# Parallell beräkning av omslutande volymer

Olov Winberg Mattias Karlsson

Januari 2010

Bachelor Thesis in Computer Science

Handledare och examinator: Thomas Larsson
Mälardalens Högskola
Akademin för innovation, design och teknik

# Parallel Computation of Bounding Volumes


## Abstract

This paper presents techniques for speeding up commonly used algorithms for bounding volume (BV) computation, such as the AABB, sphere and $k$-DOP. By exploiting the possibilities of parallelism in modern processors, the result exceeds the expected theoretical result. The methods focus on data-level-parallelism (DLP) using Intel's SSE instructions, for operations on 4 parallel independent single precision floating point values, with a theoretical speed-up factor of 4 on data throughput. Still, a speed-up between 7–9 are shown in the computation of AABBs and $k$-DOPs. For the computation of tight fitting spheres the speed-up factor halts at approximately 4 due to a limiting data dependency. In addition, further parallelization by multithreading algorithms on multi-core CPUs shows speed-up factors of 14 on 2 cores and reaching 25 on 4 cores, compared to non parallel algorithms.

# Sammanfattning

Denna rapport behandlar tekniker för att snabba upp vanligt använda algoritmer för beräkning av omslutande volymer, såsom box, sfär och $k$-DOP. Genom att utnyttja möjligheten till parallellism i dagens processorer ges resultat som överstiger det förväntade teoretiska. Fokus ligger framförallt på dataparallellism baserat på Intels SSE-instruktioner. Dessa erbjuder operationer för parallell behandling av fyra oberoende värden, vilket ger en teoretisk uppsnabbning på 4. Trots detta påvisas uppsnabbningar på mellan 7-9 gånger för box och $k$-DOP samtidigt som sfären uppnår en uppsnabbning av 4 på grund av ett begränsande databeroende. Vidare utforskas möjligheten att ytterliggare utnyttja parallellism genom multitrådade algoritmer på flerkärninga processorer. Jämfört med en ickeparallell implementering ges en uppsnabbning på upp till 14 gånger på 2 kärnor samt 25 gånger på 4 kärnor.

# Innehåll

# 1 Omslutande volymer

## 1.1 Introduktion

En omslutande volym (Bounding Volume) är en volym som kapslar in en eller flera volymer av mer komplex natur. Syftet är att snabba upp geometriska beräkningar på komplexa volymer genom att inkapsla dessa i enklare volymer. Att genomföra exempelvis skärningstest mellan komplexa objekt är kostsamt och i realtidsapplikationer i många fall inte realistiskt. Med omslutande volymer kan effektiva kollisionstest initialt ske och bara då testet ger ett positivt resultat behöver skärningen mellan de inneslutna komplexa objekten ske. I de fall då en skärning mellan objekten sker har det initiala testet tagit onödig beräkningskraft. I normalfallet är det dock få objekt som överlappar så det initiala, enkla testet medger stor prestandaökning. Förutom kollisionstester kan omslutande volymer användas för att bland annat accelerera strålföljning (Ray-Tracing) samt utsortering av dolda objekt (Culling). För en översikt av användningsområde för omslutande volymer se [AMHH08].

Som omslutande volym används typiskt boxar och sfärer då de tack vare sin enkelhet är snabba att beräkna och ger snabba geometriska tester. Att accelerera framtagningen av volymerna ytterligare är önskvärt då de kan komma att beräknas i realtid. I de fall förberäkning är möjlig är beräkningssnabbhet viktig för att sänka laddningstider.

I denna rapport undersöks möjligheten att accelerera beräkningen av volymer genom dataparallellism med SIMD som är väl anpassat för att snabba upp olika geometriska beräkningar. Flera föreslagna användningsområden finns för SIMD, exempelvis inom interaktiv strålföljning (Ray-Tracing) [WBS07] och parallella överlappningstest, till exempel mellan sfärer och boxar [Eri04,LAML07]. För en generell översikt av användningsområden se [HOM08].

Förutom dataparallellitet genom SIMD undersöks även möjligheten att ytterligare accelerera volymberäkningen genom multitrådning på flerkärniga processorer. Arbete för att snabba upp mer komplexa strukturer, hierarkier av omslutande volymer (Bounding Volume Heirarchy) har gjorts [WIP08].

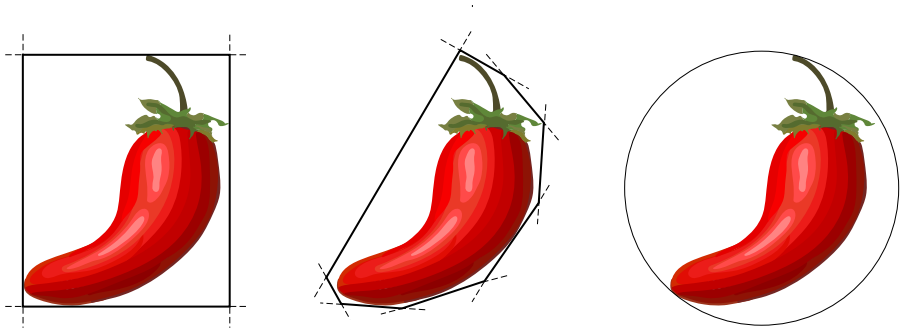
Det finns många olika typer av omslutande volymer som alla har sina styrkor och svagheter. Egenskaper som eftersträvas hos effektiva omslutande volymer är [Eri04]:

- Enkel och snabb generering av volym
- Enkla och snabba skärningstest
- Tät passform
- Litet minnesbehov
- Enkel att rotera och flytta

En tumregel är att ju tätare en volym är, desto mer komplicerad är den att generera, och desto mer krävande blir överlappstest samt minnesbehov. Däremot ger den tätare volymen färre falska överlappningar, det vill säga då de omslutande volymerna överlappar trots att de inneslutna objekten inte gör det.

## 1.2 Polyeder som omslutande volym

De flesta vanligen använda omslutningsvolymerna, med undantag för sfären, är konvexa polyedrar bestående av konvexa polygoner. Dessa har gemensamt att de kan beskrivas som en mängd av plan vars skärningar begränsar volymen. Figur 1 visar flera olika exempel av volymer i 2D.



**Figur 1:** Olika volymer i 2D.

En speciell typ av polyeder är $k$-DOP som kännetecknas av att varje plan i volymen är parallell med ett motstående plan (i undantagsfall kan ett plan utgöras av en punkt). Då planparen beskrivs av en gemensam ytnormal kommer $k/2$ normaler beskriva en volym med $k$ begränsande plan. Dessa par av plan kallas vanligen för slabs (se figur 2(b)), som förutom normalen behöver två avstånd från en fast punkt, vanligen origo, för att beskrivas. $k$-DOP (Discrete Orientation Polytope) innebär att volymen definieras av normaler som är fördefinierade (se avsnitt 1.4 $k$-DOP). En speciell variant av $k$-DOP är den vanligare AABB (Axis Aligned Bounding Box) som kännetecknas av att planens normaler överensstämmer med axlarna i aktuellt koordinatsystem (se avsnitt 1.3 AABB).

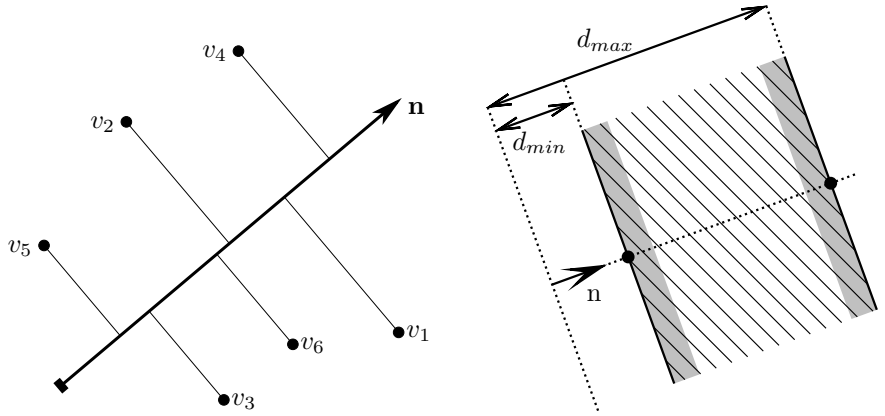
Omslutande volymer som $k$-DOP och AABB tas fram genom att beräkna min- och maxprojiceringen av modellens punkter på ett normalset (se figur 2(a)). Projiceringen beräknas på samma vis som skalärprodukten:

$$a \cdot b = \|a\|\|b\| \cos\theta$$

vilket då $b$ är av enhetslängd kommer motsvara längden på utbredningen av $a$ i riktningen av $b$.

## 1.3 AABB

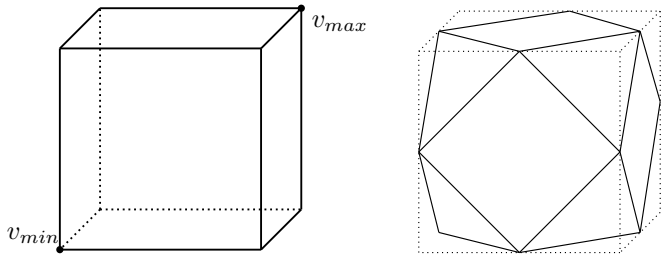
Den axelorienterade boxen (Axis Aligned Bounding Box) är en av de populäraste volymerna på grund av sin enkelhet och snabbhet både vid generering och skärningstest. Den kännetecknas av att ytnormalerna är orienterade efter axlarna i aktuellt koordinatsystem. En AABB kan lagras som två punkter utgörande motstående hörnpunkter på boxen, vilket ger ett lågt minnesbehov (6 flyttal).




(a) Projicering av $v_i \in V$ på $\mathbf{n}$ (b) Slab definierad av $\mathbf{n}$, $d_{min}$ och $d_{max}$.

**Figur 2:** Bild (a) visar projektion av punkter på en normal samt (b) en slab definierad av en normal samt två avstånd från origo.

Nackdelen med AABB är dess relativt dåliga passform vilket ger upphov till falska överlappningar. Utbredningen för en AABB beräknas genom projicering av alla punkter längs normalerna som i detta fall sammanfaller med koordinataxlarna, $x$, $y$ och $z$. Detta gör beräkning av boxen rättfram och väldigt enkel då de projicerade extrempunkterna för respektive axel $x$, $y$, $z$ kommer att utgöra hörnpunkterna i boxen $(v_{min}, v_{max})$, se figur 3(a).




(a) AABB (b) 14-DOP

**Figur 3:** Exempel på en AABB och en 14-DOP.

## 1.4 $k$-DOP

Som nämnts tidigare är en $k$-DOP (Discrete Orientation Polytope) sammansatt av ett antal parvis parallella plan (slab), och bildar på så sätt en polyeder där $k$ anger antalet sidor (figur 3(b)). Antalet normaler som behövs för att producera en $k$-Dop är därmed $k/2$. Typiskt används $k \in \{6, 8, 14, 18, 26\}$ för att producera omslutande volymer. Om normalerna för en 6-DOP väljs så att de sammanfaller med koordinatsystemets axlar kommer det producerade resultatet att motsvara en AABB.

