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](#_Toc91084371)

[Objekty a konverzia scény 4](#_Toc91084372)

[2. Algoritmy 6](#_Toc91084373)

[Procedurálne generovanie scény 6](#_Toc91084374)

[Osvetlenie - Phongov model a diffuse materiály 6](#_Toc91084375)

[Shadow-maps tiene 7](#_Toc91084376)

[Selektívne kolízie boxami 7](#_Toc91084377)

[Simulácia gravitácie 7](#_Toc91084378)

[Simulácia vetra 7](#_Toc91084379)

[Particle systém a vznik/zánik za behu – Padajúce lístie 8](#_Toc91084380)

[Framebuffery - Post-process Bloom efekt 8](#_Toc91084381)

[3. Scény, priestorové vzťahy 11](#_Toc91084382)

[Zmena scén 11](#_Toc91084383)

[*Poznámka o UV* 12](#_Toc91084384)

[*Prechod z prvej scény do druhej: Brána* 12](#_Toc91084385)

[*Prechod z druhej scény do tretej: Hala* 13](#_Toc91084386)

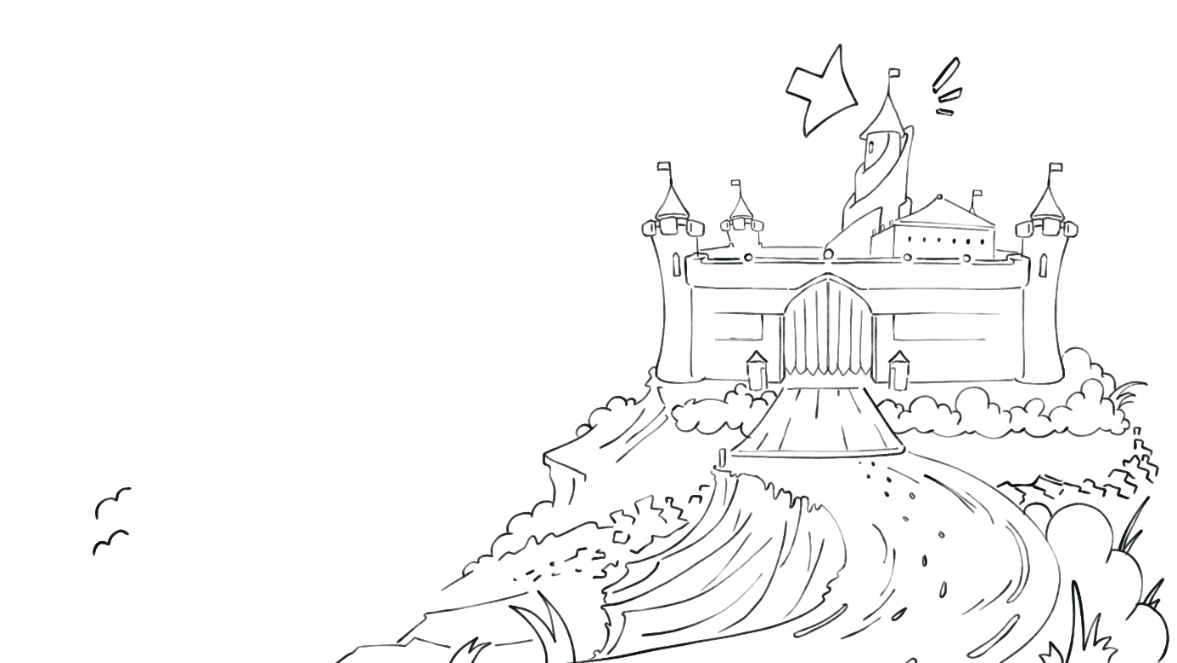
[Hierarchické transformácie 13](#_Toc91084387)

[4. Diagram tried objektov scény 14](#_Toc91084388)

[5. Doplnky 15](#_Toc91084389)

[Mapa návrhu vs implementácie 15](#_Toc91084390)

[Rozdelenie práce 15](#_Toc91084391)

update

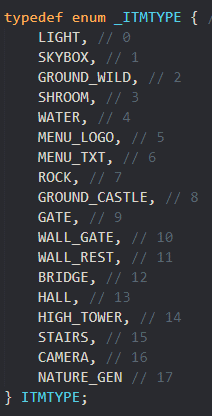
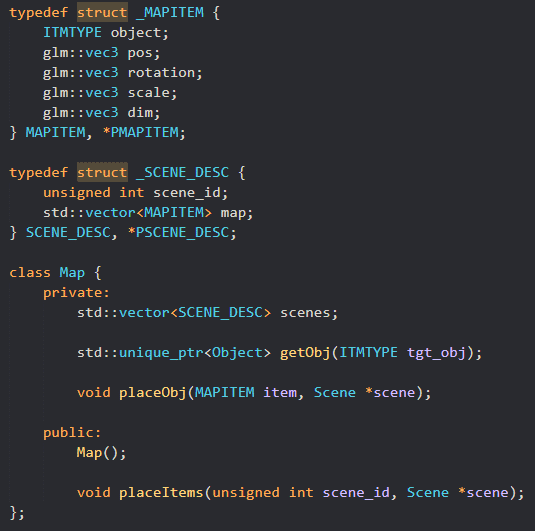
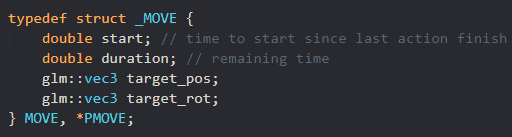
# Dátové štruktúry

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.

spomenúť opengl dátové štruktúry, ... ?



