

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

Dodatočné odovzdanie

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
	Objekty a konverzia scény	4
2.	Algoritmy	6
	Procedurálne generovanie scény	6
	Osvetlenie - Phongov model a diffuse materiály	6
	Shadow-maps tieň	8
	Selektívne kolízie boxami	8
	Simulácia gravitácie (aplikované na lístie)	8
	Simulácia vetra (aplikované na lístie)	9
	Particle systém a vznik/zánik za behu – Padajúce lístie	9
	Framebuffery - Post-process Bloom efekt	9
	Animácia objektov	11
3.	Scény, priestorové vzťahy	12
	Zmena scén	12
	<i>Poznámka o UV</i>	13
	<i>Prechod z prvej scény do druhej: Brána</i>	13
	<i>Prechod z druhej scény do tretej: Hala</i>	14
	Hierarchické transformácie	14
4.	Diagram tried objektov scény	15
5.	Doplňky	15
	Mapa návrhu vs implementácie	15
	Rozdelenie práce	16



1. 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`.

```
typedef enum _ITMTYPE {  
    LIGHT, // 0  
    SKYBOX, // 1  
    GROUND_WILD, // 2  
    SHROOM, // 3  
    WATER, // 4  
    MENU_LOGO, // 5  
    MENU_TXT, // 6  
    ROCK, // 7  
    GROUND_CASTLE, // 8  
    GATE, // 9  
    WALL_GATE, // 10  
    WALL_REST, // 11  
    BRIDGE, // 12  
    HALL, // 13  
    HIGH_TOWER, // 14  
    STAIRS, // 15  
    CAMERA, // 16  
    NATURE_GEN // 17  
} ITMTYPE;
```

```
typedef struct _MAPITEM {  
    ITMTYPE object;  
    glm::vec3 pos;  
    glm::vec3 rotation;  
    glm::vec3 scale;  
    glm::vec3 dim;  
} MAPITEM, *PMAPITEM;  
  
typedef struct _SCENE_DESC {  
    unsigned int scene_id;  
    std::vector<MAPITEM> map;  
} SCENE_DESC, *PSCENE_DESC;  
  
class Map {  
private:  
    std::vector<SCENE_DESC> scenes;  
  
    std::unique_ptr<Object> getObj(ITMTYPE tgt_obj);  
  
    void placeObj(MAPITEM item, Scene *scene);  
  
public:  
    Map();  
  
    void placeItems(unsigned int scene_id, Scene *scene);  
};
```

```
typedef struct _MOVE {  
    double start; // time to start since last action finish  
    double duration; // remaining time  
    glm::vec3 target_pos;  
    glm::vec3 target_rot;  
} MOVE, *PMOVE;
```

Objekty a konverzia scény

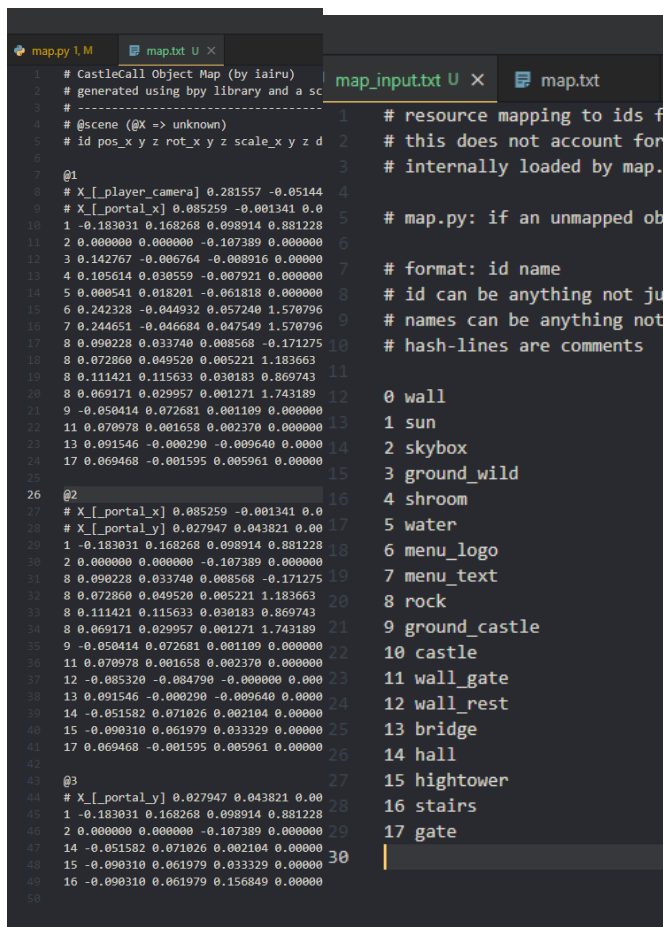
Predtým, ako priblížim rozloženie tried spomeniem spôsob, ktorým je v projekte implementované načítavanie objektov a dátové štruktúry, ktoré museli byť aplikované za týmto cieľom:

