Architektur und Programmierung von Grafik- und Koprozessoren

Rendering Algorithmen

Stefan Zellmann

Lehrstuhl für Informatik, Universität zu Köln

SS2018

Konstruktion von Hierarchien

Top-Down Konstruktion von Binärbäumen (z. B. BVHs)

```
function BaueHierarchie(Knoten)
    if Split(Knoten,AABB_L,AABB_R) then
        L ← Gruppiere(AABB_L,Knoten.Primitive)
        R ← Gruppiere(AABB_R,Knoten.Primitive)
        BaueHierarchie(L)
        BaueHierarchie(R)
    else
        Blatt(Knoten.Primitive)
    end if
end function
```

Konstruktion von Hierarchien

Funktion SPLIT(Knoten,AABB_L,AABB_R) entscheidet über Güte der BVH.

- Einfache Heuristiken wie "Median Split".
- BVHs mit hoher Güte mittels "Surface Area Heuristik".
  - Kostenfunktion, um zu entscheiden, ob Split oder nicht.
  - Top-Down Konstruktion: *lokale* Optima.

Konstruktion von Hierarchien

Surface Area Heuristic

Kostenfunktion setzt Fläche der Primitive im Parent Knoten zur Außenfläche der Umbox ins Verhältnis. Gegeben: N Dreiecke in einem bereits konstruierten Teilbaum mit Volumen V. Für Partitionierung L und R mit jeweils N_L und N_R Dreiecken sowie Volumen V_L und V_R, bestimme Kosten:

$$C := C_T + C_I \left(\frac{SA(V_L)}{SA(V)} N_L + \frac{SA(V_R)}{SA(V)} N_R \right). \tag{37}$$

C_T und C_I sind jeweils konstante, applikationsspezifische Kosten, um das Traversieren eines Strahls nach unten in der Hierarchie, bzw. den Schnitttest Strahl / Dreieck zu bewerten. Mit $SA()$ wird die Umfläche von Volumen V ermittelt.

Konstruktion von Hierarchien

Surface Area Heuristic

Greedy Heuristik:

1. Bestimme Split Ebenen Kandidaten (heuristisch, z. B. *Sweeping* entlang der Hauptachsen; diskrete Positionen entlang Hauptachsen ("*Binning*")).
2. Für alle Kandidaten, bestimme SAH-Kosten C.
3. Wähle Split Ebene mit niedrigstem C und bestimme AABB_L und AABB_R als Umboxen um die N_L und N_R Dreiecke.

Konstruktion von Hierarchien

Surface Area Heuristic

- **Sweeping**: bewege Kandidatenebene *stetig* durch Box.
- Wenn kostenoptimale Ebene gefunden, *partitioniere* ($O(n)$, vgl. Quicksort) Dreiecke *in place* in linken und rechten Teilbaum.
- Besonders aufwendig, wenn obere Levels des Baums aufgebaut werden, da viele Dreiecke zu partitionieren.
- Offensichtliche Vereinfachung: kein *stetiges* Sweeping, sondern orientiere Ebene an Dreieckseckpunkten.
- Weitere Vereinfachung: **Binning** - unterteile AABB in Bins und orientiere Kandidatenebenen daran. Projiziere *Mittelpunkte der Dreiecksumboxen* in Bins.

Konstruktion von Hierarchien

Surface Area Heuristic

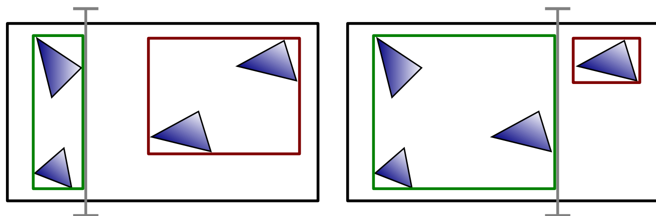


Abbildung: Zwei Kandidatenebenen. Berechne Umboxen für L und R und bewerte Kandidatenebene mit C. Kandidatenebene mit niedrigstem C determiniert Split.

