MASSIV PARALLELE SICHTBARKEITSBERECHNUNG
STAND 25. Mai 2022

FRANK LIEBAU

Inhaltsverzeichnis

1. Einleitung, Problemstellung

3D-CFD Simulationen überprüfen und optimieren in der frühen Phase der Fahrzeugentwicklung (Konzeptphase) die thermische Stabilität eines Fahrzeugkonzepts. Hierzu werden ein Modell des Gesamtfahrzeugs (inkl. Fahrzeugstrak, Motorraum, Tunnel, Unterboden und Hinterwagen) in einem virtuellen Windkanal platziert und die Temperatur- und Strömungsfelder für verschiedene Lastfälle ermittelt. Zur Bestimmung von realistischen Bauteiltemperaturen ist das Strömungsfeld mit dem Festkörper gekoppelt, dabei werden neben dem konvektiven Wärmetransport auch die Wärmestrahlung und die Wärmeleitung im Bauteil abgebildet. Simulationszeiten für ein Gesamtfahrzeug (Wärmeleitung, Strahlung und Strömung) können, auch unter Benutzung eines Rechenclusters, schon einmal mehrere Tage betragen. Die Algorithmenentwicklung orientiert sich deshalb an der Notwendigkeit, dem Anwender die Ergebnisse einer Simulationsrechnung

Date: 25. Mai 2022.

schnellstmöglich zur Verfügung zu stellen. Dabei darf die angestrebte Verkürzung der Rechenzeit selbstverständlich nicht auf Kosten der Genauigkeit gehen.

1.1. **Radiosity equation.** Das mathematische Modell für den Strahlungs- bzw- Energieaustausch zwischen diffusen grauen Oberflächen ist die *radiosity-equation*, siehe auch Siegel[9],

$$q_{\text{out}}(\mathbf{x}) = \epsilon\,\sigma\,T^4(\mathbf{x}) + \rho \int_\Gamma k(\mathbf{x},\mathbf{y})\,q_{\text{out}}(\mathbf{y})\,d\mathbf{y}, \quad \forall \mathbf{x} \in \Gamma. \tag{1}$$

Dabei sind

- Γ die Oberfläche einer Teilmenge Ω des Raumes, kurz: die Umhüllung,
- $T(\mathbf{x})$ die Temperatur im Oberflächenpunkt $\mathbf{x}$,
- ϵ der Emissionsgrad, $\rho = 1 - \epsilon$ der Reflexionsgrad und σ die Stefan-Boltzmann Konstante,
-

$$k(\mathbf{x},\mathbf{y}) := \frac{\cos\phi_{\mathbf{x}}\,\cos\phi_{\mathbf{y}}}{\pi\,\|\mathbf{x}-\mathbf{y}\|^2}\,\chi(\mathbf{x},\mathbf{y}), \qquad \mathbf{x},\mathbf{y} \in \Gamma, \tag{2}$$

der Kern des Integraloperators

$$(\mathcal{K}\,q)(\mathbf{x}) := \int_\Gamma k(\mathbf{x},\mathbf{y})\,q(\mathbf{y})\,d\mathbf{y}, \quad \mathbf{x} \in \Gamma, \tag{3}$$

- q_{out} die Helligkeit,
- $\chi(\mathbf{x},\mathbf{y})$ die visibility-Funktion, siehe (4)

Ausgehend von den Temperaturen auf den beteiligten Oberflächen wird die entsprechende Helligkeit, d. h. die Ausstrahlung oder *radiance*, durch (1) festgelegt. Eine analytische Lösung läßt sich i. Allge. nicht finden. Deshalb muss die Gleichung numerisch gelöst werden, siehe Hackbusch[2],[4].

1.2. **Ziel.** Nicht die Untersuchung der Gleichung (1) bzw. deren numerische Lösung steht im Fokus des Projektes, sondern ein notwendiger Teilschritt: die Bestimmung der Sichtbarkeiten, d h. die Auswertung der Funktion

$$\chi(\mathbf{x},\mathbf{y}) : \Gamma\times\Gamma \to \{0,1\}, \quad \chi(\mathbf{x},\mathbf{y}) = \begin{cases} 1 & \mu\mathbf{x} + (1-\mu)\mathbf{y} \notin \Gamma, \quad \mu \in (0,1) \\ 0 & \text{sonst} \end{cases}. \tag{4}$$

Die visibility-Funktion χ hat genau dann den Wert 1, wenn der Punkt $\mathbf{x}$ den Punkt $\mathbf{y}$ „sieht“, es also in gerader Linie kein Hindernis zwischen den beiden Punkten gibt. Die Definition der visibility-Funktion sieht zwar harmlos aus, aber ihre Berechnung entwickelt sich zu einem ernsthaften Problem bzgl. des (Rechen-) Zeitaufwands.

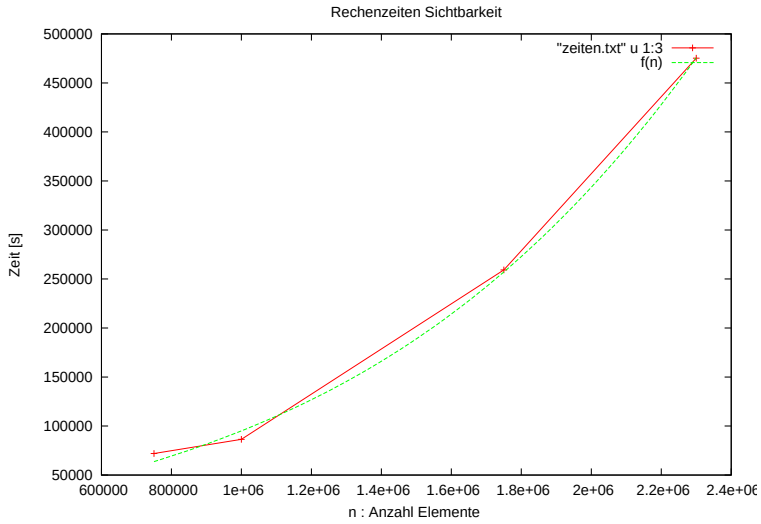

ABBILDUNG 1. Der Rechenzeitaufwand wächst quadratisch mit der Anzahl der Elemente

1.3. **Rechenaufwand.** Simulationen nutzen eine geometrische Diskretisierung des realen oder virtuellen Modells, die sogenannte Triangulierung. Die Oberfläche ist in den hier vorliegenden Fällen in mehrere Millionen Dreiecke zerlegt. Für jedes dieser Dreiecke ist zu entscheiden welche anderen Dreiecke von diesem Dreieck aus gesehen werden. Diese Menge wird als "Sichtbarkeiten" bezeichnet. Zu ihrer Bestimmung, also der Auswertung von (4), nutzt man eine Variante des "ray-triangle-intersection-Algorithm".

Ist n die Anzahl aller Dreiecke im Modell, dann sind also n Tests für ein Dreieck und insgesamt für das gesamte Modell n^2 Tests[1] notwendig. Der Rechenaufwand, das ist die notwendige Anzahl der elementaren Operationen, für den einzelnen Test "sieht Dreieck a Dreieck b?" liegt in der Größenordnung $\mathcal{O}(\log_2 n)$. Die Sichtbarkeiten für ein Dreieck lassen sich damit mit einem Aufwand proportional zu $n \log_2 n$ berechnen. Für das gesamte Problem hat man die Komplexität $\mathcal{O}(n^2 \log_2 n)$.

In Abbildung 1 sind die Rechenzeiten zur Sichtbarkeitsberechnung für 4 Testprobleme[2] aufgetragen. Die grüne Kurve ergibt sich dabei aus einem curve fitting mit dem Ansatz $f(n) = b * \left(0.001 \frac{n}{2}\right)^a$ und den Exponenten $a = 1.98$, was schon ziemlich nah am theoretischen Wert ist. Die Datenbasis ist hier mit 4 Werten allerdings auch sehr klein.

