SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 1036

Ostvarivanje vizualnog učinka kaustike u stvarnom vremenu

Teon Banek

Zagreb, lipanj 2015.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 2. ožujka 2015.

Predmet: **Diplomski rad**

DIPLOMSKI ZADATAK br. 1036

Pristupnik: **Teon Banek (0036458739)**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **Ostvarivanje vizualnog učinka kaustike u stvarnom vremenu**

Opis zadatka:

Proučiti fizikalne osnove nastajanja vizualnog učinka kaustike uslijed refleksije i refrakcije zraka svjetlosti. Proučiti načine implementacije učinka kaustike u prostoru slike i prostoru objekta te integraciju s bačenom sjenom objekta. Razmotriti mogućnosti implementacije ovog postupka na grafičkom procesoru. Ostvariti implementaciju nekih od postupaka ostvarivanja kaustike koji su ostvarivi u stvarnom vremenu. Načiniti ocjenu i usporedbu ostvarenih rezultata.
Izraditi odgovarajući programski proizvod. Rezultate rada načiniti dostupne putem Interneta. Radu priložiti algoritme, izvorne kodove i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Zadatak uručen pristupniku: 2. ožujka 2015.
Rok za predaju rada: 30. lipnja 2015.

Mentor:

Prof. dr. sc. Željka Mihajlović

Predsjednik odbora za diplomski rad profila:

Prof. dr. sc. Siniša Srbljić

Djelovođa:

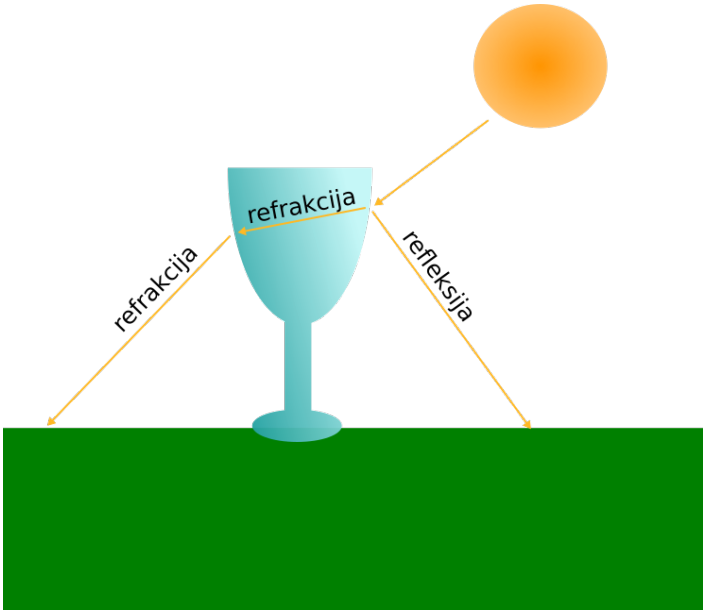
Doc. dr. sc. Tomislav Hrkać

Zahvale roditeljima i obitelji, mentoru prof. dr. sc. Željki Mihajlović, prijateljima i Gospodinu Svevišnjem na potpori i pomoći kroz cjelokupan studij.

SADRŽAJ

1. Uvod

Vizualni učinak kaustike nastaje konvergencijom više svjetlosnih zraka na jednu točku površine uslijed refleksije ili refrakcije, ilustrirano slikom 1.1. Do refrakcije dolazi kada val prelazi iz jednog medija u drugi te je opisana Snellovim zakonom 1.1.



Slika 1.1: Refrakcija i refleksija zrake

$$\frac{\sin\theta_1}{\sin\theta_2} = \frac{v_1}{v_2} = \frac{n_2}{n_1} \tag{1.1}$$

Gdje je:

- θ_1 kut upada,
- θ_2 kut refrakcije,
- v_i fazna brzina vala u mediju i i
- n_i indeks refrakcije medija i.

Refleksija nastaje kada val dolazi do drugog medija ali ne ulazi u njega već se odbija i vraća nazad u isti medij. Po zakonu refleksije upadni kut vala jednak je izlaznom odnosno reflektiranom kutu. Prema Fresnelovim jednadžbama, kada svjetlosni val dolazi

u kontakt s drugim medijem može doći do pojave i refleksije i refrakcije. Tako dolazi do efekta kaustike uslijed obje promjene svjetlosnog vala. Općenito se učinak kaustike može primijetiti kao svjetliji dijelovi tamne površine jer se svjetlost ne rasporedi jednoliko na toj površini. Najčešće se s tim efektom susrećemo kada svjetlost prolazi kroz čašu te se u njenoj sjeni mogu vidjeti zakrivljene svijetle linije kao na slici 1.2.

Slika 1.2: Učinak kaustike

Najkvalitetnija računalna grafika je ona koja se ne može razlučiti od onoga što vidimo u stvarnosti. Ostvarenje vizualnog efekta kaustike uvelike pridonosi toj kvaliteti. Postoji mnogo načina kako na računalu ostvariti taj efekt no većinom su računalno zahtjevni te se koriste kada prikaz ne mora biti interaktivan. Kod snimanja filmova se takvi algoritmi mogu izvršavati danima na računalu. Postoji i druga popularna industrija koja puno ulaže u računalnu grafiku a odavno je prestigla filmsku u godišnjim zaradama. Naravno, riječ je o industriji video igara. Računalna grafika u toj industriji mora ostvariti kvalitetu pri izvršavanju u stvarnom vremenu (engl. *real-time rendering*) kako bi interaktivnost ostala sačuvana.

U ovom radu je dan pregled različitih tehnika ostvarenja učinka kaustike. Naglasak je stavljen na ostvarenje tog efekta unutar zahtjeva interaktivnosti, to znači da izračun prikaza jedne slike se mora izvršiti unutar 16.6ms što odgovara prikazu od 60 sličica u sekundi. Iako je ustaljeno mišljenje da je za slike u pokretu dovoljno oko 24 sličice,

ljudsko oko može primijetiti razlike i kod većih frekvencija. Te razlike primijeti većina ljudi kada dođe u kontakt s interaktivnom računalnom grafikom, pa se u takvim slučajevima 60 sličica smatra optimalnom vrijednošću. Manji dio ljudi, poput vojnih pilota, čije je oko trenirano primjećivati stvari pri velikim brzinama mogu osjetiti neprirodnost animacije čak i kod prikaza manjeg od 120 sličica u sekundi.

