

Եռաչափ գարածության պարկերումը

Համլեփ Պեփրոսյան

«Քվանդ» վարժարան
Նեփազոփական աշխարհանք
2022

Բովանդակություն

1	Ներածություն	2
1.1	Դիմնական գաղափարը	2
1.2	Տարածության մարդու ընկալումը	2
1.3	Դասական լուսանկարչական սարքի աշխատանքը	3
1.4	Դամակարգչային գործերակը	3
2	Աշխատանքային միջավայրը	5
2.1	Windows-ում պարուհանի սփեղծում	5
2.2	Շեյդերներ	5
3	Ray-Marching	6
3.1	Գաղափարը	6
3.2	Իրականացման օրինակ	7
3.3	Նեռավորության ֆունկցիաներ	7
3.4	Լուսավորություն	9
3.5	Գործողություններ մարմինների հետ	10
4	Ray-Casting	12
4.1	Գաղափարը	12
4.2	Իրականացման օրինակ	12
4.3	Նեռավորության ֆունկցիաներ	13
4.4	Լուսավորություն	13
4.5	Գունային կարգավորում	14
5	Ray-Tracing	16
5.1	Գաղափարը	16
5.2	Իրականացման օրինակ	16
5.3	Նյութերի գեսակներ	17
5.4	Path-Tracing	19
6	Ամփոփում	20
7	Հղումներ	21
7.1	Օգբազործված նյութերը	21
7.2	Այլ հղումներ	21

1 Ներածություն

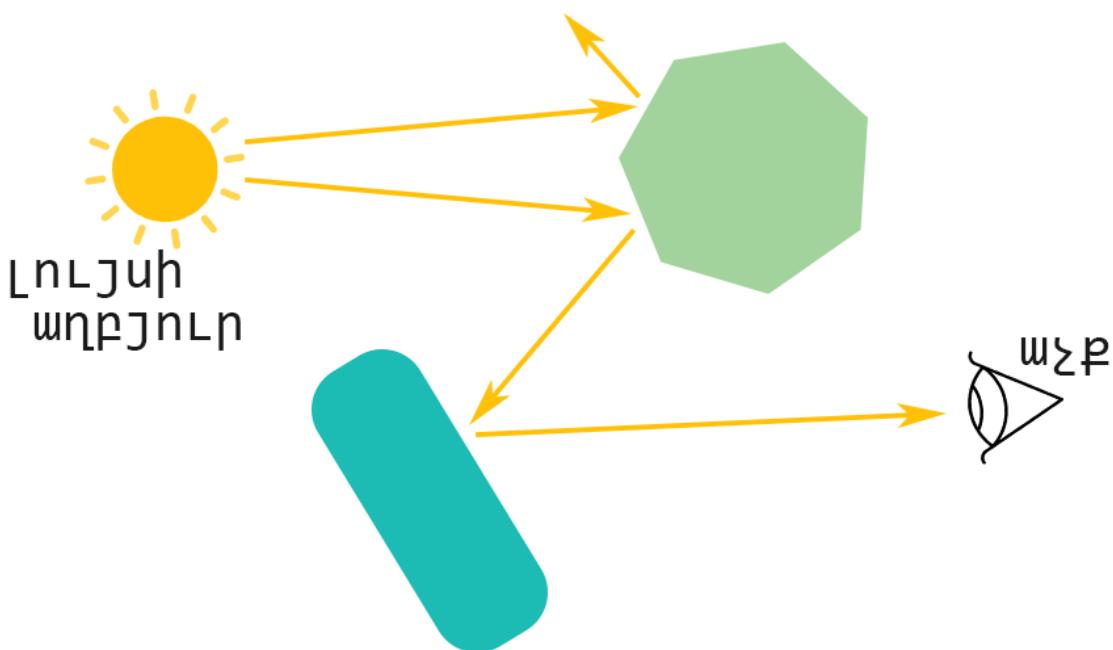
1.1 Դիմնական գաղափարը

Մենք մեզ շրջապատող աշխարհը ընկալում ենք որպես եռաչափ գործառնություն: Մարդը դարեր շարունակ ուսումնասիրել է իր շուրջը իրականացող երևույթները, փորձել գտնել օրինաչափություններ: Դրանք օգտագործելով՝ նկարիչները վարպետորեն վրձնել են ավելի ու ավելի իրականին մոտ պակերներ: Իսկ 20-րդ դարի երկրորդ կեսում, երբ հնարավորություն առաջացավ արագ հաշվարկներ կատարելու և որոշակի արդյունքներ ցուցադրելու էկրանին, ի հայր եկավ ինքնուրույն հաշվարկներով եռաչափ գործառնություններ պարկերելու ցանկություն:

Մարդուն որևէ ինֆորմացիա հաղերդելու համար սարքերը մեծամասամբ օգտագործում են էկրան: Այս դեպքը ևս բացառություն չէ: Եռաչափ գործառնության մասին պարկերացումը մեզ փրփում է էկրանի միջոցով: Սովորական էկրանը իրենից ներկայացնում է իրար կողք շարված մասնիկներից(պիքսելներից) կազմված ուղանկյուն: Յուրաքանչյուր պիքսել որոշվում է ընդամենը երկու բնութագրող թվերով՝ փողի և սյան համարներով: Ներկայացնելու դեպքում, եթե անդեսնենք նրա՝ մեզ չհետաքրքրող ֆիզիկական չափերը, գործ կունենանք երկչափ օրյեկտի հետ: Երկչափ օրյեկտներ են նաև նկարները և լուսանկարները: Եռաչափ գործառնունը պարկերվում է երկչափի մեջ, իսկ դիպողը մդրում ինքնուրույն վերսպեհծում է եռաչափ գործերակը:

1.2 Տարածության մարդու ընկալումը

Էկրանի միջոցով անհրաժեշտ է գործ ինֆորմացիա, որը դիպողը կնմանեցնի եռաչափ գործառնության: Նախ հասկանանք, որ մեզ հասանելի միակ գործիքը պիքսելներն են, որոնք կարող են ցույց դրական գույն: Մեր գլխավոր նպատակը հենց այդ գույները հաշվարկելն է: Այժմ ուսումնասիրենք իրական մոդելը՝ ֆիզիկական բացարրությունների մեջ չխորանալով:

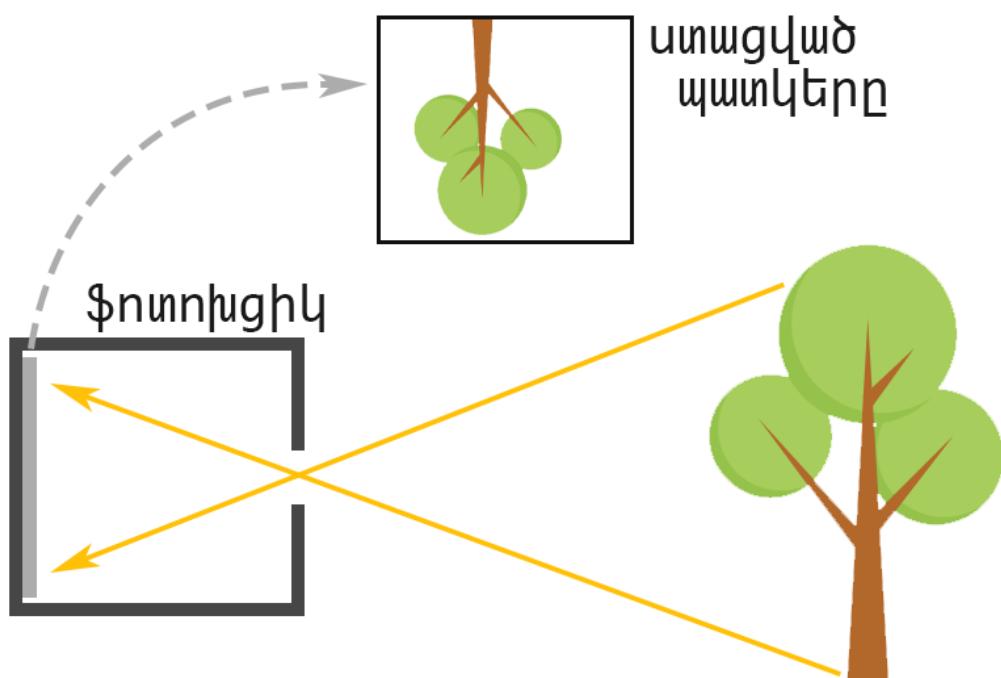


Տարածության մեջ կա լույսի աղբյուր, որը արձակում է որոշակի ուղղությամբ լույսի ճառագայթներ: Դիմնական լույսի աղբյուրը մեզ համար հանդիսանում է արևը, բայց լույսի աղբյուրներ կարող են լինել կրակը, լամպը, որոշ լուսարձակող նյութեր(օրինակ՝ ֆուֆոր) և այլն: Այդ արձակված ճառագայթները գործառնություն են, որոշները հանդիպում այլ մարմինների: Մարմինների բախվելիս ճառագայթները որոշ չափով կլանվում են, անդրադառնում մարմնի մակերևույ-

թից, բաժանվում մասերի և ցրվում գործեր ուղղություններով: Յուրաքանչյուր բախումից հետո ճառագայթները փոխում են հարկանիշները, ինչի շնորհիվ է դառնում են ավելի թույլ, գունավորում: Եվ վերջապես այդ ճառագայթներից որոշները հայտնվում են մեր գործառնության ընկալիչում՝ ազքում, որի շնորհիվ է առաջանում է մեզ հասանելի պարկերը:

1.3 Դասական լուսանկարչական սարքի աշխատանքը

Չանի որ լուսանկարչությունը այս թեմային շաբ մով ոլորք է, ավելորդ չի լինի որոշակի պարկերացում ունենալ նաև լուսանկարչական սարքերի աշխատանքի մասին: Պարզության համար դիմումը անդրանիկ սարքերի աշխատանքը:

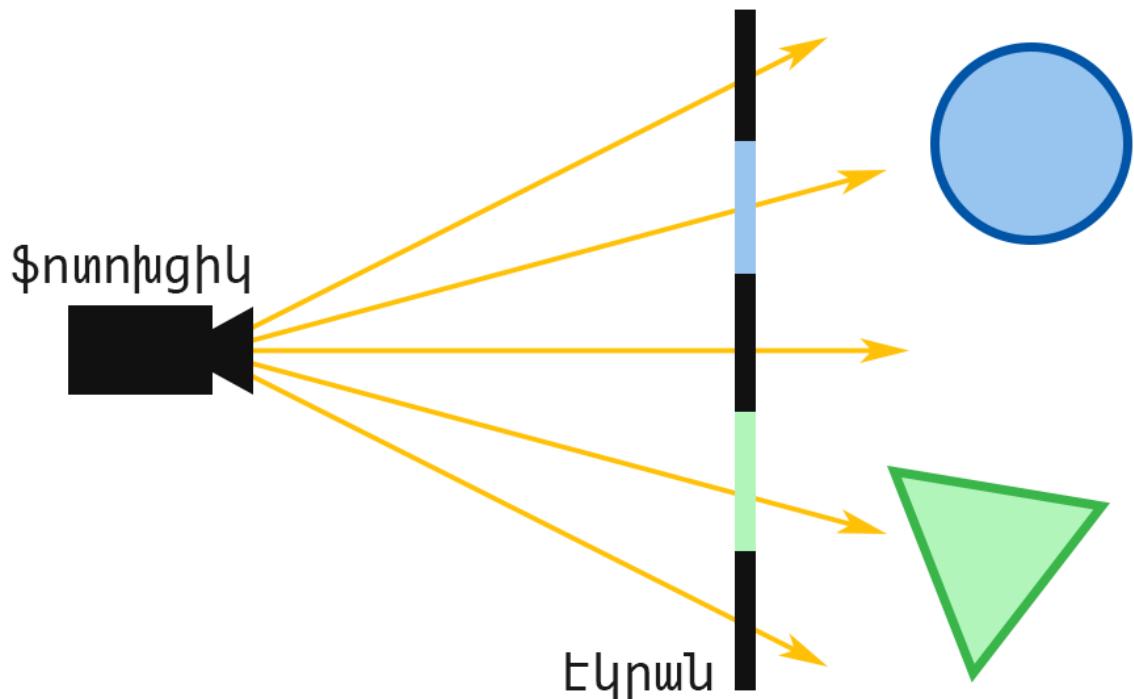


Ունենք ֆուրուստիկ, որի մի պարկին գեղադրված է նյութը, որն իր վրա պետք է պահպանի պարկերը, իսկ արկղի՝ դրան հանդիպակաց պարկին արված է անցք: Այդ անցքի միջով անցնելով լուսի ճառագայթները ընկնում են հարկուկ նյութի վրա և քիմիական ռեակցիայի միջոցով փոխում գույնը: Արդյունքում սրացվում է շրջաված և աջը ձախի փոխված պարկերը: Արկղի լուսանցքի հեռավորությունը և մեծությունը որոշում են լուսանկարի լուսավորության չափը, ինչպես նաև ֆոկուսավորման հեռավորությունը: Նենց սա էլ բացարձում է անցքի գոյությունը: Եթե հարկուկ նյութը դրվեր բոլորովին բաց միջավայրում, ապա բոլոր կողմերից լուս կընկներ, և լուսի ավելցուկից կարանայինք միագույն սպիրալ լուսանկար:

1.4 Շամակարգչային գործերակը

Այժմ եթե փորձենք ամբողջովին վերաբարել իրական գործերակը, կրախվենք խնդրի, որ լուսի աղբյուրներից բավականին շաբ ճառագայթներ ենք բաց թողնում, իսկ դրանց մեծամասնությունը նույնիսկ ազքի մեջ չի ընկնում: Չանի որ մեզ հետաքրքրում են միայն այն ճառագայթները, որոնք հայտնվելու են ազքում կամ ֆուրուստիկում, ապա հանգենք այն մոքին, որ ճառագայթները բաց թողենք հենց ֆուրուստիկից: Մեր գործը հեշտացնելու համար էլեկտրոնային կարող ենք գործերել ֆուրուստիկի դիմաց որոշ հեռավորությամբ, քանի որ այսպես կազմակերպենք ֆոկուս ընդունելու, պարկերը շրջելու խնդիրներից: Իսկ ինը վերաբերվում է ճառագայթներին, կարծես

թե բավարար է յուրաքանչյուր պիքսելի համապատասխանեցնել ֆունկցիկից իրեն ձգվող ճառագայթը և դրանից եզրակացնել իր գույնը:



Չանչ որ անձամբ դիպորդն էլ էկրանին նայում է որոշակի հեռավորությունից, այս մեթոդը բավականին իրականին մոտ արդյունք է վերադարձնում:

2 Աշխարհանքային միջավայրը

2.1 Windows-ում պարուիհանի սփեղծում

Առաջին հերթին անհարժեշգր է սփեղծել պարուիհան: Այդ առաջադրանքը հեշտությամբ կարելի է կարարել C++ լեզվի օգնությամբ, քանի որ այն աչքի է ընկնում իր արագագործությամբ և բավականին լավ հնարավորություններ է ընձեռում Windows օպերացիոն համակարգով աշխատելու համար: C++ լեզվում գոյություն ունի Windows.h գրադարանը, որը նախապեսված է հենց նոյնանուն օպերացիոն համակարգի համար, և թույլ է փալիս հեշտությամբ սփեղծել պարուիհան, բայց պիտեների հետ աշխատելու համար անհրաժեշգր կլինի ինքնուրույն գրել բավականին շատ ֆունկցիաներ: Այդ իսկ պարբառով ավելի հարմար է օգտագործել այդպիսի գործողությունների համար նախապեսված գրադարան, ինչպիսին է օրինակ SFML-ը: Գրադարանի հեղինակները այն ներկայացնում են, որպես երկասի պարկերներ սրանալու համար գործիքների հավաքածու, բայց նրա հնարավորությունները մեզ չեն սահմանափակում եռաչափի պարկերման ժամանակ:

SFML-ի հնարավորություններից ելնելով՝ կարելի է նաև ծրագրավորել սպեշնաշարի միջոցով շարժվելը, մկնիկի միջոցով դիպելու ուղղության փոփոխությունը և այլ գործողություններ: Այս ամենում ավելի չենք խորանա, քանի որ սա հիմնական թեման է:

2.2 Շեյշերներ

Ժամանակի ընթացքում հաշվարկներ պահանջող գրաֆիկական էլեմենտների ավելանալու պարզաբնակ սփեղծվել են համակարգին կցվող առանձին սարքեր (գրաֆիկական քարտեր): Այդ քարտերը բավականին փարբերվում են իշխանական պրոցեսորից: Քանի որ թեման հենց գրաֆիկական պարկեր հաշվարկելու մասին է, դրամաբանական կլինի ծրագիրը գրել հենց գրաֆիկական քարտի համար: Այդ նպարակով կաշխատենք շեյդերներով, որոնք այդ սարքերին հանձնարարվող ծրագրեր են հանդիսանում:

Շեյդերներ գրելու հայտնի լեզուներից են GLSL (OpenGL Shading Language)-ը և HLSL (High-Level Shading Language)-ը: HLSL-ը բավականին հարմարեցված է Windows-ի գրաֆիկական մաս ապահովող DirectX գրադարանի հետ աշխատելու համար: Բայց սեփական նախընդունակությունից ելնելով՝ այս հոդվածի շեյդերները կլինեն GLSL լեզվով: GLSL լեզվի Փայլերը պահպանում են .frag վեռաջակացությամբ:

Պարկերացում կազմելու համար նշենք, որ GLSL-ում արդեն իսկ կան գրված հաճախ օգտագործող Փունկցիաներ(օրինակ՝ dot - վեկտորների սկալյար արդադրյալ, normalize - վեկտորի երկարության բերում 1-ի, length - վեկտորի երկարություն) և փվյալների կառուցվածքներ (օրինակ՝ vec2, vec3, vec4 - իրենց համարին համապատասխան չափանի վեկտորներ): Նշենք նաև, որ շեյդերը յուրաքանչյուր պիքսելի համար աշխարհում է առանձին և ունի սահմանված գորբակի փոփոխականներ, որոնցից կօգտագործենք gl_FragColor-ը, ինչը իրենից ներկայացնում է vec4(red, green, blue, opacity)` համարվելով գրված պիքսելի գույնը, և gl_TexCoord-ը, որը պարունակում է վեկտորներուն վեճբարությաի վեկտորային մասին(SFML-ում շեյդերի սպացած պարկերը դուրս ենք բեկերու վեճբարությաի վեճբով): Նաև ունենք հնարավորություն սուինֆու բանայի բարի օգնությամբ հիմնական ծրագրից խնդրել որոշակի արժեքներ:

```
1 uniform vec2 u_res; // window resolution
2
3 void main{
4     vec3 color = vec3(1.0); // white
5     gl_FragColor = vec4(color, 1.0);
6 }
```

Տրված շեյտերի դեպքում կրեսնենք սպիտակ պարուիան, իսկ C++-ի կոնսոլային հավելված սպեհօնու դեպքում, նաև կրետելազմբենք, որ `ures` փոփոխականը չի օգտագործվել:

