

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

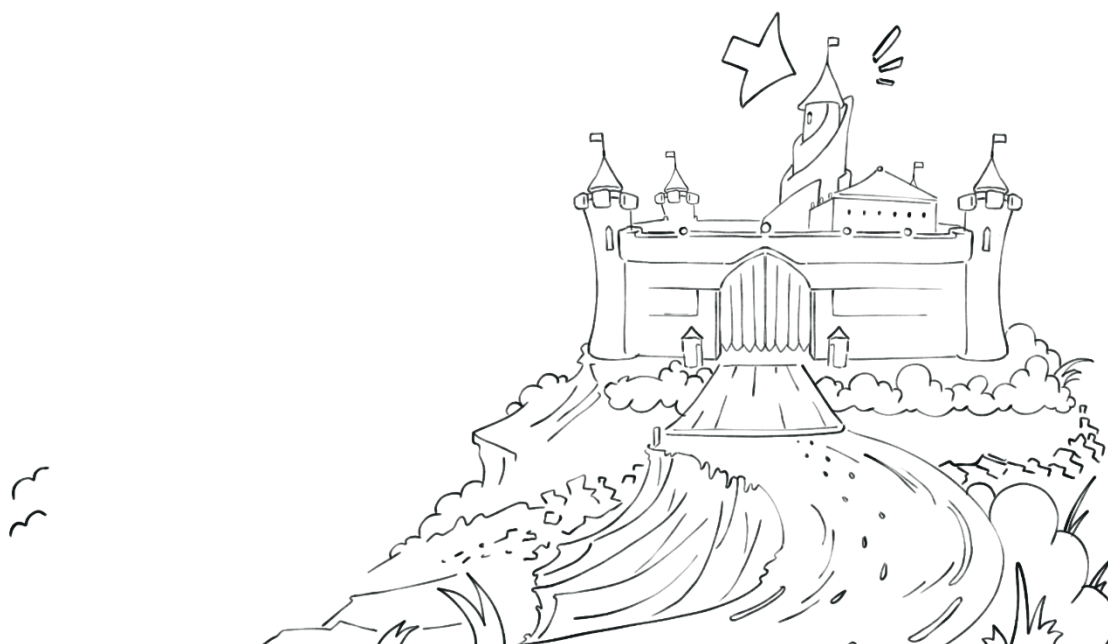
Meno cvičiaceho: Ing. Lukáš Hudec, PhD.

Časy cvičení: Utorok 16:00-17:50

Akademický rok: 2021/22 ZS

Obsah

1.	Dátové štruktúry	3
	Objekty a konverzia scény	4
2.	Algoritmy	6
	Procedurálne generovanie scény	6
	Vhodný spôsob osvetlenia	6
	Post-process bloom efekt	7
3.	Scény, priestorové vzťahy	9
	Zmena scén	9
	<i>Poznámka o UV</i>	10
	<i>Prechod z prvej scény do druhej: Brána</i>	10
	<i>Prechod z druhej scény do tretej: Hala</i>	11
	Hierarchické transformácie	11
4.	Diagram tried objektov scény	12
5.	Doplňky	13
	Mapa návrhu vs implementácie	13