## Objekty a konverzia scény

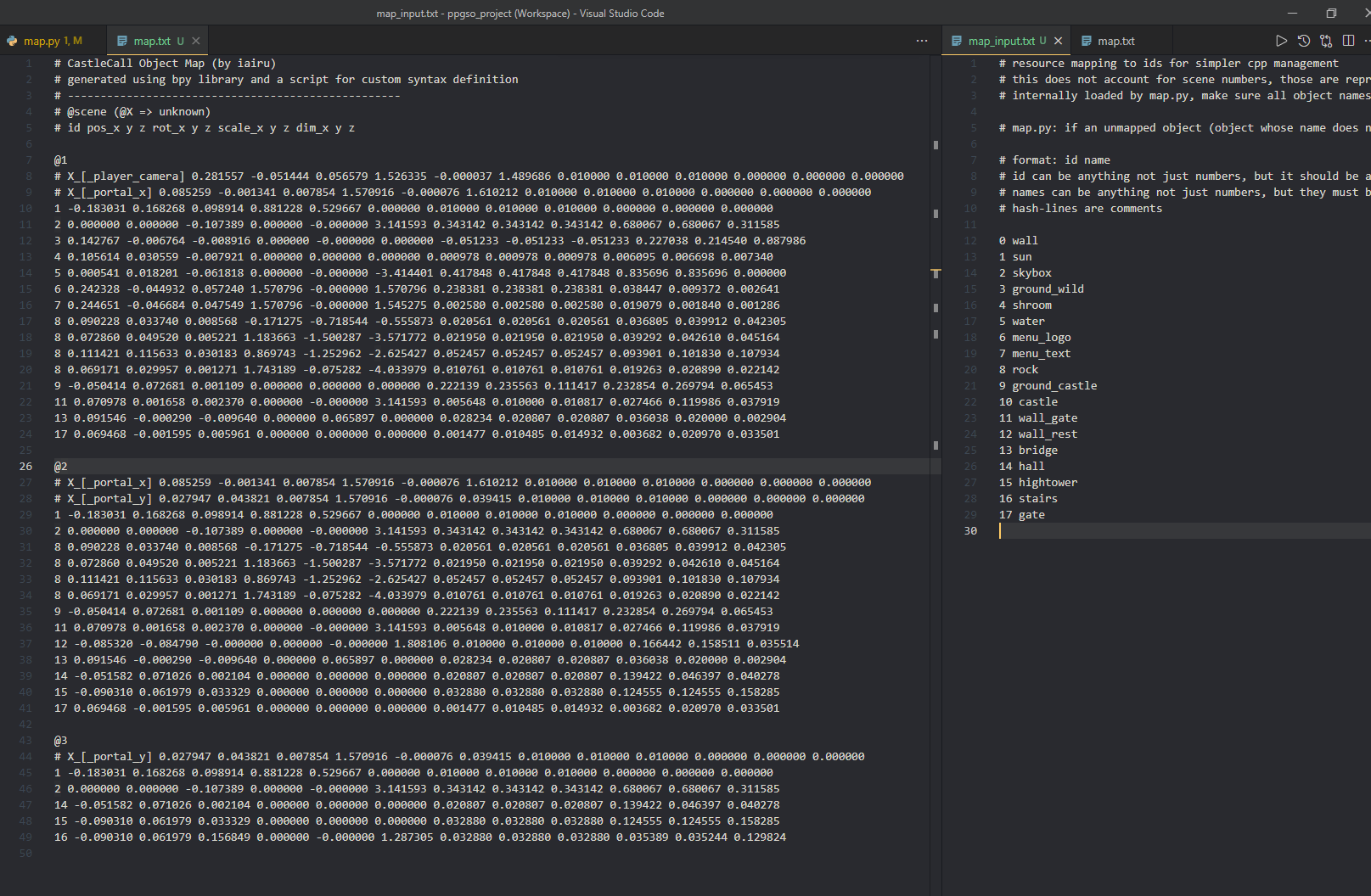
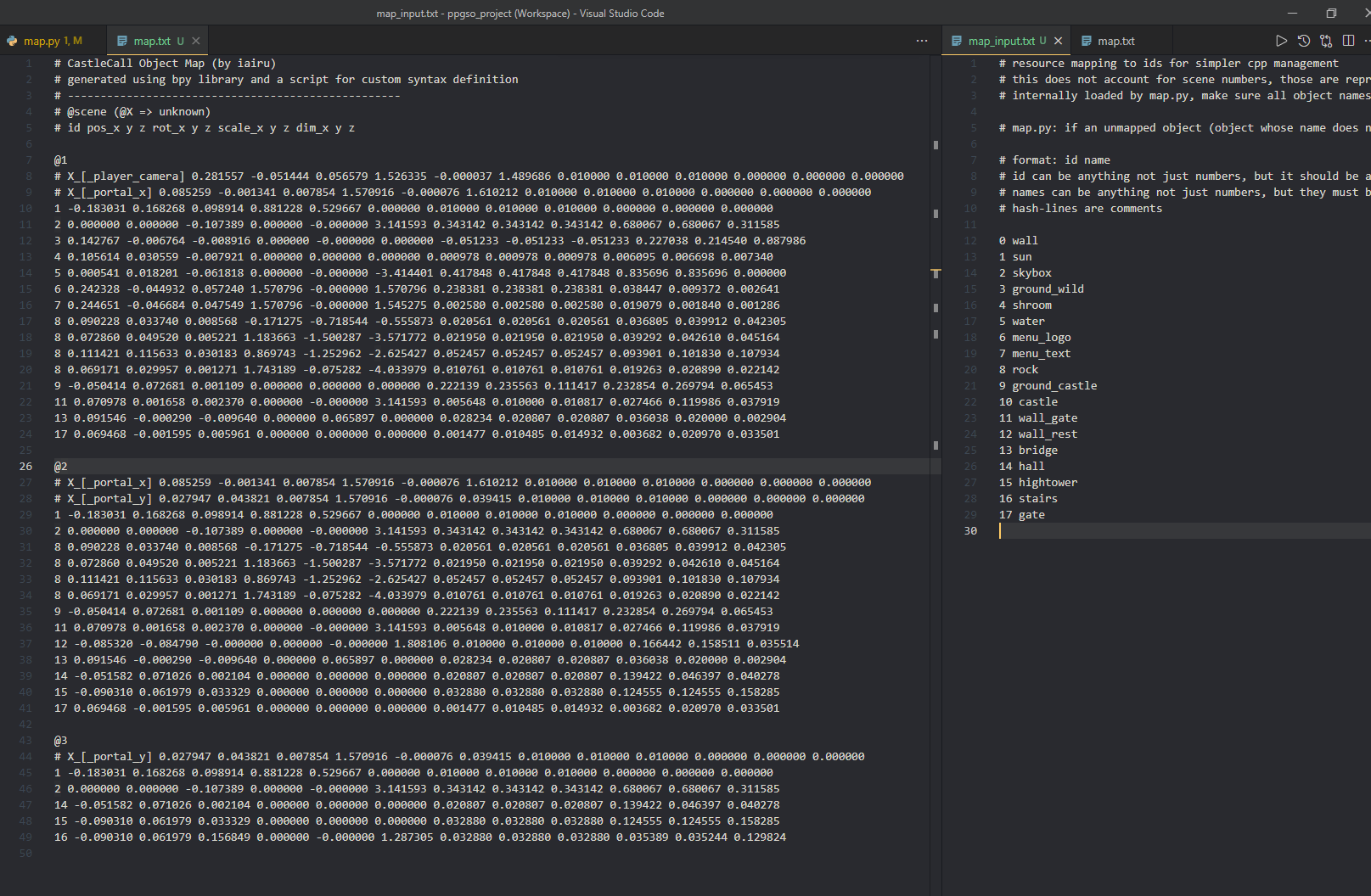
Predtým, ako priblížim rozloženie tried spomeniem **spôsob, ktorým je v projekte implementované načítavanie objektov a dátové štruktúry, ktoré museli byť aplikované za týmto cieľom:**

