SLOVENSKÁ TECHNICKÁ UNIVERZITA V BRATISLAVE
Fakulta informatiky a informačných technológií

Pavol Krajkovič, Michal Skaličan

Herňa

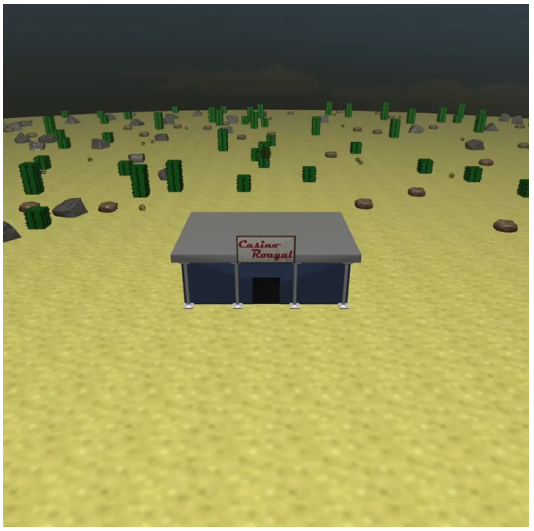
Princípy počítačovej grafiky a spracovania obrazu

Cvičenie:	Utorok 18:00
Vedúci cvičení:	Ing. Peter Kapec, PhD.
Ak. rok:	2021/2022

Storyboard a opis scén

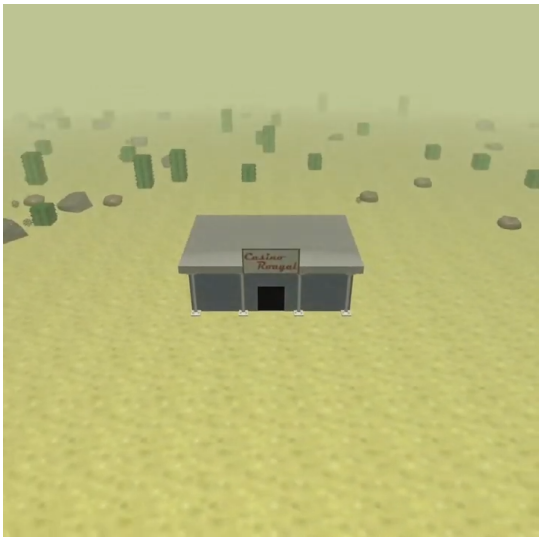
1.Scéna

V prvej scéne sa kamera pozrie na kasino z výšky a postupne začne rotovať v protismere hodinových ručičiek. Kamera je pri tomto pohybe stále vycenterovaná na kasíno s nejakým offsetom aby bol záber aj na okolitú krajinku. Počas toho kamera rotuje okolo kasína, príde piesočná búrka.



Úvodný pohľad na kasíno

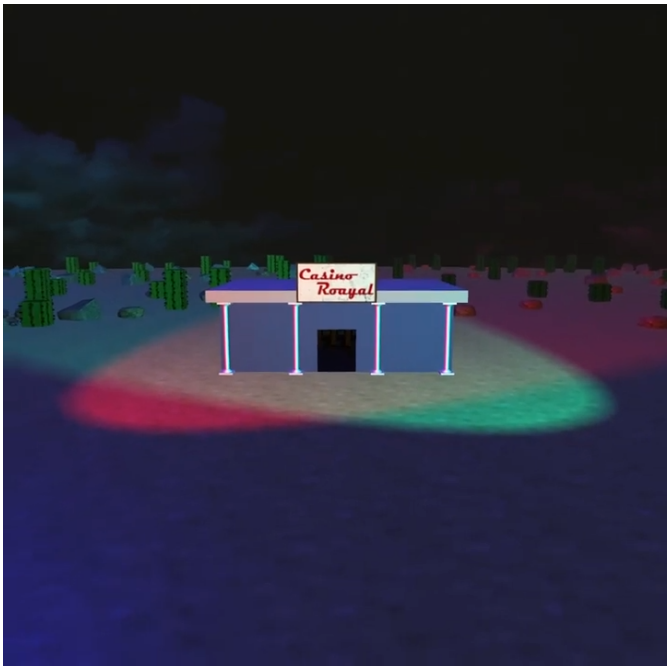
Počas piesočnej búrky máme zníženú viditeľnosť. Keď kamera dorotuje a zastane pred kasínom, tak búrka začne postupne miznúť a pomaly sa zmení deň na noc rotáciou kupole. Kamera sa pozrie kolmo hore ako sa mení deň na noc na hviezdy na nočnej oblohe. Slnko, ako zdroj svetla sa pri rotácií oblohy postupne presúva po x-ovej a z-ovej osy.