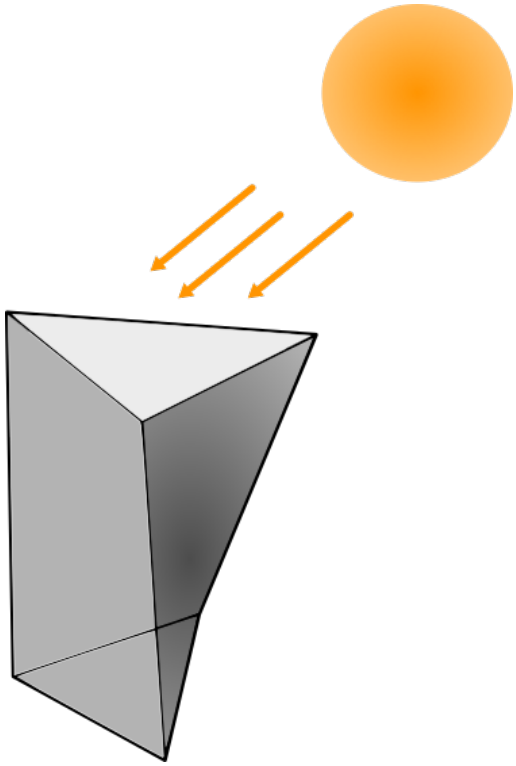
Osim usporedbe raznih metoda, kao dio rada ostvarena je i implementaciju algoritma mapiranja kaustike (engl. *caustic mapping*). U radu je opisana ta implementacija i dan pregled performansi na prosječnom računalu. Algoritam je uspoređen s izvedbom algoritma praćenja zrake (engl. *ray tracing*) koji koristi programsko sučelje Nvidia OptiX.

2. Tehnike prikaza kaustike

Kako je ovaj rad okrenut prikazu kaustike u stvarnom vremenu u nastavku su opisani algoritmi koji zadovoljavaju te zahtjeve, uključujući jedan koji tek polako ulazi u domenu interaktivnost. Svi algoritmi su temeljeni na prirodnim zakonitostima ali donose različite kompromise između brzine izvođenja i stupnja fizikalne simulacije. Posljednja opisana metoda sadrži i opis implementacije u zasebnom poglavlju ovog rada.

2.1. Prikaz kaustike zakrivljenim volumenima

Ovaj algoritam prikaza kaustike je vrlo sličan algoritmu volumena sjena (engl. *shadow volumes*) kojeg je opisao F. C. Crow (Crow, 1977). Ideja algoritma polazi od projekcije volumena u obliku prizme iz vrhova poligona na kojem dolazi do refrakcije i/ili refleksije (Watt, 1990). Svaka strana prizme je planarna čime se zanemaruje činjenica da snop zraka svjetlosti može stvoriti zakrivljenu površinu. Bolji rezultati se postižu kada se projicirane površine interpoliraju (Ernst et al., 2005). Tako dobivene površine prikazane slikom 2.1 su zakrivljene (engl. *warped volumes*) te je po njima algoritam dobio ime.



Slika 2.1: Volumen kaustike dobiven refrakcijom svjetlosti kroz trokut

Prvi korak algoritma je podjela geometrija na generatore kaustike (engl. *generators*) i geometriju koja prima efekt (engl. *receivers*). Neki objekti mogu spadati u obje grupe. Glavni dio algoritma je opisan sljedećim pseudokodom.

```
for all vidljivu točku T geometrije koja prima efekt kaustike do
    for all volumen kaustike V do
        if T unutar V then
            Izračunaj intenzitet kaustike
        end if
    end for
end for
```

Pseudokod 1: Prikaz efekta kaustike volumenom kaustike

Rezultati koje ovaj algoritam daje su vrlo dobri za interaktivan prikaz te se metoda lako nadopuni dinamičkim sjenama koristeći već spomenuti algoritam volumena sjena. Mana algoritma je u tome što generatori kaustike moraju biti trokuti te geometrija na kojoj se prikazuje efekt mora se moći zapisati u dubinsku mapu (engl. *depth map*) kako bi se moglo odrediti koje su točke unutar volumena. Kako ova metoda daje

brzo izvršavanje, prikaz kaustike nije precizan u fizikalnom smislu te kvaliteta, iako dovoljno dobra, nije na fotorealističnoj razini.

2.2. Algoritam praćenja fotona

U odnosu na prethodno opisani algoritam, sljedeći daje vjernije rezultate. Osnova je algoritam praćenja zrake (engl. *ray tracing*) te ga to čini računalno zahtjevnijim. Zanimljivo je to što na današnjim procesorima, te pogotovo na grafičkom procesoru se mogu ostvariti vrlo dobri rezultati u vrlo kratkom vremenu izvršavanja.

Učinak kaustike nastaje pomoću indirektne svjetlosti pa konvencionalno praćenje zrake se mora malo izmijeniti. Umjesto praćenja zrake iz točke očišta, zraka se počinje pratiti iz izvora svjetlosti. Taj algoritam je nazvan praćenje zrake unazad (engl. *backward ray tracing*) te se izvodi u dva koraka: mapiranje indirektne svjetlosti i uzorkovanje dobivene mape (Arvo, 1986). Dodatno se taj algoritam može pojednostaviti generiranjem dvije mape fotona umjesto kompletne indirektne svjetlosti (Jensen, 1996). Najveći problem performansi kod tih algoritama je praćenje zrake za koju se mora izračunati točka sjecišta s nekim poligonom geometrije. Grafički procesori mogu napraviti puno izračuna u paraleli te jedan od načina prilagodbe algoritma takvom izvršavanju jest praćenje zrake fotona unutar prostora ekrana (Krüger et al., 2006). Točke sjecišta se sada mogu lako odrediti pristupom dubinskoj mapi što uvelike ubrzava izvođenje. Na sličan način je izvedena implementacija ovog algoritma pomoću programskog sučelja Nvidia OptiX. Kasnije je ta implementaciju iskorištena kao primjer usporedbe.

Uz gore navedene modifikacije te izvršavanjem na grafičkom procesoru algoritam pokazuje vrlo dobre rezultate koji polako ali sigurno dolaze u domenu interaktivnosti. Također, sama kvaliteta prikaza je na visokoj razini. Glavni nedostatak koji je nastao prijenosom izračuna u prostor ekrana je nemogućnost prikaza indirektne svjetlosti odnosno kaustike dobivene objektima koji su izvan ekrana. To može narušiti uvjerljivost scene, no kako sam efekt nije toliko prominentan ti problemi lako mogu proći nezapaženo.

2.3. Mapiranje kaustike

Nakon kratkog izleta u zahtjevniji algoritam, ovo potpoglavlje se vraća unutar zahtjeva interaktivnog iscrtavanja. Ovaj algoritam je sličan metodi mapiranja sjena (engl. *shadow mapping*) jer se iz perspektive svjetlosti računa tekstura koja odgovara efektu

kaustike dobivenog refleksijom i/ili refrakcijom svjetlosnih zraka (Shah et al., 2007). Takva tekstura, nazvana mapom kaustike (engl. *caustic map*), se potom projicira na predmete kako bi se na njima prikazao vizualni učinak.

Algoritam se sastoji od sljedećih koraka.