Unter der hypothetischen Annahme, dass pro Sekunde 100000 Tests möglich sind, zeigt die nachfolgende Abschätzung in welche Dimension die Rechenzeit steigt. Für ein Fahrzeugmodell mit 1.6 Mio Dreiecken ergibt sich schon eine Rechenzeit von ca. 4 Jahren[3]. Niemand hat Zeit, solange auf ein Ergebnis zu

[1] Symmetrie und die Tatsache, dass ein Dreieck sich nicht selbst sehen kann, wird an dieser Stelle der Übersichtlichkeit halber unterschlagen

[2] Der Test zum Problem mit 2.5 Mio. Dreiecken rechnete mehr als 5 Tage, wobei ein Zusammenschluß von mehreren Workstations genutzt wurde.

[3] jetzt mit Berücksichtigung Symmetrie

warten. Die bei der GWR eingesetzte Workstation schafft etwa 50 Mio Tests pro Sekunde, was dann einer geschätzten Rechenzeit von 3 Tagen entspricht. Ein Cluster ist wiederum 10 mal schneller, also mit 500 Mio Tests pro Sekunde. Aber hier ist der Ressourceneinsatz auch wesentlich höher: 8 Rechner statt einer. Die Berechnung braucht immer noch 7 Stunden und, was nicht zu vernachlässigen ist, der Anwender konkurriert mit den anderen Usern des Rechenclusters. D. h. der Simulationsjob steht erst einmal in einer Warteschlange bis dann entsprechende Ressourcen auf dem Cluster frei sind: die Zeit bis ein Ergebnis vorliegt verlängert sich weiter.

Die Problemstellung hat aber auch eine positive Eigenschaft: die Aufgabe ist trivial parallelisierbar. Das ist leicht einzusehen, denn die Ergebnisse der Sichtbarkeitsberechnung für ein Dreieck sind völlig unabhängig von anderen Rechenergebnissen (wieder ohne Rücksicht auf die Symmetrie). Nur die geometrische Lage der Dreiecke zueinander ist entscheidend.

In Abbildung 1 sind die Rechenzeiten zur Sichtbarkeitsberechnung für 4 Testprobleme aufgetragen. Die grüne Kurve ergibt sich dabei aus einem curve fitting mit dem Ansatz

$$f(n) = b * \left(0.001\,\frac{n}{2}\right)^a, \qquad b = 0.382,\ a = 1.98. \tag{5}$$

D.h. die Rechenzeit wächst quadratisch mit der Anzahl der Elemente.

2. Cluster

Die Oberfläche Γ sei in n disjunkte Dreiecke (Paneele, panels, faces, Facetten) zerlegt

$$\Gamma = \bigcup_{i \in \mathcal{I}} t_i, \qquad \mathcal{I} \text{ eine Indexmenge}, \quad \mathcal{P} := \bigcup_{i \in \mathcal{I}} t_i$$

Dabei sind $\mathcal{I}$ eine Indexmenge und $\mathcal{P}$ die Menge der Paneele.

Definition 1 (Cluster)**.** *Jede Teilmenge $\mathcal{C} \subset P$ ist ein Cluster. Ist $\mathcal{I}$ die Indexmenge zu $\mathcal{P}$, dann definiert jede Teilmenge von $\mathcal{I}$ in kanonischer Weise einen Cluster.*

Durch das Zusammenfassen von benachbarten Dreiecken können *Cluster* τ erklärt werden. Benachbarte Cluster können wiederum zu größeren Clustern vereinigt werden. Dieses Vorgehen kann solange durchgeführt werden, bis die gesamte Oberfläche in einem einzigen Cluster zusammengefaßt ist. Das Verfahren definiert in natürlicher Weise den *cluster tree* $\mathcal{T}$: die Blätter sind die ursprünglichen Paneele, die Oberfläche Γ ist die Wurzel von $\mathcal{T}$ und die restlichen (inneren) Knoten sind die erzeugten Cluster. Eine formale Definition für $\mathcal{T}$ ist z. B. in Hackbusch[3] oder auch in Graham[1] gegeben.

Definition 2 (Clusterbaum, Hackbusch[3])**.** *Die Menge aller Teilmengen von $\mathcal{P}$ sei mit $\mathcal{S}$ bezeichnet.*

(1) *Alle Knoten, d. h. Cluster, von $\mathcal{T}$ repräsentieren Teilmengen von $\mathcal{P}$.*

(2) *Γ ist die Wurzel von $\mathcal{T}$*

(3) *Jedes Paneel $t \in \mathcal{P}$ entspricht einem Blatt von $\mathcal{T}$.*

(4) *Ist $\tau \in \mathcal{T}$ kein Blatt. Dann sei $sons(\tau)$ die Menge der Söhne; für sie gilt $|sons(\tau)| \geq 2$. Weiter gilt*

$$\tau = \bigcup_{\tau' \in sons(\tau)} \tau'.$$

Der Cluster τ repräsentiert die Vereinigungsmenge seiner Söhne.

Geeignete Teilmengen der Knoten des Clusterbaums liefern eine disjunkte Zerlegung der ursprünglichen Triangulierung (siehe Abbildung 2).

2.1. **Speicherplatzreduktion: komprimierte Sichtbarkeit.** Eine einfache Möglichkeit eine einmal ermittelte Sichtbarkeit bzgl. eines Paneels p zu speichern, ist es die Indizes aller von q sichtbaren Paneele in einer Liste zu speichern. Etwa

```
...
12   3 4 7 8 13
13   1 5 6 11 12
...
```

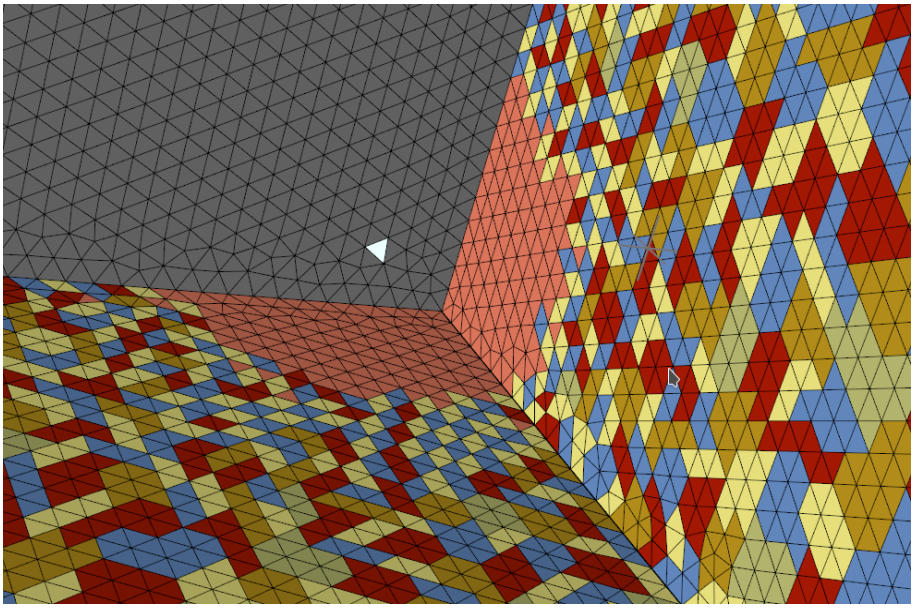

ABBILDUNG 2. Visualisierung Cluster für eine Testgeometrie. Weiß: soure Paneel, grau: nicht sichtbare Paneele, orange: Nahfeld, sonst: Cluster des Fernfeldes.