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

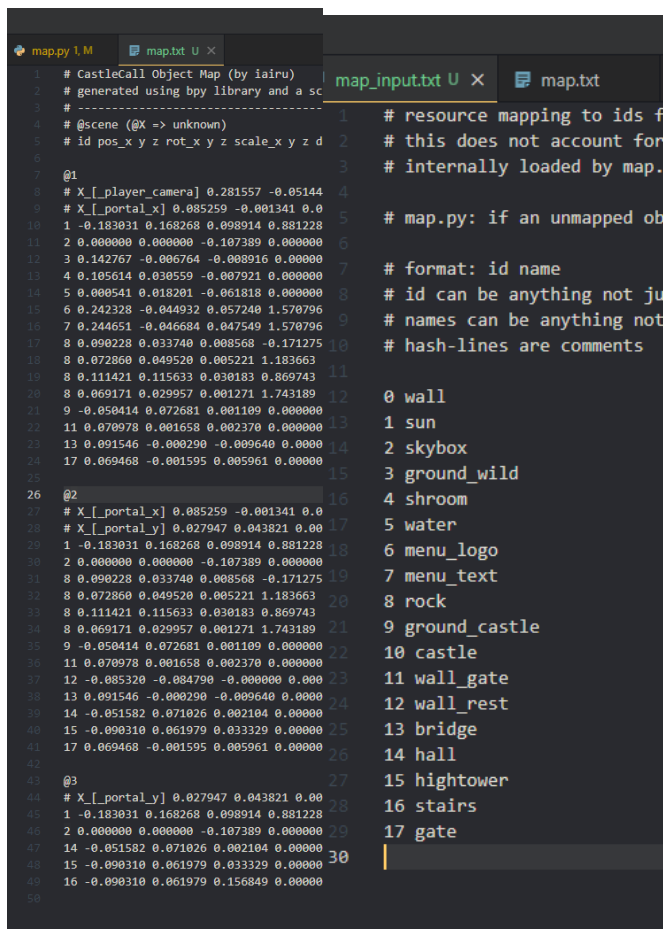
Predtým, ako priblížim rozloženie tried spomeniem **spôsob, ktorým je v projekte implementované načítavanie objektov a dátové štruktúry, ktoré museli byť aplikované za týmto cieľom:**

Načítavanie objektov prebieha v map.cpp za použitia vlastného enumerátora a switcha. Prvý priradzuje všetkým druhom statických objektov ľudsky čitateľné pomenovanie a je využitý najmä z hľadiska zjednodušenia programovacích nárokov. Druhý na základe prvého vytvorí unikátne inštanície 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 8 0.072860 0.049520 0.005221 1.183663
19 8 0.111421 0.115633 0.030183 0.869743
20 8 0.069171 0.029957 0.001271 1.743189
21 9 -0.050414 0.072681 0.001109 0.000000
22 11 0.070978 0.001658 0.002370 0.000000
23 13 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 8 0.072860 0.049520 0.005221 1.183663
33 8 0.111421 0.115633 0.030183 0.869743
34 8 0.069171 0.029957 0.001271 1.743189
35 9 -0.050414 0.072681 0.001109 0.000000
36 11 0.070978 0.001658 0.002370 0.000000
37 12 -0.085320 -0.084790 -0.000000 0.000000
38 13 0.091546 -0.000290 -0.009640 0.000000
39 14 -0.051582 0.071026 0.002104 0.000000
40 15 -0.090310 0.061979 0.033329 0.000000
41 17 0.069468 -0.001595 0.005961 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.



