Laborbericht Optimierung von Programmen

Ansgar Lichter

April 2023

1 Schnittpunkttest

1.1 Verbesserung des Quellcodes

Insgesamt sind 3 Verbesserungen beim Schnittpunkttest implementiert. Die erste Änderung führt zu einem vorzeitigen Abbruch der Berechnung des Schnittpunkts. Falls der neu berechnete t-Wert nicht kleiner als der aktuell minimale t-Wert ist, existiert bereits ein Schnittpunkt, der näher am Betrachter liegt.
Die zweite und dritte Verbesserung bezieht sich auf die Berechnung der Variablen u und v. Zuerst wird diese Berechnung an das Ende der Methode verschoben. Denn die Werte sind nur notwendig, wenn tatsächlich ein Schnittpunkt existiert. Bisher wird hierfür die Länge für jeden Vektor berechnet und anschließend durch die Fläche geteilt. Daraus folgt, dass die Funktion zur Berechnung der Quadratwurzel insgesamt 3-mal aufgerufen wird. Da die Berechnung der Quadratwurzel viele Ressourcen benötigt, lohnt es sich, die Anzahl an Aufrufen zu reduzieren. Durch mathematische Umformungen muss die Funktion zur Berechnung der Quadratwurzel nur noch 2-mal aufgerufen werden. Denn es kann zuerst das Quadrat der Länge durch das Quadrat der Fläche geteilt werden. Erst danach wird die Quadratwurzel für jeden Vektor berechnet.

```
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;
}
```

Abbildung 1: Verbesserte Schnittstellenberechnung

1.2 Messungen

Die Laufzeiten wurden unter Ubuntu 22.04 mit GCC 11.3.0 ermittelt. Die folgende Hardware wurde hierfür verwendet:

- CPU: AMD Ryzen 7 2700 @ 3,6 GHz
- RAM: 32 GB DDR4-3200
- GPU: AMD RADEON RX590

Tabelle 1: Laufzeiten Schnittpunkttest

Durchlauf	**Nicht Optimiert** [s]	**Optimiert** [s]
1	10,7821	9,9917
2	10,8680	9,6968
3	10,8098	9,7212
4	10,8325	9,8179
5	10,7824	9,6891
6	10,8378	9,6616
7	10,7955	9,6814
8	10,7985	9,6910
9	10,8385	9,7226
10	10,7847	9,6863
Durchschnitt	10,8130	9,6863

Die Verbesserung beträgt ca. 10%. Diese Performancesteigerung ist auf die Reduktion der Aufrufe für die Berechnung der Quadratwurzel zurückzuführen. Durch frühzeitiges Abbrechen wird die Quadratwurzel nur dann berechnet, wenn diese tatsächlich benötigt wird. Selbst in diesem Fall wird die Quadratwurzel durch mathematische Umformungen seltener berechnet.

2 Quadratwurzelberechnung optimieren

2.1 Verbesserung des Quellcodes

Die Quadratwurzelberechnung wird durch die Verwendung des Newton-Verfahrens verbessert. Wenn möglich, wird der Assembler-Code ohne Optimierung des Compilers verwendet, um die Lesbarkeit zu erhöhen.

```
float sqrt1(float *a)
{
  float root;

  int *ai = reinterpret_cast<int *>(a);
  int *initial = reinterpret_cast<int *>(&root);
  *initial = (1 << 29) + (*ai >> 1) - (1 << 22) - 0x4C000;

  for (size_t i = 0; i < LOOPS; i++)
  {
    root = 0.5 * (root + *a / root);
  }

  return root;
}
```