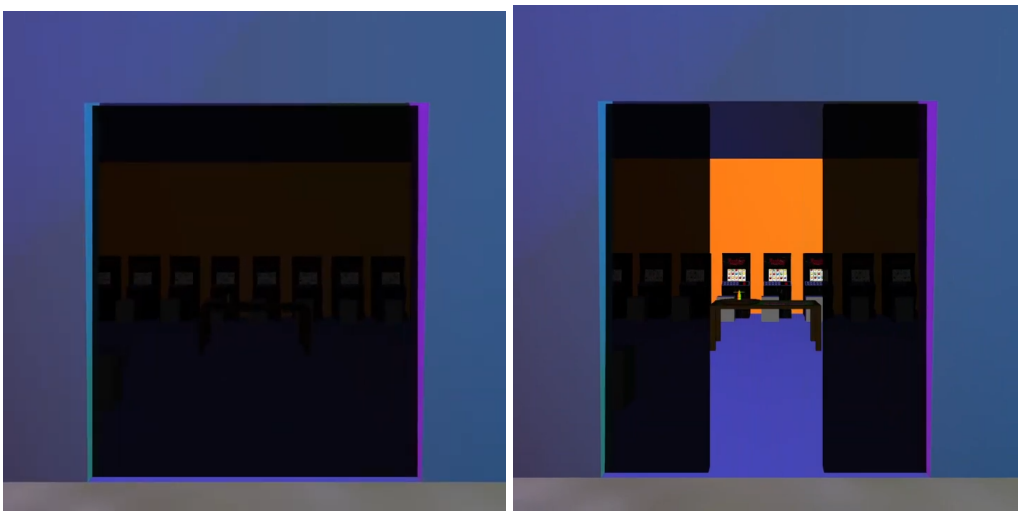
Pohľad na kasíno s piesočnou búrkou

Po tom ako sa kamera pozrie hore, tak sa zmení farba slnka na modrú, čo predstavuje noc. Kamera potom postupne klesne dole a znovu sa pozrie na kasíno. Ako sa

kamera pozerá na kasíno, tak sa postupne rozsvietia dve spotlight svetlá, ktoré v príbehu naznačujú otvorenie kasína a vzbúdzajú v sledovateľovi dojem zvedavosti, aby kamera sa priblížila a vstúpila do kasína. Kamera sa potom pozrie doľava a následne doprava, s čím sme chceli predať sledovateľovi nerozhodnosť, či vstúpiť do kasína alebo nie. Túto nerozhodnosť môžeme chápať ako jednoduchú otázku či niekto vidí sledovateľa, ktorý sa pozerá očami kamery, ako vstupuje do kasína.

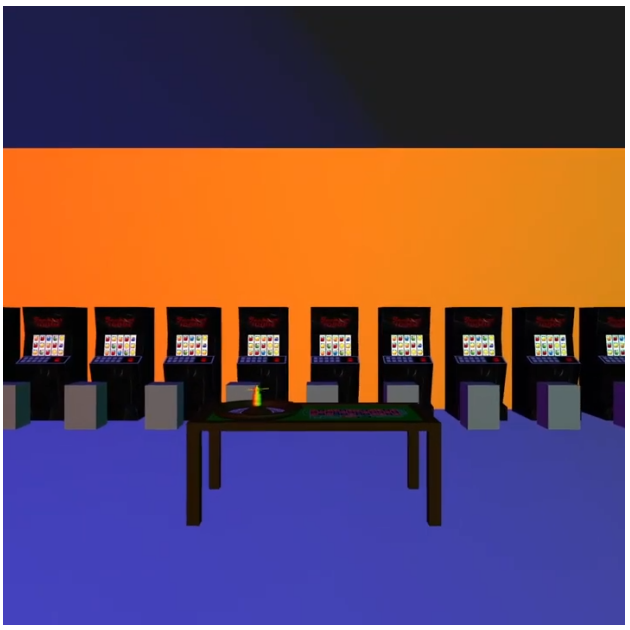


Následne sa kamera postupne približuje ku kasínu a pred dverami zastaví. Počká dokým sa dvere otvoria.