Vhodný spôsob osvetlenia

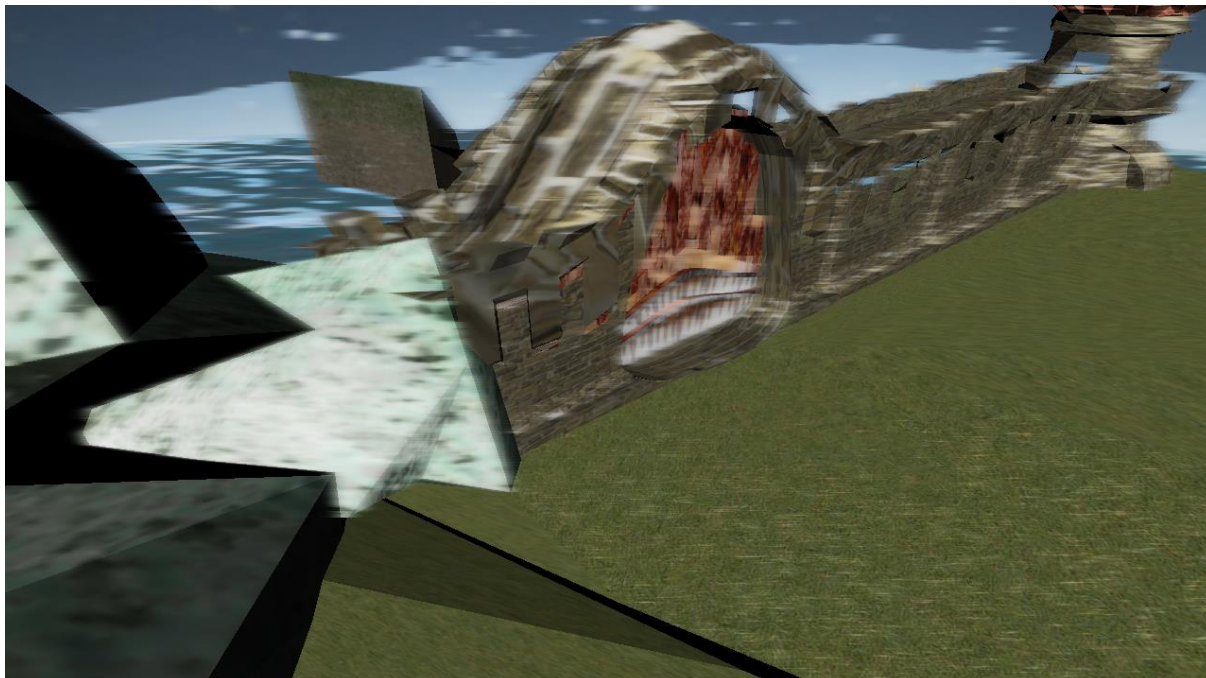
Diffuse lighting

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 neexistuje, funkcia pre výpočet smeru svetla vráti predvolenú hodnotu.

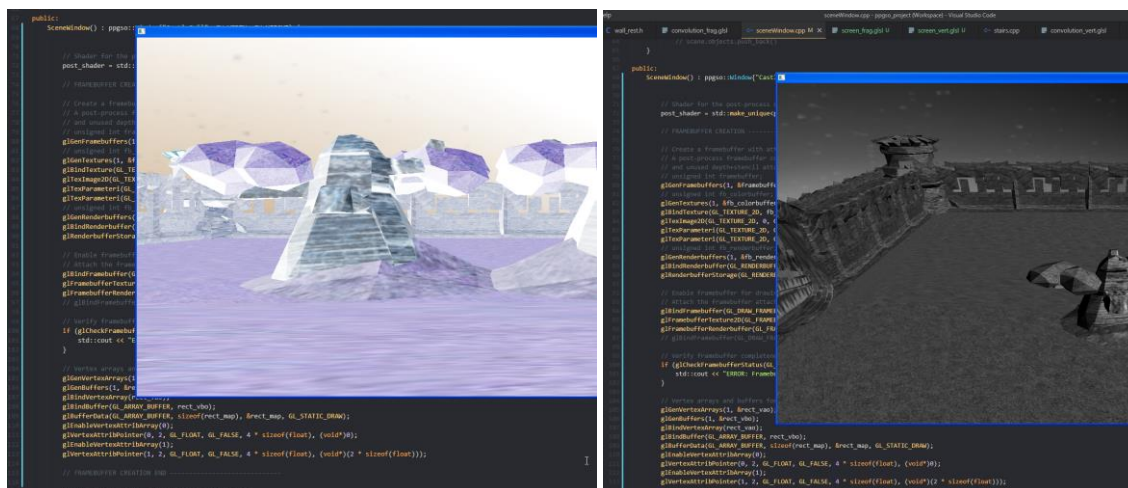
```
glm::vec3 Scene::calculateLightDirection(glm::vec3 object_pos) {
    if( ! this->hasLightSource ) return this->lightDirection;
    glm::vec3 result = object_pos - this->lightSource;
    float div = (abs(result.x) > abs(result.y)) ? ((abs(result.x) > abs(result.z)) ? abs(result.x) : abs(result.z)) : ((abs(result.y) > abs(result.z)) ? abs(result.y) : abs(result.z));
    if ( div < 1.0f ) div = 1.0f;
    result *= -1;
    return glm::vec3(result.x / div, result.y / div, result.z / div); // return normalized light dir
}
```