Genom att ha gemensamma normaler för alla volymer i en scen kan minnesanvändningen effektiviseras. Endast min- och maxavståndet till en fast punkt lagras per slab (2 flyttal/normal). Dessa volymer har fördelarna av att vara relativt snabba både vid generering och vid överlappningstest. Dessutom är de relativt täta, speciellt då ett stort antal normaler används vid framtagningen. Till nackdelarna hör att volymen måste beräknas på nytt om den inneslutna modellen roteras eller skalas.

## 1.5 Beräkning av $k$-DOP

Principen för $k$-DOP-algoritmen är att finna den största utbredningen från en fast punkt längs varje normal. Figur 4 visar en generell metod som med projicering (se avsnitt 1.2) tar fram min- och maxavstånd till varje slab ($S$ och $L$). Efter en initiering (rad 2-3) projiceras varje punkt $v \in V$ mot varje normal $\mathbf{n} \in N$ (rad 5-6), och aktuella min- och maxvärden uppdateras (rad 7-10).

$k$-DOP
**input**: $V = \{v_1, v_2, \ldots, v_m\}$, $N = \{\mathbf{n}_1, \mathbf{n}_2, \ldots, \mathbf{n}_{k/2}\}$,
**output**: $D = \{\{s_1, s_2, \ldots, s_{k/2}\}, \{l_1, l_2, \ldots, l_{k/2}\}\}$
1. **for each** $\mathbf{n}_i \in N$
2. $\quad s_i \leftarrow \textsc{Projection}(v_1, \mathbf{n}_1)$
3. $\quad l_i \leftarrow \textsc{Projection}(v_1, \mathbf{n}_1)$
4. **for each** $v_i \in V$
5. $\quad$ **for each** $\mathbf{n}_j \in N$
6. $\quad\quad p \leftarrow \textsc{Projection}(v_i, \mathbf{n}_j)$
7. $\quad\quad$ **if** $p < s_j$ **then**
8. $\quad\quad\quad s_j \leftarrow p$
9. $\quad\quad$ **if** $p > l_j$ **then**
10. $\quad\quad\quad l_j \leftarrow p$
11. **for each** $\mathbf{n}_i \in N$
12. $\quad s_i \leftarrow s_i / \|\mathbf{n}_i\|$
13. $\quad l_i \leftarrow l_i / \|\mathbf{n}_i\|$

**Figur 4:** Principen för beräkning av $k$-DOP.

## 1.6 Normaler

Valet av normaler är kritiskt eftersom noggrannt utvalda normaler markant kan minska antalet beräkningar som krävs. Genom att välja normalerna inom set av {-1, 0, 1} kan enkelt många operationer undvikas. Utifrån enhetskuben kan ett antal lämpliga normaler väljas som även är relativt jämt distribuerade i olika riktningar. De tre ytnormalerna definierar en 6-DOP eller en AABB. Ytterliggare fyra 'hörnnormaler' ger en 14-DOP. Utökas mängden normaler ytterligare med de sex 'kantnormalerna' ges en 26-DOP (se tabell 1) I detta fall kommer

en volym genererad av ett högre antal normaler att ge en minst lika bra eller bättre volym.

| | **Normaler** |
|---|---|
| Ytnormaler | (1, 0, 0), (0, 1, 0), (0, 0, 1) |
| Hörnnormaler | (1, 1, 1), (1, 1,-1), (1,-1, 1), (-1,-1, 1) |
| Kantnormaler | (1, 1, 0), (1 ,0 ,1), (1,-1, 0), (1, 0,-1), (0, 1, 1), (0, 1,-1) |

**Tabell 1:** Normalerna för AABB (6-DOP), 14-DOP och 26-DOP

När detta tillämpas i den generella metoden i figur 4 kan den inre loopen (rad 5-10) rullas upp och varje projektionsberäkning direkt styras av normalens beståndsdelar. Exempelvis kommer projiceringen mot normalen [1, 0, -1] att ges av $P = X_i - Z_i$. Av denna anledning så görs heller ingen normalisering av normalerna, istället korrigeras de framräknade avstånden med respektive normals längd (magnitud) innan algoritmen avslutas (rad 11-13).

## 1.7 Sfär

Sfären är tillsammans med AABB troligen den vanligast använda volymen. De har båda enkla och beräkningsbilliga överlappningstest. Sfären är dessutom oberoende av rotation, vilket gör att den aldrig behöver roteras utan endast flyttas till ny position. Lagringsbehovet är lågt för en sfär då den kan beskrivas av en centrumpunkt samt en radie (fyra flyttal).

## 1.8 Beräkning av sfär

Varje volym såsom AABB, $k$-DOP samt sfär, har alla en optimal volym för varje modell, det vill säga en volym som är den minsta möjliga. Det som skiljer en sfär från en $k$-DOP i detta avseende är att en $k$-DOP har ett begränsat sökområde för att finna extrempunkterna (de fördefinierade normalerna), medan en sfär kan ha de begränsande punkterna i godtycklig normal. På grund av denna beräkningsbarriär finns två olika inriktningar på algoritmer för sfärberäkning. De exakta samt de approximativa.

Metoderna för att beräkna exakta sfärer är ofta för ineffektiva för att vara aktuella i realtidsapplikationer, men kan användas till förberäknade sfärer. Alla sfärer begränsas av 2, 3 eller 4 punkter, så kallade stödpunkter, som definieras av att de ligger på sfärens yta. Det kan vara fler än fyra punkter som befinner sig på sfärens yta, men fyra är tillräckligt för att beräkna sfären. Genom att finna dessa punkter kan den optimala sfären beräknas. En möjlig metod är test av alla möjliga kombinationer (brute force) vilket har en tidkomplexitet på $O(n^5)$ och därmed olämplig i de flesta fall. En bättre metod är Gärtners algoritm [Gae99] som har en förväntad linjär komplexitet.

De approximativa metoderna är på grund av sin relativa snabbhet populära i realtidsapplikationer. Många metoder är mycket snabba men genererar samtidigt sfärer med ganska lös passning. En metod utgår från en framräknad AABB, använder dess centrum som centrumpunkt för sfären samt avståndet från centrum

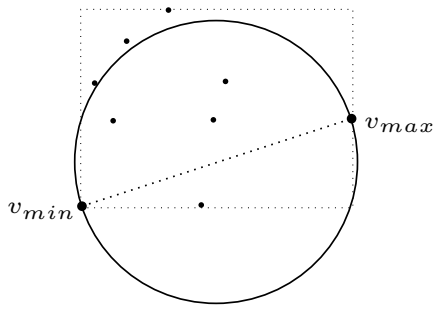
till punkten längst bort som sfärens radie. Denna metod kan väldigt snabbt skapa en omslutande sfär men resultatet är ofta av dålig kvalitet. Andra föreslagna approximativa metoder såsom Ritter och EPOS redovisas nedan.

## 1.9 Ritters algoritm

En populär approximativ metod för sfärberäkning är Ritters algoritm [Rit90], eller varianter av denna. Metoden är enkel, snabb och producerar acceptabla sfärer för de flesta tillämpningar. Ritter består av två pass där det första passet finner tre par extrempunkter, i en given mängd av punkter $V$, utmed koordinatsystemets respektive axlar (motsvarande en AABB). Av dessa tre par väljs paret som har det största euklidiska avståndet $D = \mid v_{max} - v_{min} \mid$. En approximativ sfär beräknas med $D$ som diameter och mittpunkt $c = (v_{max} - v_{min})/2$.

Andra passet går igenom punkterna $v_i \in V$ igen, och när en punkt påträffas som ligger utanför sfären, flyttas samt expanderas den för att inkludera även denna punkt. Figur 5 visar principen för framtagning av en sfär med Ritters algoritm där tre punkter initialt hamnar utanför sfären som därmed måste expanderas.

Endast en liten del av punkterna i $V$ kommer att leda till en uppdatering av sfären, dels för att varje expansion med stor sannolikhet kommer att innesluta även andra punkter utanför, samt att den initiala sfären är en god approximation.




**Figur 5:** Exempel i 2D på framtagning av en initial sfär med Ritter.

## 1.10 EPOS algoritm

Genom att utgå från en bättre approximation av den initiala sfären kan Ritters algoritm förbättras till att producera en sfär med tätare passning. En hybridalgoritm som använder sig av både en approximativ och en exakt sfärlösare är EPOS (Extremal Projection Optimal Sphere) [Lar08], som gör en ansats att snabbt hitta punkterna (eller nära aproximationer) som spänner upp den optimala sfären (figur 6). EPOS söker $k$ extrempunkter (rad 2) utefter $k/2$ normaler som sedan används i en optimal sfärlösare (rad 3), till exempel Gärtners algoritm, för att beräkna den optimala sfären för extrempunkterna $E$. Genom att

kraftigt minska antalet punkter som den optimala sfären beräknas utifrån blir användandet av en exakt sfäralgoritm realistisk i realtidstillämpningar, samtidigt som en bättre första approximation av sfären ges. Slutligen kontrolleras att alla punkter innesluts av sfären, som annars korrigeras för att innesluta även dessa (rad 4). Ett specialfall inträffar då antalet punkter i punktmängden understiger antalet sökta extrempunkter (rad 1), då dessa direkt kan lösas av den exakta sfäralgoritmen (rad 6).

ExtremalPointOptimalSphere
**input**: $V = \{v_1, v_2, \ldots, v_m\}$, $N = \{\mathbf{n}_1, \mathbf{n}_2, \ldots, \mathbf{n}_{k/2}\}$
**output**: $S = \{c, r\}$
1. **if** $n > k$ **then**
2. $E \leftarrow$ FindExtremalPoints$(V, N)$
3. $S' \leftarrow$ MinimumSphere$(E)$
4. $S \leftarrow$ GrowSphere$(V, S')$
5. **else**
6. $S \leftarrow$ MinimumSphere$(V)$

**Figur 6:** Pseudokod för EPOS-algoritmen

Detta innebär att om punkterna som begränsar sfären hittas i den första sökningen av extrempunkter så kommer den optimala sfären hittas. Sannolikheten för detta kommer att öka med ökande värde på $k$.

Antalet normaler samt val av normaler är helt fritt i implementeringen av EPOS men med fördel väljs normaler enligt samma princip som redovisats i avsnittet Normaler (se avsnitt 1.6).

Fördelarna med EPOS gentemot exempelvis Ritter är att de producerade sfärerna i de flesta fallen är betydligt tätare. Däremot så är den långsammare. EPOS-6 är ungefär 10% långsammare, 14-DOP och 26-DOP är ungefär 2 respektive 3,5 gånger långsammare än Ritter [Lar08].

