Računarski fakultet

Diplomski rad

Predmet: **Računarska grafika**

Tema: **Simulacija svetla u 2D prostoru**

Mentor: **Prof. dr Dragan Mašulović**

Student: **Marko Srećković**

Predgovor

Računarska grafika je predmet za koji sam uvek imao veliko interesovanje i kojim se se već godinu dana profesionalno i bavim. Iz tih razloga sam odlučio da izaberem temu iz predmeta računarske grafike. Tema koju sam izabrao bavi se grafikom u 2D prostoru, razlog biranja ove teme jeste jer mi ta oblast grafike daje najviše prostora za originalnost.

Želeo bih da se zahvalim mojim roditeljima i mojoj sestri jer su me podržavali u tome da upišem ovaj fakultet i podržavali u svakom smislu prilikom studiranja sve do samog kraja. Takođe zahvaljujem se i mentoru prof. dr Draganu Mašuloviću koji je pratio ceo proces nastajanja diplomskog rada i svojim savetima i entuzijazmom me usmeravao kako da prevladam probleme koji su se pojavili prilikom izrade rada.

Sadržaj

Uvod

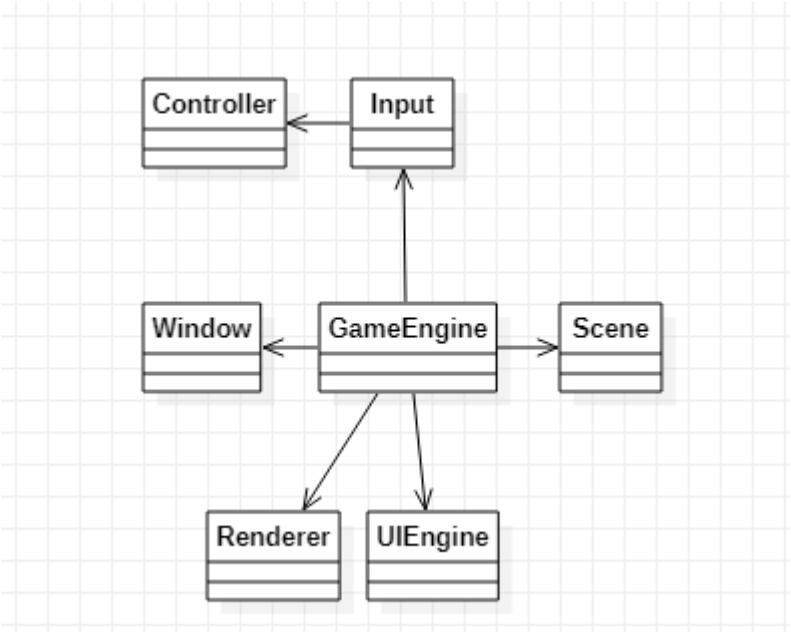
Diplomski rad “Simulacija svetla u 2D prostoru” ima za ideju crtanje scena u 2D prostoru sa akcentom na osvetljenje 2D površina što podstiče realističnost i kompleksnost scene. U ovom projektu ne simuliramo zakone fizike nego pokušavamo pomoću dosetki da napravimo sliku koja daje utisak realističnosti. Svrha ovog projekta je pružanje alata umetnicima kojim onu mogu da naprave realistične scene ili računarske igre u 2D prostoru.
Projekat pokriva simulaciju okluzije svetlosti, simulaciju osvetljavanja neravnih povrsina, simulaciju raspona i količine svetlosti kao i potrebne optimizacije kako bi ovaj projekat radio u realnom vremenu.
Okluzija svetosti je postignuta pomoću “ray-casting-a”*[3]*, simulacija osvetljivanja neravnih površina pomoću posebnih tekstura koji se koriste za 3D grafiku - normal mape*[1]* a simulacija raspona i količine svetlosti pomoću atenuacije svetlosti*[2]* i izlaznih tekstura iz predhodna dva koraka. Detaljnije o ovome govorićemo u sledećim glavama.
Jezik u kom je pisan projekat je C++ koji je korišćen zajedno sa OpenGL grafičkom bibliotekom*[4]*. Za konfiguraciju projekta koristio sam premake*[5]*. Source control koji sam koristio je Git.
Biblioteke koje su korišćene u ovom projektu su:

- GLFW (Graphics Library Framework) - Za multiplatform kreiranje prozora i interakciju sa inputom
- GLM (OpenGL mathematics) - za matematičku podršku
- Dear ImGUI - za renderovanje UI komponenti
- Stb_image - za učitavanje tekstura
- GLAD - za učitavanje dinamičke biblioteke OpenGL

Slična rešenja poput ovog koriste veliki game engini kao što su Godot*[6]* i Unity*[7]* i oni su me inspirisali da implementiram algoritme za neke od efekata koje ovi engini podržavaju. Rešenje prikazano ovde je fleksibilnije (može se upravljati većim brojem parametara za renderovanje) dok veći game engini implementiraju više efekata. Takođe mnogo više vremena po frejmu je ulozeno u mom radu na samu simulaciju svetla što dovodi do realnije slike. Veliki engini imaju mnogo više elemenata tako da stavljaju manji fokus na simulaciju svetlosti u 2D prostoru.

Organizacija engina

Glavna organizacija engina se može videti na sledećoj slici:



Game engine je polazna tačka programa i on je zadužen za životni ciklus aplikacije i ovih pojedinačnih modula kao i učitavanje demo scena, računanje delta time vrednosti koji će se koristiti kasnije u svim ostalim modulima itd.
Životni ciklusi ovih modula imaju dve faze. Prva faza je inicijalizacija koja se izvršava pri samom pokretanju engina, a druga faza je faza ažuriranja koja se dešava u svakoj vremenskoj jedinici engina. Renderer i UIEngine imaju još jednu fazu a to je render faza koja se dešava u jednakim intervalima tako da se postigne FPS koji je zadat u konfiguraciji projekta.

Window

Window klasa koristi GLFW i ima ulogu u kreiranju i održavanju prozora kao i očitavanju pozicije miša i prosleđivanju inputa tastature u engine.

Input

Input je klasa koja mapira input sa tastature na postavljene callback funkcije kao i očitavanje pozicije miša u koordinatnom prostoru ekrana.
Postoje tri tipa callback funkcija:

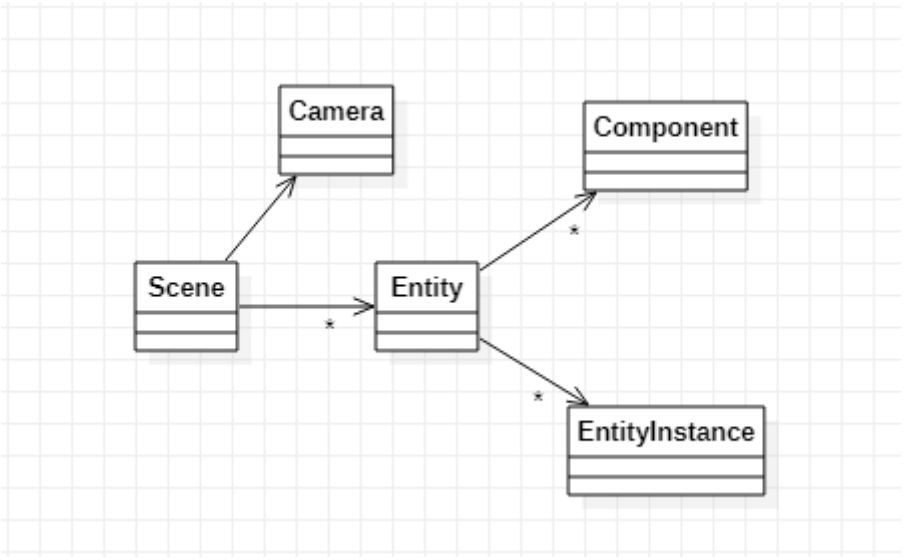
- KeyInputCallback - poziva se svaki frejm dok je određen taster pritisnut
- KeyPressedCallback - poziva se samo prvi frejm kada je taster pritisnut
- StateChangedCallback - poziva se kada taster promeni stanje,tj. kada iz pressed stanja uđe u released i obrnuto.

