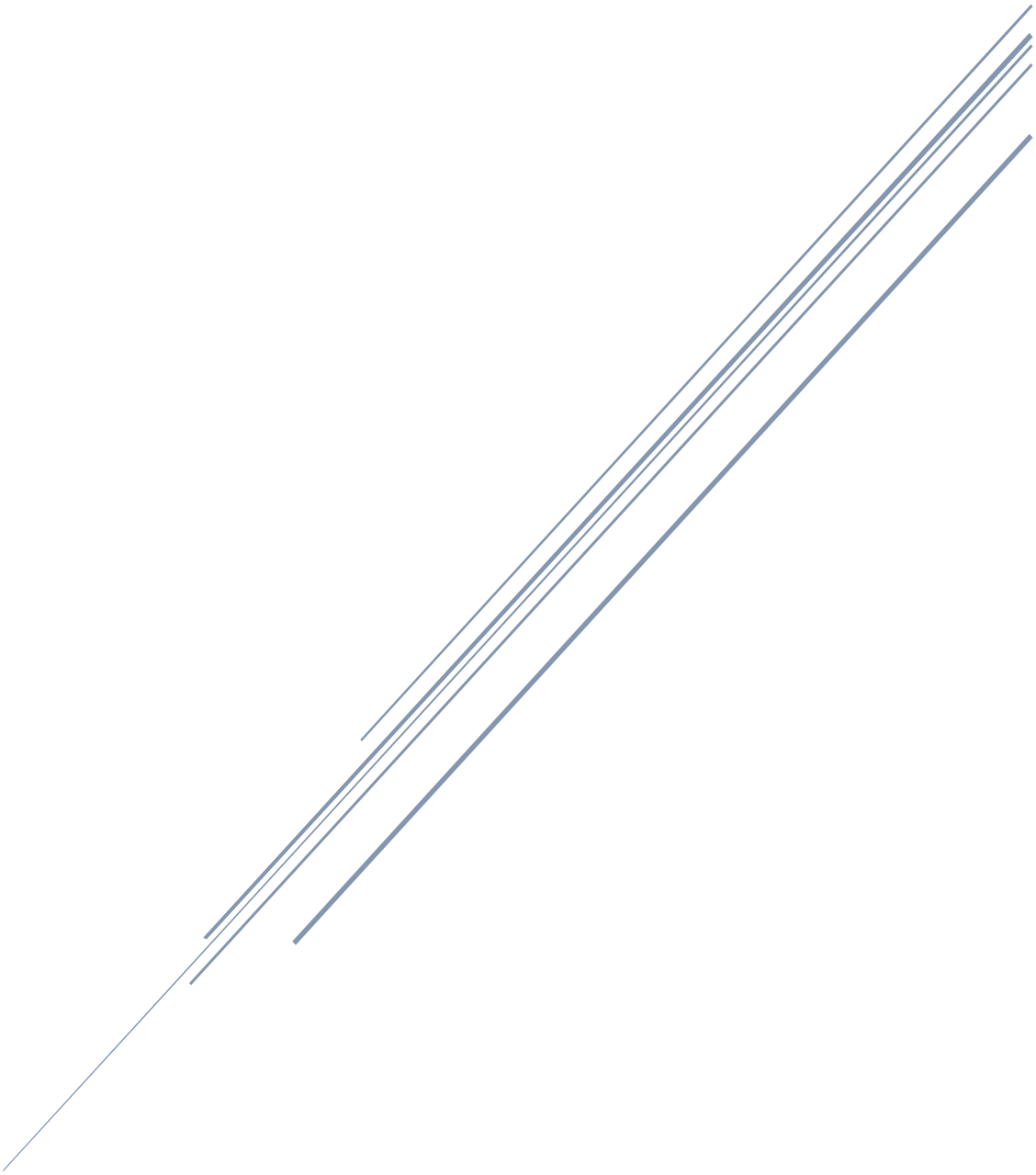
BERICHT OPTIMIERUNG

Von Rebecca Sigmund Matr.Nr. 64146



Inhaltsverzeichnis

Aufgabe 1: Schnittpunkttest optimieren

Quelltext

Ausgangsversion

```
bool intersects(Vector<T,3> origin, Vector<T,3>
      direction, FLOAT &t, FLOAT &u, FLOAT &v, FLOAT
      minimum_t = INFINITY) {

// Normale des Dreiecks bestimmen
 Vector<T, 3> normal =  cross_product(p2 - p1, p3  -
p1);

 T normalRayProduct = normal.scalar_product(
direction );

// used for u-v-parameter calculation
 T area = normal.length();

// Ist die Richtung parallel zum Dreieck?
 if ( fabs(normalRayProduct) < EPSILON ) {
  return false;
 }

 T d = normal.scalar_product( p1 );
// Wie oft wird direction benötigt, um von origin die
 // Ebene des Dreiecks zu schneiden
 t = (d - normal.scalar_product( origin ) ) /
normalRayProduct;



// Ist Dreieck in der falschen Richtung?
 if ( t < 0.0 ) {
   return false;
 }

// Der Schnittpunkt von origin + direction mit der
Ebene des Dreiecks
 Vector<T, 3> intersection = origin + t * direction;

 // Ist der Schnittpunkt innerhalb des Dreiecks?
 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;
 }
```