# 2 SIMD

## 2.1 Inledning

Mängden data som datorer idag förutsätts hantera ökar kontinuerligt vilket ställer högre krav på processorernas hastighet. Dock så har den fysiska gränsen för hur snabb en processor kan bli enbart genom att öka dess klockhastighet i det närmaste redan nåtts. Läckströmmar, värmeutveckling och bara det faktum att de elektriska signalerna inte hinner transporteras över processorchipet i en klockcykel gör att det inte är lämpligt att öka klockhastigheterna mer. Därför krävs andra lösningar. En lösning är att behandla data parallellt, så en instruktion utförs på flera dataelement samtidigt. Detta refereras ofta till som dataparallellism (jämför mot funktionell parallellism). Instruktionsuppsättningar som

medger detta kallas SIMD (Single Instruction Multiple Data).

Det är inte all data som i sin natur passar att hanteras parallellt, men där det är möjligt kan stor prestandavinst ske. Data för lagring av grafik eller ljud är ofta lämplig för parallell hantering [HOM08]. Man talar därför ofta om SIMD som Multimedia Instruction Set Architecture Extensions eller Multimedia ISA Extension.

## 2.2 Historik

SIMD har sina rötter i vektorprocessorer (arrayprocessorer) från tidigt 60-tal. Vektorprocessorer designades för att processa multipla dataelement per instruktion. Filosofin stod i kontrast till skalära processorer som normalt hanterar ett dataelement åt gången. Vektorprocessorer kom att bli vanliga i superdatorer mellan 1980 och 1990, sedan dess har utvecklingen gått mot superdatorer bestående av multipla skalära processorer med sitt egna minne och specifika uppgift. Idag innehåller de flesta skalära processorer även instruktioner för hantering av parallell data, känt under samlingsnamnet SIMD. Man talar också om SIMD som vektorinstruktioner då man avser instruktionsuppsättningar avsedda att processa data parallellt.

Förutom att skalära processorer har viss instruktionsuppsättning för dataparallellitet lever vektorprocessorparadigment kvar i dagens grafikkort som är designade för att hantera stora vektoriserade dataset. Cellprocessorn utvecklad år 2000 av IBM, Toshiba och Sony består av ett processorship med en skalär CPU och åtta vektorprocessorer. Den första kommersiella applikationen för Cellprocessorn var Sony Playstation 3.

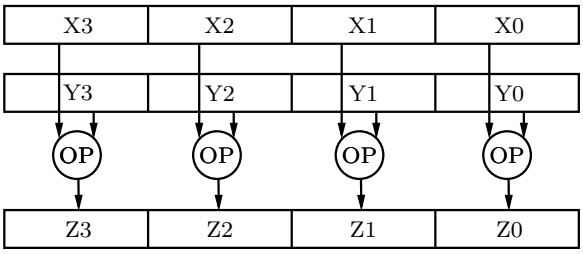
Intel introducerade SIMD i IA-32 arkitekturen 1996 och det kallades då MMX (MultiMedia eXtension). Olika tillverkare har olika Multimedia ISA extensions. I rapporten kommer Intels SIMD instruktionsuppsättning SSE att användas som också stöds av AMD.

Intels MMX följdes senare av Streaming SIMD Extension (SSE). Namnet kommer av att SIMD-instruktionerna följer ett paradigm kallat *stream processing*, vilket innebär att en ström av data behandlas.

## 2.3 Exekveringsmodell

När en SIMD-instruktion exekveras utförs samma operationssekvens parallellt på ett större antal diskreta dataset. Detta illustreras i figur 7.

Två dataset med fyra element i varje set(X0-X3 och Y0-Y3) processas samtidigt av en och samma operation OP. Resultatet placeras i fyra nya dataelement (Z0 till Z3). Registren i bilden har 128 bitars bredd vilket betyder att fyra 32 bitars tal kan hanteras parallellt. Då data ligger i sekvens i registret talar man om 128 bitar packad data (128-bit packed), eller linjerat data (aligned data), se avsnitt 2.7.




**Figur 7:** Operationen OP utförs parallellt på två SIMD-register.

## 2.4 Streaming SIMD Extension

SSE introducerades av Intel med processorfamiljen Pentium III och var en utökning av MMX. SSE hanterar 128 bitars data genom åtta register (XMM0-XMM7). SSE har fortsatt att utvecklas och finns idag i version 4.2. Samtliga iterationer av SSE har behållit bakåtkompabilitet med tidigare versioner. I en miljö där MMX samt SSE/SSE2 används får programmeraren möjlighet att utveckla algoritmer som specifikt använder datatyper och register från samtliga tre tekniker för att specialanpassa algoritmer till specifika uppgifter. SSE medger hantering av fyra 32 bitars flyttal med enkel precision eller två 64 bitars flyttal med dubbel precision. Registren som ingår i Intels SSE-modell är:

1. Åtta 128-bitars XMM-register för hantering av hel- och flyttalsdata.
2. Ett 32-bitars MXCSR-register för kontroll och statusinformation för flyttalsoperationer.
3. Åtta MMX 64-bitars MMX-register för hantering av packad heltalsdata.
4. Åtta generella register för adress- och operandhantering.
5. Ett 32-bitars EFLAGS-register för resultatet av jämförelseoperationer.

## 2.5 Övriga SSE versioner

SSE3 introducerades i och med Pentum IV och har stöd för acceleration av trådsynkning samt instruktioner för horisontella registeroperationer. SSSE3 (Supplemental SSE3) innebar en utökning av instruktionsuppsättningen med instruktioner för bland annat horisontella operationer över registren. SSE4.1 respektive SSE4.2 innehåller ytterligare instruktioner bland annat för att förbättra kompilatorvektorisering samt möjlighet för sträng- och texthanteringsalgoritmer att dra nytta av SIMD.

## 2.6 Implementering

Historiskt var programmeraren tvungen att skriva assemblerkod för att kunna dra nytta av SIMD. Idag existerar flera olika paradigm. Vid sidan av assembler

finns intrinsic-funktioner, C++ biblioteksfunktioner samt automatisk vektorisering. Metoderna har olika tillämpningsområde då de i olika grad underlättar för programmeraren men ger avkall på prestanda. Generellt gäller att med assembler finns potential att få ut mest prestanda.

### 2.6.1 Automatisk vektorisering

Automatisk vektorisering innebär att kompilatorn analyserar koden och försöker använda SIMD-instruktioner där det är möjligt. Idag har både Intels C++ kompilator och Microsofts Visual C++ kompilator denna möjlighet.

### 2.6.2 C++ Klassbibliotek

Intels kompilator levereras med ett klassbibliotek för att underlätta SIMD-hantering. Detta ger något bättre kontroll än automatisk vektorisering, men vad som framförallt medges är en objektorienterad abstraktion av SSE data.

### 2.6.3 Intrinsic

Intrinsic är en samling C funktioner som mer eller mindre direktmappar till assemblerinstruktioner. Programmeraren slipper dock att göra registerallokeringar, skedulering av instruktioner eller bry sig om olika adresseringsmetoder. Tack vare detta blir intrinsic lättare att använda men man har inte exakt kontroll över de genererade instruktionerna.

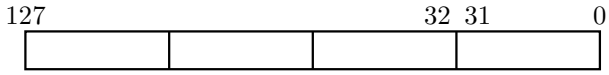
### 2.6.4 Inline Assembler

För exakt kontroll måste assemblerprogrammering nyttjas. Genom att använda assembler kan störst prestandavinst göras men det kräver mer av programmeraren. Kompilatorn är i många fall bättre på att optimera assemblerkod än en programmerare. Det gäller hela tiden att använda den teknik som ger mest prestanda för en specifik uppgift. En duktig assemblerprogrammerare skriver dock med största sannolikhet SIMD-instruktioner som är bättre eller lika bra som kompilatorn.

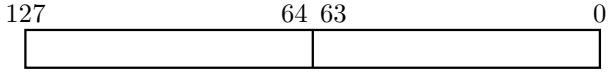
## 2.7 Datalinjering (Data alignment)

De register som används av SIMD-instruktionerna kräver att data är packat eller linjerat i grupper om 16 byte, det vill säga att data finns i strikt sekvens i minnet och därmed kan hanteras som ett stycke. De flesta SSE-instruktioner ger upphov till ett undantagsfel vid användning av icke linjerat data. Figur 8 visar allokering av flyttal med enkel respektive dubbel precision i 128 bitars register.

I Visual Studio används konstruktionen `_declspec(align(#))` före en variabeldeklaration för att få linjerat data och `_aligned_malloc(size, alignment)` för att dynamiskt allokera linjerat data. Ex:




(a) Fyra flyttal med enkel precision




(b) Två flyttal med dubbel precision

**Figur 8:** Linjering av flyttalsdata i SSE-register.

```
// Allokerar fyra 32 bitars heltalsdata i följd.
__declspec(align(32)) int a[4];

// Allokerar utrymme för 100 st heltal linjerat på 32 byte.
int* ptr = (int*)_aligned_malloc(100 * sizeof(int), 32);
```

## 2.8 Datastrukturer

För att utnyttja hela SIMD-registrens bredd måste datarepresentationen för 3D-modellerna väljas på ett klokt sätt. En vanlig organisation för att beskriva en punkt i 3D-rymden medges genom en struktur med de enskilda komponenterna lagrade som flyttal enligt nedan. En modell kan då lagras som en array av dessa strukturer (AoS eller Array of Structures). För att lagra en modell krävs lika många strukturer som i modellen ingående punkter (NbrPts).

```
typedef struct
{
    float x;
    float y;
    float z;
} Point;
Point m[NbrPts];
```

Denna representation är intuitiv och ibland lämplig, men registerbredden utnyttjas inte optimalt. Eftersom SSE hanterar 16 byte linjerat data (4 flyttal) utökas strukturen med en w-komponent som inte fyller någon funktion enligt nedan:

```
typedef struct __declspec(align(16))
{
    float x;
    float y;
    float z;
    float w;
} Point;
```

Allokerat data ligger då i ett XMM-registren enligt figur 9. Då w-komponenten inte används tar den upp onödig plats och varje SSE-operation kommer utföra

nyttigt arbete på tre flyttal parallellt mot fyra om hela registerbredden utnyttjades.

| | | | |
|---|---|---|---|
| $x_1\ y_1\ z_1\ w_1$ | $x_2\ y_2\ z_2\ w_2$ | $\ldots.$ | $x_n\ y_n\ z_n\ w_n$ |

**Figur 9:** Allokerad data i XMM-register.

För att utnyttja hela registerbredden kan en annan typ av datastruktur användas där fyra punkter lagras i varje struktur. Denna organisation kallas struktur av arrayer (SoA) och medger ett bättre utnyttjande av XMM-registren.

```
typedef struct __declspec(align(16))
{
    float fx[4];
    float fy[4];
    float fz[4];
} Group;
Group m[NbrPts/4];
```

Då varje struktur lagrar fyra punkter blir antalet strukturer $NbrPts/4$ för lagring. En konceptuell bild över minneslayouten ser ut som figur 10. $G$ motsvarar en struktur av typen *Group* som lagrar 4 punkter $v_0 - v_3$ i tre XMM-register $R_0 - R_2$. Registren $R$ benämns 4-tuple.

| | $G_0$ | | | $G_m$ |
|---|---|---|---|---|
| $R_0$ | $x_0\ x_1\ x_2\ x_3$ | $\ldots$ | $\ldots$ | $\ldots x_n$ |
| $R_1$ | $y_0\ y_1\ y_2\ y_3$ | $\ldots$ | $\ldots$ | $\ldots y_n$ |
| $R_2$ | $z_0\ z_1\ z_2\ z_3$ | $\ldots$ | $\ldots$ | $\ldots z_n$ |
| | $v_0\ v_1\ v_2\ v_3$ | | | |

**Figur 10:** SoA-datastruktur för lagring av punkter.

En nackdel med data i formen av SoA är att man måste frångå den normala hanteringen av punkterna (AoS). Metoder som *dataswizzling* kan vara intressanta och används vanligen för att ändra raderna i en matris till kolumner, samt *deswizzling* som gör motsatsen [Int09]. Eftersom *swizzling* kräver extra operationer är det bättre om data kan hanteras i en SoA-struktur även för övriga operationer.

