Ubrzanje algoritma praćenja zraka intrinsičnim funkcijama

Mato Gudelj
Voditelj: prof. dr. sc. Siniša Šegvić

2. veljače 2022.

1 Uvod

Praćenje zraka (*engl.* raytracing) je računalni algoritam za prikaz realističnih 3D slika. Dvije najvažnije komponente algoritma su **zrake** i **objekti**. Svaka zraka je opisana jednadžbom:

$$\vec{r}(t) = \vec{o} + t\vec{d}.$$

Algoritam šalje zrake naprijed s mjesta $\vec{o}$ na kojem se nalazi promatrač prema virtualnim pixelima (x, y). Smjer te zrake možemo zapisati kao: $\vec{d} = \vec{f}(x, y)$. Objekti u sceni su opisani baznim geometrijskim oblicima poput kugli i ravnina. Ako zraka ima dodirnu točku s objektom, algoritam zaključuje da je objekt vidljiv. Postojanje dodirne točke se određuje rješavanjem jednadžbe dodira za parametar t sa svakim objektom u sceni. Na primjer, ako je objekt ravnina:

$$(\vec{p} - \vec{p_0}) \cdot \vec{n} = 0$$

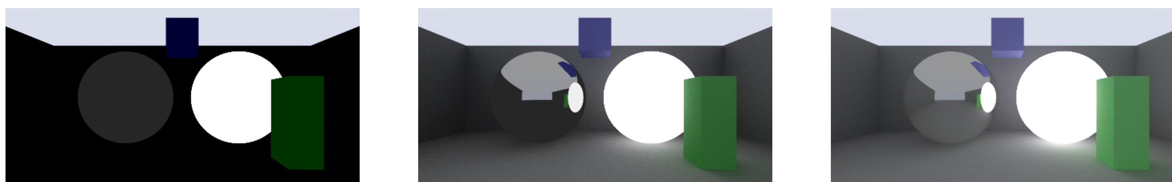
izračun dodirne točke se provodi određivanjem t_0 za koji se točka $\vec{r}(t_0)$ nalazi na ravnini, odnosno:

$$((\vec{o} + t\vec{d}) - \vec{p_0}) \cdot \vec{n} = 0 \implies t = \frac{(\vec{p_0} - \vec{o_0}) \cdot \vec{n}}{\vec{d} \cdot \vec{n}}$$

Provjerom granica izračunate dodirne točke vrlo jednostavno možemo dobiti i pravokutnike, a samim time i kvadre (svaka strana po jedan pravokutnik). Istim postupkom možemo dobiti kugle koristeći jednadžbu $(\vec{p} - \vec{c_0})(\vec{p} - \vec{c_0}) = r^2$ i pronalaskom odgovarajućeg parametra t.

Ako postoji dodirna točka s objektom, algoritam se nastavlja stvaranjem **nove zrake iz mjesta dodira**. Smjer te zrake ovisi o fizikalnim svojstvima objekta. Na primjer, kod površina koje simuliraju zrcalo smjer se određuje refleksijom (ulazni kut α je jednak izlaznom kutu β), uz mogućnost dodatka šuma ovisno o "glatkosti" zrcala. Boja virtualnog pixela ovisi o objektima koje je zraka susrela (uključujući i one nakon odbijanja), uz faktor slabljenja nakon svakog odbijanja. Zbog elementa slučajnosti svaki virtualni pixel se uzorkuje više puta i rezultantna boja je njihov prosjek.

Kako se svaki pixel uzorkuje više puta i za svaki uzorak se računa veliki broj jednadžbi dodira, algoritam je vrlo računalno intenzivan. Nezavisnost primitivnih objekata u sceni čini problem prikladnim za paralelizaciju. Tema ovog projekta je ubrzanje izračuna jednadžbi dodira koristeći **SIMD operacije nad grupama**. Projekt također prikazuje karakteristične probleme s priručnom memorijom do kojih dolazi zbog velikog broja objekata i njihovom naizmjeničnom pristupu.



