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 viel 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. Dess es kann zuerst das Quadrat der Länge durch das Quadrat der Fläche geteilt werden. Erst danach wird die Quadratwurzel für jeden Vektor berechnet werden.

```
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. 110%. 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 tatächlich benötigt wird. Zusätzlich wird die Quadratwurzel deutlich seltener berechnet. Denn die Wurzel wird 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;
  root = *reinterpret_cast<float *>(initial);

  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: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 48 83 ec 40             sub    $0x40,%rsp
   c: 48 89 7d c8             mov    %rdi,-0x38(%rbp)
  10: 64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
  17: 00 00
  19: 48 89 45 f8             mov    %rax,-0x8(%rbp)
```

```
  1d: 31 c0                   xor    %eax,%eax
{
  float root;

  int *ai = reinterpret_cast<int *>(a);
  1f: 48 8b 45 c8             mov    -0x38(%rbp),%rax
  23: 48 89 45 e8             mov    %rax,-0x18(%rbp)
  int *initial = reinterpret_cast<int *>(&root);
  27: 48 8d 45 dc             lea    -0x24(%rbp),%rax
  2b: 48 89 45 f0             mov    %rax,-0x10(%rbp)
  *initial = (1 << 29) + (*ai >> 1) - (1 << 22) - 0x4C000;
  2f: 48 8b 45 e8             mov    -0x18(%rbp),%rax
  33: 8b 00                   mov    (%rax),%eax
  35: d1 f8                   sar    %eax
  37: 8d 90 00 40 bb 1f       lea    0x1fbb4000(%rax),%edx
  3d: 48 8b 45 f0             mov    -0x10(%rbp),%rax
  41: 89 10                   mov    %edx,(%rax)
  root = *reinterpret_cast<float *>(initial);
  43: 48 8b 45 f0             mov    -0x10(%rbp),%rax
  47: c5 fa 10 00             vmovss (%rax),%xmm0
  4b: c5 fa 11 45 dc          vmovss %xmm0,-0x24(%rbp)

  for (size_t i = 0; i < LOOPS; i++)
  50: 48 c7 45 e0 00 00 00    movq   $0x0,-0x20(%rbp)
  57: 00
  58: eb 2f                   jmp    89 <_Z5sqrt1ILm2EEfPf+0x89>
  {
    root = 0.5 * (root + *a / root);
  5a: 48 8b 45 c8             mov    -0x38(%rbp),%rax
  5e: c5 fa 10 00             vmovss (%rax),%xmm0
  62: c5 fa 10 4d dc          vmovss -0x24(%rbp),%xmm1
  67: c5 fa 5e c1             vdivss %xmm1,%xmm0,%xmm0
  6b: c5 fa 10 4d dc          vmovss -0x24(%rbp),%xmm1
  70: c5 fa 58 c1             vaddss %xmm1,%xmm0,%xmm0
  74: c5 fa 10 0d 00 00 00    vmovss 0x0(%rip),%xmm1        # 7c
    <_Z5sqrt1ILm2EEfPf+0x7c>
  7b: 00
  7c: c5 fa 59 c1             vmulss %xmm1,%xmm0,%xmm0
  80: c5 fa 11 45 dc          vmovss %xmm0,-0x24(%rbp)
  for (size_t i = 0; i < LOOPS; i++)
  85: 48 ff 45 e0             incq   -0x20(%rbp)
  89: 48 83 7d e0 01          cmpq   $0x1,-0x20(%rbp)
  8e: 76 ca                   jbe    5a <_Z5sqrt1ILm2EEfPf+0x5a>
  }

  return root;
```

```
  90: c5 fa 10 45 dc          vmovss -0x24(%rbp),%xmm0
}
  95: 48 8b 45 f8             mov    -0x8(%rbp),%rax
  99: 64 48 2b 04 25 28 00    sub    %fs:0x28,%rax
  a0: 00 00
  a2: 74 05                   je     a9 <_Z5sqrt1ILm2EEfPf+0xa9>
  a4: e8 00 00 00 00          call   a9 <_Z5sqrt1ILm2EEfPf+0xa9>
  a9: c9                      leave
  aa: 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;

  root = reinterpret_cast<float *>(initial);

  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;
5ce: c4 c1 79 6f 00          vmovdqa (%r8),%xmm0