## 2.9 Eliminering av jämförelsesatser

En potentiellt försvårande omständighet när algoritmer skall parallelliseras med SIMD är jämförelsesatser. Instruktioner för min- och maxoperationer introducerades i SSE2, men databeroende kan göra att enskilda element i registren måste utvärderas. Att sekventiellt jämföra varje element i två XMM-register med varandra skulle förstöra dataparallelliteten så andra lösningar behövs. En metod är att arbeta med bitvisa maskar [Int99, GBST06]. Ett antal parallella

jämförelseinstruktioner finns i SSE så som cmpltps (compare less than), cmpgtps (compare greater than) som båda returnerar en 128-bitarsmask. Dessa maskar kan sedan användas i logiska operationer så som OCH- och ELLER-satser. I avsnitt 3.2 används vid beräkning av omslutande sfärer just bitmaskar för att eliminera jämförelsesatser (branch elemination).

# 3 Parallellisering av volymberäkningar

Beräkningsintensiva kodfragment som exekveras tillräckligt ofta och som har ett litet databeroende har potential att dra nytta av SIMD [HOM08]. De nämnda volymberäkningarna för AABB, $k$-DOP och sfär har flera av dessa egenskaper. De inbegriper ofta iterationer över stora punktmängder, som i vissa fall itereras fler än en gång (sfärer). Även om beräkningsintensiteten relativt sett inte är hög sker trots allt ett antal jämförelser för varje punkt. För $k$-DOP med $k > 6$ sker dessutom ett antal multiplikationer, additioner och subtraktioner. Databeroende som kan motverka parallelliteten för AABB och $k$-DOP är obefintligt. En viss grad av beroende uppstår vid sfärberäkning vilket visas senare och hur det problemet kan lösas.

## 3.1 AABB och $k$-DOP

Nedan visas en parallelliserad variant av $k$-DOP-beräkningen (figur 11). Beräkningen av AABB generaliseras till en $k$-DOP med $k = 6$. Grundprincipen består i att iterationen sker över ett dataset $G$ där fyra punkter hanteras i varje loop (rad 5-9). Detta betyder att samma mängd instruktioner kommer processa fyra gånger så mycket data som en sekventiell algoritm. Genom detta ökas inte hastigheten på den enskilda beräkningen utan istället på datagenomflödet (throughput).

Indata är ett punktset $G$ fördelat i grupper om tre 4-tupler samt ett normalset $N$ med $k/2$ normaler som punkterna skall projiceras på. För varje punktgrupp $G_i$ beräknas projiceringen $P$ på varje normal $\mathbf{n}_j$ (rad 7). Genom SSE-instruktionerna minps och maxps ges största och minsta projiceringarna $S_j$, $L_j$ (rad 8-9). Från de resulterande 4-tuplerna $S$ och $L$ hämtas högsta och lägsta värdet ut till $s_j$ och $l_j$ (rad 11-12). Eftersom normalerna i normalsetet $N$ inte behöver vara normerade kompenseras $s_j$ och $l_j$ för detta (rad 13-14). För att snabba upp beräkningen rullas den inre loopen (rad 6-9) upp och specifika varianter av 6-DOP, 14-DOP och 26-DOP beräkningarna implementeras för att dra nytta av de förenklade skalärproduktberäkningarna.

## 3.2 Parallellisering av sfärberäkningar

Då både EPOS och Ritter använder sig av extremvärden för att approximera en sfär i sina initiala faser, kan de dataparallella metoderna för $k$-DOP användas och ge prestandavinster även i sfärberäkningen. Dock så introduceras ett databeroende. Beräkningen av $k$-DOP ger endast de extremvärden som spänner upp

$k$-DOP-SIMD
**input**: $G = \{G_1, G_2, \ldots, G_m\}$, $N = \{\mathbf{n}_1, \mathbf{n}_2, \ldots, \mathbf{n}_{k/2}\}$
**output**: $D = \{\{s_1, s_2, \ldots, s_{k/2}\}, \{l_1, l_2, \ldots, l_{k/2}\}\}$

1. **for each** $\mathbf{n}_j \in N$
2. $\quad P \leftarrow \textsc{Projection}(G_1, \mathbf{n}_j)$
3. $\quad S_j \leftarrow P$
4. $\quad L_j \leftarrow P$
5. **for each** $G_i \in G$
6. $\quad$ **for each** $\mathbf{n}_j \in N$
7. $\quad\quad P \leftarrow \textsc{Projection}(G_i, \mathbf{n}_j)$
8. $\quad\quad S_j \leftarrow \text{minps}(S_j, P)$
9. $\quad\quad L_j \leftarrow \text{maxps}(L_j, P)$
10. **for** $j = 0$ **to** $k/2$
11. $\quad s_j \leftarrow \min(S_j)$
12. $\quad l_j \leftarrow \max(L_j)$
13. $\quad s_j \leftarrow s_j / \|\mathbf{n}_j\|$
14. $\quad l_j \leftarrow l_j / \|\mathbf{n}_j\|$

**Figur 11:** Dataparallel beräkning av $k$-DOP.

volymen. För en effektiv sfärberäkning behövs de faktiska punkter som motsvarar extremvärdena. Således behöver $k$-DOP-algoritmen modifieras så att punkterna som spänner upp volymen blir kända. Genom att modifiera algoritmen för beräkning av $k$-DOP så att extrempunkternas index blir kända kan punkterna senare användas i sfärberäkningen. Problematiken ligger i att SSE-instruktioner så som minps och maxps inte ger hänvisning till vad som uppdaterats i registret. Därmed kan inte tillhörande indexregister uppdateras utan att först ta reda på vad som förändrats. Genom att byta ut minps- och maxps-instruktionerna (rad 8-9) i figur 11 och istället använda jämförande operationer som resulterar i bitmaskar ges information om vad som skall uppdateras. Figur 12 illustrerar proceduren då det minsta värdet och tillhörande index skall hämtas. Proceduren ersätter således rad 8 i $k$-DOP-SIMD, figur 11.

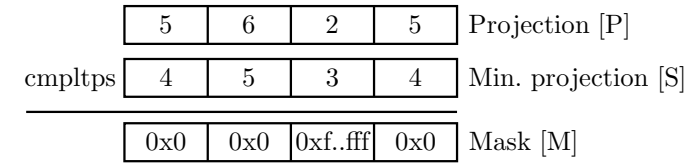
Variablerna i registren i figur 12 innehåller:
$P$ Aktuell projektion (motsvarande rad 7 i $k$-DOP-SIMD figur 11).
$C$ Index för de aktuella projektionerna ($P$). Dessa stegas upp med 4 för varje loop.
$S$ Aktuella minsta projektioner (motsvarande rad 8 i $k$-DOP-SIMD figur 11).
$A$ Index som motsvarar de aktuella minsta projektionerna ($S$).

Följande sekvens av bitvisa operationer utförs för att hämta minvärde och dess index (se figur 12):

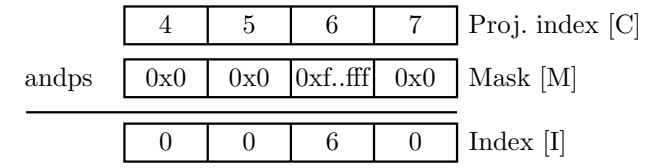
a. Jämförelseinstruktionen cmpltps (compare less than) producerar en mask $M$ för minsta projektionerna i $P$, där 0xffffff motsvarar ett sant värde.

b. Masken $M$ används med en logisk OCH-instruktion, andps, för att extrahera ut motsvarande projektionsindex $C$ till Index $I$.

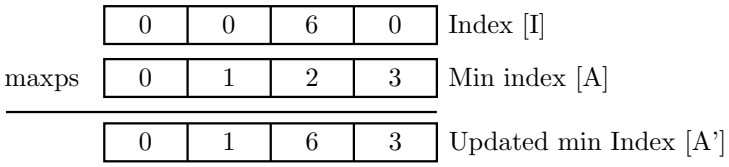
c. Min-index $A$ uppdateras med nya index från $I$ med en maxps-operation.



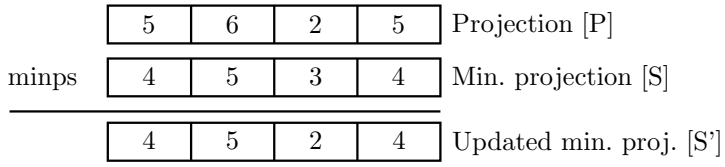

(a) SSE-instruktion cmpltps.




(b) SSE-instruktion andps.




(c) SSE-instruktion maxps.




(d) SSE-instruktion minps.

**Figur 12:** Bitvisa operationer för framtagning av minvärde och motsvarande index

Detta eftersom ett uppdaterat indexvärde alltid kommer att vara högre.

d. Till sist görs en minps-operation för att uppdatera de verkliga projektionsvärdena.

Genom detta förfarande hanteras fortfarande 4 punkter parallellt till priset av några fler instruktioner. Behovet av jämförelsesatser på enskilda element har helt eliminerats (branch elemination), flera varianter av eliminering av jämförelsesatser finns beskrivet i [GBST06].

Den kompletta pseudokoden för $k$-DOP med indexuthämtning $k$-DOP-INDEX-SIMD visas i figur 13. Indata $G$ är punktmängden som itereras och vars projicering på normalmängden $N$ beräknas. Utdata $D$ är extrempunkterna i $k$-DOP-volymen och $Z$ är index för extrempunkterna. Rad 8-12 svarar mot beräkning av minvärde och rad 13-17 mot maxvärde. På rad 9 respektive 14 görs movmsk-instruktioner för att avgöra om nya min- eller maxvärden hittats, anledningen till att denna jämförelsesats behållits är att det visat sig vara snabbare på de testade modellerna. Movmsk resulterar i ett 4-bitarsvärde innehållande de 4

mest signifikanta bitarna i $M$. Ett resultat större än 0 innebär att $M$ har minst ett element som är sant. Från de resulterade 4-tuplerregistren för minvärde och index ($S$, $A$) samt maxvärde och index ($L$, $B$) hämtas absoluta min och max sekventiellt ut till $D$ (värde) och $Z$ (index), rad 18-20. Sist normeras min- och maxprojektionerna (rad 21-22).