1. Pohrani poziciju geometrije koja prima efekt u teksturu
2. Pohrani pozicije i normale reflektivne/refraktivne geometrije u teksturu
3. Kreiraj teksturu kaustike
4. Iscrtaj scenu s efektom kaustike

Pseudokod 2: Metoda mapiranja kaustike

Pozicije i normale geometrije se pohranjuju u teksture u koordinatnom sustavu svjetlosti te se u tom sustavu izvode izračuni na grafičkom procesoru. To obilježje algoritma omogućuje lako spajanje s algoritmom mape sjena. Glavni dio algoritma je generiranje teksture kaustike predstavljen sljedećim pseudokodom.

```
for all vrh V reflektivne/refraktivne geometrije do
    R = refrakcija(V.normala, smjerSvjetlosti)
    P = procjeniPresjecište(V.pozicija, R, geometrija)
    V.pozicija = P
end for
```

Pseudokod 3: Generiranje teksture kaustike

Vektor dobiven refrakcijom se može zamijeniti reflektiranim ako želimo ostvariti učinak kaustike uslijed refleksije. Ovako dobiveni vrhovi predstavljaju mjesta gdje pojedina zraka svjetlosti dolazi na površinu geometrije koja prima efekt kaustike. Kada ovakav izračun za više vrhova vrati isti rezultat to predstavlja fokusiranje svjetlosnih zraka na određenu točku uslijed čega i nastaje željeni efekt. Procjena točke presjecišta se može odrediti iterativnom Newton-Rhapson metodom gdje u ovom slučaju algoritam daje dobre rezultate već i nakon druge iteracije.

Algoritam mapiranja kaustike daje vrlo brze rezultate te se omjer performansi i kvalitete lako može podešavati rezolucijom kojom generiramo teksturu efekta kaustike. Manja rezolucija daje brz izračun ali dolazi do pojave aliasinga koji se može umanjiti nekom od metoda zamućivanja slike. Dodatna ušteda računalnih resursa se može postići i izgradnjom hijerarhijskih mapa kaustike (Wyman, 2008). Kod te modifikacije postavi se hijerarhija tekstura kako bi se lakše odbacili nepotrebni izračuni a

točke koje završe u međusobnoj blizini se zamjenjuju jednom većom točkom. Glavna mana ovog algoritma je nepreciznost u određivanju točke presjecišta pa uz navedeni aliasing dolazi i do pogrešnog prikazivanja koje je vidljivo i pri većim rezolucijama.

3. Implementacija mapiranja kaustike

Za potrebe implementacije ovog algoritma korišten je programski jezik C++ i OpenGL biblioteka za ostvarenje 3D grafike. Programi za sjenčanje koji se izvršavaju na grafičkom procesoru su napisani pomoću jezika GLSL. Poslije samog opisa implementacije, dana je usporedba rezultate s primjerom praćenja fotona napravljenom pomoću Nvidia OptiX programskog sučelja.

3.1. Opis programskog koda

Algoritam mapiranja kaustike je vrlo jednostavan ako se gleda s visoke razine. Prva tri koraka pseudokoda 2 prenesena u implementacijski jezik C++ su dana u nastavku.

```
// Receivers (everything)
caustic_map_.BindForWritingReceiver();
RenderSceneVisitor rv(global_matrices_ubo_, false);
rv.set_render_refractive(false);
rv.set_render_receiver(true);
scene_->AcceptVisitor(rv);
// Refractors
caustic_map_.BindForWritingRefractive();
rv.set_render_receiver(false);
rv.set_render_refractive(true);
scene_->AcceptVisitor(rv);
// Create caustic map
caustic_shading_.setIsRefractive(true);
caustic_map_.BindPositionMap(GL_TEXTURE0);
caustic_shading_.setPositionMap(0);
caustic_map_.WriteCausticMap();
```

Varijabla `caustic_map_` je instanca razreda `CausticMap`. Taj razred sadrži metode za manipuliranje spremnikom slike (engl. *framebuffer*) koji je povezan s teksturama

u koje se pohranjuju podaci potrebni za generiranje mape kaustike kao i sama mapa. Razred `RenderSceneVisitor` služi za obilazak grafa scene (engl. *scene graph*) kako bi se pravilno iscrtala geometrija. U varijabli `global_matrices_ubo_` je pohranjen indikator gdje se u memoriji grafičkog procesora nalaze matrice za pretvorbu 3D pozicije modela u koordinatne sustave svijeta scene, kamere i projekcije.

Generiranje spremnika slike pomoću OpenGL biblioteke je prikazano sljedećim kodom.

```
glGenFramebuffers(1, &fbo_);
glBindFramebuffer(GL_FRAMEBUFFER, fbo_);
```

Potom se generiraju teksture za pohranu boje, pozicije i dubinske mape. Te teksture se povežu s generiranim spremnikom slike koristeći funkciju `glFrameBufferTexture2D`.

```
glGenTextures(1, &color_map_);
glBindTexture(GL_TEXTURE_2D, color_map_);
glTexImage2D(GL_TEXTURE_2D, 0, GL_RGB32F, width_, height_,
             0, GL_RGBA, GL_FLOAT, 0);
...
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT0,
                       GL_TEXTURE_2D, color_map_, 0);
glGenTextures(1, &pos_map_);
...
glFramebufferTexture2D(GL_FRAMEBUFFER, GL_COLOR_ATTACHMENT1,
                       GL_TEXTURE_2D, pos_map_, 0);
glGenTextures(1, &depth_map_);
glBindTexture(GL_TEXTURE_2D, depth_map_);
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT, width_,
             height_, 0, GL_DEPTH_COMPONENT, GL_FLOAT, 0);
...
glFramebufferTexture2D(GL_DRAW_FRAMEBUFFER,
                       GL_DEPTH_ATTACHMENT,
                       GL_TEXTURE_2D, depth_map_, 0);
```

Postavke određenih parametara tekstura poput ponašanja prilikom uvećanja ili smanjenja teksture kod prikaza su uklonjene kako bi prikaz koda bio koncizniji. Teksture spremaju podatke u formatu brojeva s posmačnom točkom veličine 32 bita. Kako podaci koji se zapisuju predstavljaju pozicije i normale takav format nudi potrebnu preciznost uz veći trošak memorije. Posljednji korak je postavljanje oznake da spremnik sadrži dva mjesta za iscrtavanje.

```
GLenum draw_bufs[] = {GL_COLOR_ATTACHMENT0,
                      GL_COLOR_ATTACHMENT1};
glDrawBuffers(2, draw_bufs);
```

Generiranje teksture kaustike prema pseudokodu 3 implementirano je kao program sjenčanja vrhova (engl. *vertex shader*).