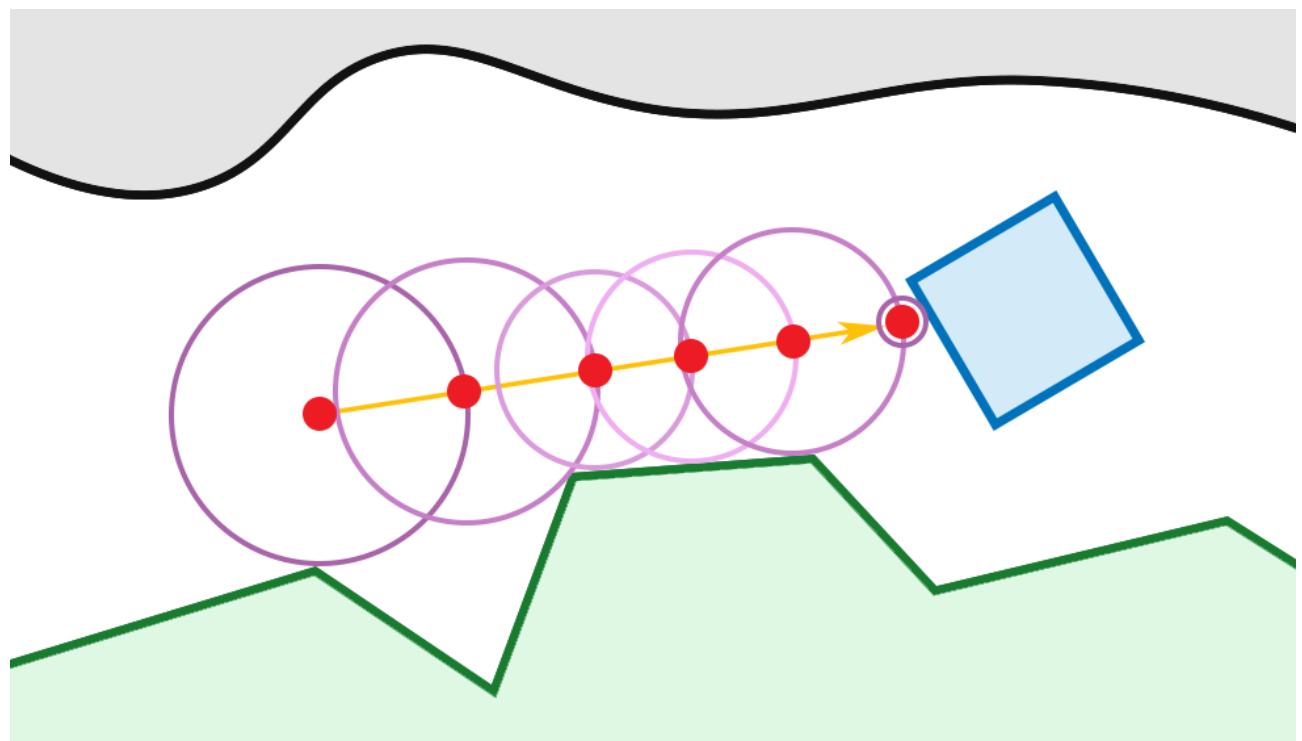
Յուրաքանչյուր պիքսելի սեփական ճառագայթը համապատասխանեցնելուց հետո, անհրաժեշտ է, այդ ճառագայթն օգտագործելով, գրնել պիքսելի գույնը: Այսինքն հետևենք ճառագայթին և գրանցենք, թե ինչ մարմինների է բախվում: Այդ բախումները գրանցելու ամենապարագած գործերակներն են Ray-Marching-ը և Ray-Casting-ը: Սպորև մանրամասն խոսենք դրանցից յուրաքանչյուրի մասին:

3 Ray-Marching

3.1 Գաղափարը

Որոշելու համար, թե պրված ճառագայթը առաջինը ո՞ր մարմնի հետք կբախվի, կադարենք հետևյալ քայլերը.

1. գպնենք ճառագայթի սկզբնակեփի հեռավորությունները բոլոր մարմիններից և վերցնենք դրանցից փոքրագույնը,
 2. դեղափոխենք սկզբնակեփը ճառագայթի ուղղությամբ և սրացված հեռավորությամբ,
 3. կրկնենք առաջին գործողությունը, քանի դեռ շաբ չենք հեռացել սկզբնական դիրքից, բավականաչափ չենք մոլոր որևէ մարմնի, կամ չենք կապարել առավելագույն քանակի քայլեր:



Վերցնելով հեռավորություններից փոքրագույնը՝ մենք բացառում ենք, որ ճառագայթը մինչ այդ ուրիշ մարմնի կիանդիպի: Սկզբնական դիրքից առավելագույն հեռավորություն սահմանելով՝ հաշվի չենք առնում դիպման կետից բավական հեռու մարմինները, ինչը որոշ դեպքերում կարող է զգալիորեն քչացնել պահանջվող գործողությունների քանակը: Այժմ հասկանանք, թե ինչու չենք համարում, որ ճառագայթը բախվել է մարմնին, միայն այն դեպքում, եթե նրանից ներկայիս սկզբնակետի հեռավորությունը 0 է: Նկատենք, որ եթե ճառագայթը մարմնի մակերևույթին չընկնի ուղղահայց, ապա սկզբնակետը անվերջ կմոդենա մարմնին, բայց նրա հեռավորությունը մակերևույթից 0 չի դառնա: Նեփաբար սպիտակած ենք բախման հեռավորություն սահմանել: Խոկ

Եթե ճառագայթը ուղղված լիներ մարմնի մակերևույթին գուգահեռ, ապա սկզբնակետը միշտ կունենար նոյն հեռավորությունը մարմնից և անվերջ կլիղափոխվեր ճառագայթի ուղղությամբ: Կարող է թվայի, որ առավելագույն հեռավորության սահմանումը կլուծի այս խնդիրը, բայց սկզբնակետի մարմնին մոտ լինելու դեպքում այդպես չէ: Նեփաքար կսահմանենք նաև քայլերի առավելագույն քանակ: Իհարկե, այս բոլոր սահմանափակումները կարող են հանգեցնել որոշակի թերությունների, բայց դեպքերի մեծամասնությունում դրանք աննշան են:

3.2 Իրականացման օրինակ

```

1 // ro - ray origin, rd - ray direction
2
3 vec3 rayMarch(vec3 ro, vec3 rd) {
4     vec3 pos = ro;
5     float surfed = 0.0;
6
7     for(int i = 0; i < MAX_STEPS; i++) {
8         float dist = getDist(p);
9         surfed += dist;
10        if(dist < HIT_D) {
11            return vec3(1.); // white
12        }
13        if(surfed > MAX_D){
14            return vec3(0.);
15        }
16        pos += rd * dist; // shifting start point
17    }
18    return vec3(0.); // black
19 }
```

Եվ օգտագործենք այն.

```

1 uniform vec2 u_res;
2 uniform vec3 u_pos; // camera position
3
4 vec3 rayMarch(vec3 ro, vec3 rd) { ... }
5
6 void main() {
7     // aligning texture to center
8     vec2 uv = (gl_TexCoord[0].xy - 0.5) * u_res / u_res.y;
9
10    vec3 rayOrigin = u_pos;
11    vec3 rayDirection = normalize(vec3(1.0, uv));
12
13    vec3 color = rayMarch(rayOrigin, rayDirection);
14
15    gl_FragColor = vec4(color, 1.0);
16 }
```

Տրված ֆունկցիան կվերադարձնի սպիրակ, որին կա բախում, հակառակ դեպքում՝ սև: Միակ բացը այսպես մնում է `getDist` ֆունկցիան, ինչը գրելու համար անհրաժեշտ է կարողանալ գիտել յուրաքանչյուր մարմնից հեռավորություն:

3.3 Նեռավորության ֆունկցիաներ

Հարմար է օգտագործել հեռավորության նշան պարունակող ֆունկցիաներ (Signed Distance Fields), որոնց բացասական լինելու դեպքում հասկանալի է դաշնում, որ հարցում ենք կապարում

մարմնի ներսից: Նաև յուրաքանչյուր մարմնի համար կարող ենք ստեղծել առանձին տիպ (struct), որը թույլ կփառ բոլոր անհրաժեշտ գովազները պահել հավաք: Սկզբու որոշ օրինակներ.

- գունդ,

```
1 // p - camera position
2 // Sphere(float radius, ...)
3 float sdSphere(vec3 p, Sphere s) {
4     return length(p) - s.radius;
5 }
```

- ուղղանկյունանիստ,

```
1 // Box(vec3 sizes, ...)
2 float sdBox(vec3 p, Box b) {
3     vec3 q = abs(p) - b.sizes;
4     return length(max(q, 0.0)) + min(max(q.x, max(q.y, q.z)), 0.0);
5 }
```

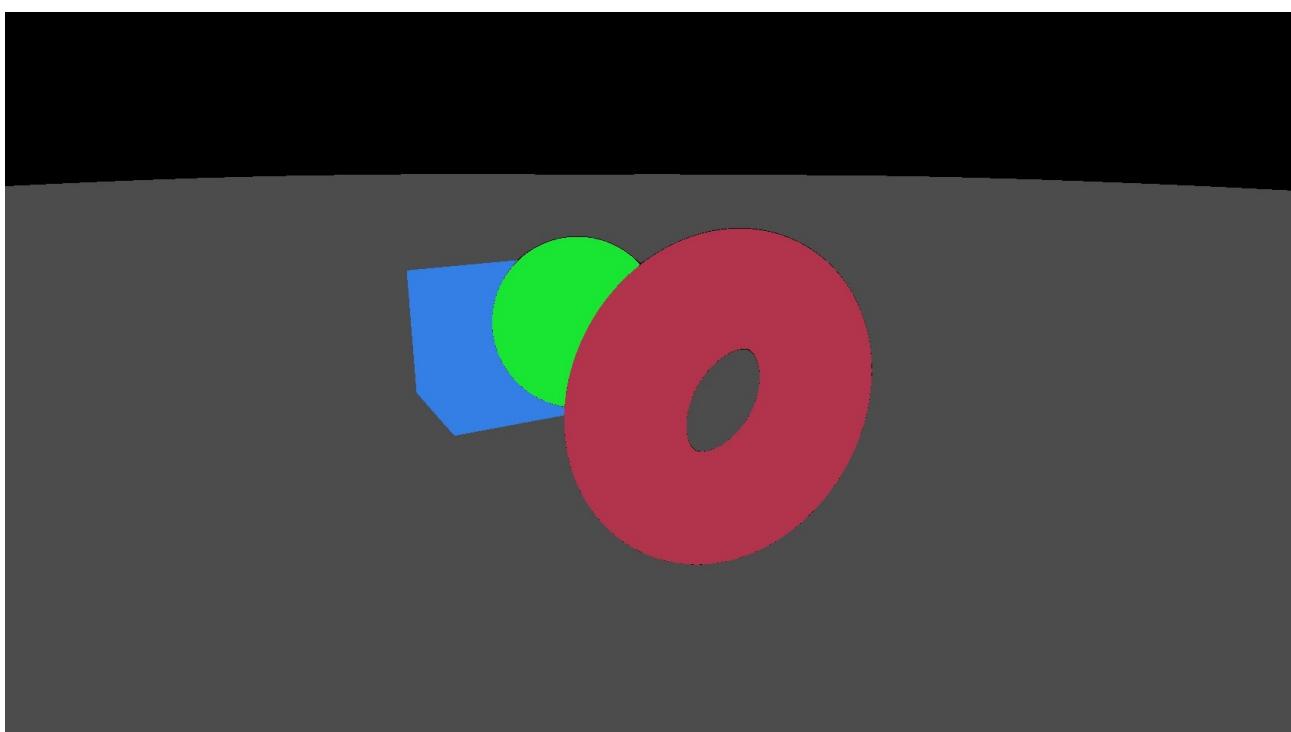
- հարթություն,

```
1 // Plane(vec3 normal, float height, ...)
2 float sdPlane(vec3 p, Plane pl) {
3     // n must be normalized
4     return dot(p, pl.normal) + pl.height;
5 }
```