$k$-DOP-INDEX-SIMD
**input**: $G = \{G_1, G_2, \ldots, G_m\}$, $N = \{\mathbf{n}_1, \mathbf{n}_2, \ldots, \mathbf{n}_{k/2}\}$
**output**: $D = \{\{s_1, s_2, \ldots, s_{k/2}\}, \{l_1, l_2, \ldots, l_{k/2}\}\}$,
$Z = \{\{a_1, a_2, \ldots, a_{k/2}\}, \{b_1, b_2, \ldots, b_{k/2}\}\}$

1. **for each** $\mathbf{n}_j \in N$
2. $\quad P \leftarrow \text{PROJECTION}(G_1, \mathbf{n}_j)$
3. $\quad S_j \leftarrow P$
4. $\quad L_j \leftarrow P$
5. **for each** $G_i \in G$
6. $\quad$ **for each** $\mathbf{n}_j \in N$
7. $\quad\quad P \leftarrow \text{PROJECTION}(G_i, \mathbf{n}_j)$
8. $\quad\quad M \leftarrow \text{cmpltps}(P, S_j)$
9. $\quad\quad$ **if** $\text{movmsk}(M) > 0$ **then**
10. $\quad\quad\quad I \leftarrow \text{andps}(C_i, M)$
11. $\quad\quad\quad A_j \leftarrow \text{maxps}(I, A_j)$
12. $\quad\quad\quad S_j \leftarrow \text{minps}(S_j, P)$
13. $\quad\quad M \leftarrow \text{cmpgtps}(P, S_j)$
14. $\quad\quad$ **if** $\text{movmsk}(M) > 0$ **then**
15. $\quad\quad\quad I \leftarrow \text{andps}(C_i, M)$
16. $\quad\quad\quad B_j \leftarrow \text{maxps}(I, B_j)$
17. $\quad\quad\quad L_j \leftarrow \text{maxps}(L_j, P)$
18. **for** $j = 0$ **to** $k/2$
19. $\quad s_j, a_j \leftarrow \min(S_j)$
20. $\quad l_j, b_j \leftarrow \max(L_j)$
21. $\quad s_j \leftarrow s_j / \|\mathbf{n}_j\|$
22. $\quad l_j \leftarrow l_j / \|\mathbf{n}_j\|$

**Figur 13:** Dataparallel beräkning av $k$-DOP med framtagning av index.

Vid implementering av algoritmen rullas loopen över normalsetet $N$ upp på samma sätt som beskrivs i avsnitt 1.6 normaler. För att beräkna minvärde och dess index rad 7-12 exekveras 6 instruktioner (samt en jämförelsesats) i de fall nya värden hittas och 3 instruktioner (samt en jämförelse) i alla andra fall, jämfört med $k$-DOP-SIMD-algoritmen som alltid exekverar 2 instruktioner. Denna komplexitetsökning gör att $k$-DOP-INDEX-SIMD får en något mindre prestandavinst.

## 3.3 Dataparallell EPOS och Ritter

Kända indexvärden för extrempunkterna kan utnyttjas för att beräkna en initial omslutande sfär [Lar08, Rit90]. Oavsett vilken metod som används måste det

sedan säkerställas att samtliga punkter i modellen verkligen ligger innanför den beräknade sfären. Då detta inbegriper en iteration över samtliga punkter finns även här dataparallellitet att utvinna. Liknande problem, som vid indexuthämtning, uppstår då fyra punkter verifieras parallellt. Den föreslagna algoritmen CheckSphere visas i figur 14.

CheckSphere
**input**: $G = \{G_1, G_2, \ldots, G_m\}, \mathbf{c}, r$
**output**: $\mathbf{c}, r$
1. **for each** $G_i \in G$
2. $D \leftarrow$ GetDistances$(G_i, \mathbf{c})$
3. $M \leftarrow$ cmpgtps$(D, r^2)$
4. **if** movmsk$(M) > 0$ **then**
5. $\mathbf{c}, r \leftarrow$ UpdateSphere$(M, D, G_i, \mathbf{c}, r)$

**Figur 14:** CheckSphere algoritmen.

Modellens punkter $G$ itereras igenom (rad 1), och fyra punkter valideras parallellt mot sfären som beskrivs av en centrumpunkt $\mathbf{c}$ och en radie $r$ . Avståndet $D$ mellan centrum och en punktgrupp $G_i$ beräknas (rad 2). Operationen cmpgtps returnerar en bitmask för de punkter som ligger utanför (rad 3). Sfären uppdateras sekventiellt för de punkter som ligger utanför (rad 5). Även i detta fall behålls jämförelsesatsen som avgör om sfären behöver uppdateras (rad 4), då uppdatering av sfären sällan sker.

# 4 Multitrådning

Genom att använda multitrådning nås en ytterliggare nivå av parallellism vid sidan av användningen av SIMD. Arbetet delas upp i ett antal delar eller trådar och fördelas ut över antalet tillgängliga processorkärnor.

## 4.1 OpenMP

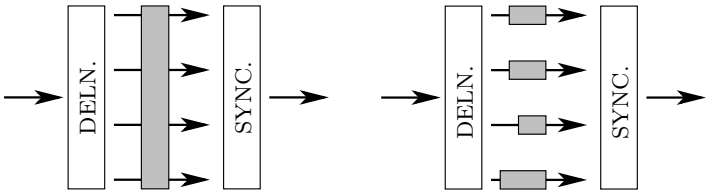
OpenMP är ett API för att förenkla implementationen av multitrådade program, utan att äventyra säkerhet, robusthet eller prestanda [KPT00]. Däremot kvarstår den vanliga problematiken gällande multitrådning, exempelvis synkronisering, därför bör trådparallella delar i applikationer planeras extra noga. Metoden bygger på att avsnitt pekas ut som parallella genom så kallade pragmadirektiv [CJP07]. Direktivet

```
#pragma parallel
```

används för att initiera ett parallellt avsnitt. Genom att använda olika direktiv kan olika typer av trådning enkelt skapas. Direktivet

```
#pragma parallel for
```

innebär att efterföljande for-loop kommer att delas upp mellan antalet trådar och alla trådar itererar varsin del av arbetet i loopen. Denna form av parallellism kan föreställa dataparallellism (se figur 15(a)). Problemet i detta fall är hur trådarna ska hantera delade variabler samt hur synkronisering av resultatet sker.




(a) Parallell FOR (b) Parallell SECTION

**Figur 15:** Parallell FOR efterliknar dataparallellism och parallell SECTION efterliknar funktionell parallellism.

Genom att ange direktivet

```
#pragma parallel sections
```

skapas flera sektioner som körs på separata trådar. Här måste varje sektion ha egen kod samt egna variabler, men problemet med synkroniseringen undviks. Då data och kod är separerat kommer detta att efterlikna funktionell parallellism (se figur 15(b)).

## 4.2 $k$-DOP med data- och trådparallellism

Algoritmen kan implementeras med båda metoderna. En $k$-DOP kan enkelt beräknas parallellt genom att dela upp mängden av punkter i ett antal delar och beräkna en $k$-DOP för varje del, för att sedan sekventiellt sammanställa delvolymerna.

Pseudokoden i figur 16 visar en variant av $k$-DOP-SIMD, implementerad multitrådad med parallella sektioner, där $G$ representerar en mängden av punkter, $N$ normaler och $C$ antalet trådar (vilket inte behöver vara samma som antalet kärnor). Arbetet fördelas i SplitWork (rad 2), genom att mängden punkter delas per tråd. Parallella sektioner initieras (rad 3) och varje sektion körs av separata trådar (rad 4, 7). Varje delresultat beräknas med $k$-DOP-SIMD funktionen (rad 5, 8) som beräknar volymen utifrån given delmängd av punkterna. Slutligen sammanställs resultatet från respektive tråd sekventiellt (rad 10-11).

I den aktuella algoritmen är majoriteten av koden parallell. Varje tråd kan arbeta med en separat mängd av punkter och producera en del av de resulterande extremvärden utan att dela variabler som kräver synkronisering eller kritiska sektioner. Endast då den parallella sektionen avslutas görs en synkronisering, vilket görs implicit av OpenMP.

$k$-DOP-SIMD-Threaded
**input**: $G = \{G_1, G_2, \ldots, G_m\}$, $N = \{\mathbf{n}_1, \mathbf{n}_2, \ldots, \mathbf{n}_{k/2}\}$, $C$
**output**: $D = \{\{s_1, s_2, \ldots, s_{k/2}\}, \{l_1, l_2, \ldots, l_{k/2}\}\}$
1. **for** $i = 1$ **to** $C$
2. $\quad W_i \leftarrow$ SplitWork$(i, m, C)$
3. **parallel sections**
4. $\quad$ **parallel section**
5. $\quad\quad T_0 \leftarrow k$-DOP-SIMD$(W_1)$
6. $\quad \ldots$
7. $\quad$ **parallel section**
8. $\quad\quad T_C \leftarrow k$-DOP-SIMD$(W_C)$
9. $\quad$ **end parallel sections**
10. $D \leftarrow$ FindExtremal$(T_1, T_2, \ldots, T_C)$

**Figur 16:** Pseudokod $k$-DOP-SIMD-Threaded beskriver en multitrådad variant av $k$-DOP-SIMD.

# 5 Resultat

## 5.1 Testmiljö

En testmiljö med möjlighet att jämföra körningar mellan olika implementeringar av metoder har använts. Både numeriska resultat loggas samt den faktiska volymen renderas för respektive modell (se bilaga A). Testmodellerna som använts visas i figur 17, och antalet punkter och trianglar för dessa redovisas i tabell 2. I viss utsträckning har även modeller med slumpmässigt genererade punkter använts.

Tidsmätningar sker via en högupplöst klocka, baserad på processorns inbyggda prestandaräknare. Denna har en teoretiskt noggrannhet på en klockcykel men kan i praktiken variera avsevärt mer beroende på störningar av operativsystem eller andra processer, (out-of-order-execution) eller energisparfunktioner [BH03]. För att kunna fördela eventuella ofrivilliga differenser samt att uppstartskostnader och liknande skulle få mindre inverkan så repeterades algoritmerna ett flertal gånger, varefter första värdet, det högsta värdet och lägsta värdet kastades och sedan beräknades medelvärdet av resterande tider. På så vis ges vad som kan anses vara bästa möjliga fall för metoden, då all data som ryms redan är inläst i cachen [BH03]. Resultat vägs emot en mätning på motsvarande sekventiell metod och presenteras i både absolut och relativ tid (speedup).

Testmiljön är skapad med Microsoft Visual Studio 2008 i release-läge. Algoritmerna är skrivna i C/C++ och alla algoritmer, utom de under avsnittet multitrådning, körs enkeltrådat. Den dator som använts vid redovisade resultat är en PC med en fyrkärnig Intel Core 2 Quad Q8200 CPU, 2.33 GHz, 4GB RAM, Windows 7.