Načítavanie objektov prebieha v map.cpp za použitia vlastného enumerátora a switcha. Prvý priraduje všetkým druhom statických objektov ľudske 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 priradovania scény 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 aplikácii – 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.



```
map.py 1, M map.txt U X
1 # CastleCall Object Map (by iairu)
2 # generated using bpy library and a sc
3 #
4 # @scene (@X => unknown)
5 # id pos_x y z rot_x y z scale_x y z d
6
7 @1
8 # X[_player_camera] 0.281557 -0.05144
9 # X[_portal_x] 0.085259 -0.001341 0.0
10 1 -0.183031 0.168268 0.098914 0.881228
11 2 0.000000 0.000000 -0.107389 0.000000
12 3 0.142767 -0.006764 -0.000916 0.000000
13 4 0.105614 0.030559 -0.007921 0.000000
14 5 0.000541 0.018201 -0.061818 0.000000
15 6 0.242328 -0.044932 0.057240 1.570796
16 7 0.244651 -0.046684 0.047549 1.570796
17 8 0.090228 0.033740 0.008568 -0.171275
18 9 0.072860 0.049520 0.005221 1.183663
19 10 0.111421 0.115633 0.030183 0.869743
20 11 0.069171 0.029957 0.001271 1.743189
21 12 -0.050414 0.072681 0.001109 0.000000
22 13 0.070978 0.001658 0.002370 0.000000
23 14 0.091546 -0.000290 -0.009640 0.000000
24 17 0.069468 -0.001595 0.005961 0.000000
25
26 @2
27 # X[_portal_x] 0.085259 -0.001341 0.0
28 # X[_portal_y] 0.027947 0.043821 0.00
29 1 -0.183031 0.168268 0.098914 0.881228
30 2 0.000000 0.000000 -0.107389 0.000000
31 8 0.090228 0.033740 0.008568 -0.171275
32 9 0.072860 0.049520 0.005221 1.183663
33 10 0.111421 0.115633 0.030183 0.869743
34 11 0.069171 0.029957 0.001271 1.743189
35 12 -0.050414 0.072681 0.001109 0.000000
36 13 0.070978 0.001658 0.002370 0.000000
37 14 -0.085320 -0.004790 -0.000000 0.000000
38 15 0.091546 -0.000290 -0.009640 0.000000
39 16 -0.051582 0.071026 0.002104 0.000000
40 17 -0.090310 0.061979 0.033329 0.000000
41 18 -0.090310 0.061979 0.156849 0.000000
42
43 @3
44 # X[_portal_y] 0.027947 0.043821 0.00
45 1 -0.183031 0.168268 0.098914 0.881228
46 2 0.000000 0.000000 -0.107389 0.000000
47 14 -0.051582 0.071026 0.002104 0.000000
48 15 -0.090310 0.061979 0.033329 0.000000
49 16 -0.090310 0.061979 0.156849 0.000000
50
map_input.txt U X map.txt
1 # resource mapping to ids f
2 # this does not account for
3 # internally loaded by map.
4
5 # map.py: if an unmapped ob
6
7 # format: id name
8 # id can be anything not ju
9 # names can be anything not
10 # hash-lines are comments
11
12 0 wall
13 1 sun
14 2 skybox
15 3 ground_wild
16 4 shroom
17 5 water
18 6 menu_logo
19 7 menu_text
20 8 rock
21 9 ground_castle
22 10 castle
23 11 wall_gate
24 12 wall_rest
25 13 bridge
26 14 hall
27 15 hightower
28 16 stairs
29 17 gate
30
```


2.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 nádvorie 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.

Jednotlivé oblasti sú zadefinované v mape. Viac oblastí vedľa seba pôsobí ako jedna veľká oblasť.