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.

## 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.** 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.

Vhodnejšie ohraničenie procedurálne generovanej scény !!bude!! doimplementované spôsobom ... todo

Obrázok

## Osvetlenie - Phongov model a diffuse materiály

(V pôvodnom riešení sme sa dostali len po kalkuláciu smeru svetla.) Momentálne riešenie je postavené na Phongovom modeli s využitím vlastného shadera obsahujúceho **tri vstupy pre jednotlivé pozície svetla a tri vstupy pre jednotlivé farby svetiel**. Okrem toho boli abstrahované konštanty tak, aby sa dali meniť priamo z scene.h, prípadne ľahšie debugovať zakomentovanými klávesovými skratkami.

Phongov model spočíva v sumárnom aplikovaní troch zložiek, ktoré decentne simulujú realistické svetlo v reálnom čase. Jedná sa o:

* **ambient**: konštantná zložka na pozadí, ktorá je jemne aplikovaná bez ohľadu na pozície svetiel – jej úlohou je dodať scéne všadeprítomný nádych k istej farbe
* **diffuse**: závisí na pozícií svetla, ale nezávisí na pozícií kamery – jedná sa o priame osvetlenie zo zdroja svetla, ktoré sa má jemne rozplynúť po povrchu objektov
* **specular**: závisí na pozícií svetla aj kamery – jedná sa o trik, pomocou ktorého sa dá ľahko simulovať lesk objektov

Okrem spomenutých zložiek je *phong\_frag.glsl* shader rozšírený o:

* podporu materiálových zložiek pre každú zložku svetla
* adjustáciu tlmenia svetla
* adjustáciu parametrov odlesku
* podporu textúr

Obrázok

## Shadow-maps tiene

Ideálny spôsob riešenia tieňov by spočíval v kontrole, či jednotlivé lúče svetla dosiahli bod na povrchu a neskončili skôr. Raycasting je ale drahá metóda a teda vznikol spôsob aproximácie pomocou shadow-maps.

Úlohou je na pozadí renderovať 3D scénu z pohľadu svetla ortografickým zobrazením do vlastného framebuffera a použiť jej depth-buffer pre získanie tieňov vo výslednom renderi scény. Shadow-maps tiene by mali byť po správnosti osvetlené len ambientným svetlom, teda bez diffuse a specular častí Phongovho modelu – aplikovanie tieňov je teda vykonané podsunutím shadowMap „textúry“ do phong fragmentového shadera, porovnaniu hĺbky priamo v shaderi a následne rozhodnutie či osvetliť daný bod alebo ho ponechať v tieni.

Po aplikovaní takto generovaných tieňov vznikajú tri najvýraznejšie druhy nepresností:

* Moiré efekt, ktorý je výsledkom nepresnosti porovnávania float typu a dá sa odstrániť orezaním čísel
* Polovica scény v tieni
* Pixelované okraje tieňov kvôli rozlíšeniu shadow-mapy, ktoré sa dajú najjednoduchšie upraviť pridaním dodatočného bluru, napr. cez blur screen shader.