**Figur 17:** De olika polygonmodellerna samt resulterande AABB (kolumn 1), 14-DOP (kolumn 2), 26-DOP (kolumn 3), och sfärer (kolumn 4). Sfärerna är beräknade med EPOS-26.

## 5.2 $k$-DOP

Resultat från körningar redovisas i tabell 2 med absoluta och relativa tider för exekvering av sekventiella och dataparallella (SSE) varianter av AABB (6-DOP), 14-DOP samt 26-DOP. Den relativa tiden, eller uppsnabbningen, ligger på mellan 7-9 för alla modeller, förutom för AABB på den största modellen (se vidare avsnitt 6.1).

## 5.3 Sfär

Tabell 3 visar motsvarande resultat för sfärberäkningar. Här redovisas förutom den sekventiella varianten, även både en sekventiell samt dataparallell variant av Ritter. Uppsnabbning av den dataparallella EPOS är i storleksordningen 3,5-4,5. Detta avsevärt sämre resultat än det för $k$-DOP förklaras av det databeroende som uppstår då index för varje uppdaterad punkt krävs. Sfärberäkningen med Ritter är i orginalutförandet väldigt snabb och vinner inte lika mycket som EPOS på dataparallella metoder.

Tabell 4 innehåller radie på de framräknade sfärerna och visar tydligt att Ritter har sfärer av sämre kvalitet samt att för EPOS ökar kvaliteten med högre antal normaler som används vid framtagning. Sammantaget är en dataparallell metod av EPOS-26 lika snabb som en sekventiell Ritter samtidigt som den ger en sfär av högre kvalitet, samt att EPOS-6 är lika snabb som den dataparallella Rittermetoden. Sfärer producerade av Ritter har typiskt en storlek som är 5-10% större än den optimala, EPOS-6 ca 2-3% större medan EPOS-26 ligger under 0.2%. Då både Ritter och EPOS-6 baserar sina initiala sfärer på en AABB så kommer båda ha problem med samma typ av modeller. Extremfallet infaller på samma modell för båda metoderna och är för Ritter 14.49% respektive EPOS-6 8.94% förstoring. EPOS-26 beräknar samma modell med endast 0.03% förstoring.

En anmärkning på resultaten i tabell 4 är att radien kan variera mellan den sekventiella och dataparallella varianten trots att beräkningen är gjord på samma modell. Detta beror på att projiceringen av två olika punkter kan ge samma skalärprodukt, speciellt då endast flyttal med enkel precision används. Skillnaden ligger sedan i hur de olika metoderna uppdaterar aktuella extrempunkter. Den sekventiella varianten kommer att ange den först funna punkten, alltså den punkt med lägst index, medan den dataparallella varianten gör detsamma för flera punkter åt gången och kan i extremfallet ha fyra identiska värden vid uthämtningsfasen. Den valda metoden för att hämta ut extremvärdet ur den sista 4-tuplen blir avgörande för vilken punkt som blir utvald. Däremot finns det inget som säger vad som är den bästa metoden då det är helt beroende av aktuellt punktset.

## 5.4 Multitrådning

De stora modellerna och metoderna (med flera rader kod) får större prestandavinster vid multitrådning då kostnaderna för trådningen kan fördelas över ett större arbete. Under bra förutsättningar når trådningen en uppsnabbning av

| | Antal | | 6-DOP-SIMD | | | 14-DOP-SIMD | | | 26-DOP-SIMD | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| **Modell** | Punkter | Trianglar | Sek | SSE | $S$ | Sek | SSE | $S$ | Sek | SSE | $S$ |
| Triceratops | 2832 | 5660 | 0.020 | 0.002 | 8.51 | 0.059 | 0.007 | 8.82 | 0.107 | 0.013 | 8.53 |
| Frog | 4010 | 7964 | 0.028 | 0.003 | 9.11 | 0.088 | 0.010 | 9.23 | 0.160 | 0.018 | 8.78 |
| Chair | 7260 | 14372 | 0.048 | 0.006 | 8.68 | 0.142 | 0.017 | 8.25 | 0.257 | 0.032 | 7.97 |
| Tiger | 30892 | 61766 | 0.205 | 0.024 | 8.64 | 0.595 | 0.073 | 8.20 | 1.079 | 0.138 | 7.84 |
| Bunny | 32875 | 65536 | 0.217 | 0.025 | 8.62 | 0.634 | 0.077 | 8.27 | 1.152 | 0.145 | 7.95 |
| Horse | 48485 | 96966 | 0.330 | 0.038 | 8.72 | 0.965 | 0.114 | 8.49 | 1.755 | 0.214 | 8.20 |
| Golfball | 100722 | 201440 | 0.670 | 0.080 | 8.34 | 1.956 | 0.240 | 8.17 | 3.548 | 0.448 | 7.93 |
| Hand | 327323 | 654666 | 2.260 | 0.514 | 4.40 | 6.482 | 0.910 | 7.12 | 11.721 | 1.588 | 7.38 |

**Tabell 2:** Exekveringstider för AABB och $k$-DOP i ms samt uppsnabbning $S$.

| | EPOS-6 | | | EPOS-14 | | | EPOS-26 | | | Ritter | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **Modell** | Sek | SSE | $S$ | Sek | SSE | $S$ | Sek | SSE | $S$ | Sek | SSE | $S$ |
| Triceratops | 0.049 | 0.013 | 3.69 | 0.090 | 0.023 | 3.85 | 0.142 | 0.039 | 3.68 | 0.035 | 0.011 | 3.10 |
| Frog | 0.071 | 0.021 | 3.32 | 0.132 | 0.036 | 3.68 | 0.207 | 0.056 | 3.69 | 0.053 | 0.019 | 2.78 |
| Chair | 0.120 | 0.033 | 3.67 | 0.223 | 0.056 | 4.00 | 0.338 | 0.091 | 3.73 | 0.088 | 0.029 | 3.02 |
| Tiger | 0.491 | 0.114 | 4.31 | 0.889 | 0.190 | 4.67 | 1.384 | 0.299 | 4.63 | 0.365 | 0.111 | 3.30 |
| Bunny | 0.524 | 0.123 | 4.25 | 0.949 | 0.203 | 4.67 | 1.474 | 0.317 | 4.65 | 0.389 | 0.119 | 3.27 |
| Horse | 0.786 | 0.190 | 4.13 | 1.428 | 0.324 | 4.40 | 2.227 | 0.509 | 4.37 | 0.595 | 0.198 | 3.00 |
| Golfball | 1.607 | 0.367 | 4.38 | 2.915 | 0.616 | 4.73 | 4.538 | 0.958 | 4.74 | 1.208 | 0.365 | 3.30 |
| Hand | 5.383 | 1.444 | 3.73 | 9.668 | 2.245 | 4.31 | 15.047 | 3.339 | 4.51 | 4.055 | 1.442 | 2.81 |

**Tabell 3:** Exekveringstider för sfär i ms samt uppsnabbning $S$.

| | | EPOS-6 | | EPOS-14 | | EPOS-26 | | Ritter | |
|---|---|---|---|---|---|---|---|---|---|
| **Modell** | Optimal | Sek | SSE | Sek | SSE | Seq | SSE | Sek | SSE |
| Triceratops | 0.50263 | 0.50343 | 0.50334 | 0.50334 | 0.50334 | 0.50263 | 0.50263 | 0.50343 | 0.50334 |
| Frog | 0.59903 | 0.61349 | 0.61349 | 0.60019 | 0.60019 | 0.59903 | 0.59903 | 0.65965 | 0.65040 |
| Chair | 0.63776 | 0.69474 | 0.68974 | 0.64359 | 0.64359 | 0.63792 | 0.63793 | 0.73014 | 0.72789 |
| Tiger | 0.51397 | 0.52531 | 0.52531 | 0.51507 | 0.51507 | 0.51507 | 0.51507 | 0.53835 | 0.53835 |
| Bunny | 0.64321 | 0.65017 | 0.65017 | 0.64423 | 0.64423 | 0.64415 | 0.64415 | 0.67694 | 0.67694 |
| Horse | 0.62897 | 0.63023 | 0.63023 | 0.62899 | 0.62899 | 0.62897 | 0.62897 | 0.63476 | 0.63476 |
| Golfball | 0.50110 | 0.50155 | 0.50154 | 0.50145 | 0.50145 | 0.50114 | 0.50114 | 0.51531 | 0.50350 |
| Hand | 0.52948 | 0.52949 | 0.52951 | 0.52949 | 0.52951 | 0.52949 | 0.52950 | 0.52949 | 0.52951 |

**Tabell 4:** Optimal radie samt den beräknade radien för varje metod och modell.

| **Modell** | SSE | 2-core | $S$ | 4-core | $S$ |
|---|---|---|---|---|---|
| Triceratops | 0.013 | 0.077 | 0.17 | 0.029 | 0.45 |
| Frog | 0.018 | 0.079 | 0.23 | 0.03 | 0.60 |
| Chair | 0.032 | 0.086 | 0.37 | 0.037 | 0.87 |
| Tiger | 0.138 | 0.139 | 0.99 | 0.06 | 2.29 |
| Bunny | 0.145 | 0.143 | 1.01 | 0.065 | 2.22 |
| Horse | 0.214 | 0.178 | 1.20 | 0.081 | 2.64 |
| Golfball | 0.448 | 0.297 | 1.51 | 0.14 | 3.20 |
| Hand | 1.588 | 0.897 | 1.77 | 0.452 | 3.51 |

**Tabell 5:** Multitrådade beräkningar av 26-DOP på två respektive fyra kärnor. Exekveringstid i ms och uppsnabbning $S$.

1,75 gånger för 2 kärnor och 3,5 gånger för 4 kärnor, vilket motsvara nära 90% av den teoretiskt möjliga uppsnabbningen. I båda fallen inträffar detta vid beräkning av 26-DOP för modellen hand med över 320000 punkter. Tabell 5 visar exekveringstider samt uppsnabbning jämfört med en icke trådparallell, men dataparallell SIMD variant.

Figur 18 visar prestandaförhållandet mellan de olika varianterna av algoritmerna som körs på en, två respektive fyra kärnor. Det syns tydligt hur de multitrådade varianterna behöver ta sig över en tröskel med uppstarts- samt synkroniseringskostnader innan prestandavinster kan göras. [GI05].