Abbildung 2: Implementierung sqrt1

Der Assembler-Code für den Aufruf von sqrt1 sieht wie folgt aus.

```
float sqrt1(float *a)
   0: 55                       push   %rbp
   1: 48 89 e5                 mov    %rsp,%rbp
   4: 48 83 ec 40              sub    $0x40,%rsp
   8: 48 89 7d c8              mov    %rdi,-0x38(%rbp)
   c: 64 48 8b 04 25 28 00     mov    %fs:0x28,%rax
  13: 00 00
  15: 48 89 45 f8              mov    %rax,-0x8(%rbp)
  19: 31 c0                    xor    %eax,%eax
{
  float root;
```

```
  int *ai = reinterpret_cast<int *>(a);
  1b: 48 8b 45 c8             mov    -0x38(%rbp),%rax
  1f: 48 89 45 e8             mov    %rax,-0x18(%rbp)
  int *initial = reinterpret_cast<int *>(&root);
  23: 48 8d 45 dc             lea    -0x24(%rbp),%rax
  27: 48 89 45 f0             mov    %rax,-0x10(%rbp)
  *initial = (1 << 29) + (*ai >> 1) - (1 << 22) - 0x4C000;
  2b: 48 8b 45 e8             mov    -0x18(%rbp),%rax
  2f: 8b 00                   mov    (%rax),%eax
  31: d1 f8                   sar    %eax
  33: 8d 90 00 40 bb 1f       lea    0x1fbb4000(%rax),%edx
  39: 48 8b 45 f0             mov    -0x10(%rbp),%rax
  3d: 89 10                   mov    %edx,(%rax)

  for (size_t i = 0; i < LOOPS; i++)
  3f: 48 c7 45 e0 00 00 00    movq   $0x0,-0x20(%rbp)
  46: 00
  47: 48 83 7d e0 01          cmpq   $0x1,-0x20(%rbp)
  4c: 77 31                   ja     7f <_Z5sqrt1ILm2EEfPf+0x7f>
  {
    root = 0.5 * (root + *a / root);
  4e: 48 8b 45 c8             mov    -0x38(%rbp),%rax
  52: c5 fa 10 00             vmovss (%rax),%xmm0
  56: c5 fa 10 4d dc          vmovss -0x24(%rbp),%xmm1
  5b: c5 fa 5e c9             vdivss %xmm1,%xmm0,%xmm1
  5f: c5 fa 10 45 dc          vmovss -0x24(%rbp),%xmm0
  64: c5 f2 58 c8             vaddss %xmm0,%xmm1,%xmm1
  68: c5 fa 10 05 00 00 00    vmovss 0x0(%rip),%xmm0        # 70
    <_Z5sqrt1ILm2EEfPf+0x70>
  6f: 00
  70: c5 f2 59 c0             vmulss %xmm0,%xmm1,%xmm0
  74: c5 fa 11 45 dc          vmovss %xmm0,-0x24(%rbp)
  for (size_t i = 0; i < LOOPS; i++)
  79: 48 ff 45 e0             incq   -0x20(%rbp)
  7d: eb c8                   jmp    47 <_Z5sqrt1ILm2EEfPf+0x47>
  }

  return root;
  7f: c5 fa 10 45 dc          vmovss -0x24(%rbp),%xmm0
}
  84: 48 8b 45 f8             mov    -0x8(%rbp),%rax
  88: 64 48 33 04 25 28 00    xor    %fs:0x28,%rax
  8f: 00 00
  91: 74 05                   je     98 <_Z5sqrt1ILm2EEfPf+0x98>
  93: e8 00 00 00 00          call   98 <_Z5sqrt1ILm2EEfPf+0x98>
```

```
  98: c9                        leave
  99: c3                        ret
```

Abbildung 3: Assembler-Code sqrt1

Die zweite Variante berechnet vier Wurzeln gleichzeitig, damit die Packed-SIMD Befehle verwendet werden können.

```
void sqrt2(float *__restrict__ a, float *__restrict__ root)
{
  int *ai = reinterpret_cast<int *>(a);
  int *initial = reinterpret_cast<int *>(root);

  initial[0] = (1 << 29) + (ai[0] >> 1) - (1 << 22) - 0x4C000;
  initial[1] = (1 << 29) + (ai[1] >> 1) - (1 << 22) - 0x4C000;
  initial[2] = (1 << 29) + (ai[2] >> 1) - (1 << 22) - 0x4C000;
  initial[3] = (1 << 29) + (ai[3] >> 1) - (1 << 22) - 0x4C000;

  for (size_t i = 0; i < LOOPS; i++)
  {
    root[0] = 0.5 * (root[0] + a[0] / root[0]);
    root[1] = 0.5 * (root[1] + a[1] / root[1]);
    root[2] = 0.5 * (root[2] + a[2] / root[2]);
    root[3] = 0.5 * (root[3] + a[3] / root[3]);
  }
}
```

