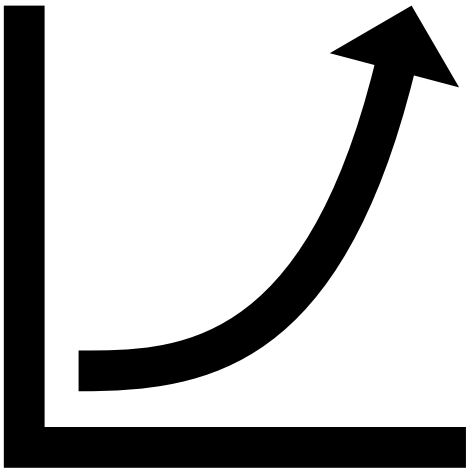
Laborbericht

Optimierung von Programmen

Wintersemester 2021

19.12.2021



von Silas Mario Schnurr (80543)

Dozent: Prof. Dr. Pape

In diesem Laborbericht werden die Ergebnisse für die drei abzugebenden Labor-Aufgaben (WiSe21) der Vorlesung Optimierung von Programmen beschrieben.

1 Aufgabe Schnittpunktest optimieren

Die erste Aufgabe ist die Optimierung der Schnittpunktberechnung eines Raytracers. Abbildung 1 zeigt die Änderungen zwischen der originalen und der optimierten Variante.

oldraytracer

```
1  ····Vector<T,·3>·normal·=··cross_product(p2·-·p1,·p3·-·p1);
2  ····
3  ····T·normalRayProduct·=·normal.scalar_product(·direction·);
4  ····T·area·=·normal.length();·//·used·for·u-v-parameter·calculation
5  
6  ····if·(·fabs(normalRayProduct)·<·EPSILON·)·{
7  ·····return·false;
8  ····}
9  
10 ····T·d·=·normal.scalar_product(·p1·);
11 ····t·=·(d·-·normal.scalar_product(·origin·)·)·/·normalRayProduct;
12 
13 ····if·(·t·<·0.0·)·{
14 ·····return·false;
15 ····}
16 ···
17 
18 ····Vector<T,·3>·intersection·=·origin·+·t·*·direction;
19 ···
20 ····Vector<T,·3>·vector·=·cross_product(p2·-·p1,·intersection·-·p1·);
21 ····if·(·normal.scalar_product(vector)·<·0.0·)·{
22 ·····return·false;
23 ····}
24 
25 ····
26 ····vector·=·cross_product(p3·-·p2,·intersection·-·p2·);
27 ····if·(·normal.scalar_product(vector)·<·0.0·)·{
28 ·····return·false;
29 ····}
30 
31 ····u·=·vector.length()·/·area;
32 
33 ····vector·=·cross_product(p1·-·p3,·intersection·-·p3·);
34 ····if·(normal.scalar_product(vector)·<·0.0·)·{
35 ·····return·false;
36 ····}
37 
38 ····v·=·vector.length()·/·area;
39 
40 
41 ····return·true;
```

newraytracer

```
1  ····Vector<T,·3>·normal·=··cross_product(p2·-·p1,·p3·-·p1);
2  ····
3  ····T·normalRayProduct·=·normal.scalar_product(·direction·);
4  
5  ····if·(·fabs(normalRayProduct)·<·EPSILON·)·{
6  ·····return·false;
7  ····}
8  
9  ····T·d·=·normal.scalar_product(·p1·);
10 ····t·=·(d·-·normal.scalar_product(·origin·)·)·/·normalRayProduct;
11 
12 ····if·(·t·<·0.0·||·t·>·minimum_t)·{
13 ·····return·false;
14 ····}
15 ···
16 
17 ····Vector<T,·3>·intersection·=·origin·+·t·*·direction;
18 ···
19 ····Vector<T,·3>·vector·=·cross_product(p2·-·p1,·intersection·-·p1·);
20 ····if·(·normal.scalar_product(vector)·<·0.0·)·{
21 ·····return·false;
22 ····}
23 
24 ····
25 ····vector·=·cross_product(p3·-·p2,·intersection·-·p2·);
26 ····if·(·normal.scalar_product(vector)·<·0.0·)·{
27 ·····return·false;
28 ····}
29 
30 ····T·uSquareOfLength·=·vector.square_of_length();
31 
32 ····vector·=·cross_product(p1·-·p3,·intersection·-·p3·);
33 ····if·(normal.scalar_product(vector)·<·0.0·)·{
34 ·····return·false;
35 ····}
36 
37 ····T·vSquareOfLength·=·vector.square_of_length();
38 ····T·areaSquareSum·=·normal.square_of_length();
39 
40 ····u·=·sqrt(uSquareOfLength·/·areaSquareSum);
41 ····v·=·sqrt(vSquareOfLength·/·areaSquareSum);
42 
43 ····return·true;
```