Konstruktion von Hierarchien

Surface Area Heuristic

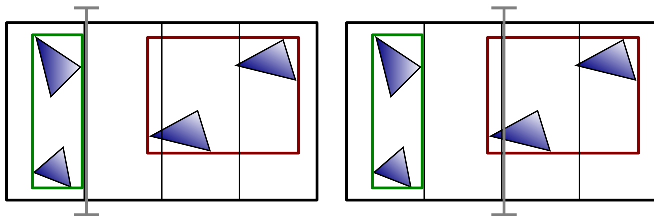


Abbildung: *Binning* diskretisiert das Verfahren. Auf höheren Ebenen weniger Dreiecke zu partitionieren.

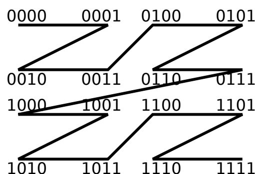
Konstruktion von Hierarchien

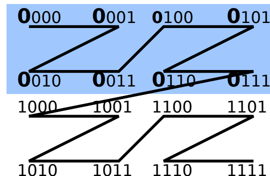
Surface Area Heuristic - Bemerkungen

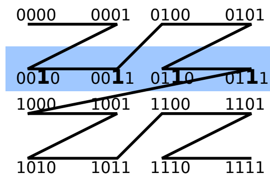
- Greedy Heuristik bestimmt nur lokale Optima. Refinement Algorithmen existieren, die nachträglich den Baum umstrukturieren (z. B. *Tree Rotations*, um Surface Area Kosten weiter zu senken ($\Rightarrow$ höhere Konstruktionskosten).
- SAH-basierte BVH Konstruktion derzeit State-of-the-Art bzgl. Traversal.

Konstruktion von Hierarchien

Bottom-Up Konstruktion







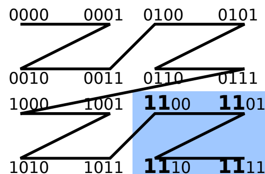


Abbildung: Morton Codes implizieren Hierarchien (vgl. Pharr, Jakob, Humphreys: Physically Based Rendering (2017)).

Konstruktion von Hierarchien

Bottom-Up Konstruktion

- Familie von Konstruktionsalgorithmen in linearer Zeit.
- *Linear BVH* (LBVH) Algorithmus:
  1. Berechne Umbox für jedes Dreieck.
  2. Berechne Liste mit 30-bit 3D Morton Codes bzgl. der *Mittelpunkte der Umboxen*.
  3. Lineares Sortieren der Mittelpunkte mit Radix Sort (z. B. auf GPU).
  4. Median Split: bestimme Split an der ersten Stelle, an der sich die signifikantesten Bits der Morton Codes unterscheiden. (Finde mittels Binärsuche).
- Niedrige Qualität, dafür sehr schnelle Konstruktion.
- LBVH Algorithmus Grundlage für Reihe von State-of-the-Art Konstruktionsalgorithmen (HLBVH, TR-BVH).

Konstruktion von Hierarchien

Bemerkungen

- Top-Down Konstruktion $\Rightarrow$ langsam, wenig Parallelismus auf Root-Level.
- Bottom-Up Konstruktion $\Rightarrow$ Bäume mit niedriger Qualität.
- Häufig hybride Verfahren, um Konstruktions- und Traversierungskosten zu balancieren:
  - Bottom-Up Konstruktion von Teilbäumen auf unterer BVH-Ebene ("*Treelets*").
  - Auf oberer Ebene, sortiere Treelets mit Surface Area Heuristik.
  - Gegenstand aktueller Forschung. Suchbegriffe, um Forschungsaufsätze zu finden: "Bonsai-BVH", "HLBVH", "TR-BVH", "ATR-BVH".

Real-Time Ray Tracing

- Erste CPU Real-Time Ray Tracer Anfang der 2000er.
- Etwa die Zeit, als SIMD Vektor Units Standard in CPUs wurden (MMX, SSE).
- Hochoptimierte Programme, zahlreiche Einschränkungen ggü. regulären Ray Tracern (damals z. B. PovRay etc.)