(a) $n = 0$ (b) $n = 1$ (c) $n = 10$

Slika 1: Izlaz u ovisnosti o maks. broju odbijanja, n

2 Izračun jednadžbe dodira kugle

Rješenje jednadžbe dodira kugle se svodi na rješenje odgovarajuće kvadratne jednadžbe:

$$
\begin{aligned}
&(\vec{p} - \vec{c_0})(\vec{p} - \vec{c_0}) = r^2 \\
&\|\vec{p}\|^2 - 2(\vec{p} \cdot \vec{c_0}) + \|\vec{c_0}\|^2 - r^2 = 0 \\
&\|(\vec{o} + t\vec{d})\|^2 - 2((\vec{o} + t\vec{d}) \cdot \vec{c_0}) + \|\vec{c_0}\|^2 - r^2 = 0 \\
&\|\vec{d}\|^2 t^2 + 2\vec{d} \cdot (\vec{o} - \vec{c_0})t + (\vec{o} - \vec{c_0})^2 - r^2 = 0
\end{aligned}
$$

Dobili smo kvadratnu jednadžbu oblika $at^2 + bt + c = 0$ za parametre:

$$
\begin{aligned}
a &= \|\vec{d}\|^2 \\
b &= 2\vec{d} \cdot (\vec{o} - \vec{c_0}) \\
c &= \|\vec{o} - \vec{c_0}\|^2 - r^2
\end{aligned}
$$

Rješenje te jednadžbe vrlo jednostavno možemo prevesti u C++ kod:

```
1  // (napomena: * operator je overloadan za vec3 tipove i predstavlja skalarni umnožak)
2  inline float Sphere::hit(const ray& r) const {
3      vec3 oc = r.orig - this->center;
4      float a = r.dir * r.dir;
5      float b = 2 * r.dir * oc;
6      float c = oc*oc - this->radius * this->radius;
7
8      float disc = (b*b - 4*a*c);
9
10     if (disc < 0) return -1.f;
11     return (-b - sqrt(disc))/(2*a);
12 }
```

Iako je izračun poprilično jednostavan, on se provodi *više milijardi* puta za svaku kuglu u prikazu scene konvencionalne veličine 1920x1080 (1000 uzoraka/px). No kako se isti niz instrukcija provodi nad svim kuglama izračun možemo znatno ubrzati koristeći instrukcije **SIMD** (*engl.* Single instruction, multiple data).

3 Naivno ubrzanje jednadžbe dodira kugle

Za implementaciju ubrzanog rješenja odabran je skup instrukcija SIMD obrađen na laboratorijskim vježbama - **instrukcije SSE** za jednostruku preciznost. Kratak opis korištenih funkcija se može pronaći u dodatku A. Dodatno ubrzanje bi se moglo postići koristeći novije instrukcije AVX koje podržavaju operacije nad 8 brojeva jednostruke preciznosti (ukupno 256b) umjesto SSE-ovih 4, no to je izvan opsega ovog projekta.

Naivnu implementaciju provodimo jednostavnim prijevodom operacija gore navedenog koda u instrukcije SSE. Funkcija je potpisa:

```
1  void inline check_sphere_SSE(const std::vector<std::shared_ptr<Sphere>>& sphs, const
   ↪  ray& r, float& t_last, std::shared_ptr<Object>& o_last) {
```

Gdje *o_last* pokazuje na trenutno najbliži objekt koji je na putu zrake, a *t_last* onaj parametar t koji daje točku njihovog presjecišta u jednadžbi $\vec{r}(t) = \vec{o} + t\vec{d}$. Prvo učitavamo komponente vektora zrake $\vec{o}$ i $\vec{d}$ u zasebne registre:

```
2      __m128 ray_ox = _mm_load_ps1(&r.orig.e[0]);
3      __m128 ray_oy = _mm_load_ps1(&r.orig.e[1]);
4      __m128 ray_oz = _mm_load_ps1(&r.orig.e[2]);
5
```

```
6      __m128 ray_dx = _mm_load_ps1(&r.dir.e[0]);
7      __m128 ray_dy = _mm_load_ps1(&r.dir.e[1]);
8      __m128 ray_dz = _mm_load_ps1(&r.dir.e[2]);
```

