Bericht Optimierung

Lukas Panni

Bericht Optimierung

1. Schnittpunkttest optimieren

Sourcecode optimierte Version:

```
bool intersects(Vector<T, 3> origin, Vector<T, 3> direction,
FLOAT &t, FLOAT &u, FLOAT &v, FLOAT minimum_t)
{
Vector<T, 3> normal = cross_product(p2 - p1, p3 - p1);

T normalRayProduct = normal.scalar_product(direction);

if (fabs(normalRayProduct) < EPSILON)
{
return false;
}

T d = normal.scalar_product(p1);
t = (d - normal.scalar_product(origin)) / normalRayProduct;

if (t < 0.0 || t > minimum_t)
{
return false;
}

Vector<T, 3> intersection = origin + t \* direction;

Vector<T, 3> vector = cross_product(p2 - p1, intersection - p1);
if (normal.scalar_product(vector) < 0.0)
{
return false;
}

vector = cross_product(p3 - p2, intersection - p2);
if (normal.scalar_product(vector) < 0.0)
```

```
{
return false;
}

Vector<T, 3> vectorV = cross_product(p1 - p3, intersection - p3);
if (normal.scalar_product(vectorV) < 0.0)
{
return false;
}

T squareArea = normal.square_of_length();

v = sqrt(vectorV.square_of_length() / squareArea);
u = sqrt(vector.square_of_length() / squareArea);
return true;
}
```

Die Änderungen beziehen sich insbesondere auf die Berechnung von u und v. Diese wurden ans Ende der Funktion verschoben, da die Werte nur benötigt werden, wenn ein Schnittpunkt existiert. Zur Reduktion der Wurzelfunktion wurde die Berechnung von u und v angepasst: Anstatt die Länge durch die Fläche zu teilen, wird das Quadrat der Länge durch das Quadrat der Fläche geteilt und im Anschluss die Wurzel gezogen. Da bei der Berechnung der Länge und der Fläche jeweils die Wurzel aus dem jeweiligen Quadrat gezogen wird, kann so die Anzahl der `sqrt` Operationen von 3 (2*Länge + Fläche) auf 2 (2*(QuadratischeLänge/QuadratischeFläche)) reduziert werden. Außerdem wird durch Prüfung, ob bereits ein näherer Schnittpunkt gefunden wird, ein früherer Abbruch der Funktion ermöglicht.

Assembler

Der Assembler-Code wurde ohne Compiler-Optimierung erzeugt, um die Lesbarkeit zu verbessern. Für die Ermöglichung des früheren Abbruchs (Überprüfung auf `t > minimum_t`) werden mehr Assembler-Befehle erzeugt. Der zweite Vergleich erfordert neben einer Vergleichsoperation (`VCOMISS`) auch einen zusätzlichen Sprungbefehl (`ja` und `jbe` im Vergleich zu nur `jbe`).

unoptimiert:

```
if (t < 0.0)
239: 48 8b 85 30 ff ff ff mov -0xd0(%rbp),%rax
240: c5 fa 10 08 vmovss (%rax),%xmm1
244: c5 f8 57 c0 vxorps %xmm0,%xmm0,%xmm0
248: c5 f8 2f c1 vcomiss %xmm1,%xmm0
24c: 76 0a jbe 258 <\
    ↪ _ZN8TriangleIfE10intersectsE6VectorIfLm3EES2_RfS3_S3_f+0
    ↪ x258>
```

```
{
return false;
24e: b8 00 00 00 00 mov $0x0,%eax
253: e9 1b 04 00 00 jmpq 673 <\
    ↪ _ZN8TriangleIfE10intersectsE6VectorIfLm3EES2_RfS3_S3_f+0
    ↪ x673>
}
```