Real-Time Ray Tracing

1.) Objektorientierung

```
class Primitive {
    virtual bool intersect() = 0;
    virtual vec3 get_normal() = 0;
};
class Triangle {
    bool intersect() { /* ... */ }
    vec3 get_normal() { /* ... */ }
};

class Material {
    virtual vec3 shade() = 0;
    virtual vec3 sample() = 0;
};
class Metal {
    vec3 shade() { /* ... */ }
    vec3 sample() { /* ... */ }
};
```

Real-Time Ray Tracing

1.) Objektorientierung

- Problem: *Late Binding* $\Rightarrow$ kein Optimierungspotential für Compiler.
- Real-Time Ray Tracing: Optimierungspotential durch *Inlining* kleiner Funktionen.
  - Lineare Algebra Funktionen.
  - Material Shading Funktionen / BRDFs.
  - ...
- Handoptimierte Codes können um Größenordnung schneller sein als objektorientierte Codes.

Real-Time Ray Tracing

2.) Dynamisches Branching

```
class Primitive {
    virtual bool intersect() = 0;
    virtual vec3 get_normal() = 0;
};
class Triangle {
    bool intersect() { /* ... */ }
    vec3 get_normal() { /* ... */ }
};
```

Polymorphe Vererbung $\Rightarrow$ Branching.

Bei Strahl/Dreiecksschnitttest: Branching in innerster Schleife.

Real-Time Ray Tracing: eliminiere Branch, unterstütze nur Dreiecke!

Real-Time Ray Tracing

3.) SIMD

- **Strategie 1**: Traversiere *Bündel* (auch: *Strahlpakete* von N (=SIMD-Breite) Strahlen durch 3-D Szene. Problem: Divergenz - problematischer, je zufälliger einzelne Strahlen im Bündel verzweigen.
  - Ideal für Algorithmus PRIMAERSTRAHLVERFOLGUNG.
  - Whitted Algorithmus recht kohärent.
  - Problematisch: stochastisches Sampling.
- **Strategie 2**: Traversiere Einzelstrahlen, dafür BVHs mit N Bounding Boxen pro innerem Knoten und N Dreiecken pro Blatt.
- **Hybride Strategien**: Bündel für obere BVH Ebenen, Einzelstrahlen für untere Ebenen, auf denen Divergenz höher ist.

Volume Rendering

Annahme bisher: nur Oberflächen, keine teilnehmenden Medien (Gas, Nebel etc.). Annahme in vielen Situationen zu restriktiv.

Volume Rendering: betrachte virtuelle Welt bzgl. des gesamten Volumens. Macht Rendering unlängs komplexer.

Volume Rendering elegant auf Strahlverfolgungsmethoden abbildbar.

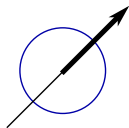
Volume Rendering

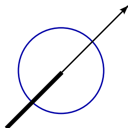
Wir betrachten ein *Partikelmodell*, mit Hilfe dessen die virtuelle Welt beschrieben wird. Im Rahmen der Lichttransportgleichung tun wir dies auch bereits, jetzt aber Annahme, dass Partikel *überall* im Raum.

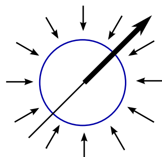
Massebehaftete Partikel interagieren mit Photonen.

Massebehaftete Teilchen können selbst Photonen *emittieren* oder sie *absorbieren*. Außerdem kann es zu Streuung kommen, und zwar in Richtung des massebehafteten Teilchens (Einwärtsstreuung / In-Scattering) oder vom massebehafteten Teilchen weg (Auswärtsstreuung / Out-Scattering).

Volume Rendering







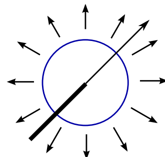


Abbildung: Photon bewegt sich entlang eines Strahls, interagiert dabei mit massebehafteten Partikeln (Kreisursprung). Dabei können verschiedene Phänomene auftreten; von links nach rechts: Emission, Absorption, Einwärtsstreuung, Auswärtsstreuung; vgl. Engel et al.: Real-Time Volume Graphics (2006).

Volume Rendering

- Wir betrachten wieder die Strahlstärke, d. h. wir interessieren uns für den Anteil des Lichts, der von allen Photonen ausgeht, die entlang des gleichen Strahls verlaufen.
- Emission und Einwärtsstreuung führen dazu, dass Strahlstärke in Strahlrichtung *zunimmt*.
- Absorption und Auswärtsstreuung führen zu *Abnahme* der Strahlstärke.