Abbildung 4: Implementierung sqrt2

Der Assembler-Code für sqrt2 wird mit der Optimierung (-O3) des Compilers erzeugt, damit die SIMD-Instruktionen tatsächlich verwendet werden. Daraus ergibt sich der folgende Assembler-Code.

```
void sqrt2(float *__restrict__ a, float *__restrict__ root)
{
  int *ai = reinterpret_cast<int *>(a);
  int *initial = reinterpret_cast<int *>(root);

  initial[0] = (1 << 29) + (ai[0] >> 1) - (1 << 22) - 0x4C000;
 7e0: c5 f8 28 12             vmovaps (%rdx),%xmm2
 7e4: c5 f8 28 42 20          vmovaps 0x20(%rdx),%xmm0
 7e9: 48 83 c0 40             add    $0x40,%rax
 7ed: 48 83 c2 40             add    $0x40,%rdx
 7f1: c5 f8 c6 4a f0 88       vshufps $0x88,-0x10(%rdx),%xmm0,%
    xmm1
 7f7: c5 e8 c6 72 d0 88       vshufps $0x88,-0x30(%rdx),%xmm2,%
    xmm6
```

```
7fd: c5 f8 c6 42 f0 dd       vshufps $0xdd,-0x10(%rdx),%xmm0,%
   xmm0
803: c5 e8 c6 52 d0 dd       vshufps $0xdd,-0x30(%rdx),%xmm2,%
   xmm2
809: c5 c8 c6 d9 88          vshufps $0x88,%xmm1,%xmm6,%xmm3
80e: c5 c8 c6 f1 dd          vshufps $0xdd,%xmm1,%xmm6,%xmm6
813: c5 e8 c6 c8 88          vshufps $0x88,%xmm0,%xmm2,%xmm1
818: c5 e8 c6 c0 dd          vshufps $0xdd,%xmm0,%xmm2,%xmm0
81d: c5 e9 72 e3 01          vpsrad $0x1,%xmm3,%xmm2
822: c5 e9 fe d5             vpaddd %xmm5,%xmm2,%xmm2
 initial[2] = (1 << 29) + (ai[2] >> 1) - (1 << 22) - 0x4C000;
 initial[3] = (1 << 29) + (ai[3] >> 1) - (1 << 22) - 0x4C000;

 for (size_t i = 0; i < LOOPS; i++)
 {
   root[0] = 0.5 * (root[0] + a[0] / root[0]);
826: c5 60 5e d2             vdivps %xmm2,%xmm3,%xmm10
82a: c5 28 58 d2             vaddps %xmm2,%xmm10,%xmm10
 initial[1] = (1 << 29) + (ai[1] >> 1) - (1 << 22) - 0x4C000;
82e: c5 e9 72 e1 01          vpsrad $0x1,%xmm1,%xmm2
833: c5 e9 fe d5             vpaddd %xmm5,%xmm2,%xmm2
   root[1] = 0.5 * (root[1] + a[1] / root[1]);
837: c5 70 5e ca             vdivps %xmm2,%xmm1,%xmm9
   root[0] = 0.5 * (root[0] + a[0] / root[0]);
83b: c5 28 59 d4             vmulps %xmm4,%xmm10,%xmm10
   root[1] = 0.5 * (root[1] + a[1] / root[1]);
83f: c5 b0 58 d2             vaddps %xmm2,%xmm9,%xmm2
   root[0] = 0.5 * (root[0] + a[0] / root[0]);
843: c4 c1 60 5e da          vdivps %xmm10,%xmm3,%xmm3
   root[1] = 0.5 * (root[1] + a[1] / root[1]);
848: c5 68 59 cc             vmulps %xmm4,%xmm2,%xmm9
 initial[2] = (1 << 29) + (ai[2] >> 1) - (1 << 22) - 0x4C000;
84c: c5 e9 72 e6 01          vpsrad $0x1,%xmm6,%xmm2
851: c5 e9 fe d5             vpaddd %xmm5,%xmm2,%xmm2
   root[2] = 0.5 * (root[2] + a[2] / root[2]);
855: c5 48 5e c2             vdivps %xmm2,%xmm6,%xmm8