Zatim računamo kvadrat duljine vektora $\vec{d}$ (koeficijent a rješenja kvadratne jednadžbe), i upisujemo korisne konstante u XMM registre:

```
9      __m128 a = _mm_set_ps1(r.dir * r.dir);
10     __m128 inv2a = _mm_set_ps1(1.f / (-2 * r.dir * r.dir));
11
12     const __m128 two_const = _mm_set_ps1(2.0f);
13     const __m128 neg_two_const = _mm_set_ps1(-2.0f);
14     const __m128 four_const = _mm_set_ps1(4.0f);
```

Započinjemo petljom koja prolazi kroz vektor kugli 4 po 4 i učitavamo 4 odgovarajuća polumjera u jedan XMM registar:

```
15     if (sphs.size() > 3) {
16         for (auto i = 3; i < sphs.size(); i += 4) {
17             __m128 sph_r = _mm_set_ps(sphs[i]->radius, sphs[i-1]->radius,
               ↪  sphs[i-2]->radius, sphs[i-3]->radius);
```

Na sličan način učitavamo i komponente $\vec{o} - \vec{c}$ vektora 4 kugli u 3 XMM registra - prvo učitamo vrijednosti komponente pozicije kugli $\vec{c}$, a zatim te vrijednosti oduzmemo od registara koje smo učitali na L2-4:

```
17             __m128 oc_x = _mm_set_ps(sphs[i]->center.e[0], sphs[i-1]->center.e[0],
               ↪  sphs[i-2]->center.e[0], sphs[i-3]->center.e[0]);
18                 oc_x = _mm_sub_ps(ray_ox, oc_x);
19             __m128 oc_y = _mm_set_ps(sphs[i]->center.e[1], sphs[i-1]->center.e[1],
               ↪  sphs[i-2]->center.e[1], sphs[i-3]->center.e[1]);
20                 oc_y = _mm_sub_ps(ray_oy, oc_y);
21             __m128 oc_z = _mm_set_ps(sphs[i]->center.e[2], sphs[i-1]->center.e[2],
               ↪  sphs[i-2]->center.e[2], sphs[i-3]->center.e[2]);
22                 oc_z = _mm_sub_ps(ray_oz, oc_z);
```

Sada možemo izračunati ostatak koeficijenata, diskriminantu i t-vrijednosti:

```
23             __m128 b = _mm_mul_ps(mul_vec3(oc_x, oc_y, oc_z, ray_dx, ray_dy,
               ↪  ray_dz), two_const);
24             __m128 c = _mm_sub_ps(vec3_sq(oc_x, oc_y, oc_z), _mm_mul_ps(sph_r,
               ↪  sph_r));
25
26             __m128 disc = _mm_sub_ps(_mm_mul_ps(b, b), _mm_mul_ps(_mm_mul_ps(a, c),
               ↪  four_const));
27             __m128 sqrt_disc = _mm_sqrt_ps(disc);
28
29             __m128 t = _mm_mul_ps(_mm_add_ps(sqrt_disc, b), inv2a); // (-b -
               ↪  sqrt(disc))/(2*a)
```

Napokon slijedno iteriramo po dobivenom registru t i ažuriramo rješenja:

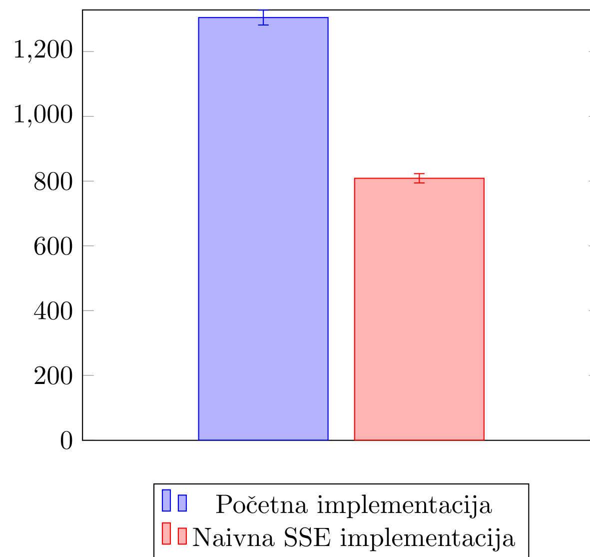
```
30             for (auto j = 0; j < 4; ++j) {
31                 if (t[j] > 0 && t[j] < t_last) {
32                     t_last = t[j];
33                     o_last = sphs[i - 3 + j];
34                 }
35             }
36         }
37     }
```

