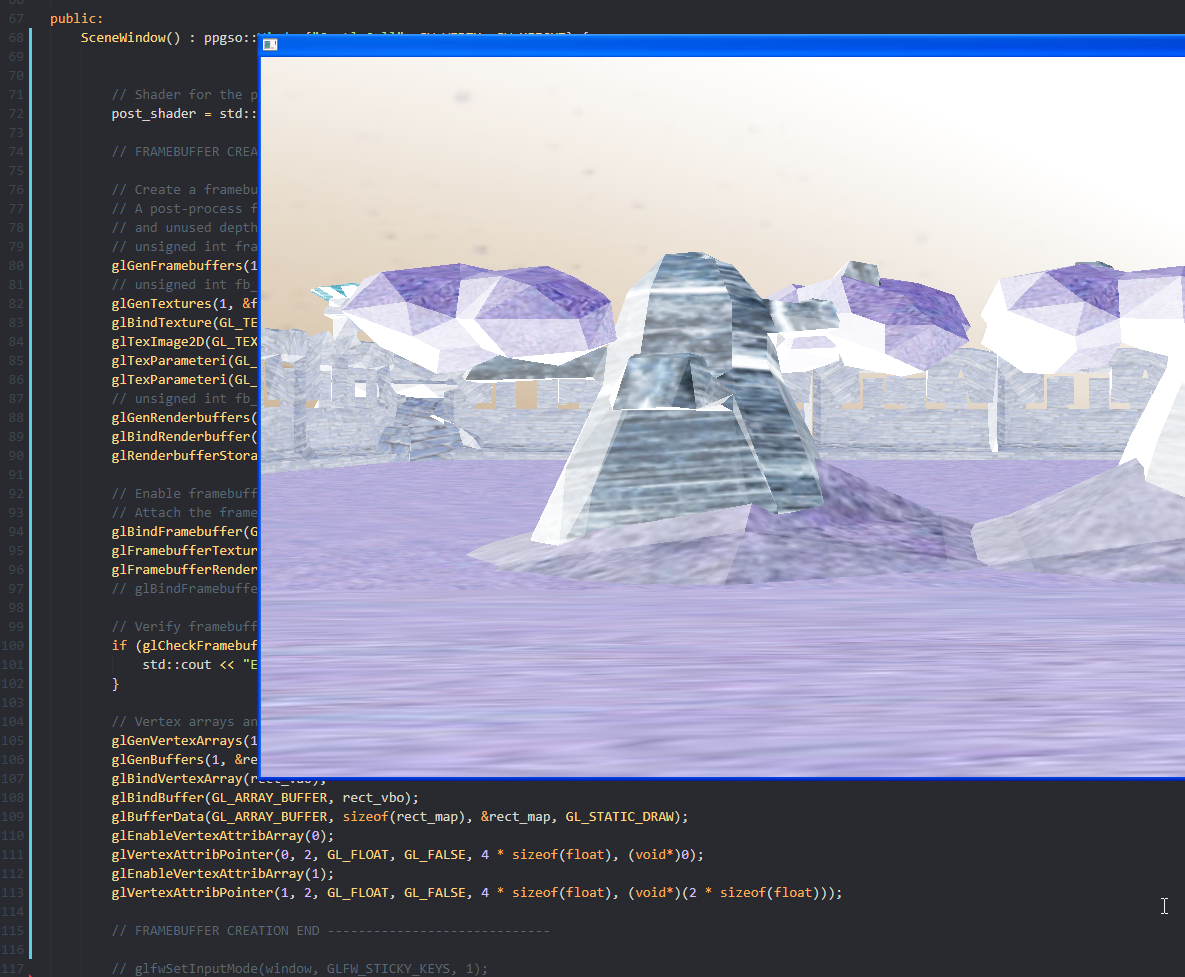