(Paneel 12 sieht die Paneele 3,4,7,8 und 13). Bei diesem Vorgehen muß mit einem Speicheraufwand von $\mathcal{O}(n^2)$ gerechnet werden[4]. Geht man von einem 4 Byte integer-Typ aus und der Annahme, dass durchschnittlich 80% aller Dreiecke sichtbar sind, so benötigt ein Problem mit $n = 353158$ Paneelen (Abb. 2) etwa 460 GB Arbeitsspeicher. Für ein Modell mit 1 Million Paneelen sind es dann schon ca. 3 Terabyte und bei 5 Million Paneelen ca. 72 TB. Das ist aussichtslos!

Ein Teilaspekt der panel clustering Idee (Hackbusch[4]), die zur numerischen Lösung der Integralgleichung (1) dient, kann auf den Bereich „Sichtbarkeit“ übertragen werden: analog zum Begriff des *zulässigen Clusters* bzgl. eines Paneels kann auch ein zulässiger Sichtbarkeitscluster festgelegt werden und damit kann auch eine minimale zulässige Sichtbarkeit-Überdeckung C_{vis} für ein Paneel angegeben werden. Der Aufwand zur Ermittlung dieser neuen Überdeckung ist zwar immer noch in der Größenordnung $\mathcal{O}(n^2)$, aber der Speicheraufwand ist nun wesentlich kleiner als n^2. Jetzt kommt eine weitere Idee ins Spiel: betrachten wir zwei Paneele p und q, die einen gemeinsamen Vorfahren (Vater) V haben (p und q sind Brüder). In sehr vielen Fällen werden sich die Sichtbarkeits-Überdeckungen $C_{\text{vis}}(p)$ von p und $C_{\text{vis}}(q)$ von q nicht viel unterscheiden. Statt nun die in *beiden* Mengen vorkommenden Cluster doppelt zu speichern, ist es ausreichend die betreffenden Cluster in dem gemeinsamen Vorfahren V abzulegen.

Für $p, q \in \mathcal{P}$ mit $V = \texttt{father}(p) = \texttt{father}(q)$, d. h. die beiden Paneele sind Brüder, seien die Sichtbarkeiten $S(p)$ und $S(q)$ ermittelt.

[4]Das Ausnutzen der Symmetrie an dieser Stelle führt dann auf Suchprobleme.

(a): eine minimale Überdeckung $C_{\text{vis}}(p,q)$ der Schnittmenge

$$S(p) \cap S(q) \qquad (= S(V))$$

wird im Knoten V gespeichert,

(b): p erhält „nur“ diejenigen Paneele, die nicht schon in $C_{\text{vis}}(p,q)$ liegen:

$$\{q \,:\, q \in S(p) \,\wedge q \notin C_{\text{vis}}(p,q)\} = S(p) \setminus S(V)$$

(Mengendifferenz). Mit der de Morganschen Regel $X \setminus (A \cap B) = X \setminus A \cup X \setminus B$ wird in p gerade eine minimale Überdeckung der Menge

$$S(p) \setminus S(q)$$

gespeichert.

(c): genauso erhält q: $S(q) \setminus S(p)$

Demnach erhält der Vater V die Schnittmenge $S(V)$ und in den beiden Söhnen ist die symmetrische Differenz $S(q) \bigtriangleup S(p)$ gespeichert. Die minimale zulässige Überdeckung für das Blatt p ergibt sich dann aus den direkt zugeordneten Clustern und der im Vater gespeicherten Clustermenge:

$$S(p) = S(V) \cup (S(p) \setminus S(q))\,.$$

Die Verfahrensschritte (a),(b) und (c) können auf alle Cluster $\tau \in \mathcal{T}$ ausgedehnt werden.

3. Parallelisierungen

AMD und Intel brachten 2005 bzw. 2006 die ersten Mehrkernprozessoren (dual core) auf den (PC-) Markt. Danach kam es zu einer rasanten Entwicklung von Vier-, Sechs- und weiteren Mehrkernprozessoren. Heute sind CPUs mit bis zu 64 Kernen erhältlich. Parallel zur Steigerung der Anzahl der Prozessorkerne wurden auch die Vektorbefehle, die es ermöglichen gleichartige Operationen parallel auszuführen, weiterentwickelt (AVX & SIMD, siehe etwa Wikipedia[10]). So nahm die Registerbreite, und damit auch die Leistungsfähigkeit, der AVX-Befehle von 64 über 128 und 256 schließlich auf 512 Bit zu.

Bis Mitte der 2000er Jahre steuerten Grafikkarten ausschließlich den Monitor an. Mit dem ersten CUDA-Release Anfang 2007 folgte auch hier eine erstaunliche Entwicklung: der Prozessor der Grafikkarte (Graphics Processing Unit oder kurz GPU) kann nun spezielle Programme ausführen, so dass dieser als "Recheneinheit" neben der CPU zum Einsatz kommt. Die Entwicklung ist hier soweit fortgeschritten, dass Highend-Grafikkarten heute, für bestimmte Anwendungen, auf dem Niveau früherer Supercomputer rechnen. So veröffentlicht Nvidia eine Leistung von 30 TeraFLOPS bei doppelter Genauigkeit für das aktuelle GPU-Spitzenmodell Hopper H100.

3.1. **Visibility, OptiX.**

Listing 1 enthält die Standardmethode zur Sichtbarkeitsbestimmung im Clusterbaum. Dabei liefern die C++-Methoden `MinimumAdmissibleVisCovering(node)` und `BerechneCompressedVisibility` die Sichtbarkeits-Überdeckung für ein Paneel entsprechend Abschnitt 2.1. Für innere Knoten des Clusterbaumes ist an dieser Stelle schon eine Parallelisierung über OpenMp-Taskgroups[5] implementiert.

```
1  void VisClustering::CalcCompressedVis(pVisClusterNode &node) {
2          if (node->IsLeaf() {
3                  node->Cvis = MinimumAdmissibleVisCovering(node);
4              } else {
5                      #pragma omp taskgroup
6                      {
7                          #pragma omp task shared(node) untied
8                          {
9                              CalcCompressedVis(node->left);
10                         }
11                         #pragma omp task shared(node) untied
12                         {
13                             CalcCompressedVis(node->right);
14                         }
15                     }
16                 node->BerechneCompressedVisibility();
17             }
18         }
19     }
```

Listing 1. C++-Methode CalcCompressedVis

[5] sihe `www.openmp.org/spec-html/5.0/openmpsu94.html`

Jeder Aufruf von `MinimumAdmissibleVisCovering` führt für jedes andere Dreieck der Triangulierung den Sichtbarkeitstest. Da die Routine auch für jedes Paneel aufgerufen wird, werden hier insgesamt $\mathcal{O}(n^2)$ Tests durchgeführt.

Eine erste kleine Verbesserung gegenüber Listing 1 ergibt sich, wenn die OpenMp-Taskgroups durch die `task_group` aus der Intel-Bibliothek Threading Building Blocks (TBB) ersetzt wird. Hierbei werden die Kerne der CPU wesentlich besser ausgenutzt als unter OpenMp in Listing 1.

```
1  void VisClustering::CalcCompressedVisTBB(pVisClusterNode &node) {
2          if (node->IsLeaf() {
3
4                  node->Cvis = MinimumAdmissibleVisCovering(node);
5
6              } else {
7
8                  tbb::task_group g;
9                  g.run([&]() { if (node->left  != nullptr) CalcCompressedVisTBB(node->left); });
10                 g.run([&]() { if (node->right != nullptr) CalcCompressedVisTBB(node->right); });
11                 g.wait();
12
13                 node->BerechneCompressedVisibility();
14             }
15         }
16     }
```

Listing 2. C++-Methode CalcCompressedVis

Der entscheidende Schritt ist aber die Ersetzung von `MinimumAdmissibleVisCovering` durch eine massiv parallele Variante auf der Grafikkarte. Der `CudaWrapper` aus Listing 3 enthält die notwendigen C++-Methoden. Die Implementierung nutzt hierbei insbesondere die NVIDIA-OptiX-Bibliothek[8]:

> NVIDIA OptiX™ Ray Tracing Engine. An application framework for achieving optimal ray tracing performance on the GPU. It provides a simple, recursive, and flexible pipeline for accelerating ray tracing algorithms. Bring the power of NVIDIA GPUs to your ray tracing applications with programmable intersection, ray generation, and shading.