   root[0] = 0.5 * (root[0] + a[0] / root[0]);
859: c4 c1 60 58 da          vaddps %xmm10,%xmm3,%xmm3
85e: c5 e0 59 dc             vmulps %xmm4,%xmm3,%xmm3
   root[2] = 0.5 * (root[2] + a[2] / root[2]);
862: c5 38 58 c2             vaddps %xmm2,%xmm8,%xmm8
 initial[3] = (1 << 29) + (ai[3] >> 1) - (1 << 22) - 0x4C000;
866: c5 e9 72 e0 01          vpsrad $0x1,%xmm0,%xmm2
86b: c5 e9 fe d5             vpaddd %xmm5,%xmm2,%xmm2
   root[3] = 0.5 * (root[3] + a[3] / root[3]);
86f: c5 f8 5e fa             vdivps %xmm2,%xmm0,%xmm7
```

```
   root[2] = 0.5 * (root[2] + a[2] / root[2]);
873: c5 38 59 c4             vmulps %xmm4,%xmm8,%xmm8
   root[3] = 0.5 * (root[3] + a[3] / root[3]);
877: c5 c0 58 fa             vaddps %xmm2,%xmm7,%xmm7
   root[1] = 0.5 * (root[1] + a[1] / root[1]);
87b: c4 c1 70 5e d1          vdivps %xmm9,%xmm1,%xmm2
   root[3] = 0.5 * (root[3] + a[3] / root[3]);
880: c5 c0 59 fc             vmulps %xmm4,%xmm7,%xmm7
   root[2] = 0.5 * (root[2] + a[2] / root[2]);
884: c4 c1 48 5e c8          vdivps %xmm8,%xmm6,%xmm1
   root[1] = 0.5 * (root[1] + a[1] / root[1]);
889: c4 c1 68 58 d1          vaddps %xmm9,%xmm2,%xmm2
88e: c5 e8 59 d4             vmulps %xmm4,%xmm2,%xmm2
   root[3] = 0.5 * (root[3] + a[3] / root[3]);
892: c5 f8 5e c7             vdivps %xmm7,%xmm0,%xmm0
   root[2] = 0.5 * (root[2] + a[2] / root[2]);
896: c4 c1 70 58 c8          vaddps %xmm8,%xmm1,%xmm1
89b: c5 f0 59 cc             vmulps %xmm4,%xmm1,%xmm1
   root[3] = 0.5 * (root[3] + a[3] / root[3]);
89f: c5 e0 14 f1             vunpcklps %xmm1,%xmm3,%xmm6
8a3: c5 e0 15 c9             vunpckhps %xmm1,%xmm3,%xmm1
8a7: c5 f8 58 c7             vaddps %xmm7,%xmm0,%xmm0
8ab: c5 f8 59 c4             vmulps %xmm4,%xmm0,%xmm0
8af: c5 e8 14 d8             vunpcklps %xmm0,%xmm2,%xmm3
8b3: c5 e8 15 c0             vunpckhps %xmm0,%xmm2,%xmm0
8b7: c5 c8 14 d3             vunpcklps %xmm3,%xmm6,%xmm2
8bb: c5 c8 15 f3             vunpckhps %xmm3,%xmm6,%xmm6
8bf: c5 f8 29 50 c0          vmovaps %xmm2,-0x40(%rax)
8c4: c5 f0 14 d0             vunpcklps %xmm0,%xmm1,%xmm2
8c8: c5 f0 15 c8             vunpckhps %xmm0,%xmm1,%xmm1
8cc: c5 f8 29 70 d0          vmovaps %xmm6,-0x30(%rax)
8d1: c5 f8 29 50 e0          vmovaps %xmm2,-0x20(%rax)
8d6: c5 f8 29 48 f0          vmovaps %xmm1,-0x10(%rax)
8db: 4c 39 e8                cmp    %r13,%rax
8de: 0f 85 fc fe ff ff       jne    7e0 <
  _Z17measure_sqrt_timeILm2EEvv+0x7e0>
 time.reset_clock();

 std::cout << "sqrt2␣(Newton␣method␣for␣sequence␣of␣4␣floats)"
   << std::endl;
 time.start_clock();
```