Osvetlenie - Phongov model a diffuse materiály

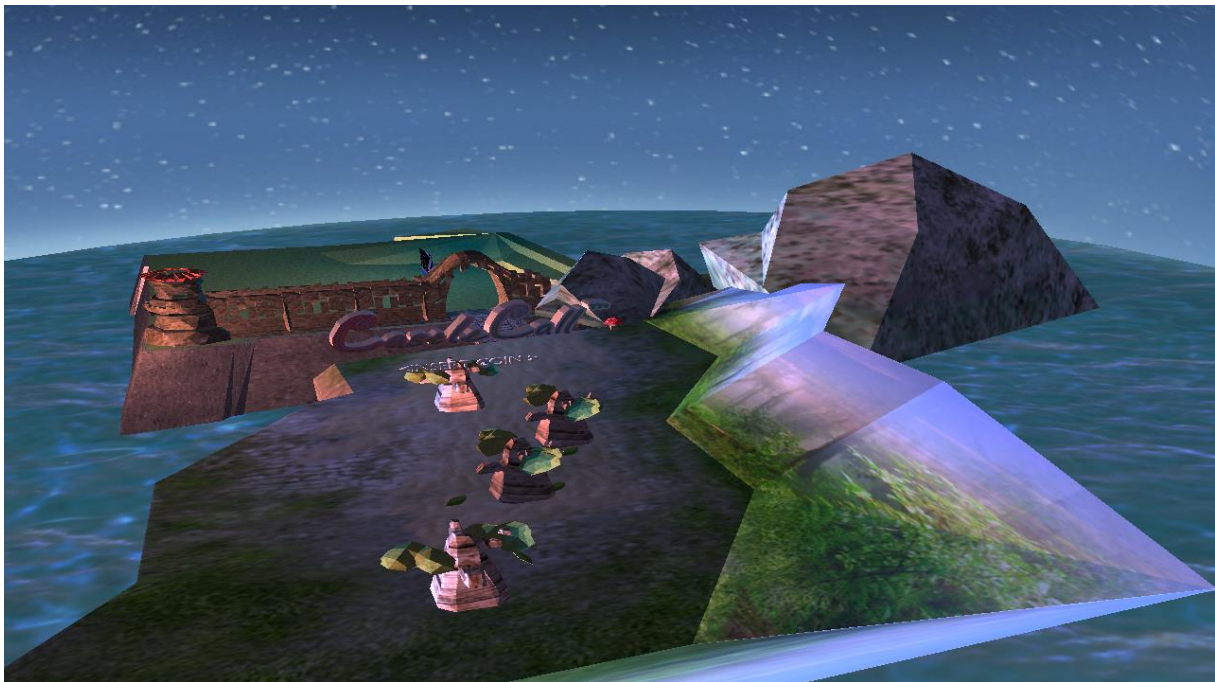
(V pôvodnom riešení sme sa dostali len po kalkuláciu smeru svetla.) Momentálne riešenie je postavené na Phongovom modeli s využitím vlastného shadera obsahujúceho **tri vstupy pre jednotlivé pozície svetla a tri vstupy pre jednotlivé farby svetiel**. Okrem toho boli abstrahované konštanty tak, aby sa 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



Shadow-maps tieňe

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 tieňe 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 aktualizácii 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 (aplikované na lístie)

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 (aplikované na lístie)

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 dĺžku pohybu vetra „tam aj naspäť“ spôsobom; ak je 0.0 tak simulácia vetra na danej osi neprebieha. Pri simulácii 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

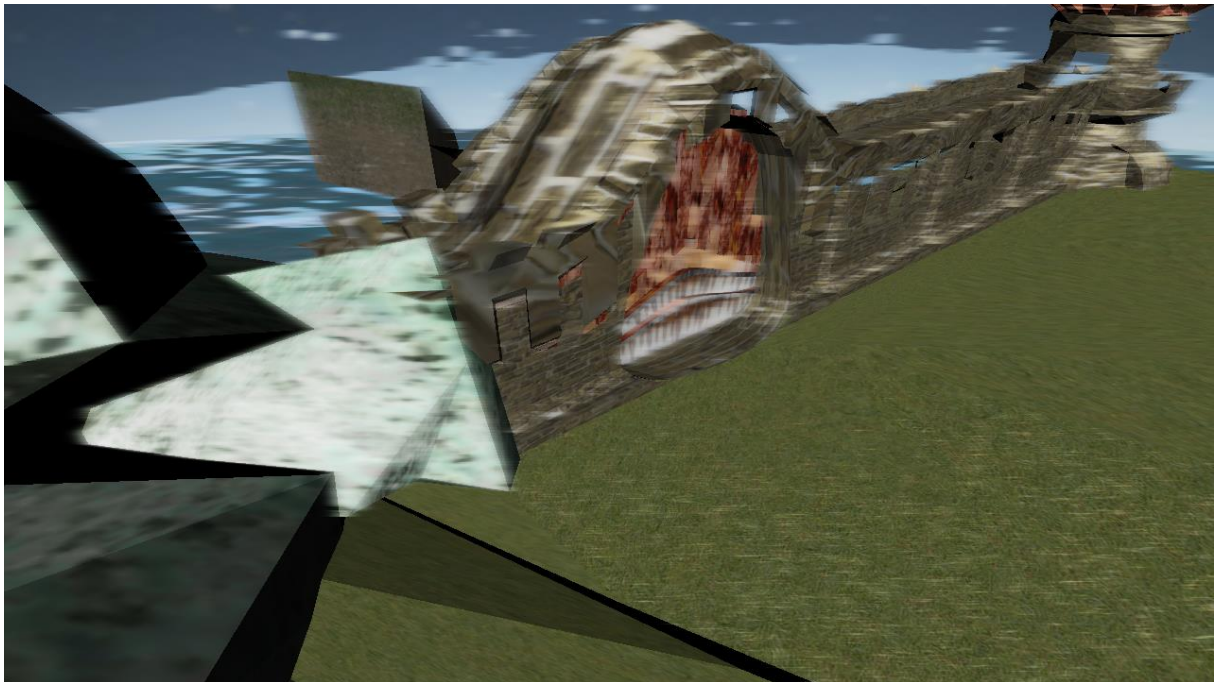
Pre demonštráciu kolízií, gravitácie a pridania vzniku/zániku objektov za behu prišlo vhodné do scény pridať padajúce lístie k existujúcim hierarchicky implementovaným stromom. Implementácia lístia a systému je nim prispôbena v leaf_particles.h/cpp.