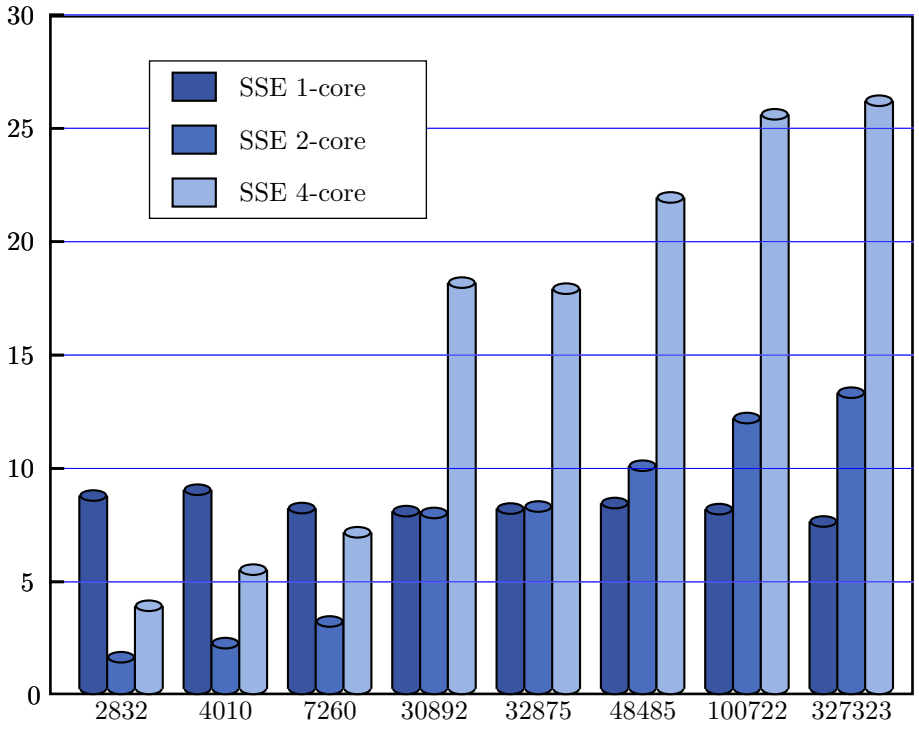
# 6 Slutsats

## 6.1 $k$-DOP

Genom att utnyttja de möjligheter till parallellisering som ges med dagens processorer visas att prestandavinsterna inte behöver stanna vid det förväntade teoretiska, utan kan ge vinster som överstiger detta. Genom att utnyttja de dataparallella SSE-instruktionerna, vilket i teorin kan öka genomströmmningen av data med fyra gånger, visas trots det på uppsnabbningar på det dubbla för beräkningar av extrempunkter på $k$-DOP. Denna stora uppsnabbning beror troligen på att jämförelsesatser helt eleminerats samt att den spatiala lokaliteten hos datat undanröjer cachemissar.

Däremot är det inte helt enkelt att dra slutsatser om orsaken i och med cacheminnets komplexa uppbyggnad i dagens processorer. Bland annat så avviker resultatet rejält i ett avseende. Dataparallella beräkningen av AABB för modellen hand med över 320 000 punkter har en prestandaökning på faktor 4 mot faktor 7-9 för övriga modeller. Vid en närmare analys visade sig en tydlig minskning av prestandan runt 170 000 punkter. Den aktuella processorn har en 2048kB L2 cache tillgänglig (eftersom metoden körs enkeltrådad) vilket motsvarar just det aktuella antalet punkter (2048kB / 4B x 3 = 170 667). Alltså kommer L2 cache att vara konsumerad och ger upphov till cachemissar.

Vad är då anledningen till att just beräkningen av AABB lider av detta och inte 14-DOP eller 26-DOP? En anledning skulle kunna vara att exekveringstiden är för kort för att en förhämtning av data skall hinna ske trots det spatiala läget




**Figur 18:** Diagramet visar prestandaförhållandet mellan de algoritmer som utnyttjar 1, 2 eller 4 kärnor. Axlarna visar antalet punkter(x-axel) samt uppsnabbning mot sekventiell algoritm(y-axel).

i minnet. Övriga metoder skalar helt linjärt över gränsen för L2 cachen vilket tyder på att förhämtningen helt kan överbrygga detta problem.

## 6.2 Sfär

Vidare utredning bör göras då EPOS används vid modeller med mycket litet antal punkter, eftersom en större andel av beräkningen då kommer att ske i den exakta sfärlösaren, Gärtner [Gae99]. Ett tänkbart problemfall kan vara då EPOS används för att skapa sfärträd, där små noder samt lövnoder kommer innehålla mycket få punkter.

## 6.3 Cachebeteende

Metoden som valdes för cachehantering vid testkörningarna gjordes för att skapa förutsättningar för bästa möjliga cachebeteende. Det gjordes förutom detta även tester för att skapa förutsättningarna för sämsta möjliga fallet, vilket då skulle betyda att vid starten av varje körning skulle cachen vara tömd på all intressant data. Detta gjordes för att försöka efterlikna en *cold-start*, de fall då modellernas data används för första gången av en applikation [BH03]. Detta

visade sig problematiskt då en modern processors minneshierarki är komplex samt att inbyggda metoder för förinhämtning (prefetch) kan göra att data ändå redan finns inläst i cachen. Det troliga är dessutom att detta ändå inte skulle efterlikna det verkliga fallet då modellernas data kan ha hanterats i tidigare beräkningssteg och därmed redan finns inläst i minnet, exempelvis vid generering av trädstrukturer.

## 6.4 Multitrådning

Genom att utnyttja trådparallellism visades på möjlighet till stora prestandavinster, speciellt på större modeller samt för mer komplexa metoder. Här finns även delar att ta upp till vidare arbete, som exempelvis EPOS-algoritmen som inte har implementerats med flera trådar. Då endast metoder för att finna $k$-DOPs har gjorts multitrådad, $k$-DOP-SIMD-Threaded, och inte varianten för uthämtning av index saknas i nuläget förutsättningar för en trådad EPOS-algoritm. Dessutom bör även CheckSphere metoden trådas. Här ligger ett större problem då centrumpunkt samt radie kan förändras i vilken tråd som helst. Att arbeta med delade variabler och kritiska sektioner verkar inte lämpligt, med tanke på tidsförlusterna, utan ett alternativt sätt skulle vara att endast beräkna ny radie i varje tråd och sedan sammanställa dessa.

En mera intrikat variant skulle kunna låta en huvudtråd köra CheckSphere med uppdatering av centrum och radie, medan övriga trådar endast söker punkter som befinner sig utanför sfären. Dessa punkter kan sedan åter kontrolleras och uppdateras av huvudtråden. Vissa punkter kommer därmed att kontrolleras flera gånger men samtidigt så behövs ingen synkronisering av delade variabler.

En annan del för vidare undersökning är att de trådparallella metoderna bör göras generella för antalet trådar samt att försöka utröna inverkan av antalet trådar per kärna, då tester visade på bättre resultat med två trådar per processorkärna. Metoden bör även själv kunna avgöra om trådning är lönsamt eller inte i det aktuella fallet.

# Referenser


[AMHH08] Tomas Akenine-Möller, Eric Haines, and Natty Hoffman. *Real-Time Rendering 3rd Edition*. A. K. Peters, Ltd., Natick, MA, USA, 2008.

[BH03] Randal E. Bryant and David R. O'Hallaron. *Computer Systems: A Programmer's Perspective*. Prentice Hall, 2003.

[CJP07] Barbara Chapman, Gabriele Jost, and Ruud van der Pas. *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation)*. The MIT Press, 2007.

[CR99] A. Crosnier and Jarek Rossignac. Tribox bounds for three-dimensional objects. *Computers & Graphics*, 23(3):429–437, 1999.

[Eri04] Christer Ericson. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[Gae99] Bernd Gaertner. Fast and robust smallest enclosing balls. In *ESA'99: Proceedings of the 7th Annual European Symposium on Algorithms*, pages 325–338, London, UK, 1999. Springer-Verlag.

[GBST06] Richard Gerber, Aart J. C. Bik, Kevin B. Smith, and Xinmin Tian. *The Software Optimization Cookbook, 2nd Edition*. Intel Press, 2006.

[GI05] Kang Su Gatlin and Pete Isensee. Reap the benefits of multithreading without all the work. *MSDN Magazine*, October 2005.

[Gol90] Roland Goldman. Intersection of three planes. In A. Glassner, editor, *Graphics Gems*, page 305. Academic Press, 1990.

[Gra72] R.L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. In *Information Processing Letters*, 1, pages 132–133, 1972.

[HOM08] M. Hassaballah, Saleh Omran, and Youssef B. Mahdy. A review of SIMD Multimedia Extensions and their usage in scientific and engineering applications. *Comput. J.*, 51(6):630–649, 2008.

[Int99] Intel®Corporation. *Using Streaming SIMD Extensions to Find the Maximum/Minimum Element of a Single-Precision Floating-point Vector and its Corresponding Index*, 1.2 edition, Jan 1999.

[Int09] Intel®Corporation. *Intel® 64 and IA-32 Architectures Optimization Reference Manual*, March 2009.

[KPT00] Bob Kuhn, Paul Petersen, and Eamonn O'Toole. OpenMP versus threading in C/C++. *Concurrency - Practice and Experience*, 12(12):1165–1176, 2000.

[LAML07] Thomas Larsson, Tomas Akenine-Möller, and Eric Lengyel. On faster sphere-box overlap testing. *journal of graphics tools*, 12(1):3–8, 2007.

[Lar08] Thomas Larsson. Fast and tight fitting bounding spheres. In *Proceedings of The Annual SIGRAD Conference*, pages 27–30. Linköping University Electronic Press, November 2008.

[Rit90] J. Ritter. An efficient bounding sphere. In A. Glassner, editor, *Graphics Gems*, pages 301–303. Academic Press, 1990.

[Str03] Gilbert Strang. *Introduction to Linear Algebra, Third Edition*. Wellesley Cambridge, 2003.

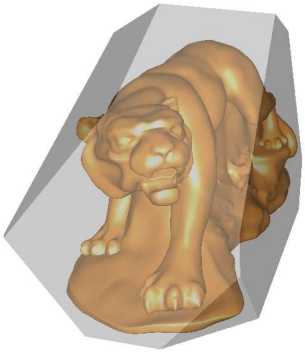
[THCS01] Ronald L. Rivest Thomas H. Cormen, Charles E. Leiserson and Clifford Stein. *Introduction to Algorithms, Second Edition*, chapter 33. Computational Geometry, pages 949–955. MIT Press and McGraw-Hill, 2001.

[WBS07] Ingo Wald, Solomon Boulos, and Peter Shirley. Ray tracing deformable scenes using dynamic bounding volume hierarchies. *ACM Transactions on Graphics*, 26(1), 2007.

[WIP08] Ingo Wald, Thiago Ize, and Steven G. Parker. Fast, parallel, and asynchronous construction of BVHs for ray tracing animated scenes. *Computers & Graphics*, 32(1):3–13, 2008.

# A Visualisering av $k$-DOP

Genom att rendera de beräknade omslutande volymerna kan algoritmerna lättare felsökas samt korrektheten enklare fastställas. Då $k$-DOP-volymer med flera olika antal begränsade plan skulle renderas utvecklades en generell metod. Detta till skillnad från andra metoder som är mer inriktade på att snabbt beräkna begränsade ytor på en specifik volym. Användningsområdet kan vara rendering i realtidsapplikationer, exempelvis som ersättning av modeller med låg detaljnivå (level-of-detail) [CR99].



**Figur 19:** Exempel av renderad volym. Modellen tiger med en 14-DOP.

## A.1 Metod

För att $k$-DOP ska vara effektiva både vid generering samt överlappstest så sparas endast information om respektive plans avstånd från origo. Denna informationen är inte användbar för utritning utan att vidare bearbetas. Pseudokoden i figur 20 visar principen för de olika steg som krävs.