Optimierte Version

```
bool intersects(Vector<T,3> origin, Vector<T,3>
      direction, FLOAT &t, FLOAT &u, FLOAT &v, FLOAT
      minimum_t) {

  // Normale des Dreiecks bestimmen
  Vector<T, 3> normal =  cross_product(p2 - p1, p3  -
p1);

  T normalRayProduct = normal.scalar_product(
direction );



  // Ist die Richtung parallel zum Dreieck?
  if ( fabs(normalRayProduct) < EPSILON ) {
    return false;
  }

  T d = normal.scalar_product( p1 );
  // Wie oft wird direction benötigt, um von origin die
  // Ebene des Dreiecks zu schneiden
  t = (d - normal.scalar_product( origin ) ) /
normalRayProduct;

  // Ist das Dreieck in der falschen Richtung? Oder gibt
es schon ein anderes
  // Dreieck, welches weiter vorne liegt?
  if ( t < 0.0 || t > minimum_t) {
    return false;
  }

  // Der Schnittpunkt von origin + direction mit der
Ebene des Dreiecks
  Vector<T, 3> intersection = origin + t * direction;

  // Ist der Schnittpunkt innerhalb des Dreiecks?
  Vector<T, 3> vector1 = cross_product(p2 - p1,
intersection - p1 );
  if ( normal.scalar_product(vector1) < 0.0 ) {
    return false;
  }

  vector1 = cross_product(p3 - p2,  intersection - p2 );
  if ( normal.scalar_product(vector1) < 0.0 ) {
    return false;
  }
```

u = vector.length() / area; vector = cross_product(p1 - p3, intersection - p3); if (normal.scalar_product(vector) < 0.0) { return false; } v = vector.length() / area; return true; }	Vector<T, 3> vector2 = cross_product(p1 - p3, intersection - p3); if (normal.scalar_product(vector2) < 0.0) { return false; } *// u und v berechnen. Wurzel jeweils erst am Ende ziehen* T area = normal.square_of_length(); u = sqrt(vector1.square_of_length() / area); v = sqrt(vector2.square_of_length() / area); return true; }

Mit

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

wird überprüft, ob bereits ein Dreieck, das vor dem zu prüfendenden Dreieck liegt, gefunden wurde. In diesem Fall muss nicht weiter gerechnet werden.

Die Berechnung der Parameter u und v wurde an das Ende der Methode verschoben, um die Parameter nicht unnötig zu berechnen, falls die Methode frühzeitig abgebrochen wird.

Auch die Berechnung des Parameters area wurde ans Ende der Methode geschoben. Außerdem wird nicht direkt die Länge bestimmt, sondern zunächst das Quadrat der Länge. Somit wird eine Quadratwurzel-Berechnung eingespart.