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“.



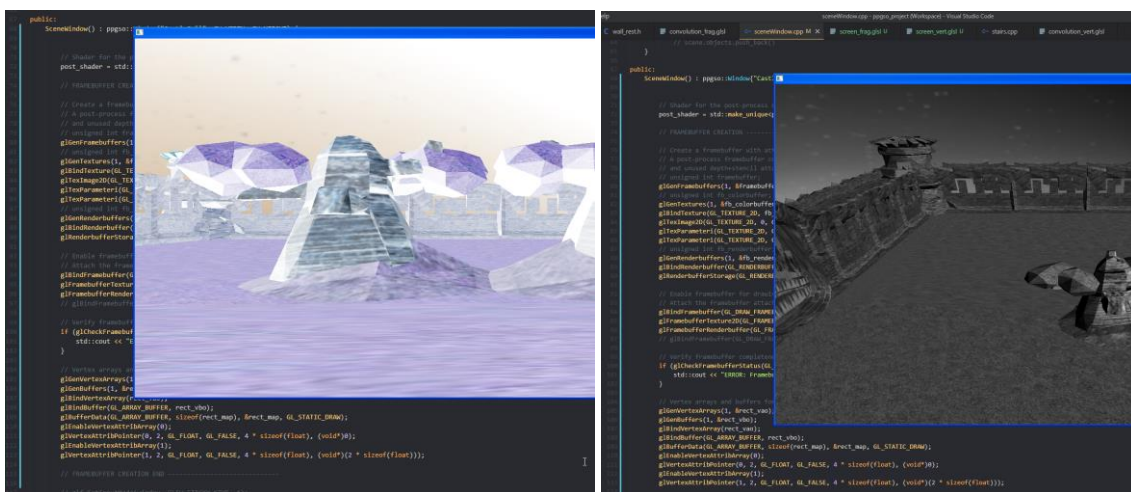
Framebuffery - Post-process Bloom efekt

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.



aktualizovať obrázok (treba zapnúť bloom shader v scenewindow.cpp na konci a vypnúť pass shader)

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 implementácii 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 obrazu 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

Animácia objektov

Animácia je riadená kľúčovými snímkami a interpoláciou medzi nimi. Existuje možnosť zapnúť loop, prípadne odstrániť objekt po dokončení animácie. Táto animácia je riešená separátne od síl ako sú vietor/gravitácia a je postavená na interpolácií medzi predgenerovanými modelMatrixami namiesto vytvárania nových. Dostupná je pre všetky objekty typu Object, stačí teda upraviť update funkciu aby používala „**generateFrameModelMatrix**“, ktorý **vykoná interpoláciu medzi modelMatrixami** na kľúčových snímkoch. Kľúčové snímky je vhodné tvoriť v konštruktore objektu pomocou addKeyframe.

Ukážkové objekty a manuálna synchronizácia medzi nimi tvoriaca **jednoduchý príbeh**:

- shroom.cpp príde k bráne a počká až sa plne otvorí
- gate.cpp sa plne otvorí a shroom.cpp cez ňu prejde, počká kým sa zatvorí a ide do kúta k veži
- shroom.cpp sa vráti k bráne, počká na otvorenie, gate.cpp sa otvorí, shroom.cpp prejde do predvolenej pozície a príbeh sa opakuje vďaka „loop“