Preostalo je još provjeriti ostatak kugli na repu vektora (maks. 3 jer provjeravamo 4 po 4), što radimo na "normalan" način:

```
38      for (auto i = sphs.size() - sphs.size()%4; i < sphs.size(); ++i) {
39          auto t = sphs[i]->hit(r);
40          if (t > 0 && t < t_last) {
41              t_last = t;
42              o_last = sphs[i];
43          }
44      }
45  }
```

Na uzorku od $n = 100$ mjerenja u sceni s 250 kugli dobivamo sljedeće rezultate:

Početna implementacija: $1305ms \pm 23ms$ Naivna SSE implementacija: $808.7ms \pm 14.3ms$



4 Ubrzanje jednadžbe dodira kugle pomoćnim strukturama

Rezultati poglavlja 3 su donekle razočaravajući - SSE implementaciji treba 62% vremena početne implementacije. Zbog izvođenja operacija nad 4 kugle od jednom očekivali bi da je rezultat bliži 25%. Bližim razmatranjem koda možemo uočiti da je vrlo vjerojatno krivac intrinsična funkcija *_mm_set_ps* koja se koristi za učitavanje pozicija i polumjera kugli u XMM registre. To možemo potvrditi ako pogledamo generiran x86-asm kod za jedan takav poziv.

Prvo se svaki element XMM registra učita iz memorije i stavlja na stog. To se ostvaruje izvođenjem sljedećeg odsječka za **svaki element** (4 puta po pozivu):

```
call        _ZNKSt6vectorISt10shared_ptrI6SphereESaIS2_EEixEm
mov           rdi, rax
call        _ZNKSt19__shared_ptr_accessI6SphereLN9__gnu_cxx12_Lock_policyE2ELb0ELb0EEptEv
movss        xmm0, dword ptr [rax + 44]
movss        dword ptr [rbp - 1312], xmm0
mov           rdi, qword ptr [rbp - 984]
mov           eax, dword ptr [rbp - 1124]
add           eax, -1
movsxd       rsi, eax
```

Zatim se te vrijednosti preslože i napokon učitaju zajedno u jedan XMM registar:

```
movss        xmm3, dword ptr [rbp - 1312]
movss        xmm2, dword ptr [rbp - 1308]
movss        xmm1, dword ptr [rbp - 1304]
movss        xmm0, dword ptr [rbp - 1300]
movss        dword ptr [rbp - 644], xmm3
movss        dword ptr [rbp - 648], xmm2
movss        dword ptr [rbp - 652], xmm1
movss        dword ptr [rbp - 656], xmm0
movss        xmm0, dword ptr [rbp - 644]
movss        xmm1, dword ptr [rbp - 648]
unpcklps        xmm1, xmm0
movss        xmm2, dword ptr [rbp - 652]
movss        xmm0, dword ptr [rbp - 656]
unpcklps        xmm0, xmm2
unpcklpd        xmm0, xmm1
movapd        xmmword ptr [rbp - 672], xmm0
movaps        xmm0, xmmword ptr [rbp - 672]
```

Vidimo da je takav poziv zaslužan za preko 50 instrukcija. Implementacija koristi 4 poziva *_mm_set_ps* u glavnoj petlji koja se izvodi više stotina milijuna puta za izračun tipične scene. Neki problemi koda su jednostavni i može ih riješiti optimizator (npr. preslagivanje), no postoje 2 glavna problema za koje je potrebno promijeniti samu strukturu programa.

Prvi problem je činjenica da intrinsična funkcija *_mm_set_ps* generira dug slijed instrukcija. U idealnom slučaju bi koristili jednoinstrukcijsku intrinsičnu funkciju poput *_mm_load_ps* koja direktno učitava 4 slijedne lokacije u XMM registar. U novijim Intelovim arhitekturama[1] *_mm_load_ps* ima CPI manji od 1.