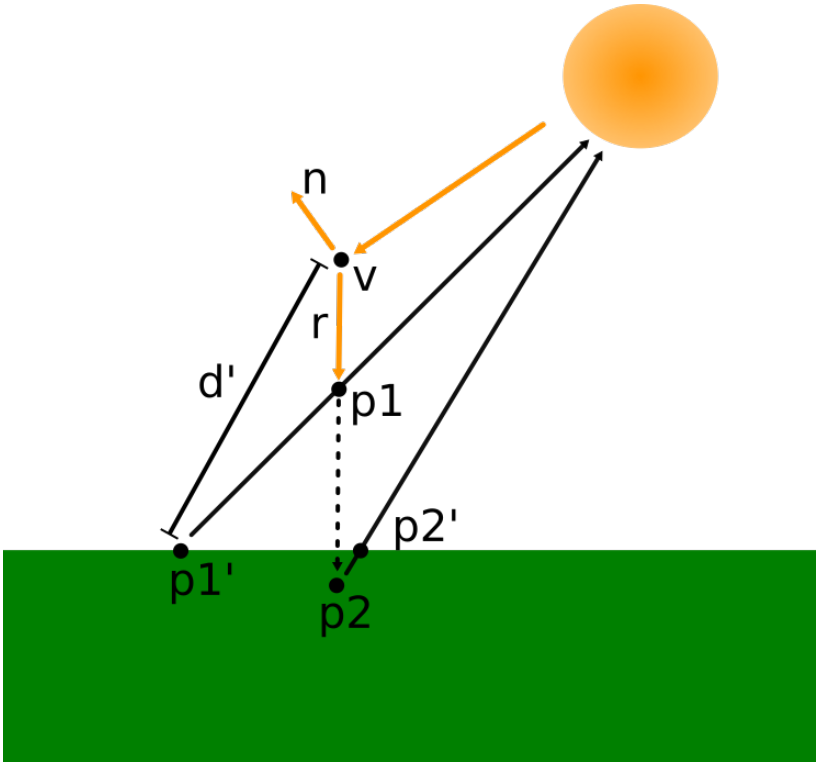
```
void displaceRefractive()
{
  vec4 inLightPos = vec4(Position, 1.0f);
  vec4 inLightNormal = vec4(Normal, 0.0f);
  inLightNormal = normalize(inLightNormal);
  vec3 lightDir = normalize(Position);
  vec3 r = refract(lightDir, inLightNormal.xyz, eta);
  r = normalize(r);
  vec3 inters = estimateIntersection(inLightPos.xyz, r, 1);
  gl_Position = transform.perspective * vec4(inters, 1.0f);
  pos0 = inLightPos.xyz;
  normal0 = inLightNormal.xyz;
}
```

Varijable `Position` i `Normal` su pozicija i normala trenutnog vrha geometrije kojeg obrađuje program te su one u koordinatnom sustavu svjetla. Nakon njihove normalizacije poziva se funkcija **refract** koja vraća refraktirani vektor. Ako želimo efekt kaustike dobiven refleksijom taj poziv možemo zamijeniti funkcijom **reflect**. Također je moguće ostvariti i učinak kaustike kod objekata koji su i reflektivni i refraktivni korištenjem obje funkcije. Zatim se pomoću dobivenog vektora odredi mjesto presjecišta funkcijom `estimateIntersection` te prenesu podaci u program za sjenčanje fragmenata (engl. *fragment shader*) varijablama `pos0` i `normal0`.

Za procjenu presjecišta je korištena modificirana metoda Newton-Rhapson (Shah et al., 2007). Ilustracija algoritma je dana slikom 3.1. Točka v predstavlja trenutan vrh geometrije, $\vec{n}$ je normala a $\vec{r}$ je vektor dobiven refrakcijom. Točke na refraktiranom vektoru su dane sljedećom jednadžbom.

$$p = v + d * \vec{r} \tag{3.1}$$

d je dužina od pozicije v uzduž vektora $\vec{r}$. Vrijednost te dužine se prvo postavi na 1 i izračuna nova pozicija $p1$. Projekcijom $p1$ iz perspektive svjetlosti na površinu geometrije dobiva se $p1'$. Zatim se udaljenost d' između projicirane točke $p1'$ i vrha v koristi u prethodnoj jednadžbi 3.1 kako bi se dobila točka $p2$. Ponovo se vrši projekcija



Slika 3.1: Izračun presjecišta

i dobiva se točka $p2'$ koja predstavlja procijenjeno presjecište. Implementacija tog algoritma se nalazi u funkciji `estimateIntersection`.

```
vec3 estimateIntersection(vec3 v, vec3 r, int iters)
{
  vec3 p1 = v + r;
  vec2 tc = calcTC(p1);
  for (int i = 0; i < iters; ++i) {
    vec4 recPos = texture(positionMap, tc);
    vec3 p2 = v + distance(v, recPos.xyz) * r;
    tc = calcTC(p2);
  }
  return texture(positionMap, tc).rgb;
}
```

Točke na površini geometrije su zapisane u teksturi kojoj se pristupa funkcijom **texture**. Ona prima referencu na teksturu putem varijable `positionMap` i koordinate teksture `tc`. Kako bi se pravilno pristupilo toj teksturi potrebno je izračunati koordinate dvodimenzionalne teksture iz trodimenzionalnih koordinata pozicije točke. Taj izračun je implementiran unutar funkcije `calcTC`. Pozicija geometrije `pos` se prvo mora projici-

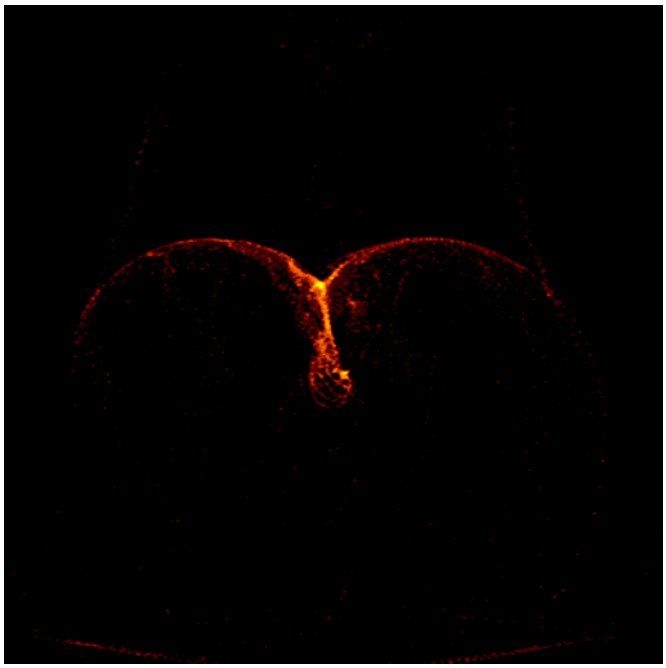
```
vec2 calcTC(vec3 pos)
{
  vec4 texPt = transform.perspective * vec4(pos, 1.0f);
  vec2 tc = vec2(0.5 * (texPt.xy / texPt.w) + vec2(0.5, 0.5));
  tc.y = 1.0 - tc.y;
  return tc;
}
```

rati perspektivnom transformacijom te se skalira unutar intervala $[0, 1]$. Koordinatu y je potrebno okrenuti jer je njena orijentacija u teksturi obrnuta u odnosu na orijentaciju iste osi geometrije. Dobivena sjecišta se proslijede programu za sjenčanje fragmenata (engl. *fragment shader*) koji ih zapiše u teksturu te ona predstavlja mapu kaustike. Kako zrake dobivene refleksijom ili refrakcijom na različitim vrhovima mogu završiti u istim sjecištima potrebno je zapisati ih u teksturu stapanjem alfa kanala transparencije (engl. *alpha blending*). Time će se i ostvariti efekt kaustike gdje će određena područja akumulirati više svjetlosti u odnosu na druge. Rezultat akumulacije i konačna mapa kaustike može se vidjeti na slici 3.2. Potom se ta tekstura koristi u završnom koraku kod iscrtavanja gdje za svaku točku geometrije se boja iz teksture doda boji geometrije. Programski kod koji radi to uzorkovanje i sjenčanje je sadržan sljedećim isječkom.