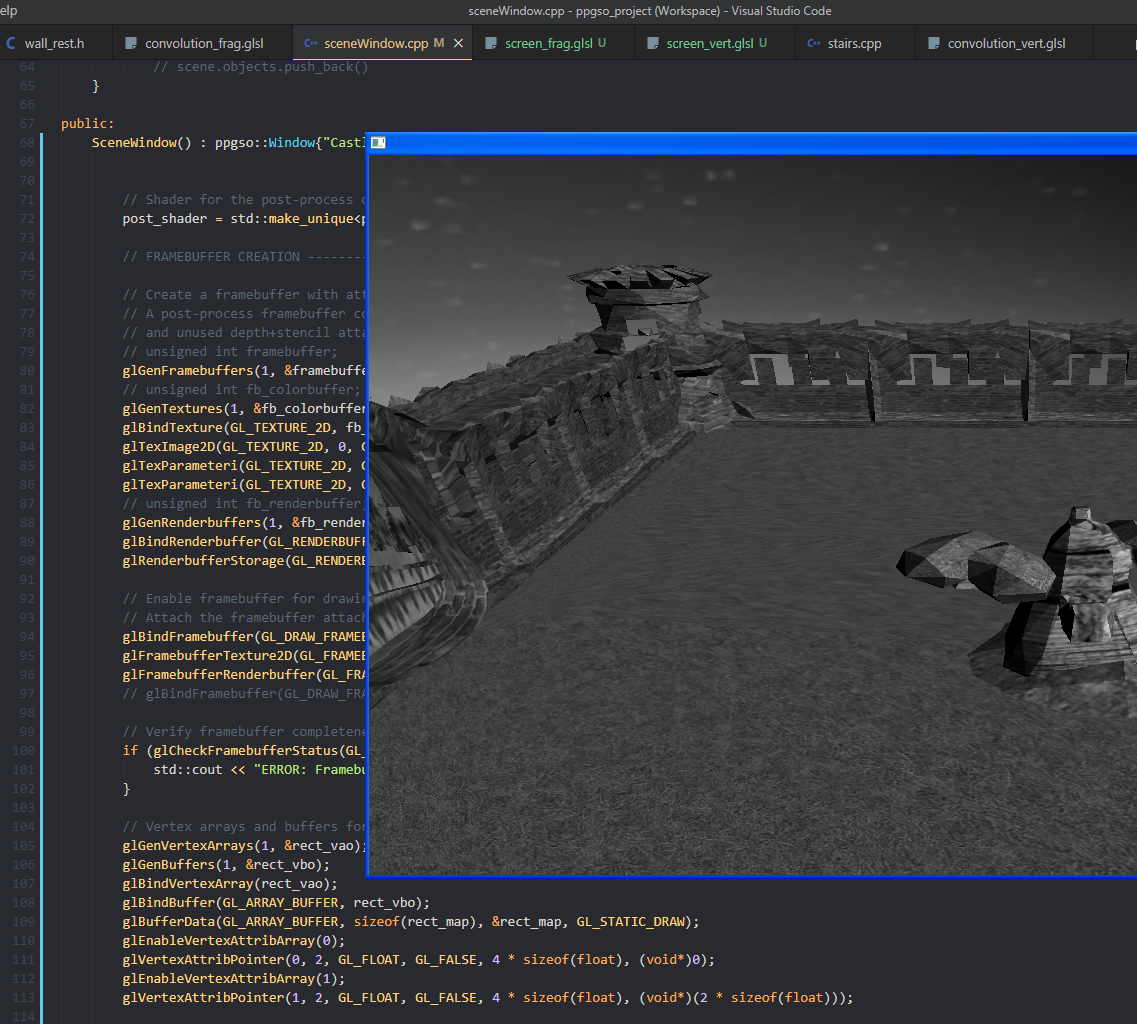
Časť kódu