Controller

Controller je apstraktna klasa koja treba da sadrži input callback funkcije i njihovo mapiranje na određene inpute. U projektu postoji realizacija ove klase koja se naziva PlayerController koja ima callback funkcije za pomeranje kamere, pomeranje igrača, izlaza iz igre kao i neke debug akcije kao što je reload shadera.

Scene

Scene sadrži sve informacije o svetu i objektima u njemu. Renderer koristi ove informacije da bi znao na koji način treba da nacrta scenu. Struktura scene se može videti na sledećoj slici:



Camera sadrži informacije o posmatraču na sceni kao što su pozicija, rotacija i zoom.
Entity je glavna jedinica scene, svaki entitet sadrži informacije koje su potrebne za crtanje na sceni kao što su teksture entiteta, flagovi za crtanje (da li odbija svetlost ili emituje svetlost, da li je u pozadini ili ispred…) kao i podešavanja za stanja.
Kako bi se entitet iscrtao na sceni on mora da se instancira, tj. da se napravi jedinstvena kopija koja ima svoju poziciju u svetu, a to je Entity Instance. Entity instance sadrži samo informacije potrebne za jednu instancu entiteta kao što je

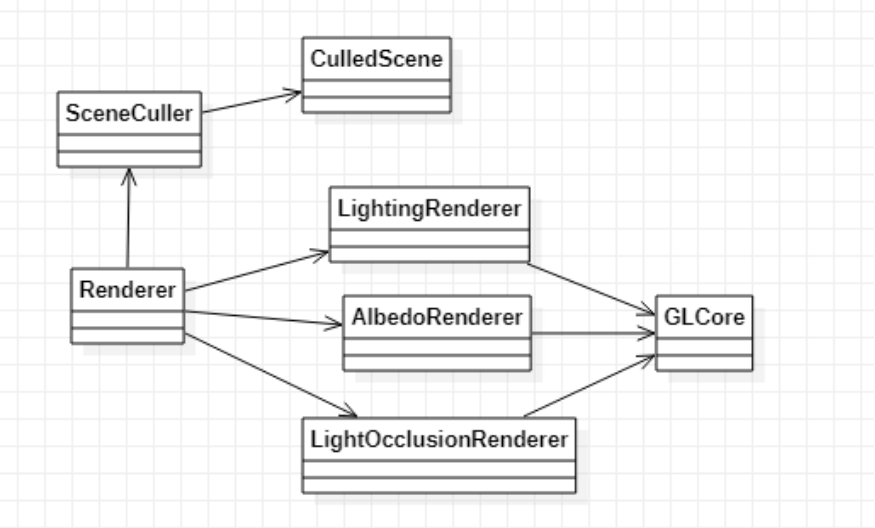
pozicija, veličina i rotacija dok su ostali parametri memorisani unutar entiteta. Na taj način se izbegava učitavanje istih resursa, npr. pri renderovanju šume gde imamo jednu teksturu stabla a hoćemo da renderujemo više stabala, umesto da napravimo više entiteta koji će svako zasebno da učitava teksture i sadrži dodatne informacije za crtanje mi napravimo jedan entitet drveta i instanciramo ga potreban broj puta pri čemu svaka instanca ima drugu lokaciju. Ova arhitektura je implementacija Flyweight dizajn šablona.
Component klasa predstavlja logički deo entiteta koji se ažurira u svakoj vremenskoj jedinici engina. Primer komponente jeste MouseFollowerComponent koji kad se doda na entitet uvek poziciju postavlja na poziciju miša tako da izgleda kao da entitet prati miša.

UIEngine

UIEngine služi za crtanje UI elemenata. Koristi ImGUI kao biblioteku za crtanje. Struktuirana je na način za jednostavnu implementaciju zasebnih elemenata. UIEngine sadrzi listu UIElement objekata i stara se za njihov životni ciklus. Kreiranje novih UIElemenata je vrlo jednostavno, treba samo naslediti klasu UIElement i implementirati metode životnog ciklusa elementa, a to su Init,Update i Render. Init sadrži logiku za inicijalizaciju elementa, Update sadrži samo logiku elementa a Render sadrži informacije kako hoćemo da izgleda taj element. Ova struktura je kompozitna, jer možemo u jednom UIElementu da imamo više manjih UIElemenata, samo moramo da se postaramo da roditelj održava životni ciklus dece.

Renderer

Renderer ima ulogu u održavanju grafičkog dela engina. Osim samog crtanja ima ulogu i u upravljanju resursima kao što je učitavanje i osvežavanje šejdera, kao i učitavanje tekstura entiteta u memoriju. Na sledećoj slici je prikazana struktura renderera:



Renderer sam po sebi ne crta ništa, nego delegira crtanje na ova tri njemu podređena podsistema.

Na sledećoj slici možemo videti strukturu jednog frejma iz perspektive renderera u kodu:

```
void Renderer::RenderFrame()
{
    PROFILE_SCOPE("RenderFrame");

    CulledScene& culledScene = m_SceneCuller->GetCulledScene(m_Scene);

    GLFunctions::ClearScreen();

    m_OcclusionRenderer->   RenderOcclusion     (culledScene);
    m_AlbedoRenderer->      RenderBackground    (culledScene);
    m_AlbedoRenderer->      RenderBase          (culledScene);
    m_LightingRenderer->    RenderLighting      (culledScene);
    m_AlbedoRenderer->      RenderOccluders     (culledScene);
    m_LightingRenderer->    RenderEmitters      (culledScene);
    m_AlbedoRenderer->      RenderForeground    (culledScene);
    DebugRenderer::Get()->  RenderDebug         (culledScene);
}
```

SceneCuller kao input uzima scenu i kao rezultat vraća CulledScene koja se prosleđuje ostalim rendererima. CulledScene predstavlja podskup scene sa elementima koji su vidljivi ili moraju da se iscrtaju u tom frame-u.

LightOcclusionRenderer crta masku okluzije svetla, albedo renderer crta albedo teksture koje kombinuje sa normal mapama i onda lighting renderer spaja te dve teksture kako bi se dobila finalna tekstura. U kasnijim poglavljima će biti više reči o ovom procesu.

GLCore sam po sebi nije klasa već fajl koji u sebi sadrži klase niskog nivoa koje su zadužene za komunikaciju sa OpenGL bibliotekom.

GLCore sadrži sledeće klase:

- ShaderInput - predstavlja index i vertex buffer enkapsuliran u ovu klasu
- UniformBuffer - read only buffer
- ShaderStorage - mapira se na OpenGL SSBO *[9]*
- Texture - tekstura u GPU memoriji
- Shader - moze biti i vertex+*geometry*+fragment ili može biti compute
- Framebuffer - tekstura na koju možemo da crtamo

GLCore osim klasa sadrži niz funkcija koje se nalaze pod namespaceom GLFunctions. Tu spadaju funkcije koje direktno komuniciraju sa APIjem kao što su Draw, Dispatch, ClearScreen, EnableAlphaBlending itd.

Shaderi

U ovom enginu postoji malo proširenje mogućnosti shadera koji se procesuju u enginu pre same kompilacije shadera. Shader se nalazi u jednom fajlu sa ekstenzijom .glsl gde mogu da se nalaze različiti tipovi šejdera, kao što su VERTEX, FRAGMENT, GEOMETRY, COMPUTE. Ovi tipovi su podeljeni direktivom #start <tip>. Na sledećoj slici se nalazi primer jednog shadera:

```
#start VERTEX

layout(location = 0) in vec2 in_Position;
layout(location = 1) in vec2 in_UV;

out vec2 UV;

void main()
{
    gl_Position = vec4(in_Position, 0.0, 1.0);
    UV = in_UV;
}

#start FRAGMENT

in vec2 UV;

uniform float u_Weight;

layout(binding = 0) uniform sampler2D u_Sampler1;
layout(binding = 1) uniform sampler2D u_Sampler2;

layout(location = 0) out vec4 FinalColor;

void main()
{
    FinalColor = ((texture(u_Sampler1, UV) * u_Weight) + (texture(u_Sampler2, UV) * (1.0f - u_Weight)));
}
```

Takođe podržana je i include direktiva, tako da možemo da imamo deljeni kod izmedju šejdera. Postoji jedan poseban fajl shaders/common.h koji sadrzi definicije koje su zajedničke za engine i šejdere.
GLSL verzija odredjena je u enginu i ne treba da se navodi u shaderu i ona je trenutno konfigurisana na 4.3.