Abbildung 1: Quelltext Schnittpunktberechnung optimiert und nicht optimiert

Um eine Ausführbare Datei zu erstellen, wird für die nicht Optimierte Variante der Befehl

```
g++ -o a-old.out -Wall -pedantic -march=native -mfpmath=sse -mavx -O3 raytracer.cc statistics.cc
```

und für die Optimierte Variante der Befehl

```
g++ -o a-new.out -D OPTIMIZED_INTERSECTS -Wall -pedantic -march=native -mfpmath=sse -mavx -O3 raytracer.cc statistics.cc
```

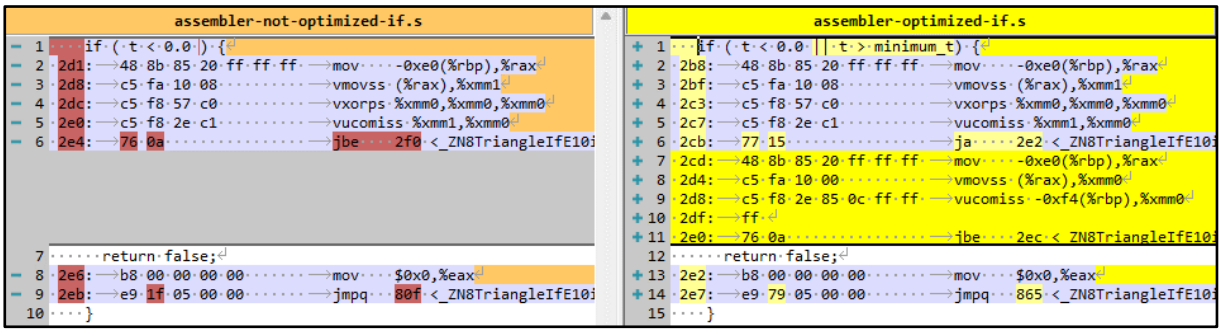
verwendet.

Des Weiteren kann der Assemblercode der optimierten Schnittpunktberechnung betrachtet werden. Der Assemblercode wird hierfür mit den folgenden Befehlen erzeugt.

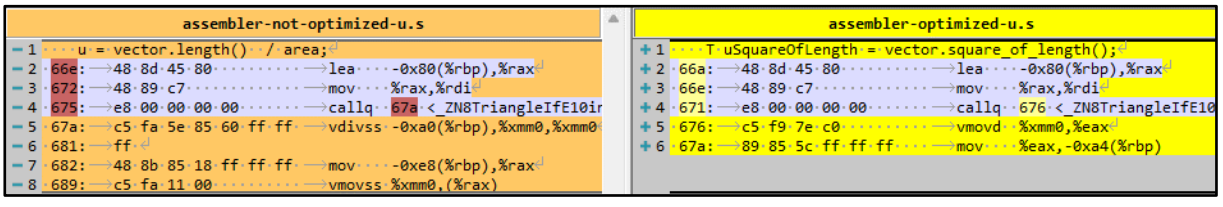
```
3  g++ -Wall -pedantic -march=native -mfpmath=sse -mavx -c -g raytracer.cc
4  objdump -S raytracer.o > raytracer-not-optimized.s

5  g++ -Wall -pedantic -D OPTIMIZED_INTERSECTS -march=native -mfpmath=sse -
   mavx -c -g raytracer.cc
6  objdump -S raytracer.o > raytracer-optimized.s
```

Hierbei fällt direkt auf, dass der Assemblercode ohne Compiler-Optimierungen (-O3) die **dreifache Anzahl an Zeilen** hat.