3.Scény, priestorové vzťahy

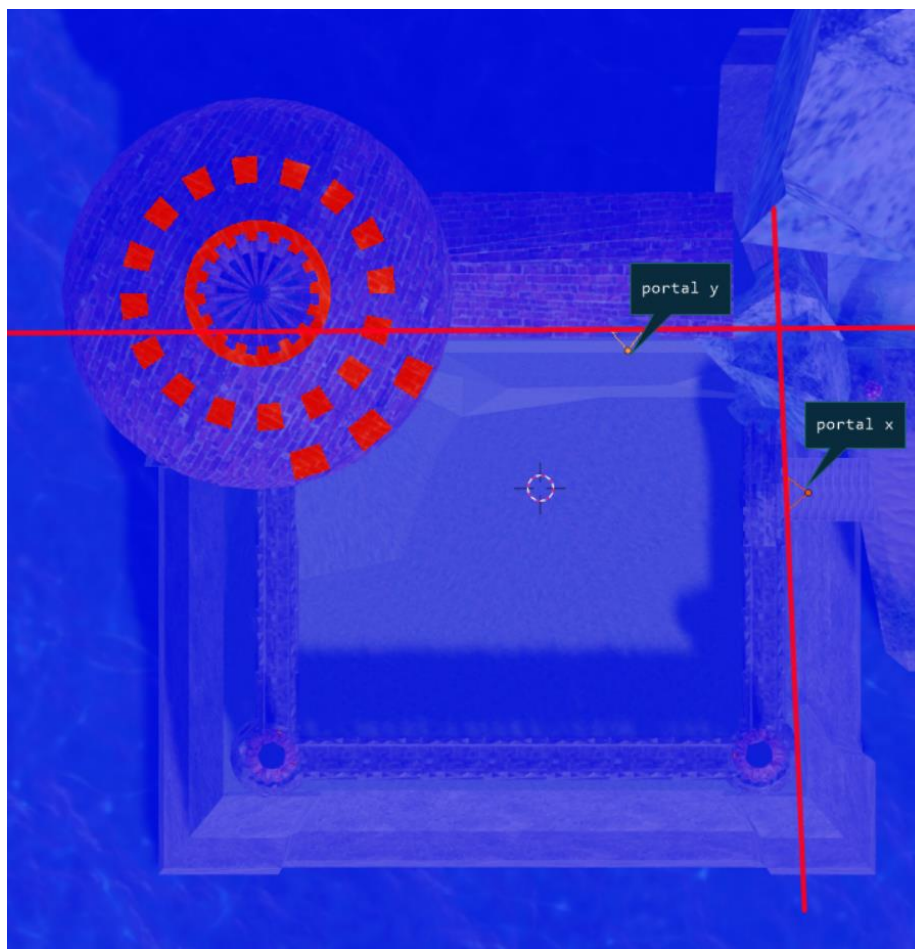
V tejto časti by som bližšie priblížil spôsob, ktorým sú scény menené, ako sú dáta o zmene scén uložené v Blenderi a následne z neho exportované pre použitie v projekte. Ďalej opäť spomeniem hierarchické transformácie.

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 prvej scény. Tieto prechodné objekty sú takto definované za účelom:

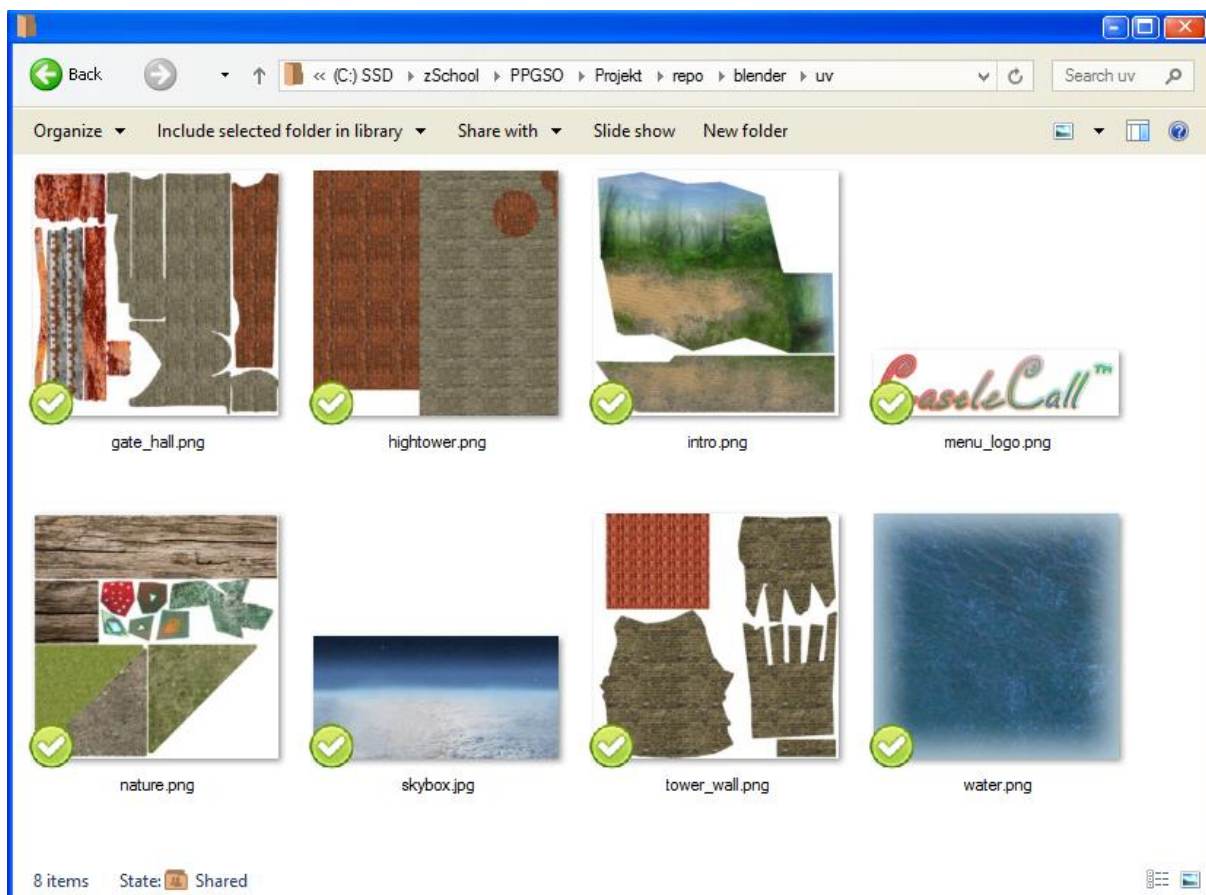
- vnesenia pocitu fluidity hráčovi,
- zaručeniu, že má stále pod sebou podlahu (aby nezačal padať do nekonečna)
- načítavanie objektov novej scény môže byť uskutočnené dynamicky na základe smerového vektora hráča a jeho rýchlosti na pozadí namiesto zobrazenia obrazovky načítavania, ktorá by oddialila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