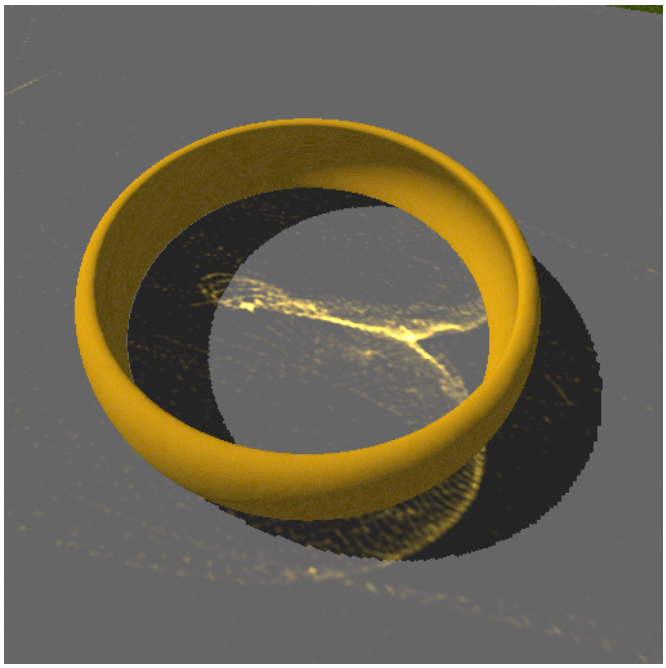
```
vec4 calcCaustic(vec4 pos)
{
  vec3 proj = pos.xyz / pos.w;
  vec2 uv;
  uv.x = 0.5f * proj.x + 0.5f;
  uv.y = 0.5f * proj.y + 0.5f;
  vec4 caustic = texture(gCausticMap, uv).rgba;
  caustic = caustic * caustic * vec4(gLight.color, 1.0f);
  return vec4(caustic.rgb * 100, 1.0f);
}
```

Koordinate pozicije u trodimenzionalnom prostoru se projiciraju u prostor teksture kao što je bio slučaj kod procjene sjecišta u funkciji `calcTC`. Varijabla `gCausticMap` referencira teksturu koja predstavlja mapu kaustike te se njena uzorkovana boja množi bojom izvora svjetlosti. U praksi se pokazalo da je dobro kvadrirati uzorkovanu boju čime se stvara veća razlika između dijelova na kojima se fokusiralo više svjetlosti i onih gdje je manje svjetlosti. Time prikaz izgleda bolje te se rijetke pogrešne procjene sjecišta i ne vide. Dobro je i drugačijim koeficijentima pomnožiti različite komponente

boje ako želimo prikazati objekt koji drugačije apsorbira valne duljine svjetlosti. Takve manipulacije bojom kaustike su moguće i u koraku generiranja mape kaustike. Rezultat prethodno opisane implementacije učinka se može vidjeti na slici 3.3. Na slici je moguće primijetiti aliasing kao i pogrešno izračunata presjecišta jer luk kaustike počinje prerano pa je izvan prstena.



Slika 3.2: Tekstura koja predstavlja mapiran učinak kaustike



Slika 3.3: Ostvareni učinak kaustike

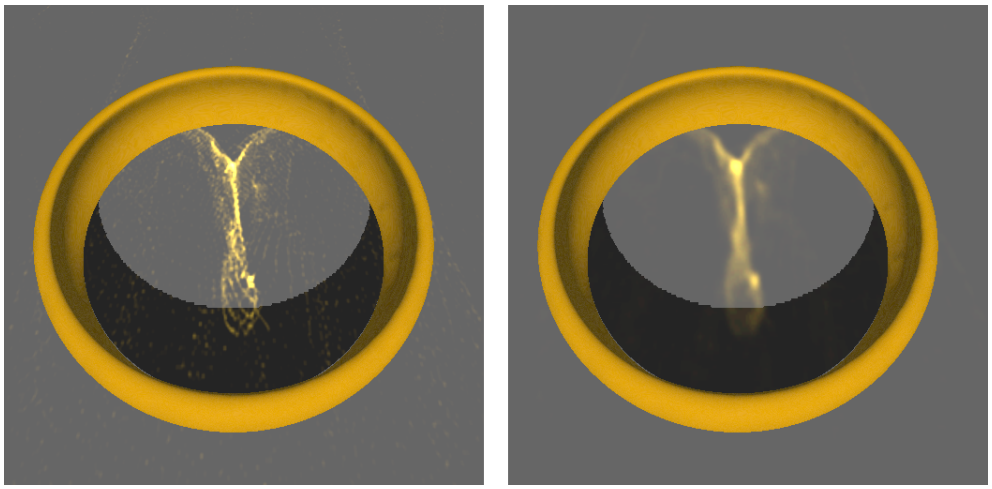
Kako bi se umanjila pojava aliasinga, teksturu kaustike su zamućene koristeći Gaussov filtar (Wolff, 2011). Implementacija filtra je napravljena unutar programa za sjenčanje te se sastoji od dva prolaza. Prvi prolaz dan je kodom u nastavku.

```
float dy = 1.0 / float(height);
vec4 sum = weights[0] * texture(texture0, texCoord0);
for (int i = 1; i < 5; ++i) {
  vec2 offset = dy * vec2(0.0, pixOffset[i]);
  sum += weights[i] * texture(texture0, texCoord0 + offset);
  sum += weights[i] * texture(texture0, texCoord0 - offset);
}
return sum;
```

Varijabla `weights` sadrži polje težina dobivenih izračunom Gaussove distribucije 3.2 za svaku točku uzorkovanja (u ovom slučaju je to deset točaka). Standardnu devijacija σ je postavljena na 2 a očekivanje μ na 0.

$$f(x, \mu, \sigma) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{(x-\mu)^2}{2\sigma^2}} \tag{3.2}$$

Drugi prolaz se razlikuje u tome što se uzorkuju elementi slike po x osi. Dobiveni rezultat se može vidjeti na slici 3.4.