- պոր:

```
1 // Torus(float outRadius, float inRadius, ...)
2 float sdTorus(vec3 p, Torus t) {
3     vec2 q = vec2(length(p.xz) - t.outRadius, p.y);
4     return length(q) - t.inRadius;
5 }
```

Մնում է միայն գրնել սրանցից փոքրագույնը getDist ֆունկցիայում և որոշել դրա գույնը:



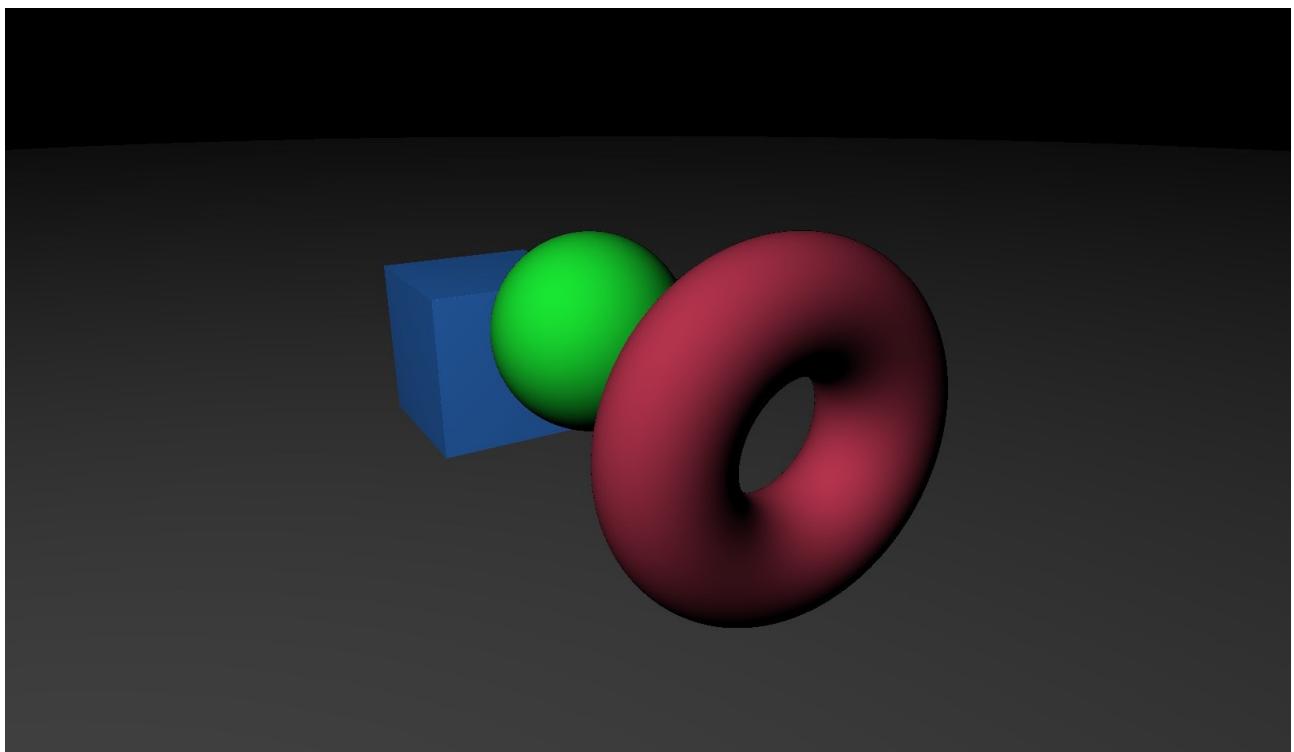
3.4 Լուսավորություն

Ինչպես երևում է վերջին նկարում, ամբողջ մարմինը ներկել ենք միևնույն գույնով, ինչը անբնական է: Բացակայում են սպիտակ, նաև լույսը: Դիմումը պահանջում է ավելացման պարզ գործություն:

Ընդունենք որևէ կոորդինատներ լույսի աղբյուրի համար: Դիմումը պահպան է մարմնին: Եթե այդ մակերևույթը ուղղված է դեպի լույսի աղբյուրը, ապա այն լուսավորված է, իսկ հակառակ դեպքում ոչ: Նեպատակ կարող ենք լուսավորության մակարդակ համարել այդ կեպում մակերևույթի նորմալի և դեպի լույսի աղբյուր ծզվող միավոր վեկտորի սկալյար արգարժյալը: Իսկ այդ կեպում կեպի նորմալը կարող ենք որոշել հայտնի հնարքով: Շարժենք այդ կեպը աննշան առանձին-առանձին Ox , Oy , Oz առանցքների ուղղություններով: Գպնենք մակերևույթից ունեցած հեռավորության փոփխությունը սկզբնական կեպի հետ համեմատած: Այդ սպացված թվերը ընդունենք, որպես վեկտորի կոորդինատներ: Դարձնենք վեկտորը միավոր և կսպանանք հարթության նորմալը, քանի որ գույնը ենք վեկտոր, որը ցույց է փալիս հարթությունից ամենաարագ հեռանալու ուղղությունը:

```

1 vec3 getNormal(vec3 p){
2     float d = getDist(p);
3     vec2 e = vec2(HIT_D, 0);
4     vec3 n = d - vec3(getDist(p - e.xyy), getDist(p - e.yxy), getDist(p - e.yyx));
5     return normalize(n);
6 }
7
8 float getLight(vec3 p){
9     vec3 lightPos = vec3(8.5, -7, -7);
10    vec3 l = normalize(lightPos - p);
11    vec3 n = getNormal(p);
12    float dif = clamp(dot(n, l), 0., 1.);
13    return dif;
14 }
```



3.5 Գործողություններ մարմինների հետ

Ray-Marching-ի ալգորիթմը հնարավորություն է գոլիս գրնել հեռավորություն ոչ միայն կոնկրետ մարմնից, այլ նաև երկու հոծ մարմինների միավորումից, նրանց հագումից, մարմնից, որից հանել ենք մեկ այլ մարմնի հետ հագումը և այլն: Ահա այդպիսի հնարավորություններ պվող գործողություններ.

- Երկու մարմինների միավորում (երկու մարմինները իրենց դիրքերում գեղադրելիս սպացվում է նույն արդյունքը),

```

1 float booleanUnion(vec3 p, Object object1, Object object2){
2     float sdA = distanceObject(p, object1);
3     float sdB = distanceObject(p, object2);
4     return min(sdA, sdB);
5 }
```

(Object-ը այսպես օգտագործված է որպես որևէ մարմնի նշանակում)

- Երկու մարմինների հագում,

```

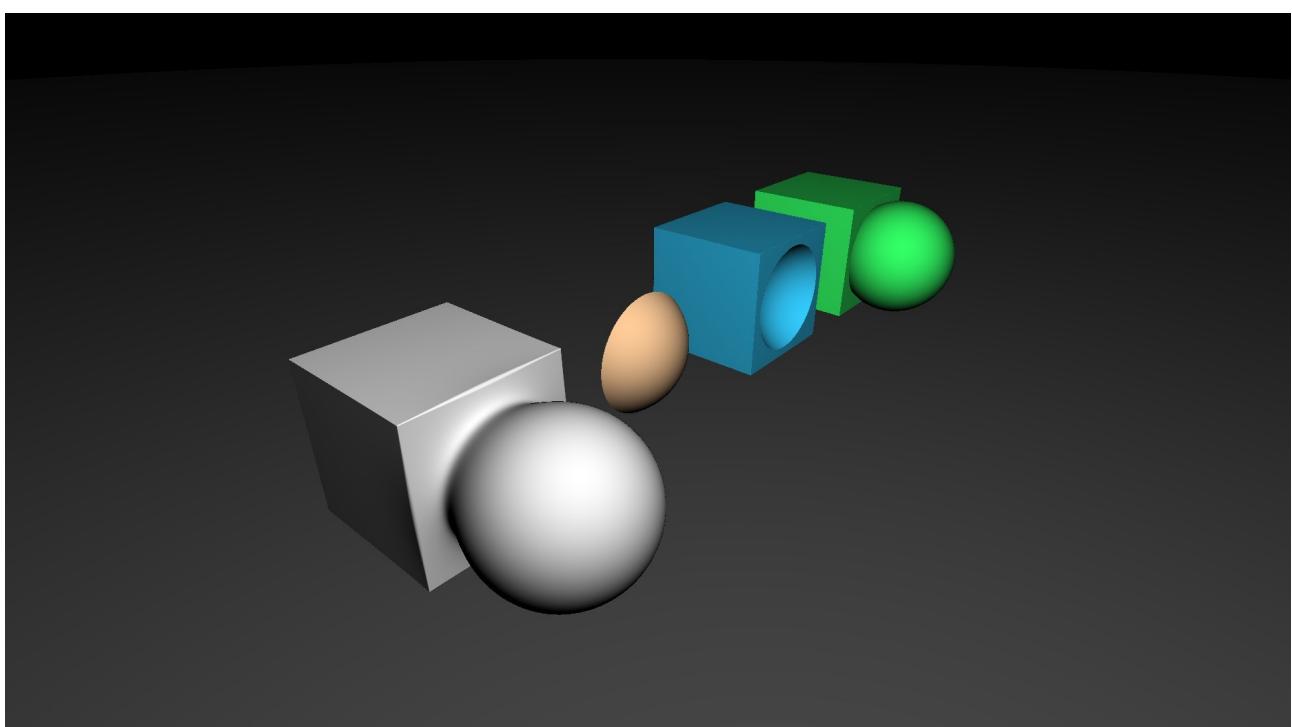
1 float booleanIntersection(vec3 p, Object object1, Object object2){
2     float sdA = distanceObject(p, object1);
3     float sdB = distanceObject(p, object2);
4     return max(sdA, sdB);
5 }
```