2. Scéna

Druhá scéna sa nachádza vo vnútri kasína. Kamera zastaví pred kasínom a otvoria sa dvere. Otváranie dverí je animované pomocou kľúčových snímkov. Po vstupe do kasína sa kamera obzrie na všetky strany. Po obvode kasína sú umiestnené výherné automaty.



Každý automat má podradený objekt obrazovky, ktorá zobrazuje textúru. Textúra má na každej obrazovke iný offset. Ruleta je riešená trojvrstvovou hierarchiou - stôl -> koleso -> hýbajúca sa časť kolesa

Ďalej kamera preletí popod ruletový stôl a otočí sa. V ľavom kúte sedí za automatom človek, kamera sa k nemu priblíži a začne ho sledovať ponad pleco. V tom momente sa nad ním zažne žiarovka, na automate sa zobrazí výherná obrazovka a začnú padať konfety.

Na zobrazenie výhry sa zmení textúra obrazovky za prednačítanú druhú textúru. Ruka stláčajúca tlačidlo automatu je animovaná procedurálne podľa sínusu a po výhre je pomocou kľúčových snímkov zdvihnutá hore.

Renderovanie scény

Výpočet matice modelu pre objekt

Výpočet matice objektu

Generovanie bodov pre procedurálny terén

Náš terén predstavuje plocha ktorá je ohraničená kupolou ktorá predstavuje nočnú a dennú oblohu. Objekty sú generované tak, že si vypočítame náhodné pozície v kruhu s polomerom kupoly. Ak je nejaký vygenerovaný bod v určitej vzdialenosti (3 jednotky) ako iný bod, tak sa bod zahodí a vygeneruje sa nový. Taktiež nemôžeme generovať objekty v blízkosti kasína, a preto sme zadali aj podmienku, ktorá berie vzdialenosť bodu od stredu (kasína) a ak je vzdialenosť menšia ako 30, tak sa bod nevygeneroval. Tieto podmienky slúžili na to aby sa objekty medzi sebou nepretínali.

```cpp
std::vector<glm::vec3> Utils::generatePoints(int radius, int n, glm::vec2 center) {
    std::vector<glm::vec3> points;

    // calculate radius and subtract offset so objects do not generate at the edge of the plane
    radius -= 10;

    while((int)points.size() != n){
        // calculate angle theta
        double theta = 2 * ppgso::PI * uniform();

        // Get length from center
        double len = sqrt(uniform()) * radius;

        float x = round((center.x + len * cos(theta))/2)*2;
        float y = round((center.y + len * sin(theta))/2)*2;

        bool dont_generate = false;
        for(auto i: points){
            if ( abs( x: x - i.x) < 3 && abs( x: y - i.z) < 3) {
                dont_generate = true;
                continue;
            }
        }
            }
        }
        if (dont_generate)
        {
            continue;
        }
        // if the point is located near the casino
        double d = sqrt( _X: pow( _X: (x - center.x), _Y: 2) + pow( _X: (y - center.y), _Y: 2));

        if ( d < 30 || (y > 30 && y < 50))
            continue;

        // the y coordinate is actually the z in our world system
        points.push_back({x, b: 0,y});
    }
    return points;
```

Svetelné zdroje

V projekte sme sa rozhodli použiť tri druhy svetelných zdrojov. Každý zdroj svetla ma zadefinovanú triedu v projekte, ktorá zahŕňa atribúty ako ambientné, difúzne a spekulárne zložky. Dostupné farby pre zdroj alebo konštanty podľa ktorých sa počíta hĺbka svetla.