Drugi problem proizlazi iz trenutnog načina pristupa memoriji koji krši **pretpostavku prostorne lokalnosti pristupa** (*engl.* spatial locality). Prilikom učitavanja elemenata registra XMM, pristupamo adresama koje su međusobno udaljene više od tipične širine linije priručne memorije (64 B). Kugle su predstavljene kao objekti a oni zauzimaju slijedni odsječak memorije. Zbog toga dohvaćena linija neće sadržavati atribute iste vrste nego atribute tog istog objekta, što rezultira promašajima prilikom učitavanja preostala 3 elementa XMM registra.

Problemi su očito povezani. Oba problema možemo riješiti koristeći **pomoćne strukture**. Organizacija komponenti iste vrste u zasebne strukture je karakteristika podatkovno orijentiranog dizajna (*engl.* Data-oriented design).[2]

U klasi scene stvaramo zasebne vektore. Po jedan za svaku vrstu atributa:

```
39    std::vector<std::shared_ptr<Rectangle>> rectangles;
40    std::vector<std::shared_ptr<Sphere>> spheres;
41    std::vector<std::shared_ptr<Cuboid>> cuboids;
42    std::vector<std::shared_ptr<Mandelbulb>> mandelbulbs;
43    int n_bounces;
44    float fog_factor;
45    gradient sky;
46
47    // SSE acceleration helpers
48    #ifdef USE_SSE
49    alignas(16) std::vector<float> radii;
50    alignas(16) std::vector<float> c0;
51    alignas(16) std::vector<float> c1;
52    alignas(16) std::vector<float> c2;
53    #endif
```

[1] Počevši s arhitekturom *Haswell*

[2] Data-oriented design — Wikipedia

Vektore punimo u funkciji za dodavanje nove kugle:

```
39  inline void Scene::add_sphere(const point3& center, const float& radius, const
    ↪  Props& props) {
40      spheres.push_back(std::make_shared<Sphere>(center, radius, props));
41
42      #ifdef USE_SSE
43      radii.push_back(radius);
44      c0.push_back(center.e[0]);
45      c1.push_back(center.e[1]);
46      c2.push_back(center.e[2]);
47      #endif
48  }
```

I dodamo prosljeđivanje pomoćnih struktura u SSE funkciju:

```
1   void inline check_sphere_SSE(const std::vector<std::shared_ptr<Sphere>>& sphs, const
    ↪  ray& r, float& t_last, std::shared_ptr<Object>& o_last, const
    ↪  std::vector<float>& radii, const std::vector<float>& c0, const
    ↪  std::vector<float>& c1, const std::vector<float>& c2) {
```

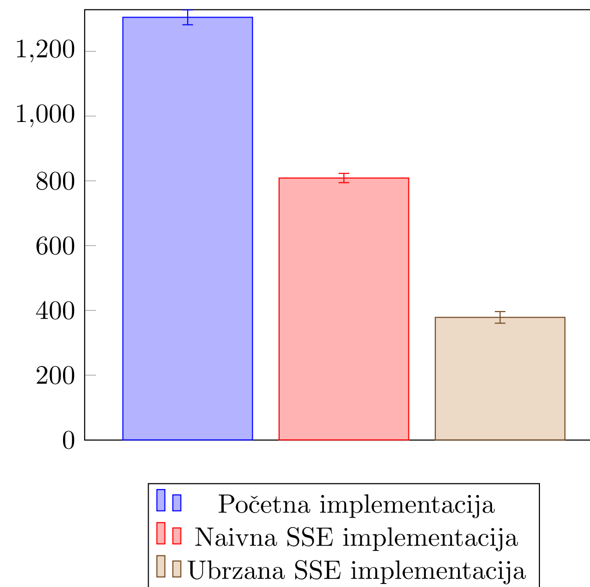
Napokon možemo zamijeniti *_mm_set_ps* intrinsične funkcije s *_mm_load_ps*.

```
15          for (auto i = 0; i < sphs.size()-3; i += 4) {
16              __m128 sph_r = _mm_load_ps(radii.data() + i);
17              __m128 oc_x = _mm_load_ps(c0.data() + i);
18                     oc_x = _mm_sub_ps(ray_ox, oc_x);
19              __m128 oc_y = _mm_load_ps(c1.data() + i);
20                     oc_y = _mm_sub_ps(ray_oy, oc_y);
21              __m128 oc_z = _mm_load_ps(c2.data() + i);
22                     oc_z = _mm_sub_ps(ray_oz, oc_z);
```

