Računarska grafika

Interaktivno 3D renderovanje:

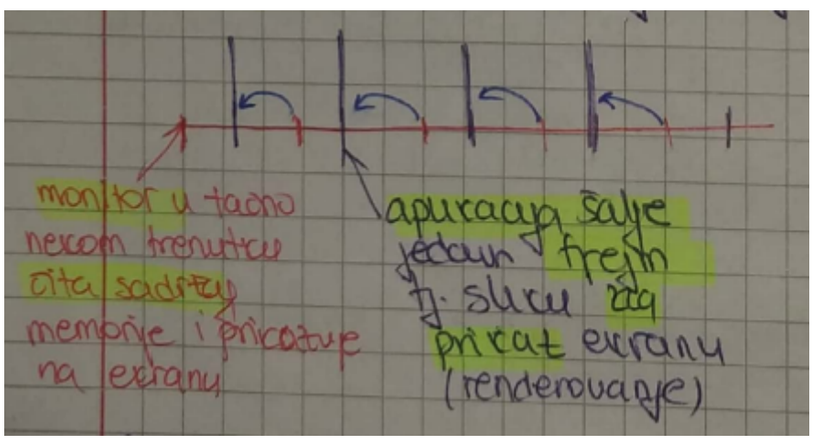
- interaktivno - u zavisnosti od korisničkog ulaza sa testature ili miša i prethodnog stanja sveta, menja se slika koju prikazujemo na ekranu.
- 3D - svi objekti su prikazani u 3D svetu.
- renderovanje - proces koji objekte, matematički definisane u 3D svetu, prikazuje u 2D svetu tako što transformiše koordinate i informacije datog objekta.

Primer interaktivne 3D grafike su **igrice** i **simulacije**, dok film nije interaktivna 3D grafika. **Petlja renderovanja**:

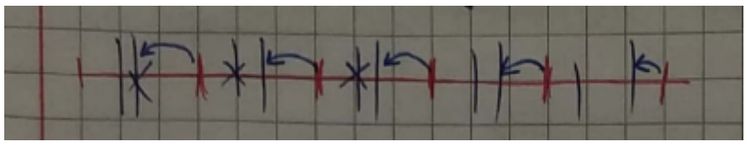
```
init();
while(do_run) {
        poll_events(); // registracija događaja, nije blokirajuća naredba, na CPU
        update(); // ažuriranje stanja, na CPU
        render(); // iscrtavanje, na GPU
}
deinit();
```

Sve se dešava nekoliko desetina ili stotina puta u sekundi. Slike se toliko brzo smenjuju da nam deluje kao neprekidan tok, a ako ove promene nisu dovoljno brze to primećujemo kao prekide. **Frejm** je kompletno renderovana slika u jednom trenutku. Monitori se osvežavaju frekvencijom od 60Hz - ekran 60 puta u sekundi promeni celokupno sliku na ekranu. **Osvežavanje ekrana** predstavlja čitanje sadržaja frejm bafera, koji popunjava petlja renderovanja, i osvežavanje sadržaja, tj. piksela na ekranu tako da odgovaraju bojama zabeležnim u frejm baferu. Noviji ekrani se osvežavaju i frekvencijama od 120Hz, 144Hz i 240Hz. **FPS** (**frame per second**) označava koliko puta u sekundi može da se renderuje celokupna slika, tj. koliko puta u sekundi program prođe kroz celu petlju renderovanja. Ako je t vreme koje je potrebno da prođemo kroz celu petlju renderovanja u milisekundama, onda važi FPS = $\frac{1000}{t}$. Na primer, ako igrica radi na 10FPS, a monitor se osvežava frekvencijom od 60Hz, onda se $\frac{60Hz}{10FPS}$ = 6 puta ponovi svaki frejm. **Ekran** je zapravo matrica ćelija, odnosno **piksela**. Ukupan broj piksela na ekranu zove se **rezolucija ekrana**. Standardna rezolucija je 1920x1080, što znači 1920 kolona i 1080 redova piksela. Svaki od njih može da emituje neki intenzitet crvene, zelene i plave boje predstavljen brojem od 0 do 1. Nula ako uopšte ne emituje intenzitet te boje, a jedan ako emituje tu boju koliko god je moguće. Sve druge boje dobijaju se kombinacijom ove tri boje.