Abbildung 5: Assembler-Code sqrt2

Es ist zu erkennen, dass die Befehle vdivaps, vaddps und vmulps verwendet werden, um die Quadratwurzeln zu berechnen.
In der dritten Funktion v4sf_sqrt werden die Vektoren v4sf und v4si als Daten-

typen verwendet. Es werden 4 Werte gleichzeitig berechnet.

```
void v4sf_sqrt(v4sf *__restrict__ a, v4sf *__restrict__ root)
{
  v4si *ai = reinterpret_cast<v4si *>(a);
  v4si *initial = reinterpret_cast<v4si *>(root);
  *initial = (1 << 29) + (*ai >> 1) - (1 << 22) - 0x4C000;

  for (size_t i = 0; i < LOOPS; i++)
  {
    *root = 0.5 * (*root + *a / *root);
  }
}
```

Abbildung 6: Implementierung sqrt3

Der dazugehörige Assembler-Code ist wie folgt aufgebaut.

```
void v4sf_sqrt(v4sf *__restrict__ a, v4sf *__restrict__ root)
   0: 55                    push   %rbp
   1: 48 89 e5              mov    %rsp,%rbp
   4: 48 89 7d d8           mov    %rdi,-0x28(%rbp)
   8: 48 89 75 d0           mov    %rsi,-0x30(%rbp)
  v4si *ai = reinterpret_cast<v4si *>(a);
   c: 48 8b 45 d8           mov    -0x28(%rbp),%rax
  10: 48 89 45 f0           mov    %rax,-0x10(%rbp)
  v4si *initial = reinterpret_cast<v4si *>(root);
  14: 48 8b 45 d0           mov    -0x30(%rbp),%rax
  18: 48 89 45 f8           mov    %rax,-0x8(%rbp)
  *initial = (1 << 29) + (*ai >> 1) - (1 << 22) - 0x4C000;
  1c: 48 8b 45 f0           mov    -0x10(%rbp),%rax
  20: c5 f8 28 00           vmovaps (%rax),%xmm0
  24: c5 f1 72 e0 01        vpsrad $0x1,%xmm0,%xmm1
  29: c5 f8 28 05 00 00 00  vmovaps 0x0(%rip),%xmm0        # 31
     <_Z9v4sf_sqrtILm2EEvPDv4_fS1_+0x31>
  30: 00
  31: c5 f1 fe c0           vpaddd %xmm0,%xmm1,%xmm0
  35: 48 8b 45 f8           mov    -0x8(%rbp),%rax
  39: c5 f8 29 00           vmovaps %xmm0,(%rax)
  for (size_t i = 0; i < LOOPS; i++)
  3d: 48 c7 45 e8 00 00 00  movq   $0x0,-0x18(%rbp)
  44: 00
  45: 48 83 7d e8 01        cmpq   $0x1,-0x18(%rbp)
  4a: 77 3a                 ja     86 <
    _Z9v4sf_sqrtILm2EEvPDv4_fS1_+0x86>
    *root = 0.5 * (*root + *a / *root);
  4c: 48 8b 45 d0           mov    -0x30(%rbp),%rax
```

```
  50: c5 f8 28 08             vmovaps (%rax),%xmm1
  54: 48 8b 45 d8             mov    -0x28(%rbp),%rax
  58: c5 f8 28 00             vmovaps (%rax),%xmm0
  5c: 48 8b 45 d0             mov    -0x30(%rbp),%rax
  60: c5 f8 28 10             vmovaps (%rax),%xmm2
  64: c5 f8 5e c2             vdivps %xmm2,%xmm0,%xmm0
  68: c5 f0 58 c8             vaddps %xmm0,%xmm1,%xmm1
  6c: c5 f8 28 05 00 00 00    vmovaps 0x0(%rip),%xmm0        # 74
     <_Z9v4sf_sqrtILm2EEvPDv4_fS1_+0x74>
  73: 00
  74: c5 f0 59 c0             vmulps %xmm0,%xmm1,%xmm0
  78: 48 8b 45 d0             mov    -0x30(%rbp),%rax
  7c: c5 f8 29 00             vmovaps %xmm0,(%rax)
  for (size_t i = 0; i < LOOPS; i++)
  80: 48 ff 45 e8             incq   -0x18(%rbp)
  84: eb bf                   jmp    45 <
    _Z9v4sf_sqrtILm2EEvPDv4_fS1_+0x45>
}
  86: 90                      nop
  87: 5d                      pop    %rbp
  88: c3                      ret
```

Abbildung 7: Assembler-Code sqrt3

Der Compiler verwendet die Befehle vdivps, vaddps und vmulps, obwohl die Optimierung (-03) nicht genutzt worden ist. D. h. die Datentypen v4sf und v4si helfen, dass der Compiler die Packed-SIMD Befehle besser verwendet

2.2 Messungen

Die Laufzeiten wurden unter Ubuntu 22.04 mit GCC 7.5.0 ermittelt. Die folgende Hardware wurde hierfür verwendet:

- CPU: AMD Ryzen 7 2700 @ 3,6 GHz
- RAM: 32 GB DDR4-3200
- GPU: AMD RADEON RX590

Tabelle 2: Laufzeiten mit 2 Iterationen Quadratwurzelberechnung

Durchlauf	**Math [ns]**	**sqrt1 [ns]**	**4*sqrt1 [ns]**	**sqrt2 [ns]**	**sqrt3 [ns]**
1	1.550.353	207.264	262.928	257.761	207.575
2	1.553.329	209.363	270.470	260.488	208.772
3	1.555.451	208.665	274.640	258.737	209.404
4	1.560.481	214.224	264.955	258.572	207.181
5	1.552.509	209.729	263.231	258.059	208.663
6	1.566.568	213.976	262.962	258.724	207.436
7	1.554.651	208.395	263.116	259.313	208.201
8	1.548.499	210.306	262.513	258.018	215.134
9	1.548.651	207.581	261.884	269.692	210.473
10	1.555.045	208.803	262.526	259.049	208.273
Durchschnitt	1.554.554	209.831	264.923	259.841	209.111

Tabelle 3: Laufzeiten mit 3 Iterationen Quadratwurzelberechnung

Durchlauf	**Math [ns]**	**sqrt1 [ns]**	**4*sqrt1 [ns]**	**sqrt2 [ns]**	**sqrt3 [ns]**
1	1.551.488	327.990	379.716	371.163	327.188
2	1.553.255	327.001	379.591	369.977	330.307
3	1.567.901	326.149	378.515	367.487	326.291
4	1.545.201	327.463	379.357	374.575	329.070
5	1.547.554	327.131	381.140	369.464	326.867
6	1.548.098	327.313	377.570	371.417	327.576
7	1.548.285	326.754	382.652	372.342	332.451
8	1.541.967	329.712	386.370	370.390	331.265
9	1.553.822	333.117	378.372	368.491	326.949
10	1.549.159	326.457	377.949	368.519	326.878
Durchschnitt	1.550.673	327.909	380.123	370.383	328.484

Tabelle 4: Laufzeiten mit 4 Iterationen Quadratwurzelberechnung

Durchlauf	**Math [ns]**	**sqrt1 [ns]**	**4*sqrt1 [ns]**	**sqrt2 [ns]**	**sqrt3 [ns]**
1	1.551.790	466.633	547.409	490.402	464.693
2	1.554.039	466.413	547.782	491.392	465.340
3	1.571.946	465.407	549.453	490.991	466.395
4	1.549.278	465.383	549.414	490.896	465.203
5	1.550.201	466.270	546.899	489.967	466.306
6	1.548.477	465.720	549.908	490.576	465.112
7	1.550.247	464.539	547.577	490.334	464.719
8	1.549.212	467.189	549.146	490.557	464.757
9	1.550.513	465.105	547.449	490.006	466.092
10	1.589.647	465.223	551.296	492.084	465.076
Durchschnitt	1.556.535	465.788	548.633	490.721	465.369

Die Laufzeiten zeigen, dass die selbst implementierten Funktionen die Quadratwurzel mit dem Newton-Verfahren deutlich schneller annähern können als die genaue Berechnung der Quadratwurzel mit der Funktion Math.sqrt. Ebenfalls steigt die Ausführungszeit mit der Anzahl an Iterationen an.
Die Variante sqrt1 und sqrt3 sind am schnellsten. Die Funktion sqrt2 ist im Vergleich etwas langsamer. Dies ist vermutlich auf eine schlechtere Optimierung des Compilers zurückzuführen. Die automatische Vektorisierung des Compilers funktioniert schlechter als die manuelle Optimierung durch die Verwendung der entsprechenden Vektordatentypen. Die Variante 4 * sqrt1 ist aufgrund der zusätzlichen inneren Schleife langsamer als sqrt1. Denn der Compiler kann einzelne Schleife besser optimieren als die verschachtelten Schleifen.
Es wird erwartet, dass die vektorisierte Berechnung der Quadratwurzel schneller erfolgt. Diese Erwartung hat sich nicht bewahrheitet. Der Großteil der Laufzeit wird nicht für die Berechnung der Quadratwurzel, sondern für Speicherbefehle benötigt. Die Vorteile der SIMD-Instruktionen sind nicht so groß wie angenommen.