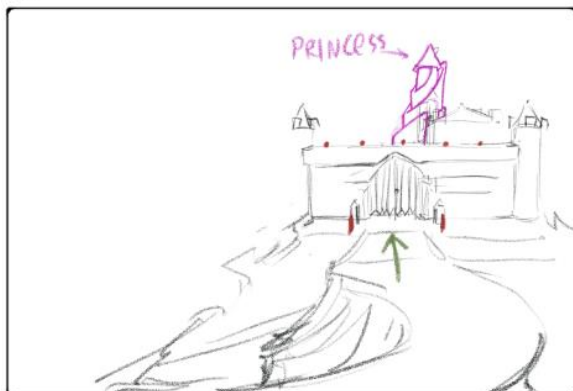
V pôvodnom pláne bola tiež zámena polygonálne zjednodušených objektov za kompletné pre lepšiu optimalizáciu aplikácie, to by avšak predĺžilo prácu na nebodovateľných častiach projektu, ktoré sú už aj napriek tomu obširné, keďže všetky scény sú zložené z plne vlastnoručne tvorených objektov a UV máp.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



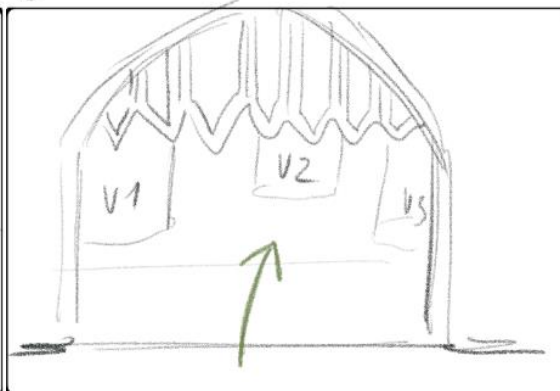
Prechod z prvej scény do druhej: Brána

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

①

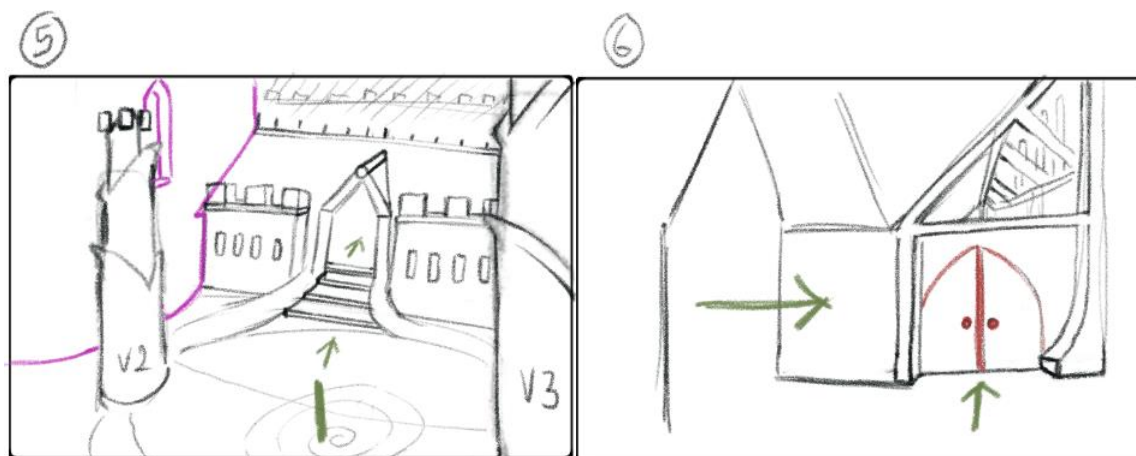


③ V → rezav



Prechod z druhej scény do tretej: Hala

Tento prechod načíta schody nachádzajúce sa vo vysokej veži.

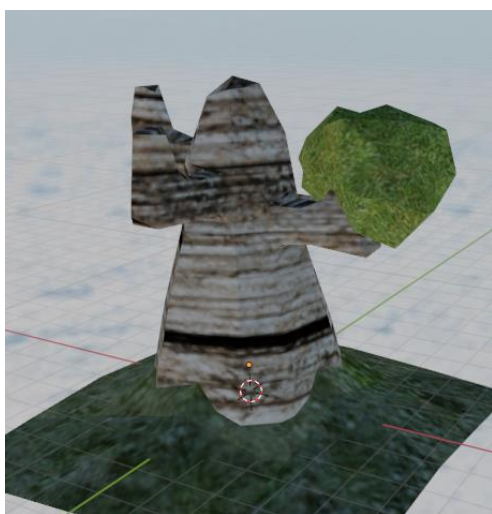


Hierarchické transformácie

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 inštancií listov (private Leaves)
- pozn. Ground bol pre lepšiu ukážku simulácie gravitácie s kolíziami nahradený 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.



```
bool Trunk::parent_update(glm::mat4 parentModelMatrix) {
    generateModelMatrix();
    glm::mat4 ownModelMatrix{1};