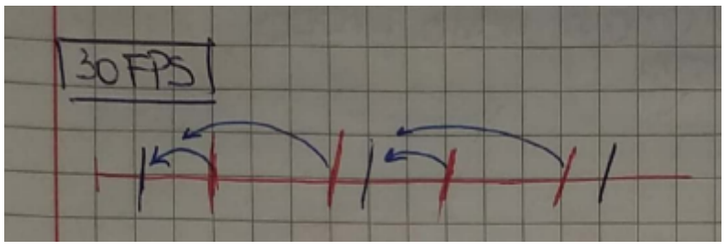
Video memorija (**VM**) je bafer u kome svaka ćelija odgovara jednom pikselu na ekranu. VM se nalazi na nekom ?. **Video kontroler** (**VC**) u nekom vremenskom intervalu dt čita sadržaj video memorije gde je na svakoj ćeliji ove memorije intenzitet RGB koji odgovarajući piksel emituje i na osnovu te 3 vrednosti postavlja intenzitet RGB svakog piksela na ekranu. Ako se monitor osvežava na 60Hz, to znači da 60 puta u sekundi, na nekom pravilnom intervalu približno jednakom dt, video kontroler čita sadržaj video memorije i postavlja vrednosti piksela na ekranu tako da odgovraju vrednostima u video memoriji.



Da ne bi došlo do situacije da VC čita iz VM, a da naša aplikacija upisuje nešto u VM, odnosno da prikaz na ekranu bude pola nove, a pola stare slike, često se koriste dva bafera tako što se u jedan piše, a iz drugog se čita, pa se zatim ti baferi zamene. Ako aplikacija radi na 120FPS, a monitor na 60Hz uvek se renderuje samo poslednje:

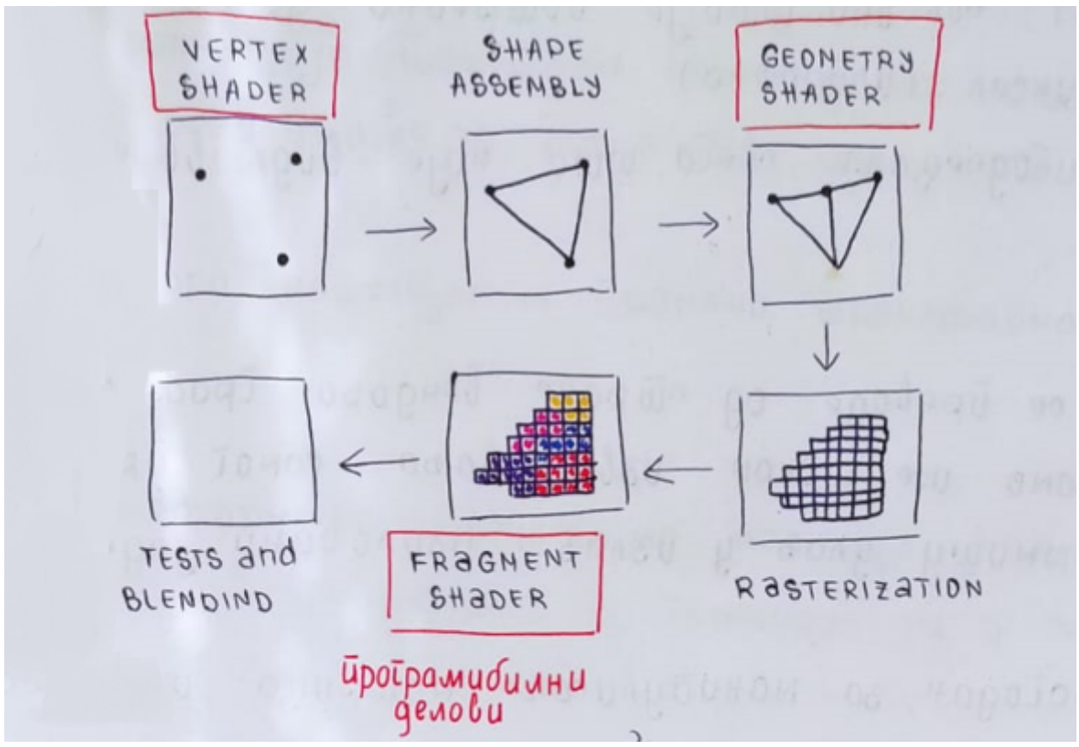


Ako aplikacija radi na 30FPS, više puta se renderuje ista slika, tj. nemamo "najskoriju" sliku:



ms označava vremenski interval između dva uzastopna frejma, u milisekundama. Obrnuto je proporcijalan FPS-u, odnosno manji ms znači brže reakcije na komande korisnika, a veći FPS znači glatkiji prikaz. Kada govorimo o ubrzanju aplikacije uvek govorimo u ms. Ako kažemo povećali smo FPS za 50 razlika je jako mala: $\frac{1000}{500}$ = 2 i $\frac{1000}{550}$ = 1,81. Međutim, poboljšanje sa 10FPS na 60FPS jeste veliko jer predstavlja poboljšanje za 84ms: $\frac{1000}{10}$ = 100 i $\frac{1000}{60}$ = 16.