Prostor

Najzastupljenija tehnika za predstavljanje objekata u 3D prostoru je pomoću MVP matrica*[8]*, sličnu tehniku možemo da primenimo i na 2D prostor. Razlika od tehnike u 3D svetu je to što koristimo 3x3 matrice i afine transformacije*[3]* u 2D prostoru. Razlog korišćenja jedne dimenzije više u 2D prostoru je isti kao i za 3D prostor, a to je da bismo omogućili transformacije koje pomeraju koordinatni početak tj. transliranje.
Objekti u sceni imaju transformaciju M koja ih premešta u prostor sveta.
Na sceni sa takodje nalazi i kamera sa svojom transformacijom V. Pomoću inverzne transformacije V možemo objekat koji se nalazi u prostoru sveta da premestimo u prostor kamere što će da bude konačan prostor koji će se prikazivati na ekranu pošto u 2D prostoru nemamo potrebu da vršimo projekciju.

Okluzija svetlosti

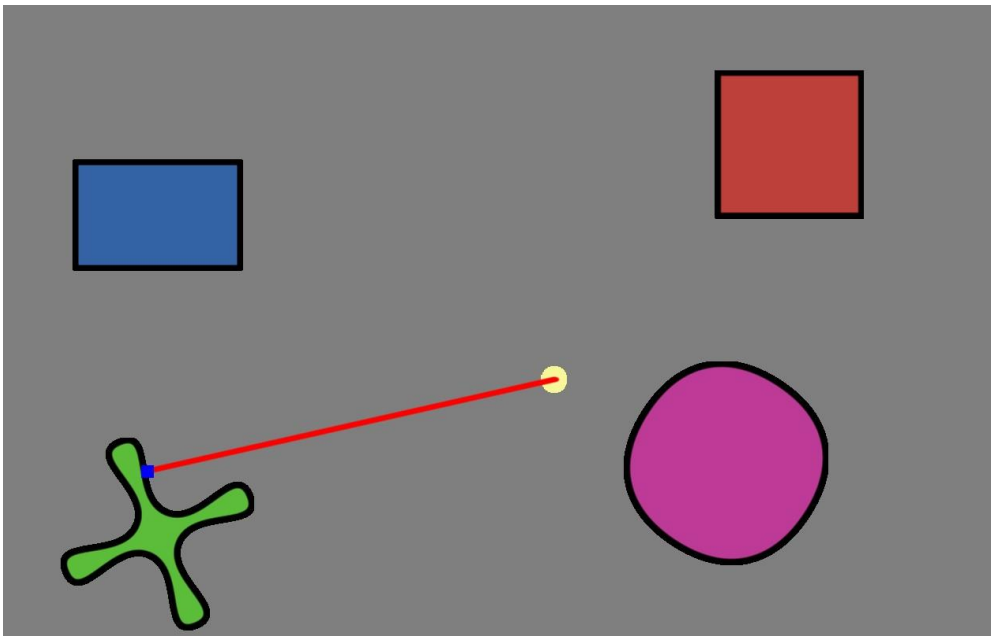
Okluzija svetlosti predstavlja efekat osvetljenja koji se dešava kada objekti na neki način blokiraju izvor svetlosti. Najvidljivija posledica okluzije svetosti jesu senke na kojima i jeste najveći fokus na ovom projektu. Pošto se radi o 2D reprezentaciji sveta ne moramo da imitiramo stvarne fizičke fenomene nego možemo sami da definišemo kako se ona manifestuje u našem svetu.

Ideju kako bi ovaj efekat trebao da izgleda u 2D prostoru sam uzeo od poznatih game engina kao što su Unity i Godot. Glavna ideja u tim enginima jeste da se koristi ray casting kako bi se odredila površina koju jedan izvor svetlosti pokriva.

Svaki entitet na sceni može da bude emitter, a može da bude i occluder. Entiteti koji su emitteri predstavljaju izvor svetlosti. Emitteri imaju i dodatne parametre kao što su boja svetlosti i opseg svetlosti.
Sa druge strane imamo occludere, oni su ti koji blokiraju svetlost. U odnosu na oblik razlikujemo dva tipa occludera, Rect occluderi i Mesh occluderi. Rect occluderi imaju oblik pravugaonika koji ima veličinu samog objekta na sceni. Mesh occluderi su malo komplikovaniji, kod njih izračunavamo približan poligon koji okružuje entitet pri učitavanju, o ovom izračunavanju biće reči u daljem tekstu. Takođe mesh occluderi imaju parameter *meshLod* koji označava detaljnost poligona koji se računa.

Raycasting

Raycasting je tehnika koja se veoma često koristi u računarskoj grafici i u industriji računarskih igara i zasniva se na ispaljivanju zrakova (eng. ray). Zrak na sceni je definisan svojom pozicijom i svojim vektorom pravca. Zadatak raycastinga jeste da u nekom prostoru u kom imamo jasno definisane objekte možemo da ispalimo zrak i dobijemo informaciju o putanji tog zraka. Kako se zrak ponaša na sceni na nama je da definišemo. U našem slučaju zrak ide svojim definisanim pravcem dok ne udari u prvi objekat na sceni.



Kao što možemo videti na slici, žuta tačka je pozicija zraka, crvena linija predstavlja putanju, a plava tačka je pozicija gde je ona udarila prvi objekat.
Naivna implementacija ovoga bi mogla biti da simuliramo zrak interativno pomerajući se za neko malo rastojanje sve dok ne uđemo u neki objekat ili izađemo iz ekrana. Ova implementacija nije toliko efikasna, a efikasnost nam je jako bitna jer je ovo operacija koja će se koristiti veliki broj puta u sekundi. Takođe nije ni konzistentna jer performansa zavisi od toga koliko je objekat udaljen od zraka.
Drugo rešenje je da matematički izračunamo tu poziciju, ali za ovo nam trebaju tačno definisani objekti na sceni. U implementaciji koja je pokazana u ovom diplomskom radu svaki objekat je definisan nizom duži koje zajedno prave jedan poligon.
Kada imamo niz duži samo treba da izračunamo presek svake duži sa zrakom i onda uzmemo onaj presek koji je najbliži kao rešenje. U slučaju da ne nađemo ni jedan presek tada ćemo uzeti presek sa ivicama ekrana.
Samo još treba da rešimo problem kako da nađemo presek duži i zraka.

Presek duži i zraka

Da bismo mogli matematički da izračunamo presek, prvi korak je matematički predstaviti zrak i duž. I duž i zrak ćemo predstaviti u sledećem formatu:
$p = A + tB$
gde A predstavlja početnu tačku, B pravac kretanja a t vreme kretanja. Tako da nam je p u ovoj formuli tačka koja predstavlja poziciju u kojoj se nalazi zrak nakon t vremenskih jedinica.

Ako imamo zrak koji kreće iz tačke R_o u pravcu vektora R_d onda zrak možemo matematički prikazati kao:

$R: p = R_o + tR_d,\ t \geq 0$

Ako imamo duž ograničenu tačkama A sa jedne strane i B sa druge strane onda duž možemo predstaviti kao:

$AB:\ p = A + t(B - A)\ \ t \in [0, 1]$

Da bismo našli presek između R i AB moramo da nađemo p za koje obe ove jednačine važe. p ćemo naći pomoću sledećeg sistema jednačina:
$p = R_o + t_1 R_d$
$p = A + t_2(B - A)$

Moraćemo takođe da izračunamo i t_1 i t_2 da bismo mogli da proverimo dodatne uslove koje smo postavili u definiciji.
Ovaj sistem jednačina možemo spojiti u jednu jednačinu po p i on izgleda ovako:
$R_o + t_1 R_d = A + t_2(B - A)$ što možemo dalje razdvojiti po x i y komponentama da bismo dobili sistem jednačina.