RENDER-$k$-DOP
**input**: $D = \{d_1, d_2, \ldots, d_k\}$, $N = \{\mathbf{n}_1, \mathbf{n}_2, \ldots, \mathbf{n}_k\}$
1. $I_u \leftarrow$ FINDINTERSECTION$(D, N)$
2. $I_o \leftarrow$ SORTPOINTS$(I_u)$
3. RENDER$k$DOP$(I_o)$

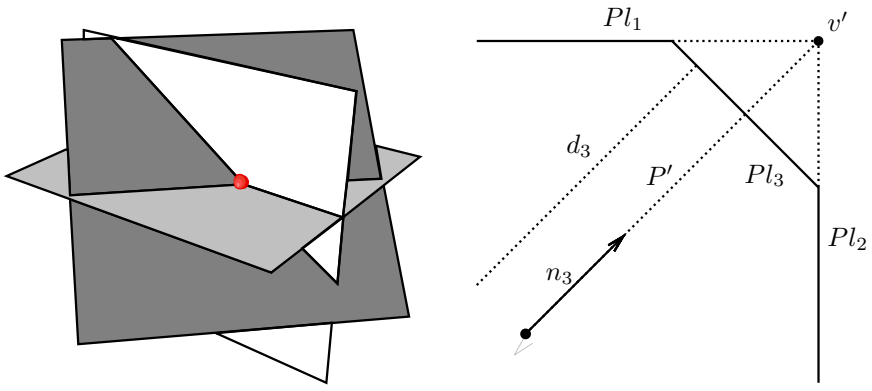
**Figur 20:** Principen för rendering av $k$-DOP

I detta avsnitt skiljer sig representationen av en $k$-DOP mot tidigare. Varje plan beskrivs av en egen normal $\mathbf{n}$ genom att normalerna inverteras och adderas till det ursprungliga normalsetet $N$. Dessutom så läggs alla avstånd till skärningsplan i samma lista $D$. På detta vis har alla plan en enhetlig definition med en normal samt ett avstånd, vilket förenklar vidare beräkningar. Det nya normalsetet $N$ samt de samlade min- och maxavstånden $D$ är indata till funktionen. Deloperationen FINDINTERSECTION returnerar alla skärningspunkter som osorterade listor per plan (se avsnitt A.2). SORTPOINTS ser till att punkter-

na sorteras efter ordningen de ska renderas (se avsnitt A.3). Den avslutande RENDER$k$DOP renderar varje plans polygon med de sorterade hörnpunkterna.

## A.2 Skärningspunkter

Varje plan i volymen begränsas av skärningar med övriga plan. Då tre plan skär varandra har vi en potentiell hörnpunkt (se figur 21(a)). Dessa hittas genom att testa varje kombination av tre olika plan och söka dess eventuella skärningspunkt.




(a) Skärningspunkt av 3 ickeparallella plan.

(b) Degenererad punkt 2D

**Figur 21:** Figur (a) visar principen för att söka en skärningspunkt samt (b) metoden för att avgöra punktens korrekthet.

Resultatet av skärningspunktsberäkningen kan vara:

- två plan är parallella och ingen skärningspunkt existerar.
- punkten är 'falsk' och befinner sig inte på volymens yta (se figur 21(b)).
- en korrekt punkt på volymens yta.
- punkten finns redan då en annan kombination av tre andra plan delar samma skärningspunkt.

I figur 22 visas detaljerna av FINDINTERSECTIONS, som ger utdata i form av en lista med alla skärningspunkter sorterat per plan. Normaler i $N$ normaliseras (rad 2) innan varje permutation av tre plan itereras, beskrivna av normalerna $\mathbf{n}_i, \mathbf{n}_j$ och $\mathbf{n}_l$ (rad 3-5).

Första kontrollen är om planen saknar skärningspunkt, det vill säga att två plan är parallella. Genom att beräkna determinanten för planens tre normaler och kontrollera om denna är lika med noll, eller ännu enklare, beräkna den skalära trippelprodukten (rad 6) [Str03]. Skalär trippelprodukt ges som skalärprodukten av den ena vektorn med kryssprodukten av de två andra, $T = a \cdot (b \times c)$. Trippelprodukten är noll då $b \times c = 0$ (planen parallella) eller om $a$ är en linjärkombination av $b$ och $c$ (rad 7). Dessutom används trippelprodukt (eller

determinant) vid framtagning av skärningspunkten. Om inga plan i permutationen är parallella (rad 7) så beräknas skärningspunkten (rad 8). Denna ges genom [Gol90]:

$$v' = v_1 \cdot \mathbf{n}_1(\mathbf{n}_2 \times \mathbf{n}_3) + v_2 \cdot \mathbf{n}_2(\mathbf{n}_3 \times \mathbf{n}_1) + v_3 \cdot \mathbf{n}_3(\mathbf{n}_1 \times \mathbf{n}_2)/Det(\mathbf{n}_1, \mathbf{n}_2, \mathbf{n}_3)$$

där $v_k$ är en punkt på aktuellt plan som defineras av normalen $\mathbf{n}_k$. Då $v_k \cdot \mathbf{n}_k$ ger avstånd till aktuellt plan kan detta förenklas till $d_k$(se indata RENDER-$k$-DOP), samt att determinaten i nämnaren ersätts med trippelprodukten ger:

$$v' = d_1(\mathbf{n}_2 \times \mathbf{n}_3) + d_2(\mathbf{n}_3 \times \mathbf{n}_1) + d_3(\mathbf{n}_1 \times \mathbf{n}_2)/\mathbf{n}_1 \cdot (\mathbf{n}_2 \times \mathbf{n}_3)$$

Då en skärningspunkt är funnen kontrolleras om punkten är giltig, det vill säga om den befinner sig på volymens yta (VALIDPOINT rad 9), samt om den är ny (NEWPOINT rad 10).

FINDINTERSECTIONS
**input**: $D = \{d_1, d_2, \ldots, d_k\}$, $N = \{\mathbf{n}_1, \mathbf{n}_2, \ldots, \mathbf{n}_k\}$
**output**: $I = \{i_1, i_2, \ldots, i_m\}$

```
1.  for each n_i ∈ N
2.      n_i ← normalize(n_i)
3.  for each n_i ∈ N
4.      for each n_j ∈ N
5.          for each n_l ∈ N
6.              T ← TRIPPLEPRODUCT(n_i, n_j, n_l)
7.              if T ≠ 0
8.                  v' ← INTERSECTIONPOINT(n_i, n_j, n_l, d_i, d_j, d_l)
9.                  if VALIDPOINT(v', D, N)
10.                     if NEWPOINT(v', I)
11.                         I ← ADDPOINT2PLANE(v', n_i, n_j, n_l)
```

**Figur 22:** Pseudokod för metoden FINDINTERSECTIONS som returnerar alla skärningspunkter i $k$-DOP.

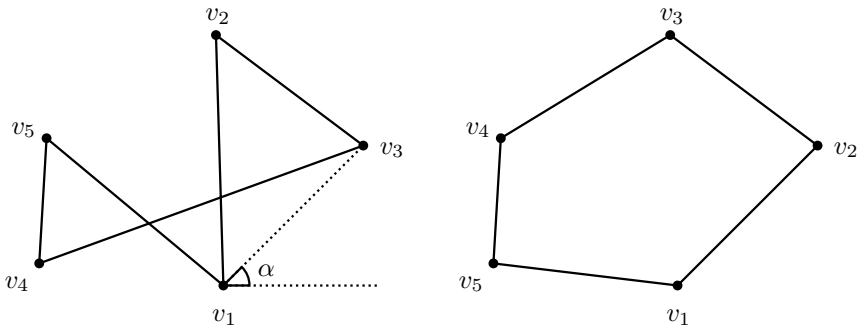
Problemet med kopior av punkter uppträder då fler än tre plan delar samma skärningspunkt. Samma punkt kommer att vara resultatet vid varje permutation av tre plan som delar denna skärningspunkt. Ett ytterliggare dilemma är att flyttalsfel ger upphov till punkter med små varianser, som egentligen representer en existerande punkt. Genom att styra antalet värdesiffror vid kontroll kan detta problem hanteras.

VALIDPOINT anger om punkten är 'falsk', eller degenererad. Dessa punkter uppkommer då förlängningen av tre plan skär varandra utanför volymen, se 2D exempel i figur 21(b).

Genom att projicera den tänkta skärningspunkten $v'$ på varje normal $\mathbf{n} \in N$ samt kontrollera dessa mot respektive avstånd $d \in D$ kan de genererade punkterna sorteras ut. Slutligen adderas de nya, icke degererade punkterna, till listan för respektive plan i den aktuell permutation(rad 11).

## A.3 Sortering av punkter

Punktlistorna $I_u$ som ges ur FindInterSection kommer att vara osorterade, vilket innebär att en rendering av punkterna troligen inte kommer resultera i det konvexa höljet av planet, utan snarare någon slumpmässig kombination av de framtagna punkterna (se figur 23(a)).




(a) Efter FindIntersection (b) Efter SortPoints

**Figur 23:** Det konvexa höljet. Punkterna i ett plan före och efter funktionen SortPoints. (a) visar ett tänkbart scenario med de osorterade punkterna. (b) visar de sorterade punkterna som representerar det konvexa höljet

Pseudokoden i figur 24 visar operationerna i SortPoints där varje plan i volymen itereras igenom (rad 1). Varefter punkterna i respektive plan roteras till $x - y$ planet i CreatePlanar (rad 2) för att kunna hantera punkter i 2D. Figur 23(b) visar det konvexa höljet av punkterna och detta söks genom att använda de första stegen av en så kallad Grahamsökning [Gra72] [THCS01]. Detta sker i OrderConvexHull(rad 3). Först söks punkt med lägsta $y$-värde (om flera punkter har samma $y$-värde väljs den punkt därav med högst $x$-värde) och används som ankare för vidare beräkning ($v_1$ i figur 23(a)). Därefter beräknas vinkel $\alpha$ mellan $x$-axeln samt vektor från ankare till varje punkt i planet. Vinkeln kommer att representera den polära utbredningen i planet och ger efter en sortering av listan det konvexa höljet för planet. Då ordningen för punkterna är känd kan planen enkelt renderas med Render$k$DOP.

SortPoints
**input**: $I_u = \{i_1, i_2, \ldots, i_m\}$, $N = \{\mathbf{n}_1, \mathbf{n}_2, \ldots, \mathbf{n}_k\}$
**output**: $I_o = \{i_1, i_2, \ldots, i_m\}$
1. **for each** $\mathbf{n}_i \in$ N
2. $\quad T_{\mathbf{n}_i} \leftarrow$ CreatePlanar($I_{\mathbf{n}_i}$, $\mathbf{n}_i$)
3. $\quad I_o \leftarrow$ OrderConvexHull($T_{\mathbf{n}_i}$)

**Figur 24:** Pseudokod för metoden SortPoints som returnerar skärningspunkter efter det konvexahöljet.