Für das hinzufügen der `minimum_t`-Abprüfung wird der entsprechende Assemblercode länger, da für den Vergleich (`jbe`) zusätzlich Bytes kopiert und Gleitkommazahlen verglichen (`vucomiss`) werden müssen.

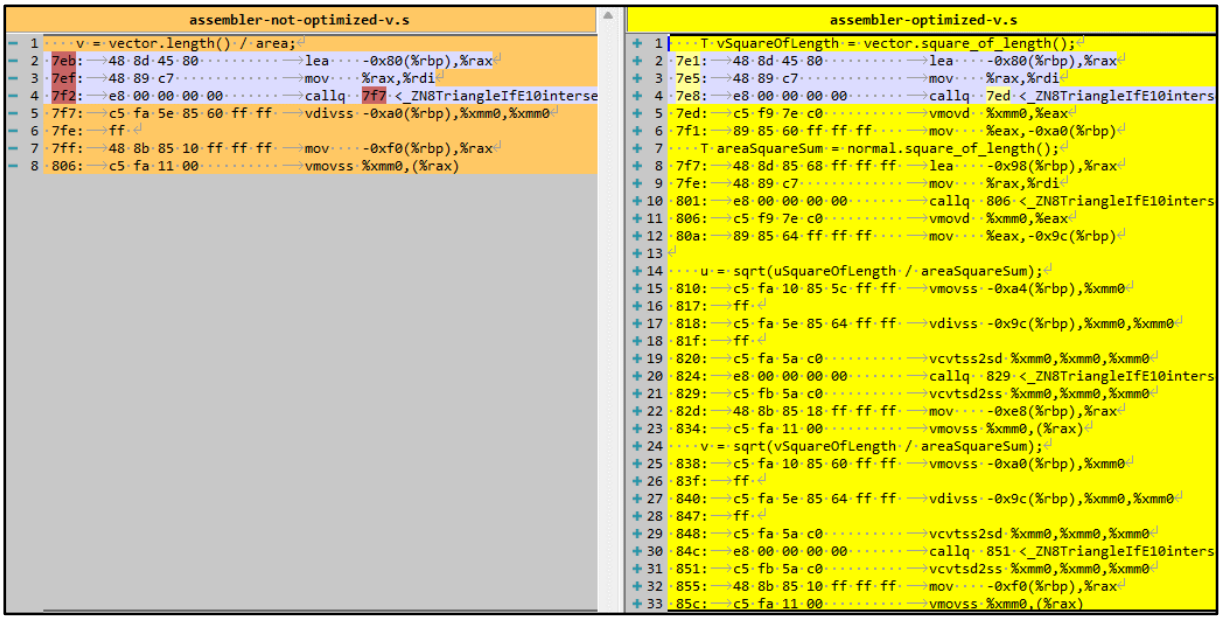


Für die Berechnung von u, beziehungsweise vorerst dem Quadrat der Länge, fällt im Assemblercode eine Division weg, für welche die Quadratwurzelberechnung erforderlich war.



Für die umgeformte Berechnung von v werden im Assemblercode nun mehr Instruktionen ausgeführt. Der Vorteil ist jedoch, dass die Berechnungen nur

erfolgen, wenn es erforderlich ist, dass insgesamt (pro Durchlauf) eine Quadratwurzel weniger (`vcvtss2sd`) berechnet werden muss und dass durch die Verwendung von Single-Instruction-Multiple-Data-Befehlen die Berechnungen schneller erfolgen können