Die OptiX-Bibliothek ist auf schnelle ray tracing Algorithmen spezialisiert, das ist genau die Funktionalität die für den Sichtbarkeitstest gebraucht werden. Damit OptiX in einem C++-Programm genutzt werden kann, sind die nachfolgenden Schritte notwendig.

SetUp: In dieser Initialisierungsphase ist für die Berechnung auf der Grafikkarte der benötigte Speicher bereitzustellen und die Triangulierung in das OptiX eigene Format zu übertragen (Listing 4, 5). Die Setzungen sind nur einmal auszuführen. Speicherallokierungen im Listing 5 mit dem Flag `RTP_BUFFER_TYPE_CUDA_LINEAR` betreffen die Grafikkarte, während `RTP_BUFFER_TYPE_HOST` Speicherplatz auf der Workstation (host) erzeugt.

```
class CudaWrapper {
    GeoObjekt *geo;
    public:
    static const bool SetHits_OnDevice;
    static const bool GetHits_OnDevice;
    static const bool dohitworkspace;
    double timeGetHits, timedocuda;
    optix::prime::Context context;
    optix::prime::Model model;
    CudaWrapper();
    ~CudaWrapper();
    CudaWrapper(const CudaWrapper &);
    void SetGeo(GeoObjekt *);
    Buffer<Ray  > raysBuffer;
    Buffer<Ray  > raysBufferOnDevice;
    Buffer<Hit_t> hits;
    Buffer<Hit_t> hitsOnDevice;
    Buffer<int>   hitWorkSpaceOnDevice;
    Buffer<int>   hitWorkSpaceOnHost;
    Buffer<float3> triangleMidpoints;
    Buffer<float3> triangleMidpointsOnDevice;
    Buffer<float3> triangleNormals;
    Buffer<float3> triangleNormalsOnDevice;
    optix::prime::Query  query;
    void CudaSetup();
    void CreateBuffer();
    void SetUpRayAndHits();
    void SetCudaRays(const int &);
    void SetCudaRaysOnDevice(const int &);
    float3 GetEye(const int &) const;
    float3 GetNormal(const int& id) const;
    panel GetPanel(const int &) const;
    Math::IntSet2 GetHits() const;
    Math::IntSet2 GetHitsFromHostWS() const;
    void GetHitsFromDevice(pVisClusterNode);
    vector<int> GetHitsBitMaskFromDevice(pVisClusterNode);
    bool IsHit(int) const;
    void printMeshInfo(const Mesh& mesh, std::ostream& out=std::cout);
};
```

Listing 3. class CudaWrapper

```
    void CudaWrapper::CudaSetup() {
        unsigned int device = 0;
        context->setCudaDeviceNumbers(1, &device);
        model = context->createModel();
        PrimeMesh mesh = geo->T.GetOptixPrimeMesh();
        model->setTriangles(mesh.num_triangles, RTP_BUFFER_TYPE_HOST, mesh.getVertexIndices(),
                            mesh.num_vertices, RTP_BUFFER_TYPE_HOST, mesh.getVertexData());
        model->update(0);
        // RTP_QUERY_TYPE_ANY: Return any hit along a ray
        query = model->createQuery(RTP_QUERY_TYPE_ANY); // RTP_QUERY_TYPE_CLOSEST);
    }
```

Listing 4. Optix-Setup

```
1   void CudaWrapper::CreateBuffer() {
2
3     const uint N = geo->T.V.size();
4
5                 raysBuffer.alloc( N, RTP_BUFFER_TYPE_HOST        , UNLOCKED );
6         raysBufferOnDevice.alloc( N, RTP_BUFFER_TYPE_CUDA_LINEAR, UNLOCKED);
7
8     hitsOnDevice.alloc ( N, RTP_BUFFER_TYPE_CUDA_LINEAR     , UNLOCKED );
9     hits.alloc         ( N, RTP_BUFFER_TYPE_HOST            , UNLOCKED );
10
11    hitWorkSpaceOnDevice.alloc( N, RTP_BUFFER_TYPE_CUDA_LINEAR     , UNLOCKED );
12    hitWorkSpaceOnHost.alloc  ( N, RTP_BUFFER_TYPE_HOST , UNLOCKED);
13
14    query->setRays(raysBufferOnDevice.count(), Ray::format   , raysBufferOnDevice.type(),
          raysBufferOnDevice.ptr());
15
16    if (SetHits_OnDevice)
17     query->setHits(hitsOnDevice.count()   , Hit_t::format, hitsOnDevice.type() , hitsOnDevice.ptr());
18    else
19     query->setHits(hits.count()       , Hit_t::format, hits.type()       , hits.ptr());
20
21
22    triangleMidpoints.alloc        (N, RTP_BUFFER_TYPE_HOST, UNLOCKED );
23    triangleMidpointsOnDevice.alloc(N, RTP_BUFFER_TYPE_CUDA_LINEAR, UNLOCKED);
24    triangleNormals.alloc          (N, RTP_BUFFER_TYPE_HOST, UNLOCKED );
25    triangleNormalsOnDevice.alloc  (N, RTP_BUFFER_TYPE_CUDA_LINEAR, UNLOCKED);
26
27    float3* tm      = triangleMidpoints.ptr();
28    float3* normals = triangleNormals.ptr();
29
30    const float scene_epsilon  = 0.0001f;
31
32    for (uint i = 0; i < N; ++i) {
33            tm[i]      = geo->T.M[i].GetFloat3();
34            normals[i] = geo->T.N[i].GetFloat3();
35    }
36
37    const auto count = triangleMidpoints.sizeInBytes();
38
39    const auto errorM = cudaMemcpy(triangleMidpointsOnDevice.ptr(), triangleMidpoints.ptr(), count,
          cudaMemcpyHostToDevice);
40
41    if (errorM != cudaSuccess) {
42        throw thrust::system_error(errorM, thrust::cuda_category(), FileLinePrint);
43    }
44
45        auto errorN = cudaMemcpy(triangleNormalsOnDevice.ptr(), triangleNormals.ptr(), triangleNormals.
              sizeInBytes() , cudaMemcpyHostToDevice);
46
47    if (errorN != cudaSuccess) {
48        throw thrust::system_error(errorN, thrust::cuda_category(), FileLinePrint);
49    }
50
51 }
```

Listing 5. CreateBuffer

SetRays: Sind $p, q \in \mathcal{P}$ die Indizes zu zwei Dreiecken der Triangulierung mit n Dreiecken und M_p, M_q die entsprechenden Mittelpunkte, dann definiert

$$r : \mathbb{R} \to \mathbb{R}^3 \ : r(s) := M_p + t\,(M_q - M_p), \quad t \geq 0 \tag{6}$$

einen ray (eine Halbgerade). Für jedes $p \in \mathcal{P}$ sind n rays zu berechnen.

Die Setzung kann sowohl auf dem host (Listing 6, TBB-Parallelisierung) als auch über ein Cuda-kernel-Function (Listing 7, Cuda-Parallelisierung) auf der Grafikkarte erfolgen. Die Cuda-Kernel-Funktion ist aber gesondert mit dem NVIDIA-Compiler `nvcc` zu compilieren. Praktischerweise funktioniert dies mit dem `cmake`-script aus Listing 10.