Assemblercode

```
Vector<T, 3> normal =  cross_product(p2 - p1, p3  - p1);

    T normalRayProduct = normal.scalar_product( direction );

    // Ist die Richtung parallel zum Dreieck?
    if ( fabs(normalRayProduct) < EPSILON ) {
     9ea:    c5 7b 10 35 70 03 00      vmovsd 0x370(%rip),%xmm14        # d62
<_Z8raytraceR6CameraR5SceneR6ScreenP6KDTree+0x5e2>
     9f1:    00
     9f2:    49 8d 1c c1               lea    (%r9,%rax,8),%rbx
    stats.no_ray_triangle_intersection_tests++;
     9f6:    48 83 07 01               addq   $0x1,(%rdi)
      difference.x[i] = this->x[i] - subtract.x[i];
     9fa:    c5 fa 10 6b 1c            vmovss 0x1c(%rbx),%xmm5
     9ff:    c5 fa 10 4b 04            vmovss 0x4(%rbx),%xmm1
     a04:    c5 d2 5c d9               vsubss %xmm1,%xmm5,%xmm3
     a08:    c5 7a 10 43 08            vmovss 0x8(%rbx),%xmm8
     a0d:    c5 fa 11 6c 24 40         vmovss %xmm5,0x40(%rsp)
     a13:    c5 fa 10 6b 14            vmovss 0x14(%rbx),%xmm5
     a18:    c4 c1 52 5c f0            vsubss %xmm8,%xmm5,%xmm6
     a1d:    c5 7a 10 6b 18            vmovss 0x18(%rbx),%xmm13
     a22:    c5 fa 10 53 0c            vmovss 0xc(%rbx),%xmm2
     a27:    c5 fa 10 3b               vmovss (%rbx),%xmm7
     a2b:    c5 12 5c df               vsubss %xmm7,%xmm13,%xmm11
     a2f:    c5 7a 11 6c 24 28         vmovss %xmm13,0x28(%rsp)
     a35:    c5 6a 5c cf               vsubss %xmm7,%xmm2,%xmm9
     a39:    c5 7a 10 6b 20            vmovss 0x20(%rbx),%xmm13
     a3e:    c5 fa 11 54 24 4c         vmovss %xmm2,0x4c(%rsp)
     a44:    c4 c1 12 5c c0            vsubss %xmm8,%xmm13,%xmm0

template <class T>
Vector<T, 3> cross_product(Vector<T, 3> v1, Vector<T, 3> v2) {
  Vector<T, 3> cross;

  cross[0] = v1[1] * v2[2] - v1[2] * v2[1];
     a49:    c5 ca 59 e3               vmulss %xmm3,%xmm6,%xmm4
      difference.x[i] = this->x[i] - subtract.x[i];
     a4d:    c5 fa 10 53 10            vmovss 0x10(%rbx),%xmm2
     a52:    c5 6a 5c d1               vsubss %xmm1,%xmm2,%xmm10
     a56:    c5 7a 11 6c 24 30         vmovss %xmm13,0x30(%rsp)
  cross[0] = v1[1] * v2[2] - v1[2] * v2[1];
     a5c:    c4 e2 29 bb e0            vfmsub231ss %xmm0,%xmm10,%xmm4
  cross[1] = v1[2] * v2[0] - v1[0] * v2[2];
     a61:    c5 b2 59 c0               vmulss %xmm0,%xmm9,%xmm0
     a65:    c4 c2 49 bb c3            vfmsub231ss %xmm11,%xmm6,%xmm0
  cross[2] = v1[0] * v2[1] - v1[1] * v2[0];
     a6a:    c4 41 2a 59 db            vmulss %xmm11,%xmm10,%xmm11
     a6f:    c4 c2 21 9b d9            vfmsub132ss %xmm9,%xmm11,%xmm3
      product += this->x[i] * factor.x[i];
     a74:    c5 7a 10 5c 24 48         vmovss 0x48(%rsp),%xmm11
     a7a:    c4 62 19 99 dc            vfmadd132ss %xmm4,%xmm12,%xmm11
     a7f:    c4 62 79 b9 5c 24 38      vfmadd231ss 0x38(%rsp),%xmm0,%xmm11
     a86:    c4 42 61 b9 df            vfmadd231ss %xmm15,%xmm3,%xmm11
     a8b:    c4 41 78 28 eb            vmovaps %xmm11,%xmm13
     a90:    c5 10 54 2d 60 03 00      vandps 0x360(%rip),%xmm13,%xmm13        # df8
<_Z8raytraceR6CameraR5SceneR6ScreenP6KDTree+0x678>
     a97:    00
     a98:    c4 41 12 5a ed            vcvtss2sd %xmm13,%xmm13,%xmm13
     a9d:    c4 41 79 2e f5            vucomisd %xmm13,%xmm14
```

```
     aa2:    0f 87 5e 02 00 00         ja     d06
<_Z8raytraceR6CameraR5SceneR6ScreenP6KDTree+0x586>
     aa8:    c5 78 28 f4               vmovaps %xmm4,%xmm14
     aac:    c4 62 19 99 f7            vfmadd132ss %xmm7,%xmm12,%xmm14
     ab1:    c4 41 78 28 ee            vmovaps %xmm14,%xmm13
     ab6:    c5 7a 10 74 24 60         vmovss 0x60(%rsp),%xmm14
     abc:    c4 62 79 b9 e9            vfmadd231ss %xmm1,%xmm0,%xmm13
     ac1:    c4 62 19 99 f4            vfmadd132ss %xmm4,%xmm12,%xmm14
     ac6:    c4 42 61 b9 e8            vfmadd231ss %xmm8,%xmm3,%xmm13
     acb:    c4 62 79 b9 74 24 5c      vfmadd231ss 0x5c(%rsp),%xmm0,%xmm14
     ad2:    c4 62 61 b9 74 24 58      vfmadd231ss 0x58(%rsp),%xmm3,%xmm14
      return false;
    }

    T d = normal.scalar_product( p1 );
    // Wie oft wird direction benötigt, um von origin die Ebene des Dreiecks zu schneiden
    t = (d - normal.scalar_product( origin ) ) / normalRayProduct;
     ad9:    c4 41 12 5c ee            vsubss %xmm14,%xmm13,%xmm13
     ade:    c4 41 12 5e db            vdivss %xmm11,%xmm13,%xmm11

    // Ist das Dreieck in der falschen Richtung? Oder gibt es schon ein anderes Dreieck,
welches weiter vorne liegt?
    if ( t < 0.0 || t > minimum_t) {