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 obrazu** 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

3.Scény, priestorové vzťahy

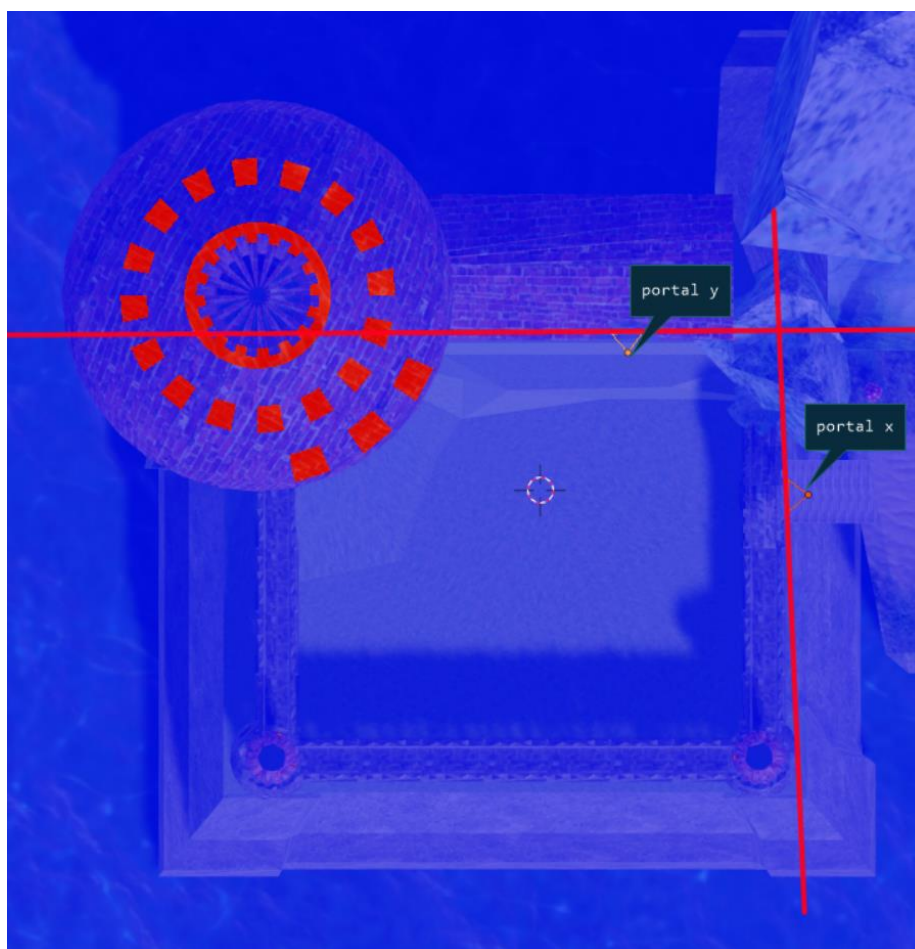
V tejto časti by som bližšie priblížil spôsob, ktorým sú scény menené, ako sú dáta o zmene scén uložené v Blenderi a následne z neho exportované pre použitie v projekte. Ďalej opäť spomeniem hierarchické transformácie.

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:

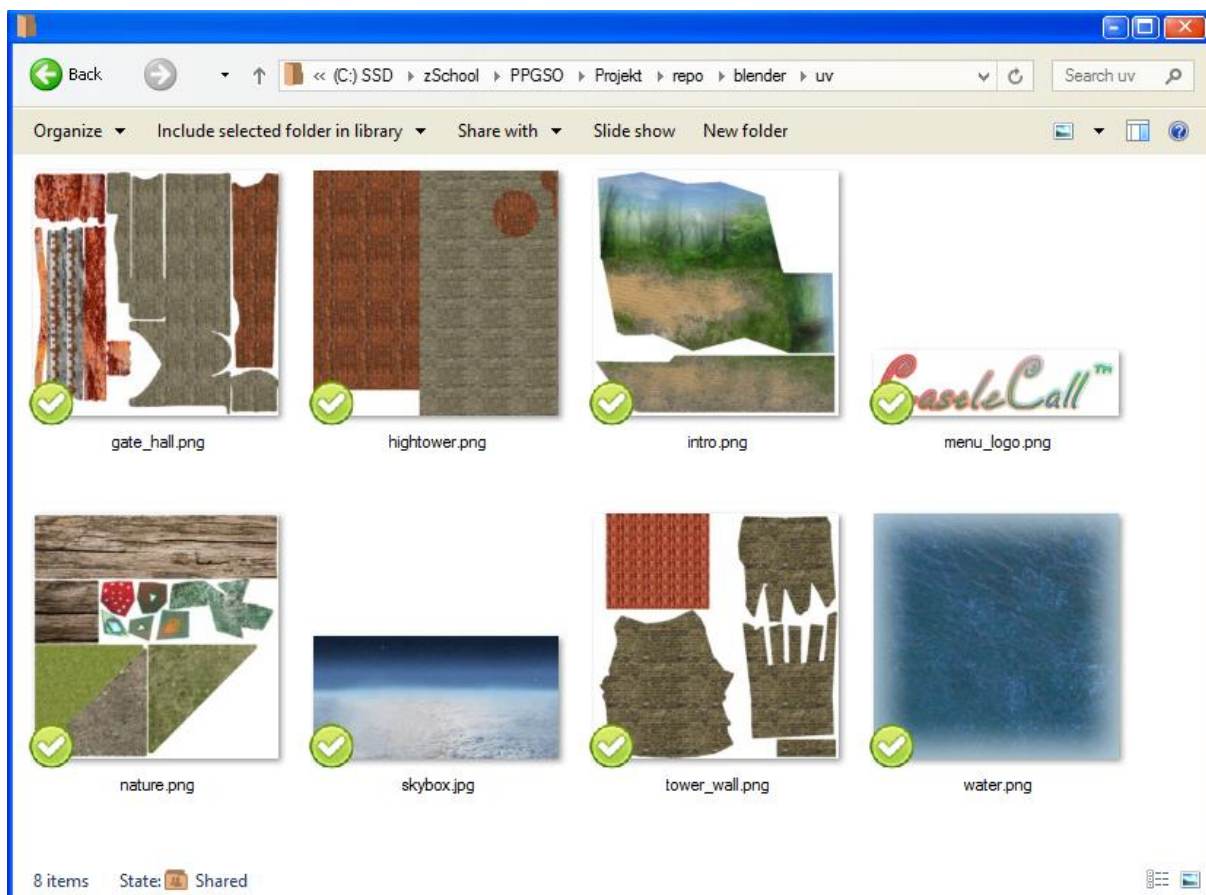
- vnesenia pocitu fluidity hráčovi,
- zaručeniu, že má stále pod sebou podlahu (aby nezačal padať do nekonečna)
- načítavanie objektov novej scény môže byť uskutočnené dynamicky na základe smerového vektora hráča a jeho rýchlosti na pozadí namiesto zobrazenia obrazovky načítavania, ktorá by oddialila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

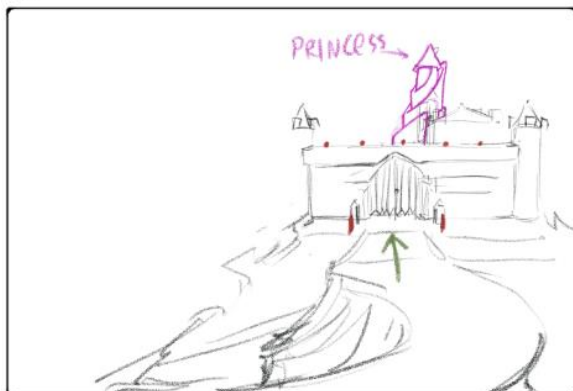
V pôvodnom pláne bola tiež zámena polygonálne zjednodušených objektov za kompletné pre lepšiu optimalizáciu aplikácie, to by avšak predĺžilo prácu na nebodovateľných častiach projektu, ktoré sú už aj napriek tomu obširné, keďže všetky scény sú zložené z plne vlastnoručne tvorených objektov a UV máp.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