Query: Hier erfolgt die eigentliche Berechnung der Funktion (4), d.h. der Sichtbarkeitstest. Der Aufruf ist, nach der Vorarbeit im Setup- und SetRays-Schritt überraschend einfach:

```
1   query->execute(0);
2   query->finish();
```

Listing 8. OptiX-Aufruf zur Sichtbarkeitsberechnung

GetHits: Die OptiX-Ergebnisse werden in den hits-Buffer (Listings 3 und 5) geschrieben und über die `GetHits`-Methode (Listing 3.1) ausgelesen. Auch diese Methode ist einfach gehalten, es ist nur zu berücksichtigen, dass gerade die "Nichttreffer", d. h. kein Hindernis zwischen den Paneelen, gefragt sind. OptiX kennzeichnet diese durch die Setzung eines negativen Wertes für den Parameter t.

Das abschließende Resultat ist die über OptiX und TBB optimierte Version der Sichtbarkeitsberechnung aus Listing 9.

```
1  void VisClustering::CalcCompressedVisCudaTBB(pVisClusterNode &node) {
2          if (node->IsLeaf()) {
3              SetCudaRaysOnDevice(node->faceindex);
4              query->execute(0);
5              query->finish();
6              node->Cvis = std::move(GetHits());
7              }
8              else {
9                  tbb::task_group g;
10                 g.run([&]() { if (node->left  != nullptr) CalcCompressedVisCudaTBB(node->left); });
11                 g.run([&]() { if (node->right != nullptr) CalcCompressedVisCudaTBB(node->right); });
12                 g.wait();
13                 node->BerechneCompressedVisibility();
14             }
15         }
```

Listing 9. Optimierte Sichtbarkeitsberechnung

Anmerkung: Für die Umsetzung wurde OptiX in der Version 6.5 gewählt. Wie sich im Laufe des Projektes herausstellte, ist diese inkompatibel zur aktuellen Version 7. Und weiterhin ist es nicht möglich Programme, die mit OptiX in der Version 6.x gelinkt sind, auf den aktuellen Grafikkarten der Architekturen ab Ampere zu nutzen. Es stellt nun die Frage, ob eine Portierung auf OptiX 7 sinnvoll ist oder ob evtl. der Einsatz

```
void CudaWrapper::SetCudaRays(const int &id) {

    Ray* rays = raysBuffer.ptr();

    // scene_epsilon
    const float scene_epsilon  = 0.0001f;
    const point3d    O = geo->T.M[id] + (geo->T.N[id] * scene_epsilon*1.1f);
    const float3 start = make_float3(O.x,O.y,O.z);

    float3* tm = triangleMidpoints.ptr();

    const int nRange=raysBuffer.count();

    tbb::parallel_for(tbb::blocked_range<int>(0, nRange),
        [&](tbb::blocked_range<int> r) {
            for (size_t i = r.begin(); i < r.end(); ++i) {
            const float targetX = tm[i].x;
            const float targetY = tm[i].y;
            const float targetZ = tm[i].z;
            const float3 direction = make_float3(targetX-start.x, targetY-start.y, targetZ-start.z);
            rays[i].origin = start;
            rays[i].tmin   = scene_epsilon;
            rays[i].dir    = direction;
            rays[i].tmax   = 1.0f - scene_epsilon;
            }
        }
    );
    memset(hits.ptr(), 1.0f, hits.count()*sizeof(Hit_t));
}
```

Listing 6. Setzung der rays, Standardmethode mittels TBB

```
__global__ void createRaysKernel(float4* rays, float3* tm, int count, float3 eye, int id) {

    const int stride = blockIdx.x * blockDim.x;
    const int idx = threadIdx.x + stride;

    const float scene_epsilon = 0.0001f;
    const float tmax = 1.0f - scene_epsilon;
    // ohne Loop
    //
    if (idx >= count)
        return;

    const float3 target = make_float3(tm[idx].x, tm[idx].y, tm[idx].z);
    const float3 direction = make_float3(target.x - eye.x, target.y - eye.y, target.z - eye.z);

    rays[2 * idx + 0] = make_float4(eye.x, eye.y, eye.z, 0.0f); // origin, tmin
    rays[2 * idx + 1] = make_float4(direction.x, direction.y, direction.z, tmax); // dir    , tmax
}
```

Listing 7. Cuda-kernel zur Setzung der rays

einer anderen Software erwogen werden sollte. Grundsätzlich wurde mit der verwendeten OptiX-Version ja das ursprüngliche Ziel erreicht. Da eine weitere Variante mit der aktuelleren OptiX-Version nicht unbedingt weitere Erkenntnisse liefern würde (abgesehen von einer weiteren Beschleunigung), und auch weil der Einsatz der Intel Threading Building Blocks sich bewährt hat, fiel die Entscheidung auch Intel Embree, siehe Abschnitt 3.3, zur Sichtbarkeitsberechnung einzusetzen.

```
1  IntSet2 CudaWrapper::GetHits() const {
2
3      const Hit_t* hit  = hits.ptr();
4      const int n       = hits.count();
5
6      int hitcounter=0;
7
8      for (uint i=0; i < n; ++i)
9              if ( (hit[i].t < 0.0) ) ++hitcounter;
10
11     IntSet2 result; result.reserve( hitcounter );
12
13         for (uint i=0; i < n; ++i) {
14             if (hit[i].t < 0.0 ) {
15                 result.push_back(i);
16             }
17         }
18         // keine doppelten Eintr ge in result, result ist sortiert(!)
19         return result;
20     }
```

```
1
2  ################################### CUDA ###################################
3
4  # Pass options to NVCC
5  find_package(CUDA 10.0 REQUIRED)
6
7  # GTX 960       : Maxwell --> "compute_52,code=sm_52"
8  # Quadro K4200  : Kepler  --> SM30 or SM_30, compute_30
9  # Tesla P100    : Pascal  --> -gencode=arch=compute_60,code=sm_60
10 # Tesla V100    : Volta   --> SM70 or SM_70, compute_70
11 # RTX 2060      : Turing  --> SM75 or SM_75, compute_75
12 # set(CUDA_NVCC_FLAGS ${CUDA_NVCC_FLAGS} -O3 -arch=sm_75)
13 #
14 # RTX 3080      : Ampere  --> SM_86, compute_86      (from CUDA 11.1 onwards)
15 set(CUDA_NVCC_FLAGS ${CUDA_NVCC_FLAGS} -O3 -arch=sm_86)
16 set(CUDA_NVCC_FLAGS ${CUDA_NVCC_FLAGS}  -gencode=arch=compute_75,code=sm_75)
17 set(CUDA_NVCC_FLAGS ${CUDA_NVCC_FLAGS}  -gencode=arch=compute_86,code=sm_86)
18
19 include_directories(${CUDA_TOOLKIT_ROOT_DIR}/targets/x86_64-linux/include)
20
21 set(OPTIX "/home/liebau/Cuda/NVIDIA-OptiX-SDK-6.0.0-linux64")
22
23 include_directories(${OPTIX}/include)
24 include_directories(${OPTIX}/SDK)
25 include_directories(${OPTIX}/SDK/sutil)
26
27 link_directories(${OPTIX}/lib64 ${CUDA_LIBRARIES})
28
29 # COMPILE CU FILES
30 file(GLOB CUDA_FILES "src/*.cu")
31
32 set(CU_O "primeKernels")
33 CUDA_ADD_LIBRARY(${CU_O} SHARED ${CUDA_FILES})
34
35 # Problem: kopiere cuda-Lib nach LIB ... so funktioniert es nicht
36 # aber "make install" kopiert ${CU_O} nach ${LIB}; siehe auch ${LIB}
37 install(TARGETS ${CU_O} DESTINATION ${LIB})
38
39 # CU_O ist die lib aus den cu-Files
40 MESSAGE( status " cuda_root : " ${CUDA_TOOLKIT_ROOT_DIR})
41 MESSAGE( status " OptiX      : " ${OPTIX})
42 MESSAGE( status " CU_O       : " ${CU_O})
43
44 ################################### CUDA ENDE ##############################
```

Listing 10. CMakeList.txt für cuda-kernels

3.2. **Threadsafe-Mengenoperationen, Intel Threading Building Blocks (TBB).** Die C++ Standard-Klassen std::set und std::map bzw. ihre unsortierten Varianten std::unordered_set und std::unordered_map (assoziative Container) sind nicht thread-safe. Eine thread-safe Alternative sind die entsprechenden Container aus der frei verfügbaren Intel-Bibliothek Threading Building Blocks[6].