Volume Rendering

Absorption + Emission Modell

Betrachte nur die Phänomene Absorption und Emission. Dazu wollen wir an jedem Punkt im Raum entlang des Lichtstrahls die *Partikeldichte* ermitteln und diese entlang des gesamten Strahls integrieren.

Mit der Partikeldichte ([0..1]) schlagen wir in einer Tabelle (*Transferfunktion*) nach, die uns statistisch sagt, wie viel Licht an der Stelle absorbiert und emittiert wird.

Volume Rendering

Absorption + Emission Modell

Bestimme Partikeldichte an Punkt entlang Strahl. Dazu betrachten *kugelrunde* Partikel mit Radius r. Im Einheitsvolumen (!) befinden sich ρ Partikel (ρ variiert räumlich, $\rho(t)$).

Wir betrachten einen dünnen Zylinder mit Kreisfläche E und Höhe Δt. Wir projizieren die Fläche $A = \pi r^2$ der im Zylinder befindlichen Partikel auf die Kreisfläche. Nehmen wir vereinfachend an, dass die projizierten Partikel nicht überlappen, dann ist die Fläche, die von diesen verdeckt wird, $\rho A E \Delta t$ (Anzahl $\times$ projizierte Fläche $\times$ Zylindervolumen). Der Anteil des Lichts, der entlang des Strahls an dem Punkt *absorbiert* wird, ist $\rho A E \Delta t / E = \rho A \Delta t$.

Volume Rendering

Absorption + Emission Modell

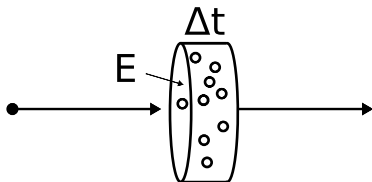


Abbildung: vgl. Nelson Max: Optical Models for Direct Volume Rendering (1995).

Wir lassen $\Delta t \Rightarrow 0$. Dann ergibt sich *bzgl. Absorption* die Differentialgleichung für die Lichtintensität I in Entfernung t entlang des Strahls:

$$\frac{dI}{dt} = -\rho(t) A I(t). \tag{38}$$

Volume Rendering

Absorption + Emission Modell

Differentialgleichung:

$$\frac{dI}{dt} = -\rho(t)AI(t), \tag{39}$$

Lösung:

$$I(t) = I_0 e^{-\int_0^t \rho(t)Adt}. \tag{40}$$

I_0 bezeichnet dabei die Lichtintensität bei Distanz $t = 0$ (Stelle, an der Strahl in Volumen eintritt). Die Lösung der Differentialgleichung gibt Anteil des Lichts an, der entlang Strahl in Distanz t durchgelassen wird.

Dieser Zusammenhang ist als Beersches Gesetz (Beer's Law) bekannt.

Volume Rendering

Absorption + Emission Modell

Nimmt man an, dass Partikel außerdem Licht emittieren können, ergibt sich für Emission:

$$I(t) = I_0 + \int_0^t C(u)\rho(u)A\,du, \qquad (41)$$

wobei C die räumlich variierende Lichtintensität in Entfernung u durch glühende Partikel bezeichnet.

Volume Rendering

Absorption + Emission Modell

Kombiniert man die beiden Modelle, ergibt sich für die Lichtintensität:

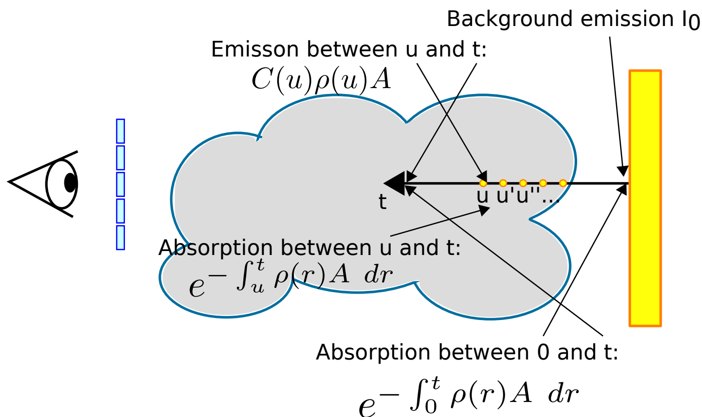
$$I(t) = I_0 e^{-\int_0^t \rho(r) A \; dr} + \int_0^t C(u) \rho(u) A e^{-\int_u^t \rho(r) A \; dr} du. \qquad (42)$$