Slika 3.4: Lijeva slika je bez filtra, desna koristi Gaussov filtar

Kompletan prikaz je dodatno unaprijeđen korištenjem metode mapiranja okoline (engl. *environment mapping*) za ostvarenje refleksije i refrakcije (Akenine-Moller et al., 2008). Prvi korak algoritma je pohrana 6 slika okoline u teksturu oblika kocke (engl. *cube map texture*) prikazani funkcijom `CubemapTexture::Load`. Zatim se pri svakom iscrtavanju reflektivnog ili refraktivnog objekta izvrši refleksija/refrakcija vektora svje-

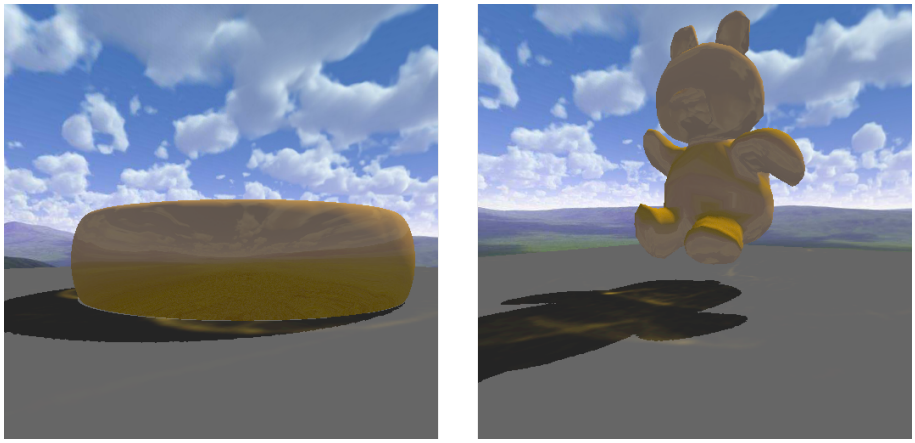
```
bool CubemapTexture::Load()
{
  glGenTextures(1, &texture);
  glBindTexture(GL_TEXTURE_CUBE_MAP, texture);
  for (int i = 0; i < 6; ++i) {
    img::Image *image = img::Image::Load(files[i]);
    if (!image) {
      std::cerr << "Error: Unable to load image: "
                << files[i] << std::endl;
      return false;
    }
    glTexImage2D(types[i], 0, GL_RGBA,
                 image->GetWidth(), image->GetHeight(), 0,
                 GL_BGR, GL_UNSIGNED_BYTE, image->GetPixels());
    delete image;
  }
  ...
  return true;
}
```

tlosti unutar program za sjenčanje vrhova. Tako dobiveni vektor određuje trodimenzionalne koordinate teksture (kod teksture oblika kocke koriste se 3D koordinate a ne 2D). Program za sjenčanje fragmenata koji računa dobivenu boju je dan sljedećim isječkom koda.

```
  if (isReflective) {
    vec4 envColor = texture(gEnvironmentMap, reflectDir);
    color = mix(color, envColor, reflectFactor);
  }
```

Kako je ova tehnika dodana u Phongov model osvjetljenja potrebno je prvo provjeriti da li je materijal reflektivan varijablom `isReflective`. Zatim se uzorkuje mapa okoline te izmiješa boja korištenjem specifičnog faktora reflektivnosti za materijal. Konačan rezultat se može vidjeti na slici 3.5.

Tijekom implementacije pojavio se problem performansi. Iako je implementacija rađena na prijenosnom računalu, napravljeni program je radio neočekivano sporo. Nakon mnogo vremena uloženog u potragu onoga što toliko usporava rad te isprobavanju mnogo alata za analizu, rješenje je pokazao alat Apitrace. Na grafičkom prikazu ana-



Slika 3.5: Lijevo je prikaz refleksije a desno refrakcije

lize se vidio gubitak vremena između rada procesora i grafičke kartice. Problem je bio u tome što je dolazilo do sinkronizacije između njih a razlog je bilo čitanje pozicija geometrije zapisane u teksturama kako bi se mogle pretvoriti u format za generiranje kaustike. Taj problem je riješen tako što je napravljena varijanta komunikacije s dvostrukim spremnikom (engl. *double buffering*). Ideja je da se asinkrono prenesu podaci na procesor, dok se za potrebe generiranja kaustike šalju podaci generirani iz prethodnog prijenosa. Na taj način prvih nekoliko sličica nema efekt kaustike a kasnije taj efekt kasni za stvarnim stanjem scene. Kašnjenje je neprimjetno za ljudsko oko a dobivena brzina izvršavanja je dvostruka u odnosu na početno. Skok s početnih 30 sličica u sekundi na 60 je itekako vidljiv i pruža mnogo ugodniju interaktivnost. Rezultati mjerenja alatom Apitrace su vidljivi na slici 3.6. Na vrhu prikaza tog alata nalazi se grafički prikaz vremena koje troše CPU (plavom bojom) i GPU (crvenom bojom). Za GPU je sadržano više mjerenja za pojedine programe sjenčanja. Tako se na gornjoj slici vidi velika praznina između kraja rada grafičke kartice i početka rada procesora. Na donjoj slici se također može primijetiti kako metoda mapiranja kaustike ne uzima puno resursa na grafičkom procesoru, a većina vremena otpada na procesor i njegovo slanje geometrije grafičkoj kartici.

Slika 3.6: Gornja slika prikazuje program prije optimizacije, donja poslije

Sama izvedba u programskom kodu je relativno jednostavna te se nalazi unutar metode `CausticMap::WriteCausticMap`. Varijabla `frame_delay` određuje koliko će mapiranje kaustike kasniti za stanjem scene ali toliko vremena više ostavljamo komunikaciji između grafičke kartice i procesora računala. Funkcija `readFrame` asinkrono čita podatke te do blokiranja dolazi samo ako ju se ponovo pozove prije no što je dovršila prijenos. Funkcija `transfer` prilagodi pročitane podatke za generiranje mape kaustike a funkcija `drawCausticMap` izvrši to generiranje s trenutno dostupnim podacima. Ovisno o karakteristikama računala treba izmijeniti `frame_delay` a još bolje rješenje bi bilo koristiti tri spremnika (engl. *triple buffering*).

Ovako implementiran program je uspoređen s drugom metodom u nastavku. Upute za instalaciju i korištenje su sadržane u privitku ovog rada.

```
void CausticMap::WriteCausticMap()
{
  static int frame_delay = 2;
  static int frame = 0;
  static unsigned count = 0;
  // Wait couple of frames while the pixels are read.
  if (frame == 0) {
    readFrame(refr_frame_, width_, height_,
              vbo_[POS_DATA], vbo_[NORMAL_DATA]);
  }
  if (frame == frame_delay) {
    count = transfer(vbo_[POS_DATA], vbo_[NORMAL_DATA],
                     width_ * height_, vbo_[VERTEX_DATA]);
  }
  drawCausticMap(caustic_frame_, vao_, vbo_[VERTEX_DATA], count);
  frame = (frame + 1) % (frame_delay + 1);
}
```

3.2. Usporedba mapiranja kaustike i praćenja fotona

Nvidia OptiX je programsko sučelje namijenjeno izradi programa koji koriste algoritam praćenja zrake. Za te potrebe nudi funkcije za lako detektiranje sudara, rada s zrakama i prijenos izračuna na grafički procesor. Ako je grafički procesor također proizvela Nvidia te ovisno o modelu, OptiX u pozadini koristi sučelje CUDA što nudi dodatno poboljšanje performansi. Algoritam kojim je ostvaren efekt kaustike je praćenje fotona koji odlično paše uz ovaj API jer ja zasnovan na algoritmu praćenja zrake.

Program čija je implementacija prethodno opisana i program koji koristi Nvidia OptiX su uspoređeni na prijenosnom računalu sljedećih karakteristika.