 time.reset_clock();

 std::cout << sqrt2 "(Newton␣method␣for␣sequence␣of␣4␣floats)"
   << std::endl;
 time.start_clock();
 for (int j = 0; j < LOOP; j++)
5d3: 48 83 c1 10             add    $0x10,%rcx
5d7: 49 83 c0 10             add    $0x10,%r8
5db: c5 e9 72 e0 01          vpsrad $0x1,%xmm0,%xmm2
5e0: c5 e9 fe 15 00 00 00    vpaddd 0x0(%rip),%xmm2,%xmm2
    # 5e8 <_Z17measure_sqrt_timeILm2EEvv+0x5e8>
5e7: 00

 root = reinterpret_cast<float *>(initial);

 for (size_t i = 0; i < LOOPS; i++)
 {
   root[0] = 0.5 * (root[0] + a[0] / root[0]);
5e8: c5 f8 5e ca             vdivps %xmm2,%xmm0,%xmm1
5ec: c5 f0 58 ca             vaddps %xmm2,%xmm1,%xmm1
5f0: c5 f0 59 0d 00 00 00    vmulps 0x0(%rip),%xmm1,%xmm1
    # 5f8 <_Z17measure_sqrt_timeILm2EEvv+0x5f8>
5f7: 00
5f8: c5 f8 5e c1             vdivps %xmm1,%xmm0,%xmm0
5fc: c5 f8 58 c1             vaddps %xmm1,%xmm0,%xmm0
600: c5 f8 59 05 00 00 00    vmulps 0x0(%rip),%xmm0,%xmm0
    # 608 <_Z17measure_sqrt_timeILm2EEvv+0x608>
607: 00
608: c5 f8 29 41 f0          vmovaps %xmm0,-0x10(%rcx)
60d: 48 39 8d e8 2a cf ff    cmp    %rcx,-0x30d518(%rbp)
614: 75 b8                   jne    5ce <
   _Z17measure_sqrt_timeILm2EEvv+0x5ce>
      end = steady_clock::now();
616: e8 00 00 00 00          call   61b <
   _Z17measure_sqrt_timeILm2EEvv+0x61b>
   {
     sqrt2<LOOPS>(floats + i, roots + i);
   }
 }
 time.stop_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 Datentypen 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;
  root = reinterpret_cast<v4sf *>(initial);

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

Abbildung 6: Implementierung v4sf_sqrt

Der dazugehörige Assembler-Code ist wie folgt aufgebaut.

```
void v4sf_sqrt(v4sf *__restrict__ a, v4sf *__restrict__ root)
   0: f3 0f 1e fa             endbr64
   4: 55                      push   %rbp
   5: 48 89 e5                mov    %rsp,%rbp
   8: 48 89 7d d8             mov    %rdi,-0x28(%rbp)
   c: 48 89 75 d0             mov    %rsi,-0x30(%rbp)
  v4si *ai = reinterpret_cast<v4si *>(a);
  10: 48 8b 45 d8             mov    -0x28(%rbp),%rax
  14: 48 89 45 f0             mov    %rax,-0x10(%rbp)
  v4si *initial = reinterpret_cast<v4si *>(root);
  18: 48 8b 45 d0             mov    -0x30(%rbp),%rax
  1c: 48 89 45 f8             mov    %rax,-0x8(%rbp)
  *initial = (1 << 29) + (*ai >> 1) - (1 << 22) - 0x4C000;
  20: 48 8b 45 f0             mov    -0x10(%rbp),%rax
  24: c5 f9 6f 00             vmovdqa (%rax),%xmm0
  28: c5 f9 72 e0 01          vpsrad $0x1,%xmm0,%xmm0
  2d: c5 f9 6f 0d 00 00 00    vmovdqa 0x0(%rip),%xmm1        # 35
     <_Z9v4sf_sqrtILm3EEvPDv4_fS1_+0x35>
  34: 00
  35: c5 f9 fe c1             vpaddd %xmm1,%xmm0,%xmm0
  39: 48 8b 45 f8             mov    -0x8(%rbp),%rax
  3d: c5 f9 7f 00             vmovdqa %xmm0,(%rax)
  root = reinterpret_cast<v4sf *>(initial);
  41: 48 8b 45 f8             mov    -0x8(%rbp),%rax
  45: 48 89 45 d0             mov    %rax,-0x30(%rbp)
```

```
  for (size_t i = 0; i < LOOPS; i++)
  49: 48 c7 45 e8 00 00 00    movq   $0x0,-0x18(%rbp)
  50: 00
  51: eb 38                   jmp    8b <
    _Z9v4sf_sqrtILm3EEvPDv4_fS1_+0x8b>
    *root = 0.5 * (*root + *a / *root);
  53: 48 8b 45 d0             mov    -0x30(%rbp),%rax
  57: c5 f8 28 08             vmovaps (%rax),%xmm1
  5b: 48 8b 45 d8             mov    -0x28(%rbp),%rax
  5f: c5 f8 28 00             vmovaps (%rax),%xmm0
  63: 48 8b 45 d0             mov    -0x30(%rbp),%rax