Zusätzlich gibt es eine alternative Implementierung (im folgenden „Variante B“ genannt), bei welcher die Berechnung von `uSquareOfLength` nach dem vorletzten `return`-statement erfolgt. Die Implementierung ist in dem folgenden Quellcodeausschnitt dargestellt.

```
Vector<T, 3> v_vector = cross_product(p1 - p3, intersection - p3);

if (normal.scalar_product(v_vector) < 0.0)
{
  return false;
}

T uSquareOfLength = u_vector.square_of_length();

T vSquareOfLength = v_vector.square_of_length();
T areaSquareSum = normal.square_of_length();
```

Die Tatsächliche Steigerung der Ausführungszeiten ist anhand der folgenden 30 Testdurchläufe (je. 10) zu erkennen. Die Ausführung im Raum E203 war leider nicht möglich, es ist jedoch für die Durchläufe und den Durchschnitt der Speedup berechnet, damit dennoch eine Beurteilung möglich ist.

Messung der Laufzeiten	Ohne Optimierung (ms)	Mit Optimierung (ms)	Speedup (%)	Mit Optimierung (Variante B) (ms)
	max **11.197,00**	min **8.020,97**	max **39,60**	min **7.877,21**
	8.968,57	8.239,86	8,84	9.330,72
	10.860,30	8.869,15	22,45	10.299,40
	10.287,00	8.218,34	25,17	8.266,68
	9.897,65	9.527,56	3,88	8.846,22
	min **8.924,78**	8.501,79	4,98	9.479,69
	10.009,80	max **9.853,52**	min **1,59**	9.049,22
	10.893,80	8.420,95	29,37	9.936,47
	10.045,90	8.585,56	17,01	max **10.876,20**
	9.883,80	8.092,75	22,13	10.564,40
Durch-schnitt	10.096,86	8.633,05	16,96	9.452,62 (6,8% Speedup)

Die Interpretation der Resultate erfolgte bereits beim Betrachten des Assemblercodes und den geänderten Laufzeiten. Durch die Optimierung wurde **die Laufzeit verbessert**, was sich zum einen auf das **Einsparen von Berechnungen** (Quadratwurzel) und zum anderen auf die Verwendung von **SIMD-Befehlen** zurückführen lässt. Dass das Einsparen von Operationen die Laufzeit verbessert, ist trivial. Dass SIMD-Befehle einen Unterschied machen, ist daran zu erkennen, dass die Laufzeit schlechter ist, wenn der raytracer ohne den Parameter `-mfpmath=sse` übersetzt wird.

2 Aufgabe Quadratwurzel

Im Folgenden werden verschiedene Implementierungen einer Quadratwurzelberechnung aufgezeigt. Alle Implementierungen verwenden das Newton-Verfahren zur Annäherung an die Quadratwurzel. Die erste Implementierung (sqrt1) ist das Newton-Verfahren ohne Abwandlungen.

```
1   template <size_t LOOPS = 2>
2   float sqrt1(float *a)
3   {
4     int initial = *reinterpret_cast<int *>(a);
5     initial = (1 << 29) + (initial >> 1) - (1 << 22) - 0x4C000;
6     float result = *reinterpret_cast<float *>(&initial);
7
8     for (int i = 0; i < LOOPS; i++)
9     {
10      result = 0.5 * (result + (*a / result));
11    }
12
13    return result;
14  }
```

Abbildung 2: Sourcecode sqrt1

```
1     result = 0.5 * (result + (*a / result));
2     6e:      48 8b 45 c8           mov    -0x38(%rbp),%rax
3     72:      c5 fa 10 00           vmovss (%rax),%xmm0
4     76:      c5 fa 10 4d d0        vmovss -0x30(%rbp),%xmm1
5     7b:      c5 fa 5e c1           vdivss %xmm1,%xmm0,%xmm0
6     7f:      c5 fa 10 4d d0        vmovss -0x30(%rbp),%xmm1
7     84:      c5 fa 58 c1           vaddss %xmm1,%xmm0,%xmm0
8     88:      c5 fa 10 0d 00 00 00  vmovss 0x0(%rip),%xmm1
9     8f:      00
10    90:      c5 fa 59 c1           vmulss %xmm1,%xmm0,%xmm0
11    94:      c5 fa 11 45 d0        vmovss %xmm0,-0x30(%rbp)
```