## Selektívne kolízie boxami

Kolízie sú riešené len pre špecifické typy objektov, ktoré ich riešenie vyžadujú – v momentálne implementácií sa jedná o ForceObject (viac nižšie). Detekcia kolízie prebieha na základe špeciálnych CollisionBox objektov, resp. kontrola toho, či sa objekt pod vplyvom sily ako je gravitácia dostal za pozíciu kolízneho boxu, a návrat objektu do pôvodnej pozície (teda stálosť) v prípade, že sa tak stalo. Jedná sa o vcelku jednoduché riešenie, no jeho využitie je dynamickejšie a lacnejšie ako kontrolovať jednotlivé body (vertexy) reálneho (ráznejšie komplikovanejšieho) povrchu. Taktiež aplikovanie kolízneho boxu len na miestach, kde je potrebný a pre špecifické objekty, ktorým je relevantný znižuje výpočtovú záťaž.

Kolízie sú ďalej riešené dynamicky v tom, že k ich kontrole dochádza aj pri každej aktualizácií po kolízií, čo avšak zvyšuje výpočtovú záťaž. V dynamickejšej scéne by toto dovolilo silám pokračovať v prípade posunu kolízneho boxu v smere sily, opačný smer nebol riešený – riešenie by som si vedel predstaviť výpočtom pozície prieniku smerového vektora kolízneho boxu voči pomyselnému boxu ForceObject.

## Simulácia gravitácie

Gravitácia je simulovaná pomocou ForceObject v spojení s kolíziami v CollisionBox. Hodnoty pre gravitáciu sú konštantne dané okrem „mass“, ktorá sa dá prispôsobiť špecifickému objektu znížením – použitá je pre spomalenie „ľahších“ objektov alebo zrýchlenie „ťažších“.

Gravitácií je pridaný aj jemný nábeh po jej reálnu rýchlosť, ktorý dodáva dynamickejší pocit padajúcim objektom a teda krivka pádu nie je lineárna.

## Simulácia vetra

Vietor je simulovaný v rámci rovnakého abstraktného objektu ForceObject na princípe poryvu vetra (angl. wind gust) vďaka kmitaniu sínusovej funkcie. Pri finálnych objektoch je podobne ako gravitáciu možné vietor pozorovať na listových časticiach hierarchických stromov (LeafParticle).

Simuláciu vetra je možné do istej miery ovládať konštantami „windEvery“ a „windStrength“, ktorými sa dá kmitanie vetra zjemniť a spomaliť. Zároveň existuje konštantný vektor „windFluctuation“, ktorý definuje diaľku pohybu vetra „tam aj naspäť“ spôsobom; ak je 0.0 tak simulácia vetra na danej osi neprebieha. Pri simulácií vetra treba byť opatrný, aby dokázali častice dopadnúť na kolízny box a teda po čase zomrieť.

## Particle systém a vznik/zánik za behu – Padajúce lístie

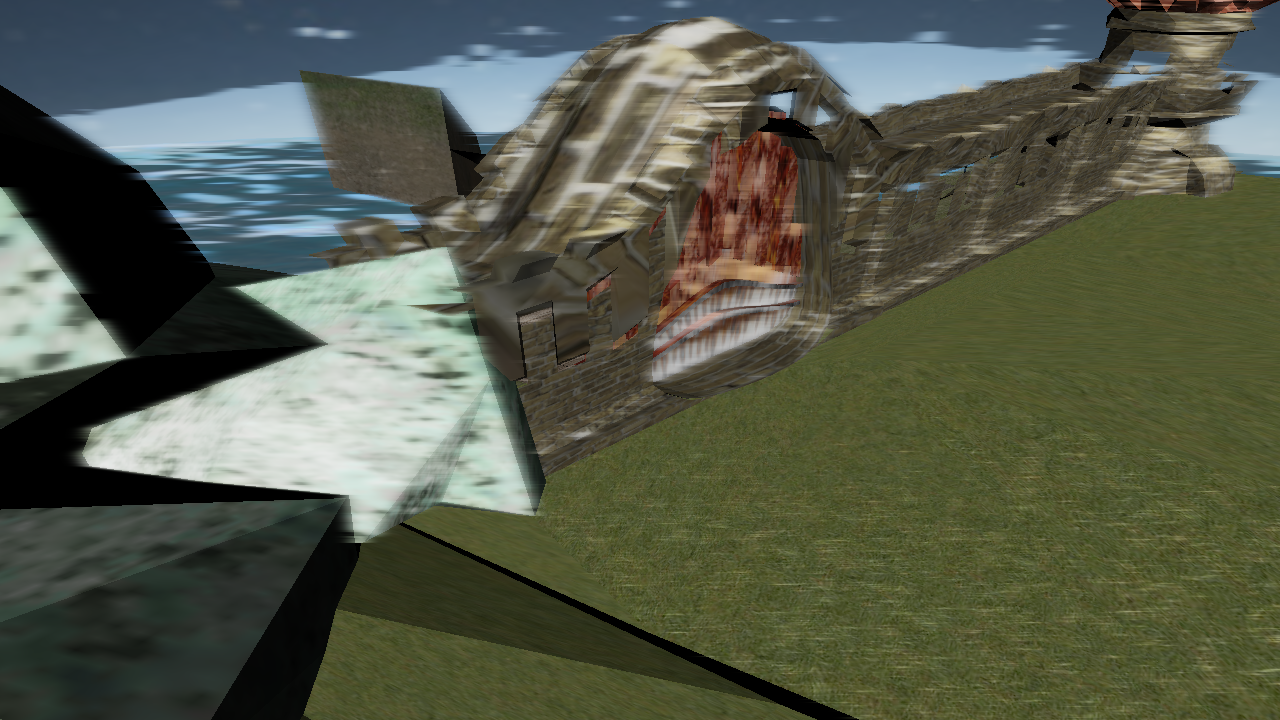
Pre demonštráciu kolízií, gravitácie a pridania vzniku/zániku objektov za behu prišlo vhodné do scény pridať padajúce listy k existujúcim hierarchicky implementovaným stromom. Implementácia lístia a systému je nim prispôsobená v leaf\_particles.h/cpp.