- Որևէ մարմնից հանում մյուսի հետ հագումը,

```

1 float booleanSubtraction(vec3 p, Object object1, Object object2){
2     float sdA = distanceObject(p, object1);
3     float sdB = distanceObject(p, object2);
4     return max(-sdA, sdB);
5 }
```

Եթե min, max ֆունկցիաները փոխարինենք, որևէ սահուն (smooth - smin, smax) համարժեքներով, կառընչվենք նկարում ձախից առաջինը պարկերվածի նման երևույթների:



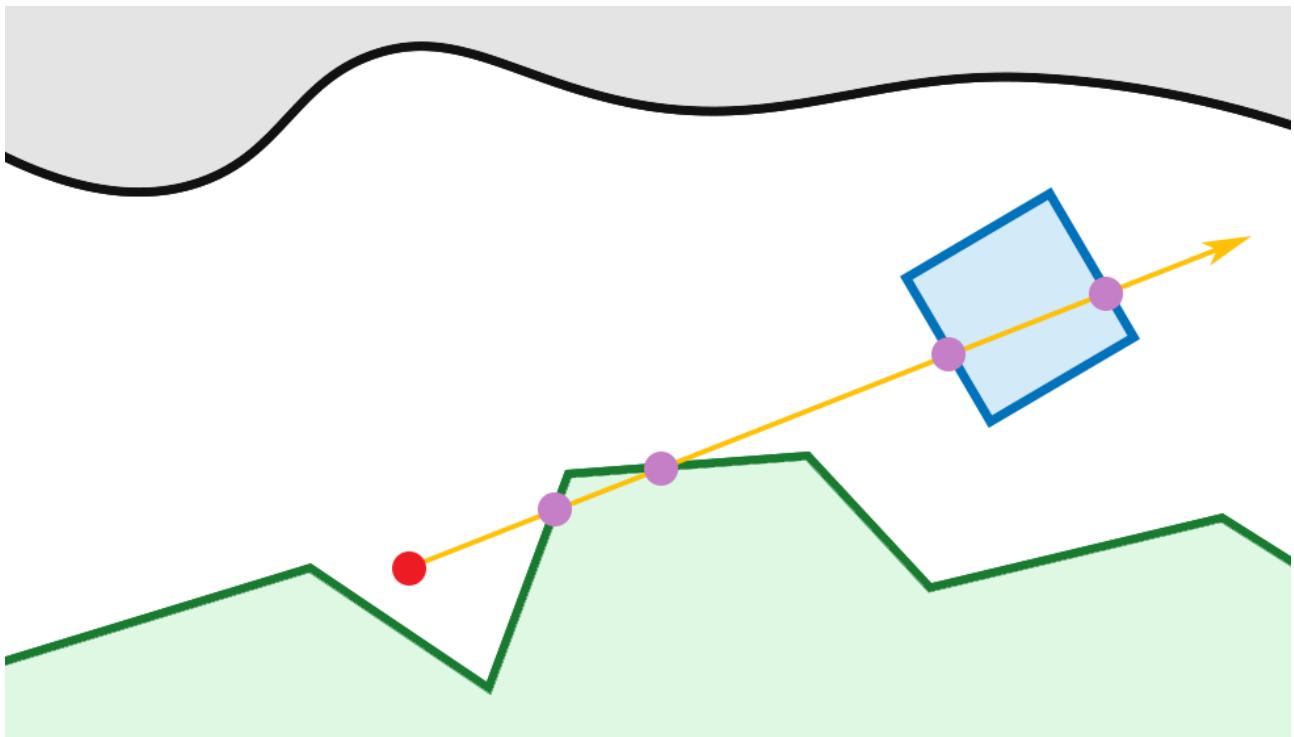
Սահուն փոքրագույն վերադարձնող ֆունկցիայի օրինակ.

```
1 float smin(float a, float b, float k){  
2     float res = exp(-k * a) + exp(-k * b);  
3     return -log(res) / k;  
4 }
```

4 Ray-Casting

4.1 Գաղափարը

Ի գարբերություն Ray-Marching-ի, այս մեթոդի մեջ բացի մաթեմատիկական չոր հաշվարկներից ոչինչ չկա: Գլուխությունը համար, թե որ մարմնի հետք է առաջինը բախվում ճառագայթը, հաշվարկում ենք ճառագայթի մարմինների հետք հարման կետերը, գլուխությունը սկզբնակետի հեռավորությունները և վերցնում փոքրագույնը: Այս գարբերակը ավելի է բարդացնում հաշվարկի բանաձևերը, գրկում մարմինների հետք արագ գործողություններ կատարելու հնարավորությունից, բայց աջքի է ընկնում իր արագագործությամբ և բարդ կառուցվածք ունեցող մարմինների հետք համեմարտաքար հեշտ աշխատելու հնարավորությամբ: Այսպես նաև բացակայում են նախորդ ալգորիթմին հապուկ պարկերման թերությունները:



Այս պարագայում նույնպես սահմանենք առավելագույն հեռավորություն շագ հեռու մարմինները անփեսելու համար: Ելնելով աշխատանքի պարզությունից՝ ուրիշ սահմանափակումներ ներմուծելու անհրաժեշտություն չկա:

4.2 Իրականացման օրինակ

```

1 vec3 rayCast(vec3 ro, vec3 rd) {
2     float dist = getDist(ro, rd);
3     if(dist == MAX_DIST) return (1.0);
4     return vec3(0.0);
5 }
```

Ինչպես նախորդ դեպքում, բերեցինք ֆունկցիայի օրինակ, որը մարմնի բախվելիս վերադարձնում է սպիրակ գույնը, հակառակ դեպքում՝ սևը: Այս ֆունկցիան համեմարտ Ray-Marching-ի զարգացման համար է թվում, բայց կբախվենք հակառակին `getDist` ֆունկցիան իրազործելիս:

4.3 Հեռավորության ֆունկցիաներ

Ray-Casting-ի օգտագործումը դժվարացնում է նշանով հեռավորությունների իրազործումը, և ծանոթությունը քարտացնելու համար կիրաժարվենք դրանցից:

Բերենք Ray-Marching-ի համար գործառնությունը՝ որը համարժեք է մասնաւոր մեջ կատարվող հեռավորության գործառնությանը:

- գունդ,

```
1 // Sphere(float radius, ...)
2 vec2 sphIntersect(vec3 ro, vec3 rd, Sphere sphere) {
3     float b = dot(ro, rd);
4     float c = dot(ro, ro) - sphere.radius * sphere.radius;
5     float h = b * b - c;
6     if(h < 0.0) return vec2(-1.0); // no intersection
7     h = sqrt(h);
8     return vec2(-b - h, -b + h); // two intersections
9 }
```

- ուղղանկյունանիստ,

```
1 // Box(vec3 sizes, ...)
2 vec2 boxIntersection(vec3 ro, vec3 rd, Box b) {
3     vec3 m = 1.0 / rd;
4     vec3 n = m * ro;
5     vec3 k = abs(m) * b.sizes;
6     vec3 t1 = -n - k;
7     vec3 t2 = -n + k;
8     float tN = max(max(t1.x, t1.y), t1.z);
9     float tF = min(min(t2.x, t2.y), t2.z);
10    if(tN > tF || tF < 0.0) return vec2(-1.0); // no intersection
11    return vec2(tN, tF); // two intersections
12 }
```

- հարթություն,

```
1 // Plane(vec3 normal, float height, ...)
2 float plaIntersect(vec3 ro, vec3 rd, Plane p) {
3     return -(dot(ro, p.normal) + p.height) / dot(rd, p.normal);
4 }
```

- իսկ գործառնությունը համար անհրաժեշտ կլինի լուծել գոնե չորրորդ կարգի հավասարում, ինչը բավականին բարդ է իրազործելը, և Ray-Casting-ի համար գործից հեռավորության ֆունկցիան չենք ներկայացնի:

4.4 Լուսավորություն

Ray-Casting-ում նույնպես կարող ենք կիրառել նախորդ գլխում առաջարկված լուսավորության մոդելը: Այս պարագայում մակերևույթների նորմալները կարող ենք հաշվարկել հսկակ բանաձևերով: Չթանազծի դեպքում կարացվի իր կենդրումից դեպի հարման կեզ ձգվող միավոր վեկտորը, ուղղանկյունանիստի դեպքում՝ իր կենդրումից դեպի մակերևույթ ուղղված միավոր վեկտորը, իսկ հարթության դեպքում՝ իր սահմանման մեջ մկան վեկտորը:

Դիշենք, որ մարմինները ոչ միայն լուսավորվում են լույսից, այլ որոշ դեպքերում այն կարող են անդրադարձնել: Վերցնենք մեր ուղղության վեկտորի անդրադարձը մակերևույթի նկարմամբ և սպուզենք, թե ինչքանով է ուղղված դեպի լույսի աղբյուրը: Սպացված արդյունքը իրակակին մոդելներու համար կարող ենք փոքրացնել այդ լույսի բաժինն՝ այն բարձրացնելով որևէ ասրի-ճան:

```

1 vec3 light = normalize(vec3(-0.5, 0.75, -1.0));
2 float diffuse = max(0.0, dot(light, n));
3 float specular = pow(max(0.0, dot(reflect(rd, n), light)), 16.0);
4 color *= mix(diffuse, specular, 0.5);

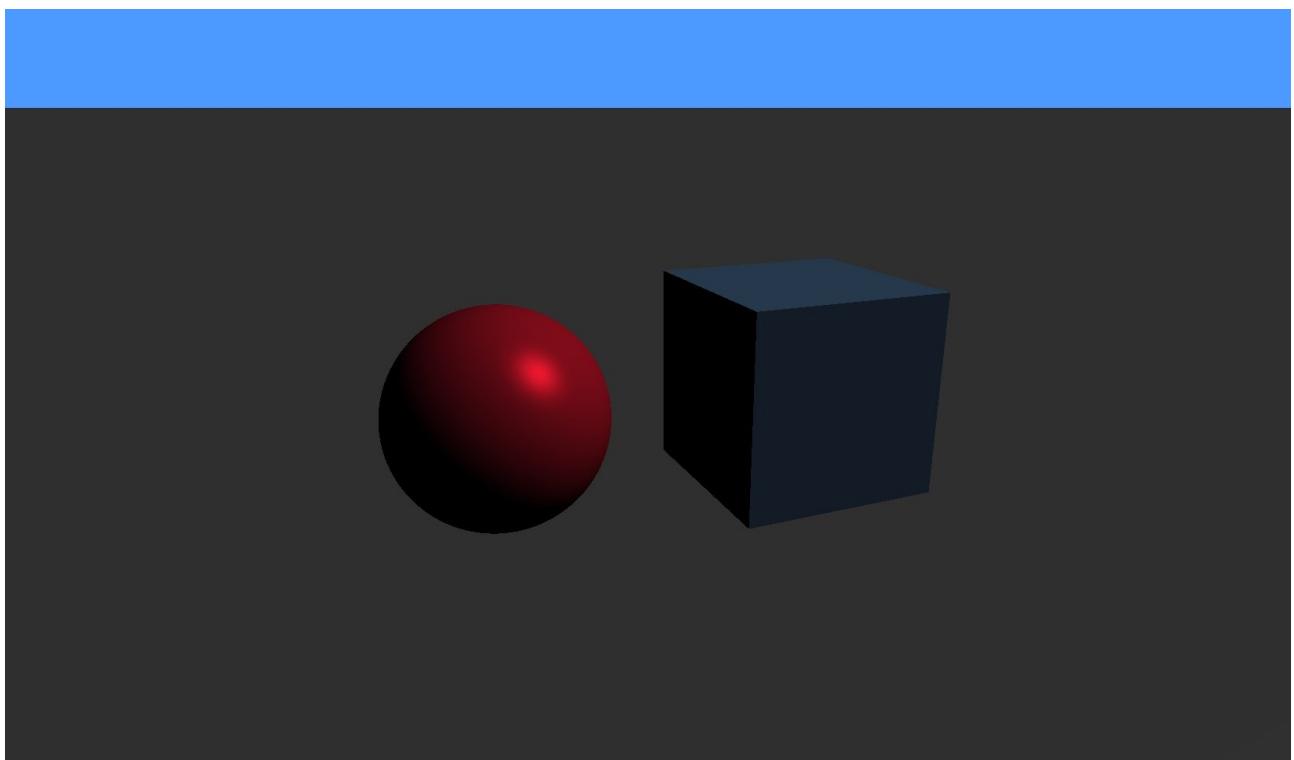
```

Մինչ այս ճառագայթի անխոչընդույք անցման դեպքում վերադարձել ենք սև գույնը: Փոխարինենք այն երկնքով: Երկնքում ավելացնենք նաև արևանման պարկեր՝ օգտվելով մարմնից լույսի անդրադարձի համար կիրառված հնարքից:

```

1 // light - light direction
2 vec3 getSky(vec3 rd, vec3 light) {
3     vec3 color = vec3(0.3, 0.6, 1.0); // color of sky
4     vec3 sunclr = vec3(0.95, 0.9, 1.0); // color of sun
5     sunclr *= pow(max(0.0, dot(rd, light)), 16.0);
6     return clamp(sunclr + clr, 0.0, 1.0); // number from 0 to 1
7 }

```



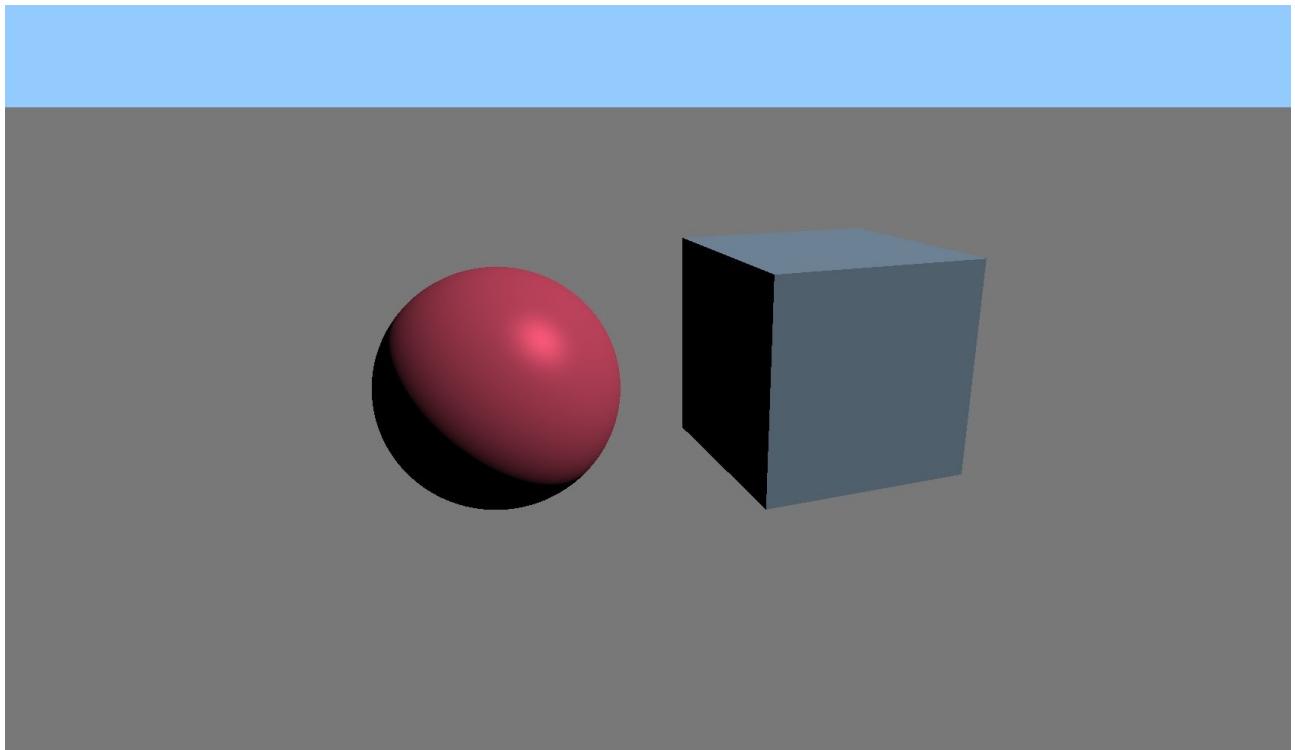
4.5 Գունային կարգավորում

Մինչ այս եղած բոլոր նկարներում գույները այդքան էլ իրականին մոփ չեն: Եթե ուշադրություն դարձնենք համակարգչում գույներից սևը համարվում է 0-ն, սպիտակը՝ 1-ը, իսկ դրանց միջին մոխրագույնը՝ 0.5: Գործ ունենք գծային համակարգի հետ: Այդ գործերակը այդքան էլ նման չէ մեր ընկալածին, այդ իսկ պարբռառով էլ խորհուրդ է գրվում կարգարել գամմա կարգավորում: Այդքան էլ անհրաժեշտություն չկա խորանալու գամմա կարգավորման աշխատանքի մեջ, ուղղակի նշենք, որ գույնին համապատասխանող թվերը պեսք է բարձրացնենք 0.45 ասդիման:

```

1 color.r = pow(color.r, 0.45);
2 color.g = pow(color.g, 0.45);
3 color.b = pow(color.b, 0.45);
4 gl_FragColor = vec4(color, 1.0);

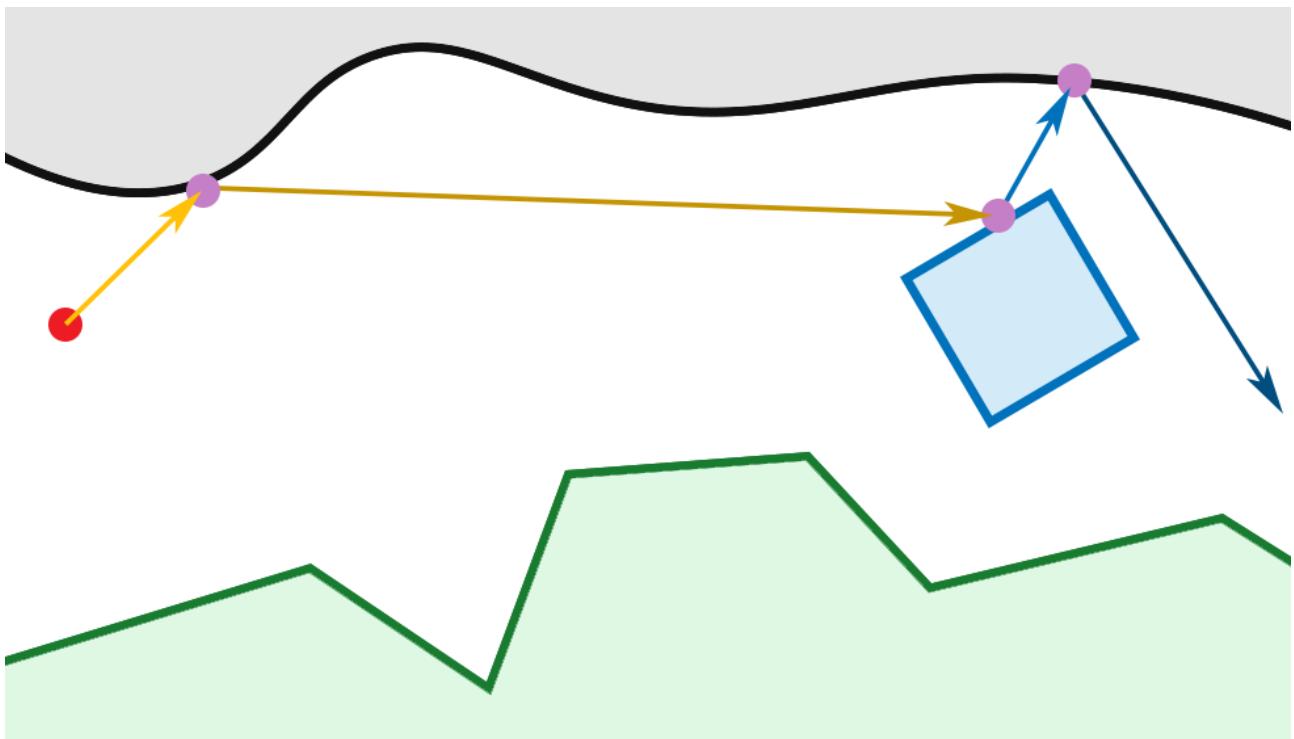
```