Ako prvý zdroj svetla máme slnko, čo je vlastné smerový zdroj svetla. Druhý typ zdroju svetla je pointLight, ktorý predstavuje obyčajné svetlo ako žiarovku. Toto svetlo sa líši tým, že počíta aj hĺbku (attenuation). Tretí a posledný zdroj svetla je spotlight. Spotlight predstavuje lúč svetla ako z reflektorov. Pre krajší efekt spotlight osvetlenia, sa generujú dva kužele. Vonkajší a vnútorný. Svetlo sa medzi vnútorným a vonkajším kuželom postupne stlmuje.

Každé svetlo má svoju difúznu, ambientnú a spekulárnu zložku. Keďže nie každý objekt je taký istý, tak sme rôznym objektom nastavili rôzny materiál. Vďaka tomuto sme nasimulovali ako by sa svetlo odrážalo od rôznych povrchov ako je napríklad kaktus alebo kameň.

```
class Light {

public:
    glm::vec3 diffuse;
    glm::vec3 ambient;
    glm::vec3 specular;
    glm::vec3 position;
    bool isActive = false;
    float turnOn = -1.f;
    Light() = default;
    Light(const Light&) = default;
    Light(Light&&) = default;
    virtual ~Light() {};


    virtual bool update(Scene &scene, float dt) = 0;

};
```

Abstraktná trieda svetla

Svetlá sa vkladajú pri vytváraní scény s ich potrebnými atribútmi pre iniclializáciu + čas v sekundách, kedy sa svetlá majú zapnúť. Každé svetlo implementuje svoju vlastnú update funkciu.

```
bool ColorBulb::update(Scene &scene, float dt) {

    if (scene.age >= turnOn)
    {
        isActive = true;
    }
    if (animate)
    {
        int switch_color = static_cast<int>(scene.age) % 4 ;
        ambient = { a: colorLights[switch_color].x * 0.1,  b: colorLights[switch_color].y * 0.1,  c: colorLights[sw
        diffuse = {colorLights[switch_color].x,  colorLights[switch_color].y,  colorLights[switch_color].z};
        specular = {colorLights[switch_color].x,  colorLights[switch_color].y,  colorLights[switch_color].z};
    }
    return true;
}
```

Update metóda objektu ColorBulb

Svetlá sa v shaderi počítajú iteráciou cez všetky zdroje svetla pre každý typ, a vráti sa vector3 s hodnotami fragmentu vďaka ambientného, difúzneho a spekulárneho svetla. Následne sa tento výsledok prenásobi textúrou a výsledná farba fragmentu je posunutá do sekcie post processingu. Ako posledný krok je nastavená priesvitnosť objektu.