## Post-process bloom efekt

Pre aplikovanie bloom efektu je potrebné extrahovať z obrazu scény dáta o bledých častiach ako kontrastný obraz, aplikovať naň rozmazanie (druhu gaussian blur) a následne kompozitne spojiť pôvodné dáta s rozmazanými kontrastnými bledými časťami, na ktoré je preto aplikovaný lighten blend mód.



Pred samotným bloom efektom, ktorého výslednú podobu je možné vidieť vyššie som implementoval grayscale/invert efekt pomocou vlastného framebuffera+renderbuffera a vlastného screen shadera.

K bloom implementácií došlo najmä v súbore sceneWindow.cpp v nasledovnom poradí:

**Konštruktor**

* inicializácia renderbuffera a framebuffera
* inicializácia colorbufferov (resp. textúr do ktorých sa buffer zapíše a ktorých obsah sa upraví shaderom)

**onIdle metóda**

* pred scene.update a scene.render bola pridaná zmena framebuffera z predvoleného (0) na vlastný
* následne prečistenie buffer bitov, zapnutie depth testu (len pre prvý buffer, ktorý reálne renderuje 3D priestor, zvyšné pracujú v 2D priestore a teda je u nich depth test/buffer vypnutý)
* aplikovanie shadera pri prvotnom 3D nie je, nakoľko jednotlivé objekty majú v sebe obsiahnutú inštanciu shadera, ktorú majú pre svoje potreby použiť, toto sa ale mení pri ďalších (už 2D) framebufferoch, kde sú pomocou jednotlivých shaderov postupne aplikované efekty

**Poradie framebufferov**

* 3D **scénový** framebuffer „framebuffer\_default“, **render** do colorbuffera „fb\_colorbuffers\_default[0]“
* 2D framebuffer „framebuffer“ pre **prvý efektový shader** „post\_shader\_bright“, ktorý vyberie kontrastné (**dostatočne bledé) časti obraz**u do separátneho colorbuffera „fb\_colorbuffers[0]“
* 2D framebuffer „pingpongFBO“, ktorého úlohou je **viac-násobne aplikovať** shader „post\_shader\_blur“, ktorý aplikuje **gaussian blur** – musí byť riešený v dvoch smeroch (horizontálne/vertikálne) separátne, pričom prvotne použije „fb\_colorbuffers[0]“ a následne svoj vlastný colorbuffer
* **posledný** 2D framebuffer, teraz už ten predvolený (0) je využitý pre aplikáciu „post\_shader\_blend“ shadera, ktorý zoberie 2 textúry z existujúcich colorbufferov – výstupov „post\_shader\_bright“ a „post\_shader\_blur“, **spojí** ich a dodatočne aplikuje gamma/exposure korekciu

**Použité fragmentové screen shadere**

* screen\_blend\_frag.glsl – Spojenie výsledkov ...bright... a ...blur... shaderov => bloom výstup
* screen\_blur\_frag.glsl – Aplikovanie iterácie gaussian blur horizontálne alebo vertikálne
* screen\_bright\_frag.glsl – Výber svetlých častí
* screen\_pass\_frag.glsl – Bez aplikovania efektov, využívaný najmä pre debugging, obsahuje prvotný grayscale a invert efekt v jednom riadku

# Scény, priestorové vzťahy

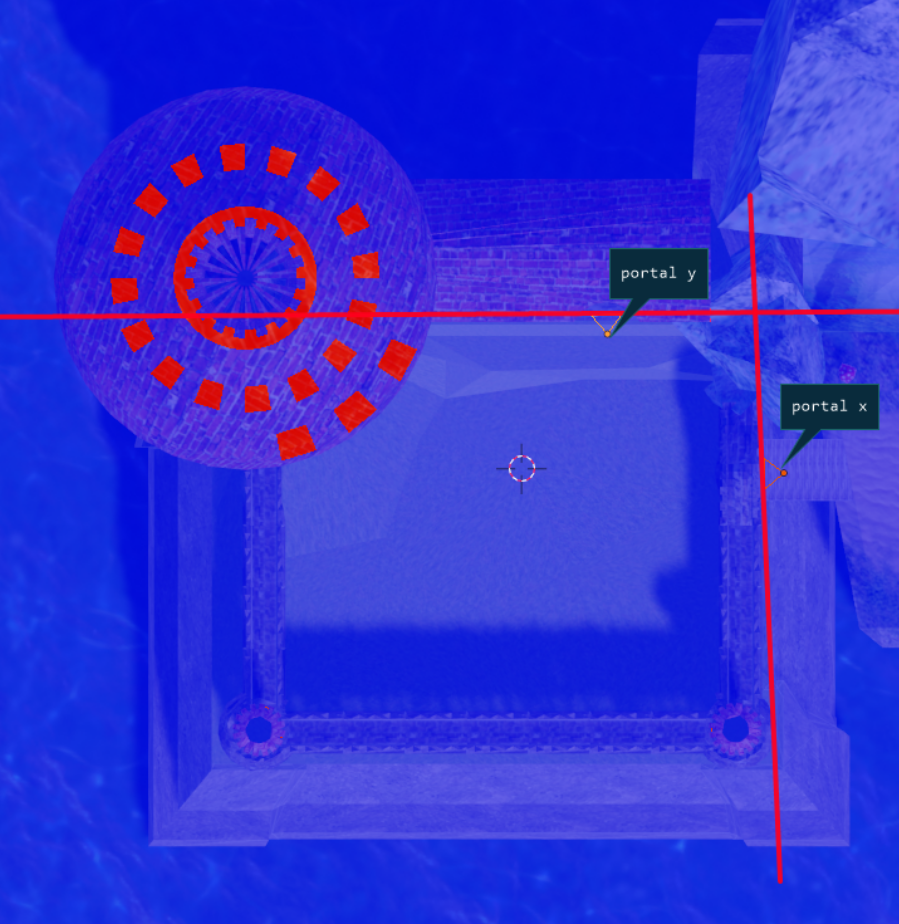
V tejto časti by som bližšie priblížil spôsob, ktorým sú scény menené, ako sú dáta o zmene scén uložené v Blenderi a následne z neho exportované pre použitie v projekte. Ďalej opäť spomeniem hierarchické transformácie a na záver ukážem štatistické údaje.