Počet častíc je daný konštantnou „n“. Pri zániku častice hneď vzniká nová. Jednotlivým časticiam je v rámci istých ohraničení pred vznikom generovaná semi-náhodná pozícia od rovnakej výšky (počiatku pádu). Po páde sa lístie dostane do „decay“ módu, kde je postupne odpočítaný náhodne pred-generovaný „decayTime“, až vďaka „update: return false“ sú rodičovskou triedou systému odstránené a hneď nahradené novými. Počet častíc jedného systému je teda vždy presne „n“.

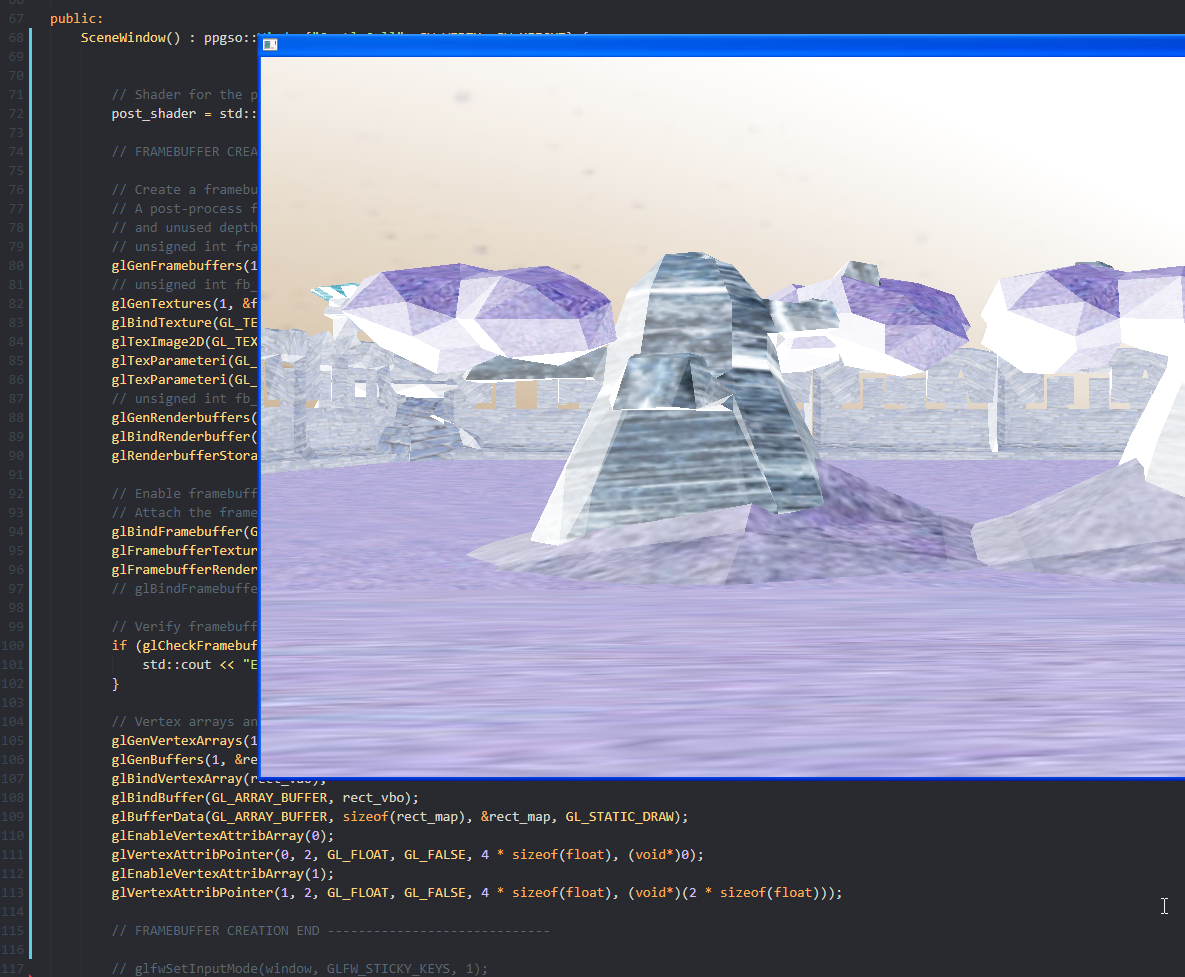
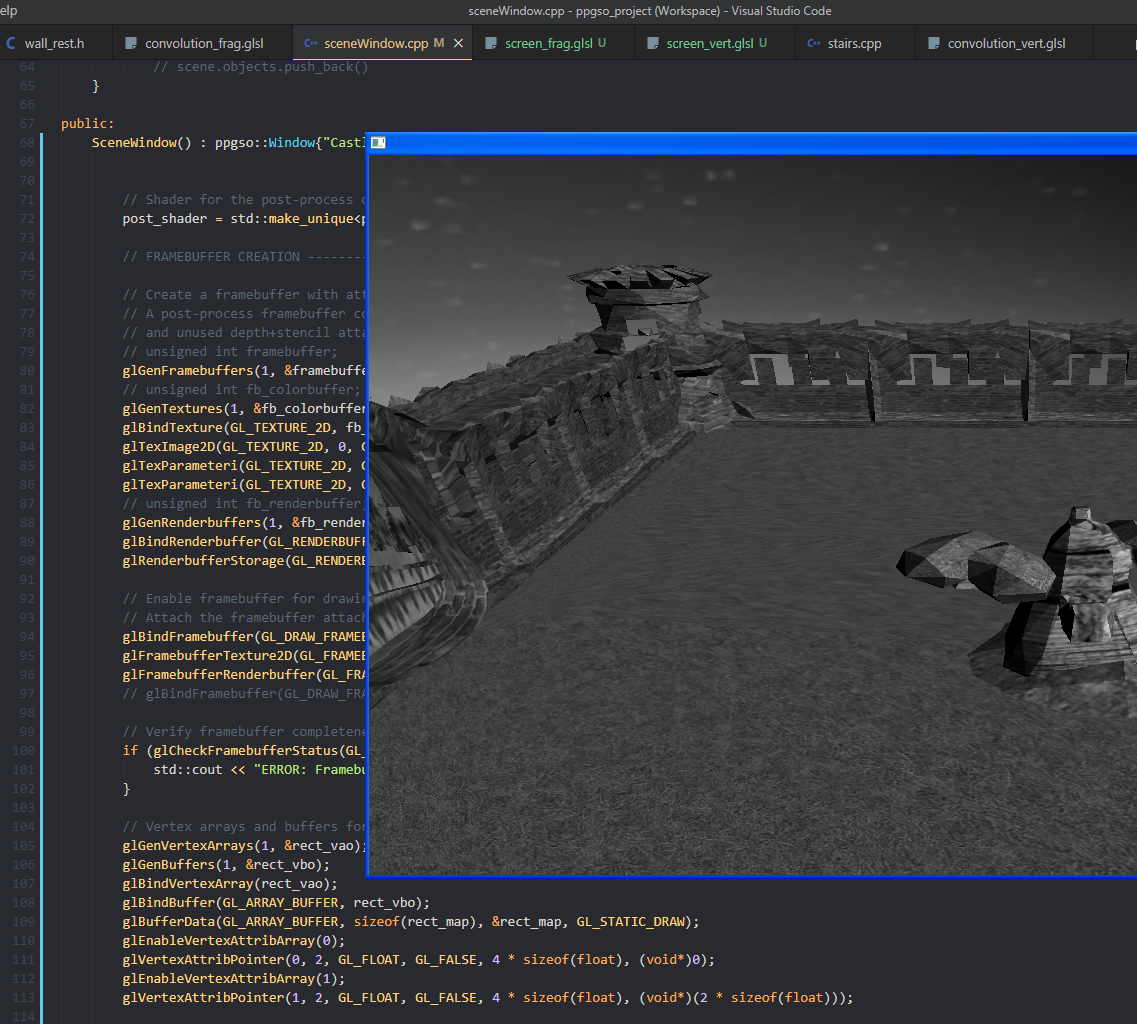
Obrázok