Abbildung 3: Assemblercode sqrt1

Für die Implementierung von `sqrt2` musste von der ausführlichen Implementierung für das Berechnen des Startwertes abgewichen werden. Da vier float-Werte iteriert werden, ist es hier relevant, dass Zugriffe auf den Speicher nicht zum Bottleneck werden.

```
1   template <size_t LOOPS = 2>
2   void sqrt2(float *__restrict__ a, float *__restrict__ root)
3   {
4     int *initial = reinterpret_cast<int *>(a);
5     int *root_initial = reinterpret_cast<int *>(root);
6
7     *(root_initial + 0) = (1 << 29) + (*(initial + 0) >> 1) - (1 << 22) - 0x4C000;
8     *(root_initial + 1) = (1 << 29) + (*(initial + 1) >> 1) - (1 << 22) - 0x4C000;
9     *(root_initial + 2) = (1 << 29) + (*(initial + 2) >> 1) - (1 << 22) - 0x4C000;
10    *(root_initial + 3) = (1 << 29) + (*(initial + 3) >> 1) - (1 << 22) - 0x4C000;
11
12    root = reinterpret_cast<float *>(root_initial);
13
14    for (int i = 0; i < LOOPS; i++)
15    {
16      *(root + 0) = 0.5 * (*(root + 0) + *(a + 0) / *(root + 0));
17      *(root + 1) = 0.5 * (*(root + 1) + *(a + 1) / *(root + 1));
18      *(root + 2) = 0.5 * (*(root + 2) + *(a + 2) / *(root + 2));
19      *(root + 3) = 0.5 * (*(root + 3) + *(a + 3) / *(root + 3));
20    }
21  }
```

Abbildung 4: Sourcecode sqrt2

```
1   *(root + 0) = 0.5 * (*(root + 0) + *(a + 0) / *(root + 0));
2   *(root + 1) = 0.5 * (*(root + 1) + *(a + 1) / *(root + 1));
3   *(root + 2) = 0.5 * (*(root + 2) + *(a + 2) / *(root + 2));
4     79:     c7 45 f4 00 00 00 00  movl   $0x0,-0xc(%rbp)
5     80:     8b 45 f4              mov    -0xc(%rbp),%eax
6     83:     48 98                 cltq
7     85:     48 83 f8 01           cmp    $0x1,%rax
8     89:     0f 87 09 01 00 00     ja     198 <_Z5sqrt2ILm2EEvPfS0_+0x198>
```

Abbildung 5: Assemblercode sqrt2

```
1   template <size_t LOOPS = 2>
2   void v4sf_sqrt(v4sf *__restrict__ a, v4sf *__restrict__ root)
3   {
4     v4si *initial = reinterpret_cast<v4si *>(a);
5     v4si *root_initial = reinterpret_cast<v4si *>(root);
6
7     *(root_initial) = (1 << 29) + (*(initial) >> 1) - (1 << 22) - 0x4C000;
8
9     root = reinterpret_cast<v4sf *>(root_initial);
10
11    for (unsigned int i = 0; i < LOOPS; i++)
12    {
13      *(root) = 0.5 * (*root + *a / *root);
14    }
15  }
```

Abbildung 6: Sourcecode sqrt3

```
1     *root = 0.5 * (*root + (*a / *root));
2     9b:      48 8b 45 c0           mov    -0x40(%rbp),%rax
3     9f:      c5 f8 28 08           vmovaps (%rax),%xmm1
4     a3:      48 8b 45 c8           mov    -0x38(%rbp),%rax
5     a7:      c5 f8 28 00           vmovaps (%rax),%xmm0
6     ab:      48 8b 45 c0           mov    -0x40(%rbp),%rax
7     af:      c5 f8 28 10           vmovaps (%rax),%xmm2
8     b3:      c5 f8 5e c2           vdivps %xmm2,%xmm0,%xmm0
9     b7:      c5 f0 58 c8           vaddps %xmm0,%xmm1,%xmm1
10    bb:      c5 f8 28 05 00 00 00  vmovaps 0x0(%rip),%xmm0
11    c2:      00
12    c3:      c5 f0 59 c0           vmulps %xmm0,%xmm1,%xmm0
13    c7:      48 8b 45 c0           mov    -0x40(%rbp),%rax
14    cb:      c5 f8 29 00           vmovaps %xmm0,(%rax)
```