## Zmena scén

Zmena scén prebieha internou výmenou aktívnych objektov, pričom časť objektov je aktívna medzi 2-3 scénami naraz, jedná sa o vymedzovacie objekty ako hradisko hradu alebo prechodné objekty ako most z prvotnej scény. Tieto prechodné objekty sú takto definované za účelom:

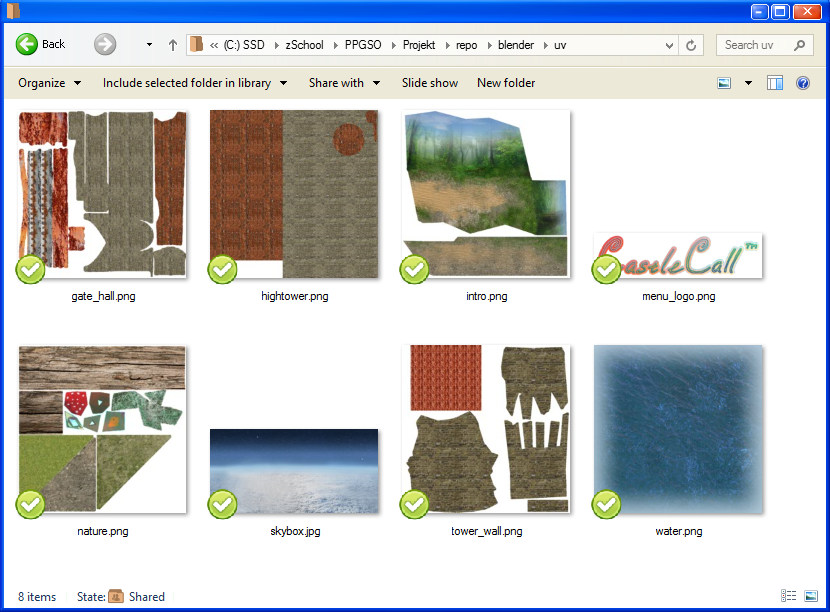
* vnesenia pocitu fluidity hráčovi,
* zaručeniu, že má stále pod sebou podlahu (aby nezačal padať do nekonečna)
* načítavanie objektov novej scény môže byť uskutočnené dynamicky na základe smerového vektora hráča a jeho rýchlosti na pozadí namiesto zobrazenia obrazovky načítavania, ktorá by oddialila hráča od akcie

Miesta zmeny scén boli uvažované ako neviditeľné portálové objekty bez kolízie, cez ktoré keď hráč prejde tak by spustil akciu načítania na pozadí aplikácie. Vďaka strategickému rozloženiu objektov na scénach toto ale nie je potrebné do veľkej miery riešiť, a stačí vymedziť x,z súradnice priamky, po ktorej prechode sa načíta nová scéna a odčíta stará. Túto implementáciu je ďalej nutné rozšíriť o limitné podmienky (meniace priamku na úsečku) aby sa jednalo len o časť scény, nakoľko aj napriek strategickému rozloženiu objektov nemôže byť všetko ideálne.