Beim Absorption + Emission Modell muss für jede (gesampelte) Entfernung t entlang eines geraden (i. d. R. Primär-)Strahls der Emissionsanteil am Punkt $\mathbf{o} + \vec{d}t$ bestimmt und um Absorption entlang des Stücks t gewichtet werden.

Volume Rendering

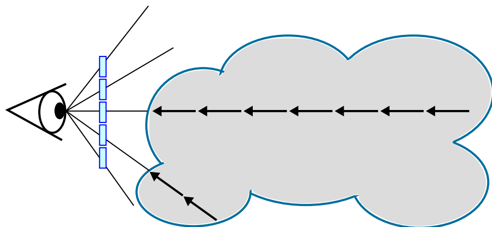
Absorption + Emission Modell

$$I(t) = I_0 e^{-\int_0^t \rho(r)A\ dr} + \int_0^t C(u)\rho(u)Ae^{-\int_u^t \rho(r)A\ dr}du.$$



Volume Rendering

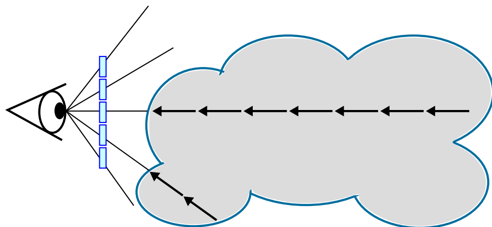
Absorption + Emission Modell



Das optische Modell Absorption + Emission lässt sich mit Riemann Integration etc. bestimmen. Dazu wählt man eine geeignete Sampling Distanz Δt und “marschiert” entlang eines Primärstrahls in Δt Schritten.

Volume Rendering

Absorption + Emission Modell



Dabei akkumuliert man RGBA Farben mit Hilfe des weiter oben beschriebenen Over Operators für teiltransparente Geometrie und schlägt Absorptions- und Emissionskoeffizienten in Transferfunktion nach.

Für ausführlichere Betrachung vgl. Max: Optical Models for Direct Volume Rendering (1995).

Volume Rendering

- Absorption + Emission omnipräsent in Wissenschaftlicher Visualisierung.
  - Medizin: CT & MRI.
  - Meteorologie.
  - ...
- Streuung unlängs aufwendiger zu berechnen, setzt u. a. Phasenfunktion voraus, die (ähnlich wie BRDF für Oberflächen) Streuverhalten an Punkt im Raum beschreibt.
- Streuung: kein Ray Marching mit Strahlen gleicher Richtung mehr, stattdessen Monte Carlo Integration etc.

Paralleles Rendering

Paralleles Rendering

Verschiedene Ebenen:

- Parallelismus innerhalb von GPU.
- Paralleler Software Ray Tracer, parallel über alle Strahlen.
- Verteilte Speichersysteme.

Paralleles Rendering

Verteilter Speicher

- Wir interessieren uns für Paralleles Rendering mit verteiltem Speicher. Jeder Prozessor hat nur Speicher für einen Teil der Geometrie.
- Jeder Prozessor z. B. zuständig für Teil der Geometrie; Teil des zu rendernden Bildes; etc.
- Beispiel: Simulation auf Supercomputer, Simulationsergebnis liegt geteilt auf Cluster Knoten vor. Nun Post-Processing (Visualisierung) auf Cluster.

Ziel: Latenzvermeidung

Oberstes Gebot: reduziere Kommunikations-Overhead, da Netzwerkkommunikation teuer.

Abwägung: verteile Geometrie, verteile Teilbilder, etc.

Entscheidung von verschiedenen Parametern abhängig (wird z. B. Animation gerendert, gibt es evtl. zeitliche Kohärenz; Lastverteilung, "sparse" besetzte Daten ggf. anders zu behandeln als voll besetzte Datenstrukturen, etc.).