  67: c5 f8 28 10             vmovaps (%rax),%xmm2
  6b: c5 f8 5e c2             vdivps %xmm2,%xmm0,%xmm0
  6f: c5 f0 58 c0             vaddps %xmm0,%xmm1,%xmm0
  73: c5 f8 28 0d 00 00 00    vmovaps 0x0(%rip),%xmm1        # 7b
     <_Z9v4sf_sqrtILm3EEvPDv4_fS1_+0x7b>
  7a: 00
  7b: c5 f8 59 c1             vmulps %xmm1,%xmm0,%xmm0
  7f: 48 8b 45 d0             mov    -0x30(%rbp),%rax
  83: c5 f8 29 00             vmovaps %xmm0,(%rax)
  for (size_t i = 0; i < LOOPS; i++)
  87: 48 ff 45 e8             incq   -0x18(%rbp)
  8b: 48 83 7d e8 02          cmpq   $0x2,-0x18(%rbp)
  90: 76 c1                   jbe    53 <
    _Z9v4sf_sqrtILm3EEvPDv4_fS1_+0x53>
}
  92: 90                      nop
  93: 90                      nop
  94: 5d                      pop    %rbp
  95: c3                      ret
```

Abbildung 7: Assembler-Code v4sf_sqrt

Der Compiler verwendet die Befehle vdivps, vaddps, 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 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 2: Laufzeiten mit 2 Iterationen Quadratwurzelberechnung

Durchlauf	**Math [ns]**	**sqrt1 [ns]**	**4*sqrt1 [ns]**	**sqrt2 [ns]**	**v4sf_sqrt [ns]**
1	554.953	750	210.728	206	229
2	554.728	769	209.032	206	229
3	557.112	757	208.248	207	231
4	555.335	761	208.360	206	230
5	556.656	833	210.461	211	230
6	555.698	760	209.861	208	236
7	555.770	763	209.603	207	238
8	558.664	752	208.333	210	234
9	554.947	747	210.201	261	230
10	555.411	748	214.551	210	234
Durchschnitt	555.927	764	209.938	213	232

Tabelle 3: Laufzeiten mit 3 Iterationen Quadratwurzelberechnung

Durchlauf	**Math [ns]**	**sqrt1 [ns]**	**4*sqrt1 [ns]**	**sqrt2 [ns]**	**v4sf_sqrt [ns]**
1	555.166	1231	327.311	326.978	331
2	555.699	1219	328.853	327.645	326
3	554.805	1220	327.711	327.440	327
4	556.986	1306	334.063	332.044	329
5	558.350	1211	328.182	326.746	327
6	554.941	1247	329.337	326.685	325
7	554.681	1226	328.448	331.811	329
8	557.345	1269	328.171	327.816	328
9	555.372	1255	327.943	328.127	328
10	556.190	1220	328.127	327.549	329
Durchschnitt	555.954	1240	328.815	328.284	328

Tabelle 4: Laufzeiten mit 4 Iterationen Quadratwurzelberechnung

Durchlauf	**Math [ns]**	**sqrt1 [ns]**	**4*sqrt1 [ns]**	**sqrt2 [ns]**	**v4sf_sqrt [ns]**
1	556.857	1209	329.266	328.744	328
2	556.357	1260	331.277	333.763	441
3	565.373	1213	329.504	327.999	327
4	555.840	1214	330.620	327.645	329
5	556.464	1412	333.951	328.275	341
6	555.660	1266	327.875	328.070	331
7	555.356	1231	332.871	329.001	327
8	556.220	1218	329.222	328.535	327
9	556.609	1269	330.240	328.736	330
10	555.946	1243	330.197	327.965	327
Durchschnitt	557.068	1254	330.502	328.873	341

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. Zusätzlich ist zu erkennen, dass die Verwendung der Packed-SIMD Instruktionen (sqrt2, v4sf_sqrt) nochmals zu einer deutlichen Beschleunigung führt. Die höhere Anzahl an Iterationen verdeutlicht diesen Effekt nochmals. Besonders mit Verwendung der Datentypen v4sf und v4si bleibt die Laufzeit unabhängig der Anzahl an Iterationen sehr gering.

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

Im Durchschnitt verbessern sich die Laufzeiten um ca. 400%. Die Implementation mit k-d Baum ist 4 mal schneller als die ursprüngliche Variante.

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 Anzahl der Schnittpunkttests wurden nahezu um das 4-fache verringert. Dies erklärt die verbesserte Laufzeit mit k-d-Bäumen. 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.