Grafička kartica (**Graphic Processing Unit** - **GPU**) je specijalizovani hardver za obradu i prikaz slika. To je čip skoro podjednake kompleksnosti kao CPU, s tim što je procesor čip opšte namene i programabilan, dok je GPU specijalizovan za renderovanje grafike i postizanje traženog frejm rejta. **Shader** je program koji se izvršava na grafičkoj kartici. Za pisanje koda koji se izvršava na GPU koristi se **GLSL jezik** koji liči na C i pogodan je za manipulisanje vektora i matrica. Osnovni tipovi su bool, uint, int, float i double. Vektor dužine n tipa float zadaje se sa vecn, a za ostale tipove sa bvecn, ivenc, uvecn, dvecn. Šejderi se prevode od strane vendora grafičke kartice, a mi ih kompajliramo tokom izvršavanja samog programa. Oni su izolovani jedni od drugih. **GPU protočna obrada**:



- **Vertex shader** kao ulaz dobija niz temena i ima zadatak da ta temena transformiše iz 3D koordinata jednog tipa u 3D koordinate drugog tipa. Uz koordinate prosleđuju se i kolekcija podataka koja definiše jedno teme kao što su boja, normale, mapa tekstura itd - **vertex atributi**. U main funkciji vrednost upisuje u promenljivu **gl_Position**.
- **Shape assembly** na osnovu koordinata gradi oblik koji crtamo.
- **Geometry shader** kreira nove oblike od postojećih dodavajući temena.
- **Rasterizacija** je proces u kome se koordinate transformišu u piksele na ekranu.
- **Fragment shader** određuje boju svakog piksela na osnovu prosleđenih informacija. **Fragment** je sve ono što je potrebno da bi se ispravno obojio jedan piksel. Ovde se vrši i **fragment interpolacija** - boja fragmenata unutar trougl se računa interpolacijom boja vertexa. U main funkciji vrednost upisuje u promenljivu **FragColor**.
- **Tests and blending** je faza u kojoj se proverava providnost objekta, da li se on poklapa sa nekim drugim i slično.

Shader program predstavlja jedan program koji linkuje Vertex Shader i Fragment Shader. Struktura šejder programa:

- **broj verzije -** `#version 330 core`
- **deklaracija ulaznih promenljivih -** `in type name`, ulazni podaci iz prethodnog šejdera u protočnoj obradi
- **deklaracija izlaznih promenljivih -** `out type name`, izlazi podaci iz trenutnog šejdera za naredni, izlaz iz Fragment Shadera mora biti boja reprezentovana tipom vec4, inače će boja biti nedefinisana
- **deklaracija uniform promenljivih -** `uniform type name`, globalne promenljive za jedan šejder program
- **main funkcija -** postavljanje vrednosti izlaznih promenljivih

glVertexAttribPointer je funkcija kojom se opisuje veličina vertexa i veličina i pozicija jednog atributa unutar vertexa. Tu postavku pokupi šejder koji obrađuje pojedinačne vertexe. Moguće je imati 16 vertexa od po 4 komponente. Na primer: `glVertexAttribPointer(0, 3, GL_FLOAT, GL_FALSE, 6 * sizeof(float), (void*)0)` označava redom da je redni broj atributa 0, da on ima 3 floata, da ne želimo da normalizujemo vrednosti, zadaje veličinu i poziciju atributa u vertexu.

Algoritmi za renderovanje:

1. **Slikarev algoritam** - sortira objekte po blizini i prvo crta najdalje, pa onda najbliže tako što ih prosto renderuje preko već nacrtanih.
2. **Z-buffer algoritam** - za svaki piksel se pored boje čuva i distanca od kamere, odnosno **z-dubina**. Ako je z-vrednost za neki piksel manja od trenutne onda piksel dobija novu vrednost. Objekti daleko od kamere imaju veliku z-vrednost, a oni blizu malu.

OpenGL radi kao automat stanja, odnosno sadrži promenljive čije stanje definiše trenutni kontest. Postoje funkcije koje menjaju stanje i funkcije koje koriste stanje. **OpenGL objekat** je skup stanja u OpenGL-u. Nema isto značenje kao C++ objekat ili Java objekat. Tokom izvršavanja može menjati tip. **GLFW** je biblioteka koja omogućava kreiranje prozora, konteksta, registrovanje korisničkog ulaza, događaja i slično. **GLAD** je biblioteka koja dinamički učitava adrese funkcija tokom pokretanja programa.

Koordinate teksture (**tekseli**) su pozicija na 2D slici koja će se mapirati na vrteks u kome su navedene. One su u intervalu [0, 1] po x i y osi. Mapiramo samo temena. **Mipmape** su kolekcija tekstura u kojoj je svaka sledeća duplo manja od prethodne, počevši od neke originalne rezolucije sve do 1x1.