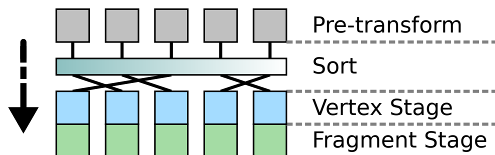
Molnars Taxonomie

Steve Molnar: A Sorting Classification of Parallel Rendering, Siggraph 1994.

- Paralleles Rendering als *Sortierproblem*:
  - Sortiere untransformierte oder teiltransformierte Geometrie.
  - Sortiere in Bildschirmkoordinaten transformierte Geometrie.
  - Sortiere Pixel oder Fragmente.
- Unterscheide danach, *wann* sortiert wird.
- Taxonomie orientiert sich an Algorithmus RASTERISIERUNG. Spielt bei GPU Hardware Design große Rolle.
- Auch auf Software Rendering anwendbar (z. B. GPU Cluster, Software sortiert und verteilt Daten per Netzwerk).

Molnars Taxonomie

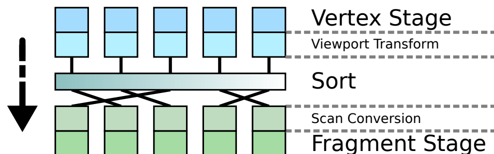
Sort-First



- Verteile im wesentlichen untransformierte *Primitive* auf Prozessoren.
  - Anfangs Verteilung beliebig.
  - *Pre-Transform*: transformiere Primitive gerade so weit, um zu wissen, welcher Fensterbereich überlappt (ggf. wird Transformation nicht einmal angewendet).
  - Umverteilung (Sort) an Prozessoren, die für Fensterbereich zuständig.
  - Frame-to-Frame Kohärenz $\Rightarrow$ ggf. nur anfangs hohe Last auf Interconnect.

Molnars Taxonomie

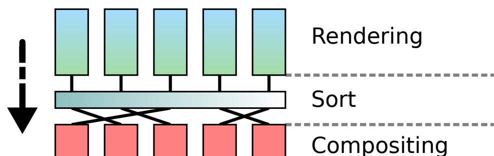
Sort-Middle



- Verteile *Primitive in Bildschirmkoordinaten* auf Prozessoren (Raster Engines).
  - Gesamte Geometrie Phase und Primitive Assembly durchlaufen.
  - Sort vor Scan Conversion.
  - Sort-Middle typisch für Hardware Rendering, kaum auf Software Rendering anwendbar.

Molnars Taxonomie

Sort-Last



- Verteile vollständig gerenderte Fragmente oder Pixel auf Prozessoren.
  - Erst vollständige Rendering Phase (egal welcher Algorithmus).
  - Sende *Intermediärbilder* über Interconnect an zuständige Prozessoren (Sort).
  - Compositing auf einem oder mehreren Prozessoren.

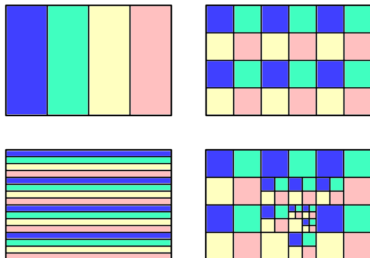
Molnars Taxonomie

Compositing

- Prozessoren für Teile der Geometrie zuständig - verwalte z. B. mittels k-d tree.
- Mit Hilfe des k-d trees können die Intermediärbilder tiefensortiert werden.
- Teiltransparente Geometrie: *Alpha Compositing* - Intermediärbilder mit Alpha Kanal, Compositing mit Over Operation (vgl. Alpha Blending).
- Opake Geometrie: *Depth Compositing*: Intermediärbilder mit Tiefenkanal, setze auf Basis von Tiefeninformation zusammen.