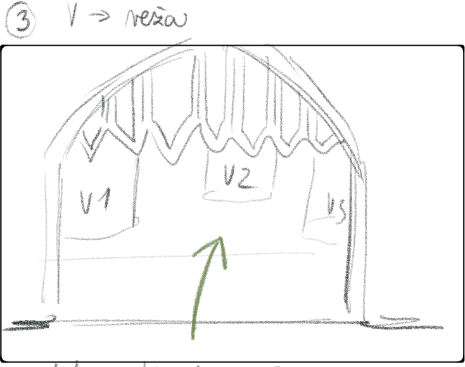
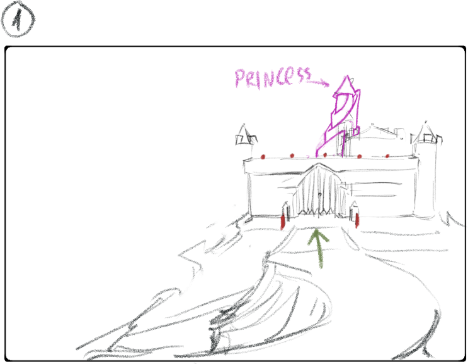
## *Poznámka o UV*

V pôvodnom pláne bola tiež zámena polygonálne zjednodušených objektov za kompletné pre lepšiu optimalizáciu aplikácie, to by avšak predĺžilo prácu na nebodovateľných častiach projektu, ktoré sú už aj napriek tomu obšírne, keďže všetky scény sú zložené z plne vlastnoručne tvorených objektov a UV máp. Z novo nadobudnutej skúsenosti môžem povedať, že UV rozklad dokáže zabrať viac času ako modelovanie, ak sa jedná len o jemne komplexnejší objekt.



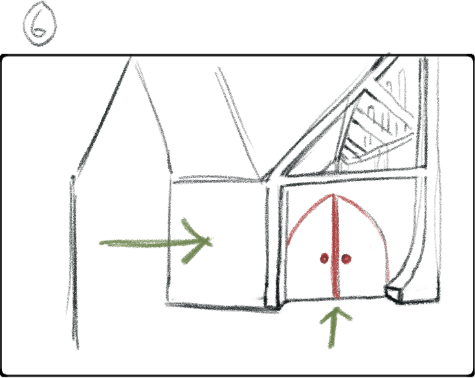
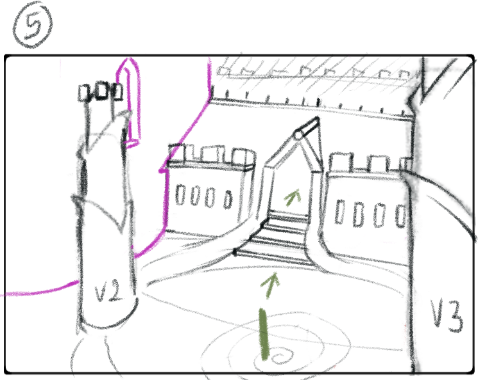
## *Prechod z prvej scény do druhej: Brána*

Tento prechod zaručí načítanie zvyšku objektov na nádvorí ako je zadná časť hradieb, hala alebo les.



## *Prechod z druhej scény do tretej: Hala*

Tento prechod načíta schody nachádzajúce sa vo vysokej veži.