     ae3:    c4 41 78 2e e3            vucomiss %xmm11,%xmm12
     ae8:    0f 87 18 02 00 00         ja     d06
<_Z8raytraceR6CameraR5SceneR6ScreenP6KDTree+0x586>
     aee:    c5 78 2e 5c 24 50         vucomiss 0x50(%rsp),%xmm11
     af4:    0f 87 0c 02 00 00         ja     d06
<_Z8raytraceR6CameraR5SceneR6ScreenP6KDTree+0x586>
      sum.x[i] = this->x[i] + addend.x[i];
     afa:    c5 7a 10 6c 24 48         vmovss 0x48(%rsp),%xmm13
     b00:    c4 62 21 a9 6c 24 60      vfmadd213ss 0x60(%rsp),%xmm11,%xmm13
      difference.x[i] = this->x[i] - subtract.x[i];
     b07:    c5 7a 11 6c 24 74         vmovss %xmm13,0x74(%rsp)
     b0d:    c5 12 5c ef               vsubss %xmm7,%xmm13,%xmm13
      sum.x[i] = this->x[i] + addend.x[i];
     b11:    c5 7a 10 74 24 38         vmovss 0x38(%rsp),%xmm14
     b17:    c4 62 21 a9 74 24 5c      vfmadd213ss 0x5c(%rsp),%xmm11,%xmm14
     b1e:    c5 7a 11 74 24 64         vmovss %xmm14,0x64(%rsp)
     b24:    c4 41 78 28 f7            vmovaps %xmm15,%xmm14
      difference.x[i] = this->x[i] - subtract.x[i];
     b29:    c5 7a 11 6c 24 68         vmovss %xmm13,0x68(%rsp)
      sum.x[i] = this->x[i] + addend.x[i];
     b2f:    c4 62 21 a9 74 24 58      vfmadd213ss 0x58(%rsp),%xmm11,%xmm14
      difference.x[i] = this->x[i] - subtract.x[i];
     b36:    c4 41 0a 5c f8            vsubss %xmm8,%xmm14,%xmm15
     b3b:    c5 7a 10 6c 24 64         vmovss 0x64(%rsp),%xmm13
     b41:    c5 12 5c e9               vsubss %xmm1,%xmm13,%xmm13
  cross[0] = v1[1] * v2[2] - v1[2] * v2[1];
     b45:    c5 7a 11 6c 24 78         vmovss %xmm13,0x78(%rsp)
     b4b:    c4 41 4a 59 ed            vmulss %xmm13,%xmm6,%xmm13
     b50:    c4 42 29 bb ef            vfmsub231ss %xmm15,%xmm10,%xmm13
  cross[2] = v1[0] * v2[1] - v1[1] * v2[0];
     b55:    c5 2a 59 54 24 68         vmulss 0x68(%rsp),%xmm10,%xmm10
      product += this->x[i] * factor.x[i];
     b5b:    c4 62 19 99 ec            vfmadd132ss %xmm4,%xmm12,%xmm13
  cross[1] = v1[2] * v2[0] - v1[0] * v2[2];
     b60:    c4 41 32 59 ff            vmulss %xmm15,%xmm9,%xmm15
     b65:    c4 e2 01 9b 74 24 68      vfmsub132ss 0x68(%rsp),%xmm15,%xmm6
      product += this->x[i] * factor.x[i];
     b6c:    c4 e2 11 99 f0            vfmadd132ss %xmm0,%xmm13,%xmm6
```

```
  cross[2] = v1[0] * v2[1] - v1[1] * v2[0];
     b71:    c4 62 31 bb 54 24 78      vfmsub231ss 0x78(%rsp),%xmm9,%xmm10
      product += this->x[i] * factor.x[i];
     b78:    c4 c2 61 b9 f2            vfmadd231ss %xmm10,%xmm3,%xmm6
    // Der Schnittpunkt der direction von origin aus mit der Ebene des Dreiecks
    Vector<T, 3> intersection = origin + t * direction;

    // Ist der Schnittpunkt innerhalb des Dreiecks?
    Vector<T, 3> vector1 = cross_product(p2 - p1,  intersection - p1 );
    if ( normal.scalar_product(vector1) < 0.0 ) {
     b7d:    c5 78 2e e6               vucomiss %xmm6,%xmm12
     b81:    0f 87 7f 01 00 00         ja     d06
<_Z8raytraceR6CameraR5SceneR6ScreenP6KDTree+0x586>
      difference.x[i] = this->x[i] - subtract.x[i];
     b87:    c5 7a 10 6c 24 74         vmovss 0x74(%rsp),%xmm13
     b8d:    c5 7a 10 7c 24 4c         vmovss 0x4c(%rsp),%xmm15
     b93:    c4 41 12 5c d7            vsubss %xmm15,%xmm13,%xmm10
     b98:    c5 7a 10 6c 24 28         vmovss 0x28(%rsp),%xmm13
     b9e:    c4 41 12 5c ff            vsubss %xmm15,%xmm13,%xmm15
     ba3:    c5 fa 10 74 24 64         vmovss 0x64(%rsp),%xmm6
     ba9:    c5 7a 10 6c 24 40         vmovss 0x40(%rsp),%xmm13
     baf:    c5 4a 5c ca               vsubss %xmm2,%xmm6,%xmm9
     bb3:    c5 92 5c d2               vsubss %xmm2,%xmm13,%xmm2
     bb7:    c5 7a 10 6c 24 30         vmovss 0x30(%rsp),%xmm13
     bbd:    c5 8a 5c f5               vsubss %xmm5,%xmm14,%xmm6
     bc1:    c5 92 5c ed               vsubss %xmm5,%xmm13,%xmm5
  cross[0] = v1[1] * v2[2] - v1[2] * v2[1];
     bc5:    c4 41 52 59 e9            vmulss %xmm9,%xmm5,%xmm13
     bca:    c4 62 69 bb ee            vfmsub231ss %xmm6,%xmm2,%xmm13
  cross[2] = v1[0] * v2[1] - v1[1] * v2[0];
     bcf:    c4 c1 6a 59 d2            vmulss %xmm10,%xmm2,%xmm2
  cross[1] = v1[2] * v2[0] - v1[0] * v2[2];
     bd4:    c4 c1 4a 59 f7            vmulss %xmm15,%xmm6,%xmm6
  cross[2] = v1[0] * v2[1] - v1[1] * v2[0];
     bd9:    c4 42 69 9b cf            vfmsub132ss %xmm15,%xmm2,%xmm9
      product += this->x[i] * factor.x[i];
     bde:    c5 78 28 fc               vmovaps %xmm4,%xmm15
  cross[1] = v1[2] * v2[0] - v1[0] * v2[2];