    ownModelMatrix *= parentModelMatrix;
    modelMatrix = parentModelMatrix;
    modelMatrix *= ownModelMatrix;
    return true;
}

bool Leaves::parent_update(glm::mat4 parentModelMatrix) {
    generateModelMatrix();
    glm::mat4 ownModelMatrix{1};
    ownModelMatrix *= parentModelMatrix;
    modelMatrix = parentModelMatrix;
    modelMatrix *= ownModelMatrix;
    return true;
}

bool Ground::parent_update(glm::mat4 parentModelMatrix) {
    generateModelMatrix();
    glm::mat4 ownModelMatrix{1};
    ownModelMatrix *= parentModelMatrix;
    modelMatrix = parentModelMatrix;
    modelMatrix *= ownModelMatrix;
    return true;
}

Tree::Tree() { ... }

bool Tree::parent_update(glm::mat4 parentModelMatrix) {
    generateModelMatrix();
    glm::mat4 ownModelMatrix{1};
    ownModelMatrix *= parentModelMatrix;
    modelMatrix = parentModelMatrix;
    modelMatrix *= ownModelMatrix;
    return true;
}

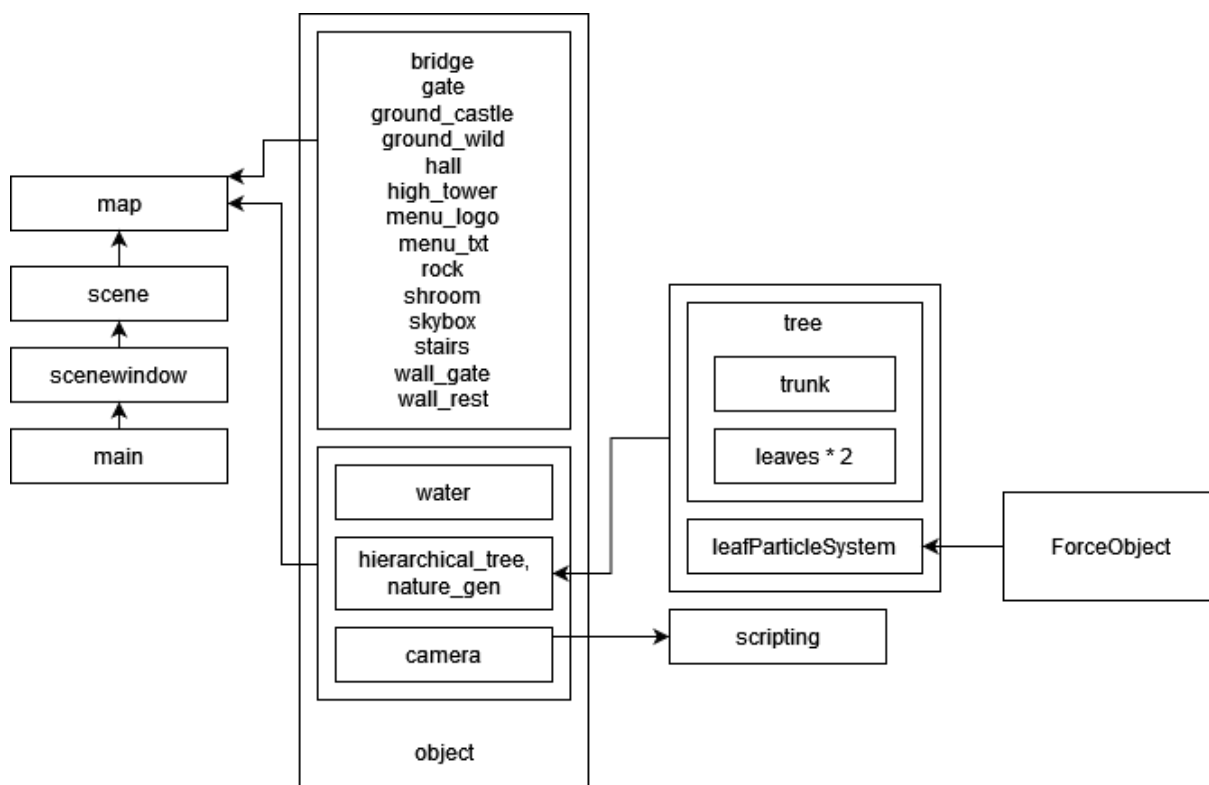
void Tree::render(Scene &scene) { ... }

HierarchicalTree::HierarchicalTree() { ... }

bool HierarchicalTree::update(Scene &scene, float time) { ... }
```