3 Optimierung mit k-d-Baum

3.1 Verbesserung des Quellcodes

Die Dreiecke werden abhängig von ihrer Position in einem entsprechenden Knoten des k-d-Baums gespeichert. Dies beschleunigt die Suche nach einem Dreieck, das mit dem Sehstrahl einen Schnittpunkt besitzt. Denn die Anzahl an zu testenden Dreiecken wird deutlich reduziert. Dreiecke, die sich nicht im Bereich des Sehstrahls befinden, sind nicht im entsprechenden Knoten gespeichert. Dadurch werden diese beim Schnittpunkttest nicht berücksichtigt.

```
#include "kdtree.h"
```

```
#include <algorithm>
#include <iostream>

BoundingBox::BoundingBox() {}

BoundingBox::BoundingBox(Vector<FLOAT, 3> min, Vector<FLOAT, 3>
    max)
    : min(min), max(max) {}

void BoundingBox::split(BoundingBox &left, BoundingBox &right)
{
  left.min = min;
  right.max = max;

  float lengthX = std::abs(max[0] - min[0]);
  float lengthY = std::abs(max[1] - min[1]);
  float lengthZ = std::abs(max[2] - min[2]);

  // Split at the longest axis
  if (lengthX >= lengthY && lengthX >= lengthZ)
  {
    float newWidth = lengthX / 2;
    left.max = Vector<float, 3>{min[0] + newWidth, max[1], max
    [2]};
    right.min = Vector<float, 3>{min[0] + newWidth, min[1], min
    [2]};
    return;
  }

  if (lengthY >= lengthX && lengthY >= lengthZ)
  {
    float newWidth = lengthY / 2;
    left.max = Vector<float, 3>{max[0], min[1] + newWidth, max
    [2]};
    right.min = Vector<float, 3>{min[0], min[1] + newWidth, min
    [2]};
    return;
  }

  float newWidth = lengthZ / 2;
  left.max = Vector<float, 3>{max[0], max[1], min[2] + newWidth};
  right.min = Vector<float, 3>{min[0], min[1], min[2] + newWidth
    };
}

bool BoundingBox::contains(Vector<FLOAT, 3> v)
```

```
{
  return v[0] >= min[0] && v[0] <= max[0] && v[1] >= min[1] && v
    [1] <= max[1] && v[2] >= min[2] && v[2] <= max[2];
}

bool BoundingBox::contains(Triangle<FLOAT> *triangle)
{
  return contains(triangle->p1) || contains(triangle->p2) ||
    contains(triangle->p3);
}

bool BoundingBox::intersects(Vector<FLOAT, 3> eye, Vector<FLOAT,
    3> direction)
{
  // slab test implementation
  FLOAT tmin[3] = {(min[0] - eye[0]) / direction[0],
                   (min[1] - eye[1]) / direction[1],
                   (min[2] - eye[2]) / direction[2]};
  FLOAT tmax[3] = {(max[0] - eye[0]) / direction[0],
                   (max[1] - eye[1]) / direction[1],
                   (max[2] - eye[2]) / direction[2]};
  FLOAT tminimum = std::min(tmin[0], tmax[0]);
  FLOAT tmaximum = std::max(tmin[0], tmax[0]);
  tminimum = std::max(tminimum, std::min(tmin[1], tmax[1]));
  tmaximum = std::min(tmaximum, std::max(tmin[1], tmax[1]));
  tminimum = std::max(tminimum, std::min(tmin[2], tmax[2]));
  tmaximum = std::min(tmaximum, std::max(tmin[2], tmax[2]));

  return tmaximum >= tminimum;
}

KDTree::~KDTree()
{
  delete left;
  delete right;
}

KDTree *KDTree::buildTree(KDTree *tree, std::vector<Triangle<
    FLOAT> *> &triangles)
{
  if (triangles.size() <= MAX_TRIANGLES_PER_LEAF)
  {
    tree->triangles.insert(std::end(tree->triangles), std::begin(
    triangles), std::end(triangles));
    return tree;
  }
```

```
  this->left = new KDTree();
  this->right = new KDTree();
  this->box.split(left->box, right->box);

  auto trianglesInLeftTree = std::vector<Triangle<float> *>();
  auto trianglesInRightTree = std::vector<Triangle<float> *>();
  for (auto const &triangle : triangles)
  {
    bool isPartOfLeftTree = tree->left->box.contains(triangle);
    bool isPartOfRightTree = tree->right->box.contains(triangle);
    if (isPartOfLeftTree && isPartOfRightTree)
    {
      tree->triangles.push_back(triangle);
    }
    else if (isPartOfLeftTree)
    {
      trianglesInLeftTree.push_back(triangle);
    }
    else if (isPartOfRightTree)
    {
      trianglesInRightTree.push_back(triangle);
    }
  }

  this->left = this->left->buildTree(left, trianglesInLeftTree);
  this->right = this->right->buildTree(right,
    trianglesInRightTree);
  return tree;
}

KDTree *KDTree::buildTree(std::vector<Triangle<FLOAT> *> &
    triangles)
{
  KDTree *root = new KDTree();
  Vector<FLOAT, 3> min = {triangles[0]->p1[0], triangles[0]->p1
    [0], triangles[0]->p1[0]};
  Vector<FLOAT, 3> max = {triangles[0]->p1[0], triangles[0]->p1
    [0], triangles[0]->p1[0]};
  for (auto iterator = std::next(triangles.begin()); iterator !=
    triangles.end(); ++iterator)
  {
    Triangle<float> *triangle = *iterator;

    max[0] = std::max({max[0], triangle->p1[0], triangle->p2[0],
    triangle->p3[0]});
```

```
    max[1] = std::max({max[1], triangle->p1[1], triangle->p2[1],
    triangle->p3[1]});
    max[2] = std::max({max[2], triangle->p1[2], triangle->p2[2],
    triangle->p3[2]});

    min[0] = std::min({min[0], triangle->p1[0], triangle->p2[0],
    triangle->p3[0]});
    min[1] = std::min({min[1], triangle->p1[1], triangle->p2[1],
    triangle->p3[1]});
    min[2] = std::min({min[2], triangle->p1[2], triangle->p2[2],
    triangle->p3[2]});
  }

  root->box = BoundingBox(min, max);
  root->buildTree(root, triangles);
  return root;
}

bool KDTree::hasNearestTriangle(Vector<FLOAT, 3> eye, Vector<
    FLOAT, 3> direction, Triangle<FLOAT> *&nearest_triangle,
    FLOAT &t, FLOAT &u, FLOAT &v, FLOAT minimum_t)
{
  if (!box.intersects(eye, direction))
  {
    return false;
  }

  if (this->left != nullptr && this->left->hasNearestTriangle(eye
    , direction, nearest_triangle, t, u, v, minimum_t))
  {
    minimum_t = t;
  }

  if (this->right && this->right->hasNearestTriangle(eye,
    direction, nearest_triangle, t, u, v, minimum_t))
  {
    minimum_t = t;
  }

  for (auto const triangle : this->triangles)
  {
    stats.no_ray_triangle_intersection_tests++;
    if (triangle->intersects(eye, direction, t, u, v, minimum_t))
    {
      stats.no_ray_triangle_intersections_found++;
      nearest_triangle = triangle;
```

```
      minimum_t = t;
    }
  }

  t = minimum_t;
  return nearest_triangle != nullptr;
}
```