- Intel(R) Core(TM) i3-4030U CPU @ 1.90GHz
- Nvidia GeForce 820M
- 4GB RAM

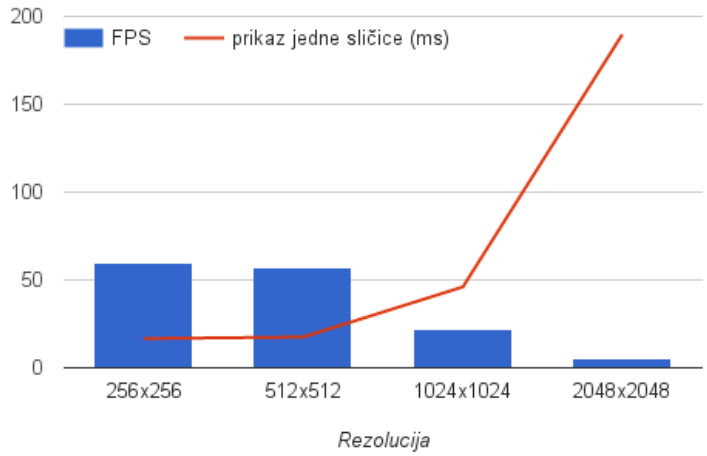
Kako navedena grafička kartica ima podršku za CUDA tehnologiju, mogu se očekivati dobri rezultati OptiX izvedbe metode praćenja fotona.

Algoritam mapiranja kaustike se izvodi u 60 sličica u sekundi (engl. *Frames Per Second, FPS*), odnosno za prikaz jedne sličice potrebno je oko 16.6ms. Rezolucija tekstura u kojoj se pohranjuju pozicije geometrije i rezultat izračuna efekta kaustike je

256x256 slikovnih elemenata (engl. *pixel*). Povećanjem rezolucije smanjuje se efekt aliasinga ali i pad brzine izvođenja. Omjer performansi i rezolucije dan je tablicom 3.1 i grafikonom 3.7. Razlike u prikazu se mogu vidjeti na slici 3.9. Cijena velike teksture skače naglo kada ona počne zauzimati toliko memorije na grafičkom procesoru da manipuliranje njome postane vrlo sporo. Ono što povećanje rezolucije ne rješava je problem preciznosti. Na dobivenoj slici 3.9 se može vidjeti pogrešno projicirana kaustika jer su presjecišta krivo procijenjena.

Rezolucija teksture (px)	FPS	prikaz jedne sličice (ms)
256x256	60.00	16.67
512x512	57.00	17.54
1024x1024	21.71	46.05
2048x2048	5.27	189.76

Tablica 3.1: Utjecaj rezolucije tekstura za mapiranje kaustike na performanse

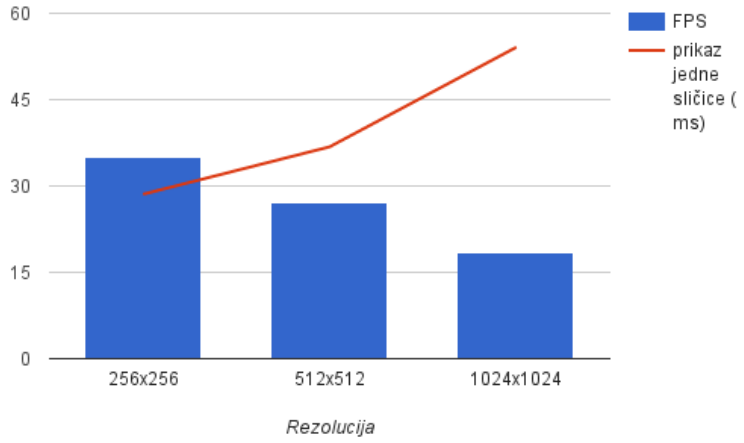


Slika 3.7: Utjecaj rezolucije teksture za mapiranje kaustike na performanse

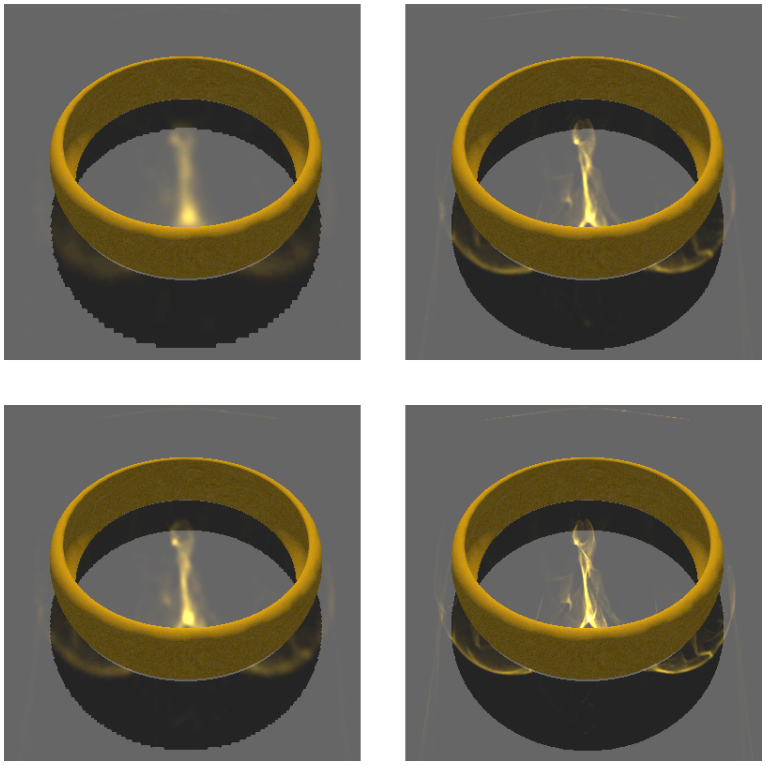
Kod algoritma praćenja fotona performanse su nešto slabije, ali iznenađujuće ne puno. Za potrebe prikaza jedne sličice potrebno je oko 50ms što odgovara izvođenju od 20 sličica u sekundi. Iako je brzina prikaza skoro 5 puta sporija, početak interaktivnosti ove metode se nazire. Glavna prednost ove metode je njena precizna i odlična kvaliteta generirane slike 3.10. Smanjenje rezolucije uzrokuje pojavu aliasinga na kompletnoj slici što ne pogoduje dobrom podešavanju i kompromisu omjera kvalitete i performansi. Utjecaj raznih rezolucija na brzinu izvođenja dan je tablicom 3.2 i grafikonom 3.8.