Dazu ein Beispiel: für eine FEM-Berechnung sei ein 3D-Modells mittels Tetraeder trianguliert. Üblicherweise wird das Gitter durch eine Liste mit den 3D-Punkten

```
0  -0.3420   0.3420  0.0020
1  -0.3420  -0.3420  0.0020
2  -0.3420   0.3420  0.6339
3  -0.3420  -0.3420  0.6339
...
```

und einer zweiten Liste für die Tetraeder mit Punktindizes

```
0 524    51485  523    104970
1 104887 52572  120512 120102
2 89182  9528   104369 104369
3 168514 166744 84550  168926
...
```

gegeben. In diesem Beispiel definiert das Dreieck zu den Punkten 524, 51485 und 523 zusammen mit dem vierten Punkt zum Index 104970 Tetraeder 0. Dementsprechend enthält die C++-Klasse der Tetraeder auch diese vier Indizes:

```
1 class Tetra {
2     public:
3           array<int,4> v;      // Indizes der Knoten
4    ...
5 };
6 vector<Tetra>  Cells;
```

Zur Aufstellung des Gleichungssystems ist es zweckmäßig, auch die "umgekehrte" Information zu kennen: Zu welchen Tetraedern gehört ein bestimmter Punkt? Dazu sei /footnotesize

```
vector<IntSet> VertexStar(Points.size());
```

/normalsize D.h. zu jedem Punkt gibt es eine Menge, die die Indizes der umgebenen Tetraeder enthält. Ein einfacher for-loop über alle Cells liefert das Gewünschte.

```
1 typedef  std::set<int> IntSet;
2 vector<IntSet> VertexStar(Points.size());
3 const uint n = Cells.size();
4 for(uint i=0; i < n; ++i) {
5  const auto u = Cells[i].v;      // Indizes der Punkte zu Cell i
6  VertexStar[ u[0] ].insert(i); VertexStar[ u[1] ].insert(i);  // cell-Index i --> VertexStar[ u[k] ]
7  VertexStar[ u[2] ].insert(i); VertexStar[ u[3] ].insert(i);
8 };
```

LISTING 11. Berechnung VertexStar, seriell

Wendet man das Verfahren aus Listing 11 auf ein real-world Beispiel mit mit 5.7 Mio Elementen (Tetraedern) an, so beträgt die Rechenzeit $9.88\,s$.

Eine kleine Verbesserung ergibt sich, wenn auf eine Sortierung der Menge verzichtet wird. Statt std::set<int> kann std::unordered_set<int> verwendet werden, denn nur die Indizes der beteiligten Tetraeder sind interessant. Eine Reihenfolge ist hier nicht wichtig. Die Rechenzeit reduziert sich auf $8.92s$. Der Versuch die for-Schleife mittels OpenMp zu parallelisieren

```
1 #pragma omp parallel for
2 for(uint i=0; i < n; ++i) {
3  const auto u = Cells[i].v;      // Indizes der Punkte zu Cell i
4  VertexStar[ u[0] ].insert(i);  VertexStar[ u[1] ].insert(i);  // i --> VertexStar[ u[0] ]
5  VertexStar[ u[2] ].insert(i);  VertexStar[ u[3] ].insert(i);
6 };
```

Listing 12. Berechnung VertexStar, openmp

endet mit einem Absturz

```
|       Tetra4Mesh::SetSparseMatrixPattern()
Speicherzugriffsfehler (Speicherabzug geschrieben)
```

Das Einfügen einer Zahl in die Menge std::set<int> ist nicht thread-safe. Die Barriere #pragma omp critical(Insert) vor jedem insert-Aufruf "rettet" zwar den Algorithmus aus 12, bringt aber keinen Fortschritt bzgl. der Rechenzeit, da dann faktisch wieder das serielle Verfahren vorliegt.

Jetzt wird std::set<int> durch tbb::concurrent_unordered_set<int> ersetzt und die for-Schleife mittels tbb::parallel_for parallelisiert.

```
1 #include "tbb.h"
2  typedef tbb::concurrent_unordered_set<int> IntSet;
3  vector<IntSet>  VertexStar(Points.size());
4  tbb::parallel_for(static_cast<uint> (0), n, [&](uint i) {
5     const auto u = Cells[i].v;
6     VertexStar[ u[0] ].insert(i);     VertexStar[ u[1] ].insert(i);
7     VertexStar[ u[2] ].insert(i);     VertexStar[ u[3] ].insert(i);
8  });
```

Listing 13. Berechnung VertexStar, parallel, TBB

Der insert-Aufruf ist nun thread-safe und die Variante nutzt alle 16 Kerne der CPU[6] Die Rechenzeit beträgt nur noch erstauliche 0.6562 s. Das ist ein Speedup von 15 (!) und eine fast perfekte Skalierung.

[6]Alle Rechnungen in diesem Abschnitt wurden auf einem AMD Ryzen 9 5950X (16 cores) durchgeführt.

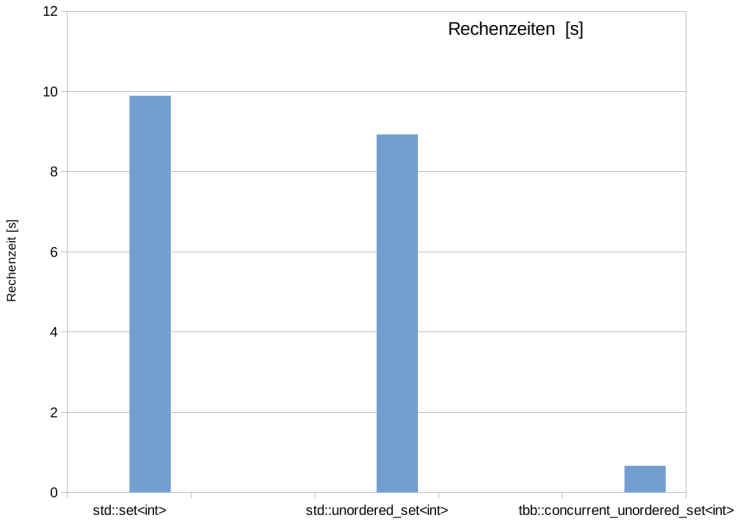

Abbildung 3. Vergleich der Rechenzeiten

3.3. **Intel Embree.** Intel Embree[5] ist eine weitere hoch spezialisierte ray tracing Software-Bibliothek. Aber im Unterschied zu Cuda/OptiX werden die Algorithmen nicht auf der Grafikkarte ausgeführt, sondern laufen auf der CPU. Das hat auch Vorteile, denn die Integration in ein bestehendes C++-Programm ist hier einfacher als im Cuda-Fall und es besteht die berechtigte Hoffnung, dass diese *immer* auf Intel-CPUs laufen werden. Desweiteren ist auch keine zusätzliche Hardware notwendig. Embree nutzt die Threading Building Blocks und je nach CPU hochoptimierte Vektorisierungen über SIMD-Befehle (single instruction, multiple data).