$X_{Ro} + t_1 X_{Rd} = X_A + t_2(X_B - X_A)$
$Y_{Ro} + t_1 Y_{Rd} = Y_A + t_2(Y_B - Y_A)$

Rešavanjem ovog sistema dobijamo rešenje:

$t_1 = \frac{X_A + t_2(X_B - X_A) - X_{Ro}}{X_{Rd}},\ t_2 = \frac{X_{Rd}(Y_A - Y_{Ro}) - Y_{Rd}(X_A - X_{Ro})}{Y_{Rd}(X_B - X_A) - X_{Rd}(Y_B - Y_A)},\ X_{Rd} \neq 0 \wedge Y_{Rd}(X_B - X_A) - X_{Rd}(Y_B - Y_A) \neq$

Prvi uslov se odnosi na slučaj kada je pravac zraka paralelan sa X osom u tom slučaju možemo dobiti direktno t_1 a t_2 dobiti preko t_1. Osim ovih uslova imamo

uslove i iz definicije zraka i ravni, tako da ako nije ispunjen uslov $t_1 \geq 0 \wedge t_2 \in [0,1]$ to znači da nije došlo do preseka. U prvom slučaju zrak je ispaljen u suprotnom smeru od smera duži, a u drugom slučaju presek se nalazi van opsega duži.

Kada smo utvrdili da postoji presek onda možemo naći tačku preseka p koju lako možemo izračunati zamenjujući t_1 ili t_2 u neku od formula.

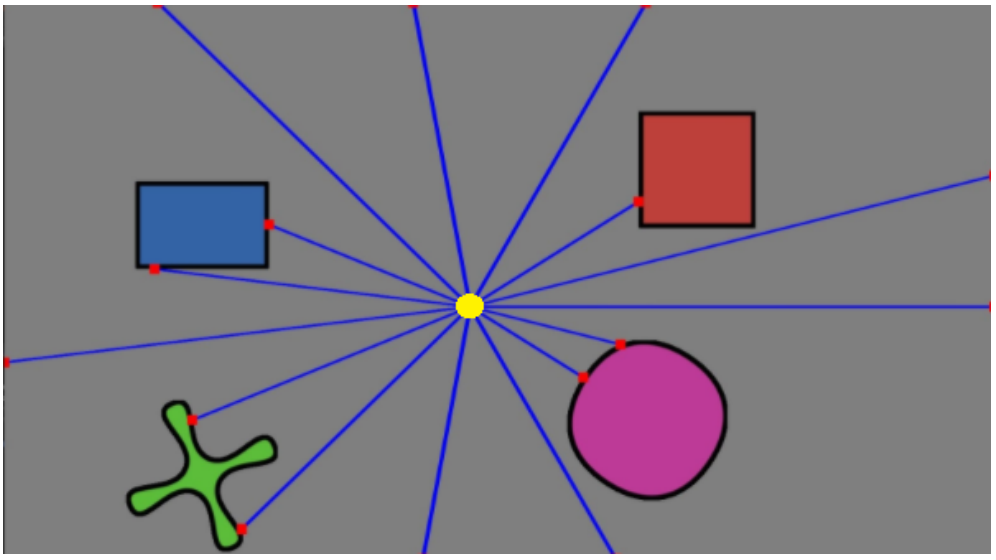
Raycasting u sceni

Da bismo implementirali raycasting koristićemo gore opisanu tehniku za ispaljivanje zrakova.

Prva informacija koja nam treba jeste kako dobiti duži u sceni nad kojima ćemo računati preseke. Ovo će biti lako u našem sistemu jer imamo occludere u sceni koji su opisani poligonima.
Ako imamo poligon sa n tačaka koje su poređane u smeru kazaljke na satu $A_0 A_1 \dots A_n$ tada definišemo duži tog poligona sa $A_i A_{i+1}$, $i = 0..n-1$.
Ako to uradimo za sve objekte dobićemo skup duži koje se nalaze na sceni.

Druga informacija koja nam je potrebna jeste skup zraka koje želimo da ispalimo. Za sad neka to bude N zraka, tako da je pozicija svakog zraka na poziciji emittera a smer zraka treba da bude jednako raspodeljen u krugu emittera.
$R_{id} = (sin(\alpha), cos(\alpha))$, $\alpha = i \frac{2\pi}{N}$

Kada ispalimo zrake očekujemo ovakav rezultat:



Gde je žuta tačka emitter, plave linije zraci, a crvene tačke preseci.

Light maska

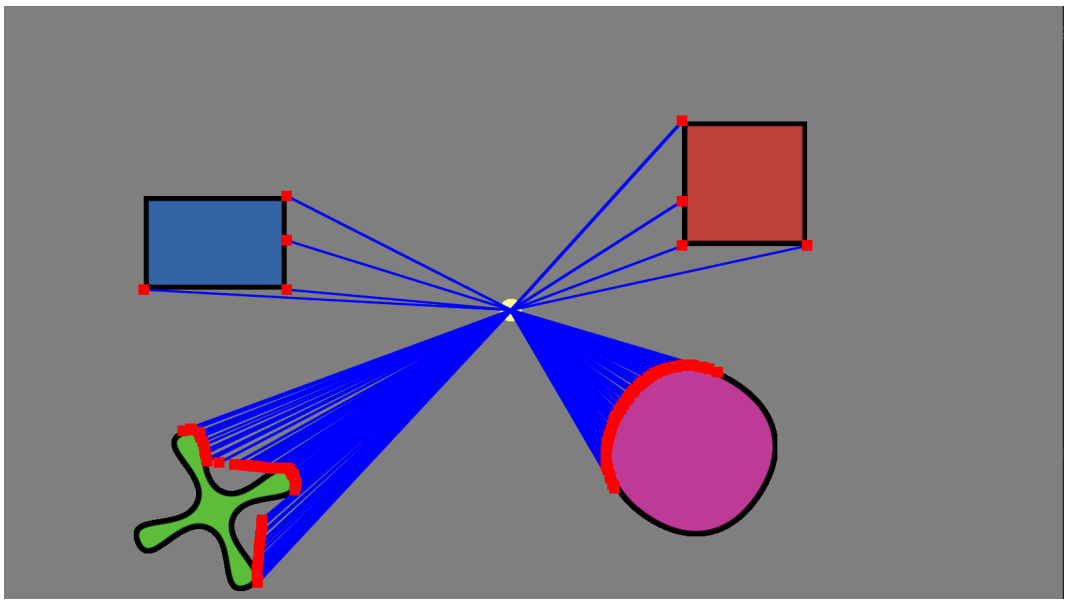
Trenutno imamo sve preseke sa objektima, ali da bismo to koristili dalje moraćemo te preseke nekako grafički da predstavimo, tj. tu površinu koju prave preseci.
Način na koji ćemo to da uradimo je pomoću light maske. To će predstavljati teksturu u kojoj će biti obojena samo površina koja je osvetljena. Nju ćemo kasnije moći da koristimo pri renderovanju scene da stvarno osvetlimo scenu tamo gde je potrebno i u meri koja je potrebna.

Pošto se crtanje u OpenGL svodi na crtanje trouglova, jedina stvar koju treba da uradimo je da nekako ove tačke preseka pretvorimo u trouglove koji će da pokrivaju površinu unutar tačaka. Koristeći činjenicu da smo zrake generisali inkrementalno po uglu možemo reći da će i preseci biti sortirani inkrementalno po uglu iz perspektive emittera. Pomoću toga konstrukciju trouglova možemo svesti na:
$T_i = (P_i, P_{i+1}, C)$, $Tn = (P_n, P_0, C)$, gde P_i predstavlja i-ti presek , a C poziciju emittera.
Trouglove ćemo bojiti u boji svetla emittera.