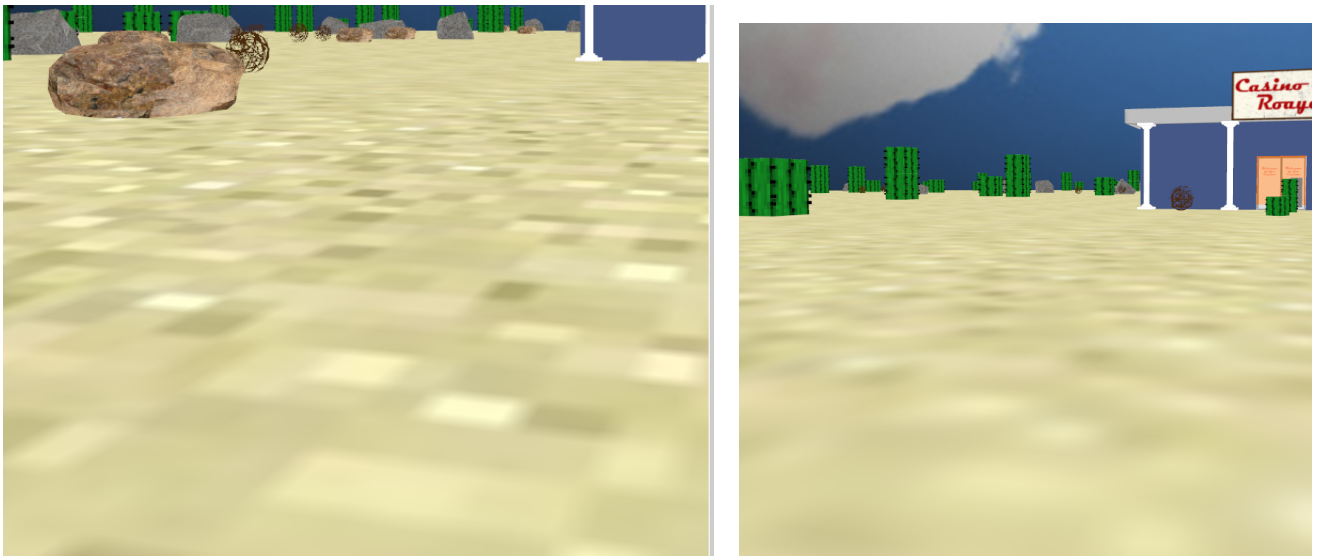
Postprocessing

V projekte máme dva typy postprocessingu, blur a fog na základe depth mapy. Keďže sa nám nepáčila textúra piesku, tak sme sa namiesto upravenia textúry rozhodli použiť rozmazanie na základe konvolučných filtrov.

```
float factor = 256.0;
int index = 0;
vec4 color = vec4(1);
if(isTerrain == 1) {

}
for (int i = -2; i <= 2; i++) {
        for (int j = -2; j <= 2; j++) {
          vec2 shift = vec2(i,j) / textureSize(Texture,0);
          color += kernel[index++] * texture(Texture, vec2(TexCoords.x, 1.0 - TexCoords.y) + shift) ;
        }
}
color = color / factor * vec4(result,1);
```

Postprocessing - konvolučný filter



Textúra bez rozmazania na ľavo a s rozmazaním na pravo

Tento filter je použitý iba čisto na terén a teda pieskovú textúru. Výsledky môžeme vidieť nižšie.

Ako druhý druh postprocessingu sme sa rozhodli pridať piesočnú búrku na základy hĺbky fragmentu v scéne. Pre tento druh postprocessingu využíva shader informáciu o scéne a presne to je čas scény.

```
vec4 fogColor = vec4(190/255.0,196/255.0,147/255.0,1);

if (SceneAge > 13 && SceneAge <40)
    FragColor = color + (fogColor - color) * min(dist / max((1500 - (SceneAge - 13)*100),150),1);
else if (SceneAge>40)
     FragColor = color + (fogColor - color) * min(dist / (150 + (SceneAge - 40)*75),1);
else
       FragColor = color + (fogColor - color) * min(dist / 1500 ,1);
```

Výpočet búrky s ohľadom na čas

Kolízie

Kolízie sú riešné tak, že každý objekt má svoj bounding box určený dvoma bodmi. Jeden bod uchováva informácie o najmenších súradniciach a druhý o najväčších. Pri kolízií potom kontrolujeme pre každú os, či sa jedna z hraníc jedného objektu nenachádza medzi hranicami druhého objektu a naopak, keďže jeden z objektov by nemusel mať v jednom smere nenulový rozmer.

```cpp
bool Object::check_collision(glm::vec3 pos, Scene &scene){
    // skontroluj pre každý objekt
    for (const auto& obj: scene.objects) {
        // objekt nemôže mať koliziu sám so sebou
        if (obj.get() == this) continue;
        if (obj->can_collide) {
            // funkcia na kontrolu kolízie konkrétnej osi
            auto collision = [](float one_s, float one_l, float two_s, float two_l) {
                return ((two_s <= one_s && one_s <= two_l) || (two_s <= one_l && one_l <= two_l)) ||
                       ((one_s <= two_s && two_s <= one_l) || (one_s <= two_l && two_l <= one_l));
            };

            auto collisionX = collision( one_s: pos.x + this->bounding_box[0].x,  one_l: pos.x + this->bounding_box[1].x,
                                         two_s: (obj->position.x + obj->bounding_box[0].x),
                                         two_l: (obj->position.x + obj->bounding_box[1].x));

            auto collisionY = collision( one_s: pos.y + this->bounding_box[0].y,  one_l: pos.y + this->bounding_box[1].y,
                                         two_s: obj->position.y + obj->bounding_box[0].y,
                                         two_l: obj->position.y + obj->bounding_box[1].y);
            auto collisionZ = collision( one_s: pos.z + this->bounding_box[0].z,  one_l: pos.z + this->bounding_box[1].z,
                                         two_s: obj->position.z + obj->bounding_box[0].z,
                                         two_l: obj->position.z + obj->bounding_box[1].z);
            //pokiaľ je na všetkých osach priesečník, objekt do niečoho narazil
            if (collisionX && collisionY && collisionZ) {
                return true;
            }
        }
    }
    return false;
}
```