optimiert:

```
if (t < 0.0 || t > minimum_t)
229: 48 8b 85 20 ff ff ff mov -0xe0(%rbp),%rax
230: c5 fa 10 08 vmovss (%rax),%xmm1
234: c5 f8 57 c0 vxorps %xmm0,%xmm0,%xmm0
238: c5 f8 2f c1 vcomiss %xmm1,%xmm0
23c: 77 15 ja 253 <\
    ↪ _ZN8TriangleIfE10intersectsE6VectorIfLm3EES2_RfS3_S3_f+0
    ↪ x253>
23e: 48 8b 85 20 ff ff ff mov -0xe0(%rbp),%rax
245: c5 fa 10 00 vmovss (%rax),%xmm0
249: c5 f8 2f 85 0c ff ff vcomiss -0xf4(%rbp),%xmm0
250: ff
251: 76 0a jbe 25d <\
    ↪ _ZN8TriangleIfE10intersectsE6VectorIfLm3EES2_RfS3_S3_f+0
    ↪ x25d>
{
return false;
253: b8 00 00 00 00 mov $0x0,%eax
258: e9 7a 04 00 00 jmpq 6d7 <\
    ↪ _ZN8TriangleIfE10intersectsE6VectorIfLm3EES2_RfS3_S3_f+0
    ↪ x6d7>
}
```

Die Optimierung der Berechnung von u und v führt zur Reduktion der Aufrufe der `sqrt` Implementierung der GLIBC. In der unoptimierten Variante wird die Fläche `area` als Länge des Normalenvektors berechnet:

```
T area = normal.length(); // used for u-v-parameter calculation
6392: 48 8d 85 7c ff ff ff lea -0x84(%rbp),%rax
6399: 48 89 c7 mov %rax,%rdi
639c: e8 ed fa ff ff callq 5e8e <\_ZNK6VectorIfLm3EE6lengthEv>
63a1: c5 f9 7e c0 vmovd %xmm0,%eax
63a5: 89 45 f8 mov %eax,-0x8(%rbp)
```

Die Berechnung der Länge erfolgt in der Funktion `Vector::length`:

```
T length(void) const {
5e8e: 55 push %rbp
5e8f: 48 89 e5 mov %rsp,%rbp
5e92: 48 83 ec 10 sub $0x10,%rsp
5e96: 48 89 7d f8 mov %rdi,-0x8(%rbp)
return sqrt( square_of_length() );
5e9a: 48 8b 45 f8 mov -0x8(%rbp),%rax
5e9e: 48 89 c7 mov %rax,%rdi
5ea1: e8 8a 11 00 00 callq 7030 <\
    ↪ _ZNK6VectorIfLm3EE16square_of_lengthEv>
5ea6: c5 f2 5a c8 vcvtss2sd %xmm0,%xmm1,%xmm1
5eaa: c4 e1 f9 7e c8 vmovq %xmm1,%rax
5eaf: c4 e1 f9 6e c0 vmovq %rax,%xmm0
5eb4: e8 47 c2 ff ff callq 2100 <sqrt@plt>
5eb9: c5 fb 5a c0 vcvtsd2ss %xmm0,%xmm0,%xmm0
}
```

Dazu wird zunächst `square_of_length()` aufgerufen (5ea1) und mit dem Ergebnis die GLIBC Implementierung der `sqrt` Funktion aufgerufen (5eb4). Auch für die Berechnung von `u` und `v` wird `Vector::length()` aufgerufen:

```
u = vector.length() / area;
6726: 48 8d 85 64 ff ff ff lea -0x9c(%rbp),%rax
672d: 48 89 c7 mov %rax,%rdi
6730: e8 59 f7 ff ff callq 5e8e <\_ZNK6VectorIfLm3EE6lengthEv>
6735: c5 fa 5e 45 f8 vdivss -0x8(%rbp),%xmm0,%xmm0
673a: 48 8b 85 28 ff ff ff mov -0xd8(%rbp),%rax
6741: c5 fa 11 00 vmovss %xmm0,(%rax)
...
v = vector.length() / area;
685b: 48 8d 85 64 ff ff ff lea -0x9c(%rbp),%rax
6862: 48 89 c7 mov %rax,%rdi
6865: e8 24 f6 ff ff callq 5e8e <\_ZNK6VectorIfLm3EE6lengthEv>
686a: c5 fa 5e 45 f8 vdivss -0x8(%rbp),%xmm0,%xmm0
686f: 48 8b 85 20 ff ff ff mov -0xe0(%rbp),%rax
6876: c5 fa 11 00 vmovss %xmm0,(%rax)
```