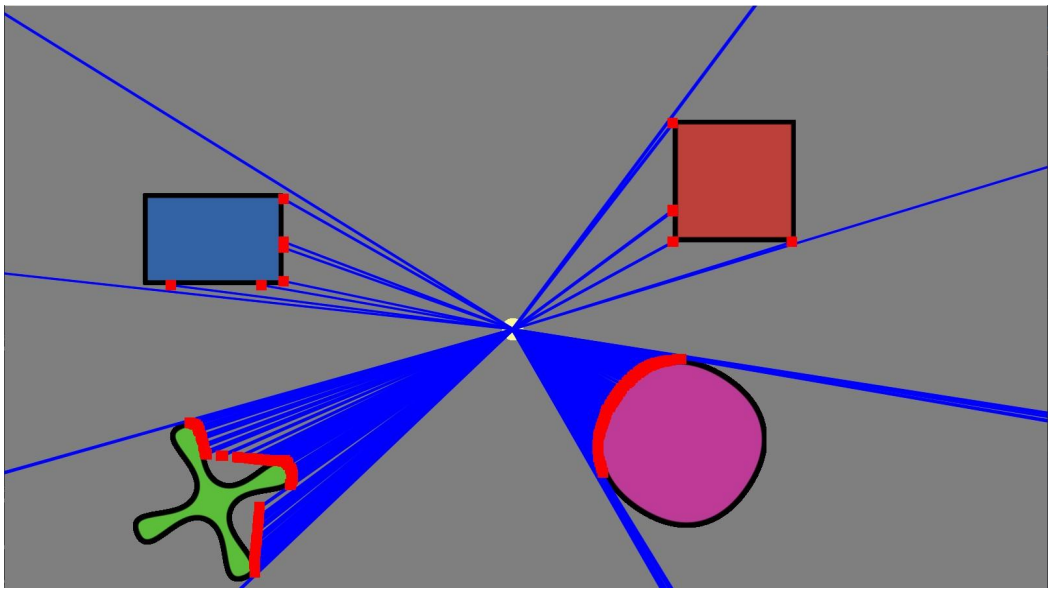
Dodatni zraci

Ispaljivanje zrakova u krug sa jednakim intervalima ne predstavlja dovoljno dobro jer može proizvesti određene grafičke artefakte. Na primer ako je objekat na sceni dovoljno mali da može da stane u prostor između dva zraka može se desiti da light maska preseca taj objekat na dva dela jer susedni pogođeni zrakovi se nalaze levo i desno od tog objekta. Osim toga u sličnoj situaciji light maska će potpuno osvetliti ili potpuno osenčiti taj objekat.
Jedan od načina za rešavanje ovog problema jeste da samo povećamo broj zraka oko emittera, ali i sa velikim brojem zrakova samo smanjujemo performanse a slični artefakti će se i dalje dešavati kod udaljenijih objekata na sceni.
Drugo rešenje jeste da osim zrakova koji idu oko emittera ubacimo i zrakove koji idu direktno na svaku tačku occuslion mesheva.
Ovde nailazimo na dodatni problem jer ćemo tako narušiti sortiranost zrakova po uglu ali to se lako rešava tako što ćemo na kraju ceo niz zrakova sortirati po uglu.
Ugao koji pravi zrak (R_o, R_d) se može izračunati na sledeći način:
$\alpha = atan2(Y_{Rd}, X_{Rd})$

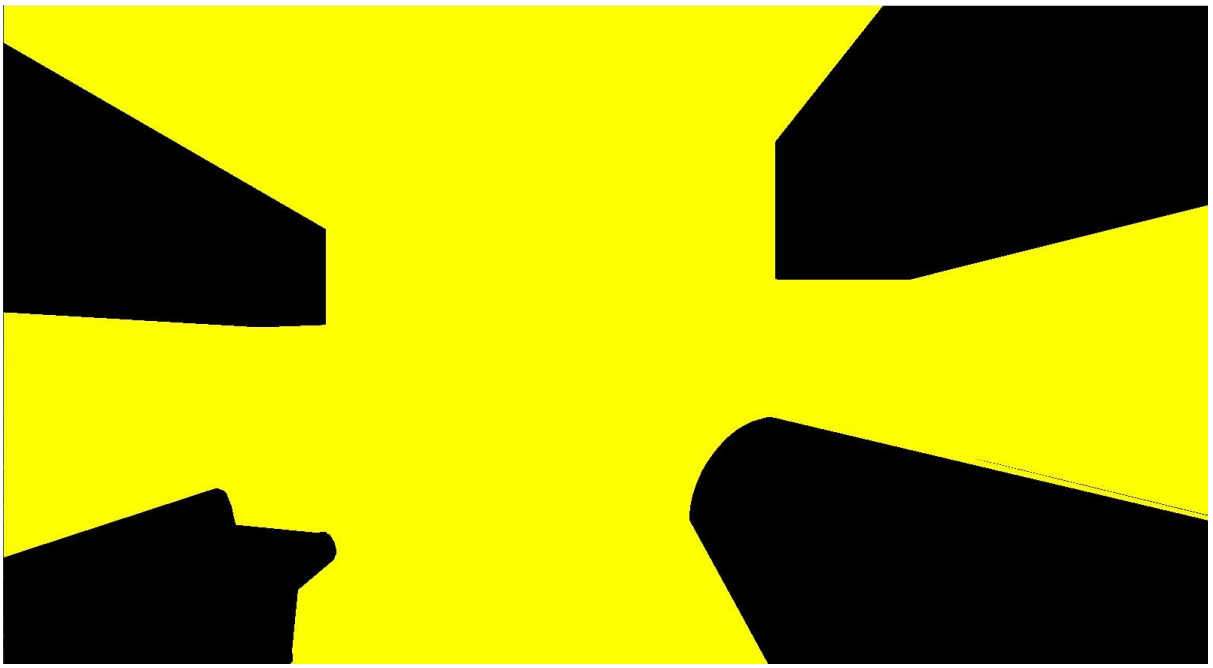
Kada nacrtamo te preseke dobijamo situaciju koja je ilustrovana na slici:



Kada bismo dodali ovakve zrake i dalje bismo imali slične artefakte. Treba zaključiti da nije ispravno gađati zrake direktno u tačke mesheva jer će im onda presek uvek zarvšavati u toj tački (ponekad pored zbog float preciznosti). Ispravan način je da za svaku tačku A dodamo dva zraka $(C, A + \varepsilon)$, $(C, A - \varepsilon)$ gde je ε neki vektor sa dovoljno malim vrednostima. Ovako dobijamo tačno preseke sa jedne i sa druge strane tačke u meshu što je dovoljno da bismo mogli potpuno da simuliramo okluziju svetlosti. Sada dobijamo preseke koji će rešiti pojavljivanje artefakta koje smo imali:



Kada iskombinujemo i zrake koji idu u krug dobijamo sledeću light masku:

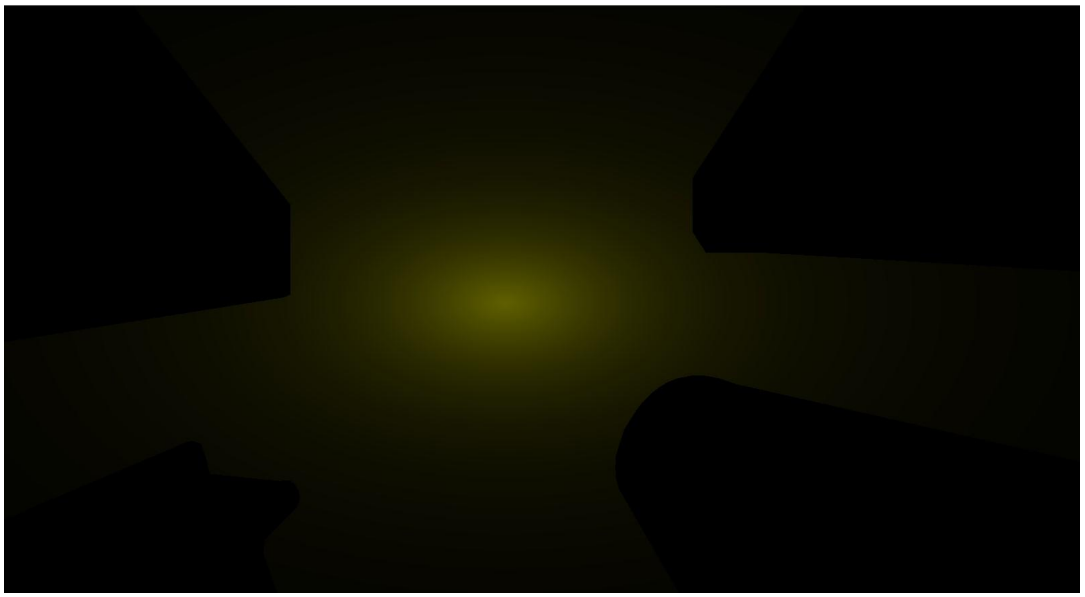


Atenuacija

Što je izvor svetlosti bliži objektu to je osvetljenje jače. Kao i u realnom svetu odnos intenziteta osvetljenja i rastojanja nije linearan, kako bismo ovo simulirali koristićemo formulu za slabljenje svetla (atenuacija) koju definišemo sledećin izrazom:

$attenuate(d) = \frac{1}{\alpha+\beta d+\gamma d^2}$, gde je d udaljenost od izvora svetlosti, a α, β, γ su parametri za osvetljenost. U implementaciji koju u ovom radu prikazujemo njihova vrednost je:
$\alpha = 0.4$, $\beta = 1$, $\gamma = 5$

Ako uključimo i atenuaciju u crtanje light maske dobijamo sledeću masku:



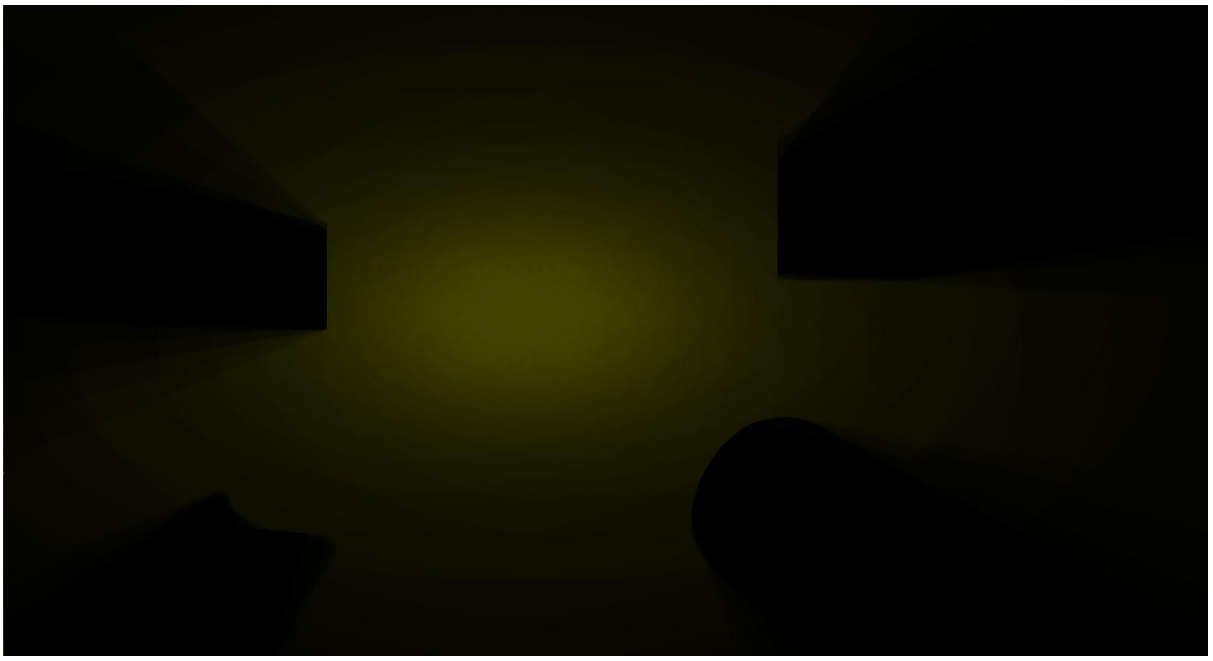
Area light

Dobili smo light masku za određenu poziciju emittera na sceni, ali na ovakav način mi emittera predstavljamo kao izvor svetlosti iz jedne tačke. Ovo za posledicu ima to da nama light maska ima jasnu granicu između osvetljene površine i neosvetljene. U realnom svetu nije to tako jednostavno. Ako uzmemo u obzir da emitter nije jedna tačka nego ima svoju površinu onda bi trebali da dobijemo masku gde su neki delovi samo parcijalno osvetljeni.
Idealan način kako bismo ovo mogli da izvedemo jeste da uzmemo u obzir beskonačno mnogo tačaka koji se nalaze na ivicama emittera i onda njihove maske iskombinujemo.

Kako ovo u realnosti nije moguće dovoljno dobra aproksimacija jeste da uzmemo konačno mnogo tačaka na ivicama koje ćemo uniformno da raspodelimo po površini i onda iskombinujemo njihove maske.
Implementacija koju u ovom radu prikazujemo ima još jedno ograničenje a to je da emitter može imati samo oblik kruga, tako da se ovaj problem svodi na to da uniformno raspodelimo tačke po kružnici emittera.
Kako u sceni imamo određen prečnik emittera, taj podatak ćemo uzeti kao prečnik kružnice i onda ćemo na sličan način kako smo raspodelili zrake kod emittera da dobijemo i izvore svetlosti emittera. Konkretno:
$C_i = C + (sin(\alpha), cos(\alpha)) * r$, $\alpha = i * \frac{2\pi}{L}$, $i = 0..L-1$, gde je L broj tačaka sa kojih ćemo da računamo light maske.

Način na koji ćemo da kombinujemo maske jeste tako što ćemo ih sve dodavati na jednu istu teksturu gde će svaka osvetljena površina imati prozirnost $\frac{1}{L}$, tako da samo oni tekseli[1] koji budu bili osvetljeni od svake maske će biti potpuno vidljivi.

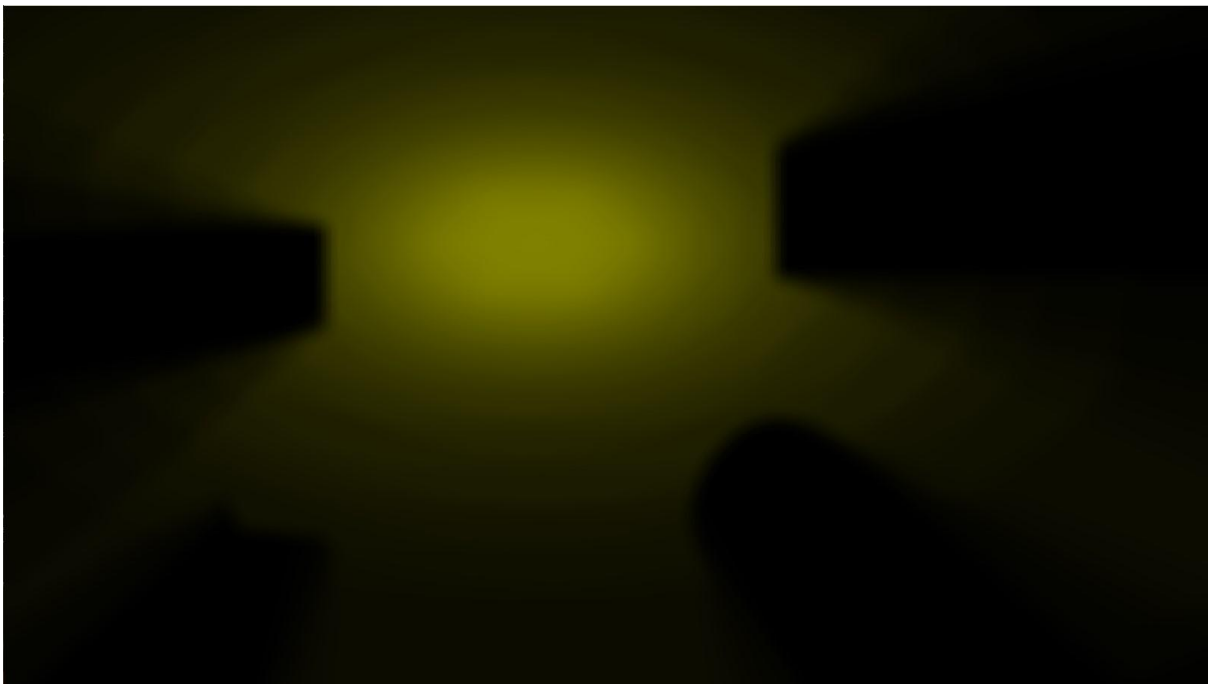
Rezultat koji dobijamo je:



[1] Teksel -piksel iz teksture

Blur

Ostao nam je još jedan korak da dobijemo dovoljno dobru light masku. Kao što vidimo na slici iznad, sada imamo drugačije artefakte koji odstupaju od realnosti a to su jasno izražene granice na mestima gde se jedna maska završava. Razlog toga ograničen broj izvora svetlosti na površini emittera. Ovi artefakti će biti veoma primetni na finalnoj slici.
Da bismo ovo izbegli opet koristimo malu aproksimaciju, u ovom slučaju pretpostavićemo da će boje između dve zone imati linearan prelaz u realnom svetu, što možemo da aproksimiramo upotrebom blur filtera. Tako da krajnji korak pri kreiranju light maske jeste primenjivanje blur filtera na celu light masku sa određenim parametrima tako da najbolje prikazuju realnost u 2D prostoru.