Do funkcie je poslaná pozícia na ktorú chce objekt ísť a skontroluje sa či na tej pozícií s niečím nekoliduje. Pokiaľ koliduje, objekt sa tým smerom nepohne, pokiaľ nie tak ide ďalej v tom smere.

Kľúčové snímky

Pomocou kľúčových snímkov animujeme niektoré objekty a kameru. Reprezentácia týchto snímkov sa pri objektoch a kamere jemne líši.

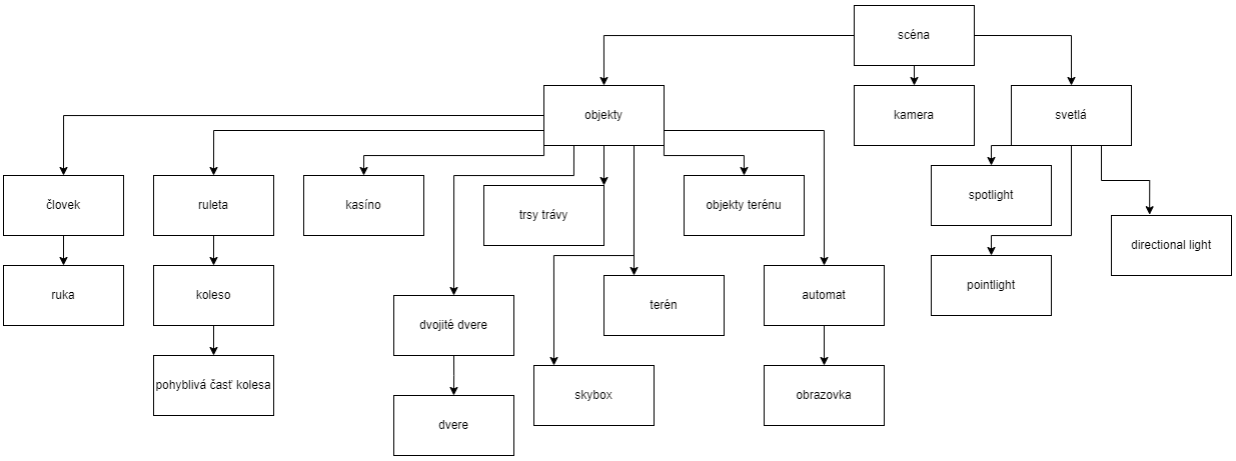
```cpp
struct key_frame {
    glm::vec3 position;
    glm::vec3 center;
    const glm::vec3 duration;
    float age = 0;
};
```

Kľúčový snímok kamery si udržiava position - kde sa má kamera nachádzať, center - kam sa má kamera pozerať, duration - koľko má trvať prechod medzi snímkami a age - ako dlho už prechod trvá.

```cpp
struct key_frame {
    glm::vec3 duration;
    const glm::vec3 position;
    const glm::vec3 rotation;
    const glm::vec3 scale ;
    float age = 0;
};
```

Kľúčový snímok objektu si uchováva pozíciu, rotáciu a škálovanie objektu. Ďalej si uchováva duration a age rovnako ako pri kamere.

Diagram scény



Zhodnotenie

Zo zadania projektu sa nám okrem tieňov podarilo splniť takmer všetky body. Niektoré implementácie sú lepšie, niektoré by potrebovali ešte trochu dorobiť ale funkčné by mali byť všetky. Toto zadanie bola veľká výzva a dali sme do nej všetku našu energiu a čas čo sme mali, výsledok však minimálne pre nás stál za to a napriek niektorým nedostatkom sme s ním spokojní.