5 Ray-Tracing

5.1 Գաղափարը

Մինչև հիմա բոլոր իրազործումներում բաց ենք թողել ճառագայթ, հանդիպել առաջին բախ-մանը և դադարեցրել: Բայց իրական կյանքում այդպես չէ: Ինչպես նշել էինք ներածությունում, լույսը բազմաթիվ բախումների է հանդիպում և փոխում իր հավկանիշները: Փորձենք հակառակ հերթականությամբ վերականգնել այդ բախումները: Այն գաղափարը, ըստ որի պեսք է հեպսենք ճառագայթի ընթացքին կանվանենք Ray-Tracing:



Փորձենք կառուցել հասարակ մոդել: Ամեն բախման ընթացքում հաշվի առնենք լույսի կլանումը, գույնի փոփոխությունը: Եվ եթե հանդիպենք լույսի աղբյուրի, կանգ առնենք: Այս պահին համարենք, որ բոլոր մարմինները հայելային են, որպեսզի ճառագայթի ապագա ուղղությունը կարողանանք հեշտությամբ որոշել: Ներմուծենք նաև առավելագույն անդրադարձների քանակ: Նկատենք, որ այլս մեզ պեսք չեն զա լույսի ու սրվերի մեր օգբագործած մոդելները, բայց որպես լույսի աղբյուր կարող ենք շարունակել օգբագործել երկինքը:

5.2 Իրականացման օրինակ

```

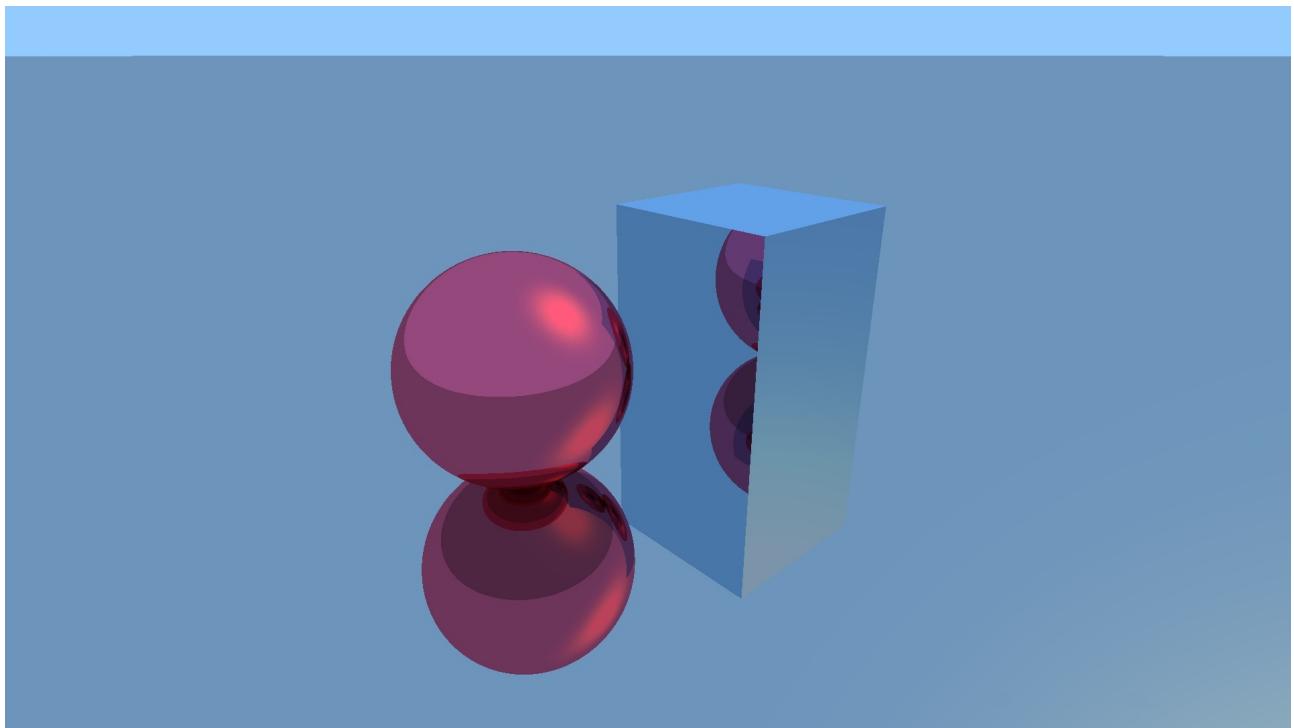
1 bool rayCast(inout vec3 ro, inout vec3 rd, inout vec3 color) {
2     vec3 clr; // color of intersection object
3     vec2 minIt = vec2(MAX_DIST); // minimum distance
4     vec3 n; // normal
5
6     // finding minimum, normal and color
7     ...
8
9     // Sky
10    vec3 light = normalize(vec3(-0.5, 0.75, -1.0));
11    if(minIt.x == MAX_DIST) {
12        color *= getSky(rd, light);
13        return true;

```

```

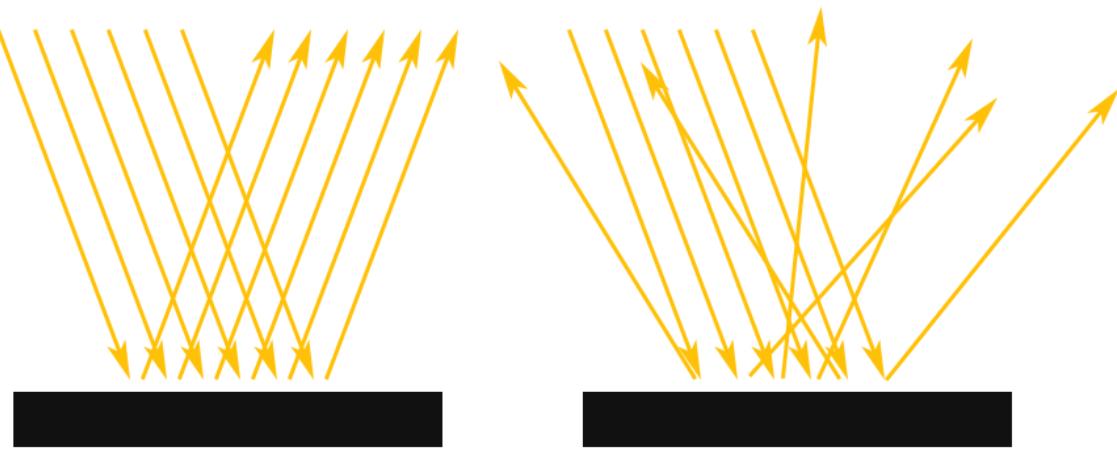
14 }
15
16 color *= clr;
17 ro += rd * (minIt.x - 0.001);
18 rd = reflect(rd, n);
19 return false;
20 }
21
22 vec3 rayTrace(vec3 ro, vec3 rd) {
23     vec3 color = vec3(1.0);
24     for(int i = 0; i <= MAX_DEPTH; i++){
25         if(rayCast(ro, rd, color)){
26             return color;
27         }
28     }
29     return color;
30 }
```

rayCast Փունկցիային փոփոխականները դրալիս ենք `inout` դեսքով, որպեսզի նա սրանա դրանց հասցեները: Վվելի մանրամասն այս ամենի մասին գրված է GLSL լեզվի պաշտոնական կայքում: Ահա զամնա կարգավորմամբ սրացված պարկերը (`MAX_DEPTH = 30`):



5.3 Նյութերի դեսակներ

Բացի հայելային նյութերից, գոյություն ունեն փայլափ, մետաղական փայլով և այլ հագլա նիշներով նյութեր: Չբարդացնելու համար համարենք, որ նյութերը ունեն մեկ հագլա նիշ, թե ինչքան են փայլում: Իրական կյանքում նյութերի փայլափ լինելը որոշում է նրանց մակերևույթի խորդութորդությունը: Ճառագայթը, ընկնելով այդպիսի մարմնի վրա, անդրադառնում է մեզ համար անորոշ ուղղությամբ:



փայլուն

փայլատ

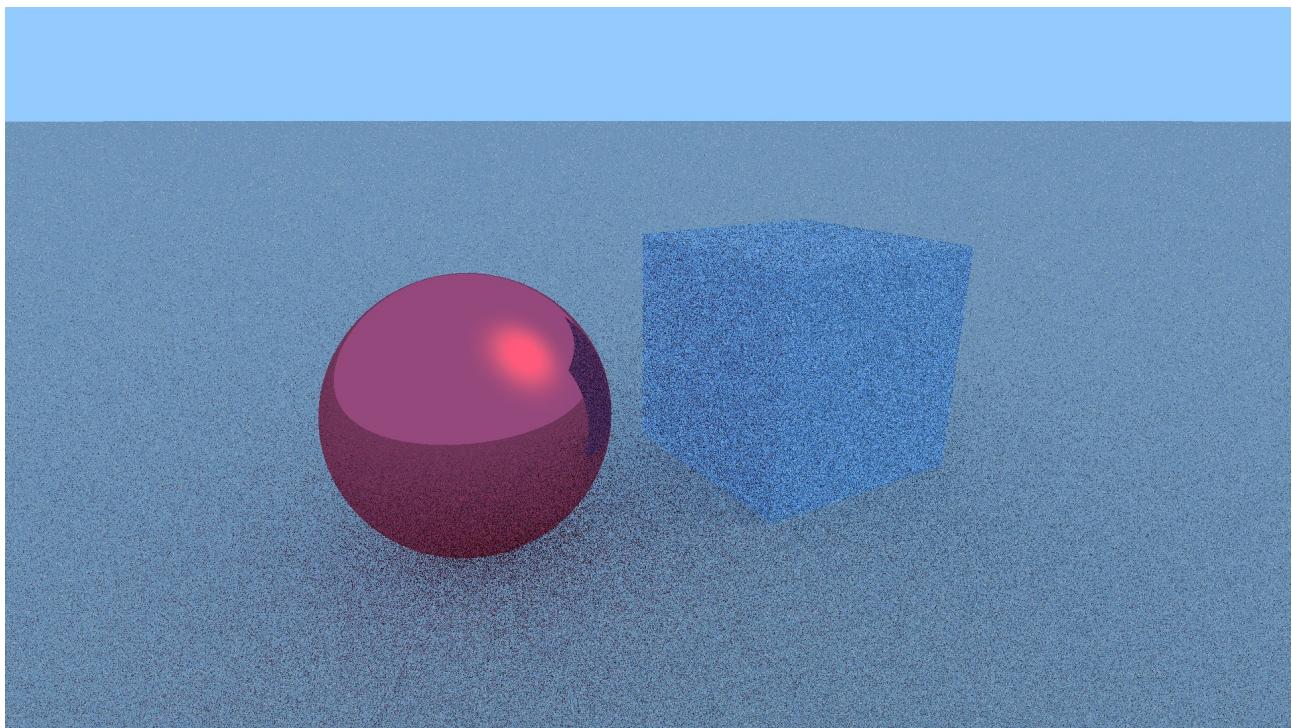
Եթե փորձենք այդ ամենը վերարփադրել մեր ծրագրում, ապա ճառագայթը այդպիսի մարմնի վրա ընկնելուց հետո պետք է անդրադարձնենք պարահական ուղղությամբ: Այս ամենը իրազործելու համար սահմանենք նյութի մոդել:

```
1 struct Material {
2     vec3 color;
3     bool light;
4     float roughness;
5 };
```