Abbildung 7: Assemblercode sqrt3

Für die Messung der Ausführungszeiten wurden die Funktionen jeweils mit zwei, drei und vier Iterationen ausgeführt. Hierfür wurde der Befehl

```
1  for i in {1..10}; do ./a.out; done
```

verwendet. Die Ergebnisse sind je Iteration in einer eigenen Tabelle. Die Ergebnisse zeigen auf, dass die Standardimplementierung für die Quadratwurzelberechnung immer am langsamsten ist und die sqrt1 Implementierung immer am schnellsten. Der Grund, weswegen diese Implementierung am schnellsten ist, liegt darin begründet, dass lediglich eine Quadratwurzel berechnet wird. Bei den anderen Verfahren werden jeweils mehrere Quadratwurzeln auf einmal berechnet. Bei den Berechnungen von vier Quadratwurzeln ist ebenfalls sqrt1 am schnellsten, was vermutlich an Optimierungen des Compilers (`-O3`) liegt.

Funktion	math sqrt	sqrt1 (one time a loop)	sqrt1 (four times a loop)	sqrt2 (sequence of 4 floats)	sqrt3 (sequence of 4 floats, SIMD)
2 Iterationen (ns)					
1	1.188.919	128.528	170.827	191.229	187.068
2	1.193.868	130.154	167.274	165.383	165.696
3	1.205.189	128.041	173.410	170.120	163.187
4	1.211.284	144.744	196.877	168.399	163.092
5	1.208.240	132.703	166.177	162.219	160.435
6	1.191.343	129.184	165.369	166.951	161.958
7	1.193.755	130.751	171.661	187.112	169.275
8	1.195.859	130.875	169.271	168.126	159.548
9	1.227.180	136.917	188.685	188.328	181.700
10	1.142.834	122.884	161.528	166.944	163.591
Durchschnitt	1.195.847	131.478	173.108	173.481	167.555

Funktion	math sqrt	sqrt1 (one time a loop)	sqrt1 (four times a loop)	sqrt2 (sequence of 4 floats)	sqrt3 (sequence of 4 floats, SIMD)
3 Iterationen (ns)					
1	1.185.187	197.623	211.616	209.728	254.723
2	1.190.135	199.688	210.720	213.492	247.229
3	1.188.263	196.662	214.598	211.306	246.485
4	1.202.075	201.387	211.197	209.774	251.505
5	1.183.808	195.871	207.493	212.030	245.140
6	1.187.731	199.749	209.789	213.371	248.326
7	1.183.374	197.589	221.285	210.320	248.762
8	1.176.765	195.891	205.392	213.244	246.382
9	1.156.323	195.695	221.941	228.915	267.508
10	1.098.437	178.770	201.027	209.291	251.368
Durchschnitt	1.175.210	195.893	211.506	213.147	250.743
4 Iterationen (ns)					
1	1.187.979	260.258	273.736	273.313	360.347
2	1.207.236	274.577	274.959	270.377	355.538
3	1.188.432	264.925	273.877	275.563	350.862
4	1.202.437	261.692	277.422	273.541	372.471
5	1.184.000	271.783	296.699	289.570	359.060
6	1.189.719	260.817	285.013	274.146	355.175
7	1.187.213	258.871	277.212	274.684	357.526
8	1.181.545	259.864	275.899	269.562	381.800
9	1.140.453	245.566	271.053	286.532	370.198
10	1.207.384	265.752	296.215	308.257	390.035
Durchschnitt	1.187.640	262.411	280.209	279.555	365.301