## Framebuffery - 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, **render** do colorbuffera
* 2D framebuffer pre **prvý efektový shader** „post\_shader\_bright“, ktorý vyberie kontrastné (**dostatočne bledé) časti obraz**u do separátneho colorbuffera
* 2D framebuffer, 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 predchádzajúci 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

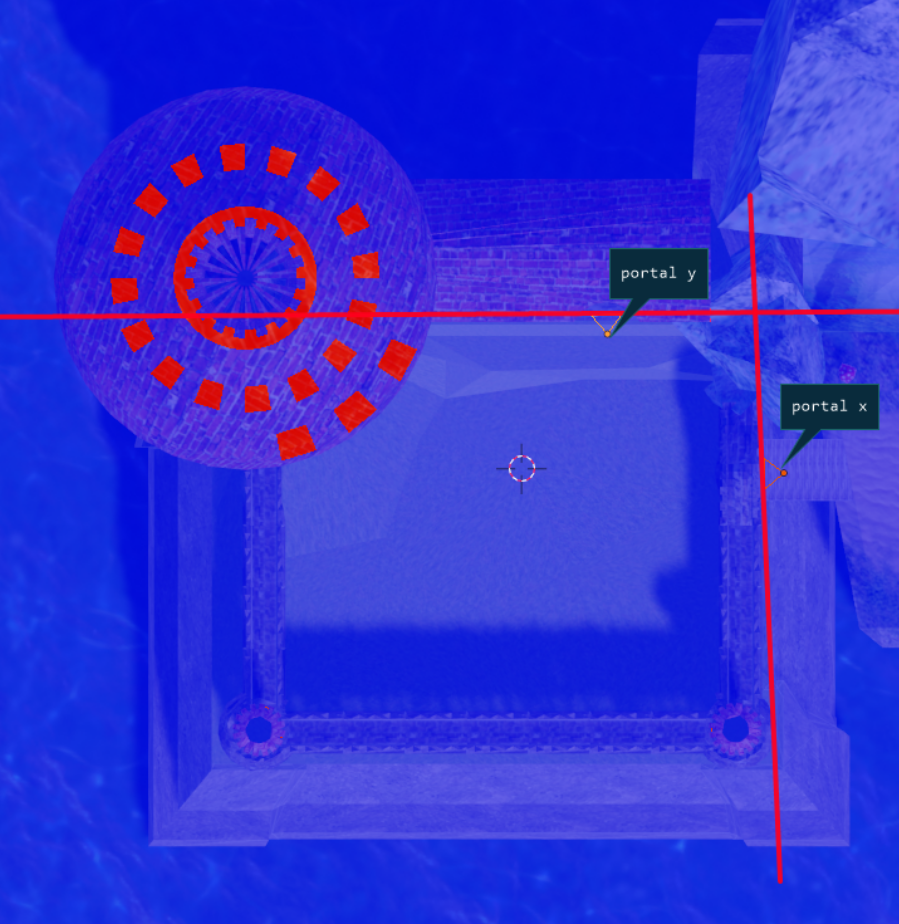
V tejto časti by som bližšie priblížil spôsob, ktorým sú scény menené, ako sú dáta o zmene scén uložené v Blenderi a následne z neho exportované pre použitie v projekte. Ďalej opäť spomeniem hierarchické transformácie.