Rezolucija slike (px)	FPS	prikaz jedne sličice (ms)
256x256	35.00	28.57
512x512	27.16	36.81
1024x1024	18.48	54.11

Tablica 3.2: Utjecaj rezolucije slike kod praćenja fotona na performanse



Slika 3.8: Utjecaj rezolucije slike na performanse kod praćenja fotona



Slika 3.9: Mapiranje kaustike pri različitim rezolucijama teksture. Gore lijevo – 256x256; dolje lijevo – 512x512; gore desno – 1024x1024; dolje desno – 2048x2048

Slika 3.10: Rezultat dobiven praćenjem fotona i korištenjem tehnologija OptiX i CUDA. Lijeva slika – 256x256; srednja – 512x512; desna slika – 1024x1024

4. Zaključak

Vizualni učinak kaustike nije izrazito prominentan vizualni efekt ali ga je vrlo teško prikazati unutar interaktivnih zahtjeva. Opisani algoritmi su jednostavni ali težina proizlazi iz biranja dobrog omjera performansi i kvalitete prikaza. Metode namijenjene izvođenju u stvarnom vremenu poput mapiranja kaustike imaju veliku manu u nepreciznom prikazivanju efekta. Potrebno je pažljivo birati za koju geometriju će se iscrtati taj vizualni učinak kako bi ovakvi problemi ostali skriveni od korisnika. Veliko iznenađenje u interaktivnom prikazu je algoritam praćenja fotona. Izvorno korišten u iscrtavanju koje nema ovako visoke zahtjeve u brzini izvođenja, prilagodbom za moderne tehnologije i grafičke procesore počinje ispunjavati uprave te zahtjeve. Trenutno je puno lakše kontrolirati potrebne računalne resurse kod metoda izvorno namijenjenih interaktivnom korištenju ali samo je pitanje vremena kada će grafički procesori biti dovoljno snažni izvršavati algoritme poput praćenja fotona i praćenja zrake unutar tih uvjeta. Tako bi gotovo svi efekti poput refleksije i refrakcije bili iscrtani jednim algoritmom te bi konačan prikaz izgledao kvalitetnije.

LITERATURA

Tomas Akenine-Moller, Eric Haines, i Naty Hoffman. *Real-Time Rendering, Third Edition*. A K Peters, Ltd., 2008.

James Arvo. Backward ray tracing. U *ACM SIGGRAPH '86 Course Notes - Developments in Ray Tracing*, stranice 259–263, 1986.

Franklin C. Crow. Shadow Algorithms for Computer Graphics. U *SIGGRAPH '77 Proceedings*, 1977.

Manfred Ernst, Tomas Akenine-Möller, i Henrik Wann Jensen. Interactive rendering of caustics using interpolated warped volumes. *Graphics Interface*, 2005.

Henrik Wann Jensen. Global illumination using photon maps. U *Rendering Techniques*, 1996.

Jens Krüger, Kai Bürger, i Rüdiger Westermann. Interactive screen-space accurate photon tracing on gpus. *Eurographics Symposium on Rendering*, 2006.

Musawir Shah, Jaakko Konttinen, i Sumanta Pattanaik. Caustics mapping: An image-space technique for real-time caustics. *IEEE Transactions on Visualization and Computer Graphics*, 2007.

Mark Watt. Ligh-water interaction using backward beam tracing. U *Computer Graphics*, 1990.

David Wolff. *OpenGL 4.0 Shading Language Cookbook*. Packt Publishing, 2011.

Chris Wyman. Hierarchical caustic maps. *ACM Symposium on Interactive 3D Graphics and Games*, 2008.

Ostvarivanje vizualnog učinka kaustike u stvarnom vremenu

Sažetak

Napredak brzine obrade podataka na računalu direktno utječe na kvalitetu računalne grafike koja se može ostvariti u interaktivnim aplikacijama. Mnogi zanimljivi, ali kompleksni vizualni efekti se sve više mogu ostvariti unutar zahtjeva interaktivnosti. Jedan od takvih efekata je kaustika. Učinak kaustike nastaje kada se svjetlosne zrake zbog refrakcije ili refleksije fokusiraju na manji dio površine. Cilj rada je usporedba različitih metoda ostvarivanja tog vizualnog učinka na računalu u stvarnom vremenu. Opisana je implementacija tehnike mapiranje kaustike u programskom jeziku C++ korištenjem biblioteke OpenGL. Osim te implementacije, za demonstraciju i usporedbu korištena je i tehnologija Nvidia OptiX za ostvarenje učinka kaustike tehnikom praćenja zrake.

Ključne riječi: svjetlost, kaustika, tehnika mapiranja kaustike, praćenje fotona, praćenje zrake, iscrtavanje u stvarnom vremenu

Real time caustics rendering

Abstract

Advancement of computer data processing speed directly influences the quality of computer graphics which can be realised for interactive software. Many interesting, albeit complex visual effects can increasingly be implemented within interactive constraints. One such effect is caustics. The visual effect of caustics appears when light concentrates on a smaller part of a surface due to reflection and refraction. The goal of this thesis is to compare the different methods for rendering of this visual effect in real time. This paper describes the implementation of caustic mapping technique using C++ programming language and OpenGL library. Alongside the described implementation and for purposes of comparison, the Nvidia OptiX technology was used to demonstrate the rendering of caustics using ray tracing algorithm.

Keywords: light, caustics, caustic mapping technique, photon tracing, ray tracing, real time rendering

5. Privitak

Kao dio ovog rada sadržan je i programski kod implementacije algoritma mapiranja kaustike. On se nalazi u direktoriju `caustic-app` . Upute za instalaciju i korištenje programa su u nastavku.

5.1. Upute za instalaciju programa

Za instalaciju su potrebne sljedeće biblioteke i programi:

- CMake
- OpenGL >= 4.0
- freeglut
- GLEW
- glm
- FreeImage

Na Linux operativnim sustavima dovoljno je pokrenuti sljedeće.

```
mkdir caustic-app/build
cd caustic-app/build
cmake ..
make
```

Naredba `cmake` će javiti ako neka od navedenih biblioteka nedostaje.

Kod instalacije programa na Windowsima pokrenite program CMake-GUI te namjestite putanje do direktorija s izvornim kodom i do direktorija gdje želite generirati projektne datoteke. Odaberite koje projektne datoteke želite stvoriti te stisnite Configure. CMake će javiti da morate postaviti putanje do biblioteka. Svaki put kada postavite putanju pritisnite ponovo Configure i ponovite postupak dok sve biblioteke nisu postavljene. Pritiskom na Generate generira se željeni projekt koji otvorite s pripadajućim programom i pokrenite prevođenje programa.

5.2. Upute za korištenje

Izvršna datoteka se pokreće naredbom:
`caustic-app [<putanja_do_modela> <rezolucija_teksture>]`
Naredbu je potrebno pokrenuti iz direktorija u kojem se nalaze direktoriji `models` i `shaders`. Aplikaciju je moguće pokrenuti bez argumenata ili predati oba. Ako želite promijeniti model koji uzrokuje pojavu kaustike koristite argument `putanja_do_modela`. Promjena rezolucije tekstura koje se koriste za izračune kaustike se predaje kao argument `rezolucija_teksture`. Nakon pokretanja aplikacije dostupne su sljedeće kontrole.

- Okretanje kamere tipkama: w, a, s, d, c i tipkom za razmak.
- Paljenje/gašenje Gaussovog filtra tipkom b.
- Paljenje/gašenje prikaza zapisanih normala refraktivne geometrije tipkom N.
- Paljenje/gašenje prikaza zapisanih pozicija refraktivne geometrije tipkom R.
- Paljenje/gašenje prikaza zapisanih pozicija geometrije koja prima učinak kaustike tipkom P
- Paljenje/gašenje prikaza teksture kaustike tipkom C.
- Gašenje svih gore navedenih prikaza tipkom n.
- Gašenje aplikacije tipkom q. Aplikacija se također može ugasiti pritiskom na X prozora.