Bei der Entwicklung der C++-Programme wurde das Vorgehen aus Abschnitt 3.1 zur Schablone: es gibt wieder eine Wrapper, Listing 16, und vier Schritte zur Durchführung.

SetUp: die Initialisierungsphase zur Setzung der Datenstrukturen und des Speicherplatzes. Insbesondere wird die Geometrie der Biblothek bekannt gemacht, Listing 17

SetRays: Setzung der rays unter Berücksichtigung des "backface culling", Listing 18. es werden rays zu Dreiecken ausgeschlossen, die nur von der Rückseite sichtbar sind oder unterhalb der durchs Dreieck induzierten Ebene liegen. Der backface-culling-test ist relativ einfach, die Zeilen 24 und 25 in Listing 18, kann aber mit den Ideen aus Johannsen[7] weiter optimiert werden.

Query: Die Sichtbarkeitsberechnung: diesmal als Aufruf von

```
1  void EmbreeWrapper::InterSection(const uint &numRays) {
2            rtcIntersect1M(scene, &context, &rayhit[0], numRays, sizeof(RTCRayHit));
3    }
```

Listing 14. Sichtbarkeitsberechnung über Embree

GetHits: Ein gültiger Treffer wird wieder über den Parameter t aus Gleichung (4) erkannt, siehe Listing 19.

Die Embree-optimierte Version (Listing 15) der Sichtbarkeitsberechnung entspricht formal der OptiX-Variante aus Listing 9.

```
1  void VisClustering::CalcCompressedVisTBBEmbree(pVisClusterNode &node) {
2          if (node->IsLeaf()) {
3              const uint numRays = SetEmbreeRays(id, useSymmetric);
4              InterSection(numRays);
5              node->Cvis = std::move(GetHits(numRays));
6
7              } else {
8                  tbb::task_group g;
9                  g.run([&]() { if (node->left  != nullptr) CalcCompressedVisTBBEmbree(node->left); });
10                 g.run([&]() { if (node->right != nullptr) CalcCompressedVisTBBEmbree(node->right); });
11                 g.wait();
12                 node->BerechneCompressedVisibility();
13             }
14         }
```

Listing 15. Sichtbarkeitsberechnung mittels Embree und TBB

```
class EmbreeWrapper {

    GeoObjekt *geo;

    uint nfaces;
    public:

    double timeGetHits, timedoEmbree;
    EmbreeWrapper();
    ~EmbreeWrapper();
    EmbreeWrapper(const EmbreeWrapper &);
    void SetGeo(GeoObjekt *);

    RTCDevice device;
    RTCScene scene;
    RTCGeometry mesh;
    RTCIntersectContext context;
    RTCRayHit* rayhit;

    vector<float3> triangleMidpoints;
    vector<float3> triangleNormals;

    void EmbreeSetup();
    void CreateBuffer();
    uint SetEmbreeRays   (const uint &, const bool useSym);

    float3 GetEye(const int &) const;
    float3 GetNormal(const int& id) const;

    IntSet2 GetHits(const int &numRays) const;
    void InterSection(const uint &numRays);
    bool IsHit(const uint &) const;

    inline uint GetNumberOfFaces() const { return nfaces; }

};
```

Listing 16. class EmbreeWrapper

```
1  // Setup Embree
2  void EmbreeWrapper::EmbreeSetup() {
3
4      const uint numVertices = geo->T.P.size();
5      float* vb = (float*) rtcSetNewGeometryBuffer(mesh, RTC_BUFFER_TYPE_VERTEX, 0, RTC_FORMAT_FLOAT3, 3 *
           sizeof (float), numVertices);
6
7      for (unsigned int i=0; i< numVertices; ++i) {
8          const float X = geo->T.P[i].x;
9          const float Y = geo->T.P[i].y;
10         const float Z = geo->T.P[i].z;
11         vb[3*i   ] = X;
12         vb[3*i+1] = Y;
13         vb[3*i+2] = Z;
14     }
15     const uint numFaces = geo->T.V.size();
16     unsigned* ib = (unsigned*) rtcSetNewGeometryBuffer(mesh, RTC_BUFFER_TYPE_INDEX, 0, RTC_FORMAT_UINT3,
           3 * sizeof (unsigned), numFaces);
17
18     for (unsigned int i=0; i< numFaces; ++i) {
19         const uint I=geo->T.V[i].k[0];
20         const uint J=geo->T.V[i].k[1];
21         const uint K=geo->T.V[i].k[2];
22         ib[3*i   ] = I;
23         ib[3*i+1] = J;
24         ib[3*i+2] = K;
25     }
26
27     nfaces = numFaces;
28     rtcCommitGeometry(mesh);
29     rtcAttachGeometry(scene, mesh);
30     rtcReleaseGeometry(mesh);
31     rtcCommitScene(scene);
32     rtcInitIntersectContext(&context);
33     context.flags = RTC_INTERSECT_CONTEXT_FLAG_COHERENT;
34     }
35
36 // Create buffers for rays and hits
37 void EmbreeWrapper::CreateBuffer() {
38
39     const uint N = GetNumberOfFaces();
40     triangleMidpoints.resize(N);
41     triangleNormals.resize(N);
42     const float scene_epsilon  = 0.0001f;
43     for (uint i = 0; i < N; ++i) {
44         triangleMidpoints[i]  = geo->T.M[i].GetFloat3();
45             triangleNormals[i]    = geo->T.N[i].GetFloat3();
46         }
47     rayhit = new RTCRayHit[N];
48     for (unsigned int j = 0; j < N; j++) {
49         const RTCRayHit ray = makeRay(triangleMidpoints[j], triangleNormals[j], scene_epsilon, 1.1f);
50         rayhit[j] = ray;
51 }
```

Listing 17. Setup embree

```
uint EmbreeWrapper::SetEmbreeRays(const uint& id, const bool useSym) {
        // scene_epsilon
        const float scene_epsilon = 0.0001f;
        const float scene_max = numeric_limits<float>::max();
        const float epsfloat = epsFloat();

        const float3 O = triangleMidpoints[id];
        const float3 N = triangleNormals[id];

        float3* tm = triangleMidpoints.data();
        float3* tn = triangleNormals.data();
        const uint oID = OrderID[id];
        const int n = GetNumberOfFaces();

        uint numRays = 0;

        for (uint i = 0; i < n; ++i) {
            const float targetX = tm[i].x;
            const float targetY = tm[i].y;
            const float targetZ = tm[i].z;

            const float3 direction(targetX - O.x, targetY - O.y, targetZ - O.z);

            const float Test1 = -dot_float3(tn[i], direction); // paneel id liegt in H^+ von paneel i
            const float Test2 =  dot_float3(N, direction);     // paneel i  liegt in H^+ von paneel id

            //  Backface Culling
            if ((Test1 > epsfloat) and (Test2 > epsfloat)) {
                makeRay(rayhit[numRays], O, direction, scene_epsilon, 1.1f);
                ++numRays;
            }
        }

        return numRays;
    }
```

Listing 18. Set rays

```
1  bool EmbreeWrapper::IsHit(const uint &i) const {
2          const float scene_epsilon  = 0.0001f;
3          if (rayhit[i].hit.geomID != RTC_INVALID_GEOMETRY_ID)
4          {
5              const bool tfarIsZero = IsZero(rayhit[i].ray.tfar-1.0f,scene_epsilon);
6              return tfarIsZero;
7          }
8          else
9              return false;
10     }
11
12 IntSet2 EmbreeWrapper::GetHits(const int &numRays) const {
13
14     uint counter=0;
15     uint maxHit=0;
16     uint minHit=0;
17
18     for (uint i=0; i < numRays; ++i) {
19             if (IsHit(i))
20             { ++counter; maxHit=i; }
21             else
22             { if (counter==0) minHit=i; }
23         }
24     IntSet2 result; result.reserve( maxHit-minHit+1 );
25     for (uint i=minHit; i < maxHit; ++i) {
26             const uint primId = rayhit[i].hit.primID;
27             if ( ( primId < numeric_limits<uint>::max()) and IsHit(i) )
28             {
29                 result.push_back(rayhit[i].hit.primID);
30             }
31     return result;
32 }
```

Listing 19. GetHits

4. Vergleiche

Die Verfahren wurden an 4 unterschiedlich großen virtuellen Modellen getestet. Die für die OptiX-Tests genutzte Grafikkarte RTX 2060 besitzt 1920 Cuda-Kerne. Die Workstation mit CPUs von den Herstellern Intel und AMD verfügen jeweils über 16 Kerne[7].