Molnars Taxonomie

Lastverteilung



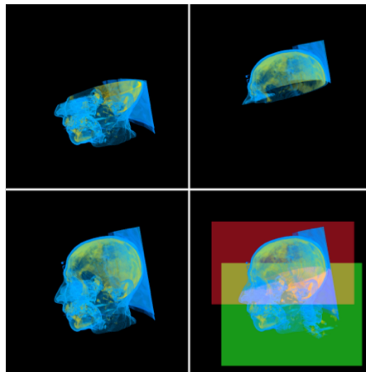


Abbildung: Links: Sort-First statische und dynamische Lastverteilung, rechts: Sort-Last mit zwei Prozessoren ohne Lastverteilung.

Molnars Taxonomie

Dynamische Lastverteilung

Bei dynamischer Lastverteilung und z. B. Sort-First wird das Bild a priori in Bildausschnitte unterteilt, die Zuweisung von Bildausschnitten zu Prozessoren geschieht jedoch dynamisch.

Bildausschnitte (z. B. Kacheln) können dazu etwa in eine Schlange einsortiert werden, Prozessoren holen Kachel aus Schlange, sobald Arbeit verrichtet.

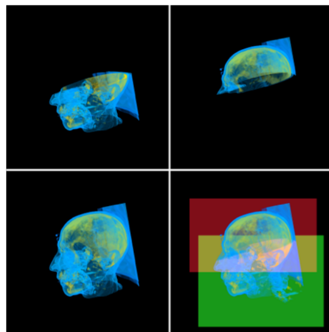
Kann mit adaptiven Verfeinerungsverfahren gepaart werden (arbeitsintensivere Regionen werden dynamisch in kleinere Kacheln unterteilt).

Bemerkung: komplizierte Lastverteilungsalgorithmen können ungünstiges Branching Verhalten etc. mich sich bringen.

Sort-Last Compositing Algorithmen

- Bei vielen Prozessoren ist nicht Rendering das Bottleneck, sondern Compositing.
- Problem ist weniger die Komplexität des Verfahrens, sondern der hohe Bandbreitebedarf.
- Beim naiven Verfahren: Bandbreitebedarf verteilt sich ungleichmäßig auf Interconnect, Verbindung zum Anzeigeknoten überlastet.

Sort-Last Compositing Algorithmen

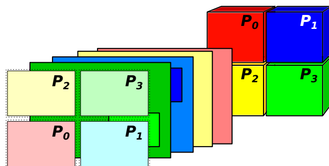


I. allg. werden beim Sort-Last Verfahren in der letzten Phase vollaufgelöste Bilder zwischen Prozessoren hin- und her geschickt.

Naiv: jeder Prozessor rendert vollaufgelöstes Bild und schickt es zum Compositing an *Anzeigeknoten*. Schlechtes Skalierungsverhalten schon bei moderat vielen Prozessoren.

Sort-Last Compositing Algorithmen

Direct-Send



Jeder von P Prozessoren für $\frac{1}{P}$ Teil des Bilds verantwortlich. Alle Prozessoren rendern, senden danach $(P-1) \times \frac{1}{P}$ Bildausschnitte an die P anderen Prozessoren. Jeder Prozessor führt Compositing für seinen Bildteil durch. Anzeigeknoten sammelt Bilder ein.

Sort-Last Compositing Algorithmen

Direct-Send

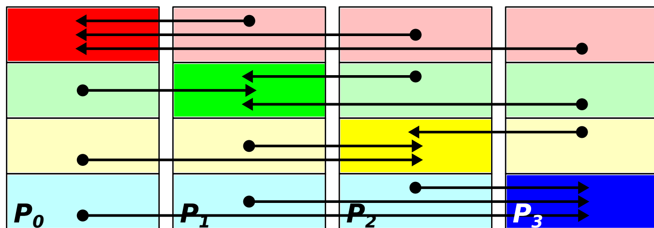


Abbildung: Kommunikationsmuster beim Direct-Send Compositing, vgl. Bethel et al.: High Performance Visualization - Enabling Extreme Scale Scientific Insight (2013).

Sort-Last Compositing Algorithmen

Baumbasiertes / Rundenbasiertes Compositing

- Teile Compositing in *Runden* auf, jede Runde entspricht einem Level in Baum.
- Direct-Send: Baum der Höhe 1, alle Arbeit innerhalb einer Runde.
- Ziel: bessere balance zwischen Anzahl gleichzeitiger Nachrichten sowie Bandbreitenbedarf einzelner Nachrichten.