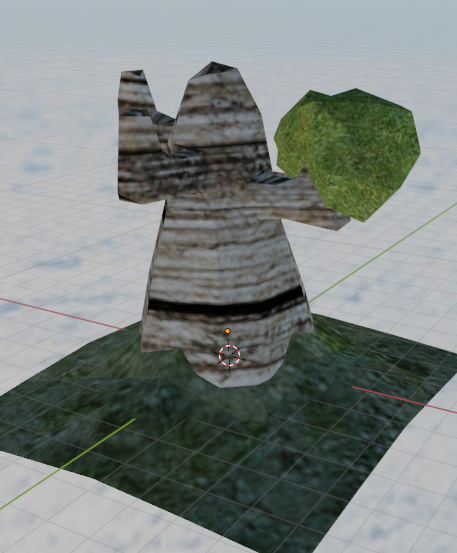
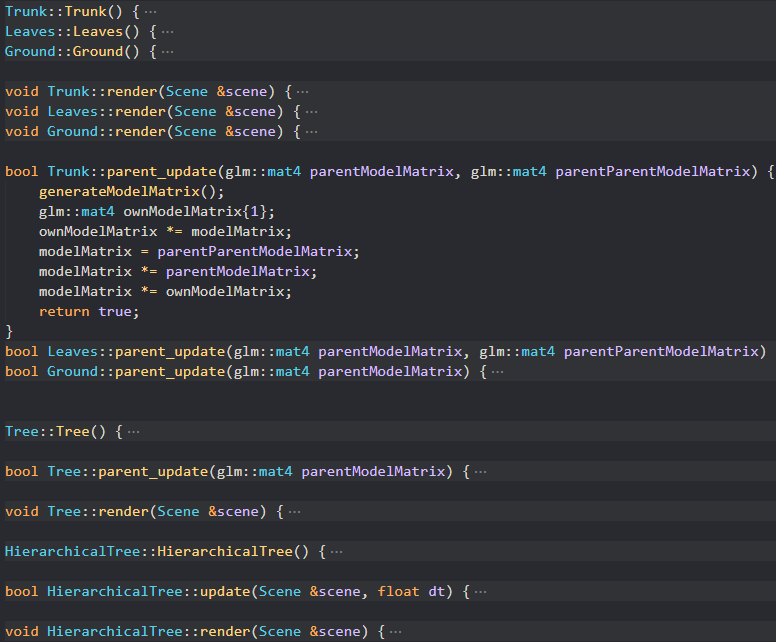


## Hierarchické transformácie

Spomenuté boli s procedurálnym generovaním scény, kde sú aplikované na jednotlivých stromoch procedurálne generovaného lesa nasledovne:

* hierarchický strom (HierarchicalTree) sa skladá zo stromu ako takého (private Tree) a kopčekovej podlahy pripomínajúcej korene pod zemou (private Ground)
* strom ako taký (private Tree) sa ďalej skladá z koruny (private Trunk) a viacerých inštancí listov (private Leaves)

Pre vhodné aplikovanie hierarchických transformácií stačí strategicky umiestniť reálne objekty (Trunk, Leaves, Ground) na predstavenej virtuálnej scéne (resp. origin bode akejkoľvek dočasne použitej scény) tak, aby spolu tvorili plnohodnotný celok (teda aby listy boli na správnych miestach stromu). Tieto transformácie sú vykonané nad jednotlivými ModelMatrix-ami samotných objektov Trunk, Leaves z bodu Tree a pre Ground z bodu HierarchicalTree. Samotné hierarchické objekty nemajú vlastný ModelMatrix, využívajú sa len pre transformácie nad existujúcim ModelMatrixom a tvoria rozhranie pre masové aplikovanie transformácií (nad objektami ktoré obsahujú) pre rodičovskú časť hierarchie, ktorá ich vidí ako bežné objekty s možnosťou transformácie.



# Diagram tried objektov scény

Hierarchia tried, rovnako ako aj hierarchia súborov je strategicky rozdelená pre lepšiu čitateľnosť a pozorovateľnosť. Jednotlivé objekty scény sme sa rozhodli v elementárnej podobe – aj napriek identickému kódu – rozdeliť do vlastných súborov pre jednoduchú rozšíriteľnosť, resp. demonštrovateľnosť potrebnej implementovanej funkcionality jednotlivých bodov hodnotenia.

Načrtnúť hierarchiu tried v draw.io

Ukážky kódu

# Doplnky

## Mapa návrhu vs implementácie

|  |  |
| --- | --- |
| C:\Users\Ondrej\AppData\Local\Temp\msohtmlclip1\02\clip_image012.png | Todo blender screenshot birds eye view |