	Testproblem 1	Testproblem 2	Testproblem 3	Testproblem 4
Name	RefBox	Greenhouse	Unterboden	Gesamtfahrzeug
Anzahl Dreiecke	155640	709042	1015890	4691854
host	Workstation			Cluster
Referenzzeit [s]	614.2	20549	12960	20845
Speedup OptiX (RTX 2060)	7.3	7.5	7.2	–
Speedup Embree	7.5	–	4.1	–
Speedup Embree + MPI	7.4	8.9	8.1	11.1

Die Workstation mit der RTX 2060 Grafikkarte ist gegenüber den Rechnern mit der Embree-Version etwas im Nachteil, da diese teilweise moderne CPU-Architekturen aufweisen. Die Rechnungen zur moderneren GPU RTX 3080 konnten aus den im Abschnitt 3.1 genannten Gründen nicht durchgeführt werden. Die RTX 3080 hat 8704 Cuda-Kerne, das ist gegenüber der RTX 2060 die mehr als vierfache Anzahl, sodass hier ein Speedup-Faktor 20 erwartet werden kann.
Die neuen Algorithmen sind dem alten Verfahren deutlich überlegen. Besonders beeindruckend ist das vierte Beispiel. Hier konnte die Rechenzeit von 5 h 45 min auf ca. 30 min reduziert werden. Die Embree-Variante nutzt 16 Cores, während OptiX die Rechnung auf 1920 Cuda-Kernen verteilt. So gesehen sind die Speedup-Faktor für den Embree-Sichtbarkeitstest schon bemerkenswert, zumal keine weiteren Anforderungen an die Hardware gestellt werden.

[7]Für den Clustertestfall wurden 480 CPU-Kerne angefordert.

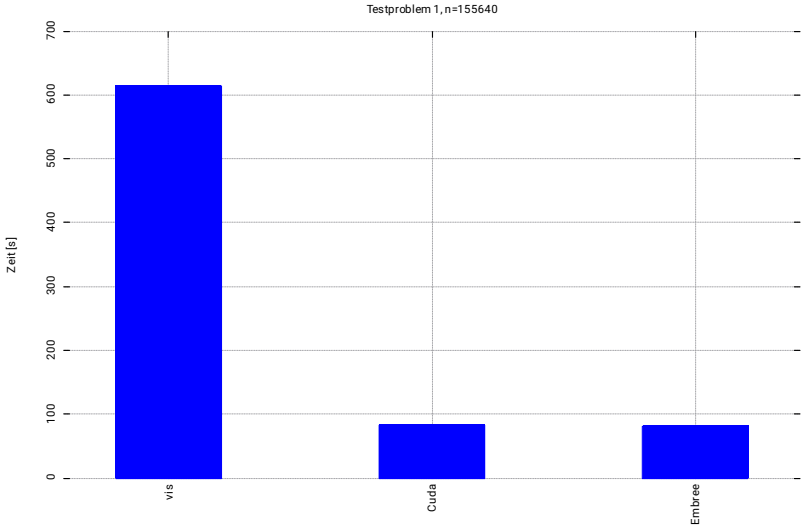

Abbildung 4. Rechenzeiten Testproblem 1

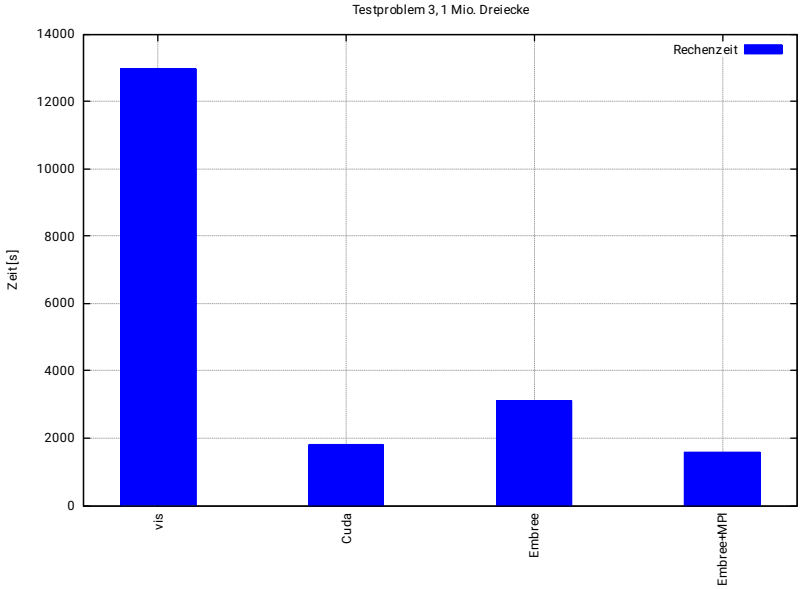

Abbildung 5. Rechenzeiten Testproblem 3

Literatur

[1] I. G. Graham, L. Grasedyck, W. Hackbusch, and S. A. Sauter. Optimal panel-clustering in the presence of anisotropic mesh refinement. *SIAM J. Numer. Anal.*, 46(1):517–543, 2007.

[2] W. Hackbusch. *Integralgleichungen: Theorie und Numerik*. Teubner, Stuttgart, 1997.

[3] W. Hackbusch. Panel Clustering Techniques and Hierarchial Matrices for BEM and FEM. Technical Report 71, Max-Planck-Institut für Mathematik in den Naturwissenschaften, Leipzig, 2003.

[4] W. Hackbusch. *Hierarchische Matrizen. Algorithmen und Analysis.* Springer, 2009.

[5] Intel. High performance ray tracing. `https://www.embree.org/`, 2022. [Online; Stand 24. Mai 2022].

[6] Intel. Intel oneapi threading building blocks. scalable parallel programming at your fingertips. `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onetbb.html#gs.193v8j`, 2022. [Online; Stand 24. Mai 2022].

[7] Andreas Johannsen and Michael B. Carter. Clustered backface culling. *J. Graphics, GPU, & Game Tools*, 3:1–14, 1998.

[8] NVIDEA. Nvidia optix^{TM} ray tracing engine. `https://developer.nvidia.com/rtx/ray-tracing/optix`, 2022. [Online; accessed 24-May-2022].

[9] R. Siegel and Howell. *Thermal Radiation Heat Transfer*. Taylor & Francis, New York, London, 4th Edition edition, 2002.

[10] Wikipedia. Flynnsche Klassifikation — Wikipedia, die freie Enzyklopädie. `de.wikipedia.org/w/index.php?title=Flynnsche_Klassifikation&oldid=164587902`, 2017. [Online; Stand 11. Februar 2019].

GWR Gesellschaft für Wissenschaftliches Rechnen mbH, Potsdamer Strasse 18a, 15413 Teltow

E-mail address: `liebau@gwr-mbh.de`