     be2:    c4 c2 49 9b ea            vfmsub132ss %xmm10,%xmm6,%xmm5
      product += this->x[i] * factor.x[i];
     be7:    c4 42 19 99 fd            vfmadd132ss %xmm13,%xmm12,%xmm15
     bec:    c5 78 29 fa               vmovaps %xmm15,%xmm2
     bf0:    c4 e2 79 b9 d5            vfmadd231ss %xmm5,%xmm0,%xmm2
     bf5:    c4 c2 61 b9 d1            vfmadd231ss %xmm9,%xmm3,%xmm2
      return false;
    }

    vector1 = cross_product(p3 - p2,  intersection - p2 );
    if ( normal.scalar_product(vector1) < 0.0 ) {
     bfa:    c5 78 2e e2               vucomiss %xmm2,%xmm12
     bfe:    0f 87 02 01 00 00         ja     d06
<_Z8raytraceR6CameraR5SceneR6ScreenP6KDTree+0x586>
      difference.x[i] = this->x[i] - subtract.x[i];
     c04:    c5 fa 10 54 24 74         vmovss 0x74(%rsp),%xmm2
     c0a:    c5 3a 5c 44 24 30         vsubss 0x30(%rsp),%xmm8,%xmm8
     c10:    c5 ea 5c 74 24 28         vsubss 0x28(%rsp),%xmm2,%xmm6
     c16:    c5 7a 10 7c 24 40         vmovss 0x40(%rsp),%xmm15
     c1c:    c5 fa 10 54 24 64         vmovss 0x64(%rsp),%xmm2
     c22:    c5 c2 5c 7c 24 28         vsubss 0x28(%rsp),%xmm7,%xmm7
     c28:    c4 41 6a 5c d7            vsubss %xmm15,%xmm2,%xmm10
     c2d:    c5 8a 5c 54 24 30         vsubss 0x30(%rsp),%xmm14,%xmm2
```

```
     c33:    c4 c1 72 5c cf           vsubss %xmm15,%xmm1,%xmm1
  cross[0] = v1[1] * v2[2] - v1[2] * v2[1];
     c38:    c4 41 3a 59 f2           vmulss %xmm10,%xmm8,%xmm14
     c3d:    c4 62 71 bb f2           vfmsub231ss %xmm2,%xmm1,%xmm14
  cross[2] = v1[0] * v2[1] - v1[1] * v2[0];
     c42:    c5 f2 59 ce              vmulss %xmm6,%xmm1,%xmm1
  cross[1] = v1[2] * v2[0] - v1[0] * v2[2];
     c46:    c5 c2 59 d2              vmulss %xmm2,%xmm7,%xmm2
  cross[2] = v1[0] * v2[1] - v1[1] * v2[0];
     c4a:    c4 c2 71 9b fa           vfmsub132ss %xmm10,%xmm1,%xmm7
      product += this->x[i] * factor.x[i];
     c4f:    c5 f8 28 cc              vmovaps %xmm4,%xmm1
  cross[1] = v1[2] * v2[0] - v1[0] * v2[2];
     c53:    c4 62 69 9b c6           vfmsub132ss %xmm6,%xmm2,%xmm8
      product += this->x[i] * factor.x[i];
     c58:    c4 c2 19 99 ce           vfmadd132ss %xmm14,%xmm12,%xmm1
     c5d:    c4 c2 79 b9 c8           vfmadd231ss %xmm8,%xmm0,%xmm1
     c62:    c4 e2 61 b9 cf           vfmadd231ss %xmm7,%xmm3,%xmm1
      return false;
    }

    Vector<T, 3> vector2 = cross_product(p1 - p3, intersection - p3 );
    if (normal.scalar_product(vector2) < 0.0 ) {
     c67:    c5 78 2e e1              vucomiss %xmm1,%xmm12
     c6b:    0f 87 95 00 00 00        ja     d06
<_Z8raytraceR6CameraR5SceneR6ScreenP6KDTree+0x586>
      square_of_length += ( this->x[i] * this->x[i] );
     c71:    c4 e2 19 99 e4           vfmadd132ss %xmm4,%xmm12,%xmm4
     c76:    c4 42 19 99 ed           vfmadd132ss %xmm13,%xmm12,%xmm13
     c7b:    c4 e2 59 99 c0           vfmadd132ss %xmm0,%xmm4,%xmm0
     c80:    c4 e2 11 99 ed           vfmadd132ss %xmm5,%xmm13,%xmm5
     c85:    c4 e2 79 99 db           vfmadd132ss %xmm3,%xmm0,%xmm3
     c8a:    c4 42 51 99 c9           vfmadd132ss %xmm9,%xmm5,%xmm9
      return false;
    }

    // u und v berechnen. Wurzel jeweils erst am Ende ziehen
    T area = normal.square_of_length();
    u = sqrt(vector1.square_of_length() / area);
     c8f:    c5 b2 5e c3              vdivss %xmm3,%xmm9,%xmm0
     c93:    c5 f1 57 c9              vxorpd %xmm1,%xmm1,%xmm1
     c97:    c5 fa 5a c0              vcvtss2sd %xmm0,%xmm0,%xmm0
     c9b:    c5 f9 2e c8              vucomisd %xmm0,%xmm1
     c9f:    c5 cb 51 f0              vsqrtsd %xmm0,%xmm6,%xmm6
     ca3:    0f 87 fd 08 00 00        ja     15a6
<_Z8raytraceR6CameraR5SceneR6ScreenP6KDTree+0xe26>
     ca9:    c4 42 19 99 f6           vfmadd132ss %xmm14,%xmm12,%xmm14
     cae:    c4 42 09 99 c0           vfmadd132ss %xmm8,%xmm14,%xmm8
     cb3:    c4 e2 39 99 ff           vfmadd132ss %xmm7,%xmm8,%xmm7
    v = sqrt(vector2.square_of_length() / area);
     cb8:    c5 c2 5e db              vdivss %xmm3,%xmm7,%xmm3
    u = sqrt(vector1.square_of_length() / area);
     cbc:    c5 cb 5a f6              vcvtsd2ss %xmm6,%xmm6,%xmm6
    v = sqrt(vector2.square_of_length() / area);
     cc0:    c5 e2 5a db              vcvtss2sd %xmm3,%xmm3,%xmm3
     cc4:    c5 f9 2e cb              vucomisd %xmm3,%xmm1
     cc8:    c5 c3 51 fb              vsqrtsd %xmm3,%xmm7,%xmm7
     ccc:    0f 87 b2 08 00 00        ja     1584
<_Z8raytraceR6CameraR5SceneR6ScreenP6KDTree+0xe04>
```