3 Aufgabe k-d-Baum

Durch Aufgabe 1 und Aufgabe 2 wurde die Geschwindigkeit der Schnittpunktberechnung erhöht. In dieser Aufgabe werden nicht die eigentlichen Berechnungen beschleunigt, sondern es wird mithilfe einer Datenstruktur (k-d-Baum) versucht, die Anzahl der erforderlichen Schnittpunktberechnungen zu verringern. Im Folgenden ist der Quellcode aufgeführt, im Anschluss die Messungen der Laufzeit und Schnittpunktberechnungen.

```
1   void BoundingBox::split(BoundingBox &left, BoundingBox &right)
2   {
3     float xLength = abs(max[0] - min[0]);
4     float yLength = abs(max[1] - min[1]);
5     float zLength = abs(max[2] - min[2]);
6
7     left.min = min;
8     right.max = max;
9
10    if (xLength > yLength && xLength > zLength)
11    {
12      float boxWidth = xLength / 2;
13      left.max = Vector<float, 3>{min[0] + boxWidth, max[1], max[2]};
14      right.min = Vector<float, 3>{min[0] + boxWidth, min[1], min[2]};
15    }
16    else if (yLength > zLength)
17    {
18      float boxHeight = xLength / 2;
19      left.max = Vector<float, 3>{max[0], min[1] + boxHeight, max[2]};
20      right.min = Vector<float, 3>{min[0], min[1] + boxHeight, min[2]};
21    }
22    else
23    {
24      float boxDepth = xLength / 2;
25      left.max = Vector<float, 3>{max[0], max[1], min[2] + boxDepth};
26      right.min = Vector<float, 3>{min[0], min[1], min[2] + boxDepth};
27    }
28  }
```

```
29 bool BoundingBox::contains(Vector<FLOAT, 3> v)
30 {
31   return v[0] >= min[0] && v[0] <= max[0] &&
32          v[1] >= min[1] && v[1] <= max[1] &&
33          v[2] >= min[2] && v[2] <= max[2];
34 }
35
36 bool BoundingBox::contains(Triangle<FLOAT> *triangle)
37 {
38   bool p1InBox = contains(triangle->p1);
39   bool p2InBox = contains(triangle->p2);
40   bool p3InBox = contains(triangle->p3);
41
42   return p1InBox || p2InBox || p3InBox;
43 }
```

```
44 bool BoundingBox::intersects(Vector<FLOAT, 3> eye, Vector<FLOAT, 3> direction)
45 {
46   FLOAT tmin[3] = {(min[0] - eye[0]) / direction[0],
47                    (min[1] - eye[1]) / direction[1],
48                    (min[2] - eye[2]) / direction[2]};
49   FLOAT tmax[3] = {(max[0] - eye[0]) / direction[0],
50                    (max[1] - eye[1]) / direction[1],
51                    (max[2] - eye[2]) / direction[2]};
52   FLOAT tminimum = std::min(tmin[0], tmax[0]);
53   FLOAT tmaximum = std::max(tmin[0], tmax[0]);
54   tminimum = std::max(tminimum, std::min(tmin[1], tmax[1]));
55   tmaximum = std::min(tmaximum, std::max(tmin[1], tmax[1]));
56   tminimum = std::max(tminimum, std::min(tmin[2], tmax[2]));
57   tmaximum = std::min(tmaximum, std::max(tmin[2], tmax[2]));
58
59   return tmaximum >= tminimum;
60 }
```

```
61  KDTree *KDTree::buildTree(KDTree *tree, std::vector<Triangle<FLOAT> *> &triangles)
62  {
63    if (triangles.size() < MAX_TRIANGLES_PER_LEAF)
64    {
65      for (auto const &triangle : triangles)
66      {
67        this->triangles.push_back(triangle);
68      }
69      return tree;
70    }
71
72    left = new KDTree();
73    right = new KDTree();
74
75    this->box.split(left->box, right->box);
76
77    std::vector<Triangle<FLOAT> *> leftTriangles, rightTriangles;
78
79    for (auto const &triangle : triangles)
80    {
81      bool isInLeft = left->box.contains(triangle);
82      bool isInRight = right->box.contains(triangle);