Abbildung 8: Implementation der k-d-Bäume

3.2 Messungen

Die Laufzeiten wurden unter Ubuntu 22.04 mit GCC 11.3.0 ermittelt. Die folgende Hardware wurde hierfür verwendet:

- CPU: AMD Ryzen 7 2700 @ 3,6 GHz
- RAM: 32 GB DDR4-3200
- GPU: AMD RADEON RX590

Tabelle 5: Laufzeiten mit k-d-Baum

Durchlauf	**Nicht Optimiert [s]**	**Optimiert [s]**
1	9,7716	2,3738
2	9,7557	2,4006
3	9,7335	2,3789
4	9,7444	2,3943
5	9,7165	2,4168
6	9,8103	2,4519
7	9,6831	2,3892
8	9,6824	2,4584
9	9,7603	2,4198
10	9,7635	2,3942
Durchschnitt	9,7421	2,4078

Tabelle 6: Laufzeiten k-d-Baum

	Ohne k-d-Baum	**Mit k-d-Baum**
Anzahl Schnittpunkttests	519.950.720	139.090.305
Anzahl gefundener Schnittpunkte	35.290	36.802

Die Implementation mit k-d Baum ist 4 mal schneller als die ursprüngliche Variante. Die verbesserte Laufzeit ist auf die verringerte Anzahl an Schnittpunkttests zurückzuführen. Die Anzahl wurde nahezu um das 4-fache verringert. Allerdings ist die Anzahl gefunderer Schnittpunkte mit dem k-d-Baum etwas höher

als ohne. Dies ist abhängig von der Reihenfolge, in der die Schnittpunkte mit den Dreiecken gesucht werden. Wenn das Dreieck mit dem besten Schnittpunkt zuerst getestet wird, kann der Test für alle folgenden Dreiecke abgebrochen werden. Falls das beste Dreieck erst später getestet wird, werden zuvor entsprechend mehr Schnittpunkte gefunden. Die räumliche Anordnung der Dreiecke in einem k-d-Baum führt zu einer veränderten Reihenfolge der Schnittpunkttests. Deshalb steigt mit dem k-d-Baum die Anzahl an gefundenen Schnittpunkten leicht an.