Na uzorku od $n = 100$ mjerenja u sceni s 250 kugli dobivamo sljedeće rezultate:

Početna implementacija: $1305ms \pm 23ms$ Naivna SSE implementacija: $808.7ms \pm 14.3ms$

Ubrzana SSE implementacija: $378.4ms \pm 18.0ms$



Rezultat je puno bolji od naivne implementacije i 29% je poprilično blizu "idealnih" 25% vremena početne implementacije. Zbog dodatnih struktura program zauzima nešto više radne memorije ($16n_k$ B), no u ovom slučaju to ne radi primjetnu razliku.

5 Zaključak

SIMD instrukcije su dobar izbor za ubrzanje računanja jednadžbi dodira. SSE implementaciji iz poglavlja 4 treba 29% vremena početne implementacije. To je blizu teoretskih 25% za obradu 4 kugle istovremeno. Iz rezultata poglavlja 3 možemo vidjeti da i direktan (naivan) prijevod funkcije za kugle u verziju s intrinsicima daje ubrzanje, no ono je daleko od brzine postignute malo pametnijom implementacijom. Važno je napomenuti da se može dogoditi da naivna SIMD implementacija neke funkcije bude *sporija* od početne implementacije. To je čest slučaj ako je izračun kratak, dok je "overhead" za učitavanje podataka u XMM registre visok (npr. ako podaci nisu slijedno u memoriji).

Literatura

[1] Peter Shirley. *Ray Tracing in One Weekend*. https://raytracing.github.io/books/RayTracingInOneWeekend.html. Prosinac 2020. URL: https://raytracing.github.io/books/RayTracingInOneWeekend.html.

[2] Wikibooks. *X86 Assembly/SSE — Wikibooks, The Free Textbook Project*. 2020. URL: https://en.wikibooks.org/w/index.php?title=X86_Assembly/SSE&oldid=3676280.

[3] Wikipedia contributors. *Line–sphere intersection — Wikipedia, The Free Encyclopedia*. 2021. URL: https://en.wikipedia.org/w/index.php?title=Line%E2%80%93sphere_intersection&oldid=1021628678.

[4] Wikipedia contributors. *Ray tracing (graphics) — Wikipedia, The Free Encyclopedia*. 2021. URL: https://en.wikipedia.org/w/index.php?title=Ray_tracing_(graphics)&oldid=1062288767.

[5] Intel. *Intel Intrinsics Guide*. URL: https://www.intel.com/content/www/us/en/docs/intrinsics-guide/index.html.

A Dodatak: Korištene intrinsične funkcije

U ovom poglavlju se nalazi popis korištenih intrinsičnih funkcija i kratak opis njihove funkcionalnosti.

_mm_load_ps1(x) Učitava vrijednost s lokacije x u sva 4 mjesta registra XMM.

_mm_set_ps1(x) Postavlja vrijednost x u sva 4 mjesta registra XMM.

_mm_load_ps(x) Učitava 4 slijedne vrijednosti počevši s lokacijom x u registar XMM.

_mm_set_ps(x1, x2, x3, x4) Postavlja vrijednost [x1,x2,x3,x4] u registar XMM.

_mm_add_ps(x, y) Zbraja pojedinačne elemente XMM registra x i XMM registra y.

_mm_sub_ps(x, y) Oduzima pojedinačne elemente XMM registra x i XMM registra y.

_mm_mul_ps(x, y) Množi pojedinačne elemente XMM registra x i XMM registra y.

_mm_div_ps(x, y) Dijeli pojedinačne elemente XMM registra x i XMM registra y.

_mm_sqrt_ps(x) Računa sqrt() nad svakim elementom XMM registra x.