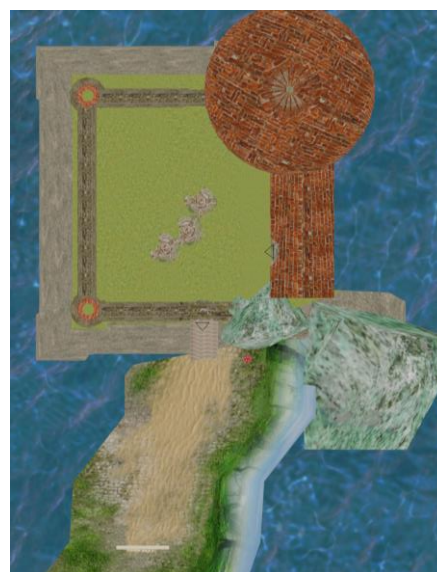
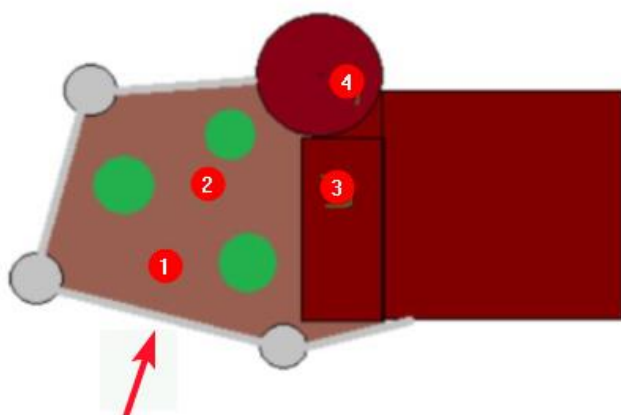
4. Diagram tried objektov scény

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. demonštrovať potrebnú implementovanú funkcionality jednotlivých bodov hodnotenia.



5. Doplnky

Mapa návrhu vs implementácie



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 map.txt do projektu map.cpp	Marek
Dokumentácia 20%	Marek
Svetlo	Marek (len výpočet smeru, žiadne shadere)
Shadow-maps	Marek (odložené z dôvodu nedostatku času)
Zmena scén	Marek (neúplná podpora)

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
modelMatrix Keyframes animácia ostatných objektov	Ondrej
Jednoduchý príbeh gate.cpp a shroom.cpp	Ondrej
Animácia odtieňov svetla	Ondrej
Doplnenie a aktualizovanie dokumentácie 80%	Ondrej
<i>(WIP) Pokus stihnúť Shadow-maps lebo Marekovi sa nedarilo</i>	<i>Ondrej</i>

Zmena scén	Marek
Obmedzenie procedurálne generovanej scény	Marek
Doplnenie obrázkov do dokumentácie	Marek
<i>(WIP) Shadow-maps 2 dni snahy, nefungujú</i>	<i>Marek</i>