## 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:

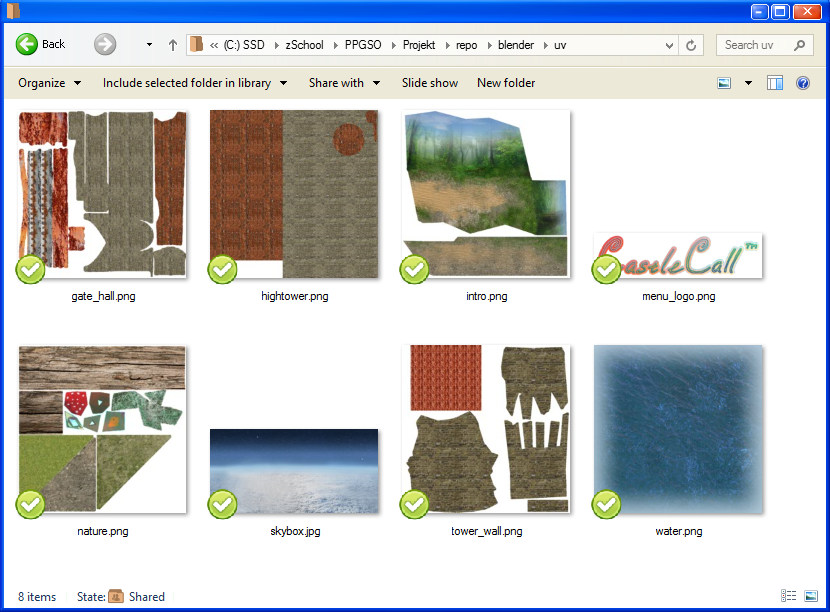
* vnesenia pocitu fluidity hráčovi,
* zaručeniu, že má stále pod sebou podlahu (aby nezačal padať do nekonečna)
* načítavanie objektov novej scény môže byť uskutočnené dynamicky na základe smerového vektora hráča a jeho rýchlosti na pozadí namiesto zobrazenia obrazovky načítavania, ktorá by oddialila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



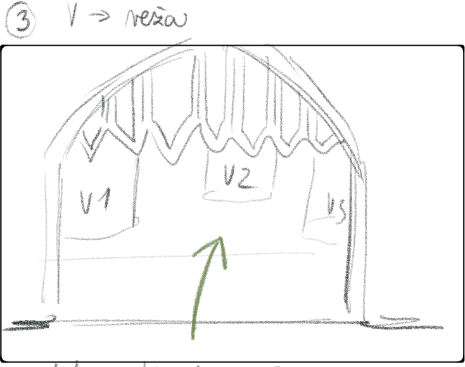
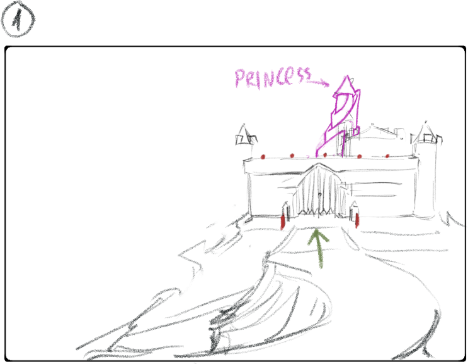
## *Poznámka o UV*

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.



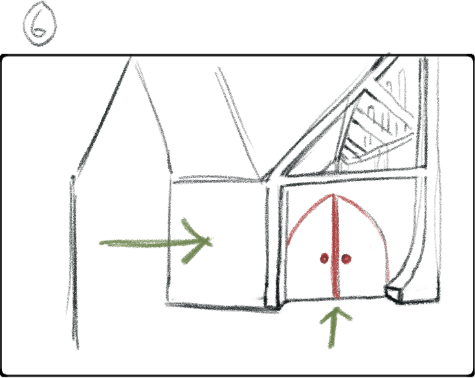
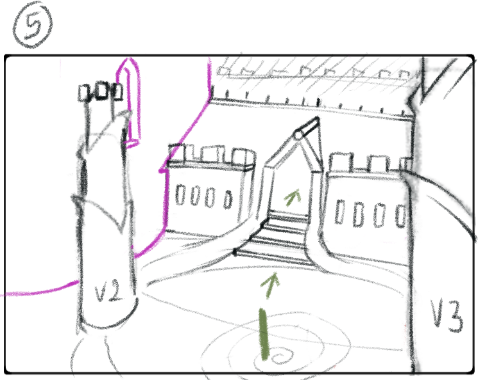
## *Prechod z prvej scény do druhej: Brána*

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.