83      if (isInLeft && isInRight)
84      {
85        this->triangles.push_back(triangle);
86      }
87      else if (isInLeft)
88      {
89        leftTriangles.push_back(triangle);
90      }
91      else if (isInRight)
92      {
93        rightTriangles.push_back(triangle);
94      }
95    }
96
97    left->buildTree(tree, leftTriangles);
98    right->buildTree(tree, rightTriangles);
99    return tree;
100 }
```

```
101 KDTree *KDTree::buildTree(std::vector<Triangle<FLOAT> *> &triangles)
102 {
103   KDTree *root = new KDTree();
104
105   float FLT_MAX = std::numeric_limits<FLOAT>::max();
106   float FLT_MIN = std::numeric_limits<FLOAT>::min();
107   Vector<FLOAT, 3> boxMin = Vector<FLOAT, 3>{FLT_MAX, FLT_MAX, FLT_MAX};
108   Vector<FLOAT, 3> boxMax = Vector<FLOAT, 3>{FLT_MIN, FLT_MIN, FLT_MIN};
109
110   for (auto triangle : triangles)
111   {
112     for (int i = 0; i < 3; i++)
113     {
114       boxMin[i] = std::min(boxMin[i], triangle->p1[i]);
115       boxMin[i] = std::min(boxMin[i], triangle->p2[i]);
116       boxMin[i] = std::min(boxMin[i], triangle->p3[i]);
117
118       boxMax[i] = std::max(boxMax[i], triangle->p1[i]);
119       boxMax[i] = std::max(boxMax[i], triangle->p2[i]);
120       boxMax[i] = std::max(boxMax[i], triangle->p3[i]);
121     }
122   }
123
124   root->box = BoundingBox(boxMin, boxMax);
125   root->buildTree(root, triangles);
126   return root;
127 }
```

```
128 bool KDTree::hasNearestTriangle(Vector<FLOAT, 3> eye, Vector<FLOAT, 3> direction, Tri-
    angle<FLOAT> *&nearest_triangle, FLOAT &t, FLOAT &u, FLOAT &v, FLOAT minimum_t)
129 {
130   if (!box.intersects(eye, direction))
131   {
132     return false;
133   }
134
135   if (left != nullptr && left->hasNearestTriangle(eye, direction, nearest_triangle, t,
    u, v, minimum_t) && t < minimum_t)
136   {
137     minimum_t = t;
138   }
139
140   if (right != nullptr && right->hasNearestTriangle(eye, direction, nearest_triangle, t,
    u, v, minimum_t) && t < minimum_t)
141   {
142     minimum_t = t;
143   }
144
145   for (Triangle<float> *triangle : this->triangles)
146   {
147     stats.no_ray_triangle_intersection_tests++;
148     if (triangle->intersects(eye, direction, t, u, v, minimum_t))
149     {
150       if (!nearest_triangle || t < minimum_t)
151       {
152         stats.no_ray_triangle_intersections_found++;
153         minimum_t = t;
154         nearest_triangle = triangle;
155       }
156     }
157   }
158
159   t = minimum_t;
160   return nearest_triangle != nullptr;
161 }
162
```

Messung der Laufzeiten	Ohne k-d-Baum (ms)	Mit k-d-Baum (ms)	Speedup (%)
	9.060,64	2.123,12	427
	8.273,15	1.990,80	416
	9.382,45	2.091,99	448
	8.802,72	2.114,73	416
	8.498,99	1.894,21	449
Durchschnitt	8.803,59	2.042,97	431

	Ohne k-d-Baum	Mit k-d-Baum
Anzahl der Berechnungen	519.950.720	130.488.452
Anzahl gefundener Schnittpunkte	35.294	35.894

Die Laufzeit und die Anzahl der benötigten Schnittpunktberechnungen sind deutlich um den gleichen Faktor verringert worden. Die Anzahl der Gefundenen Schnittpunkte ist jedoch gestiegen, was vermutlich darauf zurückzuführen ist, dass die bestehende Implementierung lediglich Schnittpunkte zählt, welche näher sind und die entwickelte k-d-Baum Lösung somit manche Schnittpunktberechnungen in der Statistik aufführt, welche zuvor übersprungen wurden.