Množenje vektora matricom odgovara primeni određene **transformacije** na taj vektor. Za 3D transformacije koristimo 4x4 matrice jer su svi vektori koje posmatramo u grafici u **homogenim koordinatama**. To je jedini način da predstavimo translaciju isto kao i ostale transformacije. Osnovne transformacije:

- **skaliranje**:

$$\begin{bmatrix} s_x & 0 & 0 & 1 \\ 0 & s_y & 0 & 1 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} s_x x \\ s_y y \\ s_z z \\ 1 \end{pmatrix}$$

  Skaliranje ne menja i vektore normala, pa iz tog razloga koristimo matricu normale. **Matrica normale** je transponovani inverz model matrice: $norm = mat3(transpose(inverse(model))) \cdot aNorm.$
- **translacija**:

$$\begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x + t_x \\ y + t_y \\ z + t_z \\ 1 \end{pmatrix}$$

- **rotacija**: OpenGL očekuje uglove u radijanima. Rotacija može da se gleda i preko **Ojlerovih uglova** ψ (**pitch**), θ (**yaw**) i ϕ (**roll**). **Gimbal lock** nastaje kada izgubimo jednu ili čak i dve ose prilikom rotacije. Rotacije oko koordinatni osa:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\phi & -\sin\phi & 0 \\ 0 & \sin\phi & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \; R_y = \begin{bmatrix} \cos\phi & 0 & \sin\phi & 0 \\ 0 & 1 & 0 & 0 \\ -\sin\phi & 0 & \cos\phi & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}, \; R_z = \begin{bmatrix} \cos\phi & -\sin\phi & 0 & 0 \\ \sin\phi & \cos\phi & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

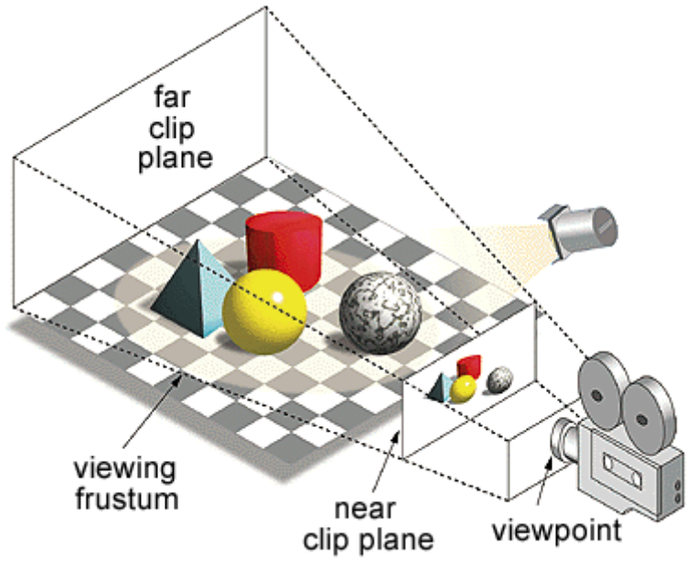
Sve transformacije se vrše tako što prvo definišemo jediničnu matricu i onda ih množimo redom u poretku: skaliranje $\rightarrow$ rotacija $\rightarrow$ translacija.

Svaka **scena** ima 3 glavna dela - **kameru**, **osvetljenje** i **objekte** na sceni. Kamera je definisana sa 4 vrednosti:

- **pozicija** u svetskim koordinatama
- **smer gledanja** (niz z-osu)
- **vektor u desno**

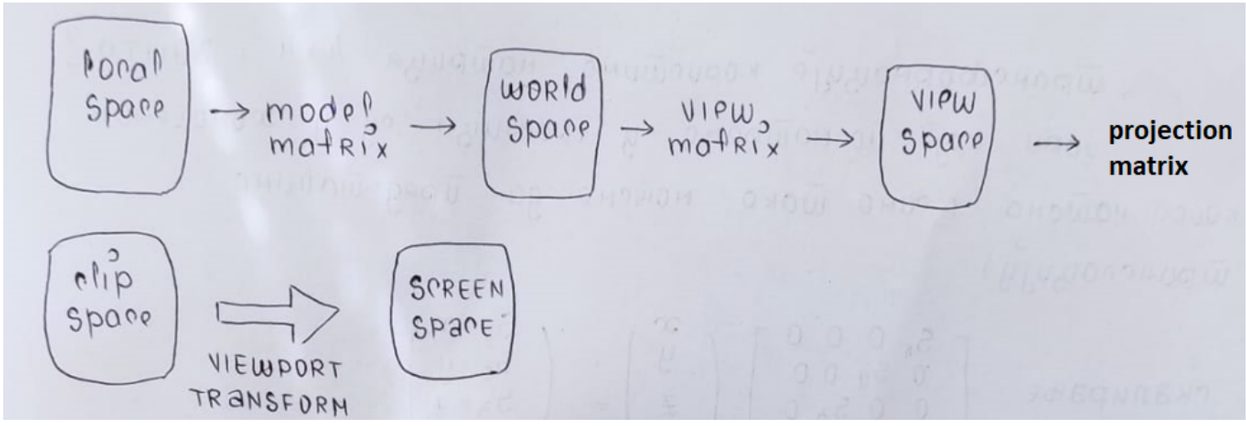
- **vektor na gore**

Uz veličinu prednje ravni odsecanja (**near clipping plane**), veličinu **zadnje ravni odsecanja** (**far clipping plane**) i **vidno polje** koje definiše ugao koji kamera zahvata formira se **zapremina pogleda** (**frustum**) u obliku zarubljene piramide. Objekti u unutrašnjosti se renderuju, a oni van zapremine pogleda se odbacuju, što se naziva **clipping**.