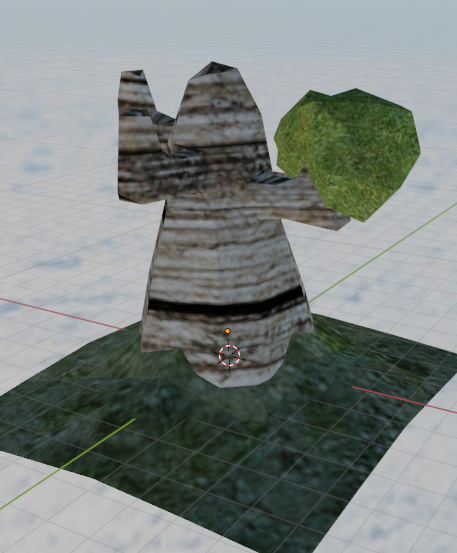
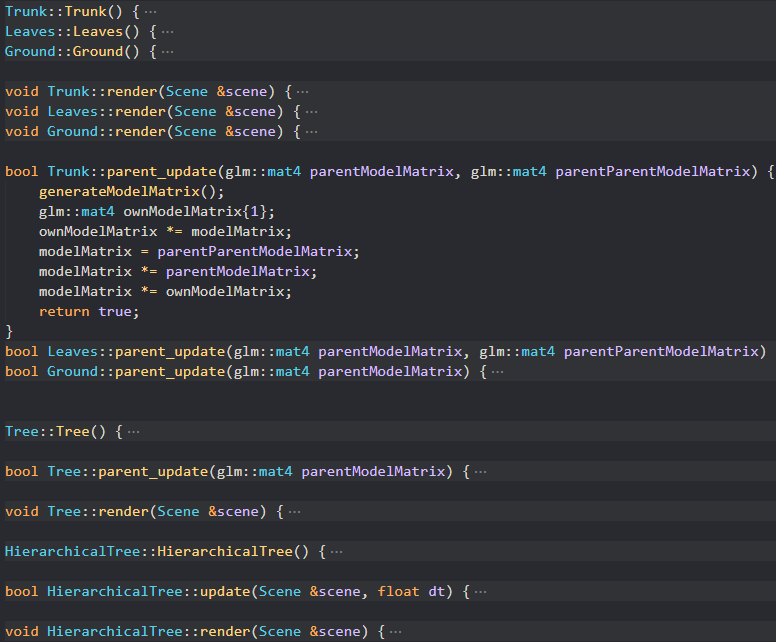


## Hierarchické transformácie

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

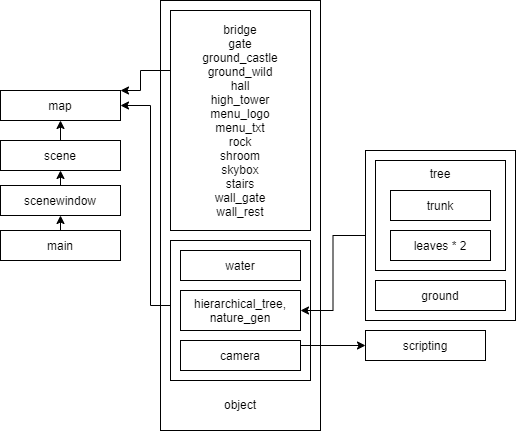
* hierarchický strom (HierarchicalTree) sa skladá zo stromu ako takého (private Tree) a kopčekovej podlahy pripomínajúcej korene pod zemou (private Ground)
* strom ako taký (private Tree) sa ďalej skladá z koruny (private Trunk) a viacerých inštancí listov (private Leaves)
* pozn. Ground bol pre lepšiu ukážku simulácie gravitácie s kolíziami nahradení časticovým systémom LeafParticleSystem

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.



aktualizovať

# Doplnky

## Mapa návrhu vs implementácie

|  |  |
| --- | --- |
| C:\Users\Ondrej\AppData\Local\Temp\msohtmlclip1\02\clip_image012.png |  |

## Rozdelenie práce

|  |  |
| --- | --- |
| **Finálne odovzdanie** |  |
| Post-process grayscale, invert, blur, bloom efekty | Ondrej |
| Hierarchické transformácie | Ondrej |
| Objekty, rozloženie objektov, UV mapping | Ondrej |
| Skript pre konverziu Blender scén map.py | Ondrej |
| Dokumentácia 80% | Ondrej |
| Demo video, odovzdanie | Ondrej |
|  |  |
| Procedurálne generovanie (stromov na nádvorí) | Marek |
| Systém animácie kamery | Marek |
| Skript pre načítanie Blender scén map.cpp | Marek |
| Dokumentácia 20% | Marek |
| Svetlo | Marek (len výpočet smeru, žiadne shadere) |
| Shadow-maps | Marek (nestihol) |
| Zmena scén | Marek (nestihol) |

|  |  |
| --- | --- |
| **Odovzdanie po termíne** |  |
| Kolízie pomocou kolíznych boxov | Ondrej |
| Gravitácia a vietor | Ondrej |
| Phongov osvetlovací model + materiály | Ondrej |
| Zmena odtieňov svetla, viacero zdrojov svetla | Ondrej |
| Zánik objektov v čase (časticový systém) | Ondrej |
| Doplnenie dokumentácie 100% | Ondrej |
| Keyframes animácia ostatných objektov | Ondrej |
| Zmena scén | ?? |
| Animácia odtieňov svetla | ?? |
|  |  |
| Shadow-maps | Marek (??) |
| Obmedzenie procedurálne generovanej scény | Marek (??) |