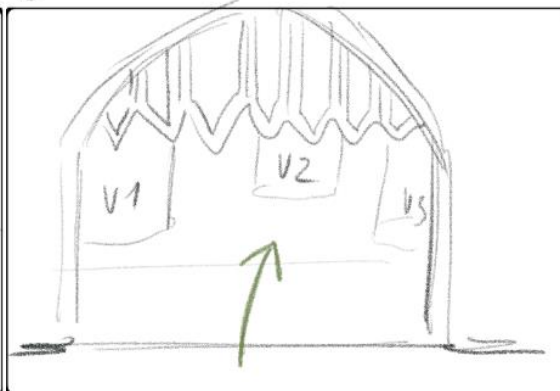
Prechod z prvej scény do druhej: Brána

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

①

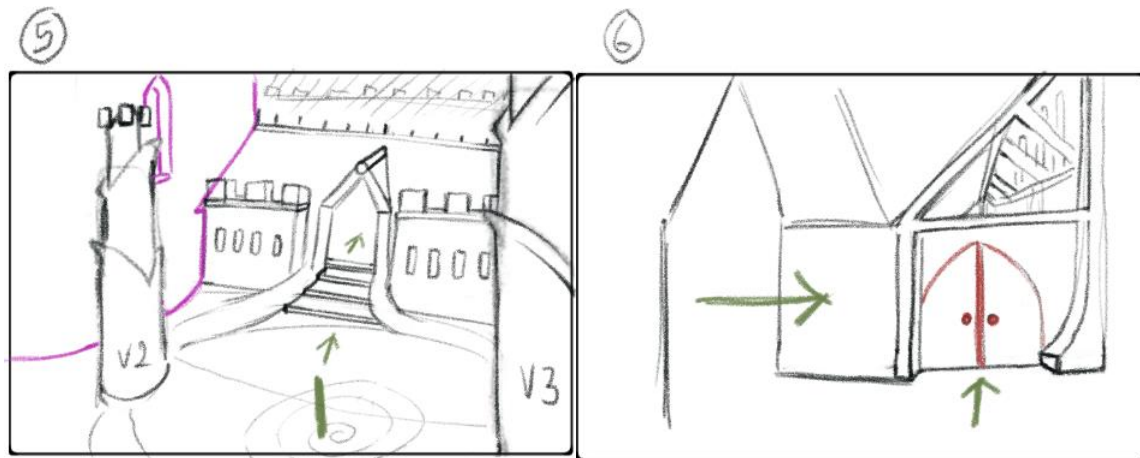


③ V → rezav



Prechod z druhej scény do tretej: Hala

Tento prechod načíta schody nachádzajúce sa vo vysokej veži.

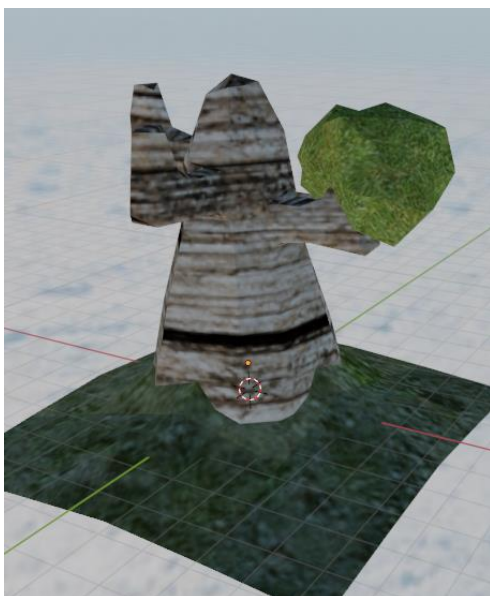


Hierarchické transformácie

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)