Da bismo preveli 3D koordinate objekata u 2D koordinate ekrana, koristimo **projekciju**. **Ortografska projekcija** podrazumeva projektovanje scene pod nekim uglom na near ravan kamere. **Perspektivna projekcija** podrazumeva da sve objekte projektujemo u tačku pozicije kamere. Koordinate mogu biti u:

- **local space** - koordinate objekta
- **world space** - pozicija u svetu
- **camera space** - pozicija u pogledu kamere
- **clip space** - **Normalized Device Coordinates** (**NDC**), koordinate u opsegu [-1.0, 1.0] po x, y i z osi
- **screen space** - koordinate ekrana



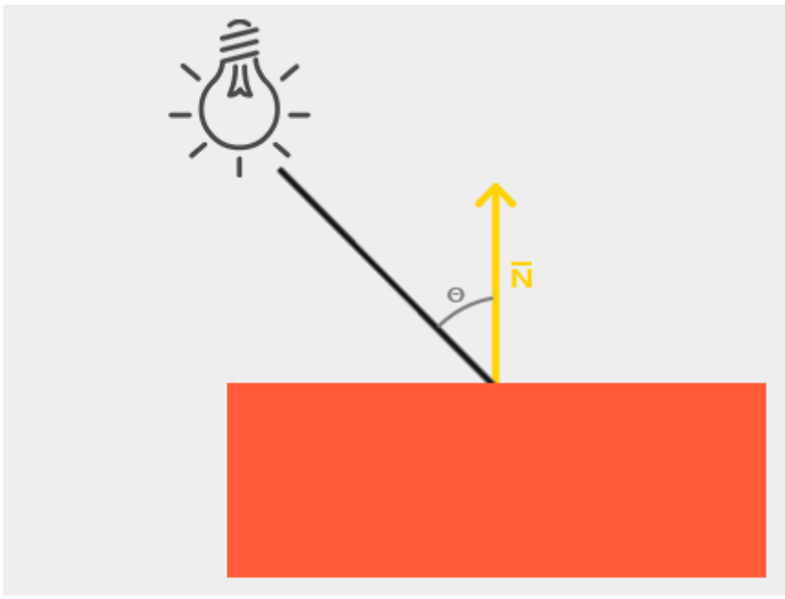
Pomeranje kamere po svetu je ekvivalentno pomeranju sveta oko kamere.

Rezultujuću boju objekta definišu boja izvora svetlosti, ali i sam objekat koji će tu boju reflektovati. **Phongov model osvetljenja**:

- **ambient**: svetlost koja na objekat pada iz svih pravaca i ne dolazi iz nekog izvora, već se odbija od scene u svim pravcima. Kao svetlo koje je beskonačno daleko u svim pravcima, ravnomernog je intenziteta.

$$ambient = ambientStrength \cdot lightColor$$

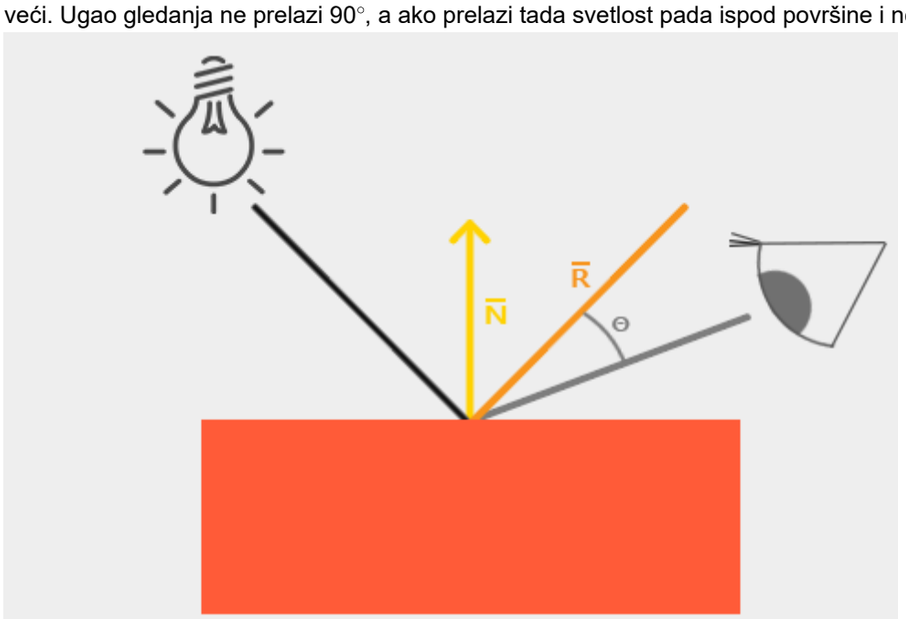
- **diffuse**: svetlost koja dolazi iz nekog izvora koji je na konačnoj razdaljini. Što je objekat "okrenutiji" izvoru svetlosti, to će on više biti osvetljen. Odnosno, što je ugao pod kojim svetlost pada na objekat veći, to je intenzitet manji. Ova komponenta najviše utiče na finalnu boju.