Zeitmessung

Zeitmessung in E203 mit dem Befehl

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

bzw.

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

Durchlauf	Zeit ohne Optimierung (in Sekunden)	Zeit nach Optimierung (in Sekunden)
1.	8.21939	6.87551
2.	8.09439	6.90676
3.	8.21939	6.96924
4.	8.07875	6.87547
5.	8.17247	6.87547
6.	8.04746	6.92238
7.	8.09433	6.84424
8.	8.07870	6.90674
9.	8.14120	6.84424
10.	8.06309	6.89112
Durchschnitt	8.120917	6.891117

Differenz des Durchschnitts: 1.2298 s

Interpretation

Die Zeit für einen Durchlauf hat sich um mehr als eine Sekunde verbessert. Dies liegt vor allem daran, dass in der nicht optimierten Version, jedes Mal eine Wurzel zu Beginn der Methode gezogen wurde. Dies geschah in der Zeile: `T area = normal.length();` Dies wurde bei einer Auflösung von 256x256 insgesamt 519.950.720 Mal ausgeführt. In den meisten Fällen wurde das Ergebnis dieser Berechnung jedoch nicht weiterverarbeitet, sondern die Methode wurde vorzeitig verlassen, weil sich kein Schnittpunkt ergab. Nur in 35.294 Fällen wird das Ergebnis benötigt, deshalb wurde die Zeile an das Ende der Methode verschoben und wird nun nur in diesen Fällen ausgeführt.

Um weitere Zeit zu sparen, wurden die drei Quadratwurzeln durch zwei ersetzt. Dazu wurden zunächst die Längen der Vektoren im Quadrat berechnet und erst am Schluss die Wurzel der berechneten Division gezogen. Dies war möglich, da folgende Rechenregel gilt:

$$\frac{\sqrt{a}}{\sqrt{b}} = \sqrt{\frac{a}{b}}$$

Weitere Zeit konnte dadurch eingespart werden, dass ein Dreieck nicht mehr überprüft wird, wenn bereits ein näheres, davor liegendes Dreieck gefunden wurde. Dies geschieht in der Zeile

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

Hierdurch hat sich die Anzahl der gefunden Schnittpunkte von 38.215 auf 35.294 reduziert. In diesen 2.921 eingesparten Fällen, konnte die Methode nach dem Vergleich von t und minimum_t abgebrochen werden und die Quadratwurzeln mussten nicht berechnet werden.

Aufgabe 2: Quadratwurzel

Quelltext

In sqrt1 wird jede Quadratwurzel-Berechnung sequenziell ausgeführt. Die beiden Float-Pointer werden zunächst zu Int-Pointern gecastet. Anschließend wird mit ihnen ein Startwert für das Newton-Verfahren berechnet. Ausgehend von diesem Startwert wird das Newton-Verfahren so oft durchgeführt, wie die Template-Variable „LOOPS“ angibt. Danach wird das Ergebnis zurückgegeben.

```
template <size_t LOOPS = 2>
float sqrt1(float * a) {
    float root = 0;
    // a zu int casten
    int * ai = reinterpret_cast<int *>(a) ;
    // initial berechnen
    int * initial = reinterpret_cast<int *>( &root ) ;
    * initial = (1 << 29) + (*ai >> 1) - (1 << 22) - 0x4C000;
    // Newton Verfahren durchfuehren
    for (unsigned int j = 0; j < LOOPS; j++) {
        root = 0.5 * ( root + (* a / root));
    }
    return root;
}
```

Sqrt2 setzt die gleiche Funktionalität, wie Sqrt1 um, jedoch wird nicht eine Variable mitgegeben, sondern ein Array aus vier Werten. Für jeden der vier Werte wird das Newton-Verfahren durchgeführt und die Ergebnisse in einem weiteren Array gespeichert.

```
template <size_t LOOPS = 2>
void sqrt2(float * __restrict__ a, float * __restrict__ root) {
    // a zu int casten
    int * ai = reinterpret_cast<int *>(a) ;
    // initial berechnen
    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;
    // Newton Verfahren durchfuehren
    for (unsigned int j = 0; j < LOOPS; j++) {
        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]));
    }
}
```