Sort-Last Compositing Algorithmen

Baumbasiertes / Rundenbasiertes Compositing

- Naive Implementierung: schicke in jeder Runde Intermediärbild an direkten Nachbarn. Dieser führt Compositing durch. In der nächsten Runde ist aus dem Prozessorpaar nur noch der Nachbar aktiv.
- Am Ende liegt das zusammengesetzte Bild im Speicher *eines* Prozessors.

Sort-Last Compositing Algorithmen

Baumbasiertes / Rundenbasiertes Compositing

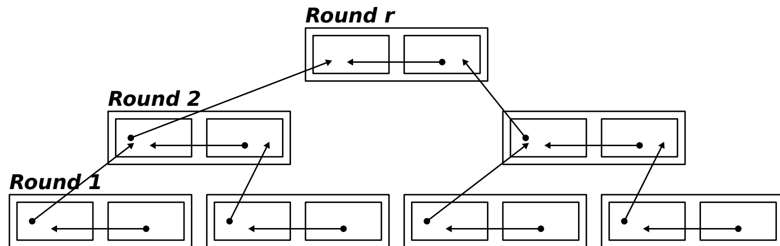


Abbildung: Problem bei naiv implementiertem rundenbasiertem Compositing (und generell bei binärbaumbasierten parallelen Algorithmen wie Reduce): Viele Prozessoren, die in frühen Runden arbeiten, sind in späteren Runden nicht beschäftigt.

Sort-Last Compositing Algorithmen

Binary-Swap

Binary-Swap Compositing löst dieses Problem,

- indem pro Runde zwei Prozessoren paarweise für eine Hälfte ihres *gemeinsamen* Bildbereichs zuständig sind (*binary*),
- und indem diese die Bilddaten, für die jeweils der andere Prozessor zuständig ist, in jeder Runde tauschen (*swap*).

In jeder Runde wird zudem der Bildausschnitt, für den zwei Prozessoren zuständig sind, halbiert. Außerdem wird die Distanz zwischen den Prozessorpaaren verdoppelt ("distance-doubling and vector-halving" Algorithmus).

Es sind stets P Prozessoren beschäftigt, wobei $P = 2^k$.

Ist der Binary-Swap Algorithmus durchgelaufen, liegt das zusammengesetzte Bild *verteilt* auf P Prozessoren vor.

Sort-Last Compositing Algorithmen

Binary-Swap

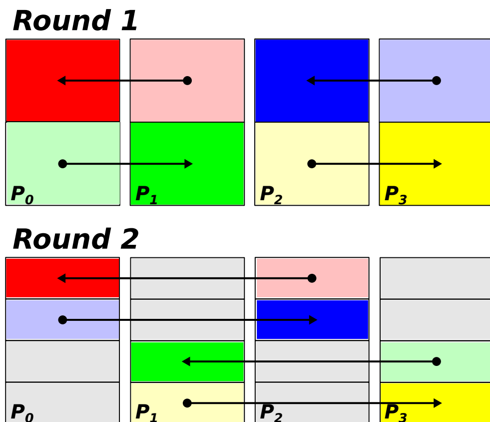


Abbildung: Kommunikationsmuster beim Binary-Swap Compositing mit vier Prozessoren, vgl. Bethel et al.: High Performance Visualization - Enabling Extreme Scale Scientific Insight (2013).

Sort-Last Compositing Algorithmen

Binary-Swap

- Bemerkung: in jeder Runde führen Paare von Prozessoren Direct-Send durch.
- Bemerkung: der Binary-Swap Algorithmus skaliert nur für $P = 2^k$ Prozessoren.

Sort-Last Compositing Algorithmen

Verallgemeinerung

Wir wollen die Algorithmen *Direct-Send* und *Binary-Swap* verallgemeinern. Dazu führen wir die folgende Notation ein.

Bezeichne r die Anzahl Runden, die der Compositing Algorithmus durchführt. Bezeichne k_i die Anzahl an Prozessoren, die in jeder Runde in jeder Kommunikationsgruppe beteiligt ist. Es ergibt sich der “k-Vektor” $\vec{k} = [k_1, k_2, ..., k_r]$.