$$diffuse = diffuseStrength * diff * lightColor,\ diff = max(dot(norm, lightDir), 0),$$

$$norm = normalize(Normal),\ lightDir = normalize(lightPos - FragPos)$$

- **specular**: komponenta koja simulira deo objekta koji se presijava. Što je ugao gledanja θ manji to će odsjaj biti veći. Ugao gledanja ne prelazi 90°, a ako prelazi tada svetlost pada ispod površine i ne treba da je osvetljava.



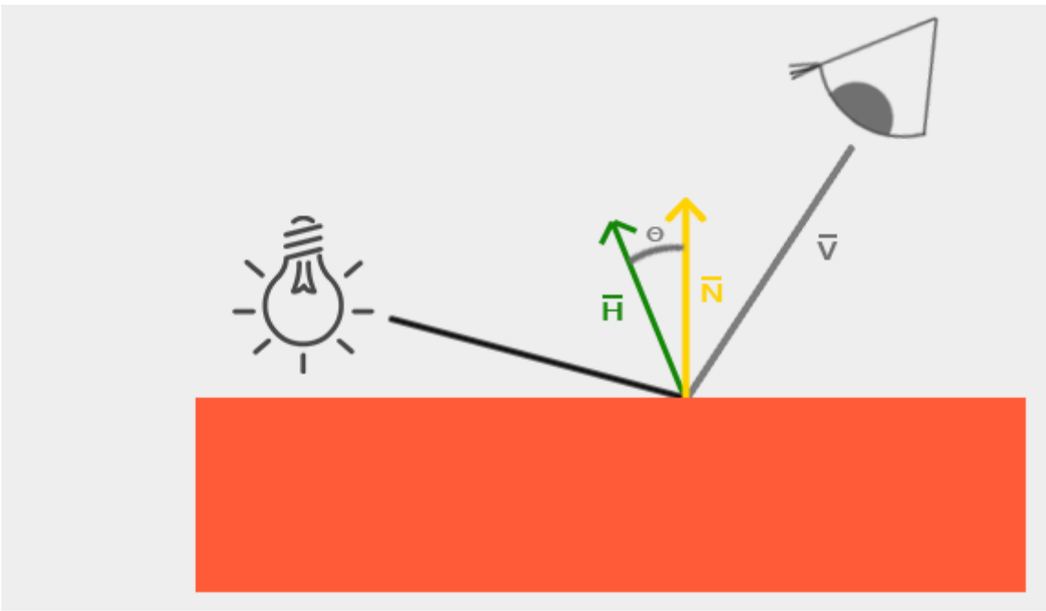
$$specular = specularStrength * spec * lightColor,\ spec = pow(max(dot(viewDir, reflectDir), 0), shiness),$$

$$viewDir = normalize(viewPos - FragPos),\ reflectDir = reflect(-lightDir, norm)$$

Na kraju, sve komponente se sabiraju i daju rezultujuću boju:

$$result = (ambient + diffuse + specular) * objectColor$$

Blinn-Phonogov model osvetljenja je varijanta Phongovog modela sa izmenjenom spekularnom komponentom. Koristi vektor koji polovi ugao između vektora pogleda kamere i vektora padanja svetlosti na fragment.



$$specular = specularStrength * spec * lightColor,\ spec = pow(max(dot(norm, halfwayDir), 0), shiness),$$

$$halfwayDir = normalize(lightDir + viewDir)$$

Materijal od kog je objekat napravljen određuje koliko objekat reflektuje svaku komponentu. **Difuzna mapa** je tekstura koja sadrži sve difuzne boje objekta. **Spekularna mapa** je tekstura koja sadrži informaciju koliko svaki deo objekta reflektuje svetlost.