որպես roughness-ը կարգահայքի խորհուրդության չափը, կլինի $[0; 1]$ միջակայքի թիվ և ցույց կփա, թե հայելային անդրադարձված ճառագայթը ինչքանով ենք շեղելու դեպի պարահական ընդունվածը:

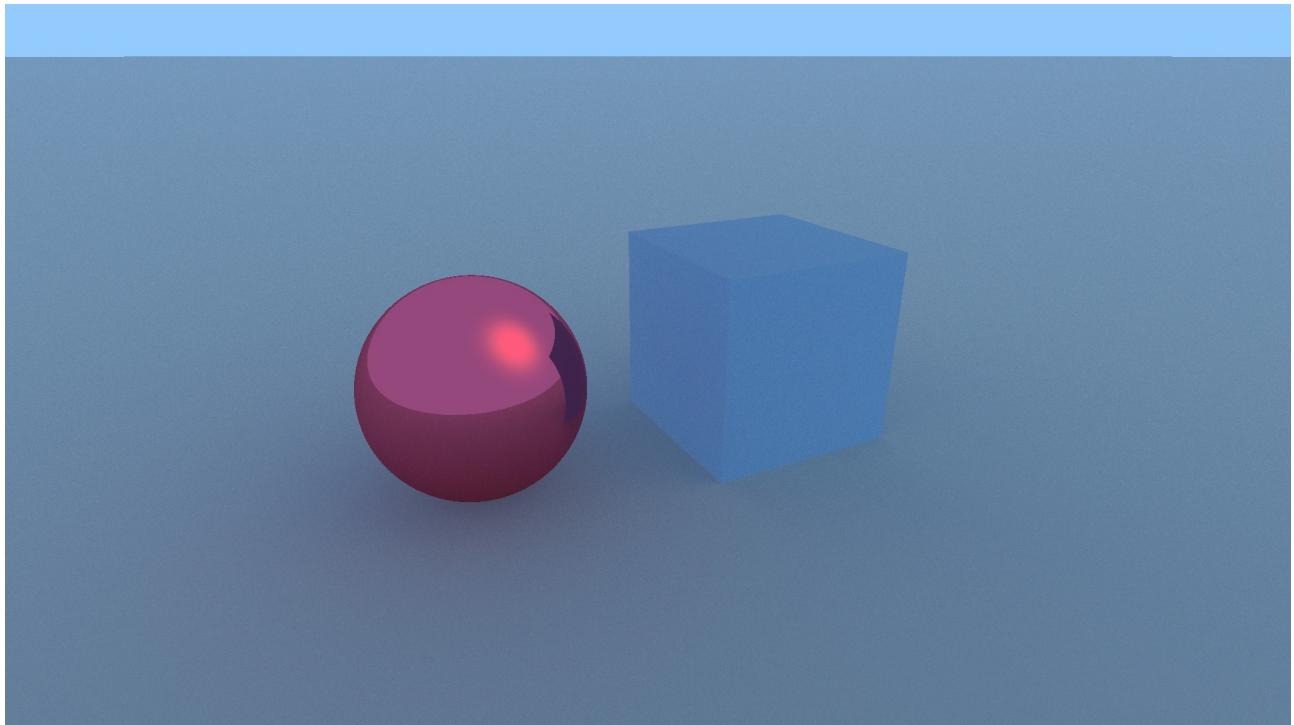
```
1 vec3 spec = reflect(rd, n);
2 vec3 diff = normalize(dot(n, randomOnSphere()) * randomOnSphere());
3 rd = mix(spec, diff, material.roughness); // reflected ray
```

Աշխափեցնենք ծրագիրը և կսրանանք նմանադիպ պարկեր.



5.4 Path-Tracing

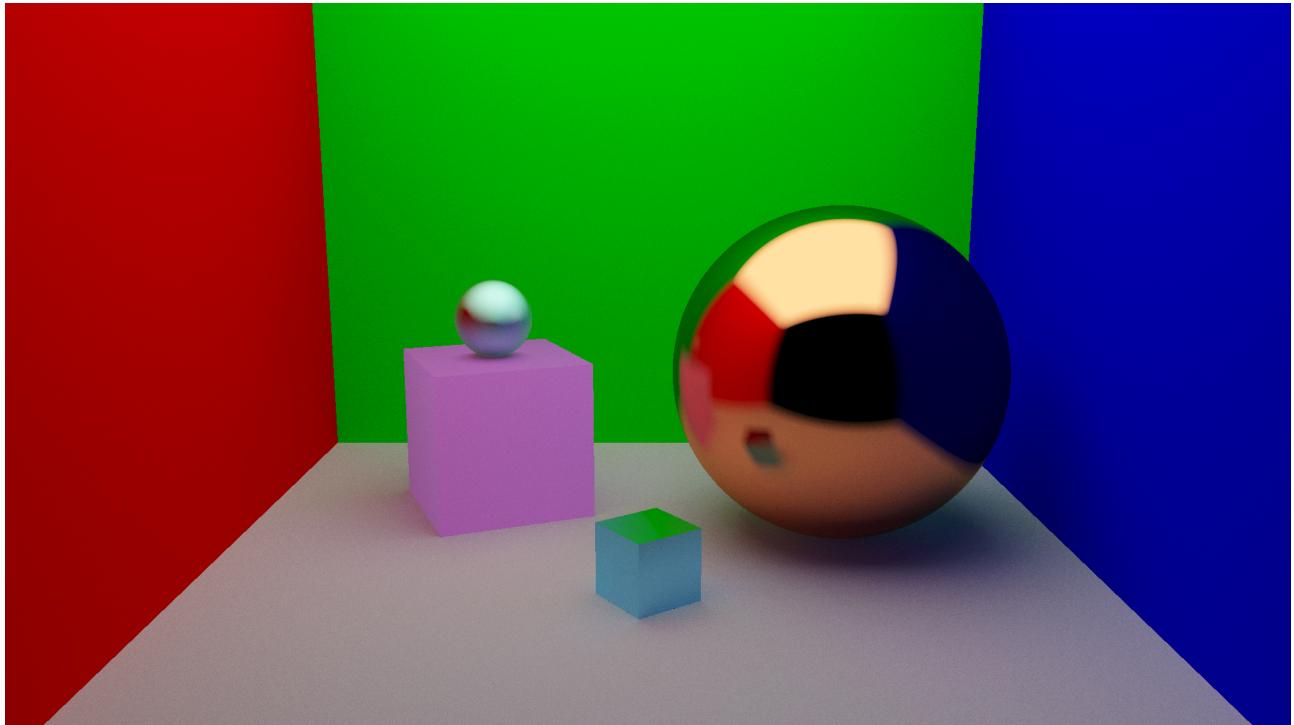
Ինչպես դեսանք վերջին նկարում պարահական ճառագայթների պարբառով փայլափ մակերևույթներին առաջանում է աղմուկ: Աղմուկների դեմ պայքարելու համար գոյություն ունեն բազմաթիվ միջոցներ: Ուսումնասիրենք Ray-Tracing-ի գործառնությունը՝ մեկ ճառագայթ: Դիշենք, որ յուրաքանչյուր պիքսելին համապատասխանեցրել ենք ընդամենը մեկ ճառագայթ: Այս գործառնությունը առաջարկվում է յուրաքանչյուր պիքսելի համար բաց թողնել բազմաթիվ ճառագայթներ, քանի որ դրանց պարահական անդրադառները կգործերվեն, ինչի շնորհիվ էլ կգործերվեն նաև նրանցից սպացված արդյունքները: Պիքսելը կներկենք այդ ճառագայթներից սպացված գույների միջինով: Կախված մեկ պիքսելին համապատասխանեցված ճառագայթների թվի մեծացումից արդյունքը կսկսի բարելավվել:



RAY_COUNT = 100

6 Ամփոփում

Եվ վերջապես, ստացված ծրագրի իրականին մոտ լինելը համոզվելու համար, փորձենք ստանալ Կորնելյան համալսարանի հանրահայք կորնելյան արկղը:



MAX_DEPTH = 64, RAY_COUNT = 64, օգլագործվել է նաև ստացված բազմաթիվ պարկերների միջինացում(Բոլոր պարկերներին համապարասիսան շեյդերները հղումներում նշված GitHub-ում):

7 Նղումներ

GitHub -

7.1 Օգբազործված նյութերը

1. Youtube Onigiri -
2. Inigo Quilez blog "3D SDFs" - <https://iquilezles.org/articles/distfunctions/>
3. Wikipedia - Gamma correction -

7.2 Այլ հղումներ

1. CodeBlocks [IDE] -
2. SFML [Գրաֆիկական գրադարան] -
3. L^AT_EX [Գրանշանային լեզու] -
4. A^MT_EX [L^AT_EX-ում հայերեն գրելու հնարավորթություն] -
5. MiK_TeX [L^AT_EX-ի գործիքների հավաքածու] -