Insgesamt macht das 3 Aufrufe von `sqrt` (67830 und 6865 über `Vector::length`). Die optimierte Version verwendet die `Vector::length` Funktion nicht. Da für die Berechnung von u und v standardmäßig jeweils zwei Quadratwurzeln dividiert werden, kann die Wurzelberechnung auch auf das Ergebnis der Division erfolgen und damit verzögert werden. In der optimierten Version werden deshalb nur 2 Aufrufe von `sqrt` benötigt (6876 und 68ac).

```
v = sqrt(vectorV.square_of_length() / squareArea);
```

```
6854: 48 8d 85 58 ff ff ff lea -0xa8(%rbp),%rax
685b: 48 89 c7 mov %rax,%rdi
685e: e8 13 08 00 00 callq 7076 <\
    ↪ _ZNK6VectorIfLm3EE16square_of_lengthEv>
6863: c5 fa 5e 45 f4 vdivss -0xc(%rbp),%xmm0,%xmm0
6868: c5 d2 5a e8 vcvtss2sd %xmm0,%xmm5,%xmm5
686c: c4 e1 f9 7e e8 vmovq %xmm5,%rax
6871: c4 e1 f9 6e c0 vmovq %rax,%xmm0
6876: e8 85 b8 ff ff callq 2100 <sqrt@plt>
687b: c5 fb 5a c0 vcvtsd2ss %xmm0,%xmm0,%xmm0
687f: 48 8b 85 10 ff ff ff mov -0xf0(%rbp),%rax
6886: c5 fa 11 00 vmovss %xmm0,(%rax)
u = sqrt(vector.square_of_length() / squareArea);
688a: 48 8d 85 64 ff ff ff lea -0x9c(%rbp),%rax
6891: 48 89 c7 mov %rax,%rdi
6894: e8 dd 07 00 00 callq 7076 <\
    ↪ _ZNK6VectorIfLm3EE16square_of_lengthEv>
6899: c5 fa 5e 45 f4 vdivss -0xc(%rbp),%xmm0,%xmm0
689e: c5 ca 5a f0 vcvtss2sd %xmm0,%xmm6,%xmm6
68a2: c4 e1 f9 7e f0 vmovq %xmm6,%rax
68a7: c4 e1 f9 6e c0 vmovq %rax,%xmm0
68ac: e8 4f b8 ff ff callq 2100 <sqrt@plt>
68b1: c5 fb 5a c0 vcvtsd2ss %xmm0,%xmm0,%xmm0
68b5: 48 8b 85 18 ff ff ff mov -0xe8(%rbp),%rax
68bc: c5 fa 11 00 vmovss %xmm0,(%rax)
```

Messungen

Kompiliert wurde jeweils unter Debian 10 mit GCC 8.3.0. Die Messungen wurden auf einem PC mit folgenden Merkmalen erstellt:

* CPU: AMD Ryzen 7 5800X @ 4.60 GHz
* RAM: 32GB DDR4-3200

Messung ohne Optimierung

Durchlauf	Zeit
1	4.62049 s
2	4.60774 s
3	4.56693 s
4	4.56833 s
5	4.58256 s
6	4.58978 s
7	4.56290 s
8	4.57446 s

Durchlauf	Zeit
9	4.55789 s
10	4.59217 s
Durchschnitt	4.582325 s

Messung mit Optimierung

Durchlauf	Zeit
1	4.02156 s
2	4.07757 s
3	4.06602 s
4	4.04903 s
5	4.07341 s
6	4.0889 s
7	4.09844 s
8	4.09043 s
9	4.08588 s
10	4.0745 s
Durchschnitt	4.072574 s

Ergibt eine Performance-Steigerung von etwa +12,5 %. Zurückzuführen ist diese Steigerung auf die Reduktion der Quadratwurzelberechnung. Durch die Ausführung der Berechnungen am Ende der Funktion werden Quadratwurzeln nur noch berechnet wenn dies notwendig ist. Zusätzlich wurde die maximale Anzahl der Quadratwurzelberechnungen reduziert und durch die Prüfung, ob bereits nähere Schnittpunkte gefunden wurden, kann die Berechnung häufiger frühzeitig abgebrochen werden.