Auch in Sqrt3 werden die Werte als vierer Array mitgegeben. Diese werden dann zu einem Vector gecastet, um immer vier Werte auf einmal zu verarbeiten. Mit diesem Vector wird dann das Newton-Verfahren auf vier Werte gleichzeitig angewandt.

```
template <size_t LOOPS = 2>
void v4sf_sqrt(v4sf *  __restrict__  a, v4sf *  __restrict__  root) {
    // a zu int casten
    v4si * ai = reinterpret_cast<v4si *>(a) ;
    // initial berechnen
    v4si * initial = reinterpret_cast<v4si *>( root ) ;
    * initial = (1 << 29) + (*ai >> 1) - (1 << 22) - 0x4C000;
    // Newton Verfahren durchfuehren
    for (unsigned int j = 0; j < LOOPS; j++) {
        * root = 0.5 * ( * root + (* a / * root));
    }
}

// wrapper für v4sf_sqrt
template <size_t LOOPS = 2>
void sqrt3(float *  __restrict__  a, float *  __restrict__  root) {
  v4sf *as =  reinterpret_cast<v4sf *>(a);
  v4sf_sqrt<LOOPS>(as, reinterpret_cast<v4sf *>(root) );
}
```

Assemblercode

```
$ cd /cygdrive/c/Users/RebeccaS/Documents/raytracer/src
g++ -Wall -pedantic -march=native -mfpmath=sse -mavx2 -O3 -c -g sqrt_opt.cc
objdump -S sqrt_opt.o > sqrt_opt.s
```

Sqrt1 Die Additionen werden einzeln durchgeführt	* initial = (1 << 29) + (*ai >> 1) - (1 << 22) - 0x4C000; 439: c4 e2 6a f7 84 24 84 sarx %edx,0x84(%rsp),%eax 440: 00 00 00 443: c5 fa 10 a4 24 84 00 vmovss 0x84(%rsp),%xmm4 44a: 00 00 44c: 05 00 40 bb 1f add $0x1fbb4000,%eax 451: 83 f9 02 cmp $0x2,%ecx 454: 89 44 24 20 mov %eax,0x20(%rsp) 458: c5 f9 6e 74 24 20 vmovd 0x20(%rsp),%xmm6 root = 0.5 * (root + (* a / root)); 45e: c5 da 5e ee vdivss %xmm6,%xmm4,%xmm5 462: c5 d2 58 ee vaddss %xmm6,%xmm5,%xmm5 466: c5 d2 59 e8 vmulss %xmm0,%xmm5,%xmm5 46a: c5 da 5e e5 vdivss %xmm5,%xmm4,%xmm4 46e: c5 da 58 e5 vaddss %xmm5,%xmm4,%xmm4 472: c5 da 59 e0 vmulss %xmm0,%xmm4,%xmm4 476: c5 fa 11 a4 24 84 35 vmovss %xmm4,0xc3584(%rsp) 47d: 0c 00 47f: 0f 84 68 10 00 00 je 14ed <_Z17measure_sqrt_timeILm2EEvv+0x14ed>
Sqrt2 SIMD-Befehle werden genutzt (256-Bit Register packed single precision)	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; // Newton Verfahren durchfuehren for (unsigned int j = 0; j < LOOPS; j++) { root[0] = 0.5 * (root[0] + (a[0] / root[0])); e98: c5 54 5e ce vdivps %ymm6,%ymm5,%ymm9 e9c: c5 34 58 ce vaddps %ymm6,%ymm9,%ymm9 initial[1] = (1 << 29) + (ai[1] >> 1) - (1 << 22) - 0x4C000; ea0: c5 cd 72 e4 01 vpsrad $0x1,%ymm4,%ymm6 ea5: c5 cd fe f3 vpaddd %ymm3,%ymm6,%ymm6 root[1] = 0.5 * (root[1] + (a[1] / root[1])); ea9: c5 dc 5e fe vdivps %ymm6,%ymm4,%ymm7 root[0] = 0.5 * (root[0] + (a[0] / root[0])); ead: c5 34 59 ca vmulps %ymm2,%ymm9,%ymm9 eb1: c4 c1 54 5e e9 vdivps %ymm9,%ymm5,%ymm5 root[1] = 0.5 * (root[1] + (a[1] / root[1])); eb6: c5 c4 58 f6 vaddps %ymm6,%ymm7,%ymm6 initial[2] = (1 << 29) + (ai[2] >> 1) - (1 << 22) - 0x4C000; eba: c5 c5 72 e1 01 vpsrad $0x1,%ymm1,%ymm7 ebf: c5 c5 fe fb vpaddd %ymm3,%ymm7,%ymm7 root[1] = 0.5 * (root[1] + (a[1] / root[1])); ec3: c5 cc 59 f2 vmulps %ymm2,%ymm6,%ymm6 root[2] = 0.5 * (root[2] + (a[2] / root[2])); ec7: c5 74 5e c7 vdivps %ymm7,%ymm1,%ymm8 root[0] = 0.5 * (root[0] + (a[0] / root[0]));