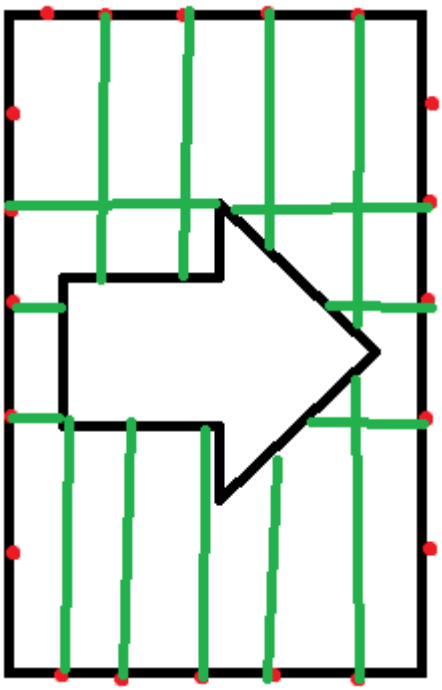
Generisanje occlusion poligona

Još jedan problem koji nam je ostao jeste kako da generišemo poligon za occludere tako da odgovaraju stvarnoj slici na sceni. Način koji je korišćen u implementaciji jeste određivanje tačaka u odnosu na alpha vrednost teksture occludera.

Stvar vredna pomena jeste da u ovom koraku ne moramo previše brinuti o performansama jer ovaj proces će se izvršavati samo jednom pri učitavanju entiteta za razliku od drugih procesa koji se dešavaju pri izračunavanju svakog frejma.

Takođe pominjali smo i meshLod parametar, koji označava detaljnost poligona koji ćemo generisati tj. broj tačaka. Konkretno $M = 20 * 2^{meshLod}$ gde je M maksimalan broj tačaka poligona. Prema ovoj formuli možemo videti da je minimalan broj tačaka 20 kada je meshLod nula. MeshLod se nalazi u eksponentu kako bismo dobili eksponencijalno više tačaka svakim sledećim nivoom detaljnosti poligona.

Prvi korak pri generisanju jeste pronalaženje tačaka koji leže na ivicama teksture tj. obliku koji bi se dobio prilikom odbacivanja skoro providnih teksela. Ovo ćemo uraditi tako što ćemo puštati zrake koji su uniformno rasporedjeni sa ivica teksture i videti gde je presek sa delom teksture koji nije providan. Za razliku od računanja light maske gde smo imali jasno definisan prostor, ovde ćemo morati da simuliramo korak po korak od početka zraka dok zrak ne dođe do teksela koji nije providan. Zrake koji ne pogode ništa ćemo samo odbaciti.



Drugi korak jeste odbacivanje nepotrebnih tačaka i sortiranje tačaka po uglu kako bi se kasnije taj poligon mogao pretvoriti u skup duži.

Scene culling

Pod scene culling podrazumevamo izbacivanje svih objekata koji ne mogu da utiču na trenutnu scenu iz daljeg procesa kako bismo optimizovali dalji proces renderovanja. Najčešći razlog izbacivanja objekata jeste to što se oni nalaze van ekrana koji se crta.

Scene culler kao input prima celu scenu i kao output daje CulledScene koja predstavlja podskup scene koji utiče na neki način na trenutnu scenu koja se renderuje.

Proveru da li se objekat nalazi unutar ekrana postižemo pomoću najosnovnije AABB (Axis Aligned Bounding Box) detekcije. Definišemo AABB ekrana tako što uzmemo koordinate ivica ekrana, i definišemo AABB objekta na osnovu njegove transformacije. Ako se objekat nalazi van AABB ekrana tada se taj objekat izbacuje iz daljeg procesa.

Kod occludera i emittera je malo kompikovanija situacija i postoje brojne tehnike posvećene rešavanju tih problema. Komplikacije sa occluderima i emitterima se javljaju iz razloga što oni mogu da utiču na scenu čak i kada nisu u opsegu ekrana. U implementaciji koju u ovom radu prikazujemo postoji samo naivna implementacija ovoga gde je glavna ideja povećavanje AABB ekrana za neki koeficijent što znači da će se occluderi i emitteri se izbacivati tek kad su budu malo dalje od ivice ekrana.

Albedo renderovanje

Rezultat albedo renderovanja jeste albedo tekstura koja predstavlja scenu bez osvetljenja. To podrazumeva crtanje samih tekstura objekata na sceni, crtanje pozadine kao i primenjivanje normal mapa za objekte koji ih poseduju.

Teksture

Crtanje tekstura je sasvim primitivno, svaki objekat je definisan kao pravugaonik koji ima UV koordinate. Pomoću tih UV koordinata dohvatamo teksturu i to crtamo, naravno nakon što primenimo ambientalno osvetljenje i normal mapu. Takodje se primenjuje i fake osvetljenje koje je bazirano samo na poziciji emittera koje može doprineti efektu u 3D koji se naziva subsurface scattering.

Prilikom crtanja pozadine, crtamo samo piksele koji se nalaze na ekranu. Prostorno pozadina je beskonačna na sceni i nju možemo da crtamo u jednom draw call-u. Moramo samo da podesimo da se iscrtavanje teksture pozadine ponavlja pri izlasku iz opsega UV koordinata. Ovo se postiže jednostavnom konfiguracijom samplera pomoću OpenGL biblioteke. Tekstura pozadine ima odredjeni offset u odnosu na poziciju kamere što daje utisak beskonačne pozadine.

Crtanje albedo tekstura podržava i alpha discarding što znači da ako je transparency neke teksture dovoljno mali taj piksel će biti odbačen.

Subsurface scattering

Subsurface scattering*[2]* predstavlja efekat koji se dobija kada svetlo prilikom refrakcije izlazi iz neke površine. Ovo možemo da aproksimiramo dodavanjem fake osvetljenja direktno na teksture. Fake osvetljenje ćemo dobiti tako što ćemo koristiti atenuaciju kao što smo je koristili pri generisanju light maske.

Ako je subsurface scattering omogućen onda će se vrednost teksture pomnožiti sa $clamp(\Sigma attenuate(d_i), 0, 1)$ gde je d_i distanca piksela od i-tog izvora svetlosti.

Normal mape

Normal mape su teksture koje se obično koriste u 3D prostoru kako bi se dodali detalji površine. U 2D prostoru nemaju neku generalnu primenu, ali se mogu koristititi na različite načine.
U slučaju implementacije opisane u ovom radu normal mape se koriste da bi pojačale doživljaj realističnog osvetljenja. Pomoću normal mape i pozicije emittera možemo utvrditi koji delovi same teksture trebaju da budu osenčeni, a koji osvetljeni. Ova tehnika nema smisla u fizičkom svetu ali daje dovoljno realistične rezultate. Ona predstavlja malo izmenjenu verziju Phong modela osvetljenja za računanje difuznog svetla. *[2]*
Prva stvar koja nam je potrebna u jeste normala na površinu koju dobijamo iz normal mape. Vrednost koju dobijamo iz normal mape se nalazi u opsegu $[0, 1]$ jer se radi o RGB formatu. Nama bi više odgovarao opseg [-1,1] jer u tom opsegu možemo lakse izvoditi operacije osvetljavanja i senčenja. Negativne vrednosti će oduzimati vrednosti iz albedo teksture(senčenje) dok će pozitivne dodavati(osvetljavanje). Veoma lako možemo vrednost prevesti u novi opseg na sledeći način:
$normal = 2 * old\_normal - 1$.

Sada kada imamo normalu na površinu možemo primeniti formulu do koje sam došao eksperimentalno koja je dala dovoljno realan rezultat:
$pixel\_value \mathrel{+}= attenuate(d_i) * dot(dir_i, normal)$, gde je d_i rastojanje piksela od i -tog izvora svetla a dir_i normalizovan vektor čiji pravac ide od pozicije piksela ka poziciji i-tog izvora svetla.
Ovo primenjujemo za svaki izvor svetlosti koji imamo na sceni. Na kraju ovu vrednost normalizujemo izmedju 0 i 1 jer radimo sa RGB formatom.