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;
}

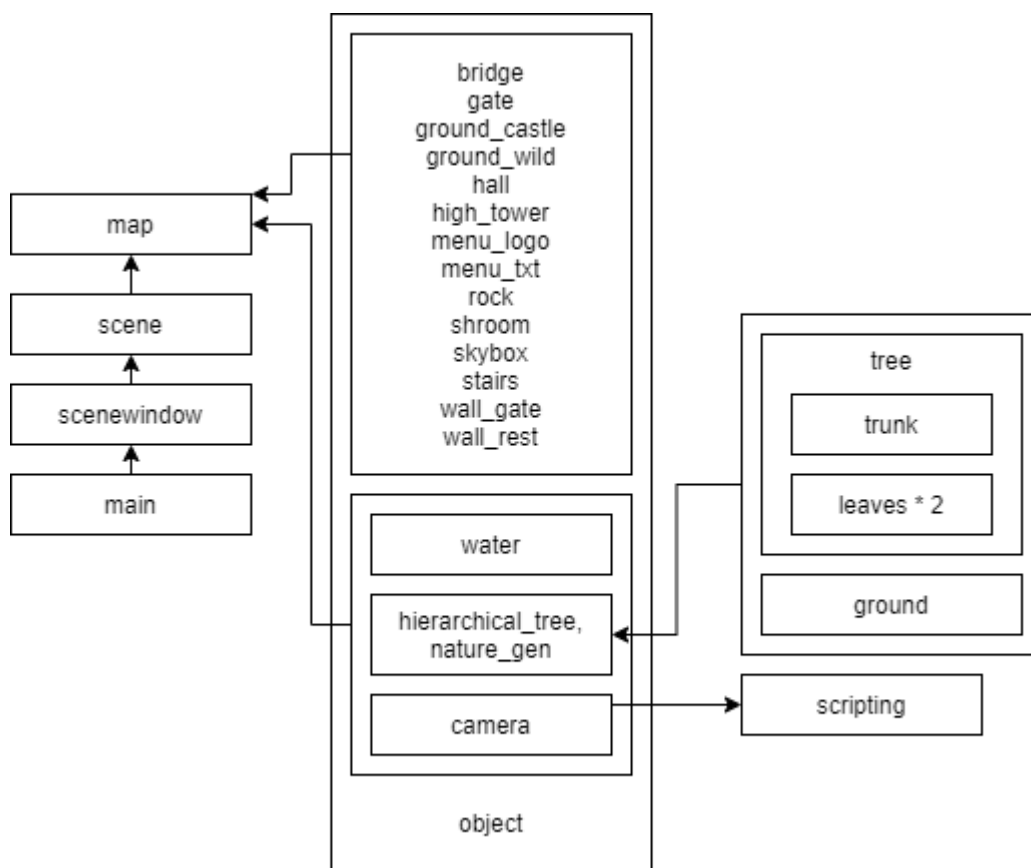
void Tree::render(Scene &scene) { ... }

HierarchicalTree::HierarchicalTree() { ... }

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

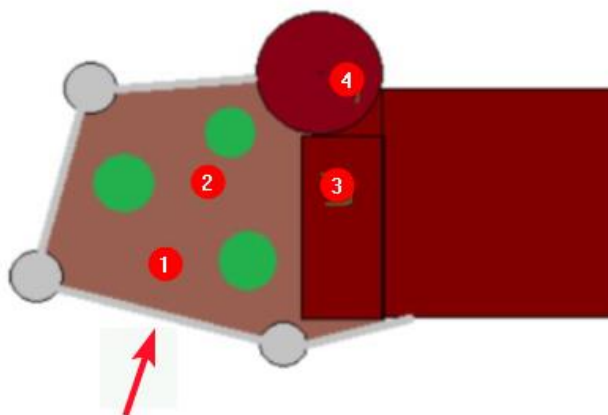
4. Diagram tried objektov scény

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



CastleCall™