	ecb: c4 c1 54 58 e9 vaddps %ymm9,%ymm5,%ymm5 ed0: c5 d4 59 ea vmulps %ymm2,%ymm5,%ymm5 root[1] = 0.5 * (root[1] + (a[1] / root[1])); ed4: c5 dc 5e e6 vdivps %ymm6,%ymm4,%ymm4 root[2] = 0.5 * (root[2] + (a[2] / root[2])); ed8: c5 3c 58 c7 vaddps %ymm7,%ymm8,%ymm8 initial[3] = (1 << 29) + (ai[3] >> 1) - (1 << 22) - 0x4C000; edc: c5 c5 72 e0 01 vpsrad $0x1,%ymm0,%ymm7 ee1: c5 c5 fe fb vpaddd %ymm3,%ymm7,%ymm7 root[2] = 0.5 * (root[2] + (a[2] / root[2])); ee5: c5 3c 59 c2 vmulps %ymm2,%ymm8,%ymm8 root[3] = 0.5 * (root[3] + (a[3] / root[3])); ee9: c5 7c 5e d7 vdivps %ymm7,%ymm0,%ymm10 root[1] = 0.5 * (root[1] + (a[1] / root[1])); eed: c5 dc 58 e6 vaddps %ymm6,%ymm4,%ymm4 ef1: c5 dc 59 f2 vmulps %ymm2,%ymm4,%ymm6 root[2] = 0.5 * (root[2] + (a[2] / root[2])); ef5: c4 c1 74 5e c8 vdivps %ymm8,%ymm1,%ymm1 root[3] = 0.5 * (root[3] + (a[3] / root[3])); efa: c5 ac 58 ff vaddps %ymm7,%ymm10,%ymm7 efe: c5 c4 59 fa vmulps %ymm2,%ymm7,%ymm7 f02: c5 fc 5e c7 vdivps %ymm7,%ymm0,%ymm0 root[2] = 0.5 * (root[2] + (a[2] / root[2])); f06: c4 c1 74 58 c8 vaddps %ymm8,%ymm1,%ymm1 f0b: c5 f4 59 ca vmulps %ymm2,%ymm1,%ymm1 root[3] = 0.5 * (root[3] + (a[3] / root[3])); f0f: c5 d4 14 e1 vunpcklps %ymm1,%ymm5,%ymm4 f13: c5 d4 15 c9 vunpckhps %ymm1,%ymm5,%ymm1 f17: c5 fc 58 c7 vaddps %ymm7,%ymm0,%ymm0 f1b: c4 e3 5d 18 f9 01 vinsertf128 $0x1,%xmm1,%ymm4,%ymm7 f21: c4 e3 5d 06 c9 31 vperm2f128 $0x31,%ymm1,%ymm4,%ymm1 f27: c5 fc 59 c2 vmulps %ymm2,%ymm0,%ymm0
Sqrt3 SIMD-Befehle werden genutzt	* initial = (1 << 29) + (*ai >> 1) - (1 << 22) - 0x4C000; 1052:c5 f9 6f 44 05 00 vmovdqa 0x0(%rbp,%rax,1),%xmm0 1058:c5 f1 72 e0 01 vpsrad $0x1,%xmm0,%xmm1 105d:c5 f1 fe d4 vpaddd %xmm4,%xmm1,%xmm2 // Newton Verfahren durchfuehren for (unsigned int j = 0; j < LOOPS; j++) { * root = 0.5 * (* root + (* a / * root)); 1061:c5 f8 5e ca vdivps %xmm2,%xmm0,%xmm1 1065:c5 f0 58 ca vaddps %xmm2,%xmm1,%xmm1 1069:c5 f0 59 cb vmulps %xmm3,%xmm1,%xmm1 106d:c5 f8 5e c1 vdivps %xmm1,%xmm0,%xmm0 1071:c5 f8 58 c1 vaddps %xmm1,%xmm0,%xmm0 1075:c5 f8 59 c3 vmulps %xmm3,%xmm0,%xmm0 1079:c4 c1 78 29 44 05 00 vmovaps %xmm0,0x0(%r13,%rax,1)

Zeitmessung

Zeitmessung für vier Iterationen in E203 mit dem Befehl

```
g++ -Wall -pedantic -march=native -mfpmath=sse -mavx2 -O3 sqrt_opt.cc
```

```
./a.exe
```

Durchlauf	Sqrt1, ein Mal pro Schleife [ns]	Sqrt1, vier Mal pro Schleife [ns]	Sqrt2 [ns]	Sqrt3 [ns]
1.	359401	375045	359385	359402
2.	359389	375031	359420	359388
3.	359405	375031	375030	359388
4.	343777	359403	375049	359404
5.	359388	375047	359387	359421
6.	359402	406295	406267	406293
7.	359385	375028	390672	359402
8.	359384	359402	359402	359402
9.	359402	375028	359402	359402
10.	343775	359419	375011	359420
Durchschnitt	356270,8	373472,9	371902,5	364092,2

Interpretation

Alle Ausführungszeiten liegen dicht beieinander. Eine Verbesserung konnte durch die SIMD-Befehle nicht erreicht werden. Es zeigt sich, dass sqrt1 mit einer Berechnung pro Schleife am schnellsten ist, obwohl alle Berechnungen Sequenziell durchgeführt werden. Sqrt3 braucht die zweit wenigste Zeit zum Ausführen, da hier die Berechnungen mit SIMD-Befehlen in vierer-Blöcken parallelisiert wurden. Das diese Version nicht schneller ist als die erste Sqrt1 Version, könnte an dem zusätzlichen Aufwand der Typkonvertierung zu Vektoren liegen. Sqrt1 mit vier Berechnungen pro Schleife benötigt die meiste Zeit. Dies könnte an der zusätzlichen Schleife liegen, die jedes Mal durchlaufen werden muss.

Aufgabe 3: k-d-Baum

Quelltext

Assemblercode

Zeitmessung

Durchlauf	Zeit ohne Optimierung (in Sekunden)	Zeit nach Optimierung (in Sekunden)
1.		
2.		
3.		
4.		
5.		
6.		
7.		
8.		
9.		
10.		
Durchschnitt		

Interpretation