Finalno renderovanje

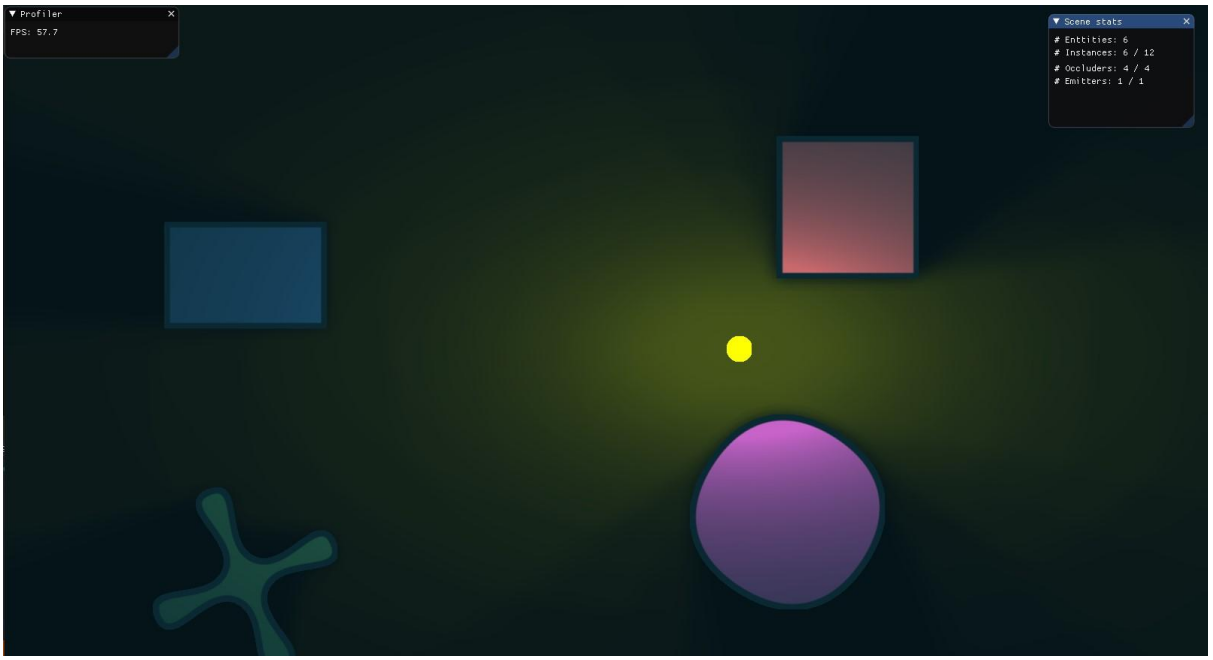
Sada kada imamo imamo light masku i albedo teksturu, trebamo da ih iskombinujemo da dobijemo finalnu scenu. Ovo je veoma jednostavan korak u kome

prvo kombinujemo ambijentalno osvetljenje sa light maskom i to na kraju primenjujemo na albedo teksturu i tako dobijamo finalni rezultat. Ovaj proces se može opisati sledećom formulom:

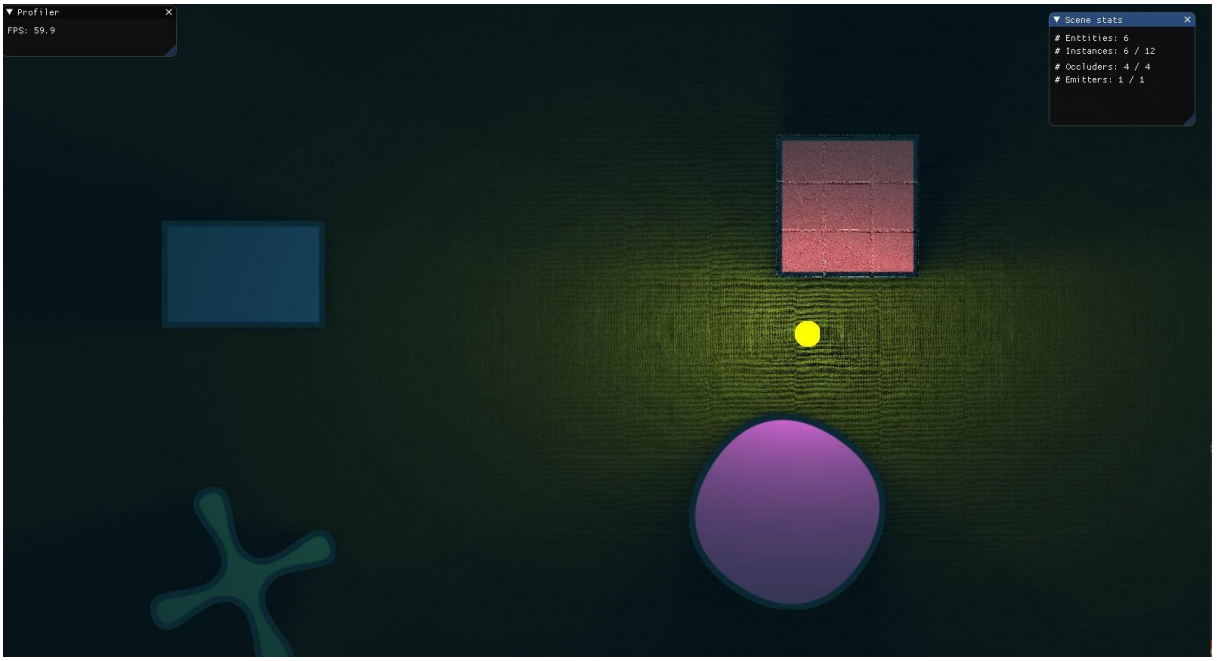
$$light = clamp(light\_mask * ambient\_light, 0.0, 1.0)$$
$$final\_color = light * albedo\_texture$$

Nakon ovoga još se renderuju UI elementi u UIEngine i ta slika se prezentuje.

Kako bih ilustrovao uticaj normal mape na celu scenu prvo ću pokazati kako izgleda finalna scena bez normal mapi.



Sada ću dodati normal mape na pozadinu i na crveni kvadrat:


▼ Profiler
FPS: 59.9
▼ Scene stats
Entities: 6
Instances: 6 / 12
Occluders: 4 / 4
Emitters: 1 / 1

Zaključak

“Simulacija svetla u 2D prostoru” istražuje jednu od nedovoljno istraženih tema u kompjuterskoj grafici a to je realistično renderovanje elemenata u 2D prostoru. Ovaj rad je samo zagrebao po površini stvarne mogućnosti ove teme. Osim optimizacije koja je itekako potrebna ovaj rad je moguće dopuniti sa simulacijom refrakcije svetla na poluprovidnim površinama, refleksijama, simulacijom vode, simulacijom magle itd. Mogućnosti su beskonačne.

Literatura

[1] https://learnopengl.com/Advanced-Lighting/Normal-Mapping
[2] Real-Time Rendering, Fourth Edition, by Tomas Akenine-Möller, Eric Haines, Naty Hoffman, Angelo Pesce, Michał Iwanicki, and Sébastien Hillaire
[3] Foundations of Game Engine Development, Volume 1: Mathematics by Eric Lengyel
[4] https://www.opengl.org/
[5] https://premake.github.io/
[6] https://godotengine.org/
[7] https://unity.com/
[8] Foundations of Game Engine Development, Volume 2: Rendering by Eric Lengyel
[9] https://www.khronos.org/opengl/wiki/Shader_Storage_Buffer_Object

Biografija

Marko Srećković je rođen u Požarevcu, Srbija datuma 17.09.1997. Završio je osnovnu školu “Kralj Aleksandar” u Požarevcu a nakon toga Požarevačku gimnaziju. Nakon srednje škole upisuje osnovne akademske studije na Računarskom Fakultetu na smeru Računarske nauke. Uporedo sa fakultetom u 2019. godini počinje da radi u kompaniji za pravljenje video igara “Ubisoft” na poziciji “Junior programer”, a godinu ipo dana nakon toga na poziciji “Render programer”.