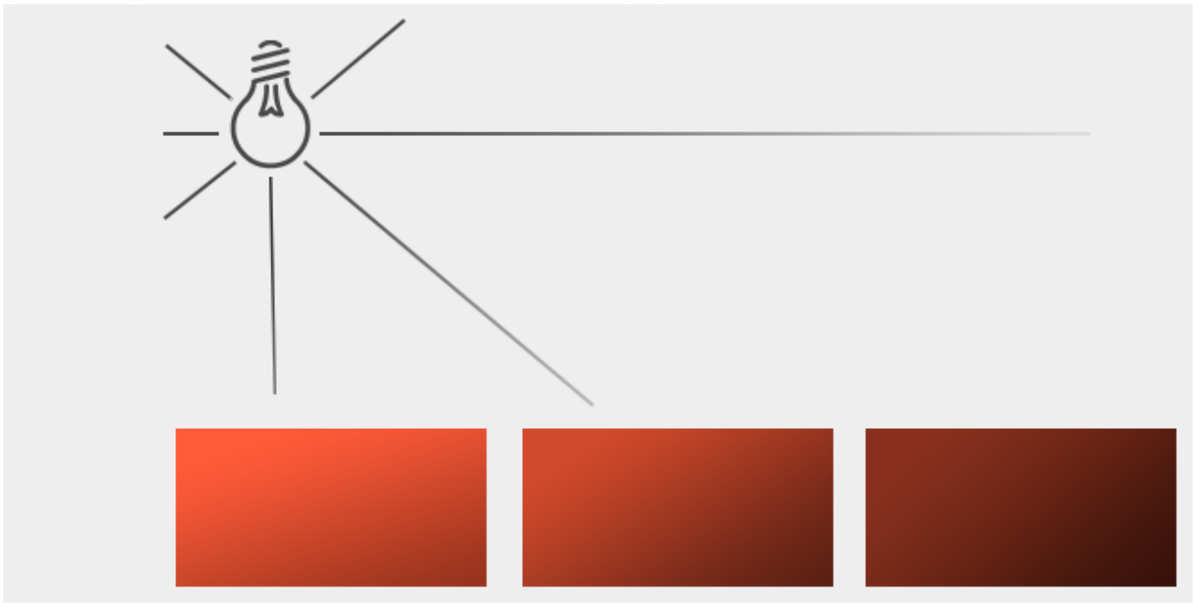
U zavisnosti od blizine izvora svetlosti i opsega kojim ona obasjava fragmente na sceni razlikujemo 3 tipa svetlosti:

- **directional** - beskonačno daleko svetlo. Na sve objekte na sceni zraci padaju paralelno u istom smeru. Koristimo vektor padanja zraka umesto pozicije svetlosti.

- **point light** - tačkasto svetlo koje se nalazi na sceni sa ostalim objektima. Što je objekat bliže izvoru svetlosti to će jače biti osvetljen. Koristi se **formula slabljenja svetlosti** (**attenuation**) u zavisnosti od rastojanja d:

$$F_{att}(d) = \frac{1}{c + l \cdot d + q \cdot d^2},$$

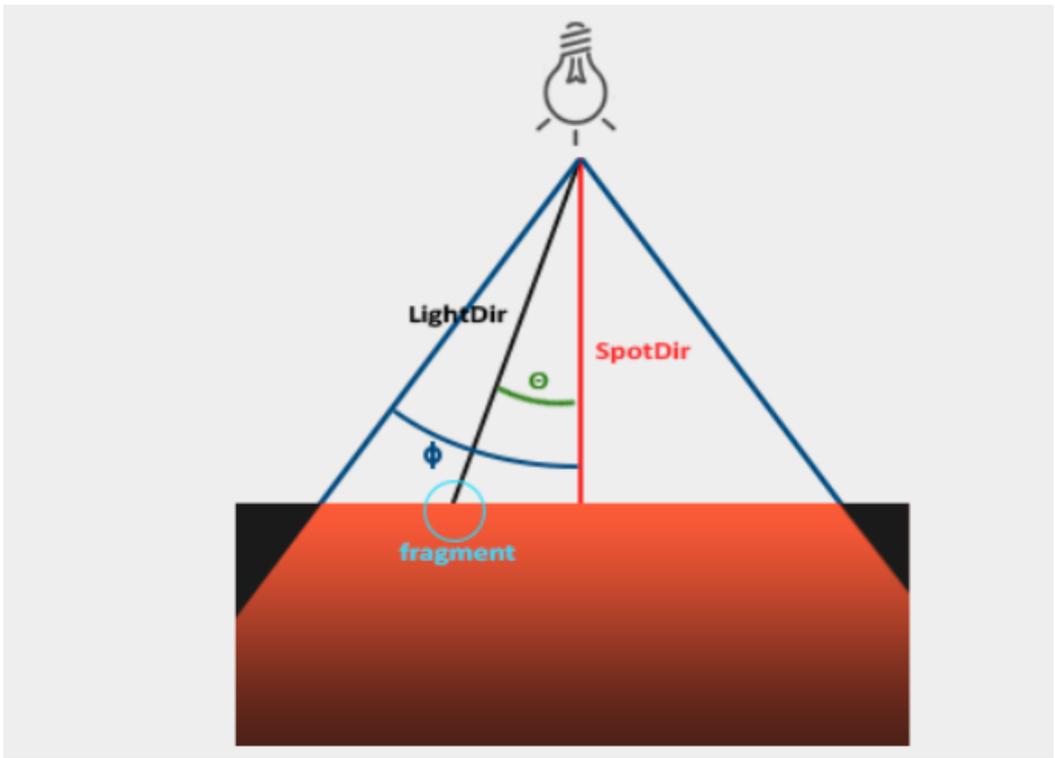
gde su c, l i q **konstantni**, **linearni** i **kvadratni faktor slabljenja svetlosti**.



- **spotlight** - svetlo koje simulira lampu, odnosno reflektorsku svetlost. Takođe se nalazi na sceni sa objektima. Samo fragmenti koji padnu pod opseg svetlosti budu osvetljeni. Određeno je pozicijom, smerom i uglom odsecanja. Da bi se postigao lepši prelaz sa osvetljenih na neosvetljene objekte može se uvesti i drugi ugao koji bi definisao "prelaznu" oblast gde su objekti osvetljeni, ali slabije.

$$lightDir = normalize(lightPos - FragPos),\ alpha = dot(lightDir, normalize(-SpotDir))$$

```
if(alpha <= theta)
        // objekat je osvetljen
else if(alpha <= phi)
        // objekat je osvetljen slabije
else
        // objekat nije osvetljen
```



Depth buffer (**z-buffer**) je memorija koja čuva vrednosti dubine svakog od fragmenata na sceni, tj. udaljenost od near ravni. Čuva vrednosti u opsegu [0.0, 1.0]. **Depth testing** vrši se nakon fragment shader-a u poslednjoj fazi - tests and blending. Radi se u screen koordinatama. U OpenGL-u koristi se komanda `glEnable(GL_DEPTH_TEST)` da se aktivira testiranje dubine. Fragmenti koji prođu test biće renderovani. Pre svakog renderovanja koristi se `glClear(... | GL_DEPTH_BUFFER_BIT)` da se očisti bafer dubine. **Z-vrednost** fragmenta može biti bilo koja vrednost između nule i udaljenosti near i far ravni odsecanja, a u z-baferu se čuvaju vrednosti od 0 do 1. Za transformaciju opsega koristi se linearna transformacija:

$$F_{depth} = \frac{z - near}{far - near}$$

Međutim, nelinearan test će imati veću preciznost za bliže objekte pa se koristi nelinearna transformacija:

$$F_{depth} = \frac{\frac{1}{z} - \frac{1}{near}}{\frac{1}{far} - \frac{1}{near}}$$

Z-koordinatu treba linearizovati jer ova transformacija nije linearna, odnosno treba se iz screen space-a vratiti u view space. **Z-fighting** nastaje kada su dva objekta toliko blizu da bafer dubine nema dovoljnu preciznost da odredi koji je ispred, pa deluje kao da se oni konstantno smenjuju. Moguća rešenja ovog problema su:

- nikad ne postavljati objekte previše blizu
- postaviti near ravan kamere što je dalje moguće
- koristiti precizniji bafer dubine

Transparentnost nam omogućava providnost objekta, odnosno za fragment koji crtamo uzimamo malo boju objekta, a malo boju pozadine. Koliko će objekat biti providan navodi se **alfa parametrom** - (R, G, B, alpha). Ako je alfa = 0, objekat je potpuno providan, a ako je alfa = 1 objekat nije providan. Postoje dve tehnike u zavisnost od toga da li želimo da fragmenti butu potpuno vidljivi/providni ili želimo da ih mešamo:

- **odbacivanje fragmenata** - rešenje za fragmente sa potpunom providnošću i fragmente bez providnosti. Fragmenti sa potpunom providnošću će prosto biti odbačeni.
- **blending** - rešenje za mešanje i napredne efekte. U OpenGL-u se korišćenje alfa parametra omogućava sa `glEnable(GL_BLEND)`. Koristi se formula:

  $$C_{res} = C_{src} \cdot F_{src} + C_{dest} \cdot F_{dest},$$

  gde je C_{src} boja izvora, C_{dest} boja koja se već nalazi u baferu, a F_{src} i F_{dest} faktori mešanja boje. Podrazumevano je F_{src} jednako alfa vrednosti izvora, a F_{dest} = 1 - alfa, ali postoje i druge formule za mešanje boja. Prvo se renderuju objekti koji se ne provide, a tek na kraju objekti koji se provide, od najdaljeg ka najbližem.

Face culling je tehnika odsecanja strana. Proverava da li je neka strana okrenuta ka kameri ili ne i ako nije odbacuje je. Zapravo, proverava se da li je strana **unutrašnja** ili **spoljašnja**. Na primer, unutrašnjost kocke se ne renderuje bez potrebe. Jedan od načina za određivanje da li je strana okrenuta ka kameri je orijentacija trougla - **winding number**. Ako je orijentacija suprotna od kazaljke na satu ta strana je okrenuta ka kameri. OpenGL računa orijentaciju redosledom kojim su vertexi navedeni. Uključuje se komandom `glEnable(GL_CULL_FACE)`, a podešava sa `glCullFace(GL_FRONT)`, `glCullFace(GL_BACK)` ili `glCullFace(GL_FRONT_AND_BACK)` što odseca redom spoljašnje (prednje), unutrašnje (zadnje) ili obe strane.